Università degli Studi di Milano - Bicocca
Faculty of Science
Department of Informatics, Systems and Communication
Ph.D. program in Computer Science, Cycle XXXVIII

Time and Space Efficient Data Structures For Computational Pangenomics

Supervisor: *Prof. Paola Bonizzoni*

Co-supervisors: *Dr. Simone Ciccolella, Dr. Luca Denti*

Tutor: *Prof. Rafael Peñaloza Nyssen*

Ph.D. Coordinator: *Prof. Leonardo Mariani*

Davide Cozzi

Registration number 829827

A dissertation submitted for the degree of Doctor of Philosophy in Computer Science

Academic Year 2024-2025

Si fa il possibile
Ma tu fai anche l'impossibile
A breve anche rallentare il tempo
Why not?

Tecnicamente sarei un teorico
Sì, il teorico più pratico che conosco

Acknowledgments

These acknowledgments won't be short.

First of all, I want to sincerely thank my supervisor Paola for being such an incredible guide throughout these three years, always encouraging and supporting me in my research. There were moments when you drove me a little crazy, but I feel truly lucky to have had the opportunity to complete this journey under your guidance, even though it took me more than a year to start addressing you informally, and I probably owe you at least thirty bottles of water.

I would like to thank my co-supervisors, Luca and Simone. I know you don't like being considered my superiors, even though you are, but after these three years, to me you are even more than colleagues: you are friends. Thank you for all the discussions, the help with projects, and for including me in new activities. You have been the driving force behind this thesis.

I want to thank Gianluca, Raffaella, and Yuri. You have been unofficial co-supervisors and a fundamental part of this wonderful experience. I will never forget Gianluca, who helped me with my Ph.D. application, Raffaella, who was my advisor during my master's thesis, and the nights in Japan spent on calls with Yuri to finalize the paper for CPM (not to mention the special issue for Manzini).

I want to thank Prof. Sadakane for hosting me at the University of Tokyo, and Prof. Richard Durbin for hosting me at the University of Cambridge. Those months were amazing, and I had the chance to meet wonderful people dedicated to very interesting researches. I also want to thank Dr. Simone Rubinacci for the many calls and, fortunately, a few days of visits at the Institute for Molecular Medicine Finland, which allowed me to try to bridge the gap between computer science and biology. I want to thank Dr. Ole Schulz-Trieglaff and Dr. Massimiliano Rossi for allowing me to collaborate with them at Illumina (including as visitor in Cambridge), giving me the opportunity to work, even if only a little, in a completely new environment.

I want to thank Prof. Travis Gagie and Prof. Christina Boucher, who contributed to my Ph.D. admission as well as to the writing of several of my papers. In one way or another, it has been four years now that I know I can count on your support. I also want to thank Prof. Dominik Köppl and Prof. Ben Langmead, my co-authors, with whom it has been a pleasure to work.

Mattia, Brian, Davide, Jorge, Victor, and Younan, our lab has always been so full

that we even had to use extra rooms. It doesn't feel like a research team, but more like a bunch of high school friends. The fun and lively atmosphere we've created makes me feel more and more like I'm one of the luckiest people in all of U14, thank you so much.

What can I say about Fede, Jack, and Mattia (yes, you again). About eight years ago, we were in the "auletta" preparing for our bachelor's exams, and after all this time, you are still the best people I've met in Bicocca. Bachelor's together, master's together, and even a few years after the master's. Thanks to you, I graduated twice, and yes, also thanks to you, this time I'm getting my Ph.D.

But you're not the only ones. I also want to thank everyone who has accompanied me over these years, enriching our lunches with fascinating "argomenti divisivi", interesting research ideas, but also less serious discussions. I will miss being the troublemaker who always ends up saying: "Coffee?".

I also want to thank Davide, my friend for 25 years, who has always supported me during the toughest moments; Marco, probably the most different person from me in this world, a constant presence in my life since elementary school; and Luca, a bit more similar to me in some ways but still a humanist, thank you for being, along with Marco, a reliable presence during relaxing evenings and dinners. Finally, thanks to Mattia (the other one this time), from our years studying physics until now, always ready to give excellent life advices.

I want to thank my parents, my grandmothers Maria and Laura, and my aunt Rita for always being by my side throughout all these years. I know you will always be there for me. I also want to thank my grandfather Angelo and my uncle Orazio. I have reached this important milestone thanks to your advice and the way you taught me to face the world: "Non siete in cielo o sottoterra, voi mi state seduti accanto".

We've reached the end, and there's just one person left, the most important one. Thank you, Greta. Without you, I wouldn't be here. Without you, I probably wouldn't have even finished my bachelor's degree. You have endured another three and a half years, managed eight hours of time difference to talk to me, months alone, and a madman who enjoyed spending evenings talking about data structures. I hope it was worth it. As I wrote in my master's thesis, more than eight years ago you changed my life, and this achievement is more yours than you might want to admit. Thank you, saying it one last time, I have truly been lucky.

Abstract

Computational pangenomics is an emerging research field in bioinformatics that comes from the idea of shifting away from the traditional concept of a linear reference genome, towards a representation that explicitly accounts for genetic variations across a large collection of individual genomes. This concept is known as the pangenome, that from a computational perspective, can be represented in different ways, for instance : a) as a (multi)set of binary sequences, *i.e.* a binary matrix, representing haplotypes and encoding the differences between a set of genomes and a reference genome, b) as the concatenation of all the genomes of the population, or c) as a node-labeled graph, where common substrings between genomes are merged into the same node label.

In this Ph.D. thesis, we first address the former type of representation, *i.e.* the haplotype panel encoded as a set of binary sequences, which is typically indexed to support queries using the Positional Burrows–Wheeler Transform (PBWT). Inspired by recent advances in classical text indexing techniques, such as the well-known r-index, we have developed several data structures to efficiently build a run-length encoded PBWT (RLPBWT), allowing us to store the PBWT compactly while still supporting common string-to-matrix pattern-matching queries. This exploration also led to the development of the μ-PBWT, a variant of the RLPBWT that stores all the information necessary to address string-to-matrix pattern-matching problems efficiently, solving computational problems that currently require excessive space and time. Thanks to the μ-PBWT, we can index a haplotype panel using up to two orders of magnitude less memory than the original PBWT, enabling the analysis of large haplotype datasets. In this thesis, we will also describe how to use the μ-PBWT to solve a common biological problem, the genotype phasing problem, combinatorially in sublinear space. Moreover, we also highlight the advantages and limitations of using a combinatorial approach instead of common probabilistic approaches.

We then focus on the pangenome graph, presenting `gindex`, a tool based on the Multidollar Burrows–Wheeler Transform that efficiently computes string-to-graph exact matches and scales better than state-of-the-art tools, up to human

pangenomes. Despite being slower in queries, `gindex` supports arbitrary query lengths and addresses the main limitations of the most used pangenome graph indexes: high memory usage during indexing and limitation of query length, due to the underlying data structures.

Finally, we address the biological problem of detecting and quantifying Alternative Splicing (AS) events using efficient graph-based models. After presenting `ESGq`, a non-pangenomic prototype, we introduce `pantas`, the first pangenome graph-based model for the detection and quantification of AS events. With `pantas`, we demonstrated that accounting for the genetic variability within the studied population improves the accuracy of AS event detection, showing that the use of pangenomes can effectively contribute to solving biological problems.

Our results demonstrate that the use of a pangenome can enable new fundamental discoveries in biology and personalized medicine, showing great potential for Genome-Wide Association Studies (GWAS). Hence, given these computational results, we are confident that the development of pangenome-based tools will be essential in the near future for analyzing large-scale genomic data, enabling novel discoveries from both computational and biological perspectives.

La pangenomica computazionale è un campo di ricerca emergente della bioinformatica che nasce dall'idea di abbandonare il concetto tradizionale di genoma di riferimento lineare a favore di una rappresentazione che tenga conto delle variazioni genetiche presenti in un insieme di genomi. Questo concetto è noto come pangenoma e, da un punto di vista computazionale, può essere rappresentato in diversi modi, come: a) come (multi)insieme di sequenze binarie, ovvero una matrice binaria, che rappresenta aplotipi e codifica le differenze tra un insieme di genomi e un genoma di riferimento, b) come concatenazione di tutti i genomi della popolazione, oppure c) come grafo etichettato, in cui le sottostringhe comuni tra i genomi vengono unite nelle stesse etichette dei nodi.

In questa tesi di dottorato, affrontiamo innanzitutto il primo tipo di rappresentazione, ovvero il pannello di aplotipi codificato come insieme di sequenze binarie, che viene tipicamente indicizzato per supportare le query utilizzando la Trasformata di Burrows-Wheeler Posizionale (PBWT). Ispirati dai recenti progressi nelle tecniche classiche di indicizzazione dei testi, come il noto r-index, abbiamo sviluppato diverse strutture dati per costruire in modo efficiente una run-length encoded PBWT (RLPBWT), memorizzando la PBWT in modo compatto, pur continuando a supportare le comuni query string-to-matrix. Questa ricerca ha portato anche allo sviluppo della μ-PBWT, una variante della RLPBWT che memorizza tutte le informazioni necessarie per affrontare in modo efficiente i problemi di pattern-matching string-to-matrix, risolvendo problemi computazionali che attualmente

richiedono spazio e tempo eccessivi. Grazie alla μ-PBWT, possiamo indicizzare un pannello di aplotipi utilizzando fino a due ordini di grandezza in meno di memoria rispetto all'originale PBWT, consentendo l'analisi di grandi dataset di aplotipi. In questa tesi descriveremo anche come utilizzare la μ-PBWT per risolvere un comune problema biologico, quello del phasing di genotipi, in modo combinatorio in uno spazio sublineare. Inoltre, evidenzieremo anche i vantaggi e i limiti dell'utilizzo di un approccio combinatorio rispetto ai più comuni approcci probabilistici.

Ci concentriamo poi sul grafo del pangenoma, presentando `gindex`, uno strumento basato sulla Trasformata di Burrows-Wheeler multidollaro che calcola in modo efficiente i match esatti tra stringhe e grafi e che scala meglio rispetto agli strumenti allo stato dell'arte, scalando su pangenomi umani. Nonostante sia più lento nelle query, `gindex` supporta query di lunghezza arbitraria e affronta le principali limitazioni degli indici per grafi più usati: l'elevato utilizzo di memoria durante l'indicizzazione e la limitazione della lunghezza delle query, causata dalle strutture dati sottostanti.

Infine, affrontiamo il problema biologico dell'individuazione e della quantificazione degli eventi di splicing alternativo (AS) utilizzando modelli efficienti basati su grafi. Dopo aver presentato `ESGq`, un prototipo non pangenomico, introduciamo `pantas`, il primo modello basato su grafi del pangenoma per il rilevamento e la quantificazione degli eventi di AS. Con `pantas`, abbiamo dimostrato che tenere conto della variabilità genetica all'interno della popolazione studiata migliora l'accuratezza del rilevamento degli eventi di AS, mostrando che l'uso dei pangenomi può contribuire efficacemente alla risoluzione di problemi biologici.

I nostri risultati dimostrano che l'uso del pangenoma può consentire nuove scoperte fondamentali nella biologia e nella medicina personalizzata, mostrando un grande potenziale per i Genome-Wide Association Studies. Pertanto, alla luce di questi risultati computazionali, siamo fiduciosi che lo sviluppo di strumenti basati sul pangenoma sarà essenziale nel prossimo futuro per l'analisi di dati genomici su larga scala, consentendo nuove scoperte sia dal punto di vista computazionale che biologico.

List of Publications

Publications discussed in this Ph.D. thesis

Paola Bonizzoni, Christina Boucher, **Davide Cozzi**, Travis Gagie, Dominik Köppl, and Massimiliano Rossi. *Data structures for SMEM-finding in the PBWT*. In International Symposium on String Processing and Information Retrieval (SPIRE 2023). Springer Nature Switzerland, 2023.⋆
https://doi.org/10.1007/978-3-031-43980-3_8

Davide Cozzi, Massimiliano Rossi, Simone Rubinacci, Travis Gagie, Dominik Köppl, Christina Boucher, and Paola Bonizzoni. *μ-PBWT: a lightweight r-indexing of the PBWT for storing and querying UK Biobank data*. Bioinformatics, 2023.
https://doi.org/10.1093/bioinformatics/btad552

Davide Cozzi†, Paola Bonizzoni, and Luca Denti†. ESGq: A*lternative Splicing Events Quantification across Conditions based on Event Splicing Graphs*. In Proceedings of the 23rd Conference Information Technologies – Applications and Theory, ITAT, 2023
https://ceur-ws.org/Vol-3498/paper31.pdf

Paola Bonizzoni, Christina Boucher, **Davide Cozzi**, Travis Gagie, and Yuri Pirola. *Solving the Minimal Positional Substring Cover Problem in Sublinear Space*. In 35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024), Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.⋆
https://doi.org/10.4230/LIPIcs.CPM.2024.12

Simone Ciccolella†, **Davide Cozzi**†, Gianluca Della Vedova, Stephen Njuguna Kuria, Paola Bonizzoni, and Luca Denti†. *Differential quantification of alternative splicing events on spliced pangenome graphs*. PLOS Computational Biology, 2024.
https://doi.org/10.1371/journal.pcbi.1012665

Davide Cozzi, Ben Langmead, Yuri Pirola, Christina Boucher, and Paola Bonizzoni. *Phasing data from genotype queries via the μ-PBWT*. In The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.
https://doi.org/10.4230/OASIcs.Manzini.10

Davide Cozzi, Brian Riccardi, Luca Denti, Simone Ciccolella, Kunihiko Sadakane, and Paola Bonizzoni. *Pangenome Graph Indexing via the Multidollar-BWT*. In 23rd International Symposium on Experimental Algorithms (SEA 2025). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.
https://doi.org/10.4230/LIPIcs.SEA.2025.13

Additional publication not discussed in this Ph.D. thesis

Sharvani Mahadevaraju, Soumitra Pal, Pradeep Bhaskar, Brennan D. Mc-Donald, Leif Benner, Luca Denti, **Davide Cozzi**, Paola Bonizzoni, Teresa M. Przytycka, and Brian Oliver. *Diverse somatic Transformer and sex chromosome karyotype pathways regulate gene expression in Drosophila gonad development*. eLife, 2024, reviewed preprint.
https://doi.org/10.7554/eLife.101641.1

† *These authors contributed equally to the work.*
⋆ *By convention, in theoretical computer science publications, authors are typically listed alphabetically and the order does not directly reflect their contributions.*

Contents

Chapter 1

Introduction

The decreasing cost of sequencing and the advancement of increasingly accurate technologies have led to the generation of vast amounts of genomic data in recent years, thus enabling both the simultaneous study of multiple genomes and moving beyond studies based on a single reference genome [1, 2, 3]. Indeed, in recent years, several international initiatives have been launched with the specific goal of collecting and analyzing sequencing data from large populations to build comprehensive catalogs of genomic variations related to relevant diseases, such as the 1000 Genomes Project (1KGP) [4, 5], the UK Biobank [6], and the All Of Us initiative [7]. In particular, researchers are increasingly moving away from the conventional linear reference genome paradigm toward a pangenome-based reference framework. This paradigm shift allows biologists to obtain more comprehensive and unbiased results, overcoming the limitations imposed by relying on a single reference genome, *i.e.* avoiding the so-called reference-bias problem.

Historically, the concept of the pangenome was first introduced in 2005 [1] in the field of microbiology to describe the core and accessory genes shared among different bacterial strains. Later, the term pangenome was used to denote a population-level reference system comprising multiple genomes from the same species [8]. Today, pangenomics has become increasingly essential in biology and personalized medicine, driven mainly by Genome-Wide Association Studies (GWAS) [9], which require the analysis of large collections of individual genomes.

From the beginning, researchers in computational biology recognized that a reference pangenome would need a suitable data structure to represent genomic sequences. Therefore, before discussing computational pangenomics, we need to

look back in time and revisit the foundational text indexing techniques developed years ago, which are still used today. In 1994, Burrows and Wheeler published the first paper on a compression algorithm [10], called the Block-sorting Lossless Data Compression Algorithm, which would later be known as the Burrows-Wheeler Transform (BWT). Initially, this transformation was designed to compress text, producing a highly compressible string with long runs of the same character. Then, the other milestone we need to cite is the FM-index by Ferragina and Manzini in 2000 [11], a BWT-based self-index. Thanks to the FM-index and the introduction of wavelet trees [12] by Grossi and Vitter in 2000, a plethora of new techniques have been developed to compute pattern matching against long strings, including genomic sequences. Extensions of the BWT are currently the algorithmic core of most pangenome indexing methods, and they form the core of most of the methods proposed in this thesis.

Since 2015, with the release of the 1000 Genomes Project (1KGP) [4, 5], which started in 2008, the opportunity to study multiple genomes from a population of individuals has become a reality. This project is the result of years of improvements in sequencing techniques, thanks to which the cost of each sequencing has been significantly reduced. Considering that the Homo Sapiens reference genome (GRCh38.p14) comprises approximately 3.1 billion base pairs and contains around 59 000 genes, as reported by NCBI[1], the 1000 Genomes Project (1KGP) has identified more than 88 million variants across the analyzed human populations: 84.7 million single nucleotide polymorphisms (SNPs), 3.6 million short insertions and deletions (indels), and approximately 60 000 structural variations (affecting sequences longer than 50 nucleotides) [2]. Moreover, it is crucial to consider that the initial goal of sequencing at least 100 000 individuals has already been achieved and surpassed, thanks to recent advances in sequencing technologies, such as Next-Generation Sequencing (NGS) and third-generation sequencing, as well as the establishment of large-scale population databases, including the UK Biobank, which now contains data of approximately 500 000 individuals. Consequently, managing this massive volume of data poses a significant challenge for current state-of-the-art algorithms. As prices decrease and the volume of genomic data produced increases, the need for dedicated data structures and computational tools has become a

[1] `https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.40/`

priority in both the theoretical computer science and bioinformatics communities.

A set of genomes can be computationally viewed in different ways [2], for instance: a) as a (multi)set of binary sequences, *i.e.* a binary matrix, encoding the differences between a set of genomes and a reference genome, b) as the concatenation of all the genomes of the population, or c) as a node-labeled graph, where common substrings between genomes are merged into the same node label.

In 2014, Richard Durbin, one of the leading scientists behind the 1KGP, proposed the Positional Burrows-Wheeler Transform(PBWT) [13], an efficient data structure for indexing and querying haplotype panels. We recall that each individual carries two haplotypes, one maternal and one paternal, whose combination defines the genotype. In this case, Durbin considered a naïve representation of a pangenome, storing a set of binary strings which encodes only the differences between a set of genomes and, again, a reference genome. Hence, although multiple genomes are being studied simultaneously, reference bias cannot be avoided. In detail, considering the bi-allelic case in which only one alternative allele is possible, a "0" is encoded if a specific position of the considered haplotype matches the reference genome, and a "1" otherwise, when the alternative allele is present. This encoding produces a set of binary strings, where each sequence represents a haplotype and each position a variation site.

Moreover, as in [14, 15, 16], a less naïve way is to concatenate multiple genomes, indexing and querying the whole concatenation, leveraging high-repetitive FM-index-like indexes such as the r-index [17]. In this representation, the common regions shared by the input strings are compressed in the index, but not in the input string itself. In fact, the high repetitiveness of this concatenation produces a high run-length encoded BWT.

The most widely used representation of a pangenome is the pangenome graph. This concept has been explored in several works, for instance in [18], where the BWT is extended to acyclic directed labeled graphs, in [19], where it is presented as a generalization of genome graphs, and in [20], which focuses on the indexing of variation graphs. In this representation, common subsequences shared among multiple genomes are compactly encoded within the label of a single node. Moreover, the compactness of this representation is fundamental for visualizing similarities between various genomes. As a new approach to encoding bioinformatics data,

many open problems still need to be addressed by the bioinformatics community [2], and this thesis aims to tackle some of them. We will explore the pangenomics framework from both theoretical and practical perspectives, discussing the foundational concepts underlying our method and presenting implementations and experiments on real pangenomic data. We will first address this problem in the context of haplotype data from the large biobanks, focusing on the algorithmic development of specialized queries over large haplotype panels. In addition, we will explore space-efficient methods for indexing and querying genomic sequences within pangenome graphs, with particular attention to the practical complexity of locating the starting nodes of the matches in these graphs. Finally, we will consider pantranscriptome graphs, which are specifically designed to address the problem of identifying gene structures induced by transcripts and are essential for tackling a key biological challenge: the detection and quantification of Alternative Splicing events.

Most relevant pangenomics queries aim to identify specific or shared patterns across collections of genomes, and this work seeks to improve the efficiency and scalability of related biological analyses. These include, for example: a) studying viral evolution and spread, which is essential for developing effective antiviral drugs and understanding the formation of viral pangenomes resulting from imperfect viral replication, b) performing Genome-Wide Association Studies (GWAS) to identify correlations between genetic variants and disease susceptibility, c) phasing genotypes, that is, determining the two parental haplotypes underlying a given genotype, and d) performing genotype imputation, the process of predicting unobserved genotypes for a given set of individuals.

In the following, we will summarize the main contributions of the thesis. Initially, we will address the issue of efficiently indexing and querying haplotype panels, extending Durbin's results on the PBWT when an external haplotype is used as a query. The first problem we tackle is reducing the space requirements of a full PBWT index. We propose a set of composite run-length encoded data structures [21], strongly inspired by the r-index and other run-length encoded BWT results, that can index a binary haplotype panel in sublinear space. We analyzed various composite data structures for run-length encoded PBWT (RLPBWT), leveraging succinct data structures [22] to encode the internal data structures of the PBWT. Finally, we

propose the μ-PBWT [23, 24], a ready-to-use space-efficient implementation of the RLPBWT, which optimizes what has already been developed for the RLPBWT. This tool can be used to compute various types of matches between a haplotype panel and an external haplotype. We address two main matching problems in the string-to-matrix pattern matching context: computing the Set-Maximal Exact Matches (SMEMs) [13], and the Minimal Positional Substring Cover (MPSC) problem [25]. Both problems can also be constrained to be shared with at least k haplotype in the input panel. We demonstrate, through a comprehensive experimental analysis of the 1KGP data, that μ-PBWT can reduce index memory requirements by two orders of magnitude compared to state-of-the-art PBWT implementations, while maintaining query-time effectiveness. We also demonstrate that the μ-PBWT can scale on large datasets, such as the UK Biobank data [26, 6], considering approximately 150 000 individuals. Finally, to fill the gaps between computer science and biology, we tried to tackle a common biological problem, *i.e.* the genotype phasing problem, by prototyping a fully combinatorial phasing model based on μ-PBWT [27]. We compare this prototype with Beagle [28, 29], one of the most widely used HMM-based tools, demonstrating that our approach can be practical when using an extensive and comprehensive haplotype reference panel, and can report the exact solution, if it exists, in contrast to Beagle.

This thesis continues by considering the most common representation of a pangenome, the pangenome graph. The first problem we address is the pangenome graph indexing, a problem partially solved by the `GCSA2` index [18, 30, 31]. In this context, we introduce `gindex` [32], a multidollar-BWT-based index for pangenome graphs designed to address the two main limitations of `GCSA2`: the occurrence of false positive matches when query length exceeds 256bp, and the exponential space requirements during indexing, which make it unfeasible to process graphs containing highly complex regions. Leveraging some multidollar-BWT properties, `gindex` can index the concatenation of node-labels and suit it alongside the graph topology to compute string-to-graph exact matches. In other words, `gindex` aims to construct an FM-index-like data structure for node-labeled graphs, similar to `GCSA2`. In `gindex` we also introduced a preprocessing step that precomputes a "cache" of matched k-mers using a dynamic programming algorithm to avoid unnecessary computations in complex regions of the graph. Note that `gindex` is

capable of indexing human pangenome graphs, such as those constructed by the Human Pangenome Reference Consortium (HPRC) [33], and, despite being slower than `GCSA2`, it remains a viable option when building a `GCSA2` index is not feasible.

Finally, we investigated the use of indexing a pangenome graph in a concrete biological setting, where the main goal is the haplotype-aware detection of splicing variants from transcriptomic data. The main purpose of using a pantranscriptomics graph instead of a linear reference is to avoid the bias introduced by a single reference gene annotation. More precisely, we present the first pantranscriptomics model built on the pangenome graph, designed to detect and quantify haplotype-aware alternative splicing (AS) events, one of the most critical biological mechanisms involved in protein synthesis. The model has been proposed in the tool `pantas` [34], which is a generalization of `ESGq` [35], a non-pangenomics graph-based model we developed for detecting and quantifying alternative splicing (AS) events. With `ESGq` we introduce the concept of event splicing graphs, a compact graph representation of annotated AS events against which we align RNA-seq reads. Graph-based models are widely used in the literature to address this problem [36, 37]; hence, it is "natural" to extend them to consider a pangenome graph. By encoding all haplotypes as variants within the pangenome graphs, we can perform RNA-seq alignments without reference bias, thereby improving the interpretability of our results by accounting for the genetic variability of the population under investigation. These alignments are used to augment the graph annotations, which are analyzed to detect and quantify both annotated and novel AS events. These augmented pangenome graphs, referred to as annotated spliced pangenome graphs, can be analyzed to detect both annotated and novel AS events. My evaluation of `pantas` on both simulated and real datasets demonstrated that it can achieve even more accurate results than state-of-the-art linear reference-based tools [38, 39, 37], thereby demonstrating the potential benefits of using pangenomic tools for classical computational problems, such as the detection of alternative splicing (AS) events from sequencing data.

Thesis outline In this thesis, Chapter 2 presents the fundamental preliminaries, encompassing both theoretical computer science concepts and bioinformatics notions. In the same chapter, we also briefly outline some of the biological foundations of this thesis. In Chapter 3, we present our results on the RLPBWT and the

μ-PBWT, ranging from theoretical results to experimental analysis. We continue in Chapter 4 with `gindex` and in Chapter 5 with `ESGq` and `pantas`, describing our models both theoretically and practically. For each of these three chapters, we provide a brief introduction to the current state of the art to contextualize the motivations behind our work, a comprehensive description of the adopted methods, the experimental analyses and relevant implementation details, and finally a short conclusion outlining possible future research directions. Finally, in Chapter 6, we conclude this thesis with a comprehensive overview of what we have done to improve the current state-of-the-art in pangenomics, presenting also potential future work.

Chapter 2

Preliminaries

This chapter introduces the theoretical foundations underlying the methods presented in this thesis. We begin by defining the basic concepts and notation for strings and graphs, followed by a review of the fundamental succinct data structures employed in this work. Next, we explore topics related to pattern matching and indexing, using strings, matrices, sets of sequences, and node-labeled graphs as reference frameworks.

Finally, we outline the key biological concepts and the bioinformatics/pangenomics principles necessary to understand the context of this thesis.

2.1 Strings and graphs

Strings A string T of length n is defined as a sequence of symbols $T[1]T[2]\cdots T[n]$ over a finite and ordered alphabet $\Sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_{|\Sigma|}\}$. In detail, we denote ε the empty string and, for $1 \leq i \leq j \leq n$, $T[i..j]$ as the substring of T starting in position i included and ending in position j included. Note that $T[i..j] = \varepsilon$ if $j > i$. $T = T' \cdot T''$ is the concatenation of two strings T' and T'', having $\cdot$ as string concatenation operator.

For $1 \leq k \leq n$, the substring $T[1..k]$ is called the k-th prefix and the substring $T[k..n]$ is called the k-th suffix. Moreover, we denote $T_{rev} = T[n]T[n-1]\cdots T[1]$ as the reverse string of T. We adopt the symbol $\prec$ for the standard lexicographic order. In this thesis, we also leverage the colexicographical order, denoted with $\prec_{co}$, having that, given two strings T' and T'', $T' \prec_{co} T''$ if and only if $T'_{rev} \prec T''_{rev}$.

We indicate $\mathsf{rot}_k(T)$ the k-th rotation of the string T, which is a string of the same length of T obtained concatenating the k-th suffix and the $(k-1)$-th prefix, *i.e.* $\mathsf{rot}_k(T) = T[k..n] \cdot T[1..k-1]$. For example, given $T = \text{AAAGGTAAAACCC}$ we have $\mathsf{rot}_6(T) = \text{TAAAACCC} \cdot \text{AAAGG} = \text{TAAAACCCAAAGG}$

Given a string T, a run is defined as a maximal substring $T[i..j]$ such that: a) $T[k] = \sigma$, for all $i \leq k \leq j$, b) $i = 0 \vee T[i-1] \neq \sigma$, and c) $j = n \vee T[j+1] \neq \sigma$. In other words, a run is a substring where each symbol is the same and can not be extended to the left or right without introducing another symbol. Each run is represented by a pair (σ, ℓ), where σ is the symbol of the run and ℓ the length of the run. With $\rho(T)$, we denote the total number of runs in T. For example, given $T = \text{AAAGGTAAAACCC}$ we have $\rho(T) = 5$ runs: $(A, 3)$, $(G, 2)$, $(T, 1)$, $(A, 4)$, and $(C, 3)$.

Given two strings A and B, we denote $\mathsf{lcp}(A, B)$ the longest common prefix shared by these two strings. Given a n-length string T and two positions i and j such that $1 \leq i, j \leq n$, we define longest common extension (LCE) as the longest common prefix shared by the i-th suffix and the j-th suffix of T, denoting it as $\mathsf{LCE}(T, i, j)$. For example, given $T = \text{AAAGGTAAAACCC}$, $\mathsf{LCE}(T, 1, 7) = AAA$, being the longest common prefix between the suffix $T[1..n] = \text{AAAGGTAAAACCC}$ and the suffix $T[7..n] = \text{AAAACCC}$, *i.e.* $\mathsf{lcp}(T[1..n], T[7..n])$.

Directed graphs We define a directed graph G as a pair (V, E) where V is the set of nodes/vertices and $E \subseteq V \times V$ is the set of edges/links represented by ordered pairs of vertices. If edges are unordered pairs of vertices, the graph is referred to as an undirected graph. We refer to labelled graphs if nodes are augmented with text labels.

Given a directed graph $G = (V, E)$ and a node $v \in V$, if there is no node v' such that $(v', v) \in E$, then v is called a source. Conversely, if there is no node v' such that $(v, v') \in E$, then v is called a sink. We can have multiple sources and sinks in the same graph. We define a k-length path as a sequence of $k+1$ nodes $\{v_1, v_2, \cdots, v_{k+1}\}$, where for each $1 \leq i \leq k$, $(v_i, v_{i+1}) \in E$, *i.e.* we have an edge for each consecutive pair of nodes. A path $\{v_1, v_2, \cdots, v_{k+1}\}$ is called cycle if $v_1 = v_{k+1}$. If $G = (V, E)$ presents cycles, it is defined as a cyclic graph; otherwise, it is an acyclic graph.

Given a node $v \in V$, we define the indegree of v as the total number of incoming edges in v, *i.e.* $indegree(v) = |\{u \in V | (u, v) \in E\}|$. Similarly, we define the outdegree of v the total number of outgoing edges from v, *i.e.* $outdegree(v) = |\{u \in V | (v, u) \in E\}|$. In a directed graph, if we can reach each node by following a path starting from any other node, we say the graph is strongly connected. In the case of an undirected graph, it is referred to as a connected graph. Without further details, we recall that a tree is an undirected, connected, and acyclic graph with $|E| = |V| - 1$.

See Cormen et al. "Introduction to Algorithms" book [40], for additional string and graph definitions and algorithms.

2.2 Succinct data structures

In his Ph.D. thesis (1988), Jacobson formalized the concept of succinct data structures [41], building upon some previous results in the literature. Formally, given some data which optimally requires $\mathcal{Z}$ bits, we define a data structure used to store and query that data to be: a) implicit, if it takes $\mathcal{Z} + \mathcal{O}(1)$ bits (for example $\mathcal{Z} + 32$ bits), b) compact, if it takes $\mathcal{O}(\mathcal{Z})$ bits (for example $2\mathcal{Z}$ bits), and c) succinct, if it takes $\mathcal{Z} + o(\mathcal{Z})$ bits (for example $\mathcal{Z} + \log \mathcal{Z}$ bits or $\mathcal{Z} + \sqrt{\mathcal{Z}}$ bits). Focusing on succinct data structures, these approaches require space close to the theoretical minimum, adding only $o(\mathcal{Z})$ bits, while maintaining operation times nearly as fast as those achievable without any space constraints.

Succinct bitvectors Proposed by Jacobson in 1989 [22], succinct bitvectors are the fundamental succinct data structure on top of which most other succinct data structures are built.

A bitvector B is a n-length binary array, having $B[i] = \{0, 1\}$ for each $1 \leq i \leq n$. A bitvector can be randomly accessed in constant time, *i.e.*, for each $1 \leq i \leq n$, the $B[i]$ can be done in $\mathcal{O}(1)$. We will also refer to the random access operations as the access function. Jacobson proved that, using $o(n)$ additional bits, we can perform in constant time two useful functions, namely rank and select, which are fundamental in building new algorithms based on querying these succinct bitvectors.

Definition 1 (rank function on succinct bitvectors)**.** *Given a bitvector B of length n and given a position i such that $1 \leq i \leq n$, the* rank *operation for position i on the bitvector B is defined as:* $\mathsf{rank}(B, i) = \sum_{k=1}^{k<i} B[k]$.

Informally, the rank function counts all the "1"s in B up to position i, excluding i itself. The rank function can be generalized to count all the occurrences of a binary symbol $\sigma = \{0, 1\}$ in B up to i. In this case, following [42], we can extend the rank definition as: $\mathsf{rank}_\sigma(B, i) = |\{k|\, 1 \leq k < i, B[k] = \sigma\}|$. Note that $\mathsf{rank}_0(B, i) = (i - 1) - \mathsf{rank}_1(B, i)$ and vice versa. If not differently specified, we use $\mathsf{rank}(B, i)$ referring to $\mathsf{rank}_1(B, i)$.

Definition 2 (select function on succinct bitvectors)**.** *Given a bitvector B of length n and given a position i such that $1 < i \leq \mathsf{rank}(B, n)$, the* select *operation for position i on the bitvector B is defined as:* $\mathsf{select}(B, i) = \min\{k|\, \mathsf{rank}(B, k+1) = i\}$.

Informally, the select function returns the position of the i-th "1" in B. Like in the rank case, the select function can be generalized to return the position of the i-th binary symbol $\sigma = \{0, 1\}$ in B. Formally, this generalization be written as: $\mathsf{select}_\sigma(B, i) = \min\{k|\, \mathsf{rank}_\sigma(B, k + 1) = i\}$. If not differently specified, we use $\mathsf{select}(B, i)$ referring to $\mathsf{select}_1(B, i)$. See example 1 to clarify the use of rank and select operations.

Example 1. *Consider this bitvector $B = \{1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0\}$ of length $n = 14$. We have, for example:* $\mathsf{rank}(B, 6) = \mathsf{rank}_1(B, 6) = 3$, $\mathsf{select}(B, 5) = \mathsf{select}_1(B, 5) = 9$, $\mathsf{rank}_0(B, 2) = 1$, *and* $\mathsf{select}_0(B, 4) = 6$.

Without storing any additional information, we can answer both rank and select scanning the bitvector in $\mathcal{O}(n)$. Jacobson [22] proved that, by storing $o(n)$ additional bits, we can perform both of them in $\mathcal{O}(1)$ time. The structures that allow these constant-time operations can be built in $\mathcal{O}(n)$ time.

Moreover, various studies have been conducted to further reduce the number of additional bits, albeit at the cost of not always guaranteeing access, rank, and select in constant time. Table 2.1, Table 2.2, and Table 2.3 summarize these results [43].

For further details and additional theoretical results, we refer to Navarro's "Compact Data Structures" book [46], to Mäkinen's et al. "Genome-Scale Algorithm Design" book [42], and to [22].

Table 2.1: Space requirements for some succinct bitvectors variants having a bitvector of length n, with m 1s and K as block size for the interleaving variant.

Variant	**Total Space**
Plain bitvector [43]	$64\left\lceil \frac{n}{64} + 1 \right\rceil$
Interleaved bitvector [43]	$\approx n\left(1 + \frac{64}{K}\right)$
H_0*-compressed bitvector* [44]	$\approx \left\lceil \log \binom{n}{m} \right\rceil$
Sparse bitvector [45]	$\approx m\left(2 + \log \frac{n}{m}\right)$

Table 2.2: Time complexity and amount of additional bits of the rank function for some succinct bitvectors variants having a bitvector of length n, with m 1s and k rank samples.

Variant	**Additional Bits**	**Time Complexity**
Plain bitvectors [43]	$0.0625 \cdot n$	$\mathcal{O}(1)$
Interleaved bitvectors [43]	128	$\mathcal{O}(1)$
H_0*-compressed bitvectors* [44]	80	$\mathcal{O}(k)$
Sparse bitvectors [45]	64	$\mathcal{O}\left(\log \frac{n}{m}\right)$

Table 2.3: Time complexity and amount of additional bits of the select function for some succinct bitvectors variants having a bitvector of length n.

Variant	**Additional Bits**	**Time Complexity**
Plain bitvectors [43]	$\leq 0.2 \cdot n$	$\mathcal{O}(1)$
Interleaved bitvectors [43]	64	$\mathcal{O}(\log n)$
H_0*-compressed bitvectors* [44]	64	$\mathcal{O}(\log n)$
Sparse bitvectors [45]	64	$\mathcal{O}(1)$

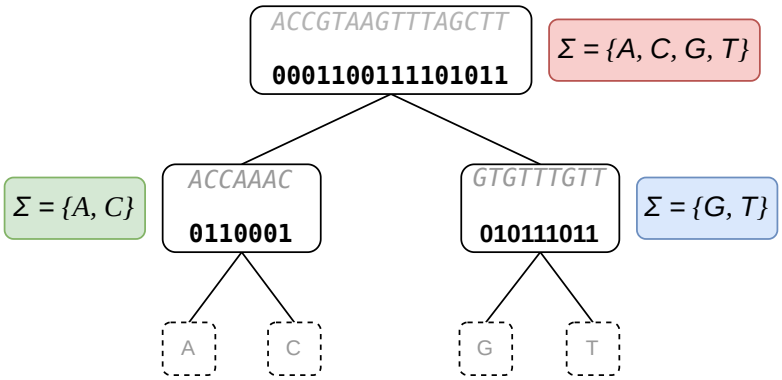


Figure 2.1: Wavelet tree of the Example 2. Note that the gray sequences and the leaves are not stored in memory.

Wavelet trees Wavelet trees, introduced in 2003 by Grossi et al., generalize the three succinct bitvector operations (access, rank, and select) to a string built over an arbitrary ordered alphabet Σ. The main idea of indexing a string T built over Σ is considering a perfectly balanced tree where each node is associated with a subset of Σ. In detail, given a node v and the correspondent $\Sigma_v \subseteq \Sigma$, we associated half of Σ_v to each child, hence the depth of the tree is given by the number of times we can split the alphabet in two, *i.e.* $\log(|\Sigma|)$. The root of the tree considers the entire alphabet Σ. Moreover, each node v of this tree should be augmented with a sequence R_v, a subsequence of T based on Σ_v. The root of the tree takes into account T, while leaves are labeled with all the symbols in Σ.

To build a succinct data structure, we can store these subsequences as succinct bitvectors, with "0"s for every σ in the first half of Σ_v and "1"s otherwise. In detail, we store these succinct bitvectors and not the subsequences. See Example 2 to clarify how a wavelet tree is stored.

Example 2. *Consider the string $T = ACCGTAAGTTTAGCTT$, having $n = |T| = 16$, $\Sigma = \{A, C, G, T\}$, and $|\Sigma| = 4$. Figure 2.1 shows the wavelet tree of T.*

In this context, $\mathsf{access}(T, i)$ returns the symbol $T[i]$, $\mathsf{rank}_\sigma(T, i)$ counts how many symbols σ occurs in T before position i excluded, and $\mathsf{select}_\sigma(T, i)$ returns the position of the i-th occurrence of the symbol σ in T.

Assuming access, rank, and select in $\mathcal{O}(1)$ on the succinct bitvectors stored in the wavelet tree and assuming to have constant time random access to the

alphabet associated with each node, we can perform access, rank, and select to a string T built Σ traversing the tree and querying the succinct bitvectors. Hence, all these three functions can be answered in $\mathcal{O}(\log|\Sigma|)$ time string storing $o(n\log|\Sigma|)$ additional bits for the succinct bitvectors. Note that a wavelet tree can be built in $\mathcal{O}(n\log|\Sigma|)$ time. If the alphabet size is not a power of two, causing $\log|\Sigma|$ not to be an integer and the tree not to be balanced, we can add dummy symbols to Σ until we get its size to be the next power of two. Note that wavelet tree complexities can vary according to the underlying variant of the succinct bitvectors.

In addition to [47] and the already cited books [46, 42], we also suggest [48] for further explanations on wavelet trees. Moreover, we highlight further improvements on this topic, such as the levelwise wavelet trees [49, 50] and the wavelet matrices [51].

2.3 Straight-line programs

In the context of grammar-based compression, a Straight Line Program (SLP) [52, 53, 54] is a context-free grammar that generates exactly one string.

Definition 3 (Straight Line Program [52])**.** *Given a n-length string T built over a finite alphabet Σ, the* SLP *of T is a context free grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{S}, \mathcal{P})$ where: $\mathcal{V}$ is the set of non-terminal symbols, Σ is the set of terminal symbols, $\mathcal{S} \in \mathcal{V}$ is the initial non-terminal symbol, and $\mathcal{P} \subseteq \mathcal{V} \times (\mathcal{V} \cup \Sigma)^*$ is the productions set. To be $\mathcal{G}$ an* SLP*, we require that: a) for each $\mathcal{A} \in \mathcal{V}$ and $\alpha \in (\mathcal{V} \cup \Sigma)^*$ we have one and only one production $\mathcal{A} \to \alpha \in \mathcal{P}$, and b) the relation $\{(\mathcal{A}, b) | \, \mathcal{A} \to \alpha \in \mathcal{P}, b \in \{T[1], T[2], \cdots, T[n]\}\}$ is acyclic.*

Given an SLP $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{S}, \mathcal{P})$, we denote its size $|\mathcal{G}|$, *i.e.* $|\mathcal{G}| = \sum_{\mathcal{A}\to\alpha\in\mathcal{P}} |\alpha|$ and we denote $eval_{\mathcal{G}}(T)$ the single word generated by the SLP $\mathcal{G}$ from the text T. The core idea behind accessing an SLP $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{S}, \mathcal{P})$ of a string T is to build a finite rooted ordered tree, namely a derivation tree, in which: a) the root is labeled with T, b) each internal node is labeled with a symbol $\alpha \in (\mathcal{V} \cup \Sigma)^*$, and c) each leaf is labeled with a symbol $\sigma \in \Sigma$. Example 3 [53] shows a brief example of an SLP and its derivation tree.

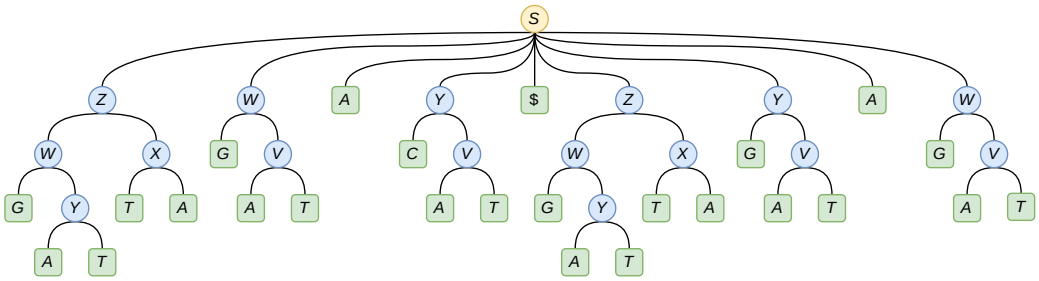


Figure 2.2: Derivation tree of the SLP proposed in the Example 3. Note that in the yellow circle node, we have the initial non-terminal symbol; in the blue circle nodes, the non-terminal symbols; and in the green square nodes, the terminal symbols.

Example 3. *Consider the string* $T = GATTAGATACAT\$GATTACATAGAT$. *We build a corresponding* SLP $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{S}, \mathcal{P})$ *where* $\mathcal{V} = \{S,Z,Y,X,W,V\}$, $\Sigma = \{A,C,G,T,\$\}$, $\mathcal{S} = S$, *and* $\mathcal{P} = \{S \to ZWAY\$ZYAW, Z \to WX, Y \to CV, X \to TA, W \to GV, V \to AT\}$. *Figure 2.2 shows the derivation tree of* $\mathcal{G}$.

The next Lemma [55] is needed for the complexity analysis of an SLP.

Lemma 1. *Given a* n*-length string* T *and the corresponding* SLP $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{S}, \mathcal{P})$, *we can compute in* $\mathcal{O}(|\mathcal{G}|)$ *time an equivalent* SLP $\mathcal{G}' = (\mathcal{V}', \Sigma, \mathcal{S}', \mathcal{P}')$ *of size* $\mathcal{O}(|\mathcal{G}|)$ *with a derivation tree of height* $\mathcal{O}(\log n)$.

Thanks to Lemma [55], given a n-length text T represented by the corresponding SLP $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{S}, \mathcal{P})$, we can perform random access in the i-th position of T through $\mathcal{G}$, *i.e.* $\mathsf{access}_{\mathcal{G}}(T, i)$, in $\mathcal{O}(\log n)$ time, traversing the tree from the root to the leaf which encode the symbol $T[i]$.

Moreover, we can leverage the use of an SLP to compute the longest common extension between two positions of a string T represented by an SLP $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{S}, \mathcal{P})$, $\mathsf{LCE}_{\mathcal{G}}(T, i, j)$, in $\mathcal{O}(\log n)$ time. In detail, after having performed $\mathsf{access}_{\mathcal{G}}(T, i)$ and $\mathsf{access}_{\mathcal{G}}(T, j)$ in $\mathcal{O}(\log n)$ time, we can look for the common subpath starting from these two leaves and the root. The height of the tree, $\log n$, bounds the total number ℓ of back steps on the tree, hence, assuming each back step requires constant time, we can compute $\mathsf{LCE}_{\mathcal{G}}(T, i, j)$ in $\mathcal{O}(\log n + \ell) = \mathcal{O}(\log n)$ time.

We refer to [52, 53, 54] for further details and additional theoretical results.

2.4 String-to-string pattern-matching

String-to-string pattern matching is one of the most studied problems in computer science [56], and it is fundamental in various research fields, including bioinformatics and pangenomics.

Given an n-length string T, namely the text, and an m-length string P, namely the pattern, we are interested in locating all the positions where an occurrence of P in T starts, assuming $m < n$. In this thesis, we will focus on some generalizations of the string-to-string pattern matching problem, in which we do not look just for the entire occurrences P in T. Hence, we are also interested in substrings of P that match a substring of T.

Maximal exact matches and matching statistics Maximal Exact Matches (MEMs) finding problem is one of these string-to-string pattern-matching generalized problems. Given two strings T and P, respectively of length n and m, with $m < n$, a maximal exact match is a common substring between T and P that can not be extended to the left or to the right without introducing a mismatch, being, respectively, both a left-maximal match and a right-maximal match.

Definition 4 (Maximal exact match)**.** *Given a n-length string T and m-length string P, with $m < n$, we define a Maximal Exact Match of P in T a substring of P of length ℓ which starts in position i, i.e. $P[i, i+\ell-1]$, such that: a) $P[i, i+\ell-1]$ occurs in T, b) $P[i-1, i+\ell-1]$ does not occur in T, being $P[i, i+\ell-1]$ a left-maximal match, and c) $P[i, i+\ell]$ does not occur in T, being $P[i, i+\ell-1]$ a right-maximal match.*

In literature, various methods exist to compute MEMs; however, in this thesis, we will leverage the strong connection between maximal matches and the concept of Matching Statistics (MS) [57].

Definition 5 (Matching Statistics)**.** *Given a n-length string T and m-length string P, with $m < n$, we define Matching Statistics as an array of pairs* $(\mathsf{pos}, \mathsf{len})$ *of size m such that for each $1 \leq i \leq n$: a) we have a match of length* $\mathsf{MS}[i].\mathsf{len}$ *shared between the suffixes* $T[\mathsf{MS}[i].\mathsf{pos}, n]$ *and $P[i..m]$, i.e.* $T[\mathsf{MS}[i].\mathsf{pos}, \mathsf{MS}[i].\mathsf{pos}+\mathsf{MS}[i].\mathsf{len}-1] =$

$P[i, i + \mathsf{MS}[i].\mathsf{len} - 1]$*, and b) this match is right-maximal, i.e.* $P[i..i + \mathsf{MS}[i].\mathsf{len}]$ *does not occur in* T*.*

The strong correlation between MEMs and MS is well summarized in the next lemma [58].

Lemma 2. *Given two strings* T *and* P*, respectively of length* n *and* m*, with* $m < n$*, and the correspondent* MS *array, the* ℓ*-length substring* $P[i, i + \ell - 1]$ *is a* ℓ*-length* MEM *if and only if* $\mathsf{MS}[i].\mathsf{len} = l \wedge \mathsf{MS}[i - 1].\mathsf{len} \leq \mathsf{MS}[i].\mathsf{len}$*. In addition, we have the special case of a 1-length* MEM *starting in the first position of* P*, which occurs if and only if* $\mathsf{MS}[1].\mathsf{len} = 1 \wedge \mathsf{MS}[1].\mathsf{len} \geq \mathsf{MS}[2].\mathsf{len}$*.*

Finally, we want to recall that MEMs are used as a starting point for aligning strings, in the "seed and extend" paradigm [59, 60, 61, 15]. This technique leverages the computation of short MEMs, the "seeds", to reduce the need for dynamic programming, which is used only to "extend" these matches. Note that a pure dynamic programming approach is infeasible due to its high memory requirements.

For other details on the string-to-string pattern matching problem, please refer to Gusfield's "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology" book [56].

2.5 String indexing

To understand how to compute MS and then MEMs, we must introduce another fundamental research field: text indexing. Consider the classical scenario of the string-to-string pattern matching problem: we have an m-length pattern P, and we want to find its occurrences in a longer n-length text T, where both T and P are built over the alphabet Σ. The key idea is to preprocess T and build an additional data structure that we store in memory to improve query time. This data structure is referred to as the full-text index, or simply the index, of T.

In this context, to resolve specific ambiguity issues, we append a unique character to the text T. Typically, this special symbol is the dollar sign, which does not belong to the text alphabet and is lexicographically smaller than any other symbol in it, *i.e.* $\$ \notin \Sigma$ and $\$ \prec \sigma, \forall \sigma \in \Sigma$. We refer to this new string $T \cdot \$$ as a

dollar-terminated string. We refer to a dollar-terminated string T of length n if we are referring to a text T of length $n-1$ concatenated with the symbol \$ with the properties described above.

For further details on the text indices (including the ones not discussed in this thesis, such as the suffix trees), please refer to [56, 46, 42].

2.5.1 Suffix array and related data structures

The first way to index a text we present is the suffix array (SA) [62, 63], introduced as a light in-memory alternative to suffix trees [64, 65] to store and query all the suffixes of a string T.

Definition 6 (Suffix array)**.** *Given a dollar-terminated string T of length n, the suffix array of T, denoted SA_T, is a n-length array such that $\mathsf{SA}_T[i] = j$ if and only if the j-th suffix $T[j..n]$ is the i-th element in the list of the lexicographically ordered suffixes of T.*

Informally, the suffix array contains the starting positions of the suffixes of a string in lexicographical order, being a permutation of the values in $\{1, 2, \cdots, n\}$; hence, it takes $\mathcal{O}(n \log n)$ bits to be stored. Note that, by definition, given two ordered positions i and j such that $1 \leq i < j \leq n$, their correspondent suffixes are ordered, *i.e.* $T[\mathsf{SA}_T[i]..n] \prec T[\mathsf{SA}_T[j]..n]$.

Similarly to the suffix array, we can define the Inverse Suffix Array (ISA) of T as an n-length array, ISA_T, such that for each $1 \leq i \leq n$, $ISA_T[i] = j$ if and only if $SA_T[j] = i$. In other words, the ISA is the inverse permutation of SA.

Using the suffix array, we can solve the string-to-string pattern matching problem, as described before, using a binary search in $\mathcal{O}(m \log n + k)$ time, where k is the number of occurrences of P in T.

Another SA variant that is interesting to introduce is the Compressed Suffix Array (CSA) [12]. The idea behind the CSA is to replace the SA_T with a function Ψ_T, which is called the successor function. This function is defined as $\Psi_T(i) = \mathsf{ISA}_T[\mathsf{SA}_T[i] + 1]$, having that $\mathsf{SA}_T[\Psi_T(i)] = SA_T[i] + 1$. Equally, we have that $\Psi_T(i) = j$ if and only if $\mathsf{SA}_T[j] = \mathsf{SA}_T[i] + 1$. Informally, this function can determine where the successor suffix is in the lexicographic order and can be used

to reconstruct the suffix array in $\mathcal{O}(n)$ time. Moreover, Ψ_T can be compressed using Δ-encoding [66] or Elias gamma encoding [67], thanks to the fact that we obtain an array that we can split into monotonously increasing regions. Storing the CSA in $\mathcal{O}(n \log |\Sigma|)$ bits, the algorithm that solves the string-to-string pattern matching problem requires $\mathcal{O}(m \log^2 n)$ time. We refer to [12] for further theoretical improvements in using the CSA.

Example 4 shows the suffix array, the inverse suffix array, and the Ψ function of the compressed suffix array of a dollar-terminated string T.

φ functions In this thesis, we will leverage the φ functions to explore the suffix array. Given a suffix array value, φ functions retrieve the adjacent values in SA.

Definition 7 (φ Function)**.** *Given a dollar-terminated string T of length n and the correspondent suffix array SA_T, we define the function φ_T, $\forall\, i \in [1..n]$, as: a) $\varphi_T(i) = \mathsf{null}$ if $\mathsf{ISA}_T[i] = 1$, or b) $\varphi_T(i) = SA_T[\mathsf{ISA}_T[i] - 1]$ otherwise.*

Informally, given $1 \leq i \leq n$, $\varphi(i)$ returns the value that precedes i in SA if it exists, *i.e.* $\varphi_T(\mathsf{SA}_T[i]) = \mathsf{SA}_T[i-1]$. Similarly, we can retrieve the value that succeeds a given SA value.

Definition 8 (φ^{-1} Function)**.** *Given a dollar-terminated string T of length n and the correspondent suffix array SA_T, we define the function φ_T^{-1}, $\forall\, i \in [1..n]$, as: a) $\varphi_T^{-1}(i) = \mathsf{null}$ if $\mathsf{ISA}_T[i] = n$, or b) $\varphi_T^{-1}(i) = \mathsf{SA}_T[\mathsf{ISA}_T[i] + 1]$ otherwise.*

Informally, given $1 \leq i \leq n$, $\varphi(i)$ returns the value that succeeds i in SA if it exists, *i.e.* $\varphi_T^{-1}(\mathsf{SA}_T[i]) = \mathsf{SA}_T[i+1]$. We refer to φ and φ^{-1} together as φ functions. Example 4 shows the result of the φ functions for a dollar-terminated string T.

Longest common prefix array The Longest Common Prefix (LCP) array [56] was introduced to lower the time bounds of solving the string-to-string pattern matching problem using the suffix array.

Definition 9 (Longest Common Prefix)**.** *Given a dollar-terminated string T of length n and the correspondent suffix array SA_T, the longest common prefix of T, LCP_T, is a n-length array such that: a) $\mathsf{LCP}_T[1] = \mathsf{null}$, or b) for each $1 < i \leq n$, $\mathsf{LCP}_T[i] = \big|\mathsf{lcp}(T[\mathsf{SA}_T[i-1], n], T[\mathsf{SA}_T[i], n])\big|$.*

Informally, the longest common prefix array stores the length of the longest common prefix shared between the i-th suffix in the colexicographical order and its predecessor. Note that the suffix array directly gives this order.

Using the LCP array, we can improve the time complexity of the SA algorithm, solving the string-to-string pattern-matching problem in $\mathcal{O}(m + \log n)$ time.

In literature, various results have been obtained on the succinct representation and compression of the LCP [68, 69, 70], and one of the most important is the Permuted Longest Common Prefix (PLCP) array. Similarly to the LCP, the PLCP of a dollar-terminated string T is a n-length array PLCP_T such that for each $1 \leq i \leq n$: a) $\mathsf{PLCP}_T[i] = \mathsf{null}$ if $\mathsf{ISA}_T[i] = 1$, or b) $\mathsf{PLCP}_T[i] = \mathsf{LCP}_T[\mathsf{ISA}_T[i]]$ if $\mathsf{ISA}_T[i] \neq 1$. The PLCP is a permutation of the LCP values based on the original suffix order and not on the suffix lexicographic order, having that $\mathsf{PLCP}_T[\mathsf{SA}_T[i]] = \mathsf{LCP}_T[i]$. In [70] was proved the high compressibility of the PLCP, proving that $\mathsf{PLCP}_T[i] \geq \mathsf{PLCP}_T[i-1] - 1$, for each $1 \leq i \leq n$, allowing a sparse representation of the PLCP.

Example 4 shows the longest common prefix and the permuted longest common prefix of a dollar-terminated string T.

Example 4. *Consider the dollar-terminated string T = MISSISSIPPI\$. In Table 2.4, we show the suffix array, the inverse suffix array, the phi functions, the longest common prefix, and the permuted longest common prefix of T.*

2.5.2 Burrows–Wheeler Transform

Introduced in 1994 by Burrows and Wheeler, the Burrows–Wheeler Transform (BWT) [10] is a reversible permutation of a string. At first, the BWT was used as a lossless compression algorithm, but in the last two decades, its use has been extended to the string-to-string pattern-matching context.

Definition 10 (Burrows–Wheeler Transform)**.** *Given a dollar-terminated string T of length n and the correspondent suffix array SA_T, the Burrows–Wheeler Transform of T, BWT_T, is a n-length array such that: a) $BWT_T[i] = \$$, if $\mathsf{SA}_T[i] = 1$ or b) $BWT_T[i] = T[\mathsf{SA}_T[i] - 1]$, if $\mathsf{SA}_T[i] \neq 1$.*

Note that the original definition of BWT in [10] was given for a general string without a terminal symbol.

Table 2.4: Suffix array, inverse suffix array, phi functions, longest common prefix, and permuted longest common prefix of the string T in Example 4. In the last column, we underline the longest common prefix between two consecutive suffixes in the lexicographic order.

Index	SA_T	ISA_T	Ψ_T	φ	φ^{-1}	LCP_T	PLCP_T	**Suffix**
1	12	6	null	2	10	null	0	\$
2	11	5	1	5	1	0	4	I\$
3	8	12	8	6	null	1	3	IPPI\$
4	5	10	11	7	6	1	2	ISSIPPI\$
5	2	4	12	8	2	4	1	ISSISSIPPI\$
6	1	11	5	4	3	0	1	MISSISSIPPI\$
7	10	9	2	9	4	0	0	PI\$
8	9	3	7	11	5	1	1	PPI\$
9	7	8	3	10	7	0	1	SIPPI\$
10	4	7	4	1	9	2	0	SISSIPPI\$
11	6	2	9	12	8	1	0	SSIPPI\$
12	3	1	10	null	11	3	null	SSISSIPPI\$

Informally, the i-th value of BWT_T is the symbol that precedes the i-th suffix in the colexicographical order. Practically, a naïve method to compute BWT_T is to construct Burrows–Wheeler Matrix (BWM), which contains all the rotations of T, *i.e.* the set $\{\mathsf{rot}_k(T) | \, 1 \leq k \leq n\}$, in lexicographic order. The last column of the BWM is BWT_T. Example 5 shows all these arrays for a dollar-terminated string.

Moreover, the first column of the BWM is called the F_T array and contains all the symbols of T in lexicographic order. Note that the lexicographic order of all the rotations of a string T is the same order induced by SA_T.

The BWT of a dollar-terminated string of length n can be computed in $\mathcal{O}(n)$ time using the suffix array. Note that also the BWT of a general string can be computed in linear time, as proved in [71].

Example 5. *Consider the dollar-terminated string T = MISSISSIPPI\$. In Table 2.5 we show BWT_T, BWM_T, and F_T along with the suffix array of T.*

Recalling that the BWT is a reversible permutation, we introduce a BWT property called LF-Mapping. This property ensures that the i-th occurrence of a symbol in BWT_T corresponds in T to the i-th occurrence of the same symbol in F_T. For example, considering the text in Example 5 and Table 2.5, the third occurrence of the symbol "I" in BWT_T, *i.e.* $\mathsf{BWT}_T[11]$, is $T[5]$, which is the same symbol we

Table 2.5: Suffix array, Burrows–Wheeler Transform, Burrows–Wheeler Matrix, and F array of the string T in Example 5.

Index	SA_T	F_T	BWM_T	BWT_T
1	12	\$	\$MISSISSIPPI	I
2	11	I	I\$MISSISSIPP	P
3	8	I	IPPI\$MISSISS	S
4	5	I	ISSIPPI\$MISS	S
5	2	I	ISSISSIPPI\$M	M
6	1	M	MISSISSIPPI\$	\$
7	10	P	PI\$MISSISSIP	P
8	9	P	PPI\$MISSISSI	I
9	7	S	SIPPI\$MISSIS	S
10	4	S	SISSIPPI\$MIS	S
11	6	S	SSIPPI\$MISSI	I
12	3	S	SSISSIPPI\$MI	I

have as the third occurrence of the symbol "I" in F_T ($\mathsf{F}_T[4]$). Summarizing, $T[5]$, $\mathsf{BWT}_T[11]$, and $\mathsf{F}_T[4]$ are exactly the same symbol "I". This property is called LF-Mapping, *i.e.* Last-First mapping, because it maps the symbols in the last column of the BWM to the first one.

