University of Verona
Department of Computer Science

Graduate School of Natural and Engineering Sciences
Doctoral Program in Computer science
Cycle XXXV

Processing and indexing large biological datasets using the Burrows-Wheeler Transform of string collections

S.S.D. INF/01

Doctoral student: Davide Cenzato

Supervisor: Zsuzsanna Lipták

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License, Italy. To read a copy of the licence, visit the web page:
http://creativecommons.org/licenses/by-nc-nd/3.0/

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial — You may not use the material for commercial purposes.

NoDerivatives — If you remix, transform, or build upon the material, you may not distribute the modified material.

Processing and indexing large biological datasets using the Burrows-Wheeler Transform of string collections — Davide Cenzato Ph.D. thesis
Verona, September 20, 2023

Abstract

In the last few decades, the advent of next-generation sequencing technologies (NGS) has dramatically reduced the cost of DNA sequencing. This has made it possible to sequence many genomes in very little time, paving the way for projects which aim at the creation of large and repetitive collections of genomic sequences. The abundance of biological data is driving the development of new memory-efficient algorithms and data structures that can scale for large datasets, thus tackling the high computational burden related to processing these data. This trend has a strong impact on the text algorithms area. In this thesis, we will study the Burrows-Wheeler Transform for processing, indexing, and compressing collections of strings.

Data compression addresses the problem of encoding the input to reduce the space needed for storing it, while text indexing focuses on finding ways to efficiently process and extract information from the data. In bioinformatics, these two concepts have been frequently used together since they allow the design of data structures that can efficiently process biological data while keeping the input compressed. The Burrows-Wheeler Transform (BWT) is a reversible transformation on strings introduced by Michael Burrows and David J. Wheeler in 1994 that plays a central role in this area. It is the key component of several compressed data structures for text processing, like the FM-index [Ferraggina and Manzini, SODA, 2000] or the r-index [Gagie et al., SODA, 2018], and some of the most important software in bioinformatics, such as the well-known Bowtie [Langmead et al., Genome Biology, 2009] and BWA [Li and Durbin, Bioinformatics, 2010].

The BWT was originally defined for individual strings, so when the focus moved from single sequences to string collections, there was the need to extend this transform. Over the years, several different tools and algorithms for computing BWT of string collections were introduced. However, even if the transforms generated by these tools frequently differ from each other, the problem of characterizing the BWT variants was never addressed properly.

In this thesis, we close this gap by presenting the first systematic study of the BWT of string collections. We identified five non-equivalent variants computed by the tools in current use and analyzed their properties to show how exactly they differ. We complete our theoretical analysis by comparing the five BWT variants on several real-life biological datasets. We show that not only the differences among the resulting

transforms can be extensive, but they also lead to significant changes in the compressibility of the BWT of the underlying string collection.

As a further complication, the BWT variants in use often depend on the input order of the sequences. This significantly impacts the number of runs r, which defines the size of BWT-based compressed data structures. In this thesis, we address the problem of reordering the input sequences by providing the first implementation of the algorithm of Bentley et al. [ESA 2020], which computes the order minimizing the number of runs of the BWT. This leads to the creation of the first tool for computing the optimal BWT, i.e., the BWT variant which guarantees the minimum number of runs. We show experimentally that the input order can dramatically affect the final result: on our real-life datasets, the optimal BWT had up to 31 times fewer runs than the BWT computed without reordering the input sequences.

The extended BWT (eBWT) of Mantaci et al. [Theor. Comput. Sci. 2007] is one of the first BWT variants explicitly designed to process string collections. Even though this transform is mathematically sound and has useful properties, its construction has been a problem for more than a decade. In this thesis, we present two linear-time algorithms for computing the eBWT of large string collections. The first is an improvement of the Bijective BWT construction algorithm of Bannai et al. [CPM 2019], while the second uses the Prefix-free parsing (PFP) method [Boucher et al., Algorithms Mol. Biol., 2019] to specifically process large and repetitive genomic sequence collections.

In the final part of the thesis, we conclude by studying, for the first time, how to index string collections using the eBWT. We present the extended r-index, an extension of the r-index to the eBWT, which maintains the same performance as the original r-index while inheriting the properties of the eBWT. We implemented this data structure using a variant of the PFP algorithm and tested it on real-life biological datasets containing circular bacterial genomes and plasmids. We show experimentally that our index has competitive query times compared to the r-index on different pattern lengths while supporting advanced pattern matching functionalities on circular sequences.

Abstract (Italian)

Negli ultimi decenni, l'avvento delle tecnologie di sequenziamento di nuova generazione (NGS) ha ridotto drammaticamente il costo del sequenziamento del DNA. Questo ha reso possibile sequenziare molti genomi in pochissimo tempo, lastricando la via a progetti per la creazione di grandi e ripetitive collezioni di sequenze genomiche. L'abbondanza di dati biologici sta guidando lo sviluppo di nuovi algoritmi e strutture dati in grado di scalare per grandi quantità di dati, quindi affrontando l'alto onore computazionale associato ad essi. Questo trend sta avendo forte impatto nell'area degli algoritmi su testo. In questa tesi, studieremo la Burrows-Wheeler Transform per processare, indicizzare e comprimere collezioni di sequenze.

La compressione dati affronta il problema di codificare i dati al fine di ridurre lo spazio necessario a salvarli in memoria, mentre l'indicizzazione si concentra su come estrarre le informazioni dai dati in maniera efficiente. In bioinformatica, questi due concetti sono spesso usati in sinergia in quanto permettono di progettare strutture dati in grado di processare dati biologici in forma compressa. La Burrows-Wheeler Transform (BWT) è una trasformata reversibile su sequenze introdotta da Michael Burrows e David J. Wheeler nel 1994 che gioca un ruolo fondamentale in quest'area di ricerca. La BWT è una componente chiave di molte strutture dati compresse su testi, come l'FM-index [Ferraggina e Manzini, SODA, 2000] o l'r-index [Gagie et al., SODA, 2018], e alcuni dei software più importanti in bioinformatica, come Bowtie [Langmead et al., Genome Biology, 2009] e BWA [Li e Durbin, Bioinformatics, 2010].

La BWT venne definita originariamente per sequenze singole, quindi quando l'attenzione si è spostata da testi singoli a collezioni di sequenze c'è stato il bisogno di estendere questa transformata. Nel corso degli anni, sono stati introdotti molti software e algoritmi per calcolare la BWT di collezioni di sequenze. Però, anche se le trasformate generate da questi software differiscono frequentemente tra di loro, il problema di caratterizzare le varianti della BWT non è mai stato affrontato.

In questa tesi, colmiamo questo vuoto presentando il primo studio sistematico della BWT di collezioni di sequenze. Abbiamo identificato cinque varianti non equivalenti calcolate dai software in uso, e abbiamo analizzato le loro proprietà per mostrare esattamente dove differiscono. Completiamo la nostra analisi teorica comparando le cinque varianti della BWT su molti dataset di dati biologici di uso corrente nella ricerca. Mostriamo che non solo le differenze tra le transformate possono essere

grandi, ma queste possono causare un cambiamento significativo nella compressibilità della BWT della corrispettiva collezione di sequenze.

Come ulteriore complicazione, le varianti della BWT in uso spesso dipendono dall'ordine di input delle sequenze. Questo impatta significativamente il numero di run r, che definisce la dimensione delle strutture dati compresse basate sulla BWT. In questa tesi affrontiamo il problema di ordinare in modo efficiente le sequenze fornendo la prima implementazione dell'algortmo di Bentley et al. [ESA 2020], che calcola l'ordine che minimizza il numero di run della BWT. Mostriamo quindi il primo software per calcolare la optimal BWT, ovvero la BWT di collezioni di sequenze che garantisce il minimo numero di run. Mostriamo sperimentalmente che l'ordine di input può cambiare radicalmente il risultato finale: su i nostri dati reali, l'optimal BWT ha avuto fino a 31 volte meno run della BWT calcolata senza riordinare la sequenze di input.

La extended BWT (eBWT) di Mantaci et al. [Theor. Comput. Sci. 2007] è una della prima varianti della BWT progettata per processare collezioni di sequenze. Anche se questa transformata è matematicamente elegante e mostra proprietà utili, la sua construzione ha rappresentato un problema per più di una decade. In questa tesi, mostriamo due algoritmi lineari per calcolare la eBWT di grandi collezioni di sequenze. Il primo è un miglioramento dell'algoritmo di Bannai et al. per construire la Bijective BWT [CPM 2019], mentre il secondo usa il Prefix-free parsing (PFP) [Boucher et al., Algorithms Mol. Biol., 2019] per processare in modo specifico collezioni di sequenze grandi e ripetitive.

Nell'ultima parte della tesi, concludiamo studiando per la prima volta come indicizzare collezioni di sequenze usando la eBWT. Qui, presentiamo l'extended r-index, un'estensione dell'r-index alla eBWT che mantiene le stesse performance dell'r-index originale ereditando le proprietà della eBWT. Abbiamo implementato questa struttura dati usando una variante dell'algoritmo PFP e l'abbiamo testata su dataset contenenti genomi circolari di batteri e plasmidi. Mostriamo sperimentalmente che il nostro indice ha tempi di query competitivi rispetto all'r-index su pattern di diversa lunghezza, mentre supporta avanzate funzionalità di pattern matching su sequenze circolari.

Acknowledgements

Most importantly, I would like to thank my supervisor, Zsuzsanna Lipták. Zsuzsa introduced me to the beautiful world of string algorithms during my master's degree. She is a passionate teacher, always available to listen and help her students, even if this means going through long and tiring meetings. She guided me through my research path, lighting the way when difficulties looked too hard to overcome. Thanks to her suggestions and remarks, I learned how to be a scientist, ask the right questions, carry out a research project, and present my research results. In a nutshell, I will always be grateful to her for having been my supervisor.

A special thanks go to the reviewers of this thesis, Leena Salmela and Enno Ohlebusch, who gave me useful feedback and many remarks that helped me to improve this document and make the final result a lot better.

Special thanks go to my officemates, Sara, Beatrice, Francesco, and Matteo, who shared with me three years of lunches, coffees, discussions, and laughs. I apologize for the number of questions I have been constantly asking for the whole time. I will always be grateful for their company. I want to thank Max for his patience and guidance; he was always available to listen to me and help me when needed. I am glad I had the chance to work with him. I wish to thank all my friends in the countryside: Mattia, Michele, Jacopo, Marco, Daniela, Davide, Gabriele, Giulia, Emanuele, Letizia, Federico, Elisa, Pier and Susanna. Being a Ph.D. student also means going through hard times comprising frustration and nights spent writing code. Thanks to them, I learned that working is not everything in life and that being carefree every now and then helps make life much more joyful. I want to thank my family: Mom, Dad, Matteo, Bruna, Bepi, Piero, Silvano, Katia, and Gilberto, who always believed in me and supported me both financially and emotionally.

Last but not least, I thank Sofia. It is not easy, to sum up what she means to me in a couple of sentences. I am grateful she has been walking by my side for the last two years. To cite one of my favorite songs, she is the rock upon which I stand. I dedicate this result to her.

Contents

List of Figures II

List of Tables IV

1 Introduction 1
- 1.1 The Burrows-Wheeler Transform of string collections 2
  - 1.1.1 The original Burrows-Wheeler Transform 2
  - 1.1.2 Extending the Burrows-Wheeler Transform 4
- 1.2 Indexing biological string collections 5
  - 1.2.1 A text index based on the extended Burrows-Wheeler Transform 6
  - 1.2.2 Constructing the extended Burrows-Wheeler Transform . 7
- 1.3 Overview of the thesis 8

2 Basics 11

3 An extensive study of the BWT of string collections 17
- 3.1 Related work 18
- 3.2 Overview 19
- 3.3 BWT variants of string collections in the literature 19
  - 3.3.1 Methods to compute the BWT variants 20
- 3.4 The effects of adding separator symbols 21
  - 3.4.1 Interesting intervals 23
  - 3.4.2 The effects on the parameter r 26
- 3.5 Permutations induced by separator-based BWT variants 27
- 3.6 Experimental results 29
  - 3.6.1 Experimental setup 30
  - 3.6.2 Datasets 30
  - 3.6.3 Results 30
- 3.7 Conclusion 31

4 Computing the optimal BWT 37
- 4.1 Overview 38
- 4.2 An algorithm for computing the optimal BWT 39
  - 4.2.1 Computing the optimal BWT using the SAP-array 39

4.3 Adapting SAIS and BCR algorithms . . . 41
4.3.1 Computing the SAP-array using SAIS . . . 41
4.3.2 Computing the SAP-array using BCR . . . 41
4.4 Experimental Results . . . 42
4.4.1 Datasets . . . 42
4.4.2 Experimental setup . . . 42
4.4.3 Results . . . 43
4.5 Conclusion . . . 44

5 Computing the eBWT using SAIS . . . 47
5.1 Overview . . . 48
5.2 A simpler algorithm for computing the eBWT and GCA . . . 48
5.2.1 The eBWT and GCA of non-primitive strings . . . 49
5.3 Cyclic types computation . . . 49
5.4 The Generalized conjugate array computation . . . 51
5.4.1 Handling non-primitive strings in SAIS . . . 52
5.5 Correctness and running time . . . 53
5.6 Computing the BWT for one single string without dollar . . . 55
5.7 The `cais` tool . . . 56
5.7.1 Implementation . . . 56

6 Computing the eBWT of large string collections . . . 59
6.1 Overview . . . 60
6.2 The cyclic prefix-free parsing . . . 60
6.3 Computing the eBWT using the PFP . . . 61
6.3.1 Keeping track of the first rotations . . . 63
6.3.2 Implementation notes . . . 63
6.4 Experimental results . . . 63
6.4.1 Datasets . . . 64
6.4.2 Experimental setup . . . 64
6.4.3 Results . . . 65
6.5 Conclusion . . . 66
6.6 RePairing the PFP . . . 67

7 Constructing the extended r-index . . . 69
7.1 Overview . . . 70
7.2 The extended r-index . . . 70
7.2.1 The data structures for the extended r-index . . . 71
7.2.2 Analysis of the extended r-index . . . 75
7.3 Efficient construction of the extended r-index . . . 77
7.4 Computing MEMs with the extended r-index . . . 79
7.5 Experimental results . . . 81
7.5.1 Implementation . . . 81
7.5.2 Handling equal conjugates . . . 82
7.5.3 Handling non-primitive strings . . . 82
7.5.4 Datasets . . . 82
7.5.5 Experimental setup . . . 83
7.5.6 Results . . . 84
7.6 Conclusion . . . 85

8 Conclusion and suggestions for future research 91
8.1 Directions for future research 92

References 93

Appendices 101

A Full experimental results for Chapter 3 101
A.1 Further information on the tools 101
A.2 Results on individual datasets 103

List of Figures

1.1 Genbank and WGS statistics. . . . 1
1.2 The BWT matrix of BANANAAN. . . . 3
1.3 The BWTs of two permutations of the same string collection. . . . 4
1.4 Backward search example. . . . 6

2.1 Figure for Example 2. . . . 14
2.2 An illustration of the eBWT for the multiset of strings. . . . 15

3.1 Results regarding r on short sequence datasets. . . . 33
3.2 Number of runs of the colexBWT with respect to optimal BWT. 34
3.3 Plot showing the average normalized Hamming distance variations. 35

4.1 The output of different separator-based BWT variants applied to a string collection. . . . 38
4.2 Results regarding the number of runs on three simulated datasets for the optBWT. . . . 45

5.1 Running example for SAIS-for-eBWT algorithm. . . . 54
5.2 Running example for SAIS-for-eBWT algorithm on a single string. . . . 56

6.1 Chromosome 19 dataset construction CPU time and peak memory usage of PFP-eBWT. . . . 66
6.2 Salmonella dataset construction CPU time and peak memory usage of PFP-eBWT. . . . 66
6.3 SARS-CoV2 dataset construction CPU time and peak memory usage of PFP-eBWT. . . . 67

7.1 An illustration of the eBWT for the multiset of strings. . . . 72
7.2 An illustration of the thresholds for calculating the matching statistics of a pattern in a set of strings. . . . 80
7.3 Dispersion plots between the number of occurrences of the r-index and the extended r-index. . . . 86
7.4 Histograms summarizing the number of occurrences difference between the extended r-index and the r-index. . . . 87

7.5 Histograms summarizing the number of matches lost between the extended r-index and the r-index. 88
7.6 Time and space to perform count and locate queries using the extended r-index and the r-index. 89

List of Tables

3.1 The different BWT variants on the multiset $\mathcal{M} = \{\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}\}$. 18
3.2 Example of mdolBWT, and colexBWT for the string collection $\mathcal{M} = \{\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}\}$. 22
3.3 Example of concBWT, and optBWT for the string collection $\mathcal{M} = \{\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}\}$. 23
3.4 Example of eBWT, and dolEBWT for the string collection $\mathcal{M} = \{\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}\}$. 24
3.5 Overview of properties of the five BWT variants. 24
3.6 Percentage of feasible permutations w.r.t. concBWT. 29
3.7 Table summarizing the main parameters of the eight datasets used in the study of the BWT variants to string collections. 32
3.8 Table summarizing the results of the eight datasets used in the study of the BWT variants to string collections. 32
3.9 Results for the SARS-CoV-2 short dataset. 33
3.10 Results for the SARS-CoV-2 genomes dataset. 34

4.1 The mdolBWT, optBWT and SAP-array of the string collection $\mathcal{M} = \{\texttt{TGA}, \texttt{CACAA}, \texttt{AGAGT}, \texttt{TAA}, \texttt{CGAGT}, \texttt{CCA}, \texttt{TA}\}$ 39
4.2 Real-life satasets used in the experiments of the `optimalBWT` tool. 42
4.3 Results on the number of runs increase compared to the optBWT and resource usage. 43
4.4 Results on the number of runs increase factor compared to the optBWT, and resource usage for simulated datasets. 44

6.1 Datasets used in the experiments of the `PFP-eBWT` tool. 64

7.1 Table summarizing the main parameters of the three datasets included in the extended r-index experimental results. 83

A.1 Results for the SARS-CoV-2 short dataset. 104
A.2 Results for the Simons Diversity reads dataset. 105
A.3 Results for the 16S rRNA short dataset. 106
A.4 Results for the Influenza A reads dataset. 107
A.5 Results for the SARS-CoV-2 long dataset. 108

A.6 Results for the 16S rRNA long dataset. 109
A.7 Results for the Candida auris reads dataset 110
A.8 Results for the SARS-CoV-2 genomes dataset. 111

1

Introduction

The big data challenge in bioinformatics consists in finding solutions to several computational problems related to storing, processing, and extracting information from large amounts of biological data. The advent of next-generation sequencing technologies (NGS) caused a drastic increase in the sequencing data by making available machines and software that allow sequencing genomes cheaply and in little time [119], literally in hours [46, 51]. The effects of these new technologies can be seen by looking at the curve showing the amount of biological data in the last 40 years; in particular, the GenBank platform statistics report that the number of sequenced bases has been doubling on average every 18 months since 1982 (see Figure 1.1). This data availability has paved the way for the creation of pangenome datasets [100, 121–123, 125], which could bring important future benefits in several fields, including real-life clinical research scenarios like personalized medicine and rare disease discovery [16]. However, these datasets are usually associated with a very high computational burden, which makes it hard to unlock those benefits.

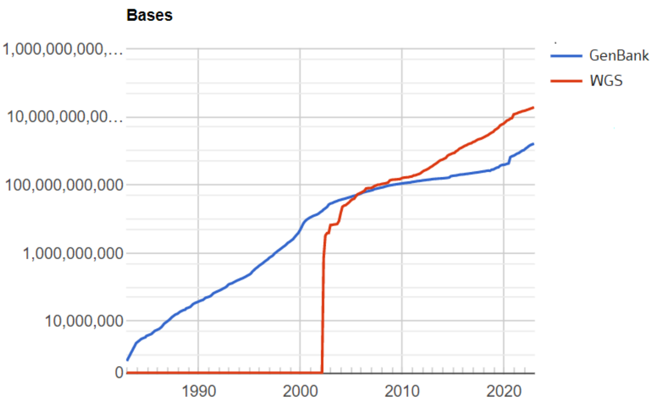


Fig. 1.1: Genbank and WGS statistics (`https://www.ncbi.nlm.nih.gov/genbank/statistics`).

One of the ways to tackle this challenge consists in developing memory-efficient algorithms which scale for large amounts of textual data, usually using compressed data structures that exploit the repetitiveness of biological data. In this thesis, we will bring contributions in the text algorithm area by focusing on three main topics: characterizing the extensions of the Burrows-Wheeler Transform of string collections, studying efficient ways to compute these text transforms, and indexing large genomic string collections using the extended Burrows-Wheeler Transform of Mantaci et al. [87].

1.1 The Burrows-Wheeler Transform of string collections

As we previously mentioned, the increasing availability of low-cost NGS technologies has made it possible to create large string collections of genomic sequences, such as the 1000 Genomes project [122], 10,000 Genomes Project [100], the 100,000 Human Genome Project [125], the 1001 Arabidopsis Project [123], and the 3,000 Rice Genomes Project (3K RGP) [121]. Such pangenome datasets reach sizes of several hundred thousand gigabytes, thus making it unpractical to store or index them as they are. In bioinformatics, this problem is overcome by applying a compression step to reduce the space needed for storing and processing the data.

Luckily, DNA sequences of the same species tend to share a lot of repeated regions. This property makes this kind of data the perfect candidate for compression techniques that aim at taking advantage of repetitions to obtain a more compact representation. The two groups of compression methods that are the most well suited to process biological data are Lempel-Ziv-based [130, 131] compression methods and compression methods based on the Burrows-Wheeler Transform [24]. In this thesis, we focus on the second class of methods due to their connection with text indexing.

1.1.1 The original Burrows-Wheeler Transform

The *Burrows-Wheeler Transform*, or BWT for short, is a reversible transform on strings introduced by Michael Burrows and David J. Wheeler in 1994 [24]. Briefly, the BWT permutes the characters of a string T by sorting lexicographically the *cyclic shifts* (*rotations*) of T and concatenating the last character of each rotation. We can imagine the BWT as the last column of the matrix containing the sorted rotations of T; in particular, the ith character of the BWT is the last character of the ith row (see Figure 1.2). This transformation has two surprising features, namely the *clustering effect* and the *invertibility*.

The sorting procedure brings together rotations starting with the same sequence of characters. It follows that if the indexed text T contains repeated substrings, then the BWT will tend to generate equal-letter runs (clusters), also called *runs*, of the same character. This is because the characters in the last column of the BWT matrix are the left contexts of the ones in the first column, i.e., the ith BWT character precedes in T the ith character of the first column. Thus, if several repeated parts in T are preceded by the same character, their left contexts will form a run in the BWT. Thanks to this property, the BWT tends to create long runs of the same character, making it more

compressible than the original text when applying any simple locally-adaptive compression scheme [11].

Furthermore, if we store the position of the row containing T in the BWT matrix, we can also invert the BWT, i.e., recover the original input T. We obtain it using the so-called *LF-property* of the BWT. The LF-property says that the ith occurrence of a character in the last column corresponds to the ith occurrence of the same character in the first column. Thus, given the ith occurrence of an arbitrary character c in the BWT, we can always find the character that precedes c in the text by locating the position of the ith rotation starting with c in the BWT matrix. Using this property, we can start from the last character of T and compute all preceding characters in a backward fashion.

	sorted rotations	BWT
	1 AANABANAN	**N**
	2 ABANANAAN	**N**
	3 ANAANABAN	**N**
	4 ANABANANA	**A**
	5 ANANAANAB	**B**
→	6 BANANAANA	**A**
	7 NAANABANA	**A**
	8 NABANANAA	**A**
	9 NANAANABA	**A**

Fig. 1.2: Let $T =$ `BANANAAN`, the BWT of T is `NNNABAAAA`, and the position of T in the BWT matrix is $i = 6$. Note that the number of equal-letter runs of T is 7, while it is 4 for the BWT.

Due to these two features, the BWT was originally introduced as a preprocessing step for text compression. In particular, in [24], the authors showed that applying a simple compression scheme such as the move-to-front (MTF) encoding in combination with Huffman or arithmetic coding on the BWT allows to obtain good compression rates, comparable to or better than other state-of-art Lempel-Ziv based compressors such as `gzip`.

However, the BWT is not only useful for compressing data. In fact, this transform has a key feature for text indexing, i.e., it supports efficient pattern matching queries in a small compressed space, usually much smaller than the size of the original input. We had the first example of this functionality in 2000 when Ferragina and Manzini introduced the FM-index showing that it is possible to support efficient pattern matching while keeping the input compressed using the BWT, thus initiating the research area of succinct text indexes based on the BWT [40, 41].

Example 1. The run-length compressed version of the BWT in Figure 1.2 is RLE(`NNNABAAAA`) = `(N,3)(A,1)(B,1)(A,4)`.

A further improvement was introduced in 2005, when Mäkinen et al. [83] proposed the run-length encoding as a compression scheme for the BWT. It is based on encoding the BWT as a list of pairs, one for each run, consisting of a character (*head*) plus the length of the run, so reducing the space needed to store long runs (see Example 1). Even though the run-length encoding is

simpler than classic compression schemes such as the MTF and the LZ parsing, it was shown that on highly repetitive string collections, it provides a better space-time tradeoff than the competitors [83,84,117].

1.1.2 Extending the Burrows-Wheeler Transform

As we previously mentioned, the large availability of genomic data has led to the development of text indexes and compressed data structures specifically designed to handle large string collections of genomic data. This also comprises developing variants of the BWT to string collections to store and index string collections rather than single sequences.

Concurrently, ever increasing dataset sizes have been driving a trend toward ever smaller data structures. Since, the BWT is usually stored in compressed form using the run-length compression scheme, whose effectiveness directly depends on r, much recent research effort has concentrated on the construction of data structures which can not only store but query, process, and mine strings in space and time proportional to r [5,31,44,104].

Moreover, the parameter r is also being increasingly seen as a measure of repetitiveness of the string, with several recent works theoretically exploring its suitability as such a measure, as well as its relationship to other such measures [3,50,95].

	sorted rotations	BWT($\mathcal{M}_1$)		sorted rotations	BWT($\mathcal{M}_2$)
1	$ABN	**N**	1	$NAN	**N**
2	$BAA	**A**	2	$ABN	**N**
3	$NAN	**N**	3	$BAA	**A**
4	A$BA	**A**	4	A$BA	**A**
6	AA$B	**B**	6	AA$B	**B**
7	ABN$	**$**	7	ABN$	**$**
8	AN$N	**N**	8	AN$N	**N**
9	BAA$	**$**	9	BAA$	**$**
10	BN$A	**A**	10	BN$A	**A**
11	N$AB	**B**	11	N$NA	**A**
12	N$NA	**A**	12	N$AB	**B**
13	NAN$	**$**	13	NAN$	**$**

Fig. 1.3: Let $\mathcal{M}_1 = [\texttt{ABN\$}, \texttt{BAA\$}, \texttt{NAN\$}]$ and $\mathcal{M}_2 = [\texttt{NAN\$}, \texttt{ABN\$}, \texttt{BAA\$}]$ be two permutations of the same string collection. We give an example of BWT of string collection for the two inputs (see Chapter 3 for more details). Note the BWT of $\mathcal{M}_1$ has 12 runs, while the BWT of $\mathcal{M}_2$ has 9 runs.

Since the BWT was originally defined for individual sequences, computing the BWT of string collections is not completely straightforward. In fact, there are different strategies being employed to construct the BWT, which, as we will see later in Chapter 3, can generate non-equivalent transforms. These differences also extend to the number r of runs of the BWT. Considering that the data structures based on the BWT rely on small r values to obtain good compression rates and fast query times, studying the relationship between BWT and number of runs is of primary importance. However, the problem of characterizing these differences was never addressed until now in the literature.

As a result, over the years, several different tools and algorithms employing the BWT of string collections were introduced [68, 112] but the method used to compute the BWT is not always clearly stated, and the selection of a particular BWT variant not motivated. All this goes with the tacit assumption that all methods are equivalent. In this thesis, we close this gap by presenting the first systematic study and characterization of the BWT variants of string collections present in the literature. In Chapter 3, we will present five BWT variants we identified in the literature as computed by 15 current tools designed to specifically compute the BWT of string collections. We show that not only can the differences among the BWT variants be extensive, but that the BWT definition chosen can greatly influence the number of equal-letter runs of the BWT r, and so the compressibility of the resulting transform.

This study is even more important considering that the r parameter is also influenced by the input order of the sequences (see example in Figure 1.3). This effect was first described in a 2012 study [32] when the authors showed that applying a specific order to a short string set reduces the number of runs of the corresponding BWT and improves the input's compressibility significantly. However, since then, this problem has been mostly overlooked, and most of the BWT implementations in use do not address the problem of reordering the input sequences.

Recently, Bentley et al. [10] introduced the first linear-time algorithm to compute the permutation of the input collection that yields the minimum r value of the resulting BWT. In this thesis, we give the first implementation of the Bentley et al. algorithm for computing the *optimal BWT* (optBWT), i.e., the BWT of a string collection that guarantees the minimum number of runs. In Chapter 4, we present a framework for constructing the optimal BWT, starting from a BWT of the collection and permuting its characters. We implemented it in the `optimalBWT` tool via an adaptation of two algorithms, SAIS [99] and BCR [8]. We tested our tool on real-life and simulated data and found that reordering the input sequences can dramatically affect the resulting BWT. On our data, the optimal BWT has up to 31 times fewer runs than the BWT computed with a random input order.

1.2 Indexing biological string collections

In the second part of the thesis, we focus on indexing string collections using the extended Burrows-Wheeler Transform of Mantaci et al. [87]. Text indexing studies how to efficiently extract information from the data. In strings, the most fundamental problem is finding the occurrences of a short string, called *pattern*, in a text that is assumed to be much larger than the pattern. This problem is called *string matching* or *pattern matching* and is solved efficiently using a text index. In this way, we can preprocess the input, building a data structure that allows answering each pattern matching query in time proportional to the length of the pattern. Without such an index, we would need to use algorithms that have query times proportional to the length of the text [2, 23, 62].

On large inputs, text indexes allow saving a lot of time by using additional space in memory to store the necessary data structures [96]. In recent research,

data compression is employed to keep these data structures as small as possible, thus allowing the processing of biological data while keeping the input compressed [96]. The BWT plays a central role in this area. In particular, it is the key component of several compressed data structures for text processing, like the FM-index [41], the run-length FM-index [83] and r-index [44], and some of the most popular software in bioinformatics to perform sequence alignment, like the well-known Bowtie [69, 70], and BWA [72, 74].

The popularity of the BWT is due to its ability to support fast pattern matching queries. This is obtained as means of the so-called *backward search* algorithm [40]. When searching for a pattern, the backward search exploits the property that all rotations starting with the same substring are grouped together in the BWT matrix. It follows that we can identify an interval in the BWT $[b..e]$ containing all rotations starting with a certain suffix of the pattern $P[i..]$. For instance, in Figure 1.2, all rotations starting with $P[2..] =$ A are in the interval $[1..5]$. Since the BWT characters are the left contexts of the characters in the first column, we can left extend $P[i..]$ with the character $P[i-1]$ using the LF-mapping on the BWT (see Figure 1.4 for an example).

	sorted rotations	BWT
1	**A**ANABANAN	**N**
2	**A**BANANAAN	**N**
3	**A**NAANABAN	**N**
4	**A**NABANANA	**A**
5	**A**NANAANAB	**B**
6	BANANAANA	**A**
7	**NA**ANABANA	**A**
8	**NA**BANANAA	**A**
9	**NA**NAANABA	**A**

Fig. 1.4: Continuing example in Figure 1.2. Backward search step example for the pattern $P =$ NA. We left-extend the pattern suffix $P[2..] =$ A with the N character.

Backward search has two major advantages: given a pattern $P[1..m]$, it runs in $O(m)$ time by computing m backward search steps. This is the same query time complexity guaranteed by other text indexes such as the suffix tree [127]. Moreover, it can be computed on a BWT in compressed form by using some additional small auxiliary data structures [54, 92]. In this case, it is important to employ a compression scheme like the run-length encoding that offers good compression rates while still allowing efficient implementation of the backward search.

1.2.1 A text index based on the extended Burrows-Wheeler Transform

In 2005, Mäkinen and Navarro [83] introduced the run-length FM-index (RLFM-index), the first text index implementing the backward search in run-length compressed space. However, even though this data structure supports count queries in space proportional to r, locating patterns still requires a sampling or compression scheme for the suffix array (SA) [86], i.e., the array containing the

indexes of the text suffixes in lexicographic order. However, these schemes do not scale well for large string collections.

In 2018 Gagie et al. solved this problem with the introduction of the r-index [44,45], the first data structure supporting count and locate queries whose size and time requirements can be described entirely using the number of runs of the BWT. This result was obtained by using the Toehold lemma [108] to augment the backward search algorithm and a new SA sampling scheme which ensures small memory requirements in case of repetitive texts.

During the development of all these data structures, however, little attention was paid to the fact that the input nowadays is typically a string collection (i.e., a multiset of strings), rather than an individual sequence. Indeed, this kind of input is effectively just treated as if it was a single sequence: most tools computing BWT of string collection just concatenate the input, adding string separator symbols to mark string boundaries. In Chapter 3, we will see that there are different ways to concatenate sequences; in fact, the output can vary significantly, and this extends to the number r of runs of the BWT. Moreover, the concatenation methods depend on the input order: if the same string collection is presented in a different order to the same tool, the text index produced may be different, including the number of runs. As a consequence, this may result in a big variation in the memory requirement of the resulting data structures, which is measured using r.

The *extended Burrows-Wheeler Transform* (eBWT) proposed by Mantaci et al. in 2007 [87], is a BWT of string collection (see Chapter 2 for the complete definition), which offers a solution to the problem of the variation of the number of runs. In particular, the eBWT is independent of the input order and thus does not suffer from the above shortcoming. The eBWT also processes every input string as a circular string without adding string separators, thus supporting advanced pattern matching functionalities. However, even though eBWT has useful properties and is the mathematically cleanest BWT extension, its application in text indexing has never been properly investigated.

In this thesis, we close this gap by presenting an extension of the r-index to the eBWT we called *extended r-index*. In Chapter 7, we detail how to construct the extended r-index, maintaining the same core functionalities as the original r-index while inheriting the properties of the eBWT. We note that our text index is the first BWT-based compressed data structure whose space requirement is the same for all input orders. We tested the extended r-index on three genomic datasets containing circular bacterial and plasmid genomes for patterns of different lengths. In all cases, our data structure obtained competitive or better query times compared to the r-index.

1.2.2 Constructing the extended Burrows-Wheeler Transform

Text indexing is not only about fast query times; another fundamental problem is constructing the index efficiently. In particular, for the extended r-index, an essential step consists of the computation of the eBWT of string collections. However, until recently, no efficient construction algorithm for computing the eBWT was known, which may be the reason why it has not been the method of choice for most tools. In 2019 Bannai et al. [7] introduced a linear-time algorithm

for computing the Bijective BWT using an adaptation of the SAIS algorithm by Nong et al. [99]. As a by-product, this algorithm can also construct the eBWT of a string collection, thus becoming the first linear-time algorithm for computing the eBWT. However, it requires a preprocessing step that can be demanding for large datasets.

In this thesis, we present a new algorithm named `SAIS_for_eBWT` that improves and simplifies the algorithm of Bannai et al., removing the need for preprocessing. We implemented this algorithm and included it in the `cais` tool. As a by-product of this result, we obtain the first linear time algorithm for computing the BWT of a single sequence that uses neither end-of-string characters nor Lyndon words.

When computing the BWT of very large datasets like pangenome datasets, it is not always possible to keep the computation in internal memory; this creates a serious obstacle for the computation of text indexes for such datasets. In the last decade, much effort was put into the development of efficient algorithms and tools to construct the BWT of very large string collections. There are different successful strategies used to address the dataset size problem, including (i) keeping part of the data structures needed by the algorithm in external memory (`BCR`) [8], (ii) keeping the growing BWT in the internal memory in compressed form (`ropeBWT2`) [73] and (iii) computing the BWT starting from a compressed version of the input which fits in the internal memory (`Big-BWT`) [17].

In 2019, Boucher et al. [21] introduced the first algorithm for computing large BWTs which keeps all computation in internal memory. It is based on the Prefix-free parsing (PFP) technique, a prepossessing step for compressing large datasets. The PFP algorithm shows excellent performance on repetitive genomic string collections and is employed as a tool to support the efficient construction of the r-index [68].

In this thesis, we present an extension of the PFP algorithm for computing the extended BWT of Mantaci et al. [87]. In Chapter 6, we present how to reach this goal by combining our `SAIS_for_eBWT` algorithm with the PFP preprocessing. We implemented this algorithm in `pfpebwt`, the first tool for computing the eBWT of large string collections. We tested it on real-life genomic data and showed that it is competitive with other tools designed to construct the BWT of large string collections. Finally, we collaborated on developing an algorithm inspired by the well-known RePair by Larsson and Moffat [71] for optimizing the PFP size. We show that it can reduce the combined size of the PFP data structures significantly on real-life datasets.

1.3 Overview of the thesis

The thesis comprises two main parts: Chapters 3, 4 deal with characterizing the BWT variants of string collections and computing the BWT variant which guarantees the minimum number of runs, while Chapters 5, 6, and 7 deal with constructing and indexing string collections using the eBWT.

In Chapter 2, we give all definitions and background necessary for the following chapters.

In the first part of this thesis, we show that different ways to extend the BWT to string collections generate non-equivalent transforms. In Chapter 3, we

present the first systematic study of the BWT variants for string collections. We define five BWT variants we identified in the literature and present where and how much exactly they differ. The chapter also includes experiments on several real-life genomic datasets. In Chapter 4, we present the first tool for computing the optimal BWT of string collections, i.e., the BWT of string collections that guarantees the minimum number of runs. We also include experiments giving the reduction in the number of runs of the optimal BWT compared to the BWT computed without reordering the input sequences.

In the second part of this thesis, we describe how to define an extension of the r-index based on the extended BWT and how to construct it. In Chapter 5, we give details of `SAIS_for_eBWT`, a linear-time algorithm for computing the eBWT of string collections. We also describe `cais`, the tool containing the implementation of this algorithm. In Chapter 6, we present `pfpebwt`, the first tool computing the eBWT of very large string collections based on the PFP preprocessing. We also include experiments on real-life genomic datasets assessing the performance of our tool compared to other tools currently in use. Finally, in Chapter 7, we present how to extend the r-index of Gagie et al. to define the extended r-index, the first text index built on the eBWT. We also detail an efficient implementation of this new data structure and show how to construct it using our `pfpebwt` tool.

