Efficient algorithms and data structures for indexing DNA sequence data

Kamil Salikhov

UNIVERSITÉ PARIS-EST

Thèse en vue de l'obtention du titre de
Docteur de l'Université Paris-Est

Spécialité : Informatique
École doctorale : MSTIC

Efficient algorithms and data structures for indexing DNA sequence data

Kamil Salikhov

Soutenue le 17 novembre 2017

Jury:

Co-directeur	Gregory Kucherov, Directeur de recherche	LIGM Université Paris-Est, France
Co-directeur	Nikolay Vereshchagin, Professor	Moscow State University, Russia
Rapporteur	Pierre Peterlongo, Chargé de recherche	INRIA/Irisa Rennes, France
Rapporteur	Alexander Kulikov, Senior research fellow	Steklov Institute of Mathematics, Saint Petersburg, Russia
Examinateur	Stéphane Vialette, Directeur de recherche	LIGM Université Paris-Est, France
Examinateur	Rayan Chikhi, Chargé de recherche	CRIStAL Université Lille, France
Examinateur	Mireille Régnier, Directrice de recherche	Ecole Polytechnique, Palaiseau, France

Acknowledgements

First, I would like to thank my PhD supervisor Gregory Kucherov, who introduced me to the amazing world of bioinformatics, always tried to be involved regardless of any geographic distance, whose expertise, creativity and support were very important for me during this research. I also want to thank my second advisor Nikolay Vereschagin, whose guidance, patience and help were invaluable during last four years.

I address a special thank you to Maxim Babenko, my first scientific advisor, who opened the world of Computer Science for me, and whose support helped me to make first steps in this world.

I would like to thank all the members of my committee, namely Pierre Peterlongo, Alexander Kulikov, Stéphane Vialette, Rayan Chikhi, and Mireille Régnier for their time and feedback.

I am incredibly thankful to my family: dad Marat, mom Rezeda, and brother Ayaz, and to my girlfriend Aliya, for their love and boundless support.

I want to express gratitude to everybody I worked with during my research: Karel Brinda, Simone Pignotti, Gustavo Sacomoto and Dekel Tsur. It was a big pleasure for me to work with you, and I learned so much from all our discussions!

I am very thankful to Sylvie Cach and Corinne Palescandolo, whose help with administrative work was priceless during all four years of my study in Université Paris-Est.

Finally, I would like to acknowledge a support of the co-tutelle PhD fellowship grant of the French government and the grant for co-tutelle PhD students of Université Paris-Est.

Abstract

Efficient algorithms and data structures for indexing DNA sequence data

Amounts of data generated by Next Generation Sequencing technologies increase exponentially in recent years. Storing, processing and transferring this data become more and more challenging tasks. To be able to cope with them, data scientists should develop more and more efficient approaches and techniques.

In this thesis we present efficient data structures and algorithmic methods for the problems of approximate string matching, genome assembly, read compression and taxonomy based metagenomic classification.

Approximate string matching is an extensively studied problem with countless number of published papers, both theoretical and practical. In bioinformatics, read mapping problem can be regarded as approximate string matching. Here we study string matching strategies based on bidirectional indices. We define a framework, called search schemes, to work with search strategies of this type, then provide a probabilistic measure for the efficiency of search schemes, prove several combinatorial properties of efficient search schemes and provide experimental computations supporting the superiority of our strategies.

Genome assembly is one of the basic problems of bioinformatics. Here we present Cascading Bloom filter data structure, that improves standard Bloom filter and can be applied to several problems like genome assembly. We provide theoretical and experimental results proving properties of Cascading Bloom filter. We also show how Cascading Bloom filter can be used for solving another important problem of read compression.

Another problem studied in this thesis is metagenomic classification. We present a BWT-based approach that improves the BWT-index for quick and memory-efficient k-mer search. We mainly focus on data structures that improve speed and memory usage of classical BWT-index for our application.

Résumé

Algorithmes et structures de données efficaces pour l'indexation de séquences d'ADN

Les volumes des données générées par les technologies de séquençage haut débit augmentent exponentiellement ce dernier temps. Le stockage, le traitement et le transfert deviennent des défis de plus en plus sérieux. Pour les affronter, les scientifiques doivent élaborer des approches et des algorithmes de plus en plus efficaces.

Dans cette thèse, nous présentons des structures de données efficaces et des algorithmes pour des problèmes de recherche approchée de chaînes de caractères, d'assemblage du génome, de compression de séquences d'ADN et de classification métagénomique de lectures d'ADN.

Le problème de recherche approchée a été bien étudié, avec un grand nombre de travaux publiés. Dans le domaine de bioinformatique, le problème d'alignement de séquences peut être considéré comme un problème de recherche approchée de chaînes de caractères. Dans notre travail, nous étudions une stratégie de recherche basée sur une structure d'indexation dite bidirectionnelle. D'abord, nous définissons un formalisme des schémas de recherche pour travailler avec les stratégies de recherche de ce type, ensuite nous fixons une mesure probabiliste de l'efficacité de schémas de recherche et démontrons quelques propriétés combinatoires de schémas de recherche efficaces. Finalement, nous présentons des calculs expérimentaux qui valident la supériorité de nos stratégies. L'assemblage du génome est un des problèmes clefs en bioinformatique.

Dans cette thèse, nous présentons une structure de données — filtre de Bloom en Cascade — qui améliore le filtre de Bloom standard et peut être utilisé pour la résolution de certains problèmes, y compris pour l'assemblage du génome. Nous démontrons ensuite des résultats analytiques et expérimentaux sur les propriétés du filtre de Bloom en Cascade. Nous présentons également comment le filtre de Bloom en Cascade peut être appliqué au problème de compression de séquences d'ADN.

Un autre problème que nous étudions dans cette thèse est la classification métagénomique de lectures d'ADN. Nous présentons une approche basée sur la transformée de Burrows-Wheeler pour la recherche efficace et rapide de k-mers (mots de longueur k). Cette étude est centrée sur les structures des données qui améliorent la vitesse et la consommation de mémoire par rapport à l'index classique de Burrows- Wheeler, dans le cadre de notre application.

Publications, posters, presentations

Papers

i) Kamil Salikhov, and Gustavo Sacomoto, and Gregory Kucherov. "Using cascading Bloom filters to improve the memory usage for de Brujin graphs." In: *Algorithms in Bioinformatics - 13th International Workshop, WABI 2013, Sophia Antipolis, France, September 2-4, 2013. Proceedings* (2013), pp. 364-376.
DOI:10.1007/978-3-642-40453-5_28

ii) Kamil Salikhov, and Gustavo Sacomoto, and Gregory Kucherov. "Using cascading Bloom filters to improve the memory usage for de Brujin graphs." In: *Algorithms for Molecular Biology* 9.2 (2014), pp. 2.
DOI:10.1186/1748-7188-9-2

iii) Gregory Kucherov, and Kamil Salikhov, and Dekel Tsur. "Approximate string matching using a bidirectional index." In: *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings* (2014), pp. 222-231.
DOI:10.1007/978-3-319-07566-2_23

iv) Gregory Kucherov, and Kamil Salikhov, and Dekel Tsur. "Approximate string matching using a bidirectional index." In: *Theoretical Computer Science* 638 (2016), pp. 145-158.
DOI:10.1016/j.tcs.2015.10.043

v) Kamil Salikhov. "Improved compression of DNA sequencing data with Cascading Bloom filters." Accepted to: *International Journal Foundations of Computer Science* (2017).

Posters

i) K. Břinda, K. Salikhov, S. Pignoti, and G. Kucherov. "Prophyle: a phylogeny-based metagenomic classifier using Burrows-Wheeler Transform." *Second Workshop on Challenges in Microbiome Data Analysis*, Boston (USA), February 16-17, 2017.

ii) K. Břinda, K. Salikhov, S. Pignoti, and G. Kucherov. "Prophyle: a phylogeny-based metagenomic classifier using Burrows-Wheeler Transform." *HitSeq session of ISMB/ECCB 2017,* Prague (Czech Republic), July 24-25, 2017.

Presentations

i) Kamil Salikhov, and Gustavo Sacomoto, and Gregory Kucherov. "Using cascading Bloom filters to improve the memory usage for de Brujin graphs." *Seminar at MSU*, Moscow (Russia), February, 2014.

ii) Kamil Salikhov, and Gustavo Sacomoto, and Gregory Kucherov. "Using cascading Bloom filters to improve the memory usage for de Brujin graphs." *Seminar at LIGM/UPEM*, Paris (France), May 20, 2014.

iii) Gregory Kucherov, and Kamil Salikhov, and Dekel Tsur. "Approximate string matching using a bidirectional index." *Combinatorial Pattern Matching - 25th Annual Symposium, CPM*, Moscow (Russia), June 17, 2014.

iv) Gregory Kucherov, and Kamil Salikhov, and Dekel Tsur. "Approximate string matching using a bidirectional index." *Workshop SeqBio 2014*, Montpellier (France), November 5, 2014.

v) K. Břinda, K. Salikhov, S. Pignoti, and G. Kucherov. "ProPhyle – a memory efficient BWT-based metagenomic classifier." *Workshop DSB 2017*, Amsterdam (Netherlands), February 22, 2017.

Papers outside the scope of the thesis

i) Maxim A. Babenko, and Kamil Salikhov, and Stepan Artamonov. "An Improved Algorithm for Packing T-Paths in Inner Eulerian Networks." In: *Computing and Combinatorics - 18th Annual International Conference, COCOON 2012, Sydney, Australia, August 20-22, 2012. Proceedings*, pp. 109-120.
DOI:10.1007/978-3-642-32241-9_10

Contents

Part I

Introduction

Contents - Part I

Chapter 1

Motivation and overview

Advances in DNA sequencing opened many new directions in genomic research. Genomic data helps to cure genetic and infectious diseases and to understand cancer medicine and evolutionary biology. Genomics actively penetrates in our everyday life – for example, recently a DNA test officially proved that Salvador Dali is not father of a Spanish woman pretended to be his daughter[1].

The whole story began in 1953 with the famous discovery of DNA double helix structure [4] and first DNA fragments sequenced in 1970s. But the unprecedented progress in genomics was associated with the uprising of Next-Generation Sequencing in the early 21st century, which led to an urgent necessity to develop new approaches for biological data storage, processing and analysis. According to Moore's law, the number of transistors in a dense integrated circuit doubles approximately every two years. In other words, its computational capacity doubles approximately every 24 months. On the other hand, in 2012 only about 1000 human genomes were sequenced, and this number tends to one million in 2017. This implicitly proves that sequenced genomic data amounts grow faster (double approximately every 7 months according to historical data, and will double every 12 month according to Illumina predictions, see more details in [5]) than a processors' computational power. For computer scientists this means that increased amounts of data can not be processed using parallel versions of existing algorithms, but new, faster and more memory efficient algorithms should be designed. Sometimes even linear-time algorithms are not fast enough and should be replaced by sublinear ones. Compression of data and compressed data structures are also playing more and more important role in genomic data storage, transfer and analysis.

Let us formulate and give a short introduction into several of the fundamental tasks in processing DNA sequencing data – genome assembly, read alignment and metagenomic classification – which we will study in more details in next chapters.

1.1 Some bioinformatic problems

1.1.1 Read alignment

One of the first challenges presented by sequencing technologies is the so-called *read alignment* problem. Next generation sequencers like Illumina or SOLID produce short sequences consisting of tens or hundred nucleotides, while sequencers like Oxford Nanopore generate sequences of length of thousands or even tens of thousands of nucleotides. These fragments extracted from a longer DNA molecule are called *reads*.

[1] http://www.bbc.com/news/world-europe-41180146

Generally, the goal of read alignment or *read mapping* is, for a given read and a *reference genome*, to find position(s) in the reference where the read matches in the best way. Mapping should be tolerant to different types of errors arising either introduced by sequencers or appeared due to differences between the sequenced specie's genome and the reference genome. There are different variations of this problem, as many types of differences between reference subsequence and read can be allowed. More about read alignment problem and its solutions can be found in Chapter 4.

1.1.2 Genome assembly

Currently, no sequencing technology can decode the entire genome sequence. All sequencers cut chromosomes into overlapping fragments and read these short sequences. Thus the natural problem of bioinformatics is to combine all *reads* together in order to reconstruct the initial genomic sequence. This process is called *genome assembly.*

Genome assembly is usually a necessary step in bioinformatics pipelines, providing a useful resource for various genomic approaches. As it was explained in Section 1.1.1, genomes are used as reference sequences for read alignment. Another possible application is evolutionary biology, when different genomes are compared to reveal relations between different species. Genomes of different organisms of the same species (for example, human population) can be examined in order to detect differences between them. In Chapter 6 we provide more information about popular approaches for solving the genome assembly problem.

1.1.3 Metagenomic classification

Recent advances in Next-Generation Sequencing technologies have allowed a breakthrough in *metagenomics* which aims to study genetic material obtained from environmental samples. This led to boosting the metagenomic analysis and to developing more and more sophisticated methods and tools. The first samples studied were seawater [6, 7], human gut [8], soil [9]. Nowadays, samples of many other different origins are investigated, including extreme environments like acid water, areas of volcanism, etc. Research is also focused on bacterial and viral populations, whereas the first one is better studied because of extreme diversity of viral genomes. The Human Microbiome project [10] has the goal to study a human microbiome and its correlation with human health.

Metagenomics via whole-genome sequencing deals with read sets of total size of hundreds of millions and billions of nucleotides. These sequences are obtained from many different species, and while some of them have a previously sequenced reference genome, others may not have a reference genome even for a close relative. Thus, the goal of metagenomic classification is to determine for every sequence the corresponding reference genome (or a set of genomes, which belong to one family, for example), or to say that it belongs to a previously not sequenced species.

1.2 Brief overview

In this thesis, we present several data structures and algorithms that can be applied to read alignment, genome assembly and metagenomic classification problems.

Although these problems seems to be quite far one from another, they actually have many things in common, as we will show in next chapters of this work. Solutions of these and many other problems are often based on similar ideas and data structures. For example, many of early techniques to solve read alignment, genome assembly and

metagenomic classification problems were based on building a hash table for k-mers (words of length k). Nowadays, for all these problems solutions exploiting memory efficiency of BWT transform become more and more popular. In other words, these problems are evolving in conjunction.

First, in Chapter 3 we describe basic data structures that are usually used as basic bricks in more complicated algorithms and methods in bioinformatics. Some of them, namely Bloom filter, hash table and BWT index are applied in our methods in next chapters.

In Chapter 4 we cover the problem of *approximate string matching* and describe main techniques used in read alignment algorithms. Next, in Chapter 5 we study strategies of approximate string matching based on bidirectional text indexes, extending and generalizing ideas of [11]. We provide both a theoretical analysis and computations, proving superiority of our methods.

In Chapter 6 we describe existing methods for solving the genome assembly problem. Then, in Chapter 7 we introduce an efficient data structure, called Cascading Bloom filter, which can be applied to fast and very memory efficient genome assembly. We also show how this data structure can help to solve the *read compression* problem in Chapter 8.

We provide more details about different solutions of metagenomic classification problem in Chapter 9. Next, in Chapter 10 we address the problem of classification when a taxonomic tree of studied species is given in advance, and we need to assign reads to nodes of this tree. We introduce several data structures and techniques that make our solution time and memory efficient.

We end with providing conclusions on the presented work in Part V.

Chapter 2

Biological context

2.1 Biological background

DNA (Deoxyribonucleic acid) is a macromolecule forming the genome of living organisms. It lays in the foundation of life as it stores the genetic information necessary for development, functioning and reproduction. DNA molecules are found in the cytoplasm of prokaryotic organism cells and in the cell nucleus of eukaryotic organisms. In multicellular eukaryotes, each nucleated cell stores a full copy of the genome, as this information is used for cell reproduction.

DNA molecules are combined into *chromosomes*. A chromosome is composed of a very long DNA molecule and consists of a linear array of *genes*, which are DNA regions encoding genetic information. In addition, DNA has other regions with structural purposes, or the ones involved in regulating the expression of genes. At a given locus (a specific location inside a chromosome) different variants of a gene, called alleles, can appear [12].

Most DNA molecules consist of two strands and form the famous double helix model, first identified by James Watson, Francis Crick and Maurice Wilkins in 1953 [4], whose model-building efforts were guided by X-ray diffraction data acquired in 1952 by Rosalind Franklin and Raymond Gosling [13]. For their discovery, Watson, Crick and Wilkins were awarded the 1962 Nobel Prize for Physiology or Medicine, "for their discoveries concerning the molecular structure of nucleic acids and its significance for information transfer in living material".

Each DNA strand consists of repeating units called nucleotides. Every nucleotide includes one of four possible nucleobases (or just bases) – adenine (A), cytosine (C), guanine (G) and thymine (T). Adjacent nucleotides are connected through the use of sugars and phosphates. Two strands are connected by hydrogen links between corresponding nucleobases – more precisely, adenine is connected to thymine, and cytosine is linked to guanine (and vice versa). Such paired bases are called complementary.

DNA replication plays a key role in the process of cell division in all organisms. For unicellular organisms, the DNA replication process is used for reproduction, and for multicellular ones, for the organism's growth. During DNA replication, DNA strands split up into two separate strands, and then, on the basis of each of them, a new two-strand DNA is synthesised. Thus, every new DNA contains one strand from the initial cell and one newly synthesised *complementary* strand.

2.2 Mutations

DNA can be damaged by many sorts of mutagens, such as radiation, viruses, some chemicals etc., which change the DNA sequence. In the same time, mutations can appear accidentally during the process of DNA replication. DNA collects mutations over time, which are then inherited. Some mutations can cause diseases like cancer, whereas others can contribute to evolution by enabling the generation of new functions in order to adopt to environmental changes.

Mutations can be divided into several types. Some of them affect only few nucleobases, while the others change long DNA segments, i.e. duplicate, remove, move or reverse them.

In this work we mainly deal with the first type of mutations – point mutations. Point mutations are either substitutions of one nucleotide in the DNA sequence for another type of nucleotide, or insertions and deletions of one or more nucleobases. Point mutations can have different effects, according to the place where they occur and the nature of the substitution, insertion or deletion.

2.3 Sequencing methods

To be able to perform computational analysis of the information enclosed in the genome, the DNA sequence (i.e. the exact order of nucleotides in the DNA molecule) needs to be determined. This is achieved via DNA sequencing methods.

First polynucleotide (77-nth yeast alanine tRNA) was sequenced twelve years after the publication of the Watson and Crick double-helix DNA structure in 1953 [4]. The RNA sequencing methods used at that time were two-dimensional chromatography and spectrophotometric procedures. Only a few base pairs per year could be sequenced using this methods. The first complete genome sequence – 3569-nucleotide-long bacteriophage MS2 RNA sequence – was also a result of sequencing using these methods.

The next methods were developed by Maxam and Gilbert [14] and Sanger [15]. The first of them is conducted by chemical cleaving specific bases of terminally labeled DNA fragments and separating them by elecrophoresis. The second method is based on the selective incorporation of chain-terminating dideoxynucleotides by DNA polymerase during DNA replication. Sanger's method became much more popular due to its simplicity, reduced use of toxic chemicals and lower amounts of radioactivity.

The first automated sequencer, based on Sanger's method, was developed by Applied Biosystem Instruments. Later, they were improved by adding computers to store and analyse collected data. This method was the most widely used until about 2005, when *Next Generation Sequencing* methods replaced it.

Next generation sequencing methods were developed due to a high demand for low-cost high-throughput technologies, that can parallelize the sequencing process. These methods sequence millions to billions of nucleotides in parallel and can produce gigabytes of data per day.

Second-generation sequencers, such as Illumina or SOLiD, implement a so-called *shotgun sequencing* approach. They do not read every DNA molecule from the beginning to the end as, for example, Sanger's method does. Instead, DNA molecules are cut into many small fragments, and then each fragment is scanned from one end or from both, generating *single-end* or *pair-end* reads (when a fragment is read from both ends). Usually, sequencers also provide information about the quality of bases in the read. Thus, for computations, a *read* is just a sequence of A, C, G, T characters with (possibly) some quality (or measure of confidence) associated with each position. A typical reads' lengths are from a few tens to

several hundreds of nucleobases (though, third-generation sequencers like PacBio can generate much longer – thousands and tens of thousands long – reads). From a computational point of view, using shotgun sequencing to retrieve reads means that we need to put them together in the correct order to obtain a whole genome.

A detailed history of first- and next- (which also can be divided into second- and third-) generation sequencing technologies can be found in many articles [16, 17, 18, 19, 20, 21, 22, 23, 24, 25].

From 1965 to 2005, the cost of humane genome sequencing (which is about 3 billions base pairs long) was reduced from millions to less than thousand of US dollars. For example, as part of the Human Genome Project, the J. C. Venter genome [26] took almost 15 years to sequence at a cost of more than 1 million dollars using the Sanger method, whereas the J. D. Watson (1962 Nobel Prize winner) genome was sequenced by NGS using the 454 Genome Sequencer FLX with about the same 7.5x coverage within 2 months and for approximately a hundredth of the price [27].

All sequencers produce sequencing errors and biases that should be corrected. The major sequencing errors are largely related to high-frequency indel polymorphisms, homopolymeric regions, GC- and AT-rich regions, replicate bias, and substitution errors [28, 29, 30].

Chapter 3

Basic data structures

In bioinformatic applications, various algorithms are designed to obtain fast and memory efficient solutions. Basic approach in many of them is to construct a data structure, usually called *index*, that supports fast queries and occupies a reasonable amount of memory. Most of them are built upon such basic and rather simple data structures like hash tables, suffix trees, suffix arrays and others.

In this chapter we will consider some of them, namely a hash table, Bloom filter, suffix tree and array, BWT index and a bidirectional version of BWT index. Even though these structures have different design and support different types of operations, in many fields of bionformatics they are used in different solutions for the same problem. Below we describe the main properties of these data structures, discuss their advantages and drawbacks and outline some problems in bioinformatics where they can be applied.

3.1 Hash table

A **hash table** can be regarded as an associative array, which stores $(key, value)$ pairs and permits to perform three operations:

- $insert(key, value)$
- $find(key)$
- $remove(key)$

A hash table uses a *hash function* h to map *keys* to integers from 0 to $m-1$, which represents the indexes of array A of size m. If for a key k $h(k) = i$, then we try to put element (k, v) in slot i of the array. It can not be guaranteed that different keys will not map to the same index. This situation introduces a *collision*: a situation, when two keys are mapped to the same position in the array.

In practice, there are two basic ways of dealing with collisions. The first method is to store a list of elements in every slot of array A, and if we need to put an element with key k in slot i, we just append it to the end of the corresponding list. This method is called *separate chaining*. Another strategy is called *open adressing*, and after failing to insert a key k in the array (because the corresponding position is not empty), the algorithm searches for another slot to insert this key.

Operations in a hash table are rather fast, as in average all of them work in constant time (although, it depends on the parameters of the hash table and the hash function). However, one downside is that one operation can take $O(n)$ time in the worst case, where

n is the number of inserted elements. In practice, hash tables usually work faster than other data structures storing $(key, value)$ pairs and allowing for a search by key.

Choosing an appopriate hash function is very important to obtain a good performance of a hash table. If a good hash function is chosen, and n elements are inserted into array of size k, then the average search of one element works in $O(1+\frac{n}{k})$ time if a separate chaining strategy is chosen for collisions resolution. Here $\frac{n}{k}$ is called a *load factor*, and it shows how many elements on average are inserted in the same slot in the array.

Although hash tables outperform many other data structures in query time performance, they usually require much more memory.

If the set of elements is known in advance, there is another way to resolve collisions. We can find a *perfect* hash function that maps elements to indices in array without collisions. There are different algorithms for perfect hash function construction, among them [31, 32, 33].

In computer science, a hash table's usage area includes, but is not limited to storing associative arrays, sets and collections in databases. In bioinformatic applications, they are often used to store sets of $k-mers$ (strings of length k) and values associated with them (see Chapters 4 and 6 for several examples).

3.2 Bloom filter

A *Bloom filter* [34] is a space-efficient data structure for representing a given subset of elements $T \subseteq U$, with support for efficient membership queries with a one-sided error. That is, if a query for an element $x \in U$ returns *no* then $x \notin T$, but if it returns *yes* then x may or not belong to T, i.e. with small probability $x \notin T$ (false positive). A Bloom filter consists of a bitmap (array of bits) B with size m and a set of p distinct hash functions $\{h_1, \ldots, h_p\}$, where $h_i : U \mapsto \{0, \ldots, m-1\}$. Initially, all bits of B are set to 0. An insertion of an element $x \in T$ is done by setting the elements of B with indices $h_1(x), \ldots, h_p(x)$ to 1, i.e. $B[h_i(x)] = 1$ for all $i \in [1, p]$. The membership queries are done symmetrically, returning *yes* if all $B[h_i(x)]$ are equal to 1 and *no* otherwise. As shown in [35], when considering hash functions that yield equally likely positions in the bit array, and for large enough array size m and number of inserted elements n, the false positive rate $\mathcal{F}$ is

$$\mathcal{F} \approx (1 - e^{-pn/m})^p = (1 - e^{-p/r})^p \tag{3.1}$$

where $r = m/n$ is the number of bits (of the bitmap B) per element. It is not hard to see that this expression is minimized when $p = r \ln 2$, giving a false positive rate of

$$\mathcal{F} \approx (1 - e^{-p/r})^p = (1/2)^p \approx 0.6185^r. \tag{3.2}$$

Assuming that hash functions are computed in constant time and the number of hash functions is limited, then insertion of an element and search for an element can be completed in $O(1)$ time.

A major advantage of a Bloom filter over other data structures for representing sets, such as search trees and hash tables, is that its size is independent of the size of inserted elements. Thus, even if an element's bit representation is rather long, the memory usage of a Bloom filter is only a few (usually 8-16) bits per inserted element. The main drawback of a Bloom filter comes from its probabilistic design, although a false positive rate is usually chosen to be rather small. It is also worth noting that the basic version of a Bloom filter does not allow deletions. However, Bloom filter has become a fruitful area for research,

and there has appeared many variations of them. Among these variations there is a Bloom filter that allows deletions, a *counting* Bloom filter and a Bloom filter that allows storing values along with keys.

In computer science, Bloom filters are widely used in web applications for storing caches. For example, a Bloom filter was implemented in Google's Chrome browser for storing malicious sites list. If there are, for example, one million suspicious sites, and the average length of their names is 25, then $25MB$ is needed to store them explicitly. Bloom filter with a 1% error rate will occupy slightly more than $1MB$ for the same dataset.

Bloom filters are often used in bioinformatic aplications, usually, but not always, for storing and counting k-mers (one of applications is discussed in Chapter 7).

3.3 Suffix tree

Suppose that we are given a string T of length n (a sequence of characters $T[0..n-1]$). Let us start with several definitions. We call any subsequence $T[i..j], 0 \leq i \leq j < n$ a *substring* of T. Substrings of form $T[i..n-1], 0 \leq i < n$ and $T[0..i], 0 \leq i < n$ are called *suffixes* and *prefixes* respectively.

Trie is a tree-like data structure for storing a dynamic set of strings S. Every edge of the trie is labelled by a character, the root of the trie corresponds to an empty string. Every string of S can be obtained while traversing trie, beginning from the root and concatenating labels on the edges. A *suffix trie* of a string T is a trie, constructed on the set of suffixes of T. A *suffix tree* is a suffix trie where two consecutive edges are merged into one (with labels concatenation) if their common node has only one ingoing and one outgoing edge.

Definition 3.3.1. (another definition of *suffix tree*) The suffix tree of a string T is a tree such that

- The tree has exactly n leaves numbered from 0 to $n-1$ if T ends with special \$ character.
- Every internal node, possibly except for the root, has at least two children.
- Each edge is labeled with a non-empty substring of S.
- No two edges starting out of a node can have string-labels beginning with the same character.
- The string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix $T[i..n-1]$, for i from 0 to $n-1$.

The concept of a suffix tree was introduced in 1973 by Weiner in [36]. He was also the first who suggested a linear-time construction algorithm. A naive construction algorithm works in $O(n^2)$ time. McCreight in [37] created a more lightweight linear time algorithm that was improved in a classic work [38] of Ukkonen. He also formulated a linear-time online construction algorithm in [38].

It is worth noting that algorithms mentioned above are linear in case of small (i.e., constant-size) alphabets. In the worst case, for the alphabet of size $O(n)$, their working time increases to $O(n \log n)$.

In recent years, several algorithms have been suggested for fast (although still linear-time, but independent of the alphabet's size) and space-efficient construction of suffix trees. Most of them are based on the ideas proposed by Farach in [39].

Basically, the suffix tree was designed for fast exact string matching (i.e., finding all exact occurrences of a pattern in a given text). Then suffix trees were applied to other problems in computer science, for example, to find the longest common substring of two (or more) strings.

Search for a pattern P of length k in a text T using a suffix tree requires $O(k+p)$ time, where p is the number of occurrences of P in T. If the number of strings to search in is more than one, a *generalized suffix tree* can be constructed. Its construction time is also linear in the sum of lengths of all strings.

Although suffix trees require a linear amount of memory, the space used for one character is rather big ($13-40$ bytes per character depending on the alphabet). This is the biggest drawback of suffix trees. More recent data structures, such as suffix arrays, can solve problems that allow suffix tree-based solutions with the same efficiency, but they require much less memory.

In bioinformatics, suffix trees are basically used in applications to read alignment (for example, [40]). Another possible application is whole genome alignment (see, for example, work of Delcher et al. [41]).

3.4 Suffix array

Again, we are given a string T of length n.

Definition 3.4.1. The suffix array SA of T is an array of starting positions of suffixes of T, sorted in lexicographical order.

In other words, A is an array where $SA[i]$ contains the starting position of i-th lexicographically smallest suffix of T, thus the property $T[SA[i-1], n-1] < T[SA[i], n-1]$ holds for any $0 < i \leq n-1$.

Suffix arrays were introduced by Manber and Myers in [42] and independently in [43] by Gonnet et al. as a memory efficient replacement of suffix trees. Whereas, being an array of n integers from 0 to $n-1$, suffix array requires $O(n \log n)$ memory, integers stored in a suffix arrays usually fit into 4 (or 8) bytes, and an overall suffix array fits into $4n$ (or $8n$) bytes, which is less than memory needed for suffix trees.

In conjunction with a suffix array, an *LCP (longest common prefix) array* is often used.

Definition 3.4.2. For $0 < i \leq n-1$, $LCP[i]$ is the length of the longest common prefix of $T[SA[i-1]..n-1]$ and $T[SA[i]..n-1]$.

A naive suffix array construction algorithm (based on simple suffix sorting) works in $O(n^2 \log n)$ time ($O(n \log n)$ comparisons for a sorting algorithm, where one comparison can be made in $O(n)$ time). A slightly better approach is to use a *radix sort*, it reduces the time complexity to $O(n^2)$. It is worth noting that a suffix array can be constructed in $O(n)$ time using a *depth-first search* if the suffix tree is given in advance.

However, there are algorithms for a direct linear time suffix array construction, that do not need the suffix tree in advance. The first one was suggested by Kärkkäinen and Sanders ([44]) in 2003. Nowadays, one of the fastest suffix array construction applications is the algorithm SA-IS from [45] by Nong, Zhang and Chan, implemented by Yuta Mori. It works in linear time, requires only 5 bytes of memory per character (minimum possible value) and is fast in practice. It is noteworthy that its implementation in C language is no longer than 100 lines of code. A good survey of state-of-the-art suffix array construction algorithms is given in [46].

Using only a suffix array, a pattern P of length m can be found in a text T of length n in $O(m \log n)$ time. This complexity can be improved, if LCP array is used, to $O(m + \log n)$. In [47] it was shown that $O(m + \log n)$ bound can be improved even further to $O(m)$, the same complexity as it is for suffix trees. The authors of [47] showed that any problem, that has a solution using suffix trees, can be solved using suffix arrays preserving the same time and space complexity. Nowadays, being much more lightweight structure than suffix trees and being able to solve same problems, suffix array replace suffix trees in applications in almost all domains of computer science.

One of the most famous problems, which can be solved by suffix array, is *longest repeated substring* problem, it can be solved in $O(n)$ time with use of suffix array and LCP.

In bioinformatics, suffix arrays are usually used in read alignment, prefix-suffix overlaps computation and sequence clustering (see, for example, [48, 49, 50]). Even though a suffix array requires much less space than a suffix tree, for human genome, for example, it occupies $12GB$ of memory (without LCP array), whereas genome itself (about 3 millions base pairs) can fit into less than 1 GB of memory. Such a big difference follows from the fact that the suffix array occupies $O(n \log n)$ memory, while the string on alphabet Σ requires only $O(n \log |\Sigma|)$ memory. Recently, new data structures like *compressed suffix array* and BWT index were designed to further reduce the memory usage.

3.5 BWT index

Both suffix array and suffix tree, along with some other structures out of scope of current manuscript, allow to perform *forward* (or *left-to-right*) search. That is, given a pattern, we scan it from the leftmost character to the rightmost one, finding occurrences of a longer and longer prefix of the pattern in the text.

BWT index is a compressed full-text string index, based on *Burrows-Wheeler transform* [51] (further BWT). It was suggested by Ferragina and Manzini in 2000 ([52]). Unlike these structures, a *BWT index* allows to perform *backward* search of the pattern in the text.

Definition 3.5.1. Burrows-Wheeler transform (also called block-sorting compression) of the string T is a permutation BWT of characters of T, such that

$$BWT[i] = \begin{cases} T[SA[i] - 1] & \text{if } SA[i] \neq 0 \\ T[n-1] & \text{otherwise} \end{cases}$$

BWT was proposed in [53] by Burrows and Wheeler in 1994.

Initially, BWT was invented for data compression, and it is used, for example, as part of *bzip2* data archiver. One of the main advantages of BWT is that it rearranges characters of the initial string such that it has runs of equal characters.

But what is more important and, maybe, surprising is the fact that Burrows-Wheeler transform if reversible. In other words, this means that given a BWT of an unknown string, we can restore this string.

Being just a permutation of a string, BWT can be stored in the same amount of memory as the initial string. Moreover, it can be computed in time linear in the size of the string, and the reverse transformation can be also performed in linear time.

In addition to BWT string, the BWT index stores some auxiliary data structures. The first of them is just an array C that for every character c from alphabet Σ stores how many characters are lexicographically smaller than c in T. For example, if $T = abacaba, \Sigma = \{a, b, c\}$, then $C[a] = 0, C[b] = 4, C[c] = 6$. C array is also called a $Count$ function.

The next part of a BWT index is a structure R that supports *rank* operations on the BWT string, that is for any $i \geq 0$ and $\sigma \in \Sigma$, $rank(\sigma, i) = number\ of\ such\ positions\ 0 \leq j \leq i\ that\ BWT[j] = \sigma$. In other words, R allows to count the number of occurrences of each character in any prefix of BWT string. This structure can be implemented on basis of *Wavelet tree* [54], for example, and it occupies $O(\frac{n \log\log n}{\log n})$ memory.

Given a BWT index, we can perform a backward search (i.e., find the interval in the suffix array which corresponds to all occurrences of the pattern) using the following algorithm:

Algorithm 1 Exact pattern matching using BWT index

Input:
string $T[0..n-1]$
BWT index for T with array C and *rank* function
pattern $P[0..m-1]$

$start = 0$
$end = n - 1$
for $i = m - 1$ to 0 **do**
  if $start > end$ **then**
    break
  end if
  $\sigma = P[i]$
  $start = C[\sigma] + rank(\sigma, start - 1)$
  $end = C[\sigma] + rank(\sigma, end) - 1$
end for
Output:
if $start > end$, then output is empty;
otherwise, all occurrences (and only they) are in interval $(start, end)$.