Given the LF-mapping, we can reconstruct a dollar-terminated string T starting from BWT_T, which we recall to be a reversible transform. The procedure starts from the last symbol of T, the \$ in our context. By Definition 10, this symbol is preceded in T by the symbol $\mathsf{BWT}_T[1]$, having $\mathsf{F}_T[1] = \$$ by construction. We can now locate the symbol $\mathsf{BWT}_T[1]$ in the F_T array using the LF-mapping property and look at the symbol in BWT_T at the same position to discover which symbol precedes $\mathsf{BWT}_T[1]$. We iterate this procedure until we reach the symbol \$ in BWT_T which indicates we have reconstructed the entire string T, having that \$ precedes the first symbol of T if we consider $\mathsf{rot}_1(T)$

Pattern matching with the BWT and the FM-Index We can exploit the LF-mapping property to solve the string-to-string pattern matching problem. This technique, known as backward-search, combines accesses to BWT_T and SA_T. The main idea is to scan the pattern from right to left and to update a suffix array interval, called Q-interval, which represents all the text suffixes that match a pattern suffix Q. Each update, *i.e.* computing the σQ-interval starting from the Q-interval, is called a backward step and exploits the LF-mapping. At the end of

the backward search, which begins with $Q = \varepsilon$ and iteratively considers increasingly longer suffixes of the pattern, we obtain a Q-interval corresponding to $Q = P$. This interval in the suffix array contains all positions in T where an occurrence of P begins. If any σQ-interval is empty during the search, it indicates that the pattern does not occur in the text.

To explain how we leverage the LF-mapping to update the Q-interval, we first need to introduce the FM-index [11], a text index that is built over the BWT and allows the computation of the LF-mapping in constant time. The FM-index consists of two functions, C and Occ, which can completely replace the BWT itself. Thanks to this property, we can refer to the FM-index as a self-index.

Definition 11 ($\mathsf{C}(\sigma)$ function)**.** *Given a n-length string T built over an ordered alphabet Σ and a symbol $\sigma \in \Sigma$, we define the function $\mathsf{C} : \Sigma \cup \{\$\} \to \{0, n\}$ which returns how many symbols in T are lexicographically smaller than σ.*

Definition 12 ($\mathsf{Occ}(\sigma, i)$ function)**.** *Given a n-length string T built over an ordered alphabet Σ, a symbol $\sigma \in \Sigma$, and an integer $1 \leq i \leq n$, we define the function $\mathsf{Occ} : (\Sigma \cup \{\$\}) \times \{1, n+1\} \to \{0, n\}$ which returns how many symbols σ are present in the $(i-1)$-th prefix of T.*

In Example 6, we show a tabular representation of the two FM-index functions. In this example, we directly apply these two functions to the BWT of the given text, indexing the BWT string.

Example 6. *Consider the dollar-terminated string $T = MISSISSIPPI\$$. In Figure 2.3.a are represented all the results of the function $\mathsf{C}(\sigma)$ and in Table 2.3.b all the results of the function $\mathsf{Occ}(\sigma, i)$ for $\mathsf{BWT}_T = IPSSM\$PISSII$.*

Naïvely, we can store the two tables representing the C and the Occ function respectively in $\mathcal{O}(|\Sigma|)$ and in $\mathcal{O}(n|\Sigma|)$ space. In this context, we can retrieve the results of these two functions in $\mathcal{O}(1)$ time. Practically, the space requirement for the Occ function is too high to index long strings (such as genomes), so typically, succinct data structures such as the wavelet trees (described in Section 2.2) are used to perform the Occ function over a text, *i.e.* the rank function, in $\mathcal{O}(\log |\Sigma|)$ time. Additionally, all the $\mathsf{C}(\sigma)$ values can be retrieved in $\mathcal{O}(\Sigma)$ time scanning the $\mathsf{Occ}(\sigma, n+1)$ values for all the symbols $\sigma \in \Sigma$.

σ	$\mathsf{C}(\sigma)$
\$	0
I	1
M	5
P	6
S	8

(a)

1	0	0	0	0	0
2	0	1	0	0	0
3	0	1	0	1	0
4	0	1	0	1	1
5	0	1	0	1	2
6	0	1	1	1	2
7	1	1	1	1	2
8	1	1	1	2	2
9	1	2	1	2	2
10	1	2	1	2	3
11	1	2	1	2	4
12	1	3	1	2	4
13	1	4	1	2	4
i/σ	\$	I	M	P	S

(b)

Figure 2.3: Tabular representation of the results of (a) $\mathsf{C}(\sigma)$ and (b) $\mathsf{Occ}(\sigma, i)$ function for Example 6.

Moreover, note that the function $\mathsf{C}(\sigma)$ can be used to determine both the number of suffixes that start with a symbol lexicographically smaller than σ and the maximum position in SA_T of a suffix beginning with a symbol lexicographically smaller than σ.

Combining these two functions, we can rewrite the LF-mapping for BWT_T and $1 \leq i \leq |\mathsf{BWT}_T|$ as $\mathsf{LF}(i) = \mathsf{C}(\mathsf{BWT}_T[i]) + \mathsf{Occ}(\mathsf{BWT}_T[i], i)$. In addition, we can efficiently compute the backward-step in $\mathcal{O}(1)$ or $\mathcal{O}(\log |\Sigma|)$ time, respectively using a naïve representation or a wavelet tree for the Occ function. In detail, given a n-length dollar-terminated string built over an ordered alphabet Σ, a symbol $\sigma \in \Sigma$ and a Q-interval $[f, g)$, the σQ-interval $[f', g')$ is equal to $[\mathsf{C}(\sigma) + \mathsf{Occ}(\sigma, f), \mathsf{C}(\sigma) + \mathsf{Occ}(\sigma, g))$. In this way, we can solve the string-to-string pattern matching problem in $\mathcal{O}(|P|)$ or $\mathcal{O}(|P| \log |\Sigma|)$, depending on the data structure we store to compute the Occ function.

BWT for a collection of strings All the concepts described above related to the suffix array and the Burrows-Wheeler Transform can be extended to consider a collection of strings. In the literature, various strategies for indexing multiple

strings are available, including building a concatenation of those strings, separated by a single dollar symbol or multiple dollar symbols. The key idea is that if we separate each string with a symbol not present in the pattern alphabet (*i.e.* the dollar), we avoid false matches across multiple strings. We refer to this variant as Multidollar-BWT.

Moreover, recent results explore ordering strategies that differ from the classical lexicographic order and demonstrate how they can be used to construct the BWT of multiple strings with specific properties, such as minimizing the number of runs in the resulting BWT [72].

We refer to [73] for further explanations on modern techniques to index a collection of strings using the BWT.

Run-Length encoded BWT and r-index The Burrows-Wheeler Transform was initially developed as a technique for compressing text. The compression capabilities of the BWT are related to an implicit behavior of this transform: in the BWT, equal symbols tend to be in consecutive positions. In detail, this behavior is caused by repetitions in the text, which are often preceded by the same symbol in the input text. Given these premises, the use of a run-length encoding for BWT has been explored in the pattern-matching literature, with the Run-Length Encoded BWT (RLBWT) and the Run-Length Encoded FM-Index (RLFM-index) [74, 75, 76, 77, 78]. Those results proved that we can count all the occurrences of an m-length pattern P in an n-length text T in $\mathcal{O}(m \log r)$ and $\mathcal{O}(r)$ space, having r runs in the RLBWT of the input text. Note that we can locate these occurrences within those time and space boundaries.

In 2018, Gagie, Navarro, and Prezza [79] proposed a further improvement for indexing the RLBWT, called r-index. Briefly, the r-index consists of the RLBWT, and a $\mathcal{O}(r)$ space sampling of the suffix array. The RLBWT is stored as a string that contains the concatenation of the run symbols and a $|\mathsf{BWT}|$-length bitvector to store the position of the runs over the BWT. This augmentation builds on previous results [80], where Policriti et al. first enhanced an LZ77-based [81] index for the RLBWT by introducing an $\mathcal{O}(r)$-space sampling of the suffix array. Using the so-called Toehold Lemma, later refined and simplified in [58], they proved that the SA interval containing all occurrences of P can be computed in $\mathcal{O}(m \log \log n)$

time. In [58], Bannai et al. proposed using the matching statistics in combination with the RLBWT and the r-index.

Finally, in 2020, Gagie et al. [82] proved that we can locate all the k occurrences of P in T in $\mathcal{O}(k \log \log n)$ time and $\mathcal{O}(r)$ space. Moreover, raising the data structures space to $\mathcal{O}(r \log \log n)$, we can perform the locate query in $\mathcal{O}(m + k)$. In this case, the SA sampling consists of storing the SA values in the positions that are at run boundaries in the BWT.

In addition to the already cited articles, we refer to [83, 84, 85, 86] for further details, including related techniques such as the prefix parsing approach for building the RLBWT, and the r-index a collection of strings.

MONI and PHONI In this paragraph, we want to recall two implementations of the RLBWT and the r-index: MONI [15] and PHONI [16], as they strongly inspired some methods in this thesis. These two tools compute MEMs from the MS array, but their approaches differ. MONI requires storing, for each pair of consecutive runs of the same symbol, the minimum LCP value between them, referred to as a threshold, and performing random access to the text, following the algorithm proposed in [58]. The procedure is shown in Algorithm 2.1. On the other hand, PHONI requires a data structure that can perform LCE queries on the input text, for instance by storing an SLP of the text. Thanks to the LCE queries, PHONI can compute pos and len values of the MS array in a single scan, while MONI requires two scans. The procedure is shown in Algorithm 2.2.

Assuming for a n-length text T to have thresholds access in $\mathcal{O}(\log \log n)$ time [58], random access to T in $\mathcal{O}(\log \log n)$ time [87], access to φ and φ^{-1} values in $\mathcal{O}(\log \log n)$ time [16], and backward steps in $\mathcal{O}(\log \log n)$ time [88], MONI can compute the MEMS shared with an m-length pattern in $\mathcal{O}(m \log \log n)$ time. On the other hand, the algorithm for computing MEMs with PHONI is bounded by the computation of LCE queries. Assuming to have an SLP which can answer LCE queries in $\mathcal{O}(\log n)$ [53, 89], PHONI can compute the MEMS shared with an m-length pattern in $\mathcal{O}(m \log n)$ time.

Algorithm 2.3 is used to leverage the φ and φ^{-1} functions and the PLCP to get all starting positions of the occurrences of P in T in $\mathcal{O}(\log \log n)$ time per occurrence [16]. We refer to [15, 16, 58, 14] for further details and the pseudocode.

Algorithm 2.1 MS array computation leveraging the use of the thresholds [58, 15].

function COMPUTE_MS(T, SA_T, BWT_T, P) ▷ $|T| = n$ and $|P| = m$
  $\mathsf{MS} \leftarrow [(\mathsf{pos} : 0, \mathsf{len} : 0) \dots (0, 0)]$ ▷ $|\mathsf{MS}| = m$
  $q \leftarrow$ position of the last occurrence of $P[m]$ in BWT_T
  $pos \leftarrow \mathsf{SA}_T[q]$
  for $i = 1$ **to** m **do** ▷ Computation of pos values of the matching statistics
    if $\mathsf{BWT}_T[q] \neq P[i]$ **then** ▷ Select the new position using the thresholds
      if $\mathsf{BWT}_T[q]$ is before the threshold between the consecutive runs of $P[i]$ **then**
        $q \leftarrow$ position of the preceding occurrence of $P[i]$ in BWT_T
      else
        $q \leftarrow$ position of the following occurrence of $P[i]$ in BWT_T
      $pos \leftarrow \mathsf{SA}_T[q]$
    $\mathsf{MS}[i].\mathsf{pos} \leftarrow pos$
    $q \leftarrow \mathsf{LF}(q)$, $pos \leftarrow pos - 1$
  for $i = 1$ **to** m **do** ▷ Computation of len values of the matching statistics
    $\mathsf{MS}[i].\mathsf{len} \leftarrow \mathsf{MS}[i-1].\mathsf{len} - 1$
    while $P[i + \mathsf{MS}[i].\mathsf{len}] = T[\mathsf{MS}[i].\mathsf{pos} + \mathsf{MS}[i].\mathsf{len}]$ **do**
      $\mathsf{MS}[i].\mathsf{len} \leftarrow \mathsf{MS}[i].\mathsf{len} + 1$
  return MS

Algorithm 2.2 MS array computation leveraging the use of the LCE queries [16].

function COMPUTE_MS(T, SA_T, BWT_T, P) ▷ $|P| = m$, $|T| = n$
  $\mathsf{MS} \leftarrow [(\mathsf{pos} : 0, \mathsf{len} : 0) \dots (0, 0)]$ ▷ $|\mathsf{MS}| = m$
  $q \leftarrow$ position of the last occurrence of $P[m]$ in BWT_T
  $\mathsf{MS}[m-1] \leftarrow (\mathsf{SA}_T[q] - 1, 1)$, $q \leftarrow \mathsf{LF}(q)$
  for $i = m - 1$ **down to** 1 **do** ▷ Computation of the matching statistics
    if $\mathsf{BWT}_T[q] = P[i]$ **then**
      $\mathsf{MS}[i] \leftarrow (\mathsf{MS}[i+1].\mathsf{pos} - 1, \mathsf{MS}[i+1].\mathsf{len} + 1)$, $q \leftarrow \mathsf{LF}(q)$
    else ▷ Select the new position using the longest LCE query
      $q' \leftarrow$ position of the preceding occurrence of $P[i]$ in BWT_T
      $q'' \leftarrow$ position of the following occurrence of $P[i]$ in BWT_T
      $l' \leftarrow \min(\mathsf{MS}[i+1].\mathsf{len}, |\mathsf{LCE}(\mathsf{SA}_T[q'], \mathsf{MS}[i+1].\mathsf{pos})|)$
      $l'' \leftarrow \min(\mathsf{MS}[i+1].\mathsf{len}, |\mathsf{LCE}(\mathsf{SA}_T[q''], \mathsf{MS}[i+1].pos)|)$
    if $l' \geq l''$ **then**
      $\mathsf{MS}[i] \leftarrow (\mathsf{SA}_T[q'] - 1, l' + 1)$, $q \leftarrow \mathsf{LF}(q')$
    else
      $\mathsf{MS}[i] \leftarrow (\mathsf{SA}_T[q''] - 1, l'' + 1)$, $q \leftarrow \mathsf{LF}(q'')$
  return MS

Algorithm 2.3 MEM occurrences starting positions computation from MS [16].

```
1: function GET_OCCURENCES(MS, i, j, P, T)   ▷ All the occurrences of P[i..j] in T
2:     if MS[i].len < j − i + 1 then
3:         return
4:     p ← MS[i].pos
5:     occ ← [ ]
6:     append(occ, p)
7:     while PLCP[p] ≥ j − i + 1 do
8:         p ← φ(p)
9:         append(occ, p)
10:    p ← φ^{-1}(MS[i].pos)
11:    while p ≠ null ∧ PLCP[p] ≥ j − i + 1 do
12:        append(occ, p)
13:        p ← φ^{-1}(p)
14:    return occ
```

2.6 String-to-matrix pattern-matching

In this section, we use (binary) matrices, sets of (binary) sequences, and (haplotypes) panel interchangeably; hence, in this context "sequence" and "row" are synonymous. For simplicity, in formal definitions we will primarily refer to sets of sequences, using the term "column" to denote the bits occupying the same position across all these sequences. We also recall that these sets can be multisets.

In the last decades, pattern-matching problems have been generalized to other contexts, such as considering the string-to-matrix pattern matching problem [13, 90, 91, 92, 93].

In this scenario, we consider a set S of h strings of length of w: $S = \{s_1, \ldots, s_h\}$, and a w-length query/pattern string. In detail, we are considering a multiset, allowing repeated sequences. This set can also be considered as a $h \times w$ matrix X.

Given two positions $1 \leq k_1, k_2 \leq w$ and a sequence index $1 \leq i \leq h$, we denote $s_i[k_1..k_2)$ as the substring of s_i starting from column k_1 and ending in column $k_2 - 1$. Now we need to define matches in this context.

Definition 13 (Locally maximal match [13])**.** *Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, and two subsequences $s_i[k_1, k_2)$ and $s_j[k_1, k_2)$, with $1 \leq i \neq j \leq h$ and $1 \leq k_1, k_2 \leq w$, we define $s_i[k_1, k_2) = s_j[k_1, k_2)$ as a locally maximal match if*

(a) Input set of sequences S.

S	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0
2	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0
3	1	0	1	0	1	0	0	1	0	0	0	1	1	0	0
4	1	1	1	0	1	1	1	1	0	0	1	0	0	0	0
5	0	1	0	1	0	0	0	0	1	1	0	1	0	0	1
6	1	0	1	0	1	1	1	1	0	0	1	1	0	0	0

(b) Pattern P with a coverage.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0

Figure 2.4: Example of SMEMs shared between a set of sequences S and a query P.

and only if $(k_1 = 1 \vee s_i[k_1 - 1] \neq s_2[k_1 - 1]) \wedge (k_2 = w \vee s_i[k_2] \neq s_j[k_2])$.

Informally, a match is locally maximal if we can not extend the two subrows in any direction without introducing a mismatch.

Definition 14 (Set-Maximal Exact Match [13])**.** *Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, a w-length string P, and two column indices $1 \leq k_1, k_2 \leq w$, we define $P[k_1, k_2) = s_i[k_1, k_2)$ as a Set-maximal Exact Match (*SMEM*) if and only if: a) the match is locally maximal, and b) there is no a longer locally maximal match shared between P and another sequence in S, i.e. s_j with $j \neq i$, which includes the column interval $[k_1, k_2)$.*

The definition of SMEM can be easily extended to k-SMEM, requiring having at least k sequences in S that share the exact same SMEM [24]. See Figure 2.4 for an example of SMEMs.

Another problem that falls into the string-to-matrix pattern-matching context is the Minimal Positional Substring Cover problem.

We define a positional substring of a w-length string P as a triplet (k_1, k_2, P), having $1 \leq k_1, k_2 \leq w$. We say that two positional substrings (k_1, k_2, P) and (k'_1, k'_2, t) are equal if and only if $k_1 = k'_1$, $k_2 = k'_2$, and $P[k_1..k_2] = t[k'_1..k'_2]$ [90]. Moreover a positional substring (k_1, k_2, P) is contained in a string t if and only if

(a) Input set of sequences S.

S	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0
2	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0
3	1	0	1	0	1	0	0	1	0	0	0	1	1	0	0
4	1	1	1	0	1	1	1	1	0	0	1	0	0	0	0
5	0	1	0	1	0	0	0	0	1	1	0	1	0	0	1
6	1	0	1	0	1	1	1	1	0	0	1	1	0	0	0

(b) Pattern P with a coverage.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0

Figure 2.5: Example of 2 possible MPSCs (leftmost-MPSC with dashed circles and rightmost-MPSC with dotted circles) shared between a set of sequences S and a query P. We show the coverage on P just for the dotted one.

$P[k_1..k_2] = t[k_1..k_2]$ [90]. Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, a positional substring (k_1, k_2, P) is a k-positional substring if (k_1, k_2, P) is contained in at least k sequences of S.

Definition 15 (Minimal Positional Substring Cover [90])**.** *Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, and a w-length string P, a Minimal Positional Substring Cover (MPSC) of P by S is a set $\mathcal{C}$ such that: a) each position $1 \leq \ell \leq w$ of P is covered by a positional substring $(k_1, k_2, t) \in \mathcal{C}$, having $k_1 \leq \ell \leq k_2$, b) each substring cover $(k_1, k_2, t) \in \mathcal{C}$ in contained in P, c) each substring cover $(k_1, k_2, t) \in \mathcal{C}$ in contained in at least one sequence of S, and d) the* MPSC *size $|\mathcal{C}|$ is minimal.*

Like for the SMEMs, Definition 15 can be extended to k-MPSC, slightly modifying point c) to consider at least k sequences of S [24]. See Figure 2.5 for an example of MPSC.

Note that every positional substring (i, j, P), which is part of a (k-)MPSC of P by S, underlies an interval that falls within a (k-)SMEM which spans across the columns i' and j', where $i' \leq i$ and $j' \geq j$, because i and j either directly constitutes a the boundary columns of a (k-)SMEM or this column range can be extended either to the left, the right, or both.

We recall that the (k-)MPSC problem does not admit a solution if at least k positional substrings can not cover some position. On the contrary, the (k-)SMEM-finding problem always admits a solution.

Some classes of MPSC have been identified [25, 90] to constrain the returned solution.

Problem 1 (MPSC variants finding problems)**.** *Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, an external w-length query P, a MPSC C, where the i-th positional substring of C for $1 \leq i \leq |C|$ is the i-th positional substring in the enumeration of the positional substrings of C by increasing the starting positions, and the length of C is the sum of the lengths of its positional substrings, we define the following problems: a) find a leftmost MPSC C of P by S, i.e. a MPSC of P by S such that any i-th substring in C starts at least as early as the i-th substring of every other MPSC of P by S, b) find a rightmost MPSC C of P by S, i.e. a MPSC of P by S such that any i-th substring in C ends at least as late as the i-th substring of every other MPSC of P by S, and c) find a length-maximal MPSC of P by S, i.e. the MPSC that has the largest length out of all MPSCs of P by S.*

Note that each of these problems can be generalized to k-MPSC. To solve the length-maximal MPSC, an additional lemma and a theorem are required.

Lemma 3 (Lemma 1, required regions [25])**.** *Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, and, an external w-length query P, there exists a contiguous nonempty range of sites for every $i \in [1..|C|]$, such that $C[i]$ contains this range of sites for all MPSCs C of P by S. Call the most extensive such range the i-th required region.*

Theorem 1. *Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, and an external P of length w, a length-maximal MPSC can be retrieved in $\mathcal{O}(w)$-time and $\mathcal{O}(n)$-space, using a leftmost MPSC, a rightmost MPSC, and the* SMEM*s.*

In this thesis, we will not focus on another type of matching, such as L-long Matches [94] or maximal blocks [95, 96], and we refer to [13, 90, 24] for further details.

2.7 Binary matrix indexing via the PBWT

As in the previous section, here we use (binary) matrices, sets of (binary) sequences, and (haplotypes) panel interchangeably; hence, in this context "sequence" and "row" are synonymous. For simplicity, in formal definitions we will primarily refer to sets of sequences, using the term "column" to denote the bits occupying the same position across all these sequences. We also recall that these sets can be multisets.

In 2014, Richard Durbin extended the key ideas of the BWT to the scenario of indexing set of binary sequences, proposing the Positional Burrows–Wheeler Transform (PBWT) [13]. Hence, we are considering set of sequences built over the alphabet $\Sigma = \{0, 1\}$, where $0 \prec 1$.

Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, the PBWT index for an arbitrary position $1 \leq k \leq w$ is based on colexicographical ordering (*i.e.* from right to left) of all the $(k-1)$-th prefixes of all the sequences in S: $\{s_1[1..k-1], \ldots, s_h[1..k-1]\}$. Note that the term "positional" comes from the fact that we can not have matches that do not involve the same positions, *i.e.* the bit in the i-th position of the pattern can match only bits in the i-th positions of the sequences in S.

The PBWT leverages the use of a pair of h-length arrays for each position, called the Prefix Array (PA) and the Divergence Array (DA).

Definition 16 (Prefix Array [13])**.** *Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, and a position $1 \leq k \leq w$, the k-th prefix array PA_k is a h-length array such that, for $1 \leq i \leq h$, $\mathsf{PA}_k[i] = j$ if and only if $s_j[1..k-1]$ is the i-th prefix in the ordering of all the $(k-1)$-th prefixes of the sequences in S.*

Definition 17 (Divergence Array [13])**.** *Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, and a position $1 \leq k \leq w$, the k-th divergence array DA_k is a h-length array which stores, for $2 \leq i \leq h$, the starting position of the longest common suffix shared between $s_{\mathsf{PA}[i]}[1..k-1]$ and $s_{\mathsf{PA}[i-1]}[1..k-1]$. By definition $\mathsf{DA}_k[1] = k$.*

We can compute PA_k and DA_k in $\mathcal{O}(h)$ time and $\mathcal{O}(h)$ space using a procedure similar to a radix-sort step, starting from PA_{k-1}, DA_{k-1} and the k-th position of S,

as shown in Algorithm 2.4. Consequently, the entire set of PA and DA arrays can be computed in $\mathcal{O}(hw)$ time and $\mathcal{O}(hw)$ space.

Algorithm 2.4 Algorithm for updating PA_k and DA_k to PA_{k+1} and DA_{k+1} [13].

```
1: function BuildPrefixAndDivergenceArrays(S, k, PA_k, DA_k)
2:     u ← 0, v ← 0
3:     p ← k + 1, q ← k + 1
4:     a ← [], b ← [], d ← [], e ← []
5:     for i = 1 to h do
6:         if DA_k[i] > p then
7:             p ← DA_k[i]
8:         if DA_k[i] > q then
9:             q ← DA_k[i]
10:        if s_{PA_k[i]}[k] = 0 then
11:            a[u] ← PA_k[i], d[u] ← p
12:            u ← u + 1, p ← 0
13:        else
14:            b[v] ← PA_k[i], e[v] ← q
15:            v ← v + 1, q ← 0
16:    PA_{k+1} ← concatenate(a, b)
17:    DA_{k+1} ← concatenate(d, e)
```

Note that Definition 17 can be changed to store the length of the longest common suffix instead of the starting position, as in [21, 23, 24, 27]. We denote these divergence array variants as the Reverse Longest Common Prefix (RLCP) array, having $\mathsf{RLCP}_k[i] = k - \mathsf{DA}_k[i]$.

Moreover, like for the suffix array, we can define the inverse of the prefix array for position k as the h-length array IPA_k such that $\mathsf{IPA}_k[i] = j$ if and only if $\mathsf{PA}_k[j] = i$. Hence, we strongly connect the text indexing data structures and the matrix/set of sequences indexing ones. The (inverse) suffix array is related to the (inverse) prefix array, the longest common prefix array to the divergence array/reverse longest common prefix array, and the Burrows-Wheeler Transform to the Positional Burrows-Wheeler Transform. We will demonstrate how this connection can be leveraged in pattern-matching algorithms.

Finally, we can permute the input set of sequences S using the PA to get the PBWT of S, which is another set of h of w-length sequences, or, for simplicity, a $h \times w$ matrix.

i	DA_5	PA_5	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
01	5	09	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1
02	1	12	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0
03	1	13	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
04	1	14	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
05	1	15	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
06	1	16	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
07	2	18	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1
08	4	19	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1
09	1	20	0	1	1	0	1	0	1	0	0	0	0	0	1	0	1
10	5	01	1	0	0	1	0	0	0	0	0	0	0	1	1	0	1
11	1	02	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1
12	1	03	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
13	1	04	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
14	3	05	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
15	1	06	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
16	1	07	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
17	1	08	0	1	0	1	0	1	0	0	0	0	0	0	1	0	1
18	1	10	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
19	1	11	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
20	1	17	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1

Figure 2.6: Example of PA, DA, and RLCP while computing the 5-th column of PBWT. Building the arrays for the column 5 we colexicographically sort all the rows up to column 4. After this ordering, the column 5 will be stored as the 5-th column of the PBWT.

Definition 18 (PBWT matrix [13])**.** *Given a set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, we define* PBWT *matrix, or simply* PBWT*, of S the $h \times w$ matrix such that: a)* $\mathsf{PBWT}[i][1] = s_i[1]$, $\forall\, i \in [1..h]$*, or b)* $\mathsf{PBWT}[i][j] = s_{\mathsf{PA}[i]}[j]$, $\forall\, i \in [1..h]$ *and* $\forall\, j \in [2..w]$.

In Figure 2.6, we show an example of PA and DA while building a PBWT column.

Computing SMEMs using the PBWT In [13], an algorithm was presented to compute all the SMEMs shared between an inputs set of h of w-length sequences $S = \{s_1, \ldots, s_h\}$, and a binary w-length query P. This algorithm, also known as Durbin's Algorithm 5, is strongly inspired by the BWT framework. In fact, it relies on three indices: f_k, g_k, and e_k, which represent the set of sequences currently sharing a left-maximal match with $P[e_k..k]$. This set of sequences is identified by the subarray $\mathsf{PA}_k[f_k..g_k - 1]$. This interval is like a Q-interval in the BWT context,

and at each iteration this interval $[f_k..g_k - 1]$ is updated to $[f_{k+1}..g_{k+1} - 1]$ using an FM-index-like approach. Note that we move from left to right in the PBWT context. Being $|\Sigma| = 2$, we need to store the number of zeros c_k in a PBWT column for the C function, being $h - c_k$ the number of ones in the k-th column, and two h-length arrays, namely u_k and v_k, for the Occ function. Than we can compute a forward step using a function $w_k : \{1, \ldots, h\} \times \Sigma \to \{0, \ldots, h\}$ such that, given $\sigma \in \Sigma$ and $1 \leq i \leq h$: $w_k(i, \sigma) = u_k[i]$ if $\sigma = 0$ or b) $w_k(i, \sigma) = c_k +_k [i]$ if $\sigma = 1$. This function works similarly to the LF-mapping; hence, we can use it to "follow" a specific sequence within the permutations induced by the prefix arrays. Moving from the left to the right, we can refer to this operation as FL-mapping, *i.e.* First-Last mapping to maintain a correspondence with the LF-mapping while "mapping" in the opposite direction. Briefly, Durbin's Algorithm 5, as shown in Algorithm 2.5, scans the query from left to right, updating the interval $[f_k..g_k - 1]$, reporting the SMEMs from column e_k to column k each time the interval $[f_{k+1}..g_{k+1} - 1]$ is empty. After reporting the current SMEMs, e_k is updated using the common suffix between the pattern and the matching sequences, while the interval is updated using DA_k along with the new starting position e_k.

Durbin's Algorithm 5 runs in $\mathcal{O}(w)$ time, according to the author [13]. Sanaullah et al. [97] proved that Durbin's Algorithm 5 runs in Avg.$\mathcal{O}(w + \gamma)$, where γ is the total number of SMEMs. Note that a naïve algorithm takes $\mathcal{O}(hw^2)$ time. These bounds assume that we have already precomputed the complete set of PA_k, DA_k, c_k, u_k, and v_k, $\forall\ 1 \leq k \leq w$. This preprocessing step takes $\mathcal{O}(hw)$ time. Moreover, assuming the scenario of multiple queries, we need to store all these arrays in $\mathcal{O}(hw)$ space, approximately $13hw$ bytes of RAM, according to the author[1]. One possible solution proposed in the original PBWT implementation is to include all queries in the input set of sequences, compute all internal SMEMs of this new set of sequences, and report those that contain at least one of the queries. Note that internal SMEMs can be computed without any preprocessing step in $\mathcal{O}(hw)$ time and $\mathcal{O}(h)$ space by scanning the set of sequences once and keeping in memory only the current PA_k and DA_k. On the other hand, we need to build and scan these matrices each time we query a new set of strings. Moreover, this approach is less

[1] `https://github.com/richarddurbin/pbwt/blob/29901a7d66dd9b86bd8866aea47a1cfa928d806e/pbwtMatch.c#L252`

Algorithm 2.5 Durbin's Algorithm 5 for computing SMEMs between a set of sequences S and a string P [13].

function FIND_SET_MAXIMAL_MATCHES_WITH_Z(S,P)
    $e \leftarrow 1,\ f \leftarrow 0,\ g \leftarrow 0$ ▷ We assume to have computed the PBWT of S
    for $k \leftarrow 1$ **to** w **do**
        $e, f, g \leftarrow$ *UPDATE_MATCHES*(k, P, e, f, g)

function UPDATE_MATCHES(k, P, e, f, g)
    $f' \leftarrow w_k(f, P[k])$
    $g' \leftarrow w_k(g, P[k])$
    if $f' < g'$ **then**
        if $k = w$ **then**
            reports SMEM*s from* e_k *a* w *with sequences* $\mathsf{PA}_k[f'..g']$
        $e' \leftarrow e$
    else
        reports SMEM*s from* e_k *a* w *with sequences* $\mathsf{PA}_k[f'..g']$
        $e' \leftarrow \mathsf{DA}_{k+1}[f'] - 1$ ▷ *Update* f_k, g_k, e_k
        if $P[e'] = 0$ **and** $f' > 0$ **then**
            $f' \leftarrow g' - 1$
            while $P[e' - 1] = \mathsf{PBWT}[\mathsf{PA}_{k+1}[f']][e' - 1]$ **do** $e' \leftarrow e' - 1$
            while $\mathsf{DA}_{k+1}[f'] \leq e'$ **do** $f' \leftarrow f' - 1$
        else
            $g' \leftarrow f' + 1$
            while $P[e' - 1] = \mathsf{PBWT}[\mathsf{PA}_{k+1}[f']][e' - 1]$ **do** $e' \leftarrow e' - 1$
            while $g' < h$ **and** $\mathsf{DA}_{k+1}[g'] \leq e'$ **do** $g' \leftarrow g' + 1$
    return e', f', g'

flexible. This is evidenced by the fact that, in the literature, numerous extensions exist for Algorithm 5 but not for this method [90, 98, 94, 9]. Additionally, the SMEMs are reported according to the order induced by the prefix arrays rather than by the original query order.

In addition, given a string P of length w, a set S of h sequences of length w, and the PBWT of S, Sanaullah et al. [90] proved that, using an approach similar to the one used for SMEMs, all the discussed variants of the MPSC problem can be solved in $\mathcal{O}(w)$ time and $\mathcal{O}(n)$ space, assuming to have already computed the PBWT. Algorithm optimality is ensured by the property of (k-)MPSC modularity [25, Lemma 2]. This (k-)MPSC algorithm of Sanaullah et al. [25] requires $\mathcal{O}(n)$-space to ensure constant-time random access to the input set of sequences and to the PBWT.

PBWT Extensions in Literature

Over the last decade, various extensions of the PBWT have been proposed in the literature to address different problems and optimize the data structure and related algorithms. We provide a brief description of some of them. Refer to [99, 100, 98, 9, 101] for other PBWT variants.

Multiallelic PBWT Introduced by Naseri et al. in 2019 [91], the Multiallelic PBWT is a natural extension of the PBWT to a generic alphabet. This extension is relevant both in the biological context [102, 103] and in the combinatorial one [2, 96]. The algorithm proposed by Durbin can be generalized to the multiallelic context, adding a usually negligible factor $|\Sigma|$ to the PBWT construction time complexity. Regarding queries with an external array, we need to store a data structure that supports the permuted column and computes the FM-index, such as a wavelet tree (see Section 2.2). Hence, both time and space bounds are increased by the ones of this additional component.

Dynamic PBWT Proposed in 2021 by Sanaullah et al. [97], the Dynamic PBWT leverages the use of linked lists instead of arrays to store the fundamental data structure of the PBWT. The main goal is to provide an efficient way to update the PBWT, adding/deleting new sequences in the input set in $\mathcal{O}(w)$. The authors

presented a new set of algorithms that can replace those proposed by Durbin. These new algorithms have the same bounds as the classical PBWT.

PBWT with the LEAP data structure Naseri et al. in 2019 proposed the use of the PBWT to find any match longer than λ shared between a set of sequences, and query [94]. The proposed algorithm takes $\mathcal{O}(w + \gamma(\rho + \lambda))$, where γ is the total number of matches and ρ the average length of the matches. Moreover, they propose storing an additional data structure, called Linked Equal/Alternating Position (LEAP), which stores some precomputed jumps (stored in 8 h-length additional arrays per column) in the a PBWT column, speeding up the λ-long matches finding algorithm. Thanks to these additional $\mathcal{O}(hw)$ space we can find all the γ λ-long matches in $\mathcal{O}(w + \gamma)$ time.

PBWT with wildcards Proposed by Bonizzoni et al. in 2023 [93], the Wild-PBWT aims to extend the algorithm of the multiallelic PBWT to consider missing data, *i.e.* the symbol "$\star$" in the input set of sequences. These missing data can be interpreted as any other symbol in the alphabet Σ, exponentially increasing the execution time to compute internal matches, such as maximal blocks, due to the possible interpretations of each set of wildcards.

Syllable-PBWT Wang et al. presented in 2023 the Syllable-PBWT [98], a space-efficient PBWT implementation that divides the haplotypes set into syllables, *i.e.* common regions of the input set of sequences are represented using a custom alphabet. After this special encoding, a PBWT is built on this compressed syllabic set of sequences, leveraging rolling hash functions for substring comparisons. The Syllable-PBWT can be used to find long matches and not set-maximal exact matches. From a complexity point of view, construct the Syllable-PBWT takes $\mathcal{O}(h\nu\beta \log h)$ time and $\mathcal{O}(h\nu + n/\rho)$ space, where $\nu = \lceil w/b \rceil$ (with b as the number of sites per each syllable), and $\rho = h\nu/|r|$ (having r dictionaries for the syllables). Querying the Syllable-PBWT takes $\mathcal{O}(w + \nu\beta \log h)$ time.

2.8 String-to-graph pattern-matching

Thanks to recent advances in pangenomics [1, 8, 2], pattern-matching algorithms and data structures have been extended to handle node-labeled graphs. Formally, node-labeled graph can be defined as a pair $(G, \mathcal{L})$, where $G = (V, E)$ is a graph and $\mathcal{L} : V \to \Sigma^+$ is the labelling function that assigns to each node $v \in V$ a label $\mathcal{L}(v) \in \Sigma^+$, where Σ^+ is set of all non-empty strings made from Σ.

Given a m-length strings P and a node-labeled graph $G = (V, E)$, an occurrence of P in G is a path $\pi = (v_1, ..., v_k)$ such that P occurs in the labels concatenation of π, *i.e.* P occurs in $T = \mathcal{L}(v_1) \cdot \cdots \cdot \mathcal{L}(v_k)$. A node-labeled graph can have multiple occurrences of a pattern.

2.9 Node-labeled graph indexing

In the literature, multiple node-labeled graph indices have been proposed to index and query not only the node labels but also the graph topology. Those indices are mainly based on the BWT and FM-index-like data structures.

In this thesis, we will focus on GCSA, but we refer to [18, 17, 30, 31, 104, 2, 105] for additional details and approaches on graph indices.

Generalized Compressed Suffix Array `GCSA` (Generalized Compressed Suffix Array) [106, 18] was the first attempt to generalize the key ideas of the BWT and FM-index to an acyclic graph. Then, cyclic graphs were considered with `GCSA2` [30, 31]. These indices are similar to those used to succinctly index de Bruijn graphs [107] using a BWT-based approach. To build the `GCSA`/`GCSA2` index, we compute the underlying de Bruijn graph [108, 109] using the "prefix-doubling" algorithm. This algorithm iteratively joins k-length paths into $2k$-length paths until it reaches the k-size used by the k-mers in the de Bruijn graph. Currently, the maximum k-mer size is $k = 256$. This approach presents two substantial limitations: a) the "prefix-doubling" algorithm requires a high memory usage, and b) the pattern length for querying is limited by the k-mers maximum size, *i.e.* $k = 256$. Finally, this de Bruijn graph can be indexed as in [107] to support operations such as the LF-mapping. From a complexity point of view, assuming

rank operations in $\mathcal{O}(\rho)$ time over the index data structure, we can find all the γ occurrences of an m-length pattern in $\mathcal{O}(\rho(m + \gamma))$ time. Refer to [18] for a complete complexity analysis.

2.10 Sequence alignment

Another fundamental topic in stringology and in bioinformatics is sequence alignment. Given two strings A and B, their alignment highlights the number of edit operations, *i.e.* insertions, deletions, and substitutions, needed to change A into B or vice versa [42]. Considering two strings, we refer to these alignments as pairwise alignments. Note that insertions and deletions are commonly referred to as indels. In other words, by counting the number of edit operations, namely by computing the so-called edit distance, and examining the operations required, we can determine both the similarities between two strings and where these similarities occur. Each operation is assigned a weight, and most alignment algorithms use dynamic programming. In this thesis, we will not extend or directly develop an alignment algorithm; hence, we will not focus on the algorithmic details of alignment approaches. For further explanations of such alignments, including complexities, dynamic programming algorithms, and optimizations, we refer to [56, 46, 42].

Firstly, we can distinguish between two primary alignment modes: global alignments [110] and local alignments [111]. The first mode performs end-to-end alignment between the two strings, while the latter aligns a string to a substring of the other string. Between these modes, there is also a hybrid semi-global alignment [112], which corresponds to a global alignment where gaps at the beginning of one sequence can be specified as penalty-free.

A plethora of tools have been developed to handle pairwise alignments, such as [60, 113, 114]. Moreover, we can consider multiple sequence alignments, which are achieved through multiple sequence alignment (MSA). These kinds of alignments are often the starting point for building pangenome graphs, and various tools are already available in the literature, such as [115, 116].

As already pointed out in Section 2.4, alignments are often based on the "seed and extend" paradigm [59, 60, 61, 15] to mitigate the high requirements of using a complete dynamic programming algorithm. This technique leverages

the computation of short MEMs, called "seeds", to reduce the use of dynamic programming, which is applied only to "extend" these matches.

Finally, in recent years, sequence alignment in bioinformatics has been extended to align strings against node-labeled graphs. From a high-level perspective, a string-to-graph alignment aims to align the input strings to sequences encoded in the node labels, reporting the alignment positions as node IDs. In literature, various tools have been proposed for string-to-graph alignments, such as [117, 104, 118, 119].

2.11 Biological background

The Deoxyribonucleic Acid (DNA) is a nucleic acid that contains all the genetic information of a living being. This information is essential for the proper development and maintenance of life, mainly by producing and utilizing proteins.

Chemically, DNA is a polymer composed of two polynucleotide chains, also called strands, arranged in a double helix, as discovered by Watson and Crick in 1953 [120]. Each nucleotide consists of a sugar called deoxyribose, a phosphate group, and a nitrogen-containing nucleobase, which can be cytosine (C), guanine (G), adenine (A), or thymine (T). The two polynucleotide strands have opposite directions, defined by the phosphate group and the deoxyribose, at the beginning and the end of each strand. Hence, we have a forward strand, usually denoted by the symbol "+", and the backward strand, traditionally denoted by "-". More technically, each DNA strand has a 5' end, named because the terminal nucleotide has a free phosphate group attached to the fifth carbon of the deoxyribose, and a 3' end, named because it terminates with a free hydroxyl group on the third carbon of the deoxyribose. By convention, the orientation of DNA is referred to as the 5'-to-3' because the synthesis in vivo occurs just in this direction.

To form the double helix, the two strands, oriented oppositely, are connected by bonds between two nucleobases, one per strand. Cytosine can only be bonded to guanine (and vice versa), while adenine can only be bonded to thymine (and vice versa). Cytosine and thymine are called pyrimidines, while guanine and adenine are called purines. This pairing allows us to retrieve one strand from the other using the reverse-and-complement technique. In fact, given the nucleobase sequence of a strand, we can reverse the sequence itself and complement each nucleobase using

the corresponding one, *i.e.* C $\longleftrightarrow$ G and A $\longleftrightarrow$ T, to obtain the other strand.

We refer to the complete DNA sequence of a living being, contained in the nucleus of each cell, as the genome. The genome is arranged into multiple chromosomes, and each region that can encode a protein is called a gene.

Ribonucleic Acid (RNA), unlike DNA, is a nucleic acid that consists of a single polynucleotide strand, composed like DNA strands, except that uracil (U) is used instead of thymine. The principal scope of the RNA is to regulate gene expression.

The unit of measurement for DNA/RNA sequences is the base pair (bp).

Protein encoding and alternative splicing In this thesis, we will discuss computational methods for Alternative Splicing; therefore, we need to briefly describe this biological mechanism. To produce a protein from a gene, we have three main steps: a) the transcription step, b) the splicing step, and c) the translation step.

During the first step, the gene sequence is copied into pre-messenger RNA (pre-mRNA), selecting a single DNA strand, which is then transcribed by RNA polymerase (an enzyme). During the second step, the pre-mRNA's non-coding regions, known as introns, are removed, leaving only the coding regions, called exons. The concatenation of these exons is called messenger RNA (mRNA) or transcript/isoform. We denote the nucleobases at the boundaries between introns and exons as splicing sites. In detail, the first two bases of an intron are called the 5'/donor splice site, while the last two are the 3'/acceptor splice site. Finally, in the third step, a portion of the mRNA, which is the coding sequence, is translated into a protein. In detail, each codon/triplet, *i.e.* a non-overlapping 3-length substring, is translated into an amino acid. Concatenating the resulting amino acids yields a protein. Usually, we have a start codon and a stop codon to bound the coding sequence.

In 1977 [121, 122], it was observed that one gene can encode more than just one protein, breaking a famous biological dogma, which associated one gene with exactly one protein. The mechanism involved in this behavior is called Alternative Splicing (AS) [123]. Thanks to AS events, more than one transcript can be encoded by a single gene. We refer to such transcripts as alternative transcripts/isoforms, whereas in the absence of any AS event, we have the canonical transcript/isoform.

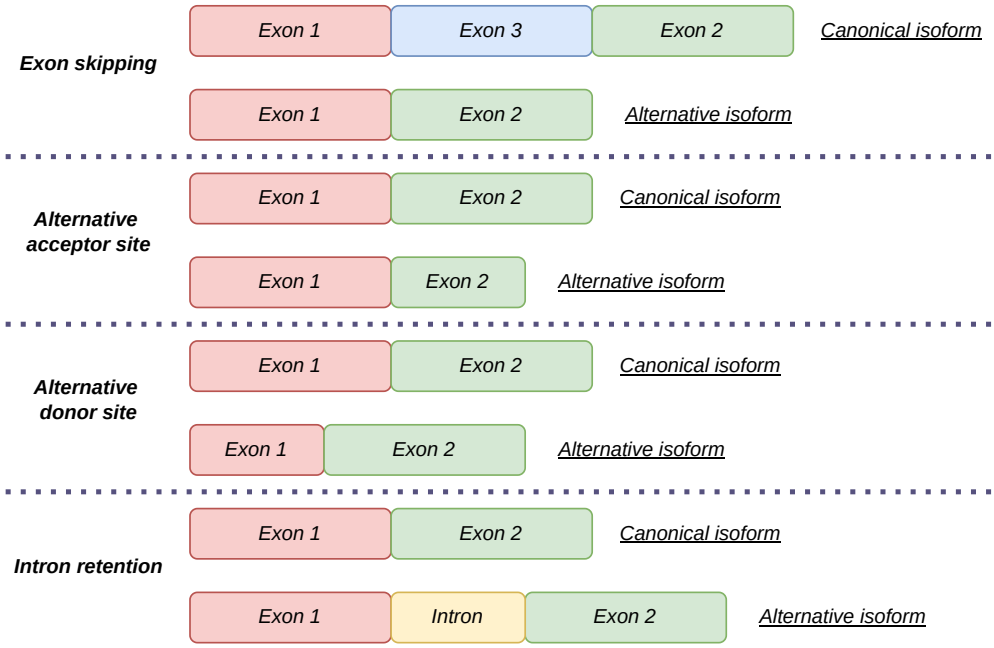


Figure 2.7: Canonical and alternative isoforms of the AS events considered in this thesis.

We have multiple categories of AS events, and, in this thesis, we are interested in: a) exon skipping/cassette exon, where one exon or multiple consecutive exons (excluding the first one and the last one) are skipped in the alternative isoform, b) alternative acceptor site, where a different 3' splice site is used in the alternative isoform, c) alternative donor site, where a different 5' splice site is used in the alternative isoform, and d) intron retention, where we include an intron in the alternative isoform.

Figure 2.7 graphically summarizes these AS events. Note that this is a high-level classification of biological mechanisms, which are much more complex in the real world, with less well-defined splice sites or multiple simultaneous events.

Haplotypes and genotypes Two additional biological concepts fundamental to this thesis are haplotypes and genotypes; therefore, we include brief descriptions of both. If we compare the genomes of two samples of the same species, we discover they are very similar, except for a very low percentage of nucleotides. These

differences are called genomic variants, and we can group them into: a) Single Nucleotide Polymorphisms (SNPs) and indels, if they involve a small DNA/RNA region, and b) structural variation, if they involve more than 50bp.

To define haplotypes and genotypes, we consider one of the two sample genomes as the reference genome. In detail, SNPs are the substitution of a single nucleotide, while indels are the insertion/deletion of one or more nucleotides. The position in which a SNP or an indel occurs is called a locus/variation site, the DNA sequence in a locus is called an allele, and the set of all alleles an individual inherits from each parent is called a haplotype. By combining the two inherited haplotypes, we obtain the individual's genotype. If the two haplotypes are identical at a specific site/region, the genotype is homozygous in that site; otherwise, it is heterozygous. In detail, when comparing multiple genomes against a reference genome, we may encounter canonical and alternative alleles, depending on whether the same site/region is present in both the considered genome and the reference.

Given a genotype, a heterozygous site/region could be phased or unphased, whether we know which haplotypes resolve that site/region. The process of detecting those haplotypes is called genotype phasing. Finally, we may have a genotype with some unknown sites/regions, and we refer to the procedure for filling these missing data as genotype imputation. In this thesis, we will formalize some of these concepts combinatorially.

2.12 Bioinformatics and pangenomics

Bioinformatics is an interdisciplinary field that leverages computational models to solve biological problems, such as those involving DNA/RNA and protein sequences. Such sequences are produced via chemical processes that have evolved over the past decades. Beginning with Sanger sequencing [124], the first sequencing technology proposed in 1977 and later utilized in the Human Genome Project [125], sequencing has progressed to Next-Generation Sequencing (NGS) technologies [126] since 2005. NGS methods, such as Illumina sequencing[2], are faster and more cost-effective, capable of producing hundreds of millions of short reads (approximately 150bp)

[2] `https://emea.illumina.com/systems/sequencing-platforms.html`

per day with an error rate of less than 1%. Since 2008, third-generation sequencing technologies [127], such as Oxford Nanopore Technology[3] and PacBio[4], have enabled the production of millions of long reads (ranging from 1000bp to 100 000bp) and ultra-long reads (exceeding 100 000bp) in a single day. Although these methods are currently more expensive than NGS and produce reads with higher error rates, the accuracy has significantly improved in recent years.

In Chapter 5, we will often refer to RNA-Seq reads. These reads, produced by NGS sequencing of RNA, represent transcripts and are fundamental for analyzing alternative splicing events. In other words, we use these reads to analyze the transcriptome of an individual, *i.e.*, the complete set of RNA transcripts, including both coding and non-coding regions.

Another essential aspect of sequencing reads is the distinction between single-reads and paired-end reads. In detail, single-read sequences capture only one end, *i.e.* the start or the finish, of a DNA or RNA fragment. In contrast, paired-end reads involve both ends, providing more informative reads that, for instance, result in more accurate alignments.

To illustrate another application of sequencing technologies, consider that in Chapter 3 we analyze haplotype panels as sets of sequences, primarily generated from NGS samples using well-established variant calling methods. This bioinformatics process involves analyzing the complete set of reads from an individual and aligning them to a reference genome to reconstruct the individual's genotype.

All these bioinformatics methods rely on a single reference; in pangenomics, we aim to avoid the so-called reference-bias by considering multiple genomes simultaneously. To understand why this is a crucial improvement, consider the scenario of sequencing an individual from Asia. If our single reference genome comes from a European sample, we would likely miss important information that could be obtained by aligning reads against an Asian reference, or, even better, against multiple references simultaneously. This is why, in recent years, various pangenome-related projects have been launched, such as those led by the Human Pangenome Reference Consortium (HPRC) [3] and the Pangaia Project [5]. These

[3] `https://nanoporetech.com/products/sequence`
[4] `https://www.pacb.com/sequencing-systems/`
[5] `https://www.pangenome.eu/`

initiatives aim to both collect pangenome data and develop tools to manage these complex datasets.

2.12.1 File formats

Biological data must be stored in specific file formats that are easily accessible and queried. We will now present some of the most commonly used standards in modern bioinformatics and pangenomics.

FASTA and FASTQ formats FASTA files [128] are the simplest way to store DNA sequences. It is a plain text file format in which, for each sequence, we store: a) a header string, starting with the symbol ">", which contains the name of the sequence and some additional information, and b) the sequence itself, possibly in multiple lines. A single FASTA file can store multiple sequences.

FASTQ files [129], developed at the Wellcome Trust Sanger Institute, extend the FASTA format to include sequencing quality information. In this case, we store four lines for each sequence: a) a header like in the FASTA format, despite starting with the symbol "@", b) the sequence, stored in one line, c) a line separator, and d) an alphanumeric string representing the sequencing quality. Each nucleobase has a quality score represented as an ASCII character; for instance, "!" is used for the lowest quality and "∼" for the highest.

Both formats can be indexed for efficient access [130].

Variant Call Format Developed in 2011 to store the results of the 1000 Genomes Project (1KGP) [4], the Variant Call Format (VCF) [131] stores genotypes, as sets of genomic variants. These variants are called between one or more samples and a reference.

At the beginning of the file, several header lines that start with "##" contain information on the genotypes described in the file and on how the file is produced. The rest of the file consists of the data section, in which each line is tab-separated into at least eight mandatory columns: a) the name of the sequence (usually the chromosome) in which we have called the genomic variant, b) the position of the variant, c) an ID (". " if unknown), d) the nucleobases string of the reference in

the given position, e) the nucleobases string of the sample (*i.e.* alternative) in the given position, f) a quality score, g) a flag to check if the variant has passed or not a filter step, and h) a list of key-value pairs with additional information. Optionally, for each sample, we can store the variant's genotype, as specified in an optional format field. Although optional, the genotype is usually one of the most essential data entries we store and query using a VCF file.

VCF files can be stored in binary and compact formats, namely Binary VCF/BCF, and both can be indexed to enhance query performance.

Gene transfer Format General Feature Format (GFF), proposed in 1998 by Durbin and Haussler at the Wellcome Trust Sanger Institute, is now deprecated, and Gene Transfer Format (GTF) [128] (*i.e.* GFF2) is a standard file format used to annotate genomic features. In other words, a GTF file stores the gene annotation.

Each feature is represented in a tab-separated line with nine columns: a) the sequence name (usually the chromosome), b) the name of the program which generated the feature, c) the feature name (gene, exon, intron, etc...), d) the starting position of the feature, e) the ending position of the feature, f) the score, g) the strand ("+" if forward and "-" if reverse), h) a flag to optionally (if not specified, we have ".") determine if the first base of the feature is the first ("0"), the second ("1"), or the third ("2") base of a codon, and i) a semicolon-separated list of attributes.

Sequence Alignment Map Proposed in 2009, the Sequence Alignment Map (SAM) format [132, 130] is used for storing sequence alignments against a reference genome. The file contains a header with information about the file and an alignment section, where each line represents an alignment. For each alignment, we have a tab-separated line with 11 mandatory columns plus various optional fields: a) the read/segment name, b) a flag to identify the type of alignment, c) the reference name, d) the alignment position on the reference genome, e) the quality of the alignment from 0 to 255 (if absent, we have 255), f) the Concise Idiosyncratic Gapped Alignment Report (CIGAR) string, which describes the alignment in terms of matches, indels, and substitutions, g) three columns which contain some

information about the paired read, h) the sequence, and i) the sequence quality as in the FASTQ format. Finally, we can include optional fields with a type flag, such as "I" for integers and "Z" for strings.

SAM files can be stored in a binary, light-weight format called Binary Alignment Map (BAM), and both formats can be indexed for efficient access after the entries have been sorted by read name and alignment position.

Graphical Fragment Assembly Proposed in 2014 by Li [133, 134, 135], the Graphical Fragment Assembly (GFA) is a plain-text format used to store labeled graphs. In this thesis, we present the GFA 1.x format, which we refer to simply as the GFA format, but note that a group of researchers (including Li, Chin, Durbin, and Myers) proposed a new release, GFA 2 [136], in 2017.

We have tab-delimited lines, with the first field indicating the line type. We describe the more useful ones, referring to the specifications for further details.

Initially, we have a one-column header, introduced by the field "H". To store labeled nodes/segments, we have 3-column lines with: a) the symbol "S", b) the node ID, and c) the sequence.

Edges/links are represented in 6-column lines with: a) the symbol "L", b) two columns to store the edge starting node ID and the orientation of that segment ("+" for forward and "-" for reverse), c) two columns to store the edge ending node ID and the orientation of that segment ("+" for forward and "-" for reverse), d) a CIGAR string to specify the overlaps between the two nodes (having "*" if not specified).

Paths are stored as 4-column lines with: a) the symbol "P", b) the path name, c) a comma-separated list of the node IDs included in the path (each followed by "+" or "-" for the node orientation), and d) a comma-separated list for the overlaps (having a single "*" if not specified).

Note that each line type can be followed by a set of tab-separated optional and/or custom fields, specified as in the SAM format.

Graph Alignment Format The Graph Alignment Format (GAF) [137] is a tab-delimited format, similar to the SAM format, for storing string-to-graph alignments. Each line contains 12 mandatory fields: a) the read name, b) the read length, c)

the staring position of the alignment on the read, d) the ending position of the alignment on the read, e) the strand ("+" or "-") relative to the path, f) a string containing the subpath to which the read is aligned as the concatenation of each node's ID, where each one is preceded by the symbol "<" or ">" to specify the node orientation, g) the string-path length (*i.e.* the length of the string obtained by concatenating each node label), h) the staring position of the alignment on the string-path, i) the ending position of the alignment on the string-path, j) the number of residue matches in the path-string, k) the alignment block length, and l) the quality of the alignment from 0 to 255 (if absent, we have 255).

Finally, we can have multiple optional fields (in the format "identifier:type:data", where the type follows the exact SAM format specifications). For instance, one of the most common optional fields is the CIGAR string, specified as "cg:Z:CIGAR", with the same type nomenclature as in the SAM format, for instance, "I" for integers and "Z" for strings.

The binary variant of the GAF format is known as the GAM format.

Chapter 3

Store and Query Large Haplotypes Data

This chapter contains all the contributions to the development of a run-length encoded variant of the Positional Burrows–Wheeler Transform (RLPBWT) to solve various string-to-matrix pattern-matching problems in sublinear space, *i.e.*, using space proportional to the total number of runs in the PBWT rather than to the size of the input. We recall that the PBWT Algorithm 2.5 takes $\mathcal{O}(hw)$ space for a set of h binary sequences of length w, resulting in $13hw$ bytes in memory, computing all the γ SMEMs $\mathcal{O}(w + \gamma)$ time, after a $\mathcal{O}(hw)$ time preprocessing.

We will present both the theoretical and the experimental results, describing: a) how we modeled at first the RLPBWT [21], testing multiple possible combinations of data structures to solve the Set-Maximal Exact matches (SMEMs) finding problem, b) the μ-PBWT [23], the final composite data structure for the RLPBWT we developed as the best trade-off between memory requirements and querying time, c) how we can augment the μ-PBWT [24] to solve other problems such as the k-SMEMs finding problem and the Minimal Positional Substring Cover (MPSC) problem, and d) a first attempt to solve the genotype phasing problem using the μ-PBWT [27].

Although the original PBWT implementation already leveraged run-length encoding to store indices in memory, the RLPBWT and μ-PBWT are the first PBWT variants that directly rely on run-length encoding to compute matches. This was similar to what Durbin just suggested in [13]. Hence, the RLPBWT and μ-PBWT represent, in a sense, a generalization of the r-index to the PBWT. Run-length

encoding the PBWT enables the computation of SMEMs and MPSC using up to two orders of magnitude less memory than solutions based on the full PBWT [13, 25, 90], thus opening the possibility of indexing large haplotype panels, such as those available in the UK Biobank datasets [26, 6]. It is worth noting that this space reduction leads to a moderate increase in query time; however, experimental results demonstrate that the slowdown is limited and represents an acceptable trade-off for the achieved space savings. We note that determining the query-time bounds of μ-PBWT is currently an open problem. However, assuming $\hat{\rho}$ as the total number of runs in the PBWT, and γ as the number of SMEMs, we can conjecture that the μ-PBWT computes all SMEMs in *Avg.* $\mathcal{O}(w(\hat{\rho} + \log \hat{\rho}) + \gamma \log(w/\hat{\rho}))$ time, but it takes $\mathcal{O}(w(w\hat{\rho} + \log \hat{\rho}) + \gamma \log(w/\hat{\rho}))$ time in the worst case, after a $\mathcal{O}(hw)$ time preprocessing. This worst-case time can also be applied in computing MPSCs, assuming γ as the total number of reference sequences that share all the positional substrings in the MPSC.

Moreover, to apply the μ-PBWT in real biological scenarios, we attempted to leverage this space reduction in addressing the genotype phasing problem. To this end, we designed a combinatorial approach that is lighter than the HMM-based probabilistic methods typically used for this task. However, it is less effective at handling genotype errors when the reference panel lacks a sufficient number of haplotypes. Regarding time complexity, solving the genotype phasing problem introduces additional computational overhead, which is discussed later in this chapter.

We can summarize the theoretical and practical results of this chapter, briefly presented above, as:

Input:	A reference panel of bi-allelic haplotypes, represented as a set of h binary sequences of length w, and an external bi-allelic haplotype, represented as a w-length query binary string
Output:	Matches, coverages, and compatible regions shared between the input panel and the query string

Publications and conferences RLPBWT results [21] were presented in Pisa, Italy, at the *30th edition of the annual Symposium on String Processing and Infor-*

mation Retrieval (SPIRE 2023) – conference proceedings published in the *Lecture Notes in Computer Science (LNCS) series – Springer*. μ-PBWT [23] was published in the *Oxford Bioinformatics* journal. The new algorithms to compute k-SMEMs and MPSC via the μ-PBWT were presented in Fukuoka, Japan, at the *35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024)* – conference proceeding published in the *Leibniz International Proceedings in Informatics (LIPIcs) – Dagstuhl*. Finally, the genotype phasing approach was published in *"The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday", Open Access Series in Informatics (OASIcs) – Dagstuhl*.

Chapter outline In Section 3.1, we will present the state-of-the-art and the motivations that led us to develop a run-length encoded and space-efficient variant of the Positional Burrows–Wheeler Transform. We will also discuss some fundamental aspects of the phasing problem. This section will include some additional preliminaries to extend what we introduced in Section 2.8. In section 3.2, we will discuss in detail our algorithmic contributions. In section 3.3 we will compare our implementation against state-of-the-art tools, *i.e.* the original PBWT implementation and other compressed variants of the PBWT. Finally, in Section 3.4 we will draw conclusions and possible future developments.

3.1 Motivations and state-of-the-art

Next-generation sequencing technologies and modern computational approaches for haplotype phasing have, over the last decade, enabled more advanced studies of chromosome-level genomic variation, with significant implications for understanding genome evolution and for developing clinical applications. For this purpose, various haplotype-resolved whole-genome sequence data were collected from thousands of individuals in large biobanks such as the UK Biobank [26] and TOPMed projects [138]. From a computational point of view, the Positional Burrows-Wheeler Transform (PBWT) [13], already discussed in Section 2.7, is the core data structure used to store and query these large haplotype datasets. Furthermore, even though a set of haplotypes is simpler than a pangenome graph and does not fully address the reference bias problem, it still represents a form of the pangenome, and the PBWT

has therefore gained relevance in the field of pangenomics [2]. In any case, these datasets are becoming larger day by day, and indexing them remains challenging.

