UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DATA STRUCTURES AND ALGORITHMS FOR ANALYZING DNA SEQUENCES IN COMPRESSED SPACE

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN COMPUTACIÓN

DIEGO ALEJANDRO DÍAZ DOMÍNGUEZ

PROFESOR GUÍA:
GONZALO NAVARRO
PROFESOR COGUÍA:
TRAVIS GAGIE

MIEMBROS DE LA COMISIÓN:
NADIA PISANTI
DIEGO ARROYUELO
JUAN ASENJO

Este trabajo ha sido parcialmente financiado por Beca ANID 21171332, Financiamiento Basal FB0001, Proyecto Fondecyt 1-171058, Proyecto Fondecyt 1-170048, Proyecto Fondecyt 1-200038 y Centro de Biotecnología y Bioingeniería (CeBiB)

SANTIAGO DE CHILE
OCTUBRE 2021

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE DOCTOR EN COMPUTACIÓN
POR: DIEGO ALEJANDRO DÍAZ DOMÍNGUEZ
FECHA: OCTUBRE 2021
PROF. GUÍA: GONZALO NAVARRO Y TRAVIS GAGIE

DATA STRUCTURES AND ALGORITHMS FOR ANALYZING DNA SEQUENCES IN COMPRESSED SPACE

Los avances en las tecnologías de secuenciación del ADN han generado que hoy en día tengamos una gran cantidad de colecciones genómicas disponibles para analizar. El reto con estas colecciones es almacenar y procesar los datos sin agotar los recursos computacionales. Muchos autores han abordado este desafío utilizando estructuras de datos compactas y algoritmos que explotan las largas repeticiones de ADN en estos datasets. Estas técnicas han demostrado ser eficaces para reducir los elevados costes computacionales. Sin embargo, se han centrado principalmente en genomas ensamblados. Su uso en datos de secuenciación sin procesar (también conocidos como lecturas) es un tema menos estudiado. El diseño de nuevas estructuras de datos compactas y métodos de compresión para lecturas es una necesidad imperante, dado que estas colecciones genómicas son las más masivas y las más comunes.

Esta tesis presenta una infraestructura algorítmica diseñada principalmente para manipular colecciones de lecturas en espacio sucinto o comprimido. Nuestro objetivo principal es reducir los altos costos de extraer información biológica a partir de lecturas.

Comenzamos introduciendo un nuevo compresor de gramáticas llamado LMSg, el cual está destinado a almacenar lecturas. Nuestro método demuestra ser rápido, altamente paralelizable y con tasas de compresión competitivas con las de compresores populares. Nuestra siguiente contribución es un algoritmo llamado infBWT, el cual calcula la BWT extendida de una colección de lecturas codificadas con la gramática LMSg. El algoritmo utiliza las características particulares de la gramática LMSg y las corridas de símbolos iguales en la BWT para acelerar los cálculos. La BWT extendida es un elemento esencial en muchos autoíndices sucintos que podríamos utilizar para extraer información. Nuestros experimentos muestran que infBWT se hace más eficiente a medida que las lecturas se vuelven más massivas y repetitivas.

Nuestra tercera contribución es un índice sucinto para lecturas cuyo objetivo es extraer información biológica. Esta representación, llamada rBOSS, codifica las lecturas en un grafo compacto de de Bruijn (BOSS) y luego extiende el grafo con una nueva estructura de datos propuesta en esta tesis: el árbol de sobrelape. Además, mostramos que es posible combinar la idea del árbol de sobrelape con la BWT extendida para producir un autoíndice que codifica más información que rBOSS. Demostramos el uso práctico de rBOSS implementando un algoritmo para ensamblar las lecturas en un genoma.

Proponemos un índice sucinto alternativo para lecturas, también pensado para realizar análisis. Este índice se basa en los grafos de de Bruijn coloreados. Esta representación construye un grafo de de Bruijn a partir de las lecturas y asigna un color específico a cada camino etiquetado con una lectura. Nuestra contribución es un algoritmo codicioso que reduce

el uso de espacio coloreando parcialmente el grafo y dando los mismos colores a diferentes lecturas cuando es posible. Este enfoque disminuye el número de colores que el índice debe almacenar. Además, diseñamos dos algoritmos sobre el índice, uno extrae las lecturas del grafo y el otro ensambla el genoma de las lecturas.

Nuestra última contribución es un algoritmo práctico para producir una gramática localmente consistente a partir de un texto. Las propiedades particulares de nuestra gramática nos permiten obtener una variación del índice de gramáticas que mejora la complejidad de tiempo para localizar patrones largos, manteniendo altas tasas de compresión. Una característica importante de nuestro algoritmo es que, a diferencia de otras gramáticas con propiedades de consistencia local, no requerimos almacenar estructuras de datos adicionales, como permutaciones, para mantener la consistencia. Esta contribución está pensada para ser utilizada en el futuro para indexar colecciones de genomas completos, más que de lecturas.

Abstract

Rapid advances in DNA sequencing technologies have generated an unprecedented amount of genomic collections available for analysis. The challenge with these collections is to store and process the data without exhausting the computational resources. Many authors have addressed the problem by using *compact data structures* and algorithms that exploit the long DNA repetitions of the datasets. These techniques have proved to be efficient in reducing the high computational costs. However, they have been focused mainly on assembled genomes. Their use on raw sequencing datasets (a.k.a *reads*) is a less studied topic. Designing new compact data structures and *compression-aware* methods for reads is a pressing need as they are the most massive and common kind of genomic dataset one can find.

This thesis develops an algorithmic infrastructure designed primarily for manipulating read collections in succinct or compressed space. Our goal is to lower the computational costs of extracting biological information from reads.

We start by introducing a new *grammar* compressor called LMSg aimed at storing reads. Our method proves to be fast, highly parallelizable, and with compression ratios competitive with those of state-of-the-art compressors. Our next contribution is a compression-aware algorithm called infBWT that computes the *extended BWT* (eBWT) of a read collection encoded as an LMSg grammar. The algorithm uses the features of the LMSg grammar and the equal-symbol runs in the eBWT to boost the computations. The eBWT is an essential element in many succinct *self-indexes* that we could use to extract information. Our experiments show that infBWT gets more efficient as the input dataset becomes more massive and repetitive.

Our third contribution is a succinct index for reads tailored to extracting biological information. This representation, called rBOSS, encodes the input reads in a compact *de Bruijn* graph (BOSS) and augments the graph with a new data structure proposed in this thesis; the *overlap tree*. Further, we show that it is possible to combine the idea of the overlap tree with the eBWT to produce a more powerful self-index than rBOSS. We demonstrate the practical use of rBOSS by implementing an algorithm to assemble the reads into a genome.

We propose an alternative succinct index for reads, also tailored for analyses. This index relies on *colored* de Bruijn graphs. This representation builds a de Bruijn graph from the reads and assigns a specific color to every path spelling a read. Our contribution is a greedy algorithm that reduces space usage by partially coloring the graph and giving the same colors to different reads when possible. This approach decreases the number of colors the index has to store. Additionally, we design two algorithms on top of the index: one extracts the reads from the graph, and the other assembles the reads' genome.

Our last contribution is a practical algorithm for producing a *locally consistent* grammar from a string collection. The particular properties of our grammar allow us to obtain a variation of the *grammar index* that improves the time complexity for locating long patterns while maintaining high compression ratios. An important feature of our algorithm is that, unlike other grammars with local consistency properties, we do not require to store additional data structures, like permutations, to maintain consistency. This contribution is intended to be used in the future for indexing collections of complete genomes, rather than reads.

Dedicado a mi familia, de ahora y siempre

Agradecimientos

Mi tiempo en el doctorado finalmente llega a su fin. Fue una etapa enriquecedora, donde conocí gente nueva y aprendí a cómo realizar investigación en el área de las ciencias de la computación. Sin embargo, siento que sólo logré entender una pequeña porción de todos los temas que se pueden estudiar. Esta sensación, por su puesto, no es algo negativo. Al contrario, me motiva aún más a seguir avanzando en mi carrera científica.

Mis años en el programa estuvieron marcados por hitos importantes, como lo son la pandemia de COVID-19 y el estallido social en Chile. Puede que no haya una relación directa entre estos eventos y mi doctorado, pero sí influenciaron el desarrollo de mi tesis. Siempre recordaré lo que significó sobrellevar estos temas con mi investigación.

Me gustaría comenzar agradeciendo a mis tutores Gonzalo Navarro y Travis Gagie. Ellos fueron un pilar fundamental en el desarrollo de mi trabajo. El profesor Gonzalo me dio la libertad para investigar las cosas que a mi me interesaban. Siempre estuvo dispuesto a colaborar conmigo, y se tomó el tiempo para revisar mis documentos de manera exhaustiva. Travis, por otro lado, constantemente me puso en contacto con otros investigadores alrededor del mundo, interesados en los mismos temas que yo, y siempre me considero para colaborar. Además, siempre estuvo preocupado de ponerme al corriente de las últimas novedades en algoritmos y estructuras de datos compactas para Genómica.

También quisiera agradecer a los miembros de la comisión de mi tesis, Nadia Pisanti, Diego Arroyuelo y Juan Asenjo, por tomarse el tiempo de leer mi documento y hacer comentarios útiles. Estos comentarios ayudaron a mejorar bastante la versión final del escrito.

Otro pilar importante para el desarrollo del doctorado fue mi Familia. Ellos fueron la contención emocional que me permitió finalizar este proceso. Quisiera agradecer a María Ignacia, mi esposa, por vivir esta etapa conmigo. María también realizó estudios de post-grado, así que entiende mejor que nadie lo que significa. Su apoyo, compresión y cariño fueron indispensables. También quisiera agradecer a Sandra, Mauricio y Esteban, mis padres y hermano, respectivamente, por su paciencia y por ayudarme en todo lo que necesité.

Finalmente quisiera agradecer al Departamento de Ciencias de la Computación de la Universidad de Chile (DCC), al Centro de Biotecnología y Bioingeniería (CeBiB) y a Agencia Nacional de Investigación y Desarrollo (ANID) por financiarme durante este tiempo. Ellos me dieron la estabilidad económica que me permitió realizar mi investigación sin tener que preocuparme de nada más.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter describes the scope and structure of the thesis. Section 1.1 introduces the concept of DNA strings and explains how this subject motivates our work. It also discusses the current solutions and open problems to process this type of strings. Section 1.2 gives a detailed description of our contributions to the field. Finally, Section 1.3 explains how this thesis' content is distributed across the different chapters.

1.1 Motivation

DNA datasets are string collections that encode the relative order of nucleotides in molecules of *Deoxyribonucleic Acid* (DNA). The analysis *in silico* of their sequences allows uncovering complex biological signals that are difficult to detect with other approaches. The rapid development of sequencing[1] technologies [79] has dramatically dropped the cost of producing these datasets while increasing the performance of sequencers[2] [88]. Thus, decoding several hundred or even thousand individual genomes has become a feasible task [165, 41]. These factors have made storing and processing genomic data a significant computational challenge.

A sequencing experiment yields a collection with millions of short strings called *reads*. The length of these strings varies depending on the technology used to produce them. They can range from a couple of hundred characters up to several thousand. Still, the reads do not represent the source DNA's full sequence; they are only small overlapping fragments. To obtain the final product, we have to further process the reads on the computer.

There are two families of computational methods for processing reads, those that are reference-free and those that are reference-based. In the former, we connect the reads via suffix-prefix overlaps, and then we collapse the strings to form a group of consensus sequences. This process is known as DNA *assembly*. In the latter approach, we align the reads against a known *reference* built from a closely related genomic source. This reference can be, for instance, the genome of another individual from the same species that was previously assembled. In this case, instead of searching for consensus sequences, we look for mismatches in

[1] The process of spelling the nucleotides from a DNA molecule.

[2] The machine that carry outs the sequencing process.

the alignments as these differences *might* represent genetic variations.

Assembling reads entails several difficulties; the most notable ones are (i) the quadratic number of suffix-prefix overlap computations, (ii) how to represent the overlaps in the computer, and (iii) how to detect sequencing errors[3]. These problems have been addressed over the years by using algorithms and appropriate data structures for strings (e.g., [47, 97, 186, 77, 48, 192]), but genomic projects have become so massive that these solutions are no longer practical.

To put the problem's challenge in perspective, *The 100K Genomes Project* [60], for instance, aims to sequence 100,000 individuals from Great Britain affected by rare diseases or cancer. The raw sequencing data this project is expected to produce is at least one petabyte[4]. They aim to gather enough genetic information to enable the so-called precision medicine [67] in the country. Another equally ambitious sequencing initiative is *The Darwin Tree of Life* [145], whose goal is to sequence 70,000 organisms from different species in Britain and Ireland. We also have our own local initiative, the *1000 Chilean Genomes* project [40], which aims to sequence the genome of 1000 Chilean individuals and 1000 genomes of endemic species. Other similar efforts are being actively developed all over the world (see for instance [144, 87, 86, 42, 30]).

Bioinformatic tools resort to lossy[5] representations such as the *de Bruijn graph* [191, 39, 116, 119, 5] to cope with the high computational costs of assembling large volumes of reads. Still, these solutions yield fragmented and incomplete genomes as they lose information. Further, they also require large amounts of computational resources when the input is huge. For instance, in recent studies [58, 59] on a read collection of 323GB, the popular assemblers `SOAPdenovo2` [119] and `SPAdes` [5] required about 800GB of working memory (RAM) and more than one day of CPU time.

These problems have motivated the development of *compact data structures* [135] and *self-indexes* [137] for processing large volumes of genomic data [113, 24, 21, 184, 102]. Compact data structures are lossless representations that maintain the data using the least possible space, but at the same time, they allow us to access and query the data efficiently. On the other hand, self-indexes are text (e.g., DNA sequences) representations that rely on compact data structures and that do both encode the original text and support queries. The advantage of self-indexes is that they do not require the original input to extract the text or perform the queries. Additionally, their space usage is at most proportional to the size of the original text (i.e., they are succinct). Recent self-indexes [35, 70] use even less space than the original text and grow sublinearly with its size by exploiting its repetitions.

The FM-index [64, 62] is the most popular self-index in Genomics. Its main feature is that the cost of finding matches between an input pattern and the indexed text depends on the pattern size. Bioinformatic tools such as `bowtie` [107] or `bwa` [113] use this fact and encode the reference genome with the FM-index so that the cost of aligning a read on it depends on

[3]When a symbol is misspelled during the sequencing.

[4]The human genome has about three billion characters (3.1 GB), and the sequencing process yields at least three to four times that number of characters for a single individual.

[5]Representing the data in less space at the cost of losing information.

the read's length, not on the genome's length. This scheme is beneficial for reference-based genomic analyses that require processing millions of short reads.

The negative aspect of the methods that rely on alignments is that novel genetic variations are usually masked due to biases in the reference. Researchers have proposed to solve this problem by using *pangenomes* [56]. A pangenome is a string representation that encodes the genomes of several individuals of the same species. Aligning reads to such a data structure can yield more accurate results as this is a more realistic model for DNA [74]. To support the alignment, some authors have adapted the FM-index to encode pangenomes [102, 131]. These indexes use the fact that the genomes composing a pangenome are highly similar, and hence, exploiting their repetitiveness to compress their sequences will yield a small representation. As a consequence, the index's space and the time complexity for aligning reads depend more on the genetic differences among the individuals than the total number of characters in the pangenome. The development of these indexes is also an important milestone as it demonstrated that some bioinformatic tasks could also be carried out in compressed space.

Although pangenomes increase genomic analyses' accuracy, reference-free methods are still preferred as they are not biased. Besides, in situations where there is no previous knowledge about the sequenced organism, it is the only option. Given the FM-index's success in reference-based approaches, it is natural to wonder what kind of compact data structures and self-indexes are suitable to process reads in compressed space when there is no reference. The problem is not trivial; read collections are more massive than full genomes, and exploiting long DNA's repetitions is no longer an option as the sequencing breaks and scatters them into the reads. Although these aspects are challenging, compacting reads could help cope with the computer bottlenecks generated when processing massive genomic data. Considering how fast sequencing technologies are developing and how much they are diversifying to answer different biological questions, there is a pressure to design a new algorithmic infrastructure aimed to process read multisets in compressed space.

The overall aim of this thesis is to develop an algorithmic infrastructure, or toolbox, to analyze string collections of DNA sequencing reads in compressed space. The software will work with any type of read collection, but will be more efficient on those produced from Illumina, the most popular sequencing platform.

1.2 Thesis statement

We propose a framework of data structures and algorithms to manipulate collections of reads in compressed or succinct space. The proposed tools will exploit the natural repetitive patterns in DNA to reduce the high computational costs of analyzing sequencing experiments.

We divide our contributions into three main parts: first, a flexible compressed representation for storing reads; second, two succinct self-indexes for reads with support for string queries; and third, a dictionary-based self-index with potential applications to pangenomic analyses and read alignment.

We envision a workflow in which the data is always manipulated in compact form. First, the sequencing company compresses the reads using our compressor, and then delivers them

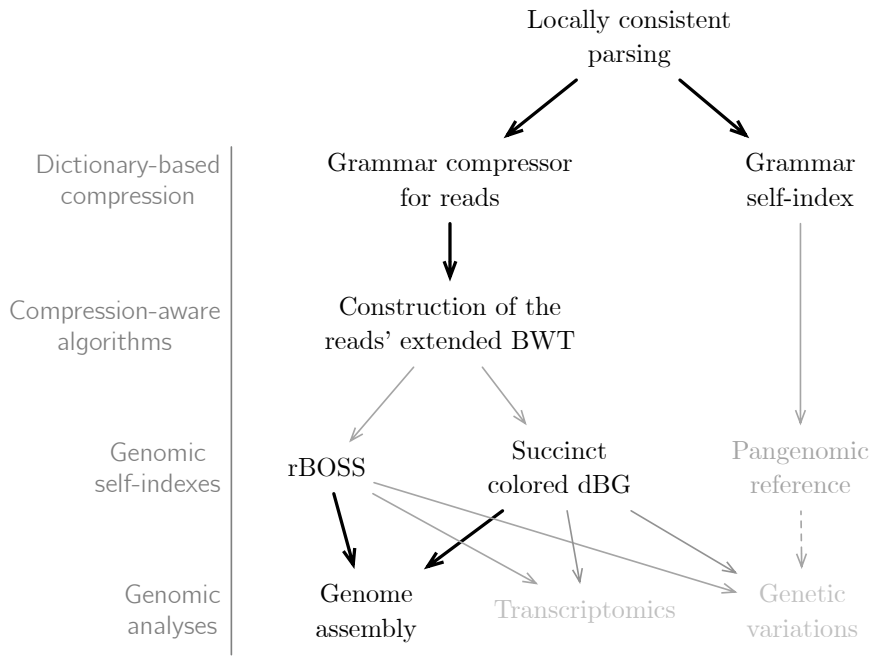

Figure 1.1: General outline of the thesis. The titles in gray to the left of the vertical line are the distinct computational topics covered in this thesis. The black titles to the right of the vertical line are the contribution of the thesis. An arrow between two contributions A and B is drawn if the output of A can be used as input for B. Gray arrows are relationships that were not developed in the thesis, but are considered for future work. Gray titles to the right of the vertical line are topics that are closely related with the contributions and are also considered for future work. The dashed vertical arrows between genomic self-indexes and genomic analyses are contributions that work in a reference-based setting, while solid arrows are contributions that work in a reference-free setting.

in compressed form. The final user receives the compressed data, and without fully decompressing it, transforms it in one of the succinct self-indexes we designed in this thesis. Finally, the user uses one of the genomic algorithms implemented on top of our self-indexes to extract biological information. Alternatively, the user can align the reads into a pangenomic data structure built on top of our dictionary-based self-index.

1.2.1 Contributions

Compressed representation for reads

We propose a new grammar compressor [96] for collections of sequencing reads. This new method is fast, achieves good compression ratios, and has a low memory footprint compared to other similar algorithms. Also, the grammar resulting from our method enables the computation of the BWT of the reads [44] in compressed space. The BWT is the main component in Wheeler graphs [69], a versatile family of graphs that supports several efficient string queries. In the context of genomics, Wheeler graphs could be helpful to extract biological information from the reads in succinct space. However, the BWT is difficult to compute when the input is massive as in the case of reads. Our algorithm for constructing the BWT exploits the repetitions captured by the grammar to make the construction process

more efficient.

Succinct data structures for genomic analyses

To obtain information for sequencing data efficiently, we need to index the reads with a succinct data structure that supports the navigation of their layout. More specifically, given the sequence of a read S, we can obtain the sequence of all the other reads that overlap S. This functionality is general enough to implement most genomic analyses.

We propose two succinct self-indexes that supports the navigation of the reads' layout; both rely on Wheeler graphs. The first one, which we called rBOSS, is an extension BOSS [24], a BWT-based encoding for de Bruijn graphs (dBGs) [46]. The main feature of rBOSS is that it allows us to compute suffix-prefix overlaps of less than k characters between dBG nodes, where k is the order of the dBG. We demonstrate the usefulness of rBOSS by implementing a simple genome assembler on top of it.

Our second self-index encodes a colored dBG [89] constructed from a read collection. It also builds on BOSS, but it takes a different approach. We give a specific color c to every read S, and then we assign c to the nodes in the dBG path labeled with S (there is only one path that meets this condition). We reduce space usage by assigning the same colors to different reads whose paths do not share nodes. The advantage of this setting is that if we reach a branching node during a graph traversal, we decide which edge to follow according the colors we have previously seen in the traversal. Most genomic analyses require traversing the dBG to extract information, but they stop when they reach branching nodes as it is not always possible to make safe assumptions about them. We also demonstrate the usefulness of our colored dBG by implementing a simple genome assembler on top it.

A grammar self-index

We propose a new grammar self-index with support for pattern matching. Our data structure uses less space than the classical FM-index as it exploits the DNA repetitions. On the other hand, it is faster for pattern matching than the regular grammar self-index when the pattern is long (hundreds of characters).

We build the index's grammar using locally consistent parsing [162]. This technique consists in partitioning a text such that the occurrences of the same pattern yield almost the same partition into blocks. The only blocks that might differ are those at the ends of the pattern's occurrences, where the context changes. When we perform pattern matching over this index, we preprocess the input string P using the same parsing algorithm we used for building the index's grammar. As the parsing is locally consistent, most of the P's blocks should exist in the grammar if P exists in the text. This property makes the search in the index more straightforward.

We believe that further development on grammar self-indexes with locally consistent properties can yield efficient pangenomic representations that support approximated alignments for long reads. This idea will become relevant in the next years due to the rapid development of recent DNA sequencing technologies such as Nanopore [180] and PacBio [17]. These technologies are increasing the read lengths at the time they improve the sequence accuracy.

1.3 Structure of the Thesis

We divide this work into ten chapters:

- Chapter 1 is the introductory part of the thesis.
- Chapter 2 describes the fundamental aspects of information theory and compact data structures.
- In Chapter 3, we review the state of the art in text compression and indexing. We also describe how these ideas extend to labeled graphs. These concepts, along with those of Chapter 2, are the basis on which our contributions are built.
- Chapter 4 addresses the main concepts in Computational Genomics. We briefly explain the different types of DNA strings and how they are obtained. We also give a general description of the computational approaches used in Bioinformatics to transform read collections into biological information. Finally, we introduce the concept of reference genomes and pangenomes and review some common techniques to align reads against them.
- In Chapter 5, we present our first contribution, the grammar algorithm to compress reads. We also describe a succinct grammar representation that we use later to compute the BWT of the reads.
- In Chapter 6, we develop an algorithmic framework for producing the extended BWT (eBWT) of the reads from the grammar representation of Chapter 5.
- Chapter 7 introduces our first self-index for navigating the reads' layout, the one we called rBOSS. We also present the genome assembler we built on top of rBOSS.
- Chapter 8 explains our second self-index for navigating the read's layout, the colored dBG. Similarly to what we did in the previous chapter, we explain the basic ideas to perform genome assembly on top of the colored dBG.
- Chapter 9 introduces a grammar self-index with local consistency properties. The grammar algorithm used for this self-index is based on the grammar algorithm we described on Chapter 5. In Chapter 9, we also briefly describe how locally consistent grammars can be useful in the future for aligning long reads in pangenomic sequences.
- The final chapter discusses our results and future work directions.

1.4 Software

All the algorithms and data structures developed in this thesis were written in `C++` and on top of the `SDSL-lite` library [76]. This library implements many compact data structures proposed in the literature. Still, we wrote our own versions of some of them as the `SDSL-lite` not always had the implementations we required, or it had them not in the way we needed them. We gather our implementations in a small library of compact data structures. The list of github repositories are listed below:

- `CDT`, a small library of compact data structures:
  `https://bitbucket.org/DiegoDiazDominguez/compact-data-structures/src/master`
- `LPG`, a grammar compressor for reads:
  `https://bitbucket.org/DiegoDiazDominguez/lms_grammar/src/bwt_imp2`

- `infBWT`, computing the eBWT from grammar-compressed reads:
  `https://bitbucket.org/DiegoDiazDominguez/lms_grammar/src/bwt_imp2`
- `rBOSS`, a self-index for navigating the reads:
  `https://bitbucket.org/DiegoDiazDominguez/eboss-dt/src/master`
- `cdBG`, a succinct colored dBG for reads:
  `https://bitbucket.org/DiegoDiazDominguez/colored_bos/src/master`
- `LPG grid`, a locally consistent grammar self-index:
  `https://github.com/ddiazdom/LPG/tree/LPG_grid`

1.5 Notation

Logarithms Many time and space complexities in this thesis involve logarithms of base 2. We refer to them just as log. For instance, we write $\log_2 n$ as $\log n$. When the logarithm base is different from 2, say x, we explicitly write it as $\log_x$.

Computation model We use the word RAM model of computation. In this model, we asume the data is stored in random-access memory and manipulated in words of $w = \Theta(\log n)$ bits, where n is the input size. These words can store values within the range $[0, 2^w - 1]$, which we can manipulate in constant time. We also assume we can perform logical and arithmetic operations over the words in constant time.

Chapter 2

Basic Concepts

In this chapter, we explain the basic ideas to understand our contributions. We start in Section 2.1 by briefly explaining some introductory concepts about data compression. In Section 2.2, we describe the most important compact data structures on which most of our contributions rely. Finally, in Section 2.3, we describe the concept of hashing and show some of its applications. Hashing is an essential tool for constructing the data structures we propose in the thesis. Besides, it is widely used in Bionformatics. In particular, the hashing applications of Sections 2.3.2, 2.3.3 and 2.3.4 serve as a base to understand the ideas behind the state-of-the-art tools to align reads to reference genomes, which we review in Section 4.

2.1 Data Compression

Data compression deals with representing the information in fewer bits. This concept is central in this thesis as we are dealing with high volumes of data. We now present the basic concepts of data compression on which most of the ideas developed in the following chapters are based.

2.1.1 Entropy

The most basic tool for measuring compression is the *worst case entropy*, here denoted $\mathcal{H}_{WC}$. Assume we have a set Σ of symbols and give equal-size unique codes of l bits to its elements. $\mathcal{H}_{WC}$ is a measure that tells us which is the minimum value for l so we can unambiguously recognize the symbols of Σ when read from a bitstream. This value is

$$\mathcal{H}_{WC}(\Sigma) = \log |\Sigma|.$$

Thus, we can store a sequence of n symbols in $n \log |\Sigma|$ bits of space. In our case, DNA strings have an alphabet of four letters, {a,c, g, t}, so we require codes of $\lceil \log 4 \rceil = 2$ bits. These codes are 00, 01, 10 and 11. Notice that this is much more succinct than the 8-bit cells used in modern computers to represent plain characters. Still, $\mathcal{H}_{WC}$ is not the limit for data compression, we can do better.

When not all the elements in Σ are equally likely to appear in a sequence, we can assign

variable-length codes to decrease the average code lengths. In *information theory*, the minimum possible value for that average is known as the *Shannon entropy*. Suppose that each $u \in \Sigma$ is produced from an infinite source with probably p_u, then the Shannon entropy is computed as

$$\mathcal{H}(\{p_u\}) = \sum_{u \in \Sigma} p_u \log \frac{1}{p_u}.$$

This value is also known as the *statistical entropy*. In general, giving shorter identifiers to the more probable symbols and longer ones to the less probable decreases the average length of the codes. In fact, the formula above suggests that the optimal length for u is $\log \frac{1}{p_u}$. The more skewed are the symbols' probabilities, the smaller is the value of $\mathcal{H}(\{p_u\})$. When all the elements have equal probability $\frac{1}{|\Sigma|}$, the statistical entropy is $\log |\Sigma|$. In this scenario, the set Σ is considered to be *incompressible*, and the best option is to use equal-sized codes.

One can generalize the concept of entropy and consider that the probability of emitting u is not independent but conditioned by the last k elements generated by the source. In this case, we have that the entropy is:

$$\mathcal{H}(\{p_{u|C}\}) = \sum_{C \in \Sigma^k} P_C \sum_{u \in \Sigma} p_{u|C} \log \frac{1}{p_{u|C}},$$

where C are the distinct strings of length k formed with the elements of Σ, P_C is the global probability for the source to emit C, and $p_{u|C}$ is the probability of u given that the last k previous characters emitted by the source form the word C.

We can extend the concept of $\mathcal{H}$ to compute the entropy of a finite sequence $S[1, n] \in \Sigma$. This measure is known as the *zeroth order empirical entropy* of S, and we compute it as

$$\mathcal{H}_0(S) = \sum_{u \in \Sigma} \frac{n_u}{n} \log \frac{n}{n_u},$$

where n_u is the frequency of u in S. The value $\frac{n_u}{n}$ is an estimate of the probability p_u for the hypothetical source that produced S to emit u. As with $\mathcal{H}$, the value of $\mathcal{H}_0(S)$ decreases as the symbol frequencies in S become skewed. In the memoryless model of statistical entropy, $n\mathcal{H}_0(S)$ is the lower bound to store S. However, in some circumstances, we can do better.

Similarly as with the generalization of the Shannon entropy, we can assume that the probability of each character $S[i]$, with $i \in [k + 1, n]$, depends on the previous k elements. This concept is known as the *kth order empirical entropy* of S:

$$\mathcal{H}_k(S) = \sum_{C \in \Sigma^k} \frac{n_C}{n} \mathcal{H}_0(S_C),$$

where S_C is the string formed by concatenating the symbols that follows C in S and n_C is the length of S_C. In this formula, a symbol $u \in \Sigma$ can have multiple frequencies depending on which distinct sequences C of length k precede it on S. We can give multiple codes to u depending on those sequences C. Consequently, the lower bound for storing S becomes $n\mathcal{H}_k(S)$. Further, if the symbols of S are better predicted by knowing the previous k elements, then $\mathcal{H}_k(S)$ is smaller than $\mathcal{H}_0(S)$, thus reaching better compression.

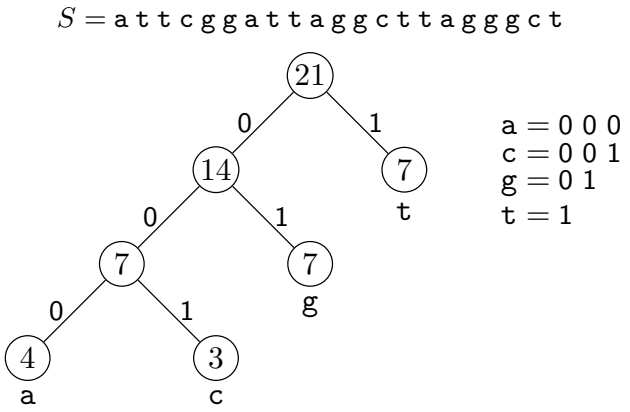

Figure 2.1: Example of a Huffman tree for the DNA string on the top. Numbers inside the leaves are the symbol frequencies. The final codes for Σ are shown on the right side of the figure.

2.1.2 Encoding Sequences

An *encoding* is an injective function $\mathcal{C} : \Sigma \to \{0,1\}^*$ that assigns a distinct sequence of bits $\mathcal{C}(c)$ to every symbol $c \in \Sigma$. To encode a string $S[1,n]$ over the alphabet Σ with $\mathcal{C}$, we scan S from right to left and append every $\mathcal{C}(S[i])$ to a new bitmap B. In general, we are interested in a function $\mathcal{C}$ that (i) reduces the length of B, but at the same time, (ii) allows decoding the original symbols of S. In Section 2.1.1, we already showed that we could achieve (i) if the codes in $\mathcal{C}$ are of variable length. For (ii), the codes must also be *unambiguous*. More specifically, there is no ambiguity in decoding the symbols of S while reading B from left to right. It is also helpful for codes in $\mathcal{C}$ to be *instantaneous*, meaning that we have enough information to determine c as soon as we finish reading the bits in $\mathcal{C}(c)$. Instantaneous codes are also *prefix-free*; no code is a prefix of another code. The advantage of prefix-free codes (and instantaneous codes) is that they do not depend on their context to be decoded. In what follows, we describe some basic encoding techniques.

Huffman

Huffman coding [84] is a popular technique to generate an optimal set of variable-length codes for Σ. Its main features are that it produces a prefix-free set of codes and that the average length of these codes almost reaches the statistical entropy. By giving Huffman codes to the symbols of S, we can compress the sequence to less than $n(\mathcal{H}_0(S) + 1)$ bits of space. This coding scheme achieves zeroth-order compression by exploiting the unbalance in the frequencies of S's symbols. More precisely, it gives longer codes to less frequent symbols and shorter codes to more frequent ones.

The algorithm to produce Huffman codes generates a binary tree from Σ. In this tree, there are $|\Sigma|$ distinct leaves, one for each symbol. The path from the tree's root to each of the leaves represents the binary code for the leaf's label. We obtain that code by traversing the path from the root to the leaf. When visiting an internal node, if the path continues through the left child, then we append a 0 to the code. In the other case, with the path continuing through the right child, we append a 1.

The algorithm starts by defining $|\Sigma|$ trees, one per distinct symbol in the alphabet. At the beginning, these trees have only one node, a leaf labeled with their corresponding symbol. Each tree also has a weight, which is the sum of the frequencies of its leaf labels. The algorithm's main idea is to pick the two trees with the smallest weights x and y, and merge them as the children of a new node with weight $x + y$. The process continues until only one tree remains. Figure 2.1 shows an example of a tree resulting from the Huffman algorithm.

When storing S using Huffman codes, we require to maintain the tree to extract the original symbols from the bitstream B. The decoding process is simple; we initialize a finger at the root of the Huffman tree. If $B[1] = 0$, then we move the finger to the left child of the root, otherwise we move the finger to the right child. If we reach an internal node, then we move one bit forward in B and repeat the process. If, on the other hand, we reach a leaf, then we spell its label, as we have finished decoding a symbol. Subsequently, we move to the next position of B and move the finger back to the root to start the decoding of the next symbol of S.

The main drawback of Huffman coding is that the space overhead of representing the tree can be considerable when Σ is large. A solution for this problem is to produce a *canonical* Huffman tree [169, 114]. In this tree, the leaf depths are nondecreasing when read from left to right. This topology enables a more efficient decoding and a more compact representation for the tree that requires $|\Sigma| \log \Sigma + \mathcal{O}(\log n)$ bits of space (see Section 2.6.3 in Navarro [135]).

Gamma and Delta

Delta (δ) and *Gamma* (γ) are techniques aimed to encode positive integers [135]. They work well when S is mainly composed of small values (we assume Σ is an alphabet of integers). Unlike Huffman, δ and γ codes do not depend on the sequence, so it is unnecessary to maintain additional data structures to retrieve the original symbols. Both schemes store $c \in \Sigma$ attached with the length $l = |c|$ of its binary representation. Still, they differ in how they represent l. In γ-codes, the formula is

$$\gamma(c) = 0^{l-1}1 \cdot [c]_{l-1},$$

where $[c]_{l-1}$ are the $l-1$ least significant bits of c. We compute l by counting the number of 0s from left to right until finding a 1 bit. This representation uses $2|c| - 1 = 2\lfloor \log c \rfloor + 1 = \mathcal{O}(\log c)$ bits of space. As the code's length is stored in unary, γ-codes are suitable when the values in S are small. For δ-codes, the formal expression is:

$$\delta(c) = \gamma(l) \cdot [c]_{l-1}.$$

Reading the symbol c from $\delta(c)$ requires to first extract l from the nested $\gamma(l)$ code. This representation uses $\log c + \mathcal{O}(\log \log c)$ bits of space, so it is suitable when the symbols in S are not so small. In general, storing S with δ-codes will be more efficient than using γ-codes if the symbols in S are equal or greater than 32. Both γ and δ codes produce prefix-free codes for Σ.

2.1.3 Direct Access to Variable-Length Codes

The problem with the variable-length codes we described in the previous section is that they do not allow direct access to the symbols. Accessing $S[i]$ requires us to decode the whole prefix $S[1, i-1]$ first.

A standard solution to support direct access is to sample positions in B at regular intervals of S. We select a parameter k and logically partition S into $\lceil n/k \rceil$ consecutive blocks. We create an extra array $P[1, \lceil n/k \rceil]$ to store the sampled pointers. For every block j in the partition, we store the bit position in B of its first element $S[(j-1)k+1]$ in $P[j]$. Thus, if we want to access $S[i]$, we have to compute its corresponding block $b = \lceil i/k \rceil$, and then start to decode symbols in B from index $P[b]$ until reaching the position that stores $S[i]$. This process decodes $i - (b-1) \cdot k \leq k$ symbols to retrieve $S[i]$, so the time complexity for the direct access of $S[i]$ is $\mathcal{O}(k)$ time. We can achieve constant time by using a second level of pointers. We define an extra array $P'[1, n]$ and store in $P'[i]$ the offset in B from the starting position of $S[(\lceil i/k \rceil - 1)k + 1]$ and the starting position of $S[i]$. In this way, we can obtain the position in B for $S[i]$ as $P[\lceil i/k \rceil] + P'[i]$.

Elias-Fano [61, 57] is a variable-length encoding that provides a more sophisticated solution for direct access by storing the code lengths apart from the codes. Suppose we represent S in the bit vector B by using a variable-length representation of any kind. In addition to B, we create another bit vector $M[1, |B|]$ in which we set $M[j] = 1$ iff $B[j]$ is the leftmost bit of a code. To access $S[i]$, we look in M for the ith and $(i+1)$th bits set. These indexes will give us the range in B for $S[i]$. An advantage of efficiently delimiting all the codes is that they do not need to be prefix-free, so they can be shorter. In particular, for storing small integers, we can encode c in B using $|c|$ bits, which makes the scheme similar to γ-codes, where instead of storing the length l as a prefix of c, we store l in M.

For the particular case of Elias-Fano encoding, we use *partial sums* to support direct access. Given a list L of integers, a partial sums data structure answers the queries:

- $\mathsf{sum}(L, i)$: cumulative sums of symbols up to position $L[i]$
- $\mathsf{search}(L, j)$: minimum index i in L such that $\mathsf{sum}(L, i) \geq j$

We augment the bit array M in the Elias-Fano encoding with partial sums so that obtaining the range in B for $S[i]$ reduces to compute $\mathsf{search}(M, i)$, $\mathsf{seach}(M, i+1) - 1$.

We implement the partial sums data structure by sampling the cumulative sums of the bits in M. We choose a parameter k and create an array $P[0, \lfloor n/k \rfloor]$ of sampled sums. In every $P[j]$, with $j \in [0, \lfloor n/k \rfloor]$, we store the value obtained by adding the bits in the prefix $M[1, jk]$. If we want to know answer $\mathsf{sum}(M, i)$, we have to perform the operation

$$P[\lfloor i/k \rfloor] + \sum_{u=\lfloor i/k \rfloor k+1}^{i} M[u],$$

which takes $\mathcal{O}(k)$ time. Answering $\mathsf{search}(M, j)$ requires us to perform a binary search over P to find the position a such that $P[a] \leq j < P[a+1]$. Once we find it, we linearly scan M

from index ak until finding a position i such that

$$P[a] + \sum_{u=ak+1}^{i} L[u] \geq j.$$

The whole process takes $\mathcal{O}(\log |P| + k)$ time.

Another option is to augment M with the select data structure (Section 2.2.1) so the cost of obtaining the range in B for $S[i]$ reduces to $\mathcal{O}(1)$.

2.2 Compact Data Structures

As its name suggests, *compact data structures* (CDS) are representations that maintain the data in a compact way, although this is not their only feature; they can also access and query the data efficiently. In many circumstances, their performance is competitive with classical textbook data structures. These characteristics make them a good alternative when dealing with massive collections.

2.2.1 Bit vectors

The *bit vector* is the basis for most of the CDSs. It consists of an array $B[1, n]$ where the only possible values are 1 or 0.

Compressed bit vectors

As with regular sequences, we can compress bit vectors when the number of 1s is much smaller than the number of 0s, or vice-versa. A popular succinct representation that exploits this fact is RRR [156]. This method partitions B into blocks of fixed size b and classify the blocks into classes. If a block in B has c 1s, then its class is c. Further, it gives identifiers to the blocks within the same class c. Every identifier in c denotes a specific way to arrange c 1s within b bits. RRR encodes every ith block $B[(i-1)b+1, ib]$ as a pair (c_i, o_i), where c_i is its class and o_i is its identifier. This pair enables the retrieval of the original block $B[(i-1)b+1, ib]$. RRR stores the classes of B in an array $C[1, \lceil n/b \rceil]$ such that $C[i] = c_i$. It also creates an array $O[1, \lceil n/b \rceil]$ storing the identifiers in the same way. Note the number of identifiers for a class c is upper-bounded by $l_c = \binom{b}{c}$, and when c is close to b, l_c is small. This also happens when c is very small compared to b. If the number of 1s and 0s is not even, then it is more likely to have blocks whose identifiers require few bits. This fact makes O be composed mainly of small numbers. RRR exploits this fact and encodes O using variable-length codes to thus achieve compression. The space usage of this representation is $n\mathcal{H}_0(B) + o(n)$ bits.

When the difference between the number of 1s and 0s is considerable in B, we can use more suitable encodings. In particular, if there are much fewer 1s than 0s, we can replace B with an integer array P storing the positions of B where the bits are set. As the resulting P is strictly increasing, we can encode its values as positive consecutive differences. More specifically, we replace every $P[i]$ with $P[i] - P[i-1]$, except for $P[1]$ that remains unchanged. We then store the differences using δ-codes (Section 2.1.2). This representation also uses space close to $n\mathcal{H}_0(B)$ bits.

In the next section, we briefly describe how to support the most important queries for bit vectors, rank and select. The methods we will explain assume the input bit vector is uncompressed.

Rank

The operation $\mathsf{rank}_1(B, i)$ returns the number of 1s up to position i in B. The solution for answering this query is rather similar to the idea of partial sums (Section 2.1.3); we choose a parameter k and logically divide B into blocks of $s = wk$ bits (w is width of the machine word). Subsequently, we create an array $R[0, \lfloor n/s \rfloor]$ in which every $R[j]$, with $j \in [1, \lfloor n/s \rfloor]$, stores the sum of 1s in the prefix $B[1, js]$. We also set $R[0] = 0$. We can now solve $\mathsf{rank}(B, i)$ by first retrieving the precomputed sum up to $i' = \lfloor i/s \rfloor s$ from $R[\lfloor i/s \rfloor]$, then counting the 1s in $B[i' + 1, i]$, and finally adding both results.

To achieve constant time, we divide every block $R[j]$ into k mini blocks of w bits each, and store their accumulative sums in another array $R'[0, \lfloor n/w \rfloor]$. More specifically, every $R'[u]$, with $u \in [1, \lfloor n/w \rfloor]$, contains the number of 1s in $B[1, uw]$ minus $R[\lfloor uw/s \rfloor]$. As before, we set $R'[0] = 0$. Note that the values in R' are never greater than s. The final equation to solve rank is:

$$\mathsf{rank}(B, i) = R[\lfloor i/s \rfloor] + R'[\lfloor i/w \rfloor] + \mathsf{popcount}(B, \lfloor i/w \rfloor w + 1, i),$$

where popcount returns the number of bits set in the range $B[\lfloor i/w \rfloor w + 1, i]$. Many programming languages natively support this function, and it is considered constant-time in practice as there are efficient implementations for it. Still, it receives as input an integer, not a bit vector. Using the two-level scheme that includes R and R', we have that the segment $B[\lfloor i/w \rfloor w + 1, i]$ spans at most w bits, which fits one computer word. We can extract that segment and store it in a machine word to pass it to popcount, and thus the rank operation takes $\mathcal{O}(1)$.

The cost of R and R' is $(n/s)w + (n/w)\log s = n/k + n\log(wk)/w$ bits. By choosing $k = w$, that space becomes $\mathcal{O}(n\log(w)/w) = \mathcal{O}(n\log\log n/\log n) = o(n)$ bits. If we also consider the n bits of B, then the total space to support rank is $n + o(n)$ bits.

Select

The operation $\mathsf{select}_1(B, r)$ returns the position in B storing the rth 1 from left to right. We can think of this function as the inverse of rank. The method we will explain to support select takes $\mathcal{O}(\log\log n)$ time and uses $o(n)$ bits on top of B. It is slower than the rank data structure we described in the previous section, and uses more space in practice. Still, it has a reasonable performance in most applications.

Let m be the number of bits set in B. We start by defining a parameter s and an array $S[0, \lceil m/s \rceil]$. We use s to sample one select answer every s 1s in B. Thus, $S[p]$, with $p \in [0, \lceil m/s \rceil - 1]$, stores the position in B for the bit 1 with rank $ps + 1$. We initialize $S[\lceil m/s \rceil] = n + 1$ as a border case.

We also create a bit vector $V[1, \lceil m/s \rceil]$ to mark the different blocks of B covered by the positions stored in S. More specifically, we set $V[p] = 1$ if the range $B[S[p], S[p + 1] - 1]$

spans more than $s \log^2 n$ bits (long block), and set $V[p] = 0$ otherwise (short block). After building V, we give rank_1 support to it. In addition, we create a vector $I[1, \mathsf{rank}_1(V, \lceil m/s \rceil)s]$ to store the select answers for the long blocks. Thus, if $V[p] = 1$, we explicitly store the s select positions of $B[S[p], S[p+1]-1]$ in the range $I[(p'-1)s+1, p's]$, where $p' = \mathsf{rank}_1(V, p)$. The last two elements of the select data structure are the arrays R and R' of Section 2.2.1. To obtain the desired complexities, we set the sampling rate of R to $s = \log^2 n \log\log n$ and the sampling rate of R' to $\log n \log\log n$.

We implement the $\mathsf{select}(B, r)$ algorithm as follows; we first obtain the block $p = \lceil r/s \rceil$. If $V[p] = 1$, then r falls in a long block. Therefore, we obtain its position in B directly from $I[(p' - 1)s + ((r - 1) \bmod s) + 1]$. When $V[p] = 0$, we need to search in B the answer, but we speed up the process using S, R, and R'. We first limit the search space in B to the range $b = S[\lceil r/s \rceil - 1]$, $e = S[\lceil r/s \rceil] - 1$. Subsequently, we perform a binary search over the range $R[\lfloor b/s \rfloor, \lfloor e/s \rfloor]$ to find the maximum position j such that $R[j] < r$. After that, we perform a second binary search over the range in R' that matches the block $R[j]$ to find the maximum position j' such that $R[j] + R'[j'] < r$. Finally, we perform a linear scan over the segment of B that matches the cell $R'[j']$ and we advance until we reach the select answer.

By using the threshold $s \log^2 n$ to classify the blocks of B, we ensure that the space usage of I stays within $o(n)$ bits. Every long block uses $s\lceil \log n \rceil$ bits to store the select answers and there are no more than $n/(s \log^2 n)$ long blocks. Therefore, the space usage of I is $s\lceil \log n \rceil \; n/(s \log^2 n) = n/\log n = o(n)$ bits. Additionally, with the sampling rates of R, R', their space usage stays $o(n)$ bits.

The expensive part of select is when r falls within a short block as we have to search it in B. Still, with the sampling rates of S,R and R', we ensure $\mathcal{O}(\log\log n)$ time for that operation. Note a short block in B spans no more than $s \log^2 n$ bits, and the sampling rate of R is s. Therefore, the binary search over R inspects no more than $\log^2 n$ consecutive cells, which takes $\mathcal{O}(\log\log n)$ time. The binary search over R' also takes $\mathcal{O}(\log\log n)$ time because a block of R spans $\log n$ cells of R'. The final linear scan over B should take $\log n \log\log n$ time as this is the number of bits a cell in R' spans. Still, we can speed up the process to $\mathcal{O}(\log\log n)$ if we advance in B by popcounting on chunks of $\log n$ bits. Thus, the final time complexity for select is $\mathcal{O}(\log\log n)$ time.

We can obtain $\mathcal{O}(1)$ time for select and maintain the space complexity in $o(n)$ bits. The general idea is to subdivide the short blocks of B into miniblocks, classify the miniblocks into short and long, and then store the select answers for the long ones. In practice, however, this approach uses a lot of extra space.

2.2.2 Wavelet Trees

In Section 2.1.2, we already described how to compress strings using variable-length codes. However, these representations cannot answer queries other than extracting the original symbols. This section describes a CDS called the *wavelet tree* [80], which also enables rank and select functionality on the sequence, and more.

Let $S[1, n]$ be a string over the alphabet Σ. The algorithm for building a wavelet tree T for S is as follows; we divide Σ in two classes, $\Sigma^l = [1, \lceil \sigma/2 \rceil]$ and $\Sigma^r = [\lceil \sigma/2 \rceil + 1, \sigma]$.

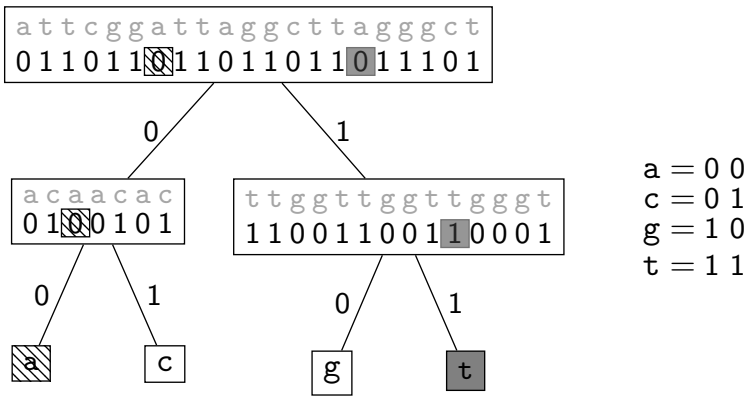

Figure 2.2: Wavelet tree T for the string `attcggattaggcttagggct` of Figure 2.1. Values in gray on top of the internal nodes are the original sequence symbols, and they are not stored explicitly in the wavelet tree. The binary strings to the right are the codes of the symbols, of length $\log \sigma$. The dashed boxes represent the path traversed for operation $\mathsf{access}(T, 7) = \mathsf{a}$. The shaded boxes show the path for operation $\mathsf{rank}_{\mathsf{t}}(T, 16) = 6$.

Subsequently, we create a binary vector $B[1, |S|]$ in which we set $B[i] = 0$ if $S[i]$ belongs to Σ^l or 1 otherwise. Then, we split S into two strings; S^l and S^r. The string S^l will store the symbols in S that belong to Σ^l and the string S^r will store the symbols in Σ^r. We maintain the relative order that the characters in S^l and S^r originally had in S. Finally, we create a root for T associated with B and with two children. The left and right children of the root are recursively created from the pairs (S^l, Σ^l) and (S^r, Σ^r), respectively. The base case of the recursion is when the alphabet has only one symbol $a \in \Sigma$, in which case we create a leaf labeled with a. After building T, we augment the bit vectors of its internal nodes with rank and select data structures. These data structures include queries for both bits, 1 and 0. Once the construction algorithm finishes, we can discard S. Figure 2.2 shows an example of a wavelet tree.

The most basic queries we can answer with the wavelet tree are:

- $\mathsf{access}(T, i)$: retrieves the symbol at position $S[i]$
- $\mathsf{rank}_a(T, i)$: number of symbols a up in the prefix $S[1, i]$
- $\mathsf{select}_a(T, r)$: position j where the rth symbol a lies in S

For answering $\mathsf{access}(T, i)$, we start a top-down traversal of T. Let B be the bit vector stored in the root v of T. If $B[i] = 0$, then we update the index as $i = \mathsf{rank}_0(B, i)$ and move to the left children of the root. On the other hand, if $B[i] = 1$, then we update the index to $i = \mathsf{rank}_1(B, 1)$ and move to the right child instead. From either child, say v', we apply the same procedure as with v, using its bit vector and the recently updated index i. The traversal of T stops when we reach a leaf u, in which case we return its label. The dashed boxes of Figure 2.2 shows an example of $\mathsf{access}(T, i)$.

The procedure for answering $\mathsf{rank}_a(T, i)$ is somewhat similar to that of access. The main difference is that the nodes we visit in T depend on the binary code of a, not of $S[i]$. Suppose during the traversal of T we reach an internal node v at depth h. If the hth most significant bit of a is 0, we move to the left child of v and compute $i = \mathsf{rank}_0(B_v, i)$, where B_v is the bit

vector of v. In the other case, when the *hth* most significant bit of a is 1, we move to the right child of v and compute $i = \mathsf{rank}_1(B_v, i)$ instead. Once we reach a leaf, we return i as the rank of a. The shaded boxes of Figure 2.2 depict the idea for $\mathsf{rank}_a(T, i)$.

The operation $\mathsf{select}_a(B, r)$ requires a bottom-up traversal of T. We descend over T to find the leaf v labeled with a. Once we find v, we move to its parent p. If v is the left child of p, then we perform $r = \mathsf{select}_0(B_p, r)$, where B_p is the bit vector of p. On the other hand, if v is the right child of p, we perform $r = \mathsf{select}_1(B_p, r)$. After updating r, we set $v = p$ and apply the same idea over the new node v. We finish when we reach the root of T. We return the last value of r as the answer for $\mathsf{select}_a(T, r)$.

The time complexity of access, rank_a and select_a is $\mathcal{O}(\log \sigma)$. This complexity is dominated by the number of steps it takes to reach a leaf of T from its root or vice-versa. That number is $\lceil \log \sigma \rceil$, the height of the tree. On the other side, the rank and select queries we perform when visiting nodes can take constant time if we use proper data structures, as we explained in Section 2.2.1.

The representation of T is composed of three elements; the bit vectors, the rank and select data structures and the tree topology. Note that the bit vector lengths in the same tree level add up to exactly n bits as they represent a recursive partition of Σ. Thus, the complete set of bit vectors in T add up to $n\lceil \log \sigma \rceil$ bits. On the other hand, the rank and select data structures we use for querying them require $o(n \log \sigma)$ bits, and the topology of T requires another $\mathcal{O}(w\sigma)$ bits if we encode it using a pointer-based representation. As a consequence, the total space of the wavelet tree is $n \log \sigma + o(n \log \sigma) + \mathcal{O}(w\sigma)$ bits.

Connection with Geometric Data Structures

We can also use the wavelet tree to encode a grid of points. We regard $T[x] = y$ in the wavelet tree T as a point in the coordinate (x, y). This scheme requires the grid to have one point per column as we cannot store different y coordinates in the same x position in T. We solve this problem by storing the points of the same column consecutively in T. In other words, if there are points in the grid with coordinates $(x, y_1), (x, y_2), \ldots, (x, y_k)$, then we have a range $T[i, i + k] = y_1, y_2, \ldots, y_k$. We augment T with a bit vector B that delimits the boundaries between points that belong to different columns. If a column has $c \geq 0$ points, we append the pattern 10^c to B. We augment B with rank and select data structures so we can map positions in T to columns.

We use the properties of the wavelet tree to answer the following queries on the grid:

- $\mathsf{rangecount}(T, x_s, x_e, y_s, y_e)$: number of pairs $(x_i, y_i) \in T$ such that $x_s \leq x_i \leq x_e$, $y_s \leq y_i \leq y_e$
- $\mathsf{rangereport}(T, x_s, x_e, y_s, y_e)$: list with the pairs (x_i, y_i) of $\mathsf{rangecount}$

We can answer $\mathsf{rangecount}$ in $\mathcal{O}(\log \sigma)$ time, and $\mathsf{rangereport}$ in $\mathcal{O}((1 + occ) \log \sigma)$ time. These functions are also useful in the context of sequences. For instance, we can use $\mathsf{rangecount}(T, i, j, a, b)$ to obtain the number of symbols $s \in \Sigma$ in $T[i, j]$ with $a \leq s \leq b$.

Compressed Wavelet Trees

We can further compress a wavelet tree by giving a Huffman shape [122] to its topology. This technique is effective when the alphabet is small. In this version, we first run the Huffman algorithm on the symbols of S and their frequencies (Section 2.1.2) to obtain variable-length codes for the symbols. The idea is to use these codes to produce the shape of T. Suppose that during the wavelet tree's construction we have to build the bit vector B_v of a node v with depth h from an input sequence S_v. If the hth bit from left to right in the Huffman code of $S_v[i]$ is $\mathsf{0}$, then we set $B_v[i] = \mathsf{0}$, and set $B_v[i] = \mathsf{1}$ otherwise. As before, we split S_v into two sequences, S_v^l and S_v^r, but this time we use the value of the hth bit in the Huffman code of $S_v[i]$ to decide if that symbol belongs to S_v^l or S_v^r. After finishing v, we build its left and right subtrees from sequences S_v^l and S_v^r, respectively. We stop expanding a subtree when the alphabet of S_v has only one symbol a, in which case we create a leaf labeled with a.

Unlike the regular wavelet tree algorithm, the symbols in the alphabet of S_v are not equally distributed in S_v^l and S_v^r as the Huffman codes are of variable length. This difference implies that we can create a leaf at any depth of T. In fact, the tree depth of a leaf encoding a symbol $a \in \Sigma$ with a Huffman code of length $|hc(a)| = h$ is $h + 1$.

Reaching the leaf of T labeled with a requires us visiting $|hc(a)|$ internal nodes. The bit vector of each of these internal nodes uses one bit per occurrence of a in S. Therefore, T uses $n_a|hc(a)|$ bits for a, where n_a is the number of occurrences of a in S. If we consider all the characters in Σ, then we have that the total number of bits spent by the bit vectors of T is $\Sigma_a n_a|hc(a)| < n(\mathcal{H}_0(S) + 1)$, exactly the length in bits of the Huffman-compressed sequence. Recall from Section 2.1.2 that the Huffman algorithm reduces the average length of the codes to the statistical entropy of S. If we also consider the space usage of the rank and select data structures along with the topology of T, then the total space usage of this wavelet tree representation is $n(\mathcal{H}_0(S) + 1)(1 + o(1)) + \mathcal{O}(\sigma w)$ bits.

The algorithms for access, rank_a and select_a remain the same. However, querying more frequent symbols in a Huffman-shaped wavelet tree is faster than in the regular version. As explained before, the time complexity of access, rank_a and select_a is dominated by the length of the path we traverse on T to reach the leaf labeled with a. In a regular wavelet tree, that path is always of length $\log \sigma$, regardless of the symbol. In a Huffman-shaped wavelet tree, however, the paths of more frequent symbols are shorter than those that are less frequent. Querying the less frequent symbols, instead, can be slower.

When using a Huffman-shaped tree is not an option, we can still compress T by storing the bit vectors of the internal nodes with the RRR representation (Section 2.2.1). This scheme reduces the overall space usage to $n\mathcal{H}_0(S) + o(n \log \sigma) + O(\sigma w)$ bits. The use of RRR does not change the tree's shape, so the query complexities remain the same[1] as in the uncompressed version of the wavelet tree.

[1] Performing rank in a bit vector compressed with RRR takes $\mathcal{O}(1)$ time.

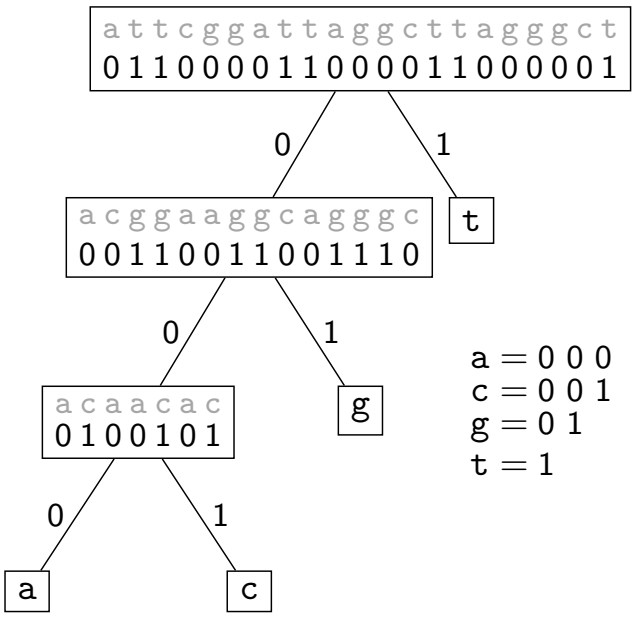

Figure 2.3: Huffman-shaped wavelet tree T for the string `attcggattaggcttagggct` of Figure 2.1. The binary strings to the right of the tree are the Huffman codes of the string's symbols.

2.2.3 Succinct Trees

A *tree* is a hierarchical data abstraction. It consists of a set of n nodes and $n - 1$ edges. Every node has exactly one predecessor (i.e., the parent) and one or more successors (the children). The only node with no predecessor is the root. When a node has no children is called a leaf, and when two or more nodes share the same parent they are siblings.

Trees are widespread in computer science as they adapt well to multiple situations. However, they can potentially use a lot of memory, especially when the information they encode is massive. A simple pointer-based representation, for instance, requires at least $n \log n$ bits of space only to store the topology. This problem has motivated the development of succinct data structures for trees. The techniques developed for this topic are among the most successful CDSs. Nowadays, we have static[2] encodings that can store a tree in $2n + o(n)$ bits of space and answer many navigational queries in constant time.

In this section, we briefly describe three of such succinct encodings for ordinal trees; LOUDS, Balanced Parentheses (BP) and DFUDS. In Table 2.1 we describe the main navigational functions we use in later chapters.

LOUDS

The acronym LOUDS stands for *Level-Order Unary Degree Sequence*. In this encoding, we regard T as bit vector $B[1, 2n + 1]$. The first two positions of B contain the pattern $\mathsf{10}$ to avoid border cases. We store the nodes of T in level-order so that if a node v has c children, then we append the sequence $\mathsf{1}^c\mathsf{0}$ to B. This bit pattern is called the description

[2] They do not allow the insertion, deletion or modification of any node or edge in the tree.

Function	Description
root	Root of the tree
fchild(v)	First child of v, if it exists
lchild(v)	Last child of v, if it exists
nsibling(v)	Next sibling of v, if it exists
psibling(v)	Previous sibling v, if it exists
parent(v)	Parent of v, if it exists
isleaf(v)	Whether v is a leaf
leafrank(v)*	Number of leaves preceeding v, plus 1 if v is a leaf
leafselect(r)*	rth leaf on the tree
internalrank(v)*	Number of internal nodes preceeding v, plus 1 if v is an internal node
internalselect(r)*	rth internal node in the tree
nodemap(v)	An identifier i in $[1, n]$ for v
nodeselect(i)	The node v with identifier i
children(v)	Number of children of v
child(v, r)	The rth child of v from left to right, if it exists
label(v)	The label of v

Table 2.1: Basic navigational operations supported in LOUDS, BP, and DFUDS. The definition of the functions with an $*$ vary depending on the tree encoding. For BP and DFUDS, leafrank(v) and internalrank(v) return the number of leaves and internal nodes preceding v in *pre-oder* (respectively), while in LOUDS they are numbered in *level-oder*. Similarly, internalselect(r) and leafselect(r) return the rth node in pre-order when the tree is in BP or DFUDS, and in level-order when it is in LOUDS.

of v. We identify v with the index where its description starts in B. The LOUDS encoding produces one 0 per distinct node and one 1 per distinct edge. These bits plus the two bits at the beginning of B add up to $2n + 1$ total bits. Figure 2.4B depicts an example of this representation.

