

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday

Manzini's Festschrift, July 25, 2025, Venice, Italy

Edited by

Paolo Ferragina

Travis Gagie

Gonzalo Navarro



Editors

Paolo Ferragina 

Scuola Superiore Sant'Anna, Pisa, Italy
University of Pisa, Italy
Paolo.Ferragina@santannapisa.it

Travis Gagie 

Dalhousie University, Halifax, Canada
travis.gagie@gmail.com

Gonzalo Navarro 

University of Chile, Santiago, Chile
gnavarro@dcc.uchile.cl

ACM Classification 2012

Theory of computation → Data structures and algorithms for data management; Theory of computation
→ Data compression

ISBN 978-3-95977-390-4

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-390-4>.

Publication date

August, 2025

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <https://portal.dnb.de>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASlcs.Manzini.2025.0

ISBN 978-3-95977-390-4

ISSN 1868-8969

<https://www.dagstuhl.de/oasics>

OASlcs – OpenAccess Series in Informatics

OASlcs is a series of high-quality conference proceedings across all fields in informatics. OASlcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 1868-8969

<https://www.dagstuhl.de/oasics>

Contents

Preface	
<i>Paolo Ferragina, Travis Gagie, and Gonzalo Navarro</i>	0:vii
Messages for Giovanni	
<i>Various authors</i>	0:ix

Papers

BWT and Combinatorics on Words	
<i>Gabriele Fici, Sabrina Mantaci, Antonio Restivo, Giuseppe Romana, Giovanna Rosone, and Marinella Sciortino</i>	1:1–1:23
A Survey of the Bijective Burrows-Wheeler Transform	
<i>Hideo Bannai, Dominik Köppl, and Zsuzsanna Lipták</i>	2:1–2:26
BWT for String Collections	
<i>Davide Cenzato, Zsuzsanna Lipták, Nadia Pisanti, Giovanna Rosone, and Marinella Sciortino</i>	3:1–3:29
Wavelet Tree, Part II: Text Indexing	
<i>Paolo Ferragina, Raffaele Giancarlo, Roberto Grossi, Giovanna Rosone, Rossano Venturini, and Jeffrey Scott Vitter</i>	4:1–4:10
FM-Adaptive: A Practical Data-Aware FM-Index	
<i>Hongwei Huo, Zongtao He, Pengfei Liu, and Jeffrey Scott Vitter</i>	5:1–5:23
Optimizing the Performance of the FM-Index for Large-Scale Data	
<i>Eddie Ferro and Christina Boucher</i>	6:1–6:21
Algorithms for Computing Very Large BWTs: a Short Survey	
<i>Diego Díaz-Domínguez, Lavinia Egidi, Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone</i>	7:1–7:28
A Taxonomy of LCP-Array Construction Algorithms	
<i>Johannes Fischer and Enno Ohlebusch</i>	8:1–8:17
Search Schemes for Approximate Pattern Matching: An Overview	
<i>Lore Depuydt, Jan Fostier, Simon Gottlieb, Gregory Kucherov, Knut Reinert, and Luca Renders</i>	9:1–9:16
Phasing Data from Genotype Queries via the μ -PBWT	
<i>Davide Cozzi, Paola Bonizzoni, Christina Boucher, Ben Langmead, and Yuri Pirola</i>	10:1–10:17
Circular Dictionary Matching Using Extended BWT	
<i>Wing-Kai Hon, Rahul Shah, and Sharma V. Thankachan</i>	11:1–11:14
Wheeler Graphs and Wheeler Languages	
<i>Nicola Cotumaccio, Giovanna D’Agostino, Daniel Gibney, Alberto Policriti, Nicola Prezza, and Sharma V. Thankachan</i>	12:1–12:28

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini’s 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Graph Indexing Beyond Wheeler Graphs

*Jarno N. Alanko, Elena Biagi, Massimo Equi, Veli Mäkinen, Simon J. Puglisi,
Nicola Rizzo, Kunihiko Sadakane, and Jouni Sirén* 13:1–13:29

BWT Indexes for Optimal Joins in Graph Databases

Diego Arroyuelo and Gonzalo Navarro 14:1–14:19

Preface

Happy 60th Birthday, Giovanni!

Wow, it seems like such a short time (in some ways!) since we were celebrating 25 years of the BWT and 50 years of Gonzalo in 2019, and now here we are celebrating 25 years of the FM-index and the CSA, and 60 years of you!

Looking through this volume, at the growing breadth of research on the BWT and FM-indexing – most of which would not have happened without you – we hope you are duly proud of what you have accomplished, both as a researcher and a mentor and role model to other researchers. Rather than add more words ourselves, we gladly leave the floor to the many colleagues whose quotes follow in the next pages – each offering their own heartfelt birthday wishes.

We are all proud of you, of course, and we wish you many more productive years!

Grazie mille e tantissimi auguri!

Paolo, Travis and Gonzalo
on behalf of the authors

■ Messages for Giovanni

Dear Giovanni,

I will not write about your groundbreaking contributions to the field of bioinformatics, as others have already done so. Instead, I want to express my personal feelings about the times we spent together.

I believe we first met at a Dagstuhl meeting, where we spent some time discussing the Lyndon factorization of genomes. The last time we met was in Pisa at DSB24, where I also had the pleasure of meeting your wife during dinner and discussing family matters. What impressed me was your curiosity, the depth of your personal relationships, and, at the same time, your being – as we say in Italian – *con i piedi per terra* (down to earth).

I did not have the privilege of working with you, but who knows – maybe in the near future. Ti auguro tutto il meglio!

Paola Bonizzoni

Dear Giovanni,

Happy 60th birthday! It is a great pleasure to contribute to this collection celebrating your remarkable career and enduring impact on the field of algorithms and data structures. Your pioneering work – most notably on the Burrows-Wheeler Transform and compressed indexing – has fundamentally shaped modern string processing, influencing not only theoretical computer science but also practical applications across bioinformatics and data compression.

I especially appreciate the opportunity to have co-authored with you. I have learned so much from your clarity of thought, depth of insight, and generous approach to collaboration. Working with you has been both intellectually enriching and personally rewarding.

Beyond your scholarly contributions, I've always valued your generosity with ideas, your commitment to the community, and your support for the next generation of researchers. You've been an inspiration to so many of us.

Wishing you continued success, good health, and many more years of discovery and collaboration.

Warm regards,

Christina Boucher

Dear Giovanni,

thinking back, I can't believe that we met, worked together and became friends because you commuted for so many years between Pisa and Alessandria: every week 250 km in each direction, braving the delays and glitches of the Italian rail service. I am glad you did it, though.

It was a privilege and a pleasure working with you. I admire your clear vision and the rigor of your scientific approach, and I appreciated the patience you showed me when you introduced me to a field that was utterly new to me. I am still very thankful that you involved me in such interesting research activities. But I also enjoyed our chats, sharing experiences about our families and everyday life at UPO.

Happy 60th birthday and all the best,

Lavinia Egidì

It's been over 25 years that Giovanni and I have been working together on research, teaching, and projects. It has been a long journey: sometimes closely connected, sometimes less so over the past 15 years, mainly due to my institutional responsibilities. Still, I always knew that if I ever needed Giovanni's support, he would be there without hesitation.

If I've to mention an anecdote of this long journey, I tell that I cannot forget that day, about 25 years ago, when I suggested the acronym FM for our compressed index. Giò, you never believed that my intention was to embed our initials in "our" index; but, now that you've reached the mature age of 60yo, it's time for the truth: I honestly hadn't realized FM were our initials, my intention was to valorize the peculiarities of the index: namely, "Full-text index in Minute space"!

Manzai, it has been an honor to collaborate with you, and to have you not only as a rigorous and scientifically outstanding colleague, but also as a friend. Wishing you continued success in your career, and above all, I strongly wish you all the best and happiness in your family life.

Paolo Ferragina

Dear Giovanni,

We have never collaborated, but your work has inspired my research for a long time. In particular your articles on suffix sorting, LCP-array computation, and the analysis of Lempel-Ziv, guided me during my PhD and Postdoc years (I openly admit that I hadn't understood the magic of the BWT at that time, and I'm still a bit puzzled today... sorry for writing this in an introduction to a book on the BWT). I hope you remain healthy and also active as a researcher in the many years to come, and that I can make up for the missed opportunities to collaborate.

Johannes Fischer

Twenty years ago I was a PhD student on holiday in Italy and thought to myself

“This is such a nice country, I should do an exchange here! The authors of that neat compressed-indexing paper had Italian-sounding names; I wonder, would one of them take me as an exchange student?”

Luckily for me, you agreed – which led to a year in Pisa, two years in Alessandria, thirty-something joint papers so far and an entire career designing compressed indexes. Grazie mille!

Happy 60th Birthday!

Travis Gagie

Field-shaping ideas in computer science are rare, and it is even rarer when they stem from elegant combinatorial constructions like the Burrows-Wheeler transform. It is yet more rare when such ideas turn out to be practical, and it is utterly rare when they become game changers in practical applications, like the FM-index did in bioinformatics.

Congratulations for your contributions and Happy birthday! Wishing you more rare pearls to discover.

Gregory Kucherov

Dear Giovanni,

Your paper describing the FM Index was the first complex academic paper that I really took the time to read and understand. It was not easy for me (I was a fresh graduate student at the time), but I was rewarded for the effort. Once I understood and adopted those ideas, the direction of my career and life changed forever!

I greatly admire the quality and impact of your work, as well as your kindness as a mentor and collaborator. Thank you.

Ben Langmead

Dear Giovanni,

Happy birthday!

Like many of the contributors of this volume, I teach some of your work on the BWT and the FM-index. Every time I introduce backward search, I tell the students “it’s magic!” Because it truly is.

Your work on the BWT and compressed indexes has had a profound influence on my own research in the past decade. But I also really value your kindness. I still remember, soon after I was hired in Verona, you came up to me at a conference, and said: Welcome to Italy!

I will never forget that. Thank you.

Zsuzsanna Lipták (Zsuzsa)

Dear Giovanni,

Happy 60th Birthday! It’s an honor to be part of this collection celebrating your remarkable career. It’s always a pleasure to work with you on some “BWT magic.” Thank you for your inspiring work, generosity, and constant support. I wish you and your family all the very best for the years ahead!

Warmly,

Felipe Louza

Dear Giovanni, happy 60th birthday!

It's been more than 20 years since I had been fascinated by the idea of self-indexes and presented my first decent contribution (or so I thought), in 2002 with a Lempel-Ziv index. It was difficult to get anyone's attention, however, because stronger indexes based on the CSA and the BWT had just appeared.

Then in 2004 you and Paolo kindly invited Veli and me to write a joint paper, which combined your FM-index with the wavelet tree. To me, that paper marks the starting point of the FM-index as a practical data structure; nobody seriously thinks today of another FM-index implementation. It was also crucial in my involvement in the field.

I always felt you could have written that 2004 paper yourselves, and see your invitation as a sign of generosity in bringing Veli and me into that recently-born area.

We have collaborated several times since then. It has always been a pleasure to work with you, both for your sharp scientific thinking and for your kind personality.

Happy birthday again!

Gonzalo Navarro

I am truly delighted to be part of the scientific community celebrating your remarkable achievements and distinguished career. You fully deserve this recognition. I am deeply grateful to have had the opportunity to work with you, learn from your expertise, and experience your unique, light, and very effective approach to research and collaboration.

Wishing you many more years filled with scientific discovery and meaningful connections.

Alberto Policriti

Dear Giovanni,

Happy 60th birthday! Your groundbreaking work on the Burrows-Wheeler Transform introduced me to the fascinating field of compressed indexing, which continues to inspire my research today. Not only the FM-index, but also the more recent Wheeler graphs have revolutionized the field of combinatorial pattern matching, unveiling new wonderful research avenues. It's truly remarkable that, 25 years later, you are still uncovering new properties of this powerful and magical combinatorial object! I'm honored to have collaborated with you on some of these recent developments and I wish you many more years of pioneering scientific discoveries.

Happy birthday again!

Ciao,

Nicola Prezza

Dear Giovanni,

I wish you all the best for your 60th birthday! I shared some time at the Max Planck Institute with Paolo Ferragina and closely followed since then the work of the "Italians". I introduce to my students the FM index as one of the true gems of Computer Science and I honestly think it is. I hope you continue contributing to the community and wish you all the best for the coming years as well as good health and happiness.

With kind regards,

Knut Reinert

I was introduced to the FM-index at the start of my PhD by my advisor, who called it the "BWT-magic." As a beginner, the backward search algorithm truly felt like magic. Like many others in this community, your work inspired me, guided my research, and shaped my career as a researcher and educator.

Thank you for your incredible contributions, and I wish you a very happy 60th birthday.

Sharma Thankachan

Congrats, Giovanni, on your 60th birthday!

We have enjoyed the opportunity to learn about the Burrows-Wheeler Transform and the FM-index over the years, and we are glad to contribute to your Festschrift by looking at new ways to improve the performance of the FM-Index in practice. The theories behind it and the compressed suffix array are quite beautiful and have been a great source of ideas for our research.

With special congratulations,

Jeffrey Vitter and Hongwei Huo

BWT and Combinatorics on Words

Gabriele Fici ✉ 

Department of Mathematics and Informatics, University of Palermo, Italy

Sabrina Mantaci ✉ 

Department of Mathematics and Informatics, University of Palermo, Italy

Antonio Restivo ✉ 

Department of Mathematics and Informatics, University of Palermo, Italy

Giuseppe Romana¹ ✉ 

Department of Mathematics and Informatics, University of Palermo, Italy

Giovanna Rosone ✉ 

Department of Computer Science, University of Pisa, Italy

Marinella Sciortino ✉ 

Department of Mathematics and Informatics, University of Palermo, Italy

Abstract

The Burrows-Wheeler Transform (BWT) is a reversible transformation on words (strings) introduced in 1994 in the context of data compression, which is a permutation of the characters in the word. Its clustering effect, i.e., the remarkable property of grouping identical characters (BWT runs) when they share common contexts, has made it a powerful tool for boosting compression performances and enabling efficient pattern searching in highly repetitive string collections.

In this chapter, we analyze the Burrows-Wheeler transform under the combinatorial point of view, and we survey known properties and connections with different aspects of combinatorics on words. In particular, we focus on the properties of words in relation to the number of their BWT runs. The value r , which counts the number of BWT runs, impacts both compression performance and indexing efficiency, and is considered a measure to evaluate the above-mentioned clustering effect and, consequently, the repetitiveness of a word. We give an overview of the results relating r to other combinatorial repetitiveness measures related to the factor complexity. The chapter also explores extremal cases of the clustering effect. Finally, some results on the sensitivity of the measure r are considered, where the effects of combinatorial operations are studied, such as reversal, edits, and the application of morphisms.

2012 ACM Subject Classification Theory of computation → Data compression; Mathematics of computing → Combinatorics; Mathematics of computing → Combinatorial algorithms

Keywords and phrases Burrows-Wheeler Transform, Combinatorics on Words, Clustering Effect, BWT Runs

Digital Object Identifier 10.4230/OASICS.Manzini.2025.1

Funding *Gabriele Fici*: Supported by MUR project PRIN 2022 APML – 20229BCXNW, funded by the European Union – Mission 4 “Education and Research” C2 - Investment 1.1. CUP Master_B53D23012910006.

Sabrina Mantaci: Supported by Project “ACoMPA” (CUP B73C24001050001) funded by the NextGeneration EU programme PNRR MUR M4 C2 Inv. 1.5 – Project ECS00000017 Tuscany Health Ecosystem (Spoke 6), CUP Master B63C22000680007.

Giuseppe Romana: Supported by the MUR PRIN Project “PINC, Pangenome INformatiCs: from Theory to Applications” (Grant No. 2022YRB97K), funded by Next Generation EU PNRR M4 C2, Inv. 1.1. and by the INdAM - GNCS Project CUP_E53C24001950001.

¹ Corresponding author.



Giovanna Rosone: Partially supported by PNRR - M4C2 - Investimento 1.5, Ecosistema dell’Innovazione ECS00000017 - “THE - Tuscany Health Ecosystem” - Spoke 6 “Precision medicine & personalized healthcare”, funded by the European Commission under the NextGeneration EU program, by MUR PRIN 2022 YRB97K “PINC” funded by Next Generation EU PNRR M4 C2 Inv. 1.1, and by the INdAM - GNCS Project CUP_E53C24001950001.

Marinella Sciortino: Supported by Project “ACoMPA” (CUP B73C24001050001) funded by the NextGeneration EU programme PNRR MUR M4 C2 Inv. 1.5 – Project ECS00000017 Tuscany Health Ecosystem (Spoke 6), CUP Master B63C22000680007, by MUR PRIN 2022 YRB97K “PINC” funded by Next Generation EU PNRR M4 C2 Inv. 1.1, and by the INdAM - GNCS Project CUP_E53C24001950001.

1 Introduction

The Burrows-Wheeler transform (BWT) is a fundamental algorithm introduced in 1994 by Michael Burrows and David Wheeler in the field of data compression [12]. Although more than 30 years have passed since its introduction, the BWT remains at the core of many tools used in bioinformatics for analyzing biological sequences (see [40, 45, 41]) and, in particular, constitutes the backbone of popular efficient compressed indexing schemes [21, 25].

The *clustering effect* of the BWT, i.e., the remarkable property of grouping identical characters when they share common contexts, has made it a powerful tool for boosting compression performances [54, 20] and enabling very efficient pattern searching in highly repetitive sequence collections [25, 38].

The BWT is a reversible transformation that lexicographically sorts all cyclic rotations of the input text w and produces a permutation of the characters of w , denoted by $bwt(w)$, by taking the last character of these sorted rotations. From a combinatorial point of view, its simplicity combined with its effectiveness has made the BWT a fascinating study object [66]. The study of the combinatorial properties of the BWT not only helps in understanding its theoretical foundations but also in exploring its potential as a new combinatorial tool for analyzing the structural properties of words. A pioneering work in this area was done by Mantaci et al. [53], who showed that the BWT can be used to characterize standard Sturmian words, an infinite family of binary words widely studied in combinatorics on words [48].

This chapter aims to highlight strong connections between the BWT and combinatorics on words. Most combinatorial studies on the BWT have focused on the properties of words produced by the transformation and their relation to its clustering effect. Here, we provide an overview of these results, showing the role of the BWT as an effective new tool for assessing the complexity of combinatorial objects.

The effect of the BWT clustering on a word w can be evaluated using the measure $r(w)$, which denotes the number of clusters (also called BWT runs) produced by the BWT applied to w , i.e., the number of runs of consecutive identical characters in $bwt(w)$. The value $r(w)$ is also used as a measure of the repetitiveness of the word w . In fact, the more repetitive a word is, the more likely it is that $bwt(w)$ contains long runs of consecutive identical characters. This property makes w highly compressible using BWT-based compression algorithms. Furthermore, the performance in terms of both space and time of text compressors and compressed indexing data structures applied on a text w can be evaluated by using $r(w)$ [24, 25]. Exploring the relationships between r and the combinatorial properties of the words, as well as its sensitivity to some combinatorial transformations on words, can also have implications for the analysis and definition of BWT-based compressed indexing schemes.

In Section 3, we provide an overview of the main combinatorial properties of the BWT and relate them to some well-known permutations that can be associated with the words produced by the BWT as output (also called BWT images) [46].

Section 4 is focused on results that relate the number of BWT runs to the number of maximal runs in the input word [52, 51]. On the one hand, this provides a measure of the effectiveness of the BWT; on the other hand, it allows a deeper investigation into the combinatorial properties of words for which the BWT is more effective. In particular, in [11, 23] this relationship has been studied for morphic words, which are infinite families of binary words obtained by iteratively applying a morphism starting from a single character. Morphisms are a well-known combinatorial tool for constructing highly repetitive families of words. Other interesting researches connect the measure r with other repetitiveness measures used in combinatorics on words, such as *factor complexity*, which counts the number of distinct factors that appear in a word for each length. Periodic words, for example, which are obtained by iterating the same factor, have a constant factor complexity. It is possible to define upper and lower bounds for the measure r as a function of the measure δ , the supremum of the normalized factor complexity [33].

Much attention has been given to the combinatorial characterization of the so-called *clustering words* under the BWT. This is the family of words for which r assumes its minimum value. These words are the most compressible using BWT-based compressors. This topic is the focus of Section 5. The case of binary words has been studied in [53]. For larger alphabets, the relationship between clustering words and the trajectories of interval exchange transformations has been investigated in [19]. The combinatorial properties of a subclass of these words, called perfectly clustering words, for which the BWT produces sequences of lexicographically decreasing maximal runs, have been extensively studied. Only recently, the first combinatorial characterization of perfectly clustering words has been established [43]. Similarly, it can be interesting to explore the combinatorial properties of words for which the number of BWT runs is maximal. However, no known combinatorial characterization of such words exists. De Bruijn words, on the other hand, are examples of infinite families of words for which the number of BWT runs is close to the maximum [32]. This result has been used in [47] to define a BWT-based algorithm to generate random de Bruijn words.

In Section 6, we present results showing how the number of BWT runs is affected by some combinatorial operations on words. Sensitivity analysis is crucial for understanding the stability of compression-based repetitiveness measures [1]. It has been shown that, unlike other repetitiveness measures, r is not invariant under reversal, with known upper and lower bounds on the ratio between r of a word and of its reverse [29]. The impact of edit operations, such as insertion, deletion, and substitution, is also analyzed, establishing both upper and lower bounds for the growth in the number of BWT runs after edits [1, 30, 31]. Additionally, the section explores results on the behavior of r under morphisms, proving that injective binary morphisms always increase or preserve r , with the well-known family of *Sturmian morphisms* being the only ones that preserve it [22].

2 Preliminaries

Let $\Sigma = \{a_1, a_2, \dots, a_\sigma\}$, with $a_1 < a_2 < \dots < a_\sigma$, be a finite ordered alphabet. The order on Σ induces the standard lexicographic order $\leq_{lex}$ on the set Σ^* of words over Σ .

Given a word $w = w_1w_2 \dots w_n \in \Sigma^*$ with each $w_i \in \Sigma$, the length n of w is denoted by $|w|$. We let ε denote the empty word, the only word of length 0. The set of words of length n over Σ is denoted by Σ^n . With $alph(w)$ we denote the subset of Σ of letters occurring in w .

We let $|w|_{a_i}$ denote the number of occurrences in w of the letter a_i . The *Parikh vector* P_w of a word w is the vector $[n_1, \dots, n_\sigma]$, where n_i is the number of occurrences of a_i in w , i.e., for each $a_i \in \Sigma$, $P_w[i] = |w|_{a_i}$.

A word v is said to be a *factor* of a word w if it occurs as a subword in w , i.e., $v = w[i, j] = w_i \cdots w_j$, for some $1 \leq i \leq j \leq n$. The set of all factors of a word w is denoted by $\text{Fact}(w)$. A factor of type $w[1, j]$ is called a *prefix*, while a factor of type $w[i, n]$ is called a *suffix*. The longest proper factor of w that is both a prefix and a suffix is called the *longest border* of w . Let $\text{Fact}(w)$ be the set of factors of w . A factor u of w is *right* (resp. *left*) *special* in w if there exist $a, b \in \Sigma$, $a \neq b$, such that $ua, ub \in \text{Fact}(w)$ (resp. $au, bu \in \text{Fact}(w)$). A factor u of w is *bispecial* if it is both right and left special.