However, in recent years, the PBWT has been applied and extended in numerous ways, such as for genotype imputation [139], for privacy-focused genotype database search methods [140], for computing all-pairs Hamming distances [141], and for finding all maximal perfect haplotype blocks [95]. Its connection with computational pangenomics is also proved by Novak et al. [142] and Sirén et al. [31] who used the PBWT to encode and index a pangenome graph for haplotype matching (g-PBWT) [2].

These facts lead us to develop a highly compressed PBWT index that can store and query a biobank-size cohort of haplotypes on a commodity machine, even a laptop. Given that the original PBWT is already space-efficient for finding matches within a reference panel, we focused on the string-to-matrix scenario, where an external haplotype is used as a query and is not included in the reference panel.

In this chapter, we consider the reference panel as a set (in detail, a multiset allowing repeated sequences) of h sequences of length w. The reference panel can be represented interchangeably as a matrix with h rows and w columns. Often, we use n to represent the total size, having $n = hw$. The external haplotype/query is a w-length string. As in Durbin's paper, we assume haplotypes in which each variation site is bi-allelic, meaning only two alleles are observed at each locus, without insertions or deletions. This assumption is common in studies of diploid species, where variants are typically filtered to include only bi-allelic sites [143, 144]. Computationally, both the reference panel and the query are built over a binary alphabet $\Sigma = \{0, 1\}$ due to this bi-allelic nature.

Another challenge in haplotype analysis is addressing the haplotype threading problem. In this problem, we aim to represent a query haplotype by combining one or more substrings of haplotypes from a reference panel through recombinations. We can also bound the problem to at least k haplotypes from the panel. As in [25, 90], this problem can be addressed by solving the MPSC problem, which was already defined in Section 2.6, and it can be used as an alternative to the classical Li and Stephens model [145] on very large reference panels.

Moving from the most theoretical results to the more practical and biological, we have various applications of the PBWT including haplotype imputation [9, 139],

ancestral inference [146, 147], and genotype phasing [148]. It is easy to prove that the PBWT is nowadays an essential component of modern haplotype analysis pipelines.

Why the run-length encoding Directly citing Durbin's original PBWT paper [13]: "Furthermore we can also expect the y arrays (*i.e.* the PBWT columns) to be strongly run-length compressible. This is because population genetic structure means that there is local correlation in values due to linkage disequilibrium, which means that haplotypes with similar prefixes in the sort order will tend to have the same allele values at the next position, giving rise to long runs of identical values in the y array. So the PBWT can easily be stored in smaller space than the original data."

In other words, recalling the PBWT construction described in Section 2.7, we can observe that the permutation of each column tends to produce high run-length compressible binary strings. In fact, two haplotype prefixes, which are consecutive in the prefix array of a specific column, will likely share the same bit in the next column, which is the one encoded in the PBWT matrix. Thanks to this behavior, we will end up with a few and long runs in each PBWT column. Hence, by exploiting the inherent similarities between the PBWT and the classical BWT, and by generalizing recent BWT run-length encoding techniques, such as the r-index [79], constructing a run-length encoded PBWT has proven to be the most "natural" approach.

In fact, in 2022, Brown et al., in a preprint paper [149] (paper final version [150]), firstly proposed a PBWT run-length encoding, which was just able to reconstruct the input set of sequences, hence, which does not support SMEMs computing. On the other hand, our contribution mainly involves handling external queries to compute set-maximal exact matches.

3.1.1 Preliminaries

As in Section 2.6 and Section 2.7, in this chapter we use (binary) matrices, sets of (binary) sequences, and (haplotypes) panel interchangeably; hence, in this context "sequence" and "row" are synonymous. For simplicity, in formal definitions we will

primarily refer to sets of sequences, using the term "column" to denote the bits occupying the same position across all these sequences. We also recall that these sets can be multisets.

In this subsection, we recall theoretical results from Chapter 2, such as set-maximal exact matches and minimal positional substring cover, to better understand the novelties of this thesis. First, we recall the (k-)Set-Maximal Exact Matches definition:

Definition 19 ((k-)SMEM)**.** *Given a binary panel composed by h sequences $S = \{S_1, \ldots, S_h\}$ of length w, a pattern $P[1..w]$, and an integer k, such that $1 \leq k \leq h$, we define $P[i..j]$, where $1 \leq i \leq j \leq w$, to be a (k-)SMEM if it occurs in at least k input sequences of S and one of the following holds: a) $i = 1$ and $j = w$, b) $i = 1$ and $P[1..j+1]$ does not occur in S, c) $j = w$ and $P[i-1..w]$ does not occur in S, and d) $P[i-1..j]$ and $P[i..j+1]$ do not occur in S. If $k = 1$, we refer to a 1-SMEMs as SMEM.*

We define the problem of identifying the SMEMs in the pattern P.

Problem 2 (SMEM-finding)**.** *Given a set $S = \{S_1, \ldots, S_h\}$ of h sequences of length w and a pattern $P[1..w]$, find the list L of pairs (p, ℓ) such that for all $(p, \ell) \in L$, $P[p..p+\ell-1]$ are the SMEMs between S and P.*

We then define the problem of locating all occurrences of the SMEMs in the panel.

Problem 3 (SMEM-locating)**.** *Given a set $S = \{S_1, \ldots, S_h\}$ of h sequences of length w and a pattern $P[1..w]$, find the list L of triples (p, ℓ, O) such that for all $(p, \ell, O) \in L$, $P[p..p+\ell-1]$ is an SMEMs between S and P where O is the list of haplotypes where the SMEM occur.*

Durbin's Algorithm 5 [13] can solve Problem 3 in $\mathcal{O}(w)$-time and $\mathcal{O}(n)$-space, with $n = hw$, which corresponds to about $13n$ bytes.

Then we recall the MPSC-related problems.

Problem 4 (k-Minimal Positional Substring Cover problem, k-MPSC [25])**.** *Given a set S of h strings of length w and a string P of length w, find, if it exists, a k-positional substring cover of P by S with the smallest size over all k-positional substring covers of P by S.*

The MPSC problem is the k-MPSC problem, where k equals 1, *i.e.*, each positional substring of the cover is contained in at least one string of the panel. It is easy to see that a solution to the problem exists iff for every i, with $1 \leq i \leq w$, the positional substrings (i, i, P) are contained in at least k distinct strings of S.

Moreover, we focus on the problem variants of finding the leftmost and rightmost MPSC.

Problem 5 (Leftmost MPSC problem)**.** *Given a set S of h strings of length w and a string P of length w, find a leftmost* MPSC *C of P by S which is a* MPSC *of P by S such that any i-th substring in C starts at least as early as the i-th substring of every other* MPSC *of P by S.*

Problem 6 (Rightmost MPSC problem)**.** *Given a set S of h strings of length w and a string P of length w, find a rightmost* MPSC *C of P by S which is a* MPSC *of P by S such that any i-th substring in C ends at least as late as the i-th substring of every other* MPSC *of P by S.*

The Phasing Problem Humans are diploid beings, with chromosomes present in two copies, called haplotypes, each inherited from one of the two parents [151]. During sequencing, these two haplotypes cannot be distinguished, as reads are fragments uniformly distributed across both chromosomal copies, and distinguishing them is biologically expensive. To solve this problem, various computational methods have been developed over the past few years.

A genotype is the combined and indistinguishable information of a pair of haplotypes. In detail, a genotype is a vector of allele pairs at each locus, where the pairs consist of homozygous alleles if the two haplotype alleles are equal, or heterozygous alleles if the two haplotype alleles are not equal.

Thanks to the complex biological nature of the evolutionary phenomenon, we observe that each haplotype inherited from a parent is itself a mosaic of the two parental haplotypes, resulting from recombination events. Recombination events regulate Mendelian inheritance laws within a family trio and have been extensively studied as a combinatorial problem [151].

From a stringology perspective, as we already discussed, a bi-allelic haplotype is formalized as a binary string. In this representation, the i-th position of the

string is assigned a "0" if the individual carries the reference allele, *i.e.* the allele matches the reference in that position, and a "1" if the individual carries the alternative allele. On the other hand, the genotype is represented as a string on the alphabet $\Sigma = \{0, 1, 2\}$, where the i-th position is 0 if both haplotypes present 0 in that position, it is 1 if both haplotypes present 1 in that position, or it is 2 if the individual, at the i-th locus, has inherited the reference allele from a parent and the alternative allele from the other parent, *i.e.* having a 0 and a 1. We call positions with 0 and 1 homozygous, while the positions with 2 are called heterozygous.

Formally, we say that a pair $\langle h_1, h_2 \rangle$ of w-length haplotypes explains a w-length genotype Q if and only if, for all $i \in [1, w]$ we have that $h_1[i] = h_2[i] = Q[i]$ if $Q[i] \neq 2$, or $h_1[i] \neq h_2[i]$ if $Q[i] = 2$.

Moreover, the genotype phasing problem is determining the exact pair of haplotypes that explains the observed genotype. The problem is straightforward for homozygous positions, where each position has an unambiguous allele; however, at heterozygous positions, the phase is ambiguous because multiple pairs of haplotypes can equally explain the genotype, *i.e.* 0|1 and 1|0 differ. Multiple heterozygous positions lead to an exponential number of possible haplotypes. This is a well-known problem in the literature. For example, Gusfield proposed solving genotype phasing using the perfect phylogeny model [152, 153], having as input a collection of genotypes.

Another approach to addressing the genotype phasing problem is haplotype threading, which models haplotype reconstruction as a mosaic of reference haplotypes. This model aims to cover a haplotype query using segments from haplotypes in an input panel, also via recombination events, combining subregions from various haplotypes in the reference panel. Note that the input panel is not guaranteed to include the two exact haplotypes that explain the target genotype; therefore, the mosaic of segments may involve multiple haplotypes from the panel.

A key result in this context is the Li and Stephens (Li-Stephens, or simply LS) hidden Markov model (HMM) [145], which produces this mosaicism probabilistically and requires time linear in the size of the input panel. However, this approach is infeasible when the haplotype panel is large. Thus, time-efficient solutions have recently been investigated, mainly using the PBWT [25].

3.2 Methods

In this section, we will start by describing our attempt to build a space-efficient RLPBWT [21]. Those results helped us to develop the μ-PBWT [23], our ready-to-use tool for indexing and querying large haplotype datasets. μ-PBWT includes all the best solutions previously tested for the RLPBWT, but avoiding the need to keep in memory a compressed representation of the input panel. Then we will discuss how to extend the μ-PBWT to compute k-SMEMs and (k-)MPSC [24], and how to use the latter to solve the genotype phasing problem [27]. In this section, we will often refer to the i-th PBWT column as $\mathsf{col}(\mathsf{PBWT})_i$.

3.2.1 Matching Statistics in the PBWT

Our approaches in this chapter are primarily based on generalizing Matching Statistics (MS) within the PBWT context. The main idea to solve the SMEMs finding problem in the RLPBWT by leveraging the MS comes from the recent results in computing MEMs with the RLBWT [58, 15, 16], as discussed in Section 2.5.

Definition 20 (Matching statistics in the PBWT)**.** *Given a pattern $P[1..w]$, the matching statistics of P with respect to S is an array $\mathsf{MS}[1..w]$ of $(\mathsf{seq}, \mathsf{len})$ pairs such that, for each position $1 \leq j \leq w$, $\mathsf{MS}[j].\mathsf{seq}$ identifies a sequence in S where a longest common suffix of length $\mathsf{MS}[j].\mathsf{len}$, ending at position j in both P and $S_{\mathsf{MS}[j].\mathsf{seq}}$, occurs.*

The problem of finding SMEMs can be cast into the problem of computing matching statistics for P, as in the following lemma.

Lemma 4. *Given a pattern $P[1..w]$ and the matching statistics MS of P with respect to S, $P[i-\ell+1..i]$ is a ℓ-length SMEM shared with $\mathsf{MS}.\mathsf{seq}[i]$, for $1 \leq i \leq w$, iff $\mathsf{MS}[i].\mathsf{len} = \ell \wedge (i = w \vee \mathsf{MS}[i].\mathsf{len} \geq \mathsf{MS}[i+1].\mathsf{len})$.*

In other words, each pair in the matching statistics array represents a left-maximal exact match, shared between the pattern and the sequence of S specified in the MS array. If two consecutive pairs in the MS present non-increasing length values, it means that those matches are overlapping and that the first one is also

a right-maximal exact match. Thus, we can report the set-maximal exact match. In other words, $\mathsf{MS}[j].\mathsf{len} \geq \mathsf{MS}[j+1].\mathsf{len}$ occurs when we cannot extend to the right the current longest common suffix of length $\mathsf{MS}[j].\mathsf{len}$ shared by P and any sequence in S. For the sake of simplicity, we illustrate an example of matching statistics in Figure 3.1.

3.2.2 Data Structures for SMEM-Finding in the RLPBWT

In this thesis, I will review each data structure proposed in [21] to provide a complete description of the RLPBWT*. However, I want to clarify that I was directly involved in the design and application of the* `mapping structure`*, the sampled column permutations, the use of an* SLP *for random access and* LCE *queries, and the development of the Φ data structure. Hence, I was primarily involved in developing the* `MAP+LCE+PERM` *composite data structure. All the other solutions were proposed and implemented by other co-authors of mine. In Section 3.3, I will make clear which experiments I personally conducted. On the other hand, for [23, 24, 27], I take responsibility for the algorithmic design, carried out jointly with the co-authors, for the implementation, and for the experimental evaluation.*

To compute the matching statistics array in the PBWT, we define two queries for our RLPBWT data structures: the `start` and `extend` queries. We define smaller data structures, namely components, that support either the `start` or the `extend` query, and, in one case, *i.e.* using the Δ-encoded divergence array, support both. These components are used to build data structures for SMEM finding in the RLPBWT.

In addition, for all our solutions, we have a `mapping structure` by which we navigate the PBWT, a data structure for random accessing the input panel (not always needed), and, if required, a Φ `data structure`, used to reconstruct the complete set of PAs/DAs arrays and report all the input sequences that share a certain SMEM. We show an outline of these components in Figure 3.2, while an outline of their complexities is presented in Table 3.1.

Computing the matching statistics using `start` and `extend` Before discussing the single data structures, we need to briefly present how these two queries

(a) Input set of sequences S.

S	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	1	0	0	0	0	0	0	0	1	1	0	1
2	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1
3	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
4	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
5	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
6	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
7	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
8	0	1	0	1	0	1	0	0	0	0	0	0	1	0	1
9	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1
10	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
11	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
12	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0
13	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
14	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
15	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
16	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
17	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1
18	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1
19	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1
20	0	1	1	0	1	0	1	0	0	0	0	0	1	0	1

(b) Pattern P with matching statistics.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	0	1	0	0	1	0	1	0	0	0	1	1	1	0	1
seq	20	20	17	16	14	14	20	20	20	20	12	12	18	18	18
len	1	2	3	4	5	6	4	5	6	7	4	5	2	3	4

Figure 3.1: PBWT matching statistics example with: (a) the input set of sequences S of 20 individuals of 15 bi-allelic sites, and (b) a query pattern P and its matching statistics with respect to S. SMEMs are circled in both the pattern and the input set of sequences S. Consider, for example, $i = 6$. Here we have a left-maximal match of length six shared with sequence 14. Being $\mathsf{MS}[6].\mathsf{len} = 6 \geq \mathsf{MS}[7].\mathsf{len} = 4$, we know that in $i = 6$ also a right-maximal match is ending. Thus $P[1..6]$ share a SMEM with sequence $\mathsf{MS}[i].\mathsf{seq} = 14$. At this point, we can not report all the other sequences that share the same SMEM (*i.e.* 9, 12, and 13).

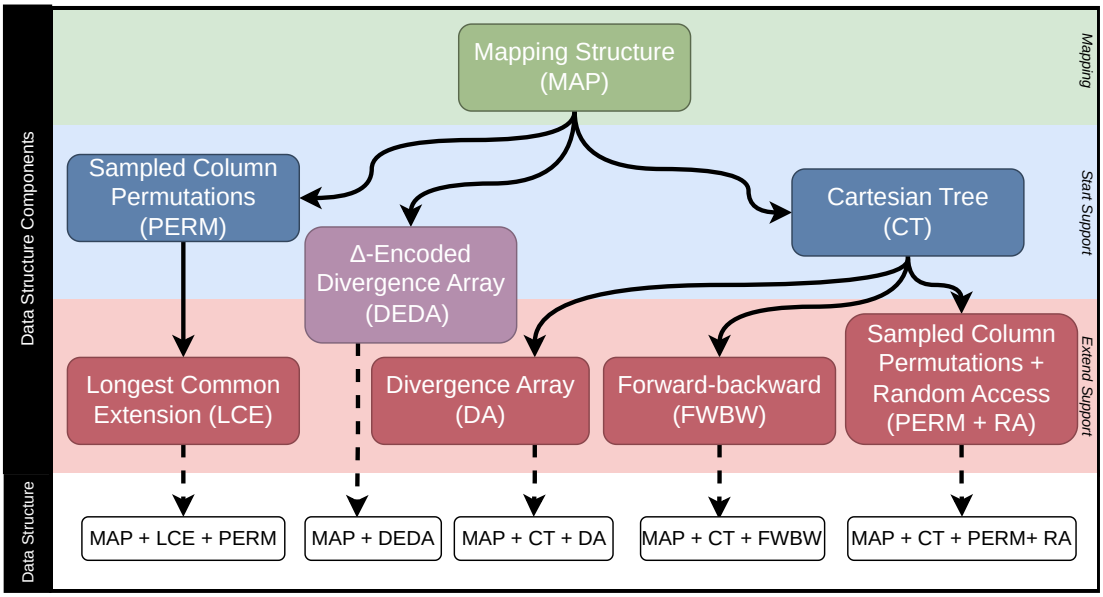


Figure 3.2: An illustration of our components and data structures to compute SMEMs in the PBWT, reporting a single input sequence per SMEM. The components are shown in colored boxes, and the data structures are shown in white boxes at the bottom. We note that the data structure for the longest common extensions depends on sampled column permutations from the level below.

help us to compute the MS array.

We start considering a random sequence in the panel as a possible match with $P[1]$. Consider a pattern substring $P[i..j]$, with $1 \leq i, j \leq w$ as a suffix of one of $S_1[1..j], \ldots, S_h[1..j]$. The `extend` query returns that we can extend the match of $P[i..j]$ to $P[i..j+1]$ if and only if $j < w$ and $P[i..j+1]$ is a suffix of one of $S_1[1..j+1], \ldots, S_h[1..j+1]$. On the other hand, the `start` query finds the smallest integer $i' \in [i..j]$ such that $P[i'..j]$ is a suffix of one of $S_1[1..j], \ldots, S_h[1..j]$. Hence, to fully compute the matching statistics array, we assume that we already computed the matching statistics up to position $i \in [1..w]$, and use the `start` query to find the smallest $i' \in [i..w]$ such that $P[i'..i' + \mathsf{MS}[i].\mathsf{len}]$ is a suffix of one of $S_1[1..i' + \mathsf{MS}[i].\mathsf{len}], \ldots, S_h[1..i' + \mathsf{MS}[i].\mathsf{len}]$. By the minimality of i', we can set $\mathsf{MS}[j].\mathsf{len} = \mathsf{MS}[j-1].\mathsf{len} - 1$ for all $j \in [i+1..i'-1]$. Then, using the `extend` query, we can find the longest prefix $P[i'..k]$ that is also a suffix of one of $S_1[1..k], \ldots, S_h[1..k]$, and we set $\mathsf{MS}[i'].\mathsf{len} = k - i' + 1$. Since $i' > i$, we can proceed by induction to compute the whole array of matching statistics. Note that

Table 3.1: RLPBWT data structures complexities outline, given ρ as the average number of runs per PBWT column, and $\hat{\rho}$ as the total number of runs in the PBWT.

Component	**Space**	**Query time**
Mapping Structure (`MAP`), bitvectors	$\mathcal{O}(\hat{\rho} \log n/\hat{\rho})$	$\mathcal{O}(\log \log n)$
Mapping Structure (`MAP`), intvectors	$\mathcal{O}(\hat{\rho})$	$\mathcal{O}(\log \rho)$
Sampled Column Permutation (`PERM`)	$\mathcal{O}(2\hat{\rho} \log h)$	$\mathcal{O}(1)$
Cartesian Trees (`CT`), *with k intervals*	$2k \log n/k + o(k \log n/k)$	$\mathcal{O}(1)$
Δ-encoded divergence array (`DEDA`)	$\mathcal{O}(\hat{\rho} \log^2 n)$	$\mathcal{O}(\log n)$
Longest Common Extension (`LCE`)	SLP *size*	$\approx \mathcal{O}(\log n)$
Random Access (`RA`), SLP	SLP *size*	$\approx \mathcal{O}(\log n)$
Random Access (`RA`), plain	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Divergence Array (`DA`)	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$
Φ data structure	$\mathcal{O}(\hat{\rho}(\log h + \log n/\hat{\rho}))$	$\mathcal{O}(\log w/\hat{\rho})$

if a column j is a complete mismatch, *i.e.* it has no bits that match the pattern bit, we set $\mathsf{MS}[j].\mathsf{seq} = -$, using $-$ as a sentinel value, and $\mathsf{MS}[j].\mathsf{seq} = 0$, resetting the MS computation in the next column, hence restarting the computation as we were in the first column.

Figure 3.3 represents the use of `start` and `extend` queries used to compute the MS array of Figure 3.1.

Mapping structure in the RLPBWT Given the position in PBWT of a bit, *i.e.* $\mathsf{PBWT}[i][j]$, the `mapping structure` can return: a) the position in $\mathsf{col}(\mathsf{PBWT})_{j+1}$ of the bit immediately to the right of $\mathsf{PBWT}[i][j]$ in $S_{\mathsf{PA}_j[i]}$, b) the position in $\mathsf{col}(\mathsf{PBWT})_j$ of the last occurrence of $\neg\mathsf{PBWT}[i][j]$ above $\mathsf{PBWT}[i][j]$, if it exists, or c) the position in $\mathsf{col}(\mathsf{PBWT})_j$ of the first occurrence of $\neg\mathsf{PBWT}[i][j]$ below $\mathsf{PBWT}[i][j]$, if it exists.

On the contrary, the first query corresponds to LF-mapping in the BWT and allows us to move from the left to the right in the PBWT, "following" in the same sequence of S in the permutations induced by the PAs. This is precisely the function described in Section 2.7 for Durbin's algorithm 5. Given: a) $u_j[i]$ as the number of zeros until i in $\mathsf{col}(\mathsf{PBWT})_j$, b) $v_j[i]$ as the amount of ones until i in $\mathsf{col}(\mathsf{PBWT})_j$, c) $c[j]$ as the total amount of zeros in $\mathsf{col}(\mathsf{PBWT})_j$, and d) $\sigma \in \{0, 1\}$ as the current pattern bit, we have that the mapping function $\mathsf{FL}[i][j]$ is equal to: a) $u_j[i] + 1$, if $\sigma = 0$, or b) $v_j[i] + c[j] + 1$, if $\sigma = 1$.

The second and third queries correspond to Rossi et al. [15] jumping up or down,

PBWT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	0	1	1	0	0	0	1	0	0	1	1	1	1
2	1	1	0	1	1	0	0	0	1	0	0	1	1	1	1
3	1	1	0	1	1	1	0	0	0	1	1	1	0	1	1
4	1	1	0	1	1	0	0	0	1	0	0	1	1	0	1
5	0	1	0	1	0	1	0	0	1	0	0	1	1	0	1
6	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
7	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0
8	0	1	0	1	1	1	0	0	0	0	0	0	1	0	1
9	0	1	0	0	1	0	0	0	1	0	0	0	1	0	1
10	0	1	0	1	0	0	0	0	1	0	0	0	0	0	1
11	0	1	0	1	1	0	0	0	0	0	0	1	1	0	1
12	0	1	0	0	1	0	1	1	0	0	0	1	0	0	1
13	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1
14	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1
15	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
17	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1
18	1	0	1	0	0	0	0	0	0	0	1	1	0	0	1
19	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1
20	0	1	0	0	0	0	0	0	0	0	1	1	0	0	1

Figure 3.3: Iteration used to compute the matching statistics array of Figure 3.1 over the PBWT for the set of sequences S depicted in Figure 3.1. At a high level, the `start` operation corresponds to vertical arrows, remaining within the same PBWT column in case of a mismatch. In contrast, `extend` corresponds to arrows between consecutive columns. We begin from an arbitrary sequence, in this case the 20th, where $\mathsf{col}(\mathsf{PBWT})_1[20] = 0$; since $P[1] = 0$, we advance to the next column, recording $\mathsf{MS}[1].\mathsf{seq} = 20$ and $\mathsf{MS}[1].\mathsf{len} = 1$. Applying the mapping function to sequence 20 moves us from the first to the second column, identifying the corresponding index $j = 15$ in $\mathsf{col}(\mathsf{PBWT})_2$, where $\mathsf{PA}_2[j] = 20$. Here we observe that $P[2] = \mathsf{col}(\mathsf{PBWT})_2[15]$, so we continue to the next column and store $\mathsf{MS}[2].\mathsf{seq} = 20$ and $\mathsf{MS}[2].\mathsf{len} = 2$. Mapping sequence 20 forward again leads to $\mathsf{col}(\mathsf{PBWT})_3[19]$, where a mismatch occurs since $P[3] \neq \mathsf{col}(\mathsf{PBWT})_3[19]$. We then consider whether to jump to the last character of the previous run, $\mathsf{col}(\mathsf{PBWT})_3[17]$, or to the first character of the next run, $\mathsf{col}(\mathsf{PBWT})_3[20]$, with $\mathsf{PA}_3[17] = 17$ and $\mathsf{PA}_3[20] = 18$. By inspecting the input panel in Figure 3.1, we see that up to column 3 (excluded), sequence 17 shares a longer suffix with sequence 20 than sequence 18 does; therefore, we select sequence 17, setting $\mathsf{MS}[3].\mathsf{seq} = 17$ and $\mathsf{MS}[3].\mathsf{len} = 3$. From this new position, we compute the mapping from column 3 to column 4 and continue in the same manner until all entries of the matching statistics array are determined.

respectively, using the so-called `thresholds` in the RLBWT when a mismatch is found. Recalling that the PBWT is built using the co-lexicographical ordering, it follows that we call this function FL-mapping and forward stepping as counterparts of the LF-mapping and the backward stepping in the BWT.

Summarizing, for each column j in the RLPBWT, which presents r runs, we store: a) the run head indices, or the run tail indices, in an r-length array p_j, b) a r-length data structure for u_j and v_j, c) the integer $c[j]$, and d) a boolean value b to retrieve the symbol corresponding to the first run. This last value is used to retrieve the current run symbol in constant time, given the current run index.

In the RLPBWT, we implement the data structures for p_j, representing the run tail indices, u_j, and v_j using run-length compressed bitvectors, occupying roughly $\mathcal{O}(\hat{\rho}\log(n/\hat{\rho}))$ bits and answering queries theoretically in $\mathcal{O}(\log\log n)$ time, where $\hat{\rho}$ is the total number of runs in the columns of the PBWT. In our implementation [21], we used the `sdsl-lite` [43] sparse bitvectors to implement them, whose theoretical bounds were already discussed in Section 2.2. For u_j, we store a bitvector as length as the number of "0"s in the PBWT columns, setting $u_j[i] = 1$ if and only if we have i symbols in the $\mathsf{rank}_{u_j}(i)$-th run of "0"s in $col(\mathsf{PBWT})_j$. For v_j, we proceed symmetrically. For example, in the RLPBWT, given $col(\mathsf{PBWT})_j = 00101111000000000000$, hence having $r = 5$, we store: a) $p_j = 01110001000000000001$, b) $u_j = 011000000000001$, c) $v_j = 10001$, d) $c[j] = 15$, and e) $b_j = \top$.

In the μ-PBWT, we store just p_j, representing the run head indices, and a single interleaved representation for the value v_j and u_j. In detail, this interleaved representation uv_j, for each integer i, with $1 \leq i \leq r$, represents the value v_j (or u_j respectively), up to the start of run i, if the i-th run consists of zeros (or ones, respectively). Both p_j and uv_j are implemented as bit-compressed intvectors of size $\mathcal{O}(r)$, with access in constant time. Hence, a single mapping operation takes $\mathcal{O}(\log r)$ time, using a binary search. For example, in the μ-PBWT, given $col(\mathsf{PBWT})_j = 00101111000000000000$, hence having $r = 5$, we store: a) $p_j = [1, 3, 4, 5, 9]$, b) $uv_j = [0, 2, 1, 3, 5]$, c) $c[j] = 15$, and d) $b_j = \top$.

Regarding thresholds, in the context of the RLPBWT, they correspond to the positions of the first minimum value of the divergence array (DA) within the range of each run in a RLPBWT column. Specifically, we must also consider the current

run interval together with the following position, due to the definition of the DA, which compares consecutive sequences in the permutation induced by the PAs. In the RLPBWT [21], thresholds are implemented by sparse bitvectors, while in the μ-PBWT [23] by bit-compressed intvectors.

`start` queries: sampled column permutations (`PERM`) Sampled column permutations (`PERM`) are used to find at least one occurrence of each SMEM. We can use the sampled column permutations as an analogue of Policriti and Prezza's [80] Toehold Lemma: for each run boundary in a column in the PBWT, we store which sequences in the input set they came from, using a total of roughly $\mathcal{O}(2\hat{\rho} \log h)$ bits for the entire RLPBWT, looking at the PA value in the corresponding positions. Having these values, we know each time which input sequence we are following: a) if we have a match, we follow the same sequence, and b) if we have a mismatch, we use thresholds to select the new run bit to follow, which is either the tail of the previous valid run or the head of the next valid run, having in both cases the PA value stored in `PERM`.

`start` queries: Cartesian tree (`CT`) To present this approach for the `start` queries, we need to define a Cartesian tree [154].

Definition 21 (Cartesian tree)**.** *Given a n-length array of integers A, a Cartesian tree $\mathcal{T}$, built upon A, is a binary tree such that: a) each element of A is stored in a node of $\mathcal{T}$, b) in $\mathcal{T}$ the value of each parent node is less than the values of its children, respecting the min-heap property, and c) A can be retrieved by performing an inorder traversal visit of $\mathcal{T}$.*

Suppose we store a representation of the shape of a Cartesian tree built upon the whole set of divergence arrays. In that case, we can support range minimum queries (RMQ) [155, 156], previous-smaller-value (PSV) [157], and next-previous-smaller-value (NSV) [157] queries on the set of DAs.

We recall that given an array $\mathsf{A}[1..n]$ of integers, a RMQ for two positions $i \leq j$ asks for the position k of the minimum in $\mathsf{A}[i..j]$, *i.e.*, $k = \text{argmin}_{k' \in [i..j]} A[k']$. We denote this query by $\mathsf{RMQ}_{\mathsf{A}}(i, j)$. Given a position i in A, we define the PSV as $\mathsf{PSV}_{\mathsf{A}}(i) = \max(\{j : j < i, \mathsf{A}[j] < \mathsf{A}[i]\} \cup \{0\})$, and NSV as $NSV_{\mathsf{A}}(i) = \min(\{j :$

$j > i, \mathsf{A}[j] < \mathsf{A}[i]\} \cup \{n+1\})$. These queries return only a position and cannot easily support random access.

We consider three representations of the tree shape: a) an augmented balanced-parentheses (BP) representation occupying roughly $2n + o(n)$ bits [158] and answering queries in constant time, b) a simple DAG-compressed representation (with each non-terminal storing the size of its expansion) answering queries in time bounded by its height, and c) an interval-tree storing selected intervals corresponding to nodes in the cartesian tree and answering queries in constant time. We do not include Gawrychowski et al.'s compressed RMQ data structure [159] because we are not aware of an implementation, and we see no easy way to estimate its space usage for storing the set of DAs.

The constructions of the BP representation and DAG-compressed representations are standard, while the interval-tree structure needs some explanation. We query the Cartesian tree while forward stepping through the PBWT, when the considered interval contains only "0"s and we want a "1", or vice versa. To proceed, we must ascend the underlying tree and widen our interval until it includes a copy of the bit we want to retrieve. Note that the query interval corresponds to a node v in the Cartesian tree, and that the PBWT interval we seek corresponds to the lowest ancestor u of v whose interval is not unary. Hence, we need to store in our interval-tree only the PBWT intervals for nodes u in the Cartesian tree such that the interval for at least one of u's children is unary but u's interval is not unary.

Since our intervals can nest but not otherwise overlap, we can store our interval tree in a more space-efficient manner than usual, writing out a string with open-parens, close-parens, and "0"s, with each open-close pair indicating an interval and the number of "0"s before, between, and after them indicating its starting point, length, and ending point. Then we encode that string as one bitvector with "0"s indicating "0"s and "0"s indicating parens, and another bitvector with "0"s indicating open-parens and "1"s indicating close-parens, having the combination of the bitvectors as a wavelet tree for the string. Finally, we store a BP representation of the tree structure of the stored intervals.

If we store k intervals, our first bitvector has $n + 2k$ bits and $2k$ copies of "1", our second bitvector has $2k$ bits with k copies of "0" and k copies of 1, and the tree structure has k nodes and so its BP representation takes $2k + o(k)$

bits. Overall for the `CT` structure we use roughly $2k \log \frac{n+2k}{2k} + 2k + 2k + o(k) = 2k \log(n/k) + o(k \log(n/k))$ bits, and can answer queries in constant time.

We note that, since even our query intervals can nest but not contain or otherwise overlap any of our stored intervals, we can query with a single endpoint instead of a whole interval.

`extend` queries: divergence array (`DA`) and Δ-encoded DA (`DEDA`) A naive, uncompressed divergence array is the simplest possible data structure to support the `extend` query, which finds the length of each SMEM. It is also the heaviest in memory, requiring roughly $\mathcal{O}(n \log n)$ bits for the set of DAs. In reality, each DA requires $\mathcal{O}(h \log h)$ bits; therefore, having w DAs, the $\mathcal{O}(n \log n)$ bound is clearly an overestimation, but it is helpful for comparison with other solutions.

Instead of the DA we can use a Δ-encoded divergence array, storing each entry of $\mathsf{DA}[i][j]$ with: a) $\mathsf{DA}[i][j] = 0$ if $i = 1$, and b)the difference $\mathsf{DA}[i][j] - \mathsf{DA}[i-1][j]$ if $i > 1$. If the PBWT is highly run-length compressible, then the Δ-encoded DA is small. In fact, if $\mathsf{PBWT}[i..i+\ell-1][j]$ is a run of equal bits in the j-th column of the PBWT and $\mathsf{col}(\mathsf{PBWT})_{j+1}[i'..i'+\ell-1]$ are the bits immediately to their right in the input set of sequences, then $\mathsf{DA}[i'+k][j+1] = \mathsf{DA}[i+k][j]+1$ for $1 \leq k \leq \ell-1$. Therefore, $\mathsf{DA}[i'+k][j+1] - \mathsf{DA}[i'+k-1][j+1] = \mathsf{DA}[i+k][j] - \mathsf{DA}[i+k-1][j]$ for $2 \leq k \leq \ell-1$, so the Δ-encoded of $\mathsf{DA}[i'+1..i'+\ell-1][j+1]$ is the same as that of $\mathsf{DA}[i+1..i+\ell-1][j]$. It follows that, having $\hat{\rho}$ runs in the PBWT, the (linearized) Δ-encoded DA has a string attractor of size $\mathcal{O}(\hat{\rho})$ and, thus, it can be represented using a SLP, which requires $\mathcal{O}(\hat{\rho} \log^2 n)$ bits [160, Lemma 3.14].

Moreover, increasing the size of this SLP by a small constant factor, we can store at each non-terminal the length, sum, and minimum prefix sum of its expansion, supporting random access, RMQ, PSV, and NSV queries on the divergence array in $\mathcal{O}(\log n)$-time. This is similar to how Gagie et al. [82] used an SLP for their Δ-encoded LCP array, and this allows us to use `DEDA` to support both `start` and `extend` queries.

`extend` queries: longest common extension (`LCE`) We already defined LCE queries in Section 2.1 and how to use an SLP to implement them, answering queries in approximately $\mathcal{O}(\log n)$.

Suppose we have arrived at column $j+1$ in the PBWT and we know that the longest suffix of pattern $P[1..j]$ that occurs in some sequences of S ending at column j has an occurrence immediately followed by $\mathsf{PBWT}[i][j+1]$. Suppose we also know from which sequence of the set S that bit $\mathsf{PBWT}[i][j+1]$ comes from. If $P[j+1] = \mathsf{PBWT}[i][j+1]$ then the longest suffix of $P[1..j+1]$ that occurs in in some sequences of S ending at column $j+1$ has an occurrence ending with $\mathsf{PBWT}[i][j+1]$. Therefore, we assume $P[j+1] \neq \mathsf{PBWT}[i][j+1]$. By the definition of the PBWT, there is an occurrence of the longest suffix of $P[1..j+1]$ that occurs in some sequences of S ending at column $j+1$, ending either at the last occurrence of $P[j+1] = \neg\mathsf{PBWT}[i][j+1]$ above $\mathsf{PBWT}[i][j+1]$ (if it exists), or at the first occurrence of that bit below $\mathsf{PBWT}[i][j+1]$ (if it exists). We recall that we use the mapping structure to quickly find these occurrences of that bit.

`extend` **queries: forward-backward (`FWBW`)** With this method, we rely on a Cartesian tree to ensure the following invariant: the search interval in the PBWT contains all the bits immediately following occurrences of the longest suffix of the pattern prefix processed so far, restricted to the desired columns of the PBWT. If a copy of the next bit of the pattern appears within this search interval, we can continue with a forward step without accessing the Cartesian trees. The Cartesian trees are queried just when the interval does not contain a copy of the bit in the pattern, marking that we have reached the right end of an SMEM. Using the mapping structure and the Cartesian trees, we can therefore locate all right endpoints of SMEMs. By maintaining analogous structures for the reversed input set of sequences, we can also locate all the left endpoints. Thanks to the maximality property of SMEMs, we can easily pair these endpoints and obtain the SMEMs. This approach, however, doubles both time and space usage.

Data structure for Random access (`RA`) This last component provides efficient random access to the input panel and can be used to determine the length of a given SMEM. Although the total length of the SMEMs can be quadratic in the length of the pattern, we can leverage their non-nesting property to prove that we need only a linear number of random accesses to the input to compute them. If we combine a random access data structure with Cartesian trees, then the number

of random accesses is equal to the number of SMEMs, and the total length of the sequence needed to extract from S is linear in the total length of the SMEMs. As a random access data structure, we can use: a) an SLP of all the sequences in S answering queries in $\mathcal{O}(\log n)$ time, and b) a plain and constant time access representation of the input panel (with 8 bits packed into each byte) occupying roughly n bits.

Φ data structure Using PERM, we have PA values only at each run boundary. However, to retrieve all the rows/sequences that share a certain SMEM, we at least need access to the entire set of PAs. For this purpose, we store a small data structure that answers queries corresponding to φ and φ^{-1} in the string/suffix array context: given a column index and a prefix array value j, it returns the previous and next values. These two values correspond to the sequences that we need to consider for finding common suffixes with sequence j, *i.e.* other possible sequences in S that share the same SMEM. We refer to these as Φ queries in the PBWT.

Formally, given an index column k, we let IPA_k be the inverse permutation of PA_k, *i.e.* $\mathsf{IPA}_k[\mathsf{PA}_k[i]] = i$. Hence, we define the Φ function, similarly to classical the string context [68], for all $1 < \ell \leq h$, as $\Phi_k(\ell) = \mathsf{PA}_k[\mathsf{IPA}_k[\ell] - 1]$. Therefore if $\mathsf{IPA}_k[\ell] = i$, or equivalently $\mathsf{PA}_k[i] = \ell$, we have that $\Phi_k(\mathsf{PA}_k[i]) = \mathsf{PA}_k[i-1]$. In less formal words, given a value of PA_k in position i, the Φ function returns the preceding value of PA_k in position $i-1$. Analogously, we can define the inverse of Φ for all $1 \leq i < h$ as $\Phi_k^{-1}(i) = \mathsf{PA}_k[\mathsf{IPA}_k[i] + 1]$. For example, assuming $\mathsf{PA}_6 = [15, 16, 1, 10, 11, 17, 9, 12, 13, 14, 19, 20, 2, 3, 4, 18, 5, 6, 7, 8]$ and $i = 3$, we have that $\Phi_6(4) = 3$ and $\Phi_6^{-1}(3) = 18$.

To understand how the Φ function works, note that in a PBWT column, when two consecutive symbols are equal (either "0" or "1"), their corresponding input sequence indices values in the PA remain consecutive in the permutation of the next PBWT column. This holds even though they may occupy two consecutive, but different, positions with respect to the previous PBWT column after applying the FL function, due to the stability of the sorting algorithm underlying the PBWT. Formally, for all $1 \leq j < w$ and for all $1 \leq i < h$, if $\mathsf{col}(\mathsf{PBWT})_j[i] = \mathsf{col}(\mathsf{PBWT})_j[i-1]$ then $\mathsf{FL}[i][j] = \mathsf{FL}[i-1][j]+1 = k$ and therefore, $\mathsf{PA}_j[i] = \mathsf{PA}_{j+1}[k]$ and $\mathsf{PA}_j[i-1] = \mathsf{PA}_{j+1}[k-1]$. We can leverage this property by storing the PA

samples at the beginning and at the end of each PBWT run, and the whole PA_w column. Then, we can compute the value of $\Phi(\mathsf{PA}_j[i])$, *i.e.* $\mathsf{PA}_j[i-1]$, by applying FL mappings as long as the corresponding PBWT values are the same. Assuming k is the column and i' is the sequence corresponding to the PBWT values mismatch, we have that $\mathsf{PA}_k[i'] = \mathsf{PA}_j[i]$ and $\mathsf{PA}_k[i'-1]$ is sampled, being at the end of a run, and we can retrieve the value of $\mathsf{PA}_j[i-1] = \mathsf{PA}_k[i'-1]$. An example of iterative FL mapping to perform Φ queries is depicted in Figure 3.4. For the Φ^{-1} function, we do symmetrically.

We observe that to check if a sequence shares the same SMEM directly, we can use DA samples together with the information of the current sequence, which shares a SMEM with the pattern and the next/previous sequence retrieved by the Φ function. Hence, we extend the data structure to also encode the whole set of DAs. If we store the DA sample at the beginning of each PBWT run, while computing the Φ function for $\mathsf{PA}_j[i]$, we can recover the value of $\mathsf{DA}_j[i]$ as $\mathsf{DA}_k[i'] - (k-j)$, *i.e.* removing from the sampled value $\mathsf{DA}_k[i']$ the number of times we applied the FL mapping. This approach can be inefficient if two prefix array values "diverge" after many PBWT columns, requiring the mapping structure to be queried many times. The other adjacent DA value can be retrieved symmetrically, while computing the Φ^{-1} function.

To solve this inefficiency and avoid performing $\mathcal{O}(k-j-1)$ FL steps, we can store a successor data structure which marks in which column each sequence appear as PA sample at the beginning of a run and storing the corresponding sample at the end of the previous run and the DA sample as satellite information. We also need to store the data structures for when a PA value is at the end of a run, store the corresponding DA value, and store the PA values sampled at the head of the next run. In detail, we can store, for each sequence, two w-length sparse bitvectors, with a bit set in each position where the considered sequence is the head/tail of a run, respectively. We also store two support bit-compressed intvectors: one storing the previous PA value and the other storing the next PA value. Given a sequence a, we have the corresponding data structures: a)rank_a^{Φ}, which performs rank queries on the sparse bitvector, and b) the support bit-compressed intvector Φ^a_{supp}. Hence, in column k we have: a) $\Phi_k(a) = null$ if $\Phi^a_{supp}[\mathsf{rank}_a^{\Phi}(k)] = h+1$, *i.e.* a is the first PA value not having a predecessor, or b) $\Phi_k(a) = \Phi^a_{supp}[\mathsf{rank}_a^{\Phi}(k)]$ otherwise.

On contrary, assuming the corresponding data structure for $\Phi^{-1\,a}_{supp}$ we have: a) $\Phi^{-1}_{k}(a) = null$ if $\Phi^{-1\,a}_{supp}[\mathsf{rank}^{\Phi^{-1}}_{a}(k)] = h+1$, *i.e.* a is the last PA value not having a successor, or b) $\Phi^{-1}_{k}(a) = \Phi^{-1\,a}_{supp}[\mathsf{rank}^{\Phi^{-1}}_{a}(k)]$ otherwise. In both construction cases, we store a sentinel value $h+1$ as a predecessor/successor, depending on whether we are considering the first/last PA value.

Gagie et al. in [82] showed that the Φ function, and consequently the Φ^{-1} function, for the suffix array can be stored in $\mathcal{O}(r)$ words, and evaluated in $\mathcal{O}(\log\log_{\eta}(n/r))$-time, where n is the size of the indexed text, r the number of runs in the BWT, and $\eta = \Omega(\log n)$. In the RLPBWT and μ-PBWT, the Φ data structure implemented with sparse bitvectors takes $\mathcal{O}(\log w/\hat{\rho})$ time to perform a single Φ /Φ^{-1} operation, with $\hat{\rho}$ as the total number of runs in the PBWT. The whole data structure takes approximately $\mathcal{O}(\hat{\rho}(\log h + \log n/\hat{\rho}))$ space. Theoretically, we can store a total of $\hat{\rho}$-length intvectors instead of sparse bitvectors, in $\mathcal{O}(\hat{\rho})$ space, computing the two functions in $\mathcal{O}(\log\hat{\rho})$ time using binary searches.

Note that we are aware that Nishimoto and Tabei [161] and Bertram et al. [162] recently improved the time of the Φ function in the BWT to $\mathcal{O}(1)$-time, but an inspired approach is not implemented in either our RLPBWT nor in the μ-PBWT.

3.2.3 Composite data structures for the RLPBWT

In the previous subsections, we have already discussed how to use `MAP+CT+FWBW` and `MAP+CT+PERM+RA` to compute SMEMs, in the previous section, while describing the Cartesian tree and the forward-backwards data structures. Now we discuss the other approaches introduced in Figure 3.2: a) `MAP+CT+DA`, b) `MAP+DEDA`, and c) `MAP+LCE+PERM`.

Composite data structures: `MAP+CT+DA` and `MAP+DEDA` For these two approaches, we recall what we discussed about Cartesian trees.

Consider the scenario in which we have a search interval I that contains only "0" bits, but we want to find a "1" bit. The other way follows by symmetry. We query for the minimum value v in the DA, and compute the positions a and b of the previous and next smaller value of v, respectively, having as the invariant that $I \subset (a..b)$ and $(a..b)$ matches a node in the Cartesian tree. Finally, accessing

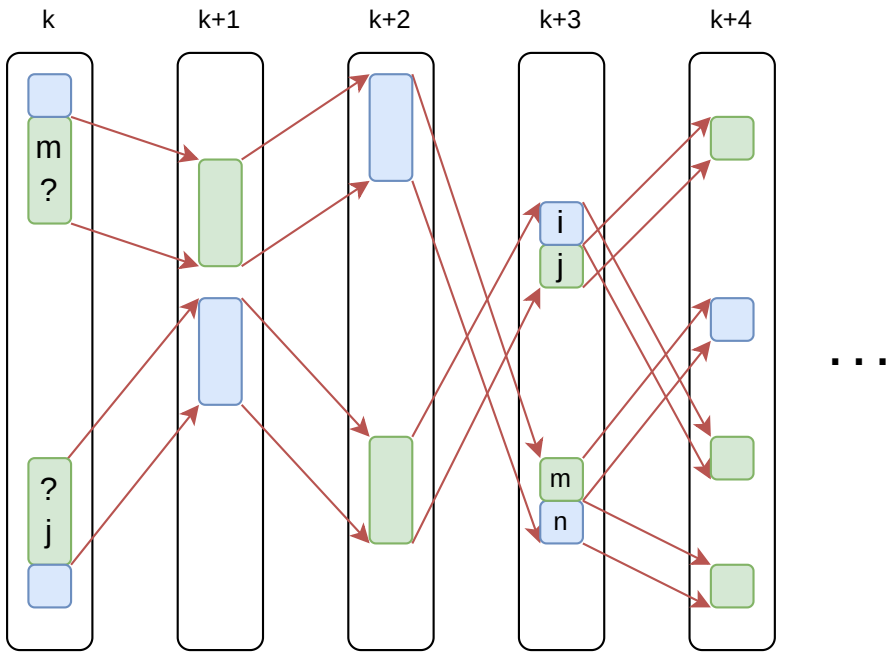


Figure 3.4: Assume that with green blocks we denote bits of value 0 and with blue blocks bits of value 1, and suppose we want to compute $\Phi_k(j)$. Following the FL mapping, these two equal bits in column $k+1$ correspond to two consecutive positions that have two bits of the same value. The same happens when going from the $(k+1)$-th column to the $(k+2)$-th column. Another FL mapping iteration reaches two different bits in column $k+3$. In this column, $\Phi_k(j)$ is at a run boundary (by definition of run, having two consecutive different bits). Hence, we can extract its value i from PA samples, understanding that $\Phi_k(j) = i$ also at columns k, $k+1$, and $k+2$. In the symmetric case, $\Phi_k^{-1}(m) = n$ is computed as illustrated in the figure.

the DA, we can determine the length of a match. To obtain RMQ, NSV, and PSV support, and compute SMEMs, we can use the Δ-encoded DA (`MAP+DEDA`) or, as already discussed, a Cartesian tree combined with the DA (`MAP+CT+DA`).

We now briefly present the `MAP+DEDA` composite data structure. Suppose we store a linearized version of the DA in memory. To access $\mathsf{DA}[i][j]$, we need to access the t-th entry in the linearized DA, where $t = ih + j$. Moreover, by definition of the Δ-encoding, $\mathsf{DA}[i][j]$ is equal to the sum of the first t values in the linearized Δ-encoded DA. Hence, we can compute $\mathsf{DA}[i][j]$ like we would calculate a partial sum of keys in a binary search tree. In detail, as a binary search tree, we can use the SLP parse tree built over the linearized Δ-encoded DA. We start at the root of the SLP's parse tree, and we traverse the tree up to its t-th leaf. Note that we can descend the tree in time proportional to its depth, because we have stored the sizes of the non-terminals' expansions. As we descend the path, we sum all the first $t - 1$ numbers in the Δ-encoded DA. We can perform this operation because these numbers are in the expansions of symbols that are not on the path, which are the left children of symbols on the path, and we have previously stored the sums of those left children. Finally, we add the t-th number in the Δ-encoded DA, which is stored at the t-th leaf.

To compute SMEMs, we also need to compute RMQs, PSV, and NSV, as we discussed for the Cartesian tree. For RMQs, we use the fact that the sum of the Δ-encoded entries to the left of a non-terminal's expansion, plus the minimum prefix sum of its expansion, *i.e.* of Δ-encoded entries, is the minimum entry in the corresponding interval of the plain DA. Given a query range $[i..j]$, we can find $\mathcal{O}(\log n)$ subtrees of the parse tree whose leaves are the i-th through j-th, and return the minimum plain DA entry in that interval. For PSV queries, we descend to the specified t-th leaf of the parse tree, and we trace back until we find a left child v hanging off the path we just traversed, whose expansion includes a smaller DA value than the t-th. Finally, we descend into and find the rightmost entry in v's expansion smaller than the t-th entry. NSV queries are symmetric. By being able to compute RMQs, PSV, and NSV, we can apply the same approach discussed for the Cartesian tree to compute the SMEMs, utilising the `DEDA` data structure for both `start` and `extend` queries.

Composite data structures: `MAP+LCE+PERM` This algorithm is strongly inspired by Boucher et al. [16] and computes the matching statistics in a single-pass over the input pattern P, scanned from left to right. We need to build an SLP of the concatenation of all the sequences in S, where each sequence is encoded reversed to have LCE queries from the right to the left. At each step, we use the LCE to compute the length of the longest common suffix between the longest match of the pattern at the current position, and the sequences in S, to detect the sequence with the longest match with the pattern.

We assume that we have computed the matching statistics up to position $k-1$ and are now processing the k-th column. Let i be the sequence of the PBWT that matches the longest suffix of $P[1..k-1]$ that is suffix of $S_1[1..k-1], \ldots, S_h[1..k-1]$, and let p be the corresponding sequence in S. Hence, for all $j \in [1..h]$, we have $\mathsf{lcs}(P[1..k-1], S_{\mathsf{PA}_k[i]}[1..k-1]) \geq \mathsf{lcs}(P[1..k-1], S_{\mathsf{PA}_k[j]}[1..k-1])$ with $p = \mathsf{PA}_k[i]$. If we have a match with the pattern, *i.e.* $\mathsf{col}(\mathsf{PBWT})_k[i] = P[k]$, then seq i can be used to extend the match, hence we can assign $\mathsf{MS}[k].\mathsf{seq} = p$, $\mathsf{MS}[k].\mathsf{len} = \mathsf{MS}[k-1].\mathsf{len}+1$, $i = \mathsf{LF}(i,k)$, and p does not change. Otherwise, if if we have a mismatch with the pattern, *i.e.* $\mathsf{col}(\mathsf{PBWT})_k[i] \neq P[k]$, let $\mathsf{col}(\mathsf{PBWT})_k[s..e]$ be a maximal run containing position i, then the longest suffix of $P[1..k]$ that is suffix of $S_1[1..k], \ldots, S_h[1..k]$ is either the one corresponding to the preceding end or following start of a run of $P[k]$ in $\mathsf{col}(\mathsf{PBWT})_k$ with respect to position i, *i.e.* either $S_{\mathsf{PA}_k[s-1]}[1..k]$ if $s > 1$ or $S_{\mathsf{PA}_k[e+1]}[1..k]$ if $e < n$. Having PA sampled at run boundaries and knowing p, we can use the LCE queries to compute the two longest common suffixes above, and then continue matching using the sequence corresponding to the longest common suffix. Finally, by accessing the Φ data structure, we retrieve all occurrences of the same SMEM. Note that, in the RLPBWT, the Φ data structure stores only the PAs.

3.2.4 From the RLPBWT to the μ-PBWT

The main limitation of the RLPBWT implemented as `MAP+LCE+PERM` is computing the SLP to support LCE queries for large panels. In cases like the UK Biobank data, computing the SLP is unfeasible. To address this problem, we developed the μ-PBWT, which no longer relies on LCE but instead uses a reverse-mapping

function for `extend` queries.

Figure 3.5 shows all the μ-PBWT components. In terms of RLPBWT nomenclature, the μ-PBWT can be named as `MAP+PERM`. The mapping structure and thresholds are implemented using bit-compressed intvectors rather than sparse bitvectors, as they have proven more efficient in practice for real haplotype datasets. Moreover, PA and DA `Samples` in Figure 3.5 correspond to the `PERM` component of the RLPBWT, augmented with DA samples at run boundaries. Finally, the Φ data structure is used to store both the PAs and the DAs. As anticipated, to avoid using LCE queries, we use the reverse FL mapping, which can be performed in the worst case for a previous PBWT column with r runs in $\mathcal{O}(r)$ time. Formally, given the position of the bit $\mathsf{PBWT}[i][j]$ in the PBWT, this function, namely LF function to be consistent with the FL function, return the position in column $\mathsf{col}(\mathsf{PBWT})_{j-1}$ of the bit immediately to the left of $\mathsf{PBWT}[i][j]$ in $S_{\mathsf{PA}_{j}[i]}$. Iterating the LF function, we can virtually reconstruct the suffixes up to a certain column of a pair of sequences in S, computing the LCE between them. Note that we do not need to store any additional information regarding the mapping structure. Moreover, to be able to apply the LF function, we must keep track of the exact position in the original PBWT column where we fall after applying the FL function. Hence, in the μ-PBWT algorithm, the MS array is now a triplet of arrays, rather than a pair.

Overall, the complete set of data structures that compose the μ-PBWT takes $\mathcal{O}(\hat{\rho})$ space, with $\hat{\rho}$ as the total number of runs in the PBWT, to compute the MS array and all the SMEMs. As already mentioned, we have not performed a comprehensive time analysis of the μ-PBWT query algorithm. However, having $\hat{\rho}$ as the total number runs in the PBWT, and γ as the number of SMEMs, we can conjecture that the μ-PBWT computes all SMEMs in *Avg.* $\mathcal{O}(w(\hat{\rho} + \log \hat{\rho}) + \gamma \log(w/\hat{\rho}))$ time, but it takes $\mathcal{O}(w(w\hat{\rho} + \log \hat{\rho}) + \gamma \log(w/\hat{\rho}))$ time in the worst case, after a $\mathcal{O}(n)$ time preprocessing. In fact, computing the seq array of the matching statistics requires $\mathcal{O}(w \log \hat{\rho})$ time on average, involving w iterations of the FL mapping and w iterations of the LF mapping, assuming limited overlap among SMEMs. In the worst case, when all SMEMs overlap, meaning that we have a SMEM ending at every position $i \in [1..w]$, we require $\mathcal{O}(w^2)$ iterations of the LF mapping. Finally, evaluating the Φ function γ times takes $\mathcal{O}(\gamma \log(w/\hat{\rho}))$ time.

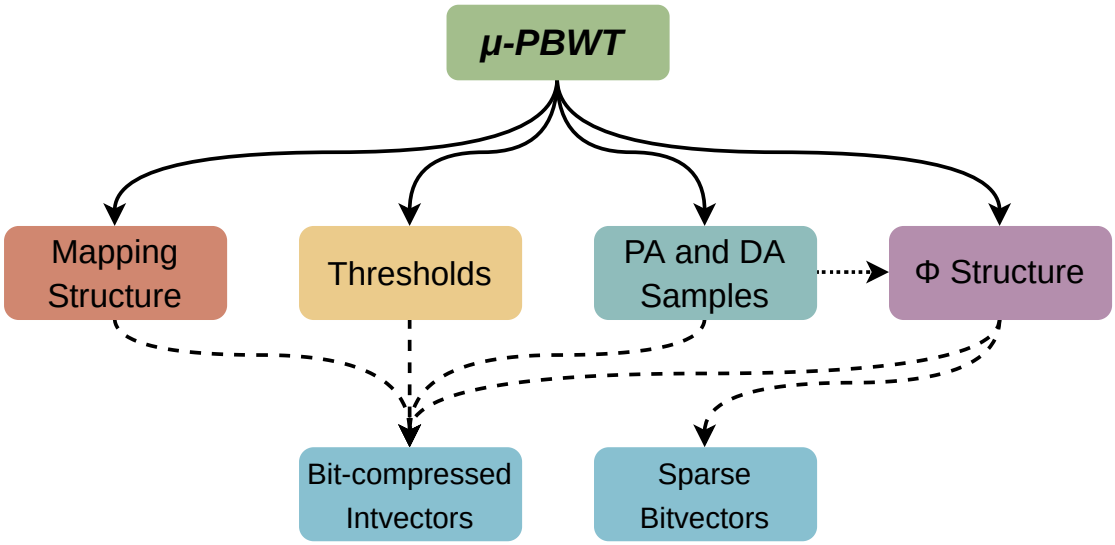


Figure 3.5: An illustration of the μ-PBWT data structures. PA and DA Samples are the PERM component of the RLPBWT, augmented with DAs samples at run boundaries. Dashed lines indicate the type of succinct data structure used to implement each component. The dotted line between PA and DA Samples and Φ Structure is used to recall that the latter, to be able to reconstruct the full sets of PAs and DAs, is built using PA and DA samples

Computing k-MS and k-SMEMs using μ-PBWT

In 2023, Tatarnikov et al. [163] extended MONI [15] to compute k-MEMs, *i.e.* maximal exact matches of a pattern against a text that occur at least k times in text. Being the μ-PBWT algorithm strongly inspired by [15], we developed an approach to compute k-SMEMs, *i.e.* SMEMs occurring in at least k sequences of the input set of sequences S. Our algorithm starts with the computation of the k-MS array, which extends the definition of MS for the PBWT with the constraints to consider matches with at least k sequences from the haplotype set S.

Definition 22 (k-Matching Statistics in the PBWT)**.** *Given a binary panel composed by h sequences $S = \{S_1, \ldots, S_h\}$ of length w, a pattern $P[1..w]$ and a value k, we define the k-Matching Statistics of P with respect to S as an array $k\text{-MS}[1..w]$ of $(\mathsf{seq}, \mathsf{len})$ pairs such that, for each position $1 \leq j \leq w$: a) $S_{k\text{-MS}[j].\mathsf{seq}}[j - k\text{-MS}[j].\mathsf{len} + 1..j] = P[j - k\text{-MS}[j].\mathsf{len} + 1..j]$, i.e. there exists a match of length $k\text{-MS}[j].\mathsf{len}$ shared between P, $k\text{-MS}[j].\mathsf{seq}$ and at least other $k - 1$ sequences in S that ends in j, b) $P[j - k\text{-MS}[j].\mathsf{len}..j]$ does not occur as a suffix ending in the j-th column in any subset S' of sequences of S with $|S'|$ greater or equal of k, i.e. we guarantee the left maximality of the match, and c) $k\text{-MS}[j].\mathsf{seq} = -$ and $k\text{-MS}[j].\mathsf{len} = 0$ iff $P[j]$ does not occur at least k time in column j.*

To compute the k-MS, we need to define and store an additional $\mathcal{O}(r)$-length array for a PBWT column with r runs.

