Approximate string matching for high-throughput sequencing

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

Enrico Siragusa

am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

Berlin 2014

Datum des Disputation: ***XX.XX.2014***

Gutachter:
Prof. Dr. Knut Reinert*, Freie Universität Berlin, Deutschland*
Prof. Dr. Raffaele Giancarlo*, Università degli Studi di Palermo, Italien*

Abstract

Over the last years, high-throughput sequencing (HTS) has become an invaluable method of investigation in molecular and medical biology. HTS technologies allow to sequence cheaply and rapidly an individual's DNA sample under the form of billions of short DNA reads. The ability to assess the content of a DNA sample at base-level resolution opens to a myriad of applications, including individual genotyping and assessment of large structural variations, measurement of gene expression levels and characterization of epigenetic features. Nonetheless, the quantity and quality of data produced by HTS instruments call for computationally efficient and accurate analysis methods.

In this thesis, I present novel methods for the mapping of high-throughput sequencing DNA reads, based on state of the art approximate string matching algorithms and data structures. Read mapping is a fundamental step of any HTS data analysis pipeline in resequencing projects, where DNA reads are reassembled by aligning them back to a previously known reference genome. The ingenuity of approximate string matching methods is crucial to design efficient and accurate read mapping tools.

In the first part of this thesis, I cover practical indexed and filtering methods for exact and approximate string matching. I present state of the art algorithms and data structures, give their pseudocode and discuss their implementation. Furthermore, I provide all implementations within SeqAn, the generic C++ template library for sequence analysis, which is freely available under `http://www.seqan.de/`. Subsequently, I experimentally evaluate all implemented methods, with the aim of guiding the engineering of new sequence alignment software. To the best of my knowledge, this is the first work providing a comprehensive exposition, implementation and evaluation of such methods.

In the second part of this thesis, I turn to the engineering and evaluation of read mapping tools. First, I present a novel method to find all mapping locations per read within a user-defined error rate. This method is published in the peer-reviewed journal Nucleic Acids Research and packaged in a open source tool nicknamed Masai. Afterwards, I present a refined method to greedily find all stratified read mapping locations up to a user-defined rate of suboptimality. This method, packaged in a tool called Yara, provides a more practical, yet sound solution to the read mapping problem. Extensive evaluation, both on simulated and real datasets, shows that Masai and Yara have better speed and accuracy than de-facto standard read mapping tools.

CONTENTS

CHAPTER

1 Introduction

The sequencing of the whole human genome has been one of the major scientific achievements of the last decades. In February 2001, the three billion dollars publicly funded *Human Genome Project* (HGP) published the first draft *covering more than 96 % of the euchromatic part of the human genome* [Consortium, 2001]. Concurrently, the privately funded company *Celera Genomics* published *a 2.91 billion base pair consensus sequence of the euchromatic portion of the human genome* [Venter *et al.*, 2001]. Other international sequencing consortiums, analogous to the HGP, contributed to characterize the entire genomes of several other model organisms, including mouse [Chinwalla *et al.*, 2002], nematode worms [Sulston *et al.*, 1992] and yeasts. In addition, Celera Genomics carried on the sequencing of fruit fly [Myers *et al.*, 2000] and mouse [Mural *et al.*, 2002].

The application of efficient computational methods has been crucial for the accomplishment of these projects. Whole genome sequencing projects used the *Sanger method* [Sanger *et al.*, 1977] with *capillary electrophoresis*, a technology producing high fidelity DNA reads long 700 bp in average, at a throughput of about 150 *kilo base pairs per hour* (Kbp/h). Because of such technological limitation, sequencing of whole genomes had to be coupled with the *shotgun* approach. Shotgun sequencing consists of chopping long DNA fragments up into smaller segments and then generating *reads* of these short nucleotide sequences. Then, it is left up to *bioinformatics* to reassemble these reads and produce a holistic representation of the original genome. At the pace of Sanger sequencing, it took years and hundreds of million dollars for these projects to complete. Nonetheless, their success did not mark the end of the sequencing era but its beginning.

1.1 High-throughput sequencing

Since then, sequencing technology steadily improved and evolved into what is now called *high-throughput sequencing* (HTS) or *next-generation sequencing* (NGS). In 2004, *454 Life Science* commercialized the *Genome Sequencer FLX*, an instrument based on large-scale parallel *pyrosequencing*, capable of sequencing DNA in form of 400 bp reads at a throughput of about 20 Mbp/h. High-throughput sequencing was born.

In 2006, Solexa released its *1G Genetic Analyzer*, based on a massively parallel technique of *reversible terminator-based sequencing*. The instrument produced reads as short as 30 bp with lower accuracy than Sanger sequencing but at very high-throughput: it allowed *resequencing* a human genome in three months for about $100,000. Following this

success, Solexa was acquired by *Illumina*, which is nowadays the market leader. At the beginning of 2014, Illumina announced the *HiSeq X Ten*, allowing in less than three days the sequencing of many whole human genomes at $1,000 each.

I return to sequencing technologies in chapter 5, where I give insights on the kind of data produced by HTS instruments, in order to understand the problems linked with their analysis. In the rest of this section, I continue to provide the context of this work. I first introduce most relevant *applications* of HTS and then explain how HTS *analysis pipelines* work. The reader familiar with HTS can jump directly to section 1.2, where I outline the structure of this work and state my contributions.

1.1.1 Protocols and applications

In the last years, HTS has become an invaluable method of investigation for computational molecular biologists. Abundant and cost-effective production of sequencing data permits viewing not only genomic DNA but also transcripts and epigenetic features at single-base resolution. For instance, DNA resequencing applications include genotyping and discovery of structural variations, the sequencing of RNA transcripts allows to assess gene expression as well as to analyse non-coding RNAs, epigenetic applications predict gene regulation according to methylation patterns and transcription factor binding-sites activity.

Whole genome and exome-seq

Whole genome sequencing (WGS, DNA-reseq) allows discovery of genetic variations across the whole genome. These may be in the form of single nucleotide variants (SNVs), small insertions or deletions (indels), or large structural variants such as transversions, translocations, and copy number variants (CNVs). *Whole exome sequencing* (WES, exome-seq) is a cost-effective, yet powerful alternative to WGS. This protocol consists in the *targeted sequencing* of the *exome*, i.e. the protein coding subset of a genome.

RNA-seq

RNA sequencing (RNA-seq) is a protocol to sequence the *transcriptome*, i.e. the set of RNA molecules of an organism, including mRNAs, rRNAs, tRNAs and other non-coding RNAs. Actual HTS technologies perform deep-sequencing of RNA molecules after they have been reverse-transcribed into cDNA. RNA-seq provides a myriad of information on gene expression, alternative splicing, intron/exon boundaries, untranslated regions (UTRs), and genetic variation with single-base accuracy.

ChIP-seq

Chromatin immunoprecipitation with next-generation sequencing (ChIP-seq) is a protocol that allows the selective sequencing of genomic DNA bound by a specific protein. The process isolates chromatin-protein complexes and sheares them by sonication. Subsequently, immunoprecipitation with a protein-specific antibody pulls down genomic DNA

sequences bound by a specific protein. These DNA sequences are finally sequenced by HTS. ChIP-seq is used to study the mechanisms of gene regulation. Through this sequencing protocol it is possible to determine global methylation patterns, identify transcription factor binding sites, histone modifications, and chromatin remodelling proteins.

1.1.2 Analysis pipelines

The analysis of HTS data consists of numerous steps, ranging from the initial instrument specific data processing to the final application specific interpretation of the results. It is conventional wisdom to subdivide such data analysis pipelines in three stages of *primary*, *secondary*, and *tertiary* analysis. The primary stage of analysis consists of instrument specific processes for generation, collection, and processing of raw sequencing data. The secondary stage reconstructs the original sequence of the donor genome by applying sequence analysis methods to the raw sequencing data. The tertiary stage characterizes features of the donor genome specific to the sequencing application e.g. genetic variations in exome-seq, then provides interpretations e.g. by aggregating multiple samples and performing data mining.

Primary analysis

Primary analysis consists of instrument specific steps to call base pairs and compute quality metrics. The base caller processes instrument signals and assigns a base to each peak, i.e. A, C, G, T, or N. The software assigns a quality value to each called base, estimating the probability of a base calling error. On early generation instruments, users could provide their own base calling tool. Now this process happens automatically on special hardware (e.g. FPGAs or GPUs) bundled within the instrument. The result of primary analysis is a standard *FASTQ* file containing DNA read sequences and their associated quality scores in *Phred* format (see section 5.1.1).

Secondary analysis

Secondary analysis aims at reconstructing the original sequence of the donor genome. There are two main *plans* to reassemble the original genome: (A) *de novo* assembly, and (B) *reference-guided* assembly (commonly called *read mapping*). *De novo* assembly is very involved as it essentially requires finding a *shortest common superstring* (SCS) of the reads, which is a NP-complete problem [Maier and Storer, 1977; Gallant *et al.*, 1980; Turner, 1989]. Computational methods in plan A first scaffold the reads by performing *overlap alignments* [Myers, 2005] or equivalently by constructing *de Bruijn graphs* [Pevzner *et al.*, 2001]. The knowledge of a *reference genome*, highly similar to the donor, simplifies the problem and opens the door to plan B. The reads are simply aligned to the reference, tolerating a few base pair errors. It is worth mentioning that some combinations of plans A and B have been proposed [Li, 2012].

Plan B is always preferred in resequencing projects, as it is computationally more viable than plan A and directly provides a way to assess genetic variation w.r.t. a reference genome. In this manuscript, I follow only plan B and present methods that work within

this specific plan. Nonetheless, many of the algorithmic components that I intriduce in the first part of this manuscript are ubiquitous in bioinformatics, thus applicable, if not already applied, to plan A. According to plan B, the secondary analysis step consists of three tasks: *quality control*, *read mapping*, and *consensus alignment*.

Quality control (Q/C) checks the quality of raw reads. Reads produced by current HTS technologies contain sequencing artefacts, in form of single miscalled bases or stretches of oligonucleotides. In order to circumvent this problem, various techniques have been developed, ranging from the simple *trimming* of low quality read stretches to sophisticated methods of *read error correction* [Weese *et al.*, 2013]. Sometimes, Q/C is simply omitted.

Read mapping takes care of aligning the reads to the reference genome. Read mappers adopt state of the art *approximate string matching* methods to efficiently analyze the deluge of data produced by HTS instruments. Approximate matching accounts for two kinds of errors: residual sequencing artefacts not removed by Q/C, and small genetic variations in the donor genome to which reads belong. Consequently, a read mapper must take into account errors when mapping reads.

Mapped reads, often stored in de-facto standard BAM files [Li *et al.*, 2009a], are sorted by genomic coordinate and eventually multiply aligned in order to construct a *consensus sequence* of the donor genome. The height of the pileup denotes the sequencing depth at each locus in the donor genome, thus the average height corresponds to the average sequencing coverage; higher depth implies more confidence in the consensus sequence of the donor genome and thus more accuracy in tertiary analysis.

Tertiary analysis

Tertiary analysis aims at interpreting information provided by secondary analysis. This pipeline stage groups a wide range of analyses specific to the sequencing application. In some pipelines, downstream analysis performs data mining over aggregated data coming from multiple samples.

Within DNA resequencing, *genotyping* consists of determining the variations between the donor and the reference genome. The result of *variant calling* is a set of variations characterizing the donor genome, usually stored in de-facto standard VFC files. Subsequently, *genome-wide association studies* (GWAS) associate genetic variants with phenotypic traits by examining *single-nucleotide polymorphisms* (SNPs), variants relatively common among individuals of the same population.

In RNA-seq, tertiary analysis consists of computing transcripts abundance by measuring *reads per kilobase per million mapped reads* (RPKM) [Mortazavi *et al.*, 2008]. Subsequently, relative gene expression is determined by comparing multiple RNA samples. In ChIP-seq, this stage consists of calling peaks in correspondence of mapped reads, determining which peaks identify feasible transcription factor binding sites, then interesting sites affecting gene regulation.

1.2 Outline

This thesis presents novel methods for the *efficient* and *accurate* mapping of high-throughput sequencing DNA reads, based on state of the art *approximate string matching* algorithms and data structures. Read mapping is a non-trivial, ubiquitous task in all resequencing applications. Efficiency is mandatory to keep the pace of sequencing technologies, exponentially increasing in throughput. Accuracy is required to enable downstream data analysis at single-base resolution. The ingenuity of state of the art approximate string matching methods is crucial for the design and implementation of efficient and accurate read mapping programs.

The contributions of this work are of purely practical interest, nonetheless I follow a rigorous approach. I subdivide this work in two parts. part I covers practical approximate string matching methods, whose interest is beyond HTS applications. part II describes the application of such methods to engineer two HTS read mapping programs developed by myself. In the two following sections, I describe the contents of part I and II.

1.2.1 Approximate string matching

In chapter 2, I introduce basic stringology concepts. I give an overview of the basic online, indexed and filtering methods for exact and approximate string matching. In particular, in section 2.4.2, I introduce the concept of full-text index and define a set of *generic* top-down traversal operations.

In chapter 3, I cover *indexed methods* for exact and approximate string matching. First, I describe some classic full-text indices (suffix arrays and q-gram indices) and succinct full-text indices (uncompressed FM-index variants). Afterwards, I introduce generic string matching algorithms, working on any of these data structures, and provide their experimental evaluation. To the best of my knowledge, this is the first work providing a comprehensive exposition of these methods together with their experimental evaluation. In addition, my implementation of all these algorithms and data structures is publicly available in source form within the C++ library SeqAn [Döring *et al.*, 2008].

In chapter 4, I cover *filtering methods* for approximate string matching. I consider two classes of full-sensitive filtering methods: those based on *seeds* and those based on *q-grams*. From the former class, I cover: *exact seeds* [Baeza-Yates and Perleberg, 1992], *approximate seeds* [Myers, 1994; Navarro and Baeza-Yates, 2000]. From the latter one: *contiguous q-grams* [Jokinen and Ukkonen, 1991], *gapped q-grams* [Burkhardt and Kärkkäinen, 2001], *multiple gapped q-grams* (also called *q-gram families*) [Kucherov *et al.*, 2005]. Again, to the best of my knowledge, this is the first work providing a comprehensive exposition of these methods together with their experimental evaluation. In addition, I introduce a formal framework for (multiple) gapped q-grams, which leads to the formulation of approximable (APX) and fully polynomial-time randomized approximation scheme (FPRAS) algorithms answering some combinatorially hard filter design questions.

1.2.2 Read mapping

In chapter 5, I return to read mapping. Before diving into the plethora of tools published during the last years, I give a quick overview of market-leading HTS technologies and introduce the *de facto* standard analysis paradigms adopted to analyze the data produced by such instruments. By reviewing some recent works [Derrien *et al.*, 2012; Lee and Schatz, 2012], I try to delineate which are the limits of these HTS technologies and data analysis paradigms. Finally, I give an overview of the most popular read mappers. In the last two chapters, I consider these popular programs in the evaluation of my own tools, Masai [Siragusa *et al.*, 2013] and Yara [**?**].

In chapter 6, I present my first attempt at engineering a read mapping tool. My method is packaged in a C++ tool nicknamed *Masai*, which stands for *m*ultiple backtracking of *a*pproximate seeds on a *s*uffix *a*rray *i*ndex. Masai is part of the SeqAn library [Döring *et al.*, 2008], it is distributed under the BSD license and can be downloaded from `http://www.seqan.de/projects/masai`. The result of this work has been published [Siragusa *et al.*, 2013] in the peer-reviewed journal *Nucleic Acids Research*.

In chapter 7, I present *Yara*, a state of the art read mapping tool. Yara is capable of quickly reporting all stratified mapping locations within a given error rate. The tool works with Illumina or Ion Torrent reads, supports paired-end and mate-pair protocols, computes accurate mapping qualities, offers parallelization via multi-threading, has a low memory footprint thanks to the FM-index, and does not require ad-hoc parameterization. Yara is the foremost contribution of this work.

Part I

APPROXIMATE STRING MATCHING

CHAPTER

2 Preliminaries

In this chapter, I introduce fundamental definitions and problems of stringology. The reader familiar with basic stringology can skip this chapter and proceed to chapter 3.

2.1 Definitions

Definition 2.1. An *alphabet* is a finite ordered set of symbols (or characters).

Definition 2.2. A *string* (or word) over an alphabet is a finite sequence of symbols from that alphabet. I denote the length of a string s by $|s|$, and by ϵ the empty string s.t. $|\epsilon| = 0$.

Definition 2.3. Given an alphabet Σ, the set $\Sigma^0 = \{\epsilon\}$ contains the empty string, Σ^n contains all strings of length n over Σ, and $\Sigma^* = \cup_{n=0}^{\infty}\Sigma^n$ all strings over Σ.

Definition 2.4. A string x is a *prefix* of another string y iff $x = y_{1...i}$ for some $1 \leq i \leq |y|$.

Definition 2.5. A string x is a *suffix* of another string y iff $x = y_{i...n}$ for some $1 \leq i \leq |y|$. For convenience, I often denote the suffix of y beginning at position i simply as $y_{i...}$.

Definition 2.6. A string x is a *substring* of another string y iff $x = y_{i...j}$ for some $1 \leq i \leq j \leq |y|$.

Definition 2.7. A string x *occurs* in another string y iff it is a substring of y. Given $x = y_{i...j}$, the occurrence of x starts at position i and ends at position j in y.

Definition 2.8. Given an alphabet Σ, I define the *distance function* $\delta : \Sigma \times \Sigma \to \{0, 1\}$ s.t. $\delta(a, b) = 1$ for any two distinct $a, b \in \Sigma$ and 0 otherwise.

Definition 2.9. Given an alphabet Σ of size σ, I denote by the function $\rho : \Sigma \to [1 ... \sigma]$ the *lexicographic rank* of any alphabet symbol, s.t. $\rho(a) < \rho(b) \iff a < b$ for any distinct $a, b \in \Sigma$.

Definition 2.10. The *lexicographical order* $<_{lex}$ between two non-empty strings x, y is defined as $x <_{lex} y \iff x_1 < y_1$, or $x_1 = y_1$ and $x_{2...} <_{lex} y_{2...}$

Definition 2.11. A *string collection* is an ordered multiset $\mathbb{S} = \{s^1, s^2, \ldots, s^c\}$ of non necessarily distinct strings over a common alphabet Σ. I denote by $\|\mathbb{S}\| = \sum_{i=1}^{c} |s^i|$ the total length of the string collection. I extend the notation of prefix, suffix and substring also to multisets, e.g. $\mathbb{S}_{(d,i)...(d,j)}$ denotes the substring $s^d_{i...j}$.

Definition 2.12. I call *terminator* a symbol $\$ \notin \Sigma$ s.t. $\rho(\$) < \rho(a)$ for any $a \in \Sigma$.

Definition 2.13. Given a string s over Σ, I call *padded string* the concatenation of s with a terminator symbol \$.

Definition 2.14. Given a string collection $\mathbb{S}$ over Σ, I call *padded string collection* the collection consisting of strings $s^i \in \mathbb{S}$ padded with terminator symbols $\$^i$ s.t. $\rho(\$^i) < \rho(\$^j) \iff i < j$.

Definition 2.15. Matrix notation D[:i]

2.2 Transcripts, alignments and distances

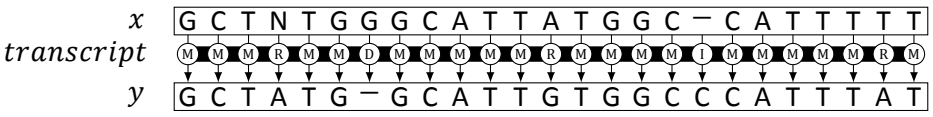
I now define basic edit operations to transform one string into another. Given two strings x, y of equal length n, the string x is easily transformed into the string y by substituting (or replacing) all symbols x_i s.t. $x_i \neq y_i$ into y_i, for $1 \leq i \leq n$. If the two strings have different lengths, some symbols from x must be necessarily inserted or deleted in order to obtain y. This fact motivates the following definition of *edit transcript*.

Definition 2.16. [Gusfield, 1997] An edit transcript for any two given strings x, y is a finite sequence of substitutions, insertions and deletions transforming x into y.

An *alignment* is an alternative way of visualizing a transformation between strings. While an edit transcript provides an explicit sequence of edit operations transforming one string into another, an alignment explicitly relates pairs of symbols from the two strings. Nonetheless, some symbols in one string are not related to any symbol in the other string, i.e. when some symbols are inserted or removed. For this reason, it is necessary to introduce an additional gap symbol $-$, not being part of the string alphabet Σ. The definition of alignment follows. Figure 2.1 shows an example of edit transcript with its associated alignment.

Definition 2.17. An alignment of two strings of length m, n over Σ is a string of length between $\min\{m, n\}$ and $m + n$ over the pair alphabet $(\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \setminus \{(-, -)\}$.

Figure 2.1: *Example of edit transcript and alignment. The string x is transformed into y. The transcript character M indicates a match, R a replacement, I an insertion, and D a deletion.*



At this point, I give the definition of two fundamental distance functions between strings.

Definition 2.18. [Hamming, 1950] The *Hamming distance* $d_H : \Sigma^n \times \Sigma^n \to \mathbb{N}_0$ between two strings $x, y \in \Sigma^n$ counts the number of substitutions necessary to transform x into y.

Definition 2.19. [Levenshtein, 1966] The *Levenshtein* or *edit distance* $d_E : \Sigma^* \times \Sigma^* \to \mathbb{N}_0$ between two strings $x, y \in \Sigma^*$ counts the *minimum* number of edit operations necessary to transform x into y.

The problem of finding an optimal alignment between two strings is equivalent to the problem of finding their minimum distance [Gusfield, 1997]. While the Hamming distance between any two strings of length n is easily computed in time $\mathcal{O}(n)$, computing the edit distance involves solving a non-trivial optimization problem.

2.3 Edit distance computation

The edit distance between two strings is efficiently computed via *dynamic programming* (DP). Let x, y be two strings of length $n \geq m$. The edit distance $d_E(x_{1\ldots i}, y_{1\ldots j})$ between any their prefixes $x_{1\ldots i}$ and $y_{1\ldots j}$ is defined recursively. The base conditions of the recurrence relation are:

$$d_E(\epsilon, \epsilon) = 0 \tag{2.1}$$

$$d_E(x_{1\ldots i}, \epsilon) = i \text{ for all } 1 \leq i \leq n \tag{2.2}$$

$$d_E(\epsilon, y_{1\ldots j}) = j \text{ for all } 1 \leq j \leq m \tag{2.3}$$

and the recursive case for all $1 < i \leq n$ and $1 < j \leq m$ is as follows:

$$d_E(x_{1\ldots i}, y_{1\ldots j}) = \min \begin{cases} d_E(x_{1\ldots i-1}, y_{1\ldots j}) & + \quad 1 \\ d_E(x_{1\ldots i}, y_{1\ldots j-1}) & + \quad 1 \\ d_E(x_{1\ldots i-1}, y_{1\ldots j-1}) & + \quad \delta(x_i, y_j) \end{cases} \tag{2.4}$$

Algorithm 2.1 computes the above recurrence relation in time $\mathcal{O}(nm)$ using a dynamic programming table D of $(n+1) \times (m+1)$ cells, where cell $D[i, j]$ stores the value of $d_E(x_{1\ldots i}, y_{1\ldots j})$. The sole edit distance without any alignment can be computed in space $\mathcal{O}(m)$, as only column $D[: j-1]$ is needed to compute column $D[: j]$. An optimal alignment can be computed in time $\mathcal{O}(m+n)$ via *traceback* on the table D: the traceback starts in the cell $D[m, n]$ and goes backwards (either left, up-left, or up) to the previous cell by deciding which condition of equation 2.4 yields the value of $D[m, n]$.

2.4 String matching

Exact string matching is one of the most fundamental problems in stringology.

Definition 2.20. [Gusfield, 1997] Given a string p of length m, called the *pattern*, and a string t of length n, called the *text*, the exact string matching problem is to find all occurrences of p into t.

This problem has been extensively studied from the theoretical standpoint and is well solved in practice [Faro and Lecroq, 2013]. Nonetheless, the definition of distance functions between strings lends to a more challenging problem: *approximate string matching*.

Definition 2.21. [Galil and Giancarlo, 1988] Given a text t, a pattern p, and a *distance threshold* $k \in \mathbb{N}$, the approximate string matching problem is to find all occurrences of p into t within distance k.

The approximate string matching problem under the Hamming distance is commonly referred as the *k-mismatches* problem and under the edit distance as the *k-differences* problem. Figure 2.2 shows an example of occurrence for k-differences. For k-mismatches and k-differences, it must hold $k > 0$ as the case $k = 0$ corresponds to exact string matching, and $k < m$ as a pattern trivially occurs at any position in the text if all its m characters are substituted. Frequently, the problem's input respects the condition $k \ll m \ll n$.

Definition 2.22. Under the edit or Hamming distance, the *error rate* is defined as $\epsilon = k/m$, with $0 < \epsilon < 1$ given the above conditions.

String matching problems are subdivided in two categories, *online* and *offline*, depending on which string, the pattern or the text, is given first. Algorithms for online string matching work by preprocessing the pattern and scanning the text from left to right (or right to left). Algorithms for offline string matching are instead allowed to preprocess the text, hence they build an index of the text beforehand to speed up subsequent searches. In practice, if the text is long, static and searched frequently, offline methods outperform online methods in terms of runtime, provided the necessary amount of memory for text indexing.

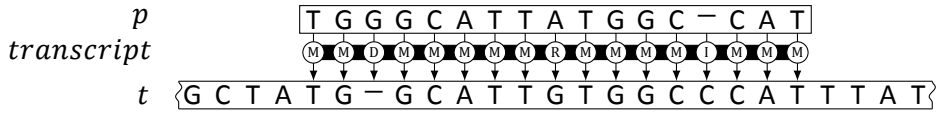
It goes without saying that offline string matching algorithms are tightly bound to text indexing data structures. Almost all of these algorithms require a *full-text index*, i.e. a data

Algorithm 2.1 EDITDISTANCE(x, y)

Input x : string of length n
y : string of length m
Output the edit distance $d_E(x, y)$ between x and y

1: $D[0, 0] \leftarrow 0$
2: **for** $j \leftarrow 1$ **to** m **do**
3: $\quad D[j, 0] \leftarrow D[j-1, 0] + 1$
4: **for** $i \leftarrow 1$ **to** n **do**
5: $\quad D[0, i] \leftarrow D[0, i-1] + 1$
6: $\quad$ **for** $j \leftarrow 1$ **to** m **do**
7: $\quad\quad D[i, j] \leftarrow \min\{ D[i-1, j] + 1, D[i, j-1] + 1, D[i-1, j-1] + \delta(t_i, p_j) \}$
8: **return** D[m,n]

Figure 2.2: *Example of occurrence for k-differences. Pattern p occurs in text t at edit distance 3, i.e. with a 19 % error rate. The alignment between p and any substring of t is called semi-global, as opposed to the global alignment of two complete strings.*



structure representing all substrings of the text. Very often, such *full-text index* is realized as the *suffix tree* [Weiner, 1973], a fundamental data structure in stringology. Among its virtues [Apostolico, 1985], the suffix tree natively provides exact string matching in optimal time and approximate string matching via backtracking [Ukkonen, 1993]. Often, the suffix tree finds its use within hybrid *filtering methods* rather than on its own.

Filtering methods first discard uninteresting portions of the text and subsequently verify only narrower areas. These methods work either online or offline. Online filtering methods try to jump over the text while scanning it; instead, offline filtering methods use an index to place anchors in the text. Both classes of filtering methods then require a native online method to verify the anchors.

Filtering methods outperform *native* online and indexed methods for a vast range of inputs, i.e. when the error rate is low, and are thus very appealing from a practical standpoint. Nonetheless, filtering methods are just opportunistic combinations of native online and indexed methods.

In this manuscript, I often consider *multiple* string matching problems, i.e. variants in which many patterns are given at once, instead of *single* problems where patterns are given one by one in an online fashion. Obviously, any method for the single case can solve the multiple case and vice versa. However, it is clear that multiple methods have an advantage over single methods. For instance, multiple online methods are allowed to preprocess all patterns and then scan the text only once, while single online methods have to scan the text every time a pattern is given. Thus, provided multiple patterns at once, multiple string matching methods are more appealing than single methods.

In the remainder of this section, I give a quick overview of the fundamental algorithms and data structures adopted by classic string matching methods. This overview serves as an introduction to the indexed and filtering methods presented in the next two chapters. For an extensive treatment of this subject, the reader is referred to complete surveys on online [Navarro, 2001] and indexed [Navarro *et al.*, 2001] approximate string matching methods.

2.4.1 Online methods

The DP algorithm 2.1 to compute the edit distance between two strings is easily turned into an online k-differences algorithm. Since an approximate occurrence of the pattern can start (and end) anywhere in the text, the problem involves computing the edit dis-

tance between the pattern and *any substring* of the text. Algorithm 2.2 efficiently solves this problem by computing the edit distance between the text and the pattern without penalizing leading and trailing deletions in the text.

Algorithm 2.2 KDIFFERENCES(t, p, k)

Input t : text string of length n
p : pattern string of length m
k : integer bounding the number of errors
Output all end positions of k-differences occurrences of p in t

1: $D[0,0] \leftarrow 0$
2: **for** $j \leftarrow 1$ **to** m **do**
3: $\quad D[j,0] \leftarrow j$
4: **for** $i \leftarrow 1$ **to** n **do**
5: $\quad D[0,i] \leftarrow 0$
6: $\quad$ **for** $j \leftarrow 1$ **to** m **do**
7: $\quad\quad D[i,j] \leftarrow \min\{D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+\delta(t_i,p_j)\}$
8: $\quad$ **if** $D[i,m] \leq k$ **then**
9: $\quad\quad$ **report** i

Consider the recurrence relation described by equations 2.1–2.4 and pose $x = t$ and $y = p$. Because an occurrence of the pattern can start anywhere in the text, the base condition 2.2 of the edit distance recurrence relation becomes:

$$d(x_{1\ldots i}, \epsilon) = 0 \text{ for all } 1 \leq i \leq n \tag{2.5}$$

and algorithm 2.2 initializes the top row $D[0\ :]$ of the DP matrix accordingly. Then, as an occurrence of the pattern can end anywhere in the text, algorithm 2.2 checks any cell in the bottom row for the condition $D[i,m] \leq k$.

2.4.2 Indexed methods

I now introduce *suffix tries*, idealized data structures to index full texts. I define a set of generic operations to traverse them in a top-down fashion. In chapter 3, I introduce various data structures replacing suffix tries in practice. The generic traversal operations here introduced lead to the formulation of generic algorithms that solve string matching problems on any of these data structures.

Trie

Consider a padded string collection $\mathbb{S}$ (definition 2.14) consisting of c strings. Note that padding is necessary to ensure that no string $s^i \in \mathbb{S}$ is a prefix of another string $s^j \in \mathbb{S}$.

Definition 2.23. The *trie* $\mathcal{S}$ of $\mathbb{S}$ is a lexicographically ordered tree data structure having one node designated as the root and c leaves, denoted as $l_1 \ldots l_c$, where leaf l_i points to string s^i. The edge entering any node of $\mathcal{S}$ is labeled with a symbol in Σ, while the edge entering any leaf of $\mathcal{S}$ is labeled with a terminator symbol. Any path from the root to a leaf l_i spells the string s^i, including its terminator symbol $\$^i$.

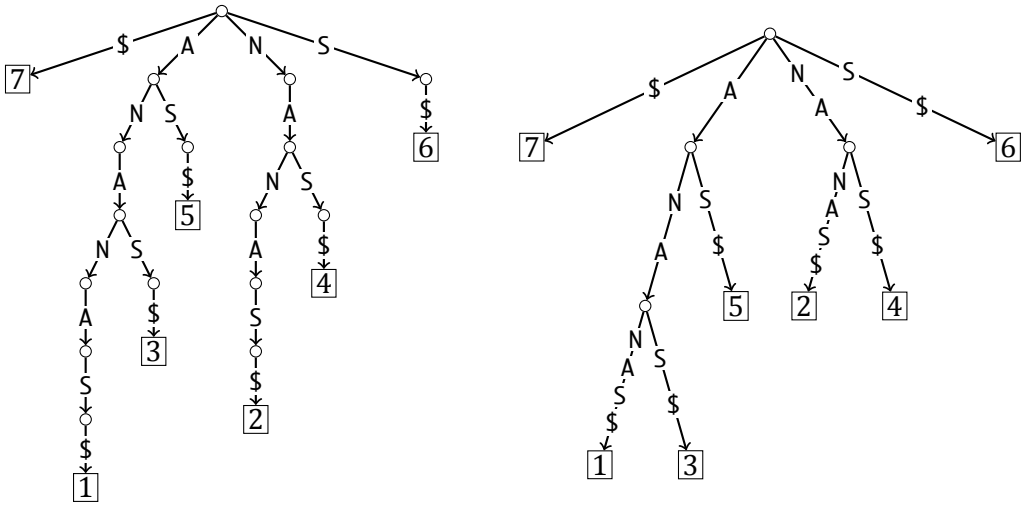
Suffix trie

Definition 2.24. Given a padded string s (definition 2.13) of length n, the suffix trie $\mathcal{S}$ is the trie of all suffixes of s. The *suffix trie* $\mathcal{S}$ has n leaves, where leaf l_i points to suffix $s_{i..n}$.

