Pan-genome Search and Storage

by

Guillaume Holley

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor rerum naturalium
at the Faculty of Technology, Bielefeld University
February 2018

Referees:

Professor Doctor Jens Stoye
Doctor Faraz Hach
Doctor Rayan Chikhi

ACKNOWLEDGEMENTS

I would like to thank everyone who has been directly or indirectly involved in this thesis. I thank Jens Stoye for welcoming me as his student in the Genome Informatics group and providing me the guidance I needed to achieve this thesis. I am thankful to Roland Wittler for supporting my ideas and helping me out the numerous times I entered his office and said “Do you have five minutes?” while knowing pertinently it would take much longer. I also thank Faraz Hach from who I have learned a lot during my research visit in Vancouver.
My gratitude goes to the entire population of the U10 and V10 floors scattered in multiple groups, to the former members of the AGGI group I have met during the past few years and to the badminton gang. A big thank goes to all DiDy students for all these ping-pong games, cool retreats, dinners at the chinese restaurant, cinema and board game evenings. A special thank goes to Nina, my “office wife”, for sharing an office (and temporarily a house) with me and reading thoroughly this thesis. I could not have found a better office mate (thank you again Roland!) with whom I could play ping-pong, discuss scientific issues and share all my jokes of questionable quality. I would like to extend this thank to Tina who did not back down when she started to use the rustic first version of my code. This acknowledgement would be incomplete without thanking Pierre Peterlongo for introducing me to the awesome field of computational biology and sharing his contagious passion of the de Bruijn graph.
I would like to thank all my friends met before and during my time in Bielefeld. I cannot thank enough Monika and Kerstin for being such good friends, for helping me in so many occasions, for all these geocaching afternoons and sushi dinners. I am also especially grateful to Sebastien, Jean-Baptiste and Paul for staying in touch after I arrived in Bielefeld while we did not see each others for a long time. I am thankful to my family for the enormous support provided during all these years and for visiting me numerous times. Particularly, I thank my parents who taught me creativity and perseverance. Who would have guessed that installing

an old computer in my room fifteen years ago would have resulted in the present thesis?

Clearly, nothing of this would have been possible without the overwhelming support and the unlimited patience of Violette. The past few years have been a hell of an adventure and I am so glad we did it together.

Finally, I would like to acknowledge funding from the International Research Training Group GRK/1906 “Computational Methods for the Analysis of the Diversity and Dynamics of Genomes”, also known as DiDy, during three years and funding from the Bielefeld Genome Informatics group and Faculty of Technology scholarship for the past few months.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS v

LIST OF FIGURES x

LIST OF TABLES xiv

ABSTRACT xv

CHAPTER

I. Introduction 1

1.1 Biological Background 1

1.1.1 Preliminaries 1

1.1.2 DNA Sequencing 3

1.1.3 *de novo* Read Assembly 6

1.1.4 Comparative Genomics 6

1.1.5 Pan-genomes 7

1.2 Computational Background 10

1.2.1 Preliminaries 11

1.2.2 Graphs 11

1.2.3 Trees 13

1.2.4 Hash Tables 19
1.2.5 Bloom Filters 23
1.2.6 Burrows-Wheeler Tranform 26
1.2.7 FM-Index 29
1.3 Thesis Overview 30

II. Methods and Tools for Pan-genome Indexing 31

2.1 Introduction 31
2.2 Reference-based Methods 32
2.3 Alignment-based Methods 35
2.4 Graph-based Methods 36
2.5 de Bruijn graph Methods 37
2.6 k-mer based Methods 40

III. Pan-genome Indexing 42

3.1 Introduction 42
3.2 The Bloom Filter Trie 43
3.2.1 Uncompressed Container 44
3.2.2 Compressed Container 45
3.2.3 Color Set 49
3.3 Operations 49
3.3.1 Container Insertion 50
3.3.2 Tree Insertion 52
3.3.3 Container Look-up 53

3.3.4 Tree Look-up 54
3.4 Successors and Predecessors Traversing 55
3.5 Evaluation 58
3.5.1 *P. aeruginosa* Dataset 61
3.5.2 Human Tissues Dataset 63
3.6 Conclusion 65

IV. Pan-genome Storage 67

4.1 Introduction 67
4.1.1 Existing Approaches 68
4.1.2 Contributions 70
4.2 The guided de Bruijn Graph 70
4.3 Compression 73
4.3.1 Read Clustering and Merging. 74
4.3.2 Spanning Super Read Encoding 76
4.3.3 Partition Encoding 77
4.3.4 Meta Data and gdBG Compression 80
4.4 Update and Decompression 80
4.5 Results 80
4.6 Conclusion 86

V. Conclusion 87

5.1 Perspectives 88

BIBLIOGRAPHY 91

LIST OF FIGURES

Figure

1.1 Representation of a DNA molecule. Each strand is shown with a different grey color and their respective extremities are annotated with “3’” and “5’” to indicate their orientation. Nucleotides located on the strands are illustrated with a white color and are annotated with the symbol of the nucleobase they represent. Grey colored links between nucleotides represent the hydrogen bonds. 2

1.2 Comparison of sequencers based on their sequencing capacity and the length of the reads they produce (Nederbragt, 2016). . . 4

1.3 Annual sequencing cost of the human genome. The plot was produced by the National Human Genome Research Institute and is available at **www.genome.gov/sequencingcostsdata/**. 5

1.4 Representation of a pan-genome with 3 strains. 8

1.5 Trie of strings “aa”, “aba”, “abb”, “ba” and “bba” for an alphabet $\mathcal{A} = \{a, b\}$. 15

1.6 Compaction methods for tries of strings “aa”, “aba”, “abb”, “ba” and “bba” for an alphabet $\mathcal{A} = \{a, b\}$. 16

1.7 Suffix trie (a) and tree (b) of string “abaab$”. 17

1.8 Suffix array of string “abaab$” for alphabet order $ < a < b. . . 18

1.9 Burst trie of strings “aa”, “abb”, “ba” and “bba” for an alphabet $\mathcal{A} = \{a, b\}$. Leaves have a capacity of two suffixes. Inserting string “aba” in the burst trie of (a) triggers a bursting of the left leaf, resulting in the burst trie setup shown in (b). A terminal symbol is added at the end of each string in order to handle the case of one string which is the prefix of another string. 20

1.10 Hash table with collisions. 21

1.11 Hash table with chaining. 22

1.12 Open addressing methods. 23

1.13 Insertion of two items e_1 and e_2 into a BF. 24

1.14 Query of two items. Item e_3 is a true negative and item e_4 is a false positive. 24

3.1 Insertion of six suffixes (that are here complete k-mers) with different colors into a BFT with $k = 12$, $l = 4$ and $\phi = 5$. In (a), the first five suffixes are inserted at the root into an uncompressed container *uc*. When a sixth suffix “gcgccaggaatc” is inserted, *uc* exceeds its capacity and is burst, resulting in the BFT structure shown in (b) with one compressed container and four uncompressed containers. Note that in practice, container vertices might have more than one container and suffixes might have more than one color. 45

3.2 BF of four edge labels “aggc”, “ctca”, “gccc” and “gcgc” with $f = 2$ hash functions h_1 and h_2. 46

3.3 Tree representation of four edge labels “aggc”, “ctca”, “gccc” and “gcgc”. 47

3.4 Exact representation of four edge labels “aggc”, “ctca”, “gccc” and “gcgc” in a compressed container with $|label_p| = 2$ and $|label_s| = 2$. 48

3.5 Insertion of edge label “gtat” in a compressed container with four edge labels “aggc”, “ctca”, “gccc” and “gcgc”. Inserted or changed parts are highlighted. Array *edge* is not represented. . . 52

3.6 Traversed paths for predecessors $pred_a$ of k-mer x = “aggctat-gctca” such that $pred_a = a \odot x(1, |x| - 1)$ for all $a \in \mathcal{A}$. Content of vertices is only shown for the root, other vertices only have one traversed container and are represented with an empty rounded box. Color set vertices are not represented. 57

3.7 Traversed paths for successors $succ_a$ of k-mer x = “aggctatgctca” such that $succ_a = x(2, |x| - 1) \odot a$ for all $a \in \mathcal{A}$. Content of vertices is only shown for the leaves, other vertices have only one traversed container and are represented with an empty rounded box. Color set vertices are not represented. 58

3.8 Internal representation of a compressed container with five edge labels “aggc”, “ctca”, “gccc”, “gcgc”, “gtat”, adapted as “ggca”, “tcac”, “cccg”, “cgcg”,“tatg”, respectively, for predecessor and successor traversal. Array *edge* is not represented. 59

3.9 Insertion of edge label “tggc” adapted as “ggct” in a compressed container with edge labels “aggc”, “ctca”, “gccc”, “gcgc” and “gtat”. Array *edge* is not represented. Inserted parts are highlighted. 59

4.1 The gdBG of sequence $S =$ “cgtaagtaat” as constructed by Algorithm 5 with $k = 3$. 72

4.2 The gdBG of sequence $S =$ cgtaagtaat using 3-mers overlapping on $k - l = 1$. The last symbol of S is not encoded in the gdBG as it cannot be part of a k-mer. 73

4.3 Minimizer clustering of 4 reads. Minimizers of length 2 are underlined. For the sake of convenience, the reverse-complement is not considered. 74

4.4 Merging of two reads into a super read. Minimizers of length 2 are underlined. For the sake of convenience, the reverse-complement is not considered. 75

4.5 Merging of three super reads into an SSR “acgttgatt”. Minimizers of length 2 are underlined with a dashed line. Secondary minimizers (for merging) are underlined with a plain line. For the sake of convenience, the reverse-complement is not considered. . 75

4.6 Extraction of 4-mers overlapping on $k - l = 2$ from two similar SSRs, ssr_1 and ssr_2. 76

4.7 The gdBG of SSRs $ssr_1 =$ “acgtac” and $ssr_2 =$ “tccttc” using 4-mers ($l = 2$). Dotted edges are false implicit edges. The labeled solid edge exists by using the starting overlap of ssr_2 after the traversal of ssr_1, as described in Section 4.3.2. 79

4.8 Compression ratios in paired-end mode (left) and single-end mode (right). 82

4.9 Disk sizes in paired-end mode (left) and single-end mode (right). 82

4.10 DARRC disk size distribution in paired-end mode (left) and single-end mode (right). 83

4.11 Compression times in paired-end mode (left) and single-end mode (right). . . . 84

4.12 Decompression times in paired-end mode (left) and single-end mode (right). . . . 84

4.13 Compression main memory peaks in paired-end mode (left) and single-end mode (right). . . . 85

4.14 Decompression main memory peaks in paired-end mode (left) and single-end mode (right). . . . 85

LIST OF TABLES

Table

1.1 BWT computation of string “abaab$”. The computed BWT is highlighted. . . . 27

1.2 Reconstruction of a string from the BWT string “bba$aa”. The reconstructed string is highlighted. . . . 28

1.3 FM-Index of the BWT string “bba$aa”. . . . 29

3.1 BFT and SBT construction evaluation for 473 *P. aeruginosa* isolates. Best results are highlighted. . . . 61

3.2 SBT and BFT querying evaluation for sequences of length 100 bp from the sequencing experiment ERR431077. Best results are highlighted. . . . 62

3.3 Evaluation of k-mer extraction and branching queries for a BFT constructed from 473 *P. aeruginosa* isolates. . . . 63

3.4 BFT and SBT construction evaluation for 2,652 human tissue RNA-Seq experiments. Best results are highlighted. . . . 64

3.5 SBT and BFT querying evaluation for the GENCODE sequences database. Best results are highlighted. . . . 65

3.6 Evaluation of k-mer extraction and branching queries for a BFT constructed from 2,652 human tissue RNA-Seq experiments. . . . 65

4.1 Naive representation of a partition set composed of integers 12534, 12567 and 28911. . . . 79

4.2 Delta and Vbyte encoded representation of the same partition set used in Table 4.1. . . . 79

ABSTRACT

Pan-genome Search and Storage
by
Guillaume Holley

High Throughput Sequencing (HTS) technologies are constantly improving and making genome sequencing more affordable. However, HTS sequencers can only produce short overlapping genome fragments that are erroneous and cover the sequenced genomes unevenly. These genome fragments are assembled based on their overlaps to produce larger contiguous sequences. Since *de novo* genome assembly is computationally intensive, some species have a reference genome used as a guide for assembling genome fragments from the same species or as a basis for comparative genomics methods. Yet, assembling a genome is an error-prone process depending on the quality of the sequencing data and the heuristics used during the assembly. Furthermore, analyses based on a reference are biased towards the reference. Finally, a single reference cannot reflect the dynamics and diversity of a population of genomes. Overcoming these issues requires to move away from the single-genome reference-centric paradigm and take advantage of the multiple sequenced genomes available for each species. For this purpose, pan-genomes were introduced as sets of genomes from different strains of the same species. A pan-genome is represented by a multi-genome index exploiting the similarity and redundancy of the genomes it contains. Still, pan-genomes are more difficult to analyze than single genomes because of the large amount of data to be stored and indexed.

Current data structures for pan-genome indexing do not fulfill all requirements for pan-genome analysis. Indeed, these data structures are often immutable while

the size of a pan-genome grows constantly with newly sequenced genomes. Frequently, these data structures consider only assemblies as input, while unassembled genome fragments abound in databases. Also, indexing variants and similarities between the genomes of a pan-genome usually requires time and memory consuming algorithms such as sequence alignments. Sometimes, pan-genome analysis tools just assume variants and similarities are provided as input. While data structures already exist for pan-genome indexing, no solution is currently proposed for genome fragment compression in a pan-genome context. Indeed, it is often of interest to transmit and store all genome fragments of a pan-genome. However, HTS-specific compression tools are not dynamic and cannot update a compressed archive of genome fragments with new fragments of a genome without decompression. Hence, those tools are poorly adapted to the transmission and storage of genome fragments in a pan-genome context.
In this thesis, we aim to provide scalable solutions for pan-genome indexing and storage. We first address the problem of pan-genome indexing by proposing a new alignment-free, reference-free and incremental data structure that considers genome fragments as well as assemblies in input: the Bloom Filter Trie (BFT). The BFT is a tree data structure representing a colored de Bruijn graph in which k-mers, words of length k from the input genomes, are associated with sets of colors representing the genomes in which they occur. The BFT makes extensive use of Bloom filters to navigate in the tree and optimize the graph traversal. A "bursting" method is employed to perform an efficient path and level compaction of the tree. We show that the BFT outperforms a data structure that has similar features but is based on an approximation of the set of indexed k-mers.
Secondly, we address the problem of genome fragments compression in a pan-genome context by proposing a new abstract data structure, the guided de Bruijn graph. It augments the de Bruijn graph with k-mer partitions such that the graph traversal is guided to reconstruct exactly the genome fragments when decompressing. Different techniques are proposed to optimize the storage of fragments in the graph and the partition encoding. We show that the BFT described previously has all features required to index a guided de Bruijn graph and is used in the implementation of our compression method named DARRC. The evaluation of DARRC on a large pan-genome dataset compared to state-of-the-art HTS-specific and general purpose compression tools shows a 30 % compression ratio improvement over the second best performing tool of this evaluation.

CHAPTER I

Introduction

1.1 Biological Background

Section 1.1.1 introduces the preliminary biological notions that will be used in defining the problems this thesis aims to solve. Section 1.1.2 describes how genomic data are acquired and Section 1.1.3 details the complexity of transforming these data in order to analyze them. Then, Section 1.1.4 shows which analyses can be carried out on these data and what are the current limitations. Finally, Section 1.1.5 presents a new representation of genomic data from different sources to overcome these limitations.

1.1.1 Preliminaries

DNA (DeoxyriboNucleic Acid) is the foundation of the genetic support for life. It was discovered and isolated by Friedrich Miescher in 1869, but the paper of Watson and Crick (1953) is considered as the work that pioneered the study of DNA. It is a macromolecule, a large group of atoms tied together by chemical bonds, composed of two *strands* forming a double helix shape. A DNA strand is a chain of *nucleotides*, each being composed of a nucleobase that is either Adenine (symbol “a”), Cytosine (symbol “c”), Guanine (symbol “g”) or Thymine (symbol “t”), plus a deoxyribose group and a phosphate group. Nucleotides are chained in a strand via their deoxyribose group binding to the phosphate group of the next nucleotide in the chain. The *5’ end* extremity of a strand is the nucleotide with a non-bound phosphate group and the *3’ end* extremity of a strand is the

nucleotide with a non-bound deoxyribose group. The strand extremities define its orientation: from the 5' end to the 3' end. Two strands of a DNA molecule are maintained together through hydrogen bonds connecting the nucleotides of each respective strand: Adenine is always paired with Thymine through two hydrogen bonds and Cytosine is always paired with Guanine through three hydrogen bonds. Two paired nucleotides from different strands of the same DNA molecule form a *base pair* (abbreviated *bp*) and each nucleotide of a pair is the *complement* of the other. The length of a DNA molecule is then the number of base pairs it contains. In a DNA molecule, each strand is the *reverse-complement* of the other strand: The sequence of nucleotides from a strand can be obtained by "reading" the other strand in the reverse direction (from the 3' end to the 5' end) and complementing each nucleotide read. A DNA molecule is schematically illustrated in Figure 1.1.

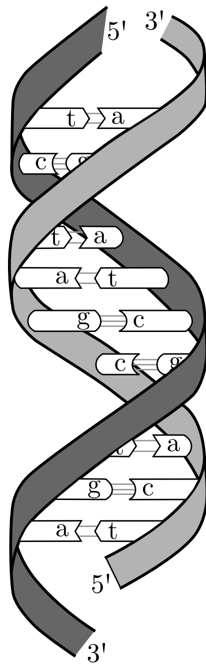


Figure 1.1: Representation of a DNA molecule. Each strand is shown with a different grey color and their respective extremities are annotated with "3'" and "5'" to indicate their orientation. Nucleotides located on the strands are illustrated with a white color and are annotated with the symbol of the nucleobase they represent. Grey colored links between nucleotides represent the hydrogen bonds.

A *gene* is a sequence of nucleotides which encodes a function such as metabolism, the process by which an organism maintains life. A *locus* is the position of a gene on a *chromosome*, a structure in which DNA is packed. A *genome* encodes all genetic information of an organism through its DNA mainly contained in chromo-

somes, but also through extrachromosomal DNA contained in other structures (mitochondria, chloroplasts and plasmids). Each cell of an organism contains a copy of the genome which length varies greatly among different organisms. A genome is said to be *diploid* if it contains two sets of chromosomes, one copy from each parent, such as in the human genome. To the contrary, a genome having only one set of chromosomes from one parent corresponds to a *haploid* genome. *Heredity* refers to the process of inheriting genetic traits from the parents. During this process, the DNA of the inherited genome is permanently altered and such alterations are called *mutations*. An *allele* is one of the possible forms of a gene that has been mutated. *Polymorphism* refers to a recurrent variation of a genome within a population of genomes and a *Single Nucleotide Polymorphism* (SNP) is a polymorphism of a single nucleotide at a given position of a genome within a population of genomes. A *species* is the most basic category of biological classification and refers to a group of similar organisms able to breed with each other. A *strain* is a variant within a species.

1.1.2 DNA Sequencing

DNA sequencing is the process of determining the order of nucleotides in a DNA strand. It is performed by a *sequencer*, an instrument able to "read" the DNA of a genome. Yet, sequencers cannot read the full length of the DNA in a genome but, instead, only fragments from several copies of the sequenced genome corresponding to overlapping nucleotide sequences called *reads*. The read length depends on the sequencing technology and ranges currently between a few dozen to a few thousand of base pairs. A *paired-end* read refers to a pair of nucleotide sequences read by a sequencer from different ends of the same DNA fragment. The *coverage* of a genome with respect to a set of reads indicates the expected number of reads covering a specific base of the sequenced genome. The coverage can be uneven depending on the sequenced genome and the sequencing technology.

Sequencing machines produce *sequencing errors* for which the rate, the distribution and the type (insertion, deletion and substitution of a nucleotide) vary upon the sequencing technology. Also, other attributes are used to characterize DNA sequencers such as the sequencing capacity (number of bases produced per

run), the throughput (number of bases produced in a given amount of time) and the cost of the machine. A comparison of sequencers based on their sequencing capacity and the length of the reads they produce is given in Figure 1.2 (Nederbragt, 2016).

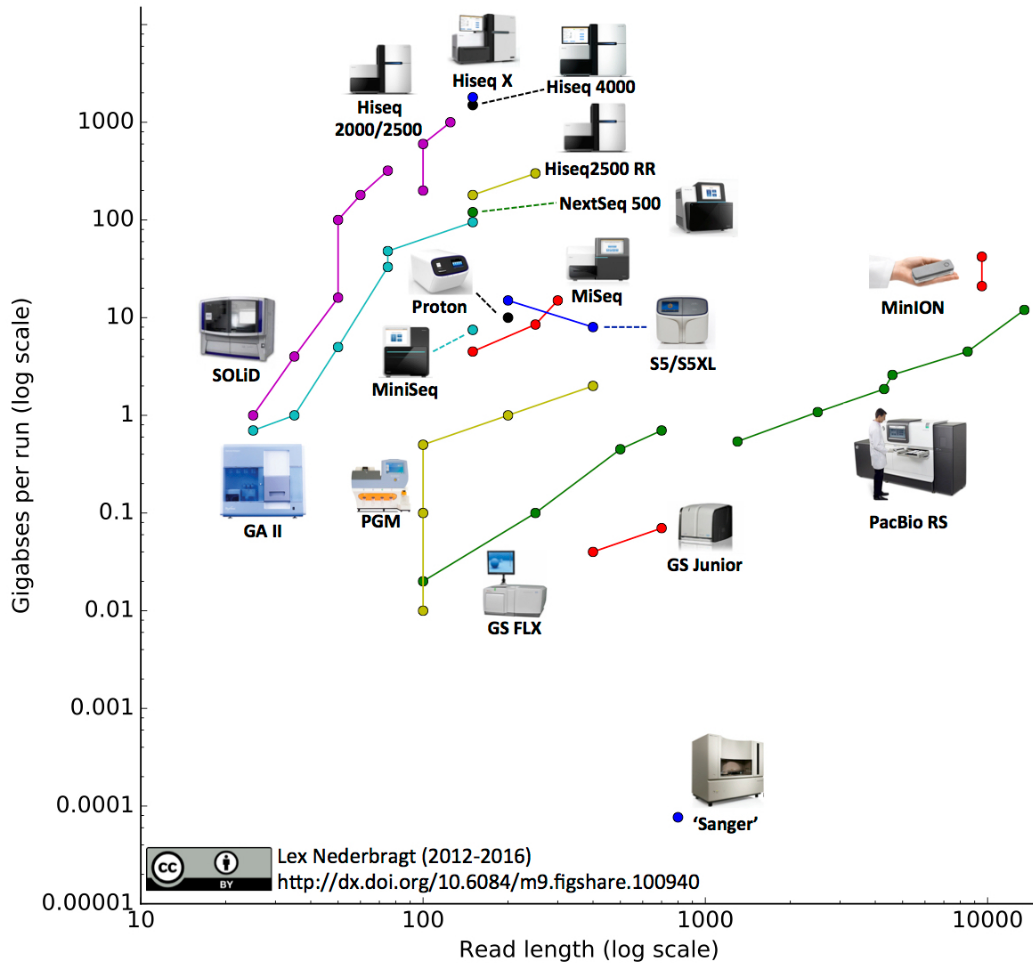


Figure 1.2: Comparison of sequencers based on their sequencing capacity and the length of the reads they produce (Nederbragt, 2016).

The first widely used sequencing technology, also called first generation sequencing, was developed by Sanger et al. (1977). It is characterized by reads up to 750 bp long (Schuster, 2008), a low error rate of 0.001 % to 1 % (Hoff, 2009) and a low sequencing capacity (Nederbragt, 2016). The length of the reads and their accuracy made this technology very successful for several decades but its cost was prohibitive and prevented its use to sequence long genomes. The second generation of sequencing technology, also known as Next Generation Sequencing (NGS) or High Throughput Sequencing (HTS) was introduced in 2005 as

highly parallel sequencing methods with a high throughput, multiple orders of magnitude more than with Sanger sequencing. Reads produced by HTS technologies are characterized by a shorter length than reads produced by Sanger sequencing and a low error rate of 0.01 % to 1 % (Glenn, 2011). The availability of such technologies marked the beginning of a new era in genomics because of its afforable price (Loh et al., 2012) making sequencing more accessible to the scientific community. Figure 1.3 shows that the cost of sequencing the human genome in 2001 (International Human Genome Sequencing Consortium, 2001; Venter et al., 2001) was estimated to 100 million dollars while ten years later, this price was 10,000-fold smaller. Finally, the third generation of sequencing technologies encompasses new methods such as Single Molecule Real Time sequencing and nanopore sequencing. Reads produced by these technologies are long, a few thousand of base pairs, but noisy as the error rate ranges from 13 % to 38 % (Rhoads and Fai Au, 2015). More detailed reviews of sequencing technologies are given in (Shendure and Ji, 2008; Thudi et al., 2012; Rhoads and Fai Au, 2015).

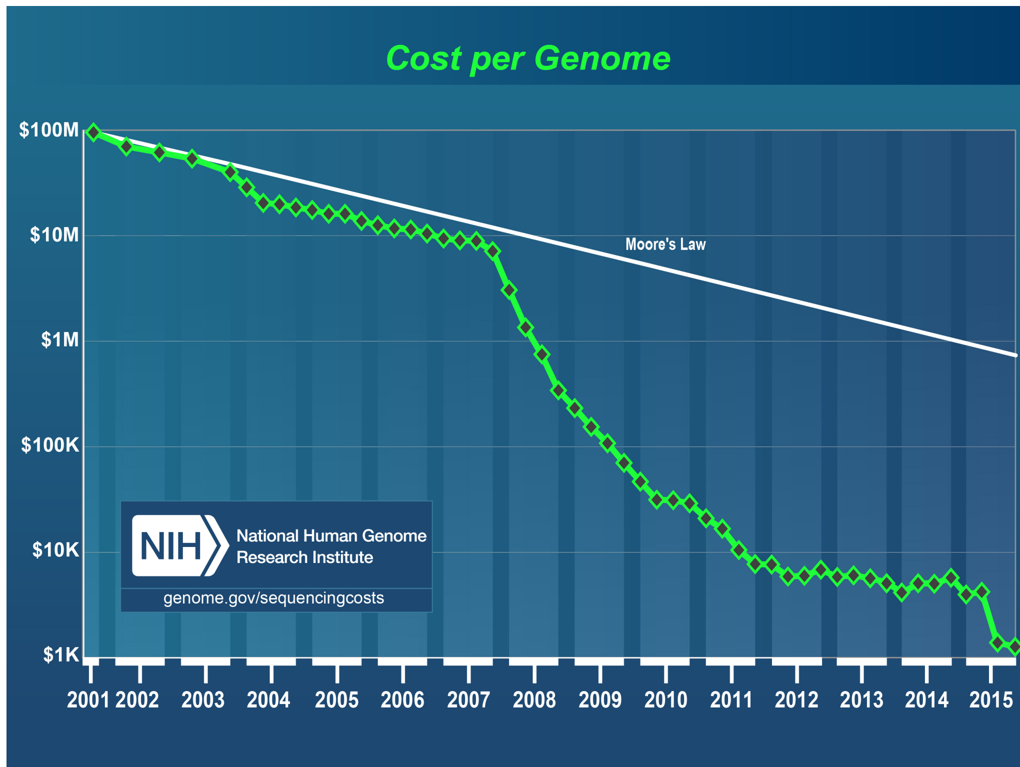


Figure 1.3: Annual sequencing cost of the human genome. The plot was produced by the National Human Genome Research Institute and is available at **www.genome.gov/sequencingcostsdata/**.

1.1.3 *de novo* Read Assembly

The *de novo read assembly* problem (Compeau et al., 2011) is the problem of reconstructing a genome as a sequence of nucleotides per chromosome from a set of reads only. To this end, *genome assembly* software use the overlaps of the reads to assemble them into larger sequences forming an *assembly*. This problem has been largely studied in computational biology because of its complexity and necessity. Indeed, genome analysis relies mainly on assembled genomes. Assembling reads that are erroneous, potentially short and with an uneven coverage that may be low or equal to zero in some regions of the sequenced genome is very challenging. Also, the sequenced genome can be difficult to assemble as it might contain repeated regions. Furthermore, finding overlaps requires to store and index the reads to assemble, a very time and memory consuming tasks as a single run of sequencing can generate up to billions of reads. Hence, the completeness and quality of an assembly depends upon all these factors as well as on the assembly algorithms that can create mis-assemblies. A more detailed description of the theory and practice of read assembly is given in (Simpson and Pop, 2015).

1.1.4 Comparative Genomics

The field of *comparative genomics* consists in the comparison of genomes to understand what are their similarities and differences. The answer to this question is of main interest to study the evolution of a species and, therefore, shed light on vital life mechanisms. Comparing a set of genomes to a *reference* genome has been a standard in computational biology for many years. Indeed, it is more time and memory efficient to compare all genomes of a set to a reference genome than performing all possible pairwise comparisons of genomes from the set. Yet, reference-centric comparisons are biased towards the reference: Dolle et al. (2017) established that in the latest human reference genome GRCh38, 3.2 Mbp were removed and 13.7 Mbp were added from the previous reference GRCh37. Thus, results obtained from a comparison with the human reference GRCh37 might be different with the updated reference GRCh38. With the computing power growth, the increased capability of external storage, and the improvements made in the domain of HTS technologies, multiple reference genomes are nowadays available for the same species. However, reference-centric

analysis methods cannot account for the very large and diverse variability among individuals of a same species. As an example, the 1000 Genomes Project (1000 Genomes Project Consortium, 2015) has sequenced more than 2,504 individuals from 26 populations representing a large set of 88 million variants. An ideal analysis of a newly sequenced human genome would benefit from a comparison with all 2,504 human genomes of such a project. In contrast to the human species which has a low degree of polymorphism, *Ciona savignyi* is a sea squirt species which has a very high degree of polymorphism, known to be one of the highest of any species (Leffler et al., 2012). Hence, a single genome of *C. savignyi* cannot be the representative of its species. Also, Mosquera-Rendón et al. (2016) mention that most of the issues in the development of an effective vaccine against *Pseudomonas aeruginosa*, an important pathogen in multiple types of infections, is because each strain of the species exhibits different mechanisms responsible for its pathogenesis. A study from Tettelin et al. (2005) indicates the same obstacles with the pathogen *Streptococcus agalactiae*. Therefore, developing an efficient vaccine against *P. aeruginosa* and *S. agalactiae* is unfeasible using reference-centric analysis methods. Consequently, linear references do not fulfill the requirements necessary for these problematics and must be augmented with information of multiple genomes.