Definition 23 (k-support values)**.** *Given a PBWT column j, a run endpoint index b in the $(j-1)$-th column and the corresponding interval $\mathsf{DA}_j[\mathsf{FL}(b)-k+2..\mathsf{FL}(b)+k-1]$, namely k-interval, we define $(\mathsf{off}_b, \mathsf{L}_b)$ as the k-support values of the end point b, where: a) off_b stores the offset from $\mathsf{FL}(b)$ to get the beginning of the sub-interval of size $k-1$ of the k-interval that maximizes the minimum divergence array value d across all the possible sub-intervals of size $k-1$. If this interval starts in $\mathsf{FL}(b)+1$, we have $\mathsf{off}_b = -1$, and b) $\mathsf{L}_b = d$.*

Less formally, k-support values determine, for a run endpoint b, the $(k-1)$-length sub-interval of $\mathsf{DA}_j[\mathsf{FL}(b) - k + 2..\mathsf{FL}(b) + k - 1]$ which has the longest possible common suffix (of length d) shared by all the $k-1$ sequences stored in the

	DA_6	1	2	3	4	5	$col(\mathsf{PBWT})_6$
1	0	0	0	0	1	0	0
2	3	0	1	0	1	0	1
3	5	0	1	0	1	0	1
4	5	0	1	0	1	0	0
5	1	0	1	0	0	0	0
6	4	1	1	0	0	0	1
7	1	1	0	0	1	0	0
8	3	0	1	0	1	0	1

Figure 3.6: k-support values example with $k = 4$. The figure shows the co-lexicographical ordering up to column 5 used to compute $col(\mathsf{PBWT})_6$ and DA_6. Suppose that some run boundary b in $col(\mathsf{PBWT})_5$ is mapped to $col(\mathsf{PBWT})_6[4]$ (the green 0 in the $col(\mathsf{PBWT})_6$ column). We consider the 4-interval $\mathsf{DA}_6[\mathsf{FL}(b) - k + 2..\mathsf{FL}(b) + k - 1] = \mathsf{DA}_6[2..7]$, circled in green in the DA_6 column. We now consider all the possible subintervals of size $k-1$, here 3, considering their minimum divergence array value. By definition, $\mathsf{DA}_j[i]$ compares sequence i to sequence $i-1$ in the co-lexicographical ordering, so, with $k = 3$, we are considering four sequences. For example, with $\mathsf{DA}_6[2]$, we are taking into account also sequence 1 to compute that divergence array value. These subintervals, including the additional sequences, are identified by circles in the panel. We are interested in the optimal subinterval, which is the one with the maximum common extension to the left. In this example, the orchid one, *i.e.* $\mathsf{DA}_6[2..4]$, is the optimal one, with the maximum extension to the left $d = 3$. In conclusion, we store the offset $\mathsf{off}_b = 2$ and $\mathsf{L}_b = 3$.

same sub-interval in PA_j. Note that, by the DA definition, we also need to include the previous sequence; hence, we need to include the sequence that precedes the interval in this set of sequences. In Figure 3.6 we illustrate an example of k-interval and k-support values.

We can compute the k-support values or at indexing time, accessing the plain PA/DA in constant time, or at querying time using the Φ data structure. In the first case, we can compute the k-support values in $\mathcal{O}(k)$ time, while in the second case, we can compute these values in $\mathcal{O}(k \log n/\hat{\rho})$ time, where $\hat{\rho}$ is the total number of runs in the PBWT. It follows that precomputing these values increases the indexing time and the index size in both RAM and on disk, since it requires storing $\mathcal{O}(\hat{\rho})$ additional values. However, this enables constant-time queries. Conversely, computing them only when needed avoids this overhead but increases query time. The theoretical increase in query time is greater than that in indexing time, as will

be confirmed by our experimental results.

Despite increasing the space requirements in RAM and on disk, precomputing these values does not change the μ-PBWT space complexity, as they require, for a PBWT column with r runs, $\mathcal{O}(r)$ space, storing two additional integers at each run boundary. hence, we add $\mathcal{O}(\hat{\rho})$ space to the μ-PBWT, with $\hat{\rho}$ as the total number of runs of the PBWT.

We will now discuss how to augment the μ-PBWT algorithm used to fill the MS array to compute the k-MS array. To compute the k-MS.len array, we need to store and update two additional arrays, namely len_k and len_t. Moreover, for the PBWT column j, we add a support index s_j. This index is a position in the column j and verifies whether at least k sequences in S have the same left-maximal match up to the j-th column. In other words, s_j marks the beginning of an interval in the PBWT column, ensuring that there are at least k sequences for the current match represented by the k-MS. Given s_j, len_k stores in position j the length of the left-maximal matches shared by at least k sequences in the set S, which are marked by the sub-interval that we get from s_j. On the other hand, len_t acts as the classical MS.len array, as we are considering $k = 1$. In column j, $\mathsf{len}_t[j]$ stores the length of the semi-left maximal match shared between the input sequence k-MS$[j]$.seq and the pattern. We say semi-left maximal because, unlike classical the MS array, we cannot extend to the left a match if we break the k sequences constraints, *i.e.* if a column i, such that $1 \leq i < j$, does not contain at least k symbols $P[i]$. See Figure 3.7 for an example of len_k and len_t. Note that len_k and len_t are not entirely stored in memory, needing just two variables which store their values in column j.

To compute the k-MS array we start from the first column and if the it contains at least k occurrences of the symbol $P[1]$, then we assign the values k-MS$[1]$.seq $= s$ and k-MS$[1]$.len $= 1$, where s belongs to the set S with the condition that $S_s[1] = P[1]$. On contrary, if there are fewer than k instances of $P[1]$ in the first column, we set k-MS$[1]$.seq to a sentinel value "$-$" and k-MS$[1]$.len $= 0$.

To update k-MS$[j]$ from k-MS$[j-1]$, with $j > 1$, we need to follow k-MS$[j-1]$.seq using the FL function, trying to extend to column j the pseudo left-maximal match obtained in column $j - 1$. Mapping to position i in the j-th column in the PBWT, we need to consider two possible cases: a) we have a mismatch with $P[j]$ with the bit i in col(PBWT)$_j$, and b) we have a match with pattern $P[j]$ with the bit i in

S	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	1	0	0	0	0	0	0	0	1	1	0	1
2	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1
3	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
4	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
5	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
6	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
7	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
8	0	1	0	1	0	1	0	0	0	0	0	0	1	0	1
9	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1
10	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
11	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
12	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0
13	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
14	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
15	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
16	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
17	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1
18	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1
19	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1
20	0	1	1	0	1	0	1	0	0	0	0	0	1	0	1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	0	1	0	0	1	0	1	0	0	0	1	1	1	0	1
seq	17	17	17	16	14	14	–	19	12	12	12	12	18	18	18
len_t	1	2	3	4	5	6	0	1	2	3	4	5	2	3	4
len_3	1	2	3	4	5	6	0	5	6	5	2	3	7	8	9
len	1	2	3	4	5	6	0	1	2	3	2	3	2	3	4

Figure 3.7: Example of 3-SMEMs results. We consider an input set of sequences S consisting of 20 sequences of length 15, and a pattern P of the same length. We show the 3-SMEMs using the same color in S and P. In addition, we show the k-MS array, with all the values that represent a SMEM underlined, the len_3 array, and the len_t array.

$\mathsf{col}(\mathsf{PBWT})_j$.

Assume we have a mismatch, *i.e.* $\mathsf{col}(\mathsf{PBWT})_j[i] \neq P[j]$. We need to consider some different sub-cases. If $\mathsf{col}(\mathsf{PBWT})_j$ does not contain at least k occurrences of $P[j]$, having a complete mismatch, we set k-$\mathsf{MS}[j].\mathsf{seq} = -$ and $\mathsf{len}_t[j] = \mathsf{len}_k[j] = 0$, resetting the k-MS computation in the next column, restarting the computation as we were in the first column. Otherwise, we proceed as in the classical MS algorithm, having k-$\mathsf{MS}[j].\mathsf{seq}$ equal to $\mathsf{MS}[j].\mathsf{seq}$ and $\mathsf{len}_t[j]$ equal to $\mathsf{MS}[j].\mathsf{len}$. Then we need to select the new run boundary b as in the classical MS and update the new variables $\mathsf{len}_k[j]$ and s_{j+1} by leveraging the k-support values as follows: a) $\mathsf{len}_k[j] = L_b$, and b) $s_{j+1} = \mathsf{FL}(b) - \mathsf{off}_b$. We recall that $\mathsf{len}_k[j]$ and s_{j+1} mark the beginning of the sub-interval of size k, consisting of sequences that share the longest common suffix of length $\mathsf{len}_k[j]$ up to column j.

Assume now to have a match, *i.e.* $\mathsf{col}(\mathsf{PBWT})_j[i] = P[j]$. Again, we have some sub-cases. If $\mathsf{col}(\mathsf{PBWT})_j$ does not contain at least k occurrences of the symbol $P[j]$, having an unfeasible match, we proceed like we have a complete mismatch: we set k-$\mathsf{MS}[j].\mathsf{seq} = -$ and $\mathsf{len}_t[j] = \mathsf{len}_k[j] = 0$, resetting the k-MS computation in the next column. On the other hand, if we have $\mathsf{col}(\mathsf{PBWT})_j[s_j] = \cdots = \mathsf{col}(\mathsf{PBWT})_j[s_j + k - 1] = P[j]$ then we have a left-maximal match shared by at least k sequences. Hence, we can use the classical MS algorithm, setting: k-$\mathsf{MS}[j].\mathsf{seq} = k$-$\mathsf{MS}[j-1].\mathsf{seq}$, $s_{j+1} = \mathsf{FL}(s_j)$, $\mathsf{len}_t[j] = \mathsf{len}_t[j-1]+1$, and $\mathsf{len}_k[j] = \mathsf{len}_k[j-1]+1$. In less formal words, we follow the same sub-interval of PBWT j in column $j+1$ using the FL function, adding 1 to all the length values to consider the match. Note that we can check the condition $\mathsf{col}(\mathsf{PBWT})_j[s_j] = \cdots = \mathsf{col}(\mathsf{PBWT})_j[s_j + k - 1] = P[j]$ without scanning entirely $\mathsf{col}(\mathsf{PBWT})_j[s_j..s_j + k - 1]$, leveraging the property that this condition is satisfied iff $\mathsf{col}(\mathsf{PBWT})_j[s_j]$ and $\mathsf{col}(\mathsf{PBWT})_j[s_j + k - 1]$ lay in the same run. Given r as the number of runs in $\mathsf{col}(\mathsf{PBWT})_j$, we can make this check in $\mathcal{O}(\log r)$ time.

Finally, we need to consider the sub-cases we have if there are symbols within $\mathsf{col}(\mathsf{PBWT})_j[s_j..s_j + k - 1]$ that do not match $P[j]$. Also this check can be performed in $\mathcal{O}(\log r)$ time, for a PBWT column with r runs, by looking in which run the first and the last symbols of the interval lay: if $\mathsf{col}(\mathsf{PBWT})_j[s_j]$ and $\mathsf{col}(\mathsf{PBWT})_j[s_j + k - 1]$ lay on different runs it means that we have at least a run of a different symbol, *i.e.*, by run definition, at least the run we have in between those. To handle this

scenario, we need to identify the new support index s_{j+1}, necessary to mark a new k-length sub-interval. Being all the information stored just at run boundaries, we need to select a new run boundary b, such that $\mathsf{col}_j(\mathsf{PBWT})[b] = P[j]$, as in the mismatch case, assuming to consider i (being consistent with the explanation of the mismatch case) as the index of the first mismatch in $\mathsf{col}(\mathsf{PBWT})_j[s_j..s_j + k - 1]$. Then we can update $\mathsf{len}_t[j]$ computing the length of the common suffix up to the j-th column between k-$\mathsf{MS}[j-1].\mathsf{seq}$, which is the sequence that we are currently following by the FL mapping, and $\mathsf{PA}_j[b]$ In detail, we compare this common suffix length to $\mathsf{len}_t[j-1]+1$, selecting the minimum to retain only the suffix that encompasses at least k sequences in S. Formally, denoting $\mathsf{lcs}_j(A, B)$ as the longest common suffix up to the j-th column between two sequences A and B, we set $\mathsf{len}_t[j] = \min(\mathsf{lcs}_j(k\text{-}\mathsf{MS}[j-1].\mathsf{seq}, \mathsf{PA}_j[b]), \mathsf{len}_t[j-1]+1)$. Note that a ℓ-length $\mathsf{lcs}_j(A, B)$ can be computed using the LF mapping, *i.e.* the reverse mapping function, in $\mathcal{O}(\ell\hat{\rho})$ time, for a PBWT with a total of $\hat{\rho}$ runs. Finally, we update k-$\mathsf{MS}[j].\mathsf{seq}$ using the prefix array samples as in the MS algorithm, and we update s_{j+1} and $\mathsf{len}_k[j]$ leveraging the k-support values as: $\mathsf{len}_k[j] = L_b$ and $s_{j+1} = \mathsf{FL}(b) - \mathsf{off}_b$. We recall that in each case that requires jumping to a new sequence, if no run is available for the jump, for example when we have a PBWT column with only two runs, we fall back to a complete mismatch case.

Having computed $\mathsf{len}_t[j]$ and $\mathsf{len}_k[j]$ for a position j, we can finally compute the pseudo left-maximal matches length, assigning the value to k-$\mathsf{MS}[j].\mathsf{len}$ as: $k\text{-}\mathsf{MS}[j].\mathsf{len} = \min(\mathsf{len}_t[j], \mathsf{len}_k[j])$, for all $j = 1..w$.

After having entirely scanned P and computed the k-MS array, we can report the set k-SMEMs, similarly to the classical SMEM algorithm case described in Lemma 4. In fact, a k-SMEM of length k-$\mathsf{MS}[j].\mathsf{len}$ occurs between P and the sequence of the set S k-$\mathsf{MS}[j].\mathsf{seq}$, starting from position $j - k\text{-}\mathsf{MS}[j].\mathsf{len} + 1$ in P, if $k\text{-}\mathsf{MS}[j].\mathsf{len} \neq 0$ and either $j = w$ or $k\text{-}\mathsf{MS}[j].\mathsf{len} \geq k\text{-}\mathsf{MS}[j+1].\mathsf{len}$.

In terms of time complexity, we obtain the same asymptotical bounds as the μ-PBWT; however, increasing k leads to a corresponding increase in the number of reported k-SMEMs γ.

3.2.5 Solve the (k-)MPSC problem via μ-PBWT

In this subsection, we will show how to address the (k-)MPSC problem by using the (k-)MS array.

We start by leveraging the connection between (k-)MS and (k-)SMEMs, and the property that each positional substring belonging to a (k-)MPSC is contained within a (k-)SMEM. Moreover, for the non-inclusion property of (k-)SMEM, each starting position of a (k-)SMEM can fall in one and only one (k-)SMEM. Hence, we can use all these properties to prove that there exists a (k-)MPSC in which each substring is a prefix of a (k-)SMEM. In addition, given a pattern P, we can leverage another (k-)SMEM property: there exists only one (k-)SMEM that covers $P[w]$, *i.e.* the last symbol of the pattern. Therefore, we can include this ending (k-)SMEM as a positional substring in the (k-)MPSC. Then, we iterate the process of including each time the single (pseudo) left-maximal match, represented in the (k-)MS array in position j, that ends in the column j just before the starting column $j+1$ of the last positional substring added to the (k-)MPSC set. This (pseudo) left-maximal match is unique by definition of (k-)MS. The minimality condition of the (k-)MPSC set is guaranteed by the fact that no better solution exists, which should be, by absurdity, a single (pseudo) left-maximal match δ that covers two of the (pseudo) left-maximal matches, α and β, already present in the (k-)MPSC set. If such a δ exists, it must begin before α by the left-maximal property and extend at least up to the ending position of β at the j-th column, having that δ should be represented by the j-th value of the (k-)MS array. This is a contradiction for the (k-)MS uniqueness property, because in position j of the (k-)MS array, we already selected the (pseudo) left-maximal match β. We will soon see that this procedure computes not an arbitrary set of (k-)MPSCs but solves the leftmost (k-)MPSC.

Solving the leftmost, rightmost, and length-maximal MPSC problems We now demonstrate how to leverage MS and SMEMs to solve the leftmost, rightmost, and length-maximal MPSC problems in sublinear space. For simplicity, we present our approaches using $k = 1$, but they can be extended, without loss of generality, to $k > 1$ by adopting k-MS and k-SMEMs. To be consistent with the original definitions of these MPSC variants in [25, 90], we assume that a positional substring

Algorithm 3.1 Leftmost MPSC by MS.

```
1: function LeftMost(MS)
2:     j ← w                                  ▷ |MS| = w
3:     j' ← 0
4:     while j ≠ 0 do
5:         j' ← j − MS[j].len
6:         report (j' + 1, j, MS[j].seq)
7:         j ← j'
```

Algorithm 3.2 Rightmost MPSC by MS.

```
1: function RightMost(MS)
2:     i ← 1                                  ▷ |MS| = w
3:     for j = 1 → w − 1 do
4:         if MS[j].len ≥ MS[j + 1].len then
5:             report (i, j, MS[j].seq)
6:             i ← j + 1
7:     report (i, w, MS[w].seq)
```

covers each pattern position. Algorithms 3.1 and 3.2 implement our solutions, respectively, the leftmost, related to Lemma 5, and rightmost, related to Lemma 6, MPSC problems.

We discuss first the scenario of computing the leftmost MPSC.

Lemma 5 (Leftmost MPSC)**.** *Given a panel S and the* MS *array computed for a pattern P with respect to S, Algorithm 3.1 computes the leftmost* MPSC *C of P by S in time $\mathcal{O}(|C|)$ and $\mathcal{O}(\hat{\rho})$-space, with $\hat{\rho}$ as the total number of runs in the* PBWT*.*

To prove the correctness of Algorithm 3.1 and Lemma 5, note that we are computing an MPSC set C such that each i-th positional substring in C starts not after any other i-th positional substring in C. In other words, a leftmost MPSC set is composed of left-maximal positional substrings, being these positional substrings prefixes of SMEMs. This last property can be proved by the fact that, otherwise, we could extend the positional substring to the left, contradicting the assumption that it is in the leftmost MPSC set. Hence, we can start considering the SMEM that covers the last position of the pattern, including it in the set of the leftmost MPSC, as it is the only one that covers $P[w]$. The starting position j of this positional substring is $w - \mathsf{MS}[w].\mathsf{len} + 1$; therefore, we add $(j, w, \mathsf{MS}[w].\mathsf{seq})$ to the set C, which is now a leftmost MPSC of the positions from j to w. Then we proceed to scan the MS array from right to left. Note that a leftmost MPSC of columns 1 to $j - 1$ must include the left-maximal match that ends at $j - 1$; so, given $j' = (j - 1) - \mathsf{MS}[j - 1].\mathsf{len} + 1$, by the definition of SMEMs, it follows that we cannot have a SMEM that includes $\mathsf{MS}[j - 1].\mathsf{seq}$ in position $j - 1$ and starts before j'. Thus, j' is the starting position of the next $\mathsf{MS}[j-1].\mathsf{len}$-length positional substring C for the positions from j' to w. We iterate this procedure until we

consume P, resulting in C as the leftmost MPSC which covers all pattern positions. As shown by Algorithm 3.1, at each iteration, we add a positional substring to C, accessing the MS array in constant time. Hence Algorithm 3.1 runs in $\mathcal{O}(|C|)$ time.

We discuss now the scenario of computing the rightmost MPSC.

Lemma 6 (Rightmost MPSC)**.** *Given a panel S of w-length strings and the* MS *array of a pattern P with respect to S, Algorithm 3.2 computes the rightmost* MPSC *of P by S in time $\mathcal{O}(w)$ and $\mathcal{O}(\hat{\rho})$-space, with $\hat{\rho}$ as the total number of runs in the* PBWT*.*

To prove the correctness of Algorithm 3.2 and Lemma 6, note that we are computing an MPSC set C such that each i-th positional substring in C ends no earlier than any other i-th positional substring in C. Symmetrically to what we have for the leftmost MPSC, a rightmost MPSC set contains right-maximal positional substrings, being these positional substrings suffixes of SMEMs. We recall that, given the MS array, a position j marks the end of a SMEM iff $\mathsf{MS}[j].\mathsf{len} \geq \mathsf{MS}[j+1].\mathsf{len}$ or $j = w$. Without the loss of generality, we assume that $\mathsf{MS}[w+1].len = 0$. The construction of the rightmost MPSC C is symmetric to that of the leftmost MPSC, and, also in this case, we are not interested in overlaps between positional substrings. Hence, we can start considering the SMEM that covers the first position of the pattern, including it in the set of the rightmost MPSC, as it is the only one that covers $P[1]$. Scanning the MS array from left to right, we find the position j in which this SMEM ends, and we add the positional substring $(1, j, \mathsf{MS}[j].\mathsf{seq})$ to the set C. Then we continue to scan the pattern and to find all the positions js where a SMEM ends, *i.e.* where $\mathsf{MS}[j].\mathsf{len} \geq \mathsf{MS}[j+1].\mathsf{len}$. In these positions, there are no other right-maximal matches, meaning there cannot exist a SMEM containing $\mathsf{MS}[j].\mathsf{seq}$ at position j that extends beyond position j. Hence, we add the positional substring $(i+1, j, \mathsf{MS}[j].\mathsf{seq})$ to the rightmost MPSC set C, where i is the ending column of the last positional substring added to C.

As shown by Algorithm 3.2, we scan each MS array position, accessing the corresponding values in constant time, resulting in an algorithm that runs in $\mathcal{O}(w)$ time. Figure 3.8 shows how our algorithms compute the leftmost and the rightmost MPSC.

Finally, we consider the problem of finding a length-maximal MPSC. Sanaullah

S	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0
2	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0
3	1	0	1	0	1	0	0	1	0	0	0	1	1	0	0
4	1	1	1	0	1	1	1	1	0	0	1	0	0	0	0
5	0	1	0	1	0	0	0	0	1	1	0	1	0	0	1
6	1	0	1	0	1	1	1	1	0	0	1	1	0	0	0

MS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
seq	6	4	4	4	4	3	1	1	1	1	6	6	3	3	3
len	1	2	3	4	5	4	5	6	7	8	5	6	2	3	4

(a) Splitting a SMEM into positional substrings using the leftmost and rightmost MPSC. We illustrate a panel M and a pattern P together with the corresponding MS array. In the panel, dashed circles mark the leftmost MPSC and dotted circles mark the rightmost MPSC. SMEMs are highlighted with matching colors.

MS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
seq	6	4	4	4	4	3	1	1	1	1	6	6	3	3	3
len	1	2	3	4	5	4	5	6	7	8	5	6	2	3	4
j/j'	$\leftarrow$	2	$\leftarrow$	$\leftarrow$	$\leftarrow$	6	$\leftarrow$	$\leftarrow$	$\leftarrow$	$\leftarrow$	11	$\leftarrow$	$\leftarrow$	$\leftarrow$	15

(b) Computing leftmost MPSC (identified by circles in the P row) by the MS array. In the last line we show the jumps used to skip the overlaps as in Algorithm 3.1.

MS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
seq	6	4	4	4	4	3	1	1	1	1	6	6	3	3	3
len	1	2	3	4	5	4	5	6	7	8	5	6	2	3	4
i	1	–	–	–	–	6	–	–	–	–	11	–	13	–	–

(c) Computing rightmost MPSC (identified by circles in the P row) by the MS array. For the sake of simplicity, in the last line, we show the updating of the variable i in Algorithm 3.2.

Figure 3.8: Example in (a) of the relationship between leftmost/rightmost MPSC and SMEMs on the same set of sequences of [90]. In (b) and (c), we show how Algorithm 3.1 and Algorithm 3.2, respectively, work on the same example.

et al. [90] showed how to address in $\mathcal{O}(n + |\gamma|)$ space the length-maximal MPSC problem leveraging the leftmost MPSC, the rightmost MPSC, and the set γ of SMEMs, which have been computed in $\mathcal{O}(n)$ space, all combined in a solution which requires additional $\mathcal{O}(|\gamma|)$ space. Using the μ-PBWT we can compute the SMEMs, the leftmost MPSCs and the rightmost MPSCs in $\mathcal{O}(\hat{\rho})$-space, with $\hat{\rho}$ as the total number of runs in the PBWT. Therefore, the following lemma holds.

Lemma 7. *Given μ-PBWT for a set S of h strings of length w, a string P of length w, and the set γ of* SMEM*s shared by S and P, a length-maximal* MPSC *of P by S can be computed $\mathcal{O}(\hat{\rho} + |\gamma|)$ space.*

3.2.6 Phasing using μ-PBWT

We present a method for addressing the genotype phasing problem. The μ-PBWT framework consider a binary alphabet $\Sigma = \{0, 1\}$ while, as introduced in Section 3.1, genotype query is built over the alphabet $\Sigma = \{0, 1, 2\}$. Hence, we adapted the μ-PBWT to compute a variation of an MPSC that handles these kinds of queries. Note that, from a theoretical point of view, large reference panels are likely to include a haplotype pair that explains a genotype query. This last assumption could not be true, considering more compact real reference panels. We recall that a haplotype is a binary sequence/row in our input set/panel.

We designed a novel approach to reconstruct haplotype pairs that explain a given genotype by computing positional substrings (PSs) and complementary positional substrings (CPSs), which will be formally defined later, from a reference set S of bi-allelic haplotypes. This approach handles cases where recombinations are required to fully explain the genotype, aiming to scale to real-world data. At the same time, other old combinatorial state-of-the-art tools assume a perfect haplotype match [152, 151, 153]. On the other hand, currently used probabilistic state-of-the-art tools, such as Beagle [28, 29], are based on the Li and Stephens Hidden Markov Model [145], whereas μ-PBWT is deterministic in the absence of mutation and recombination.

We consider two primary scenarios: a) an ideal case where we do not need any recombination of the reference panel haplotypes to fully explain the input genotype, and b) a more realistic setting where we need to recombine haplotypes to explain the genotype query.

Positional substrings and complementary positional substrings In this context, we relax the problem of computing MPSC to compute an even non-minimal Positional Substring Cover of P by S, namely PSC. Moreover, to cover also heterozygous regions of the query, *i.e.* substrings of "2"s in the genotype query, we define the Complementary Positional Substring (CPS) as a pair of positional substrings (i, j, P) and (i, j, P'), which are bitwise complementary. For simplicity, a CPS will be identified by only the triple (i, j, P), being unambiguous to retrieve the bitwise complementary positional substring (i, j, P').

Formally, we define a genotype query Q as a w-long string over the alphabet $\Sigma = \{0, 1, 2\}$. We consider a binary set of sequences S composed of h sequences $S = \{S_1, \ldots, S_h\}$ of length w as a reference haplotype panel. Given a genotype Q, we denote PSC_g as a set C of positional substrings (i, j, Q) that occur in at least two sequences of S, *i.e.* a 2 positional substring (2PS), and a 2 positional complementary substring (2CPS) (k, l, S) of S such that: a) (k, l, Q) is a substring of Q of only "2"s, and b) C covers all positions of Q. We require at least two sequences due to the nature of the phasing problem. Less formally, a PSC_g consists of a coverage for the homozygous regions of Q, substrings of strings on $\Sigma = \{0, 1\}$, and of a coverage for all the heterozygous regions of Q, substrings of Q on $\Sigma = \{0, 1, 2\}$. Note that each 2PS and a 2CPS is associated with a subset of sequences of S.

The phasing problem is then reduced to cover Q with positional substrings (PSs) or complementary positional substrings (CPSs) derived from a panel of haplotypes S, *i.e.* covering it with a PSC_g. A natural optimization goal for this problem is to minimize the size of the PSC_g. However, to further reduce the number of recombinations and achieve greater consistency with the biological problem, we focus on minimizing the number of distinct pairs of haplotype sequences from S used to cover Q. From a computational point of view, we aim to minimize the number of 2PSs and CPSs composed of different haplotype pairs in S, addressing a computational problem known as the minimum recombination genotype-positional phasing problem. In detail, we say that there are no recombinations if all 2PSs and CPSs contain the same pair of haplotypes. We want to highlight that we could potentially have multiple pairs that fully explain a genotype without recombination.

Computing 2PSs via the μ-PBWT

We assume for simplicity that each PBWT column contains at least two "0"s and two "1"s, ensuring that a 2PS cover each homozygous position in Q with the reference haplotype sequences set S. In addition, to be able to handle recombinations in more possible recombinant columns, we add a constraint: each 2PS is of length at most two. We will explain the rationale behind this design choice later in this section.

To compute the 2PSs set C of non-overlapping 2PSs, we need to "reset" the computation of two matching statistics each time $\mathsf{MS}[j].\mathsf{len} \geq 2$, like the case where we have a complete mismatch in column j. In this way, each entry of the MS array contains pseudo left-maximal matches with a length of at most two, covered by at least two sequences in S. In this MS array, we can have positions that represent unmatched substrings of Q. In other words, whenever $Q[j] = 2$, we have a complete mismatch. Then, to guarantee the non-overlapping property, we apply the Algorithm 3.1 on this constrained MS array with another slight modification: we allow the presence of uncovered positions in this relaxation of the leftmost MPSC solution, *i.e.* all the heterozygous regions of Q remain uncovered.

Genotype phasing problem without recombinations via μ-PBWT We begin by discussing the simple case, specifically phase a genotype querying in the absence of recombinations. Assuming we have already computed the set C of 2PSs, we proceed to analyze, at each iteration, two consecutive 2PSs of C. We aim to identify possible haplotype pairs in S that can be used as two homozygous consecutive region anchors for phasing the heterozygous region of the query Q that lies between them. We aim to identify a CPS that spans a heterozygous region and shares at least a pair of sequences with the previous 2PS and the next 2PS, which serve as anchors. This CPS will be covered by at least a haplotype sequence pair, which also covers both anchors, having those sequences complement each other bitwise in the heterozygous region. Practically, given two consecutive 2PSs, we intersect the corresponding haplotype sets and we extract the submatrix between the two 2PSs, filtering out the sequences not in the intersection. Hence, with a naïve approach, we find all the complementary haplotype pairs that can explain

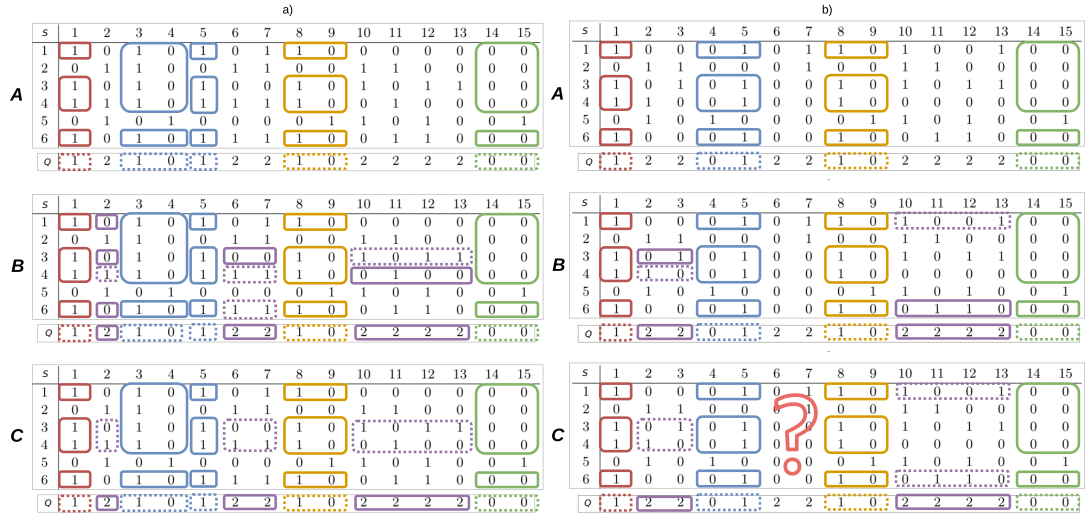


Figure 3.9: Example of addressing the phasing problem using 2PSs: (a) when two haplotypes fully explain the query genotype, *i.e.*, without mutations or recombinations; and (b) when no such pair exists, requiring different haplotype pairs across genotype regions, *i.e.* recombinations.

the heterozygous region of the genotype and the two anchors, *i.e.* the Q substring spanning from the starting position of the first 2PS to the last position of the second 2PS.

To avoid unnecessary computation, we consider only previously computed pairs, storing all possible pairs that explain Q up to the current position. This set is denoted S_{pair}. Hence, at each iteration, instead of considering two consecutive 2PSs in C, we filter the left anchor using S_{pair} before intersecting it with the right anchor. We recall that, under the assumption of no recombination, we always have at least one pair that explains each Q substring, *i.e.* both S_{pair} and the intersection are never empty. The filtering with S_{pair} is also applied in the case of two consecutive and adjacent 2PSs, when there is no heterozygous region in Q between them. At the end of each iteration, S_{pair} is updated with the current set of feasible pairs, which explain the current prefix of Q.

To show an example, we refer to Figure 3.9.a. In Figure 3.9.a.A, we show the 2PSs set C shared between a reference panel S and a genotype query Q. The filtering iterative approach is depicted in Figure 3.9.a.B. Initially, we have several pairs after intersecting the first two 2PS consistent with the heterozygous position

2: $S_{pair} = \{(1,4),(3,4),(4,6)\}$. Note that, for example, sequences 2 and 5 are excluded because they have a mismatch at position 1. Next, we have the situation of two consecutive 2PSs, without any heterozygous gaps between them. S_{pair} is used to filter the left anchor, and the result is intersected with the sequences represented by right 2PS, resulting in this case in the same S_{pair}, having the intersection with the right anchor equal to S_{pair} itself. Now we consider the fourth 2PS as the right anchor in positions 8 and 9. Note that sequence 1, in positions 6 to 7, which is a heterozygous region in the genotype query, does not complement any of the other sequences in $\{3,4,6\}$, which are the unique sequences in the current S_{pair}. After intersecting and complementing, the only feasible pair to explain the genotype up to position 9 is $S_{pair} = \{(3,4)\}$. This pair also explains the remaining part of the genotype, as shown in Figure 3.9.a.B. Finally, Figure 3.9.a.C shows the final selected haplotype pair.

Genotype phasing problem with recombinations via μ-PBWT In practical cases where sequencing errors occur or there are few reference haplotypes, the introduction of recombinations becomes necessary. Note that we can also have query mutations, *i.e.* errors in the substring (for example, having a "0" instead of a "1"), which create a small recombination event around the mutation site. We recall that a recombination is a change in the pair that explains different substrings of the genotype query. For example, haplotypes a and b explain the query Q from the beginning up to position i, while haplotypes c and d explain the query from position $i+1$ to the end. In this case, i is a recombination position/spot. We can have multiple recombination positions.

As in the zero-recombination case, we proceed to scan the set C of 2PSs from left to right, processing consecutive 2PSs and updating the current set of feasible pairs S_{pair}. In this scenario, without assuming the absence of recombination, we have different cases for handling two consecutive 2PSs. Recall that each adjacent 2PS is associated with a list of sequences.

The first case is that we get an empty intersection when considering the two anchors. In this context, we have two other possible scenarios: a) there exist two sequence pairs, one from the left anchor, filtered by S_{pair}, and one from the right anchor, that share at least one common haplotype but not both, or b) no such two

pairs exist. In the first scenario, we have one haplotype that recombines somewhere between the two 2PSs, whereas in the second scenario, both haplotypes recombine. In both cases, we need to store the temporary results up to the end of the first 2PS, and restart the iteration using all possible sequence pairs associated with the right anchor, using this list of pairs as the new S_{pair}.

In the second case, we have two consecutive 2PSs, always considered after the filtering step by S_{pair}, separated by a heterozygous region that any pair cannot explain. Again, we have two possible scenarios: a) a CPS covers the heterozygous region, but it is not consistent with any pair currently available from the previous iterations, and b) the two anchor 2PSs do not share sequence in S.

Both cases need to be thoroughly analyzed to determine compatible haplotype pairs before proceeding. Initially, we can attempt to reset S_{pair} by updating it with all the left anchor pairs, assuming a recombination. If we can explain the underlying heterozygous region with pairs shared by this new anchor and the one related to the right 2PS, we update S_{pair} with these pairs, also storing the previous S_{pair} in memory for "tracebacking". On the contrary, if such pairs do not exist or the CPS is not associated with a possible reference sequence pair, we skip that heterozygous region, restarting the iteration from the right 2PS and updating S_{pair} with all the possible pairs encoded in this right anchor. In this case, the heterozygous region remains unphased. Note that each time we need to reset and store the current S_{pair}, we also record the last position at which the stored S_{pair} was consistent with the target genotype. Hence, at each recombination spot, we store S_{pair} and the recombination position.

At the end of the genotype scanning, we have the final S_{pair}, which may partially cover the genotype. For each genotype region, we select the optimal haplotype pair from all possible pairs that can explain a specific area. The selection aims to minimize the number of recombinant haplotypes, *i.e.*, from a combinatorial perspective. For each region, we greedily select the haplotype pair that minimizes the total number of unique pairs that explain the input genotype. In detail, we sort all pairs by frequency, then we select the most frequent pair for each region. Hence, the chosen haplotype pair for each region is the one that also covers the most other Q substrings. If some pairs have the same frequency, the selection is weighted by how many times that pair was previously selected in different regions

to ensure consistency. This greedy approach is not optimal.

Figure 3.9.b illustrates a simple case of recombination. In Figure 3.9.b.A, considering the first two 2PSs, we obtain $S_{pair} = \{(3, 4)\}$. This pair cannot explain the genotype at positions 6 and 7, even after resetting the left anchor, because there are no CPS in that region; therefore, positions 6 and 7 remain unphased. Then, as shown in Figure 3.9.b.B, we restart from the next 2PS pair, updating $S_{pair} = \{(1, 6)\}$. This pair explains the last genotype region. We recall that we have stored the previous S_{pair}, *i.e.* $S_{pair} = \{(3, 4)\}$ and the corresponding recombination position 5. At the end of the iteration, having excluded all other possible pairs, we store $S_{pair} = \{(1, 6)\}$ and position 15. Analyzing all the stored S_{pair}s, we report a recombination event between $(3, 4)$ and $(1, 6)$, but we leave positions 6 and 7 unphased.

Advantages of 2PSs against 2-MPSCs As shown in Figure 3.10.a, 2-MPSC are less flexible in detecting anchors. In detail, we are unable to detect recombinations within the region covered by a 2-MPSC. As shown in the example of Figure 3.10, the two heterozygous regions cannot be explained by any sequence pair induced by the corresponding 2-MPSC anchors, despite the CPS covering those regions. The first and second 2-MPSC-associated pairs can not explain the heterozygous region from position 2 to position 3. The same happens restarting the iteration from the second 2-MPSC, thus updating the S_{pair} with all the possible pairs associated with this anchor, and considering the last 2-MPSC: positions 8 and 9 cannot be explained by any pairs coming from these anchors.

However, using 2PS, as in Figure 3.10.b, *i.e.* splitting the 2-MPSC which cover the region from 4 to 7 into two 2PS, we can select the pair $S_{pair} = \{1, 2\}$ which explains the genotype up to position 5, including the genotype query region from position 2 to position 3, and the pair $S_{pair} = \{2, 4\}$ that explains the genotype from position 6 to the end including the genotype query region from position 8 to position 9.

Time and space complexity of phasing via μ-PBWT

Computing 2PSs and MPSC share the same time complexity, but we need to consider that the number of sequences that share the same match differs. Since

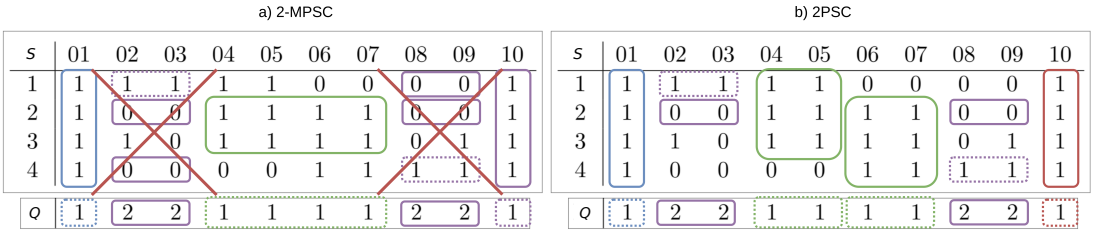


Figure 3.10: Example of the advantages of using 2PS and CPS compared to 2-MPSC. In (a), using the MPSC, it is not possible to phase the two heterozygous regions between sites 2–3 and 8–9, due to a recombination event within the MPSC at sites 5–6. In (b), using 2PS and CPS, these regions are phased correctly.

2PSs generally involve smaller matches, resulting in matches shared by more reference sequences in S. This leads to an increase in computational overhead.

Additionally, our approach is quadratic in time with respect to the number of possible pairs shared between the two anchors. Regarding space overhead, we need to store the input sequence set S as a set of sparse bitvectors to reconstruct submatrices used for the heterozygous regions. Note that these submatrices can be reconstructed using the Φ data structure, which would significantly increase execution time. Finally, we must consider that storing all 2PSs and the S_{pair}s is memory-intensive, with a worst-case total number of pairs quadratic in the number of haplotypes.

3.3 Results

The tested composite data structures for the RLPBWT and the μ-PBWT are implemented in C++ and compiled with the `-O3` flag. The phasing algorithm based on the μ-PBWT is also implemented in C++. Both the source code and the experimental pipeline are available at: a) `https://github.com/dlcgold/rlpbwt.git` for [21], b) `https://github.com/dlcgold/muPBWT.git` for [23], c) `https://github.com/dlcgold/muPBWT/tree/k-smem` and `https://github.com/dlcgold/muPBWT/tree/k-smem-live` for [24], and d) `https://github.com/dlcgold/muPBWT/tree/phase` for [27]. All the experimental Snakemake [164] pipelines on publicly available data are included in the corresponding repository, except for the results presented in [23], which are available at `https://github.com/dlcgold/`

`muPBWT-1KGP-workflow.git`.

All our tools rely on the Succinct Data Structure Library (`sdsl-lite`) [43], publicly available at `https://github.com/simongog/sdsl-lite.git`, for succinct data structures implementations such as `int_vectors` and `sd_vectors` with `rank` and `select` support, and on `htslib` [165], publicly available at `https://github.com/samtools/htslib.git`, for VCF/BCF file reading.

We conducted our experiments on computing the SMEMs with RLPBWT and μ-PBWT on a machine equipped with an Intel Xeon CPU E5-2640 v4 (2.40GHz), 756GB of RAM, and 756GB of swap space, running Ubuntu 20.04.4 LTS 64-bit. On the other hand, k-SMEMs, MPSC and phasing results are obtained on a machine equipped with an Intel Xeon CPU E5-4610 v2 (2.30GHz), 256GB of RAM, 8GB of swap, and Ubuntu 20.04.6 LTS 64-bit. The principal reason for this double-server setup is the comparison with the original PBWT for SMEMs computation, which requires more than 256GB of RAM on some tested haplotype panels.

For RLPBWT, other co-authors tested some simulated data on an Intel Core i3-9100 CPU (3.60GHz) with 128 GB RAM, running Debian 11; C source code available at `https://github.com/koeppl/pbwt`.

In all our experiments, we used the Unix `/usr/bin/time --verbose` utility to get execution time and memory peaks.

3.3.1 Experimental datasets

We evaluated both simulated and real data in our experiments. Consider that not all datasets are used in all experiments, according to which data are considered valid or not for testing in different scenarios.

Simulated datasets To simulate data, we employed the same approach as in [13], running the Markovian coalescent simulator `MaCS` [166] with command-line parameters `100000 2e7 -t 0.001 -r 0.001`, to generate a haplotype matrix with 100 000 individuals and 360 000 sites. To scale our experiments, we subsample this dataset with a parameter ξ such that, given a column of length h having o ones, we skip this column if $o/h < \xi$, setting ξ to be equal to 0.01, 0.03, 0.05, 0.08, and 0.10, varying the panel degree of repetitiveness.

Table 3.2: Statistics for the simulated `MaCS` dataset.

	Sample Parameter ξ				
Statistic	0.01	0.03	0.05	0.08	0.10
Sequences/haplotypes	100 000	100 000	100 000	100 000	100 000
Columns/sites	93 000	71 000	60 000	51 000	47 000
PBWT runs	2 400 000	2 300 000	2 300 000	2 300 000	2 300 000

Table 3.3: Statistics for the `msprime` simulated dataset.

Panel ID	Rows/haplotypes	Columns/sites	PBWT runs
1	20 000	209 531	9
2	200 000	743 171	13
3	200 000	2 271 035	4
4	500 000	2 271 035	8
5	2 000 000	2 271 035	18

We also simulated a 10 megabase region of European samples using `msprime` [167]. These bi-allelic panels have an increasing number of haplotypes up to 2 million (10 000, 100 000, and 1 000 000 samples, namely panels 1,2, and 3). We also sub-sampled the panel with 1 000 000 individuals to obtain two additional panels with 100 000 and 250 000 samples, namely panels 4 and 5. In Table 3.3, we present statistics for these panels.

1000 Genomes Project data The first real data dataset considered is phase 3 of the 1000 Genomes Project (1KGP) [4, 5], one of the first multi-sample whole-genome sequencing projects, which now counts over 2000 samples. All the VCF files are publicly available at `https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/`.

To consider just bi-allelic sites, those panels are filtered using bcftools [130], via the command `bcftools view -m2 -M2 -v snps`. In some experiments, we considered all the autosome panels, while in others, we used a subset of autosomes. An autosome is a chromosome that is not a sex chromosome, *i.e.* we exclude the chromosome X and the chromosome Y. All these panels are constructed using whole-genome sequencing data from 2504 samples, resulting in 5008 haplotypes/rows. Table 3.4 summarizes the statistics of these panels.

Table 3.4: 1000 Genome Project panels statistics. All panels have 5008 sequences/haplotypes.

Chr	Columns/sites	Avg. PBWT runs	Chr	Columns/sites	Avg. PBWT runs
1	6 196 151	11	12	3 698 099	10
2	6 786 300	10	13	2 727 881	10
3	5 584 397	10	14	2 539 149	11
4	5 480 936	10	15	2 320 474	12
5	5 037 955	9	16	2 596 072	12
6	4 800 101	10	17	2 227 080	12
7	4 517 734	10	18	2 171 378	11
8	4 417 368	10	19	1 751 878	13
9	3 414 848	11	20	1 739 315	11
10	3 823 786	10	21	1 054 447	14
11	3 877 543	10	22	1 055 454	14

Note that these panels are very sparse, having fewer "1"s compared to "0"s, approximately the "1"s are the 0.03% of the total. This sparsity is confirmed by the average number of runs per column in the PBWT.

UK Biobank data Although not publicly available, we request access to the UK Biobank data to evaluate our data structures using UK Biobank SNP array panels of all autosomes and high-coverage whole-genome sequencing data on chromosome 20 [6]. For SNP array data, we applied the standard QC as suggested by the authors, phasing the whole dataset using SHAPEIT4 [168], and obtaining a total of 976 754 haplotypes/rows and 670 741 SNPs/columns across all the autosomes. Table 3.5 summarizes the sizes of these panels.

Regarding the whole-genome sequencing panel for chromosome 20, available on the UK Biobank research analysis platform [26], we used the most recent version, phased with SHAPEIT5 [148], resulting in a 300 238 haplotypes/rows and 13 780 193 bi-allelic sites/columns panel. Using the UK Biobank research analysis platform, we applied our method to 13 regions of this panel, which have at least 4 megabases and 4 centimorgans. Table 3.6 summarizes the sizes of these panels.

Genotype phasing dataset For the genotype phasing experiments, we used a combination of real and simulated data. As a reference panel, we considered the HGSVC2 human bi-allelic panel [169, 170] for chromosome 2, with 68 hap-

Table 3.5: UK Biobank SNP array data statistics. All panels have 976 754 sequences/haplotypes.

Chr	Columns/sites
1	54 432
2	53 433
3	44 935
4	41 678
5	40 020
6	46 515
7	36 682
8	34 141
9	29 581
10	33 086
11	33 827
12	32 011
13	22 344
14	21 708
15	21 286
16	24 314
17	22 856
18	19 432
19	19 845
20	17 515
21	9940
22	11 160

Table 3.6: UK Biobank whole-genome sequencing regions for chromosome 20 statistics. All regions have 150 119 sequences/haplotypes.

Region	Columns/sites
chr20:60061-4060065	865 267
chr20:4060066-8060066	880 899
chr20:8060067-12515479	961 591
chr20:12515480-16768988	917 468
chr20:16768989-21050967	931 010
chr20:21050968-31549151	1 919 134
chr20:31549152-38282825	1 436 549
chr20:38282826-43181963	1 056 144
chr20:43181964-47619489	955 970
chr20:47619490-51789198	923 178
chr20:51789199-55789212	911 452
chr20:55789213-59874964	925 442
chr20:59874965-64334101	1 096 089
Total	13 780 193

Table 3.7: Total number of heterozygous variation sites as the mutation rate varies in the simulated dataset for the genotype phasing experiments.

Mutation rate	Heterozygous sites
0 %	60 604
1 %	60 862
3 %	61 362
5 %	61 995
10 %	63 153
20 %	65 265

lotypes/rows and 229 300 columns/sites. As a simulated target, we randomly selected a sample from this reference panel, and combined its haplotypes into an unphased genotype target. This genotype has 60 604 heterozygous sites, which is approximately 26% of the total number of variants on chromosome 2.

Moreover, we mutate the target genotype by replacing the allele at each position with probability $\varepsilon \in \{1\%, 3\%, 5\%, 10\%, 20\%\}$. We refer to ε as the mutation rate. Table 3.7 summarizes the total of heterozygous sites after this random mutation step. Note that these mutations cause small spurious recombinations at nearby sites, which are consistent with a more realistic scenario.

In our input VCF/BCF files, "0|0" and "1|1" represent phased homozygous sites, "0|1" and "1|0" represent phased heterozygous sites, "0/0" and "1/1" represent unphased homozygous sites, and "0/1" represent unphased heterozygous sites.

3.3.2 Comparison tools

To evaluate our approaches, we compared them against state-of-the-art tools according to the experimental scenario.

Durbin's PBWT We compare our approaches to the original PBWT implementation for building the index and computing SMEMs. The C source code is available at `https://github.com/richarddurbin/pbwt`. In detail, we have three methods to compute SMEMs with an external query: a) the `matchNaive` algorithm, not based on the PBWT and used just for benchmarking, the `matchIndexed` algorithm, also referred to as `PBWT-index`, which is the implementation of Durbin's algorithm 5 (*i.e.* Algorithm 2.5 in Section 2.7), and c) the `matchDynamic` algorithm, also

referred as `PBWT-dynamic`. The latter requires a further explanation. This algorithm, whose results are briefly discussed in [13] as the `batch algorithm`, involves directly merging all external haplotypes into the reference panel and computing internal SMEMs. This approach is very space-efficient because we can compute the SMEMs by loading the information of just two consecutive columns into RAM, resulting in a $\mathcal{O}(h)$ space algorithm. We have two main cons: a) we need to rebuild the PBWT index each time we have a new set of queries, having to re-index the input panel, and b) the SMEMs are reported in the order induced by the PAs and not by the order of the queries. Finally, this algorithm is considered difficult to extend for solving other PBWT problems, which is why most approaches in the literature are merely extensions of the original Algorithm 5 to address different matching tasks. As input, this implementation accepts both `MaCS` and VCF/BCF files, using the `htslib` library for the latter.

Syllable-PBWT Implemented in C++ and publicly available at `https://github.com/ZhiGroup/Syllable-PBWT.git`, the Syllable-PBWT is used in our experiments as a compressed PBWT comparison. Being developed to compute ℓ long matches instead of SMEMs, we excluded the querying comparisons. Note that ℓ long matches are a superset of SMEMs. As input, this implementation accepts only VCF files, and not BCF files, implementing a manual input reading instead of `htslib`. This design choice prevents experiments with larger panels, which require on disk a few gigabytes as BCF but hundreds of gigabytes as VCF.

BGT Developed in C to compress, store, and query VCF/BCF files, BGT [171], available at `https://github.com/lh3/bgt.git`, has been considered for the index disk footprint size comparison. Due to its inability to handle external queries, BGT is excluded from all experiments involving SMEMs computation. In other words, BGT is not designed for pattern matching but instead emphasizes fast random access. As input, this implementation accepts VCF/BCF files, using the `htslib` library.

k-MPSC tool Developed in C++ and available at `http://genome.cshlp.org/lookup/suppl/doi:10.1101/gr.277673.123/-/DC1`, the k-MPSC implementation

by Sanaullah et al. [25, 90] is used as a comparison tool for MPSC and k-MPSC computation. Note that the code required a few modifications, as it was unable to index 1KGP-size panels due to design choices in the dynamics allocations. Moreover, the execution halts if an unfeasible column is encountered, *i.e.* we do not have enough occurrences of the current pattern symbol in that PBWT column. We edited the source code, resolving the allocation issue and enabling the computation to restart after encountering an unfeasible column. As Syllable-PBWT, this implementation can read only VCF files.

Beagle Beagle [28, 29] is a state-of-the-art tool for genotype phasing based on the use of a haplotype reference panel with a Li-Stephens HMM model. It is developed in Java and it is publicly available at `https://faculty.washington.edu/browning/beagle/beagle.html`. Despite not relying on the `htslib` library, Beagle can read gzipped [172, 173] VCF files but not BCF files.

3.3.3 Computing SMEMs with the RLPBWT and the μ-PBWT

In this subsection, we will discuss experimental results on computing SMEMs using composite data structures for the RLPBWT, and compare the best-performing ones with the μ-PBWT.

RLPBWT memory benchmark on `MaCS` simulated dataset From an implementation point of view, for the various RLPBWT data structures on the `MaCS` simulated dataset, we considered: a) the mapping structure implemented using sparse bitvectors, b) the Δ-encoded divergence array and the Cartesian tree implemented with grammar compression, c) the forward and backward data structure implemented by building the PBWT in both the forward and backward directions, d) the sampled prefix arrays at run boundaries, and e) the LCE data structure and the panel random access implemented by an SLP. Table 3.8 shows the sizes of the single data structures used to implement the RLPBWT.

I recall that I was not directly involved in the RLPBWT *implementation based on the* RLPBWT *variants based on the (Δ-encoded) divergence array, on the Forward-Backwards, and on the Cartesian tree.*

Table 3.8: Estimated bits size of the single data structures and of the combinations of them for the RLPBWT. In boldface, the best performance. Forward-Backwards (FWBW) are not listed as just an additional Mapping Structure or Cartesian Tree in memory.

	Sample Parameter ξ				
Component	0.01	0.03	0.05	0.08	0.10
Mapping structure (MAP)	57MB	53MB	52MB	51MB	51MB
Δ-encoded divergence array (DEDA)	479MB	452MB	435MB	426MB	418MB
Cartesian tree (CT)	472MB	472MB	458MB	420MB	402MB
Longest common extension (LCE)	96MB	88MB	88MB	80MB	80MB
Sampled column permutations (PERM)	80MB	76MB	76MB	76MB	76MB
Divergence array (DA)	125GB	92BG	77GB	64GB	58GB
SLP Random access (RA)	96MB	88MB	88MB	80MB	80MB
Data Structure					
MAP + LCE + PERM	**233MB**	**217MB**	**216MB**	**207MB**	**207MB**
MAP + DEDA	536MB	505MB	487MB	477MB	469MB
MAP + CT + FWBW	1.1GB	1.1GB	1.0GB	942MB	906MB
MAP + CT + DA	126GB	93GB	78GB	64GB	58GB
MAP + CT + PERM + RA	705MB	689MB	674MB	627MB	609MB

Across all the combinations, in this experimental scenario, MAP+LCE+PERM is the most space efficient, followed by MAP+DEDA. We highlight that the space performance of MAP+DEDA is surprising, being inspired by Gagie et al. [82], which was not implemented as a theoretically space inefficient data structure. Increasing the ξ value results in a decrease in the size of all the data structures. This behavior can be explained by the column selection based on the ξ value.

μ-PBWT benchmark on msprime simulated dataset We compared μ-PBWT against Syllable-PBWT on the msprime simulated dataset. Syllable-PBWT can read and index only the two smaller panels, *i.e.* panel 1 and panel 2, and cannot read BCF files. For these two panels, as shown in Table 3.9 and in Figure 3.11.b, Syllable-PBWT index on disk takes up to approximately 25 times more space than the index produced by the μ-PBWT.

For building the index, μ-PBWT requires about 16 times less memory but about twice as much time as Syllable-PBWT.

Note that, considering the estimated $13n$ bytes required for the original PBWT algorithm 5, μ-PBWT reduces the memory consumption by up to 10 000 times on

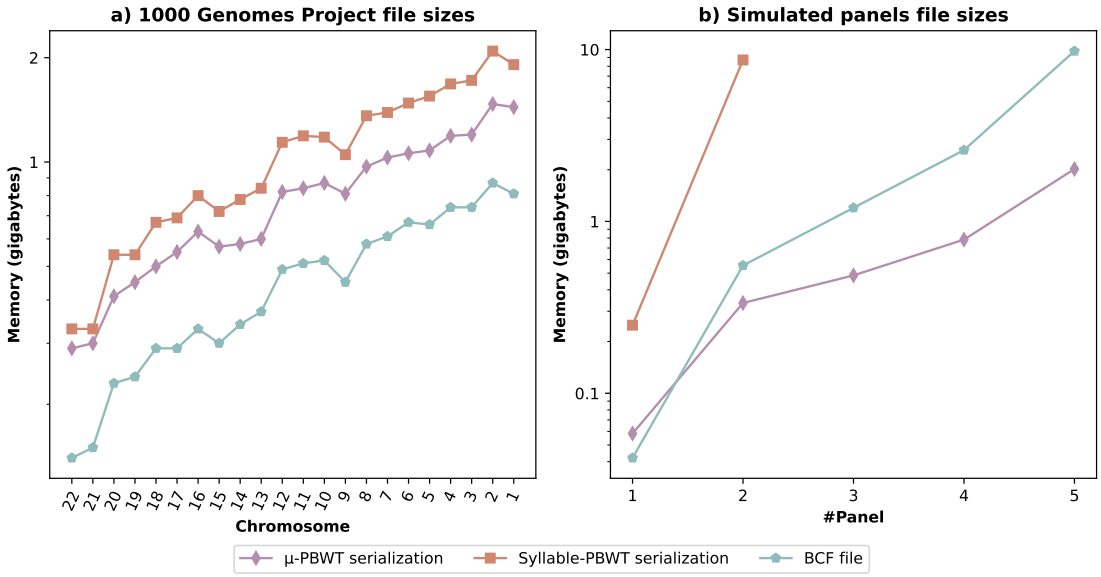


Figure 3.11: BCF, Syllable-PBWT, and μ-PBWT disk usage comparison on a) 1000 genome Project data and b) `msprime` simulated panels.

Table 3.9: Simulated panels statistics and size on disk of the input BCF file, the μ-PBWT index file and the Syllable-PBWT index file.

Panel	Rows	Columns	BCF (GB)	μ-PBWT (GB)	Syllable-PBWT (GB)
1	20 000	209 531	0.04	0.06	0.25
2	200 000	743 171	0.55	0.34	8.70
3	200 000	2 271 035	1.2	0.49	-
4	500 000	2 271 035	2.6	0.78	-
5	2 000 000	2 271 035	9.8	2.02	-

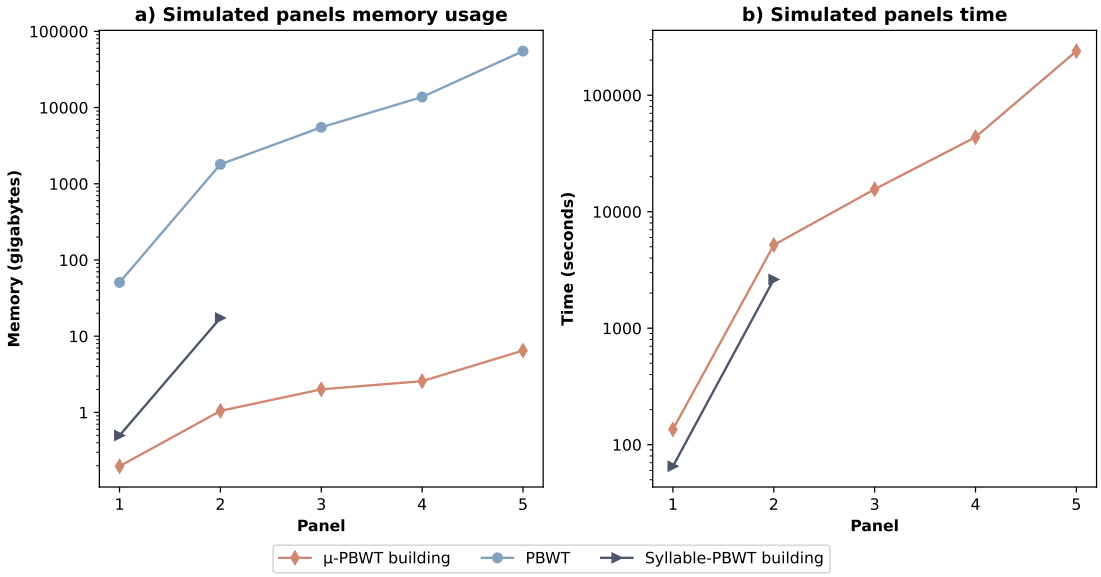


Figure 3.12: Index building results comparison of μ-PBWT, and Syllable-PBWT on `msprime` simulated panels: (a) memory usage, and (b) running time. In (a), results regarding the original implementation of the PBWT are estimated and plotted as baseline.

these simulated panels. This behavior is strongly correlated with the number of runs, as reported in Table 3.3, since μ-PBWT scales approximately linearly in both space and time with respect to the number of runs.

Regarding the μ-PBWT internal data structures, as shown in Figure 3.16.b, the Φ data structure takes most of the memory of the μ-PBWT, as expected, storing two sparse bitvectors and two bit-compressed intvectors for each haplotype.

Finally, we want to highlight that the each μ-PBWT produced in these simulated experiments is loaded in less than 30 seconds on a commodity laptop (equipped with an AMD Ryzen7 3700U and 16GB RAM). These results prove that we can drastically reduce the hardware requirements needed for sharing and analyzing whole-genome sequencing data.

μ-PBWT benchmark on 1000 Genomes Project data We demonstrate the performance of μ-PBWT by comparing it on all the autosomes from the 1KGP data against Durbin's PBWT Algorithm 5 (*i.e.* the `matchIndexed` algorithm in the implementation) and the Syllable-PBWT. In detail, we evaluate the μ-PBWT and

the original PBWT considering: a) the memory usage peak, and b) the execution time required for indexing and SMEMs finding. In addition, as for the experiments with the `msprime` simulated dataset, we compared the μ-PBWT against the Syllable-PBWT just for the indexing step, benchmarking both indexing space and time.

To evaluate SMEMs computing, we extracted 100, 500, and 1000 haplotypes from each input panel (reducing the input panel to 4908, 4508, and 4008 haplotypes), which we used as queries/external haplotypes. Consider that, for 100 queries, the number of SMEMs ranges from about 1000 to about 2500.

Figure 3.13.a shows indexing memory peak of μ-PBWT and Syllable-PBWT while Figure 3.13.b shows the indexing execution time. These results show that Syllable-PBWT performs slightly better in indexing than μ-PBWT, using approximately half the memory and computation time across all tested panels. On the contrary, the original PBWT index is mostly built at query time in the implementation, so it has been excluded in this plot. In Figure 3.13.b, we also included BGT, which does not appear in Figure 3.13.a due to the logarithmic scale of the plot. In fact, BGT uses a negligible amount of memory. Note that BGT is up to two times faster than the Syllable-PBWT and four times faster than the μ-PBWT, but we need to take into account that it is a lighter index, which does not support external queries.