Definition 2.25. Given a padded string collection $\mathbb{S}$ (definition 2.14), any leaf of the *generalized suffix trie* is labeled with a pair (i, j) where i points to the string $s^i \in \mathbb{S}$ and $1 \leq j \leq n_i$ points to one of the n_i suffixes of s^i. Thus each path from the root to a leaf (i, j) spells the suffix $\mathbb{S}_{(i,j)\ldots}$ (definition 2.11).

Note that the (generalized) suffix trie here defined contains $\mathcal{O}(n^2)$ nodes, yet I only consider the suffix trie as an idealized data structure to formulate generic algorithms. The optimal data structure to represent in linear space all suffixes of a given string is the *suffix tree* [Morrison, 1968]. However, the suffix tree comes with the restriction that internal nodes must have more than one child and with the property that edges can labeled by strings of arbitrary length (see figure 2.3). This latter property slightly complicates the exposition of string matching algorithms but it does not affect their runtime complexity nor their result. For this reason, in the remainder of this manuscript, I consider w.l.o.g. always suffix tries instead of suffix trees.

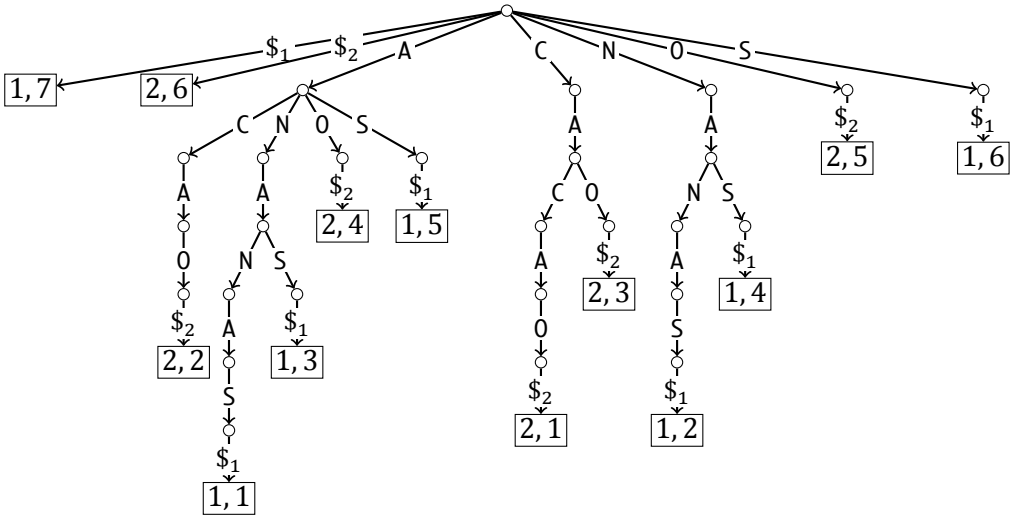
Figure 2.3: *Suffix trie and suffix tree of the string* `ANANAS$` *.*



Top-down traversal

I define a set of generic operations to traverse tries in a top-down fashion. Given a trie $\mathcal{T}$, I define the following operations inspecting any pointed node x of $\mathcal{T}$:

Figure 2.4: *Generalized suffix trie of the string collection* $\mathbb{S} = \{$ ANANAS$\$_1$, CACAO$\$_2$ $\}$.



- ISROOT(x) returns true iff the pointed node is the root;
- ISLEAF(x) returns true iff all outgoing edges are labeled by terminator symbols;
- LABEL(x) returns the symbol labeling the edge entering x;
- OCCURRENCES(x) returns the list of positions pointed by leaves below x;

Moreover, I define the following operations moving from pointed node x, and returning true on success and false otherwise:

- GODOWN(x) moves to the lexicographically smallest child of x;
- GODOWN(x, c) moves to the child of x whose entering edge is labeled by c;
- GORIGHT(x) moves to the lexicographically next child of x;
- GOUP(x) moves to the parent node of x.

Time complexities of the above operations depend on the data structure implementing the trie. Usually LABEL is $\mathcal{O}(1)$, both variants of GODOWN and GORIGHT can be $\mathcal{O}(1)$ or logarithmic in n, OCCURRENCES can be linear in the number of occurrences, GOUP is $\mathcal{O}(1)$ but with an additional $\mathcal{O}(n)$ space complexity to stack all parent nodes. Algorithms 2.3 and 2.4 show how to implement respectively GODOWN and GORIGHT using GODOWN a symbol and GOUP, although with a worst-case time complexity of $\mathcal{O}(\sigma)$. In chapter 3, I consider various data structures to implement suffix tries, I show how to implement these operations and give their complexities.

2.4.3 Filtering methods

Filtering methods work in two stages: the *filtration stage* discards portions of the text unlikely or unable to contain an occurrence of the pattern, subsequently the *verification stage* checks the remaining portions. The filtration stage proceeds online by scanning

Algorithm 2.3 GODOWN(x)

Input x : pointer to a suffix trie node
Output boolean indicating success
1: **if** ISLEAF(x) **then**
2: **return false**
3: **if** GODOWN($x, \min_{lex}\Sigma$) **then**
4: **return true**
5: **else**
6: **return** GORIGHT(x)

Algorithm 2.4 GORIGHT(x)

Input x : pointer to a suffix trie node
Output boolean indicating success
1: **if not** ISROOT(x) **then**
2: **while** $c \leftarrow \text{next}_{lex}\Sigma$, LABEL($x$) **do**
3: GOUP(x)
4: **if** GODOWN(x, c) **then**
5: **return true**
6: **return false**

the text or alternatively offline using an index of the text. The verification stage uses a conventional method, e.g. the online dynamic programming method or some variation of it. The crux of filtering methods is thus to accurately and efficiently classify text portions as containing or not an occurrence of the pattern.

Specificity and sensitivity

Any filtering method is thus a binary classification method. Any text location is *true* if it coincides with the beginning of an occurrence of the pattern and *false* if it does not. The outcome of the classification method is *positive* for text locations filtered in and *negative* for locations filtered out. Therefore, as shown in table 2.1, any text location belongs either to the set of *true positives* (TP), *false positives* (FP), *true negatives* (TN), or *false negatives* (FN). Standard *specificity* and *sensitivity* measure the accuracy of filtering methods. The specificity of a filter f on a text t is defined as:

$$\frac{|TN_f(t)|}{|TN_f(t)| + |FP_f(t)|} \tag{2.6}$$

and the sensitivity as:

$$\frac{|TP_f(t)|}{|TP_f(t)| + |FN_f(t)|} \tag{2.7}$$

Definition 2.26. A filter is *lossless* or *full-sensitive* if its sensitivity is 1, i.e. it produces no false negatives, otherwise it is *lossy*.

Many practical applications, e.g. read mapping, do not require strictly lossless filtration. Nonetheless, a predictable or controlled lossy filter helps to interpret the results and insure their quality. For this reason, I focus on criteria yielding filters which are lossless or lossy in a predictable fashion.

Efficiency

The total runtime of a filtering method is given by the sum of the runtimes of its filtration and verification stages. Filtration specificity determines how much time is spent in the

Table 2.1: *Classification of text locations by filtering methods.*

Occurrence	Filtered in	Filtered out
Yes	True positive (TP)	False negative (FN)
No	False positive (FP)	True negative (TN)

verification stage. Clearly, the number of false positives must be low to keep verification time small. Nonetheless, the filtration stage must also run in a reasonable amount of time. Hence, an efficient filtering method balances its runtime between filtration and verification. In any way, no filtering method can be efficient when the number of true positive locations approaches the text length.

Filtering methods for string matching work under the assumption that patterns occur in the text with a *low average probability*. For k-differences, the occurrence probability is a function of the error rate ϵ and the alphabet size σ, and can be computed or estimated under the assumption of the text being generated by a specific random source. Under the uniform Bernoulli model, where each symbol of Σ occurs with probability $1/\sigma$, Navarro [2001] experimentally finds that $\epsilon < 1 - 1/\sqrt{\sigma}$ is a tight upper bound on the error rate which ensures few occurrences and for which filtering algorithms are effective. For higher error rates, non-filtering methods, either online or offline, work better. On the one hand, the filtration efficiency bound for the DNA alphabet is 50 % error rate, while on the other hand HTS read mapping applications require a range of 3 to 10 % error rate. According to the above considerations, filtering methods are expected to be very efficient in HTS applications.

Seeds versus q-grams

Filtering methods apply combinatorial criteria to determine which portions of the text might contain some occurrence of the pattern. These criteria are in general valid for both online and offline variants of the problem. In practice, one specific criterion might be more convenient for one variant of the problem rather than the other. The combinatorial criterion underlying a filter is of paramount importance as it provides guarantees on filtration sensitivity.

In chapter 4, I consider two classes of combinatorial filtering methods: those based on *seeds* and those based on *q-grams*. Filters in the former class partition the pattern into *non-overlapping* factors called seeds. Application of the pigeonhole principle yields full-sensitive partitioning strategies. Instead, filters in the latter class consider all *overlapping* substrings of the pattern having length q, the so-called q-grams. Simple counting lemmata give lower bounds on the number of q-grams that must be present in a narrow window of the text, as necessary conditions for an approximate occurrence of the pattern.

CHAPTER

3 Indexed methods

Suffix trees are elegant data structures but they are rarely used in practice. Although suffix trees provides theoretically optimal construction and query time, their high space consumption prohibits practical applicability to large string collections. A practical study on suffix trees by Kurtz [1999] reports that efficient implementations achieve sizes between $12n$ and $20n$ bytes per character. For instance, two years before completing the sequencing of the human genome, Kurtz conjectured the resources required for computing the suffix tree for the complete human genome (consisting of about $3 \cdot 10^9$ bp) in 45.31 GB of memory and nine hours of CPU time, and concluded that *"it seems feasible to compute the suffix tree for the entire human genome on some computers"*.

One might be tempted to think that such memory requirements are not anymore a limiting factor as, at the time of writing, even standard workstations come with 32 GB of main memory. Indeed, over the last decades, the semiconductors industry followed the exponential trends dictated by Moores' law and yielded not only exponentially faster microprocessors but also larger memories. Unfortunately, memory latency improvements have been more modest, leading to the so called memory wall effect [Wilkes, 1995]: data access times are taking an increasingly fraction of total computation times. Thus, if Knuth [1973] wrote that *"space optimization is closely related to time optimization in a disk memory"*, forty years later one can simply say that space optimization is always related to time optimization.

Over the last years, significant effort has been devoted to the engineering of more space-efficient data structures to replace the suffix tree in practical applications. In particular, much research has been done into designing succinct (or even compressed) data structures providing efficient query times using space proportional to that of the uncompressed (or compressed) input. Thanks to these advances, a succinct index of the human genome consumes as little as 3.5 GB of memory and often even improves query time over classic indices.

In this chapter, I introduce some classic full-text indices (suffix arrays and q-gram indices) and subsequently succinct full-text indices (uncompressed FM-index variants). Afterwards, I introduce generic string matching algorithms that work on any of these data structures, and at the same time provide their experimental evaluation. My implementation of all these algorithms and data structures is publicly available in source form within the C++ library SeqAn [Döring *et al.*, 2008].

3.1 Classic full-text indices

3.1.1 Suffix array

The key idea of the suffix array (SA) [Manber and Myers, 1990] is that most information explicitly encoded in a suffix trie is superfluous for string matching. The explicit representation of suffix trie's internal nodes and outgoing edges can be omitted. Leaves pointing to the sorted suffixes are sufficient to perform exact string matching or even top-down traversals. Indeed, on the SA, any path from the root to an internal node is computed on the fly via binary search over the leaves. In this way, an additional logarithmic time complexity is paid to reduce space consumption by a linear factor. I formally define the (generalized) suffix array and later show how to emulate suffix trie traversals.

Definition 3.1. The *suffix array* of a padded string s of length n is an array A containing a permutation of the interval $[1, n]$, s.t. $s_{A[i]\ldots n} <_{lex} s_{A[i+1]\ldots n}$ for all $1 \leq i < n$.

Definition 3.2. The *generalized suffix array* (GSA) of a padded string collection $\mathbb{S}$ (definition 2.14), consisting of c strings of total length n, is an array A of length n containing a permutation of all pairs (i, j) where i points to a string $s^i \in \mathbb{S}$ and j points to one of the n_i suffixes of s^i. Pairs are ordered s.t. $\mathbb{S}_{A[i]\ldots} <_{lex} \mathbb{S}_{A[i+1]\ldots}$ for all $1 \leq i < n$.

The SA is constructed in $\mathcal{O}(n)$ time, for instance using the [Kärkkäinen and Sanders, 2003] algorithm, or using non-optimal but practically faster algorithms, e.g. [Schürmann

Figure 3.1: *(Generalized) suffix array. (a) Suffix array of the string* ANANAS$*. (b) Generalized suffix array of the string collection* $\mathbb{S} = \{$ ANANAS$_1$, CACAO$_2$ $\}$.

(a) *Suffix array.*

i	$A[i]$	$s_{A[i]\ldots n}$
1	7	$
2	1	ANANAS$
3	3	ANAS$
4	5	AS$
5	2	NANAS$
6	4	NAS$
7	6	S$

(b) *Generalized suffix array.*

i	$A[i]$	$\mathbb{S}_{A[i]\ldots}$
1	(1,7)	$$_1$
2	(2,6)	$$_2$
3	(2,2)	ACAO$$_2$
4	(1,1)	ANANAS$$_1$
5	(1,3)	ANAS$$_1$
6	(2,4)	AO$$_2$
7	(1,5)	AS$$_1$
8	(2,1)	CACAO$$_2$
9	(2,3)	CAO$$_2$
10	(1,2)	NANAS$$_1$
11	(1,4)	NAS$$_1$
12	(2,5)	O$$_2$
13	(1,6)	S$$_1$

Algorithm 3.1 L(x, c)

Input x : pointer to a suffix array node
c : character to query
Output integer denoting the left interval

1: $l_1 \leftarrow x.l$
2: $l_2 \leftarrow x.r$
3: **while** $l_1 < l_2$ **do**
4: $\quad i \leftarrow \lfloor \frac{l_1+l_2}{2} \rfloor$
5: $\quad$ **if** $\mathbb{S}_{A[i]+x.d} <_{lex} c$ **then**
6: $\quad\quad l_1 \leftarrow i + 1$
7: $\quad$ **else**
8: $\quad\quad l_2 \leftarrow i$
9: **return** l_1

Algorithm 3.2 R(x, c)

Input x : pointer to a suffix array node
c : character to query
Output integer denoting the right interval

1: $r_1 \leftarrow x.l$
2: $r_2 \leftarrow x.r$
3: **while** $r_1 < r_2$ **do**
4: $\quad i \leftarrow \lfloor \frac{r_1+r_2}{2} \rfloor$
5: $\quad$ **if** $\mathbb{S}_{A[i]+x.d} \leq_{lex} c$ **then**
6: $\quad\quad r_1 \leftarrow i + 1$
7: $\quad$ **else**
8: $\quad\quad r_2 \leftarrow i$
9: **return** r_1

and Stoye, 2007]. The space consumption of the suffix array is $n \log n$ bits. When $n < 2^{32}$, a 32 bit integer is sufficient to encode any value in the range $[1, n]$. Consequently, the space consumption of suffix arrays for texts shorter than 4 GB is $4n$ bytes.

Weese [2013] gives a generalization of Kärkkäinen and Sanders algorithm to construct the GSA in $\mathcal{O}(n)$ time. The space consumption of the GSA is $n \log cn^*$ bits, where $n^* = \max n_i$. For instance, the human genome GRCh37 is a collection of 24 sequences, among whose the largest one consists of 248 Mbp. Thus, the GSA of GRCh37 stores pairs of $1 + 4$ bytes and fits in 15 GB [Weese, 2013].

Top-down traversal

I now concentrate on describing suffix trie functionalities, as I implemented them within the SeqAn library. Any suffix trie node is univocally identified by an interval of the suffix array A. Thus, while traversing the trie, I maintain the interval $[l, r]$ associated to the current node. In addition, I also remember the depth d of the current node. The root node is represented by the interval $[1, n]$ containing the whole suffix array. The edge label entering any internal node at depth d is defined by the d-th symbol in any suffix $\mathbb{S}_{A[i]}$ with $i \in [l, r]$. Any leaf node is defined (see section 2.4.2) to have all outgoing edges labeled by terminator symbols. The occurrences below any node correspond by definition to the interval $[l, r]$ of A. Summing up, I represent the current node x by the integers $\{l, r, d\}$ and define the following operations on it:

- GOROOT(x) initializes x to $\{1, n, 0\}$;
- ISLEAF(x) returns true iff $A[x.r] + x.d = n_{A[x.r]}$;
- LABEL(x) returns $\mathbb{S}_{A[x.l]+x.d}$;
- OCCURRENCES(x) returns $A[x.l \ldots x.r]$.

Binary search is the key to implement function GODOWN a symbol. Functions L (algorithm 3.1) and R (algorithm 3.2) compute in $\mathcal{O}(\log n)$ binary search steps the position in

Algorithm 3.3 GODOWN(x)

Input x : pointer to a suffix array node
Output boolean indicating success

1: **if** ISLEAF(x) **then**
2: **return false**
3: $x.d \leftarrow x.d + 1$
4: $x.l \leftarrow \mathrm{R}(x, \epsilon)$
5: $c_l \leftarrow \mathbb{S}_{A[x.l]+x.d}$
6: $c_r \leftarrow \mathbb{S}_{A[x.r]+x.d}$
7: **if** $c_l \neq c_r$ **then**
8: $x.r \leftarrow \mathrm{R}(x, c_l)$
9: **return** $x.l < x.r$

Algorithm 3.4 GORIGHT(x)

Input x : pointer to a suffix array node
Output boolean indicating success

1: **if** ISROOT(x) **then**
2: **return false**
3: $c_l \leftarrow \mathbb{S}_{A[x.l]+x.d}$
4: $c_r \leftarrow \mathbb{S}_{A[x.r]+x.d}$
5: **if** $c_l \neq c_r$ **then**
6: $x.l \leftarrow x.r$
7: $x.r \leftarrow \mathrm{R}(x, c_l)$
8: **return** $x.l < x.r$

A of the left and right interval corresponding to the child node following the edge labeled by a given symbol c. Note that line 6 of algorithms 3.1–3.2 may involve a comparison beyond the end of strings in $\mathbb{S}$, hence I define t_i as the empty word ϵ if $i > |t|$ and $\epsilon <_{lex} c$ for all $c \in \Sigma$.

Algorithms 3.3 and 3.4 show how to implement respectively GODOWN and GORIGHT to exhibit a time complexity independent of the alphabet size σ. Indeed, the time complexity of GODOWN and GORIGHT is $\mathcal{O}(\log n)$, as they rely on a single call of R. In this way, the time complexity of exact string matching algorithm 3.11 is $\mathcal{O}(m \log n)$.

Note that the suffix array can be binary searched spelling a full pattern within a single call of L and R: in line 6 of algorithms 3.1–3.2, instead of comparing a single character, it suffices to compare the full pattern to the current suffix. However, the worst case runtime of algorithm 3.11 remains $\mathcal{O}(m \log n)$, as each step of the binary search now requires a full lexicographical comparison between the pattern and any suffix of the text, which takes $\mathcal{O}(m)$ time in the worst case. As shown in [Manber and Myers, 1990], the worst case runtime can be decreased to $\mathcal{O}(m + \log n)$ at the expense of additional $n \log n$ bits,

Algorithm 3.5 GODOWN(x, c)

Input x : pointer to a suffix array node
c : character to query
Output boolean indicating success

1: **if** ISLEAF(x) **then**
2: **return false**
3: $x.d \leftarrow x.d + 1$
4: $x.l \leftarrow \mathrm{L}(x, c)$
5: $x.r \leftarrow \mathrm{R}(x, c)$
6: **return** $x.l < x.r$

by storing the precomputed longest common prefixes (LCP) between any two consecutive suffixes $\mathbb{S}_{A[i]...}$, $\mathbb{S}_{A[i+1]...}$ for all $1 \leq i < n$. Alternatively, the *average case* runtime is reduced to $\mathcal{O}(m + \log n)$, without storing any additional information, by using the MLR heuristic [Manber and Myers, 1990]. In practice, the MLR heuristic outperforms the SA + LCP algorithm, due to the higher cost of fetching additional data from the LCP table [Weese, 2013].

3.1.2 Suffix tree realizations

I briefly introduce two suffix tree realizations from the literature: the enhanced suffix array (ESA) [Abouelhoda *et al.*, 2004] and the lazy suffix tree (LST) [Giegerich *et al.*, 2003]. These realizations explicitly implement or implicitly emulate a suffix tree rather than a suffix trie. The algorithms of section 3.3 work on tries, yet any tree can be easily traversed as a trie. The implementation of these data structures within SeqAn, equally generalized to multiple sequence, is due to Weese. Hence, for an extensive illustration, the reader is invited to consult [Weese, 2013].

Enhanced suffix array

The enhanced suffix array (ESA) [Abouelhoda *et al.*, 2004] supplements the SA and LCP tables (see section 3.1.1) with another table called *child* table. Each SA value represents one leaf of the suffix tree, while each LCP value represent the length of one edge of the suffix tree. What is still missing, in order to represent a full suffix tree, are the SA intervals of the children of each inner node. These intervals would have to be computed in logarithmic time by GORIGHT during a top-down SA traversal. As proposed by Abouelhoda *et al.* [2004], these intervals are computed in linear time, within one single bottom-up traversal, and stored in the child table, which consumes additional $n \log n$ bits, thus $4n$ bytes for collections smaller than 4 GB.

Lazy suffix tree

The lazy suffix tree (LST) [Giegerich *et al.*, 1999] variant proposed by [Weese, 2013] is composed by a partially sorted SA plus a node directory. The SA initially reflects the ordering of the suffixes up to depth 1, and the node directory table contains only the root node. During a top-down traversal, the current node at depth i is expanded by means of the *wotd*-algorithm [Giegerich *et al.*, 1999], which calls one round of radix sort to refine the ordering of the suffixes up to depth $i + 1$ and inserts the newly computed children nodes in the directory. The construction of the full LST takes $\mathcal{O}(n^2 + \sigma n)$ time in the worst case.

3.1.3 q-Gram index

The logarithmic factor introduced by the SA is removed by restricting the traversal of the idealized suffix trie to a maximum depth q. The idea is to supplement the SA with a so-

called q-gram directory: an additional array D of $\Sigma^q + 1$ integers, storing the suffix array ranges computed by algorithm 3.1 for any possible word of length q.

With the aim of addressing q-grams in the directory D, I impose a canonical code on q-grams through a bijective function $h : \Sigma^q \to [1, \sigma^q]$ defined as in [Knuth, 1973]:

$$h(p) = 1 + \sum_{i=1}^{q} \rho_0(p_i) \cdot \sigma^{q-i} \tag{3.1}$$

where $p \in \Sigma^q$ is any q-gram and ρ_0 is the zero-based lexicographic rank defined on Σ (recall the lexicographic rank function ρ from definition 2.9 and pose $\rho_0(x) = \rho(x) - 1$). The canonical code assigned by h preserves the lexicographical ordering for all words not longer than q, i.e. $v <_{lex} w$ iff $h(v) < h(w)$ for all $v, w \in \Sigma^{\leq q}$. The hash function h allows to store in and retrieve from D the left SA interval returned by algorithm 3.1 for each q-gram, i.e. $p \in \Sigma^q$, $D[h(p)] = \mathrm{L}(1, n, p)$. Note that the right interval returned by algorithm 3.2 is equivalent to the left interval of the lexicographical successor q-gram and therefore available in $D[h(p) + 1]$.

In practice, the q-gram index is applicable only to relatively small alphabets and tree depths. For instance, parameters $|\Sigma| = 4$ and $q = 14$ require a q-gram directory consisting of 268 M entries. Using a 32 bits integer encoding, the directory alone consumes 1 GB of memory.

Top-down traversal

I now describe how I extended the SA traversal operations of section 3.1.1 to use the q-gram directory D, within the generic text indexing framework of the SeqAn library. Again, while traversing the trie, I maintain the current range $[l, r]$ and the current depth d. In

Figure 3.2: *2-Gram index of the string* ANANAS$ *over the alphabet* $\Sigma = \{A, N, S\}$*. The example shows the lookup of the* 2*-gram* NA*. The hash value* $h(\mathrm{NA}) = 4$ *addresses a lookup in* $D[4 : 5]$ *that in turn provides the range* $[5, 6]$ *in A.*

p	$h(p)$	$D[h(p)]$	i	$A[i]$	$s_{A[i]\ldots n}$
AA	1	2	1	7	$
AN	2	2	2	1	ANANAS$
AS	3	4	3	3	ANAS$
NA	4	5	4	5	AS$
NN	5	6	5	2	NANAS$
NS	6	6	6	4	NAS$
SA	7	6	7	6	S$
SN	8	6			
SS	9	6			
	10	6			

addition, I maintain the interval $[l_h, r_h]$ in D and, in order to answer LABEL(x), the label e of the edge entering the current node. Summing up, I represent the current node x by the elements $\{l, r, d, l_h, r_h, e\}$. I define the basic node operations as follows:

- GOROOT(x) initializes x to $\{1, n, 0, 1, \sigma^q, \epsilon\}$;
- ISLEAF(x) returns true iff $x.d = q$;
- LABEL(x) returns $x.e$;
- OCCURRENCES(x) returns $A[x.l \,..\, x.r]$.

Algorithms 3.6 and 3.7 respectively show functions L and R using the directory D instead of A. In this way, both variants of GODOWN (algorithms 3.5 and 3.3) and GORIGHT (algorithm 3.4) take $\mathcal{O}(1)$ time.

If the top-down traversal is bounded to a maximum depth q, the directory D alone suffices. The suffix array A is accessed only to locate text locations pointed by the leaves. In this case, the total ordering of the text suffixes in the SA can be relaxed to prefixes of length q. This gives a twofold advantage, as one can (i) construct the SA more efficiently using bucket sorting and (ii) maintain leaves in each bucket sorted by their relative text positions. The latter property allows to compress the SA bucket-wise e.g. using Elias δ encoding [Elias, 1975] or to devise cache-oblivious strategies to process the occurrences [Hach *et al.*, 2010].

The q-gram index is still useful even if the traversal needs to go below depth q. An hybrid traversal can use the directory D up to depth q and later continue with binary searches on the suffix array A. This hybrid traversal cuts the most expensive binary searches and increases memory locality. Furthermore, this traversal becomes useful whenever the SA is too big to fit in main memory and has to reside in external memory.

3.1.4 Trie and radix tree realizations

Before turning to succinct full-text indices, I briefly describe how I reused some classic text-index implementations to implement also tries and radix trees in the SeqAn library. A trie is easily emulated by means of a partial SA. I index only the first suffix of each string in the collection and subsequently construct the SA-based trie via quicksort in time $\mathcal{O}(n \log n)$, where n is the cardinality of the collections. The top-down traversal based on binary search still works as described in section 3.1.1. This trie is also extendable by a q-gram directory as in section 3.1.3. A radix tree is constructed is an analogous way, starting

Algorithm 3.6 L(x, c)

Input x : pointer to a q-gram node
c : character to query
Output integer denoting the left interval

1: $x.l_h \leftarrow x.l_h + \rho_0(c) \cdot \sigma^{x.d}$
2: **return** $D[x.l_h]$

Algorithm 3.7 R(x, c)

Input x : pointer to a q-gram node
c : character to query
Output integer denoting the right interval

1: $x.r_h \leftarrow x.r_h - \rho_0(c) \cdot \sigma^{x.d}$
2: **return** $D[x.r_h]$

Algorithm 3.8 GODOWN(x, c)

Input x : pointer to a q-gram node
c : character to query
Output boolean indicating success

1: **if** ISLEAF(x) **then**
2: **return false**
3: $x.d \leftarrow x.d + 1$
4: $x.e \leftarrow c$
5: $x.l \leftarrow \mathrm{L}(x, x.e)$
6: $x.r \leftarrow \mathrm{R}(x, x.e)$
7: **return** $x.l < x.r$

from the LST of section **??**. I fill the LST's partial SA as described above and subsequently apply the *wotd*-algorithm [Giegerich *et al.*, 1999] constructs the radix tree in time $\mathcal{O}(\sigma n)$.

3.2 Succinct full-text indices

The *Burrows-Wheeler transform* (BWT) [Burrows and Wheeler, 1994] is a transformation defining a permutation of an input string. The transformed string exposes two important properties: *reversibility* and *compressibility*. The former property allows to reconstruct the original string from its BWT, while the latter property makes the transformed string more amenable to compression. Because of these two properties, the BWT is a fundamental method for text compression, practically used in the bzip2 tool [Seward, 1996].

More recently, Ferragina and Manzini have proposed the BWT as a method for full-text indexing. They show in [Ferragina and Manzini, 2000] that the BWT alone allows to perform exact string matching and engineer in [Ferragina and Manzini, 2001] a compressed full-text index called FM-index. Over the last years, the FM-index has been widely employed under different re-implementations by many popular bioinformatics tools e.g. Bowtie [Langmead *et al.*, 2009] and BWA [Li and Durbin, 2009], and is now considered a fundamental method for the indexing of genomic sequences.

In the following, I give the fundamental ideas behind the BWT. Subsequently, I discuss my succinct FM-index implementations covering strings and string collections.

3.2.1 Burrows-Wheeler transform

Let s be a padded string (definition 2.13) of length n over an alphabet Σ. In the following, consider the string s to be cyclic and its subscript s_i to be *modular*, e.g. $s_0 = s_n$ and $s_{n+i} = s_i$ for any $i \in \mathbb{N}$. Consider the square matrix consisting of all cyclic shifts of the string s sorted in lexicographical order, where the i-th cyclic shift has the form $s_{i\ldots n}s_{1\ldots i-1}$. Figure 3.3 shows an example. Note how the cyclic shifts matrix is related to the suffix array A of s: the i-th cyclic shift is $s_{A[i]\ldots n}s_{1\ldots A[i]-1}$.

Figure 3.3: *Cyclic shifts matrix of the string* ANANAS$. *Column* $s_{A[i]-1}$ *represents the Burrows-Wheeler transform.*

i	$A[i]$	$s_{A[i]}$	...	$s_{A[i]-1}$
1	7	$	ANANA	S
2	1	A	NANAS	$
3	3	A	NAS$A	N
4	5	A	S$ANA	N
5	2	N	ANAS$	A
6	4	N	AS$AN	A
7	6	S	$ANAN	A

Definition 3.3. The BWT of s is the string $s_{A[i]-1}$, i.e. the string obtained concatenating the symbols in the last column of the cyclic shifts matrix of s.

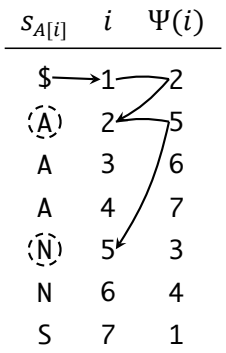
The BWT easily generalizes to string collections. Indeed, definition 3.3 still holds for a padded string collection $\mathbb{S}$ (definition 2.14) and its cyclic shifts matrix sorted in lexicographical order.

The cyclic shifts matrix is conceptual and does not have to be constructed explicitly to derive the BWT. The BWT can be obtained in linear time by scanning the suffix array A and assigning the symbol $s_{A[i]-1}$ to the i-th BWT symbol. However, constructing the BWT from the SA is still not desirable, especially for small alphabets, as the SA consumes $n \log n$ bits in addition to the $n \log \sigma$ bits of the BWT. Therefore, various direct BWT construction algorithms working within $o(n \log \sigma)$ bits plus constant space have been recently proposed in [Bauer *et al.*, 2013; Crochemore *et al.*, 2013].

Figure 3.4: *Functions LF and* Ψ *of the string* ANANAS$. *The example shows that* $LF = \Psi^{-1}$, *e.g.* $LF(\Psi(5)) = 5$ *and* $\Psi(LF(3)) = 3$. *Moreover, the example shows that the relative order of characters between* l *and* r *is preserved, e.g. the first occurrence of N in* l *corresponds to the first occurrence in* f.

i	$\Psi(i)$	$LF(i)$	$s_{A[i]}$	...	$s_{A[i]-1}$
1	2	7	$	ANANA	S
2	5	1	A	NANAS	$
3	6	5	A	NAS$A	N
4	7	6	A	S$ANA	N
5	3	2	N	ANAS$	A
6	4	3	N	AS$AN	A
7	1	4	S	$ANAN	A

Figure 3.5: *Recovering the string* ANANAS$ *from the permutation* Ψ*. The example shows only the first two steps of the inversion recovering* AN*.*



Inversion

I now describe how to invert the BWT to reconstruct the original text. For convenience, I denote the first column $s_{A[i]}$ by f and the last column $s_{A[i]-1}$ by l. Inverting the BWT means being able to know where any BWT character occurs in the original text. To this intent, I introduce two permutations $LF : [1,n] \rightarrow [1,n]$ and $\Psi : [1,n] \rightarrow [1,n]$, with $LF = \Psi^{-1}$, where the value of $LF(i)$ gives the position j in f where character l_i occurs and the value $\Psi(j)$ gives back the position i in l where f_j occurs. Figure 3.4 illustrates. I define the iterated Ψ as

$$\begin{aligned} \Psi^0(j) &= j \\ \Psi^{i+1}(j) &= \Psi(\Psi^i(j)) \end{aligned} \tag{3.2}$$

and the iterated LF as

$$\begin{aligned} LF^0(j) &= j \\ LF^{i+1}(j) &= LF(LF^i(j)). \end{aligned} \tag{3.3}$$

The character s_i is recovered as $f_{\Psi^{i-1}(j)}$, while $\bar{s}_i$ is recovered as $s_{LF^{i-1}(j)}$. The full string s is recovered by starting in f at the position of \$ and following the cycle defined by the permutation Ψ. Conversely, the reverse string $\bar{s}$ is recovered by starting in l at the position of \$ and following the cycle defined by the permutation LF. Figure 3.5 exemplifies.