If we augment B with the rank and select data structures of Section 2.2.1, then we can answer several navigational queries in constant time.

In general, LOUDS is considered the most simple encoding. It is the one that uses the least space, but also is the most limited in terms of navigational queries. Still, it supports all the operations listed in Table 2.1.

Balanced Parentheses

We can also succinctly store T as a sequence of *balanced parentheses* (BP) encoded as a bit vector $B[1, 2n]$. Every node v in T is represented by a pair of parentheses `(..)` that enclose the encoding of the subtree rooted at v. We identify each node of T with the position in B of its open parenthesis. We build the BP representation by traversing T in pre-order. When we enter the subtree of a node v we append an opening parenthesis `(` to B. Then, when we exit the subtree, we append a closing parenthesis `)`. Figure 2.4C shows an example of BP.

The BP representation supports more navigational queries than LOUDS, but in practice

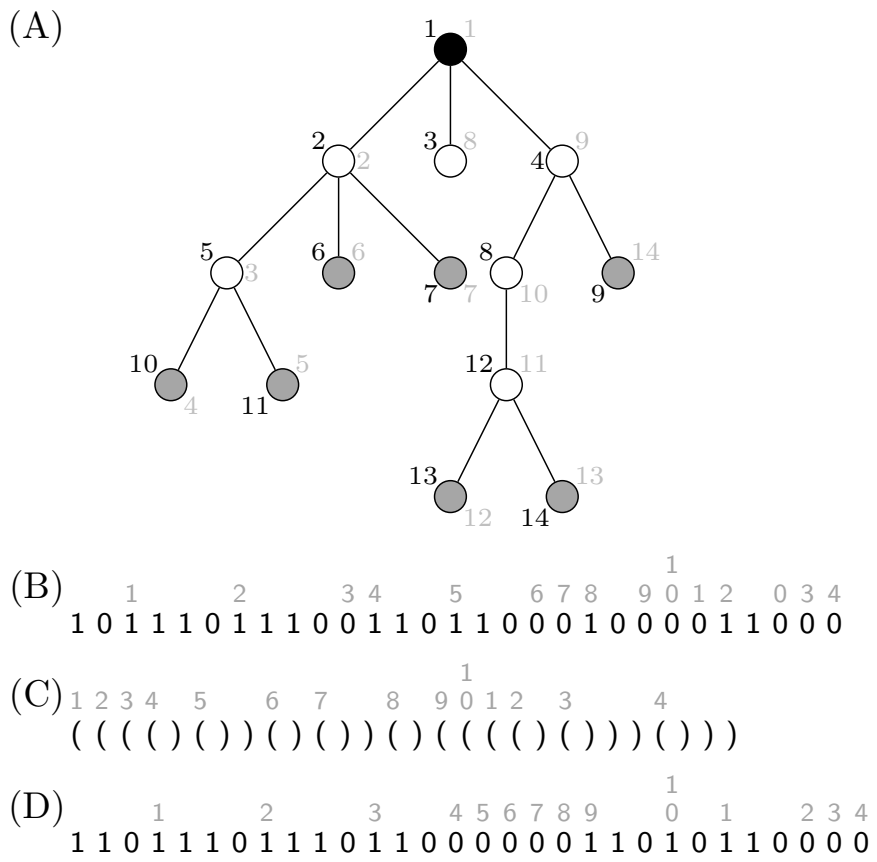

Figure 2.4: Succinct tree encodings. (A) Ordinal tree. The numbers in gray to the right of the nodes are their pre-orders, while the numbers in black to the left are their level-orders. (B) LOUDS representation for the tree of (A). The numbers in gray on top indicate where are located the nodes in the encoding (using their level-orders). (C) BP representation of the tree of (A). The numbers in gray on top indicate where are located the nodes in this encoding (using their preorder position). (D) DFUDS representation of the tree of (A).

it is a bit slower and uses more space. BP resorts to a set of primitives over a data structure called range min-max tree [138]. A simple implementation of this range min-max tree adds $o(n)$ bits on top of B, and its primitives take $\mathcal{O}(\log n)$ time. Consequently, the operations in BP that rely on this data structure also take $\mathcal{O}(\log n)$ time. Other more sophisticated solutions [43] that build on range min-max trees reduce the time complexity of the navigational queries in BP to $\mathcal{O}(\log \log n)$ and still require $o(n)$ bits.

DFUDS

The words DFUDS means *Depth-First Unary Degree*. To build this encoding, we initialize an empty bit vector B and append the sequence $\mathsf{110}$ to it to avoid border cases. Then, we start a pre-order traversal over T, and for every node v, we append the pattern $\mathsf{1}^c\mathsf{0}$ to B, where c is the number of children of v. Similarly to the other representations, the final space usage for B is $2n + 2$ bits.

In DFUDS, the children information of every node v is stored close to v in B, like in LOUDS. This feature simplifies the computation of primary functions like $\mathsf{children}$, child, and

childrank, which are slower to answer in BP in practice. On the other hand, the disposition of the nodes in pre-order allows DFUDS to use the range min-max tree to support more queries than LOUDS. We could consider DFUDS as a hybrid encoding that combines the best aspects of both LOUDS and BP.

2.3 Hashing

2.3.1 Hash Tables

A *hash table* is an associative container that maps a set $\mathcal{K}$ of keys to values. There is a value for every key $k \in \mathcal{K}$ in the table. Keys are unique, but the values of distinct keys can be equal. The keys and the values can be of any type, like strings, integers, or floats. The basic operations a hash table can support are:

- insert(k, v) : insert key k associated with value v
- delete(k) : delete key k and its value
- find(k) : returns true if key k exists as key in the hash table
- retrieve(k) : returns the value associated with key k

We can implement a hash table by first choosing a parameter m and then building an array $A[1, m]$ along with a hash function $h : \mathcal{U} \to [1, m]$. The universe $\mathcal{U}$ contains all the possible keys we can see in $\mathcal{K}$, and h maps its elements to slots in A. The idea is to store the value of k in $A[h(k)]$. Ideally, each pair of distinct keys $k, k' \in \mathcal{U}$ should map to different positions of A. However, this is not always the case. When $h(k) = h(k')$, we have a *collision*.

When the set $\mathcal{K}$ is unknown, and $\mathcal{U}$'s size is greater than m, ensuring no collisions is impossible. Nevertheless, we can select a good hash function to reduce their probability. Intuitively, a "good function" ensures that each key is equally likely to map to any of the slots of A. A typical approach to achieve this property is by randomly selecting a hash function that does not depend on the keys of $\mathcal{K}$. This concept is called *universal hashing*.

Let $\mathcal{H}$ be a set of hash functions that map the keys in $\mathcal{U}$ to $[1, m]$. Such set is said to be universal if by randomly picking a function $h \in \mathcal{H}$, it holds $\Pr(h(x) = h(y)) \leq m^{-1}$ for any x, y. A simple universal set $\mathcal{H}_{ab}$ (or family) for integers is that of

$$h_{ab}(x) = ((ax + b) \mod p) \bmod m,$$

where $0 < a < p$, $0 \leq b < p$, and p is a prime number as large as the range of keys. The family is conformed by all the possible values of a and b.

There are several techniques to resolve collisions. Here we briefly review the most popular ones.

Chaining

Every slot $A[j]$ points to a linked list that contains all the pairs (k, v) with $h(k) = j$. When there is no key in the hash table mapped to $A[j]$, its pointer is null. During the insertion of (k, v) into the hash table, we create a new linked list L and store (k, v) in its head if $A[h(k)]$

is null. In addition, we store in $A[h(k)]$ a pointer to L's head. On the other hand, if $A[h(x)]$ already stores a pointer to some linked list L, we create a new entry in L for (k, v), provided k is not already in the list.

The performance of the hash table will depend on the length of the linked lists; if k maps to a slot $A[j]$ with collisions, then we have to compare k against the keys in list L to which $A[j]$ points. Let us denote with n the total number of elements in all the liked lists and $\alpha = n/m$ the load factor of the hash table. Assuming we selected a hash function that uniformly maps the keys to the slots, the average time for scanning the linked lists is α. If the hash computation takes constant time, then the average case for operating the table is $\mathcal{O}(1 + \alpha)$. That value can be $\mathcal{O}(1)$ if n is proportional to m.

Open Addressing

In *open addressing*, we store the pairs (k, v) in the hash table, not in linked lists. When a collision occurs, we repeatedly probe other positions in the table until finding an available slot. The probing mechanism must be reproducible in the sense that we have to replicate the same sequence of visited positions when searching for the key again. Therefore, the slots we probe must depend on the key value. The advantage of open addressing compared to chaining is that we do not require extra pointers. It is also potentially more cache-friendly if the probing makes the colliding keys to be stored contiguously in the table.

The procedure for inserting a pair (k, v) is simple. Let $h(k, i)$ be a hash function with probing step i. We compute first $j = h(k, 0)$. If $A[j]$ is already occupied, then we compute $j' = h(k, 1)$ and check if position $A[j']$ is available. We continue increasing i until finding an empty slot. In other types of hashing schemes that use open addressing, if the slot for k is already occupied, we do not find another one. Instead, we swap (k, v) with the pair in its slot and find a new slot for the evicted pair. The condition for swapping pairs might vary depending on the technique.

If we consider that each possible probed cell is equally probable, then the expected number of probes when searching for a key is $1/(1-\alpha)$, where α is the load factor (note that it must hold $\alpha < 1$ with open addressing). If we fix α to some threshold, then searching for a key in the hash table takes $\mathcal{O}(1)$ time. For instance, if the load factor is 0.9, then the expected number of probes is $1/(1 - 0.9) = 10$. It is customary to maintain the load factor in about 0.5 so the expected number of visited slots is just 2. When the table exceeds this load factor, we increase the table size and rehash all its elements.

There are different ways of defining the probing step. Here we briefly review the classical methods.

Linear Probing We define an auxiliary hash function $h' : \mathcal{U} \to [1, m]$, and perform

$$h(k, i) = (h'(k) + i) \bmod m.$$

Linear probing is simple to implement and makes the probing cache-friendly. Nevertheless, it tends to create long runs of occupied slots, thus requiring smaller α values (i.e., bigger tables) to maintain the same amount of probes. Let us see the effect of maintaining a high

value for α; the expected number of probes in linear probing is

$$\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right).$$

If we set α to 0.5, we expect to perform 2.5 probes, which is not so far from the random assumption (2 probes). However, if we set α to 0.9, the expected number of probes increases to 50.5, which is much more than in the random assumption (10 probes).

Quadratic Probing In *quadratic probing*, the slots are computed as

$$h(k,i) = (h'(k) + x \cdot i + y \cdot i^2) \bmod m,$$

where h' is an auxiliary function and x and y are constant values. The performance of quadratic probing is usually better than linear probing, but x, y and m must be constrained. Besides, the keys also tend to group in runs in the table, although in a milder way.

Double Hashing *Double hashing* selects two auxiliary functions $h_1(k)$ and $h_2(k)$, and obtains the hash value as

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

The performance of double hashing is better than quadratic and linear probing. Its main feature is that both the initial key's slot and the subsequent probed positions depend only on k. Thus, even if two keys collide on the same initial slot, there is a high chance that their probing steps will be different. This property reduces the table's clustering, and hence, the search time.

Robin Hood Hashing In *Robin Hood hashing* [29] each cell in the hash table, in addition to the key-value pair, stores an offset that indicates how far is the key from its original slot. Let i be the probing step in the insertion of (k,v), and let $A[j]$ be the slot of the probing step. Further, let (k',v') be the pair stored in $A[j]$ and o the offset for k'. If $i > o$, then k is farther away from its original slot than k'. In such a case, we swap the elements; we store (k,v) in $A[j]$ with offset $o = i$ and find a new position for (k',v') from probing step $i = o+1$. As we find a new position for the pair, we continuously swap the key to be inserted when we reach a slot with an offset smaller than i. Although it maintains the same average search time, Robin Hood probing reduces the variance, as it tends to equalize the search costs of the different keys.

Cuckoo Hashing In *Cuckoo hashing* [149] we select two independent hash functions h_1 and h_2, and two hash tables[3], H_1 and H_2, with the same number of slots. We use h_1 for assigning slots in H_1 and h_2 for assigning slots in H_2. When inserting a new pair (k,v) into the data structure, we store it in whichever of its slots, $H_1[h_1(k)]$ or $H_2[h_2(k)]$, is empty. If both are occupied, we swap (k,v) with the pair in one of its slots. Suppose we

[3]Some variants consider one table subdivided into two segments.

arbitrarily chose to swap (k, v) with $H_1[h_1(k)] = (k', v')$. Now we insert (k', v') in its other slot $H_2[h_2(k')]$, and if that slot is also occupied, then we swap (k', v') with the pair there and repeat the process for the evicted pair. We continue until we find an available slot. It is customary to select a threshold for the number of times we can swap pairs in the hash tables. When we exceed that threshold, we trigger a rehashing process for all the keys. One of the advantages of cuckoo probing is that the lookup operation guarantees $\mathcal{O}(1)$ worst-case as every key has at most two places where can be stored in the hash table. The disadvantage, on the other hand, is that it is not cache-friendly. The inspection of the two slots of a key produces two cache misses.

2.3.2 Rolling Hashing

Rolling hashing [91] is a linear-time technique to compute integer values (or fingerprints) for the substrings of length p in a string $S[1, n]$. It was originally developed for pattern matching, but it also generalizes to other related problems, as we will see in Section 3.5.1.

We first choose a parameter $p \leq n$ and define a polynomial hash function

$$h(\langle s_1, s_2, \ldots, s_{p-1}, s_p \rangle) = \left(\sum_{i=1}^{p} s_{p-i+1} x^{i-1} \right) \bmod q,$$

where q is a prime number and x is an arbitrary integer greater than one. This function maps a string of length p to an integer value in the range $[0, q-1]$. We produce the fingerprints in $\mathcal{O}(n)$ time by evaluating h on the $n-p+1$ substring of length p in S, but without evaluating the complete polynomial $n-p+1$ times. We begin the process by computing $h(S[1, p])$ using Horner's rule. Subsequently, we slide a window of length p over S to obtain fingerprints for the rest of substrings of length p. We compute the value of a window $S[i, j]$, with $j-i+1 = p$, by updating the fingerprint of the previous window $S[i-1, j-1]$. For this purpose, we use the formula

$$h(S[i, j]) = ((h(S[i-1, j-1]) - S[i-1]x^{p-1})x + S[j]) \bmod q,$$

where $h(S[i-1, j-1])$ is the fingerprint of $S[i-1, j-1]$. Notice this formula is equivalent to $h(S[i, j])$, but we are not evaluating the complete polynomial, as we anticipated. The advantage of rolling hashing is that, after obtaining $h(S[1, p])$ in $\Theta(p)$ time, we perform $n - p + 1$ constant-time updates to calculate the fingerprints of the other substrings of S.

Extending rolling hashing for pattern matching is simple. Let $P[1, p]$ be the pattern we have to search for in S. We choose a polynomial function for strings of length p and a prime number q to build the hash function h. Subsequently, we compute the fingerprint for P as $f_P = h(P[1, p])$, and then we start to roll h over S using the mechanism described in the previous paragraph. Every time we reach a substring $S[i, j]$ such that $h(S[i, j]) = f_p$, we report i as a match if $S[i, j]$ equals P.

Checking $S[i, j]$ against P when they have the same fingerprint is necessary as h can return the same value for different strings (i.e., a collision). This situation occurs when the evaluation of $S[i, j]$ and P in the polynomial of h yield two values (say a and b) that are congruent modulo q ($a \equiv b \mod q$). On the other hand, if $S[i, j]$ and P have different

fingerprints, we are sure they do not have the same sequence, so the check is unnecessary. When h distributes randomly on $[0, q-1]$, the expected number of times we compare P against a substring $S[i, j]$ during the scan of S is upper bounded by $v + n/q$, where v is the number of true matches of P and n/q is the expected number of unsuccessful matches. Thus, the pattern matching algorithm runs in $\mathcal{O}(n + p(v + n/q))$ time.

We can reduce the probability for two sequences to have the same fingerprint by choosing a large enough random prime number q. Also, a good decision is to select x and q such that the product xq fits in a computer word.

2.3.3 Bloom Filters

A *bloom filter* [18] is a lossy probabilistic data structure that encodes a set Q in succinct space. The most basic operations it supports are inserting an element q into Q and checking if an input q' is already present. The fields where this data structure has applications are manifold, from Genomics to networks and databases. In the particular case of Genomics, it has been used to compute the q-grams (also known as kmers) of DNA sequences [179, 177, 151].

Bloom filters are fast and space-efficient, so they are a good choice when the set Q is massive, and the memory requirements of traditional hash tables are too high. Nevertheless, they have the disadvantage of being lossy. When the data structure tells us that a given element q does not belong to Q, the result is always correct. However, when it tells us that q does belong, there is a slight chance that this is not true. In other words, we have false positives.

A bloom filter has two components; a bit vector $B[1, m]$ and a set of x hash functions $\mathcal{H} = \{h_1, h_2, \ldots, h_{x-1}, h_x\}$ that map an element q to x integer values within the range $[1, m]$. To insert $q \in Q$ into the data structure, we first hash it with each function $h_i \in \mathcal{H}$, and then set every $B[h_i(q)]$ to $\mathtt{1}$. Later, if we need to check if q is in the filter, we evaluate it with all the $\mathcal{H}$'s functions again, and return true if every cell $B[h_i(q)]$ is set to $\mathtt{1}$, or false otherwise. The false-positive problem arises when the B's cells associated with q were independently set to $\mathtt{1}$ by other elements of Q, but q itself has not been inserted yet. Thus, if we test for q, the data structure will report it as present when it is not. Notice also that deleting q from the bloom filter is not safe as the positions in B for different elements of Q might overlap. As a consequence, if we flip the bits of q in B to $\mathtt{0}$, we might accidentally delete another element of the filter.

The only parameters for building the bloom filter are m, the size of B, and $\mathcal{H}$. The probability of false positives will depend on the values we chose for these parameters. If we assume that the hash functions are independent and that they uniformly map the positions in B, then the probability for $B[i]$ to be $\mathtt{0}$ after inserting n elements into the filter is

$$\left(1 - \frac{1}{m}\right)^{nx}.$$

To account for the probability of false positives during the membership test, we need to

consider the chance of x independent positions in B to be set to 1. This value is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nx}\right)^{x}.$$

The formula above is approximately $(1 - e^{-nx/m})^x$. A typical method for reducing the probability of false positive is to chose $x = \ln(n) \cdot m/n$ different hash functions as this number minimizes the formula.

2.3.4 Document Similarity

Andrei Broder [26] proposed an space-efficient method to compute the similarity between two documents A and B. This technique, popularly known as *MinHash*, has been widely used in the detection of highly similar web pages, images, and genomic analyses [148, 99].

Before explaining the method, we give some definitions. Let us denote $K(X, k)$ the set of all the distinct substrings of length k in a document X, where k is a parameter. The distance between A and B is then computed as

$$J(A, B) = \frac{|K(A, k) \cap K(B, k)|}{|K(A, k) \cup K(B, k)|}.$$

This value is the Jaccard coefficient for two sets. Broder, however, showed that we can obtain an unbiased estimate of $J(A, B)$ if we use a random sample of $K(A, k)$ and $K(B, k)$. This observation is particularly useful when we need to compute the distance between huge documents under memory constraints.

He uses a concept called the *sketch* of a document X. Assume we give numeric identifiers to the elements in $K(X, k)$ from a totally ordered universe U of size $|\Sigma^k|$. In addition, we need a permutation $\pi : U \to U$ chosen uniformly at random. The sketch of X, denoted here as $S(X, s, k)$, is then the set with the s smallest elements of $\pi(K(X, k))$, where s is a parameter. Once we compute the sketch of A and B, we can estimate $J(A, B)$ as

$$\frac{|S(A, s, k) \cap S(B, s, k)|}{|S(A, s, k) \cup S(B, s, k)|}.$$

Intuitively, $S(X, s, k)$ are the s smallest permuted identifiers of the k-substrings in X. Still, when k is too large, the numeric identifiers for U might not fit the computer word. We can solve this problem by using a hash function h that maps the k-substrings to a range $[1, 2^l - 1]$ where l is a suitable length smaller or equal than the machine word's length. A good choice in this case would be to use the rolling hashing idea of Section 2.3.2 to compute the new identifiers. We need to carefully select the parameters for the hash function so that the number of collisions does not affect the outcome of the Jaccard estimation.

Other Methods

Winnowing [166] and *Minimizers* [158] are other similar techniques for computing document similarity. Winnowing was developed to identify plagiarism while Minimizers to compute

suffix-prefix overlaps between DNA sequences. Although these algorithms serve different purposes, they perform almost the same steps. In both cases, we receive an input document (or sequence) X and parameters k and w. The idea is to scan the text from left to right and select the k-substring with the smallest identifier in each window of w consecutive k-substrings. The distinct sequences selected by the algorithm are called the minimizers of X. As before, we use a random hash function to assign identifiers to the k-substrings of X. For each minimizer, we store its sequence and a list with its occurrences in X.

Winnowing and Minimizers produce a text sampling with the following properties; first, every pair of sampled substrings are at most w characters apart. Second, if two different substrings of length at least $k + w + 1$ are equal, then they must have at least one sampled k-sequence in common. In recent years, the Minimizers technique has gained popularity in Bioinformatics as a fast preprocessing step for comparing long DNA strings [13, 154, 90, 112]. The rationale is that if two strings are somehow similar, they must share at least some minimizers. In Section 4.4, we review some of these methods.

Chapter 3

Indexing and Compressing Text

This chapter reviews the state-of-the-art techniques to index and compress *texts*. We consider a text to be a string $S[1, n]$ over alphabet $\Sigma = [1, \sigma]$ that carries information. *Indexing* consists of augmenting S with extra data structures so we can extract that information efficiently. In situations where S is large, and the space overhead of the index is considerable, we can use a *succinct self-index*. This data structure maintains S in compressed form and answers the queries by operating over the text without decompressing it. We can further extend self-indexes to store labeled graphs, another way of encoding text. Labeled graphs are helpful for genomic applications as they are a more accurate DNA representation.

We begin this chapter by describing the classical text indexes (Section 3.1). In Section 3.2, we review the most popular techniques for compressing text. In Section 3.3, we explain how to combine text compression and indexing to develop self-indexes. Section 3.4 extends the ideas of self-indexing to represent labeled graphs. Finally, in Section 3.5, we review algorithms for indexing text that exploit repetitions. These concepts are fundamental for the ideas we develop in the following chapters.

For convenience, we usually consider the last symbol $S[n] = \$$ to be lexicographically smaller than any other character in S.

3.1 Classical Indexes

Text indexing consists of building a data structure I_S (a.k.a., the *index*) from S to perform queries. This technique is useful when the input document is large and performing linear searches is too slow for practical purposes. Unfortunately, most text indexes are static; if we modify the document, the index becomes invalid, and we have to build it from scratch.

The most basic operations I_S can answer are:

- $\mathsf{count}(I_S, P)$: number of occurrences of the pattern P in S
- $\mathsf{locate}(I_S, P)$: report every $j \in [1, |S| - |P| + 1]$ such that $S[j, j + |P| - 1]$ is an occurrence of P

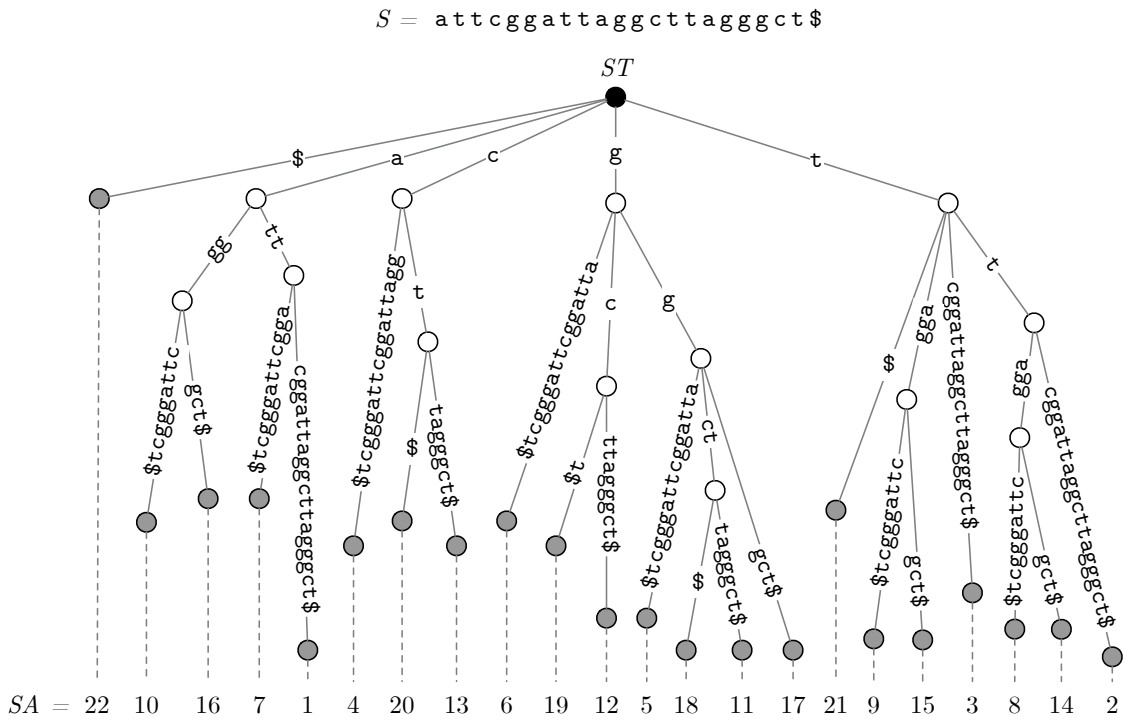

Figure 3.1: Classical text indexes for a string S. The suffix tree (ST) and the suffix array (SA). The dashed lines map a branch in the ST with its corresponding position in the SA.

In this section, we describe two basic techniques for text indexing; the *suffix array* and the *suffix tree*. Most of the other indexing approaches we describe in later sections try to resemble their functionality while reducing the space usage.

3.1.1 Suffix Array

The suffix array [78, 125] is an array $SA[1, n]$ that stores the positions of the suffixes of S according their lexicographical order. Thus, it holds that $S[SA[1], n]$ is lexicographically smaller than $S[SA[2], n]$, $S[SA[2], n]$ is smaller than $S[SA[3], n]$, and so on. An example of this structure is shown in Figure 3.1.

A key property of the suffix array is that if a pattern P occurs several times in S, then the suffixes prefixed by P form a contiguous range in SA. We find this range by performing two binary searches over SA. In the first one, we obtain the position $SA[x]$ such that suffix $S[SA[x]..]$ is prefixed by P, and the suffix $S[SA[x-1]..]$ is not. In the second binary search, we obtain the position $y \geq x$ such that $S[SA[y]..]$ is prefixed by P, and $S[SA[y+1]..]$ is not. The resulting segment $SA[x, y]$ will contain all the suffixes of S prefixed by P. Note that every time we visit a new position $SA[i]$ during the binary search, we need to compare P against the prefix of length $|P|$ in $S[SA[i]..]$. Therefore, each binary search takes $\mathcal{O}(|P| \log n)$ time. Once we obtain $SA[x, y]$, answering count reduces to returning $y - x + 1$, and answering locate reduces to reporting the indexes in $SA[x, y]$. The time complexity of count is thus $\mathcal{O}(|P| \log n)$, and the time complexity of locate is $\mathcal{O}(|P| \log n + occ)$, where occ is the number of occurrences of P in S.

3.1.2 Suffix Tree

Consider a *trie* T built with the suffixes of S. For every $S[i, n]$, there is a path $U = v_1, v_2, \ldots, v_k$ of length $k = n - i + 2$ in T, where v_1 is the root and v_k is a leaf (there is one leaf for every suffix of S). Further, each edge (v_j, v_{j+1}) in U is labeled with a symbol in Σ, and concatenating the edge labels from v_1 to v_k produces $S[i, n]$. We build T such that if two or more suffixes of S have the same j-prefix, then their paths in the trie share the first $j+1$ nodes. This feature produces a cardinal tree T; edges originating from the same internal node cannot have the same labels.

We can compact T as follows. For every path $U = v_i, \ldots, v_j$ where v_i and v_j are the only nodes with more than one children, we remove the subpath $U' = v_{i+1}, \ldots, v_{j-1}$ and connect v_i with v_j by an edge (v_i, v_j) labeled with the concatenation of the labels in U'. The result of this procedure is a compact trie T' of n leaves and less than n internal nodes called the *suffix tree* of S [188].

Let $l(v)$ be the string spelled by the path in T' starting in the root and ending in v. One of the main features of the suffix tree is that the sequence in $X = l(v)$ spelled by every internal node v is *right maximal*; the frequency of X in S is greater than the frequency of any of its right extensions Xa, with $a \in \Sigma$. Further, the children of the internal nodes are sorted according the lexicographical order of the edge labels. Therefore, the distinct suffixes of S are also encoded in lexicographical order. Finally, every leaf v_i in the suffix tree stores the index i of its corresponding suffix $l(v_i) = S[i, n]$. An example of a suffix tree is shown in Figure 3.1A.

We can extend the suffix tree to index a string collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ instead of just S. This variation is known as the *generalized suffix tree* [14]. The main difference is that every leaf v_i now stores a list of positions, not just one value. Each element in the list of v_i is a pair (j, x) that indicates that $l(v_i)$ is the suffix $S_j[x..]$ in $S_j \in \mathcal{S}$. For the construction of the index, we use the string $S = S_1\$S_2\$ \ldots S_m\$$ that represents the concatenation of $\mathcal{S}$. The $\$$ characters mark the boundaries between the strings, and are lexicographically smaller than any other symbol in Σ. We can use any standard algorithm for building T' from S [182]. The only caveat is that if the label of an edge (u, v) in T' contains a $\$$ symbol, v becomes a leaf.

It is easy to see that we can answer count and locate for a pattern P in $\mathcal{O}(|P| \log \sigma + occ)$ time using T'. We have to find the node v with the lowest tree depth such that $l(v)$ is prefixed by P, and then visit the leaves under the subtree of v to access the occurrences of P in S. To find v, we start a descent over T' from the root. Every time we reach a new internal node v', we perform a binary search over the labels of the edges that connect v' with its children. The aim is to find the child u such the edge (v', u) is prefixed by symbol $P[|\mathsf{label}(v')| + 1]$. We continue the tree descent through u if $l(u)$ is a prefix of P or if P is a prefix of $l(u)$. In the latter case, we stop the descent as u is indeed v. We visit no more than $|P|$ internal nodes, and the binary search in each of them takes $\mathcal{O}(\log \sigma)$ time (every node has at most σ children). Further, there are occ leaves under v, and we have to visit no more than occ internal nodes to reach them as T' is compacted.

We can augment the suffix tree with extra edges to support string queries other than count and locate. A *suffix link* (u, v) from node u to node v occurs if $l(u) = aA$ and $l(v) = A$,

where A is a string and $a \in \Sigma$. Conversely, a *Weiner link* (u, v, a) from u to v, and labeled with $a \in \Sigma$, exists if $l(u) = A$ and $l(v) = aA$. Additionally, a Weiner link is said to be *implicit* if aA exists in S but it is not right maximal. That is, there is no node v' in T' such that $l(v') = aA$.

An important drawback with the suffix tree is its high space consumption. It requires $\Theta(n \log n)$ bits on top of S, and in practice uses about 10 times the size of S [103]. In the following sections, we review representations to encode fully-functional suffix trees in succinct space.

3.2 Text Compression

In Section 2.1, we explained how to compress strings when their symbol frequencies are uneven. Still, that is not the only way. We can also reduce space usage when the input text is *repetitive*. That is, it is composed of highly similar strings. The general idea is that if we have k repetitive documents, we store one of them explicitly as a reference and store the others as edits of the reference.

Repetitiveness is another prominent feature of DNA collections. The concatenation of thousands of individual genomes might require several TBs, but their differences are usually less than 1%. We can then reduce space usage significantly if we exploit the repetitions. Genomic projects do not store individual genomes explicitly. Instead, they produce one reference genome and store the others as edits, using the same idea we explained in the previous paragraph.

One might think that if a text is repetitive, it is also statistically compressible. Nevertheless, this is not the case. Consider, for instance, an incompressible text $S[1, n]$ where all the symbols have the same frequency. The statistical entropy $\mathcal{H}_0(S)$ equals the worst-case entropy, so S requires $n \log \sigma$ bits of space. Now consider the strings $S' = S^t$ that represents the concatenation of t copies of S. The relative symbol frequencies are the same in both S and S', so it holds $\mathcal{H}_0(S) = \mathcal{H}_0(S') = \log \sigma$. As a consequence, S' requires $tn \log \sigma$ bits of space. On the other hand, we can store S' as a tuple (t, S) to indicate that the string is the concatenation of t copies of S. This arrangement uses slightly more space than $n \log \sigma$ bits, but it uses much less than $tn \log \sigma$ when t and n are large.

Another relevant topic closely related to repetitive text is that of *kernels* [71]. A kernel is an abstract set of atomic elements that we can use to represent the text. A simple analogy for this concept is that if we regard the text as a LEGO construction, then its kernel is the set of LEGO pieces. When the text is repetitive, we can produce it with a few concatenations of strings from a small kernel. We can exploit this fact and use the kernel not just to reduce space usage but also to boost string analyses. For instance, it has been shown in the past that this idea can speed up pattern matching in large genomic databases [167, 187, 45].

We will describe now two compression schemes that capture the repetitive patterns in the text. We use them in later chapters of the thesis to process DNA sequences.

As in the previous section, we consider the string $S[1, n]$ over the alphabet $\Sigma = [1, \sigma]$ to

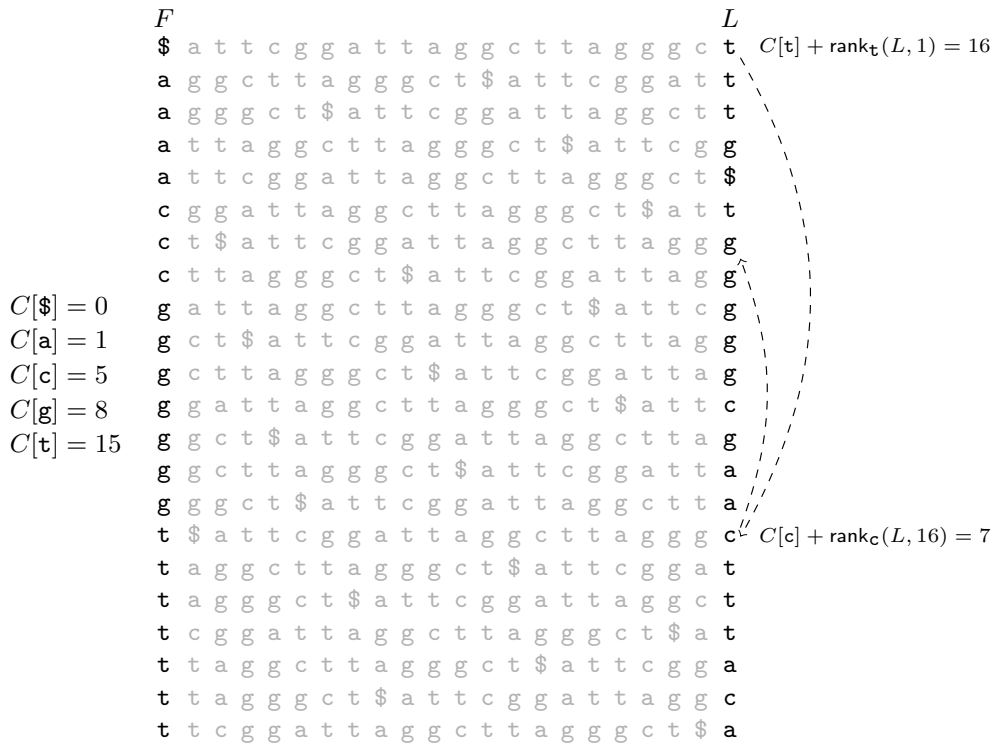

Figure 3.2: Matrix M with the cyclic shifts of the string `attcggattaggcttagggct` of Figure 2.1. The F and L columns are shown in black. The column to the left of F depicts its representation as the C array. The dashed lines to the right of L show two LF steps.

be the input text. The symbol $\$ = \Sigma[1]$ is used as a terminator symbol, and it is mapped to the smallest character in Σ.

3.2.1 The Burrows-Wheeler Transform

The *Burrows-Wheeler transform* (BWT) [27] is a *reversible* string transformation. The traditional way of explaining it is as follows: we generate the n cyclic shifts of S and put them in lexicographical order in a matrix M (every row is a specific shift). The BWT of S is then the array L resulting from extracting the last column of M. Another way to understand the BWT is in terms of the suffix array. Each $L[i] = S[SA[i] - 1]$ is the symbol that precedes the suffix $S[SA[i], n]$. For technical reasons, S is considered to be circular, which means that $L[1] = S[n]$.

To reverse the BWT, we use the so-called LF function. Given an input position $L[j]$ that maps some symbol $S[i]$, $\mathsf{LF}(j) = j'$ returns the index j' such that $L[j'] = S[i-1]$ maps the preceding symbol of $S[i]$. This procedure allows us to spell the sequence of S in reverse text order (from right to left) directly from L; we iteratively apply LF from $L[1]$, the symbol to the left of $S[n] = \$$, and stop when we reach $L[j] = \$$.

To implement LF, we require the first column of M, popularly denoted as F. We logically divide F into σ runs of symbols, or *buckets*, where all the rows in M starting with $b \in \Sigma$ belong to bucket b. We represent F as an array $C[1, \sigma]$, where $C[b]$ stores the number of

rows in M that are lexicographically smaller than any row of bucket b. Now we can compute $\mathsf{LF}(j) = j'$ as $C[a] + \mathsf{rank}_a(L, j) = j'$, where a is the symbol at $L[j]$. The formula works because the occurrences of a in the BWT are sorted according the lexicographical rank of their right contexts. Hence, if $L[j] = a$ has rank r, then $S[i, n]$ is the *rth* suffix prefixed with a in S. To obtain the rank j' of $S[i, n]$ among the other suffixes of S, we add r and $C[a]$. Due to the definition of the BWT, we know that $L[j']$ is the symbol that precedes $S[i, n]$ in the text.

If necessary, we can also implement the inverse function LF^{-1}. In other words, given the position $L[j] = S[i]$, $\mathsf{LF}^{-1}(j) = j'$ returns the index j' such that $L[j']$ maps $S[i + 1]$. We implement LF^{-1} as $\mathsf{select}_a(L, j - C[a])$, where a is the bucket of F where $L[j]$ lies [109].

If we use the wavelet tree of Section 2.2.2, then we can encode L using $n(\mathcal{H}_0(L) + 1)(1 + o(1)) + O(\sigma w)$ bits of space, and support the operations rank_a and select_a in $\mathcal{O}(\log \sigma)$ time.

High-Order Compression

The lexicographical sorting of the cyclic shifts causes the symbols in S with similar right contexts to be grouped in L. All the characters that precede the occurrences of a pattern P of length k (for any k) will be stored in one single block $L[i, j]$. If the same character always precedes P in S, then $L[i, j]$ will be an equal-symbol run. If not, $L[i, j]$ probably contains few distinct symbols anyway.

We can use run-length encoding to compress L as it has few equal-symbol runs compared to S. Still, this is not the only way of reducing space usage. Note that , because $L[i, j]$ has few distinct symbols, its $\mathcal{H}_0$ value is small. We can exploit this fact and partition L so that every distinct block $L[i, j]$ stores the preceding symbols of a specific pattern $P \in \Sigma^k$. Thus, by independently applying zeroth-order compression to each $L[i, j]$, we can store S into its kth-order entropy.

Note the definition of $L[i, j]$ is similar to that of the string S_C in the formula of $\mathcal{H}_k(S)$ in Section 2.1.1 (P and C have the same meaning in this context). The only difference is that $L[i, j]$ has the left contexts of P while S_C has the right contexts. We can modify the $\mathcal{H}_k$ formula to make it symmetric[1]; it does not matter if we consider the symbols to the left or right of P; we obtain an equivalent entropy value. This result demonstrates that by compressing each $L[i, j]$ to its zeroth-order entropy, we achieve high-order compression for S. Recall from Section 2.1.1 that we can obtain space close to $n\mathcal{H}_k(S)$ if we give different codes to the symbols depending on the distinct sequences of length k that precede them on S.

We can still support rank in $\mathcal{O}(\log \sigma)$ time if we include a table with the precomputed ranks of the symbols before every block. It is also necessary to include a bitmap that marks the starting positions of the blocks in L. By choosing a k small enough such that $k + 1 < \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$, the space usage of this scheme can be as little as $nH_k(S) + o((H_k(S) + 1)n)$ bits [127].

[1] See Section 11.3.2 in the book *Compact Data Structures: A Practical Approach* [135] to see how to obtain a symmetric formula for $\mathcal{H}_k$.

The BWT of a String Collection

Ferragina and Venturini [66] were one of the first to consider the BWT of a string collection. Given a multiset $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of lexicographically sorted string, they build L from $S = S_1\$_1 S_2\$_2 \ldots S_m\$_m\#$, the concatenation of $\mathcal{S}$. The symbols $\$$ and $\#$ are special characters lexicographically smaller (respectively, greater) than any symbol in Σ.

The lexicographical ordering of $\mathcal{S}$ allows us to simulate in L circular scans of the strings. More specifically, given that we know the position $L[j]$ storing the preceding symbol of $\$_{i-1}S_i$, we can jump in $\mathcal{O}(1)$ time to the position $L[j']$ storing the preceding symbol of $\$_i S_{i+1}$. The indexes j and j' are within the range $[1, m]$ as both precede $\$$ characters. Recall that $\$$ is the smallest value in Σ and there are m copies of it in S. The key observation of Ferragina and Venturini was that the lexicographical sorting of $\mathcal{S}$ produces $\$_{i-1}S_i$ and $\$_i S_{i+1}$ to be contiguous in the suffix array of S. Therefore, it holds $j' = j + 1$.

Mantaci et al. [126] proposed a BWT variation regarding $\mathcal{S}$ as a multiset of primitive strings. That is, no $S_i \in \mathcal{S}$ can be obtained by concatenating two or more copies of another $S_j \in \mathcal{S}$. Their scheme yields a transform of circular strings; if $L[j]$ maps to $S_i[1]$, then $L[\mathsf{LF}(j)]$ maps to the end of S_i. This feature makes S_i to be independent of the other elements in $\mathcal{S}$, meaning that we cannot reach a character of S_i by continuously applying LF from a BWT position that maps a character of S_{i+1}.

To build L, they consider a special string ordering called $<_\omega$. Before explaining this concept, we need to give some basic definitions. Let X^k be a string X concatenated k times (i.e., X^k is a power of X), and let X^ω be the infinite concatenation of X. Additionally, $root(X) = W$ denotes a unique primitive word that we can use to rewrite $X = W^k$. The operator $exp(X) = k$ is the exponent of W. We also lift the operator $<_{lex}$ to refer to the lexicographical order of the strings.

Given two string A and B over the alphabet Σ^*, the $<_\omega$ order is described as

$$A \preceq_\omega B \iff \begin{cases} exp(A) \leq exp(B) & \text{if } root(A) = root(B) \\ A^w <_{lex} B^w & \text{otherwise.} \end{cases}$$

Mantaci et al. build a list $\mathcal{C}$ with all the cyclic shifts of $\mathcal{S}$ and sort the list in $<_\omega$ order. Subsequently, they produce L by concatenating the last symbols in the sorted cyclic shifts without changing their relative orders. To recognize the string boundaries in L, they also include a sparse bit vector that marks every $L[j]$ mapping to the first symbol of a string. They called the resulting L the *extended* BWT (eBWT). Recently, Bannai et al. [6] proposed a linear-time algorithm for building the bijective BWT that can be used to compute the eBWT in linear time.

The algorithm of Bauer et al. [7] produces a relaxed BWT, usually denoted as the BCR BWT. In their variation, the order in L for symbols preceded by equal suffixes of $\mathcal{S}$ depends on the disposition of these suffixes in the collection. For instance, suppose there are three suffixes $aA\$_x$, $bA\$_y$ and $cA\$_z$ in $\mathcal{S}$, with $a, b, c \in \Sigma$ and $x < y < z$. The three of them end with the same sequence $A\$$. In the BCR BWT, we will have a range in $L[i, j] = (a, b, c)$. This property does not necessarily holds in the other variations. In the BWT of Ferragina

and Venturini, the order of a, b and c will depend on the string S_{x+1}, S_{y+1}, and S_{z+1}, and in the eBWT, the order will depend on S_x^ω, S_y^ω, and S_z^ω. This relaxation in the model allowed Bauer et al. to develop semi-external algorithms to construct the BCR BWT efficiently. These algorithms build L incrementally. They start by defining a partial version of L for the characters that precede the suffixes of length 1 in $\mathcal{S}$. Then, in every iteration i, they insert into L the symbols that precede the suffixes of size i. Other analogous ideas were also developed [111, 117, 118, 20].

Bauer et al. [7] also postulated that we could reduce the number of equal-symbol runs in the BCR BWT if we sort $\mathcal{S}$ in colexicographical order[2] first. Similarly, Bentley et al. [12] proposed a linear-time algorithm for sorting $\mathcal{S}$ that reduces the number of BWT runs by an $\Omega(\log_\sigma m)$ factor. A more recent work [68] explored the idea of guiding the ordering of $\mathcal{S}$ using a known reference string. Their experimental results showed that we could achieve 15% of extra compression with this approach in genomic data.

3.2.2 Grammars

A *context-free grammar* (CFG), or just grammar, is a tuple $\mathcal{G} = \{V, \Sigma, \mathcal{R}, \mathsf{S}\}$ that describes rewriting rules producing a set of strings in Σ^*. In this tuple, V is the alphabet of *nonterminal* symbols, Σ is the alphabet of *terminal* symbols, $\mathcal{R}$ is a list of productions that maps nonterminals to strings over $\Sigma \cup V$, and $\mathsf{S} \in V$ is the start symbol of $\mathcal{G}$. The nonterminals rewrite as strings, while terminal symbols cannot be replaced. The rules in $\mathcal{R}$ are represented as $\mathsf{A} \to B$, where $\mathsf{A} \in V$ is the nonterminal and B is a string over the alphabet $\Sigma \cup V$ that replaces A. The set of string in Σ^* we can obtain by recursively rewriting nonterminals is the language generated by $\mathcal{G}$, $\mathcal{L}(\mathcal{G})$. The *parse tree* of a string $S \in \mathcal{L}(\mathcal{G})$ is a labeled ordinal tree that represents the recursive nonterminal replacements leading to S. The root is labeled with S, the leaves are labeled with terminals spelling out S left to right, and the internal nodes are labeled with nonterminals: the children of A are, left to right, the symbols of B for some rule $\mathsf{A} \to B \in \mathcal{R}$.

The aim in *grammar compression* is to encode an input string $S[1, n]$ by finding a small grammar $\mathcal{G}$ whose language is $\mathcal{L}(\mathcal{G}) = \{S\}$. In this grammar, there is exactly one rule $\mathsf{A} \to B$ per $\mathsf{A} \in V$; we call $\mathsf{exp}(\mathsf{A}) \in \Sigma^*$ the only string of terminals derived from A, and then $S = \mathsf{exp}(\mathsf{S})$. The *size* $G = |\mathcal{G}|$ of the grammar is the sum of the lengths of the right-hand sides of the rules. Then we significantly compress S if we manage to build a grammar of size $G \ll |S|$ that generates only S.

In general, it is convenient to enforce some properties in $\mathcal{G}$ to avoid redundancies:

- Each nonterminal is the left-hand side in only one rule
- Right-hand sides in $\mathcal{R}$ cannot be empty strings
- Every nonterminal must appear at some point in the derivation of S
- Distinct nonterminals produce different strings in Σ^*
- Every nonterminal appears at least twice in the right-hand sides of $\mathcal{R}$
- There is no pair $\mathsf{XY} \in \Sigma \cup V$ occurring more than once in the right-hand sides of $\mathcal{R}$

[2] Sort the strings in lexicographical order from right to left.

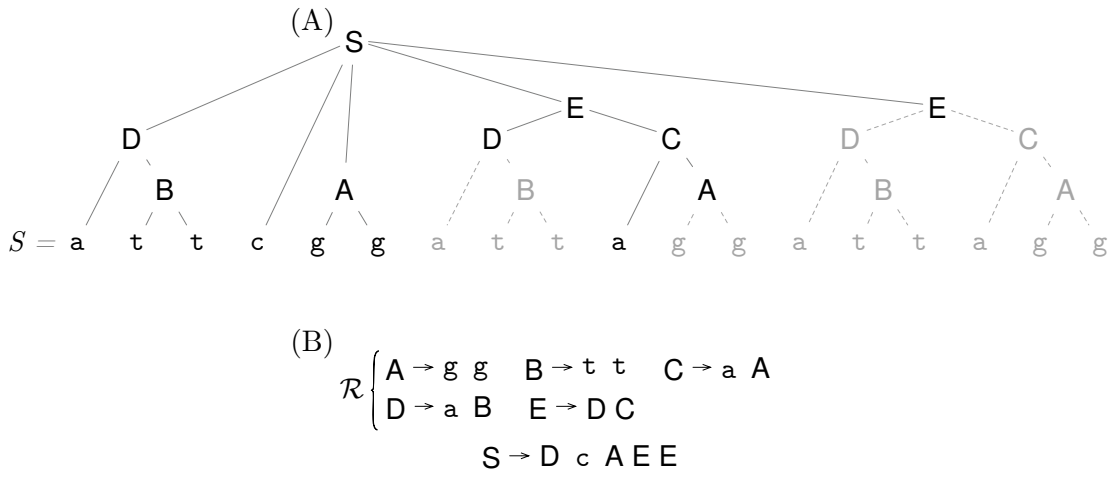

Figure 3.3: The CFG resulting from running RePair on the string S = `attcggattaggattagg`. (A) The parse tree of the grammar. Gray nodes and edges are the pruned elements of the grammar tree built in a pre-order traversal of the parse tree. (B) The set of rules $\mathcal{R}$ and the start symbol S.

Kieffer et al. [96] called the grammars that satisfy these properties *irreducible*. They also showed that irreducible grammars reach the kth order entropy of a source. Similarly, Ochoa and Navarro [143] demonstrated that the same holds for the kth order empirical entropy of individual strings. Grammars not just achieve compression under the statistical model, they also capture the repetitiveness of the text; the size of $\mathcal{G}$ is much smaller than n when S is repetitive [136].

There are important trade-offs between grammars and other compression methods. For instance, it is a well-known fact that obtaining the smallest grammar $\mathcal{G}^*$ that produces S is NP-complete [178, 32], while computing the BWT or Lempel-Ziv parse (see Section 3.2.3) of S takes linear time [141, 159, 178]. Besides, the size G^* of $\mathcal{G}^*$ is never smaller than the number of Lempel-Ziv phrases in S. On the other hand, there is no clear dominance between grammars and the BWT; G^* can be smaller or larger than r, the number of BWT runs [136].

Despite the drawbacks, grammars are still interesting because they support direct access to S with a logarithmic penalty [15] and within $\mathcal{O}(G \log n)$ bits. In contrast, with the Lempel-Ziv scheme, we can support direct access in $\mathcal{O}(z)$ time [101], where z is the number of phrases. In the case of the BWT, we can obtain efficient direct access, but it is still unknown if we can do it using space proportional to r.

Another important advantage of grammars is that there are good heuristics that perform well in practice for build them, RePair [108] being the most popular one. The RePair algorithm consists of recursively replacing the most frequent pair of symbols in S. Let `ab` be the most frequent pair at some point during the algorithm's execution. We create a new rule B → `ab` and replace the occurrences of `ab` in S with B. Then, we extract the new most frequent symbol pair in S and repeat the same procedure. The algorithm stops when all the pairs in S have frequency one. RePair achieves linear time by maintaining the pairs in a max priority queue, so the most repeated pair is always at the top after each update of S.

RePair produces grammars of size comparable to the Lempel-Ziv77 parse (LZ77) [193] in most of the cases. Nevertheless, RePair has a high cost in practice, which limits its use in big datasets. Although it runs in linear time and space, its working memory footprint is too high. Competitor tools like `p7zip` [150], which is based on LZ77, achieve slightly better compression ratios and require a negligible amount of working memory. To solve the problem, Gagie et al. [72] proposed a variation of RePair that preprocesses S first to catch long repetitive blocks. Their experimental results showed that they use between 7% and 11% of the working memory of RePair, while maintaining competitive compression ratios. Other more recent grammar algorithms [142] are faster and require less working memory than RePair, but produce bigger grammars. Still, they are faster at decompressing the text.

A run-length context-free grammar (RLCFG) [139] is an extension of CFGs that allows rules of the type $\mathsf{X} \to B^c$, where B^c represents in constant space the concatenation of $c > 2$ copies of symbol $B \in V \cup \Sigma$. A RLCFG usually compresses better than a regular CFG. For instance, in the string $S = A^c$, the smallest CFG has size $G^* = \Theta(\log n)$, whereas a RLCFG can reach size $G^*_{RLCFG} = \mathcal{O}(1)$. Of course, the general case is $G^*_{RLCFG} \leq G^*$.

Locally Consistent Grammars

A type of grammar relevant for this thesis is that generated from a *locally consistent parsing* [162, 128]. This procedure consists of partitioning a text $S[1, n]$ in a way such that the identical substrings are largely parsed in the same form. More specifically, a parsing is locally consistent if there are two integers a, b (which may depend on n) such that, for every pair of equal substrings $S[j, j+u] = S[j', j'+u]$, only their first a and their last b phrase boundaries can differ. In general, a locally consistent parsing algorithm puts a phrase boundary in S wherever some specific symbol combination arises. In our case, the first and last phrases of $S[j, j+u]$ might be formed in a different way than those in $S[j', j'+u]$ because they might be preceded or followed by different symbols. Note that this approach differs from other parsing algorithms such as Lempel-Ziv or RePair, which use global information on S to define its partition.

We build a *locally consistent grammar* by applying successive rounds of locally consistent parsing over $S[1, n]$. In every round i, we capture the distinct phrases in the input text S^i ($S^1 = S$) and create new nonterminals rewriting to them. We then build a new text S^{i+1} by replacing the phrases in S^i with their corresponding nonterminal symbols. This new text S^{i+1} is the input for the next round. The algorithm stops when S^i can no longer be partitioned. If the phrases in every S^i are of length at least 2, then the string S^{i+1} is at most half the length of S^i, and thus the number of parsing rounds is $\mathcal{O}(\log n)$ and the total running time is of the same order as for parsing S.

The algorithm described above produces a balanced grammar $\mathcal{G}$, which is probably bigger than the one we obtain with RePair. In exchange, if a pattern P appears more than once in T, then the parse subtrees containing its occurrences will be almost identical, differing only in a few nodes at the ends of every tree level. The internal part of the subtrees remains unchanged regardless P's context. This can be exploited to speed up pattern matching.

Recently, Christiansen et al. [34] proposed a locally consistent RLCFG of size $G = \mathcal{O}(\gamma \log(n/\gamma))$, where γ is the size of the smallest attractor of S [95]. In their algorithm,

the parsing rounds have two steps. In the first one, they create new nonterminal rules with the equal-symbol runs of S^i. These rules are of the form $\mathsf{A} \to a^l$. Then, they produce a new string $\hat{S}^i$ by replacing the runs with their generating nonterminals. In the second round step, they define a random permutation $\pi : \hat{\Sigma}^i \to [1, |\hat{\Sigma}^i|]$ for the symbols in the alphabet $\hat{\Sigma}^i$ of $\hat{S}^i$, and use this permutation to partition $\hat{S}^i$: each phrase ends in a local minimum, which is a position $\hat{S}^i[j]$ such that $\pi(\hat{S}^i[j-1]) > \pi(\hat{S}^i[j]) < \pi(\hat{S}^i[j+1])$.

The Grammar Tree

Another relevant concept is the *grammar tree* of $\mathcal{G}$ [36] (also referred to as the partial parse-tree by Rytter [161]). This representation is a pruned version of the parse tree. We build it as follows; we start a walk over the parse tree in some specific order. Every time we reach a internal node v whose label $\mathsf{X} \in V$ has not been seen before in the traversal, we create a new internal node in the grammar tree labeled with X. On the other hand, if X has been seen before, we create a leaf labeled with X instead, and skip the subtree of v from the traversal. When we reach a leaf v in the parse tree, we also create a leaf in the grammar tree. The resulting grammar tree contains exactly $G + 1$ nodes and $|\mathcal{R}|$ internal nodes. The example of Figure 3.3A depicts the idea.

Random Access

When storing S as a grammar, we are also interested in supporting random access to any $S[i]$ in compressed space. Recall that this feature is one of the main advantages of grammars over other dictionary-based approaches. A simple way to achieve random access is by storing the accumulative sums of nonterminals. Let $|\mathsf{A}|$ be the length of $\mathsf{exp}(\mathsf{A})$. Then, for each rule $\mathsf{X} \to \mathsf{A}_1 \dots \mathsf{A}_k \in \mathcal{R}$, we store the sequence $l_0 = 0, l_1 = l_0 + |\mathsf{A}_1|, \dots, l_k = l_{k-1} + |\mathsf{A}_k|$. Also, we define a predecessor data structure that, given the sequence $l_0, l_1, \dots, l_k$ of a rule and a position i, returns the element j such that $l_{j-1} < i \leq l_j$. We extract $S[i]$ from the grammar by first obtaining the position j for i in $\mathsf{S} \to C$. Then, we update $i = i - l_{j-1}$ and recursively apply the same idea using the rule of $C[j]$ and i as inputs. We continue the recursion until reaching a terminal symbol, which is the answer for $S[i]$.

When the grammar is balanced (i.e., its height is $\mathcal{O}(\log n)$), we require $\mathcal{O}(\log n)$ predecessor operations. Besides, with the data structure of Belazzougui and Navarro [11], we can answer these predecessor queries in $\mathcal{O}(\log \log_w n)$. In this way, accessing $S[i]$ takes us $\mathcal{O}(\log n \log \log_w n)$ time. If the right-hands of $\mathcal{R}$ are of constant length, then we can reduce the predecessor queries to $\mathcal{O}(1)$, reaching thus $\mathcal{O}(\log n)$ time to access $S[i]$. Bille et al. [15] recently demonstrated that we can obtain $\mathcal{O}(\log n)$ time to extract $S[i]$ from any type of grammar, not necessarily balanced. They also showed how to extract any substring $S[i, j]$ in $\mathcal{O}(j - i + \log n)$ time. Ganardi et al. [73] recently showed that we can transform any grammar of size G into a balanced grammar of size $\mathcal{O}(G)$, where the right-hand sides of $\mathcal{R}$ are of length two.

3.2.3 Other Compression Methods

There are other methods apart from Grammars and the BWT to compress repetitive text. Among them, Lempel-Ziv (LZ) is maybe the most popular. This algorithm greedily partitions S into a sequence of phrases. More precisely, LZ parses S from left to right, and given the

unprocessed suffix $S[i, n]$, it finds the longest prefix $S[i, j]$ whose sequence also appears at some position $S[i', j']$ before $S[i]$ (i.e., $i' < i$). Subsequently, it defines a new parse phrase $S[i, j + 1]$, and continues the processing from suffix $S[j + 2, n]$. The substring $S[i', j']$ is referred to as the source of $S[i, j + 1]$. The resulting parse is stored as a list of triplets $(i', j + 1 - i, S[j + 1])$. If this list has z elements, then the space usage of the representation is $O(z \log n)$ bits. When the text is repetitive, the phrases cover long segments of S, so z is much smaller than n. LZ has several variants, but LZ77 [193] and LZ78 [194] are the best known ones. In LZ77, the source $S[i', j']$ has to start within the l symbols previous to $S[i]$, where l is a parameter . In LZ78, the source must also be a phrase. In the other variants, the source can cross one or more previously created phrases. LZ78 is the variant that compresses the least, but allows us to extract any substring $S[i, j]$ in optimal time $\mathcal{O}(j - i + 1)$.

Another method for text compression is the Compact Directed Acyclic Word Graph (CDAWG) [19]. This data structure is an automaton obtained by merging all the identical subtrees in the suffix tree of S. As the text becomes more repetitive, the number of these identical subtrees increases, so the CDAWG becomes smaller. The size of the CDAWG is usually denoted as e, and represents the sum of its arcs and its edges. Still, CDAWGs are not as powerful as the other compression methods. In some string families, it holds that e is $\Theta(n)$ times larger than z or r [9] and $\Theta(n/\log n)$ larger than G [8].

3.3 Self-Indexes

A *self-index* is a versatile data structure that (i) stores a text $S[1, n]$ in compressed form, (ii) supports pattern matching on S in time sublinear [3] in n, and (iii) allows random access to S's substrings. These representations, in addition to answering count and locate (Section 3.1), also support the query:

- display(I_S, i, j) : extract the sequence $S[i, j]$ from the self-index I_S

Self-indexes are an excellent alternative when the space overhead of the suffix array or suffix tree is too high for practical purposes. This situation occurs, for example, when we need to index massive string collections. In particular, self-indexes have thrived in genomic applications that require locating small DNA sequences within large genome databases.

Popular self-indexes exploit the unbalance in symbol frequencies or regularities in the text's suffix array to reduce space usage. In the first case, their compression performance is usually assessed in terms of the empirical $\mathcal{H}_k$ entropy. Recall from Section 2.1.1 that this value is a lower bound for the average number of bits we require to represent the symbols in S if we use kth order statistical compression. This framework, although useful for many text collections, is not suitable when S is repetitive (see Section 3.2). Other self-indexes, explicitly designed for repetitive collections, achieve better compression. They factorize the text using dictionary-based methods such as Lempel-Ziv or CFGs, or exploit the equal-symbol runs in the text's BWT. Nevertheless, their pattern matching functionalities are usually slower than those of entropy-based self-indexes.

[3] $\mathcal{O}((m^c + occ) \log^{\epsilon} n)$, where c and ϵ are constants > 0, m is the pattern length and occ is the number of occurrences of the pattern.

In this section, we review the most successful self-index that relies on statistical entropy. Besides, we review other self-indexes tailored to repetitive strings with potential applications in modern genomic analyses.

3.3.1 FM-Index

The FM-Index [65] is a self-index that represents S in terms of the arrays L and C of Section 2.2. The first one is S's BWT while the second stores in $C[i]$, with $i \in [1, \sigma]$, the number of symbols in S smaller than i. We saw in Section 2.2 that we can encode L and S in $n(\mathcal{H}_0(L) + 1)(1 + o(1)) + \mathcal{O}(\sigma w)$ bits of space so that reconstructing S takes us $\mathcal{O}(n \log \sigma)$ time.