Figure 3.14.a shows the peak memory usage during SMEM finding for both μ-PBWT and Durbin's PBWT, while Figure 3.14.b reports the corresponding query execution times. For μ-PBWT, since the entire set of queries is loaded into memory to guarantee the output order in the multithreads scenario, RAM usage increases with the number of queries. To make a fair comparison against Durbin's PBWT, since it builds most of its indices at query time, we need to sum μ-PBWT building and querying time and compare the result to PBWT running time. According to our experiments, μ-PBWT is up to six times slower than the original PBWT Algorithm 5, using 1000 queries; however, as the number of queries decreases, the execution time gap narrows, with μ-PBWT requiring only about twice as much time. Despite being slower, the μ-PBWT is significantly lighter in memory, requiring up to 80 times less memory than Durbin's PBWT.

By comparing each query independently, we can run μ-PBWT using multiple threads. In Figure 3.15, we show that using 12 threads, μ-PBWT requires a non-

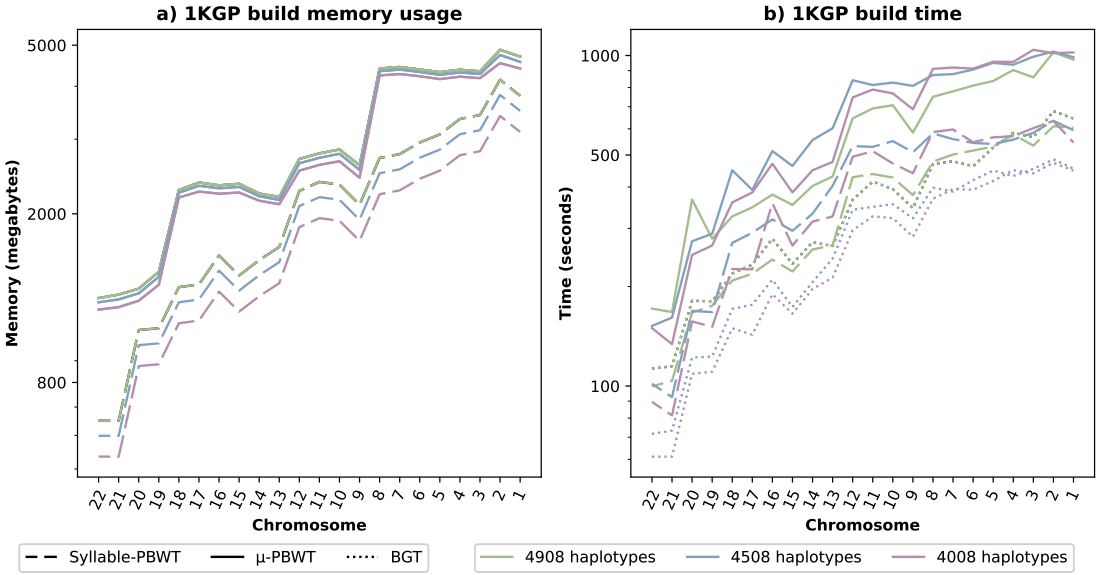


Figure 3.13: Indexing results comparison on 1000 Genomes Project data for μ-PBWT, Syllable-PBWT, and BGT: (a) maximum memory usage, and (b) execution time.

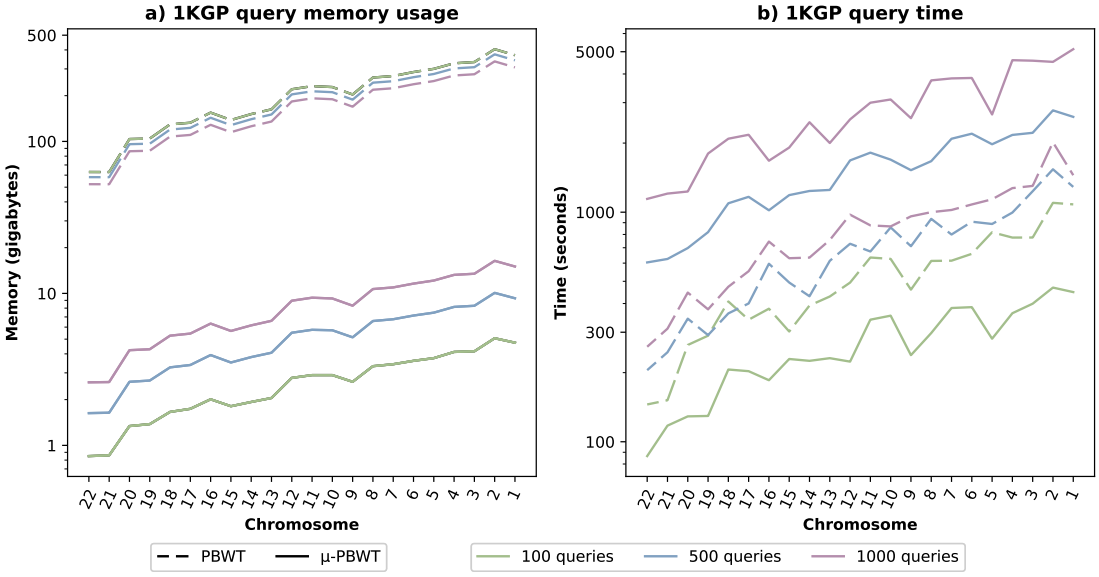


Figure 3.14: Querying results comparison on 1000 Genomes Project data with 100, 500, and 1000 queries for μ-PBWT and Durbin's PBWT: (a) maximum memory usage, and (b) execution time.

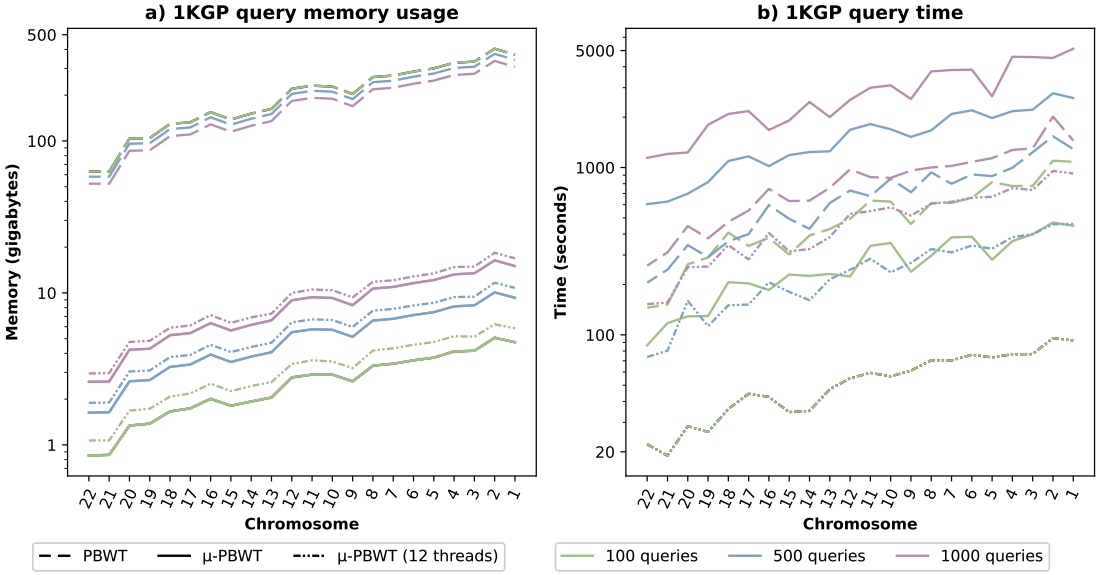


Figure 3.15: Querying results comparison on 1000 Genomes Project data with 100, 500, and 1000 queries for μ-PBWT with 12 threads and Durbin's PBWT: (a) maximum memory usage, and (b) execution time

negligible amount of additional memory, having multiple matching statistics arrays to compute and keep in memory at the same time, but computes SMEMs 4 times faster than the monothread execution.

Next, we analyzed the memory usage breakdown of μ-PBWT data structures. As for the `msprime` simulated dataset, the Φ data structure is the component that requires the most memory, accounting for approximately 40% of the total. Then we have PA/DA samples, which require approximately 30% of the memory, mapping structures, which require approximately 20%, and finally thresholds, which require approximately 10%. Figure 3.16.a shows these results.

Finally, as shown in Table 3.10 for the panel with 4908 haplotypes, the μ-PBWT requires only twice the disk usage compared to the input BCF file and use 25% less disk space than the Syllable-PBWT. In Figure 3.17, which considers all the panels we get after extracting 100, 500, and 1000 queries, we plot the indices disk usage, including BGT, and using the input BCF disk size as baseline.

Table 3.10: 1000 Genome Project panels statistics and size on disk of the input BCF file, the μ-PBWT index file and the Syllable-PBWT index file. Each panel has 4908 haplotypes. We also reported the average number of runs per column.

Chr	Sites	Avg. Runs	BCF (GB)	μ-PBWT (GB)	Syllable-PBWT (GB)
1	6 196 151	11	0.78	1.44	1.91
2	6 786 300	10	0.84	1.47	2.09
3	5 584 397	10	0.71	1.20	1.72
4	5 480 936	10	0.71	1.19	1.69
5	5 037 955	9	0.63	1.08	1.55
6	4 800 101	10	0.64	1.06	1.48
7	4 517 734	10	0.58	1.03	1.39
8	4 417 368	10	0.56	0.97	1.36
9	3 414 848	11	0.43	0.81	1.05
10	3 823 786	10	0.50	0.87	1.18
11	3 877 543	10	0.49	0.84	1.19
12	3 698 099	10	0.47	0.82	1.14
13	2 727 881	10	0.35	0.60	0.84
14	2 539 149	11	0.32	0.58	0.78
15	2 320 474	12	0.29	0.57	0.72
16	2 596 072	12	0.32	0.63	0.80
17	2 227 080	12	0.28	0.55	0.69
18	2 171 378	11	0.28	0.51	0.67
19	1 751 878	13	0.23	0.45	0.54
20	1 739 315	11	0.22	0.41	0.54
21	1 054 447	14	0.14	0.30	0.33
22	1 055 454	14	0.14	0.29	0.33

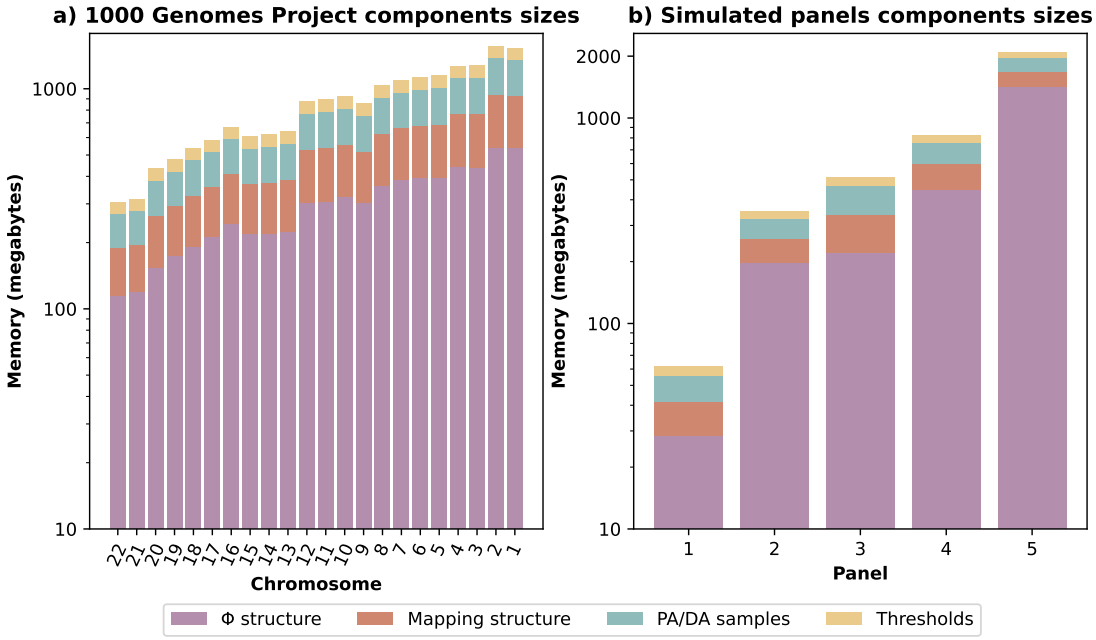


Figure 3.16: μ-PBWT components sizes breakdown on: (a) the 1000 Genomes Project data, and (b) the `msprime` simulated panels.

RLPBWT and μ-PBWT benchmark on 1KGP data For this experiment, we evaluated the best theoretical RLPBWT composite data structure, `MAP+LCE+PERM`, against the μ-PBWT, the Durbin's PBWT Algorithm 5, *i.e.* the `matchIndexed` algorithm (`PBWT-index` in the plots), and the Durbin's PBWT batch algorithm, *i.e.* the `matchDynamic` algorithm (`PBWT-dynamic` in the plots). We considered a subset of the 1KGP data, selecting the bi-allelic panels for Chromosomes 22, 20, 18, 16, and 1. From these panels, we extracted 100 queries, resulting in reference panels with 4908 haplotypes.

Figure 3.18 summarizes the indexing performances. Note that we reported just one PBWT result, being the construction for `PBWT-index` and `PBWT-dynamic` identical (and partial, not building the entire index for Algorithm 2.5 as already pointed out). The figure shows that μ-PBWT requires more memory than the original PBWT but less memory than `MAP+LCE+PERM`. Regarding execution time, there is a negligible difference between the performance of `MAP+LCE+PERM` and μ-PBWT. Overall, both our methods are outperformed by Durbin's PBWT in indexing the input panel.

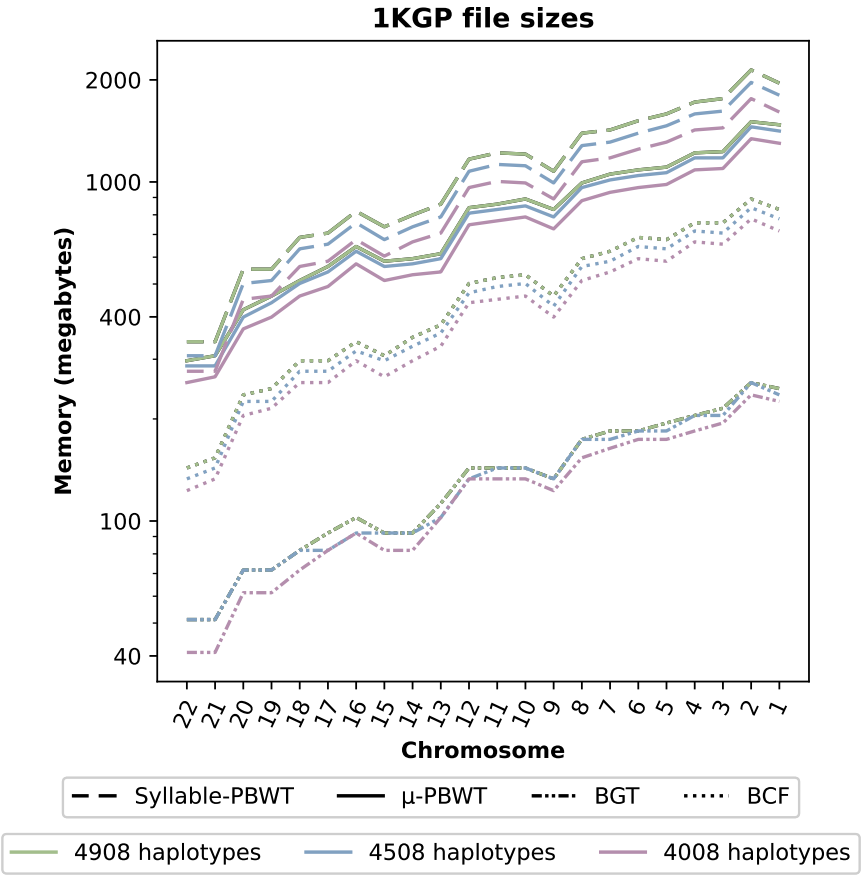


Figure 3.17: 1000 Genome Project panel sizes on disk of the input BCF file, the μ-PBWT index file, the Syllable-PBWT index file, and the BGT index file. For this plot, we considered the three panels after the queries extraction, *i.e.* having 908, 4508, and 4008 haplotypes.

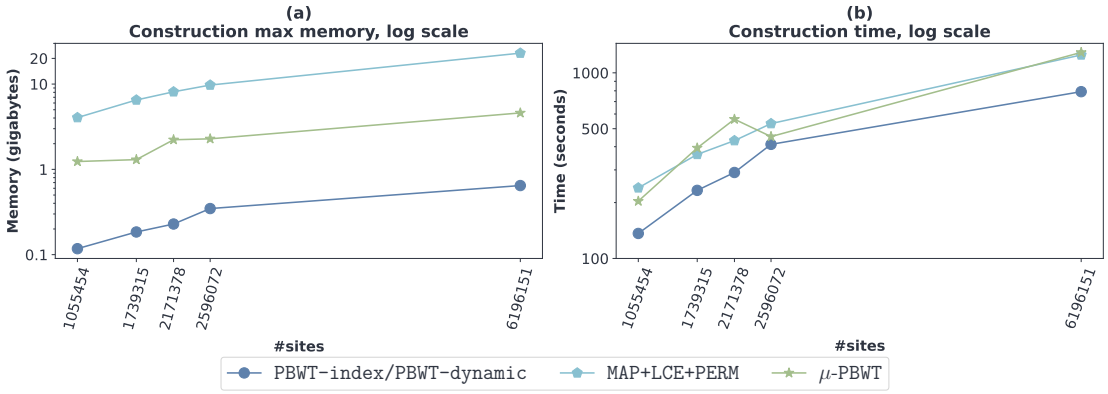


Figure 3.18: Indexing results comparison on 1000 Genomes Project data for MAP+LCE+PERM, μ-PBWT, and Durbin's PBWT: (a) maximum memory usage, and (b) execution time. On the x-axis, we show the number of sites in the panels corresponding to chromosomes 22, 20, 18, 16, and 1.

Figure 3.19 summarizes the querying performances. Note that Figure 3.19.c shows the mean execution time required when each query is given as an individual query. PBWT-dynamic is the fastest when the queries are given at once, but slowest when the queries are given one at a time. This is an expected behavior, having this algorithm to rebuild the entire PBWT for each query. On the other hand, PBWT-indexed required more memory than all the other competing methods, more than 20 times more than our MAP+LCE+PERM, as expected since it is the direct implementation of Durbin's Algorithm 5. The PBWT-indexed approach is the second slowest in the classical scenario of multiple queries, but it is the fastest when queries are given individually. Note that this method does not rebuild the PBWT each time. Moreover, we see that MAP+LCE+PERM is at most ten times slower than PBWT-MatchIndexed when queries are given individually. On contrary, in the classical scenario, MAP+LCE+PERM is approximately two times slower than PBWT-MatchIndexed, but it is almost ten times faster than PBWT-MatchDynamic when the queries are all at once. Compared to the μ-PBWT, MAP+LCE+PERM used slightly more memory and takes a few more time than μ-PBWT. We want to highlight that MAP+LCE+PERM computes the matching statistics in just one pass while μ-PBWT requires two passes. Theoretically, this difference makes MAP+LCE+PERM more suitable for an online setting when the SMEMs can be found as the input is

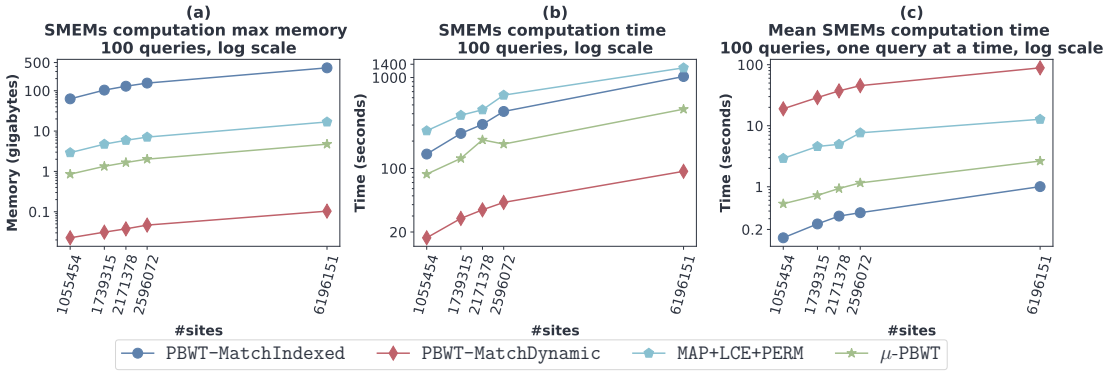


Figure 3.19: Querying results comparison on 1000 Genomes Project data for `MAP+LCE+PERM`, μ-PBWT, Durbin's `PBWT-index`, and Durbin's `PBWT-dynamic`: (a) maximum memory usage, (b) execution time, and (c) average execution time per query. In (c), we omit the standard deviation bars because they are too small. On the x-axis, we show the number of sites in the panels corresponding to chromosomes 22, 20, 18, 16, and 1.

read in.

μ-PBWT benchmark on UK Biobank data Table 3.12 summarizes UK Biobank SNP array data results across all autosomes. In both simulated and 1KGP experiments, the μ-PBWT index requires approximately three times as much disk space as the input BCF file; hence, the high number of haplotypes and the low sparsity of these SNP array data cause this behavior. For example, we have on average 13 462 runs per column in the chromosome 20 panel. Regarding execution time, indexing all the autosomal panels in parallel takes less than 2 hours.

Taking into account the UK Biobank high-coverage whole genome sequencing data on chromosome 20, the μ-PBWT can index the entire panel in 13GB of disk space, *i.e.* almost three times less than the input BCF file. Full results for this panel, obtained by indexing in parallel 13 regions of this panel, are available in Table 3.11.

3.3.4 Computing k-SMEMs and (k-)MPSC using the μ-PBWT

We extended the μ-PBWT for computing k-MS, k-SMEMs and (k-)MPSC with pre-computed k-support values (`https://github.com/dlcgold/muPBWT/tree/k-smem`)

Table 3.11: Indexing results for the UK Biobank high-coverage whole genome sequencing data on chromosome 20 information. In the last row, we report the total results, except for execution time, because each region is indexed in parallel. Each panel has 150 119 haplotypes.

Region	Sites	BCF (GB)	μ-PBWT (GB)	Time (s)	RAM (GB)
chr20:60061-4060065	865 267	1.9	0.88	385	2.27
chr20:4060066-8060066	880 899	2	0.85	388	2.22
chr20:8060067-12515479	961 591	2.1	0.77	424	2.05
chr20:12515480-16768988	917 468	2	0.73	407	1.97
chr20:16768989-21050967	931 010	2	0.71	413	1.92
chr20:21050968-31549151	1 919 134	4.2	1.20	834	3.06
chr20:31549152-38282825	1 436 549	2.8	0.99	662	2.63
chr20:38282826-43181963	1 056 144	2.2	0.76	462	2.06
chr20:43181964-47619489	955 970	2	0.79	416	2.09
chr20:47619490-51789198	923 178	2	0.80	404	2.12
chr20:51789199-55789212	911 452	2	0.81	405	2.13
chr20:55789213-59874964	925 442	2	0.84	409	2.20
chr20:59874965-64334101	1 096 089	2.4	0.93	480	2.42
Total	13 780 193	**29.6**	**11.06**	**-**	**29.15**

Table 3.12: BCF input files and μ-PBWT index disk usage results on UK Biobank SNP array data. All panels have 976 754 sequences/haplotypes.

Chr	Sites	BCF (GB)	μ-PBWT (GB)	Chr	Sites	BCF (GB)	μ-PBWT (GB)
1	54 432	4.2	13	12	32 011	2.6	8
2	53 433	4.2	13	13	22 344	1.9	6.3
3	44 935	3.6	11	14	21 708	1.8	5.8
4	41 678	3.3	10	15	21 286	1.8	6.1
5	40 020	3.2	9.6	16	24 314	2.0	6.4
6	46 515	3.9	9.4	17	22 856	1.9	6.2
7	36 682	3.0	8.9	18	19 432	1.6	5.8
8	34 141	2.7	8.2	19	19 845	1.6	5.3
9	29 581	2.4	7.7	20	17 515	1.5	5.2
10	33 086	2.7	8.2	21	9940	0.8	3.4
11	33 827	2.7	7.8	22	11 160	0.9	3.6

and without precomputed k-support values, *i.e.* recomouting them at query time, (`https://github.com/dlcgold/muPBWT/tree/k-smem-live`). We compared our tool against the k-MPSC implementation by Sanaullah et al. [90], referred here as k-MPSC, on a subset of the autosome panels from the 1000 Genomes Project data. In detail, we selected the panels for chromosomes 22, 20, 18, and 2, extracting 30 haplotypes to use as queries. After this step, the input panels have 4978 haplotypes.

We note that k-MPSC requires the input in VCF while μ-PBWT can also read BCF files. Despite this, we note that only 30% of the k-MPSC indexing time is spent reading the VCF input; however, this is a substantial limitation if we want to analyze larger panels due to the VCF's disk footprint. Moreover, in [90], the k-MPSC authors pre-processed the panel with P-smoother [100] to correct the panel before computing k-MPSC. This is outside the scope of the comparison proposed in this thesis; hence, we do not use P-smoother on the tested panels.

Regarding μ-PBWT, having $k = 1$, the computation of leftmost MPSC, a rightmost MPSC, and a length-maximal MPSC (from the set of SMEMs), is directly proportional to the classical μ-PBWT computation of the MS. Moreover, we can compare our k-SMEM results with Sanaullah's k-MPSC, since computing k-MPSC from k-MS requires at most a single scan over the k-MS array, just as for computing k-SMEMs. To conduct complete experiments, we tested $k = 1$, as a baseline, and two non-trivial values for k: $k = 50$ and $k = 200$.

μ-PBWT benchmark on the 1KGP data In Table 3.13 we report both indexing and querying results. As expected, with pre-computed k-support values, the μ-PBWT takes up to half the computation time of the variant without precomputed values, as k increases. Note that, having few runs in each PBWT column, we only need to store short bit-compressed integer vectors for the k-support values, resulting in only approximately 5% increase in memory usage to store the pre-computed values. We highlight that for $k = 1$, we do not require k-support values; thus, the memory usage remains the same in both variants.

Focusing on k-MPSC, its indexing phase consists of the PBWT computation of both panel and queries. Requiring a full uncompressed PBWT, k-MPSC takes almost two orders of magnitude more memory than both μ-PBWT approaches. These results are also consistent with what we already discussed regarding the

memory usage of Durbin's PBWT.

Regarding query results, computing k-support values at query time increases execution time. This increase scales with the value of k, and it is mainly caused by the use of the Φ data structure to navigate DA values. With $k = 200$, the k-SMEMs computation with the precomputed values is three times faster than computing them at query time. We observe that k−MPSC, having random access in constant time to all the PBWT arrays, is 20 times faster in querying than μ-PBWT with precomputed values and up to 60 times faster than the variant without pre-computed k-support values. Looking at k-MPSC results,

These experimental results confirmed the theoretical discussion in Section 2.6: the k-MPSC algorithm does not scale linearly with k, as it requires keeping the full PBWT in memory. Regarding μ-PBWT query memory usage, we can apply the same reasoning as for indexing, with only a small overhead due to the pre-computed k values. Finally, we conjecture that the increase in memory usage for larger k values is mainly due to the additional support variables used during computation.

In Table 3.13, k-MPSC results for the chromosome 2 panel are absent due to the memory usage of k−MPSC, which exceeds our hardware constraints (256GB of RAM). Instead, μ-PBWT results are reported, proving again that it scales sublinearly in space and can index and query large haplotype panels.

3.3.5 Genotype phasing with the μ-PBWT

Before discussing our results, we want to highlight that using the μ-PBWT to solve the genotype phasing problem combinatorially is a proof-of-concept, which is inefficient for most of its subtasks; however, we present some preliminary experimental results. In detail, the selection of the complementary sequences and the greedy algorithm used to select the best haplotype pairs can be further optimized, both theoretically and practically.

We compared μ-PBWT with Beagle [28, 29], executed with default parameters, on the HapMap [174] chromosome 2 genetic map for the GRCh38 reference genome, using the same phasing dataset described earlier. We recall that we have a SNP-only reference panel with 68 haplotypes/rows and 229 300 columns/sites. As a target, we have a simulated genotype with mutation rate $\varepsilon \in \{1\%, 3\%, 5\%, 10\%, 20\%\}$.

Table 3.13: Indexing and querying wall clock time and max memory usage comparison on chromosomes 2/18/20/22 panels from 1KGP data. In this setup, we consider 30 queries, and reference panels with 4978 sequences.

			Wall Clock Time (seconds)			Max Memory usage (GB)		
			μ-PBWT			μ-PBWT		
Chr.	Task	k	(pre)	(no-pre)	k-MPSC	(pre)	(no-pre)	k-MPSC
2	*Index*	1	1381	1384	-	6.45	6.45	-
		50	1643	1387	-	6.62	6.45	-
		200	2798	1418	-	6.62	6.45	-
	Querying	1	254	247	-	5.87	5.87	-
		50	1687	2471	-	6.57	6.42	-
		200	3563	11834	-	8.45	8.31	-
18	*Index*	1	425	424	5750	3.26	3.26	168.58
		50	697	450	5750	3.36	3.26	168.58
		200	807	429	5750	3.36	3.26	168.58
	Querying	1	70	70	128	1.92	1.92	171.19
		50	739	817	229	2.13	2.08	171.30
		200	1740	4263	119	2.74	2.71	171.22
20	*Index*	1	347	346	4485	1.74	1.74	129.83
		50	535	354	4485	1.78	1.74	129.83
		200	671	347	4485	1.78	1.74	129.83
	Querying	1	55	55	173	1.54	1.54	136.04
		50	485	629	740	1.73	1.68	136.28
		200	1177	3487	464	2.24	2.19	136.98
22	*Index*	1	191	189	2616	1.77	1.77	79.01
		50	304	193	2616	1.84	1.77	79.01
		200	400	194	2616	1.84	1.77	79.01
	Querying	1	31	32	130	0.98	0.98	82.31
		50	336	392	171	1.11	1.07	82.85
		200	726	2054	133	1.47	1.44	82.48

To evaluate the phasing accuracy, we computed: a) the switch error rate, which is the rate of phase changes between the predicted haplotypes with respect to the ground-truth haplotypes,, b) the mismatch rate, which is the rate at which two switch errors occur at consecutive positions, and c) the percentage of heterozygous sites that have been phased over the total number of heterozygous sites.

To compute these metrics, we used several utility scripts from HapCUT2 [175][1], considering the two original haplotypes used to generate the target non-mutated genotype as the ground truth.

In Figure 3.20, we present the accuracy obtained by Beagle and μ-PBWT at different mutation rates; more detailed results are in Table 3.14. With $\varepsilon = 0\%$, we have results consistent with how the two tools address the haplotype phasing problem. In fact, Beagle, being a statistical approach, is not expected to recover the exact solution, even when two haplotypes in the reference panel fully explain the target genotype. In contrast, the μ-PBWT phasing algorithm adopts a combinatorial strategy that guarantees finding the haplotype pair describing the target, whenever such a pair exists in the reference panel. With $\varepsilon = 0\%$, μ-PBWT correctly detects these two haplotypes, resolving the target genotype, with both the switch error rate and the mismatch rate equal to 0.0%. At the same time, Beagle's output contains some errors, as evidenced by both metrics being greater than zero. In any case, note that Beagle tries to phase all the heterozygous sites every time, while the μ-PBWT algorithm is designed to leave unphased sites if a phasing solution is not supported by surrounding homozygous regions.

The accuracy of μ-PBWT decreases dramatically when we increase ε. As in Figure 3.20.c, even with $\varepsilon = 5\%$, μ-PBWT phases only 62.6% of heterozygous sites. Increasing the mutation rate, the target will have larger heterozygous regions that lack valid anchors, remaining unphased. This issue can be addressed by increasing the number of haplotypes in the reference panel, highlighting the strong dependency of μ-PBWT on the quality of the reference panel. Moreover, μ-PBWT can estimate haplotype pairs only if the positional cover supports them; thus, around each mutation site, μ-PBWT will likely not report the original target haplotypes without mutations, as it is unable to match them. However, examining the μ-PBWT results, we found that one of the two selected haplotypes often coincides with the target

[1] `https://github.com/vibansal/HapCUT2/tree/master/utilities`

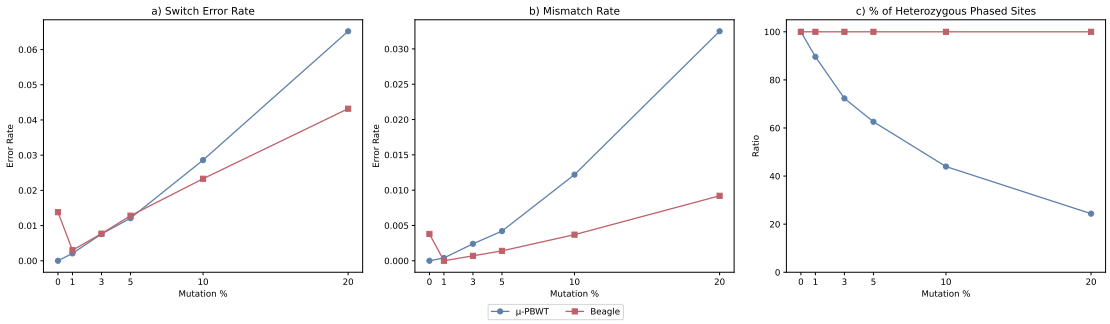


Figure 3.20: Phasing accuracy metrics results of μ-PBWT and Beagle at different genotype mutation rates: (a) switch error rate, (b) mismatch rate, and (c) the percentage of heterozygous sites phased over the total number of heterozygous sites.

sample's haplotype. Beagle can better handle high mutation rates, thanks to its probabilistic approach. The underlying Li-Stephens HMM model appears to be more robust to genotype mutations, allowing Beagle to phase more accurately, even for highly mutated targets.

Figure 3.21 presents the computational resources used by μ-PBWT and Beagle. Despite the running time being similar, μ-PBWT becomes slower as the mutation rate increases. This occurs because the μ-PBWT algorithm must switch between more haplotype pairs, frequently recomputing new sets of consistent pairs rather than continuing with the current matching pair. Each time the algorithm cannot continue phasing the next genotype region with any current pair, it must restart, recomputing larger sets of possible pairs. This increases both the overall execution time and memory usage.

Increasing the reference panel size, in terms of haplotypes, would worsen indexing performance, both for computing and storing the index on disk. However, it would likely reduce the number of times we need to compute a new set of consistent haplotype pairs. On the other hand, the running time of Beagle remains constant as the mutation rate increases, being independent of the number of haplotype recombinations required to explain the target.

Regarding memory, μ-PBWT is very memory-efficient, requiring approximately 7 times less RAM than Beagle. These results demonstrate again that μ-PBWT can be run on low-end hardware, including a commodity laptop, or on the public cloud

Table 3.14: Additional phasing accuracy metrics results of μ-PBWT and Beagle at different genotype mutation rates. The N50 is the length of the shortest haplotype block such that half of all phased variants are contained within blocks of that length or longer. The flat rate is the minimum hamming distance between the two assembled haplotypes for a given block and the ground truth.

(a) Results with 0% and 1% mutation rate.

Mutation rate	0 %		1 %	
	Beagle	μ-PBWT	Beagle	μ-PBWT
Switch error rate	0.0138	0.0	0.003	0.0021
Mismatch rate	0.0038	0.0	0.0	0.0004
Flat rate	0.491	0.0	0.454	0.0121
Phased count	60 604	60 604	60 862	54 547
N50	242 130 407	242 130 407	242 130 407	242 130 407
No. of SNPs max block	60 604	60 604	60 862	54 547

(b) Results with 3% and 5% mutation rate.

Mutation rate	3 %		5 %	
	Beagle	μ-PBWT	Beagle	μ-PBWT
Switch error rate	0.0077	0.0076	0.0128	0.0121
Mismatch rate	0.0007	0.0024	0.0014	0.0042
Flat rate	0.4721	0.0272	0.4754	0.0397
Phased count	61 362	44 365	61 995	38 803
N50	242 130 407	242 130 407	242 065 471	242 065 471
No. of SNPs max block	61 362	44 365	61 995	38 803

(c) Results with 10% and 20% mutation rate.

Mutation rate	10 %		20 %	
	Beagle	μ-PBWT	Beagle	μ-PBWT
Switch error rate	0.0233	0.0286	0.0432	0.0652
Mismatch rate	0.0037	0.0122	0.0092	0.0325
Flat rate	0.4878	0.0745	0.4920	0.1638
Phased count	63 153	27 760	65 265	15 903
N50	242 130 407	242 046 972	242 133 632	242 088 948
No. of SNPs max block	63 153	27 760	65 265	15 903

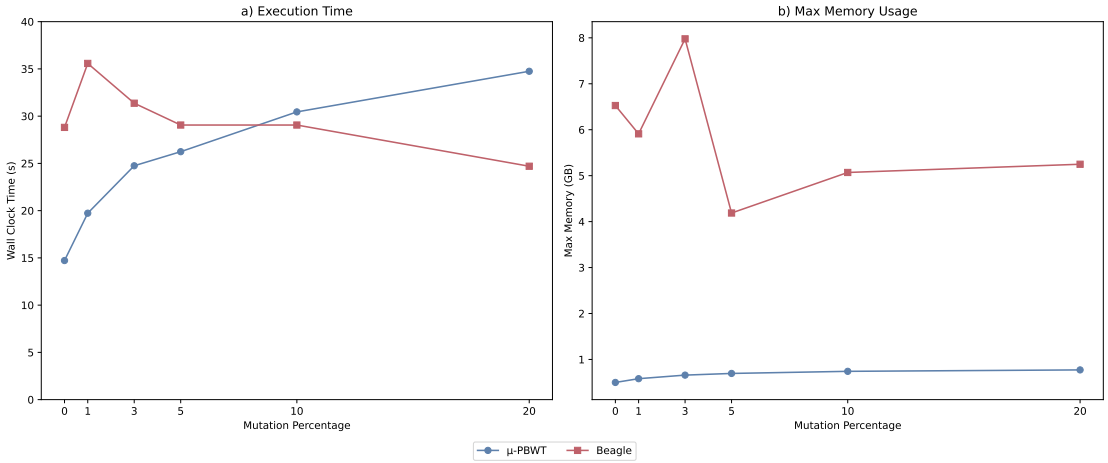


Figure 3.21: Genotype phasing performances comparison between μ-PBWT and Beagle at different mutation rates: (a) running time, and (b) memory usage.

at a fraction of the cost of running Beagle. We conjecture that we can further optimize the μ-PBWT execution time but not its space usage, since in genotype phasing approaches it is often dominated by the reference panel index space, which is strongly compressed by μ-PBWT. On the other hand, additional data stored during phasing by the μ-PBWT, such as the haplotype pairs set, can be stored more efficiently; however, we do not expect substantial improvements in memory requirements.

3.4 Conclusions

In this chapter, we presented and benchmarked various data structures and algorithms for the run-length encoded Positional Burrows–Wheeler transform (RLPBWT). We evaluated both theoretically and experimentally several possible composite data structures for the RLPBWT and proposed the μ-PBWT as the one offering the best trade-off between index size and query efficiency, demonstrating that it outperforms the `MAP+LCE+PERM` implementation of RLPBWT, theoretically identified as the most space efficient RLPBWT composite data structure. We demonstrated that μ-PBWT is not only capable of finding set-maximal exact matches (SMEMs) shared between a set of haplotype sequences and a query hap-

lotype, but can also be extended to address a range of other problems. These include stringology-related tasks, such as k-SMEMs and (k-)MPSC, as well as more biologically oriented problems, such as genotype phasing.

We proved that μ-PBWT index is lighter than other compressed PBWT indices, like the Syllable-PBWT [98], and even two orders of magnitude lighter than a full PBWT [13, 25, 90], indexed for computing matches with an external query not included in the reference panel itself. We also showed that μ-PBWT is not significantly less time-efficient than the original Durbin PBWT for computing SMEMs, and that, in specific scenarios, such as high-quality genotype targets and large reference panels, it can be effectively used for haplotype phasing.

Overall, focusing on the μ-PBWT, we proposed a tool capable of scaling over large haplotype datasets, such as the UK Biobank's whole-genome sequencing data, potentially allowing for indexing and querying such panels on a commodity laptop. Hence, we proved that it should be used for large-scale phasing, genotyping, and imputation workflows [139, 9, 148, 176]. The high versatility of the μ-PBWT matching algorithm, based on the use of the matching statistics array, makes the μ-PBWT suitable for inclusion in various state-of-the-art phasing and imputation tools, such as GLIMPSE2 [139, 9, 148], opening up unprecedented opportunities for comprehensive genetic studies on population-scale datasets.

Additional future work should focus on further optimizing the μ-PBWT data structure. One possible direction is the integration of the so-called move data structure [161, 162]. This would make the MS algorithm optimal in time. Since both the SMEMs and MPSC algorithms are derived from MS, they would also become optimal, achieving a query running time of $\mathcal{O}(w)$ for an input set of w-length sequences, without losing the sublinearity in space.

Chapter 4

Indexing Pangenome Graph

In this chapter, we present a method for indexing and querying pangenome graphs to solve the String-to-Graph pattern-matching problem. In detail, we developed `gindex`, a multidollar-BWT–based pangenome graph index designed to overcome two significant limitations of the most widely used state-of-the-art index: its high memory requirements during index construction and its possibility to report false positives for queries longer than 256bp. We want to highlight that `gindex` is more of a prototype than a ready-to-use tool, developed to explore a different index technique.

Theoretically, our algorithm runs in $\mathcal{O}(|P||\mathsf{BWT}|\beta)$ time, where P is the query, BWT is the multi-dollar BWT, and β denotes the time required for a single backward step, while using at least $\mathcal{O}(|\mathsf{BWT}|)$ space (not considering optimized BWT indexes like the r-index). In any case, both time and space complexity depend on the chosen BWT index approach. It is worth noting that the current implementation introduces some additional overhead. Nevertheless, it can index human pangenomes such as those produced by the Human Pangenome Reference Consortium (HPRC) [33]. However, it is considerably slower in querying compared to the state-of-the-art tool, `GCSA2`, which, however, is not always able to build the index due to its high memory requirements.

Recall that the String-to-Graph pattern-matching problem can be formalized in high-level as:

Input:	Pangenome graph $G = (V, E)$ and query string P
Output:	All exact occurrences of P in G

Publications and conferences `gindex` [32], was presented in Venice, Italy, at the *23rd Symposium on Experimental Algorithms (SEA 2025)* – conference proceedings published in the *Leibniz International Proceedings in Informatics (LIPIcs) – Dagstuhl*.

Chapter outline In section 4.1 we will present the state-of-the-art and the motivations that led us to develop `gindex`. This section will include some additional preliminaries to extend what we introduced in Section 2.8. In section 4.2, we will discuss in detail our algorithmic contribution. In section 4.3 we will compare our implementation against the state-of-the-art tool, *i.e.* GCSA2, and, finally, in section 4.4 we will draw conclusions and possible future developments.

4.1 Motivations and State-Of-The-Art

A pangenome graph is a directed graph with nodes labeled by genomic sequences. As described in Section 2.8, string-to-graph exact pattern matching is one of the most common problems in this scenario. In computational biology, a pangenome graph can represent multiple genomic sequences in a single data structure without redundancy. Moreover, this new paradigm replaces the traditional use of a single genome as a reference with a representation that can encode variations in a population of sequences or genomes [2].

From a theoretical perspective, indexing a pangenome graph to facilitate pattern matching is a well-studied problem [177, 178]. From a more practical point of view, the Burrows-Wheeler Transform [10] is the core mechanism used to build pangenome graph indices, such as `GCSA` [18], which enables indexing and querying paths in an acyclic sequence-labeled graph. While `GCSA` was based on indexing the paths of an automata representing the pangenome graph, the new version, `GCSA2` [30, 31], relies on a succinct de-Bruijn graph to index k-mers, using a core data structure strongly inspired by the well-known BOSS data structure [107]. Unfortunately, `GCSA2` presents two major issues: a) using queries of length greater

than 256 base pairs would result in false positive results, and b) indexing has high memory requirements.

Note that nowadays we deal with particularly long queries/reads using modern sequencing technologies, such as PacBio and Nanopore. In addition, in recent years, we were able to build large pangenome graphs, such as the ones constructed by the Human Pangenome Reference Consortium [3] built from a large collection of human genomes [33]. Moreover, in recent years, new graph indexes have appeared in the literature, such as the Graph BWT [142], in short GBWT, used to index and query the haplotypes in a pangenome graph. Finally, similar indexing techniques are used for Elastic Degenerate Texts [179]. In this context, a generalization of the extended-BWT [180, 73], *i.e.* the BWT for a collection of sequences, is used as an index for Elastic Degenerate Texts, which are a restriction of directed acyclic labeled graphs.

4.1.1 Preliminaries

Most of the theoretical preliminaries needed to fully understand this chapter are already described in Chapter 2, so we refer the reader to that chapter for string/graph definition, string indexing, and the fundamental notions on string-to-graph pattern matching. In this subsection, we will recall some fundamental definitions and some additional concepts. If not differently specified, in this chapter, when we refer to a graph, we refer to a node-labeled graph.

First, we recall the definition of the multidollar-BWT and some properties we will leverage in the methods section.

Definition 24 (Multidollar-BWT)**.** *The Multidollar-*BWT *of a multiset of strings* $X = \{x_1, \ldots, x_h\}$ *is the Burrows-Wheeler Transform of* $T_X = x_1\$_1x_2\$_2 \ldots x_h\$_h$*, still denoted by* BWT*.*

As a consequence of Definition 24, we have two properties which will be used in the `gindex` approach.

Property 1. *Let* $T = x_1\$_1 \cdots x_h\$_h$*, and let* $1 \leq i, j \leq |T|$ *be such that* $\mathsf{BWT}[i] = \$_j$*. If* $j = h$*, then* $T[\mathsf{SA}[i]] = x_1[1]$*; otherwise,* $T[\mathsf{SA}[i]] = x_{j+1}[1]$*.*

Property 2. *Let $T = x_1\$_1 \cdots x_h\$_h$. Then, for every $1 \leq i \leq h$ it holds that $\mathsf{BWT}[i] = x_i[|x_i|]$. That is, the first h symbols of* BWT *correspond to the last symbols of $x_1, \ldots, x_h$.*

We will describe the use of these properties in the next section and in Example 7.

Moreover, we now recall the node-labeled graph definition. Then, we formally define the type of occurrences we are interested in when solving the string-to-graph exact pattern matching problem.

Definition 25 (Node-labeled Graph)**.** *A node-labeled graph is a triplet $G = (V, E, \ell)$ such that V is a set of nodes, $E \subseteq V \times V$ is a set of directed edges, and $\ell : V \to \Sigma^+$ is a labeling function that assigns to each node $v \in V$ a label $\ell(v) \in \Sigma^+$.*

We extend the labeling function $\ell(\cdot)$ to any path $\pi = (v_1, \ldots, v_k)$ by simply representing $\ell(\pi) = \ell(v_1) \cdots \ell(v_k)$. To locate where a string pattern P occurs in the node-labeled graph G, we define two notions: a) the coarse occurrence, in which we are interested only in the starting node (say v) of a path where P occurs, and b) the fine occurrence, where we are also interested in the full path where P occurs. Formally:

Definition 26 (Coarse and Fine Occurrences)**.** *Let $G = (V, E, \ell)$ be a graph, and P be a pattern. A fine occurrence of P in G is a path $\pi = (v_1, v_2, \ldots, v_k)$ such that P occurs in $\ell(\pi)$, starting in v_1 and ending in v_k. In such a case, we also say that the sole node v_1 is a coarse occurrence of P in G.*

Notice that, in contrast to the string-to-string exact pattern matching problem, both a fine and a coarse occurrence can fall in more than one path. Given these two occurrences' definitions, the goal of this work is to solve the following:

Problem 7 (Coarse (Fine) Graph Pattern Matching Problem)**.** *Given as input a graph $G = (V, E, \ell)$, and a pattern P, the goal is to return as output the set of all coarse (fine) occurrences of P in G.*

A node-labeled graph index: `GCSA2`

To conclude this section, we want to focus on the main state-of-the-art tool in this context: `GCSA2`. `GCSA` (Generalized Compressed Suffix Array) [18] and `GCSA2` [30,

31] are BWT-based and FM-index-based tools developed initially for index and query directed acyclic graphs. `GCSA` was later extended also to support cyclic graphs in `GCSA2`, which is currently integrated as a part of the `VG toolkit` [20].

The author, in the original paper of `GCSA2`, proved the efficiency of this index, as `GCSA2` performs pattern matching on the pangenome graph almost as well as FM-index approaches on texts with a similar amount of base pairs [30].

The two main limitations of `GCSA2`, namely the high memory requirements during indexing and the occurrence of false positives for long queries, are largely due to its underlying algorithm, which uses multiple iterations of the prefix-doubling method to construct the underlying de-Bruijn graph. This algorithm repeatedly merges paths of length k into paths of length $2k$, resulting in hundreds of gigabytes of RAM usage for large graphs, even though temporary data can be stored on disk [30]. Moreover, the length of the queries is limited by the k-mer size k used in this de-Bruijn graph. As noted in the `GCSA2` documentation[1], the current maximum allowed k-mer size is $k = 256$.

4.2 Methods

In this work, we focus on solving Problem 7 on pangenome graphs, where nodes are labeled with strings over the DNA alphabet $\Sigma = \{A, C, G, T\}$ or $\Sigma = \{A, C, G, T, N\}$, with the symbol N denoting missing information. It is worth noting that our method can be generalized to any node-labeled graph. For simplicity, given a node-labeled graph $G = (V, E)$, we assume $V = v_1, v_2, \ldots, v_{|V|}$, and we denote the labeling function as $L_i = \ell(v_i)$ for each $i = 1, 2, \ldots, |V|$.

Coarse occurrences Consider the classical scenario in which we are interested in finding all the coarse occurrences of a string pattern P in G. The main idea is to simultaneously search for increasing suffixes of P across all labels of G. We start from the $|P|$-th suffix of P, *i.e.* the last symbol of the pattern, and we proceed in a backward-search way until we consider all the occurrences of the 1-th suffix of P, *i.e.* the entire pattern. Note that we are scanning the pattern from the right

[1] `https://github.com/jltsiren/gcsa2`

to the left. Whenever we reach the beginning of a label L_i, we jump to all the incoming neighbors of the corresponding node v_i and continue the search inside their labels. The predecessors can be retrieved by looking at the graph topology. To do it efficiently, we construct $\mathsf{BWT}_{\mathscr{L}}$: the multidollar-BWT of the collection of all the node labels $\mathscr{L} = \{L_i | \forall\, i = 1, 2, \ldots, |V|\}$. We will refer to $\mathsf{BWT}_{\mathscr{L}}$ as BWT and our index, namely `gindex`, can be identified as the pair (BWT, G).

To link the graph G and BWT, we leverage Properties 1 and 2, which we now restate in a way that better suits our graph-oriented application.

Property 3. *Let $G = (V, E, \ell)$ be a graph, let* BWT *denote the multidollar-*BWT *constructed over the collection of all the node labels of G, and let $[b, e]$ be a* c*-backward-interval obtained after a backward-step, where $\mathsf{c} \in \Sigma$. Consider an index i such that $b \leq i \leq e$ and $\mathsf{BWT}[i] = \$_j$ for some $1 \leq j \leq |V|$. If $j = |V|$, then label L_1 has been exhausted; otherwise, label L_{j+1} is exhausted.*

Property 4. *Let $G = (V, E, \ell)$ be a graph, and let* BWT *denote the multidollar-*BWT *constructed over the collection of labels of G. Then, all the* BWT *symbols $\mathsf{BWT}[1], \ldots, \mathsf{BWT}[|V|]$ correspond to the last characters of the labels $L_1, \ldots, L_{|V|}$.*

We can now present our algorithm, which works similarly to the classical string-to-string BWT-based pattern matching algorithm. As already mentioned, we proceed by scanning the pattern P from the right to the left.

In the classical string-to-string algorithm, a backward-interval is transformed into a new backward-interval via the LF-mapping. This step is commonly known as backward-extension or backward-step. Dealing with a node-labeled graph, we need to generalize this property: in case a label L_i is exhausted, all the predecessor nodes of v_i must be explored afterwards, adding one new backward-interval per node. We maintain a list $\mathcal{L}$ of all such backward-intervals, which can be distinguished into two types: a) those obtained by jumping to a predecessor node, using Property 4 after having exhausted a node label, and b) at most one additional backward-interval that represents all the occurrences that fall entirely inside single labels. The latter is the same backwards interval we would have in each iteration of the classical string-to-string algorithm, where the pattern suffix can not fall in between two consecutive strings due to the absence of the dollar symbol in the pattern. In the

graph context, this backward interval represents all possible matches of the current pattern suffix within single labels, without spanning across multiple labels.

Algorithm 4.1 presents our approach to solving Problem 7, finding all coarse occurrences. It is divided into two functions: the main function MATCH and the auxiliary procedure MULTIEXTEND, which is the generalization of the backward-extension to consider multiple intervals.

We start computing the generalized backward-search algorithm (lines 3–4). In this way, after scanning the entire pattern, we will have in $\mathcal{L}$ all the backward-intervals which represent the starting nodes of all the occurrences of P in G. In detail, MULTIEXTEND($\mathcal{L}, \mathsf{c}$) is used to update the backward-interval set $\mathcal{L}$, extending each interval in the set using the current pattern symbol, and querying the graph topology to retrieve new intervals when labels are exhausted. After each backward-extension, whose result is added to $\mathcal{L}$ if and only if the new backward-interval is not empty (lines 21–23), we check the presence of dollar symbols in this new interval. Thanks to property 3, each dollar found marks that a label is exhausted and that we need to continue the pattern matching in all the predecessors of the corresponding node. According to Property 4, we know how to retrieve the last symbols of the labels of these nodes in the BWT; therefore, we add to $\mathcal{L}$ a new backward-interval of size one that covers exactly each one of these BWT symbols (lines 24–27), *i.e.* potentially one backward-interval per predecessor. To avoid adding unnecessary labels, we could add the new interval only if the predecessor node ends with the following pattern symbol. In any case, these intervals would be discarded if there is a mismatch in the next iteration. At the end of each iteration, we can merge all the consecutive backward-intervals, reducing the number of backward-extensions that will be performed in the next iteration (line 28).

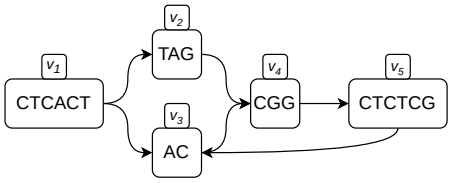
After visiting the whole pattern, we obtain the final set of backward-intervals $\mathcal{L}$ which represents all the occurrences of P in G. Now we need to retrieve the starting nodes of these occurrences. At first, if $\mathcal{L}$ is empty we do not have any occurrence of P in G (lines 5–6). Conversely, if $\mathcal{L} \neq \emptyset$, we have to apply the same algorithm used to apply the inverse permutation on the BWT, *i.e.* reconstructing the original text from its BWT. From now on, each backward-interval in $\mathcal{L}$ is split into one-size intervals. In detail, for each interval in $\mathcal{L}$, we perform the classical BWT text reconstruction algorithm, iteratively extending the current one-size interval with

the current BWT symbol, until we reach a dollar. In other words, $\mathcal{L}$ contains the starting points of each match, which fall within node labels that are unknown at this stage. These node IDs are unknown either because the interval is the result of a merging step or because the match falls on a single label. Hence, to get each starting node, we need to traverse these labels up to their first symbol to exploit Property 3 (lines 7–16).

Fine occurences If we want to return all fine occurrences and not just the course occurrences, we need to augment this procedure by storing, for each backward-interval, the ordered list of the already traversed nodes. These lists are updated each time we consider a new predecessor node. Another difference with respect to the coarse occurrences algorithm is that we cannot merge consecutive backward-intervals. In fact, merging two consecutive backward-intervals would require additional information to disambiguate the visited node lists of the two starting intervals, significantly increasing both time and memory requirements.

Without this merging step, we have many more backward-intervals to extend at each iteration, resulting in slower performances in practice. Furthermore, if different paths visit the same interval, we have to add to $\mathcal{L}$ the same interval for each visited path, resulting in high redundancy. Notice that in this scenario $\mathcal{L}$ needs to be considered as a multiset. We will present some experimental results on considering fine occurrences in Section 4.3.

Example 7. *To better understand our approach, we present a simple example here. Consider the following node-labeled graph:*



We extract the five labels from the nodes: $L_1 = CTCACT$, $L_2 = TAG$, $L_3 = AC$, $L_4 = CGG$, $L_5 = CTCTCG$.

We concatenate all of them in a text T*, separating them with five unique*

Algorithm 4.1 High-level view of all coarse occurrences finding algorithm.

```
1: function MATCH(P, BWT, G)
2:     L ← {(1, |BWT|)}
3:     for i = |P| downto 1 do   ▷ Scan the pattern and update the backward-intervals set L
4:         L ← MULTIEXTEND(L, P[i])
5:     if L = ∅ then
6:         P not found in G
7:     for all (b, e) ∈ L do                          ▷ Find all starting nodes of the matches
8:         for all i ∈ BWT[b..e − 1] do
9:             c ← BWT[i]
10:            while c ≠ $_j for some j do
11:                (i, i + 1) ← EXTEND(i, i + 1, c)          ▷ Classic BWT backward-extension
12:                c ← BWT[i]
13:            if c = $_|V| then
14:                Report occurrence at node v_1
15:            else
16:                Report occurrence at node j + 1, where c = $_j
17:
18: function MULTIEXTEND(L, c)
19:     L_new ← ∅
20:     for all (b, e) ∈ L do
21:         (b′, e′) ← EXTEND(b, e, c)                    ▷ Classic BWT backward-extension
22:         if e′ > b′ then
23:             PUSHINTERVAL(b′, e′, L_new)                ▷ Add interval (b′, e′) to set L
24:         NODES ← GETENDNODES(b, e)          ▷ Get ending nodes by dollars in [b..e − 1]
25:         for all v ∈ NODES do
26:             for all v_a ∈ PREDECESSORS(v) do           ▷ Jump to all the predecessors of v
27:                 PUSHINTERVAL(a, a + 1, L_new)
28:     L_new ← MERGE(L_new)                               ▷ Merge consecutive intervals
29:     return L_new
```

terminating symbols:

$$T = CTCACT\$_1TAG\$_2AC\$_3CGG\$_4CTCTCG\$_5,$$

*and we build its multidollar-*BWT*. Next, we show the whole multidollar-*BWT*, including the rotation matrix and the* F *array. We recall that this array is the first column of the rotation matrix and contains all the symbols of T in lexicographical order.*

i	F		BWT
1	$\$_1$	TAG$\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCAC	**T**
2	$\$_2$	AC$\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1$TA	**G**
3	$\$_3$	CGG$\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2$A	**C**
4	$\$_4$	CTCTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3$CG	**G**
5	$\$_5$	CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTC	**G**
6	A	C$\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1$TAG	$\$_\mathbf{2}$
7	A	CT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CT	**C**
8	A	G$\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1$	**T**
9	C	$\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2$	**A**
10	C	ACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$C	**T**
11	C	G$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTC	**T**
12	C	GG$\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2$AC	$\$_\mathbf{3}$
13	C	T$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTC	**A**
14	C	TCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG	$\$_\mathbf{5}$
15	C	TCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$C	**T**
16	C	TCTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3$CGG	$\$_\mathbf{4}$
17	G	$\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1$T	**A**
18	G	$\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3$C	**G**
19	G	$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCT	**C**
20	G	G$\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3$	**C**
21	T	$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCA	**C**
22	T	AG$\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCACT	$\$_\mathbf{1}$
23	T	CACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$	**C**
24	T	CG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CT	**C**
25	T	CTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$	**C**

We consider, for example, the backward-interval $[13, 17)$ *to illustrate how the extension step works. Suppose we want to extend this interval with a specific pattern*

symbol $\mathsf{c} \in \Sigma$. *In the positions given by the interval, we have* $\$_4$ *and* $\$_5$ *in the* BWT*, meaning that we are, respectively, at the beginning of node 5 and node 1. Node 1 has no predecessors, while node 5 has only node 4, therefore, we add the interval* [4..4] *to our backward-intervals set* $\mathcal{L}$*, before continuing with the next iteration. Finally, we need to extend* [13..16] *by* c *and add the resulting interval to* $\mathcal{L}$ *to continue the pattern search.*

4.2.1 Complexity analysis

The complexity of one iteration in Algorithm 4.1 scales in the size of the backward-intervals set, having to perform one backward-extension for each interval. Assuming we can merge all consecutive intervals, solving coarse Problem 7, we need to consider $|\mathsf{BWT}|/2$ intervals for the next step in the worst case. On the other hand, if we consider finding fine occurrences, we can not bind $\mathcal{L}$ cardinality due to possible intervals redundancy, making it difficult to establish a time and space complexity bound.

We focus on the first case, *i.e.* coarse occurrences. Having to iterate the pattern P and having to perform at most $|\mathsf{BWT}|/2$ extension for each symbol of P, we have in total $|P|(|\mathsf{BWT}|/2)$ backward-steps. In addition, we need to retrieve the ending nodes in each backward-interval and the corresponding predecessors. This time is bounded by the number of possible backward-intervals multiplied by the number of nodes: $(|\mathsf{BWT}|/2) \cdot |V|$. Assuming $|\mathsf{BWT}| \gg |V|$ (having at least two symbols, including the separator, per node in BWT), we can conclude that our algorithm runs in $\mathcal{O}(|P||\mathsf{BWT}|\beta)$ time, where β is the time of a single backward-step, which can vary according to the data structure used to implement the FM-index for the classical BWT backward-extension.

About space, we are bounded by the size of the BWT and the graph. We need to store the multidollar-BWT index, which requires theoretically at least $\mathcal{O}(|\mathsf{BWT}|)$ space (without considering run-length implementations), the graph topology as an adjacent list, which requires $\mathcal{O}(|V| + |E|)$ space, and the set of backward-intervals, which requires $\mathcal{O}(|\mathsf{BWT}|)$ space. In any case, the actual space requirements depend on the choice of the BWT index, which also affects the time complexity.

