Approximate String Matching

Improving Data Structures and Algorithms

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

Christopher Maximilian Pockrandt

am Fachbereich
Mathematik und Informatik
der Freien Universität Berlin

Berlin 2019

Tag der Disputation: ***11.04.2019***

Gutachter:

Prof. Dr. Knut Reinert, *Freie Universität Berlin*
Prof. Dr. Sven Rahmann, *Universität Duisburg-Essen und Technische Universität Dortmund*

Dedicated to my beloved father († January 7, 2016)

illustrated by Anna Fricke

Abstract

This thesis addresses important algorithms and data structures used in sequence analysis for applications such as read mapping. First, we give an overview on state-of-the-art FM indices and present the latest improvements. In particular, we will introduce a recently published FM index based on a new data structure: EPR dictionaries. This rank data structures allows search steps in constant time for unidirectional and bidirectional FM indices. To our knowledge this is the first and only constant-time implementation of a bidirectional FM index at the time of writing. We show that its running time is not only optimal in theory, but currently also outperforms all available FM index implementations in practice.

Second, we cover approximate string matching in bidirectional indices. To improve the running time and make higher error rates suitable for index-based searches, we introduce an integer linear program for finding optimal search strategies. We show that it is significantly faster than other search strategies in indices and cover additional improvements such as hybrid approaches of index-based searches with in-text verification, i.e., at some point the partially matched string is located and verified directly in the text.

Finally, we present a yet unpublished algorithm for fast computation of the mappability of genomic sequences. Mappability is a measure for the uniqueness of a genome by counting how often each k-mer of the sequence occurs with a certain error threshold in the genome itself. We suggest two applications of mappability with prototype implementations: First, a read mapper incorporating the mappability information to improve the running time when mapping reads that

match highly repetitive regions, and second, we use the mappability information to identify phylogenetic markers in a set of similar strains of the same species by the example of E. coli. Unique regions allow identifying and distinguishing even highly similar strains using unassembled sequencing data.

The findings in this thesis can speed up many applications in bioinformatics as we demonstrate for read mapping and computation of mappability, and give suggestions for further research in this field.

Acknowledgments

I want to thank Knut Reinert for being a better supervisor and advisor as I could have ever wished for. He not only supported me throughout the creation of my thesis, but also was an inexhaustible source of motivation during the past years.

I am grateful to the International Max Planck Research School of Computational Biology and Scientific Computing for funding my work and numerous conferences as well as offering a variety of education and support. I want to express my thanks to our PhD coordinator Kirsten Kelleher to whom I could always reach out, and also my thesis advisory committee for feedback during my research, Martin Vingron and Ralf Herwig.

Special thanks to my colleagues Hannes, Marcel, René, Sara, and Svenja for having the best coding retreats. 

Also, I would like to thank Ron Wenzel for proofreading my thesis and everyone who will read and prevent it from catching dust in the book shelf.

My greatest appreciation goes to Wolfgang Durdel, my former computer science teacher in high school. Without his computer science class I might have ended up in a different subject, but for sure without him drawing my attention to the Freie Universität Berlin I would not have spent the past eight years studying and researching computer science and bioinformatics there. I would have missed out on a lot of contacts, studies abroad and memorable times, and probably the opportunity to write my PhD thesis with Knut Reinert. An unforgettable time at the Freie Universität Berlin goes to an end, but I hope to return one day.

Contents

1 Introduction

1.1 Sequence Analysis

In the field of sequence analysis, searching biological data is an essential step in many bioinformatics applications. These applications such as read mapping or protein search tools, to name a few, rely on searching enormous amounts of data, such as DNA or protein sequences, that have been sequenced at some point and have been accumulated in databases over decades. The main challenges that we come across in this field are due to the advancements in sequencing technologies.

Since sequencing costs dramatically decrease as new methods are developed, the amount of accumulated data grows faster and faster. The growth can be observed in large sequence databases such as the Sequence Read Archive (SRA) which is a public repository led by an international collaboration including the National Center for Biotechnology (NCBI). It contains primarily DNA sequencing data from high-throughput sequencing experiments. High-throughput sequencing (next-generation sequencing, NGS) classifies multiple sequencing technologies that were developed for low-cost and highly parallelizable sequencing.

Figure 1.1 compares the data growth of the SRA to Moore's law, an observation that the number of transistors in integrated circuits doubles every 18 to 24 months which comes along with a similar performance increase of CPUs [Moore, 1965]. It indicates that advancements in computer architecture are not sufficient to process the ever growing databases.

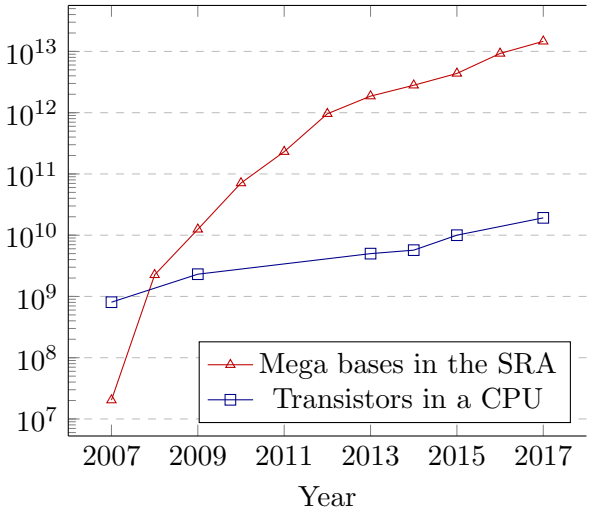


Figure 1.1: The number of mega bases sequenced in the SRA compared to the transistor count on a logarithmic scale.[1]

One of the major applications in bioinformatics is read mapping. When a genome is sequenced, instead of a single sequence many short sequences (so-called reads) are outputted by the sequencing machine. Depending on the sequencing technology common read lengths for high-throughput sequencing range from 100 to 250 base pairs such as for Illumina HiSeq experiments. Multiple sequencing runs are performed. In each run the reads are likely to begin at slightly different positions in the genome, i.e., the reads of different sequencing runs overlap and can be reassembled to retrieve the original genome. There are two kind of assemblies: *de-novo* and *mapping* assembly. While de-novo assembly is typically used if the genome is sequenced for the first time and no reference genome exists, mapping assembly requires a reference genome and maps each read to it [Mäkinen, Veli and Belazzougui, Djamal and Cunial, Fabio and Tomescu, Alexan-

dru I, 2015]. The mapping locations are then used to reconstruct the genome.

Going a step further to the field of metagenomics, reads are not searched in a single reference genome, but in a large database of different genomes. This is done when sequencing an unknown sample to determine the species present in the sample. Similar applications are protein search tools. Samples can be sequenced for RNA transcripts that will later be translated to proteins. These transcripts are searched in transcript or protein databases such as UniProtKB [Apweiler et al., 2004] to analyze the transcriptome of cells.

All these applications that map sequences to a reference sequence or search them in a large database of sequences share common characteristics: the databases searched are rather static. Reference genomes are seldom updated, databases such as UniProt are currently updated every 4 weeks[2]. Due to the large throughput of next-generation sequencing machines (e.g., 160 GB per day, HiSeq 2500 [Reuter et al., 2015]), an enormous amount of reads is produced. Since each read is searched in the reference sequence or database, the data is indexed prior to searching. This significantly improves the running time of the search.

Another challenge that is addressed in this thesis is the demand for higher accuracy in many applications. Whenever sequences are searched in a reference data set, a certain amount of errors has to be taken into account. Due to sequencing errors [Yang et al., 2012] or genetic variations [Pavlopoulos et al., 2013], reads from the sample might not occur exactly in the reference data set. The most common form of genetic variations are single nucleotide polymorphisms

[1]The transistor count data was taken from `https://ourworldindata.org/technological-progress` and the SRA statistics from `https://trace.ncbi.nlm.nih.gov/Traces/sra/sra.cgi`, accessed on January 3rd, 2019

[2]https://www.uniprot.org/help/synchronization, accessed on December 11th, 2018

(SNPs), i.e., a single base mutates. It can either be replaced by a different base (i.e., a substitution) or a base is inserted respectively deleted. The same kind of error types can occur during the sequencing process, while the error profile varies depending on the sequencing technology. Illumina sequencing for example produces mostly substitution errors.

To account for these errors, *approximate* matches are also considered, i.e., matches that are identical except up to a certain percentage of substitutions, insertions, or deletions. Searching and allowing for errors, which is referred to as approximate string matching, is expensive for larger number of errors. Thus, most read mappers and search tools only have a feasible running time for a small number of errors.

1.2 Outline

This thesis will cover two crucial steps in the aforementioned applications: indexing of biological sequences for efficient searching and approximate string matching algorithms in indices.

In chapter 2 we introduce one of the most common state-of-the-art string indices. The FM index is used by many notable read mapping tools such as Bowtie 2 [Langmead and Salzberg, 2012b], BWA [Li and Durbin, 2009a], and search tools for nucleotide sequences or proteins such as LAST [Kielbasa et al., 2011] or Lambda [Hauswedell et al., 2014]. We also present improved unidirectional and bidirectional FM indices that are significantly faster, both in theory and in practice. Their open-source implementations are currently the fastest available [Pockrandt et al., 2017]. Due to the rapidly growing sequence databases and the faster sequencing technologies, not only more data is being sequenced that has to be mapped or searched, but also larger databases increase the time for searching. Using faster indexing data structures such as an improved FM index, the algorithmic side can keep pace with the technological advancements in sequencing.

In chapter 3 we introduce algorithmic approaches for approximate string matching in indices. We present an improved framework and an integer linear program to compute optimal search strategies in bidirectional indices leading to a faster string matching algorithm [Kianfar et al., 2018] that proves to be faster in practice than other approaches. This allows for even higher error rates for many bioinformatics applications.

An example for applications with higher error rates is presented in section 3.9. For CRISPR/Cas9 experiments we searched for off-targets in the human genome for given guideRNA sequences. These guideRNA are short RNA sequences that are designed to bind to specific locations in the genome that – hopefully – will not bind anywhere else. These locations of other potential matches are called off-targets. Since RNA-binding can also be prone to errors, we have to perform approximate string matching when searching for off-targets. As part of the pipeline, guideRNA of length 23 were searched in the genome with up to 8 errors which corresponds to an error rate of more than 33%, which many read mapping tools are not suited for.

In chapter 4 we present some of our applications that make use of the two presented improvements in the field of sequence analysis: faster string indices and faster approximate matching in indices. We will cover the concept of mappability, i.e., determining which regions of a genome or a set of genomes are unique and which are repetitive. We will give a faster algorithm for the computation of the mappability and suggest new applications of mappability. First, we introduce an improved read mapper incorporating mappability information into the mapping process. Second, we use mappability information on a set of similar strains of the same species to identify marker genes. These are short unique sequences in the data set that allow determining the specific strain using only its unassembled sequencing data.

2 Indexing Data Structures

2.1 Notation

First, we introduce some notation that will be used throughout this thesis.

Definition 2.1.1 (Strings).
We call T a string of length $|T| = n$ over the alphabet Σ, i.e., $T \in \Sigma^n$. Strings are indexed from 1 to n, $T[i]$ represents the ith character with $1 \leq i \leq n$.

In most programming languages strings and arrays are indexed from *zero*. Since this makes the notation and formulas more complex, all strings and arrays will be indexed beginning from *one*, if not stated otherwise.

Definition 2.1.2 (Infixes, Suffixes and Prefixes).
An infix of T is a substring written as $T[i..j]$ with $1 \leq i \leq n+1$ and $1 \leq j \leq n$. If $i > j$, the infix is empty, e.g., $T[1..0] = \varepsilon$. Suffixes $T[i..n]$ with $1 \leq i \leq n+1$ can be denoted as $T[i..]$, a prefix $T[1..i]$ with $0 \leq i \leq n$ as $T[..i]$.

Definition 2.1.3 (String concatenation).
Two strings can be concatenated using the concatenation operator $\cdot : \Sigma^m \times \Sigma^n \to \Sigma^{m+n}$. Characters are considered strings of length 1. Let T_1 and T_2 be two strings of length m and n:

$$(T_1 \cdot T_2)[i] = \begin{cases} T_1[i] & , \quad i \leq m \\ T_2[i-m] & , \quad \text{otherwise} \end{cases}$$

If not stated otherwise, all logarithms in this work are to base 2.

2.2 Overview

In bioinformatics applications such as read mapping many short sequences are searched in a text. Both, the query sequences and the text are represented as strings over the same ordered, finite alphabet $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$. $< \subseteq \Sigma \times \Sigma$ is a total order with $c_1 < c_2 < \cdots < c_\sigma$. Since the text T is in general much longer than each query sequence and mostly static, i.e., does not change often over time, T is preprocessed before the search and a so-called string index is built to speed up the searches. A query sequence (also referred to as the pattern) is denoted as P.

There are multiple string indices that mainly differ in space consumption and time complexity to search them. Before we introduce some of these indices we define some common terms.

Definition 2.2.1 (Full-text index).
A full-text index, sometimes also referred to as a (sub-)string index, is a data structure that allows searching substrings in the text in sublinear time of the text's length.

Definition 2.2.2 (Searching and locating).
When introducing full-text indices, we will look at two operations separately: *searching* a query sequence and *locating* it. *Searching* determines whether it occurs in the text and if so, to count the number of occurrences. *Locating* retrieves the starting positions of the searched query sequence in the text. *Locating* requires *searching* the query sequence in advance. Since these two operations require different data structures and operations on these, we will cover them separately.

Definition 2.2.3 (Forward and backward searches).
In most indices query sequences are searched character by character, either starting from the left or from the right. Searching a pattern P of length m starting from the left and extending the prefix character by character to the right is referred to as forward searches. A search step is denoted as $P[..j] \rightarrow P[..j+1]$ for $0 \leq j < m$. Searching a pattern from right to left by extending the suffix, is referred to as backward searches. A backward search step is denoted as $P[i..] \rightarrow P[i-1..]$ for $1 < i \leq n+1$.

Definition 2.2.4 (Unidirectional index).
Unidirectional indices support either only forward searches or backward searches.

Definition 2.2.5 (Bidirectional index).
Bidirectional indices support both forward and backward searches. A query sequence can be extended by a character to the left or to the right, regardless of the direction of the previous character extension. Hence, one can start the search at any position in the pattern and extend the infix of P to the left or to the right in any arbitrary order.

We will only consider full-text indices that allow to search substrings of arbitrary length. Some indices are limited in the length of the substring, such as k-mer indices, which are also referred to as n-gram indices. The idea is to speed up the search by storing a precomputed dictionary containing the text positions of each Σ^k (or just the number of occurrences). As the applications are more limited due to the fixed length k of substrings, we will not further consider these kind of indices.

For building an index, we need to define a sentinel character and the lexicographical order on strings over Σ.

Definition 2.2.6 (Sentinel character).
For comparing strings of different length and to define a unique ordering of cyclic rotated strings, we introduce a sentinel character, denoted as $\$ \notin \Sigma$. It is defined to be smaller than any character in Σ. It will be appended to the end of a string and cannot occur anywhere else in the string. For simplicity of notation, we assume when introducing a string T of length n over Σ that it already has the sentinel character appended, i.e., $T[n] = \$$.

Definition 2.2.7 (Lexicographical order).
The lexicographical order of two strings X and Y of equal length with $X = x_0x_1 \dots x_k$ and $Y = y_0y_1 \dots y_k$ is defined such that

$$X <_{lex} Y \text{ iff. } \exists\, i\, \forall i' < i : x'_i = y'_i \wedge x_i < y_i \ .$$

Strings of different lengths are compared by padding the shorter string with sentinel characters. This definition corresponds to alphabetical sortings generally used in conventional dictionaries. $\leq_{lex}$ is defined analogously. Additionally, we define the comparison operator $<_{lex,m}$ (and $\leq_{lex,m}$ analogously) that compares only the first m characters, i.e.,

$$X <_{lex,m} Y \text{ iff. } X[1..\min\{|X|, m\}] <_{lex} Y[1..\min\{|Y|, m\}] \ .$$

2.3 Suffix Arrays

2.3.1 Definition

Before we take a look at the state-of-the-art indices for many bioinformatics applications, we introduce suffix arrays which are fundamental for understanding FM indices and give a short introduction into building and searching these indices.

Definition 2.3.1 (Suffix array).
Given a string T of length n over the alphabet Σ, the suffix array $SA[1..n]$ is an integer array indicating the starting positions of each suffix of T in lexicographical order.

To put it another way, $T[SA[i]..]$ is the ith lexicographically smallest suffix of T for every $1 \leq i \leq n$. Figure 2.1 shows the suffix array of the text $T = mississippi\$$ with the corresponding suffixes.

i	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	m	i	s	s	i	s	s	i	p	p	i	$
SA[i]	12	11	8	5	2	1	10	9	7	4	6	3

$ i$ ippi$ issippi$ ississippi$ mississippi$ pi$ ppi$ sippi$ sissippi$ ssippi$ ssissippi$

Figure 2.1: Suffix array of the text $T = mississippi\$$.

Since the largest value in SA is n, we need $\lceil \log n \rceil$ bits per value leading to a space consumption of $n \cdot \lceil \log n \rceil \in \Theta(n \log n)$ bits when storing it in a bitcompressed array.

2.3.2 Construction

The suffix array can be trivially constructed by sorting the range of integers $[1, n]$ such that $i < j$ if and only if $T[i..] <_{lex} T[j..]$. Using a comparison-based sorting algorithm such as Merge Sort [Knuth, 1997] with $\Theta(n \log n)$ comparisons, this approach yields an overall worst-case running time of $\Theta(n^2 \log n)$.

Proof. A comparison of two suffixes takes $\mathcal{O}(n)$ time leading to a worst-case running time of $\mathcal{O}(n^2 \log n)$. We show that the worst-case bound is tight. Let $T = aa \dots a\$$ and n even (to avoid rounding). We only consider the $\frac{n}{2}$ suffixes starting in the first half of T. Since all suffixes have a length larger than $\frac{n}{2}$, each comparison takes more than $\frac{n}{2}$ steps. This results in more than $\frac{n}{2} \cdot \frac{n}{2} \log\left(\frac{n}{2}\right) \in \Omega(n^2 \log n)$ steps for sorting the first half of suffixes. □

This trivial approach is not recommended in practice. Especially for strings such as whole genomes containing repetitive subsequences, comparison-based algorithms can be disadvantageous. There are much more advanced algorithms such as the Skew algorithm which also achieve linear running time for the construction of the suffix array [Kärkkäinen and Sanders, 2003].

2.3.3 Search

Once the suffix array is constructed, a query sequence or pattern P of length m can be searched in T using the suffix array. The general idea is to find all suffixes whose prefixes are P. The positions of these suffixes in the text are also the locations of P in T. Since the positions of suffixes are sorted in lexicographical order of the suffixes, the indices, whose corresponding prefix is P, form a continuous block in the suffix array. Thus, the occurrences of P in T can be represented as an interval $[a, b]$ where a represents the first and b the last position in the suffix array whose values are the starting positions of P in T.

The positions of all occurrences in T can be written as $\{SA[i] \mid i \in [a, b]\}$. Equation 2.1 shows the mathematical computation of $[a, b]$. If P does not occur in T the suffix array range is empty, i.e., $a > b$.

$$\begin{aligned} a &= min(\{i \mid P \leq_{lex,m} T[SA[i]..]\} \cup \{n+1\}) \\ b &= max(\{i \mid T[SA[i]..] \leq_{lex,m} P\} \cup \{0\}) \end{aligned} \tag{2.1}$$

Example 2.3.1.
The occurrences of $P = iss$ in $T = mississippi\$$ are represented by the suffix array range $[4, 5]$. The text positions can be retrieved from the suffix array: $SA[4] = 4$ and $SA[5] = 1$ (see figure 2.1).

Computing such a suffix array range can simply be performed by two binary searches on the suffix array to find the first and the last suffix whose prefix is P. This leads to a running time of $\mathcal{O}(m \log n)$, since in the worst case the comparison of P with a suffix takes $\mathcal{O}(m)$ in each step of the binary search. Algorithm 1 gives a trivial search algorithm computing the left and right suffix array bound $[a, b]$ with binary searches [Gröpl and Reinert, 2013].

There are multiple improvements to this search algorithm. By using enhanced suffix arrays with additional data structures faster running times of $\mathcal{O}(m + \log n)$ [Manber and Myers, 1993] or even $\mathcal{O}(m)$ [Abouelhoda et al., 2004] can be achieved. These running times only reflect search queries, i.e., determining whether P occurs in T or to count the number of occurrences. Locating the occurrences by a lookup in the suffix array takes additional time which is linear in the number of occurrences.

Algorithm 1 Computing the suffix array bounds for a pattern

```
1:  procedure SEARCH(P[1..m], T[1..n], SA[1..n])
2:      if P ≤_{lex,m} T[SA[1]..] then                    ▷ Left bound
3:          a ← 1
4:      else if P >_{lex,m} T[SA[n]..] then
5:          a ← n + 1
6:      else
7:          (ℓ, r) ← (1, n)
8:          while r − ℓ > 1 do
9:              m ← ⌈(ℓ+r)/2⌉
10:             if P ≤_{lex,m} T[SA[m]..] then
11:                 r ← m
12:             else
13:                 ℓ ← m
14:         a ← r
15:     if P ≥_{lex,m} T[SA[n]..] then                    ▷ Right bound
16:         b ← n
17:     else if P <_{lex,m} T[SA[1]..] then
18:         b ← 0
19:     else
20:         (ℓ, r) ← (1, n)
21:         while r − ℓ > 1 do
22:             m ← ⌈(ℓ+r)/2⌉
23:             if P ≥_{lex,m} T[SA[m]..] then
24:                 ℓ ← m
25:             else
26:                 r ← m
27:         b ← ℓ
28:     return (a, b)
```

2.4 FM Indices

2.4.1 Definition

The FM index, short for Full-text index in Minute space [Ferragina and Manzini, 2000], is a very fast and space efficient full-text index that allows searching a text similarly to the suffix array. It is based on the Burrows-Wheeler transform (BWT) [Burrows and Wheeler, 1994] and uses significantly less space than suffix array implementations.[1]

Definition 2.4.1 (Burrows-Wheeler transform).
Let $T[1..n]$ be a text and $SA[1..n]$ the corresponding suffix array. The Burrows-Wheeler transform $L[1..n]$ of T is defined as

$$L[i] = \begin{cases} T[SA[i]-1] & , \quad SA[i] > 1 \\ \$ & , \quad \text{otherwise} \end{cases}$$

The Burrows-Wheeler transform, which is a reordering of the characters in T, can be seen and constructed from a different angle. For that reason we define cyclic shifts on strings.

Definition 2.4.2 (Cyclic shift).
Given a string T of length n, the ith cyclic shift or cyclic rotation of T is denoted as $T^{(i)}$ and defined as $T^{(i)} = T[i..n] \cdot T[1..i-1]$, i.e., the text is shifted in a cyclic manner by $i-1$ characters to the left respectively $n-i+1$ to the right.

Conceptually the Burrows-Wheeler transform can be interpreted as the last column in a matrix by writing down all cyclic shifts $T^{(1)}$, $T^{(2)}, \ldots, T^{(n)}$ of the text T and sorting them lexicographically. Since the text has a trailing unique sentinel character \$, the sorted cyclic shifts correspond to the sorted suffixes of T. Figure 2.2 illustrates this concept for the text $T = mississippi\$$.

[1] The structure and notation of this chapter on FM indices is based on the lecture script by David Weese and my master thesis. [Weese, 2013, Pockrandt, 2015]

				F	L
$T^{(1)}$	mississippi$		$T^{(12)}$	$mississippi	i
$T^{(2)}$	ississippi$m		$T^{(11)}$	i$mississip	p
$T^{(3)}$	ssissippi$mi		$T^{(8)}$	ippi$missis	s
$T^{(4)}$	sissippi$mis		$T^{(5)}$	issippi$mis	s
$T^{(5)}$	issippi$miss		$T^{(2)}$	ississippi$m	
$T^{(6)}$	ssippi$missi	$\overset{sort}{\Longrightarrow}$	$T^{(1)}$	mississippi$	
$T^{(7)}$	sippi$missis		$T^{(10)}$	pi$mississi	p
$T^{(8)}$	ippi$mississ		$T^{(9)}$	ppi$mississi	
$T^{(9)}$	ppi$mississi		$T^{(7)}$	sippi$missi	s
$T^{(10)}$	pi$mississip		$T^{(4)}$	sissippi$mi	s
$T^{(11)}$	i$mississipp		$T^{(6)}$	ssippi$miss	i
$T^{(12)}$	$mississippi		$T^{(3)}$	ssissippi$mi	

Figure 2.2: Cyclic shifts and lexicographically sorted cyclic shifts of $T = mississippi\$$. The last column of the right matrix represents the BWT read from top to bottom: $L = ipssm\$pissii$.

We denote $\mathcal{M}$ as the matrix of lexicographically sorted cyclic shifts of T, and $\mathcal{M}_i$ as the ith row of $\mathcal{M}$, i.e., $\mathcal{M}_i = T^{(SA[i])}$. Based on this conceptual view of the BWT, important properties can be observed between the first column, denoted as F and the last column, denoted as L, the Burrows-Wheeler transform.

Definition 2.4.3 (LF mapping).
The LF mapping is a bijective function $LF : [1..n] \rightarrow [1..n]$, mapping the rank of a row in $\mathcal{M}$ to the rank of this row shifted by one character to the right (or $n - 1$ characters to the left).

$$LF(\ell) = f \Leftrightarrow M_f = M_\ell^{(n)}$$

This mapping is crucial for searching T using the Burrows-Wheeler

transform.

Lemma 2.4.1 (Character identity).
Given any f and ℓ with $f = LF(\ell)$, the characters $F[f]$ and $L[\ell]$ are identical and furthermore, correspond to the same position in the text, i.e., $SA[f] = SA[\ell] + n - 1 \pmod{n}$.

Proof. This results directly from the definition of the LF mapping and the matrix $\mathcal{M}$.

$$\begin{array}{lrcl} & LF(\ell) & = & f \\ \iff & M_f & = & M_\ell^{(n)} \\ \iff & T^{(SA[f])} & = & T^{(SA[\ell])^{(n)}} \\ \iff & T^{(SA[f])} & = & T^{(SA[\ell]+n-1)} \\ \iff & SA[f] & \equiv & SA[\ell] + n - 1 \pmod{n} \end{array}$$

□

Example 2.4.1.
Given the example in figure 2.3, let $\ell = 8$ and $f = LF(8) = 3$. Both characters are identical $L[8] = F[3] = i$ and both refer to the same character $T[8]$.

Lemma 2.4.2 (Rank preservation).
For all $i, j \in [1..n]$ with $L[i] = L[j]$: $i < j \Rightarrow LF(i) < LF(j)$.

Proof. From $i < j$ we can conclude that $\mathcal{M}_i <_{lex} \mathcal{M}_j$. Furthermore, from $L[i] = L[j]$ follows that $\mathcal{M}_i[1..n-1] <_{lex} \mathcal{M}_j[1..n-1]$.

$$\begin{array}{lrcl} & L[i] \cdot \mathcal{M}_i[1..n-1] & <_{lex} & L[j] \cdot \mathcal{M}_j[1..n-1] \\ \iff & \mathcal{M}_i^{(n)} & <_{lex} & \mathcal{M}_j^{(n)} \\ \iff & \mathcal{M}_{LF[i]} & <_{lex} & \mathcal{M}_{LF[j]} \\ \iff & LF[i] & < & LF[j] \end{array}$$

□

An example of the rank preservation property is illustrated in figure 2.3. From lemma 2.4.1 and 2.4.2 we can conclude the following observation.

Observation 2.4.1.
The ith occurrence of a character $c \in \Sigma$ in L corresponds to the same position in the text as the ith occurrence of c in F.

Example 2.4.2.
The 2nd $i \in \Sigma$ in F occurs in $\mathcal{M}_3$, the 2nd i in L in $\mathcal{M}_8$ (see figure 2.3). Both refer to the same position in the text, i.e., the 3rd i in $T = mississippi\$$, which is $T[8]$.

Observation 2.4.1 is crucial in the design and implementation of the FM index. It allows us to formulate the LF as a function, that can easily be computed from L:

Definition 2.4.4 (LF mapping tables).
We define two functions on L that can be stored as precomputed values in a table:

- $C : \Sigma \to [0..n]$: $C(c)$ counts the number of occurrences of characters in T that are strictly smaller than c, including the sentinel character \$.
- $Occ : \Sigma \times [1..n] \to [0..n]$: $Occ(c, i)$ counts the number of occurrences of c in the prefix $L[1..i]$.

The LF mapping can then be formulated as

$$LF(\ell) = C(L[\ell]) + Occ(L[\ell], \ell) \ .$$

Theorem 2.4.3.
The definitions of the LF mapping in 2.4.3 and 2.4.4 are equivalent.

Proof. By definition, the characters in F are sorted lexicographically. Thus, $C(L[\ell])$ points to the beginning of the block of $L[\ell]$. $Occ(L[\ell], \ell)$ counts the number of occurrences of $L[\ell]$ in $L[1..\ell]$. From observation 2.4.1 follows that the character in column F in row $C(L[\ell]) + Occ(L[\ell], \ell)$ maps to the same position in the text as the character in column L in row ℓ. □

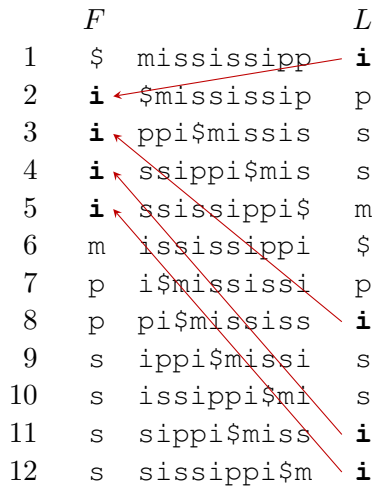


Figure 2.3: Illustration of the LF mapping. The ranks of the characters in F and L are preserved.

In the next two subsections we will take a look at how to search query sequences in the text using C and Occ, both unidirectionally and bidirectionally. For that we will only use the BWT L. Locating the text positions of the occurrences will be covered in section 2.4.5. While C is trivially implemented as an array of size σ consuming $\mathcal{O}(\sigma \cdot \log n)$ bits of space and thus is neglectable especially for small

alphabets, *Occ* requires a lot more space. Possible data structures and implementation details with running time and space consumption analyses are postponed to section 2.4.6. Instead, we will substitute the running time for *Occ* queries as $\mathcal{T}_{Occ}$.

2.4.2 Unidirectional Search

Definition 2.4.5 (Backward search).
Let $[a_i, b_i]$ be the suffix array interval whose suffixes start with the query P. The suffix array interval $[a_{i-1}, b_{i-1}]$ representing the occurrences of $c_j P$ for $c_j \in \Sigma$ can be computed as follows:

$$\begin{aligned} a_{i-1} &= C(c_j) + Occ(c_j, a_i - 1) + 1 \\ b_{i-1} &= C(c_j) + Occ(c_j, b_i) \end{aligned} \tag{2.2}$$

Proof. Given $[a_i, b_i]$, one is interested in all rows within this range whose character in the last column of $\mathcal{M}$ is c_j (as the rows are cyclic rotations of the text). From the lexicographical sorting we can conclude that the range $[a_{i-1}, b_{i-1}]$ will form a continuous block. The size of the new range is the number of occurrences of c_j in $L[a_i..b_i]$. The last open question is how to determine a_{i-1}. Due to the rank preservation property, an LF mapping could be performed on the first row in $[a_i, b_i]$ whose last character is c_j. Since the tables do not allow to determine this efficiently, an LF mapping on the last occurrence of c_j before row $\mathcal{M}_{a_i}$ is performed and the value is increased by one.

$$a_{i-1} = C(c_j) + Occ(c_j, a_i - 1) + 1$$

Having computed the beginning of the range and the size of the new range, determining the end of the range can be simplified by substituting a_{i-1}:

$$b_{i-1} = a_{i-1} + Occ(c_j, b_i) - Occ(c_j, a_i - 1) = C(c_j) + Occ(c_j, b_i)$$

□

When searching a sequence P of length m, one starts with the empty pattern and extends it character by character from right to left. The empty sequence ε is represented by the interval $[a_m, b_m] = [1, n]$. The formula for the first backward search of the rightmost character c_j in P, i.e., $P[m]$ can be simplified to:

$$\begin{aligned} a_m &= C(c_j) + 1 \\ b_m &= C(c_{j+1}) \end{aligned} \tag{2.3}$$

c_{j+1} is the smallest character in Σ that is larger than c_j. If there is no such character, the C value is defined to be n. Note, that using these definitions the first backward search can actually not be computed by equation 2.2 as $Occ(c_j, a_m - 1) = Occ(c_j, 0)$ is not defined. This is only relevant for the first backward search as we do not allow the sentinel character to be part of the pattern P and thus $a_i \neq 1$ except for the very first backward search ($i = m$).