We conclude in Chapter 8, with an outlook to future research directions.

The contents of Chapters 3, 4, 5, 6, 7 have been published in refereed conference proceedings. An extended version of Chapter 7 is under review for publication. Special thanks go to all co-authors of these papers and my two reviewers, Leena Salmela and Enno Ohlebusch, whose remarks made this thesis a lot better.

2

Basics

Let Σ be a finite ordered alphabet of size σ. We use the notation $T = T[1..n]$ for a string T of length n over Σ, $T[i]$ for the ith character, and $T[i..j]$ for the substring $T[i] \cdots T[j]$ of T, where $i \leq j$, and $1 \leq i, j \leq n$; $|T|$ denotes the length of T, and ε the empty string. We refer to $T[i..j]$ as a *substring* (or *factor*) of T, to $T[1..j]$ as the j-th *prefix* of T, and to $T[i..n] = T[i..]$ as the i-th *suffix* of T. A substring S of T is called *proper* if $T \neq S$. Given a positive integer $i \leq n$ and a symbol $c \in \Sigma$, a rank query $rank_c(T, i)$ returns the number of occurrences of c in the prefix $T[1..i]$, while $select_c(T, i)$ returns the position of the ith occurrence of c in T. Given two strings S and T, we denote by $\texttt{lcp}(S, T)$ the length of the *longest common prefix* of S and T, i.e., $\texttt{lcp}(S, T) = \max\{i \mid S[1..i] = T[1..i]\})$. Given a string T and an integer $m > 0$, we refer to T^m as the m-fold concatenation of T, and to T^ω as the infinite string $TT \cdots$ obtained by concatenating an infinite number of copies of T.

Every string T can be written uniquely as $T = U^m$, where U is primitive. We refer to U as the *root* of T, $\text{root}(T)$, and to m as the *exponent* of T, $\exp(T)$, i.e. $T = \text{root}(T)^{\exp(T)}$. A string T is called *primitive* if $T = U^m$ implies $T = U$ and $m = 1$. A *run* in a string T is a maximal substring consisting of the same character; we denote by $\text{runs}(T)$ the number of runs of T. An end-of-string character, usually denoted by `$`, is a special character appended at the end of T; this character is not an element of Σ and is assumed to be smaller than all characters from Σ. Note that appending a `$` makes any string primitive.

The string S is a *conjugate* of the string T if $S = T[i..n]T[1..i-1]$, for some $1 \leq i \leq n$ (also called the *i-th rotation* of T). The conjugate S is also denoted $\text{conj}_i(T)$. The string T is primitive if and only if it has n distinct conjugates. A *Lyndon word* is a primitive string that is lexicographically smaller than all of its conjugates. Given a string T, U is a *circular* or *cyclic substring* of T if it is a factor of TT of length at most $|T|$, or equivalently if it is the prefix of some conjugate of T. For example, `ATA` is a cyclic substring of `AGCAT`. A string $T[1..n]$ can also be regarded as a *circular* or *cyclic* string; in this case we set $T[0] = T[n]$ and $T[n+1] = T[1]$.

For two strings S, T, the *(unit-cost) edit distance* $\text{dist}_{\text{edit}}(S, T)$ is defined as the minimum number of operations necessary to transform S into T, where an operation can be deletion or insertion of a character, or substitution of a

character by another. The *Hamming distance* $\mathrm{dist}_{\mathrm{H}}(S,T)$, defined only if $|S| = |T|$, is the number of positions i such that $S[i] \neq T[i]$.

The order relations

The *lexicographic order* on Σ^* is defined by $S <_{\mathrm{lex}} T$ if S is a proper prefix of T, or if there exists an index j s.t. $S[j] < T[j]$ and for all $i < j$, $S[i] = T[i]$. The *colexicographic order*, or *colex-order* (referred to as *reverse lexicographic order* in [32, 73]) is defined by $S <_{\mathrm{colex}} T$ if $S^{\mathrm{rev}} <_{\mathrm{lex}} T^{\mathrm{rev}}$, where $X^{\mathrm{rev}} = X[n]X[n-1] \cdots X[1]$ denotes the reverse of the string $X = X[1..n]$.

The *omega order* [47, 87], or ω*-order* is defined by $S \prec_\omega T$ if $\mathrm{root}(S) = \mathrm{root}(T)$ and $\exp(S) < \exp(T)$, or $S^\omega <_{\mathrm{lex}} T^\omega$ (this implies $\mathrm{root}(S) \neq \mathrm{root}(T)$). One can easily verify that the ω-order relation is different from the lexicographic one. For example, $CG <_{\mathrm{lex}} CGA$ but $CGA \prec_\omega CG$.

The Suffix Array

Given a string $T[1..n]$, the *suffix array* [86], denoted by $\mathrm{SA} = \mathrm{SA}_T$, is defined as the permutation of $\{1, \ldots, n\}$ such that $T[\mathrm{SA}[i]..]$ is the i-th lexicographically smallest non-empty suffix of T, $T[\mathrm{SA}[i]..] <_{lex} T[\mathrm{SA}[i+1]..]$. For an example, see Fig 2.1. Given the SA of T, we denote the *inverse suffix array* as ISA, and define it as $\mathrm{ISA}[\mathrm{SA}[i]] = i$ for all $i = 1, \ldots, n$. In several data structures (such as the FM-index [41] or the r-index [45]), instead of storing the entire array SA, only a proper subset is stored; this will be referred to as an SA-*sample*.

A basic property of the SA is that, given a substring P of T, the set of occurrences of P appear as an interval in SA. Thus, P defines a unique interval $[s_P, e_P]$ of SA, namely $\mathrm{SA}[s_P]$ is the lexicographically least suffix and $\mathrm{SA}[e_P]$ the lexicographically largest suffix which have P as a prefix.

The Conjugate Array and the Burrows-Wheeler Transform

Given a string $T[1..n]$, the *conjugate array*, also called *circular suffix array* in [7, 58] and *BW-array* in [66, 106], $\mathrm{CA} = \mathrm{CA}_T$ of T is defined as the permutation of $\{1, \ldots, n\}$ such that $\mathrm{CA}[i] = j$ if $\mathrm{conj}_j(T)$ is the i-th conjugate of T with respect to the lexicographic order, with ties broken according to string order, i.e., if $\mathrm{CA}[i] = j$ and $\mathrm{CA}[i'] = j'$ for some $i < i'$, then either $\mathrm{conj}_j(T) <_{\mathrm{lex}} \mathrm{conj}_{j'}(T)$, or $\mathrm{conj}_j(T) = \mathrm{conj}_{j'}(T)$ and $j < j'$. Note that if T is a Lyndon word, then $\mathrm{CA}[i] = \mathrm{SA}[i]$ for all $1 \leq i \leq n$ [48].

The *Burrows-Wheeler Transform* [24], denoted as BWT, is a reversible transformation extensively used in data compression. Given a string $T[1..n]$, $\mathrm{BWT}(T)$ is a permutation of the letters of T defined as the concatenation of the last characters of all lexicographically sorted conjugates of T. For an example, see Fig 2.1. The mapping $T \mapsto \mathrm{BWT}(T)$ is reversible, up to rotation, and can be made uniquely reversible by adding to $\mathrm{BWT}(T)$ an index indicating the rank of T in the lexicographic order of all of its conjugates. Given $\mathrm{BWT}(T)$ and an index i, the original string T can be computed in linear time [24]. The BWT itself can be computed from the conjugate array since for all $i = 1, \ldots, n$, $\mathrm{BWT}(T)[i] = T[\mathrm{CA}[i] - 1]$, where T is considered to be cyclic, so if $\mathrm{CA}[i] = 1$ then $\mathrm{BWT}(T)[i] = T[n]$.

The end-of-string character

It should be noted that in many applications, it is assumed that an end-of-string character, denoted as \$, is appended at the end of the string. Since $T\$$ is the only conjugate ending with \$, $\mathrm{BWT}(T\$)$ is now uniquely reversible without the need for the additional index i. Moreover, adding a final \$ makes the string primitive, and $\$T$ is a Lyndon word. Therefore, computing the conjugate array becomes equivalent to computing the suffix array, since $\mathrm{CA}_{T\$}[i] = \mathrm{SA}_{T\$}[i]$. Thus, applying one of the linear-time suffix array computation algorithms [94, 111] leads to linear-time computation of the BWT.

When no \$-character is appended to the string, the situation is slightly more complex. For primitive strings T, first the Lyndon conjugate of T has to be computed (in linear time, [115]) and then a linear-time suffix array algorithm can be employed [48]. For strings T which are not primitive, one can take advantage of the following well-known property of the BWT: let $T = S^k$ and $\mathrm{BWT}(S) = U[1..m]$, then $\mathrm{BWT}(T) = U[1]^k U[2]^k \cdots U[m]^k$ (Prop. 2 in [88]). Thus, in order to compute $\mathrm{BWT}(T)$, it suffices to compute the BWT of $\mathrm{root}(T)$. The root of T can be found by computing the border array b of T: T is not primitive if and only if $n/(n - b[n])$ is an integer, which is then also the length of $\mathrm{root}(T)$. For example, the border array can be computed by the preprocessing phase of the KMP-algorithm for pattern matching [62], in linear time in the length of T.

The LF- and ϕ-mappings

The LF-*mapping* (last-to-first mapping) [24] plays a central role in BWT-based algorithms. Given a string S of length n, the LF-mapping is a permutation of $\{1, \ldots, n\}$ defined as: $\mathrm{LF}(j) < \mathrm{LF}(j')$ if $S[j] < S[j']$ or $S[j] = S[j']$ and $j < j'$. This permutation is also called *standard permutation of S* [78]; when S is the BWT of some primitive string, then the lexicographically smallest such T can be constructed using the LF-mapping as follows [24]: $T[n] = S[1]$ and for $i \geq 1$, $T[n-i] = T[\mathrm{LF}^i(1)]$. In particular, the LF-mapping allows to walk through the original string T, in a back-to-front direction, by mapping the lexicographically jth conjugate $\mathrm{conj}_i(T)$ to the lexicographic rank of $\mathrm{conj}_{i-1}(T)$. The same mapping is fundamental for *backward search* [41], the efficient pattern matching algorithm on BWT-based data structures.

Another fundamental permutation is the ϕ-*mapping* introduced in [60]. This permutation is defined as $\phi(i) = \mathrm{SA}[\mathrm{ISA}[i] - 1]$ if $\mathrm{ISA}[i] > 1$, and $\phi(i) = \mathrm{SA}[n]$ otherwise. In other words, ϕ maps the ith suffix $T[i..]$ to the lexicographically next smaller suffix. In terms of the suffix array, this can be expressed as: $\phi(\mathrm{SA}[1]) = \mathrm{SA}[n]$ and $\phi(\mathrm{SA}[j]) = \mathrm{SA}[j-1]$, for $j > 1$. See Figure 2.1 for an example. Note that the ϕ-function can be similarly defined on the basis of the conjugate array CA, and the LF-mapping on that of the SA if an end-of-string marker is present.

Example 2. Let $T = \texttt{GATAT}$, and $T' = \texttt{GATAT\$}$ be two input strings. In Figure 2.1, for each string, we list its BWT, LF-mapping, and ϕ. We also give the CA and the SA for T and T', respectively.

$T = \texttt{GATAT}$

i	ϕ_T	CA_T	LF_T	BWT_T	
1	4	2	3	G	ATATG
2	5	4	4	T	ATGAT
3	1	1	5	T	GATAT
4	2	3	1	A	TATGA
5	3	5	2	A	TGATA

$T' = \texttt{GATAT\$}$

i	$\phi_{T'}$	$\text{SA}_{T'}$	$\text{LF}_{T'}$	$\text{BWT}_{T'}$	
1	2	6	5	T	$GATAT
2	4	4	6	T	AT$GAT
3	5	2	4	G	ATAT$G
4	6	1	1	$	GATAT$
5	1	5	2	A	T$GATA
6	3	3	3	A	TAT$GA

Fig. 2.1: Figure for Example 2.

The r-index

A fundamental parameter of the BWT is the number of runs r. For example, in Figure 2.1, $r = 3$ for the string $T = \texttt{GATAT}$ since $BWT(T) = \texttt{GTTAA}$. The memory required by the run-length encoded version of the BWT is $O(r)$. The r-index [45], building on the run-length encoded FM-index of Mäkinen and Navarro [83, 84] and the Toehold Lemma of Policriti and Prezza [108], is a data structure that not only requires $O(r)$ memory only, but supports pattern matching queries in time which depends almost entirely on the size of the pattern. In detail, let w be the size of a computer word. Given a *pattern* P of length p, the r-index returns the number *occ* of occurrences of P in $O(p \log\log_w(\sigma + n/r))$ time (a *count query*), and, after having answered the count query, returns all *occ* occurrences of P in $O(\log\log_w(n/r))$ time per occurrence. In other words, it answers *locate queries* in total time $O(\log\log_w(\sigma + n/r) + occ \log\log_w(n/r))$ time.

Two crucial ideas are combined in the workings of the r-index: (1) the Toehold Lemma [108] allows to produce *one* occurrence along with answering a count query (the *toehold value*), and (2) repeated application of the ϕ-function returns all occurrences contained in the SA-interval of P.

The r-index consists of three main components: (1) a data structure that stores the run-length encoded BWT supporting LF-mapping queries, (2) an SA-sample for each of the r runs, and (3) a data structure supporting ϕ operations. In particular, in [45], (1) builds on the RLFM-index of Mäkinen et al. [84] combined with the data structures of Belazzougui and Navarro [9], while (2) is an array storing the SA-samples at the end of each run. Finally, (3) is implemented as a predecessor data structure built on SA-samples at the beginning of each

run, with the corresponding SA-sample at the end of the previous run as satellite information.

The Generalized Conjugate Array and extended Burrows-Wheeler Transform

Given a multiset of strings $\mathcal{M} = \{T_1[1..n_1], \ldots, T_m[1..n_m]\}$, we denote the total length of the strings in $\mathcal{M}$ as $||\mathcal{M}||$, i.e., $||\mathcal{M}|| = |T_1| + \ldots + |T_m|$. Alternatively, we denote $N = ||\mathcal{M}||$. The *generalized conjugate array* of $\mathcal{M}$, denoted by $\mathrm{GCA}_{\mathcal{M}}$ or just by GCA, contains the list of the conjugates of all strings in $\mathcal{M}$, sorted according to the ω-order relation. More formally, $\mathrm{GCA}[i] = (d, j)$ if $\mathrm{conj}_j(T_d)$ is the i-th string in the $\preceq_\omega$-sorted list of the conjugates of all strings of $\mathcal{M}$, with ties broken first w.r.t. the index of the string (in case of identical strings), and then w.r.t. the index in the string itself. The *text order* for $\mathcal{M}$ is defined by $g <_{text} g'$, where $g = (d, j)$ and $g' = (d', j')$, if $d < d'$ or $d = d'$ and $j < j'$. Given a string $P[1..p]$ (the *pattern*), we say that P *occurs* in $\mathcal{M} = \{T_1, T_2, \ldots, T_m\}$ if P occurs as a cyclic substring of one of $T_1, T_2, \ldots, T_m$. More formally, we define an *(cyclic) occurrence* of P in $\mathcal{M}$ as a pair (d, j) such that P is a prefix of $\mathrm{conj}_j(T^d)$.

$$\mathcal{M} = \{\texttt{AAT}, \texttt{TAGA}, \texttt{AT}\}$$

i	$\mathrm{GCA}_{\mathcal{M}}$	$\mathrm{LF}_{\mathcal{M}}$	$\mathrm{eBWT}_{\mathcal{M}}$	ω-sorted conjugates
1	(1,1)	7	T	AAT
2	(2,2)	8	T	AGAT
3	(1,2)	1	A	ATA
4	(2,4)	6	G	ATAG
5	(3,1)	9	T	AT
6	(2,3)	2	A	GATA
7	(1,3)	3	A	TAA
8	(2,1)	4	A	TAGA
9	(3,2)	5	A	TA

Fig. 2.2: An illustration of the eBWT for the multiset of strings $\mathcal{M}$. From left to right, we report the index i, the generalized conjugate array GCA for $\mathcal{M}$, the LF permutation, the eBWT, and the conjugates of $\mathcal{M}$ sorted according to the ω-order.

The *extended Burrows-Wheeler Transform* (eBWT) is an extension of the BWT to a multiset of strings [87]. It is a bijective transformation that, given a multiset of strings $\mathcal{M} = \{T_1, \ldots, T_m\}$, produces a permutation of the characters on the strings in the multiset $\mathcal{M}$. Formally, $\mathrm{eBWT}(\mathcal{M})$ can be computed by sorting all the conjugates of the strings in the multiset according to the $\preceq_\omega$-order, and the output is the string obtained by concatenating the last character of each conjugate in the sorted list, together with the set of indices representing the positions of the original strings of $\mathcal{M}$ in the list. Given the

generalized conjugate array of $\mathcal{M}$ we can construct eBWT($\mathcal{M}$) in linear time, since eBWT($\mathcal{M}$)$[i] = T_d[j-1]$ if GCA$[i] = (d, j)$, where again, the strings in $\mathcal{M}$ are considered to be cyclic. It is easy to see that when $\mathcal{M}$ consists of only one string, i.e. $\mathcal{M} = \{T\}$, then eBWT($\mathcal{M}$) = BWT(T).

Example 3. In Figure 2.2 we show the eBWT, GCA, and LF-mapping for the multiset of strings $\mathcal{M} = \{\texttt{AAT}, \texttt{TAGA}, \texttt{AT}\}$. From the GCA we can compute eBWT($\mathcal{M}$) = `TTAGTAAAA`, with index set $\{1, 5, 8\}$. We highlight the difference between the ω-order and lexicographic order in the GCA by noting that $\text{GCA}_{\mathcal{M}}[4] = (2, 4)$ and $\text{GCA}_{\mathcal{M}}[5] = (3, 1)$ corresponding to conjugates `ATAG` and `AT` respectively.

Similarly to the BWT, the eBWT is a reversible transformation, i.e., starting from eBWT($\mathcal{M}$) the original strings of the multiset $\mathcal{M}$ can be recovered (up to rotation). Such a recovery can be realized by using the LF-mapping denoted by $\text{LF}_{\mathcal{M}}$ and defined as the standard permutation of the string eBWT($\mathcal{M}$). In particular, for each $d = 1, \ldots, m$, the string T_d is constructed as follows: $T_d[|T_d|] = \text{eBWT}(\mathcal{M})[k_d]$, with $\text{GCA}[k_d] = (d, 1)$, and for $i \geq 1$, $T_d[|T_d| - i] = T_d[\text{LF}^i_{\mathcal{M}}(k_d)]$. Finally, the definition of the ϕ-mapping can be naturally extended to the generalized conjugate array $\text{GCA}_{\mathcal{M}}$ as follows: $\phi_{\mathcal{M}}(\text{GCA}[1]) = \text{GCA}[||\mathcal{M}||]$ and for each $h > 1$, $\phi_{\mathcal{M}}(\text{GCA}[h]) = \text{GCA}[h-1]$.

3

An extensive study of the BWT of string collections

In this chapter, we present an extensive analysis of different BWT variants of string collections present in the literature. The contents of this chapter were published in [28–30].

As mentioned in the introduction, the BWT was originally designed for individual strings, and it is not completely straightforward how to compute the BWT of string collections. In fact, there exist multiple ways to extend the BWT that generate non-equivalent results. As an effect, there are several publicly available tools explicitly designed to compute the BWT of string collections that use not only different algorithms but also output different data structures. We identified 15 such tools computing variants of the BWT; among these `BEETL`, `BCR_LCP_GSA` [8], `ropebwt2` [73], `nvSetBWT` [105], `msbwt` [57], `Merge-BWT` [116], `eGSA` [82], `BigBWT` [21], `bwt-lcp-parallel` [13], `eGAP` [36], `gsufsort` [81], `G2BWT` [33], `grlBWT` [34], `cais` and `pfpebwt` [17]. We show a first example, in Table 3.1, where we give 5 BWT variants on a toy example of 5 DNA strings.

The classical way of computing text indexes of string collections is to concatenate the strings, adding a different end-of-string-symbol at the end of each string, and then computing the index for the concatenated string. This is the method traditionally used for generating classical data structures such as suffix trees and suffix arrays for more than one string, and results in the so-called *generalized suffix tree* resp. *generalized suffix array* (see e.g. [56,101]). The drawback of this method is an increase of the size of the alphabet, from σ, often a small constant in applications, to $\sigma + k$, where k is the number of elements in the collection, typically in the thousands or even tens or hundreds of thousands.

In the literature, we can find two main classes of methods designed to avoid this drawback: using only conceptually different end-of-string characters as in `ropebwt2` [73] and `gsufsort` [81] or concatenating the string with the same dollar as in `BigBWT` [21]. Many studies nowadays use string collections in experiments (e.g. [5, 67, 112]), and apply one of the methods to avoid the alphabet size increase; however, the exact method used by the software is not always stated explicitly. Underlying this is the implicit assumption that all methods are equivalent.

In 2007, Mantaci et al. [87] introduced the eBWT, which is the first generalization of the BWT to a multiset of strings not using dollars. The eBWT, like

variant	result on example	tools
eBWT	CGGGATGTACGTTAAAAA	pfpebwt [17], cais [17]
dolEBWT	GGAAACGG$$$TTACTGTAAA	G2BWT [33], pfpebwt [17], msbwt [57] cais [17]
mdolBWT	GAGAAGCG$$$TTATCTGAAA	BEETL [8], ropebwt2 [73], nvSetBWT [105], Merge-BWT [116], eGSA [82], eGAP [36], bwt-lcp-parallel [13], gsufsort [81], grlBWT [34], BCR_LCP_GSA [8]
concatBWT	$AAGAGGGC$#$TTACTGT$AAA$	BigBWT [21], tools for single-string BWT
colexBWT	AAAGGCGG$$$TTACTGTAAA	ropebwt2 [73]

Table 3.1: The different BWT variants on the multiset $\mathcal{M} = \{\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}\}$.

the BWT, is reversible; moreover, it is independent of the order in which the strings in the collection are presented. This is not true of any of the methods mentioned above. All tools but pfpebwt and cais (see Chapters 5 and 6) append an end-of-string character to the input strings, explicitly or implicitly, and as a consequence, the resulting data structures differ from the one defined in [87]. Moreover, the output in most cases depends on the input order of the sequences (except for [33], [57] and using a specific option [73]). As a further complication, the exact nature of this dependence differs from one data structure to another.

The result is that the BWT variants computed by different tools on the same dataset, or by the same tool on the same dataset but given in a different order, may vary considerably. As we will show, this variability extends to the parameter r, the number of runs of the BWT. This is all the more important given the fact that r (and the related parameter n/r, the average length of a run) is increasingly being used as a parameter characterizing both the BWT-based data structures and the datasets themselves, namely as a measure of their repetitiveness (see e.g. [5, 20, 31]).

In this chapter, we show what is, to the best of our knowledge, the first systematic treatment of the different BWT variants in use for collections of strings. We define five distinct BWT variants, which are computed by 15 current tools designed explicitly for string collections. Given these transforms, we formally describe the differences between these, identifying specific intervals to which differences are restricted. We show the influence of the input order on the output and how it interacts with the different BWT variant definitions. Finally, we describe the consequences on the number of runs r of the BWT and complement our theoretical analysis with extensive experiments, comparing the five BWT variants on eight real-life datasets with different characteristics.

3.1 Related work

In this chapter, we consider tools for string collections, so we did not include any tool that computes the BWT of a single string, such as libdivsufsort [90], sais-lite-lcp [42], libsais [53], bwtdisk [37]. Even though in several cases, these are the tools used for collections of strings, the data structure they compute depends

on the specific method used to concatenate the strings of the input collections. Nor did we include other BWT variants for single strings, such as the bijective BWT [49, 65], since, again, these were not designed for string collections.

In addition, we did not include any tools that compute the xBWT of Ferragina et al. [38, 39], among these `Big-xBWT` [43] and the tool in [102]. The xBWT, as opposed to all other BWT variants we review, first maps the input to a tree and then applies the xBWT to it, making it a BWT-like index for labeled trees, rather than for strings. Moreover, the xBWT is not a permutation of the input characters; it can be shorter than a BWT. Due to this, the output is not easily comparable to the other BWT variants we review. Last `Big-xBWT` requires a reference sequence as input, in contrast to all other tools.

There has been considerable interest recently in the parameter r, the number of runs of the BWT: it was put in relation with other measures of repetitiveness in [61], while both [25] and [10] studied the question which permutation of the input strings of the collection results in the lowest value for r. Since the BWT variant used in [25] (the BWT of the strings concatenated with the same separator symbol but without an additional end-of-string character) differs from all BWT variants that have been implemented by some tools, we do not include it in this study. The result by Bentley et al. [10], on the other hand, is more related to the BWT variants we review, and we employ it as a benchmark in our experimental comparisons (Section 3.6). See Chapter 4 for an implementation and extensive experiments on the algorithm of Bentley et al.

3.2 Overview

In Section 3.3, we present the BWT variants and discuss how to construct them. In Section 3.4, we discuss the consequences of adding the separator characters and show the effects on the parameter r. In Section 3.5, we discuss the permutations induced by the separator-based BWT variants. Finally, we present our experimental results and some conclusions from our study in Sections 3.6 and 3.7.

3.3 BWT variants of string collections in the literature

We identified five distinct transforms, which we list below.

These BWTs are computed by the software we listed in Table 3.1. Let $\mathcal{M} = \{T_1, \ldots, T_k\}$ be a multiset of strings, with total length $N_{\mathcal{M}} = \sum_{i=1}^{k} |T_i|$. Since several of the data structures depend on the order in which the strings are listed, we implicitly regard $\mathcal{M}$ as a list $[T_1, \ldots, T_k]$, and write $(\mathcal{M}, \pi)$ explicitly for a specific permutation π in which the strings are presented.

1. eBWT($\mathcal{M}$): the extendedBWT of $\mathcal{M}$ of Mantaci et al. [87].
2. The *multidollar BWT* (mdolBWT) is defined as the BWT of the concatenation of all input strings separated with different dollars, mdolBWT($\mathcal{M}$) = BWT($T_1\$_1T_2\$_2 \cdots T_k\$_k$), where the dollars are assumed to be smaller than all characters in Σ and $\$_1 < \$_2 < \ldots < \$_k$. Alternatively, the mdolBWT

can be defined as the eBWT of the strings terminated with different dollars $\text{mdolBWT}(\mathcal{M}) = \text{eBWT}(\{T_i\$_i \mid T_i \in \mathcal{M}\})$.
3. The *dollar-eBWT* (dolEBWT) is defined as the mdolBWT of a string collection, $\text{dolEBWT}(\mathcal{M}) = \text{mdolBWT}(\mathcal{M}, \delta)$ where δ is the permutation corresponding to the lexicographic order. Alternatively, the dolEBWT can be defined as the eBWT of a string collection where all input strings are terminated with the same dollar, $\text{dolEBWT}(\mathcal{M}) = \text{eBWT}(\{T_i\$ \mid T_i \in \mathcal{M}\})$.
4. The *concatenated BWT* (concBWT) is defined as the BWT of the concatenation of all input strings separated with the same dollar plus a final eof character, $\text{concBWT}(\mathcal{M}) = \text{BWT}(T_1\$T_2\$\cdots T_k\$\#)$, where $\# < \$$.
5. The *colexicographic BWT* (colexBWT) is defined as the mdolBWT of a collection of strings, $\text{colexBWT}(\mathcal{M}) = \text{mdolBWT}(\mathcal{M}, \gamma)$, where γ is the permutation corresponding to the colexicographic (aka 'reverse lexicographic') order of the strings in $\mathcal{M}$.

Because all BWT variants except the eBWT use additional end-of-string symbols as string separators, we refer to these four by the collective term *separator-based* BWT variants. In Table 3.5, we show the five data structures on our running example of five DNA strings and give the first properties of these data structures. For ease of exposition and comparison, we replaced all separator symbols with the same dollar-sign \$ for all string separator symbols, even where, conceptually or concretely, different dollar-signs are assumed to terminate the individual strings, as is the case for mdolBWT. Moreover, the concBWT contains one additional character, the final end-of-string symbol, here denoted by #, which is smaller than all other characters; thus, the additional rotation starting with # is the smallest and results in an additional dollar in the first position of the transform. For ease of comparison, we remove this first symbol from concBWT and replace the # by \$, we refer to this transform as *adapted concBWT*.

Finally, we note that both the dolEBWT and the colexBWT can be defined using the mdolBWT. These two transforms correspond to the special case where the input strings are sorted according to a specific order relation, namely the lexicographic and the colexicographic order, respectively.

3.3.1 Methods to compute the BWT variants

`pfpebwt` and `cais` are the only current tools in the literature for computing the eBWT (see Chapters 5 and 6 for a complete description). Note that the eBWT differs from the other BWT variants in several ways, most importantly in the order relation for sorting conjugates: while the BWT uses lexicographic order, the eBWT uses the so-called omega order (see Chapter 2). This can be clearly seen in Table 3.4, where appending dollars leads to a completely different order of the strings' conjugates. Due to this, these two tools feature different characteristics from all other software. In particular, they do not use dollars to separate the strings in the input and compute the final transform by sorting the strings' conjugates circularly rather than sorting suffixes.

On the other hand, the separator-based BWTs represent the vast majority of the BWT of string collections used in the literature. Here we need to make an important distinction between the tools computing the mdolBWT and

the concBWT. The tools computing the mdolBWT append implicitly different end-of-string-symbols, i.e. they use the same dollar-sign and apply string input order to distinguish them. However, from an algorithmic point of view, there are different ways to implement this method. For example, the `gsufsort` tool explicitly produces the concatenation of the input strings separated by the same dollar sign; then it computes the SA and the BWT of the concatenation, breaking ties between equal suffixes using the dollar positions. On the other hand, the `BEETL` tool only simulates the suffix sorting procedure without concatenating the strings or even appending the end-of-string characters. However, in terms of the final transform, all these tools produce an equivalent BWT.

As for computing the concBWT, we need to separate the input strings using the same end-of-string-symbol and compute the BWT of the concatenation; in this case, a different end-of-string-symbol # has to be added to the end of the concatenated string, as in `BigBWT` [21] to ensure correctness. An equivalent solution is concatenating the input strings without removing the end-of-line and end-of-file characters since these act as separators. In Section 3.5, we will show that it is not possible to define the mdolBWT using the concBWT. Thus, we cannot use the same algorithmic approaches for both transforms, noting that they are not equivalent. Finally, we note that the concBWT can be directly computed from the SA of input strings concatenated with the same dollar without adding the final end-of-string symbol #. However, in this case, when inverting the concBWT we need to use a slightly different algorithm which causes the loss of the original order of the sequences in the concatenation.

3.4 The effects of adding separator symbols

The first obvious difference, when adding separator symbols, is in the length of the transform. The eBWT($\mathcal{M}$) has length $N_{\mathcal{M}}$ while all other BWT variants have length $N_{\mathcal{M}} + k$ since they contain an additional end-of-string character for each input string.

In all four separator-based transforms, the k-length prefix of the transform consists of a permutation of the last characters of the input strings. This is because the k rotations starting with dollar are always the lexicographically smallest among all rotations. On the other hand, in the eBWT, these characters occur interspersed in the transform, namely in the positions corresponding to the omega-ranks of the input strings T_i.

The next point is that adding a \$ to the end of each string introduces a difference between the suffixes and the other substrings occurring as internal factors: since the separators are smaller than all other characters, occurrences of a substring as suffix will be listed en bloc before all other occurrences of the same substring. On the other hand, in the eBWT, these occurrences will be listed interspersed with the other occurrences of the same substring.

Example 4. Let $\mathcal{M} = \{\texttt{AACGAC}, \texttt{TCAC}\}$ and $U = \texttt{AC}$. U occurs both as a suffix and as an internal factor; the characters preceding it are `A` (internal substring) and `C,G` (suffix), and we have eBWT($\mathcal{M}$) = `CGACATAACC`, dolEBWT($\mathcal{M}$) = `CC$GCAAATAC$` since `ACGACA` $<_{lex}$ `ACTA` while `ACGAC$A` $>_{lex}$ `AC$TA`.

index	mdolBWT	rotation
(1,6)	G	$\$_1$ATATG
(2,4)	A	$\$_2$TGA
(3,4)	G	$\$_3$ACG
(4,5)	A	$\$_4$ATCA
(5,4)	A	$\$_5$GGA
(2,3)	G	A$\$_2$TG
(4,4)	C	A$\$_4$ATC
(5,3)	G	A$\$_5$GG
(3,1)	$\$_3$	ACG$\$_3$
(1,1)	$\$_1$	ATATG$\$_1$
(4,1)	$\$_4$	ATCA$\$_4$
(1,3)	T	ATG$\$_1$AT
(4,3)	T	CA$\$_4$AT
(3,2)	A	CG$\$_3$A
(1,5)	T	G$\$_1$ATAT
(3,3)	C	G$\$_3$AC
(2,2)	T	GA$\$_2$T
(5,2)	G	GA$\$_5$G
(5,1)	$\$_5$	GGA$\$_5$
(1,2)	A	TATG$\$_1$A
(4,2)	A	TCA$\$_4$A
(1,4)	A	TG$\$_1$ATA
(2,1)	$\$_2$	TGA$\$_2$

index	colexBWT	rotation
(1,5)	A	$\$_1$ATCA
(2,4)	A	$\$_2$GGA
(3,4)	A	$\$_3$TGA
(4,4)	G	$\$_4$ACG
(5,6)	G	$\$_5$ATATG
(1,4)	C	A$\$_1$ATC
(2,3)	G	A$\$_2$GG
(3,3)	G	A$\$_3$TG
(4,1)	\$	ACG$\$_4$
(5,1)	\$	ATATG$\$_5$
(1,1)	\$	ATCA$\$_1$
(5,3)	T	ATG$\$_5$AT
(1,3)	T	CA$\$_1$AT
(4,2)	A	CG$\$_4$A
(4,3)	C	G$\$_4$AC
(5,5)	T	G$\$_5$ATAT
(2,2)	G	GA$\$_2$G
(3,2)	T	GA$\$_3$T
(2,1)	\$	GGA$\$_2$
(5,2)	A	TATG$\$_5$A
(1,2)	A	TCA$\$_1$A
(5,4)	A	TG$\$_5$ATA
(3,1)	\$	TGA$\$_3$

Table 3.2: From left to right we show the mdolBWT, and the colexBWT of the string collection $\mathcal{M} = \{\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}\}$.

In addition, it should be noted that adding end-of-string symbols to the input strings changes the definition of the order applied. As observed above, the omega-order coincides with the lexicographic order on all pairs of strings S, T where neither is a proper prefix of the other. This is because the lexicographic order always ranks first the string that is a prefix of the other, while this is not true with the omega-order. It follows that if the two distinct strings have the same length, neither can be a proper prefix of the other; thus, the two ordering relations are equivalent. This condition is often unrealistic when considering large string collections, where we may need to sort many conjugates of different lengths. However, when adding different end-of-string characters, no conjugate can be a proper prefix of another since all dollars are unique. Thus, when sorting the rotations of the $T_i\$_i$'s, the omega-order and the lexicographic order of the conjugates are always equivalent.

Example 5. Let $\mathcal{M} = \{\texttt{AGC}\$_1, \texttt{GC}\$_2\}$ a string collection where we appended a different dollar at the end of each string, we have the following omega-order sorted conjugates list: $\$_1\texttt{AGC}, \$_2\texttt{GC}, \texttt{AGC}\$_1, \texttt{C}\$_1\texttt{AG}, \texttt{C}\$_2\texttt{G}, \texttt{GC}\$_1, \texttt{GC}\$_2$, which is equivalent to the lexicographic order sorting.

Moreover, appending different end-of-string characters has an important effect on the relationship between the lexicographic order of the suffixes and the

index	concBWT	rotation
23	A	$#ATATG$TGAACGATCA$GGA
10	A	ACGATCAGGA#ATATG$TGA
14	G	$ATCA$GGA$#ATATG$TGA$ACG
19	A	GGA#ATATGTGAACG$ATCA
6	G	TGAACG$ATCA$GGA$#ATATG
22	G	A$#ATATG$TGAACGATCA$GG
9	G	AACGATCAGGA#ATATG$TG
18	C	AGGA#ATATGTGAACG$ATC
11	$	ACG$ATCA$GGA$#ATATG$TGA$
1	$	ATATGTGAACG$ATCA$GGA$#
15	$	ATCAGGA#ATATGTGAACG$
3	T	ATGTGAACG$ATCA$GGA$#AT
17	T	CAGGA#ATATGTGAACG$AT
12	A	CG$ATCA$GGA$#ATATG$TGA$A
13	C	G$ATCA$GGA$#ATATG$TGA$AC
5	T	GTGAACG$ATCA$GGA$#ATAT
21	G	GA$#ATATG$TGAACGATCA$G
8	T	GAACGATCAGGA#ATATG$T
20	$	GGA$#ATATG$TGAACGATCA$
2	A	TATGTGAACG$ATCA$GGA$#A
16	A	TCAGGA#ATATGTGAACG$A
4	A	TGTGAACG$ATCA$GGA$#ATA
7	$	TGAACGATCAGGA#ATATG$

index	optBWT	rotation
(1,4)	A	$\$_1$TGA
(2,4)	A	$\$_2$GGA
(3,5)	A	$\$_3$ATCA
(4,4)	G	$\$_4$ACG
(5,6)	G	$\$_5$ATATG
(1,3)	G	A$\$_1$TG
(2,3)	G	A$\$_2$GG
(3,4)	C	A$\$_3$ATC
(4,1)	$	ACG$\$_4$
(5,1)	$	ATATG$\$_5$
(3,1)	$	ATCA$\$_3$
(5,3)	T	ATG$\$_5$AT
(3,3)	T	CA$\$_3$AT
(4,2)	A	CG$\$_4$A
(4,3)	C	G$\$_4$AC
(5,5)	T	G$\$_5$ATAT
(1,2)	T	GA$\$_1$T
(2,2)	G	GA$\$_2$G
(2,1)	$	GGA$\$_2$
(5,2)	A	TATG$\$_5$A
(3,2)	A	TCA$\$_3$A
(5,4)	A	TG$\$_5$ATA
(1,1)	$	TGA$\$_1$

Table 3.3: From left to right we show the concBWT and the optBWT of the string collection $\mathcal{M} = \{\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}\}$.

omega-order of the conjugates. In fact, these two order relations coincide if there are no two strings S, T such that they have a common suffix or one suffix is a proper prefix of another. Thus, when using different dollars, the omega-order of the conjugates, the lexicographic order of the conjugates, the lexicographic order of the strings' suffixes, and the lexicographic order of the concatenation's suffixes are all equivalent.