Two words $x, y \in \Sigma^*$ are *conjugates* (or x is a *cyclic rotation* of y), if $x = uv$ and $y = vu$, for some $u, v \in \Sigma^*$. Conjugacy between words is an equivalence relation over Σ^* . A word z is called a *circular factor* of w if it is a factor of some conjugate of w . A word w is *primitive* if $w \neq u^h$ for every nonempty word u and integer $h > 1$. The cyclic rotations of a primitive word are all distinct. The (*primitive*) *root* of a word w , denoted by $\sqrt{w}$, is the shortest word such that $w = (\sqrt{w})^h$, for some integer $h \geq 1$. So, a word is primitive if and only if it coincides with its root. A *Lyndon word* is a primitive word that is smaller than all of its proper conjugates (or, equivalently, suffixes).

The *reversal* of the word $w = w_1 \cdots w_n$ is the word $\tilde{w} = w_n \cdots w_1$. If $w = \tilde{w}$, then w is called a *palindrome*. The empty word is also assumed to be a palindrome. A word w has the *two-palindrome property* if it can be written as uv where u and v are palindromes. This is equivalent to the fact that w is *symmetric*, i.e., w is a conjugate of its reversal $\tilde{w}$.

A *maximal run* (or *cluster*) u in a word w is the factor $u = w[i, j]$ (with $i < j$) where $w_i = w_{i+1} = \cdots = w_j$, and $w_{i-1} \neq w_i$, $w_j \neq w_{j+1}$ if w_{i-1} and/or w_{j+1} exist. Informally, a maximal run is a maximal factor of consecutive occurrences of the same letter.

Morphisms

Given two finite alphabets Σ and Γ , a *morphism* μ is a map from Σ^* to Γ^* such that $\mu(uv) = \mu(u)\mu(v)$ for all words $u, v \in \Sigma^*$. Therefore, a morphism μ can be defined by specifying its action on the letters of Σ . A morphism μ is *prolongable* on a letter $a \in \Sigma$ if a is a prefix of $\mu(a)$. Very well-known examples of morphisms on the binary alphabet $\{a, b\}$ are the *Fibonacci morphism* φ , which maps a to ab and b to a , and the *Thue–Morse morphism* τ , which maps a to ab and b to ba . Fibonacci and Thue–Morse morphisms can be used to define two well-known infinite sequences, obtained by recursively applying the morphisms starting from a : the *Fibonacci infinite word* $f = abaababaabaab \cdots$ and the *Thue–Morse word* $t = abbabaabbaabba \cdots$, respectively.

Sturmian and Episturmian Words

An infinite word over an alphabet Σ is an infinite sequence of letters from Σ . An infinite word w is *ultimately periodic* if there exist words u and v such that $w = uvvvv \cdots$; otherwise it is *aperiodic*.

An important class of aperiodic infinite words is the class of *Sturmian words*. Sturmian words appear in many contexts and have many equivalent definitions, witnessing the richness of their combinatorial properties.

A (finite or infinite) word is *balanced* if, for any pair of its factors u, v with $|u| = |v|$, and for every $a \in \Sigma$, $|u|_a - |v|_a \leq 1$. A finite word is *circularly balanced* if all its conjugates are balanced. *Sturmian words* are infinite words over a two-letter alphabet that are balanced and aperiodic. Note that, for instance, the Fibonacci infinite word f is balanced and aperiodic,

therefore, it is Sturmian, whereas the Thue–Morse word t is not, since it is aperiodic but not balanced (it contains the factors aa and bb). Equivalently, an infinite binary word is Sturmian if it has exactly one left special factor of length n for each $n \geq 0$ [48].

Within the family of Sturmian words, the *standard* (or *characteristic*) Sturmian words can be considered as representative, in the sense that for every Sturmian word there is exactly one standard Sturmian word with the same set of factors. A Sturmian word is called standard if every prefix is left special (and is therefore the only left special factor of that length). The following equivalent definition of standard Sturmian word highlights its structure. Let $d_1, d_2, \dots, d_n, \dots$ be a sequence of integers, with $d_1 \geq 0$ and $d_i > 0$ for $i > 1$. Consider the following sequence of words $\{s_n\}_{n \geq 0} : s_0 = b, s_1 = a$, and $s_{n+1} = s_n^{d_n} s_{n-1}$ for $n \geq 1$. The sequence $\{s_n\}_{n \geq 0}$ converges to a standard Sturmian word. The sequence $\{s_n\}_{n \geq 0}$ is called the *standard sequence* of s and $(d_1, d_2, \dots, d_n, \dots)$ is the *directive sequence* of s . Each finite binary word s_n appearing in some standard sequence is called *standard word* of order n . For example, the Fibonacci infinite word is a standard Sturmian word, with directive sequence $d_i = 1$ for every i , and the standard sequence of the Fibonacci infinite word is the sequence of *Fibonacci words*, which can also be defined by: $f_0 = b$, $f_1 = a$, and $f_n = f_{n-1}f_{n-2}$ for every $n \geq 2$.

A binary word is a *Christoffel word* if it is the lexicographically minimal (Lyndon) conjugate of a standard word (note that standard words are always primitive).

Infinite Sturmian words have many combinatorial characterizations. There exist several generalizations of Sturmian words to alphabets of cardinality larger than 2. Unfortunately, these generalizations are no longer equivalent, leading to distinct families of infinite words. Here, we consider some of these definitions, which generalize, from a specific perspective, the notion of the Sturmian word.

An infinite word w over an alphabet Σ of cardinality at least 2 is *episturmian* [18] if $\text{Fact}(w)$ is closed under reversal and w has at most one left special factor (or, equivalently, at most one right special factor) of each length. Sturmian words are binary episturmian words. However, episturmian words may not be aperiodic, and may not be balanced.

An infinite word w is *standard episturmian*, or *epistandard*, if all of its left special factors appear as prefixes in w . The *Rauzy rules*, described below, allow one to recursively construct, similarly to standard Sturmian words, any epistandard word.

Let $\Sigma = \{a_1, a_2, \dots, a_\sigma\}$ be an alphabet of cardinality $\sigma \geq 2$. The sequence $(R_n)_{n \geq 0}$ of Rauzy tuples $R_n = (A_n^{(1)}, A_n^{(2)}, \dots, A_n^{(\sigma)})$ is recursively defined by:

- $R_0 = (a_1, a_2, \dots, a_\sigma)$;
- R_{n+1} is obtained from R_n by applying one of the Rauzy rules, labeled with $1, 2, \dots, k$, with the rule $i \in \{1, 2, \dots, \sigma\}$ being defined by:
 - $A_{n+1}^{(i)} = A_n^{(i)}$,
 - $A_{n+1}^{(j)} = A_n^{(i)} A_n^{(j)}$ for $j \neq i$.

A word $w \in \Sigma^*$ is a *finite epistandard word* if it is an element of a tuple R_n , for some $n \geq 1$. Note that if $\sigma = 2$, this definition coincides with the one defining standard words. In this sense, epistandard words are a generalization of standard words.

3 BWT Matrix and Permutations

The *Burrows–Wheeler matrix* (BWT matrix) of a word w is the matrix $\mathcal{B}_w$ whose rows are the conjugates of w in ascending lexicographic order. Let us denote by F and L , respectively, the first and the last column of the BWT matrix of a word w . We have that F is the sequence of sorted letters of w , i.e., $F = a_1^{n_1} a_2^{n_2} \dots a_\sigma^{n_\sigma}$, where $(n_1, n_2, \dots, n_\sigma)$ is the Parikh vector of w ; L , instead, is the *Burrows–Wheeler Transform* of w and is denoted by $\text{bwt}(w)$.

► **Remark 1.** By definition, the Burrows–Wheeler transform is the same for all words in the same conjugacy class. Moreover, the Burrows–Wheeler transform is an injective map on the set of conjugacy classes of words: two words u and v are conjugates if and only if $bwt(u) = bwt(v)$.

A word is a *BWT image* if it is the Burrows–Wheeler Transform of some word. Not every word is a BWT image. The Burrows–Wheeler Transform is therefore a (non-surjective) map from the set of conjugacy classes of words to the set of words.

► **Proposition 2** ([53, Proposition 2]). *Let η_n be the morphism defined by $\eta_n(a_i) = a_i^n$ for every $a_i \in \Sigma$. If $w = v^n$, then $bwt(w) = \eta_n(bwt(v))$.*

The *standard permutation* of a word $u = u_1u_2 \cdots u_n$ over Σ is the permutation π_u of $\{1, 2, \dots, n\}$ such that $\pi_u(i) < \pi_u(j)$ if and only if $u_i < u_j$ or $u_i = u_j$ and $i < j$. In other words, π_u orders distinct letters of u lexicographically, and equal letters by occurrence order.

A word u is a BWT image of a primitive word if and only if π_u is a length- n cycle, in which case the standard permutation of u is also called *LF-mapping*. More generally, the problem of characterizing BWT images has been faced in [46], where the authors prove two necessary and sufficient conditions. The first one describes the words that are BWT images, while the second one explains which words can be converted to BWT images using a natural “pumping” procedure. Both conditions can be checked in linear time. In particular, the following theorem is proved.

► **Theorem 3** ([46, Theorem 1]). *Given a word $u \in \Sigma^*$, $u = bwt(w)$ for some $w \in \Sigma^*$ if and only if the number of cycles of the standard permutation π_u equals the greatest common divisor of the lengths of maximal runs in u .*

By using the theorem one can deduce that the word $u = bccaaab = b^1c^2a^3b^1$ is not a BWT image because the π_u decomposes into the 2 cycles (1 4) and (2 6 3 7 5), but the lengths 1, 2, 3, 1 of the maximal runs in u are coprime.

In the paper [46], the authors also characterize the words $v = c_1 \cdots c_h$ that can be transformed to BWT images by a sort of “pumping”: each letter c_i can be replaced by $c_i^{p_i}$ for an arbitrary positive number p_i . In particular, they prove the following theorem that uses the notion of global ascent² of a word.

► **Theorem 4** ([46, Theorem 2]). *Let $v = c_1 \cdots c_h$ be an arbitrary word with at least two different letters. A BWT image of the form $c_1^{p_1} c_2^{p_2} \cdots c_h^{p_h}$, where $p_i > 0$ for all i , exists if and only if v has no global ascents.*

Using the algorithm provided in their paper, starting from the word $v = dacba$, one can construct the word $w = daccbaa$ that is the *bwt* image of *aacbcad*. An interesting open question is that of constructing the shortest possible *bwt* image with a given order of letters.

The *inverse standard permutation* π_u^{-1} of a word u , i.e., the inverse permutation of the standard permutation of u , can be obtained by listing in left-to-right order the positions of a_1 in u , then the positions of a_2 , and so on. The inverse standard permutation of a BWT image is also called *FL-mapping*. If w is a Lyndon word, the inverse standard permutation of $bwt(w)$, in cycle form, can also be obtained by reading the Inverse Suffix Array (ISA) of w as a cycle [16, 59]. Recall that the ISA of a word w is the array whose i th entry is the ranking of the suffix of w starting at position i .

In the rest of this chapter, unless otherwise specified, we will deal with primitive words only.

² A word v has a *global ascent* if $v = xy$, where x, y are nonempty and the biggest letter of x is less than or equal to the smallest letter of y .

If w is a Lyndon word and u is its BWT, then w can be uniquely reconstructed from u . In fact, we have: $w_n = u_1$ and, for every $j = 1, \dots, n-1$, $w_{n-j} = u_{\pi_u^j(1)}$. In other words, if $\pi_u = (j_1 \ j_2 \ \dots \ j_n)$ is the cycle corresponding to the standard permutation of u , with $j_1 = 1$, then $w = u_{j_n} u_{j_{n-1}} \dots u_{j_1}$. If w is a primitive word that is not Lyndon, it is sufficient to memorize its index in the BWT matrix to reconstruct it uniquely from its BWT.

► **Remark 5.** The reversibility of $bwt(w)$ is achieved thanks to the two known fundamental properties:

- for all $i = 1, \dots, n$, the letter $F[i]$ circularly follows the letter $L[i]$ in the word w ;
- for each $a \in \Sigma$ and for each k , the k -th occurrence of a in L corresponds to the k -th occurrence of a in F .

► **Example 6.** The BWT matrix $\mathcal{B}_w$ of the word $w = aabacacb$ is shown in Figure 1. The word $u = babccaaa$ is a BWT image since $u = bwt(w)$. The standard permutation of u is $\pi_u = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 1 & 6 & 7 & 8 & 2 & 3 & 4 \end{pmatrix}$, or, in cycle form, $\pi_u = (1 \ 5 \ 8 \ 4 \ 7 \ 3 \ 6 \ 2)$.

The inverse standard permutation of u is $\pi_u^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 6 & 7 & 8 & 1 & 3 & 4 & 5 \end{pmatrix} = (1 \ 2 \ 6 \ 3 \ 7 \ 4 \ 8 \ 5)$ (notice that, as cycle, it is the reversal of π_u), and $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 6 & 3 & 7 & 4 & 8 & 5 \end{pmatrix} = ISA(aabacacb)$.

$$\begin{pmatrix} a & a & b & a & c & a & c & b \\ a & b & a & c & a & c & b & a \\ a & c & a & c & b & a & a & b \\ a & c & b & a & a & b & a & c \\ b & a & a & b & a & c & a & c \\ b & a & c & a & c & b & a & a \\ c & a & c & b & a & a & b & a \\ c & b & a & a & b & a & c & a \end{pmatrix}$$

■ **Figure 1** BWT-matrix $\mathcal{B}_w$ of the word $w = aabacacb$.

4 Combinatorics of BWT Runs

One of the key properties of the BWT is that it produces a word that is typically more compressible than the word given as input. Such compressibility arises from the fact that sorting the conjugates of a given word w places those with equal or similar prefixes consecutively in the matrix $\mathcal{B}_w$, so the characters preceding those similar contexts are grouped in $bwt(w)$. As a result, repetitions in the input word w tend to produce long runs of equal characters in $bwt(w)$ (so-called *clustering effect*). For this reason, a commonly used measure to evaluate the repetitiveness of a text, and consequently its compressibility via the BWT, is the number of BWT runs, i.e., the number of maximal runs the BWT applied produces, after being applied to the text. More formally, given a word w , the measure $r = r(w)$ denotes the number of the *BWT runs* and it is defined as

$$r(w) = runs(bwt(w)),$$

where $runs(v)$ denotes the number of maximal runs of a word v . The more repetitive a word w is, the lower the number r of its BWT runs.

From a combinatorial point of view, the parameter r has been studied to obtain more information about the combinatorial complexity and repetitiveness of a word from the number of BWT runs it produces. The following proposition, which immediately follows from Proposition 2, shows that the number of BWT runs of a periodic word is exactly the number of BWT runs of its primitive root.

► **Proposition 7.** *Let w be a word and $n > 0$ an integer. Then, $r(w) = r(w^n)$.*

4.1 BWT Runs and Clustering Effect

A first combinatorial approach to the BWT clustering effect has been investigated in [51, 52], where the so-called *BWT-clustering ratio* $\rho(w)$ of a word w has been studied, defined as

$$\rho(w) = \frac{r(w)}{\text{runs}(w)}.$$

If two words u and v are conjugates, then $|\text{runs}(u) - \text{runs}(v)| \leq 1$, and, within a conjugacy class, a power of a Lyndon word is one of the conjugates with the fewest maximal runs. Analyzing the clustering effect of the BWT from a combinatorial perspective is equivalent to studying how the number of BWT runs can grow with respect to the number of maximal runs in the input word. Based on Remark 1 and Proposition 7, we restrict our attention to Lyndon words, and the following theorem shows that the number of BWT runs can be at most twice the number of maximal runs in the input word.

► **Theorem 8** ([52, Theorem 3.3]). *Let w be a Lyndon word over a finite alphabet Σ . Then:*

$$|\text{alph}(w)| \leq r(w) \leq 2 \cdot \text{runs}(w).$$

This is equivalent to stating that, for a Lyndon word w , $0 < \rho(w) \leq 2$. The following result shows that all positive rational numbers less than 2 are possible BWT-clustering ratios. Note that the proof in [51] explicitly shows how to construct a Lyndon word with a given BWT-clustering ratio.

► **Theorem 9** ([51, Theorem 2]). *For every $q \in \mathbb{Q} \cap (0, 2]$, there exists a Lyndon word $w \in \{a, b\}^*$ having $\rho(w) = q$.*

The BWT-clustering ratio has also been analyzed for infinite families of highly repetitive words, specifically those generated by a morphism. In particular, in [11], the value of the measure r and the combinatorial properties of the BWT on finite words generated by compositions of the Thue–Morse morphism and some Sturmian morphisms have been studied. It has been proved that for such words the BWT-clustering ratio is significantly less than 1. This analysis has been generalized in [23], where all binary prolongable morphisms have been considered. The following result shows that all finite words generated by iterating the application of a morphism an arbitrary number of times, starting from a letter of the alphabet, are highly compressible after applying the BWT.

► **Theorem 10** ([23, Theorem 28]). *Let μ be a binary morphism prolongable on a letter a , such that $\mu(a) \neq ab^m$ and $\mu(b) \neq b^n$, for every $m \geq 1$, $n \geq 1$. Then*

$$\lim_{i \rightarrow \infty} \rho(\mu^i(a)) = 0.$$

4.2 BWT Runs and Factor Complexity

Great attention has been given to the relationship between the measure r and other complexity measures used in combinatorics on words, such as the factor complexity. Understanding these connections can provide deeper insights into compressibility and the structural properties of words. In the next sections, some results on the number of BWT runs for words with minimal and maximal factor complexity are reported.

4.2.1 Insights from Special Factors

The factor complexity of a word is closely related to the number of its special factors [14, Proposition 4.5.3]³, and provides important information about the structural complexity of the word itself. In this subsection, we present some results on the relationship between BWT runs and some measures related to special factors.

Given an alphabet Σ , a word $w \in \Sigma^*$ and a factor $u \in \text{Fact}(w)$, the *right* (resp. *left*) *extensions* of u in w are defined as $e_r(u) = |\{ua \mid a \in \Sigma, ua \in \text{Fact}(w)\}| - 1$ (resp. $e_\ell(u) = |\{au \mid a \in \Sigma, au \in \text{Fact}(w)\}| - 1$).

So, u is right (resp. left) special if $e_r(u) \geq 1$ (resp. $e_\ell(u) \geq 1$), while it is bispecial if and only if $e_r(u), e_\ell(u) \geq 1$. A bispecial factor u of w is called *weakly bispecial* if $|\text{Fact}(w) \cap \Sigma u \Sigma| = \max\{e_\ell(u), e_r(u)\} + 1$. We denote the set of (resp. *weakly*) *bispecial factors* of w by $BS(w)$ (resp. $WBS(w)$).

With a slight abuse of notation, if $[w]$ is the conjugacy class of w , $\text{Fact}([w])$ denotes the set of circular factors of w , i.e., the factors of all words in $[w]$, while $e_r(u)$ (resp. $e_\ell(u)$) are the right (resp. left) extensions of u in all words of the conjugacy class $[w]$. Analogously, $BS([w])$ (resp. $WBS([w])$) denotes the set of bispecial (resp. weakly bispecial) factors in $\text{Fact}([w])$.

The following lemma shows the relationship between the number of (weakly) bispecial factors in $[w]$ and the number of BWT runs of any word in $[w]$.

► **Lemma 11** ([23, Lemma 13], [65, Lemma 19]). *Let $w \in \Sigma^*$ be a finite word. Then,*

$$\left(\sum_{u \in WBS([w])} \min\{e_\ell(u), e_r(u)\} \right) + 1 \leq r(w) \leq \left(\sum_{u \in BS([w])} e_r(u) \right) + 1.$$

4.2.2 BWT Runs and δ Measure

In the context of data compression, recent research has led to the design of runlength-compressed BWT-based indexing data structures whose space complexity is bounded by a function of r . The most relevant one is the r -index [24, 25], a fully compressed index of size $O(r)$ that allows one to locate the *occ* occurrences of a factor in a text of length n in $O(\text{occ} \log \log n)$ time. These structures are very effective for indexing collections of highly repetitive texts. Other repetitiveness measures have been used to evaluate the performance of compressed indexing data structures for highly repetitive string collections. A survey of these measures can be found in [56]. Examples of such measures include, but are not limited to, the number z of factors in LZ77 factorization [44, 36], the number g of rules in the smallest context-free grammar generating the word [35, 67], the size b of the smallest bidirectional

³ In [14] the study of the factor complexity of infinite words is considered; however, we can extend their results by observing that the circular factor complexity of a finite word w is equivalent to the factor complexity of the infinite periodic word $w^\omega = www \dots$.

macroscheme [71], the size (number of nodes plus number of edges) e of the Compact Directed Acyclic Word Graph [9] and the size γ of the smallest string attractor [34, 49]; some of these measures have been also investigated for well-known families of words [15, 39, 2, 68, 62, 55, 13]. All these measures are lower-bounded by the measure δ , which is defined by using the factor complexity:

$$\delta(w) = \max_{1 \leq i \leq |w|} \left\{ \frac{|Fact(w) \cap \Sigma^i|}{i} \right\}.$$

Here, we are interested in deriving an upper bound for r in terms of δ . In [33], an upper bound has been proved when the measure r is computed for words with an end-of-string symbol appended. Note that, as shown in [31], the presence of the end-of-word symbol can significantly affect the value of r^4 . However, in the next theorem, we show that an analogous bound established in [33] also holds for r when a generic word is considered.

► **Theorem 12.** *Every word $w \in \Sigma^n$ satisfies*

$$r(w) = O \left(\delta(\sqrt{w}) \log \delta(\sqrt{w}) \cdot \max \left\{ 1, \log \frac{|\sqrt{w}|}{\delta(\sqrt{w}) \log \delta(\sqrt{w})} \right\} \right)$$

Proof. By [33, Theorem 3.7] it follows that

$$r(w\$) = O \left(\delta(w) \log \delta(w) \cdot \max \left\{ 1, \log \frac{n}{\delta(w) \log \delta(w)} \right\} \right),$$

where $\$ \notin \Sigma$ is an end-of-word symbol smaller than any symbol from Σ . Let us suppose w is primitive. By [65, Lemma 16], for every word $w = uv$ such that vu is Lyndon it holds that $r(vu\$) - 2 \leq r(w) \leq r(vu\$) - 1$, while by [65, Lemma 17] we have that $|\delta(uv) - \delta(vu)| < 1$, and clearly $\delta(vu\$) - 1 \leq \delta(vu) < \delta(vu\$)$. Hence, for every primitive word $w = uv$ such that vu is Lyndon, it holds

$$\begin{aligned} r(vu) &= r(uv) = \Theta(r(uv\$)) \\ &= O \left(\delta(uv\$) \log \delta(uv\$) \cdot \max \left\{ 1, \log \frac{|uv\$|}{\delta(uv\$) \log \delta(uv\$)} \right\} \right) \\ &= O \left(\delta(vu) \log \delta(vu) \cdot \max \left\{ 1, \log \frac{n}{\delta(vu) \log \delta(vu)} \right\} \right). \end{aligned}$$

On the other hand, if $w = (uv)^h$ for some $h > 1$ such that vu is Lyndon, by Proposition 7 it holds that $r(w) = r(\sqrt{w}) = r(vu)$, and the thesis follows. ◀

5 Extremal Cases in the Number of BWT Runs

In the previous section, we have shown that $r(w)$, the number of maximal runs of the BWT of a word w , can range from the size of the alphabet to, at most, the double of the number of runs of the original word w . In this section, we focus on these special extreme cases, namely, the one where $r(w) = |\Sigma|$ and the one where $r(w) = 2 \cdot \text{runs}(w)$.