1.1.5 Pan-genomes

The term *pan-genome* stems from the greek prefix *pan* which is a combining form of "all" or "every". It was first mentioned by Sigaux (1999) to study genome and transcriptome alterations in the context of cancer evolution. However, the analysis of multiple *S. agalactiae* genomes by Medini et al. (2005) laid the foundation for the concept of pan-genomes as we know it nowadays. In the study of Medini et al. (2005), a pan-genome is defined as the global gene repertoire of a species and is composed of two parts.

The *core genome* is defined as genes present in all strains of the species. Yet, some studies such as (Mosquera-Rendón et al., 2016) relax this definition to genes present in almost all strains of the species. These so-called housekeeping genes encode vital functions for survival and are the genomic basis for the phylogeny of the species. Evidences from the *P. aeruginosa* pan-genome study (Mosquera-

Rendón et al., 2016) show that its core genome encodes its pathogenicity through genes involved in lung infections and antibiotic resistance genes present in 95% to 100% of the strains studied. A similar conclusion from Donati et al. (2010) was established from the pan-genome of *Streptococcus pneumoniae*: 31 proteins out of 47 analyzed proteins that are surface-exposed, known to be involved with host interaction or with virulence are conserved in the core genome of *S. pneumoniae*.

The *accessory genome*, also called *dispensable genome*, is the pool of genes that are not present in the core genome. These genes account for the diversity of the species by encoding functions non-essential for their survival. They are often the result of environmental adaptation or are acquired from different organisms of the same environment through gene transfer. Mosquera-Rendón et al. (2016) highlight that each strain of the *P. aeruginosa* pan-genome exhibits a large variety of resistance mechanisms to the immune system in its accessory genome.

Genes specific to a strain of the pan-genome are sometimes distinguished as *singleton genes* (Blom et al., 2009), hence forming the *singleton genome*. A graphical representation of a pan-genome is provided in Figure 1.4.

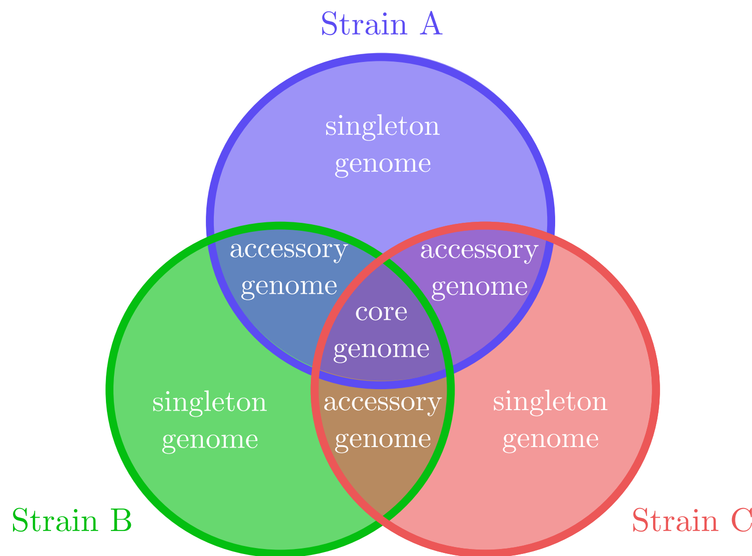


Figure 1.4: Representation of a pan-genome with 3 strains.

The proportion of core genome, accessory genome and singleton genome is variable from one species to another. An analysis of the *Escherichia coli* pan-genome of Lukjancenko et al. (2010) shows that the *E. coli* accessory genome is predominant and represents about 90% of its pan-genome. Similarly, the accessory genome of *P. aeruginosa* (Mosquera-Rendón et al., 2016) composes 85% of

its pan-genome. In contrast, the accessory genome of the *S. pneumoniae* pan-genome (Donati et al., 2010) represents an average 16% of its pan-genome, a number supported by Medini et al. (2005) establishing the accessory genome of the group B of *Streptococcus* to 18% of its pan-genome and the singleton genome to 1.5% of its pan-genome.

Tettelin et al. (2005) were the first to model the growth of a pan-genome as a function of the number of unique genes acquired per additional genome analyzed. The goal is to predict how many genomes are necessary to fully characterize the core and accessory genomes of a species. For this purpose, regression analysis is used to estimate statistically the pan-genome growth parameters from known data. In (Tettelin et al., 2008), two non-linear regression analyses were performed using the data from different species: a power law regression whose function makes the output vary as a power of the input and an exponential regression whose function makes the ouput vary as an exponent of the input. Results show that the power law regression was more adapted than the exponential one. This conclusion is also supported by the data of the *P. aeruginosa* pan-genome study from Mosquera-Rendón et al. (2016). Another model is the finite supragenome model (Hogg et al., 2007) which attempted to improve the method of Tettelin et al. (2005) by considering an unequal sampling probability of genes in the population. Donati et al. (2010) compared the power law regression model to the finite supragenome model and showed that for a large number of genomes analyzed (>40), the power law regression was more appropriate to model the pan-genome growth.

More specifically, the Heap's law belonging to the class of power laws is well suited to model the pan-genome growth. It is generally used to describe the growth of unique attributes in a set of entities. For example, Heap's law is used in linguistics to model the size of a vocabulary V_H (number of unique words present in a set of documents) as a function of the number of documents n considered. It is defined as follows:

$$V_H(n) = Kn^{\gamma}$$

in which parameters K and γ are determined empirically.

Power law regression helps to determine the completeness of a pan-genome by estimating the number of genomes necessary to fully describe the species'

pan-genome. A pan-genome is said to be *closed* if it can be determined with an estimated finite number of genomes (Tettelin et al., 2005). The opposite is an *open* pan-genome in which each new strain analyzed is estimated to contribute with new genes to the pan-genome. An open pan-genome is usually the sign of an extremely adaptive species that is inclined to new evolutionary opportunities. More theoretically, a parameter $\gamma \leq 1$ in the power law regression indicates that the number of new genes converges to a constant as expected in a closed pan-genome. To the contrary, $\gamma > 1$ indicates an infinite number of genes in the pan-genome, potentially with a very slow gene discovery rate if $\gamma = 1$ (Tettelin et al., 2008).

An outline of pan-genomes in different application domains as well as a description of pan-genomic approaches from a computational perspective are provided in (Computational Pan-Genomics Consortium, 2016).

1.2 Computational Background

The purpose of this section is to outline all computational notions that will be used throughout this thesis. Notations and preliminary notions are first introduced in Section 1.2.1, followed by a description of graphs and their variants in Section 1.2.2. Then, Section 1.2.3 details a specific type of graph, a tree, and provides an overview of trees specialized in string indexing with their variants. Hash tables are presented afterwards in Section 1.2.4 as a data structure type for indexing elements. The description of Bloom filters and their variants in Section 1.2.5 is derived from the description of hash tables. Finally, Section 1.2.6 presents the Burrows-Wheeler Transform and Section 1.2.7 presents the FM-Index as a string indexing method and data structure, respectively. Note that in the figures of this section, grey colored labels and arrows are for illustration but are not stored in practice.

1.2.1 Preliminaries

A *string* S is a sequence of symbols drawn from a finite non-empty set of symbols called the *alphabet* $\mathcal{A}$. Its *length* is denoted by $|S|$. A string of length 0 is an *empty string*. A *substring* of S is a string occurring in S: it has a starting position i, a length l and is denoted by $S(i, l)$. A *prefix* of S is a substring starting at position 1 while a *suffix* of S is a substring finishing at position $|S|$. The concatenation operator $\odot$ joins together two strings such that $S = p \odot s$ for (potentially empty) prefix p and suffix s.

A *repeat* is a substring that occurs at least twice in a string, i.e., $S(i, l) = S(j, l)$ for two positions $i \neq j$ in a string S and $l > 0$. A repeat is *left maximal* if it cannot be extended to the left without introducing a mismatch, i.e., $i = 1$ or $j = 1$ or $S(i-1, 1) \neq S(j-1, 1)$ for all pair of positions $i \neq j$ with $S(i, l) = S(j, l)$. Reciprocally, it is *right maximal* if it cannot be extended to the right without introducing a mismatch, i.e., $i = |S|-l+1$ or $j = |S|-l+1$ or $S(i+l, 1) \neq S(j+l, 1)$ for all pair of positions $i \neq j$ with $S(i, l) = S(j, l)$. A *maximal* repeat is left and right maximal. A *Maximal Exact Match* (MEM) is a tuple (i, S_1, j, S_2, l) representing a maximal substring match of two strings S_1 and S_2, i.e., $S_1(i, l) = S_2(j, l)$ with $S_1(i, l)$ and $S_2(j, l)$ being maximal over S_1 and S_2.

An *array* is an organization of elements in a contiguous space. Each element of an array A has a position i and is denoted by $A[i]$. Arrays are *directly addressable*: each element is accessible in $\mathcal{O}(1)$ time through its position in the array. A range of elements between positions $i < j$ (inclusives) is denoted by $A[i..j]$. A dynamic array can be reallocated to change its size.
A *list* is an abstract data structure representing a sequence of ordered values. The most common implementation of a list is the *linked list* which is represented by a sequence of elements, each containing a value and a pointer to the next element of the list.

1.2.2 Graphs

Definition 1. A graph $G = (V, E)$ is an abstract data structure composed of a set of vertices V and a set of edges E such that each edge $e \in E$ connects two vertices of V.

A *graph* is *directed* if each edge has a start vertex and an end vertex such that the edge points away from the start vertex. The edge *in-degree*, reciprocally *out-degree*, of a vertex v is the number of edges ending on, reciprocally starting from, vertex v. The edge degree of a vertex is the sum of the edge in-degree and out-degree of this vertex. A vertex is *branching* if it has an edge in-degree > 1 or an edge out-degree > 1. A *path* is a sequence of vertices such that each pair of consecutive vertices in the sequence has an edge connecting them in the graph. A path is branching if it contains at least one branching vertex. A *bubble* is formed by two paths sharing only the same first vertex and the same last vertex. A graph is *cyclic* if it contains at least one cycle, i.e., a path starting and ending on the same vertex. A graph is *connected* if there exists a path between all pairs of vertices.

1.2.2.1 de Bruijn Graphs

A *de Bruijn graph* (dBG) is a directed graph $G = (V, E)$ in which each vertex $v \in V$ represents a k-mer, a string of length k over $\mathcal{A}$. A directed edge $e \in E$ from vertex v to vertex v' representing k-mers x and x', respectively, exists if and only if $x(2, k-1) = x'(1, k-1)$. Each k-mer x has $|\mathcal{A}|$ possible *successors* $x(2, k-1) \odot a$ and $|\mathcal{A}|$ possible *predecessors* $a \odot x(1, k-1)$ with $a \in \mathcal{A}$. Note that in the original definition of the dBG, all possible k-mers for an alphabet $\mathcal{A}$ are represented in the graph. However, the definition of dBG used in the computational biology literature is less strict and only a subset of all possible k-mers are represented in the graph. These k-mers are extracted from the sequences the dBG is built from. A *compacted de Bruijn graph* merges all maximal non-branching paths of η vertices from the dBG into single vertices representing words of length $k + \eta - 1$. A *colored de Bruijn graph* (cdBG) is a graph $G = (V, E, C)$ in which (V, E) is a dBG and C is a set of colors such that each vertex $v \in V$ maps to a subset of C.
A lightweight representation of dBGs and cdBGs does not store edges since they are implicitly given by vertices overlapping on $k - 1$ symbols. However, implicit edges can falsely connect vertices that share an overlap of $k-1$ but do not overlap in the sequences the graph was built from.

The dBG is a long-studied abstract data structure used in computational

biology. It is particularly useful for the problem of *de novo* read assembly in which it is necessary to find a Hamiltonian cycle in the graph, a path starting and ending on the same vertex that visits each vertex exactly once. Although heuristics exist to extract Hamiltonian cycles from a graph, the read assembly problem is yet to be solved because a Hamiltonian cycle is only one possible reconstruction of the original genome the graph was built from. Furthermore, the Hamiltonian cycle formulation of the assembly problem has no immediate practical application due to coverage gaps and genomic variants.

1.2.3 Trees

Definition 2. A tree $T = (V_T, E_T)$ is an acyclic connected graph.

A *tree* is a special type of graph. In a tree T, vertices are separated in two groups. On one hand, the *internal* vertices of T are characterized by an edge degree > 1. On the other hand, the *leaves* are vertices with an edge degree of exactly 1. If the tree is rooted, a vertex called the *root* is designated. A rooted tree may be directed in which case the edges point from the root towards the leaves. Thus, we can define the notion of *children* and *parents* for each vertex of a directed rooted tree T. The children of a vertex $v \in V_T$, denoted by $children(v, T)$, is a set of vertices that can be reached by an edge starting at v. Reciprocally, the parent of a vertex $v \in V_T$, denoted by $parent(v, T)$, is the vertex having an edge that ends at v. The root of a rooted tree does not have a parent and the leaves of a rooted tree do not have children. The *depth* of a vertex $v \in V_T$ is denoted by $depth(v, T)$ and is the number of edges between the root of T and v. The *height* of T, denoted by $height(T)$, is the number of edges on the longest path from the root of T to one of its leaves. The level $i \geq 1$ of T is defined as the set of vertices with a depth of $i - 1$. A k-ary rooted tree T is a tree for which all vertices $v \in V_T$ have $|children(v, T)| \leq k$ (a 2-ary rooted tree is also called a binary tree). A subtree T' of a tree T is a tree for which the vertices and edges are connected subsets of the ones in T.
A search tree is a tree used to index elements, known as *keys*, potentially associated with *values*. There exists a large variety of search trees such as the binary search tree, the B-tree and the Red-or-Black tree. A more detailed description of those trees is given by Cormen et al. (2009). In the following, we will focus on a

subset of search trees named *tries*.

The premises of the trie were first introduced by De La Briandais (1959) but the concept of tries was mostly completed by Fredking (1960). The name "trie" comes from the word *re**trie**val* in an attempt to name this subset of search trees specialized in indexing sets of variable length strings. Strings are especially challenging to index in a memory-efficient manner compared to other key types because of their different and potentially long lengths. Also, strings offer a potential for redundancy to exploit as they usually come from natural language texts in which repetitions are expected to occur. Finally, indexing and retrieving strings might require type-specific operations such as prefix search. Note that tries do not restrict the key type to strings. For example, tries are typically used in routers for fast IP address lookup in which an IP address is a series of four 8 or 32 bits integers.

Definition 3. A trie $T = (V_T, E_T)$ is a directed rooted tree in which each edge is labeled with a single symbol of an alphabet $\mathcal{A}$ such that for any vertex $v \in V_T$, all edges starting at v are labeled with a different symbol. A path from a vertex $v \in V_T$ to a vertex $v' \in V_T$ represents a substring of length $depth(v', T) - depth(v, T)$, which is the concatenation of all symbols on the edges of this path.

The string obtained from the concatenation of all symbols on the edges of a path from the root to a vertex v is the prefix of a string stored in T and is denoted by $prefix(v, T)$. A subtrie of T rooted at vertex v represents a set of suffixes denoted by $suffixes(v, T)$ such that for each suffix s in this set, $prefix(v, T) \odot s$ is a string that is stored in T. If the vertex v is a leaf, then $|s| = 0$. Thus, tries inherit the dynamics of trees such that it is not necessary to know before insertion how many strings will be stored in a trie. An example of a trie is given in Figure 1.5.

A trie can be implemented using various data structures such as a linked list or an array. The choice of the supporting data structure for a trie is of main importance, as well as the implementation design that must be chosen carefully. Indeed, a trie can consume significant memory if implemented in a naive way, such as representing edges with pointers to memory locations, a typical imple-

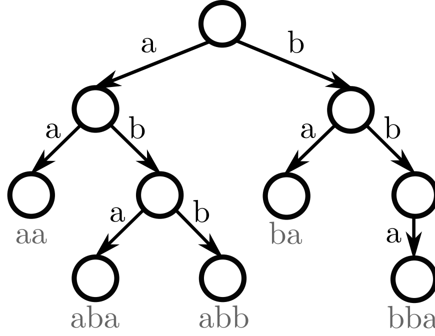


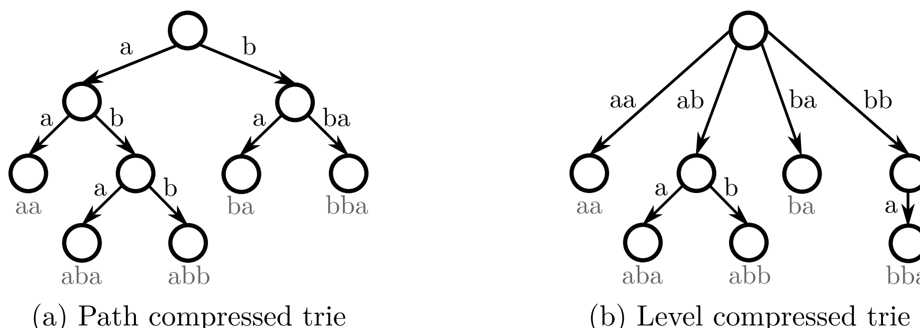
Figure 1.5: Trie of strings "aa", "aba", "abb", "ba" and "bba" for an alphabet $\mathcal{A} = \{a, b\}$.

mentation based on linked lists. This design scatters vertices over the memory of a computer (main memory or disk memory) and forces the Central Processing Unit (CPU) to visit dispersed locations of the memory in a short time while traversing a trie for example. At each traversed vertex, the CPU copies the data located at the memory address to a small multi-layer high-speed memory called *cache* which is located near the CPU. The first assumption made by a CPU is that memory accessed is likely to be modified: operations carried out in the cache are much faster than in Random Access Memory (RAM) and a lot faster than on disk. Secondly, a CPU assumes the data locality property: It is likely that a memory location close of the one accessed at the first place is going to be read and eventually modified. If the content of a memory location to be accessed is not already present in one layer of the cache, the processor must fetch it – a so-called *cache-miss*. Cache-misses are the main bottleneck in applications with no data locality. Hence, a naive pointer-based implementation of a trie is inefficient because it is cache-oblivious and each indexed symbol is the label of an edge using a 64 bits pointer on current computer architectures. A trie has a dynamic structure and inserting new elements triggers the insertion of new vertices and edges. If the supporting data structure is cache friendly and memory efficient but requires to move large chunks of memory for each new edge or vertex inserted, the data structure will be time inefficient. Finally, even though shared prefixes are stored in a single trie path, the same suffixes must be stored multiple times if their corresponding prefixes are different.

In order to improve the memory efficiency of tries and decrease the number of cache-misses while traversing them, the *radix trie* was proposed. A radix trie

employs a *path compaction* method which merges a path of $n > 1$ vertices having each at most one child into one vertex whose in-going edge is labeled with a substring of n symbols. A binary radix trie is also referred to as a *PATRICIA trie* from Morrison (1968).

An orthogonal compaction method was presented by Andersson and Nilsson (1993) as a *level compaction* method. Level-compaction applies to a k-ary subtrie t for which each internal vertex v from level $i = 1$ (the root) to level $j > i$ has $|children(v,t)| = k$ children. For such a subtrie t, each path from the root to a vertex v at level $j + 1$ is replaced by a directed edge from the root to v. This edge is labeled with the concatenation of the symbols on the path. Path and level compactions are illustrated in Figure 1.6.



(a) Path compressed trie (b) Level compressed trie

Figure 1.6: Compaction methods for tries of strings “aa”, “aba”, “abb”, “ba” and “bba” for an alphabet $\mathcal{A} = \{a, b\}$.

Path and level compactions were combined by Nilsson and Karlsson (1999) in a data structure named the *LC-trie*. A more detailed review of tries is given by Askitis and Sinha (2010).

1.2.3.1 Suffix Tries and Suffix Trees

A trie as described above indexes a set of strings such that a prefix search can be performed in a natural top-down traversal of a trie. In order to perform a substring search, a *suffix trie* is required. Introduced by Weiner (1973), a suffix trie of a string S indexes all $|S|$ suffixes of S. A prefix search can then be performed over the trie indexing the suffixes and therefore, simulate a substring search. As each suffix s of S has a length of at most $|S|$ symbols, the total number of substrings in S is $\mathcal{O}(|S|^2)$ and corresponds to the number of vertices in the

suffix trie of S.

The *suffix tree* is a path compacted suffix trie which has at most $|S| - 1$ internal vertices. Each internal vertex of the tree except the root has at least two children. To ensure this property, an extra symbol "\$" called *terminal* symbol is added at the end of S to guarantee that no suffix is the prefix of another suffix. Labels on the edges are not stored explicitly as they all relate to the same string: Each label of an edge can be instead replaced by its start position in S and a reference to the child list of the vertex pointed by the edge. Hence, instead of the quadratic space of the suffix trie, the suffix tree reduces the space usage to $\mathcal{O}(|S|)$ and can be built in linear time using the McCreight (1976), Ukkonen (1995) and Farach (1997) algorithms.

A vertex v of a suffix tree T represents the prefix of one or multiple suffixes. A *suffix link* starting at a non-root vertex v for which $prefix(v, T) = a \odot s$ connects to a vertex v' for which $prefix(v', T) = s$ with $a \in \mathcal{A}$ and s is a (potentially empty) string. Suffix links are not part of the suffix tree but are used by the McCreight (1976) and Ukkonen (1995) algorithms. The *Generalized Suffix Tree* (GST) is a suffix tree of at least two strings. It is built from a collection of $N > 1$ strings $S_1 \odot ... \odot S_N$: Each string S_i finishes with a different terminal symbol $\$_i$ such that $\$_1 < ... < \$_N$. Suffix trees are useful for numerous applications such as finding MEMs and MUMs.

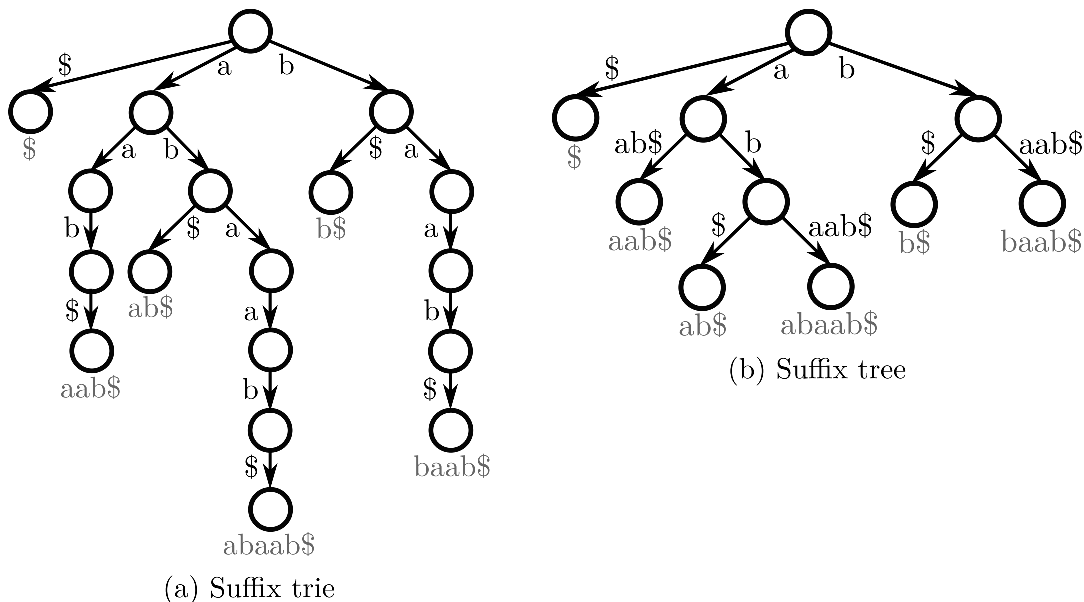


Figure 1.7: Suffix trie (a) and tree (b) of string "abaab\$".

As a standard suffix tree uses $\mathcal{O}(|S| \log |S|)$ bits for a string S over an alphabet $\mathcal{A}$, the high memory cost of such a data structure for long strings has motivated the need for more lightweight data structures in order to get closer to the $\mathcal{O}(|S| \log |\mathcal{A}|)$ bits required to store S itself. Hence, multiple proposals have been made for a *Compressed Suffix Tree* (CST) (Sadakane, 2007; Fischer et al., 2009; Ohlebusch et al., 2010; Russo et al., 2011). The most lightweight representation of a CST was proposed by Russo et al. (2011) and uses $|S|H_k + o(|S|)$ bits for a string S with H_k being the empirical entropy of S. This representation supports all operations of the suffix tree in $\mathcal{O}(\log |S| \log\log |S|)$ time, except returning the child of a vertex corresponding to a searched symbol which is in $\mathcal{O}(\log |S| \cdot (\log\log |S|)^2)$ time.

The *suffix array* by Manber and Myers (1993) is the counterpart of the suffix tree. It is an array of the sorted suffixes of a string S in which suffixes are represented by their start position in S. The lexicographic order of the suffixes mimics a depth-first traversal order of the suffix tree. A *suffix array interval* is an interval of positions in the suffix array corresponding to a set of contiguous suffixes starting with a given prefix p. Such an interval represents the subtree of a suffix tree T for which the root has $prefix(root, T) = p$. A suffix array, just as the suffix tree, can be built in $\mathcal{O}(|S|)$ time and uses $\mathcal{O}(|S|)$ space. As a suffix array does not cover all functionalities of a suffix tree, an *Enhanced Suffix Array* (ESA) (Abouelhoda et al., 2004) augments the suffix array with at least one additional structure, the *Longest Common Prefix* (LCP) array, in order to enable all operations supported by a suffix tree. In practice, the suffix array and the ESA are preferred to the suffix tree as they can be implemented more efficiently. The suffix array and the ESA also improve the data locality compared to the suffix tree because an array is stored contiguously in memory.

	SA	
1	6	$
2	3	aab$
3	4	ab$
4	1	abaab$
5	5	b$
6	2	baab$

Figure 1.8: Suffix array of string “abaab$” for alphabet order $ < a < b.

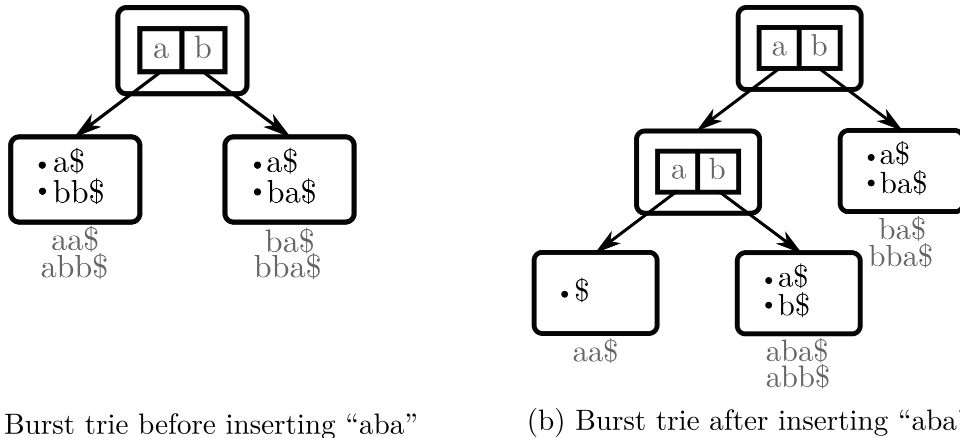
1.2.3.2 Burst Tries

Path and level compactions of tries are efficient methods in a limited number of cases, more specifically if a long unique substring or all possible equal-length substrings are rooted from a vertex. The burst trie is a compacted trie designed by Heinz et al. (2002). It is based on the simple idea that the main cause of memory and time inefficiency in tries is because of subtries $\textit{suffixes}(v, T)$ in which each path from v to a leaf is not branching. Therefore, a more efficient representation of such subtries of suffixes is needed. In the burst trie, edges starting from an internal vertex v are stored in an array of length $|\mathcal{A}|$ contained in v. Each element of such an array represents the label $a \in \mathcal{A}$ of the edge it stores. An edge exists at the position represented by label $a \in \mathcal{A}$ only if the edge points to a child for which all $s \in \textit{suffixes}(v, T)$ have $s(1, 1) = a$. Leaves of the burst trie are dynamic arrays representing sets of suffixes with a limited capacity. When the capacity of a leaf suffix set is exceeded during an insertion, a *burst* is triggered. Bursting a leaf replaces it with a new internal vertex storing edges labeled with the first symbol of every suffix the leaf contained. These edges point to new leaves containing the remaining symbols of the suffixes. An example of bursting is provided in Figures 1.9. Askitis and Sinha (2010) proposed multiple variants named the *HAT-trie*, the *hybrid HAT-trie* and the *HAT*$^{+}$*-tri*e which explore different splitting techniques as well as different data structures for the leaves. Those variants are expected to be more space-efficient than the HAT-trie while having the disadvantage to be slower to construct and search.