Example 6. Let $\mathcal{M} = \{\texttt{AGC}\$_1, \texttt{GC}\$_2\}$, we have the following lexicographically sorted suffixes list: $\$_1, \$_2, \texttt{AGC}\$_1, \texttt{C}\$_1, \texttt{C}\$_2, \texttt{GC}\$_1, \texttt{GC}\$_2$, which is equivalent to the sorting computed applying the omega-order to the strings' conjugates.

3.4.1 Interesting intervals

Regarding the differences among the four separator-based BWT variants, we show that all differences occur in certain well-defined intervals of the BWT, called *interesting intervals*, and that the differences depend only on the permutation of $\{1, \cdots, k\}$, given by the combination of the input order, the lexicographic order of the input strings, and the BWT variant applied. In Tables 3.2, 3.3, and 3.4 we give the full BWT matrices for all five BWT variants, along with the optimal BWT minimizing the number of runs, see Section 3.4.2.

index	eBWT	rotation
(4,4)	C	AATC
(3,1)	G	ACG
(5,3)	G	AGG
(1,1)	G	ATATG
(4,1)	A	ATCA
(1,3)	T	ATGAT
(2,3)	G	ATG
(4,3)	T	CAAT
(3,2)	A	CGA
(3,3)	C	GAC
(5,2)	G	GAG
(1,5)	T	GATAT
(2,2)	T	GAT
(5,1)	A	GGA
(1,2)	A	TATGA
(4,2)	A	TCAA
(1,4)	A	TGATA
(2,1)	A	TGA

index	dolEBWT	rotation
(3,4)	G	$ACG
(1,6)	G	$ATATG
(4,5)	A	$ATCA
(5,4)	A	$GGA
(2,4)	A	$TGA
(4,4)	C	A$ATC
(5,3)	G	A$GG
(2,3)	G	A$TG
(3,1)	$	ACG$
(1,1)	$	ATATG$
(4,1)	$	ATCA$
(1,3)	T	ATG$AT
(4,3)	T	CA$AT
(3,2)	A	CG$A
(3,3)	C	G$AC
(1,5)	T	G$ATAT
(5,2)	G	GA$G
(2,2)	T	GA$T
(5,1)	$	GGA$
(1,2)	A	TATG$A
(4,2)	A	TCA$A
(1,4)	A	TG$ATA
(2,1)	$	TGA$

Table 3.4: From left to right we show the eBWT, and the dolEBWT of the string collection $\mathcal{M} = \{\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}\}$.

BWT variant	example	order of shared suffixes	independent of input order?
non-sep.based			
eBWT($\mathcal{M}$)	CGGGATGTACGTTAAAAA	omega-order of strings	yes
separator-based			
dolEBWT($\mathcal{M}$)	GGAAACGG$$$TTACTGTAAA	lexicographic order of strings	yes
mdolBWT($\mathcal{M}$)	GAGAAGCG$$$TTATCTGAAA	input order of strings	no
concBWT($\mathcal{M}$)	AAGAGGGC$$$TTACTGTAAA	lexicographic order of subsequent strings in input	no
colexBWT($\mathcal{M}$)	AAAGGCGG$$$TTACTGTAAA	colexicographic order	yes

Table 3.5: Overview of properties of the five BWT variants considered in this chapter. The colors in the example BWTs correspond to interesting intervals in separator-based variants.

Let us call a string U a *shared suffix* w.r.t. multiset $\mathcal{M}$ if it is the suffix of at least two strings in $\mathcal{M}$. Let b be the lexicographic rank of the smallest rotation beginning with $U\$$ and e the lexicographic rank of the largest rotation beginning with $U\$$, among all rotations of strings $T\$$, where $T \in \mathcal{M}$. (One can think of $[b, e]$ as the suffix-array interval of $U\$$.) We call $[b, e]$ an *interesting interval* if there exist $i \neq j$ s.t. U is a suffix of both T_i and T_j, and the preceding

characters in T_i and T_j are different, i.e., the two occurrences of U as suffix of T_i and T_j constitute a match that cannot be extended on the left (left-maximal repeat). (Interesting intervals correspond to internal nodes in the suffix tree of the reverse string within the subtree of \$).

Clearly, $[1,k]$ is an interesting interval unless all strings end with the same character. Note that interesting intervals differ both from the *SAP-intervals* of [32] and from the *tuples* of [10] (called *maximal row ranges* in [89]): the former are the intervals corresponding to *all* shared suffixes U, even if not left-maximal, while the latter include also suffixes U that are not shared. In particular, an interesting interval is a SAP-interval containing at least two different characters. The next lemma follows from the fact that no two substrings ending in \$ can be a prefix of the other.

Lemma 1. *Any two distinct interesting intervals are disjoint.*

Proof. Follows immediately from the fact that no two distinct substrings ending in \$ can be one prefix of the other.

We can now narrow down the differences between any two separator-based BWTs of the same multiset. The next proposition states that these can only occur in interesting intervals (part 1). This implies that the dollar-symbols appear in the same positions in all separator-based variants except for one very specific case (part 2). Moreover, we get an upper bound on the Hamming distance between two separator-based BWTs (part 3).

Proposition 1. *Let L_1 and L_2 be two separator-based BWTs of the same multiset $\mathcal{M}$.*

1. *If $L_1[i] \neq L_2[i]$ then $i \in [b,e]$ for some interesting interval $[b,e]$.*
2. *Let $\mathcal{I}_1$ resp. $\mathcal{I}_2$ be the positions of the dollars in L_1 resp. L_2. If $\mathcal{I}_1 \neq \mathcal{I}_2$ then there exist $i \neq j$ such that T_i is a proper suffix of T_j.*
3. $\mathrm{dist}_{\mathrm{H}}(L_1, L_2) \leq \sum_{[b,e] \text{ interesting interval}} (e-b+1).$

Proof. 1. Let $L_1[i] = \mathtt{x}$ and $L_2[i] = \mathtt{y}$ be two BWT positions, such that $L_1[i] \neq L_2[i]$. The two BW-matrices consist of the lexicographically ordered suffixes of the same strings in $\mathcal{M}$. Thus, if $\mathtt{x} \neq \mathtt{y}$ there exists two occurrences of a suffix U in $\mathcal{M}$ which are preceded by $\mathtt{x}$ and $\mathtt{y}$, and both have rank i in the two BWTs. Assume that position i is not part of any interesting interval. This means that all occurrences of U in $\mathcal{M}$ are preceded by the same character, i.e., U does not constitute a left-maximal repeat. However, this contradicts the initial statement $L_1[i] \neq L_2[i]$. It follows that i is part of an interesting interval $[b,e]$, which is the SA-interval of U. Parts *2.* and *3.* follow from *1.*

Proposition 1 implies that the variation of the different transforms can be explained based solely on what rule is used to break ties for shared suffixes. We will see next how the different rules used by the different BWT variants to break ties affect the parameter r.

3.4.2 The effects on the parameter r

In this section we investigate the effect of the different input permutations π of the strings in $\mathcal{M}$, induced by the BWT variants, on the number of runs of the BWT. As the following example shows, the number of runs can differ significantly between different variants.

Example 7. Let $\mathcal{M} = \{\texttt{AAAA}, \texttt{AGCA}, \texttt{GCAA}, \texttt{GTCA}, \texttt{CAAA}, \texttt{CGCA}, \texttt{TCAA}, \texttt{TTCA}\}$ a string collection of eight strings. We have that mdolBWT($\mathcal{M}$) = `AAAAAAAAACACACACACACAC$$GTGTGT$$AC$$GT$$`, and colexBWT($\mathcal{M}$) = `AAAAAAAAAAAACCCCAACCAC$$GGTTGT$$AC$$GT$$`.
We see that runs(mdolBWT($\mathcal{M}$)) = 28, and runs(colexBWT($\mathcal{M}$)) = 18, i.e. the colexBWT has about 55% fewer runs than the mdolBWT.

In the previous section, we saw that the differences in r between the separator-based BWTs can only occur in the interesting intervals. In particular, the number of runs in each of these intervals depends on how well the BWT characters can be mixed. If an interesting interval contains a single equal-letter run, we will have no differences between the different BWTs. On the other hand, if we have a similar number of occurrences for all characters, then we will have many chances to create new runs. In the following lemma, we will show the maximum number of runs we can achieve on a specific interesting interval.

Lemma 2. *Let $[b, e]$ be an interesting interval, and $(n_1, \ldots, n_\sigma)$ the Parikh vector of $L[b..e]$, i.e. n_i is the number of occurrences of the ith character. Let* a *be such that $n_{\mathtt{a}} = \max_i n_i$, and $N_{\mathtt{a}} = (e - b + 1) - n_{\mathtt{a}}$, the sum of the other character multiplicities. Then the maximum number of runs in interval $[b, e]$ is $e - b + 1$ if $n_{\mathtt{a}} - 1 \leq N_{\mathtt{a}}$, and $2N_{\mathtt{a}} + 1$ otherwise.*

Proof Place the $n_{\mathtt{a}}$ a-characters in a row, creating $n_{\mathtt{a}} + 1$ gaps, namely one between each adjacent a, and one each at the beginning and at the end. Now place all b-characters, each in a different gap; since $n_{\mathtt{a}}$ is maximum, there are enough gaps. Then place all c's, first filling gaps that are still empty, if any, then into gaps without c, etc. We never have to place two identical characters in the same gap. If the total number of non-a-characters is at least $n_{\mathtt{a}} - 1$, then we can fill every gap, thus separating all a's, and creating a run for every character of I. If we have fewer than $n_{\mathtt{a}} - 1$ characters, then we are still creating two runs with each non-a-character, but we cannot separate all a's.

This lemma shows that it is possible to draw a definition of the variability of a string collection. Given a Parikh vector if the maximum value is much smaller than the sum of all other characters' occurrences then we will have few chances to create new runs, otherwise, we will be able to mix the characters better.

We will use this lemma to measure the variability of a dataset:

Definition 1. *Let $\mathcal{M}$ be a multiset. For an interesting interval $[b, e]$, let $var([b, e])$ be the upper bound on the number of runs in $[b, e]$ from Lemma 1. Then the* variability *of $\mathcal{M}$ is*

$$var(\mathcal{M}) = \frac{\sum_{[b,e] \text{ interesting interval}} var([b, e])}{\sum_{[b,e] \text{ interesting interval}} (e - b + 1)}.$$

Example 8. Let $L = (0, 2, 1, 1, 0)$ a Parikh vector containing 2 `A`s one `C` and one `G`. Here we have the maximal variability since we can mix the characters to create a string with the maximum number of runs, `ACAG`.

Which of the BWT variants produces the fewest runs? As we have shown, this depends on the input order with most BWT variants, and the only possible variation is within interesting intervals. The colexBWT has been shown experimentally to yield a low number of runs of the BWT [32, 73]. Even though it does not always minimize r (one can easily create small examples where other permutations yield a lower number of runs), we can bound its distance from the optimum.

Proposition 2. *Let L be the colexBWT of multiset $\mathcal{M}$, and let r_{OPT} denote the minimum number of runs of any separator-based BWT of $\mathcal{M}$. Then* $\text{runs}(L) \leq r_{OPT} + 2 \cdot c_{\mathcal{M}}$*, where $c_{\mathcal{M}}$ is the number of interesting intervals.*

Proof. Let $I = [b_I, e_I]$ be an interesting interval containing d distinct characters, and let U be the shared suffix defining I. Since the strings are listed according to the colex order, all strings in which U is preceded by the same character will appear in one block, and therefore, L has exactly d runs in the interval I. Let $L_{b_I-1} = \mathtt{x}$ and $L_{e_I+1} = \mathtt{y}$. If $\mathtt{x}$ occurs in I and it is not the first run of I (i.e., $L_{b_I} \neq \mathtt{x}$), then listing first the strings where U is preceded by $\mathtt{x}$ would reduce the number of runs by 1; similarly, listing those where $\mathtt{y}$ precedes U as last of the group would reduce the number of runs by 1. By Prop. 1, this is the only possibility for varying the number of runs.

Bentley, Gibney, and Thankachan recently gave a linear-time algorithm for computing the order of the dollars which minimizes the number of runs [10], i.e. the optimal order for mdolBWT. We refer to this BWT as optimal BWT (optBWT) and provide an algorithm to construct this BWT variant in Chapter 4. The idea consists in starting from the colex-order and adjusting, where possible, the order of the runs within interesting intervals to minimize character changes at the borders, i.e. such that the first and the last run of each interesting interval is identical to the run preceding and following that interesting interval. This is equivalent to sorting groups of sequences sharing the same left-maximal suffix. This sorting can be done on each interesting interval independently without affecting the other interesting intervals. In Table 3.3, we show the result on our toy example, where it reduces the number of runs by 2 w.r.t. colexicographic order. We computed the number of optimal runs of our datasets according to the method of [10] and compared the number of runs of each of the five BWT variants to the optBWT (Section 3.6).

3.5 Permutations induced by separator-based BWT variants

In the previous section, we have seen that every BWT variant outputs a different permutation of the BWT characters as a consequence of using different input permutations. In this section, we further describe the relationship between the input order and the final transform.

Let us now restrict ourselves to $\mathcal{M}$ being a set, i.e., no string occurs more than once. (This is just for convenience since now the input order uniquely defines a permutation w.r.t. lexicographic order; the results of this section apply equally to multisets $\mathcal{M}$.) As we showed, the only differences between the different separator-based BWT variants are given by the order in which shared suffixes are listed. It is also clear that the same order applies in each interesting interval, as well as to the k-length prefix of the transform. Therefore, it suffices to study the permutation π of the k dollars in this prefix.

Since the strings are all distinct, they each have a unique lexicographic rank within the set $\mathcal{M}$. Thus the input order can be seen as a permutation ρ of the lexicographic ranks; if the strings are input in lexicographic order, then $\rho = id$. For our toy example $\mathcal{M} = [\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}]$, we have $\rho = 25134$. For those used to thinking about suffix arrays, ρ can be seen as the inverse suffix array of the input if the strings are thought of as meta-characters.

Let us now define as *output permutation* π the permutation of the last characters of the input strings, as found in the k-length prefix of the BWT variant in question. We will denote the output permutations of the dolEBWT, mdolBWT, concBWT, and colexBWT by $\pi_{de}, \pi_{md}, \pi_{conc}$, and π_{colex}, respectively. Again, we give these permutations w.r.t. the lexicographic ranks of the strings. In our running example, we have $\pi_{de} = 12345$, $\pi_{md} = 25134$, $\pi_{conc} = 45132$, and $\pi_{colex} = 34512$.

Example 9. Let $\mathcal{M} = [\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}]$ be a string collection where $\rho = 25134$. In the colexBWT we concatenate the sequences following the colexicographic order. Let $\mathcal{M}_{\text{colex}} = [\texttt{ATCA}, \texttt{GGA}, \texttt{TGA}, \texttt{ACG}, \texttt{ATATG}]$ be the re-ordered string collection, π_{colex} is the list of lexicographic ranks of the re-ordered sequences $\pi_{colex} = 34512$.

It is easy to see that the permutation π_{md} is equal to ρ, since the dollar-symbols are ordered according to ρ. For the dolEBWT, the rank of $\$T_i$ equals the lexicographic rank of T_i among all input strings, i.e., $\pi_{de} = id$. Further, $\pi_{colex} = \gamma$ by definition, where γ denotes the colexicographic order of the input strings. The situation is more complex in the case of concBWT. Since the # is the smallest character, the last string of the input will be the first, while for the others, the lexicographic rank *of the following string* decides the order. In our running example, $\pi_{conc} = 45132$. We next formalize how to map any ρ to π_{conc}.

Let Φ_ρ be the *linking permutation* [66] of ρ, defined by $\Phi_\rho(i) = \rho(\rho^{-1}(i)+1)$, for $i \neq \rho(k)$, and $\Phi_\rho(\rho(k)) = \rho(1)$, the permutation that maps each element to the element in the next position and the last element to the first. Let us also define, for $j \in \{1, \ldots, k\}$ and $i \neq j$, $f_j(i)$ by $f_j(i) = i$ if $i < j$ and $i-1$ otherwise, i.e. $f_j(i)$ gives the rank of element i in the set $\{1, \ldots, k\} \setminus \{j\}$. The next lemma gives the precise relationship between ρ and π_{conc}.

Lemma 3. *Let ρ be the permutation of the input order w.r.t. the lexicographic order, i.e. the ith input string has lexicographic rank $\rho(i)$. Then $\pi_{conc} = \pi_{conc}(\rho)$ is given by:*

$$\pi_{conc}(1) = \rho(k), \qquad \text{and for } i \neq \rho(k) : \pi_{conc}^{-1}(i) = f_{\rho(1)}(\Phi_\rho(i)) + 1. \tag{3.1}$$

Proof. Follows straightforwardly from the tie-breaking rule of concBWT.

Essentially, Lemma 3 says that π_{conc} is the BWT of ρ. Actually, we can take a string collection and construct a new string T^ρ concatenating the lexicographic ranks of the strings in it with a final dollar, in our example $T^\rho = 25134\$$. The π_{conc} is the BWT of T^ρ, BWT(25134\$) = 45\$132 = 45132 without the dollar. The same holds if we have more than one sequence with the same lexicographic rank. For example, if we have $T^\rho = 215521324\$$ we can easily compute π_{conc} by BWT(21521324\$) = 42253121.

Example 10. The mapping $\rho \mapsto \pi_{conc}$ for $k = 3$ is as follows: $123 \mapsto 312$, $132 \mapsto 231$, $312 \mapsto 231$, $213 \mapsto 321$, $231 \mapsto 132$, and $321 \mapsto 123$. Note that no ρ maps to 213.

Since not all BWT strings have a pre-image [75], not all permutations π_{conc} are reached by this mapping (as can be seen already for $k = 3$). We will call a permutation π *feasible* if there exists an input order ρ such that $\pi_{conc}(\rho) = \pi$, or alternatively if BWT(ρ\$) $= \pi$. For $k = 4$, there are 18 feasible permutations (out of 24), for $k = 5$, 82 (out of 120). In Table 3.6, we give the percentage of feasible permutations π, for k up to 11. The lexicographic order is always feasible, namely with $\rho = k, k-1, \ldots, 2, 1$; however, the colex order is not always feasible, as the following example shows.

Example 11. Let $\mathcal{M} = \{\texttt{GAA}, \texttt{ACA}, \texttt{TGA}\}$, thus $\gamma = 213$, but as we have seen, no permutation of the strings in $\mathcal{M}$ will yield this order for concBWT. In particular, the colexBWT($\mathcal{M}$) = `AAAACGG$AT$$` has 7 runs, and it is not a feasible concBWT. While all feasible concBWTs have at least 8 runs: `AAAGACG$AT$$`, `AAACGAG$AT$$`, `AAAAGCG$AT$$`, `AAAGCAG$AT$$`, `AAACAGG$AT$$`.

An important consequence is that given in input permutation ρ the output permutations induced by mdolBWT and concBWT are always different: $\pi_{md} \neq \pi_{conc}$ holds always, since $\pi_{conc}(1) = \rho(k)$. This means that in whatever order the strings are given w.r.t. lexicographic order, on most string sets, the resulting transforms, mdolBWT and concBWT, will differ. Note that two different output permutations can lead to identical BWTs. Moreover, the mdolBWT is much more flexible since every input order maps to a different permutation of the BWT characters, while for the concBWT some input permutations map to the same π. This implies that the π_{conc} that minimizes the number of runs of the BWT cannot always be reached (see Example 11).

no. of seq's k	3	4	5	6	7	8	9	10	11
	83.33%	75.0%	68.33%	63.89%	60.12%	57.29%	54.8%	52.81%	51.0%

Table 3.6: Percentage of feasible permutations w.r.t. concBWT.

3.6 Experimental results

We computed the five different BWT variants for eight genomic datasets with different characteristics and reported several statistics.

3.6.1 Experimental setup

All datasets are stored in FASTA format. We used three tools for computing the five BWT variants; `pfpebwt`, `ropebwt2` and `Big-BWT`. In order to make the BWTs comparable, we made some adaptations to both tools and inputs. We modified `ropebwt2` to make it work with the same character order as the other tools, i.e. $\$ < \mathtt{A} < \mathtt{C} < \mathtt{G} < \mathtt{N} < \mathtt{T}$. Then we used `ropebwt2` for computing both the mdolBWT and the colexBWT using the `-R` and `-R -s` flags respectively. We used `pfpebwt` for constructing both the eBWT and the dolEBWT variants. In order to compute the dolEBWT, we modified the input files, appending an end-of-string character at the end of each sequence. Finally, for computing the concBWT, we removed the headers from the FASTA files, arranging the sequences in newline separated files, and ran `Big-BWT` without additional flags on these newline separated files.

3.6.2 Datasets

We computed the five BWT variants for eight different genomic datasets with different characteristics. Four of the datasets contain short reads: SARS-CoV-2 short [118], Simons Diversity reads [85], 16S rRNA short [128], Influenza A reads [126], and four contain long sequences: SARS-CoV-2 long [52], 16S rRNA long [35], Candida auris reads [129], and SARS-CoV-2 genomes which contains whole viral genomes [17]. The main features of the datasets, including the number of sequences, sequence length, and the mean run-length of the optimal BWT, are reported in Table 3.7.

3.6.3 Results

On each of the datasets, we computed the pairwise Hamming distance between separator-based BWTs. To compare them to the eBWT, we computed the pairwise edit distance on a small subset of the sequences (for obvious computational reasons), computing also the Hamming distance on the small set for comparison. We generated some statistics on each of the data sets: the number of interesting intervals, the fraction of positions within interesting intervals defined as the ratio between the total length of the interesting intervals and the length of the transform, and the dataset's variability (Def. 1). To study the variation of the r-parameter, we implemented the algorithm by Bentley et al. [10], and computed r_{OPT} for each data set (see Chapter 4 for algorithm details), comparing it to the number of runs of all five BWT variants. In Table 3.9 and 3.10, we include a compact version of these results for the two datasets with the highest and the lowest variation between the BWT variants, the SARS-CoV-2 short sequences and the SARS-CoV-2 genomes, respectively. The complete experimental results for all eight datasets are contained in the Appendix.

In Table 3.8, we give a brief summary of the results, reporting, for each dataset, the fraction of positions in interesting intervals, the dataset's variability, the average pairwise Hamming distance between separator-based BWT variants, and the maximum and minimum value, among the five BWT variants, of the average run-length (n/r) of the BWT.

The experiments showed a high variation in the number of runs, particularly on datasets of short sequences. The highest difference was between colexBWT and concBWT, by a multiplicative factor of over 4.2, on the SARS-CoV-2 short dataset. In Figure 3.1, we plot the average run-length n/r for the four short sequence datasets and the percentage increase of the number of runs w.r.t. r_{OPT}. On the other hand, the r variation is less pronounced on the one dataset which is less repetitive (small n/r (opt)), namely Simons Diversity reads. Recall that the mdolBWT and concBWT vary depending on the input order, so we may obtain different results starting with different input permutations. Finally, on most long sequence datasets the differences were quite small.

In our experiments the colexBWT was the BWT variant that always showed the smallest number of runs among the 5 BWTs reviewed. To better understand how far the colexBWT is from the optimum w.r.t. the number of runs, we plot in Figure 3.2 the number of runs of colexBWT w.r.t. to r of the optBWT, on all eight datasets. The strongest increase is on short sequences, where the variation among all BWT variants is high, as well; on the long sequence datasets, with the exception of SARS-CoV-2 long sequences, the colexBWT is very close to the optimum; however, note that on those datasets, all BWTs are close to the optimum.

To sum up, we showed that the average number of runs and the average pairwise Hamming distance strongly depend on the length of the sequences in the input collection. If the collection has a lot of short sequences which are very similar, then the differences between the BWTs both w.r.t. the number of runs, and as measured by the Hamming distance, can be large. This is because the only variation in the separator-based BWTs is allowed in the interesting intervals. Thus, when working with short sequences we end up with a lot of maximal shared suffixes, and so many positions are in interesting intervals. To better understand this relationship, we plotted, in Figure 3.3, the average Hamming distance against the two parameters variability and fraction of positions in interesting intervals. We see that the two datasets with the highest average Hamming distance, SARS-CoV-2 short dataset, and the Simons Diversity reads, have at least one of the two values very close to 1, while for those datasets where both values are very low, the BWT variants do not differ very much.

3.7 Conclusion

In this chapter, we presented the first study of the different variants of the Burrows-Wheeler-Transform for string collections. We found that the data structures computed by different tools differ not insignificantly, as measured by the pairwise Hamming distance: up to 12% between different BWT variants on the same dataset in our experiments. We showed that most BWT variants in use are input order dependent, so the same tool can produce different variants if the input set is permuted. These differences also extend to the number of runs r, a parameter that is central in the analysis of BWT-based data structures and which is increasingly being used as a measure of the repetitiveness of the dataset itself.

With string collections replacing individual sequences as the prime object of research and analysis and thus becoming the standard input for text indexing

dataset	no. seq	total length	avg	min	max	n/r (opt)
SARS-CoV-2 short	500,000	25,000,000	50	50	50	35.125
Simons Diversity reads	500,000	50,000,000	100	100	100	8.133
16S rRNA short	500,000	75,929,833	152	69	301	44.873
Influenza A reads	500,000	115,692,842	231	60	251	50.275
SARS-CoV-2 long	50,000	53,726,351	1,075	265	3,355	74.498
16S rRNA long	16,741	25,142,323	1,502	1,430	1,549	47.140
Candida auris reads	50,000	124,150,880	2,483	214	8,791	1.732
SARS-CoV-2 genomes	2,000	59,610,692	29,805	22,871	29,920	523.240

Table 3.7: Table summarizing the main parameters of the eight datasets. From left to right we report the dataset name, the number of sequences, the total length, the average, minimum and maximum sequence length and the optimum average run-length (n/r), according to [10].

dataset	ratio pos.s in intr.int.s	varia-bility	avg. Hamming d. betw. \$-sep. BWTs	max n/r (avg. run-length)	min n/r (avg. run-length)
SARS-CoV-2 short	0.792	0.210	0.11754	31.524	7.494
Simons Diversity reads	0.107	0.976	0.07195	7.873	5.299
16S rRNA short	0.741	0.058	0.02982	44.253	18.836
Influenza A reads	0.103	0.363	0.02609	49.172	23.100
SARS-CoV-2 long	0.175	0.037	0.00464	73.204	57.568
16S rRNA long	0.047	0.104	0.00289	46.879	45.015
Candida auris reads	0.007	0.497	0.00246	1.732	1.726
SARS-CoV-2 genomes	0.001	0.148	0.00012	521.610	499.549

Table 3.8: Table summarizing the results on the eight datasets. From left to right we report dataset names followed by the ratio of positions in interesting intervals, the variability of the dataset (see Def. 1), the average normalized Hamming distance between any two separator-based BWT variants. In the last two columns we report the maximum and minimum average run-length (n/r) taken over all five BWT variants.

algorithms, we believe that it is all the more important for users and researchers to be aware that not all methods are equivalent, and to understand the precise nature of the BWT variant produced by a particular tool. We suggest further to standardize the definition of the parameter r for string collections in order to make it independent of the input order, using either the colexicographic order or the optimal order of Bentley et al. [10].

SARS-CoV-2 short (500,000 short sequences)

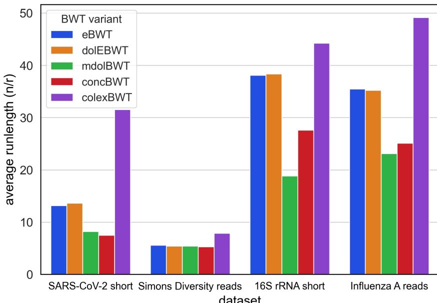
Hamming d. / *norm. Hamming d.*	*Hamming distance on the big dataset*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	3,014,183	2,926,602	2,912,860
mdolBWT	0.11820	0	3,013,908	3,102,887
concBWT	0.11477	0.11819	0	3,013,634
colexBWT	0.11423	0.12168	0.11818	0

dataset properties	
no. sequences	500,000
average length	50
total length	25,000,000
no. of interesting intervals	116,598
total length intr.int.s	20,187,840
fraction pos.s in intr.int.s	0.792
variability	0.210

edit d. / *norm. edit d.*	*edit distance on a subset of 5,0000 sequences*				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	28,702	43,903	43,828	46,936
dolEBWT	0.11256	0	17,000	16,921	20,104
mdolBWT	0.17217	0.06667	0	16,130	20,812
concBWT	0.17187	0.06636	0.06325	0	20,830
colexBWT	0.18406	0.07884	0.08162	0.08169	0

no. runs big dataset		
	r	n/r
eBWT	1,902,148	13.143
dolEBWT	1,868,581	13.647
mdolBWT	3,113,818	8.189
concBWT	3,402,513	7.494
colexBWT	808,906	31.524
optimum	725,979	35.125

Table 3.9: Results for the SARS-CoV-2 short dataset. Top left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. Top right: summary of the dataset properties. Bottom left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Bottom right: number of runs and average run-length (n/r) taken over all BWT variants.



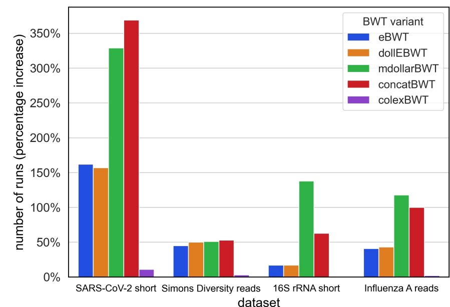


Fig. 3.1: Results regarding r on short sequence datasets, of all BWT variants. Left: average run-length (n/r). Right: number of runs (percentage increase with respect to optimal BWT).

SARS-CoV-2 genomes (2,000 long sequences)

Hamming d. / *norm. Hamming d.*	*Hamming distance on the big dataset*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	7,958	7,900	7,263
mdolBWT	0.00013	0	7,958	7,957
concBWT	0.00013	0.00013	0	7,990
colexBWT	0.00012	0.00013	0.00013	0

dataset properties	
no. sequences	2,000
total length	59,612,692
average length	29,085
no. interesting intervals	1863
total length intr.int.s	80,486
fraction pos.s in intr.int.s	0.001
variability	0.148

edit d. / *norm. edit d.*	*edit distance on a subset of 50 sequences*				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	786	795	801	791
dolEBWT	0.00053	0	98	107	86
mdolBWT	0.00053	0.00007	0	105	112
concBWT	0.00054	0.00007	0.00007	0	114
colexBWT	0.00053	0.00006	0.00008	0.00008	0

no. runs big dataset		
	r	n/r
eBWT	117,628	506.773
dolEBWT	117,410	507.731
mdolBWT	118,870	501.495
concBWT	119,334	499.549
colexBWT	114,287	521.605
optimum	113,930	523.240

Table 3.10: Results for the SARS-CoV-2 genomes dataset. Top left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. Top right: summary of the dataset properties. Bottom left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Bottom right: number of runs and average run-length (n/r) taken over all BWT variants.

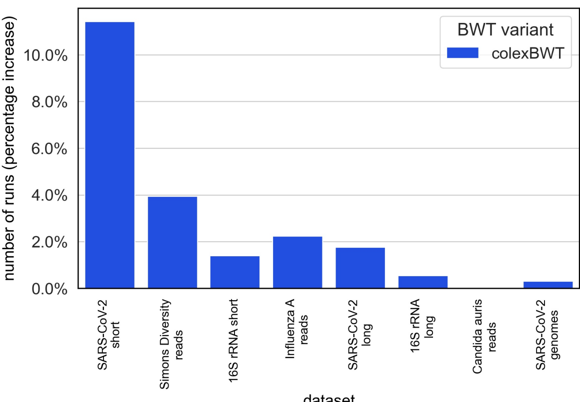


Fig. 3.2: Number of runs of the colexBWT with respect to optimal BWT (percentage increase) on all eight datasets.

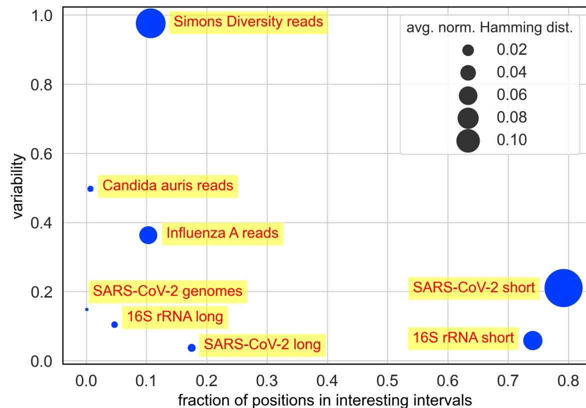


Fig. 3.3: Average normalized Hamming distance variations with respect to variability and fraction of positions in interesting intervals on all datasets.

4

Computing the optimal BWT

In this chapter, we present a linear-time algorithm for computing the optimal BWT, i.e., the BWT of string collections which guarantees the minimum number of runs. The contents of this chapter were published in [26, 27].

Since the run-length encoding scheme guarantees good compression rates while supporting fast pattern matching tasks, several compressed data structures on strings [44, 83] adopted this compression scheme. However, even though the time and space requirements of these data structures are measured using r, most of the available tools computing the BWT of string collections give in output a transform that depends on the input order of the sequences. Moreover, as we pointed out in Chapter 3, different input permutations can generate large variations of r.

The first attempt to fix the r variation was presented in 2012 in [32], when the authors introduced the *rlo-heuristic* (named *colexBWT* in Chapter 3) and *sap-heuristic*. These two heuristics are implemented by permuting the symbols within special intervals of the BWT, called SAP-intervals (*same-as-previous*), associated with suffixes equal up to the dollars. These SAP-intervals are a more general case than the interesting intervals described in Chapter 3. Within SAP-intervals, one can permute the symbols by grouping them into as few runs as possible, which is equivalent to reordering the input sequences. However, it is easy to construct examples on which neither of these two heuristics results in a BWT with a minimal number of runs (see Figure 4.1).

Recently, in 2020 Bentley et al. [10] presented a linear-time algorithm that computes a permutation of the input collection minimizing r, but they gave no implementation. In this chapter, we present an implementation of the Bentley et al. algorithm [10] for computing a BWT with the minimum number of runs we refer to as optimal BWT (optBWT). We implemented this algorithm in `optimalBWT`, the first tool that guarantees to output a BWT of a string collection with the minimal number of runs in terms of reordering of input strings. We present a framework for constructing this transform starting from two elements: a BWT and the SAP-array of Cox et al. [32]. We provide an on-the-fly construction of the SAP-array using two different algorithms: one is our adaptation of the SAIS-based algorithm we will present in Chapter 5, the other is the BCR algorithm [8].

We performed several experiments both on simulated and real-life short-read datasets. Since the variation in the number of runs is highest on string collections containing many similar short reads, in our experiments, we concentrate on short-read data. For each of these, we report the decrease in the number of runs provided by the optimal BWT compared with BWT variants computed with other input permutations, showing that the improvement can be very significant. In particular, on our real-life data, the optBWT obtains up to 31 times fewer runs with only a 1.39× slowdown, making `optimalBWT` competitive with other tools computing the BWT of string collections in terms of running time and space usage.

Our result is also significant because it allows using the number of runs of the optimal BWT as a repetitiveness measure for string collections. As we pointed out in Chapter 3, the parameter r should be standardized since it is being increasingly used as a parameter for string collections. Note that r is not well-defined if computed for a BWT variant which depends on the input order.

4.1 Overview

The rest of the chapter is organized as follows. In Section 4.2, we present an algorithm to compute the optBWT starting from the BWT and its SAP-array. In Section 4.3, we present the adaptations of two algorithms SAIS and BCR to compute the SAP-array and optBWT. Finally, in Section 4.4, and 4.5, we present our experimental results and give final comments.

multidollar BWT approaches				different orderings					sorted
$S_1\$_1 \cdot\cdot S_5\$_5\#$	$\{S_1\$_1, .., S_5\$_5\}$	$S_1\$_1 \cdot\cdot S_5\$_5$	SAP	mdolBWT	dolEBWT	colexBWT	sapBWT	optBWT	suffixes
$\$_5$									#
A	A	A	**0**	A	T	A	A	**T**	$
A	A	A	**1**	A	A	A	A	**T**	$
T	T	T	**1**	T	T	T	T	**T**	$
T	T	T	**1**	T	A	T	T	**A**	$
T	T	T	**1**	T	T	T	T	**A**	$
G	G	G	**0**	G	A	A	G	**A**	A$
A	A	A	**1**	A	G	G	A	**G**	A$
G	G	G	0	G	G	G	G	G	AA$
T	T	T	**0**	T	G	G	T	**G**	CCT$
G	G	G	**1**	G	T	T	G	**T**	CCT$
T	T	T	0	T	T	T	T	T	CGA$
C	C	C	**0**	C	C	C	C	**T**	CT$
T	T	T	**1**	T	C	C	C	**C**	CT$
C	C	C	**1**	C	T	T	T	**C**	CT$
C	C	C	0	C	C	C	C	C	GA$
G	G	G	0	G	G	G	G	G	GAA$
$\$_4$	$\$_5$	$\$_4$	0	$	$	$	$	$	GCCT$
$\$_1$	$\$_2$	$\$_1$	0	$	$	$	$	$	GGAA$
C	C	C	**0**	C	C	C	C	**C**	T$
C	C	C	**1**	C	C	C	C	**C**	T$
C	C	C	**1**	C	C	C	C	**C**	T$
$\$_2$	$\$_3$	$\$_2$	0	$	$	$	$	$	TCCT$
#	$\$_1$	$\$_5$	0	$	$	$	$	$	TCGA$
T	T	T	0	T	T	T	T	T	TCT$
$\$_3$	$\$_4$	$\$_3$	0	$	$	$	$	$	TTCT$
number of equal-letter runs				17	17	14	17	11	

Fig. 4.1: The output of different separator-based BWT variants applied to the string collection $\mathcal{M} = \{\texttt{TCGA}, \texttt{GGAA}, \texttt{TCCT}, \texttt{TTCT}, \texttt{GCCT}\}$. The SAP-array (SAP-intervals in bold) and the corresponding BWTs with different orderings of $\mathcal{M}$.

4.2 An algorithm for computing the optimal BWT

In this section, we describe the computation of the optBWT in two steps: *i*) computing an arbitrary BWT and its SAP-array, *ii*) determining the optBWT.