4.2.2 Preprocessing

Recalling Algorithm 4.1, after the first iteration, we will add to $\mathcal{L}$ one interval for each predecessor of every node that starts with the last symbol of the pattern. In fact, without considering any optimization for simplicity, like the merging step, we have to check all the dollars in the backward-interval we get after the first backward-extension. This interval will cover all the positions which have $P[|P|]$ in the F array. In Example 7, suppose that $P[|P|] = C$. Extending with C, we fall into the backward-interval $[9, 17)$, which contains three dollars: $\$_3$, $\$_5$ and $\$_4$. By Property 3, C is the first symbol of labels L_4, L_5, and L_1, and we need to add the corresponding backward-interval of each predecessor of these three nodes to $\mathcal{L}$.

We can expect, on average, that $|\mathcal{L}| \approx |V|/|\Sigma|$, assuming a uniform distribution of the first symbols on the labels L. Iteration by iteration, we expect the set size to decrease exponentially, having discarded all mismatching paths. This behavior is shown in Figure 4.1.

Now it is easy to see that one of the main bottlenecks of this algorithm is the extension of the backward-intervals set in the first iterations, due to the cardinality of $\mathcal{L}$. To scale on large pangenome graphs, such as whole-genome human pangenome graphs, we implemented a dynamic programming preprocessing algorithm that computes, for a fixed x, the backward-interval sets for any possible x-length string over the alphabet Σ during the indexing phase. In detail, we precompute all the matches for all $|\Sigma|^x$ strings, *i.e.* for any k-mer of size x. We refer to these precomputed sets as a cache of length x. Using these sets, we can skip the first x iterations during the querying phase and access the backward-interval sets for $P[|P| - x..|P|]$ directly. Therefore, despite the initial computation time and additional disk usage, we achieve a considerable speedup in query time. Note that this cache must be stored on disk and loaded entirely into memory at querying time. Increasing the value of x allows us to skip a longer pattern suffix, at the cost of additional disk and RAM usage. Therefore, we analyzed the results from Figure 4.1 to select a threshold that provides an acceptable trade-off.

To optimize this preprocessing computation, we developed a parallel algorithm that starts from the backward-intervals already computed for each $\sigma \in \Sigma$, which is then extended with each $\sigma' \in \Sigma$, using a dynamic programming approach; such

extensions can be computed in parallel. The process is repeated x times until all x-length strings have been matched. Note that using a naïve approach, we would have many redundant extensions to compute, *i.e.* each time two substrings of the x-length strings share the same suffix. For example, two patterns $CAAA$ and $TAAA$ both have the common AAA suffix, whose backward interval can be computed just once, since the backward-interval set is the same for all common suffixes. In practice, we implemented a tree-based approach in which each iteration extends a suffix with all the σ values, without extending already computed suffixes. With this approach, we have only $\sum_{i=1}^{x} |\Sigma|^i$ extensions, instead of $x \cdot |\Sigma|^x$. Being each extension independent, they can be executed in parallel. This parallel approach heavily reduces the impact of the cache computation time.

Figure 4.1 shows how the cardinality of the backward-interval sets $\mathcal{L}$ changes as we proceed with the backward-steps. This cardinality exponentially decreases at each iteration, reaching a sufficiently small value by around step 5 in the tested graphs. The vertical lines in the plot indicate the average positions at which the cardinalities reach 1. We experimented with various cache lengths to evaluate the impact of cache size, suggesting a length of around 7 to achieve a good trade-off between computational resources and query speed. Using high-length values would also lead to considering patterns that rarely occur during pattern matching (due to their higher granularity), resulting in wasted resources.

Although the cardinality of $\mathcal{L}$ is not strictly monotonically decreasing, because multiple predecessors may share the same ending symbol and temporarily increase its size, it is likely to decrease shortly afterward, eventually converging to the actual occurrences of the pattern.

4.3 Results

We implemented `gindex` in C++ using MFMI[2], a RopeBwt2 [181] and RLCSA [182, 78] fork, as multidollar-BWT index implementation, compiling with the `-O3` flag. Both the source code and the experimental pipeline are available at: `https://github.com/dlcgold/graph_index.git`. We conducted our experiments on a

[2] `https://github.com/ldenti/mfmi.git`

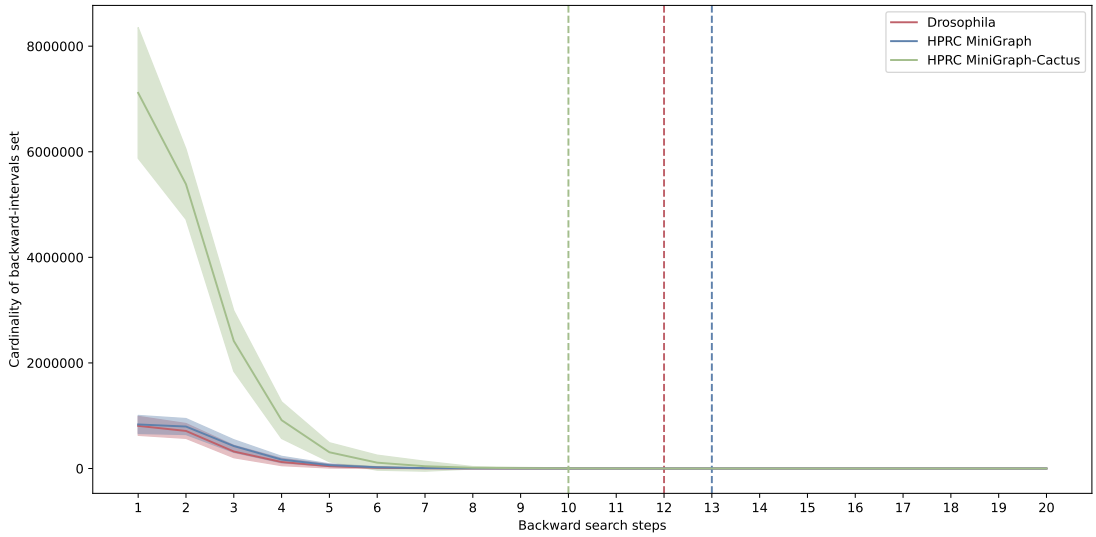


Figure 4.1: Average cardinality of backward-intervals set $\mathcal{L}$ the three input graphs discussed in Section 4.3, computed from a simulation of 100 random queries. The vertical lines represent the average positions over the 100 queries at which the cardinality drops to 1 for the first time.

machine equipped with an Intel Xeon CPU E5-4610 v2 (2.30GHz), 256GB of RAM, 8GB of swap, and Ubuntu 20.04.6 LTS 64-bit with kernel 5.15.0. We used the Unix `time` command to obtain the execution time and peak memory usage.

We evaluated the performance of `gindex` in solving Problem 7 by comparing it against `GCSA2` [30, 31], built using the `VG toolkit` [20], on real pangenome graphs using simulated queries. In detail, we evaluated both graph indexing and index querying. To simulate fixed-length queries from the input graph, we randomly select a starting node and an offset within its label, then traverse a random path originating from that node, concatenating the corresponding node labels until the desired length is reached.

4.3.1 Experimental datasets

In our experiments, we evaluated three pangenome graphs that differ in both graph size and the total length of the encoded sequences. At first, we considered a Drosophila pangenome graph with 10 million nodes, 12 million edges, a maximum degree of 15, an average degree of 1.18, and 144 million base pairs. It was built

by `VG` using 205 haplotypes [183]. This graph has been pruned to remove complex regions with numerous variants that prevent indexing with `GCSA2`[3], and for the same reason, each node contains at most 32 base pairs. Then we considered the Human Pangenome Reference Consortium (HPRC) pangenome graph [33] built using MiniGraph [184]. This graph has 12 million nodes, 13 million edges, a maximum degree of 74, an average degree of 1.02, and 3.2 billion base pairs. We slightly modified this graph to have all nodes with at most 256 base pairs, trying to avoid issues with `GCSA2` indexing[4] and fixing the maximal length of the labels to the maximum k-mers size supported by the tool: 256 base pairs. Finally, we considered the HPRC pangenome graph built using MiniGraph-Cactus [185]. This graph has 80 million nodes, 110 million edges, a maximum degree of 39, an average degree of 1.39, and 3.1 billion base pairs. No modification has been necessary to use this graph. Note that HPRC released another graph, built using PGGB [186]. This graph has 110 million nodes, 154 million edges, a maximum degree of 4255, an average degree of 1.4, and 8.4 billion base pairs. We cannot index it because it takes more RAM than is available; hence, we decided to exclude this graph from our experiments.

4.3.2 Experimental results

Table 4.1 shows the size on disk of all the auxiliary files of `gindex` for the three input graphs. We need to store the RopeBwt2 index, the adjacency list of the graph (storing the predecessors of each node), the label names (to map from an incremental integer nomenclature used in the implementation to the original node IDs), and the BWT dollars order. The latter is needed because RopeBwt2 does not distinguish between different dollars, using 0 to encode each dollar, $\{1, 2, 3, 4\}$ to encode $\{A, C, G, T\}$ (ignoring lowercase/uppercase), and 5 for every other symbol, including the N. Table 4.1 also reports the `GCSA2` index size.

We tested `gindex` on all the pangenome graphs with three different execution modes. Initially, we consider a no-cache mode, in which the graph is indexed without any precomputed cache, and only the starting nodes of each match (coarse

[3] `https://github.com/vgteam/vg/issues/4404`
[4] `https://github.com/jltsiren/gcsa2/issues/44`

Table 4.1: Auxiliary files produced by the methods, `GCSA2` failed to compute on the two HPRC graphs on our machine.

	Method					
	`gindex` (MB)					`GCSA2` (MB)
Graph	*RopeBwt2*	*Graph*	*Labels*	*Dollars*	***Total***	*Full Index*
Drosophila	79	174	80	160	**493**	566
MiniGraph	1496	200	99	198	1993	-
MiniGraph-Cactus	1491	1456	611	1222	4780	-

occurrences) are returned. Next, we introduce a cache length x mode, where the graph is indexed to return the starting nodes of each occurrence (coarse occurrences) while precomputing the cache for strings of length x. Finally, we define a full-path mode, where the graph is indexed without any precomputed cache, but the complete paths matching the query (fine occurrences) are returned.

Note that we do not use a cache in the full-path mode, as this would require storing all possible paths in the precomputed intervals, significantly impacting performance without a clear benefit, since `GCSA2` only reports the starting node of each match.

Drosophila graph Table 4.2 shows the indexing results for the Drosophila pangenome graph. We tested cache lengths 1, 4, 7, 10, and the no-cache mode. We also tested the full-path mode, which requires the same index step as the no-cache mode. Cache length 10 is presented for illustrative purposes only, and we do not expect it to be used in a real scenario. We consider the no-cache mode as the baseline. Using a cache length of 10 results in a 120x increase in computational time, whereas a cache length of 7 yields only a 3x increase. On the other hand, although an increase in disk space is inevitable, it is minimal, with a maximum of 2.41GB for cache 10, which is negligible on modern systems. For this graph, all the other `gindex` auxiliary files take less than 500MB on disk. Moreover, increasing the cache length requires more memory resources. The RAM usage scales smoothly than the time requirements, with a maximum value of 24.9GB (approximately 3x for cache length 10 compared to the baseline). These results indicate that cache length 7 is most likely the best preprocessing string length for this experiment. Despite its limitations, `GCSA2` can fully index small pangenome graphs, such as

Table 4.2: Index time and maximum memory result on drosophila pangenome graph.

Tool	Mode	Wall Clock (s)	Max Mem (GB)	Cache (GB)
`gindex`	Cache length 1	282	7.6	0.37
	Cache length 4	344	16.5	1.06
	Cache length 7	877	19.1	1.73
	Cache length 10	30 185	24.9	2.41
	No-Cache	251	7.5	-
`GCSA2`	-	3174	20.1	-

this one. However, as shown in Table 4.2, it is slightly less efficient than `gindex` in terms of both RAM usage and indexing time (with the sole exception of cache 10). Regarding index size on disk, `GCSA2` requires about 1.5GB, *i.e.* an approximately 3x increase with respect to `gindex` without cache and about 70% of the space necessary for `gindex` with cache length 7.

In Table 4.3, we show the results of querying 10000 randomly generated queries of different fixed lengths. `GCSA2` outperforms `gindex` for any metric in each tested scenario. We did not explore the amount of false positive matches reported by `GCSA2`.

Regarding `gindex`, increasing the query length does not lead to significant increases in RAM usage or execution time, except for some fluctuations related to the high frequency of some sub-patterns in the reads that occur across nodes in the graph. Note that short patterns can occur more frequently in the graph, which explains why, in some cases, we observe an increase in execution time for queries of length 100, due to the overhead of reporting more matches. We highlight that memory requirements increase with larger cache lengths, as more precomputed backward-intervals must be loaded into RAM.

The full-path mode output requires more time than just outputting coarse occurrences, as explained in the algorithm's complexity analysis. However, we avoid the memory overhead of the merging step, thereby reducing the memory usage. In any case, given its limitations, the full-path mode is only suitable for small graphs, such as the one considered in this experiment.

HPRC graphs Considering the experimental results on the Drosophila graph, we ran `gindex` on the HPRC graphs only in no-cache and cache length 7 modes.

Table 4.3: Query time and max memory results on drosophila pangenome graph.

Tool	Mode	Length	Wall Clock (s)	Max Mem (GB)
gindex	Cache length 1	100	128 207	4.80
		2575	117 383	4.68
		5050	117 212	4.68
		7525	126 400	4.68
		10 000	129 999	4.68
	Cache length 4	100	6871	6.58
		2575	6281	6.58
		5050	6219	6.58
		7525	5200	6.58
		10 000	7196	6.58
	Cache length 7	100	1036	9.70
		2575	1547	9.70
		5050	1576	9.70
		7525	1193	9.70
		10 000	1202	9.70
	Cache length 10	100	887	13.25
		2575	891	13.25
		5050	940	13.25
		7525	991	13.25
		10 000	1130	13.25
	No-Cache	100	56 008	1.54
		2575	65 742	1.50
		5050	64 691	1.50
		7525	61 583	1.50
		10 000	66 479	1.49
	Full-Path	100	69 195	1.24
		2575	65 525	1.24
		5050	66 636	1.24
		7525	65 831	1.24
		10 000	67 955	1.24
GCSA2	-	100	1.72	0.61
		2575	25.54	0.61
		5050	52.80	0.61
		7525	74.20	0.61
		10 000	96.58	0.61

Table 4.4: HPRC pangenome graphs indexing results.

Graph	Mode	Wall Clock (s)	Max Mem (GB)	Cache (GB)
MiniGraph	No-Cache	5580	42	-
	Cache length 7	7380	45	2.4
MiniGraph-Cactus	No-Cache	8880	104	-
	Cache length 7	15 540	183	16

This cache length is selected as a good trade-off between indexing performance and query speed-up.

Table 4.4 shows the indexing results on the two tested pangenome graphs. We were unable to index either of them using `GCSA2`, as we exceeded the available RAM (256GB) after more than 20 hours of computation time, using 64 threads.

As for the Drosophila experiments, here we consider the no-cache mode as the baseline. Using `gindex`, precomputing the cache of length 7 for the MiniGraph pangenome graph requires only a 1.3x time increase with respect to the baseline. Note that we did not record any substantial increase in memory (3GB) and disk usage (2.4GB).

We get different results for the MiniGraph-Cactus pangenome graph, where computing and storing the cache requires almost double the time and memory. In addition, the cache size on disk (16GB) is not negligible. We would highlight that both the size of the graph, *i.e.* the number of nodes and edges, and the topology of the graph, *i.e.* frequency of bubbles and repetitions in the labels, make an essential impact on the index computation.

In Table 4.5, we show querying results using 100 randomly generated queries with increasing fixed lengths. `gindex` combined with a cache length 7 yields approximately 10x querying speed-up compared to the no-cache mode, but costing approximately 4x more RAM for the MiniGraph graph and approximately 8x RAM for the MiniGraph-Cactus graph. We also tested the full-path mode, which generally behaved similarly to the no-cache mode, except for some cases involving MiniGraph-Cactus, likely due to the nature of the generated queries. Notably, in these two large graphs, the merging step appears to consume more memory than storing all the matched paths. Despite the high memory requirements of using the cache, its cost is heavily offset by the benefit in query time. We advise the user to wisely select the cache length according to the actual scenario of the experiments.

Table 4.5: HPRC graphs query results with 100 queries of various lengths.

Graph	**Mode**	**Length**	**Wall Clock (s)**	**Max Mem (GB)**
MiniGraph	Cache length 7	100	67	13.7
		2575	69	13.7
		5050	65	13.7
		7525	65	13.7
		10 000	68	13.7
	No-Cache	100	747	3.1
		2575	716	3.1
		5050	849	3.1
		7525	816	3.1
		10 000	786	3.1
	Full-Path	100	741	2.8
		2575	798	2.8
		5050	900	2.8
		7525	821	2.8
		1,0000	753	2.8
MiniGraph-Cactus	Cache length 7	100	550	84.6
		2575	466	83.2
		5050	469	83.3
		7525	524	83.3
		10 000	482	83.3
	No-Cache	100	5406	11.7
		2575	4534	11.7
		5050	4535	11.7
		7525	4558	11.7
		10 000	4506	11.7
	Full-Path	100	4563	10.7
		2575	4392	10.9
		5050	4372	10.7
		7525	5062	38.1
		10 000	4486	17.5

GIN-TONIC Thanks to the reviews of `gindex` paper, we became aware of GIN-TONIC [105], a pangenome graph approach similar to our `gindex`, which was published a month before the SEA 2025 deadline; hence, we do not have a published comparison in [32]. GIN-TONIC implements a similar approach, extending the FM-index and precomputing a cache in a more sophisticated manner than `gindex`. In any case, we tried GIN-TONIC on the Drosophila pangenome graph. Despite not taking a GFA file as input, we were able to index the graph in 840s, using 31.5GB of RAM and producing a 1.3GB disk footprint, with default parameters. We recall that, under the same settings, `gindex` indexed the graph in 282s in no-cache mode and 877s with a cache length of 7, using 7.5GB and 19.1GB of RAM, respectively, and producing disk footprints of 493MB and 2.2GB, respectively. Unfortunately, we were unable to match the queries due to segmentation faults, likely caused by the use of a non-standard input file format.

4.4 Conclusions

In this chapter, we present `gindex`, a multidollar-BWT-based graph indexing approach designed to solve the string-to-graph exact pattern-matching problem. Our index is capable of scaling to human pangenome graphs, such as those produced by the Human Pangenome Reference Consortium. To efficiently manage large graphs, we leverage a preprocessing cache constructed via dynamic programming, enabling us to "skip" the initial and most computationally demanding phases of the indexing process.

We demonstrated that on small pangenome graphs, such as the Drosophila graph, `GCSA2` outperforms `gindex` in query efficiency, albeit at the cost of greater indexing time and memory. In contrast, `gindex` can be used to index much larger pangenome graphs, such as the HPRC ones, which `GCSA2` cannot index on our current hardware setup, as it exceeds 256GB of RAM.

As future developments, we should propose a more succinct representation of the backward-interval set. In addition, we should work on improving the merging step, allowing the use of the full-path mode, *i.e.* reporting the full matching path and not just the starting node, even without considering redundant backward-intervals. Moreover, we should include the indexing of colored pangenome graphs, in which

the matches have to fall on a specific graph paths, which represent particular haplotypes or genomes. Using the classical nomenclature in graph pangenomics, we should be able to index both variation graphs (colored) and sequence graphs (uncolored). `gindex` can be extended to store the colors of the visited nodes, augmenting the stored paths in the full-path mode, and check whether the color consistency of two consecutive nodes before performing the backward-extension.

Another potential future direction is the exploration of other BWT variants for string collections, such as the multidollar-EBWT [187], to evaluate whether changing the BWT representation used in `gindex` can improve our approach in terms of query time or index space requirements.

Finally, we are interested in testing `gindex` in downstream analysis on human pangenome graphs, leveraging the lower requirements of our approach to analyze larger graphs without the limitations of `GCSA2`.

Chapter 5

Graph-Based Models for Alternative Splicing

In this chapter, we present `pantas` [34], a pangenomic approach for quantifying alternative splicing (AS) events using a pangenome graph. We recall that AS events are a fundamental biological mechanism, allowing a single gene to encode multiple proteins, and are therefore studied in many biological analyses, such as in [188]. To introduce this result, we also present `ESGq` [35], another graph-based model which leverages the notion of non-pangenome event splicing graphs for quantifying AS events in a classical non-pangenomics scenario. Note that `ESGq` still incorporates pangenomics tools such as the `VG` toolkit [20], to index and query graphs.

Although these results are not directly related to the research line presented in the previous chapter and are not explicitly focused on indexing and pattern-matching techniques, they represent practical applications of modern bioinformatics and pangenomics techniques to address common biological problems. The biological applications represent the ultimate goal of developing efficient algorithms and data structures, so it is essential to explore these scenarios. In fact, to the best of our knowledge, as of the date of its publication, `pantas` is the first attempt to leverage the pangenome graph in this field.

Comparisons against state-of-the-art tools on both simulated and real data demonstrate that `pantas` obtains results comparable to those of state-of-the-art methods based on a linear reference, while also accounting for the genetic variability of the population under study. Considering 77 AS events from a human real RT-

PCR validated dataset [189], `pantas` reported the highest number of AS events, 64 out of 77, being the best tool across the compared ones. On the other real dataset, for which we do not have a ground truth, `pantas` produces results highly comparable to `rMATS`, probably the most widely used tool in AS analysis. Hence, `pantas` could serve as a milestone for future pantranscriptomic studies on the detection and quantification of AS events.

The Alternative Splicing quantification problem can be formalized at a high level as:

Input:	One or more genomes, with the corresponding annotations, and RNAseq reads
Output:	The quantification of alternative splicing events with respect to canonical events

Publications and conferences `ESGq` [35] was presented in Tatranské Matliare, Slovakia, at the *Workshop on Bioinformatics and Computational Biology (WBCB 2023)*, part of *Information Technologies - Applications and Theory (ITAT 2023) conference* – conference proceedings published in the *CEUR Workshop Proceedings*. `pantas` [34] was published in the *PLOS Computational Biology* journal.

Chapter outline In section 5.1 we will present the state-of-the-art and the motivations that led us to develop `ESGq` and `pantas`. In section 5.2, we will discuss in detail our algorithmic contribution. In section 5.3, we will compare our implementation against single reference-based state-of-the-art tools, such as `rMATS` and `SUPPA2`. Finally, in section 5.4, we will conclude and discuss possible future developments.

5.1 Motivations and State-Of-The-Art

Alternative Splicing (AS) is a post-transcriptional regulation mechanism that contributes to isoform and protein diversity in eukaryotes. We recall that the classical AS events categories [190], already presented in Section 2.11, are: exon skipping, alternative 3' (acceptor) splice sites, alternative 5' (donor) splice sites, intron retention,

and mutually exclusive exons. Thanks to this mechanism, a single gene can code for multiple isoforms, resulting in the production of multiple proteins. This gene expression process is complex to understand, both from a biological and a combinatorial perspective. For instance, more than 95% of multi-exon human genes [191, 192], and more than 60% of multi-exon Drosophila Melanogaster genes [193] present more than one isoform. Moreover, AS is associated with various diseases, including cancer [194, 195], neurodegenerative diseases [196], and aging [197]. For these reasons, the analysis of AS events has become increasingly fundamental, and thanks to the RNA-sequencing technology (RNA-Seq), the analysis of the transcriptome and AS events is currently much faster and precise. In detail, an RNA-Seq analysis compares two conditions (for example, a canonical splicing event vs an AS event), checking the changes in terms of isoform abundances [198, 199]. Therefore, in this context, alterations in the relative abundances of an isoform of a gene are generally considered strong evidence of differential splicing. In addition, despite the challenges of establishing their impact on alternative splicing [200], we also have evidence of genetic variations altering splicing patterns in many diseases [201, 202]. Therefore, the development of pangenomic tools for studying these mechanisms is expected to become essential in the near future.

In this context, some tools are based on analyzing the transcript, while other approaches directly detect and quantify AS events. Hence, instead of quantifying entire isoforms, several methods focus on the most fine-grained level, specifically at exon-exon boundaries, also known as splice junctions. These boundaries are directly used to detect and quantify AS events from RNA-Seq datasets by tools such as [38, 203, 36, 39, 204, 205, 37] and our `ESGq` [35], providing more accurate results than transcript-based approaches like [198, 199, 206]. Another issue is that most of these tools introduced their own format for the AS events, complicating any downstream analysis.

In 2023, Sibbesen et al. [119] proposed a method for transcriptomic data analysis in the pangenomic context, enhancing the pangenome with additional gene annotation information to create a so-called spliced pangenome or pantranscriptome, thereby enabling more accurate and comprehensive analysis. This new approach allows haplotype-aware transcript quantification, improving accuracy compared to reference-based methods [207]. Accounting for the genetic variability within the

population, this new approach mitigates the reference bias problem and increases the number of reads correctly mapped in the presence of heterozygous variants.

Note that, to the best of our knowledge, `pantas` is the only work that integrates transcriptomics with graph-based pangenomics, even though the use of graphs (e.g., splicing graphs) is already common in computational transcriptomics for modeling isoform variability [208, 209, 210, 203, 36, 35]. This emerging field, known as pantranscriptomics, calls for further research and the development of dedicated tools to fully unlock its applications in biology. Hence, encouraged by the results in [119], we attempted to develop pangenomic tools for haplotype-aware detection and differentially quantification of AS events across conditions, closing the gap in the analysis of the relationship between genetic variations and alternative splicing.

5.2 Methods

In this section, we will start by presenting `ESGq` [35], a non-pangenomics graph-based tool that uses event splicing graphs for quantifying AS events across conditions. Despite not relying on a pangenome graph, `ESGq` is based on standard pangenome graph tools, such as the VG toolkit [20]. Then, we will discuss how we generalized and refined this method in the pangenomics context with `pantas`, enriching the notion of splicing pangenome graph to that of an annotated spliced pangenome graph.

5.2.1 AS quantification via event splicing graphs: `ESGq`

`ESGq` is a novel graph-based method for the differential quantification of AS events across conditions. The required inputs are the classical ones in this context: a reference genome, a gene annotation, and a two-condition RNA-Seq dataset, with optional replicates. The RNA-Seq dataset can be paired-end and single-end. As output, `ESGq` reports the differential expression of annotated AS events. As output, `ESGq` reports the differential expression of annotated AS events. This differential expression is typically quantified using two standard metrics: the Percent-Spliced In (PSI, ψ) for each input replicate, and $\Delta\psi$, which summarizes the differential expression of each event between the two conditions.

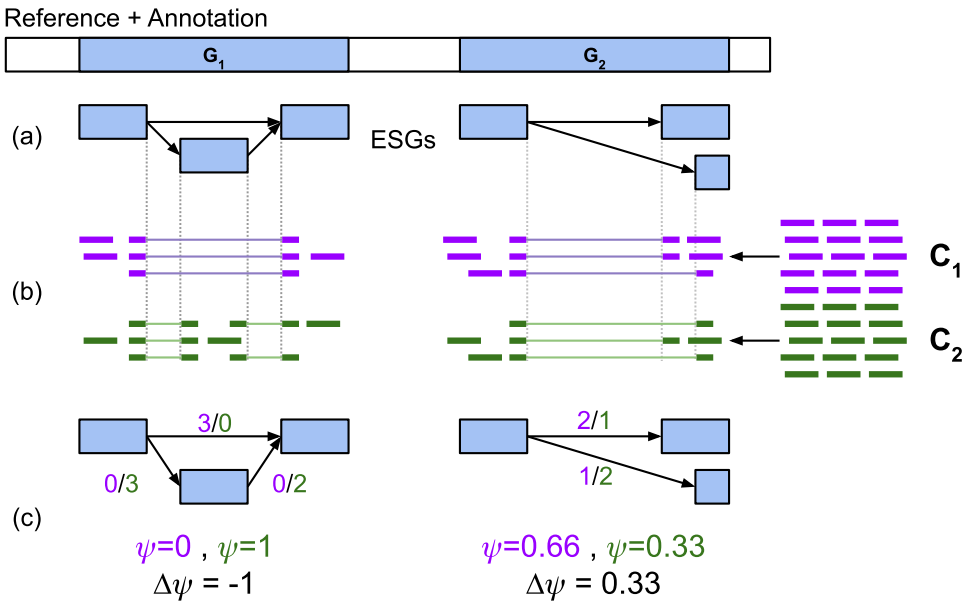


Figure 5.1: ESGq method overview: a) ESGq builds the event splicing graphs from the reference genome and the gene annotation, b) we align RNA-Seq reads from the two conditions (C_1 and C_2) to the event splicing graphs, and c) we leverage the alignments to weight junction edges to compute the ψ values (one per condition) and the $\Delta\psi$ value (one per dataset).

ESGq is focused on four categories of alternative splicing events: exon skipping (SE), intron retention (RI), alternative acceptor site (A3), and alternative donor site (A5). Moreover, unlike event-based approaches that rely on spliced read alignment or k-mer–based quasi-mapping to a reference genome, ESGq leverages string-to-graph alignments to directly represent the AS events to be quantified. Note that ESGq considers less complex graphs than a full splicing graph as in [203, 36]. In detail, ESGq uses Event Splicing Graphs (ESGs), which encode only the exons and splice junctions involved in an alternative splicing event. Hence, ESGq does not represent all known transcripts of a gene, nor the entire gene loci and intergenic regions; instead, it encodes only the portions of the two transcripts involved in an AS event, achieving high accuracy results efficiently.

As shown in Figure 5.1, ESGq consists of three steps: a) build the event splicing graphs, b) align the reads against the event splicing graphs, and c) compute the AS quantification metrics ψ and $\Delta\psi$. ESGq quantifies only annotated AS events

and does not consider novel AS events.

Build the event splicing graphs For the first step of Figure 5.1, we start by extracting the annotated alternative splicing events from the input annotation, using the same method described and implemented for `SUPPA2` [39]. This results in a list of annotated alternative splicing events, each represented as a pair consisting of its type (SE, RI, A3, A5) and the genomic coordinates of the corresponding splice junctions, with both isoforms already annotated. Analyzing this list, `ESGq` builds the event splicing graphs, one per event. We note that multiple graphs can be built from the same gene. In these graphs, the exons involved in each event are encoded as nodes, based on the genomic coordinates of the event.

For each AS event, we have two isoforms: the canonical isoform, encoded in the graph by the path $\mathcal{P}_C$, and the alternative isoform, encoded in the graph by the path $\mathcal{P}_A$. Different AS event categories require different approaches to represent the two isoforms, as in Figure 5.2.

An exon skipping event (SE) is modeled using three exons, encoded as three nodes n_1, n_2, n_3 in the event splicing graphs. Node n_2 represents the exon that is skipped during the event; thus, the canonical isoform is defined by the path $\mathcal{P}_C = n_1 \to n_2 \to n_3$, including all three nodes, while the alternative isoform skips node n_2 in the path $\mathcal{P}_A = n_1 \to n_3$.

An alternative acceptor site event (A3) is modeled using three exons, encoded as three nodes n_1, n_2, n_3 in the event splicing graphs. Node n_1 represents the canonical upstream exon, n_2 the canonical downstream exon, and n_3 the alternative downstream exon with the alternative acceptor splice site; thus, the canonical isoform is defined by the path $\mathcal{P}_C = n_1 \to n_2$, which involves the shared upstream exon and the canonical downstream exon, while the alternative isoform changes the downstream exon with the alternative one in the path $\mathcal{P}_A = n_1 \to n_3$.

An alternative donor site event (A5) is modeled using three exons, encoded as three nodes n_1, n_2, n_3 in the event splicing graphs. Node n_1 represents the canonical upstream exon, n_2 the canonical downstream exon, and n_3 the alternative upstream exon with the alternative donor splicing site; thus, the canonical isoform is defined by the path $\mathcal{P}_C = n_1 \to n_2$, which involves the shared canonical upstream exon and the shared downstream exon, while the alternative isoform changes the upstream

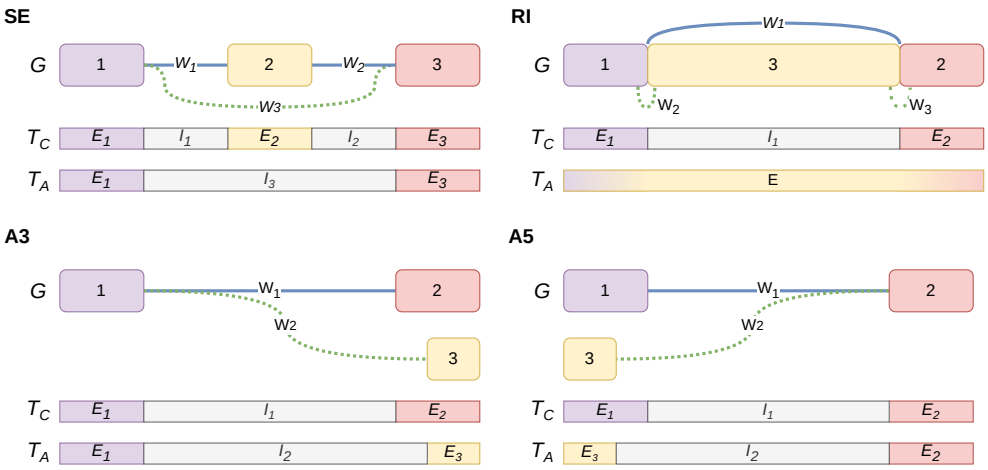


Figure 5.2: Event splicing graphs computed by `ESGq`. For each AS event type, we report the event splicing graph G, and the two annotated isoforms involved in the event: the canonical transcript T_C (represented in G by blue edges) and the alternative transcript T_A (represented in G by dotted green edges). In both transcripts, E blocks represent exons and I blocks represent introns. The edge labels W are the weights computed by the alignment step.

exon with the alternative one in the path $\mathcal{P}_A = n_3 \to n_2$.

Finally, an intron retention event (RI) is modeled using three exons, encoded as three nodes n_1, n_2, n_3 in the event splicing graphs. This case is more complicated to describe, as the three nodes n_1, n_2, n_3 do not directly correspond to three exons, except for node n_3, which corresponds to a portion of the central exon, as in Figure 5.2. More specifically, nodes n_1 and n_2 represent the two upstream and downstream exons. In contrast, node n_3 represents the retained intron, which is the internal portion of the exon linking the upstream and downstream canonical exons. Hence, the canonical isoform is defined by the path $\mathcal{P}_C = n_1 \to n_2$, involving the upstream exon and the downstream exon. On the other hand, the alternative isoform includes the retained intron in the path $\mathcal{P}_A = n_1 \to n_3 \to n_2$.

Note that, although each event theoretically contributes to a single event splicing graph, `ESGq` builds a single graph with one connected component per event. In other words, we consider a graph with multiple connected components, where each of them is an event splicing graph as described above.

Computing the event splicing graphs weights via string-to-graph alignments In the second step of Figure 5.1, `ESGq` indexes each event splicing graph to align RNA-Seq reads using the GBWT [31]-based graph aligner Giraffe [104], which is part of the `VG` toolkit [20]. Although this type of pipeline is mainly used in a pangenomic context, we recall that `ESGq` takes as input a single reference rather than a pangenome. Note that each RNA-Seq read is aligned to a path of the graph.

Next, each replicate in the RNA-Seq dataset is independently aligned to the graph using the Giraffe aligner. By default, the `VG` toolkit splits nodes longer than 32bp into smaller segments of at most 32bp before indexing. To maintain correspondence between the nodes in the `ESGq` graph and those used in the index, `ESGq` performs this node splitting during the graph construction step and links the resulting nodes accordingly, preserving the two paths that represent the isoforms. As a result, in each event splicing graph, we have two types of edges: a) edges connecting nodes that together represent a single exon, created during the node-splitting step, and b) edges representing the splice junctions between exons of AS events. `ESGq` considers just the latter for the AS events quantification, hence the other type of edges is omitted in Figure 5.2.

Computing ψ and $\Delta\psi$ measures via the event splicing graphs In the third and last step of Figure 5.1, `ESGq` computes the ψ value of each event for every replicate. It then summarizes these values by comparing the two conditions to compute the $\Delta\psi$ for each AS event, which quantifies the differential expression between the two input conditions. `ESGq` computes these two measures by assigning a weight to each splicing junction based on the RNA-Seq alignments and counting only the alignments that span edges representing splicing junctions. If a read is aligned to a single node of the graphs, *i.e.* without falling over any edge, the alignment is discarded. In other words, the weight of the splicing junction edges represents the number of RNA-Seq reads that have been spliced aligned over it. Finally, `ESGq` leverages these weights to compute the ψ value of each AS event, using the classical ψ formulation [211], hence considering the proportion of reads supporting the standard isoform over the reads supporting both isoforms. Note that `ESGq` considers only the spliced read counts for the ψ measure, while other approaches also consider non-spliced reads. Thus, the support of each isoform is

approximated by these "spliced" counts, without analyzing the full coverage of each isoform. As we will see in Section 5.3, our experimental evaluation seems to confirm the goodness of this approximation.

Referring to the variable assignments in Figure 5.2, each AS event category has the corresponding approach for computing the ψ measure. For an exon skipping event (SE), the ψ value is calculated as $\psi_{SE} = \frac{\frac{w_1+w_2}{2}}{\frac{w_1+w_2}{2}+w_3}$, where w_1, w_2 are the weight of the canonical isoform and w_3 is the weight of the alternative isoform. The ψ_{SE} value formulation is similar to [212]. For both alternative acceptor site event (A3) and alternative donor site event (A5), we have the same formulation for the ψ measure: $\psi_{A3} = \frac{w_1}{w_1+w_2}$ and $\psi_{A5} = \frac{w_1}{w_1+w_2}$, where w_1 is the weight of the canonical isoform and w_2 is the weight of the alternative isoform. Finally, for an intron retention event (RI), the ψ measure is calculated as $\psi_{RI} = \frac{w_1}{w_1+\frac{w_2+w_3}{2}}$, where w_1 is the weight of the canonical isoform and w_2, w_3 are the weights of the alternative isoform. Note that for the ψ_{RI} we use the mean of the weights of the alternative isoform.

After having calculated one ψ value per AS event per replicate, `ESGq` computes the differential quantification $\Delta\psi$ across the two input conditions. This measure is the difference between the absolute value of the ψ means in the two conditions. Note that `ESGq` does not report the p value of the $\Delta\psi$, as a consequence of considering only spliced RNA-Seq read counts.

5.2.2 AS quantification via pangenomes: `pantas`

As in the `ESGq` scenario, we considered the problem of detecting and quantifying AS events supported by an input RNA-Seq dataset comparing two conditions. The novelty in the `pantas` approach is detecting and quantifying AS events with respect to a spliced pangenome graph, *i.e.* analyze AS events in a haplotype-aware context.

The `pantas` pipeline is built around the concept of an annotated spliced pangenome, which will be described shortly. In some respects, it resembles the `ESGq` pipeline, although `ESGq` does not incorporate a pangenome graph. The workflow consists of three main steps: a) constructing the annotated spliced pangenome and enriching it with RNA-Seq read alignment results, b) analyzing the annotated spliced pangenome graph to detect AS events, and c) computing the AS

quantification metrics ψ and $\Delta\psi$.

Build and annotate the annotated spliced pangenome We begin with the pangenomic definition of a variation graph from [2], which is a node-labeled directed graph in which each node is labeled with a genomic sequence, edges indicate consecutiveness between node labels, and colored walks represent individual haplotypes, *i.e.* the genomic sequences of each input individual.

We want to highlight that, usually, the input haplotypes/genomes are split per chromosome. However, without loss of generality, we will consider haplotype walks as single walks in the graph.

A spliced pangenome graph, firstly described in [119], is a variation graph with two types of walks: a) the haplotype walks $\{H_1, H_2, \ldots, H_h\}$ which represent the haplotypes as in a classical variation graph, and the transcript walks $\{T_1, T_2, \ldots, T_t\}$ which encode the haplotype-aware transcripts. In detail, each haplotype-aware transcript is a sequence of exons, *i.e.* the coding regions of a gene, that belongs to a specific haplotype in $\{H_1, \ldots, H_h\}$. Note that, as a consequence of the haplotype similarities, two haplotype-aware transcripts can fall on the same transcript walk in the graph. These graphs can present cycles, but in `pantas` we consider only acyclic graphs.

In a spliced pangenome, if a node belongs to at least one haplotype-aware transcript walk, it represents an exonic/coding region. On the contrary, all other nodes represent intronic/non-coding regions, or intergenic regions that belong to some haplotypes. If an edge belongs to some transcript walks $\{T_1, T_2, \ldots, T_t\}$ but not to any haplotype walks $\{H_1, H_2, \ldots, H_h\}$, it is denoted as an annotated spliced junction. These edges, in fact, connect two nodes that are not consecutive in the haplotype walk, spanning an intronic or non-coding region between the exons corresponding to the two nodes.

Given these preliminary definitions, we define an annotated spliced pangenome as a spliced pangenome where nodes and links are annotated with additional information necessary to detect AS events. Refer to Figure 5.3 for an example of an annotated spliced pangenome. In detail, each node v of an annotated spliced pangenome is annotated with: a) the set $\mathcal{E}(v)$ of its exon labels, and b) the set $\mathcal{T}(v)$ of its haplotype-aware transcripts. We will refer to the set $\mathcal{E}(v)$ simply as exons.

Note that an exon is uniquely identified by: a) the haplotype-aware transcript to which it belongs, and b) its position order within that transcript. In an annotated spliced pangenome, a single exonic region can be represented by one or more nodes, potentially originating from different haplotype-aware transcripts, either due to haplotype similarity or the presence of an alternative exon form caused by an alternative splicing event. As a result, $\mathcal{E}(v)$ represents a set of exons rather than a single exon.

For simplicity, if $\mathcal{E}(v)$ or $\mathcal{T}(v)$ are empty, we say that node v has no annotation. This occurs, for example, when considering a node in an intronic or intergenic region. Moreover, in an annotated spliced pangenome, each edge $e = \langle u, v \rangle$ that represents a splice junction is annotated with the set $\mathcal{T}(e)$, which stores the haplotype-aware transcripts the edge belongs to. As for nodes, many haplotype-aware transcripts can share the same splice junction.

All these data are required to fully represent the known haplotype-aware transcripts encoded in a spliced pangenome, but we can also annotate additional information. For example, each edge $e = \langle u, v \rangle$ can be annotated with a numeric value $w(e)$ that reflects the level of support or coverage by RNA-Seq reads. This can be estimated by counting the number of reads aligned across that edge.

Alternative splicing event detection We will now formally describe how we detect AS events in `pantas`. To detect AS events using the annotated spliced pangenome, we need to compare the splice junctions of two transcripts locally. Thus `pantas` analyzes two sets of junctions: a) the set of junctions coming from the canonical transcript, and b) the set of junctions coming from the alternative transcript. The AS event detection starts from the alternative junctions and compares them to the canonical ones. We say that an AS event is annotated if the junctions of the two transcripts involved in the AS event are already annotated. In other words, an AS event is annotated if the graph already contains the annotated edges representing those junctions. On the contrary, a novel AS event occurs if two conditions are met: a) the splice junctions of only one of the two transcripts are already annotated in the annotated spliced pangenome, and b) the junctions of the other transcript are supported by read alignments only.

Based on this annotated spliced pangenome graph, `pantas` models AS events

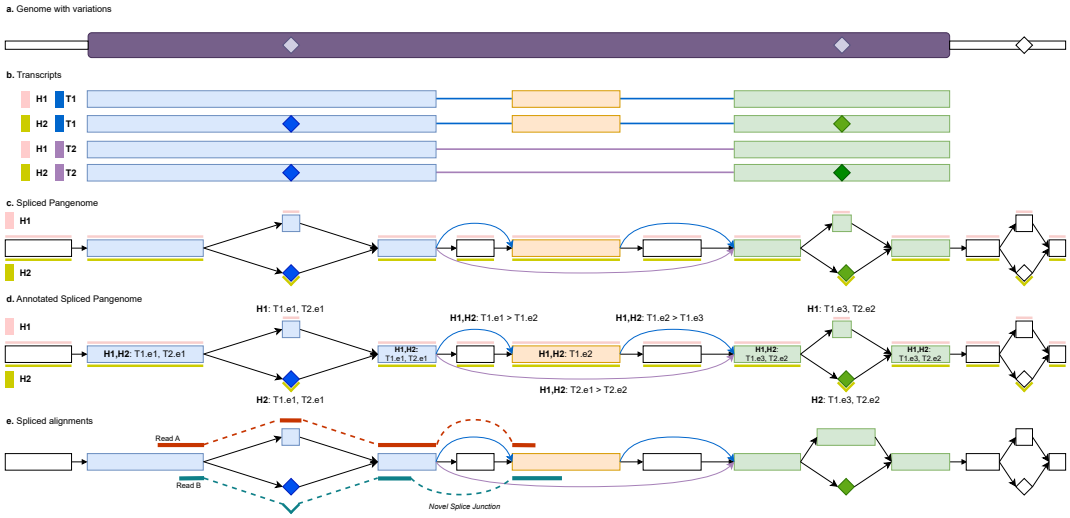


Figure 5.3: Annotated spliced pangenome example. In subfigure (a), the reference genome is shown with a purple box marking a gene locus and diamonds denoting variant alleles. Two haplotypes are considered: H1 for the reference alleles and H2 for the alternate alleles. In subfigure (b), we show the gene with two transcripts, T1 and T2, expressing an exon-skipping event, resulting in four haplotype-aware transcripts in total. Alternate alleles are colored according to the exon in which they fall. In subfigure (c), we show the spliced pangenome constructed from the reference genome, gene annotation, and variants. Colored nodes and edges encode transcript walks, representing portions of exons and splice junctions, while white nodes correspond to intronic or intergenic regions not included in transcript walks. H1 and H2 are represented as colored bars above and below the nodes. In subfigure (d), we show the annotated spliced pangenome, where each colored node, corresponding to an exon segment, is annotated with the transcripts and exons to which it belongs. In this annotated spliced pangenome, each colored edge, *i.e.* each junction, is annotated with splice junction information. In subfigure (e), two RNA-Seq reads are aligned to the annotated spliced pangenome: read A aligns to the reference allele of the first variant, supporting the annotated splice junction between the first and second exons, represented by the blue edge, while read B aligns to the alternate allele, supporting a novel splice junction between the two exons, inducing an alternative donor event.

directly on the graph structure, leveraging the graph topology and its annotations to define each event formally. This approach does not rely on the coordinates of a single linear reference genome, unlike other methods, including `ESGq`.

We will describe separately how `pantas` detects annotated AS events and how it detects novel AS events. As annotated AS events, we consider the same categories already discussed for `ESGq`: exon skipping, intron retention, alternative 3' splicing/acceptor site, and alternative 5' splicing/donor site. As novel AS events, we consider the same four categories when they are not already annotated, as well as cases in which those categories of AS events occur in intronic regions.

At the end of the detection step, `pantas` represents each AS event as a pair of sets of edges. These sets represent the two junction sets, one coming from the canonical isoform and one from the alternative isoform involved in the event.

Annotated AS events detection To detect annotated AS events, as summarized in Figure 5.4, we need to analyze each annotated splice junction independently. Each annotated junction can potentially identify an AS event if certain event-type-dependent conditions are met.

Formally, we consider a splice junction $e = \langle a, b \rangle$ between two exonic regions having a transcript $T_1 \in \mathcal{T}(a) \cap \mathcal{T}(b)$. Assuming to have annotated the junction coverage $w(e)$, we discard junctions with $w(e)$ smaller than a certain threshold. The user chooses this threshold to discard AS events without enough evidence. Moreover, given a node v, we denote with $next(v)$ and $prev(v)$ the set of successors and predecessors of the node v in the annotated spliced pangenome graph, respectively.

An exon skipping event for the annotated splice junction $j = \langle a, b \rangle$ is called if there is a transcript T_2 of the same gene with an exon between nodes a and b. Note that $T_1 \neq T_2$ since T_1 uses $\langle a, b \rangle$, not having any exon between a and b. Formally, there exists a transcript $T_2 \in \mathcal{T}(a) \cap \mathcal{T}(b) \setminus \mathcal{T}(j)$ with two exons $e_a \in \mathcal{E}(a)$ and $e_b \in \mathcal{E}(b)$ that are not consecutive in t. In Figure 5.4.a, we show an example of annotated exon skipping event detection with `pantas` in which $a = n_1$, $b = n_4$, and where *T1.e2* is the skipped exon for the transcript *T2*.

An annotated alternative 5' splicing/donor site event for the annotated junction $j = \langle a, b \rangle$ is called if there is another transcript of the same gene for which the first exon extends further towards the 3' without incorporating the second

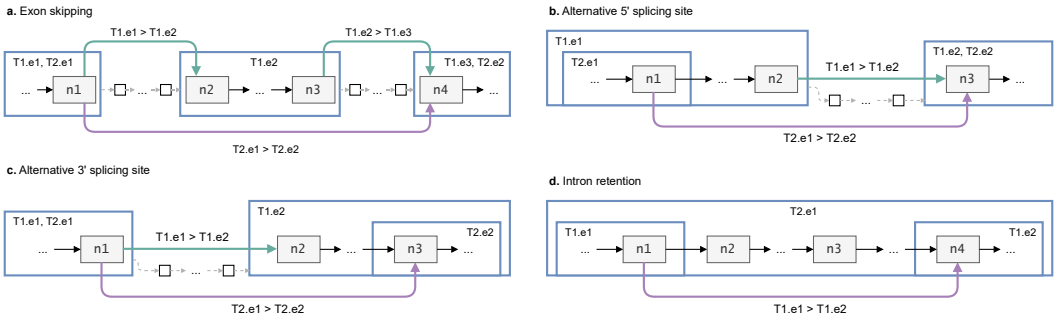


Figure 5.4: All the annotated events expressed within the annotated spliced pangenome, together with the different tags used. Haplotype information and weights are omitted for readability. Blue squares represent exons with their tags, while green and purple edges correspond to annotated splice junctions with their tags. Nodes with a gray border represent exonic regions, while nodes with a black border represent intronic regions. Exons are labeled according to the transcript walks represented in the figure.

exon of the junction j. Formally, there exists a node $v \in \mathit{next}(a)$ and an exon $e \in \mathcal{E}(a) \cap \mathcal{E}(v) \setminus \mathcal{E}(b)$, *i.e.* the exon e covers both the node a and one of its successors v without covering node b, such that $\mathcal{T}(e) \cap \mathcal{T}(b) \neq \emptyset$. It follows by construction that there is a transcript $T_2 \in \mathcal{T}(e)$, $T_1 \neq T_2$ which does not cover the junction j but covers both exons connected by j. In Figure 5.4.b, we show an example of annotated alternative 5' splicing/donor site event detection with `pantas` in which $a = n_1$, $b = n_3$, and there are nodes $(n_1, ..., n_2]$ which extend "to the right" exonic nodes of *T1.e1* but not of *T2.e1*.

An annotated alternative 3' splicing/acceptor site event for the annotated junction $j = \langle a, b \rangle$ is called symmetrically in respect to how we detect an annotated alternative 5' splicing/donor site event. In detail, we identify a predecessor $v \in \mathit{prev}(b)$ and an exon $e \in \mathcal{E}(b) \cap \mathcal{E}(v) \setminus \mathcal{E}(a)$ such that $\mathcal{T}(e) \cap \mathcal{T}(b) \neq \emptyset$. In Figure 5.4.c, we show an example of annotated alternative 3' splicing/acceptor site event detection with `pantas` in which $a = n_1$, $b = n_3$, and there are nodes $[n_2, ..., n_3)$ which extend "to the left" exonic nodes of *T1.e2* but not of *T2.e2*.

Finally, an annotated intron retention event for the annotated junction $j = \langle a, b \rangle$ is called if there is another transcript of the same gene for which a, b are in the same exon, having that there is a third exon that spans both exon a and b and

including the intron in between them. Formally, there exist two edges $\langle a, u\rangle$ and $\langle v, b\rangle$, with u and v not necessarily distinct, such that $\mathcal{E}(a) \cap \mathcal{E}(b) \cap \mathcal{E}(u) \cap \mathcal{E}(v) \neq \emptyset$. In Figure 5.4.d, we show an example of annotated intron retention event detection with `pantas` in which $a = n_1$, $b = n_4$, which are part of exons *e1* and *e2* for transcript *T1* and part of just *e1* for *T2*, and n_2 and n_3 are intronic for *T2*.

Novel AS events detection We now describe how `pantas` detects novel AS events, as summarized in Figure 5.5. In this description, we consider an edge $e = \langle a, b\rangle$ between two nodes such that $\mathcal{T}(a) \cap \mathcal{T}(b)$ is not empty, except for the alternative 3'/acceptor site and 5'/donor site intronic cases. This edge e can be defined as a novel junction if it is not already annotated in the annotated spliced pangenome graph. We leverage these novel junctions to classify novel AS events. As in the case of annotated AS events, we discard novel junctions with $w(e)$ smaller than a certain user-specified threshold, recalling that $w(e)$ is a weight assigned to the edge e by the RNA-Seq alignment coverage/support.

For any definition that requires covering a range of nodes $[x, \ldots, y]$, we do not always perform a complete graph traversal. Instead, we iteratively check whether there exists a covered node $x_{i+1} \in \mathit{next}(x_i)$ and a covered node $y_{i+1} \in \mathit{prev}(y_i)$ for $i < w$ (using a user-specified window of length w), or until the sets of nodes converge. In addition, note that the following formulations are based on both novel and annotated junctions. This heuristic is less robust than an actual graph traversal, but it is computationally efficient. Our experimental results, which will be described in Section 5.3, seem to prove the goodness of our approach.

A novel exon skipping event for the novel splicing junction $j = \langle a, b\rangle$ is called if there is a transcript of the same gene for which none of the exons in the set $\mathcal{E}(a)$ and in the set $\mathcal{E}(b)$ are consecutive. Formally, there is at least one transcript in $\mathcal{T}(a) \cap \mathcal{T}(b)$ in which exons from $\mathcal{E}(a)$ and $\mathcal{E}(b)$ are not consecutive. In Figure 5.5.a, we show an example of a novel exon skipping event in `pantas` in which $a = n_1$, $b = n_4$, which are part of exons *e1* and *e3* for transcript *T1*, resulting in one exon skipped.

A novel cassette exon event (witch is an exon skipping where the novel and non-annotated exons falls in an intronic region) for the annotated junction $j = \langle a, b\rangle$ is called if there are an exon from $\mathcal{E}(a)$ and one from $\mathcal{E}(b)$ that are consecutive

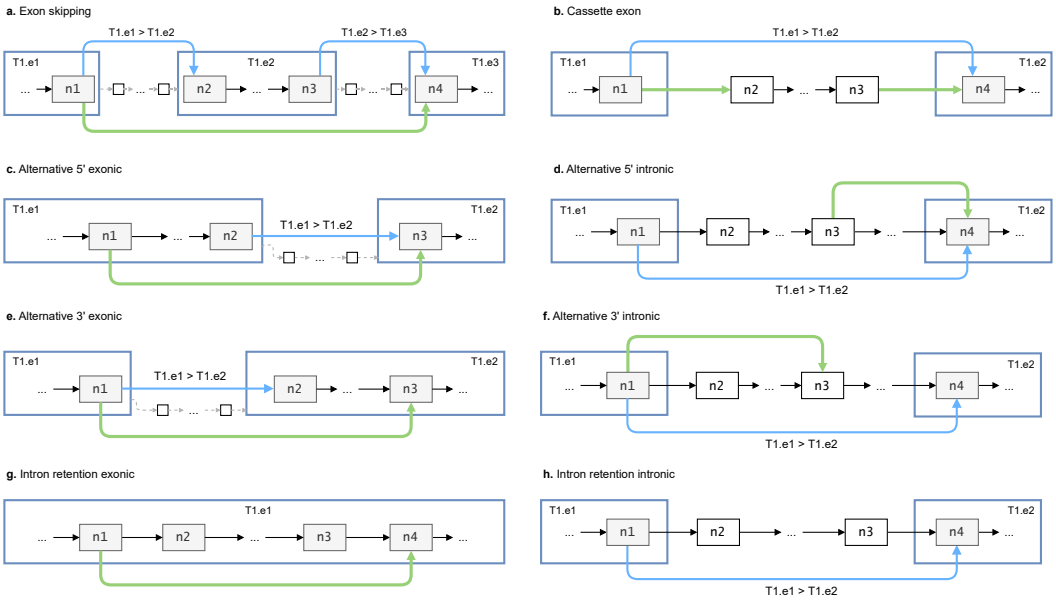


Figure 5.5: All the novel events expressed within the annotated spliced pangenome, together with the different tags used. Haplotype information and weights are omitted for readability. Blue squares represent exons with their tags, and blue edges correspond to annotated junctions with their tags; green edges indicate novel links. Nodes with a gray border represent exonic regions, while nodes with a black border represent intronic regions.

for a transcript, and there are two novel covered edges $\langle a, x\rangle$, $\langle y, b\rangle$. Formally, it is required that: a) there exist two exons $e_a \in \mathcal{E}(a), e_b \in \mathcal{E}(b)$ such that e_a, e_b are consecutive for at least one transcript in $\mathcal{T}(a) \cap \mathcal{T}(b)$, and b) there are two novel covered junctions $\langle a, x\rangle$, $\langle y, b\rangle$. In Figure 5.5.b, we show an example of a novel cassette exon event in `pantas` in which $a = n_1$, $b = n_4$ are in two consecutive exons and $x = n_2$, $y = n_3$ are the intronic covered nodes in between.

A novel alternative 5' exonic splicing/donor site event for the novel junction $j = \langle a, b\rangle$ is called if, given an exon from the set $\mathcal{E}(a)$ and an exon from the set $\mathcal{E}(b)$ that are consecutive for a transcript, there is a transcript of the same gene for which the first exon extends further towards the 3'. Formally, it is required that: a) there exist two exons $e_a \in \mathcal{E}(a), e_b \in \mathcal{E}(b)$ such that e_a, e_b are consecutive for at least one transcript in $\mathcal{T}(a) \cap \mathcal{T}(b)$, and b) $\mathcal{E}(a) \subseteq \cup_{v \in next(a)} \mathcal{E}(v)$. In Figure 5.5.c, we show an example of a novel alternative 5' exonic splicing/donor site event in `pantas` in which $a = n_1$, $b = n_3$, and there are nodes in $(n_1, ..., n_2]$ which extend "to the right" exonic nodes of *T1.e1*.

A novel alternative 5' intronic splicing/donor site event for the novel junction $j = \langle a, b\rangle$ is called if, being a intronic and $\mathcal{E}(b)$ exonic, there is an annotated junction $\langle x, b\rangle$ such that one exon from the set $\mathcal{E}(x)$ and one exon from the set $\mathcal{E}(b)$ are consecutive for a transcript, and such that the nodes $[x, ..., a]$ are covered. Formally, it is required that: a) $\mathcal{T}(a) \cap \mathcal{T}(b)$ is empty, b) there exists an annotated junction $\langle x, b\rangle$, c) $I = \mathcal{T}(x) \cap \mathcal{T}(b)$ is not empty, d) there exist two exons $e_a \in \mathcal{E}(a), e_b \in \mathcal{E}(b)$ such that e_a, e_b are consecutive for at least one transcript in I, and e) each node in $[x, ..., a]$ is covered. In Figure 5.5.d, we show an example of a novel alternative 5' intronic splicing/donor site event in `pantas` in which $a = n_3$, $b = n_4$, $x = n_1$, and the nodes $[n_2, ..., n_3]$ need to be covered.

A novel alternative 3' exonic splicing/acceptor site event for the novel junction $j = \langle a, b\rangle$ is called if, given an exon from the set $\mathcal{E}(a)$ and an exon from the set $\mathcal{E}(b)$ that are consecutive for a transcript, there is a transcript of the same gene for which the second exon extends further towards the 5'. Formally, it is required that: a) there exist two exons $e_a \in \mathcal{E}(a), e_b \in \mathcal{E}(b)$ such that e_a, e_b are consecutive for at least one transcript in $\mathcal{T}(a) \cap \mathcal{T}(b)$, and b) $\mathcal{E}(b) \subseteq \cup_{v \in prev(b)} \mathcal{E}(v)$. In Figure 5.5.e, we show an example of a novel alternative 3' exonic splicing/acceptor site event in `pantas` in which $a = n_1$, $b = n_3$ and there are nodes in $[n_2, ..., n_3)$ which extend

"to the left" exonic nodes of *T1.e2*.

A novel alternative 3' intronic splicing/acceptor site event for the novel junction $j = \langle a, b \rangle$ is called if, being b intronic and $\mathcal{E}(a)$ exonic, there is an annotated junction $\langle a, x \rangle$ such that one exon from the set $\mathcal{E}(x)$ and one exon from the set $\mathcal{E}(a)$ are consecutive for a transcript, and such that the nodes $[b, ..., x]$ are covered. Formally, it is required that: a) $\mathcal{T}(a) \cap \mathcal{T}(b)$ is empty, b) there exists an annotated junction $\langle a, x \rangle$, c) $I = \mathcal{T}(a) \cap \mathcal{T}(x)$ is not empty, d) there exist two exons $e_a \in \mathcal{E}(a), e_b \in \mathcal{E}(b)$ such that e_a, e_b are consecutive for at least one transcript in I, and e) each node in $[b, ..., x]$ is covered. In Figure 5.5.f, we show an example of a novel alternative 3' intronic splicing/acceptor site event in `pantas` in which $a = n_1$, $b = n_3$, $x = n_4$, and the nodes $[n_3, ..., n_4]$ need to be covered.

A novel exonic intron retention event for the novel junction $j = \langle a, b \rangle$ is called if, belonging a and b to the same exon, there is at least one node in $prev(a)$ and at least one node in $next(b)$ that are in the same exon on which a and b fall. Formally, it is required that: a) $\mathcal{E}(a) \cap \mathcal{E}(b)$ is not empty, b) $\mathcal{E}(a) \cap \mathcal{E}(b) \cap \cup_{v \in next(a)} \mathcal{E}(v) \cap \cup_{e \in prev(b)} \mathcal{E}(v)$ is not empty. In Figure 5.5.g, we show an example of a novel exonic intron retention event in `pantas` in which $a = n_1$, $b = n_4$, which are part of the exon *e1* for transcript *T1*, and n_2, n_3 are nodes belonging to the same exon.