Edge label arrays in vertices of the burst trie are directly addressable. The drawback of this representation is that empty elements occupy memory. Also, even though suffixes are grouped together into leaf suffix sets, multiple locations of the memory representing internal vertices still need to be visited in order to reconstruct the prefixes.

1.2.4 Hash Tables

A *hash table* is an implementation of the so-called *associative array* or *dictionary*, also known as a “key-value” data structure. The purpose of such data structures is to associate information to elements, known as *values* and *keys* re-



(a) Burst trie before inserting "aba" (b) Burst trie after inserting "aba"

Figure 1.9: Burst trie of strings "aa", "abb", "ba" and "bba" for an alphabet $\mathcal{A} = \{a, b\}$. Leaves have a capacity of two suffixes. Inserting string "aba" in the burst trie of (a) triggers a bursting of the left leaf, resulting in the burst trie setup shown in (b). A terminal symbol is added at the end of each string in order to handle the case of one string which is the prefix of another string.

spectively, such that those values can be retrieved from the keys alone. One of the main differences with tree-like data structures accomplishing a similar purpose is that basic key-value operations (inserting, getting and removing a key-value) are expected to be carried out in constant time. Formally, n keys from a universe U have to be inserted and in most of the use cases, $n \ll |U|$. Therefore, creating a direct addressing array of size $|U|$ is not a suitable solution. Instead, a hash table H is an array of length m with $m \ll |U|$ and each key e to insert will be mapped to an element of H through a *hash function* h such that $h(e) : e \mapsto \{1, .., m\}$. For the sake of convenience, we assume in the following a set representation based on a simplified hash table model in which the keys to insert are unique and there are no values. A key e is stored and can be found in the element $H[h(e)]$, while removing e from H deletes the content of $H[h(e)]$. All these operations take $\mathcal{O}(1)$ time and H uses $\mathcal{O}(n + m)$ space.

1.2.4.1 Collisions

The previously described hash table design is minimal and creates *collisions*, i.e., two keys e_1 and e_2 collide if $e_1 \neq e_2$ and $h(e_1) = h(e_2)$. Figure 1.10 illustrates a hash table in which two keys are colliding.

A hash function that minimizes the number of collisions is chosen indepen-

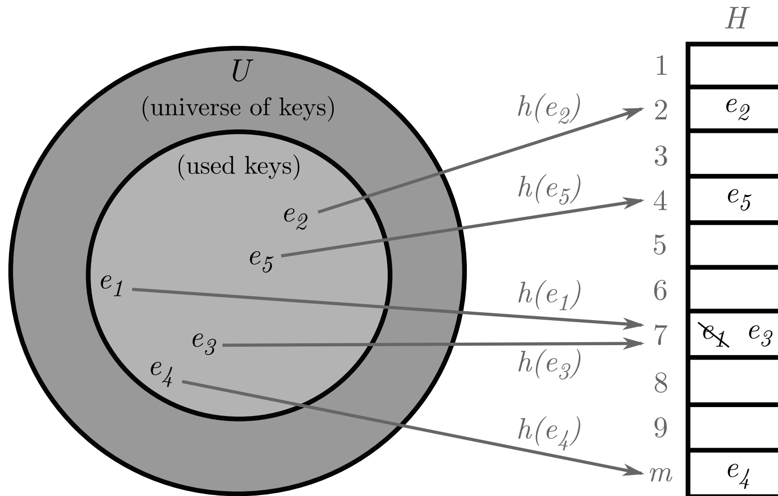


Figure 1.10: Hash table with collisions.

dently from the keys, i.e., through randomization. As the set of all such functions is very large, *universal hashing* is an approach that can design a smaller class of hash functions: A *τ-universal hash function* has a collision probability between two random distinct keys e_1 and e_2 of $prob(h(e_1) = h(e_2)) \leq \frac{\tau}{m}$ with m being the size of the hash table. A more detailed description of universal hashing is given by Cormen et al. (2009).

Chaining is a collision resolution method: each element of H does not contain a single key anymore but a chain of keys stored in a list. Inserting a key in H remains a $\mathcal{O}(1)$ time operation, but finding or deleting a key takes $\mathcal{O}(1 + n/m)$ expected time because the expected list length of each element is n/m. In the worst case, the hash function h maps all keys to the same element and finding or deleting a key in H takes $\Theta(n)$ time. An example of a hash table with chaining is given in Figure 1.11.

1.2.4.2 Open Addressing

Open addressing methods resolve collisions in hash tables by providing alternate locations, also known as *probing*, to keys that are colliding. Hash tables using open-addressing are self-contained in contrast to hash tables using chaining as they do not store the values outside the hash table. A first open addressing method is *linear probing* and inserts a key e in element $H[b]$ which is the first non-occupied element from $H[h(e)]$ with $b \geq h(e)$. If no element at position $b \leq m$ is

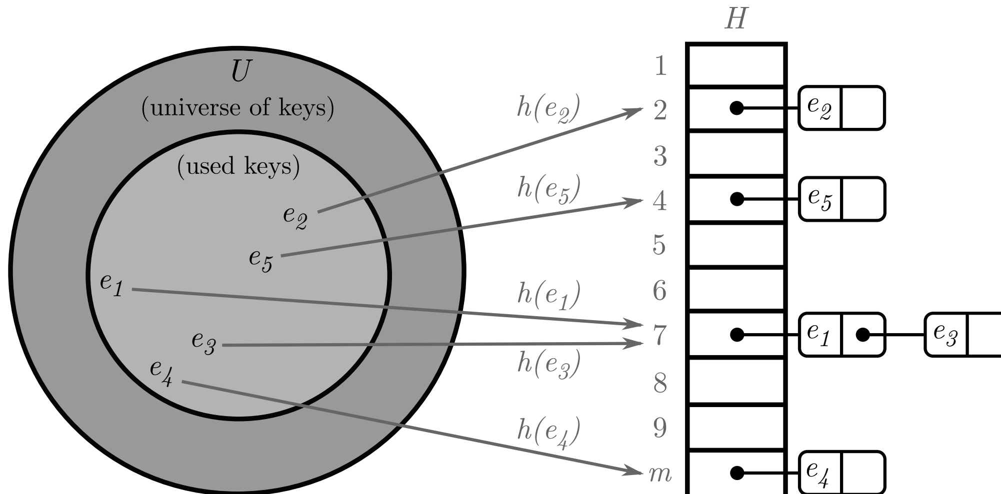


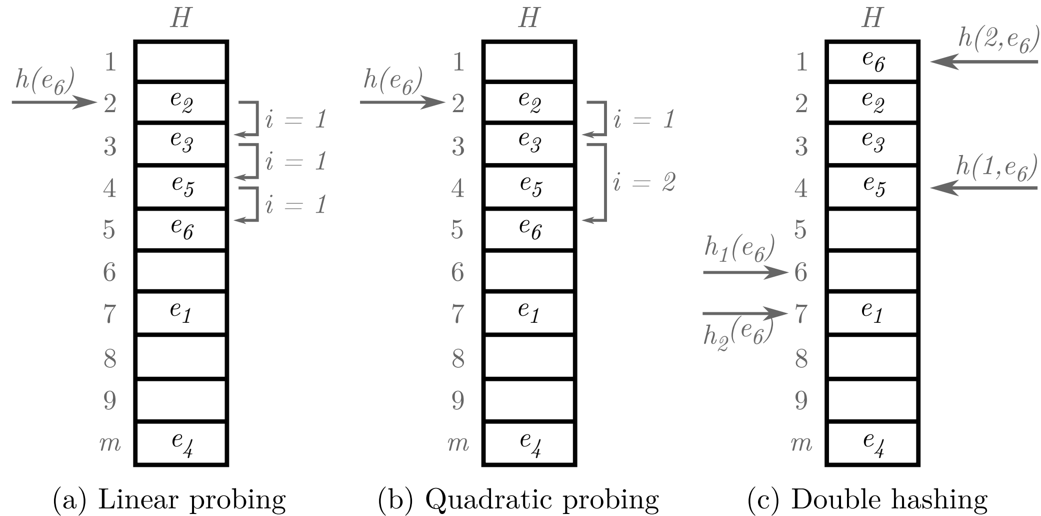
Figure 1.11: Hash table with chaining.

available, the hash table can be extended to contain $|H| > m$ elements. Finding e consists in iterating over the elements of H from position $h(e)$ with an increment of $i = 1$ until e is found or an empty element is encountered, in which case e is not present. Therefore, inserting and finding a key in H takes $\Theta(|H|)$ worst case time. Removing a key from a hash table using linear probing is more complex than from a chained hash table. Indeed, simply removing a key from its element $H[b]$ creates an empty element that might cause the search of a different key e' stored in an element at position $b' > b$ to stop prematurely. It is therefore necessary to store a supplementary information for each element indicating if it is empty because it does not contain a key or because the key it contained was removed, in which case a search going through this element would continue. *Quadratic probing* is similar to linear probing but uses a quadratic increment to iterate over the elements of the hash table in case of a collision. Finally, the *double hashing* method combines two hash functions in the probing:

$$h(i, e) = ((h_1(e) + i \cdot h_2(e)) \bmod m) + 1$$

If element $H[h(i, e)]$ is already occupied, i is incremented until an empty element is found. Compared to linear and quadratic probing, double hashing allows to avoid long ranges of occupied elements in the hash table. The different open addressing methods are illustrated in Figure 1.12

Although hash tables are advantageous because of the key-value association handled by constant time operations, hash functions with good hashing perfor-



(a) Linear probing (b) Quadratic probing (c) Double hashing

Figure 1.12: Open addressing methods.

mance can be long to compute in practice because of their complex randomization machinery, particularly in the context of cryptographic applications. Furthermore, not all elements of a hash table are used if the hash table size was pre-/re-allocated without a priori knowing the number of keys to insert. Finally, when a hash table contains a number of keys n converging to the size of the hash table, collisions occur more often during insertion and open addressing methods spend more time finding empty elements, making the hash table time inefficient.

1.2.5 Bloom Filters

Introduced by Bloom (1970), a *Bloom filter* (BF) is a space and time efficient data structure that records the approximate membership of elements in a set. Based on the hash table principle, look-up and insertion times are constant. The BF is composed of a binary array B of m elements, initialized with 0s, in which the presence of n items is recorded. A set of f hash functions $h_1, ..., h_f$ is used such that inserting an item into B and testing for its presence are then

$$\mathsf{Insert}_{\mathsf{bf}}(e, B) : B[h_i(e)] \leftarrow 1 \text{ for all } i = 1, ..., f$$

and

$$\mathsf{MayContain}(e, B) : \bigwedge_{i=1}^{f} B[h_i(e)],$$

respectively, in which $\bigwedge$ is the logical conjunction operator. These two operations are illustrated in Figures 1.13 and 1.14 respectively.

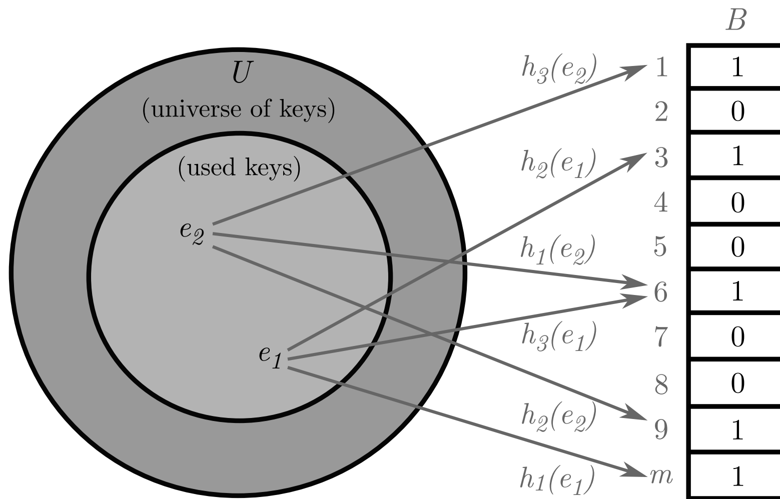


Figure 1.13: Insertion of two items e_1 and e_2 into a BF.

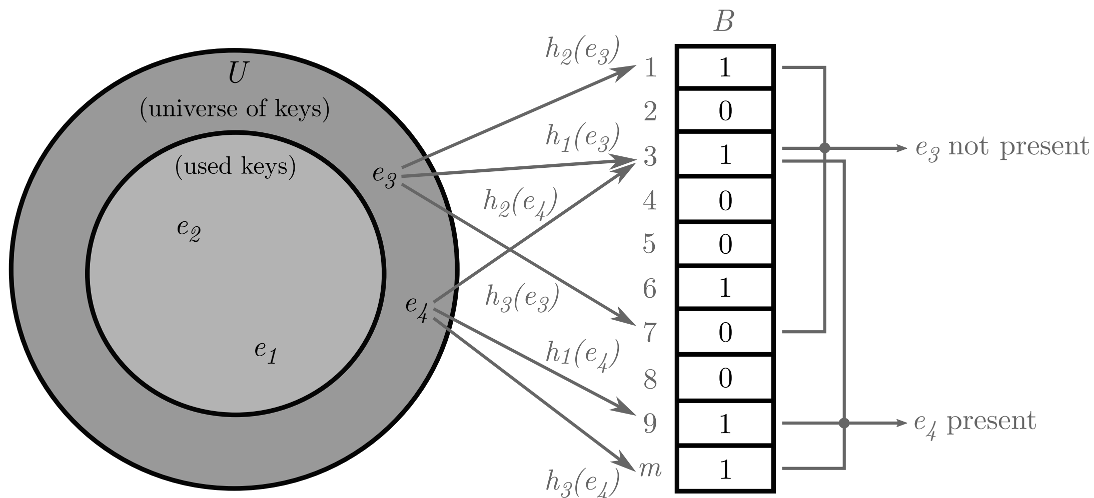


Figure 1.14: Query of two items. Item e_3 is a true negative and item e_4 is a false positive.

Kirsch and Mitzenmacher (2006) demonstrated that two hash functions combined in a double hashing technique can be applied to BFs in order to simulate more than two hash functions and obtain similar hashing performance. The BF does not support deletion as an element of B set to 1 might have been set by different hash functions for different items. The BF does not generate false negatives but may generate false positives, as MayContain can report the presence of items which are not present but a result of independent insertions. In contrast to a hash table, a BF never overflows because it is always possible to

insert new items. However, the number of false positives scales exponentially with the number of inserted items. Indeed, the false positive ratio φ (Kirsch and Mitzenmacher, 2006) is

$$\varphi \approx \left(1 - e^{\frac{-fn}{m}}\right)^f.$$

The number of hash functions that minimizes the false positive ratio is

$$f = \frac{m}{n}\ ln\ 2 \approx 0.7^{\frac{m}{n}},$$

and, given f and φ, the optimal size of a BF is

$$m = \frac{n\ ln\ \varphi}{(ln\ 2)^2}.$$

The union and intersection of two sets represented by BFs with the same parameters are

$$B_1 \cap B_2 = B_1 \wedge B_2,$$

and

$$B_1 \cup B_2 = B_1 \vee B_2,$$

respectively, in which $\vee$ is the logical disjunction operator. Such union and intersection are computed in $\mathcal{O}(|B|)$ time. Note that the false positive ratio of the intersection of two BFs B_1 and B_2 is upper-bounded by the smallest false positive ratio of B_1 and B_2. It is because the number of bits set to 1 in the BF intersection is at most the smallest number of bits set to 1 in B_1 and B_2. To the contrary, the false positive ratio of the union of two BFs B_1 and B_2 is lower-bounded by the greatest false positive ratio of B_1 and B_2. It is because the the number of bits set to 1 in the BF union is at least the greatest number of bits set to 1 in B_1 and B_2.

One main drawback of BFs is their poor data locality. Indeed, for one item, multiple and potentially distant locations of the BFs have to be visited and modified. In practice, these locations are first copied from main memory to cache and are then copied back from cache to main memory if modified. Such transfers affect the cache efficiency and slow down the running time.

BFs, as hash tables, have been extensively studied and applied in numerous domains, resulting in a large number of variants. Among the most popular

variants, Fan et al. (2000) proposed the *Counting Bloom Filter* (CBF) which supports deletions by maintaining a counter in each element of B instead of a binary value. Counters are initialized with 0s and the operation $\mathsf{Insert_{bf}}$ increments the counters instead of setting them to 1. Similarly, the deletion operation decrements the counters. The CBF is more memory consuming due to the counter size which, furthermore, must be extended when the maximum value they can hold is exceeded. Almeida et al. (2007) proposed the *Scalable Bloom filter* which is designed with a maximum false positive ratio but without a priori knowledge of the number of items to insert. It is made of a series of BFs for which the size is determined by a geometric progression. However, potentially all BFs of the series have to be queried to determine if an item is present. Putze et al. (2009) overcame the data locality problem of BFs with the *Blocked Bloom Filter* (BBF), an array of smaller BFs individually fitting into one or multiple cache lines. To insert or look-up an item, a supplementary hash function is used to determine which BF to load. As some BFs might be overloaded while some others might be sparse, it is difficult to establish the expected false positives ratio of a BBF. The authors of the data structure advise to use a BBF size 30% higher than a standard BF for the same parameters.

1.2.6 Burrows-Wheeler Tranform

The *Burrows-Wheeler Transform* (BWT) by Burrows and Wheeler (1994) is a method to index and compress a string. The key idea of the BWT is to rearrange an input string S such that symbols with similar context are aggregated together. The transformation of S, denoted $bwt(S)$, can be reversed to recover S without any additional information. Although $bwt(S)$ has the same length as S, its runs of same symbols makes it more compressible than S by compression methods such as move-to-front encoding followed by run-length encoding. Similarly to the GST (Section 1.2.3.1), it is possible to build the BWT of multiple strings by concatenating strings that end with different terminal symbols. The BWT is based on a lexicographic sorting of all cyclic permutations of an input string S as rows of a matrix. The last column of the matrix, identified by the label L, corresponds to the computed BWT. Note that the first column of the matrix, identified by label F, corresponds to the characters of S given in lexicographic order. An example is given in Table 1.1, in which the cyclic permutations of a

string “abaab$” are first computed (Table 1.1a) and are then sorted (Table 1.1b). The computed BWT corresponds to the concatenation of the last symbol of all sorted permutations. The example of Tables 1.1a and 1.1b shows that the computed BWT contains two runs of two symbols while the input string only contains one run of two symbols.

F					L
a	b	a	a	b	$
$	a	b	a	a	b
b	$	a	b	a	a
a	b	$	a	b	a
a	a	b	$	a	b
b	a	a	b	$	a

(a) Cyclic permutations

F					L
$	a	b	a	a	**b**
a	a	b	$	a	**b**
a	b	$	a	b	**a**
a	b	a	a	b	**$**
b	$	a	b	a	**a**
b	a	a	b	$	**a**

(b) Sorted cyclic permutations

Table 1.1: BWT computation of string “abaab$”. The computed BWT is highlighted.

The BWT construction algorithm previously described is inefficient and uses $\mathcal{O}(|S|^2)$ space to compute $bwt(S)$. Instead, $bwt(S)$ can be computed in $\mathcal{O}(|S|)$ space by keeping in each row of the matrix a pointer to the corresponding permutation in the original string. Burrows and Wheeler, in the original BWT paper, noticed that by using a terminal symbol in S, the sorting step after computing the cyclic permutations of S is equivalent to sorting the suffixes of S. As described in Section 1.2.3.1, this is achieved in $\mathcal{O}(|S|)$ space and time using a suffix tree or array.

Reconstructing a string S from $bwt(S)$ can be achieved naively with an iterative algorithm in $\mathcal{O}(|S|^2)$ time and space. This algorithm is described first to give an intuition on how the reconstruction operates. A more efficient algorithm running in linear time and space is explained afterwards. The naive algorithm starts from an empty matrix and for each iteration, fills the first empty column of the matrix with $bwt(s)$ and sorts lexicographically all the rows of the matrix. The reconstruction stops when all columns of the matrix are used and the reconstructed string is in the row which has the terminal symbol as last symbol. An example is provided in Table 1.2.

A more efficient reversing is based on the fact that each symbol in a row of column F is preceded in S by the symbol in the same row of column L.

L
b
b
a
$
a
a

(1) Prepend

L
$
a
a
a
b
b

(2) Sort

	L
b	$
b	a
a	a
$	a
a	b
a	b

(3) Prepend

	L
$	a
a	a
a	b
a	b
b	$
b	a

(4) Sort

		L
b	$	a
b	a	a
a	a	b
$	a	b
a	b	$
a	b	a

(5) Prepend

		L
$	a	b
a	a	b
a	b	$
a	b	a
b	$	a
b	a	a

(6) Sort

			L
b	$	a	b
b	a	a	b
a	a	b	$
$	a	b	a
a	b	$	a
a	b	a	a

(7) Prepend

			L
$	a	b	a
a	a	b	$
a	b	$	a
a	b	a	a
b	$	a	b
b	a	a	b

(8) Sort

				L
b	$	a	b	a
b	a	a	b	$
a	a	b	$	a
$	a	b	a	a
a	b	$	a	b
a	b	a	a	b

(9) Prepend

				L
$	a	b	a	a
a	a	b	$	a
a	b	$	a	b
a	b	a	a	b
b	$	a	b	a
b	a	a	b	$

(10) Sort

F					L
b	$	a	b	a	a
b	a	a	b	$	a
a	a	b	$	a	b
$	a	b	a	a	b
a	b	$	a	b	a
a	b	a	a	b	$

(11) Prepend

F					L
$	a	b	a	a	b
a	a	b	$	a	b
a	b	$	a	b	a
a	**b**	**a**	**a**	**b**	**$**
b	$	a	b	a	a
b	a	a	b	$	a

(12) Sort

Table 1.2: Reconstruction of a string from the BWT string “bba$aa”. The reconstructed string is highlighted.

This ensures a property named *LF-Mapping*: the occurrence of the i-th symbol a in F matches the occurrence of the i-th same symbol a in L. Therefore, $bwt(S)$ can be reversed unambiguously in a back-to-front manner using columns F and L only. As column F can be computed from column L, reversing $bwt(S)$ takes $\mathcal{O}(|S| + |\mathcal{A}|)$ space and time. The LF-Mapping property also enables pattern matching directly in the BWT of a string S. It is done by iteratively matching pattern P in $bwt(S)$ from the end to the beginning of P. At each iteration $i = 1, ..., |P|$, the symbol at position $P(|P|-i+1, 1)$ is searched within an interval corresponding to the suffix $P(|P|-i+1, i)$. The iterations stop when P has been entirely matched or when the symbol $P(|P|-i+1, 1)$ cannot be found within the

current interval, in which case it indicates P is not present. This method known as *backward search* is more detailed in (Li and Durbin, 2009). The LF-Mapping property on which relies the backward search is rarely used as it is, but instead with the support on the FM-Index described in the following section.

1.2.7 FM-Index

The BWT is not only useful for compression but also for indexing because of its connections with the suffix array. The *FM-Index* (Ferragina and Manzini, 2000) has been conceived as a direct application of the BWT by proposing a constant time LF-Mapping over the BWT of a string S. This allows to locate and count the occurrences of a pattern P in S with sublinear complexities. The FM-Index is made of a vector C and a matrix Occ. The vector C counts for each symbol $a \in \mathcal{A}$ of $bwt(S)$ the number of occurrences of lexicographically lower symbols in $bwt(S)$. Matrix Occ represents a function indicating, for a symbol $a \in \mathcal{A}$ and a position pos in $bwt(S)$, the occurrence number of a in $L(1, pos - 1)$. Examples are given in Tables 1.3a and 1.3b.

\$	a	b
0	1	4

(a) Vector C

	1	2	3	4	5	6
\$	0	0	0	1	1	1
a	0	0	1	1	2	3
b	1	2	2	2	2	2

(b) Matrix Occ

Table 1.3: FM-Index of the BWT string “bba\$aa”.

Given C and Occ, the LF-Mapping at position i in $bwt(S)$ is $\textsf{LF-Mapping}(i) = C[L[i]] + Occ(L[i], i)$. Using the constant time LF-Mapping from the FM-Index, counting the occurrences of a pattern P in S and locating them takes $\mathcal{O}(|P|)$ time. As the matrix Occ can be large, only a subset of the columns is stored in practice and the values of the non-stored columns are recomputed on the fly if needed. More details about the FM-Index and compression techniques to reduce its size are provided in (Ferragina et al., 2004).

1.3 Thesis Overview

Pan-genomes are a new opportunity in comparative genomics to better understand life compared to single-genome reference-based methods. Yet, they constitute a tremendous challenge regarding our capacity to simultaneously explore and analyze data from different sources. On the basis of Sections 1.1 and 1.2 detailing the biological and computational background of this thesis, Chapter II describes the current state-of-the-art for pan-genome indexing through the data structures used in pan-genome analysis methods. From this literature, the first aim of this thesis is to establish a list of missing or inadequate features in data structures used to index pan-genomes. Based on this list, we present in Chapter III a new data structure for pan-genome indexing: the Bloom Filter Trie (BFT). We show that the BFT outperforms a data structure providing the same features but based on an approximation of the data indexed.

The second aim of this thesis is to show that the problem of pan-genome read storage has never been addressed before and current techniques for compressed storage of reads are not adapted. Hence, Chapter IV defines first the characteristics that a pan-genome read compression method must have. Then, a new tool for pan-genome read compression named DARRC is detailed. We show that the BFT is adapted for the task of pan-genome read compression and is the supporting data structure of this tool. DARRC outperforms all tested tools based on individual and independent genome compression.

CHAPTER II

Methods and Tools for Pan-genome Indexing

2.1 Introduction

Along the years, the definition of pan-genome has acquired a broader meaning than its original definition by Medini et al. (2005). It nowadays also refers to a collection of sequences from different genomes (Computational Pan-Genomics Consortium, 2016). Such a collection can be analyzed naively by indexing separately each sequence it is composed of. This representation is however wasteful in memory since core genome and accessory genome need to be stored as many times as they occur in the strains. Furthermore, such a representation would require analyzing each genome separately, leading to time inefficient methods. If a pan-genome is composed of similar genomes, it is naturally more efficient to index variants of a collection of genomes with respect to a reference. Such methods will be reviewed in Section 2.2. Yet, computing similarities or variants is a complex task achieved by aligning sequences of the pan-genome. As a consequence, much attention has been paid to methods that either provide scalable sequence alignments for many similar genomes or extract directly from a sequence alignment the shared and unique regions of the pan-genome (Section 2.3). This exercise is not trivial and is the reason for the design of graph-based methods (Section 2.4) that take advantage of the similarity and difference not only with a reference but also with other indexed genomes. A special case of graph-based methods are the de Bruijn graph methods (Section 2.5) which are alignment-free and reference-free. Other methods reviewed in Section 2.6 enable large scale indexing and querying of genomic databases. Note that the following sections focus only on sequence-based data structures and methods for pan-genome indexing and querying at

the DNA level in contrast to gene-based methods. Other tools for computational pan-genomics are reviewed in (Computational Pan-Genomics Consortium, 2016). This chapter is based on (Zekic et al., 2018) and is a collaboration with Tina Zekic and Jens Stoye.

2.2 Reference-based Methods

Among the reference-based methods, Wandelt et al. (2013) proposed the **Referentially Compressed Search Index** (RCSI), a tool that uses referential indexing to index a pan-genome and to search for exact or inexact matches. The reference is first indexed using a CST (Section 1.2.3.1) and the other genomes are then encoded as ordered lists of subsequence matches to the reference, called *referential match entries*. The index is built with respect to a maximum query length and a parameter representing the maximum number of edit distance operations allowed during an inexact search. RCSI exploits the *seed-and-extend paradigm* (Rasmussen et al., 2006) based on the pigeonhole principle stating that for a sequence S, a pattern P and at most M mismatches, there exists at least one matching substring, called *seed*, of length $\left\lfloor \frac{|P|}{M+1} \right\rfloor$ between S and P. When queried, the reference is first searched for matching seeds that are then extended. Matches found are transferred onto the referential match entries using a hashing structure to identify genomes in which matches are found with at most M mismatches.

Another approach of referential indexing is to index variants instead of similarities. The **Multiple Genome Index** (MuGI) from Danek et al. (2014) applies this approach by maintaining a database of variants. Additionally, the occurrence positions of k-mers in the genomes are stored in arrays, sorted by lexicographic order of k-mers. The seed-and-extend paradigm employs the k-mer arrays for the exact or inexact matching algorithms such that genomes can be searched without scanning the reference, hence offering faster queries than other state-of-the-art data structures while using less memory during the search. MuGI requires less memory than RCSI to build the index but its search algorithms can handle only mismatches, while RCSI also supports insertions and deletions. Also, MuGI considers that variants are provided as input, which allows to keep the reference non-indexed while RCSI computes the subsequence matches itself by indexing

the reference.

The **Journaled String Tree** (JST) from Rahn et al. (2014) is another data structure for indexing similar genomes based on encoding genome variants with respect to a reference. The novelty of this work is that it can be "plugged-in" with existing sequential pattern matching algorithms instead of proposing a new search method. A JST is composed of a reference and an array of *branch-nodes* representing variants of one or multiple genomes forming branches with respect to the reference. Each genome is referentially compressed as a *journaled string* consisting of an insertion buffer and a binary search tree. The insertion buffer is a string which is the concatenation of genome subsequences not present in the reference (insertions). The binary search tree maintains positions and lengths of inserted or shared subsequences with respect to the reference. Pattern matching algorithms scanning a sequence in a left-to-right manner can be used in combination with the JST: as soon as a branch-node occurs in the reference, the state of the search algorithm is saved for later processing and the algorithm processes the occurring branch that is extracted from the corresponding journaled strings. Experiments with exact and inexact pattern matching algorithms show better running time and memory consumption compared to a naive scheme where sequences are processed in a sequential manner. However, MuGI shows better performance during the search, mainly due to its fully indexed search, while JST supports only sequential search.