First we define the *SAP-array* [32], as a binary array of length $||\mathcal{M}||$ such that $\text{SAP}[i] = 1$ if and only if the symbol $L[i]$ is associated with a suffix which is *same as* its *previous* suffix (up to the dollar) in the list of sorted suffixes. An SAP-*interval* $L[b..e]$ is a maximal interval in BWT such that $\text{SAP}[i] = 1$, for all $b < i \leq e$. The SAP-intervals which contain more than one character correspond to left-maximal shared suffixes, which were called *interesting intervals* in Chapter 3. We also introduce the *reduced* SAP-*array* obtained from the SAP-array by setting $\text{SAP}_{\text{red}}[i] = 0$, $b < i \leq e$, for any SAP-interval $L[b..e]$ which is a run of the same symbol (see Table 4.1).

We will first explain how to obtain optBWT from an arbitrary BWT and the SAP-array (or equivalently, the reduced SAP-array). Then we describe how to obtain the SAP-array during the BWT-construction using an adaptation of the SAIS-based BWT construction algorithm described in Chapter 5, and finally, how to obtain the SAP-array during BWT-construction with BCR [8]. Note that the SAP-intervals containing more than one character correspond to *interesting intervals*.

4.2.1 Computing the optimal BWT using the SAP-array

mdolBWT	AATATAA	GAACT	CT	C	$	GG	C	A	$	$	$	T	AC	AA	GG	$	$	$
tuples	(A,T)	(A,C,G,T)	(C,T)	(C)	($)	(G)	(C)	(A)	($)	($)	($)	(T)	(A,C)	(A)	(G)	($)	($)	($)
tuples opt	(T,A)	(A,G,C,T)	(T,C)	(C)	($)	(G)	(C)	(A)	($)	($)	($)	(T)	(C,A)	(A)	(G)	($)	($)	($)
optBWT	TTAAAAA	AAGCT	TC	C	$	GG	C	A	$	$	$	T	CA	AA	GG	$	$	$
SAP-array	0111111	01111	01	0	0	01	0	0	0	0	0	0	01	01	01	0	0	0
reduced SAP-a.	0111111	01111	01	0	0	00	0	0	0	0	0	0	01	00	00	0	0	0

Table 4.1: The mdolBWT and optBWT on the string collection $\mathcal{M} = \{\texttt{TGA}, \texttt{CACAA}, \texttt{AGAGT}, \texttt{TAA}, \texttt{CGAGT}, \texttt{CCA}, \texttt{TA}\}$ together with their SAP-array and reduced SAP-array.

It is clear that as we pointed out in Section 3, all characters of the BWT are fixed except those within interesting intervals, and therefore, the BWT can be varied only within these. In fact, the two heuristics employed in [32, 73] reduce the number of runs *within* interesting intervals by grouping together all characters of the same type. The algorithm of Bentley et al. [10] further reduces the number of runs by grouping together runs of the same character at *borders* of interesting intervals, wherever possible. The authors show that this can be modeled as a problem they refer to as *tuple ordering problem*, which in turn can be turned into a shortest path problem in a DAG. The correctness of the algorithm and the fact that the output BWT is optimal follow from [10] and properties of interesting intervals shown in [29].

Each SAP-interval is mapped to a tuple containing those characters which occur in the interval at least once, while a position i outside any SAP-interval with $L[i] = c$ is mapped to (c) (See Table 4.1 for an example). We compute

Algorithm 1: Procedure to process a Parikh vector P

```
1  if Stack is empty then
2  |   if there is exactly one j such that P[j] > 0 then
3  |   |   write P[j] copies of character j          // interval not interesting
4  |   else
5  |   |   if P[x] > 0 where x is the last character inserted in the BWT then
6  |   |   |   write P[x] copies of the character x, P[x] <- 0
7  |   |   Stack <- pushTop(P)       // push a new Parikh vector on the stack
8  else
9  |   T <- Stack.top()                          // first element of the stack
10 |   if there are at least two j s.t. T[j] > 0 and P[j] > 0 then
11 |   |   Stack <- pushTop(P)
12 |   else
13 |   |   write corresponding characters for each T in Stack      // see text for
   |   |     details
```

the optBWT in a single left-to-right scan of the input BWT and the SAP-array. For every pair of neighboring SAP-intervals, the goal is to place identical character runs on either side of the border. If more than one character is shared between the two intervals, then this choice is not unique. Note that this implies that both intervals are interesting. The choice of the character may be further restricted by the other neighbors of the two intervals. Thus, an arbitrary number of consecutive interesting intervals may have to be kept track of before the decision on which characters to place at the borders can be made.

We maintain a stack to keep track of the Parikh vectors of the tuples for which the BWT has not yet been output. For each new tuple, if the stack is empty, either we can output the BWT immediately (Algorithm 1 lines 2-3), or check if there exists a match with the last character output in the BWT. If so, we remove the character from the Parikh vector and output its occurrences (lines 5-7), then we place it on the stack (line 8). Otherwise, if the stack is not empty, we check whether the characters can now be assigned (lines 11-16). This is the case if the top Parikh vector shares 1 or 0 characters with the current one: if it is 1, then that character must be taken, otherwise an arbitrary character can be chosen. We can now empty the stack and write the corresponding parts of the BWT. Finally, if some characters of the current Parikh vector were not written, we place the remaining Parikh vector on the stack.

In Table 4.1, the BWT starts with three interesting intervals. The corresponding Parikh vectors are placed on the stack. Arriving at $i = 16$, the stack contains $(0,5,0,0,2),(0,2,1,1,1),(0,0,1,0,1)$. The current Parikh vector is $(0,0,1,0,0)$ (corresponding to `C`), and `C` is the only character in the intersection with the top Parikh vector $(0,0,1,0,1)$. The BWT corresponding to the three interesting intervals can now be output and the stack emptied: `TTAAAAA|AAGCT|TC|C`, where we marked borders between interesting intervals by |. Note that if the symbols in the second interesting interval were permuted as `AACGT`, then we would also get the minimal number of runs. We use as default the inverse lexicographic order.

4.3 Adapting SAIS and BCR algorithms

In this section, we briefly present how we adapted two algorithms, one using a SAIS-based approach [99] and BCR [8], for computing the SAP-array. Finally, given the BWT and its SAP-array we can use the algorithm described in the previous section to permute the BWT characters and compute the final optBWT. While our SAIS-based algorithm keeps all computation in internal memory, BCR stores the growing BWT and SAP-array on the disk. Thus, in `optimalBWT`, we can use the SAIS-based algorithm to process datasets up to a few tens of gigabytes, while we can use the BCR algorithm to compute the optBWT of very large datasets, even hundred thousands of gigabytes.

4.3.1 Computing the SAP-array using SAIS

We generate the SAP-array during the computation of the BWT, using our adaptation of the SAIS-based algorithm described in Chapter 5. This is done by computing it in each recursion step and propagating it while mapping back one recursion level up. The SAP-array within a step can be computed along with the SA while inducing the L- and S-type suffixes. This is achieved via an adaptation of the inducing step that allows propagating the information that we are within a shared suffix: Let $S_i[t..n_i]\$$ be a shared suffix; if at least two positions are preceded by the same character c then $cS_i[t..n_i]\$$ corresponds to another SAP-interval. Since all occurrences of the same suffix are listed together in the SA, we can compute all SA-values in the new SAP-interval sequentially during the inducing step. This is carried out by keeping track of suffixes starting with the same character, and updating the SAP-array accordingly in case they are induced by the same shared suffix.

4.3.2 Computing the SAP-array using BCR

The BCR algorithm is based on the idea of right-to-left scanning, at the same time, all the k strings and building the BWT through $\ell + 1$ iterations, where ℓ is the length of the longest string. At each iteration, BCR considers a "slice" of (at most) k characters from the strings: it starts by concatenating the symbols preceding all $\$_i$, for all i, building a *partial* BWT (BWT_0). Then, at iteration j, for $j = 1, \ldots, k$, the symbols circularly preceding the suffixes $S_i[n_i-j+1..n_i]$ (for all $1 \leq i \leq k$) are inserted in the partial BWT_{j-1} by simulating the insertion of these suffixes in the lexicographically sorted list of suffixes of length h (for all $h < j$).

During the j'th step, we are able to compute and propagate from one iteration to the next the SAP-interval information (see also [8, 32]). Note that unlike [32], we compute SAP-intervals for the current iteration. Indeed, when inserting symbols circularly preceding a shared suffix $S_i[n_i-j+1..n_i]$ (for some i), we can deduce the length of the SAP-interval that these symbols form (i.e., their number). Furthermore, we can distinguish whether a SAP-interval is an interesting interval or not (i.e., the symbols form a equal-character run), so that we can incrementally build along with the BWT both the SAP-array and the reduced SAP-array.

4.4 Experimental Results

In this section, we evaluate the performance of our tool, named `optimalBWT`. Tests were performed on a DELL PowerEdge R630 machine, 24-core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40 GHz, with 128 GB of internal memory.

4.4.1 Datasets

To evaluate the performance of `optimalBWT`, we have designed a series of tests on both simulated and real-life short-read datasets.

	dataset	description	length BWTn	len.	no. seq	r_{opt}	n/r_{opt}	n/r
1	ERR732065–70	*HIV-virus*	1,345,713,812	150	8,912,012	11,539,661	116.62	27.62
2	SRR12038540	*SARS-CoV-2 RBD*	1,690,229,250	50	33,141,750	14,864,523	113.71	8.08
3	ERR022075_1	*E. Coli str. K-12*	2,294,730,100	100	22,720,100	71,203,469	32.23	8.83
4	SRR059298	*Deformed wing virus*	2,455,299,082	72	33,634,234	48,376,632	50.75	9.83
5	SRR065389–90	*C. Elegans*	14,095,870,474	100	139,563,074	921,561,895	15.30	6.26
6	SRR2990914_1	*Sindibis virus*	15,957,722,119	36	431,289,787	105,250,120	151.62	4.81
7	ERR1019034	*H. Sapiens*	123,506,926,658	100	1,222,840,858	10,860,229,434	11.37	5.35

Table 4.2: Real-life datasets used in the experiments together with the number of runs (r_{opt}) and the average run-length (n/r_{opt}) of the optBWT compared to the average run-length (n/r) of the inputBWT.

As for the simulated data, we generated short reads datasets by varying read lengths using the ART tool https://www.niehs.nih.gov/research/resources/software/biostatistics/art (sequencing machine Illumina HiSeq 2500) and three reference sequences CP068259.2 *H. Sapiens, chr.19*, NC_002516 .2 *P. Aeruginosa PAO1*, NC_003197.2 *S. enterica*. While for the real-life data, we downloaded seven short reads datasets with different characteristics from the NCBI platform. We summarized the main features of the datasets as well as the accession codes in Table 4.2.

4.4.2 Experimental setup

The experiments are arranged as a pipeline that runs the two steps described in the previous section and, for building the BWTand the SAP-array, provides two approaches: one is an adaptation of our SAIS-based algorithm (Chapter 5) that mainly works in internal memory, and the other is the BCR approach working in semi-external memory. We selected one of the two methods depending on the resources available.

We compare the number of runs in the optBWT with respect to the input order (inputBWT), the lexicographic order (dolEBWT), and the two heuristics, *rlo-heuristic* (colexBWT) and *sap-heuristic* (sapBWT) defined in [32] (see also Fig. 4.1). Note that in this section, we denote as inputBWT the mdolBWT of a string collection sorted according to an arbitrary random input order. Both heuristics reduce the number of runs within interesting intervals by grouping together all characters of the same type: the *rlo-heuristic* achieves this implicitly, since by sorting the input strings in colexicographic order, identical characters are grouped together within each interesting interval. The *sap-heuristic* can be

thought of as an approximation of the rlo-heuristic, in which the permutation of symbols within interesting intervals occurs during the on-the-fly construction of the BWT(through BEETL-BCRext) and the SAP-array information is implicitly obtained by computing a SAP status.

4.4.3 Results

We summarize the results for the real-life datasets in Table 4.3. For each comparison we report both the *factor increase* and the *percentage increase* obtained by $\frac{r-r_{opt}}{r_{opt}} \cdot 100$, where r is the number of the runs of the BWT variant. We also report the time and memory peak to construct the optBWT from scratch by choosing the algorithmic approach which has the best trade-off performance between the two proposed. We note that the increase of r with respect to the optBWT is significant for all different read lengths and n/r values. In particular, the two short-read datasets SRR2990914_1 and SRR1203854, featuring high n/r_{opt}, show 31.5 and 14.07 times fewer runs than the input order BWTspending only a 1.39× and 1.15× overhead in time by using the BCR- and SAIS-based approaches, respectively. On the other hand, on the large human dataset [85] (122.3 Gb) even if the factor is smaller than the others, the r saved is still over 10 billion with only a 1.48× time overhead.

As for the simulated data, we note that the difference in the number of runs of the inputBWT compared to the optBWT increases while decreasing the sequence length. As expected, we have the largest differences on the sequences of length 50, while for length 150 all numbers of runs were much more similar as shown in Figure 4.2. We summarized the results for Pseudomonas Aeruginosa in Table 4.4. Here we had the highest factor increase of 7.5 for the dataset of length 50, with a time overhead of 1.64x and 1.12x for the BCR and SAIS-based algorithms, respectively. However, the factor increase still is substantial for datasets with longer sequences, and the overhead to compute the optBWT remains small for all read lengths.

data	number of runs increase compared to optimal BWT				resource usage	
set	inputBWT	colexBWT (rlo)	sapBWT	dolEBWT	RAM (GB)	Time (hh:mm:ss)
1	**4.22** (322.26%)	1.03 (3.48%)	1.53 (53.06%)	1.30 (30.13%)	6.45 (1.02×)	7:18 (1.12×)
2	**14.07** (1306.95%)	1.15 (14.54%)	1.21 (20.75%)	3.52 (252.39%)	8.08 (1.03×)	6:32 (1.15×)
3	**3.65** (264.90%)	1.07 (6.52%)	1.30 (29.63%)	2.07 (107.01%)	11.15 (1.04×)	18:29 (1.26×)
4	**5.17** (416.52%)	1.04 (4.38%)	1.55 (55.33%)	1.55 (54.87%)	21.03 (1.02×)	22:08 (1.08×)
5	**2.44** (144.36%)	1.05 (5.05%)	1.16 (15.73%)	2.03 (103.35%)	4.31 (1.04×)	2:25:46 (1.28×)
6	**31.49** (3048.66%)	1.04 (4.30%)	1.79 (79.40%)	1.89 (89.17%)	8.86 (1.05×)	1:59:46 (1.39×)
7	**2.13** (112.56%)	1.04 (4.17%)	1.12 (11.89%)	1.96 (96.04%)	34.42 (1.03×)	26:24:18 (1.48×)

Table 4.3: Results on the number of runs increase compared to the optBWT and resource usage. For each BWT variant we report the increase factor and the percentage increase (in brackets). Total overhead in time and memory for building the optBWT from scratch with respect to the inputBWT is shown in brackets. For the first four datasets we used the SAIS-based approach, and the BCR-based one for the last three.

dataset	len.	no. seq	no. runs increase	RAM (GB)		time (mm:ss)	
				BCR-based	SAIS-based	BCR-based	SAIS-based
NC_002516.2	50	56,379,600	**7.50** (650.20%)	1.02 (1.05×)	13.91 (1.03×)	25:13 (1.64×)	25:45 (1.12×)
P. aeruginosa	75	37,586,250	**4.96** (395.76%)	0.98 (1.03×)	13.84 (1.04×)	27:39 (1.58×)	25:59 (1.13×)
	100	28,189,800	**3.78** (277.91%)	0.52 (1.05×)	13.82 (1.04×)	30:39 (1.54×)	26:13 (1.17×)
	125	22,551,750	**3.08** (208.18%)	0.51 (1.04×)	13.83 (1.04×)	34:14 (1.57×)	26:33 (1.16×)
	150	18,792,900	**2.67** (167.48%)	0.51 (1.03×)	13.83 (1.04×)	36:26 (1.50×)	26:32 (1.18×)

Table 4.4: Results on the number of runs increase factor (percentage increase in brackets) compared to the optBWT, and resource usage for simulated datasets. Overhead in time and memory for building the optBWT from scratch using both approaches with respect to the inputBWT is shown in brackets.

4.5 Conclusion

In this chapter, we presented the first tool for computing the optimal BWT of string collections. We showed that the improvement obtained in terms of r reduction with respect to the input order is significant, and the overhead created by the computation of the optimal BWT is negligible. This makes our tool competitive with other tools for BWT computation in terms of running time and space usage and allows it to scale for large string collections. In particular, on real data the optBWT we obtained on the *Sindibis virus* datasets had up to 31 times fewer runs with only a 1.39× slowdown. Finally, on simulated data, we had the best results for the shortest sequences, even if we obtained a significant reduction in the number of runs for all sequence lengths.

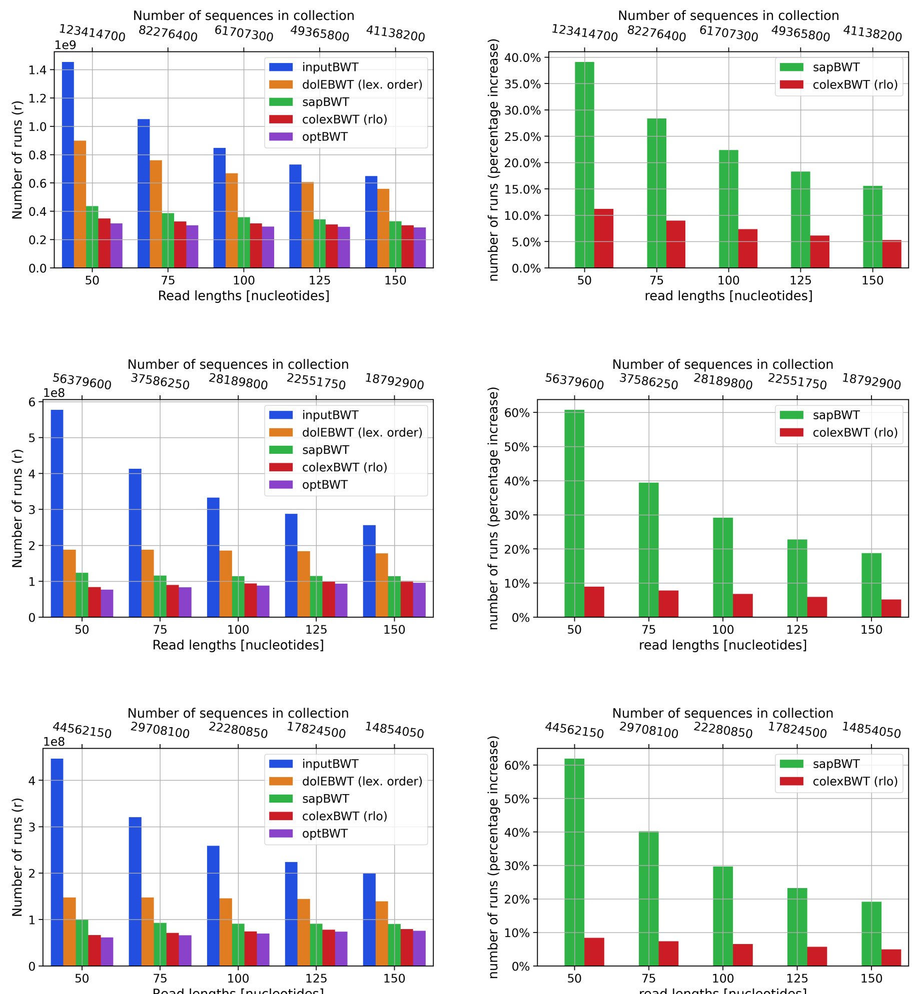


Fig. 4.2: Top to bottom: results regarding the number of runs on three simulated datasets of H. Sapiens chr.19 (cov. 100x), P. Aeruginosa (cov. 450x) and Salmonella (cov. 450x) varying read lengths. Left: number of runs. Right: percentage increase of sapBWT and colexBWT with respect to the optimal BWT.

5

Computing the eBWT using SAIS

In this chapter, we present `SAIS_for_eBWT`, a linear-time algorithm for computing the eBWT. The contents of this chapter were published in [18, 20].

Although it has been known for a long time how to efficiently compute the original BWT of a single string, this only holds if the string is terminated with an end-of-string character (dollar for short). In fact, if a sequence ends with a dollar, sorting the string's suffixes is equivalent to sorting the string's conjugates; thus, it is possible to compute the BWT using any suffix sorting algorithm, ensuring the transform's correctness. However, while the problem of sorting suffixes in linear time was solved in 2003 [63], sorting conjugates looks much more difficult. It is meaningful that until 2021, no linear-time algorithms were known for constructing the BWT of a sequence without appending a dollar or applying a preprocess to the input. The same problem extends to the BWT of string collections, where all variants use terminator characters to ensure the correct order of the rotations.

In 2007 Mantaci et al. [87] introduced the eBWT, a BWT variant that has different properties than all other BWT extensions; first of all, it uses the omega order. The computation of this order relation represents a bottleneck that prevented the efficient computation of this data structure for a long time. In fact, sorting the string's conjugates according to the omega order cannot be solved using suffix sorting since, in general, it requires more character comparisons and a more complex procedure than suffix sorting.

Since its introduction, there have been several attempts to overcome these difficulties and develop a linear time to compute the eBWT. The first one, proposed in 2007 [87], is based on sorting the strings' conjugates by using their H length prefixes, where H is a value computed using the Fine and Wilf theorem. This gives a bound on the maximum number of character comparisons we need to sort each conjugate. However, no linear time implementation was proposed for this algorithm, and using a naive radix sort procedure, the number of character comparisons would still be quadratic in the size of the input. In 2012 Hon et al. proposed an algorithm with an $O(n \log n)$ time complexity [58]. It first computes the GCA of the string collections via a recursive procedure and then uses the GCA to compute the eBWT. In 2014 Bonomo et al. [15] proposed an incremental algorithm that takes in input a sorted collection of Lyndon words for constructing the eBWT. At each step, this algorithm inserts a new character in

the eBWT built so far by computing the position of the corresponding rotation. It runs in $O(||\mathcal{M}||(t_1 + t_2))$ time, where t_1 and t_2 are the time for inserting a character and computing a rank query on a dynamic string. However, also in this case, no practical implementations were provided.

Finally, in 2019 Bannai et al. proposed a linear time algorithm for computing the Bijective BWT (BBWT) [6, 7] based on the well-known SAIS algorithm by Nong et al. [99]. The BBWT is a bijective variant of the BWT and is defined as the concatenation of the last characters of the lexicographically sorted rotations of all Lyndon factors of the input string. The Bannai et al. algorithm computes the BBWT by applying the Lyndon factorization to the input string and computing the eBWT of the resulting multiset of Lyndon factors. It follows that this algorithm can also be used to compute the eBWT of a string collection in linear time, even if it is first necessary to preprocess the input strings in a way that the strings are presented as a sequence of non-increasing Lyndon words, such as the output of the Lyndon factorization. However, this preprocessing can be expensive for large string collections.

In this chapter, we present the `SAIS_for_eBWT` algorithm, an improvement of the algorithm of Bannai et al. that removes the need for a preprocessing of the input. As a by-product of these results, we obtain the first algorithm to compute the BWT of a single string without a dollar and without preprocessing the input. We implemented this algorithm and included it in `cais`, a tool to compute the eBWT as well as three other types of BWT.

5.1 Overview

The rest of the chapter is organized as follows. In Section 5.2, we give first information about our algorithm. In Section 5.3, we explain how to compute the SAIS types cyclically and how this removes the need for preprocessing. In Section 5.4, and 5.5, we discuss in detail how to compute the GCA of a string collection and our algorithm's correctness and running time. In Section 5.6, we present how to use our algorithm to compute the BWT of a single sequence without using an end-of-string symbol and applying a preprocess. Finally, in Section 5.7, we present the `cais` tool implementing the `SAIS_for_eBWT` algorithm.

5.2 A simpler algorithm for computing the eBWT and GCA

In the following sections, we describe all steps of our algorithm, `SAIS_for_eBWT`, to compute the eBWT of a multiset of strings $\mathcal{M}$. Our algorithm is an adaptation of the well-known SAIS algorithm of Nong et al. [99], which computes the suffix array of a single string T ending with an end-of-string character \$ via a recursive procedure. Our adaptation is similar to that of Bannai et al. [7] for computing the BBWT, which can also be used for computing the eBWT. Even though our algorithm does not improve the latter asymptotically (both

are linear time), it is significantly simpler since it does not require first computing and sorting the Lyndon rotations of the input strings. In this chapter, we will focus on describing the differences between our algorithm and the original SAIS. Detailed explanations of SAIS can be found in [80, 99, 101].

The main differences between our algorithm and the original SAIS algorithm are: (1) we are comparing conjugates rather than suffixes, (2) we have a multiset of strings rather than just one string, (3) the comparison is done w.r.t. the omega-order rather than the lexicographic order, and (4) the strings are not terminated by an end-of-string symbol.

5.2.1 The eBWT and GCA of non-primitive strings

Even if in the original definition of eBWT [87], the input multiset $\mathcal{M}$ was assumed to contain only primitive strings, our definition is more general and also allows for non-primitive strings. For example, $\text{eBWT}(\{\texttt{ATA}, \texttt{TATA}\}) = \texttt{TATTAAA}$, with index set $\{2, 6\}$, while $\text{eBWT}(\{\texttt{ATA}, \texttt{TA}, \texttt{TA}\}) = \texttt{TATTAAA}$, with index set $\{2, 6, 7\}$. This does not affect the correctness of the transform. The linear-time algorithm for recovering the original multiset starting from the eBWT and its index set can be straightforwardly extended.

The following lemma shows how to construct the generalized conjugate array $\text{GCA}_{\mathcal{M}}$ of a multiset $\mathcal{M}$ of strings (not necessarily primitive), once we know the generalized conjugate array $\text{GCA}_{\mathcal{R}}$ of the multiset $\mathcal{R}$ of the roots of the strings in $\mathcal{M}$. The lemma follows straightforwardly from the fact that equal conjugates will end up consecutively in the GCA.

Lemma 4. *Let $\mathcal{M} = \{T_1, \ldots, T_m\}$ be a multiset of strings and let $\mathcal{R}$ the multiset of the roots of the strings in $\mathcal{M}$, i.e. $\mathcal{R} = \{S_1, \ldots, S_m\}$, where $T_i = (S_i^{r_i})$, with $r_i \geq 1$ for $1 \leq i \leq m$. Let $\text{GCA}_{\mathcal{R}}[1..K] = [(i_1, j_1), (i_2, j_2), \ldots, (i_K, j_K)]$, where $K = \sum_{i=1}^{m} |S_i|$. The generalized conjugate array is then given by*

$$\begin{aligned}\text{GCA}_{\mathcal{M}}[1..N] = [&(i_1, j_1), (i_1, j_1 + |S_{i_1}|), \ldots, (i_1, j_1 + (r_{i_1} - 1) \cdot |S_{i_1}|),\\ &(i_2, j_2), (i_2, j_2 + |S_{i_2}|), \ldots, (i_2, j_2 + (r_{i_2} - 1) \cdot |S_{i_2}|),\\ &\ldots\\ &(i_K, j_K), (i_K, j_K + |S_{i_K}|), \ldots, (i_K, j_K + (r_{i_K} - 1) \cdot |S_{i_K}|)],\end{aligned}$$

with $N = \sum_{i=1}^{m} |S_i| \cdot r_i$.

We will assume for the rest of the chapter that all strings in $\mathcal{M}$ are primitive since we can use Lemma 4 to compute the eBWT and GCA of $\mathcal{M}$ otherwise.

Finally, in Section 5.4.1, we show an alternative method to compute the eBWT of non-primitive strings without computing their roots and exponents by modifying the algorithm described in Section 5.4.

5.3 Cyclic types computation

To describe our algorithm, we need the following definition, which is the cyclic version of the classic definitions of L- and S-type in [99] (where S stands for smaller, L for larger, and LMS also called S^*, for leftmost-S):

Definition 2 (Cyclic types, *LMS*-substrings). *Let T be a primitive string of length at least 2, and $1 \leq i \leq |T|$. Position i of T is called* (cyclic) S-type *if $conj_i(T) <_{\text{lex}} conj_{i+1}(T)$, and* (cyclic) L-type *if $conj_i(T) >_{\text{lex}} conj_{i+1}(T)$. An S-type position i is called* (cyclic) LMS-position *if $i-1$ is L-type (where we view T as a cyclic string). An* LMS-substring *is a cyclic substring $T[i,j]$ of T such that both i and j are LMS-positions, but there is no LMS-position between i and j. Given a conjugate $conj_i(T)$, its* LMS-prefix *is the cyclic substring from i to the first LMS-position strictly greater than i (viewed cyclically).*

Since T is primitive, no two conjugates are equal, and in particular, no two adjacent conjugates are equal. Thus, the type of every position is defined.

Example 12. Let $\mathcal{M} = \{\texttt{GTACAACG}, \texttt{CGGCACACACGT}, \texttt{C}\}$, we assign the L- and S-types as follows (we mark *LMS*-positions with a $*$),

```
G T A C A A C G   C G G C A C A C A C G T
S L S L S S S S   S L L L S L S L S S S L
    *   *         *       *   *   *
```

The *LMS*-substrings are `ACA`, `AACGGTA`, `CGGCA`, and `ACGTC`. The *LMS*-prefix of the conjugate $\text{conj}_7(T_1) = \texttt{CGGTACAA}$ is `CGGTA`.

Lemma 5 (Cyclic type properties). *Let T be a primitive string of length at least 2. Let a_1 be the smallest and a_σ the largest character of the alphabet. Then the following hold, where T is viewed cyclically:*

1. *if $T[i] < T[i+1]$, then i is type S, and if $T[i] > T[i+1]$, then i is type L,*
2. *if $T[i] = T[i+1]$, then the type of i is the same as the type of $i+1$,*
3. *i is of type S iff $T[i'] > T[i]$, where $i' = \min\{i+j \mid j > 0, T[i+j] \neq T[i]\}$,*
4. *if $T[i] = a_1$, then i is type S, and if $T[i] = a_\sigma$, then i is type L.*

Proof. 1. follows from the fact that for all $b, c \in \Sigma$, if $b < c$ then for all $U, V \in \Sigma^*$, $bU \prec_\omega cV$; 2. follows by induction from the fact that for all $U, V \in \Sigma^*$, if $U \prec_\omega V$, then $cU \prec_\omega cV$; 3. and 4. follow from 2. by induction.

Corollary 1 (Linear-time cyclic type assignment). *Let T be a primitive string of length at least 2. Then all positions can be assigned a type in altogether at most $2|T|$ steps.*

Proof. Once the type of one position is known, then the assignment can be done in one cyclic pass over T from right to left by Lemma 5. Therefore, it suffices to find the type of one single position. Any position of character a_1 or of character a_σ will do; alternatively, any position i such that $T[i+1] \neq T[i]$, again by Lemma 5. Since T is primitive and has length at least 2, the latter must exist and can be found in at most one pass over T.

The advantage of processing a multiset of Lyndon words is that we know the last position of each Lyndon word is always an L-type. Thus, we can compute all types with a single left to right scan of each sequence. Moreover, if the sequences are presented in non creasing order, and we remove the Lyndon words of length 1, then the cyclic types match the classic SAIS types [7]. Note that this property is not true for general sequences. Our cyclic type assignment does

not have any restriction on the characteristics of input strings; thus, it does not require computing the Lyndon conjugates of the input strings and sorting them. In fact, for each sequence, we only need to find the first L-type with the first scan and then compute the other types with the second scan.

5.4 The Generalized conjugate array computation

Let N be the total length of the strings in $\mathcal{M}$. The algorithm constructs an initially empty array A of size N, which, at termination, will contain the GCA of $\mathcal{M}$. The algorithm also returns the set $\mathcal{I}$ containing the set of indices in A representing the positions of the strings of $\mathcal{M}$. The overall procedure consists of the following steps (each step will be explained below):

Algorithm `SAIS-for-`eBWT

Step 1 remove strings of length 1 from $\mathcal{M}$ (to be added back at the end)
Step 2 assign cyclic types to all positions of strings from $\mathcal{M}$
Step 3 use procedure `Induced Sorting` to sort cyclic *LMS*-substrings
Step 4 assign names to cyclic *LMS*-substrings; if all distinct, go to Step 6
Step 5 recurse on new multiset $\mathcal{M}'$, returning array A', map A' back to A
Step 6 use procedure `Induced Sorting` to sort all positions in $\mathcal{M}$, add length-1 strings in their respective positions, return $(A, \mathcal{I})$

At the heart of the algorithm is the procedure `Induced Sorting` of [99] (Algorithms 3.3 and 3.4), which is used once to sort the *LMS*-substrings (Step 3), and once to induce the order of all conjugates from the correct order of the *LMS*-positions (Step 6), as in the original SAIS. Before sketching this procedure, we need to define the order according to which the *LMS*-substrings are sorted in Step 2. Our definition of *LMS*-order extends the *LMS*-order of [99] to *LMS*-prefixes. It can be proved that these definitions coincide for *LMS*-substrings.

Definition 3 (*LMS*-order). *Given two strings S and T, let U resp. V be their LMS-prefixes. We define $U <_{LMS} V$ if either V is a proper prefix of U, or neither is a proper prefix of the other and $U <_{\text{lex}} V$.*

The procedure `Induced Sorting` for the conjugates of the multiset is analogous to the original one, except that strings are viewed cyclically. First, the array A is subdivided into so-called *buckets*, one for each character. For $c \in \Sigma$, let n_c denote the total number of occurrences of the character c in the strings in $\mathcal{M}$. Then the buckets are $[1, n_{a_1}], [n_{a_1} + 1, n_{a_1} + n_{a_2}], \ldots, [N - n_{a_\sigma} + 1, N]$, i.e., the k-th bucket will contain all conjugates starting with character a_k. The procedure `Induced Sorting` first inserts all *LMS*-positions at the end of their respective buckets, then induces the L-type positions in a left-to-right scan of A, and finally, induces the S-type positions in a right-to-left scan of A, possibly overwriting previously inserted positions. We need two pointers for each bucket $\mathbf{b}$, *head*$(\mathbf{b})$ and *tail*$(\mathbf{b})$, pointing to the current first resp. last free position.

Procedure `Induced Sorting` [99]

1. insert all *LMS*-positions at the end of their respective buckets; for all buckets $\mathbf{b}$, initialize *head*$(\mathbf{b})$, *tail*$(\mathbf{b})$ to the first resp. last position

2. induce the L-type positions in a left-to-right scan of A: for i from 1 to $N-1$, if $A[i] = (d, j)$ then $A[\mathit{head}(\mathit{bucket}(T_d[j-1]))] \leftarrow (d, j-1)$; increment $\mathit{head}(\mathit{bucket}(T_d[j-1]))$
3. induce the S-type positions in a right-to-left scan of A: for i from N to 2, if $A[i] = (d, j)$ then $A[\mathit{tail}(\mathit{bucket}(T_d[j-1]))] \leftarrow (d, j-1)$; decrement $\mathit{tail}(\mathit{bucket}(T_d[j-1]))$

At the end of this procedure, the *LMS*-substrings are listed in correct relative *LMS*-order (see Lemma 7), and they can be named according to their rank. For the recursive step, we define, for $i = 1, \ldots, m$, a new string T'_i, where each *LMS*-substring of T_i is replaced by its name (i.e. its rank). The algorithm is called recursively on $\mathcal{M}' = \{T'_1, \ldots, T'_m\}$ (Step 5).

Finally (Step 6), the array $A' = \mathrm{GCA}(\mathcal{M}')$ from the recursive step is mapped back into the original array, resulting in the placement of the *LMS*-substrings in their correct relative order. This is then used to induce the full array A. All length-1 strings T_i, which were removed in Step 1, can now be inserted between the L- and S-type positions in their bucket (Lemma 6). See Figure 5.1 for a full example.

5.4.1 Handling non-primitive strings in SAIS

Step 2 of our algorithm requires that each string in $\mathcal{M}$ contains at least two different characters to assign the cyclic types. If not, it means that $\mathcal{M}$ contains a non-primitive string with a root of length 1; thus, we cannot assign the cyclic types. Let T be a string of length n; if it is non-primitive, we know we can rewrite it in the following way, $T = S^r$, where $S = T[1..(n/r)]$ is the root, and $r \geq 2$ is the exponent. Let a position $1 \leq i \leq (n/r)$ in S be an LMS-type position, since S is repeated periodically all $\{i + (n/r \cdot 0), i + (n/r \cdot 1), ..., i + (n/r \cdot r - 1)\}$th positions will be LMS-positions. This implies that all cyclic LMS-substrings in R are repeated p times (see Example 13).

Example 13. Let $\mathcal{M} = \{\texttt{TGAGTGAG}, \texttt{ACCAACCAACCA}\}$ be a string collection containing two non-primitive words, where $\mathcal{R} = \{\texttt{TGAG}, \texttt{ACCA}\}$.

1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	10	11	12
T	G	A	G	T	G	A	G	A	C	C	A	A	C	C	A	A	C	C	A
L	L	S	S	L	L	S	S	S	L	L	S	S	L	L	S	S	L	L	S
		*				*					*				*				*

The two distinct *LMS*-substrings are `AGTGA`, and `AACCA`, and the new string collection constructed using the LMS substrings' ranks is $\mathcal{M}' = \{\texttt{aa}, \texttt{bbb}\}$. At this point we cannot proceed with step 2 since we cannot assign the cyclic types.

It follows that at each recursive step, the length of the root of T reduces at least by half until it reaches length 1. In this case, we end up with a string containing p equal characters $T' = c^1..c^p$ for which we cannot assign the cyclic types. However, we note that all conjugates in T' have the same rank, which is the same rank as a string containing a single occurrence of the character c, $T'' = c^1$. It follows that in Step 1, we can remove all non-primitive strings with

a root of length 1 along with all strings of length 1 and add their conjugates in the correct position in Step 6.

5.5 Correctness and running time

The following lemma shows that the individual steps of `Induced Sorting` are applicable for the ω-order on conjugates of a multiset (part 1), that L-type conjugates (of all strings) come before the S-type conjugates within the same bucket (part 2), and that length-1 strings are placed between S-type and L-type conjugates (part 3). The second property was originally proved for the lexicographic order between suffixes in [64]:

Lemma 6 (Induced sorting for multisets). *Let $U, V \in \Sigma^*$.*

1. *If $U \prec_\omega V$, then for all $c \in \Sigma$, $cU \prec_\omega cV$.*
2. *If $U[i] = V[j]$, i is an L-type position, and j an S-type position, then $conj_i(U) \prec_\omega conj_j(V)$.*
3. *If $U[i] = V[j] = c$, i is an L-type position, and j an S-type position, then $conj_i(U) \prec_\omega c \prec_\omega conj_j(V)$.*

Proof. *1.* follows directly from the definition of ω-order. *3.* implies *2.* For *3.*, let i' be the nearest character following i in U such that $U[i'] \neq c$. By Lemma 5, $U[i'] < c$, and thus $\text{conj}_i(U) <_{\text{lex}} c^{|U|}$, and therefore, $conj_i(U) \prec_\omega c$. Analogously, if j' is the next character in V s.t. $V[j'] \neq c$, then by Lemma 5, $V[j'] > c$, and therefore, $c \prec_\omega conj_j(V)$.