Inverting the generalized BWT works in the same way. Indeed, permutations Ψ and LF are composed of c cycles, where each cycle corresponds to a distinct string in the collection. The character s^i is recovered by starting at the position of $\$^i$ and following the cycle of Ψ (or LF) associated to s^i.

Permutation LF

Permutation LF is conceptual: it is not necessary to encode it explicitly. Luckily, it is possible to deduce it from the BWT, with the help of some additional character counts. This is possible due to two simple observations on the cyclic shifts matrix.

Observation 3.1. *[Burrows and Wheeler, 1994] For all $i \in [1, n]$, the character l_i precedes the character f_i in the original string s.*

Observation 3.2. *[Burrows and Wheeler, 1994] For all characters $c \in \Sigma$, the i-th occurrence of c in f corresponds to the i-th occurrence of c in l.*

These observation are evident, indeed $f = s_{A[i]}$ and $l = s_{A[i]-1}$ (see figure 3.4). Given the two above observations, Ferragina and Manzini define the permutation LF as:

$$LF(i) = C(l_i) + Occ(l_i, i) \tag{3.4}$$

where $C : \Sigma \rightarrow [1, n]$ denotes the total number of occurrences in s of all characters alphabetically smaller than c, and $Occ : \Sigma \times [1, n] \rightarrow [1, n]$ the number of occurrences of character c in the prefix $l_{1...i}$. The key problem of encoding the permutation LF lies in representing function Occ, as function C is easily tabulated by a small array of size $\sigma \log n$ bits. In the next subsection, I address the problem of representing function Occ efficiently. Subsequently, I explain how to implement generic full-text index traversal using the permutation LF.

3.2.2 Rank dictionaries

The question *"how many times a given character c occurs in the prefix $l_{1...i}$?"*, has to be answered efficiently, ideally in constant time and linear space. The general problem on arbitrary strings has been tackled by several studies on the succinct representation of data structures [Jacobson, 1989]. This specific question takes the name of *rank query* and a data structure answering rank queries is called *rank dictionary* (RD).

Definition 3.4. Given a string s over an alphabet Σ and a character $c \in \Sigma$, $\text{rank}_c(s, i)$ returns the number of occurrences of c in the prefix $s_{1...i}$.

The key idea of RDs is to maintain a succinct (or even compressed) representation of the input string and attach a dictionary to it. By doing so, Jacobson shows how to answer rank queries in constant time (on the RAM model) using $n + o(n)$ bits for an input binary string of n bits [Jacobson, 1989]. Here, I cover only the most practical succinct RDs and discuss some implementation aspects, crucial to obtain practical efficiency. I first consider the binary alphabet $\mathbb{B} = \{0, 1\}$ and subsequently the case of an arbitrary alphabet.

Binary alphabet

Here, I follow the explanation of [Navarro and Mäkinen, 2007]. Hence, I start by describing a simple *one-level* rank dictionary answering rank queries in constant time but consuming $2n$ bits. Subsequently, I describe an extended *two-levels* RD consuming only $n + o(n)$ bits. In addition to that, I briefly discuss my implementation of practical *multi-level* RDs.

Figure 3.6: *Binary rank dictionaries (RDs) of the string $s = 010101100100$. (a) One-level RD with $b = 4$; in the example, $rank_1(s, 6) = R[2] + rank_1(s_{5\ldots8}, 2) = 3$. (b) Two-levels RD with $b = 2$; in the example, $rank_1(s, 6) = R^2[2] + R[3] + rank_1(s_{5\ldots8}, 1) = 3$.*

(a) *One-level rank dictionary.*

i	s_i	$R[\lfloor i/4 \rfloor]$
1	0	0
2	1	
3	0	
4	1	
5	0	2
6	1	
7	1	
8	0	
9	0	4
10	1	
11	0	
12	0	

(b) *Two-levels rank dictionary.*

i	s_i	$R[\lfloor i/2 \rfloor]$	$R^2[\lfloor i/4 \rfloor]$
1	0	0	0
2	1		
3	0	1	
4	1		
5	0	0	2
6	1		
7	1	1	
8	0		
9	0	0	4
10	1		
11	0	1	
12	0		

The binary one-level RD partitions the binary input string $s \in \mathbb{B}^*$ in blocks of b symbols and complements it with an array R of length $\lfloor n/b \rfloor$. The j-th entry of R provides a summary of the number of occurrences of the bit 1 in s before position jb, i.e. $R[1] = 0$ and $R[j] = \text{rank}_1(s, jb - 1)$ for any $j > 1$. R summarizes only rank_1, as $\text{rank}_0(s, i) = i - \text{rank}_1(s, i)$. Therefore, the rank query is rewritten as:

$$\text{rank}_1(s, i) = R[\lfloor i/b \rfloor] + \text{rank}_1(s_{\lfloor i/b \rfloor \ldots \lfloor i/b \rfloor + b}, i \bmod b) \tag{3.5}$$

and answered in time $\mathcal{O}(b)$ by (i) fetching the rank summary from R in constant time and (ii) counting the number of occurrences of the bit 1 within a block of $\mathcal{O}(b)$ bits. Figure 3.6a illustrates. Jacobson poses $b = \log n$ in order to answer step (ii) in time $\mathcal{O}(1)$ with the four-Russians tabulation technique [**?**]. As the array R stores $\lfloor n/\log n \rfloor$ positions and each position in s requires $\log n$ bits, R consumes n bits. Thus, the binary one-level RD consumes $2n$ bits.

A binary *two-levels* RD squeezes space consumption to $n + o(n)$ bits. The idea is to add another array R^2 summarizing the ranks on b^2 bits boundaries and let the initial array R store only local positions within the corresponding blocks defined by R^2. Accordingly, the rank query becomes:

$$\text{rank}_1(s, i) = R^2[\lfloor i/b^2 \rfloor] + R[\lfloor i/b \rfloor] + \text{rank}_1(s_{\lfloor i/b \rfloor \ldots \lfloor i/b \rfloor + b}, i \bmod b). \tag{3.6}$$

Figure 3.6b exemplifies. Each entry of R now represents only b^2 possible values and thus consumes only $2 \log b$ bits. Summing up, this RD consumes n bits for the input string,

$\mathcal{O}(\frac{n \log n}{b^2})$ bits for R^2 and $\mathcal{O}(\frac{n \log b}{b})$ bits for R. By posing $b = \log n$ as above, it follows $\mathcal{O}(\frac{n}{\log n})$ bits for R^2 and $\mathcal{O}(\frac{n \log \log n}{\log n})$ bits for R. Hence, the binary two-levels RD consumes $n + o(n)$ bits.

I implemented generic *multi-levels* RDs, where the block size b is a template parameter adjustable at compile time. Whenever the input string is smaller than 4 GB, I employ the one-level RD with $b = 32$ bits or the two-level RD with $b = 16$ bits and $b^2 = 32$ bits; otherwise, I employ the two-levels RD with $b = 32$ bits and $b^2 = 64$ bits, or the three-levels RD with $b = 16$ bits, $b^2 = 32$ bits and $b^3 = 64$ bits. In order to reduce the number of cache misses, the succinct representation of the input string is *interleaved* with the lowest level summaries array R. Moreover, I use the SSE 4.2 popcnt instruction [Intel, 2011] to count symbols within a block in time $\mathcal{O}(b/w)$, where w is the total SSE register width (on modern processors $w = 256$ bits).

Small alphabets

The extension of binary RDs to arbitrary alphabets is easy. Here, I show how to extend the one-level RD. However, the space consumption of such RD has a linear dependency in the alphabet size. This fact renders such extension appealing only for small alphabets, e.g. Σ_{DNA}.

Consider an input string s of length n over Σ, thus consisting of $n \log \sigma$ bits. As in the binary case, this one-level RD partitions s in blocks of b symbols. It complements the string s with a matrix R_σ of size $\lfloor n/b \rfloor \times \sigma$ entries, summarizing the number of occurrences for each symbol in Σ. The rank query is rewritten accordingly:

$$\text{rank}_c(s, i) = R_\sigma[\lfloor i/b \rfloor, \rho(c)] + \text{rank}_c(s_{\lfloor i/b \rfloor \ldots \lfloor i/b \rfloor + b}, i \bmod b). \tag{3.7}$$

Figure 3.7 shows an example of one-level DNA RD. Answering this query requires counting the number of occurrences of the character c inside a block of $\mathcal{O}(b)$ symbols. In order to answer this query in constant time, I consider blocks of $\lfloor \log n / \log \sigma \rfloor$ symbols, i.e. I pose $b = \log n$ bits as in the binary RD case. The matrix R_σ has thus $\lfloor n \log \sigma / \log n \rfloor \times \sigma$ entries, each one consuming $\log n$ bits. Thus, R_σ consumes $\sigma n \log \sigma$ bits and the whole RD $n \log \sigma(\sigma + 1)$ bits.

Wavelet tree

Grossi *et al.* propose a *hierarchical* RD, called the *wavelet tree* (WT), to mitigate the factor σ affecting the RD just exposed. This tree data structure recursively partitions the alphabet Σ in balanced subsets and accordingly decomposes the input string in *subsequences* containing symbols from one subset. Any tree node represents one alphabet partition and its associated subsequence. I first give the formal definition of WT and then discuss how to answer rank queries.

Definition 3.5. [Grossi *et al.*, 2003; Navarro and Mäkinen, 2007] The wavelet tree of a string $s \in \Sigma^*$ is a balanced binary tree of height $\lceil \log \sigma \rceil$. The root represents all symbols in Σ and each leaf exactly one symbol $c \in \Sigma$. Any non-leaf node v represents some subset of symbols Σ_v whose lexicographic rank is in range $[i, j]$ i.e. $\Sigma_v = \{c \in \Sigma : \rho(c) \in [i, j]\}$,

Figure 3.7: *One-level DNA rank dictionary of the string* $s = \texttt{CTCGCA}$ *with* $b = 2$*. In the example,* $rank_C(s, 4) = R_\sigma[2, 2] + rank_C(s_{3\ldots4}, 1) = 2$.

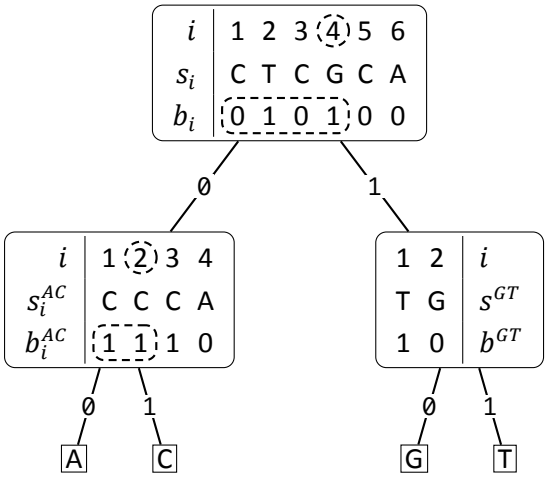
i	s_i	$R_\sigma[\lfloor i/2 \rfloor, \{\texttt{A, C, G, T}\}]$
1	C	[0, 0, 0, 0]
2	T	
3	C	[0, 1, 0, 1]
4	G	
5	C	[0, 2, 1, 1]
6	A	

its left child l represents the subset Σ_l of symbols in range $[i, \frac{i+j}{2}]$ while its right child r represents in Σ_r those in range $[\frac{i+j}{2} + 1, j]$. Node v implicitly represents the subsequence s^v of all symbols of s in Σ_v and explicitly encodes its decomposition as a binary string b^v s.t. $b_i^v = 0$ if $s_i^v \in \Sigma_l$ and 1 otherwise.

Any query $\text{rank}_c(s, i)$ is decomposed as a sequence of $\mathcal{O}(\log \sigma)$ binary rank queries. The sequence of queries starts in the root node and follows the path to the leaf corresponding to symbol c. On any non-leaf node v, the traversal goes left if c belongs to Σ_l, otherwise it goes right. Suppose w.l.o.g. that c belongs to Σ_l. The rank of symbol c in s^v is established as $\text{rank}_0(b^v, j)$, where j is the rank of c in the parent node or i in the root node. Figure 3.8 illustrates.

The WT encodes any binary string b^v associated to some non-leaf node v using a

Figure 3.8: *Wavelet tree of the DNA string* $s = \texttt{CTCGCA}$*. The alphabet* Σ_{DNA} *is recursively partitioned as* $\{\{A, C\}, \{G, T\}\}$*. In the example,* $rank_C(s, 4) = 2$ *is decomposed as* $rank_0(b, 4) = 2$ *on the root node and then* $rank_1(b^{AC}, 2) = 2$ *on the left inner node.*



separate binary RD. The WT contains $\lceil \log \sigma - 1 \rceil$ non-leaf levels and any such level encodes n bits overall. Using two-levels binary RDs (section 3.2.2), the WT consumes $(n + o(n)) \log \sigma$ bits, i.e. $n \log \sigma (1 + o(1))$ bits, and answers any rank query in time $\mathcal{O}(\log \sigma)$. At the same time, the WT does not need to store the original input string s of $n \log \sigma$ bits.

3.2.3 FM-index

I now turn to the problem of implementing a full-text index based on the permutation LF. First, I show how to emulate a top-down traversal of the suffix trie, which is sufficient to count the number of occurrences of any substring in the original text. Later, I focus on how to represent the leaves, which are necessary to locate occurrences in the original text.

Top-down traversal

Given a padded string collection $\mathbb{S}$, its associated permutation LF recovers substrings of $\bar{\mathbb{S}}$, i.e. substrings of $\mathbb{S}$ in *backward* direction. Nonetheless, the top-down traversal needs to recover substrings of $\mathbb{S}$ in *forward* direction, as the suffix trie $\mathcal{S}$ spells all forward substrings of $\mathbb{S}$. Therefore, I consider the BWT of $\bar{\mathbb{S}}$, such that LF recovers any substring of $\mathbb{S}$. To encode LF, I supplement the BWT of $\bar{\mathbb{S}}$ with its rank dictionary, either multi-levels or wavelet tree. In this way, the top-down traversal is able to use the permutation LF to decode SA intervals.

I represent the current node x by the elements $\{l, r, e\}$, where $[l, r]$ represents the current suffix array interval and e is the label of the edge entering the current node. Therefore, I define the following node operations:

- GOROOT(x) initializes x to $\{1, n, \epsilon\}$;
- ISLEAF(x) returns true iff $x.l_r = \$$;
- LABEL(x) returns $x.e$.

The traversal easily goes from the root node to its child node labeled by c: it suffices to derive the interval $[C(c), C(c+1)]$. Suppose the traversal is on an arbitrary node v of known interval $[b_v, e_v]$ s.t. the path from the root to v spells the substring s_v. Now, the traversal goes down to a child node w of unknown interval $[b_w, e_w]$ s.t. the path from the root to w spells $c \cdot s_v$ for some $c \in \Sigma$. The known interval $[b_v, e_v]$ contains all prefixes of $\bar{\mathbb{S}}$ ending with s_v, i.e. all suffixes of $\mathbb{S}$ starting with s_v, while the unknown interval $[b_w, e_w]$ contains all prefixes of $\bar{\mathbb{S}}$ ending with $c \cdot s_v$, i.e. all suffixes of $\mathbb{S}$ starting with $s_v \cdot c$. All these characters c are in $l_{b_v \ldots e_v}$, since l_i is the character $\mathbb{S}_{A[i]-1}$ preceding the suffix pointed by $A[i]$. Moreover, these characters c are *contiguous* and *in relative order* in f (see observations 3.1–3.2). If b and e are the first and last position in l within $[b_v, e_v]$ such that $l_b = c$ and $l_e = c$, then $b_w = LF(b)$ and $e_w = LF(e)$. Therefore $LF(b)$ becomes:

$$\begin{aligned} LF(b) &= C(l_b) + Occ(l_b, b) \\ &= C(c) + Occ(c, b) \\ &= C(c) + Occ(c, b_v - 1) + 1 \end{aligned} \tag{3.8}$$

Algorithm 3.9 GODOWN(x, c)

Input x : pointer to an FM-index node
c : char to query
Output boolean indicating success

```
1: if ISLEAF(x) then
2:     return false
3: x.l ← LF(x.l, c)
4: x.r ← LF(x.r, c)
5: x.e ← c
6: return x.l < x.r
```

and analogously $LF(e)$ becomes $C(c) + Occ(c, e_v)$.

Algorithm 3.9 implements operation GODOWN a symbol. This algorithm computes two values of permutation LF and thus runs in time $\mathcal{O}(1)$. Conversely, GODOWN and GORIGHT are provided by generic algorithms 2.3 and 2.4 running in time $\mathcal{O}(\sigma)$.

Sampled suffix array

The suffix array A is required to locate occurrences, yet it is not appealing to maintain the whole array. As proposed by Ferragina and Manzini, I maintain a *sampled* suffix array A^ϵ containing positions sampled at regular intervals in the input string. In order to determine if and where I sampled any $A[i]$ in A^ϵ, I maintain a binary rank dictionary S of length n: if $S[i] = 1$, then I sampled $A[i]$ in $A^\epsilon[rank_1(S, i)]$. I obtain any $A[i]$ by finding the smallest $j \geq 0$ such that $LF^j(i)$ is in A^ϵ, and then $A[i] = A[LF^j(i)] + j$.

By sampling one text position out of $\log^{1+\epsilon} n$, for some $\epsilon > 0$, then A^ϵ consumes $\mathcal{O}(\frac{n}{\log^\epsilon n})$ space and OCCURRENCES(x) returns all occurrences in $\mathcal{O}(o \cdot \log^{1+\epsilon} n)$ time [Ferragina and Manzini, 2000]. In practice, I sample text positions at rates between 2^{-3} and 2^{-5}. The rank dictionary S consumes $n + o(n)$ extra space, independently of the sampling rate.

3.3 Algorithms

In this section, I give generic string matching algorithms that use the traversal operations of section 2.4.2 and are thus valid for any of the data structures presented so far. I first consider a simple algorithm performing a top-down traversal bounded by depth. Then, I present algorithms for exact string matching and k-mismatches.I finally give, for the first time to the best of my knowledge, algorithms solving multiple variants of indexed exact string matching and k-mismatches. At the same time, I show the results of an experimental evaluation of each of these algorithms on various suffix trie implementations.

As data structures, I consider the suffix array (SA), the q-gram index with $q = 12$ (q-Gram), the FM-index with a two-levels DNA rank dictionary (FM-TL), the FM-index with

a wavelet tree composed of two-levels binary rank dictionaries (FM-WT). As text, I take the *C. elegans* reference genome (WormBase WS195), i.e. a collection of 6 DNA strings of about 100 Mbp total length. As patterns, I use sequences extrapolated from an Illumina sequencing run (Sequence Read Archive ID SRR065390).

All experiments run on a desktop computer running Linux 3.10.11, equipped with one Intel® Core i7-4770K CPU @ 3.50 GHz, 32 GB RAM and a 2 TB HDD @ 7200 RPM. The plots show always *average* runtimes per pattern, both in single and multiple string matching variants. Moreover, they consider only traversal times, while they exclude times to locate the patterns in the text by following leaf pointers, e.g. uncompressing SA values.

3.3.1 Construction

Table 3.1 shows index construction times and index memory consumption for all indices. The construction of all indices uses the DC7 algorithm [Dementiev *et al.*, 2008] in external memory. In addition, the construction of the ESA uses the algorithms in [Kasai *et al.*, 2001; Abouelhoda *et al.*, 2004], while the FM-WT construction follows [Grossi *et al.*, 2003]. The construction of the FM-indices adds a 15–18% additional runtime over the simple SA construction, the q-gram index directory construction adds only 3% runtime, while the more involved ESA's LCP and child tables 36%. As expected, the FM-TL and FM-WT are the most compact data structures, while the ESA is the most space inefficient one. For additional information on the SA and ESA construction algorithms and their runtimes, refer to [Weese, 2013].

Table 3.1: *Index construction runtime and index memory footprint.*

	SA	ESA	q-Gram	FM-WT	FM-TL
Time [s]	50.58	68.66	52.17	59.48	58.26
Memory [MB]	573.75	1338.73	637.75	119.56	119.55

3.3.2 Top-down traversal bounded by depth

Before turning to proper string matching algorithms, I present a simple algorithm that helps to comprehend subsequent backtracking algorithms. Algorithm 3.10 performs a top-down traversal of a suffix trie in depth-first order. The traversal is bounded, i.e. it stops after visiting any node at depth d.

The experimental evaluation shown in figure 3.9 provides a first glimpse on what are the practical performances of various suffix trie implementations. As expected, the WT FM-index is always slower than the TL FM-index. The q-gram index is never slower than the SA alone, however the contribution of the q-gram directory becomes insignificant for deep traversals.

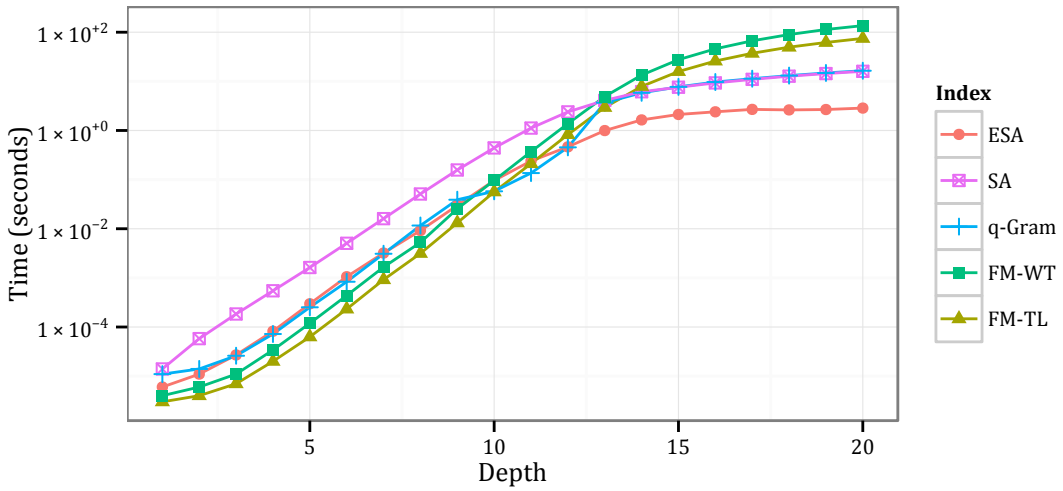
Depth 12 marks the turning point, as the indices become sparse. The TL FM-index is the fastest index up to depth 10, while the q-gram index is the fastest at depths 10–11. Below depth 12, the ESA (which is a tree) becomes significantly faster than all other trie indices. Conversely, the FM-indices (which are based on backward search) become up to two order of magnitude slower than the ESA.

Algorithm 3.10 DFS(x, d)

Input x : pointer to the root node of a suffix trie
d : integer bounding the traversal depth

```
1: if d > 0 then
2:     if GODOWN(x) then
3:         repeat
4:             DFS(x, d − 1)
5:         until GORIGHT(x)
```

Figure 3.9: *Runtime of top-down traversal of various suffix trie implementations.*



3.3.3 Exact string matching

I now give a simple algorithm performing exact string matching on a generic suffix trie. In the following, I assume the text t to be indexed by its suffix trie $\mathcal{T}$. Algorithm 3.11 searches the pattern p by starting in the root node of $\mathcal{T}$ and following the path spelling the pattern. If the search ends up in a node x, then each leaf l_i below x points to a distinct suffix $t_{i..n}$ such that $t_{i..i+m}$ equals p. If GODOWN is implemented in constant time and OCCURRENCES

in linear time, all occurrences of p into t are found in optimal time $\mathcal{O}(m + o)$, where m is the length of p and o its number of occurrences in t.

Figure 3.10 shows the results of the experimental evaluation of algorithm 3.11. On forward indices the search time becomes constant for patterns of length above 15, i.e. when the index becomes sparse. Conversely, on backward (FM) indices the search time is linear in the pattern length. The SA alone is at least 20 % slower than the q-gram index, hence never competitive. In particular, the SA shows a runtime peak for patterns of length 10, due to the fact that binary search algorithms 3.1–3.2 converge more slowly for shorter patterns. The ESA is never faster than the q-gram index despite its higher memory consumption. Concerning FM-indices, the WT variant is almost twice as slow as the TL variant, as the WT-based rank dictionary performs twice the number of random memory accesses than the levels rank dictionary. Summing up, the TL FM-index is the fastest index to match exact patterns within length 30, while the q-gram index is the fastest for patterns above length 30.

3.3.4 Backtracking k-mismatches

I now give an algorithm that solves k-mismatches by backtracking a generic suffix trie. The idea of backtracking a suffix tree has been first proposed in [Ukkonen, 1993]. Recently, various popular bioinformatics tools, e.g. Bowtie [Langmead *et al.*, 2009] and BWA [Li and Durbin, 2009], adopted variations of this method in conjunction with an FM-index. Yet, the idea dates back to more than twenty years ago.

Algorithm 3.12 performs a top-down traversal on the suffix trie $\mathcal{T}$, spelling incrementally all distinct substrings of t. While traversing each branch of the trie, this algorithm incrementally computes the distance between the query and the spelled string. If the computed distance exceeds k, the traversal backtracks and proceeds on the next branch. Conversely, if the pattern p is completely spelled and the traversal ends up in a node x, each leaf l_i below x points to a distinct suffix $t_{i..n}$ such that $d_H(t_{i..i+m}, p) \leq k$.

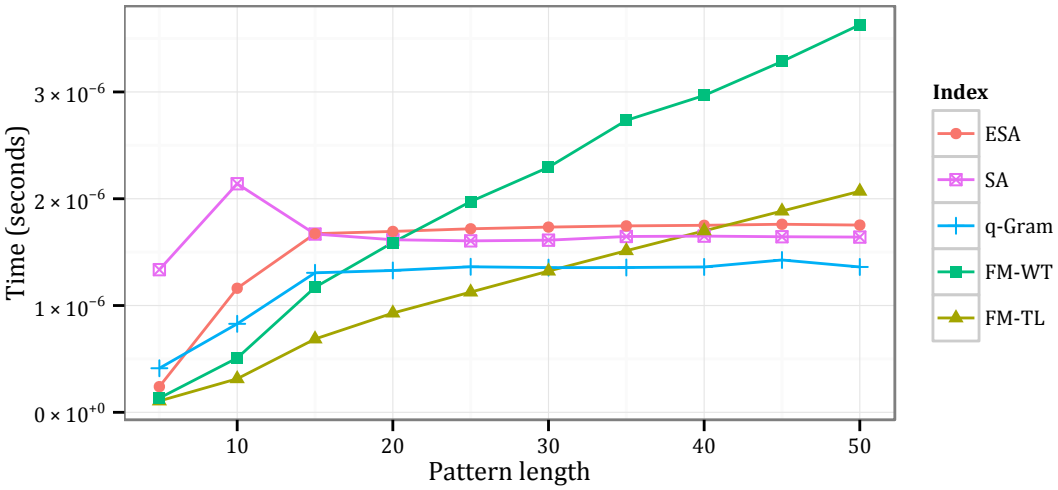
Figure 3.11 shows the results of the experimental evaluation of algorithm 3.12 for $k = 1$. The TL FM-index is always faster than any other index: for instance, on patterns of length 30, the SA it is 3 times slower; even the q-gram index is 50 % slower than the

Algorithm 3.11 ExactSearch(t, p)

Input t : pointer to the root node of the suffix trie of the text
p : pointer to the pattern
Output list of all occurrences of the pattern in the text

```
1: if atEnd(p) then
2:     report occurrences(t)
3: else if goDown(t, value(p)) then
4:     goNext(p)
5:     ExactSearch(t, p)
```

Figure 3.10: *Runtime of exact string matching on various suffix trie implementations.*



Algorithm 3.12 KMISMATCHES(t, p, k)

Input t : pointer to the root node of the suffix trie of the text
p : pointer to the pattern
k : integer bounding the number of mismatches
Output list of all occurrences of the pattern in the text

```
1:  if k = 0 then
2:      ExactSearch(t, p)
3:  else
4:      if atEnd(p) then
5:          report occurrences(t)
6:      else if goDown(t) then
7:          repeat
8:              d ← δ(label(t), value(p))
9:              goNext(p)
10:             kMismatches(t, p, k − d)
11:             goPrevious(p)
12:         until goRight(t)
```

TL FM-index. On the TL FM-index, 1-approximate matching of patterns of length 30 is 16 times slower than exact matching: on average, exact matching takes 1.3 microseconds (μs), while 1-approximate matching spends 21 μs.

3.3.5 Multiple exact string matching

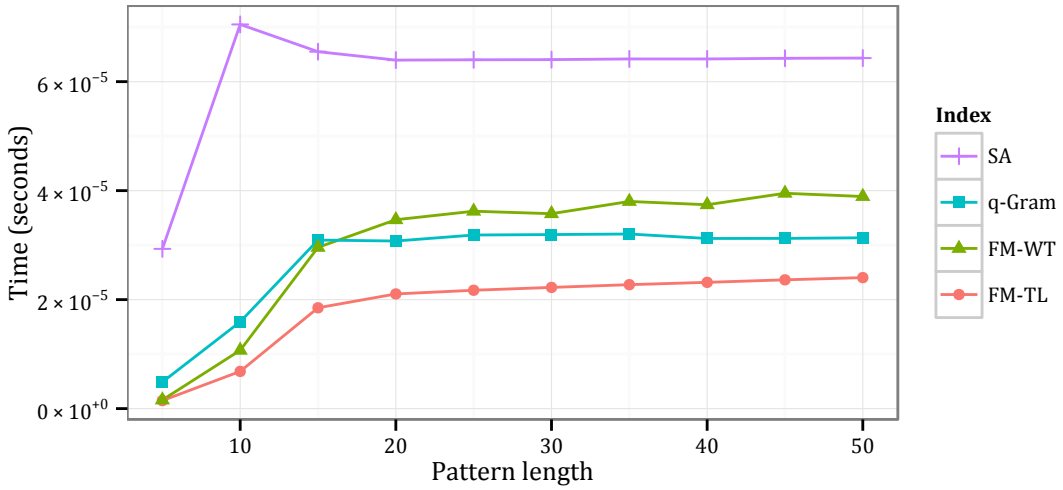
Before turning to multiple k-mismatches, I describe a simpler algorithm for multiple exact string matching. In addition to the text t, multiple exact string matching provides a collection of patterns $\mathbb{P}$. Hence, in addition to the suffix trie $\mathcal{T}$ of t, algorithm 3.13 considers the trie $\mathcal{P}$ of $\mathbb{P}$. Algorithm 3.13 matches simultaneously in $\mathcal{T}$ all patterns indexed in $\mathcal{P}$. The traversal performed by algorithm 3.13 visits pairs of nodes in $\mathcal{T} \times \mathcal{P}$ whose entering edges have the same label. Such traversal implicitly *intersects* the two tries. However, algorithm 3.13 is not symmetric: $\mathcal{T}$ and $\mathcal{P}$ cannot be interchanged. The traversal stops whenever it reaches a leaf node in $\mathcal{P}$ and reports the occurrences pointed by all the leaves beneath the current node in $\mathcal{T}$.

The experimental evaluation compares algorithm 3.13 (Multiple) with algorithm 3.11 processing patterns in random order (Single) and in lexicographic order (Sorted). Figure 3.12 shows the results. These three methods ran on 10 M patterns of length 30: runtimes shown in figure 3.12 (histogram Single) correspond to runtimes shown in figure 3.10 (plots at pattern length 15).

Figure 3.12 shows that a simple lexicographical sort of the patterns (histogram Sorted) speeds up algorithm 3.11 on the SA and ESA by a factor of 2. The same trick does not yield a significant speed-up on FM-indices nor on the q-gram index, as the q-gram directory already provides a cache local access pattern.

Algorithm 3.13 (histogram Multiple) further reduces the traversal time. Nonetheless, its runtime is dominated by the additional preprocessing time paid to construct the trie of the patterns. This algorithm becomes more useful within the multiple k-mismatches algorithm.

Figure 3.11: *Runtime of* 1*-mismatch search on various suffix trie implementations.*



Algorithm 3.13 MULTIPLEEXACTSEARCH(t, p)

Input t : pointer to the root node of the suffix trie of the text
p : pointer to the root node of the trie of the patterns
Output list of all occurrences of any pattern in the text

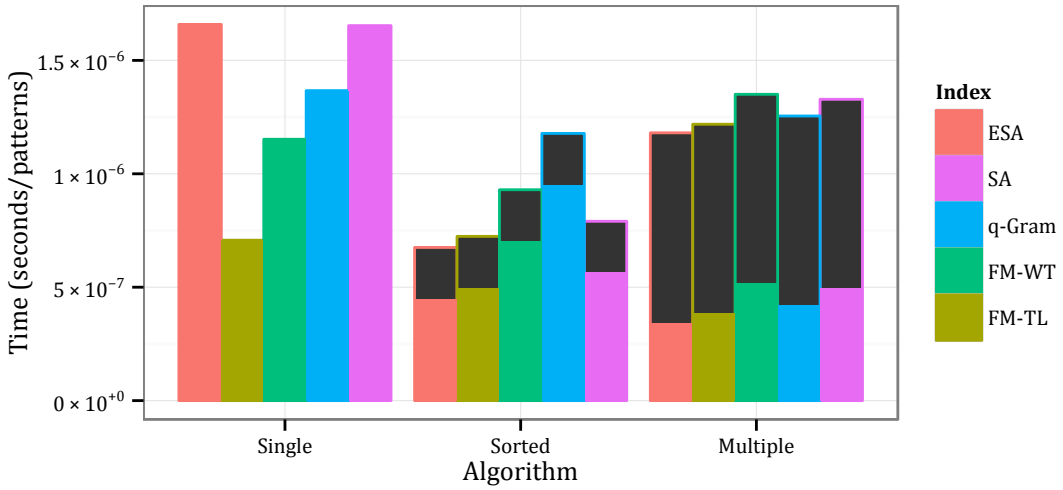
```
1: if ISLEAF(p) then
2:     report OCCURRENCES(t) × OCCURRENCES(p)
3: else
4:     GODOWN(p)
5:     repeat
6:         if GODOWN(t, LABEL(p)) then
7:             MULTIPLEEXACTSEARCH(t, p)
8:             GOUP(t)
9:     until GORIGHT(p)
```

3.3.6 Multiple k-mismatches

Algorithm 3.14 is the straightforward generalization of algorithm 3.13 to k-mismatches. The algorithm receives a collection of patterns $\mathbb{P}$ and performs backtracking on $\mathcal{T}$ as in algorithm 3.12, this time using the associated trie $\mathcal{P}$.