Self-indexes, like the FM-Index (Section 1.2.7) and the CST, are data structures that compress and index data while providing random access to the indexed data as well as pattern matching functionalities. Mäkinen et al. (2010) introduced a new family of self-indexes (Navarro, 2012) for storage, retrieval and search of highly similar sequence collections. The central idea of this approach is that previous self-indexes could not capture the high similarity of multiple sequences that differ only by few variants. Hence, the newly proposed family extends existing self-indexes to achieve greater compression than the high-order entropy of the sequence collection. The approach is then used to introduce new basic structures, including an advanced suffix array sampling scheme, that are adapted afterwards to a CST. This representation of a collection of highly repetitive sequences uses $\mathcal{O}(n \log \frac{N}{n} + \omega \log^2 N)$ bits on average in which N is the total length of the sequence collection, n is the length of the reference and ω is the total number of

mutations. Exact pattern matching of a pattern P is achieved in $\mathcal{O}(|P| \log N)$ time. The provided basic structures require each sequence to be pairwise aligned with the reference of the collection to provide basic query operations.

BWBBLE (Huang et al., 2013) builds an *augmented reference* by including all variants detected between a set of genomes to a reference. For this purpose, a linear augmented reference is built and its BWT (Section 1.2.6) is computed such that it can then be used by an aligner based on the FM-Index. SNPs are handled by making use of the International Union of Pure and Applied Chemistry (IUPAC) nucleotide code in which each symbol represents either one nucleotide or multiple possible nucleotides. More complex variants such as indels (insertions or deletions) are linearized: One of the indel is incorporated in the augmented reference and its surrounding characters in the augmented reference are padded to the extremities of the other indels. These other indels are then concatenated to the augmented reference using a special separation symbol. Exact and inexact matching algorithms with the augmented reference use an extension of the BWT-based backward search (Sections 1.2.6 and 1.2.7) that handles IUPAC nucleotide symbols. As a IUPAC nucleotide symbol can represent multiple nucleotides, a query can match several different substrings in the augmented reference and, thus, different suffix array intervals. While being more accurate with the augmented reference than BWA with a single reference (Li and Durbin, 2009), the aligner is slower and uses significantly more memory.

Durbin (2014) describes another data structure based on the BWT, the **positional Burrows-Wheeler Transform** (pBWT), to represent a haplotype sequence collection with w binary allelic variable sites for efficient haplotype matching and compact storage. The proposed algorithms derive first a reverse prefix ordering of the sequences instead of the classical lexicographic ordering of suffixes in the suffix array. This new ordering ensures that locally maximal sequences are adjacent. Consequently, forward search of the sequences can be used to find all set-maximal matches from a new sequence or a sequence from the collection. Both matchings can be computed in linear time, $\mathcal{O}(w)$ time in the former case and $\mathcal{O}(wN)$ time in the latter case. The pBWT is highly compressible and uses the FM-Index for the forward search.

2.3 Alignment-based Methods

Panseq (Laing et al., 2010) is an online tool for pan-genome analysis relying on Pairwise Sequence Alignment (PSA), Multiple Sequence Alignment (MSA) and Local Sequence Alignment (LSA) for its different modules described in the following. The *Novel Region Finder* module extracts novel regions out of a set of input sequences by iteratively adding them to a database if there are no PSA matches with sequences already in the database. The *Core and Accessory Genome Finder* module eliminates first the singleton genome of the input sequences by creating a pool of input sequence segments that match at least two genomes. Segments are then broken into fragments of user-specified length such that an LSA determines if they are part of the core or accessory genome. Core genome sequences are built for each input sequence by concatenating its core genome fragments. The core genome is provided as an MSA of the core genome sequences while the accessory genome is provided as a binary matrix indicating the presence of each fragment in the input sequences. Finally, the *Loci Selector* module constructs a locus set based on the unique number of fingerprints and the discriminatory power of the loci among the input sequences.

The **Harvest suite** (Treangen et al., 2014) uses a variant of whole genome alignment for rapid core genome extraction of higly similar microbial genomes ($\geq$ 97 % average nucleotide similarity). The suite is composed of two tools: *Parsnp* for core genome MSA and *Gingr* for dynamic visualization of large scale MSA as well as core genome SNPs tree exploration. The first tool relies on the fact that core genome MSA has a lower complexity than MSA because it focuses only on regions shared by all genomes. Parsnp starts by indexing an input genome using a CST which is then queried with the remaining genomes in order to identify in a first step multiple Maximum Unique Matches (multi-MUMs) present in all input genomes and in a second step Locally Collinear Blocks of multi-MUMs. Gaps between collinear multi-MUMs are then aligned. Additional post-processing steps are performed to filter out unreliable SNPs and build the core genome SNPs tree. Parsnp was compared to whole-genome alignment methods, k-mer based methods and read mapping methods. Results show that its accuracy is a compromise between whole-genome alignment methods and read mapping methods but depends on the input data (draft or finished assemblies). However, Parsnp has the advantage to run in a small fraction of the other methods' running

times.

Nguyen et al. (2015) formulated the pan-genome construction problem as a genome rearrangement problem in which the arrangement and orientation of sequence alignment blocks for a set of genomes partitioned by homology must be optimized. Its complexity has been shown to be NP-hard and a heuristic using Cactus graphs (Paten et al., 2011) was provided.

2.4 Graph-based Methods

The Variant-based Graph (VG) representation of a collection of genomes was introduced in the tool **GenomeMapper** (Schneeberger et al., 2009) for sequence alignment against multiple genomes. This representation follows a similar principle as reference indexing methods with the exception that the underlying data structure does not index variants with respect to a reference but instead implements a fully indexed multi-genome reference in the form of a graph. GenomeMapper fragments each input genome into blocks of equal length maintained in a *block table*. Blocks corresponding to shared regions in the genomes are stored only once and each block is connected to its neighboring blocks, thus forming a graph in which bubbles are formed by blocks of divergent sequences. In addition, a k-mer hash-based index is built to map each k-mer occurring in the input genomes to a list of blocks and positions in the blocks. This index is used to identify *seeds* for the alignment.

The **Generalized Compressed Suffix Array** (GCSA) (Sirén et al., 2011, 2014) is a self-index data structure which indexes an MSA of genomes into a finite automaton. The latter is encoded using a generalization of the BWT to finite automata, achieving in the expected case for a constant-size alphabet $\mathcal{A}$ an $\mathcal{O}(n \cdot (1 + \frac{|\mathcal{A}|}{\omega})^{\mathcal{O}(\log n)})$ bits representation. Possible applications are read alignment, split-read alignment using splicing graphs, probe and primer design as well as alignment to assembly graphs. GCSA has been improved into GCSA2 (Sirén, 2017) which is the core data structure of a VG toolkit, vg (vg team, 2015), providing sequence alignment and read mapping over a multi-genome reference using the seed-and-extend paradigm.

HISAT2 (Kim et al., 2016) is a VG alignment tool for populations of genomes

based on HISAT (Kim et al., 2015). It uses a *Hierarchical Graph FM-Index* (HGFM) composed of a main Graph FM-Index (GFM) based on GCSA to represent the general population of genomes and a large number of small GFMs, each representing a small genomic region.

PanCake (Ernst and Rahmann, 2013) is an extension of string graphs, known from genome assembly (Myers, 2005), which achieves compression based on PSA. PanCake graph vertices represent a reference subsequence and a set of tuples called *feature instances*. Such a tuple contains a chromosome identifier, the position of a subsequence located in the identified chromosome and similar to the reference subsequence, as well as compressed information necessary to reconstruct the subsequence. Feature instances from the same chromosome are ordered in a doubly linked list, such that the entire chromosome can be reconstructed by iterating over the list and concatenating the reconstructed subsequences. New genomes to be inserted in the data structure are pairwise aligned with the chromosomes already inserted in the graph. One application of PanCake graphs is core and singleton genomes extraction: Singleton regions are the vertices that contain a unique feature instance, while core regions are the vertices containing at least one feature instance from every genome stored in the graph.

2.5 de Bruijn graph Methods

A usual step following the dBG (Section 1.2.2.1) construction is to *compact* it in order to decrease the memory requirements and simplify the graph for further analysis. As building a compacted dBG requires to build the uncompacted dBG first, much attention has been focused on algorithms enabling the direct construction of the compacted dBG without its uncompacted counterpart.

SplitMEM (Marcus et al., 2014) makes use of the dBG to extract shared regions of a set of similar genomes. The algorithm directly builds the compacted dBG in $\mathcal{O}(N \log g)$ time and $\mathcal{O}(N + |G|)$ space with g being the length of the longest genome and $|G|$ the size of the compacted dBG. To this end, SplitMEM exploits the relations between dBG and suffix tree (Section 1.2.3.1) with the use of *suffix skips*, a generalization of *suffix links* that allow to connect internal vertices of a suffix tree by trimming multiple characters at the beginning of a suffix. The

augmented suffix tree allows to identify first all MEMs of length at least k in the genomes. As MEMs can overlap and be nested, they are split using suffix skips in order to build the set of *repeatNodes*, substrings occurring at least twice in the pan-genome. Then, *uniqueNodes*, unique substrings of the pan-genome that link repeatNodes, as well as outgoing edges for each vertex, are identified. Core or accessory genome extraction can be performed with core genome defined as subsequences of the pan-genome occurring in at least 70 % of the indexed genomes.

Baier et al. (2016) improved SplitMEM with two algorithms, one using the CST and the other using the BWT. The CST algorithm follows the same two steps as SplitMEM: first, identifying repeatNodes based on left and right maximality of repeats, then identifying uniqueNodes that link repeatNodes, as well as edges of the graph. It runs in $\mathcal{O}(N)$ instead of $\mathcal{O}(N \log g)$ time, as suffix skips are not required to identify repeatNodes and a non-comparison sorting algorithm is used to identify uniqueNodes and edges of the dBG. Instead, the BWT algorithm computes the complete compacted dBG in a single backward pass over the pan-genome and runs in $\mathcal{O}(N \log |\mathcal{A}|)$ time for an alphabet $\mathcal{A}$.

TwoPaCo (Minkin et al., 2016) is a highly parallel method to build the compacted dBG of many similar whole genome sequences. It reduces the problem of finding maximal non-branching paths in the dBG to finding in the input sequences the positions of *junctions*, k-mers that are branching and k-mers that start or finish an input sequence. For this purpose, the algorithm considers first a set of candidate junction positions in the input sequences. For each such position i, the two $(k+1)$-mers starting at positions i and $i-1$ are inserted into a data structure D. Then, each such position i is processed again by querying D for all possible successors and predecessors of the k-mer starting at position i. If the k-mer has an edge in-degree of 1 and an edge out-degree of 1, it is not a junction. TwoPaCo uses this method in a first pass with a BF (Section 1.2.5) as data structure D to be memory efficient and select the candidate junction positions. Those positions are then used as input for the second pass with a hash table (Section 1.2.4) as data structure D to remove the false positive junctions generated by the BF. Because the two-pass method can still be memory intensive, k-mers are divided into partitions and the two-pass method deals with each partition once at a time. The expected running time is $\mathcal{O}(Nf + k \cdot (|G| + \sigma\phi\psi))$

with f being the number of hash functions used in the BF, σ being the number of non-junctions in G, ϕ being the probability of a non-junction to be a false positive and ψ being the average number of times a false positive junction occurs in an input sequence. The expected memory is $\mathcal{O}(\mathsf{Max}(m, k \cdot (\delta + \sigma\phi)))$ with m being the number of bits in the BF and δ being the number of junctions.

BCALM (Chikhi et al., 2015) and by extension its highly parallel version BCALM 2 (Chikhi et al., 2016) are methods to compact efficiently the dBG from reads and whole genome sequences. The approach used is the inverse of the one of TwoPaCo: Instead of identifying junction k-mers, BCALM 2 progressively compacts the dBG such that junction k-mers naturally arise from it. In BCALM 2, k-mers are first partitioned in clusters with regard to their minimizer Roberts et al. (2004), the lexicographically smallest of their p-mers with $p < k$. Each k-mer x is attributed to the cluster corresponding to the minimizer min_p of its prefix $x(1, k-1)$. It is also attributed to the cluster corresponding to the minimizer min_s of the suffix $x(2, k-1)$ if $min_p \neq min_s$. This ensures that k-mers with the same left or right minimizer are in the same cluster. Then, the clusters are compacted in parallel based on their respective $(k-1)$-length overlaps and minimizers. Each such compaction produces a set of strings with each string of length $k + \eta - 1$ being the compaction of η k-mers. Each compacted string of cluster min might have a left (reciprocally right) *lonely end*, a $(k-1)$-length prefix (reciprocally suffix) for which min is not the minimizer. These strings are candidates to merge with strings of other clusters in a reuniting step.
Note that TwoPaCo and BCALM 2 build the compacted dBG of single or multiple genomes, but neither index the dBG nor store the dataset identity from which the k-mers originate.

The usage of cdBGs (Section 1.2.2.1) in a pan-genome context was introduced by the *de novo* assembler **Cortex** (Iqbal et al., 2012). In a pan-genomic context, colors of the cdBG represent genomes in which the k-mers occur. Cortex can perform assembly of single genomes and populations of genomes that scale to eukaryotic genome sizes, and its algorithms can genotype simple and complex variants. Bubble calling and path divergence algorithms make extensive use of the colors to aggregate information from different samples and accurately identify the variant types. The cdBG data structure of Cortex is a hash table in which hashed k-mers are associated to various information such as colors and coverage.

Vari (Belk et al., 2016) is also a succinct data structure for indexing a pangenome as a cdBG. It uses the BOSS representation (Bowe et al., 2012) of succinct dBGs based on an adaptation of the FM-Index. The BOSS representation of a dBG G is defined by the edge-BWT of G and two bitvectors. The edge-BWT of G is the sequence of edge labels sorted according to the edges' co-lexicographic order of their starting nodes. The bitvectors register the positions of specific edges for each node. Vari builds the BOSS representation of the union of individual dBGs and stores additionally a two-dimensional binary array containing the colors of each edge. Vari was compared to Cortex on multiple sets of similar whole genome sequences by constructing the cdBG and performing a bubble calling algorithm. Results show that Vari outperforms Cortex regarding the memory efficiency while Cortex is the most time efficient algorithm.

2.6 k-mer based Methods

As q-gram indexes, also called k-mer indexes, require large amount of memory and are often used for sequence alignment purposes, Claude et al. (2010) proposed two compressed q-gram indexes dedicated to highly repetitive sequence collections. The first method divides a sequence collection of total length N into blocks of length l and establishes an index that associates every occurring q-gram to a list of blocks in which it appears. The sequence collection is encoded using a grammar-based compressor that can efficiently encode long repetitions and provides linear time decompression. Lists of blocks are first delta-encoded to limit the size of each number in the list. Then, each number is encoded using a variable-length encoding technique that assigns a shorter encoding to smaller numbers. Finally, LZ-77 (Ziv and Lempel, 1977) compresses the previously encoded list to take advantage of its internal repetitions. Search time for a q-gram is in $\mathcal{O}(N \cdot (1 - (1 - \frac{l}{N})^{occ}))$ average time and $\mathcal{O}(\mathsf{Min}(l \cdot occ, N))$ worst-case time, with occ being the occurrence number of a searched q-gram. The second proposed index is a grammar-compressed self-index based on a Straight-Line Program (SLP), a grammar generating a unique string. Indexing a collection of sequences based on SLPs of minimum size can achieve some compression, but computing it is an NP-complete problem. Instead, the authors proposed to use a grammar-based compressor that does not exactly generate an SLP but provides a good heuristic for this problem. This self-index requires

$\gamma \cdot (\log r + \log \gamma + \frac{\log N}{|\mathcal{A}|}) + r \cdot (3 \log r + \log N)$ bits where γ is the length of the final compressed stream, r is the number of rules and $\mathcal{A}$ is the grammar used. Worst-case running time is in $\mathcal{O}((q \cdot (q + \log N) + occ) \log N \log r)$.

The **Sequence Bloom Tree** (SBT) from Solomon and Kingsford (2016) is a data structure for large scale querying of genomic experiments. It is designed as a binary tree with BFs as vertices. Leaves of the tree approximately represent genomic experiments: k-mers are extracted from each such experiment and inserted into the BF of the corresponding leaf. An internal vertex is the union of its two children BFs, i.e. a BF in which an element is set to 1 if the element at the same position in at least one of the two children is 1. As BFs generate false positives, leaves do not represent dBGs but approximations of k-mer pools instead. Although the BF size must be the same for all vertices of the tree, BFs are compressed using RRR Raman et al. (2007) such that over-sized BFs have a higher compression ratio. An SBT is queried with a pattern P by decomposing it into k-mers which are in turn used to query the SBT in a top-down manner. For each vertex v of an SBT t traversed during the querying, the subset of k-mers that are present according to the vertex BF is determined. This subset is then used to query $children(v, t)$. It enables to prune the search for branches that do not contain the queried k-mers. A match of pattern P in a genomic experiment represented by leaf v is reported if $\theta \cdot (|P| - k + 1)$ k-mers from P are reported present in v with $0 \leq \theta \leq 1$ being a user-defined threshold. Because of the false positives and the heuristics used for the pattern matching, the number of experiments in which P occurs is a subset of the ones reported by the SBT.

CHAPTER III

Pan-genome Indexing

3.1 Introduction

As described in Chapter II, efficient tools have been proposed to index and analyze pan-genomes. However, such methods and data structures do not cover all expected features for pan-genome analysis. Most of them operate only on draft or finished assemblies as input, while such assemblies are available only for a small fraction of species. Furthermore, hundreds or thousands of such assemblies might be required to characterize the pan-genome of a species, a number far much larger than what is available in most cases. By the end of February 2017, the National Center for Biotechnology Information (NCBI) Genome database (NCBI, 2017) contained 23,004 assembled genomes for which about 85 % only have one assembly available. However, unassembled reads abound in databases and represent the vast majority of data available. By the end of February 2017, the NCBI Sequencing Read Archive (SRA) database (NCBI, 2007) contained about 9.8 petabases of reads. Also, methods using an assembly as reference introduce a bias in the analysis towards the reference. Finally, it has been shown that assembly errors can lead to an over-estimation of the number of genes inferred from an assembled genome (Denton et al., 2014). It might cause an over-estimation of the size and growth of the core, accessory and singleton genomes. Hence, an ideal data structure indexing a pan-genome should be reference-free and consider assemblies as well as reads as input to take advantage of all the data available in genomic databases.

Most methods presented in the previous chapter also use time or memory costly algorithms to compute variants and similarities between the input genomes,

often relying on sequence alignments. Some of these tools simply leave to the user the burden of obtaining a reference and a set of variants with respect to this reference. Rather, a data structure for pan-genome indexing should be capable of computing variants without using sequence alignment methods. More importantly, almost all methods we are aware of consider that pan-genomes are immutable despite the fact that they are continuously growing with newly sequenced genomes. Indeed, large scale sequencing projects such as the 1000 Genomes Project (1000 Genomes Project Consortium, 2015) might take years to complete and regularly release new sequenced genomes. Because methods indexing and analyzing pan-genomes are not incremental, any new genome added to the pan-genome causes its entire index to be recomputed.

In this chapter, we propose a new lightweight data structure for indexing a pan-genome as a cdBG (Section 1.2.2.1), the Bloom Filter Trie (BFT). It is alignment-free, reference-free, incremental and considers assemblies as well as reads in input. The BFT provides insertion and look-up operations for strings of fixed length associated with a set of colors. The data structure is described in the next section, followed by the operations it supports. Then, a description of the traversal method of a cdBG stored as a BFT is provided. Finally, experimental results showing the performance of the data structure are presented. The work described in this chapter was published in (Holley et al., 2015, 2016) and is joint work with Roland Wittler and Jens Stoye. Note that in the figures of this chapter, grey colored labels and arrows are for illustration but are not stored in practice.

3.2 The Bloom Filter Trie

The Bloom Filter Trie (BFT) is a data structure based on the burst trie (Section 1.2.3.2) that implements a cdBG (Section 1.2.2.1). Each input genome represented by a set of assembled sequences or by a set of reads is first decomposed into its constituent k-mers. These k-mers are inserted into the BFT with colors representing the genomes in which they occur.

In the following, let $bft = (V_{bft}, E_{bft})$ be a trie that we name BFT. The BFT is created for a certain value of k in which we assume that k is a multiple of an integer l such that k-mers can be split into $\frac{k}{l}$ equal-length substrings. The

BFT has two types of vertices: *container vertices* and *color set vertices*. A container vertex is a list of *containers* which are *compressed* or *uncompressed*. Such a list starts with zero or more compressed containers and finishes with at most one uncompressed container. A color set vertex contains a color set and is a leaf of the BFT. An edge $(v, v') \in E_{bft}$ starting from the compressed container of a container vertex v and pointing to a vertex $v' \in children(v, bft)$ represents a substring of length l. Thus, the maximum height of a BFT bft is $height_{max}(bft) = \frac{k}{l}$ since a k-mer is the concatenation of $\frac{k}{l}$ substrings of length l, each of which is represented by at most one edge in a path starting from the root. Such a path ends with a color set vertex. Indeed, a color set vertex is always located at depth $j = height_{max}(bft)$ while a container vertex is always located at a depth $i < j$. A k-mer may be represented by a path having less than $height_{max}$ edges because its suffix may be stored in an uncompressed container, together with the color set of the k-mer. The *bursting* method replaces an uncompressed container by a compressed one in order to represent the indexed k-mers more efficiently. This method is adapted from the burst trie and is described in the following sections with a more detailed description of the BFT.

3.2.1 Uncompressed Container

An uncompressed container of a container vertex v in a BFT is used to index a small number of k-mer suffixes and their colors. It is a limited capacity set of tuples $<s, color_{p \odot s}>$ where $p \odot s$ is a k-mer from which p is the prefix represented by the path from the root to v, s is the suffix and *color* is the color set associated to the k-mer $p \odot s$. Tuples are lexicographically ordered in the uncompressed container according to their suffixes. The set of tuples is implemented with a dynamic array.

When the number of suffixes stored exceeds the capacity ϕ of the uncompressed container, it is burst into a compressed container to represent more efficiently the suffixes and colors it contains. In this process, each suffix s of the uncompressed container is split into a prefix s_{pref} of length l and a suffix s_{suf} of length $|s| - l$ such that $s = s_{pref} \odot s_{suf}$. Prefixes are stored in a new compressed container replacing the uncompressed one in vertex v. Such prefixes are the labels of edges starting from the compressed container and ending on new children containing new uncompressed containers storing the suffixes and their color sets. Suffixes

sharing a common prefix are stored in the same child. In the following, we refer to the prefixes of k-mer suffixes stored in compressed containers as *edge labels*. An example of a BFT and a bursting is given in Figure 3.1.

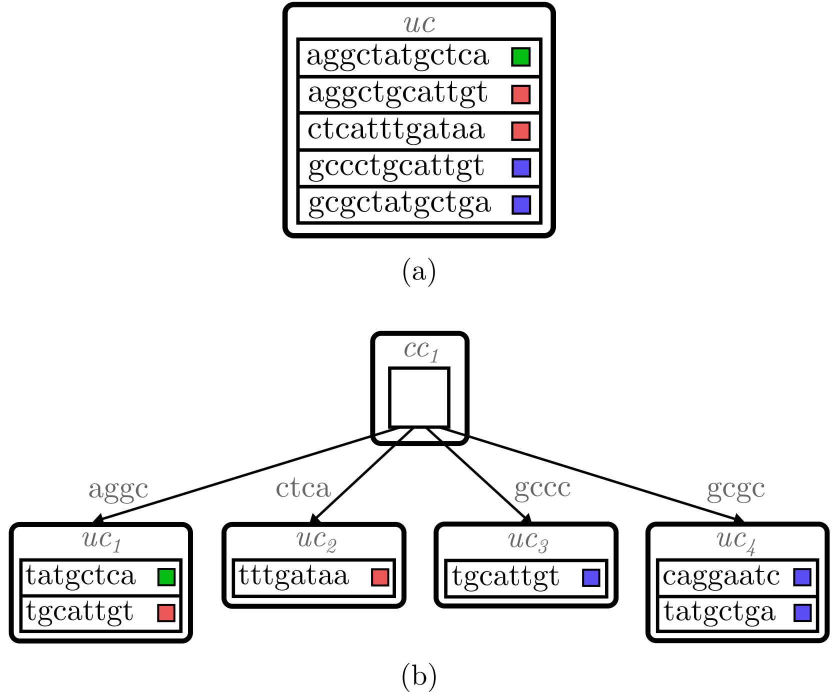


Figure 3.1: Insertion of six suffixes (that are here complete k-mers) with different colors into a BFT with $k = 12$, $l = 4$ and $\phi = 5$. In (a), the first five suffixes are inserted at the root into an uncompressed container uc. When a sixth suffix “gcgccaggaatc” is inserted, uc exceeds its capacity and is burst, resulting in the BFT structure shown in (b) with one compressed container and four uncompressed containers. Note that in practice, container vertices might have more than one container and suffixes might have more than one color.

3.2.2 Compressed Container

A bursting replaces an uncompressed container by a compressed one, used to store $q \leq \phi$ edge labels in compressed form (in Figure 3.1(b), $q = 4$) and their corresponding edges pointing to children containing the suffixes. A compressed container is composed of multiple layers. The first layer is a BF (Section 1.2.5) used for the approximate membership of edge labels while the second layer is based on an exact representation of the edge labels. Although those two layers contain redundant information, the primary purpose of the first layer is to accelerate container insertion and look-up as it can be queried in constant time

rather than linear time for the second layer. BFs of compressed containers are denoted by bf and each such BF is a bit array of length m associated with f hash functions. An example of a BF for the edge labels of Figure 3.1b is provided in Figure 3.2.

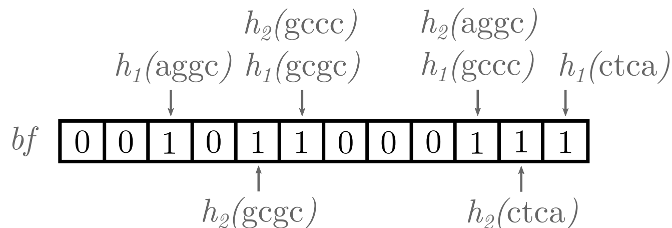


Figure 3.2: BF of four edge labels "aggc", "ctca", "gccc" and "gcgc" with $f = 2$ hash functions h_1 and h_2.

As a BF is a one-sided data structure that generates false positives when queried, it is necessary to have an exact representation of the edge labels in a second layer of the compressed containers. An ideal index for this task is a path and level compacted tree, but as described in Section 1.2.3, tree representations can be highly memory inefficient and they must be designed with special care. For this purpose, we propose a representation of an edge label tree relying on a hierarchy of arrays. It is based on a method called *quotienting* from Bender et al. (2012) that was first suggested by Knuth (1998). In his book "The Art of Computer Programming: Sorting and Searching", Knuth describes a simple form of hashing called *division hashing* in which the hash fingerprint of a key e is $h(e, i) = (\lfloor \frac{e}{i} \rfloor \bmod m) + 1$ and corresponds to the quotient of the key e divided by an integer i. This method is generally not used with hash tables because of its poor hashing performance. Bender et al. (2012) reused this concept as the core method of the *Quotient filter*, an approximate membership query data structure supporting all operations of the BF as well as deletion and resizing. In the Quotient filter, the division hashing is used differently: a key e is first hashed with a universal hash function and its computed z bits fingerprint is partitioned into its λ leftmost bits, the quotient, and the μ rightmost bits, the remainder, such that $z = \lambda + \mu$. Naturally, the quotient is a binary prefix of the fingerprint and the remainder is a binary suffix of the fingerprint. While all remainders are stored in the Quotient filter, the same quotients are stored only once with additional information to retrieve all remainders of each quotient. Although efficient, the Quotient filter cannot be used to index edge labels in compressed containers, because only the hash fingerprint is stored and not the

keys themselves. Furthermore, false positives occur in such a data structure if and only if two different keys hash to the same fingerprint. Instead, the BFT uses the quotienting method in the compressed containers by directly splitting each edge label *label* into a prefix $label_p$ and a suffix $label_s$ with respective binary representations α (the quotient) and β (the remainder) of length λ and μ bits. The alphabet we consider is the DNA alphabet $\mathcal{A} = \{\mathrm{a}, \mathrm{c}, \mathrm{g}, \mathrm{t}\}$ for which each symbol can be stored using two bits. Edge label prefixes and suffixes are indexed in a three level tree: Edges between the first and second level index the prefixes while edges between the second and third level index the suffixes. An example of such a tree (cf. Figure 3.1b) is provided in Figure 3.3.

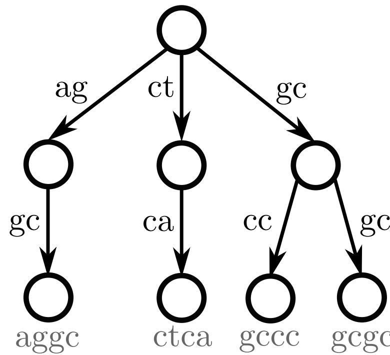


Figure 3.3: Tree representation of four edge labels “aggc”, “ctca”, “gccc” and “gcgc”.