The running time for algorithm 2 is $\mathcal{O}(m \cdot \mathcal{T}_{Occ})$. The running times for retrieving Occ values depend on the implementations which are introduced and compared in section 2.4.6. Figure 2.4 shows the intermediate steps for searching $P = iss$.

$$\begin{array}{lcllcll} a_3 & & & & & = & 1 \\ b_3 & & & & & = & 12 \\ a_2 & = & C(s) & & & = & 9 \\ b_2 & = & C(s+1) & & & = & 12 \\ a_1 & = & C(s) + Occ(s, 9-1) + 1 & = & 8 + 2 + 1 & = & 11 \\ b_1 & = & C(s) + Occ(s, 12) & = & 8 + 4 & = & 12 \\ a_0 & = & C(i) + Occ(i, 11-1) + 1 & = & 1 + 2 + 1 & = & 4 \\ b_0 & = & C(i) + Occ(i, 12) & = & 1 + 4 & = & 5 \end{array} \tag{2.4}$$

	F		L
1	$	mississipp	i
2	i	$mississip	p
3	i	ppi$missis	s
4	i	ssippi$mis	s
5	i	ssissippi$	m
6	m	ississippi	$
7	p	i$mississi	p
8	p	pi$mississ	i
9	s	ippi$missi	s
10	s	issippi$mi	s
11	s	sippi$miss	i
12	s	sissippi$m	i

(a) $P[4..3] = \varepsilon$

	F		L
1	$	mississipp	i
2	i	$mississip	p
3	i	ppi$missis	s
4	i	ssippi$mis	s
5	i	ssissippi$	m
6	m	ississippi	$
7	p	i$mississi	p
8	p	pi$mississ	i
9	**s**	ippi$missi	s
10	**s**	issippi$mi	s
11	**s**	sippi$miss	i
12	**s**	sissippi$m	i

(b) $P[3..3] = s$

	F		L
1	$	mississipp	i
2	i	$mississip	p
3	i	ppi$missis	s
4	i	ssippi$mis	s
5	i	ssissippi$	m
6	m	ississippi	$
7	p	i$mississi	p
8	p	pi$mississ	i
9	s	ippi$missi	s
10	s	issippi$mi	s
11	**s**	**s**ippi$miss	i
12	**s**	**s**issippi$m	i

(c) $P[2..3] = ss$

	F		L
1	$	mississipp	i
2	i	$mississip	p
3	i	ppi$missis	s
4	**i**	**ss**ippi$mis	s
5	**i**	**ss**issippi$	m
6	m	ississippi	$
7	p	i$mississi	p
8	p	pi$mississ	i
9	s	ippi$missi	s
10	s	issippi$mi	s
11	s	sippi$miss	i
12	s	sissippi$m	i

(d) $P[1..3] = iss$

Figure 2.4: Searching for $P = iss$ using backward searches (figure taken from [Pockrandt, 2015]). The computation steps of the ranges are listed in equation 2.4.

Algorithm 2 Searching P in a unidirectional FM index

```
1: procedure SEARCH(P, C, Occ)
2:     a ← 1
3:     b ← n
4:     i ← length(P)
5:     if P ≠ ε then
6:         a ← C(P[i]) + 1
7:         b ← C(P[i] + 1)                    ▷ next larger character
8:     while i > 1 do
9:         i ← i − 1
10:        a ← C(P[i]) + Occ(P[i], a − 1) + 1
11:        b ← C(P[i]) + Occ(P[i], b)
12:        if a > b then
13:            break
14:    return [a, b]
```

2.4.3 Bidirectional Search

While backward searches can be performed efficiently on a unidirectional FM index, a forward search is not suitable to do in a unidirectional FM index without accessing the suffix array. To search patterns from left to right instead from right to left, the text only needs to be reversed upon construction of the FM index. The text positions of the occurrences then have to be translated as the positions are pointing to the last character of the pattern instead of the first. Assuming that the query sequences to be searched are not derived from a stream and thus can be reversed before searching, there is no algorithmic benefit to reverse the text to perform forward searches instead of backward searches.

This is different for bidirectional FM indices, i.e., to allow for both, forward and backward searches, in any arbitrary order. That is, the direction in which the pattern is extended to can be switched between

search steps. There are multiple algorithmic benefits, e.g., running time improvements on approximate string matching which will be covered extensively in chapter 3.

The bidirectional FM index was first introduced by Lam et al. [Lam et al., 2009]. An FM index $\mathcal{I}$ is built on the text T to support backward searches, another FM index $\mathcal{I}^{rev}$ is built on the reversed text T^{rev} to allow for forward searches. This means that the BWT of $\mathcal{I}^{rev}$ is also constructed on the reversed text.

To search a pattern from right to left, one performs regular backward searches on the index $\mathcal{I}$. To search the entire pattern into the opposite direction, one performs backward searches on the index $\mathcal{I}^{rev}$ which correspond to forward searches in $\mathcal{I}$. To switch directions during the search, it is crucial that for each search step performed in one index, the other index is synchronized, i.e., the suffix array range is updated. This allows us for every single search step to use $\mathcal{I}$ for backward searches and $\mathcal{I}^{rev}$ for forward searches.

Consider the scenario in figure 2.5. We have searched the pattern P using backward and/or forward searches. $[a_i, b_i]$ represents the occurrences of P in $\mathcal{I}$ and $[a_i^{rev}, b_i^{rev}]$ the occurrences of P^{rev} in $\mathcal{I}^{rev}$, i.e., $\mathcal{I}$ and $\mathcal{I}^{rev}$ are synchronized.

When P is extended by another character, a backward search is performed, either in $\mathcal{I}$ or $\mathcal{I}^{rev}$. W.l.o.g. we assume that a forward search $P \rightarrow Pc_j$, $c_j \in \Sigma$ is performed, which corresponds to a backward search $P^{rev} \rightarrow c_j P^{rev}$ in $\mathcal{I}^{rev}$. The new suffix array interval $[a_{i-1}^{rev}, b_{i-1}^{rev}]$ is computed as in equation 2.2, i.e., a regular backward search in $\mathcal{I}^{rev}$ is performed. The suffix array interval in $\mathcal{I}$ representing the occurrences of Pc_j can be computed according to lemma 2.4.4.

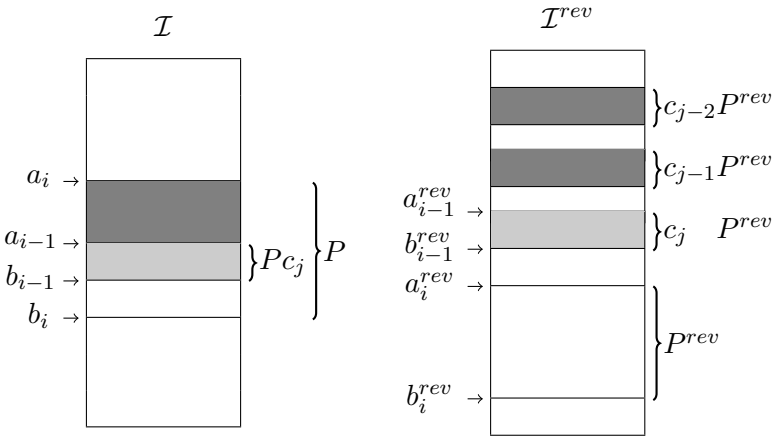


Figure 2.5: Synchronization of a bidirectional FM index [Pockrandt et al., 2017]. First, the backward search $P^{rev} \rightarrow c_j P^{rev}$ is performed and afterwards the interval of Pc_j is computed by determining the interval sizes of every $c_{j'}P^{rev}$ with $j' < j$.

Lemma 2.4.4 (Synchronization in a bidirectional FM index). Let $[a_i, b_i]$ be the suffix array range of P in $\mathcal{I}$ and $[a^{rev}_{i-1}, b^{rev}_{i-1}]$ the suffix array range of $c_j P^{rev}$ in $\mathcal{I}^{rev}$. For the range $[a_{i-1}, b_{i-1}]$ representing the occurrences of Pc_j in $\mathcal{I}$ the following holds:

$$\begin{aligned} a_{i-1} &= a_i + \sum_{\substack{j'<j \\ [a,b] \text{ representing } c_{j'}P^{rev}}} b - a + 1 \\ b_{i-1} &= a_{i-1} + b^{rev}_{i-1} - a^{rev}_{i-1} \end{aligned} \tag{2.5}$$

Proof. First of all, the range $[a_{i-1}, b_{i-1}]$ must be a subrange of $[a_i, b_i]$, since the suffixes whose prefix is Pc_j are also suffixes starting with P. Second, the range must be of the same size as $[a^{rev}_{i-1}, b^{rev}_{i-1}]$, since P occurs in T the same number of times as P^{rev} occurs in T^{rev}. The

interval is then computed by counting the suffixes in $[a_i, b_i]$ that are lexicographically smaller than Pc_j, i.e., that are starting with $Pc_{j'}$ for $c_{j'} < c_j$. These can be computed by performing a separate backward search for each $c_{j'}$ and adding up the number of occurrences. □

Using *Occ* queries the forward search can be computed by equation 2.6. To simplify the formula and lay the foundation of new data structures presented in section 2.4.6, we introduce a new kind of query, *Prefix-Occ*. *Prefix-Occ*(c, i) counts the number of occurrences of all characters lexicographically smaller than c in $L[1..i]$.

$$\begin{aligned} a_{i-1} = a_i + \sum_{j'<j} \Big(& Occ(c_{j'}, b_i^{rev}) && - Occ(c_{j'}, a_i^{rev} - 1) \Big) \\ & a_i + \sum_{j'<j} Occ(c_{j'}, b_i^{rev}) && - \sum_{j'<j} Occ(c_{j'}, a_i^{rev} - 1) \\ & a_i + \textit{Prefix-Occ}(c_{j-1}, b_i^{rev}) && - \textit{Prefix-Occ}(c_{j-1}, a_i^{rev} - 1) \end{aligned} \tag{2.6}$$

Definition 2.4.6 (Prefix-Occ queries).
We introduce a new table *Prefix-Occ* : $\Sigma \times [1..n] \to [0..n]$. It stores the number of occurrences of characters lexicographically smaller or equal to c in the prefix $L[1..i]$, i.e.,

$$\textit{Prefix-Occ}(c, i) = \sum_{c' \leq c} Occ(c', i)$$

2.4.4 Constant Time Rank Support

Before we finally take a look at the core data structure of FM indices, the representation of the BWT to perform *Occ* queries, we introduce an abstract data type and show one of many possible implementations that are later used for supporting efficient *Occ* and *Prefix-Occ* queries.

Definition 2.4.7 (Rank dictionary).
A rank dictionary on a bit vector B of length n supports rank queries $\text{rank}_b(B, i)$ with $b \in \{0, 1\}$ and $1 \leq i \leq n$. Rank queries count the numbers of bits set or not set in the prefix $B[1..i]$. b indicates whether 0s or 1s are counted.

Since we are only interested in counting 1s for our applications, we will omit the notation of b. The computation of $\text{rank}_0(B, i)$ can be reduced to counting 1s, since $\text{rank}_0(B, i) = i - \text{rank}_1(B, i)$.

An efficient way to perform rank queries was proposed by Jacobson [Jacobson, 1988]. An additional data structure can be built on top of a bit vector of length n that requires only $o(n)$ additional space and can perform rank queries in constant time.

The general idea is to precompute rank queries in a two level hierarchy of arrays, L_1 and L_2. The precomputed values $L_1[q]$ are referred to as *superblocks*. They span ℓ^2 bits each of the bit vector and count the bits set in $B[1..(q \cdot \ell^2)]$. $L_2[p]$, which we call *blocks* are of size ℓ and count the number of bits set starting from the overlapping superblock until the end of the pth block, i.e., $B[(1 + k)..(p \cdot \ell)]$ where $k = \lfloor (p - 1)/\ell \rfloor \cdot \ell^2$.

A rank query $\text{rank}(B, i)$ can then be translated into a lookup in both L_1 and L_2. The bits that lie in the last block containing the position i need to be counted separately, which we refer to as an in-block query. This can be done by using a lookup table P containing all possible bit vectors V of length ℓ and all possible positions within a block: $P[V][j] = \text{rank}(V, j)$ for $1 \leq j \leq \ell$.

Lemma 2.4.5.
To compute $\text{rank}(B, i)$ the indices of the previous superblock $q = \lfloor (i - 1)/\ell^2 \rfloor$ and block $p = \lfloor (i - 1)/\ell \rfloor$ are needed.

$$\text{rank}(B, i) = L_1[q] + L_2[p] + P[B[(1 + p\ell)..((p + 1)l)]][i - p\ell] \quad (2.7)$$

Example 2.4.3.
Figure 2.6 shows the precomputed blocks, superblocks and the lookup table P of a given bit vector. The bits set to 1 in the bit vector up to position 15 can be computed by the formula in equation 2.7 as follows ($q = \lfloor (15-1)/3^2 \rfloor = 1$ and $p = \lfloor (15-1)/3 \rfloor = 4$):

$$\begin{aligned}
&\text{rank}(B, 15) \\
&\quad = L_1[1] + L_2[4] + P[B[(1 + 4 \cdot 3)..(4 \cdot 3)]][15 - 4 \cdot 3] \\
&\quad = L_1[1] + L_2[4] + P[B[13..15]][3] \\
&\quad = L_1[1] + L_2[4] + P[010][3] \\
&\quad = 5 + 3 + 1 = 9
\end{aligned}$$

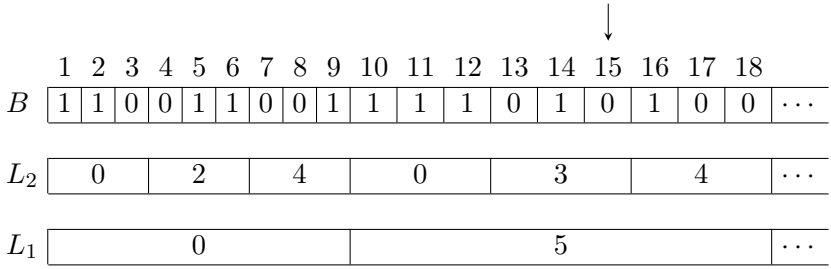
Assuming all arithmetic operations are performed in constant time in the underlying machine model, the running time of a rank query is clearly constant, since only three table lookups are necessary.

Theorem 2.4.6 (Space consumption).
The rank support by Jacobson requires only $o(n)$ additional space for a bit vector of length n.

Proof. For arbitrary ℓ the space consumption is as follows:

- L_1: There are $\frac{n}{\ell^2}$ values in superblocks that each need $\log n$ bits per entry, i.e., $\frac{n}{\ell^2} \cdot \log n$ bits in total.
- L_2: There are $\frac{n}{\ell}$ values in blocks that each need $\log \ell^2$ bits per entry, since they count the bits within a superblock of length ℓ^2, i.e., $\frac{n}{\ell} \cdot \log \ell^2$ bits in total.
- P: There are 2^ℓ possible bit vectors and ℓ positions leading to $\ell \cdot 2^\ell$ entries of $\log \ell$ bits per entry, i.e., $\ell \cdot 2^\ell \cdot \log \ell$ bits in total.

To obtain a sublinear bound, the choice of ℓ is important. This can be achieved by choosing $\ell = \lfloor \log n/2 \rfloor$.



(a) Bit vector B of length 64 with $\ell = \lfloor \log 64/2 \rfloor = 3$ bits per block and $\ell^2 = 9$ bits per superblock.

V	$P[V][1]$	$P[V][2]$	$P[V][3]$
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

(b) Vector $P[\{0,1\}^3][1..\ell]$

Figure 2.6: Additional tables for rank support on B illustrated at an exemplary bit vector.

- L_1: $\mathcal{O}\left(\frac{n}{\ell^2}\log n\right) = \mathcal{O}\left(\frac{n}{\log n}\right) = o(n)$
- L_2: $\mathcal{O}\left(\frac{n}{\ell}\log \ell^2\right) = \mathcal{O}\left(n\frac{\log\log n}{\log n}\right) = o(n)$
- P: $\mathcal{O}\left(\ell \cdot 2^\ell \cdot \log \ell\right) = \mathcal{O}\left(\log n \cdot 2^{\frac{\log n}{2}} \cdot \log\log n\right)$

$$= \mathcal{O}(\log n\sqrt{n}\log\log n) = o(n)$$

□

In practice the lookup table $P[V][j]$ is replaced by a *shift* and a *popcount* operation [González et al., 2005]. Popcount is a hardware CPU operation that works on 64 bit words and counts the number of bits set in a 64 bit integer within a constant number of clock cycles. A popcount operation on modern Intel CPUs has a latency of 3 clock cycles and a throughput of 1 cycle [Granlund, 2017]. Latency is the number of cycles until the data of the instruction is available for the next instruction, throughput is the number of cycles it takes to perform the instruction. Since we only count the bits set before or at the position considered, all bits set behind j are shifted out and the remaining bits set are counted. If we index the bit vector from *zero*, the in-block query can be computed as follows:

$$P[V][j] = popcount(V \gg (j \bmod \ell))$$

2.4.5 Sampled Suffix Array

Even though the search itself is performed entirely without using the suffix array (except for building the Burrows-Wheeler transform), we still require the suffix array. So far we are only able to answer decision queries whether a pattern occurs in the text, and to count the number of occurrences. For locating the occurrences in the text, we need to access the suffix array the same way as we did in section 2.3.3. Since the motivation for the FM index is to reduce the space consumption by avoiding to keep the entire suffix array, only a part of the suffix array is stored.

If only a subset of suffix array values are stored, the missing values can be computed upon request using the LF mapping. Let $SA[i]$ be the position of any suffix in T. Then $SA[LF(i)]$ represents the suffix in T starting one character to the left of $SA[i]$ and $SA[LF(LF(i))]$ starting two characters to the left. With this approach missing SA values can be computed from existing SA values.

Lemma 2.4.7.
For any $1 \leq i \leq n$ and $k \in \mathbb{N}$ the following equation holds:

$$SA[i] = ((SA[\underbrace{LF(\dots(LF(i))\dots)}_{k \text{ times}}] + k - 1) \bmod n) + 1$$

Note that for some i the LF mapping $LF(i) = 1$. For $T = mississippi\$$ this is true for $i = 5$, see figure 2.3. It follows, that the ith smallest suffix is $T[SA[i]..] = T$. The LF mapping cannot retrieve the suffix starting one position further to the left, since the ith suffix is already the longest suffix of T. Due to the cyclic rotations while constructing the BWT, it will instead retrieve the last suffix in T, which is \$. This is taken into account by using modulo.

A single SA value is sufficient to recompute any other SA value using the LF mapping. To reduce the space consumption only a few suffix array values are kept and the missing ones are computed upon request using the LF mapping. There are different approaches on how to sample the suffix array values. The two main strategies are *suffix order sampling* and *text order sampling*. Let $\eta \in \mathbb{N}$. Both have in common that only η^{-1} elements from the suffix array are stored in the sampled suffix array SSA. The two approaches differ in the strategy which values to sample, leading to different worst-case running times for locating an occurrence (i.e., the maximum number of LF mappings necessary to encounter a sampled suffix array value) and partly additional space requirements to identify the positions of sampled values.

2.4.5.1 Suffix Order Sampling

The simplest approach to sample the suffix array is to keep every ηth SA value, e.g., $SA[i]$ is stored if and only if $i \bmod \eta = 1$. While this can be implemented straightforward as in algorithm 3, the running time to compute a missing SA value is $\mathcal{O}(\frac{\eta-1}{\eta} \cdot n \cdot \mathcal{T}_{Occ})$, the space consumption for SSA is reduced from $\mathcal{O}(n \log n)$ to $\mathcal{O}(\frac{n}{\eta} \log n)$ bits.

Algorithm 3 Locating occurrences in a SSA (suffix order sampling)

```
1: procedure LOCATE(i, SSA)
2:     steps ← 0
3:     while i mod η ≠ 1 do
4:         i ← LF(i)
5:         steps ← steps + 1
6:     return ((SSA[i div η] + steps − 1) mod n) + 1
```

Lemma 2.4.8.
The worst-case running time of locating an occurrence using suffix order sampling is $\mathcal{O}(\frac{\eta-1}{\eta} \cdot n \cdot \mathcal{T}_{Occ})$.

Proof. Figure 2.7 illustrates the worst-case scenario. Consider η to be fixed (W.l.o.g. we assume $\eta = 3$) and set the text $T = A^kCA^kCA^kC\$$ (i.e., η repetitions of the string A^kC) on the binary alphabet $\Sigma = \{A, C\}$ for arbitrary $k \in \mathbb{N}$. All suffixes $SA[i]$ with $i \bmod \eta = 1$ are sampled, $\lceil n/\eta \rceil$ suffixes in total. These suffixes are $T[SA[1]..] = T[n..] = \$$ and $T[SA[1 + i * \eta]..] = T[(1 + i)..] = A^{k-i}C \cdot (A^kC)^{\eta-1}$ for $0 \leq i < \lceil n/\eta \rceil$. Thus retrieving the suffix array value storing the text position of $C\$ = T[(n-1)..] = T[SA[n - \eta + 1]..]$ will take the most LF mappings to reach the first sampled suffix array value, which is the one storing the suffix $T[\lceil \frac{n}{\eta} \rceil - 1] = CA^kCA^kC\$$, i.e., in total $(n-1) - (\lceil \frac{n}{\eta} \rceil - 1) \in \mathcal{O}(\frac{\eta-1}{\eta}n)$ LF mappings.

This proof works with any sampling condition $i \bmod \eta = x$ for $0 \leq x < \eta$, since by construction the suffixes whose values are sampled always lie in a continuous block in the text. Thus, the worst-case running time is achieved by simply retrieving the suffix array value whose position in the text is one character left from the sampled block or the last position in the text if the sampled block is starting at the beginning of the text. □

i	1	2	3	4	5	6	7	8	9	10	11	12	13
T[i]	A	A	A	C	A	A	A	C	A	A	A	C	$
SA[i]	(13)	9	5	(1)	10	6	(2)	11	7	(3)	12	8	(4)

i	1	2	3	4	5
SSA[i]	13	1	2	3	4

Figure 2.7: Worst-case running time example for suffix order sampling. $T = AAACAAACAAAC\$$ with $\eta = 3$. Sampled suffix array values are encircled. To retrieve the value $SA[11]$ a total of 8 LF mappings are necessary: $SA[11] = SA[8] + 1 = SA[5] + 2 = \cdots = SA[13] + 8 = SSA\left[\left\lfloor \frac{13}{3} \right\rfloor\right] = 3 + 8 = 11$.

2.4.5.2 Text Order Sampling

A better worst-case performance can be achieved by sampling the SA values that fulfill $SA[i] \bmod \eta = 1$. This guarantees a running time of $\mathcal{O}(\eta \cdot \mathcal{T}_{Occ})$, since every ηth position in the text is sampled in SSA and it takes at most $\eta - 1$ LF mappings to encounter a sampled suffix array value. The downside is that during the location we cannot evaluate whether the ith smallest suffix has been sampled or not, since we do not know the value of $SA[i] \bmod \eta$. Thus, a bit vector of length n is necessary for indicating the sampled positions, i.e., $B[i] = 1$ if and only if $SA[i] \bmod \eta = 1$. Furthermore, we need rank support for this bit vector to retrieve the rank of the sampled suffix array value, i.e., how many suffix array values $SA[j]$ with $j < i$ have been sampled. The rank value indicates the index where the value in SSA is stored. This approach improves the worst-case running time, but requires

$n(1 + o(1))$ additional bits of space for the bit vector and the rank support data structure by Jacobson.

Algorithm 4 Locating occurrences in a SSA (text order sampling)

```
1: procedure LOCATE(i, SSA)
2:     steps ← 0
3:     while B[i] ≠ 1 do
4:         i ← LF(i)
5:         steps ← steps + 1
6:     return ((SSA[rank_1(B, i)] + steps − 1) mod n) + 1
```

By additionally enforcing the sampling of position i such that $SA[i] = 1$, the return statement in algorithm 4 can be simplified to $SSA[\text{rank}_1(B, i)] + steps$, since an overflow can be ruled out.

The space consumption of the sampled suffix array can be reduced further, since the least significant bits do not have to be stored. If η is a power of two, the lowest $\log \eta$ bits are equal for each value in SSA. In practice one would select values such that $SA[i] \bmod \eta = 0$ and shift them by $\log \eta$ bits to the right. Hence, each value can be stored using $\lceil \log(n/\eta) \rceil$ bits. To restore the original SSA values, they have to be shifted by $\log \eta$ bits to the left again.

2.4.5.3 Summary

Now we have all it needs for fully functional unidirectional and bidirectional full-text indices allowing backward and forward searches. Searching a pattern in the FM index needs an efficient way to answer C, *Occ* and *Prefix-Occ* queries on the BWT, locating a pattern additionally requires a sampled suffix array. All this together constitutes the FM index.

Searching a query takes $\mathcal{O}(m \cdot \mathcal{T}_{Occ})$, locating all of its z occurrences takes additionally $\mathcal{O}(z \cdot \eta \cdot \mathcal{T}_{Occ})$ using a sampled suffix array with text order sampling.

2.4.6 Representation of the Burrows-Wheeler Transform

The last remaining question is how to store the BWT. It has to support three kind of queries:

1. $Occ(c, i)$, $c \in \Sigma$ and $1 \leq i \leq n$
2. $Occ(L[i], i)$, $1 \leq i \leq n$
3. *Prefix-Occ*(c, i), $c \in \Sigma$ and $1 \leq i \leq n$

Occ queries are needed for searching a sequence in the index, random access to the BWT itself is needed by the LF mapping for locating text positions in the sampled suffix array and *Prefix-Occ* queries are needed for synchronizing a bidirectional FM index which can be reduced to *Occ* queries.

The operations can be trivially implemented by precomputed tables. To answer *Occ* queries in constant time $n\,\sigma \log n$ bits are necessary if the values are stored for each character and each position in L. Since this implementation requires a lot of space, we present different data structures supporting these operations and compare them in terms of asymptotic running time, space complexity and benchmarks.

All data structures in table 2.1 will be covered for unidirectional FM indices only. For bidirectional indices they have to be built on the BWT of the reversed text as well, leading to twice the amount of space. For better readability ceiling functions were omitted.

Data structure	Backward search	Forward search	Space consumption in bits
Array	$\mathcal{O}(1)$	$\mathcal{O}(\sigma)$	$n\sigma \log n$
BV + RS	$\mathcal{O}(1)$	$\mathcal{O}(\sigma)$	$n\sigma(1 + o(1))$
WT + RS	$\mathcal{O}(\log \sigma)$	$\mathcal{O}(\log \sigma)$	$\mathcal{O}(n \log \sigma) + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$
r-ary WT + RS	$\mathcal{O}(\log_r \sigma)$	$\mathcal{O}(\log_r \sigma)$	$\mathcal{O}\left(\log_r \sigma \left(rn\frac{\log\log n}{\log_r n}\right)\right) + \mathcal{O}(\sigma \log n)$
BV + PRS	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$n(\sigma - 1) \cdot (1 + \cdot o(1))$
EPR	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n \log \sigma) + o(n\sigma \log \sigma)$

Table 2.1: Comparison of data structures for representing the BWT. We will present two improved implementations, both supporting constant running time for backward and forward searches. (BV = plain bit vectors, RS = rank support by Jacobson, PRS = prefixsum rank support, WT = wavelet tree, EPR = enhanced prefixsum rank dictionary). Small data structures such as the C table are neglected.

2.4.6.1 Bit Vectors with (Prefixsum) Rank Support

A straightforward approach to answer *Occ* queries in constant time is to store a bit vector $B_c[1..n]$ with rank support for each $c \in \Sigma$ indicating whether $L[i] = c$. An example is given in Figure 2.8.

Definition 2.4.8 (Bit vectors for BWT).
We define bit vectors B_c of length n for all characters $c \in \Sigma$ such that $B_c[i] = 1$ if and only if $L[i] = c$ for $1 \leq i \leq n$.

Theorem 2.4.9.
Backward searches in a unidirectional FM index can be performed in constant time, backward and forward searches in a bidirectional FM index in $\mathcal{O}(\sigma)$ time each, and LF mappings in $\mathcal{O}(\sigma)$. The data structure takes up $n\sigma(1+o(1))$ bits of space for a unidirectional index.

Proof. *Occ* queries can be reduced to constant-time rank queries on the bit vector, *Prefix-Occ* queries need $\mathcal{O}(\sigma)$ *Occ* queries leading to a synchronization overhead in bidirectional FM indices of $\mathcal{O}(\sigma)$ for both forward and backward searches. Random access to the BWT takes also $\mathcal{O}(\sigma)$, since L is not stored explicitly and all bit vectors B_c have to be examined at position i to determine $L[i]$. □

i	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	i	p	s	s	m	$	p	i	s	s	i	i
$B_\$[i]$	0	0	0	0	0	1	0	0	0	0	0	0
$B_i[i]$	1	0	0	0	0	0	0	0	0	0	1	1
$B_m[i]$	0	0	0	0	1	0	0	1	0	0	0	0
$B_p[i]$	0	1	0	0	0	0	1	0	0	0	0	0
$B_s[i]$	0	0	1	1	0	0	0	0	1	1	0	0

Figure 2.8: BWT of $T = mississippi\$$ stored in separate bit vectors for each character with rank support.

When using bit vectors with rank support for each character, we suggest a different definition of the bit vectors to improve the asymptotic running time of *Prefix-Occ* queries and random access to the BWT [Pockrandt et al., 2017]:

Definition 2.4.9 (Prefixsum bit vectors for BWT).
We define prefixsum bit vectors PB_c of length n for all characters $c \in \Sigma$ such that $PB_c[i] = 1$ if and only if $L[i] \leq c$ for all $1 \leq i \leq n$.

Theorem 2.4.10.
Occ queries and *Prefix-Occ* queries can be computed in $\mathcal{O}(1)$, random access to the BWT in $\mathcal{O}(\log \sigma)$ while taking up $n(\sigma - 1)(1 + o(1))$ bits of space. This improves backward and forward searches to $\mathcal{O}(1)$ for bidirectional FM indices while storing one bit vector less.