However, unlike the suffix array or tree, Algorithm 1 does not provide a method for translation from a suffix array position to a position in the text. For these, a sampled suffix array stores suffix array values for selected positions (usually, for every K-th position for some constant K). Using a sampled suffix array, the position in the text can be retrieved using Algorithm 2.

Algorithm 2 Translate SA position x to position in the text

Input:
text T
BWT index of T
SSA (sampled suffix array of T)

$steps = 0$
while (x is not sampled in SSA) **do**
  $c = BWT[x]$ ▷ previous character in the text
  $x = C[c] + rank(c, x)$
  steps = steps + 1
end while
Output:
$SSA[x] + steps$

Together, algorithms 1 and 2 provide a linear-time approach to perform the exact pattern matching. Overall, the BWT index occupies $O(n)$ bits of memory, using much less bits per character than the suffix array and the suffix tree. For example, for bioinformatic applications the memory usage can range from 2 to 4 bits per character of the text.

Being much more memory-efficient, BWT index is slower than suffix trees and suffix arrays in practice.

Besides the exact pattern matching problem, BWT index can also be used for solving the approximate pattern matching problem. In bioinformatics, it is applied in solutions problems like read alignment (see Chapter 4), genome assembly (Chapter 6) and many others. The BWT index has now been used in many practical bioinformatics software programs, e.g. [55, 56, 57].

3.6 Bidirectional BWT index

A suffix tree and a suffix array allow for a forward search, and a BWT index for a backward one. However, none of them allows to extend pattern in both directions. A naive way to make this possible is to store two structures together – for example, using a suffix tree (or suffix array, or BWT index) both for a string and the reverse of the string. Then, to perform a backward search, we should use the BWT index for the initial string, and to perform a forward search, we should perform a backward search of the reversed pattern in the BWT index for the reversed string. However, this method does not permit to change the direction of a pattern extension "on-the-fly", during one search. On the other hand, it requires exactly twice more memory to store data structures.

The possibility to alter the search direction can be provided by an improved ("bidirectional") version of a BWT index, as it was shown in [11, 58, 59, 60].

Let us consider here how BWT index can be made bidirectional. Let us start with a definition of a bidirectional search in BWT index.

For string $S[0..n-1]$, let S^R be a reverse of S, e.g. $S^R = S[n-1][S[n-2]..S[1]S[0]$. Let BWT^R be a BWT for S^R. If (s, e) is a suffix array interval for pattern P and string T, then let (s^R, e^R) denote the suffix array interval for pattern P and string T^R.

Let BWT and BWT^R be the Burrows-Wheeler transform with the support of the additional *Count* and *Rank* operations for strings T and T^R respectively. Then the following lemma holds:

Lemma 1. *Given a pattern P, a character $\sigma \in \Sigma$, and suffix array intervals (s, e) and (s^R, e^R) for P and P^R with respect to T and T^R correspondingly, suffix array intervals (s', e') and (s'^R, e'^R) for patterns $P\sigma$ and $(P\sigma)^R$ with respect to T and T^R correspondingly can be computed in $O(|\Sigma|)$ time, where $|\Sigma|$ is the size of the alphabet.*

This lemma provides an idea how to perform a forward search (in other words, how to extend the pattern to the right) supporting suffix array intervals both for the text and for reversed text.

Proof. First, we show how to compute (s'^R, e'^R). It is obvious that the suffix array interval for P^R with respect to T^R is just (s, e). Then, (s'^R, e'^R) can be computed using a backward search in T^R of pattern $(P\sigma)^R$.

The tricky part is how to compute suffix array interval (s', e') of $P\sigma$. It is obvious that (s', e') is a subinterval of (s, e), as all suffixes in (s', e') should start with P. Then, $(s', e') = (s + x, s + x + y - 1)$, where x is the number of suffixes of type $P\sigma'$ with σ' lexicographically smaller than σ, and y is the number of suffixes of type $P\sigma$. For arbitrary

σ', the number of suffixes of type $P\sigma'$ can be computed as the size of the suffix array interval of $(P\sigma')^R$ with respect to T^R. This means that (s', e') can be computed in $O(\Sigma)$ time. ■

For the backward search, the similar lemma holds:

Lemma 2. *Given a pattern P, a character $\sigma \in \Sigma$, and suffix array intervals (s, e) and (s^R, e^R) for P and P^R with respect to T and T^R correspondingly, suffix array intervals (s', e') and (s'^R, e'^R) for patterns σP and $(\sigma P)^R$ with respect to T and T^R correspondingly can be computed in $O(|\Sigma|)$ time, where $|\Sigma|$ is the size of the alphabet.*

In both lemmas 2 and 1 we support suffix array intervals both for T and T^R. From this, two properties of a bidirectional BWT index follow:

- the search direction can be changed during the search (for example, we can extend the pattern to the right, then to the left, and then to the right again)
- it is enough to store only one sampled suffix array (say, for T) to be able to translate suffix array positions to positions in the text. This means that the memory needed by bidirectional BWT index is less than memory occupied by two copies of BWT index.

Bidirectional search can be successfully applied to approximate pattern matching problem, as it will be shown in Chapter 5.

Part II

Efficient approximate string search

Contents - Part II

Chapter 4

Algorithmic methods for read alignment

4.1 Sequence alignment

To discover differences between a newly sequenced organism, represented by a huge number of short reads, and a previously sequenced genome of the same (or related) species, the reads should be mapped (or *aligned*) to the genome. Short read alignment (we also refer to it as *short read mapping*, which is equivalent) is a common first step of genomic data analysis and plays a critical role in medical, population genetics and many other fields of bioinformatics.

Definition 4.1.1. An *alignment* of two sequences $S = s_1 \dots s_n$ and $T = t_1 \dots t_n$ over the same alphabet Σ consists of two equally sized sequences $S' = s'_1 \dots s'_l$ and $T' = t'_1 \dots t'_l$ obtained by inserting zero or more gaps (represented by symbol $-$) between the characters of S and T respectively with the constraint that $\nexists h \in 1 \dots l : s'_h = t'_h = -$.

Thus, an alignment of two sequences allows us to identify the similarities of these sequences. Such positions i that $S'_i \neq -$ and $T'_i \neq -$ and $S'_i \neq T'_i$ correspond to mismatches, positions i that $S'_i = -$ or $T'_i = -$ correspond to indels, and if $S'_i = T'_i$, then this is a match.

Read alignment is a particular case of general sequence alignment, when one of the sequences is a read and another one is a reference genome.

A *score*, which is in general a function representing the "similarity" of two sequences, is associated with every alignment. The goal of sequence alignment is to find an alignment (or several alignments) which maximize the scoring function, so-called "optimal" alignment(s). If whole sequences are aligned, such an alignment is called *global*. It makes sense if two sequences are supposed to be similar in their entirety. In another type of alignments, *local* alignments, only subsequences of the initial sequences are aligned to identify regions of similarity within long sequences that are often widely divergent overall.

There exist two different types of output, produced by different aligners:

- *best mapping* aims to find the best alignment,
- *all best mappings* is to find all occurrences of the pattern having a score close to the best score.

Finding only one best mappings is computationally a less intensive task than calculating all best matches, although some good alignments could be omitted in this case. Methods, discussed in Chapter 5, are suitable for both cases.

Depending on whether the read has actually been mapped by the algorithm or not, and whether it is present in the genome or not, all reads are divided into four categories:

- **True positives (TP)**: the reads that are present in the reference and are mapped correctly by the algorithm
- **False positives (FP)**: the reads that are mapped by the algorithm by mistake
- **True negatives (TN)**: the reads which are correctly not mapped
- **False negatives (FN)**: the reads which should be mapped, but are not mapped by the algorithm

Alignment algorithms are compared by many statistical parameters, the two main ones, *Sensitivity* and *specificity*, are defined below (see, e.g, [61] for more details).

Definition 4.1.2. *Sensitivity* is the ratio of correctly mapped reads to all reads that should be mapped, i.e. $Sn = \frac{|TP|}{|TP|+|FN|}$.

Definition 4.1.3. *Specificity* is the ratio of correctly unmapped reads to all reads that should not be mapped, i.e. $Sp = \frac{|TN|}{|TN|+|FP|}$.

Most of modern fast read alignment algorithms build auxiliary data structures, called *indices*, for the reference sequence or for the reads set. This permits to eliminate huge parts of the reference where the read can not be found. Such a step is usually called *filtration*. Thus, a general idea of all these methods is first to limit significantly the search space, and then to apply a more time-consuming algorithm only for limited regions.

In Section 4.2 we first examine the read alignment problem from the computer science point of view. After that we overview methods based on dynamic programming in Section 4.3. Then we describe the seed-and-extend strategy in Section 4.4. In Section 4.4.1 we introduce spaced seeds that efficiently replace contiguous seeds in many applications. In Section 4.5 we briefly describe methods utilizing suffix tree-like data structures for sequence alignment. For a more detailed overview of existing aligners we refer to [62, 63, 64].

4.2 Read alignment as string matching

The read alignment problem (that is, to find a read's origin in a genome sequence) can be regarded as a *string matching* problem.

Definition 4.2.1. Given a text T of length n and a pattern P of length m over the same alphabet Σ, the *exact string matching* problem is to find all the occurrences of P in T.

The exact string matching can be viewed as the "ideal" case of read alignment, where the read corresponds to the pattern and the genome corresponds to the text, and we do not allow any mismatches or indels (insertions and deletions) in the alignment.

The naive algorithm which attempts to find an occurrence of the pattern starting from every position in the text runs in $O(nm)$ time in the worst case and is thus impractical for read mapping. First linear-time algorithms were created by Boyer-Moore [65] and Knuth-Morris-Pratt [66]. They work in $O(n+m)$ time, which is still too slow to find an alignment of genomes when n or m can be millions or billions.

Typically, string matching algorithms build auxiliary indices so that all occurrences of pattern can be found without full scan of the text. Usually, it is assumed that texts are static, and we do not need to reconstruct the index during matching.

Suffix trees and suffix arrays, described in Sections 3.4 and 3.3, are classical data structures that allow us to solve the exact string matching problem in $O(m + occ)$ time, where *occ* is the number of occurrences of the pattern in the text. The BWT index, examined in Section 3.5, is a more memory-efficient data structure designed to solve the exact string matching problem.

Exact string matching, being an extreme variant of a read alignment problem, is usually useless in practice, as there are always differences between the reference genome and the genome of the organism that was sequenced due to mutations and sequencing errors. Thus, any algorithm useful in practice should be able to find approximate occurrences of the pattern. This leads to the *approximate* string matching problem. First, in definitions 4.2.2 and 4.2.3 we introduce the concept of *distance* between two strings. The Hamming distance and the edit distance are two of the simplest score functions for alignment of two sequences.

Definition 4.2.2. The *Hamming distance* between two strings S and T of equal length n and over the same alphabet Σ is the number of positions $i : 1 \leq i \leq n$ so that $S_i \neq T_i$.

In other words, a hamming distance is the number of mismatches between two strings of equal length.

Definition 4.2.3. For two strings $S = S_1 \dots S_n$ and $T = T_1 \dots T_m$, the *edit distance* is the minimum length of a series of edit operations, that transforms S into T. Operations allowed are

- substitution: replace a single character in S with another one
- insertion: insert any single character at any place in S
- deletion: remove any single character from S.

The *weighted* edit distance between two strings is defined similarly to the simple edit distance, but instead of the number of operations it counts the sum of the weights of substitutions, deletions and additions.

For a given $k > 0$, text T of length n and pattern P of length m, k-mismatch problem [67, 68, 69] is to find all substrings of T within the Hamming distance k from P. Dynamic programming based approaches solve this problem in $O((n + m)k)$ time. Similarly, k-difference problem [70, 71] is to find all substrings of T within the edit distance k from P.

More general scoring systems for substitutions are usually defined by a *substitution matrix* A, where $A[x, y]$ corresponds to the cost of $x \to y$ character substitution. These matrices are usually symmetrical (see, for example, BLOSUM62 [72] or PAM120[73]). Under some limitations, the approximate string matching under the edit distance with a given substitution matrix is a problem equivalent to the read alignment.

When deletions and insertions are allowed, the deletion or insertion of consecutive characters is called a *gap*. More formally, a gap is a maximal sequence of dashes in the alignment. Usually, gaps are penalized using *affine functions* of the form $a + b\ell$, where ℓ is the length of the gap. For the case of affine score functions, dynamic programming based approaches solve the problem in $O((n + m)k)$ time. However, in case of large genomes it is impossible to spend $O(nm)$ or even $O((n + m)k)$ time to map millions of reads.

4.3 Dynamic programming based methods

The first appearance of biological sequence alignment problem dates back to the 1960s, when first proteins were sequenced. The original goal for aligning two protein sequences was to discover the evolutionary relationships between them. Assuming that they had a common ancestor, matches correspond to the characters from this common ancestor, while the positions where proteins differ represent mutations.

The first algorithm to find an optimal alignment of two sequences was proposed by Needleman and Wunsch in 1970 [74]. It implements the so-called *dynamic programming* paradigm. In general, the Needleman-Wunsch algorithm works as follows: characters of the sequences are put in the 0-th row and in the 0-th column of the n by m matrix, and every cell (i, j) of the matrix contains the optimal alignment of the prefixes $S_1 \dots S_i$ and $T_1 \dots T_j$. The values in the grid are computed from left to right and from top to bottom, and the value in a cell is calculated with the use of already computed values. The working time of this method is $O(nm)$.

The Needleman-Wunsch algorithm allows us to compare two sequences in their entirety, but may fail to find local similarities. Smith and Waterman in 1981 [75] adapted this method so that it finds local alignments and also works in $O(nm)$ time.

Both Needleman-Wunsch and Smith-Waterman algorithms work reasonably fast only on small sequences, as they need to perform a full comparison of two sequences. They are too costly when two genomes have to be aligned to one another, or when many short reads are to be aligned against a long genome. However, under the Hamming or edit distance, if the number of errors is limited by some k, then Smith-Waterman algorithm works in $O((n + m)k)$ time. Moreover, there are several techniques of Smith-Waterman algorithm parallelization. Some of them utilizes properties of modern CPUs and uses SIMD (*single instruction - multiple data*) instructions such as SSE2. Another algorithms, such as famous Myers bit-parallel algorithm [76], can speed-up alignment under a simple scoring system.

4.4 Methods based on seed-and-extend strategy

The seed-and-extend approach consists of two steps: first, all exact occurrences of some set of *seeds* are found, and then these occurrences are extended using algorithms like the Smith-Waterman one. A seed is characterized by a *template* which is a sequence of zeros and ones, where 1 corresponds to the position that should be matched, and 0 indicate the position that may or may not match. The number of ones in the seed template is called its *weight*. The seed "hits" the reference at some position if two subsequences extracted from the reference at this position and from the read according to the seed template are equal. For example, if seed template is 101, the reference is $ACAGT$ and the read is $CTGA$, then we can extract subsequence C-G ("-" corresponds to 0 from the seed template) from the read and the same subsequence from the reference, and obtain a "hit". It is assumed that seeds match the reference without errors. If they are comparatively long and only several matching positions are found in the reference, then the second step (extension) can be performed quickly. If the seeding step is also fast, then the overall performance of the algorithm will be good.

The simplest variant of a seed is a seed containing only 1s, and we extract and compare only contiguous k-mers (strings over Σ of length k). Thus, the general seed-and-extend algorithm works in three steps:

1. extract the set of k-mers from the read (all k-mers, or only "informative" ones, may be extracted).

2. these k-mers are queried against a data structure (usually, hash table) that stores all occurrences of all k-mers from the reference genome.

3. perform the extension step for the regions of the reference which contain one or several k-mers from the read.

The first algorithms that followed the seed-and-extend approach were FASTA [77] and BLAST [78]. Interestingly, being one of the first tools implementing seed-and-extend strategy, FASTA package gave rise to the well-known format of the same name. FASTA was followed by BLAST, which became one of the leading tools for sequence comparison. It uses k equal to 11 for DNA data by default, which permits to reduce the search space significantly, while being small enough to fit into memory. The basic BLAST algorithm was improved to be able to find alignments of different types, for example, alignments with gaps. Besides the alignment itself, BLAST provides a very useful feature: it can estimate the probability that each mapping position was found by chance. It was one of the most popular aligners for more than twenty years and has tens of thousands citations nowadays.

Many other alignment algorithms follow the seed-and-extend strategy. Some of them, like SOAP [79, 80], Novoalign[1], Mosaik [81], BFAST [82], index the reference genome. Other approaches, like MAQ [83], RMAP [84], ZOOM [85], SHRiMP [86], build the index for the reads' set. The methods based on hash tables usually provide high sensitivity, but occupy a lot of memory.

Many methods above utilize the *pigeonhole principle*. It states that if there are m containers and there are n items in them, and $n > m$, then at least one container contains at least 2 items. In case of "lossless" (i.e., with a goal to find all approximate occurrences of the pattern) read mapping with up to two errors, it means that we can divide the read into three parts and at least one of them will be error-free. To solve k-mismatch problem, RMAP follows pigeonhole principle explicitly, subdividing the read into $k+1$ parts. Eland, MAQ and SOAP split the read into four parts of the same length in order to make at least two of them error-free in the case of 2-error problem. MAQ applies this strategy only to the first 28 most reliable characters of the read.

Some aligners consider only regions of the reference where several seeds are found. The idea behind this is the following one: if the pattern (read) r matches the substring w of the text (reference) with up to e errors (mismatches or/and gaps), then the pattern and w share at least $|r| + 1 - (k + 1)e$ k-mers. This statement, known as q-gramm lemma, allows us to consider only regions with multiple k-mer hits when solving the e-error matching problem. SHRiMP follows this approach, looking for an alignment only in regions with sufficiently large number of seed matches. Another software, utilizing a "multiple-hit" strategy (and based on spaced seeds, see Section 4.4.1), is YASS [87].

It should be mentioned that seeding using hash tables works efficiently only for short reads and/or for reads with low sequencing error rate. For long reads with a high rate of sequencing errors they become inefficient due to the fact that seeds should be comparatively short, and the extension step is very slow.

Candidate positions found at the seeding step should be extended to an alignment of the read under the given error model. Usually, simpler models allow for faster algorithms. If the score system is the Hamming distance, then two strings can be compared using a linear scan. This strategy is utilized, for example, in MAQ. Under more complicated error models the Smith-Waterman algorithm is used to compute an alignment (see, for example, BFAST). NovoAlign and SHRiMP apply SIMD-vectorized Smith-Waterman algorithm implementations.

[1] http://www.novocraft.com

4.4.1 Spaced seeds

BLAST and many other approaches mentioned above use contiguous seeds to perform the seeding step. Ma et al. in [88] and Burkhardt and Kärkkäinen in [89] introduced *spaced seeds*, that require a nonconsecutive sequence of characters to match. Thus, spaced seeds have templates containing both 1s and 0s.

While for seeding with contiguous (or *solid*) seeds of weight k choosing a big value of k may lead to missing some positions and choosing a small k can produce an excessive number of false positive hits that will not extend to a full alignment, an appropriately chosen spaced seed provides a better chance of hitting true matching positions and produces less false positives. Ma et al. [88] discovered that using spaced seeds instead of consecutive ones leads to a more sensitive alignment algorithm.

PatternHunter [88] was one of the first algorithm that used space seeds to align reads. PatternHunter uses spaced seeds to perform a "lossy" alignment, when some matches could be missed, but the associated error rate is limited. Another program, Eland[2], uses six seed templates to index the reads and to find all occurrences of the read with up to two mismatches. SOAP [79] provides a solution to the same problem by indexing the reference instead of the reads. SeqMap [90] and MAQ [83] apply the same approach to solve the k-mismatch problem using $\binom{2k}{k}$ seeds. RMAP [84] uses only $k+1$ seed templates to solve k-mismatch problem, but the weight of the seeds is comparatively small.

Designing "optimal" sets of seed templates, as well as designing seed templates, is a complicated and important problem, as it directly affects the performance of algorithms. A general framework for automatic spaced seed design was suggested by Kucherov et al. in [91] and implemented in software IEDERA[3]. Lin et al. [85] showed how to construct a minimal number of spaced seeds, given a read length, the sensitivity requirement and memory usage. For example, to solve a 2-mismatch problem for reads of length 32 ZOOM program [85] constructs 5 seed templates of weight 14 while Eland uses 6 templates of weight 16. As the time needed to search using seeds can be considered to be proportional to the number of seed templates and the weight of the seeds, ZOOM has better time asymptotics than Eland if they fit the same memory requirements.

4.5 Methods based on suffix tree-like structures

Some approximate matching algorithms use standard text indexes, such as suffix tree (3.3), suffix array (3.4) or BWT index (3.5). All such algorithms reduce the approximate pattern matching problem to the exact matching problem. They usually implement, explicitly or implicitly, two steps: finding exact matches and then building inexact alignments around exact ones. Finding exact matches is trivial and was discussed in Section 3. Extending exact matches to approximate ones is much more tricky and usually is performed using backtracking. This idea is covered in Chapter 5.

The advantage of using suffix tree-like structures, in comparison to hash tables, is that all occurrences of a pattern correspond to one node in the tree, thus the extension step should be easier. Another feature of such methods is that the choice of the data structure (suffix tree, suffix array, BWT index, etc.) is independent from the approximate matching algorithm itself.

For large datasets occurring in modern applications, such indexes as suffix tree and suffix array are known to take too much memory. Suffix arrays and suffix trees typically

[2] part of Illumina software

[3] http://bioinfo.cristal.univ-lille.fr/yass/iedera.php

require at least 4 or 10 *bytes* per character respectively. The last years saw the development of *succinct* or *compressed full-text indexes* that occupy virtually as much memory as the sequence itself and yet provide very powerful functionalities [92]. The BWT index [52], based on the Burrows-Wheeler Transform [53], may occupy 2–4 *bits* of memory per character for DNA texts. BWT index has now been used in many practical bioinformatics software programs, e.g. [55, 56, 57]. Even if succinct indexes are primarily designed for exact pattern search, using them for approximate matching naturally became an attractive opportunity.

Among published aligners, MUMmer [41] and OASIS [40] are based on a suffix tree, and Vmatch [49] and Segemehl [50] are designed on the basis of suffix array. As it was noted above, the most memory efficient (and most popular) methods are based on BWT index, among them Bowtie [93], BWA-MEM [94], SOAP2 [80], BWT-SW [95] and BWA-SW [96].

Nowadays, one of the most popular DNA sequence aligner is BWA-MEM [94]. Along with BWA-Backtrack [97] and BWA-SW [95], it is based on a very efficient implementation of BWT-index, which is an improvement of an index implemented in BWT-SW [95]. First, for every suffix of a read BWA-MEM finds minimum exact matches (MEMs), used as seeds. Then short seeds are thrown out, too long seeds are shortened and nearby seeds are merged. Finally, the Smith-Waterman algorithm is applied to find the alignment.

Interestingly, succinct indexes can provide even more functionalities than classical ones. In particular, succinct indexes can be made *bidirectional*, i.e. can perform pattern search in both directions [11, 58, 59, 60]. Lam et al. [11] showed how a bidirectional BWT index can be used to efficiently search for strings up to a small number (one or two) errors. The idea is similar to one discussed in 4.4.1: to partition the pattern into $k + 1$ equal parts, where k is the number of errors, and then perform multiple searches on the BWT index, where each search assumes a different distribution of mismatches among the pattern parts. It has been shown experimentally in [11] that this improvement leads to a faster search compared to the best existing read alignment software. Bidirectional search, introduced in [11], was further improved in [1]. Related algorithmic ideas appear also in [58].

In Chapter 5 we extend and generalize the ideas of using a bidirectional BWT index for approximate pattern matching presented in [11].

Chapter 5

Approximate string matching using a bidirectional index

5.1 Overview

In this chapter, we study the approximate string matching problem, both under the Hamming distance and the edit distance (see Chapter 4.2). Let k be the maximum allowed number of mismatches (for the Hamming distance) or errors (for the edit distance). We study strategies of approximate pattern matching that exploit bidirectional text indexes, extending and generalizing ideas of [11] in two main directions. We consider the case of arbitrary k and propose to partition the pattern into more than $k+1$ parts that can be of *unequal* size. To demonstrate the benefit of both ideas, we first introduce a general formal framework for this kind of algorithm, called *search scheme*, that allows us to easily specify them and to reason about them (Section 5.2). Then, in Section 5.3 we perform a probabilistic analysis that provides us with a quantitative measure of performance of a search scheme, and give an efficient algorithm for obtaining the optimal pattern partition for a given scheme. Furthermore, we prove several combinatorial results on the design of efficient search schemes (Section 5.4). Finally, Section 5.5 contains comparative analytical estimations, based on our probabilistic analysis, that demonstrate the superiority of our search strategies for many practical parameter ranges. We further report on large-scale experiments on genomic data supporting this analysis in Section 5.5.2. We end with providing directions for future development in Section 5.6.

5.2 Bidirectional search

In the framework of text indexing, pattern search is usually done by scanning the pattern online and recomputing *index points* referring to the occurrences of the scanned part of the pattern. With classical text indexes, such as suffix trees or suffix arrays, the pattern is scanned left-to-right (*forward search*). However, some compact indexes such as BWT index provide a search algorithm that scans the pattern right-to-left (*backward search*).

Consider now approximate string matching. For ease of presentation, we present most of our ideas for the case of Hamming distance (see 4.2.2), although our algorithms extend to the edit distance (4.2.3) as well. Section 5.3.1 below will specifically deal with the edit distance.

Assume that k letter mismatches are allowed between a pattern P and a substring of length $|P|$ of a text T. Both forward and backward search can be extended to approximate search in a straightforward way, by exploring all possible mismatches along the search,

as long as their number does not exceed k and the current pattern still occurs in the text. For the forward search, for example, the algorithm enumerates all substrings of T with Hamming distance at most k to a *prefix* of P. Starting with the empty string, the enumeration is done by extending the current string with the corresponding letter of P, and with all other letters provided that the number of accumulated mismatches has not yet reached k. For each extension, its positions in T are computed using the index. Note that the set of enumerated strings is closed under prefixes and therefore can be represented by the nodes of a trie. Similar to forward search, *backward search* enumerates all substrings of T with Hamming distance at most k to a *suffix* of P.

Clearly, backward and forward search are symmetric and, once we have an implementation of one, the other can be implemented similarly by constructing the index for the reversed text. However, combining both forward and backward search within one algorithm results in a more efficient search. To illustrate this, consider the case $k = 1$. Partition P into two equal length parts $P = P_1P_2$. The idea is to perform two complementary searches: forward search for occurrences of P with a mismatch in P_2 and backward search for occurrences with a mismatch in P_1. In both searches, branching is performed only after $|P|/2$ characters are matched. Then, the number of strings enumerated by the two searches is much less than the number of strings enumerated by a single standard forward search, even though two searches are performed instead of one.

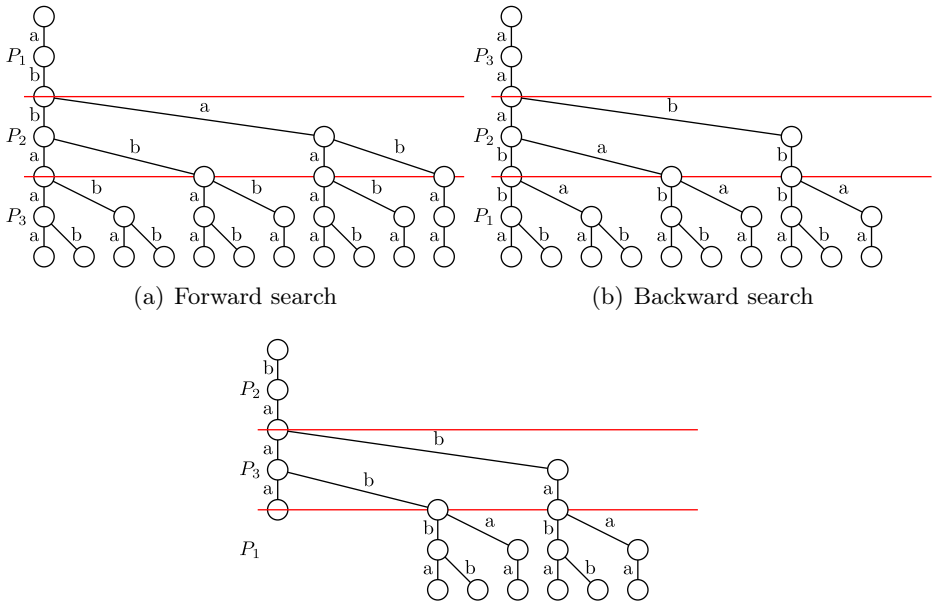
A *bidirectional index* of a text (a bidirectional BWT index was described in Section 3.6) allows one to extend the current string A both left and right, that is, compute the positions of either cA or Ac from the positions of A. Note that a bidirectional index allows forward and backward searches to alternate, which will be crucial for our purposes. Lam et al. [11] showed how the BWT index can be made bidirectional. Other succinct bidirectional indexes were given in [58, 59, 60]. Using a bidirectional index, such as BWT index, forward and backward searches can be performed in time linear in the number of enumerated strings. Therefore, our main goal is to organize the search so that the number of enumerated strings is minimized.

Lam et al. [11] gave a new search algorithm, called *bidirectional search*, that utilizes the bidirectional property of the index. Consider the case $k = 2$, studied in [11]. In this case, the pattern is partitioned into three equal length parts, $P = P_1P_2P_3$. There are now 6 cases to consider according to the placement of mismatches within the parts: 011 (i.e. one mismatch in P_2 and one mismatch in P_3), 101, 110, 002, 020, and 200. The algorithm of Lam et al. [11] performs three searches (illustrated in Figure 5.1):

1. A forward search that allows no mismatches when processing characters of P_1, and 0 to 2 accumulated mismatches when processing characters of P_2 and P_3. This search handles the cases 011, 002, and 020 above.

2. A backward search that allows no mismatches when processing characters of P_3, 0 to 1 accumulated mismatches when processing characters of P_2, and 0 to 2 accumulated mismatches when processing characters of P_1. This search handles the cases 110 and 200 above.

3. The remaining case is 101. This case is handled using a *bidirectional search*. It starts with a forward search on string $P' = P_2P_3$ that allows no mismatches when processing characters of P_2, and 0 to 1 accumulated mismatches when processing the characters of P_3. For each string A of length $|P'|$ enumerated by the forward search whose Hamming distance from P' is exactly 1, a backward search for P_1 is performed by extending A to the left, allowing one additional mismatch. In other words, the search allows 1 to 2 accumulated mismatches when processing the characters of P_1.



(a) Forward search

(b) Backward search

(c) Bidirectional search

Figure 5.1: The tries representing the searches of Lam et al. for binary alphabet $\{\mathrm{a}, \mathrm{b}\}$, search string $P = \mathrm{abbaaa}$, and number of errors 2. Each trie represents one search and assumes that all the enumerated substrings exist in the text T. In an actual search on a specific T, each trie contains of a subset of the nodes, depending on whether the strings of the nodes in the trie appear in T. A vertical edge represents a match, and a diagonal edge represents a mismatch.

We now give a formal definition for the above. Suppose that the pattern P is partitioned into p parts. A *search* is a triplet of strings $S = (\pi, L, U)$ where π is a permutation string of length p over $\{1, \ldots, p\}$, and L, U are strings of length p over $\{0, \ldots, k\}$. The string π indicates the order in which the parts of P are processed, and thus it must satisfy the following *connectivity property*: For every $i > 1$, $\pi(i)$ is either $(\min_{j<i} \pi(j)) - 1$ or $(\max_{j<i} \pi(j)) + 1$. The strings U and L give upper and lower bounds on the number of mismatches: When the j-th part is processed, the number of accumulated mismatches between the active strings and the corresponding substring of P must be between $L[j]$ and $U[j]$. Formally, for a string A over integers, the *weight* of A is $\sum_i A[i]$. A search $S = (\pi, L, U)$ *covers* a string A if $L[i{+}1] \leq \sum_{j=1}^{i} A[j] \leq U[i]$ for all i (assuming $L[p{+}1] = 0$). A *k-mismatch search scheme* $\mathcal{S}$ is a collection of searches such that for every string A of weight k, there is a search in $\mathcal{S}$ that covers A. For example, the 2-mismatch scheme of Lam et al. consists of searches $S_f = (123, 000, 022)$, $S_b = (321, 000, 012)$, and $S_{bd} = (231, 001, 012)$. We denote this scheme by $\mathcal{S}_{\text{LLTWWY}}$.

We introduce two types of improvements over the search scheme of Lam et al.

Uneven partition. In $\mathcal{S}_{\text{LLTWWY}}$, search S_f enumerates more strings than the other two searches, as it allows 2 mismatches on the second processed part of P, while the other two searches allow only one mismatch. If we increase the length of P_1 in the partition of P, the number of strings enumerated by S_f will decrease, while the number of strings enumerated by the two other searches will increase. We show that for some typical parameters of the problem, the decrease in the former number is larger than the increase of the latter number, leading to a more efficient search.

More parts. Another improvement can be achieved using partitions with $k + 2$ or more parts, rather than $k + 1$ parts. We explain in Section 5.3.2 why such partitions can reduce the number of enumerated strings.

5.3 Analysis of search schemes

In this section we show how to estimate the performance of a given search scheme $\mathcal{S}$. Using this technique, we first explain why an uneven partition can lead to a better performance, and then present a dynamic programming algorithm for designing an optimal partition of a pattern.

5.3.1 Estimating the efficiency of a search scheme

To measure the efficiency of a search scheme, we estimate the number of strings enumerated by all the searches of $\mathcal{S}$. We assume that performing single steps of forward, backward, or bidirectional searches takes the same amount of time. It is fairly straightforward to extend the method of this section to the case when these times are not equal. Note that the bidirectional index of Lam et al. [11] reportedly spends slightly more time (order of 10%) on forward search than on backward search.

For the analysis, we assume that characters of T and P are randomly drawn uniformly and independently from the alphabet. We note that it is possible to extend the method of this section to a non-uniform distribution. For more complex distributions, a Monte Carlo simulation can be applied which, however, requires much more time than the method of this section.

Hamming distance

Our approach to the analysis is as follows. Consider a fixed search S, and the trie representing this search (see Figure 5.1). The search enumerates the largest number of strings when the text contains all strings of length m as substrings. In this case, every string that occurs in the trie is enumerated. For other texts, the set of enumerated strings is a subset of the set of strings that occurs in trie. The expected number of strings enumerated by S on random T and P is equal to the sum over all nodes v of the trie of the probability that the corresponding string appears in T. We will first show that this probability depends only on the depth of v (Lemmas 3 and 4 below). Then, we will show how to count the number of nodes in each level of the trie.

Let $p_{n,l,\sigma}$ denote the probability that a random string of length l is a substring of a random string of length n, where the characters of both strings are randomly chosen uniformly and independently from an alphabet of size σ. The following lemma gives an approximation for $p_{n,l,\sigma}$ with a bound on the approximation error.

Lemma 3. $|p_{n,l,\sigma} - (1 - e^{-n/\sigma^l})| \leq \begin{cases} 4nl/\sigma^{2l} & \text{if } l \geq \log_\sigma n \\ 4l/\sigma^l & \text{otherwise} \end{cases}$.