To represent such a tree, three structures *pref*, *suf* and *clust* are used:

- *pref* represents the root of the edge label tree and its out-going edges. It is a bit array of length 2^λ, initialized with 0s and used to record prefix presence exactly. Here the binary representation α of a prefix $label_p$ is interpreted as an integer such that $pref[\alpha] = 1$ records the presence of $label_p$ in the edge label tree;

- *suf* represents the leaves of the edge label tree and their in-going edges. It is an array of q suffixes $label_s$ sorted in ascending lexicographic order of *label*;

- *clust* represents the vertices at the second level of the edge label tree. It is an array of q bits, one per suffix of array *suf*, that represent cluster starting points. A cluster is a list of consecutive suffixes in array *suf* sharing the same prefix. It represents a set of leaves in the edge label tree sharing the

same parent. Each cluster has an index $1 \leq i_{cluster} \leq 2^{\lambda}$ and a start position $i_{cluster} \leq pos_{cluster} \leq q$ in the array *suf*. Position $pos_{cluster}$ in array *clust* is set to 1 to indicate that the suffix in $suf[pos_{cluster}]$ starts a cluster because it is the lexicographically smallest suffix of its cluster. A cluster contains $n \geq 1$ suffixes and, therefore, each position $pos_{cluster} < i < pos_{cluster} + n$ in array *clust* is set to 0. The end of a cluster is implicitly indicated by the beginning of the next cluster or if $pos_{cluster} \geq q$.

The exact representation of the edge labels shown in Figure 3.3 is shown in Figure 3.4.

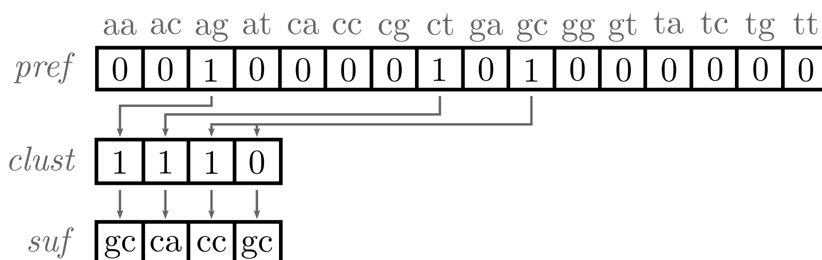


Figure 3.4: Exact representation of four edge labels "aggc", "ctca", "gccc" and "gcgc" in a compressed container with $|label_p| = 2$ and $|label_s| = 2$.

Each compressed container has a third and last layer which is the array *edge*. It contains q edges, one for each edge label stored, i.e., one for each suffix of array *suf*. Each of these edges points to a child.

Arrays *suf*, *clust* and *edge* are dynamic such that function $\mathsf{Insert}_{\mathsf{array}}(e, pos, A)$ inserts an item e at position *pos* of the array A by reallocating it and shifting every item having an index $i \geq pos$ by one position in $\mathcal{O}(|A|)$ time. The dynamic nature of those arrays allows to store contiguously in memory all children accessible through the edges of a compressed container. In practice, only two pointers must be stored per compressed container, one for all container vertices and one for all color set vertices, to access all children of a compressed container. Therefore, the size of edges in a BFT is negligible.

The size of q edge labels indexed in a compressed container is $m + 2^{\lambda} + q \cdot (\mu + 1)$ bits. The prefix length $|label_p|$, its bit length λ and the BF size m are chosen such that the set of edge labels stored in a compressed container does not occupy more memory than their original representation in an uncompressed container, i.e., $m + 2^{\lambda} \leq q \cdot (\lambda - 1)$. Each edge label inserted after a bursting occupies only $\mu + 1$ extra bits. When the average size per edge label stored is close to $\mu + 1$

bits, the arrays *pref*, *suf* and *clust* can be recomputed by increasing $|label_p|$ and decreasing $|label_s|$, such that $2^{\lambda'} + q \cdot \mu' < 2^{\lambda} + q \cdot \mu$, where λ' and μ' are the values of λ and μ, respectively, after resizing.

3.2.3 Color Set

A color set is represented by a dynamic bit array *color* initialized with 0s. Each color has an index i_{color} such that $color_x[i_{color}] = 1$ records that k-mer x has color i_{color}. Color sets can be compressed by storing sets that are identical for multiple k-mers once. To this end, a list of all color sets occurring in the BFT is built and sorted in decreasing order of total size which we define as the number of k-mers sharing a color set multiplied by its size. Then, by iterating over the list, each color set is added incrementally to an external array if the integer encoding its position in the array uses less space than the size of the color set itself. Finally, each color set present in the external array is replaced in the BFT by its position in the external array.

3.3 Operations

The BFT supports all operations necessary for storing, traversing and searching a pan-genome. Here we describe the most basic ones of them, insertion and look-up of k-mers and their colors, as well as how the sets of colors are compressed. The traversal of the graph is discussed in the next section.

The operations described in the following use three auxiliary functions:

- $\mathsf{HammingWeight}(\alpha, \mathit{pref})$ counts the number of 1s in $\mathit{pref}[1..\alpha]$ and corresponds to the number of prefixes represented in array *pref* that are lexicographically smaller than or equal to a prefix $label_p$ with binary representation α of length λ bits. This requires $\mathcal{O}(2^{\lambda})$ time.

- $\mathsf{Select}(i, \mathit{clust})$ returns the position of the i-th entry 1 corresponding to the start position of cluster i in array *clust* of length q. If the entry is not found, the function returns $q+1$ as a position. While Select could be implemented

in $\mathcal{O}(1)$ time (Ferragina and Manzini, 2000), we use a more naive but space efficient $\mathcal{O}(q)$ time implementation.

- $\mathsf{Insert}_{\mathsf{cc_arrays}}(bv, s, pos, cc)$ performs two tasks. First, it inserts a binary value bv and a suffix s at position pos of arrays $clust$ and suf, respectively, in compressed container cc. Then, it creates a new edge pointing to a new child and inserts this edge at position pos of array $edge$ in compressed container cc. This function takes $\mathcal{O}(q)$ time with q being the number of edge labels in the compressed container cc. It returns the end vertex of the edge stored in $edge[pos]$.

3.3.1 Container Insertion

This section describes the functions inserting a k-mer suffix into an uncompressed container and inserting an edge label into a compressed container. These functions are used in the next section to describe how to perform a k-mer insertion in a BFT.

The function $\mathsf{Insert}_{\mathsf{uc}}$ inserts a k-mer suffix s into an uncompressed container uc. Because these containers are lexicographically sorted, the function simply performs a binary search to find the position where s must be inserted. $\mathsf{Insert}_{\mathsf{uc}}$ takes $\mathcal{O}(\log \phi + \phi)$ time with ϕ being the capacity of uc, and returns the color set associated with s such that new colors can be added into the set.

Inserting an edge and its label $label = label_p \odot label_s$ with binary representation $\alpha \odot \beta$ into a compressed container cc is achieved with the function $\mathsf{Insert}_{\mathsf{cc}}$ provided in Algorithm 1. First, line 1 records the presence of $label$ in bf in $\mathcal{O}(f)$ time. Next, line 3 records the exact presence of prefix $label_p$ by setting the value of $pref[\alpha]$ to 1. While those two insertions are simple, inserting $label_s$ into its cluster is a more complicated task, especially in the case where the cluster already exists as it must be expanded. First, line 4 computes in $\mathcal{O}(2^\lambda)$ time the Hamming weight i of $label_p$, i.e., the index $id_{cluster}$ of the (potentially new) cluster in which suffix $label_s$ will be inserted. From the index $id_{cluster}$, line 5 computes the cluster start position $pos_{cluster}$ using Select. If $pref[\alpha] = 1$ prior to insertion, it means a cluster of $n \geq 1$ suffixes already exists for prefix $label_p$. Thus, lines 8 to 10 compute the insertion position pos of $label_s$ in cluster $id_{cluster}$ such that

after the insertion of $label_s$ into suf, suffixes of the cluster remain lexicographically sorted. In lines 11 to 18, suffix $label_s$ is inserted into $suf[pos]$ such that $pos_{cluster} \leq pos \leq pos_{cluster} + n$. If $pos = pos_{cluster}$, $label_s$ starts its cluster: a bit 1 is inserted into $clust[pos]$ and $clust[pos+1]$ is set to 0. Otherwise, a 0 is inserted into $clust[pos]$. If $pref[\alpha] \neq 1$ prior to insertion, cluster $id_{cluster}$ is a new cluster: a bit 1 is inserted into $clust[pos_{cluster}]$ to create a new cluster, and $label_s$ is inserted into $suf[pos_{cluster}]$ in line 20. Lines 5 to 20 take $\mathcal{O}(q)$ time. Hence, function $\mathsf{Insert_{cc}}$ takes $\mathcal{O}(f + 2^{\lambda} + q)$ time. The function returns the end vertex of the edge labeled with *label*. Note that if *label* is already present in the container, lines 12 and 17 ensure that it is not inserted a second time in addition to line 19 which returns the end vertex of its edge. The returned vertex is used by Algorithm 2 to recursively insert k-mer suffixes in subtries of the BFT.

Algorithm 1 $\mathsf{Insert_{cc}}(label_p \odot label_s, cc)$

1: $\mathsf{Insert_{bf}}(label_p \odot label_s, cc.bf)$
2: $prev \leftarrow cc.pref[\alpha]$
3: $cc.pref[\alpha] \leftarrow 1$
4: $i \leftarrow \mathsf{HammingWeight}(\alpha, cc.pref)$
5: $pos \leftarrow \mathsf{Select}(i, cc.clust)$
6: **if** $prev = 1$ **then**
7: $\quad prev \leftarrow pos$
8: $\quad$ **if** $pos \leq |suf|$ **and** $cc.suf[pos] < label_s$ **then** $pos \leftarrow pos + 1$
9: $\quad$ **while** $pos \leq |suf|$ **and** $cc.clust[pos] = 0$ **and** $cc.suf[pos] < label_s$ **do**
10: $\quad\quad pos \leftarrow pos + 1$
11: $\quad$ **if** $pos > |suf|$ **then return** $\mathsf{Insert_{cc_arrays}}(0, label_s, pos, cc)$
12: $\quad$ **else if** $cc.suf[pos] \neq label_s$ **then**
13: $\quad\quad$ **if** $prev = pos$ **then**
14: $\quad\quad\quad cc.clust[pos] \leftarrow 0$
15: $\quad\quad\quad$ **return** $\mathsf{Insert_{cc_arrays}}(1, label_s, pos, cc)$
16: $\quad\quad$ **else return** $\mathsf{Insert_{cc_arrays}}(0, label_s, pos, cc)$
17: $\quad$ **else if** $cc.clust[pos] = 1$ **and** $prev \neq pos$ **then**
18: $\quad\quad$ **return** $\mathsf{Insert_{cc_arrays}}(0, label_s, pos, cc)$
19: $\quad$ **else return** $cc.edge[pos]$
20: **else return** $\mathsf{Insert_{cc_arrays}}(1, label_s, pos, cc)$

As an example, the internal representation of the compressed container shown in Figure 3.1b after insertion of the edge label “gtat” is given in Figure 3.5. The presence of prefix “gt” is recorded in $pref[12]$. Then, its cluster index and start position are computed as 4 and 5, respectively. Consequently, after reallocation of arrays suf and $clust$, suffix “at” is inserted in $suf[5]$ and $clust[5]$ is set to 1 to

indicate *suf* [5] starts a new cluster.

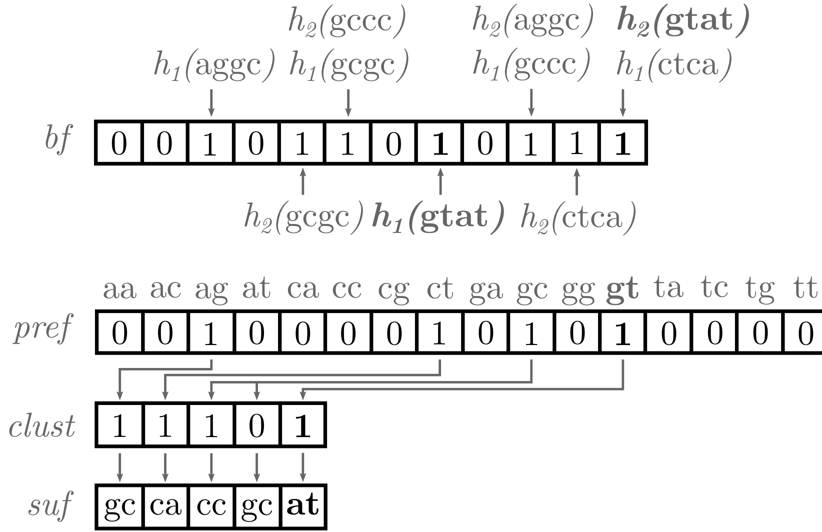


Figure 3.5: Insertion of edge label "gtat" in a compressed container with four edge labels "aggc", "ctca", "gccc" and "gcgc". Inserted or changed parts are highlighted. Array *edge* is not represented.

3.3.2 Tree Insertion

Function TreeInsert described in Algorithm 2 inserts a k-mer into a BFT *bft*. Such an insertion is performed recursively by traversing *bft* from the root in a top-down manner. In each traversed container vertex v, either an edge and its label are inserted into a compressed container, or a k-mer suffix into an uncompressed container using the functions of Section 3.3.1. For this purpose, TreeInsert iterates over the containers of v from the head to the tail of the container list and the insertion of the edge label *label* into a compressed container is only triggered if its BF reports *label* as present in the container. If it is indeed present in the container, the insertion continues recursively on the end vertex of the edge labeled with *label* in the container. At this point, *label* might not be present in the container because it is a false positive of the BF. False positives of compressed container BFs are "recycled", which is a nice property of the BFT: the BF remains unchanged, i.e., it is not necessary to perform line 1 of Algorithm 1, and only *pref*, *suf*, *clust* and *edge* are updated as described in lines 2 to 20 of Algorithm 1. It allows to limit the number of burstings and compressed containers in a BFT. Algorithm 2 is initially called as TreeInsert($x, l, root, bft$) to insert a k-mer x into a BFT *bft* with *root* being the root vertex of *bft*. In the worst case, all container

vertices on a traversed path represent all possible edge labels and the BFs have a false positive ratio of 0. In such case, each traversed container vertex has $\left\lceil \frac{|\mathcal{A}|^l}{\phi} \right\rceil$ containers. The longest path of a BFT has $\frac{k}{l}$ container vertices. Therefore, the worst case time of $\mathsf{TreeInsert}$ is $\mathcal{O}\left(\frac{k}{l} \cdot \left(\left\lceil \frac{|\mathcal{A}|^l}{\phi} \right\rceil \cdot f + 2^\lambda + q\right)\right)$. The function returns the color set associated to the inserted k-mer such that colors can be inserted in it.

Algorithm 2 $\mathsf{TreeInsert}(s, l, v, \mathit{bft})$

```
1: for each container cont in v do
2:     if cont is compressed then
3:         if MayContain(s(1, l), cont.bf) then
4:             v_child ← Insert_cc(s(1, l), cont)
5:             if v_child is a color set vertex then return v_child.color
6:             else return TreeInsert(s(l + 1, |s| − l), l, v_child, bft)
7:     else return Insert_uc(s, cont)
```

3.3.3 Container Look-up

Similarly to the previous sections, we describe here the functions looking for a k-mer suffix in an uncompressed container and looking for an edge in a compressed container. These functions are used in the next section to describe how to perform a k-mer look-up in a BFT.

Searching for a k-mer suffix s in an uncompressed container uc is performed by the function $\mathsf{Contain}_{\mathsf{uc}}$ which is similar to the function $\mathsf{Insert}_{\mathsf{uc}}$. A binary search of suffix s is performed in $\mathcal{O}(\log \phi)$ time, where ϕ is the capacity of uc. It returns the set of colors associated with the suffix s in uc if it is found or an empty set otherwise.

The function $\mathsf{Contain}_{\mathsf{cc}}$, testing whether an edge label $label = label_p \odot label_s$ with binary representation $\alpha \odot \beta$ is stored in a compressed container cc, is given in Algorithm 3. Line 1 verifies the presence of $label$ in bf and its prefix $label_p$ in the array $pref$ in $\mathcal{O}(f)$ time. If $label_p$ is present, line 2 computes in $\mathcal{O}(2^\lambda)$ time the Hamming weight i of $label_p$, i.e., the index of the cluster in which suffix $label_s$ is possibly situated. Line 3 locates the rank of i, i.e., the start position of the cluster, and lines 4 to 7 compare the suffixes of the cluster to $label_s$. Lines 3 to 7 are computed in $\mathcal{O}(q)$ time. Algorithm 3 has therefore a worst case time of

$\mathcal{O}(f + 2^\lambda + q)$. If *label* is found, the algorithm returns the end vertex of the edge labeled with *label*. Otherwise, a vertex containing an empty color set is returned.

Algorithm 3 $\mathsf{Contain_{cc}}(label_p \odot label_s, cc)$

1: **if** $\mathsf{MayContain}(label_p \odot label_s, cc.bf)$ **and** $cc.pref[\alpha] = 1$ **then**
2: $\quad i \leftarrow \mathsf{HammingWeight}(\alpha, cc.pref)$
3: $\quad start \leftarrow \mathsf{Select}(i, cc.clust)$
4: $\quad pos \leftarrow start$
5: $\quad$ **while** $pos \leq |suf|$ **and** $(pos = start$ **or** $cc.clust[pos] = 0)$ **do**
6: $\quad\quad$ **if** $cc.suf[pos] = label_s$ **then return** $cc.edge[pos]$
7: $\quad\quad$ **else if** $cc.suf[pos] < label_s$ **then** $pos \leftarrow pos + 1$
8: $\quad\quad$ **else** $pos \leftarrow |suf| + 1$
9: **return** a vertex containing $\varnothing$

3.3.4 Tree Look-up

The function $\mathsf{TreeContain}$, testing whether a k-mer x is present in a BFT *bft*, is given in Algorithm 4. Each container vertex v is the root of a subtrie representing k-mer suffixes. The BFT look-up function traverses *bft* top-down from the root and, for a container vertex v, queries its containers from the head to the tail of the container list for the k-mer suffix $x_{suf} = x(l \cdot depth(v,t) + 1, |x| - l \cdot depth(v,t))$. If the queried container is compressed, its BF is queried for $x_{suf}(1,l)$ using the function $\mathsf{MayContain}$ in $\mathcal{O}(f)$ time. In case of a positive answer, the function $\mathsf{Contain_{cc}}$ is used for an exact membership of $x_{suf}(1,l)$. The usage of $\mathsf{MayContain}$ before $\mathsf{Contain_{cc}}$ allows to reduce the querying time of this container from $\mathcal{O}(f + 2^\lambda + q)$ to $\mathcal{O}(f)$ if the BF reports $x_{suf}(1,l)$ as not present. If $x_{suf}(1,l)$ is found, the traversing procedure continues recursively on the corresponding child. Otherwise, the absence of $x_{suf}(1,l)$ indicates the absence of x in *bft* since $x_{suf}(1,l)$ cannot be in another container of v because of the tree insertion process explained in Section 3.3.2. If the container is uncompressed, the presence of x_{suf} is detected using the function $\mathsf{Contain_{uc}}$. As an uncompressed container has no children, a match indicates the presence of the k-mer. Algorithm 4 is initially called as $\mathsf{TreeContain}(x, l, root, bft)$ and its worst case time is $\mathcal{O}\left(\frac{k}{l} \cdot \left(\left\lceil \frac{|\mathcal{A}|^l}{\phi} \right\rceil \cdot f + 2^\lambda + q\right)\right)$. If the queried k-mer is found, its color set is returned. Otherwise, an empty color set is returned.

Algorithm 4 $\mathsf{TreeContain}(s, l, v, \mathit{bft})$

```
1: for each container cont in v do
2:     if cont is compressed then
3:         if MayContain(s(1, l), cont.bf) then
4:             v_child ← Contain_cc(s(1, l), cont)
5:             if v_child is a color set vertex then return v_child.color
6:             else return TreeContain(s(l + 1, |s| − l), l, v_child, bft)
7:     else return Contain_uc(s, cont)
8: return ∅
```

3.4 Successors and Predecessors Traversing

The BFT is an implementation of the cdBG used to index a pan-genome. In order to analyze a pan-genome indexed with a BFT, it is necessary to traverse the cdBG it represents. For example, the traversal can be used to extract all paths of the cdBG for which the k-mers have all colors in their color sets. Such paths belong to the core genome.

Let bft be a BFT representing a cdBG G. For a k-mer x, visiting all its predecessors or successors in G, denoted by $\mathit{pred}(x, G)$ and $\mathit{succ}(x, G)$, respectively, implies the look-up of $|\mathcal{A}|$ different k-mers in bft. Such a look-up would visit in the worst case $|\mathcal{A}| \cdot \mathit{height}_{max}(\mathit{bft})$ container vertices in bft. This section describes how to reduce the number of vertices and containers visited in bft during the traversal of a vertex in G.

Observation 1. Let G be a cdBG represented by a BFT bft and x a k-mer corresponding to a vertex of G. All k-mers of $\mathit{succ}(x, G)$ share $x(2, k-1)$ as a common prefix and therefore share a common subpath in bft starting at the root. However, all k-mers of $\mathit{pred}(x, G)$ have different first symbols and, therefore, except for the root of bft do not share a common subpath. Hence, the maximum number of visited container vertices in bft for all k-mers of $\mathit{succ}(x, G)$ is $\mathit{height}_{max}(\mathit{bft})$ and for all k-mers of $\mathit{pred}(x, G)$ is $1 + |\mathcal{A}| \cdot (\mathit{height}_{max}(\mathit{bft}) - 1)$.

Lemma 1. Let G be a cdBG represented by a BFT bft, x a k-mer in bft and v a container vertex of bft that terminates the shared subpath of the k-mers in $\mathit{succ}(x, G)$. If $\mathit{depth}(v, \mathit{bft}) = \mathit{height}_{max}(\mathit{bft}) - 1$, the suffixes of $\mathit{succ}(x, G)$ may be stored in any container of v. If not, they are stored in the uncompressed container of v.

Proof. In a BFT bft representing a cdBG G, a container vertex v is the root of a subtrie storing k-mer suffixes of length $l \cdot (height_{max}(bft) - depth(v, bft))$ with $l = \frac{k}{height_{max}(bft)}$. Let s be the suffix of a k-mer from $succ(x, G)$ that is rooted at a vertex $v \in V_{bft}$. If $depth(v, bft) \neq height_{max}(bft) - 1$ but s is rooted at a compressed container in v, then this compressed container stores $s(1, l)$, and $s(l+1, |s|-l)$ is rooted in one of its children. As the divergent symbol between the k-mer suffixes of $succ(x, G)$ is in position $|s| - 1$, this symbol is in $s(l+1, |s|-l)$, rooted at one child of this compressed container. Therefore, v does not terminate the common subpath shared by $succ(x, G)$ k-mers. □

Observation 1 and Lemma 1 prove that the only two cases where a look-up of $pred(x, G)$ or $succ(x, G)$ must search in different containers of a vertex are:

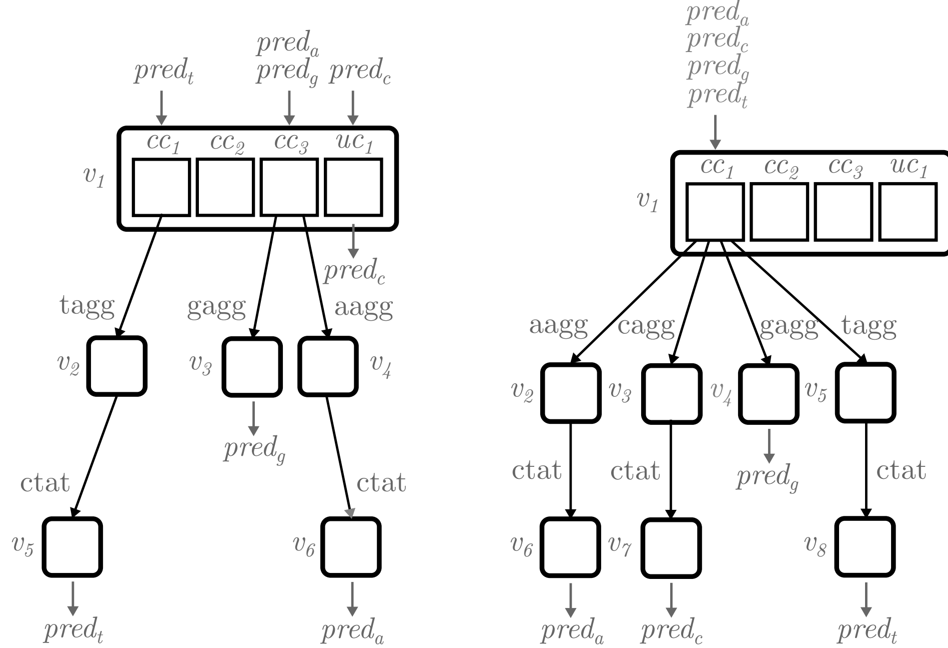
- searching at the root of bft for k-mers of $pred(x, G)$,
- if $depth(v, bft) = height_{max}(bft) - 1$, searching at vertex v for suffixes of $succ(x, G)$.

Restricting the hash functions used in the compressed containers to take only positions 2 through $l - 1$ of edge labels into account allows to limit the search space.

Lemma 2. Let bft be a BFT in which the f hash functions h_i of bf have the form $h_i(label) : label(2, l-1) \rightarrow \{1, .., m\}$ for $i = 1, ..., f$. Then, for a vertex container v of bft and an edge label $label$, all possible edge labels $label' = a_1 \odot label(2, l-1) \odot a_2$ are contained in the same container of v.

Proof. Assume a k-mer suffix s inserted into a container vertex v of bft. A look-up for s analyzes the containers of v from the head to the tail of the container list. In the worst case, s can be rooted, according to BFs, from all compressed containers as a true positive or as a false positive. However, a look-up stops either on the first compressed container claiming to contain the edge label $label = s(1, l)$, or on the uncompressed container. Therefore, as the hash functions of the BFs consider only $label(2, l - 1)$, a look-up will stop on the same container for any edge label $label' = a_1 \odot label(2, l - 1) \odot a_2$. □

Figures 3.6 and 3.7 illustrate how the containers of container vertices are traversed for the insertion of k-mer "aggctatgctca" predecessors and successors, before and after restricting the BF hash functions.

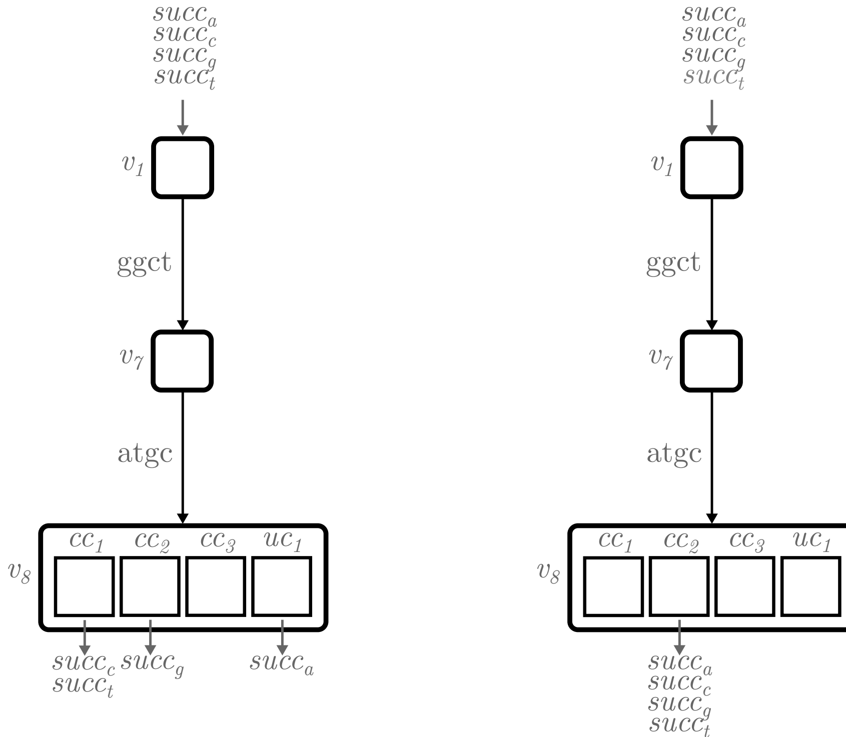


(a) Before restricting the BF hash functions. Multiple containers are traversed at the root vertex v_1.

(b) After restricting the BF hash functions. Only one container is traversed for each container vertex.

Figure 3.6: Traversed paths for predecessors $pred_a$ of k-mer x = "aggctatgctca" such that $pred_a = a \odot x(1, |x|-1)$ for all $a \in \mathcal{A}$. Content of vertices is only shown for the root, other vertices only have one traversed container and are represented with an empty rounded box. Color set vertices are not represented.

As a consequence of Lemma 2, all edge label *label* stored or to store in arrays *pref*, *suf* and *clust* of BFTs are modified such that $label' = label(2, l) \odot label(1, 1)$, which guarantees that any $label'' = label'(1, l-2) \odot a_2 \odot a_1$ are in the same container. Furthermore, suffixes stored in array *suf* are required to have a minimum length of two symbols to ensure that symbols a_1 and a_2, the variable parts between the different $label''$, are stored in array *suf*. Hence, as any $label''$ share $label'(1, l-2)$ as a prefix, they share the same cluster in arrays *suf* and *clust*. Suffix prefixes $label'' = label'(1, l-1) \odot a_1$ also have consecutive suffixes in their cluster. As an example, Figure 3.8 shows the compressed container of five edge labels "aggc", "ctca", "gccc", "gcgc" and "gtat" adapted for predecessor and successor traversal. Figure 3.9 illustrates the same compressed container in which the edge label "tggc" is inserted. The edge label "tggc" is, as the prefix "aggc"



(a) Before restricting the BF hash functions. Multiple containers are traversed at the leaf v_8.

(b) After restricting the BF hash functions. Only one container is traversed for each container vertex.

Figure 3.7: Traversed paths for successors $succ_a$ of k-mer x = "aggctatgctca" such that $succ_a = x(2, |x| - 1) \odot a$ for all $a \in \mathcal{A}$. Content of vertices is only shown for the leaves, other vertices have only one traversed container and are represented with an empty rounded box. Color set vertices are not represented.

is already present in the container, a predecessor for a k-mer x with $x(1, 3)$ = "ggc".