Finally, a novel intronic intron retention event for the covered annotated junction $j = \langle a, b \rangle$ is called for each transcript in $\mathcal{T}(a) \cap \mathcal{T}(b)$, calling one intronic intron retention event per transcript, if the nodes $[a, ..., b]$ are sufficiently covered. In Figure 5.5.h, we show an example of an intronic intron retention event in `pantas` in which $a = n_1$ and $b = n_4$ are linked by the annotated junction, and $[n_2, ..., n_3]$ are the intronic nodes.

We must make additional considerations when detecting novel AS events. In cases such as novel alternative 5' exonic splicing/donor site events, a new splice junction may start or end within a node. To correctly represent these cases, we split the affected node into two separate nodes, ensuring that junction endpoints align with node boundaries and allowing precise localization of the corresponding novel AS events.

Computing ψ and $\Delta\psi$ measures via the annotated spliced pangenome After detecting AS events in each replicate, `pantas` quantifies them by combining

the results across replicates. The quantification is based on the analysis of the support of the junctions in each replicate, computing the ψ measure leveraging the ratio between the support of the canonical isoform and the alternative isoform [213], averaging across the replicates. Formally, for the AS event type ε, given ν replicates, χ_i as the canonical isoform coverage for the i-th replicate and α_i as the alternative isoform coverage for the i-th replicate, the ψ is computed as $\psi_\varepsilon = \frac{1}{\nu}\sum_{i=1}^{\nu}\frac{\chi_i}{\chi_i+\alpha_i}$. Then, `pantas` reports the differential quantification of the AS events supported by both conditions by the $\Delta\psi$ measure of each AS event, which is computed as the difference between the absolute value of the ψ means in the two conditions.

Final considerations on the `pantas` method As already mentioned, one of the novelties of `pantas` is being haplotype-aware. When we align a read to the annotated spliced pangenome, we can align it to any node of the graph, including the alternative alleles, even those not initially annotated. Moreover, a read can be aligned over the alternate allele of a variant, allowing it to support novel splice sites by introducing edges not present in the graph (as illustrated by the alignment of read B in Figure 5.3). This approach improves the accuracy of AS event detection, even when the variant is near a splice junction.

Additionally, to simplify the downstream analysis, `pantas` can also surject the positions of the splice junctions involved in annotated AS events back to a reference haplotype. This is done by reporting the positions of the splice junctions with respect to the selected walk. This approach works well when the junctions are already annotated.

For novel events, `pantas` can surject only the junctions derived from annotated isoforms, while novel junctions are not surjected. Therefore, novel AS events, which typically involve new splice sites and an alignment-induced splice junction that can split a graph node into two parts, must be analyzed and handled differently. However, by leveraging the information stored during the augmentation of the annotated spliced pangenome, the correct breakpoints can be precisely identified. The lack of surjection for novel junctions does not appear to impact the overall results of `pantas`, since having at least one junction surjected to the reference haplotype is sufficient for inspecting the genomic locus with state-of-the-art tools such as the Integrative Genomics Viewer [214].

5.3 Results

In this section, we will discuss the experimental results of both `ESGq` [35] and `pantas` [34].

The `ESGq` pipeline is implemented in Python, and the source code is publicly available at `https://github.com/AlgoLab/ESGq`. For graph indexing and string-to-graph alignments, we rely on the VG toolkit [20], which is implemented in C++. The `pantas` toolkit is implemented in Python and C, and the source code is publicly available at `http://github.com/algolab/pantas`. Annotated spliced pangenome graphs generated with `pantas` using a real dataset are available at `https://zenodo.org/records/13740102`. For both experiments, we used a Snakemake pipeline [164] to ensure the reproducibility and replicability of our results.

All the experiments were performed on a 64bit Linux (Kernel 5.15.0) system (Ubuntu 20.04.4 LTS) equipped with two 16-core AMD EPYC 7301 2.2GHz processors and 128GB of RAM.

5.3.1 Implementation detail of `pantas`

The implementation of `ESGq` follows straightforwardly the approach described in Section 5.2 and Figure 5.1, taking as input a reference in FASTA format, the gene annotations in GTF format, and the RNA-Seq reads in FASTA/FASTQ format. Meanwhile, since `pantas` is a complex pipeline, its implementation should be discussed in detail.

As input, `pantas` takes a spliced pangenome in GFA format and the RNA-Seq spliced alignments to this graph in GAF format. As these inputs are not always readily available, their preparation is included in the Snakemake pipeline [164]. The pipeline is based on the `VG` toolkit [20], which is the only toolkit that provides a spliced aligner for pangenome graphs. In any case, the annotation and quantification of `pantas` is independent of the aligner tool (despite the use of the standard GAF file format for the alignments) and can be adapted to other approaches in the future. Regarding alignments, we discard alignments with score ≥ 20, resulting in more robust coverages for splice junctions. Moreover, we discard any splice junction not

supported by at least 3 alignments. This threshold can be changed at execution time, but a value of 3 resulted in a good trade-off between efficiency and accuracy.

To fully control each `pantas` construction step and correctly annotate the graph, building the annotated spliced pangenome as described in Section 5.2, we choose to construct and index the graph manually using the `VG index` utility instead of the `VG autoindex` utility. In fact, this `autoindex` utility does not preserve transcript and splice junction information, which are needed for correctly detecting AS events. Following the recommendation of `vg` authors, our procedure begins by chunking the input files (reference genome, gene annotation, and variations) by chromosome. Hence, we build one graph per chromosome using `vg construct` to generate a pangenome from the reference chromosome and its associated variations. Finally, this graph is augmented by the `vg rna` utility, which provides transcript and splice junction information.

The resulting spliced pangenome contains new edges representing the annotated splice junctions, but does not yet include the haplotype-specific transcripts. As suggested by `vg` authors, to not make the construction too inefficient, it is more convenient to first build the spliced pangenome, then extract the complete haplotypes set from the graph using `vg gbwt`, and finally rerun `vg rna` to project the reference transcripts onto the haplotypes, thereby obtaining the set of haplotype-aware transcripts. All these steps are fully included in `pantas`.

The next step is to simplify this spliced pangenome with haplotype-aware transcripts, removing the more complex regions to make graph indexing feasible. In `pantas`, three levels of simplification steps are included, and the user has to select them depending on the specific use case. The default one is pruning only the intergenic and intronic regions of the graph, including all haplotype-aware transcripts, and then using the `vg prune` utility with the `--restore-paths` option to restore all paths, *i.e.* the transcripts, in the graph. This level, which retains all haplotype-aware transcripts, provides the most accurate results; however, it is also the most computationally expensive due to the cost of graph indexing. The second level is a more aggressive one. In this case, we discard all the transcripts that are not the reference transcripts. To perform this, we include the reference transcripts in the graph and then use the `vg prune` utility to simplify the exonic portions of the genes if they are too complex. This level, which provides the more aggressive

simplification, produces a graph that is easier to index, but results in lower precision in calling haplotype-aware events, as it has removed most of the haplotype-aware transcripts. The third and final simplification level involves removing all intergenic regions, retaining only the genic loci, and constructing a graph consisting of connected components, each representing a single gene locus. Note that a gene locus can potentially include multiple genes in the case of overlapping genes. This last level builds the smallest graph while maintaining complete information on haplotype-aware transcripts. It is useful when the user is interested in analyzing a small panel of genes coming from real datasets.

After the simplifying step, we have the annotating step. In this step, `pantas` iterates over the retained haplotype-aware transcripts and tags each node belonging to an exon with both the name of the transcript it belongs to and its exon number along that transcript. In contrast, each edge representing a splice junction is tagged with the numbers of the two linked exons and the corresponding haplotype-aware transcript. If during this transcript scanning an edge is not present in the corresponding haplotype, it is tagged by definition as a spliced junction. Multiple transcripts can share nodes and junctions, and therefore, nodes and junctions are annotated with various tags; hence, we store sets for these tags, which are fundamental for inferring AS events.

After all these steps, the chromosome-level annotated spliced pangenome graphs are merged, maintaining the same node IDs space. This graph is indexed using the `vg index` utility, which builds the `GCSA2` index [30, 31].

All these steps, *i.e.* the spliced pangenome construction and indexing, are performed once, and the index can be loaded each time we want to quantify AS events using an RNA-Seq dataset. In fact, the RNA-Seq dataset can be aligned to the indexed graph using the `mpmap` aligner [119], which is already included in the `VG` toolkit. This aligner was explicitly developed to align RNA-Seq to spliced pangenomes, being able to align over novel splice junctions, allowing `pantas` to quantify novel AS events.

Table 5.1: Drosophila Melonogaster real dataset (SRA BioProject ID: `PRJNA718442`) data, including for each paired-end replicate the IDs, the number of reads, and the disk usage.

Condition	Replicate	n.Pairs	Size (GB)
Day 1	SRR14101759	26 658 610	19.2
	SRR14101760	25 474 257	18.4
	SRR14101761	28 339 185	22
Day 60	SRR14101762	24 985 317	18
	SRR14101763	25 569 084	18.6
	SRR14101764	24 605 265	17.8

5.3.2 Experimental datasets

We selected both simulated and real datasets to evaluate `ESGq` and `pantas`. Note that not all datasets are used to assess both methods.

Drosophila Melonogaster real dataset We tested a recent [215] real dataset of RNA-Seq reads (SRA BioProject ID: `PRJNA718442`) on the correlation between aging and differential gene expression in Drosophila Melonogaster.

For this dataset, in `ESGq`, we considered the reference, gene annotation, and transcripts provided by FlyBase [216], release 6.51. For `pantas`, we also considered indels from the Drosophila Genetic Reference Panel (v2) [183].

The genome-wide differential expression analysis is conducted at two different time points: day 1 and day 60. In this dataset, we have three replicates for each of the two conditions, for a total of 6 Illumina HiSeq samples, with paired-end reads of 151bp for each sample. These conditions are studied from the perspective of differential quantification of alternative splicing events. Table 5.1 summarizes some statistics for this dataset.

Drosophila Melonogaster simulated dataset We simulated an RNA-Seq dataset using `asimulator` [217], which simulates RNA-Seq datasets while introducing AS events. We simulated this dataset from the Drosophila Genetic Reference Panel (v2) [183] and gene annotation (FlyBase 6.51 [216]), the same real dataset described above. To simplify the simulation and facilitate the use of `asimulator`,

we removed all overlapping genes from the gene annotation and the corresponding SNPs. Moreover, we consider a minimum count and frequency for the non-reference allele of 1 and 0.01, respectively. At the end of this filtering process, we are left with a reference panel consisting of 2 311 679 SNPs and 205 samples.

Then, we extracted a random sample from the reference panel, creating the genomic sequences of both its haplotypes with `bcftools consensus` [130]. From these two haplotypes, we simulated an RNA-Seq dataset with two conditions and one replicate using `asimulator` [217]. In detail, this RNA-Seq dataset contains 25 000 000 reads, where we simulated exon skipping, alternative 3' splicing/acceptor site, alternative 5' splicing/donor site, and intron retention events, with at most one event per gene. In detail, `asimulator` selects a "template" transcript from the set of all transcripts, introduces AS events by modifying this transcript, and generates additional alternate transcripts, reporting the corresponding new gene annotations. Note that `asimulator` may merge different transcripts in a single template transcript, thus creating chimeric transcripts not present in the original input gene annotation. These new annotations contain all transcripts used to simulate reads, including both the template transcript and alternate transcripts. Additionally, we produce annotations using only the template transcript, which are then used to simulate an edge case annotation and test the accuracy of calling novel AS events. Given the noise present in the truth created by `asimulator`, we retained only AS events with $|\Delta\psi| \geq 0.05$, discarding events with very low differential expression.

Human real RT-PCR validated dataset The last dataset considered is a real human RNA-Seq dataset with RT-PCR validated AS events as ground truth. We considered the RNA-Seq dataset provided by [189] (SRA BioProject ID: `PRJNA255099`), as other recent results in literature [39, 36, 37]. In this RNA-Seq dataset of 101bp-long reads, we have three replicates for two conditions: the control condition, namely TRA2A, and the double knockdown of the TRA2 splicing regulatory proteins, namely TRA2B. In addition, we considered the human genome and gene annotation from Ensembl [218] (release 109) and the 1000 Human Project phase 3 VCF files [5]. We analyzed 128 random samples from the `EUR` superpopulation, and the corresponding 256 haplotypes are encoded in the pangenome graph,

resulting in a graph that is almost three times the number of haplotypes provided by the Human Pangenome Reference Consortium [33]. We also encode all SNPs and indels with at least one alternate allele expressed in the considered population.

To quantify the RT-PCR validated AS events, we only require a small panel of genes, consisting of 77 RT-PCR-validated genes. In fact, the original RT-PCR truthset provided by [39] consists of 83 events on 82 genes, reported on the `hg19` reference genome. In our experiments, we analyzed the newer `hg38` reference genome; hence, we used the `liftover` utility[1], which is based on [219], to convert genomic coordinates from the `hg19` to the `hg38`. The remapped coordinates of three of the 83 events disagreed with the gene annotation due to a change in the start/end position of the skipped exons; therefore, they were filtered out. Moreover, from the remaining 80 RT-PCR validated AS events, we removed three events with a low differential change between the two conditions, *i.e.* $|\Delta\psi| < 0.05$. At the end of these two filtering steps, we obtained an RT-PCR validated AS events ground truth with 77 AS events.

5.3.3 Comparison tools

In this subsection, we briefly describe the state-of-the-art tools used as a comparison in our experimental setup.

`rMATS` and `STAR` `rMATS` (replicate Multivariate Analysis of Transcript Splicing) [38] is a statistical model for the differential quantification of alternative splicing events. `rMATS` is a reference-based tool that leverages read alignment to the reference genome. These alignments are provided by the `STAR` (Spliced Transcripts Alignment to a Reference) aligner [114], which is a seed-and-extend string-to-string pairwise aligner based on the suffix array. `rMATS` is capable of detecting and quantifying novel AS events.

`SUPPA2` and `Salmon` `SUPPA2` [39] is a method for the differential quantification of alternative splicing events based on the quasi-mapping of transcripts and not on alignments against a reference genome. In detail, this method is based on leveraging

[1] `https://liftover.broadinstitute.org/`

the abundances of each transcript, using the transcript quantification provided by `Salmon` [220], to compute the AS events ψ. `SUPPA2` is not capable of detecting and quantifying novel AS events.

`whippet` `whippet` [37] is a tool for quantifying alternative splicing. This method analyzes alignments to a custom graph, called Contiguous Splice Graph, which represents the transcripts, encoding all non-overlapping exons coming from the gene annotation. Note that `whippet` is used only for the evaluation of `pantas`.

`whippet` allows for augmenting the Contiguous Splice Graph with novel splice sites supported by read alignments; however, we were unable to run this augmentation successfully as `whippet` crashed when we attempted to include novel splice sites from a BAM file. Without novel splice sites, `whippet` can only report novel exon skipping events, hence we excluded it from any comparison involving novel AS events.

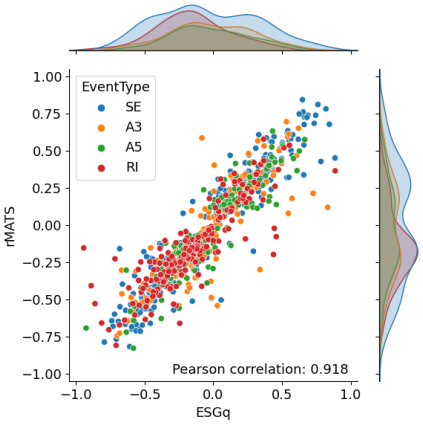
5.3.4 `ESGq` experimental results

In this experimental setup, we considered the Drosophila Melonogaster real dataset, comparing `ESGq` against `rMATS+STAR` (shortly `rMATS`) and `SUPPA2+Salmon` (shortly `SUPPA2`). We ran all tools using their default parameters and 16 threads, when possible, and we compared the reported $\Delta\psi$.

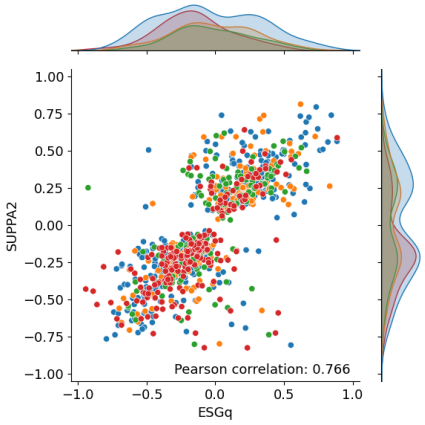
Both `ESGq` and `SUPPA2` quantify AS events starting from a list of events, and, each time an AS event is not quantified, they assign $\Delta\psi = \mathsf{null}$ to that event, not considering this AS event as correctly quantified and discarding it. This occurs, for example, if the event is not sufficiently supported by the RNA-Seq reads.

`ESGq` reported 3276 AS events, `rMATS` reported 3699 AS events, and `SUPPA2` reported 1619 AS events. These AS events are correctly quantified. These results demonstrate the complexity of this task, highlighting the inconsistency between different methodologies based on varying filtering criteria, as previously noted in [206].

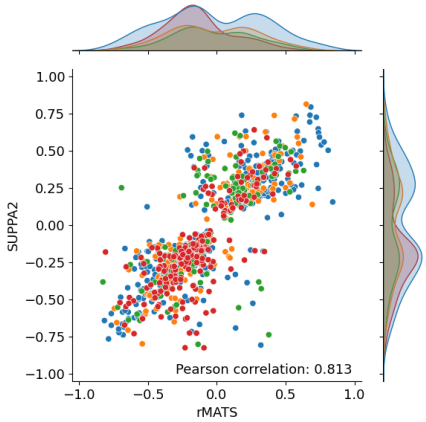
For `rMATS` and `SUPPA2`, which compute the p-value of the quantification, we selected a threshold of ≤ 0.05 to discard not statistically significant AS events from the results of all three tools. After this filtering step, we are left with 933 AS events



(a) `ESGq` vs `rMATS`



(b) `ESGq` vs `SUPPA2`



(c) `rMATS` vs `SUPPA2`

Figure 5.6: Correlation plots of the $\Delta\psi$ values computed by `ESGq`, `rMATS`, and `SUPPA2`. Results refer to the analysis of a 151bp paired-end sample, with $k = 31$ used as the k-mer size for `Salmon` and `SUPPA2`.

Table 5.2: Pearson correlation coefficients between all pairs of tools: `ESGq`, `rMATS`, and `SUPPA2`, across different experimental settings, varying read lengths (51bp, 101bp, and 151bp) and paired- or single-end datasets. For `SUPPA2` we tested various k-mer lengths for transcript quantification by `Salmon`.

Dataset	Read length	Tool1	Tool2	Pearson coefficient
Paired-end	51	`ESGq`	`rMATS`	0.907
			`SUPPA2` (k13)	0.764
			`SUPPA2` (k21)	0.766
			`SUPPA2` (k31)	0.764
		`rMATS`	`SUPPA2` (k13)	0.807
			`SUPPA2` (k21)	0.809
			`SUPPA2` (k31)	0.808
	101	`ESGq`	`rMATS`	0.921
			`SUPPA2` (k13)	0.763
			`SUPPA2` (k21)	0.763
			`SUPPA2` (k31)	0.761
		`rMATS`	`SUPPA2` (k13)	0.797
			`SUPPA2` (k21)	0.796
			`SUPPA2` (k31)	0.797
	151	`ESGq`	`rMATS`	0.918
			`SUPPA2` (k13)	0.766
			`SUPPA2` (k21)	0.766
			`SUPPA2` (k31)	0.766
		`rMATS`	`SUPPA2` (k13)	0.811
			`SUPPA2` (k21)	0.812
			`SUPPA2` (k31)	0.813
Single-end	151	`ESGq`	`rMATS`	0.909
			`SUPPA2` (k13)	0.728
			`SUPPA2` (k21)	0.724
			`SUPPA2` (k31)	0.725
		`rMATS`	`SUPPA2` (k13)	0.789
			`SUPPA2` (k21)	0.785
			`SUPPA2` (k31)	0.786

commonly detected by all the tools. In detail, the AS events categories breakdown is: 374 exon skipping (40%), 190 alternative 3' (20%), 154 alternative 5' (17%), and 215 intron retention (23%).

We report the results of our experiments in Figure 5.6 and Table 5.2. To assess whether the results of the considered methodologies are influenced by read length, we began with the 151-bp paired-end dataset. Then, we manually trimmed the input reads to 51bp and 101bp using `seqtk` [221], thereby generating two additional datasets. We also evaluated whether the paired-end information for the reads improves the accuracy of the AS events quantification. Hence, we merged the two pairs of each replicate into a single sample, simulating a single-end dataset.

Although all tools achieved comparable results, `ESGq` and `rMATS`, which are both based on read alignment, showed a strong correlation, with a Pearson correlation coefficient of 0.918, as shown in Figure 5.6.a. This is an interesting result, considering that the `ESGq` approach is much simpler and less sophisticated than the probabilistic method used by `rMATS`, even though both `ESGq` and `rMATS` rely on alignments against event splicing graphs and a reference genome, respectively. Note that, even using short reads, the three tools achieved a very similar correlation coefficient. Additionally, the use of paired-end and single-end datasets has minimal impact on the results, underscoring the robustness of the considered approaches, with Pearson coefficient differences ranging from 0.009 to 0.041.

Since `SUPPA2` relies on the k-mer–based quasi-mapping of `Salmon`, we also evaluated the impact of k-mer size on its results. Hence, we ran `Salmon` (and consequently `SUPPA2`) with $k = 13$ and $k = 21$, in addition to the default setting $k = 31$. The choice of k seems not to affect `SUPPA2` differential quantification results.

Looking at results in Figure 5.6.b and in Figure 5.6.c, `SUPPA2`, with any k value, is less correlated to the two other approaches, with a Pearson correlation coefficient ranging from 0.766 to 0.813. Being `SUPPA2` based on transcript quantification instead of alignments, this result is expected [37].

In Table 5.3 we report the Pearson correlation coefficient between each pair of tools, distinguishing each AS event category. No clear trend can be observed. Consistent with what is reported in Table 5.2, `ESGq` and `rMATS` exhibit the highest correlation coefficient, especially for exon skipping events. Their lower correlation

Table 5.3: Pearson correlation coefficients between `ESGq`, `rMATS`, and `SUPPA2`, by AS event category on the 151bp paired-end dataset. For `SUPPA2` we ran `Salmon` with $k = 31$.

Event type	Tool1	Tool2	Pearson coefficient
SE	`ESGq`	`rMATS`	0.952
	`ESGq`	`SUPPA2`	0.808
	`rMATS`	`SUPPA2`	0.836
A3	`ESGq`	`rMATS`	0.868
	`ESGq`	`SUPPA2`	0.786
	`rMATS`	`SUPPA2`	0.862
A5	`ESGq`	`rMATS`	0.920
	`ESGq`	`SUPPA2`	0.666
	`rMATS`	`SUPPA2`	0.708
RI	`ESGq`	`rMATS`	0.859
	`ESGq`	`SUPPA2`	0.677
	`rMATS`	`SUPPA2`	0.760

result occurs in the case of intron retention events. `ESGq` and `SUPPA2` (ran only with the default k value for `Salmon`) show higher correlation on exon skipping events and lower correlation on alternative donor events, while `rMATS` and `SUPPA2` show higher correlation on alternative acceptor site events and lower correlation on alternative donor events. A more detailed biological analysis would be required to fully interpret these results, but this is beyond the scope of the present thesis.

Regarding execution performances, `ESGq` is computationally efficient, requiring 1GB of RAM to complete the analysis in half an hour. `ESGq` is outperformed in time by `SUPPA2`, which takes 10 minutes, requiring 1.5GB of RAM. On the contrary, `rMATS` is the most expensive approach, taking more than 5 hours to complete the analysis and requiring 8GB of RAM. These poor performances are mainly caused by reading `STAR` alignments, which alone required between half an hour and two hours per sample. These results show that `ESGq`, leveraging the event splicing graph for fast and accurate read alignments, is 10x faster than `rMATS`, achieving comparable results.

5.3.5 `pantas` experimental results

We evaluate `pantas` efficacy and correctness in quantifying AS events across conditions, on both simulated and real data. As for `ESGq`, we ran all tools with their default parameters, filtering out AS events according to the reported statistical significance, as suggested by the authors of the tools, for example, based on the p-value when reported. As `ESGq`, `pantas` does not report the p-value or any other metrics to evaluate the statistical significance; hence, we filter AS events by a minimum splice junction coverage/supporting threshold w, having $w = 3$ by default.

We proposed three experimental setups, using the three datasets previously described: a) quantify both annotated and novel AS events on the simulated Drosophila Melanogaster dataset, b) quantify just annotated AS events on the real RNA-Seq dataset from Drosophila Melanogaster, and c) quantify RT-PCR validated AS events from the real human RNA-Seq dataset.

`pantas` results on the simulated Drosophila Melanogaster dataset In this experimental setup, we evaluate the correctness of `pantas` in detecting, and not quantifying, both annotated and novel AS events using the dataset simulated with `asimulator`. We compared `pantas` against `rMATS` [38], `SUPPA2` [39], and `whippet` [37], evaluating precision, recall, and F1-measure in respect to the ground truth AS events reported by `asimulator`. For `pantas` we used the default simplification level. Not being focused on the AS events quantification, in these experiments, we did not perform any post-filtering based on the statistical significance of the AS events, but we conjecture that this filtering step would have improved the precision of the tools while lowering their recall. For annotated AS events, we define an annotated AS event as a true positive if its reported splice junctions match those provided by `asimulator`. On the other hand, for novel AS events, given the higher complexity of correctly detecting novel splice junctions, we considered a novel AS event a true positive if at least one of its splice junctions is correctly identified.

We performed experiments varying the number of minimum reads $\mathcal{W}$ needed to support the AS events, hence filtering the original ground truth dataset into multiple truth sets, testing $\mathcal{W} \in \{1, 3, 5, 10, 20\}$. Note that a bigger $\mathcal{W}$ value results in a smaller set of highly supported events, as we discard all the AS events not

Table 5.4: Number of AS events by category reported by `asimulator` and supported by at least $\mathcal{W}$ reads.

$\mathcal{W}$	Exon Skippings	Alternative 3' Sites	Alternative 5' Sites	Intron Retentions
1	214	219	209	231
3	152	170	155	178
5	129	140	130	153
10	95	108	95	120
20	65	65	55	75

supported by at least $\mathcal{W}$ reads according to `asimulator`. Some statistics of the number of AS events varying $\mathcal{W}$ are reported in Table 5.4.

We used these filtered sets to compute the true positives and false positives for each tool, while the original unfiltered truth set was used to compute the false negatives. This strategy penalizes tools for identifying false events with low support, without penalizing them for correctly calling low-supported true events.

In Figure 5.7, Table 5.5, Table 5.6, and Table 5.7, we report the results for annotated events. Considering all events reported by `asimulator` without any filtering, *i.e.* for $\mathcal{W} = 1$, all tools achieved higher precision at the expense of their recall. On the filtered datasets, noting that a higher threshold $\mathcal{W}$ resulted in a smaller truth set of events and fewer true positives, the recall of the tools increases while lowering the precision. In this setup, `pantas` and `rMATS` are the most accurate tools for detecting alternative splicing events, followed by `SUPPA2`.

Lacking a proper statistical validation, `pantas` increases its performance while increasing $\mathcal{W}$ more than the competitors, especially when moving from $\mathcal{W} = 1$ to $\mathcal{W} = 3$. Recalling that `pantas` filters AS events if at least 3 reads do not cover splice junctions, the results in Figure 5.7 are expected. In fact, when considering all AS events reported by `asimulator` with a coverage of at least 3, `pantas` achieved the highest accuracy across nearly all event types. These results are also shown in Table 5.5. For $\mathcal{W} = 3$, `pantas` achieved a recall of 0.725, which is 0.045 lower than `rMATS`, and a precision of 0.563, which is 0.07 higher than `rMATS`, making it the most accurate approach with an F1 score of 0.634.

Assuming `rMATS` as the best competitor, as in the experiments made for `ESGq`, `pantas` fails to achieve similar results only in the case of intron retentions. However, `pantas` achieved the best precision for each AS category.

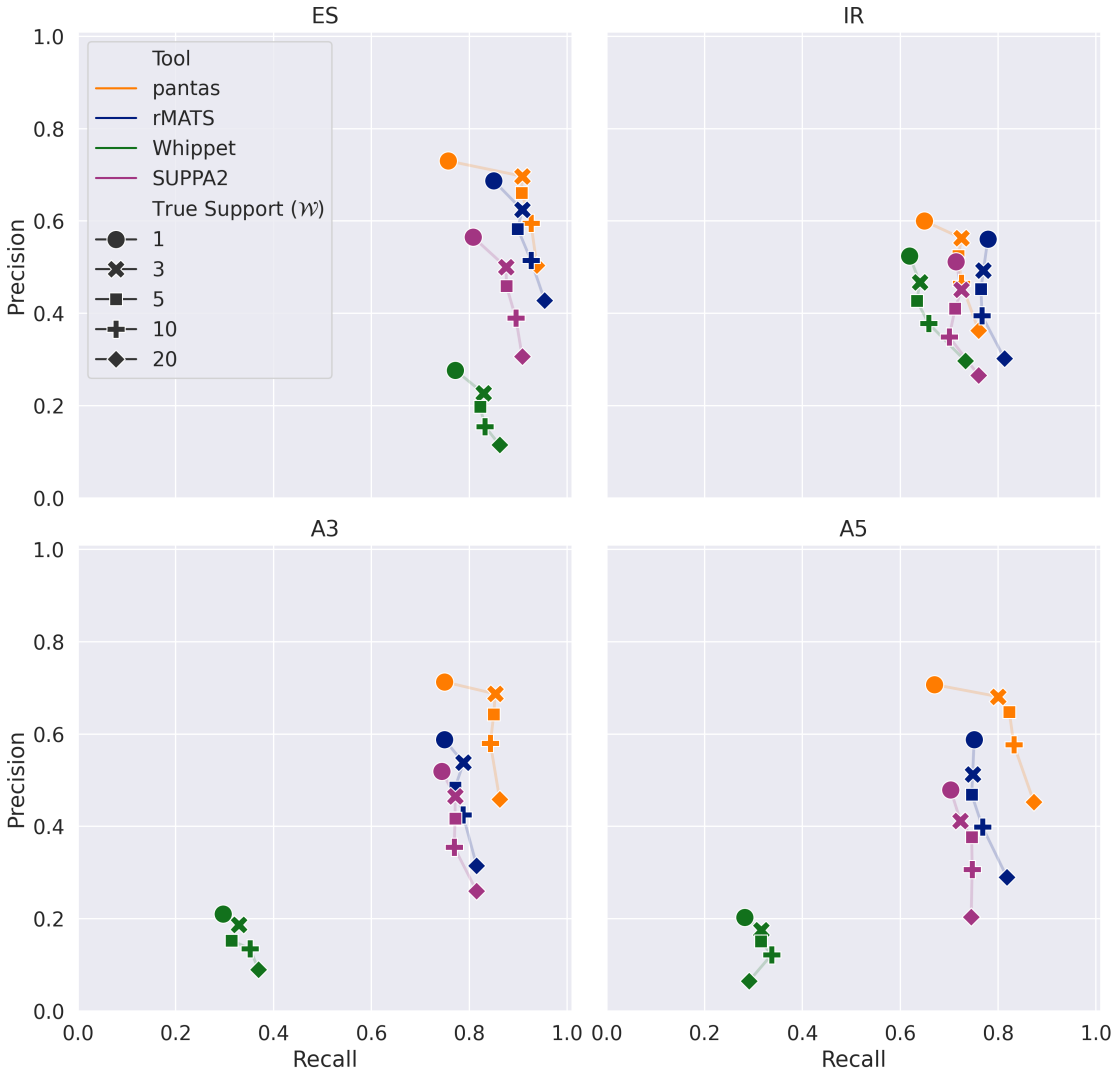


Figure 5.7: Annotated AS events detection results on simulated Drosophila Melanogaster data. For `pantas` we used the default threshold value $w = 3$. Precision and recall are computed by comparing the `asimulator` truth set filtered by $\mathcal{W}$ with the output of each tool. Results are reported by event category: ES (exon skipping), IR (intron retention), A3 (alternative 3' splicing/acceptor site), and A5 (alternative 5' splicing/donor site). Different colors represent the tools, and different markers represent $\mathcal{W} \in \{1, 3, 5, 10, 20\}$.

Table 5.5: First table for full detection results on annotated AS events from simulated data from Drosophila Melanogaster. Results are reported by event category: ES (exon skipping), IR (intron retention), A3 (alternative 3' splicing/acceptor site), and A5 (alternative 5' splicing/donor site). Metric are reported for $\mathcal{W} \in \{1, 3, 5\}$.

$\mathcal{W}$	**Event**	**Tool**	**TP**	**FN**	**FP**	**Precision**	**Recall**	**F1**
1	ES	pantas	162	52	60	0.73	0.757	0.743
1	ES	rMATS	182	32	83	0.687	0.85	0.76
1	ES	whippet	165	49	430	0.277	0.771	0.408
1	ES	SUPPA2	173	41	133	0.565	0.808	0.665
1	IR	pantas	150	81	100	0.6	0.649	0.624
1	IR	rMATS	180	51	141	0.561	0.779	0.652
1	IR	whippet	143	88	130	0.524	0.619	0.567
1	IR	SUPPA2	165	66	157	0.512	0.714	0.597
1	A3	pantas	164	55	66	0.713	0.749	0.731
1	A3	rMATS	164	55	115	0.588	0.749	0.659
1	A3	whippet	65	154	244	0.21	0.297	0.246
1	A3	SUPPA2	163	56	151	0.519	0.744	0.612
1	A5	pantas	140	69	58	0.707	0.67	0.688
1	A5	rMATS	157	52	110	0.588	0.751	0.66
1	A5	whippet	59	150	231	0.203	0.282	0.236
1	A5	SUPPA2	147	62	160	0.479	0.703	0.57
3	ES	pantas	138	14	60	0.697	0.908	0.789
3	ES	rMATS	138	14	83	0.624	0.908	0.74
3	ES	whippet	126	26	430	0.227	0.829	0.356
3	ES	SUPPA2	133	19	133	0.5	0.875	0.636
3	IR	pantas	129	49	100	0.563	0.725	0.634
3	IR	rMATS	137	41	141	0.493	0.77	0.601
3	IR	whippet	114	64	130	0.467	0.64	0.54
3	IR	SUPPA2	129	49	157	0.451	0.725	0.556
3	A3	pantas	145	25	66	0.687	0.853	0.761
3	A3	rMATS	134	36	115	0.538	0.788	0.64
3	A3	whippet	56	114	244	0.187	0.329	0.238
3	A3	SUPPA2	131	39	151	0.465	0.771	0.58
3	A5	pantas	124	31	58	0.681	0.8	0.736
3	A5	rMATS	116	39	110	0.513	0.748	0.609
3	A5	whippet	49	106	231	0.175	0.316	0.225
3	A5	SUPPA2	112	43	160	0.412	0.723	0.525

Table 5.6: Second table for full detection results on annotated AS events from simulated data from Drosophila Melanogaster. Results are reported by event category: ES (exon skipping), IR (intron retention), A3 (alternative 3' splicing/acceptor site), and A5 (alternative 5' splicing/donor site). Metrics are reported for $\mathcal{W} \in \{5, 10\}$.

$\mathcal{W}$	**Event**	**Tool**	**TP**	**FN**	**FP**	**Precision**	**Recall**	**F1**
5	ES	pantas	117	12	60	0.661	0.907	0.765
5	ES	rMATS	116	13	83	0.583	0.899	0.707
5	ES	whippet	106	23	430	0.198	0.822	0.319
5	ES	SUPPA2	113	16	133	0.459	0.876	0.603
5	IR	pantas	110	43	100	0.524	0.719	0.606
5	IR	rMATS	117	36	141	0.453	0.765	0.569
5	IR	whippet	97	56	130	0.427	0.634	0.511
5	IR	SUPPA2	109	44	157	0.41	0.712	0.52
5	A3	pantas	119	21	66	0.643	0.85	0.732
5	A3	rMATS	108	32	115	0.484	0.771	0.595
5	A3	whippet	44	96	244	0.153	0.314	0.206
5	A3	SUPPA2	108	32	151	0.417	0.771	0.541
5	A5	pantas	107	23	58	0.648	0.823	0.725
5	A5	rMATS	97	33	110	0.469	0.746	0.576
5	A5	whippet	41	89	231	0.151	0.315	0.204
5	A5	SUPPA2	97	33	160	0.377	0.746	0.501
10	ES	pantas	88	7	60	0.595	0.926	0.724
10	ES	rMATS	88	7	83	0.515	0.926	0.662
10	ES	whippet	79	16	430	0.155	0.832	0.262
10	ES	SUPPA2	85	10	133	0.39	0.895	0.543
10	IR	pantas	87	33	100	0.465	0.725	0.567
10	IR	rMATS	92	28	141	0.395	0.767	0.521
10	IR	whippet	79	41	130	0.378	0.658	0.48
10	IR	SUPPA2	84	36	157	0.349	0.7	0.465
10	A3	pantas	91	17	66	0.58	0.843	0.687
10	A3	rMATS	85	23	115	0.425	0.787	0.552
10	A3	whippet	38	70	244	0.135	0.352	0.195
10	A3	SUPPA2	83	25	151	0.355	0.769	0.485
10	A5	pantas	79	16	58	0.577	0.832	0.681
10	A5	rMATS	73	22	110	0.399	0.768	0.525
10	A5	whippet	32	63	231	0.122	0.337	0.179
10	A5	SUPPA2	71	24	160	0.307	0.747	0.436

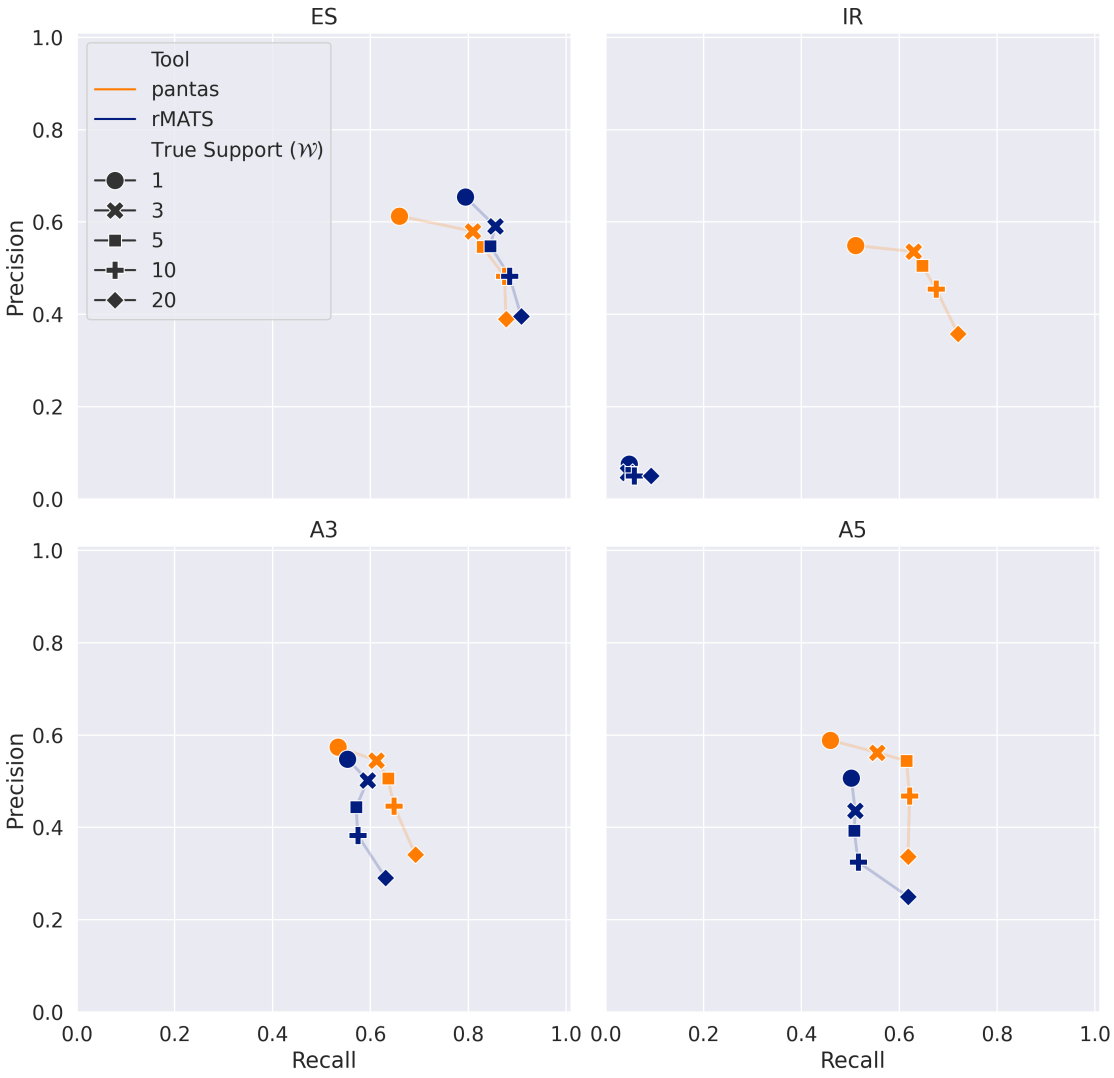


Figure 5.8: Novel AS events detection results on simulated data from Drosophila Melanogaster (novel events). For `pantas` we used the default threshold value $w = 3$. Results are broken down by event category, merging exonic and intronic variants of the novel AS events: ES (exon skipping), IR (intron retention), A3 (alternative 3' splicing/acceptor site), and A5 (alternative 5' splicing/donor site). Different colors represent the tools, and different markers represent $\mathcal{W} \in \{1, 3, 5, 10, 20\}$.

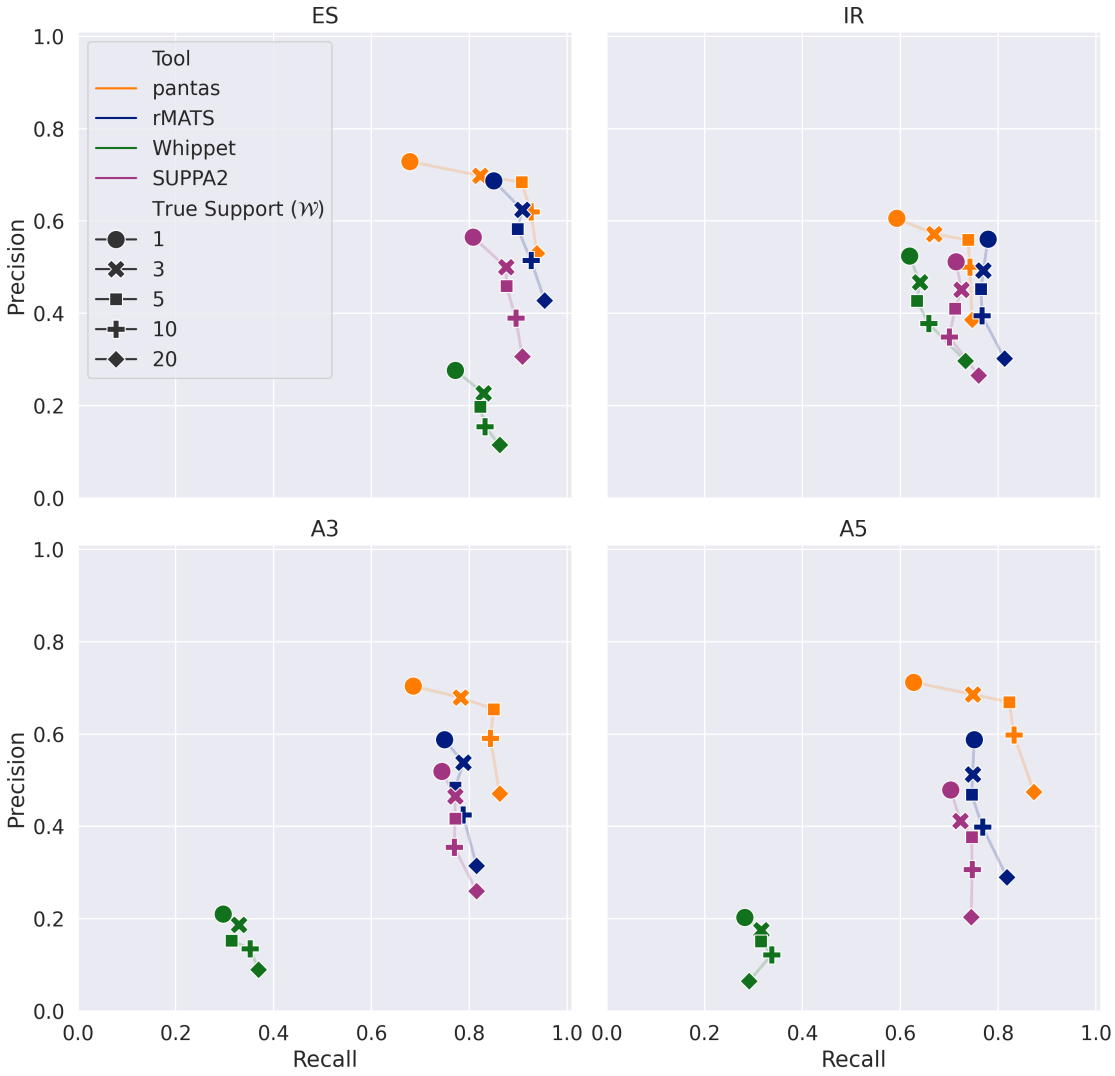


Figure 5.9: Annotated AS events detection results on simulated Drosophila Melanogaster data. For `pantas` we used the threshold value $w = 5$. Precision and recall are computed by comparing the `asimulator` truth set filtered by $\mathcal{W}$ with the output of each tool. Results are reported by event category: ES (exon skipping), IR (intron retention), A3 (alternative 3' splicing/acceptor site), and A5 (alternative 5' splicing/donor site). Different colors represent the tools, and different markers represent $\mathcal{W} \in \{1, 3, 5, 10, 20\}$.

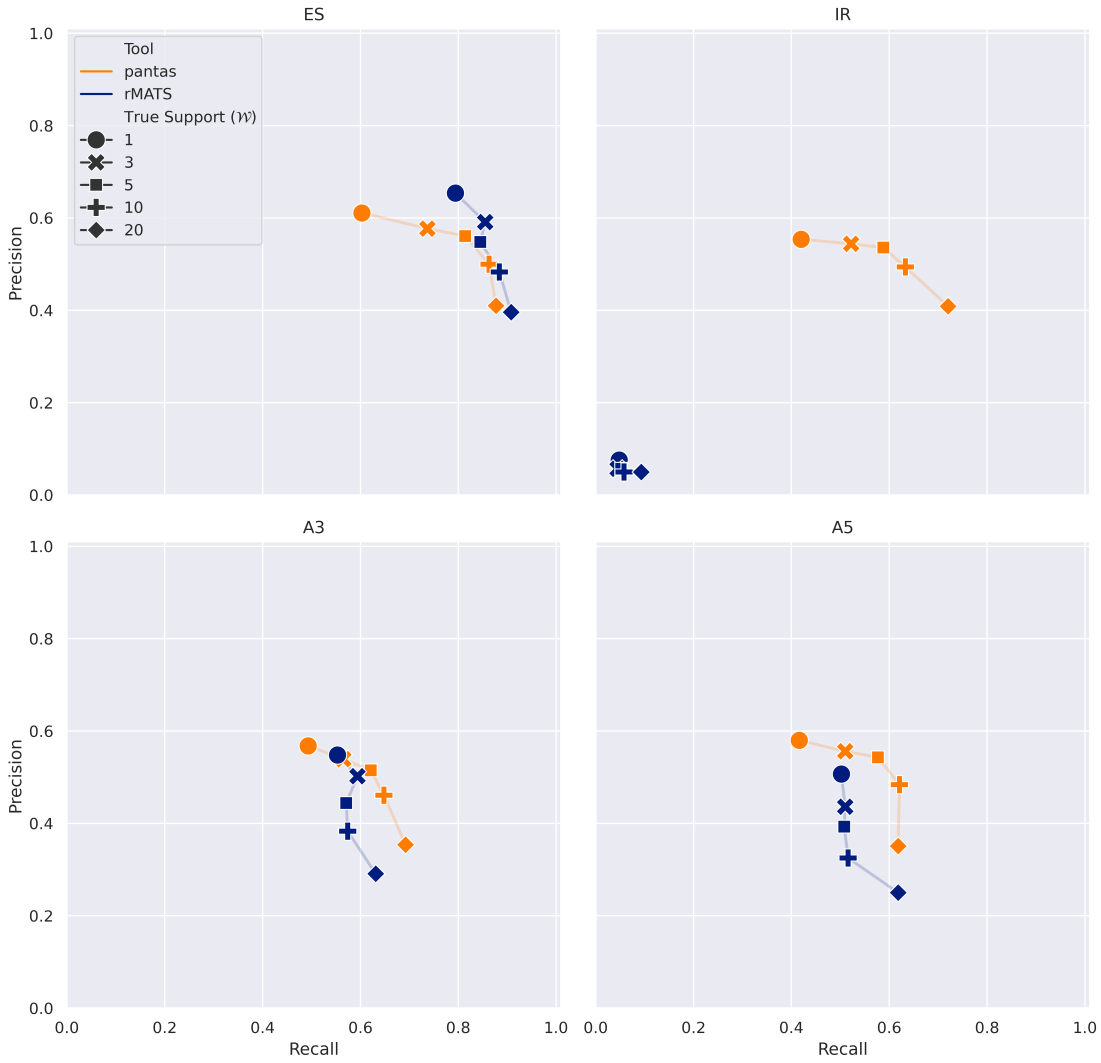


Figure 5.10: Novel AS events detection results on simulated data from Drosophila Melanogaster (novel events). For `pantas` we used the threshold value $w = 5$. Results are broken down by event category, merging exonic and intronic variants of the novel AS events: ES (exon skipping), IR (intron retention), A3 (alternative 3' splicing/acceptor site), and A5 (alternative 5' splicing/donor site). Different colors represent the tools, and different markers represent $\mathcal{W} \in \{1, 3, 5, 10, 20\}$.

Table 5.7: Third table for full detection results on annotated AS events from simulated data from Drosophila Melanogaster. Results are reported by event category: ES (exon skipping), IR (intron retention), A3 (alternative 3' splicing/acceptor site), and A5 (alternative 5' splicing/donor site). Metrics are reported for $\mathcal{W} \in \{20\}$.

$\mathcal{W}$	**Event**	**Tool**	**TP**	**FN**	**FP**	**Precision**	**Recall**	**F1**
20	ES	`pantas`	61	4	60	0.504	0.938	0.656
20	ES	`rMATS`	62	3	83	0.428	0.954	0.59
20	ES	`whippet`	56	9	430	0.115	0.862	0.203
20	ES	`SUPPA2`	59	6	133	0.307	0.908	0.459
20	IR	`pantas`	57	18	100	0.363	0.76	0.491
20	IR	`rMATS`	61	14	141	0.302	0.813	0.44
20	IR	`whippet`	55	20	130	0.297	0.733	0.423
20	IR	`SUPPA2`	57	18	157	0.266	0.76	0.394
20	A3	`pantas`	56	9	66	0.459	0.862	0.599
20	A3	`rMATS`	53	12	115	0.315	0.815	0.455
20	A3	`whippet`	24	41	244	0.09	0.369	0.144
20	A3	`SUPPA2`	53	12	151	0.26	0.815	0.394
20	A5	`pantas`	48	7	58	0.453	0.873	0.596
20	A5	`rMATS`	45	10	110	0.29	0.818	0.429
20	A5	`whippet`	16	39	231	0.065	0.291	0.106
20	A5	`SUPPA2`	41	14	160	0.204	0.745	0.32

Finally, as shown in Figure 5.8 and Table 5.8, we can discuss the results in the novel AS events scenario. Note that we could not include `SUPPA2` and `whippet` in the analysis, as they are not able to handle novel events. As expected, both `pantas` and `rMATS` achieved lower accuracy compared to the annotated AS events scenario, but the comparison followed the same trend. In fact, `pantas` achieves the best trade-off between precision and recall when $\mathcal{W} = 3$.

Note that, in this scenario, `pantas` did not achieve the accuracy of `rMATS` when calling exon skipping events, showing a loss of 0.014 in F1 when $\mathcal{W} = 3$. At the same time, `pantas` achieved the highest precision and recall for all other event categories, particularly when the true events were highly supported, and proved to be the most accurate approach for detecting novel intron retentions. Note that in literature novel intron retentions are the hardest AS event category to detect correctly [222].

Overall, the experiments proved the good accuracy `pantas`, with a proper setup for the threshold w, in detecting both annotated and novel scenarios, confirming the validity and the efficacy of the use of a pangenome graph in the context of

Table 5.8: Full detection results on novel AS events from simulated data from Drosophila Melanogaster. Results are reported by event category: ES (exon skipping), IR (intron retention), A3 (alternative 3' splicing/acceptor site), and A5 (alternative 5' splicing/donor site). Metrics are reported for $\mathcal{W} \in \{1, 3, 5, 10, 20\}$.

$\mathcal{W}$	**Event**	**Tool**	**TP**	**FN**	**FP**	**Precision**	**Recall**	**F1**
1	ES	`pantas`	141	73	89	0.613	0.659	0.635
1	ES	`rMATS`	170	44	90	0.654	0.794	0.717
1	IR	`pantas`	118	113	97	0.549	0.511	0.529
1	IR	`rMATS`	11	220	133	0.076	0.048	0.059
1	A3	`pantas`	117	102	87	0.574	0.534	0.553
1	A3	`rMATS`	121	98	100	0.548	0.553	0.55
1	A5	`pantas`	96	113	67	0.589	0.459	0.516
1	A5	`rMATS`	105	104	102	0.507	0.502	0.505
3	ES	`pantas`	123	29	89	0.58	0.809	0.676
3	ES	`rMATS`	130	22	90	0.591	0.855	0.699
3	IR	`pantas`	112	66	97	0.536	0.629	0.579
3	IR	`rMATS`	8	170	133	0.057	0.045	0.05
3	A3	`pantas`	104	66	87	0.545	0.612	0.576
3	A3	`rMATS`	101	69	100	0.502	0.594	0.544
3	A5	`pantas`	86	69	67	0.562	0.555	0.558
3	A5	`rMATS`	79	76	102	0.436	0.51	0.47
5	ES	`pantas`	107	22	89	0.546	0.829	0.658
5	ES	`rMATS`	109	20	90	0.548	0.845	0.665
5	IR	`pantas`	99	54	97	0.505	0.647	0.567
5	IR	`rMATS`	8	145	133	0.057	0.052	0.054
5	A3	`pantas`	89	51	87	0.506	0.636	0.563
5	A3	`rMATS`	80	60	100	0.444	0.571	0.5
5	A5	`pantas`	80	50	67	0.544	0.615	0.578
5	A5	`rMATS`	66	64	102	0.393	0.508	0.443
10	ES	`pantas`	83	12	89	0.483	0.874	0.622
10	ES	`rMATS`	84	11	90	0.483	0.884	0.625
10	IR	`pantas`	81	39	97	0.455	0.675	0.544
10	IR	`rMATS`	7	113	133	0.05	0.058	0.054
10	A3	`pantas`	70	38	87	0.446	0.648	0.528
10	A3	`rMATS`	62	46	100	0.383	0.574	0.459
10	A5	`pantas`	59	36	67	0.468	0.621	0.534
10	A5	`rMATS`	49	46	102	0.325	0.516	0.398
20	ES	`pantas`	57	8	89	0.39	0.877	0.54
20	ES	`rMATS`	59	6	90	0.396	0.908	0.551
20	IR	`pantas`	54	21	97	0.358	0.72	0.478
20	IR	`rMATS`	7	68	133	0.05	0.093	0.065
20	A3	`pantas`	45	20	87	0.341	0.692	0.457
20	A3	`rMATS`	41	24	100	0.291	0.631	0.398
20	A5	`pantas`	34	21	67	0.337	0.618	0.436
20	A5	`rMATS`	34	21	102	0.25	0.618	0.356

alternative splicing.

To further validate this claim, we also tested `pantas` with $w = 5$ in both annotated and novel scenarios. Even if this parameter does not match the expected coverage, `pantas` achieved the best trade-off between precision and recall. `pantas` achieve competitive, if not superior, accuracy, particularly for highly supported events, as shown in Figure 5.9 for annotated AS events and in Figure 5.10 for novel AS events.

Note that, as expected, this time `pantas` achieve the best trade-off between precision and recall for $\mathcal{W} \geq 5$.

We conjecture that the default value $w = 3$ is the most suitable for most scenarios, but setting this parameter is not straightforward since RNA-Seq coverage is not always uniform and `pantas` does not perform any statistical validation.

`pantas` results on the real Drosophila Melanogaster dataset In this experimental setup, we considered the real RNA-Seq dataset from Drosophila Melanogaster, evaluating the accuracy of `pantas` in differentially quantifying annotated AS events. We compared, all with default parameters, `pantas` against `rMATS` [38], `SUPPA2` [39], and `whippet` [37], therefore we considered only annotated AS events. For `pantas` we used the more aggressive simplification level, to make the annotated spliced pangenome graph indexing step feasible. We recall that this level can alter complex haplotype-aware transcripts to reduce the graph complexity. This was necessary because `vg mpmap` required more than three days to align a single replicate, using 32 threads, with the annotated spliced pangenome graph produced using the default simplification level.

Time and space performances of `pantas` and other tools are reported in Table 5.9, which includes the results of the preprocessing steps. Without considering the input preparation step, `pantas` can be viewed as a practical solution, having completed its differential analysis in less than 2 hours, using 32 threads and requiring 16GB of RAM.

Figure 5.11 and Figure 5.12 show the quantification results of AS events for all the tools. The latter reports the results with a breakdown per AS event category. For this dataset, we do not have a wet-lab validation/ground truth dataset, hence we performed an all-vs-all comparison of the $\Delta\psi$ reported, as we did in the `ESGq`

Table 5.9: Time and space results on real data from Drosophila Melanogaster.

Tool	Time (min.)	RAM (GB)
vg index	128	20
vg mpmap	388*	3
pantas weight	26*	5
pantas call	11*	16
pantas quant	1	1
STAR index	1	4
STAR align	20*	6
rMATS	2	1
salmon index	0.5	1
salmon quant	1*	1
SUPPA2	0.5	1
whippet index	2	1
whippet quant	9*	1
whippet delta	1	1

* average over the 6 samples.

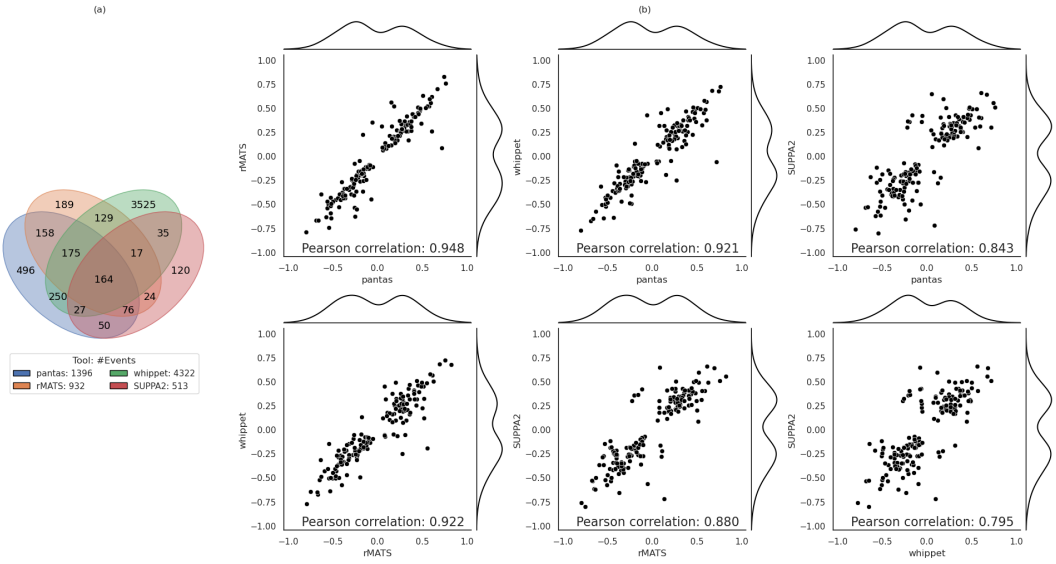


Figure 5.11: Results on real Drosophila melanogaster data. In subfigure (a), a Venn diagram shows the number of AS events reported by each tool. In subfigure (b), all-vs-all correlation plots of the $\Delta\psi$ values are shown for the 164 events shared among all the tools.

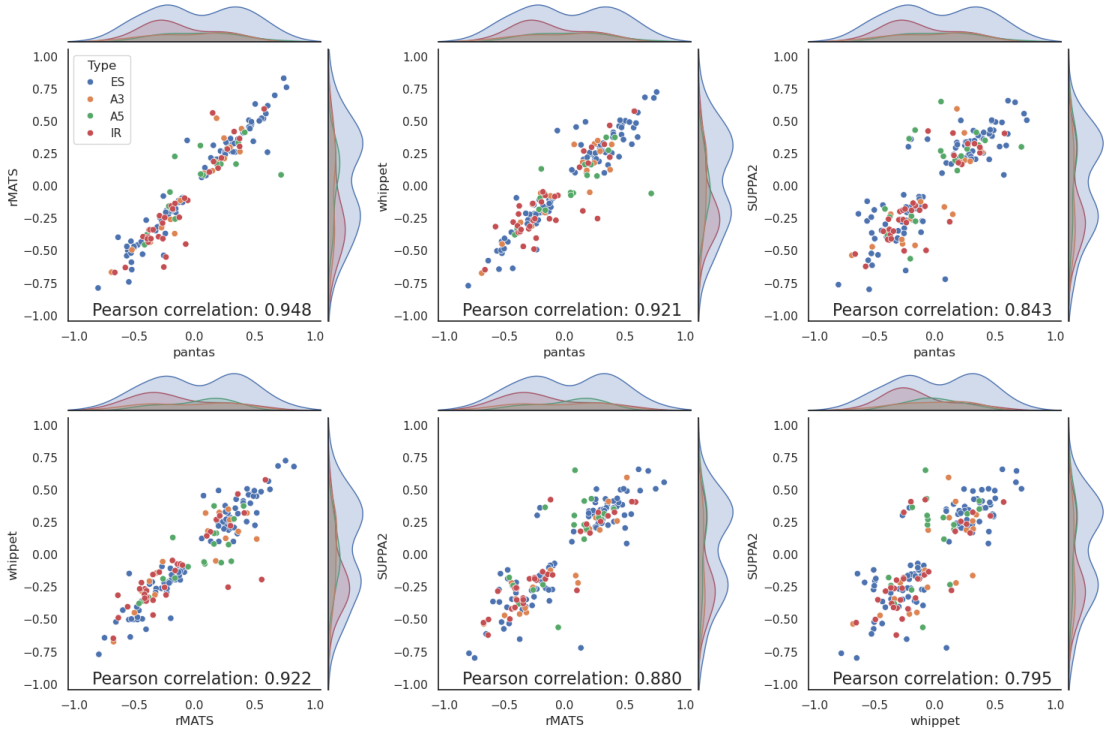


Figure 5.12: Results on real data from Drosophila Melanogaster. All-vs-all correlation plots of the $\Delta\psi$ reported by the considered tools for the 164 events shared among all the tools. Each point represents an AS event, colored according to its category.