The experimental evaluation compares algorithm 3.14 (Multiple) with algorithm 3.12 processing patterns in random order (Single) and in lexicographic order (Sorted). All three methods ran on 10 M patterns of length 30, with k fixed to 1. Thus, runtimes shown

Figure 3.12: *Runtime of multiple exact string matching on various suffix trie implementations. Pattern length is fixed to 15. Preprocessing times are shown in black.*



in figure 3.13 (histogram Single) correspond to runtimes shown in figure 3.11 (plots at pattern length 30).

Algorithm 3.12 on lexicographically sorted patterns (histogram Sorted) is faster by a factor of 2 or more, on all indices. The time to sort the patterns becomes insignificant compared to the traversal time. Algorithm 3.13 (histogram Multiple) reduces traversal time by a factor of 5 on SA and ESA, thus the time to construct the trie of the patterns is easily justified. In practice, algorithm 3.13 fills the gap between the runtime of the SA and the q-gram index. Surprisingly, algorithm 3.13 increases traversal time on FM-indices.

This algorithms works according to a *cache-friendly* memory access pattern, which holds for forward search but not for backward search. Using forward search, the traversal of a suffix trie becomes less expensive as it proceeds towards bottom nodes. Indeed, traversal towards a child node involves the computation of a subinterval of the current suffix array interval; such computation accesses memory locations within the current interval, having good chances to be in the cache. Conversely, using backward search, the traversal becomes more expensive as it proceeds deeper in the trie; traversal downwards involves the computation of intervals outside of the current one, unlikely to be in the cache as they are accessed less often than top intervals. Multiple backtracking factorizes the traversal of top nodes, thus it pays off with forward search rather than with backward search.

Figure 3.14 shows the average runtime of the three approaches on the SA by varying the number of patterns. While the average runtime of the single method is constant, both multiple methods clearly benefit from receiving a higher number of patterns. In particular, method Multiple is constantly faster than Sorted, and the runtime gap increases with the number of patterns. The speed-up of multiple methods slowly decreases, though there is still some space of improvement with more than 10 M patterns.

Figure 3.15 presents the same evaluation of figure 3.14, but for the TL FM-index. Multiple methods exhibit again decreasing average runtimes by number of patterns. However, here method Sorted is constantly faster than Multiple, but the runtime gap decreases with the number of patterns. Moreover, the speed-up of both multiple methods slowly increases with the number of patterns instead of decreasing.

Algorithm 3.14 MULTIPLEKMISMATCHES(t, p, k)

Input t : pointer to the root node of the suffix trie of the text
p : pointer to the root node of the trie of the patterns
k : integer bounding the number of mismatches
Output list of all occurrences of any pattern in the text

```
1:  if k = 0 then
2:      MULTIPLEEXACTSEARCH(t, p)
3:  else
4:      if ISLEAF(p) then
5:          report OCCURRENCES(t) × OCCURRENCES(p)
6:      else if GODOWN(t) then
7:          repeat
8:              GODOWN(p)
9:              repeat
10:                 d ← δ(LABEL(t), LABEL(p))
11:                 MULTIPLEKMISMATCHES(t, p, k − d)
12:             until GORIGHT(p)
13:             GOUP(p)
14:         until GORIGHT(t)
```

Figure 3.13: *Runtime of multiple* 1*-mismatch on various suffix trie implementations. Pattern length is fixed to 30. Preprocessing times are shown in black.*

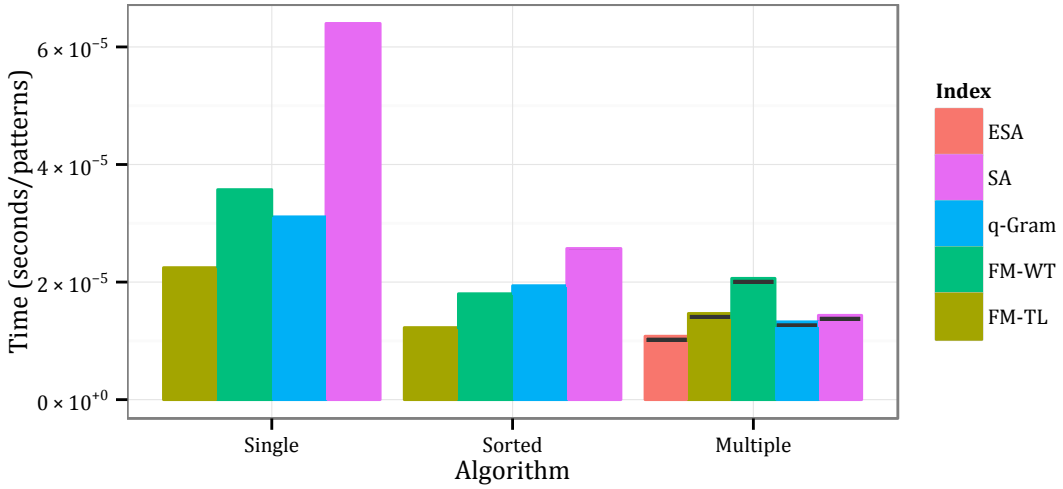


Figure 3.14: *Speed-up of multiple* 1*-mismatch by number of patterns on the SA. Pattern length is fixed to 30. Traversal times without preprocessing are shown by dashed lines.*

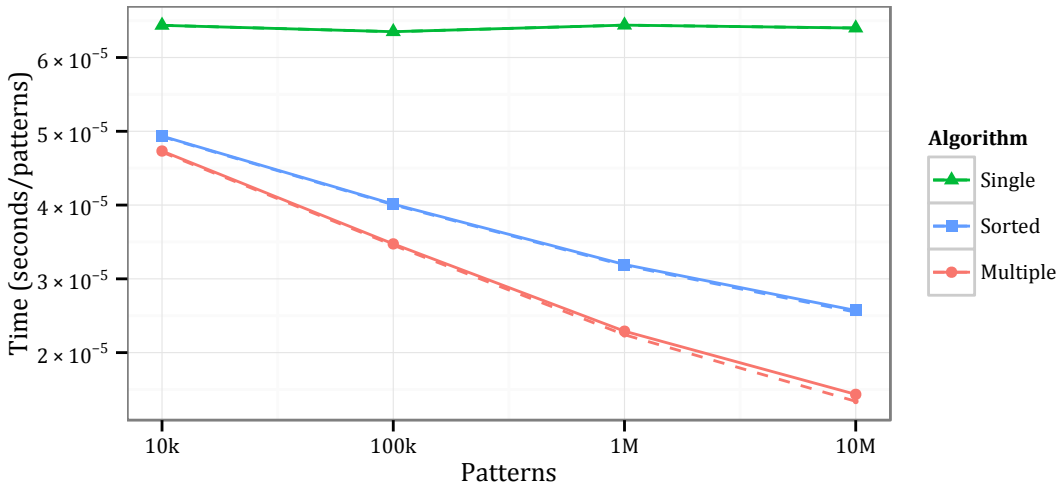
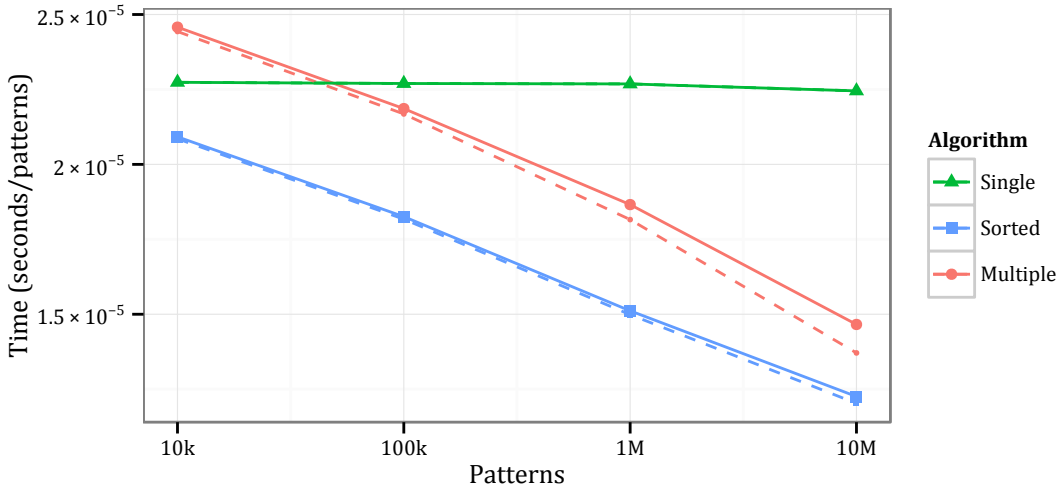


Figure 3.15: *Speed-up of multiple* 1*-mismatch by number of patterns on the TL FM-Index. Pattern length is fixed to 30. Traversal times without preprocessing are shown by dashed lines.*



CHAPTER

4 Filtering methods

In this chapter, I present various filtering methods for approximate string matching. I consider two classes of filtering methods: those based on *seeds* and those based on *q-grams*. Filters of the former class partition the pattern into *non-overlapping* factors called seeds, while filters of the latter class consider all *overlapping* substrings of the pattern having length q, the so-called q-grams. Both classes include various combinatorial filtering methods of increasing specificity and complexity, always providing filtration schemes with guarantees on filtration sensitivity.

I consider the following seed filtering methods: *exact seeds* [Baeza-Yates and Perleberg, 1992], *approximate seeds* [Myers, 1994; Navarro and Baeza-Yates, 2000]. Exact seeds partition the pattern in $k + 1$ non-overlapping seeds, to be searched exactly. Approximate seeds increase specificity by factorizing the pattern in less than $k + 1$ non-overlapping seeds, to be searched within a smaller distance threshold.

I consider the following q-gram filtering methods: *contiguous q-grams* [Jokinen and Ukkonen, 1991], *gapped q-grams* [Burkhardt and Kärkkäinen, 2001], *multiple gapped q-grams* (also called *q-gram families*) [Kucherov *et al.*, 2005]. Contiguous q-grams rely on a counting argument to filter out text regions containing less than a given threshold of q-gram occurrences. Gapped q-grams introduce *don't care positions* to lower the correlation between occurrences of consecutive q-grams. Multiple gapped q-grams conjunct multiple patterns of don't care positions to further increase specificity.

It will become clear through this chapter that seed filters are more practical, flexible, straightforward to design and implement than q-gram filters. All seed filters and contiguous q-grams provide full-sensitive filtration schemes for the k-differences problem, while (multiple) gapped q-grams only for k-mismatches. The design of highly specific yet full-sensitive filtration schemes for q-gram filters is combinatorially hard, while it is quite straightforward for seed filters. Also implementation-wise, q-gram filters are more involved than seeds filter. In fact, seed filters lend themselves well to both online and offline variants of the problem, while q-gram filters are better suited for the online variant. Finally, the experimental evaluation shows that seed filters outperform q-gram filters for most practical inputs. For these reasons, I design the applications of chapters 6 and 7 around seed filtering methods.

4.1 Exact seeds

Filtration with exact seeds is one of the naïvest filtering methods for approximate string matching. I first explain the underlying combinatorial principle, then I discuss implementation details and lastly give some insights on the efficiency of this method.

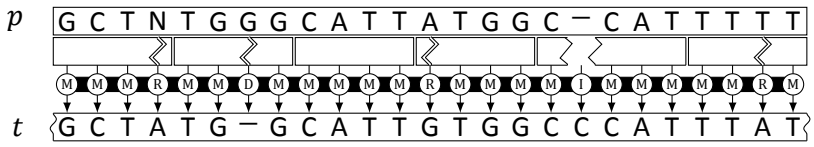
4.1.1 Principle

I consider the case of two arbitrary strings x, y within edit distance k. The generalization to k-differences is straightforward.

Lemma 4.1. *[Baeza-Yates and Perleberg, 1992] Let x, y be two strings s.t. $d_E(x, y) = k$. If y is partitioned w.l.o.g. into $k + 1$ non-overlapping seeds, then at least one seed occurs as a factor of x.*

It is immediate to see that any edit distance error can cover at most one seed. Therefore, at least one seed of y will not be covered by any seed and hence occur as a factor of x. Figure 4.1 shows an example.

Figure 4.1: *Filtration with exact seeds. Pattern p occurs in text t at edit distance 5. Filtration with 6 exact seeds detects this occurrence. In the example, all seeds are covered by one errors, except the third one.*



This filtering method reduces the approximate search into multiple smaller exact searches. It solves k-differences by partitioning the pattern into $k + 1$ seeds, searching all seeds in the text, and verifying a text window around each occurrence of any seed in the text. As lemma 4.1 is valid for *any substring* of the text within distance k from the pattern, this method finds all approximate occurrences of the pattern in the text.

4.1.2 Efficiency

The efficiency of filtration with exact seeds clearly depends on the number of verifications produced, as filtration time is always $\mathcal{O}(m)$. Thus, how many verifications does this filtering method trigger? It is straightforward to derive the expected number of verifications under the assumption of the text being generated according to the uniform Bernoulli model. I introduce the random variable C, counting the number of occurrences of a word in a text. The emission probability of any symbol in Σ is $p = \frac{1}{\sigma}$ and under i.i.d. assumptions the emission (and occurrence) probability of any word of length q is simply

$$\Pr(C > 0) = \frac{1}{\sigma^q} \tag{4.1}$$

thus the expected number of occurrences of a seed of length q in a text of length n is

$$E[C] = \sum_{i=1}^{n-q+1} \Pr(C > 0) = \frac{n-q+1}{\sigma^q} \leq \frac{n}{\sigma^q}. \tag{4.2}$$

Lemma 4.1 requires to partition the pattern into $k+1$ seeds but leaves the freedom to choose their length. This leads to the problem of finding an optimal pattern partitioning to minimize the expected number of verifications. I fix[1] the length of all seeds to be

$$q = \left\lfloor \frac{m}{k+1} \right\rfloor \tag{4.3}$$

to minimize the expected number of occurrences of any seed. Under these conditions, the expected number of verifications produced by filtration with exact seeds is

$$E[V] = E[C] \cdot (k+1) < \frac{n(k+1)}{\sigma^q}. \tag{4.4}$$

Nonetheless, inputs of practical interest like genomes and natural texts do not fit well the uniform Bernoulli model. On those texts, uniform seed length often leads to suboptimal filtration.

4.2 Approximate seeds

The simple analysis of section 4.1.2 shows that filtration specificity is strongly correlated to the seed length. Therefore, the crux of designing a stronger filter lies into increasing the seed length while maintaining the full-sensitivity constrains. Myers, subsequently followed by Navarro and Baeza-Yates, proposed *approximate seeds* as a practical and effective generalization of exact seeds, yielding stronger filters for k-differences. The key idea of filtration with approximate seeds is to reduce the approximate search into smaller approximate searches, as opposed to filtration with exact seeds that reduces into smaller exact searches.

4.2.1 Principle

Again, I start by considering two arbitrary strings x, y within edit distance k. The result then holds for any substring of the text within distance k from the pattern.

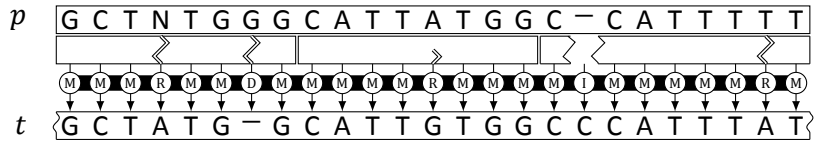
Lemma 4.2. *[Myers, 1994; Navarro and Baeza-Yates, 2000] Let x, y be two strings s.t. $d_E(x, y) = k$. If y is partitioned w.l.o.g. into s non-overlapping seeds s.t. $1 \leq s \leq k+1$, then at least one seed occurs as a factor of x within distance $\lfloor k/s \rfloor$.*

To prove full-sensitivity it suffices to see that, if none of the seeds occurs within its assigned distance, the total distance must be greater than $s \cdot \lfloor k/s \rfloor = k$. Figure 4.2 illustrates.

[1] For simplicity I ignore that some seed could have length $\left\lceil \frac{m}{k+1} \right\rceil$.

Approximate seeds provide filtration of variable specificity. The fastest but weakest filtration is given by $s = k + 1$, while the most specific filtration is obtained for $s = 1$ i.e. perfect filtration without any verification step. Alternatively, filtration specificity is controlled by acting on the minimum seed length q. Fixing q yields $s = \lfloor m/q \rfloor$, or vice versa, fixing the number of seeds s gives $q = \lfloor m/s \rfloor$. Filtration specificity is expected to increase with seed length.

Figure 4.2: *Filtration with approximate seeds. Pattern p occurs in text t at edit distance 5. A filtration scheme with thresholds $\mathbb{k} = (1, 1, 1)$ detects this occurrence. In the example, the left and right seeds are covered by two errors, but the central seed is covered only by one error and thus preserved.*



4.2.2 Filtration schemes

Lemma 4.2 assigns the same distance threshold to all seeds, yet this is not obligatory. Hence, I give a more general definition of *filtration scheme* for approximate seeds.

Definition 4.1. A seeds filtration scheme is an integer vector $\mathbb{k} = (k_1, \ldots, k_s)$, where integer $k_i \in \mathbb{N}_0$ represents the threshold assigned to the i-th seed.

Lemma 4.3. *Any filtration scheme $\mathbb{k} = (k_1, \ldots, k_s)$ s.t.*

$$s + \sum_{i=1}^{s} k_i > k \tag{4.5}$$

is full-sensitive for k-differences (and k-mismatches).

Example 4.1. The filtration schemes $(0, 0, 0, 0, 0)$, $(1, 1, 0)$, $(2, 1)$, (4) are full-sensitive for 4-differences. For instance, given a pattern of length $m = 100$, according to equation 4.3, q is respectively $20, 33, 50, 100$.

How to choose a *good* filtration scheme in practice? Myers, Navarro and Baeza-Yates carried out involved analysis to estimate the optimal parameterization. Navarro and Baeza-Yates find out that a number of seeds of $\Theta(\frac{m}{\log_\sigma n})$ yields an overall time complexity sublinear for an error rate $\epsilon < 1 - \frac{e}{\sqrt{\sigma}}$. Myers reports an analogous sublinear time when $q = \Theta(\log_\sigma n)$ is the seed length. Yet, these results do not necessarily translate into optimal filtration schemes in practice. The parameterization depends on the full-text index, the verification algorithm, the statistical properties of the text. Missing the optimal number of seeds by one often results in a runtime penalty of an order of magnitude.

Having established the number of seeds, or their length, thresholds have to be assigned. Lemma 4.3 allows to assign arbitrary distance thresholds. In practice, it is convenient to distribute distance thresholds evenly, as seeds with the highest threshold dominate the overall filtration time. The most strict threshold assignment is to give distance $\lfloor k/s \rfloor$ to $(k \bmod s) + 1$ seeds and distance $\lfloor k/s \rfloor - 1$ to the remaining seeds [Siragusa *et al.*, 2013].

4.3 Contiguous q-grams

q-Gram filters rely on counting arguments to filter out text regions containing less than a given threshold of q-gram occurrences. The first q-gram counting filter has been proposed in [Jokinen and Ukkonen, 1991]. More general filters have been proposed and implemented in *QUASAR* [Burkhardt *et al.*, 1999], *SWIFT* [Rasmussen *et al.*, 2006], *STELLAR* [Kehr *et al.*, 2011].

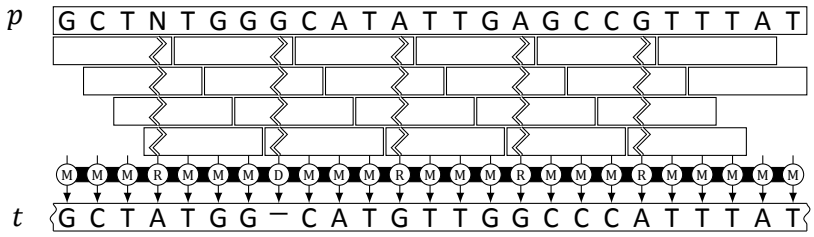
4.3.1 Principle

The counting argument of contiguous q-gram filters is based on the so-called *q-gram similarity measure* $\tau_q : \Sigma^* \times \Sigma^* \to \mathbb{N}_0$, defined as the number of substrings of length q common to two given strings. The following lemma relates q-gram similarity to edit distance. It gives a lower bound on the q-gram similarity $\tau_q(x, y)$ of any two strings x, y within edit distance k. As for seed filters, the result then easily translates to k-differences.

Lemma 4.4 (The q-gram lemma)**.** *[Jokinen and Ukkonen, 1991] Let x, y be two strings s.t. $d_E(x, y) = k$, and assume w.l.o.g. $|x| \leq |y|$ and $|x| = m$. Then x and y have q-gram similarity $\tau_q(m, k) \geq m - q + 1 - kq$.*

The first part of the threshold function τ counts the number of q-grams of x (i.e. $m - q + 1$), while the second part counts how many q-grams can be covered by k errors (i.e. at most q per error, hence kq in total). Figure 4.3 illustrates. The threshold function τ depends only on parameters (m, k) in addition to q, and not on the specific q-gram characters. Any pair (q, t) solving a (m, k) instance is a full-sensitive q-gram filtration scheme.

Figure 4.3: *Filtration with contiguous q-grams.*



4.3.2 Filtration schemes

Which is the biggest q-gram length yielding lossless filtration given m and k? In order to satisfy lemma 4.4, the q-gram threshold must be greater than zero, i.e. it must hold $\tau_q(m,k) \geq 1$. Thus, by substituting τ, it follows that the q-gram length must be $q \leq \left\lfloor \frac{m}{k+1} \right\rfloor$, analogously to equation 4.3 of seed filters.

However, the biggest q-gram length does not yield always the most specific filtration scheme. For instance, a threshold of 1 discards completely the counting argument of lemma 4.4 and makes filtration very unspecific in practice. Hence, on certain (m,k) instances, filtration schemes with non-optimal q-gram length potentially yields a more specific filtration. Example 4.2 shows alternative filtration schemes for a fixed (m,k) instance.

Example 4.2. Given a $(100,4)$ instance, the following (q,t) filtration schemes are full-sensitive: $(20,1)$, $(19,6)$, $(18,11)$.

4.3.3 Bucketing

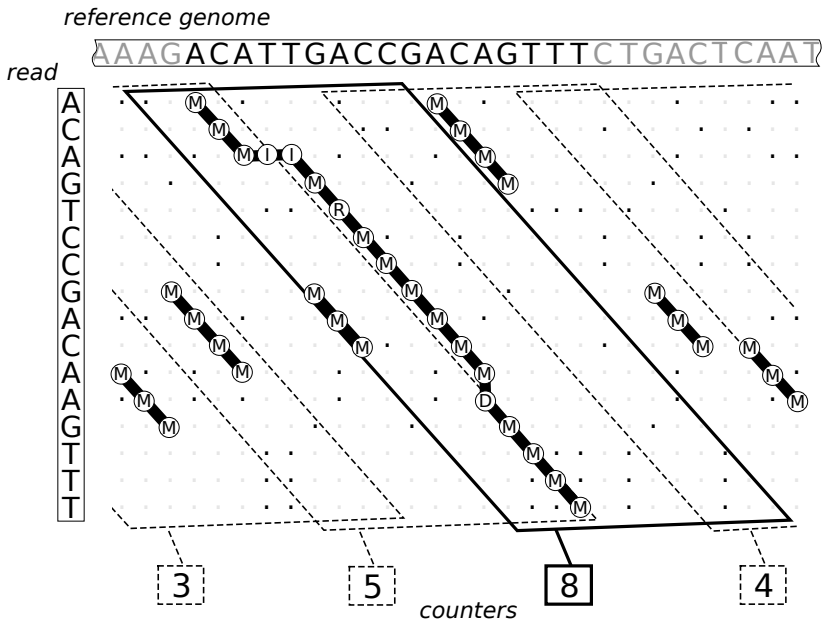
This filtering method requires *bucketing* the text in windows, in order to enforce the counting argument of lemma 4.4. Buckets are obtained by subdividing the implicit DP matrix in parallelograms and projecting them on the text. Figure 4.4 illustrates this concept: any approximate occurrence of the pattern in the text spans at most k diagonals and is thus enclosed inside a parallelogram of width $k+1$ [Rasmussen *et al.*, 2006]. Hence the projection of any text bucket has length $2k+1$ and any occurrence has length between $m-k$ and $m+k$. The implementations described in [Rasmussen *et al.*, 2006; Kehr *et al.*, 2011; Weese *et al.*, 2009] use more efficient bucketing strategies with larger, overlapping parallelograms.

This method lends itself to work in a multiple online fashion rather than offline. The filtration stage scans the text and counts how many q-grams of the pattern fall into each bucket. The verification stage then verifies only parallelograms exceeding threshold $\tau_q(m,k)$. As long as the filter scans the text, it is necessary to remember only buckets spanning the patterns' lengths. Conversely, the program QUASAR [Burkhardt *et al.*, 1999] uses a q-gram index of the text to speed up the filtration phase. However, this implementation requires more memory, as it must keep the text index in memory and bucket the whole text.

4.4 Gapped q-grams

The idea of *gapped q-grams* is to lower the correlation between consecutive q-grams. The occurrence of any contiguous q-gram is strongly correlated to the occurrences of its preceding and following q-grams. One single edit distance error affects a cluster of q consecutive q-grams, as evidenced by the q-gram lemma. Gapped q-grams hence define patterns of *don't care positions* to skip characters at fixed positions. Such don't care

Figure 4.4: *Parallelogram buckets. Picture from [Weese* et al.*, 2009].*



positions are immune to mismatches, but not to insertions and deletions. Hence, this generalization of contiguous q-grams solves k-mismatches but not k-differences.

Gapped q-grams rely on a generalization of the q-gram similarity measure (section 4.3) to *subsequences*. A subsequence is a non-contiguous sequence of symbols of a given string. Hence, instead of substrings, filtration with gapped q-grams counts the number of subsequences of length q common to two strings, whose positions are taken from a fixed set Q. The formal definition of gapped q-gram follows.

Definition 4.2. A Q-gram is a finite sequence Q of natural numbers starting with the unit element, i.e. $Q \subset \mathbb{N}$ and $1 \in Q$. The cardinality $|Q|$ is called the *weight* of Q and denoted as $w(Q)$. The maximum element of Q is named *span* and indicated by $s(Q)$.

Figure 4.5: *Filtration with gapped q-grams.*

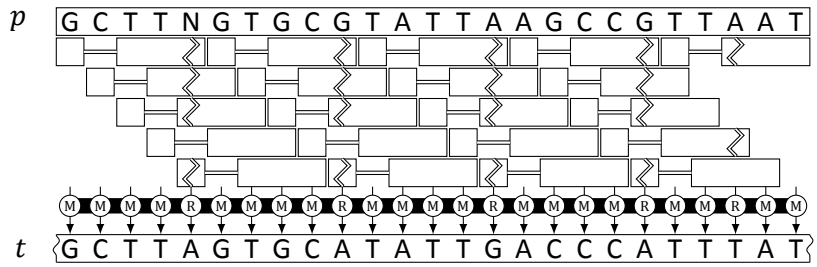


Figure 4.5 shows an example of gapped q-gram. As in the q-gram lemma, the threshold depends only on Q and parameters m, k. Indeed, the pattern of occurring q-grams

does not depend on the text or pattern sequences but only on their transcript, i.e. on the mismatch positions. Therefore, in the following, I refer to gapped q-gram filtration schemes (Q, t) solving (m, k) instances. As opposed to contiguous q-grams, don't care positions are not affected by mismatches. Thus any gapped q-gram potentially yields a higher threshold than the contiguous q-gram of same weight. Unfortunately, the q-gram lemma (4.4) does not give anymore a tight threshold, but only a lower bound.

Gapped q-grams raise hard combinatorial questions:

1. Does a gapped q-gram yield any filtration scheme solving an (m, k) instance?
2. If the answer to question 1 is yes:
   (a) Which is the maximum distance k^* for which full-sensitivity is guaranteed?
   (b) Which is the maximum threshold t^* that guarantees full-sensitivity?
3. If the answer to question 1 is no, which is the expected sensitivity of the lossy filtration scheme?
4. Which is the expected specificity of a filtration scheme?

Question 2b has been first considered in [Burkhardt and Kärkkäinen, 2001; Kucherov *et al.*, 2005], the more general question 1 has been introduced in [Nicolas and Rivals, 2005], while I consider here for the first time questions 2a, 3 and 4. With the aim of answering these questions, I first introduce simple characteristic functions to formally define transcripts detected by gapped q-grams. Afterwards, I recapitulate known results and present new exact and approximate solutions.

4.4.1 Characteristic functions

I consider any Hamming distance transcript σ as a m-dimensional vector over $\mathbb{B} = \{0, 1\}$. Accordingly, I denote by $|\sigma|_0 = m - \sum \sigma$ the Hamming distance of the transcript, and by $\mathbb{B}_k^m \subset \mathbb{B}^m$ the set containing all transcripts σ s.t. $|\sigma|_0 = k$. I now define the event of detection of a transcript by a filtration scheme (Q, t).

Definition 4.3. A Q-gram *occurs* at position i in a transcript σ iff $\forall j \in Q\ \sigma_{i+j} = 1$.

Definition 4.4. A (Q, t) filtration scheme *detects* σ iff the Q-gram occurs at least t times in σ.

Boolean functions

I introduce boolean functions to characterize the set of transcripts detected by filtration schemes of the form $(Q, 0)$ according to definition 4.4. The utility of these functions is manyfold. In section 4.4.4, I use them directly within an ILP that computes an exact solution to Optimal Threshold and thus answers Non Detection. In section 4.4.5, I reduce the estimation of filtration sensitivity to a counting problem on boolean functions.

Let $T_Q^m : \mathbb{B}^m \to \mathbb{B}$ denote a *boolean function* such that $T_Q^m(\sigma)$ is true iff the Q-gram occurs at least one time in a transcript σ of length m. I define such boolean function as

the disjunction

$$T_Q^m(\sigma) = \bigvee_{i=1}^{m-s(Q)+1} \bigwedge_{j \in Q} \sigma_{i+j} \tag{4.6}$$

where each *clause* of T_Q^m represents a single possible occurrence of Q in σ. According to definition 4.3, filtration scheme (Q, t) detects σ iff σ satisfies at least t clauses of T_Q^m. I define an analogous boolean function for a Q-gram family F as the disjunction

$$T_F^m(\sigma) = \bigvee_{Q_i \in F} T_{Q_i}^m(\sigma) \tag{4.7}$$

By definition, T_Q^m and T_F^m are *monotone nondecreasing* boolean functions in *disjunctive normal form* (*DNF*). Since all monotone boolean functions in DNF are minimal, T_Q^m and T_F^m are *minimal*.

Pseudo-boolean functions

I now consider pseudo-boolean functions, corresponding to functions 4.6 and 4.7, to associate a filtration threshold to each transcript. The following pseudo-boolean functions exhibit the combinatorial properties of super or submodularity, which are important in respect to their optimization.

Let the function $t_Q^m : \mathbb{B}^m \to \mathbb{N}_0$ be the boolean function T_Q^m acting on $\mathbb{N}_0$. I define such *pseudo-boolean function* as

$$t_Q^m(\sigma) = \sum_{i=1}^{m-s(Q)+1} \prod_{j \in Q} \sigma_{i+j} \tag{4.8}$$

Here $t_Q^m(\sigma)$ *counts* how many times a Q-gram occurs in a transcript σ of length m. It is useful to define the complementary function $\bar{t}_Q^m$, counting how many times a Q-gram does not occur in a transcript σ, as

$$\bar{t}_Q^m(\sigma) = m - s(Q) + 1 - t_Q^m(\sigma) \tag{4.9}$$

Analogously, I define a pseudo-boolean function for a Q-gram family F

$$t_F^m(\sigma) = \sum_{Q_i \in F} t_{Q_i}^m(\sigma) \tag{4.10}$$

along with its complementary function

$$\bar{t}_F^m(\sigma) = \sum_{Q_i \in F} (m - s(Q_i) + 1) - t_F^m(\sigma) \tag{4.11}$$

All of the above functions expose important properties which lead to approximate solutions. *Nondecreasing monotonicity* of functions t_Q^m and t_F^m follow from nondecreasing monotonicity of their boolean counterparts T_Q^m and T_F^m. Consequently $\bar{t}_Q^m$ and $\bar{t}_F^m$ are *monotone nonincreasing*. From definition **??**, function t_Q^m is *supermodular*, thus it follows that $\bar{t}_Q^m$ is *submodular*. Since super and submodular functions are closed under nonnegative linear combination, functions t_F^m and $\bar{t}_F^m$ are respectively super and submodular.