3.5 Evaluation

In this section, we compare the BFT implementation written in C, version 0.8.5, to the SBT (Section 2.6) implementation written in C++, version 0.3.5. Indeed, the SBT presents the same characteristics as the BFT: it is alignment-free, reference-free, incremental and considers assemblies as well as reads in input by decomposing each input experiment into k-mers. The BFT implementation is available at `https://github.com/GuillaumeHolley/BloomFilterTrie`.
All evaluations were carried out using a single thread on a server with 378 GB

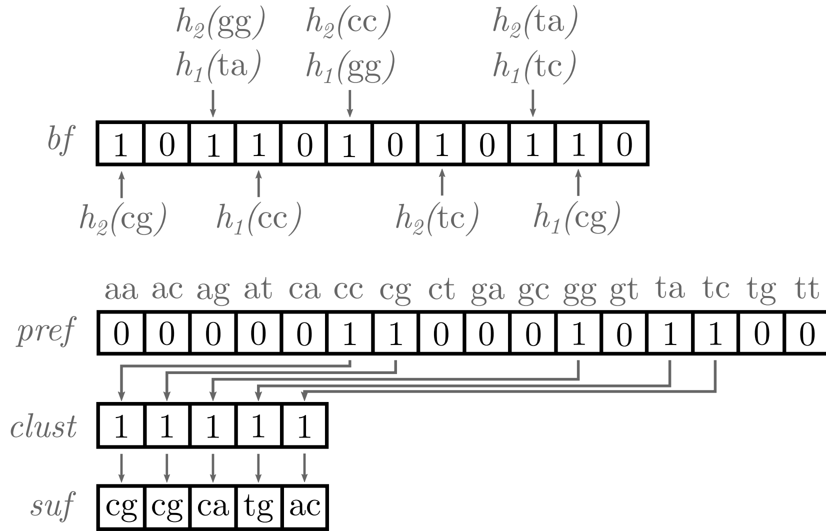


Figure 3.8: Internal representation of a compressed container with five edge labels “aggc”, “ctca”, “gccc”, “gcgc”, “gtat”, adapted as “ggca”, “tcac”, “cccg”, “cgcg”,“tatg”, respectively, for predecessor and successor traversal. Array *edge* is not represented.

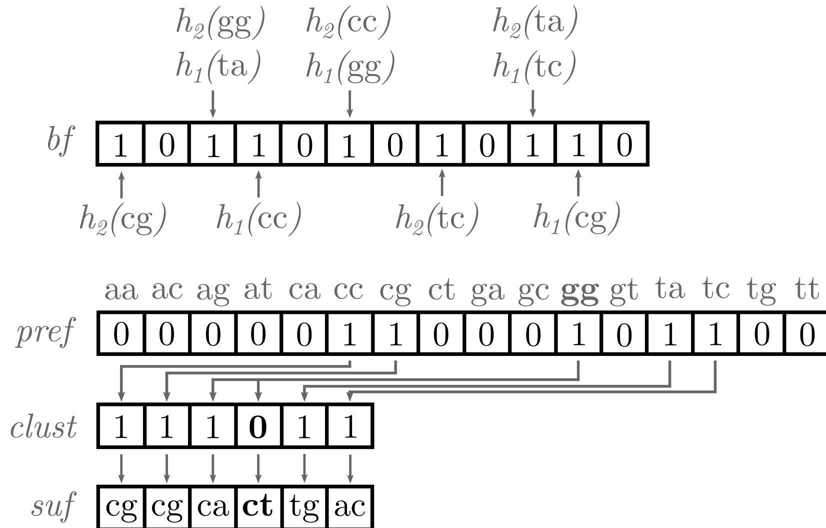


Figure 3.9: Insertion of edge label “tggc” adapted as “ggct” in a compressed container with edge labels “aggc”, “ctca”, “gccc”, “gcgc” and “gtat”. Array *edge* is not represented. Inserted parts are highlighted.

of RAM and two 8-core Intel Xeon E5-2630 v3 2.4 GHz processors. All input files were placed on a mechanical hard drive also used to write the output files. Because this hard drive was shared among users of the employed server, all times reported in the following evaluations are *user times* which do not include time spent to wait for disk input and output operations, to the contrary of wall-clock

times. Note that this is at the advantage of the SBT implementation which is mainly disk-based while the BFT implementation is mainly RAM-based (the disk is used to perform color sets compression as described in Section 3.2.3).

The BFT implementation takes as input a list of k-mers for each experiment to index such that users can employ the tool of their choice to extract k-mers from sequencing experiments. Then, extracted k-mers are inserted into a BFT. The capacity ϕ of uncompressed containers influences the compression ratio as well as the time for insertion and look-up. We chose a value of $\phi = 248$, as it showed a good compromise in practice. Also, the edge label length l determines the size of several internal structures of the BFT and how efficiently they can be stored. We selected $l = 9$, as this limits the internal fragmentation of the memory.

The SBT implementation takes as input FASTQ or FASTA files and extracts their k-mers with the tool Jellyfish (Marçais and Kingsford, 2011) which is directly integrated in the implementation. For each sequencing experiment to index, its constituent k-mers are extracted and inserted into a BF that is a leaf of the SBT. Once all leaves are constructed, they are used by the implementation to build the internal vertices. Then, all vertices are compressed using RRR (Raman et al., 2007).
Because the SBT implementation integrates k-mer counting and leaf construction into a single operation without the possibility to distinguish each step, we discarded the time to construct the leaves from the SBT construction time. It allows to be as fair as possible by not reporting the time spent by Jellyfish to count the k-mers. Hence, SBT construction time includes the time to construct the internal vertices from the leaves and the time to compress all vertices of the SBT.
Configuring the SBT implementation is not straightfoward. While the data structure is incremental, all its BFs must have the same size designed with respect to a number of k-mers to insert and a number of hash functions in the BFs. However, the number of k-mers per experiment of a dataset may be fluctuent. Furthermore, in order to avoid saturation of 1s in the BFs composing the lowest levels of the data structure, Solomon and Kingsford (2016) advise to compute the total count of k-mers for a subset of the experiments to insert and design the BFs' bit size with respect to this number. However, it is unclear how this simplistic approach

scales with new insertions in the SBT. For all computed SBTs in the following, we used one hash function for the BFs as in (Solomon and Kingsford, 2016), and computed the total count of k-mers for a subset of the experiments representing about 3.8 % of each dataset, as in the evaluation performed in (Solomon and Kingsford, 2016).

The BFT and SBT were used to index and query two pan-genome datasets, a bacterial dataset and a human tissue dataset, for which the evaluations are shown in Sections 3.5.1 and 3.5.2, respectively. For each querying evaluation, we used two different thresholds $\theta = 0.7$ and $\theta = 0.9$ indicating the ratio of k-mers from a query that must be present in an indexed sequencing experiment to have the query reported as present in this experiment.

3.5.1 *P. aeruginosa* Dataset

The dataset used for this evaluation is composed of 473 clinical isolates of *P. aeruginosa* sampled from 34 patients (NCBI BioProject PRJEB5438), resulting in 338.61 Gbp of sequences. For each data structure, we inserted all 36-mers with a minimum number of 3 occurrences in each sequencing experiment of the dataset. As the size of BFs used by the SBT implementation must be specified prior to the k-mer insertion and should be the same for all vertices, we computed from 18 isolates a total count of 123,994,112 36-mers that the SBT implementation used as the BFs' bit size. From this size and the maximum number of 36-mers extracted from a file of the dataset, a BF in a leaf of the SBT has a maximum false positive ratio of 6.2 % in this evaluation. In total, 3,270,528,926 36-mers were inserted in both data structures. The construction evaluation is shown in Table 3.1.

Table 3.1: BFT and SBT construction evaluation for 473 *P. aeruginosa* isolates. Best results are highlighted.

	BFT	SBT
Insertion time	127.74 min	$>$ 6.69 min
Main memory usage peak	3.82 GB	**0.51 GB**
Disk usage peak	**3.82 GB**	13.64 GB
Disk size	**0.68 GB**	4.83 GB

The difference of construction times between BFT and SBT can be explained by the fact that SBT construction time does not include the time spent to build the leaves. Also, the user time was measured instead of wall-clock time. Indeed, the BFT implementation is RAM-based while the SBT implementation is disk-based, suggesting that in terms of wall-clock time, the SBT construction is several times slower than its reported user time. A more accurate construction comparison using a dedicated hard drive, measuring wall-clock time and distinguishing k-mer counting from leaf construction would provide more insights into the real time performance of each data structure during construction.
While the SBT used about 3.57 times more disk space than the BFT during the construction, the BFT used about 7.46 times more main memory than the SBT. The disk size indicates that BFT is more suitable for long-term disk storage as it uses 7.1 times less disk space than the SBT.

We queried both constructed data structures with a subset of 1,000 and 10,000 sequences of length 100 bp from the sequencing experiment ERR431077. As the average size of an SBT vertex is 5.23 MB, we configured the SBT implementation to load the SBT in main memory by batch of 150 vertices such that the main memory consumption of both data structures is equivalent. The results of the querying evaluation are shown in Table 3.2.

Table 3.2: SBT and BFT querying evaluation for sequences of length 100 bp from the sequencing experiment ERR431077. Best results are highlighted.

		Time		Main memory peak	
		BFT	SBT	BFT	SBT
1,000 queries	$\theta = 0.7$	**1.66 s**	47.66 s	**786.70 MB**	815.97 MB
	$\theta = 0.9$	**1.71 s**	30.76 s	**786.59 MB**	833.77 MB
10,000 queries	$\theta = 0.7$	**4.58 s**	390.17 s	**786.75 MB**	898.28 MB
	$\theta = 0.9$	**4.01 s**	328.16 s	**786.62 MB**	895.52 MB

Results show that for a similar main memory usage, the BFT is 18 to 85 times faster to query than the SBT. The main memory usage of the BFT during querying is lower than during construction because its color sets are already compressed while during construction, the raw color sets are stored in main memory before being compressed. An interesting result of this evaluation is that for a fixed number of queries but different thresholds θ (ratio of k-mers that must be present in the queries to report the query as present), the querying time of

the BFT is stable while the querying time of the SBT varies greatly. It can be explained by the difference between the BFT and SBT indexing methods: The BFT indexes colors with respect to the k-mers while the SBT indexes the k-mers with respect to the colors. Hence, each k-mer is queried multiple times in the SBT but is queried only once in the BFT.

Finally, a last evaluation for this dataset demonstrates the operations provided by the BFT which are absent from the SBT because it is based on an approximation of the indexed k-mers. For this evaluation, we first extracted all k-mers stored in the BFT and then, used these k-mers as queries to compute the number of branching k-mers in the cdBG represented by the BFT. Table 3.3 reports the time and main memory usage of both operations.

Table 3.3: Evaluation of k-mer extraction and branching queries for a BFT constructed from 473 *P. aeruginosa* isolates.

	Time	Main memory peak
Extracting k-mers	7.40 s	786.39 MB
Branching queries	146.84 s	786.71 MB

In this evaluation, 55,355,097 36-mers were extracted and 1,518,746 of them are branching in the cdBG represented with the BFT.

3.5.2 Human Tissues Dataset

In this evaluation, we reused the dataset of the SBT evaluation in (Solomon and Kingsford, 2016). It is composed of 2,652 human tissue (blood, brain and breast) RNA-Seq experiments from the SRA database. This dataset represents about 5 TB of SRA files or about 2.7 TB of FASTA files compressed with Gzip (Ziv and Lempel, 1977). Because the k-mer length must be a multiple of a $l = 9$ in the BFT, we extracted k-mers of length $k = 18$ in order to be as close as possible to the original SBT evaluation in which $k = 20$ was used. Also, we reused the k-mer abundance thresholds computed in (Solomon and Kingsford, 2016) to discards k-mers that do not occur enough and which are then considered as sequencing errors. Hence, for a FASTA file *file* from the dataset and its corresponding size $\mathsf{Size}(\mathit{file})$, the k-mer abundance threshold is 1 for $\mathsf{Size}(\mathit{file}) \leq 300\,\text{MB}$, 3 for $300\,\text{MB} < \mathsf{Size}(\mathit{file}) \leq 500\,\text{MB}$, 10 for $500\,\text{MB} < \mathsf{Size}(\mathit{file}) \leq 1\,\text{GB}$, 20 for

1 GB < $\mathsf{Size}(\mathit{file}) \leq 3$ GB and 50 for $\mathsf{Size}(\mathit{file}) > 3$ GB. Using these abundance thresholds, 15,755,778,039 18-mers were extracted. We reused as well the size of BFs mentioned in (Solomon and Kingsford, 2016). Using this size and the maximum number of 18-mers extracted from a file of the dataset, a BF in a leaf of the SBT has a maximum false positive ratio of 3.8 % in this evaluation. The construction evaluation is shown in Table 3.4.

Table 3.4: BFT and SBT construction evaluation for 2,652 human tissue RNA-Seq experiments. Best results are highlighted.

	BFT	SBT
Insertion time	19.3 h	> 4.89 h
Main memory usage peak	144.21 GB	**18.5 GB**
Disk usage peak	**144.21 GB**	1,174.65 GB
Disk size	**13.11 GB**	92.46 GB

Similarly to the *P. aeruginosa* dataset, the difference of construction times between BFT and SBT can be explained by the fact that SBT construction time does not include the time spent to build the leaves. Also, user time was measured instead of wall-clock time.
While SBT used about 8.15 times more disk than BFT, the BFT used about 7.8 times more RAM than SBT. The disk size of both data structures confirms the result from the *P. aeruginosa* dataset evaluation: The BFT is more adapted for long-term disk storage than SBT as it uses 7 times less disk space than the SBT.

Then, the SBT and BFT were queried with a subset of 1,000 and 10,000 transcript sequences from the GENCODE (Harrow et al., 2012) database, version 25 (GENCODE group, 2013). These transcript sequences are transcripts from the reference chromosomes of the human genome. Because the average size of an SBT vertex in this evaluation is 17.85 MB, we configured the SBT implementation to load the SBT in main memory by batch of 800 vertices to have a similar main memory consumption to the BFT. The results of this querying evaluation are shown in Table 3.5. Results show that for a similar main memory usage, the BFT is 9.51 to 52.66 times faster to query than the SBT and confirm the results obtained from the querying evaluation of the *P. aeruginosa* dataset in Section 3.5.1.

Finally, we extracted all k-mers stored in the BFT and used these k-mers as

Table 3.5: SBT and BFT querying evaluation for the GENCODE sequences database. Best results are highlighted.

		Time		Main memory peak	
		BFT	SBT	BFT	SBT
1,000 queries	$\theta = 0.7$	**39.95 s**	655.07 s	**14.11 GB**	14.51 GB
	$\theta = 0.9$	**39.94 s**	379.89 s	**14.04 GB**	16.67 GB
10,000 queries	$\theta = 0.7$	**109.98 s**	5,792.04 s	**14.11 GB**	15.33 GB
	$\theta = 0.9$	**102.42 s**	2,944.09 s	**14.11 GB**	16.19 GB

queries to compute the number of branching k-mers in the cdBG represented by the BFT. Table 3.6 reports the time and main memory usage of both steps. In this evaluation, 836,937,335 18-mers were extracted and 138,625,655 of them are branching in the cdBG represented with the BFT.

Table 3.6: Evaluation of k-mer extraction and branching queries for a BFT constructed from 2,652 human tissue RNA-Seq experiments.

	Time	Main memory peak
Extracting k-mers	1.58 min	13.92 GB
Branching queries	35.89 min	13.92 GB

3.6 Conclusion

We proposed a novel alignment-free, reference-free, incremental data structure called the Bloom Filter Trie (BFT) to index a pan-genome as a colored de Bruijn graph (cdBG). It accepts assemblies and reads as input to take advantage of all sequencing data representations. The BFT is based on a burst trie and stores k-mers with their colors representing the sequencing experiments in which the k-mers occur. A new representation of vertices is proposed to compress and index shared substrings of k-mers. It uses different types of containers to quickly verify the presence of substrings. The containers use Bloom filters to navigate in the trie and accelerate the cdBG traversal. The BFT was compared to a data structure presenting the same features but based on an approximation of the indexed k-mers. Using two pan-genome datasets, we showed that the BFT has a smaller disk size, can be queried multiple times faster and propose additional

operations such as the extraction of k-mers and the traversal of the cdBG. Future work concerns a more advanced compression scheme of the color sets.

CHAPTER IV

Pan-genome Storage

4.1 Introduction

As the number of sequenced genomes grows exponentially (Land et al., 2015), storing and accessing these data is a problem of main importance. For example, the SRA public database was endangered in 2011 because of budgetary constraints (Genome Biology Editorial Team, 2011). In order to reduce storage and transmission costs, raw sequencing data are often compressed using general purpose compression tools such as gzip based on LZ-77 (Ziv and Lempel, 1977) or bzip based on the BWT (Section 1.2.6). Although these classic tools compressed most of the public data, they are not optimized for HTS compression (Holland and Lynch, 2013; Deorowicz and Grabowski, 2013; Giancarlo et al., 2014; Hosseini et al., 2016; Numanagić et al., 2016). The FASTQ format represents raw sequencing data and each record of this format has three major components: (i) unique identifier, (ii) read sequence and (iii) quality scores. A large variety of HTS-specific compression tools were proposed (Jones et al., 2012; Hach et al., 2012; Bonfield and Mahoney, 2013; Roguski and Deorowicz, 2014; Saha and Rajasekaran, 2014; Rozov et al., 2014; Grabowski et al., 2015; Patro and Kingsford, 2015; Kingsford and Patro, 2015; Benoit et al., 2015) to compress either FASTQ files or only the read sequences. While these tools are very efficient, they are not adapted to the context of large-scale sequencing projects that produce tens to several thousand of genomes per species. A pan-genome is characterized by a high degree of similarity and redundancy between the genomes (Section 1.1.5) but all HTS-specific compression tools can only consider redundancy and similarity within a single genome and not in a collection of genomes. Hence, a tool for pan-

genome read compression must compress reads and the identity of the genomes they belong to while taking advantage of the similarity and redundancy within a pan-genome. Also, large-scale sequencing projects such as the 1000 Genomes Project (1000 Genomes Project Consortium, 2015) may take years to complete, making pan-genomes continually growing. Therefore, a tool for pan-genome read compression must also be incremental such that an archive containing the compressed reads of a pan-genome can be updated with compressed reads of a new genome.

In the following, Section 4.1.1 presents the existing approaches for compressing reads from a single genome and Section 4.1.2 provides an overview of our contributions to pan-genome read compression. Section 4.2 describes an abstract data structure for read indexing inspired from the colored de Bruijn graph (Section 1.2.2.1): the guided de Bruijn graph (gdBG). The methods using the gdBG for compression and the preprocessing operated on the reads to optimize the compression ratio are detailed in Section 4.3. The decompression and update operations are explained in Section 4.4. Finally, Section 4.5 shows the evaluation of our tool and single-genome read compression tools on a pan-genome dataset. Note that in the figures of this chapter, grey colored labels and arrows are for illustration but are not stored in practice.

4.1.1 Existing Approaches

HTS-specific compression tools are divided into two categories: *reference-based* and *de novo*. Reference-based methods generally provide high compression ratio by encoding similarities or differences between the read sequences and a reference usually by mapping the read sequences to the reference. In the following, we use interchangeably the terms “read sequences” and “reads”. Reference-based compression tools require a reference available for the species from which the reads are generated or a similar genome. Note that only a small fraction of sequenced species that are accessible in public databases have such a reference available. Furthermore, the reference used for compression must be provided with the compressed archive for decompression, adding extra storage and transmission costs. On the other hand, *de novo* compression tools perform similarity search within a set of reads in order to exploit its redundancy.

BARCODE (Rozov et al., 2014) is a reference-based method that makes use of BFs (Section 1.2.5) to compress reads. It inserts perfectly matching reads to a reference into a BF that generates false positives. To reduce the number of false positives, BARCODE subsequently inserts them into cascading BFs (Salikhov et al., 2014) to tell apart false positives from true positives. The reference is then used during decompression to query the BFs. Kpath (Kingsford and Patro, 2015) constructs a dBG (Section 1.2.2.1) from the reference and encodes each read as a path within the graph. The paths within the graph are then encoded via arithmetic coding (Witten et al., 1987). The beginnings of such paths are stored separately in a trie and encoded with LZ-77. QUIP (Jones et al., 2012) uses a lossless compression algorithm based on adaptive arithmetic encoding of the identifier, read and quality score streams of the FASTQ format. A reference and a sequence alignment of the reads can be used to improve compression of the reads. QUIP can also perform assembly-based compression in which it builds reference sequences by assembling a small portion (user defined) of the reads using a dBG and then mapping the reads back to the reference sequences. Similar methods are used in FASTQZ and FQZCOMP (Bonfield and Mahoney, 2013). SCALCE (Hach et al., 2012) uses *core substrings* as a measure of similarity in order to cluster similar reads together. Those core substrings are generated via Locally Consistent Parsing (LCP) (Sahinalp and Vishkin, 1996). SCALCE compresses the reads in each cluster with gzip. ORCOM (Grabowski et al., 2015) re-orders reads by similarity as well: it creates clusters of reads that share the same minimizer (Roberts et al., 2004), i.e., the lexicographically smallest p-mer of each read with p usually between 8 and 15. Reads of the same cluster are then merged and compressed. Similar to ORCOM, Mince (Patro and Kingsford, 2015) uses the minimizer approach for clustering. However, it builds the clusters in two steps. A cluster is represented by a k-mer and a set of q-mers from the reads it contains with $q < k$. For each read to process, a set of candidate clusters is first established from the k-mers it is composed of. The read is then assigned to the candidate cluster that maximizes the number of q-mers they share. If the read has no candidate cluster, it is assigned to a new cluster corresponding to its minimizer of length k. FQC (Saha and Rajasekaran, 2014) clusters reads using a k-mer hashing method. Clusters are used to build lists of potentially neighboring reads that share an overlap. From each neighboring list, a consensus sequence is built and used as a reference for compressing the reads of the list. DSRC 2 (Roguski and Deorowicz, 2014) compresses the different streams of FASTQ files with

different methods: arithmetic coding, Huffman coding (Huffman, 1952), as well as 2 bits per base in the case of the DNA sequence stream. Finally, LEON (Benoit et al., 2015) encodes the reads as paths of a dBG represented with a BF. The dBG is built from *solid* k-mers of the reads, i.e., k-mers occurring multiple times in the reads. A read is anchored in the graph if it contains at least one solid k-mer and encoded as a list of graph bifurcations from this anchor.

4.1.2 Contributions

In this chapter, we present a new Dynamic Alignment-free and Reference-free Read Compression method (DARRC). The main contribution of this work is the guided de Bruijn graph (gdBG) inspired from the cdBG (Section 1.2.2.1) which allows a unique traversal to reconstruct the reads it was built from. The gdBG is indexed using the BFT (Chapter III) which enables the update of the gdBG with reads of other similar genomes. Additional methods are presented to optimize the encoding of the reads. On a large *P. aeruginosa* dataset, DARRC outperforms all other tested tools. It provides a 30% compression ratio improvement in single-end mode compared to the best performing state-of-the-art HTS-specific and general purpose compression method in our experiments. The work presented in this chapter was published in (Holley et al., 2017) and is a joint-work with Roland Wittler, Jens Stoye and Faraz Hach.

4.2 The guided de Bruijn Graph

The read assembly problem shows that different traversals of dBGs are possible. In the worst case, the number of possible paths between two vertices in a graph is infinite if the graph is cyclic, and exponential otherwise. Given a dBG built from a sequence and a starting vertex for the traversal, the dBG must be augmented with information to guide its traversal in order to reconstruct the sequence it was built from.

Definition 4. Given a dBG G built from a sequence S, a *partition* $part(G, S)$ is a subgraph G' of G such that G' is a path graph that reconstructs a subsequence of S.

A path graph (V, E) is a non-branching connected graph with $|E| = |V| - 1$. A guided de Bruijn graph (gdBG) is a cdBG $G = (V, E, P)$ built from a sequence S. The set of colors, now denoted as P, represents partitions guiding the traversal of G to reconstruct S. Self-overlapping k-mers, for which the prefix of length $k - 1$ is equal to the suffix of length $k - 1$, require a special treatment to avoid looping on themselves within the same partition. Algorithm 5 creates a gdBG G from a sequence S using vertices of length k. It returns all information necessary to reconstruct S: the gdBG encoding S and the $k - 1$ length prefix of the first k-mer of S starting the graph traversal for decoding. Note that self-overlapping k-mers terminate their partition such that the next inserted k-mers start a new partition (line 9). The algorithm requires $\mathcal{O}(|S|)$ time and $\mathcal{O}(|G|)$ space where $|G| = |V| + |P|$ if the gdBG uses an implicit representation of edges.

Algorithm 5 Encode(S, k)

```
1: p ← 1                                                   ▷ partition index
2: G ← the empty graph
3: for i ← 1, ..., |S| − k + 1 do
4:     x ← S(i, k)
5:     Y ← {y | y successor of x in G with p ∈ G[y]}
6:     if Y ≠ ∅ then p ← p + 1
7:     if x ∈ G then G[x].add(p)                  ▷ add p to vertex x in G
8:     else G.add(x, p)                     ▷ insert vertex x with p in G
9:     if x(2, k − 1) = x(1, k − 1) then p ← p + 1
10: return (G, S(1, k − 1))
```

Algorithm 6 decodes a sequence S from a gdBG G using vertices of length k starting with k-mer prefix x. Algorithm 5 guarantees that for any k-mer and one of its partitions, this k-mer can only have zero or one successor in the graph with the same partition. Therefore, Algorithm 6 traverses the graph by searching, for each traversed vertex, the successor with the same partition. If it is not found, the partition index is incremented and the traversal continues. As for Algorithm 5, the algorithm requires $\mathcal{O}(|S|)$ time and $\mathcal{O}(|G|)$ space.

Figure 4.1 represents a simple cyclic dBG built from a sequence containing a repetition. An infinite number of sequences could be extracted from the dBG because of the cycle. However, by augmenting the dBG with partitions, Algorithm 6 will traverse the cycle only once during the reconstruction of the sequence. Indeed, when Algorithm 5 tries to insert k-mer "agt" with partition 1, a successor with the same partition is found. Therefore, k-mer "agt" is inserted

Algorithm 6 Decode(G, x, k)

1: $p \leftarrow 1$
2: $z \leftarrow k$-mer y in G with $y(1, k-1) = x$ and $p \in G[y]$
3: $x \leftarrow z$
4: $S \leftarrow z$
5: $Z \leftarrow \{z\}$
6: **if** $z(2, k-1) = z(1, k-1)$ **then** $p \leftarrow p + 1$
7: **while** $Z \neq \varnothing$ **and** $p \in P$ **do**
8: $\quad Z \leftarrow \{z \mid z$ successor of x in G with $p \in G[z]\}$
9: $\quad$ **if** Z contains exactly one k-mer z **then**
10: $\quad\quad S \leftarrow S \odot z(k, 1)$
11: $\quad\quad x \leftarrow z$
12: $\quad\quad$ **if** $x(2, k-1) = x(1, k-1)$ **then** $p \leftarrow p + 1$
13: $\quad$ **else**
14: $\quad\quad p \leftarrow p + 1$
15: $\quad\quad Z \leftarrow \{x\}$
16: **return** S

with partition 2 such that the cycle is not contained in one partition.

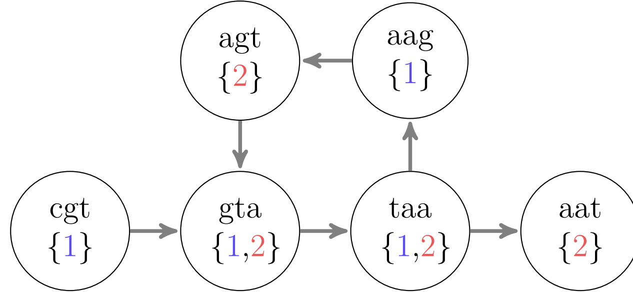


Figure 4.1: The gdBG of sequence S = “cgtaagtaat” as constructed by Algorithm 5 with $k = 3$.

An important property of gdBGs using implicit edges is that no false implicit edge can be traversed during the decoding.

Proposition 1. Let G be a gdBG built from a sequence S using an implicit representation of edges. An edge between vertices v and v' corresponding to k-mers x and x' respectively, such that $x(2, k-1) = x'(1, k-1)$ but $x \odot x'(k, 1)$ is not a substring of S is called a false implicit edge. Algorithm 6 does not consider any false implicit edge when traversing G to reconstruct S.

Proof. If a false implicit edge connects vertices not sharing a partition, Algorithm 6 will not consider this edge as only successors with the same partition

are traversed. If a false implicit edge connects vertices v and v' which share a partition, the edge out-degree of v is at least 2 and the edge in-degree of v' is at least 2: one true implicit edge each and at least one false implicit edge each. As these vertices are branching, Definition 4 guarantees that v and v' are not in the same partition. □

Algorithm 5 does not distinguish true implicit edges from false implicit edges, ensuring that Definition 4 is always respected during the encoding.

Furthermore, partitions allow to apply the following generalized definition of edges in dBGs to gdBGs:

Definition 5. In a dBG, a directed edge from vertex v to vertex v' representing k-mers x and x', respectively, exists if and only if $x(l+1, k-l) = x'(1, k-l)$ with $l \geq 1$.

For a sequence S to encode in a gdBG and $l > 1$, $\left\lfloor \frac{|S|-k+1}{l} \right\rfloor$ k-mers will be inserted instead of $|S|-k+1$. However, the graph can contain more partitions as each vertex has now $|\mathcal{A}|^l$ possible successors and predecessors. Figure 4.2 gives the gdBG encoding the same sequence as in Figure 4.1 using a k-mer overlap of $k-2$ instead of $k-1$. The resulting gdBG contains only half the number of vertices than the one in Figure 4.1.