These extreme cases are important because they could have implications for the compressibility of the input text. Indeed, when the number of maximal runs is minimized, it is possible to achieve the optimal performance of the BWT-based compressor by using techniques such

⁴ There can be a multiplicative $\Theta(\log n)$ [31, Propositions 43], or an additive $\Theta(\sqrt{n})$ [31, Propositions 44] factor difference between $r(w)$ and $r(w\$)$, where $n = |w|$.

as run-length encoding (RLE) and move-to-front (MTF) [7]. In the opposite case, using a BWT-based compressor may not be effective. To understand the “clustering effect” from a combinatorial perspective, several studies have analyzed the structural properties of words that produce the maximum or minimum number of maximal runs when a text is transformed using the BWT.

5.1 Minimum Number of BWT Runs

We first consider the case of the minimum number of BWT runs, i.e., the size of the alphabet. To introduce the results, we first need to recall some definitions and give some notation.

A primitive word w over a totally ordered alphabet $\Sigma = \{a_1, a_2, \dots, a_\sigma\}$, with $a_1 < a_2 < \dots < a_\sigma$ is a *clustering word* if in $bwt(w)$ all equal letters appear in the same cluster. This means that $bwt(w)$ has exactly $|\Sigma|$ clusters. Up to now, no combinatorial or structural characterization for all clustering words has been established. However, in [19] it has been proved that clustering words are exactly those words w such that ww occurs in a trajectory of an interval exchange transformation.

► **Example 13.** The word $w = bcabcbcabcbca$ is a clustering word. Indeed, $bwt(w) = c^5a^3b^5$.

Perfectly clustering words, widely studied in the literature, are a special case of clustering words.

A *perfectly clustering word* is a word over a totally ordered alphabet $\Sigma = \{a_1, a_2, \dots, a_\sigma\}$, with $a_1 < a_2 < \dots < a_\sigma$, such that $bwt(w) = a_\sigma^{n_\sigma} \dots a_2^{n_2} a_1^{n_1}$ for some non-negative integers $n_1, n_2, \dots, n_\sigma$.

► **Example 14.** The word $v = cacbcac$ is perfectly clustering, since $bwt(v) = c^4ba^2$. Note also that the word $w = bcabcbcabcbca$ in Example 13, although clustering, is not perfectly clustering.

Perfectly clustering words have first been studied and characterized for binary alphabets in [53]. The main result is formalized as follows:

► **Theorem 15** ([53, Corollary 11]). *Given a word $w \in \{a, b\}^*$, the following conditions are equivalent:*

1. $bwt(w) = b^p a^q$, with $p, q \geq 0$, i.e., w is perfectly clustering;
2. w is a circularly balanced word;
3. w is a conjugate of a power of a standard word.

► **Example 16.** Let $f_6 = abaababaabaab$ be the Fibonacci word of order 6. One can easily verify that $bwt(f_6) = b^5a^8$. Recall that Fibonacci words are special cases of standard words.

It is an easy remark that, when $|alph(w)| > 1$, the first letter of $bwt(w)$ cannot be the least letter in the alphabet, so the only clustering words over a binary alphabet are perfectly clustering. Then the powers of standard words and their conjugates are the only possible clustering words in the binary case.

Having established a characterization of perfectly clustering words for the two-letter alphabet, the natural next step is to explore larger alphabets. This extension, however, is not straightforward. Unlike the binary case, the defining properties of standard words for larger alphabets no longer coincide, leading to different word sets. Consequently, finding a complete characterization of perfectly clustering words that extends Theorem 15 to larger alphabets has been complex and lasted nearly twenty years, marked by a series of insightful intermediate achievements.

The first of these results, due to Simpson and Puglisi [69], is stated in the next theorem and provides a strategy to construct primitive perfect clustering words over the ternary alphabet $\{a, b, c\}$. It starts from the set of all standard words over the binary alphabets $\{b, c\}$ and $\{a, c\}$, and considers the minimal set T including such words that is closed under the three ternary morphisms μ_1 , μ_2 and μ_3 , where μ_1 maps a to a , b to ac , c to ac ; μ_2 maps a to c , b to b , c to a , and $\mu_3(x)$ is obtained from x by inserting a b in the middle of each occurrence of the factors aa , cc , ba and bc , by considering x as a circular word.

► **Theorem 17** ([69, Theorem 3.11]). *All perfectly clustering ternary words are precisely the conjugates of the words in the set T and the powers of these conjugates.*

Note that no extension of such construction for larger alphabets is known. In the same paper [69], the following characterization of the Parikh vectors of clustering words is provided. This result has been independently given in [58].

► **Theorem 18** ([58, Lemma 1], [69, Theorems 3.9 and 3.12]). *Let w be a primitive ternary word with Parikh vector $P_w = [\alpha, \beta, \gamma]$. The word w is perfectly clustering if and only if $\gcd(\alpha, \beta, \gamma) = \gcd(\alpha + \beta, \beta + \gamma) = 1$.*

► **Example 19.** The word $w = cacacacbbbbbb$ is perfectly clustering, since $bwt(w) = c^4b^6a^3$. Indeed its Parikh vector $P_w = [3, 6, 4]$ is such that $\gcd(3, 6, 4) = 1$ and $\gcd(3 + 6, 6 + 4) = 1$.

Extending Theorem 18 to the case of larger alphabets is still an open problem.

Another characterization of perfectly clustering ternary words in terms of the Three-distance theorem has been recently given in [8].

Perfectly clustering words have also been characterized by looking at the corresponding BWT matrix. The following result was obtained independently in [69, 63].

► **Theorem 20** ([69, Theorem 4.3], [63, Theorem 4.2]). *A word w is perfectly clustering if and only if $\mathcal{B}_w = R$, where R is the matrix obtained from the BWT-matrix $\mathcal{B}_w$ by a rotation of 180° .*

The previous theorem, although it is an immediate consequence of the BWT properties described in Remark 5, allows one to observe that the rows of R correspond to the conjugates of $\tilde{w}$ and for each row i , the reverse of the i -th conjugate of w in $\mathcal{B}_w$ coincides with the $(n + 1 - i)$ -th conjugate of w in $\mathcal{B}_w$. Using this fact, the following corollary can be derived:

► **Corollary 21** ([69, Corollary 4.4]). *Every conjugate of a perfectly clustering word has the two-palindrome property.*

► **Example 22.** The word $w = cacacacbbbbbb$ in Example 19 has the two-palindrome property, since it is the concatenation of $cacacac$ and $bbbbbb$.

The connection between palindromic factors and the property of being perfectly clustering is deepened in the following results. Recall that a word w is *circularly rich* if $w^2 = ww$ contains the maximum number $(|w^2| + 1)$ of distinct palindromic factors. In [63], the authors show a relationship between perfectly clustering words and circularly rich words.

► **Theorem 23** ([63, Theorem 6.2]). *If a word is perfectly clustering, then it is circularly rich.*

► **Example 24.** The word $w = cacacacbbbbbb$ of length 13 in Example 19 is circularly rich since w^2 has 27 distinct palindromic factors.

As mentioned before, extending the results of Theorem 15 to the case of larger alphabets is not easy. In fact, the following sets:

- circularly balanced words;
- perfectly clustering words;
- conjugates of powers of finite epistandard words,

which are equivalent for a two-letter alphabet, do not coincide when the alphabet has size ≥ 3 .

Also observe that, while the notions of epistandard, circularly balanced, and circularly rich word remain invariant under character permutation, the property of being perfectly clustering depends on the order of the alphabet. Hence, the equivalence among these three notions, as stated in the following theorem, holds for some permutation of the characters in the alphabet, but under the restrictive assumption of being a circularly balanced word.

► **Theorem 25** ([64, Theorem 8.16]). *Let Σ be an alphabet, and let $w \in \Sigma^*$ be a circularly balanced word over Σ . The following statements are equivalent, for some character permutation:*

- i) *w is a perfectly clustering word;*
- ii) *w is a circularly rich word;*
- iii) *w is a conjugate of a power of a finite epistandard word.*

However, Theorem 25 does not provide a complete characterization of perfectly clustering words. In fact, there exist circularly rich words that, although not circularly balanced, are perfectly clustering: an example is $w = bbbbacaca$.

Only recently, a complete characterization of the perfectly clustering words has been given by Lapointe and Reutenauer in [43]. In order to introduce this characterization, we first need some definitions.

Let w be a word over a totally ordered alphabet Σ . A *special factorization* of w is a factorization of the form

$$w = a_1 w_1 a_2 w_2 \cdots a_{k-1} w_{k-1} a_k,$$

where $a_1 < a_2 < \cdots < a_k$ and $w_1, \dots, w_{k-1} \in \Sigma^*$. If w satisfies this equality, we set $W = a_k w_{k-1} \cdots w_2 a_2 w_1 a_1$. We call this special factorization *palindromic* if each word w_i is a palindrome; in this case, $W = \tilde{w}$.

The characterization of perfectly clustering words is stated in the following theorem:

► **Theorem 26** ([43, Theorem 3.1]). *The following conditions are equivalent, for a primitive word w over a totally ordered finite alphabet Σ :*

1. *w is a perfectly clustering Lyndon word;*
2. *w has the two-palindrome property and w has a palindromic special factorization;*
3. *w has a special factorization such that w is a conjugate of W .*

Interestingly, Theorem 26 generalizes two important results about Christoffel words (that are clustering Lyndon words in the binary case). In particular, the equivalence of 1 and 2 is a generalization of a result of de Luca and Mignosi [17], and the equivalence of 1 and 3 is a generalization of a result of Pirillo [60].

Note that in [43, Corollary 6.1], the authors proved that perfectly clustering Lyndon words have a unique special factorization. Furthermore, the following theorem shows another interesting combinatorial property of the BWT matrix of a perfect clustering word. In Example 28, several properties of perfectly clustering words described in this section are illustrated.

► **Theorem 27** ([43, Theorem 6.1]). *Let w be a perfectly clustering word. Then, for any two consecutive rows of its Burrows-Wheeler matrix $\mathcal{B}_w$, viewed as two words w' and w'' , one has*

$$w' = ymx, \quad w'' = ym\tilde{x},$$

for some words x, y, m such that xy is one of the palindromes in the special palindromic factorization of w .

Providing a combinatorial characterization of the palindromic words that appear in the special palindromic factorization of perfectly clustering words remains an open problem. The case of the ternary alphabet has been studied in [42].

$$\begin{pmatrix} a & c & b & c & b & c & a & d & a & d & a & d & \mathbf{d} \\ a & d & a & c & b & c & b & c & a & d & a & d & \mathbf{d} \\ a & d & a & d & a & c & b & c & b & c & a & d & \mathbf{d} \\ a & d & a & d & a & d & a & c & b & c & b & c & \mathbf{d} \\ a & d & a & d & a & d & a & d & a & c & b & c & \mathbf{b} \\ b & c & a & d & a & d & a & d & a & d & a & c & \mathbf{b} \\ b & c & b & c & a & d & a & d & a & d & a & d & \mathbf{c} \\ c & a & d & a & d & a & d & a & d & a & c & b & \mathbf{b} \\ c & b & c & a & d & a & d & a & d & a & d & a & \mathbf{b} \\ c & b & c & b & c & a & d & a & d & a & d & a & \mathbf{a} \\ d & a & c & b & c & b & c & a & d & a & d & a & \mathbf{a} \\ d & a & d & a & c & b & c & b & c & a & d & a & \mathbf{a} \\ d & a & d & a & d & a & c & b & c & b & c & a & \mathbf{a} \\ d & a & d & a & d & a & d & a & c & b & c & b & \mathbf{a} \end{pmatrix}$$

■ **Figure 2** BWT-matrix $\mathcal{B}_w$ of $w = acbcbcadadadad$, $bwt(w)$ is highlighted in bold.

► **Example 28.** Consider the word $w = acbcbcadadadad$ over the 4-letter alphabet $\{a, b, c, d\}$, with $a < b < c < d$. Figure 2 shows the BWT matrix $\mathcal{B}_w$. According to Theorem 26, the following properties can be verified: 1) w is perfectly clustering, in fact $bwt(w) = d^4c^3b^2a^5$; 2) $w = (acbcbca)(dadadad)$ has the two-palindrome property and $w = a(cbc)bc(adadada)d$ has a palindromic special factorization; and 3) from the special factorization $w = a(cbc)bc(adadada)d$ one can get $W = d(adadada)cb(cbc)a$, which is the last row in the $\mathcal{B}_w$, i.e., it is a conjugate of w . Moreover, let us consider the consecutive rows $w' = bcadadadadacbc$ and $w'' = bcbcadadadadac$ corresponding to rows 6 and 7 of the BWT-matrix $\mathcal{B}_w$. As stated in Theorem 27, the following factorizations $w' = (bc)(adadadadacb)(c)$ and $w'' = (bc)(bcbadadadada)(c)$ can be obtained. Finally, the matrix $\mathcal{B}_w$ is invariant under a 180° -rotation (cf. Theorem 20).

5.2 Maximum Number of BWT Runs

In this section, we restrict our attention to the binary alphabet. For a word w of length n , it is possible in principle that the BWT of w has n clusters. For example, the BWT of $w = aaababbbab$ is $babababab = (ba)^5$. Of course, if this happens, then n must be even, i.e., $n = 2m$. Moreover, this can only happen under specific restrictions on the length, as it is shown below in this section. In particular, it is not known if this can happen for infinitely many lengths.

Recall that a prime number q is called *2-rooted* if 2 is a generator of the cyclic group $\mathbb{Z}_q^*$. For example, 5 is 2-rooted since $\{2^i \bmod 5 \mid i = 1, \dots, 4\} = \{2, 4, 3, 1\} = \mathbb{Z}_5^*$, while 7 is not 2-rooted since $\{2^i \bmod 7 \mid i = 1, \dots, 6\} = \{1, 2, 4\}$. The sequence of 2-rooted primes is A001122 in OEIS [70]: 3, 5, 11, 13, 19, 29, 37, 53, 59, 61, 67, etc.

$$\begin{pmatrix} a & a & a & b & a & b & b & \mathbf{b} \\ a & a & b & a & b & b & b & \mathbf{a} \\ a & b & a & b & b & b & a & \mathbf{a} \\ a & b & b & b & a & a & a & \mathbf{b} \\ b & a & a & a & b & a & b & \mathbf{b} \\ b & a & b & b & b & a & a & \mathbf{a} \\ b & b & a & a & a & b & a & \mathbf{b} \\ b & b & b & a & a & a & b & \mathbf{a} \end{pmatrix}$$

■ **Figure 3** BWT-matrix of the de Bruijn word $w = aaababbb$, $bwt(w)$ is highlighted in bold.

► **Proposition 29** ([52, Proposition 4.3]). *We have that $(ba)^m$ is a BWT image if and only if $2m + 1$ is a 2-rooted prime.*

This follows from the fact that the inverse standard permutation of $(ba)^m$, in cycle form, is equal to $(1 \ 2 \ 2^2 \ \dots \ 2^{2^m}) \bmod (2m + 1)$; the fact that it is a permutation means that its elements are all distinct, and therefore coincides with the cyclic group $\mathbb{Z}_{2m+1}^*$.

► **Conjecture 30.** *There are infinitely many values of m for which $(ba)^m$ is a BWT image.*

The previous conjecture is related to the famous conjecture of Emil Artin on primitively rooted primes.

However, one can show that there exist infinitely many lengths n for which the BWT has *almost* n runs. This is the case of binary de Bruijn words.

Recall that a binary word w is a de Bruijn word of order $d > 0$ if all the 2^d possible words of length d occur exactly once in it circularly. For example, $aaababbb$ is a de Bruijn word of order 3. A binary de Bruijn word of order d has length 2^d .

The BWT of a binary de Bruijn word of order d is always a word in the set $\{ab, ba\}^{2^{d-1}}$. Indeed, all the factors of length 2^{d-1} must occur twice in it, once preceded by a and once preceded by b , and in consecutive rows of the BWT matrix. This property was used in [47] for designing an efficient algorithm that generates every de Bruijn word with positive probability.

► **Example 31.** The BWT matrix of the de Bruijn word $aaababbb$ is depicted in Figure 3. So that $bwt(aaababbb) = baabbaba \in \{ab, ba\}^4$.

As a consequence, we have (see also [52]):

► **Proposition 32.** *The BWT of any binary de Bruijn word of order $d > 2$ has at least $2^{d-1} + 2$ and at most $2^d - 2$ clusters.*

A binary de Bruijn word of order $d > 2$ cannot have 2^d clusters, i.e., no binary de Bruijn word of order $d > 2$ can have BWT equal to $(ba)^{2^{d-1}}$. This follows from Proposition 29 and the fact that 3 and 5 are the only 2-rooted primes of the form $2^d + 1$; in the cases $d = 1, 2$ we have, respectively, the de Bruijn words ab and $aabb$, whose BWT have 2 and 4 clusters, respectively.

6 Sensitivity of r to Modifications of the Input Word

In this section, we present some recent results concerning the sensitivity of r with respect to some combinatorial operations on words. Sensitivity analysis is a fundamental aspect of understanding the stability of compression-based repetitiveness measures.

6.1 Reverse Operation

Several repetitiveness measures, such as b , g , γ , and δ , are invariant under the reverse operation. This is not the case for the measure r , despite experimental results showing that, on real text collections, it exhibits a behavior that brings it closer to the other above measures [5, 61]. A first upper bound on the ratio between r and the number of runs in the BWT of the reverse of the word was provided in [33], where it was proven that $\frac{r(\tilde{v})}{r(v)} = \mathcal{O}(\log^2 n)$, where n is the length of the word, leaving open the question of whether this bound is tight.

An answer to this problem was given in [29], where the impact of the reverse operation on the number of BWT runs was studied for an infinite family of words, the so-called Fibonacci-plus words.

A word v is *Fibonacci-plus* if and only if $v = sb$ for some Fibonacci word s of order $2k$, $k \geq 2$, or $v = sa$ for some Fibonacci word s of order $2k + 1$, $k \geq 2$. The Fibonacci-plus word v has *even order* in the first case and *odd order* otherwise.

► **Example 33.** The words $v_1 = abaababaabaabb = f_6b$ and $v_2 = abaababaabaababababaa = f_7a$ are Fibonacci-plus words. It can be easily verified that $r(v_1) = r(v_2) = 4$ and $r(\tilde{v}_1) = r(\tilde{v}_2) = 6$. This shows that r is not invariant under the reverse operation.

More specifically, if v is a Fibonacci-plus word, then $r(v) = \Theta(1)$, while $r(\tilde{v}) = \Theta(\log(|v|))$, as synthesized in the following theorem.

► **Theorem 34** ([29, Theorem 1]). *Let v be a Fibonacci-plus word, and let $|v| = n$. Then $\frac{r(\tilde{v})}{r(v)} = \Theta(\log n)$.*

Closing the gap between the $\Omega(\log n)$ lower bound (established in [29]) and the $O(\log^2 n)$ upper bound (given in [33]) remains an open problem.

6.2 Edit Operations

In [1], the sensitivity of the measure r with respect to a single edit operation on a word (deletion, substitution, insertion of a character) has been studied. More precisely, the *multiplicative sensitivity* MS of r is defined as the maximum multiplicative factor by which a single-character edit operation on a word w can alter the value $r(w)$. Furthermore, the *additive sensitivity* AS , i.e., the maximum additive factor that an edit operation on a word w may introduce in $r(w)$. Formally, if we denote by $D(w)$, $S(w)$, and $I(w)$ the set of words obtained after a single deletion, substitution, and insertion on a word w respectively, the multiplicative and additive sensitivities for each of the edit operations are defined as follows:

$$\begin{aligned} MS_{del}(n) &= \max_{\substack{w \in \Sigma^n \\ w' \in D(w)}} \left\{ \frac{r(w')}{r(w)} \right\}; & AS_{del}(n) &= \max_{\substack{w \in \Sigma^n \\ w' \in D(w)}} \{r(w') - r(w)\}; \\ MS_{sub}(n) &= \max_{\substack{w \in \Sigma^n \\ w' \in S(w)}} \left\{ \frac{r(w')}{r(w)} \right\}; & AS_{sub}(n) &= \max_{\substack{w \in \Sigma^n \\ w' \in S(w)}} \{r(w') - r(w)\}; \\ MS_{ins}(n) &= \max_{\substack{w \in \Sigma^n \\ w' \in I(w)}} \left\{ \frac{r(w')}{r(w)} \right\}; & AS_{ins}(n) &= \max_{\substack{w \in \Sigma^n \\ w' \in I(w)}} \{r(w') - r(w)\}. \end{aligned}$$

By combining the results on the sensitivity of the measure δ [1, Theorem 2] and on the relationship between the measure r and δ [33, Theorem 3.7], the following upper bounds on the sensitivities of r hold.

► **Proposition 35** ([1, Corollary 3]). *The following upper bounds on the sensitivity of the measure r hold:*

- $MS_{del}(n) = MS_{sub}(n) = MS_{ins}(n) = O(\log n \log r);$
- $AS_{del}(n) = AS_{sub}(n) = AS_{ins}(n) = O(r \log n \log r);$

On the other hand, the lower bounds for the multiplicative and the additive sensitivities have been proved by showing the effect of single edit operations on specific families of words [29, 30].

► **Proposition 36** ([29, Theorem 1], [30, Propositions 5, 6, and 9]). *The following lower bounds on the sensitivity of the measure r hold:*

- $MS_{del}(n) = MS_{sub}(n) = MS_{ins}(n) = \Omega(\log n);$
- $AS_{del}(n) = AS_{sub}(n) = AS_{ins}(n) = \Omega(\sqrt{n});$

The lower bounds on the multiplicative sensitivity for deletion, substitution, and insertion have been proved via Fibonacci words, where an increase by a multiplicative $\Theta(\log n)$ factor is attained after each edit operation is applied. Since the measure r for Fibonacci words is $\Theta(1)$ [53, Theorem 9], this proves that the upper bound on MS given in Proposition 35 is tight.

The $\Omega(\sqrt{n})$ lower bound on the additive sensitivity is rather obtained with a novel family of words defined in [30]. Note, however, that such a family is already loosely compressible via BWT, as the measure r is $\Theta(\sqrt{n})$ on the words from this family before and after the edit operations are applied. The problem of proving the tightness of the upper bound $O(r \log n \log r)$ given in Proposition 35 for the additive sensitivity remains open.

6.3 Morphisms

Another interesting question is how the measure r changes under the application of a morphism on a word w . This problem has been explored by Brlek et al. in [11], where they studied the BWT runs for words obtained by applying a composition of the Thue–Morse morphism with some *Sturmian morphisms*. Recall that a morphism is Sturmian if it maps Sturmian words to Sturmian words. Recall also that every Sturmian morphism is a composition of morphisms taken from the set $\{E, \varphi, \tilde{\varphi}\}$, where:

$$E : \begin{cases} a \mapsto b \\ b \mapsto a \end{cases} \quad \varphi : \begin{cases} a \mapsto ab \\ b \mapsto a \end{cases} \quad \tilde{\varphi} : \begin{cases} a \mapsto ba \\ b \mapsto a \end{cases}.$$

Specifically, Brlek et al. proved that $r(\tau^{n+1}(a)) = r((\tau \circ \varphi)^n(a))$, for all $n \geq 1$ [11, Theorems 2 and 5], and the same has been experimentally observed for compositions of the Thue–Morse morphism with other Sturmian morphisms. In fact, this is a consequence of the next theorem, which provides a characterization of Sturmian morphisms in terms of their effect on the measure r .