4.4.2 Full-sensitivity

Problem definition

Instance A Q-gram, a (m, k) instance.
Question Does it exist a transcript $\sigma \in \mathbb{B}_k^m$ such that $T_Q^m(\sigma)$ is false?

Hardness results

Nicolas and Rivals [2005] show that NON DETECTION is *strongly* NP-complete by giving an indirect reduction of EXACT COVER BY 3-SETS to NON DETECTION. Strong NP-completeness implies that no *FPTAS* nor any *pseudo-polynomial* algorithm for NON DETECTION exist, under the assumption that $P \neq NP$. This fact motivates me to give ILP or approximate solutions to optimization and counting problems related to NON DETECTION.

4.4.3 Minimum lossy distance

Problem definition

Instance A Q-gram, an integer $m > 0$.
Solution The smallest integer k^* such that NON DETECTION for Q, (m, k^*) answers *no*.

Recalling pseudo-boolean function 4.8, I define this problem as the minimization of a linear function subject to submodular constraints

$$\begin{aligned} \min \quad & |\sigma|_0 \\ w.r.t. \quad & \\ & \sigma \in \mathbb{B}^m \\ & \bar{t}_Q^m(\sigma) = 0 \end{aligned} \tag{4.12}$$

ILP solution

I reduce the problem to MINIMUM SET COVER [Vazirani, 2001] and solve it with the following ILP

$$\begin{aligned} \min \quad & \sum \bar{\sigma} \\ w.r.t. \quad & \\ & \sigma \in \mathbb{B}^m \\ & A\sigma \geq b \end{aligned} \tag{4.13}$$

where the value A_{ij} of the coefficient matrix A is defined as

$$A_{ij} = \begin{cases} 1 & \text{if } i - j + 1 \in Q \\ 0 & \text{if } i - j + 1 \notin Q \end{cases} \tag{4.14}$$

and b is the unitary vector $\mathbb{1}^{m-s(Q)+1}$. Given the ILP solution $\bar{\sigma}^*$, the minimum distance for which the Q-gram is lossy is $k^* = |\sigma^*|_0$.

Observation 4.1. *Contiguous q-grams provide an interesting special case of this ILP. If A has the* Consecutive Ones Property*, it is* totally unimodular *[?]. The* polytope *defined by a totally unimodular coefficient matrix is* integral*. Hence the optimal solution of the relaxed LP is also the optimal solution of the original ILP.*

APX solution

I adapt the greedy algorithm by Chvatal to compute an approximate solution to MINIMUM SET COVER. Algorithm 4.1 computes via *gradient descent* a solution with an APX-ratio of $H_{w(Q)}$ [Chvatal, 1979].

Algorithm 4.1 MINLOSSYDISTANCE(Q, m)

Input Q : Q-gram sequence
m : integer denoting the pattern length
Output integer indicating the minimum lossy distance

1: $k \leftarrow 0$
2: $\sigma \leftarrow \mathbb{1}^m$
3: **while** $t_Q^m(\sigma) > 0$ **do**
4: i $\leftarrow \arg\max_{j=1}^{m} \frac{\partial t_Q^m(\sigma)}{\partial \sigma_j}$
5: $\sigma_i \leftarrow 0$
6: $k \leftarrow k + 1$
7: **return** k

4.4.4 Optimal threshold

Problem definition

Instance A Q-gram, a (m, k) instance.
Solution The largest integer t^* such that (Q, t^*) solves (m, k).

Recalling Q-gram pseudo-boolean functions 4.8, I define OPTIMAL THRESHOLD as the minimization of a supermodular function subject to linear constraints

$$\begin{aligned} \min \quad & t_Q^m(\sigma) \\ w.r.t. \quad & \\ & \sigma \in \mathbb{B}_k^m \end{aligned} \tag{4.15}$$

Recalling Q-gram pseudo-boolean functions 4.9, I consider the *complementary* OPTIMAL THRESHOLD problem as the maximization of a submodular function subject to linear constraints:

$$\begin{aligned} \max \quad & \bar{t}_Q^m(\sigma) \\ w.r.t. \quad & \\ & \sigma \in \bar{\mathbb{B}}_k^m \end{aligned} \tag{4.16}$$

Previous results

OPTIMAL THRESHOLD is fixed-parameter tractable (FPT) in the span of the Q-gram. Burkhardt and Kärkkäinen [2001] give a DP algorithm computing the optimal threshold in time $\mathcal{O}(m \cdot k \cdot 2^{s(Q)})$. Kucherov *et al.* [2005] give an extension for Q-gram families.

Exact ILP solution

I reduce this problem to MAXIMUM COVERAGE [Vazirani, 2001] and solve it with the following ILP

$$\begin{aligned} \max \quad & \sum c_j \\ w.r.t. \quad & \\ & \sigma \in \mathbb{B}_k^m \\ & c \in \mathbb{B}^{m-s(Q)+1} \\ & \sigma_i \geq c_j \end{aligned} \tag{4.17}$$

where variable c_j indicates the truthfulness of the j-th clause in T_Q^m. From the ILP solution c^*, I derive the exact solution to OPTIMAL THRESHOLD as $t^* = m - s(Q) + 1 - \sum c^*$.

APX solution

Algorithm 4.2 computes via *gradient descent* an approximate solution to OPTIMAL THRESHOLD. This algorithm has an APX-ratio of $1 + 1/e$ [Vazirani, 2001] for the complementary OPTIMAL THRESHOLD problem. The same *absolute error* applies to OPTIMAL THRESHOLD.

Algorithm 4.2 OPTIMALTHRESHOLD(Q, m, k)

Input Q : Q-gram sequence
m : integer denoting the pattern length
k : integer denoting the mismatches threshold
Output integer indicating the maximum lossless threshold

1: $\sigma \leftarrow \mathbb{1}^m$
2: **while** $k > 0$ **do**
3: i $\leftarrow \arg\max_{j=1}^{m} \frac{\partial \bar{t}_Q^m(\sigma)}{\partial \sigma_j}$
4: $\sigma_i \leftarrow 0$
5: $k \leftarrow k - 1$
6: **return** $m - s(Q) + 1 - \bar{t}_Q^m(\sigma)$

4.4.5 Specificity

Which is the expected specificity of a filtration scheme (Q, t)? Assuming the text to be generated according to the uniform Bernoulli model, the expected specificity of a filtra-

tion scheme is proportional to the number of detected transcripts. Among full-sensitive filtration schemes, one that maximizes the expected specificity is preferable.

Problem definition

Instance A filtration scheme (Q, t), a (m, k) instance.
Solution The number of transcripts $\sigma \in \mathbb{B}^m$ s.t. $t_Q^m(\sigma) \geq t$

FPRAS solution

I am interested in the number of transcripts:

$$\#\mathrm{P}^m(Q, t) = |\{\sigma \in \mathbb{B}^m : t_Q^m(\sigma) \geq t\}| \tag{4.18}$$

For threshold $t = 1$, this number is simply the number of true assignments of T_Q^m:

$$\#\mathrm{P}^m(Q, 1) = |\{\sigma \in \mathbb{B}^m : T_Q^m(\sigma)\}| \tag{4.19}$$

Karp *et al.* introduce a *fully polynomial-time randomized approximation scheme* (FPRAS) [Vazirani, 2001] to count the number of true assignments of a boolean function in DNF [Karp *et al.*, 1989]. This method is known as *importance sampling* [Vazirani, 2001]. Algorithm 4.3 works for an arbitrary threshold.

Algorithm 4.3 TRANSCRIPTSDETECTED(Q, t, m)

Input	Q : Q-gram sequence
	t : integer denoting the threshold
	m : integer denoting the pattern length
Output	integer indicating the number of transcripts detected

4.5 Evaluation

In this section, I show the results of an experimental evaluation of all filtration methods exposed so far. I consider seed filtration schemes with exact, 1 and 2-approximate seeds (Exact, 1-Apx and 2-Apx seeds), contiguous q-grams filtration schemes using the maximum lossless value of q for a threshold of 1 (q-Grams, $t \geq 1$) and 4 (q-Grams, $t \geq 4$). For indexed filters, I use the q-gram index with $q = 10$ (see 3.1.3). To perform edit distance verifications, I use a banded version of the Myers' algorithm [Myers, 1999]. As text, I take the *C. elegans* reference genome (WormBase WS195), i.e. a collection of 6 DNA strings of about 100 Mbp total length. As patterns, I extrapolated $200k$ DNA sequences of length 100 bp from an Illumina sequencing run (Sequence Read Archive ID SRR065390).

All experiments run on a desktop computer running Linux 3.10.11, equipped with one Intel® Core i7-4770K CPU @ 3.50 GHz, 32 GB RAM and a 2 TB HDD @ 7200 RPM.

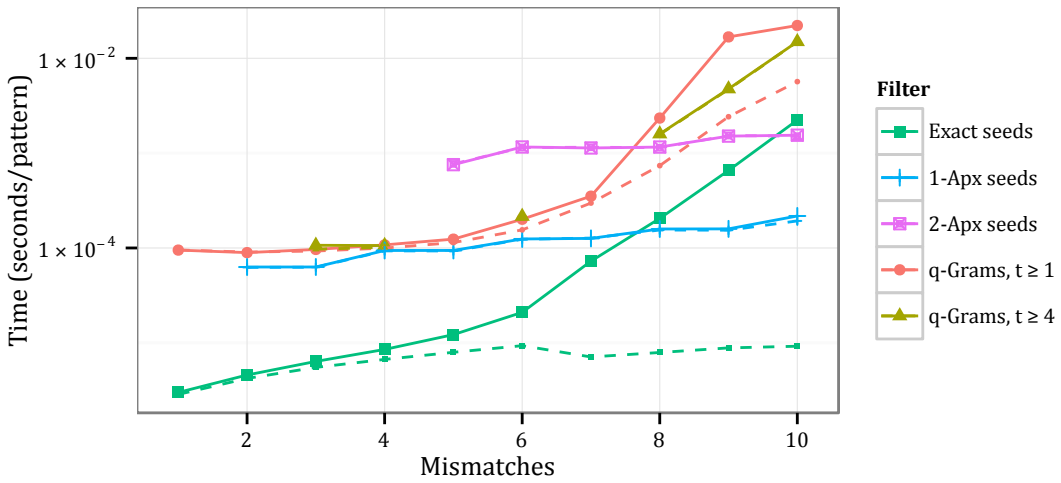
I repeated the experiment for k-mismatches and k-differences, varying k in the range $[1, 10]$. I measured the runtime of the filtration phase only, and then of the filtration plus verification phase. The plots show always *average* runtimes (or values) per pattern.

4.5.1 Runtime

Figure 4.6 shows the results for k-mismatches. Exact seeds are the best filtration method for $k \leq 7$, mainly due to their superior filtration speed, while 1-approximate seeds are better for $k \geq 8$. 2-Approximate seeds start to dominate exact seeds only for $k \geq 10$, i.e. they provide too strong filtration on this text. Both q-gram filtration schemes are always slower than 1-approximate seeds. It can be seen that enforcing $t \geq 4$ improves the total runtime for those instances where $t \geq 1$ renders the filter too weak.

Figure 4.7 shows the results for k-differences. The more involved edit distance verification slightly fills the gap between exact seeds and contiguous q-grams, yet exact seeds continue to be always the fastest alternative.

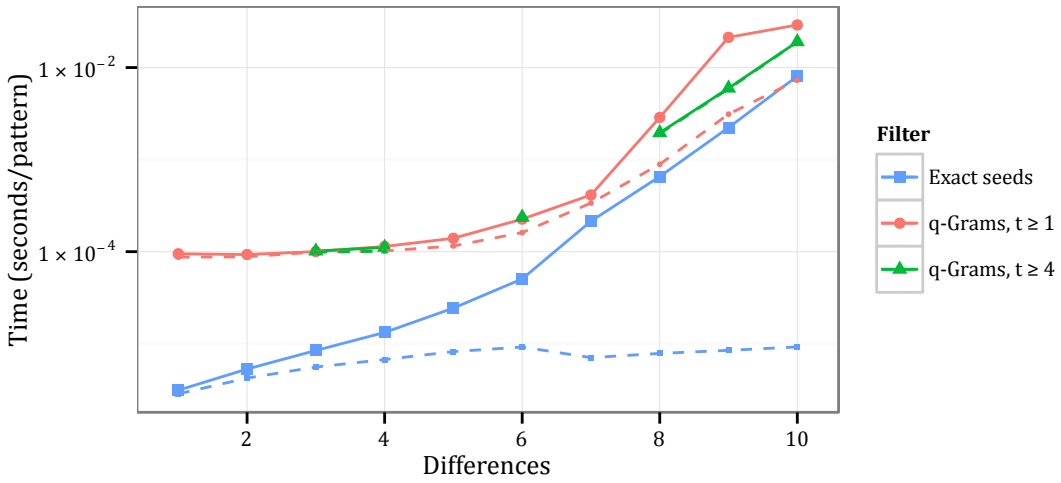
Figure 4.6: *Filters runtime on k-mismatches. Pattern length is fixed to 100. Dashed lines represent filtration time only.*



4.5.2 Verification versus filtration

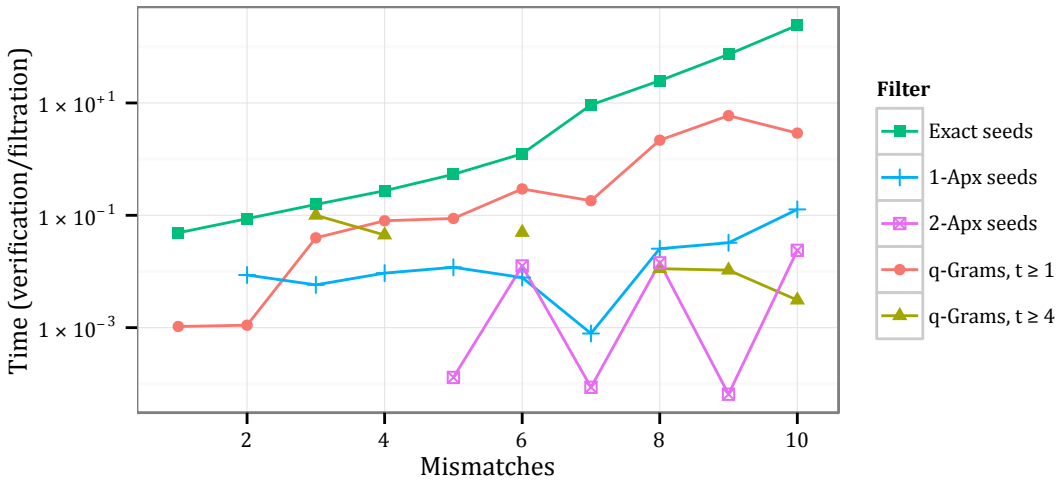
Figure 4.8 shows the ratios between the runtimes of the verification and filtration phases of each filtration scheme. For $k \leq 6$, all schemes spend more time on filtration rather than verification. The weakest scheme, filtration with exact seeds, shows the closest ratio to 1. As shown in the runtime plots (figures 4.6-4.7), quick filtration pays off for low error rates. Here, a quicker full-text index would be beneficial. For $k \geq 7$, contiguous q-grams with $t \geq 1$ show the closest ratio to 1, nonetheless the fastest alternative is provided by filtration with 1-approximate seeds, for which only 10 % of the runtime goes

Figure 4.7: *Filters runtime on k-differences. Pattern lengths are fixed to 100 bp. Dashed lines represent filtration time only.*



in verifications. Here, a judicious mix of exact and 1-approximate seeds could improve the total runtime. The ratios for k-difference show analogous patterns (data not shown).

Figure 4.8: *Ratio on k-mismatches. Pattern lengths are fixed to 100 bp.*



4.5.3 Positive predictive value

Instead of measuring filtration specificity, as introduced in section 2.4.3, I measure the *positive predictive value* (PPV). As shown in table 4.1, I define *true positives* (TP), *false pos-*

Table 4.1: *Measurement of filtering methods efficiency. $V_f(t)$ counts the number of verifications, $C(t)$ the number of approximate occurrences, and $||t||$ the total length of the text collection. Since all considered filtering methods are full-sensitive, the number of false negatives is always 0.*

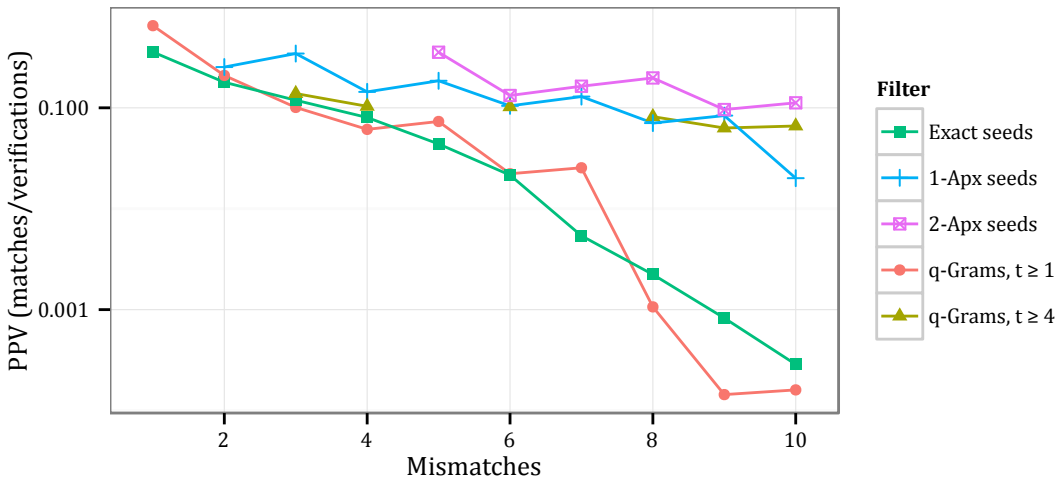
	Positive	Negative
True	$C(t)$	$\|\|t\|\| - C(t)$
False	$V_f(t) - C(t)$	0

itives (FP), *true negatives* (TN), or *false negatives* (FN) in terms of the number of verifications $V_f(t)$ triggered by the filtration phase and the number of approximate occurrences $C(t)$ of the pattern in the text. Therefore, I define the PPV as:

$$\frac{|TP_f(t)|}{|TP_f(t)| + |FP_f(t)|} = \frac{C(t)}{V_f(t)} \tag{4.20}$$

Figure 4.9 shows the results for k-mismatches. As expected, 2-approximate seeds have always the highest PPV, followed by 1-approximate seeds. The PPV of contiguous q-grams with $t \geq 1$ oscillates around that one of exact seeds. In particular, when t approaches 1, e.g. for $k = 9$, the PPV of contiguous q-grams stays below that one of exact seeds. Enforcing $t \geq 4$ boosts the PPV of contiguous q-grams to that one of 1-approximate seeds. The PPVs for k-difference show analogous patterns (data not shown).

Figure 4.9: *Filters specificity on k-mismatches. Pattern lengths are fixed to 100 bp.*



Part II

READ MAPPING

CHAPTER

5 Background

In this chapter, I provide the reader with background knowledge in the fast-moving field of high-throughput sequencing (HTS) read mapping. In section 5.1, I briefly introduce the two most prominent HTS technologies and the kind of sequencing data they produce. Afterwards, in section 5.2, I present two popular paradigms for reference-guided assembly: best-mapping and all-mapping. In section 5.3, I review two studies on the hardness of HTS data analysis. Finally, in section 5.4, I give an overview of the most popular read mapping tools.

5.1 High-throughput sequencing data

As anticipated in section 1.1, actual HTS technologies produce DNA reads which are shorter than Sanger sequencing and more likely to contain systematic sequencing artifacts. HTS data does not consist of read sequences only, but includes also base quality scores annotating the quality of the sequencing process.

5.1.1 Phred base quality scores

Base quality scores have been introduced by the base calling tool *Phred* [Ewing *et al.*, 1998; Ewing and Green, 1998] to assess sequencing quality of single bases in capillary reads. Instead of discarding low-quality regions present in capillary reads, Phred output each base and annotates it with the probability that it has been wrongly called. The tool encodes the probability ϵ_i of miscalling the i-th base in a read under the form of a base quality Q_i in logarithmic space:

$$Q_i = -10 \log_{10} \epsilon_i. \tag{5.1}$$

This method has been unanimously accepted. All sequencing technologies complement DNA reads with base quality scores, encoded in Phred-scale or similar.

5.1.2 Read sequences

I consider only sequencing data produced by the most prominent HTS instruments. *Illumina* is the actual market leader for HTS instruments, followed by *Life Technologies*'s *Ion Torrent*. Some instruments, e.g. the *GS* by *454 Life Sciences* or the *SOLiD* by *Applied*

Biosystems, became popular around 2006–2010, but are now discontinued. Other *third-generation* instruments, like the *single-molecule real-time* sequencing (SMRT) *RS II* by *Pacific Biosciences*, have great potential but still low impact on the HTS market.

Illumina

All Illumina instruments use the *sequencing by synthesis* (SBS) technology [**?**] developed by Solexa. In the library preparation phase, the DNA sample is sheared into smaller fragments. During the sequencing phase, single-stranded DNA fragments are attached on a slide called *flow cell* and amplified *in situ* using a variant of *polymerase chain reaction* (PCR) called *bridge amplification*. Clusters of amplified fragments are then used as templates for multiple cycles of synthetic sequencing with four differentially labeled fluorescent reversible terminator *deoxyribonucleotide triphosphates* (dNTPs). In each sequencing cycle, dNTPs are incorporated into the fragments in each cluster, their corresponding fluorescent reversible terminators are imaged by a high-resolution camera, and then dNTPs are cleaved to allow incorporation of the next base.

After the sequencing phase, a base calling software converts all images corresponding to one cluster into one read. The software measures the intensities of the four colors imaged during the i-th cycle, calls the i-th read base, and assigns a base quality score. Eventual replication errors, made by DNA polymerase during bridge amplification, result in mixed signal intensities within a cluster, and hence to base calls of lower confidence.

Illumina's SBS technology further allows the sequencing of both ends of each DNA fragment. Two kind of libraries are available: *paired-end* libraries, consisting of reads from short-insert 300–600 bp fragments, and *mate pair* libraries, consisting of reads separated by several kilobases. The former libraries are adopted for high-resolution genome resequencing, while the latter ones provide accurate *de novo* sequence assembly or detection of large-scale structural variation.

Ion Torrent

Life Technologies's *Ion Torrent* instruments use a semiconductor-based technology [**?**]. During library preparation, the sheared single-stranded template DNA is embedded into *microwells* on a semiconductor chip with DNA polymerase. Microwells are sequentially flooded with unmodified A, C, G or T dNTP. The DNA polymerase incorporates the introduced dNTP into the growing strand only if this is complementary to the leading template nucleotide. Such polymerization reaction releases an hydrogen ion, which in turn changes the pH of the solution. A *ion-sensitive field-effect transistor* (ISFET) in the sequencing instrument measures this pH change. Any *homopolymer* in the template causes multiple dNTP to be incorporated in a single cycle and results in a higher pH change, which is not precisely measured by the instrument and thus causes systematic sequencing errors.

5.2 Data analysis paradigms

The goal of secondary analysis is to reconstruct the original sequence of the donor genome. The *reference-guided* assembly plan assumes prior knowledge of a reference genome similar to the donor. Reads are thus mapped (i.e. aligned) to the reference genome w.r.t. a given scoring scheme and threshold. The scoring scheme accounts for eventual genomic variation, as well as for any sequencing artefacts. Under these assumptions, an alignment of optimal score for a read implies its *original location* on the reference genome. Conversely, no alignment within the score threshold implies too many sequencing errors, too much genetic variation, or sample contamination. A problem arises in presence of co-optimal or close sub-optimal alignments: the read cannot be mapped confidently to one single location.

The problem of confidently mapping high-throughput sequencing reads comes from the non-random nature of genomic sequences. Genomes evolved through multiple types of duplication events, including whole-genome duplications [Wolfe and Shields, 1997; Dehal and Boore, 2005] or large-scale segmental duplications in chromosomes [Bailey *et al.*, 2001; Samonte and Eichler, 2002], transposition of repetitive elements as short tandem repeats (microsatellites) [Wang *et al.*, 1994; Wooster *et al.*, 1994] and interspersed nuclear elements (LINE, SINE) [Smit, 1996], proliferation of repetitive structural elements such as telomeres and centromeres [Meyne *et al.*, 1990]. As a result of these events, for instance, about 50 % of the human genome is composed of repeats.

Repeats present in general technical challenges for all *de novo* assembly and sequence alignment programs [Treangen and Salzberg, 2011]. Due to repetitive elements, a non-ignorable fraction of high-throughput sequencing reads cannot be mapped confidently. In general, the shorter the reads, the higher the challenges due to repeats. I quantify this phenomenon more precisely in section 5.3. Here I focus on analysis strategies to deal with *multi-reads*, i.e. reads that cannot be mapped confidently as they align equally well to multiple locations.

It is not evident how to treat *multi-reads*. According to Treangen and Salzberg, common strategies to deal with multi-reads are (i) to discard them all, (ii) to randomly pick one best mapping location, (iii) to consider all or up to k best mapping locations within a given distance threshold. A *de facto* standard strategy, combining strategies (i) and (ii), emerged over the last years. The read mapper randomly picks one best mapping location and complements it with its *mapping quality*, i.e. the probability of the mapping location being correct (see section 5.2.1). Subsequently, downstream analysis tools apply a mapping quality score cutoff to discard reads not mapping confidently to any location. The other popular strategy adopted by analysis tools is to consider all mapping locations within an edit distance threshold. In this case, it is not clear whether downstream analysis tools consider equally all mapping locations regardless of their distance.

In the light of these facts, I define two broad paradigms for the secondary and tertiary analysis of HTS data: *best-mapping* and *all-mapping*. The best-mapping paradigm considers a single mapping location per read along with its confidence, while the all-mapping paradigm considers a comprehensive set of mapping locations per read. It goes without saying that read mapper and downstream analysis tools must agree on a com-

mon paradigm. Thus these paradigms are valid not only for read mappers but also for any downstream analysis tool, e.g. variant callers. Read mapping and variant calling are indeed tightly coupled steps within reference-based HTS pipelines.

5.2.1 Best-mapping

As said, best-mapping methods rely on one single mapping location per read. In order to maximize recall, best-mappers often adopt complex scoring schemes taking into account gaps and base quality values, and at the same time implement sophisticated heuristics to speed up the search. Best-mappers annotate any mapping location with its mapping quality. Subsequently, in order to maximize precision, variant calling tools decide whether to consider or discard reads not mapping confidently to any location. The GATK [DePristo *et al.*, 2011] and Samtools [Li *et al.*, 2009a] are popular best-mapping tools to call small variants. In section 5.3, I discuss how this paradigm systematically discards reads belonging some critical genomic regions, thus is limited to the analysis of *high mappability* regions.

Mapping quality score

Mapping quality has been introduced in the tool MAQ [Li *et al.*, 2008]. The study considers short 30–40 bp reads, produced by early Illumina and ABI/SOLiD sequencing technologies, whose sequencing error rates were quite high. Given the short lengths and high error rates, a significant fraction of such reads can be aligned to multiple mapping locations, even considering only co-optimal Hamming distance locations.

The argument of Li *et al.* is that the Hamming distance is not an adequate scoring scheme to guess the correct mapping location of many reads. The authors claim that *"it is possible to act conservatively by discarding reads that map ambiguously at some level, but this leaves no information in the repetitive regions and it also discards data, reducing coverage in an uneven fashion, which may complicate the calculation of coverage."* Though, Li *et al.* do not show in their study what is the effect of relying on mapping quality rather than on mapping uniqueness. Since base callers output base call probabilities in Phred-scale along with the reads, Li *et al.* propose a novel probabilistic scoring scheme called *mapping quality*, encoding the probability that a read aligns correctly at a mapping location in the reference genome.

Fix the alphabet $\Sigma = \{ \text{A, C, G, T} \}$. Consider a known donor genome g over Σ and a read r sequenced at location l from the template $g_{l \dots l+|r|-1}$. The base calling error ϵ_i from equation 5.1 represents the probability ϵ_i of miscalling a base r_i instead of calling its corresponding base g_{l+i-1} in the donor genome. The probability $p(r_i|g_{l+i-1})$ of observing the base r_i given the donor genome base g_{l+i-1}, is:

$$p(r_i|g_{l+i-1}) = \begin{cases} 1-\epsilon_i & \text{if } g_{l+i-1} = r_i \\ \frac{\epsilon_i}{|\Sigma|-1} & \text{if } g_{l+i-1} \in \Sigma \setminus \{r_i\} \end{cases} \tag{5.2}$$

and assuming i.i.d. base calling errors, it follows that the probability $p(r|g, l)$ of observing

the read r, given the donor genome template $g_{l\ldots l+|r|-1}$, is:

$$p(r|g,l) = \prod_{i=1}^{|r|} p(r_i|g_{l+i-1}). \tag{5.3}$$

By applying Bayes' theorem, Li *et al.* derive the posterior probability $p(l|g,r)$, that location l in the reference genome g is the correct mapping location of read r. Assuming uniform coverage, each location $l \in [1, |g| - |r| + 1]$ has equal probability of being the origin of a read in the donor genome, thus the prior probability $p(l)$ is simply:

$$p(l) = \frac{1}{|g| - |r| + 1} \tag{5.4}$$

Therefore, recalling $p(r|g,l)$ from equation 5.3, the posterior probability $p(l|g,r)$ equals the probability of the read r originating at location l, normalized over all possible locations in the reference genome:

$$p(l|g,r) = \frac{p(r|g,l)}{\sum_{i=1}^{|g|-|r|+1} p(r|g,i)} \tag{5.5}$$

which in Phred-scale becomes:

$$Q(l|g,r) = -10\log_{10}[1 - p(l|g,r)]. \tag{5.6}$$

Computing the exact mapping quality as in equation 5.6 requires aligning each read to all positions in the reference genome. On the one hand, this computation is practically infeasible. On the other hand, sub-optimal locations not close to the optimum one contribute very little to the sum in equation 5.5. Therefore, read mapping programs approximate equation 5.5 using only the mapping locations they repute relevant.

Mapping quality has been initially used in MAQ [Li *et al.*, 2008] and BWA [Li and Durbin, 2009] to maximize variant calling confidence by discarding reads whose best mapping location is below a given mapping quality threshold. Since then, this method has been widely accepted. Nowadays, mapping quality is computed by best-mappers and used by almost all variant calling tools e.g. the Genome Analysis ToolKit (GATK) [DePristo *et al.*, 2011].

Despite their wide adoption, some important objections can be raised against mapping quality scores. First, this score is derived under the unlikely assumption of the reference genome being equal to the donor genome. In other words, mapping quality considers only errors due to base miscalls and disregards genetic variation; thus the risk is to prefer mapping locations supported by known low base qualities rather than by true but unknown SNVs. Second, mapping quality is nonetheless strongly correlated to mapping uniqueness, as discussed in section 5.3; it is easy to see that the probability of any location in equation 5.5 dilutes in presence of a large number of co-optimal or close sub-optimal mapping locations. Third, mapping quality tends to become less relevant as base calls improve, due to advances of sequencing technologies, and thus degenerates into a shallow measure of uniqueness.

5.2.2 All-mapping

All-mapping analysis methods consider a comprehensive set of locations per read. Almost all read mappers in this category adopt edit distance and report all mapping locations within an error threshold, absolute or relative w.r.t. to the length of the reads. Variant calling algorithms based on all-mapping have the potential to detect a wider spectrum of genomic variation events than their best-mapping counterparts. For instance, variant callers based on the all-mapping paradigm detect CNVs [Alkan *et al.*, 2009], and SNVs in homologous regions [Simola and Kim, 2011]. A practical challenge of all-mapping is represented by reporting and handling huge sets of mapping locations.

5.3 Limits of high-throughput sequencing

A fraction of high-throughput sequencing reads cannot be mapped confidently due to repetitive elements. Which regions of a model organism's genome cannot be resequenced confidently by a high-throughput sequencing technology? And how accurate is downstream analysis on these low confidence regions? Two recent studies [Derrien *et al.*, 2012; Lee and Schatz, 2012] answer these questions. Below, I report their key ideas and most relevant findings.

5.3.1 Genome mappability

Derrien *et al.* define *genome mappability* as a function of a genome for a fixed q-gram length, distance measure i.e. the Hamming or edit distance, and distance threshold k. Given a genomic sequence g, they define the (q,k)-frequency $F_k^q(l)$ of the q-gram $g_{l\ldots l+q-1}$ at location l in g as the number of occurrences of the q-gram in g and its reverse complement $\bar{g}$. The (q,k)-mappability $M_k^q(l)$ is the inverse (q,k)-frequency, i.e. $M_k^q(l) = F_k^q(l)^{-1}$ with $M_k^q : \mathbb{N} \rightarrow]0,1]$. Note that $M_k^q(l)$ can be seen as the prior probability that any read of length q originating at location l will be mapped correctly. The values of (q,k)-frequency and mappability obviously vary with the distance threshold k. Nonetheless, under any distance measure, it hold that the q-gram at location l is unique up to distance k iff $M_k^q(l) = 1$ and repeated otherwise.

Unique mappability determines which fraction of a genome can be analyzed according to strategy (i) of [Treangen and Salzberg, 2011] (i.e. discarding non-unique reads, see section 5.2). Derrien *et al.* quantify the *unique mappability* of whole human, mouse, fly, and worm genomes. Mimicking typical Illumina read mapping setups, they consider q-grams of length 36, 50 and 75 bp, and Hamming distance 2. They find out that about 30 % of the whole human genome is not unique w.r.t. $(36,2)$-mappability. At $(75,2)$-mappability, 17 % of the human genome is not yet unique. This last result is slightly optimistic, as typical mapping setups call for up to 3–4 edit distance errors in order to map a significant fraction of the reads. Table 5.1 shows some results obtained from [Derrien *et al.*, 2012].