This representation also allows us to count the number of occurrences of a pattern $P[1, m]$ in S in $\mathcal{O}(m \log \sigma)$ time (the operation $\mathsf{count}(I_S, P)$ of Section 3.1). The procedure is called $\mathsf{backwardsearch}(I_S, P)$. Recall from Section 3.1 that the suffixes in S prefixed by P form a consecutive range $SA[s, e]$, and its corresponding segment $L[s, e]$ stores the preceding symbols of P in S. Let us denote $SA[s_j, e_j]$ the range that contains the suffixes of S prefixed by $P[j, m]$, with $j \in [1, m]$. The backward search builds on the observation that if we already know $SA[s_j, e_j]$, then we can compute the range $SA[s_{j-1}, e_{j-1}]$ directly from $L[s_j, e_j]$. The procedure starts by obtaining the range $SA[s_m, e_m]$, with $s_m = C[P[m]] + 1$ and $e_m = C[P[m] + 1]$. Then, for every suffix $P[j, m]$, with $j \neq m$, we obtain the range $SA[s_{j-1}, e_{j-1}]$ from $L[s_j, e_j]$ with the formula:

$$s_{j-1} = C[P[j-1]] + \mathsf{rank}_{P[j-1]}(L, s_j - 1) + 1$$
$$e_{j-1} = C[P[j-1]] + \mathsf{rank}_{P[j-1]}(L, e_j).$$

We refer to this operation as a $\mathsf{backwardsearch}$ *step*. After applying $m - 1$ steps, the resulting range (s_1, e_1) is indeed $SA[s, e]$. Finally, we report $e - s + 1$ as the number of occurrences of P in S. Obtaining (s_m, e_m) takes constant time, and computing each pair (s_j, e_j) takes $\mathcal{O}(\log \sigma)$ time if we encode L as a wavelet tree (Section 2.2.2). Thus, the total time for $\mathsf{backwardsearch}(I_S, P)$ is $\mathcal{O}(m \log \sigma)$ time. Notice that this function does not explicitly require the suffix array of S.

If we also require to support $\mathsf{locate}(I_S, P)$, then we can augment L and C with the suffix array of S. In this way, this operation reduces to perform $\mathsf{backwardsearch}(I_S, P)$ and then report the positions in $SA[s, e]$. The whole operation is thus implemented in $\mathcal{O}(m \log \sigma + occ)$ time. However, the $n \log n$ extra bits of the suffix array can be too expensive. A common technique to reduce the space overhead is to sample SA at regular *text* intervals.

We define a sampling rate l and collect all the suffix array values that satisfy $SA[i] \bmod l = 0$. We store such samples in another vector $A[1, \lceil n/l \rceil]$ without changing their relative orders. In addition, we define a bit vector $B[1, n]$ that marks the sampled positions of the SA. That is, $B[j] = 1$ if $SA[j] \bmod l = 0$. The representation for S now becomes C, L, A and B.

Now displaying the text positions in $SA[s, e]$ is a bit different. Let j be a value in the range (s, e). If $B[j] = 1$, then we report $SA[j] = A[\mathsf{rank}_1(B, j)]$. If, on the other hand, $B[j] = 0$, then $SA[j]$ was not sampled, so we have to infer it. Starting from $L[j]$, we perform

LF iteratively until finding a position j' such that $B[j'] = 1$. Assume we found j' after k LF operations, then we report $SA[j] = A[\mathsf{rank}_1(B, j')] + k$. As we sampled a suffix array value every l text positions, and LF enumerates the symbols in reverse text order, finding $SA[j]$ takes us $\mathcal{O}(l \log \sigma)$ time. One can set $l = \log^{1+\epsilon} n / \log \sigma$ for any given constant $\epsilon > 0$ so that the sampled values require $(n/l) \log n = o(n \log \sigma)$ bits and computing a non-sampled value takes $\mathcal{O}(\log^{1+\epsilon} n)$ time.

The last operation we need to support is $\mathsf{display}(I_S, i, j)$. We implement it by first retrieving the position $SA[j'] = j + 1$ for $S[j+1, n]$, then applying $\mathsf{LF}^{j-i+1}(j')$, and finally reporting the reversed sequence of symbols we accessed in L during the LF calls. The problem with this idea is that we do not know j'. To solve it, we augment the FM-index with the *inverse suffix array*. This data structure is a vector $SA^{-1}[1, n]$ that tells us for every suffix $S[u, n]$ its lexicographical rank u'. In other words, $SA^{-1}[u] = u'$ implies that $SA[u'] = u$. We reduce the space requirements by maintaining a sampled version of SA^{-1}. We chose a sampling rate l and create an array $A^{-1}[1, \lceil n/l \rceil]$ to store the SA^{-1} values of the text positions that are multiples of l. As a special case, we can store $A^{-1}[\lceil n/l \rceil]$ for $S[n]$, although it is not necessary as this suffix is always mapped to $SA[1]$. To get j', we first obtain the suffix array position $x = A^{-1}[\lceil (j+1)/l \rceil]$ for the smallest suffix to the right of $S[j+1, n]$ that was sampled. If $j + 1$ is multiple of l (or n), then j' is x. If not, then we apply no more than l LF steps from $L[x]$ to get j'. Thus, answering $\mathsf{display}(S, i, j)$ takes us $\mathcal{O}((j - i + l) \log \sigma)$ time.

The complete FM-index is composed by the data structures L, C, B, A and A^{-1}. If we use the same sampling rate for A and A^{-1}, then its total space usage is $n(\mathcal{H}_0(L) + 1)(1 + o(1)) + \mathcal{O}(\sigma w) + \mathcal{O}((n/l) \log n)$ bits.

3.3.2 Bidirectional FM-Index

A limitation of the FM-Index is that it is *asymmetric*. Given the range $SA[s, e]$ of suffixes prefixed with P, we can easily obtain, in one backward search step, the suffix array range for aP, with $a \in \Sigma$. On the other hand, if we require the suffix array range for pattern Pa, computing it from $SA[s, e]$ is not that simple.

A common solution for this limitation is to have a *bidirectional* FM-index [104]. This variation considers two BWTs. The first one, L, is for S, and other, $\bar{L}$, is for the reversed version of S, denoted as $\bar{S}$. To simplify the explanations, we will refer to the string aP as a left extension of P, and the string Pa as a right extension.

When performing a backward search step in the bidirectional FM-index, we maintain both BWTs synchronized. If $L[s, e]$ has the symbols that precede P in S, $\bar{L}[p, q]$ has the symbols that precede $\bar{P}$ in $\bar{S}$. Put another way, $\bar{L}[p, q]$ has the symbols that follow P in S. These two ranges have the same sizes as the number of occurrences of P in S is the same as the number occurrences of $\bar{P}$ in $\bar{S}$.

Assume we perform a backward step over $(L[s, e], \bar{L}[p, q])$ using $c \in \Sigma$. The new pair of ranges $(L[s', e'], \bar{L}[p', q'])$ store the symbols that precede cP in S and $\bar{P}c$ in $\bar{S}$, respectively. Note that (p', q') is fully contained within (p, q) as the frequency of $\bar{P}c$ in $\bar{S}$ is equal or smaller than the frequency of $\bar{P}$, and both patterns share the same prefix. Further, we know that the right extensions of $\bar{P}$ in $\bar{S}$ are equivalent to the left extensions of P in S. Hence,

to maintain the BWTs synchronized, we perform a backward search step over $L[s, e]$ to get $L[s', e']$, and then use the operation $\mathsf{rangecount}(L, s, e, 1, c-1)$ of Section 2.2.2 to count the k symbols in $L[s, e]$ that are lexicographically smaller than c. Using this information, we compute the range $p' = p + k$ and $q' = p' + e' - s'$ for $\bar{L}$.

The operation above describes a synchronized left extension cP. If we want a right extension Pc, we have to perform the opposite procedure; compute (p', q') with a backward search step on $\bar{L}[p, q]$, and then compute $k = \mathsf{rangecount}(\bar{L}, p, q, c)$ to set $s' = s + k$ and $e' = s' + q' - p'$.

There is an interesting connection between the bidirectional FM-index and the suffix tree of S; spelling a prefix $\bar{S}[1, j]$ using LF over $\bar{L}$ is equivalent to spelling the path X in the suffix tree of S labeled with $S[n - j + 1, n]$. Using this observation, we can simulate a traversal over S's suffix tree directly from $\bar{L}$. In doing so, we map specific ranges in $\bar{L}$ to specific suffix tree nodes. Let P be a prefix in the label of X and let $\bar{L}[p, q]$ be the range storing the symbols that precede $\bar{P}$ in $\bar{S}$. If $\bar{L}[p, q]$ contains more than one distinct symbol, then there is an internal node v in the suffix tree of S labeled with P, whose subtree contains $p - q + 1$ leaves. In contrast, if $\bar{L}[p, q]$ is an equal-symbol run, then there is no internal node labeled with P as it is not right maximal.

Using arrays L and $\bar{L}$ we can simulate some navigational operations over the suffix tree. For instance, we can obtain the Weiner links of v using backward search steps over $L[i, j]$. Suppose we perform a backward search step in $L[i, j]$ using one of its distinct symbols, say $c \in \Sigma$. If the resulting range $L[i', j']$ is an equal-symbol run, it represents an implicit Weiner link (see Section 3.1.2). On the other hand, if $[i', j']$ has more than one distinct symbol, then it maps a node v' such as that an explicit Weiner link (v, v') labeled with c exists in the suffix tree of S.

We can access the kth child of v from $\bar{L}[p, q]$. Assume c is the kth smallest symbol appearing in $\bar{L}[p, q]$. We apply a backward search step over $\bar{L}[p, q]$ using c and continue performing backward search steps until finding a range $\bar{L}[p', q']$ with more than one distinct symbol. If we maintain synchronized L and L' in every step, then the resulting $L[i', j']$ maps the kth child of v.

Computing the suffix link (v, v') of v is also possible, but we must augment L with additional data structures. Using the operations $\mathsf{LF}^{-1}(i) = i'$ and $\mathsf{LF}^{-1}(j) = j'$, we obtain a subrange $[i', j']$ within the range of v' in L, which we use to answer the suffix link query. The problem, however, is that we need the topology of the suffix tree to expand $[i', j']$ to the range of v'. There is a similar problem when computing the parent of v.

3.3.3 The r-index

The r-index [70] is a variation of the FM-index that requires space proportional to the number of equal-symbol runs in the BWT of S. This number is usually denoted as r, and serves as an ad-hoc measure to assess the repetitiveness of a text. If S is indeed repetitive, then its value for r is small compared to n. Although the FM-index represents S within its statistical entropy (Section 3.3.1), this scheme is insensitive to the redundancy on the text (see Section 3.2). Hence, it grows linearly with n regardless the kind of new data we append

to S. In contrast, an r-index grows with the amount of new information on S. This means that if we add new sequences to S that are identical or highly similar to those already in the text, then the index's space usage will grow slowly.

The r-index is composed of two data structures; a run-length compressed version of the BWT with rank support [122] and a sampled suffix array that stores one value per BWT run. We first describe the run-length compressed BWT, and then explain how to perform backwardsearch on it. We refer to this data structure as the RLBWT of S. We then explain the sampling technique for the suffix array and how to report the text positions within the SA range reported by backwardsearch.

The first element of the RLBWT is a vector $L'[1, r]$ storing the first symbol of every equal-symbol run of L, without changing the relative order of the runs. Another array $C'[1, \sigma]$ stores in $C'[i]$ the number of symbols in L' that are lexicographically smaller than i. We also create an array $R[1, r]$ to store information about the run lengths. Every range $R[a, b]$, with $a = C'[s] + 1$ and $b = C'[s + 1]$, is associated with the runs of symbol $s \in \Sigma$. More specifically, position $R[u]$, with $a \leq u \leq b$, has the cumulative lengths of the first $u - a + 1$ runs of s in L. The last component of the RLBWT is a predecessor data structure. The operation $\mathsf{pred}(j) = (j', p)$ receives as input a position $j \in [1, n]$ and returns a pair where $j' \leq j$ is the position of the leftmost symbol in the run of $L[j]$ and p the rank of that run in L.

We now can simulate the operation $\mathsf{rank}_s(L, j)$ over the RLBWT as follows; we first call the operation $\mathsf{pred}(j) = (j', p)$. The output tells us that $L[j]$ lies within the pth BWT run, and that the leftmost symbol in that run is $L[j']$. Subsequently, we obtain the number of runs for s in $L'[1, p - 1]$ as $x = \mathsf{rank}_s(L', p - 1)$. Using x, we obtain the real rank of s in $L[1, j]$ as $l = R[C'[s] + x]$. Additionally, when $s = L'[p]$, we add $(j - j' + 1)$ to l to consider the $(j - j' + 1)$ occurrences of s between $L[j']$ and $L[j]$. Finally, we return l as the answer for $\mathsf{rank}_s(L, j)$.

The time complexity for the function described above is dominated by the $\mathsf{pred}(j)$ and $\mathsf{rank}_s(L', s)$ operations. Gagie et al. [70] use the predecessor data structure of Belazzougui and Navarro [11] to support $\mathsf{pred}(j)$ in $\mathcal{O}(\log\log_w(n/r))$ time and $\mathcal{O}(r \log n)$ space, where w is the machine word's length in bits. Besides, they use an alternative representation for L' that supports rank in $\mathcal{O}(\log\log_w \sigma)$ time. Thus, the time complexity for $\mathsf{rank}_s(L, j)$ becomes $\mathcal{O}(\log\log_w(\sigma + n/r))$.

Now that we can simulate the operation $\mathsf{rank}_s(L, j)$ in the RLBWT, we can implement $\mathsf{backwardsearch}(I_S, P)$ in $\mathcal{O}(m \log\log_w(\sigma + n/r))$ time, where I_S is the r-index and P is an input pattern of length m. This result implies that we can also implement $\mathsf{count}(I_S, P)$ in the same time and within $\mathcal{O}(r \log n)$ bits of space.

Supporting $\mathsf{locate}(S, P)$ in $\mathcal{O}(r \log n)$ space is a bit more complicated but not impossible. We start by creating an array $A[1, r]$ to complement R. $A[j]$ stores the suffix array value for the last entry of the run referenced by $R[j]$. Now, for answering $\mathsf{locate}(S, P)$, we modify the backwardsearch algorithm so that when we obtain the range $SA[s, e]$ for P, we also know $SA[e]$. Assume we already have the boundaries (s_j, e_j) for $P[j, m]$ and its corresponding value $SA[e_j]$. We now have to perform a backward search step to obtain the information of

$P[j-1,m]$. To compute $SA[e_{j-1}]$, we use $L[s_j,e_j]$ and the new array A. If $L[e_j]$ is equal to $P[j-1]$, then $SA[e_{j-1}] = SA[e_j]-1$. If, on the other hand, $L[e_j]$ differs from $P[j-1]$, then we need to find in $L[s_j,e_j]$ the position y with the last occurrence of $P[j-1]$. We know $L[y]$ is the last element in a BWT run so its SA value is in A. We infer y using the same mechanism for supporting $\mathsf{rank}_s(L,j)$ in the RLBWT; $y = C'[P[j-1]] + \mathsf{rank}_{P[j-1]}(L', \mathsf{pred}(e_j).j' - 1)$. Finally, the value for $SA[e_j]$ is $A[y]-1$.

The last aspect to address for locate is how to obtain the other suffix array values within the range $SA[s,e-1]$. Gagie et al. [70] made the following observation; let $L[j]$ and $L[j-1]$ be two consecutive symbols that belong to the same run. The equality $SA[j-1]-SA[j] = SA[\mathsf{LF}(j-1)]-SA[\mathsf{LF}(j)]$ holds as long as the LF operations for j and $j-1$ redirect us to positions in the BWT that also belong to the same run. Put differently, let j and $i = j-1$ be two consecutive positions in L. Suppose we iteratively apply $j = \mathsf{LF}(j)^k$ and $i = \mathsf{LF}(i)^k$, where k is the number of steps needed for $L[j]$ and $L[i]$ to belong to different BWT runs. In each of these steps, the values of j and i changed, but they remained contiguous ($j-i=1$). Therefore, the difference $d = SA[i]-SA[j]$ stayed the same in all of them. However, after the kth LF step, $L[j]$ and $L[i]$ are positioned in different runs, so applying $\mathsf{LF}(j)$ and $\mathsf{LF}(i)$ will yield two values j and i that are no longer contiguous in the BWT. This difference means that $d = SA[i]-SA[j]$ now is a different subtraction. Note that, from all the k distinct values assigned to j during the LF steps, only $\mathsf{LF}^k(j)$ (the last one) represents a run head in L.

We exploit the ideas described in the previous paragraph as follows; we create another predecessor data structure that returns for every $SA[j]$ the previous text position $SA[j'] = SA[j]-k$ such that $S[SA[j']-1]$ is a run head in the BWT. In addition, for every $SA[j']$ encoded in the predecessor data structure, we store the difference with its previous suffix array value as $d(SA[j']) = SA[j'-1]-SA[j']$. Now suppose we already know $SA[j]$, with $j \in [s,e]$. For computing $SA[j-1]$, we search for $SA[j']$ in the predecessor data structure and then report $SA[j-1] = SA[j]+d(SA[j'])$. During the computation of an SA value, the time complexity is dominated by the predecessor operation, which takes $\mathcal{O}(\log\log_w(n/r))$ time if we use the data structure of Belazzougui and Navarro [11]. Thus, reporting the values in $SA[s,e]$ takes $\mathcal{O}((e-s+1)\log\log_w n/r)$ time.

As a conclusion, locate in the r-index takes $\mathcal{O}(m\log\log_m(\sigma+n/r)+(e-s+1)\log\log_w n/r)$ time, and $\mathcal{O}(r\log n)$ space.

3.3.4 The Grammar Index

A regular grammar index [36, 37] consists of a CFG generating only S (Section 3.2.2) and a geometric data structure [31] used to perform efficient matching on S. This data structure is an interesting alternative to the r-index. Although both are sensitive to text repetitions, the r-index usually requires more space. Experimental results [37] showed that, on repetitive collections, the grammar index uses about 64% of the r-index's space. In less repetitive collections, this percentage reduces to 53%. Another recent work [38] showed that, on microbial genomes with a small number of BWT runs ($n/r = 51.2$), the grammar index requires 73% of the r-index's space. However, in microbial organisms where r is higher ($7.5 \leq n/r \leq 40$), this percentage decreases to $37\% - 49\%$. These results show that the r-index's space usage

grows faster than the grammar index's space usage as the text repetitiveness drops. However, performing pattern matching in the grammar index is slower than in the FM-index (experimental results of Cobas et al. [38] showed that it is about four times slower than the r-index).

Before explaining this data structure, it is convenient to recall some notation from Section 3.2.2. Let $\mathcal{G} = (V, \Sigma, \mathcal{R}, \mathsf{S})$ be a CFG that only produces S. The symbol G is the size of $\mathcal{G}$ and the symbol $g = |\mathcal{R}|$ is the total number of nonterminals. In addition, the grammar tree of $\mathcal{G}$ is denoted as T'. The internal node of T' encoding the first preorder occurrence of a nonterminal X in the parse tree is called the *locus* of X.

The grammar index is composed of two elements; a succinct representation of T' and a succinct representation of the geometric data structure. We use T' to generate a partition over S, and then we index the resulting phrases in the geometric data structure to support pattern matching. The leaves in T' induce the partition as follows; let v be a leaf in T' generated from a node v' in the parse tree of $\mathcal{G}$. The substring $S[i, j]$ whose symbols match the leaves under the subtree of v' is the phrase induced by v.

The Grammar Representation

In order to use T' in the grammar index, its associated grammar $\mathcal{G}$ must have the following properties:

1. For every terminal $a \in \Sigma$, there is a nonterminal rule $\mathsf{X}_a \to a$
2. Each nonterminal appears at least in two distinct left-hand sides in $\mathcal{R}$. The only exceptions are S and the nonterminals of property 1
3. There is no rule in $\mathcal{R}$ whose left-hand side is empty or a single nonterminal symbol
4. If $\mathsf{X} < \mathsf{Y}$, then the reverse sequence of $\mathsf{exp}(\mathsf{X})$ is lexicographically smaller than the reverse sequence of $\mathsf{exp}(\mathsf{Y})$

When building T', we encode all the nonterminals of property 1 as leaves, including those that are the first preorder occurrence in the parse tree. Once we produce T', we encode its topology using DFUDS (Section 2.2.3). We store its node labels in an array $X[1, G]$, where each $X[j]$, with $j \in [1, G]$, is the label of the node with preorder $j + 1$ (we skip the root). We represent X using the data structure of Belazzougui and Navarro [11] that supports $\mathsf{select}_a(X, k)$ in $\mathcal{O}(\log \log g)$ time, where $a \in [1, g]$ is a symbol in X and k is its rank. In addition, we create a bitmap $C'[1, g]$ in which we mark the nonterminals of property 1. Finally, we create a bit map $L'[1, n]$ and set $L'[j] = 1$ if a partition phrase starts at position $S[j]$.

Using this representation, we can simulate a traversal over the parse tree of $\mathcal{G}$. We start from any internal node of T' we want to expand and traverse its subtree top-down as long as we visit internal nodes. When we reach a leaf v, we first obtain its label $\mathsf{Y} = X[\mathsf{preorder}(v) - 1]$. If $C'[l] = 1$, then Y encodes the terminal symbol $\mathsf{rank}_1(C', l)$, so we report it. If, on the other hand, $C'[l] = 0$, then we obtain the locus v' of Y, and recursively complete the traversal from v' before continuing the main traversal. We compute the locus' preorder first as $p = \mathsf{select}_{\mathsf{Y}}(X, 1)$ in $\mathcal{O}(\log \log g)$ time, and then use $\mathsf{nodeselect}(p)$ to get v'.

The procedure described above enables the expansion $F = \mathsf{exp}(\mathsf{X})$ of any nonterminal X in $\mathcal{O}(|F|)$ traversal steps. However, we still have a $\log \log g$ penalty when computing a locus from a grammar tree leaf. Claude et al. [37] showed that we can augment the representation of T' with $\mathcal{O}(G)$ extra bits so that we can move from a leaf to a locus in constant time. They also showed that we can augment the representation of T' with another $(g-\sigma)\log\sigma+\epsilon\log\sigma+\mathcal{O}(g)$ bits to decompress the first or last l symbols of F in $\mathcal{O}(l/\epsilon)$ time.

The bit vector L' allows us to map in constant time any node v in T' to the position in S of the first (from left to right) induced phrase under its subtree. This operation, denoted as $\mathsf{p}(v)$, is computed as $\mathsf{select}_1(L', \mathsf{leafrank}(v) + 1)$.

Geometric Data Structure

We first define two string sets; the first one, $\mathcal{Y}$, will have g strings, and the second, $\mathcal{X}$, will have $G - g + \sigma$ strings. The sets are built as follows; let $\mathsf{A} \to \mathsf{B}_1 \dots \mathsf{B}_t \in \mathcal{R}$ be a nonterminal rule and let v be the locus for A in T'. For every $i \in [1, t]$, we insert the reverse sequence of $\mathsf{exp}(\mathsf{B}_i)$ into $\mathcal{Y}$. Additionally, for every proper suffix $\mathsf{B}_i \dots \mathsf{B}_t$, with $i \in [2, t]$, we insert the string $\mathsf{exp}(\mathsf{B}_i \dots \mathsf{B}_t)$ into $\mathcal{X}$. We build a matrix M of $g \times (G - g + \sigma)$ cells and use $\mathcal{Y}$ and $\mathcal{X}$ to label its rows and columns, respectively. Every row j is labeled with the string in $\mathcal{Y}$ with lexicographical rank j. Equivalently, every column j' is labeled with the string in $\mathcal{X}$ with lexicographical rank j'. The cell of M in the intersection of the row labeled with the reverse of $\mathsf{exp}(\mathsf{B}_i)$ and the column labeled with $\mathsf{exp}(\mathsf{B}_{i+1}) \dots \mathsf{exp}(\mathsf{B}_t)$ stores the identifier of the child number $i + 1$ of v from left to right.

In order to support pattern matching, we require to implement the following operations:

1. For any given m, extract the first m symbols of a row label $l \in \mathcal{Y}$
2. For any given m, extract the first m symbols of a column label $l' \in \mathcal{X}$
3. Given a segment (x_1, y_1, x_2, y_2) in M, report the k pairs (x, y), with $x_1 \le x \le x_2$ and $y_1 \le y \le y_2$ such that the cell (x, y) is not empty

The nonterminal symbols of $\mathcal{G}$ are the lexicographical ranks of their reversed string expansions. A convenient consequence of this property is that each row j in M is labeled with the nonterminal j. Hence, it is not necessarily to store $\mathcal{Y}$ explicitly. We can map the nonterminal j to its locus in T', and from that locus, obtain its string expansion in linear time (operation 1 above). Another important observation is that every column in M has only one used cell. If a column is labeled with the string $F = \mathsf{exp}(\mathsf{B}_{i+1}) \dots \mathsf{exp}(\mathsf{B}_t)$, then its used cell contains the locus in T' for the occurrence of B_{i+1} that belongs to the sequence $\mathsf{B}_{i+1} \dots \mathsf{B}_t$. From that locus, we can easily decompress F in linear time (operation 2 above). Thus, it is not necessary to store the labels of $\mathcal{X}$ either.

If we use the representation of Chan et al. [31] to encode M, we can extract the cell value for a column in $\mathcal{O}(1)$ time and perform operation 3 in $\mathcal{O}((k + 1)(1 + \log g / \log\log G)$ time. This data structure has a space complexity of $(G - g + \sigma)(\log g + \log G) + o(G \log g)$ bits.

Pattern Matching

To search for a pattern $P[1, m]$ in the grammar index, we classify its occurrences in two types. *Primary occurrences* span two or more phrases in the partition of S while *secondary occurrences* are fully contained within a phrase. The strategy to locate P in S consists of using M to find the loci in T' of the lowest nonterminals whose subtrees have primary occurrences of P. Once we find them, we locate the secondary occurrences of P by visiting the leaves of T' labeled with them or with their ancestors.

Let $S[i, j]$ be a primary occurrence of P intersecting k different phrases in the partition of S. These phrases were induced from a group of leaves that appear consecutively in T'. We need to find their least common ancestor v because the string expansion of the label of v is a substring in S that contains P. Note that in every primary occurrence of P, the sequence of intersected phrases is always different, and so is the least common ancestor. We need to find all such ancestor nodes to report all the primary occurrences of P.

Finding Primary Occurrences in the Grid

For every possible partition point $1 \leq u < m$, we cut the pattern into two halves $P[1, u]$ and $P[u + 1, m]$. The idea is to find the range of rows (y_1, y_2) in M whose labels are suffixed by $P[1, u]$ and the range (x_1, x_2) of columns prefixed by $P[u + 1, m]$. The non-empty cells within the grid segment (x_1, y_1, x_2, y_2) indicate the nodes in T' with primary occurrences of P. We perform two binary searches to locate this segment; one for the reverse of $P[1, u]$ in the prefixes of $\mathcal{Y}$, and another for $P[u + 1, m]$ in the prefixes of $\mathcal{X}$. When comparing $P[1, u]$ against the row labels, we extract the last u characters of a string in $\mathcal{Y}$ in $\mathcal{O}(u/\epsilon)$ time using operation 1 of the geometric data structure. Similarly, when comparing $P[u + 1, m]$ against the column labels, we extract the first $m - u + 2$ symbols of a string of $\mathcal{X}$ in $\mathcal{O}((m - u)/\epsilon)$ time using operation 2. Thus, we obtain (x_1, y_1, x_2, y_2) in $\mathcal{O}((m/\epsilon) \log G)$ time. We need to repeat this procedure with the $m - 1$ distinct cuts of P. Therefore, the final time complexity to get the grid segments with primary occurrences raises to $\mathcal{O}((m^2/\epsilon) \log G)$. We can reduce this time to $\mathcal{O}((m^2/\epsilon) \log \log n)$ by augmenting the index with $\mathcal{O}(G)$ extra bits implementing sampled Patricia trees [129]. Still, the binary search remains quadratic on m. Once we get the grid segments, we retrieve the values in their non-empty cells using operation 3 of the geometric data structure. Thus, the time complexity to find the loci in T' of the *occ* primary occurrences of P is $\mathcal{O}((m^2/\epsilon) \log \log n + (m + occ)(1 + \log g / \log \log G))$.

Reporting Primary and Secondary Occurrences

Let v be one of grid values for the cut $P_l \cdot P_r$. So far, we know that the string expansion $F = \mathsf{exp}(\mathsf{label}(v))$ is prefixed by P_r. We can easily obtain the position in S of F using the operation $\mathsf{p}(v)$. Thus, the location of the primary occurrence of P associated to v is $\mathsf{p}(v) - |P_l| + 1$. The next step is to report the secondary occurrences of v. Note that the string expansions of the nonterminals labeling the ancestors of $\mathsf{parent}(v)$ also have P as a substring. Therefore, the set X of leaves in T' labeled with these nonterminals expand to phrases that contain secondary occurrences of P. This idea recursively applies for the ancestors of the leaves in X. Let u be one of the ancestors of v; we compute its associated nonterminal Y. Then, for every leaf u' in T' labeled with Y, we report the position $p(v) - |P_l| + p(u') - p(u)$ in S as a secondary occurrence of P. Besides, every time we reach a new leaf u', we also

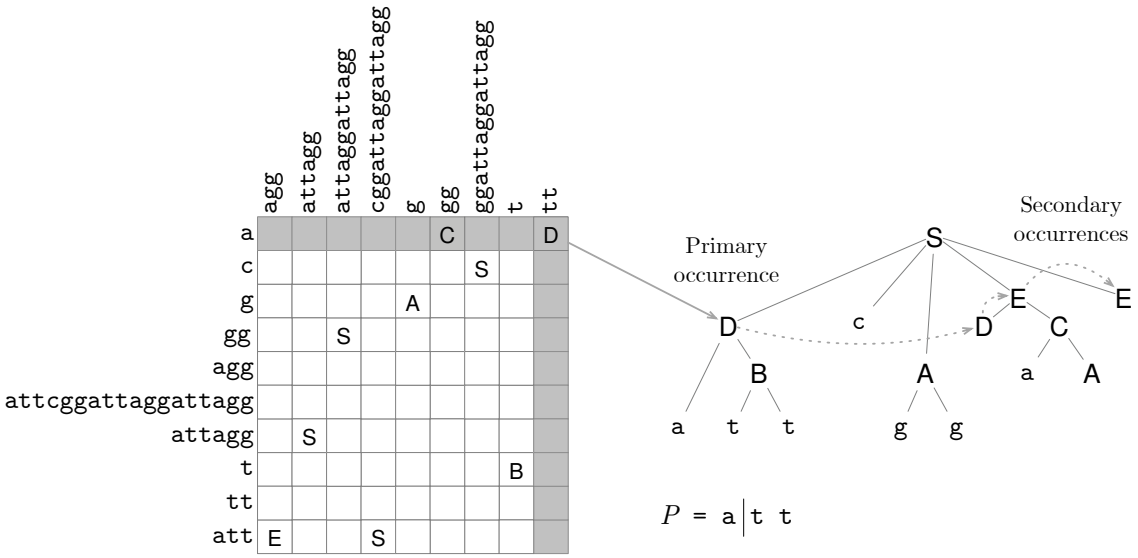

Figure 3.4: A grammar index built from a CFG $\mathcal{G}$ that only generates the string S = `attcggattaggattagg`. The grammar tree T' for $\mathcal{G}$ is depicted on the right side of the figure. T' is the same as shown in Figure 3.3. The matrix M with the indexed rules of $\mathcal{G}$ is shown on the left side of the figure. The strings in the rows belong to the set $\mathcal{Y}$, and the vertical strings belong to the set $\mathcal{X}$. A cell $M[i, j]$ is labeled with the nonterminal symbol of the rule from which the string $\mathcal{Y}[i]\cdot\mathcal{X}[j]$ was expanded. The figure also illustrates the procedure to locate the occurrences of a pattern P = `att` in S. The vertical line in P is a cut we try on M. The gray cells in the M are the rows and columns reported by the binary searches associated with the halves of the cut. The gray arrow from M to T' represents the mapping of a primary occurrence of P to its locus in T'. The dashed arrows in T' are the leaves we visit to report the secondary occurrences of P.

have to inspect the nonterminals of its ancestors, and find the leaves labeled with them. The complete algorithm processes all the grid point of each cut of P. Given the representation of Claude et al. [37], we can report the *occ* secondary occurrences of P in $\mathcal{O}(occ(1/\epsilon + \log\log g))$ time. This is because the grammar was preprocessed so that every nonterminal has another occurrence as a leaf in T', and thus the work done on the ancestors of each occurrence amortizes to some other occurrence of P.

The Resulting Index

The grammar index described by Claude et al. [37] uses at most $G\log n + 2G\log g + \epsilon g\log g + o(G\log g) + \mathcal{O}(G)$ bits of space for any constant $0 < \epsilon \leq 1$, and can find the *occ* occurrences of a pattern $P[1, m]$ in S in time $\mathcal{O}(m^2\log\log n + (m + occ)\log g/\log\log g)$. We can adjust ϵ so that the upper bounds become $G\log n + (2+\epsilon)G\log g$ bits for space and $\mathcal{O}((m^2 + occ)\log G)$ time for pattern matching. Figure 3.4 shows an example of the grammar index.

Improving the Pattern Matching

Christiansen et al. [34] showed that, if we build the grammar index of Claude et al. [37] using their locally consistent grammar (Section 3.2.2), then we require to test only $\mathcal{O}(\log m)$ cuts of P to find its primary occurrences in $\mathcal{G}$. Their idea consists in preprocessing P at query

time with the same algorithm they used to build $\mathcal{G}$. In every round i, they build the parse P^{i+1} by querying a hash table that maps the phrases in P^i to nonterminals in the grammar. The prefix $P^i[1, a]$ and the suffix $P^i[b, |P^i|]$ that do not represent complete phrases do not have symbols in P^{i+1}; analogously with $\hat{P}^i[1, \hat{a}]$. The preprocessing yields a list Q with the positions in P that limit incomplete parsing phrases. More specifically, every position $q \in Q$ is either the rightmost symbol under $P^i[a]$ or $\hat{P}^i[\hat{a}]$ in P's parse tree, or the leftmost symbol under $P^i[b]$. The elements in Q denote the cuts we try in the geometric data structure. As there are $\mathcal{O}(1)$ incomplete phrases per parsing level i, there are $\mathcal{O}(\log m)$ cuts in total.

The time obtained [34] is $O(m + (occ + 1)\log^{\epsilon} n)$ for any constant $\epsilon > 0$, but the index is complicated and likely much larger than the regular one.

3.4 BWT Indexes for Labeled Directed Graphs

Using the BWT framework we can produce succinct indexes for labeled directed graphs. A relevant advantage of this approach is that it does not require us to store the graph topology explicitly, only the labels plus some auxiliary bit vectors of length proportional to the number of edges. In BWT-based indexes for graphs, the time of the navigational operations have a $\log \sigma$ slowdown factor as they build on the LF operation. In addition, we can search for the occurrences of a pattern $P[1, m]$ in the graph paths in $\mathcal{O}(m \log \sigma)$ time using backwardsearch. These features have made BWT-based indexes popular in Genomics. Recall that the DNA alphabet is very small, so the $\log \sigma$ factor is negligible in practice for genomic data. Before explaining the ideas, we give some relevant definitions.

Let $G = (V, E, \Sigma)$ be a directed labeled graph. V is the set nodes, E is the set of edges, and $\Sigma = [1, \sigma]$ is the alphabet of edge labels. The direction of every edge $(u, v) \in E$ is from node u to node v. The operator $l(u, v) \in \Sigma$ denotes the label of (u, v) and the operator $\prec$ represents the ordering between the symbols in Σ. A path in G is a sequence of nodes $v_i, v_{i+1}, \ldots, v_{i+k}$ such that the edges $(v_i, v_{i+1}), (v_{i+1}, v_{i+2}), \ldots, (v_{i+k-1}, v_{i+k})$ exist in E. Indexing G using the BWT framework described in previous sections requires G to be a *Wheeler graph* [69]:

Definition 1 G is a Wheeler graph if there is an ordering of the nodes such that nodes with in-degree 0 precede those with positive in-degree and, for any pair of edges (u, v) and (u', v'), labeled with a and a' respectively, the following monotonicity properties hold:

$$a \prec a' \Rightarrow v < v'$$
$$(a = a') \wedge (u < u') \Rightarrow v \leq v'.$$

Definition 2 G is path coherent if there is a total order of nodes such that for any consecutive range $[i, j]$ of nodes and a string S, the nodes reachable from those in $[i, j]$ in $|S|$ steps by following edges whose labels form S when concatenated, themselves form another range $[i', j']$.

If G is a Wheeler graph, then it is path coherent (Lemma 3 [69]). This feature is relevant

Function	Description	Time Complexity
$\mathsf{outdegree}(v)$	number of outgoing edges of v	$\mathcal{O}(1)$
$\mathsf{indegree}(v)$	number of incoming edges of v	$\mathcal{O}(1)$
$\mathsf{outneighbor}(v, k)$	kth outgoing neighbor of v in the graph ordering	$\log \sigma$
$\mathsf{inneighbor}(v, k)$	kth incoming neighbor of v in the graph ordering	$\log \sigma$
$\mathsf{find}(P[1, m])$	report all paths in G prefixed by P	$\mathcal{O}(m \log \sigma)$

Table 3.1: Queries supported by BWT graph indexes

to support queries over G using the LF and $\mathsf{backwardsearch}$ operations. The problem is that determining if G belongs to the Wheeler class is NP-complete for any edge label alphabet of size $\sigma > 2$ [75]. Although the formal characterization of a Wheeler graph is relatively new [69], the idea is not. Several BWT-based indexes in the literature [172, 63, 24] are representations for graphs that belong to the Wheeler class. We briefly describe two such indexes: for *labeled tries* [63] and *directed acyclic graphs* [175]. They are the basis for the genomic representations we develop in later chapters. Table 3.1 summarizes the common queries they support and their time complexities.

3.4.1 Labeled Tries

Let $T = (V, E, \Sigma)$ be a cardinal labeled trie, where every node $v \in V$ is labeled with a symbol $l(v) \in \Sigma$. The special character # is the smallest one in Σ, and only labels the root of T, denoted as r'. Each edge $(u, v) \in E$ is directed from the child node u to its parent v. We use the operator $\bar{l}(v)$ to refer to the string resulting from the concatenation of the labels in the path $v_i, v_{i+1}, \ldots, v_{i+k}$, where $v_i = v$ and $v_{i+k} = r'$. We call $\bar{l}(v)$ the *extended label* of v. Note that, unlike Wheeler graphs, the labels of T are in its nodes, not in its edges. In the particular case of T, this difference does not have a relevant effect on the model. We can assume that a node's label is associated with the edge that connects the node with its parent.

To build a BWT index for T, we proceed as follows; we define an empty list Q and start a preorder traversal over T. For every edge $(u, v) \in E$ we visit, we append the triplet $(l(u), \bar{l}(v), \texttt{int})$ into Q. The field `int` is a bit flag set to 0 if u is a leaf or set to 1 otherwise. Once we scan T, we stably sort Q according the lexicographical order of the second components. After the sorting, the information of every internal node v now lies in a specific range $Q[i, j]$ of $c = j - i + 1$ consecutive triplets, where c is the number of children of v. We create a bit vector $B[1, |Q|]$ in which we set $B[j] = 1$ for every different range $Q[i, j]$. We also create a bit vector $I[1, |Q|]$ to concatenate the `int` flags and a list $L[1, |Q|]$ to concatenate the labels in the first components of Q. In both lists I and L, we insert the elements in the same order as they appear in Q. Similarly as with the matrix M in the BWT, we do not need to explicitly store the second components of Q (the extended node labels). Instead, we create a vector $C[1, \sigma]$, where each $C[c]$ stores the number of distinct extended labels that are lexicographically smaller than symbol c. The BWT index of T (denoted XBW-transform in [63]) is thus conformed by B, I, L and C.

We identify every internal node $v \in V$ of T in the XBW-transform using the pair $[i, j]$ that represents the location of v in the index. Recall that $L[i, j]$ has the labels of its incoming edges, and $I[i, j]$ marks which of its children are leaves. We assume that we always know the range $[i, j]$ for v when performing queries in the XBW-transform. Notice the leaves do not have an associated pair $[i, j]$ as they do not have incoming edges. This difference is not a problem in practice. The most relevant queries for a leaf u are accessing its label and moving to one of its siblings or to its parent. If we know the range $[i, j]$ of u's parent, we obtain its label by inspecting $L[i, j]$. Similarly, if we know that $L[i']$ is the label of u, then we obtain the range $[i, j]$ of its parent using B. We need to find the greatest $i' < i$ such that $B[i' - 1] = 1$ and the smallest $j \geq i'$ such that $B[j] = 1$.

Implementing the function $\mathsf{outdegree}(v)$ is not necessary as all the nodes have only one outgoing edge, the parent. On the other hand, the function $\mathsf{indegree}(v)$ is just $j - i + 1$.

As anticipated, moving top-down in T is implemented using a variation of the LF function. Suppose we know the range $L[i, j]$ of the internal node v, and we want to know the range $[i', j']$ of the kth child of v, for $k \leq j - i + 1$. This child, say u, is encoded at position $q = i + k - 1$, and its label is $l(u) = L[q] = c$. If $I[q] = 0$, we return an invalid range $[0, 0]$ to indicate u is a leaf. If not, then we compute its corresponding range $[i', j']$ with the formula:

$$\begin{aligned} b &= C[c] + \mathsf{rank}_c(L, q) \\ i' &= \mathsf{select}_1(B, b - 1) + 1 \\ j' &= \mathsf{select}_1(B, b). \end{aligned}$$

The value b is the lexicographical rank of $\bar{l}(u) = c \cdot \bar{l}(v)$ among the other extended labels of T. It indicates that the information of u is stored in the bth range of L. Thus, we compute the boundaries of u in L using the select_1 operations over b. This top-down navigation is equivalent to the function $\mathsf{inneighbor}(v, k)$ of Table 3.1.

Now let us perform the inverse procedure; moving from the index position $[i', j']$ of u to the index position $[i, j]$ of its parent v. The idea is as follows; we first obtain the lexicographical rank of the extended path of u as $r = \mathsf{rank}_1(B, j')$. We perform a binary search over C to find the position $c = l(u)$ such that $C[c] < r \leq C[c + 1]$. We then compute the value $b' = \mathsf{rank}_1(B, j') - C[c]$, which is the lexicographical rank of $\bar{l}(u)$ among the other extended paths of T prefixed by c. Subsequently, we get the BWT position $q = i + k - 1 = \mathsf{select}_c(L, b')$. The value of k means that u is the kth child of its parent. From q, we obtain the greatest $i \leq q$ such that $B[i - 1] = 1$ and the smallest $j \geq q$ such that $B[j] = 1$. The summary of the steps is as follows;

$$\begin{aligned} b' &= \mathsf{rank}_1(B, j') - C[c] \\ q &= \mathsf{select}_1(L, b') \\ b &= \mathsf{rank}_1(B, q) \\ i &= \mathsf{select}_1(B, b - 1) + 1 \\ j &= \mathsf{select}_1(B, b). \end{aligned}$$

We adapt the $\mathsf{backwardsearch}$ procedure to implement the function $\mathsf{find}(P[1, m])$ of Ta-

ble 3.1. Let $L[s, e]$ be the range in the backward search step for suffix $P[x, m]$. We start with the range $L[1, |V|]$ for $x = m + 1$ and then, for $x = m, \ldots, 1$, compute the new range $L[s', e']$ for $P[x - 1, m]$, with $p = P[x - 1]$, as:

$$s' = C[p] + \mathsf{rank}_p(L, s - 1) + 1$$
$$e' = C[p] + \mathsf{rank}_p(L, e)$$
$$s' = \mathsf{select}_1(B, s' - 1) + 1$$
$$e' = \mathsf{select}_1(B, e').$$

As in previous BWT indexes, we store L as a wavelet tree (Section 2.3) to support rank_c and select_c in $\mathcal{O}(\log \sigma)$ time. In addition, we augment B with rank_1 and select_1 data structures to traverse T. Thus, the total space usage of the BWT-based index for T is $2n + n(\mathcal{H}_0 + 1)(1 + o(1)) + \mathcal{O}(w\sigma)$ bits of space, where $n = |V|$. This space is small compared to the $\mathcal{O}(n \log n + n \log \sigma)$ bits of a pointer-based representation.

3.4.2 Directed Acyclic Graphs

In a labeled trie, navigational queries are simple as all the nodes have only one parent (i.e., they have out-degree one). In contrast, in a labeled directed acyclic graph (DAG), the nodes can have more than one outgoing edge, which invalidates the LF formula. We solve this problem by adding a bitmap that encodes the out-degree of the nodes. We follow the nomenclature of Mäkinen et al. [120] to explain the idea.

Let $G = (V, E, \Sigma)$ be a labeled DAG, where each edge $(u, v) \in E$ is directed from u to v, and it is labeled with a symbol in $\Sigma = [1, \sigma]$. We assume G has a source node s and a sink node t. Let $l(v_1, v_2, \ldots, v_k)$ be the concatenation of the edge labels in the path $v_1, v_2, \ldots, v_k$ of G. The operator $\bar{l}(v)$ denotes all the path labels $\{l(v_1, v_2, \ldots, v_k) \mid v_1 = v,\ v_k = t,\ (v_i, v_{i+1}) \in E,\ i \in [1, k - 1]\}$. For G to be indexed using the Wheeler framework, it must meet the following properties:

- The outgoing edges of s are labeled with $\$ = \sigma$, the greatest symbol in Σ. We also assume t has an artificial outgoing edge labeled with $\# = 1$, the smallest symbol in Σ.
- The nodes are *reverse deterministic*; the incoming edges of every $v \in V$ have distinct labels.
- The nodes are *strongly distinguishable*; all the strings in $\bar{l}(v)$ are prefixed by some (maximal) string $P_v \in [1, \sigma]^+$ and there is no other node $x \in V$ such that P_v is also a prefix in $\bar{l}(x)$. We refer to P_v as the distinguishable prefix of v.

A DAG G' may not meet these conditions, but we can modify it so that it does. We make G' reverse deterministic using the classical powerset construction algorithm for determinizing finite automata (see Section 9.6.3 in Mäkinen et al. [120]). Additionally, we can adapt the prefix-doubling technique for suffix array construction to make G' strongly distinguishable (see Section 6 of Sirén et al. [175]). We now describe the procedure to build the BWT-index for G.

We create a list Q with all the possible pairs $(l(u, v), P_v)$ of G, where u is one of the incoming nodes of v. We stably sort Q according the lexicographical rank of the strings

in the second component of the pairs. As in the previous section, the sort step places the information of v in a contiguous range $Q[i, j]$. We create a bit vector $B[1, |Q|]$ to mark the last element $B[j] = 1$ of every distinct $v \in V$. In addition, we create another bit vector O that will encode the nodes' out-degrees. For each distinguishable prefix P_v we see in Q (from left to right), we append the sequence of bits 10^{o-1} to O, where o is the out-degree of v in G. Finally, we create the array $L[1, |Q|]$ with the first components of Q and the array $C[1, \sigma]$. In this case, $C[c]$ stores the number of edges labeled with symbols lexicographically smaller than symbol c. The final BWT-index for G is composed of B, O, L and C.

The navigational queries for the BWT index of G are rather similar to those of Section 3.4.1. We implement $\mathsf{indegree}(v)$ in constant time as $j - i + 1$. We can also implement $\mathsf{outdegree}(v)$ in constant time using the formula $\mathsf{select}_1(O, b_v + 1) - \mathsf{select}_1(O, b_v)$, where $b_v = \mathsf{rank}_1(B, j)$ is the lexicographical rank of P_v among the distinguishable prefixes of G. To implement $\mathsf{inneighbor}(v, k) = u$, we slightly modify the LF operation. Let u be the kth incoming node of v encoded at position $q = i + k - 1$, and with label $l(u, v) = L[q] = c$. We obtain the range $[i', j']$ for u with the following formula:

$$
\begin{aligned}
b_u &= \mathsf{rank}_1(O, C[c] + \mathsf{rank}_c(L, q)) \\
i' &= \mathsf{select}_1(B, b_u - 1) + 1 \\
j' &= \mathsf{select}_1(B, b_u).
\end{aligned}
$$

Note that $C[c] + \mathsf{rank}_c(L, q)$ does not necessarily gives us the lexicographical rank of $P_u = cP_v$. If u has out-degree o, then there is some range in L, where there are o consecutive occurrences of c, and all of them lead us to u. To find the correct lexicographical rank b_u of P_u, we use the operation $\mathsf{rank}_1(O, C[c] + \mathsf{rank}_c(L, q))$. The rest of the formula above is equivalent to moving top-down in the XBW-transform.

The operation $\mathsf{outneighbor}(u, k) = v$ is a bit more elaborate. Suppose we know for u the lexicographical rank b_u of P_u and its range $L[i', j']$. We first perform a binary search over C to find the symbol c such that $C[c] < i' \leq C[c + 1]$ (i.e., the prefix of $P_u = cP_v$). Subsequently, we perform the successive steps:

$$
\begin{aligned}
x &= \mathsf{select}_1(O, b') + k - 1 - C[c] \\
b_v &= \mathsf{rank}_1(B, \mathsf{select}_c(L, x)) \\
i &= \mathsf{select}_1(B, b_v - 1) + 1 \\
j &= \mathsf{select}_1(B, b_v).
\end{aligned}
$$

The operation $\mathsf{select}_c(L, x)$ gives us the position q in L storing the label of edge (u, v). After computing q, we obtain the lexicographical rank b_v of P_v, and then $[i, j]$.

The implementation of $\mathsf{find}(P[1, m])$ in the DAG index is almost equal to that of the labeled trie. The only difference is that now we have to consider the array O. Suppose the previous backward search step yielded the range $L[s', e']$, and now we have to compute the

next step using symbol c. The successive steps are:

$$s' = C[c] + \mathsf{rank}_c(L, s' - 1) + 1$$
$$e' = C[c] + \mathsf{rank}_c(L, e')$$
$$s' = \mathsf{select}_1(B, \mathsf{rank}_1(O, s') - 1) + 1$$
$$e' = \mathsf{select}_1(B, \mathsf{rank}_1(O, e')).$$

3.5 Algorithms for building the SA and the BWT

The design of algorithms for building the suffix array and the BWT is a relevant topic in stringology as these structures are the main components in several succinct self-indexes. There are several methods proposed in the literature that are efficient in terms of time or space (see, for instance, [98, 147, 10, 132]). However, the hidden constants in their complexities are too high for practical applications on massive collections.

A recent trend in the computation of the BWT is to exploit the text redundancies [22, 93, 94]. The general idea consists of factorizing S to create a dictionary D of phrases. Subsequently, we sort D in some specific order and then extrapolate the results to the whole text. If the text is repetitive enough, then D should be small, and computing the BWT should be efficient.

In Section 3.5.1, we describe an algorithm for building the BWT that uses this approach. Section 3.5.2 describes a general-purpose linear-time algorithm for computing the BWT and the suffix array that can also be adapted to use this idea. This latter method is relevant for the thesis as it has applications in the compression of sequencing reads and the production of locally consistent grammars (Section 3.2.2).

3.5.1 Prefix-Free Parsing

Prefix-free parsing (PFP) [22] is a linear-time procedure that transforms an input text $S[1, n]$ into a sequence P of overlapping prefix-free phrases. The set D with the distinct phrases is referred to as the *dictionary* while P is referred to as the *parse* of S. Consecutive phrases in P overlap by x characters, where x is an input parameter. The strings in the dictionary are sorted in lexicographical order, and the phrases in P are replaced by their ranks in the dictionary. We now describe how to build D and P, and then we explain how to use them to boost the computation of the BWT.

To perform PFP, we first create a new input string $S' = \#S\$^x$, where # and \$ are symbols lexicographically smaller than any character in S. These values are appended to S to avoid border cases. Subsequently, we choose a hash function for strings of length x and a prime number p. We roll the hash over S' (Section 2.3.2), and every time we find a substring $S'[i, j]$ of length x whose fingerprint modulo p equals 0, we consider it as a *trigger* of a new phrase. If $S'[i', j']$, with $i', j' < i$, was the previous trigger in S', then the new phrase in P is $S'[i', j]$. As we move on through S', we also hash the phrases to build D. Once we finish the scan, we sort D in lexicographical order and replace the phrases in S' with their ranks in D.

Let $Z \in D$ be a phrase and let $A = Z[u..]$ be a string of length $> x$ that only occurs as

a proper suffix in Z. If Z has z occurrences in S, then the suffixes of S prefixed by A form a range $SA[i,j]$ of length z in the suffix array. Further, as A only appears in Z, the BWT range $L[i,j]$ is an equal-symbol run of length $j-i+1$ for $Z[u-1]$. Using this observation, we can infer several segments of the BWT of S directly from D. Still, there are some situations we cannot handle. For instance, if A were a non-proper suffix of Z (i.e., $Z = A$), we could not access its preceding symbol from Z, and hence, we could not know the symbol for $L[i,j]$.

Another situation we cannot handle just with D is when a string that appears as a suffix in two or more dictionary phrases has different left contexts. Suppose A is a suffix in two phrases U and Y and the symbols that precede A in U and Y (a_u and a_y, respectively) are different. If U occurs o_u times in S and Y occurs o_y times, then the suffixes of S prefixed by A form a range $SA[i,j]$ of length $o_u + o_y$. Although we know that the corresponding $L[i,j]$ contains o_u copies of a_u and o_y copies of a_y, we cannot infer their relative orders.

We handle those situations using P's BWT. Recall that every distinct symbol b in P represents a specific dictionary phrase $B \in D$. Hence, the disposition of the occurrences of b in P's BWT also represents the relative order of the occurrences of B in S' when sorted according the lexicographical ranks of the suffixes that follow them. Now let us return to the example in the previous paragraph. In one scan of P's BWT, we obtain the relative order of the occurrences of U and Y to get the relative order of symbols a_u and a_y in $L[i,j]$. We know briefly discuss how to implement these ideas in an algorithm for computing the BWT L.

We first obtain the PFP of S'. Subsequently, we sort the distinct suffixes in D of length $> x$ in lexicographical order. For every suffix A in D of length $> x$, we create a pair (f,a). The value f is the cumulative frequency of the phrases where A occurs, and a is the preceding symbol of A in those phrases. When A is not a proper suffix or has more than one distinct left context in D, we replace a with a placeholder # that we will fill later. We store the pairs in a list L', sorted according to the lexicographical ranks of the suffixes from which they were generated. More specifically, if A has rank r among the distinct suffixes of length $> x$, then (f,a) is the rth pair in L'.

Let us denote L_P the BWT of P to differentiate it from L. We can induce a partition over L_P so that the bth block stores the preceding symbols of the suffixes in P prefixed by $b \in [1, |D|]$. This partition allows us to fill the placeholders in L. Let $L_P[i,j]$ be the BWT range storing the symbols preceding b in P. We create an empty list L^b and start a scan over $L_P[i,j]$ from left to right. For every symbol $L_P[k]$, with $k \in [i,j]$, we retrieve its associated phrase $F \in D$ and append F's last symbol to L^b. After scanning $L_P[i,j]$, we map b to its phrase $B \in D$ and retrieve the rank r of B among the distinct suffixes in D of length $> x$. Finally, we replace the rth pair in L' with L^b. Notice that the pair that L^b replaced in L' had a placeholder. We can run-length compress L^b so it matches the format of the pairs in L'.

We fill the remaining placeholder positions in L' with another linear scan of L_P –although we could use the same scan of the previous paragraph. We first create a list L^A for every distinct proper suffix A of D with two or more different left contexts. We also need a mechanism to keep track of the occurrences of A in the distinct dictionary phrases. Subsequently, we scan L_P from the left; if the phrase $F \in D$ that maps the current character $L_P[i]$ has an

occurrence of $A = F[j..]$, then we append the symbol $F[j-1]$ to L^A. The important aspect of this scan is to maintain the BWT order of the symbols we insert into L^A. After we finish the scan, we replace A's pair $(f, \#)$ in L' with L^A. As before, we can run-length compress L^A to match the format. The resulting list L' is a partially run-length compressed version of S's BWT.

The space and time complexity of this BWT algorithm is proportional to the size of D and P. If S is repetitive enough, then one would expect these values to be small. Of course, the parse size does not depend only on the text. It also depends on the values we choose for x and p.

3.5.2 Induced Suffix Sorting

Induced suffix sorting (ISS) [98] is a technique that computes the lexicographical ranks of a subset of suffixes in S and then uses the result to *induce* the order of the rest. This method is the underlying procedure in several algorithms that build the suffix array [141, 140, 117] and the BWT [147, 22] in linear time. The ISS idea introduced by the suffix array algorithm SA-IS [141] is of interest to this thesis. The authors give the following definitions:

Definition 3 A character $S[i]$ is called L-type if $S[i] > S[i+1]$ or if $S[i] = S[i+1]$ and $S[i+1]$ also L-type. On the other hand, $S[i]$ is said to be S-type if $S[i] < S[i+1]$ or if $S[i] = S[i+1]$ and $S[i+1]$ is also S-type. By default, symbol $S[n]$, the one with \$, is S-type.

Definition 4 A character $S[i]$, with $i \in [1, n]$, is called leftmost S-type, or LMS-type, if $S[i]$ is S-type and $S[i-1]$ is L-type.

Definition 5 A LMS substring is (i) a substring $S[i, j]$ with both $S[i]$ and $S[j]$ being LMS characters, and there is no other LMS character in the substring, for $i \neq j$; or (ii) the sentinel itself.

SA-IS is a recursive approach. In every recursion level i, we first scan the input text S^i, with $S^1 = S$, from right to left to classify its suffixes as L-type, S-type or LMS-type. As we move through the text, we record the positions of the LMS substrings. Then, we sort the LMS substrings using ISS as follows; we create an array $A^i[1, n]$ and logically divide it into σ^i buckets, one for the suffixes starting with each symbol in $\Sigma^i = [1, \sigma^i]$ (the alphabet of S^i). Each bucket is, in turn, divided in two sub-buckets, the first one, the L-bucket, is for the suffixes prefixed by L-type characters and the second one, the S-bucket, is for the suffixes prefixed by S-type characters. Then, we perform the following operations:

1. We insert the positions of the LMS substrings at the end of the S-buckets in A^i. In every one of these buckets, we maintain the order of the LMS strings as they appear in S^i. The S-bucket of an LMS string $S^i[j, j']$ is that of the bucket of symbol $S^i[j]$.
2. We scan A^i from left to right and for every j such that $S^i[A^i[j] - 1]$ is L-type, we insert the index $A^i[j] - 1$ in the leftmost available position of the L-bucket of symbol

$S^i[A^i[j]-1]$. After the scan, the elements in all the S-buckets are discarded.

3. We scan A^i from right to left and for every j such that $S^i[A^i[j]-1]$ is S-type, we insert the index $A^i[j]-1$ in the rightmost available position of the S-bucket of $S^i[A^i[j]-1]$.

This procedure sorts the LMS substrings in a way that is slightly different from lexicographic ordering. In particular, if an LMS substring $S[a,b]$ is a prefix of another LMS substring $S[a',b']$, then $S[a,b]$ gets higher order. However, the higher rank of $S[a,b]$ implies that the suffix $S[a..]$ is lexicographically greater than the suffix $S[a'..]$. The cause of this property is explained in Section 2 of Ko and Aluru [98].

After finishing the procedure, we still have to calculate the relative order of the suffixes that start with the same LMS substring. For that purpose, we create a new string S^{i+1} in which we replace the LMS substrings with their lexicographical ranks and use this new string as input for another recursive call $i+1$ of SA-IS. The base case for the recursion is when all the suffixes in A^i are prefixed by different symbols, in which case we return A^i without further processing.

When the $(i+1)$th recursive call ends, the suffixes of S^i prefixed by the same LMS substrings are completely sorted in A^{i+1}. Therefore, we are ready to induce the order of the rest of the suffixes. For doing so, we reset A^i and repeat the same ISS procedure. The only difference is that in step 1 we put the LMS-substrings at end of the S-buckets of A^i arranged as they appear in A^{i+1}. Step 2 and 3 are executed without changes. Once we complete all the recursive calls, the suffix array of S is in A^1.

Steps 1, 2 and 3 take time proportional to the size of S^i. In addition, every time we enter a new recursion step $i+1$, the length of its input text S^{i+1} is at most half the size of S^i. This feature implies that, if we consider all the S^i from all the recursive steps, their lengths do not sum more than $2n$ characters. As a consequence, the running time of SA-IS is $\mathcal{O}(n)$. On the other hand, the array A^i dominates the space complexity. As every cell requires $\log n$ bits, and all the A^i arrays do not sum more than $2n$ cells, the working space of SA-IS is $\mathcal{O}(n \log n)$ bits.

Chapter 4

Computational Genomics

Computational Genomics is an interdisciplinary field that uses computational and statistical methods to study how genome sequences control biological processes. In recent years, it has become an essential means for biological discovery due to the abundance of collections of DNA strings. Still, a remaining problem is how to efficiently manipulate those collections in the computer. In most cases, they are too massive to use conventional algorithms and data structures. In this chapter, we review the main techniques used in Computational Genomics to process DNA strings.

4.1 DNA Sequences

A *DNA sequence* S is a string over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{t}, \mathtt{g}, \mathtt{n}\}$ (which we map to $[2, \sigma]$). The symbols in Σ represent the distinct *nucleotides* that conform DNA. The only exception is `n`, which usually denotes an unknown nucleotide. DNA is *double stranded*, meaning that there are two possible sequences for the same molecule, one for each strand. However, these sequences are complementary; every time we see an `a` at some position in one strand, we see a `t` at the same position in the other, and vice-versa. The same applies for `c` and `g`. The DNA strands have different orientations, and so have their sequences. One sequence is read from left to right (the forward strand), and the other is read from right to left (the reverse strand). For simplicity, we store the sequence of one strand in the computer, as we can easily infer the other from the information we already have. An example of a DNA sequence is shown in Figure 4.1A.

Formally, the DNA *complement* is a permutation $\pi[2, \sigma]$ that reorders the symbols in Σ exchanging `a` with `t` and `c` with `g`. The *reverse complement* of S, denoted S^{rc}, is a string transformation that reverses S and then replaces every symbol $S[i]$ by its complement $\pi(S[i])$. For technical convenience we add to Σ the so-called *dummy* symbol `$`, which is always mapped to 1.

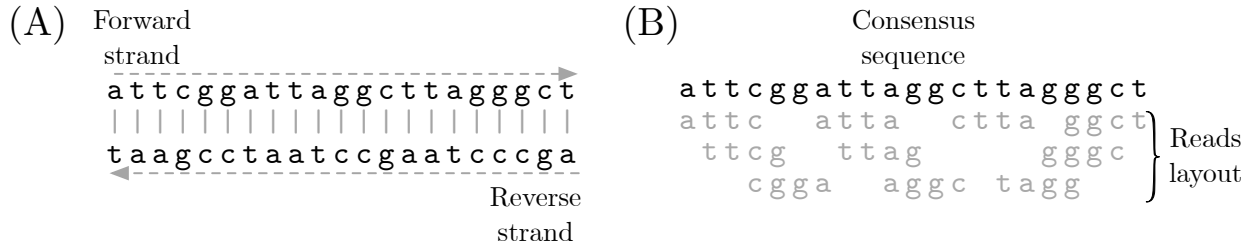

Figure 4.1: DNA strings and sequencing. (A) Schematic representation of a double-stranded DNA molecule. Forward strand (upper sequence) is read from left to right, while the reverse strand (lower sequence) is read from right to left, and corresponds to the reverse complement of the forward. (B) Sequencing experiment of the DNA molecule of Figure (A). Gray sequences are the reads. The layout is inferred via suffix-prefix overlaps between the reads.