Proof. Let A and B be random strings of length l and n, respectively. Let E_i be the event that A appears in B at position i. The event E_i is independent of the events $\{E_j : j \in \{1, 2, \ldots, n-l+1\} \setminus F_i\}$, where $F_i = \{i-l+1, i-l+2, \ldots, i+l-1\}$. By the Chen-Stein method [98, 99],

$$\left|p_{n,l,\sigma} - (1 - e^{-n/\sigma^l})\right| \leq \frac{1 - e^{-\lambda}}{\lambda} \sum_{i=1}^{n-l+1} \sum_{j \in F_i} (\Pr[E_i] \Pr[E_j] + \Pr[E_i \cap E_j]),$$

where $\lambda = n/\sigma^l$. Clearly, $\Pr[E_i] = \Pr[E_j] = 1/\sigma^l$. It is also easy to verify that $\Pr[E_i \cap E_j] = 1/\sigma^{2l}$. Therefore, $|p_{n,l,\sigma} - (1 - e^{-n/\sigma^l})| \leq ((1 - e^{-\lambda})/\lambda) \cdot 4nl/\sigma^{2l}$. The lemma follows since $(1 - e^{-\lambda})/\lambda \leq \min(1, 1/\lambda)$ for all λ. ∎

The bound in Lemma 3 on the error of the approximation of $p_{n,l,\sigma}$ is large if l is small, say $l < \frac{1}{2} \log_\sigma n$. In this case, we can get a better bound by observing that $p_{n,l,\sigma} \geq p_{n,l_0,\sigma}$, where $l_0 = \frac{3}{4} \log_\sigma n$. Since $p_{n,l_0,\sigma} \geq 1 - e^{-n/\sigma^{l_0}} - 4l_0/\sigma^{l_0}$, we obtain that $|p_{n,l,\sigma} - (1 - e^{-n/\sigma^l})| \leq \max(e^{-n/\sigma^l}, e^{-n/\sigma^{l_0}} + 4l_0/\sigma^{l_0})$.

Let $\#\mathrm{str}(S, X, \sigma, n)$ denote the expected number of strings enumerated when performing a search $S = (\pi, L, U)$ on a random text of length n and random pattern of length m, where X is a partition of the pattern and σ is the alphabet size (note that m is not a parameter for #str since the value of m is implied from X). For a search scheme $\mathcal{S}$, $\#\mathrm{str}(\mathcal{S}, X, \sigma, n) = \sum_{S \in \mathcal{S}} \#\mathrm{str}(S, X, \sigma, n)$.

Fix S, X, σ, and n. Let $\mathcal{A}_l$ be the set of enumerated strings of length l when performing search S on a random pattern of length m, partitioned by X, and a text $\hat{T}$ containing all strings of length at most m as substrings. Let $A_{l,i}$ be the i-th element of $\mathcal{A}_l$ (an order on $\mathcal{A}_l$ will be defined in the proof of the next lemma). Let $nodes_l = |\mathcal{A}_l|$, namely, the number of nodes at depth l in the trie that represents the search S. Let P^* be the string containing the characters of P according to the order they are read by the search. In other words, $P^*[l]$ is the character such that every node at depth $l-1$ of the trie has an edge to a child with label $P^*[l]$.

Lemma 4. *For every l and i, the string $A_{l,i}$ is a random string with uniform distribution.*

Proof. Assume that the alphabet is $\Sigma = \{0, \ldots, \sigma - 1\}$. Consider the trie that represents the search S. We define an order on the children of each node of the trie as follows: Let v be a node in the trie with depth $l - 1$. The label on the edge between v and its leftmost child is $P^*[l]$. If v has more than one child, the labels on the edges to the rest of the children of v, from left to right, are $(P^*[l] + 1) \bmod \sigma, \ldots, (P^*[l] + \sigma - 1) \bmod \sigma$. We now order the set $\mathcal{A}_l$ according to the nodes of depth l in the trie. Namely, let $v_1, \ldots, v_{nodes_l}$ be the nodes of depth l in the trie, from left to right. Then, $A_{l,i}$ is the string that corresponds to v_i. We have that $A_{l,i}[j] = (P^*[j] + c_{i,j} - 1) \bmod \sigma$ for $j = 1, \ldots, l$, where $c_{i,j}$ is the rank of the node of depth j on the path from the root to v_i among its siblings. Now, since each letter of P is randomly chosen uniformly and independently from the alphabet, it follows that each letter of $A_{l,i}$ has uniform distribution and the letters of $A_{l,i}$ are independent. ■

By the linearity of the expectation,

$$\#\mathrm{str}(S, X, \sigma, n) = \sum_{l \geq 1} \sum_{i=1}^{nodes_l} \Pr_{T \in \Sigma^n}[A_{l,i} \text{ is a substring of } T].$$

By Lemma 4 and Lemma 3,

$$\#\mathrm{str}(S, X, \sigma, n) = \sum_{l=1}^{m} nodes_l \cdot p_{n,l,\sigma} \approx \sum_{l=1}^{m} nodes_l (1 - e^{-n/\sigma^l}). \tag{5.1}$$

We note that the bounds on the approximation errors of $p_{n,l,\sigma}$ are small, therefore even when these bounds are multiplied by $nodes_l$ and summed over all l, the resulting bound on the error is small.

In order to compute the values of $nodes_l$, we give some definitions. Let $nodes_{l,d}$ be the number of strings in $\mathcal{A}_l$ of length l with Hamming distance d to the prefix of P^* of length l. For example, consider search $S_{bd} = (231, 001, 012)$ and partition of a pattern of length 6 into 3 parts of length 2, as shown in Figure 5.1(c). Then, $P^* = \mathrm{baaaba}$, $nodes_{5,0} = 0$, $nodes_{5,1} = 2$ (strings baabb and babab), and $nodes_{5,2} = 2$ (strings baaba and babaa).

Let π_X be a string obtained from π by replacing each character $\pi(i)$ of π by a run of $\pi(i)$ of length $X[\pi(i)]$, where $X[j]$ is the length of the j-th part in the partition X. Similarly, L_X is a string obtained from L by replacing each character $L[i]$ by a run of $L[i]$ of length $X[\pi(i)]$, and U_X is defined analogously. In other words, values $L_X[i], U_X[i]$ give lower and upper bounds on the number of allowed mismatches for an enumerated string of length i. For example, for S_{bd} and the partition X defined above, $\pi_X = 223311$, $L_X = 000011$, and $U_X = 001122$.

Values $nodes_l$ are given by the following recurrence.

$$nodes_l = \sum_{d=L_X[l]}^{U_X[l]} nodes_{l,d} \tag{5.2}$$

$$nodes_{l,d} = \begin{cases} nodes_{l-1,d} + (\sigma - 1) \cdot nodes_{l-1,d-1} & \text{if } l \geq 1 \text{ and } L_X[l] \leq d \leq U_X[l] \\ 1 & \text{if } l = 0 \text{ and } d = 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

For a specific search, a closed formula can be given for $nodes_l$. If a search scheme $\mathcal{S}$ contains two or more searches with the same π-strings, these searches can be merged in order to eliminate the enumeration of the same string twice or more. It is straightforward to modify the computation of $\#\mathrm{str}(\mathcal{S}, X, \sigma, n)$ to account for this optimization.

Consider equation (5.1). The value of the term $1 - e^{-n/\sigma^l}$ is very close to 1 for $l \leq \log_\sigma n - O(1)$. When $l \geq \log_\sigma n$, the value of this term decreases exponentially. Note that $nodes_l$ increases exponentially, but the base of the exponent of $nodes_l$ is $\sigma - 1$ whereas the base of $1 - e^{-n/\sigma^l}$ is $1/\sigma$. We can then approximate $\#\text{str}(S, X, \sigma, n)$ with function $\#\text{str}'(S, X, \sigma, n)$ defined by

$$\#\text{str}'(S, X, \sigma, n) = \sum_{l=1}^{\lceil \log_\sigma n \rceil + c_\sigma} nodes_l \cdot (1 - e^{-n/\sigma^l}), \tag{5.4}$$

where c_σ is a constant chosen so that $((\sigma - 1)/\sigma)^{c_\sigma}$ is sufficiently small.

From the above formulas we have that the time complexities for computing $\#\text{str}(\mathcal{S}, X, \sigma, n)$ and $\#\text{str}'(\mathcal{S}, X, \sigma, n)$ are $O(|\mathcal{S}|km)$ and $O(|\mathcal{S}|k \log_\sigma n)$, respectively.

Edit distance

We now show how to estimate the efficiency of a search scheme for the edit distance.

We define $\#\text{str}_{\text{edit}}$ analogously to $\#\text{str}$ in the previous section, except that edit distance errors are allowed. Fix a search $S = (\pi, L, U)$ and a partition X. We assume without loss of generality that π is the identity permutation. Similarly to the Hamming distance case, define $\mathcal{A}_l$ to be the set of enumerated strings of length l when performing the search S on a random pattern of length m, partitioned by X, and a text $\hat{T}$ containing all the strings of length at most $m + k$ as substrings. Unlike the case of Hamming distance, here the strings of $\mathcal{A}_l$ are not distributed uniformly. Thus, we do not have the equality $\#\text{str}_{\text{edit}}(S, X, \sigma, n) = \sum_{l=1}^{m} nodes_l \cdot p_{n,l,\sigma}$. We will use $\sum_{l=1}^{m} nodes_l \cdot p_{n,l,\sigma}$ as an approximation for $\#\text{str}_{\text{edit}}(S, X, \sigma, n)$, but we do not have an estimation on the error of this approximation. Note that in the Hamming distance case, the sizes of the sets $\mathcal{A}_l$ are the same for every choice of the pattern, whereas this is not true for edit distance. We therefore define $nodes_l(P)$ to be the number of enumerated strings of length l when performing the search S on a pattern P of length m, partitioned by X, and a text $\hat{T}$. We also define $nodes_l$ to be the expectation of $nodes_l(P)$, where P is chosen randomly.

We next show how to compute values $nodes_l$. We begin by giving an algorithm for computing $nodes_l(P)$ for some fixed P. Build a non-deterministic automaton $\mathcal{A}_P$ that recognizes the set of strings that are within edit distance at most k to P, and the locations of the errors satisfy the requirements of the search [100, 101] (see Figure 5.2 for an example). For a state q and a string B, denote by $\hat{\delta}_P(q, B)$ the set of all states q' for which there is a path in $\mathcal{A}_P$ from q to q' such that the concatenation of the labels on the path is equal to B. For a set of states Q and a string B, $\hat{\delta}_P(Q, B) = \cup_{q \in Q} \hat{\delta}_P(q, B)$. Clearly, $nodes_l(P)$ is equal to the number of strings B of length l for which $\hat{\delta}_P(q_0, B) \neq \emptyset$, where q_0 is the initial state. Let $nodes_{l,Q}(P)$ be the number of strings B of length l for which $\hat{\delta}_P(q_0, B) = Q$. The values of $nodes_{l,Q}(P)$ can be computed using dynamic programming and the following recurrence.

$$nodes_{l,Q}(P) = \sum_{c \in \Sigma} \sum_{Q' : \hat{\delta}_P(Q', c) = Q} nodes_{l-1,Q'}(P).$$

The values $nodes_{l,Q}(P)$ gives the values of $nodes_l(P)$, since by definition,

$$nodes_l(P) = \sum_{Q} nodes_{l,Q}(P),$$

where the summation is done over all non-empty sets of states Q.

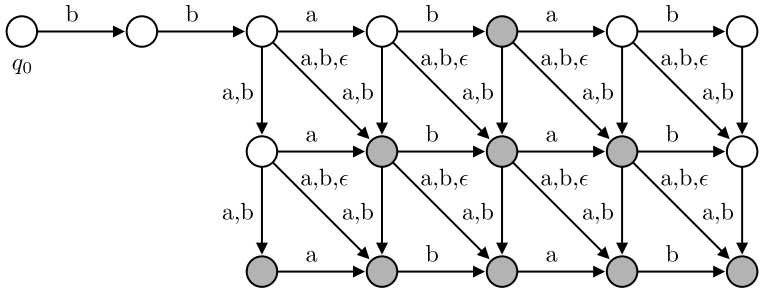


Figure 5.2: Non-deterministic automaton corresponding to the search $S = (12, 00, 02)$ and pattern $P =$ bbabab over the alphabet $\Sigma = \{\mathrm{a}, \mathrm{b}\}$. A path from the initial state q_0 to the state in the i-th row and j-column of the automaton correspond to a string with edit distance $i - 1$ to $P[1..j - 1]$. The nodes of the set Q_4 are marked by gray.

Note that for a string B of length l, set $\hat{\delta}_P(q_0, B)$ is a subset of a set of $(k+1)^2$ states that depends on l. This set, denoted Q_l, includes the $l+1$-th state in the first row of the automaton, states $l, l+1, l+2$ on the second row, states $l-1, l, \ldots, l+3$ on the third row, and so on (see Figure 5.2). The size of Q_l is $1+3+5+\cdots+(2k+1) = (k+1)^2$. Therefore, the number of sets Q for which $nodes_{l,Q}(P) > 0$ is at most $2^{(k+1)^2}$. If $(k+1)^2$ is small enough, a state can be encoded in one machine word, and the computation of $\hat{\delta}_P(Q', c)$ can be done in constant time using precomputed tables. Thus, the time for computing all values of $nodes_{l,Q}(P)$ is $O(2^{k^2}\sigma m)$.

Now consider the problem of computing the values of $nodes_l$. Observe that for $Q \subseteq \mathrm{Q}_l$, the value of $\hat{\delta}_P(Q, c)$ depends on the characters of $P[l-k+1..l+k+1]$, and does not depend on the rest of the characters of P. Our algorithm is based on this observation. For an integer l, a set $Q \subseteq \mathrm{Q}_l$, and a string P' of length $2k+1$, define

$$nodes_{l,Q,P'} = \sum_{P:P[l-k+1..l+k+1]=P'} nodes_{l,Q}(P).$$

Then,

$$nodes_{l,Q,P'} = \sum_{c' \in \Sigma} \sum_{c \in \Sigma} \sum_{Q':\hat{\delta}_{P_c}(Q',c)=Q} nodes_{l-1,Q',P'_c},$$

where $P'_c = c'P'[1..2k]$, and P_c is a string satisfying $P_c[(l-1)-k+1..(l-1)+k+1] = P'_c$ (the rest of the characters of P_c can be chosen arbitrarily).

From the above, the time complexity for computing $\#\mathrm{str}_{\mathrm{edit}}(S, X, \sigma, n)$ is $O(|\mathcal{S}|2^{k^2}\sigma^{2k+3}m)$. Therefore, our approach is practical only for small values of k.

5.3.2 Uneven partitions

In Section 5.2, we provided an informal explanation why partitioning the pattern into unequal parts may be beneficial. We now provide a formal justification for this. To this end, we replace (5.4) by an even simpler estimator of $\#\mathrm{str}(S, X, \sigma, n)$:

$$\#\mathrm{str}''(S, X, \sigma, n) = \sum_{l=1}^{\lceil \log_\sigma n \rceil} nodes_l. \tag{5.5}$$

As an example, consider scheme $\mathcal{S}_{\mathrm{LLTWWY}}$. Denote by x_1, x_2, x_3 the lengths of the parts in a partition X of P into 3 parts. It is straightforward to give closed formulas for

$\#\text{str}''(S, X, \sigma, n)$ for each search of $\mathcal{S}_{\text{LLTWWY}}$. For example,

$$\#\text{str}''(S_f, X, \sigma, n) = \begin{cases} N & \text{if } N \leq x_1 \\ c_1(N - x_1)^3 + c_2(N - x_1)^2 + c_3(N - x_1) + N & \text{otherwise} \end{cases}$$

where $N = \lceil \log_\sigma n \rceil$, $c_1 = (\sigma - 1)^2/6$, $c_2 = (\sigma - 1)/2$, and $c_3 = -(\sigma - 1)^2/6 + (\sigma - 1)/2$. Similar formulas can be given for S_b and S_{bd}. If x_1, x_2, and x_3 are close to $m/3$ and $N < m/3$ then $\#\text{str}''(\mathcal{S}_{\text{LLTWWY}}, X, \sigma, n, m) = 3N$ and an equal sized partition is optimal in this case. However, if $m/3 < N < 2m/3$, then

$$\begin{aligned}\#\text{str}''(\mathcal{S}_{\text{LLTWWY}}, X, \sigma, n) = {} & c_1(N - x_1)^3 + c_2(N - x_1)^2 + c_3(N - x_1) \\ & + c_1'(N - x_3)^2 + c_2'(N - x_3) + c_1''(N - x_2)^2 + c_2''(N - x_2) + 3N.\end{aligned}$$

It is now clear why the equal sized partition is not optimal in this case. The degree of $N - x_1$ in the above polynomial is 3, while the degrees of $N - x_2$ and $N - x_3$ are 2. Thus, if $x_1 = x_2 = x_3 = m/3$, decreasing x_2 and x_3 by, say 1, while increasing x_1 by 2 reduces the value of the polynomial.

5.3.3 Computing an optimal partition

In this Section, we show how to find an optimal partition for a given search scheme $\mathcal{S}$ and a given number of parts p. An optimal partition can be naively found by enumerating all $\binom{m-1}{p-1}$ possible partitions, and for each partition X, computing $\#\text{str}'(\mathcal{S}, X, \sigma, n)$. We now describe a more efficient dynamic programming algorithm.

We define an optimal partition to be a partition that minimizes $\#\text{str}(\mathcal{S}, X, \sigma, n)$. Let $N = \lceil \log_\sigma n \rceil + c_\sigma$. If $m \geq pN$, then any partition in which all parts are of size at least N is an optimal partition. Therefore, assume for the rest of this section that $m < pN$. We say that a partition X is *bounded* if the sizes of the parts of X are at most N. If X is not bounded, we can transform it into a bounded partition by decreasing the sizes of parts which are larger than N and increasing the sizes of parts which are smaller that N. This transformation can only decrease the value of $\#\text{str}(\mathcal{S}, X, \sigma, n)$. Therefore, there exists an optimal partition which is bounded. Throughout this section we will consider only bounded partitions. For brevity, we will use the term partition instead of bounded partition.

Our algorithm takes advantage of the fact that the value of $\#\text{str}'(\mathcal{S}, X, \sigma, n)$ does not depend on the entire partition X, but only on the partition of a substring of P of length N induced by X. More precisely, consider a fixed $S = (\pi, L, U) \in \mathcal{S}$. By definition, $\#\text{str}'(S, X, \sigma, n)$ depends on the values $nodes_1, \ldots, nodes_N$ (the number of nodes in levels $1, \ldots, N$ in the trie that correspond to the search S). From Section 5.3.1, these values depend on the strings L and U which are fixed, and on the string $\pi_X[1..N]$. The latter string depends on $\pi[1..i_{X,\pi}]$, where $i_{X,\pi}$ is the minimum index such that $\sum_{j=1}^{i_{X,\pi}} X[\pi(j)] \geq N$ and on the values $X[\pi(1)], \ldots, X[\pi(i_{X,\pi})]$.

The algorithm works by going over the prefixes of P in increasing length order. For each prefix P', it computes a set of partitions of P' such that at least one partition in this set can be extended to an optimal partition of P. In order to reduce the time complexity, the algorithm needs to identify partitions of P' that cannot be extended into an optimal partition of P. Consider the following example. Suppose that $m = 13$, $p = 5$, $N = 4$ and $\mathcal{S} = \{S_1, S_2, S_3\}$, where the π-strings of S_1, S_2, S_3 are $\pi^1 = 12345$, $\pi^2 = 32451$, and $\pi^3 = 43215$, respectively. Consider a prefix $P' = P[1..8]$ of P, and let Y_1, Y_2 be two partitions of P', where the parts in Y_1 are of sizes 3,3,2, and the parts in Y_1 are of sizes

4,2,2. Note that Y_1 and Y_2 have the same number of parts, and they induce the same partition on $P[8-N+1..8] = P[5..8]$. We claim that one of these two partitions is always at least as good as the other for every extension of both partitions to a partition of P. To see this, let Z denote a partition of $P[9..13]$ into two parts, and consider the three searches of $\mathcal{S}$.

1. For search S_1 we have that $\pi^1_{Y_1 \cup Z}[1..N] = 1112$ for every Z, and $\pi^1_{Y_2 \cup Z}[1..N] = 1111$ for every Z. It follows that the value of $\#\text{str}'(S_1, Y_1 \cup Z, \sigma, n)$ is the same for every Z, and the value of $\#\text{str}'(S_1, Y_2 \cup Z, \sigma, n)$ is the same for every Z. These two values can be equal or different.

2. For the search S_2 we have that $\pi^2_{Y_1 \cup Z}[1..N] = \pi^2_{Y_2 \cup Z}[1..N] = 3322$. It follows that $\#\text{str}'(S_2, Y_1 \cup Z, \sigma, n) = \#\text{str}'(S_2, Y_2 \cup Z, \sigma, n)$ for all Z and this common value does not depend on Z.

3. For the search S_3 we have that $\pi^3_{Y_1 \cup Z}[1..N] = \pi^3_{Y_2 \cup Z}[1..N]$ for every Z. For example, if Z is a partition of $P[9..13]$ into parts of sizes 2,2 then $\pi^3_{Y_1 \cup Z}[1..N] = \pi^3_{Y_2 \cup Z}[1..N] = 4433$. It follows that $\#\text{str}'(S_3, Y_1 \cup Z, \sigma, n) = \#\text{str}'(S_3, Y_2 \cup Z, \sigma, n)$ for every Z. This common value depends on Z.

We conclude that either $\#\text{str}'(\mathcal{S}, Y_1 \cup Z, \sigma, n) < \#\text{str}'(\mathcal{S}, Y_2 \cup Z, \sigma, n)$ for every Z, or $\#\text{str}'(\mathcal{S}, Y_1 \cup Z, \sigma, n) \geq \#\text{str}'(\mathcal{S}, Y_2 \cup Z, \sigma, n)$ for every Z.

We now give a formal description of the algorithm. We start with some definitions. For a partition Y of a substring $P' = P[m''..m']$ of pattern P, we define the following quantities: m_Y is the length of P', l_Y is the length of the last part of Y, p_Y is the number of parts in Y, and q_Y is the left-to-right rank of the part of Y containing $P'[m' - N + 1]$. Let prefix(Y) be the partition of $P[m''..m' - l_Y]$ of P' that is composed from the first $p_Y - 1$ parts of Y. For the example above, $m_{Y_1} = 8$, $l_{Y_1} = 2$, $p_{Y_1} = 3$, $q_{Y_1} = 2$, and prefix(Y_1) is a partition of $P[1..6]$ with parts sizes $3, 3$.

For a partition Y of a prefix P' of P, $\mathcal{S}(Y)$ is a set containing every search $S \in \mathcal{S}$ such that q_Y appears before $p_Y + 1$ in the π-string of S. If the length of P' is less than N we define $\mathcal{S}(Y) = \emptyset$, and if $P' = P$ we define $\mathcal{S}(Y) = \mathcal{S}$. For the example above, $\mathcal{S}(Y_1) = \{S_1, S_2\}$.

Let Y_1 be a partition of a substring $P_1 = P[i_1..j_1]$ of P, and Y_2 be a partition of a substring $P_2 = P[i_2..j_2]$. We say that Y_1 and Y_2 are *compatible* if these partitions induce the same partition on the common substring $P' = P[\max(i_1, i_2).. \min(j_1, j_2)]$. For example, the partition of $P[4..6]$ into parts of sizes $1, 2$ is compatible with the partition of $P[1..6]$ into parts of sizes $2, 2, 2$.

Lemma 5. *Let Y be a partition of a prefix of P of length at least N. Let $S \in \mathcal{S}(Y)$ be a search. The value $\#\text{str}'(S, X, \sigma, n)$ is the same for every partition X of P whose first p_Y parts match Y.*

Proof. Let i' be the index such that $\pi(i') = p_Y + 1$. Since q_Y appears before $p_Y + 1$ in string π, from the connectivity property of π we have that (1) Every value in π that appears before $p_Y + 1$ is at most p_Y. In other words, $\pi(i) \leq p_Y$ for every $i < i'$. (2) $q_Y, \ldots, p_Y$ appear before $p_Y + 1$ in π. By the definition of q_Y, $\sum_{j=q_Y}^{p_Y} X[j] \geq N$. Therefore, $i_{X,\pi} < i'$ and $\pi(1), \ldots, \pi(i_{X,\pi}) \leq p_Y$. Thus, string $\pi[1..i_{X,\pi}]$ and values $X[\pi(1)], \ldots, X[\pi(i_{X,\pi})]$ are the same for every partition X that satisfies the requirement of the lemma. ■

For a partition Y of a prefix of P of length at least N, define $v(Y)$ to be $\sum_{S \in \mathcal{S}(Y)} \#\text{str}'(S, X, \sigma, n)$, where X is an arbitrary partition of P whose first p_Y parts match Y (the choice of X does

not matter due to Lemma 5). For a partition Y of a prefix of P of length less than N, $v(Y) = 0$. Define

$$\Delta(Y) = v(Y) - v(\text{prefix}(Y)) = \sum_{S \in \mathcal{S}(Y) \setminus \mathcal{S}(\text{prefix}(Y))} \#\text{str}'(S, X, \sigma, n).$$

Lemma 6. *Let Z be a partition of a substring $P[m''..m']$ such that $p_Z \geq 2$ and $m_{\text{prefix}(Z)} = \min(N, m' - l_Y)$. Let $p' \geq p_Z$ be an integer. The value of $\Delta(Y)$ is the same for every partition Y of $P[1..m']$ with p' parts that is compatible with Z.*

Proof. We assume $N < m' - l_Y$ (the case $N \geq m' - l_Y$ is similar). Since $m_{\text{prefix}(Z)} = \min(N, m' - l_Y)$, the set $\mathcal{S}(Y) \setminus \mathcal{S}(\text{prefix}(Y))$ is the same for every partition Y of $P[1..m']$ with p' parts that is compatible with Z. For a search $S = (\pi, L, U)$ in this set, q_Y appears before $p_Y + 1$ in π, and p_Y appears before $q_{\text{prefix}(Y)}$. Let $i = i_{X,\pi}$, where X is an arbitrary partition of P whose first p_Y parts are the parts of Y. We obtain that $q_{\text{prefix}(Y)} \leq \pi(1), \ldots, \pi(i) \leq p_Y$, and the lemma follows. ∎

For Z, p' that satisfy the requirements of Lemma 6, let $\Delta(Z, p')$ denote the value of $\Delta(Y)$, where Y is an arbitrary partition of $P[1..m']$ with p' parts that is compatible with Z.

For $m' \leq m$, $p' \leq p$, and a partition Z of $P[\max(m' - N + 1, 1)..m']$ with at most p' parts, let $v(m', p', Z)$ be the minimum value of $v(Y)$, where Y is a partition of $P[1..m']$ into p' parts that is compatible with Z.

Lemma 7. *For $m' \leq m$, $2 \leq p' \leq p$, and a partition Z of $P[\max(m' - N + 1, 1)..m']$ with at most p' parts,*

$$v(m', p', Z) = \min_{Z'} \big(v(m' - l_{Z'}, p' - 1, \text{prefix}(Z')) + \Delta(Z', p')\big)$$

where the minimum is taken over all partitions Z' of a substring $P[m''..m']$ of P that satisfy the following: (1) Z' is compatible with Z, (2) $2 \leq p_{Z'} \leq p'$, (3) $m_{\text{prefix}(Z')} = \min(N, m' - l_{Z'})$, (4) $p_Z = p'$ if $m'' = 1$.

An algorithm for computing the optimal partition follows from Lemma 7. The time complexity of the algorithm is $O\big((|\mathcal{S}|kN + m) \sum_{j=1}^{\min(p-1,N)} (p-j)\binom{N-1}{j-1}\big)$, where $|\mathcal{S}|kN \sum_{j=1}^{\min(p-1,N)} (p-j)\binom{N-1}{j-1}$ is time for computing Δ values, and $O\big(m \sum_{j=1}^{\min(p-1,N)} (p-j)\binom{N-1}{j-1}\big)$ is time for computing v values.

5.4 Properties of optimal search schemes

Designing an efficient search scheme for a given set of parameters consists of (1) choosing a number of parts, (2) choosing searches, (3) choosing a partition of the pattern. While it is possible to enumerate all possible choices, and evaluate the efficiency of the resulting scheme using Section 5.3.1, this is generally infeasible due to a large number of possibilities. It is therefore desirable to have a combinatorial characterization of optimal search schemes.

The *critical string* of a search scheme $\mathcal{S}$ is the lexicographically maximal U-string of a search in $\mathcal{S}$. A search of $\mathcal{S}$ is *critical* if its U-string is equal to the critical string of $\mathcal{S}$. For example, the critical string of $\mathcal{S}_{\text{LLTWWY}}$ is 022, and S_f is the critical search. For typical parameters, critical searches of a search scheme constitute the bottleneck. Consider a search scheme $\mathcal{S}$, and assume that the L-strings of all searches contain only zeros. Assume further that the pattern is partitioned into equal-size parts. Let ℓ be the maximum index

such that for every search $S \in \mathcal{S}$ and every $i \leq \ell$, $U[i]$ of S is no larger than the number in position i in the critical string of $\mathcal{S}$. From Section 5.3, the number of strings enumerated by a search $S \in \mathcal{S}$ depends mostly on the prefix of the U-string of S of length $\lceil \lceil \log_\sigma n \rceil / (m/p) \rceil$. Thus, if $\lceil \lceil \log_\sigma n \rceil / (m/p) \rceil \leq \ell$, a critical search enumerates an equal or greater number of strings than a non-critical search.

We now consider the problem of designing a search scheme whose critical string is minimal. Let $\alpha(k, p)$ denote the lexicographically minimal critical string of a k-mismatch search scheme that partitions the pattern into p parts. The next theorems give the values of $\alpha(k, k+2)$ and $\alpha(k, k+1)$. We need the following definition. A string over the alphabet of integers is called *simple* if it contains a substring of the form 01^j0 for $j \geq 0$.

Lemma 8. *(i) Every string A of weight k and length at least $k+2$ is simple.*

(ii) If A is a non-simple string of weight k and length $k+1$ then $A[1] \leq 1$, $A[k+1] \leq 1$, and $A[i] \leq 2$ for all $2 \leq i \leq k$. Moreover, there are no two consecutive 2's in A.

Proof. *(i)* The proof is by induction on k. It is easy to verify that the lemma holds for $k = 0$. Suppose we proved the lemma for $k' < k$. Let A be a string of weight k and length $p \geq k+2$. If $A[1] \geq 1$ then by the induction hypothesis $A[2..p]$ is simple, and therefore A is simple. Suppose that $A[1] = 0$. Let $i > 1$ be the minimum index such that $A[i] \neq 1$ (i must exist due to the assumption that $p \geq k+2$). If $A[i] = 0$ then we are done. Otherwise, we can use the induction hypothesis on $A[i+1..p]$ and obtain that A is simple.

(ii) Let A be a non-simple string of weight k and length $k+1$. If $A[1] \geq 2$ then $A' = A[2..k+1]$ has weight $k - A[1] \leq k-2$ and length k, and thus by *(i)* we obtain that A' is simple, contradicting the assumption that A is non-simple. Similarly, $A[k+1]$ cannot be greater than 1. For $2 \leq i \leq k$, if $A[i] \geq 3$ then either $A[1..i-1]$ or $A[i+1..k+1]$ satisfies the condition of *(i)*. Similarly, if $A[i] = A[i+1] = 2$ then either $A[1..i-1]$ or $A[i+2..k+1]$ satisfies the condition of *(i)*. ∎

We use the following notation. For two integers i and j, $[i, j]$ denotes the string $i(i+1)(i+2)\cdots j$ if $i \leq j$, and the empty string if $i > j$. Moreover, $\overline{[i, j]}$ denotes the string $i(i-1)(i-2)\cdots j$ if $i \geq j$, and the empty string if $i < j$.

Theorem 9. *$\alpha(k, k+1) = 013355\cdots kk$ for every odd k, and $\alpha(k, k+1) = 02244\cdots kk$ for every even k.*

Proof. We first give an upper bound on $\alpha(k, k+1)$ for odd k. We build a search scheme as follows. The scheme contains searches $S_{k,i,j} = ([i, k+2]\overline{[i-1, 1]}, 0\cdots0, [0, j]jk\cdots k)$ for all i and j, which cover all simple strings of weight k and length $k+1$. In order to cover the non-simple strings, the scheme contains the following searches.

1. $S^1_{k,i,j} = ([i, k+1]\overline{[i-1, 1]}, 0\cdots0, 013355\cdots jj(j+1)k\cdots k)$ for every odd $3 \leq j \leq k$ (for $j = k$, the U-string is $013355\cdots kk$).

2. $S^2_{k,i,j} = (\overline{[i, 1]}[i+1, k+1], 0\cdots0, 013355\cdots jj(j+1)k\cdots k)$ for every odd $3 \leq j \leq k$ (for $j = k$, the U-string is $013355\cdots kk$).

Let A be a non-simple string of weight k and length $k+1$. By Lemma 8, $A = X0A_10A_20\cdots0A_d0Y$ where each of X and Y is either string 1 or empty string, and each A_i is either 2, 12, 21, or 121. A string A_i is called a *block* of type 1, 2, or 3 if A_i is equal to 12, 21, or 121, respectively. Let $B_1, \ldots, B_{d'}$ be the blocks of type 1 and type 2, from left to right.

We consider several cases. The first case is when X and Y are empty strings, and B_1 is of type 1. Since the weight of A is odd, it follows that d' is odd. If A has no other blocks, A is covered by search $S^1_{k,i,k}$, where $i+1$ is the index in A in which B_1 starts. Otherwise, if B_2 is of type 1, then A is covered by search $S^1_{k,i,j}$, where $i+1$ is the index in A in which B_1 starts, and $i+j+1$ is the index in which the first block to the right of B_1 starts (this block is either B_2, or a block of type 3). Now suppose that B_2 is of type 2. If B_3 is of type 2, then A is covered by search $S^2_{k,i,j}$, where $i-1$ is the index in A in which B_3 ends, and $i-j-1$ is the index in which the first block to the left of B_3 ends. By repeating these arguments, we obtain that A is covered unless the types of $B_1, \ldots, B_{d'}$ alternate between type 1 and type 2. However, since d' is odd, $B_{d'}$ is of type 1, and in this case A is covered by $S^1_{k,i,j}$, where $i+1$ is the index in A in which B_1 starts, and $k-j$ is the index in which the first block to the left of B_1 ends.

Now, if X is empty string and $Y = 1$, define a string $A' = A20$. By the above, A' is covered by some search $S^{j'}_{k+2,i,j}$. Then, A is covered by either $S^{j'}_{k,i,j}$ or $S^{j'}_{k,i,j-2}$. The same argument holds for the case when $X = 1$. The proof for the case when B_1 is of type 2 is analogous and thus omitted.

The lower bound on $\alpha(k, k+1)$ for odd k is obtained by considering the string $A = 012020\cdots20$. The U-string of a search that covers A must be at least $013355\cdots kk$.

We next give an upper bound on $\alpha(k, k+1)$ for even k. We define k-mismatch search schemes $\mathcal{S}_k$ recursively. For $k = 0$, $\mathcal{S}_0$ consists of a single search $S_{0,1} = (1, 0, 0)$. For $k \geq 2$, $\mathcal{S}_k$ consists of the following searches.

1. For every search $S_{k-2,i} = (\pi, 0\cdots0, U)$ in $\mathcal{S}_{k-2}$, $\mathcal{S}_k$ contains a search $S_{k,i} = (\pi \cdot k(k+1), 0\cdots0, U \cdot kk)$.
2. A search $S_{k,k} = (\overline{[k+1,1]}, 0\cdots0, 01kk\cdots k)$.
3. A search $S_{k,k+1} = (k(k+1)\overline{[k-1,1]}, 0\cdots0, 01kk\cdots k)$.