To estimate single-base resequencing accuracy, Derrien *et al.* consider the mappability of all possible q-grams spanning any single genomic location. They define *pileup*

Table 5.1: *Mappability of model genomes. Data obtained from [Derrien* et al.*, 2012].*

	H.sapiens (hg19)	M.musculus (mm9)	D.mel (dm3)
Repeats content [%]	45.25	42.33	26.50
Uniqueome (36 bp, 2 msm) [%]	69.99	72.07	68.09
Uniqueome (50 bp, 2 msm) [%]	76.59	77.06	69.44
Uniqueome (75 bp, 2 msm) [%]	83.09	81.65	71.00

mappability P_k^q at position i as the average mappability of all q-grams spanning position i:

$$P_k^q(i) = 1/q \sum_{j=i}^{i+1} M_k^q(j). \tag{5.7}$$

Derrien *et al.* find out in their own resequencing studies that *"low pileup-mappability regions are more prone to show a high value of heterozygosity than those with high mappability"* [Derrien *et al.*, 2012]. Ideally, variant calling tools call a locus as heterozygous whenever the consensus alignment column at that locus contains two distinct bases. This situation tends to arise more frequently whenever the consensus alignment contains reads originating from similar yet distinct regions.

5.3.2 Genome mappability score

Genome mappability score (GMS) [Lee and Schatz, 2012], analogously to pileup mappability, estimates single-locus resequencing accuracy for a specific sequencing technology. Instead of considering the inverse q-gram frequency, Lee and Schatz use mapping quality (see section 5.2.1) to estimate the probability that a read originating at a given position can be mapped correctly. Subsequently, they derive the average mapping probability of any read spanning a location l of a reference genome g as:

$$p(l|g) = \sum_{r \in \mathcal{R}(l)} \frac{p(l|g,r)}{|\mathcal{R}(l)|} \tag{5.8}$$

which in Phred-scale becomes:

$$Q(l|g) = \sum_{r \in \mathcal{R}(l)} \frac{1 - 10^{-\frac{Q(l|g,r)}{10}}}{|\mathcal{R}(l)|}. \tag{5.9}$$

Thus, fixed a genomic sequence g, they define the genome mappability score $\mathrm{GMS}(l)$ in percentual value:

$$\mathrm{GMS}(l) = 100\, Q(l|g) \tag{5.10}$$

Lee and Schatz proceed as follow to compute GMS. They first simulate reads from all genomic locations, having length and error profiles similar to those issue by actual sequencing technologies. Subsequently, they compute mapping quality scores by mapping all simulated reads with the best-mapper BWA [Li and Durbin, 2009]. Then, as just explained, they compute GMS at any location by averaging the quality scores. Finally, they define *low GMS* regions as those locations for which GMS(l) $\leq$ 10 and *high GMS* otherwise. Table 5.2 shows the performance of various sequencing technologies on the whole human genome (data obtained from [Lee and Schatz, 2012]).

Table 5.2: *Human genome mappability score of various sequencing technologies. Data obtained from [Lee and Schatz, 2012].*

Sequencing technology	Read length [bp]	Error rate [%] (msm, ins, del)	Low GMS [%]	High GMS [%]
Illumina-like	100	(0.10, 0.00, 0.00)	10.51	89.49
Ion Torrent-like	200	(0.04, 0.01, 0.95)	9.35	90.65
Roche/454-like	800	(0.18, 0.54, 0.36)	8.91	91.09
PacBio-like	2000	(1.40, 11.47, 3.43)	100.0	0.00
PacBio EC-like	2000	(0.33, 0.33, 0.33)	8.61	91.39

Lee and Schatz measure variant calling accuracy by GMS for the popular combination of best-mapping tools BWA and SAMtools [Li *et al.*, 2009a]. They simulate an Illumina-like resequencing study and feed it to such analysis pipeline. They find out that, at 30 × sequencing coverage, accuracy approaches 100 % in high GMS regions, while it levels off to 25 % in low GMS regions. Their analysis *"shows that most SNP detection errors are false negatives, and most of the missing variations are in regions with low GMS scores"* [Lee and Schatz, 2012]. These are the limits of the *de facto* standard best-mapping paradigm for the analysis of high-throughput sequencing data.

5.4 Popular read mappers

Following the boom of NGS technologies, recent bioinformatics research has produced dozens of tools to perform read mapping. Two surveys [Li and Homer, 2010; Fonseca *et al.*, 2012] try to help bioinformaticians to extricate themselves from the jungle of read mapping tools. The survey by Li and Homer first classifies read mapping algorithms by data structure: those based on hash tables and those based on suffix/prefix trees. However, the adopted data structure is often an implementation detail, indeed most algorithms covered in the survey fit into both classes. The survey primarily considers the application of SNP calling; in the considered setup, tools enumerating a comprehensive set of locations always lag behind those designed to report only one location per read. The survey by Fonseca *et al.* instead catalogs read mappers by the features ex-

posed to the user. It considers supported input–output formats, rate of errors and variation, number and type (i.e. local or semi-global read alignments) of mapping locations reported. After this exhaustive catalog, the survey concludes that the choice of a read mapper *"involves application-specific requirements such as how well it works in conjunction with downstream analysis tools (i.e. variant callers)"*.

Read mapping and variant calling are indeed tightly coupled steps within reference-based HTS analysis pipelines. As explained above, secondary and tertiary analysis methods are based on one of the two following paradigms: best-mapping and all-mapping. In the light of the above consideration, the most important feature of a read mapper is the number of mapping locations reported, followed by their type, while the other features are mostly of technical relevance. Most read mappers are specifically designed to fit one paradigm, while others are versatile enough to work well in both cases.

The rest of this section presents most popular read mapping tools. Among them, BWA [Li and Durbin, 2009], Bowtie [Langmead *et al.*, 2009] and Bowtie 2[Langmead and Salzberg, 2012], and Soap [Li *et al.*, 2009b] are prominent tools designed for best-mapping, while SHRiMP 2 [David *et al.*, 2011], mr(s)Fast [Alkan *et al.*, 2009; Hach *et al.*, 2010], RazerS [Weese *et al.*, 2009] and RazerS 3 [Weese *et al.*, 2012], and Hobbes 2 [Kim *et al.*, 2014] are designed for all-mapping. Grosso modo, most prominent best-mappers recursively enumerate substrings on a suffix/prefix tree of the reference genome via backtracking algorithms. Backtracking alone is impractical as its time complexity grows exponentially with the number of errors considered, hence best-mappers apply heuristics to reduce and prioritize enumeration. Conversely, all-mappers are based on filtering algorithms for approximate string matching. They quickly determine, often with the help of an index, locations of the reference genome candidate to contain approximate occurrences, then verify them with conventional methods. Their efficiency is bound to filtration specificity and thus deteriorates with increasing error rates and genome lengths. Finally, the most recent tools GEM [Marco-Sola *et al.*, 2012], Masai [Siragusa *et al.*, 2013], and Yara [**?**], fit both best and all-mapping paradigms. They speed up best-mapping by stratifying mapping locations by edit distance and prioritizing filtration accordingly. In addition to that, they also speed up all-mapping by means of more specific filters based on approximate seeds. Table 5.3 gives an overview of all these tools.

5.4.1 Bowtie and Bowtie 2

Bowtie [Langmead *et al.*, 2009] is a mapper designed to have a small memory footprint and quickly report a few good mapping locations for early generation short Illumina and ABI/SOLiD reads. The tool achieves the former goal by indexing the reference genome with an FM-index and the latter one by performing a greedy backtracking on it. The greedy top-down traversal visits first the subtree yielding the least number of mismatches and stops after having found a candidate (not guaranteed to be optimal when $k > 1$). In addition, Bowtie speeds up backtracking by applying *case pruning* [Mäkinen *et al.*, 2010], a simple application of the pigeonhole principle. However, this technique is mostly suited for $k = 1$ and requires the index of the forward and reverse reference genome. Bowtie can be configured to search by strata, but the search time increases significantly while

the traversal still misses a large fraction of the search space due to seeding heuristics.

Bowtie 2 [Langmead and Salzberg, 2012] has been designed to quickly report a couple of mapping locations for longer Illumina, Ion Torrent and Roche/454 reads, usually having lengths in the range from 100 bp to 400 bp. This tool uses an heuristic seed-and-extend approach, collecting seeds of fixed length, partially overlapping, and searching them exactly in the reference genome using an FM-index. Bowtie 2 randomly chooses candidate locations, to avoid uncompressing large suffix array intervals and executing many DP instances. The tool verifies candidate locations using a striped vectorial dynamic programming algorithm by Farrar [2007], implemented using SIMD instructions. Bowtie 2 can be configured to report semi-global or local alignments, scored using a tunable affine scoring scheme.

5.4.2 BWA

BWA [Li and Durbin, 2009] is designed to map Illumina reads and report a few best semi-global alignments. The program backtracks the FM-index of the reference genome with a *greedy breadth-first search*. The tool ranks nodes to be visited by edit distance score: the best node is popped from a priority queue and visited, its children are then inserted again in the queue. The traversal considers indels using a more involved 9-fold recursion. Li and Durbin speed up backtracking by adopting a more stringent pruning strategy [Mäkinen *et al.*, 2010] that nonetheless takes some preprocessing time and requires the index of the reverse reference genome. BWA performs paired-end alignments by trying to anchor both paired-end reads and verifying the corresponding mate, within an estimated insert size, using the classic DP-based Smith-Waterman algorithm. Consequently, the program in paired-end mode aligns reads at a slower rate than in single-end mode. The program is not fully multi-threaded, therefore it scales poorly on modern multi-core machines.

5.4.3 Soap

Soap 2 [Li *et al.*, 2009b] has been designed to produce a very quick but shallow mapping of short Illumina reads, up to 2 mismatches and without indels. The tool performs backtracking using the so-called bi-directional (or 2-way) BWT [Belazzougui *et al.*, 2013]. Soap 2 supports paired-end mapping but at a slower alignment rate, it lacks native output in the *de-facto* standard SAM format, and it is not open source.

5.4.4 SHRiMP 2

The *SHort Read Mapping Program* (SHRiMP 2) [David *et al.*, 2011] is designed to map short Illumina and ABI/SOLiD reads. The tool achieves high accuracy at the expense of speed. SHRiMP 2 indices the reference genome using multiple gapped q-grams. At query time, it projects each read to identify candidate mapping locations, which are verified with the Smith□Waterman algorithm [Smith and Waterman, 1981]. The SHRiMP 2 project has been recently discontinued.

5.4.5 RazerS and RazerS 3

RazerS [Weese *et al.*, 2009] has been designed to report all mapping locations within a fixed Hamming or edit distance error rate. It is based on a full-sensitive q-gram counting filtration method (see section 4.3) combined with the edit distance verification algorithm by Myers [Myers, 1999]. On demand, the tool throttles filtration to be more specific at the expense of a controlled loss rate. Stronger filtration reduces the number of candidate locations and improves the overall speed of the program. All in all, the SWIFT filter is very slow while not highly specific.

RazerS 3 [Weese *et al.*, 2012] is a faster version featuring shared-memory parallelism, a banded version of Myers' algorithm, and a quicker filtration method based on exact seeds (see section 4.1). Such filtration method however turns out to be very weak on mammal genomes. Because of this fact, RazerS 3 is one-two orders of magnitude slower than Bowtie 2 and BWA on such datasets.

All RazerS versions index the reads and scan the reference genome. One positive aspect of this strategy is that no preprocessing of the reference genome is required. However, other mapping strategies beyond all-mapping, e.g. mapping by strata, cannot be efficiently implemented. Moreover, these programs exhibit an high memory footprint as they remember the mapping locations of all input reads until the whole reference genome has been scanned.

5.4.6 mrFast and mrsFast

The tools mrsFast [Hach *et al.*, 2010] and mrFast [Ahmadi *et al.*, 2012] are designed to map Illumina reads. They report all mapping locations within a fixed absolute number errors, respectively under the edit and Hamming distance. Similarly to RazerS 3, these two programs implement full-sensitive filtration using exact seeds (section 4.1). Their peculiarity is a cache-oblivious strategy to mitigate the high cost of verifying clusters of candidate locations. In addition, mrsFast computes the edit distance between one read and one mapping location in the reference genome with an antidiagonal-wise vectorial dynamic programming algorithm, implemented using SIMD instructions. These tools perform only all-mapping, produce files of impractical size and lack multi-threading support.

5.4.7 Hobbes 2

Hobbes 2 [Kim *et al.*, 2014] is designed to identify all read mapping locations within a fixed Hamming or edit distance threshold. In order to improve filtering efficiency, the tool employs a novel technique of so-called *prefix q-grams* that enriches the reference genome q-gram index. However, this technique does not guarantee full-sensitivity.

5.4.8 GEM

The GEM mapper [Marco-Sola *et al.*, 2012] is a flexible read aligner for Illumina and Ion Torrent reads. It is and can be configured either as an all-mapper, as a best/unique-

mapper, or to search by strata. GEM implements full-sensitive filtration with approximate seeds (see section 4.2). The program indexes the reference genome with an FM-index, tries to find an optimal filtration scheme per read, and verifies candidate locations using Myers' algorithm Myers [1999]. GEM maps paired-reads in two ways: either it maps both ends independently and then combines them, or maps one end and then verifies the other end using an online method. Unfortunately, the tool is not open source and provides obscure parameterization.

Table 5.3: *Overview of popular read mappers.*

	sequencer		paradigm			alignment			index		
	Illumina	Ion	best	strata	all	type	optimal	method	type	reference	reads
Bowtie	short	✗	✓	✓	✗	mismatches	✗	backtracking	FM-index	✓	✗
Bowtie 2	✓	✓	✓	✗	✗	local	✗	exact seeds	FM-index	✓	✗
BWA	✓	✗	✓	✗	✗	indels	✗	backtracking	FM-index	✓	✗
Soap 2	short	✗	✓	✓	✗	mismatches	✗	backtracking	FM-index	✓	✗
RazerS	✓	✗	✗	✓	✓	indels	✓	q-grams	q-gram index	✗	✓
RazerS 3	✓	✗	✗	✓	✓	indels	✓	exact seeds	q-gram index	✗	✓
SHRiMP 2	✓	✓	✓	✗	✓	local	✗	q-grams	q-gram index	✓	✗
mrsFast	short	✗	✗	✗	✓	mismatches	✓	exact seeds	q-gram index	✓	✓
mrFast	✓	✗	✗	✗	✓	indels	✓	exact seeds	q-gram index	✓	✓
Hobbes 2	✓	✗	✗	✗	✓	indels	✗	prefix q-grams	q-gram index	✓	✗
GEM	✓	✓	✓	✓	✓	indels	✓	apx seeds	FM-index	✓	✗
Masai	✓	✗	✓	✗	✓	indels	✓	apx seeds	generic	✓	✓
Yara	✓	✓	✓	✓	✓	indels	✓	apx seeds	FM-index	✓	✗

CHAPTER

6 Masai

This chapter presents the first efficient of an efficient all-mapper for Illumina reads. When I started this project, in October 2011, the fastest all-mappers (mrFast and RazerS 3) were two order of magnitude slower than popular best-mappers (Bowtie and BWA). On the one hand, such all-mappers employ filtration based on exact seeds, which is fine for short reference genomes but becomes too weak for mammal genomes; clearly, a stronger filtration method would had been beneficial. On the other hand, such best-mappers are based on heuristic backtracking, which is inadequate to map longer Illumina reads.

After a thorough literature review, I came out with a novel read mapping method combining seed-based filtration with backtracking, published [Siragusa *et al.*, 2013] in the peer-reviewed journal *Nucleic Acids Research*. My method is packaged in a C++ tool nicknamed *Masai*, which stands for *m*ultiple backtracking of *a*pproximate seeds on a *s*uffix *a*rray *i*ndex. Masai is part of the SeqAn library, it is distributed under the BSD license and can be downloaded from `http://www.seqan.de/projects/masai`.

In the engineering section, I expose how Masai's filtration method works, which data structures it uses for indexing, and how it performs seed extension. In particular, the filtration method is based on approximate seeds: by employing approximate seeds instead of exact seeds, the tool obtains stronger, non-heuristic and quasi full-sensitive filtration for mammal reference genomes. Masai find approximate seeds by backtracking the index of the reference genome. Moreover, the tool speeds up the backtracking phase by searching all seeds simultaneously, with the help of an additional index and the multiple backtracking algorithm. Lastly, Masai implements a method to perform best-mapping in a more efficient way.

In the evaluation section, I extensively compare Masai with popular read mappers, both on simulated and real datasets. Compared to the all-mappers mrFast and RazerS 3, Masai is an order of magnitude faster and has comparable sensitivity. In addition, Masai in best-mapping is 2–4 times faster and more accurate than Bowtie 2 [Langmead and Salzberg, 2012] and BWA [Li and Durbin, 2009]. Finally, I discuss the limitations of Masai that led me to engineer Yara, yet another read aligner.

6.1 Engineering

I first give an outline of the read mapping method implemented in Masai. Then, I explain each step in details, motivating relevant engineering choices that led me to the final im-

plementation.

Masai requires an index capable of simulating a top-down traversal of the suffix trie of the reference genome. The tool gives to users the possibility to choose among various indices (see section 6.1.2). Similarly to all read mappers relying on an index of the reference genome, the tool indexes the reference genome only once, stores it on disk and reuses it for all subsequent read mapping jobs.

At mapping time, Masai requires two parameters to be provided: a maximum number of errors per read and a minimum seed length. Default parameters work well for actual Illumina reads, otherwise the user has to adequately parametrize the tool for optimal performance. Nonetheless, independently of the chosen parameterization, filtration is guaranteed to be quasi full-sensitive (see section 6.1.1).

Masai partitions all reads (and their reverse complements) into non-overlapping seeds and indexes them in a conceptual *trie*. Using the *multiple backtracking* algorithm of section 3.3.6, the tool backtracks simultaneously all indexed seeds in the suffix trie of the reference genome. The program verifies all candidate locations, reported by the multiple backtracking algorithm, performing seed extension with a banded version of *Myers bit-vector algorithm* [Myers, 1999] (details in section 6.1.3).

6.1.1 Filtration

My original intent was to improve the speed of the all-mapper RazerS [Weese *et al.*, 2009] while preserving full-sensitivity under the edit distance. RazerS was based on a q-gram filter; I was aware that gapped q-grams could have brought a huge speedup, but I could not see any straightforward generalization of gapped q-grams to the edit distance. At the same time, I experienced that weaker but quicker filtration using exact seeds is more advantageous than filtration using q-grams. Indeed, a typical Illumina read mapping setup requires only moderate error rates, in the range of 4–6 %. For instance, RazerS 3 [Weese *et al.*, 2012] went back to filtration with exact seeds (similarly to mrFast). Nonetheless, I wanted to improve filtration specificity of exact seeds, as the runtime of RazerS 3 on mammal genomes became dominated by verifications. I knew that to improve filtration specificity I had to increase the seed length.

While reviewing past literature in the field of approximate string matching, I rediscovered the works of Myers [1994] and Navarro and Baeza-Yates [2000] on approximate seeds, obtaining stronger filtration than exact seeds while preserving full-sensitivity under the edit distance. Their idea is to partition the pattern into fewer non-overlapping seeds, which obviously can be longer than exact seeds but have to be searched approximately. First, I slightly improved the filtration lemma of [Navarro and Baeza-Yates, 2000] to use approximate seeds with variable thresholds (see section 4.2). Then, as discussed in section 4.2, I chose to parameterize Masai's filter by the seed length rather than by the number of seeds. Indeed, the minimum seed length provides a direct estimate of the expected number of filtration specificity in terms of candidate locations. The resulting filter is thus flexible: by increasing the seed length, filtration becomes more specific at the expense of a higher filtration time. In practice, the optimal seed length depends on the reference genome as well as on read lengths and the absolute number of errors. In

section 6.2.4, I experimentally evaluate filtration schemes with exact and approximate seeds. When mapping current Illumina reads on short to medium length genomes, exact seeds are still more efficient than approximate seeds. Conversely, on larger genomes (e.g. mammalian genomes) 1-approximate seeds outperform exact seeds by an order of magnitude.

Following [Navarro and Baeza-Yates, 2000], I decided to find approximate seeds by backtracking the suffix trie of the reference genome. In section 6.1.2, I recall the engineering work I did to implement efficient backtracking of approximate seeds. In order to achieve faster filtration, I opted to find approximate seeds only under the Hamming distance. For this reason, when resorting to approximate seeds, Masai does not attain strict full-sensitivity under the edit distance. Nonetheless, such implementation choice sacrifices only 0.1% sensitivity (see section 6.2).

Best-mapping

Masai is a tool primarily designed to perform all-mapping rather than best-mapping. In best-mapping, Masai simply reports the first optimal mapping location encountered per read. Clearly, this policy makes sense if the edit distance is effective at identifying original mapping locations. The evaluation of section 6.2.1 shows that Masai is competitive in best-mapping with tools using more complex scoring schemes. Nonetheless, Masai's best-mapping method is ad-hoc and limited. In best-mapping, the tool does not compute mapping qualities nor supports the paired-end and mate-pair protocols. The reader is thus referred to the next chapter (section 7.1) for the complete description of an efficient best-mapping method that, in standard scenarios, is an order of magnitude faster than all-mapping.

6.1.2 Indexing

Initially, the SeqAn library provided me with only two indices capable of simulating a top-down suffix trie traversal: the enhanced suffix array (ESA) [Abouelhoda *et al.*, 2004] and the lazy suffix tree (LST) [Giegerich *et al.*, 1999]. To improve the efficiency of Masai, I implemented a generic top-down traversal for some additional indices, namely the suffix array (SA) [Manber and Myers, 1990], the q-gram index, and various specializations of the full-text minute index (FM-index) [Ferragina and Manzini, 2001]. Below I discuss the performance of these indices within Masai, while I refer the reader to chapter 3 for their extensive explanation.

Indexing the reference genome

I initially chose the ESA over the LST because of better construction times. Indeed, the ESA provides a linear time construction algorithm (an adaptation of the DC7 algorithm [Dementiev *et al.*, 2008] to multiple sequences [Weese, 2013] for the generalized SA, followed by the algorithms proposed in [Kasai *et al.*, 2001; Abouelhoda *et al.*, 2004]), while the LST construction algorithm takes quadratic time (using the radix sort based *wotd*-algorithm [Giegerich *et al.*, 1999]). Apart from that, both ESA and LST implementations

require 13 bytes per base pair and exhibit comparable query speed. Thus, Masai constructed the ESA of the human reference genome (GRCh38) in about 1.5 hours, consuming 39 GB of memory.

At this point, Masai required high-end hardware to process large reference genomes. Therefore, thinking about a space-time trade-off, I designed a generic suffix trie top-down traversal for the SA (see section 3.1.1). The SA consumes only 5 bytes per base pair but is theoretically slower than the ESA, as it adds a logarithmic factor to query times. However, with surprise, I found out that within Masai the SA had equal or better performance than the ESA (see table **??**). Ultimately, this change brought down the memory footprint of the index from 39 GB to 15 GB but preserved query speed.

I tried to further improve query speed by removing the logarithmic factor introduced by the SA. Therefore, to cut the most expensive binary searches, I put a q-gram index on top of the SA and extended my generic suffix trie top-down traversal accordingly (see section 3.1.3). Yet, the q-gram index did not bring significant speedup to the application; indeed, the lookup table turned out to be useful when searching patterns one by one, but not when coupled with the multiple backtracking algorithm.

Finally, I explored additional space-time trade-offs. Building on the work of [**?**], I implemented a generalized FM-index based on a wavelet tree [Grossi *et al.*, 2003]. This initial FM-index consumed about 1.5 bytes per base pair with a SA sampling of 10 %. Thus the memory footprint of the index went down to 4.5 GB, but Masai became almost twice as slow (see table **??**).

To sum up, I preferred the SA as it provides a good compromise between query speed and memory consumption. Nevertheless, Masai leaves to the user the possibility of choosing among the aforementioned data structures. Table **??** summarizes the runtime of the program with various indices.

Indexing the reads

In order to improve index query speed, I designed and implemented the algorithms presented in sections 3.3.5 and 3.3.6. These algorithms search simultaneously many exact or approximate seeds, achieving a speedup of 2–5 times over their conventional counterparts. As this multiple string matching algorithm requires a trie of the seeds, I also engineered an efficient trie implementation. A short explanation can be found in section 3.1.4

Building the SA via quicksort turned out to be faster than building the LST via radix sort but, within Masai, the more involved LST data structure payed off in terms of query time (see section 3.3.2). Indeed, the LST stores all trie nodes and thus provides node traversal in constant time, while the SA explicitly stores only the leaves and then derives internal nodes via binary search. As the memory footprint of the trie is negligible within this application, I chose the LST to perform multiple backtracking of approximate seeds.

When performing multiple backtracking of exact seeds, the LST construction time dominates the overall filtration time (see section 3.3.5). Therefore, I decided to resort to the *q-gram index* to emulate a trie in this case: I build a partial q-gram index efficiently and in linear time by bucket sort, again considering only the first suffix of each seed in the

collection. Such index represents a trie truncated at depth q (which I fixed to 12 in Masai). Truncation is only a minor concern: at depth q the search continues separately on each active node using the conventional search algorithms (see sections 3.3.3 and 3.3.3).

6.1.3 Verification

To verify candidate locations reported by the filtration algorithm, Masai employs a banded version of Myers bit-vector algorithm [Myers, 1999]. Myers' algorithm is an efficient DP alignment algorithm [Needleman and Wunsch, 1970] for edit distance. Instead of computing DP cells one after another, this algorithm encodes the whole DP column in two bit-vectors and computes the adjacent column in a constant number of 12 logical and 3 arithmetical operations. SeqAn provides a bit-parallel version that computes only a diagonal band of the DP matrix, faster and more specific than the original algorithm by Myers. More details can be found in section **??**.

RazerS 3 [Weese *et al.*, 2012] already uses Myers' algorithm. However, RazerS 3 performs one semi-global alignment to verify a parallelogram surrounding any seed occurrence. Conversely, Masai performs two global alignments on both ends of any seed occurrence. Given a seed occurring with e errors, the tool first performs seed extension on the left side within an error threshold of $k - e$ errors. Only if the seed extension on the left side succeeds, Masai performs a seed extension on the right side within the remaining error threshold. Moreover, the tool first computes the longest common prefix on each side of the seed extension and let the global alignment algorithm start from the first mismatching positions. This approach is up to two times faster than the one implemented by RazerS 3 (data not shown).

6.2 Evaluation

In order to evaluate Masai, I propose three experiments: the Rabema benchmark and variant detection on simulated data, and performance on real data. This evaluation focuses on the capability of the mappers to retrieve the location of a single read without the help of its paired-end, which can of course disambiguate some mapping locations. As references, I use whole genomes of *E. coli* (NCBI NC_000913.2), *C. elegans* (WormBase WS195), *D. melanogaster* (FlyBase release 5.42), and *H. sapiens* (GRCh37.p2).

I compare Masai in best-mapping with Bowtie 2, BWA and Soap 2, while in all-mapping with RazerS 3, Hobbes, mrFAST and SHRiMP 2. Bowtie 2, BWA, Soap 2 and SHRiMP 2 rely on scoring schemes taking into account base quality values, while Masai, RazerS 3, Hobbes and mrFAST use edit distance. When relevant, I configured some read mappers with the appropriate absolute number of errors (Masai, mrFAST, Hobbes, Soap 2) or error rate (RazerS 3). In section A.1, I give the exact parameterization of the read mappers considered in this evaluation.

6.2.1 Rabema benchmark on simulated data

I first consider the Rabema benchmark [Holtgrewe *et al.*, 2011] (v1.1) for a thorough evaluation and comparison of read mapping sensitivity. The benchmark contains the categories *all*, *all-best*, *any-best*, *precision*, and *recall*. In the categories all, all-best and any-best, a read mapper has to find all, all of the best, or any of the best edit distance locations for each read. The categories precision and recall require a read mapper to find the *original* location of each simulated read, which is a measure independent of the used scoring model, e.g. edit distance or quality based. A simulated read is mapped *correctly* if the mapper reports its original location, and it is mapped *uniquely* if the mapper reports only one location. Rabema defines *recall* to be the fraction of reads which were correctly mapped and *precision* the fraction of uniquely mapped reads that were mapped correctly.

Similarly to [Langmead and Salzberg, 2012], I used the read simulator Mason [Holtgrewe, 2010] with default profile settings to simulate, from each whole genome, 100 k reads of length 100 bp having sequencing errors distributed like in a typical Illumina run. I performed the benchmark for an error rate of 5 %, which corresponds to edit distance 5 for reads of length 100 bp. Therefore, I built a Rabema gold standard for each dataset by running RazerS 3 in full-sensitive mode up to edit distance 5. I further classified mapping locations in each category by their edit distance.

For a more fair and thorough comparison, I also consider BWA and Bowtie 2 in all-mapping (Soap 2 cannot be configured accordingly). To this extent, I parametrized these tools to be highly sensitive and output all found mapping locations. Since BWA and Bowtie 2 were not designed to be used in this way, they spent much more time than proper all-mappers, i.e. up to 3 hours in a run compared to several minutes. However, the aim of this experiment is to investigate read mapping sensitivity, therefore I do not report any running times. Table 6.1 shows the results on *H. sapiens*.

Best-mapping

Masai shows the best recall values, not loosing more than 2.3 % recall on edit distance 5. Conversely, the recall values of BWA and Bowtie 2 drop significantly with increasing edit distance and loose up to 15.4 % and 11.5 % on edit distance 5. As expected, Soap 2 turns out to be inadequate for mapping reads of length 100 bp at this error rates. Precision values have less variance than recall values. Masai shows the best precision values with 97.8 %, followed by Soap 2 with 97.7 %, and BWA with 97.5 %. Interestingly, Bowtie 2 shows the worst precision values, loosing up to 5.6 % on edit distance 5.

All-mapping

As expected, RazerS 3 shows full-sensitivity. In contrast, mrFAST looses a minimal percentage of mapping locations. Overall, Masai does not loose more than 0.1 % of all mapping locations. In particular, Masai is full-sensitive for low-error locations and looses only a small percentage of high-error locations, i.e. its loss is limited to 0.1 % and 1.4 % of mapping locations at edit distance 4 and 5.

Table 6.1: *Rabema benchmark results on* $100\,k \times 100\,bp$ *Illumina-like reads. Rabema scores are given in percent (average fraction of edit distance locations reported per read). Large numbers show total scores in each Rabema category and small numbers show the category scores separately for reads with* $\binom{0\ 1\ 2}{3\ 4\ 5}$ *errors.*

	method	all		all-best		any-best		precision		recall	
best-mappers	Masai	93.26	99.18 98.73 97.93 95.60 85.77 43.60	97.91	97.79 97.88 98.03 97.98 98.20 96.70	99.95	100.00 100.00 100.00 99.98 99.93 98.71	97.79	97.88 97.84 97.79 97.68 97.61 97.93	97.75	97.88 97.84 97.79 97.68 97.56 96.74
	Bowtie 2	92.04	99.18 98.72 96.80 93.44 81.94 40.19	96.16	97.79 97.85 95.80 94.83 93.37 88.86	98.08	100.00 99.96 97.55 96.62 94.93 90.46	96.58	98.01 97.72 95.98 95.19 95.22 94.37	95.94	98.01 97.72 95.55 94.24 92.79 89.52
	BWA	92.18	99.18 98.72 97.81 94.25 80.92 37.65	96.81	97.79 97.87 97.88 96.59 92.63 83.47	98.81	100.00 99.95 99.81 98.55 94.28 85.37	97.50	97.93 97.70 97.37 97.11 97.17 97.57	96.41	97.93 97.69 97.25 95.77 91.98 84.61
	Soap 2	65.93	99.18 95.55 91.34 8.67 0.70 0.00	69.89	97.79 94.74 91.37 8.98 0.79 0.00	71.37	100.00 96.78 93.18 9.21 0.81 0.00	97.69	98.05 97.74 97.73 94.87 84.13 91.67	69.91	98.05 94.62 91.20 11.85 1.41 0.36
all-mappers	Masai	99.90	100.00 100.00 100.00 100.00 99.94 98.58	99.96	100.00 100.00 100.00 100.00 99.93 98.71	99.96	100.00 100.00 100.00 100.00 99.93 98.71	100.00	100.00 100.00 100.00 100.00 100.00 100.00	99.96	100.00 100.00 100.00 100.00 99.93 98.78
	Bowtie 2	95.69	99.98 99.91 99.45 97.99 90.69 55.14	98.85	99.74 99.79 98.61 98.21 97.55 93.84	99.16	100.00 99.98 99.01 98.63 97.94 94.17	99.84	100.00 99.95 99.87 99.64 99.67 99.29	98.54	99.74 99.58 98.27 97.64 96.87 94.40
	BWA	95.89	99.96 99.88 99.49 97.13 87.79 64.11	97.98	98.81 99.01 99.02 97.83 93.95 85.20	98.82	100.00 99.95 99.82 98.56 94.34 85.37	98.12	93.21 97.63 98.36 98.49 98.68 99.56	97.80	99.03 98.96 98.75 97.35 93.43 86.36
	RazerS 3	100.00	100.00 100.00 100.00 100.00 100.00 100.00	100.00	100.00 100.00 100.00 100.00 100.00 100.00	100.00	100.00 100.00 100.00 100.00 100.00 100.00	100.00	100.00 100.00 100.00 100.00 100.00 100.00	100.00	100.00 100.00 100.00 100.00 100.00 100.00
	Hobbes	96.56	99.41 99.00 98.76 97.80 93.20 73.05	97.08	97.23 96.59 97.01 97.38 98.16 97.42	98.01	97.92 97.51 97.96 98.43 99.12 98.46	99.97	99.96 99.97 99.97 99.98 99.95 99.96	96.41	95.49 95.84 96.54 97.03 97.98 97.79
	mrFAST	99.97	100.00 100.00 100.00 100.00 99.99 99.53	99.97	100.00 100.00 100.00 100.00 100.00 99.10	99.97	100.00 100.00 100.00 100.00 100.00 99.13	100.00	100.00 100.00 100.00 100.00 100.00 100.00	99.97	100.00 100.00 100.00 99.99 100.00 99.18
	SHRiMP 2	96.53	99.87 99.82 99.53 98.37 92.58 64.63	99.50	99.34 99.50 99.60 99.64 99.65 98.32	99.85	99.87 99.90 99.91 99.89 99.84 98.57	99.95	100.00 100.00 100.00 99.93 99.89 99.23	99.25	99.35 99.30 99.24 99.30 99.09 98.48

Conversely, BWA and Bowtie 2 miss 35 % and 45 % of all mapping locations at edit distance 5 and their recall values as all-mappers do not substantially increase. Likewise, SHRiMP 2 is not able to enumerate all mapping locations, although its recall values are good. Again, Hobbes has the worst performance.