Table 5.10: Alternative splicing events count for Drosophila Melanogaster experiments on real data.

Event	`pantas`	`rMATS`	`whippet`	`SUPPA2`
ES	370	366	3205	140
A3	337	200	401	129
A5	367	189	385	131
IR	322	177	331	113
All	1396	932	4322	513

Table 5.11: Alternative splicing events count for Drosophila Melanogaster experiments without any filtering on minimum $\Delta\psi$, minimum probability, and p-value.

Event	pantas	rMATS	whippet	SUPPA2
ES	553	1220	12 030	755
A3	627	754	731	944
A5	630	638	705	909
IR	506	536	537	663
All	2316	3148	14 003	3271

experiments. We also discarded all the AS events reported as non-statistically significant by the tool, if metrics for this evaluation are available. In detail, for rMATS and SUPPA2, we filtered out AS events with a p-value greater than or equal to 0.05, while, for whippet, we filtered out all the AS events with a probability smaller than 0.9.

In Figure 5.11.a, we present a Venn diagram with the detected As events. In detail, pantas detects 1369 AS events, rMATS detects 932 AS events, whippet detects 4322 AS events, and SUPPA2 detects 513 AS events. Table 5.10 reports these numbers broken down by event category, and, as shown in Figure 5.11.a, 164 AS events are shared among all the tools.

Figure 5.11.b shows the quantification results for this subset of 164 AS events, in terms of $\Delta\psi$ and Pearson correlation coefficient. We can note a strong correlation shared among the tested tools. pantas showed a very high correlation with rMATS, with a Pearson correlation of 0.948, and with whippet, with a Pearson correlation of 0.921; however, pantas had a lower correlation with SUPPA2, with a Pearson correlation of 0.843. All the tools show lower correlation with SUPPA2. This is somewhat expected, since SUPPA2 quantifies AS events from transcript quantification, using Salmon for a quasi-mapping approach based on k-mer analysis, rather than from read alignment.

Figure 5.13 and Table 5.11 reports the same results without any filtering on minimum $\Delta\psi$, minimum probability, and p-value, having more AS events to consider, as shown in the Venn diagram in Figure 5.13.a. Note that in Figure 5.13.b we can notice a substantial loss of correlation, even if pantas and rMATS have been able to achieve the strongest correlation, with a Pearson correlation of 0.832.

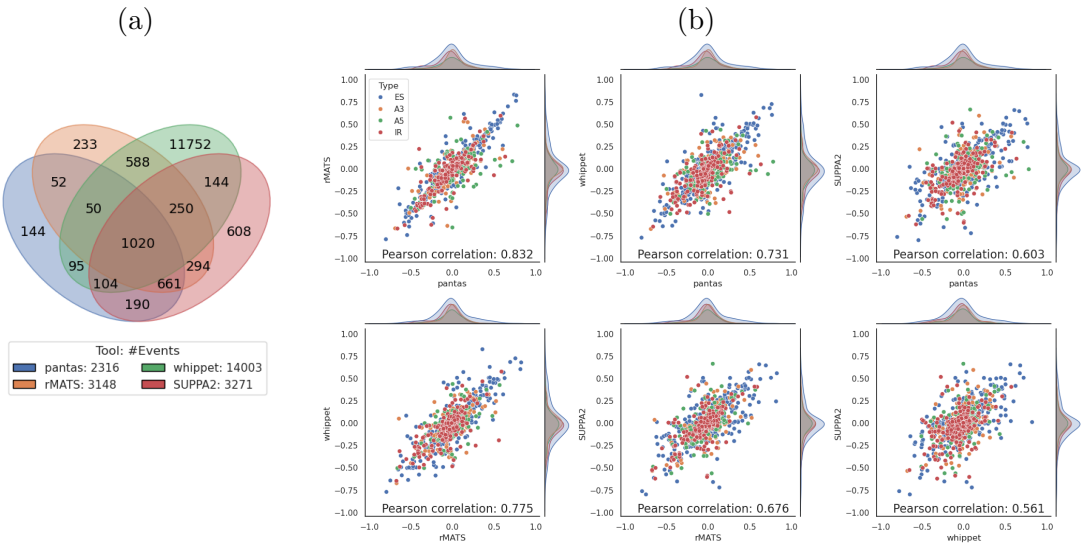


Figure 5.13: Results on real Drosophila melanogaster data without any filtering on minimum $\Delta\psi$, minimum probability, and p-value. In subfigure (a), a Venn diagram shows the number of AS events reported by each tool. In subfigure (b), all-vs-all correlation plots of the $\Delta\psi$ values are shown for the 1020 events shared among the four tools. Each point represents an event, colored according to its category.

Overall, the high correlation between `pantas` and all other read-alignment–based approaches demonstrates that read alignment to the annotated spliced pangenome graph is effective and can be reliably used for differential quantification of AS events also in a real case scenario.

`pantas` results on the human real RT-PCR validated dataset As the final experimental scenario, we considered the real human RNA-Seq dataset with 77 RT-PCR validated AS events. These RT-PCR validated AS events are used as ground truth to evaluate `pantas` in detecting and quantifying AS events. For `pantas`, we used the third simplification level, having a small panel of genes as input, which is a common scenario in transcriptomics [223]. Hence, we constructed an annotated spliced pangenome graph limited to the genomic loci of interest. This simplification level is also required because we were unable to build the entire human spliced pangenome `GCSA2` index, exhausting the RAM available on our machine (256GB). By retaining only the genes of interest, we reduced the total pipeline runtime to under 3 hours and the RAM usage to below 8GB, without compromising the accuracy of AS event detection. The time and space performances of the tested tools are presented in Table 5.12, which includes the results of the preprocessing steps.

Note that aligning the input RNA-Seq dataset against a reduced graph may produce low-quality alignments. For example, reads might be mapped to the wrong location in the graph if the correct gene over which the read should align has been discarded. This also affects the execution time of the alignment. To ensure a fair comparison, while running the other tools on the original dataset, we decided to filter the input RNA-Seq dataset for `pantas` using `shark` [223], retaining only reads that are likely to originate from the genes of interest. Our results show that this preprocessing step does not affect the quantification results. As in previous experiments, we ran all the tools with default parameters, evaluating the detection results using the RT-PCR validated AS events dataset. The accuracy of the quantification is validated by comparing the $\Delta\psi$ of each tool with the experimental $\Delta\psi$ provided by RT-PCR validation.

We filtered AS events using the same criteria described for the Drosophila Melonogaster real dataset, discarding events with $|\Delta\psi| < 0.05$. Figure 5.14 presents

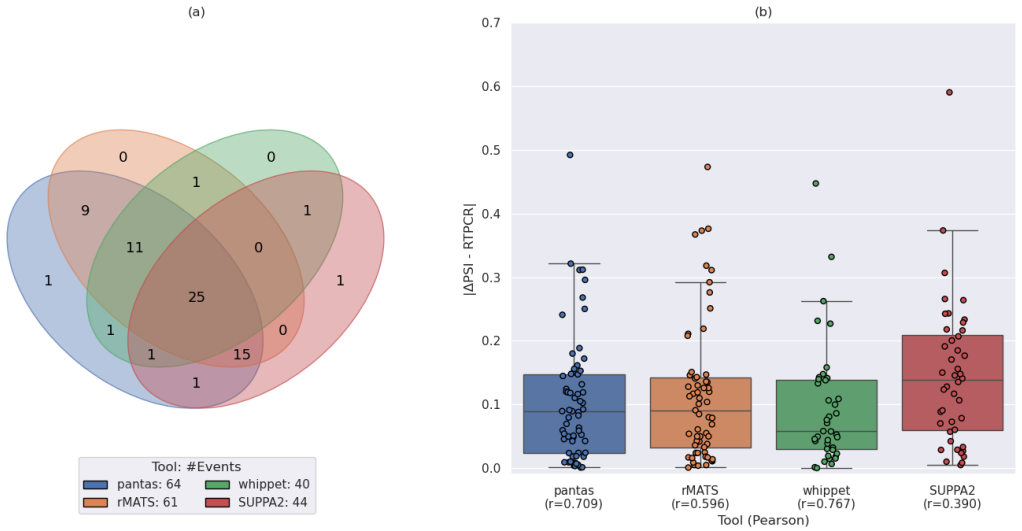


Figure 5.14: Results on the human real RT-PCR validated dataset. In subfigure (a), a Venn diagram shows the number of statistically significant differential AS events reported by each tool, with the legend indicating the total number of events reported. In subfigure (b), a boxplot shows the distribution of differences between the $\Delta\psi$ predicted by each tool and the $\Delta\psi$ measured by RT-PCR. The x-axis labels report the tool name and the Pearson correlation coefficient r between the predicted and RT-PCR $\Delta\psi$ values. Note that, since `pantas` does not compute statistical significance, events were filtered only by the reported $\Delta\psi$.

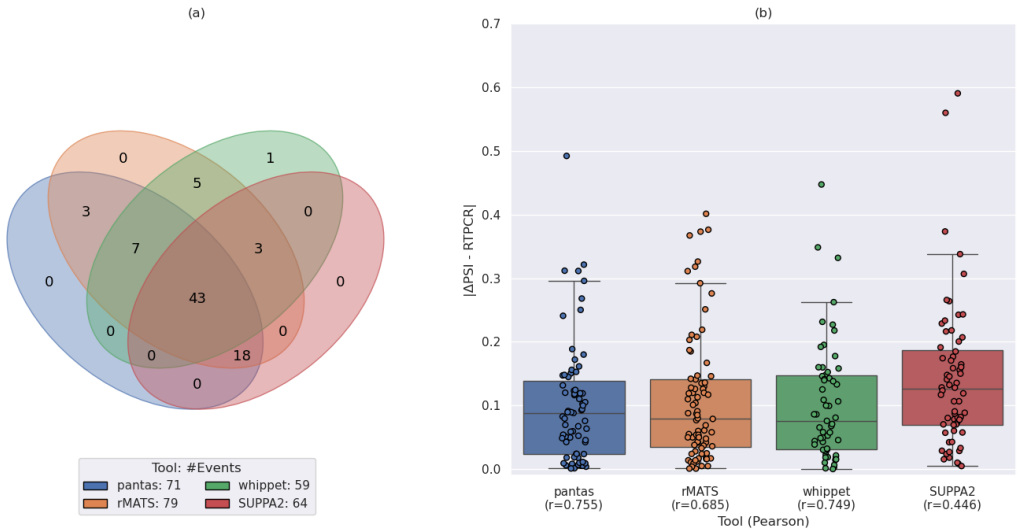


Figure 5.15: Results on the human real RT-PCR validated dataset without any filtering step. In subfigure (a), a Venn diagram shows the number of AS events reported by each tool, with the legend indicating the total number of events reported. In subfigure (b), a boxplot displays the distribution of differences between the $\Delta\psi$ predicted by each tool and the $\Delta\psi$ measured by RT-PCR. The x-axis labels indicate the tool name and the Pearson correlation r between predicted and RT-PCR $\Delta\psi$.

.

Table 5.12: Time and space results on real Human data.

Tool	Time (min)	RAM (GB)
`shark`	2*	2
`vg index`	2	8
`vg mpmap`	20*	3
`pantas weight`	1*	1
`pantas call`	1*	1
`pantas quant`	1	1
STAR index	43	38
STAR align	9*	32
rMATS	9	2
salmon index	2	1
salmon quant	2*	3
SUPPA2	13	1
whippet index	45	5
whippet quant	29*	6
whippet delta	2	1

* average over the 6 samples.

the results for the filtered dataset, while Figure 5.15 reports the results without any filtering step.

As shown in Figure 5.14.a `pantas` reported the highest number of AS events, 64 out of 77, followed by `rMATS`, 61 out of 77, `SUPPA2`, 44 out of 77, and `whippet`, 40 out of 77. Only 25 events were reported by all the tools. `pantas` is the only tool to detect one AS event, while it is not able to detect three AS events detected by other tools, probably due to the filtering and preprocessing steps. Considering the scenario without any filtering, each tool reported a higher number of AS events, and the number of events reported by all four tools increased to 43. In this setting, there were no AS events reported exclusively by `pantas`. Note that, without running `whippet`, the number of events reported by the three other tools increases to 61, demonstrating good agreement among the three tools. In this setting, `pantas` failed to detect nine AS events identified by the other tools, even though both `STAR` and `vg mpmap` alignments over the corresponding loci show consistent results, with no support in the control replicates and only minimal support, *i.e.* one or

two alignments, in the knockdown replicates. This also happens for the three events missed by only `pantas` in the scenario with filtered AS events. In detail, two events show no support in the control replicates, while one event is reported with $\Delta\psi = 0.04$. We conjecture `pantas`, with a proper statistical analysis, should be able to detect these events. Considering the filtered dataset again, 10 RT-PCR events were not reported by any tool. These AS events are detected in the no-filtering settings.

Figure 5.14.b reports quantification results. All the correlations are weak, as already noted in literature [39], but `whippet` quantification showed the best correlation with the RT-PCR expected quantification, with a Pearson correlation coefficient of 0.767, followed by `pantas`, with a Pearson correlation coefficient of 0.709. We want to highlight that `whippet` reported fewer events than `pantas`: 40 against 64. On the other hand, despite the distributions of the differences between the RT-PCR $\Delta\psi$ and the quantification provided by `pantas` and `rMATS` being very similar, with just a 0.01 difference in advantage for `pantas`, `rMATS` showed lower correlation than `pantas`. The same trend can be observed in the no-filtering setting, as shown in Figure 5.15.b. One difference is that in the no-filtering experiments, `pantas` quantification and RT-PCR quantification increased, reaching the correlation achieved by `whippet`, which indicates a more statistically robust confidence score for `pantas` quantification.

As a final test, as in [39], we assessed the false positive rate of each tool using 44 RT-PCR–negative exon skipping events. These events are selected because they showed no appreciable change between the two conditions. With filtering, `pantas` reported only one false positive, followed by `SUPPA2` with 28, `whippet` with 32, and `rMATS` with 37. Without filtering, `pantas` did no report any false event, `SUPPA2` 2, `whippet` 4, and `rMATS` 10. Hence, `pantas` can detect a good amount of RT-PCR validated events without introducing false calls. Moreover, being alignment-based, `pantas` is proven to be robust for differential quantification, despite not providing proper statistical validation.

5.4 Conclusions

In this chapter, we discussed two graph-based models for detecting and quantifying alternative splicing events across two conditions.

`ESGq` is a non-pangenomics tool based on read alignment against local graph structures, namely event splicing graphs. Experiments on real data demonstrated that `ESGq` achieves results comparable to other genome–alignment–based approaches, while being an order of magnitude faster. Its main future development, hence considering multiple reference genomes, is already realized by `pantas`; however, outside the pangenomic context, an open challenge remains the statistical validation of `ESGq` results.

Then, we presented `pantas`, which is a pantrascriptome tool based on an annotated spliced pangenome graph. With this method, we can move from a reference-based gene annotation to a haplotype-aware gene annotation, with a multi-genome annotation [207]. As of the date of its publication, `pantas` can be seen as the first pangenomic tool for alternative splicing detection and quantification across RNA-Seq conditions, advancing recent results in literature [119] for spliced pangenomes.

Our experiments show that `pantas` outperforms competing state-of-the-art linear reference-based methods in accuracy, thanks to its formal definition of AS events on a graph structure able to encode the genetic variability of the population under investigation. These results highlight the opportunity to analyze the impact of haplotype-aware annotation on reducing reference bias in AS event quantification.

Despite the positive results of `pantas`, many open questions remain to be addressed. Initially, the modelling should be improved to handle complex and ambiguous cases. For instance, if an in-frame insertion on a haplotype produces a more extended exon, it remains unclear whether this exon should be classified as a novel exon or as an alternative to the "reference" one. This ambiguity also affects AS event detection. Hence, it may necessitate new definitions, such as haplotype-specific AS events, as `pantas` currently extends reference-based AS events definitions to handle haplotype-aware annotations. Note that this non-haplotype-specific definition is an acceptable simplification for real-world scenarios, where, for instance, transcripts coming from different individuals should not be mixed.

Other `pantas` improvements are related to the annotated spliced pangenome graph. Currently, we consider acyclic graphs, but, augmenting the pangenome graph with cycles, we should be able to analyze more complex chromosomal rearrangements, more complex AS events such as mutually exclusive exons, and Local Splicing Variations as in [204, 205].

Moreover, the main limitation of `pantas` is the preprocessing step, specifically the pangenome graph indexing by `vg index`, which is the main bottleneck, followed by the string-to-graph alignments using `vg mpmap`. Currently, we mitigate these limitations without affecting accuracy by simplifying the graph. Still, new graph indexing techniques should be explored for building and indexing a spliced pangenome graph, as well as for aligning RNA-Seq reads to the graph. We should also consider multi-string pangenomic indexes instead of a graph index, such as the r-index [79, 15] and the Movi index [224], which are very efficient on highly repetitive texts, such as sets of haplotypes.

Finally, as for `ESGq`, `pantas` should be extended to include a statistical validation of its results.

Chapter 6

Conclusions and Final Remarks

In this thesis, we presented various techniques to address new pangenomics problems, demonstrating the ability to combine novel theoretical results with ready-to-use bioinformatics/pangenomics tools. We explored two different pangenome representations, the haplotype panel and the pangenome graph, developing novel algorithms and data structures that can be used to solve both computational and biological problems.

Initially, we presented the RLPBWT and the μ-PBWT, aiming to efficiently store and query haplotype panels, which are simplified representations of a pangenome. Strongly inspired by recent results on the run-length encoding of the classical BWT, we generalized the same theoretical mechanism to the positional BWT. Aiming to build a data structure capable of computing SMEMs shared between a panel and an external haplotype in sublinear space, we designed various efficient data structures that require two orders of magnitude less memory than the original PBWT implementation. Moreover, we demonstrated that the μ-PBWT can compute other types of matches, such as k-SMEMs and (k-)MPSC, and it has the potential to be used in more biological scenarios, including genotype phasing.

Moving to the classical pangenome representation, we developed `gindex`, a pangenome graph index based on the multidollar-BWT, which encodes the node labels and, combined with the graph topology, leverages its properties to extend the classical string backward-search algorithm to graphs. We demonstrated that `gindex` is space-efficient at index time compared to the state-of-the-art, despite being slower at query time. We proposed a tool that scales to human pangenome graphs without any strong limitation on query length or available memory, unlike

GCSA2, thereby allowing large graphs to be indexed without any splitting into subgraphs and preserving all information.

Finally, we presented pantas, an AS events detection and quantification tool, the first dedicated tool based on the pangenome graph. To introduce pantas, we also discussed ESGq, a non-pangenomics graph-based model designed to detect and quantify AS events. This was fundamental to prototyping various techniques in a non-pangenomics context before transitioning to the pantascriptomics scenario with pantas. Thanks to ESGq, we were able to explore the use of standard pangenomics tools even in a linear context, facilitating the generalization of the approach to pangenome graphs in pantas. Our experimental results demonstrated that pantas can replace commonly used reference-based tools, providing higher accuracy in AS event detection.

A common future development shared by all the solutions proposed in this thesis is their incorporation into state-of-the-art bioinformatics pipelines, facilitating the transition from a linear reference to a pangenome reference. Hence, future work should focus on demonstrating the effectiveness of these computational approaches in addressing open biological problems, rather than solely improving them from a computational perspective. We plan to conduct further experimental analyses, discussing the results with biologists and collaborating with them on potential extensions of our tools. In any case, since the core of this thesis is computational and theoretical, we also need to address open theoretical problems related to improving the computational bounds of our algorithms, both in time and space. For instance, future work could explore new directions such as including the Move data structures in the μ-PBWT, investigating alternative BWT-based strategies for gindex, and designing a dedicated graph index better suited to the specific requirements of pantas. Finally, we will need to address all the open problems and future directions already reported in Section 3.3, Section 4.3, and Section 5.3.

Overall, we show that pangenomics is a relatively new and rapidly evolving research field, with considerable research activity, which still offers significant opportunities for the development of efficient tools and the extension of existing linear reference-based methods to the pangenomic context. In this thesis, we presented different approaches that help advancing the development and application of pangenomics tools. We can now scale to large biobank datasets with μ-PBWT, index

human pangenome graphs with `gindex`, and analyze AS events using pangenome data with `pantas`. These contributions not only provide practical tools for current biological studies but also lay a theoretical foundation for future research. Hence, we need to optimize computational bounds further, generalize models, and apply these methods to increasingly complex and large real genomic datasets, filling the gap between computational advances and biological discovery.

List of Abbreviations

A3	Alternative 3' splicing/acceptor site
A5	Alternative 5' splicing/donor site
AS	Alternative Splicing
BP	Balanced-Parentheses
BWM	Burrows–Wheeler Matrix
BWT	Burrows–Wheeler Transform
CPS	Complementary Positional Substring
CSA	Compressed Suffix Array
CT	Cartesian tree
DA	Divergence Array
DAG	Directed Acyclic Graph
DEDA	Δ-Encoded Divergence Array
ES	Exon skipping
FL	First-Last
FWBW	ForWard-BackWard
GBWT	Graph Burrows–Wheeler Transform

GCSA	Generalized Compressed Suffix Array
IPA	Inverse Prefix Array
IR	Intron retention
ISA	Inverse Suffix Array
LCE	Longest Common Extension
LCP	Longest Common Prefix (array)
LCS	Longest Common Suffix
LF	Last-First
MEM	Maximal Exact Match
MPSC	Minimal Positional Substring Cover
NSV	Next Smaller Value
PA	Prefix Array
PBWT	Positional Burrows–Wheeler Transform
PERM	Sampled column permutations
PLCP	Permuted Longest Common Prefix (array)
PS	Positional Substring
PSC	Positional Substring Cover
PSV	Previous Smaller Value
RA	Random Access
RLBWT	Run-Length Burrows–Wheeler Transform
RLCP	Reverse Longest Common Prefix
RLPBWT	Run-Length Positional Burrows–Wheeler Transform

RMQ	Range Minimum Query
ROT	Rotation
SA	Suffix Array
SLP	Straight-Line Program
SMEM	Set-Maximal Exact Match

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] Hervé Tettelin, Vega Masignani, Michael J Cieslewicz, Claudio Donati, Duccio Medini, Naomi L Ward, Samuel V Angiuoli, Jonathan Crabtree, Amanda L Jones, A Scott Durkin, et al. Genome analysis of multiple pathogenic isolates of Streptococcus agalactiae: implications for the microbial "pan-genome". *Proceedings of the National Academy of Sciences*, 102(39):13950–13955, 2005.

[2] Jasmijn A Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Natural Computing*, pages 1–28, 2022.

[3] Ting Wang, Lucinda Antonacci-Fulton, Kerstin Howe, Heather A Lawson, Julian K Lucas, Adam M Phillippy, Alice B Popejoy, Mobin Asri, Caryn Carson, Mark JP Chaisson, et al. The Human Pangenome Project: a global resource to map genomic diversity. *Nature*, 604(7906):437–446, 2022.

[4] 1000 Genomes Project Consortium Corresponding author Richard Durbin. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

[5] 1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68, 2015.

[6] Clare Bycroft, Colin Freeman, Desislava Petkova, Gavin Band, Lloyd T Elliott, Kevin Sharp, Allan Motyer, Damjan Vukcevic, Olivier Delaneau, Jared O'Connell, et al. The UK Biobank resource with deep phenotyping and genomic data. *Nature*, 562(7726):203–209, 2018.

[7] All of Us Research Program Genomics Investigators. Genomic data in the all of us research program. *Nature*, 627(8003):340–346, March 2024. Epub 2024 Feb 19.

[8] The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 10 2016.

[9] Simone Rubinacci, Robin J Hofmeister, Bárbara Sousa da Mota, and Olivier Delaneau. Imputation of low-coverage sequencing data from 150,119 UK Biobank genomes. *Nature Genetics*, 55(7):1088–1090, 2023.

[10] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. 1994.

[11] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000.

[12] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 397–406, 2000.

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

[14] Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14:1–15, 2019.

[15] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: a pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 29(2):169–187, 2022.

[16] 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 *2021 Data Compression Conference (DCC)*, pages 193–202. IEEE, March 2021.

[17] Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical computer science*, 698:67–78, 2017.

[18] 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.

[19] Benedict Paten, Adam M Novak, Jordan M Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome research*, 27(5):665–676, 2017.

[20] Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature biotechnology*, 36(9):875–879, 2018.

[21] Paola Bonizzoni, Christina Boucher, Davide Cozzi, Travis Gagie, Dominik Köppl, and Massimiliano Rossi. Data Structures for SMEM-Finding in the PBWT. In *International Symposium on String Processing and Information Retrieval*. Springer Nature Switzerland, 2023.

[22] Guy Jacobson. Space-efficient static trees and graphs. In *30th annual symposium on foundations of computer science*, pages 549–554. IEEE Computer Society, 1989.

[23] Davide Cozzi, Massimiliano Rossi, Simone Rubinacci, Travis Gagie, Dominik Köppl, Christina Boucher, and Paola Bonizzoni. μ-PBWT: a lightweight r-indexing of the PBWT for storing and querying UK Biobank data. *Bioinformatics*, 39(9), 2023.

[24] Paola Bonizzoni, Christina Boucher, Davide Cozzi, Travis Gagie, and Yuri Pirola. Solving the Minimal Positional Substring Cover Problem in Sublinear Space. In *35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024)*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[25] Ahsan Sanaullah, Degui Zhi, and Shaoije Zhang. Haplotype threading using the positional Burrows-Wheeler transform. In *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[26] Bjarni V Halldorsson, Hannes P Eggertsson, Kristjan HS Moore, Hannes Hauswedell, Ogmundur Eiriksson, Magnus O Ulfarsson, Gunnar Palsson, Marteinn T Hardarson, Asmundur Oddsson, Brynjar O Jensson, et al. The sequences of 150,119 genomes in the UK Biobank. *Nature*, pages 1–9, 2022.

[27] Davide Cozzi, Paola Bonizzoni, Christina Boucher, Ben Langmead, and Yuri Pirola. Phasing Data from Genotype Queries via the μ-PBWT. In *The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday (2025)*, pages 10–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.

[28] Brian L Browning, Xiaowen Tian, Ying Zhou, and Sharon R Browning. Fast two-stage phasing of large-scale sequence data. *The American Journal of Human Genetics*, 108(10):1880–1890, 2021.

[29] Brian L Browning, Ying Zhou, and Sharon R Browning. A one-penny imputed genome from next-generation reference panels. *The American Journal of Human Genetics*, 103(3):338–348, 2018.

[30] Jouni Sirén. Indexing variation graphs. In *2017 Proceedings of the ninteenth workshop on algorithm engineering and experiments (ALENEX)*, pages 13–27. SIAM, 2017.

[31] Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020.

[32] Davide Cozzi, Brian Riccardi, Luca Denti, Simone Ciccolella, Kunihiko Sadakane, and Paola Bonizzoni. Pangenome Graph Indexing via the Multidollar-BWT. In *23rd International Symposium on Experimental Algorithms (SEA 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.

[33] Wen-Wei Liao, Mobin Asri, Jana Ebler, Daniel Doerr, Marina Haukness, Glenn Hickey, Shuangjia Lu, Julian K Lucas, Jean Monlong, Haley J Abel, et al. A draft human pangenome reference. *Nature*, 617(7960):312–324, 2023.

[34] Simone Ciccolella, Davide Cozzi, Gianluca Della Vedova, Stephen Njuguna Kuria, Paola Bonizzoni, and Luca Denti. Differential quantification of alternative splicing events on spliced pangenome graphs. *PLOS Computational Biology*, 20(12):e1012665, 2024.

[35] Davide Cozzi, Paola Bonizzoni, and Luca Denti. ESGq: Alternative Splicing Events Quantification across Conditions based on Event Splicing Graphs. In *Proceedings of the 23rd Conference Information Technologies – Applications and Theory, ITAT*, 2023.

[36] Luca Denti, Raffaella Rizzi, Stefano Beretta, Gianluca Della Vedova, Marco Previtali, and Paola Bonizzoni. ASGAL: aligning RNA-Seq data to a splicing graph to detect novel alternative splicing events. *BMC bioinformatics*, 19:1–21, 2018.

[37] Timothy Sterne-Weiler, Robert J. Weatheritt, Andrew J. Best, Kevin C.H. Ha, and Benjamin J. Blencowe. Efficient and Accurate Quantitative Profiling of Alternative Splicing Patterns of Any Complexity on a Laptop. *Molecular Cell*, 72(1):187–200.e6, 2018.

[38] Shihao Shen, Juw Won Park, Zhi-xiang Lu, Lan Lin, Michael D Henry, Ying Nian Wu, Qing Zhou, and Yi Xing. rMATS: robust and flexible detection of differential alternative splicing from replicate RNA-Seq data. *Proceedings of the National Academy of Sciences*, 111(51):E5593–E5601, 2014.

[39] Juan L Trincado, Juan C Entizne, Gerald Hysenaj, Babita Singh, Miha Skalic, David J Elliott, and Eduardo Eyras. SUPPA2: fast, accurate, and

uncertainty-aware differential splicing analysis across multiple conditions. *Genome biology*, 19:1–11, 2018.

[40] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[41] Guy Joseph Jacobson. *Succinct static data structures*. Carnegie Mellon University, 1988.

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

[43] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[44] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *International Symposium on Experimental Algorithms*, pages 295–306. Springer, 2012.

[45] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70. SIAM, 2007.

[46] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

[47] Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, et al. High-order entropy-compressed text indexes. In *SODA*, volume 3, pages 841–850, 2003.

[48] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[49] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[50] Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In *LATIN 2006: Theoretical Informatics: 7th Latin American Symposium,*

Valdivia, Chile, March 20-24, 2006. Proceedings 7, pages 703–714. Springer, 2006.

[51] Francisco Claude, Gonzalo Navarro, and Alberto Ordónez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[52] Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups-Complexity-Cryptology*, 4(2):241–299, 2012.

[53] Travis Gagie, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, Louisa Seelbach Benkner, Yoshimasa Takabatake, et al. Practical random access to SLP-compressed texts. In *International Symposium on String Processing and Information Retrieval*, pages 221–231. Springer, 2020.

[54] Djamal Belazzougui, Patrick Hagge Cording, Simon J Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Algorithms-ESA 2015*, pages 142–154. Springer, 2015.

[55] Moses Ganardi, Artur Jeż, and Markus Lohrey. Balancing straight-line programs. *Journal of the ACM (JACM)*, 68(4):1–40, 2021.

[56] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[57] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4):327–344, 1994.

[58] Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.

[59] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10:1–10, 2009.

[60] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012.

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

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

[63] Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In *International Symposium on String Processing and Information Retrieval*, pages 268–284. Springer, 2018.

[64] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.

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

[66] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[67] Peter Elias. Universal codeword sets and representations of the integers. *IEEE transactions on information theory*, 21(2):194–203, 1975.

[68] Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi. Permuted Longest-Common-Prefix Array. In *Combinatorial Pattern Matching*, pages 181–192. Springer Berlin Heidelberg, 2009.

[69] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *SODA*, volume 2, pages 225–232, 2002.

[70] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

[71] Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original ebwt faster, simpler, and

with less memory. In *International Symposium on String Processing and Information Retrieval*, pages 129–142. Springer, 2021.

[72] Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT of very large string collections. In *2023 Data Compression Conference (DCC)*, pages 71–80. IEEE, 2023.

[73] Davide Cenzato and Zsuzsanna Lipták. A survey of BWT variants for string collections. *Bioinformatics*, 40(7):btae333, 2024.

[74] Veli Mäkinen and Gonzalo Navarro. Run-length FM-index. In *Proc. DIMACS Workshop:"The Burrows-Wheeler Transform: Ten Years Later"(Aug. 2004)*, pages 17–19, 2004.

[75] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56. Springer, 2005.

[76] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *International Symposium on String Processing and Information Retrieval*, pages 164–175. Springer, 2008.

[77] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of individual genomes. In *Research in Computational Molecular Biology: 13th Annual International Conference, RECOMB 2009, Tucson, AZ, USA, May 18-21, 2009. Proceedings 13*, pages 121–137. Springer, 2009.

[78] 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.

[79] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477. SIAM, 2018.

[80] Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.

[81] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[82] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM (JACM)*, 67(1):1–54, 2020.

[83] 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.

[84] 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.

[85] 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(1):1–15, 2019.

[86] Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-indexing the ebwt. In *International Symposium on String Processing and Information Retrieval*, pages 3–12. Springer, 2021.

[87] Heng Li. Tabix: fast retrieval of sequence features from generic TAB-delimited files. *Bioinformatics*, 27(5):718–719, 2011.

[88] Alberto Policriti and Nicola Prezza. From LZ77 to the Run-Length Encoded Burrows-Wheeler Transform, and Back. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:10, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[89] Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling RePair with Rsync. In *Proc. of International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 35–44, 2019.

[90] Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. Minimal positional substring cover is a haplotype threading alternative to Li and Stephens model. *Genome research*, 33(7):1007–1014, 2023.

[91] Ardalan Naseri, Degui Zhi, and Shaojie Zhang. Multi-allelic positional Burrows-Wheeler transform. *BMC bioinformatics*, 20(11):1–8, 2019.

[92] Lucia Williams and Brendan Mumey. Maximal perfect haplotype blocks with wildcards. *Iscience*, 23(6):101149, 2020.

[93] Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Mattia Sgrò. Multiallelic maximal perfect haplotype blocks with wildcards via pbwt. In *International Work-Conference on Bioinformatics and Biomedical Engineering*, pages 62–76. Springer, 2023.

[94] Ardalan Naseri, Erwin Holzhauser, Degui Zhi, and Shaojie Zhang. Efficient haplotype matching between a query and a panel for genealogical search. *Bioinformatics*, 35(14):i233–i241, 2019.

[95] Jarno Alanko, Hideo Bannai, Bastien Cazaux, Pierre Peterlongo, and Jens Stoye. Finding all maximal perfect haplotype blocks in linear time. *Algorithms for Molecular Biology*, 15:1–7, 2020.

[96] Jorge Avila Cartes, Paola Bonizzoni, Simone Ciccolella, Gianluca Della Vedova, and Luca Denti. PangeBlocks: customized construction of pangenome graphs via maximal blocks. *BMC bioinformatics*, 25(1):344, 2024.

[97] Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. d-PBWT: dynamic positional Burrows–Wheeler transform. *Bioinformatics*, 37(16):2390–2397, 02 2021.

[98] Victor Wang, Ardalan Naseri, Shaojie Zhang, and Degui Zhi. Syllable-PBWT for space-efficient haplotype long-match query. *Bioinformatics*, 39(1):btac734, 2023.

[99] Ardalan Naseri, William Yue, Shaojie Zhang, and Degui Zhi. Efficient haplotype block matching in bi-directional PBWT. In *21st International Workshop on Algorithms in Bioinformatics (WABI 2021)*, pages 19–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021.

[100] William Yue, Ardalan Naseri, Victor Wang, Pramesh Shakya, Shaojie Zhang, and Degui Zhi. P-smoother: efficient PBWT smoothing of large haplotype panels. *Bioinformatics Advances*, 2(1):vbac045, 2022.

[101] Rick Wertenbroek, Ioannis Xenarios, Yann Thoma, and Olivier Delaneau. Exploiting parallelization in positional Burrows–Wheeler transform (PBWT) algorithms for efficient haplotype matching and compression. *Bioinformatics advances*, 3(1):vbad021, 2023.

[102] Alan Hodgkinson and Adam Eyre-Walker. Human triallelic sites: evidence for a new mutational mechanism? *Genetics*, 184(1):233–241, 2010.

[103] Ian M Campbell, Tomasz Gambin, Shalini N Jhangiani, Megan L Grove, Narayanan Veeraraghavan, Donna M Muzny, Chad A Shaw, Richard A Gibbs, Eric Boerwinkle, Fuli Yu, et al. Multiallelic positions in the human genome: challenges for genetic analyses. *Human mutation*, 37(3):231–234, 2016.

[104] Jouni Sirén, Jean Monlong, Xian Chang, Adam M Novak, Jordan M Eizenga, Charles Markello, Jonas A Sibbesen, Glenn Hickey, Pi-Chuan Chang, Andrew Carroll, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, 2021.

[105] Ünsal Öztürk, Marco Mattavelli, and Paolo Ribeca. GIN-TONIC: non-hierarchical full-text indexing for graph genomes. *NAR Genomics and Bioinformatics*, 6(4):lqae159, 2024.

[106] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing finite language representation of population genotypes. In *International Workshop on Algorithms in Bioinformatics*, pages 270–281. Springer, 2011.

[107] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012.

[108] Nicolaas Govert De Bruijn. A combinatorial problem. *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, 49(7):758–764, 1946.

[109] Irving John Good. Normal recurring decimals. *Journal of the London Mathematical Society*, 1(3):167–169, 1946.

[110] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[111] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[112] Michael Brudno, Sanket Malde, Alexander Poliakov, Chuong B Do, Olivier Couronne, Inna Dubchak, and Serafim Batzoglou. Glocal alignment: finding rearrangements during alignment. *Bioinformatics*, 19(suppl_1):i54–i62, 2003.

[113] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[114] Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. STAR: ultrafast universal RNA-Seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[115] Kazutaka Katoh, Kazuharu Misawa, Kei-ichi Kuma, and Takashi Miyata. MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucleic acids research*, 30(14):3059–3066, 2002.

[116] Robert C Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792–1797, 2004.

[117] Mikko Rautiainen and Tobias Marschall. GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome biology*, 21(1):253, 2020.

[118] Jorge Avila Cartes, Paola Bonizzoni, Simone Ciccolella, Gianluca Della Vedova, Luca Denti, Xavier Didelot, Davide Cesare Monti, and Yuri Pirola. RecGraph: recombination-aware alignment of sequences to variation graphs. *Bioinformatics*, 40(5):btae292, 2024.

[119] Jonas A Sibbesen, Jordan M Eizenga, Adam M Novak, Jouni Sirén, Xian Chang, Erik Garrison, and Benedict Paten. Haplotype-aware pantranscriptome analyses using spliced pangenome graphs. *Nature Methods*, pages 1–9, 2023.

[120] James D Watson and Francis HC Crick. Molecular structure of nucleic acids: a structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, 1953.

[121] Louise T Chow, Richard E Gelinas, Thomas R Broker, and Richard J Roberts. An amazing sequence arrangement at the 5' ends of adenovirus 2 messenger RNA. *Cell*, 12(1):1–8, 1977.

[122] Susan M Berget, Claire Moore, and Phillip A Sharp. Spliced segments at the 5' terminus of adenovirus 2 late mRNA. *Proceedings of the National Academy of Sciences*, 74(8):3171–3175, 1977.

[123] Douglas L Black. Mechanisms of alternative pre-messenger RNA splicing. *Annual review of biochemistry*, 72(1):291–336, 2003.

[124] Frederick Sanger, Steven Nicklen, and Alan R Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the national academy of sciences*, 74(12):5463–5467, 1977.

[125] J Craig Venter, Mark D Adams, Eugene W Myers, Peter W Li, Richard J Mural, Granger G Sutton, Hamilton O Smith, Mark Yandell, Cheryl A Evans, Robert A Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001.

[126] Alex Bateman and John Quackenbush. Editorial. *Bioinformatics*, 25(4):429–429, 02 2009.

[127] Glennis A Logsdon, Mitchell R Vollger, and Evan E Eichler. Long-read human genome sequencing and its applications. *Nature Reviews Genetics*, 21(10):597–614, 2020.

[128] Hongen Zhang. Overview of sequence data formats. *Statistical genomics: Methods and protocols*, pages 3–17, 2016.

[129] Peter JA Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research*, 38(6):1767–1771, 2010.

[130] Petr Danecek, James K Bonfield, Jennifer Liddle, John Marshall, Valeriu Ohan, Martin O Pollard, Andrew Whitwham, Thomas Keane, Shane A McCarthy, Robert M Davies, and Heng Li. Twelve years of SAMtools and BCFtools. *GigaScience*, 10(2), 02 2021. giab008.

[131] Petr Danecek, Adam Auton, Goncalo Abecasis, Cornelis A Albers, Eric Banks, Mark A DePristo, Robert E Handsaker, Gerton Lunter, Gabor T Marth, Stephen T Sherry, et al. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011.

[132] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and SAMtools. *bioinformatics*, 25(16):2078–2079, 2009.

[133] Gene Myers. GFA: Graphical Fragment Assembly (GFA) Format Specification — gfa-spec.github.io. `https://gfa-spec.github.io/GFA-spec/`. [Accessed 03-06-2025].

[134] Heng Li. A proposal of the Grapical Fragment Assembly format — lh3.github.io. `http://lh3.github.io/2014/07/19/`

a-proposal-of-the-grapical-fragment-assembly-format. [Accessed 03-06-2025].

[135] Heng Li. First update on GFA — lh3.github.io. http://lh3.github.io/2014/07/23/first-update-on-gfa. [Accessed 03-06-2025].

[136] Gene Myers. Graphical Fragment Assembly (GFA) 2.0 Format Specification — gfa-spec.github.io. https://gfa-spec.github.io/GFA-spec/GFA2.html. [Accessed 03-06-2025].

[137] Heng Li. gfatools/doc/rGFA.md at master · lh3/gfatools — github.com. https://github.com/lh3/gfatools/blob/master/doc/rGFA.md. [Accessed 03-06-2025].

[138] Daniel Taliun, Daniel N Harris, Michael D Kessler, Jedidiah Carlson, Zachary A Szpiech, et al. Sequencing of 53,831 diverse genomes from the NHLBI TOPMed Program. *Nature*, 590(7845):290–299, 2021.

[139] Simone Rubinacci, Olivier Delaneau, and Jonathan Marchini. Genotype Imputation using the Positional Burrows Wheeler Transform. *PLoS Genetics*, 16(11):e1009049, 2020.

[140] Kana Shimizu, Koji Nuida, and Gunnar Rätsch. Efficient privacy-preserving string search and an application in genomics. *Bioinformatics*, 32(11):1652–1661, 2016.

[141] Veli Mäkinen and Tuukka Norri. Applying the Positional Burrows-Wheeler Transform to all-pairs Hamming distance. *Information Processing Letters*, 146:17–19, 2019.

[142] Adam M Novak, Erik Garrison, and Benedict Paten. A graph extension of the positional Burrows–Wheeler transform and its applications. *Algorithms for Molecular Biology*, 12:1–12, 2017.

[143] Jiafan Zhu, Dingqiao Wen, Yun Yu, Heidi M Meudt, and Luay Nakhleh. Bayesian inference of phylogenetic networks from bi-allelic genetic markers. *PLoS Computational Biology*, 14(1):e1005932, 2018.

[144] Nádia Pinto, Marta Magalhães, Eduardo Conde-Sousa, Cláudia Gomes, Rui Pereira, Cíntia Alves, Leonor Gusmão, and António Amorim. Assessing paternities with inconclusive STR results: the suitability of bi-allelic markers. *Forensic Science International: Genetics*, 7(1):16–21, 2013.

[145] Na Li and Matthew Stephens. Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. *Genetics*, 165(4):2213–2233, 2003.

[146] William A Freyman, Kimberly F McManus, Suyash S Shringarpure, Ethan M Jewett, Katarzyna Bryc, 23, Me Research Team, and Adam Auton. Fast and robust identity-by-descent inference with the templated positional burrows–wheeler transform. *Molecular Biology and Evolution*, 38(5):2131–2151, 2021.

[147] Yaoling Yang, Richard Durbin, Astrid KN Iversen, and Daniel J Lawson. Sparse haplotype-based fine-scale local ancestry inference at scale reveals recent selection on immune responses. *medRxiv*, pages 2024–03, 2024.

[148] Robin J Hofmeister, Diogo M Ribeiro, Simone Rubinacci, and Olivier Delaneau. Accurate rare variant phasing of whole-genome and whole-exome sequencing data in the UK Biobank. *Nature genetics*, 55(7):1243–1249, 2023.

[149] Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. *CoRR*, abs/2112.04271, 2021.

[150] Nathaniel K Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. *LIPIcs: Leibniz international proceedings in informatics*, 233:16, 2022.

[151] Paola Bonizzoni, Gianluca Della Vedova, Riccardo Dondi, and Jing Li. The haplotyping problem: an overview of computational models and solutions. *Journal of Computer Science and Technology*, 18:675–688, 2003.

[152] Dan Gusfield. Haplotyping as perfect phylogeny: conceptual framework and efficient solutions. In *Proceedings of the sixth annual international conference on Computational biology*, pages 166–175, 2002.

[153] Paola Bonizzoni. A linear-time algorithm for the perfect phylogeny haplotype problem. *Algorithmica*, 48:267–285, 2007.

[154] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[155] Erik D Demaine, Gad M Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014.

[156] Johannes Fischer. Optimal succinctness for range minimum queries. In *Latin American Symposium on Theoretical Informatics*, pages 158–169. Springer, 2010.

[157] Johannes Fischer. Combined data structure for previous-and next-smaller-values. *Theoretical Computer Science*, 412(22):2451–2456, 2011.

[158] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[159] Paweł Gawrychowski, Seungbum Jo, Shay Mozes, and Oren Weimann. Compressed range minimum queries. *Theoretical Computer Science*, 812:39–48, 2020.

[160] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proc. of ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 827–840, 2018.

[161] Takaaki Nishimoto and Yasuo Tabei. Optimal-Time Queries on BWT-Runs Compressed Indexes. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 101:1–101:15, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[162] Nico Bertram, Johannes Fischer, and Lukas Nalbach. Move-r: Optimizing the r-index. In *22nd International Symposium on Experimental Algorithms (SEA 2024)*, pages 1–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[163] Igor Tatarnikov, Ardavan Shahrabi Farahani, Sana Kashgouli, and Travis Gagie. MONI Can Find k-MEMs. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[164] Johannes Köster and Sven Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

[165] James K Bonfield, John Marshall, Petr Danecek, Heng Li, Valeriu Ohan, Andrew Whitwham, Thomas Keane, and Robert M Davies. HTSlib: C library for reading/writing high-throughput sequencing data. *GigaScience*, 10(2), 02 2021. giab007.

[166] Gary K Chen, Paul Marjoram, and Jeffrey D Wall. Fast and flexible simulation of DNA sequence data. *Genome Research*, 19(1):136–142, 2009.

[167] Franz Baumdicker, Gertjan Bisschop, Daniel Goldstein, Graham Gower, Aaron P Ragsdale, Georgia Tsambos, Sha Zhu, Bjarki Eldon, E Castedo Ellerman, Jared G Galloway, Ariella L Gladstein, Gregor Gorjanc, Bing Guo, Ben Jeffery, Warren W Kretzschumar, Konrad Lohse, Michael Matschiner, Dominic Nelson, and Nathaniel S Pope et al. Efficient ancestry and mutation simulation with msprime 1.0. *Genetics*, 220(3), March 2022.

[168] Olivier Delaneau, Jean-Franccois Zagury, Matthew R Robinson, Jonathan L Marchini, and Emmanouil T Dermitzakis. Accurate, scalable and integrative haplotype estimation. *Nature communications*, 10(1):1–10, 2019.

[169] Mark JP Chaisson, Ashley D Sanders, Xuefang Zhao, Ankit Malhotra, David Porubsky, Tobias Rausch, Eugene J Gardner, Oscar L Rodriguez, Li Guo, Ryan L Collins, et al. Multi-platform discovery of haplotype-resolved structural variation in human genomes. *Nature communications*, 10(1):1784, 2019.

[170] Peter Ebert, Peter A Audano, Qihui Zhu, Bernardo Rodriguez-Martin, David Porubsky, Marc Jan Bonder, Arvis Sulovari, Jana Ebler, Weichen Zhou, Rebecca Serra Mari, et al. Haplotype-resolved diverse human genomes and integrated analysis of structural variation. *Science*, 372(6537):eabf7117, 2021.

[171] Heng Li. BGT: efficient and flexible genotype query across many samples. *Bioinformatics*, 32(4):590–592, 2016.

[172] Peter Deutsch. GZIP file format specification version 4.3. Technical report, 1996.

[173] Jean-loup Gailly and Mark Adler. Gnu gzip. *GNU Operating System*, pages 8–18, 1992.

[174] The International Hapmap 3 Consortium. Integrating common and rare genetic variation in diverse human populations. *Nature*, 467(7311):52–58, September 2010.

[175] Peter Edge, Vineet Bafna, and Vikas Bansal. HapCUT2: robust and accurate haplotype assembly for diverse sequencing technologies. *Genome research*, 27(5):801–812, 2017.

[176] Naga Sai Kavya Vaddadi, Taher Mun, and Ben Langmead. Minimizing Reference Bias with an Impute-First Approach. *bioRxiv*, 2023.

[177] Massimo Equi, Veli Mäkinen, and Alexandru I Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. *Theoretical Computer Science*, 975:114128, 2023.

[178] Massimo Equi, Veli Mäkinen, Alexandru I Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Transactions on Algorithms*, 19(3):1–25, 2023.

[179] Lapo Cioni, Veronica Guerrini, Giovanna Rosone, et al. The burrows-wheeler transform of an elastic-degenerate string. In *25th Italian Conference on Theoretical Computer Science (ICTCS)*, volume 3811, pages 66–80, 2024.

[180] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007.

[181] Heng Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014.

[182] Jouni Sirén. Compressed suffix arrays for massive data. In *International Symposium on String Processing and Information Retrieval*, pages 63–74. Springer, 2009.

[183] Trudy FC Mackay, Stephen Richards, Eric A Stone, Antonio Barbadilla, Julien F Ayroles, Dianhui Zhu, Sònia Casillas, Yi Han, Michael M Magwire, Julie M Cridland, et al. The Drosophila melanogaster genetic reference panel. *Nature*, 482(7384):173–178, 2012.

[184] Heng Li, Xiaowen Feng, and Chong Chu. The design and construction of reference pangenome graphs with minigraph. *Genome biology*, 21:1–19, 2020.

[185] Glenn Hickey, Jean Monlong, Jana Ebler, Adam M Novak, Jordan M Eizenga, Yan Gao, Tobias Marschall, Heng Li, and Benedict Paten. Pangenome graph construction from genome alignments with Minigraph-Cactus. *Nature biotechnology*, 42(4):663–673, 2024.

[186] Erik Garrison, Andrea Guarracino, Simon Heumos, Flavia Villani, Zhigui Bao, Lorenzo Tattini, Jörg Hagmann, Sebastian Vorbrugg, Santiago Marco-Sola, Christian Kubica, et al. Building pangenome graphs. *Nature Methods*, pages 1–5, 2024.

[187] Davide Cenzato, Zsuzsanna Lipták, Nadia Pisanti, Giovanna Rosone, and Marinella Sciortino. BWT for String Collections. In Paolo Ferragina, Travis Gagie, and Gonzalo Navarro, editors, *The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday*, volume 131 of *Open Access Series in Informatics (OASIcs)*, pages 3:1–3:29, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[188] Sharvani Mahadevaraju, Soumitra Pal, Pradeep Bhaskar, Brennan D. McDonald, Leif Benner, Luca Denti, Davide Cozzi, Paola Bonizzoni, Teresa M. Przytycka, and Brian Oliver. Diverse somatic Transformer and sex chromosome karyotype pathways regulate gene expression in Drosophila gonad development. *eLife*, 2024.

[189] Andrew Best, Katherine James, Caroline Dalgliesh, Elaine Hong, Mahsa Kheirolahi-Kouhestani, Tomaz Curk, Yaobo Xu, Marina Danilenko, Rafiq

Hussain, Bernard Keavney, et al. Human Tra2 proteins jointly control a CHEK1 splicing switch among alternative and constitutive target exons. *Nature communications*, 5(1):4760, 2014.

[190] Yan Wang, Jing Liu, BO Huang, Yan-Mei Xu, Jing Li, Lin-Feng Huang, Jin Lin, Jing Zhang, Qing-Hua Min, Wei-Ming Yang, et al. Mechanism of alternative splicing and its regulation. *Biomedical reports*, 3(2):152–158, 2015.

[191] Eric T Wang, Rickard Sandberg, Shujun Luo, Irina Khrebtukova, Lu Zhang, Christine Mayr, Stephen F Kingsmore, Gary P Schroth, and Christopher B Burge. Alternative isoform regulation in human tissue transcriptomes. *Nature*, 456(7221):470–476, 2008.

[192] Qun Pan, Ofer Shai, Leo J Lee, Brendan J Frey, and Benjamin J Blencowe. Deep surveying of alternative splicing complexity in the human transcriptome by high-throughput sequencing. *Nature genetics*, 40(12):1413–1415, 2008.

[193] Brenton R Graveley, Angela N Brooks, Joseph W Carlson, Michael O Duff, Jane M Landolin, Li Yang, Carlo G Artieri, Marijke J Van Baren, Nathan Boley, Benjamin W Booth, et al. The developmental transcriptome of Drosophila melanogaster. *Nature*, 471(7339):473–479, 2011.

[194] Sophie C Bonnal, Irene López-Oreja, and Juan Valcárcel. Roles and mechanisms of alternative splicing in cancer—implications for care. *Nature reviews Clinical oncology*, 17(8):457–474, 2020.

[195] A Sveen, S Kilpinen, A Ruusulehto, RA Lothe, and RI Skotheim. Aberrant RNA splicing in cancer; expression changes and driver mutations of splicing factor genes. *Oncogene*, 35(19):2413–2427, 2016.

[196] Giuseppe Biamonti, Angela Amato, Elisa Belloni, Anna Di Matteo, Lucia Infantino, Davide Pradella, and Claudia Ghigna. Alternative splicing in Alzheimer's disease. *Aging clinical and experimental research*, 33:747–758, 2021.

[197] Malini Bhadra, Porsha Howell, Sneha Dutta, Caroline Heintz, and William B Mair. Alternative splicing in aging and longevity. *Human genetics*, 139:357–369, 2020.

[198] Cole Trapnell, David G Hendrickson, Martin Sauvageau, Loyal Goff, John L Rinn, and Lior Pachter. Differential analysis of gene regulation at transcript resolution with RNA-Seq. *Nature biotechnology*, 31(1):46–53, 2013.

[199] Yin Hu, Yan Huang, Ying Du, Christian F Orellana, Darshan Singh, Amy R Johnson, Anaïs Monroy, Pei-Fen Kuan, Scott M Hammond, Liza Makowski, et al. DiffSplice: the genome-wide detection of differential splicing events with RNA-Seq. *Nucleic acids research*, 41(2):e39–e39, 2013.

[200] Jun Cheng, Thi Yen Duong Nguyen, Kamil J Cygan, Muhammed Hasan cCelik, William G Fairbrother, Julien Gagneur, et al. MMSplice: modular modeling improves the predictions of genetic variant effects on splicing. *Genome biology*, 20(1):1–15, 2019.

[201] Núria López-Bigas, Benjamin Audit, Christos Ouzounis, Genís Parra, and Roderic Guigó. Are splicing mutations the most frequent cause of hereditary disease? *FEBS letters*, 579(9):1900–1903, 2005.

[202] Yang I Li, Bryce Van De Geijn, Anil Raj, David A Knowles, Allegra A Petti, David Golan, Yoav Gilad, and Jonathan K Pritchard. RNA splicing is a primary link between genetic variation and disease. *Science*, 352(6285):600–604, 2016.

[203] Andre Kahles, Cheng Soon Ong, Yi Zhong, and Gunnar Rätsch. SplAdder: identification, quantification and testing of alternative splicing events from RNA-Seq data. *Bioinformatics*, 32(12):1840–1847, 2016.

[204] Jorge Vaquero-Garcia, Alejandro Barrera, Matthew R Gazzara, Juan Gonzalez-Vallinas, Nicholas F Lahens, John B Hogenesch, Kristen W Lynch, and Yoseph Barash. A new view of transcriptome complexity and regulation through the lens of local splicing variations. *elife*, 5:e11752, 2016.

[205] Yang I Li, David A Knowles, Jack Humphrey, Alvaro N Barbeira, Scott P Dickinson, Hae Kyung Im, and Jonathan K Pritchard. Annotation-free quantification of RNA splicing using LeafCutter. *Nature genetics*, 50(1):151–158, 2018.

[206] Amit Fenn, Olga Tsoy, Tim Faro, Fanny LM Rößler, Alexander Dietrich, Johannes Kersting, Zakaria Louadi, Chit Tong Lio, Uwe Völker, Jan Baumbach, et al. Alternative splicing analysis benchmark with DICAST. *NAR Genomics and Bioinformatics*, 5(2):lqad044, 2023.

[207] Paulo Amaral, Silvia Carbonell-Sala, Francisco M De La Vega, Tiago Faial, Adam Frankish, Thomas Gingeras, Roderic Guigo, Jennifer L Harrow, Artemis G Hatzigeorgiou, Rory Johnson, et al. The status of the human gene catalogue. *Nature*, 622(7981):41–47, 2023.

[208] Stefano Beretta, Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, and Raffaella Rizzi. Modeling alternative splicing variants from RNA-Seq data with isoform graphs. *Journal of Computational Biology*, 21(1):16–40, 2014.

[209] Stefano Beretta, Paola Bonizzoni, Luca Denti, Marco Previtali, and Raffaella Rizzi. Mapping RNA-Seq data to a transcript graph via approximate pattern matching to a hypertext. In *Algorithms for Computational Biology: 4th International Conference, AlCoB 2017, Aveiro, Portugal, June 5-6, 2017, Proceedings 4*, pages 49–61. Springer, 2017.

[210] Mark F Rogers, Julie Thomas, Anireddy SN Reddy, and Asa Ben-Hur. SpliceGrapher: detecting patterns of alternative splicing from RNA-Seq data in the context of gene models and EST data. *Genome biology*, 13(1):1–17, 2012.

[211] Nuno L Barbosa-Morais, Manuel Irimia, Qun Pan, Hui Y Xiong, Serge Gueroussov, Leo J Lee, Valentina Slobodeniuc, Claudia Kutter, Stephen Watt, Recep Colak, et al. The evolutionary landscape of alternative splicing in vertebrate species. *Science*, 338(6114):1587–1593, 2012.

[212] Kuan-Ting Lin and Adrian R Krainer. PSI-Sigma: a comprehensive splicing-detection method for short-read and long-read RNA-Seq analysis. *Bioinformatics*, 35(23):5048–5054, 2019.

[213] Sebastian Schafer, Kui Miao, Craig C Benson, Matthias Heinig, Stuart A Cook, and Norbert Hubner. Alternative splicing signatures in RNA-Seq data: percent spliced in (PSI). *Current protocols in human genetics*, 87(1):11–16, 2015.

[214] Helga Thorvaldsdóttir, James T Robinson, and Jill P Mesirov. Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration. *Briefings in bioinformatics*, 14(2):178–192, 2013.

[215] Morteza Bajgiran, Azali Azlan, Shaharum Shamsuddin, Ghows Azzam, and Mardani Abdul Halim. Data on RNA-Seq analysis of Drosophila melanogaster during ageing. *Data in brief*, 38:107413, 2021.

[216] Jim Thurmond, Joshua L Goodman, Victor B Strelets, Helen Attrill, L Sian Gramates, Steven J Marygold, Beverley B Matthews, Gillian Millburn, Giulia Antonazzo, Vitor Trovisco, et al. FlyBase 2.0: the next generation. *Nucleic acids research*, 47(D1):D759–D765, 2019.

[217] Quirin Manz, Olga Tsoy, Amit Fenn, Jan Baumbach, Uwe Völker, Markus List, and Tim Kacprowski. ASimulatoR: splice-aware RNA-Seq data simulation. *Bioinformatics*, 37(18):3008–3010, 2021.

[218] Fergal J Martin, M Ridwan Amode, Alisha Aneja, Olanrewaju Austine-Orimoloye, Andrey G Azov, If Barnes, Arne Becker, Ruth Bennett, Andrew Berry, Jyothish Bhai, et al. Ensembl 2023. *Nucleic acids research*, 51(D1):D933–D941, 2023.

[219] Giulio Genovese, Nicole B Rockweiler, Bryan R Gorman, Tim B Bigdeli, Michelle T Pato, Carlos N Pato, Kiku Ichihara, and Steven A McCarroll. BCFtools/liftover: an accurate and comprehensive tool to convert genetic variants across genome assemblies. *Bioinformatics*, 40(2):btae038, 2024.

[220] Rob Patro, Geet Duggal, Michael I Love, Rafael A Irizarry, and Carl Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature methods*, 14(4):417–419, 2017.

[221] Wei Shen, Shuai Le, Yan Li, and Fuquan Hu. SeqKit: a cross-platform and ultrafast toolkit for FASTA/Q file manipulation. *PloS one*, 11(10):e0163962, 2016.

[222] Lucile Broseus and William Ritchie. Challenges in detecting and quantifying intron retention from next generation sequencing data. *Computational and structural biotechnology journal*, 18:501–508, 2020.

[223] Luca Denti, Yuri Pirola, Marco Previtali, Tamara Ceccato, Gianluca Della Vedova, Raffaella Rizzi, and Paola Bonizzoni. Shark: fishing relevant reads in an RNA-Seq sample. *Bioinformatics*, 37(4):464–472, 2021.

[224] Mohsen Zakeri, Nathaniel K Brown, Omar Y Ahmed, Travis Gagie, and Ben Langmead. Movi: a fast and cache-efficient full-text pangenome index. *Iscience*, 27(12), 2024.