Note that the critical string of $\mathcal{S}_k$ is $02244\cdots kk$ corresponding to item 1 above. We now claim that all number strings of length $k+1$ and weight at most k are covered by the searches of $\mathcal{S}_k$. The proof is by induction on k. The base $k = 0$ is trivial. Suppose the claim holds for $k-2$. Let A be a number string of length $k+1$ and weight $k' \leq k$. If $A[k] + A[k+1] \leq 1$, then A is covered by either $S_{k,k}$ or $S_{k,k+1}$. Otherwise, the weight of $A' = A[1..k-1]$ is at most $k'-2$. By induction, A' is covered by some search $S_{k-2,i}$. Then search $S_{k,i}$ covers A.

To prove that $\alpha(k, k+1) \geq 02244\cdots kk$ for even k, consider the string $A = 0202\cdots020$. It is easy to verify that the U-string of a search that covers A must be at least $02244\cdots kk$. ■

Theorem 10. *$\alpha(k, k+2) = 0123\cdots(k-1)kk$ for every $k \geq 1$.*

Proof. We first give an upper bound on $\alpha(k, k+1)$. We build a k-mismatch search scheme $\mathcal{S}$ that contains searches $S_{k,i,j} = ([i, k+2]\overline{[i-1,1]}, 0\cdots0, [0, j]jk\cdots k)$ for all i and j. Let A be a string of weight k and length $k+2$. By Lemma 8 there are indices i and j such that $A[i..i+j+1] = 01^j0$, and therefore A is covered by $S_{k,i,j}$.

The lower bound is obtained from the string $A = 011\cdots110$. It is easy to verify that the U-string of a search that covers A must be at least $0123\cdots(k-1)kk$. ■

An important consequence of Theorems 9 and 10 is that for some typical cases, partitioning the pattern into $k+2$ parts brings an advantage over $k+1$ parts. For $k = 2$, for example, we have $\alpha(2, 3) = 022$ while $\alpha(2, 4) = 0122$. Since the second element of 0122 is

smaller than that of 022, a 4-part search scheme potentially enumerates less strings than a 3-part scheme. On the other hand, the average length of a part is smaller when using 4 parts, and therefore the branching occurs earlier in the searches of a 4-part scheme. The next section shows that for some parameters, $(k+2)$-part schemes outperform $(k+1)$-part schemes, while for other parameters the inverse occurs.

5.5 Case studies

In this Section, we provide results of several computational experiments we have performed to analyse practical applicability of our techniques.

We designed search schemes for 2, 3 and 4 errors provided below using a greedy algorithm. The algorithm iteratively adds searches to a search scheme. At each step, the algorithm considers the uncovered string A of weight k such that the lexicographically minimal U-string that covers A is maximal. Among the searches that cover A with minimal U-string, a search that covers the maximum number of uncovered strings of weight k is chosen. The L-string of the search is chosen to be lexicographically maximal among all possible L-string that do not decrease the number of uncovered strings. For each search scheme and each choice of parameters, we computed an optimal partition.
For 2 mismatches or errors:

1. Slightly modified scheme $\mathcal{S}_{\text{LLTWWY}}$. The searches are: $S_f = (123, 000, 022)$, $S_b = (321, 000, 012)$, and $S'_{bd} = (213, 001, 012)$. Note that the π-string of S'_{bd} is 213 and not 231 as in S_{bd}. While S_{bd} and S'_{bd} have the same efficiency for equal-size partitions, this in not the case for unequally sized parts.

2. 4-part scheme with searches $(1234, 0000, 0112)$, $(4321, 0000, 0122)$, $(2341, 0001, 0012)$, and $(1234, 0002, 0022)$.

For 3 mismatches or errors:

1. 4-part scheme with searches $(1234, 0000, 0133)$, $(2134, 0011, 0133)$, $(3421, 0000, 0133)$, and $(4321, 0011, 0133)$.

2. 5-part scheme with searches $(12345, 00000, 01233)$, $(23451, 00000, 01223)$, $(34521, 00001, 01133)$, and $(45321, 00012, 00333)$.

For 4 mismatches or errors:

1. 5-part scheme with searches $(12345, 00000, 02244)$, $(54321, 00000, 01344)$, $(21345, 00133, 01334)$, $(12345, 00133, 01334)$, $(43521, 00011, 01244)$, $(32145, 00013, 01244)$, $(21345, 00124, 01244)$ and $(12345, 00034, 00444)$.

2. 6-part scheme with searches $(123456, 00000, 012344)$, $(234561, 00000, 012344)$, $(654321, 000001, 012244)$, $(456321, 000012, 011344)$, $(345621, 000023, 011244)$, $(564321, 000133, 003344)$, $(123456, 000333, 003344)$ $(123456, 000044, 002444)$, $(342156, 000124, 002244)$ and $(564321, 000044, 001444)$.

5.5.1 Numerical comparison of search schemes

We first performed a comparative estimation of the efficiency of search schemes using the method of Section 5.3.1 (case of Hamming distance). More precisely, for a given search

Table 5.1: Values of $\#\mathrm{str}(\mathcal{S}, X, 4, 4^{16})$ for 2-mismatch search schemes, for different pattern lengths m. Second column corresponds to search scheme $\mathcal{S}_{\mathrm{LLTWWY}}$ with three equal-size parts, the other columns show results for unequal partitions and/or more parts. The partition used is shown in the second sub-column.

m	3 equal	3 unequal		4 unequal		5 unequal	
24	1197	1077	9,7,8	959	7,4,4,9	939	7,1,6,1,9
36	241	165	15,10,11	140	12,5,7,12	165	11,1,9,1,14
48	53	53	16,16,16	51	16,7,9,16	53	16,1,15,1,15

Table 5.2: Values of $\#\mathrm{str}(\mathcal{S}, X, 30, 30^7)$ for 2-mismatch search schemes.

m	3 equal	3 unequal		4 unequal		5 unequal	
15	846	286	6,4,5	231	5,2,3,5	286	5,1,3,1,5
18	112	111	7,6,5	81	6,2,4,6	111	6,1,4,1,6
21	24	24	7,7,7	23	7,3,4,7	24	7,1,6,1,6

scheme $\mathcal{S}$, we estimated the number of strings $\#\mathrm{str}(\mathcal{S}, X, \sigma, n)$ enumerated during the search.

Results for 2 mismatches are given in Table 5.1 and Table 5.2 for 4-letter and 30-letter alphabets respectively. Table 5.3 contains estimations for nonuniform letter distribution. Table 5.4 contains estimations for 3 mismatches for 4-letter alphabet.

We first observe that our method provides an advantage only on a limited range of pattern lengths. This conforms to our analysis (see Section 5.3.2) that implies that our schemes can bring an improvement when $m/(k+1)$ is smaller than $\log_\sigma n$ approximately. When $m/(k+1)$ is small, Tables 5.1–5.4 suggest that using more parts of unequal size can bring a significant improvement. For big alphabets (Table 5.2), we observe a larger gain in efficiency, due to the fact that values $nodes_l$ (see equation (5.2)) grow faster when the alphabet is large, and thus a change in the size of parts can have a bigger influence on these values. Moreover, if the probability distribution of letters in both the text and the pattern is nonuniform, then we obtain an even larger gain (Table 5.3), since in this case, the strings enumerated during the search have a larger probability to appear in the text than for the uniform distribution.

For 3 mismatches and 4 letters (Table 5.4), we observe a smaller gain, and even a loss for pattern lengths 36 and 48 when shifting from 4 to 5 parts. This is explained by Theorem 9 showing the difference of critical strings between odd and even numbers of errors. Thus, for 3 mismatches and 4 parts, the critical string is 0133 while for 5 parts it is 01233. When patterns are not too small, the latter does not lead to an improvement strong enough to compensate for the decrease of part length. Note that the situation is different for even number of errors, where incrementing the number of parts from $k+1$ to $k+2$ leads to transforming the critical strings from $0224\cdots$ to $0123\cdots$.

Another interesting observation is that with 4 parts, obtained optimal partitions have equal-size parts, as the U-strings of all searches of the 4-part scheme are all the same (see Section 5.5.2).

These estimations suggest that our techniques can bring a significant gain in efficiency for some parameter ranges, however the design of a search scheme should be done carefully for each specific set of parameters.

Table 5.3: Values of $\#\text{str}(\mathcal{S}, X, 4, 4^{16})$ for 2-mismatch search schemes, using a non-uniform letter distribution (one letter with probability 0.01 and the rest with probability 0.33 each).

m	3 equal	3 unequal		4 unequal		5 unequal	
24	3997	3541	10,8,6	3592	6,7,1,10	3541	6,1,7,1,9
36	946	481	16,10,10	450	11,6,6,13	481	10,1,9,1,15
48	203	157	18,15,15	137	16,7,9,16	157	15,1,14,1,17

Table 5.4: Values of $\#\text{str}(\mathcal{S}, X, 4, 4^{16})$ for 3-mismatch search schemes. Best partitions obtained for 4 parts are equal.

m	4 equal/unequal		5 unequal	
24	11222	6,6,6,6	8039	4,6,5,1,8
36	416	9,9,9,9	549	6,11,5,1,13
48	185	12,12,12,12	213	11,11,11,1,14

5.5.2 Experiments on genomic data

To perform large-scale experiments on genomic sequences, we implemented our method using the 2BWT library provided by [11] (http://i.cs.hku.hk/2bwt-tools/). We then experimentally compared different search schemes, both in terms of running time and average number of enumerated substrings. Below we only report running time, as in all cases, the number of enumerated substrings produced very similar results.

The experiments were done on the sequence of human chromosome 14 (*hr14*). The sequence is $88 \cdot 10^6$ long, with nucleotide distribution 29%, 21%, 21%, 29%. Searched patterns were generated as i.i.d. sequences. For every search scheme and pattern length, we ran 10^5 pattern searches for Hamming distance and 10^4 searches for the edit distance.

Search schemes

The following search schemes were used in experiments in this Section.
For 2 mismatches or errors:

1. Slightly modified scheme $\mathcal{S}_{\text{LLTWWY}}$. The searches are: $S_f = (123, 000, 022)$, $S_b = (321, 000, 012)$, and $S'_{bd} = (213, 001, 012)$. Note that the π-string of S'_{bd} is 213 and not 231 as in S_{bd}. While S_{bd} and S'_{bd} have the same efficiency for equal-size partitions, this in not the case for unequally sized parts.

2. 4-part scheme with searches $(1234, 0000, 0112)$, $(4321, 0000, 0122)$, $(2341, 0001, 0012)$, and $(1234, 0002, 0022)$.

For 3 mismatches or errors:

1. 4-part scheme with searches $(1234, 0000, 0133)$, $(2134, 0011, 0133)$, $(3421, 0000, 0133)$, and $(4321, 0011, 0133)$.

2. 5-part scheme with searches $(12345, 00000, 01233)$, $(23451, 00000, 01223)$, $(34521, 00001, 01133)$, and $(45321, 00012, 00333)$.

For 4 mismatches or errors:

Table 5.5: Total time (in sec) of search for 10^5 patterns in *hr14*, up to 2 mismatches. 2nd column contains time obtained on partition into three equal-size parts. The 3rd (respectively 4th and 5th) column shows the running time respectively for the 3-unequal-parts, 4-equal-parts and 4-unequal-parts searches, together with their ratio (%) to the corresponding 3-equal-parts value.

m	3 equal	3 unequal		4 equal	4 unequal	
15	24.8	25.4 (102%)	6,6,3	25.3 (102%)	25.3 (102%)	3,5,1,6
24	5.5	4.2 (76%)	10,7,7	5.2 (95%)	4.0 (73%)	7,4,4,9
33	1.73	1.45 (84%)	13,10,10	2.07 (120%)	1.25 (72%)	11,5,6,11
42	0.71	0.71 (100%)	14,14,14	1.24 (175%)	0.82 (115%)	14,6,8,14

Table 5.6: Total time (in sec) of search for 10^5 patterns in *hr14*, up to 3 mismatches.

m	4 equal	5 equal	5 unequal	
15	241	211 (86%)	206 (85%)	2,3,5,1,4
24	19.7	26.7 (136%)	19.6 (99%)	2,9,3,1,9
33	4.3	6.9 (160%)	4.7 (109%)	6,9,6,1,11
42	1.85	2.52 (136%)	2.05 (111%)	10,10,9,1,12
51	1.07	1.57 (147%)	1.06 (99%)	12,13,12,1,13

1. 5-part scheme with searches $(12345, 00000, 02244)$, $(54321, 00000, 01344)$, $(21345, 00133, 01334)$, $(12345, 00133, 01334)$, $(43521, 00011, 01244)$, $(32145, 00013, 01244)$, $(21345, 00124, 01244)$ and $(12345, 00034, 00444)$.

2. 6-part scheme with searches $(123456, 00000, 012344)$, $(234561, 00000, 012344)$, $(654321, 000001, 012244)$, $(456321, 000012, 011344)$, $(345621, 000023, 011244)$, $(564321, 000133, 003344)$, $(123456, 000333, 003344)$ $(123456, 000044, 002444)$, $(342156, 000124, 002244)$ and $(564321, 000044, 001444)$.

Hamming distance

For the case of 2 mismatches, we implemented the 3-part and 4-part schemes (see Section 5.5.2), as well as their equal-size-part versions for comparison. For each pattern length, we computed an optimal partition, taking into account a non-uniform distribution of nucleotides. Results are presented in Table 5.5.

Using unequal parts for 3-part schemes yields a notable time decrease for patterns of length 24 and 33 (respectively, by 24% and 16%). Furthermore, we observe that using unequal part lengths for 4-part schemes is beneficial as well. For pattern lengths 24 and 33, we obtain a speed-up by 27% and 28% respectively. Overall, the experimental results are consistent with numerical estimations of Section 5.5.1.

For the case of 3 mismatches, we implemented 4-part and 5-part schemes from Section 5.5.2, as well as their equal part versions for comparison. Results (running time) are presented in Table 5.6. In accordance with estimations of Section 5.5.1, here we observe a clear improvement only for pattern length 15 and not for longer patterns.

Edit distance

In the case of edit distance, along with the search schemes for 2 and 3 errors from the previous section, we also implemented search schemes for 4 errors (see Section 5.5.2). Results are shown in Table 5.7 (2 errors), Table 5.8 (3 errors) and Table 5.9 (4 errors).

Table 5.7: Total time (in sec) of search for 10^4 patterns in *hr14*, up to 2 errors (edit distance).

m	3 equal	3 unequal		4 equal	4 unequal	
15	11.5	11.4 (99%)	6,6,3	10.9 (95%)	11.1 (97%)	3,5,1,6
24	2.1	1.3 (62%)	11,5,8	1.5 (71%)	1.0 (48%)	7,4,4,9
33	0.34	0.22 (65%)	13,10,10	0.35 (103%)	0.19 (56%)	11,5,6,11
42	0.08	0.08 (100%)	14,14,14	0.18 (225%)	0.08 (100%)	14,6,8,14

Table 5.8: Total time (in sec) of search for 10^4 patterns in *hr14*, up to 3 errors (edit distance).

m	4 equal	5 equal	5 unequal	
15	233	174 (75%)	168 (72%)	2,2,6,1,4
24	13.5	13.2 (98%)	10.8 (80%)	3,8,3,1,9
33	0.74	1.81 (245%)	1.07 (145%)	5,10,5,1,12
42	0.28	0.45 (161%)	0.37 (132%)	9,10,9,1,13
51	0.13	0.24 (185%)	0.14 (108%)	12,12,12,1,14

Table 5.9: Total time (in sec) of search for 10^4 patterns in *hr14*, up to 4 errors (edit distance).

m	5 equal	5 unequal		6 equal	6 unequal	
15	4212	3222 (76%)	3,1,8,1,2	4028 (96%)	3401 (81%)	2,2,1,7,1,2
24	145	133 (92%)	7,3,5,1,8	131 (90%)	113 (78%)	2,7,3,4,5,3
33	6.5	5.8 (89%)	8,7,5,8,5	6.6 (102%)	5.1 (78%)	4,8,6,3,5,7
42	1.66	1.16 (70%)	12,8,7,8,7	1.51 (91%)	1.17 (70%)	7,8,8,5,2,12
51	0.60	0.49 (82%)	13,11,9,9,9	0.74 (123%)	0.54 (90%)	9,10,9,9,1,13
60	0.28	0.24 (86%)	14,13,11,11,11	0.44 (157%)	0.28 (117%)	11,12,11,11,1,14

Table 5.10: Total time (in sec) of search for 10^5 reads in *hr14*, up to 4 errors. First row corresponds to read set with constant error rate 0.03. Second row corresponds to read set with error rate increasing from 0.0 to 0.03.

m	5 equal	6 equal	6 unequal	
100	247	250 (101%)	283 (115%)	20,20,20,19,1,20
100	415	367 (88%)	350 (84%)	20,20,20,19,1,20

For 2 errors, we observe up to two-fold speed-up for pattern lengths 15, 24 and 33. For the case of 3 errors, the improvement is achieved for pattern lengths 15 and 24 (respectively 28% and 20%). Finally, for 4 errors, we obtain a significant speed-up (18% to 30%) for pattern lengths between 15 and 51.

Experiments on simulated genomic reads

Experiments of Section 5.5.2 have been made with random patterns. In order to make experiments closer to the practical bioinformatic setting occurring in mapping genomic reads to their reference sequence, we also experimented with patterns simulating reads issued from genomic sequencers. For that, we generated realistic single-end reads of length 100 (typical length of ILLUMINA reads) from *hr14* using DWGSIM read simulator (https://github.com/nh13/DWGSIM). Two sets of reads were generated using two different error

rate values (parameter `-e` of DWGSIM): `0.03` for the first dataset and `0.0-0.03` for the second one. This means that in the first set, error probability is uniform over the read length, while in the second set, this probability gradually increases from 0 to 0.03 towards the right end of the read. The latter simulates the real-life situation occurring with current sequencing technologies including ILLUMINA.

The results are shown in Table 5.10. As expected, due to a large pattern length, our schemes did not produce a speed-up for the case of constant error rate. Interestingly however, for the case of non-uniform distribution of errors, our schemes showed a clear advantage. This illustrates another possible benefit of our techniques: they are better adapted to a search for patterns with non-uniform distribution of errors, which often occurs in practical situations such as mapping genomic reads.

5.6 Discussion

Methods described in this chapter can be seen as the first step towards an automated design of efficient search schemes for approximate string matching, based on bidirectional indexes. More research has to be done in order to allow an automated design of optimal search schemes. It would be very interesting to study an approach when a search scheme is designed simultaneously with the partition, rather than independently as it was done in our work.

We expect that search schemes similar to those studied in this work can be applied to hybrid approaches to approximate matching (see 5.1), as well as possibly to other search strategies.

Part III

Efficient representation of large genomic data with Cascading Bloom filters

Contents - Part III

Chapter 6

Algorithmic methods for genome assembly

6.1 Genome assembly

Accurate assembly of genomes lies in the basis of many modern bioinformatic and medical projects. The goal of assembly is, given a set of reads, to build an initial DNA sequence. However, due to reasons we discuss below (repeats and sequencing errors), it is usually impossible, and long contiguous DNA sequences, called *contigs*, are reconstructed instead. Assembled contigs can then be used as a reference for read alignment, thus, in some sense, this is a necessary step before read alignment can be performed. Genome assembly can be compared to solving a puzzle, when you need to combine many (from millions to billions) small fragments into much longer pieces. In this thesis, we focus on efficient data structures for storing reads for their assembly, while many other related algorithmic questions remain out of scope of our work.

Let $\mathcal{R}$ be a set of reads, and our goal is to construct a set of contigs $\mathcal{C}$. Although there is no formal criteria for the "best" set of contigs, different sets can be compared with respect to various characteristics like the size of $\mathcal{C}$, the average contig length or the N50 value. More details about different genome assemblers comparison can be found in [102, 103, 104]. A recent comparison of popular assemblers was made on `insidedna.me` platform (`https://insidedna.me/tutorials/view/benchmark-seven-popular-genome-assemblers`). There was also an open-data competition called Assemblathon [105] where assemblers sent by many teams are compared.

The main challenge for all assembly algorithms is repeated sequences in genomes. Different regions often share perfect or almost perfect repeats, whose origin can not be distinguished, if reads are shorter than these repeats. Additionally, errors introduced during sequencing make the assembly even more complicated. Assembly tools, taking into consideration possible errors, can find some false positive connections between reads.

Here we describe two main algorithmic approaches to the genome assembly problem: *overlap-layout-consensus* (OLC) and *de Bruijn graph* based approaches. The key idea of both of them is to design a data structure representing overlaps between reads, and then merge reads together.

OLC-based approaches generally implement three major steps:

1. calculate *overlaps* among all the reads,
2. find *layout*: find series of overlapping reads,

3. obtain *consensus* joining reads merging overlaps.

Overlap-layout-consensus implementations are usually based on the *string* (or *assembly*) *graph* concept: the graph whose nodes correspond to reads, and edges represent overlaps between two reads. Usually, a basic operation that a string graph implementation should support is to quickly return reads which have a significant overlap with a given read.

Many computational tools dealing with NGS data, especially those devoted to *genome assembly*, are based on the concept of a *de Bruijn graph*, see e.g. [106]. A *de Bruijn graph*, for a given parameter k, of a set of reads $\mathcal{R}$ is entirely defined by the set $T \subseteq U = \Sigma^k$ of k-mers present in $\mathcal{R}$. The nodes of the graph are precisely the k-mers of T and for any two vertices $u, v \in T$, there is an edge from u to v if the suffix of u of size $k-1$ is equal to the prefix of v of the same size. Note that this is actually a *subgraph* of the de Bruijn graph under its classical combinatorial definition. However, we still call it de Bruijn graph to follow the terminology common to the bioinformatics literature. The value of k is an open parameter, which can be, and is often required to be, specified by the user.

De Bruijn graph-based assemblers follow the same general outline:

1. extract k-mers from the reads,
2. construct the de Bruijn graph on the obtained set,
3. simplify the graph,
4. output paths in the graph as contigs.

There are many assemblers based both on Overlap-Layout-Consensus paradigm and on de Bruijn graphs. However, from the algorithmic point of view, these two approaches are very different. Finding the layout (step 2) in a graph corresponds to finding a Hamiltonian path, which is a well-known NP-hard problem. At the same time, steps 3 and 4 involve an Eulerian path construction in de Bruijn graph, which can be solved in linear, in number of edges and nodes, time. Thus, the de Bruijn graph approach reduces the assembly problem to an algorithmically less complicated task. Moreover, finding read overlaps (step 1) is computationally very intensive.

In Section 6.2 we describe solutions based on the string graph concept, and in Section 6.3, we cover de Bruijn graph based approaches.

6.2 Overlap-Layout-Consensus

Historically, first solutions of the assembly problem followed OLC paradigm, which is more natural than de Bruijn graph approach. The OLC approach was introduced in [107] by Staden and then expanded by many scientists. OLC was very successful in processing reads generated by Sanger sequencing technologies. Many widely used assemblers implemented OLC approaches, such as Arachne [108], Celera Assembler [109], CAP3 [110], PCAP [111], Phrap [112], Phusion [113] and Newbler [114]. Celera [109] was used to assemble one of the first versions of the human genome.

The main difficulty of such an approach is to compute pairwise read overlaps. Given $n = |\mathcal{R}|$ (number of reads in the input set), a naive algorithm involves $O(n^2)$ overlap computations, which is infeasible in the case of large n even if one overlap computation is very fast. Then there are two major issues of these approach: reduction of number of computations and fast pairwise overlap search.

The basic case of overlap computations assumes that the reads are sequenced without errors, then exact suffix-prefix overlaps can be computed. However, in reality we need to find *approximate* overlaps, allowing some number of mismatches or errors. This means that overlap computation becomes a complicated problem itself.

Most of modern methods for computing approximate read overlaps implement the so-called *filtering* approach, when the search of reads which have overlaps with a given one is performed in two steps:

1. candidate reads are identified (filtering),
2. candidates are checked to verify the desired matching condition.

To reduce the number of overlap computations, Arachne [108] first extracts k-mers from reads, sort them and calculate overlaps only for those read pairs which share one or more k-mers (quite similar to FASTA [77] algorithm). Filtering algorithms, being often very efficient in practice, usually do not yield good theoretical time bounds. For example, *spaced seeds* (see Section 4.4.1 for the details) are often used in filtering based implementations (see, as example, [89]). Another assembler that follows the filtering approach is SGA assembler [115], which uses a basic substring filtering. Välimäki et al. [116] suggested to apply a modified version of suffix filters earlier proposed by Kärkkäinen and Na [117]. In [118], it was shown how this algorithm can be combinatorially improved to reach better time boundaries. Many OLC paradigm-based assemblers, including Celera [109], eliminate many read overlap computations skipping so-called transitive edges, when there are overlaps between reads A and B (suffix of A matches prefix of B), B and C, and A and C. Several methods, like CAP3 [110] and PCAP [111], combine (sub)set of reads into one big sequence with separators and use BLAST[78]-like techniques to calculate positions where a suffix or a prefix of a read can be found. These positions indicate the reads which potentially overlap with a given read.

6.3 De Bruijn graph

The idea of using de Bruijn graph for genome assembly goes back to the "pre-NGS era" [119]. Note, however, that *de novo* genome assembly is not the only application of those graphs when dealing with NGS data. There are several others, including: *de novo* transcriptome assembly [120] and *de novo* alternative splicing calling [121] from transcriptomic NGS data (RNA-seq); metagenome assembly [122] from metagenomic NGS data; and genomic variant detection [123] from genomic NGS data using a reference genome.

Due to a very large size of NGS datasets, it is essential to represent de Bruijn graphs as compactly as possible. This has been a very active line of research. Recently, several papers have been published that propose different approaches to compressing de Bruijn graphs [124, 125, 126, 127, 128].

Conway and Bromage [124] proposed a method based on classical succinct data structures, i.e. bitmaps with efficient rank/select operations. In the same direction, Bowe *et al.* [127] proposed a very interesting succinct representation that, assuming only one string (read) is present, uses only $4m$ bits, where m is the number of edges in the graph. A more realistic case, where there are M reads, can be easily reduced to the one string case by concatenating all M reads using a special separator character. However, in this case the size of the structure is $4m + O(M \log m)$ bits ([127], Theorem 1). While many methods are built upon Bloom filters, implementation from [127] is based on the Burrows-Wheeler transform.

Ye at al. [125] proposed a different method based on a sparse representation of de Bruijn graphs, where only a subset of k-mers present in the dataset are stored. Pell et al. [128] proposed a method to represent it approximately, using the so called *probabilistic de Bruijn graph*. In their representation, de Bruijn graph is represented using a Bloom filter B. This approach has the disadvantage of having false positive nodes, as direct consequence of the false positive queries in the Bloom filter, which can create false connections in the graph (see [128] for the influence of false positive nodes on the topology of the graph). The naive way to remove those false positive nodes, by explicitly storing (e.g. using a hash table) the set of all false positives of B, is clearly inefficient, as the expected number of elements to be explicitly stored is $4^k\mathcal{F}$, where $\mathcal{F}$ is the probability of false positive. Finally, Chikhi and Rizk [126] improved Pell's scheme in order to obtain an exact representation of the de Bruijn graph. Their approach is implemented in a software called MINIA, which is a part of GATB analysis tool [129]. This was, to our knowledge, the best *practical* representation of an exact de Bruijn graph based on Bloom filters.

One of the problems of de Bruijn graph based approaches is that k should be chosen in advance. This issue is solved by a *variable-order* de Bruijn graph. Boucher et al. [130] presented an improvement of [127] which allows changing the order of the graph "on the fly", while insignificantly increasing the space usage of the original representation. Supporting different values of k is especially useful when some regions of the graph are sparser than others. In [131] it was shown how to make the same structure *bidirectional*, allowing traversal in both directions.

Chikhi et al. [132] use minimizers to obtain BWT-based de Bruijn graph implementation with only 1.5 GB memory usage for the human genome. In [133] Belazzougui et al. show how to make de Bruijn graph implementation fully dynamic, allowing deletions along with insertions.

Chapter 7

De Bruijn graph representation using Cascading Bloom filters

7.1 Overview

In this chapter, we focus on the method proposed in [126] by Chikhi and Rizk which is based on Bloom filters (see Section 3.2) and implements de Bruijn graph approach (see Section 6.3). Let $\mathcal{R}$ be a set of reads and T_0 be the set of occurring k-mers (nodes of the de Bruijn graph) that we want to store. The key idea of [126] is to explicitly store only a subset of all false positives of B, the so-called *critical false positives*. This is possible because in order to perform an exact (without false positive nodes) graph traversal, only potential neighbors of nodes in T_0 are queried. In other words, the set of critical false positives consists of the potential neighbors of T_0 that are false positives of B, i.e. the k-mers from $U = \Sigma^k$ that overlap the k-mers from T_0 by $k-1$ letters and are false positives of B. Thus, the size of the set of critical false positives is bounded by $8|T_0|$, since each node of T_0 has at most $2|\Sigma| = 8$ neighbors (for each node, there are $|\Sigma|$ k-mers overlapping the $k-1$ suffix and $|\Sigma|$ overlapping the $k-1$ prefix). Therefore, the expected number of critical false positives can be upper-estimated by $8|T_0|\mathcal{F}$.

Our contribution is an improvement of this scheme by changing the representation of the set of false positives. We achieve this by iteratively applying a Bloom filter to represent the set of false positives, then the set of "false false positives" etc. We introduce the corresponding data structure, the so-called *Cascading Bloom filter*, and show how it can replace the standard Bloom filter in Section 7.2. Then in Section 7.3 we show analytically that this cascade of Bloom filters allows for a considerable further economy of memory, improving the method of [126]. Depending on the value of k, our method requires 30% to 40% less memory with respect to the method of [126]. Moreover, with our method, the memory grows very little as k grows. Finally, we implemented our method and tested it against [126] on real datasets. The tests, presented in Section 7.4, confirm the theoretical predictions for the size of structure and show a 20% to 30% *improvement* in query times. We conclude with Discussions in 7.5.

7.2 Cascading Bloom filter

As stated in Section 7.1, the method of [126] stores T_0 via a bitmap B_1 using a Bloom filter, together with the set T_1 of critical false positives. T_1 consists of those k-mers which have a $k-1$ overlap with k-mers from T_0 but which are stored in B_1 "by mistake", i.e.

belong[1] to B_1 but not to T_0. B_1 and T_1 are sufficient to represent the graph provided that the only queried k-mers are those which are potential neighbors of k-mers of T_0.

The idea we introduce in this chapter is to use this structure recursively and represent the set T_1 by a new bitmap B_2 and a new set T_2, then represent T_2 by B_3 and T_3, and so on. More formally, starting from B_1 and T_1 defined as above, we define a series of bitmaps $B_1, B_2, \ldots$ and a series of sets $T_1, T_2, \ldots$ as follows. B_2 stores the set of false positives T_1 using another Bloom filter, and the set T_2 contains the critical false positives of B_2, i.e. "true nodes" from T_0 that are stored in B_2 "by mistake" (we call them **false**[2] positives). B_3 and T_3, and, generally, B_i and T_i are defined similarly: B_i stores k-mers of T_{i-1} using a Bloom filter, and T_i contains k-mers stored in B_i "by mistake", i.e. those k-mers that do not belong to T_{i-1} but belong to T_{i-2} (we call them **false**^{i} positives). Observe that $T_0 \cap T_1 = \emptyset$, $T_0 \supseteq T_2 \supseteq T_4 \ldots$ and $T_1 \supseteq T_3 \supseteq T_5 \ldots$.

The following lemma shows that the construction is correct, that is it allows one to verify whether or not a given k-mer belongs to the set T_0.

Lemma 11. *Given a k-mer (node) K, consider the smallest i such that $K \notin B_{i+1}$ (if $K \notin B_1$, we define $i = 0$). Then, if i is odd, then $K \in T_0$, and if i is even (including 0), then $K \notin T_0$.*

Proof. Observe that $K \notin B_{i+1}$ implies $K \notin T_i$ by the basic property of Bloom filters that membership queries have one-sided error, i.e. there are no false negatives. We first check the Lemma for $i = 0, 1$.

For $i = 0$, we have $K \notin B_1$, and then $K \notin T_0$.

For $i = 1$, we have $K \in B_1$ but $K \notin B_2$. The latter implies that $K \notin T_1$, and then K must be a false[2] positive, that is $K \in T_0$. Note that here we use the fact that the only queried k-mers K are either nodes of T_0 or their neighbors in the graph (see [126]), and therefore if $K \in B_1$ and $K \notin T_0$ then $K \in T_1$.

For the general case $i \geq 2$, we show by induction that $K \in T_{i-1}$. Indeed, $K \in B_1 \cap \ldots \cap B_i$ implies $K \in T_{i-1} \cup T_i$ (which, again, is easily seen by induction), and $K \notin B_{i+1}$ implies $K \notin T_i$.

Since $T_{i-1} \subseteq T_0$ for odd i, and $T_{i-1} \subseteq T_1$ for even i (for $T_0 \cap T_1 = \emptyset$), the lemma follows. ∎

Naturally, the lemma provides an algorithm to check if a given k-mer K belongs to the graph: it suffices to check successively if it belongs to $B_1, B_2, \ldots$ until we encounter the first B_{i+1} which does not contain K. Then, the answer will simply depend on whether i is even or odd: K belongs to the graph if and only if i is odd.

In our reasoning so far, we assumed an infinite number of bitmaps B_i. Of course, in practice we cannot store infinitely many (and even simply many) bitmaps. Therefore, we "truncate" the construction at some step t and store a finite set of bitmaps $B_1, B_2, \ldots, B_t$ together with an explicit representation of T_t. The procedure of Lemma 11 is extended in the obvious way: if for all $1 \leq i \leq t$, $K \in B_i$, then the answer is determined by directly checking $K \in T_t$.

[1] By a slight abuse of notation, we also view B_j as the set of all k-mers on which the filter B_j returns the positive answer.

7.3 Analysis of the data structure

7.3.1 Memory and time usage

First, we estimate the memory needed by our data structure, under the assumption of an infinite number of bitmaps. Let N be the number of "true positives", i.e. nodes of T_0. As stated in Section 7.1, if T_0 has to be stored via a bitmap B_1 of size rN, the false positive rate can be estimated as c^r, where $c = 0.6185$. And, the expected number of critical false positive nodes (set T_1) has been estimated in [126] to be $8Nc^r$, as every node has eight extensions, i.e. potential neighbors in the graph. We slightly refine this estimation to $6Nc^r$ by noticing that for most of the graph nodes, two out of these eight extensions belong to T_0 (are real nodes) and thus only six are potential false positives. Furthermore, to store these $6Nc^r$ critical false positive nodes, we use a bitmap B_2 of size $6rNc^r$. Bitmap B_3 is used for storing nodes of T_0 which are stored in B_2 "by mistake" (set T_2). We estimate the number of these nodes as the fraction c^r (false positive rate of filter B_2) of N (size of T_0), that is Nc^r. Similarly, the number of nodes we need to put to B_4 is $6Nc^r$ multiplied by c^r, i.e. $6Nc^{2r}$. Keeping counting in this way, the memory needed for the whole structure is $rN + 6rNc^r + rNc^r + 6rNc^{2r} + rNc^{2r} + ...$ bits. The number of bits per k-mer is then

$$r + 6rc^r + rc^r + 6rc^{2r} + ... = (r + 6rc^r)(1 + c^r + c^{2r} + ...) = (1 + 6c^r)\frac{r}{1 - c^r}. \qquad (7.1)$$

A simple calculation shows that the minimum of this expression is achieved when $r = 5.464$, and then the minimum memory used per k-mer is 8.45 bits.