► **Theorem 37** ([22, Theorem 21]). *Let μ be a binary morphism. Then $r(w) = r(\mu(w))$ for every word w with $|\text{alph}(w)| = 2$ if and only if μ is a Sturmian morphism.*

► **Example 38.** Let $w = abbaba$. In Figure 4 we show the BWT-matrices for the words w and $\varphi(w)$. Since $\tilde{\varphi} \in [\varphi(w)]$, one has $\mathcal{B}_{\varphi(w)} = \mathcal{B}_{\tilde{\varphi}(w)}$. Moreover, one can see that $\text{bwt}(\varphi(w)) = \text{bwt}(\tilde{\varphi}(w)) = E(\text{bwt}(\tilde{w}))a^{|w|_a}$.

$$\begin{array}{cc}
 \left(\begin{array}{c} a \ a \ b \ b \ a \ \mathbf{b} \\ a \ b \ a \ a \ b \ \mathbf{b} \\ a \ b \ b \ a \ b \ \mathbf{a} \\ b \ a \ a \ b \ b \ \mathbf{a} \\ b \ a \ b \ a \ a \ \mathbf{b} \\ b \ b \ a \ b \ a \ \mathbf{a} \end{array} \right) & \left(\begin{array}{c} a \ a \ a \ b \ a \ a \ b \ a \ \mathbf{b} \\ a \ a \ b \ a \ a \ b \ a \ a \ \mathbf{b} \\ a \ b \ a \ a \ a \ b \ a \ a \ \mathbf{b} \\ a \ b \ a \ a \ b \ a \ b \ a \ \mathbf{a} \\ a \ b \ a \ b \ a \ a \ a \ b \ \mathbf{a} \\ b \ a \ a \ a \ b \ a \ a \ b \ \mathbf{a} \\ b \ a \ a \ b \ a \ b \ a \ a \ \mathbf{a} \\ b \ a \ b \ a \ a \ a \ b \ a \ \mathbf{a} \end{array} \right) \\
 \text{(a) } \mathcal{B}_w. & \text{(b) } \mathcal{B}_{\varphi(w)} = \mathcal{B}_{\tilde{\varphi}(w)}.
 \end{array}$$

■ **Figure 4** In Subfigure 4a is shown the BWT-matrix for the word $w = abbaba$, while in Subfigure 4b is shown the BWT-matrix for the word $\varphi(w) = abaaabaab$, where φ is the Fibonacci morphism.

In [22], the more general problem of the sensitivity of the measure r to the application of morphisms has been tackled. In particular, they showed that for every binary word w and every binary injective morphism μ , $r(\mu(w)) \geq r(w)$ holds, so the Sturmian morphisms are the only binary morphisms achieving such a lower bound.

For the family of *Thue–Morse-like morphisms* defined as $\bigcup_{p,q \geq 1} \{\tau_{p,q} : \{a,b\}^* \rightarrow \{a,b\}^* \mid \tau_{p,q}(a) = ab^p, \tau_{p,q}(b) = ba^q\}$, the inequality $r(\tau_{p,q}(w)) \leq r(w) + 2$ holds [22, Proposition 23]. It is natural to ask whether or not the sensitivity of r to the application of a binary morphism is bounded, i.e., if for each binary injective morphism μ there exists a constant k that solely depends on μ such that $r(\mu(w)) \leq r(w) + k$, for every binary word w . However, the *period-doubling morphism* $\theta : \{a,b\}^* \rightarrow \{a,b\}^*$, with $\theta(a) = ab$ and $\theta(b) = aa$, negatively answers this question.

► **Lemma 39** ([22, Lemma 29]). *Let θ be the period-doubling morphism. For any positive integer k there exists a word w such that $r(\theta(w)) > r(w) + k$.*

The problem of characterizing binary morphisms based on their effect on the measure r is still open.

7 Conclusions

The study of combinatorial aspects of the BWT has attracted significant interest over the years. On the one hand, it has led to a deeper understanding of the properties that make the BWT so distinctive. On the other hand, it has opened the way for different types of extensions or generalizations that have not been covered in this chapter. One such extension concerns the input. As described in Section 3, the BWT can be viewed as a mapping from the set of conjugacy classes (also called necklaces) of words in Σ^* to Σ^* , where Σ is a finite ordered alphabet. However, this map is not surjective. In 2007, Mantaci et al. introduced in [50] the *Extended Burrows Wheeler Transform* (EBWT), which is instead a bijective map between the multisets of conjugacy classes of words in Σ^* and Σ^* . Efficient computations of this transformation have been recently provided [10, 4, 57]. The EBWT also allows to define a bijective transformation on Σ^* , known as the Bijective BWT (BBWT), which uses the unique factorization of a word into non-increasing Lyndon factors [37]. The relationship between the number of runs produced by the BBWT and the number of BWT runs is studied in [3]. Transformations EBWT and BBWT are discussed in other chapters of this Festschrift.

Another generalization of the BWT is based on the order used to sort the cyclic rotations of a word. In [26], the authors introduced and studied the basic properties of the Alternating BWT (ABWT). It works similarly to the BWT, but instead of using the standard lexicographic order, the cyclic rotations of the input word are sorted according to the alternating lexicographic order. The alternating lexicographic order is defined as follows: the first characters are compared using the given alphabetic order, in case of equality, the second characters are compared using the opposite order, and so on, alternating between the two orders for even and odd positions. In [27], the authors showed that the *ABWT* retains most of the key properties that make the BWT a powerful transformation. In particular, the *ABWT* can be computed and inverted in linear time and can also be used to construct an efficient compressed full-text index for the transformed word. In [28], a new class of BWT variants has been introduced based on local orderings, where the alphabet order depends only on a fixed portion of the preceding context of each character. These transformations maintain the key properties of both the BWT and *ABWT*, by showing that an *r*-index can always be built on top of a BWT defined by a local ordering. This makes such transformations particularly well-suited for indexing highly repetitive text collections. While it is possible to find in linear time, for a given word w , a BWT variant based on context-adaptive alphabet orderings that minimizes the number of BWT runs [28], it has been proven that the problem of finding an alphabet reordering that minimizes the number of BWT runs is NP-complete [6].

References

- 1 Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *Inf. Comput.*, 291:104999, 2023. doi:10.1016/J.IC.2022.104999.
- 2 Hideo Bannai, Mitsuru Funakoshi, Tomohiro I, Dominik Köppl, Takuya Mieno, and Takaaki Nishimoto. A separation of γ and b via Thue-Morse words. In *SPIRE*, volume 12944 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 2021. doi:10.1007/978-3-030-86692-1_14.
- 3 Hideo Bannai, Tomohiro I, and Yuto Nakashima. On the Compressiveness of the Burrows-Wheeler Transform. In *CPM*, volume 331 of *LIPICs*, pages 17:1–17:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICs.CPM.2025.17.
- 4 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing and indexing the bijective and extended Burrows-Wheeler transform. *Inf. Comput.*, 297:105153, 2024. doi:10.1016/J.IC.2024.105153.
- 5 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite Repetition-Aware Data Structures. In *26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *Lecture Notes in Computer Science*, pages 26–39. Springer, 2015. doi:10.1007/978-3-319-19929-0_3.
- 6 Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the Complexity of BWT-Runs Minimization via Alphabet Reordering. In *Proceedings of the 28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *LIPICs*, pages 15:1–15:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ESA.2020.15.
- 7 Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A Locally Adaptive Data Compression Scheme. *Commun. ACM*, 29(4):320–330, 1986. doi:10.1145/5684.5688.
- 8 Valérie Berthé and Christophe Reutenauer. On the Three-Distance Theorem. *The Mathematical Intelligencer*, 46(2):183–188, 2024. doi:10.1007/s00283-023-10316-z.
- 9 Anselm Blumer, J. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987. doi:10.1145/28869.28873.
- 10 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the Original eBWT Faster, Simpler, and with Less Memory. In

- 28th International Symposium on String Processing and Information Retrieval (SPIRE), volume 12944 of *Lecture Notes in Computer Science*, pages 129–142. Springer, 2021. doi:10.1007/978-3-030-86692-1_11.
- 11 Srečko Brlek, Andrea Frosini, Iliaria Mancini, Elisa Pergola, and Simone Rinaldi. Burrows-Wheeler Transform of Words Defined by Morphisms. In *IWOCA*, volume 11638 of *Lecture Notes in Computer Science*, pages 393–404. Springer, 2019. doi:10.1007/978-3-030-25005-8_32.
 - 12 Michael Burrows and David J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DIGITAL System Research Center, 1994.
 - 13 Julien Cassaigne, France Gheeraert, Antonio Restivo, Giuseppe Romana, Marinella Sciortino, and Manon Stipulanti. New string attractor-based complexities for infinite words. *Journal of Combinatorial Theory, Series A*, 208:105936, 2024. doi:10.1016/J.JCTA.2024.105936.
 - 14 Julien Cassaigne and François Nicolas. Factor complexity. In Valérie Berthé and Michel Rigo, editors, *Combinatorics, Automata and Number Theory*, Encyclopedia of Mathematics and its Applications, pages 163–247. Cambridge University Press, 2010. doi:10.1017/CBO9780511777653.
 - 15 Sorin Constantinescu and Lucian Ilie. The Lempel–Ziv complexity of fixed points of morphisms. *SIAM J. Discret. Math.*, 21(2):466–481, 2007. doi:10.1137/050646846.
 - 16 Maxime Crochemore, Jacques Désarménien, and Dominique Perrin. A note on the Burrows-Wheeler transformation. *Theoret. Comput. Sci.*, 332(1-3):567–572, 2005. doi:10.1016/j.tcs.2004.11.014.
 - 17 Aldo de Luca and Filippo Mignosi. Some Combinatorial Properties of Sturmian Words. *Theor. Comput. Sci.*, 136(2):361–285, 1994. doi:10.1016/0304-3975(94)00035-H.
 - 18 Xavier Droubay, Jacques Justin, and Giuseppe Pirillo. Episturmian words and some constructions of de Luca and Rauzy. *Theoret. Comput. Sci.*, 255(1-2):539–553, 2001. doi:10.1016/S0304-3975(99)00320-5.
 - 19 Sébastien Ferenczi and Luca Q. Zamboni. Clustering Words and Interval Exchanges. *Journal of Integer Sequences*, 16(2):Article 13.2.1, 2013. URL: <https://cs.uwaterloo.ca/journals/JIS/VOL16/Zamboni/zamboni2.pdf>.
 - 20 Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Marinella Sciortino. Boosting textual compression in optimal linear time. *J. ACM*, 52(4):688–713, 2005. doi:10.1145/1082036.1082043.
 - 21 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
 - 22 Gabriele Fici, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. On the Impact of Morphisms on BWT-Runs. In *CPM 2023*, volume 259 of *LIPICs*, pages 10:1–10:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.CPM.2023.10.
 - 23 Andrea Frosini, Iliaria Mancini, Simone Rinaldi, Giuseppe Romana, and Marinella Sciortino. Logarithmic equal-letter runs for BWT of purely morphic words. In *DLT 2022*, volume 13257 of *Lecture Notes in Computer Science*, pages 139–151. Springer, 2022. doi:10.1007/978-3-031-05578-2_11.
 - 24 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1459–1477, Philadelphia, PA, 2018. SIAM. doi:10.1137/1.9781611975031.96.
 - 25 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. doi:10.1145/3375890.
 - 26 Ira M. Gessel, Antonio Restivo, and Christophe Reutenauer. A bijection between words and multisets of necklaces. *European Journal of Combinatorics*, 33(7):1537–1546, 2012. Groups, Graphs, and Languages. doi:10.1016/J.EJC.2012.03.016.
 - 27 Raffaele Giancarlo, Giovanni Manzini, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. The alternating BWT: An algorithmic perspective. *Theor. Comput. Sci.*, 812:230–243, 2020. doi:10.1016/j.tcs.2019.11.002.

- 28 Raffaele Giancarlo, Giovanni Manzini, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. A new class of string transformations for compressed text indexing. *Inf. Comput.*, 294:105068, 2023. doi:10.1016/J.IC.2023.105068.
- 29 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 *SOFSEM*, volume 12607 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2021. doi:10.1007/978-3-030-67731-2_18.
- 30 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. Bit Catastrophes for the Burrows-Wheeler Transform. In *DLT 2023*, volume 13911 of *Lecture Notes in Computer Science*, pages 86–99. Springer, 2023. doi:10.1007/978-3-031-33264-7_8.
- 31 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. Bit Catastrophes for the Burrows-Wheeler Transform. *Theory Comput. Syst.*, 69(2):19, 2025. doi:10.1007/s00224-024-10212-9.
- 32 Peter M. Higgins. Burrows-Wheeler transformations and de Bruijn words. *Theor. Comput. Sci.*, 457:128–136, 2012. doi:10.1016/j.tcs.2012.07.019.
- 33 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. *Commun. ACM*, 65(6):91–98, 2022. doi:10.1145/3531445.
- 34 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *STOC*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.
- 35 John C. Kieffer and En hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000. doi:10.1109/18.841172.
- 36 Dmitry Kosolobov and Arseny M. Shur. Comparison of LZ77-type parsings. *Inf. Process. Lett.*, 141:25–29, 2019. doi:10.1016/J.IPL.2018.09.005.
- 37 Manfred Kufleitner. On bijective variants of the Burrows-Wheeler transform. In *Proceedings of PSC*, pages 65–79, 2009. URL: <http://www.stringology.org/event/2009/p07.html>.
- 38 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. doi:10.1089/CMB.2019.0309.
- 39 Kanaru Kutsukake, Takuya Matsumoto, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. On repetitiveness measures of Thue-Morse words. In *SPIRE*, volume 12303 of *Lecture Notes in Computer Science*, pages 213–220. Springer, 2020. doi:10.1007/978-3-030-59212-7_15.
- 40 Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon C. K. Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *2009 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2009, Washington, DC, USA, November 1-4, 2009, Proceedings*, pages 31–36, Los Alamitos, CA, 2009. IEEE Computer Society. doi:10.1109/BIBM.2009.42.
- 41 Ben Langmead and Steven L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012. doi:10.1038/nmeth.1923.
- 42 Mélodie Lapointe and Nathan Plourde-Hébert. Perfectly Clustering Words and Iterated Palindromes over a Ternary Alphabet. In *International Conference on Random and Exhaustive Generation of Combinatorial Structures*, 2024. URL: <https://cgi.cse.unsw.edu.au/~eptcs/paper.cgi?GASCom2024.28>.
- 43 Mélodie Lapointe and Christophe Reutenauer. Characterizations of Perfectly Clustering Words, 2024. arXiv:2407.19140.
- 44 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976. doi:10.1109/TIT.1976.1055501.
- 45 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinform.*, 26(5):589–595, 2010. doi:10.1093/bioinformatics/btp698.

- 46 Konstantin M. Likhomanov and Arseny M. Shur. Two combinatorial criteria for BWT images. In *Computer Science - Theory and Applications - 6th International Computer Science Symposium in Russia, CSR 2011, St. Petersburg, Russia, June 14-18, 2011. Proceedings*, volume 6651 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2011. doi:10.1007/978-3-642-20712-9_30.
- 47 Zsuzsanna Lipták and Luca Parmigiani. A BWT-Based Algorithm for Random de Bruijn Sequence Construction. In *LATIN (1)*, volume 14578 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2024. doi:10.1007/978-3-031-55598-5_9.
- 48 M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002. doi:10.1017/CB09781107326019.
- 49 Sabrina Mantaci, Antonio Restivo, Giuseppe Romana, Giovanna Rosone, and Marinella Sciortino. A combinatorial view on string attractors. *Theor. Comput. Sci.*, 850:236–248, 2021. doi:10.1016/J.TCS.2020.11.006.
- 50 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.
- 51 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Burrows-Wheeler Transform and Run-Length Encoding. In *WORDS 2017*, volume 10432 of *Lecture Notes in Computer Science*, pages 228–239. Springer, 2017. doi:10.1007/978-3-319-66396-8_21.
- 52 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, Marinella Sciortino, and Luca Versari. Measuring the clustering effect of BWT via RLE. *Theor. Comput. Sci.*, 698:79–87, 2017. doi:10.1016/j.tcs.2017.07.015.
- 53 Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows-Wheeler transform and Sturmian words. *Information Processing Letters*, 86:241–246, 2003. doi:10.1016/S0020-0190(02)00512-4.
- 54 Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001. doi:10.1145/382780.382782.
- 55 Takuya Mieno, Shunsuke Inenaga, and Takashi Horiyama. RePair grammars are the smallest grammars for Fibonacci words. In *CPM*, volume 223 of *LIPICs*, pages 26:1–26:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.CPM.2022.26.
- 56 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.
- 57 Jannik Olbrich, Enno Ohlebusch, and Thomas Böhler. Generic non-recursive suffix array construction. *ACM Trans. Algorithms*, 20(2), 2024. doi:10.1145/3641854.
- 58 Igor Pak and Amanda Redlich. Long cycles in abc-permutations. *Functional Analysis and Other Mathematics*, 2:87–92, 2008. doi:10.1007/s11853-008-0017-0.
- 59 Dominique Perrin and Antonio Restivo. Words. In Miklós Bóna, editor, *Handbook of Enumerative Combinatorics*, Discrete Mathematics and its Applications, pages 485–539. Chapman & Hall/CRC, 2015. doi:10.1201/b18255.
- 60 Giuseppe Pirillo. A new characteristic property of the palindrome prefixes of a standard Sturmian word. In *Séminaire Lotharingien de Combinatoire 43*, 1999. URL: <http://eudml.org/doc/120436>.
- 61 Alberto Policriti and Nicola Prezza. From LZ77 to the Run-Length Encoded Burrows-Wheeler Transform, and Back. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.17.
- 62 Antonio Restivo, Giuseppe Romana, and Marinella Sciortino. String attractors and infinite words. In *LATIN*, volume 13568 of *Lecture Notes in Computer Science*, pages 426–442. Springer, 2022. doi:10.1007/978-3-031-20624-5_26.
- 63 Antonio Restivo and Giovanna Rosone. Burrows-Wheeler transform and palindromic richness. *Theoretical Computer Science*, 410(30):3018–3026, 2009. doi:10.1016/j.tcs.2009.03.008.

- 64 Antonio Restivo and Giovanna Rosone. Balancing and clustering of words in the Burrows-Wheeler transform. *Theoretical Computer Science*, 412(27):3019–3032, 2011. doi:10.1016/j.tcs.2010.11.040.
- 65 Giuseppe Romana. *Repetitiveness Measures based on String Attractors and Burrows-Wheeler Transform: Properties and Applications*. PhD thesis, University of Palermo, 2023. URL: <https://hdl.handle.net/10447/595273>.
- 66 Giovanna Rosone and Marinella Sciortino. The Burrows-Wheeler Transform between Data Compression and Combinatorics on Words. In *CiE 2013*, volume 7921 of *Lecture Notes in Computer Science*, pages 353–364. Springer, 2013. doi:10.1007/978-3-642-39053-1_42.
- 67 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.
- 68 Luke Schaeffer and Jeffrey O. Shallit. String attractors for automatic sequences, 2021. doi:10.48550/arXiv.2012.06840.
- 69 Jamie Simpson and Simon J. Puglisi. Words with Simple Burrows-Wheeler Transforms. *Electron. J. Comb.*, 15(1), 2008. doi:10.37236/807.
- 70 Neil J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. Available electronically at <http://oeis.org>.
- 71 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

A Survey of the Bijective Burrows-Wheeler Transform

Hideo Bannai 

M&D Data Science Center, Institute of Integrated Research, Institute of Science Tokyo, Japan

Dominik Köppl 

Department of Computer Science and Engineering, University of Yamanashi, Japan

Zsuzsanna Lipták 

Department of Computer Science, University of Verona, Italy

Abstract

The Bijective BWT (BBWT), conceived by Scott in 2007, later summarized in a preprint by Gil and Scott in 2009 (arXiv 2012), is a variant of the Burrows-Wheeler Transform which is bijective: every string is the BBWT of some string. Indeed, the BBWT of a string is the extended BWT [Mantaci et al., 2007] of the factors of its Lyndon factorization. The BBWT has been receiving increasing interest in recent years. In this paper, we survey existing research on the BBWT, starting with its history and motivation. We then present algorithmic topics including construction algorithms with various complexities and an index on top of the BBWT for pattern matching. We subsequently address some properties of the BBWT as a compressor, discussing robustness to operations such as reversal, edits, rotation, as well as compression power. We close with listing other bijective variants of the BWT and open problems concerning the BBWT.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Mathematics of computing → Combinatorics on words

Keywords and phrases Burrows-Wheeler Transform, compression, text indexing, repetitiveness measure, Lyndon words, index construction algorithms, bijective string transformation

Digital Object Identifier 10.4230/OASICS.Manzini.2025.2

Funding *Hideo Bannai*: JSPS KAKENHI Grant Number JP24K02899

Dominik Köppl: JSPS KAKENHI Grant Numbers JP23H04378 and JP25K21150

Zsuzsanna Lipták: Partially funded by the Italian Ministry of University and Research (MUR) PRIN Project PINC, Pangenome INformatiCs: from Theory to Applications (Grant No. 2022YRB97K), and by the INdAM - GNCS Project CUP_E53C24001950001.

1 Introduction

The famous Burrows-Wheeler transform [17] (BWT) of a string is defined as the string obtained by concatenating the last letter of all rotations of the string, in the lexicographic order of the rotations. The BWT is often described as a reversible transform, meaning it must be injective, but clearly it is not, since all rotations of the string share the same output. Therefore, to make it injective and thus reversible, information to encode which of the strings among all rotations was the original is required; typically in the form of an explicit end-of-string symbol (often denoted by \$), or an integer indicating the position in the BWT that corresponds to the lexicographic rank of the original string among all rotations, and therefore to the last character of the original string. It is also clear by a simple counting argument, that the transform is not surjective, meaning that there exist strings which are not images of BWT. Note that this is true even with the addition of the extra information to make it injective [37]. For example, \$ab (assuming $a < b$) is not a BWT image.¹

¹ As a famous example, **banana** is not a BWT image, nor is \$banana, b\$anana, ..., banana\$; in other words, one cannot insert \$ anywhere for it to become a BWT image. For more on this problem, see [37].



The bijective variant of the BWT, *Bijective BWT* (BBWT, a.k.a. BWTS for *BWT Scottified*) considered in this paper was first announced by David Allen Scott in a Usenet thread in December 2007 [66], where the process of making the BWT bijective was termed the *Scottification* of the BWT. While Scott provided a working implementation together with explanations using concrete examples, a rigorous formal description of the procedure was not given, and the excitement of the discovery was unfortunately met with great skepticism towards its correctness and usefulness. In 2009, Joseph Gil, together with Scott, made available a preprint on Scott’s website [68] – later uploaded to arXiv in 2012 [32] – which contained a more accessible description of the process and a proof of correctness. Also in 2009, Kufleitner [47] provided another formal description of the BBWT, along with another BWT variant (which we will not cover in this survey). Scott later mentions [67] that his paper with Gil had been submitted to the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) but was rejected, highlighting the reviewer comment:

“Of course one can easily achieve invertability [sic!] in Burrows-Wheeler transform, by appending a special character to the input string.”

To the best of our knowledge, the paper has not been subsequently published in a journal or proceedings².

Three advantages of BBWT over BWT are mentioned by Gil and Scott. First, it allows to get rid of the extra information mentioned above. Gil and Scott report empirical evidence that compression using BBWT is almost always better (albeit only slightly) than compression using BWT, on the Calgary corpus [9]. Second, they claim that in the context of applying encryption after compression, bijectivity is preferable, since *not* being bijective would reveal information to an attacker. Third, it is simply mathematically more elegant.

As can be seen from the reviewer’s comment to their paper (which misses the point), the difference between BBWT and BWT may be quite subtle and perhaps difficult to understand. Although the immediately observable merits of BBWT may have been scarce, some recent works have started to realize and tap into the value of this transform.

In this survey, we summarize currently known results on the BBWT, concerning its construction (Section 3), its use as an index (Section 4), and its performance as a compressor (Section 5).