Proof. Adding rank support to the prefixsum bit vectors, leads to constant time *Prefix-Occ* queries instead of *Occ* queries. An *Occ* query can be performed by two (prefixsum) rank query. To access $L[i]$ a binary search on Σ can be performed:

$$
\begin{aligned}
\textit{Prefix-Occ}(c, i) &= \operatorname{rank}(PB_c, i) \\
\textit{Occ}(c_j, i) &= \begin{cases} \operatorname{rank}(PB_{c_j}, i) - \operatorname{rank}(PB_{c_{j-1}}, i) & , j > 0 \\ \operatorname{rank}(PB_{c_j}, i) & , j = 0 \end{cases} \\
L[i] &= c_j, \text{ s.t. } PB_{c_j}[i] = 1 \wedge (j = 1 \vee PB_{c_{j-1}}[i] = 0)
\end{aligned}
$$

Since $PB_{c_\sigma}[i] = 1$ and $\operatorname{rank}(PB_{c_\sigma}, i) = i$ for all i, PB_{c_σ} does not have to be stored. Thus one bit vector less is needed, including rank support tables. □

We will refer to rank support on prefixsum bit vectors as *prefixsum rank support*. As there is only one sentinel character in T, it is not necessary to store the bit vector for \$. Instead it is sufficient to store the index position *pos* of \$ in L.

i	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	i	p	s	s	m	$	p	i	s	s	i	i
$PB_{\$}[i]$	0	0	0	0	0	1	0	0	0	0	0	0
$PB_i[i]$	1	0	0	0	0	1	0	0	0	0	1	1
$PB_m[i]$	1	0	0	0	1	1	0	1	0	0	1	1
$PB_p[i]$	1	1	0	0	1	1	1	1	0	0	1	1
$PB_s[i]$	1	1	1	1	1	1	1	1	1	1	1	1

Figure 2.9: BWT of $T = mississippi\$$ stored in separate prefixsum bit vectors for each character with rank support. PB_s can be omitted.

$$\begin{aligned} B_{\$}[i] &= PB_{\$}[i] = \begin{cases} 1 & , i = pos \\ 0 & , \text{otherwise} \end{cases} \\ \text{rank}(B_{\$}, i) &= \text{rank}(PB_{\$}, i) = \begin{cases} 1 & , i \geq pos \\ 0 & , \text{otherwise} \end{cases} \end{aligned} \tag{2.8}$$

This trick can be applied similarly to all of the data structures presented in this chapter.

2.4.6.2 Wavelet Trees with Rank Support

A more space efficient data structure for representing the BWT and answer *Occ* queries are wavelet trees [Grossi et al., 2003] that were later suggested for bidirectional FM indices by Schnattinger et al. [Schnattinger et al., 2010]. Wavelet trees are balanced binary trees. Each node v has an alphabet Σ_v associated. The root node is denoted as *root* and represents the entire alphabet of the BWT, i.e., $\Sigma_{root} = \Sigma$.

Each node v with $|\Sigma_v| > 1$ has a left and a right child, denoted as *v.left* and *v.right*. At each node v the alphabet is partitioned into two sets of roughly equal size. Let $\Sigma_v = \{c_\ell, c_{\ell+1}, \ldots, c_r\}$ with $1 \leq \ell < r \leq \sigma$. Then we split it into $\Sigma_{v.left} = \{c_j \mid \ell \leq j \leq m\}$ and $\Sigma_{v.right} = \{c_j \mid m+1 \leq j \leq r\}$ with $m = \lfloor(\ell + r)/2\rfloor$. $\Sigma_{v.left}$ and $\Sigma_{v.right}$ are the alphabets associated with the left and right child of v. A node v becomes a leaf if $|\Sigma_v| = 1$.

Observation 2.4.2.
The binary tree is balanced and of height $\Theta(\log \sigma)$.

Additionally, each node v has a string L_v associated which is the BWT L with only the characters that are in Σ_v, i.e., all other characters $\Sigma \setminus \Sigma_v$ are removed from L. This string is not stored, but used conceptually to build and store a bit vector $B_v[1..|L_v|]$ for each inner node v. $B_v[i] = 1$ if and only if $L_v[i] \in \Sigma_{v.right}$, i.e., a *one* indicates that the character belongs to the alphabet of the right child.

Figure 2.10 illustrates the conceptual wavelet tree. Only the bit vectors and the tree structure are stored. Additionally, rank support is needed on all bit vectors to answer *Occ* queries and perform random access on the BWT string efficiently.

Theorem 2.4.11.
The data in the wavelet tree with rank support takes $\mathcal{O}(n \log \sigma) + o(n \log \sigma)$ bits of space. For representing the tree structure $\mathcal{O}(\sigma \log n)$ additional bits are needed.

Proof. The bit vectors on each level of the tree can be concatenated leading to a bit vector of length n taking up $n(1 + o(1))$ bits of space including rank support. As there are $\lceil \log \sigma \rceil$ levels, no more than $\mathcal{O}(n \log \sigma) + o(n \log \sigma)$ bits are needed. Let B_d be the concatenated bit vector of nodes on level d. For each of the $\sigma - 1$ inner nodes an interval $[\alpha_v, \beta_v]$ is stored indicating the start and end position of B_v in B_d, i.e., $B_d[\alpha_v, \beta_v] = B_v$. This requires $\mathcal{O}(\sigma \log n)$ additional space. Rank queries can still be computed in constant time. If B_v is

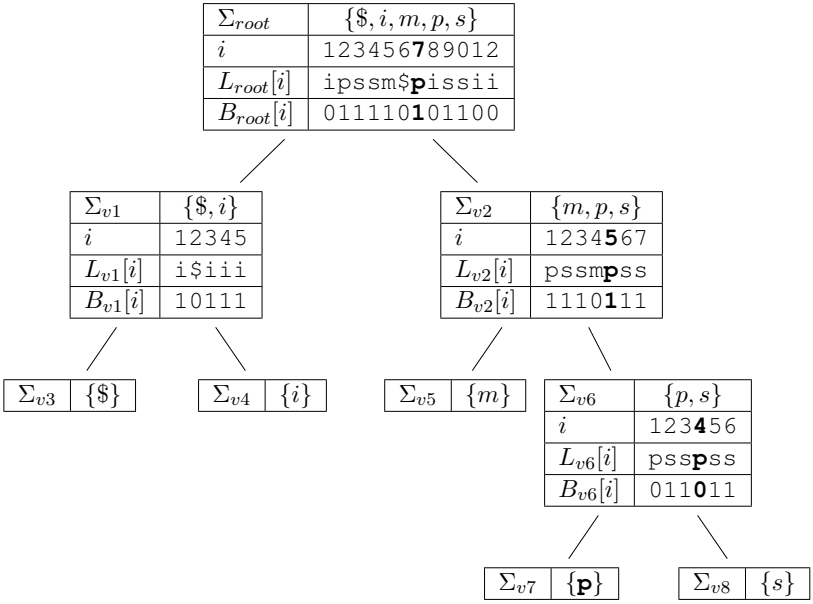


Figure 2.10: Wavelet tree of the BWT string $L = ipssm\$pissii$. $L_{root}[7]$ is highlighted to illustrate random access and *Occ* queries.

not the first bit vector in B_d, the rank query is computed as follows: $\text{rank}_1(B_v, i) = \text{rank}_1(B_d, \alpha_v + i - 1) - \text{rank}_1(B_d, \alpha_v - 1)$. □

Theorem 2.4.12.
Occ queries and random access to the BWT can be performed in $\Theta(\log \sigma)$.

Algorithm 5 gives the pseudocode for accessing a character $L[i]$. Starting from the root node of the tree, the bit at $B_{root}[i]$ is examined. If it is set, then $L[i] \in \Sigma_{root.right} = \{c_{m+1}, c_{m+2}, \ldots, c_\sigma\}$, otherwise $L[i] \in \Sigma_{root.left}$ with $m = \lfloor (1 + \sigma)/2 \rfloor$. This is continued recursively in the corresponding child. W.l.o.g. we assume that $B_{root}[i] = 1$ and

continue in the right child. Since $L_{root.right}$ only contains the characters from L that are in $\Sigma_{root.right}$, the position to consider in node *root.right* changes. Let $i' = \text{rank}_1(B_v, i)$, then $B_{root}[i]$ corresponds to $B_{root.right}[i']$. The recursion ends when a leaf is encountered and the character represented by the leaf is returned.

Algorithm 5 Random access to L

```
1: procedure WT-GetValue(i, v, [ℓ..r])
2:     if ℓ = r then
3:         return c_ℓ
4:     m ← ⌊(ℓ+r)/2⌋
5:     if B_v[i] = 0 then
6:         return WT-GetValue(rank_0(B_v, i), v.left, [ℓ..m])
7:     else
8:         return WT-GetValue(rank_1(B_v, i), v.right, [m+1..r])
9:
10: procedure WT-GetValue(i)
11:     return WT-GetValue(i, r, [1..σ])
```

Example 2.4.4 (Random access).
Consider retrieving $L[7]$ from the wavelet tree in figure 2.10. As pointed out earlier, only the bit vectors B_v with rank support are stored. Since $B_{root}[7] = 1$, it follows that $L[7] \in \{m, p, s\}$. The search is continued in the right child *root.right*. $L_{root.right}$ only contains the characters $\{m, p, s\}$, the character from $L[7] = L_{root}[7]$ is located in $L_{root.right}[\text{rank}_1(B_{root}, 7)] = L_{v2}[5]$ and thus $B_{v2}[5]$ is considered now. Since the bit is set, the search is again continued in the right child with the recomputed position, i.e., $L_{v2.right}[\text{rank}_1(B_{v2}, 5)] = L_{v6}[4]$. Since $B_{v6}[4] = 0$, the search is continued in the left child. The left child is a leaf, the alphabet size is reduced to 1, hence we can conclude that $L[7] = p$.

Performing *Occ* queries works similarly as it can be seen in algorithm 6. While in algorithm 5 the child is chosen based on whether the corresponding bit is set or not, for $Occ(c, i)$ queries the child whose associated alphabet contains c is chosen. The result of the *Occ* query is the last rank query performed before ending in a leaf. This rank query returns the number of occurrences up to the original position in L of the character that is represented in the leaf, which is c. Taking up on the previous example (assuming $Occ(p, 7)$ is computed), $\text{rank}_0(B_{v6}, 4) = 2$ represents the occurrence value $Occ(p, 7)$.

Additionally a value s, which is short for *smaller* can be computed. While going down in the tree, the number of occurrences in $L[1..i]$ that are smaller than c are counted. This is achieved by adding up $\text{rank}_0(B_v, i)$ in each node whenever we go down the right child and thus eliminate characters smaller than c in $L_{v.right}$. Hence, while computing the $Occ(c, i)$ value, $s = \textit{Prefix-Occ}(c-1, i)$ is computed as well without increasing the time complexity for bidirectional searches.

Algorithm 6 Computing Occ and Prefix-Occ values

```
1:  procedure WT-Occ(c_j, i, v, [ℓ..r], s)
2:      if ℓ = r then
3:          return (i, s)
4:      m ← ⌊(ℓ+r)/2⌋
5:      if B_v[i] = 0 then                          ▷ equivalent to j ≤ m
6:          return WT-Occ(c_j, rank_0(B_v, i), v.left, [ℓ..m], s)
7:      else
8:          s ← s + rank_0(B_v, i)
9:          return WT-Occ(c_j, rank_1(B_v, i), v.right, [m + 1..r], s)
10:
11: procedure WT-Occ(c_j, i)
12:     return WT-Occ(c_j, i, r, [1..σ], 0)
```

2.4.6.3 Generalized Wavelet Tree with Rank Support

The idea of wavelet trees for storing the BWT has been generalized from a binary wavelet tree to an r-ary wavelet tree [Ferragina et al., 2007]. Wavelet trees are balanced binary trees partitioning the alphabet Σ_v at each node v into two sets and storing a bit vector B_v indicating which set the corresponding character in the sequence L_v belongs to. r-ary wavelet trees partition the alphabet at each node into r sets of roughly equal size and store an integer string I_v over the alphabet $[1, r]$ (instead of a bit vector B_v). $I_v[i]$ indicates which set the character in $L_v[i]$ belongs to. Figure 2.11 shows the same Burrows-Wheeler transform from figure 2.10 represented in a 3-ry wavelet tree.

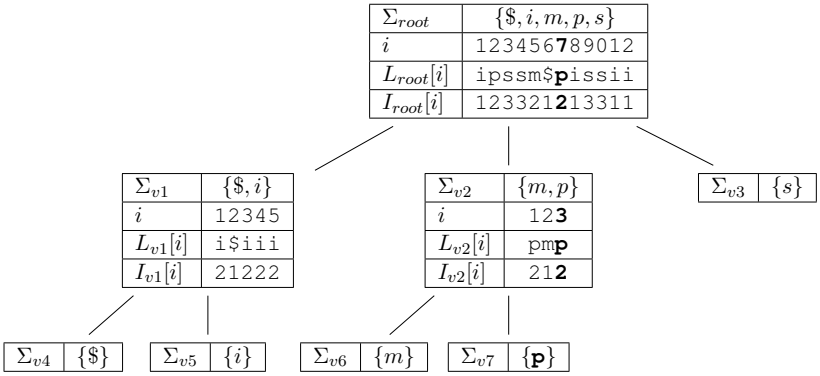


Figure 2.11: 3-ary wavelet tree of the BWT string $L = ipssm\$pissii$. $L[7]$ is highlighted to illustrate random access and Occ queries.

Similarly to binary wavelet trees, rank support is needed on the integer strings.

Definition 2.4.10 (Rank queries on integer strings).
Let I be an integer string over the alphabet $[1, r]$. $\text{rank}_c(B_v, i)$ counts the number of occurrences of c in the prefix $I[1..i]$ for $1 \leq c \leq r$ and $1 \leq i \leq |I|$.

The details of the sequence representation and constant-time rank support on integer strings [Raman et al., 2002] are not elaborated here as the data structure is more of theoretical interest. Instead, we will show in the next section how efficient rank support on integer strings can be realized in practice while still achieving optimal running time.

Performing queries on the wavelet tree is identical for both, binary and generalized wavelet trees. While the height of the tree is reduced and thus the running time of *Occ* queries and random access is improved, the space consumption increases. *Occ* queries and random access can be performed in $\mathcal{O}(\log_r \sigma)$ time. The space consumption of the sequence representation of the BWT and rank support is $\mathcal{O}\left(\log_r \sigma \left(rn\frac{\log\log n}{\log_r n}\right)\right)$. Additionally $\mathcal{O}(\sigma \log n)$ bits are needed for storing the tree structure respectively the bounds of delimiters of the concatenated sequences on each level similar to the binary wavelet tree.

The authors showed further that for $\sigma \in \mathcal{O}(\text{polylog}(n))$[2] the parameter r can be chosen such that *Occ* queries and random access take only constant time while still having an efficient space consumption. The details are omitted here as they are purely theoretical.

However, generalized wavelet trees do not support efficient *Prefix-Occ* queries and thus no efficient synchronization of bidirectional FM indices. When going down a node in the wavelet tree the *smaller* value has to be computed similarly to binary wavelet trees. Consider the 3-ary wavelet tree from figure 2.11. When $Occ(s, 9)$ is computed and the algorithm goes down to the rightmost child of the root node, the rank value $\text{rank}_c(I_{root}, 9)$ for each $1 \leq c < s$ has to be computed.

[2] $\mathcal{O}(\text{polylog}(n)) = \mathcal{O}(\log^k n)$ for some $k \in \mathbb{N}$

This increases the running time to $\mathcal{O}(r \log_r \sigma)$ as for each level there can be up to $r-1$ characters smaller than c that need to be considered.

2.4.6.4 Enhanced Prefixsum Rank Dictionary

An optimal running time with fast practical implementations can be achieved by EPR dictionaries, short for *Enhanced Prefixsum Rank* dictionaries [Pockrandt et al., 2017]. All three, *Occ* and *Prefix-Occ* queries as well as random access to the BWT can be performed in constant time leading to FM indices with backward and forward searches in $\mathcal{O}(1)$, and locating in $\mathcal{O}(\eta)$ for each occurrence.

Broadly speaking, EPR dictionaries store the BWT string L in its binary representation L_{bin} and build a constant time rank support data structure on top. The alphabet $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ is represented by the binary encoding of the integers $[0, \sigma - 1]$. Thus, a single character needs $w = \lceil \log \sigma \rceil$ bits. L takes up $n \cdot w$ bits of space when stored with bit packing, i.e., storing multiple characters in a single byte or larger word.

Now rank support is built on top of L_{bin} counting prefixsum rank values for each character $c \in \{c_1, c_2, \ldots, c_{\sigma-1}\}$. As we have seen in section 2.4.6.1, it is not necessary to build prefixsum rank support for c_σ, thus it is neglected throughout this section. It has to be considered that a position does not correspond to a single bit, but to w bits in L_{bin}. We assume for now that the block length ℓ is a multiple of $2w$. If this is not the case, padding strategies can be applied.

We define a few precomputed bitmasks, all of length ℓ: M_{even}, that has the bits set at even positions (considering the character width w), M_{carry} storing the value 1 in binary representation at odd positions and M_{c_j} for each character c_j with $1 \leq j < \sigma$ storing the value $j - 1$ at even positions and the value 1 at odd positions (c_j is represented as $j - 1$, since the alphabet is mapped from $[1, \sigma]$ to $[0, \sigma - 1]$ in the computer representation):

$$
\begin{array}{lcccccc}
M_{even} & = & 0^w\,1^w & 0^w\,1^w & ... & 0^w\,1^w \\
M_{carry} & = & 0^{w-1}1\,0^w & 0^{w-1}1\,0^w & ... & 0^{w-1}1\,0^w \\
M_{c_j} & = & 0^{w-1}1\,(j{-}1)_{bin} & 0^{w-1}1\,(j{-}1)_{bin} & ... & 0^{w-1}1\,(j{-}1)_{bin}
\end{array}
$$

We will now introduce the EPR bit vector $B_{EPR}(c_j)$ that is of length $|L_{bin}|$ and indicates by a bit at each characters position whether the character in L_{bin} is smaller or equal to c_j. It can be computed block by block. We denote $\overline{L}_{bin}$ as a block of length ℓ of L_{bin}. The fundamental idea is to use the precomputed bitmask M_{c_j}, perform a subtraction and count how many carry bits were introduced. This is done for even and odd positions separately to generate space for an overflow bit and avoid interference with carry bits of adjacent positions.

Step 1 To count occurrences of a character c_j at even positions, the characters at odd positions are masked out by $\overline{L}_{bin}\,\&\,M_{even}$. If the result is subtracted from M_{c_j}, characters at even positions that are larger than c_j will introduce a carry bit that will be eliminated by the subtraction, since the rightmost bit at odd positions in M_{c_j} is set. If the character is smaller or equal to c_j there will not be an underflow, hence no carry bit. The preset bit in M_{c_j} remains. In the end we can count the remaining carry bits (the rightmost bit at odd positions) by selecting them with a bitwise *and* with M_{carry}. By shifting the result by w bits to the right, the bits are aligned with the corresponding character positions in L_{bin}. The resulting bit vector is called $B_{EPR}(c_j)_{even}$ indicating the characters smaller or equal than c_j by a single bit.

$$B_{EPR}(c_j)_{even} = ((M_{c_j} - (\overline{L}_{bin}\,\&\,M_{even}))\,\&\,M_{carry}) \gg w$$

Step 2 The same is done for odd positions of $\overline{L}_{bin}$ by shifting it by w bits to the right and applying the same strategy as for even positions again. The resulting bit vector is called $B_{EPR}(c_j)_{odd}$. No shifting of the resulting bit vector has to be performed, since the carry bits are already aligned with the odd character positions.

$$B_{EPR}(c_j)_{odd} = (M_{c_j} - ((\overline{L}_{bin} \gg w) \,\&\, M_{even})) \,\&\, M_{carry}$$

Step 3 The EPR bit vector for c_j is then retrieved by merging both bit vectors $B_{EPR}(c_j)_{even}$ and $B_{EPR}(c_j)_{odd}$ with a bitwise *or*.

$$B_{EPR}(c_j) = B_{EPR}(c_j)_{even} \mid B_{EPR}(c_j)_{odd}$$

Figure 2.12 illustrates these steps in a small example. We omit the representation of the sentinel character as described in section 2.4.7.2, hence $w = 2$ for the DNA alphabet.

Prefixsum rank support on L can now be defined using rank queries on its transformed EPR bit vector:

$$\textit{Prefix-Occ}(c_j, i) = \text{rank}(B_{EPR}(c_j), i \cdot w) \tag{2.9}$$

Occ queries can be computed from *Prefix-Occ* queries:

$$\textit{Occ}(c_j, i) = \begin{cases} \textit{Prefix-Occ}(c_j, i) - \textit{Prefix-Occ}(c_{j-1}, i) & , j > 0 \\ \textit{Prefix-Occ}(c_j, i) & , j = 0 \end{cases} \tag{2.10}$$

In practice ℓ will be the size of CPU registers, which is 64 bit (or another power of two). There can never be an odd number of values in a 64 bit word (assuming $w \mid 64$). Hence, we can always shift $\overline{L}_{bin}$ to the right to compute $B_{EPR}(c)_{odd}$. If $w \nmid 64$, it is guaranteed that there will be at least one unused bit (at the right end) which will allow us shifting $\overline{L}_{bin}$ (and the bitmasks) by at least one bit to the right.

M_G		01	10	01	10	01	10	01	10
$\overline{L}_{bin}$ & M_{even}	−	00	01	00	01	00	11	00	11
		-	(C)	-	(C)	-	(T)	-	(T)
	=	**01**	01	**01**	01	**00**	11	**00**	11
M_{carry}	&	01	00	01	00	01	00	01	00
	=	**01**	00	**01**	00	**00**	00	**00**	00
$B_{EPR}(G)_{even}$	$\gg_w$	00	**01**	00	**01**	00	**00**	00	**00**

(a) retrieving $B_{EPR}(G)_{even}$

M_G		01	10	01	10	01	10	01	10
$(\overline{L}_{bin} \gg w)$ & M_{even}	−	00	00	00	10	00	10	00	00
		-	(A)	-	(G)	-	(G)	-	(A)
	=	**01**	10	**01**	00	**01**	00	**01**	10
M_{carry}	&	01	00	01	00	01	00	01	00
$B_{EPR}(G)_{odd}$	=	**01**	00	**01**	00	**01**	00	**01**	00

(b) retrieving $B_{EPR}(G)_{odd}$

$B_{EPR}(G)_{even}$		00	01	00	01	00	00	00	00
$B_{EPR}(G)_{odd}$	|	01	00	01	00	01	00	01	00
$B_{EPR}(G)$	=	01	01	01	01	01	00	01	00

(c) retrieving $B_{EPR}(G)$

Figure 2.12: An example for $\Sigma = \{A, C, G, T\}$ that shows how to retrieve the EPR transformed bit vector $B_{EPR}(G)$ from $\overline{L} = ACGCGTAT$. The resulting bit vector $B_{EPR}(G)$ has a 1 for each character smaller or equal to G.

This will create enough space on the left of the word for counting the carry bit of the very first character.

The space consumption can be analyzed similarly to rank support on bit vectors by Jacobson. A two-level hierarchy is built on top of $B_{EPR}(c_j)$ for every $c_j \in \{c_1, c_2, \ldots, c_{\sigma-1}\}$. Thus, superblocks of length ℓ^2 and blocks of length ℓ are precomputed and stored in L_1 and L_2 respectively as follows.

$$\begin{aligned} L_1[q][c_j] &= \text{rank}(B_{EPR}(c_j), q \cdot \ell^2) \\ L_2[p][c_j] &= \text{rank}(B_{EPR}(c_j)[1+k..], p \cdot \ell) \end{aligned} \tag{2.11}$$

$k = \lfloor (p-1)/\ell \rfloor \cdot \ell^2$ is the number of bits before the beginning of the corresponding superblock. The lookup table $P[V][m][c_j] = \text{rank}(V, m \cdot w)$ stores precomputed prefixsum rank values for each bit vector V of length ℓ, representing a block of $B_{EPR}(c_j)$, each position m in V and each character c_j.

Again, we choose $\ell = \lfloor (\log n)/2 \rfloor$ to achieve only an additional space consumption of $o(\sigma \log \sigma n)$ bits.

Superblocks We have $\sigma - 1$ EPR bit vectors of length $n\lceil \log \sigma \rceil$. Each bit vector is split into $\lfloor (n\lceil \log \sigma \rceil)/\ell^2 \rfloor$ superblocks. Each entry takes $\lceil \log n \rceil$ bits for storing the prefixsum rank value:

$$\mathcal{O}\left(\sigma \cdot \frac{n \log \sigma}{\ell^2} \cdot \log n\right) = \mathcal{O}\left(\sigma \cdot \log \sigma \cdot \frac{n}{\log n}\right) = o(\sigma \log \sigma n)$$

Blocks Each bit vector is split into $\lfloor (n\lceil \log \sigma \rceil)/\ell \rfloor$ blocks. Every entry takes $\lceil \log \ell^2 \rceil$ bits of space for storing the prefixsum rank value within a superblock:

$$\mathcal{O}\left(\sigma \cdot \frac{n \log \sigma}{\ell} \cdot \log \ell^2\right) = \mathcal{O}\left(\sigma \cdot \log \sigma \cdot n \cdot \frac{\log \log n}{\log n}\right) = o(\sigma \log \sigma n)$$

Lookup table There are $2^{\lfloor \frac{\ell}{w} \rfloor}$ possible bit vectors of length ℓ (i.e., for each position only the rightmost bit can be set to 1) and $\lfloor \ell/w \rfloor$ characters represented in V. The result takes up $\lceil \log(\ell/w) \rceil$ bits ($w = \lceil \log \sigma \rceil$). Since the lookup table only counts carry bits and is independent of the character, it only needs to be stored once:

$$\mathcal{O}\left(2^{\frac{\ell}{w}} \cdot \frac{\ell}{w} \cdot \log \frac{\ell}{w}\right) = \mathcal{O}\left(2^{\frac{\log n}{2w}} \cdot \log(n-\sigma) \cdot \log\log(n-\sigma)\right)$$

$$= \mathcal{O}\left(\sqrt[2w]{n} \cdot \log n \cdot \log\log n\right) = o(n)$$

Theorem 2.4.13.
Backward and forward searches as well as LF mappings in an FM index can be computed in $\mathcal{O}(1)$ using EPR dictionaries taking only $\mathcal{O}(n \log \sigma) + o(n\sigma \log \sigma)$ bits of space and $\mathcal{O}(\sigma \log n)$ bits to store C.

Proof. Similarly to Jacobson prefixsum rank queries can be computed in constant time. Applying theorem 2.4.10, we can compute *Occ* and *Prefix-Occ* queries in constant time. Random access to the BWT can be performed by accessing L directly. Storing L takes $\mathcal{O}(n \log \sigma)$ bits. With $\ell = \lfloor \log n/2 \rfloor$ the size of the additional tables is $o(n\sigma \log \sigma)$. □

Theorem 2.4.14.
For small alphabets, i.e., $\sigma \in \mathcal{O}\left(\frac{\log n}{\log\log n}\right)$ EPR dictionaries achieve the same asymptotic space consumption as wavelet trees, $\mathcal{O}(n \log \sigma)$.

Proof. The largest of all rank support tables are the precomputed blocks for $\ell = \lfloor \log n/2 \rfloor$. For sufficiently small alphabet sizes the o-term meets the O-term for storing the BWT.

$$\mathcal{O}\left(\sigma \cdot \log \sigma \cdot n \cdot \frac{\log\log n}{\log n}\right) = \mathcal{O}(n \log \sigma)$$

□

2.4.7 Implementation Details

2.4.7.1 Rank Support in Practice

In the previous sections we chose the block lengths of bit vectors with rank support such that a good asymptotic behavior in terms of space consumption is achieved, especially for the precomputed lookup table P. Since we replace P by a popcount operation in practice and it would also be advisable to take the CPU architecture into account, we show two different implementations of a hierarchical rank support that are currently used in the SDSL and SeqAn2.

SDSL The fastest constant-time rank support implemented in the SDSL is called *rank9* [Vigna, 2008]. It increases the space consumption by 25% when built on a bit vector. The hierarchy consists of two levels: superblocks and blocks. A superblock spans 512 bits of the bit vector and stores the values in 64 bits, a block spans 64 bits and stores the values in $\log_2 512 = 9$ bits. Since the first precomputed block value is 0, it does not need to be stored explicitly. Thus 9 block values of 7 bits each, can be stored in a 64 bit word. This leads to 128 bits of precomputed information for each 512 bits of the original bit vector, hence a 25% overhead.

SeqAn2 This approach [Reinert et al., 2017] is a more general approach for arbitrary alphabets, i.e., it can also be used for rank support for non-binary alphabets, such as the EPR dictionary. SeqAn2 rank support allows up to 3 levels. The underlying string (such as the BWT or a bit vector) is stored in an array of 64 bit words. Each 64 bit word contains $64/\lceil \log \sigma \rceil$ values. If $\lceil \log \sigma \rceil \nmid 64$, some bits in each word are left unused. For the block, superblock (and possibly ultrablock) no bit packing is performed for a faster running time, but at the expense of a higher space consumption. The block size is 64 bit (resp. $64/\lceil \log \sigma \rceil$ values) and can be increased by a multiple of this value leading to more than one popcount operation. The

highest level (in our example ultrablocks) is stored in a 64 bit array, superblocks and blocks in a 32 respectively 16 bit array. To reduce the number of ultrablocks and superblocks, they span the maximum number of possible values. Ultrablocks span at most $2^{32} - 1$ values and superblocks at most $2^{16} - 1$, since superblocks and blocks cannot represent values larger than $2^{32} - 1$ respectively $2^{16} - 1$.

The size of the data structure can be reduced for smaller bit vectors by using a 32 bit array for the highest level instead of a 64 bit array. The integer size of the lower levels are divided by 2 as well.

For plain bit vectors used such as in wavelet trees the value size is set to $\sigma = 2$.

2.4.7.2 Eliminating the Sentinel Character

One trick used by libraries such as SeqAn2 is the elimination of the sentinel character. When appending it to the text to be indexed, it increases the alphabet size leading to a larger space consumption and possibly increased running time for some of the data structures. This can be significant for small alphabets such as the DNA alphabet which grows by 25% when adding the sentinel character.

Since the sentinel character only occurs once in the text at an unknown position in the BWT, it can be replaced by a character from the original alphabet and the position sen_pos of the sentinel is stored. Whenever an $Occ(c, i)$ query is performed, the result needs to be adjusted if c is the sentinel substitute and $i \geq$ sen_pos by decrementing the result. The same applies to *Prefix-*$Occ(c, i)$ queries. If the sentinel substitute is larger than c, the result has to be incremented. For random access to the BWT $L[i]$ one has to return the sentinel character if and only if $i =$ sen_pos. This eliminates the need for rank support for the sentinel character.

2.4.8 Benchmarks

After analyzing the asymptotic running time and space consumption, we will now compare different implementations in practice. The only available implementations of FM indices are based on binary wavelet trees and EPR dictionaries. We compare our implementations of wavelet trees and EPR dictionaries for both, unidirectional and bidirectional FM indices and also compare them to other open source implementations of bidirectional FM indices based on wavelet trees, one from the SDSL (Succinct Data Structure Library) [Gog et al., 2014] and the first implementation of a bidirectional FM index using wavelet trees [Schnattinger et al., 2010]. We will refer to these two bidirectional implementations as 2WT_{SDSL} and $2\text{WT}_{\text{Schna}}$.

Our FM index implementations are part of the sequence analysis library SeqAn2 [Reinert et al., 2017] that are based on wavelet trees and EPR dictionaries. The benchmarks were run on both unidirectional and bidirectional indices [Pockrandt et al., 2017]. They are referred to as WT_{Seq} and EPR_{Seq} respectively 2WT_{Seq} and 2EPR_{Seq}.

Other bidirectional indices such as bidirectional suffix arrays (called affix arrays [Strothmann, 2007]) are excluded as the construction of the index did not terminate within several days on our test data set for alphabets with $\sigma > 4$.

All tests were conducted on Debian GNU/Linux 7.1 with an Intel Xeon E5-2667V2 CPU. To avoid dynamic overclocking effects in the benchmark, the CPU frequency was fixed to 3.3 GHz and the benchmark was performed on a single thread. The data was stored on a virtual file system in main memory to avoid loading it from disk during the benchmark which might affect the results due to I/O operations.

A text of length 10^8 and 1 million sampled query sequences of length 200 were generated for different, typical biological alphabet sizes:

- 4 (DNA and RNA alphabet)
- 10 (reduced amino acid alphabets such as Murphy10 [Murphy et al., 2000])
- 16 (DNA alphabet for representing uncertainties via subsets)
- 27 (20 amino acids plus additional characters, e.g., for representing ambiguities or stop codons)

The queries were searched from right to left in unidirectional indices. In bidirectional indices, the first half of each query was searched using forward searches and the second half using backward searches. Table 2.2 shows the running times measured for the benchmark.