As mentioned earlier, in practice we store only a finite number of bitmaps $B_1, \ldots, B_t$ together with an explicit representation (such as array or hash table) of T_t. In this case, the memory taken by the bitmaps is a truncated sum $rN + 6rNc^r + rNc^r + ..$, and a data structure storing T_t takes either $2k \cdot Nc^{\lceil \frac{t}{2} \rceil r}$ or $2k \cdot 6Nc^{\lceil \frac{t}{2} \rceil r}$ bits, depending on whether t is even or odd. The latter follows from the observations that we need to store $Nc^{\lceil \frac{t}{2} \rceil r}$ (or $6rNc^{\lceil \frac{t}{2} \rceil r}$) k-mers, each taking $2k$ bits of memory. Consequently, we have to adjust the optimal value of r minimizing the total space, and re-estimate the resulting space spent on one k-mer.

Table 7.1 shows estimations for optimal values of r and the corresponding space per k-mer for $t = 4$ and $t = 6$, and several values of k. The data demonstrates that even such small values of t lead to considerable memory savings. It appears that the space per k-mer is very close to the "optimal" space (8.45 bits) obtained for the infinite number of filters. Table 7.1 reveals another advantage of our improvement: the number of bits per stored k-mer remains almost constant for different values of k.

The last column of Table 7.1 shows the memory usage of the original method of [126], obtained using the estimation $(1.44 \log_2(\frac{16k}{2.08}) + 2.08)$ the authors provided. Note that according to that estimation, doubling the value of k results in a memory increment by 1.44 bits, whereas in our method the increment is of 0.11 to 0.22 bits.

Let us now estimate preprocessing and query times for our scheme. If the value of t is small (such as $t = 4$, as in Table 7.1), the preprocessing time grows insignificantly in comparison to the original method of [126]. To construct each B_i, we need to store T_{i-2} (possibly on disk, if we want to save on the internal memory used by the algorithm) in order to compute those k-mers which are stored in B_{i-1} "by mistake". The preprocessing time increases little in comparison to the original method of [126], as the size of B_i decreases exponentially and then the time spent to construct the whole structure is linear on the size of T_0.

k	optimal r for $t = 4$	bits per k-mer for $t = 4$	optimal r for $t = 6$	bits per k-mer for $t = 6$	bits per k-mer for $t = 1$ ([126])
16	5.777	8.556	5.506	8.459	12.078
32	6.049	8.664	5.556	8.47	13.518
64	6.399	8.824	5.641	8.49	14.958
128	6.819	9.045	5.772	8.524	16.398

Table 7.1: 1st column: k-mer size; 2nd and 4th columns: optimal value of r for Bloom filters (bitmap size per number of stored elements) for $t = 4$ and $t = 6$ respectively; 3rd and 5th columns: the resulting space per k-mer (for $t = 4$ and $t = 6$); 6th column: space per k-mer for the method of [126] ($t = 1$)

The query time can be split in two parts: the time spent on querying t Bloom filters and the time spent on querying T_t. Clearly, using t Bloom filters instead of a single one introduces a multiplicative factor of t to the first part of the query time. On the other hand, the set T_t is generally much smaller than T_1, due to the above-mentioned exponential decrease. Depending on the data structure for storing T_t, the time saving in querying T_t vs. T_1 may even dominate the time loss in querying multiple Bloom filters. Our experimental results (Section 7.4 below) confirm that this situation does indeed occur in practice. Note that even in the case when querying T_t weakly depends on its size (e.g. when T_t is implemented by a hash table), the query time will not increase much, due to our choice of a small value for t, as discussed earlier.

7.3.2 Using different values of r for different filters

In the previous section, we assumed that each of our Bloom filters uses the same value of r, the ratio of bitmap size to the number of stored k-mers. However, formula (7.1) for the number of bits per k-mer shows a difference for odd and even filter indices. This suggests that using different parameters r for different filters, rather than the same for all filters, may reduce the space even further. If r_i denotes the corresponding ratio for filter B_i, then (7.1) should be rewritten to

$$r_1 + 6r_2c^{r_1} + r_3c^{r_2} + 6r_4c^{r_1+r_3} + ..., \tag{7.2}$$

and the minimum value of this expression becomes 7.93 (this value is achieved with $r_1 = 4.41; r_i = 1.44, i > 1$).

In the same way, we can use different values of r_i in the truncated case. This leads to a small 2% to 4% improvement in comparison with case of unique value of r. Table 7.2 shows results for the case $t = 4$ for different values of k.

7.3.3 Query distribution among filters

The query algorithm that follows from Lemma 11 simply queries Bloom filters $B_1, \ldots, B_t$ successively as long as the returned answer is positive. The query time then directly depends on the number of filters need to apply before getting a negative answer. Therefore, it is instructive to analyse how the query frequencies to different filters are distributed. We provide such an analysis in this section.

In our analysis, we assume that all true nodes of the graph and all their potential neighbors are queried with the same frequency. This is supported by the fact that during graph

k	r_1, r_2, r_3, r_4	bits per k-mer different values of r	bits per k-mer single value of r
16	5.254, 3.541, 4.981, 8.653	8.336	8.556
32	5.383, 3.899, 5.318, 9.108	8.404	8.664
64	5.572, 4.452, 5.681, 9.108	8.512	8.824
128	5.786, 5.108, 6.109, 9.109	8.669	9.045

Table 7.2: Estimated memory occupation for the case of different values of r vs. single value of r, for 4 Bloom filters ($t = 4$). Numbers in the second column represent values of r_i on which the minimum is achieved. For the case of single r, its value is shown in Table 7.1.

traversal, each time after querying a true node we also query all its potential neighbors, as there is no other mean to tell which of those neighbors are real. Note however that this assumption does not take into account structural properties of the de Bruin graph, nor any additional statistical properties of the genome (such as genomic word frequencies).

For a filter B_i, we want to estimate the number of k-mers for which B_i returns *no*. This number is the difference of the number of k-mers submitted to B_i and the number of k-mers for which B_i returns *yes*. Note that the k-mers submitted to B_i is precisely those on which the previous filter B_{i-1} returns *yes*.

If the input set T_0 contains N k-mers, we estimate by $7N$ the number of all tested k-mers, a typical node of the graph will have two true neighbors and six potential neighbors not belonging to the graph. Filter B_1 will return *yes* on $N + 6c^r N$ k-mers, corresponding to the number of true and false positives respectively. For an arbitrary i, filter B_i returns *yes* precisely on the k-mers inserted to B_i (i.e. k-mers B_i is built on), and the k-mers which are inserted to B_{i+1} (which are critical false positives for B_i). The counts then easily follow from the analysis of Section 7.3.1.

Table 7.3 provides counts for the first four filters, together with the estimated fraction of k-mers on which each filter returns the concluding answer (*no*).

	B_1	B_2	B_3	B_4
submitted k-mers	$7N$	$(1 + 6c^r)N$	$(6c^r + c^r)N$	$(c^r + 6c^{2r})N$
k-mers returning *yes*	$(1 + 6c^r)N$	$(6c^r + c^r)N$	$(c^r + 6c^{2r})N$	$(6c^{2r} + c^{2r})N$
k-mers returning *no*	$(7 - 1 - 6c^r)N$	$(1 - c^r)N$	$(6c^r - 6c^{2r})N$	$(c^r - c^{2r})N$
percentage of ending queries	79.5	13.25	5.76	0.96

Table 7.3: Theoretically counted number of queries returning 0 and 1 for bitmaps B_1, B_2, B_3, B_4 for r corresponding to the infinite bitmaps case.

As Table 7.3 shows, 99.48% of all queries end in first 4 bitmaps.

7.4 Experimental results

7.4.1 Construction algorithm

In practice, constructing a cascading Bloom filter for a real-life read set is a computationally intensive step. To perform it on a commonly-used computer, the implementation makes an essential use of external memory. Here we give a short description of the construction algorithm for up to four Bloom filters. Extension for larger number of filters is

strightforward.

We start from the input set T_0 of k-mers written on disk. We build the Bloom filter B_1 of appropriate size by inserting elements of T_0 successively. Next, all possible extensions of each k-mer in T_0 are queried against B_1, and those which return true are written to the disk. Then, this set is traversed and only the k-mers absent from T_0 are kept. This results in the set T_1 of critical false positives, which is also kept on disk. Up to this point, the procedure is identical to that of [126].

Next, we insert all k-mers from T_1 into B_2 and to obtain T_2, we check for each k-mer in T_0 if a query to B_2 returns true. This results in set T_2. Thus, at this point we have B_1, B_2 and T_2, a complete representation for $t = 2$. In order to build the data structure for $t = 4$, we continue this process, by inserting T_2 in B_3 and retrieving T_3 from T_1 (stored on disk). It should be noted that to obtain T_i we need T_{i-2}, and by always storing it on disk we guarantee not to use more memory than the size of the final structure. The set T_t (that is, T_1, T_2 or T_4 in our experiments) is stored as a sorted array and is searched by a binary search. We found this implementation more efficient than a hash table.

7.4.2 Implementation and experimental setup

We implemented our method using the MINIA software [126] and ran comparative tests for 2 and 4 Bloom filters ($t = 2, 4$). Note that since the only modified part of MINIA was the construction step and the k-mer membership queries, this allows us to precisely evaluate our method against the one of [126].

The first step of the implementation is to retrieve the list of k-mers that appear more than d times using DSK [134] – a constant memory streaming algorithm to count k-mers. Note, as a side remark, that performing counting allows us to perform off-line deletions of k-mers. That is, if at some point of the scan of the input set of k-mers (or reads) some of them should be deleted, it is done by a simple decrement of the counter.

Assessing the query time is done through the procedure of graph traversal, as it is implemented in [126]. Since the procedure is identical and independent on the data structure, the time spent on graph traversal is a faithful estimator of the query time.

We compare three versions: $t = 1$ (i.e. the version of [126]), $t = 2$ and $t = 4$. For convenience, we define 1 Bloom, 2 Bloom and 4 Bloom as the versions with $t = 1, 2$ and 4, respectively.

7.4.3 *E.coli* dataset, varying k

In this set of tests, our main goal was to evaluate the influence of the k-mer size on principal parameters: size of the whole data structure, size of the set T_t, graph traversal time, and time of construction of the data structure. We retrieved 10M *E. coli* reads of 100bp from the *Short Read Archive* (ERX008638) without read pairing information and extracted all k-mers occurring at least two times. The total number of k-mers considered varied, depending on the value of k, from 6,967,781 ($k = 15$) to 5,923,501 ($k = 63$). We ran each version, 1 Bloom ([126]), 2 Bloom and 4 Bloom, for values of k ranging from 16 to 64. The results are shown in Fig. 7.1.

The total size of the structures in bits per stored k-mer, i.e. the size of B_1 and T_1 (respectively, B_1, B_2,T_2 or B_1, B_2, B_3, B_4,T_4) is shown in Fig. 7.1(a). As expected, the space for 4 Bloom filters is the smallest for all values of k considered, showing a considerable improvement, ranging from 32% to 39%, over the version of [126]. Even the version with just 2 Bloom filters shows an improvement of at least 20% over [126], for all values of k. Regarding the influence of the k-mer size on the structure size, we observe that for 4 Bloom

filters the structure size is almost constant, the minimum value is 8.60 and the largest is 8.89, an increase of only 3%. For 1 and 2 Bloom the same pattern is seen: a plateau from $k = 16$ to 32, a jump for $k = 33$ and another plateau from $k = 33$ to 64. The jump at $k = 32$ is due to switching from 64-bit to 128-bit representation of k-mers in the table T_t.

The traversal times for each version is shown in Fig. 7.1(c). The fastest version is 4 Bloom, showing an improvement over [126] of 18% to 30%, followed by 2 Bloom. This result is surprising and may seem counter-intuitive, as we have four filters to apply to the queried k-mer rather than a single filter as in [126]. However, the size of T_4 (or even T_2) is much smaller than T_1, as the size of T_i's decreases exponentially. As T_t is stored in an array, the time economy in searching T_4 (or T_2) compared to T_1 dominates the time lost on querying additional Bloom filters, which explains the overall gain in query time.

As far as the construction time is concerned (Fig. 7.1(d)), our versions yielded also a faster construction, with the 4 Bloom version being 5% to 22% faster than that of [126]. The gain is explained by the time required for sorting the array storing T_t, which is much higher for T_0 than for T_2 or T_4. However, the gain is less significant here, and, on the other hand, was not observed for bigger datasets (see Section 7.4.5).

7.4.4 *E. coli* dataset, varying coverage

From the complete *E. coli* dataset (≈44M reads) from the previous section, we selected several samples ranging from 5M to 40M reads in order to assess the impact of the coverage on the size of the data structures. This strain *E. coli* (K-12 MG1655) is estimated to have a genome of 4.6M bp [135], implying that a sample of 5M reads (of 100bp) corresponds to ≈100X coverage. We set $d = 3$ and $k = 27$. The results are shown in Fig. 7.2. As expected, the memory consumption per k-mer remains almost constant for increasing coverage, with a slight decrease for 2 and 4 Bloom. The best results are obtained with the 4 Bloom version, an improvement of 33% over the 1 Bloom version of [126]. On the other hand, the number of distinct k-mers increases markedly (around 10% for each 5M reads) with increasing coverage, see Fig. 7.2(b). This is due to sequencing errors: an increase in coverage implies more errors with higher coverage, which are not removed by our cutoff $d = 3$. This suggests that the value of d should be chosen according to the coverage of the sample. Moreover, in the case where read qualities are available, a quality control pre-processing step may help to reduce the number of sequencing errors.

7.4.5 Human dataset

We also compared 2 and 4 Bloom versions with the 1 Bloom version of [126] on a large dataset. For that, we retrieved 564M Human reads of 100bp (SRA: SRX016231) without pairing information and discarded the reads occurring less than 3 times. The dataset corresponds to ≈17X coverage. A total of 2,455,753,508 k-mers were indexed. We ran each version, 1 Bloom ([126]), 2 Bloom and 4 Bloom with $k = 23$. The results are shown in Table 7.4.

The results are in general consistent with the previous tests on *E.coli* datasets. There is an improvement of 34% (21%) for the 4 Bloom (2 Bloom) in the size of the structure. The graph traversal is also 26% faster in the 4 Bloom version. However, in contrast to the previous results, the graph construction time increased by 10% and 7% for 4 and 2 Bloom versions respectively, when compared to the 1 Bloom version. This is due to the fact that disk *i/o* operations now dominate the time for the graph construction, and 2 and 4 Bloom versions generate more disk accesses than 1 Bloom. As stated in Section 7.4.1, when constructing the 1 Bloom structure, the only part written on the disk is T_1 and it

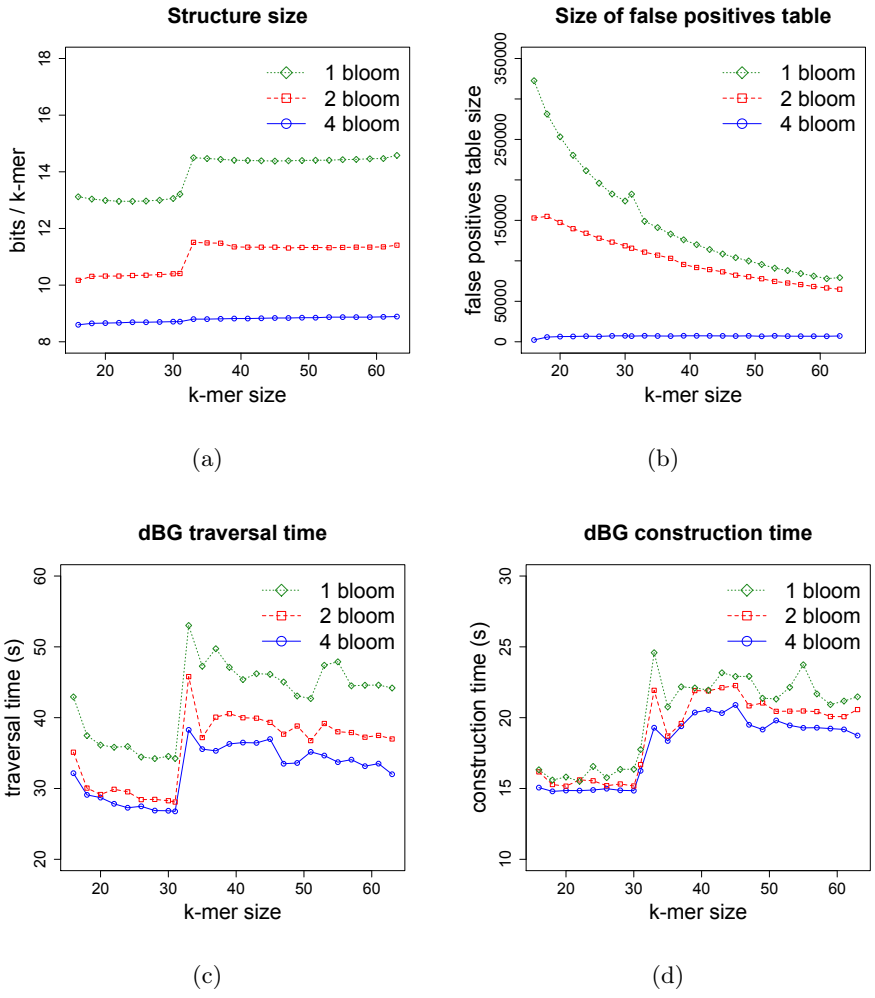
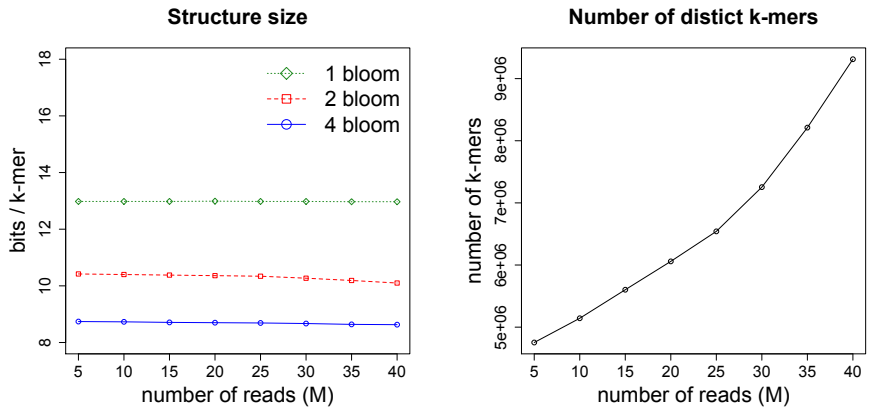


Figure 7.1: Results for 10M E.coli reads of 100bp using several values of k. The *1 Bloom* version corresponds to the one presented in [126]. (a) Size of the structure in bits used per k-mer stored. (b) Number of false positives stored in T_1, T_2 or T_4 for 1, 2 or 4 Bloom filters, respectively. (c) De Bruijn graph construction time, excluding k-mer counting step. (d) De Bruijn graph traversal time, including branching k-mer indexing.



(a) (b)

Figure 7.2: Results for *E.coli* reads of 100bp using $k = 27$. The *1 Bloom* version corresponds to the one presented in [126]. (a) Size of the structure in bits used per k-mer stored. (b) Number of distinct k-mers.

is read only once to fill an array in memory. For 4 Bloom, T_1 and T_2 are written to the disk, and T_0 and T_1 are read at least one time each to build B_2 and B_3. Moreover, since the size coefficient of B_1 reduces, from $r = 11.10$ in 1 Bloom to $r = 5.97$ in 4 Bloom, the number of false positives in T_1 increases.

Method	1 Bloom	2 Bloom	4 Bloom
Construction time (s)	40160.7	43362.8	44300.7
Traversal time (s)	46596.5	35909.3	34177.2
r coefficient	11.10	7.80	5.97
Bloom filters size (MB)	$B_1 = 3250.95$	$B_1 = 2283.64$ $B_2 = 323.08$	$B_1 = 1749.04$ $B_2 = 591.57$ $B_3 = 100.56$ $B_4 = 34.01$
False positive table size (MB)	$T_1 = 545.94$	$T_2 = 425.74$	$T_4 = 36.62$
Total size (MB)	3796.89	3032.46	2511.8
Size (bits/k-mer)	**12.96**	**10.35**	**8.58**

Table 7.4: Results of 1, 2 and 4 Bloom filters version for 564M Human reads of 100bp using $k = 23$. The *1 Bloom* version corresponds to the one presented in [126].

7.5 Discussion

Using cascading Bloom filters for storing de Bruijn graphs brings a clear advantage over the single-filter method of [126]. In terms of memory consumption, which is the main parameter here, we obtained an improvement of around 30%-40% in all our experiments. Our data structure takes 8.5 to 9 bits per stored k-mer, compared to 13 to 15 bits by the method of [126]. This confirms our analytical estimations. The above results were obtained using only four filters and are very close to the estimated optimum (around 8.4

bits/k-mer) produced by the infinite number of filters. An interesting characteristic of our method is that the memory grows insignificantly with the growth of k, even slower than with the method of [126]. Somewhat surprisingly, we also obtained a significant decrease, of order 20%-30%, of query time. The construction time of the data structure varied from being 10% slower (for the human dataset) to 22% faster (for the bacterial dataset).

An interesting prospect for further possible improvements of our method is offered by work [136], where an efficient replacement to Bloom filter was introduced. The results of [136] suggest that we could hope to reduce the memory to about 5 bits per k-mer. However, there exist obstacles on this way: an implementation of such a structure would probably result in a significant construction and query time increase.

Chapter 8

Improved compression of DNA sequencing data with Cascading Bloom filters

8.1 Overview

Rozov et al. in [137] showed how Cascading Bloom filter, which we designed to solve the genome assembly problem, can be applied to the read compression problem, introduced below. In this chapter, we first show how Cascading Bloom filter is applied in [137] to the lossless reference-free compression of read sets (Section 8.2). Then we present an improvement of this technique in Section 8.3. Through computational experiments on real data, we demonstrate that our improvement results in a significant associated memory reduction in practice in Section 8.4. We end with Discussion in Section 8.5.

The disk space used for storing NGS data can easily reach tens or even hundreds of GBs. The time needed for transmission of this data between different servers via Internet may cause significant delay of the experiments. Thus, problems of compact storage and fast transmission of read sets are becoming more and more significant nowadays and are providing a motivation to develop different methods and algorithms.

All compression tools can be compared in terms of two important characteristics: compression ratio (i.e., the ratio of compressed file size to the original file size), and reads set decoding/encoding time. Specialized methods that take into account the fact that reads were obtained from genomes with mutations and errors turn out to provide better compression ratio and speed performance than general tools like gzip.

All specialized methods can be classified into *reference-based*, which typically align reads on a reference sequence [138, 139, 140, 141, 142], and *reference-free* [138, 140, 143, 144, 145, 146, 147, 148, 149].

To compress reads, reference-based methods first align the reads to the reference genome in order to find the best alignment position. Typically, such alignment-based compression tools allow only for few errors per read to be efficient. Information about positions and differences between reads and the reference genome is then effectively encoded and compressed using general compression tools. Although such approaches achieve good compression ratios, they take significantly more time in comparison to reference-free methods as they require a time-consuming mapping procedure on the reference genome.

Reference-free methods implement various techniques. Some of them ([144, 147]) reorder reads in order to group similar reads together and then use general purpose compressing tools. This reordering can improve the compression ratio due to local similarity

of reads. Another popular approach is to first assemble reads into contigs, and then use assembled contigs as a reference. Such methods benefit from some advantages of reference-based approaches and do not need a reference genome to decompress the input set of reads.

A detailed comparison of many existing NGS reads compression tools can be found in [138].

Paper [137] proposes a method for compressing a set of reads of fixed length ℓ, based on the idea of extracting reads from a reference genome instead of storing them explicitly. The method from [137] achieves a good trade-off between speed and compression ratio, where the gain in speed is due to avoiding the mapping process and the gain in compression is due to the use of a Cascading Bloom filter. Below we improve results of [137] by applying the recursive compression strategy to other subsets of the reads.

8.2 Description of the basic algorithm

In this section we briefly describe encoding and decoding processes proposed in [137].

Let G be a genome of length N and S be a set of reads, sequenced from G (with errors and mutations). All reads from S should be of the same length, denoted as ℓ.

For simplicity, let us assume that all reads are unique, and all of them consists of characters $[A, C, G, T]$ only. First, the algorithm adds all reads from S to a Bloom filter B. Then we query all substrings of G of length ℓ against B to identify all reads that potentially can be reconstructed from G. As some of queried substrings are *false positives*, we add such substrings (which are accepted by B, but do not belong to S) to the set of false positives FP. As the set S was sequenced from the genome of a species that contains mutations in comparison to genome G, then some reads from S are not covered by substrings of G. Moreover, sequencing errors also lead to reads that can not be reconstructed from G. Such reads, that can not be obtained as a substring of G, are added to the set of *false negatives* FN. Finally, set S is encoded by B, set of false positives FP and set of false negatives FN. They can be further compressed by general purpose compressing tools.

The encoding process is described in Algorithm 3.

Algorithm 3 Encoding

```
add all repeated reads and reads containing ambiguous characters to FN
S = S \ FN
add all reads from S to Bloom filter B
P = ∅
for (r ∈ set of all substrings of length ℓ from G) do
    if r ∈ B then
        if (r ∈ S) then
            P = P ∪ {r}
        else
            FP = FP ∪ {r}
        end if
    end if
end for
FN = FN ∪ (S \ P)
```

Decoding works in the same way as encoding. First, we scan all substrings of G of length ℓ, query them against B and check whether they are present in FP. If the substring is in

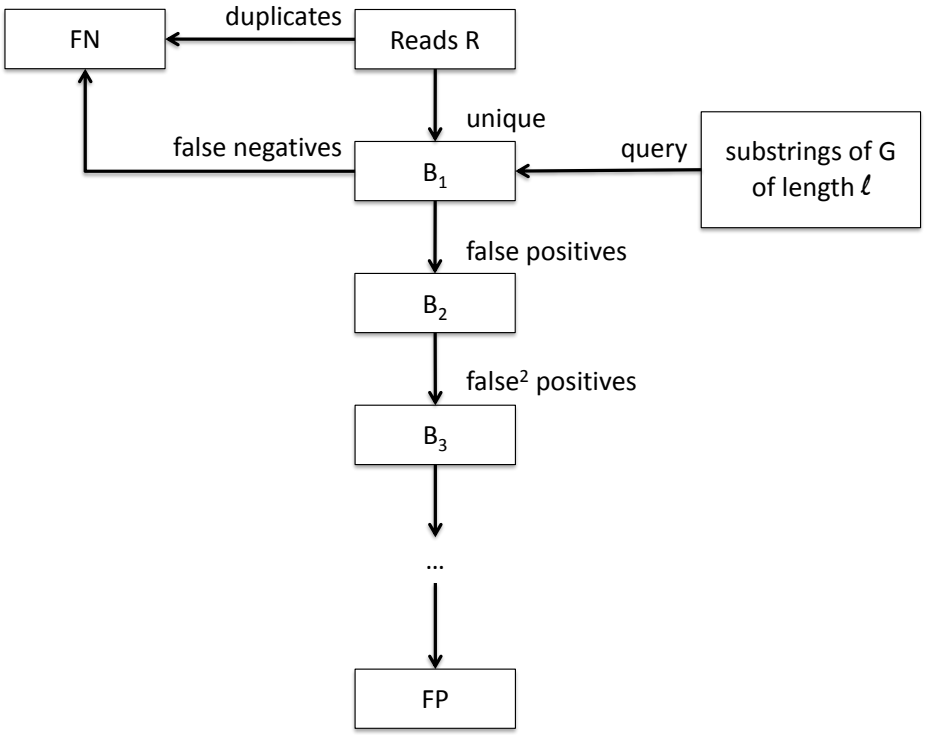


Figure 8.1: Encoding of read set S with a reference genome G.

B, but not in FP, then we add it to the set $S_{encoded}$. Finally, we add all reads from FN to $S_{encoded}$.

To relax condition on repeats and ambiguous characters (non-$[A, C, G, T]$ characters, like N) in S, we simply add all such reads to (now multiset) FN. The encoding and decoding processes remains the same.

The algorithm above does not turn out to achieve a good compression ratio. Partially, this is because the set FP is comparatively large. To reduce its size, we store FP in a Cascading Bloom filter, as it was suggested in [2] and Chapter 7.

Bloom filter B (denoted also as B_1) can be considered as the first Bloom filter in the cascade. Each element from FP (denoted also as FP_1) is added to Bloom filter B_2 – it stores false positive reads, and thus it should reject true reads. It means that all unique reads without ambiguous characters from S, that are in B_1 (denote this set as S'), should be rejected by B_2. So, we query all reads from S' against B_2 to obtain FP_2 – *false false positive* reads. This procedure can be continued infinitely, but in practice we should stop after fixed number of filters. If n Bloom filters are in the cascade, then S is encoded by $B_1, B_2, \ldots B_n, FP_n, FN$.

The described encoding algorithm is schematically shown on Figure 8.1.

To decode the reads, we query all substrings of G of length ℓ against B_1, $B_2 \ldots$ until one of them rejects the substring. Let B_i be the first Bloom filter that rejected a substring r. If i is even, then r is a true read (as $B_1, B_3, B_5 \ldots$ store true reads, and $B_2, B_4, B_6 \ldots$ store false positives), and is a false positive read otherwise. More detailed proof can be found in Chapter 7 and in [137, 2].

Since elements inserted to B_i are a subset of B_{i-2}, then the size of Bloom filters decreases exponentially. The exact formula for the size of Bloom filters (in other words, for number of bits used for one read) can be found in [137].

8.3 Improvements

We propose to save on storing FN too. FN consists of two disjoint sets: set FN_0 of reads from S which are not substrings of G and a (multi-)set FN_{rep} of reads which are substrings of G and occur in S multiple times (more than once). For FN_0, there is no way to store it compactly in the described paradigm, as reads from this set can not be easily extracted from G. We propose to compress set FN_{rep} as follows.

First, let us represent FN_{rep} as a set T of pairs $(read, multiplicity)$. As for each element of T has multiplicity at least 2, then we can decrease all multiplicities by 1, and this procedure is reversible. Then we partition reads from T into two subsets: a subset FN^1_{rep} of reads that have multiplicity 1 and a subset $FN^{\geqslant 2}_{rep}$ of reads that have multiplicity at least 2. Since elements of FN^1_{rep} have no duplicates, FN^1_{rep} can again be compressed by applying Cascading Bloom filter described in Section 8.2. Depending on the size of $FN^{\geqslant 2}_{rep}$, this procedure may be iteratively applied to $FN^{\geqslant 2}_{rep}$ again. However, in our experiments described below we didn't do that, as the size of $FN^{\geqslant 2}_{rep}$ (reads in S with multiplicity at least 3 occurring in G exactly) appeared to be small.

Encoding of FN_{rep} is shown in Algorithm 4.

The decoding of the initial set FN_{rep} is straightforward. First, set FN^1_{rep} is reconstructed applying algorithm described in Section 8.2. Then sets FN^1_{rep} and $FN^{\geqslant 2}_{rep}$ are merged, and multiplicity of each read is incremented by 1. The obtained set joint with FN_0 finish the reconstruction of FN. Then full initial set S can be decoded as it was described in Section 8.2.

Algorithm 4 *FN* encoding

1: FN_{rep} = reads from FN that are substrings of G and occur in S multiple times
2: FN_{rep} = FN_{rep} with decreased multiplicities by one
3: FN^{1}_{rep} = reads from FN_{rep} having multiplicity one
4: $FN^{\geqslant 2}_{rep}$ = reads from FN_{rep} having multiplicity two or more
5: compress FN^{1}_{rep} using Algorithm 3
6: repeat lines 2-6 for $FN^{\geqslant 2}_{rep}$

As a remark, note that the set of reads repeated multiple time (FN_{rep}) depends only on G and S, and this set is larger when the coverage of G by the read set S is higher. Moreover, multiple reads are more likely to occur in G exactly, as the probability that such a read was sequenced with errors is lower. This supports the idea that compressing FN_{rep} can be advantageous.

8.4 Experiments

We modified the code of BARCODE[1] software [137] to implement the above extension. We experimented with *Mycobacterium abscessus* complete genome (about 5Mbp). We tried the following coverage values $30, 50, 75, 100, 150$. Single-end reads of length 100 with 0.000 to 0.005 base mutation rate were generated using DWGSIM[2] simulator (options DWGSIM -C [30-150] -H -e 0.0-0.005 -R 0.0 -1 100 -2 0 -y 0.0). The extensive experimental comparison with other compression tools is provided in [137], and our goal was to simply estimate the influence of ideas described in Section 8.3. We didn't apply the post-compression with SCALCE, as our goal was limited to compare the size of compressed with BARCODE FN_{rep} with uncompressed FN_{rep}.

For coverage values 30 and 50, the fraction of repeated reads (set FN_{rep}) was relatively small, with the coverage by FN_{rep} smaller than four. For example, for coverage 50, the coverage by FN_{rep} is 3.5, the size of FN_{rep} is 17MB, and total size of Bloom filters, false positive and false negative sets for FN_{rep} is 16MB. Therefore, in this case, the additional compression is insignificant compared to the size of FN_{rep}.

For coverage values between 75 and 150, which are common, for example, for bacterial sequencing data, we observed a significant additional compression of FN_{rep}. Results are shown in Table 8.1. For both versions of compression (initial BARCODE and the one with our improvements) we provide the total sum of all data structure sizes to give some estimations on the total size, even though it is not correct to directly sum their sizes, as they can be compressed afterwards with different compression ratios.

The three of four last rows show sets that replace FN_{rep}. Sets of false positives for filters storing FN_{rep} turned out to be insignificantly small.

First of all, as it was theoretically predicted, coverage of FN set is growing when coverage of S increases. Secondly, Table 8.1 shows that the larger FN_{rep} is, the more significant improvement is. Finally, using Cascading Bloom filters for compressing FN_{rep} set seems to be profitable for examined values of coverage.

[1] available at http://www.cs.tau.ac.il/~heran/cozygene/software.shtml
[2] https://github.com/nh13/DWGSIM

Table 8.1: Sizes of different data structures for coverage values 75, 100 and 150. Size of FN_{rep} is shown after removing one copy of each read.

Coverage	75X	100X	150X
Set of reads S (M)	3.8	5.1	7.7
Bloom filters $\{BF_i\}$ (MB)	4	4	2.5
FP (MB)	0.9	6	90
FN (MB)	181	268	466
FN_0 (MB)	105	138	200
FN_{rep} (MB)	40	69	146
Total size (MB)	149.9	217	438.5
Bloom filters for FN_{rep} (MB)	11	14	6
False positives for FN_{rep} (MB)	0	0	0
False negatives for FN_{rep} (MB)	10	21	57
Total size of improved structure (MB)	130.9 (87%)	183 (84%)	355.5 (81%)

8.5 Discussion

Rozov et al. in [137] showed that Cascading Bloom filters can be effectively applied to read compression problem. Interestingly, without utilizing a Cascading Bloom filter the presented technique does not show a good performance. Thus, there can also be another applications where Cascading Bloom filter can be a more efficient replacement of a standard Bloom filter.