2 Basics

Let Σ denote the *alphabet*, i.e., the set of symbols (or characters), and Σ^* the set of (finite) strings (or texts) over Σ . We index strings from 0. We denote the empty string by ε , the length of a string T by $|T|$, and the substring from i to j by $T[i..j]$, where $T[i..j] = \varepsilon$ if $j < i$. We will also use $T[i..j]$ to denote $T[i..j - 1]$. The reverse of string T is denoted T^{rev} . Let $\text{rot}(T) = T[1..|T|]T[0]$. Then, $\text{rot}^i(T) = T[i..|T|]T[0..i]$ is the rotation of T that starts at position i . A *cyclic string* T is a string whose first character $T[0]$ is considered as being subsequent to its last character $T[|T| - 1]$. In other words, arithmetic on the positions $0, 1, \dots, |T| - 1$ of T is done modulo $|T|$.

A string T is called *primitive* if it cannot be represented as $T = U^k$ for some string U and integer $k \geq 2$; otherwise T is called a *power*. A string S is a *rotation* (or *conjugate*) of a string T if there exist $U, V \in \Sigma^*$ such that $T = UV$ and $S = VU$. Conjugacy is an

² An interesting parallel with the original BWT paper [17].

equivalence relation, and it is easy to see that the conjugacy class $[T] = \{VU : T = UV\}$ has cardinality $|T|$ if and only if T is primitive.

We denote the number of the maximal same symbol runs, or simply *runs*, of a string T by $\rho(T)$. For example, $\rho(\text{cabbbbaa}) = 4$. Note that $\rho(T)$ is the size of the run-length encoding of T .

We assume a total order on Σ and denote by $\prec$ the *lexicographic order* on Σ^* induced by this total order. Furthermore, we define the *omega-order* (also called *infinite periodic order* [32, 64]) on Σ^* as follows: $T \prec_\omega S$ if either $T^\omega \prec S^\omega$, or $T^\omega = S^\omega$ and $|T| < |S|$. Here, X^ω denotes the infinite string obtained by concatenating string X an infinite number of times. Note that the two orders $\prec$ and $\prec_\omega$ coincide if neither of two strings is a proper prefix of the other but may differ otherwise. For example, $\text{ab} \prec \text{aba}$ but $\text{aba} \prec_\omega \text{ab}$. The omega-order can also be viewed as an order on cyclic strings.

Since all rotations of a string have the same length, the two orders $\prec$ and $\prec_\omega$ coincide on the conjugacy classes. Given a string T , let $\text{minrot}(T)$ denote the smallest rotation of T in lexicographic, or equivalently in this case, in omega-order, i.e. $\text{minrot}(T) = \min[T]$. If $T = \text{minrot}(T)$ then T is called a *necklace*³; if T is in addition primitive, then it is called a *Lyndon word*. Accordingly, $\text{minrot}(T)$ is also referred to as *necklace rotation* resp. *Lyndon rotation* of T . For instance, $\text{minrot}(\text{baba}) = \text{abab}$ is a necklace and $\text{minrot}(\text{baaba}) = \text{aabab}$ is a Lyndon word.

The *Lyndon factorization* [20] of a string T is a factorization $T = L_1^{e_1} \cdots L_{\ell(T)}^{e_{\ell(T)}}$ where for all $1 \leq i \leq \ell(T)$, $e_i \geq 1$ and L_i is a Lyndon word, and $L_i \succ_\omega L_{i+1}$. Note that our definition differs slightly from the usual one in that also here we are using the omega-order rather than the lexicographic order. Since on Lyndon words, the two orders coincide [27, Theorem 20], the two definitions are equivalent. We will refer to L_i above, for $1 \leq i \leq \ell(T)$, as the *i*th *Lyndon factor* of T . It is known that the Lyndon factorization of a string is unique [20] and can be computed in linear time in a left-to-right scan with Duval's algorithm [28].

The *suffix array* SA of a string T of length n is a permutation of the index set $\{0, \dots, n-1\}$, defined as follows: $\text{SA}[j] = i$ if the *i*th suffix $T[i..n-1]$ of T is the *j*th in lexicographic order of all suffixes (counting from 0). The *conjugate array* CA (also called *circular suffix array* and denoted $\text{SA}_\circ$) of a string T is a permutation of the index set $\{0, \dots, n-1\}$, defined as follows: $\text{CA}[j] = i$ if the *i*th rotation $\text{rot}^i(T)$ is the *j*th in lexicographic order (or, in this case equivalently, in omega-order) among all rotations of T . (If T is not primitive, then there are equal rotations, which are listed in CA in text order.) Note that SA and CA coincide if T is a Lyndon word, or if it is terminated by an end-of-string symbol, but otherwise they can differ.

Standard Sturmian words, or simply *standard words*, are strings that can be defined via a so-called *directive sequence*: this is an infinite sequence of integers $(d_i)_{i \geq 0}$ such that $d_0 \geq 0$ and for all $i > 0$, $d_i > 0$. This sequence generates a sequence of finite strings $(S_i)_{i \geq 0}$, called *standard words*, as follows: $S_0 = \mathbf{b}$, $S_1 = \mathbf{a}$, and $S_{i+1} = S_i^{d_i-1} S_{i-1}$ for $i \geq 1$. For instance, the directive sequence $1, 1, 1, 1, \dots$ generates the well-known sequence of *finite Fibonacci words*: $F_0 = \mathbf{b}$, $F_1 = \mathbf{a}$, $F_2 = \mathbf{ab}$, $F_3 = \mathbf{aba}$, $F_4 = \mathbf{abaab}$, $F_5 = \mathbf{abaababa}$, $F_6 = \mathbf{abaababaabaab}$, $F_7 = \mathbf{abaababaabaabaabaab}$, $\dots$ For more details, see [52, Ch. 2].⁴

³ Note that a *necklace* is sometimes defined as a conjugacy class w.r.t. rotation. The two definitions are closely connected: a necklace viewed as a string is a particular representative of a necklace viewed as a conjugacy class. In this sense, e.g. for counting arguments, the two definitions are equivalent.

⁴ The indexing of Fibonacci words is not uniform in the literature, so one may find a shift by one or two in different papers. In particular, even and odd order Fibonacci words may be exchanged.

CA[i]	$\text{rot}^{\text{CA}[i]}(T)$	BWT[i]	$\pi_{\text{BWT}}[i]$	i
10	aaabracadabr	r	10	0
11	aabracadabra	a	0	1
7	abraaabracad	d	9	2
0	abracadabraa	a	1	3
3	acadabraaabr	r	11	4
5	adabraaabraca	c	8	5
8	braaabracada	a	2	6
1	bracadabraaa	a	3	7
4	cadabraaabra	a	4	8
6	dabraaabraca	a	5	9
9	raaabracadab	b	6	10
2	racadabraaab	b	7	11

i	0	1	2	3	4	5	6	7	8	9	10	11
T[i]	a	b	r	a	c	a	d	a	b	r	a	a

■ **Figure 1** The BWT of the text $T = \text{abracadabraa}$ with its standard permutation $\pi_{\text{BWT}}[i]$.

2.1 The BWT and the BBWT

The *Burrows-Wheeler Transform* (BWT) of a string T is a permutation of the characters of T . Consider the *multiset* of rotations of T , $\{\text{rot}^i(T) : 0 \leq i < n\}$. If $T = U^k$ for a primitive string U , then each element of $[T]$ will appear k times in this multiset. Now $\text{BWT}(T)$ can be obtained by concatenating the last characters of these strings when they are given in omega-order (or, equivalently in this case, in lexicographic order). For example, $\text{BWT}(\text{abracadabraa}) = \text{radarcaaaabb}$. See also Figure 1. The number of runs of the BWT of a string T is denoted $r(T)$, i.e. $r(T) = \rho(\text{BWT}(T))$. Thus, $r(\text{abracadabraa}) = 8$.

Let S be a string of length n . The *standard permutation* of S is a permutation π_S of the index set $\{0, \dots, n-1\}$ of S , defined as follows: $\pi_S(i) < \pi_S(j)$ if either $S[i] < S[j]$, or $S[i] = S[j]$ and $i < j$. In other words, $\pi_S(i)$ is the position of character $S[i]$ in the stable sort of the characters of S . For example, for $S = \text{radarcaaaabb}$, $\pi_S = [10, 0, 9, 1, 11, 8, 2, 3, 4, 5, 6, 7] = (0, 10, 6, 2, 9, 5, 8, 4, 11, 7, 3, 1)$, where we give π both in one-line notation and in cycle notation. When S is the BWT of some string, then the standard permutation of S is also known as LF-mapping [29], which can be used to recover the original string up to rotation. For example, given π_S as above, we can recover the Lyndon word T with $\text{BWT}(T) = S$, spelling it back-to-front, as follows: $T[n-1] = S[0] = \text{r}$, and for $i > 1$, $T[n-i] = S[\pi^i(0)]$. We get $T = \text{aaabracadabr}$, which is the Lyndon rotation of abracadabraa . The process can be modified to uniquely recover the input string by either adding the lexicographic rank of the rotation to be recovered, or by adding a unique end-of-string marker to the string.

Since all rotations of T have the same BWT, it is clear that the BWT is not a bijection on Σ^* . It is well known that a string S is the BWT of a primitive string if and only if π_S is cyclic. Likhomanov and Shur gave a more general characterization [50]: S is the BWT of some string if and only if the number of cycles of π_S equals the greatest common divisor of the lengths of the runs of S . This characterization encompasses powers, as well.

In fact, the number of strings of length n which are the BWT of some string (referred to as *BWT images* in [50]) equals the number of necklaces of length n . This follows from the definition of the BWT and the fact that the LF-mapping uniquely recovers the conjugacy class of the input string. In other words, the BWT is a bijection between necklaces of length

n and BWT images of length n . The number of necklaces of length n is $\sum_{d|n} Lyn(d, \sigma)$, where σ is the alphabet size and $Lyn(d, \sigma)$ is the number of Lyndon words of length d over an alphabet of size σ , or equivalently, the number of conjugacy classes of primitive words of length d . This can be seen by a counting argument on necklaces, distinguishing the case where the necklace is primitive (of which there are $Lyn(n, \sigma)$ many), and the case where it is a power U^k , with U primitive (in which case $d = |U|$ is a divisor of n). It is known [51] that $Lyn(d, \sigma) = \frac{1}{d} \sum_{d'|d} \mu(d') \sigma^{\frac{d}{d'}}$, where μ is the Möbius function defined as: $\mu(1) = 1$, $\mu(n) = (-1)^j$ if n is the product of j distinct primes, or 0 otherwise (n is divisible by a square number). Note that for all n , the number of necklaces of length n is at least σ^n/n , with equality if and only if n is prime.⁵

The extended BWT (eBWT) [54] is essentially⁶ a bijection between multisets of Lyndon words of total length n and strings of length n . It is defined as the concatenation of the last characters of the rotations of the input strings, given in omega-order. Underlying the eBWT is the observation that the BWT is a special case of the inverse of the Gessel-Reutenauer bijection [31] between permutations of a given type and descent set and strings of a given type (see [21] for more details).

The eBWT is a straightforward generalization of the BWT in that for any primitive string T , $\text{eBWT}(\{T\}) = \text{BWT}(T)$. Given a string S , the standard permutation π_S of S (i.e., its LF-mapping) can be used to construct a multiset $\mathcal{M}$ with eBWT equal to S in the same way as before, yielding one string per cycle of π_S . For more details on the eBWT, see [18] in this volume. Note that the eBWT is surjective: every string is the eBWT of something. However, the preimage is not necessarily a string; in general, it is a multiset of strings.

The *bijective BWT* (BBWT), on the other hand, is a variant of the BWT which is a bijection from Σ^* to Σ^* : every string is the BBWT of some string (of the same length).

► **Definition 1** (Bijective BWT). *The Bijective BWT, $\text{BBWT}(T)$, of a string T is the string obtained by concatenating the last character of the rotations of the multiset of the Lyndon factors of T , in ω -order of the rotations.*

The BBWT is thus the eBWT of the multiset of Lyndon factors of T (see Example 2). The inverse of the BBWT is constructed as follows: Take a string S , and let $\{L_1, \dots, L_m\}$ be the multiset of Lyndon words that we get via the LF-mapping from S , i.e., one string per cycle of π_S . Now concatenate the strings L_i in non-increasing order. This yields the unique string T whose BBWT is S .

We define the *generalized conjugate array* GCA of a string T as follows: $\text{GCA}[j] = i$ if the conjugate of L_k starting in position i is j th in omega-order among all conjugates of the Lyndon factors of T , where L_k is the Lyndon factor containing position i of T .

► **Example 2.** The BBWT of the 6th Fibonacci word is given by $\text{BBWT}(\text{abaababaabaab}) = \text{bbbaababaaaaa}$. The Lyndon factorization of abaababaabaab is $(\text{ab})^1(\text{aabab})^1(\text{aab})^2$. See Figure 2 for a visualization. We can recover the original string as follows. The standard permutation of bbbaababaaaaa is $[8, 9, 10, 0, 1, 11, 2, 12, 3, 4, 5, 6, 7] = (8, 3, 0)(1, 9, 4)(2, 10, 5, 11, 6)(12, 7)$. From this, we get: $\text{aab}, \text{aab}, \text{aabab}, \text{ab}$. Sorting these Lyndon words yields $\text{ab} \preceq_{\omega} \text{aabab} \preceq_{\omega} \text{aab} \preceq_{\omega} \text{aab}$, and thus, we get $T = \text{ab} \cdot \text{aabab} \cdot \text{aab} \cdot \text{aab}$.

⁵ It is sometimes imprecisely stated, e.g. [32, 47], that $(1/n)$ th of all strings are BWT images, but this is only true for n prime.

⁶ The original definition of eBWT does not assume the strings to be Lyndon words, but stores information concerning the rotations of the input strings.

CA[i]	rot ^{CA[i]} (T)	BWT[i]	i	GCA[i]	BBWT[i]
7	aabaababaabab	b	0	7 aab	b
2	aababaabaabab	b	1	10 aab	b
10	aababaababaab	b	2	2 aabab	b
5	abaabaababaab	b	3	8 aba	a
0	abaababaabaab	b	4	11 aba	a
8	abaababaababa	a	5	5 abaab	b
3	ababaabaababa	a	6	3 ababa	a
11	ababaababaaba	a	7	0 ab	b
6	baabaababaaba	a	8	9 baa	a
1	baababaabaaba	a	9	12 baa	a
9	baababaababaa	a	10	6 baaba	a
4	babaabaababaa	a	11	4 babaa	a
12	babaababaabaa	a	12	1 ba	a

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	a	b	a	a	b	a	b	a	a	b	a	a	b

■ **Figure 2** The BWT (left) and BBWT (right) for the text $T = \text{abaababaabaab}$, with Lyndon factorization $(ab)^1(aabab)^1(aab)^2$. We have $\text{BWT}(T) = \text{bbbbbaaaaaaa}$ and $\text{BBWT}(T) = \text{bbbaababaaaaa}$.

Notice that the standard BWT can be considered as a special case of BBWT since $\text{BBWT}(\text{minrot}(T)) = \text{BWT}(T)$. Indeed, the two transforms coincide exactly on the set of necklaces:

► **Proposition 3** ([12, Proposition 1]). $\text{BWT}(T) = \text{BBWT}(T)$ if and only if T is a necklace.

If we fix the inverse BWT to be always the necklace representative of the conjugacy class, then one can consider the BWT as a bijection from a subset of strings (necklaces) to a subset of strings (BWT images). The BBWT thus is a generalization of the BWT, since it is a bijection from Σ^* to Σ^* and coincides with the BWT on this subset.

Similarly to the BWT, we will be interested in the number of runs of the BBWT of a string T , denoted $r_B(T)$, i.e. $r_B(T) = \rho(\text{BBWT}(T))$. Thus, $r_B(\text{abaababaabaab}) = 6$.

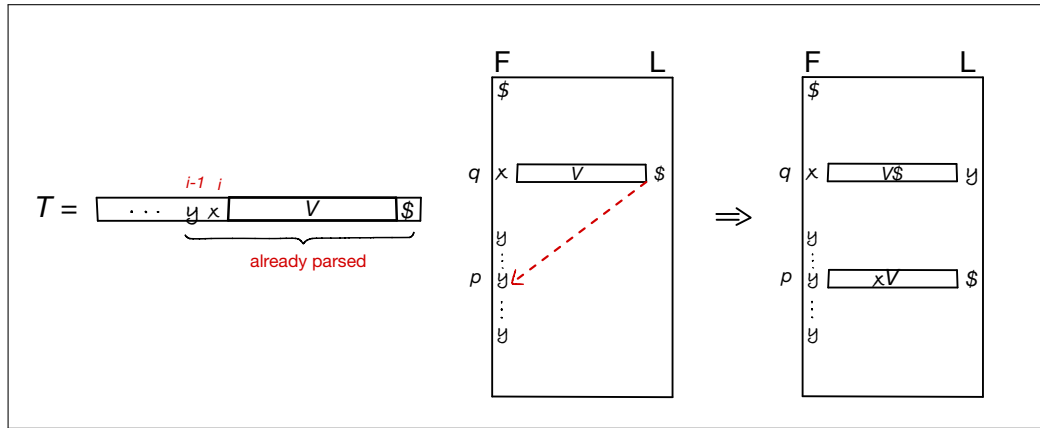
In the rest of the paper, n denotes the length of the string under investigation (if no misunderstanding is possible).

3 Construction

Gil and Scott [32] sketched an algorithm based on the linear-time suffix array construction algorithm DC3 by Kärkkäinen, Sanders, and Burkhardt [41], for which they claim that the sorting “shall require $O(n \lg n)$ time”. Unfortunately, the authors seem not to address the soundness of their algorithm nor provide source or pseudo code to give practical evidence that their approach indeed computes the BBWT. Mantaci et al. [54, before Proposition 10] conjectured that a generalization of DC3 should make it possible to compute the eBWT (and hence the BBWT) in linear time.

3.1 Dynamic construction

The first analysis of a BBWT construction algorithm with sound time bounds is due to Bannai et al. [6], who draw a connection to the algorithm of [14] constructing the eBWT,



■ **Figure 3** Online construction of the BWT on the reversed input text T described at the beginning of Section 3.1. Say that $T[i] = x$ and $T[i - 1] = y$, and our task is to update the BWT of $T[i..]$ to the BWT of $T[i - 1..]$. For that, it suffices to add the conjugate $yxV\$$ into the BWT matrix, where V is the suffix $T[i + 1..n - 2]$ after x up to the last character $\$$. Suppose that q is the BWT position storing $\$$, cf. the bottom left figure showing F and L , where L is the BWT and F the array storing the character of L sorted. A backward search step with character y from q gives a suffix starting with y and having the longest common prefix with $yxV\$$. To make the LF mapping jump from q to p , we exchange $\text{BWT}[q] = y$. To close the cycle, we insert $\$$ at position p . This gives the L array (i.e., the BWT of $T[i - 1..]$) on the bottom right. We can verify that the introduced conjugate at BWT position p is in the lexicographic order due to how the backward search works.

and were able to adapt this algorithm for computing the BBWT in $O(n \log n / \log \log n)$ time. This BBWT construction algorithm works in an *online* fashion, i.e., it builds the BBWT per Lyndon factor, starting with the first one, and then incrementally indexes each further factor until adding the last Lyndon factor. Compared to the BWT, there is no known algorithm computing the BWT in an online model that allows reading a specific prefix of the text (such as its Lyndon factors). However, it is possible to construct the BWT online by reading the text from the reverse order (e.g., [22]) – a setting for which we do not know whether a solution for the BBWT exists. The main idea of the BWT construction algorithm is that it computes the BWT of $T\$$, where $\$$ is a sentinel character smaller than all other characters appearing in T . The construction starts with BWT just storing $\$$, and at any time later, the BWT contains exactly one $\$$ at a specific position. By the definition of the LF mapping, going from this position backwards would give the last character of T . Now, assume that we have built the BWT of $T[i..n - 1]$ and want to insert $T[i - 1]$. For that, we perform a backward search step from the position of $\$$ for the character $T[i - 1]$. Say we end up at a position p , then we exchange $\$$ with $T[i - 1]$ but insert $\$$ at position p and recurse, see Figure 3 for a visualization. The same trick works without $\$$ for the BBWT if we process the Lyndon factors sequentially, starting from the leftmost and lexicographically largest one. This is because, when inserting a new Lyndon factor L_x , we know that it is lexicographically smaller than all other Lyndon factors. Hence, we can insert the last character c of L_x at the first position in the BBWT and backward search c to find the insertion position of the previous character in L_x . This gives the following complexities.

► **Theorem 4** ([6, Theorem 22]). *We can construct the BBWT of a string of length n in $O(n \log n / \log \log n)$ time in an online model, where we receive at each step one Lyndon factor of the input.*

3.2 Linear-time algorithms

In this section, we will sketch the two existing linear-time algorithms for BBWT construction. Both algorithms are non-trivial modifications of linear-time suffix array construction algorithms. The first, due to Bannai et al. [7] (2021), is based on the well-known SAIS-algorithm of Nong et al. [63]. The second, due to Olbrich et al. [64] is a modification of the algorithm GSACA by Baier [3]. In what follows, we will sketch the proofs of the following theorem:

► **Theorem 5** (Linear time construction of BBWT [7, 8, 15, 64]). *We can construct the BBWT of a string of length n in $O(n)$ time.*

In the case of a string that is terminated by an end-of-string character, the BWT can be computed in linear time from the suffix array, by noticing that $\text{BWT}[i] = T[n-1]$ if $\text{SA}[i] = 0$, and $\text{BWT}[i] = T[\text{SA}[i]-1]$ otherwise. This, however, uses the fact that the last suffix is the smallest among all suffixes, which is not necessarily the case in general. In actual fact, the conjugate array CA would be needed, since for all $0 \leq i \leq n-1$, we have $\text{BWT}[i] = T[\text{CA}[i]-1]$, where the subtraction is modulo n . One solution is to first compute the Lyndon rotation of T because for Lyndon words, the suffix array and the conjugate array coincide. The first algorithm that directly computes the BWT of a string T without an end-of-string character, and without computing Lyndon rotations, was given in [15] (and was inspired by the SAIS-based algorithm [7] we are about to see).

For the BBWT, we need is the GCA of T , due to the following connection:

$$\text{BBWT}[i] = \begin{cases} T[\text{GCA}[i] + |L_k| - 1] & \text{if GCA}[i] \text{ is the starting position of a Lyndon factor } L_k, \\ T[\text{GCA}[i] - 1] & \text{otherwise.} \end{cases}$$

The two linear-time algorithms below take advantage of this idea by modifying known SA-construction algorithms to construct the GCA instead of the SA.

3.2.1 A SAIS-based algorithm

The first $O(n)$ -time construction algorithm of the BBWT was presented in [7, 8]. It is an adaptation of the SAIS (Suffix Array by Induced Sorting) algorithm [63], a recursive linear-time algorithm. The algorithm of [7, 8] was then adapted and simplified for eBWT construction by Boucher et al. [15]. Some of the simplifications are also applicable for BBWT construction, so our presentation here is a mixture of the two.