EPR dictionaries achieve a speedup between 40% (DNA) and 240% (amino acid) for unidirectional FM indices and are between 110% (DNA) and 360% (amino acid) faster for bidirectional FM indices when compared to wavelet trees. As bidirectional indices require synchronization, they are slower than unidirectional indices. The table also shows that the wavelet tree based indices in SeqAn are comparable to those of other libraries. 2WT_{SDSL} and 2WT_{Seq} are both faster than $2\text{WT}_{\text{Schna}}$, but 2WT_{SDSL} catches up in terms of running time with 2WT_{Seq} for larger alphabets. One of the reasons is that SeqAn eliminates the sentinel character, while the SDSL does not. This leads to an increased alphabet size. As a search step takes $\mathcal{O}(\log \sigma)$, the effect is getting smaller for larger alphabets. The unidirectional indices, WT_{SDSL} and WT_{Seq} show a similar pattern, however less significant.

Furthermore the running time of EPR dictionaries across different alphabet sizes increases only slightly compared to wavelet trees. It is not constant as in theory, since larger alphabets increase the size of the data structures which lead to more cache misses.

EPR dictionaries reduce the running time from $\mathcal{O}(\log \sigma)$ to $\mathcal{O}(1)$ in theory. This holds also in practice as the speedup factor is very

close to $\log \sigma$. Similar results were observed for the DNA alphabet with real biological data on the human genome (except for $2WT_{Schna}$ which crashed during the construction of the larger FM index).

The same benchmark for $2EPR_{Seq}$ was also performed on multiple threads. The speedup for 4, 8 and 16 threads is 3.4, 7 and 12.9 which shows that it is suitable for parallelized algorithms, but has a memory bottleneck which is expected for string indices with frequent random memory access and only few computations.

	DNA		Murphy10		DNA16		Amino acid	
Index	Time	Factor	Time	Factor	Time	Factor	Time	Factor
WT_{Seq}	20.7s	1.00	52.4s	1.00	66.5s	1.00	85.6s	1.00
EPR_{Seq}	**15.1s**	**1.37**	**22.3s**	**2.35**	**23.4s**	**2.84**	**25.3s**	**3.38**
WT_{SDSL}	28.2s	0.73	59.3s	0.88	73.3s	0.91	94.4s	0.91
$2WT_{Seq}$	41.2s	1.00	66.6s	1.00	98.7s	1.00	121.0s	1.00
$2EPR_{Seq}$	**20.1s**	**2.05**	**23.8s**	**2.80**	**24.4s**	**4.05**	**26.1s**	**4.64**
$2WT_{SDSL}$	43.5s	0.95	74.7s	0.89	89.1s	1.11	109.4s	1.11
$2WT_{Schna}$	59.6s	0.69	91.3s	0.73	107.0s	0.92	130.0s	0.93

Table 2.2: Running times of various implementations in seconds and their speedup factors with respect to the unidirectional respectively bidirectional wavelet tree in SeqAn.

Table 2.3 compares the space consumption of different rank data structures. It only includes the wavelet trees and the EPR dictionaries. Additional data structures such as sampled suffix arrays are not included as their space is independent from the BWT representation and depend on further parameters such as the sampling strategy, sampling rate and bit packing. As expected, $2WT_{Seq}$ and $2EPR_{Seq}$ take exactly twice the amount of space as their unidirectional counterparts. Again, $2WT_{SDSL}$ is smaller for all alphabets than $2WT_{Schna}$ and approaches the space consumption of the SeqAn implementation $2WT_{Seq}$ for larger alphabets which is due to the explicit sentinel character in the SDSL. For smaller alphabets the space increases only by

40% (DNA), and are up to 6.6 as big for large alphabets (amino acid).

The space consumption for affix arrays is significantly larger. That makes it impractical for many biological applications. We included the index size for DNA alphabet and refer to it as *AF* [Meyer et al., 2011].

Index	DNA	Murphy10	DNA16	Amino acid
WT_{Seq}	30	51	60	72
WT_{SDSL}	34	53	61	73
EPR_{Seq}	42	156	227	478
$2WT_{Seq}$	60	102	120	144
$2WT_{SDSL}$	68	105	122	145
$2WT_{Schna}$	75	108	123	146
$2EPR_{Seq}$	84	311	454	955
AF	2670	-	-	-

Table 2.3: Space consumption of the rank data structure in Megabyte of various implementations.

For larger alphabets it can be beneficial to increase the number of levels of prefixsum rank support to 3. As table 2.4 shows, the space consumption can decrease by up to 40% (amino acid) while the running time increases by 23% for another array lookup.

Index	DNA	Murphy10	DNA16	Amino acid
2 level	42	156	227	478
3 level	36	109	150	291

Table 2.4: Space consumption of unidirectional EPR dictionaries with 2 and 3 level prefixsum rank support in Megabyte.

Finally, we take a look at the $o(n \log \sigma)$-term of wavelet trees and the $o(n\sigma \log \sigma)$-term of EPR dictionaries. To verify that for both data structures the $\mathcal{O}(n \log \sigma)$-term in practice is more dominant than the o-term, we build the EPR dictionary and wavelet tree on strings of different lengths and alphabets. For $\sigma = 4$ respectively $\sigma = 16$ (DNA and DNA16 alphabets) and lengths $10^4, 10^5, \ldots, 10^9$ we measured the space consumption and divided the space by the $\mathcal{O}$-term, namely $n\lceil \log \sigma \rceil$. This is the space that is used by both data structures for storing the BWT respectively the bit vectors (without rank support). Figure 2.13 visualizes the impact of the o-term. As expected, for both data structures the ratio converges to a constant. Since the o-term of EPR dictionaries is larger than the o-term of wavelet trees, the ratio converges more quickly. Furthermore, the space consumption for the rank support of EPR dictionaries is higher.

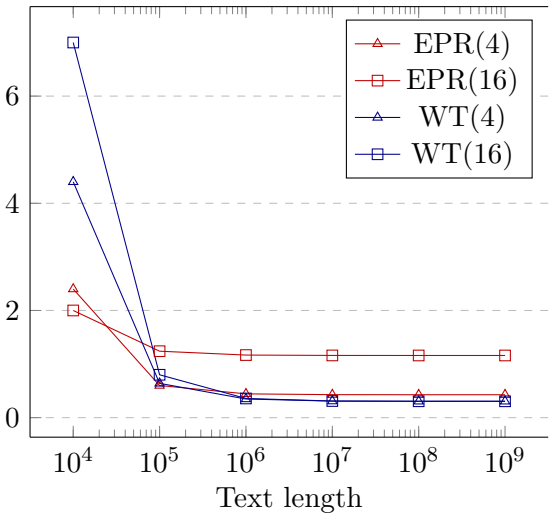


Figure 2.13: Decreasing impact of the o-term with respect to the $\mathcal{O}$-term in the space complexity.

In summary, EPR dictionaries improve the running time of both, unidirectional and bidirectional FM indices. They achieve an optimal running time in theory which we verified in benchmarks. It is also the first constant-time implementation. Other approaches with optimal running times do not have (publicly available) implementations [Ferragina et al., 2007, Belazzougui et al., 2013]. This makes EPR dictionaries probably the fastest bidirectional indices available at the time of this writing, especially useful for small alphabets as they are already twice as fast as wavelet trees for DNA alphabets while the total increase in space for the entire FM index is significantly less than 40% for text order sampling with a sampling rate of $\eta = 10$.

3 Indexed Search

3.1 Introduction

In the previous chapter we have shown how to search a query sequence in an index. For many bioinformatics applications one is not only interested in the locations of a given query sequence in the indexed text, but also the locations of approximate matches, i.e., sequences that are similar to the original query sequence based on some similarity measure.

Depending on the use case, there are different kind of error types and distance metrics to be considered. In sequence analysis we focus on substitutions (also referred to as mismatches), insertions and deletions, accounting for both sequencing errors and genetic mutations [Yang et al., 2012, Pavlopoulos et al., 2013]. Other error types such as translocations (swapping two characters at arbitrary positions) are not considered here as they are less relevant for sequence analysis.

```
Text        ...GTGAACACCAAATGACG-GGGGGGAG...
Alignment          |||x|||||| ||| |||||
Query              AACTCCAAAT-ACGCGGGGG
                   0123456789012345678 9
```

Figure 3.1: Example of different error types. At position 3 is a substitution, at position 10 a deletion and at position 14 an insertion of a character in the query sequence.

Definition 3.1.1 (String metric).
String or distance metrics measure the distance of two strings, i.e., the inverse of the strings similarity. A metric $D : \Sigma^* \times \Sigma^* \to \mathbb{N}_0$ has to fulfill four properties. Let x, y and z be strings of any length:

1. $D(x, y) \geq 0$ (non-negativity)
2. $D(x, y) = D(y, x)$ (symmetry)
3. $D(x, y) = 0 \iff x = y$ (identity of indiscernibles)
4. $D(x, z) \leq D(x, y) + D(y, z)$ (triangle-inequality)

Two widely used distance metrics are the *Hamming distance* and the *Levenshtein distance*, the latter is often referred to as *Edit distance*.

Definition 3.1.2 (Hamming distance).
Given two strings S and T of equal length, the Hamming distance $D_H(S, T)$ is the minimum number of character substitutions to transform one string into the other [Hamming, 1950].

Definition 3.1.3 (Levenshtein distance).
Given two strings S and T of arbitrary length, the Levenshtein distance $D_L(S, T)$ is the minimum number of character substitutions, insertions and deletions to transform one string into the other [Levenshtein, 1966].

An important problem in applications such as read mapping is approximate string matching: a query sequence S is searched in the indexed reference genome T with up to e errors given a distance metric.

Definition 3.1.4 (Approximate string matching).
Approximate string matching refers to finding all sequences S' in a text T with $D(S, S') \leq e$ for a given sequence S, error threshold e and distance metric D [Galil and Giancarlo, 1988].

The number of sequences S' grows exponentially in the number of errors. Given a sequence S of length k over an alphabet of size σ, equation 3.1 counts the number of sequences S' with $D_H(S', S) \leq e$.

$$\sum_{i=0}^{e} \binom{k}{i} \cdot (\sigma - 1)^i \tag{3.1}$$

Since this number gets even larger considering additional error types such as insertions and deletions, a lot of research has been dedicated to this area trying to speed up approximate string matching in an index. Instead of searching every single possible approximate match S' in the index, one makes use of combinatorial arguments to reduce the computational effort. Some of these are being introduced and compared in the following sections. If not stated otherwise, we will only consider Hamming distance. Many of the arguments and algorithms can be applied to Levenshtein distance just as well, but get more complex in the details.

In this chapter we only consider approaches that use full-text string indices such as suffix trees, (enhanced) suffix arrays, or FM indices that allow searching a query sequence of arbitrary length character by character. Furthermore we require a bidirectional index, i.e., we have to be able to extend characters to the left as well as to the right in any order. Thus we do not consider approaches using other string indices such as k-mer indices that only allow for a fixed query length.

In the following sections we will present different approaches to the approximate string matching problem. Starting from the trivial algorithmic solution of enumerating all possible approximate matches S' of a given query sequence S via backtracking, we will continue with combinatorial arguments such as the pigeonhole principle. Subsequently a framework called *search schemes* by Kucherov et al. is introduced. It allows formalizing search strategies for approximate string matching in indices and comparing their efficiency. After that, we will give an integer linear program to solve the approximate string

matching problem optimally (with certain constraints). We will compare the aforementioned strategies and search schemes given by Kucherov et al. to our optimal search schemes computed by the ILP from theoretical aspects as well as benchmarks on sequencing data. In the end we will discuss an additional technique to speed up exact and approximate string matching in indices that can be combined with all algorithms mentioned in this thesis.

3.2 Simple Backtracking

One of the simplest solutions to approximate string matching, besides searching each sequence S' one after the other in the index, is to search each common prefix of two sequences only once. This is achieved by a backtracking approach, illustrated in figure 3.2, first proposed by Ukkonen for approximate string matching in suffix trees [Ukkonen, 1993]. Starting from the root node, the original sequence S is searched in the index character by character. Each search step in the index is represented by an edge. Edges downwards represent matches, diagonal edges represent substitutions. For illustration purposes we display the character searched in the index next to some of the edges. The occurrences of the approximate matches are represented by leaves. We allow for substitutions at each position in S and continue the search recursively while keeping track of the remaining number of errors allowed. This strategy not only works for suffix trees, but for any of the aforementioned string indices.

The pseudocode is given in algorithm 7. The iterators wrap suffix array ranges of a suffix array or FM index. Instead of forward searches, backward searches can be performed if the sequences are reversed in advance. In line 10 the result of the Boolean expression is interpreted as an integer and used to decrement the number of errors in case of a mismatch.

The improvements in the following sections are all based on the

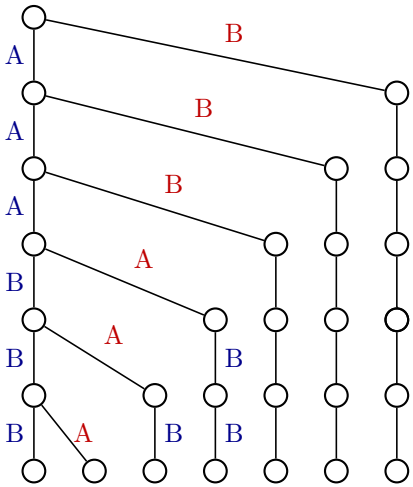


Figure 3.2: Simple backtracking approach for the query sequence $S = AAABBB$ on the binary alphabet $\Sigma = \{A, B\}$ with up to 1 substitution (Hamming distance).

backtracking approach and try to reduce the number of edges in the backtracking tree by the use of combinatorial arguments. To compare different strategies for the approximate string matching problem we will compare the number of edges in such backtracking trees, since going down an edges corresponds to a search step in the index.

The number of edges for this approach is given be the recursive formula in equation 3.2. Searching an error-free k-mer requires k search steps in an index. A non-error-free search leads to σ edges going down from the current node, one for each character. One of them matching the character in the query sequence, the others leading to a substitution.

$$c_{k,e} = \begin{cases} \sigma + c_{k-1,e} + (\sigma - 1) \cdot c_{k-1,e-1} & , k > 0 \wedge e > 0 \\ k & , \text{otherwise} \end{cases} \tag{3.2}$$

Algorithm 7 Simple Backtracking

```
 1: procedure BACKTRACKING(iterator, S, e)
 2:     if S = ε then                                   ▷ leaf reached
 3:         report iterator
 4:     else if e = 0 then                              ▷ no errors left
 5:         if iterator′ ← forward_search(iterator, S) then
 6:             report iterator′
 7:     else                                            ▷ errors left
 8:         for c ∈ Σ do
 9:             if iterator′ ← forward_search(iterator, c) then
10:                 e′ ← e − (c ≠ S[0])
11:                 BACKTRACKING(iterator′, S[1..], e′)
12:
13: procedure BACKTRACKING(S, e)
14:     iterator ← root              ▷ iterator stores suffix array ranges
15:     BACKTRACKING(iterator, S, e)
```

It is noteworthy that we only consider complete backtracking trees, i.e., we assume that all possible sequences S' occur in the text. In general, this not true. Some paths in the tree will be cut because the corresponding prefix does not occur in the text. In that case the search will not be continued in the corresponding subtree, since no approximate match with this prefix can occur in the text. While we assume in our model that the tree is complete, this should be kept in mind throughout the entire chapter. As we will see later in the experiments and benchmarks, the number of edges in a complete backtracking tree is still a suitable measure for comparing the performance of different search strategies.

3.3 Pigeonhole Search Strategy

Similar to the pigeonhole principle [Fletcher and Patty, 1987], which gives a lower bound when distributing objects into boxes we can derive the following observation giving an upper bound:

Theorem 3.3.1.
If we distribute n objects over m boxes, then there must be at least one box that contains at most $\left\lfloor \frac{n}{m} \right\rfloor$ objects.

Proof. By contradiction, we assume that the statement is not true, i.e., all boxes contain more than $\left\lfloor \frac{n}{m} \right\rfloor$ objects, given m boxes and n objects. We conclude that when counting the objects in all boxes, we would have at least $m + 1$ objects, a contradiction:

$$\begin{aligned} n \cdot \left(\left\lfloor \frac{m}{n} \right\rfloor + 1 \right) &\geq n \cdot \left(\frac{m}{n} - \frac{n-1}{n} + 1 \right) \\ &= m - (n-1) + n \\ &= m + 1 \end{aligned}$$

□

The general idea of improving the previous backtracking approach is to conceptually divide the sequence S into p pieces (representing boxes). The errors represent objects that can be distributed in any manner onto the pieces. Applying theorem 3.3.1, we consider the number of errors e fixed and set p accordingly. Thus we know that no matter how the errors (at most e) in S' are distributed, at least one of the pieces of that sequence contains at most $e' = \left\lfloor \frac{e}{p} \right\rfloor$ errors. The search algorithm then searches each of the p pieces separately with at most e' errors using the simple backtracking approach from the previous section, and after a successful search continues with the remaining pieces allowing for up to e errors in total. These cases all together guarantee that each possible error pattern (distribution of

errors onto pieces) is covered. The pieces can be chosen of any length as long as they remain fixed over all cases. For now we consider all pieces of equal length (for simplicity we assume $p \mid k$). If we set $p > e$ there is at least one piece that has to match without any error as $\left\lfloor \frac{e}{e+1} \right\rfloor = 0$. Any $p > e + 1$ will not improve the algorithm further with respect to theorem 3.3.1.

Definition 3.3.1 (Pigeonhole Search Strategy).
We call the strategy of choosing $p = e + 1$ the pigeonhole search strategy. It consists of p searches where each search starts with a different piece that is searched error-free and afterwards extended with up to e errors.

Figures 3.3 and 3.4 show the backtracking approach for $e = 1$ and $e = 2$ using the pigeonhole search strategy. This leads to two cases respectively three cases in which one piece is searched without errors and extended by the other piece(s) allowing for one respectively two errors. In the second tree for $e = 2$ one can see that it is necessary to have a bidirectional string index. The piece P_2 in the middle of the pattern is searched first and is extended afterwards to the right and to the left.

Just as for the simple backtracking approach, it is guaranteed that every error pattern is covered. While the simple backtracking approach guarantees that every error pattern is covered exactly once, this is not the case for the pigeonhole search principle. As the example in figure 3.3 shows, the exact match $S = AAABBB$ is covered by both cases.

Observation 3.3.1.
Every error pattern with no more than $\left\lfloor \frac{e}{p} \right\rfloor$ errors in each piece will be covered by more than one case.

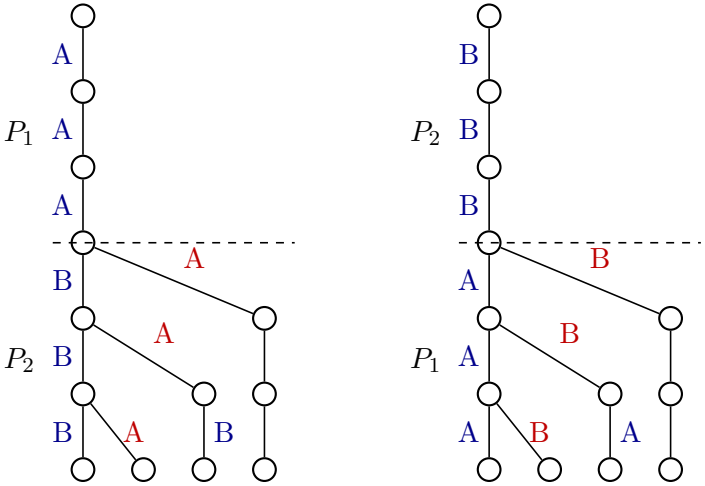


Figure 3.3: Searching the query sequence $S = AAABBB$ on the binary alphabet $\Sigma = \{A, B\}$ with up to one substitution (Hamming distance). The sequence is divided conceptually into two pieces P_1 and P_2 of equal size.

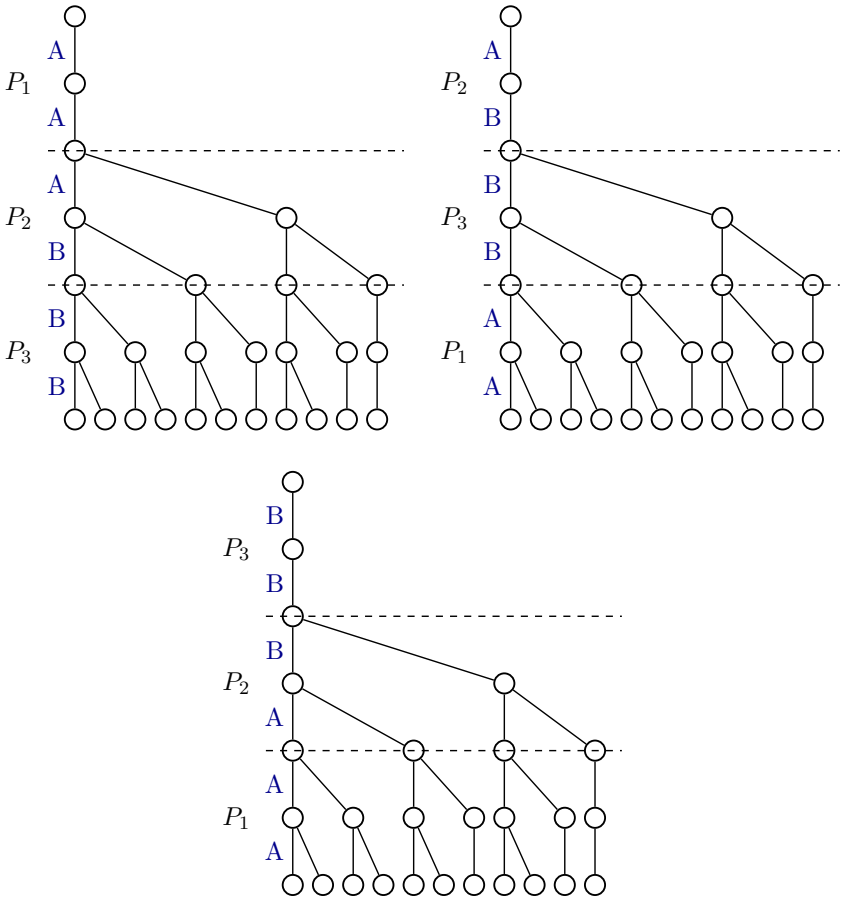


Figure 3.4: Searching the query sequence $S = AAABBB$ on the binary alphabet $\Sigma = \{A, B\}$ with up to two substitutions (Hamming distance). The sequence is divided conceptually into three pieces P_1, P_2, and P_3 of equal size. Each case starts searching a different piece without errors.

Example 3.3.1.
If we searched a sequence with $e = 5$ errors and $p = 2$ pieces, all error patterns that have up to $\lfloor \frac{5}{2} \rfloor = 2$ errors in each piece are covered by both cases, i.e., occurrences in the text that match these error patterns will be reported twice.

If multiple cases cover the same error pattern, occurrences of approximate matches are reported multiple times. For many applications such as read mapping this requires filtration of duplicates after searching. This is only true for Hamming distance though. For Levenshtein distance, even cases with disjoint error patterns or the simple backtracking approach can lead to redundant matches. This lies in the nature of the Levenshtein distance. A match can have multiple alignments, e.g., a single substitution can be expressed as an insertion followed by a deletion. It is also possible that different alignments have the same start and end positions in the text.

Applying the pigeonhole search strategy does not improve the running time of approximate string matching asymptotically, since after searching the first piece the rest of the sequence will be searched with the remaining error count. It still makes each of the backtracking trees sparser by the elimination of branching in the $\frac{k}{p}$ topmost nodes and thus decreasing the effect of the exponential growth. In the given example, the reduction of edges can be up to 50% depending on k. The simple backtracking approach for $e = 1$ according to equation 3.2 leads to $c_{k,1} = \sigma \cdot k + (\sigma - 1)\frac{k \cdot (k-1)}{2}$ edges for a sequence of length k whereas the pigeonhole search strategy leads to $\frac{k}{2}$ edges in the exact matching piece and $c_{\frac{k}{2},1}$ edges for the erroneous piece. For both cases this adds up to $k + 2c_{\frac{k}{2},1}$. For growing k a reduction of 50% of edges is achieved. W.l.o.g. we assume that k is even.

Equation 3.3 quickly approaches $\frac{1}{2}$. For the DNA alphabet ($\sigma = 4$) sequences as short as 6 base pairs have a reduction of 30% of edges, while sequences of length 14 already save 40% of edges. Searching a sequence of length 100 such as Illumina reads using the pigeonhole

search strategy, reduces the number of edges by 48.5%.

$$\begin{aligned}
&\lim_{k\to\infty} \frac{2\left(\frac{k}{2} + c_{\frac{k}{2},1}\right)}{c_{k,1}} \\
&= \lim_{k\to\infty} \frac{2\frac{k}{2} + 2\sigma\frac{k}{2} + 2(\sigma-1)\frac{1}{2}\frac{k}{2}(\frac{k}{2}-1)}{\sigma k + (\sigma-1)\frac{k(k-1)}{2}} \\
&= \lim_{k\to\infty} \frac{\frac{1}{k} + \sigma\frac{1}{k} + (\sigma-1)(\frac{1}{4} - \frac{1}{2k})}{\sigma\frac{1}{k} + (\sigma-1)(\frac{1}{2} - \frac{1}{2k})} \\
&= \frac{(\sigma-1)\frac{1}{4}}{(\sigma-1)\frac{1}{2}} = \frac{1}{2}
\end{aligned} \tag{3.3}$$

As shown later in section 3.7 the speedup in practice is greater than 2. This divergence comes from our assumption that the backtracking trees are complete. In general, we do not find every approximate match S' in the text. Intuitively speaking, the simple backtracking approach has to do many search steps near the root node as for short prefixes of S all approximate prefixes occur in the text. Thus, branching nodes close to the root should be avoided. This is addressed by the pigeonhole search strategy.

In comparison to the simple backtracking approach which requires only a unidirectional index, we need to be able to search into both directions when applying the pigeonhole principle. Especially for searches starting with a piece in the middle of the query sequence we need a bidirectional index such that one can switch directions during the search of the sequence.

The reduction of edges gets smaller with growing e. For two errors it approaches 11%, for more errors it even gets negative as we will see later in the experiments. Thus, we will need to consider stronger combinatorial arguments.

3.4 01*0 Search Strategy

Another combinatorial argument was given by [Vroland et al., 2016]. It is based on the following theorem.

Theorem 3.4.1 (01*0 Seeds).
If we have $n+2$ ordered boxes and distribute n objects among them ($n \in \mathbb{N}$), there must be a sequence of boxes such that the left- and rightmost boxes contain no object and the boxes in between exactly one object each.

Proof. There are at least 2 empty boxes. If μ is the number of boxes that contain more than one object, we can conclude that there are at least $\mu + 2$ empty boxes. Hence, there are $\mu + 1$ pairs of boxes with only non-empty boxes in between. Since there are only μ boxes with more than one object inside, only μ pairs can be interleaved by such a box. Thus, at least one pair of empty boxes remains that does not have a box in between with multiple objects. We conclude that there can only be boxes in between that each contain exactly one object or none. This guarantees that there is such a sequence of 01*0 boxes. □

This theorem also holds when less than n objects are distributed among $n + 2$ boxes. Thus it can be applied straightforward to the approximate string matching problem with up to e errors. The sequence is split into $p = e+2$ pieces representing boxes. Up to e errors can be distributed among those pieces. Each approximate match S' as well as the exact match S will have at least two error-free pieces where all pieces in between have exactly one error. This subsequence is called a 01*0 seed. It can then be extended to search the rest of the sequence.

Example 3.4.1.
Taking up the example from the previous section, the query sequence is divided into three pieces for $e = 1$. There are three possible error patterns containing 01^*0 seeds: $00x$, 010 and $x00$ where x represents a wildcard, i.e., any distribution of the remaining errors. Again, as for the pigeonhole principle, the sizes of pieces can be arbitrary as long as they are fixed over all cases.

Let S be a sequence that shall be searched with up to e errors in a string index. It is split into $p = e + 2$ pieces with $S = P_1 \cdot P_2 \cdot ... \cdot P_p$. The simplest, straightforward approach is to consider each possible pair (i, j) of pieces with $1 \leq i < j \leq p$ separately. In the first step, $P_i \cdot P_{i+1} \cdot ... \cdot P_j$ is searched from left to right where no errors are allowed in P_i and P_j. During the search of the pieces in between, backtracking is performed requiring exactly one error in each piece. In the second step, if the search did not abort beforehand, the entire sequence is extended to the right $P_{j+1} \cdot P_{j+2} \cdot ...P_p$ and to the left $P_1 \cdot P_2 \cdot ...P_{i-1}$ allowing for the remaining $e-(j-i-1)$ errors. This is implemented using simple backtracking again. It leads to $\binom{e+2}{2}$ cases respectively backtracking trees. Figure 3.5 illustrates an example with the corresponding backtracking trees.

To reduce the number of cases and thus the total number of search steps, one does not consider each pair separately, but each error-free starting piece leading to $p-1$ cases.Consider the search starting with piece P_i. It searches P_i without errors and continues with backtracking of P_{i+1} allowing for up to one error. If there is a path in the backtracking tree that matches P_{i+1} without errors, a 01^*0 seed has been found with no erroneous pieces in between and that path is continued by extending the sequence to the right and to the left with the remaining number of errors, i.e., e errors.

The other paths must have had exactly one error in P_{i+1}. They are continued by performing the previous step recursively, i.e., the next piece P_{i+2} is searched with up to one error. An exact match

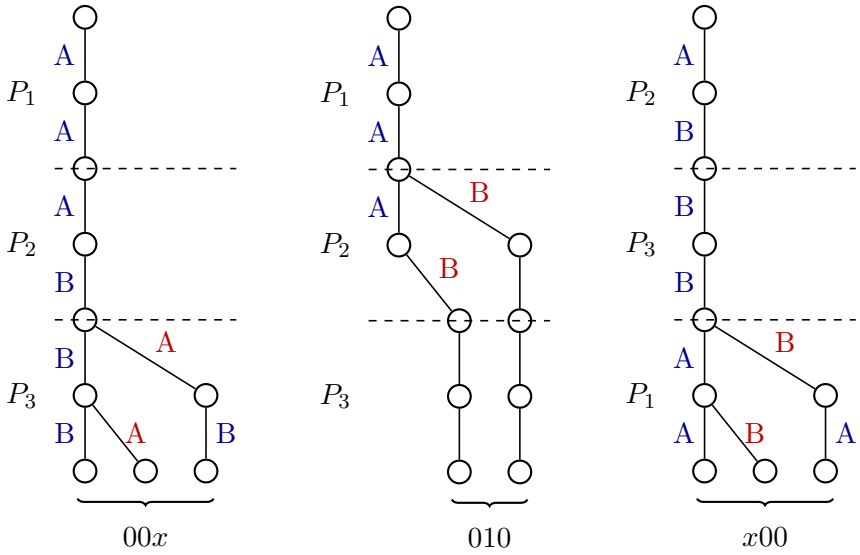


Figure 3.5: Searching the query sequence $S = AAABBB$ on the binary alphabet $\Sigma = \{A, B\}$ with up to one substitution (Hamming distance) applying the 01*0 search strategy. Backtracking is performed for each 01*0 seed separately.

of P_{i+2} finishes this step and continues extending the rest of the query sequence with the remaining number of $e-1$ errors. The paths that spent an error in P_{i+2} are continued recursively until the search aborts as no partial approximate matches can be found or the last piece P_p is reached, that must be error-free. Figure 3.6 illustrates the (complete) backtracking trees for $e = 1$ using this approach. It is superior to the previous one considering each 01*0 seed separately, since it avoids searching redundant prefixes multiple times.