In this chapter, we showed that the read compression method using Cascading Bloom filter can be made even more efficient, if we use the compression strategy recursively on the set of false negative reads. We experimentally demonstrated that compression of read sets with coverage values between 75 and 150 can be improved using this method. Finally, coverage values like 150 and higher possibly can benefit from using more than one filter in the cascade for FN_{rep}.

Part IV

Metagenomic classification

Contents - Part IV

Chapter 9

Algorithmic methods for metagenomic classification

Analysis of multiple genomes, "living" in some environment, is usually called metagenomics or community genomics. It became a popular and booming field of research in bioinformatics in recent years. In this work we concentrate on the problem of metagenomic classification with a given taxonomic tree, which represents the evolutionary relationships between various biological species based on their similarities and differences in their genetic characteristics. Thus, the goal of such classification is, for every given read, to find its origin in this tree.

Metagenomic classification can be regarded as a read alignment problem, with the only difference that sequences should be mapped to many genomes, not to one. Thus different solutions for read alignment problem can be applied (such methods, applied to metagenomic classification, are called *alignment-based*). Several specialized tools, like MEGAN [150], MetaPhlAn [151], MetaPhyler [152], follow this approach, but they are not able to deal with metagenomic datasets of thousands and tens of thousands reference genomes, whose total length can be of billions and tens of billions base pairs. Using even the most efficient applications for NGS reads mapping as BWA [97], Novoalign[1], Bowtie [93] and many others can not provide sufficient speed for classification. One more obstacle is that read mapping tools usually find only alignments with a score close to the best alignment score and can not identify occurrences with lower quality.

Consequently, novel methods that avoid read alignment for classification should be developed. Such methods are called *alignment-free* and were actively studied in recent years. The majority of such methods are based on the analysis of shared words between reference genomes and sequences. One approach can be to analyse the *frequency vectors*, which is an array for every word containing the number of its occurrences in the sequence. However, such an approach usually requires too much memory and time to construct vectors and to compare them, thus in recent applications only the existence of every word (usually of fixed length, k-mer) in a string is taken into consideration, without information about the number of occurrences. This approach is shown schematically on Figure 9.1.

Generally, all k-mer based approaches implement the following steps:

1. extract k-mers from reference genomes and construct an index for them,

2. extract k-mers from reads one by one and query them against the index,

[1] http://www.novocraft.com/

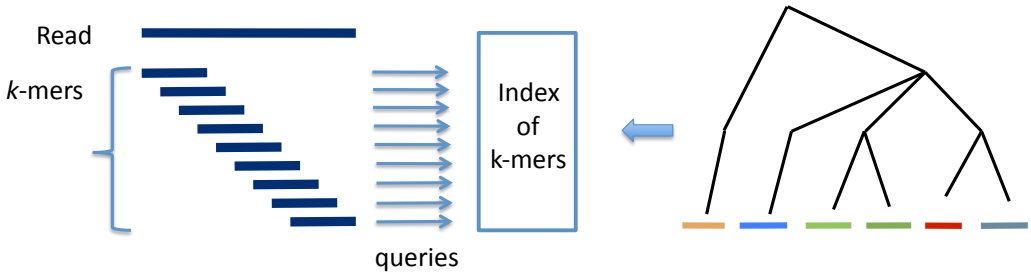


Figure 9.1: Extraction of k-mers and construction of an index.

3. based on matched nodes for all k-mers from the read, assign the read to a node in the taxonomic tree (or leave it unassigned).

Here we briefly list the most popular algorithmic techniques and metagenomic classification tools.

Although extracting k-mers and indexing them is the dominant basic method used in practice, there are some other algorithmic ideas worth to mention. One of them is based on the read assembly into longer contigs [153]. This method can significantly reduce the number of queried sequences, but it is sensitive to errors of assembly which should be done very carefully. At the same time, assembly step can be very expensive in terms of time and memory. Another possible improvement is to build an index not for all reference sequences, but for most informative regions only [154, 24, 151]. The problem of such an approach could be that it misses a big part of informative reads, leaving them unclassified.

The first popular k-mer based metagenomic classification tool was LMAT [155]. For every k-mer it stores the list of all nodes in the taxonomic tree that contain this k-mer. This correspondence is stored in a hash table with k-mers as keys and lists of node ids as values. Then, after find matching nodes for all k-mers from the read, the read itself is assigned to a "best matching" node in the tree. Optionally, LMAT can construct a reduced database for most informative k-mers only. The biggest problem of this tool is its huge memory requirements (hundreds of GBs for a bacterial genome database [156]).

Kraken [157] is, to our knowledge, the most widely used classification tool nowadays. It implements an approach similar to LMAT. For every k-mer in reference genomes, only the LCA (*lowest common ancestor*) of nodes containing the k-mer is stored in a hash table. Kraken provides a very fast classification (partially due to storing k-mers with the same minimizer in the same place in memory), but requires huge memory (more than a hundred GBs for index construction and an index of size 70 GBs for the bacterial database mentioned above). Storing LCAs instead of full information about matching k-mers leads to one more drawback of Kraken, its *lossy* nature.

Both Kraken and LMAT use hash tables for storing a k-mers set. As it was discussed in Chapter 4, being extremely efficient in terms of query time (with appropriate implementation), such approaches have a very big memory footprint. To overcome this problem, recently many tools started to utilize BWT index to store k-mers.

Centrifuge [158] is a BWT index-based classifier. After binarizing the taxonomic tree, for each pair of nodes it propagates similar subsequences up along edges of the taxonomic tree. Then it creates a BWT index (implementation from Bowtie2 [159]). Reportedly, this

leads to a very small memory usage, namely 4 GB index for a database of 4,300 prokaryotic genomes. However, similar to Kraken, some information is lost during merge of similar sequences step.

Kaiju [160] is a protein level classifier based on a BWT index. Kaiju algorithm translates input reads in six possible reading frames and searches for maximum exact matches (MEMs) of amino acid sequences in a given database of annotated proteins from microbial reference genomes. Kaiju outputs the node in the tree if a match is found (the LCA of nodes if there are many equally good matches). The memory footprint of Kaiju is only 6 GB for the default database of bacterial, archaeal, and viral protein sequences.

Another idea of storing k-mers is to hash them in Bloom filter 3.2. This approach is implemented in FACS [161] and CoMeta [162] classifiers. Both of them utilize only k-mer presence in reference genomes and do not take into consideration their frequencies. This approach also leads to large memory requirements (around 40 GB for bacterial database and $k = 30$ for CoMeta).

A more detailed comparison of different metagenomic classification tools can be found in multiple papers [163, 164, 165, 166, 167, 168].

Chapter 10

Data structures for BWT-index-based metagenomic classification

10.1 Overview

In this chapter, we study the problem of metagenomic classification that was defined in Section 9. We follow the *alignment-free* approach (see Section 9 for the details): we extract all k-mers from genomes and store them in a data structure (*index*). Then, for every read we extract all k-mers from it and query them against the index. Based on the information about k-mers found in the index, the read is assigned to a node in the taxonomic tree.

We present data structures that we use in our metagenomic classification tool called *ProPhyle*. We decided to create ProPhyle on the basis of a full-text search index, as such indices occupy a small amount of memory and allow for comparatively fast queries, at the same time providing a possibility to store full information about k-mer presence in the reference sequences. We are mainly focusing on algorithmic methods that allow ProPhyle to be fast and memory-efficient. In Sections 10.2 and 10.3 we describe the index construction process and provide query details. In Sections 10.4 and 10.5 we present data structures and methods that improve ProPhyle's query speed and memory usage correspondingly. Then we provide experimental results in Section 10.6 and conclusions in Section 10.7.

10.2 Index construction

As an input, we are given a taxonomic tree and a set of genomes, each of them corresponding to some leaf in the tree. Our goal is to design a data structure that will allow fast queries of k-mers returning the list of genomes (i.e., nodes in the taxonomic tree) where a k-mer appears.

The index construction consists of 5 steps:

- extracting k-mers from reference genomes,
- k-mers propagation along the edges of the taxonomic tree,
- assembly of k-mers at each node into longer contigs,
- merging all contings for all nodes into one file,

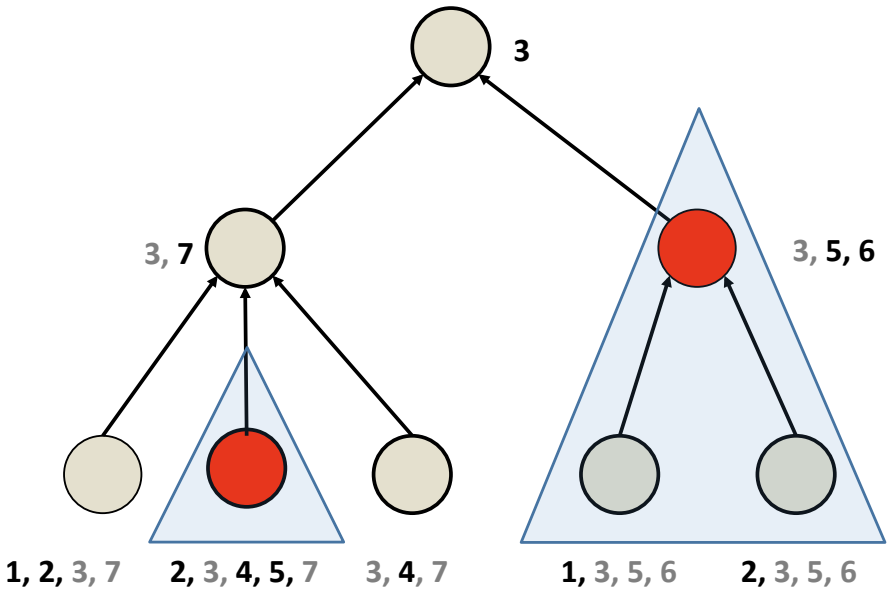


Figure 10.1: Propagation of k-mers. k-mers in transparent font are propagated to the parent. k-mer forest is shown for $K = 5$.

- constructing a BWT index (see Section 3.5 for the details) with some additional data structures.

Below we describe each of the steps above in more detail.

Extracting k-mers from reference genomes For each leaf node in the taxonomic tree (with a corresponding reference genome) we extract all k-mers and remove duplicates. We do not add reverse complements of k-mers, as they will be processed during the next steps.

k-mers propagation This step is one of the key ideas of the whole algorithm. Usually, reference genomes share a huge number of k-mers, especially if there are many close relatives in the database under study. This leads to the idea of "propagation" of a k-mer K to the parent node if all its children contain K. In other words, all shared k-mers are propagated to parent nodes, and the total number of k-mers in all nodes decreases. An example of the propagation is shown in Figure 10.1. Here we come to the concept of *k-mer forest*: instead of storing a k-mer K in all nodes where reference genome contains K, we store only roots of subtrees where all leaves contain K. In Figure 10.1, we can see an example of k-mer forest for $K = 5$. This allows us to save on storing k-mers that occur in genomes of related species.

Assembly of k-mers After k-mer propagation, we aim to compress k-mer sets at every node. In order to do this we assemble k-mers into longer contigs, without loosing or adding any information, using the *de Bruijn graph* concept (see Section 6.3): two k-mers are merged into one $k + 1$-mer if and only if they have a suffix-prefix overlap of length

$k-1$. The following simple greedy algorithm is used: the algorithm randomly chooses a k-mer to start with and then extends it in both directions, every time choosing a random k-mer with a suffx-prefix overlap of length $k-1$ if there are multiple choices.

Merging contings for all nodes into one FASTA file This step is straightforward: one big FASTA file containing all contigs for all nodes is created. Contigs for one node are put together in this file.

Constructing a BWT index with some additional data structures A FASTA file, described in Paragraph 10.2, is used as an input for a BWT index construction. We use BWA as BWT index implementation (see more details about the implementation in Section 10.6).

A detailed explanation of these construction steps can be found in [61].

10.3 Index query

Suppose we have a BWT index P constructed for a text T as described in Section 10.2 and a read R that we want to classify. Following the strategy, described in Section 9, we extract all k-mers from R and query them independently against index P. We call this type of search "restarted", as we query all k-mers independently and do not use the fact that two neighbouring k-mers share $(k-1)$-mer.

Here we describe how to query a k-mer K against index P. As P is, in fact, a BWT index, then the general query process consists of two steps:

- find a Suffix Array (or SA) interval (s, e) corresponding to K,
- translate every position from (s, e) to a position in the initial text.

In our case, we actually do not need exact positions in the text, but only the name of the node in the taxonomic tree. BWA provides a translation from the text position to the contig, so we need to add one more step to BWA to obtain the corresponding node in the taxonomic tree.

If a k-mer appears in the text, then the SA interval search works in $O(k)$ time (see Section 3.5 for more details). Thus, the search of SA intervals for all k-mers in the read can be completed in $O(\ell k)$ time, where $\ell = |R| \gg k$. The second step of the query process is usually implemented using a *sampled suffix array*: if a current position in SA is sampled, then there is an explicit translation to the text position; if it is not sampled, then we move to the suffix array position corresponding to the previous position in the text, and repeat this operation until the current position is sampled. Finally, the answer will be the found text position shifted by the number of steps back.

It is also important that both parts of the query process require a significant time if the k-mer appears in the text.

We found two inefficient parts in the reasoning above:

- the restarted search of k-mers is not efficient as we do not use the information that two consecutive k-mers share $k-1$-mer,
- we do not need exact positions in the text where a k-mer appears, therefore we can reduce information that has to be stored in the index.

In Sections 10.4 and 10.5 we address both these problems.

10.4 Improvements of query algorithm

Recall that a query in a BWT index consists of two steps: search for a SA interval and translation of SA positions to text positions. First, let us focus on finding a Suffix Array interval.

Suppose that we are querying two k-mers k_2 and k_1 of the form Xb and aX, where X is a $(k-1)$-mer and $a, b \in \Sigma$. Recall from Section 3.5 that in a BWT index search is done in the backward direction, therefore we first query Xb and then aX. If (s_b, e_b) is a suffix interval corresponding to $k_2 = Xb$, then the suffix array interval for X is (s_X, e_X) satisfying $s_X \leq s_b$ and $e_b \leq e_X$, because X and k_2 share the same prefix X. The procedure of obtaining (s_X, e_X) from (s_b, e_b) can be regarded as removing the last character from the pattern.

Below we describe a data structure called $kLCP$ that can be used for removing the last character from the pattern during search in a BWT index.

10.4.1 $kLCP$ array

Recall that LCP of a string T of length n is the array of integers such that $LCP[i]$ is the length of the longest common prefix of $T[SA[i-1]..n-1]$ and $T[SA[i]..n-1]$, where SA is the suffix array of T.

Definition 10.4.1. Define a $kLCP$ array as

$$kLCP[i] = \begin{cases} LCP[i] \text{ if } LCP[i] \leq k-1 \\ k, \text{ otherwise} \end{cases}$$

Definition 10.4.2. Define $kLCP_{0-1}$ as

$$kLCP_{0-1}[i] = \begin{cases} 0 \text{ if } LCP[i] < k-1 \\ 1, \text{ otherwise} \end{cases}$$

Using LCP, removing the last character can be performed in the following way: decrease s while $LCP[s] \geq |X|$ and increase e while $LCP[e+1] \geq |X|$, and the interval obtained will be exactly the suffix array interval for X. As for all our queries $|X| \leq k-1$, then LCP can be replaced by $kLCP$ in this process. Moreover, if $|X| = k-1$, then removing the last character can be performed using $kLCP_{0-1}$ instead of LCP as we need to know only whether $kLCP[i] \geq k-1$ or not.

$kLCP$ and $kLCP_{0-1}$ can be constructed in linear time if LCP is known in advance. There exist many linear-time algorithms for LCP array construction, and both $kLCP$ and $kLCP_{0-1}$ can be constructed in linear time too. Below we present another algorithm for $kLCP_{0-1}$ construction that uses only a part of the BWT index.

Lemma 12. *Given a text T of length n and a BWT index for T, $kLCP_{0-1}$ can be constructed in $O(nk)$ time in the worst case using no additional memory.*

Proof. We will construct $kLCP_{0-1}$ with enumeration through all $(k-1)$-mers and trying to find them in the BWT index. If at some step we obtain empty SA interval, then we stop this branch of backtracking (this is the first heuristic). If some SA interval contains only one element, we also stop this branch (this is the second heuristics).

If we find a $(k-1)$-mer in the BWT index, and the SA interval is of size at least 2, we set ones to the corresponding elements of $kLCP_{0-1}$. Finally, we set all remaining elements of $kLCP_{0-1}$ to zero.

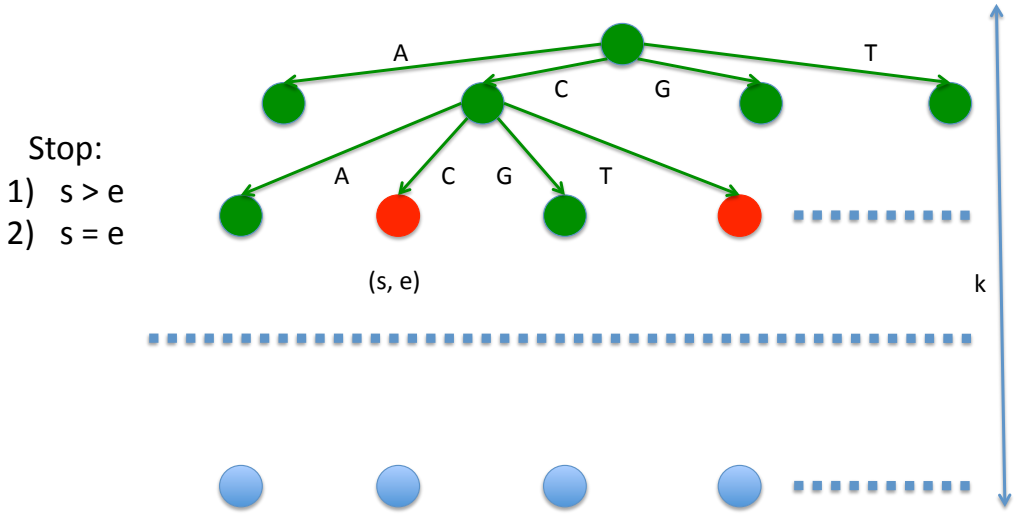


Figure 10.2: $kLCP_{0-1}$ construction. The tree represents backtracking process, where each nodes corresponds to extending the pattern by one character. If at some node the SA interval (s, e) contains zero or one element, we stop this branch of backtracking – such nodes are red.

Let us prove that in the worst case, this algorithm works in $O(nk)$ time.

The whole backtracking can be considered as traversing a tree. We can stop at a node V with the corresponding SA-interval (s, e) for two reasons:

- $s > e$ (the SA interval is empty). In this case some non-stopping (or stopping by the second heuristics) branch of backtracking contains the parent of V. The number of branches non-stopping or stopping by the second heuristics is up to n, so the number of nodes where we stopped backtracking is up to nk (the number of nodes in non-stopping branches) multiplied by 4 (size of alphabet).
- $s = e$. For each such branch there is an element in SA, so the number of nodes in such branches is up to nk again.

Finally, we can traverse up to $4nk + nk + n = O(nk)$ vertices in the tree, and this proves the lemma.

In Figure 10.2 the traversing of a tree is shown schematically. ■

It is worth to note that we actually do not use the whole BWT index in Lemma 12: for example, the sampled Suffix Array is not used.

The algorithm from Lemma 12 can be easily parallelized as it works independently for all k-mers. This algorithm requires memory for storing $kLCP_{0-1}$ itself and for a part of BWT index, and does not require the Suffix Array or the full LCP array.

To continue with the $kLCP$ construction algorithm, we need to prove several statements.

Lemma 13. *For two strings P_1 and P_2, corresponding suffix array intervals either do not intersect or are enclosed one to another.*

Proof. If one of the strings (for certainty, P_1) is a prefix of P_2, then obviously suffix array interval of P_2 is enclosed into suffix array interval of P_1. Otherwise, they can not intersect. ■

Lemma 14. *Given a text T of length n and a part of BWT index for T (namely, the BWT string and data structures supporting count and rank operations), $kLCP$ can be constructed in $O(nk)$ time in the worst case using no additional memory.*

Proof. Let us follow the algorithm described in Lemma 12. Both heuristics there can be used for $kLCP$ construction too. The only step that changes is filling the $kLCP$ array: in Lemma 12 we put ones in $kLCP_{0-1}$ if and only if some $(k-1)$-mer is found during the search, whereas for $kLCP$ construction we need to distinguish all patterns of length from 1 to $k-1$.

We need to prove that this increased number of insertions into the $kLCP$ array does not affect the working time asymptotics. As it follows from Lemma 13, for a fixed length ℓ, SA intervals for two different strings of length ℓ can not intersect. This means that for a fixed ℓ, we can make up to n insertions in $kLCP$, and overall for lengths $1 \le \ell \le k-1$ number of insertions is up to nk, and thus the construction time of $kLCP$ remains $O(nk)$. ■

10.4.2 Using $kLCP$ array for removing the last character

On the one hand, using $kLCP$ for removing the last character during pattern search in a BWT index is straightforward: given a SA interval (s, e) for pattern Xa where $|X| \le k-1$, we can decrease s while $kLCP[s] \ge |X|$ and increase e while $LCP[e+1] \ge |X|$, and the new interval (s_X, e_X) will be the SA interval for X. On the other hand, the working time of this naive algorithm is $O(|s - s_X| + |e - e_X|)$, and if X appears in the text many more times than Xa, then the naive algorithm can be time-consuming.

Decreasing s while $kLCP[s] \ge |X|$ can be viewed from another side: for a given s to find a maximum $s_X \le s$ such that $kLCP[s_X] < |X|$. In this formulation, this problem can be solved using a data structure supporting rank and select operations: first, find the number t = number of $s' < s$ such that $kLCP[s'] < |X|$ and then select the index s_X such that s_X is the t-th position satisfying $kLCP[s_X] < |X|$. The same statement holds for the $kLCP_{0-1}$ array. Thus, we proved the following lemma:

Lemma 15. *The last character of a k-mer can be removed and the suffix array interval of the corresponding $(k-1)$-mer can be updated in $O(1)$ time with use of $kLCP_{0-1}$ augmented by an auxiliary data structure for fast rank and select operations.*

Using the approach described above, we perform search of k-mers one by one in the backward direction. If the SA interval for a k-mer k_2 is not empty, then we remove the last character of k_2 and then in $O(1)$ obtain the SA interval for the next k-mer k_1. We call this approach *the rolling window* search.

In Section 10.6 we show that using $kLCP_{0-1}$ improves the working time of the SA interval search. We also show how to efficiently store $kLCP$ without using rank and select operations.

10.4.3 Speeding up SA-to-text translation

Suppose that there is only one occurrence of each consecutive k-mers aX and Xb at positions i and j of the text. Obviously, it is very likely that they appear at consecutive positions of the text (this is true even for a random text for a sufficiently big k). The

statement above remains true even if there are multiple occurrences of k-mers. This means that after we found position(s) of Xb in the text, we can avoid the translation of SA positions for aX in most cases.

Our goal is, given SA intervals for aX and Xb ((s_a, e_a) and (s_b, e_b) correspondingly), and text positions for Xb (array Q), to find text positions for aX. A first obvious idea is to prove that if $e_a - s_a = e_b - s_b$ (in other words, there is equal number of occurrences of aX and Xb in the text), then positions of aX can be obtained from positions of Xb by shifting them by 1. However, this is not correct. Let $T = TACTACGTCGT, a = A, X = C, b = G$. Then there are two occurrences of $aX = AC$ at positions 2 and 5, and two occurrences of $Xb = CG$ at positions 6 and 9. Thus, there are equal number of occurrences of AC and CG, but only at positions 5 and 6 they appear together, while the other occurrences of these k-mers are far one from another. This, in some sense, proves that if we use the restarted search, then the information about k-mers (SA intervals) is not enough to speed up translation SA positions to text positions. But for the rolling window search speeding up SA-to-text translation is possible, which is proved in the following lemma:

Lemma 16. *Let (s_a, e_a), (s_b, e_b) and (s_X, e_X) be SA intervals for aX, Xb and X correspondingly found during the rolling window search. Then, if $e_a - s_a = e_b - s_b = e_X - s_X$ then the positions in the text where aX appears can be obtained from the positions for Xb by shifting by one position to the left.*

Proof. Every time when aX or Xb appear in the text, X also appears there. Thus $e_X - s_X \geq e_a - s_a$ and $e_X - s_X \geq e_b - s_b$. Suppose that at some position aX is not followed by b, then there are at least $e_a - s_a + 1$ occurrences of X in the text. So the opposite is correct, and aX and Xb appear at consecutive positions in the text. ■

Applying Lemma 16 to the translation step during the rolling window search we can reduce the time of the translation, as shifting positions by one is much more lightweight operation than translation using sampled Suffix Array. In Section 10.6 we will show that ideas described in this section improve working time significantly in practice.

10.5 Memory usage improvement

As it was mentioned in Section 10.3, for our purposes we do not need exact text positions where k-mers appear, but only corresponding taxonomic node identifier. A sampled Suffix Array used for the translation from SA positions to the text positions can be replaced by a more lightweight data structure that allows a direct translation to node ids.

For this part it is important that in our case the "text" is a concatenation of several contigs without any separator between them. Consecutive contigs belong to the same node from the taxonomic tree. An example of such a text is shown in Figure 10.3.

One idea is to store not a text position but a corresponding *node_id* (we call this array *node id array*) for every sampled position of suffix array. We call this new array, storing only node ids instead of text positions, *node_id* array further in the text. Then, the translation can be done in the same way as for a sampled suffix array: move to the previous text position until we find a sampled position, and the answer will be the node id corresponding to this position. Such an approach can return a wrong node id in several situations: if while moving backward in the text we cross a border of two nodes, then the answer will be wrong. Another problem is that every found position should be checked whether the k-mer spans the border of two consecutive contigs (and should not be in the output in this case). However, we can check whether a position is at the border of two

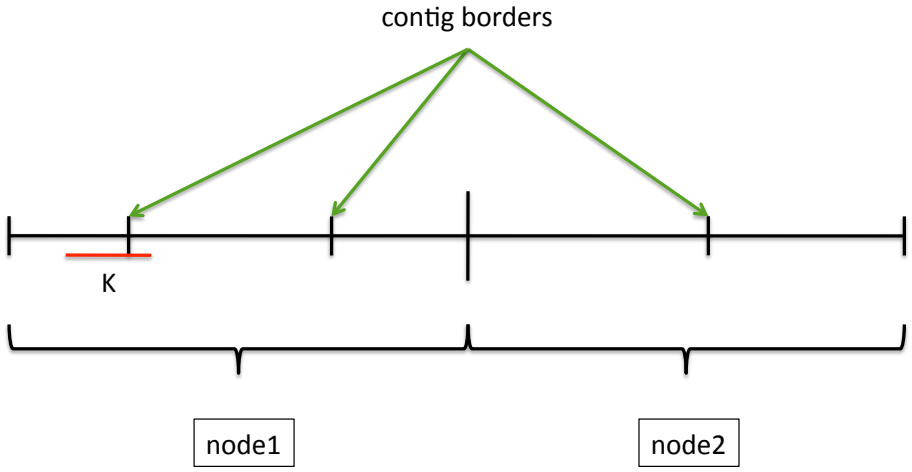


Figure 10.3: Text consisting of several concatenated contigs. Consecutive contigs belong to the same node from the taxonomic tree. Red horizontal line represents a k-mer K on the border of two contigs.

contigs only if we have exact text positions. If we translate SA positions directly to node ids then a direct check becomes impossible. In Figure 10.3 k-mer K spans the border of two contigs, and should not be in the output of the query.

However, even such a naive approach can sometimes provide good results (in other words, the error rate may be small). Imagine, for example, a text of length 10^6 that consists of 1000 contigs of equal length, and there are 10 nodes each including 100 contigs. Let every 10th position be sampled in the suffix array. At each border of two contigs we will return an incorrect answer for 10 positions in average, and overall for approximately $10 \cdot 999 = 9900$ positions the node id will be wrong. Thus, the probability of an error for such an input is only $\frac{9900}{10^6} \approx 0.01$.

Further in this section we will construct a more sophisticated data structure that avoids the problems described above.

Let T be a text of length n, SSA the sampled suffix array with sampling distance s and let l be the number of contigs which T consists of and m number of nodes. SSA can be stored in two very different ways:

1. sample every s-th position of the text
2. sample every s-th position in Suffix Array

The main difference is that in the first version the distances between consecutive sampled positions are equal (and the distances between concescutive sampled SA positions are different), and vice-versa for the second version.

Let SA^{-1} be the inverse of SA (such that $SA^{-1}[SA[i]] = i$).

Algorithm 5 describes the usual process of translation from SA positions to texts position.

Checking whether a SA position x is sampled or not is different for different types of SSA: for the second version it is obvious (we just check $x \mod s = 0$ or not), whereas for the first version we need to store a data structure supporting rank and select operations.

Algorithm 5 SA position to text translation

1: Input: BWT, sampled suffix array SSA, starting SA position x
2: $steps = 0$
3: **while** x is not sampled **do**
4: $x = SA^{-1}[SA[x] - 1]$ (in other words, suffix index that corresponds to previous position in the text, it is found using BWT)
5: $steps = steps + 1$
6: **end while**
7: Output: $SSA[x] + steps$

It is easy to see that the second variant of SSA occupies less memory and is faster than the first one at the price of the worst case.

For a position t in the text let $next(t)$ be the next sampled position of the text.

Actually, both problems with the *node_ id* array described above come from the same source: we do not know exact positions in the text and can not compare them with contig borders. The basic idea of our solution is to store contig borders in another way, i.e., if b_1, b_2.. are contigs borders between t and $next(t)$ (for uniqueness, including t and not including $next(t)$), then we store pairs $(t, b_1 - t), (t, b_2 - t)$... To maintain a common enumeration, let the border between positions x and $x+1$ be x. As t is a sampled position, then this is equivalent to storing set of $(key, value)$ pairs $C = (SA^{-1}[t], b_1 - t), SA^{-1}[t], b_2 - t)$... Below we show how this set of $(key, value)$ pairs can be stored efficiently.

Suppose that there is at most one contig border between any pair of sampled positions $(t, next(t))$. As it was mentioned above, the first type of SSA implementation is usually slower and less memory efficient than the second one. For example, BWA, which we use in ProPhyle, implements the second variant of SSA. Below we will describe how to replace SSA with a *node_ id* array for the second type of SSA. However, same statements, with small and rather technical changes, apply to the first type of SSA too.

Definition 10.5.1. Define an array D of size $\frac{n}{s}$ as

$$D[i] = \begin{cases} -1, \text{ if sampled positions } SA[i \cdot s] \text{ and } next(SA[i \cdot s]) \text{ belong to the same contig} \\ b - SA[i \cdot s] \text{ (where } b \text{ is a border position between } SA[i \cdot s] \text{ and } next(SA[i \cdot s])), \text{ otherwise.} \end{cases}$$

If x and *steps* are values found by Algorithm 5, then there are two possibilities:

1. $D[x] = -1$: there is no contig borders between $SA[x]$ and $next(SA[x])$, and the node id is *node_ id*$[x]$,
2. $D[x] \geq 0$:
   (a) $steps < D[x] - k + 1$: return *node_ id*$[x]$,
   (b) $D[x] - k + 1 \leq steps \leq D[x]$: position is at the border of two contigs, it should not be present in the output,
   (c) $steps > D[x]$: return *node_ id*$[SA^{-1}[next(SA[x])]]$ (we will explain how to get id of $next(t)$ later).

Here we utilize the fact that all contigs belonging to one node are grouped together in the text. Also we assume that sequences corresponding to different nodes are concatenated in the predefined order (in other words, we have a correspondence between node identifiers

and nodes in the taxonomic tree). In Algorithm 10.5 we also use some properties of the BWT index implementation from BWA, namely, the fact that contigs are concatenated without any separators, thus in our case $k-1$ positions at the border of two contigs are *false positives*. Under the discussed constraint, Algorithm 10.5 can determine a correct node id:

Lemma 17. *The algorithm described in 10.5 returns a correct node id if and only if there is up to one contig border between t and $next(t)$ for any sampled text position t.*

Proof. One part of this statement is obvious: if all positions in contig i are not sampled, then if a k-mer appears at the first position of contig $i+1$, Algorithm 5 can find $(i-1)$-th contig as an answer, and the algorithm will not return the correct answer.

The second part of the statement is also easy to prove: as the BWT implementation of BWA just concatenates contigs together without separators, and exactly $k-1$ positions on the border are forbidden, then this situation corresponds to the second case of Algorithm 10.5. Positions to the left of the border belong to the node $node\_id[x]$, and to the right of the border – to the node $node\_id[SA^{-1}[next(SA[x])]]$. ■

As we are working with the second type of SSA, then there is no upper bound for values of the array D. This means that each element of D occupies $\log n$ bits of memory, and overall D occupies $\frac{n}{s}\log n$ memory, which is equal to the SSA memory usage. However, we know that the average value of $D[i]$ is s which means that we can store D more compactly. Below we show how to do that.

10.5.1 Storing D compactly

The idea is to store only those distances $D[i]$ in the array $D_{cut}[i]$ which do not exceed some predefined constant d_{max}, and if $D[i] \geq d_{max}-2$ then to store -2. In other words, we replace D with D_{cut} defined as

Definition 10.5.2.

$$D_{cut}[i] = \begin{cases} D[i], \text{ if } D[i] < d_{max}-2 \\ -2, \text{ otherwise.} \end{cases}$$

Array D_{cut} allows us to use the same algorithm to determine node id from a SA position, but we introduce one more type of errors at the border of two contigs: if the distance between two sampled positions in the text is too big (more than $d_{max}-3$), then we may return an incorrect node id. For such sampled positions, we store values of D which exceed $d_{max}-3$ explicitly in a hash table or a sorted list E.

The smaller is d_{max}, the smaller values are stored in D_{cut}, but the bigger is the size of E, and vice-versa. Thus, we need to find a trade-off between the size of D_{cut} and the size of E by choosing an appropriate value of d_{max}. Below we analytically evaluate the optimal value of d_{max} and the sizes of D_{cut} and E. In Section 10.6 we also provide some experimental results confirming our idea.

We can store E as a sorted array, containing pairs (x, d), where x is a sampled position in the suffix array and d is the distance to the nearest contig border to the right. As $1 \leq x \leq n$ and $1 \leq d \leq n$, then we need $2\log n$ bits of memory for each entry of E. For each entry of D_{cut} we need $\log d_{max}$ bits of memory. There are $\frac{n}{s}$ elements in D_{cut} and $p \cdot \frac{n}{s}$ elements in E, where p is the probability that the distance from a sampled position to the contig border exceeds $d_{max}-3$. Next lemma estimates this probability:

Lemma 18. *Let T be the text of length n and let S be $\frac{n}{s}$ randomly and uniformly distributed different positions in T ("random" analogue of the SSA). Then the probability $P(S[i+1] - S[i] > d_{max})$ can be estimated as $(1 - \frac{d_{max}}{n})^{\frac{n}{s}}$.*

Proof. It is obvious that $P(S[i+1] - S[i] > d_{max}) \approx P(\min S[i] > d_{max}) = P(\forall i\ 1 \leq i \leq \frac{n}{s} : S[i] > d_{max})$. Let us remove the condition of the lemma that positions are different (this does not significantly change the distribution of distances), then $P(\forall i\ 1 \leq i \leq \frac{n}{s} : S[i] > d_{max}) = \prod_{i=1}^{\frac{n}{s}} P(S[i] > d_{max})$. Recall that positions are chosen independently and uniformly, thus $P(S[i] > d_{max}) = 1 - \frac{d_{max}}{n}$, and finally the sought-for probability is $(1 - \frac{d_{max}}{n})^{\frac{n}{s}}$. ∎

We can simplify the probability from Lemma 18: in our setup, $d_{max} \ll n$ and $s \ll n$, thus we can apply $(1 + \frac{a}{x})^x \approx e^a$ if $x \to \infty$ and finally obtain $(1 - \frac{d_{max}}{n})^{\frac{n}{s}} = ((1 - \frac{d_{max}}{n})^{\frac{n}{d_{max}}})^{\frac{d_{max}}{s}} \approx e^{\frac{-d_{max}}{s}}$ for big enough values of n.