We first give a quick recap of SAIS. The SAIS algorithm first categorizes each text position i in L - or S -type, according to whether the suffix $T[i..]$ is larger (L) or smaller (S) than the next suffix $T[i+1..]$. S -type positions following L -type positions are called LMS -positions (for “leftmost S ”). The recursive step is done on a new text, where each so-called LMS -substring of T is replaced by a meta-character; LMS -substrings are minimal substrings containing two consecutive LMS -positions. The meta-characters are assigned to the LMS -substrings respecting their LMS -order $<_{LMS}$, a slight modification of the lexicographic order.⁷ For a suffix i of T let U_i be its LMS -prefix, i.e., U_i is the prefix of $T[i..]$ that is a suffix of the LMS -substring in which i lies (and therefore, the full LMS -substring if i is an LMS -position). Nong et al. [63] show that if $U_i <_{LMS} U_j$ then $T[i..n] <_{\text{lex}} T[j..n]$, while if

⁷ Even though the original definition is more complex, it can be shown that for two LMS -substrings U, V : $U <_{LMS} V$ if either V is a proper prefix of U , or neither is a prefix of the other, and $U <_{\text{lex}} V$.

$U_i = U_j$, then the order is determined by the subsequent LMS-substrings. This is the basis of the recursive step. The recursion terminates when all meta-characters are distinct, at which point the suffix array of the meta-text on the current recursion depth can be trivially computed. The suffix array of the meta-text implies the relative order of the suffixes starting at LMS-positions in T , which is then used to induce the SA-positions of the remaining suffixes in two linear passes over SA, the first left-to-right, inducing L -type positions, the second right-to-left, inducing S -type positions.

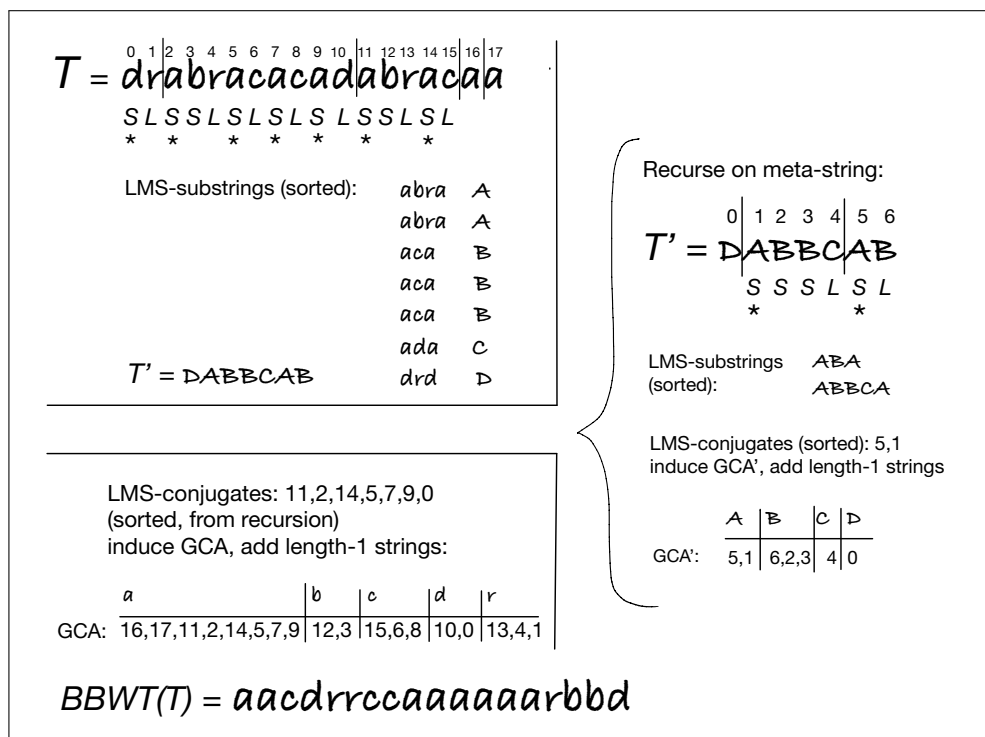
This algorithm has to be adapted for (a) computing the omega-order, rather than the lexicographic order, and (b) between conjugates of different strings, rather than suffixes of the same string.

Let us first assume that there are no Lyndon factors of length one. The first modification is that the position types must be defined cyclically within the Lyndon factors. This can be done in linear time, similarly to the SAIS algorithm, using the fact that the smallest rotation of Lyndon words start in their first position. Second, it can be shown that, using the cyclically defined types, the LMS-substrings and LMS-prefixes, defined cyclically, have a similar property: $U_i <_{LMS} U_j$ then $T[i..n] <_{\omega} T[j..n]$, while if $U_i = U_j$, then the order is determined by the subsequent LMS-substrings [15, Lemma 4]. This again enables recursion, using the additional insight that the starting positions of Lyndon factors constitute a subset of the starting positions of the LMS substrings [8, Lemma 3.4]. This implies that the meta-string's Lyndon factorization is the one induced by the Lyndon factorization of T . Finally, the inducing step (inducing the GCA using the relative order of the LMS-conjugates) can be done analogously to SAIS, because here, too, all L -type conjugates come before all S -type conjugates in the same bucket.

It can also be shown that length-1 strings are always placed between L - and S -type conjugates in the same bucket. Accordingly, the algorithm removes length-1 Lyndon factors at the beginning and places them in the correct place at the end, when the rest of the GCA has been filled in. For an example, see Example 6 and Figure 4.

Finally, Boucher et al. [15] showed that essentially the same algorithm works on *any* multiset of input strings, given in *any* order – their work targets the eBWT construction, where the input strings are not assumed to be Lyndon words, nor is it assumed that they are given in a relative order. The main algorithmic change in that case is in how the type assignment is done (scanning each string at most twice).

► **Example 6.** We give an example with a recursive call in Figure 4, on the string $T = \text{drabracacacadabraca}$. We start on the upper left box with the classification of the text positions of T into S - and L -types, omitting positions that correspond to Lyndon factors of length 1. Each S succeeding an L (considered cyclically) gets a star ($\star$) to denote an LMS-type position. The Lyndon factorization of T is denoted by vertical bars between the characters of T . Next we replace the LMS-substrings by meta-characters $A < B < C < D$, according to their rank in LMS-order, to form the meta-text based on these meta-characters. Since not all characters of T' are distinct, we recurse on T' (right side); here, all LMS conjugates are distinct such that we can immediately start the inducing step. We insert the entry for $T'[0] = D$ at the end of the recursive call at the end of the D-bucket of GCA' (there being no S -type conjugates for the largest character). GCA' translates back to give the relative order of the LMS-conjugates of T , which is then used to induce the entire GCA, in the usual way, except that i induces $i - 1$ cyclically w.r.t. the factor it is in. Finally, the entries for $T[16] = T[17] = a$ are inserted at the beginning of the a-bucket of GCA at the end of the execution (there being no L -type conjugates for the smallest character). We give the final BBWT(T) at the bottom of the figure.



■ **Figure 4** Construction of the GCA via a modification of the SAIS algorithm as explained in Section 3.2.1. See Example 6 for details.

3.2.2 A GSACA-based algorithm

Baier [3] in 2016 presented the first non-recursive linear-time suffix array construction algorithm, called GSACA. The algorithm came with an implementation, which was, however, not competitive with the best practical suffix construction tools. A modified version by Bertram et al. [10] later proved to be very efficient practically, as well. The original GSACA algorithm was then adapted for eBWT and BBWT construction by Olbrich et al. [64].

As was shown in [30], the GSACA algorithm uses Lyndon prefixes to compute a partial order of the suffixes. Given a string X , its *Lyndon prefix* is the longest prefix U which is a Lyndon word, which is also the first factor in its Lyndon factorization [28]. For example, the Lyndon prefix of *abracadabraa* is *abracad*. GSACA first groups the suffixes according to their Lyndon prefixes (Phase 1). This order then gets refined by sorting the suffixes within the same group (Phase 2). Since the groups appear in lexicographic order according to the groups' Lyndon prefixes, this yields the final suffix array. The crucial observation is that if U, V are the Lyndon prefixes of strings X, Y , respectively, then $U <_{\text{lex}} V$ implies $X <_{\text{lex}} Y$.

Astonishingly, the distribution of the suffixes into their groups can be done in one single right-to-left scan. First the suffixes are partitioned into groups according to their first character; these are necessarily the first characters of their Lyndon prefixes. This initial partition is then refined by removing elements from groups and extending their labels to longer prefixes, where the label of a group is a common prefix shared among all group elements. At the end of this scan, each element will be in a group whose label equals the suffix's Lyndon prefix. This is done as follows: proceeding from the highest group

towards the lowest (i.e., right to left), for every suffix i in the current group, we will take another suffix $j = \text{prev}(i)$ from a lower group, if such a j exists, and insert it into a new group right after its current group, with its label extended by the label of i 's group. Here, $\text{prev}(i) = \max\{i' < i : i' \text{ is in a lower group than } i\}$. See Figure 5 for an example. This completes Phase 1.

Now, in Phase 2, the correct order within each group is induced by starting from the smallest suffix, which in the original algorithm, is the end-of-string character at position $n - 1$. In the modified algorithm for computing the BBWT, we need to take care of the circular manner of inducing within Lyndon factors. Note that with the Lyndon grouping from Phase 1, we also have the Lyndon factorization of T , so we can use the first position of each Lyndon factor to start the inducing step.

► **Example 7.** We give an example on the same string $T = \text{drabracacadabracaa}$ as in Example 6, see Figure 5. In Phase 1, we first place all suffixes in a group according to their first character. We then start with the highest group, the one with label **r**: for the element 1, $\text{prev}(1) = 0$, which is in group **d**. We remove 0 from its group and move it into a new group, immediately after, with label **dr**, which is the concatenation of the label of 0's group (**d**) and the label of 1's group (**r**). Similarly, 4 and 13 cause the removal of 3 resp. 12 from their initial group **b** and are inserted into a new group, immediately after, with label **br**. Then we move to the group **dr**, whose only element 0 does not have any element to induce. 10 in group **d** induces the removal of 9 and its insertion into the new group **ad**, etc. The resulting Lyndon grouping is shown at the bottom left of the figure. In Phase 2, each group is sorted, via induced sorting with one pass from left to right. In (a) we show the result of the original algorithm constructing the SA of $T\$$, from which the BWT can be computed. (Note that in that case, the algorithm assumes that T has a final dollar-symbol.) In (b) we show the result if the inducing is done w.r.t. Lyndon factors, to get the GCA and thus the BBWT. Differences between the two must necessarily lie within Lyndon groups. For example, note the difference in the last group (labeled **r**), where 1 is now placed after the other two elements because it is followed by **d** in its Lyndon factor **dr** (and not by **a** as in T). We give $\text{BBWT}(T)$ at the bottom of the figure.

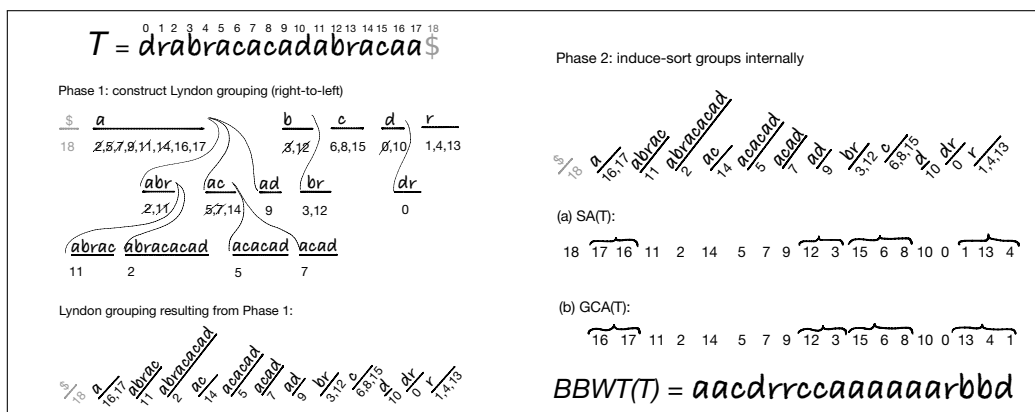
3.3 In-place construction

Köppl et al. [46] discovered that the BBWT construction algorithm of Theorem 4, which incrementally builds the BBWT from the text by one Lyndon factor at a time, can be used in the setting of in-place construction, where only $O(1)$ additional working space is allowed to turn T into its BBWT. Here, the main observation is that the in-place construction algorithm of [22] for the BWT can be changed to compute the BBWT instead of the BWT. That is because we can compute the Lyndon factorization with Duval's algorithm with constant space while constructing the BBWT. In detail, we let Duval's algorithm detect the next Lyndon factor, then pause it to consume this factor, and recurse by resuming Duval's algorithm.

► **Theorem 8** (in-place construction [46]). *We can compute the BBWT in $O(n^2)$ time using $O(1)$ words of working space.*

For restoring the original text from the BBWT, there is a need for supporting the FL-mapping, the inverse of the LF-mapping. The FL-mapping can be implemented by a select query on BBWT, which however needs $O(n^{1+\varepsilon})$ time for a solution with constant working space [59] for a constant $\varepsilon > 0$.

► **Theorem 9** ([46]). *We can invert the BBWT in $O(n^{2+\varepsilon})$ time using $O(1)$ words of working space for a constant $\varepsilon > 0$.*



■ **Figure 5** Construction of the BBWT using the GSACA-based algorithm, see Section 3.2.2. (a) shows the internal order of the groups we would get if we wanted to compute the SA, for the BWT of T ; (b) shows the order we get for GCA, needed to compute the BBWT of T . Notice the difference in the groups labeled **a** and **r**. (As the dollar-symbol is needed only for the SA-algorithm, it and its group are marked in grey.) See Example 7 for more details.

A more involved analysis revealed that it is even possible to turn BWT into BBWT directly without inversion using $O(n^{2+\epsilon})$ time and $O(1)$ working space. The idea is to run Duval's algorithm using FL-mappings to detect the Lyndon factorization. Since Duval's algorithm uses three pointers to text positions, we can simulate the pointers by pointers into the BWT and use the FL-mapping to advance a pointer. When a Lyndon factor has been detected, we try to move $\$$ such that the Lyndon factor has its own orbit in the standard permutation of the BBWT. However, this change can also break the cycle of the remaining BWT, for which additional moves have to be performed.

The authors also gave an algorithm for computing the run-length compressed BBWT online, Lyndon factor by Lyndon factor. Similar to computing the run-length compressed BWT online reading the reversed text, the authors also show how to compute the run-length compressed BBWT in $O(r_B)$ space and $O(n \log n / \log \log n)$ time.

3.4 Implementations

On the practical side, we are aware of the implementations by David A. Scott⁸, Branden Brown⁹, Yuta Mori in his OpenBWT library¹⁰, and of Neal Burns¹¹. While the first two are naive but easily understandable implementations calling general sorting algorithms on all conjugates to directly compute the BBWT, the implementation of Yuta Mori seems to be an adaptation of the SAIS algorithm to induce the BBWT. The implementation of Neal Burns takes an already computed suffix array SA as input, and transforms SA into GCA by shifting values to the right until they fit. Hence, the running time is based on the lengths of these shifts, which can be $O(n^2)$, but seem to be negligible in practice for common texts.

Finally, the two linear-time algorithms we presented, due to Bannai et al. [8]¹² and Olbrich et al. [64]¹³, have both been implemented.

⁸ <https://web.archive.org/web/20210615215623/http://bijective.dogma.net/bwts.zip>

⁹ <https://github.com/zephyrtronium/bwst>

¹⁰ <https://web.archive.org/web/20170306035431/https://encode.ru/attachment.php?attachmentid=959&d=1249146089>

¹¹ <https://github.com/NealB/Bijective-BWT>

¹² <https://github.com/mmpiatkowski/bbwt>

¹³ <https://gitlab.com/qwerzuiop/lfgsaca>

4 Indexing

The eBWT is a generalization of the BWT for multiple input texts. As such, it is possible to build an FM-index-like data structure upon the eBWT. Indeed, a more advanced compressed text index built upon the eBWT, the so-called *extended r-index*, was recently presented in [16], which can efficiently answer pattern matching queries while requiring only $O(r)$ space.

The reason this kind of index is not immediately applicable for the BBWT is because, by interpreting the BBWT as the multiset of Lyndon factors of T , we would be restricted to patterns that occur within the Lyndon factors. In order to use the BBWT for an index for regular pattern matching on the original text, we must (a) remove superfluous matches that match the Lyndon factor in a cyclic sense but do not occur in the original string, and (b) find a way to detect matches that span multiple Lyndon factors, as these are not found by the standard backward search on the BBWT.

Such an index was presented in [6, 8]. We present this index under the facilitated setting that all Lyndon factors of T are distinct, i.e., $T = L_1 \cdots L_{\ell(T)}$ with $e_1 = \dots = e_{\ell(T)} = 1$. This imposes not a restriction because we first detect the actual powers $e_1, \dots, e_{\ell(T)}$ in a precomputation step before discarding duplicate Lyndon factors. After building the index we will introduce, we augment it such that the correct numbers are counted for each Lyndon factor [8, after Theorem 4.7]. We also limit our survey to counting queries, i.e., to return the number of occurrences of a pattern in the text.

The index we are going to present works like the FM-index. The only difference is that we need to take action whenever the LF-mapping interval contains a position corresponding to the starting position of a Lyndon factor. Otherwise, it applies the backward search step in exactly the same way as the FM-index does. This is because a Lyndon word cannot cross two Lyndon factors of the text, meaning that any pattern that is a Lyndon word cannot be the suffix of some L_k and the prefix of some L_{k+1} , since necessarily $L_k \succ L_{k+1}$. Consequently, a Lyndon pattern can be found with the same techniques as used by the FM-index since it can only occur inside a Lyndon factor of the text. For finding *all* patterns P , we use the Lyndon factorization of P —by our observation, we can be assured to match at least the last factor of P correctly.

By the above observation, we are sure that we can find the last Lyndon factor of P by the backward search without any modification. However, a backward search step of the preceding character may already give unexpected answers if this last factor is a prefix of a factor of the text. For instance, for the particular case that the pattern ends with L_k , the LF-mapping may report an occurrence of this pattern (regardless of the remaining prefix) because the LF-mapping extracts the original cyclic Lyndon words by iterative application. Hence, only at those backward search steps that read the last character of a previous Lyndon factor of the pattern, we can observe phenomena that need to be taken care of, namely:

- a superfluous match:** caused by matching with the same Lyndon factor L_k of the text, and
- a missed match:** caused by the fact that we do not check the characters starting before L_k in text order for a potential occurrence.

Both phenomena happen at most p' times, where p' is the number of Lyndon factors of P . Each time, we need to take care of at most one superfluous and one missed match. We track these individual matches by mapping them individually with the LF-mapping such that the final cost for matching P are $O(|P|p')$ rank queries. To get the final result, we note that after the longest so-called *significant suffix* [39] of P has been matched, there is only one match remaining to keep track of. Since the longest significant suffix can contain $O(\log |P|)$ different Lyndon factors, we obtain the following result.

► **Theorem 10** (Pattern matching using BBWT [6, 8]). *Given a text T and a pattern P of length m , we can compute all occurrences of P in T with the FM-index built on $\text{BBWT}(T)$ with $O(m \log m)$ rank/select operations.*

The rank/select operations in Theorem 10 are not only necessary for the backward search, but for a bit vector B with rank/select support marking the positions in BBWT corresponding to the distinct Lyndon factors.

The BBWT can also be run-length compressed into $O(r_B)$ space while still supporting count queries. For that, the bit vector B can be entropy-compressed [8].

5 Compression

The BWT was originally introduced for compression: Burrows and Wheeler state that the BWT “tends to group characters together so that the probability of finding a character close to another instance of the same character is increased substantially” [17]. An intuitive reason to why this is so, is that since the rotations of the string are sorted in lexicographic (or ω -)order, the BWT is a sequence of symbols that precede substrings with a similar context. It has been shown that achieving zero order entropy compression on appropriately partitioned blocks of the BWT achieves higher order entropy compression of the original string [57]. Thus, a string can be compressed by first applying BWT, and then applying some simple compression algorithms such as Move-To-Front encoding or run-length encoding.

While the analysis of the compression performance of the BWT has received much attention, the compression performance of the BBWT is less well studied. Scott [66] and Gil & Scott [32] reported the compression performance of BWT and BBWT on the Large Calgary corpus [9], often used as a benchmark dataset. On all but one of the 18 files, the BBWT resulted in a slightly better compression than the BWT: the average gain was around 1% of the original text size. These data sets are very small by today’s standards. We only know of empirical evaluations on larger data sets which compare the number of runs in the BWT and BBWT for various data sets (the Calgary Corpus, Canterbury Corpus, Pizza&Chili Corpus, and Silesia Corpus), where it is reported that they are roughly equal [8].

Recently, it has become recognized that, for *highly repetitive data* that are prevalent today in many real-world applications, analyses of compressors in terms of entropy measures, i.e., on statistical properties of the data, are not useful as they do not capture the repetitiveness of the data very well [61]. Instead, a trend is to analyze the compression performance of compressors in terms of the sizes of compressed representations of the individual strings, using representations, in particular, based on *dictionary compression*, that can better model the repetitiveness of the data.

The systematic study of *repetitiveness measures* related to dictionary compression was initiated by Kempa and Prezza [44]. Dictionary compression is a family of compressed representations that are essentially based on copy and paste operations, and include representations, for example, those computed by well-known algorithms like LZ77, grammar-based representations (e.g., LZ78, RE-PAIR), run-length encoding, and run-length encoded BWT (RLBWT). See [61] for a comprehensive survey on the subject.

In the following, we introduce results focused on theoretical analyses on the compressiveness of BBWT. Recall that, for a string T , $r(T)$ denotes the number of runs of $\text{BWT}(T)$ and $r_B(T)$ the number of runs of $\text{BBWT}(T)$.

5.1 r_B as a measure of dictionary compression

The Run-Length compressed BWT (RLBWT) [53] is a representation of T of size $r(T)$. From its definition, it is not clear why the RLBWT can be considered as a form of dictionary compression, since the transform is applied before the run-length encoding. However, suppose for each run in the BWT, we define the reference (copying) of a symbol to point to the previous symbol in the run (except for the first one in the run). Then, if we consider all of these references on corresponding positions of the text, the text can be partitioned into at most $2r(T) + 1$ phrases, where each phrase is either a single symbol that corresponds to the beginning of a run, or, a maximal substring such that the offsets of the references are the same for all positions in the phrase [62, Theorem 9]¹⁴. The latter phrases can then be encoded by two integers – the length and offset, resulting in a representation for T of size $O(r(T))$, known as bidirectional macro schemes (BMSs) [70], the most general (and most powerful) representation in dictionary compression.

The above relation can be shown as follows: Suppose two lexicographically adjacent rotations $\text{rot}^{i+1}(T)$ and $\text{rot}^{j+1}(T)$ have the same BWT symbol (i.e., $T[i] = T[j]$). Then, position j will reference position i , with offset $i - j$. Due to the property of LF mapping, $\text{rot}^i(T)$ and $\text{rot}^j(T)$ will also be adjacent rotations, and, if they have the same BWT symbol as well, position $j - 1$ will reference position $i - 1$, with offset $(i - 1) - (j - 1) = i - j$ and therefore have the same offset. This continues until the adjacent rotations do not have the same BWT symbol, which can happen at most $r(T) - 1$ times – at boundaries of runs. A minor point we have overlooked above is that the offsets $i - j$ and $(i - 1) - (j - 1)$ would always be the same in the cyclic sense, but they can become different in the linear sense when $i = 0$ or $j = 0$, since position -1 wraps around to position $n - 1$. This can happen at most twice. Since there are exactly $r(T)$ single symbol phrases, the total number of phrases is at most $(r(T) - 1) + 2 + r(T) = 2r(T) + 1$.