By taking a quick glance one can see the improvement of 01*0 seeds over the pigeonhole search strategy. The 01*0 search strategy splits the sequence into $e + 2$ pieces, while the pigeonhole search

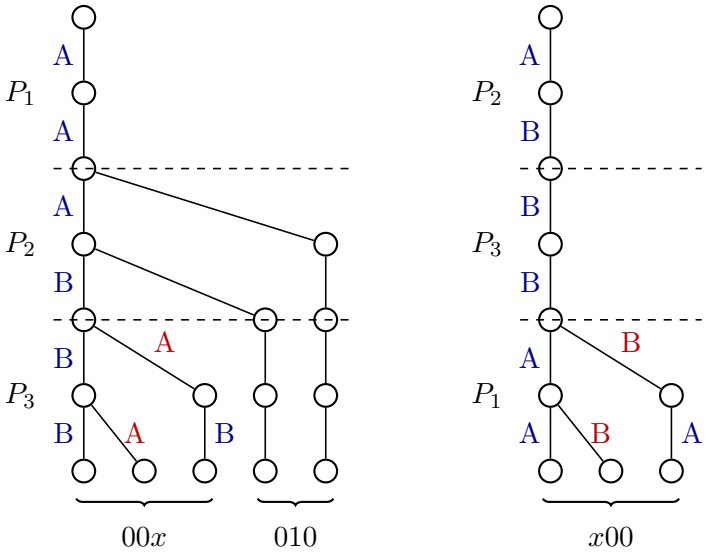


Figure 3.6: Searching the query sequence $S = AAABBB$ on the binary alphabet $\Sigma = \{A, B\}$ with up to one substitution (Hamming distance) applying the 01^*0 search strategy. Backtracking is performed for each piece P_i separately with $1 \leq i < p$.

strategy splits it into $e+1$ pieces. Thus, the pieces are slightly larger when using the pigeonhole search strategy. This leads to a marginally longer exact string search before errors are allowed which reduces the edges further. Whereas the pigeonhole search strategies continues with the maximum number of errors e from the second piece on, 01^*0 seeds only allows for up to one error in the next piece. This reduction of edges is much higher than the increase by the slightly shorter exact match before, which makes the 01^*0 approach superior to the pigeonhole principle for $e > 1$.

Analogously to the pigeonhole search strategy redundant approx-

imate and exact matches can occur, since they can contain multiple 01^*0 seeds. For example, an exact match contains multiple 00 seeds for $e > 0$, $p - 1$ seeds in total. This might require a subsequent filtration phase for duplicates depending on the application.

Vroland et al. implemented this approach in a tool called *Bwolo*. It is noteworthy, that they only use a unidirectional FM index, i.e., they search for 01^*0 seeds in the index and can only extend the seed afterwards to the right. It does not allow extending the sequence to the left (the indexed sequence as well as the query sequence are reversed to search it from left to right). Therefore after searching the 01^*0 seed $P_i \cdot P_{i+1} \cdot ... \cdot P_j$, they extend it to $P_i \cdot P_{i+1} \cdot ... \cdot P_p$ using backtracking, locate the positions in the text and verify whether $P_1 \cdot P_2 \cdot ...P_{i-1}$ matches at those text positions with the remaining number of errors. The authors use this approach, since at the time of publication SeqAn did not offer a bidirectional FM index. As we will see later in the benchmarks, this approach has significant speedups to pure index-bases searches. We will discuss so-called *in-text verification* in section 3.8.

3.5 Search Schemes

When Lam et al. introduced bidirectional FM indices [Lam et al., 2009], they also suggested search strategies for one and two errors in a bidirectional string index. For $e = 1$ they apply the pigeonhole search strategy as described in section 3.3, for $e = 2$ they suggest to split the query sequence into 3 pieces and describe their strategy verbally.

Kucherov et al. took up on that and introduced search schemes [Kucherov et al., 2016], a generalized framework to formalize and evaluate search strategies in a bidirectional full-text index.

Definition 3.5.1 (Search).
A search $S = (\pi, L, U)$ is a triplet that combines multiple error patterns. It defines a permutation string π and two integer strings L and U for lower and upper error bounds. π is a permutation of $\{1, 2, \ldots, p\}$ indicating the order in which the pieces are searched. L and U are both integer strings over $\{0, 1, \ldots, e\}$ of length p determining the number of errors allowed for each piece as cumulative values, i.e., when the ith piece is processed the number of accumulated errors must be between $L[i]$ and $U[i]$.

For a search to be valid, it needs to fulfill the *connectivity property* in equation 3.4 that ensures that the sequence can actually be searched in a bidirectional index, e.g., when beginning with pieces P_2 and P_3, the next pieces to be searched can only be P_1 and P_4, not P_5.

$$\forall i > 1 : \pi[i] \in \{\min_{j<i} \pi[j] - 1, \max_{j<i} \pi[j] + 1\} \tag{3.4}$$

Definition 3.5.2 (Error pattern).
Error patterns are formalized as integer strings A of length p that define the number of errors for each piece. Its weight is the number of total errors $\sum_{i=1}^{p} A[i]$.

Definition 3.5.3 (Error pattern coverage).
A search S *covers* an error pattern if and only if

$$\forall 1 \leq i \leq p : L[i] \leq \sum_{j=1}^{i} A[\pi[j]] \leq U[i] \ .$$

We adjusted the original definition of error pattern coverage by Kucherov et al., since their definition is more restrictive and does not allow to increase the minimum number of errors during the search of the last piece. In particular, they require $L[i+1] \leq \sum_{j=1}^{i} A[\pi[j]]$ and define $L[p+1] = 0$.

Definition 3.5.4 (e-mismatch search scheme).
An e-mismatch search scheme $\mathbb{S}$ is a set of search triplets covering all possible error patterns with up to e errors.

Table 3.1 lists the one and two error mismatch search schemes by Lam et al. using the original definition of error pattern coverage by Kucherov at al. While their described search strategies allow for exactly one respectively two errors, Kucherov extended them to allow for fewer errors as well. All of the aforementioned search strategies in this chapter (except for simple backtracking), as well as the search schemes in table 3.1 have error patterns that are covered by multiple searches.

Strategy	Search scheme	Error patterns
$\mathbb{S}_{Lam,1}$	$(12, 00, 01)$	00, 01
	$(21, 00, 01)$	00, 10
$\mathbb{S}_{Lam,2}$	$(123, 000, 022)$	000, 001, 010, 011, 002, 020
	$(321, 000, 012)$	000, 100, 010, 110, 200
	$(231, 001, 012)$	001, 100, 101

Table 3.1: Formalized search schemes based on Lam et al.

Definition 3.5.5 (Disjoint searches).
We call the searches of a search scheme disjoint, if no error pattern is covered by more than one search.

Using the adjusted definition of error pattern coverage, we can formulate more efficient search schemes, e.g., $\mathbb{S}_{Lam',1} = \{(12, 00, 01), (21, 01, 01)\}$ which reduces the number of edges by one and covers each error pattern only once. The number of edges saved increases with more errors and a higher lower bound in the last piece. As we

will see later the search scheme $\mathbb{S}_{Lam,2}$ formulated by Kucherov et al. is not optimal either.

All of the strategies we presented in the previous sections can be formalized as searches and search schemes.

Observation 3.5.1 (Simple backtracking search schemes).
The simple backtracking approach with up to e mismatches is simply one search with one piece allowing between 0 and e errors: $\mathbb{S}_{simple} = \{(1, 0, e)\}$.

Observation 3.5.2 (Pigeonhole search schemes).
The pigeonhole search strategy with e errors can be represented by a search scheme with $e + 1$ searches and $e + 1$ pieces. Each search starts with a different piece while all searches have the lower and upper bounds $L = 00...0$ and $U = 0ee...e$.

Observation 3.5.3 (01^*0 search schemes).
01^*0 seeds can be formalized as search schemes such that each pair of error-free pieces is defined as a separate search. Searches with the same first error-free block can also be merged into a single search. The L strings are merged by selecting the minimum at each position, the U strings by selecting the maximum at each position, i.e., $L = 00...0$ and $U = 01ee...e$. Since there is only one search starting at P_{p-1}, no merging is necessary, hence $U = 00ee...e$. This reduces the number of searches from $\binom{e+2}{2}$ to $e + 1$.

Example 3.5.1 (01^*0 search schemes).
Let $e = 2$ and consider the 01^*0 seeds whose left error-free piece is P_1, i.e., the error patterns $00xx$, $010x$ and 0110 (where x can be any number of the remaining errors). They can be represented by the searches $(1234, 0000, 0022)$, $(1234, 0111, 0112)$ and $(1234, 0122, 0122)$. These three searches can be merged into a single search $(1234, 0000, 0122)$. All 01^*0 search schemes with merged searches for up to 4 errors are listed in the appendix in table A.4.

To compare the efficiency of different search schemes, the number of nodes for backtracking in an index can be computed using dynamic programming. This is done analogously to counting the edges for simple backtracking in equation 3.2. The number of nodes for a sequence of length k of a single search can be computed by equation 3.5 given $\overline{L}$ and $\overline{U}$. These integer strings are derived from L and U as follows. Let π be the order of pieces, $X[1..p]$ the length of each piece and L and U the lower and upper bounds. Each character $L[i]$ and $U[i]$ is then replaced by a run of the same character of length $X[\pi[i]]$, e.g., for $L = 0012$ with pieces of length 2 each, $\overline{L} = 00001122$ is derived.

The number of nodes is computed by equation 3.5. $n_{k,e}$ counts the number of prefixes of length k with exactly e mismatches, i.e., the number of leaves in the backtracking tree of this prefix. If we sum up the number of leaves for each prefix of length k and each allowed number of errors, we count the number of nodes in the entire backtracking tree.

$$n_k = \sum_{e=\overline{L}[k]}^{\overline{U}[k]} n_{k,e}$$

$$n_{k,e} = \begin{cases} n_{k-1,e} + (\sigma - 1)n_{k-1,e-1} & , k > 0 \wedge \overline{L}[k] \leq e \leq \overline{U}[k] \\ 1 & , k = 0 \wedge e = 0 \\ 0 & , \text{otherwise} \end{cases} \quad (3.5)$$

Kucherov et al. show two improvements over the search strategies by Lam et al. It can be beneficial to split the sequence into more than $e+1$ pieces and it might be more efficient if not all pieces are of equal length. The backtracking tree of the first search of $\mathbb{S}_{Lam,2}$ has significantly more nodes than the other searches (which can already be observed by looking at the U string: the second piece allows up to

two errors, while the other searches only allow for up to one error). Hence, it can be beneficial to decrease the size of the second piece. In their paper they show that up to some point, enlarging the first and shrinking the second piece leads to a larger decrease of nodes in the first search than an increase of nodes in the other two searches. We will discuss these two improvements in the experiments in section 3.6.3.

All search schemes suggested by Kucherov et al. were computed by a greedy algorithm. It tries to minimize the *critical* U string, which is the lexicographically largest U string among all searches of a search scheme. They solved it for $P = K+1$ and $P = K+2$ pieces. While this approach computes search schemes that turn out to be quite efficient, they are in general not optimal or do not have disjoint searches.

This leads to the question how search schemes can be computed that are optimal in the number of nodes, even for more than $K + 2$ pieces, and how to retrieve optimal search schemes that have disjoint searches, i.e., each error pattern is covered by exactly one search.

3.6 Optimum Search Schemes

Kucherov et al. made an important step to formalize search schemes and allow comparing different search strategies by the number of nodes. By introducing an integer linear program (ILP) [Kianfar et al., 2018], we try to improve the performance of approximate string matching further using search schemes that are optimal, i.e., minimal in the number of nodes and also more practical, e.g., have disjoint searches to avoid filtration of duplicate matches.

3.6.1 Definition

The underlying problem that we want to solve is the optimal e-mismatch search scheme problem.

Definition 3.6.1 (Optimal e-mismatch search scheme problem). What is the e-mismatch search scheme that minimizes the number of search steps in an index while covering all possible error patterns by at least one search?

We will formulate an integer linear program, that will solve this problem by minimizing the number of nodes across all searches. An ILP is a mathematical optimization problem in the form of equation 3.6. A given objective function has to be maximized while not violating any given constraint. The variables are required to be integers. To solve a minimizing problem or give lower bounds in the constraints, the expressions can be negated. A Boolean variable y_i can be introduced as an integer variable with the additional constraints $0 \leq y_i$ and $y_i \leq 1$.

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x_i \in \mathbb{Z} \end{aligned} \tag{3.6}$$

We will introduce an ILP solving the optimal e-mismatch search scheme problem. Given an upper bound of the number of searches, the pattern length and the number of pieces the pattern is partitioned into, the search scheme with the fewest number of nodes is computed.

3.6.2 ILP Formulation

We introduce the following input parameters of the ILP:

- $\overline{S}$ (maximum number of searches)
- R (length of the pattern)
- K (maximum number of errors allowed)
- P (number of pieces)
- σ (alphabet size)

The objective function is the number of nodes across all searches, which is to minimize. We will later show how to compute $n_{s,l,d}$ in the ILP.

$$\min \sum_{s=1}^{\overline{S}} \sum_{l=1}^{R} \sum_{d=0}^{K} n_{s,l,d} \tag{3.7}$$

To represent the π string of a search, we introduce decision variables. $x_{s,i,j} = 1$ if and only if search s searches the jth piece of the pattern at iteration i. We have to ensure that for each search every piece of the pattern is searched exactly once (i.e., a search cannot search the same piece twice and there cannot be two pieces at the same iteration of the same search). To simplify the notation of the subsequent constraints, we define $x_{s,i,0} = x_{s,i,P+1} = 0$ for all searches s and iterations i.

$$\begin{aligned}
&\sum_{i=1}^{P} x_{s,i,j} = 1 && \text{for all } s \text{ and } j \\
&\sum_{j=1}^{P} x_{s,i,j} = 1 && \text{for all } s \text{ and } i \\
&x_{s,i,j} \in \{0,1\} && \text{for all } s,\ i \text{ and } j
\end{aligned} \tag{3.8}$$

Next, we have to ensure that the connectivity property is fulfilled. $\sum_{h=1}^{i} x_{s,h,j}$ indicates whether piece j has been searched until the ith iteration. We sum up the absolute values of the differences of adjacent iterations:

$$\sum_{j=1}^{P} \left| \sum_{h=1}^{i} x_{s,h,j} - \sum_{h=1}^{i} x_{s,h,j-1} \right| = 2$$

For one piece the difference will be $+1$ and for another one -1, which are the first and last pieces searched. For the other pieces the difference will be zero. This equation can be formulated as a linear equation by introducing the binary auxiliary variables $t^{+}_{s,i,j}$ and $t^{-}_{s,i,j}$.

$$\begin{aligned}
&\sum_{h=1}^{i} x_{s,h,j} - \sum_{h=1}^{i} x_{s,h,j-1} = t^{+}_{s,i,j} - t^{-}_{s,i,j} \\
&\text{for all } s,\ i = 2, ..., P-1 \text{ and } j = 1, ..., P+1 \\
&\sum_{j=1}^{P+1} (t^{+}_{s,i,j} + t^{-}_{s,i,j}) = 2 \quad \text{for all } s,\ i = 2, ..., P-1 \\
&t^{+}_{s,i,j}, t^{-}_{s,i,j} \in \{0, 1\} \qquad \text{for all } s,\ i \text{ and } j
\end{aligned} \tag{3.9}$$

Now we have to consider the L and U strings of each search. We first enforce that both, L and U are non-decreasing.

$$\begin{aligned}
&L_{s,i} \leq L_{s,i+1} \quad \text{for all } s \text{ and } i = 1, ..., P-1 \\
&U_{s,i} \leq U_{s,i+1} \quad \text{for all } s \text{ and } i = 1, ..., P-1 \\
&L_{s,i}, U_{s,i} \geq 0 \quad \text{for all } s \text{ and } i \\
&L_{s,i}, U_{s,i} \in \mathbb{Z} \quad \text{for all } s \text{ and } i
\end{aligned} \tag{3.10}$$

Furthermore, we need to ensure that every error pattern is covered at least once. We precompute all possible error patterns up to e errors for P pieces. Error pattern q is represented by $a_{q,j}$ for $j = 1, 2, ..., P$ with $\sum_{j=0}^{P} a_{q,j} \leq K$ errors. We introduce a binary decision variable $\lambda_{q,s}$ that captures whether error pattern q is covered by search s. The first line of the equation enforces $\lambda_{q,s} = 1$ if the search s covers that error pattern.

$$
\begin{gathered}
L_{s,i} + K(\lambda_{q,s} - 1) \leq \sum_{h=1}^{i} \sum_{j=1}^{P} a_{q,j} x_{s,h,j} \leq U_{s,i} + K(1 - \lambda_{q,s}) \\
\text{for all } q,\ s \text{ and } i \\
\sum_{s=1}^{\overline{S}} \lambda_{q,s} \geq 1 \quad \text{for all } q \\
\lambda_{q,s} \in \{0, 1\} \text{ for all } q \text{ and } s
\end{gathered}
\tag{3.11}
$$

Finally, the computation of the nodes $n_{s,l,d}$ used in the objective function is enforced by the constraints of the ILP. The formula is taken from equation 3.5. $\overline{z}_{s,l,d}$ and $\underline{z}_{s,l,d}$ are binary variables that are both set to 1 by the first two constraints if and only if $L_{\lceil l/m \rceil} \leq d \leq U_{\lceil l/m \rceil}$ where l refers to the level in the backtracking tree. $n_{s,l,d}$ is then computed by the third constraint. If $\overline{z}_{s,l,d} = \underline{z}_{s,l,d} = 1$, it reduces to $n_{s,l,d} \geq n_{s,l-1,d} - (\sigma - 1)n_{s,l-1,d-1}$. Since the objective function is minimized, this computes $n_{s,l,d}$. The left-hand side of the third constraint $\binom{l}{d}(\sigma - 1)^d$ is a lower bound of the right-hand side not affecting the computation of $n_{s,l,d}$ in the recursive case and evaluates to 1 for $n_{s,0,0}$. The ILP only considers even partitions, i.e., each piece has the same length.

$$d - (L_{s,\lceil l/m \rceil} - m\lceil l/m \rceil + l) + 1 \leq (K+m)\underline{z}_{s,l,d}$$
$$\text{for all } s,\ l \text{ and } d$$

$$U_{s,\lceil l/m \rceil} + 1 - d \leq (K+1)\overline{z}_{s,l,d}$$
$$\text{for all } s,\ l \text{ and } d \tag{3.12}$$

$$\tbinom{l}{d}(\sigma-1)^d(\overline{z}_{s,l,d} + \underline{z}_{s,l,d} - 2) \leq n_{s,l,d} - n_{s,l-1,d} - (\sigma-1)n_{s,l-1,d-1}$$
$$\text{for all } s,\ l \text{ and } d$$

$$\overline{z}_{s,l,d}, \underline{z}_{s,l,d} \in \{0,1\} \quad \text{for all } s,\ l \text{ and } d$$

$$n_{s,l,d} \geq 0 \quad \text{for all } s,\ l \text{ and } d$$

This completes the ILP. As experiments showed, solving this optimization problem even for small instances can be computationally expensive. Hence, we further improved upon it by eliminating symmetric solutions to speed up solving the ILP.

First of all each search scheme can be transformed into a search scheme with the same number of nodes by reversing the π string of all searches. Since for each search the last piece searched is always either piece 1 or piece P, we enforce that the first search must end with the last piece. Thus, we disallow the symmetric solution where all the π strings are in reversed order and the first search ends with the last piece.

$$x_{1,P,P} = 1 \tag{3.13}$$

Search schemes are defined as a set of searches, but in the ILP they can only be represented as a sequence of searches. Thus, the same search scheme can be represented by $\overline{S}!$ search schemes by simply reordering the searches. To address this issue, searches are sorted by the piece in the first iteration. This is achieved by enforcing that if

piece j is assigned in the first iteration of search s, the subsequent searches cannot have pieces 1 to $j-1$ assigned in the first iteration. If $x_{s,1,j} = 1$, the right-hand side is evaluated to 0, i.e., no piece $1, ..., j-1$ can be searched in the first iteration of the searches stored in larger indices $s+1, ..., \overline{S}$. If $x_{s,1,j} = 0$, the constraint does not impose any new restrictions that is not enforced by previous constraints already. The right-hand side evaluates to $\overline{S} - s$ which is an upper bound as the left-hand side sums up the decision variables of the first iteration of $\overline{S} - s$ searches.

$$\sum_{t=s+1}^{\overline{S}} \sum_{k=1}^{j-1} x_{t,1,k} \leq (\overline{S} - s)(1 - x_{s,1,j}) \quad \text{for all } s \text{ and } j = 2, ..., P \tag{3.14}$$

The number of searches $\overline{S}$ chosen as input is an upper bound. If the optimal search scheme with up to $\overline{S}$ searches has actually less searches, the ILP will make those searches invalid, i.e., for some i: $L_{s,i} > U_{s,i}$.

This ILP is already very powerful as it can solve the optimization problem for a arbitrary number of pieces and errors for which Kucherov et al. only solved some instances using a greedy algorithm.

Nonetheless, computing optimal search schemes have shown that the running time for slightly larger input parameters can be quite slow, e.g., for more than 3 errors, it is not suitable to allow more than $\overline{S} = 4$ searches. The ILP can be improved further by optimizing search schemes for uneven partitions or eliminate the input parameter P to determine the optimal number of pieces when solving the ILP. Before that, more research has to be done to improve upon the running time of solving the ILP.

The ILP can be modified to enforce a minimum number of errors by omitting error patterns that have less errors. While this does not significantly improve the number of nodes or the running time as the dominating factor are the U strings, it is of practical interest for read

mappers. Read mappers come with different search modes, such as *best*, *all*, *all-best* and *strata* modes. The *all* mode finds all approximate matches with up to the maximum number of errors allowed. This is the case that we considered throughout this chapter. The *best* and *all-best* modes find a single respectively all occurrences with the lowest number of errors possible. The *x-strata* mode finds all approximate matches with $b + x \leq K$ errors where b the number of errors of a best match and K is still the maximum number of errors allowed.

Search schemes with lower error and upper error bounds can be used by these additional search modes to iteratively increase the maximum number of errors allowed until a best match is found. For the *x-strata* mode one iteratively increases the maximum number of errors allowed until a best match is found. Afterwards one performs an additional search allowing up to $b + x$ errors. During these iterations error patterns can be neglected that have been searched for in an earlier iteration.

3.6.3 Experiments

We formulated the ILP in C++ using the CPLEX 12.7.1 solver [CPLEX, 2009] and solved instances with up to $\overline{S} = 3$ searches and $K + 1$, $K + 2$ and $K + 3$ pieces for read lengths of $R = 101$ and an alphabet size of $\sigma = 4$. The computed optimal search schemes are listed in table A.3 in the appendix.

Observation 3.6.1.
All searches of optimal search schemes computed in table A.3 are disjoint. No error pattern is covered multiple times.

While it is uncertain whether this is always the case, the ILP can be changed to enforce this in the third constraint in equation 3.11 by enforcing that each error pattern is not covered at least once, but exactly once.

	Hamming distance				Edit distance			
K	1	2	3	4	1	2	3	4
Backtracking	15.55	15.60	11.62	6.86	4.12	11.15	2.26	3.67
Pigeonhole	**8.00**	13.94	14.68	11.15	**2.09**	9.98	2.88	6.06
01*0 pairs	8.88	9.75	9.10	6.59	2.31	6.90	1.77	3.57
01*0 merged	8.82	11.92	12.19	8.98	2.30	8.45	2.37	4.83
Kucherov $K+1$		9.38	8.76	6.56		6.64	1.70	3.47
Kucherov $K+2$		9.85	7.22	7.22		6.94	1.39	3.85
OSS_3 $K+1$	**8.00**	8.92	6.78	4.06	**2.09**	6.31	1.29	2.13
OSS_3 $K+2$	8.92	**8.54**	**6.51**	**3.92**	2.33	**6.02**	**1.24**	**2.05**
OSS_3 $K+3$	**8.00**	**8.35**	**6.40**	**3.88**	**2.09**	**5.89**	**1.22**	**2.03**
OSS_4 $K+2$			**6.50**				**1.24**	
MAN_{best}				**3.91**				**2.05**
Scale	10^3	10^5	10^7	10^9	10^4	10^6	10^9	10^{11}

Table 3.2: Number of nodes for different search strategies formulated as search schemes and optimum search schemes for Hamming and Edit distance for read length $R = 101$ and $\sigma = 4$. $K + x$ indicates the number of pieces of search schemes.

Table 3.2 compares the optimal search schemes to the aforementioned search strategies: simple backtracking, the pigeonhole search strategy, 01*0 seeds by Vroland et al. and the search schemes by Kucherov et al. The 01*0 seeds are formulated as 01*0 *pairs* where each search covers a pair of error-free pieces representing the seed, and 01*0 *merged* where all searches beginning with the same error-free piece are merged into one (see table A.4). The search scheme by Kucherov et al. for $K = 2$ and $P = 3$ is close to the one suggested by Lam et al., which also allows for fewer errors (see table A.1), but is not optimal.

Most of the non-trivial search schemes have in common that each of their searches start with an error-free piece. This is not true for the optimal search schemes for $K = 3$ and $K = 4$ though. They

contain a search that allows already for up to 2 respectively 3 errors in the first iteration. Hence, we computed an optimal search scheme for $K = 3$ with $\overline{S} = 4$ searches and $P = 5$ pieces, which happened to have zero errors in each first iteration. For $K = 4$ the ILP solver did not finish within several days for more than $\overline{S} > 3$ searches. Instead, we manually improved upon the optimal search scheme for $K = 4$ and $P = 6$ with 3 searches and recursively divided each search with errors in the first piece by multiple searches with a lexicographically smaller U string until the first piece of each search was error-free. This result is most likely not optimal with respect to the number of nodes, but as we will see later, performs much better in practice. We refer to the optimal search scheme with 4 searches as OSS_4 and to the manually constructed search scheme for $K = 4$ as MAN_{best} (see table A.2). We also ran the ILP for $K = 1$ and $K = 2$ for different P and up to 4 searches. Each of them returned the same search scheme as OSS_3, i.e., those searches were already optimal for up to 4 searches.

Interestingly, the simple backtracking approach is not the worst strategy, as for $K > 2$ the pigeonhole search strategy produces significantly more nodes across all searches. 01^*0 seeds are an improvement over both approaches. Despite 01^*0 *pairs* having significantly more searches than 01^*0 *merged*, their backtracking trees have fewer nodes. The lexicographically smaller U string of the searches have a larger reduction in nodes than the overhead by performing more searches.

The table confirms the observation by Kucherov at al, that $K + 2$ pieces can sometimes be superior to $K + 1$ pieces. This is true for the search schemes by Kucherov as well as the optimal search schemes. Thanks to the ILP we were able to compute optimal search schemes for $P = K + 3$, which are even more efficient than all the other search schemes with respect to the number of nodes.

Whereas it is not surprising that optimal search schemes do not improve for $K = 1$ over the pigeonhole search strategy, we observe significant improvements for more errors to trivial approaches and even to advanced approaches such as 01^*0 seeds. For $K = 4$ we were

able to improve the search schemes by Kucherov et al. by more than 40%.

Even though the ILP is based on Hamming distance and the computed search schemes might not be optimal for Edit distance, the search schemes can be used for an Edit distance based search as well with similar improvements compared to Hamming distance.

R and σ have been chosen to represent typical sizes used for mapping Illumina reads. For larger common read lengths we obtained similar results. We believe that even for other read sizes and alphabet sizes, the computed search schemes might still be optimal. Nonetheless, this remains an open research question.

Uneven partitions of optimal search schemes only achieved an improvement of up to 5% in the number of nodes and an even smaller improvement with respect to the running time in practice. The best partitioning of an optimal search scheme was found by brute force. Due to this neglectable improvement, we do not cover uneven partitions for our experiments and benchmarks. Instead, we recommend for further research to include uneven partitioning into the ILP, since different partitioning could lead to different optimal search schemes and thus might achieve better improvements.

3.7 Benchmarks

To compare the performance of different search strategies in practice, we indexed GRCh38 (build 38 of the human genome by the Genome Reference Consortium) and searched 100 000 reads sampled from a whole genome sequencing experiment[1] using Illumina HiSeq 2500. The paired end reads of length 202 were truncated to 101 base pairs to obtain single end reads. All reads and their reverse complement were searched and located in the genome to be more comparable to read mapping tools. Since every read mapper handles N bases a

[1] Experiment ERX1959065 on the Sequence Read Archive

bit differently, we replaced them randomly by A, C, G or T in the reads and in the genome prior to indexing. Search schemes used a bidirectional FM index based on EPR dictionaries and a sampled suffix array with text order sampling and a sampling rate of 10. The benchmarks were run on the same computer with the same setup as described in section 2.4.8.

Strategy	$K = 1$	$K = 2$	$K = 3$	$K = 4$
Backtracking	22.8s	276s	42m 32s	5h 5m
Pigeonhole	**7.5s**	28.6s	2m 9s	8m 28s
01*0 pairs	8.3s	24.7s	1m 13s	3m 23s
01*0 merged	**7.4s**	22.8s	1m 19s	4m 17s
Kucherov $K + 1$		22.2s	**56s**	**2m 51s**
Kucherov $K + 2$		25.0s	**56s**	**3m 12s**
OSS_3 $K + 1$	**7.4s**	**20.9s**	1m 20s	6m 56s
OSS_3 $K + 2$	8.2s	**21.0s**	1m 17s	7m 7s
OSS_3 $K + 3$	7.8s	**21.3s**	1m 20s	48m 42s
OSS_4 $K + 2$			**57s**	
MAN_{best}				**3m 1s**
01*0 bidir.	7.5s	21.4s	1m 3s	2m 56s
Bowtie	28s	76s	3m 16s	N/A
Reads mapped	91.2 %	93.8 %	94.7 %	95.2 %

Table 3.3: Running times of different search strategies formulated as search schemes using Hamming distance. Additionally comparing to the read mapper Bowtie and 01*0 seeds implemented without search schemes. 78.4% of the reads could be mapped without errors to at least one strand. A timeout of 9 hours was set for all runs.

Table 3.3 compares all the aforementioned search strategies that can be formalized as search schemes using Hamming distance. Fur-

thermore we compare them to a C++ implementation of 01*0 seeds in a bidirectional FM index[2]. This is not the original implementation by Vroland et al., since the original one does not support Hamming distance. We also benchmark against a widely used read mapper tool named Bowtie [Langmead et al., 2009] that supports searching and locating *all* possible occurrences with up to K mismatches using Hamming distance. Bowtie uses two unidirectional FM indices on the original genome and the reversed genome. To allow for mismatches, they perform backtracking in the index. Since the two unidirectional indices are not synchronized, they cannot switch directions during the search and thus can only speed up backtracking for $K = 1$ using the pigeonhole search strategy. To find all approximate matches, we ran Bowtie with the options −v <K> -a -t (more than 3 errors are not supported). The best running times of search schemes for each K are highlighted in the table. Since the follow-up version Bowtie 2 [Langmead and Salzberg, 2012a] is significantly slower for *all* mapping (i.e., not terminating within 12 hours), we exclude it from our benchmarks.

Except for the pigeonhole search strategy, which performs significantly better than the comparison of node counts would suggest, the results show a correlation between the number of nodes and the performance in practice. The improvements of OSS_4 and MAN_{best} over the optimal search schemes with fewer searches confirm that it is beneficial to enforce an error-free piece at the beginning of each search. It might be surprising that for 4 errors the optimal search schemes with $K + 3$ pieces are several factors slower than for $K + 2$ pieces. Taking a look at table A.3 in the appendix this seems reasonable. While most of their searches are similar, the first search for $K + 3$ starts with up to 3 errors in the very first piece where the corresponding search for $K + 2$ pieces starts with 2 errors in the first piece. At the same time the pieces of $K + 3$ are several characters

[2] `https://github.com/7/bav`, commit id 8b134d3

shorter, since the read is partitioned into more pieces. To improve the search scheme one would have to allow for more than 3 searches.