Now we can calculate the expected number of elements in $E : |E| = e^{\frac{-d_{max}}{s}} \cdot \frac{n}{s}$, and the total expected size of D and E together that we want to minimize is $e^{\frac{-d_{max}}{s}} \cdot \frac{n}{s} \cdot 2 \log n + \frac{n}{s} \log d_{max}$. The following lemma finds a condition on d_{max} when this minimum is achieved:

Lemma 19. *The total expected size of E and G is minimum when $d_{max} = c \cdot s$, where c satisfies $ce^{-c} = \frac{1}{2 \log n}$.*

Proof. We want to minimize $f(d_{max}) = e^{\frac{-d_{max}}{s}} \cdot \frac{n}{s} \cdot 2 \log n + \frac{n}{s} \log d_{max}$. Below we replace d_{max} by d to simplify equations. As $\frac{n}{s}$ is constant, then it is equivalent to minimizing $g(d) = e^{\frac{-d}{s}} \cdot 2 \log n + \log d$. The minimum is achieved when $0 = g'(d) = \frac{1}{d} - \frac{1}{s} e^{\frac{-d}{s}} \cdot 2 \log n$, which is equivalent to $de^{\frac{-d}{s}} = \frac{s}{2 \log n}$. Let $d = cs$, then for c holds $ce^{-c} = \frac{1}{2 \log n}$, which proves the statement of the lemma. ∎

The default value for s is usually 32 (BWA also uses it). We numerically computed the optimal value of d_{max} for $n = 10^9$ (typical value for our applications) using Lemma 19 and obtained $d_{max} = 187$. Then the optimal size of D is $\frac{n}{s} \log d_{max} \approx 0.23n$, and the optimal size of E is $|E| = 2e^{\frac{-d_{max}}{s}} \cdot \frac{n}{s} \log n \approx 10^{-4} n \log n$.

Now we show how to solve the two remaining issues: the first one concerning multiple contig borders between t and $next(t)$, and the second one about finding the correct node id for $next(t)$ from Algorithm 10.5.

10.5.2 Multiple contig borders between t and $next(t)$

Let us return to the initial formulation of the problem: there is a set C consisting of pairs (x, d) where $1 \leq x \leq \frac{n}{s}$ and $-2 \leq d < d_{max} - 2$ (remember that we store pairs with $d \geq d_{max}$ in a separate structure, so we can consider that $-1 \leq d < d_{max}$). Moreover, there is at least one value associated with every key x. As we showed above, if there is at most one value associated with every key, then we can store them in the usual array optimally.

When there is more than one value associated with some keys , we use a standard trick to store C: we sort all pairs from C in the ascending order to obtain a sorted array C_{sorted} (first sorted by keys, and if keys are equal then sorted by values), and store only values of all pairs in the array D_{cut} (we will use the same name for clarity) in the same order. Additionally we store a bit array K that contains $|C|$ values: for every index $1 \leq i \leq |C|$ we store 1 if $C_{sorted}[i].key \neq C_{sorted}[i-1].key$ (here we assume $C_{sorted}[-1] = (-1, -1)$).

On top of K we store a compact data structure that supports a fast *select* operation, i.e., for a given i finds i-th 1 in K. We call K together with this auxiliary data structure K_{select}. K_{select} occupies only $|K| + o(|K|)$ bits and performs *select* operation in $O(1)$ time (see, for example, [169, 170]).

Everything proved in Section 10.5 remains true when we switch from D_{cut} for the single-border case to D_{cut} augmented by K_{select}.

10.5.3 Correct node id for $next(t)$

The last issue to be solved is to determine a correct node id when we cross the border in Algorithm 10.5. The idea is to store one bit showing whether crossing the corresponding border changes the node id or not for every entry of C. For every t the position $next(t)$ belongs to the same node or to the next one, thus for every entry one bit is enough to store this information, and we store them in array B. Algorithm 10.5 can be modified accordingly to support multiple borders for the same sampled position and to identify node id correctly.

Overall, we replace the sampled Suffix Array with the following data structures: $node\_id$ array, D_{cut} array, E (sorted list or map), K_{select} and B.

Let us describe an example of obtained data structures. Suppose we have 4 contigs of length 20, 30, 15 and 35 correspondingly. Let first two of them belong to the first node and the others to the second node. We enumerate all positions starting from 0 (so the last one will be 99). Let the sampling distance s be 32, and let positions 15, 30, 70 and 90 be sampled. Assume that they correspond to Suffix Array positions 0, 32, 64 and 96 correspondingly (in reality, the order can be different). We will use numbers 0, 1, 2 and 3 to index sampled positions 15, 30, 70 and 90, respectively. Finally, let k (the length of k-mers) be 4 and d_{max} be 25.

First, $node\_id[0] = 1, node\_id[1] = 1, node\_id[2] = 2$ and $node\_id[3] = 2$. Between positions 15 and 30 there is one border (at position 19), between positions 30 and 70 there are two borders (at positions 49 and 64) and between positions 70 and 90 there is no borders. Then, we need to store four $(key, value)$ pairs describing borders: $(0, 19 - 15) = (0, 4)$, $(1, 49 - 30) = (1, 19)$, $(1, 64 - 30) = (1, 34)$, $(2, -1)$ and $(3, -1)$. Note that $34 > d_{max} - 3$, so we add the pair $(1, 34)$ to E (hash table or sorted list, for example). Then we store the rest of the pairs in the array as $4, 19, -2, -1, -1$. As every entry in this array, except the third one, corresponds to a new sampled position, then the bit array K will be $1, 1, 0, 1, 1$. Finally, the array B described above will look like $0, 1, 0, 0, 0$, as only for the second sampled position the next contig belongs to the next node.

Obtained data structures are shown on Figure 10.4.

10.5.4 Memory improvement estimation

Altogether, for $s = 32$, data structures D_{cut}, E, K_{select} and B occupy approximately $0.23n + 10^{-4} n \log n + \frac{2n}{32} + 2g + \frac{o(n)}{s}$ bits of memory. $node\_id$ array itself occupies $\frac{n}{s} \log m$ bits of memory, where m is the number of nodes (thousands or tens of thousands in our case) and g is the number of contigs. Let us compare it with the memory needed for the sampled Suffix Array, which is $\frac{n}{s} \log n$ bits. Assume that $s = 32$. In terms of bits used per one character of the text, our structures occupy $A = \frac{\log m}{32} + 0.23 + 10^{-4} \log n + \frac{1}{16} + \frac{2g}{n}$ bits per character, and the sampled Suffix Array occupies $B = \frac{\log n}{32}$. A typical value of n for our inputs is of order of 10^{10}, the number of nodes m is of order of 10^4, and the average contig length is around 100, thus $g \sim \frac{n}{100}$. For $n = 10^{10}$, $\log n \approx 33$, and thus positions are stored in 64 bits in practice (for example, BWA stores values of sampled Suffix Array

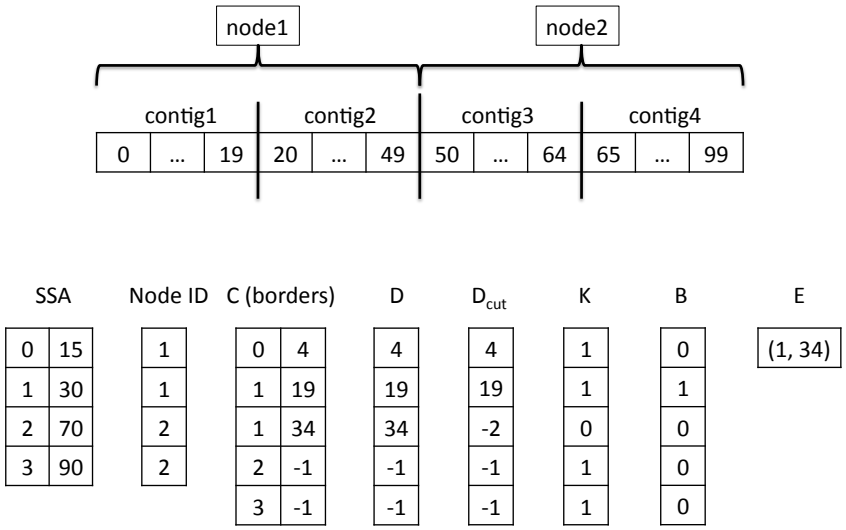


Figure 10.4: Example of data structures, replacing sampled Suffix Array.

in 64 bits). For $m = 10^4$, each entry of $node\_id$ array is stored in 16 bits. Then, for $n = 10^{10}, m = 10^4, g = 10^7$, $A \approx \frac{16}{32} + 0.23 + \frac{1}{16} + \frac{2}{100} = 0.8125$, $B = 2$. Therefore A is 2.5 times less than B, and we can save around 60% of memory compared to storing the sampled Suffix Array.

10.6 Experimental results

In this section we provide experiments on ProPhyle index properties. First, in Section 10.6.1 we introduce the way we store the $kLCP$ array. Then in Section 10.6.2 we measure the index query speed and the influence of query improvements, described in Section 10.4. In Section 10.6.3 we reveal some experimental properties of the $kLCP$ array. In Section 10.6.4 we find out how using of Node ID array instead of Suffix Array influences the memory usage and the time performance. Finally, in Section 10.6.5 we provide experimental properties of the whole ProPhyle index.

For all our experiments we use the RefSeq database containing 2787 bacterial genomes [156]. We generated reads using DWGSIM (https://github.com/nh13/DWGSIM) simulator (options DWGSIM -1 100 -2 0 -e 0.02 -C 0.1). All experiments have been done on a computer with Ubuntu, 24 cores and 64 GB of memory.

10.6.1 Storing $kLCP$ array

In Section 10.4 we briefly mentioned two ways of storing the $kLCP$ array. A naive approach is to store a bit array and iterate over its elements one by one in order to extend a SA interval. This approach is straightforward and easy to implement, however, if the SA

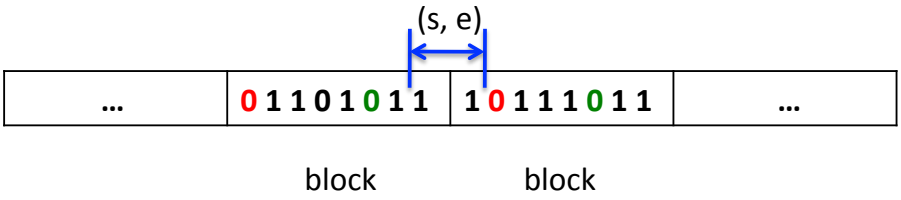


Figure 10.5: Storing $kLCP$ array as blocks of 8 bits. First and last zero bits are red and green correspondingly. To extend the interval (s, e), we need to find the first zero to the left and to the right.

interval can be extended significantly, then this naive approach can be slow. The second idea is to store an auxiliary data structure supporting rank and select operations to be able to find new SA interval borders in $O(1)$. This approach requires an additional memory and may be slower than the naive algorithm when extensions are generally small.

Here we present another way to store the $kLCP$ array (see Figure 10.5 for reference). First, for practical reasons elements of the $kLCP$ bitarray are grouped in blocks of p elements (usually, p is $8, 16, 32, 64$ or 128). We choose p to be 16 and for every possible value of the block (from 0 to $2^{16} - 1$) we compute the place of the first and the last zero bits in its bit representation. Then, while extending a SA interval (s, e), we iterate over blocks, not individual bits, and for every block check whether it has zero bit or not to the left or to the right of s or e. If it does, then the result is the adjacent bit to the first or last zero (depending on the direction of the extension) in this block. Otherwise we move to the next block. This algorithm requires just $2^{16} \cdot 4 \cdot 2 < 10^6$ bytes of memory to store the first and the last zero bits for every block. This approach is faster than the naive one for big SA interval extensions, and it is faster than the second algorithm described above in the case of considerably small extensions. Finally, in the case of metagenomic classification, when extensions are generally small, this algorithm can be a good way for storing $kLCP$.

10.6.2 Experiments on query time

In this section we measure the efficiency of query improvements, introduced in Section 10.4.

First, recall that a BWT query consists of two parts: finding a SA interval (s, e) and translation of every position from (s, e) to a text position. In some border cases, one of these steps can take much more time than the other one. Suppose that, in some artificial experiment, for every queried k-mer we stop the query on $(k-1)$-mer because of the empty SA interval. Then the translation step will not be used at all, so the SA interval search will occupy 100% of time. On the other hand, it is easy to imagine an input when the translation step takes much more time than the SA interval search: suppose that the text is big enough and random, and we query k-mers with a very small k. Then SA intervals found on the first step will be very big, and the translation will be extremely slow. However, both these situations (and many others) are not usual cases when we deal with the metagenomic classification problem, as many k-mers will be found in the index. At the same time, due to the propagation, for big enough values of k SA intervals will be rather small. These considerations, in some sense, prove that both improvements, described in Section 10.4,

can significantly reduce the query time in practice.

Query speed for different query algorithms are shown in Table 10.1: the basic one, the one using $kLCP$ for SA search, and the one with both improvements. It is clear that both improvements improve the query speed, and together they make queries 2.5 times faster.

Table 10.1: Query speed (in reads per minute) for different query algorithms.

Algorithm	RPM
basic	190000
using kLCP for SA search only	234000
using kLCP both for SA search and for sa-to-text translation	434000

10.6.3 $kLCP$ properties

In this section, we experimentally study statistical properties of $kLCP$ array. They are interesting from two points of view. First, they allow us to better understand how to store and query $kLCP$ array. Secondly, they may lead to a better understanding of the whole index.

First, we calculate the distribution of ones and zeros in the $kLCP$ array and the average length of a consecutive run of ones for different values of k. The results are presented in Table 10.2.

Table 10.2: $kLCP$ properties.

k	16	20	24	31
percentage of ones	94%	35%	26%	24%
average length of run of ones	17.3	2.82	3.05	3.09

From Table 10.3, we observe that the number of ones decreases when k increases. Moreover, the percentage of ones is 94% for $k = 16$ which can be explained by the fact that many 16-mers appear multiple times in the text as k is quite small. For $k = 20$ the percentage (35%) is much lower.

The average length of consecutive ones is big for $k = 16$ and is almost the same for $k = 20, 24, 31$. Moreover, the average length even increases insignificantly. The latter may be due to the fact that "random" (and thus single) ones disappear when k increases from $k = 20$ to $k = 31$, while "real" ones are almost the same for $k = 20, 24, 31$.

Next, we use $kLCP$ to query simulated reads, and calculate the average SA interval extension length while removing last character. The results for different values of k are shown in Table 10.3.

Table 10.3: Average SA interval extension length while removing the last character.

k	16	20	24	31
average SA interval extension	48	0.79	0.23	0.14

Tables 10.2 and 10.3 show that, with increase of k, the average number of ones, the average length of a run of ones and, most importantly, the average extension of SA interval decrease. Thus, the bigger k we use, the faster the removal of the last character works.

10.6.4 Experiments on Node ID

We implemented a basic version of *node_id* array, replacing a sampled Suffix Array. Instead

of storing a correspondence between SA positions and text positions, for every sampled SA position we store corresponding node id. In our implementation, every entry of Suffix Array occupies 64 bits (as it does not fit into 32 bits provided by *int*32 type), while *node_id* array elements can be stored in 16 bits, as number of nodes is only 2787. Thus, *node_id* requires less than 1 GB of memory in comparison to Suffix Array occupying more than 3 GB.

As it was discussed in Section 10.5, Suffix Array replacement by *node_id* array can lead to incorrect results in two cases:

1. some SA positions can be translated to a wrong node id, as a border of two nodes can be passed while moving backward in the text,
2. positions on a border of two consecutive contigs should be excluded from results, which is impossible with *node_id* array, as contig borders can not be checked without knowing exact text position.

We queried simulated reads against the index for values of $k = 20, 24, 31$. For every position in Suffix Array that we translated to a node id, we compared the translation using the sampled Suffix Array (that is always correct) to the translation using *node_id* array. There are three possibilities:

1. the corresponding k-mer spans a border of two consecutive contigs (this corresponds to error described in case 2 above),
2. the resulting node ids are different (see 1),
3. they are the same (this number should be close to 100% if *node_id* can replace Suffix Array without the big number of false answers).

The results of these experiments are presented in Table 10.4.

Table 10.4: Translation error rate using *node_id* array.

k	20	24	31
correct translations, %	97.2	98.5	98.9
incorrect node id, %	0.006	0.006	0.006
position on a border, %	2.79	1.49	1.05

From Table 10.4, we can see that the percentage of wrong node ids is very low. The primary reason for this is that the number of nodes (2787) is very small in comparison to size of the text ($\approx 10^{10}$ characters for bacteria dataset [156]). Thus, the probability that we cross the border of two nodes moving backward using the sampled Suffix Array is very small.

The number of situations when we cross the border of two contigs is much bigger. Partially this can be explained by the fact that the number of contigs (of order of 10^8 in our case) is much bigger than the number of nodes. With k increasing from 20 to 31, the percentage of k-mers that span borders becomes significantly lower. We believe that this is due to the fact that most such situations are caused by "random" matches. k-mers spanning a border of two contigs are in some sense random, as they appeared as a concatenation of the end of one contig and the beginning of another contig. As a consequence, using just the *node_id* array instead of the sampled Suffix Array leads to a significant memory usage reduction, while almost not affecting the results of classification as a small number of these random k-mers spanning the border will not change lists of matching k-mers for nodes in the tree significantly.

10.6.5 Experiments on ProPhyle index

For the current implementation, the index construction for the bacteria dataset [156] for $k = 31$ takes around 5 hours. The main reason is non-parallel algorithms used during many steps of construction. The most memory-consuming step is BWT transform (around 2 hours). The $kLCP$ array construction takes around 1.5 hours. We believe that the construction time can be strongly reduced as $kLCP$ construction algorithm can be easily parallelized.

ProPhyle can classify around 434000 *reads per minute* (RPM) (this time does not include index loading time). This time is achieved using the rolling window search. The restarted search works around 2.5 times slower (190000 RPM). This time includes only reads query time, and does not include time needed for the classification.

Memory usage during ProPhyle index construction is 13 GB (for example, Kraken [157] uses 120 Gb). For queries, ProPhyle occupies 12.4 GB in case of restarted search and 14.2 GB for rolling window search. More details about the space occupied by different substructures are provided in Table 10.5.

Table 10.5: ProPhyle memory usage.

structure	space, GB
BWT string + rank/select structure	7.5
sampled Suffix Array	3.8
$kLCP$ array	1.9
information about contig borders	1.0

Main ProPhyle repository is `https://github.com/prophyle/prophyle`. Index construction and query described in this manuscript are implemented in `https://github.com/prophyle/prophex`. A comparison with Kraken [157] can be found in [61].

10.7 Discussion

We introduced several data structures that are used in our metagenomic classification tool called ProPhyle. Our BWT-index-based solution utilizes only a small amount of memory and, at the same time, allows for fast queries. Moreover, our index is *lossless*, storing a full information about k-meran approach to approximate string matching that relies on a bidirectional index of the text presence in the reference genomes.

We believe that ProPhyle, being extensively developing, can be refined further in many directions. From the performance point of view, index construction, query speed and memory footprint can be improved.

The index construction can be strongly speeded up as non-parallel algorithms are used in most steps. The most time consuming index construction step is the calculation of BWT. As it was stated in Section 10.6, the construction of a $kLCP$ array occupies significant time. In Section 10.4 we mentioned that our construction algorithm is easily parallelizible, and this may significantly reduce the construction time.

Full implemetation of *node_id* array can significantly reduce the memory used by a whole index, while providing the same results. Moreover, taxonomic tree binarization and better k-mer assembly can improve memory usage even further.

Part V

Conclusions

Conclusions

In this thesis, we studied algorithmic methods and data structures for string matching, genome assembly and metagenomic classification problems. Here we summarize our main results and enumerate directions for the future research.

Approximate string matching In Part II we studied an approach to approximate string matching that relies on a bidirectional index of the text. We presented a framework called *search schemes* and provided a probabilistic measure of their efficiency. Then we provided two types of improvements and experimentally proved their efficiency. We also discovered and proved several combinatorial properties of optimal search schemes.

We believe that efficient search schemes, based on bidirectional indexes, can be designed automatically. We expect that the methods, discussed in this work, can be applied in practice to hybrid approaches to approximate string matching, when filtering of potential matched positions is followed by approaches based on backtracking.

Genome assembly and Read compression In Part III we introduced a novel memory-efficient data structure called Cascading Bloom filter.

We described how Cascading Bloom filter can be applied to the genome assembly problem. We provided analytical calculations showing that Cascading Bloom filter uses 8 to 9.5 bits per k-mer, compared to 13 to 15 bits used by the method of [126]. We incorporated our data structure into Minia software [126] and showed that using Cascading Bloom filter leads 30%-40% smaller memory footprint and 20%-30% decrease in query time.

In Chapter 8 we showed how Cascading Bloom filter can be applied to reference-based read set compression. We introduced an extension of the strategy presented in [137] that led to a notable improvement of the compression ratio.

More memory efficient replacements of Bloom filter, such as presented in [136], may lead to more efficient implementation of Cascading Bloom filter.

Metagenomic classification In Part IV we presented data structures used in our metagenomic classification tool called *ProPhyle*. We improved the standard BWT index for the purpose of our application both in terms of query speed and memory usage. We introduced data structures called kLCP and Node ID array and showed their efficiency both theoretically and experimentally.

We believe that *ProPhyle* can be improved even further both in terms of efficiency and versatility and can be applied to different genomic analyses involving metagenomic datasets. Using parallel versions of algorithms, we could strongly improve ProPhyle index construction time. Memory usage could be reduced applying taxonomic tree binarization, better k-mer assembly and full implementation of $node\_id$ array.

One more direction for future is developing precise and accurate assigning algorithms. Currently only simple assigning algorithms that choose a node with best k-mer hit number

or coverage are implemented in Python and C++. We believe that much more accurate assigning algorithms, utilizing the lossless nature of our index, can be designed.

Bibliography

[1] G. Kucherov, K. Salikhov, and D. Tsur. "Approximate string matching using a bidirectional index". In: *Theor. Comput. Sci.* 638 (2016), pp. 145–158. DOI: 10.1016/j.tcs.2015.10.043. URL: https://doi.org/10.1016/j.tcs.2015.10.043 (cit. on pp. 4, 31).

[2] K. Salikhov, G. Sacomoto, and G. Kucherov. "Using cascading Bloom filters to improve the memory usage for de Brujin graphs". In: *BMC Algorithms for Molecular Biology* 9.1 (2014), p. 2. URL: http://www.almob.org/content/9/1/2 (cit. on pp. 4, 70).

[3] K. Salikhov. "Improved compression of DNA sequencing data with Cascading Bloom filters". In: *Special issue of the International Journal Foundations of Computer Science (IJFCS) for the international Student Conference on Mathematical Foundations in Bioinformatics (MatBio)* (2017). To appear. (cit. on p. 4).

[4] J. D. Watson and F. H. Crick. "Molecular structure of nucleic acids; a structure for deoxyribose nucleic acid." In: *Nature* 171.4356 (Apr. 25, 1953), pp. 737–738. URL: http://www.ebi.ac.uk/citexplore/citationDetails.do?externalId=13054692%5C&dataSource=MED (cit. on pp. 9, 12, 13).

[5] Z. D. Stephens et al. "Big Data: Astronomical or Genomical?" In: *PLoS Biol* 13.7 (July 2015), pp. 1–11. DOI: 10.1371/journal.pbio.1002195. URL: http://dx.doi.org/10.1371/journal.pbio.1002195 (cit. on p. 9).

[6] J. C. Venter. "Environmental Genome Shotgun Sequencing of the Sargasso Sea". In: *Science* 304.5667 (2004), pp. 66–74. DOI: 10.1126/science.1093857. URL: http://www.sciencemag.org/cgi/doi/10.1126/science.1093857%20http://www.ncbi.nlm.nih.gov/pubmed/15001713 (cit. on p. 10).

[7] E. Karsenti et al. "A Holistic Approach to Marine Eco-Systems Biology". In: *PLoS Biology* 9.10 (2011), e1001177. DOI: 10.1371/journal.pbio.1001177. URL: http://dx.plos.org/10.1371/journal.pbio.1001177%20http://www.ncbi.nlm.nih.gov/pubmed/22028628%20http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3196472 (cit. on p. 10).

[8] J. Qin et al. "A human gut microbial gene catalogue established by metagenomic sequencing". In: *Nature* 464.7285 (2010), pp. 59–65. DOI: 10.1038/nature08821. URL: http://www.nature.com/doifinder/10.1038/nature08821 (cit. on p. 10).

[9] T. M. Vogel et al. "TerraGenome: a consortium for the sequencing of a soil metagenome". In: *Nature Reviews Microbiology* 7.4 (2009), pp. 252–252. DOI: 10.1038/nrmicro2119. URL: http://www.nature.com/nrmicro/journal/v7/n4/full/nrmicro2119.html%20http://www.nature.com/doifinder/10.1038/nrmicro2119 (cit. on p. 10).

[10] J. Peterson et al. "The NIH Human Microbiome Project". In: *Genome Research* 19.12 (2009), pp. 2317–2323. DOI: 10.1101/gr.096651.109. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2792171%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract%20http://genome.cshlp.org/cgi/doi/10.1101/gr.096651.109 (cit. on p. 10).

[11] T. W. Lam et al. "High Throughput Short Read Alignment via Bi-directional BWT". In: *Proc. IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2009, pp. 31–36 (cit. on pp. 11, 21, 31–33, 35, 47).

[12] B. Alberts et al. *Molecular Biology of the Cell*. Fifth. Other, Nov. 20, 2007. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20%5C&path=ASIN/0815341067 (cit. on p. 12).

[13] R. E. Franklin and R. G. Gosling. "Molecular Configuration in Sodium Thymonucleate". In: *Nature* 171.4356 (Apr. 25, 1953), pp. 740–741. DOI: 10.1038/171740a0. URL: http://dx.doi.org/10.1038/171740a0 (cit. on p. 12).

[14] A. M. Maxam and W. Gilbert. "A new method for sequencing DNA." In: *Proceedings of the National Academy of Sciences of the United States of America* 74.2 (Feb. 1977), pp. 560–564. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC392330/ (cit. on p. 13).

[15] F. Sanger, S. Nicklen, and A. R. Coulson. "DNA sequencing with chain-terminating inhibitors." In: *Proceedings of the National Academy of Sciences of the United States of America* 74.12 (Dec. 1, 1977), pp. 5463–5467. DOI: 10.1073/pnas.74.12.5463. URL: http://dx.doi.org/10.1073/pnas.74.12.5463 (cit. on p. 13).

[16] E. R. Mardis. "Next-Generation DNA Sequencing Methods". In: *Annual Review of Genomics and Human Genetics* 9.1 (June 19, 2008), pp. 387–402. DOI: 10.1146/annurev.genom.9.081307.164359. URL: http://dx.doi.org/10.1146/annurev.genom.9.081307.164359 (cit. on p. 14).

[17] E. R. Mardis. "A decade's perspective on DNA sequencing technology". In: *Nature* 470.7333 (Feb. 9, 2011), pp. 198–203. DOI: 10.1038/nature09796. URL: http://dx.doi.org/10.1038/nature09796 (cit. on p. 14).

[18] M. L. Metzker. "Sequencing technologies - the next generation." In: *Nature reviews. Genetics* 11.1 (Jan. 8, 2010), pp. 31–46. DOI: 10.1038/nrg2626. URL: http://dx.doi.org/10.1038/nrg2626 (cit. on p. 14).

[19] J. Thompson and P. Milos. "The properties and applications of single-molecule DNA sequencing". In: *Genome Biology* 12.2 (2011), pp. 217+. DOI: 10.1186/gb-2011-12-2-217. URL: http://dx.doi.org/10.1186/gb-2011-12-2-217 (cit. on p. 14).

[20] M. Margulies et al. "Genome sequencing in microfabricated high-density picolitre reactors". In: *Nature* 437.7057 (July 31, 2005), pp. 376–380. DOI: 10.1038/nature03959. URL: http://dx.doi.org/10.1038/nature03959 (cit. on p. 14).

[21] H. Y. K. Lam et al. "Performance comparison of whole-genome sequencing platforms". In: *Nature Biotechnology* 30.1 (Dec. 18, 2011), pp. 78–82. DOI: 10.1038/nbt.2065. URL: http://dx.doi.org/10.1038/nbt.2065 (cit. on p. 14).

[22] M. Barba, H. Czosnek, and A. Hadidi. "Historical Perspective, Development and Applications of Next-Generation Sequencing in Plant Virology". In: *Viruses* 6.1 (2014), pp. 106–136. DOI: 10.3390/v6010106. URL: http://www.mdpi.com/1999-4915/6/1/106 (cit. on p. 14).

[23] L. T. França, E. Carrilho, and T. B. Kist. "A review of DNA sequencing techniques." In: *Quarterly reviews of biophysics* 35.2 (May 2002), pp. 169–200. URL: http://view.ncbi.nlm.nih.gov/pubmed/12197303 (cit. on p. 14).

[24] B. Liu et al. "Accurate and fast estimation of taxonomic profiles from metagenomic shotgun sequences". In: *BMC Genomics* 12.2 (2011), S4. DOI: 10.1186/1471-2164-12-S2-S4 (cit. on pp. 14, 76).

[25] J. K. Kulski. *Next-Generation Sequencing — An Overview of the History, Tools, and "Omic" Applications*. 2015. DOI: 10.5772/61964. URL: http://dx.doi.org/10.5772/61964 (cit. on p. 14).

[26] S. Levy et al. "The Diploid Genome Sequence of an Individual Human". In: *PLoS Biol* 5.10 (Sept. 4, 2007), e254+. DOI: 10.1371/journal.pbio.0050254. URL: http://dx.doi.org/10.1371/journal.pbio.0050254 (cit. on p. 14).

[27] D. A. Wheeler et al. "The complete genome of an individual by massively parallel DNA sequencing". In: *Nature* 452.7189 (Apr. 17, 2008), pp. 872–876. DOI: 10.1038/nature06884. URL: http://dx.doi.org/10.1038/nature06884 (cit. on p. 14).

[28] L. M. Bragg et al. "Shining a Light on Dark Sequencing: Characterising Errors in Ion Torrent PGM Data". In: *PLoS Comput Biol* 9.4 (Apr. 11, 2013), e1003031+. DOI: 10.1371/journal.pcbi.1003031. URL: http://dx.doi.org/10.1371/journal.pcbi.1003031 (cit. on p. 14).

[29] A. Gilles et al. "Accuracy and quality assessment of 454 GS-FLX Titanium pyrosequencing". In: *BMC Genomics* 12.1 (May 19, 2011), pp. 245+. DOI: 10.1186/1471-2164-12-245. URL: http://dx.doi.org/10.1186/1471-2164-12-245 (cit. on p. 14).

[30] S. Huse et al. "Accuracy and quality of massively parallel DNA pyrosequencing". In: *Genome Biology* 8.7 (July 20, 2007), R143+. DOI: 10.1186/gb-2007-8-7-r143. URL: http://dx.doi.org/10.1186/gb-2007-8-7-r143 (cit. on p. 14).

[31] F. C. Botelho, Y. Kohayakawa, and N. Ziviani. "A Practical Minimal Perfect Hashing Method". In: *Experimental and Efficient Algorithms*. 2005, pp. 488–500. DOI: 10.1007/11427186_42. URL: http://dx.doi.org/10.1007/11427186%5C_42 (cit. on p. 16).

[32] D. Belazzougui et al. "Monotone Minimal Perfect Hashing: Searching a Sorted Table with <italic>O</italic>(1) Accesses". In: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*. 2009, pp. 785–794. DOI: 10.1137/1.9781611973068.86. URL: http://epubs.siam.org/doi/abs/10.1137/1.9781611973068.86 (cit. on p. 16).

[33] A. Limasset et al. "Fast and scalable minimal perfect hashing for massive key sets". In: *CoRR* abs/1702.03154 (2017) (cit. on p. 16).

[34] B. H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. DOI: 10.1145/362686.362692. URL: http://doi.acm.org/10.1145/362686.362692 (cit. on p. 16).

[35] A. Kirsch and M. Mitzenmacher. "Less hashing, same performance: Building a better Bloom filter". In: *Random Struct. Algorithms* 33.2 (Sept. 2008), pp. 187–218. URL: http://dx.doi.org/10.1002/rsa.v33:2 (cit. on p. 16).

[36] P. Weiner. "Linear Pattern Matching Algorithms". In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*. SWAT '73. Washington, DC, USA: IEEE Computer Society, 1973, pp. 1–11. DOI: 10.1109/SWAT.1973.13. URL: http://dx.doi.org/10.1109/SWAT.1973.13 (cit. on p. 17).

[37] E. M. McCreight. "A Space-Economical Suffix Tree Construction Algorithm". In: *J. ACM* 23.2 (Apr. 1976), pp. 262–272. DOI: 10.1145/321941.321946. URL: http://doi.acm.org/10.1145/321941.321946 (cit. on p. 17).

[38] E. Ukkonen. "On-line construction of suffix trees". In: *Algorithmica* 14.3 (1995), pp. 249–260. DOI: 10.1007/BF01206331. URL: http://dx.doi.org/10.1007/BF01206331 (cit. on p. 17).

[39] M. Farach. "Optimal suffix tree construction with large alphabets". In: *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*. Oct. 1997, pp. 137–143. DOI: 10.1109/SFCS.1997.646102 (cit. on p. 17).

[40] C. Meek, J. M. Patel, and S. Kasetty. "OASIS: an online and accurate technique for local-alignment searches on biological sequences". In: *Proceedings of the 29th international conference on Very large data bases - Volume 29* (2003), pp. 910–921 (cit. on pp. 18, 31).

[41] A. L. Delcher et al. "Alignment of whole genomes". In: *Nucleic Acids Research* 27.11 (1999), pp. 2369–2376. DOI: 10.1093/nar/27.11.2369. URL: http://nar.oxfordjournals.org/content/27/11/2369.abstract (cit. on pp. 18, 31).

[42] U. Manber and G. Myers. "Suffix Arrays: A New Method for On-line String Searches". In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '90. San Francisco, California, USA: Society for Industrial and Applied Mathematics, 1990, pp. 319–327. URL: http://dl.acm.org/citation.cfm?id=320176.320218 (cit. on p. 18).

[43] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. "Information Retrieval". In: ed. by W. B. Frakes and R. Baeza-Yates. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. Chap. New Indices for Text: PAT Trees and PAT Arrays, pp. 66–82. URL: http://dl.acm.org/citation.cfm?id=129687.129692 (cit. on p. 18).

[44] J. Kärkkäinen and P. Sanders. "Simple Linear Work Suffix Array Construction". In: *Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30 – July 4, 2003 Proceedings*. Ed. by J. C. M. "Baeten et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 943–955. DOI: 10.1007/3-540-45061-0_73. URL: http://dx.doi.org/10.1007/3-540-45061-0_73 (cit. on p. 18).