Badkobeh et al. [2] showed that for the Run-Length compressed BBWT (RLBBWT), a similar BMS of size $O(r_B(T))$ can be induced. An important observation to achieve this bound is that $r_B(T)$ can be lower bounded by the number of distinct Lyndon factors $\ell(T)$ of the Lyndon factorization of T [16, 2]. This follows from [16, Corollary 9], which states that:

► **Lemma 11** ([16, Corollary 9]). *Let $\mathcal{M} = \{T_1, \dots, T_m\}$ 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 $N/r \leq l$.*

Since, in the case of BBWT, $\mathcal{M} = \{L_1, \dots, L_{\ell(T)}\}$ where all the Lyndon factors are distinct and primitive, we have:

► **Corollary 12** ([2]). *For any string T , $\ell(T) \leq r_B(T)$.*

Applying the same referencing of symbols for BBWT as BWT described above, the same arguments hold, except that there can be more instances where the references wrap around, i.e., when i or j corresponds to the beginning of a Lyndon factor. However, this is at most twice for each distinct Lyndon factor, since identical Lyndon factors occur adjacently. Therefore, by Corollary 12, the total number of phrases can be bounded by $O(r_B(T) + \ell(T)) = O(r_B(T))$.

► **Theorem 13** ([2, Lemma 1]). *For any string T , there exists a BMS of size $O(r_B(T))$ that represents T .*

¹⁴We note that [62] assumes that strings are terminated with a \$ and the proved bound is $2r(T)$. For strings that are not terminated by \$, there can be $2r(T) + 1$ phrases, e.g., with $T = ababaaaabab$.

Therefore, RLBWT and RLBBWT respectively induce BMSs of size $O(r(T))$ and $O(r_B(T))$, where all references point to a smaller (omega-order) position, and can be considered as a form of dictionary compression.

5.2 Relation to LZ77

Denote by $z(T)$ the size of the LZ77¹⁵ factorization [48] of T . It is known that $z(T)$ is the size of a smallest BMS where all references point to a smaller (text-order) position. The question of whether $r(T)$ can be bounded in terms of $z(T)$ was answered by Kempa and Kociumaka [43, Theorem 3.2], who showed that for any string T , $r(T) = O(z(T) \log^2 n)$ holds.

Badkobeh et al. [2] showed that this proof can be extended to the case of $r_B(T)$. The term $z(T)$ appears in Kempa and Kociumaka's proof when considering, for given k , the set T_k of all substrings of T^ω of length k , where $|T_k|$ can be bounded by $O(z(T)k)$ since any substring of T^ω must have an occurrence containing the last position of an LZ77 phrase (including the last phrase). The main difference in the proof is in the definition of the set T_k , which should be modified to be the set of all substrings of length k of $\{(L_1)^\omega, \dots, (L_{\ell(T)})^\omega\}$, rather than T^ω . This can be bounded by $O(z(T)k + \ell(T)k)$, since such a substring is either a substring of T , or, is a substring of $(L_i)^\omega$ that is not a substring of L_i . $|T_k| = O(z(T)k)$ is obtained by applying the result by Urabe et al. [71] that shows $\ell(T) \leq 4z(T)$. The rest of the proof is essentially the same, giving:

► **Theorem 14** ([2, Theorem 1]). *For any string T , $r_B(T) = O(z(T) \log^2 n)$.*

For BWT, a tighter bound of $r(T) = O(\delta(T) \log^2 n)$ where $\delta(T) = \max_{1 \leq d \leq n} \{T_d/d\}$ follows easily, since $\delta(T) \geq |T_k|/k$, $|T_k| = O(\delta(T)k)$. Kempa and Kociumaka further show a bound of $r(T) = O(\delta \log \delta \max\{1, \log \frac{n}{\delta \log \delta}\})$ [43, Theorem 3.7]. We note that $\delta(T)$ is known to lower-bound (and can be strictly smaller than) all dictionary compression measures [45]. Tighter bounds for $r_B(T)$ are not yet known.

5.3 Robustness of r_B

The following results are known about the change that r_B may undergo if some operation is applied to T .

5.3.1 String reversal

Giuliani et al. [34] studied the robustness of r with respect to string reversal, i.e., how $r(T)$ and $r(T^{\text{rev}})$ can differ. They gave an infinite family of strings (which they called Fibonacci-plus words) for which $r(T) = O(1)$ and $r(T^{\text{rev}}) = \Theta(\log |T|)$, i.e., with a runs-ratio that is logarithmic in the length of the string. This lower bound is nearly tight, in view of the known upper bound of $r(T)/r(T^{\text{rev}}) = O(\log^2 |T|)$ [43]. Strings with higher runs-ratio than the family of [34] (albeit only experimentally) were given in [33].

Biagi et al. [11, 12] paralleled this result for the BBWT, giving a lower bound of $\Theta(\log n)$ on $r_B(T)/r_B(T^{\text{rev}})$.

► **Theorem 15** ([12]). *There exists an infinite family of strings such that $r_B(T) = O(1)$ and $r_B(T^{\text{rev}}) = \Theta(\log n)$.*

¹⁵We use the name LZ77 which is prevalent in the literature, though it has been mentioned that LZ76 is more precise [60].

The family $(T_k)_{k \in \mathbb{N}}$ provided in the proof of Theorem 15 are Lyndon rotations of Fibonacci words: On the one hand, these words have $r_B(T_k) = 2$, by Proposition 3 and Theorem 23. On the other, the authors show that $\text{BBWT}(T_k^{\text{rev}})$ has exactly $2(k-2)$ runs. This is done by studying the Lyndon factorization of T_k^{rev} : the authors show that it consists of the Lyndon rotations of Fibonacci words of order 0 to $k-2$, with Lyndon rotations of even order words, in increasing order, first, followed by Lyndon rotations of odd order words, in decreasing order, with the last Lyndon factor, $\mathbf{a}$, repeated twice. Therefore, $r_B(T_k^{\text{rev}})$ equals $e\text{BWT}(\mathcal{S}_{k-2})$, where $\mathcal{S}_j = \{F_i : i = 0, \dots, j\}$, since the BWT is invariant w.r.t. rotation and multiple occurrences of a string in a multiset do not impact on the number of runs of the eBWT [54]. The authors then give the exact form of $e\text{BWT}(\mathcal{S}_j)$, which is shown to contain exactly $2j$ runs.

In the same paper, experimental results were given both on $r_B(T)/r_B(T^{\text{rev}})$ (multiplicative difference) and on $r_B(T) - r_B(T^{\text{rev}})$ (additive difference), as well as on the connection between these two functions and the number $\ell(T)$ of distinct Lyndon factors.

5.3.2 Single character edits

A side result of [34] is that the BWT is sensitive to one-character edits, in the sense that there exist strings on which $r(T)/r(T') = \Omega(\log |T|)$, where T' results from T by appending one character to T . Akagi et al. [1] introduced the term *sensitivity* of a measure to denote the asymptotic change when a string undergoes single-character edits of different types. This was studied in detail in [35, 36] and logarithmic lower bounds were given for the multiplicative sensitivity $r(T)/r(T')$, and an $\Omega(\sqrt{n})$ bound for the additive sensitivity, $r(T) - r(T')$.

Jeon and Köppl [40] showed that this result can be translated to the BBWT with the same bounds by studying, among others, the Lyndon rotations of Fibonacci words and applying Corollary 12.

► **Theorem 16** ([40]). *There is a family of strings $(T_n)_{n \geq 1}$ with $r_B(T_n)/r_B(T'_n) = \Theta(\log n)$ and a family of strings $(S_n)_{n \geq 1}$ with $r_B(S_n) - r_B(S'_n) = \Theta(\sqrt{n})$ such that $|T_n| = \Theta(|S_n|) = \Theta(n)$ and T'_n and S'_n differ from T_n and S_n by one edit, respectively.*

5.4 The relationship between r and r_B

Badkobeh et al. [2] considered the difference between $r(T)$ and $r_B(T)$. Notice that a lower bound on the multiplicative difference between $r(T)$ and $r_B(T)$ is a lower bound on the multiplicative difference between $r_B(T)$ and $r_B(T')$ for any rotation T' of T due to Proposition 3, since $r(T)$ is equivalent to $r_B(\text{minrot}(T))$, the RLBBWT of a specific rotation of T . A family of strings where $r_B(T)$ could be a $\Theta(\log n)$ factor larger than $r(T)$ was reported.

► **Theorem 17** ([2, Theorem 2]). *There exists an infinite family of strings such that $r(T) = O(1)$ and $r_B(T) = \Theta(\log n)$.*

The family for Theorem 17 is the Fibonacci words of odd order, i.e., $\{F_{2k+1} : k = 0, 1, \dots\}$. It is well known that $r(F_k) = 2$ for any $k \geq 2$. For $r_B(F_{2k+1})$, using a result by Melançon on the Lyndon factorization of F_{2k+1} [58], it is shown that the number of Lyndon factors is $k+1$. Then, by Corollary 12, $r_B(F_{k+1}) = \Omega(k)$ is obtained.

The opposite case was shown in [5]: that $r(T)$ can be a $\Theta(\log n)$ factor larger than $r_B(T)$.

► **Theorem 18** ([5, Theorem 2]). *There exists an infinite family of strings such that $r_B(T) = O(1)$ and $r(T) = \Theta(\log n)$.*

The family used in the proof of Theorem 18 was also related to Fibonacci words. It is noticed that prepending a symbol b to the Lyndon rotation of a Fibonacci word is conjugate to the string used by Giuliani et al. [34, 35, 36] for BWT, thus giving $\Theta(\log n)$ runs, while r_B is easily verifiable to be 3. An alternate (perhaps simpler) direct proof based on morphisms is also given in [5].

As for upper bounds, poly-logarithmic bounds were derived from the upper bounds $O(z(T) \log^2 n)$ of both $r(T)$ and $r_B(T)$ mentioned above, as well as other known bounds $z(T) = O(\delta(T) \log n)$ and $r(T), r_B(T) = \Omega(\delta(T))$ [44, 45, 61].

► **Theorem 19** ([5, Lemma 11]). *For any T and $T' \in [T]$, $r_B(T')/r_B(T) = O(\text{polylog}(n))$.*

Similar to what was observed for the lower bound, Theorem 19 implies an upper bound on the multiplicative difference between $r(T)$ and $r_B(T)$, as well.

► **Corollary 20** ([5, Corollary 12]). *For any string T , $\max\{r(T)/r_B(T), r_B(T)/r(T)\} = O(\text{polylog}(n))$.*

5.5 Rotation and BBWT

As seen in Theorem 18, rotating the string can in some cases improve the compression performance of RLBBWT by a $\Theta(\log n)$ factor. Such a rotation can be encoded as an extra integer ($\log n$ bits). Recall that in order to recover the string, BWT required extra information (an integer of $\log n$ bits¹⁶) to specify which rotation was the original string. Since the BWT is equivalent to BBWT for a specific rotation $\text{minrot}(T)$ of T (Proposition 3), it is always possible to find an encoding of a string using RLBBWT preceded by a rotation, that uses at most the space required for RLBBWT (and the extra information). It is known that $\text{minrot}(T)$ can be computed in linear time [69]. Badkobeh et al. raise the question of whether the optimal rotation, i.e., $\arg \min_{0 \leq i < n} \rho(\text{BBWT}(\text{rot}^i(T)))$ can be computed in sub-quadratic time, and give partial results towards this goal [2].

Badkobeh et al. further consider a compression scheme where multiple rotations and BBWT operations can be used. They give the following conjecture for strings with the same Parikh vector, where the Parikh vector of a string T is a vector of the length of the alphabet storing the frequencies of the characters in T .

► **Conjecture 21** ([2, Conjecture 1]). *Given two strings with the same Parikh vector, it is possible to transform one to the other by using only rotation and BBWT operations.*

If the conjecture is true, this implies that any string can be represented, for example, by its Parikh vector and a sequence of integers alternately representing rotations or the number of times BBWT should be applied to obtain the string.

They show that the conjecture holds for the case where the alphabet is binary, or, when each symbol occurs only once.

► **Theorem 22** ([2, Theorem 4]). *Given two strings of the same length with the same Parikh vector, it is possible to transform one to the other by using only rotation and BBWT operations if all symbols are distinct, or if the alphabet is binary.*

¹⁶ Here, we do not consider the case of adding an explicit terminal symbol – using an extra explicit terminal symbol outside the alphabet may increase the encoding of each symbol by 1 bit (e.g. when $\sigma = 2^k$ for some k), which would then require an extra $n + \log(\sigma + 1)$ bits.

The theorem is proved by showing that, under the assumption of the alphabet, a string that is not the lexicographically smallest string can always be transformed, using a combination of rotations and an inverse BBWT operation, into a lexicographically smaller string. The theorem then follows from the bijectivity of rotations and BBWT.

5.6 Clustering effects

As mentioned at the beginning of Section 5, BWT is known to have a property which tends to group characters together. Other than by entropy arguments, this so-called *clustering effect* has also been analyzed with respect to the run-length encoding.

The binary strings which are clustered completely by BWT have been completely characterized by Mantaci et al. [56]:

► **Theorem 23** ([56, Corollary 11]). *BWT(w) = 2 if and only if w is the power of a rotation of a standard Sturmian word.*

A similar result was shown for the BBWT in [12]:

► **Theorem 24** ([12, Theorem 2]). *BBWT(T) = 2 if and only if $T = \mathbf{b}^k \mathbf{a}^\ell$ for some $k, \ell \geq 1$, or T is a power of the Lyndon rotation of a standard word.*

In the first case, $\text{BBWT}(T) = \mathbf{a}^\ell \mathbf{b}^k$, while in the second case, $\text{BBWT}(T) = \mathbf{b}^k \mathbf{a}^\ell$, such that T is the m th power of the Lyndon rotation of a standard word, where $m = \gcd(k, \ell)$.

Note that Fibonacci words (a subset of standard Sturmian words) have many runs: $\rho(w) = \Theta(n)$. This implies that in some cases, the BWT is able to transform, with respect to run-length encoding, a least compressible string to a most compressible string. The same holds for BBWT, again due to Proposition 3.

It is important to note that the BWT does not always make the string more compressible¹⁷, and there exist strings T for which $r(T) > \rho(T)$. For example, $\text{BWT}(\mathbf{aabb}) = \mathbf{baba}$. However, Mantaci et al. [55] showed that $r(T)$ can never be more than twice the number of runs of T :

► **Theorem 25** ([55, Theorem 3.3]). *For any string T , $r(T) = \rho(\text{BWT}(T)) \leq 2\rho(T)$.*

Badkobeh et al. [2] showed that the result can be further generalized for BBWT.

► **Theorem 26** ([5, Theorem 14]). *For any string T , $r_B(T) = \rho(\text{BBWT}(T)) \leq 2\rho(T)$.*

These results indicate that, with respect to run-length encoding, BWT and BBWT transform the string in an asymmetric way; while it is possible sometimes to make the string drastically compressible (Theorems 23 and 24), it will not make it much less compressible than it currently is (Theorems 25 and 26).

The examples of Theorems 23 and 24 are in some sense remarkable with respect to run-length encoding. However, the compressibility of the string with respect to dictionary compression does not change: the example only shows a compressible string (with respect to r and r_B) being transformed into a compressible string.

Bannai et al. [5] analyze how BWT and BBWT change the string with respect to other repetitiveness measures. For both BWT and BBWT, it is proved that the transforms will not increase the repetitiveness measure by more than a poly-logarithmic factor.

¹⁷In fact, there can exist no such lossless compression algorithm [49].

► **Theorem 27** ([5, Theorem 15]). $\mathbf{m}(\text{BWT}(T)) = O(\mathbf{m}(T)\text{polylog}(n))$, $\mathbf{m}(\text{BBWT}(T)) = O(\mathbf{m}(T)\text{polylog}(n))$ hold for any repetitiveness measure $\mathbf{m}$ such that $\mathbf{m}(T) = O(\rho(T))$ and $\mathbf{m}(T) = \Omega(\delta(T))$.

The proof is based on the simple observation that since $\mathbf{m}(T) = O(\rho(T))$, $\mathbf{m}(\text{BWT}(T)) = O(\rho(\text{BWT}(T))) = O(r(T)) = O(\delta(T) \log^2 n) = O(\mathbf{m}(T) \log^2 n)$. A poly-logarithmic bound for BBWT can be shown as well, with the use of Corollary 20. Thus, $\mathbf{m}(\text{BWT}(T))/\mathbf{m}(T) = O(\text{polylog}(n))$ and $\mathbf{m}(\text{BBWT}(T))/\mathbf{m}(T) = O(\text{polylog}(n))$.

Furthermore, it is shown that there is an infinite family of strings such that the BBWT transforms a maximally incompressible string, with respect to any measure lower-bounded by δ , to a very compressible string.

► **Theorem 28** ([5, Theorem 16]). *There exists an infinite family of strings such that $\delta(T) = \Omega(n/\log n)$ and $\mathbf{m}(\text{BBWT}(T)) = O(1)$ for any $\mathbf{m}$ s.t. $\mathbf{m}(T) = O(r(T))$ for all T , and $\mathbf{m}(\text{BBWT}(T)) = O(\log n)$ for any $\mathbf{m}$ s.t. $\mathbf{m}(T) = O(g(T))$ for all T , where g is the size of the smallest grammar compressed representation for T .*

The family of strings identified is again the Fibonacci words, which are very compressible, but here used as images of BBWT, i.e., T is such that $\text{BBWT}(T) = F_k$ for some k . It is proved that the string T obtained by applying the inverse BBWT to Fibonacci words are not compressible by dictionary compression. Thus, BBWT transforms strings in an asymmetric way with respect to other repetitiveness measures as well. An interesting implication of this result is that in some cases, by applying BBWT in a pre-processing step before applying a dictionary compression, it is possible to transcend any dictionary compressor. It is not yet known whether analogous strings exist for BWT; the same construction using inverse BWT does not work since Fibonacci words are not always BWT images.

The proof of Theorem 28 is based on an elegant characterization of the LF mapping (or π_{F_k}) on the Fibonacci word F_k . It utilizes a bit string representation of integers based on the fact (re)discovered¹⁸ by Zeckendorf [72], that any integer can be uniquely represented as the sum of a set of distinct and non-consecutive Fibonacci numbers. It is shown that if a position i in F_k is represented as a k -bit string s_i based on the Zeckendorf representation (for $j \in [0, k)$, the j th bit is 1, if and only if the $(j+1)$ st Fibonacci number is used), the bit-string corresponding to the position $\pi_{F_k}(i)$ is a rotation of s_i . This implies that each conjugacy class of k -bit strings for positions in $[0, |F_k|)$ correspond to a cycle in the standard permutation π_{F_k} of F_k . Then, by showing that the length of cyclic strings corresponding to the cycles of π_{F_k} can be bounded by k , and, when k is prime, that the Lyndon rotation of each of the strings occur uniquely in T , it follows that $\delta(T) \geq (|F_k| - 2k + 1)/2k = \Omega(n/\log n)$.

6 A class of bijective BWTs

In some way, it would be more natural to define the BBWT as a bijective Burrows-Wheeler transform in the sense that it is not the only variant that is an isomorphism on Σ^n that sorts rotations of substrings of the input by the inverse of the Gessel-Reutenauer transform. In fact, we can exchange the Lyndon factorization with any factorization such that

- it is uniquely defined,
- it produces a set of primitive factors, and
- there is a unique way of restoring the text from this set of primitive strings.

¹⁸ See https://proofwiki.org/wiki/Zeckendorf%27s_Theorem/Historical_Note.

The Lyndon factorization is not the only factorization that has these properties. Daykin et al. [23] introduced the term *unique maximal factorization family* (UMFF) to address a factorization that is unique and greedy in the choice of selecting the next factor. The Lyndon factorization is an example for an UMFF, but not the only one. In [25] and [24] a bijective variant of the BWT was proposed, based on the so-called V-order and B-order, respectively. For both variants, the authors gave a semantic definition similar to Lyndon words, called *V-words* and *B-words*, by switching the lexicographical order with the V-order or the B-order, respectively. Both approaches apply a factorization similar to the Lyndon factorization and sort the conjugates in the respective order. The authors give linear time constructions for both approaches based on SAIS and show how to recover the original text.

Hendrian et al. [38] proposed a bijective BWT variant based on the Galois factorization [65]. An extension to the Galois factorization is the generalized Lyndon factorization [26], which applies a generalized lexicographic order, using a different order of the characters for each text position to compare. Similarly, we can create a bijective variant of the BWT that is based on the Nyldon factorization [19] or the canonical inverse Lyndon factorization. For the latter, in [13] so-called anti-Lyndon words and inverse Lyndon words were proposed. Anti-Lyndon words are Lyndon words for the inverse order of the characters. An inverse Lyndon word is the lexicographically largest among all its suffixes [13, Definition 3.1]. (Inverse Lyndon words can also be defined on properties based on their rotations [13, Proposition 3.1], which we here skip since these properties need further definitions.) In particular, an inverse Lyndon word can have a border, while an anti-Lyndon word cannot. A class of factorizations were studied in [13], which factorize a string into inverse Lyndon words. The authors also gave one canonical factorization that uniquely creates inverse Lyndon words, where the inverse Lyndon factors $F_1, \dots, F_m$ are in lexicographically increasing order, i.e., $F_1 \prec_{lex} \dots \prec_{lex} F_m$. However, an obstacle to define a bijective BWT variant based on inverse Lyndon factors is that an inverse Lyndon word is not necessarily primitive. It would be interesting to study whether any type of these factorizations can form a bijective variant of the BWT which can be used for pattern matching.

7 Conclusion

We presented the BBWT, a bijective variant of the BWT which is mathematically elegant in the sense that it is designed to be invertible without requiring additional information such as end-of-string markers or marking a dedicated position in the BWT. We have surveyed the following topics.

Construction. We studied the history of construction algorithms, starting with $O(n \log n)$ time, leading to current $O(n)$ time construction algorithms. It is also possible to construct the BBWT in-place in $O(n^2)$ time. All approaches have in common that they adapt algorithms for suffix array or BWT construction.

Indexing and pattern matching. The BBWT can be used for text indexing, similar to BWT-based or eBWT-based FM-indexing. For that, a slight modification of the FM-index suffices, which takes individual measures whenever the search interval matches a prefix of a Lyndon factor of the pattern with the starting position of a Lyndon factor of the text. This additional handling can slow down the computation by requiring up to a multiplicative factor of $O(\log m)$ additional queries to a rank/select data structure built on the BBWT, where m is the pattern length.

Compression performance. Like the BWT, the BBWT tends to group similar characters together, making it effective for run-length encoding. However, this grouping can be affected by rotating the input text and by the number of Lyndon factors of the input, which serves as a lower bound on the number of character runs. We draw connections to dictionary compression techniques like LZ77 and macro schemes and give pointers to studies that show that the BBWT can achieve comparable or better compression ratios than the BWT in specific cases.

Open problems

- Can we construct the BBWT in $O(n/\log_\sigma n)$ time? It is possible to do so for constructing the BWT [42] and the Lyndon array [4], from which we can extract the Lyndon factorization.
- Can we match a pattern of length m with the techniques of Theorem 10 with $O(m)$ rank operations? The multiplicative term $\log m$ seems too pessimistic: we are confident that a refined analysis could improve the time complexity.
- Reachability: Can we express any primitive string T by its Parikh vector p and a sequence of BBWT applications and rotations such that applying this sequence of transforms on the lexicographically smallest string having p as its Parikh vector leads to T ? Given the Parikh vector is $\vec{x} = (x_1, \dots, x_\sigma)$ for an alphabet $\Sigma = \{c_1, \dots, c_\sigma\}$, the lexicographically smallest string having $\vec{x}$ as its Parikh vector is $S = c_1^{x_1} \dots c_\sigma^{x_\sigma}$. Cyclic rotations and the BBWT are bijective transformations, which we can apply in an arbitrary amount and order sequentially, and it seems that there is always such a sequence of transforms that turn S into T , cf. Conjecture 21.
- Can we find the cyclic rotation of any primitive T that leads to the minimum number of runs in the BBWT in $o(n^2)$ time? (Cf. Section 5.5)
- Can we compute the smallest integer $x \geq 1$ such that $\text{BBWT}^{(x)}(T) = T$ in $o(nx)$ time for any string T of length n ?
- How far can GCA and SA differ, for instance with respect to Hamming distance? This problem is relevant for the analysis of the worst-case complexity of the (practical) algorithm computing GCA from SA by shifting.