4.2 DNA Sequencing

Sequencing consists of determining the order of the nucleotides in a DNA sample. Although several methods have been proposed over the years for this task [157], the most successful ones have been *next generation sequencing* (NGS) techniques [168]. They are cheap, fast, and produce high-quality data. There are several NGS platforms, but the most popular one is Illumina [85]. This method breaks the source DNA into smaller fragments and introduces them into a machine (a.k.a., the sequencer) that reads their nucleotides in parallel. The collection of fragments is usually known as the *library*. There are two types of libraries, *single-end*, and *paired-end*. In the former, the sequencer produces one string per fragment. In the latter, it creates two; one for the left end and the other for the reverse complement of the right end. The strings obtained from the fragments are referred to as the *reads*.

A major problem with NGS sequencers is that they cannot process long molecules. The number of nucleotides they can scan from a single fragment ranges from a few dozens up to a few hundred. Still, complex genomes, such as those of mammals, are several billion nucleotides long. NGS technologies solve this problem by producing overlapping read collections. More precisely, they use multiple copies of the source DNA for the library construction, break those copies at random, and sequence the resulting fragments. Then it is necessary to *assemble* the reads to infer the sequence of the source DNA. The idea of producing overlapping reads is also helpful to deal with sequencing errors, i.e., when the sequencer emits an incorrect nucleotide. If a read has an error, we can fix it using the overlapping reads that contain the correct information.

In recent years, a new generation of sequencing technologies has emerged as an alternative to NGS, the so-called *third-generation* platforms [185]. They produce much longer reads than NGS, although their throughput is smaller and their accuracy is still lower. However, these limitations should be solved soon. Recently, the company PacBio [17] presented its Hi-Fi protocol [190], which produces long reads comparable in accuracy to Illumina's. On the other hand, the company Nanopore [180] offers small and affordable sequencers that can yield volumes of data even more significant than those of Illumina and with much longer reads. Although the accuracy of Nanopore is still not as good as that of Illumina or PacBio Hi-Fi. Table 4.1 describes the most popular sequencing technologies nowadays.

Platform	Instrument	Accuracy	Median read length	Maximum read length	Throughput per run	Run time
Illumina	MiSeq	99.9%	2×300B	2×300B	15 GB	Up to 55 h
	NextSeq	99.9%	2×150B	2×150B	120 GB	Up to 30 h
	NovaSeq	99.9%	2×250B	2×250B	6 TB	Up to 44 h
PacBio	Sequel II	88% - 99.9%	45KB - 190KB	300 KB	20–50 GB	Up to 96 h
Nanopore	MinION	97.5% - 98.3%	Variable	Variable	Up to 42 GB	Up to 72 h
	GridION	97.5% - 98.3%	Variable	Variable	Up to 210 GB	Up to 72 h
	PromethION	97.5% - 98.3%	Variable	Variable	Up to 11.7 TB	Up to 72 h

Table 4.1: Comparative table with the different sequencing technologies. The accuracy is reported as the percentage of nucleotides in the source DNA with a high probability of being correctly sequenced (Phred score). PacBio reads with 99.9% of accuracy are those produced with the Hi-Fi protocol. Throughput and read lengths are reported in bytes (B), megabytes (MB), gigabytes (GB) or terabytes (TB). The data in this table was extracted from the official webpages of Illumina, PacBio and Nanopore.

DNA sequencing is not a linear problem. In most of the cases, the source DNA is a set of molecules that have highly similar sequences. The origins of these molecules are varied, for example, the many cells from which the DNA was extracted[1], the different copies of the same *chromosome*[2], or individual genomes of the same species that were sequenced all together. In such cases, the desired result is not a string, but an labeled directed acyclic graph (DAG) representing the highly similar sequences as paths in a graph. Figure 4.2A depicts the idea.

It is hard to tell if a variation is real or if it is a sequencing error just by looking at the reads, but some heuristics can be applied to make an educated guess. However, other difficulties cannot be addressed only by using string queries. An example of these difficulties are the gaps in the coverage, i.e., places in the source DNA that are not covered by any read. From a biological point of view, the repetitiveness is also a problem; if a given substring in the source DNA is a *long repeat* bigger than any read, then it is not always possible to decide which is the correct ordering of sequences surrounding it.

4.2.1 Sequencing File Format

Reads are typically stored as `FASTQ` files. This format uses four lines per read. The first line is the identifier of the read. The second line is the DNA sequence in `ASCII` code. The third line is a separator, denoted by `+`, and the fourth line is a string encoding the sequencing qualities (Phred score) of the nucleotides spanned by the read. This string is also encoded in `ASCII`. When the library is paired-end, the reads of the same pair are consecutive in the file. Figure 4.3 shows a typical entry of a `FASTQ` file.

[1] DNA mutates at random and independently in every cell.

[2] Polyploid genomes contain two or more copies of the same chromosome.

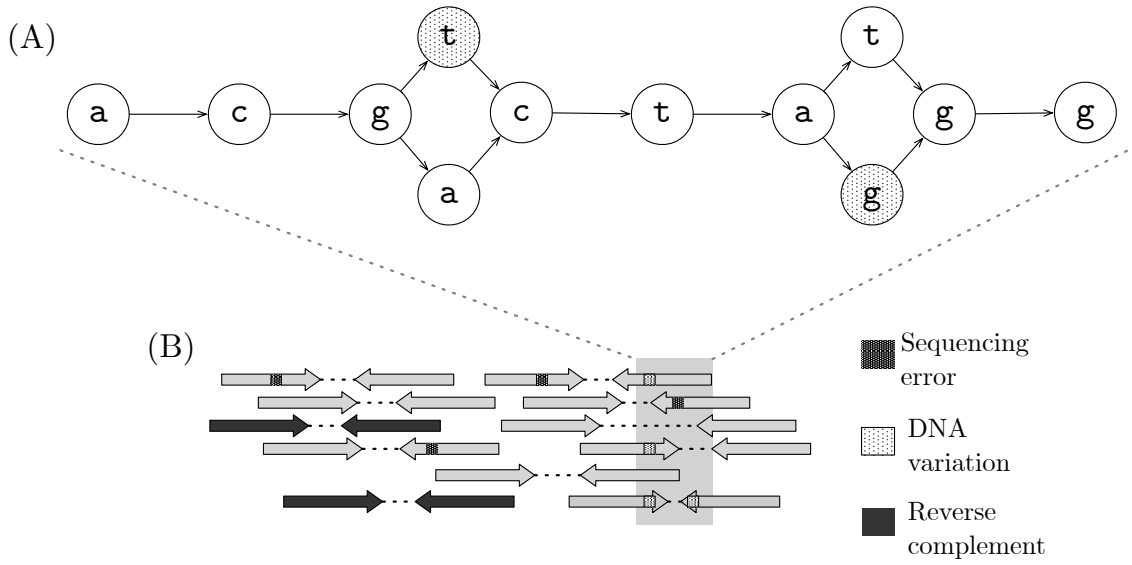

Figure 4.2: A paired-end library. (A) A segment of the source DNA of the reads represented as a labeled DAG. Dotted circles denote variable positions. (B) Read collection generated from an NGS experiment. Every arrow represents a read. Dashed lines connect reads that belong to the same pair. Gray pairs were obtained from the the forward strand of the source DNA, so we have to transform the right read into its reverse complement. Conversely, black pairs were generated from the the reverse strand, so we transform the left read and invert the pair order; the left read becomes the right read and vice-versa. The gray rectangle in the layout shows how the DAG of (A) looks like in the reads.

4.3 The de novo Assembly Problem

The typical analogy for the *de novo* assembly problem is that of a puzzle in which the reads are the pieces, and the source DNA is the picture we have to reconstruct. The process consists of estimating the disposition of reads across the genome (a.k.a., the *layout*), and then obtain the sequence of the source DNA by collapsing the reads (Figure 4.1B depicts the idea). The classical way of calculating the layout is by computing suffix-prefix overlaps. To be precise, for every read S_i, we find other reads in the collection with prefixes that match the suffixes of S_i. We do not have any prior information about the possible genomic location of S_i, so we have to compare its sequence against the sequence of all[3] the other reads. This task requires a quadratic number of suffix-prefix alignments, making the layout's calculation expensive in practice. We also have to consider the reverse complements of the reads when computing the overlaps. Recall that we do not know their original strands. This problem doubles the number of alignments we have to perform, making the process even more challenging.

The de novo assembly problem is usually centered in the reconstruction of genomes. Although this definition is not strict; we can also assemble the set of transcripts being expressed in a cell (*transcriptome*) or the collection of microbial genomes that live in the same environment (*metagenome*). In practice, however, the algorithmic techniques do not vary much. For simplicity, we further explain the de novo assembly problem in terms of *diploid genomes*[4]

[3]When the read collection is paired-end, we can discard the overlaps between S_i and its pair.

[4]Each chromosome in the genome has two copies.

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATC
+
!''*((((***+))%%%++)(%%
```

Figure 4.3: Read entry in `FASTQ` format.

as this is the case of humans.

Given the limitations of sequencing technologies (see Section 4.2), it is impossible to reproduce the reads' exact layout. This problem prevents the complete reconstruction of the source DNA. Most of the programs for assembling reads (a.k.a., *assemblers*) produce strings representing only segments of the chromosomes, the so-called *contigs*. Several aspects limit the size of the contigs. The most important ones are lack of sequencing coverage, sequencing errors, and the repetitive regions of the genome. Reads with sequencing errors are more prone to be misplaced in the layout, while reads produced from repetitive genomic regions have several equally probable positions where to be placed. When the assemblers detect these problems, they just cut the contigs to avoid producing sequences that do not exist in the genome.

Contigs can be further extended to *scaffolds* using the paired-end information of the reads. The rationale is simple: suppose several reads that belong to contig A are paired with several other reads that belong to contig B. In this situation, we can place contigs A and B in one scaffold as it is highly probable that their sequences lie close to each other in the genome. Scaffolds are the best result most NGS assemblers can achieve. One can further join scaffolds into chromosomes using extra molecular information, but assemblers usually do not carry out this procedure. The final genome is a collection R with one string per distinct chromosome. R can also have other smaller sequences representing contigs or scaffolds that could not be assigned to any place.

Some genomic analyses require to produce different strings for the copies of the same chromosome as their nucleotide differences represent important biological information. Still, obtaining such scaffolds is impossible using NGS data alone, the reads are too short. More recent assemblers [189] use third-generation sequencing data to produce near-complete genomes and identify scaffolds that belong to different copies of a same chromosome.

We introduce some notation before continuing with the explanations. Let S_i and S_j be two any strings. The operator $S_i \oplus^o S_j$ denotes an *overlap* between the o-suffix of S_i and the o-prefix of S_j. The *consensus* string of $S_i \oplus^o S_j$ is the sequence $S_i{\cdot}S_j[p+1..]$. Similarly, $Q = S_1 \oplus^{o_1} S_2 \ldots \oplus^{o_{r-1}} S_r$ is a sequence of r consecutive overlaps, and the operator $c(Q) = S_1{\cdot}S_2[o_1+1..] \ldots S_r[o_{r-1}+1..]$ refers to its consensus string.

A *genome graph* $G = (V, E)$ is a directed graph in which every node is labeled with a string that appears in the reads. The operator $l(v)$ refers to the label of node $v \in V$. A directed edge $(u, v) \in E$ from node u to node v exists if there is an overlap $l(u) \oplus^o l(v)$ such that o is equal or greater to some parameter m. Let $P = v_1, v_2, \ldots, v_p$ be a path in G from node v_1 to node v_p. The label $l(P)$ of P is the consensus string obtained from the overlap sequence $l(v_1) \oplus^{o_1} l(v_2) \ldots l(v_{p-1}) \oplus^{o_{p-1}} l(v_p)$.

We spell contigs from G by finding paths whose labels are *probable* to exist in the chromosomes. Recently, Tomescu et al. [181] formalized the concept of "probable". They define a genome reconstruction as a string $l(P)$ whose path P completely covers G[5]. In their edge-centric description of G, P visits all the edges. In the node-centric version, it visits all the nodes. Note that, in both cases, we can produce several genome reconstructions from G, but not all of them are real genome segments. Now, let P' be a path in G. Its label $l(P')$ is said to be *safe* (or probable to occur in the chromosomes), if it appears as a substring in all the genome reconstructions of G. The most basic type of safe string is the label of a unary path; all the nodes in the path, except the first one and last one, have one in-neighbor and one out-neighbor. However, there are other graph structures that also produce safe strings.

Tomescu et al. proposed the concept of *omnitigs* as a way to characterize all the paths of G whose labels are safe. In the edge-centric model, $P' = v_1, v_2, \ldots, v_p$ is an omnitig if for any $1 < i \leq j < p$, there is no proper path from v_j to v_i with first edge different from (v_j, v_{j+1}) and last edge different from (v_{i-1}, v_i) (Definition 5 of [181]). This definition can be easily extended to the node-centric model. Thus, if P' is an omnitig, then $l(P')$ is safe.

There are two types of genome graphs in literature; *de Bruijn graph* (dBG) [46] and *overlap graph* (OG) [133]. Assemblers relying on dBGs ([191, 28, 171, 5]) consume less computational resources than those using overlap graphs. However, they produce more fragmented genomes. Assemblers that use OGs ([134, 192]), on the other hand, although they consume more resources, produce longer and more accurate contigs. In the following, we discuss these two frameworks.

4.3.1 The de Bruijn Framework

A dBG of order k of a string collection $\mathcal{S} = \{S_1, S_2, \ldots, S_q\}$ is a labeled directed graph $G = (V, E)$ where every node $v \in V$ is labeled by a distinct substring of $\mathcal{S}$ of length $k-1$. A directed edge $(v, u) \in E$ exists if the string $l(v) \oplus^{k-2} l(u)$ appears as a substring in at least one element of $\mathcal{S}$. The label of (v, u) is the last symbol of $l(u)$. Figure 4.4 shows an example of a dBG.

To build G, we scan $\mathcal{S}$ and store into a set H the distinct substrings of lenght k (a.k.a., *kmers*). Then, we create a node in G for every distinct substring of length $k-1$ in H. Finally, for each kmer $K \in H$, we create an edge (v, u) in G between the node v labeled with $K[1, k-1]$ and the node u labeled with $K[2, k]$.

There is an interesting link between the dBG and the reads' layout. Suppose a group of reads $S_1, S_2, \ldots, S_r$ in $\mathcal{S}$ form an overlap sequence $Q = S_1 \oplus^{o_1} S_2 \ldots \oplus^{o_{r-1}} S_r$ with every $o_i \geq k-1$. In that case, G will have a path $P = v_1, v_2, \ldots, v_{r'}$ such that $l(P) = c(Q)$. Further, if the $(k-1)$mers in $c(Q)$ do not appear in the other reads of $\mathcal{S}$, then P is a unary path. This last feature allows us to unequivocally spell A from G without having to compute suffix-prefix overlaps between the reads of Q.

The dBG is simple to construct, and allows us to spell contigs without computing suffix-

[5]Their definition considers a circular genome composed of one chromosome. These types of genomes are naturally found in microorganisms.

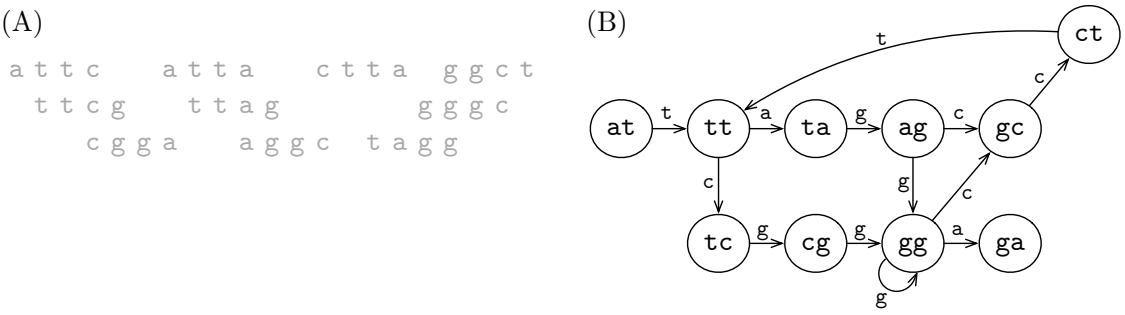

Figure 4.4: The dBG framework. (A) A set of reads disposed according the layout. (B) Example of a dBG with order $k = 3$ produced from the reads of (A).

prefix overlaps between the reads of $\mathcal{S}$. Still, it has some important disadvantages. If some of the $(k-1)$mers appearing in $c(Q)$ also appear in other reads of $\mathcal{S}$, then P gets entangled with other paths of G, and it is no longer possible to unequivocally obtain $c(Q)$. Besides, if there are other reads in $\mathcal{S}$ overlapping strings in Q by less than $k-1$ characters, the paths in G spelling those reads will not be connected to P. Changing the dBG order mitigates the problem, but there is no one single value for k that captures all the valid overlaps of $\mathcal{S}$. If we use a small order, then the graph becomes too tangled, but if we use a high order, the graph becomes too disconnected.

Despite the limitations, dBGs are still a popular solution in Bioinformatics. Their use has been extended from de novo assembly to other genomic tasks such as correction of sequencing errors [164], detection of genetic variations [89] or measurement of gene expression [25].

The BOSS Representation

BOSS [24] is a succinct encoding for dBGs that builds on the idea of Wheeler graphs (Section 3.4). In BOSS, the nodes are represented as rows in a matrix of $k-1$ columns, and are sorted in colexicographical order. All the edge labels (one-symbol) of the graph are stored in a unique sequence L sorted by the BOSS order of the source nodes. This ordering produces the labels of the outgoing edges of each node to fall within a contiguous range in L. A bit vector B of size $e = |L|$ marks the last outgoing symbol in L of every node. We also include a bit vector $I[1, e]$ that encodes in unary the in-degree of the nodes[6]. This in-degree information is stored in I according the BOSS order of the nodes. Finally, an array $C[1, \sigma]$ stores in $C[i]$ the number of edge labels lexicographically smaller than i. The complete index is composed of the vectors L, C, B, and I. Their combined spaces add up to a total of $2e + e(\mathcal{H}_0(E) + 1)(1 + o(1)) + \mathcal{O}(\sigma w)$ bits.

Prefixes in $\mathcal{S}$ of size $d < k$ are artificially represented in BOSS as strings of length k padded at the left with $k-d$ symbols \$. Equivalently, suffixes of size $d < k$ are represented as strings of length k padded at the right with $k-d$ symbols \$. Strings formed only by symbols \$ are also called dummy. The introduction of these extra strings yields a dBG $G' = (V', E')$ with the same order as G, but with more edges and labels.

[6]The original data structure of Bowe et al [24] does not include I. Instead, they mark the edges in L leading us to the same node.

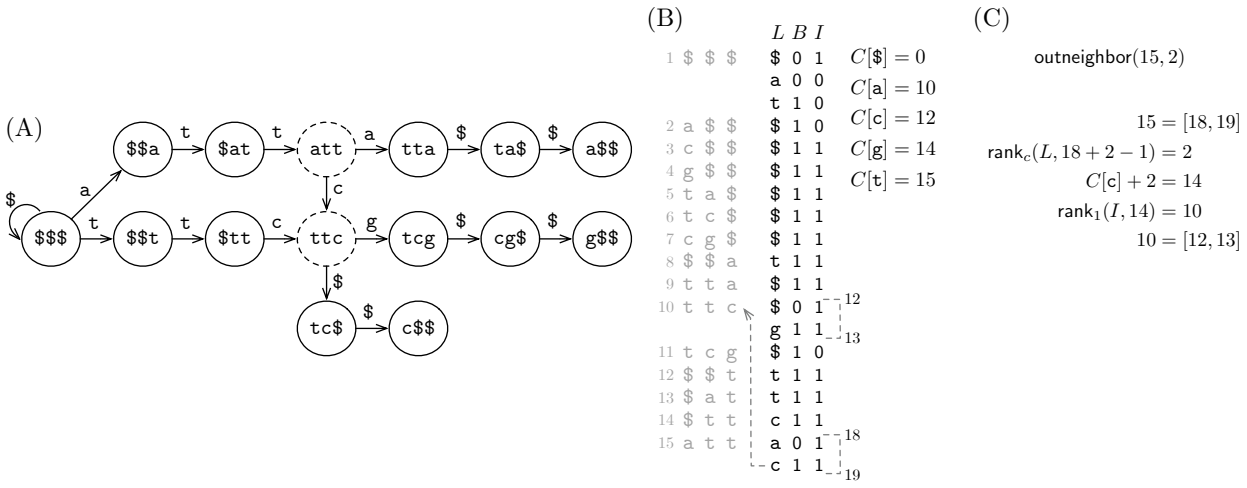

Figure 4.5: Succinct de Bruijn graph. (A) A dBG G' with order $k = 4$ for the string collection {attc, ttcg, atta}. G' includes the dummy strings. (B) BOSS representation for G'. (C) Implementation of the function outneighbor$(15, 2)$ in BOSS using the algorithm for inneighbor in the BWT-index for DAGs (Section 3.4.2). The same operation is depicted in (B) with a dashed arrow, and in (A) with dashed circles.

We identify every node $v \in V'$ in BOSS using the colexicographical order of its label. We use the identifier of v to obtain the range $L[i, j]$ storing the labels of its outgoing edges. If the identifier of v in BOSS is r, then its range in L is $i = \mathsf{select}_1(B, r - 1) + 1$, $j = \mathsf{select}_1(B, r)$. We can navigate G' with a $\log \sigma$ slowdown factor if we use the functions of Table 3.1. Their implementation is similar to that described in Section 3.4.2 for DAGs, but we have to invert their algorithms as the node ordering in BOSS is from right to left (colexicographical), while in the index for DAGs is from left to right (lexicographical). More precisely, the function outneighbor in BOSS is implemented as inneighbor in the BWT-index for DAGs. Equivalently, inneighbor in BOSS is outneighbor in DAGs. The same occurs with functions outdegree and indegree. Note the bit vector I in BOSS serves the same purpose as the bit vector O in the BWT-index for DAGs.

For convenience, we also consider the following functions:

- label2node$(P[1, k - 1])$: node v labeled with P, if exists
- nodelabel(v): label of node v
- edgesymbol(v, k): symbol of the kth outgoing edge of v

The function label2node is an adaptation of find in DAGs (see Table 3.1). We implement nodelabel by performing a backward traversal of G' starting from v. The aim is to find any node v' reachable from v in $k - 1$ applications of the function inneighbor. We append the labels of the edges we visit in the traversal into a list A, and once we reach v', we invert A's sequence to return it as the label of v. Finally, to answer edgesymbol, we obtain the range $L[i, j]$ of v and return the symbol $L[i + k - 1]$.

Variable-Order dBG

Lin et al. [115] generalized the concept of the dBG as a way to deal with the limitations imposed by k when assembling genomes. In their representation, called the manifold dBG, the node labels have arbitrary lengths, and two nodes v and u are connected by an edge (v, u) if the sequence $l(v) \oplus^o l(u)$ exists in $\mathcal{S}$. In this case, o can be of any length, not just $k-2$ like in the regular dBG. The value for o can be even 0, which means that the string $l(v) \cdot l(u)$ appears in some string of $\mathcal{S}$.

Computing the node labels for the manifold dBG requires us to have the suffix tree of $\mathcal{S}$, which makes the representation less practical as the suffix tree contains enough information to perform genome assembly.

Boucher et al. [21] proposed an alternative generalization for the dBG that does not require the suffix tree. They noticed that the BOSS data structure implicitly stores all the dBGs of order $k' < k$. More precisely, if we only consider the last $k'-1$ columns of the BOSS matrix, then we will induce a partition where every range $[i, j]$ of rows with the same $k'-1$ label encodes a node v in the dBG of order $k' < k$. The outgoing edges of v are the distinct symbols in the segment $L[\mathsf{select}_1(B, i-1)+1, \mathsf{select}_1(B, j)]$.

To allow changing the order of the dBG, Boucher et al. augmented BOSS with the *longest common suffix* (LCS) array. The LCS array stores, for every node of order k, the size of the longest suffix shared with its predecessor node in the BOSS matrix. They called this new index the *variable-order* BOSS (VO-BOSS).

- $\texttt{shorter}(v, k'')$: node whose label is the k''-suffix of v's label
- $\texttt{longer}(v, k'')$: list U with all the nodes whose labels have length $k'' < k$ and are suffixed by v's label
- $\texttt{maxlen}(v, a)$: a node at order k whose label is suffixed by v's label, and that has an outgoing edge labeled a

By using a wavelet tree (Section 2.2.2), the LCS can be stored in $n \log k + o(n \log k)$ bits, the function $\texttt{shorter}([i, j], k')$ can be answered in $\mathcal{O}(\log k)$ time and the function $\texttt{longer}([i, j], k')$ in $\mathcal{O}(|U| \log k)$ time. The function $\texttt{maxlen}([i, j], a)$ is implemented using the arrays B and L, and hence it is answered in $\mathcal{O}(\log \sigma)$ time.

Colored dBG

The *colored* dBG enriches the edges of the graph with colors. This idea was introduced by Iqbal et al. [89]. Their version builds a union dBG G from several string collections and assigns the color j to the edges that encode kmers appearing in the jth collection. The compacted version of the colored dBG [130], called VARI, represents the topology of G using the BOSS index and the colors using a binary matrix $\mathcal{C}$, where the cell $\mathcal{C}[i, j]$ is set to 1 if the kmer represented by the ith edge in the ordering of BOSS is assigned color j. The rows of $\mathcal{C}$ are then stored using the compressed representation for bit vectors of [156] or using Elias-Fano encoding [61, 57, 146] if the rows are very sparse. Other compacted versions of the colored dBG have also been proposed by Almodaresi et al. [3, 4, 2] and Holley et al. [81].

We can also build a dBG from a string collection and assign each edge (u, v) c distinct colors, where c is the number of strings containing the kmer encoded by (u, v). This setting is handy for genome assembly. Suppose we have a colored dBG for a collection of reads. We can consider the edge colors to build contigs so that if we traverse a path colored with a and reach a branching node, we continue through the outgoing edge colored with a (if exists). The problem, however, is that the number of columns in $\mathcal{C}$ grows with the size of the collection. For reads, this feature implies that a colored dBG could require millions of colors, which increases the space usage too much for practical purposes.

Alipanahi et al. [1] noticed that we could reduce the columns in $\mathcal{C}$ by using the same colors in those strings that have no common kmers. This new problem was named the *CDBG-recoloring*, and formally stated as follows; given a collection $\mathcal{S}$ of strings and its dBG G, find the minimum number of colors such that i) every string $S_i \in \mathcal{S}$ is assigned one color and ii) strings having two or more kmers in common in G cannot have the same color. Alipanahi et al. also showed that the decision version of *CDBG-Recoloring* is NP-complete. They proposed a simple greedy heuristic that, in practice, significantly reduces the number of colors.

4.3.2 The Overlap Graph Framework

An *overlap graph* $G = (V, E)$ of a string set $\mathcal{S} = \{S_1, S_2, \ldots, S_q\}$ is a directed graph where every node $v \in V$ stores the label of some string $l(v) = S_j \in \mathcal{S}$. A directed edge $(u, v) \in E$ from u to v exists if there is an overlap $l(u) \oplus^o l(v)$, where o is above some threshold m. Additionally, the edge (u, v') is considered *transitive* if there is another node v such that the sequence of valid overlaps $l(u) \oplus^{o_u} l(v) \oplus^{o_v} l(v')$ exists, otherwise (u, v') is considered *irreducible*. Overlap graphs where all transitive connections are removed are called *string graphs* [133], or *irreducible overlap graphs* [120].

Succinct Construction of the Overlap Graph

We now explain how to efficiently construct the overlap graph G of $\mathcal{S}$ using the FM-index (Section 3.3.1). As the original DNA strand of the reads is unknown in a NGS experiment[7], we also have to consider the collection $\mathcal{S}^{rc}$ with the reverse complements of the strings in $\mathcal{S}$. This definition implies that $\mathcal{S}^{rc}$ also has m strings. Thus, for building G, we use the collection $\mathcal{S}^* = \mathcal{S} \cup \mathcal{S}^{rc}$ as input.

The first step is to create a string $S = S_1\$_1 S_2\$_2 \ldots S_{2q}\$_{2q}$ representing the concatenation of the strings in $\mathcal{S}^*$. We assume for the sake of explanation that the strings in S are sorted in lexicographical order, although this condition is not necessary. We compute the BWT of S and store it as a vector L with rank and select support. We can use, for instance, the wavelet tree of Section 2.2.2 to encode L. We also need the array $C[1, \sigma]$ with the frequencies of S's symbols. We refer to L and C as a partial FM-index as we do not include the suffix array nor the inverse suffix array.

Before computing the edges of G, we define a minimum threshold o such that any suffix-prefix overlap less than o characters long is not considered for building G.

[7] There are some specific protocols where we can know the strand, but they are not intended for assembly.

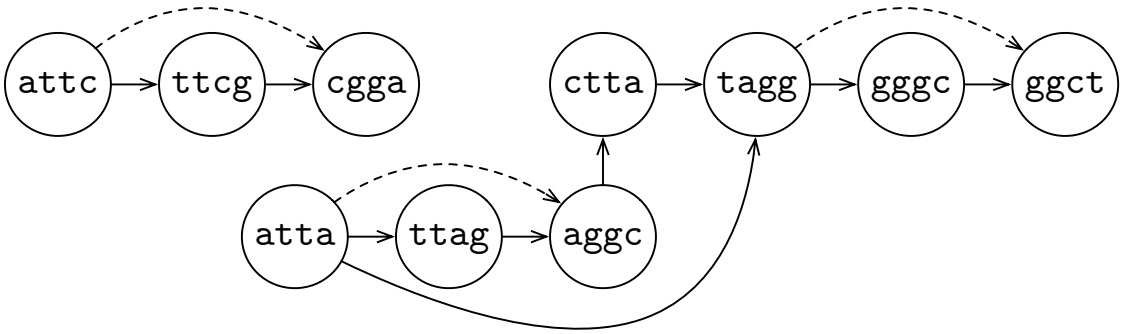

Figure 4.6: The overlap graph framework. Example of an overlap graph produced from the reads of Figure 4.4A. Dashed edges represent transitive connections. This example has no minimum threshold for the overlaps.

We proceed as follows for each $S_i \in \mathcal{S}^*$; we first call the function backwardsearch with S_i as input. Let $P = S_i[j..]$ be the suffix in the current backward search step and $L[s_j, e_j]$ its associated BWT range. If $L[s_j, e_j]$ contains \$ symbols and $|P| \geq o$, we perform an extra backward search step in $L[s_j, e_j]$ with symbol \$. Note that \$ symbols in $L[s_j, e_j]$ denote the strings in $\mathcal{S}^*$ with P as a prefix and overlapping the $|P|$-suffix of S_i. Also note that if a suffix $S[j'..]$ starts with $\$_i$, then the value $u \in [1, 2m]$ of $SA[u] = j'$ is the lexicographical rank of S_{i+1}. Hence, the range $L[s', e']$, with $1 \leq s' \leq e' \leq 2m$, we obtain with the extra backward search step stores the lexicographical ranks of the reads overlapping the $|P|$-suffix of S_i. Recall that the rank of a read is also its identifier as they are sorted lexicographically in S. Once we compute $L[s', e']$, we report every read whose identifier is $u \in [s', e']$ as overlapping S_i. After reporting the overlaps, we continue with the regular backward search steps from $L[s_j, e_j]$. Using this approach, we obtain all the strings of $\mathcal{S}^*$ that have an exact overlap with S_i. Mäkinen et al. [120] modified this procedure to compute the irreducible overlap graph instead of the overlap graph.

The main problem with this algorithm for building G is that computing the BWT of $\mathcal{S}^*$ can be computationally prohibitive given the high volumes of data in sequencing experiments. Besides, computing exact overlaps is not realistic as reads usually have sequencing errors.

4.4 Reference Genomes

A *reference genome* is a string collection produced from the assembly of a particular individual's genome. The purpose of a reference is to have a template to compare other closely-related genomes (usually from the same species). When a new individual is sequenced, we align (or map) its reads against the reference to assign them a genomic location. Subsequently, we search for differences (mismatches, insertions, or deletions) in the alignments as they can be potential genetic variations. We must, however, be careful not to confuse real genetic variations with misalignments or sequencing errors. In this regard, the main tools to avoid false positives are the sequencing coverage and the sequencing qualities.

Popular tools to align reads (a.k.a., *aligners*) such as `bowtie` [105] or `bwa` [113] build an FM-index (Section 3.3.1) of the reference genome and its reverse complement sequences, and assign genomic locations to the reads using a modified version of backwardsearch. This

version supports inexact matches; the read and its genomic location are allowed to differ (mismatches or gaps) in up to k different positions, where k is a parameter. The idea worked well beacuse NGS experiments produced short reads (<70 characters) with few sequencing errors. However, this approach rapidly became obsolete as NGS technologies improved their read lengths. To enable the alignment of longer reads (between 100 and 1,000,000 characters), posterior aligners [106, 113] adopted a *seed-and-extend* approach. Given an input read S, this method uses backwardsearch to find exact matches between substrings of S and substrings of the reference genome (the seeding phase), and then extend those matches using the Smith-Waterman [176] dynamic programming algorithm (the extension phase). This idea also proved to be more sensitive to perform alignments that require to split the reads into different genomic locations.

With the emergence of third-generation sequencing technologies (Section 4.2), seed-and-extend aligners were adapted for datasets with a high number of ultra-long reads (> 100,000 characters), and with higher sequencing error rates. For instance, `minimap2` [112] replaces the FM-index with a hash table storing the positions of the minimizers in the reference genome (Section 2.3.4). Later, `minimap2` computes exact matches between the input read S and the reference genome by obtaining the minimizers of S and looking for them in the hash table. The algorithm then extends the exact matches using colinear chaining, or dynamic programming if necessary. The advantage of this scheme is that the seeding phase is much easier to compute as we only need to perform lookups in the hash table. In contrast, the backwardsearch approach used by the previous aligners can be expensive if the substrings of S for which we search for exact matches are long. This scenario is highly probable, considering third-generation reads are ultra-long.

The `mashmap` aligner of Jain et al. [90] combines the ideas of sketching and minimizers (Section 2.3.4) to quickly find approximate alignments between a collection $\mathcal{S}$ of long reads and the reference genome G. Given an input $S_j \in \mathcal{S}$, their algorithm finds all the substrings G_i of length S_j in the reference genomes such that the Jaccard distance $J(S_j, G_i)$ is above some predefined threshold. When $\mathcal{S}$ is huge, computing all the $J(S_j, G_i)$ distances is expensive. Jain et al. solve the problem by using the winnowed-minhash estimate $\mathcal{J}(S_j, G_i)$ for $J(S_j, G_i)$, which it is cheaper to obtain. This estimate is similar to that of Minhash (Section 2.3.4), but instead of using the sketches of S_j and G_i, it uses the sketches of $W(S_j)$ and $W(G_i)$, which are the set of minimizers for S_j and G_i, respectively. Similarly to `minimap2`, `mashmap` also indexes the minimizers of G. This index allows them to quickly filter the G_i substrings that are unlikely to have a match with S_j. Experimental results showed that `mashmap` is space and time efficient and that it maintains sensitivity even when the reads have high error rates (<20%). It is also much faster than methods that rely on the FM-index and dynamic programming approaches. However, it does not support gapped alignments.

4.5 Pangenomes

An important problem with reference genomes is that they bias the genomic analyses. When the resequenced individuals have insertions or deletions in their genomes that are not in the reference, the aligner gets confused and mistakenly considers these variations to be sequencing errors. A possible solution to deal with the bias is to build a pangenome, a generalization in which the reference is not one genome but a set of individual genomes of the same species.

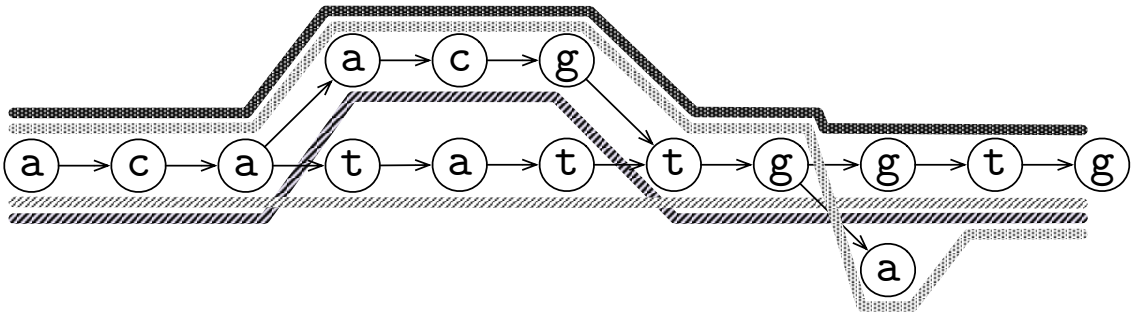

Figure 4.7: A pangenome obtained from the string collection {acatattggtg, acaacgtggtg, acaacgtgatg, acaacgtggtg}. The textures depict the paths in the DAG spelling the different strings.

We can regard a pangenome as a regular string collection. However, a more accurate model is a labeled directed acyclic graph (DAG) representing the collapse of all the strings in the collection. We have one subgraph per chromosome in the species, and the chromosomal differences (insertions, deletions, mismatches) between the individuals are encoded as "bubbles" in the subgraphs. An example is shown in Figure 4.7.

The idea of a pangenomic index with support for pattern matching was first considered by Schneeberger et al. [167] and Mäkinen et al. [123], and then by Sirén et al. [174], Huang et al. [83], and Danek et al. [45]. Still, the first succinct index that regarded the pangenome as a DAG was introduced by Sirén et al. [175]. They proposed a method to transform the DAG into a Wheeler graph to encode it with the BWT framework (Section 3.4). In addition to being succinct, the BWT framework also enables the computation of all the paths in the DAG labeled with a pattern $P[1, m]$ in $\mathcal{O}(m \log \sigma)$ time (Section 3.4).

One of the main features of the DAG approach is that pattern matching can report occurrences of P even if it does not exist in the collection (i.e., a false positive). This situation happens because paths encoding different strings of the pangenome are entangled in the DAG, and traversing these entangled paths might spell chimeric strings[8]. This feature is an advantage or a disadvantage depending on the context. In biology, a recombination event occurs when homologous chromosomes from different individuals get combined to form a new one. Therefore, P may not be a false positive but a new recombination event that has not been seen in the pangenome. Still, it is not easy to differentiate false positives from recombination events just with the graph topology. Sirén et al. 2020 [173] tried to address this difficulty by augmenting the pangenomic index [175] with the positional BWT [54], a data structure that enables the detection of path shifts. Similarly, Mäkinen et al. [121] proposed the founder graph, a pangenomic representation that facilitates the detection of recombination events. The founder graph can also be represented using the BWT framework.

Other representations consider the pangenome to be the concatenation of several individual genomes, not a DAG. They exploit the fact that individuals of the same species are highly similar, so their genomes are repetitive. This text redundancy enables the development of indexes that use little space. For instance, the implementation of the hybrid index

[8]The concatenation of subsequences from different strings of the pangenome.

by Valenzuela et al. [183] parses the text with the Lempel-Ziv algorithm (Section 3.2.3) to create a kernel sequence (Section 3.2), which is later stored using the BWT framework. This data structure achieves high compression ratios in practice but limits the maximum length of the reads that can be aligned. On the other hand, Kuhnle et al. [102] uses the r-index (Section 3.3.3) to encode the pangenome. They use the PFP procedure (Section 3.5.1) to reduce the computational resources when building the pangenome's BWT for the r-index. Their experimental results showed that they require less than 10% of the working memory of `bowtie`'s when indexing. Further, although the hybrid index uses less space, the r-index offers the best space/time trade-offs for pangenomes [38].

A limitation of the r-index, and BWT-based data structures in general, is that the alignment of reads is practical only when they are short and have few or no sequencing errors. How to efficiently support the inexact alignment of long strings is still an open question. In this regard, some authors [23, 160] have considered the problem of efficiently finding *maximal exact matches* (MEM) between a pattern and a string collection encoded as an r-index. These ideas could enable the implementation of a pattern-matching procedure on top of the r-index that uses the seed-and-extend approach of `bowtie` or `bwa`.

Chapter 5

Grammar-Compressed Reads

In this chapter, we describe a new grammar compressor for storing DNA sequencing reads. The novelty of this representation is that it can be used to compute the eBWT (Section 3.2.1) of the reads directly from the grammar. Our motivation is to perform in succinct space genomic analyses that require complex string queries not yet supported by dictionary-based self-indexes. Our approach is to maintain the collection of reads as a grammar as long as they are not used. However, when an analysis is required, we quickly compute their eBWT without fully decompressing the text.

5.1 Motivation

As explained in Section 3.2.2, the benefits of using grammars for encoding text are that we can achieve high compression ratios when the input is repetitive, and that we can directly access any substring with only an additive logarithmic time penalty [16]. We consider these features the starting point to develop an algorithmic framework to process high volumes of DNA sequencing data in little space.

Still, the functionality offered by grammar-based self-indexes is still limited compared to the complex sequence analyses required in computational biology scenarios [120]. In this regard, the suffix tree is one of the few data structures that supports sufficiently elaborated queries as to process genomic experiments. However, its space usage is several times the size of the input, making it impractical for big collections. We can reduce the costs by using the FM-index. In that way, we can compress the reads to their zeroth order empirical entropy without losing the suffix tree functionality. The problem, however, is that sequencing experiments are so massive that even the FM-index's space usage can be prohibitive. The ideal solution would be then to have a data structure that compresses the data by exploiting the DNA repetitions, but at the same time, supports string queries similar to those of the suffix tree.

The so-called run-length BWT (RLBWT) [124, 70] is a possible alternative. This compression scheme exploits the fact that, on highly repetitive text collections, the BWT consists of a small number of long runs of the same letter (see Section 3.2.1). It can then enable complex

sequence analyses in little space. Still, on read collections, the RLBWT does not compress significantly [53]. Grammars and Lempel-Ziv are still preferred for permanent storage as they obtain better space reductions in practice.

There are several genomic tools in the literature that rely on the BWT of the reads [170, 155, 53]. If we put aside the cost of computing the BWT, the idea is compelling as it enables an efficient reference-free processing of the data. This feature is desirable because reference genomes bias the results (see Section 4.4). In reality, however, we still have the problem of constructing the BWT [92]. Although there are algorithms that run in linear time [147, 82], in practice they still require significant storage and processing resources. As an alternative, we can use efficient external algorithms [44, 55, 20] for building BWT variants for string collections, but they are mostly intended for short reads. More recent in-memory approaches [22] reduce the costs by factorizing the repetitions of the text, but they are aimed at collections of assembled genomes, and do not work well on reads. All these limitations make reference-free methods still difficult to implement.

All the limitations mentioned above leave us with the following tradeoff. On one side, we have the RLBWT, which is expensive to compute and whose compression ratio is not that good on reads. However, it still allows us to process genomic data in succinct space and perform reference-free analyses. On the other, we have dictionary-based methods like Grammars or Lempel-Ziv; they achieve much better compression ratios than the RLBWT, but their self-indexes are not versatile enough as to process genomic data. Considering this scenario, we propose an intermediate solution; a grammar encoding tailored for reads from which we can compute the eBWT. Our algorithm for producing the eBWT uses the repetitive patterns captured by the grammar to boost the computations. As far as we know, this idea has been implemented only from Lempel-Ziv compression and is considerably slow [153]. As discussed, the Lempel-Ziv format does not enable, on the other hand, direct access to the reads for other purposes. With our approach, we maintain a low memory footprint when the reads are not used, but if an analysis is required, we obtain the RLBWT in a efficient way.

We call our grammar compressor algorithm LMSg. Similar to the work of Nunes et al. [142], our method builds on the SA-IS algorithm (Section 3.5.2). However, our approach considers some extra modifications to compress the text even further and facilitate the computation of the eBWT. We encode the final grammar using a variation of the grammar tree (Section 3.2.2). We implemented the ideas described in this chapter in a C++ tool called LPG.

Our experiments in real data showed that the space reduction we achieve with LPG is competitive with Lempel-Ziv-based methods and better than BWT-based approaches (FM-index and RLFM-index). Compared to other popular grammars, such as BigRePair, we achieve 12% extra compression in DNA and require less working space and time. Besides, the working memory LPG requires for building the grammar is 50%–60% the space of the input, which is far less than most grammar construction algorithms. A preliminary version of this work [52] was presented at the *21st Data Compression Conference* (DCC'21).

5.2 Definitions

Let the string collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be a multiset of m reads over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{n}, \mathtt{t}\}$, where the longest string has length k. We consider the string $S = S_1\$_1 S_2\$_2 \ldots S_m\$_m$ to be the concatenation of the reads separated by a dummy symbol $\$$ smaller than any character in Σ. The size of S is n. Let $\mathcal{G} = \{V, \Sigma, \mathsf{S}, \mathcal{R}\}$ be a context-free grammar that only produces S. V is the set of nonterminals, Σ is the alphabet of terminals, S is the start symbol and $\mathcal{R}$ is the set of rules. Additionally, we denote the number of rules as $g = |\mathcal{R}|$. The grammar size G is defined as the sum of the lengths of the right-hand sides of $\mathcal{R}$. We refer to the string C on the right-hand side of the rule of S as the *compressed string* of $\mathcal{G}$, and its size is denoted as $c = |C|$. We use the string ordering that the SA-IS algorithm induces over the LMS substrings (Section 3.5.2). It is similar to lexicographical ordering. The only difference is that when a string A is a prefix of another string B, A gets higher order. We lift the operator $\prec_{LMS}$ to refer to this special ordering. We also use the function $\mathsf{exp}(\mathsf{X})$ to refer to the string in Σ^* resulting from the recursive expansion of nonterminal $\mathsf{X} \in V$.

5.3 The LMSg Algorithm

LMSg is an iterative algorithm that produces $\mathcal{G}$ in several rounds of parsing. In every round i, we classify the symbols of S^i ($S^1 = S$) as L-type, S-type or LMS-type to generate a partition over S^i (see definitions in Section 3.5.2). Each block (or phrase) in the partition starts in a position $S[j]$ such that $S[j-1]$ is LMS-type and ends in the smallest position $j' > j$ such that $S[j']$ is also LMS-type. We refer to these blocks as *LMS phrases*. We create a dictionary $\mathcal{D}^i$ with all the distinct phrases of S^i. Then, we create a new rule $\mathsf{X} \to F$ for every $F \in \mathcal{D}^i$, where X is the greatest symbol on $\Sigma \cup V$ before round i plus the $\prec_{LMS}$ order of F among the other phrases in the dictionary. After generating the new rules from $\mathcal{D}^i$, we create the parse S^{i+1} by replacing the LMS phrases in S^i with their $\prec_{LMS}$ orders, and perform another parsing round $i+1$ using S^{i+1} as input. LMSg ends in the parsing round i' where all the phrases of $\mathcal{D}^i$ have frequency one in S^i, in which case we create the rule $\mathsf{S} \to S^i$ for the start symbol of $\mathcal{G}$.

Note that for every new rule $\mathsf{X} \to F$ we create from $\mathcal{D}^i$, the sequence of its right-hand side does not contain grammar values, but symbols in S^i. We solve this problem by keeping track of the nonterminals assigned to the symbols in the alphabet of S^i. More specifically, if $S[j] = r$ is the $\prec_{LMS}$ order of the phrase $R \in \mathcal{D}^{i-1}$, then the nonterminal for r is the one that was assigned to R in the previous parsing round $i-1$. We use this information to replace every $F[u]$ with its nonterminal when we create the rule $\mathsf{X} \to F$. We do the same when we create the rule for S. This modification maintains the grammar consistency.

Our procedure is similar to that of Nunes et al. [142]. Still, we go further and try to reduce the grammar size without losing information for computing the eBWT of $\mathcal{S}$.

5.3.1 LMSg is for String Collections

LMSg is oblivious to the number of documents encoded in the input text. It might produce, for instance, a nonterminal $\mathsf{E} \in V$ expanding to a substring $S_x[u..]\$_x \ldots \$_{y-1}S_y$ that represents

an incomplete string $S_x[u..]$ of $\mathcal{S}$ concatenated with one or more other strings. This type of nonterminals makes the construction of the eBWT more difficult. An important aspect of our BWT algorithm is to assign ranks to the symbols in V based on their string expansions. However, in the case of E, its expansion is not a valid substring in the circular[1] rotations of $\mathcal{S}$, so the rank for E has no meaning in our model. We avoid nonterminals like E by enforcing the following property on $\mathcal{G}$:

Definition 6 $\mathcal{G}$ is *string independent* iff every nonterminal $\mathsf{E} \in V$ expands to a substring $E = \mathsf{exp}(\mathsf{E})$ of S that meets either of the following conditions:

1. $E = S_x[a, b]$ is an internal substring of some $S_x \in \mathcal{S}$ (i.e., E is not a suffix in S_x)
2. $E = S_x[a..]\$_x$ is a suffix of S_x concatenated with a \$ symbol

We ensure the conditions of Definition 6 by performing an extra step in every parsing round i. Suppose that during the scan of S^i, we reach an LMS phrase F whose recursive expansion yields a string in the form ABC. A is a suffix of some $S_x\$_x$, B is either an empty string or the concatenation $S_{x+1}\$_{x+1} \ldots S_{x+p}\$_{x+p}$, and C is a prefix of $S_{x+p+1}\$_{x+p+1}$. We create a new phrase with the prefix of F expanding to A. If B is not empty, then we produce a new phrase with every segment $F[a', b']$ that expands to some substring $S_{x'}\$_{x'}$ of B. Finally, we create the last phrase with the suffix of F expanding to C. We record these new elements into $\mathcal{D}^i$ afterward. The new phrases whose recursive expansions end with a \$ symbol are called *border phrases*.

Every parsing round i of LMSg now also considers an input bit vector B^i that tells us which symbols in the alphabet of S^i recursively expand to suffixes of $\mathcal{S}$. This bit vector facilitates the detection of LMS phrases spanning two or more strings of $\mathcal{S}$. Once we produce S^{i+1}, we create its associated bit vector B^{i+1} to pass it as input for the next round.

5.3.2 Simplifying the Grammar

While parsing S^i, we discard the phrases that are not useful for either compressing or producing the eBWT of $\mathcal{S}$. We insert the symbols in these phrases directly into $\mathcal{D}^i$ to *transfer* them to subsequent parsing rounds, hoping they will be encapsulated within more useful contexts. We discard a substring in two cases; (i) all its symbols appear only once in S^i or (ii) its length is less than two.

After finishing a parsing round, we sort the phrases of $\mathcal{D}^i$ in $\prec_{LMS}$ order and scan $\mathcal{D}^i$ from left to right to create the new nonterminal rules. If a phrase F has length > 1 (a non-transferred symbol), we proceed in the same way as in Section 5.3. However, when a phrase F has length one (transferred symbol), we update the nonterminal $\mathsf{F} \in \mathcal{R}$ previously assigned to it. The new value is $p + b$, where p is the size of $\mathcal{R}$ before iteration i and b is the $\prec_{LMS}$ order of F in $\mathcal{D}^i$. This update requires us to change the left-hand side of F's rule and the occurrences of F in the right-hand sides of $\mathcal{R}$. The use of transferred symbols also changes the stop condition for LMSg; the algorithm ends the parsing rounds when all the phrases of length > 1 in $\mathcal{D}^i$ have frequency one.

[1] Recall that the eBWT considers the string in $\mathcal{S}$ to be circular.

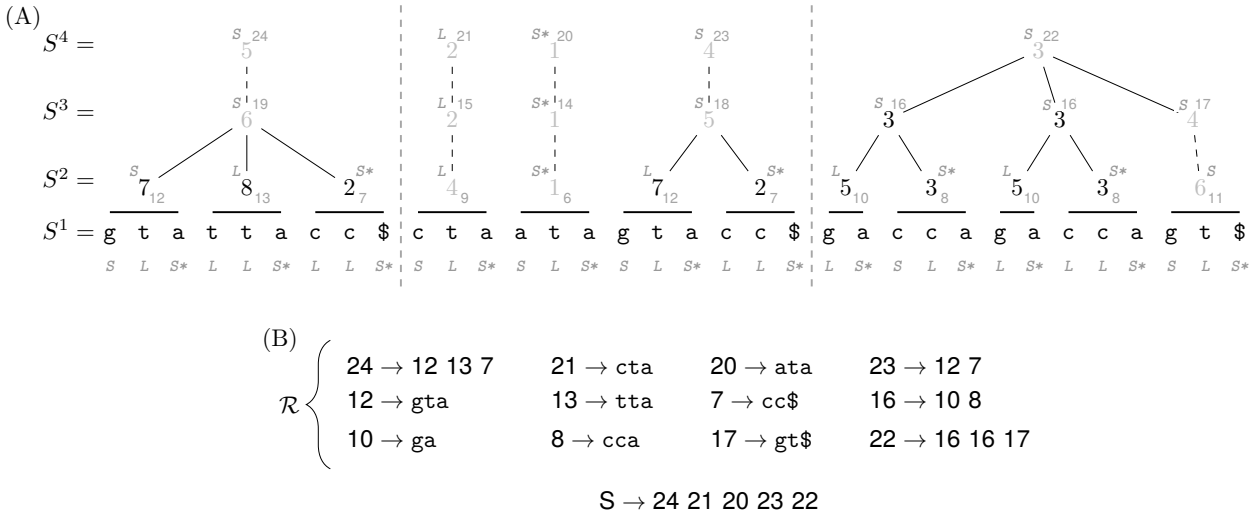

Figure 5.1: (A) Running example of LMSg. The symbols in gray below S^1 are character types (L-type=L, S-type=S, LMS-type=S^*). Dashed vertical lines mark the limits between the strings in $\mathcal{S}$. Every horizontal line on top of S^1 spans one of the phrases generated in the parsing round 1 of LMSg. The rest of the parsing rounds are depicted on top of S^1. Light gray symbols have frequency one in S^i. Dashed edges indicate symbols that were transferred to the next parsing round. The gray character at the top of every $S^i[j]$ denotes its suffix type and the gray number to the left is its assigned nonterminal in $\mathcal{G}$. (B) The grammar $\mathcal{G}$ resulting from the parsing rounds of (A). The size G of the grammar is 38, the number g of nonterminal is 13, and the length c of the compressed string is 5. For clarity, the nonterminal values were not collapsed.

The nonterminals produced by LMSg could be non-consecutive due to the transfer of symbols. We need to collapse their values to produce a more compact grammar representation. For that purpose, we scan $\mathcal{R}$ and change every left-hand symbol with the smallest unused symbol in V. As we do the replacements, we keep track of the changes to update the references of the characters on the right-hand sides of $\mathcal{R}$. Figure 5.1 shows a complete running example of LMSg.

5.3.3 Analysis of LMSg

We now present the upper bound for constructing $\mathcal{G}$ using LMSg and S as input. We describe our result with the following theorem.

Theorem 1 The LMSg algorithm runs in $\mathcal{O}(n \log k)$ time, where k is the longest string on $\mathcal{S}$.

Proof. SA-IS, the method on which LMSg relies upon, runs in $\mathcal{O}(n)$ time because the length of every S^{i+1} is at most half the size of the previous S^i. In this way, the algorithm processes less than $2n$ symbols in total. However, in our case, we cannot ensure that property because we transfer symbols from one parsing round to the next one, meaning that the length of S^{i+1}

can be more than $S^i/2$. This drawback implies that the parsing of every S^i takes $\mathcal{O}(n)$ time.

We know that LMSg incurs in at most $\log k$ parsing rounds as every new phrase we produce from S^i spans at least two symbols, and the length of the recursive expansion of a new phrase is never longer than k, the longest string in $\mathcal{S}$. We enforce the latter property with the string independence of Definition 6. On the other hand, the transferred symbols of S^i that belonged to the same discarded phrase maintain their lexicographical relationships. Hence, if encapsulated in later parsing rounds, they will belong to the same phrase. Also, if a substring $S^i[a, b]$ whose recursive expansion matches a substring $S_j\$_j$, with $S_j \in \mathcal{S}$, is composed only of unique symbols, it will not be further compressed. Instead, its symbols will continue to be transferred until LMSg stops (see, for instance, substring $S^3[2, 4] = 215$ of Figure 5.1). As a consequence, LMSg runs in $\mathcal{O}(n \log k)$ time. □

5.3.4 Efficient Dictionary Construction

In every parsing round i of LMSg, we use a hash table to record the distinct LMS phrases of the dictionary $\mathcal{D}^i$. Each phrase F is the key and its associated value is a boolean flag that indicates if F is repeated in S^i. When the text is repetitive, the first parsing round ($i = 1$) produces a small dictionary so the hash table will not require much space. Still, as we move on to the next rounds, the number of distinct phrases quickly increases in S^i, so the working memory for building $\mathcal{D}^i$ becomes considerable.

We can reduce the computing time by building $\mathcal{D}^i$ in parallel during the parsing round. We cut S^i into p different chunks, where p is the number of working threads, and obtain the LMS phrases in parallel in every chunk. We collapse the phrases recorded by the threads afterward to get $\mathcal{D}^i$. Still, having one hash table per thread would be expensive for the parsing rounds $i > 1$, so it is not an option. In contrast, having only one hash table that is concurrently accessed by the threads decreases the efficiency due to synchronization issues.

We deal with the efficiency problems by creating a semi-external bit-compressed hash table to construct $\mathcal{D}^i$ in parallel and using an amount of working memory defined by the user. We start by defining a buffer B of b bits, where b is a parameter. Subsequently, we divide B into p blocks of $u = \lfloor b/p \rfloor$ bits. Additionally, we subdivide every block B_j, with $j \in [1, p]$, into two halves. The left half B_j^l stores the hash table T_j of the jth working thread and the right half B_j^r is a buffer that stores the hashed pairs of T_j. We implement T_j using Robin Hood probing (Section 2.3.1) to work at high load factors (we use 0.8). Every cell $T_j[u]$ uses 8 bytes; the first 2 bytes in the cell encode the distance to the real hashing position of the key associated with $T_j[u]$. The last 6 bytes store the bit index q in B_j^r, where the key-value pair of $T_j[u]$ is stored. In $B_j^r[q]$, we encode the information as follows; the first 4 bytes contain the length l of the key. The next $x = l \cdot \log \sigma^i$ bits store the key sequence, where σ^i is the alphabet of S^i, and the last bit is the value associated with the key, i.e., the boolean flag that indicates if the LMS phrase is repeated or not.

When inserting a new key-value pair (F, b) into T_j, we store it in the rightmost available position of B_j^b. If inserting (F, b) produces B_j^r to exceed its capacity of $u/2$ bits, then we dump B_j^r into the disk and reset the complete block B_j. Similarly, when the load factor of T_j exceeds the threshold of 0.8, we also dump B_j^r to disk and reset B_j. Alternatively, we can

check if B_j^r still contains free space to shift the boundary between B_j^l and B_j^r to the right and give more bits to B_j^a. In that way, we can increase the size of T_j. This mechanism will avoid the disk dump, but it will trigger a rehashing.

Once we finish the parallel partition of S^i, we collapse the dumped data of the p working threads in one single hash table, which later will become $\mathcal{D}^i$. Note that in the parsing round $i = 1$, the hash tables will contain almost the same phrases. In the worst case, each T_j will be a full copy of $\mathcal{D}^i$, but as the dictionary is small at this level, the number of disk dumps will be close to zero. In later parsing rounds $i > 1$, $\mathcal{D}^i$ can be large, but it is less probable for the distinct T_j to share keys as the phrase frequencies in S^i are likely to be small. This feature will reduce the data dumps triggered due to redundancy in the hash tables. Working at high load factors in the hash tables and maintaining the data in bit-compressed form in B_j^b also help us to reduce the number of data dumps.

5.4 Recompressing the Grammar

After running LMSg, we recursively create new rules from the *maximal* suffixes of size two or more that appear repeated in the right-hand sides of $\mathcal{R}$. We refer to these new nonterminals as RS (repeated suffix). Figure 5.2 depicts the idea. The concept of maximal suffixes is similar as in the suffix tree. We consider a string F to be maximal if it appears $c > 1$ times as a suffix in the right-hand sides of $\mathcal{R}$, and for any $b \in \Sigma \cup V$, its left extension bF appears $c' < c$ times as a suffix. The RS nonterminals are helpful to reduce the size G of the grammar, but they are also convenient for computing the eBWT of $\mathcal{S}$ as we will see in Chapter 6.

To create the RS nonterminals, we record in a hash table the distinct suffixes of length two in the right-hand sides of $\mathcal{R}$, and create new rules with those that have frequency more than one. We replace the occurrences of the repeated suffixes with their new nonterminal symbols and continue hashing suffixes of length two until no new rule can be created. Subsequently, we remove the RS rules whose left-hand symbol occurs only once in the right-hand sides of $\mathcal{R}$. The only problem with this idea is that the strings in $\mathcal{R}$ are not so repetitive, so we might end recording a lot of sporadic pairs that are later discarded because they are unique. We can reduce the number of unnecessary pairs in the hash table by including a simple condition; both symbols of the suffix must be repeated in $\mathcal{R}$. We can mark every repeated nonterminal with a bit map prior the creation of the RS rules.

It might happen that the complete sequence F of an LMSg rule $\mathsf{X} \rightarrow F$ appears as a proper suffix in one or more right-hand sides. In such situation, we do not create a new rule but reuse the value of X to replace those proper suffixes. When this happens, we consider X to have a *dual* context as it occurs as an LMSg nonterminal but also as an RS nonterminal. Figure 5.2 shows an example of this situation.

5.5 Encoding the Grammar

We use the *grammar tree* data structure proposed by Claude et al. [36] (denoted here as $\mathcal{P}$) to store $\mathcal{G}$. We make, however, some modifications to later compute the eBWT of $\mathcal{S}$ in a more efficient way.

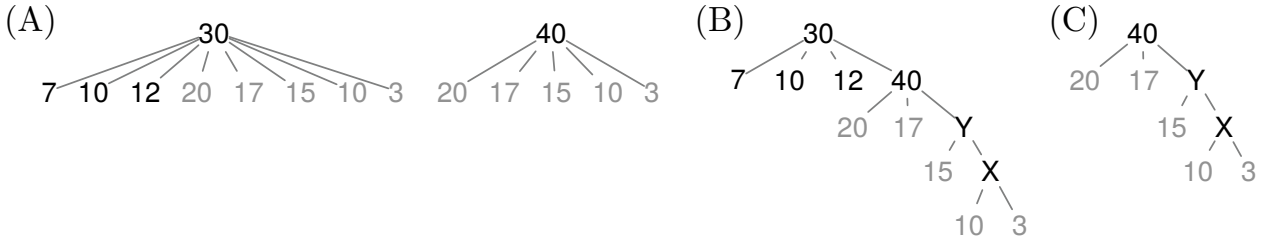

Figure 5.2: Example of RS nonterminals. (A) The parse tree of two distinct nonterminals 30 and 40 produced by the LMSg algorithm. Gray symbols denote the suffixes that are repeated in the right-hand sides of $\mathcal{R}$. (B) The parse tree of (A) now with the RS rules included. Symbols X and Y are new RS nonterminals while 40 is an LMSg nonterminal with dual context as it also appears as a suffix under the subtree of 30.

We create $\mathcal{P}$ in one level-order traversal of the parse tree of $\mathcal{G}$. The procedure is as follows; every time we visit a new node v in the parse tree, we check first if its label $\mathsf{X} \in V$ has dual context. If it does, and v is an RS occurrence, then we create a new leaf v' in $\mathcal{P}$. Subsequently, we check if there is already an internal node in $\mathcal{P}$ for X. If there is one, we assign its label to v'; we leave it unlabeled otherwise. When X has dual context, but v is an LMSg occurrence, we create v' as an internal node. In this case, the label for v' is $x + \sigma$, where x is the number of internal nodes in level-order in $\mathcal{P}$ up to v'. We also label all the previous leaves of $\mathcal{P}$ that represent occurrences of this nonterminal. When X does not have dual context, but v is the first node we visit in the traversal that is labeled with it, we create v' as an internal node and label it with $x + \sigma$. When v is not the first node for X we see, we create v' as a leaf and label it with the value we used for the internal node in $\mathcal{P}$ that encodes the first occurrence of X. Finally, when v represents a terminal symbol b, we create v' as a leaf labeled with b. The parent of v' in $\mathcal{P}$ is the internal node that maps to the parent of v in the parse tree. Additionally, when we create v' as a leaf, we discard the subtree rooted at v from the rest of the parse tree traversal.

We encode the topology of $\mathcal{P}$ in a bit vector K using LOUDS (Section 2.2.3). Additionally, we store the leaf labels in a vector Z using the data structure for canonical Huffman codes of Schwartz and Kallick [169]. We augment Z with sampled pointers for direct access (Section 2.1.3). Figure 5.3 depicts the resulting grammar tree for the running example of Figure 5.1.

The grammar tree construction algorithm ensures that if X has several occurrences on the right-hand sides of $\mathcal{R}$, only one of them is stored as an internal node in $\mathcal{P}$. The others are stored as leaves. We refer to this internal node as the *locus* of X in $\mathcal{P}$, and the locus's label as the *identifier* of X. Our algorithm also ensures that if X has dual context, then its locus in $\mathcal{P}$ will always be an LMSg occurrence. We use this property during the construction of the eBWT in Chapter 6.

Theorem 2 The grammar tree representation for $\mathcal{P}$ requires $2G + o(G) + (G - g)(\mathcal{H}_0(Z) + 1) + (g + \sigma)\log(g + \sigma) + \sigma w$ bits of space, where Z is the vector containing the Huffman codes of the grammar tree labels.

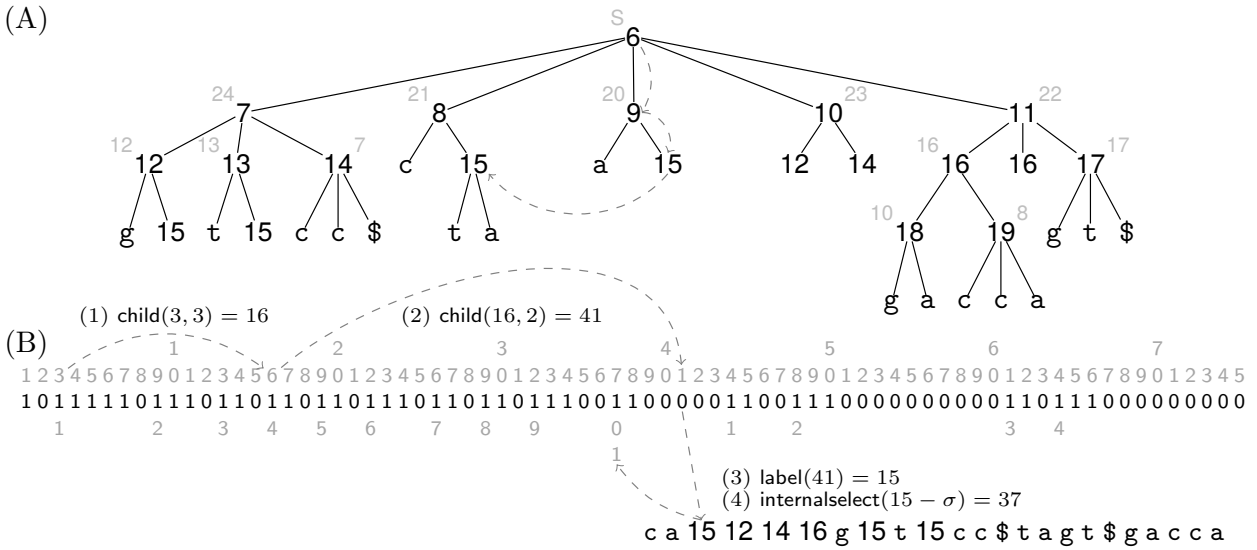

Figure 5.3: (A) The grammar tree of Figure 5.1B. Numbers on top of the internal nodes are the original nonterminals of the grammar. Symbol **15** is an RS nonterminal. Dashed arrows simulate a traversal over the parse tree of $\mathcal{G}$ to decompress the word `ta` from $S^1[14, 15]$ (Figure 5.1). (B) LOUDS encoding for (A). The bit stream stores the shape of the tree. Gray numbers on top are the bit indexes. Gray numbers below the stream are the internal ranks of the nodes. The integer vector below the stream contains the leaf labels. Dashed arrows mark the same decompression path as in (A), but using the LOUDS functions.

Proof. $\mathcal{P}$ has $G+1$ nodes and g internal nodes. As the bit array K is a LOUDS representation of the topology of P, it uses $2G + o(G)$ bits. On the other hand, the vector Z contains the labels of the $G-g$ leaves of $\mathcal{P}$, whose values are over the alphabet $[1, g+\sigma]$. The bit array with the Huffman codes of Z uses $(G-g)(\mathcal{H}_0(Z)+1)$ bits, and the other auxiliary data structures in the representation of Schwartz and Kallic require $(g+\sigma)\log(g+\sigma)+\mathcal{O}(\log^2(G-g))$ extra bits. The sampled pointers for Z use $\lceil (G-g)/k \rceil w$ bits, where k is a parameter. Assuming the word machine w is large enough so that any value for G fits on it, we can choose $k = w^2$ to obtain a space complexity for the sampled pointers of $(G-g)/w < (G-g)/\log(G-g) = o(G-g)$ bits. Finally, the σw bits stand for the integer array that maps the alphabet of terminals in $\mathcal{G}$ to the original symbols in $\mathcal{S}$. □

Theorem 3 Accessing the label of a grammar tree node v in the representation of Theorem 2 takes $\mathcal{O}(k^2 \log(G-g))$ time, where k^2 is the sampling rate of Z.

Proof. When v is an internal node, computing its label takes $\mathcal{O}(1)$ time as we obtain it using the LOUDS operation $\mathsf{internalrank}(v) + \sigma$. When v is a leaf, we extract its label from $Z[\mathsf{leafrank}(v)]$. Decoding a symbol in Z takes us $\mathcal{O}(\log(G-g))$ time as the longest length a Huffman code can have in Z is $\mathcal{O}(\log(G-g))$ bits. The reason is that Z has $G-g$ symbols, and thus the Huffman codes we obtain from this vector cannot have frequency less than $1/(G-g)$. As we chose a sampling rate of k^2 for the sampled pointers, accessing a position in Z requires us to decode at most k^2 symbols, which gives us the final time complexity of $\mathcal{O}(k^2 \log(G-g))$. □

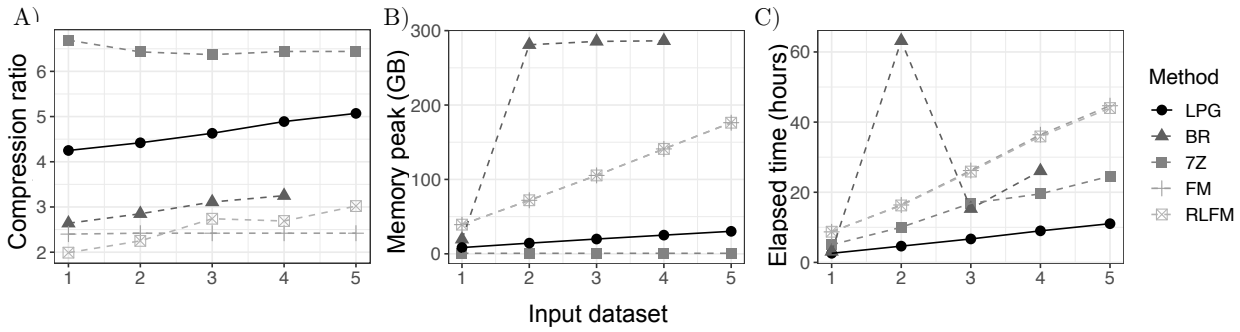

Figure 5.4: Performance of the different compressors. The compression ratio is measured as the size of the plain text divided by the size of the final compressed representation, so higher is better.

For simulating in $\mathcal{P}$ a top-bottom traversal of the parse tree of $\mathcal{G}$ we use the constant-time navigational function child defined for LOUDS, but also an extra function $\mathsf{label}(v)$ that returns the label of a node v.

The traversal of the parse tree is as follows, we navigate $\mathcal{P}$ top-down using the child operation as long as the nodes we visit are internal nodes. When we reach a leaf u, if $\mathsf{label}(u) \leq \sigma$, then we stop the traversal because we have reached a terminal symbol. If that is not the case, then we continue the traversal from the subtree rooted at $v = \mathsf{internalselect}(K, \mathsf{label}(v) - \sigma)$. See Figure 5.3.

5.6 Experiments

We implemented our grammar compressor as a tool called `LPG` (`https://bitbucket.org/DiegoDiazDominguez/lms_grammar/src/bwt_imp2`). The software is written in `C++` and uses the `SDSL-lite` library [76]. We compared the performance of `LPG` against BigRepair [72] (`BR`), `7-zip` [150] (`7Z`) and the FM-index [65]. BigRepair is a space-efficient variation of RePair for large repetitive collections. We encoded the BigRepair grammars with the recent representation of Gagie et al. [163], which allows fast random accession to substrings of the text. For the FM-Index, we considered both the regular version (denoted as `FM`) and the run-length compressed version (denoted as `RLFM`). The BWTs for the FM-indexes were calculated using `egap` [55]. When parallelization was possible, we ran the experiments with 10 threads.

We used as input five distinct collections of reads produced from different human individuals. This data was obtained from the Human Genome Diversity Project[2]. The datasets were identified with the number of individuals they contained. Their sizes in GB were 1=12.77, 2=23.43, 3=34.30, 4=45.89 and 5=57.37. All the reads were 152 characters long and had an alphabet of six symbols (`a,c,g,t,n,$`). The instance of `BR` with collection 5 returned an error and therefore it was not included in the analyses. For dataset 1, we allowed `BR` to use at most 72 GB (6x the input size) of working memory. However, with the rest of the collections we had to increase that value to 275.36 GB as the program was taking too long to finish. The performance of the compressors is shown in Figure 5.4.

[2] `https://www.internationalgenome.org/data-portal/data-collection/hgdp`.

	Random access							
Input	LPG	BR	RLFM			FM		
			0.05	0.5	1	0.05	0.5	1
1	104.30	98.67	804.11	116.71	78.45	682.40	104.92	71.16
2	111.35	101.59	788.37	115.91	77.86	692.62	104.09	70.85
3	124.04	98.56	784.40	116.96	77.85	682.09	104.66	71.18
4	128.58	104.72	821.12	118.71	81.01	681.80	104.65	71.62

Table 5.1: Random access. Average time in μsecs to randomly access a read. The columns of `RLFM` and `FM` indicate the different sampling rates (0.05, 0.5, and 1) we used in those instances for randomly accessing the reads.

We measured the time for randomly accessing the reads from the compressed representations. We sampled reads at regular text intervals in the FM-index instances (`RLFM` and `FM`) to support fast access. For every sampled string, we stored the BWT position of its last character. We selected three sampling rates; 0.05, 0.5, and 1. We store one BWT position every 20 reads with the first sampling rate; one position every two strings with the second one, and we stored the BWT positions for all the reads with the last one. We excluded `7Z` from this experiment as its current implementation does not support random access.

We augmented the `LPG` instances with a bit vector $B[1, c]$ that marks in $\mathcal{P}$ the nodes at depth one that recursively expand to string suffixes. We augmented B with select structures to access the substrings of C that map complete reads. We also encoded the leaf labels of $\mathcal{P}$ using arrays of $\log r$-bit cells instead of Huffman-compressing them. This representation allowed us to access the grammar tree labels in $\mathcal{O}(1)$ time. The results of the random access experiments are depicted in Table 5.1.

5.7 Results and Discussion

The average compression ratio of `LPG` was 4.65. This result was better than the one obtained by `BR` and `RLFM` (2.96 and 2.54, respectively), but worse than that of `7Z` (6.47). Although `7Z` outperformed the other methods at reducing the space, the difference decreased as the inputs grew and became more repetitive. For instance, the gap in the compression ratio between `7Z` and `LPG` for collection 1 was 2.44, while for collection 5 it was 1.37. The poor performance of `BR` may be because its preprocessing step (Prefix-Free Parsing) did not capture well the repetitiveness in the reads. `BR` produced, on average, 36% more grammar rules than `LPG`. On the other hand, the small compression ratios obtained by `RLFM` can be due to the number of BWT runs. In reads, this value is usually not as small as in other text families. The run heads represented, on average, 23% of our inputs. Regarding the memory peaks, the consumption of `7Z` was negligible (0.7 GB). In contrast, `LPG` required a much more considerable amount of working space (about 58% of the input size). Still, this value was far less than that of `BR` and `RLBWT`, which used 7 and 3 times the input size, respectively. In elapsed time, `LPG` outperformed all the other methods. The instance of `BR` with collection 2 took much more time compared to collections 3 and 4 (63.18 hours versus 15.31 and 26.08 hours, respectively). We assume this behavior is a bug in the implementation.

The average time for accessing a random string in LPG was 117.06 μsecs. This result was competitive with the performance of BR (100.89 μsecs, on average). The outcome of RLFM and FM varied according to the sampling rate we used. In general, FM outperformed RLFM in all the experiments. We expected this result as RLFM needs to carry out additional operations to solve the rank queries over the run-length compressed representation of the BWT.

Interestingly, FM and RLFM became competitive with LPG and BR only when we sampled more than 50% of the reads. FM and RLFM were the fastest methods in those instances where we stored pointers for all the reads (columns 6 and 9 of Table 5.1). However, with a sampling rate of 0.05, the average performance of FM and RLFM decreased dramatically, becoming the slowest methods (see columns 4 and 7 of Table 5.1).

The extra space overhead required to support random access was small in all the cases. For LPG, we used 16.80% of the original size of the grammar tree data structure. The space overhead for FM and RLFM varied according to the sampling. However, it was smaller than in LPG in all the cases. Using the sampling rate of 0.05, the space overhead in the FM-indexes ranged from 0.30% to 0.42%. Using the sampling of 0.5 ranged from 2.96% to 4.23%, and with the sampling rate of 1, it ranged from 5.91% to 8.47%.

Chapter 6

Computing the eBWT

This chapter describes a new algorithm called infBWT to compute the eBWT of a string collection from the grammar representation of Chapter 5. As explained earlier, the purpose of that grammar is to store massive collections of raw sequencing data (reads) using little space. Producing the eBWT, on the other hand, enables the efficient extraction of biological information in succinct space. Further information on these ideas can be found in the introduction of Chapter 5.

Our algorithm infBWT exploits the repetitive text patterns captured by the grammar rules to reduce the working memory and CPU time. Thus, the amount of resources it consumes depends more on the new information we add to the collection than on its size. For instance, if the input grammar encodes two copies of the same sequencing experiment, then the requirements of infBWT increase by a factor smaller than two compared to a grammar encoding only one copy. This feature can be helpful in the processing of massive DNA experiments as they usually contain several genomes of the same species, which are almost identical.

We implemented infBWT as a module of `LPG`, the `C++` implementation of the grammar compressor of Chapter 5. The name of the module is `G2BWT`. Our experiments on real datasets showed that `G2BWT` is competitive with the state-of-the-art algorithms that build the BWT for string collections, and that it can be the most efficient when the input is massive and with high DNA coverage.

6.1 Encoding Information with Circular Strings

We choose to build the eBWT as the strings in this representation are considered to be circular. This feature allows us to encode the paired-end information of the reads for free (see Section 4.2). Concretely, given we know a BWT position for a character in a read S_l, we can infer the sequence of its pair S_r by performing LF steps. This idea also apply backwards; we can obtain S_l provided we know a BWT position for S_r. Most genomic pipelines use the pairing information to resolve ambiguities in the DNA sequence. For instance, when assembling a genome using the overlap graph framework (Section 4.3), we might discard

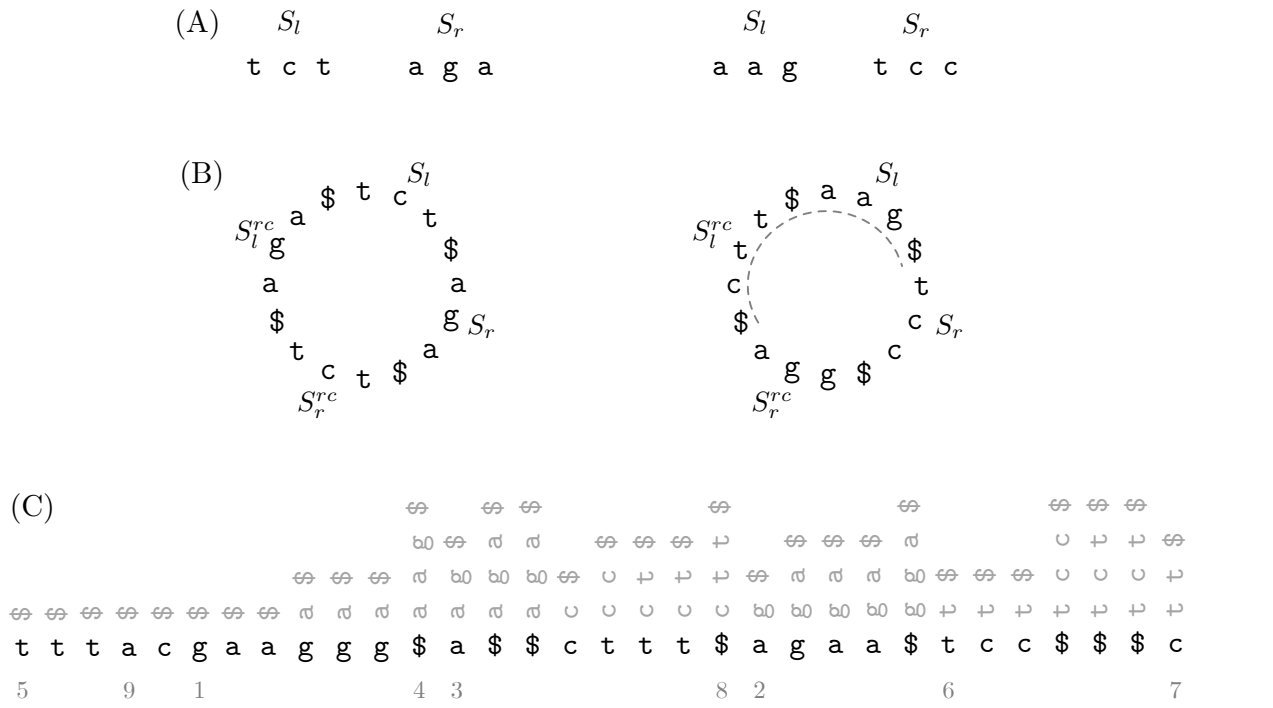

Figure 6.1: Example of an eBWT. (A) Paired-end reads {tct, aga} and {aag, tcc}. (B) The circular encodings for the reads of (A), which include their reverse complements. (C) The eBWT for the circular strings of (B). The vertical gray characters are the left contexts of the BWT symbols. The gray numbers below the BWT are the LF steps we perform to obtain the sequences of S_l = aag and its reverse complement S_l^{rc} = ctt. This operation is also depicted with a dashed line in the right circle of (B).

overlaps between reads that belong to the same pair as, in some cases, we know they are too far away in the genome as to have an overlap. In the BCR BWT (Section 3.2.1), the strings are not circular so we need an extra array to store the links explicitly.

During the construction of the eBWT, we consider every pair (S_l, S_r) to be one circular string $S_l\$S_r\$$. We can also consider the extra pair (S_r^{rc}, S_l^{rc}) with the reverse complements of (S_l, S_r). The four string are thus encoded as one circular string $S_l\$S_r\$S_r^{rc}\$S_l^{rc}\$$. By including the reverse complements in the BWT we can know which other strings in the collection overlap them. This information is necessary as, in most of the cases, we do not know the relative strands of the reads from different pairs. An example of the resulting BWT is depicted in Figure 6.1.

We note our version of the eBWT is slightly different from the one described in Section 3.2.1. The main difference is that we are including $ symbols to delimit boundaries between strings, while the original version uses a bit vector for the same purpose. The $ symbols come from the grammar of the reads, and it was not clear to us how to get rid of them as we construct the BWT.

Algorithm 1 Overview of infBWT