[45] G. Nong, S. Zhang, and W. H. Chan. "Linear Suffix Array Construction by Almost Pure Induced-Sorting". In: *2009 Data Compression Conference*. Mar. 2009, pp. 193–202. DOI: 10.1109/DCC.2009.42 (cit. on p. 18).

[46] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. "A Taxonomy of Suffix Array Construction Algorithms". In: *ACM Comput. Surv.* 39.2 (July 2007). DOI: 10.1145/1242471.1242472. URL: http://doi.acm.org/10.1145/1242471.1242472 (cit. on p. 18).

[47] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. "Replacing Suffix Trees with Enhanced Suffix Arrays". In: *J. of Discrete Algorithms* 2.1 (Mar. 2004), pp. 53–86. DOI: 10.1016/S1570-8667(03)00065-0. URL: http://dx.doi.org/10.1016/S1570-8667(03)00065-0 (cit. on p. 19).

[48] K. Malde, E. Coward, and I. Jonassen. "Fast sequence clustering using a suffix array algorithm". In: *Bioinformatics* 19.10 (July 1, 2003), pp. 1221–1226. DOI: 10.1093/bioinformatics/btg138. URL: http://dx.doi.org/10.1093/bioinformatics/btg138 (cit. on p. 19).

[49] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. "Replacing suffix trees with enhanced suffix arrays". In: *Journal of Discrete Algorithms* 2.1 SPEC. ISS. (2004), pp. 53–86. DOI: 10.1016/S1570-8667(03)00065-0 (cit. on pp. 19, 31).

[50] C. Otto, P. F. Stadler, and S. Hoffmann. "Fast and sensitive mapping of bisulfite-treated sequencing data". In: *Bioinformatics* 28.13 (2012), pp. 1698–1704. DOI: 10.1093/bioinformatics/bts254 (cit. on pp. 19, 31).

[51] M. Burrows and D. J. Wheeler. *A block-sorting lossless data compression algorithm*. Tech. rep. 1994 (cit. on p. 19).

[52] P. Ferragina and G. Manzini. "Opportunistic data structures with applications". In: *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. 2000, pp. 390–398. DOI: 10.1109/SFCS.2000.892127 (cit. on pp. 19, 31).

[53] M. Burrow and D. Wheeler. *A block-sorting lossless data compression algorithm*. Technical Report 124. Digital Equipment Corporation, California, 1994 (cit. on pp. 19, 31).

[54] R. Grossi, A. Gupta, and J. S. Vitter. "High-order Entropy-compressed Text Indexes". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '03. Baltimore, Maryland: Society for Industrial and Applied Mathematics, 2003, pp. 841–850. URL: http://dl.acm.org/citation.cfm?id=644108.644250 (cit. on p. 20).

[55] B. Langmead et al. "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome". In: *Genome Biology* 10.3 (2009), R25 (cit. on pp. 21, 31).

[56] H. Li and R. Durbin. "Fast and accurate short read alignment with Burrows-Wheeler transform". In: *Bioinformatics* 25.14 (2009), pp. 1754–1760 (cit. on pp. 21, 31).

[57] J. Simpson and R. Durbin. "Efficient de novo assembly of large genomes using compressed data structures". In: *Genome Research* 22.3 (2012), pp. 549–556. DOI: 10.1101/gr.126953.111. URL: http://genome.cshlp.org/content/22/3/549.abstract (cit. on pp. 21, 31).

[58] L. Russo et al. "Approximate String Matching with Compressed Indexes". In: *Algorithms* 2.3 (2009), pp. 1105–1136. DOI: 10.3390/a2031105. URL: http://www.mdpi.com/1999-4893/2/3/1105 (cit. on pp. 21, 31, 33).

[59] T. Schnattinger, E. Ohlebusch, and S. Gog. "Bidirectional search in a string with wavelet trees and bidirectional matching statistics". In: *Information and Computation* 213 (2012), pp. 13–22 (cit. on pp. 21, 31, 33).

[60] D. Belazzougui et al. "Versatile Succinct Representations of the Bidirectional Burrows-Wheeler Transform". In: *Proc. 21st European Symposium on Algorithms (ESA)*. 2013, pp. 133–144 (cit. on pp. 21, 31, 33).

[61] K. Břinda. "Novel computational techniques for mapping and classifying Next-Generation Sequencing data". PhD thesis. Université Paris-Est, 2016. DOI: DOI:10.5281/zenodo.1045316. URL: http://brinda.cz/publications/brinda_phd.pdf (cit. on pp. 26, 80, 94).

[62] H. Li and N. Homer. "A survey of sequence alignment algorithms for next-generation sequencing". In: *Briefings in Bioinformatics* 11.5 (2010), pp. 473–483. DOI: 10.1093/bib/bbq015 (cit. on p. 26).

[63] P. Ribeca. "Short-Read Mapping". In: *Bioinformatics for High Throughput Sequencing*. New York, NY: Springer New York, 2012, pp. 107–125. DOI: 10.1007/978-1-4614-0782-9_7 (cit. on p. 26).

[64] S. Canzar and S. L. Salzberg. "Short Read Mapping: An Algorithmic Tour". In: *Proceedings of the IEEE* (2015), pp. 1–23. DOI: 10.1109/JPROC.2015.2455551 (cit. on p. 26).

[65] R. S. Boyer and J. S. Moore. "A Fast String Searching Algorithm". In: *Commun. ACM* 20.10 (Oct. 1977), pp. 762–772. DOI: 10.1145/359842.359859. URL: http://doi.acm.org/10.1145/359842.359859 (cit. on p. 26).

[66] D. E. Knuth, J. James H. Morris, and V. R. Pratt. "Fast Pattern Matching in Strings". In: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350. DOI: 10.1137/0206024. URL: https://doi.org/10.1137/0206024 (cit. on p. 26).

[67] M. J. Fischer and M. S. Paterson. "String-matching and other products". In: *Symposium on Complexity of Computation: SIAM-AMS Proceedings Volume 7 1974*. Vol. 7. American Mathematical Society. 1974, pp. 113–125 (cit. on p. 27).

[68] Z. Galil and R. Giancarlo. "Improved string matching with k mismatches". In: *ACM SIGACT News* 17.4 (1986), pp. 52–54 (cit. on p. 27).

[69] A. Amir, M. Lewenstein, and E. Porat. "Faster algorithms for string matching with k mismatches". In: *Journal of Algorithms* 50.2 (2004), pp. 257–275 (cit. on p. 27).

[70] G. M. Landau and U. Vishkin. "Fast string matching with k differences". In: *Journal of Computer and System Sciences* 37.1 (1988), pp. 63–78 (cit. on p. 27).

[71] Z. Galil and K. Park. "An improved algorithm for approximate string matching". In: *Automata, Languages and Programming: 16th International Colloquium Stresa, Italy, July 11–15, 1989 Proceedings*. Ed. by G. Ausiello, M. Dezani-Ciancaglini, and S. R. Della Rocca. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 394–404. DOI: 10.1007/BFb0035772. URL: https://doi.org/10.1007/BFb0035772 (cit. on p. 27).

[72] S. Henikoff and J. G. Henikoff. "Amino acid substitution matrices from protein blocks." In: *Proceedings of the National Academy of Sciences* 89.22 (1992), pp. 10915–10919. DOI: 10.1073/pnas.89.22.10915 (cit. on p. 27).

[73] S. F. Altschul. "Amino acid substitution matrices from an information theoretic perspective". In: *Journal of Molecular Biology* 219.3 (1991), pp. 555–565. DOI: 10.1016/0022-2836(91)90193-A (cit. on p. 27).

[74] S. B. Needleman and C. D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453. DOI: 10.1016/0022-2836(70)90057-4. URL: http://www.ncbi.nlm.nih.gov/pubmed/5420325%20http://linkinghub.elsevier.com/retrieve/pii/0022283670900574 (cit. on p. 28).

[75] T. F. Smith and M. S. Waterman. "Identification of common molecular subsequences." In: *Journal of molecular biology* 147.1 (1981), pp. 195–7. DOI: 10.1016/0022-2836(81)90087-5. URL: http://www.sciencedirect.com/science/article/pii/0022283681900875%20http://www.sciencedirect.com/science/article/pii/0022283681900875/pdf?md5=c13489402e4edce622fe841d3a3e0940%7B%5C&%7Dpid=1-s2.0-0022283681900875-main.pdf%20http://linkinghub.elsevier.com/retrieve/pii/0022283681900875%20http://www.ncbi.nlm.nih.gov/pubmed/7265238 (cit. on p. 28).

[76] G. Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming". In: *Journal of the ACM* 46.3 (1999), pp. 395–415. DOI: 10.1145/316542.316550 (cit. on p. 28).

[77] W. R. Pearson and D. J. Lipman. "Improved tools for biological sequence comparison." In: *Proceedings of the National Academy of Sciences of the United States of America* 85.8 (1988), pp. 2444–8. DOI: 10.1073/pnas.85.8.2444. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=280013%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract (cit. on pp. 29, 55).

[78] S. F. Altschul et al. "Basic local alignment search tool". In: *Journal of Molecular Biology* 215.3 (1990), pp. 403–410. DOI: 10.1016/S0022-2836(05)80360-2. URL: http://www.ncbi.nlm.nih.gov/pubmed/2231712%20http://linkinghub.elsevier.com/retrieve/pii/S0022283605803602 (cit. on pp. 29, 55).

[79] R. Li et al. "SOAP: Short oligonucleotide alignment program". In: *Bioinformatics* 24.5 (2008), pp. 713–714. DOI: 10.1093/bioinformatics/btn025 (cit. on pp. 29, 30).

[80] R. Li et al. "SOAP2: an improved ultrafast tool for short read alignment". en. In: *Bioinformatics* 25.15 (2009), pp. 1966–1967. DOI: 10.1093/bioinformatics/btp336. URL: http://www.ncbi.nlm.nih.gov/pubmed/19497933%20http://bioinformatics.oxfordjournals.org/cgi/doi/10.1093/bioinformatics/btp336 (cit. on pp. 29, 31).

[81] W.-P. Lee et al. "MOSAIK: A Hash-Based Algorithm for Accurate Next-Generation Sequencing Short-Read Mapping". In: *PLoS ONE* 9.3 (Mar. 5, 2014), e90581+. DOI: 10.1371/journal.pone.0090581. URL: http://dx.doi.org/10.1371/journal.pone.0090581 (cit. on p. 29).

[82] N. Homer, B. Merriman, and S. F. Nelson. "BFAST: an alignment tool for large scale genome resequencing." In: *PloS one* 4.11 (2009), e7767. DOI: 10.1371/journal.pone.0007767. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2770639%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract (cit. on p. 29).

[83] H. Li, J. Ruan, and R. Durbin. "Mapping short DNA sequencing reads and calling variants using mapping quality scores." In: *Genome research* 18.11 (2008), pp. 1851–1858. DOI: 10.1101/gr.078212.108. URL: http://genome.cshlp.org/cgi/doi/10.1101/gr.078212.108%20http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2577856%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract (cit. on pp. 29, 30).

[84] A. D. Smith et al. "Updates to the RMAP short-read mapping software". In: *Bioinformatics* 25.21 (2009), pp. 2841–2842. DOI: 10.1093/bioinformatics/btp533 (cit. on pp. 29, 30).

[85] H. Lin et al. "ZOOM! Zillions of oligos mapped." en. In: *Bioinformatics* 24.21 (2008), pp. 2431–7. DOI: 10.1093/bioinformatics/btn416. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2732274%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract (cit. on pp. 29, 30).

[86] M. David et al. "SHRiMP2: Sensitive yet Practical Short Read Mapping". en. In: *Bioinformatics* 27.7 (2011), pp. 1011–1012. DOI: 10.1093/bioinformatics/btr046. URL: http://www.ncbi.nlm.nih.gov/pubmed/21278192%20http://bioinformatics.oxfordjournals.org/cgi/doi/10.1093/bioinformatics/btr046 (cit. on p. 29).

[87] L. Noé and G. Kucherov. "YASS: enhancing the sensitivity of DNA similarity search." In: *Nucleic acids research* 33.Web Server issue (2005), W540–3. DOI: 10.1093/nar/gki478 (cit. on p. 29).

[88] B. Ma, J. Tromp, and M. Li. "PatternHunter: faster and more sensitive homology search". en. In: *Bioinformatics* 18.3 (2002), pp. 440–445. DOI: 10.1093/bioinformatics/18.3.440. URL: http://bioinformatics.oxfordjournals.org/cgi/doi/10.1093/bioinformatics/18.3.440 (cit. on p. 30).

[89] S. Burkhardt and J. Kärkkäinen. "Better Filtering with Gapped q-Grams". In: *Combinatorial Pattern Matching: 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1–4, 2001 Proceedings*. Ed. by A. Amir. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 73–85. DOI: 10.1007/3-540-48194-X_6. URL: http://dx.doi.org/10.1007/3-540-48194-X_6 (cit. on pp. 30, 55).

[90] H. Jiang and W. H. Wong. "SeqMap: Mapping massive amount of oligonucleotides to the genome". In: *Bioinformatics* 24.20 (2008), pp. 2395–2396. DOI: 10.1093/bioinformatics/btn429 (cit. on p. 30).

[91] G. KUCHEROV, L. NOÉ, and M. ROYTBERG. "A UNIFYING FRAMEWORK FOR SEED SENSITIVITY AND ITS APPLICATION TO SUBSET SEEDS". In: *Journal of Bioinformatics and Computational Biology* 04.02 (2006), pp. 553–569. DOI: 10.1142/S0219720006001977. URL: http://www.worldscientific.com/doi/abs/10.1142/S0219720006001977 (cit. on p. 30).

[92] G. Navarro and V. Mäkinen. "Compressed full-text indexes". In: *ACM Computing Surveys* 39.1 (2007) (cit. on p. 31).

[93] B. Langmead et al. "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome." en. In: *Genome biology* 10.3 (2009), R25. DOI: 10.1186/gb-2009-10-3-r25. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2690996%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract (cit. on pp. 31, 75).

[94] H. Li. *poster Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM*. 2013 (cit. on p. 31).

[95] T. W. Lam et al. "Compressed indexing and local alignment of DNA". In: *Bioinformatics* 24.6 (2008), pp. 791–797. DOI: 10.1093/bioinformatics/btn032 (cit. on p. 31).

[96] H. Li and R. Durbin. "Fast and accurate long-read alignment with Burrows-Wheeler transform". In: *Bioinformatics* 26.5 (2010), pp. 589–595. DOI: 10.1093/bioinformatics/btp698 (cit. on p. 31).

[97] H. Li and R. Durbin. "Fast and accurate short read alignment with Burrows-Wheeler transform". en. In: *Bioinformatics* 25.14 (2009), pp. 1754–1760. DOI: 10.1093/bioinformatics/btp324. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2705234%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract%20http://bioinformatics.oxfordjournals.org/cgi/doi/10.1093/bioinformatics/btp324 (cit. on pp. 31, 75).

[98] L. H. Y. Chen. "Poisson approximation for dependent trials". In: *The Annals of Probability* 3.3 (1975), pp. 534–545 (cit. on p. 36).

[99] A. D. Barbour, L. Holst, and S. Janson. *Poisson approximation*. Clarendon Press Oxford, 1992 (cit. on p. 36).

[100] S. Mihov and K. U. Schulz. "Fast approximate search in large dictionaries". In: *Computational Linguistic* 30.4 (2004), pp. 451–477 (cit. on p. 38).

[101] J. Kärkkäinen and J. C. Na. "Faster Filters for Approximate String Matching". In: *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2007, pp. 84–90 (cit. on p. 38).

[102] Z. Li et al. "Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph". In: *Briefings in Functional Genomics* 11.1 (2012), p. 25. DOI: 10.1093/bfgp/elr035. URL: +%20http://dx.doi.org/10.1093/bfgp/elr035 (cit. on p. 53).

[103] W. Zhang et al. "A Practical Comparison of De Novo Genome Assembly Software Tools for Next-Generation Sequencing Technologies". In: *PLoS ONE* 6.3 (Mar. 14, 2011), e17915+. DOI: 10.1371/journal.pone.0017915. URL: http://dx.doi.org/10.1371/journal.pone.0017915 (cit. on p. 53).

[104] S. L. Salzberg et al. "GAGE: A critical evaluation of genome assemblies and assembly algorithms". In: *Genome Research* 22.3 (Dec. 6, 2011), pp. 557–567. DOI: 10.1101/gr.131383.111. URL: http://dx.doi.org/10.1101/gr.131383.111 (cit. on p. 53).

[105] K. R. Bradnam et al. "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species". In: *GigaScience* 2.1 (2013), p. 1. DOI: 10.1186/2047-217X-2-10. URL: +%20http://dx.doi.org/10.1186/2047-217X-2-10 (cit. on p. 53).

[106] J. R. Miller, S. Koren, and G. Sutton. "Assembly algorithms for next-generation sequencing data". In: *Genomics* 95.6 (June 2010), pp. 315–327 (cit. on p. 54).

[107] R. Staden. "A mew computer method for the storage and manipulation of DNA gel reading data". In: *Nucleic Acids Research* 8.16 (1980), p. 3673. DOI: 10.1093/nar/8.16.3673. URL: +%20http://dx.doi.org/10.1093/nar/8.16.3673 (cit. on p. 54).

[108] S. Batzoglou et al. "ARACHNE: a whole-genome shotgun assembler." In: *Genome research* 12.1 (Jan. 1, 2002), pp. 177–189. DOI: 10.1101/gr.208902. URL: http://dx.doi.org/10.1101/gr.208902 (cit. on pp. 54, 55).

[109] E. W. Myers et al. "A Whole-Genome Assembly of Drosophila". In: *Science* 287.5461 (Mar. 24, 2000), pp. 2196–2204. DOI: 10.1126/science.287.5461.2196. URL: http://dx.doi.org/10.1126/science.287.5461.2196 (cit. on pp. 54, 55).

[110] X. Huang and A. Madan. "CAP3: A DNA sequence assembly program." In: *Genome research* 9.9 (Sept. 1, 1999), pp. 868–877. DOI: 10.1101/gr.9.9.868. URL: http://dx.doi.org/10.1101/gr.9.9.868 (cit. on pp. 54, 55).

[111] X. Huang et al. "PCAP: A whole-genome assembly program". In: *Genome Res.* 13.9 (2003), pp. 2164–2170 (cit. on pp. 54, 55).

[112] M. de la Bastide and W. R. McCombie. "Assembling genomic DNA sequences with PHRAP." In: *Current protocols in bioinformatics / editoral board, Andreas D. Baxevanis ... [et al.]* Chapter 11 (Mar. 2007). DOI: 10.1002/0471250953.bi1104s17. URL: http://dx.doi.org/10.1002/0471250953.bi1104s17 (cit. on p. 54).

[113] J. C. Mullikin and Z. Ning. "The Phusion Assembler". In: *Genome Research* 13.1 (Jan. 1, 2003), pp. 81–90. DOI: 10.1101/gr.731003. URL: http://dx.doi.org/10.1101/gr.731003 (cit. on p. 54).

[114] M. Margulies et al. "Genome sequencing in microfabricated high-density picolitre reactors". In: *Nature* 437.7057 (July 31, 2005), pp. 376–380. DOI: 10.1038/nature03959. URL: http://dx.doi.org/10.1038/nature03959 (cit. on p. 54).

[115] J. T. Simpson and R. Durbin. "Efficient de novo assembly of large genomes using compressed data structures". In: *Genome Research* 22.3 (Mar. 1, 2012), pp. 549–556. DOI: 10.1101/gr.126953.111. URL: http://dx.doi.org/10.1101/gr.126953.111 (cit. on p. 55).

[116] N. Välimäki, S. Ladra, and V. Mäkinen. "Approximate all-pairs suffix/prefix overlaps". In: *Information and Computation* 213 (2012), pp. 49–58. DOI: http://dx.doi.org/10.1016/j.ic.2012.02.002. URL: http://www.sciencedirect.com/science/article/pii/S0890540112000260 (cit. on p. 55).

[117] J. Kärkkäinen and J. C. Na. "Faster Filters for Approximate String Matching". In: *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 84–90. DOI: 10.1137/1.9781611972870.8. URL: http://epubs.siam.org/doi/abs/10.1137/1.9781611972870.8 (cit. on p. 55).

[118] G. Kucherov and D. Tsur. "Improved Filters for the Approximate Suffix-Prefix Overlap Problem". In: *String Processing and Information Retrieval: 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*. Ed. by E. Moura and M. Crochemore. Cham: Springer International Publishing, 2014, pp. 139–148. DOI: 10.1007/978-3-319-11918-2_14. URL: http://dx.doi.org/10.1007/978-3-319-11918-2_14 (cit. on p. 55).

[119] P. A. Pevzner, H. Tang, and M. S. Waterman. "An Eulerian path approach to DNA fragment assembly". In: *Proc. Natl. Acad. Sci. U.S.A.* 98.17 (Aug. 2001), pp. 9748–9753 (cit. on p. 55).

[120] M. G. Grabherr et al. "Full-length transcriptome assembly from RNA-Seq data without a reference genome". In: *Nat Biotech* 29.7 (July 2011), pp. 644–652 (cit. on p. 55).

[121] G. Sacomoto et al. "KISSPLICE: de-novo calling alternative splicing events from RNA-seq data". In: *BMC Bioinformatics* 13.Suppl 6 (2012), S5 (cit. on p. 55).

[122] Y. Peng et al. "Meta-IDBA: a de Novo assembler for metagenomic data". In: *Bioinformatics* 27.13 (2011), pp. i94–i101 (cit. on p. 55).

[123] Z. Iqbal et al. "De novo assembly and genotyping of variants using colored de Bruijn graphs". In: *Nat. Genet.* 44.2 (Feb. 2012), pp. 226–232 (cit. on p. 55).

[124] T. Conway and A. Bromage. "Succinct data structures for assembling large genomes". In: *Bioinformatics* 27.4 (2011), pp. 479–486 (cit. on p. 55).

[125] C. Ye et al. "Exploiting sparseness in de novo genome assembly". In: *BMC Bioinformatics* 13.Suppl 6 (2012), S1. URL: http://www.biomedcentral.com/1471-2105/13/S6/S1 (cit. on pp. 55, 56).

[126] R. Chikhi and G. Rizk. "Space-Efficient and Exact de Bruijn Graph Representation Based on a Bloom Filter". In: *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*. Ed. by B. J. Raphael and J. Tang. Vol. 7534. Lecture Notes in Computer Science. Berlin: Springer, 2012, pp. 236–248 (cit. on pp. 55–60, 62–66, 96).

[127] A. Bowe et al. "Succinct de Bruijn Graphs". In: *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*. Ed. by B. Raphael and J.Tang. Vol. 7534. Lecture Notes in Computer Science. Berlin: Springer, 2012, pp. 225–235 (cit. on pp. 55, 56).

[128] J. Pell et al. "Scaling metagenome sequence assembly with probabilistic de Bruijn graphs". In: *Proc. Natl. Acad. Sci. U.S.A.* 109.33 (Aug. 2012), pp. 13272–13277 (cit. on pp. 55, 56).

[129] E. Drezen et al. "GATB: Genome Assembly & Analysis Tool Box". In: *Bioinformatics* 30.20 (Oct. 15, 2014), pp. 2959–2961. DOI: 10.1093/bioinformatics/btu406. URL: http://dx.doi.org/10.1093/bioinformatics/btu406 (cit. on p. 56).

[130] C. Boucher et al. "Variable-Order De Bruijn Graphs". In: *Proceedings of the 2015 Data Compression Conference*. DCC '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 383–392. DOI: 10.1109/DCC.2015.70. URL: http://dx.doi.org/10.1109/DCC.2015.70 (cit. on p. 56).

[131] D. Belazzougui et al. "Bidirectional Variable-Order de Bruijn Graphs". In: *LATIN 2016: Theoretical Informatics - 12th Latin American Symposium, Ensenada, Mexico, April 11-15, 2016, Proceedings*. 2016, pp. 164–178. DOI: 10.1007/978-3-662-49529-2_13. URL: http://dx.doi.org/10.1007/978-3-662-49529-2_13 (cit. on p. 56).

[132] R. Chikhi et al. "On the Representation of De Bruijn Graphs". In: *Proceedings of the 18th Annual International Conference on Research in Computational Molecular Biology - Volume 8394*. RECOMB 2014. Pittsburgh, PA, USA: Springer-Verlag New York, Inc., 2014, pp. 35–55. DOI: 10.1007/978-3-319-05269-4_4. URL: http://dx.doi.org/10.1007/978-3-319-05269-4_4 (cit. on p. 56).

[133] D. Belazzougui et al. "Fully Dynamic de Bruijn Graphs". In: *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*. 2016, pp. 145–152. DOI: 10.1007/978-3-319-46049-9_14. URL: http://dx.doi.org/10.1007/978-3-319-46049-9_14 (cit. on p. 56).

[134] G. Rizk, D. Lavenier, and R. Chikhi. "DSK: k-mer counting with very low memory usage". In: *Bioinformatics* 29.5 (Mar. 2013), pp. 652–3 (cit. on p. 62).

[135] F. R. Blattner et al. "The Complete Genome Sequence of Escherichia coli K-12". In: *Science* 277.5331 (1997), pp. 1453–1462 (cit. on p. 63).

[136] E. Porat. "An Optimal Bloom Filter Replacement Based on Matrix Solving". In: *Computer Science - Theory and Applications, Fourth International Computer Science Symposium in Russia, CSR 2009, Novosibirsk, Russia, August 18-23, 2009. Proceedings*. Vol. 5675. Lecture Notes in Computer Science. Berlin: Springer, 2009, pp. 263–273 (cit. on pp. 66, 96).

[137] R. Rozov, R. Shamir, and E. Halperin. "Fast lossless compression via cascading Bloom filters". In: *BMC Bioinformatics* 15.9 (2014), S7. DOI: 10.1186/1471-2105-15-S9-S7. URL: http://dx.doi.org/10.1186/1471-2105-15-S9-S7 (cit. on pp. 67, 68, 70–72, 96).

[138] J. K. Bonfield and M. V. Mahoney. "Compression of FASTQ and SAM Format Sequencing Data". In: *PLOS ONE* 8.3 (Mar. 2013), pp. 1–10. DOI: 10.1371/journal.pone.0059190. URL: https://doi.org/10.1371/journal.pone.0059190 (cit. on pp. 67, 68).

[139] M. H. Fritz et al. "Efficient storage of high throughput DNA sequencing data using reference-based compression". In: *Genome Research* 21.5 (May 1, 2011), pp. 734–740. DOI: 10.1101/gr.114819.110. URL: http://dx.doi.org/10.1101/gr.114819.110 (cit. on p. 67).

[140] D. C. Jones et al. *Compression of next-generation sequencing reads aided by highly efficient de novo assembly*. July 10, 2012. URL: http://arxiv.org/abs/1207.2424 (cit. on p. 67).

[141] C. Kingsford and R. Patro. "Reference-based compression of short-read sequences using path encoding". In: *Bioinformatics* 31 (2015). DOI: 10.1093/bioinformatics/btv071. URL: https://doi.org/10.1093/bioinformatics/btv071 (cit. on p. 67).

[142] C. Kozanitis et al. "Compressing Genomic Sequence Fragments Using SlimGene". In: *Research in Computational Molecular Biology*. Ed. by B. Berger. Vol. 6044. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2010. Chap. 20, pp. 310–324. DOI: 10.1007/978-3-642-12683-3_20. URL: http://dx.doi.org/10.1007/978-3-642-12683-3%5C_20 (cit. on p. 67).

[143] G. Benoit et al. "Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph". In: *BMC Bioinformatics* 16.1 (Sept. 14, 2015), p. 288. DOI: 10.1186/s12859-015-0709-7. URL: https://doi.org/10.1186/s12859-015-0709-7 (cit. on p. 67).

[144] A. J. Cox et al. "Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform". In: *Bioinformatics* 28.11 (2012), p. 1415. DOI: 10.1093/bioinformatics/bts173. URL: +%20http://dx.doi.org/10.1093/bioinformatics/bts173 (cit. on p. 67).

[145] S. Deorowicz and S. Grabowski. "Compression of dna sequence reads in fastq format". In: *Bioinformatics* 27 (2011). DOI: 10.1093/bioinformatics/btr014. URL: https://doi.org/10.1093/bioinformatics/btr014 (cit. on p. 67).

[146] S. Grabowski, S. Deorowicz, and Ł. Roguski. "Disk-based compression of data from genome sequencing". In: *Bioinformatics* 31 (2014) (cit. on p. 67).

[147] F. Hach et al. "SCALCE: boosting sequence compression algorithms using locally consistent encoding". In: *Bioinformatics* 28.23 (2012), p. 3051. DOI: 10.1093/bioinformatics/bts593. URL: +%20http://dx.doi.org/10.1093/bioinformatics/bts593 (cit. on p. 67).

[148] L. Janin, O. Schulz-Trieglaff, and A. J. Cox. "Beetl-fastq: a searchable compressed archive for dna reads". In: *Bioinformatics* 30 (2014) (cit. on p. 67).

[149] R. Patro and C. Kingsford. "Data-dependent bucketing improves reference-free compression of sequencing reads". In: *Bioinformatics* 31 (2015). DOI: 10.1093/bioinformatics/btv248. URL: https://doi.org/10.1093/bioinformatics/btv248 (cit. on p. 67).

[150] D. H. Huson et al. "Integrative analysis of environmental sequences using MEGAN4". In: *Genome Research* 21.9 (2011), pp. 1552–1560. DOI: 10.1101/gr.120618.111 (cit. on p. 75).

[151] N. Segata et al. "Metagenomic microbial community profiling using unique clade-specific marker genes". In: *Nat Meth* 9.8 (Aug. 2012), pp. 811–814. URL: http://dx.doi.org/10.1038/nmeth.2066 (cit. on pp. 75, 76).

[152] B. Liu et al. "MetaPhyler: Taxonomic profiling for metagenomic sequences". In: *2010 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2010, pp. 95–100. DOI: 10.1109/BIBM.2010.5706544 (cit. on p. 75).

[153] J. Pell et al. "Scaling metagenome sequence assembly with probabilistic de Bruijn graphs". In: *Proceedings of the National Academy of Sciences* 109.33 (2012), pp. 13272–13277. DOI: 10.1073/pnas.1121464109 (cit. on p. 76).

[154] J. Berendzen et al. "Rapid phylogenetic and functional classification of short genomic fragments with signature peptides". In: *BMC Research Notes* 5.1 (2012), p. 460. DOI: 10.1186/1756-0500-5-460 (cit. on p. 76).

[155] S. K. Ames et al. "Scalable metagenomic taxonomy classification using a reference genome database". In: *Bioinformatics* 29.18 (2013), pp. 2253–2260. DOI: 10.1093/bioinformatics/btt389 (cit. on p. 76).

[156] T. Tatusova et al. "RefSeq microbial genomes database: new representation and annotation strategy". In: *Nucleic Acids Research* 42.D1 (2014), pp. D553–D559. DOI: 10.1093/nar/gkt1274. URL: http://nar.oxfordjournals.org/lookup/doi/10.1093/nar/gkt1274 (cit. on pp. 76, 90, 93, 94).

[157] D. E. Wood and S. L. Salzberg. "Kraken: ultrafast metagenomic sequence classification using exact alignments." In: *Genome biology* 15.3 (2014), R46. DOI: 10.1186/gb-2014-15-3-r46. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4053813%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract%20http://genomebiology.biomedcentral.com/articles/10.1186/gb-2014-15-3-r46%20http://www.ncbi.nlm.nih.gov/pubmed/24580807%20http://www.pubmedcentral.nih.gov/arti (cit. on pp. 76, 94).

[158] D. Kim et al. "Centrifuge: rapid and sensitive classification of metagenomic sequences". In: *bioRxiv preprints* (2016). DOI: 10.1101/054965. URL: http://biorxiv.org/lookup/doi/10.1101/054965 (cit. on p. 76).

[159] B. Langmead and S. L. Salzberg. "Fast gapped-read alignment with Bowtie 2." en. In: *Nature methods* 9.4 (2012), pp. 357–9. DOI: 10.1038/nmeth.1923. URL: http://www.nature.com/nmeth/journal/v9/n4/abs/nmeth.1923.html%20http://www.ncbi.nlm.nih.gov/pubmed/22388286%20http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3322381%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract (cit. on p. 76).

[160] P. Menzel, K. L. Ng, and A. Krogh. "Fast and sensitive taxonomic classification for metagenomics with Kaiju". In: *Nature Communications* 7 (2016), p. 11257. DOI: 10.1038/ncomms11257. URL: http://www.nature.com/doifinder/10.1038/ncomms11257 (cit. on p. 77).

[161] H. Stranneheim et al. "Classification of DNA sequences using Bloom filters". In: *Bioinformatics* 26.13 (July 1, 2010), pp. 1595–1600. DOI: 10.1093/bioinformatics/btq230. URL: http://dx.doi.org/10.1093/bioinformatics/btq230 (cit. on p. 77).

[162] J. Kawulok and S. Deorowicz. "CoMeta: Classification of Metagenomes Using k-mers". In: *PLOS ONE* 10.4 (2015), e0121453. DOI: 10.1371/journal.pone.0121453 (cit. on p. 77).

[163] S. Lindgreen, K. L. Adair, and P. P. Gardner. "An evaluation of the accuracy and speed of metagenome analysis tools". In: *Scientific Reports* 6 (2016), p. 19233. DOI: 10.1038/srep19233. URL: http://www.nature.com/articles/srep19233 (cit. on p. 77).

[164] R. J. Randle-Boggis et al. "Evaluating techniques for metagenome annotation using simulated sequence data". In: *FEMS Microbiology Ecology* 92.7 (2016), fiw095. DOI: 10.1093/femsec/fiw095. URL: http://femsec.oxfordjournals.org/lookup/doi/10.1093/femsec/fiw095 (cit. on p. 77).

[165] D. O. Ricke, A. Shcherbina, and N. Chiu. "Evaluating performance of metagenomic characterization algorithms using in silico datasets generated with FASTQSim". In: *bioRxiv* (2016), p. 046532. DOI: 10.1101/046532. URL: http://biorxiv.org/content/early/2016/03/31/046532.abstract (cit. on p. 77).

[166] M. A. Peabody et al. "Evaluation of shotgun metagenomics sequence classification methods using in silico and in vitro simulated communities". In: *BMC Bioinformatics* 16.1 (2015), p. 363. DOI: 10.1186/s12859-015-0788-5. URL: http://www.biomedcentral.com/1471-2105/16/363 (cit. on p. 77).

[167] H. Vinje et al. "Comparing K-mer based methods for improved classification of 16S sequences". In: *BMC Bioinformatics* 16.1 (2015), p. 205. DOI: 10.1186/s12859-015-0647-4. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4487979%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract (cit. on p. 77).

[168] Pavlopoulos et al. "Metagenomics: Tools and Insights for Analyzing Next-Generation Sequencing Data Derived from Biodiversity Studies". In: *Bioinformatics and Biology Insights* (2015), p. 75. DOI: 10.4137/BBI.S12462. URL: http://www.la-press.com/metagenomics-tools-and-insights-for-analyzing-next-generation-sequenci-article-a4809 (cit. on p. 77).

[169] D. Clark. "Compact Pat Trees". PhD thesis. University Waterloo, 1996 (cit. on p. 89).

[170] J. I. Munro. "Tables". In: *16th FST TCS LNCS* 1180 (1996), pp. 37–42 (cit. on p. 89).