Next, we show that after applying procedure `Induced Sorting`, the conjugates will appear in A such that they are correctly sorted w.r.t. to the *LMS*-order of their *LMS*-prefixes, while the order in which conjugates with identical *LMS*-prefixes appear in A is determined by the input order of the *LMS*-positions.

Lemma 7 (Extension of Thm. 3.12 of [99]). *Let $T_1, T_2 \in \mathcal{M}$, let U be the LMS-prefix of $conj_i(T_1)$, with i' the last position of U; let V be the LMS-prefix of $conj_j(T_2)$, and j' the last position of V. Let k_1 be the position of $conj_i(T_1)$ in array A after the procedure* `Induced Sorting`*, and k_2 that of $conj_j(T_2)$.*

1. *If $U <_{LMS} V$, then $k_1 < k_2$.*
2. *If $U = V$, then $k_1 < k_2$ if and only if $conj_{i'}(T_1)$ was placed before $conj_{j'}(T_2)$ at the start of the procedure.*

Proof. Both claims follow from Lemma 6, and the fact that from one *LMS*-position to the previous one, there is exactly one run of L-type positions, preceded by one run of S-type positions.

The next lemma shows that the *LMS*-order of the *LMS*-prefixes respects the ω-order.

Lemma 8. *Let $S, T \in \Sigma^*$, let U be the LMS-prefix of S and V the LMS-prefix of T. If $U <_{LMS} V$ then $S \prec_\omega T$.*

Step 1 - remove strings of length 1 from $\mathcal{M}$

Step 2 - assign cyclic types to all positions of strings from $\mathcal{M}$

	T_1								T_2												T_3
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	10	11	12	1
$\mathcal{M} = \{$	G	T	A	C	A	A	C	G ,	C	G	G	C	A	C	A	C	A	C	G	T ,	C }
	S	L	S	L	S	S	S	S	S	L	L	L	S	L	S	L	S	S	S	L	
			*		*				*				*		*		*				

Step 3 - use procedure `Induced Sorting` to sort cyclic *LMS*-substrings

	A						C							G					T	
S*		2	2	2	1	1							2							
		5	7	9	3	5							1							
L							2	2	2	1				2	2				1	2
$\longrightarrow$							4	6	8	4				3	2				2	12
S	1	2	2	1	1	2					2	1	2			1	1	2		
$\longleftarrow$	5	5	7	3	6	9					1	7	10			8	1	11		

A						C							G					T	
1	2	2	1	1	2	2	2	2	1	2	1	2	2	2	1	1	2	1	2
5	5	7	3	6	9	4	6	8	4	1	7	10	3	2	8	1	11	2	12
*	*	*	*		*					*									

Step 4 - Assign names to cyclic *LMS*-substrings

A A C G G T A	a
A C A	b
A C G T C	c
C G G C A	d

$T'_1 = b\ a$
$T'_2 = d\ b\ b\ a$

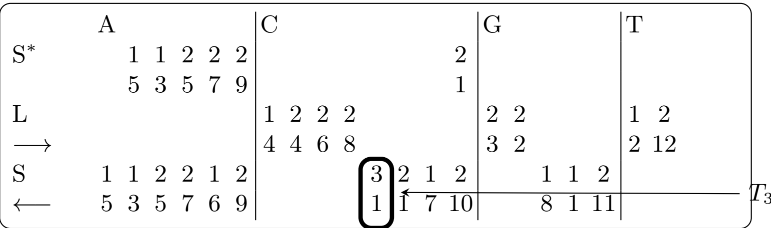
Step 5 - recurse on new string multiset $\mathcal{M}'$

	T'_1		T'_2			
	1	2	1	2	3	4
$\mathcal{M}' = \{$	b	a ,	d	b	b	c }
	L	S	L	S	S	S
		*		*		

	a	b			c	d
S*	1			2		
	2			2		
L		1				2
$\longrightarrow$		1				1
S	1		2	2	2	
$\longleftarrow$	2		2	3	4	

A'	1	1	2	2	2	2
	2	1	2	3	4	1

Step 6 - use procedure `Induced Sorting` to sort all positions in $\mathcal{M}$, add length-1 strings in their respective positions



	A						C								G					T	
S*		1	1	2	2	2								2							
		5	3	5	7	9								1							
L							1	2	2	2					2	2				1	2
$\longrightarrow$							4	4	6	8					3	2				2	12
S	1	1	2	2	1	2					3	2	1	2			1	1	2		
$\longleftarrow$	5	3	5	7	6	9					1	1	7	10			8	1	11		

Generalized conjugate array of $\mathcal{M}$

GCA	1	1	2	2	1	2	1	2	2	2	**3**	**2**	1	2	2	2	1	**1**	2	1	2
	5	3	5	7	6	9	4	4	6	8	**1**	**1**	7	10	3	2	8	**1**	11	2	12
eBWT	C	T	C	C	A	C	A	G	A	A	C	T	A	A	G	C	C	G	C	G	G

Fig. 5.1: The algorithm `SAIS-for-`eBWT on Example 13. The start positions of input strings are marked in bold.

Proof. If neither U nor V is a proper prefix one of the other, then there exists an index i s.t. $S[i] = U[i] < V[i] = T[i]$, and therefore, $S \prec_\omega T$. Otherwise, V is a proper prefix of U. Let $i = |V|$ and $c = V[i]$. Since both U and V are *LMS*-prefixes, with i being the last position of V but not of U, this implies that $V[i] = T[i]$ is of type S, while $U[i] = S[i]$ is of type L. Let j be the next character in S s.t. $S[j] \neq c$, and k be the next character in T s.t. $T[k] \neq c$. By Lemma 5, $S[j] < c$, $T[k] > c$, and by definition of j, k all characters inbetween equal c. Then for $i' = \min(j, k)$, we have $S[i'] < T[i']$, with i' being the first position where S and T differ. Therefore, $S \prec_\omega T$.

Theorem 1. *Algorithm* `SAIS-for-`eBWT *correctly computes the* GCA *and* eBWT *of a multiset of strings* $\mathcal{M}$ *in time* $O(N)$*, where* N *is the total length of the strings in* $\mathcal{M}$*.*

Proof. By Lemma 5, Step 2 correctly assigns the types. Step 3 correctly sorts the *LMS*-substrings by Lemma 7. It follows from Lemma 8 that the order of the conjugates of the new strings T_i' coincides with the relative order of the *LMS*-conjugates. In Step 6, the *LMS*-conjugates are placed in A in correct relative order from the recursion; by Lemmas 7 and 8, this results in the correct placement of all conjugates of strings of length > 1, while the positioning of the length-1 strings is given by Lemma 6.

For the running time, note that Step 1 takes time proportional to $2N$. The `Induced Sorting` procedure also runs in linear time $O(N)$. Finally, since no two *LMS*-positions are consecutive, and we remove strings of length 1, the problem size in the recursion step is reduced to at most $N/2$.

5.6 Computing the BWT for one single string without dollar

The special case where $\mathcal{M}$ contains one single string T leads to a new algorithm for computing the BWT, since for a singleton set, the eBWT coincides with the BWT. To the best of our knowledge, this is the first linear-time algorithm for computing the BWT *of a string without an end-of-string character* that uses neither Lyndon rotations nor end-of-string characters. The modified procedure consists of the following steps:

Algorithm `SAIS-for-`eBWT `(one string)`

Step 1 if T has length one, return $(A, \mathcal{I}) = (1, 1)$
Step 2 assign cyclic types to all positions of T
Step 3 use procedure `Induced Sorting` to sort cyclic *LMS*-substrings
Step 4 assign names to cyclic *LMS*-substrings; if all distinct, go to Step 6
Step 5 recurse on new string T', returning array A', map A' back to A
Step 6 use procedure `Induced Sorting` to sort all positions in T, return $(A, \mathcal{I})$

We demonstrate the algorithm on a well-known example, $T =$ `banana`. We get the following types, from left to right: *LSLSLS*, and all three S-type positions are *LMS*. We insert $2, 4, 6$ into the array A; after the left-to-right pass,

indices are in the order $2, 4, 6, 1, 3, 5$, and after the right-to-left pass, in the order $6, 2, 4, 1, 3, 5$. The LMS-substring `aba` (pos. 6) gets the name A, and the LMS-substring `ana` (pos. 2,4) gets the name B. In the recursive step, the new string $T' =$ `ABB`, with types SLL and only one LMS-position 1, the GCA gets induced in just one pass: $1, 3, 2$. This maps back to the original string: $6, 2, 4$, and one more pass over the array A results in $6, 4, 2, 1, 5, 3$ and the BWT `nnbaaa`.

Step 2

1	2	3	4	5	6
b	a	n	a	n	a
L	S	L	S	L	S
	*		*		*

Step 3

	a			b	n	
S*	2	4	6			
L				1	3	5
S	6	2	4			
	6	2	4	1	3	5

Step 4

6	a	b	a	A
2	a	n	a	B
4	a	n	a	B

Step 5

			A	B	
1	2	3	1		
A	B	B		3	2
S	L	L	1	3	2
*					

Step 6

	a			b	n	
	6	4	2			
				1	5	3
GCA	6	4	2	**1**	5	3
BWT	n	n	b	a	a	a

Fig. 5.2: Example for computing the BWT for one string, start index marked in bold.

5.7 The `cais` tool

We implemented the `SAIS_for_eBWT` algorithm in the `cais` (Conjugate array induced sorting) tool (`https://github.com/davidecenzato/cais.git`) for computing four different BWT variants, the BWT of a single string without the end-of-string character, the BBWT, the eBWT and the dollar eBWT.

5.7.1 Implementation

The `cais` tool is implemented in C++ and only requires the `sdsl-lite` library (`https://github.com/simongog/sdsl-lite.git`), a modern `gcc` compiler, and a make command to be installed. It takes as input either string collections in fasta or fastq format or single texts in plain ASCII format. The user can set the command-line arguments to select which of the four transforms to compute.

Given a string collection $\mathcal{M} = \{T_1, \ldots, T_m\}$, as for the eBWT, we concatenate all strings into a single string $C[1..N] = T_1T_2, ..., T_m$, and construct a bitvector D of length $N + 1$ marking the starting positions of the input strings in the concatenation. Thus, the ith entry of D is $D[i] = 1$ if $C[i]$ is the starting position of a string in C, $D[i] = 0$ otherwise. The $(N+1)$th entry of D is always set to 1. Finally, we compute the data structures supporting rank and select queries on D using `sdsl-lite`. Since we are not using dollar characters, the D bitvector is essential to identify the string boundaries in C and process the strings circularly. By default, the tool computes a plain bitvector that takes $N + 1$ bits in memory that supports constant time rank and select queries. We also allow the user to select an Elias-Fano encoded bit vector implementation to represent sparse bivectors.

The `cais` tool uses the `SAIS_for_eBWT` algorithm to compute the GCA of the concatenation $C[1..N]$ by sorting the strings' conjugates circularly. It takes as input C and D and at each recursive step, computes D' of the new string

C' constructed by renaming the LMS-substrings. The eBWT is computed and written directly to disk using the following equation,

$$\text{eBWT}[i] = \begin{cases} C[\text{GCA}[i]-1], & \text{if } D[i]=0, \\ C[\text{select}_1(D, \text{rank}_1(D, \text{GCA}[i])+1)-1], & \text{otherwise}, \end{cases}$$

where $\text{rank}(D, i)$ is the number of 1 in $D[1..i]$, while $\text{select}(D, i)$ is the position of the ith 1 in D. The dollar BWT is computed similarly, just appending the dollar characters at the end of the strings.

As for the BBWT, the procedure is identical except for the way we construct C and D. Given an input string T, we initialize $C = T$ and compute D by marking the starting positions of the Lyndon factors of T.

Finally, the `cais` tool uses the `SAIS_for_eBWT (one string)` algorithm to compute the CA of a string $T[1..n]$ by sorting its conjugates circularly. The BWT is computed and written directly to disk using the following equation,

$$\text{BWT}[i] = \begin{cases} T[\text{CA}[i]-1], & \text{if } i > 0, \\ T[n], & \text{otherwise}. \end{cases}$$

6

Computing the eBWT of large string collections

In this chapter, we present `pfpebwt`, a tool implementing a linear-time algorithm for computing the eBWT of large and repetitive string collections using prefix-free parsing preprocessing. Finally, in the last section, we present an algorithm to optimize the size of the PFP data structures. The contents of this chapter were published in [17, 18], and [103].

Computing the eBWT, as well as other BWT variants, using a SAIS-based algorithm not only guarantees that the algorithm runtime will grow linearly with the size of the input, but it is also fast in practice. Thus, it is not surprising that two of the fastest implementations available to compute the SA and the BWT of a text, `libsais` and `sais-lite-lcp`, are based on SAIS. However, this type of algorithm requires keeping the whole SA or GCA in internal memory during the computation. Due to this, applying these algorithms to large string collections is impractical since it would require an enormous amount of internal memory. In order to overcome this limitation, several attempts were made to design algorithms implementing the suffix sorting procedure using the external memory [12, 59, 77, 98]. However, they trade more efficient memory usage with a slower running time.

In 2018 Boucher et al. [21, 22], introduced the *prefix-free parsing* (PFP). PFP is a preprocessing technique that was originally introduced to construct the BWT of large and repetitive texts. Briefly, with one scan, the PFP divides the input in overlapping segments, called *phrases*, of variable length, which are then used to construct what is referred to as the *dictionary* D and *parse* P of the input. Then, with a separate procedure, it is possible to construct the BWT of the input directly from D and P; thus using space proportional to the combined size of these two data structures. The key of the PFP is that if the input collection is repetitive enough, the combined size of D and P will be much smaller than the size of the original input.

In Section 3, we listed a number of tools for computing the BWT of large string collections. However, all of them use dollars, and most of them produce a transform that depends on the input order, thus, losing the original properties of the eBWT. In this chapter, we show how to combine our new eBWT construction, `SAIS_for_eBWT` presented in Chapter 5, with a variant of a PFP to develop the first algorithm for computing the eBWT of large repetitive string collections fast and keeping all computation in internal memory. We imple-

mented our algorithm in the `pfpebwt` tool and measured the time and memory required to build the eBWT on three genomic sequence datasets.

The PFP algorithm uses a simple and fast procedure that computes the parse and dictionary data structures with a single pass on the input text. This algorithm delivers excellent results in practice; however, it gives no guarantees on the output size. In [103], we collaborated on developing an algorithm inspired by the well-known RePair by Larsson and Moffat [71] to optimize the PFP size. This algorithm is based on joining pairs of phrases to reduce the combined size of the parse and dictionary. This algorithm, implemented in the `rePFP-CST` tool, can significantly reduce the combined size of the PFP data structures on real datasets and unlock the computation of a compressed suffix tree data structure for one terabyte of data.

6.1 Overview

The rest of the chapter is organized as follows. In Section 6.2, we show how to extend the PFP algorithm to cyclic strings. In Section 6.3, we present the algorithm to compute the eBWT starting from the parse and dictionary constructed using the cyclic PFP. In Section 6.4, and 6.5, we present our experimental results and give final comments. Finally, in Section 6.6, we give some highlights of an algorithm inspired by the RePair of Larsson and Moffat to reduce the size of the PFP data structures.

6.2 The cyclic prefix-free parsing

In this section, we show how to extend the prefix-free parsing to build the eBWT. We define the *cyclic prefix-free parse* for a multiset of strings $\mathcal{M} = \{T_1, T_2, \ldots, T_m\}$ (with $|T_i| = n_i$, $1 \leq i \leq m$) as the multiset of parses $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ with dictionary D, where we consider T_i as circular, and P_i is the parse of T_i. We denote by p_i the length of the parse P_i.

Next, given a positive integer w, let E be a set of strings of length w called *trigger strings*. We assume that each string $T_h \in \mathcal{M}$ has length at least w and at least one cyclic factor in E.

We divide each string $T_h \in \mathcal{M}$ into overlapping phrases as follows: a phrase is a circular factor of T_h of length $> w$ that starts and ends with a trigger string and has no internal occurrences of a trigger string. The set of phrases obtained from strings in $\mathcal{M}$ is the dictionary D. The parse P_h can be computed from the string T_h by replacing each occurrence of a phrase in T_h with its lexicographic rank in D.

Example 14. Let $\mathcal{M} = \{T_1 : \texttt{CACGTGCTAT}, T_2 : \texttt{CCACTTGCTAGA}, T_3 : \texttt{CACTTGCTAT}\}$ and let $E = \{\texttt{AC}, \texttt{GC}\}$. The dictionary D of the multiset of parses $\mathcal{P}$ of $\mathcal{M}$ is $D = \{\texttt{ACCAC}, \texttt{ACGTGC}, \texttt{ACTTGC}, \texttt{GCTAGAC}, \texttt{GCTATCAC}\}$ and $\mathcal{P} = \{\texttt{2}\,\texttt{5}, \texttt{3}\,\texttt{4}\,\texttt{1}, \texttt{3}\,\texttt{5}\}$, where $P_2 = \texttt{3}\,\texttt{4}\,\texttt{1}$ means that the parsing of T_2 is given by the third, the fourth and the first phrase of the dictionary. Note that the string T_2 has a trigger string `AC` that spans the first position of T_2.

We denote by $\mathcal{S}$ the set of suffixes of D having a length greater than w. The first important property of the dictionary D is that the set $\mathcal{S}$ *prefix-free*, i.e., no string in $\mathcal{S}$ is the prefix of another string of $\mathcal{S}$. This follows directly from [21].

Example 15. Continuing Example 14, we have that

$$\begin{aligned}\mathcal{S} = \{&\texttt{ACCAC}, \texttt{ACGTGC}, \texttt{ACTTGC}, \texttt{AGAC}, \texttt{ATCAC}, \texttt{CAC}, \texttt{CCAC}, \texttt{CGTGC},\\ &\texttt{CTAGAC}, \texttt{CTATCAC}, \texttt{CTTGC}, \texttt{GAC}, \texttt{GCTAGAC}, \texttt{GCTATCAC}, \texttt{GTGC},\\ &\texttt{TAGAC}, \texttt{TATCAC}, \texttt{TCAC}, \texttt{TGC}, \texttt{TTGC}\}\end{aligned}$$

6.3 Computing the eBWT using the PFP

The computation of eBWT from the prefix-free parse consists of three steps: computing the cyclic prefix-free parse of $\mathcal{M}$ (denoted as $\mathcal{P}$), computing the eBWT of $\mathcal{P}$ by using the algorithm described in Section 5.2; and lastly, computing the eBWT of $\mathcal{M}$ from the eBWT of $\mathcal{P}$ using the lexicographically sorted dictionary $D = \{D_1, D_2, \ldots, D_{|D|}\}$ and its prefix-free suffix set $\mathcal{S}$. We now describe the last step as follows. We define δ as the function that uniquely maps each character of $T_h[j]$ to the pair (i, k), where with $1 \le i \le p_h$, $1 \le k \le |P_h[i]| - w$, and $T_h[j]$ corresponds to the k-th character of the $P_h[i]$-th phrase of D. We call i and k the *position* and the *offset* of $T_h[j]$, respectively. Furthermore, we define α as the function that uniquely associates to each conjugate $conj_j(T_h)$ the element $s \in \mathcal{S}$ such that s is the k-th suffix of the $P_h[i]$-th element of D, where $(i, k) = \delta(T_h[j])$. By extension, i and k are also called the *position* and the *offset* of the suffix $\alpha(\mathrm{conj}_j(T_h))$.

Example 16. In Example 14, $\delta(T_2[4]) = (1, 2)$ since $T_2[4]$ is the second character (offset 2) of the phrase ACTTGC, which is the first phrase (position 1) of P_2. Moreover, $\alpha(\mathrm{conj}_4(T_2)) =$ CTTGC since CTTGC is the suffix of D_3, which is prefix of $\mathrm{conj}_4(T_2) =$ CTTGCTAGACCA.

Lemma 9. *Given two strings $T_g, T_h \in \mathcal{M}$, if $\alpha(conj_i(T_g)) <_{\mathrm{lex}} \alpha(conj_j(T_h))$ it follows that $conj_i(T_g) \prec_\omega conj_j(T_h)$.*

Proof. It follows from the definition of α that $\alpha(conj_i(T_g))$ and $\alpha(conj_j(T_h))$ are prefixes of $conj_i(T_g)$ and $conj_j(T_h)$, respectively.

Proposition 3. *Given two strings $T_g, T_h \in \mathcal{M}$. Let $conj_i(T_g)$ and $conj_j(T_h)$ be the i-th and j-th conjugates of T_g and T_h, respectively, and let $(i', g') = \delta(T_g[i])$ and $(j', h') = \delta(T_h[j])$. Then $conj_i(T_g) \prec_\omega conj_j(T_h)$ if and only if either $\alpha(conj_i(T_g)) <_{\mathrm{lex}} \alpha(conj_j(T_h))$, or $conj_{i'+1}(P_g) \prec_\omega conj_{j'+1}(P_h)$, i.e., $P_g[i']$ precedes $P_h[j']$ in* eBWT($\mathcal{P}$).

Proof. By definition of α, $\mathrm{conj}_i(T_g) = \alpha(\mathrm{conj}_i(T_g))T_g[i+g'']T_g[i+g''+1]\ldots T_g[i-1]$ and $\mathrm{conj}_j(T_h) = \alpha(\mathrm{conj}_j(T_h))T_h[j+h'']T_h[j+h''+1]\ldots T_h[j-1]$, where $g'' = |\alpha(\mathrm{conj}_i(T_g))|$ and $h'' = |\alpha(\mathrm{conj}_j(T_h))|$, respectively. Moreover, $\mathrm{conj}_i(T_g) \prec_\omega \mathrm{conj}_j(T_h)$ if and only if either $\alpha(\mathrm{conj}_i(T_g)) <_{\mathrm{lex}} \alpha(\mathrm{conj}_j(T_h))$ or $\mathrm{conj}_{i+g''-w}(T_g) \prec_\omega \mathrm{conj}_{j+h''-w}(T_h)$, where w is the length of trigger strings. It is easy to verify that

the position of $T_g[i + g'' - w]$ and $T_h[j + h'' - w]$ is $i' + 1$ and $j' + 1$, respectively. Moreover, since $T_g[i + g'' - w]$ and $T_h[j + h'' - w]$ are the first character of a phrase, we have that $\text{conj}_{i+g''-w}(T_g) \prec_\omega \text{conj}_{j+h''-w}(T_h)$ if and only if $\text{conj}_{i'+1}(P_g) \prec_\omega \text{conj}_{j'+1}(P_h)$.

Next, using Proposition 3, we define how to build the eBWT of the multiset of strings $\mathcal{M}$ from $\mathcal{P}$ and D. First, we note that we will iterate through all the suffixes in $\mathcal{S}$ in lexicographic order and build the eBWT of $\mathcal{M}$ in blocks corresponding to the suffixes in $\mathcal{S}$. Hence, it follows that we only need to describe how to build an eBWT block corresponding to a suffix $s \in \mathcal{S}$. Given $s \in \mathcal{S}$, we let $\mathcal{S}_s$ be the set of the lexicographic ranks of the phrases of D that have s as a suffix, i.e., $\mathcal{S}_s = \{i \mid 1 \leq i \leq |D|, s \text{ is a suffix of } D_i \in D\}$. Moreover, given the string $T_h \in \mathcal{M}$, we let $\text{conj}_i(T_h)$ be the i-th conjugate of T_h, let j and k be the position and offset of $T_h[i]$, and lastly, let p be the position of $P_h[j]$ in eBWT($\mathcal{P}$). We define $f(p, k) = D_{P_h[j]}[k-1]$ if $k > 1$, otherwise $f(p, k) = D_{P_h[j-1]}[|D_{P_h[j-1]}| - w]$ where we view P_h as a cyclic string.

Example 17. In Example 14, eBWT($\mathcal{P}$) = `4515323`. Let us consider $\text{conj}_4(T_2)$ and $\text{conj}_3(T_3)$ that are both mapped to the suffix `CTTGC` by the function α. By using Example 16, the positions and the offsets of $T_2[4]$ and $T_3[3]$ are 1 and 2, respectively. The positions of $P_2[1] = P_3[1] = 3$ in eBWT($\mathcal{P}$) are 5 and 7, respectively, because $\text{conj}_2(P_2) \prec_\omega \text{conj}_2(P_3)$. This implies that $\text{conj}_4(T_2) \prec_\omega \text{conj}_3(T_3)$ by Proposition 3. Furthermore, $f(5, 2) = T_2[3] = A$.

Finally, we let $\mathcal{O}_s$ be the set of pairs (p, c) such that for all $d \in \mathcal{S}_s$, p is the position of an occurrence of d in eBWT($\mathcal{P}$), and c is the character resulting in the application of the f function considering as k the offset of s in D_d, i.e., $c = f(p, |D_d| - |s| + 1)$. Formally, $\mathcal{O}_s = \{(p, f(p, |D_{\text{eBWT}(\mathcal{P})[p]}| - |s| + 1) \mid \text{eBWT}(\mathcal{P})[p] \in \mathcal{S}_s\}$.

Example 18. In Example 14, if $s = \texttt{CAC} \in \mathcal{S}$ and $\mathcal{S}_s = \{\texttt{1}, \texttt{5}\}$, where `1` : `ACCAC` and `5` : `GCTATCAC`, then it follows that $\mathcal{O}_s = \{(3, \texttt{C}), (2, \texttt{T}), (4, \texttt{T})\}$ since the phrase `1` is in position 3 in the eBWT($\mathcal{P}$) and the suffix `CAC` starts in position 3 of D_1, the character preceding the occurrences of `CAC` corresponding to the phrase `1` is `C`. Analogously, the phrase `5` is in positions 2 and 4 in the eBWT($\mathcal{P}$) and the suffix `CAC` starts in position 6 of D_5, hence the character preceding the occurrences of `CAC` corresponding to the phrase `5` is `T`.

To build the eBWT block corresponding to $s \in \mathcal{S}$, we scan the set $\mathcal{O}_s$ in increasing order of the first element of the pair, i.e., the position of the occurrence in eBWT($\mathcal{P}$), and concatenate the values of the second element of the pair, i.e., the character preceding the occurrence of s in T_h.Note that if all the occurrences in $\mathcal{O}_s$ are preceded by the same character c, we do not need to iterate through all the occurrences but rather concatenate $|\mathcal{O}_s|$ copies of the character c.

Example 19. In Example 14, eBWT($\mathcal{M}$) = `GCCCTTTTCTAAGGGAAATTTCCCCAATGTCC`, where the block of the eBWT corresponding to the suffix $s = \texttt{CAC} \in \mathcal{S}$ is underlined. Given $\mathcal{O}_s = \{(3, \texttt{C}), (2, \texttt{T}), (4, \texttt{T})\}$, we generate the block by sorting $\mathcal{O}_s$ by the first element of each pair – resulting in $\mathcal{O}_s = \{(2, \texttt{T}), (3, \texttt{C}), (4, \texttt{T})\}$ – and concatenating the second element of each pair obtaining `TCT`.

6.3.1 Keeping track of the first rotations

So far, we have shown how to compute the first component of the eBWT. Now we show how to compute the second component of the eBWT i.e., the set of indices marking the first rotation of each string. The idea is to keep track of the starting positions of each text in the parse by marking the offset of the first position of each string in the last phrase of the corresponding parse. We propagate this information during the computation of the eBWT of the parse. When scanning the suffixes of $\mathcal{S}$, we check if one of the phrases sharing the same suffix $s \in \mathcal{S}$ is marked as a phrase containing a starting position, and if the offset of the starting position coincides with the offset of the suffix. If so, when generating the elements of $\mathcal{O}_s$, we mark the element corresponding to the occurrence of the first rotation of a string, and we output the index of the eBWT when that element is processed.

6.3.2 Implementation notes

In practice, as in [21], we implicitly select the set of trigger strings E, by rolling a Karp-Rabin hash over consecutive windows of size w and take as trigger strings of length w all windows such that their hash value is congruent 0 modulo a parameter p. In our version of the PFP, we also need to ensure that there is at least one trigger string on each sequence of the collection. Hence, we change the way we select the trigger strings as follows. We define a set $\mathcal{D}$ of remainders, and we select a window of length w as a trigger string with hash value congruent d modulo p if $d \in \mathcal{D}$. Note that if we set $\mathcal{D} = \{0\}$, we obtain the same set of trigger strings as in the original definition. We choose the set $\mathcal{D}$ in a greedy way. We start with $\mathcal{D} = \{0\}$ by scanning the set of sequences and checking if the current sequence has a trigger string according to the current $\mathcal{D}$. As soon as we find one, we move to the next sequence. If we don't find any trigger string, we take the remainder of the last window we checked, and we include it in the set $\mathcal{D}$.

We note that we consider $\mathcal{S}$ to be the set of suffixes of the phrases of D such that $s \in \mathcal{S}$ is not a phrase in D, nor it has length smaller than w in the implementation. This allows us to compute f more efficiently since we can compute the preceding character of all the occurrences of a suffix in $\mathcal{S}$ from its corresponding phrase in D. Moreover, as in [21], for each phrase in D, we keep an ordered list of their occurrences in the eBWT of the parse. For a given suffix $s \in \mathcal{S}$, we do not generate $\mathcal{O}_s$ all at once and sort it – but rather, we visit the elements of $\mathcal{O}_s$ in order using a heap data structure as we merge the ordered lists of the occurrences in the eBWT of the parse of the phrases that share the same suffix s.

6.4 Experimental results

We implemented the algorithm for building the eBWT and measured its performance on real biological data. We performed the experiments on a server with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz with 16 cores and 62 gigabytes of

Name	Description	σ	$n/10^6$	n/r
chr19	Human chromosome 19	5.00	121 086.62	2199.21
salmonella	*Salmonella* genomes	4.00	48 791.75	112.72
sars-cov2	SARS-CoV2 genomes	5.00	11 930.96	1424.65

Table 6.1: Datasets used in the experiments. We give the alphabet size in column 3. We report the length of the file and the ratio of the length to the number of runs in the eBWT in columns 4 and 5, respectively.

RAM running Ubuntu 16.04 (64bit, kernel 4.4.0). The compiler was `g++` version 9.4.0 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. We recorded the runtime and memory usage using the wall clock time, CPU time, and maximum resident set size from `/usr/bin/time`. The source code is available online at: https://github.com/davidecenzato/PFP-eBWT.

We compared our method (`pfpebwt`) with the BCR algorithm implementation of [73] (`ropebwt2`), `gsufsort` [81], and `egap` [36]. We did not compare against `G2BWT` [33], `lba` [14], and `BCR` [8] since they are currently implemented only for short reads[1]. We did not compare against `egsa` [82] since it is the predecessor of `egap` or against methods that construct the BWT of a multiset of strings using one of the methods we evaluated against, i.e., `LiME` [55], `BEETL` [32], `metaBEETL` [4], and `ebwt2snp` [109, 110].

6.4.1 Datasets

We evaluated our method using 2,048 copies of human chromosomes 19 from the 1000 Genomes Project [122]; 10,000 *Salmonella* genomes taken from the GenomeTrakr project [120], and 400,000 SARS-CoV2 genomes from EBI's COVID-19 data portal [124]. The sequence data for the *Salmonella* genomes were assembled, and the assembled sequences that had length less than 500 bp were removed. In addition, we note that we replaced all degenerate bases in the SARS-CoV2 genomes with N's and filtered all sequences with more than 95% N's. A brief description of the datasets is reported in Table 6.1. We used 12 sets of variants of human chromosome 19 (`chr19`), containing 2^i variants for $i = 0, \ldots, 11$ respectively. We used 6 collections of *Salmonella* genomes (`salmonella`) containing 50, 100, 500, 1,000, 5,000, and 10,000 genomes respectively. We used 5 sets of SARS-CoV2 genomes (`sars-cov2`) containing 25,000, 50,000, 100,000, 200,000, 400,000 genomes, respectively. Each collection is a superset of the previous one.

6.4.2 Experimental setup

We run `pfpebwt` and `ropebwt2` with 16 threads, and `gsufsort` and `egap` with a single thread since they do not support multi-threading. Using `pfpebwt`, we set $w = 10$ and $p = 100$. Furthermore, for `pfpebwt` on the `salmonella` dataset, we used up to three different remainders to build the eBWT. We used `ropebwt2`

[1] `G2BWT` crashed and `BCR` did not terminate within 48 hours with the smallest of each dataset; `lba` works only with sequences of length up to 255

with the `-R` flag to exclude the reverse complement of the sequences from the computation of the BWT. All other methods were run with default parameters.

We repeated each experiment five times, and reported the average CPU time and peak memory for the set of chromosomes 19 up to 64 distinct variants, for *Salmonella* up to 1,000 sequences, and for all SARS-CoV2. The experiments that exceeded 48 hours of wall clock time or exceeded 62 GB of memory were omitted for further consideration, e.g., 128 sequences of `chr19`, 5000 sequences of `salmonella` and 400,000 sequences of `sars-cov2` for `egap`. Furthermore, `gsufsort` failed to successfully build the eBWT for 256 sequences of `chr19`, 5000 sequences of `salmonella`, and 400,000 sequences of `sars-cov2` or more, because it exceeded the 62GB memory limit.

6.4.3 Results

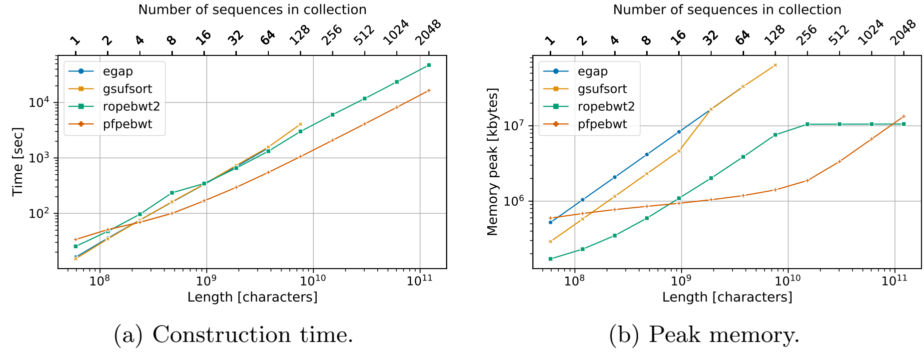
In Figures 6.1, 6.2, and 6.3 we illustrate the construction time and memory usage to build the eBWT and the BWT of collections of strings for the chromosome 19 dataset, the *Salmonella* dataset, and the SARS-CoV2 dataset, respectively.

`pfpebwt` was the fastest method to build the eBWT of 4 or more sequences of chromosome 19, with a maximum speedup of 7.6x of wall-clock time and 2.9x of CPU time over `ropebwt2` on 256 sequences of chromosomes 19, 2.7x of CPU time over `egap` on 64 sequences, and 3.8x of CPU time over `gsufsort` on 128 sequences. On *Salmonella* sequences, `pfpebwt` was always the fastest method, except for 10,000 sequences where `ropebwt2` was the fastest method on wall-clock time. `pfpebwt` had a maximum speedup of 3.0x of wall-clock time over `ropebwt2` on 100 sequences of `salmonella`. Considering the CPU time, `pfpebwt` was the fastest for $\geq$ 500 sequences with a maximum speedup of 1.7x over `ropebwt2` on 100 sequences and 1.2x over `gsufsort` and `egap` on 1,000 sequences. On SARS-CoV2 sequences, `pfpebwt` was always the fastest method, with a maximum speedup of 2.4x of wall-clock time over `ropebwt2` while a maximum speedup of 1.3x of CPU time over `ropebwt2` on 400,000 sequences, 2.9x over `gsufsort` and 2.7x over `egap` on 200,000 sequences of SARS-CoV2.

Considering the peak memory, on the chromosomes 19 dataset, `ropebwt2` used the smallest amount of memory for 1, 2, 4, 8, and 2,048 sequences, while `pfpebwt` used the smallest amount of memory in all other cases. `pfpebwt` used a maximum of 5.6x less memory than `ropebwt2` on 256 sequences of chromosomes 19, 28.0x less than `egap` on 64 sequences, and 45.3x less than `gsufsort` on 128 sequences. On *Salmonella* sequences, `pfpebwt` used more memory than `ropebwt2` for 50, 100, and 10,000 sequences, while `pfpebwt` used the smallest amount of memory on all other cases. The largest gap between `ropebwt2` and `pfpebwt` memory peak is of 1.7x on 50 sequences. On the other hand, `pfpebwt` used a maximum of 17.0x less memory than `egap` and `gsufsort` on 1,000 sequences. On SARS-CoV2 sequences, `pfpebwt` always used the smallest amount of memory, with a maximum of 6.4x less memory than `ropebwt2` on 25,000 sequences of SARS-CoV2, 57.1x over `gsufsort` and `egap` on 200,000 sequences.

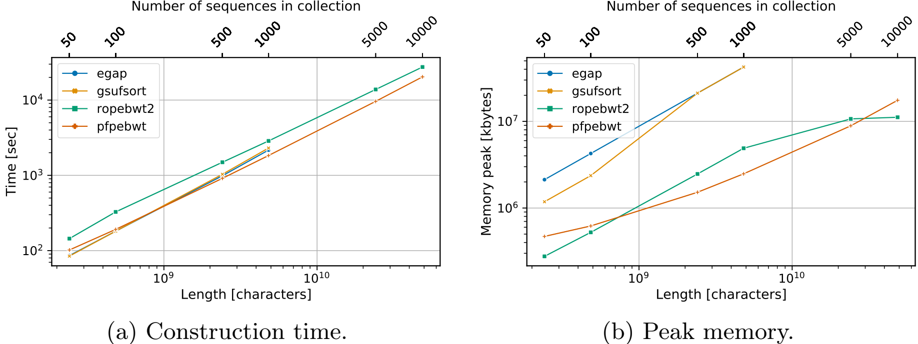
The memory peak of `ropebwt2` is given by the default buffer size of 10 GB, and the size of the run-length encoded BWT stored in the rope data structure. This explains the memory plateau on 10.5 GB of `ropebwt2` on the chromosomes 19 dataset. However, `ropebwt2` is able only to produce the BWT of the input

sequence collection, while `pfpebwt` can be trivially extended to produce also the samples of the conjugate array at the run boundaries with negligible additional costs in terms of time and peak memory.