```
1: proc infBWT(P)                                   ▷ returns the eBWT of S*
2:     Compute the alphabets of the parses and store them in disk
3:     Load Σ^h = (L^h, R^h, f^i) from disk
4:     Compute the eBWT B^h of C from Σ^h and P
5:     for i = h to 2 do
6:         Load Σ^{i-1} = (L^{i-1}, R^{i-1}, f^{i-1}) from disk
7:         Induce B^{i-1} using B^i, Σ^i, Σ^{i-1} and P
8:         Discard B^i and Σ^i
9:         i ← i − 1
10:    return B^1
```

6.2 Definitions

Let $\mathcal{S} = \{S_1 \dots S_m\}$ be a collection of m paired-end reads. The strings in $\mathcal{S}$ are over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{n}, \mathtt{t}\}$. For simplicity, we map Σ to the range $[2, |\Sigma| + 1]$, and leave the character $\$ = 1$ as a separator symbol. We assume that for each odd position $j \in [1, m-1]$, the strings $S_j, S_{j+1} \in \mathcal{S}$ represent reads of the same pair. We also define a set $\mathcal{S}^*$ that encodes the read pairs of $\mathcal{S}$ and their reverse complements together. Each element $S_x \in \mathcal{S}^*$ is a circular string of the form $S_l\$S_r\$S_r^{rc}\$S_l^{rc}\$$, where $S_l, S_r \in \mathcal{S}$ are the reads of the same pair and S_l^{rc} and S_r^{rc} are their reverse complements, respectively. We also define the string $S = S_1 S_2 \dots S_{m/2}$ that represents the concatenation of the elements in $\mathcal{S}^*$. We do not insert extra separator symbols in S as we know that every four $ characters we have a string of $\mathcal{S}^*$. The total length of S is denoted as n. Let $\mathcal{G} = \{V, \Sigma, \mathsf{S}, \mathcal{R}\}$ be the grammar resulting from running the algorithm of Chapter 5 over S. G is the grammar size, g is the number of nonterminals and C is the string that represents the compressed version of S. The length of C is denoted as c. We also consider $\mathcal{P}$ to be the grammar tree of $\mathcal{G}$ obtained with the algorithm of Section 5.5. Let h be the number of parsing rounds LMSg performed to build $\mathcal{G}$, and let S^i be the input text for round i. We denote as $\mathcal{D}^i$ the set of phrases generated during the partition of S^i in the execution of LMSg. The operator $\prec_{LMS}$ denotes the LMS ordering of the strings (Section 3.5.2).

6.3 Overview of infBWT

We divide the algorithm in three main steps. In step one, we reconstruct the alphabet Σ^i of every S^i. We represent Σ^i using three components; L^i, R^i and f^i. The set $L^i \in [1, \sigma + g]$ stores the identifiers in $\mathcal{P}$ (see Section 5.5) for the nonterminals assigned to the symbols in S^i, the set $R^i \in [1, |L^i|]$ encodes the alphabet of S^i, and the function $f^i : L^i \to R^i$ maps an identifier in L^i to its symbol in R^i. In step two of infBWT, we compute the eBWT of C using the alphabet Σ^h. We consider the circularity of the strings compressed in C to arrange the symbols in B^h. Finally, in step three, we perform an iterative process in which we induce the eBWT B^i of S^i from the already computed transform B^{i+1} and the alphabets Σ^{i+1} and Σ^i. Once we finish the iterations, we return B^1 as the eBWT of $\mathcal{S}^*$. Algorithm 1 depicts the whole idea.

6.4 Reconstructing the Alphabets

We propose an iterative approach to reconstruct the alphabets of the parses. We proceed as follows in every iteration i; we find the loci in $\mathcal{P}$ of the nonterminals that map symbols in S^i (see Section 5.3), and insert their identifier in L^i. Subsequently, we assign ranks to the elements in L^i and store them in R^i. The computation of these ranks requires the previous triplet $(L^{i-1}, R^{i-1}, f^{i-1})$. Once we build R^i, we use the new triplet $\Sigma^i = (L^i, R^i, f^i)$ as input for the next iteration $i+1$. Our procedure requires a total of h iterations, one for each parsing round of LMSg.

We implement the function f^i by encoding L^i as a bit vector $L[1, r + \sigma]$, where $L[l]$ is set to 1 if $l \in L^i$ and 0 otherwise. Additionally, we augment L with constant-time rank support (Section 2.2.1), so that $\mathsf{rank}(L, l)$ is the number of 1s in $L[1, l]$. We store at position $R^i[\mathsf{rank}(L, l)]$ the rank associated to l.

6.4.1 Finding the Nonterminals in the Parse Tree

For deciding whether a node label in $\mathcal{P}$ belongs to L^i, we use the following lemmas:

Observation 1 Let $\mathsf{X} \to F \in \mathcal{R}$ be a nonterminal rule generated by the LMSg algorithm. Assume all the suffixes of F up to position $1 < k \leq |F|-1$ appear in more than one right-hand side in $\mathcal{R}$. After creating the RS rules in $\mathcal{G}$ (Section 5.4), every internal node v in the parse tree labeled with X will have its last $|F| - k + 1$ children recursively encapsulated from right to left inside RS nonterminals. This encapsulation pattern will generate a stair-like shape in the children of v (Figure 5.2B depicts the stair-like shape).

By using the stair-like pattern described in Observation 1, we can recognize occurrences of LMSg nonterminals just by looking at the topology of the parse tree of $\mathcal{G}$.

Lemma 1 An internal node v of $\mathcal{P}$ is the locus of a nonterminal produced in the iteration i of LMSg if its leftmost child is labeled with a symbol $l \in L^{i-1}$ and either v if the leftmost child of its parent or the left sibling of v is labeled with a symbol $l' \notin L^{i-1}$.

Proof. A nonterminal v whose first child has a label $l \in L^{i-1}$ is either an LMSg nonterminal of the iteration i or an RS nonterminal. If it is RS, then, due to the stair-like pattern, the label of its left sibling must be in L^{i-1}, otherwise v is LMSg. □

Building L^i requires us to scan the internal nodes of $\mathcal{P}$ one by one to check Lemma 1. We can mark the internal nodes that were already visited during the reconstruction of previous alphabets to avoid checking them again. Our grammar tree algorithm of Section 5.5 ensures that if a nonterminal has a dual context, then its locus in $\mathcal{P}$ is always a LMSg occurrence. In this way, building L^i requires only visiting the internal nodes of $\mathcal{P}$, not its leaves.

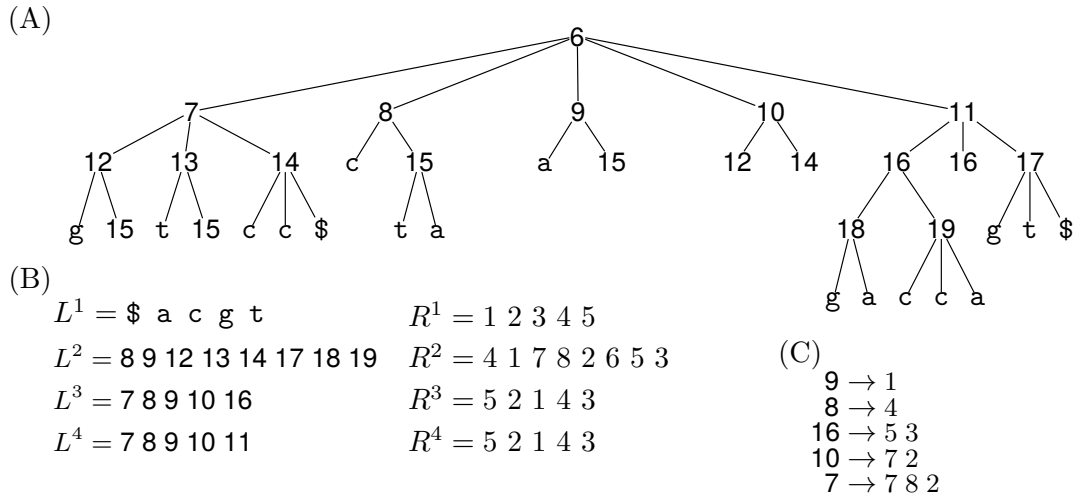

Figure 6.2: Example of alphabet reconstruction. (A) The grammar tree $\mathcal{P}$ of Figure 5.3. (B) The four dictionaries (L^i, R^i) obtained from $\mathcal{P}$. The mapping functions f^i were omitted. The labels of every set L^i, with $i > 1$, are sorted in level order. The R^i lists store the ranks of the labels in $\prec_{LMS}$ order. (C) Partial decompressions for L^3. The symbols of L^3 (left side) are arranged according to their ranks in R^3. Their partial decompressions are shown on the right side.

6.4.2 Giving Ranks to the Labels

Once we compute the labels in L^i, we need a mechanism to assign them ranks (the values in R^i). For that end, we regard L^{i-1} as a set of logical leaves in the parse tree of $\mathcal{G}$. If during the decompression of an internal node $v = \mathsf{internalselect}(l - \sigma)$, with $l \in L^i$, (Section 5.5) we reach a node v' with $\mathsf{label}(v') \in L^{i-1}$, then we do not visit its subtree but spell its symbol $f^{i-1}(\mathsf{label}(v')) \in R^{i-1}$. We concatenate all the characters in f^{i-1} spelled during the traversal of v's subtree in one single string. We refer to this string as the *partial decompression* of l, or just $\mathsf{pd}^i(l)$. Note that the set of partial decompressions obtained from L^i is actually $\mathcal{D}^{i-1}$; the dictionary of phrases generated during the partition of S^{i-1} (see Section 5.3). We sort $\mathcal{D}^i$ in $\prec_{LMS}$ order so that if $\mathsf{pd}^i(l)$ has order o in $\mathcal{D}^i$, then the associated value of l in R^i is o. Figure 6.2 shows the distinct alphabets we obtain from the grammar tree of Figure 5.3, and Example 1 shows how to implement pd^i.

Example 1 Partial decompression $\mathsf{pd}^3(7)$ of symbol $7 \in L^3$ in Figure 6.2. The symbol 7 identifies a nonterminal whose locus in $\mathcal{P}$ is the internal node $v = \mathsf{internalselect}(7 - \sigma)$. We simulate in $\mathcal{P}$ a pre-order traversal over the subtree rooted at v in the parse tree and we find that the grammar tree labels 12,13 and 14 of its children belong to L^2. We replace their values with their ranks in R^2 and insert them to the partial decompression of 7. The resulting string is 7 8 2. When the label l for pd^i identifies a transferred symbol (Section 5.3.2), it is not necessary to traverse the subtree. For instance, 8 appears in both L^3 and L^2, so $\mathsf{pd}^3(8)$ is just 4, its rank in R^2.

In practice, we sort the string in $\mathcal{D}^{i-1}$ along with their distinct proper suffixes of length > 1. The reason for this decision will be clear in Section 6.6. Maintaining all those strings

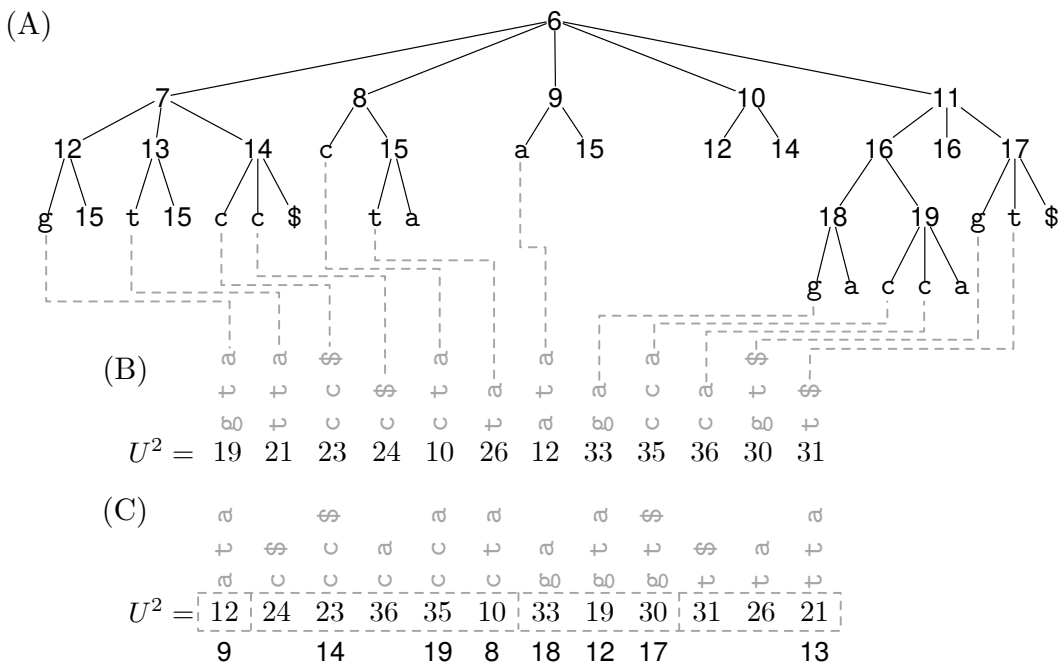

Figure 6.3: Sorting example to compute R^2. (A) The same grammar tree of Figure 5.3. (B) Array U^2 storing the grammar pointers for the distinct suffixes of length > 1 in $\mathcal{D}^1$. The gray dashed lines map the grammar pointers in U^2 to their corresponding positions in $\mathcal{P}$. The vertical strings above U^2 are the partial decompressions obtained from those pointers. (C) Array U^2 after sorting the grammar pointers according the $\prec_{LMS}$ order of their partial decompressions. The gray dashed rectangles are the distinct buckets of U^2. The number below U^2 are the labels in L^2 for the parent nodes of the grammar pointers.

in plain form during the sorting might require a lot of working memory. On the other hand, decompressing them on demand from $\mathcal{P}$ each time we access them can be slow. We came up with a practical parallel solution to solve the problem.

First, we create an array U^i storing pointers to nodes in $\mathcal{P}$. These nodes encode the distinct suffixes in $\mathcal{D}^{i-1}$ of length > 1. If we want to access the non-proper suffix of a phrase $F \in \mathcal{D}^{i-1}$, we use the leftmost child of the internal node v from which we partially decompress F. In other words, the pointer is $\mathsf{child}(\mathsf{internalselect}(l - \sigma), 1)$, with $l \in L^i$ and $\mathsf{pd}^i(l) = F$. The next case is when a proper suffix $F' = F[j..]$ is unique in $\mathcal{D}^{i-1}$. In that situation, we use the child v' of v from which we can partially decompress F'. Finally, in the case F' appears in different strings of $\mathcal{D}^{i-1}$, we use the leftmost child of the internal node v' from which we can obtain F'. Note that v' is the locus of an RS nonterminal by definition, so there is only one possible position for that node in $\mathcal{P}$. We store in U^i the level orders of these nodes to reduce the space usage. We refer to them as *grammar pointers*. We will use them again in Section 6.6.

We use counting sort to reorder the grammar pointers in U^i according the first symbol of their partial decompressions. The idea is to partition the array into buckets; all the grammar pointers whose partial decompressions are prefixed with the same symbol $b \in \Sigma^{i-1}$ appear together in the bth bucket of U^i. This presorting is fast as we can obtain the symbol b

associated to $U^i[j]$ in constant-time as $f^{i-1}(\mathsf{label}(\mathsf{nodeselect}(U^i[j])))$. Once we finish the counting sort, we sort the distinct buckets of U^i in parallel using quicksort. When we sort a bucket, we decompress the strings on demand from $\mathcal{P}$, except the pivot which we maintain in plain form. There is an overhead in decompressing the nodes on demand, but we amortize it by quick-sorting the distinct buckets in parallel. Figure 6.3 shows en example of the whole process.

6.4.3 Time Complexity for the Alphabet Reconstruction

We now give time and space upper bounds for reconstructing the alphabets of the h different parses S^i generated during the execution of LMSg (Line 2 of Algorithm 1). These upper bounds consider the procedures described in Sections 6.4.1 and 6.4.2. We summarize our results with the following theorem.

Theorem 4 The time complexity for reconstructing the alphabets of the parsing rounds of LMSg is $\mathcal{O}((g+Gf)h)$ time, where f is longest right-hand side rule in the grammar of $\mathcal{P}$ and h is the number of parsing rounds of LMSg. This task requires $\mathcal{O}(Gh \log G)$ bits of working space on top of $\mathcal{P}$.

Proof. Let us first analyze the time complexity for constructing one alphabet Σ^i. The first step is to visit the internal nodes of $\mathcal{P}$ to check which of them have labels in L^i. We use the LOUDS function internalselect to move from one internal node to the next in $\mathcal{O}(1)$ time. Additionally, checking if a node meets Lemma 1 requires us to perform a constant number of operations. Therefore, as $\mathcal{P}$ has $g+1$ internal nodes, the construction of L^i takes $\mathcal{O}(g)$ time. The next step is constructing R^i. Our approach considers all the *distinct* suffixes of length > 1 in $\mathcal{D}^{i-1}$. There cannot be more than G of such suffixes in $\mathcal{D}^{i-1}$ as each one corresponds to a distinct symbol in the right-hand sides of $\mathcal{P}$'s grammar. On the other hand, these cannot be more than f symbols long as this value is the maximum length a right-hand side can have. Thus, the number of symbols we have to process to produce R^i is $\mathcal{O}(Gf)$. To obtain the desired time complexity, we can use the sorting algorithm for strings described in Lemma 8.7 of Mäkinen et al. [120]. Given a string collection $\mathcal{W} = \{W_1, \ldots, W_n\}$ over the alphabet $\Sigma = [1, \sigma]$, and with a total of N symbols, this algorithm sorts $\mathcal{W}$ in $\mathcal{O}(\sigma + N)$ time and uses $\mathcal{O}(N \log N)$ bits of working space. In our case, the alphabet of the suffixes is g, and N is Gh, so the complexities for sorting the suffixes become $\mathcal{O}(g + Gh)$ time and $\mathcal{O}(Gh \log G)$ bits of working space.

We have to reconstruct L^i and R^i h times, one for every parsing round of LMSg. Consequently, the time complexity for reconstructing all the alphabets is $\mathcal{O}((g + Gf)h)$ time. After computing (L^i, R^i), we can discard all the auxiliary data structures to obtain the next pair (L^{i+1}, R^{i+1}). In this way, the space complexity for reconstructing the alphabets remains in $\mathcal{O}(Gh \log G)$ bits. □

We use quicksort to produce R^i instead of the algorithm described in Mäkinen et al. as it is faster in practice and does not require auxiliary data structures. With this change, the space complexity to produce the alphabets decreases to $\mathcal{O}(G \log G)$ bits on top of $\mathcal{P}$, which stands for the array U^i with the grammar pointers. Quicksort incurs in no more than

$G \log G$ comparisons on average to sort the $\mathcal{O}(G)$ suffixes encoded in U^i. In each of these comparisons, we need to partially decompress a phrase from $\mathcal{P}$. If we replace the Huffman representation of $\mathcal{P}$'s labels (array Z of Section 5.5) with an array using fixed-length cells of $\log(g + \sigma)$ bits, then we can access the grammar tree labels in $\mathcal{O}(1)$ time. This modification allows us to partially decompress a phrase F from $\mathcal{P}$ in $\mathcal{O}(|F|)$ time and thus speed up the suffix comparisons during the execution of quicksort.

Note that the analyses for reconstructing the alphabets are rather pessimistic. Assuming that the number of distinct suffixes in $\mathcal{D}^{i-1}$ is G implies that this dictionary stores all the right-hand sides of $\mathcal{P}$'s grammar. Further, assuming that every dictionary has G distinct suffixes implies that all the right-hand sides of the grammar were generated in the first parsing round of LMSg, and later were transferred to subsequent parsing rounds, which is not possible. However, we could not find better upper bounds. The alphabet reconstruction depends on $t = \sum_1^h |\mathcal{D}^i|$. It is unclear which is the maximum number of distinct phrases we can generate in a parsing round (value for $|\mathcal{D}^i|$). On the other hand, it is also unclear which is maximum value for t as the dictionaries are not disjoint. It is a value in $G < t < Gh$.

6.5 Computing the eBWT of the Compressed Text

Unlike the regular BWT, the position of each $C[j]$ in our version of the eBWT does not depend on the whole suffix $C[j + 1..]$, but on the string $S' = C[j + 1, j + p']C[j - p, j]$. This sequence is a circular permutation of the compressed version of some string $S_x = S_l\$S_r\$S_r^{rc}\$S_l^{rc}\$ \in \mathcal{S}^*$ encoded in the range $C[j - p, j + p']$. Computing S' from $\mathcal{P}$ is simple as $\mathcal{G}$ is string independent (see Definition 6). This feature means that if we recursively expand every symbol of S' and concatenate the result, then we obtain the exact sequence of S_x. We do not have to deal with border cases in which the prefix of S_x is a proper suffix in the recursive expansion of $C[j - p]$ or cases in which a suffix of S_x is a proper prefix in the recursive expansion of $C[j + p']$.

For constructing the eBWT of C we require $\mathcal{P}$ and the alphabet $\Sigma^h = (L^h, R^h, f^h)$. Given the definition of $\mathcal{P}$, we can easily obtain the root child v encoding $C[j]$ as $v = \mathsf{nodeselect}(j + 1)$. Once we retrieve v, we obtain $C[j]$ with $f^h(\mathsf{label}(v))$. For accessing the circular string S' from $C[j]$, we define the function cright. This procedure receives as input a position $j \in [1, c]$ and returns another position $j' \in [1, c]$ such that $C[j']$ is the circular right context of $C[j]$. We use cright as the underlying operator for another function, ccomp. This method compares lexicographically two circular permutations located at different positions of C. Similarly, we define a function cleft that returns the circular left context of $C[j]$. We use cleft to get the eBWT symbols once we sort the circular permutations. To support these operations, we consider the border cases $C[u' + 1] = C[u]$ and $C[u - 1] = C[u']$ for every $S_x \in \mathcal{S}^*$. These exceptions require us to include a bit vector $E[1, c]$ that marks as $E[j] = 1$ every jth root child of $\mathcal{P}$ such that $j \bmod 4 = 0$ and its recursive expansion is suffixed by \$. The functions cleft, cright and ccomp are described in Algorithm 2.

We start the computation of the eBWT of C by creating a table $A[1, c]$ with $|R^h|$ lexicographical buckets. Then, we scan the children of the root of $\mathcal{P}$ from left to right, and for every node v, we store its child rank in the leftmost available cell of bucket $f^h(\mathsf{label}(v))$ in A. This process yields a partial sorting of circular permutations of C; every bucket b contains

Algorithm 2 Functions to simulate circularity over C

Require: A bitmap $E[1, |C|]$ marking the symbols of C expanding to phrases suffixed by \$.

```
1: proc cright(j)          ▷ returns a j' such that C[j'] is the circular right context of C[j]
2:     if E[j] then
3:         j ← j − 1
4:         while U[j] is false do
5:             j ← j − 1
6:     return j + 1

7: proc cleft(j)           ▷ returns a j' such that C[j'] is the circular left context of C[j]
8:     if U[j − 1] then
9:         while U[j] is false do
10:            j ← j + 1
11:        return j
12:    else
13:        return j − 1