It seems worthwhile to improve the ILP further to solve harder instances with more searches and find even better performing search schemes. Since MAN_{best} was generated manually trying to enforce an error-free piece in the first iteration, we are optimistic that with an even stronger ILP we might outperform the non-optimal search schemes by Kucherov et al. As for now, optimal search schemes respectively MAN_{best} seem to perform similar as the search schemes by Kucherov et al.

Similar results were obtained for Edit distance for $K = 1$ and $K = 2$, shown in table 3.4. Surprisingly optimum search schemes and MAN_{best} perform significantly better than other search schemes, many of them did not even terminate within 9 hours.

We also compared the search schemes to Bwolo (01*0 seeds), the original implementation by Vroland et al., as well as the two read mappers BWA [Li and Durbin, 2009b] (with the options aln -N -n <K> -i 0 -l 101 -k <K>) and Yara [Siragusa, 2015] (with the options –e <K> -s <K> -y full) in *all* mapping modes.

For Edit distance we can see that the search schemes by Kucherov et al. perform worse than the optimum search schemes whereas they were evenly efficient for Hamming distance. This supports our hypothesis that the optimal search schemes for Hamming distance do not have to be optimal for Edit distance and vice versa. Even more importantly, one cannot in general infer the performance of a search scheme with respect to a distance metric from the performance based on other metrics.

It is remarkable that Bwolo achieved a running time significantly faster than any of the search schemes and was even faster than the bidirectional 01*0 seed algorithm. The speedup can be explained mainly by three improvements of Bwolo. Since it uses only a unidirectional FM index, after finding a seed and extending it to the right in the index, it has to locate the preliminary matches in the text

Strategy	$K = 1$	$K = 2$	$K = 3$	$K = 4$
Backtracking	43.6s	21m 9s	7h 40m	N/A
Pigeonhole	**11.0s**	3m 52s	59m 29s	N/A
01*0 pairs	11.7s	2m 43s	37m 16s	N/A
01*0 merged	**11.0s**	2m 41s	38m 44s	N/A
Kucherov $K + 1$		2m 23s	25m 23s	7h 29m
Kucherov $K + 2$		2m 33s	24m 58s	8h 57m
OSS_3 $K + 1$	**10.8s**	**2m 3s**	**22m 20s**	**4h 56m**
OSS_3 $K + 2$	11.9s	**1m 56s**	**22m 14s**	5h 5m
OSS_3 $K + 3$	11.4s	**1m 58s**	**22m 20s**	N/A
OSS_4 $K + 2$			**21m 31s**	
MAN_{best}				**4h 44m**
01*0 bidir.	11.0s	2m 33s	34m 29s	N/A
Bwolo	1m 13s	2m 26s	5m 46s	18m 18s
BWA	15.9s	2m 52s	31m 56s	5h 21m
Yara	21s	1m 53s	3m 44s	14m 38s
Reads mapped	91.8 %	94.7 %	95.8 %	96.4 %

Table 3.4: Running times of different search strategies formulated as search schemes using Edit distance. Additionally comparing to the read mappers BWA and Yara, as well as Bwolo, the original implementation of 01*0 seeds by Vroland et al. and 01*0 seeds in a bidirectional FM index. A timeout of 9 hours was set for all runs.

and perform a verification step, i.e., verify the left unmapped part in the text with the remaining number of errors. This in-text verification can be significantly faster than a search in an index. From table 2.2 we can conclude that a single search step in an FM index (i.e., an edge in the backtracking tree) costs at least 100 clock cycles. When during the search of a read the number of potential matches already reduces to only a few (which is the case for many reads that do not fall into repeat-rich regions), it can be much faster to verify each potential match instead of finishing the search in the subtree of the backtracking tree. This can be done by simply comparing the remaining characters by a linear scan over the remaining part of the read for Hamming distance or performing a verification for Edit distance using dynamic programming algorithms or the Myers bit vector algorithm [Myers, 1999].

The second improvement from Bwolo stems from the improved backtracking for Edit distance. While search schemes allow at any point for any kind of error, Bwolo reduces the amount of possible alignments. For example, there are no insertions allowed at the beginning or end of a sequence. Insertions followed by a deletion and vice versa are excluded as they can be represented as a single substitution leading to a better alignment. For compatibility with other approximate string matching algorithms in SeqAn2 we did not change the behavior and searched for every possible alignment.

Finally, Bwolo tries to filter duplicate occurrences as early as possible. When searching for 01^*0 seeds it performs backtracking in a combination of breadth-first and depth-first search manner. When searching a single block with one error backtracking is performed as usual in a depth-first search manner. At the end of searching a block, all suffix array ranges of the searched prefix of the 01^*0 seeds are stored in a hash map and the search with the next block continues again for suffix array ranges one by one using backtracking in a depth-first search fashion. The hash map filters the occurrences such that no suffix array range with respect to the length of the matched

prefix is considered twice when processing the next block. To put it differently, the position of an error in the block does not matter as long as the searched sequence in the index is identical. Hence, the same occurrences are not processed multiple times even though they might stem from different alignments.

The effect of in-text verification can also be observed on the read mappers: Yara works similarly to Bwolo. A seeding phase is performed in a unidirectional FM index followed by locating the matches and verifying them using the Myers bit vector algorithm. It is slightly faster than Bwolo. BWA performs the entire search in a unidirectional FM index using backtracking. Consequently it is slower than the optimal search schemes and MAN_{best}. In section 3.8 we will investigate the effect of in-text verification in combination with search schemes further.

We also compared the *strata* mapping mode of Yara to optimum search schemes. Table 3.5 shows the running time of Yara in x-strata mode (with the options –e <K> -sc <x> -y full) for $x \in \{0, 1\}$. Optimal search schemes were computed that enforce a minimum number of errors greater than zero. We denote such an optimal search scheme as $OSS_{\text{minK},K}$.

For 0-strata, which is equivalent to *all-best* mapping each read is searched without any errors first. Only if no occurrence was found, it is then searched with $OSS_{1,1}$. This is repeated with $OSS_{K',K'}$ until the read eventually matches or the maximum number of errors allowed is reached, i.e., $K' = K$.

For 1-strata mode this becomes a bit more tricky. The read is first searched using $OSS_{0,1}$. If there is match with zero errors, the mapping is continued with the next read. If only occurrences with 1 error are found, one also has to find matches with two errors using $OSS_{2,2}$ (since we search all occurrences with $b + 1$ errors where b is the number of errors of a best match). If during the search of $OSS_{0,1}$ no match was found and $K > 1$, one continues with $OSS_{2,\min(3,K)}$. Again, this is repeated until a read is eventually found and has been

searched with one error more than its best match or the maximum number of errors allowed is reached.

Strategy	$K=0$	$K=1$	$K=2$	$K=3$	$K=4$
OSS 0-strata	2.1s	2.9s	4.1s	11.2s	98.5s
Yara 0-strata	3.7s	4.7s	11.9s	19.3s	64.1s
OSS 1-strata		10.5s	16.1s	28.4s	144.3s
Yara 1-strata		21.0s	27.3s	43.0s	111.6s

Table 3.5: Yara and optimum search schemes compared in strata mode using Edit distance. For zero errors an index-based search is performed, for one and two errors OSS_3 $K+1$ respectively OSS_3 $K+2$ was chosen, for three and four errors OSS_4 and MAN_{best}. The search schemes have a lexicographically larger L-string to account for the minimum number of errors.

When we computed $OSS_{min,max}$ using the ILP, we chose the parameters P and S by selecting the best search schemes from tables 3.3 and 3.4, i.e., $P = K+1$ for one error, $P = K+2$ for more than one error. For three errors we set a maximum of $\overline{S} = 4$ searches. For four errors we did not compute an optimum search scheme, but modified MAN_{best}. All these optimum search schemes with $minK > 0$ are identical to the optimum search schemes with $minK = 0$ except that for each search the minimum number of errors in the last block is set to $minK$, i.e., $L[P] = minK$. Hence, we updated MAN_{best} accordingly. This yields disjoint searches with a minimum number of errors for each error pattern.

In strata mode the search schemes perform significantly better than Yara for up to 3 errors. Since optimum search schemes in the all mapping mode are significantly faster for 1 error and about as fast as Yara for 2 errors, the strata approach can maintain its lead for up to 3 errors.

From now on we will always consider the best performing search schemes from tables 3.3 and 3.4 and refer to them as OSS^*, i.e., for one error OSS_3 $K+1$, for two errors OSS_3 $K+2$ and for three and four errors OSS_4 respectively MAN_{best}.

We emphasize that optimum search schemes are not supposed to replace the entire read mapping process, but improve the speed of the seeding phase or increase the number of errors that are suitable for an index-based approximate search. Many read mapping tools still perform seeding in unidirectional or bidirectional string indices using exact string matching or simple backtracking. Our experiments have shown that optimal search schemes have significantly fewer nodes than other approaches or search schemes. There is still room for improvements as we have manually found search schemes with larger bounds that we were not able to compute with the current ILP. The practical benchmarks confirm that optimal search schemes are also faster in practice. For Hamming distance and $K \in \{3, 4\}$ it seems that better optimal search schemes can still be found.

3.8 In-Text Verification

As it can be concluded from table 2.2, a single search step, i.e., going down an edge in the backtracking tree in an FM index based on EPR dictionaries costs at least 100 clock cycles, which gets even more expensive for wavelet tree based indices. If only a couple of potential matches are left while going down the backtracking tree, it can be beneficial to locate these potential matches early and verify them in the text, i.e., check whether the partially searched pattern at its position in the text can be extended to the entire pattern with regard to the remaining number of errors.

The break-even point at which it pays off in a node to leave the index-based search and perform an in-text verification is hard to compute as it depends on a number of factors, of which an important one

is unknown. Given the node of a backtracking tree, we know the number of occurrences from the size of the suffix array range, but we do not know the number of matches eventually remaining after processing the entire subtree. This can be estimated, but is prone to errors as the text as well as the patterns of sequencing data do not have an underlying random distribution of characters. Other factors are known such as the number of remaining errors, the distance metric, the number of characters left to be matched, the alphabet size and the suffix array sampling rate. Assuming text order sampling with a sampling rate of η, locating an occurrence requires an expected number of $\eta/2$ LF mappings which are equally expensive as a unidirectional search step. The overhead of locating false positive matches (i.e., an occurrence is discarded after the verification) has to be taken into account.

Furthermore it has to be considered whether a search query is performed to only count occurrences or whether they also have to be located. For search queries, in-text verification requires locating all potential matches while not even the verified occurrences are needed in the end which means an additional overhead compared to pure index-based searching has to be considered.

While this topic can be addressed in an entire chapter and analyzed from a theoretical point of view by estimating the number of true positive matches in a subtree of the backtracking tree, we will show pitfalls and benefits of in-text verification for index-based approximate string matching using optimum search schemes.

From the experiments in table 3.4 we have seen that in-text verification can lead to a significant speedup as achieved by Bwolo. Their implementation switches from an index-based search to in-text verification at a fixed point. Since they use a unidirectional FM index, the unmatched pattern left of the 01^*0 seed is verified, independent of factors such as the number of potential matches or remaining number of errors left. That this can be highly unfavorable is shown in table 3.6. We sampled 10 reads of length 101 from the human genome and

planted 10^4, 10^5 and 10^6 instances each with 3 errors in the genome, 50 % of them as their reverse complement. 33% of the errors were insertions or deletions. This leads to large suffix array ranges in the backtracking tree which are expensive to verify. We then searched the 10 sampled reads in the modified genome. It shows clearly that for repeat regions an early or even any in-text verification should be avoided. For reads that were found 10^5 times, Bwolo is 2.7x slower compared to optimum search schemes when allowing for up to 3 errors. On a genome with 10^6 planted repeats it is even 44x slower. On real Illumina reads Bwolo was up to 4 times faster than optimum search schemes, for $K = 4$ even up to 16 times.

Occurrences per read	Strategy	$K = 1$	$K = 2$	$K = 3$
10^4	OSS*	<1s	<1s	43s
	Bwolo	2s	4s	30s
10^5	OSS*	<1s	3s	44s
	Bwolo	3s	6s	120s
10^6	OSS*	<1s	3s	68s
	Bwolo	3s	29s	3000s

Table 3.6: Running time comparison of pure index-based searches with optimum search schemes and Bwolo, a hybrid approach of index-based search and in-text verification.

While there are no repeats of length 101 in the human genome with millions of occurrences, there are plant genomes that are more repetitive. Other applications than read mapping also perform approximate string matching on much shorter sequences and even higher error rates. An example is given in section 3.9.

For reads that are not highly repetitive in-text verification can be very advantageous. We implemented in-text verification for Ham-

ming distance in combination with optimum search schemes. As a condition to leave the index-based search we simply chose an upper bound on the size of the suffix array range. This already shows significant improvements, i.e., the optimum search schemes get about twice as fast. This approach is also immune against repetitive reads as shown in table 3.7 (b).

The experiment of section 3.7 is conducted again with OSS^* and in-text verification in table 3.7 (a). Our in-text verification is tested on the simulated repetitive data in table 3.7 (b). Again, we sampled 10 reads from the human genome and planted 10^6 approximate matches with 3 errors, this time only considering Hamming distance.

We chose the condition for switching from an index-based search to in-text verification (ITV) to be dependent on the size of the suffix array range as this should be immune to repetitive data as well. $\text{ITV}_{\text{occ}_i}$ leaves the index as soon as the suffix array range stores less than i occurrences. Additionally we compared it to $\text{ITV}_{\text{blocks}}$, which similarly to Bwolo leaves the index-based search at a certain depth of the backtracking tree. Here we chose the last block in each search to be verified in the text. ITV_{off} is the pure index-based search with OSS^* given as a reference from the benchmarks in the previous section.

Our benchmarks show that a simple in-text verification such as $\text{ITV}_{\text{occ}_{25}}$ that is robust to repeats already can achieve significant improvements. For Illumina reads OSS^* with in-text verification is between 1.6x and 2.1x faster than OSS^*. For repetitive reads it shows even speedups of up to 5.5x. The impact of the size of the suffix array range is only minor. While $\text{ITV}_{\text{occ}_{50}}$ leads to a significant increase in the number of verifications, it has almost no effect on the running time. We observed the same also for higher values such as 250 and 500. Verifying the last block of search schemes also improves the running time in both scenarios, but the speedups are significantly smaller. The disadvantage for repetitive data is not as big as in table 3.6, since we consider only Hamming distance which can be verified faster by a linear scan over the pattern than Edit distance by a

	$K=1$		$K=2$		$K=3$		$K=4$	
Strategy	Time	Verif.	Time	Verif.	Time	Verif.	Time	Verif.
ITV_{off}	7.4	0	21	0	57	0	181	0
$\text{ITV}_{\text{occ}_{25}}$	4.7	3.3	11	5.0	27	2.2	86	9.5
$\text{ITV}_{\text{occ}_{50}}$	4.7	4.2	11	6.3	28	2.9	89	12.7
$\text{ITV}_{\text{block}}$	6.4	92.6	18	9.6	48	1.6	158	2.4
Scale		10^5		10^6		10^7		10^7

(a) Mapping 100.000 Illumina reads of length 101 to GRCh38 (as in section 3.7).

	$K=1$		$K=2$		$K=3$		$K=4$	
Strategy	Time	Verif.	Time	Verif.	Time	Verif.	Time	Verif.
ITV_{off}	<1	0	1.5	0	19	0	61	0
$\text{ITV}_{\text{occ}_{25}}$	<1	0.06	1.7	2.1	8	7.0	11	1.2
$\text{ITV}_{\text{occ}_{50}}$	<1	0.1	1.9	4.2	8	7.1	11	1.3
$\text{ITV}_{\text{block}}$	1.7	175.5	2.7	17.9	13	5.4	33	1.1
Scale		10^4		10^5		10^6		10^7

(b) Mapping 10 reads of length 101 with 10^6 approximate occurrences in a modified version of GRCh38.

Table 3.7: Running time in seconds and the number of potential matches verified for OSS^* with different in-text verification conditions and errors for Hamming distance. ITV_{off} is equivalent to OSS^*. $\text{ITV}_{\text{occ}_i}$ starts in-text verification if the subtree has less than i potential matches, $\text{ITV}_{\text{block}}$ verifies that last block of OSS^* in the text.

banded dynamic programming algorithm as performed by Bwolo.

The running time of mapping does not correlate with the number of verifications performed in most cases. As described earlier a lot of factors such as the depth in the backtracking tree as well as the number of potential matches in relation to the expected number of matches has to be considered as well. This can be examined in great detail for developing better conditions for in-text verification to gain even higher speedups as well as an implementation for Edit distance. This is out of the scope of this thesis and we consider it as future research.

3.9 High Error Rates

Read mapping tools are not only used for mapping reads to reference genomes, but for all kind of sequence mappings. We will introduce an application requiring significantly higher error rates than conventional read mapping and show that optimum search schemes with in-text verification can be a suitable choice for high error rates in an index-based search.

In the past years, CRISPR/Cas9 has become a popular method for genome editing [Doudna and Charpentier, 2014]. CRISPR/Cas9 is originally found as an adaptive immune system in prokaryotes, i.e., bacteria and archaea. The Cas9 enzyme targets sequence-specific RNA as a defense mechanism against viruses or plasmids. To detect invasive RNA of viruses and plasmids, it uses so-called CRISPR sequences which store known RNA of such viruses. If the Cas9 nuclease detects complementary RNA to CRISPR sequences, it binds to the invasive RNA, cleaves this region and ultimately destroys it.

The process can be adapted for genome editing by designing synthetic CRISPR sequences to bind to certain locations in the genome where a gene is to be knocked out. These sequences are referred to as single guideRNA (sgRNA). The designed sgRNA and the Cas9 nucle-

ase are then delivered to the cell where the genome is cut. The cells DNA repair mechanisms will reassemble the two pieces of the genome leading to some errors at the cut site. This knocks out the gene in most cases. To deliver the Cas9 enzyme and sgRNA into the cell, one possible way is to attach them to viruses that will dock onto the hosts cell membrane and inject its viral DNA, including the sgRNA and Cas9 nuclease. While CRISPR/Cas9 is found in prokaryotes, this approach can also be applied to eukaryotes, e.g., human cells.

To knock out a certain gene, one has to choose a single guideRNA that is complementary to a part of that gene and can bind there. When using the Cas9 enzyme, the sgRNA needs to have a length of 23 base pairs. Furthermore the region must be flanked by a Protospacer Adjacent Motif (PAM) that allows the Cas9 nuclease to bind to the DNA and eventually cleave it. The PAM depends on the nuclease used. For Cas9 the region must by flanked by the 3-mer NGG where N can be any nucleotide. The computational challenge is to find guideRNA that flanks a PAM, but that also does not occur anywhere else in the genome to prevent binding and cleaving at a wrong location. Those other locations are called off-targets. For the human genome of over 3 billion base pairs a 23-mer is often not unique. Furthermore, sgRNA can also bind to locations that are only an approximate match. Hence, when searching for off-targets one has to allow for errors as well. In experiments it has been observed that single guideRNA can potentially bind with up to 8 substitutions [Tsai et al., 2015, Cameron et al., 2017]. However, not all of these off-targets become active off-targets, i.e., the Cas9 nuclease will not cleave at these loci.

Searching for off-targets is often performed with read mapping tools. Since they are not designed for high error rates and *all* mapping, we developed VARSCOT, a variant-aware detection and site-scoring tool to search for off-targets and use a predictive model to identify possible active off-targets [Wilson et al., 2018].

We searched 9 guideRNA and their reverse complement of length

23 in the human genome (GRCh38) with up to 8 mismatches while the last two bases had to be an exact match (NGG). Since computing optimal search schemes with up to 8 errors seemed to be infeasible with the current ILP, we split the single guideRNA into two pieces and searched each of them with up to 4 errors using OSS^*. An in-text verification is then performed on the other half of the RNA, also checking for an exact match of the PAM.

With the same setup as before it took 80 seconds per guideRNA and its reverse complement to search and locate all occurrences with up to 8 mismatches in the human genome. When executed in parallel, it takes less than 2 minutes to search all guideRNA on 9 threads. This is magnitudes faster than Elevation [Listgarten et al., 2018], the state-of-the-art pipeline for off-target detection and activity prediction. Elevation took 4 hours for searching a single read with up to 8 mismatches. Common read mappers either do not support high error rates as high as 33% or have infeasible running times.

This shows that even short sequences with high error rates can be searched quickly using optimum search schemes and in-text verification.

4 Mappability

4.1 Introduction

We will now cover a concept in sequence analysis that is similar to read mapping, and which again can be used for the read mapping process itself: mappability. Given a genomic sequence, we are interested in distinguishing unique and non-unique, i.e., repetitive regions in it. This concept is defined as genome or sequence mappability and was introduced by Koehler and Derrien [Koehler et al., 2010, Derrien et al., 2012].

Definition 4.1.1 ((k,e)-frequency and (k,e)-mappability). The (k,e)-frequency of a sequence T of length n counts for every single k-mer in T its number of occurrences in T with up to e errors. We denote the k-mer starting at position i as T_i. The values are stored in a frequency vector F of length $n-k+1$ such that

$$F[i] = |\{j \mid D(T_i, T_j) \leq e, 1 \leq j \leq n-k+1\}| \tag{4.1}$$

The inverse of it is called the (k,e)-mappability and stored in a mappability vector M with $M[i] = 1/F[i]$ for $1 \leq i \leq n-k+1$. D can be any distance metric on strings such as Hamming or Edit distance.

Again, we will for now only consider Hamming distance, but it can be applied to other distance metrics such as Edit distance as well. For applications in sequence analysis one often also considers the reverse strand of a sequence when computing the genome mappability.

Hence, $F[i]$ not only counts the occurrences of T_i in T, but also in its reverse complement. If not stated otherwise, we do not consider the reverse complement in our examples.

Example 4.1.1.
In figure 4.1 we give an example of the frequency vector for 4-mers with 0 and 1 error, denoted as F_0 and F_1. Consider the 4-mer $T_2 =$ TCTA. It occurs two times without errors (at positions 2 and 15), hence $F_0[2] = 2$. Since there is another 4-mer that has only one mismatch, namely $T_{10} =$ GCTA, the frequency value $F_1[2]$ is 3.

In practice, one usually uses the mappability instead of the frequency, since it is normalized and reduced to the interval $(0, 1]$. For reasons of clarity we will instead consider the frequency throughout this chapter.

In the next sections we will present the algorithm by Derrien et al. on how to compute F. The algorithm uses a heuristic to compute the frequency even for long repetitive genomes, i.e., some of the values are only approximated, but are expected to be close to the correct value. It can also be run in an exact mode at the expense of a longer running time.

We then introduce a new, unpublished algorithm to compute the mappability a magnitude faster without the use of any heuristics and compare its running time against the approximate and exact algorithm by Derrien et al.

Afterwards we will cover some of the applications of mappability. It can be used straightforward to examine the repetitiveness of genomes, even highly repetitive plant genomes. We further show how it can be incorporated into read mapping. As searching and aligning reads from repeat regions is the most expensive part in read mapping, and the user might not be interested in every single alignment to repeat regions, it is beneficial to detect repeat regions as soon as possible during the search and only map the reads to those locations that are rather unique.

i:	1	**2**	3	4	5	6	7	8	9	10	11	12	13	14	**15**	16	17	18
$T[i]$:	A	**T**	**C**	**T**	**A**	G	C	T	T	G	C	T	A	A	**T**	**C**	**T**	**A**
$F_0[i]$:	2	**2**	1	1	1	1	1	1	1	1	1	1	1	2	**2**			

(a) $(4, 0)$-frequency

i:	1	**2**	3	4	5	6	7	8	9	**10**	11	12	13	14	**15**	16	17	18
$T[i]$:	A	**T**	**C**	**T**	**A**	G	C	T	T	**G**	**C**	**T**	**A**	A	**T**	**C**	**T**	**A**
$F_1[i]$:	3	**3**	3	2	4	2	2	2	2	**4**	2	1	1	3	**3**			

(b) $(4, 1)$-frequency

Figure 4.1: (k, e)-frequency vectors F_e for $k = 4$ and $e \in \{0, 1\}$ on the same sequence. A frequency of 1 indicates that the k-mer starting at that position in the text is unique in the entire sequence without errors respectively with up to 1 mismatch.

Mappability can also be applied to multiple genomes, species or strains at once. Unique regions that belong to so-called marker genes among a set of strains allow distinguishing them by short k-mers. When sequencing an unknown sample one can search for such unique k-mers in the reads to determine the species or even the strain without assembling it or searching the sequenced data in a database of reference genomes.

4.2 An Inexact Algorithm

Derrien et al. propose an algorithm for computing the frequency of a sequence by searching each k-mer in T in an index and use a heuristic for computationally expensive repeat regions, i.e., some frequency values are approximated to speed up the computation (see algorithm 8). The frequency vector F is first initialized with zeros. In the next step the algorithm iterates over the text and searches each k-mer T_i with up to e mismatches in the text. The occurrences are counted and stored in $F[i]$. If the number of occurrences is greater than some threshold parameter t, the computed occurrences are located in the text. Let j be the position of such an occurrence in T. Since T_i has a sufficiently high frequency (i.e., $F[i] > t$) and $D(T_i, T_j) \leq e$, it is likely that T_j also has a high frequency as both share common approximate matches, hence $F[j]$ is set to the same value as $F[i]$.

Algorithm 8 Inexact algorithm to compute the (k, e)-frequency

```
1:  procedure INEXACT_FREQUENCY(T, k, e, t)
2:      F[1..|T| − k + 1] ← {0}
3:      for i = 1, ..., |F| do
4:          if F[i] = 0 then
5:              F[i] ← |P|
6:              P ← approximate matches with e errors
7:              if |P| > t then
8:                  for j ∈ P do
9:                      F[j] ← max(F[j], |P|)
10:     return F
```

k-mers that already have an approximate frequency value assigned will be skipped while iterating over the text to save time. If a position j is located multiple times as an approximate match of a repetitive k-mer, $F[j]$ is assigned the maximum frequency of all these k-mers to avoid underestimating the frequency value $F[j]$. This algorithm

is exact for $e = 0$. To turn off the heuristic for $e > 0$, the threshold only has to be set high enough, i.e., $t \geq n - k + 1$.

The frequency values are stored using 8 bit. Since much higher frequency values than 256 are not unusual, the frequency values are grouped to bins, i.e., each bin represents a continuous range of frequency values. The bins are represented as 8-bit numbers. Lower values represent small frequency values and rather small intervals. Large 8-bit numbers represent high frequency values that are grouped to larger intervals, since the difference of a frequency value of 2 or 3 is more significant than a value of 1002 or 1003. Figure 4.1 illustrates this at the binning of frequency values of C. elegans. The packing into bins depends on the data, but is not elaborated by the authors.

Bin:	0	1	...	6	7	...	12	13	...	18	19	...	255
Frequency:	1	2	...	7-8	9-10	...	24-28	29-35	...	81-97	120-146	...	MAX

Table 4.1: Bins to store the $(50, 2)$-frequency of C. elegans.

The authors show that their approximation leads to an equal distribution among under- and overestimated values for the $(50, 2)$-frequency of the C. elegans genome and chromosome 19 of the human genome with a threshold of $t = 6$ respectively $t = 7$. Their experiments on the human genome chromosome show that almost 90 % of the 50-mers with a frequency of 3 are correct, for 50-mers with frequency values between 8 and 12 only 75 % are correct (similar errors for C. elegans). This can be led back to an overestimation of rather unique k-mers.

The algorithm is part of the GEM (GEnome Multitool) suite that is based on an FM index.

4.3 A Fast and Exact Algorithm

We present an exact algorithm for computing the (k, e)-frequency and will benchmark it against the GEM algorithm by Derrien et al. in the exact and approximate mode. Similar to their algorithm we search every k-mer of T with errors in an index. With a few improvements we try to eliminate redundant computations as much as possible.

1st improvement First, we try to reduce redundant computations due to approximate string matching in an index. Hence, we choose a bidirectional index that allows the use of advanced approximate string matching algorithms such as optimum search schemes. In particular, we use a bidirectional FM index based on EPR dictionaries and search the k-mers with at most e errors using OSS*.

2nd improvement Second, adjacent k-mers in T are highly similar, since they have a large overlap. Hence, it should be avoided to search every k-mer separately. Consider the k-mers $T_j, T_{j+1}, \ldots, T_{j+s-1}$ for some integer $s \leq k$ which all share the common sequence $T[j + s - 1..j + k - 1]$. Since we already need to allow for up to e errors in their common sequence when searching each k-mer, this infix should only be searched once. Hence, we propose not to search the entire k-mer with optimum search schemes, but only this infix with up to e errors using OSS*.

After searching this infix using optimum search schemes, the number of occurrences for each k-mer $T_j, T_{j+1}, \ldots, T_{j+s-1}$ still needs to be retrieved. This can be done using simple backtracking to the left as well as to the right allowing for the remaining number of errors not spent in the search of the infix. Figure 4.2 (a) illustrates this approach. To reduce the number of redundant computations further, the set of overlapping k-mers is recursively divided into two equally sized sets of k-mers that each share a larger common overlap among each other. This overlap is then searched using simple backtracking

before the next recursive partitioning of k-mers. The recursion ends when a single k-mer is left and the number of occurrences can be reported and summed up, or no hits are found. The recursive extension is shown in figure 4.2 (b).

Note, that there are two recursions involved: subdividing the set of k-mers and simple backtracking in each recursion step. Hence, the same partitioning steps and backtracking steps have to be performed for each set of preliminary matches represented by suffix array ranges.

The question remains on how to choose s. The optimal value for s depends on a variety of factors: the length of k-mers, the number of errors, the considered distance metric and finally the data itself. Since searching the infix using optimum search schemes is more efficient than extending this infix with simple backtracking, the infix should be longer for larger number of errors. Similarly, if k grows, so should s. As a rule of thumb we choose

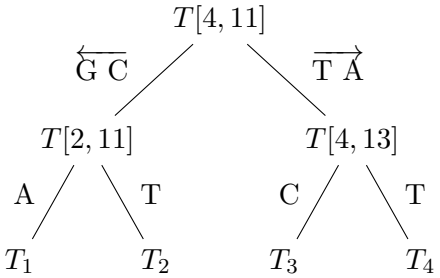
$$s = \begin{cases} \lfloor k \cdot 0.7 \rfloor & , e = 0 \\ \left\lfloor k \cdot \left(\text{clamp}\left(\frac{k}{100}, 0.3, 1.0\right) \cdot 0.7^e \right) \right\rfloor & , \text{otherwise} \end{cases} \tag{4.2}$$

where $\text{clamp}(v, l, r)$ returns v if it lies within the range, i.e., $l \leq v \leq r$, and returns l or r if it is less or greater. s is set in proportion to k: without any errors s is set to $0.7 \cdot k$. If we allow for errors, this percentage grows with larger k and shrinks with larger e. The formula is close to the optimal choice of s that we verified experimentally on the human genome and turns out to be a good choice for more repetitive genomes such as barley (hordeum vulgare) as well.

3rd improvement After reducing the number of redundant computations for approximate string matching as well as similar k-mers, searching and counting the same k-mer multiple times is eliminated. Especially for computationally expensive repeat regions some k-mers may occur multiple times in the genome, even without errors. During

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$T[i]$	A	G	C	C	G	T	A	C	A	A	G	T	A	T	...
T_1	A	G	C	C	G	T	A	C	A	A	G				
T_2		G	C	C	G	T	A	C	A	A	G	T			
T_3			C	C	G	T	A	C	A	A	G	T	A		
T_4				C	G	T	A	C	A	A	G	T	A	T	

(a) First, the common overlap (light gray) is searched using optimum search schemes. Second, the search of T_1 and T_2 is continued recursively by extending the previously identified approximate matches of the infix in the index by GC to the left (allowing for the remaining number of errors; medium gray). T_1 and T_2 are then retrieved separately by simple backtracking in the index by one character to the left and one character to the right (allowing for an error, if any left; dark gray). T_3 and T_4 are extended analogously in a recursive manner.