(a) Construction time. (b) Peak memory.

Fig. 6.1: Chromosome 19 dataset construction CPU time and peak memory usage. We compare `pfpebwt` with `ropebwt2`, `gsufsort`, and `egap`.

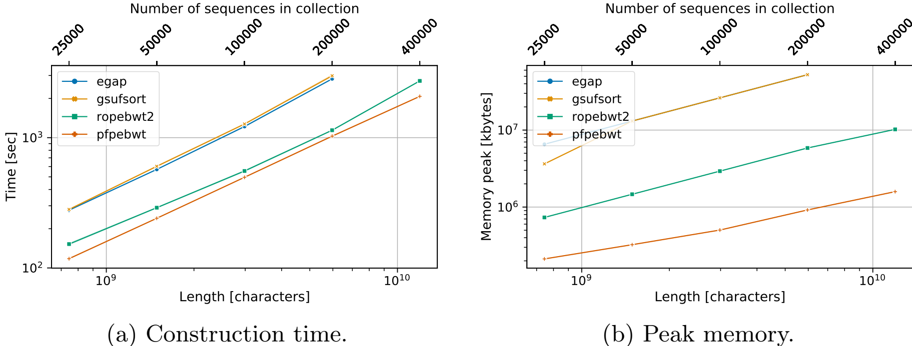


(a) Construction time. (b) Peak memory.

Fig. 6.2: *Salmonella* dataset construction CPU time and peak memory usage. We compare `pfpebwt` with `ropebwt2`, `gsufsort`, and `egap`.

6.5 Conclusion

We described the first linear-time algorithm for building the eBWT of a large collection of genomic sequences that does not require the manipulation of the input sequences, i.e., neither the addition of an end-of-string character, nor computing and sorting the Lyndon rotations of the input strings. We reached this result by combining the `SAIS_for_eBWT` algorithm with an extension of the prefix-free parsing to cyclic strings. We demonstrated `pfpebwt` was efficient with respect to both memory and time in a scenario where the input is highly repetitive.



(a) Construction time. (b) Peak memory.

Fig. 6.3: SARS-CoV2 dataset construction CPU time and peak memory usage. We compare `pfpebwt` with `ropebwt2`, `gsufsort`, and `egap`.

6.6 RePairing the PFP

The original PFP algorithm relies on two parameters, the length of the trigger strings w and the integer p, to tune the size of the PFP data structures of $T[1..n]$. In particular, it computes the trigger strings by selecting all substring $T[i..i + w]$ whose Karp-Rabin hash is congruent 0 modulo p [21]. Each PFP phrase is then defined as a substring of the text starting and ending with a trigger string. Since the parse contains an integer for each phrase, if we set low w and p values, the PFP algorithm will generate many short phrases associated with a small dictionary and a large parse. On the other hand, if we set high w and p values, we will have many long unique phrases associated with a large dictionary and a small parse. Due to this, we want to select w and p to minimize the combined size of the parse and dictionary.

In [103], we collaborated on developing an algorithm to optimize the combined size of the PFP dictionary and parse. This algorithm is inspired by Larsson and Moffat's RePair [71], which iteratively substitutes the most frequent character pairs in the text with a new character. However, in our algorithm, we substitute pairs of phrases rather than pairs of characters. The selection of the best pairs is driven by a cost function that assigns a value to each trigger string. Thus, at each step, we remove the pairs associated with the trigger string having the highest cost. The method depicted above is implemented in the `RePFP-CST` tool and provides significant compression of the PFP data structures while enabling the construction of the *compressed suffix tree* (CST) [20] index of very large string collections; in particular, it allowed to compute the CST of one terabyte of data. Here is a summary of the main highlights of the algorithm.

Filtering the trigger strings In our method, we merge pairs of phrases in the parse by removing trigger strings. In particular, we search for the trigger strings that allow decreasing the combined size of parse and dictionary the most and filter them out. This method is implemented by defining a cost function to track the effect of removing a specific trigger string.

Let $\mathcal{D} = \{d_1, ..., d_{|D|}\}$ be the dictionary and $\mathcal{P} = [p_1, ..., p_{|P|}]$ be the parse of a text T. We aim to compute a new pair D' and P' such that

$||D'|| + W|P'| < ||D|| + W|P|$ where W is the size of one element of the parse. Since each pair of consecutive phrases (p_j, p_{j+1}) in P overlaps by a trigger string of length w we can define for each trigger string T_i, the set L_i containing all pairs of phrases (p_j, p_{j+1}) such that p_j ends and p_{j+1} starts with T_i. We can also define two sets L_{1_i} and L_{2_i} containing the first and second element of the pairs in L_i, respectively. Finally, we refer to $f(q,p)$ as the number of occurrences of the pair (q,p) in P. Using the set L_i, we can compute the effect of removing T_i on the combined size of the dictionary and parse. We define as $C_D(T_i) = \sum_{p \in L_{1_i} \cup L_{2_i}} |p| - \sum_{(p_1,p_2) \in L_i} (|p_1| + |p_2| - w)$ the reduction of the dictionary size removing the trigger string T_i. While $C_P(T_i) = \sum_{(p_1,p_2) \in L_i} f(p_1,p_2) \cdot W$ is the reduction of the parse size. Altogether, the effect of removing one trigger string T_i is $C(T_i) = C_D(T_i) + C_P(T_i)$.

Updating the trigger string costs Our algorithm uses a greedy procedure; at each step, it selects the trigger string T_i with the highest cost $C(T_i)$ and replaces each pair of phrases $(p,q) \in L_i$ with a new symbol u. Then, it updates the dictionary by inserting the phrases associated with a new symbol u and removing the phrases that no longer appear in the parse. Moreover, it updates the cost of the other trigger strings. In order to compute all these operations efficiently, we need three additional data structures: (i) a double-linked list DL for the parse, (ii) a priority queue PQ to store the trigger string costs, (iii) a hash table H storing for each T_i all pointers to the occurrences of the L_i pairs in P. We remove a trigger string T_i and update the cost of all others in $O(occ \log |T| + occ \log occ)$, where $|T|$ is the number of trigger strings, and occ is the number of occurrences of T_i in P.

We can summarize this procedure in four steps: 1) select the trigger string with the highest cost $T_{i_{max}}$ using the priority queue PQ, 2) substitute all pairs $(p,q) \in L_{i_{max}}$ in P with a new symbol u, all positions of the pairs to substitute are contained in the $H[T_{i_{max}}]$ entry. 3) update the dictionary, and 4) update the cost of all remaining trigger strings in PQ. We compute this last step efficiently by storing all positions where we inserted a new symbol u in the parse, and then, we update the costs by scanning only these positions.

Experimental results The performance of our method was evaluated on two datasets, the repetitive corpus from Pizza&Chilli [107] and a dataset containing 5,000 sequences of chromosomes 17,18, and 19. As for the space reduction, the best results were observed on the Pizza&Chilli Einstein and chromosomes datasets. Here, our algorithm compressed the combined size of PFP data structures of the two datasets by 57% and 40%, respectively. As for constructing the CST data structure, on chromosomes dataset, `rePFP-CST` was able to reduce the CPU-time by 37.17% and the memory peak by 27.65% compared to `PFP-CST` tool [20], the only competitor that can construct the CST. Finally, `rePFP-CST` was able to construct, for the first time, the CST of a dataset of 1 terabyte containing 5,000 chromosome sequences.

7

Constructing the extended r-index

In this chapter, we present the extended r-index, a novel extension of the r-index to the eBWT. A preliminary version of the contents of this chapter was published in [19], while an extended version is under review for publication.

Even if string collections resulting from sequencing projects, such as the 100K Human Genome Project [125], the 1001 Arabidopsis Project [123], and the 3,000 Rice Genomes Project (3K RGP) [121], are often very large, usually only a small fraction of all information is useful to identify genomic variations. This is because, between cultivars or individuals of these sequencing projects, genomes share a lot of identical regions, leading to many repetitions in the datasets. In this context, the challenge of text indexing consists in developing data structures able to exploit the biological data repetitiveness to get a compressed representation of the input, which can still be queried efficiently.

The FM-index [41] is a data structure that has been the cornerstone of text indexing for two decades, as it was part of two of the most well-known read alignment algorithm: BWA [74] and Bowtie [70]. However, this data structure does not scale well when using pangenome datasets as a reference for string alignment. Since these datasets exhibit a small number of runs r of the resulting BWT, the research moved toward the development of an index able to answer the same queries of the FM-index while showing a space requirement linear in the number of runs of the BWT, rather than in the size of the input.

In 2005 Mäkinen and Navarro [83, 84] introduced the *run-length FM-index* (RLFM), a text index built on the run-length compressed BWT (RL BWT) which solves locate queries to find all occurrences of a pattern $P[1..p]$ in a text $T[1..n]$ in $O\big((p+s\cdot occ)\big(\frac{\log\sigma}{\log\log r}+(\log\log n)^2\big)\big)$ time, where occ is the number of occurrences of P, σ is the alphabet size, and $s \geq 1$ is a parameter. The downside of this data structure is that it requires $O(r+n/s)$ space to store the RL BWT and the SA-samples. Finally, in 2018 Gagie et al. [44, 45] introduced the *r-index* data structure; the first text index which fully supports locate queries in space $O(r)$ linear in the number of runs of the BWT.

However, during the development of all these data structures, little attention has been paid to the fact that the input nowadays is typically a string collection rather than an individual sequence. In particular, as we detailed in Chapter 3, there are different ways to concatenate the input sequences, which generate non-equivalent results. Due to this, treating a string collection such as a single

sequence can result in a big variation in the memory requirement of the resulting data structures, which is measured using r. Since the eBWT introduced by Mantaci et al. [87] is independent of the input order, it offers a solution to the previously mentioned problem. This is because the number of runs of the resulting transform does not change while modifying the input order. However, until recently, no efficient eBWT algorithm implementation was available. We closed this gap in Chapters 5 and 6, where we described and implemented two algorithms for constructing the eBWT, unlocking the efficient computation of this BWT variant even for large string collections.

In this chapter, we extend the eBWT construction algorithm described in Chapter 6 for computing the analog of the r-index based on the eBWT. We refer to this new data structure as *extended r-index*. We show how to adapt the PFP algorithm to the new task, thus enabling us to handle very large string collections. The extended r-index has similar time and space requirements to the r-index of Gagie et al. while inheriting the properties of the eBWT. In contrast to other BWT variants, which append terminator characters at the end of the strings, the eBWT naturally supports circular pattern matching queries. In particular, the eBWT considers the input strings to be indexed as circular; thus, it allows searching for patterns spanning the end and beginning of a string.

We implemented the extended r-index and evaluated it on real-life biological data. In particular, we constructed the extended r-index of two collections of circular *Salmonella Enterica* and *Escherichia Coli* genomes, and a set of plasmids from various microbial species, and tested them with patterns of varying length. We found that the extended r-index always has a negligible memory overhead and similar query time compared to the r-index. We also compared the number of occurrences as reported by the extended r-index with the r-index and show that the differences between the number of occurrences reported by the two indexes can be significant, especially when considering long patterns: for patterns of length $10,000$, the r-index reports, on average, 50% fewer matches than the extended r-index.

7.1 Overview

The rest of the chapter is organized as follows. In Section 7.2, we present the extended r-index data structure, and in Section 7.3 detail its construction using the cyclic PFP algorithm. In Section 7.4, we explain how to construct the thresholds along with the extended r-index to answer MEM queries, and in Section 7.5, we present our experimental results. We close with an outlook in Section 7.6.

7.2 The extended r-index

In this section, we show how to extend the r-index to the eBWT. Throughout the rest of the chapter, we assume that all strings in $\mathcal{M}$ are primitive, and that the multiset $\mathcal{M}$ is *conjugate-free*, i.e. no two strings are conjugates. The first

assumption is justified, since we showed in Chapter 5 how, given a multiset of strings $\mathcal{M} = \{T_1, \ldots, T_m\}$, the $\text{GCA}_\mathcal{M}$ can be computed from the GCA of the roots of the strings in $\mathcal{M}$ or via the modification of the SAIS algorithm presented in Section 5.4.1. In addition, as we detail in Section 7.5.3, the ϕ-mapping can be adapted to work with non-primitive strings by storing some additional information, such as the exponents of the strings in $\mathcal{M}$. The assumption that $\mathcal{M}$ does not contain two strings which are conjugate is more restrictive. In Section 7.5, we detail how we deal with input collections for which this condition does not hold. Note in particular that the fact that $\mathcal{M}$ is conjugate-free implies that $\mathcal{M}$ is a set. Finally, given a pattern $P[1..p]$, we assume the pattern length to be $p \leq min\{|s| : s \in \mathcal{M}\}$. This assumption ensures the correctness of our pattern matching algorithm on the eBWT, and is realistic in most practical scenarios. We recall that from now on, we denote the total length of strings in $\mathcal{M}$ by $N = ||\mathcal{M}||$.

Note that the results we present in this chapter hold in the RAM model of computation, i.e., we assume our software runs on a random-access memory which supports constant time access and processing of words of $w = \Omega(\log N)$ bits. It implies that all arithmetic and logical operations on such a machine word are performed in constant time.

7.2.1 The data structures for the extended r-index

The *extended r-index* consists of three main components: (1) a data structure supporting backward search queries on the run-length encoded eBWT, (2) a data structure computing the toehold value during the backward search, and (3) a data structure computing $\phi_\mathcal{M}$ operations on the GCA.

We construct (1) by extending the RLFM-index by Mäkinen and Navarro [83] to the eBWT, and describe it using the definitions and notation in [45]. It consists of four elements:

(i) a data structure $E[1..r]$ which supports predecessor search queries and stores the first position of each eBWT run (*run heads*). The ith entry $E[i]$ is the beginning of the ith run;
(ii) a data structure $L[1..r]$ which supports rank and select queries and stores the eBWT run characters. The ith entry $L[i]$ is the eBWT character of the ith run;
(iii) a data structure $D[1..r]$ which stores the cumulative lengths of the eBWT runs of the same symbol after stably sorting them according to the lexicographic order of their characters;
(iv) two vectors $C[1..\sigma]$ and $C'[1..\sigma]$: entry $C[c]$ contains the number of characters smaller than c in the eBWT, and $C'[c]$ contains the number of characters smaller than c in L.

In Example 20, we show the content of such data structures when the set $\mathcal{M} = \{\texttt{AAT}, \texttt{AATAT}, \texttt{GATAATAA}, \texttt{AGA}\}$ is considered.

We implement the backward search procedure to search a pattern $P[1..p]$ in the eBWT using these four data structures. In particular, we compute the interval $\text{GCA}[i..j]$ for P by using $2p$ rank queries on the run-length encoded eBWT. Each rank query counts the occurrences of a character c in an eBWT

prefix eBWT[1..i] and can be computed efficiently using the E, L, D and C' data structures using the procedure shown in [45, Section 2.5] (see Example 20). Then, once the results of the rank queries have been computed, the C vector is used to get the correct eBWT interval. All these data structures can be stored in $O(r)$ words with up-to-date implementations (see Section 7.5.1). Note that also the select queries are computed using the same four data structures. We recall that given an array S of n elements, $rank_c(S, i)$ returns the number of occurrences of c in the prefix $S[1, i]$; while $select_c(S, i)$ returns the position of ith occurrence of c in S. Moreover, given an ordered array S' and a symbol c', where S' contains elements from a totally ordered set U and $c' \in U$, the predecessor query $pred(S', c')$ returns the position of the largest element in S' smaller than or equal to c'. In Example 20, we show how these operations work on the previously defined data structures and how to apply them for the computation described above.

$\mathcal{M} = \{\texttt{AAT}, \texttt{AATAT}, \texttt{GATAATAA}, \texttt{AGA}\}$

i	GCA$_{\mathcal{M}}$	LF$_{\mathcal{M}}$	eBWT$_{\mathcal{M}}$	
1	(4,3)	13	G	AAG
2	(3,7)	15	T	AAGATAAT
3	(3,4)	16	T	AATAAGAT
4	(1,1)	17	T	AAT
5	(2,1)	18	T	AATAT
6	(4,1)	1	A	AGA
7	(3,8)	2	A	AGATAATA
8	(3,5)	3	A	ATAAGATA
9	(3,2)	14	G	ATAATAAG
10	(1,2)	4	A	ATA
11	(2,4)	19	T	ATAAT
12	(2,2)	5	A	ATATA
13	(4,2)	6	A	GAA
14	(3,1)	7	A	GATAATAA
15	(3,6)	8	A	TAAGATAA
16	(3,3)	9	A	TAATAAGA
17	(1,3)	10	A	TAA
18	(2,5)	11	A	TAATA
19	(2,3)	12	A	TATAA

Fig. 7.1: An illustration of the eBWT for the multiset of strings $\mathcal{M}$. From left to right, we report the index i, the generalized conjugate array GCA for $\mathcal{M}$, the LF permutation, the eBWT, and the conjugates of $\mathcal{M}$ sorted according to the ω-order. Highlighted in red the GCA-samples at the beginning and at the end of an eBWT run.

Example 20. In Figure 7.1 we have the eBWT and the GCA of a string collection $\mathcal{M}$. Following the above list, (i) $E = [1, 2, 6, 9, 10, 11, 12]$ and answers predecessor queries like $pred(E, 7) = 3$, where $E[3] = 6$ is the predecessor of 7 in E; (ii) $L = [\texttt{G}, \texttt{T}, \texttt{A}, \texttt{G}, \texttt{A}, \texttt{T}, \texttt{A}]$ and answers rank and select queries like $rank_{\texttt{A}}(L, 5) = 2$,

and $select_{\mathtt{A}}(L, 2) = 5$; (iii) $D = [3, 4, 12, 1, 2, 4, 5]$ corresponding to the sorted runs $(\mathtt{A}, \mathtt{3})(\mathtt{A}, \mathtt{1})(\mathtt{A}, \mathtt{8})(\mathtt{G}, \mathtt{1})(\mathtt{G}, \mathtt{1})(\mathtt{T}, \mathtt{4})(\mathtt{T}, \mathtt{1})$; (iv) $C = [0, 12, 14]$, and $C' = [0, 3, 5]$ for the $\mathtt{A}, \mathtt{G}, \mathtt{T}$ characters.

We compute the rank query $rank_{\mathtt{A}}(\text{eBWT}, 13)$ as follows. The predecessor of 13 in E is 12 in position 7. We know that eBWT[13] is in a run of A since $L[7] = \mathtt{A}$. With $rank_{\mathtt{A}}(L, 7) = 3$ we detect that eBWT[13] is within the third run of A. We now access the $D[C'[A] + (3 - 1)] = 4$ value in D, and compute the final result $rank_{\mathtt{A}}(\text{eBWT}, 13) = 4 + (13 - 12) + 1 = 6$.

We construct (2) by augmenting the backward search procedure to find a toehold [45, Lemma 3.2] in the GCA, rather than in the SA, and (3) by extending the locate machinery in [45, Section 3] to compute ϕ operations on the GCA rather than on the SA. We do this by including the following data structures:

(v) a data structure $G[1..r]$ which stores the GCA-samples at the end of the eBWT runs. The ith entry $G[i]$ is the GCA-sample at the end of the ith run;

(vi) a data structure $F[1..r]$ which supports circular predecessor search queries and stores the GCA-samples at the beginning of the eBWT runs in text order. The ith entry $F[i]$ is the ith GCA-sample at the beginning of an eBWT run in text order;

(vii) a data structure $FirstToRun[1..r]$ which stores the mapping between the GCA-samples in F and G. The ith entry $FirstToRun[i] = i'$ indicates that $F[i]$ and $G[i']$ are the GCA-samples at the beginning and end of the i'th eBWT run, respectively;

(viii) a data structure $B[1..m]$ which stores the length of each input string. The ith entry $B[i]$ is the length of the ith string in $\mathcal{M}$.

The content of such data structures for the input string collection $\mathcal{M} = \{\mathtt{AAT}, \mathtt{AATAT}, \mathtt{GATAATAA}, \mathtt{AGA}\}$ is described in Example 21. Notice that since we are working with collections of circular strings, the predecessor search queries on F have to work circularly. Given a GCA-sample, $s = (d, j)$, the circular predecessor query $pred_{circ}(F, s)$ returns the position of the GCA-sample (d, j') in F such that j' is the largest positive integer smaller or equal than j. If there is no such a GCA-sample in F, we ensure the circular behavior by searching the predecessor of $(d, B[d])$, i.e., the sample (d, j') such that j' is the largest integer in $\{j + 1, .., B[d]\}$.

Example 21. Continuing Example 20, following the above list, (v) stores the GCA-samples at the end of the eBWT runs $G = [(4, 3), (2, 1), (3, 5), (3, 2), (1, 2), (2, 4), (2, 3)]$, (vi) stores the GCA-samples at the beginning of the eBWT runs sorted in text order $F = [(1, 2), (2, 2), (2, 4), (3, 7), (3, 2), (4, 1), (4, 3)]$. The (vii) stores the mapping between G and F, $FirstToRun = [5, 7, 6, 2, 4, 3, 1]$. Note that since $FirstToRun[4] = 2$, $F[4] = (3, 7)$ and $G[2] = (2, 1)$ are the beginning and end GCA-samples of the second eBWT run. Finally, (viii) stores the input string lengths $B = [3, 5, 8, 3]$.

We now describe how to compute the toehold value for the GCA during the backward search procedure. The first step consists in augmenting the backward search in a way that, at each step, in addition to the interval GCA$[i..j]$, it also

returns GCA[j]. We always store as toehold the last value of the interval since later we will use ϕ to compute all other values in GCA[$i..j-1$]. Assume we are searching for a pattern $P[1..p]$ and we matched it up to position p'. The interval corresponding to $P[p'..p]$ is GCA[$i..j$]; the toehold for this interval is GCA[j]. We update the toehold for the next backward search step as follows. Let GCA[$i'..j'$] be the interval for $P[p'-1..p]$ and $c = P[p'-1]$; we need to distinguish between two cases. If eBWT[j] $\neq c$, then eBWT[j'] is the last position of a run. Thus, we compute the new toehold value by using a predecessor search query on E and a rank and select query on the eBWT to compute the position of the eBWT run containing the eBWT[j'] character. Finally, we use $FirstToRun$ to get the the GCA[j'] value stored in G. Otherwise, if eBWT[j] $= c$, we simply update the current toehold value (d, x) to $(d, x-1)$. Notice that also the update of the toehold value has to work circularly. If $x = 1$, we compute the updated toehold as $(d, x-1) = (d, B[d])$. We keep updating the toehold with this procedure until we obtain the final GCA-interval for $P[1..p]$.

Example 22. Continuing Example 20, assume we are locating pattern $P =$ `AAG`. The interval corresponding to the empty string is GCA[1..19] and the toehold value is $(2, 3)$. Using the LF-mapping, we compute the new GCA-interval for $P[3..3] =$ `G`, GCA[13..14]. Since eBWT[19] $\neq$ `G`, we compute the new toehold as follows, $rank_G(\text{eBWT}, 14) = 2$; thus, $select_G(\text{eBWT}, 2) = 9$ i.e. the second occurrence of G is in position 9. Finally, we compute $pred(E, 9) = 4$, and update the toehold value $G[4] - 1 = (3, 1)$. Next, we compute the GCA-interval for $P[2..3] =$ `AG`, GCA[6..7]. In this case, eBWT[14] $=$ `A`; thus, we simply compute the circular predecessor of $(3, 1)$ in T_3, i.e., $t = (3, 8)$. We end this example computing the interval GCA[1..2] for $P[1..3]$ with toehold $(3, 7)$.

The last element we need to implement in our locate machinery is a data structure which can compute ϕ operations on the GCA. Given GCA[i] $= (d, j)$ we compute $\phi(\text{GCA}[i]) = \text{GCA}[i-1]$ by using a predecessor search query on F and one access to G and $FirstToRun$. First, we obtain the circular predecessor of (d, j), $pred_{circ}(F, (d, j)) = (d, j')$. Then, we compute the number of LF-steps between the two GCA-samples, $\Delta = j - j'$. Note that if $j' > j$, the distance is computed in a circular way as $\Delta = j + (B[d] - j')$. Finally, we get the final result by adding Δ to the sample in $G[FirstToRun[j] - 1] = (d', x)$, i.e., $\phi(\text{GCA}[i]) = (d', (x + \Delta) \bmod B[d'])$.

Example 23. Continuing Example 20, let GCA[3..5] be the interval of $P =$ `AAT` with toehold GCA[5] $= (2, 1)$. We get $\phi(\text{GCA}[5])$ as follows; we compute a circular predecessor search query $pred_{circ}(F, (2, 1)) = 3$; where $F[3] = (2, 4)$. We compute the number of LF steps separating the two GCA samples, $\Delta = (2, 1) - (2, 4) = 1 + (B[2] - 4) = 2$. Now we get the GCA-sample at the end of the previous run $G[FirstToRun[3] - 1] = (1, 2)$, and compute the final result $\text{GCA}[3] = (1, 2) + \Delta = (1, (2 + 2) \bmod B[1]) = (1, 1)$.

We store G, F, B, and $FirstToRun$ in $O(r)$ words (Prop. 5), and give details of the implementation in Section 7.5.1.

7.2.2 Analysis of the extended r-index

We will now show that the memory requirement of our data structure is $O(r)$ words, and that it can answer count and locate queries in time analogous to the original r-index. Recall also that all strings in $\mathcal{M}$ are assumed to be primitive and that $\mathcal{M}$ is a conjugate-free set.

We first need a technical lemma:

Lemma 10. *Let $\mathcal{M} = \{T_1, \ldots, T_m\}$ be a conjugate-free set of primitive strings of total length N,* GCA *its generalized conjugate array, with* $\text{GCA}[h] = (d_h, j_h)$ *for $1 \leq h \leq N$. Let k be such that* $\text{eBWT}[k] = \text{eBWT}[k-1]$*, and let* $k' = \text{LF}(k)$*. Then* $\text{LF}(k-1) = k' - 1$*. In particular,* $\text{GCA}[k'-1] = (d_{k-1}, j_{k-1} - 1)$*.*

Proof. This follows directly from the fact that the eBWT has the LF-property, i.e., that same characters appear in the same order in the last and the first column of the eBWT-matrix, see Prop. 10 in [87].

Example 24. Figure 7.1 illustrates an example of the property in Proposition 10. Considering rows 10 and 11, we have $\text{GCA}[12] = (2, 2)$ and $\text{GCA}[13] = (4, 2)$. With $\text{LF}[12] = 5$ and $\text{LF}[13] = 6$, we have $\text{GCA}[5] = (2, 1)$ and $\text{GCA}[6] = (4, 1)$.

To analyze the space required by the extended r-index, we show that every string in $\mathcal{M}$ is sampled at least once among the run-beginning samples and at least once among the run-end samples.

Proposition 4. *Let $\mathcal{M} = \{T_1, \ldots, T_m\}$ be a conjugate-free set of primitive strings of total length N,* GCA *its generalized conjugate array, with* $\text{GCA}[h] = (d_h, j_h)$ *for $h = 1, \ldots, N$. Then, for each i, $1 \leq i \leq m$, there exist integers k and k', such that $d_k = d_{k'} = i$ and*

1. *either $k = 1$ or* $\text{eBWT}[k] \neq \text{eBWT}[k-1]$*, and*
2. *either $k' = N$ or* $\text{eBWT}[k'] \neq \text{eBWT}[k'+1]$*.*

Proof. We assume (for contradiction) that there exists an integer i, with $1 \leq i \leq m$, such that at least one of the two following conditions holds:

1. $d_1 \neq i$ and, for all $1 < h \leq N$ such that $d_h = i$, $\text{eBWT}_{\mathcal{M}}[h] = \text{eBWT}_{\mathcal{M}}[h-1]$;
2. $d_N \neq i$ and, for all $1 \leq h < N$ such that $d_h = i$, $\text{eBWT}_{\mathcal{M}}[h] = \text{eBWT}_{\mathcal{M}}[h+1]$.

We assume w.l.o.g. that 1 holds. Then, by Lemma 10 there exists an integer $1 \leq i' \leq m$, with $i' \neq i$, such that for all $1 < h \leq N$ such that $d_h = i$ it holds that $d_{h-1} = i'$. If we let $u = \text{conj}_{j_{h-1}}(T_{d_{h-1}})$ and $v = \text{conj}_{j_h}(T_{d_h})$ then we see that $\text{root}(u) = \text{root}(v)$ by Lemma 10. In fact, every time we make an LF-step, the conjugate index in $\text{GCA}_{\mathcal{M}}$ is decreased by 1, and after $|\text{root}(v)|$ steps, the index of the conjugate of u must be j_{h-1}. This implies that $\text{root}(u)$ and $\text{root}(v)$ are conjugate. Since all strings in $\mathcal{M}$ are primitive, then we conclude that u and v are conjugate, which is a contradiction to the hypothesis that $\mathcal{M}$ is a conjugate-free set.

Proposition 4 allows us to deduce a relationship between the number of runs r and the number of strings m, and, if all strings have the same length, also with the average run-length:

Corollary 2. *Let $\mathcal{M} = \{T_1, \ldots, T_m\}$ be a conjugate-free set of primitive strings of total length N, r the number of runs of its* eBWT*. Then $m \leq r$. Moreover, if all strings T_i have the same length ℓ, then $N/r \leq \ell$.*

Proof. Both statements follow directly from Proposition 4.

We are ready to state the memory requirement of our data structure:

Proposition 5. *The extended r-index of a conjugate-free set of primitive strings requires $O(r)$ words of storage, where r is the number of runs of the* eBWT *of $\mathcal{M}$.*

Proof. The data structures E, L, D, G, F, and $FirstToRun$, as presented in Sec. 7.2.1, occupy $O(r)$ words each, C and C' occupy $O(\sigma)$ words, and B occupies $O(m)$ words; since $m \leq r$ by Cor. 2, altogether we have $O(r)$ space.

Count queries are done analogously to the r-index:

Proposition 6. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N, and r the number of runs of* eBWT($\mathcal{M}$)*. We can build an index of $O(r)$ words such that, later, given a pattern $P[1..p]$, we can return the number of cyclic occurrences of P in $\mathcal{M}$ in $O(p \log\log_w(\sigma + N/r))$ time.*

Proof. Count queries are answered by applying p backward search steps, i.e., $2p$ rank queries, starting from the complete interval $[1, N]$ and computing the interval $[s_P, e_P]$ which contains the conjugates of which the pattern P is a prefix. The number of occurrences is then $occ_P = e_P - s_P + 1$. Since our strategy and data structures are exactly analogous to the ones used for the r-index, the time complexity follows by using similar arguments as in [45, Lemma 2.1].

Next we give the analog of the Toehold Lemma [45, 108]:

Proposition 7. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N and r the number of runs of* eBWT($\mathcal{M}$)*. We can build an index of $O(r)$ words such that, later, given a pattern $P[1..p]$, we can return the interval $[s_P, e_P]$ of cyclic occurrences of P in $\mathcal{M}$, along with one position q and the content* GCA[q]*, in $O(p \log\log_w(\sigma + N/r))$ time.*

Proof. As described in Sec. 7.2.1, we compute the toehold value similarly to the r-index, employing the additional data structures for the extended r-index, which allow us to handle circular predecessor queries and navigation between different strings in the string collection. The running time follows again analogously to [45, Lemma 3.2].

The next lemma, the analog of Lemma 3.5 in [45], states that one ϕ-value can be computed in $O(\log\log_w(N/r))$ time. Together with Prop. 7, this will allow us, analogously to the r-index, to output all *occ* occurrences of a pattern P.

Lemma 11. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N and r the number of runs of* eBWT($\mathcal{M}$)*. Using the extended r-index of $\mathcal{M}$, we can evaluate $\phi_{\mathcal{M}}$ in $O(\log\log_w(N/r))$ time.*

Proof. As described in Sec. 7.2.1, the evaluation of $\phi_{\mathcal{M}}$ is realized analogously to the r-index. Our additional data structures that allow the predecessor data structure E to work circularly, do not affect the $O(\log\log_w(N/r))$ query time proved in [9].

We summarize the properties of the extended r-index in the following theorem.

Theorem 2. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N, and let r be the number of runs of* eBWT($\mathcal{M}$)*. We can build an index of $O(r)$ words such that, later, given a pattern $P[1..p]$, we can return the number of cyclic occurrences of P in $\mathcal{M}$ in $O(p\log\log_w(\sigma+N/r))$ time, and after counting, return all occ_P cyclic occurrences of P in $\mathcal{M}$ in $O(occ\log\log_w(N/r))$ time.*

Proof. The extended r-index can be stored in $O(r)$ words by Prop. 5. Given a query pattern P, in $O(p\log\log_w(\sigma + N/r))$ time the GCA-interval of its occurrences $[s_P, e_P]$ can be computed, along with one occurrence in $\mathcal{M}$ (the toehold value) by Prop. 7. By our construction, this toehold value is always the last in the interval, i.e., GCA(e_P). Then, by repeated applications of ϕ, we can compute all *occ* occurrences of P, each in time $O(\log\log_w(N/r))$ by Lemma 11.

7.3 Efficient construction of the extended r-index

In Chapters 6, we showed how to efficiently construct the eBWT of large string collections by using a variant of the PFP algorithm in a way that preserves the original definition of Mantaci et al. [87]. In this section, we augment the PFP algorithm to compute the GCA-samples as well as the run-length encoded eBWT. The PFP is a preprocessing technique originally introduced to construct the BWT of large and repetitive string collections. We now recall a brief overview of the algorithm. With one scan, the PFP divides the input in overlapping substrings of variable length, called *phrases*, which are used to construct what is referred to as the *dictionary* $\mathcal{D}$ and *parse* $\mathcal{P}$ of the input collection. Then a separate procedure constructs the BWT of the input directly from $\mathcal{D}$ and $\mathcal{P}$; thus using space proportional to the combined size of these two data structures. The key of the PFP is that if the input collection is repetitive enough, the combined size of $\mathcal{D}$ and $\mathcal{P}$ will be much smaller than the size of the original input. In 6 we presented a variant of the PFP, called *cyclic PFP*. This variant is designed to process circular strings; in particular, it constructs a dictionary that also contains phrases spanning the beginning of the input strings.

In [68] Kuhnle et al. showed how to construct the SA, and the SA-samples along with the run-length encoded BWT using the PFP data structures. This is performed by computing for each BWT entry, BWT[i], its corresponding SA value SA[i]. On the other hand, the SA-samples are computed by checking at each step whether BWT[i] $\neq$ BWT[$i-1$]; if this is the case we store both the sample at the end, SA[$i-1$], and at the beginning, SA[i] of the previous

and current BWT run respectively. The SA entries are computed by storing $||\mathcal{P}||$ additional integers, one for each phrase, representing the offsets of the last characters of the PFP phrases in the original text. Later in the algorithm, when processing each sorted suffix s of one of the phrases in $\mathcal{D}$, we use the offset of the corresponding phrase and the length of s to output the SA value together with the BWT character.

Here we extend the cyclic PFP algorithm we have previously described in Chapter 6 to compute the GCA-samples rather than the SA-samples. We do this by storing $||\mathcal{P}||$ additional offsets $o = (d_1, j_1), ..., (d_m, j_m)$ representing the last positions of the cyclic PFP phrases in $\mathcal{M}$ and implementing an algorithm similar to the one in [68] to compute the GCA-samples by using these offsets. Briefly, every time we process a suffix s in $\mathcal{D}$, we get the offset of the corresponding phrase (d, j) and use the suffix length to compute the correct GCA-sample $(d, j - |s|)$. The only exception we need to handle is the case where we process a phrase spanning the beginning of a string, i.e., $|s| \geq j$. Here, in order to ensure the correct circular GCA-sample computation, we need to maintain only m additional integers storing the length of each input string as we did with B. In particular, when processing a suffix s in $\mathcal{D}$, if s spans the beginning of a string T_i, we retrieve the string length and compute the correct circular GCA-sample for s. As a result, we compute the run-length eBWT as well as the GCA samples using the augmented cyclic PFP algorithm in time linear in the size of the input collection $\mathcal{M}$ and space linear in the combined size of $\mathcal{D}$ and $\mathcal{P}$.

Finally, given these two data structures, we construct the extended r-index using the procedure described in [45, Appendix A]. As for the RLFM-index, we compute the run heads, the cumulative lengths of the sorted runs, and the content of C and C' while scanning the run-length encoded eBWT in $O(r)$ time. Moreover, we compute the E and D predecessor search data structures in $O(r)$ time and $O(r)$ words of space using up-to-date data structures (see Section 7.5.1). As for the locate machinery, we sort the GCA samples at the beginning of the runs and store them in F in $O(r \log r)$ time and $O(r)$ words of space, where $r \log r$ is the time required to sort r integers with a comparison sort algorithm such as introsort [93]. Since we have already sorted the samples in F in text order, we compute the $FirstToRun$ in $O(r)$ time by storing the original positions of those samples in F. Note that the values in B are computed during the first pass of the cyclic PFP algorithm by maintaining a counter for the length of the input strings.

Proposition 8. *Given a conjugate-free set of primitive strings $\mathcal{M}$, we compute the run-length encoded eBWT and the* GCA*-samples in $O(||\mathcal{M}||)$ time and $O(||\mathcal{D}|| + ||\mathcal{P}||)$ words of space, where $\mathcal{D}$ and $\mathcal{P}$ are the dictionary and parse defined by the cylic* PFP*, and construct the extended r-index in $O(r \log r)$ time and $O(r)$ words of space.*

Proof. In the previous discussion, we described how to adapt the method in [68] to augment the cyclic PFP algorithm introduced in Chapter 6 to compute the GCA-samples along with the run-length eBWT. This requires, in addition to the PFP data structures, to store $||\mathcal{P}||$ additional phrase offsets and m additional integers for the string lengths. Since $\mathcal{P}$ contains at least one phrase for each input string, both are stored in $O(||\mathcal{P}||)$ words of space. In addition, we need $O(r)$

additional steps to compute and output the samples, which are implemented by means of rank and select queries on a bitvector. Thus, altogether we maintain the same $O(||\mathcal{M}||)$ time and $O(||\mathcal{D}|| + ||\mathcal{P}||)$ space complexity of the original algorithm of Chapter 6. Finally, it follows from the previous discussion that given the run-length eBWT and the GCA-samples, we construct the extended r-index in $O(r \log r)$ time and $O(r)$ space by extending the procedure described in [45, Appendix A].