14: proc ccomp(a,b)        ▷ circular lexicographical comparison of C[a] and C[b]
15:    r1 ← f^h(label(nodeselect(a + 1)))
16:    r2 ← f^h(label(nodeselect(b + 1)))
17:    while r1 ≠ r2 do
18:        a ← cright(a), b ← cright(b)
19:        r1 ← f^h(label(nodeselect(a + 1)))
20:        r2 ← f^h(label(nodeselect(b + 1)))
21:    return r1 < r2
```

the permutations that start with symbol b. To finish the sorting, we apply a local quicksort in every bucket using ccomp as the comparison function (something similar to what we did in Section 6.4.2). Finally, we produce B^h by scanning A from left to right and appending every symbol $f^h(\mathsf{label}(\mathsf{nodeselect}(\mathsf{cleft}(A[j]) + 1)))$ with $j \in [1, |A|]$.

6.6 Inducing the eBWT

This section describes a method called nextBWT, which induces the extended B^{i-1} of the parse S^{i-1} from the already computed eBWT B^i of the parse S^i (Line 7 of Algorithm 1). For this task, we consider an extra function f^i_{inv} that maps a symbol $B^i[j] \in R^i$ to its respective label $l \in L^i$. We use this new function to partially decompress the phrase $F = \mathsf{pd}^i(l) = S^{i-1}[u, u'] \in \mathcal{D}^{i-1}$ associated with $B^i[j]$ ($l = f^i_{inv}(B[j])$). The general idea of nextBWT is to decompress all the phrases in S^{i-1} from B^i and place their symbols in B^{i-1}. We note that, in most of the cases, the suffix $F[u + 1..]$ gives us enough right context to place $F[u]$ in B^{i-1}. When this information is not sufficient, we complete the operation by using the (circular) partial ordering of B^i.

Lemma 2 Let A and B be distinct strings of lengths $a, b > 1$ (respectively) that appear as suffixes in two or more phrases of $\mathcal{D}^{i-1}$. Additionally, let $S^{i-1}[o, o + a - 1] = A$ and $S^{i-1}[p, p + b - 1] = B$ be any pair of occurrences of these strings as suffixes in phrases of S^{i-1}. Assume these occurrences of A and B are prefixes in two substrings $S^{i-1}[o, j]$ and $S^{i-1}[p, j']$ (respectively) that recursively expand to different suffixes of $\mathcal{S}^*$. If $A \prec_{LMS} B$, then $S^i[o, j]$ is lexicographically smaller than $S^i[p, j']$.

Proof. Clearly, if A and B are not one a prefix of the other, then their $\prec_{LMS}$ order is that of the expanded strings, $S^{i-1}[o, j]$ and $S^{i-1}[p, j']$. The problem arises when one string is prefix of the other.

This situation does not happen if one of them is a border phrase (Section 5.3.1) or a transferred symbol (Section 5.3.2). Border phrases never occur as prefixes of other phrases because their last symbols always recursively expand to strings in Σ suffixed by a \$, and due to the string independence of $\mathcal{G}$, this character cannot lie within a phrase. In the case of transferred symbols, they have frequency one in S^{i-1}.

The only scenario in which A can be a prefix of B (or vice-versa) is when both are suffixes of LMS phrases (Section 5.3). Still, we can obtain their $\prec_{LMS}$ orders by inspecting the suffix classification of their symbols. Let a string D over the alphabet $[0, 1]$ be the *description* of an LMS phrase F. If $F[j]$ is L-type, then $D[j] = 1$ and if $F[j]$ is S-type or LMS-type, then $D[j] = 0$. Now consider the set $\mathcal{U}$ with the descriptions of all the LMS phrases of $\mathcal{D}^{i-1}$. As the pattern $LS = 10$ only appears as a suffix in the descriptions, $\mathcal{U}$ is a prefix-free set. Therefore, if A is a prefix of B or vice-versa, then we can still decide their orders as long as both have length more than one. □

Now let us go back to the partial decompression F extracted from $B^i[j]$. We can use Lemma 2 to obtain the $\prec_{LMS}$ order of every distinct suffix $F[u + 1..]$ of length > 1 and thus estimate an range for position for $F[u]$ in B^{i-1}. In particular, if $F[u + 1..]$ has rank b among the other suffixes in $\mathcal{D}^{i-1}$, then $F[u]$ belongs to the bth block of B^{i-1}, where a block is a contiguous segment of B^{i-1} containing symbols that are followed in S^{i-1} by the same suffix $F[u + 1..]$. In the following, we refine this idea to complete the induction of B^{i-1}.

The problem with the method described above is that we cannot obtain the $\prec_{LMS}$ order for the last suffix $s = F[|F|]$ as it does not have the minimum length > 1 for Lemma 2. We solve this limitation by building an FM-index of B^i. Thus, in addition to obtaining F, we also compute the partial decompression F' from $B^i[\mathsf{LF}^{-1}(j)]$, the (circular) right context of $B^i[j]$. Our purpose is to obtain the right extension sF' so we get enough information to find the range for symbol $F[|F| - 1]$ in B^{i-1}. We refer to sF' as an *artificial string* because it does not necessarily exist in $\mathcal{G}$ due to the sequences' circularity. The FM-index also helps us to find the symbols that precede F in S^{i-1}. We obtain the phrase F'' from $B^i[\mathsf{LF}(j)]$ and we place the last symbol of F'' in the bth block of B^{i-1}, where b is the $\prec_{LMS}$ order of F.

Now we have all the necessary elements to describe how to assign every $F[u] \in R^{i-1}$ extracted from B^i to a specific block in B^{i-1}. We consider a new set $\mathcal{D}^{i-1}_{ext}$ that contains all the distinct suffixes of length > 1 in $\mathcal{D}^{i-1}$, the transferred symbols in $\mathcal{D}^{i-1}$, and the artificial strings obtained from B^i. We use $\mathcal{D}^{i-1}_{ext}$ to induce a partition over B^{i-1}; every block in this

partition stores the symbols that are followed in S^{i-1} by the same phrase $A \in \mathcal{D}_{ext}^{i-1}$. Thus, if $F[u]$ is followed by A in S^{i-1}, then it belongs to the bth block of B^{i-1}, where b is the $\prec_{LMS}$ order of A in $\mathcal{D}_{ext}^{i-1}$. The only thing left to compute is the relative order of the symbols within the blocks of B^{i-1}. We note we can induce these orders from B^i.

Lemma 3 Let $B^i[j]$ and $B^i[j']$ be two BWT symbols at different positions j and j', with $j < j'$, and whose pd^i phrases are F and F', respectively. Also let P_j and $P_{j'}$ be suffixes of F and F' with the same sequence $P \in \mathcal{D}_{ext}^{i-1}$. The occurrence P_j is lexicographically smaller than $P_{j'}$.

Proof. As P_j and $P_{j'}$ are equal, their relative orders depend on the lexicographical ranks of the phrases to the (circular) right of F and F' in S^{i-1}. As $B^i[j]$ appears before (from left to right) than $B^i[j']$, the right context of P_j is lexicographically smaller than the right context of $P_{j'}$. □

If we generalize Lemma 3 to $x \geq 1$ occurrences of P, then we can use the following lemma for building the block of B^{i-1} associated with P:

Lemma 4 Let $P \in D_{ext}^{i-1}$ be a string with x occurrences as a suffix in the phrases of S^{i-1}. Let $J = j_1, j_2, \ldots, j_x$ be a strictly increasing list of integers. Every $B^i[j_o]$, with $j_o \in J$, is a position where the partial decompression $\mathsf{pd}^i(B^i[j_o])$ is suffixed by P. Assume we scan J from left to right, and for every j_o, we extract the symbol in S^{i-1} that precedes the occurrence $B^i[j_o]$ of P. The resulting list of symbols matches the block in B^{i-1} for P.

Proof. Because of Lemma 3, we know that the suffix of S^{i-1} prefixed by the occurrence $B^i[j_o]$ of P is lexicographically smaller than the suffix prefixed by the occurrence $B^i[j_{o+1}]$. This holds for every j_o, with $o \in [1, x-1]$. In other words, the suffixes of S^{i-1} prefixed by P are already sorted in lexicographical order in J. Now suppose we access the occurrences of P in S^{i-1} in the same order they are encoded in J and append their preceding symbols into a list O_P. The sequence of the resulting list O_P will match a range of B^{i-1}. That range will be the bth block in the partition induced by $\mathcal{D}_{ext}^{i-1}$, where b is the $\prec_{LMS}$ order of P among the strings in $\mathcal{D}_{ext}^{i-1}$. □

We use Lemma 4 to build all the distinct blocks of B^{i-1} in one linear scan of B^i. Then, we use Lemma 2 to sort the blocks according their right contexts. More specifically, suppose the symbols in block $B^{i-1}[o, o']$ are followed by the same string $A \in \mathcal{D}_{ext}^{i-1}$. If A has $\prec_{LMS}$ order b in the set, then $B^{i-1}[o, o']$ is the bth block of B^{i-1}. Note that the number of strings we sort to get B^{i-1} is small compared to its size. We use one string per block of B^{i-1}, regardless of the block length. On the other hand, it is not necessary to maintain these strings in plain form as we can access them from $\mathcal{P}$. In the following, we will see that most of these right context strings were already sorted in a previous step of infBWT, so the whole process of building B^{i-1} is not exhaustive.

We implement nextBWT as follows; we create two empty list Q and Q'. Then, we start

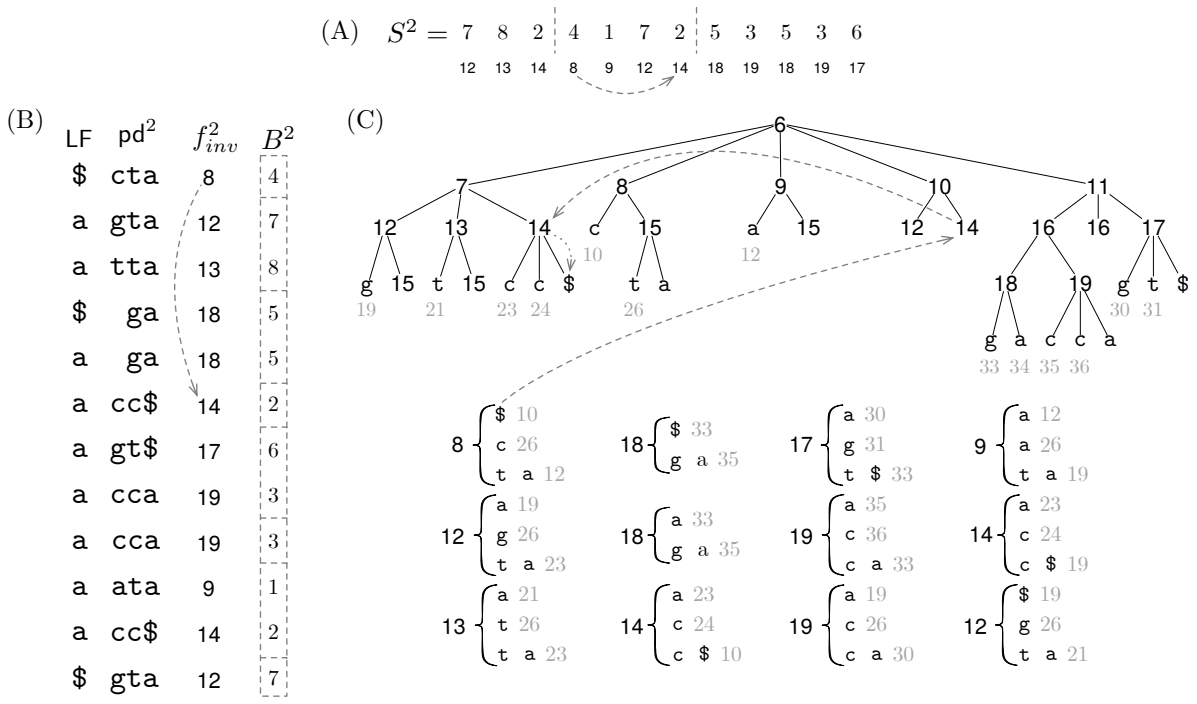

Figure 6.4: Example of nextBWT. (A) The parse S^2 of Figure 5.1. The dashed vertical lines are the boundaries between the original strings. The numbers below S^2 are the nonterminals in $\mathcal{P}$ assigned to each symbol. The dashed arrow indicates the circular left context of $S^2[4]$. (B) The eBWT B^2 of S^2. The dashed rectangles are the suffix array buckets. Column f^2_{inv} contains the labels in L^2 for the symbols in B^2. The strings in column pd^2 are the phrases in $\mathcal{D}^{i-1}$ (partial decompressions) obtained from B^2. The symbols in column LF are the circular left contexts in S^1 for the strings in f^2_{inv}. The dashed arrow indicates an LF step for $B^i[1]$. (C) The information we retrieve the scan of B^2. Each curly bracket contains the information from one position of B^2. These curly brackets are read from top to bottom and from left to right. The symbol to the left of a curly bracket is the label $l \in L^i$ for the symbol in B^2. The tuples to the right are the elements we append to Q or Q'. They gray values below some nodes of $\mathcal{P}$ are their level-orders, and correspond to the grammar pointers we insert into Q or Q'. The dashed arrows indicate the steps to get the circular right context with an LF step.

a scan of B^i from left to right. For every $B^i[j]$, we obtain first the partial decompression F' from $B^i[\mathsf{LF}(j)]$ and insert the pair $(F'[|F'|], p)$ to Q, where p is the grammar pointer (Section 6.4.2) from which we obtain the sequence of $F = \mathsf{pd}(l)$, with $l = f^i_{inv}(B^i[j])$. Then, for every suffix $F[u+1..]$ of length > 1, we insert the pair $(F[u], p)$ to Q, where p is the grammar pointer for the sequence $F[u+1..]$. After consuming F, we obtain the label $l' = f^i_{inv}(B^i[\mathsf{LF}^{i-1}(j)]) \in L^i$ that identifies in $\mathcal{P}$ the nonterminal assigned to $B^i[\mathsf{LF}^{-1}(j)]$. Finally, we insert the triplet $(F[|F|-1], F[|F|], p)$ to Q', where p is the grammar pointer from which we obtain the sequence of $\mathsf{pd}^i(l')$. Figure 6.4 shows the related concepts.

After the scan of B^i, the next step is to merge Q and Q' to produce B^{i-1}. The idea is simple; we stably sort Q by the $\prec_{LMS}$ order of the partial decompressions (second component). Subsequently, we stably sort Q' by the $\prec_{LMS}$ order of the artificial strings (second and third components). Finally, we combine the first components of Q and Q' in B^{i-1}. This last step

Algorithm 3 Inferring B^{i-1}

Require: $\mathcal{P}$

1: **proc** nextBWT($f^i_{inv}, L^{i-1}, R^{i-1}, f^{i-1}$)
2: $Q \leftarrow Q' \leftarrow \varnothing$
3: **for** $j = 1$ to $|B^i|$ **do**
4: $F' \leftarrow \mathsf{pd}^i(f^i_{inv}(B^i[\mathsf{LF}(j)]))$
5: $v \leftarrow \mathsf{internalselect}(f^i_{inv}(B^i[j]) - \sigma)$
6: push pair $(F'[|F'|], \mathsf{nodemap}(v))$ to Q
7: **if** $\mathsf{label}(v) \notin L^{i-1}$ **then** ▷ partial decompression of $\mathsf{label}(v)$
8: $b \leftarrow f^{i-1}(\mathsf{label}(\mathsf{child}(v, 1)))$
9: $y \leftarrow \mathsf{child}(v, 2)$
10: **while** true **do**
11: **if** $\mathsf{nsibling}(y) \neq 0$ **then** ▷ y is not the rightmost child of its parent
12: push pair $(b, \mathsf{nodemap}(y))$ to Q
13: $b \leftarrow f^{i-1}(\mathsf{label}(y))$
14: $y \leftarrow \mathsf{nsibling}(y)$
15: **else**
16: **if** $\mathsf{label(y)} \notin L^{i-1}$ **then** ▷ RS nonterminal
17: $y \leftarrow \mathsf{child}(\mathsf{internalselect}(\mathsf{label}(y) - \sigma), 1)$
18: push pair $(b, \mathsf{nodemap}(y))$ to Q
19: $b \leftarrow f^{i-1}(\mathsf{label}(y))$
20: $y \leftarrow \mathsf{nsibling}(y)$
21: **else** ▷ rightmost symbol in the partial decompression
22: $z \leftarrow \mathsf{child}(\mathsf{internalselect}(f^i_{inv}(B^i[\mathsf{LF}^{-1}(j)]) - \sigma), 1)$
23: push triplet $(b, f^{i-1}(\mathsf{label}(y)), \mathsf{nodemap}(z))$ to Q'
24: **break**
25: Sort Q by second component and Q' by second and third components
26: $B^{i-1} \leftarrow$ merge $Q.first$ and $Q'.first$
27: **return** FM-index of B^{i-1}

is equivalent to merging two sorted lists. Let a *block* be a contiguous range in Q where all the pairs contain the same grammar pointer p as second component. Similarly, a block in Q' is a contiguous range where all the tuples have the same combination of second and third components (b, l). Let q and q' the current blocks of Q and Q' (respectively) in the merge. If the partial decompression encoded by p has a smaller $\prec_{LMS}$ order than the artificial string encoded by (b, l), then we append to B^{i-1} the first components of block q and move forward to the next block $q + 1$. On the other hand, if the partial string of (b, l) is smaller, then we insert the first components in the block q' and move forward to next block $q' + 1$. The procedure of nextBWT is explained in more detail in Algorithm 3.

It might seem like that merging Q and Q' requires to decompress several strings from $\mathcal{P}$ and then sort them, but it does not. On one side, the distinct grammar pointers of Q were already sorted in Section 6.4.2, and stored in the array U^i. The only thing left is to reorder Q according to them. More specifically, we scan Q from left to right, and if the second component of a pair appears in the jth cell of U^i then we stably move that pair to the jth block of Q. On the other hand, the artificial strings of Q' are partially sorted by the third

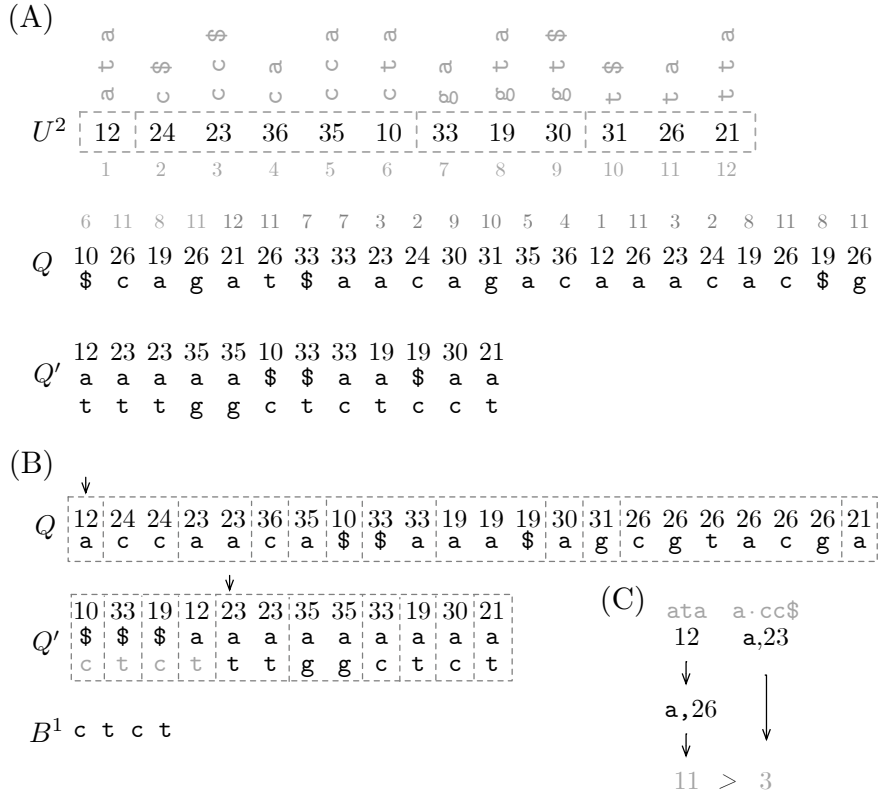

Figure 6.5: The merge of Q and Q' in nextBWT. (A) The array U^2 of Figure 6.3 and the lists Q and Q' obtained from Figure 6.4(C). The gray symbols above Q are the positions where the grammar pointers occur in U^2. (B) Sorted versions of Q and Q' and the partially constructed B^1. The dashed boxes are the sorting blocks of Q and Q'. Arrows above Q and Q' indicate the current blocks in the merge. Symbols in gray in Q' were already inserted into B^1. (C) Comparison of the sorting blocks of Q and Q'. The grammar pointer for the block in Q is 12 (ata). The artificial string for the block of Q' is (a, 23) (a·cc$), where 23 is the grammar pointer for cc$. As both strings start with a, their orders are decided by the next suffixes ta and cc$. The grammar pointer of ta is 26, and its occurrence in U^2 is at position 11. On the other hand, the grammar pointer of cc$ occurs at position 3 in U^2. Hence, the first components in the block of Q' go first in B^1.

component. Let $Q'[x]$ and $Q'[y]$ be two different triplets, with $x < y$. Also let p_x be the third component of $Q'[x]$ and p_y be the third component of $Q'[y]$. If $p_x \neq p_y$, then the partial decompression referenced by p_x has a smaller $\prec_{LMS}$ order than the partial decompression of p_y as $Q'[x]$ appears first than $Q'[y]$. As a consequence, reordering Q' reduces to a stable sort by the symbol in the second component.

When inserting the symbols to B^{i-1}, we have to compare the strings that represent the blocks q in Q and q' in Q'. Suppose the grammar pointer for block q is p_x and the pair of second and third components for block q' is (b, p_y). Then, the string $X = \mathsf{pd}^i(\mathsf{label}(\mathsf{nodeselect}(p_x)))$ represents the block for Q and $Y = b \cdot \mathsf{pd}^i(\mathsf{label}(\mathsf{nodeselect}(p_y)))$ represents the block for Q'. If $X[1]$ is equal to $Y[1] = b$, then the relative order of X and Y is decided by the suffixes $X[2..]$ and $Y[2..]$. Still, references to these strings already exist in U^i, and we know they

are in $\prec_{LMS}$ order. Therefore, it is not necessary to decompress X and Y from $\mathcal{P}$. In U^i, the cell whose value is $\mathsf{nodemap}(\mathsf{child}(\mathsf{nodeselect}(p), 2))$ references $X[2..]$ and the cell whose value p_y references $Y[2..]$. From these two cells, the rightmost one indicates which string, X or Y, has the greatest $\prec_{LMS}$ order. Figure 6.5 shows an example of how to merge Q and Q' using array U^i.

6.7 Implicit Occurrences of the LMS Phrases

So far, we have assumed that the strings in $\mathcal{D}_{ext}^{i-1}$ that we partially decompress from B^i (those we insert in Q) always appear as suffixes in the phrases of S^{i-1}, but this is not always the case as sometimes they occur in between phrases.

Definition 7 Let $F = XYZ$ be a LMS phrase of S^{i-1}. The position $S^{i-1}[j, j+2] = F$ is said to be an implicit occurrence of F if, during the parsing of LMSg, $S^{i-1}[j] = \mathsf{X}$ becomes a suffix of the phrase at $S^i[j-p, j]$, with $p \geq 1$, and $S^i[j+1, j+2] = YZ$ is considered another phrase.

When we execute $\mathsf{nextBWT}$, the implicit occurrences of F are inserted to Q', but the explicit occurrences are inserted to Q. The problem with that situation is that, in the construction of B^{i-1}, F is the right context for two sorting blocks, one in Q and other in Q'. We know they represent, in practice, one single block of B^{i-1}, but we do not know how to merge their symbols. We need to detect the implicit occurrences of F during the scan of B^i and insert them into Q to fix the problem. We note that an implicit occurrence appears when $F[1]$ is classified as LMS-type during the parsing of S^{i-1}. We use the following lemma to detect this situation:

Lemma 5 The locus v of F in $\mathcal{P}$ has two children, the left one has a label in L^{i-1} and the right one as a label in L^i are encodes the occurrence of an RS nonterminal.

Proof. Let $F = XYZ$ and $A = YZ$ be two LMS phrases in the partition of S^{i-1}. Their grammar rules are $\mathsf{F} \rightarrow \mathsf{XYZ}$ and $\mathsf{A} \rightarrow \mathsf{YZ}$, respectively. As YZ is a repeated suffix, LMSg has to create a RS nonterminal for it, but it already exists, it is A. Thus, LMSg reuses it and replaces YZ with A in the right-hand side of F's rule. Now A becomes a nonterminal with dual context. □

Before running $\mathsf{nextBWT}$, we scan L^i from left to right to find the internal nodes of $\mathcal{P}$ that meet Lemma 5. For every node v' that meets the lemma, we create a pair (l_l, l_r) with the labels in $\mathcal{P}$ of its left and right children, respectively. Then, we record the pair in a hash table $\mathcal{H}$ associated with the value $p = \mathsf{nodemap}(\mathsf{child}(v', 1))$ (a grammar pointer). During the execution of $\mathsf{nextBWT}$, when we obtain the partial decompression F' from $B^i[\mathsf{LF}(j)]$ (Line 4 of Algorithm 3), we add an extra step. We check if the pair formed by the label in L^{i-1} of $F'[|F'|]$ and the label $f_{inv}^i(B^i[j]) \in L^i$ has an associated value p in $\mathcal{H}$. If that happens, then we insert $(F'[|F'|-1], p)$ to Q. Equivalently, when we process the partial decompression F of $B^i[j]$, we check if the pair formed by the label in L^{i-1} of $F[|F|]$ and

the label $l = f^i_{inv}(B^i[\mathsf{LF}^{-1}(j)])$ has a value in $\mathcal{H}$. If so, then we do *not* insert the triplet $(F[|F|-1], F[|F|], p)$ to Q', with $p = \mathsf{nodemap}(\mathsf{child}(1, \mathsf{internalselect}(l - \sigma)))$. This last step avoids inserting duplicated symbols in B^{i-1}.

6.8 Inducing the BWT in Run-Length Compressed Space

In every new round i of nextBWT, the size of B^i increases, but its alphabet decreases. This pattern ensures that, in the last iterations of infBWT, we partially decompress a small number distinct phrases from B^i several times. To reduce the time overhead produced by this monotonous way of decompression, we exploit the runs of equal symbols in B^i. More specifically, if a given range in $B^i[j, j']$ of length $x = j' - j + 1$ has the same symbol $s \in R^i$, then we partially decompress the associated phrase once and multiply the result by x instead of performing the same operations x times. In other words, suppose the partial decompression from label $f^i_{inv}(s) \in L^i$ yielded a list of pairs of the type (b, p) and one tuple (b, b', p'), where b and b' are symbols in R^{i-1} and p and p' are grammar pointers. The pairs of type (b, p) are those we insert to Q while the tuple (b, b', p) is the one we insert to Q'. By Lemma 3, we know that the x copies of (b, p) appear in a consecutive range of B^{i-1}. The same applies for the x copies of (b, b', p'). Therefore, when we do not insert every (b, p) x times to Q, but insert (x, b, p) only once. We do the same for the triplet (b, b', p') of Q'.

It is also probable that in the last iterations of infBWT, BWT runs lying close to each other in B^i will have the same symbol. Consider, for instance, three runs (x, s), (y, s') and (z, s) placed consecutively at some range of B^i. The values $s, s' \in R^i$ are the run symbols and x, y and z are their lengths. In the current version of nextBWT, we process (x, s) and (z, s) independently as we are unaware that they are close to each other. However, these runs are contiguous occurrences of s in B^i, so it is safe to insert $(x + z, b, p)$ to Q instead of two separated pairs (Lemma 3).

We modify the scan of B^i so that when we visit the second run (z, s), we do not insert its pairs (z, b, p) into Q. Instead, we increment the first component of rightmost tuple (x, b, p) in Q by z. To implement this idea, we maintain a small hash table that records the information of the last visited symbols of B^i. When we reach a new run (x, s), we check first if s exists as key in the hash table. If so, we extract the rightmost positions in Q for the distinct pairs (b, p) in the partial decompression $\mathsf{pd}^i(f^i_{inv}(s))$, and increment their first components by x. If s is not in the hash table, then we partially decompress its phrase from scratch and store its information in the hash table. We can limit the size of the hash table so it is always maintained in some of the L1-3 caches, and when we exceed this limit, we just simply reset the hash. Figure 6.6 shows an example.

Another way of exploiting the equal-symbol runs of B^i is with the strings in $\mathcal{D}^{i-1}_{ext}$ that are unique in the grammar. Consider an internal node v of $\mathcal{P}$ whose partial decompression $\mathsf{pd}^i(\mathsf{label}(v))$ is F. If the jth child y of v from left to right has a label in L^{i-1}, then the suffix $F[j..] \in \mathcal{D}^{i-1}_{ext}$ is decompressed only from y. We know this because y was not encapsulated by any RS nonterminal during the grammar construction. This property means that if the symbol $s = f^i(\mathsf{label}(v)) \in R^i$ has x occurrences in B^i, then the string $F[j..]$ has x occurrences in S^{i-1} as well. All these occurrences are preceded by the same symbol $F[j-1]$. Therefore, we can insert $(x, F[j-1], \mathsf{nodemap}(y))$ to Q only once and forget about $F[j..]$ during the scan

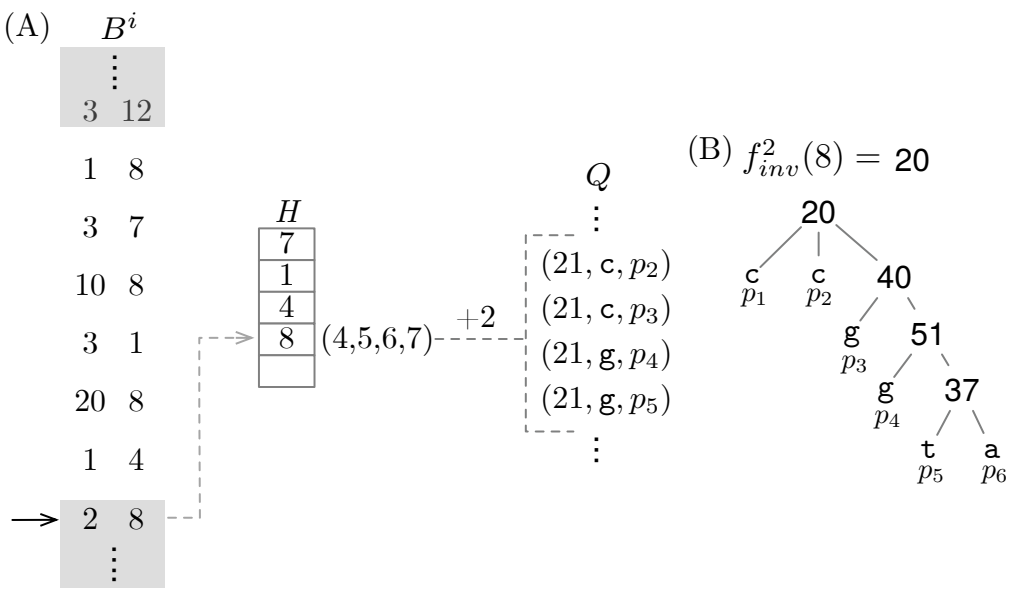

Figure 6.6: Scanning B^i in run-length compressed space. (A) Run-length compressed version of B^i. The right column has the run heads and the left column has the run lengths. The arrow to the left indicates that the scan of B^i is currently at the run $(2, 8)$. The information of the runs within the white segment of B^i is stored in the hash table H. For each of these runs, we have the indexes in Q where the pairs of their partial decompressions were stored. For simplicity, H only shows the information for 8. The positions in Q for the pairs in the partial decompression of 8 are $(4, 5, 6, 7)$. In those pairs, we increase the first component (the frequency) by 2, which is the length of the run $(2, 8)$. The grammar subtree for the nonterminal assigned to 8 is depicted in (B).

of B^i. Computing all the distinct strings like $F[j..]$ from B^i is simple; for each label $l \in L^i$, we get the internal node $v = \mathsf{internalselect}(l - \sigma)$, and its number of children $a = \mathsf{nchildren}(v)$. If $a > 2$, then we count the x occurrences of $s = f^i(l) \in R^i$ in B^i using the rank operation in the FM-index of B^i. Then, for every child y of v whose child rank is in the range $[2, a-1]$, we push the pair $(x, b, \mathsf{nodemap}(y))$ to Q, where $b \in R^{i-1}$ is the symbol obtained from the left sibling of y. Then, during the scan of B^i, we just skip these y nodes. In Figure 6.6(A), the grammar pointer p_2 encodes a substring `cggta` that only occurs under the nonterminal 20 in the grammar.

6.8.1 Practical Considerations of nextBWT

The lists Q and Q' can be large. However, as we access them linearly, it is not necessary to have them completely in main memory during the scan of B^i. We can represent them as semi-external vectors. Additionally, we can build them in parallel. Let p the number of working threads. We divide B^i into p chunks and we build an independent pair of lists Q_u and Q'_u per each chunk $u \in [1, p]$ of B^i. Once we finish the scan, we merge all the Q_u lists in one single Q and all the Q'_u in another list Q' and continue as before. The parallel scan of B^i help us to amortize the partial decompressions of the BWT symbols.

Once we infer B^{i-1}, we produce a RLFM-index from it. This task is not difficult as the

Number of collections	Plain size (GB)	Compressed size (GB)	% BWT runs
1	12.77	3.00	31.46
2	23.43	5.30	26.11
3	34.30	7.41	22.70
4	45.89	9.38	20.12
5	57.37	11.31	18.74

Table 6.1: Input datasets for building multi-string BWTs. The compressed size is the space of the LPG representation. We measured the percentage of BWT runs in a dataset (fourth column) as the number of equal-symbol runs in its multi-string BWT divided by its total number of symbols and multiplied by 100. We used the multi-string BWT produced by eGap to perform the computations.

resulting B^{i-1} is actually half-way of being run-length compressed. The only drawback of this representation is that manipulating B^{i-1} can be slow. The RLFM-Index usually represents the BWT using a wavelet tree (Section 2.2.2). In our case, this feature implies that accessing a symbol in B^i or performing rank has a slowdown factor of $\mathcal{O}(\log r)$. This value can be too slow for our purposes. In practice, we use a bit-compressed array to represent B^{i-1} instead of a Wavelet Tree. We also include an integer vector K of the same size of B^{i-1}. In every $K[j]$ we store the rank of symbol $B^{i-1}[j]$ up to position j. Thus, when we need to perform LF over $B^{i-1}[j]$, we replace the rank part in the equation with $K[j]$. Notice it is not necessary to fully load K into main memory as its access pattern is linear. We load it in chunks as we scan B^{i-1} during iteration $i-1$.

6.9 Experiments

We implemented infBWT as a C++ tool called G2BWT. This software uses the SDSL-lite library [76] and is available at https://bitbucket.org/DiegoDiazDominguez/lms_grammar/src/bwt_imp2. As far as we know, there is no available implementation for the eBWT, so we compared G2BWT against the tools that produce the BCR BWT, the data structure that most closely resembles the eBWT. We assesed the tools eGap (EG) [55], gsufsort-64 (GS64) [118] and BCR_LCP_GSA (BCR) [7]. EG and BCR are algorithms for constructing the BCR BWT of a string collection in external or semi-external settings, while GS64 is an in-memory algorithm for building the suffix array of an string collection, but it can also build the BCR BWT. We also considered the tool bwt-lcp-em [20] for the experiments. Still, by default it builds both the BCR BWT and the LCP array, and there is no option to turn off the LCP array, so we decided not to use it. For BCR, we used the implementation of https://github.com/giovannarosone/BCR_LCP_GSA. All the tools were compiled according their authors' description. For G2BWT, we used the compiler flags -O3 -msse4.2 -funroll-loops.

We used the same five read collections described in Section 5.6 for building the BWTs. All the reads were 152 characters long and had an alphabet of six symbols (a,c,g,t,n,$). The input datasets are described in Table 6.1.

During the experiments, we limited the RAM usage of EG to at most three times the input size. For BCR, we turned off the construction of the data structures other than the BCR BWT, and left the memory parameters by default[1]. In the case of GS64, we used the parameter -bwt to indicate that only the BCR BWT had to be built. The other options were left by default. For G2BWT, we first grammar-compressed the datasets using LPG (Chapter 5) and then used the resulting files as input for building the eBWTs. In addition, we let G2BWT to use up to 18 threads. The other tools ran in serial as none of them supported multi-threading.

There is an extra decompression cost when working with BCR, EG, and GS64 that G2BWT does not have. Sequencing platforms or public repositories of DNA sequences usually deliver read collections in compressed form[2], but BCR, EG, and GS64 manipulate the input in plain form. This difference in the input format means that we have to decompress the reads first to use these tools. In contrast, G2BWT processes the input in compressed form as it works on top of the grammar computed with LPG.

We simulated this extra cost by compressing our input datasets using p7-zip and then measuring the decompression time. We assessed the performance of G2BWT first without adding that cost to BCR, EG, and GS64 and then adding it. All the experiments were carried out on a machine with Debian 4.9, 736 GB of RAM, and processor Intel(R) Xeon(R) Silver @ 2.10GHz, with 32 cores.

6.10 Results and Discussion

The results of our experiments without considering the decompression costs for BCR, GS64 and EG are summarized in Figure 6.7. The fastest method for building the eBWT was GS64, with a mean time of 0.91 μsecs per input symbol. It is then followed by BCR, G2BWT and EG, with mean times of 0.94, 1.32 and 2.61 μsecs per input symbol, respectively (Figure 6.7A). Regarding the working space, the most efficient method was BCR, with an average space of 0.17 bytes of RAM per input symbol. G2BWT is the second most efficient, with an average of 0.78 bytes. EG and GS64 are much more expensive, using 3.07 and 10.54 bytes on average, respectively (Figure 6.7B). Adding the decompression overhead to BCR, GS64 and EG increases the running times by 0.02 μsecs per input symbols in all the cases. This negligible penalty owes to the fact that p7-zip is fast at decompressing the text.

Although G2BWT is not the most efficient method on average, it is the only one that becomes more efficient as the size of the collection increases. While the space and time functions of BCR, EG and GS64 seem to be linear with respect the input size, and with a non-negative slope in most of the cases, in G2BWT these functions resemble a decreasing logarithmic function. This behavior is due to the fact that G2BWT processes several occurrences of a given phrase as a single unit, unlike the other methods. Thus, the cost of building the eBWT depends of $\mathcal{S}$ more on the number of distinct patterns in the input rather than on its size. As genomes are repetitive, appending new read collections to a dataset increases its size considerably, but

[1] We asked the authors how to select the best parameters for I/O buffers. They advised us not to increase the buffers too much and find a value that fitted our architecture. We decided to leave it as the default.

[2] gzip is the preferred format.

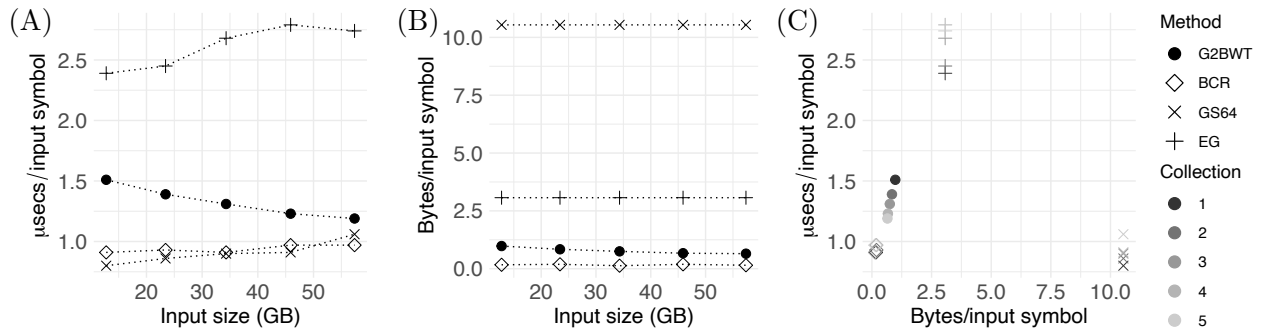

Figure 6.7: Performance of the tools for building the eBWTs. These results do not include the decompression overhead for `BCR`, `GS64`, and `EG`.

not its number of distinct patterns. As a consequence, the per-symbol processing efficiency increases.

The performance improvement of `G2BWT` is also observed when we asses the trade-off between time and space (Figure 6.7C). Although `BCR` is the one with the best trade-off, the instances of `G2BWT` move toward the bottom-left corner of the graph as we concatenate more read collections. In other words, the more massive and repetitive the input is, the less is the time and space we spend on average per input symbol to build the eBWT. This is an important remark, considering that the input collections are not as repetitive as other types of texts. In most of the datasets, the number of eBWT runs is relatively high (see Table 6.1).

Chapter 7

An Index for Navigating the Layout of Reads

The algorithmic ideas we have developed so far (Chapters 5 and 6) facilitate manipulating raw sequencing data in compressed space. In particular, we can compute the eBWT of a collection of reads directly from its grammar representation. However, this functionality *per se* is still not enough for practical biological analyses.

In this chapter, we describe a new framework based on Wheeler graphs (Section 3.2.1) that enables the extraction of biological information in succinct space. We first formalize the idea of the *layout query*, and explain how this concept can be used to perform genomic analyses. Then, we show how to answer the layout query within the context of variable-order de Bruijn graphs (vo-dBG) (Section 4.3.1). Using these ideas, we derive a new data structure that we call the *overlap tree*, which allows us to answer the layout query in a much more efficient way.

We augment BOSS (Section 4.3.1), the succinct representation for dBGs, with the overlap tree and implement a genome assembler on top it. We call the resulting data structure rBOSS. Our rBOSS index increases the space by $4n + o(n)$ bits, where n is the number of rows in the BOSS matrix. In exchange, we can compute the layout query in $\mathcal{O}((k+o)\log\sigma)$ time, where k is the dBG order and o the number of overlapping strings in the input. In contrast, VO-BOSS (Section 4.3.1), the succinct encoding of a vo-dBG, uses $n\log k(1+o(1))$ extra bits and it is much more slower at answering the layout query.

Our experimental results showed that, by using $k = 100$, the assembler implemented on top of rBOSS produces contigs of mean sizes over 10,000, which is twice the size obtained by using a pure dBG of fixed length k. Additionally, rBOSS was 20% smaller than VO-BOSS in our datasets.

The use of the overlap tree is not limited to dBGs only. We describe at the end of the chapter how to adapt it for other BWT-based representations, such as the BCR BWT or the eBWT. This last subject is of interest considering the algorithmic ideas we developed in the previous chapters of this thesis.

A preliminary version of this work [51] was presented at the *30th Annual Symposium on Combinatorial Pattern Matching* (CPM 2019).

7.1 Definitions

Let $\mathcal{S} = \{S_1, \ldots, S_q\}$ be a collection of q sequencing reads with average length z. Further, let $\mathcal{S}^* = \{S_1, S_1^{rc}, \ldots, S_q, S_q^{rc}\}$ be a collection of size $2q$ that contains the strings in $\mathcal{S}$ along with their reverse complements. Let us denote G the dBG of order k obtained from $\mathcal{S}^*$ and G' the variable-order dBG with maximum order k obtained from $\mathcal{S}^*$. A *walk* $P = v_1, v_2, \ldots, v_t$ over G, or over G', is a sequence of t nodes where every v_i, with $i \in [1, t-1]$, is connected with v_{i+1} by an edge. P will be a *path* if all the nodes are different, except possibly the first and the last ones. If $v_0 = v_t$, P is said to be a *cycle*. P is *unary* if the nodes $(v_0, \ldots, v_{t-1})$ have out-degree one and the nodes $(v_1, \ldots, v_t)$ have in-degree one. P will be a *right walk* if we follow the edge from v_i to v_{i+1}, and a *left walk* if we follow the edge from v_{i+1} to v_i (i.e., in reverse direction to the edge). The string formed by the concatenation of the edge symbols of P is referred to as its label.

We assume G and G' are encoded using the BOSS and VO-BOSS representations (respectively). Let M be the matrix (implicitly) encoding the nodes with order k in BOSS and VO-BOSS. We denote the range of rows in M suffixed by the label of v as $\overset{\bullet}{v}$. Additionally, we make use of the BOSS functions $\mathsf{outneighbor}$, $\mathsf{nodelabel}$, $\mathsf{label2node}$, and the VO-BOSS function $\mathsf{shorter}$ (see Section 4.3.1).

Rows of M representing substrings of size $k-1$ in $\mathcal{S}^*$ are called *solid* nodes and rows representing artificial $(k-1)$-length strings padded with dummy symbols from the left, and that represent prefixes in $\mathcal{S}^*$, are called *linker* nodes. For a linker node v, the function $\mathsf{llabel}(v)$ returns the non-\$ suffix of its label. A solid node appearing as a suffix in $\mathcal{S}^*$ is called an *s-node* and a solid node appearing as a prefix in $\mathcal{S}^*$ is called a *p-node*. A linker node v is said to be *contained* within another node v' (solid or linker) if $\mathsf{llabel}(v)$ is a suffix in the label of v'.

An overlap of size o between two solid nodes v and u, denoted $v \oplus^o u$, occurs when the o-length suffix of v is equal to the o-length prefix of u. Relative to v, $v \oplus^o u$ is a *forward* overlap and $u \oplus^o v$ is a *backward* overlap. We consider a minimum threshold $m < k-1$ so that the overlap $v \oplus^o u$ is *valid* if (i) $m \leq o < k-2$ and u is a p-node, or (ii) $o = k-2$ and u is a solid node of any kind.

The consensus string formed by the union of the solid nodes v and u is denoted $\mathsf{label}(v \oplus^o u)$. We lift the operator $v_1 \oplus^{o_1} v_2 \ldots v_{p-1} \oplus^{o_{p-1}} v_p$ to denote a sequence Q of p solid nodes in which every v_i, with $i \in [1, p-1]$, has a valid forward overlap with the nodes $v_{i+1}, \ldots, v_p$. The function $\mathsf{label}(Q)$ returns the consensus string formed by the union of the nodes in Q. We say that Q is *right-maximal* for v_1 if there is no solid node v_{p+1} such that $Q \oplus^{o_p} v_{p+1}$ is a valid sequence of forward overlaps. Equivalently, an overlap sequence $Q = v_p \oplus^{o_p} v_{p-1} \ldots v_2 \oplus^{o_2} v_1$ is *left-maximal* for v_1 if there is no other solid node v_{p+1} such that $v_{p+1} \oplus^{o_{p+1}} Q$ is a valid sequence of forward overlaps. When Q is not right-maximal for v_1 and $\mathsf{label}(Q)$ is a prefix in more than one different right-maximal overlap sequence for v, then Q is *ambiguous*. We also extend this nomenclature for reads in $\mathcal{S}^*$, not just dBG nodes.

7.2 The Layout Query

We define the layout query as a function that receives as input a string $\mathcal{S}_i \in \mathcal{S}^*$ and returns a list of tuples $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_e\}$, where each $Q_j \in \mathcal{Q}$ is a set of reads that form a right-maximal sequence of overlaps for S_i. Depending on the output of the layout query, we can recognize different biological signals. We describe some of them and their causes:

1. $\mathcal{Q}$ contains only one tuple Q. If the read coverage is enough, then it is highly likely that the string $\mathsf{label}(Q)$ exists as a substring in only one place of the source DNA.
2. $\mathcal{Q} = \{Q_1, Q_2\}$ has two elements and $A = \mathsf{label}(Q_1)$ and $B = \mathsf{label}(Q_2)$ are almost identical strings. In this case, it is possible that A and B match the same segment of the source DNA, and that the matched segment contains variable nucleotides (i.e., a SNP). The chances for A and B to represent this kind of signal increases if Q_1 and Q_2 have similar lengths and the source DNA was extracted from a diploid organism (Section 4.2).
3. $\mathcal{Q}$ has two elements and their labels are almost identical (as in case 2). However, one of the tuples is smaller, and it has only one element. This situation might indicate that the label of the smaller tuple contains a sequencing error.
4. $\mathcal{Q}$ has several tuples, and all with similar lengths. This signal might indicate that S_i is a string that appears several times in the source DNA, with different right contexts.

Using the signals described above we can perform most genomic analyses that rely on sequencing data. For instance, if we are assembling a genome, we can append new reads into a contig as long as the layout query returns tuples of type 1, 2 and 3 for them. If, on the other hand, we are interested in SNP calling, we can focus on reads with tuples of type 2. Still, it might happen that a sequence $Q' = S_1 \oplus^1 v_2 \ldots S_{e-1} \oplus^{S_{e-1}} S_e$ is a prefix in more than one element of $\mathcal{Q}$. In such case, it is better not to use Q' to make any biological inference as it is ambiguous. Considering this detail, we define a confidence measure for each $Q_j \in \mathcal{Q}$:

$$\mathsf{weight}(Q_j) = \frac{|Q_j| - |Q'_j|}{x},$$

where Q'_j is the longest ambiguous prefix that Q_j shares with another element of $\mathcal{Q}$ and x is the number of reads that overlap S_i and that were uniquely assigned to one sequence of $\mathcal{Q}$. The reads in Q_j and Q'_j are a subset of all the reads overlapping S_i, so $\mathsf{weight}(Q_j)$ is always in the range $[0, 1]$. With this formula, we formally define the function $\mathsf{layout}(S_i)$ as follows:

- $\mathsf{layout}(S_i)$: returns a tuple (S_j, O_j, w_j) for every element of $\mathcal{Q}_j \in \mathcal{Q}$, where S_j is the rightmost read of Q_j, O_j is the suffix of $\mathsf{label}(Q_j)$ that does not overlap S_i, and $w_j = \mathsf{weight}(Q_j)$.

Figure 7.1 depicts graphical examples of these ideas.

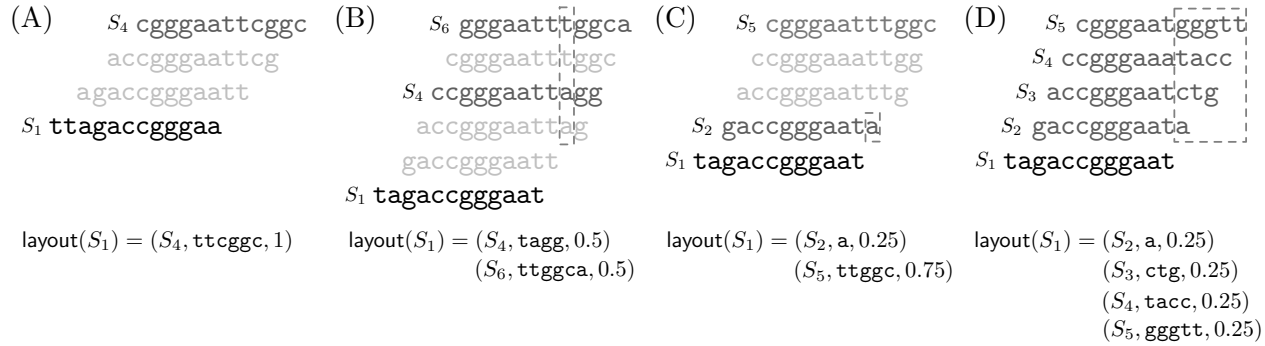

Figure 7.1: Graphical examples of the signals described in Section 7.2. In (B), the read gaccgggaatt (above S_1) is not assigned to any overlap sequence because it is ambiguous.

To our knowledge, there is no data structure that supports the layout query. The classical dBG and string graph could not answer it because they lack information. In a dBG of order k, the overlaps of less than $k-1$ characters between the reads are not encoded. In a string graph, on the other hand, the transitive edges are removed to perform structural compression, but these edges are necessary to compute the tuples in $\mathcal{Q}$. We consider $\mathsf{layout}(S_i)$ to be a useful primitive to traverse the reads and make inferences on the fly about the underlying source DNA.

We note that the concept of $\mathsf{layout}(S_i)$ can be easily adapted to dBGs; instead of using reads as input, we use solid nodes. In later sections, we show how to augment BOSS to support the layout primitive. We believe this extra feature would make BOSS almost as powerful as the BCR BWT or the eBWT for extracting genomic information. The idea is compelling if we consider that this succinct dBG representation requires less space than the BCR or the eBWT, and it is cheaper to construct.

The first step to implement layout in BOSS is to have a primitive that computes overlaps between node labels. This function would allow us to detect overlaps of less than $k-1$ characters between reads. The only succinct dBG representation we know of that could support the layout query is VO-BOSS (Section 4.3.1), the one aimed for vo-dBGs. In the next section, we show how varying the order in VO-BOSS is related to computing overlaps of less than $k-1$ characters between reads.

7.3 Computing Overlaps in a vo-dBG

Let $\mathcal{A} = \{S_i, \ldots, S_{i+x}\}$ be a set of reads in $\mathcal{S}^*$ forming a valid sequence of forward overlaps $Q = S_i \oplus^{o_i} S_{i+1} \ldots S_{i+x-1} \oplus^{o_{i+x-1}} S_{i+x}$, where the overlaps $o_i, \ldots, o_{i+x-1}$ are of length $\geq k-1$. Also, let W_{i+j} be the walk in G spelling the read S_{i+j}, with $j \in [0, x]$. Given the definition of the dBG, we know the nodes in W_{i+j} are reachable from the nodes in any of the other walks of G spelling strings of $\mathcal{A}$. Still, these walks can be entangled with other walks in G spelling other strings of $\mathcal{S}^*$, which makes the navigation of Q difficult. We can reduce the entanglement of G by choosing a high order k, but this decision can make the dBG too disconnected to be useful. Notice, however, that the nodes in W_{i+j} remain connected, regardless the value of k. The real problem in using a high dBG order is that the overlaps

between reads of Q will not be captured by edges of G if the length of the overlaps is less than $k-1$, which becomes more likely when k is high. We can solve this problem using the VO-BOSS index for G', the vo-dBG constructed from $\mathcal{S}^*$.

Assume we select a high order k for building G'. We pick an arbitrary node y at order k representing a prefix of S_i (a p-node). We can easily check that by inspecting whether the label in one of the incoming nodes of y is padded from the left with a dummy symbol. Then, we perform a right walk over G' starting from y. If we reach a solid node v with out-degree one, and whose outgoing edge is labeled with \$, then $\mathsf{nodelabel}(v)$ only appears as a suffix in the reads of $\mathcal{S}^*$ (as s-node), and the overlaps of these reads (if any) are less than $k-1$ characters long. We need to find a p-node whose label overlaps the label of v to continue with the navigation of Q. For that purpose, we decrease the order of v using $\mathsf{shorter}$ to obtain a node u whose label is both a suffix of $\mathsf{nodelabel}(v)$ and a prefix of some read in $\mathcal{S}^*$. From u, we retrieve the overlapping p-nodes of v using the function $\mathsf{outneighbor}$.

In VO-BOSS, however, $\mathsf{shorter}$ does not ensure that the label of u appears as a prefix in $\mathcal{S}^*$. The next lemma precises the condition that must hold to ensure this.

Lemma 6 In VO-BOSS, applying the operation $\mathsf{shorter}$ to a node v of order $k' \leq k$ will return a node u of order $k'' < k'$ that encodes a forward overlap for v iff $\overrightarrow{u}[1]$ is a linker node contained by v.

Proof. Let the left contexts of u be the prefixes of length $k-k''$ in the range of rows $\overrightarrow{u}$ of M. If all the left contexts of u are non-dummy strings, then $\mathsf{nodelabel}(u)$ does not appear as a prefix in $\mathcal{S}^*$, and hence, following none of its edges will lead to a valid overlap of v. On the other hand, if $\mathsf{nodelabel}(u)$ appears as a prefix in $\mathcal{S}^*$, then there is a node v' in G' at order k whose label is formed by the concatenation of a dummy string and $\mathsf{nodelabel}(u)$, and that by definition is a linker node contained by v. As the rows in M are sorted in colexicographical order, the ties for the labels in $\overrightarrow{u}$ are broken by the left contexts of u. Therefore, v' is placed in $\overrightarrow{u}[1]$ because the dummy string is always colexicographically the smallest. □

Lemma 6 allows us to spell the sequence of a read $S_i \in \mathcal{S}^*$ provided we already spelled a read $S_j \in \mathcal{S}^*$ that overlaps one of the prefixes of S_i. Unlike the regular dBG, this property holds even if S_i and S_j overlap by less than $k-1$ characters. With Lemma 6 there is no problem in selecting a high dBG order for G' as we can still retrieve most of the overlap sequences for the reads (the Q list). Recall that a high value for k avoids the entanglement of the dBG. The only caveat is that we need to define a minimum threshold $1 < m < k-1$ for the overlaps to reduce spurious connections between strings of $\mathcal{S}^*$. We now formalize the idea of Lemma 6 in a function called $\mathsf{nextlinker}$. Let v be a vo-dBG node at order k, then the function is defined as follows:

- $\mathsf{nextlinker}(v)$: returns the greatest linker node $v' < v$ at order k whose llabel represents both a suffix of $\mathsf{label}(v)$ and a prefix of some other node in G'.

Theorem 5 There is an algorithm that solves $\mathsf{nextlinker}(v)$ in $\mathcal{O}((k-m)k \log \sigma)$ time.

Algorithm 4 Build set E with the linker nodes contained by v

```
1: proc nextlinker(v, m)          ▷ returns the greatest linker node v' contained by v with |llabel(v')| ≥ m
2:     d ← v.order − 1
3:     while d ≥ m do
4:         u ← shorter(v, d)
5:         if u• ⊃ v• then
6:             if islinker(u) and |llabel(u)| = d then
7:                 return u•[1]
8:             v ← u
9:         d ← d − 1
10:    return 0                                   ▷ dummy node

11: proc getlinkers(v, m)         ▷ v is a vo-dBG node and m is the minimum suffix size
12:    E ← ∅
13:    c ← nextlinker(v, m)
14:    while c>0 do
15:        E ← E ∪ {c}
16:        c ← nextlinker(c, m)
17:    return E
```

Proof. Incrementally decrease the order of v by one until reaching a node u with $\overset{\bullet}{u} \supset \overset{\bullet}{v}$, and that satisfies Lemma 6. If such u exists, return it. If the order of v decreases below m before finding u, then v does not contain any linker node v' with $|\mathsf{llabel}(v')| \geq m$. In such case, we return a dummy vo-dBG node. The function reduces the order at most $k-m-2$ times. In each decrement of order, we use the operations shorter and llabel to check Lemma 6, which take $\mathcal{O}(\log k)$ and $\mathcal{O}(k \log \sigma)$ time (respectively). Thus, the total time is $\mathcal{O}((k-m)k \log \sigma)$. □

Notice, however, that a vo-dBG node in G' might have more than one contained linker node v' with $|\mathsf{llabel}(v')| \geq m$. A useful operation then would be one that returns a set E with the identifiers in VO-BOSS of all those relevant linkers. We can then follow the outgoing edges of every $l \in E$ to infer the solid nodes that overlap v by at least m symbols.

- getlinkers(v, m): the set E with all the linker nodes contained by v that represent a suffix of v of length $\geq m$.

The function getlinkers applies nextlinker iteratively until it returns a node u whose llabel has length less than m. The rationale is that if v contains v', and in turn v' contains v'', then v also contains v''. If we iterate over the $k-m$ orders of the vo-dBG, then we can obtain all the linkers contained by v. Consequently, nextlinker and getlinkers have the same worst-case time complexity under the vo-dBG model. Algorithm 4 shows the details.

Now we use getlinkers to define a new primitive for VO-BOSS:

- foverlaps(v): returns the set of p-nodes whose prefixes overlap suffixes of v with length $\geq m$.

We implement this function by computing the set E = getlinkers(v) and then traversing all the forward paths that start in a node $l \in E$ and end in a p-node. We use the BOSS function forward to traverse these paths. Thus, the time complexity for foverlaps(v) is given by the next theorem.

Theorem 6 Computing foverlaps(v) takes $\mathcal{O}((k-m)((k+\sigma^{k-m})\log\sigma))$ time.

Proof. Computing getlinkers(v) takes $\mathcal{O}((k-m)k\log\sigma)$ time, and the resulting list E contains no more than $k-m$ linker nodes. If we assume that the path P_l staring in a node $l \in E$ and ending in a p-node is unary, then its traversal requires no more than $k-m$ outneighbor operations. Thus, the total time for reaching all the p-nodes from the linker nodes in E is $\mathcal{O}((k-m)^2\log\sigma)$. In reality, however, each node in P_l can have up to σ distinct outgoing edges, which rises the time complexity to $\mathcal{O}((k-m)\sigma^{k-m}\log\sigma)$. This time complexity plus the time complexity for computing E gives us the final result for this theorem. □

If we chose $k = z+1$ to build VO-BOSS index for G', then we can simulate the full overlap graph in compact space. Every vo-dBG node v at order k with an outgoing edge labeled with \$ encodes a specific read $S_i \in \mathcal{S}^*$. The arcs of the node labeled with S_i in the overlap graph are not stored explicitly, but computed on the fly from v. More specifically, we can reach from v all the vo-dBG nodes whose labels are reads in $\mathcal{S}^*$ and that overlap suffixes of S_i. For this task, we compute E = getlinkers(v) and then follow the outgoing edges of every node $l \in E$ until reaching another node that has an outgoing edge labeled with \$. Still, the complexities of the involved operations nextlinker makes VO-BOSS slow for exhaustive traversals, which is our main interest. We describe a faster alternative in the next section.

7.4 The Overlap Tree and rBOSS

We can regard the function getlinkers as a bottom-up traversal of the trie T induced by the $(k-1)$-length labels of M read in reverse. Every trie node t corresponds to a vo-dBG node v whose order is the string depth of t. The traversal starts in the trie leaf t corresponding to the vo-dBG node v given to getlinkers, and continues upward until finding the last ancestor t' of t with string depth $\geq m$. The movement from t to t' is equivalent to the successive application of nextlinker. In each such application, we move from a node t to its nearest ancestor t' that is maximal (i.e., has more than one child) and whose leftmost child edge is labeled by a \$.

Since non-maximal nodes in T are not relevant for building E, we can implement nextlinker using the topology of the *compact* trie T (i.e., collapsing unary paths) instead of using shorter. In this way, we get rid of the LCS structure of VO-BOSS (see Section 4.3.1). Moreover, we can succinctly encode the topology of the compact trie if we use one of the representations of Section 2.2.3. For this particular case, we will use BP [138].

Replacing the LCS with the topology of T poses two problems, though. First, it is not possible to define a minimum dBG order m from which overlaps are not allowed, and second, Lemma 6 cannot be checked. Both problems arise because, unlike the LCS, the BP representation does not encode the string depths of the tree nodes (and thus, the represented

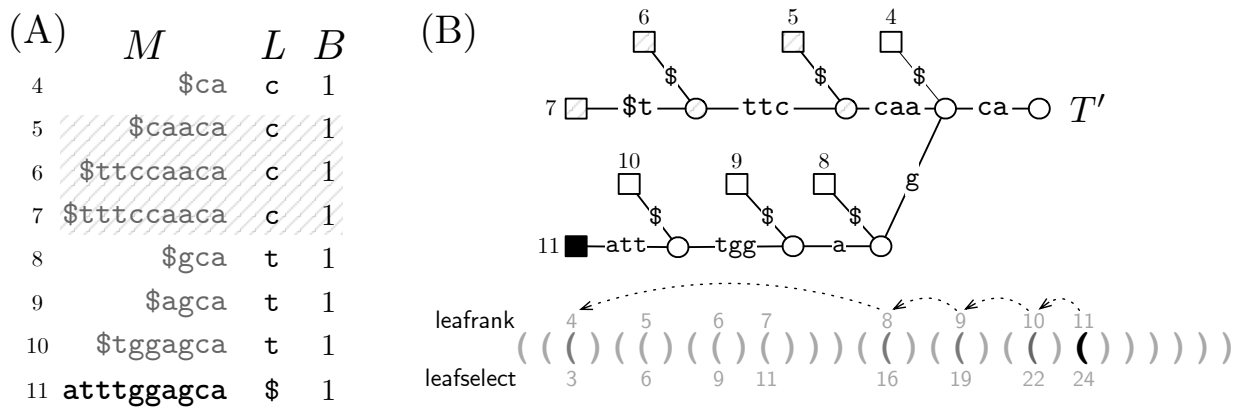

Figure 7.2: Example of rBOSS with parameters $k = 11$ and $m = 2$. (A) The range of rows in M for solid node 11 (`atttggagta`) and the linker nodes contained by it (rows 4 to 10). Columns L and B are the elements of the succinct dBG encoding. The dashed rows are linker nodes, but they are not contained by node 11. For simplicity, we did not include the padding of `$` characters in M. (B) Subtree of T' mapping the range $[4, 11]$ in M. The squares are leaves and the circles are internal nodes. The leaf in black maps node 11 in M. The numbers next to the leaves are their leaf ranks in pre-order. The dashed leaves map the rows in M that are not contained by 11. The array of parentheses below represents the same subtree in BP format. The rank of every leaf (leafrank) is shown above its relative position in the BP array (leafselect). Each dashed arrow represents a call of the function nextlinker and all the arrows together stand for the function getlinkers.

node lengths). Still, we can reduce the topology of T to precisely the nodes of interest for nextlinker, and thus avoid any check. We call this structurally-compressed version of T the *overlap tree*.

Theorem 7 There is a structure using $4 + o(1)$ bits per dBG node that implements the function nextlinker in $\mathcal{O}(1)$ time and the function getlinkers in $\mathcal{O}(k - m)$ time.

Proof. The structure is the BP encoding of a tree T' that we obtain by removing some nodes from the compact trie T (T has one leaf per dBG node, and less than 2 nodes per dBG node because it is compact). First, we discard all the internal nodes of T with string depth below m, and connect the subtrees left to the root of T'. Second, we discard every internal node $t' \in T$ whose leftmost-child edge is not labeled by a dummy string, and recursively connect its children to the parent of t'. Note that all the leaves of T are in T'.

T' has precisely the nodes of interest to implement nextlinker(v). We simply find the ith left-to-right leaf t in T', where i is the row of M corresponding to v (note that rows of M and leaves of T and T' are in the same order). Then, we move to the parent t' of t and return its leftmost child. An exception occurs if the leftmost child of t' is precisely t, which means that v is a linker node and thus its next contained node is the leftmost child of the parent of t'. Finally, we return the rank in of the desired leftmost leaf. Algorithm 5 shows the details.

All the operations are implemented in $\mathcal{O}(1)$ time in BP, and thus nextlinker(v) is also

Algorithm 5 Function nextlinker implemented with the topology of T'