As mentioned in section **??**, Masai is not full-sensitive whenever approximate seeds are used, e.g. on *H. sapiens*. Indeed, Masai loses 0.1 % overall sensitivity in respect to RazerS 3. In general, RazerS 3 should be used when full-sensitivity is required, i.e. in read mapping benchmarks. However, these results show that Masai can replace RazerS 3 or mrFAST in practical all-mapping setups.

6.2.2 Variant detection on simulated data

The second experiment analyzes the theoretical performance of Masai and other read mappers in genomic variation pipelines. Similarly to [David *et al.*, 2011], this experiment considers simulated reads containing sequencing errors, SNPs and indels. Each simulated read has an edit distance of at most 5 to its genomic origin, and it is grouped according to the number of contained SNPs and indels, where class (s, i) consists of all reads with s SNPs and i indels. The experiment considers a read to be mapped *correctly* if a mapping location is reported within 10 bp of its genomic origin; it considers a read to map *uniquely* if only one location is reported by the mapper. For each class, the experiment defines *recall* to be the fraction of reads which were correctly mapped and *precision* the fraction of uniquely mapped reads that were mapped correctly.

I simulated 5 million Illumina-like reads of length 100 bp from the whole human genome using Mason. I mapped the reads with each tool and measured its sensitivity in each class. Table 6.2 shows the results.

Table 6.2: *Variant detection results on* $5\,M \times 100$ *bp Illumina-like reads. The table shows percentages of found origins (recall) and fraction of unique reads mapped to their origin (precision) classed by reads with* s *SNPs and* i *indels* (s, i).

		(0,0)		(2,0)		(4,0)		(1,1)		(1,2)		(0,3)	
	method	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.
best-mappers	Masai	98.2	98.2	97.6	97.5	96.8	96.8	97.8	97.2	97.9	97.9	97.2	97.2
	Bowtie 2	97.6	97.3	94.6	92.0	92.6	82.5	95.3	93.3	93.5	92.3	96.1	95.4
	BWA	98.2	97.9	97.6	95.3	94.9	85.1	97.4	90.9	97.1	80.3	96.3	66.5
	Soap 2	98.1	82.9	97.4	31.0	0.0	0.0	90.6	6.2	0.0	0.0	0.0	0.0
all-mappers	Masai	100.0	100.0	100.0	99.9	100.0	100.0	100.0	99.3	100.0	100.0	100.0	100.0
	RazerS 3	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	Hobbes	99.9	99.9	99.9	99.9	100.0	100.0	100.0	99.8	100.0	93.6	99.6	90.5
	mrFAST	100.0	99.9	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	SHRiMP 2	100.0	99.4	100.0	99.7	100.0	99.7	100.0	99.5	100.0	99.2	100.0	99.6

Best-mapping

Masai shows the highest precision and recall in all best-mapping classes. In particular, Masai does not loose more than 3.2 % recall in class (4,0), whether Bowtie 2 and BWA loose respectively 17.5 % and 14.9 % and Soap 2 is not able to map any read. The recall values of Bowtie 2, BWA and Soap 2 are negatively correlated with the amount of genomic variation. For instance, in the Rabema benchmark, Bowtie 2 looses respectively 7.2 % and 11.5 % of mapping locations at distance 4 and 5, but in class (4,0) of this experiment it looses 17.5 % recall. A similar trend is observable for BWA and Soap 2. The low performance of Soap 2 is also due to its limitation to at most 2 mismatches and no support for indels.

All-mapping

Looking at all-mapping results, Masai shows 100 % precision and recall in all classes, except for classes (2,0) and (1,1) where it looses only 0.1 % and 0.7 % recall. Masai is therefore roughly comparable to the full-sensitive read mappers RazerS 3 and mrFAST. SHRiMP 2 shows 100 % precision in all classes but looses between 0.3 % and 0.8 % recall in each class. Hobbes has the lowest performance among all-mappers: it appears to have problems with indels, indeed it looses 9.5 % recall in class (0,3).

6.2.3 Performance on real data

The last experiment focuses on comparing read mappers performance on real data. I mapped the first 10 M×100 bp reads from an Illumina lane of *E. coli* (ERR022075, Genome Analyzer IIx), *D. melanogaster* (SRR497711, HiSeq 2000), *C. elegans* (SRR065390, Genome Analyzer II), and *H. sapiens* (ERR012100, Genome Analyzer II). Whenever possible, I configured the tools to map the reads within edit distance 5. I measured mapping times on a cluster of nodes with 72 GB RAM and 2 Intel Xeon X5650 processors running Linux 3.2.0.

For an accurate running time comparison, I ran the tools using a single thread and used local disks for I/O. I measured running times and peak memory consumptions.

I cannot measure precision and recall values as real reads have unknown origins. Therefore, for this evaluation, I adopt the commonly used measure of percentage of *mapped reads*, i.e. the fraction of reads for which the read mapper reports a mapping location. However, as some mappers report mapping locations without constraints on the number of errors, I also include Rabema *any-best* scores. As explained in section 6.2.1, the Rabema any-best benchmark assigns a point for a read if the mapper reports at least one mapping location with the optimal (i.e. minimal) number of errors; final Rabema any-best scores are normalized by the number of reads. Results for *C. elegans* and *H. sapiens* are shown in table 6.3.

Best-mapping

On the *C. elegans* dataset, Masai is 7.7 times faster than Bowtie 2, 8.2 times faster than BWA and 1.5 times faster than Soap 2. On the *H. sapiens* dataset, Masai is 2.6 times faster than Bowtie 2, 3.6 times faster than BWA but 2.1 times slower than Soap 2. On one end, Soap 2 is not able to map a consistent fraction of reads because of its limitation to 2 mismatches. On the other end, Bowtie 2 reports more mapped reads than Masai but, taking any-best scores into account, it reports less mapping locations than Masai. In fact, Bowtie 2 uses a scoring scheme based on quality values and does not impose a maximal error rate threshold. Despite that, Bowtie 2 misses respectively 22.0 % and 20.7 % of reads mappable at edit distance 5 on the *C. elegans* and *H. sapiens* datasets.

All-mapping

On the *C. elegans* dataset, Masai is 2.0 times faster than RazerS 3, 10.9 times faster than Hobbes, 6.3 times faster than mrFAST and 50.1 times faster than SHRiMP 2. Hobbes constantly crashes and maps less reads than all other mappers in this category. On the *H. sapiens* dataset, Masai is 11.9 times faster than RazerS 3, 14.6 times faster than mrFAST, and 7.6 times faster than Hobbes. The current version of Hobbes constantly crashes and maps only half of the reads. SHRiMP 2 is not able to map the *H. sapiens* dataset within 4 days. Likewise for Bowtie 2, also SHRiMP 2 does not impose a maximal error rate threshold and reports more mapped reads than Masai. However, its Rabema any-best score is inferior to Masai. This could be due to the use of a different scoring scheme where two mismatches cost less than opening a gap. Anyway, this hypothesis does not explain why SHRiMP 2 does not report some mapping locations at distance 0.

6.2.4 Filtration efficiency

This experiment assesses the contribution of approximate seeds and multiple backtracking on runtime results. To this intent, I performed all-mapping with Masai on each previously considered dataset, this time using either exact or approximate seeds in combination with either single or multiple backtracking. Table 6.4 shows the results. Filtration time consists of the time spent to index the seeds (in case of multiple backtracking) and

Table 6.3: *Performance on real data using* $10\,M \times 100\,bp$ *Illumina reads.*
Rabema any-best: in large are shown the percentage of reads mapped with the minimal number of errors (up to 5%) and in small the percentage of reads that were mapped with $\left(\begin{smallmatrix} 0 & 1\% & 2\% \\ 3\% & 4\% & 5\% \end{smallmatrix}\right)$ *errors.*
Mapped reads: in large are shown the percentage of mapped reads and in small the cumulative percentage of reads that were mapped with $\left(\begin{smallmatrix} 0 & 1\% & 2\% \\ 3\% & 4\% & 5\% \end{smallmatrix}\right)$ *errors.*
Remarks: SHRiMP 2 is not able to map the H. sapiens *dataset within 4 days; Hobbes constantly crashes and is not able to map completely nor the* C. elegans *nor the* H. sapiens *dataset.*

	dataset	SRR065390 C. elegans				ERR012100 H. sapiens			
	method	time [min:s]	memory [Mb]	Rabema any-best [%]	mapped reads [%]	time [min:s]	memory [Mb]	Rabema any-best [%]	mapped reads [%]
best-mappers	Masai	3:10	2936	100.00 100.00 100.00 100.00 100.00 100.00	89.49 75.01 83.80 86.38 87.83 88.79 89.49	22:35	19711	99.99 100.00 100.00 100.00 99.94 99.91 99.53	93.76 75.99 87.84 90.67 92.02 92.99 93.76
	Bowtie 2	24:14	135	99.21 100.00 99.30 93.38 88.61 84.03 77.96	92.58 75.01 83.74 86.20 87.61 88.57 89.27	57:41	3180	99.45 100.00 99.75 96.02 92.88 87.86 79.26	96.72 75.99 87.81 90.54 91.85 92.76 93.44
	BWA	25:53	325	99.33 100.00 99.09 95.57 89.70 85.86 82.29	89.33 75.01 83.72 86.25 87.64 88.59 89.32	80:58	4475	99.54 100.00 99.50 98.01 93.39 88.92 84.42	93.53 75.99 87.78 90.59 91.91 92.82 93.53
	Soap 2	4:37	748	95.98 100.00 96.57 92.38 0.33 0.04 0.02	85.95 75.01 83.50 85.94 85.95 85.95 85.95	11:11	5357	95.66 100.00 94.94 86.54 0.32 0.16 0.16	89.73 75.99 87.24 89.73 89.73 89.73 89.73
all-mappers	Masai	10:49	2821	100.00 100.00 100.00 100.00 100.00 100.00 100.00 100.00	89.49 75.01 83.80 86.38 87.83 88.79 89.49	307:16	20130	100.00 100.00 100.00 100.00 100.00 99.97 99.53	93.76 75.99 87.84 90.67 92.02 92.99 93.76
	RazerS 3	21:18	11489	100.00 100.00 100.00 100.00 100.00 100.00 100.00 100.00	89.49 75.01 83.80 86.38 87.83 88.79 89.49	3653:03	17298	100.00 100.00 100.00 100.00 100.00 100.00 100.00 100.00	93.77 75.99 87.84 90.67 92.02 92.99 93.77
	Hobbes	117:46	3885	89.77 91.04 80.63 86.47 88.32 88.47 85.17	80.34 68.29 75.38 77.61 78.89 79.74 80.34	2319:27	71685	59.02 59.24 58.65 57.48 56.92 56.70 56.32	55.35 45.01 51.96 53.59 54.36 54.91 55.35
	mrFAST	67:41	875	99.99 100.00 99.98 99.88 99.87 99.93 99.51	89.49 75.01 83.80 86.38 87.83 88.78 89.49	4462:25	929	99.98 100.00 100.00 100.00 99.99 99.99 97.46	93.75 75.99 87.84 90.67 92.02 92.99 93.75
	SHRiMP 2	541:20	2735	98.51 99.59 96.81 91.76 87.60 81.88 74.77	91.91 74.71 83.22 85.59 86.88 87.69 88.27	–	–	–	–

to perform backtracking. Candidates reports the number of candidate locations reported by the filter for which seed extension is subsequently performed. I report in bold the optimal combination of seeding and backtracking that I used to parameterize Masai.

Since this experiment focuses on filtration, I do not consider the time spent performing seed extensions and I/O, i.e. loading the reference genome and its index, loading the reads, writing the results. Such time is independent of any combination of seeding or backtracking and can be extrapolated by subtracting bold filtration times of table 6.4 from respective Masai all-mapping times of table 6.3 and table **??**.

On *E. coli*, *D. melanogaster* and *C. elegans*, approximate seeds reduce the number of candidates respectively by 2.1 times, 9.9 times, and 4.3 times. Nevertheless I still prefer exact seeds as filtration dominates the total runtime. Multiple backtracking on exact seeds compared to single backtracking speeds up filtration by 2.9 times on *E. coli*, and 3.8 times on *D. melanogaster* and *C. elegans*. Without the contribution of multiple backtracking, Masai would not be faster than RazerS 3, the second fastest all-mapper.

Approximate seeds become effective on *H. sapiens*, where they reduce the number of candidates by 10.8 times. On *H. sapiens*, seed extensions largely dominate the total runtime, therefore I prefer approximate seeds. Multiple backtracking on approximate seeds provides a speed-up of 3.2 times over single backtracking. The combination of the two methods makes Masai an order of magnitude faster than any other all-mapper.

Table 6.4: *Masai filtration efficiency results for all-mapping. Filtration time is given as [min:s] and includes seeds indexing time.*

organism	dataset	seeding	backtracking	filtration time	candidates
E. coli	ERR022075	exact	single	3:55	69.17 M
E. coli	ERR022075	**exact**	**multiple**	**1:20**	**69.17 M**
E. coli	ERR022075	approximate	single	38:42	33.08 M
E. coli	ERR022075	approximate	multiple	9:00	33.08 M
D. melanogaster	SRR497711	exact	single	8:15	1020.28 M
D. melanogaster	SRR497711	**exact**	**multiple**	**2:11**	**1020.28 M**
D. melanogaster	SRR497711	approximate	single	100:18	102.78 M
D. melanogaster	SRR497711	approximate	multiple	20:48	102.78 M
C. elegans	SRR065390	exact	single	8:25	1065.70 M
C. elegans	SRR065390	**exact**	**multiple**	**2:11**	**1065.70 M**
C. elegans	SRR065390	approximate	single	102:02	246.65 M
C. elegans	SRR065390	approximate	multiple	21:33	246.65 M
H. sapiens	ERR012100	exact	single	55:54	294943.86 M
H. sapiens	ERR012100	exact	multiple	41:52	294943.86 M
H. sapiens	ERR012100	approximate	single	165:45	27396.01 M
H. sapiens	ERR012100	**approximate**	**multiple**	**52:15**	**27396.01 M**

6.2.5 Runtime with different indices

6.3 Discussion

Masai consists of three important algorithmic methods: approximate seeds, multiple backtracking and best-mapping filtration. Approximate seeds are of paramount importance to obtain very specific, yet full-sensitive filtration; their adoption speeds up Masai by one order of magnitude. Multiple backtracking further speeds up the filtration phase by 3–5 times on a (enhanced) suffix array index; this technique makes Masai twice as fast. Best-mapping filtration prioritizes the analysis of optimal mapping locations, such that the program can stop whenever it finds one best location; because of this method, Masai in best-mapping is an order of magnitude faster than in all-mapping.

Is the edit distance sufficient to perform best-mapping? Both Rabema benchmark and variant detection results show that Masai has constantly better accuracy than other best-mappers relying on more complex scoring schemes. In particular, the Rabema benchmark results show that Rabema any-best values are tightly bound to recall values. Hence, the edit distance is a pertinent and adequate scoring scheme for best-mapping. Vice versa, best-mappers using scoring schemes based on quality values show a generalized and substantial loss of mapping accuracy. This is likely due to the heuristics on which these tools rely. To sum up, it is better to stick to edit distance and guarantee full-sensitivity rather than to adopt an involved scoring scheme and explore the alignment space heuris-

tically, hence partially.

How many mapping locations do heuristic best-mappers miss? By looking at precision and recall values on simulated data, or at Rabema any-best values on real data, it can be deduced that Bowtie 2, BWA and Soap 2 miss up to 20 % of reads mappable at 5 % error rate. Yet, it is not evident how these results affects variant calling pipelines.

Summing up, Masai in all-mapping is an order of magnitude faster and thus a valid alternative to tools like RazerS 3 and mrFast. Computational requirements of all-mapping are now close to those of best-mapping. Indeed, Masai in all-mapping is only 4 times slower than BWA in best-mapping, despite reporting two orders of magnitude more mapping locations. Masai in best-mapping is 2–4 times faster and more accurate than Bowtie 2 [Langmead and Salzberg, 2012] and BWA [Li and Durbin, 2009]. The achieved speedup is huge when RazerS 3 is used for best-mapping: in this scenario, Masai is roughly 200 times faster!

Despite these good results, Masai is not being widely used. This is mainly because the tool lacks some commonly requested features, including: parallelization via multithreading, low memory footprint, direct support of paired-end or mate-pair protocols, computation of mapping qualities, automatic parameterization. Because of my initial inexperience and unclear or wrong design goals, I neglected these features while engineering the tool. The next chapter introduces *Yara*, *y*et *a*nother *r*ead *a*ligner, a tool fulfilling these requirements.

CHAPTER

7 Yara

Yara (*Y*et *a*nother *r*ead *a*ligner) is an exhaustive, non-heuristic read mapper, capable of quickly reporting all stratified mapping locations within a given error rate. Yara works with Illumina or Ion Torrent reads, supports paired-end and mate-pair protocols, computes accurate mapping qualities, offers parallelization via multi-threading, has a low memory footprint thanks to the FM-index, and does not require ad-hoc parameterization.

7.1 Engineering

7.1.1 Adaptive filtration

Specific yet rapid filtration is fundamental in the design of an efficient read mapping tool. Read mappers like RazerS 3 [Weese *et al.*, 2012] and mrFast [Ahmadi *et al.*, 2012] are designed around naïve filtration with exact seeds. This filtration method is always very quick, however it is not specific enough on long, repetitive reference genomes like the human genome. Masai [Siragusa *et al.*, 2013] circumvents this problem by enforcing a minimum seed length, whose optimal value must be tuned for a specific reference genome, and eventually resorting to approximate seeds in order to guarantee full-sensitivity. This filtration method speeds up Masai by an order of magnitude but has some drawbacks: it needs external parametrization, lacks flexibility and is suboptimal in practice.

Yara applies an adaptive filtration scheme *per read* because, under any fixed filtration scheme, the number of verifications per read is not uniform: within a typical human genome resequencing, most reads produce very few verifications and are easily mappable, while few other reads are problematic and often not even confidently mappable to one single location. Consequently, any fixed filtration scheme turns out to be too weak for some reads yet too strong for others, thus suboptimal in practice. An adaptive filtration scheme per read improves filtration efficiency by optimizing the ratio between filtration speed and specificity. Yara thus automatically chooses an adaptive filtration scheme per read, without requiring manual parameterization by the user.

Adaptive filtration works as follows. Yara initially applies filtration with exact seeds to all reads. The tool counts the number of verifications to be performed for each read, thus decides if it is worth proceeding with the verification phase or alternatively applying a stronger filtration scheme. This decision depends on fine-tuned internal verification thresholds. Under standard Illumina setups, exact seeds provide efficient filtration for

up to 70–80 % of the reads; on the remaining reads, a filtration scheme using 1– or 2–approximate seeds works better. Thus, Yara starts with the quickest filtration scheme and becomes more specific whenever it pays off to do so.

7.1.2 Stratified mapping

All-mapping methods consider a set of relevant mapping locations per read. Yet, this definition leaves open what relevant means. In all-mapping under the edit distance, the user defines relevant mapping locations by imposing a distance threshold. Despite being sound, this definition does not work well in practice. On the one hand a very low threshold leaves a consistent fraction of the reads unmapped, on the other hand a moderate threshold produces a deluge of mapping locations for some reads. In practice, at 5 %, error rate, Illumina reads map on average to hundreds of mapping locations on the human genome. It is questionable whether all these locations are relevant for the downstream analysis. Thus, a finer definition of all-mapping relevance is necessary in practice.

Stratification of mapping locations yields an equally sound yet practical definition of all-mapping under the edit distance. The e-stratum

$$S(r, e) = \{(i, j, e) : d_E(g_{i\ldots j}, r) = e\} \tag{7.1}$$

denotes the set of all mapping locations of a read r at edit distance e from the reference genome g. According to the above definition, conventional all-mapping under the edit distance defines the set

$$S(r, 0) \cup S(r, 1) \cup \ldots \cup S(r, k) \tag{7.2}$$

as relevant mapping locations within an *absolute* error threshold k. Stratified all-mapping refines this definition by considering only mapping locations being co-optimal, or sub-optimal up to a certain degree. Formally, if the distance of any optimal mapping location for read r is

$$e^* = \min\{e \in [0, k] : S(r, e) \neq \emptyset\} \tag{7.3}$$

stratified all-mapping considers mapping locations

$$S(r, e^*) \cup \ldots \cup S(r, \min\{e^* + l, k\}) \tag{7.4}$$

within a *relative* error threshold l to be relevant.

Efficient filtration by strata

Yara significantly improves the runtime of stratified all-mapping over conventional all-mapping. Obviously, the most straightforward way to achieve stratified all-mapping consists into performing conventional all-mapping and subsequently filtering out any irrelevant mapping location. For instance, RazerS 3 implements this method and gets no speedup. Another naïve method consists into applying up to k rounds of conventional

all-mapping, using filtration schemes for thresholds going from 0 to $\min\{e^* + l, k\}$. It is easy to see that also this method performs redundant computation: the total work for any read that maps at distance greater or equal to k corresponds to the sum of all k filtration schemes.

The following stratified all-mapping method, implemented in Yara, guarantees not to perform more work than conventional all-mapping. Indeed, the key idea is to simply reduce any filtration scheme full-sensitive within distance k to be full-sensitive within distance $\min\{e^* + l, k\}$. Given a filtration scheme $\mathbb{k} = (k_1, \ldots, k_s)$ full-sensitive within distance k, any subset consisting of $s^* \leq s$ seeds with thresholds $\mathbb{k}^* = (k_1^*, \ldots, k_{s^*}^*)$ is full-sensitive within distance $\min\{e^* + l, k\}$ if it satisfies

$$s^* + \sum_{i=1}^{s^*} k_i^* > \min\{e^* + l, k\}. \tag{7.5}$$

The proof is analogous to that one given in section 4.2.

Example 7.1. Let $k = 5$ be the absolute threshold, and $l = 0$ be the relative threshold, restricting relevant locations to co-optimal ones. A read r maps at distance 1, i.e. $|S(r, 0)| = 0$ and $|S(r, 1)| > 0$, thus $e^* = 1$. Given the filtration scheme $\mathbb{k} = (1, 1, 1)$, any subset full-sensitive up to distance $\min\{e^*+l, k\} = 1$ finds all relevant mapping locations. All full-sensitive subsets of $\mathbb{k}$ are $(0, 0, -)$, $(0, -, 0)$, $(-, 0, 0)$, $(1, -, -)$, $(-, 1, -)$, $(-, -, 1)$, where a $-$ at position i indicates that the i-th seed is unnecessary. Thus, the verification of all candidates, either produced by *any* two exact seeds or by one 1-approximate seed, yields all mapping locations in the 1-stratum of r.

Greedy verification strategy

In addition, Yara implements a simple greedy strategy to minimize the number of verifications necessary to find all relevant stratified mapping locations. As candidate locations can be verified in any order, Yara chooses an ordering of seeds producing the minimum number of verifications. The tool first finds all seeds and ranks them by number of candidate locations produced. Then it processes all candidate locations, from the least to the most frequent seed, until it explores l strata from the first non-empty one, or until it attains the absolute distance threshold k.

Yara performs best-mapping by means of stratified filtration. Best-mapping requires one primary mapping location along with its confidence. Under the edit distance, without any further assumptions, any co-optimal location is equally likely to be correct. Thus, Yara performs stratified all-mapping with a relative threshold $l = 0$, picks one random co-optimal location, and subsequently estimates its mapping quality using all found mapping locations (see section 7.1.4). Thanks to this method, Yara in best-mapping is an order of magnitude faster than in all-mapping.

7.1.3 Paired-end and mate-pair protocols

Paired-end and mate-pair protocols are the sequencing protocols of choice of Illumina instruments. Reads are sequenced in pairs from both ends of the same insert. Properly

paired reads are expected to map within the insert size adopted in the sequencing protocol. Thus, the added information of an expected insert size allows a read mapper to map read pairs to their original locations more confidently than in the single-end protocol. Nonetheless, a read mapper should report equally important unpaired mapping locations, as the lack of any proper pair of mapping locations signals a potential structural variation, e.g. a long indel or an inversion.

In the paired-end or mate-pair workflows, Yara maps paired reads independently, exactly as in the single-end workflow, and reports all relevant mapping locations per read. However, in addition to the single-end workflow, Yara implements a finer strategy to choose primary mapping locations. For any reads pair, among all pairs of co-optimal mapping locations, the tool selects the one with minimal deviation from the expected insert size. Since Yara outputs all relevant mapping locations, the choice of primary locations can be always corrected a posteriori.

7.1.4 Mapping qualities

Yara computes mapping qualities using the number of mapping locations stratified by error rate.

7.1.5 Indexing

Yara uses an efficient FM-index specialization for the DNA alphabet, based on interleaved rank dictionaries (see section 3.2.2). This interleaved FM-index exhibits a fourfold speedup over the first FM-index implementation bundled with Masai. Surprisingly, the interleaved FM-index is also faster than any other index, both in exact and approximate search (see section 3.3). Moreover, this index consumes only 1.23 bytes per base pair with a SA sampled at 10 %, thus its memory footprint for the human genome is 3.7 GB. Under these terms, there is no space-time indexing trade-off: the FM-index always provides the most convenient suffix trie implementation.

Yara does not use the multiple search algorithms of section 3.3.6 to search seeds on the FM-index. As shown in section 3.3, on the FM-index it is always faster to search exact queries in a naïve way and approximate queries after sorting them in lexicographical order. This fact considerably simplifies a fine-grained parallelization of the tool.

7.2 Evaluation

The evaluation consists of three experiments: accuracy on simulated data, Rabema benchmark on simulated and real data, and throughput on real data. In each experiment, I compare Yara in best-mapping with GEM, Bowtie 2 and BWA, while in all-mapping with GEM, RazerS 3, and Hobbes 2.

7.2.1 Experimental setup

Read mappers parametrization

In appendix A.2, I give the exact parametrization of each read mapper considered in the evaluation. Whenever possible, I configured the tools with the appropriate error rate (Yara, GEM, RazerS 3) or absolute number of errors (Hobbes 2). When processing paired-end reads, I provided the tools with appropriate insert size information.

Infrastructure

All tools run on a desktop computer running Linux 3.10.11, equipped with one Intel® Core i7-4770K CPU @ 3.50 GHz, 32 GB RAM and a 2 TB HDD @ 7200 RPM. For maximum throughput, all tools run using eight threads. For accurate running time comparisons, I disabled Intel Turbo Boost; therefore, real throughputs might be slightly higher than the measured ones.

Datasets

The reference in all experiments is the human whole genome (GRCh38). The simulated data consists of 1 M Illumina-like 2×100 bp paired-end reads, simulated from the reference genome using Mason [Holtgrewe, 2010]. The mean insert size is `INS = 175` and the deviation `ERR = 225`. The real data is a publicly released sequencing run (SRA/ENA id: ERR161544) by the Beijing Genome Institute; the genomic DNA used in this study came from an anonymous male Han Chinese individual who has no known genetic diseases. This dataset consists of 2×100 bp whole genome sequencing reads, produced by an Illumina HiSeq 2000 instrument. BWA reports mean insert size `INS = 178` and deviation `ERR = 200`.

7.2.2 Accuracy on simulated data

I introduce an *accuracy benchmark* to measure the accuracy of read mappers in finding the *original* location of simulated reads. This benchmark considers for each read only the *first primary mapping location* encountered while scanning the SAM file [Li *et al.*, 2009a] produced by each tool; this policy mimics the behavior of de-facto standard *best-mapping analysis pipelines*, e.g. the GATK [DePristo *et al.*, 2011]. This accuracy benchmark counts a simulated read as *correctly mapped* if the found mapping location corresponds to its original location, it computes the *recall* of each tool as the fraction of correctly mapped reads and the *precision* as the fraction of mapped reads that are correct; for a thorough evaluation, it also classes all simulated reads by the *error rate* of their original locations, then computes recall and precision within each error rate class.

I repeated this experiment twice: first providing the simulated reads as unpaired, then as paired-end with additional insert size information. Results are shown in table 7.1.

Table 7.1: *Accuracy results on the human whole genome. The left panel shows the results of mapping 1 M Illumina-like 2 × 100 bp reads as unpaired; the right panel shows the results of mapping the same reads as paired-end, providing additional insert size information. Big numbers show total scores, while small numbers show marginal scores for the reads at $\binom{0\ 1\ 2}{3\ 4\ 5}$ % error rate.*

		hiseq_hg19_like_se H. sapiens SE								hiseq_hg19_like_pe H. sapiens PE							
	tool	recall				precision				recall				precision			
best	Yara	97.67	97.70	97.62	97.57	97.67	97.70	97.62	97.57	98.50	98.55	98.42	98.28	98.50	98.55	98.42	98.28
			97.32	97.76	97.03		97.32	97.82	97.19		97.90	98.51	97.61		97.90	98.56	97.77
	Gem	97.65	97.69	97.56	97.56	97.65	97.69	97.56	97.56	98.50	98.54	98.48	98.34	98.50	98.54	98.48	98.34
			96.54	96.86	97.20		96.56	96.91	97.20		97.49	97.71	98.43		97.49	97.71	98.43
	Bowtie 2	97.56	97.70	97.61	95.60	97.59	97.70	97.61	96.01	98.46	98.50	98.47	98.08	98.47	98.50	98.47	98.12
			94.27	94.68	95.80		95.13	95.44	96.35		97.58	96.59	97.53		97.74	96.96	98.17
	BWA	97.53	97.69	97.58	97.28	97.64	97.69	97.58	97.35	98.39	98.54	98.47	98.19	98.50	98.54	98.47	98.26
			90.86	76.53	68.01		96.30	95.80	97.40		91.56	77.06	68.34		97.05	96.47	97.87
all	Yara	97.67	97.70	97.61	97.61	97.67	97.70	97.61	97.61	98.50	98.55	98.42	98.27	98.50	98.55	98.42	98.27
			97.18	97.50	96.95		97.18	97.55	97.11		97.90	98.56	97.61		97.90	98.62	97.77
	Gem	97.65	97.70	97.57	97.58	97.65	97.70	97.57	97.58	98.50	98.53	98.45	98.35	98.50	98.53	98.45	98.35
			96.59	96.86	97.20		96.59	96.91	97.20		97.41	97.66	98.43		97.41	97.66	98.43
	Hobbes 2	98.34	98.38	98.29	98.20	90.00	89.63	90.68	91.74	89.22	89.10	89.46	89.99	89.92	89.60	90.47	91.53
			97.56	97.76	97.77		91.83	94.01	95.18		89.41	87.33	84.09		91.92	93.72	96.41
	RazerS 3	91.00	90.52	91.90	93.14	91.00	90.52	91.90	93.14	94.13	93.95	94.48	94.95	94.17	93.98	94.51	94.99
			94.01	96.11	96.87		94.01	96.11	96.87		95.40	96.38	92.75		95.53	96.79	97.74

7.2.3 Rabema benchmark on simulated and real data

The *Rabema benchmark* [Holtgrewe *et al.*, 2011] (v1.1) measures the sensitivity of read mappers in finding *relevant* mapping locations of simulated or real reads. This experiment considers the Rabema benchmark category *all* for all-mappers and *all-best* for best-mappers. In the category all, Rabema counts as relevant, for each read, all mapping locations within a maximal edit distance error rate; in the category all-best it considers just co-optimal mapping locations. Rabema computes the *sensitivity* of each tool as the fraction of relevant mapping locations found per read. Analogously to the accuracy benchmark, Rabema classes mapping locations by their *error rate*, then computes sensitivity within each error rate class. The benchmark reports percentual scores normalized by the number of reads.

I applied the Rabema benchmark both on simulated and real data, within an error rate of 5 %. Therefore, I built a Rabema gold standard for each dataset by running RazerS 3 in full-sensitive mode up to 5 % error rate. I provided unpaired reads to each tool as the Rabema benchmark by definition does not consider paired-end reads. Results are shown in table 7.2.