(b) The same strategy presented as a backtracking tree. It is traversed for each suffix array range reported by the search of the infix $T[4, 11]$ using optimum search schemes. Each edge also has to account for remaining errors, i.e., approximate string matching is performed using simple backtracking.

Figure 4.2: Searching s overlapping k-mers using optimum search schemes for the infix and extending it using simple backtracking. Illustrated for $k = 11$ and $s = 4$.

the previously described search, the suffix array range representing the exact matches of a k-mer is stored (using OSS* and simple backtracking there can only be one suffix array range without errors for each k-mer). After the approximate search the positions of the exact matches are located and the total number of occurrences (including errors) are assigned to the frequency vector at these positions. Since all these k-mers are identical, they share the same frequency value.

We observed in our experiments that this leads to longer runs of frequency values to be forwarded to positions with uncomputed frequency values. A small technical improvement can now be added: when the search of a set of s k-mers reaches such a run, it is not necessary to recompute these values and they can be skipped. If only some of the s positions have been computed already, we skip leading and trailing k-mers with computed frequency values, such that a smaller, but continuous block of k-mers can be searched with the aforementioned strategies. This leads to a larger infix to be searched by optimum search schemes and fewer extensions necessary using simple backtracking. It is still possible that some frequency values are recomputed in the reduced set of k-mers. To discard these k-mers one would have to split it into subsets and search the common infix of each subset separately. Since this introduces further redundant computations, we neglect it and accept that some frequency values might be recomputed.

The algorithm can also search each k-mer on the reverse strand. The previously explained steps are executed for each set of k-mers as well as their reverse complement which leads to a doubling of the running time. Some tools instead create the reverse complement of the genome during indexing which doubles the size of the index, but is only slightly slower than computing the frequency without considering the reverse strand. This is only suitable for genomes that are not too large, but since the running time gets only critical for large, repetitive genomes such as plants, we do not index the sequence of the reverse strand.

Our algorithm is implemented in a C++ stand-alone application called *GenMap* using SeqAn 2 and is available on GitHub[1]. Table A.5 in the appendix lists the available parameters and features for computing the frequencies. Besides computing the frequencies, we also offer a binary to convert the frequency vector into a mappability vector and into common formats such as *wig*-files that can be loaded into genome browsers to visualize the mappability.

Currently our implementation only supports Hamming distance. It can easily be extended to Edit distance, but one has to carefully consider the definition of frequency based on Edit distance. While for Hamming distance it is highly unlikely that a k-mer matching a position j in the text is also matching positions $j-1$ or $j+1$, this is not true for Edit distance. Multiple alignments to the same region are possible with slightly different starting and ending positions. Implementing the mappability for Edit distance in a straightforward manner would assign these k-mers a high frequency even though they might not belong to a repeat. Hence, the definition of frequency based on Edit distance has to be defined thoroughly depending on the application.

4.4 Benchmarks

For comparison we used the only available version (1.759 beta) of the GEM suite that included the mappability program. We did not reach the authors for a newer version including the mappability tool. Other available and newer versions do not offer this feature anymore. The approximation mode was run with $t = 7$ which Derrien et al. suggested for the human genome. For GenMap we computed the parameter s (number of overlapping k-mers) according to equation 4.2. Again, we run the benchmarks on the same computer with the same setup as described in section 2.4.8.

[1] `https://github.com/cpockrandt/mappability`

Table 4.2 (a) compares the running times on the human genome for computing the (k, e)-frequency for shorter k that are of interest for applications such as identifying marker genes, presented in section 4.6, whereas table 4.2 (b) shows typical instances used for applications in read mapping which we will present in section 4.5.

Tool	$(36, 0)$	$(24, 1)$	$(36, 2)$	$(50, 2)$	$(75, 3)$
GEM exact	5h 10m	N/A	N/A	N/A	N/A
GEM approx.	22m 44s	N/A	7h 11m	5h 50m	4h 26m
GenMap	3m 8s	23m 12s	1h 19m	42m 12s	1h 36m

(a) Instances are taken from [Derrien et al., 2012].

Tool	$(101, 0)$	$(101, 1)$	$(101, 2)$	$(101, 3)$	$(101, 4)$
GEM exact	44m 10s	7h 28m	7h 34m	7h 45m	8h 8m
GEM approx.	28m 8s	2h 40m	3h 17m	3h 31m	3h 49m
GenMap	2m 29s	7m 5s	16m 35s	49m 27s	3h 7m

(b) Comparing the running times for a typical Illumina read length with growing number of mismatches.

Table 4.2: Running times for computing the frequency of the human genome (GRCh38) using 16 threads. Timeouts of 1 day are represented as N/A.

For all computed instances, GenMap is faster than GEM. Compared to the approximate mode we are almost a magnitude faster for smaller number of errors, but for 4 errors the heuristic of GEM pays off and is almost as fast as our algorithm. Interestingly the increase of the running time of GEM in its exact mode gets smaller with more errors. For 101-mers with 1 to 4 errors the running time is always about 7 to 8 hours, nonetheless GenMap is still faster by a factor from 2.5 of up to 50 (4 and 1 errors).

Even in the exact mode where no advanced backtracking such as optimum search schemes is performed, our tool is faster by a factor of between 17 and 100 (for 101-mers and 36-mers). Especially for short k-mers with errors, GEM takes significantly longer, often does not even terminate within 24 hours and 16 threads.

The running times we measured for GEM approx. differs considerably from the running times published by the authors. Even when we ran it on a similar CPU with the same number of cores we were 2 to 5 times slower than their published benchmarks. One reason might be that the only available version of GEM with the mappability functionality was published as a beta version, however it was published in 2013, one year after the paper. Nonetheless, GenMap is still faster than the running times published by Derrien et al.

We are also significantly faster than GEM when computing the mappability of small genomes like D. melanogaster. Since smaller genomes are generally less challenging, we omit the benchmarks here. For the human genome the memory consumption of GenMap is about 10 GB (using a bidirectional FM index with EPR dictionaries and a suffix array sampling rate of 10), while GEM takes up 4.5 GB (using an unspecified FM index implementation with a suffix array sampling rate of 32).

In conclusion, GenMap is a magnitude faster than GEM in its exact mode, and still faster than GEM using its heuristics, while GenMap is always exact. For even up to 4 errors GenMap achieves a reasonable running time. This is due to the three techniques described in the previous section. Without using optimum search schemes and searching the infix of overlapping k-mers only once, our tool would achieve a comparable running time to GEM in its exact mode. Further improvements can be implemented which might speed up the algorithm even further, such as in-text verification. The benefit might be even greater than the benchmarks in the previous chapter suggest, since extending the infix is performed using simple backtracking which is not only considerably slower than optimum search schemes but also

has to be performed on multiple suffix array ranges separately, which are reported by the searches of OSS*.

GenMap is also suitable to compute the mappability of larger and more repetitive genomes than the human genome. We computed the $(50, 2)$-frequency of the barley genome (hordeum vulgare, [Mascher et al., 2017]) as it contains large amounts of repetitive DNA [Ranjekar et al., 1976].

Barley has 4.8 billion base pairs while the human genome has 3.2 billion base pairs. As expected the human genome has considerably more unique regions than the barley genome. When examining the $(50, 2)$-frequency of both genomes, 75.4 % of the k-mers were unique in the human genome, and only 26.4 % in the barley genome. There are 12.0 % (54.4 %), 7.6 % (42.1 %) and 4.8 % (25.6 %) k-mers in human DNA (resp. barley DNA) with at least 10, 100 and 1,000 occurrences. Computing the $(50, 2)$-frequency of barley on 16 threads took less than 1h 15m with GenMap and nearly a day with GEM using its heuristic with $t = 6$ (automatically chosen by GEM).

4.5 Read Mapping

As shown in the previous chapter, read mapping gets computationally more expensive with growing error rates for obvious reasons. Precisely, this is due to reads that match to many locations in the reference genome. When mapping those reads, subtrees in the backtracking tree are cut less frequently, hence the exponential growth in the number of edges is more significant.

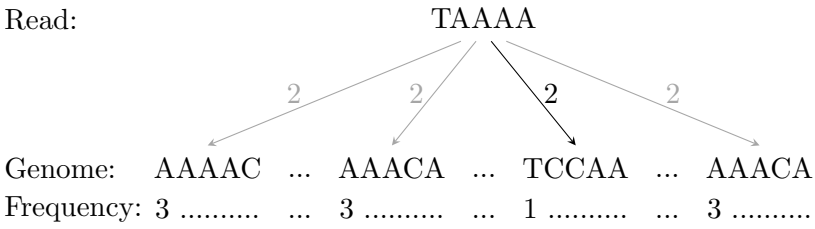
When a read mapper is run in *best* mapping mode, mapping reads to repeat regions is rather neglectable than for *all-best*, *strata* or even *all* mapping modes. For some applications the user might not be interested in mapping the read to all the repeat regions, but only to the less repetitive regions. This can be taken care of by performing repeat masking, i.e., masking repeats in the genome prior to indexing

the reference genome. Tools such as RepeatMasker [Smit et al., 1996] search the reference genome or any input sequence for repeat regions by comparing it to a database of known repeats and replace them by runs of N's. While there are also tools that identify repeats without using existing libraries [Edgar and Myers, 2005], so-called de-novo repeat annotation, they all have in common that the parameters for masking repeats have to be set prior to indexing and no adjustments are possible without masking and rebuilding the index again, another time consuming step.

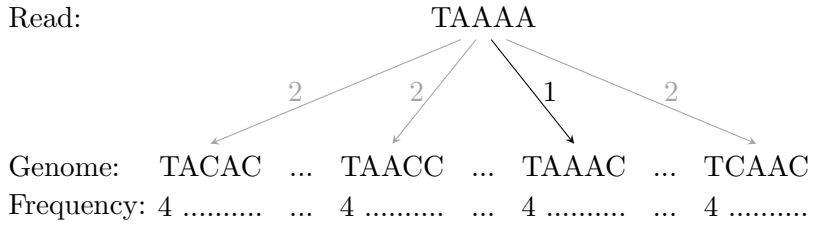
Instead, we suggest a technique that takes the mappability information into account during the mapping of reads [Bönigk, 2018]. Once the reference genome is indexed, the (k, e)-frequency is computed prior to mapping the reads where k is the read length and e is set with respect to the error rate of the sequencing technology, e.g., the maximum number of errors that the reads are later mapped with. A threshold parameter $\mathcal{T} \in \mathbb{N}$ is introduced.

Definition 4.5.1 (Mappable and unmappable locations). Given the (k, e)-frequency of a sequence T, a position i is called *mappable* if $F[i] \leq \mathcal{T}$, and *unmappable* otherwise.

Reads are guaranteed to be mapped to mappable locations, i.e., to positions in the text with a frequency value of up to $\mathcal{T}$. If a read matches a location with a greater frequency than $\mathcal{T}$, this position might not be mapped to as one tries to skip unmappable positions as soon as possible during the read mapping process. When a read matches a position, only the frequency value at the starting position of the read is considered to determine whether it is mappable. It is possible that a read matches uniquely with up to e errors to the reference genome, while the location itself is unmappable, i.e., has a high frequency, see figure 4.3 (b) for an example. Even though this rarely happens, the different terms of matching and mapping reads should be kept in mind.



(a) Mapping a read with up to 2 mismatches to a genome with computed $(5, 2)$-frequency. The read matches multiple locations of which only one is mappable and the others are unmappable, since their frequency value is greater than $\mathcal{T}$.



(b) Mapping a read with up to 1 mismatch to a genome with computed $(5, 1)$-frequency. The read matches only one position in the genome, even though it belongs to a mappable position. The other locations and the read each have a Hamming distance of 2.

Figure 4.3: Example of matching and mapping reads with $\mathcal{T} = 2$.

4.5.1 Constructing Frequency Vectors

The actual frequency values are not of interest since we only need to determine whether a location has a value greater than $\mathcal{T}$, hence we reduce the frequency vector F to a frequency bit vector, denoted as B. For simplicity we extend the frequency bit vector of length $n - k + 1$ to n by appending 1s with $n = |T|$. Since the reads of length k are too long to map to the last $k - 1$ text positions, they will be discarded at some point during backtracking anyway.

$$B[i] = \begin{cases} 0 & , \quad i \leq n - k + 1 \text{ and } M[i] \leq \mathcal{T} \\ 1 & , \quad \text{otherwise} \end{cases} \qquad \forall\, 1 \leq i \leq n \quad (4.3)$$

A read matching a location i with up to e errors is considered mappable if and only if $B[i] = 0$, i.e., it only depends on the frequency value of the first position of the read in the genome and the threshold parameter $\mathcal{T}$.

We want to be able to identify repeat regions during the mapping of a read. Since the preliminary occurrences during the backtracking are represented as suffix array ranges, the frequency bit vector has to be reordered. We will show how to access the frequency bit vector during approximate string matching in a bidirectional FM index using (optimum) search schemes to determine whether the eventual starting positions of the read are mappable or not. Most search schemes consist of three searches: a forward search (searching the read from left to right), a backward search (searching the read from right to left) and a bidirectional search (starting in the middle of the read and switching directions at some point). For each of these three searches we describe how the frequency bit vector has to be reordered and how the relevant information in a search step can be accessed. Let $\mathcal{I}$ and $\mathcal{I}^{rev}$ be the indices on the original and the reversed text's index forming the bidirectional FM index.

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T[i]$:	A	C	C	C	A	A	C	G	A	C	G	G	A	A	C	G	\$
$SA[i]$:	17	13	5	1	14	6	9	4	3	2	15	7	10	16	12	8	11

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T^{rev}[i]$:	G	C	A	A	G	G	C	A	G	C	A	A	C	C	C	A	\$
$SA^{rev}[i]$:	17	16	11	3	12	8	4	15	10	2	7	14	13	9	1	6	5

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$F[i]$:	1	2	2	3	3	2	2	3	2	1	1	2	3	∞	∞	∞	∞
$B[i]$:	0	0	0	1	1	0	0	1	0	0	0	0	1	1	1	1	1

Figure 4.4: $(4,1)$-frequency with appended frequency values set to ∞ for the text $T = ACCCAACGACGGAACG\$$. Since a sentinel character is appended, not only the last $k-1$ text positions have undefined frequency values, but also the kth position from behind. Bit vector is built for $\mathcal{T} = 2$.

Example 4.5.1.
We consider the text given in figure 4.4 and want to map reads of length 4 with up to 1 mismatch. The $(4,1)$-frequency vector is computed and reduced to a bit vector for $\mathcal{T} = 2$. The read $ACCG$ matches the locations 1, 5 and 13 with one substitution and location 9 without any error. Only the locations 1 and 9 are mappable and 5 and 13 belong to a repeat with respect to $\mathcal{T}$. Hence, we want to skip the locations 5 and 13 before they are matched in the index, preferably as soon as possible.

Forward search If we want to search a read $P[1..k]$ from left to right in a bidirectional FM index, we will perform backward search steps on the reversed text's index $\mathcal{I}^{rev}$. The text positions stored in the corresponding suffix array range of $\mathcal{I}$ point to the first character of the partially matched read $P[1..j]$ for $1 \leq j \leq k$. Since we need to retrieve the corresponding frequency bit of the beginning of the fully mapped read and since we search the read from left to right, we can simply define a bit vector B_{fwd} in suffix array order of $\mathcal{I}$:

$$B_{\text{fwd}}[i] = B[SA[i]] \qquad \forall\, 1 \leq i \leq n \tag{4.4}$$

Hence, while searching in $\mathcal{I}^{rev}$ the suffix array range $[a, b]$ of $\mathcal{I}$ can be used to examine the frequency values of the potential matches of the read at any time by accessing $B_{\text{fwd}}[i]$ for $i \in [a, b]$, see figure 4.5 for an example. This allows filtering repetitive locations during the search. We will explain the details later.

Example 4.5.2 (Forward search).
We consider the read $ACCG$ from the previous example. During approximate string matching when extending the partial read from A to AA (allowing a mismatch), the suffix array range $[a, b] = [2, 3]$ in $\mathcal{I}$ (highlighted in red) reveals that all positions are not mappable, since $B_{\text{fwd}}[i] = 1$ for $i \in [a, b]$ and the backtracking does not have to be continued for this branch.

In another branch during backtracking we will have searched for ACG represented by the suffix array range $[a, b] = [5, 7]$ (highlighted in blue). The algorithm might continue with an in-text verification. Since $B_{\text{fwd}}[5] = 1$ the position 5 in the genome does not have to be verified and can be discarded immediately.

i	F	L	B_{fwd}
1	\$ ACCCAACGACGGAAC	G	1
2	**A A**CG\$ACCCAACGACG	G	**1**
3	**A A**CGACGGAACG\$ACC	C	**1**
4	A CCCAACGACGGAACG	\$	0
5	**A CG**\$ACCCAACGACGG	A	**1**
6	**A CG**ACGGAACG\$ACCC	A	**0**
7	**A CG**GAACG\$ACCCAAC	G	**0**
8	C AACGACGGAACG\$AC	C	1
9	C CAACGACGGAACG\$A	C	0
10	C CCAACGACGGAACG\$	A	0
11	C G\$ACCCAACGACGGA	A	1
12	C GACGGAACG\$ACCCA	A	0
13	C GGAACG\$ACCCAACG	A	0
14	G \$ACCCAACGACGGAA	C	1
15	G AACG\$ACCCAACGAC	G	0
16	G ACGGAACG\$ACCCAA	C	1
17	G GAACG\$ACCCAACGA	C	0

Figure 4.5: Index $\mathcal{I}$ on T with the frequency bit vector B_{fwd} for forward searches ($(4,1)$-frequency with $\mathcal{T} = 2$).

Backward search For searching a read $P[1..k]$ from right to left it is the other way around. Backward searches are performed in $\mathcal{I}$. The suffix array range of $\mathcal{I}$ represents the occurrences of $P[j..k]$ for $1 \leq j \leq k$ in the text. Thus, with every backward search the locations refer to a different starting position of a suffix of P in the text. To be able to access the frequency information after each backward search, we need the starting positions of the read to be fixed. Hence, we use the suffix array range $[a^{rev}, b^{rev}]$ of $\mathcal{I}^{rev}$ which stores the locations of $P^{rev}[1..(k-j+1)]$. We will arrange the bit vector in suffix array order of $\mathcal{I}^{rev}$. Since B is in text order and not in reversed text order, it is accessed via $B[n - SA^{rev}[i] + 1]$ with $i \in [a^{rev}, b^{rev}]$. As the occurrences of the read in the reversed text are also reversed, the frequency bit vector B_{bwd} needs to be shifted by $k-1$ to the right such that the frequency values of the k-mers are not aligned to the left, but to the right. If $SA^{rev}[i] \geq n-k+1$ the read will eventually not match, since there are not enough characters left at the end of the text, hence we set the corresponding bits of the frequency bit vector to 1, i.e., mark these positions as unmappable, allowing the matching to these locations to abort earlier. For $1 \leq i \leq n$ we define B_{bwd}:

$$B_{\text{bwd}}[i] = \begin{cases} B[n - SA^{rev}[i] - k + 2] & , \quad SA^{rev}[i] \leq n-k+1 \\ 1 & , \quad \text{otherwise} \end{cases} \tag{4.5}$$

Example 4.5.3 (Backward search).
Again, we search the read $ACCG$ with up to one substitution from right to left. We consider a backtracking branch that checks for an error at the very first position searched, i.e., the last character in the read. Let the mismatch character be C. The suffix array range in $\mathcal{I}^{rev}$ is $[a^{rev}, b^{rev}] = [8, 13]$. Since the first four locations are not mappable, i.e., $B_{\text{bwd}}[i] = 1$ for $i \in [8, 11]$, one only has to consider the subrange $[12, 13]$.

The running time improvement in this example is negligible, since

we only allow for one substitution in total and $[8, 11]$ would be eliminated anyway when trying to match the next character. However, it should be kept in mind that these are minimal examples. For many applications we allow for more errors and deal with significantly larger suffix array ranges.

i	F	L	B_{bwd}
1	\$ GCAAGGCAGCAACCC	A	1
2	A \$GCAAGGCAGCAACC	C	1
3	A ACCCA\$GCAAGGCAG	C	1
4	A AGGCAGCAACCCA\$G	C	0
5	A CCCA\$GCAAGGCAGC	A	0
6	A GCAACCCA\$GCAAGG	C	0
7	A GGCAGCAACCCA\$GC	A	0
8	**C** A\$GCAAGGCAGCAAC	C	**1**
9	**C** AACCCA\$GCAAGGCA	G	**1**
10	**C** AAGGCAGCAACCCA\$	G	**1**
11	**C** AGCAACCCA\$GCAAG	G	**1**
12	**C** CA\$GCAAGGCAGCAA	C	**0**
13	**C** CCA\$GCAAGGCAGCA	A	**0**
14	G CAACCCA\$GCAAGGC	A	0
15	G CAAGGCAGCAACCCA	\$	1
16	G CAGCAACCCA\$GCAA	G	0
17	G GCAGCAACCCA\$GCA	A	0

Figure 4.6: Index $\mathcal{I}^{rev}$ on T with the frequency bit vector B_{bwd} for backward searches ($(4, 1)$-frequency with $\mathcal{T} = 2$).

Bidirectional search Pure forward and backward searches have always one end of the read aligned that will not be extended. This ensures that the frequency bit vector can always be accessed using one of the two suffix array ranges. For bidirectional searches this is different. Let us assume that the search starts at some position j in the read, performs an extension to the right and afterwards extends it to the left. When searching to the right, the read is anchored to the left at position j, when it is extended to the left, it is anchored to the right at the end of the read. For the extension to the left we can access B_{bwd} like for pure backward searches, for the extension to the right we need another frequency bit vector B_{bi} that is built similarly to B_{fwd}, but aligned $j-1$ positions to the right. An example is given in figure 4.7. Again, if $SA[i] < j$ the matching will eventually fail, hence we set the corresponding bits of the frequency bit vector to 1.

$$B_{\text{bi}}[i] = \begin{cases} B[SA[i] - (j-1)] & , \quad SA[i] \geq j \\ 1 & , \quad \text{otherwise} \end{cases} \qquad \forall\, 1 \leq i \leq n \tag{4.6}$$

Example 4.5.4 (Bidirectional search).
Again, we search the read $ACCG$ with up to one substitution, starting from the third character, extending it to the right and afterwards to the left, hence, the bit vector B_{bi} is needed for $j = 3$. If we start the search with an exact match of the 3rd character, the suffix array range $[a, b] = [8, 13]$ is retrieved. $B_{\text{bi}}[i]$ for $i \in [8, 13]$ indicates that only the first two text positions $SA[8]$ and $SA[9]$ might eventually match a mappable position in the text, which are $SA[8] - (j-1) = 4 - 2 = 2$ and $SA[9] - (j-1) = 3 - 2 = 1$. Eventually the read will only be mapped to position 1 and not to position 2 as this requires more than one mismatch.

i	F	L	B_{bi}
1	\$ ACCCAACGACGGAAC	G	1
2	A ACG\$ACCCAACGACG	G	0
3	A ACGACGGAACG\$ACC	C	0
4	A CCCAACGACGGAACG	\$	1
5	A CG\$ACCCAACGACGG	A	0
6	A CGACGGAACG\$ACCC	A	1
7	A CGGAACG\$ACCCAAC	G	0
8	**C** AACGACGGAACG\$AC	C	**0**
9	**C** CAACGACGGAACG\$A	C	**0**
10	**C** CCAACGACGGAACG\$	A	**1**
11	**C** G\$ACCCAACGACGGA	A	**1**
12	**C** GACGGAACG\$ACCCA	A	**1**
13	**C** GGAACG\$ACCCAACG	A	**1**
14	G \$ACCCAACGACGGAA	C	1
15	G AACG\$ACCCAACGAC	G	0
16	G ACGGAACG\$ACCCAA	C	0
17	G GAACG\$ACCCAACGA	C	0

Figure 4.7: Index $\mathcal{I}$ on T with the frequency bit vector B_{bi} for bidirectional searches beginning at position $j = 3$ in the read and extending it to the right first ($(4, 1)$-frequency with $\mathcal{T} = 2$).

While going down the backtracking tree, the frequency values of a suffix array range can always be examined by using one of the created bit vectors. In the next step we want to examine the frequency bit vectors during the index-based search and skip as many partially matched reads that map to repetitive regions, i.e., have the frequency bit set to 1. To better analyze the amount of mappable positions for a given suffix array range, we add constant-time rank support to all bit vectors. We present two approaches for eliminating unmappable locations from the search during backtracking. In general the infix of the frequency bit vector contains both, zeros and ones in a mixed order. To skip unmappable locations, a suffix array range can be split into multiple subranges and backtracking is continued with each subrange separately, or in case the infix contains only few mappable positions, they can be located and verified by an in-text verification as presented in section 3.8.

4.5.2 Splitting Suffix Array Ranges

While splitting suffix array ranges into subranges to skip unmappable locations sounds promising, it has two downsides. First, splitting one range into multiple smaller ones requires continuing backtracking separately on each of the subranges, increasing the overall running time. To compensate for this, one has to cut out large continuous blocks of unmappable locations. Second, when choosing a subrange or splitting the range into subranges, in most cases the bidirectional FM index cannot be synchronized anymore. An arbitrary suffix array range is generally not a continuous suffix array range in the reversed text's index (see example 4.5.5). Hence, when selecting a subrange one has to finish the entire mapping of these locations using a unidirectional FM index or in-text verification. Furthermore, one also loses the ability to access the frequency information during a unidirectional search. The unidirectional search always performs backward searches, but one needs the suffix array range of the other index to retrieve the

text locations and its frequency bits of the matched prefix.

Example 4.5.5 (Selecting subranges).
Consider the suffix array range $[2, 7]$ in $\mathcal{I}$ matching the prefix A (single character) in figure 4.5. When shrinking this range to $[2, 5]$, the longest common prefix is still A. The corresponding suffix array indices in $\mathcal{I}^{rev}$ are 2, 4, 5 and 7, hence they do not form a continuous block.

In conclusion, one can examine the frequency information after each search step, but one can only split the suffix array range once on each path in the backtracking tree. In the subsequent search steps the frequency information cannot be examined anymore. If the splitting is performed in a bidirectional search of the search scheme before switching directions, eventually an in-text verification has to be performed. Hence, it should be carefully considered if and when to shrink or split the suffix array range.

As an indicator whether to split the suffix array range, we examine the infix of the corresponding frequency bit vector with respect to the number of mappable locations and whether they form larger continuous blocks. The former is rated by the percentage of mappable locations which can be computed using rank queries, the latter by the percentage of bits flipped in the infix of the bit vector, see equation 4.7. The number of bit flips can efficiently be counted by $\text{count}(B[a..b] \oplus (B[a..b] \gg 1))$ where $\oplus$ denotes bitwise xor. Counting can be performed by using popcount on each 64-bit word.

$$\begin{aligned} \frac{\text{rank}_0(B, b) - \text{rank}_0(B, a)}{b - a} &\leq c_0 \\ \frac{|\{\ell \mid \ell \in [a, b-1] \wedge B[\ell] \neq B[\ell + 1]\}|}{b - a} &\leq c_1 \end{aligned} \tag{4.7}$$

The partition into subranges is performed by a linear scan over the infix of the frequency bit vector. Instead of selecting subranges

that contain only mappable locations, a small number of unmappable locations are allowed in each subrange to reduce the number of subranges.

4.5.3 In-Text Verification

In-text verification can be used at any time during the backtracking by retrieving the text positions and verify whether the rest of the read matches, especially if the suffix array range $[a, b]$ or a subrange only has few mappable locations, i.e.,

$$\text{rank}_0(B, b) - \text{rank}_0(B, a) \leq c_2 \tag{4.8}$$

Determining the optimal parameters c_0, c_1 and c_2 is even more challenging than for pure in-text verification. Again, this depends on the number of errors and characters left as they have an impact on the running time needed for the remaining backtracking, but even more importantly on the underlying data and its repetitiveness.

4.5.4 Benchmarks

To evaluate the performance of read mapping using mappability information, we implemented a prototype for Hamming distance which is available on GitHub[2]. We ran the benchmarks using $100,000$ sampled Illumina reads from a whole genome sequencing experiment of the human genome from section 3.7. As parameters we chose $c_0 = 0.1$, $c_1 = 0.3$ and $c_2 = 25$, i.e., suffix array ranges are only split if they contain at most 10 % mappable positions and at most 30 % of bit flips in the corresponding infix of the frequency bit vectors. This ensures that suffix array ranges are only split when there is a large subset of unmappable positions and they are not too widespread into small blocks. In-text verification is performed for less than 25 occurrences as in the previous chapter.

[2] `https://github.com/svnbgnk/mappability`

Table 4.3 compares the running times of OSS* and OSS* with in-text verification to our new approach incorporating mappability information into the search. We evaluate the performance for different threshold values $\mathcal{T}$. We can observe a noticeable drop in running times for larger errors of up to 30 % (for $K = 3$) when comparing it to optimum search schemes with in-text verification. It is worth mentioning that a large increase of $\mathcal{T}$ from 7 to 100 only reduces the performance increase by 6 %, and an increase to 250 only by another 2 %. Hence, we can still improve the running time while guaranteeing that reads are mapped to all locations with a frequency of less or equal than 250.

Strategy	$K = 1$	$K = 2$	$K = 3$
OSS*	7.4s	21.0s	57.0s
OSS*+ ITV	4.7s	11.0s	27.0s
OSS*+ ITV, $\mathcal{T} = 7$	4.0s	7.4s	19.3s
OSS*+ ITV, $\mathcal{T} = 10$	4.0s	7.5s	19.6s
OSS*+ ITV, $\mathcal{T} = 100$	4.3s	8.2s	21.0s
OSS*+ ITV, $\mathcal{T} = 250$	4.3s	8.4s	21.5s

Table 4.3: Running times for approximate string matching with mappability information. Reads do not have to be mapped to locations with a frequency value greater than $\mathcal{T}$.

Tables 4.4 (a) and (b) show the percentage of reads that only mapped to unmappable locations and were not reported, and reads that mapped to both, mappable and unmappable locations where unmappable locations were dropped during the search by splitting the suffix array range or performing in-text verification on single suffix array values. Naturally, the number of unmappable reads shrinks for greater threshold values, but at the same time the number of reads,

for which unmappable locations were dropped tends to increase. This supports our choice of the parameters c_0, c_1 and c_2 as even for greater threshold values and hence fewer unmappable positions, more unmappable locations were dropped by splitting the suffix array range.

$\mathcal{T}$	$K = 1$	$K = 2$	$K = 3$
7	2.09 %	4.26 %	5.37 %
10	1.54 %	3.61 %	4.65 %
100	0.48 %	1.74 %	2.45 %
250	0.32 %	1.23 %	1.86 %

(a) Percentage of completely unmappable reads that match at least one position in the genome.

$\mathcal{T}$	$K = 1$	$K = 2$	$K = 3$
7	2.35 %	1.54 %	1.66 %
10	2.50 %	1.73 %	1.87 %
100	2.48 %	2.34 %	2.60 %
250	2.25 %	2.53 %	2.75 %

(b) Percentage of reads that match both, mappable and unmappable locations.

Table 4.4: Percentage of reads that were not mapped at all respectively only mapped to some of its location based on the frequency bit vectors. Reads are only considered if they match at least once in the genome.

The experiments show that our prototype implementation of a read mapper incorporating mappability information even further speeds up the mapping of reads to repeat regions. Instead of mappability information one can also use the masking information of a repeat masked genome to construct a bit vector to neglect mapping to repeat regions identified by repeat masking tools. No rebuilding of the FM index is necessary, only the bit vectors have to be constructed and reordered.

All the frequency vectors have only to be computed once for a given read length and error rate. The frequency bit vectors can be constructed by a linear scan over the suffix arrays for a given $\mathcal{T}$, i.e., for each read mapping run the threshold parameter can be adjusted at low cost in terms of running time.