```
1: proc nextlinker(v)                                  ▷ v is a vo-dBG node at order k
2:     t ← leafselect(T', v)                           ▷ node in T' mapping v
3:     t' ← parent(T', t)
4:     if fchild(T', t') = t then                      ▷ t is already the leftmost child of its parent
5:         t' ← parent(T', t')
6:     l ← fchild(T', t')
7:     return leafrank(T', l)                          ▷ vo-dBG node mapping l
```

implemented in $\mathcal{O}(1)$ time. The function getlinkers stays the same, and its cost is dominated by the (at most) $k - m$ calls to nextlinker. □

In what follows we devise a more efficient approach for computing foverlaps(v) that uses the topology of the overlap tree and the reverse complements of the node labels. We need to define first the idea of bidirectionality in rBOSS.

7.5 Simulating Bidirectionality

We build rBOSS on $\mathcal{S}^*$ because the reads in $\mathcal{S}$ are produced from any of the strands of the source DNA. Consequently, there are several combinations in which two strings, $S_i, S_j \in \mathcal{S}$ can have a valid suffix-prefix overlap: (S_i, S_j), (S_i, S_j^{rc}), (S_i^{rc}, S_j), or (S_i^{rc}, S_j^{rc}), and all must be encoded in T'. An interesting consequence of including the reverse complements is that the topology of the dBG becomes symmetric.

Lemma 7 The incoming symbols of a dBG node v are the DNA complements of the outgoing symbols of v^{rc}, the node labeled with the reverse complement of nodelabel(v). Equivalently, the outgoing nodes of v^{rc} are the same as the DNA complements of the incoming nodes of v.

Proof. Consider the $(k-1)$-length substring bXc of $\mathcal{S}$, and a symbol a that appears to the left of some occurrences of bXc. We consider both $abXc$ and its reverse complement $(abXc)^{rc} = c^c X^{rc} b^c a^c$ to build rBOSS. This decision produces the dBG node v labeled bXc to have an incoming symbol a, and the dBG node v^{rc} labeled $(bXc)^{rc} = c^c X^{rc} b^c$ to have an outgoing symbol a^c. Thus, the label of node outneighbor(v^{rc}, a^c) will be $X^{rc} b^c a^c$, which is the reverse complement of string abX, the label of node inneighbor(v, a). □

As a result of including the reverse complements of the reads, the cost of computing the incoming symbols of node v becomes proportional to the cost of computing the position of v^{rc} in the BOSS matrix. We refer to this latter operation as rcnode, and formally define it as follows:

- rcnode(v): node v^{rc} such that nodelabel(v^{rc}) = nodelabel(v)^{rc}

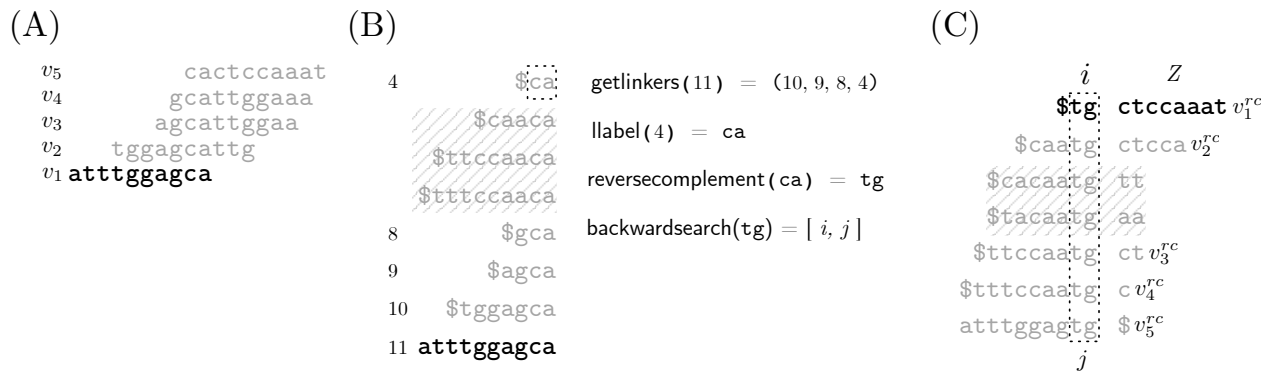

Figure 7.3: Implementing foverlap using the bidirectionality of rBOSS. (A) A solid node v_1, with label atttggacga, along with the other nodes that overlap its suffixes (gray sequences). (B) The range of rows in M for v_1 = 11 and the linker nodes contained by it. The dashed box indicates llabel(4) = ca. (C) Range $[i, j]$ of nodes in M suffixed with X = llabel(4)^{rc} = tg. The strings to the right of M are path labels that match prefixes of Z = ctccaaat.

Theorem 8 Computing rcnode(v) takes $\mathcal{O}(k \log \sigma)$ time. By augmenting rBOSS with $s \log n$ extra bits, s being the number of solid nodes, the time decreases to $\mathcal{O}(1)$.

Proof. We compute first the string X = nodelabel(v)^{rc} in $\mathcal{O}(k \log \sigma)$ time using the BOSS primitives, and then we perform label2node(X) to find v^{rc}. Therefore, rcnode(v) takes $\mathcal{O}(k \log \sigma)$ time. Alternatively, we can store an explicit permutation on the s solid nodes in M, so that using $s \log n$ bits we find the position of v^{rc} in M in $\mathcal{O}(1)$ time. □

Theorem 8 allows us to compute the forward overlaps of v in time proportional to the size of the label of v.

Theorem 9 Using the bidirectionality of rBOSS we can implement foverlaps(v) in $\mathcal{O}((k + o) \log \sigma)$ time, where o is the number of nodes that overlap suffixes of v by at least m characters.

Proof. We first obtain E = getlinkers(v) to get the linker node $l = E[|E|]$ representing the smallest suffix of v. Then, we compute X = llabel(l)^{rc} and search for the range $[i, j]$ of rows in M where the labels are suffixed by X. From the nodes in $[i, j]$, we follow the dBG paths spelling prefixes of Z = nodelabel(v)$^{rc}[|X| + 1..]$. However, instead of traversing these paths independently, we apply the function outneighbor in batch. The idea is as follows; let p be the current step in the path traversals and let $a = Z[p]$ be the symbol that the paths should match at step p. During step p, we obtain the nodes in $[i, j]$ with an outgoing edge labeled with \$, and report their reverse complement nodes as overlaps for v. Then, we update $[i, j]$

for the next traversal step $p + 1$ as:

$$
\begin{aligned}
i &= \mathsf{select}_1(B, i - 1) + 1) \\
j &= \mathsf{select}_1(B, j)) \\
i &= \mathsf{rank}_1(I, C[a] + \mathsf{rank}_a(L, i)) \\
j &= \mathsf{rank}_1(I, C[a] + \mathsf{rank}_a(L, j)),
\end{aligned}
$$

where C and I are elements of BOSS. C is the array with the symbol frequencies and I is the bit vector encoding the nodes' in-degree. We continue doing this process until the symbols of Z are consumed completely or $[i, j]$ becomes empty. Computing the first range $[i, j]$ takes $\mathcal{O}((k - m) + k \log \sigma)$ time. Then, every batched outneighbor step takes $\mathcal{O}(\log \sigma)$ time, and there are no more than $k - m$ of them as this is the maximum length for Z. Additionally, extracting the o distinct \$ symbols takes $\mathcal{O}(o \log \sigma)$ time, and computing their reverse complement nodes takes $\mathcal{O}(1)$ if we use the permutations. Thus, foverlaps can be performed in $\mathcal{O}((k + o) \log \sigma)$ time. □

Figure 7.3 exemplifies the overlap function. Note we can obtain the backward overlaps for v if we use foverlaps on the reverse complement node for v. We formally define the symmetric function boverlaps:

- boverlaps(v): the set of s-nodes whose suffixes overlap prefixes of v with length $\geq m$.

7.6 Implementing the Layout Query

In this section, we explain how to implement layout on top of rBOSS. First, we slightly modify the definitions of Section 7.2 to adapt the function for dBGs. The input will be a solid node v in BOSS representation for G instead of a read in $\mathcal{S}^*$. Additionally, the list $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_e\}$ will contain the e distinct right-maximal overlap sequences for v. In this case, however, the strings of every $Q_j = v \oplus^{o_v} v_2 \ldots v_{u-1} \oplus^{o_{u-1}} v_u \in \mathcal{Q}$ are not reads but solid nodes.

Implementing layout(v) requires us to modify the regular foverlaps function as follows: once we obtain the range $[i, j]$ of M's rows for X (see Section 7.5), we initialize three vectors, Y, O and P. $Y = i, \ldots, j$ will initially store the nodes in $[i, j]$, and O and P will be empty. In every batched outneighbor step p, we remove $Y[u - i + 1]$ if node u, with $u \in [i, j]$, does not have an outgoing edge labeled with $a = Z[p]$, and maintain the relative order of the other nodes that remain in Y. Additionally, when a node u in $[i, j]$ has an outgoing edge labeled with \$, we move $Y[u - i + 1]$ to O and push the pair $(u^{rc}, |X| + p)$ to P. After finishing the batched outneighbor steps, O will contain the nodes in the first range $[i, j]$ from which we can start a path traversal labeled with a prefix of Z. On the other hand, P will contain the information of the solid nodes that overlap label(v).

The next step is to get the different right-maximal overlap sequences of v using O, P, and T'. For simplicity, we assume O was left sorted in increasing order after the batched outneighbor steps. We first initialize an empty list X and two temporal variables $c = |O|$ and $l = 1$. Also, we obtain the leaf y in T' that maps the dBG node $O[c]$. Then, we begin a bottom-up traversal in T' from y. In each step, we go to the leftmost sibling y' of y. When

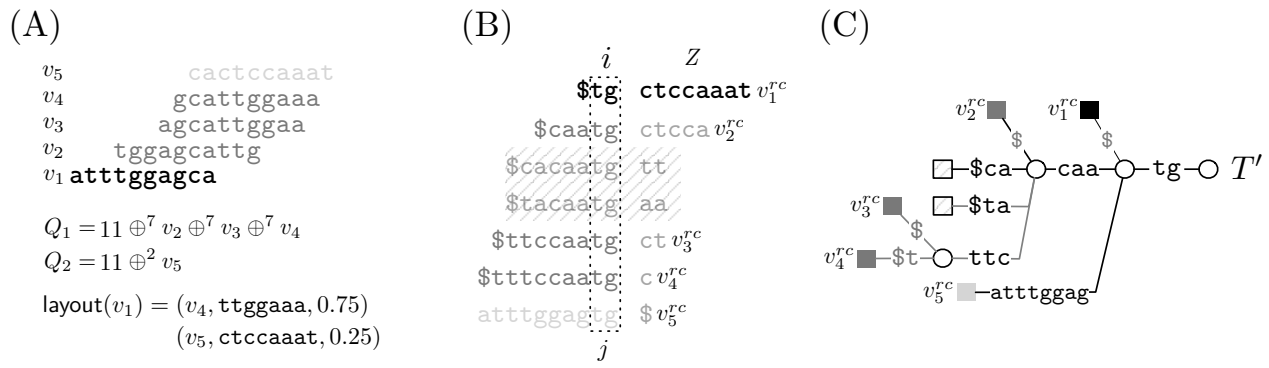

Figure 7.4: Implementing layout in rBOSS. (A) The same solid node v_1 of Figure 7.3 along with its overlapping nodes. The node v_5 (light gray) belongs to the right-maximal overlap sequence Q_2, while nodes v_2,v_3 and v_4 (dark gray) belong to the right-maximal sequence Q_1. The output of layout(v_1) is depicted below. (B) Range $[i, j]$ of labels in M suffixed with tg = ca^{rc} and the subtree of T' that maps the range $[i, j]$ of M. The leaf colors (except the black one) denote the right-maximal sequences to which the corresponding dBG nodes belong.

y is already the leftmost child of its parent, y' becomes the leftmost sibling of the parent of y. If y' maps the dBG node $O[c-1]$, then we increment l by one, decrement c by one, and set $y = y'$. When y' does not map $O[c-1]$, we distinguish two cases:

- The leaf y' maps some node $O[c']$, with $c' < c-1$. This situation means that the solid node in $P[c']$ can be assigned to more than one sequence of $\mathcal{Q}$. We mark $O[c']$ to skip it later.
- The leaf y' maps a node u in $[i, j]$, but u is not in O. No path starting at u and finishing in a solid node matches a prefix of Z. In this case, we do not mark any node.

When one of these two situations occur, we insert the pair (x, s, l) to X, where $P[c+l-1] = (x, s)$. We also set $l = 1$ and decrement c until finding the next unmarked node in O. The tuple in X is a succinct encoding for an overlap sequence $Q_j \in \mathcal{Q}$, where x is the rightmost node in Q_j, s is the length of the prefix in nodelabel(x) that overlaps nodelabel(v) and l is the length of Q_j. After updating c and l, we update y to the leave in T' that maps the dBG node $O[c]$ and start a new bottom-up traversal of T' from y. We stop when O is completely consumed.

In the last step of layout, we obtain from X the tuple (v_j, O_j, w_j) of every overlap sequence $\mathcal{Q}_j \in \mathcal{Q}$. Let (x, s, l) be the tuple in $X[j]$, then we set $v_j = x$, O_j = label(x)$[s+1..]$, and $w_j = l/l'$, where l' is the sum of all the distinct l values in X. Figure 7.4 shows an example of the process.

Theorem 10 The function layout(v) can be implemented in $\mathcal{O}((k + r + e(k-m)) \log \sigma + o)$ time in rBOSS, where r is the length of the first range $[i, j]$, o is the number solid nodes that overlap suffixes of v and e is the size of $\mathcal{Q}$.

Function	Description	Time Complexity
$\mathsf{nextlinker}(v)$	greatest linker node contained by v	$\mathcal{O}(1)$
$\mathsf{getlinkers}(v)$	all linker nodes contained by v	$\mathcal{O}(k)$
$\mathsf{rcnode}(v)$	node v^{rc} with $\mathsf{nodelabel}(v^{rc}) = \mathsf{nodelabel}(v)^{rc}$	$\mathcal{O}(1)$ / $\mathcal{O}(k \log \sigma)$
$\mathsf{foverlaps}(v)$	solid nodes overlapping suffixes of v	$\mathcal{O}((k+o)\log\sigma)$
$\mathsf{boverlaps}(v)$	solid nodes overlapping prefixes of v	$\mathcal{O}((k+o)\log\sigma)$
$\mathsf{layout}(v)$	right-maximal overlaps sequences for v	$\mathcal{O}((k+r+e(k-m))\log\sigma+o)$

Table 7.1: Primitives for rBOSS. The time complexity of rcnode vary depending on if we add the permutation.

Proof. Computing the first range $[i,j]$ and then the batched outneighbor steps takes us $\mathcal{O}((k-m)+k\log\sigma)$ time as before. However, this time we have to check which positions in the range $[i,j]$ of every outneighbor step p do not match $a = Z[p]$. For doing so, we consider the array L of BOSS (Section 4.3.1) to be run-length compressed. In every step p, we use the select_a operation on L to visit the equal-symbol runs labeled with a within $[i,j]$. The symbols between these runs are mismatches for a, so we use their positions to build the sets P and O. Note that the number of equal-symbol runs of a we visit in every distinct range $[i,j]$ depends on the number of mismatches. In turns, the sum of mismatches in all the outneighbor steps is no more than r, the length of the first range $[i,j]$. Therefore, we do not incur in more than r select_a operations, which take $\mathcal{O}(r\log\sigma)$ time. Then, building the set X takes $\mathcal{O}(o)$ time as the only leaves of T' we visit are in O. Building the e distinct tuples of layout from X takes us $\mathcal{O}(e(k-m)\log\sigma)$ time. This last complexity is due to the extraction of string O_j. Thus, the final time complexity for layout is $\mathcal{O}((k+r+e(k-m))\log\sigma+o)$. □

Table 7.6 summarizes all the new primitives that rBOSS supports.

7.7 The Layout Query and the BWT of the Reads

We can implement the layout function described in Section 7.2 using the eBWT of $\mathcal{S}^*$ (Section 2.2.2). Before explaining the idea, let us first define the function $\mathsf{spellsuffix}(i)$, which is equivalent to nodelabel in BOSS. Let L be the eBWT for $\mathcal{S}^*$. Given the position $L[j]$, $\mathsf{spellsuffix}(j)$ returns the sequence of the jth suffix of $\mathcal{S}^*$ in $<_\omega$ order (see Section 3.2.1). We implement spellsuffix by performing LF^{-1} from $L[j]$ until reaching the range $L[1,2q]$ with the preceding symbols of the \$ characters in $\mathcal{S}^*$. For simplicity, we identify every read S_i with its lexicographical rank i in $\mathcal{S}^*$. In this way, we can obtain its sequence as $\mathsf{spellsuffix}(\mathsf{select}_{\$}(L,i))$.

The function $\mathsf{layout}(i)$ receives as input the rank i of S_i, and returns a list $\mathcal{Q}$ with all the left-maximal overlap sequences for S_i. Every list $Q_j \in \mathcal{Q}$ is encoded as a tuple (S_j, O_j, w_j), where S_j is the identifier of leftmost read in Q_j, O_j is the prefix of S_j that does not overlap S_i and w_j is the weight of Q_j as described in Section 7.2.

We compute the left-maximal overlaps because the ordering in the BWT is inverse to

that in BOSS. In the former representation, the suffixes of $\mathcal{S}^*$ are sorted in $>_\omega$ and L stores their proceeding symbols, while in the latter, the substrings of length $k-1$ are sorted in colexicographical order, and L stores the symbols that follows them in $\mathcal{S}^*$. For the same reason, we build the overlap tree T' for the eBWT from the topology of the generalized suffix tree of $\mathcal{S}^*$. We also need to consider a bitmap $B[1, |L|]$ marking equal suffixes of $\mathcal{S}^*$. In other words, if $B[i]$ and $B[j]$ are set to 1, and no other position in between is set to 1, then spellsuffix(u), with $u \in [i+1, j]$, returns the same string. Note that the number of 1s in B is the same as the number of leaves in T'.

The algorithm for layout remains almost the same. The only difference is that we replace outneighbor with the LF function, nodelabel becomes spellsuffix, and that k becomes z in the time complexity, where z is the average length of the reads in $\mathcal{S}^*$. However, unlike rBOSS, we can enrich the result as described in Section 6.1. Recall that variation of the eBWT uses the string circularity to encode the reverse complements of the reads, or to encode their pairs when $\mathcal{S}^*$ is a paired-end library (Section 4.2). We use the information encoded with the circularity to remove erroneous overlaps from the result of layout. For instance, when $S_i \in \mathcal{S}^*$ overlaps its reverse complement or its mate in a paired-end library.

7.8 Genome Assembly

In this section, we explain how to use the rBOSS framework to spell contigs of the source DNA of $\mathcal{S}^*$ (Section 4.3). Our method considers an extended version of G that contains extra labeled edges. The idea is as follows; let v and u be two solid nodes in G, where v is an s-node with out-degree one and u is a p-node. An edge (v, u) labeled with O_u exists if layout$(v) = \mathcal{Q}$ contains the tuple (u, O_u, w_u). The definition of (v, u) differs from that of a regular dBG edge as the label O_u is a string, not a character. However, for our purposes, this difference helps us to extend the contig lengths. We do not store (v, u) explicitly as we can compute it on the fly using the rBOSS primitives as we traverse G.

Similar to most genome assemblers, we perform walks over G and store their labels as contigs. Recall from Section 4.3 that the walks over unary paths are the most likely to spell real segments of the source DNA. Still, unary paths represent only a small fraction of the underlying genome. An alternative to unary paths would be to search for omnitigs in G, but these types of walks produce safe strings only if the underlying genome is circular, there are no gaps in the sequencing coverage, and there are no sequencing errors. These model constraints make the idea of omnitigs difficult to implement.

We solve the limitations of unary paths and omitigs by spelling contigs from walks without loops (i.e., paths) that can be deterministically extended to the left or right. We formally describe these walks as follows.

Definition 8 A path $P = v_1, v_2, \ldots, v_p$ is right-maximal if all its nodes have out-degree one, except v_p. Equivalently, P is left-maximal if all its nodes have in-degree one, except v_1.

Definition 9 A path $P = v_{i-p'} \ldots v_i \ldots v_{i+p}$ is maximal if there is a node v_i such that the

prefix $P_l = v_{i-p'} \dots v_i$ is left-maximal and the suffix $P_r = v_i \dots v_{i+p}$ is right-maximal.

Maximal paths have fewer constraints than unary paths, so we expect them to produce longer contigs. On the other hand, although maximal paths could spell shorter contigs than omnitigs, they do not have the model constraints that omnitigs have.

Another advantage of maximal paths is that they are safe, i.e., any genomic reconstruction we produce from G will have the label of a maximal path P as substring (see Section 4.3). This property makes the label of P likely to exists in the source DNA of $\mathcal{S}^*$. We say "likely" because sequencing errors, sporadic overlaps, lack of sequencing coverage, among other things, introduce spurious edges (or remove real edges) in G, making it difficult to differentiate between walks spelling real contigs from those that do not. Safe paths are the best tool we have under the dBG model.

We demonstrate that P is safe with the following lemma.

Lemma 8 Let $P = v_{i-p'} \dots v_i \dots v_{i+p}$ be a maximal path, where the prefix $P_l = v_{i-p'}, \dots, v_i$ is left-maximal and the suffix $P_r = v_i, \dots, v_{i+p}$ is right-maximal. Then the label of P is safe.

Proof. Let us first demonstrate that P_l is safe. Let us denote $\mathcal{W}_l$ the set of all possible left walks over G that cross v_i and $v_{i-p'}$. As all the nodes in P_l have in-degree one, except $v_{i-p'}$, all the walks in $\mathcal{W}_l$ must visit the nodes of P_l in an orderly manner from right to left. This property implies that the label of P_l appears as a substring in all the labels spelled from $\mathcal{W}_l$, and by definition, that means that P_l is safe. Extending P_l with one of the incoming nodes of $v_{i-p'}$ is not safe. Node $v_{i-p'}$ has in-degree > 1, and the nodes in P_l are allowed to have out-degree > 1. Hence, several left walks can converge in $v_{i-p'}$. We do not have enough information to select an incoming node of $v_{i-p'}$, say x, so that the label of the new left walk (x, P_l) exists in the source DNA. The whole argument that proves that P_l is safe applies symmetrically to P_r. Let $\mathcal{W}_r$ be the set with all right walks over G that cross v_i and v_{i+p}. The elements in $\mathcal{W}_r$ must visit the nodes of P_r as all of them have out-degree one, except v_{i+p}. Consequently, the label of P_r appears as a substring in all the labels spelled from $\mathcal{W}_r$, meaning that P_r is safe. Further extending P_r to the right is not safe as v_{i+p} has out-degree > 1, and the nodes in P_r are allowed to have in-degree > 1, which produces several right walks to converge in v_{i+p}. As P_l and P_r are safe and share the same node v_i, then P is also safe. Figure 7.5 depicts examples of these ideas. □

We now explain how to produce a contig from a maximal path from G using rBOSS. We pick an arbitrary node v from G and initialize a new contig $C_f = \mathsf{nodelabel}(v)$. We start a right walk over G from v and continue as long as the nodes we visit have out-degree one. If we reach an s-node x with out-degree one at some point of the right walk, and the label of its outgoing edge is \$, we call the function $\mathcal{Q} = \mathsf{layout}(x)$ to compute its extended outgoing edges. If $\mathcal{Q}$ has only one tuple (y, O_y, w_y), we append the sequence O_y to C_f and continue the walk from y. When $\mathsf{layout}(x)$ has more than one tuple, we stop the right walk.

Considering $\mathcal{Q}$ only if it has one tuple is too strict, especially if $\mathcal{S}^*$ contains sequencing errors (see case 3 for the layout query in Section 7.2). Alternatively, we can define a threshold

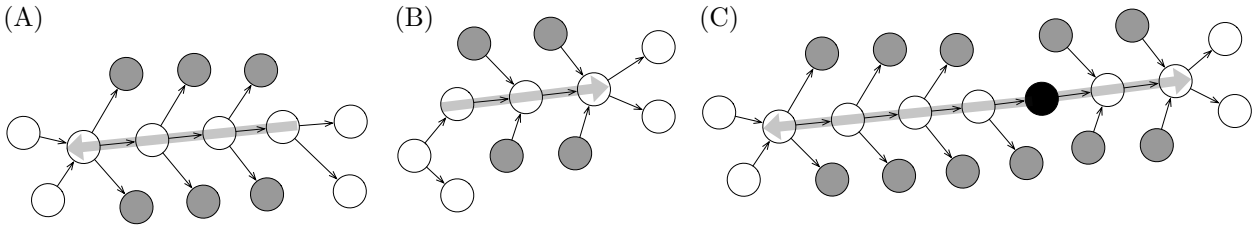

Figure 7.5: (A) A left-maximal path. The gray arrow indicates the nodes that belong to the path. It also indicates the direction in which the nodes are visited (from right to left). White nodes are adjacent to the nodes at the ends of the path. Gray nodes are adjacent to the internal nodes of the path. (B) A right-maximal path. (C) A maximal path. The black node connects the left-maximal prefix with the right-maximal suffix.

ω, and if $\mathcal{Q}$ has several tuples, we only keep those with weight $w_y \geq \omega$. By choosing a relatively high value for ω, say > 0.8, we can discard the tuples representing sequencing errors from $\mathcal{Q}$, leaving (hopefully) one extended outgoing edge for x. This filtering will increase the length of the right walk, producing thus longer contigs.

Once we finish the right walk, we begin a left walk from v to extend the contig to the left. For that purpose, we compute the node $v^{rc} = \mathsf{rcnode}(v)$ with the reverse complement label of v. We also create an empty string C_r. We start a right walk from v^{rc} and repeat the same process as with v. However, instead of inserting symbols in C_f, we insert them in C_r. After finishing the right walk, we create the final contig as $C_r^{rc} \cdot C_f$.

7.9 Experiments

We implemented rBOSS as a `C++` tool on top of the `SDSL-lite` library [76]. Our implementation encodes L using run-length compression [122] to exploit repetitions in the reads. For the experiments, we included an extra bitmap $A[1, n]$ that marks the solids nodes in rBOSS. We augmented A with a select_1 data structure to support fast access to the solid nodes. We did not include the permutation to compute the reverse complements of the dBG nodes in constant time. Instead, we used $\mathsf{label2node}$ as stated in Theorem 8. Additionally, we implemented the VO-BOSS data structure by modifying our rBOSS implementation and merging it with segments of the code[1] from Boucher et al. [21]. Our code is available at `https://bitbucket.org/DiegoDiazDominguez/eboss-dt/src/master/`. We used the compilation flags `-msse4.2 -O3 -funroll-loops -fomit-frame-pointer -ffast-math`.

We used `wgsim` [110] to simulate a sequencing dataset (in `FASTQ` format) from the E.coli genome with 15x of sequencing coverage. We generated a total of 549,845 reads, each 150 bases long, yielding a dataset of 185 MB. The input parameters for building rBOSS were k and m. We used a minimum value of 50 for k, and increased it up to 110 in intervals of 5. For every k, we used six values for m, from 15 to 40, also in intervals of 5. This setting makes up 72 indexes. We also built equivalent VO-BOSS instances using the same values for k.

[1] `https://github.com/cosmo-team/cosmo`.

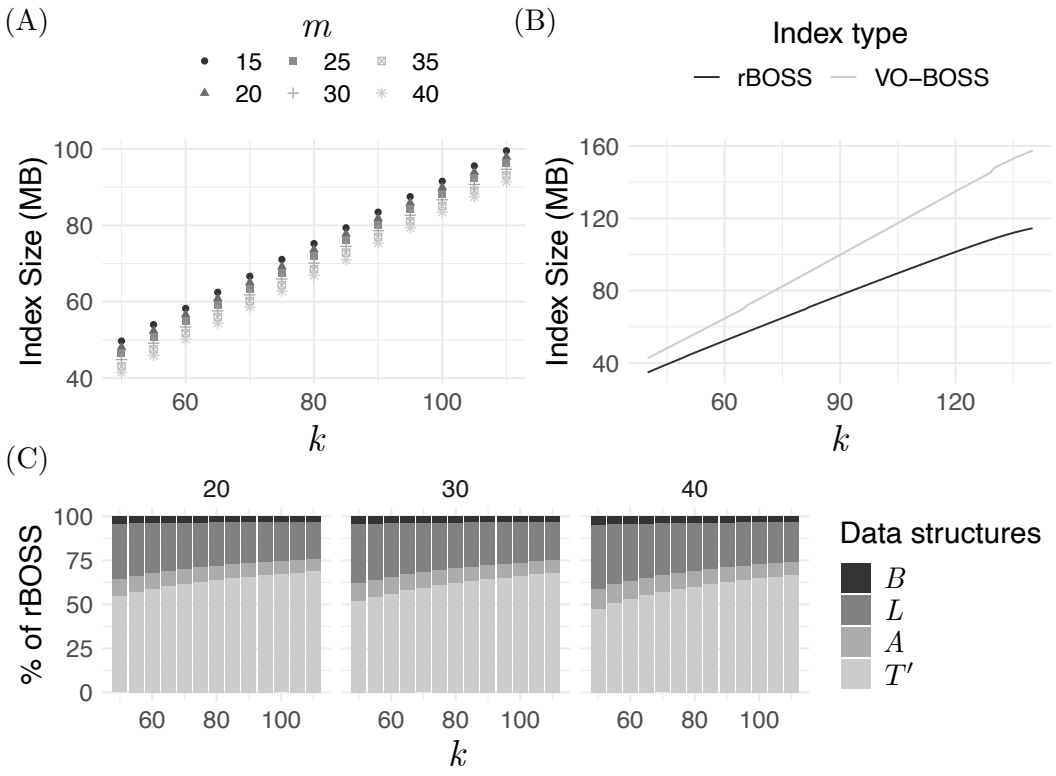

Figure 7.6: Index size statistics. (A) Sizes of the different rBOSS instances. The shapes denote distinct values for m. (B) Comparison of rBOSS against VO-BOSS. The rBOSS instances were built using $m = 30$. (C) Stacked barplot with the percentage that each substructure uses in rBOSS. The numbers on top of the plot are the m values.

7.9.1 Space and Construction Time

The sizes of the resulting rBOSS indexes are shown in Figure 7.6.A. They grow linearly with k, at $0.29 + 0.036k$ bits per input symbol, and do not depend much on m. Figure 7.6B compares the sizes of VO-BOSS and rBOSS, showing that rBOSS is 20% smaller on average. The space breakdown of rBOSS is given in Figure 7.6C. The most expensive data structure in terms of space (50%–65%) is the BP representation of T'. The sequence L uses 20%–35%, and the rest are the bitmaps B and A.

Figure 7.7 shows the elapsed times and memory peaks for the construction of the different rBOSS instances. Both metrics grow linearly with k. However, it is not clear how the parameter m affects the elapsed time. In the case of the memory peak, the smaller the value for m, the greater the peak. This pattern may be generated because smaller values of m produce larger overlap trees.

7.9.2 Time for the Primitives

We took 1000 solid nodes at random for every index and computed the mean elapsed time for the functions nextlinker, getlinkers, foverlaps. For the rBOSS indexes, we also measured the mean elapsed time for rcnode. Table 7.2 shows the results. The function nextlinker is the fastest operation among the implemented functions, with a stable time of around 1.5 μsec

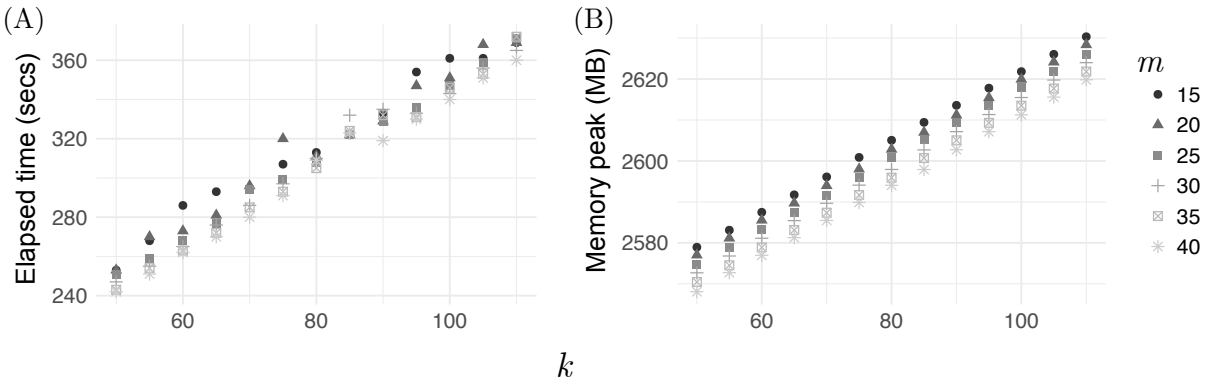

Figure 7.7: Statistics about the construction of rBOSS.

		rBOSS				VO-BOSS		
k	m	nextlinker	getlinkers	foverlaps	rcnode	nextlinker	getlinkers	foverlaps
50	20	1.49	5.09	389.42	1226.53	225.93	804.81	825.11
50	30	1.53	4.22	352.41	1209.02	216.47	581.31	802.23
50	40	2.00	3.38	255.02	1226.56	191.62	337.95	770.70
70	20	1.55	6.46	601.94	1620.22	311.46	1614.49	1155.22
70	30	1.57	5.82	546.53	1620.78	310.74	1382.25	1115.33
70	40	1.54	5.26	517.43	1621.98	297.23	1083.36	1080.17
90	20	1.73	8.11	828.12	2013.00	374.09	2441.96	1495.37
90	30	1.58	7.35	768.83	2012.36	368.71	2211.05	1444.93
90	40	1.56	6.67	714.42	2016.41	372.76	1871.19	1398.07
110	20	1.67	9.25	1088.41	2411.10	429.86	3491.07	1865.60
110	30	1.77	8.64	1014.32	2410.03	428.17	3226.45	1801.85
110	40	1.64	8.10	942.17	2414.11	436.15	2965.48	1745.31

Table 7.2: Mean elapsed time in μseconds for some of the new functions proposed in this chapter.

across different values of m and k. Operation getlinkers becomes slower as we increase k but faster as we increase m. We expected this trend because the larger k, the longer the traversal through T', but if m grows, the traversal shortens. In all the cases, getlinkers takes under 10 μsec. The cost of foverlaps grows linearly with k but also decreases as we increase m, reaching the millisecond. This performance is much slower than previous operations, dominated by the time to find the reverse complement of the shortest linker node with backwardsearch. Finally, the time of rcnode is also a few milliseconds, growing steadily with k regardless of the value for m.

Table 7.2 also compares the implementations of rBOSS and VO-BOSS. All the functions are slower in VO-BOSS, by two orders of magnitude for nextlinker and getlinkers, and by a factor around 2 for foverlaps.

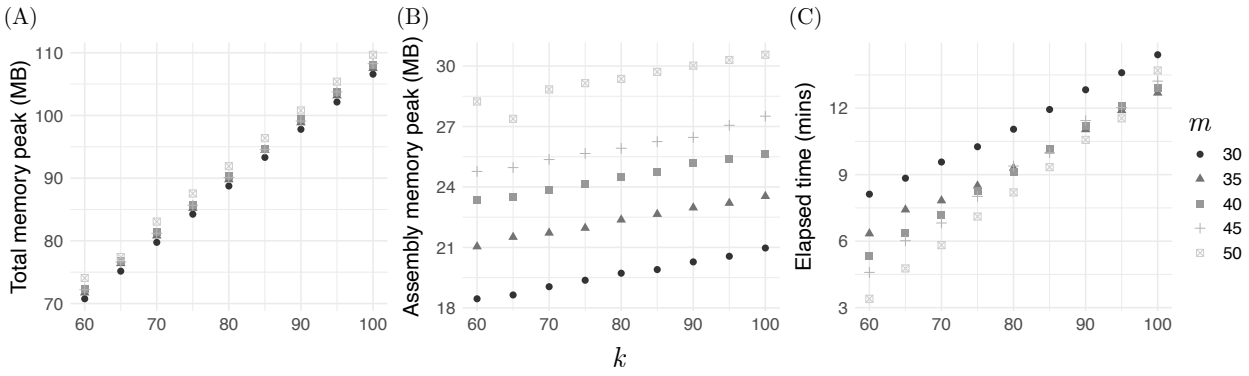

Figure 7.8: Experiments on genome assembly with rBOSS. (A) The memory peak for the assembly. (B) The memory peak of the assembly, but without considering the index size. (C) Elapsed time for the assembly.

7.9.3 Genome Assembly

We empirically assessed the genome assembler described in Section 7.8. We partially implemented the idea because our current version of rBOSS does not fully support layout yet. Our implemented assembler differs from the method described in Section 7.8 in that when it reaches an s-node x with out-degree one during a walk, it calls E = getlinkers(x), and if all the nodes in E have out-degree one; it continues the walk from the linker node in E whose label matches the smallest suffix of nodelabel(x).

We tested our assembler using the same E. coli dataset we used in the previous experiments. As before, we produced several instances of rBOSS. We chose a minimum value for k of 60, increasing it up to 100 in intervals of 5. For each k, we selected five values for m, from 30 to 50, also in intervals of 5. We produced contigs in those instances of rBOSS using our assembly algorithm. We selected the assemblies obtained with the rBOSS instances of m = 50 and compared them against the results of `bcalm` [33], a compact representation for dBGs that spells contigs from the unary paths. We build equivalent dBGs instances for `bcalm` using the same values for k we used for rBOSS. We selected m = 50 because this value produced the longest contigs in rBOSS. We assessed the assembly results using the mean and maximum contig sizes.

The memory peak and the elapsed time for the assemblies were again linear on k (see Figure 7.8). The memory peaks on top of rBOSS were generally small (between 18 and 31 MB, see Figure 7.8B). We expected this behavior as spelling contigs from the dBG is an online task. Interestingly, the memory peaks increased for higher values of m. Choosing high values for m reduces the number of valid overlaps in the extended dBG, so the assembler produces a larger number of contigs, but their lengths are shorter than those obtained with small values of m. The problem is that our method generates overlapping contigs, and maintaining them in RAM increases the memory peak.

On the other hand, the elapsed time decreased with higher values of m (Figure 7.8C). This trend is more evident for small values of k, less than 70. High values for m generate the dBG nodes to have fewer valid overlaps, so the assembler spent less time computing the

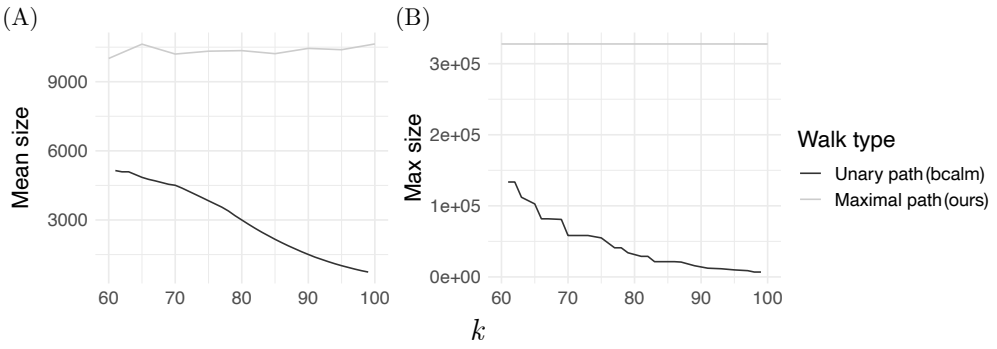

Figure 7.9: Comparison of the contigs generated with our assembler and `bcalm`. The x-axis shows the different values we used for k. The dark gray lines represent the contigs spelled by `bcalm` from the unary paths of the regular dBG. The light gray lines are the contigs spelled by our assembler from the maximal paths in the extended dBG of rBOSS ($m = 50$). The (A) side of the figure depicts the mean contig sizes for the different values of k while the (B) side depicts the length of the longest contig.

extended edges. The impact of m becomes less relevant as we increase k. We believe this is because the dBG becomes highly disconnected when k is high, and most of the elapsed time is spent on computing extended edges, regardless of the value we use for m.

The `bcalm` framework consumed less computational resources than ours in the experiments. The time it took `bcalm` to produce a dBG and then spell contigs from it was no more than 2 minutes in all its instances, with a memory peak of at most 674 MB. In contrast, the time our framework took to build a dBG and then spell the contigs was more than 7 minutes in all the instances, with a memory peak of at least 2.5 GB. Interestingly, the memory peak of `bcalm` did not increase linearly with k. The greatest memory peak was 492.16 MB ($k = 63$), and the smallest one was 674.94 ($k = 64$). It is important to note that rBOSS and our assembler are not optimized yet. We construct the dBG using the LCP array and the suffix array of $\mathcal{S}^*$. In contrast, `bcalm` uses bloom filters and minimizers (see Section 2.3), which are faster in practice.

Regarding the assembly results, our method produced, on average, much longer contigs than `bcalm` (see Figure 7.9A). The mean contig size of our assembler does not vary much as we modify k. It increased from $10,009$ characters with $k = 60$ to $10,639$ characters with $k = 100$. In contrast, the mean contig size for `bcalm` decreased with higher values of k. Using $k = 61$, the mean contig size was 5,145, while for $k = 99$, it reduced to 4,112. We see a similar pattern for the longest contig (Figure 7.9B). The longest contig was, on average, much longer with our method than with `bcalm` (327,742 versus 45,608, respectively). Additionally, the length of the longest contig in our assembler remained similar throughout the different values of k (between 327,710 and 327,784). On the other hand, the longest contig in `bcalm` dramatically reduced its length as we increased k; from 133,531 with $k = 61$ to 6,853 with $k = 99$.

The advantage of our method is that we can use the extended edges of layout to reconnect the dBG at our convenience when k is high. This feature allows producing long contigs even though the graph is highly disconnected.

Chapter 8

Succinct Colored de Bruijn Graphs

This chapter describes a succinct index for reads that we can use to extract biological information. Similarly to the framework described in Chapter 7, this data structure uses a dBG built from the reads. However, instead of computing extra overlaps, we enrich the graph with colors (see Section 4.3.1). The idea of coloring the dBG is not new, nor are the data structures we use to encode the final representation. The main novelty of our work is the way we assign the colors to the graph and the algorithms we develop on top of the index to answer biological questions.

A colored dBG G of order k, built from a string collection $\mathcal{S}$, assigns a specific integer value c_i (a *color*) to every $S_i \in \mathcal{S}$, and then stores c_i in the edges of the walk of G that spell S_i. When two or more strings in $\mathcal{S}$ have kmers in common, their paths in G get entangled (i.e., they share some nodes and edges). For that reason, the edges in G can have more than one possible color assigned. We then store the colors as a binary matrix $\mathcal{C}$, where the rows represent the edges and the columns represent the colors.

The concept of a colored dBG is useful for genomic analyses because we can disentangle the strings of $\mathcal{S}$ as we walk through G. Suppose we are traversing a path where the last z edges are colored with some c_i. When we reach a node v with out-degree > 1, we can continue the traversal through the outgoing edge colored with c_i (if any). This process guarantees that the label spelled by the last $z + 2$ nodes in the path exists in $\mathcal{S}$. Supporting this feature is not possible in a regular dBG as all the outgoing edges of v are equally likely to produce a string that does not exist in $\mathcal{S}$.

The original colored dBG (Section 4.3.1) considers one single graph built from multiple string collections, where the colors are assigned to the collections instead of the strings. The color space overhead in this scheme is not significant as the number of distinct collections is small or moderate in most genomic applications. However, in our case, we consider $\mathcal{S}$ to be a multiset of reads, and we require to assign a color to every $S_i \in \mathcal{S}$. This alternative scheme enables later to disentangle the sequence of S_i from G. The space overhead in this setting is considerable as $\mathcal{S}$ can contain millions of reads, meaning that we need a color matrix with millions of columns.

Our main interest in coloring G is to disentangle strings in $\mathcal{S}$. This allows us to relax the representation. First, it unnecessary to color the full walk in G spelling $S_i \in \mathcal{S}$, only the critical points. Second, similarly to the idea of Alipanahi et al. [1], we can reuse the same colors for the reads in $\mathcal{S}$ that do not have kmers in common.

Considering these ideas, we propose a greedy algorithm that partially colors G and reuses the same colors for different reads when possible. Once we assign the colors, we encode G using the BOSS representation and $\mathcal{C}$ using compact data structures. We develop two algorithms on top of our representation that can serve as a base to perform bioinformatic analyses in succinct space. The first algorithm extracts the original reads from the dBG and the second algorithm assembles contigs of the source DNA from which $\mathcal{S}$ was obtained.

Our experimental results show that, on average, the percentage of nodes in BOSS that need to be colored is about 12.4%, the space usage of the whole index is about half the space of the plain representation of $\mathcal{S}^*$ (taken as 1 byte/DNA symbol), and that more than 99% of the original reads can be reconstructed from the index.

A preliminary version of this work [49] was presented at the *26th International Symposium on String Processing and Information Retrieval* (SPIRE'19).

8.1 Definitions

Let $\mathcal{S} = \{S_1, \ldots, S_q\}$ be a collection of q reads, and let $\mathcal{S}^* = \{S_1, S_1^{rc}, \ldots, S_q, S_q^{rc}\}$ be a collection of size $2q$ that contains the strings in $\mathcal{S}$ along with their reverse complements. Let us denote the dBG of order k constructed from $\mathcal{S}^*$ as $G = (V, E)$. Similarly, we denote an instance of BOSS for G as $BOSS(G) = (V', E')$, where V' and E' include the dummy nodes and their edges (see Section 4.3.1). A node in V' is considered a *starting* node if its label is of the form $\$A$, where $\$$ is a dummy symbol and A is a $k-2$ prefix in one or more sequences of $\mathcal{S}^*$. Equivalently, a node is considered an *ending* node if its label is of the form $A\$$, with A being a $k-2$ suffix in one or more sequences of $\mathcal{S}^*$. Nodes whose labels do not contain dummy symbols are *solid*, and solid nodes with at least one incoming node with out-degree > 1 are *critical*. For practical reasons, we define two extra functions for BOSS, isstarting and isending. We use them to check if a node is starting or ending, respectively.

A *walk* $P = v_1, v_2, \ldots, v_t$ over $BOSS(G)$ is a sequence of t nodes, where every v_i, with $i \in [1, t-1]$, is connected by an edge with v_{i+1}. P is a *path* if all its nodes are different. When $v_1 = v_t$, P is said to be a *cycle*. A sequence $S_i \in \mathcal{S}^*$ is *unambiguous* if there is a path in $BOSS(G)$ whose label matches the sequence of S_i and if no pair of colored nodes in $(u, v) \in P$ share a predecessor node $v' \in P$. In any other case, S_i is *ambiguous*. Finally, the path P_i that spells the sequence of S_i is said to be *safe* if every one of its branching nodes has only one successor colored with the color assigned to S_i.

We assume that $\mathcal{S}^*$ is a *factor-free* collection, i.e., no $S_i \in \mathcal{S}^*$ is also a substring of another sequence S_j, with $i \neq j$.

8.2 Coloring a dBG of Reads

In this section, we define a coloring scheme for $BOSS(G)$ that generates a more succinct color matrix, and that allows us to reconstruct and assemble unambiguous sequences of $\mathcal{S}^*$. We use the dBG of $\mathcal{S}^*$ because most of the bioinformatic analyses require the inspection of the reverse complements of the reads. Unlike previous works (Section 4.3.1), the rows in $\mathcal{C}$ represent the nodes in $BOSS(G)$ instead of the edges.

8.2.1 Partial Coloring

We make $\mathcal{C}$ more sparse by coloring only those nodes in the graph that are *strictly* necessary for reconstructing the sequences. We formalize this idea with the following lemma:

Lemma 9 For the path P_i in $BOSS(G)$ spelling an unambiguous sequence $S_i \in \mathcal{S}^*$ to be safe, we have to color the starting node $s_i \in P_i$ that encodes the $k-2$ prefix of S_i, the ending node $e_i \in P_i$ that encodes the $k-2$ suffix of S_i and the critical nodes in P_i.

Proof. We start a walk from s_i using the following rules: (i) if the current node v in the walk has out-degree one, then we follow its only outgoing edge, (ii) if v has out-degree > 1 , then we inspect its successor nodes and follow the one colored with the same color of s_i and (iii) if v is equal to e_i, then we stop the walk. □

Note that the successor nodes of a branching node (i.e., with out-degree > 1) are critical by definition, so they are always colored. On the other hand, nodes with out-degree one do not require a color inspection because they have only one possible way out.

Coloring the nodes s_i and e_i for every S_i is necessary; otherwise, it would be difficult to know when a path starts or ends. Consider, for example, using the solid nodes that represent the $k-1$ prefix and the $k-1$ suffix of S_i as starting and ending points respectively. It might happen that the starting point of S_i can also be a critical point of another sequence S_j. If we start a reconstruction from s_i and pick the color of S_j, then we will generate an incomplete sequence. A similar argument can be used for ending nodes. The concepts associated with our coloring idea are depicted in Figure 8.2.

8.2.2 Unsafe Coloring

We can use the recoloring idea of Alipanahi et al. [1] to reduce the number of columns in $\mathcal{C}$. Still, using the same colors for unrelated strings is not safe for reconstructing unambiguous sequences.

Lemma 10 Using the same color c for two unambiguous sequences $S_i, S_j \in \mathcal{S}^*$ that do not share any $k-1$ substring can make the dBG paths for S_i or S_j unsafe.

Proof. Assume there is another pair of sequences $S_x, S_y \in \mathcal{S}^*$ that do not share any $k-1$ subsequence either, to which we assign them color c'. Suppose that the paths of S_x and S_y

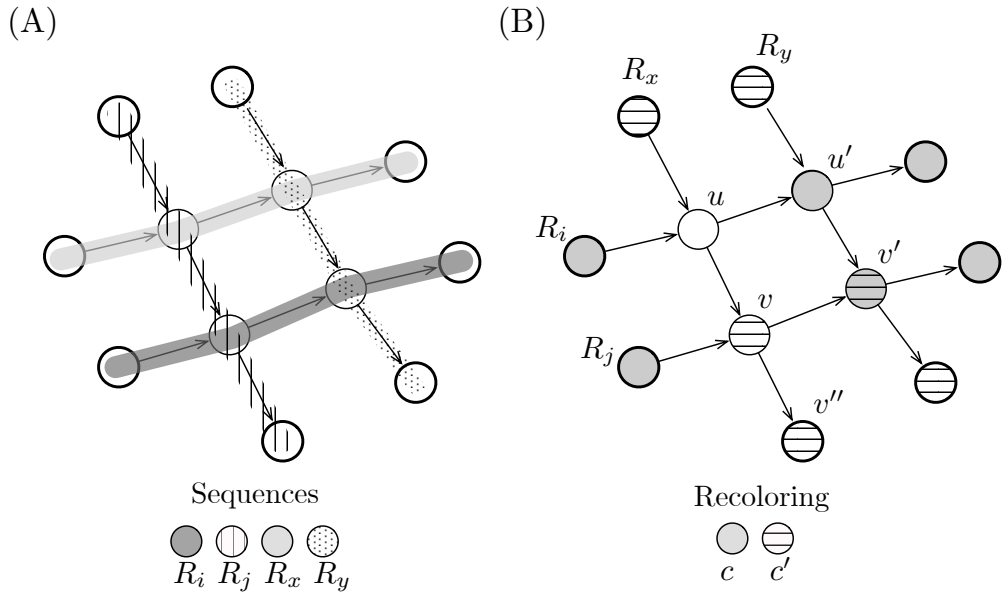

Figure 8.1: Example of unsafe paths produced by a graph recoloring. (A) The dBG generated from the unambiguous sequences R_i, R_j, R_x and R_y. Every texture represents the path of a specific string. (B) Recolored dBG. Sequences R_i and R_j are assigned the same color c (light gray) as they do not share any $k-1$ substring. Similarly, sequences R_x and R_y are assigned another color c' (horizontal lines) as they do not share any $k-1$ sequence neither. Nodes u, u', v, v' and v'' are those mentioned in the proof of Lemma 10. The sequences of R_i and R_x cannot be reconstructed as their paths become unsafe after the recoloring.

cross the paths of S_i and S_j such that the resulting dBG topology resembles a grid. In other words, if S_i has the edge (u, u') and S_j has the edge (v, v'), then S_x has the edge (u, v) and S_y has the edge (u', v'). In this scenario, v will have two successors, node v' from the path of S_j and some other node v'' from the path of S_x. Both v' and v'' are critical by definition so they will be colored with c and c' respectively. The problem is that node v' is also a critical node for S_y, so it will also have color c'. The reason is that u', a node that precedes v', appears in S_i and S_y. As a consequence, the path of S_x is no longer safe because one of its nodes (v in this example) has two successors colored with c'. A similar argument can be made for S_i and color c. Figure 8.1 depicts the idea of this proof. □

When spurious edges connect paths of unrelated sequences that are assigned the same color (as the in the proof of Lemma 10), we can generate chimeric strings if, by error, we follow one of those edges. In our coloring algorithm, we solve this problem by assigning different colors to those strings with sporadic edges, even if they do not share any $k-1$ substring.

8.2.3 Safe and Greedy Coloring

Our greedy coloring algorithm starts by marking in a bit vector $N = [1, |V'|]$ the p nodes of $BOSS(G)$ that need to be colored (starting, ending and critical). After that, we create an array M of p entries. Every $M[j]$ with $j \in [1, p]$ will contain a vector that stores the colors

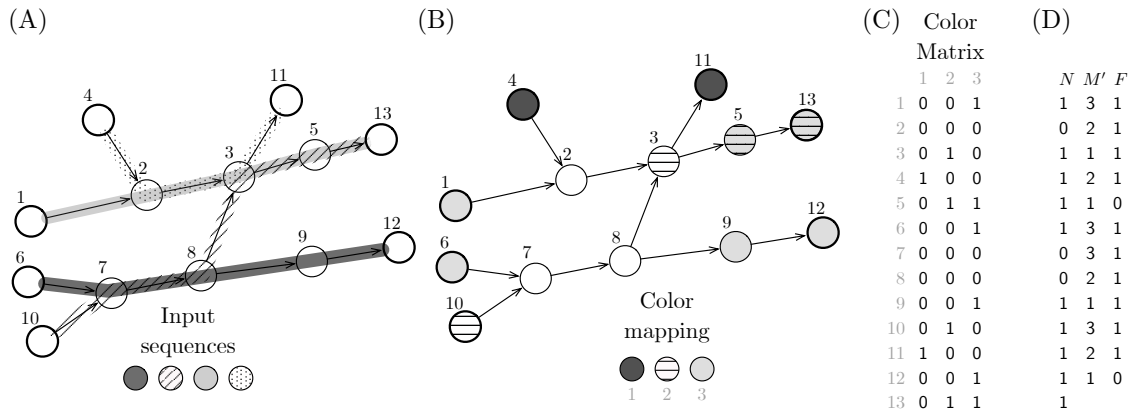

Figure 8.2: Succinct colored dBG. (A) The topology of the graph. Colors and textures represent the paths that spell the input sequences of the dBG. Numbers over the nodes are their identifiers. Nodes 4,1,6 and 10 are starting nodes (darker borders). Nodes 11,13 and 12 are ending nodes and nodes 3,9,11 and 5 are critical. (B) Result of our greedy coloring algorithm. (C) The binary matrix $\mathcal{C}$ that encodes the colors of Figure B. The left side is $\mathcal{C}$ in its uncompressed form and the right side is our succinct version of $\mathcal{C}$ using the arrays N, M', and F.

of the jth colored node in the BOSS ordering. We also add rank_1 support to N to map a node $v \in V$ to its array of colors in M. Thus, its position can be inferred as $\mathsf{rank}_1(N, v)$.

The only inputs we need for the algorithm are N, $\mathcal{S}^*$ and $BOSS(G)$. For every $S_i \in \mathcal{S}^*$ we proceed as follows; we append a \$ symbol at the ends of S_i, and then use the function label2node (Section 4.3.1) to find the node v labeled with the $k-1$ prefix of S_i. Note that this prefix will map a starting node as we appended \$ to S_i. From v, we begin a walk on $BOSS(G)$ and follow the edges whose symbols match the characters in the suffix $S_i[k..]$. As we move through the edges, we store in an array W_i the starting, ending, and critical nodes associated with S_i. Additionally, we store into another array I_i the neighboring nodes of the walk that need to be inspected to assign a color to S_i. The rules for adding elements into I_i are as follows;

- If v is a node with out-degree > 1 in the path of S_i, then we add all its outgoing nodes into I_i.
- If v is a node with in-degree > 1 in the path of S_i, then we visit every incoming node v' of v, and if v' has out-degree > 1, then we insert into I_i the outgoing nodes of v'.

Once we finish the walk, we create a set H and fill it with the colors that were previously assigned to the nodes in I_i and W_i. After that, we pick the smallest color c' that is not in H, and add it to every array $M[\mathsf{rank}_1(N, j)]$ with $j \in W_i$. After we process all the sequences in $\mathcal{S}^*$, the final list of colors is represented by the values in M. The complete process for coloring S_i is described in more detail by the procedure greedycol in Algorithm 6.

The construction of the sets W_i and I_i is independent for every string in $\mathcal{S}^*$, so it can be done in parallel. However, the construction of H and the assignment of the color c' to the elements of W_i has to be performed sequentially as all the sequences in $\mathcal{S}^*$ need concurrent

Algorithm 6 Function greedycol

```
1: proc greedycol(N, S_i, M)          ▷ N is a bitmap, S_i is a string and M is array of lists
2:     S_i ← $S_i$                                  ▷ append dummy symbols at the ends of S_i
3:     v ← label2node(S_i[1, k − 1])
4:     W ← ∅
5:     I ← I ∪ rank_1(N, v)
6:     for each r ∈ S_i[k − 1..] do                 ▷ traverse the dBG path of S_i
7:         o ← outdegree(v)
8:         if o > 1 then
9:             for j ← 1 to o do
10:                I ← I ∪ rank_1(N, outneighbor(v, j))
11:        i ← indegree(v)
12:        if i > 1 then
13:            for j ← 1 to i do
14:                v' ← inneighbor(v, j)
15:                o' ← outdegree(v')
16:                if o' > 1 then
17:                    for j ← 1 to o' do
18:                        I ← I ∪ rank_1(N, outneighbor(v', j))
19:        if N[v] is true then
20:            W ← W ∪ rank_1(N, v)
21:        v ← outneighbor(v, r)
22:    W ← W ∪ rank_1(N, v)
23:    I ← I ∪ rank_1(N, v)
24:    initialize set H
25:    for each n ∈ I do                            ▷ compute the colors already used
26:        for each c ∈ M[n] do
27:            insert(H, c)
28:    c' ← minimum color not in H
29:    for each n ∈ W do                            ▷ color the nodes
30:        M[n] ← M[n] ∪ c'
```

access to M.

8.2.4 Ambiguous Sequences

The main limitation of our coloring method is that it cannot be used to safely retrieve ambiguous sequences.

Lemma 11 Ambiguous sequences of $\mathcal{S}^*$ cannot be reconstructed safely from the color matrix $\mathcal{C}$ and $BOSS(G)$.

Proof. Assume the collection $\mathcal{S}^*$ is composed just by one string $S_1 = XbXc$, where X is a repeated substring and b, c are two different symbols in Σ. Consider also that the order of $BOSS(G)$ is $k = |X| + 1$. The instance of $BOSS(G)$ will have a node v labeled with X, with two outgoing edges, whose symbols are b and c. Given our coloring scheme, the successor nodes of v will be both colored with the same color. As a consequence, if during a walk we reach v, we will get stuck because there is not enough information to decide which is the correct edge to follow (both successor nodes have the same color). □

A sequence S_i will be ambiguous if it has the same $k-1$ pattern in two different contexts. Another case in which S_i is ambiguous is when a spurious edge connects an uncolored node of S_i with two or more critical nodes in the same path. Note that an ambiguous sequence will always be encoded by an unsafe path, regardless of the recoloring algorithm. In general, the number of ambiguous sequences will depend on the value we use for k.

8.3 Compressing the Colored dBG

The pair (M, N) can be regarded as a compact representation of $\mathcal{C}$, where the empty rows were discarded. Every $M[i]$, with $i \in [1, |M|]$, is a row with at least one value, and every color $M[i][j]$, with $j \in [1, |M[i]|]$, is a column. However, M is not succinct enough to be practical. We are still using a machine word for every color of M. Besides, we need $|M|$ extra words to store the pointers for the lists in M.

We compress M by using an idea similar to the one implemented in BOSS to store the edges of the dBG. The first step is to sort the colors of every list $M[i]$. Because the greedy coloring generates a set of unique colors for every node, each $M[i]$ becomes an array of strictly increasing elements after the sorting. Thus, instead of storing the values explicitly, we encode them as differences, i.e., $M[i][j] = M[i][j] - M[i][j-1]$. After transforming M, we concatenate all its values into one single list M' and create a bit map $F = [1, |M'|]$ to mark the first element of every $M[i]$ in M'. We store M' using Elias-Fano encoding (Section 2.1.3) and F using a compressed representation for bit vectors (Section 2.2.1). Finally, we add select_1 support to F to map a range of elements in M' to an array in M. The full representation of the color matrix is $\mathcal{C} = N + F + M'$ (see Figure 8.2). The complete index of the colored dBG is thus composed of our version of $\mathcal{C}$ plus $BOSS(G)$. We now formalize the idea of retrieving the colors of a node from the succinct representation of $\mathcal{C}$.

- getcolors(v): list of colors assigned to node v.

Theorem 11 the function getcolors(v) computes in $\mathcal{O}(c)$ time the c colors assigned to v.

Proof. We first compute the rank r of node v within the colored nodes. This operation is carried out with $r = \mathsf{rank}_1(N, v)$. After retrieving r, we obtain the range $M'[i, j]$ where the colors assigned to v lie. For this purpose, we perform two select_1 operations over F, $i = \mathsf{select}_1(F, r)$ and $j = \mathsf{select}_1(F, r + 1) - 1$. Finally, we scan the range $M'[i, j]$, and as we read the values, we incrementally reconstruct the colors from the differences. All the rank and select operations take $\mathcal{O}(1)$ time, and reading the $c = j - i + 1$ entries from M' takes $\mathcal{O}(c)$ time as retrieving an element from an Elias-Fano-encoded array takes $\mathcal{O}(1)$ time. In conclusion, computing the colors of v takes $\mathcal{O}(c)$ time. □

8.4 Reconstructing Unambiguous Sequences

We describe now an online algorithm that works on top of our index and that reconstructs all the unambiguous sequences in $\mathcal{S}^*$. We cannot tell, however, if a reconstructed string S_i was present in the original collection $\mathcal{S}$ or if it was its reverse complement S_i^{rc}. This is not really a problem, because a sequence and its reverse complement are equivalent in most bioinformatic analyses.

The algorithm receives as input a starting node v. It first computes an array A with the colors assigned to v using the function getcolors, and then initializes a string $S = \mathsf{nodelabe}(v)$. For every color $a \in A$, the algorithm performs the following steps; initializes two temporary variables, an integer $v' = v$ and string $S' = S$, and then begins a walk over $BOSS(G)$ staring from v'. If the out-degree of v' is one, then the next node in the walk is the outgoing node $v' = \mathsf{outneighbor}(v', 1)$. On the other hand, if the out-degree of v' is more than one, then the algorithm inspects all the outgoing nodes of v' to check which one of them is the node v'' colored with a. If there is only one such v'', then it sets $v' = v''$. This procedure continues until v' becomes an ending node. During the walk, the edge symbols are appended to S'. When the algorithm reaches an ending node, it reports $S'[1, |S'| - 1]$ as the reconstructed sequence.

If at some point during a walk, the algorithm visits a node with out-degree > 1, and with more than one outgoing node colored with a, then it aborts the reconstruction of the string as the path is unsafe for color a. Then, it returns to v and continues with the next sequence. The complete procedure is detailed in the function buildseqs of Algorithm 7.

8.5 Assembling Contigs

We propose a simple procedure called contigassm that traverses the dBG and uses the color information to weight the outgoing edges on the fly in order to estimate which are the most probable walks of $BOSS(G)$ spelling contigs (Section 4.3). This procedure is equivalent to simulating a traversal over the layout of unambigous reads. Our algorithm is not a full

Algorithm 7 Function buildseqs

```
1:  proc buildseqs(v)                                   ▷ v is a starting node
2:      L ← ∅                                           ▷ list of sequences rebuilt from v
3:      A ← getcolors(v)
4:      S ← nodelabel(v)                                ▷ initialize an string with the label of v
5:      for each a ∈ A do
6:          v' ← v                                      ▷ temporal dBG node
7:          S' ← S, m ← 0
8:          while isending(v') is false do
9:              o ← outdegree(v')
10:             if o is 1 then
11:                 S' ← S' ∪ edgesymbol(v', 1)         ▷ push the new symbol into S'
12:                 v' ← outneighbor(v', 1)
13:             else
14:                 m ← 0, i' ← 0, t ← 0
15:                 for i ← 1 to o do                   ▷ check which successors of v' has color a
16:                     x ← outneighbor(v', i)
17:                     if a ∈ getcolors(x) then
18:                         t ← x, m ← m + 1, i' ← i
19:                 if m = 1 then
20:                     S' ← S' ∪ edgesymbol(v', i')
21:                     v' ← t
22:                 else
23:                     break
24:         if m = 1 then
25:             L ← L ∪ S[2, |S| − 1]
26:     return L
```

assembler. It only deals with the extraction of contigs from the dBG. It does not address other technical issues such as paired-end reads, scaffolding or gap filling.

For simplicity, we identify every unambiguous reads $S_i \in \mathcal{S}^*$ in $BOSS(G)$ with the pair (c, v), where c is the color assigned to S_i and v is the starting node of its path.

The input for contigassm is a starting node v and a set L with the identifiers of the reads that were already assigned to contigs. The output will be the list $\mathcal{P}_v = \{C_1^v, \ldots, C_u^v\}$ with the contigs spelled by walks in $BOSS(G)$ beginning at v.

We obtain $\mathcal{P}_v$ by iterating through the colors of v. For every $c_i \in \mathsf{getcolors}(v)$, we proceed as follows. We initialize a hash table H, where the keys are colors and their values are nodes in $BOSS(G)$. We also initialize the contig $C_i = \mathsf{label}(v)$ and insert the pair (c_i, v) to H. Subsequently, we begin a walk over $BOSS(G)$ from v. If v has out-degree > 1, we follow the outgoing node labeled with c_i. For every new node v' we reach during the walk, we check if one of its incoming nodes, say u, is a starting node. If so, then for each $c \in \mathsf{getcolors}(u)$, we insert the pair (c, u) into H. On the other hand, if one of the outgoing nodes of v', say u', is an ending node, then we obtain the set $C = \mathsf{getcolors}(u')$. For every $c \in C$, we check if

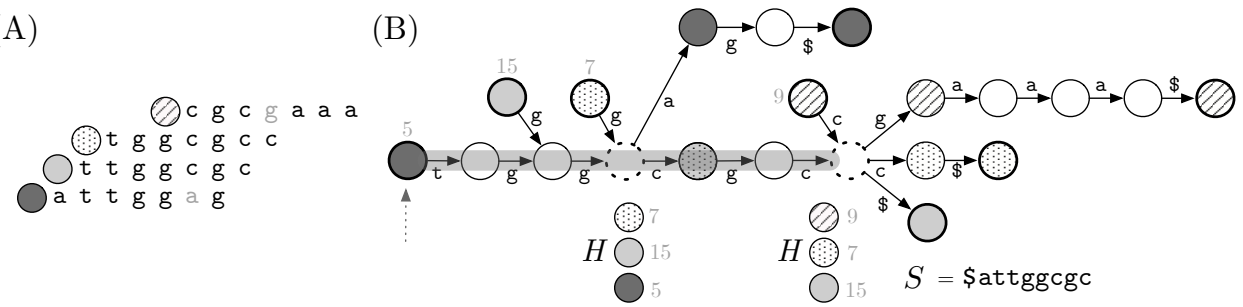

Figure 8.3: Example of the assembly of a contig using our index. (A) Inexact overlap of four sequences. The circle to the left of every string represents its color in the dBG. Light gray symbols are mismatches in the overlap. (B) The colored dBG of the sequences. Circles with darker borders are starting and ending nodes. Light gray values over the starting nodes are their identifiers. The contig assembly begins in node 5 (denoted with a dashed arrow) and the threshold x to continue the extension is set to 0.5. The state of the hash table H when the walk reaches a branching node (dashed circles) is depicted below the graph. The assembly ends in the rightmost branching node as it has not a successor node that contains at least 50% of the colors in Q. The final contig is shown as a light grey path over the graph, and its sequence is stored in S.

c exists in H as a key. If it does, then we extract its associated node x, remove (c, x) from H, and insert it to L. After checking the neighboring nodes of v', we continue the walk by selecting one of the outgoing edges of v'. If v' has out-degree one, we take its only outgoing edge. However, if v' has out-degree $>$ 1, we inspect first how the colors in H distribute between the outgoing nodes of v'. If there is only one successor of v', say v'', colored with at least f% of the colors in H, where f is a parameter, we follow v' and remove the other colors from H, along with their values. Once we select the outgoing node, we add the label of (v', v'') to C_i. The walk stops if:

- There is no such v'' that meets the f threshold.
- There is more than one outgoing node of v' with the same color.
- The node v' has out-degree one, but the outgoing node is an ending node.

After finishing the walk, we insert the suffix $C_i[2..]$ to $\mathcal{P}_v$. Algorithm 8 describes in detail contigassm, and a graphical example is shown in Figure 8.3.

8.6 Experiments

We used a set of reads generated from the E.coli genome[1] to test the ideas described in this chapter. The raw file was in `FASTQ` format and contained 14,214,324 reads of 100 characters each. We preprocessed the file by removing sequencing errors using the tool `SPAdes` [5], and discarding reads with `n` symbols. The preprocessing yielded a dataset of 8,655,214 reads (a `FASTQ` file of 2GB). Additionally, we discarded sequencing qualities and the identifiers of the reads as they are not considered in our data structure. From the resulting collection $\mathcal{S}$ (a

[1] `http://spades.bioinf.spbau.ru/spades_test_datasets/ecoli_mc`.

Algorithm 8 Function contigassm