7.2.4 Throughput on real data

This experiment complements the sensitivity evaluation on real data provided by the Rabema benchmark. The goal of this experiment is to determine if read mappers are able to sustain sequencing throughput of the instrument or constitute a potential bottleneck in the data analysis pipeline. This experiment measures read mapping throughput in *giga*

Table 7.2: *Rabema benchmark results on the human whole genome. The left panel shows the results of mapping* 1 *M Illumina-like* 2 × 100 *bp simulated reads; the right panel shows the results of mapping* 1 *M Illumina* 2 × 100 *bp real reads. Big numbers show total Rabema scores, while small numbers show marginal scores for the mapping locations at* $\left(\begin{smallmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{smallmatrix}\right)$ *% error rate.*

		hiseq_hg19_like H. sapiens			ERR161544 H. sapiens		
	tool	all	all-best	any-best	all	all-best	any-best
best	Yara	92.09 100.00 97.99 85.94 / 40.28 10.10 2.56	100.00 100.00 100.00 100.00 / 100.00 99.94 99.71	100.00 100.00 100.00 100.00 / 100.00 99.94 99.83	90.49 100.00 94.64 75.66 / 50.19 30.86 14.27	99.99 100.00 100.00 100.00 / 99.96 99.88 99.40	99.99 100.00 100.00 100.00 / 99.96 99.90 99.53
best	Gem	93.71 100.00 98.73 90.88 / 61.29 31.48 15.01	99.99 100.00 99.99 99.95 / 99.84 99.23 97.97	100.00 100.00 100.00 99.98 / 99.95 99.71 98.39	93.25 100.00 96.64 84.54 / 72.12 55.35 31.75	99.94 100.00 99.98 99.91 / 99.71 99.48 94.16	99.95 100.00 99.99 99.97 / 99.84 99.77 95.12
best	Bowtie 2	90.98 98.86 96.87 83.98 / 38.88 9.79 2.47	97.60 97.67 97.81 95.73 / 94.89 95.59 95.64	99.88 100.00 99.99 97.74 / 96.83 97.14 97.72	88.40 98.59 92.21 70.69 / 44.59 25.30 9.41	95.82 97.17 95.00 87.74 / 81.88 76.13 64.68	99.57 100.00 99.94 97.32 / 94.98 90.74 81.12
best	BWA	90.93 98.86 96.87 84.84 / 37.15 7.77 1.73	97.59 97.67 97.82 97.70 / 91.33 76.10 67.27	99.88 100.00 99.99 99.88 / 93.38 77.51 68.61	88.40 98.59 92.22 71.29 / 44.08 24.74 9.45	95.90 97.17 94.99 89.67 / 82.49 75.82 65.10	99.68 100.00 99.93 99.56 / 96.06 90.95 82.74
all	Yara	99.82 100.00 100.00 100.00 / 99.99 99.56 96.27	100.00 100.00 100.00 100.00 / 100.00 99.94 99.71	100.00 100.00 100.00 100.00 / 100.00 99.94 99.83	99.83 100.00 100.00 100.00 / 99.98 99.72 97.17	99.99 100.00 100.00 100.00 / 99.96 99.88 99.43	99.99 100.00 100.00 100.00 / 99.97 99.89 99.54
all	Gem	95.27 100.00 99.04 94.93 / 70.97 48.00 34.47	99.75 100.00 99.06 99.61 / 99.64 99.20 97.88	100.00 100.00 100.00 99.99 / 99.98 99.71 98.39	95.18 100.00 98.13 92.41 / 80.50 67.39 47.99	99.65 100.00 98.32 98.97 / 99.03 99.02 93.72	99.95 100.00 99.98 99.97 / 99.82 99.76 95.10
all	Hobbes 2	99.91 100.00 100.00 100.00 / 100.00 99.89 98.04	100.00 100.00 100.00 100.00 / 99.99 100.00 99.91	100.00 100.00 100.00 100.00 / 100.00 100.00 100.00	–	–	–
all	RazerS 3	100.00 100.00 100.00 100.00 / 100.00 100.00 100.00	100.00 100.00 100.00 100.00 / 100.00 100.00 100.00	100.00 100.00 100.00 100.00 / 100.00 100.00 100.00	100.00 100.00 100.00 100.00 / 100.00 100.00 100.00	100.00 100.00 100.00 100.00 / 100.00 100.00 100.00	100.00 100.00 100.00 100.00 / 100.00 100.00 100.00

base pairs per hour (Gbp/h). The Illumina HiSeq 2500 in a six days run produces[1] up to 800 Gbp as 2 × 100 bp paired-end reads. Under this measure the maximum throughput of the Illumina HiSeq 2500 is 5.56 Gbp/h.

For an accurate throughput estimation, best-mappers processed 10 M Illumina 2 × 100 bp reads, while for all-mappers it was sufficient to process 1 M reads. For a fair comparison, best-mappers produced a SAM file as expected by de-facto standard best-mapping pipelines, while all-mappers output a file in native format (SAM for Yara and Hobbes 2, custom for Gem and RazerS 3).

[1] According to the specifications at `http://res.illumina.com/documents/products/datasheets/datasheet_hiseq2500.pdf` for high output run mode with dual flow cell.

Table 7.3: *Read mapping throughput on the human whole genome. All tools run using 8 threads on a desktop computer equipped with an Intel Core i7-4770K CPU. The left panel shows the results of mapping* 2×100 *bp Illumina HiSeq 2000 reads as single-end; the right panel shows the results of mapping the same reads as paired-end. The maximum throughput of an Illumina HiSeq 2500 is* 5.56 *Gbp/h.*

		ERR161544 100 bp		ERR161544 2x100 bp	
	tool	throughput [Gbp/h]	memory [Mb]	throughput [Gbp/h]	memory [Mb]
best	Yara	12.59	5072	11.39	5093
	Gem	8.19	4438	13.35	4430
	Bowtie 2	6.60	3326	6.69	3370
	BWA	4.36	4579	3.77	4759
all	Yara	1.27	5549	1.34	6210
	Gem	0.84	5356	1.64	4733
	Hobbes 2	–	–	–	–
	RazerS 3	0.10	19065	0.17	9139

Appendix A

Read mappers parameterization

A.1 Masai evaluation

In the following, I give the exact parameterization of each read mapper considered in the evaluation of section **??**.

Masai Version 0.5 was used. In order to use Masai as an all-mapper, I passed the argument `–all`, otherwise the argument `–any-best` is used by default. I set the maximal edit distance using the parameter `-e`. I configured the seed length with the parameter `–seed-length`; on *E. coli*, *D. melanogaster* and *C. elegans* I chose a seed length of 16, while on *H. sapiens* I chose a seed length of 33. I selected the SAM output format with `-os` and enabled CIGAR output with `-oc`.

Bowtie 2 Version 2.0.0-beta6 was used. I used the parameter `–end-to-end` to enforce semi-global read alignments. For the Rabema experiment I used the parameter `-k 100`.

BWA Version 0.6.1-r104 was used. For the Rabema experiment I passed the parameter `-N` to `aln` and `-n 100` to `samse`.

Soap 2 Version 2.1 was used.

RazerS 3 Version 3.1 was used. I mapped with indels using the pigeonhole filter (default) and set the error rate through the parameter `-i`, e.g. `-i 95` to map within an error rate of 5 %. I selected the native or SAM output format with `-of 0` or `-of 4`.

Hobbes Version 1.3 was used. I built the index using the recommended q-gram length 11. Since I focus on edit distance, I used the 16 bit bit-vector version. I enabled indels with `–indels` and set maximal edit distance using the parameter `-v`. For resource measurement I used the output without CIGAR, for analyzing the results I enabled CIGAR output using `–cigar`.

mrFAST Version 2.1.0.6 was used. I set maximal edit distance using the parameter `-e`.

SHRiMP 2 Version 2.2.2 was used.

A.2 Yara evaluation

In the following, I give the exact parameterization of each read mapper considered in the evaluation of section 7.2. Below, `MIN` and `MAX` are placeholders for minimal and maximal insert size, while `INS` is the mean insert size and `ERR` its allowed deviation, i.e. `INS = (MIN + MAX) / 2`, `ERR = (MAX - MIN) / 2`.

Yara Version 1.0 was used. To perform all-mapping, I passed the argument `–all`; by default, the tool runs as a best-mapper. I set the error rate using the parameter `-e`. In paired-end mode, the parameters used were `–library-length INS –library-error ERR`. The number of threads was set with the parameter `-t`.

GEM Version 1.376 was used. I set the error rate using the parameters `-m` and `-e`, then I disabled adaptive mapping using the parameter `–quality-format ignore`. In best-mapping, to analyze only the best stratum, I passed the argument `-s 0`; in all-mapping, to analyze all strata, I passed `-d all -D all -s all –max-big-indel-length 0`. In single-end mode, I passed the parameter `–expect-single-end-reads`; in paired-end mode, I passed `–paired-end-alignment`, along with `–min-insert-size MIN –max-insert-size MAX`, and `–map-both-ends` to select the workflow mapping both reads independently. The number of threads was selected using the parameter `-t`.

Bowtie 2 Version 2.2.1 was used. I used the parameter `–end-to-end` to enforce semi-global read alignments. In paired-end mode, I used the parameters `–minins MIN –maxins MAX`. The number of threads was selected using the parameter `-p`.

BWA Version 0.7.7-r441 was used. I used the parameter `-t` to select the number of threads in the `aln` step; the `sampe` and `samse` steps were performed using one thread since BWA does not offer any parallelization here.

Hobbes 2 Version 2.1 was used. I built the index using the recommended q-gram length 11. I enabled edit distance with `–indels` and set the distance threshold using the parameter `-v`. In paired-end mode, I used the parameters `–pe –min MIN –max MAX`. Multi-threading was enabled using `-p`.

RazerS 3 Version 3.2 was used. I set the error rate through the parameter `-i`, e.g. `-i 95` to map within an error rate of 5 %. I passed the option `-rr 100` to set the recognition rate to 100 % and `-m 1000000` to output all mapping locations per read. In paired-end mode, the parameters used were `–library-length INS –library-error ERR`. The number of threads was set with the `-tc` parameter.

Appendix

B Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Enrico Siragusa
October 26, 2014

BIBLIOGRAPHY

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, **2**(1), pages 53–86.

Ahmadi, A., Behm, A., Honnalli, N., Li, C., Weng, L., and Xie, X. (2012). Hobbes: optimized gram-based methods for efficient read alignment. *Nucleic Acids Res.*, **40**(6), page e41.

Alkan, C., Kidd, J. M., Marques-Bonet, T., Aksay, G., Antonacci, F., Hormozdiari, F., Kitzman, J. O., Baker, C., Malig, M., Mutlu, O., Sahinalp, S. C., Gibbs, R. A., and Eichler, E. E. (2009). Personalized copy number and segmental duplication maps using next-generation sequencing. *Nat. Genet.*, **41**(10), pages 1061–1067.

Apostolico, A. (1985). The myriad virtues of subword trees. In *Combinatorial algorithms on words*, pages 85–96. Springer.

Baeza-Yates, R. A. and Perleberg, C. H. (1992). Fast and practical approximate string matching. In *Combinatorial Pattern Matching*, pages 185–192. Springer.

Bailey, J. A., Yavor, A. M., Massa, H. F., Trask, B. J., and Eichler, E. E. (2001). Segmental duplications: organization and impact within the current human genome project assembly. *Genome research*, **11**(6), pages 1005–1017.

Bauer, M. J., Cox, A. J., and Rosone, G. (2013). Lightweight algorithms for constructing and inverting the bwt of string collections. *Theoretical Computer Science*, **483**, pages 134–148.

Belazzougui, D., Cunial, F., Kärkkäinen, J., and Mäkinen, V. (2013). Versatile succinct representations of the bidirectional burrows-wheeler transform. In *Algorithms–ESA 2013*, pages 133–144. Springer.

Burkhardt, S. and Kärkkäinen, J. (2001). Better filtering with gapped q-grams. In *Proc. of the 12th Annual Symposium on Combinatorial Pattern Matching*, CPM '01, pages 73–85. Springer.

Burkhardt, S., Crauser, A., Ferragina, P., Lenhof, H.-P., Rivals, E., and Vingron, M. (1999). q-gram based database searching using a suffix array (QUASAR). In *Proc. of the 3rd Annual International Conference on Research in Computational Molecular Biology*, RECOMB '99, pages 77–83. ACM Press.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm.

Chinwalla, A. T., Cook, L. L., Delehaunty, K. D., Fewell, G. A., Fulton, L. A., Fulton, R. S., Graves, T. A., Hillier, L. W., Mardis, E. R., McPherson, J. D., *et al.* (2002). Initial sequencing and comparative analysis of the mouse genome. *Nature*, **420**(6915), pages 520–562.

Chvatal, V. (1979). A greedy heuristic for the set-covering problem. *Mathematics of operations research*, **4**(3), pages 233–235.

Consortium, I. H. G. S. (2001). Initial sequencing and analysis of the human genome. *Nature*, **409**(6822), pages 860–921.

Crochemore, M., Grossi, R., Kärkkäinen, J., and Landau, G. M. (2013). A constant-space comparison-based algorithm for computing the burrows–wheeler transform. In *Combinatorial Pattern Matching*, pages 74–82. Springer.

David, M., Dzamba, M., Lister, D., Ilie, L., and Brudno, M. (2011). SHRiMP2: sensitive yet practical short read mapping. *Bioinformatics*, **27**(7), pages 1011–1012.

Dehal, P. and Boore, J. L. (2005). Two rounds of whole genome duplication in the ancestral vertebrate. *PLoS biology*, **3**(10), page e314.

Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. (2008). Better external memory suffix array construction. *J. Exp. Algorithmics*, **12**, pages 3.4:1–3.4:24.

DePristo, M. A., Banks, E., Poplin, R., Garimella, K. V., Maguire, J. R., Hartl, C., Philippakis, A. A., del Angel, G., Rivas, M. A., Hanna, M., *et al.* (2011). A framework for variation discovery and genotyping using next-generation dna sequencing data. *Nature genetics*, **43**(5), pages 491–498.

Derrien, T., Estellé, J., Marco Sola, S., Knowles, D. G., Raineri, E., Guigó, R., and Ribeca, P. (2012). Fast computation and applications of genome mappability. *PLoS ONE*, **7**(1), page e30377.

Döring, A., Weese, D., Rausch, T., and Reinert, K. (2008). SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, **9**, page 11.

Elias, P. (1975). Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, **21**(2), pages 194–203.

Ewing, B. and Green, P. (1998). Base-calling of automated sequencer traces using phred. ii. error probabilities. *Genome research*, **8**(3), pages 186–194.

Ewing, B., Hillier, L., Wendl, M. C., and Green, P. (1998). Base-calling of automated sequencer traces using phred. i. accuracy assessment. *Genome research*, **8**(3), pages 175–185.

Faro, S. and Lecroq, T. (2013). The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys (CSUR)*, **45**(2), page 13.

Farrar, M. (2007). Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, **23**(2), pages 156–161.

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, P. and Manzini, G. (2001). An experimental study of an opportunistic index. In *SODA*, pages 269–278.

Fonseca, N. A., Rung, J., Brazma, A., and Marioni, J. C. (2012). Tools for mapping high-throughput sequencing data. *Bioinformatics*, **28**(24), pages 3169–3177.

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

Gallant, J., Maier, D., and Astorer, J. (1980). On finding minimal length superstrings. *Journal of Computer and System Sciences*, **20**(1), pages 50–58.

Giegerich, R., Kurtz, S., and Stoye, J. (1999). Efficient implementation of lazy suffix trees. In *Algorithm Engineering*, pages 30–42. Springer.

Giegerich, R., Kurtz, S., and Stoye, J. (2003). Efficient implementation of lazy suffix trees. *Softw., Pract. Exper.*, pages 1035–1049.

Grossi, R., Gupta, A., and Vitter, J. S. (2003). High-order entropy-compressed text indexes. In *Proc. of the 14th annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: Computer science and computational biology*. Cambridge University Press, New York, NY, USA.

Hach, F., Hormozdiari, F., Alkan, C., Hormozdiari, F., Birol, I., Eichler, E. E., and Sahinalp, S. C. (2010). mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nat. Methods*, **7**(8), pages 576–577.

Hamming, R. W. (1950). Error detecting and error correcting codes. *Syst. Tech. J.*, **29**, pages 147–160.

Holtgrewe, M. (2010). Mason – a read simulator for second generation sequencing data. Technical Report TR-B-10-06, Institut für Mathematik und Informatik, Freie Universität Berlin.

Holtgrewe, M., Emde, A.-K., Weese, D., and Reinert, K. (2011). A novel and well-defined benchmarking method for second generation read mapping. *BMC Bioinformatics*, **12**, page 210.

Intel (2011). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

Jacobson, G. (1989). Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE.

Jokinen, P. and Ukkonen, E. (1991). Two algorithms for approxmate string matching in static texts. In *Mathematical Foundations of Computer Science 1991*, pages 240–248. Springer.

Kärkkäinen, J. and Na, J. (2007). Faster filters for approximate string matching. In *Workshop on Algorithm Engineering and Experiments (ALENEX07)*.

Kärkkäinen, J. and Sanders, P. (2003). Simple linear work suffix array construction. *ICALP*, pages 943–955.

Karp, R. M., Luby, M., and Madras, N. (1989). Monte-carlo approximation algorithms for enumeration problems. *Journal of algorithms*, **10**(3), pages 429–448.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM*, pages 181–192.

Kehr, B., Weese, D., and Reinert, K. (2011). Stellar: fast and exact local alignments. *BMC Bioinf.*, **12**(Suppl 9), page S15.

Kim, J., Li, C., and Xie, X. (2014). Improving read mapping using additional prefix grams. *BMC bioinformatics*, **15**(1), page 42.

Knuth, D. (1973). *The Art of Computer Programming. Volume 3, Addision-Wesley*.

Kucherov, G., Noé, L., and Roytberg, M. (2005). Multiseed lossless filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, **2**(1), pages 51–61.

Kurtz, S. (1999). Reducing the space requirement of suffix trees. *Software-Practice and Experience*, **29**(13), pages 1149–71.

Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with Bowtie 2. *Nat. Methods*, **9**(4), pages 357–359.

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

Lee, H. and Schatz, M. C. (2012). Genomic dark matter: the reliability of short read mapping illustrated by the genome mappability score. *Bioinformatics*, **28**(16), pages 2097–2105.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics – Doklady*, **10**, pages 707–710.

Li, H. (2012). Exploring single-sample snp and indel calling with whole-genome de novo assembly. *Bioinformatics*, **28**(14), pages 1838–1844.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, **25**(14), pages 1754–1760.

Li, H. and Homer, N. (2010). A survey of sequence alignment algorithms for next-generation sequencing. *Brief Bioinform.*, **11**(5), pages 473–483.

Li, H., Ruan, J., and Durbin, R. (2008). Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, **18**(11), pages 1851–1858.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., and 1000 Genome Project Data Processing Subgroup (2009a). The sequence alignment/map format and SAMtools. *Bioinformatics*, **25**(16), pages 2078–2079.

Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., and Wang, J. (2009b). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, **25**(15), pages 1966–1967.

Maier, D. and Storer, J. A. (1977). A note on the complexity of the superstring problem. *Computer Science Laboratory, Report*, (233).

Mäkinen, V., Välimäki, N., Laaksonen, A., and Katainen, R. (2010). Unified view of backward backtracking in short read mapping. In T. Elomaa, H. Mannila, and P. Orponen, editors, *Algorithms and Applications*, volume 6060 of *Lecture Notes in Computer Science*, pages 182–195. Springer Berlin Heidelberg.

Manber, U. and Myers, G. (1990). Suffix arrays: a new method for on-line string searches. In *SODA*, pages 319–327.

Marco-Sola, S., Sammeth, M., Guigó, R., and Ribeca, P. (2012). The gem mapper: fast, accurate and versatile alignment by filtration. *Nature methods*, **9**(12), pages 1185–1188.

Meyne, J., Baker, R. J., Hobart, H. H., Hsu, T., Ryder, O. A., Ward, O. G., Wiley, J. E., Wurster-Hill, D. H., Yates, T. L., and Moyzis, R. K. (1990). Distribution of non-telomeric sites of the (ttaggg) n telomeric sequence in vertebrate chromosomes. *Chromosoma*, **99**(1), pages 3–10.

Morrison, D. R. (1968). Patricia – practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, **15**(4), pages 514–534.

Mortazavi, A., Williams, B., McCue, K., Schaeffer, L., and Wold, B. (2008). Mapping and quantifying mammalian transcriptomes by RNA-seq. *Nat. Methods*, **5**(7), pages 621–628.

Mural, R., Adams, M., Myers, E., Smith, H., Miklos, G., Wides, R., Halpern, A., Li, P., Sutton, G., Nadeau, J., Salzberg, S., Holt, R., Kodira, C., Lu, F., Chen, L., Deng, Z., Evangelista, C., Gan, W., Heiman, T., Li, J., Li, Z., Merkulov, G., Milshina, N., Naik, A., Qi, R., Shue, B., Wang, A., Wang, J., Wang, X., Yan, X., Ye, J., Yooseph, S., Zhao, Q., Zheng, L., Zhu, S., Biddick, K., Bolanos, R., Delcher, A., Dew, I., Fasulo, D., Flanigan, M., Huson, D., Kravitz, S., Miller, J., Mobarry, C., Reinert, K., Remington, K., Zhang, Q., Zheng, X., Nusskern, D., Lai, Z., Lei, Y., Zhong, W., Yao, A., Guan, P., Ji, R., Gu, Z., Wang, Z., Zhong, F., Xiao, C., Chiang, C., Yandell, M., Wortman, J., Amanatides, P., Hladun, S., Pratts, E., Johnson, J., Dodson, K., Woodford, K., Evans, C., Gropman, B., Rusch, D., Venter, E., Wang, M., Smith, T., Houck, J., Tompkins, D., Haynes, C., Jacob, D., Chin, S., Allen, D., Dahlke, C., Sanders, R., Li, K., Liu, X., Levitsky, A., Majoros, W., Chen, Q., Xia, A., Lopez, J., Donnelly, M., Newman, M., Glodek, A., Kraft, C., Nodell, M., Ali, F., An, H., Baldwin-Pitts, D., Beeson, K., Cai, S., Carnes, M., Carver, A., Caulk, P., Center, A., Chen, Y., Cheng, M., Coyne, M., Crowder, M., Danaher, S., Davenport, L., Desilets, R., Dietz, S., Doup, L., Dullaghan, P., Ferriera, S., Fosler, C., Gire, H., Gluecksmann, A., Gocayne, J., Gray, J., Hart, B., Haynes, J., Hoover, J., Howland, T., Ibegwam, C., Jalali, M., Johns, D., Kline, L., Ma, D., MacCawley, S., Magoon, A., Mann, F., May, D., McIntosh, T., Mehta, S., Moy, L., Moy, M., Murphy, B., Murphy, S., Nelson, K., Nuri, Z., Parker, K., Prudhomme, A., Puri, V., Qureshi, H., Raley, J., Reardon, M., Regier, M., Rogers, Y., Romblad, D., Schutz, J., Scott, J., Scott, R., Sitter, C., Smallwood, M., Sprague, A., Stewart, E., Strong, R., Suh, E., Sylvester, K., Thomas, R., Tint, N., Tsonis, C., Wang, G., Williams, M., Williams, S., Windsor, S., Wolfe, K., Wu, M., Zaveri, J., Chaturvedi, K., Gabrielian, A., Ke, Z., Sun, J., Subramanian, G., Venter, J., Pfannkoch, C., Barnstead, M., and Stephenson, L. (2002). A comparison of whole-genome shotgun-derived mouse chromosome 16 and the human genome. *Science*, **296**(5573), pages 1661–71.

Myers, E. W. (1994). A sublinear algorithm for approximate keyword searching. *Algorithmica*, **12**(4-5), pages 345–374.

Myers, E. W. (2005). The fragment assembly string graph. *Bioinformatics*, **21**(suppl 2), pages ii79–ii85.

Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., Kravitz, S. A., Mobarry, C. M., Reinert, K. H., Remington, K. A., Anson, E. L., Bolanos, R. A., Chou, H. H., Jordan, C. M., Halpern, A. L., Lonardi, S., Beasley, E. M., Brandon, R. C., Chen, L., Dunn, P. J., Lai, Z., Liang, Y., Nusskern, D. R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G. M., Adams, M. D., and Venter, J. C. (2000). A whole-genome assembly of Drosophila. *Science*, **287**, pages 2196–2204.

Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, **46**(3), pages 395–415.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM Comput. Surv.*, **33**(1), pages 31–88.

Navarro, G. and Baeza-Yates, R. A. (2000). A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, **1**(1), pages 205–239.

Navarro, G. and Mäkinen, V. (2007). Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, **39**(1), page 2.

Navarro, G., Baeza-Yates, R. A., Sutinen, E., and Tarhio, J. (2001). Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, **24**(4), pages 19–27.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, pages 443–453.

Nicolas, F. and Rivals, E. (2005). Hardness of optimal spaced seed design. In *Combinatorial Pattern Matching*, pages 144–155. Springer.

Pevzner, P. A., Tang, H., and Waterman, M. S. (2001). An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, **98**(17), pages 9748–9753.

Rasmussen, K. R., Stoye, J., and Myers, E. W. (2006). Efficient q-gram filters for finding all ε-matches over a given length. *J. Comput. Biol.*, **13**(2), pages 296–308.

Samonte, R. V. and Eichler, E. E. (2002). Segmental duplications and the evolution of the primate genome. *Nature Reviews Genetics*, **3**(1), pages 65–72.

Sanger, F., Nicklen, S., and Coulson, A. R. (1977). DNA sequencing with chain-terminating inhibitors. *PNAS*, **74**(12), pages 5463–5467.

Schürmann, K.-B. and Stoye, J. (2007). An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, **37**(3), pages 309–329.

Seward, J. (1996). bzip2 and libbzip2. *avaliable at http://www. bzip. org*.

Simola, D. F. and Kim, J. (2011). Sniper: improved snp discovery by multiply mapping deep sequenced reads. *Genome Biol*, **12**(6), page R55.

Siragusa, E., Weese, D., and Reinert, K. (2013). Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic Acids Res.*

Smit, A. F. (1996). The origin of interspersed repeats in the human genome. *Current opinion in genetics & development*, **6**(6), pages 743–748.

Smith, T. F. and Waterman, M. S. (1981). Identification of Common Molecular Subsequences. *J. Mol. Biol.*, **147**, pages 195–197.

Sulston, J., Du, Z., Thomas, K., Wilson, R., Hillier, L., Staden, R., Halloran, N., Green, P., Thierry-Mieg, J., Qiu, L., *et al.* (1992). The c. elegans genome sequencing project: a beginning. *Nature*, **356**(6364), pages 37–41.

Treangen, T. J. and Salzberg, S. L. (2011). Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, **13**(1), pages 36–46.

Turner, J. S. (1989). Approximation algorithms for the shortest common superstring problem. *Information and computation*, **83**(1), pages 1–20.

Ukkonen, E. (1993). Approximate string-matching over suffix trees. In *CPM*, pages 228–242.

Vazirani, V. V. (2001). *Approximation algorithms*. springer.

Venter, J. C., Adams, M. D., Myers, E. W., Li, P. W., Mural, R. J., Sutton, G. G., Smith, H. O., Yandell, M., Evans, C. A., Holt, R. A., Gocayne, J. D., Amanatides, P., Ballew, R. M., Huson, D. H., Wortman, J. R., Zhang, Q., Kodira, C. D., Zheng, X. H., Chen, L., Skupski, M., Subramanian, G., Thomas, P. D., Zhang, J., Gabor Miklos, G. L., Nelson, C., Broder, S., Clark, A. G., Nadeau, J., McKusick, V. A., Zinder, N., Levine, A. J., Roberts, R. J., Simon, M., Slayman, C., Hunkapiller, M., Bolanos, R., Delcher, A., Dew, I., Fasulo, D., Flanigan, M., Florea, L., Halpern, A., Hannenhalli, S., Kravitz, S., Levy, S., Mobarry, C., Reinert, K., Remington, K., Abu-Threideh, J., Beasley, E., Biddick, K., Bonazzi, V., Brandon, R., Cargill, M., Chandramouliswaran, I., Charlab, R., Chaturvedi, K., Deng, Z., Di Francesco, V., Dunn, P., Eilbeck, K., Evangelista, C., Gabrielian, A. E., Gan, W., Ge, W., Gong, F., Gu, Z., Guan, P., Heiman, T. J., Higgins, M. E., Ji, R. R., Ke, Z., Ketchum, K. A., Lai, Z., Lei, Y., Li, Z., Li, J., Liang, Y., Lin, X., Lu, F., Merkulov, G. V., Milshina, N., Moore, H. M., Naik, A. K., Narayan, V. A., Neelam, B., Nusskern, D., Rusch, D. B., Salzberg, S., Shao, W., Shue, B., Sun, J., Wang, Z., Wang, A., Wang, X., Wang, J., Wei, M., Wides, R., Xiao, C., Yan, C., Yao, A., Ye, J., Zhan, M., Zhang, W., Zhang, H., Zhao, Q., Zheng, L., Zhong, F., Zhong, W., Zhu, S., Zhao, S., Gilbert, D., Baumhueter, S., Spier, G., Carter, C., Cravchik, A., Woodage, T., Ali, F., An, H., Awe, A., Baldwin, D., Baden, H., Barnstead, M., Barrow, I., Beeson, K., Busam, D., Carver, A., Center, A., Cheng, M. L., Curry, L., Danaher, S., Davenport, L., Desilets, R., Dietz, S., Dodson, K., Doup, L., Ferriera, S., Garg, N., Gluecksmann, A., Hart, B., Haynes, J., Haynes, C., Heiner, C., Hladun, S., Hostin, D., Houck, J., Howland, T., Ibegwam, C., Johnson, J., Kalush, F., Kline, L., Koduru, S., Love, A., Mann, F., May, D., McCawley, S., McIntosh, T., McMullen, I., Moy, M., Moy, L., Murphy, B., Nelson, K., Pfannkoch, C., Pratts, E., Puri, V., Qureshi, H., Reardon, M., Rodriguez, R., Rogers, Y. H., Romblad, D., Ruhfel, B., Scott, R., Sitter, C., Smallwood, M., Stewart, E., Strong, R., Suh, E., Thomas, R., Tint, N. N., Tse, S., Vech, C., Wang, G., Wetter, J., Williams, S., Williams, M., Windsor, S., Winn-Deen, E., Wolfe, K., Zaveri, J., Zaveri, K., Abril, J. F., Guigó, R., Campbell, M. J., Sjolander, K. V., Karlak, B., Kejariwal, A., Mi, H., Lazareva, B., Hatton, T., Narechania, A., Diemer, K., Muruganujan, A., Guo, N., Sato, S., Bafna, V., Istrail, S., Lippert, R., Schwartz, R., Walenz, B., Yooseph, S., Allen, D., Basu, A., Baxendale, J., Blick, L., Caminha, M., Carnes-Stine, J., Caulk, P., Chiang, Y. H., Coyne, M., Dahlke, C., Mays, A., Dombroski, M., Donnelly, M., Ely, D., Esparham, S., Fosler, C., Gire, H., Glanowski, S., Glasser, K., Glodek, A., Gorokhov, M., Graham, K., Gropman, B., Harris, M., Heil, J., Henderson, S., Hoover, J., Jennings, D., Jordan, C., Jordan, J., Kasha, J., Kagan, L., Kraft, C., Levitsky, A., Lewis, M., Liu, X., Lopez, J., Ma, D., Majoros, W., McDaniel, J., Murphy, S., Newman, M., Nguyen, T., Nguyen, N., Nodell, M., Pan, S., Peck, J., Peterson, M., Rowe, W., Sanders, R., Scott, J., Simpson, M., Smith, T., Sprague, A., Stockwell, T., Turner, R., Venter, E., Wang, M., Wen, M., Wu, D.,

Wu, M., Xia, A., Zandieh, A., and Zhu, X. (2001). The sequence of the human genome. *Science*, **291**, pages 1304–1351.

Wang, Z., Weber, J. L., Zhong, G., and Tanksley, S. (1994). Survey of plant short tandem dna repeats. *Theoretical and applied genetics*, **88**(1), pages 1–6.

Weese, D. (2013). *Indices and Applications in High-Throughput Sequencing*. Ph.D. thesis, Freie Universität Berlin.

Weese, D., Emde, A.-K., Rausch, T., Döring, A., and Reinert, K. (2009). RazerS–fast read mapping with sensitivity control. *Genome Res.*, **19**(9), pages 1646–1654.

Weese, D., Holtgrewe, M., and Reinert, K. (2012). RazerS 3: Faster, fully sensitive read mapping. *Bioinformatics*. 10.1093/bioinformatics/bts505.

Weese, D., Schulz, M. H., Holtgrewe, M., and Richard, H. (2013). Fiona: a versatile and automatic strategy for read error correction. *to appear*.

Weiner, P. (1973). Linear pattern matching algorithms. In *SWAT (FOCS)*, pages 1–11. IEEE.

Wilkes, M. V. (1995). The memory wall and the cmos end-point. *ACM SIGARCH Computer Architecture News*, **23**(4), pages 4–6.

Wolfe, K. H. and Shields, D. C. (1997). Molecular evidence for an ancient duplication of the entire yeast genome. *Nature*, **387**(6634), pages 708–712.

Wooster, R., Cleton-Jansen, A.-M., Collins, N., Mangion, J., Cornelis, R., Cooper, C., Gusterson, B., Ponder, B., Von Deimling, A., Wiestler, O., *et al.* (1994). Instability of short tandem repeats (microsatellites) in human cancers. *Nature genetics*, **6**(2), pages 152–156.

LIST OF FIGURES

LIST OF TABLES

LIST OF AlgorithmS