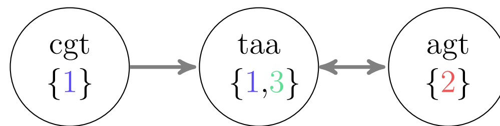


Figure 4.2: The gdBG of sequence S = cgtaagtaat using 3-mers overlapping on $k-l=1$. The last symbol of S is not encoded in the gdBG as it cannot be part of a k-mer.

4.3 Compression

Section 4.2 presented methods to encode a sequence as a gdBG and to decode it. In this section, we describe how to use this methodology to compress reads. To improve compression efficiency, we preprocess the reads.

4.3.1 Read Clustering and Merging.

A simple form of read assembly extended from ORCOM (Grabowski et al., 2015) is performed to reduce the input data. It clusters reads according to their minimizer, then merges reads sharing an overlap within each cluster and finally merges reads sharing an overlap but originating from different clusters. These three steps are described in the following.

4.3.1.1 Clustering

The minimizer (Roberts et al., 2004) of a read r is the lexicographically smallest of its p-mers with $p \ll |r|$. The canonical minimizer of r is the lexicographically smaller minimizer of r and its reverse-complement $\overline{r}$. The following method is based on the simple assumption that reads sharing a minimizer are likely to share a longer overlap and therefore be similar. Thus, the canonical minimizer m is computed for each read r such that r or $\overline{r}$ is assigned to its cluster m. An example of clustering is illustrated in Figure 4.3.

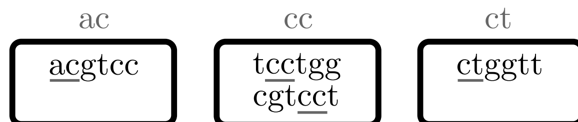


Figure 4.3: Minimizer clustering of 4 reads. Minimizers of length 2 are underlined. For the sake of convenience, the reverse-complement is not considered.

4.3.1.2 Intra-cluster Merging

Within each cluster, the reads are sorted by decreasing position of their minimizer, in which reads sharing the same minimizer position are sorted lexicographically. For each read r and its minimizer m at position p_m, all reads r' with minimizers at positions $p'_m \leq p_m$ are considered for merging, in decreasing order of positions p'_m to maximize the overlap lengths. To merge reads r and r', they are first anchored at the position of their minimizers such that they overlap on $o = p'_m + \mathsf{Min}(|r| - p_m, |r'| - p'_m)$ symbols. Reads are merged into a *super read* (Zimin et al., 2013) if $r(p_m - p'_m, o) = r'(1, o)$ with at most d mismatches. The same process is applied to the created super read in order to merge it with

the remaining reads of the cluster. For each super read, we encode all of its read meta data in separate streams: position, length, reverse-complement information and mismatches. An example of intra-cluster merging (cf. Figure 4.3) is shown in Figure 4.4.

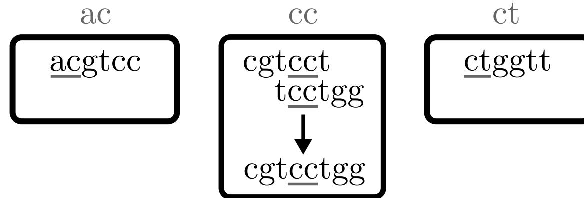


Figure 4.4: Merging of two reads into a super read. Minimizers of length 2 are underlined. For the sake of convenience, the reverse-complement is not considered.

4.3.1.3 Inter-cluster merging

As an extension of the previous steps used by ORCOM, we additionally perform a process similar to the intra-cluster read merging described previously to merge super reads from multiple clusters. For each super read sr and its minimizer m at position p_m, a new minimizer m' is computed in $sr(p_m + 1, |sr| - p_m)$ and $\overline{sr}$. All super reads of cluster m' are considered for a merging with sr or $\overline{sr}$. Merging two or more super reads creates a *Spanning Super Read* (SSR). The same process is applied to the created SSR until no super reads can merge with it. An example of inter-cluster merging (cf. Figure 4.4) is provided in Figure 4.5.

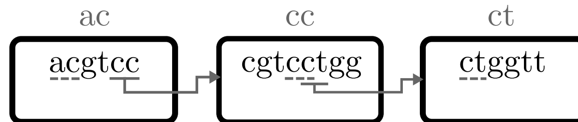


Figure 4.5: Merging of three super reads into an SSR "acgttgatt". Minimizers of length 2 are underlined with a dashed line. Secondary minimizers (for merging) are underlined with a plain line. For the sake of convenience, the reverse-complement is not considered.

4.3.1.4 Paired-end Reads

Each mate of a pair is considered as a single read that is clustered and merged using the previously described methods. However, the clustering and merging

steps keep track of the position of the mates in the SSRs. This information is used afterwards to store in each read meta data whether the read is the first mate of its pair. In such case, the position of its corresponding mate in the SSRs is stored as well.

4.3.2 Spanning Super Read Encoding

Encoding a set of SSRs using a gdBG requires to extract k-mers from the SSRs. If edges represent overlaps of length $k-1$, all k-mers of the SSRs are extracted. If edges represent overlaps of length $k-l$ with $l > 1$, k-mers are extracted every l positions. As a consequence, similar SSRs can have different sets of k-mers. An example is given in Figure 4.6, in which two similar SSRs, ssr_1 and ssr_2, do not share any k-mers because they are extracted every $l = 2$ positions from the first position of each SSR. By shifting the k-mer extraction start position by one position in the second SSR, as shown with ssr_2', two extracted k-mers are shared with the first SSR.

$ssr_1 =$ a c g t c c t g a a t $\quad$ $ssr_2 =$ g a c g t c c g g a a $\quad$ $ssr_2' =$ g a c g t c c g g a a

ssr_1	ssr_2	ssr_2'
a c g t	g a c g	**a c g t**
g t c c	c g t c	**g t c c**
c c t g	t c c g	c c g g
t g a a	c g g a	g g a a

Figure 4.6: Extraction of 4-mers overlapping on $k-l = 2$ from two similar SSRs, ssr_1 and ssr_2.

In order to keep the growth of the gdBG small when inserting a new SSR, we determine the k-mer extraction start position, called *start position* in the following, that maximizes the number of k-mers already stored in the gdBG. To this end, we maintain in memory a k-mer index recording all k-mers extracted. As the cost in time and memory of such an index is prohibitive, we use a BF having constant time insertion and look-up instead of a BFT having linear time insertion and look-up. Algorithm 7 is a greedy approach making use of the BF to iteratively detect for each SSR of a set R its optimal start position and updating the BF with all novel k-mers. To encode all SSRs completely, it not only returns the k-mers to insert into a gdBG, because these do not necessarily cover the entire SSRs. It also returns the *head* and *tail* of each SSR, which are the

uncovered prefix and suffix, respectively, not encoded in the gdBG. Additionally, to provide an entry point into the gdBG for the decoding, it returns the *starting overlap* of each SSR, which is the $k-l$ length prefix of the first k-mer. More precisely, we denote by x and y the first and last k-mers extracted, respectively, from an SSR ssr with pos_x and pos_y as their respective occurrence positions in ssr. Then, the head of ssr is the prefix $ssr(1, pos_x - 1)$, the tail of ssr is the suffix $ssr(pos_y + k, |ssr| - pos_y - k + 1)$, and the starting overlap of ssr is $ssr(pos_x, k - l)$. SSR heads, tails and starting overlaps are encoded in separate streams and compressed separately from the gdBG.

4.3.3 Partition Encoding

4.3.3.1 Encoding

Partition sets associated with k-mers in gdBGs are represented as lists of sorted integers. A naive way to store a partition set is to use a fixed number of bytes for each partition. For example, 4 bytes is a standard size for integers on current computer architectures. In order to decrease the memory footprint while keeping the lists indexed, partitions are first delta encoded by storing the difference between each integer and its predecessor in the list (or 0 if the integer is in first position). The resulting values are called *deltas*. However, it only decreases the minimum number of bits necessary to encode the partitions but not their final representation. Consequently, deltas are Vbyte encoded (Williams and Zobel, 1999): each byte used to encode a delta has one bit indicating whether the byte starts a new delta or not, allowing to remove unnecessary bytes from each delta. Thus, partitions use a variable number of bytes proportional to the minimum number of bits necessary to encode their deltas. An example of a partition set naively represented is described in Table 4.1. The table summarizes the minimum number of bits (log) necessary to encode the partitions, their binary code and the number of bytes they use. Table 4.2 describes the same set represented with a delta and Vbyte encoding: it uses 5 bytes instead of 12 in the naive representation.

Algorithm 7 ExtractKmers(k, l, R)

```
1:  function ExtractPosition(r, k, l, B)
2:      best_count ← 0
3:      best_pos ← 1
4:      for i ← 1, ..., l do
5:          pos ← i
6:          count ← 0
7:          while pos ≤ |r| − k + 1 do
8:              if MayContain(r(pos, k), B) then count = count + 1
9:              pos = pos + l
10:         if count > best_count then
11:             best_count ← count
12:             best_pos ← i
13:     return (best_pos, best_count)
14:
15: function ExtractKmers(k, l, R)
16:     K ← ∅                                   ▷ ordered set of extracted k-mers
17:     L ← ∅                                   ▷ ordered set of SSR heads and tails
18:     O ← ∅                                   ▷ ordered set of SSR starting overlaps
19:     B ← empty Bloom filter
20:     for each ssr ∈ R do
21:         a ← ExtractPosition(ssr, k, l, B)
22:         b ← ExtractPosition(\overline{ssr}, k, l, B)
23:         if a.best_count > b.best_count then
24:             ssr′ ← ssr
25:             pos ← a.best_pos
26:         else
27:             ssr′ ← \overline{ssr}
28:             pos ← b.best_pos
29:         L ← L ∪ {ssr′(1, pos − 1)}
30:         O ← O ∪ {ssr′(pos, k − l)}
31:         while pos ≤ |ssr′| − k + 1 do
32:             K ← K ∪ {ssr′(pos, k)}
33:             if MayContain(ssr′(pos, k)), B) = false then
34:                 Insert(ssr′(pos, k)), B)
35:             pos ← pos + l
36:         L ← L ∪ {ssr′(pos + k, |ssr′| − pos − k + 1)}
37:     return (K, L, O)
```

4.3.3.2 Recycling

As a small delta produces a small encoding, partition integers are recycled instead of naively using the next higher integer for every new partition as, for the

Table 4.1: Naive representation of a partition set composed of integers 12534, 12567 and 28911.

Partitions	log	Binary code	Number of bytes
12534	13.61	00000000 00000000 00110000 11110110	4
12567	13.62	00000000 00000000 00110001 00010111	4
28911	14.82	00000000 00000000 01110000 11101111	4

Table 4.2: Delta and Vbyte encoded representation of the same partition set used in Table 4.1.

Partitions	log	Binary code	Number of bytes
$\Delta_1 = 12534$	13.61	0110000**0** 1110110**1**	2
$\Delta_2 = 12567 - \Delta_1 = 33$	5.04	0100001**1**	1
$\Delta_3 = 28911 - \Delta_1 - \Delta_2 = 16344$	14.00	1111111**0** 1011000**1**	2

sake of convenience, described in Algorithm 5. Partition sets a and b can share the same partition integer if they are not neighbors in the graph, i.e., no k-mer suffix or prefix of a overlaps a k-mer prefix or suffix of b, for suffixes and prefixes of length $k - l$. A trivial example is provided in Figure 4.7 in which k-mer *cttc* uses the same partition integer as k-mer *acgt* because they are not neighbors in the graph.

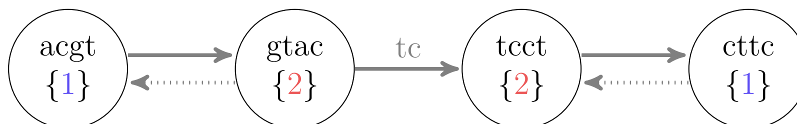


Figure 4.7: The gdBG of SSRs ssr_1 = “acgtac” and ssr_2 = “tccttc” using 4-mers ($l = 2$). Dotted edges are false implicit edges. The labeled solid edge exists by using the starting overlap of ssr_2 after the traversal of ssr_1, as described in Section 4.3.2.

As there can be a large number of partitions in the graph, verifying the connectivity of one partition to all other partitions is often impractical. We propose instead a heuristic that verifies the connectivity only to the last t partitions inserted, t being a user-defined threshold, such that these t partitions are the only candidates for recycling. Using partition recycling requires to save the partitions traversal order which cannot be incremental anymore as proven in Algorithms 5 and 6.

4.3.4 Meta Data and gdBG Compression

Steps described previously generate meta data specific to one input file such as read lengths and positions in SSRs. These meta data are first encoded in separate streams and are then compressed using an LZ-type algorithm, LZMA (Pavlov, 1999). After all k-mers and partitions are inserted in the gdBG, the latter is written to disk. As it must be loaded in memory for every update and decompression, the gdBG is compressed with Zstd (Collet, 2015), a compression method based on Huffman coding and Asymmetric Numeral Systems (Duda, 2013) that favors compression and decompression speed over compression ratio.

4.4 Update and Decompression

In order to update a compressed archive with a new input file, only the gdBG previously created is decompressed and loaded in memory, as meta data are not used for the update. A fast procedure iterates over all k-mers of the gdBG and inserts them into the BF of Algorithm 7 instead of starting with an empty BF in order to optimize the choice of the k-mer extraction start positions in the SSRs. The gdBG is then updated with the new k-mers and partitions. The starting partition index is greater than the partition indexes already present in the gdBG, ensuring that each input file is encoded with a unique set of partitions.

Decompressing a read file starts with decompressing its meta data and the gdBG it is encoded in. The gdBG is then loaded in memory and Algorithm 6 is used to traverse the gdBG, but only following those partitions that are specific to the read file to decompress. This way, single files can be decompressed separately. As Algorithm 6 decodes SSRs, meta data are used afterwards to extract the actual reads. If reads are paired-end, meta data are also used to reorganize them such that corresponding mates of the same pair are together in the decompressed file.

4.5 Results

DARRC is implemented in C and uses the BFT library for its gdBG. The BFT provides time and space efficient functionalities that are required by DARRC.

These functionalities include: (i) the ability to update the BFT with new k-mers and colors without recomputing the index, (ii) k-mers extraction from the BFT and (iii) prefix search over the set of k-mers within the BFT. The software is available at `https://github.com/GuillaumeHolley/DARRC`. We compared DARRC to three state-of-the-art *de novo* DNA sequence compression tools: ORCOM (Grabowski et al., 2015), LEON (Benoit et al., 2015) and Mince (Patro and Kingsford, 2015). DARRC was also compared to the same LZ-type algorithm used to compress its meta data, LZMA (Pavlov, 1999). Experiments were carried out on a server with 378 GB of RAM and two 8-core Intel Xeon E5-2630 v3 2.4 GHz processors. All input files were placed on a dedicated mechanical hard drive. Compressed archives and decompressed files, during compression and decompression, respectively, together with temporary files such as read clusters were written to a RAM-based partition when the tools allowed to specify an output directory. As the current version of DARRC does not take advantage of parallelism, all software were run using a single thread, except Mince which requires a minimum of four threads. All *de novo* DNA sequence compression tools were run using their default parameters. LZMA was run with the same compression level as the one used to compress DARRC meta data. DARRC default parameters are minimizers of length 9 for the clustering, 5 mismatches allowed per read merging and 36-mers overlapping on 11 symbols for the gdBG. ORCOM, LEON, Mince and LZMA compressed all files in separate archives while DARRC updated the same archive iteratively with the files to compress: each iteration decompressed and reloaded the necessary data from the data written to disk in the previous iteration. The dataset used for the experiment consists of 473 clinical isolates of *P. aeruginosa* sampled from 34 patients (NCBI BioProject PRJEB5438), resulting in 338.61 Gbp of high coverage sequences. Reads are 100 bp paired-end reads generated by Illumina HiSeq 2000. Pair mates were placed in different files for every isolate. The experiment was run in single-end mode and paired-end mode for all tools such that in the single-end mode, every mate file is considered as a single-end read file. The appropriate single-end and paired-end modes were used for DARRC and Mince. The mates were concatenated for the paired-end experiment of ORCOM as the tool neither preserves the order of the reads nor stores the paired-end information. LEON and LZMA do not have an explicit paired-end mode but keep the original order of the reads, thus the mate files of every isolate were concatenated for the paired-end experiments of LEON and LZMA.

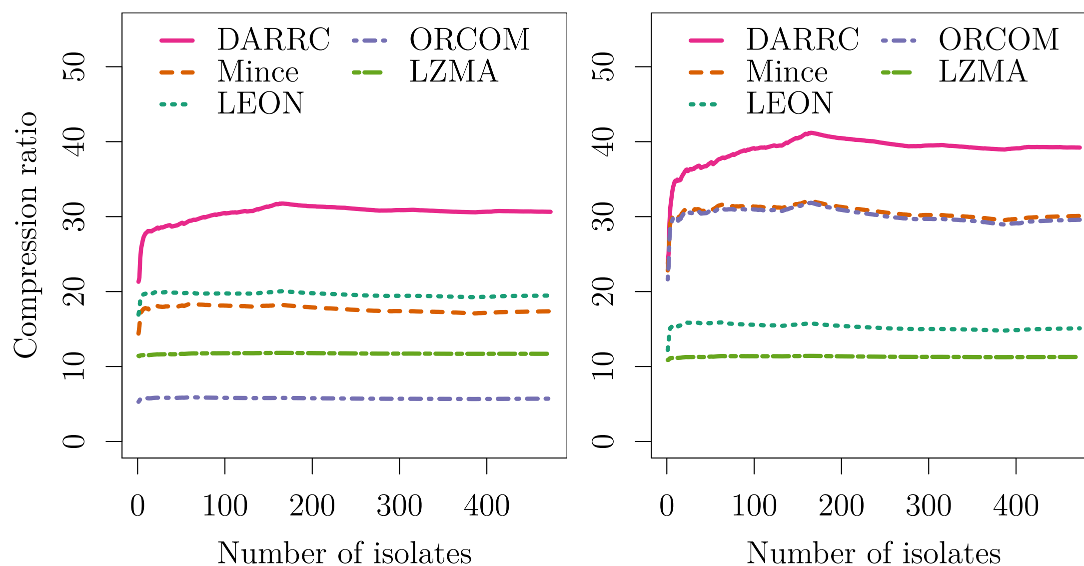


Figure 4.8: Compression ratios in paired-end mode (left) and single-end mode (right).

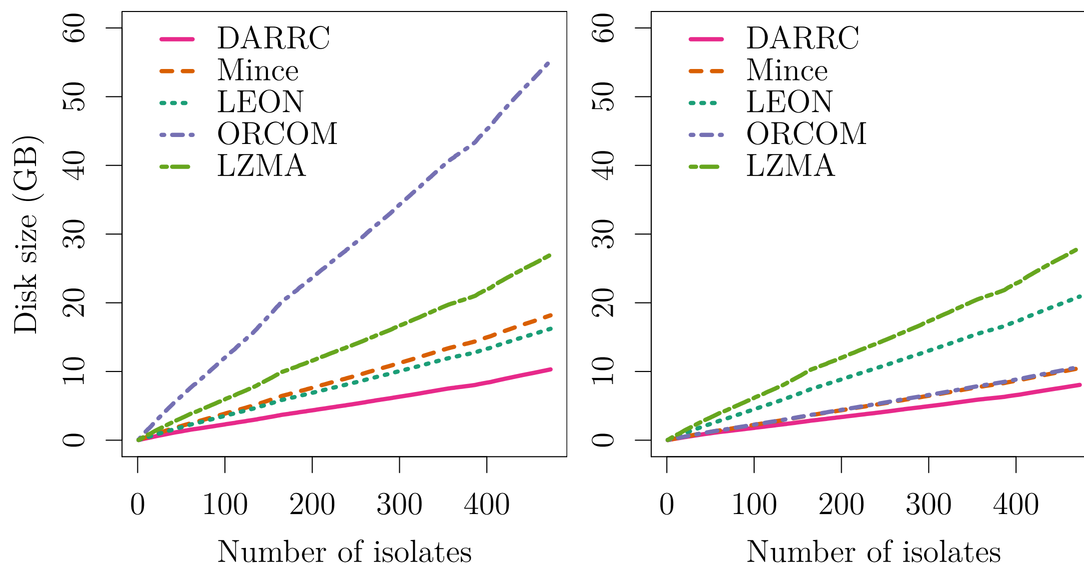


Figure 4.9: Disk sizes in paired-end mode (left) and single-end mode (right).

Compression ratios in paired-end mode and single-end mode are shown in Figure 4.8. DARRC clearly outperforms all the other tested tools in both modes. In paired-end mode and single-end mode, DARRC uses about 0.261 bits per base and 0.204 bits per base, corresponding to a 57 % and 30 % compression ratio

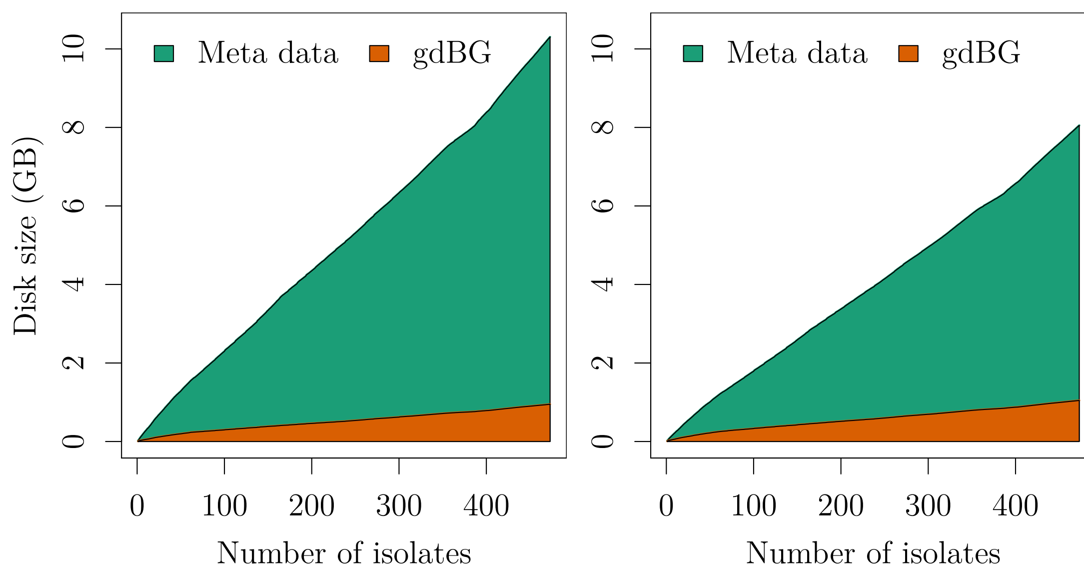


Figure 4.10: DARRC disk size distribution in paired-end mode (left) and single-end mode (right).

improvement compared to the second best results, respectively. The paired-end compression ratio of ORCOM compared to its single-end compression ratio shows that the tool is not adapted to paired-end read compression. Corresponding disk sizes are shown in Figure 4.9. The distributions of DARRC meta data and gdBG disk sizes are displayed in Figure 4.10. The gdBG represents about 10 % and 13 % of the data written to disk in paired-end mode and single-end mode, respectively.

DARRC compressed more than two times faster than LZMA but used the most time to decompress, as shown in Figures 4.11 and 4.12, respectively. The compression time overhead of DARRC is explained by the fact that at each iteration, the gdBG must be decompressed, loaded in memory and updated with new k-mers and partitions.
Main memory usage during compression and decompression is shown in Figures 4.13 and 4.14, respectively. All tools performed compression and decompression using a maximum of 3.4 GB of main memory, an amount nowadays available on most desktop computers and laptops. Even by updating the same archive iteratively, DARRC compression used an amount of main memory similar to the memory footprint of the other tested tools.

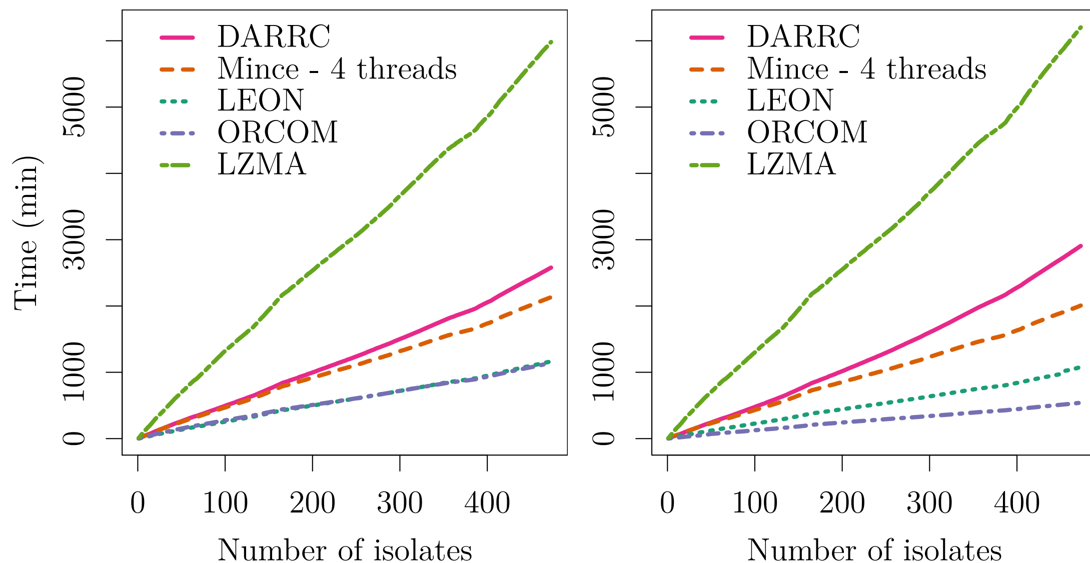


Figure 4.11: Compression times in paired-end mode (left) and single-end mode (right).

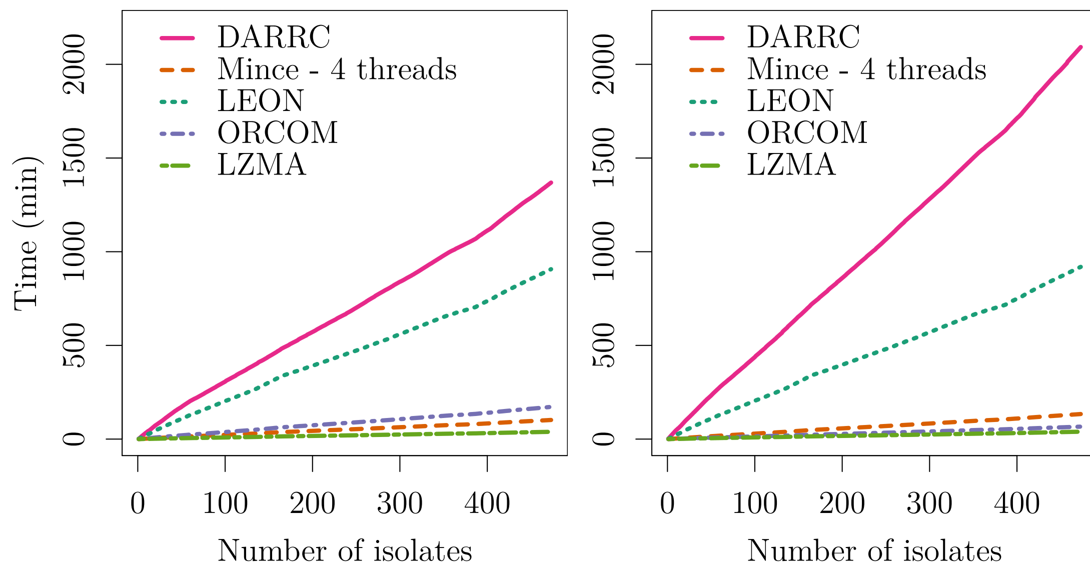


Figure 4.12: Decompression times in paired-end mode (left) and single-end mode (right).

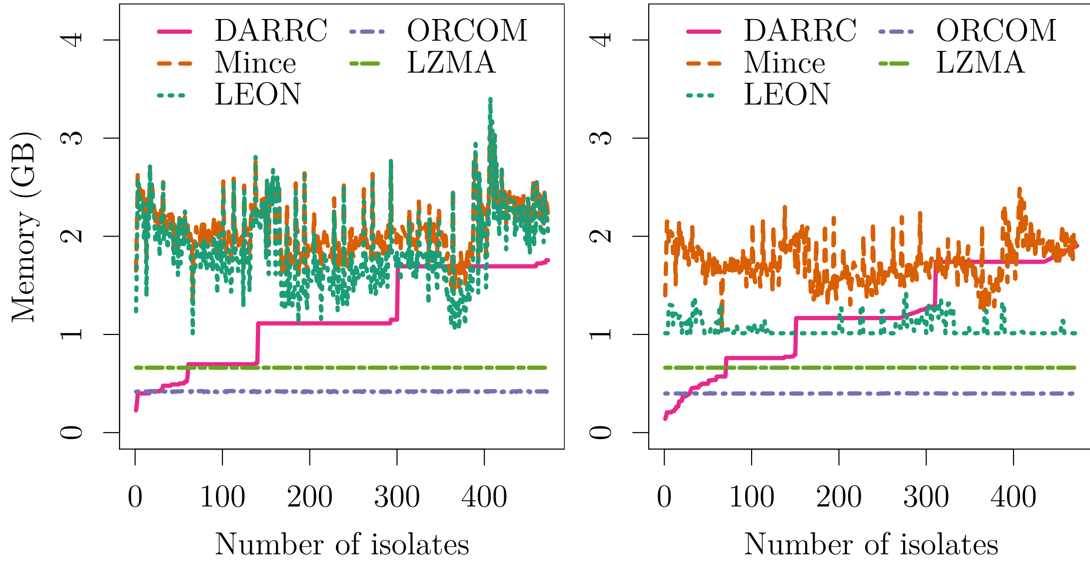


Figure 4.13: Compression main memory peaks in paired-end mode (left) and single-end mode (right).