```
1: proc contigassm(v, f, L)          ▷ v is a starting node, f is a threshold, and L is a set
2:     P_v ← ∅                                                  ▷ list of contigs obtained from v
3:     for each c ∈ getcolors(v) do
4:         C_c ← nodelabel(v)                                   ▷ contig produced with color c
5:         insert(H, c, v)
6:         v' ← v
7:         while true do
8:             if indegree(v') > 1 then
9:                 x ← inneighbor(v', 1)
10:                if isstarting(x) then                        ▷ add new reads to the contig
11:                    for each c' ∈ getcolors(x) do
12:                        insert(H, c', x)
13:            if o ← outdegree(v') > 1 then
14:                m ← 0, i' ← 0, t ← 0, A' ← ∅
15:                for i ← 1 to o do                ▷ compute the most likely outgoing node
16:                    x ← outneighbor(v', i)
17:                    A ← getcolors(x)
18:                    if isending(x) then                   ▷ flag reads ending at x as used
19:                        for each c' ∈ A do
20:                            u ← retrieve(H, c')
21:                            delete(H, c')
22:                            insert(L, (c', u))
23:                    else
24:                        w ← % of colors in A appearing in H as keys
25:                        if w ≥ f then
26:                            t ← x, m ← m + 1, i' ← i, A' ← A
27:                if m = 1 then          ▷ only one outgoing node meets the threshold
28:                    C_c ← C_c ∪ edgesymbol(v, i')
29:                    v' ← t
30:                    Delete entries in H whose keys are not in A'
31:                else
32:                    break          ▷ more than on possible way to extend the contig
33:            else
34:                s ← edgesymbol(v', 1)
35:                if s = $ then break
36:                C_c ← C_c ∪ s
37:                v' ← outneighbor(v', 1)
38:        P_v ← P_v ∪ C_c
39:    return P_v
```

dBG order	Total number of nodes	Number of solid nodes	Number of edges	Index size	Compression ratio
25	106,028,714	11,257,781	120,610,151	446.38	1.86
30	142,591,410	11,425,646	157,186,548	443.82	1.87
35	179,167,289	11,561,630	193,773,251	441.18	1.88
40	215,751,326	11,667,364	230,365,635	438.23	1.90
45	252,337,929	11,743,320	266,958,709	435.30	1.91
50	288,925,674	11,791,640	303,552,318	432.13	1.92

Table 8.1: Statistics about the different colored dBGs generated in the experiments. The index size is expressed in MB and considers the space of $BOSS(G)$ plus the space of our succinct version of $\mathcal{C}$. The compression ratio was calculated as the space of the plain representation of the reads (833.67 MB) divided by the index size.

text file of 833.67 MB), we created another collection $\mathcal{S}^*$ that considers the elements in $\mathcal{S}$ and their reverse complements.

Our index for colored dBGs and the algorithms greedycol, buildseqs were implemented in C++[2] on top of the `SDSL-lite` library [76]. In our implementation, we precompute the lengths of arrays M' and F before coloring the graph so that when we apply our greedy algorithm, we store the color information directly to them. We do not use the array M described in Section 8.2.3 because we cannot store its vectors contiguously in RAM, making the coloring process not cache-friendly. The problem is that we do not know how many colors each of the lists of M will have before applying our greedy algorithm. All our code, except contigassm, can be executed using multiple threads.

We built six instances of our index using $\mathcal{S}^*$ as input. We chose different values for k, from 25 to 50 in steps of five. The coloring of every one of these instances was carried out using eight threads. Statistics about the topology of the graphs are shown in Table 8.1, and statistics about the coloring process are shown in Table 8.2. In every instance, we reconstructed the unambiguous reads (see Table 8.1). Additionally, we generated an FM-index of $\mathcal{S}^*$ to locate the reconstructed reads and checked that they were real sequences. All the tests were carried out on a machine with Debian 4.9, 252 GB of RAM and processor Intel(R) Xeon(R) Silver @ 2.10GHz, with 32 cores.

8.7 Results

The average compression ratio achieved by our index is 1.89, meaning that, in all the cases, the data structure used about half the space of the plain representation of the reads (see Table 8.1). We also note that the smaller the value for k, the greater the size of the index. This behavior is expected as the dBG becomes denser when we decrease k. Thus, we have to store a higher number of colors per node.

The number of colors of every instance is several orders of magnitude smaller than the

[2] `https://bitbucket.org/DiegoDiazDominguez/colored_bos/src/master`.

dBG order	Number of colored nodes	Number of colors	Color space usage	Ambiguous sequences	Elapsed Time	Memory peak
25	21,882,874	6,552	83.03	1904	97.25	4,391
30	21,907,324	4,944	79.14	1502	92.52	4,119
35	21,926,687	2,924	75.27	1224	85.52	3,847
40	21,942,083	2,064	71.40	1054	81.20	3,575
45	21,954,138	1,888	67.51	714	75.12	3,303
50	21,964,947	1,689	63.58	176	69.98	3,030

Table 8.2: Statistics about our greedy coloring algorithm. The column "Color space usage" refers to the percentage of the index space used by our succinct version of $\mathcal{C}$. Elapsed time and memory peak are expressed in minutes and MB, respectively, and both consider only the process of building, filling, and compacting the color matrix.

number of reads, being $k = 25$ the instance with the most colors (6552) and $k = 50$ the instance with the fewest (1689). Even though the fraction of colored nodes in every instance is small, the percentage of the index space used by the color matrix is still high (73% on average). Regarding the time for coloring the graph, it seems to be reasonable for practical purposes if we use several threads. In fact, building, filling and compacting $\mathcal{C}$ took 83.59 minutes on average, and the value decreases if we increment k. The working space, however, is still considerable. We had memory peaks ranging from 3.03 GB to 4.3 GB, depending on the value for k (see Table 8.2).

The process of reconstructing the reads yielded a small number of ambiguous sequences in all the instances ($2,760$ sequences on average), and decreases with higher values of k, especially for values above 40 (see Table 8.2).

Chapter 9

Practical Locally Consistent Grammar

Most of the contributions of the previous chapters have been focused on algorithms and data structures for processing massive collections of sequencing reads in succinct space. In this chapter, we deviate a little from that topic to develop other ideas of independent interest, but with important applications in Genomics.

We show that the LMS-based parsing technique we used in Chapter 5 5 to obtain a compressed representation for reads yields a *locally consistent* grammar (Section 3.2.2). Given a string $S[1, n]$, we use this method to produce a locally consistent grammar that only produces S, and then use this representation as input to build the grammar index of Section 3.3.4. The result is a compressed representation for S that can locate the occurrences of a pattern $P[1, m]$ in $\mathcal{O}((m \log m + occ) \log G)$ time, where G is the size of the grammar (see Section 3.2.2). We believe this idea can serve in the future to develop a self-index for pangenomes that exploits DNA repetitions not just for compressing but also for boosting analyses.

We briefly describe the aspects of the grammar index that are relevant for this chapter. We refer the reader to Section 3.3.4 for more details on this data structure. A grammar self-index has two elements; a grammar $\mathcal{G}$ that only produces S and a grid to perform efficient pattern matching on S. The leaves in the grammar tree of $\mathcal{G}$ (Section 3.2.2) induce a partition over S. The occurrences of P spanning two or more phrases are called primary, while those fully contained within a phrase are secondary. When locating P, we use the grid to find its primary occurrences and the grammar tree of $\mathcal{G}$ to find its secondary occurrences. Looking for the primary occurrences requires us to generate a cut $P[1, q]P[q + 1, m]$, binary search for $P[1, q]$ in the row labels of the grid, and then binary search for $P[q + 1, m]$ in the column labels. The time for performing both binary searches adds up to $\mathcal{O}(m \log G)$, where G represents the number of columns in the grid. However, any of the m possible cuts of P can yield primary occurrences, so we need to try them all. This problem raises the time complexity for locating the primary occurrences to $\mathcal{O}(m^2 \log G)$[1].

Grammar indexes could be helpful for genomic applications as they use small space when the text is repetitive. However, the quadratic dependence on m for finding the primary occur-

[1]This time complexity does not consider the cost of extracting the information from the grid cells.

rences makes them less competitive than other methods such as the r-index (Section 3.3.3). Christiansen et al. [34] recently proposed an algorithm that generates a locally consistent grammar to solve this problem (see Section 3.2.2). They build the self-index of Section 3.3.4 using their grammar and show that they only require to try $\mathcal{O}(\log m)$ cuts of P in the grid.

Their idea consists of preprocessing the pattern at query time using the grammar algorithm. The local consistency of their algorithm produces P and its occurrences in S (if any) to have almost the same parse tree (same topology and node labels). This property reduces the number of cuts to try in the grid (see Sections 3.2.2 and 3.3.4 for more details on this idea).

The algorithm of Christiansen et al. differs from classical greedy algorithms such as RePair (Section 3.2.2) in which the creation of new nonterminals does not depend on the phrase frequencies but on the way the symbols are arranged in S. This non-greedy approach makes their grammar potentially large. They solve this problem by creating random permutations for the nonterminals as they build the grammar. This idea reduces the lengths of the right-hand sides of the rules, thus reducing the space usage. The drawback of this technique is that it requires storing the permutations for the self-index as they are necessary to parse P at query time. Although the space overhead of the permutations is proportional to G, they have a considerable impact in practice in the index size[2].

The algorithm we propose in this chapter also yields a locally consistent grammar. However, unlike Christiansen et al., we do not have to store permutations of the nonterminals to perform pattern matching. When querying P in the index, we use the lexicographical relations of its symbols to infer how its occurrences in S would be parsed. By not generating the random permutations, we are likely to obtain a larger representation than Christiansen et al. We offset this disadvantage by simplifying our grammar in a way that does not affect its locally consistent properties. We use our result to build the index of Section 3.3.4, thus requiring $\mathcal{O}(\log m)$ cuts for finding the primary occurrences of P. This self-index is smaller than that of Christiansen et al. and retains the same complexities.

Our method builds on the concept of induced suffix sorting (Section 3.3). The idea is a simpler version of the LMS-based parsing technique we developed for the grammar of Chapter 5. Using induced suffix sorting for grammar compression is not an original work of this chapter, nor is the idea of simplifying the grammar [142, 52]. Our contribution is to establish a new way of performing locally consistent parsing that has not been used before. Further, self-indexing on grammars that rely on this idea, and exploiting their properties to boost pattern matching, is an original work of this chapter.

Our experimental results showed that our grammar is comparable in size (G value) with that of Christiansen et al., but, as explained before, it does not require the permutations. Further experiments also showed that the grammar index built with our algorithm is larger than that built with RePair, but considerably faster as the pattern length grows.

The work on this chapter is a collaboration with the Ph.D. thesis of Alejandro Pacheco.

[2]This claim is true at least in the theoretical version of the data structure of Christiansen et al., where the permutations are stored explicitly. In practice, however, we can replace them with pseudorandom permutations and Karp-Rabin fingerprints, thus avoiding the space overhead.

The contribution of our thesis was to notice and prove that the parsing induced by the LMS substrings is indeed locally consistent. We also implemented the grammar algorithm described in Section 9.2.1 and the procedure that preprocesses the input pattern P at query time to find the $\mathcal{O}(\log m)$ cuts (Section 9.2.2). The source code of this grammar algorithm[3] is a modification of the implementation of the grammar compressor described in Chapter 5. The contribution of Alejandro Pacheco's thesis was to implement the grammar index of Section 3.3.4 and the locate function to find the occurrences of P in the grammar. He also ran the experiments described in Section 9.3.

Our results [50] will be presented at the *28th International Symposium on String Processing and Information Retrieval* (SPIRE'21).

9.1 Definitions

We consider the string $S[1, n]$ to be the input of our algorithm. We denote as $\mathcal{G}$ the grammar that only produces S and that we obtain with our algorithm. As in previous chapters, $\mathcal{R}$ is the set of rules and G is the grammar size. We define G as the sum of the lengths of the right-hand sides of $\mathcal{R}$. Our method uses the classifications L-type, S-type and LMS-type (Section 3.5.2) to create the nonterminals in $\mathcal{G}$. We also make use of the $\prec_{LMS}$ ordering described in Section 5.2. Let $S'[1, n']$ be any string of length n', and let $D[1, n']$ be a bit vector of equal size. If $S'[j]$ is L-type, then we set $D[j]$ to 0. When $S'[j]$ is S-type or LMS-type, we set $D[j]$ to 1. We refer to D as the *description* of S'.

9.2 A Grammar Self-Index based on LMS Parsing

9.2.1 LMS parsing

We define *LMS parsing* as the procedure of parsing S using its LMS substrings. The idea is similar to the method described in the SA-IS algorithm (Section 3.5.2). We compute the description of S, and define a phrase $S[i, i']$ for every consecutive pair of LMS-type positions $S[i-1]$ and $S[i']$. We refer those phrases as *LMS phrases*.

The LMS parsing is locally consistent. We demonstrate this claim by proving that equal substrings of S have the same descriptions, except possibly at their endpoints.

Lemma 12 The LMS parsing is locally consistent.

Proof. Let $S[a, b] = S[a', b']$ be two equal substrings. Let their suffixes of length $u \geq 1$ be equal-symbol runs, and symbols $S[b-u]$ and $S[b'-u]$ be different from $S[b-u+1]$ and $S[b'-u+1]$, respectively. The symbols within the same run have the same types, by definition. However, those types might differ if $S[b]$ and $S[b']$ are followed by different symbols. In particular, if $S[b-u+1]$ is L-type and $S[b'-u+1]$ is not, then $S[b'-u+1]$ can be LMS-type, and thus a phrase may end at $S[b'-u+1]$ and not at $S[b-u+1]$ (or vice versa).

[3] `https://github.com/ddiazdom/LPG/tree/LPG_grid`.

Instead, the positions $S[b-u]$ and $S[b'-u]$ preceding those runs will always have the same type because they are followed by the same symbol, $S[b-u+1] = S[b'-u+1]$. Furthermore, the equal substrings $S[a+1, b-u-1]$ and $S[a'+1, b'-u-1]$ will also have the same types because they are preceded and followed by the same symbols.

Finally, the types of $S[a]$ and $S[a']$ may differ because they may depend on the preceding symbol. Both or none can be L-type, but if they are not, then one may be LMS-type and the other be S-type, depending on the symbols at $S[a-1]$ and $S[a'-1]$. Therefore, one substring may have an LMS phrase ending at the first position and not the other.

To conclude, there can be at most one LMS phrase boundary appearing in each extreme of one of the substrings and not in the order. □

We then produce a locally consistent grammar $\mathcal{G}$ using several rounds of LMS parsing. In every round i, we create a dictionary $\mathcal{D}^i$ with all the distinct LMS phrases of S^i. Then, for every $F \in \mathcal{D}^i$, we create a new rule $X \to F$, where X is the number of rules in $\mathcal{R}$ built before round i plus the $\prec_{LMS}$ rank of F among the strings in $\mathcal{D}^i$. After generating the new rules, we create S^{i+1} by replacing the LMS phrases in S^i with their nonterminal symbols. If there are still repeated symbols in S^{i+1}, we perform another parsing round $i+1$ using S^{i+1} as input.

Note that this procedure is very similar to that of Christiansen et al. [34]. They randomly permute the alphabet and place a phrase boundary after every local minimum. In our LMS parsing, we place a phrase boundary after every LMS-type symbol, which is also a local minimum. The key difference is that Christiansen et al. need to store the symbol permutations of the parsing rounds to replicate the same parsing on the search pattern, whereas our parsing is given by the lexicographic order and thus can be applied on the pattern without further information.

To further reduce the grammar, we create a new rule $Y \to X^l$ for every maximal equal-symbol run X^l appearing on a right-hand side. The grammar tree represents rules $Y \to X^l$ as $Y \to X\,X^{l-1}$, where X^{l-1} is a special leaf. This unique cut is enough to detect the occurrences of any pattern, provided a special procedure is carried out to report the secondary occurrences inside X^{l-1} [34].

We also reduce space by replacing the nonterminals appearing once with the right-hand sides of their rules, unless they represent equal-symbol runs. The rules of those replaced symbols are then removed from $\mathcal{R}$.

9.2.2 Computing the cuts during the pattern matching

We use our grammar to build the self-index of Claude et al. [37] (Section 3.3.4), with the special provision for run-length rules. The main change of our version compared to the original one is the way we cut the pattern.

Let us call the *projection* of $P^i[p]$ the index $q \in [1, m]$ such that $P[p]$ is the rightmost leaf under the subtree rooted at $P^i[p]$ in P's parse tree; similarly with the projection of $\hat{P}^i[p]$.

Our procedure for finding the cuts for P is analogous to that of Christiansen et al. [34]. We start with an empty set Q and apply successive rounds of LMS parsing over P. In every round i, we insert into Q the projection of $P^i[1]$ and $\hat{P}^i[1]$. We then hash the distinct LMS phrases in P^i. We discard for the hashing, however, the prefix $P^i[1, a]$ where $P^i[a]$ is the leftmost LMS-type symbol, and the suffix $P^i[b..]$ where $P^i[b-1]$ is the rightmost LMS-type symbol. These elements can be incomplete phrases, so we do not use them for the next round. Still, we do record in Q the projection of $P^i[a]$ and the projection of $P^i[b-1]$ in P's parse tree. Additionally, when the rightmost equal-symbol run of P^i (last symbol in $\hat{P}^i$) has length $u > 1$ and $P^i[|P^i| - u]$ is L-type, we also insert the projection of $P^i[|P^i| - u + 1]$ into Q. We consider this position because if the rightmost run of P^i is S-type, then $P^i[|P^i| - u + 1]$ can be LMS-type if $P^i[|P^i| - u]$ is L-type. After scanning P^i, we sort the hashed phrases in $\prec_{LMS}$ order and create a new string P^{i+1} that replaces the phrases' occurrences with their $\prec_{LMS}$ orders. The new parse P^{i+1} is the input for the next round. The processing of P stops when there are no more LMS-type symbols in P^i.

The length of P^{i+1} is at most half that of P^i, so we scan $\mathcal{O}(m)$ symbols along the $\mathcal{O}(\log m)$ parsing rounds. On the other hand, we sort the distinct LMS subtrings of P^i in $\mathcal{O}(|P^i|)$ time [141], which adds up to $\mathcal{O}(m)$ total time. Therefore, the complete preprocessing of P requires $\mathcal{O}(m)$ time.

To find the primary occurrences, we binary search the cut $P[1, q]P[q + 1, m]$ associated to every $q \in Q$. Since $|Q| \in O(\log m)$, the total time to look for the primary occurrences is $O(|Q| m \log G) \subseteq \mathcal{O}(m \log m \log G)$, plus $\mathcal{O}(|Q| \log G) \subseteq \mathcal{O}(\log m \log G)$ for the geometric searches, plus $\mathcal{O}(occ \log G)$ to extract the grid points. Our final result borrows the space figures of Claude et al. [37].

Theorem 12 Let our grammar, built for $S[1, n]$, be of size G. Then our index uses $G \log n + (2+\epsilon)G \log G$ bits of space, for any constant $\epsilon > 0$, and finds all the occ occurrences of $P[1, m]$ in time $\mathcal{O}((m \log m + occ) \log G)$.

We note that, still within $\mathcal{O}(G)$ space, we can use Patricia trees to speed up the binary searches, obtaining $\mathcal{O}(m \log m + (\log m + occ) \log G)$ time.

9.3 Experiments

We implemented our version of the grammar index in `C++` on top of the `SDSL-lite` library [76]. The source code is available at `https://github.com/ddiazdom/LPG/tree/LPG_grid`. This software combines source code of the grammar compressor[4] of Section 5 and the grammar index[5] of Claude et al. [37]. We generated two versions of our index. The regular version (`lms-ind`) implements the wavelet tree of the grid data structure using plain bit vectors. The second variant (`lms-ind-rrr`) encodes the wavelet tree using the data structure for compressed bit vectors of Section 2.2.1. We compared our software against the state-of-the-art self-indexes for repetitive collections:

[4] `https://bitbucket.org/DiegoDiazDominguez/lms_grammar/src/bwt_imp2`.

[5] `https://github.com/apachecom/grammar_improved_index`.

- `r-ind`[6]: The run-length compressed FM-index described in Section 3.3.3. It uses $\mathcal{O}(r \log n)$ bits, where r is the number of runs in the text's BWT, and supports locate within that space.
- `lz-ind`[7]: A self-index based on Lempel-Ziv [101] that guarantees $\mathcal{O}(z \log z)$ bits of space over a Lempel-Ziv parse of z phrases. We also included the variant that uses LZ-end parsing (`lz-end-ind`) [100].
- `slp-ind` : An optimized implementation of the grammar index of Claude et al. 2012 [36]. This version speeds up the binary searches by storing q-grams[8] of the prefixes to which the nonterminals expand (grid labels). We used three q-gram values in our experiments; 4,8, and 16. We refer to these variants as `slp-ind4`, `slp-ind8` and `slp-ind16`, respectively.
- `g-ind`: The grammar index of Claude et al [37] described in Section 3.3.4. The first variant of this index (`g-ind-bs`) uses binary searches over the grid labels to find the primary occurrences of P. The second variant (`g-ind-pt`) speeds up the search by maintaining two Patricia trees, one for a subset of uniformly sampled column labels in the grid and the other for a subset of uniformly sampled row labels. We used three sampling rates; 1/4, 1/16, and 1/64. We refer to these variants as `g-ind-pt4`, `g-ind-pt16`, and `gt-ind-pt64`, respectively.

We used five data sets of the *Pizza&Chilli*[9] corpus for the experiments. The characteristics of these datasets are shown in Table 9.1. We assessed the compression ratio and the running time for locating patterns. We extracted random substrings from the datasets and then we searched them back with the different indexes. The length of these patterns ranged from 10 to 100 characters.

We also compared our grammar algorithm against RePair and the method of Christiansen et al. [34]. The metric we used for the comparison was the grammar size (G value). The algorithm of Christiansen et al. has no formal implementation, so we produced one ourselves.

All the experiments were carried out on a machine with eight Intel(R) Xeon(R) CPU E5-2407 processors at 2.40 GHz and 250 GB RAM. We compiled our source code using full compiler optimizations and we do not use multi threading.

9.4 Results and Discussion

The grammars produced with our method were, on average, 4.2 times bigger than the RePair grammars (see columns 4 and 5 of Table 9.1). This considerable difference is expected as our grammar algorithm prioritizes consistency over compression. By further processing the grammars produced with our method (see Section 9.2), we reduced their sizes by 41% on average. However, their final sizes were still far from those of RePair; they were 1.82 times bigger on average. Interestingly, the sizes of our post-processed grammars were similar to those of Christiansen et al. (columns 6 and 7 of Table 9.1), even though we are not using

[6] `https://github.com/nicolaprezza/r-index`.

[7] `https://github.com/migumar2/uiHRDC/tree/master/uiHRDC/self-indexes/LZ`.

[8] They are also referred to as kmers.

[9] `http://pizzachili.dcc.uchile.cl`.

Dataset	n	σ	RePair	LMS	LMS post	LC
para	429,265,758	5	5,344,480	22,787,047	8,933,303	8,888,002
cere	461,286,644	5	4,069,450	37,426,507	6,802,801	4,069,450
influenza	154,808,555	15	1,957,370	4,259,746	3,304,035	4,477,322
einstein.en	467,626,544	139	212,903	643,338	427,142	601,755
kernel	257,961,616	162	1,374,650	3,769,839	2,870,350	3,795,801

Table 9.1: Datasets. The second and third columns are the number of symbols and alphabet, respectively. The rest of the columns are grammar sizes (value for G) obtained with different grammar algorithms. The column for RePair considers the postprocessing described in Section 3.3.4. LMS is the grammar obtained with LMS parsing and LMS post is grammar resulted from the postprocessing described in Section 9.2. The last column (LC) refers to the grammar of Christiansen et al. 2020.

random permutations. It is important to note that it is unknown how to further simplify the grammar of Christiansen et al. without losing local consistency.

Figure 9.1 shows the trade-offs between index space usage and time for locating patterns of length 100. The results varied widely depending on the dataset. For instance, in *cere* and *para*, the index that used the most space was `r-ind` (1.93 and 2.76 bps, respectively), but it was also the fastest (0.34 and 0.37 μsecs). The second-largest index was `lms-ind` (1.32 and 1.89 bps), and the second-fastest after `r-ind`. In both datasets, the variant `lms-ind-rrr` reduced the space usage and stayed competitive for locating, but in the other datasets, `lms-ind-rrr` reduced the space at the cost of becoming slower. The smallest representation in *cere* and *para* was `lz-ind` (0.49 and 0.70 bps), but it was the slowest at locating (10.2 and 15.5 μsecs). In *einstein.en*, `lms-ind` was the biggest index (0.076 bps), even bigger than `r-ind`, which remained the fastest. However, `lms-ind` was the fastest dictionary-based data structure. The `lms-ind-rrr` variant reduced the space, but it did not outperform `r-ind`.

Things went differently with *influenza* and *kernel*. The Lempel-Ziv data structures (`lz-ind` and `lz-end-ind`) were competitive with `r-ind` for locating (14.16 and 18.05 μsecs versus 5.89 μsecs, respectively). Nevertheless, they used less space. This result is unexpected as *influenza* and *kernel* are not as repetitive as *einstein.en*. Our index performed poorly in these datasets. Both variants (`lms-ind` and `lms-ind-rrr`) were the biggest dictionary-based data structures, and they were not the fastest ones. However, they were competitive with `r-ind` in terms of space, with `lms-ind-rrr` using less space than `r-ind` in *kernel* (0.81 bps versus 0.89 bps, respectively), although they were significantly slower.

Figure 9.2 shows the performance of the indexes for the locate operation using different pattern lengths (from 100 to 800). In *para* and *cere*, `lms-ind` greatly outperformed the other dictionary-based indexes as the pattern length increased. This was not the case in *einstein.en* and *kernel*, where the performance of `lms-ind-rrr` was not different from that of `slp-ind`. We also noted that in those datasets, the performance of `lz-ind` was very close to that of `r-ind`. Interestingly, the performance of `r-ind` remained steady as the pattern length increased.

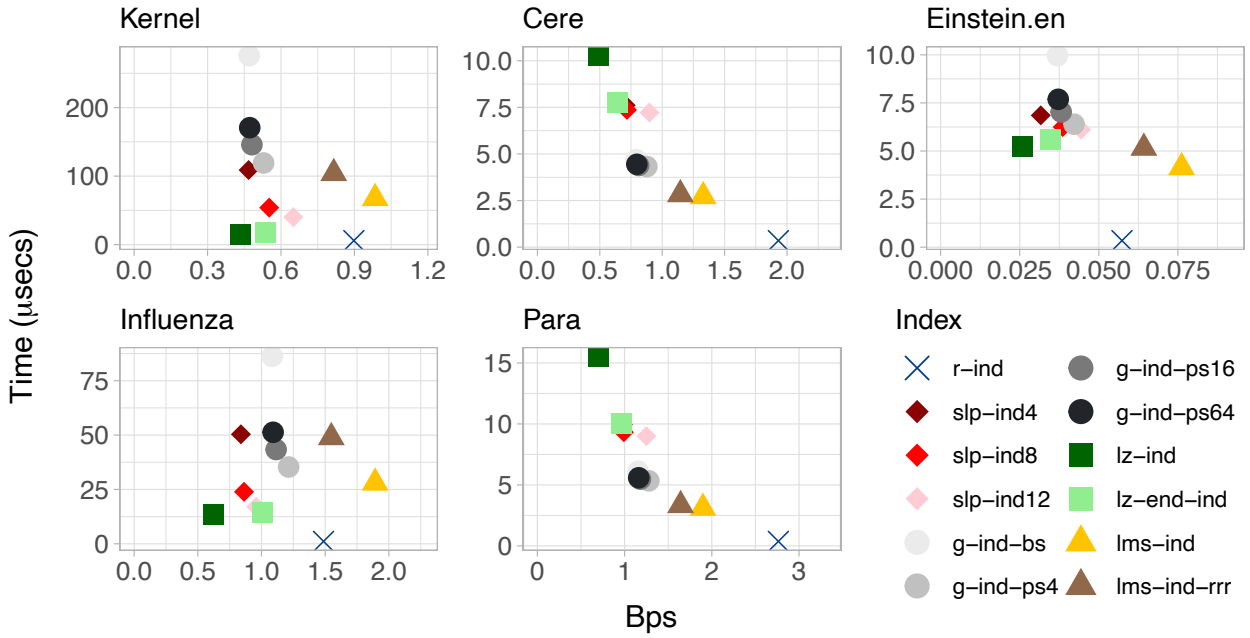

Figure 9.1: Time-space tradeoffs for locating 1000 random pattern of length 100 on different collections and indexes. The time (y-axis) is given in μsecs per occurrence and the index space (x-axis) in bits per symbol (bps).

9.5 Locally Consistent Grammars and Pangenomes

A relevant application (not explored yet) of locally consistent grammars is indexing pangenomic collections (Section 4.5). Putting aside the fact that we could achieve high compression ratios as pangenomes are repetitive, they could also help develop efficient methods for aligning third-generation reads (Section 4.2) in compressed space. These strings are long (several thousands of characters) and more prone to sequencing errors than regular NGS reads. Therefore, aligning them to a pangenome requires a self-index with efficient support for approximate pattern matching of long strings. The r-index is the state-of-the-art compressed index for pangenomes, but most of its variations are tailored to perform exact pattern matching of short strings.

In the previous sections, we saw that locally consistent parsing improves the time complexity when locating a pattern in the grammar self-index of Section 3.3.4, thus enabling the alignment of long strings. However, the benefits of this technique are not limited to the grammar index. We believe that locally consistent parsing could also serve as a base to develop other indexes that support approximate string matching. We now briefly sketch our idea.

Assume S is a concatenated pangenome, and we require to search for the occurrences of a (possibly) long read $P[1, m]$, allowing a few mismatches. We use locally consistent parsing to detect long substrings of P that have exact matches in S. We use the matched substrings of S as anchors for possible inexact alignments, which we later extend using, for instance, dynamic programming or colinear chaining. This is the classic method most aligners use to map DNA sequences to genomic databases (see Section 4.4 for examples of these ideas). The

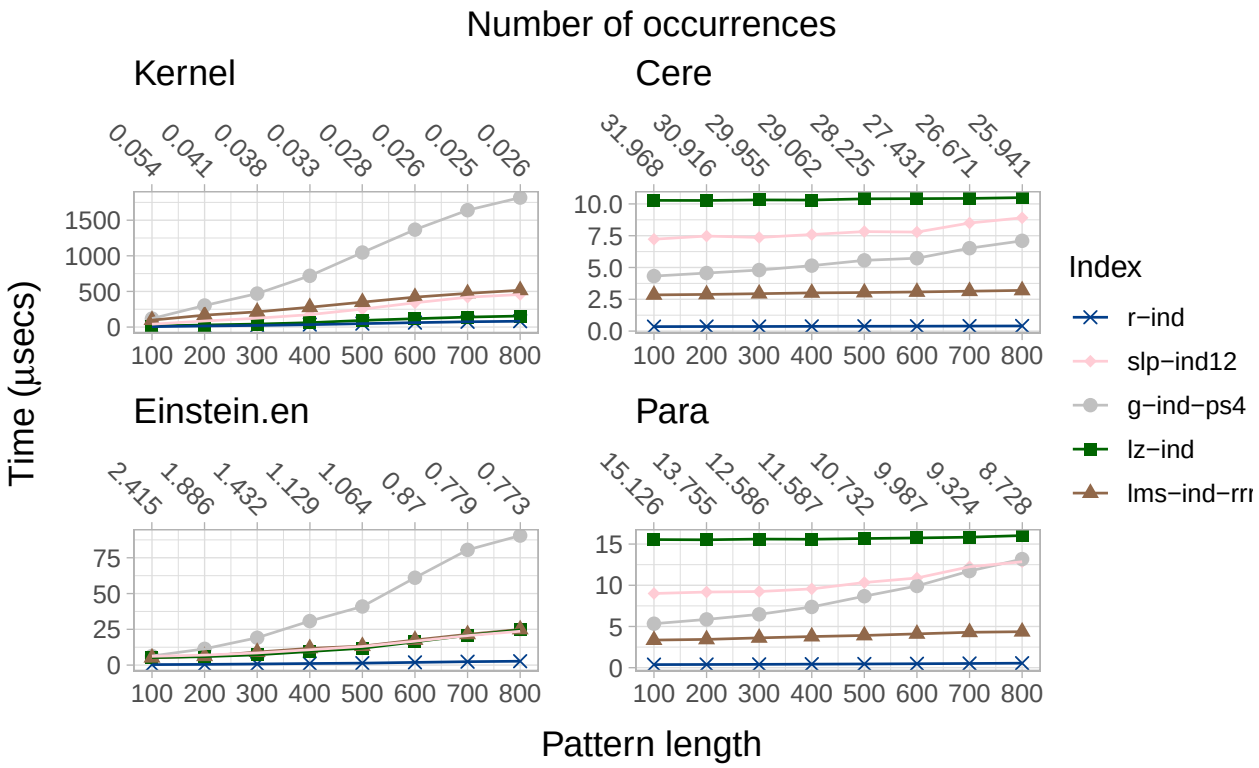

Figure 9.2: Locating time for increasing pattern lengths. The time is given in μsecs per occurrence. The upper x-axes indicate the number of occurrences (in millions) searched in every combination of document and pattern length.

only difference here is that we are using locally consistent parsing to compute the anchors.

We find the substrings of P with exact matches in S as follows. Suppose we have a locally consistent grammar $\mathcal{G}$ that only produces S and an index for the rules of $\mathcal{G}$. We preprocess P using the same algorithm we used for $\mathcal{G}$. In every parsing round i, we query the index to find the distinct phrases of P^i appearing in the right-hand sides of $\mathcal{G}$'s rules. We extract their associated nonterminals to build the next parse P^{i+1}. In doing so, we put wildcards in P^{i+1} to represent the phrases that were not found in $\mathcal{G}$. In the subsequent parsing round, we only query complete phrases that do not contain wildcards. Those substrings of P^{i+1} with incomplete phrases are replaced in the next parse with wildcard symbols. The preprocessing finishes when P^i has no more valid phrases. The substrings of P^h (the last parse) containing non-wildcard symbols recursively expand to substrings in P that have exact matches in S. The positions in S for those matches will be the anchors for the inexact alignments.

It is not yet clear how to locate the positions in S for the anchors in compressed space. We are also unclear on extending the alignments in compressed space once we find the anchors. We believe that encoding $\mathcal{G}$ using the Wheeler framework (Section 3.4) could help us perform these tasks.

Chapter 10

Conclusion and Further Work

We addressed the problem of manipulating massive collections of genomic data in succinct or compressed space. Our contributions were focused on, but not limited to, processing large datasets of short strings called sequencing reads. These are the raw form in which DNA sequences are delivered and require further processing on the computer to obtain the final product (i.e., a genome). We implemented different compact data structures and compression-aware algorithms for reads that exploit the repetitive patterns of DNA. Our experiments demonstrated that these methods reduce space usage and improve the results of the analyses. In this regard, it is important to note that, unlike complete genomes, the types of read collections we considered in this thesis have short repetitive patterns. We conclude thus that it is possible, in practice, to use the text redundancies to lower the computational costs of manipulating large datasets, even in not so highly repetitive scenarios.

10.1 Summary of contributions

- In Chapter 5, we described a practical algorithm called LMSg to compress a read collection $\mathcal{S}$. This method reduces the space usage by constructing a context-free grammar $\mathcal{G}$ that only produces $\mathcal{S}$. The particular construction of $\mathcal{G}$ allows us to compute the eBWT of $\mathcal{S}$ without fully decompressing the text. On the other hand, if we augment $\mathcal{G}$ with extra data structures, we can also support random access to individual reads. We believe these features are suitable for genomic applications where the reads are processed once and then discarded from downstream analyses. An example of this situation is, for instance, the assembly of an individual genome.

  The experimental results demonstrated that our method is a practical alternative to the state-of-the-art compressors when $\mathcal{S}$ is massive. `LPG`, the software that implements LMSg, was the fastest method in the benchmarks. Its memory peak was about 58% of the input size. This value is considerable compared to the negligible amount of memory that the popular tool `7-zip` required. However, it was far less than what the RePair variation for large collections (BigRePair) used. `LPG` also used less working memory than `egap`, the software we used for building the BCR BWT of $\mathcal{S}$. The experimental results also showed that our compressed representation for $\mathcal{S}$ can support random access without increasing the space too much. The extra data structures we required for this

task increased the space of $\mathcal{G}$'s encoding by about 17%. In exchange, we could extract random strings of length 152 in about 100 μsecs. This result was competitive with the other tested methods.

- In Chapter 6, we described infBWT, an algorithm for building from $\mathcal{G}$ the eBWT of $\mathcal{S}$. We also described a variation of the eBWT that we can use to encode extra information about the reads. This variation offers the following functionality. Let $\mathcal{S}^*$ be a string collection with the reads of $\mathcal{S}$ along with their reverse complements, and let L be the eBWT of $\mathcal{S}^*$. Given that we know a position in L encoding a symbol in $S_i \in \mathcal{S}^*$, we can access the other BWT positions spelling the DNA reverse complement of S_i or the BWT positions spelling its mate if $\mathcal{S}$ is a paired-end collection. Supporting this functionality does not require extra space on top of L.

  The experimental results showed that infBWT was not the fastest method, nor was the most space-efficient. On the other hand, it was not the method that required the most time or space either. However, it was the only one whose performance improved as the repetitiveness of the input increased (i.e., as we appended to $\mathcal{S}$ more read collections from other individuals of the same species). The number of bytes and μsecs per input symbol our algorithm took decreased with every new collection we appended to $\mathcal{S}$. This result implies that infBWT is indeed exploiting the repetitions. We believe that, if we continue appending more collections to $\mathcal{S}$, infBWT will eventually become the most efficient method.

- In Chapter 7, we presented rBOSS; a BWT-based index for short reads that enables the navigation of the reads' layout (Section 4.3) in succinct space. This index augments the BOSS representation for dBGs of Bowe et al. [24] with a new compact data structure that speeds up the computation of overlaps between strings. We call this new structure the overlap tree. We showed that the use of the overlap tree is not limited to dBGs. We can also combine it with the eBWT of $\mathcal{S}$ to produce a self-index that computes overlaps between reads. We experimentally compared rBOSS against other succinct representation for variable-order dBGs called VO-BOSS. Our results showed that rBOSS uses, on average, 20% less space than VO-BOSS and it is faster at reporting string overlaps.

  The framework of rBOSS supports new string queries tailored for genomic analyses. To our knowledge, these queries had not been proposed before in literature. By combining these new functions with the features of BOSS we can navigate the dBG and take more informed decisions. In particular, we can better estimate if the edges we visit in the traversal represents real genome information or sequencing noise. We implemented a simple genome assembler (Section 4.3) on top of rBOSS and showed that we produce longer genome segments (a.k.a contigs) than regular approaches based on dBGs of fixed order.

- In Chapter 8 we presented an alternative succinct index for short reads. We build a dBG G from $\mathcal{S}$ and assign a specific color c_i to every string $S_i \in \mathcal{S}$. We then store c_i in the path of G that spells S_i. The idea is to use the colors to disentangle the reads from G. We reduce the space usage of the colors in the index by using an observation made by Alipanahi et al [1]; it is safe to assign the same color to those reads that do not share kmers. We further develop their idea showing that (i) reads spelled in G by walks containing cycles cannot be disentangled, (ii) it is not safe to assign the same color to those reads that do not share kmers if their paths in G are connected by spurious edges, and (iii) it is not necessary to color all the nodes in G to disentangle the reads. We

developed a new greedy coloring coloring for G that considers our observations. We encode the resulting color matrix using compact data structures and the topology of G using BOSS. We also implemented two practical algorithms on top of our version of the colored dBG. The first one, buildseqs, extracts the reads from G, while the second, contigassm, assembles contigs from G taking into consideration the path colors.

Our experimental results show that, on average, the percentage of nodes in G that need to be colored is about 12.4%, the space usage of the whole index is about half the space of $\mathcal{S}$, and more than 99% of the reads can be spelled from the colored graph.

- In Chapter 9, we demonstrated that the parsing technique we used in Chapter 5 to compress $\mathcal{S}$ is locally consistent. We exploited this fact to develop a practical algorithm that, given an input string S, produces a locally consistent grammar $\mathcal{G}'$ whose language only contains the string S. We use $\mathcal{G}'$ as input to build the grammar index of Claude et al [37] (Section 3.3.4). Similarly to Christiansen et al [34], we combine the machinery of the grammar index with the local consistency of $\mathcal{G}'$ to locate the *occ* occurrences in S of a pattern $P[1, m]$ in $\mathcal{O}((m \log m + occ) \log G)$ time, where G is the size of $\mathcal{G}'$. In the regular index of Claude et al., the same operation is implemented in $\mathcal{O}((m^2 + occ) \log G)$ time. Our advantage compared to Christiansen et al. is that we do not require to store extra information to retain the local consistency of $\mathcal{G}'$ in the index.

  Our experimental results showed that our method is a practical alternative to the technique of Christiansen et al., as we obtain a locally consistent grammar of comparable size without storing the symbol permutations. The resulting self-index is thus not much larger than the other popular dictionary-based indexes but generally faster at locating patterns, especially long ones.

10.2 Further Work

The techniques we developed in this thesis mainly focused on processing NGS reads (short strings) as these datasets were the most widespread four years ago when we started our research. However, the rapid advances in sequencing technologies make things go fast in Genomics. Third-generation reads (long strings) are nowadays becoming equally or more relevant than NGS data. Because of this fact, our future research direction will be to adapt the ideas we developed in this thesis for manipulating third-generation datasets. We are also interested in adapting our algorithmic framework for indexing pangenomes, as this topic is also relevant for the Bioinformatics community. We now discuss the future research directions and open problems in our different contributions.

- The most relevant improvement for LMSg (Chapter 5) is to reduce the size of the grammar $\mathcal{G}$ without losing its features for constructing the eBWT of $\mathcal{S}$. A possible solution for this problem was recently pointed out to us by Dominik Köppl at DCC'21. He suggested that we could reduce the number of nonterminals in $\mathcal{G}$ by considering the strings in $\mathcal{S}$ to be circular. The purpose is to remove the extra \$ symbol we have to append to every string in $\mathcal{S}$ when constructing the grammar. The rationale of his idea is that by removing one symbol from the alphabet of $\mathcal{S}$ (\$ in this case), the number of distinct nonterminals we can generate from the reads also decreases. The only caveat is that we require to add a bit vector of size proportional to C, the compressed version of $\mathcal{S}$ in $\mathcal{G}$, to mark every symbol in $C[i]$ representing the end of a string.

We would also like to find theoretical bounds for LMSg. Recall that our method builds on the ISS idea of Section 3.5.2. In this algorithm, the size of the parse S^{i+1} is always at most half the size of the previous parse S^{i+1}, so the number of symbols the algorithm processes is proportional to the size of the input text. However, LMSg can transfer symbols from one parsing round to the next one, meaning that the aforementioned property of ISS no longer holds for our method. A simple solution would be to run the algorithm of Nunes et al. [142] and then simplify the grammar. This idea would make LMSg linear-time. However, it is not clear for us if we can use the resulting grammar to compute the eBWT of $\mathcal{S}$.

Another possible application of LMSg is the compression of pangenomes. The idea is interesting, not just because we can achieve better compression ratios than with reads but also because we can exploit the long DNA repetitions to boost the computation of the pangenome's eBWT. Several pangenomic self-indexes based on the r-index are being actively developed nowadays, and the construction of the collection's BWT is a crucial topic in all of them. Our current implementation of LMSg speeds up the grammar construction by exploiting the fact that the reads in $\mathcal{S}$ are short. We need to modify our source code to make it compatible with the cases when the strings in $\mathcal{S}$ are arbitrarily long.

- Although infBWT (Chapter 6) increases its performance as $\mathcal{S}$ becomes repetitive, it is still slow compared to other alternatives for constructing the BCR BWT that run in semi-external mode. A possible solution to overcome this problem is to avoid re-constructing the alphabet of symbols for every parse S^i (see Section 6.4). Ideally, one would like to have a grammar construction with the following property. Let X and X be two nonterminals. If X < Y, then exp(X) $<_{LMS}$ exp(Y). We conjecture that this property would enable the direct use of nonterminals for computing the eBWT, thus avoiding the alphabet reconstruction. However, this idea makes sense only if $\mathcal{G}$ is balanced and the nonterminals of every parse tree level are disjoint, which is not our case. We have some preliminary ideas about renaming the symbols in $\mathcal{G}$, so we do not require the alphabet reconstructions, but they require modifying big portions of LMSg and infBWT.

  One of the main arguments we gave in this thesis for building the eBWT from $\mathcal{G}$ was the indexing of $\mathcal{S}$. However, most self-indexes for reads require other data structures as well. We plan to extend the idea of infBWT to compute the LCP array while producing the eBWT. We also wonder if it is possible to obtain the dBG with order k of $\mathcal{S}$ by modifying infBWT. We also plan to address this challenge in the future.

- One of the most important topics unfinished in rBOSS (Chapter 7) is its implementation on top of our variation of the eBWT (Section 6.1). There are several technical problems we need to tackle first. First, we need an efficient method to compute the overlap tree from the generalized suffix tree of $\mathcal{S}$. We intend to modify infBWT to get the overlap tree while it infers the eBWT L of $\mathcal{S}$, but it is not clear to us yet how to carry out this task. Another problem is that rBOSS considers the reverse complements of $\mathcal{S}$, but the array L we obtain with infBWT does not have these symbols as $\mathcal{G}$ does not store the reverse complements. We can tackle this problem by modifying infBWT to produce an alternative eBWT L^{rc} for the reverse complements, and then merge L and L^{rc} into one array. This task, however, is not trivial, and (again) we are not clear about how to proceed. Besides, we still need to find a way to produce the overlap tree for the final

eBWT.

Another relevant feature that was not implemented in rBOSS is the computation of inexact overlaps. This functionality is not difficult to implement, but it will affect the performance of layout. However, it is necessary as it is a more realistic model for the reads.

We also plan to improve the genome assembler we created on top of rBOSS. Although the contigs we obtained with our assembler were longer than those obtained with `bcalm`, they overlap. The problem arises because maximal paths in rBOSS are not disjoint. A simple solution is to reassemble the contigs recursively. We build another rBOSS instance with a higher maximum order k using the contigs as input, and then repeat the same assembly process. We keep doing this idea until the contigs no longer overlap. This approach is similar to the iterative dBG-based genome assembler of Peng et al. [152] and Pevzner et al. [5].

- We believe that a more careful algorithm for constructing the colored dBG (Chapter 8) is still necessary to reduce the memory peaks. Further compaction of the color matrix can be achieved by using more elaborated compression techniques. However, this extra compression can increase the construction time of the colored dBG and produce a considerable slowdown in the algorithms that work on top of it for extracting information from the reads. Comparison of our results with other similar data structures is difficult at the moment. Most of the indexes based on colored dBGs were not designed to handle huge sets of colors like ours. On the other hand, the greedy recoloring of Alipanahi et al. [1] does not scale well and it needs extra information for reconstructing the reads.
- The performance for locating patterns of the grammar index we obtained in Chapter 9 was better than in the previous variations of this index. However, we are still not competitive with the widely used r-index, at least not for the time for the locate operation. On the other hand, the relatively big grammar that results from our algorithm and the heavy machinery required to implement the grammar index produces our final data structure to be big compared to other dictionary-based self-indexes. These drawbacks do not mean that the ideas we developed are impractical. We believe that we can increase our performance in both time and space if we encode our locally consistent grammar using the Wheeler Graph framework (Section 3.4). The grammar tree would be replaced by a BWT obtained from the right-hand sides of the grammar rules, and the grid functionality would be simulated using the LF and backwardsearch operations.

  Another aspect we would like to address in the future is the use of local consistency to support approximate mismatches in the grammar-based self-index. The idea we gave in Section 9.5 is just a sketch and needs to be further developed. Enabling approximate mismatches is an important feature we must develop to design practical pangenomic representations.

Bibliography

[1] Bahar Alipanahi, Alan Kuhnle, and Christina Boucher. Recoloring the colored de Bruijn graph. In *Proc. 25th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 1–11, 2018.

[2] Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An efficient, scalable and exact representation of high-dimensional color information enabled via de Bruijn graph search. In *Proc. 24th International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 1–18, 2019.

[3] Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. Rainbowfish: a succinct colored de Bruijn graph representation. In *Proc. 17th International Workshop on Algorithms in Bioinformatics (WABI)*, 2017. Article 18.

[4] Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 2018.

[5] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey Gurevich, Mikhail Dvorkin, Alexander Kulikov, Valery Lesin, Sergey Nikolenko, Son Pham, Andrey Prjibelski, Alexey Pyshkin, Alexander Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max Alekseyev, and Pavel Pevzner. SPAdes: A New Genome Assembly Algorithm and Its Applications to Single–Cell Sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

[6] Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Picatkowski. Constructing the bijective BWT. In *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2021. Article 4.

[7] Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013.

[8] Djamal Belazzougui and Fabio Cunial. Representing the suffix tree with the CDAWG. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2017. Article 7.

[9] Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition–aware data structures. In *Proc. 26th Annual Symposium*

on Combinatorial Pattern Matching (CPM), pages 26–39, 2015.

[10] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear–Time String Indexing and Analysis in Small Space. *ACM Transaction on Algorithms*, 16(2), 2020.

[11] Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):1–21, 2015.

[12] Jason Bentley, Daniel Gibney, and Sharma V. Thankachan. On the Complexity of BWT–Runs Minimization via Alphabet Reordering. In *Proc. 28th Annual European Symposium on Algorithms (ESA)*, volume 173, 2020. Article 15.

[13] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James Drake, Jane Landolin, and Adam Phillippy. Assembling large genomes with single-molecule sequencing and locality–sensitive hashing. *Nature Biotechnology*, 33(6):623–630, 2015.

[14] Paul Bieganski, John Riedl, John V. Carlis, and Ernest F. Retzel. Generalized suffix trees for biological sequence data: applications and implementation. In *Proc. 27th Hawai International Conference on System Sciences (HICSS)*, pages 35–44, 1994.

[15] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar–compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[16] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random Access to Grammar–Compressed Strings and Trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[17] Pacific Biosciences. PacBio. Available online at `https://www.pacb.com`, last accessed: 2021-02-09.

[18] Burton Bloom. Space/time trade–offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[19] Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[20] Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Computing the multi–string BWT and LCP array in external memory. *Theoretical Computer Science*, 862:42–58, 2021.

[21] Christina Boucher, Alexander Bowe, Travis Gagie, Simon Puglisi, and Kunihiko Sadakane. Variable–Order de Bruijn Graphs. In *Proc. 25th Data Compression Conference (DCC)*, pages 383–392, 2015.

[22] Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix–free parsing for building big BWTs. *Algorithms for Molecular*

Biology, 14, 2019. Article 13.

[23] Christina Boucher, Travis Gagie, I. Tomohiro, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: Streamed Matching Statistics with Multi-Genome References. In *Proc. 31st Data Compression Conference (DCC)*, pages 193–202, 2021.

[24] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn Graphs. In *Proc. 12th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 225–235, 2012.

[25] Nicolas L. Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near–optimal probabilistic RNA–seq quantification. *Nature Biotechnology*, 34(5):525–527, 2016.

[26] Andrei Broder. On the resemblance and containment of documents. In *Proc. 1st Compression and Complexity of Sequences (SEQUENCES)*, pages 21–29, 1997.

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

[28] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A. Shlyakhter, Matthew K. Belmonte, Eric S. Lander, Chad Nusbaum, and David B. Jaffe. ALLPATHS: de novo assembly of whole–genome shotgun microreads. *Genome Research*, 18(5):810–820, 2008.

[29] Pedro Celis, Per-Ake Larson, and J. Ian Munro. Robin hood hashing. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 281–288, 1985.

[30] The Human Pangenome Reference Center. The Human Pangenome Project. Available online at `https://humanpangenome.org`, last accessed: 2021-02-09.

[31] Timothy Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th Annual Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

[32] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[33] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.

[34] Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal–Time Dictionary-Compressed Indexes. *ACM Transactions on Algorithms*, 17(1), 2021. Article 8.

[35] Francisco Claude and Gonzalo Navarro. Self–Indexed Grammar–Based Compression. *Fundamenta Informaticae*, 111(3):313–337, 2011.

[36] Francisco Claude and Gonzalo Navarro. Improved grammar–based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 180–192, 2012.

[37] Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar–compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021.

[38] Dustin Cobas, Travis Gagie, and Gonzalo Navarro. A Fast and Small Subsampled R–index. In *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2021. Article 13.

[39] Phillip Compeau, Pavel Pevzner, and Glenn Tesler. How to apply de Bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, 2011.

[40] 1000 Genomas Chile Consortium. Chile secuencia a Chile: 1000 Genomas Chile. Available online at `http://www.1000genomas.cl`, last accessed: 2021-02-08.

[41] 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 2012.

[42] Earth Biogenome Project Consortium. Earth Biogenome Project. Available online at `https://www.earthbiogenome.org`, last accessed: 2021-02-08.

[43] Joshimar Cordova and Gonzalo Navarro. Simple and efficient fully–functional succinct trees. *Theoretical Computer Science*, 656:135–145, 2016.

[44] Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large–scale compression of genomic sequence databases with the Burrows–Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012.

[45] Agnieszka Danek, Sebastian Deorowicz, and Szymon Grabowski. Indexes of large genome collections on a PC. *PloS one*, 9(10):e109384, 2014.

[46] Nicolaas de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie Wetenschappen*, 49(49):758–764, 1946.

[47] Arthur Delcher, Simon Kasif, Robert Fleischmann, Jeremy Peterson, Owen White, and Steven Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[48] Gennady Denisov, Brian Walenz, Aaron Halpern, Jason Miller, Nelson Axelrod, Samuel Levy, and Granger Sutton. Consensus generation and variant detection by Celera Assembler. *Bioinformatics*, 24(8):1035–1040, 2008.

[49] Diego Díaz-Domínguez. An Index for Sequencing Reads Based on the Colored de Bruijn Graph. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 304–321, 2019.

[50] Diego Díaz-Domínguez. An LMS-based Grammar Self-index with Local Consistency Properties. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 304–321, 2021.

[51] Diego Díaz-Domínguez, Travis Gagie, and Gonzalo Navarro. Simulating the DNA overlap graph in succinct space. In *Proc. 30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2019. Article 26.

[52] Diego Díaz-Domínguez and Gonzalo Navarro. A grammar compressor for collections of reads with applications to the construction of the BWT. In *Proc. 31st Data Compression Conference (DCC)*, pages 83–92, 2021.

[53] Dirk D. Dolle, Zhicheng Liu, Matthew Cotten, Jared T. Simpson, Zamin Iqbal, Richard Durbin, Shane A. McCarthy, and Thomas M. Keane. Using reference–free compressed data structures to analyze sequencing reads from thousands of human genomes. *Genome Research*, 27(2):300–309, 2017.

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

[55] Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14:6, 2019.

[56] Jordan M. Eizenga, Adam M. Novak, Jonas A. Sibbesen, Simon Heumos, Ali Ghaffaari, Glenn Hickey, Xian Chang, Josiah D. Seaman, Robin Rounthwaite, Jana Ebler, Mikko Rautiainen, Shilpa Garg, Benedict Paten, Tobias Marschall, Jouni Sirén, and Erik Garrison. Pangenome Graphs. *Annual Review of Genomics and Human Genetics*, 21:139–162, 2020.

[57] Peter Elias. Efficient Storage and Retrieval by Content and Address of Static Files. *Journal of the ACM*, 21(2):246–260, 1974.

[58] DELL EMC. De Novo Assembly with SPAdes assembler. Technical report, DELL Inc., 2017.

[59] DELL EMC. Genome Assembly on Deep Sequencing data with SOAPdenovo2. Technical report, DELL Inc., 2018.

[60] Genomics England. The 100,000 Genomes Project. Available online at `https://www.genomicsengland.co.uk/about-genomics-england/the-100000-genomes-project`, last accessed: 2021-02-08.

[61] Robert Fano. *On the number of bits required to implement an associative memory.* Computer Structures Group, Project MAC, Massachusetts, 1971.

[62] Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Marinella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.

[63] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and Senthilmurugan Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1), 2009. Article 4.

[64] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

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

[66] Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1), 2010. Article 10.

[67] U.S. Food and Drug Administration. Precision Medicine. Available online at `https://www.fda.gov/medical-devices/in-vitro-diagnostics/precision-medicine`, last accessed on 2021-02-09.

[68] Travis Gagie, Garance Gourdel, and Giovanni Manzini. Compressing and indexing aligned readsets. In *Proc. 21st International Workshop on Algorithms in Bioinformatics (WABI)*, 2021. Article 13.

[69] Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017.

[70] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT–Runs Bounded Space. *Journal of the ACM*, 67(1), 2020. Article 2.

[71] Travis Gagie and Simon J. Puglisi. Searching and indexing genomic databases via kernelization. *Frontiers in Bioengineering and Biotechnology*, 3:12, 2015.

[72] Travis Gagie, I. Tomohiro, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling RePair with rsync. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 35–44, 2019.

[73] Moses Ganardi, Artur Jeż, and Markus Lohrey. Balancing straight-line programs. In *Proc. 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1169–1183, 2019.

[74] Erik Garrison, Jouni Sirén, Adam M. Novak, Glenn Hickey, Jordan M. Eizenga, Eric T. Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F. Lin, Benedict Paten, and Richard Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36(9):875–879, 2018.

[75] Daniel Gibney and Sharma Thankachan. On the hardness and inapproximability of recognizing wheeler graphs. In *Proc. 27th Annual European Symposium on Algorithms (ESA)*, 2019. Article 51.

[76] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

[77] Giorgio Gonnella and Stefan Kurtz. Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC Bioinformatics*, 13, 2012. Article 82.

[78] Gaston Gonnet, Ricardo Baeza-Yates, and Tim Snider. Information Retrieval: Data Structures and Algorithms, chapter 3: New indices for text: Pat trees and Pat arrays, 1992.

[79] Sara Goodwin, John McPherson, and W. Richard McCombie. Coming of age: ten years of next–generation sequencing technologies. *Nature Reviews Genetics*, 17:333–351, 2016.

[80] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High–Order Entropy–Compressed Text Indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[81] Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biology*, 21(1):1–20, 2020.

[82] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time–and–space barrier in constructing full–text indices. *SIAM Journal on Computing*, 38(6):2162–2178, 2009.

[83] Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013.

[84] David Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[85] Illumina, Inc. Illumina. Available online at `https://www.illumina.com`, last accessed: 2021-02-09.

[86] Human Phenome Institute. The Personal Genome Project China. Available online at `http://pgpchina.org`, last accessed: 2021-02-08.

[87] National Human Genome Research Institute. The Cancer Genome Atlas (TCGA). Available online at `https://www.genome.gov/Funded-Programs-Projects/Cancer-Genome-Atlas`, last accessed: 2021-02-08.

[88] National Human Genome Research Institute. The Cost of Sequencing a Human Genome.

[89] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226–232, 2012.

[90] Chirag Jain, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M. Phillippy. A fast approximate algorithm for mapping long reads to large reference databases. In *Proc. 13th International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 66–81. Springer, 2017.

[91] Richard Karp and Michael Rabin. Efficient randomized pattern–matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[92] Alice M. Kaye and Wyeth W. Wasserman. The genome atlas: navigating a new era of reference genomes. *Trends in Genetics*, 2021.

[93] Dominik Kempa and Tomasz Kociumaka. String Synchronizing Sets: Sublinear–Time BWT Construction and Optimal LCE Data Structure. In *Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, page 756–767, 2019.

[94] Dominik Kempa and Ben Langmead. Fast and Space–Efficient Construction of AVL Grammars from the LZ77 Parsing. arXiv:2105.11052, 2021.

[95] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proc. 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 827–840, 2018.

[96] John C. Kieffer and En Hui Yang. Grammar–based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[97] Szymon Kiełbasa, Raymond Wan, Kengo Sato, Paul Horton, and Martin Frith. Adaptive seeds tame genomic sequence comparison. *Genome Research*, 21(3):487–493, 2011.

[98] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

[99] David Koslicki and Hooman Zabeti. Improving MinHash via the containment index with applications to metagenomic analysis. *Applied Mathematics and Computation*, 354:206–215, 2019.

[100] Sebastian Kreft and Gonzalo Navarro. LZ77-Like Compression with Fast Random Access. In *Proc. 10th Data Compression Conference (DCC)*, page 239–248, 2010.

[101] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[102] Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan–genomics read alignment. *Journal of Computational Biology*, 27(4):500–513, 2020.

[103] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[104] Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, and Siu-Ming Yiu.

High throughput short read alignment via bi–directional BWT. In *Proc. 3rd IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 31–36, 2009.

[105] Ben Langmead. Aligning short sequencing reads with Bowtie. *Current Protocols in Bioinformatics*, 32(1):11–7, 2010.

[106] Ben Langmead and Steven L. Salzberg. Fast gapped–read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

[107] Benjamin Langmead. *Highly Scalable Short Read Alignment with the Burrows–Wheeler Transform and Cloud Computing*. PhD thesis, University of Maryland, 2009.

[108] Jesper Larsson and Alistair Moffat. Off–line dictionary–based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[109] Sunho Lee and Kunsoo Park. Dynamic rank/select structures with applications to run-length encoded texts. *Theoretical Computer Science*, 410(43):4402–4413, 2009.

[110] Heng Li. wgsim – Read simulator for next generation sequencing. *Bioinformatics*, 28:593–594, 2012.

[111] Heng Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014.

[112] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[113] Heng Li and Richard Durbin. Fast and accurate long–read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[114] Mike Liddell and Alistair Moffat. Decoding prefix codes. *Software: Practice and Experience*, 36(15):1687–1710, 2006.

[115] Yu Lin and Pavel A. Pevzner. Manifold de bruijn graphs. In *Proc. 14th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 296–310, 2014.

[116] Yongchao Liu, Bertil Schmidt, and Douglas Maskell. Parallelized short read assembly of large genomes using de Bruijn graphs. *BMC Bioinformatics*, 12, 2011. Article 354.

[117] Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, 678:22–39, 2017.

[118] Felipe A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and Giovanna Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms for Molecular Biology*, 15, 2020. Article 18.

[119] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, Jingbo Tang, Gengxiong Wu, Hao

Zhang, Yujian Shi, Yong Liu, Chang Yu, Bo Wang, Yao Lu, Changlei Han, David W. Cheung, Siu-Ming Yiu, Shaoliang Peng, Zhu Xiaoqian, Guangming Liu, Xiangke Liao, Yingrui Li, Huanming Yang, Jian Wang, Tak-Wah Lam, and Jun Wang. SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. Technical report, Gigascience, 2012.

[120] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru Tomescu. *Genome–Scale Algorithm Design*. Cambridge University Press, 2015.

[121] Veli Mäkinen, Bastien Cazaux, Massimo Equi, Tuukka Norri, and Alexandru I. Tomescu. Linear Time Construction of Indexable Founder Block Graphs. In *Proc. 20th International Workshop on Algorithms in Bioinformatics (WABI)*, 2020. Article 7.

[122] Veli Mäkinen and Gonzalo Navarro. Succinct Suffix Arrays based on Run–Length Encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[123] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of individual genomes. In *Proc. 13th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 121–137, 2009.

[124] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and Retrieval of Highly Repetitive Sequence Collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[125] Udi Manber and Gene Myers. Suffix arrays: a new method for on–line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[126] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler Transform. *Theoretical Computer Science*, 387(3):298–312, 2007.

[127] Giovanni Manzini. An analysis of the Burrows–Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[128] Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[129] Donald R. Morrison. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[130] Martin D. Muggli, Alexander Bowe, Noelle R. Noyes, Paul S. Morley, Keith E. Belk, Robert Raymond, Travis Gagie, Simon J. Puglisi, and Christina Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017.

[131] Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching reads to many genomes with the r–index. *Journal of Computational Biology*, 27(4):514–518, 2020.

[132] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space–Efficient Construction of Compressed Indexes in Deterministic Linear Time. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 408–424, 2017.

[133] Eugene Myers. The fragment assembly string graph. *Bioinformatics*, 21(Suppl. 2):ii79–85, 2005.

[134] Eugene W. Myers, Granger G. Sutton, Art L. Delcher, Ian M. Dew, Dan P. Fasulo, Michael J. Flanigan, Saul A. Kravitz, Clark M. Mobarry, Knut H.J. Reinert, Karin A. Remington, Eric Anson, Randall A. Bolanos, Hui-Hsien Chou, Catherine M. Jordan, Aaron L. Halpern, Stefano Lonardi, Ellen M. Beasley, Rhonda C. Brandon, Lin Chen, Patrick J. Dunn, Zhongwu Lai, Yong Liang, Deborah R. Nusskern, Ming Zhan, Qing Zhang, Xiangqun Zheng, Gerald M. Rubin, Mark D. Adams, and J. Craig Venter. A whole–genome assembly of Drosophila. *Science*, 287(5461):2196–2204, 2000.

[135] Gonzalo Navarro. *Compact Data Structures: A Practical Approach.* Cambrige University Press, 2016.

[136] Gonzalo Navarro. Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Computing Surveys*, 54(2), 2021. Article 29.

[137] Gonzalo Navarro and Veli Mäkinen. Compressed full–text indexes. *ACM Computing Surveys*, 39(1), 2007. Article 2.

[138] Gonzalo Navarro and Kunihiko Sadakane. Fully–Functional Static and Dynamic Succinct Trees. *ACM Transactions on Algorithms*, 10(3), 2014. Article 16.

[139] Takaaki Nishimoto, Tomohiro I., Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully Dynamic Data Structure for LCE Queries in Compressed Space. In *Proc. 41st International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2016. Article 72.

[140] Ge Nong. Practical linear–time O(1)-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, 31(3), 2013. Article 15.

[141] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced–sorting. In *Proc. 19th Data Compression Conference (DCC)*, pages 193–202, 2009.

[142] Daniel Saad Nogueira Nunes, Felipe A. Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. A grammar compression algorithm based on induced suffix sorting. In *Proc. 28th Data Compression Conference (DCC)*, pages 42–51, 2018.

[143] Carlos Ochoa and Gonzalo Navarro. RePair and All Irreducible Grammars are Upper Bounded by High–Order Empirical Entropy. *IEEE Transactions on Information Theory*, 65(5):3160–3164, 2019.

[144] National Institutes of Health. Human Microbiome Project. Available online at `https://commonfund.nih.gov/hmp`, last accessed: 2021-02-08.

[145] The Darwin Tree of Life Consortium. The Darwin Tree of Life. Available online at `https://www.darwintreeoflife.org`, last accessed: 2021-02-08.

[146] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy–compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

[147] Daisuke Okanohara and Kunihiko Sadakane. A linear–time Burrows-Wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 90–101, 2009.

[148] Brian Ondov, Todd Treangen, Páll Melsted, Adam Mallonee, Nicholas Bergman, Sergey Koren, and Adam Phillippy. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17, 2016. Article 132.

[149] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[150] Igor Pavlov. 7-Zip. Available online at `https://www.7-zip.org`, last accessed: 2021-02-09.

[151] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James Tiedje, and Titus Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[152] Yu Peng, Henry C.M. Leung, Siu-Ming Yiu, and Francis Y.L. Chin. IDBA-A Practical Iterative de Bruijn Graph De Novo Assembler. In *Proc. 14th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 426–440, 2010.

[153] Alberto Policriti and Nicola Prezza. From LZ77 to the run–length encoded Burrows–Wheeler transform, and back. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 78, 2017. Article 17.

[154] Victoria Popic and Serafim Batzoglou. Privacy–Preserving Read Mapping Using Locality Sensitive Hashing and Secure Kmer Voting. bioRxiv:046920, 2016.

[155] Nicola Prezza, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. Variable–order reference-free variant discovery with the Burrows–Wheeler Transform. *BMC Bioinformatics*, 21, 2020. Article 260.

[156] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k–ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007. Article 43.

[157] Jason Reuter, Damek Spacek, and Michael Snyder. High–Throughput sequencing technologies. *Molecular Cell*, 58(4):586–597, 2015.

[158] Michael Roberts, Wayne Hayes, Brian Hunt, Stephen Mount, and James Yorke.

Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[159] Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981.

[160] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomics index for finding MEMs. In *Proc. 25th International Conference on Research in Computational Molecular Biology (RECOMB)*, 2021.

[161] Wojceich Rytter. Application of Lempel–Ziv factorization to the approximation of grammar–based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[162] Cenk Sahinalp and Uzi Vishkin. Data compression using locally consistent parsing. Technical report, UMIACS Technical Report, 1995.

[163] Hiroshi Sakamoto, Louisa Seelbach Benkner, and Yoshimasa Takabatake. Practical Random Access to SLP–Compressed Texts. In *Proc. 27th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 221–231, 2020.

[164] Leena Salmela and Eric Rivals. LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, 30(24):3506–3514, 2014.

[165] Mark Sawicki, Ghassan Samara, Michael Hurwitz, and Edward Passaro. Human genome project. *The American Journal of Surgery*, 165(2):258–264, 1993.

[166] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. 22nd ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 76–85, 2003.

[167] Korbinian Schneeberger, Jörg Hagmann, Stephan Ossowski, Norman Warthmann, Sandra Gesing, Oliver Kohlbacher, and Detlef Weigel. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10(9), 2009. Article R98.

[168] Stephan Schuster. Next-generation sequencing transforms today's biology. *Nature Methods*, 5(1):16–18, 2008.

[169] Eugene S. Schwartz and Bruce Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

[170] Jared T. Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

[171] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein, Steven J.M. Jones, and Inanç Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

[172] Jouni Sirén. Compressed suffix arrays for massive data. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 63–74,

2009.

[173] Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype–aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020.

[174] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing finite language representation of population genotypes. In *Proc. 11th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 270–281, 2011.

[175] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

[176] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[177] Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, 34(3):300–302, 2016.

[178] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[179] Henrik Stranneheim, Max Käller, Tobias Allander, Björn Andersson, Lars Arvestad, and Joakim Lundeberg. Classification of DNA sequences using Bloom filters. *Bioinformatics*, 26(13):1595–1600, 2010.

[180] Oxford NANOPORE Technologies. Nanopore. Available online at `https://nanoporetech.com`, last accessed: 2021-02-09.

[181] Alexandru I. Tomescu and Paul Medvedev. Safe and complete contig assembly through omnitigs. *Journal of Computational Biology*, 24(6):590–602, 2017.

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

[183] Daniel Valenzuela. CHICO: A compressed hybrid index for repetitive collections. In *Proc. 15th International Symposium on Experimental Algorithms (SEA)*, pages 326–338, 2016.

[184] Daniel Valenzuela, Tuukka Norri, Niko Välimäki, Esa Pitkänen, and Veli Mäkinen. Towards pan–genome read alignment to improve variation calling. *BMC Genomics*, 19(87):124–180, 2018.

[185] Erwin L. van Dijk, Yan Jaszczyszyn, Delphine Naquin, and Claude Thermes. The third revolution in sequencing technology. *Trends in Genetics*, 34(9):666–681, 2018.

[186] Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.

[187] Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser. RCSI: Scalable similarity search in thousand (s) of genomes. *Proceedings of the VLDB Endowment*, 6(13):1534–1545, 2013.

[188] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11, 1973.

[189] Neil I. Weisenfeld, Vijay Kumar, Preyas Shah, Deanna M. Church, and David B. Jaffe. Direct determination of diploid genome sequences. *Genome Research*, 27(5):757–767, 2017.

[190] Aaron M. Wenger, Paul Peluso, William J. Rowell, Pi-Chuan Chang, Richard J. Hall, Gregory Concepcion, Jana Ebler, Arkarachai Fungtammasan, Alexey Kolesnikov, Nathan Olson, Armin Töpfer, Michael Alonge, Medhat Mahmoud, Yufeng Qian, Chen-Shan Chin, Adam M. Phillippy, Michael C. Schatz, Gene Myers, Mark A. DePristo, Jue Ruan, Tobias Marschall, Fritz J. Sedlazeck, Justin M. Zook, Heng Li, Sergey Koren, Andrew Carroll, David R. Rank, and Hunkapiller Michael W. Accurate circular consensus long–read sequencing improves variant detection and assembly of a human genome. *Nature Biotechnology*, 37(10):1155–1162, 2019.

[191] Daniel Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829, 2008.

[192] Aleksey Zimin, Guillaume Marçais, Daniela Puiu, Michael Roberts, Steven Salzberg, and James Yorke. The MaSuRCA genome assembler. *Bioinformatics*, 29(21):2669–2677, 2013.

[193] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[194] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable–rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.