For one error the optimal search scheme consists of a forward and a backward search, i.e., only two bit vectors are necessary. For two errors a third bit vector is needed for the bidirectional search. Since OSS^* for three errors has four searches of which two are bidirectional with different starting positions, four bit vectors are required in total. We did not evaluate $K = 4$ since our best search scheme for four errors contains searches that switch the direction multiple times. This is not yet supported by our prototype and would also require additional bit vectors. The space consumption of each bit vector is exactly the length of the genome in bits, this yields 383 MB per bit vector for the human genome. Since rank support can be built quickly by a linear scan over the bit vector, it is not stored on disk, but created when the index and bit vectors are loaded by the read mapper.

The chosen parameters c_0, c_1 and c_2 show a significant speedup for mapping the reads, but more research on choosing these parameters or even other strategies for eliminating unmappable locations during the search might improve the running time even further. An extension of the prototype is currently being developed to also support Edit distance with mappability information and in-text verification based on the Myers bit vector algorithm.

4.6 Marker Genes

Marker genes, also referred to as phylogenetic markers or marker sequences, are short subsequences of genomes whose presence or absence allows determining the organism, species or even strain when sequencing an unknown sample or help building phylogenetic trees [Patwardhan et al., 2014]. These subsequences can be rather short or span entire genes. Depending on the data set a single gene might not be sufficient, e.g., to distinguish different strains of the same species with a high sequence similarity. Hence, the combination of multiple marker genes might be necessary.

Without assembling the sequencing data we cannot search for marker genes straightforward. Depending on the marker length it can span dozens of reads. Instead of assembling the strain or applying experimental methods such as PCR-based AFLP (amplified fragment length polymorphism) [Vos et al., 1995], we propose using the mappability. The mappability can be computed on a set of strains, i.e., each k-mer is searched and counted in all strains at once. We consider two use cases: on the one hand we want to identify k-mers that match a sequence uniquely to determine the exact strain. On the other hand, we want to search for k-mers shared by many strains in the same phylogenetic group.

Small adjustments to the mappability algorithm are necessary. The frequency vector can be computed on multiple strains at once, but the algorithm does not distinguish whether a k-mer of some strain matches the same strain multiple times or different strains. While the original algorithm simply counts the occurrences, we are now interested in the number of different strains it matches for finding marker sequences. A k-mer with a frequency greater than 1 can still be a suitable marker sequence if it only matches one of the strains.

An adapted version of our mappability algorithm to search for marker genes is available on GitHub on a separate branch[3]. We ran experiments on E. coli strains to demonstrate the use of mappability information for identifying marker sequences.

It was shown that E. coli can be grouped into four major phylogenetic groups (A, B1, B2, and D) [Clermont et al., 2000]. The authors identified two marker genes (chuA and yjaA) and an anonymous DNA fragment (TspE4.C2) whose combination of presence or absence in the genome can determine the phylogenetic group. They used 230 E. coli strains and achieved an accuracy of 99 % in classifying these strains based on the correlation of genes and groups listed in figure 4.8.

[3] `https://github.com/cpockrandt/mappability`, branch marker_genes

Group	chuA	yjaA	TspE4.C2
A	-		-
B1	-		+
B2	+	+	
D	+	-	

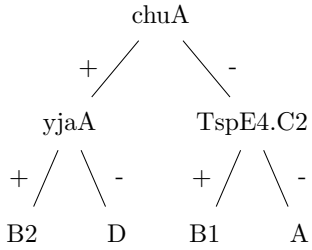


Figure 4.8: Genes and DNA fragments for classifying E. coli strains into four phylogenetic groups A, B1, B2, and D. + and - represent a correlation for the presence or absence of a gene or sequence in 100 % of all cases. 77 % of A strains and 9 % of B1 strains contained the yjaA gene, 93 % of B2 and 10 % of D strains contained the TspE4.C2 sequence. Based on this data, a decision tree can be built.

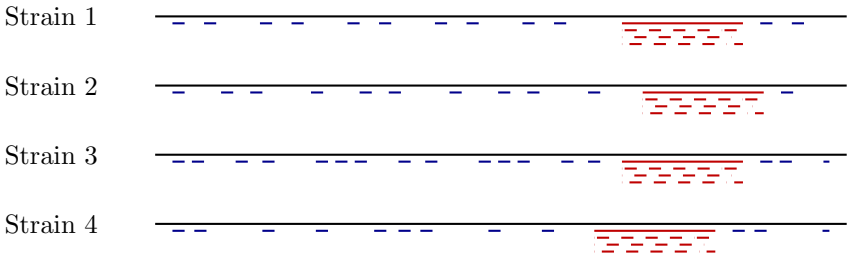
For the first experiment we select four different strains of the phylogenetic group B1 and compute the $(30, 2)$-mappability. According to the study all strains within B1 share the anonymous DNA fragment TspE4.C2 of 152 base pairs. We used the mappability tool to search for both, unique k-mers among all strains as well as k-mers that occur in each strain at least once. We observed that TspE4.C2 is an exact match in all strains and the 30-mers in this region also have a mappability value of exactly 0.25. We further found numerous 30-mers with a mappability of 1, thus allowing to determine a strain among those four, while still accounting for sequencing errors and mutations. Figure 4.9 (a) illustrates the experiment, statistics on the identified unique 30-mers are given in table 4.5. We counted the number of k-mers matching only one strain, i.e., the strain the k-mer originated from. We refer to this count as *unique*. Additionally, we counted how many of these k-mers matched multiple times

to the strain, referred to as *pseudo*. To avoid counting highly overlapping k-mers in large unique regions, we break down the numbers for non-adjacent k-mers as well, i.e., for a k-mer to be considered it must have a preceding k-mer with a frequency value greater than 1.

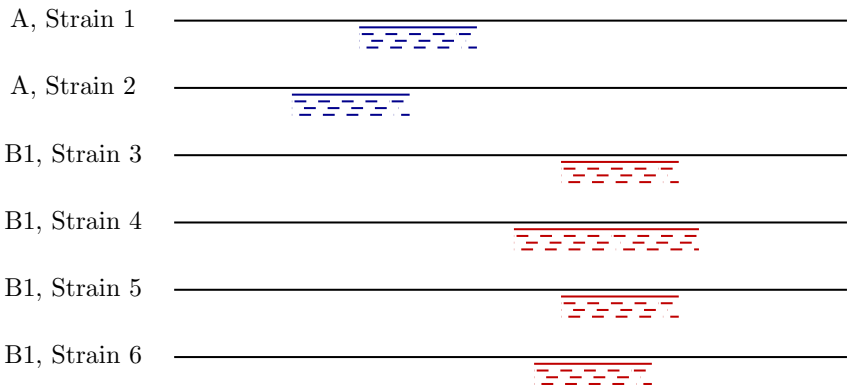
	all k-mers			non-adjacent k-mers		
Strain	Unique	Pseudo	∅ Dist.	Unique	Pseudo	∅ Dist.
IAI1	171,942	4,992	27 ± 627	1,829	81	$2,476 \pm 5,560$
SE11	305,439	10,365	15 ± 447	2,356	176	$1,942 \pm 4,708$
11128	260,305	40,101	20 ± 953	2,494	685	$2,049 \pm 9,517$
11368	434,033	108,968	13 ± 912	3,142	1,116	$1,674 \pm 10,592$

Table 4.5: $(30, 2)$-mappability on four strains of E. coli assigned to the phylogenetic group B1 based on the known marker genes by Clermont et al. The number of unique 30-mers with up to 2 errors were counted (*unique*), as well as unique k-mers among the strains with multiple occurrences in the strain itself (*pseudo*). We computed the mean distance of the unique marker sequences and their standard deviation. We also filtered out adjacent unique k-mers to count regions with many overlapping unique k-mers only once.

In table 4.6 we present the data of a second experiment, where we select strains from more than one group (A and B1), see figure 4.9 (b) for an illustration. Again, we computed the $(30, 2)$-mappability, but this time we counted k-mers that match all strains in one group but no strain in the other group. As the data shows the number of unique k-mers inside a group does not have to be equal for each strain. Consider a k-mer k_1 from *W3110* matching in *HS* with one error (we call the corresponding k-mer in *HS* k_2), and matching no strain in B1, since at least three errors might be required for k_1 to match. Hence, k_1 is counted as a unique k-mer for group A. Due to the symmetry of distance metrics, k_2 of *HS* matches k_1 in *W3110*,



(a) Four strains belonging to the same phylogenetic group. The sequence in red is conserved within this group and a marker gene. The red k-mers belonging to this marker gene are also all found in the other strains. The k-mers in blue are unique among all four strains and allow distinguishing each of the strains.



(b) Six sequences belonging to two different phylogenetic groups. Marker sequences are highlighted in red and blue that only occur in one the groups and are present in all of its strains.

Figure 4.9: Illustration of the experiments performed on E. coli sequences in tables 4.5 and 4.6.

but this time it might also match a k-mer in B1 with only two errors. For $e = 0$ the number of unique k-mers in a group are identical for each strain.

		all k-mers		non-adjacent k-mers	
Group	Strain	Unique	∅ Dist.	Unique	∅ Dist.
A	W3110	109,375	41 ± 731	2,398	$1,867 \pm 4,577$
A	HS	111,179	39 ± 709	2,414	$1,796 \pm 4,471$
B1	IAI1	125,042	37 ± 680	3,063	$1,485 \pm 4,091$
B1	SE11	127,302	38 ± 690	3,123	$1,510 \pm 4,148$
B1	11128	121,325	42 ± 766	3,275	$1,548 \pm 4,408$
B1	11368	131,121	41 ± 814	3,473	$1,537 \pm 4,763$

Table 4.6: $(30, 2)$-mappability on six strains of E. coli of the groups A and B1. Only k-mers were counted that perfectly separated the strains in A from B1, i.e., if and only if the k-mer matched all strains of A and no strain of B1 and vice versa.

We downloaded the whole genome assemblies of the strains listed in table A.6 and computed the $(30, 2)$-mappability on the entire data set. We were able to verify the marker genes from Clermont et al. for groups A and B1, although on a small data set. Our experiments enumerate large numbers of possible marker sequences. Since computing the $(30, 2)$-mappability on a few E. coli takes even on a consumer laptop less than a minute, this method is suitable to run on large sets of similar E. coli strains to identify marker sequences, even with errors accounting for uncertainty from sequencing and mutations such as SNPs.

5 Conclusion

In this thesis we covered two important parts of many bioinformatics applications: indexing data structures and approximate string matching algorithms for an index-based search. We improved both areas by introducing unidirectional and bidirectional FM indices based on a novel data structure, EPR dictionaries. To our knowledge this is not even the only available implementation with constant running time per search step in bidirectional indices, but also outperforms all other available FM index implementations in practice.

Furthermore we improved backtracking strategies for approximate string matching in bidirectional FM indices both, in theory and practice. We formulated the concept of search schemes as an integer linear program and computed optimal search schemes. We showed that they can easily speed up the seeding phase of state-of-the-art read mappers and even allow searching short sequences with high-error rates in an index. We have seen that there are even better search schemes than we were able to compute with the ILP. We are optimistic that improving the ILP to handle larger instances will allow us to find even better performing search schemes. Additionally, we have investigated hybrid approaches using in-text verification and observed that it is beneficial to stop an index-based search at some point and verify the partial matches directly in the text. Even though we have achieved significant speedups already, we expect that this strategy can be improved even more.

These advancements in indexing data structures and approximate string matching allowed us to develop an algorithm for computing the mappability that is a magnitude faster than the efficient read map-

ping tool suite GEM while not using any heuristics. Based on the mappability we accomplished even further speedups for read mapping due to reads mapping to highly repetitive regions. What has proven to be effective for Hamming distance, such as in-text verification or read mapping with mappability information can be continued for other distance metrics such as Edit distance and is considered as future research.

References

[Abouelhoda et al., 2004] Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86.

[Apweiler et al., 2004] Apweiler, R., Bairoch, A., Wu, C. H., Barker, W. C., Boeckmann, B., Ferro, S., Gasteiger, E., Huang, H., Lopez, R., Magrane, M., et al. (2004). UniProt: the universal protein knowledgebase. *Nucleic acids research*, 32(suppl_1):D115–D119.

[Belazzougui et al., 2013] Belazzougui, D., Cunial, F., Kärkkäinen, J., and Mäkinen, V. (2013). Versatile succinct representations of the bidirectional Burrows-Wheeler transform. In *European Symposium on Algorithms*, pages 133–144. Springer.

[Burrows and Wheeler, 1994] Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. *Digital Systems Research Center Research Reports*.

[Bönigk, 2018] Bönigk, S. (2018). *Incorporating mappability into a read mapper*. Internship report, Freie Universität Berlin.

[Cameron et al., 2017] Cameron, P., Fuller, C. K., Donohoue, P. D., Jones, B. N., Thompson, M. S., Carter, M. M., Gradia, S., Vidal, B., Garner, E., Slorach, E. M., et al. (2017). Mapping the genomic landscape of CRISPR–Cas9 cleavage. *Nature methods*, 14(6):600.

[Clermont et al., 2000] Clermont, O., Bonacorsi, S., and Bingen, E. (2000). Rapid and simple determination of the Escherichia

coli phylogenetic group. *Applied and environmental microbiology*, 66(10):4555–4558.

[CPLEX, 2009] CPLEX, I. I. (2009). V12.1: User's manual for CPLEX. *International Business Machines Corporation*, 46(53):157.

[Derrien et al., 2012] Derrien, T., Estellé, J., Sola, S. M., Knowles, D. G., Raineri, E., Guigó, R., and Ribeca, P. (2012). Fast computation and applications of genome mappability. *PloS one*, 7(1):e30377.

[Doudna and Charpentier, 2014] Doudna, J. A. and Charpentier, E. (2014). The new frontier of genome engineering with CRISPR-Cas9. *Science*, 346(6213):1258096.

[Edgar and Myers, 2005] Edgar, R. C. and Myers, E. W. (2005). PILER: identification and classification of genomic repeats. *Bioinformatics*, 21(suppl_1):i152–i158.

[Ferragina and Manzini, 2000] Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE.

[Ferragina et al., 2007] Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. (2007). Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):20.

[Fletcher and Patty, 1987] Fletcher, P. and Patty, C. (1987). *Foundations of higher mathematics*, page 27. Prindle, Weber & Schmidt series in mathematics. PWS-Kent Pub. Co.

[Galil and Giancarlo, 1988] Galil, Z. and Giancarlo, R. (1988). Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4(1):33–72.

[Gog et al., 2014] Gog, S., Beller, T., Moffat, A., and Petri, M. (2014). From theory to practice: plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337.

[González et al., 2005] González, R., Grabowski, S., Mäkinen, V., and Navarro, G. (2005). Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38.

[Granlund, 2017] Granlund, T. (2017). Instruction latencies and throughput for AMD and Intel x86 processors. `https://gmplib.org/~tege/x86-timing.pdf`. [Online; accessed 4-December-2018].

[Grossi et al., 2003] Grossi, R., Gupta, A., and Vitter, J. S. (2003). High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics.

[Gröpl and Reinert, 2013] Gröpl, C. and Reinert, K. (2013). Lecture notes on suffix arrays. `http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture1Materials/script.pdf`. [Online; accessed 4-December-2018].

[Hamming, 1950] Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160.

[Hauswedell et al., 2014] Hauswedell, H., Singer, J., and Reinert, K. (2014). Lambda: the local aligner for massive biological data. *Bioinformatics*, 30(17):i349–i355.

[Jacobson, 1988] Jacobson, G. J. (1988). *Succinct static data structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

[Kärkkäinen and Sanders, 2003] Kärkkäinen, J. and Sanders, P. (2003). Simple linear work suffix array construction. In *International Colloquium on Automata, Languages, and Programming*, pages 943–955. Springer.

[Kianfar et al., 2018] Kianfar, K., Pockrandt, C., Torkamandi, B., Luo, H., and Reinert, K. (2018). Optimum search schemes for approximate string matching using bidirectional FM-index. *bioRxiv*, page 301085.

[Kielbasa et al., 2011] Kielbasa, S. M., Wan, R., Sato, K., Horton, P., and Frith, M. (2011). Adaptive seeds tame genomic sequence comparison. *Genome research*, pages gr–113985.

[Knuth, 1997] Knuth, D. E. (1997). *The art of computer programming: sorting and searching*, volume 3, page 158. Pearson Education.

[Koehler et al., 2010] Koehler, R., Issac, H., Cloonan, N., and Grimmond, S. M. (2010). The uniqueome: a mappability resource for short-tag sequencing. *Bioinformatics*, 27(2):272–274.

[Kucherov et al., 2016] Kucherov, G., Salikhov, K., and Tsur, D. (2016). Approximate string matching using a bidirectional index. *Theoretical Computer Science*, 638:145–158.

[Lam et al., 2009] Lam, T. W., Li, R., Tam, A., Wong, S., Wu, E., and Yiu, S.-M. (2009). High throughput short read alignment via bi-directional BWT. In *Bioinformatics and Biomedicine, 2009. BIBM'09. IEEE International Conference on*, pages 31–36. IEEE.

[Langmead and Salzberg, 2012a] Langmead, B. and Salzberg, S. (2012a). Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359.

[Langmead and Salzberg, 2012b] Langmead, B. and Salzberg, S. L. (2012b). Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357.

[Langmead et al., 2009] Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

[Li and Durbin, 2009a] Li, H. and Durbin, R. (2009a). Fast and accurate short read alignment with Burrows–Wheeler transform. *bioinformatics*, 25(14):1754–1760.

[Li and Durbin, 2009b] Li, H. and Durbin, R. (2009b). Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760.

[Listgarten et al., 2018] Listgarten, J., Weinstein, M., Kleinstiver, B. P., Sousa, A. A., Joung, J. K., Crawford, J., Gao, K., Hoang, L., Elibol, M., Doench, J. G., et al. (2018). Prediction of off-target activities for the end-to-end design of CRISPR guide RNAs. *Nature Biomedical Engineering*, 2(1):38.

[Mäkinen, Veli and Belazzougui, Djamal and Cunial, Fabio and Tomescu, Ale
Mäkinen, Veli and Belazzougui, Djamal and Cunial, Fabio and Tomescu, Alexandru I (2015). *Genome-scale algorithm design*. Cambridge University Press.

[Manber and Myers, 1993] Manber, U. and Myers, G. (1993). Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948.

[Mascher et al., 2017] Mascher, M., Gundlach, H., Himmelbach, A., Beier, S., Twardziok, S. O., Wicker, T., Radchuk, V., Dockter, C., Hedley, P. E., Russell, J., et al. (2017). A chromosome conformation capture ordered sequence of the barley genome. *Nature*, 544(7651):427.

[Meyer et al., 2011] Meyer, F., Kurtz, S., Backofen, R., Will, S., and Beckstette, M. (2011). Structator: fast index-based search for RNA sequence-structure patterns. *BMC bioinformatics*, 12(1):214.

[Moore, 1965] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117.

[Murphy et al., 2000] Murphy, L. R., Wallqvist, A., and Levy, R. M. (2000). Simplified amino acid alphabets for protein fold recognition and implications for folding. *Protein Engineering*, 13(3):149–152.

[Myers, 1999] Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415.

[Patwardhan et al., 2014] Patwardhan, A., Ray, S., and Roy, A. (2014). Molecular markers in phylogenetic studies-a review. *Journal of Phylogenetics & Evolutionary Biology*, 2014.

[Pavlopoulos et al., 2013] Pavlopoulos, G. A., Oulas, A., Iacucci, E., Sifrim, A., Moreau, Y., Schneider, R., Aerts, J., and Iliopoulos, I. (2013). Unraveling genomic variation from next generation sequencing data. *BioData mining*, 6(1):13.

[Pockrandt et al., 2017] Pockrandt, C., Ehrhardt, M., and Reinert, K. (2017). EPR-Dictionaries: a practical and fast data structure for constant time searches in unidirectional and bidirectional FM-indices. In *International Conference on Research in Computational Molecular Biology*, pages 190–206. Springer.

[Pockrandt, 2015] Pockrandt, C. M. (2015). *Generic implementation of a bidirectional FM-index in SeqAn and applications.* Master's thesis, Freie Universität Berlin.

[Raman et al., 2002] Raman, R., Raman, V., and Rao, S. S. (2002). Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242. Society for Industrial and Applied Mathematics.

[Ranjekar et al., 1976] Ranjekar, P., Pallotta, D., and Lafontaine, J. (1976). Analysis of the genome of plants: II. Characterization of repetitive DNA in barley (Hordeum vulgare) and wheat (Triticum aestivum). *Biochimica et Biophysica Acta (BBA)-Nucleic Acids and Protein Synthesis*, 425(1):30–40.

[Reinert et al., 2017] Reinert, K., Dadi, T. H., Ehrhardt, M., Hauswedell, H., Mehringer, S., Rahn, R., Kim, J., Pockrandt, C., Winkler, J., Siragusa, E., et al. (2017). The SeqAn C++ template library for efficient sequence analysis: a resource for programmers. *Journal of biotechnology*, 261:157–168.

[Reuter et al., 2015] Reuter, J. A., Spacek, D. V., and Snyder, M. P. (2015). High-throughput sequencing technologies. *Molecular cell*, 58(4):586–597.

[Schnattinger et al., 2010] Schnattinger, T., Ohlebusch, E., and Gog, S. (2010). Bidirectional search in a string with wavelet trees. In *Annual Symposium on Combinatorial Pattern Matching*, pages 40–50. Springer.

[Siragusa, 2015] Siragusa, E. (2015). *Approximate string matching for high-throughput sequencing.* PhD thesis, Freie Universität Berlin.

[Smit et al., 1996] Smit, A., Hubley, R., and Green, P. (1996). RepeatMasker. `http://repeatmasker.org`.

[Strothmann, 2007] Strothmann, D. (2007). The affix array data structure and its applications to RNA secondary structure analysis. *Theoretical Computer Science*, 389(1-2):278–294.

[Tsai et al., 2015] Tsai, S. Q., Zheng, Z., Nguyen, N. T., Liebers, M., Topkar, V. V., Thapar, V., Wyvekens, N., Khayter, C., Iafrate, A. J., Le, L. P., et al. (2015). GUIDE-seq enables genome-wide profiling of off-target cleavage by CRISPR-Cas nucleases. *Nature biotechnology*, 33(2):187.

[Ukkonen, 1993] Ukkonen, E. (1993). Approximate string-matching over suffix trees. In *Annual Symposium on Combinatorial Pattern Matching*, pages 228–242. Springer.

[Vigna, 2008] Vigna, S. (2008). Broadword implementation of rank/select queries. In *International Workshop on Experimental and Efficient Algorithms*, pages 154–168. Springer.

[Vos et al., 1995] Vos, P., Hogers, R., Bleeker, M., Reijans, M., Lee, T. v. d., Hornes, M., Friters, A., Pot, J., Paleman, J., Kuiper, M., et al. (1995). AFLP: a new technique for DNA fingerprinting. *Nucleic acids research*, 23(21):4407–4414.

[Vroland et al., 2016] Vroland, C., Salson, M., Bini, S., and Touzet, H. (2016). Approximate search of short patterns with high error rates using the $01*0$ lossless seeds. *Journal of Discrete Algorithms*, 37:3–16.

[Weese, 2013] Weese, D. (2013). Lecture notes on the FM-index. `http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture7Materials/script.pdf`. [Online; accessed 4-December-2018].

[Wilson et al., 2018] Wilson, L. O. W., Hetzel, S., Pockrandt, C., Reinert, K., and Bauer, D. C. (under review, 2018). VARSCOT: Variant-aware detetion and site-scoring enables sensitive and personalized off-target detection. *Manuscript submitted for publication.*

[Yang et al., 2012] Yang, X., Chockalingam, S. P., and Aluru, S. (2012). A survey of error-correction methods for next-generation sequencing. *Briefings in bioinformatics*, 14(1):56–66.

Appendix

On the next pages all search schemes used in the experiments and benchmarks are listed.

	$K = 2$	$K = 3$	$K = 4$
Kucherov ($P = K + 1$)	$(123, 000, 022)$ $(321, 000, 012)$ $(213, 001, 012)$	$(1234, 0000, 0133)$ $(2134, 0011, 0133)$ $(3421, 0000, 0133)$ $(4321, 0011, 0133)$	$(12345, 00000, 02244)$ $(54321, 00000, 01344)$ $(21345, 00133, 01334)$ $(12345, 00133, 01334)$ $(43521, 00011, 01244)$ $(32145, 00013, 01244)$ $(21345, 00124, 01244)$ $(12345, 00034, 00444)$
Kucherov ($P = K + 2$)	$(1234, 0000, 0112)$ $(4321, 0000, 0122)$ $(2341, 0001, 0012)$ $(1234, 0002, 0022)$	$(12345, 00000, 01233)$ $(23451, 00000, 01223)$ $(34521, 00001, 01133)$ $(45321, 00012, 00333)$	$(123456, 000000, 012344)$ $(234561, 000000, 012344)$ $(654321, 000001, 012244)$ $(456321, 000012, 011344)$ $(345621, 000023, 011244)$ $(564321, 000133, 003344)$ $(123456, 000333, 003344)$ $(123456, 000044, 002444)$ $(342156, 000124, 002244)$ $(564321, 000044, 001444)$

Table A.1: Search schemes proposed in [Kucherov et al., 2016].

	$K = 3$	$K = 4$
OSS_4	$(12345, 00003, 02233)$	
	$(23451, 00022, 01223)$	
	$(34521, 00111, 01123)$	
	$(54321, 00000, 00333)$	
$\mathrm{MAN}_{\mathrm{best}}$		$(123456, 000004, 033344)$
		$(234561, 000000, 022334)$
		$(324561, 011111, 022334)$
		$(432561, 012222, 012334)$
		$(654321, 000033, 004444)$

Table A.2: Optimal search scheme for $K = 3$ (with $P = K + 2$ and $\overline{S} = 4$) and non-optimal search scheme for $K = 4$ derived from an optimal search scheme by lexicographically minimizing the U-string such that each string starts with 0.

	$K=1$	$K=2$	$K=3$	$K=4$
OSS_3	$(12, 00, 01)$	$(123, 002, 012)$	$(1234, 0003, 0233)$	$(12345, 00004, 03344)$
$P = K + 1$	$(21, 01, 01)$	$(321, 000, 022)$	$(2341, 0000, 1223)$	$(23451, 00000, 22334)$
		$(231, 011, 012)$	$(3421, 0022, 0033)$	$(54321, 00033, 00444)$
OSS_3	$(123, 001, 001)$	$(1234, 0011, 0022)$	$(12345, 00022, 00333)$	$(123456, 000004, 033344)$
$P = K + 2$	$(321, 000, 011)$	$(3214, 0000, 0112)$	$(43215, 00000, 11223)$	$(234561, 000000, 222334)$
		$(4321, 0002, 0122)$	$(54321, 00003, 02233)$	$(654321, 000033, 004444)$
OSS_3	$(1234, 0000, 0011)$	$(12345, 00011, 00222)$	$(123456, 000003, 022233)$	$(1234567, 0111111, 3333334)$
$P = K + 3$	$(4321, 0001, 0011)$	$(43215, 00000, 00112)$	$(234561, 000000, 111223)$	$(1234567, 0000000, 0044444)$
		$(54321, 00002, 01122)$	$(654321, 000022, 003333)$	$(7654321, 0000004, 0333344)$

Table A.3: Optimal search schemes for different number of errors and parts with at most $\overline{S} = 3$ searches. The schemes were computed using the ILP with $\sigma = 4$ (e.g., DNA alphabet) and a read length of $R = 101$.

	$K=1$	$K=2$	$K=3$	$K=4$
01*0 merged	$(123, 000, 011)$ $(231, 000, 001)$	$(1234, 0000, 0122)$ $(2341, 0000, 0122)$ $(3421, 0000, 0022)$	$(12345, 00000, 01333)$ $(23451, 00000, 01333)$ $(34521, 00000, 01333)$ $(45321, 00000, 00333)$	$(123456, 000000, 014444)$ $(234561, 000000, 014444)$ $(345621, 000000, 014444)$ $(456321, 000000, 014444)$ $(564321, 000000, 004444)$

Table A.4: 01*0 seeds by Vroland et al. formulated as search schemes. All 01*0 seeds with the same first error-free block are merged into one search.

Argument	Description
--index	path to the pre-built index (required)
--output	path to the mappability vector file (required)
--length	length of k-mer (required)
--errors	number of errors (required)
--revcompl	searches each k-mer also on the reverse strand (flag)
--high	increases the maximum mappability value stored from 255 to 65535 when set (flag)
--threads	number of threads (default: maximum supported by the system)
--mmap	turns on memory-mapping, i.e., the index is not loaded into main memory at the beginning, but in a lazy loading fashion (flag)
--overlap	number of adjacent k-mers searched at once (default: see equation 4.2)

Table A.5: Arguments and flags of GenMap

Strain	Group	GenBank accession
K-12 substr. W3110	A	GCA_000010245.1
HS O9:H4	A	GCA_000017765.1
IAI1 O8	B1	GCA_000026265.1
SE11 O152:H28	B1	GCA_000010385.1
11128 O111:H-	B1	GCA_000010765.1
11368 O26:H11	B1	GCA_000091005.1

Table A.6: List of E. coli strains with their phylogenetic groups and accession numbers used in the experiments.

Zusammenfassung

Diese Arbeit behandelt zentrale Algorithmen und Datenstrukturen aus der Sequenzanalyse am Beispiel von Read Mapping. Zuerst geben wir einen Überblick über den Stand der Forschung von FM-Indizes und stellen die aktuellsten Fortschritte vor, insbesondere die kürzlich veröffentlichten EPR-Dictionaries. Sie ermöglichen uni- und bidirektionale Suchen in FM-Indizes mit konstanter Laufzeit pro Suchschritt. Nach unserem Kenntnisstand handelt es sich dabei um die erste und derzeit einzige Implementierung bidirektionaler FM-Indizes mit konstanter Laufzeit. Wir zeigen, dass unsere Datenstruktur nicht nur optimale Laufzeiten für FM-Indizes in der Theorie erreichen, sondern auch in der Praxis schneller sind als andere frei verfügbare Implementierungen.

Im zweiten Kapitel widmen wir uns der approximativen Suche in bidirektionalen Indizes. Um die Laufzeit weiter zu verbessern und somit auch höhere Fehlerraten in indexbasierten Suchen zu ermöglichen, haben wir ein ganzzahliges lineares Programm entwickelt, um optimale Suchstrategien in bidirektionalen Indizes zu finden. Wir zeigen, dass die berechneten Strategien schneller sind als andere bisherige Ansätze und zeigen zusätzlich weitere Möglichkeiten der Laufzeitverbesserung auf. Mittels Verifizierungen kann eine index-basierte Suche vorzeitig abgebrochen, die möglichen Treffer lokalisiert und anschließend direkt im Text verifiziert werden.

Abschließend stellen wir einen noch unveröffentlichten Algorithmus vor um die Mappability von Genomen zu berechnen. Sie ist ein Maß für die Eindeutigkeit einer Sequenz. Sie wird berechnet, indem jede Subsequenz der Länge k im Genom selbst gesucht und ihre Vor-

kommen unter Berücksichtigung einer maximalen Anzahl von Fehlern gezählt wird. Wir zeigen zwei Anwendungen von Mappability auf. Zum einen, dass durch diese Informationen Read Mapper beim Suchen von Reads aus hoch repetitiven Regionen beschleunigt, und zum anderen wie phylogenetische Marker gefunden werden können. Innerhalb einer Menge von ähnlichen Stämmen der gleichen Spezies können so eindeutige Subsequenzen ermöglichen Stämme untereinander auf Grund von Sequenzierungsdaten zu unterscheiden.

Die Erkenntnisse dieser Arbeit können zahlreiche Anwendungen in der Bioinformatik verbessern. Dies zeigen wir anhand von Read Mappern und Mappability. Außerdem zeigen wir weitere Forschungsmöglichkeiten in diesem Bereich auf.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe. Ich erkläre weiterhin, dass ich die vorliegende Arbeit oder deren Inhalt nicht in einem früheren Promotionsverfahren eingereicht habe.

Berlin, 13. April 2019 _____

Christopher Maximilian Pockrandt