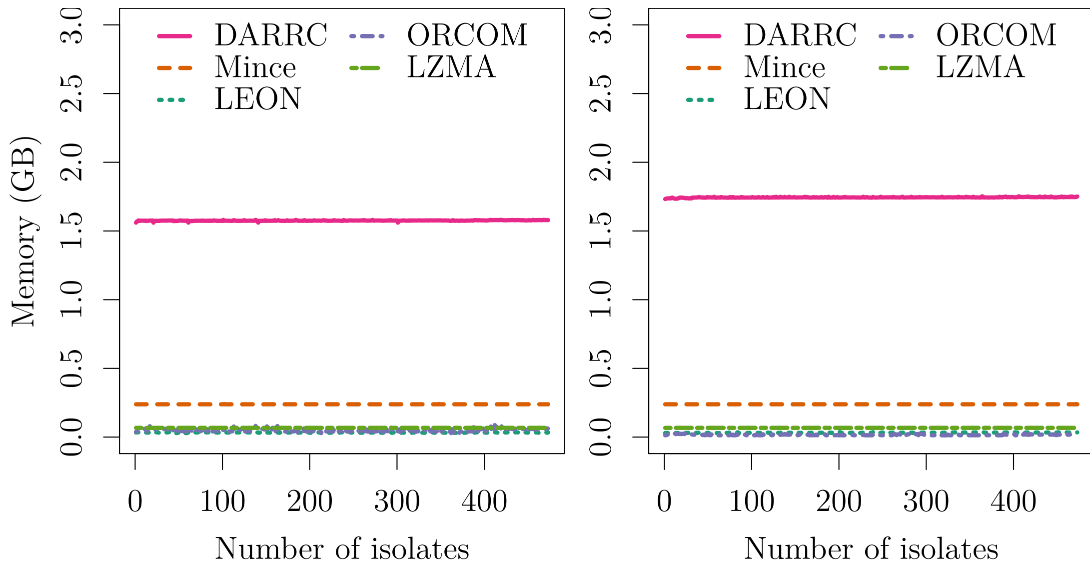


Figure 4.14: Decompression main memory peaks in paired-end mode (left) and single-end mode (right).

4.6 Conclusion

We presented DARRC, a dynamic alignment-free and reference-free read compression method that can incrementally update compressed archives with new genome sequences without full decompression of the archives. DARRC uses a new abstract data structure, the guided de Bruijn graph, that allows a unique traversal of the de Bruijn graph to reconstruct the sequences it is built from. We showed that, on a large pan-genome dataset, our method outperforms several state-of-the-art DNA sequence compression methods and a general purpose compression tool regarding the compression ratio while achieving reasonable running time and main memory usage. Furthermore, we showed that the compression ratio of DARRC is attractive even with only few files compressed. Future work concerns the parallelization of the software, particularly the read clustering and merging phase which offers a lot of potential for multi-threading.

CHAPTER V

Conclusion

In this thesis, we explored new solutions for the problems of pan-genome indexing and storage. After introducing biological notions and providing a retrospective on the acquisition of genomic data, we established the current limitations of single-genome reference-centric methods in comparative analysis. These limitations can be overcome by using a pan-genome — a set of genomes from different strains of the same species — as a reference instead of a single genome that cannot represent an entire population of genomes. Then, we provided a description of pan-genomes and their characteristics. Next, we reviewed the literature of data structures and methods to index and analyze pan-genomes at the DNA sequence level in Chapter II.

In Chapter III, we showed the current limitations and inadequacies of data structures for pan-genome indexing mentioned in the literature. We established that the colored de Bruijn graph is an ideal representation of a pan-genome because it is alignment-free and reference-free. However, current data structures to represent colored de Bruijn graphs are neither incremental nor time and memory efficient. Therefore, we proposed to represent a pan-genome as a colored de Bruijn graph with a new lightweight data structure, the Bloom Filter Trie, which is alignment-free, reference-free, incremental and considers assemblies as well as reads as input. The Bloom Filter Trie is based on a burst trie to efficiently represent k-mer suffixes in the deeper levels of the trie. The usage of Bloom filters allows to efficiently navigate in the trie while optimizing the traversal of the colored de Bruijn graph. We evaluated the Bloom Filter Trie in comparison to a data structure presenting similar features on two pan-genome datasets, a bacterial dataset and a human tissue dataset. We showed that while the two data structures have different construction advantages, the Bloom Filter Trie has the

smallest disk size, is faster to query and propose supplementary functionalities over the other data structure.
In Chapter IV, we discussed that pan-genomes are challenging to store because of the increasing amount of data produced by large scale sequencing projects. Compression is a solution to this problem, but no compression tool is currently taking advantage of the genome similarity and redundancy within a pan-genome. Furthermore, current compression tools are not dynamic and produce compressed archives that cannot be updated with new data. For this purpose, we proposed a new abstract data structure — the guided de Bruijn graph — which augments k-mers stored in the graph with a partition information. The guided de Bruijn graph is used to insert k-mers of the reads to compress and the partitions are used during decompression to guide the traversal of the graph to exactly reconstruct the compressed reads. We also described different techniques to optimize the storage of reads in the graph and the partition encodings. As the guided de Bruijn graph is inspired from the colored de Bruijn graph, we used the Bloom Filter Trie as the main data structure to implement this compression tool named DARRC. We evaluated DARRC in comparison to other state-of-the-art HTS-specific and general purpose compression tools on a bacterial dataset. We show that DARRC outperforms the other tools with a 30 % compression ratio improvement over the second best performing tool of the evaluation.

5.1 Perspectives

A first interesting perspective is that large scale sequencing projects are flourishing because they benefit from the diversity of sequencing technologies and the decrease of sequencing cost. Hence, the size of pan-genomes, currently up to a few thousand sequenced genomes, is going to increase at a very fast pace within the next few years. Indeed, Stephens et al. (2015) estimate that the human pan-genome will be composed of 100 million to 2 billion sequenced genomes by 2025. For example, the 100,000 Genomes Project (100,000 Genomes Project, 2012) undertaken in 2012 is expected to have sequenced 100,000 human genomes by 2017, while a US-based project (Ledford, 2016) aims to sequence two million individuals within ten years. Hence, data structures for pan-genome indexing should be ready to handle this massive amount of data.
Besides indexing, the analysis of a pan-genome based on efficient data struc-

tures as presented in this thesis is an important next step. A first application of such a pan-genome index is conducted by Tina Zekic at the Bielefeld University and consists in the extraction of core, accessory and singleton genomes directly from a colored de Bruijn graph represented with a Bloom Filter Trie. This work can be used in the context of ancestral genome reconstruction (Luhmann et al., 2016) to sheld light on ancient species from which damaged DNA has been recovered. Indeed, the assembly of these data is usually difficult because of the short length and degraded quality of the reads. Such reads from ancient species can be included in the pan-genome of extant strains to assist comparative genomics methods aiming to extract conserved regions between ancient and extant strains, as defined by the core genome. These conserved regions then allow to analyze genome rearrangements in a phylogenetic context.
A third interesting perspective is the emergence of third generation sequencing technologies. The Oxford Nanopore (Mikheyev and Tin, 2014) is a third generation sequencer that has the size of a large USB stick and costs around 1,000 $. Its small size and cost indicate an upcoming fast deployment of this sequencing technology all around the world. Third generation sequencing technologies produce long reads that can disambiguate repeats in genome assembly and variant calling. However, such reads usually have a high error rate that can be corrected with the help of short and more accurate reads produced by HTS technologies. As these technologies are complementing each other, pan-genome indexes should neither be limited to one type of representation — assemblies or reads — nor should they be limited by the sequencing technology used to produce reads. To the contrary, the future of pan-genome indexing resides in the unification of all types of sequencing data into a single index such that comparative methods can take advantage of all features proposed by each sequencing technology. An example of such unification is the preliminary work of Ehsan Haghshenas from Simon Fraser University, who is performing hybrid single variant detection using long and short reads. The core idea of this work is to map long reads to a colored de Bruijn graph of short reads represented with a Bloom Filter Trie. The colors can be used to distinguish short reads from different strains to enhance the chances of mapping a long read to the graph. An orthogonal concept is to include long and short reads in the same colored de Bruijn graph represented with a Bloom Filter Trie such that long reads can assist the assembly of repeated regions. While such a representation is made possible by using the guided de Bruijn graph presented in Chapter IV, the high k-mer diversity of long noisy reads makes it challenging

to index them efficiently.

Finally, the work presented in this thesis shows that the Bloom Filter Trie is adapted to pan-genome indexing and compression. The next logical step is to combine both approaches into a single one enabling sequence queries directly on a compressed archive of DARRC without having to decompress the sequences beforehand and index them with a BFT afterwards. Besides the considerable amount of compute time and memory saved, it would allow to perform large scale complex methods such as read alignment and variant calling using multiple genomes at once.

BIBLIOGRAPHY

BIBLIOGRAPHY

1000 Genomes Project Consortium (2015). A global reference for human genetic variation. *Nature*, 526(7571):68–74.

100,000 Genomes Project (2012). Genomics England. `https://www.genomicsengland.co.uk/`. [Online; accessed 16-February-2017].

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algor.*, 2(1):53–86.

Almeida, P. S., Baquero, C., Preguiça, N., and Hutchison, D. (2007). Scalable bloom filters. *Inform. Process. Lett.*, 101(6):255–261.

Andersson, A. and Nilsson, S. (1993). Improved behaviour of tries by adaptive branching. *Inform. Process. Lett.*, 46(6):295–300.

Askitis, N. and Sinha, R. (2010). Engineering scalable, cache and space efficient tries for strings. *The VLDB Journal*, 19(5):633–660.

Baier, U., Beller, T., and Ohlebusch, E. (2016). Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. *Bioinformatics*, 32(4):497–504.

Belk, K., Boucher, C., Bowe, A., Gagie, T., Morley, P., Muggli, M. D., Noyes, N. R., Puglisi, S. J., and Raymond, R. (2016). Succinct Colored de Bruijn Graphs. *bioRxiv*, page 040071.

Bender, M. A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B. C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R. P., and Zadok, E. (2012). Don't thrash: how to cache your hash on flash. *Proc. of the VLDB Endowment*, 5(11):1627–1637.

Benoit, G., Lemaitre, C., Lavenier, D., Drezen, E., Dayris, T., Uricaru, R., and Rizk, G. (2015). Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC Bioinform.*, 16(1):288.

Blom, J., Albaum, S. P., Doppmeier, D., Pühler, A., Vorhölter, F.-J., Zakrzewski, M., and Goesmann, A. (2009). EDGAR: A software framework for the comparative analysis of prokaryotic genomes. *BMC Bioinform.*, 10(1):154.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Comm. ACM*, 13(7):422–426.

Bonfield, J. K. and Mahoney, M. V. (2013). Compression of FASTQ and SAM format sequencing data. *PLoS One*, 8(3):e59190.

Bowe, A., Onodera, T., Sadakane, K., and Shibuya, T. (2012). Succinct de Bruijn graphs. In *Proc. of 12th International Workshop on Algorithms in Bioinformatics (WABI'12)*, volume 7534, pages 225–235.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. *Digital SRC Research Report 124*.

Chikhi, R., Limasset, A., Jackman, S., Simpson, J. T., and Medvedev, P. (2015). On the representation of de Bruijn graphs. *J. Comp. Biol.*, 22(5):336–352.

Chikhi, R., Limasset, A., and Medvedev, P. (2016). Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208.

Claude, F., Farina, A., Martínez-Prieto, M. A., and Navarro, G. (2010). Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. of the IEEE International Conference on BioInformatics and BioEngineering (BIBE'10)*.

Collet, Y. (2015). ZSTD compression library. `https://github.com/facebook/zstd`. [Online; accessed 20-December-2016].

Compeau, P. E., Pevzner, P. A., and Tesler, G. (2011). How to apply de Bruijn graphs to genome assembly. *Nat. Biotechnol.*, 29(11):987–991.

Computational Pan-Genomics Consortium (2016). Computational pan-genomics: status, promises and challenges. *Brief. Bioinform.*, page bbw089.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT Press.

Danek, A., Deorowicz, S., and Grabowski, S. (2014). Indexes of large genome collections on a PC. *PloS One*, 9(10):e109384.

De La Briandais, R. (1959). File searching using variable length keys. In *Proc. of the Western Joint Computer Conference (IRE-AIEE-ACM'59)*, pages 295–298.

Denton, J. F., Lugo-Martinez, J., Tucker, A. E., Schrider, D. R., Warren, W. C., and Hahn, M. W. (2014). Extensive error in the number of genes inferred from draft genome assemblies. *PLoS Comput. Biol.*, 10(12):e1003998.

Deorowicz, S. and Grabowski, S. (2013). Data compression for sequencing data. *Algorithms Mol. Biol.*, 8:25.

Dolle, D.-D., Liu, Z., Cotten, M. L., Simpson, J. T., Iqbal, Z., Durbin, R., McCarthy, S. A., and Keane, T. M. (2017). Using reference-free compressed data structures to analyse sequencing reads from thousands of human genomes. *Genome Res.*, 27(2):300–309.

Donati, C., Hiller, N. L., Tettelin, H., Muzzi, A., Croucher, N. J., Angiuoli, S. V., Oggioni, M., Dunning Hotopp, J. C., Hu, F. Z., Riley, D. R., et al. (2010). Structure and dynamics of the pan-genome of Streptococcus pneumoniae and closely related species. *Genome Biol.*, 11(10):R107.

Duda, J. (2013). Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv:1311.2540*.

Durbin, R. (2014). Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272.

Ernst, C. and Rahmann, S. (2013). PanCake: A Data Structure for Pangenomes. In *Proc. of the German Conference on Bioinformatics 2013 (GCB'13)*, volume 34, pages 35–45.

Fan, L., Cao, P., Almeida, J., and Broder, A. Z. (2000). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293.

Farach, M. (1997). Optimal suffix tree construction with large alphabets. In *Proc. of the 38th Symposium on Foundations of Computer Science (FOCS'97)*, pages 137–143.

Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *Proc. of the 41st Symposium on Foundations of Computer Science (FOCS'00)*, pages 390–398.

Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. (2004). An alphabet-friendly FM-index. In *Proc. of the International Symposium on String Processing and Information Retrieval (SPIRE'04)*, volume 3246, pages 150–160.

Fischer, J., Mäkinen, V., and Navarro, G. (2009). Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364.

Fredking, E. (1960). Trie Memory. *Comm. ACM*, 3(9):490–499.

GENCODE group (2013). GENCODE database. `https://www.gencodegenes.org`. [Online; accessed 23-February-2017].

Genome Biology Editorial Team (2011). Closure of the NCBI SRA and implications for the long-term future of genomics data storage. *Genome Biol.*, 12(3):402.

Giancarlo, R., Rombo, S. E., and Utro, F. (2014). Compressive biological sequence analysis and archival in the era of high-throughput sequencing technologies. *Brief. Bioinform.*, 15(3):390–406.

Glenn, T. C. (2011). Field guide to next-generation DNA sequencers. *Mol. Ecol. Res.*, 11(5):759–769.

Grabowski, S., Deorowicz, S., and Roguski, L. (2015). Disk-based compression of data from genome sequencing. *Bioinformatics*, 31(9):1389–1395.

Hach, F., Numanagić, I., Alkan, C., and Sahinalp, S. C. (2012). SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28(23):3051–3057.

Harrow, J., Frankish, A., Gonzalez, J. M., Tapanari, E., Diekhans, M., Kokocinski, F., Aken, B. L., Barrell, D., Zadissa, A., Searle, S., et al. (2012). GENCODE: the reference human genome annotation for The ENCODE Project. *Genome Res.*, 22(9):1760–1774.

Heinz, S., Zobel, J., and Williams, H. E. (2002). Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223.

Hoff, K. J. (2009). The effect of sequencing errors on metagenomic gene prediction. *BMC Genom.*, 10:520.

Hogg, J. S., Hu, F. Z., Janto, B., Boissy, R., Hayes, J., Keefe, R., Post, J. C., and Ehrlich, G. D. (2007). Characterization and modeling of the Haemophilus influenzae core and supragenomes based on the complete genomic sequences of Rd and 12 clinical nontypeable strains. *Genome Biol.*, 8(6):R103.

Holland, R. C. and Lynch, N. (2013). Sequence squeeze: an open contest for sequence compression. *GigaScience*, 2:5.

Holley, G., Wittler, R., and Stoye, J. (2015). Bloom Filter Trie–A Data Structure for Pan-Genome Storage. In *Proc. of the 15th Workshop on Algorithms in Bioinformatics (WABI'15)*, volume 9289, pages 217–230.

Holley, G., Wittler, R., and Stoye, J. (2016). Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms Mol. Biol.*, 11:3.

Holley, G., Wittler, R., Stoye, J., and Hach, F. (2017). Dynamic Alignment-free and Reference-free Read Compression. In *Proc. of 21st International Conference on Research in Computational Molecular Biology (RECOMB'17)*, volume 10229 of *Lecture Notes in Computer Science*, pages 50–65.

Hosseini, M., Pratas, D., and Pinho, A. J. (2016). A Survey on Data Compression Methods for Biological Sequences. *Information*, 7(4):56.

Huang, L., Popic, V., and Batzoglou, S. (2013). Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40(9):1098–1101.

International Human Genome Sequencing Consortium (2001). Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921.

Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., and McVean, G. (2012). De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet.*, 44(2):226–232.

Jones, D. C., Ruzzo, W. L., Peng, X., and Katze, M. G. (2012). Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Res.*, 40(22):e171.

Kim, D., Langmead, B., and Salzberg, S. L. (2015). HISAT: a fast spliced aligner with low memory requirements. *Nat. Methods*, 12(4):357–360.

Kim, D., Langmead, B., and Salzberg, S. L. (2016). HISAT2 implementation. `https://github.com/infphilo/hisat2`. [Online; accessed 23-February-2017].

Kingsford, C. and Patro, R. (2015). Reference-based compression of short-read sequences using path encoding. *Bioinformatics*, 31(12):1920–1928.

Kirsch, A. and Mitzenmacher, M. (2006). Less hashing, same performance: Building a better Bloom filter. In *Proc. of the European Symposium on Algorithms (ESA'06)*, volume 4168, pages 456–467.

Knuth, D. E. (1998). *The Art of Computer Programming: Sorting and Searching*, volume 3. Pearson Education.

Laing, C., Buchanan, C., Taboada, E. N., Zhang, Y., Kropinski, A., Villegas, A., Thomas, J. E., and Gannon, V. P. J. (2010). Pan-genome sequence analysis using Panseq: an online tool for the rapid analysis of core and accessory genomic regions. *BMC Bioinform.*, 11(1):461.

Land, M., Hauser, L., Jun, S.-R., Nookaew, I., Leuze, M. R., Ahn, T.-H., Karpinets, T., Lund, O., Kora, G., Wassenaar, T., et al. (2015). Insights from 20 years of bacterial genome sequencing. *Funct. Integr. Genomics*, 15(2):141–161.

Ledford, H. (2016). AstraZeneca launches project to sequence 2 million genomes. *Nature*, 532(7600).

Leffler, E. M., Bullaughey, K., Matute, D. R., Meyer, W. K., Segurel, L., Venkat, A., Andolfatto, P., and Przeworski, M. (2012). Revisiting an old riddle: what determines genetic diversity levels within species? *PLoS Biol.*, 10(9):e1001388.

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

Loh, P.-R., Baym, M., and Berger, B. (2012). Compressive genomics. *Nat. Biotechnol.*, 30(7):627–630.

Luhmann, N., Thévenin, A., Ouangraoua, A., Wittler, R., and Chauve, C. (2016). The SCJ Small Parsimony Problem for Weighted Gene Adjacencies. In *Proc. of the International Symposium on Bioinformatics Research and Applications (ISBRA'16)*, volume 9683, pages 200–210.

Lukjancenko, O., Wassenaar, T. M., and Ussery, D. W. (2010). Comparison of 61 sequenced Escherichia coli genomes. *Microb. ecol.*, 60(4):708–720.

Mäkinen, V., Navarro, G., Sirén, J., and Välimäki, N. (2010). Storage and retrieval of highly repetitive sequence collections. *J. Comp. Biol.*, 17(3):281–308.

Manber, U. and Myers, G. (1993). Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948.

Marçais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770.

Marcus, S., Lee, H., and Schatz, M. C. (2014). SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483.

McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272.

Medini, D., Donati, C., Tettelin, H., Masignani, V., and Rappuoli, R. (2005). The microbial pan-genome. *Curr. Opin. Genet. Dev.*, 15(6):589–594.

Mikheyev, A. S. and Tin, M. M. Y. (2014). A first look at the Oxford Nanopore MinION sequencer. *Mol. Ecol. Res.*, 14(6):1097–1102.

Minkin, I., Pham, S., and Medvedev, P. (2016). TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*, page btw609.

Morrison, D. R. (1968). PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534.

Mosquera-Rendón, J., Rada-Bravo, A. M., Cárdenas-Brito, S., Corredor, M., Restrepo-Pineda, E., and Benítez-Páez, A. (2016). Pangenome-wide and molecular evolution analyses of the pseudomonas aeruginosa species. *BMC Genom.*, 17:45.

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

Navarro, G. (2012). Indexing highly repetitive collections. In *Proc. of the 23rd International Workshop on Combinatorial Algorithms (IWOCA'12)*, volume 7643, pages 274–279.

NCBI (2007). NCBI Sequencing Read Archive database. `https://trace.ncbi.nlm.nih.gov/Traces/sra/`. [Online; accessed 23-February-2017].

NCBI (2017). NCBI Genome database. `https://www.ncbi.nlm.nih.gov/genome/`. [Online; accessed 23-February-2017].

Nederbragt, L. (2016). Developments in NGS. `https://figshare.com/articles/developments_in_NGS/100940`. [Online; accessed 20-December-2016: 10.6084/m9.figshare.100940.v9].

Nguyen, N., Hickey, G., Zerbino, D. R., Raney, B., Earl, D., Armstrong, J., Haussler, D., and Paten, B. (2015). Building a pangenome reference for a population. *J. Comput. Biol.*, 22(5):387–401.

Nilsson, S. and Karlsson, G. (1999). IP-address lookup using LC-tries. *IEEE J. Sel. Area. Comm.*, 17(6):1083–1092.

Numanagić, I., Bonfield, J. K., Hach, F., Voges, J., Ostermann, J., Alberti, C., Mattavelli, M., and Sahinalp, S. C. (2016). Comparison of high-throughput sequencing data compression tools. *Nat. Methods*, 13(12):1005–1008.

Ohlebusch, E., Fischer, J., and Gog, S. (2010). CST++. In *Proc. of the International Symposium on String Processing and Information Retrieval (SPIRE'10)*, volume 6393, pages 322–333.

Paten, B., Diekhans, M., Earl, D., John, J. S., Ma, J., Suh, B., and Haussler, D. (2011). Cactus graphs for genome comparisons. *J. Comput. Biol.*, 18(3):469–481.

Patro, R. and Kingsford, C. (2015). Data-dependent bucketing improves reference-free compression of sequencing reads. *Bioinformatics*, 31(17):2770–2777.

Pavlov, I. (1999). LZMA compression library. `http://www.7-zip.org`. [Online; accessed 20-December-2016].

Putze, F., Sanders, P., and Singler, J. (2009). Cache-, hash- and space-efficient bloom filters. *ACM J. Exp. Algorithmic*, 14:9.

Rahn, R., Weese, D., and Reinert, K. (2014). Journaled string tree—a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics*, 30(24):3499–3505.

Raman, R., Raman, V., and Rao, S. S. (2007). Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algor.*, 3(4):43.

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

Rhoads, A. and Fai Au, K. (2015). PacBio Sequencing and Its Applications. *Genomics, Proteomics & Bioinformatics*, 13(5):278–289.

Roberts, M., Hayes, W., Hunt, B. R., Mount, S. M., and Yorke, J. A. (2004). Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369.

Roguski, L. and Deorowicz, S. (2014). DSRC 2–Industry-oriented compression of FASTQ files. *Bioinformatics*, 30(15):2213–2215.

Rozov, R., Shamir, R., and Halperin, E. (2014). Fast lossless compression via cascading Bloom filters. *BMC Bioinform.*, 15(9):S7.

Russo, L., Navarro, G., and Oliveira, A. L. (2011). Fully compressed suffix trees. *ACM Trans. Algor.*, 7(4):53.

Sadakane, K. (2007). Compressed suffix trees with full functionality. *Theor. Comput. Syst.*, 41(4):589–607.

Saha, S. and Rajasekaran, S. (2014). Efficient algorithms for the compression of FASTQ files. In *Proc. of the International Conference on Bioinformatics and Biomedicine (BIBM'14)*.

Sahinalp, S. C. and Vishkin, U. (1996). Efficient approximate and dynamic matching of patterns using a labeling paradigm. *Proc. of the 37th Annual Symposium on Foundation of Computer Science (FOCS'96)*.

Salikhov, K., Sacomoto, G., and Kucherov, G. (2014). Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithm. Mol. Biol.*, 9:2.

Sanger, F., Nicklen, S., and Coulson, A. R. (1977). DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. USA*, 74(12):5463–5467.

Schneeberger, K., Hagmann, J., Ossowski, S., Warthmann, N., Gesing, S., Kohlbacher, O., and Weigel, D. (2009). Simultaneous alignment of short reads against multiple genomes. *Genome Biol.*, 10(9):R98.

Schuster, S. C. (2008). Next-generation sequencing transforms today's biology. *Nat. methods*, 5:16–18.

Shendure, J. and Ji, H. (2008). Next-generation DNA sequencing. *Nat. Biotechnol.*, 26(10):1135–1145.

Sigaux, F. (1999). Cancer genome or the development of molecular portraits of tumors. *Bulletin de l'Academie nationale de medecine*, 184(7):1441–7.

Simpson, J. T. and Pop, M. (2015). The Theory and Practice of Genome Sequence Assembly. *Annu. Rev. Genom. Hum. Genet.*, 16:153–172.

Sirén, J. (2017). Indexing Variation Graphs. In *Proc. of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX'17)*, pages 13–27.

Sirén, J., Välimäki, N., and Mäkinen, V. (2011). Indexing finite language representation of population genotypes. In *Proc. of the 11th International Workshop on Algorithms in Bioinformatics (WABI'11)*, volume 6833, pages 270–281.

Sirén, J., Välimäki, N., and Mäkinen, V. (2014). Indexing graphs for path queries with applications in genome research. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, 11(2):375–388.

Solomon, B. and Kingsford, C. (2016). Fast search of thousands of short-read sequencing experiments. *Nat. Biotechnol.*, 34(3):300–302.

Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., and Robinson, G. E. (2015). Big data: astronomical or genomical? *PLoS Biol.*, 13(7):e1002195.

Tettelin, H., Masignani, V., Cieslewicz, M. J., Donati, C., Medini, D., Ward, N. L., Angiuoli, S. V., Crabtree, J., Jones, A. L., Durkin, A. S., et al. (2005). Genome analysis of multiple pathogenic isolates of Streptococcus agalactiae: implications for the microbial "pan-genome". *Proc. Natl. Acad. Sci. USA*, 102(39):13950–13955.

Tettelin, H., Riley, D., Cattuto, C., and Medini, D. (2008). Comparative genomics: the bacterial pan-genome. *Curr. Opin. Microbiol.*, 11(5):472–477.

Thudi, M., Li, Y., Jackson, S. A., May, G. D., and Varshney, R. K. (2012). Current state-of-art of sequencing technologies for plant genomics research. *Brief. Funct. Genomics*, 11(1):3–11.

Treangen, T. J., Ondov, B. D., Koren, S., and Phillippy, A. M. (2014). The Harvest suite for rapid core-genome alignment and visualization of thousands of intraspecific microbial genomes. *Genome Biol.*, 15(11):524.

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

Venter, J. C., Adams, M. D., Myers, E. W., Li, P. W., Mural, R. J., Sutton, G. G., Smith, H. O., Yandell, M., Evans, C. A., Holt, R. A., et al. (2001). The Sequence of the Human Genome. *Science*, 291(5507):1304–1351.

vg team (2015). vg implementation. `https://github.com/vgteam/vg`. [Online; accessed 23-February-2017].

Wandelt, S., Starlinger, J., Bux, M., and Leser, U. (2013). RCSI: Scalable similarity search in thousand(s) of genomes. *Proc. of the VLDB Endowment*, 6(13):1534–1545.

Watson, J. D. and Crick, F. H. (1953). The structure of DNA. In *Cold Spring Harb. Symp. Quant. Biol.*, volume 18, pages 123–131.

Weiner, P. (1973). Linear pattern matching algorithms. In *Proc. of the 14th Annual Symposium on Switching and Automata Theory (SWAT'73)*.

Williams, H. E. and Zobel, J. (1999). Compressing integers for fast file access. *Comput. J.*, 42(3):193–201.

Witten, I. H., Neal, R. M., and Cleary, J. G. (1987). Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540.

Zekic, T., Holley, G., and Stoye, J. (2018). Pangenome Storage and Analysis Techniques. In Setubal, J., Stoye, J., and Stadler, P., editors, *Comparative Genomics*, volume 1704 of *Methods in Molecular Biology*.

Zimin, A. V., Marçais, G., Puiu, D., Roberts, M., Salzberg, S. L., and Yorke, J. A. (2013). The MaSuRCA genome assembler. *Bioinformatics*, 29(21):2669–2677.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343.

Printed on non-ageing paper °° ISO 9706