7.4 Computing MEMs with the extended r-index

Maximal exact matches (MEMs) are exact matches between two sequences G and R that can neither be extended to the left nor to the right. A classic application for MEMs is finding seeds between a short sequence and a genome for computing multiple sequence alignments of both short and long reads [113]. As previously noted, the main components of the r-index, namely the run-length BWT and the SA-samples, are not enough to find MEMs efficiently.

Bannai et al. [5] showed that MEM-finding can be supported by computing the *matching statistics* (MS) of a query string P w.r.t. the text T, defined as an array of size P, with $P[i] = (\ell, q)$, where ℓ is the length of the longest substring starting at position i in P which occurs somewhere in T, and q one of its occurrences in T, i.e., some position s.t. $T[q..q + \ell - 1] = P[i..i + \ell - 1]$. From the matching statistics for P, we can compute the occurrence of a MEM using a two-pass algorithm: first, working right to left, we process each suffix of P until we find a substring of the text that matches for as long as possible; then, working left to right, we use random access to T to determine the length of those matches.

In order to compute the matching statistics, Bannai et al. described the addition of a small data structure to the r-index referred to as *thresholds*. However, they did not explain how to construct it efficiently. Rossi et al. [113] solved this problem by redefining a threshold between a consecutive pair of runs of the same character as the position of the minimum LCP value in the interval between them. They further showed how to compute the thresholds efficiently using the PFP based construction algorithm of the r-index.

The same modification can also be made for the extended r-index, allowing the thresholds for the eBWT to be constructed along with the GCA-samples. In particular, we need to extend our `pfpebwt` algorithm to find the positions of the minimum LCP values in all intervals between pairs of consecutive eBWT runs of the same character. This is performed using two data structures, 1) the LCP-array of $\mathcal{D}$, supporting range minimum queries, and 2) the LCP-array of the parse $\mathcal{P}$ counting longest common prefixes with respect to the number of characters in the original strings. In particular, every time we find a new eBWT run, we search for the minimum value in 1) or 2) depending on whether the preceding run of the same character shares the same PFP phrase suffix or not.

Corollary 3. *Given a conjugate-free set of primitive strings $\mathcal{M}$, we can build the thresholds in addition to the extended r-index in $O(||\mathcal{M}||)$ time and $O(||\mathcal{D}|| + ||\mathcal{P}||)$ words of space, where $\mathcal{D}$ and $\mathcal{P}$ are the dictionary and parse defined by the cyclic* PFP.

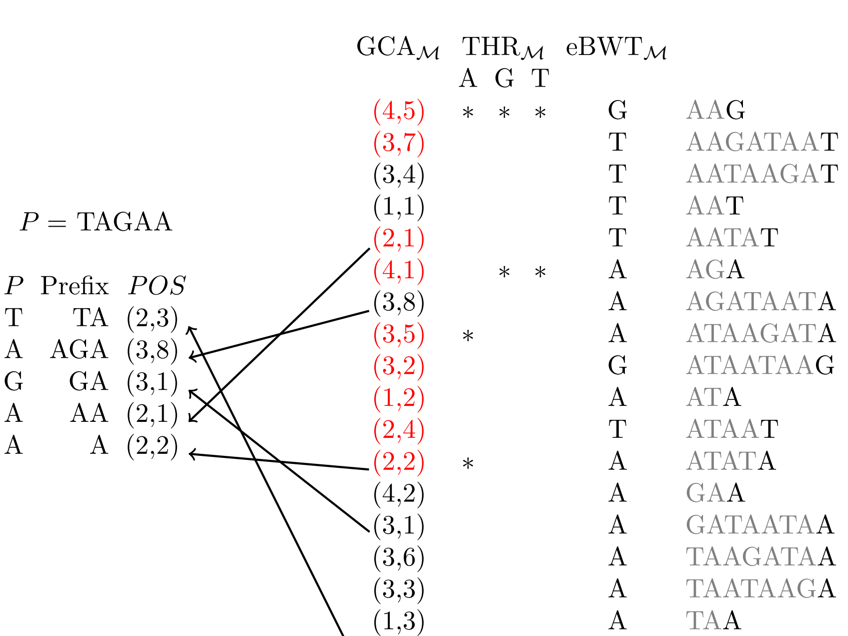


Fig. 7.2: An illustration of the thresholds for calculating the matching statistics of a query string $P[1..m]$ in a set of strings $\mathcal{M}$. Shown on the left is P, the longest *Prefix* of the suffix of P that occurs in $\mathcal{M}$, and the position of the corresponding prefix in the text. Shown on the right, continuing from left to right, is the $\mathrm{GCA}_{\mathcal{M}}$ of $\mathcal{M}$ with the entries corresponding to either the beginning or the end of a run highlighted in red, the thresholds $\mathrm{THR}_{\mathcal{M}}$ for the characters `A`, `G`, and `T`, the $\mathrm{eBWT}_{\mathcal{M}}$, and all rotations of all strings in $\mathcal{M}$. The arrows illustrate the position in the $\mathrm{GCA}_{\mathcal{M}}$ which corresponds to the prefix on the left.

These data structures can be used for finding MEMs on the eBWT in the same way as for the r-index. It follows directly from Bannai et al. [7] and Rossi et al. [113], that, given a query string $P[1..p]$ which has *occ* occurrences of a MEM in $\mathcal{M}$, we can find a single MEM in $O(p(\log\log_w(\sigma + N/r) + t_{RA}))$-time and $O(r+s_{RA})$ words of space, where t_{RA} and s_{RA} are the time and space of any data structure that is able to provide random access to the string. Moreover, we extend this search to find all *occ* occurrences in additional $O(occ \log\log_w(N/r))$-time.

Figure 7.2 depicts an example of matching statistics query of the pattern $P =$ `TAGAA` against the collection of strings $\mathcal{M} = \{$`AAT`, `AATAT`, `GATAATAA`, `AGA`$\}$.

7.5 Experimental results

We implemented the extended r-index in C++ by adapting the code of the r-index (https://github.com/nicolaprezza/r-index), and made it available in https://github.com/davidecenzato/extended_r-index.git. We compute all data structures necessary to construct the extended r-index via an adaptation of our cyclic PFP algorithm to enable our construction algorithm to scale for large string collections. We compared the performance of the extended r-index with the r-index using real-life biological data. We used the r-index implementation given in https://github.com/alshai/r-index.git since this software version allows to compute the BWT as well as the SA-samples of large datasets using the Big-BWT algorithm [68, 91].

Note that, unlike the r-index, our implementation supports circular pattern matching, i.e., we report occurrences of a pattern that span the end and beginning of a string in addition to the ones already reported by the r-index. We compute all necessary data structures via an adaptation of `pfpebwt` we described in Chapter 6 to enable our construction algorithm to scale for large string collections.

7.5.1 Implementation

We construct the RLFM-index computing the interval GCA$[i..j]$ for a pattern P by extending the implementation described in [45, Section 7.1]. We compute L by constructing the wavelet tree of the BWT runs characters using the Huffman-encoded wavelet tree implementation of `sdsl` (`wt_huff`) supporting rank and select queries. We compute E and D by encoding the run heads and the cumulative lengths using the unary encoding and storing the final binary strings in two `sdsl` Elias-Fano compressed bitvectors (`sd_vector`) supporting rank and select queries. In our implementation, we do not store C' explicitly since we split D in σ different substrings, thus enabling direct access to D. Finally, we store the C vector using the `sdsl` (`int_vector`) implementation. Altogether these data structures take $(1+\epsilon)r(\log(n/r)+2)+r\log\sigma+\sigma\log n$ bits, where $\epsilon = 0.5$ is a parameter decided in input.

We extend the locate machinery described in [45] to support ϕ operations on the GCA rather than on the SA. We construct G, the data structure storing the GCA-samples at the end of the runs, as well as the $FirstToRun$ vector using the `sdsl` (`int_vector`) implementation. Further, we compute F by storing the GCA-samples at the beginning of a run using a gap-encoded bitvector. In particular, we construct a bitvector $B_f[1..N]$ for F such that the ith entry $B_f[i] = 1$ if $i \in F$. We construct the B data structure that we additionally need in the extended r-index in a similar way; in particular, we define a bitvector $B_b[1..n]$ for B such that the ith entry $B_b[1..N] = 1$ if i is the position of the first character of one of the input string. Note that we represent the GCA-samples as we did in Section 5.7.1. In particular, we store samples as offsets in the concatenation $T_1T_2...T_m$ and keep a bitvector to get the string indexes. Thus, the indexes encoded by B_f refer to positions in the concatenation, and B_b stores the string boundaries in the concatenation. We implement both B_f

and B_b using the Elias-Fano compressed bitvector of `sdsl` (`sd_vector`) implementation. Altogether these data structures implementing the locate machinery take $r \log n + r \log r + r(\log(n/r) + 2) + m(\log(n/m) + 2)$ bits.

7.5.2 Handling equal conjugates

Proposition 4 ensures that if in an input collection $\mathcal{M}$, no two strings have the same set of conjugates, then we will sample at least two GCA-samples for each string, one at the beginning and one at the end of a eBWT run. However, when working with string collections containing circular genomes of the same specie, this condition may not be fulfilled since we may find two sequences such that one is a rotation of the other. In this case, we will have some sequences with no GCA-samples. We solve this problem by sampling some additional GCA values corresponding to the first rotation of each string in $\mathcal{M}$. We added this functionality to the algorithm described in Section 7.3: while computing the eBWT, every time we process the first rotation of a string, we check whether the corresponding character eBWT$[i]$ is the first character of a new run. If not, we instantiate a new run and store two GCA-samples, one at the beginning of the new run GCA$[i]$, and one at the end of the previous run GCA$[i-1]$. At the end of this procedure, we will sample at most $2m$ additional GCA-samples.

Example 25. In Figure 7.2, assume we want to sample the GCA positions corresponding to the first rotation of each string in $\mathcal{M}$. In this case, we also need to store $(1,1)$, $(1,2)$ and $(1,3)$, which are the samples corresponding to the `AAT`, `AATAT` and `GATAATAA` rotation, respectively. Note that $(1,4)$ was already at the beginning of the third eBWT run; thus, we do not need to sample it again.

7.5.3 Handling non-primitive strings

As for handling non-primitive strings, we need two additional functionalities: (i) computing the GCA and eBWT of string collections containing non-primitive strings and (ii) supporting ϕ operations on such a GCA. We obtain (i) by modifying the PFP algorithm as described in Section 5.2.1; this includes running the algorithm on the roots $\{S_1, ..., S_m\}$ of the strings in $\mathcal{M}$ and using the exponents $\{k_1, ..., k_m\}$ to output the correct eBWT and GCA-samples. As for (ii), we implement it by using the information provided by the exponents; in particular, when computing $\phi(\text{GCA}[i])$, where $\text{GCA}[i] = (d, j)$, if $j > k_d$ then the query will report $(d, j - k_d)$ as a result, otherwise it applies the same procedure described in Section 7.2.1.

7.5.4 Datasets

We evaluated the extended r-index, and the r-index using three genomic datasets: the first dataset contains circular assembled genomes of *Salmonella Enterica* (`Salmonella`); the second dataset contains circular assembled genomes of *Escherichia Coli* (`E.coli`) strains; the last dataset contains a set of plasmid (`plasmids`) genomes. Features of the three datasets are summarized in Table 7.1. We downloaded the `Salmonella` and `E.coli` datasets from NCBI using

the accession ids of the reference genomes from the ZymoBIOMICS High-Molecular-Weight DNA Mock Microbial community (ZymoMC) used to evaluate the method in [1]. We downloaded the most recent assemblies of this data, and removed the ones marked as anomalous. In addition, we only retained the reference genomes and removed additional contigs. This resulted in 846 Salmonella enterica and 1,362 Escherichia coli assembled genomes. As for `plasmids`, we downloaded the sequences from the PLSDB database containing a collection of 34,513 plasmid sequences gathered from the NCBI and INSDC platforms [114]. Finally, we removed all plasmid sequences shorter than our maximum pattern length and, since we did not yet implement the functionalities described in Section 7.5.3, we also removed the non-primitive sequences. This resulted in a dataset containing 25,916 plasmid sequences

dataset	no. seq	total length	avg. length	min. length	max. length	n/r (eBWT)
`Salmonella`	846	4,121,587,394	4,871,853	4,482,093	5,700,307	70.574
`E.coli`	1,362	7,024,773,608	5,157,690	4,456,672	6,162,417	88.011
`Plasmids`	25,916	3,458,859,947	133,464	10,005	4,605,385	4.220

Table 7.1: Table summarizing the main parameters of the three datasets. From left to right, we report the dataset name, the number of sequences, the total length, the average, minimum, and maximum sequence length, and the average run-length of the eBWT (n/r).

7.5.5 Experimental setup

We performed the experiments on a server with Intel(R) Core(TM) CPU i9-11900 @ 2.50GHz with 8 cores and 64 gigabytes of RAM running Ubuntu 22.04 LTS 64-bit. The compiler was g++ version 11.3.0 using C++ 17 standard and `-Ofast -fstrict-aliasing -march=native -DNDEBUG` options. We recorded the memory usage using the maximum resident set size retrieved from `/usr/bin/time`. We computed the count and locate queries on the r-index using the `occ` and `locate_all` functions, respectively.

From each dataset, we extracted three pattern sets containing 1 million sequences each of length 100, $1,000$, and $10,000$, respectively. Patterns were extracted by randomly sampling substrings from the input strings. In particular, for each pattern, we computed a random offset on an input string and extracted a substring starting from the offset. We also included patterns spanning the end and the beginning of a string. For each set of patterns, we computed both count and locate queries on all patterns and collected the number of occurrences, the total memory usage, and the average time to process a pattern.

In our experiments, we built and queried both the extended r-index and the original r-index. Given a collection of strings $\mathcal{M} = \{T_1, T_2, ..., T_m\}$, we build the r-index of the string $S = T_1\$T_2\$\cdots\$T_m\#$, where \$ and # are characters not occurring in $\mathcal{M}$ preventing spurious occurrences to be found, e.g., occurrences of the pattern P spanning T_1T_2. For the sake of completeness, we included in our experiments a data structure that builds on top of the r-index and supports circular count, and locate queries with minimal implementation effort. We refer to this data structure as *circular r-index*, and it is built as follows. Given a

collection of strings $\mathcal{M} = \{T_1, T_2, ..., T_m\}$ we build a r-index on the collection $\mathcal{M}$ and we also build a r-index on the collection $\mathcal{M}' = \{T_1T_1, T_2T_2, ..., T_mT_m\}$. Given a pattern P, circular count queries are performed by computing a count query on the circular r-index of $\mathcal{M}'$, which returns twice the number of occurrences of P that are entirely contained in $\{T_1, T_2, ..., T_m\}$ and once the number of occurrences of P that spans the end and the beginning of each string in $\mathcal{M}$; then, we subtract from this value the result of a count query on the r-index of $\mathcal{M}$. Thus, every count query on the circular r-index consists of two count queries. Finally, the locate queries on the circular r-index are computed as a locate query on the r-index of $\mathcal{M}'$, where all occurrences located in the second copy of each string are discarded.

7.5.6 Results

In Figure 7.3, we illustrate the difference in the number of occurrences reported by the r-index and the extended r-index, for those patterns where these do not coincide. For each of `Salmonella`, `E. coli`, and `plasmids`, we ran count queries for 3 million patterns: 1 million each of length 100, $1,000$, and $10,000$.

Patterns with a different number of occurrences between the r-index and the extended r-index are patterns that occur as circular substrings spanning the beginning of a sequence in the dataset. Therefore, the number of occurrences strongly depends on two factors: the lengths of the indexed sequences and the lengths of the patterns. In the first case, having long indexed sequences and short patterns, it is less likely to extract a pattern that spans the beginning of a string; thus, most patterns have the same number of occurrences in both the r-index and the extended r-index. On `Salmonella`, and `E.coli` datasets, only 0.02%, 0.04%, 0.30% and 0.04%, 0.06%, 0.27% of the patterns show a different number of occurrences for the 100, 1000, and 10000 pattern length, respectively. On the other hand, when the dataset contains short sequences, it is more likely to extract patterns that span the beginning of a string. On the `plasmids` dataset, the 3.59%, 9.44%, and 24.77% of the patterns show a different number of occurrences between the two indexes for the three pattern lengths, respectively (see Figure 7.4). Notice that for all datasets, the longest pattern length, $10,000$, is always associated with the largest percentage of patterns with different number of occurrences between the two indexes.

The pattern length also affects the difference of the number of occurrences between the r-index and the extended r-index. Given a pattern P, we defined the percentage of matches lost as $[1 - c(P)/c'(P)] \cdot 100$, where $c(P)$ and $c'(P)$ are the number of occurrences of P in the input reported by the r-index and the extended r-index, respectively. We focus on the patterns for which $c(P) \neq c'(P)$. For patterns of length 10,000, a pattern loses on average more than 50% of its matches when using the r-index. In particular, we report 68.0%, 71.9%, and 78.4% average matches lost for the `Salmonella`, `E.coli`, and `plasmids` dataset, respectively. While for patterns of length 1,000, the average match loss is 42.5%, 32.7%, and 20.0% for the three datasets. Finally, the reported average match loss is 6.0%, 3.7%, and 5.1% on patterns of length 100. In Figure 7.5, we report the number of patterns for each of the percentages of matches lost.

Runtime analysis

In Figure 7.6, we illustrate the time and the memory usage to process the three sets of patterns containing 1 million queries of lengths 100, 1,000, and 10,000 for `Salmonella`, `E.coli`, and `plamids` datasets. In particular, we report the resident set size on the x-axis and the average time to process a pattern on the y-axis.

The runtime and memory requirements for the extended r-index are similar to the r-index for all datasets and pattern lengths. As for the time, the extended r-index was slightly faster for the locate queries on `Salmonella` and `E.coli` with pattern length $p \geq 1,000$, and `Plasmids` with $p = 1,000$ with a maximum 1.03x speedup factor. In contrast, the extended r-index was slightly slower in the other cases with a maximum slowdown of 1.11x reported in the `plasmids` dataset. The same holds for memory usage, where the extended r-index used at most 1.005x more memory than the r-index. The differences are more evident when looking at the circular r-index data structure. Note that this data structure, unlike the r-index, can locate the circular occurrences as the extended r-index. As for the count queries, the extended r-index is always faster than the circular r-index with a maximum speedup of 1.97x wall-clock time. This is due to the fact that a circular r-index has always run two different count queries. For the locate queries, the running times are comparable since the extended r-index had a maximum speedup of 1.19x wall-clock time; however, we did not implement the function to filter the occurrences of the circular r-index locate queries. The extended r-index always uses less memory than the circular r-index with a maximum of 1.04x space reduction.

In summary, we show important differences in the number of occurrences between the original r-index and the extended r-index—especially when working with datasets of short sequences and long patterns. Moreover, the space and time requirements of the extended r-index are similar to the r-index on all datasets. The extended r-index always allows performing circular pattern matching queries faster and using less memory than the circular r-index. The extended r-index is also simpler to construct than the circular r-index since we do not need to double the size of the dataset.

7.6 Conclusion

In this chapter, we described how the fundamentals of the r-index can be transferred to the context of the eBWT. We note that the eBWT has the advantage over other BWT-based data structures for string collections that it is independent of the order of the input strings. The r-index based on the eBWT, which we call *extended r-index*, inherits this important property. Yet, we note that the applicability of this data structure has not been fully explored. We implemented the extended r-index and showed that on large biological datasets, it has competitive performance compared to the r-index. We further notice that from a more theoretical point of view, recasting some of the more recent results—including the results of Nishimoto and Tabei [97], Bannai et al. [5], and Cobas et al. [31]—regarding the r-index to the context of eBWT merits attention since it could provide a further improvement of our text index.

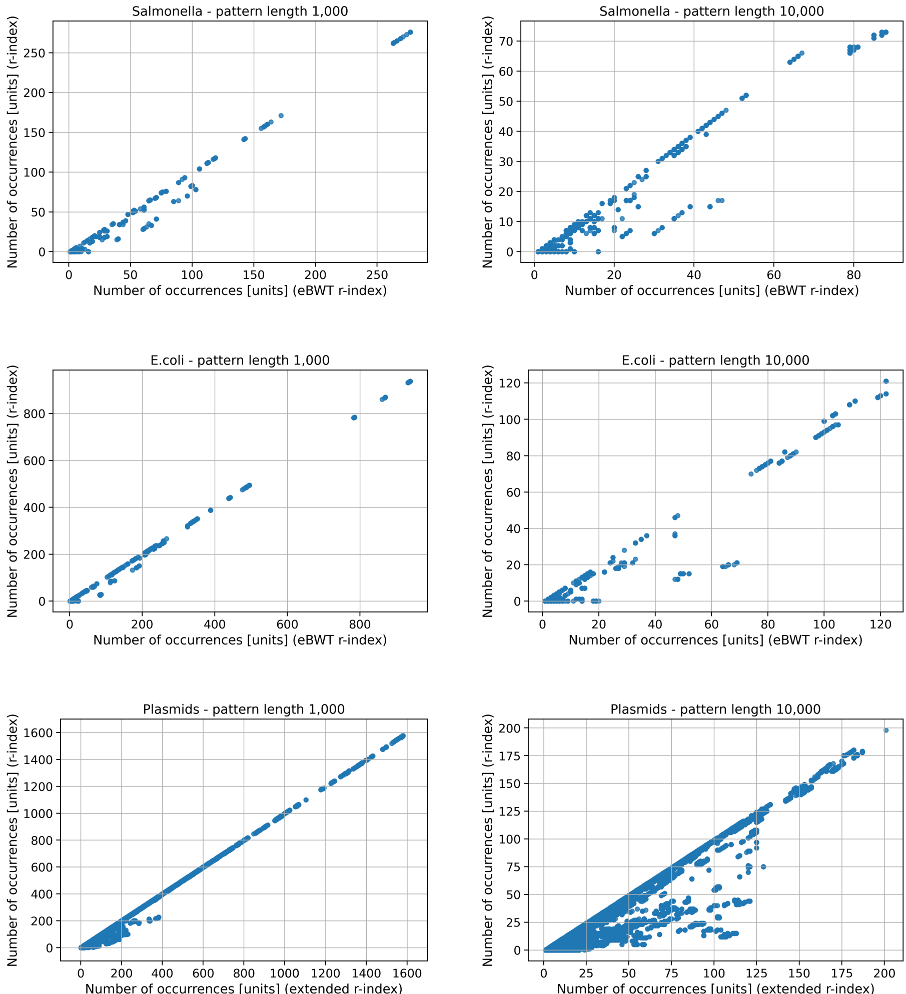


Fig. 7.3: Dispersion plots between the number of occurrences of the r-index and the extended r-index on 3 sets of patterns of different lengths, containing 1 million sequences each. Each point in the figures represents the number of occurrences of the two indexes on a specific pattern. We omitted the patterns having the same number of occurrences for the two indexes.

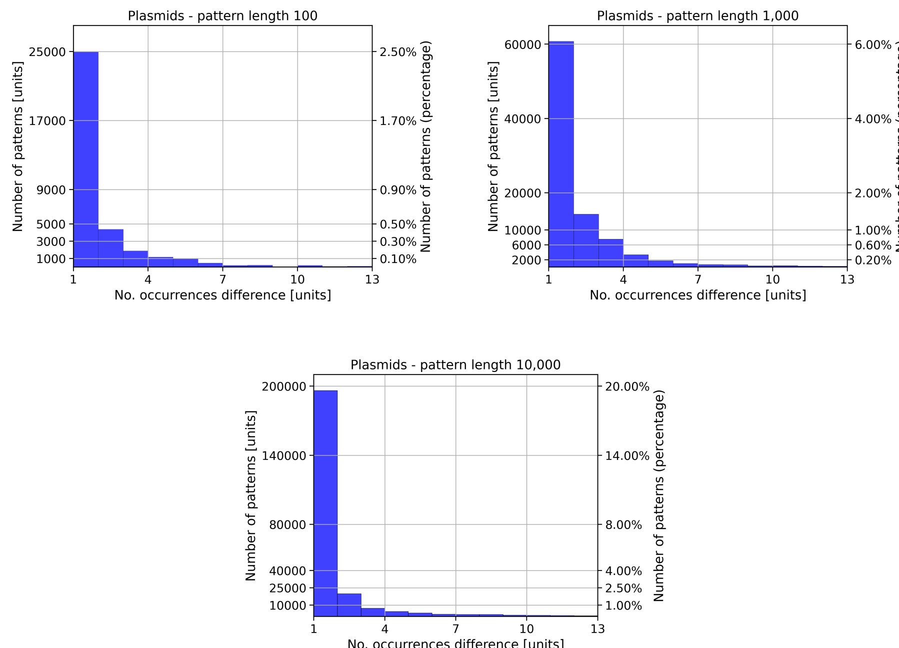


Fig. 7.4: Histograms summarizing the number of occurrences difference between the extended r-index and the r-index on the `plasmids` dataset. Each bar reports the percentage of the number of patterns with a given difference in the number of occurrences between the two indexes. For ease of presentation, we cut the x-axis at 13 since all other bars have a value on the x-axis below 1,000. We omitted the first bin containing the number of patterns whose difference between the two indexes is zero.

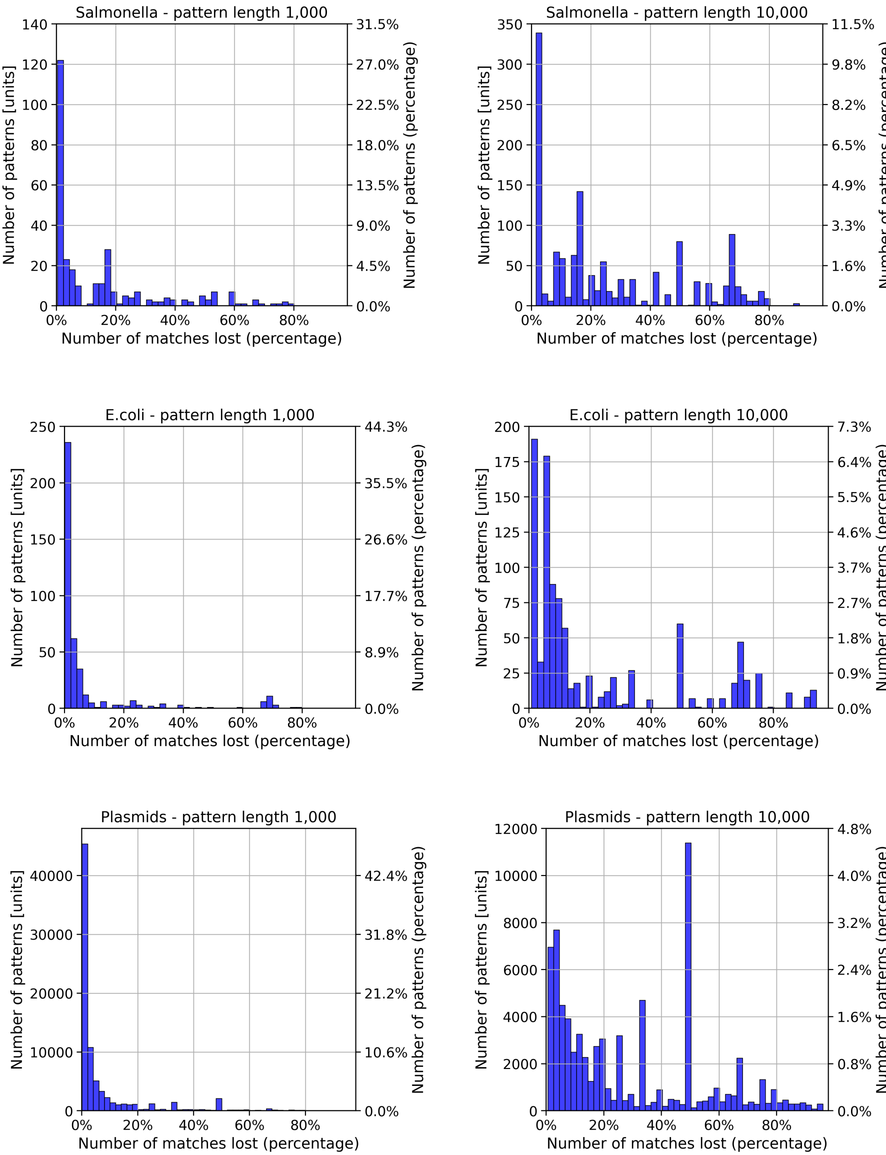


Fig. 7.5: Histograms summarizing the number of matches lost between the extended r-index and the r-index on the three datasets. Each bar shows the percentage of the number of patterns with a given percentage of pattern lost between the two indexes. For this plot we only include the patterns whose number of occurrences is different between the two indexes. We omitted the last bin containing the patterns with zero matches using the r-index.

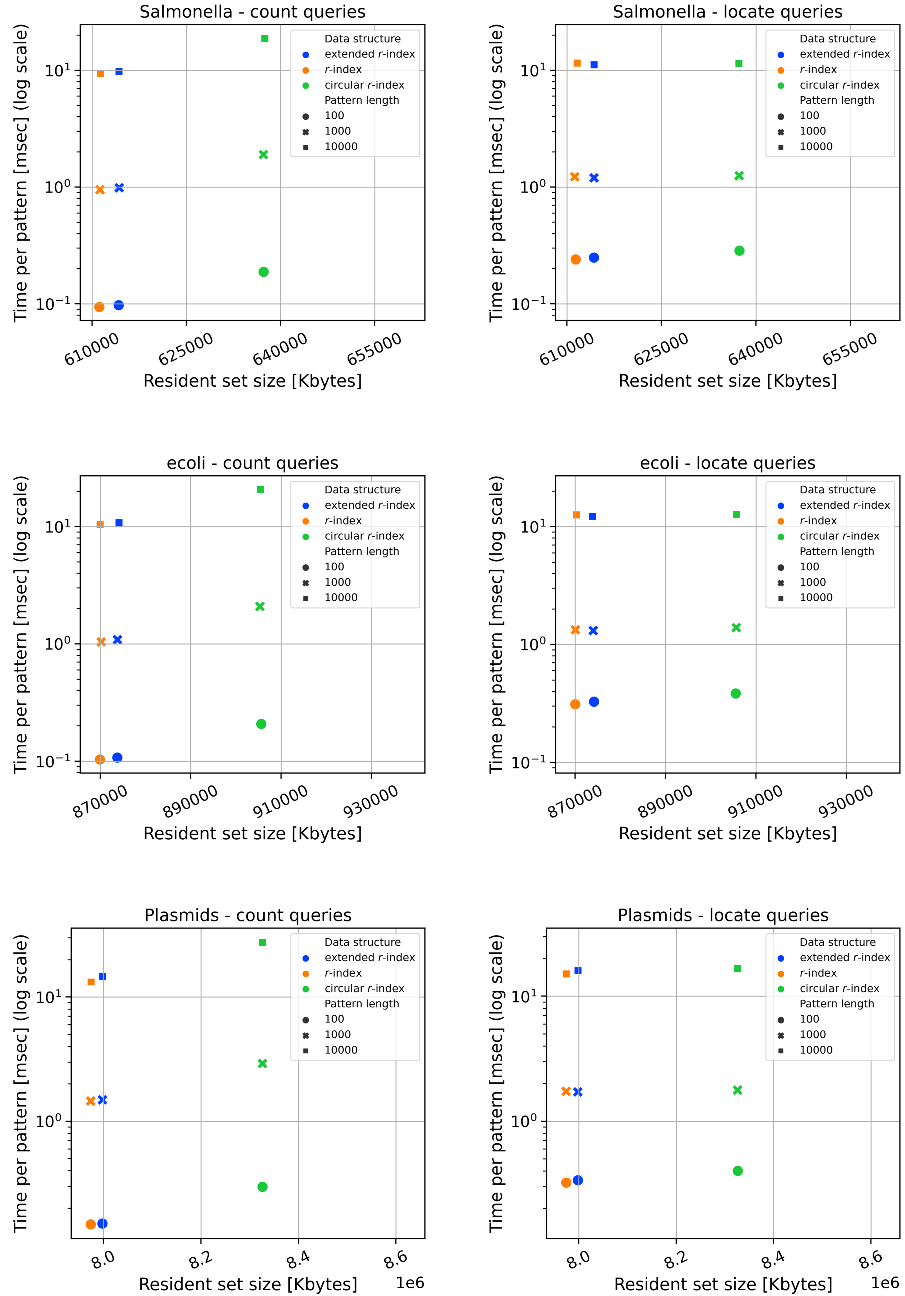


Fig. 7.6: Time and space to perform count and locate queries using the extended r-index and the r-index on 3 sets of patterns containing 1 million sequences of different lengths. Each point in the figures represents a pair consisting of the average time to process a pattern and the peak memory usage of one of the three indexes.

8

Conclusion and suggestions for future research

In this thesis, we focused on studying and developing algorithms to construct the BWT of string collections with a focus on data compression and text indexing.

In the first part of the thesis, we addressed the study of the different BWT extensions to string collections present in the literature: In Chapter 3, we systematically reviewed five different BWT variants and showed that there are important differences between them. Most importantly, these differences also extend to r, the number of runs of the BWT. Moreover, the r parameter is also strongly influenced by the input order of the sequences, especially for string collections containing short sequences. Due to this, we argued for the importance of standardizing r by fixing an input order, such as the optimal order of Bentley et al. In addition to ensuring that r is well-defined, using the optimal BWT also allows to obtain the best compression and performance of BWT-based data structures. We further investigated this topic in Chapter 4 where we presented the first tool to compute the optBWT, i.e. the BWT of string collections which guarantees the minimum number of runs. We showed both on simulated and real-life data that the reduction of the number of runs provided by the optBWT can be large. On our data, we observed a reduction of r by up to a factor of 31.

In the second part of the thesis, we addressed the problem of constructing the extended BWT (eBWT) of Mantaci et al., and a text index based on the eBWT. Among the BWT variants in the literature, the eBWT is the mathematically cleanest generalization of the BWT to string collections; it does not need the addition of the separator characters and is independent of the input order. However, no efficient implementations of the eBWT were available before 2021, and therefore, the eBWT was not part of any tool until now. In Chapters 5 and 6, we presented two efficient linear time algorithms to compute the eBWT of string collections, whose implementations are contained in two tools: `cais` and `pfpebwt`. In addition, we showed that `pfpebwt` can compute the eBWT of very large genomic string collections and has competitive performance compared to other currently available tools. We continued this work by presenting the first text index based on the eBWT: In Chapter 7, we defined an extension of the r-index of Gagie et al. based on the eBWT; we named it *extended r-index*. We implemented this index and showed it has similar time and space requirements as the original r-index while supporting efficient circular pattern matching queries.

8.1 Directions for future research

We conclude this chapter with suggestions for future research directions opened by this thesis.

The number of runs reduction provided by optimal BWT (Chapter 4) opens several new future research directions. An interesting possibility consists in developing a short reads compressor based on the optimal BWT, which exploits the minimum number of runs guaranteed by this transform. A second direction consists in the creation of a text index for large string collections based on the optBWT. This would allow defining a new data structure built on the optimal BWT similar to the extended r-index described in Chapter 7, which guarantees the minimum size of the resulting index. Finally, it would be interesting to extend our `optimalBWT` tool with an algorithm computing the optBWT of large and repetitive string collections by adapting the PFP preprocessing technique described in Chapter 6 to construct the eBWT.

Regarding the extended r-index defined in Chapter 7, as mentioned in the conclusion of the chapter, an interesting future direction consists in extending the more recent improvements of the r-index to the eBWT. This includes implementing the subsampling procedure described in [31], which would reduce the number of sampled GCA positions and the size of the extended r-index, and implementing optimal time queries in r bounded space [97] to speed up pattern search on circular genomes.

A further improvement consists in extending our extended r-index with the functionalities described in Section 7.4 and 7.5.3. This includes implementing the matching statistics and MEM computation on circular strings with our index, computing the thresholds of Rossi et al. [113] using `pfpebwt` and computing the LCP-array of the GCA using `cais`. As for handling non-primitive strings, it is necessary to extend our circular locate machinery and adapt the implementations in Chapter 5 and 6 to construct the GCA of non-primitive strings.

References

1. Omar Ahmed, Massimiliano Rossi, Sam Kovaka, Michael C. Schatz, Travis Gagie, Christina Boucher, and Ben Langmead. Pan-genomic matching statistics for targeted Nanopore sequencing. *iScience*, 24(6):102696, 2021.
2. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.
3. Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *CoRR*, abs/2107.08615, 2021.
4. Christina Ander, Ole Schulz-Trieglaff, Jens Stoye, and Anthony J. Cox. metaBEETL: high-throughput analysis of heterogeneous microbial populations from shotgun DNA sequences. *BMC Bioinform.*, 14(S-5):S2, 2013.
5. Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theor. Comput. Sci.*, 812:96–108, 2020.
6. Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing the bijective BWT. *CoRR*, abs/1911.06985, 2019.
7. Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing the Bijective and the extended Burrows-Wheeler Transform in linear time. In *Proc. of 32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, volume 191 of *LIPIcs*, pages 7:1–7:16, 2021.
8. Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483:134–148, 2013.
9. Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015.
10. Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the complexity of BWT-runs minimization via alphabet reordering. In *Proc. of 28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *LIPIcs*, pages 15:1–15:13, 2020.
11. Jon Louis Bentley, Daniel Dominic Sleator, Robert Endre Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, 1986.
12. Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and LCP arrays in external memory. *ACM J. Exp. Algorithmics*, 21(1):2.3:1–2.3:27, 2016.
13. Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Multithread multistring Burrows-Wheeler Transform and Longest Common Prefix array. *J. Comput. Biol.*, 26(9):948–961, 2019.
14. Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Computing the multi-string BWT and LCP array in external memory. *Theor. Comput. Sci.*, 862:42–58, 2021.

15. Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting conjugates and suffixes of words in a multiset. *Int. J. Found. Comput. Sci.*, 25(8):1161, 2014.
16. Pascal Borry, Heidi Beate Bentzen, Isabelle Budin-Ljøsne, Martina C. Cornel, Heidi Carmen Howard, Oliver Feeney, Leigh Jackson, Deborah Mascalzoni, Álvaro Mendes, Borut Peterlin, Brigida Riso, Mahsa Shabani, Heather Skirton, Sigrid Sterckx, Danya Vears, Matthias Wjst, and Heike Felzmann. The challenges of the expanded availability of genomic information: an agenda-setting paper. *Journal of Community Genetics*, 9(2):103–116, Apr 2018.
17. Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proc. of 28th International Symposium on String Processing and Information Retrieval (SPIRE 2021)*, volume 12944 of *LNCS*, pages 129–142, 2021.
18. Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. *CoRR*, abs/2106.11191, 2021.
19. Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-indexing the eBWT. In *Proc. of 28th International Symposium on String Processing and Information Retrieval (SPIRE 2021)*, volume 12944 of *LNCS*, pages 3–12, 2021.
20. Christina Boucher, Ondrej Cvacho, Travis Gagie, Jan Holub, Giovanni Manzini, Gonzalo Navarro, and Massimiliano Rossi. PFP compressed suffix trees. In *Proc. of 23rd Symposium on Algorithm Engineering and Experiments (ALENEX 2021)*, pages 60–72. SIAM, 2021.
21. Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019.
22. Christina Boucher, Travis Gagie, Alan Kuhnle, and Giovanni Manzini. Prefix-free parsing for building big BWTs. In *Proc. of 18th International Workshop on Algorithms in Bioinformatics (WABI 2018)*, volume 113 of *LIPIcs*, pages 2:1–2:16, 2018.
23. Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
24. Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
25. Bastien Cazaux and Eric Rivals. Linking BWT and XBW via Aho-Corasick automaton: Applications to run-length encoding. In *Proc. of 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*, volume 128 of *LIPIcs*, pages 24:1–24:20, 2019.
26. Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT of very large string collections. *CoRR*, abs/2212.01156, 2022.
27. Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT of very large string collections. In *Proc. of 33rd Data Compression Conference (DCC 2023)*, pages 71–80, 2023.
28. Davide Cenzato and Zsuzsanna Lipták. On different variants of the Burrows-Wheeler-Transform of string collections. In *Proc. of 32nd Data Compression Conference (DCC 2022)*, page 448, 2022.
29. Davide Cenzato and Zsuzsanna Lipták. A theoretical and experimental analysis of BWT variants for string collections. In *Proc. of 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*, volume 223 of *LIPIcs*, pages 25:1–25:18, 2022.

30. Davide Cenzato and Zsuzsanna Lipták. A theoretical and experimental analysis of BWT variants for string collections. *CoRR*, abs/2202.13235, 2022.
31. Dustin Cobas, Travis Gagie, and Gonzalo Navarro. A fast and small subsampled r-index. In *Proc. of 32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, volume 191 of *LIPIcs*, pages 13:1–13:16, 2021.
32. Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinform.*, 28(11):1415–1419, 2012.
33. Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the extended BWT from grammar-compressed DNA sequencing reads. *CoRR*, abs/2102.03961, 2021.
34. Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. In *Proc. of 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*, volume 223 of *LIPIcs*, pages 29:1–29:18, 2022.
35. Robert C. Edgar. Updating the 97% identity threshold for 16S ribosomal RNA OTUs. *Bioinf.*, 34(14):2371–2375, 2018.
36. Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms Mol. Biol.*, 14(1):6:1–6:15, 2019.
37. Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.
38. Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. of 46th IEEE Symposium on Foundations of Computer Science (FOCS 2005)*, pages 184–193, 2005.
39. Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33, 2009.
40. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. of 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398, 2000.
41. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
42. Johannes Fischer and Florian Kurpicz. sais-lite-lcp. https://github.com/kurpicz/sais-lite-lcp. Accessed: 2022-02-05.
43. Travis Gagie, Garance Gourdel, and Giovanni Manzini. Compressing and indexing aligned readsets. In *Proc. of 21st International Workshop on Algorithms in Bioinformatics (WABI 2021)*, volume 201 of *LIPIcs*, pages 13:1–13:21, 2021.
44. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. of 39th ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, pages 1459–1477, 2018.
45. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.
46. Miranda Galey et al. 3-hour genome sequencing and targeted analysis to rapidly assess genetic risk. *medRxiv*, 2022.
47. Ira M. Gessel and Christophe Reutenauer. Counting permutations with given cycle structure and descent set. *J. Comb. Theory, Ser. A*, 64(2):189–215, 1993.
48. Raffaele Giancarlo, Antonio Restivo, and Marinella Sciortino. From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. *Theor. Comput. Sci.*, 387(3):236–248, 2007.

49. Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012.
50. Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Nicola Prezza, Marinella Sciortino, and Anna Toffanello. Novel results on the number of runs of the Burrows-Wheeler-Transform. In *Proc. of 47th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2021)*, volume 12607 of *LNCS*, pages 249–262, 2021.
51. John E Gorzynski et al. Ultrarapid nanopore genome sequencing in a critical care setting. *N. Engl. J. Med.*, 386(7):700–702, 2022.
52. Allison J. Greaney et al. A SARS-CoV-2 variant elicits an antibody response with a shifted immunodominance hierarchy. *PLOS Pathogens*, 18:1–27, 02 2022.
53. Ilya Grebnov. libsais. `https://github.com/IlyaGrebnov/libsais`. Accessed: 2022-02-05.
54. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850. ACM/SIAM, 2003.
55. Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone. Metagenomic analysis through the extended Burrows-Wheeler transform. *BMC Bioinform.*, 21-S(8):299, 2020.
56. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.
57. James Holt and Leonard McMillan. Merging of multi-string BWTs with applications. *Bioinform.*, 30(24):3524–3531, 2014.
58. Wing-Kai Hon, Tsung-Han Ku, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Efficient algorithm for circular Burrows-Wheeler Transform. In *Proc. of 23rd Annual Symposium on Combinatorial Pattern Matching (CPM 2012)*, volume 7354 of *LNCS*, pages 257–268, 2012.
59. Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova. Engineering external memory induced suffix sorting. In *Proc. of 19th Workshop on Algorithm Engineering and Experiments, (ALENEX 2017)*, pages 98–108. SIAM, 2017.
60. Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proc. of 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.
61. Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler Transform conjecture. In *Proc. of 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS 2020)*, pages 1002–1013, 2020.
62. Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
63. Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proc. of 14th Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *LNCS*, pages 200–210, 2003.
64. Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2):143–156, 2005.
65. Dominik Köppl, Daiki Hashimoto, Diptarama Hendrian, and Ayumi Shinohara. In-place Bijective Burrows-Wheeler Transforms. In *Proc. of 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *LIPIcs*, pages 21:1–21:15, 2020.
66. Gregory Kucherov, Lilla Tóthmérész, and Stéphane Vialette. On the combinatorics of suffix arrays. *Inf Process Lett*, 113(22-24):915–920, 2013.
67. Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics

read alignment. In *Proc. of 23rd Annual Conference in Computational Molecular Biology (RECOMB 2019)*, volume 11467 of *LNCS*, pages 158–173, 2019.

68. Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. *J. Comput. Biol.*, 27(4):500–513, 2020.
69. Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012.
70. 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:R25, 2009.
71. N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proc. of IEEE*, 88(11):1722–1732, 2000.
72. Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *ArXiv*, 1303, 2013.
73. Heng Li. Fast construction of FM-index for long sequence reads. *Bioinform.*, 30(22):3274–3275, 2014.
74. Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.
75. Konstantin M. Likhomanov and Arseny M. Shur. Two combinatorial criteria for BWT images. In *Proc. of 6th International Computer Science Symposium in Russia (CSR 2011)*, volume 6651 of *LNCS*, pages 385–396, 2011.
76. Chi-Man Liu, Ruibang Luo, and Tak Wah Lam. GPU-accelerated BWT construction for large collection of short reads. *CoRR*, abs/1401.7457, 2014.
77. Weijun Liu, Ge Nong, Wai Hong Chan, and Yi Wu. Induced sorting suffixes in external memory with better design and less space. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *Proc. of 22nd International Symposium on String Processing and Information Retrieval (SPIRE 2015)*, volume 9309 of *LNCS*, pages 83–94, 2015.
78. M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.
79. Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theor. Comput. Sci.*, 678:22–39, 2017.
80. Felipe A. Louza, Simon Gog, and Guilherme P. Telles. *Construction of Fundamental Data Structures for Strings*. Springer Briefs in Computer Science. Springer, 2020.
81. Felipe A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and Giovanna Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms Mol. Biol.*, 15(1):18, 2020.
82. Felipe A. Louza, Guilherme P. Telles, Steve Hoffmann, and Cristina Dutra de Aguiar Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms Mol. Biol.*, 12(1):26:1–26:16, 2017.
83. Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.
84. Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.
85. Swapan Mallick et al. The Simons Genome Diversity Project: 300 genomes from 142 diverse populations. *Nature*, 538(7624):201–206, 2016.
86. Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
87. Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.

88. Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows-wheeler transform and sturmian words. *Inf. Process. Lett.*, 86(5):241–246, 2003.
89. Giovanni Manzini. XBWT tricks. In *Proc. of 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, volume 9954 of *LNCS*, pages 80–92, 2016.
90. Yuta Mori. libdivsufsort. https://github.com/y-256/libdivsufsort. Accessed: 2022-02-05.
91. Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching reads to many genomes with the r-index. *J. Comput. Biol.*, 27(4):514–518, 2020.
92. J. Ian Munro. Tables. In Vijay Chandru and V. Vinay, editors, *Proc. of 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1996)*, volume 1180 of *LNCS*, pages 37–42.
93. David R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exp.*, 27(8):983–993, 1997.
94. Gonzalo Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016.
95. Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2021.
96. Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2, 2007.
97. Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proc. of 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *LIPIcs*, pages 101:1–101:15, 2021.
98. Ge Nong, Wai Hong Chan, Sheng Qing Hu, and Yi Wu. Induced sorting suffixes in external memory. *ACM Trans. Inf. Syst.*, 33(3):12:1–12:15, 2015.
99. Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.
100. Genome 10K Community of Scientists. A proposal to obtain whole-genome sequence for 10,000 vertebrate species. *J Hered.*, 100:659-674, 2009.
101. Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.
102. Enno Ohlebusch, Stefan Stauß, and Uwe Baier. Trickier XBWT tricks. In *Proc. of 25th International Symposium in String Processing and Information Retrieval (SPIRE 2018)*, volume 11147 of *LNCS*, pages 325–333, 2018.
103. Marco Oliva, Davide Cenzato, Massimiliano Rossi, Zsuzsanna Lipták, Travis Gagie, and Christina Boucher. CSTs for Terabyte-Sized Data. In *Proc. of 32nd Data Compression Conference (DCC 2022)*, pages 93–102. IEEE, 2022.
104. Marco Oliva, Massimiliano Rossi, Jouni Sirén, Giovanni Manzini, Tamer Kahveci, Travis Gagie, and Christina Boucher. Efficiently merging r-indexes. In *Proc. of 31st Data Compression Conference (DCC 2021)*, pages 203–212, 2021.
105. Jacopo Pantaleoni. BWT of large string sets. *CoRR*, abs/1410.0562, 2014.
106. Denis Perrin and Antonio Restivo. Enumerative combinatorics on words. In *Handbook of Enumerative Combinatorics, ed. by Miklos Bona.* 2015.
107. Pizza & Chili repetitive corpus. Available at http://pizzachili.dcc.uchile.cl/repcorpus.html. Accessed 16 April 2020.
108. Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.
109. Nicola Prezza, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. SNPs detection by eBWT positional clustering. *Algorithms Mol. Biol.*, 14(1):3:1–3:13, 2019.

110. Nicola Prezza, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. Variable-order reference-free variant discovery with the Burrows-Wheeler Transform. *BMC Bioinform.*, 21-S(8):260, 2020.
111. Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):4, 2007.
112. Simon J. Puglisi and Bella Zhukova. Document retrieval hacks. In *Proc. of 19th International Symposium on Experimental Algorithms (SEA 2021)*, volume 190 of *LIPIcs*, pages 12:1–12:12, 2021.
113. Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *J. Comput. Biol.*, 29(2):169–187, 2022.
114. Georges P Schmartz, Anna Hartung, Pascal Hirsch, Fabian Kern, Tobias Fehlmann, Rolf Müller, and Andreas Keller. PLSDB: advancing a comprehensive database of bacterial plasmids. *Nucleic Acids Res.*, 50(D1):D273–D278, 11 2021.
115. Yossi Shiloach. Fast canonization of circular strings. *J Algorithms*, 2(2):107–121, 1981.
116. Jouni Sirén. Burrows-Wheeler Transform for terabases. In *Proc. of 26th Data Compression Conference (DCC 2016)*, pages 211–220, 2016.
117. Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. of 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008)*, volume 5280 of *LNCS*, pages 164–175, 2008.
118. Tyler N. Starr et al. Deep mutational scanning of SARS-CoV-2 receptor binding domain reveals constraints on folding and ACE2 binding. *Cell*, 182(5):1295–1310.e20, 2020.
119. Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: Astronomical or genomical? *PLOS Biology*, 13(7):1–11, 07 2015.
120. Eric L. Stevens et al. The public health impact of a publically available, environmental database of microbial genomes. *Front Microbiol.*, 8:808, 2017.
121. Chen Sun et al. RPAN: rice pan-genome browser for 3000 rice genomes. *Nucleic Acids Res.*, 45(2):597–605, 2017.
122. The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.
123. The 1001 Genomes Consortium. Epigenomic Diversity in a Global Collection of Arabidopsis thaliana Accessions. *Cell*, 166(2):492–505, 2016.
124. The COVID-19 Data Portal. Available at https://www.covid19dataportal.org/. Accessed 17-05-2021.
125. Clare Turnbull et al. The 100,000 genomes project: bringing whole genome sequencing to the NHS. *Br. Med. J.*, 361, 2018.
126. Silvie Van den Hoecke, Judith Verhelst, Marnik Vuylsteke, and Xavier Saelens. Analysis of the genetic diversity of influenza A viruses using next-generation DNA sequencing. *BMC Genomics*, 16(1):79, 2015.
127. Peter Weiner. Linear pattern matching algorithms. In *Proc. of 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11, 1973.
128. Raf Winand et al. Targeting the 16s rRNA gene for bacterial identification in complex mixed samples: Comparative evaluation of second (Illumina) and third (Oxford nanopore technologies) generation sequencing technologies. *Int. J. of Mol. Sci.*, 21(1):298, 2019.
129. Michael H. Woodworth et al. Sentinel case of Candida auris in the Western United States Following Prolonged Occult Colonization in a Returned Traveler from India. *Microb Drug Resist*, 25(5):677–680, 2019.

130. Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.
131. Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, 24(5):530–536, 1978.

A

Full experimental results for Chapter 3

A.1 Further information on the tools

We extensively tested all tools we cited in Chapter 3 and determined which data structure they compute, using both our tests and the algorithm descriptions in the respective papers. In this section, we include further information about the tools.

- `pfpebwt` is the tool computing the eBWT of string collections (`https://github.com/davidecenzato/PFP-eBWT.git`) we described in Chapter 6. It takes in input a fasta file and gives in output the eBWT in either plain ASCII text or RLE (run-length-encoded) format. We used (a) no flags for long sequences, and (b) the flags `-w 10 -p 10 -n 3 --reads` for short sequences. We included it in two different rows of Table 3.1 because by default `pfpebwt` computes the eBWT, but it can compute the dolEBWT if the sequences have explicit end-of-string characters (not in multi-thread mode).
- `cais` is the tool implementing the `SAIS_for_eBWT` (Chapter 5) algorithm, which computes four BWT types: (i) the eBWT, (ii) the dolEBWT, (iii) the BWT of a single sequence without an end-of-string symbol, and (iv) the BBWT of a text (`https://github.com/davidecenzato/cais.git`) depending on the input flag. It takes in input a fasta file, a fastq file, or a plain text file and gives in output one of the four transforms in plain ASCII text. The `-c` and `-a` flags enable the conjugate array computation, while `-s` flag allows using a `sdsl's sd_vector` bitvector to represent the string boundaries.
- `BEETL` is a suite containing several tools, including a tool computing the mdolBWT of string collections using an implementation of the BCR and BCR-ext algorithms [8] (`https://github.com/BEETL/BEETL.git`). All input sequences have to have the same length. We tested this tool using `-output-format ASCII` and `-concatenate-output` flags. This tool also computes the sapBWT (Chapter 4) by using the `-sap-ordering` flag (BCR-ext mode only).
- `BCR_LCP_GSA` is a tool computing the mdolBWT of sting collections in semi-external memory (`https://github.com/giovannarosone/BCR_LCP_GSA`). It implements an algorithm similar to `BCR` contained in the BEETL tool, but

it can process a string collection containing sequences of different lengths. It takes in input a fasta file, a fastq file, or a gz-compressed fastq file. It computes the mdolBWT following the method of Bauer et al., described in [8]. We set the `'dataTypeLengthSequences'` variable in `Parameters.h` to 1.

- `ropebwt2` is a tool computing the FM-index and the mdolBWT of string collections (`https://github.com/lh3/ropebwt2.git`), using an approach similar to `BCR`. It takes in input a fasta file, a fastq file, or a gz compressed fastq file. We listed it in two different rows of Table 3.1 because it computes the mdolBWT or the colexBWT, depending on the flags. We used the `-R` and the `-R -s` flags, respectively, to obtain the two transforms. In addition, we modified `main.c` in order to change the order of the characters to `$ < A < C < G < N < T`.
- `BigBWT` computes the concBWTof string collections, and optionally the suffix array, of a highly repetitive text or string collection (`https://github.com/alshai/Big-BWT.git`). It takes in input a newline separated file or a fasta file. This tool with the `-f` flag is used internally in the r-index (`https://github.com/alshai/r-index`), producing the BWT of the input strings concatenated without dollars; the separator symbols have to be added beforehand in the fasta file. On the other hand, the tool without the `-f` flag will compute the BWT of the fasta files without skipping the fasta headers. We used standard parameters and new-line separated files as input; the output then is the concBWT.
- `grlBWT` is a tool computing the mdolBWT of string collections using an algorithm that keeps the intermediate data structures in compressed form (`https://github.com/ddiazdom/grlBWT`). It takes in input a concatenated string collection and gives in output the mdolBWT in run-length compressed form. We tested it with the default parameters and used new-line separated files as input. We parsed the results in plain text format.
- `merge-BWT` computes the mdolBWT of string collections by merging the BWTs of subcollections of the input (`https://github.com/jltsiren/bwt-merge.git`). It takes in input a list of one or several mdolBWTs. The order of the dollars will depend on the order in which the input BWTs are listed. We tested it using `-i plain_sorted and -o plain_sorted` flags. We computed the BWTs of the subcollections using `ropebwt2`.
- `nvSetBWT` is a tool included in `nvbio` suite (`https://github.com/NVlabs/nvbio.git`). It takes in input either a fastq or a newline separated file. We tested it using the `-R` flag for skipping the reverse strand. However, even if the algorithmic descriptions in [76,105] seem to describe the mdolBWT, the output of the current version (version 1.1) does not correspond to a possible BWT because the Parikh vector is different from that of the input.
- `eGSA` computes the generalized enhanced suffix array and the mdolBWT of string collections (`https://github.com/felipelouza/egsa.git`). It takes in input a text file, a fasta file, or a fastq file. It uses the gSACA-K [79] algorithm for computing the suffix array of subcollections of the input and then merges all suffix arrays. Thus it computes the mdolBWT. We tested it with the `-b` flag.

- eGAP computes the mdolBWT, and optionally the LCP-array and DA (document array) of string collections (https://github.com/felipelouza/egap.git). It works in semi-external memory by merging BWTs of increasing subsets of the input. It takes in input a newline separated file, a fasta file, or a fastq file. We tested it with default settings.
- bwt-lcp-parallel computes the mdolBWT and the LCP-array of short string collections (https://github.com/AlgoLab/bwt-lcp-parallel.git). It takes in input fasta files and does not support the N character. We tested it using standard settings.
- gsufsort computes the SA, LCP and mdolBWT of string collections (https://github.com/felipelouza/gsufsort.git), using the gSACA-K algorithm of [79]. It takes in input a newline separated file, a fasta file, or a fastq file. We tested it using --fasta and --bwt flags.
- G2BWT is a tool computing the dolEBWT of short string collections (https://bitbucket.org/DiegoDiazDominguez/lms_grammar/src/bwt_imp2). It takes in input newline separated files. Even though it is not stated explicitly, this tool computes the dolEBWT because, when it constructs the grammar, it uses dollars for separating adjacent strings. Thus, also the string rotations will contain dollars. We tested it using the default settings.
- msbwt is a tool implementing the Holt and McMillan [57] merge-based BWT construction algorithm (https://github.com/holtjma/msbwt.git). It takes in input a list of one or several fastq files. Even if this tool uses the BCR approach [8] for computing the BWTs to merge, it actually computes the dolEBWT. This is because it features a preprocessing where it sorts the input strings lexicographically. Thus, the resulting mdolBWT corresponds to the dolEBWT.

A.2 Results on individual datasets

In this section, we include the tables containing the full experimental results on the eight datasets used in Chapter 3.

SARS-CoV-2 short (500,000 short sequences)

Hamming d. / *norm. Hamming d.*	*Hamming distance on the big dataset*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	3,014,183	2,926,602	2,912,860
mdolBWT	0.11820	0	3,013,908	3,102,887
concBWT	0.11477	0.11819	0	3,013,634
colexBWT	0.11423	0.12168	0.11818	0

dataset properties	
no. sequences	500,000
average length	50
total length	25,000,000
no. of interesting intervals	116,598
total length intr.int.s	20,187,840
fraction pos.s in intr.int.s	0.792
variability	0.210

no. runs big dataset		
	r	n/r
eBWT	1,902,148	13.143
dolEBWT	1,868,581	13.647
mdolBWT	3,113,818	8.189
concBWT	3,402,513	7.494
colexBWT	808,906	31.524
optimum	725,979	35.125

Hamming d. / *norm. Hamming d.*	*Hamming distance on a subset of 5,000 sequences*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	21,362	21,196	20,626
mdolBWT	0.08377	0	21,376	21,256
concBWT	0.08312	0.08383	0	21,259
colexBWT	0.08089	0.08336	0.08337	0

small dataset properties	
no. of sequences	5,000
total length	250,000
average length	100
no. of interesting intervals	2,476
total length intr.int.s	180,038
fraction pos.s in intr.int.s	0.706
variability	0.173

edit d. / *norm. edit d.*	*edit distance on a subset of 5,000 sequences*				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	28,702	43,903	43,828	46,936
dolEBWT	0.11256	0	17,000	16,921	20,104
mdolBWT	0.17217	0.06667	0	16,130	20,812
concBWT	0.17187	0.06636	0.06325	0	20,830
colexBWT	0.18406	0.07884	0.08162	0.08169	0

no. runs small dataset		
	r	n/r
eBWT	52,979	4.719
dolEBWT	50,803	5.019
mdolBWT	54,766	4.656
concBWT	54,698	4.662
colexBWT	37,320	6.833
optimum	35,904	7.102

Table A.1: Results for the SARS-CoV-2 short dataset. First row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. First row right: summary of the dataset properties. Second row: number of runs and average run-length (n/r) of all BWT variants. Third row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants on a subset of the input collection. Third row right: summary of the dataset properties of a subset of the input collection. Fourth row left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Fourth row right: number of runs and average run-length (n/r) of all BWT variants on a subset of the input collection.

Simons Diversity reads (500,000 short sequences)

Hamming d. / *norm. Hamming d.*	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	3,624,283	3,602,362	3,594,438
mdolBWT	0.07249	0	3,628,799	3,623,154
concBWT	0.07133	0.07186	0	3,617,679
colexBWT	0.07189	0.07246	0.07168	0

Hamming distance on the big dataset

dataset properties	
no. of sequences	500,000
total length	50,000,000
average length	100
no. of interesting intervals	316,013
total length intr.int.s	5,387,549
fraction pos.s in intr.int.s	0.107
variability	0.976

no. runs big dataset	r	n/r
eBWT	8,974,105	5.572
dolEBWT	9,337,122	5.409
mdolBWT	9,362,564	5.394
concBWT	9,530,334	5.299
colexBWT	6,414,356	7.873
optimum	6,209,567	8.133

Hamming distance on a subset of 5,000 sequences

Hamming d. / *norm. Hamming d.*	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	23,742	23,461	23,535
mdolBWT	0.04748	0	23,785	23,722
concBWT	0.04646	0.04710	0	23,660
colexBWT	0.04707	0.04744	0.04685	0

small dataset properties	
no. of sequences	5,000
total length	500,000
average length	100
no. of interesting intervals	3,111
total length intr.int.s	35,404
fraction pos.s in intr.int.s	0.070
variability	0.989

edit distance on a subset of 5,000 sequences

edit d. / *norm. edit d.*	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	72,898	72,878	72,918	74,026
dolEBWT	0.14435	0	17,820	17,560	22,481
mdolBWT	0.14431	0.03529	0	17,726	22,586
concBWT	0.14439	0.03477	0.03510	0	22,595
colexBWT	0.14659	0.04452	0.04472	0.04474	0

no. runs small dataset	r	n/r
eBWT	77,646	6.439
dolEBWT	81,758	6.177
mdolBWT	81,883	6.167
concBWT	82,779	6.101
colexBWT	64,229	7.862
optimum	62,117	8.130

Table A.2: Results for the Simons Diversity reads dataset. First row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. First row right: summary of the dataset properties. Second row: number of runs and average run-length (n/r) of all BWT variants. Third row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants on a subset of the input collection. Third row right: summary of the dataset properties of a subset of the input collection. Fourth row left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Fourth row right: number of runs and average run-length (n/r) of all BWT variants on a subset of the input collection.

16S rRNA short (500,000 short sequences)

Hamming d. / norm. Hamming d.	Hamming distance on the big dataset			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	2,202,008	2,540,310	1,748,072
mdolBWT	0.02881	0	2,201,003	2,202,717
concBWT	0.03324	0.02880	0	2,784,600
colexBWT	0.02287	0.02882	0.03643	0

dataset properties	
no. of sequences	500,000
total length	75,929,833
average length	152
no. of interesting intervals	54,366
total length intr.int.s	56,708,529
fraction pos.s in intr.int.s	0.742
variability	0.058

no. runs big dataset		
	r	n/r
eBWT	1,992,130	38.115
dolEBWT	1,992,211	38.364
mdolBWT	4,057,541	18.836
concBWT	2,767,797	27.614
colexBWT	1,727,127	44.253
optimum	1,703,234	44.873

Hamming d. / norm. Hamming d.	Hamming distance on a subset of 5,000 sequences			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	20,159	23,229	15,835
mdolBWT	0.02635	0	20,024	20,092
concBWT	0.03036	0.02617	0	25,464
colexBWT	0.02070	0.02626	0.03329	0

small dataset properties	
no. of sequences	5,000
total length	765,037
average length	152
no. of interesting intervals	1,376
total length intr.int.s	139,041
fraction pos.s in intr.int.s	0.182
variability	0.222

edit d. / norm. edit d.	edit distance on a subset of 5,000 sequences				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	51,683	62,799	63,303	61,732
dolEBWT	0.06756	0	16,968	20,180	14,166
mdolBWT	0.08209	0.02218	0	16,695	19,371
concBWT	0.08274	0.02638	0.02182	0	21,683
colexBWT	0.08069	0.01852	0.02532	0.02834	0

no. runs small dataset		
	r	n/r
eBWT	35,262	21.554
dolEBWT	35,293	21.677
mdolBWT	50,581	15.125
concBWT	38,900	19.667
colexBWT	30,568	25.027
optimum	30,007	25.495

Table A.3: Results for the 16S rRNA short dataset. First row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. First row right: summary of the dataset properties. Second row: number of runs and average run-length (n/r) of all BWT variants. Third row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants on a subset of the input collection. Third row right: summary of the dataset properties of a subset of the input collection. Fourth row left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Fourth row right: number of runs and average run-length (n/r) of all BWT variants on a subset of the input collection.

Influenza A reads (500,000 short sequences)

Hamming d. / *norm. Hamming d.*	*Hamming distance on the big dataset*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	3,040,590	3,038,509	2,938,706
mdolBWT	0.02617	0	3,039,095	3,041,816
concBWT	0.02615	0.02616	0	3,089,670
colexBWT	0.02529	0.02618	0.02659	0

dataset properties	
no. of sequences	500,000
total length	116,192,842
average length	231
no. of interesting intervals	213,735
total length intr.int.s	11,995,246
fraction pos.s in intr.int.s	0.103
variability	0.363

no. runs big dataset		
	r	n/r
eBWT	3,258,605	35.504
dolEBWT	3,298,502	35.226
mdolBWT	5,030,032	23.100
concBWT	4,629,150	25.100
colexBWT	2,362,987	49.172
optimum	2,311,133	50.275

Hamming d. / *norm. Hamming d.*	*Hamming distance on a subset of 5,000 sequences*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	23,456	23,456	22,873
mdolBWT	0.02018	0	23,509	23,407
concBWT	0.02018	0.02023	0	24,061
colexBWT	0.01968	0.02014	0.02070	0

small dataset properties	
no. of sequences	5,000
total length	1,162,319
average length	231
no. of interesting intervals	3,062
total length intr.int.s	36,019
fraction pos.s in intr.int.s	0.031
variability	0.966

edit d. / *norm. edit d.*	*edit distance on a subset of 5,000 sequences*				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	75,966	75,935	75,991	76,437
dolEBWT	0.06536	0	18,043	18,316	21,869
mdolBWT	0.06533	0.01552	0	17,835	22,536
concBWT	0.06538	0.01576	0.01534	0	23,078
colexBWT	0.06576	0.01881	0.01939	0.01986	0

no. runs small dataset		
	r	n/r
eBWT	81,992	14.115
dolEBWT	85,489	13.596
mdolBWT	89,256	13.022
concBWT	87,867	13.228
colexBWT	70,534	16.479
optimum	68,900	16.870

Table A.4: Results for the Influenza A reads dataset. First row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. First row right: summary of the dataset properties. Second row: number of runs and average run-length (n/r) of all BWT variants. Third row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants on a subset of the input collection. Third row right: summary of the dataset properties of a subset of the input collection. Fourth row left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Fourth row right: number of runs and average run-length (n/r) of all BWT variants on a subset of the input collection.

SARS-CoV-2 long (50,000 long sequences)

Hamming d. \ norm. Hamming d.	*Hamming distance on the big dataset*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	248,189	248,205	255,357
mdolBWT	0.00462	0	248,572	248,631
concBWT	0.00462	0.00462	0	248,765
colexBWT	0.00475	0.00462	0.00463	0

dataset properties	
no. sequences	50,000
total length	53,776,351
average length	1,075
no. of interesting intervals	31,931
total length intr.int.s	9,436,894
fraction pos.s in intr.int.s	0.17548
variability	0.03716

no. runs big dataset		
	r	n/r
eBWT	882,634	60.870
dolEBWT	879,608	61.137
mdolBWT	934,129	57.568
concBWT	934,117	57.569
colexBWT	734,610	73.204
optimal	721,845	74.498

Hamming d. \ norm. Hamming d.	*Hamming distance on a subset of 1,500 sequences*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	4,936	4,939	5,296
mdolBWT	0.00306	0	4,884	4,908
concBWT	0.00306	0.00303	0	5,012
colexBWT	0.00328	0.00304	0.00310	0

small dataset properties	
no. sequences	1,500
total length	1,612,956
average length	1,075
no. of interesting intervals	1,046
total length intr.int.s	152,035
fraction pos.s in intr.int.s	0.094
variability	0.047

edit d. \ norm. edit d.	*edit distance on a subset of 1,500 sequences*				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	19,140	21,809	21,796	22,618
dolEBWT	0.01186	0	4,345	4,322	5,186
mdolBWT	0.01351	0.00269	0	4,110	4,820
concBWT	0.01350	0.00306	0.00255	0	4,893
colexBWT	0.01401	0.00321	0.00299	0.00303	0

no. runs small dataset		
	r	n/r
eBWT	45,262	35.636
dolEBWT	45,155	35.754
mdolBWT	45,572	35.426
concBWT	45,644	35.371
colexBWT	42,516	37.973
optimum	42,093	38.355

Table A.5: Results for the SARS-CoV-2 long dataset. First row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. First row right: summary of the dataset properties. Second row: number of runs and average run-length (n/r) of all BWT variants. Third row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants on a subset of the input collection. Third row right: summary of the dataset properties of a subset of the input collection. Fourth row left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Fourth row right: number of runs and average run-length (n/r) of all BWT variants on a subset of the input collection.

16S rRNA long (16,741 long sequences)

Hamming d. / norm. Hamming d.	Hamming distance on the big dataset			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	85,960	42,948	67,103
mdolBWT	0.00342	0	85,961	82,890
concBWT	0.00171	0.00342	0	71,264
colexBWT	0.00267	0.00329	0.00283	0

dataset properties	
no. sequences	16,741
total length	25,159,064
average length	1,501
no. of interesting intervals	9,918
total length intr.int.s	1,173,284
fraction pos.s in intr.int.s	0.047
variability	0.104

no. runs big dataset		
	r	n/r
eBWT	547,991	45.881
dolEBWT	547,793	45.928
mdolBWT	555,687	45.276
concBWT	558,902	45.015
colexBWT	536,682	46.879
optimum	533,712	47.140

Hamming d. / norm. Hamming d.	Hamming distance on a subset of 1,500 sequences			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	4,740	3,104	3,926
mdolBWT	0.00210	0	4,716	4,783
concBWT	0.00137	0.00209	0	4,208
colexBWT	0.00174	0.00212	0.00186	0

small dataset properties	
no. sequences	1,500
total length	2,260,229
average length	1,501
no. of interesting intervals	946
total length intr.int.s	72,933
fraction pos.s in intr.int.s	0.032
variability	0.104

edit d. / norm. edit d.	edit distance on a subset of 1,500 sequences				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	18,328	22,194	21,021	21,987
dolEBWT	0.00811	0	4,410	2,761	3,858
mdolBWT	0.00982	0.00195	0	4,323	4,691
concBWT	0.00930	0.00122	0.00191	0	4,146
colexBWT	0.00973	0.00171	0.00208	0.00183	0

no. runs small dataset		
	r	n/r
eBWT	62,077	36.386
dolEBWT	62,031	36.437
mdolBWT	62,712	36.041
concBWT	62,800	35.991
colexBWT	61,235	36.911
optimal	60,979	37.066

Table A.6: Results for the 16S rRNA long dataset. First row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. First row right: summary of the dataset properties. Second row: number of runs and average run-length (n/r) of all BWT variants. Third row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants on a subset of the input collection. Third row right: summary of the dataset properties of a subset of the input collection. Fourth row left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Fourth row right: number of runs and average run-length (n/r) of all BWT variants on a subset of the input collection.

Candida auris reads (50,000 long sequences)

Hamming d. / *norm. Hamming d.*	*Hamming distance on the big dataset*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	306,071	306,431	305,665
mdolBWT	0.00246	0	305,649	305,713
concBWT	0.00247	0.00246	0	305,469
colexBWT	0.00246	0.00246	0.00246	0

dataset properties	
no. sequences	50,000
total length	124,200,880
average length	2,483
no. interesting intervals	39,076
total length intr.int.s	913,721
fraction pos.s in intr.int.s	0.007
variability	0.497

no. runs big dataset		
	r	n/r
eBWT	72,014,777	1.724
dolEBWT	71,972,783	1.726
mdolBWT	71,972,346	1.726
concBWT	71,973,221	1.726
colexBWT	71,725,274	1.732
optimal	71,704,473	1.732

Hamming d. / *norm. Hamming d.*	*Hamming distance on a subset of 1,500 sequences*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	6,333	6,393	6,260
mdolBWT	0.00169	0	6,411	6,354
concBWT	0.00170	0.00171	0	6,294
colexBWT	0.00167	0.00169	0.00168	0

small dataset properties	
no. sequences	1,500
total length	3,755,776
average length	2,503
no. of interesting intervals	1,189
total length intr.int.s	18,372
fraction pos.s in intr.int.s	0.005
variability	0.530

edit d. / *norm. edit d.*	*edit distance on a subset of 1,500 sequences*				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	30,345	30,552	30,562	30,794
dolEBWT	0.00808	0	4,835	4,860	6,005
mdolBWT	0.00813	0.00129	0	4,824	6,105
concBWT	0.00814	0.00129	0.00128	0	6,035
colexBWT	0.00820	0.00160	0.00163	0.00161	0

no. runs small dataset		
	r	n/r
eBWT	2,635,300	1.425
dolEBWT	2,633,676	1.426
mdolBWT	2,633,652	1.426
concBWT	2,633,727	1.426
colexBWT	2,629,094	1.429
optimum	2,628,470	$1,429$

Table A.7: Results for the Candida auris reads dataset. First row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. First row right: summary of the dataset properties. Second row: number of runs and average run-length (n/r) of all BWT variants. Third row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants on a subset of the input collection. Third row right: summary of the dataset properties of a subset of the input collection. Fourth row left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Fourth row right: number of runs and average run-length (n/r) of all BWT variants on a subset of the input collection.

SARS-CoV-2 genomes (2,000 long sequences)

Hamming d. / *norm. Hamming d.*	*Hamming distance on the big dataset*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	7,958	7,900	7,263
mdolBWT	0.00013	0	7,958	7,957
concBWT	0.00013	0.00013	0	7,990
colexBWT	0.00012	0.00013	0.00013	0

dataset properties	
no. sequences	2,000
total length	59,612,692
average length	29,085
no. interesting intervals	1863
total length intr.int.s	80,486
fraction pos.s in intr.int.s	0.001
variability	0.148

no. runs big dataset		
	r	n/r
eBWT	117,628	506.773
dolEBWT	117,410	507.731
mdolBWT	118,870	501.495
concBWT	119,334	499.549
colexBWT	114,287	521.605
optimum	113,930	523.240

Hamming d. / *norm. Hamming d.*	*Hamming distance on a subset of 50 sequences*			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	105	119	90
mdolBWT	0.00007	0	124	116
concBWT	0.00008	0.00008	0	118
colexBWT	0.00006	0.00008	0.00008	0

small dataset properties	
no. sequences	50
total length	1,490,184
average length	29,802
no. interesting intervals	43
total length intr.int.s	271
fraction pos.s in intr.int.s	$1.8 \cdot 10^{-4}$
variability	0.690

edit d. / *norm. edit d.*	*edit distance on a subset of 50 sequences*				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	786	795	801	791
dolEBWT	0.00053	0	98	107	86
mdolBWT	0.00053	0.00007	0	105	112
concBWT	0.00054	0.00007	0.00007	0	114
colexBWT	0.00053	0.00006	0.00008	0.00008	0

no. runs small dataset		
	r	n/r
eBWT	25,258	58.997
dolEBWT	25,255	59.006
mdolBWT	25,274	58.961
concBWT	25,285	58.936
colexBWT	25,221	59.085
optimum	25,210	59.111

Table A.8: Results for the SARS-CoV-2 genomes dataset. First row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. First row right: summary of the dataset properties. Second row: number of runs and average run-length (n/r) of all BWT variants. Third row left: absolute and normalized pairwise Hamming distance between separator-based BWT variants on a subset of the input collection. Third row right: summary of the dataset properties of a subset of the input collection. Fourth row left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Fourth row right: number of runs and average run-length (n/r) of all BWT variants on a subset of the input collection.