References

- 1 Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *Inf. Comput.*, 291:104999, 2023. doi:10.1016/J.IC.2022.104999.
- 2 Golnaz Badkobeh, Hideo Bannai, and Dominik Köppl. Bijective BWT based compression schemes. In Zsuzsanna Lipták, Edleno Silva de Moura, Karina Figueroa, and Ricardo Baeza-Yates, editors, *String Processing and Information Retrieval - 31st International Symposium, SPIRE 2024, Puerto Vallarta, Mexico, September 23-25, 2024, Proceedings*, volume 14899 of *Lecture Notes in Computer Science*, pages 16–25. Springer, 2024. doi:10.1007/978-3-031-72200-4_2.
- 3 Uwe Baier. Linear-time suffix sorting - A new approach for suffix array construction. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 23:1–23:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.23.
- 4 Hideo Bannai and Jonas Ellert. Lyndon arrays in sublinear time. In *Proc. ESA*, volume 274 of *LIPIcs*, pages 14:1–14:16, 2023. doi:10.4230/LIPIcs.ESA.2023.14.
- 5 Hideo Bannai, Tomohiro I, and Yuto Nakashima. On the compressiveness of the Burrows-Wheeler transform. In Paola Bonizzoni and Veli Mäkinen, editors, *36th Annual Symposium*

- on *Combinatorial Pattern Matching, CPM 2025, June 17-19, 2025, Milan, Italy*, volume 331 of *LIPICs*, pages 17:1–17:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICs.CPM.2025.17.
- 6 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Indexing the bijective BWT. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPICs*, pages 17:1–17:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.CPM.2019.17.
 - 7 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing the bijective and the extended Burrows-Wheeler Transform in linear time. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wroclaw, Poland*, volume 191 of *LIPICs*, pages 7:1–7:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CPM.2021.7.
 - 8 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing and indexing the bijective and extended Burrows-Wheeler transform. *Inf. Comput.*, 297:105153, 2024. doi:10.1016/J.IC.2024.105153.
 - 9 Timothy C. Bell, Ian H. Witten, and John G. Cleary. Modeling for text compression. *ACM Comput. Surv.*, 21(4):557–591, 1989. doi:10.1145/76894.76896.
 - 10 Nico Bertram, Jonas Ellert, and Johannes Fischer. Lyndon words accelerate suffix sorting. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPICs*, pages 15:1–15:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ESA.2021.15.
 - 11 Elena Biagi, Davide Cenzato, Zsuzsanna Lipták, and Giuseppe Romana. On the number of equal-letter runs of the Bijective Burrows-Wheeler Transform. In Giuseppa Castiglione and Marinella Sciortino, editors, *Proceedings of the 24th Italian Conference on Theoretical Computer Science, Palermo, Italy, September 13-15, 2023*, volume 3587 of *CEUR Workshop Proceedings*, pages 129–142. CEUR-WS.org, 2023. URL: <https://ceur-ws.org/Vol-3587/4564.pdf>.
 - 12 Elena Biagi, Davide Cenzato, Zsuzsanna Lipták, and Giuseppe Romana. On the number of equal-letter runs of the Bijective Burrows-Wheeler Transform. *Theoretical Computer Science*, page 115004, 2024. doi:10.1016/j.tcs.2024.115004.
 - 13 Paola Bonizzoni, Clelia De Felice, Rocco Zaccagnino, and Rosalba Zizza. Inverse Lyndon words and inverse Lyndon factorizations of words. *Adv. Appl. Math.*, 101:281–319, 2018. doi:10.1016/j.aam.2018.08.005.
 - 14 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. doi:10.1142/S0129054114400309.
 - 15 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 129–142. Springer, 2021. doi:10.1007/978-3-030-86692-1_11.
 - 16 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. *r*-indexing the eBWT. *Inf. Comput.*, 298:105155, 2024. doi:10.1016/J.IC.2024.105155.
 - 17 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, Systems Research Center, 1994. SRC Research Report 124.
 - 18 Davide Cenzato, Zsuzsanna Lipták, Nadia Pisanti, Giovanna Rosone, and Marinella Sciortino. BWT for string collections. In Paolo Ferragina, Travis Gagie, and Gonzalo Navarro, editors,

- The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday*, volume 131 of *OASIS*s, pages 3:1–3:29, 2025.
- 19 Émilie Charlier, Manon Philibert, and Manon Stipulanti. Nyldon words. *J. Comb. Theory, Ser. A*, 167:60–90, 2019. doi:10.1016/j.jcta.2019.04.002.
 - 20 K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958. URL: <http://www.jstor.org/stable/1970044>.
 - 21 Maxime Crochemore, Jacques Désarménien, and Dominique Perrin. A note on the Burrows-Wheeler transformation. *Theor. Comput. Sci.*, 332(1-3):567–572, 2005. doi:10.1016/J.TCS.2004.11.014.
 - 22 Maxime Crochemore, Roberto Grossi, Juha Kärkkäinen, and Gad M. Landau. Computing the Burrows-Wheeler transform in place and in small space. *J. Discrete Algorithms*, 32:44–52, 2015. doi:10.1016/j.jda.2015.01.004.
 - 23 David E. Daykin, Jacqueline W. Daykin, and William F. Smyth. Combinatorics of unique maximal factorization families (umffs). *Fundam. Informaticae*, 97(3):295–309, 2009. doi:10.3233/FI-2009-202.
 - 24 Jacqueline W. Daykin, Richard Groult, Yannick Guesnet, Thierry Lecroq, Arnaud Lefebvre, Martine Léonard, and Élise Prieur-Gaston. Binary block order rouen transform. *Theor. Comput. Sci.*, 656:118–134, 2016. doi:10.1016/J.TCS.2016.05.028.
 - 25 Jacqueline W. Daykin and William F. Smyth. A bijective variant of the Burrows-Wheeler Transform using V-order. *Theor. Comput. Sci.*, 531:77–89, 2014. doi:10.1016/J.TCS.2014.03.014.
 - 26 Francesco Dolce, Antonio Restivo, and Christophe Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777:232–242, 2019. doi:10.1016/j.tcs.2018.12.015.
 - 27 Francesco Dolce, Antonio Restivo, and Christophe Reutenauer. Some variations on Lyndon words (invited talk). In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPICs*, pages 2:1–2:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.CPM.2019.2.
 - 28 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.
 - 29 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
 - 30 Frantisek Franek, Michael Liut, and William F. Smyth. On Baier's sort of maximal Lyndon substrings. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2018, Prague, Czech Republic, August 27-28, 2018*, pages 63–78. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2018. URL: <http://www.stringology.org/event/2018/p07.html>.
 - 31 Ira M. Gessel and Christophe Reutenauer. Counting permutations with given cycle structure and descent set. *J. Comb. Theory A*, 64(2):189–215, 1993.
 - 32 Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012. arXiv:1201.3077.
 - 33 Sara Giuliani. *Sensitivity of the Burrows-Wheeler Transform to Small Modifications, and Other Problems on String Compressors in Bioinformatics*. PhD thesis, University of Verona, 2023.
 - 34 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. SOFSEM*, volume 12607 of *LNCS*, pages 249–262, 2021. doi:10.1007/978-3-030-67731-2_18.
 - 35 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. Bit catastrophes for the Burrows-Wheeler Transform. In Frank Drewes and Mikhail Volkov, editors, *Developments in Language Theory - 27th International Conference*,

- DLT 2023, Umeå, Sweden, June 12-16, 2023, Proceedings*, volume 13911 of *Lecture Notes in Computer Science*, pages 86–99. Springer, 2023. doi:10.1007/978-3-031-33264-7_8.
- 36 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. Bit Catastrophes for the Burrows-Wheeler Transform. *Theory of Computing Systems*, 2025. to appear.
 - 37 Sara Giuliani, Zsuzsanna Lipták, Francesco Masillo, and Romeo Rizzi. When a dollar makes a BWT. *Theor. Comput. Sci.*, 857:123–146, 2021. doi:10.1016/J.TCS.2021.01.008.
 - 38 Diptarama Hendrian, Dominik Köppl, Ryo Yoshinaka, and Ayumi Shinohara. Algorithms for Galois words: Detection, factorization, and rotation. In *Proc. CPM*, volume 296 of *LIPICs*, pages 18:1–18:16, 2024. doi:10.4230/LIPICs.CPM.2024.18.
 - 39 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016. doi:10.1016/J.TCS.2016.03.005.
 - 40 Hyodam Jeon and Dominik Köppl. Compression sensitivity of the Burrows-Wheeler transform and its bijective variant. *Mathematics*, 13(7)(1070):1–46, March 2025. doi:10.3390/math13071070.
 - 41 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.
 - 42 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proc. STOC*, pages 756–767, 2019. doi:10.1145/3313276.3316368.
 - 43 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. *Commun. ACM*, 65(6):91–98, 2022. doi:10.1145/3531445.
 - 44 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.
 - 45 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Trans. Inf. Theory*, 69(4):2074–2092, 2023. doi:10.1109/TIT.2022.3224382.
 - 46 Dominik Köppl, Daiki Hashimoto, Diptarama Hendrian, and Ayumi Shinohara. In-place bijective Burrows-Wheeler Transforms. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPICs*, pages 21:1–21:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.CPM.2020.21.
 - 47 Manfred Kufleitner. On bijective variants of the Burrows-Wheeler Transform. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 - September 2, 2009*, pages 65–79. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009. URL: <http://www.stringology.org/event/2009/p07.html>.
 - 48 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theory*, 22(1):75–81, 1976. doi:10.1109/TIT.1976.1055501.
 - 49 Ming Li and Paul M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications, Third Edition*. Texts in Computer Science. Springer, 2008. doi:10.1007/978-0-387-49820-1.
 - 50 Konstantin M. Likhomanov and Arseny M. Shur. Two combinatorial criteria for BWT images. In Alexander S. Kulikov and Nikolay K. Vereshchagin, editors, *Computer Science - Theory and Applications - 6th International Computer Science Symposium in Russia, CSR 2011, St. Petersburg, Russia, June 14-18, 2011. Proceedings*, volume 6651 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2011. doi:10.1007/978-3-642-20712-9_30.
 - 51 M. Lothaire. *Combinatorics on Words*. Cambridge Mathematical Library. Cambridge University Press, 2 edition, 1997.

- 52 M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.
- 53 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.
- 54 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.
- 55 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, Marinella Sciortino, and Luca Versari. Measuring the clustering effect of BWT via RLE. *Theor. Comput. Sci.*, 698:79–87, 2017. doi:10.1016/J.TCS.2017.07.015.
- 56 Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows-Wheeler transform and Sturmian words. *Inf. Process. Lett.*, 86(5):241–246, 2003. doi:10.1016/S0020-0190(02)00512-4.
- 57 Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001. doi:10.1145/382780.382782.
- 58 Guy Melançon. Lyndon words and singular factors of Sturmian words. *Theor. Comput. Sci.*, 218(1):41–59, 1999. doi:10.1016/S0304-3975(98)00249-7.
- 59 J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theor. Comput. Sci.*, 165(2):311–323, 1996. doi:10.1016/0304-3975(95)00225-1.
- 60 Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.
- 61 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.
- 62 Gonzalo Navarro, Carlos Ochoa, and Nicola Prezza. On the approximation ratio of ordered parsings. *IEEE Trans. Inf. Theory*, 67(2):1008–1026, 2021. doi:10.1109/TIT.2020.3042746.
- 63 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. doi:10.1109/TC.2010.188.
- 64 Jannik Olbrich, Enno Ohlebusch, and Thomas Böhler. Generic non-recursive suffix array construction. *ACM Trans. Algorithms*, 20(2):18, 2024. doi:10.1145/3641854.
- 65 Christophe Reutenauer. Mots de Lyndon généralisés. *Séminaire Lotharingien de Combinatoire*, 54(B54h):1–16, 2005.
- 66 David A. Scott. A truly BIJECTIVE BWT is here! <https://groups.google.com/g/comp.compression/c/SDTLJypCWvc/m/ElbLTWJbnH8J>, December 2007. Usenet thread, accessed Mar. 11, 2025.
- 67 David A. Scott. BIJECTIVE BWTS PAPER. <https://groups.google.com/g/comp.compression/c/IN4XwmstgHk/m/X1RHMf2lyTgJ>, September 2009. Usenet thread, accessed Mar. 11, 2025.
- 68 David A. Scott. The paper “A Bijective String Sorting Transform”. <https://groups.google.com/g/comp.compression/c/hmMqcF0lCn8/m/AUQ7jasIiaoJ>, July 2009. Usenet thread, accessed Mar. 11, 2025.
- 69 Yossi Shiloach. Fast canonization of circular strings. *J. Algorithms*, 2(2):107–121, 1981. doi:10.1016/0196-6774(81)90013-4.
- 70 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.
- 71 Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. On the size of overlapping Lempel-Ziv and Lyndon factorizations. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPICs*, pages 29:1–29:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.CPM.2019.29.
- 72 Edouard Zeckendorf. Représentations des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas. *Bulletin de La Society Royale des Sciences de Liege*, pages 179–182, 1972. URL: <https://cir.nii.ac.jp/crid/1570009749187075840>.

BWT for String Collections

Davide Cenzato 

Ca' Foscari University of Venice, Italy

Zsuzsanna Lipták 

University of Verona, Italy

Nadia Pisanti 

University of Pisa, Italy

Giovanna Rosone 

University of Pisa, Italy

Marinella Sciortino 

University of Palermo, Italy

Abstract

We survey the different methods used for extending the BWT to collections of strings, following largely [Cenzato and Lipták, CPM 2022, Bioinformatics 2024]. We analyze the specific aspects and combinatorial properties of the resulting BWT variants and give a categorization of publicly available tools for computing the BWT of string collections. We show how the specific method used impacts on the resulting transform, including the number of runs, and on the dynamicity of the transform with respect to adding or removing strings from the collection.

We then focus on the number of runs of these BWT variants and present the optimal BWT introduced in [Cenzato et al., DCC 2023], which implements an algorithm originally proposed by [Bentley et al., ESA 2020] to minimize the number of BWT-runs. We also discuss several recent heuristics and study their impact on the compression of biological sequences.

We conclude with an overview of the applications and the impact of the BWT of string collections in bioinformatics.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Burrows-Wheeler transform, Extended Burrows-Wheeler transform, compressed text indexes, text compression, string collections, bioinformatics

Digital Object Identifier 10.4230/OASICS.Manzini.2025.3

Funding Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. GR and NP are partially supported by PNRR – M4C2 – Investimento 1.5, Ecosistema dell’Innovazione ECS00000017 – “THE – Tuscany Health Ecosystem” – Spoke 6 “Precision medicine & personalized healthcare”, funded by the European Commission under the NextGeneration EU programme. ZsL, NP, GR, MS, are partially supported by MUR PRIN 2022 YRB97K “PINC” funded by Next Generation EU PNRR M4 C2 Inv. 1.1. ZsL, GR, and MS are also partially supported by the INdAM - GNCS Project CUP_E53C24001950001.

Davide Cenzato: DC is funded by ERC StG “REGINDEX: Compressed indexes for regular languages with applications to computational pan-genomics” grant nr 101039208.

Nadia Pisanti: NP is also supported by the PANGAIA and ALPACA projects that received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No. 872539 and 956229, respectively.

Marinella Sciortino: MS is partially supported by Project “ACoMPA” (CUP B73C24001050001) funded by the NextGeneration EU programme PNRR MUR M4 C2 Inv. 1.5 – Project ECS00000017 Tuscany Health Ecosystem (Spoke 6), CUP B63C22000680007.



© Davide Cenzato, Zsuzsanna Lipták, Nadia Pisanti, Giovanna Rosone, and Marinella Sciortino; licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini’s 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 3; pp. 3:1–3:29

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

With the advent of high-throughput sequencing technologies, the amount of genomic data available in public databases has undergone an exponential increase [43]. Similar data explosion can be witnessed in other areas based on textual data, such as webpages, versioned texts (such as Wikipedia), versioned software repositories (such as Github), digital books, and many others (for more details, see [93]).

However, not only has the amount or size of textual data increased, but it has also changed character. Most importantly, nowadays most data of interest consists of *string collections* rather than of individual strings. For the purpose of this survey, a string collection is any multiset of strings (or sequences, two terms we use interchangeably). Early examples include the famous 1000 Genomes Project [116], in the course of which almost 2000 human genomes were sequenced, followed by the 10,000 Genomes Project [56], the 100,000 Human Genome Project [120], the 1001 Arabidopsis Project [117], the 3,000 Rice Genomes Project (3K RGP) [115], and others. Projects of the last few years include the COVID-19 Genomics UK Consortium's [38] and the H3Africa Consortium's work [34].

The Burrows-Wheeler Transform (BWT) [23] is one of the most fundamental methods for string processing and compressed data storage [48, 83, 54]; the tools built upon these BWT-based compressed data structures are among the most used in bionformatics [71, 76, 69, 78]. It is therefore natural that researchers and practitioners have gone ahead and applied BWT-based methods to this new kind of data. The BWT, however, was originally defined for *one* string, and it is not immediately clear how to extend it to several strings.

As it turns out, several different methods have been used to generalize the BWT to string collections. Some of these methods were carefully designed and consciously chosen, while others appear to have been simply implementation choices. In fact, in many publications it is not explicitly specified how the BWT of a string collection was computed. Underlying this is the implicit assumption – incorrect, as we will see – that all methods are equivalent.

The classic method, paralleling *generalized suffix trees* and *generalized suffix arrays* (see e.g., [65] or [95]), is to concatenate the strings in the collection, using distinct end-of-string symbols to separate consecutive strings. In effect, this means turning the string collection into one string, at which point one can apply the classic BWT in the usual way. Following established terminology, we will refer to these end-of-string-symbols as 'dollars'; so we have a distinct dollar ($\$_i$) for each input string. In actual implementations, of course, the same dollar-symbol is used, because otherwise the alphabet would be blown up enormously, not a viable choice. However, the dollars remain distinct conceptually; one common way to handle this is to use the position of the dollars for breaking ties. This method, concatenating the strings with distinct dollars, is what we will refer to in this paper as *multidollarBWT*.

Another method that is commonly used is to concatenate the input strings but drop the distinction between the dollars; this is a much simpler choice on an implementation level because one does not have to take care of distinguishing between the different dollars. In order to ensure correctness (of BWT construction, LF-mapping, backward search, etc.), it is necessary to append one more symbol at the very end of the concatenated string (often termed 'hash' and denoted $\#$), which is smaller than all other characters, including the dollar. We will call this method *concatBWT*.

In 2007, a mathematically clean extension of the BWT to string collections was proposed by Mantaci et al. [84], which the authors called *extended BWT* (EBWT). The EBWT is essentially the inverse of the Gessel-Reutenauer bijection [57], known in combinatorics on words, but hitherto unknown in the research community working on textual data structures

■ **Table 1** The five BWT variants and the three BWTs using specific string orders, on the multiset $\mathcal{M} = (\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA})$. Double listings due to options of the software. We note also that 1FGSACA is a library, while the others are dedicated command-line tools for BWT computation.

	<i>result on example</i>	<i>tools</i>
<i>BWT variants</i> (Sections 3 and 4)		
EBWT	GGGCTACTCACACCTCTAGCG	PFP-eBWT [101], cais [27], 1FGSACA [73]
dolEBWT	ACACAGGGCGCCTAT\$\$\$TCTC\$\$G\$C	G2BWT [49], PFP-eBWT [101], cais [27], msbwt [88]
mdolEBWT and mdolBWT	AGCACAGGGCGCCTTA\$\$\$TTCC\$\$G\$C	BEETL [8], BCR_LCP_GSA [7], ropeBWT2 [107], ropeBWT3 [108], Merge-BWT [87], gsufsort [61], gr1BWT [60], eGAP [45], eGSA [47], bwt-lcp-parallel [24], bwt-lcp-em [25]
concatBWT	\$ACAGCAGCGGCCTAT\$\$TCTC\$\$G\$C	BigBWT [15], r-pfbwt [105], CMS-BWT [37], tools for single-string BWT
<i>specific string orders</i> (Section 5)		
colexBWT	AAACCGGGCGCCTAT\$\$\$TCTC\$\$G\$C	BCR_LCP_GSA [7], ropeBWT2 [107], ropeBWT3 [108]
plusBWT	AAACCGGGCGCCTTA\$\$\$TTCC\$\$G\$C	BCR_LCP_GSA [7]
optBWT	AAAGCCCGGGCCTTA\$\$\$TTCC\$\$G\$C	optimalBWT [99]

and string processing. The EBWT is a generalization of the BWT from one string to a multiset of primitive strings. Similarly to the BWT, it is a permutation of the characters of the strings in input and therefore has length equal to the total length of the multiset in input. It also allows backward search and has the same ‘clustering effect’ as the BWT, i.e., it tends to produce long same-character runs on repetitive inputs. We refer to this method simply as EBWT.

Applying the classic BWT to a string which terminates with a dollar is different in many ways from applying it to one without. (For example, $\text{BWT}(\text{banana}\$) = \text{annb}\$aa$, while $\text{BWT}(\text{banana}) = \text{nnbaaa}$.) Similarly, appending a dollar to the ends of the input strings changes the properties of the EBWT. Again, one possibility is to append the same dollar to every input string; this is what happens, for example, reading in strings line-by-line, since each line is terminated by a newline symbol. We will call this method *dollarEBWT*.

Finally, if one appends different dollars to the input strings (or the same dollar but considering them as different), then we get what we call *multidollarEBWT*. Regarding the resulting transform, this method is similar to the multidollarBWT method (concatenate the strings with distinct dollars separating them), except that the indices of the dollars in the final transform are shifted by one.

In Table 1, we give a classification of existing BWT construction tools, geared specifically to string collections, according to the BWT variant they produce. As it is not always possible to distinguish the underlying method, our categorization is based on a combination of the information given in the accompanying papers and on reverse engineering the final output of the tools.

In this paper, we survey the five different methods sketched above, giving their exact definitions and combinatorial properties. As we will see, which method is chosen has important consequences for the resulting transform. This includes issues such as input order dependence (giving the string collection in a different order may or may not result in different outputs), dynamicity (how does adding or removing a string from the collection impact on

the transform), and number of runs of the BWT. This last parameter, often denoted r , is central in BWT-based data structure design, since modern compressed data structures such as the r -index [54] require storage space linear in r .

We will in fact show that if the goal is to reduce the number of runs, then either the multidollarEBWT or the multidollarBWT should be chosen, because methods leading to few runs – the exact method, called *optimalBWT*, or heuristics like *colexBWT* or *plusBWT* – only work with these variants. Moreover, both allow adding or removing strings from the collection without having to recompute the BWT. On the other hand, if input order independence is desired, then the EBWT or the dollarEBWT are the right choice.

We will also see that the concatBWT, the variant where the strings are concatenated using the same dollar, has several undesirable properties.

The number r of runs of the BWT appears increasingly as a parameter characterizing the input data: r , or equivalently, the average runlength n/r of the BWT, is often used as a measure of how repetitive the data is. This is well justified in the case of one string, as many recent publications on the theory of r as a repetitiveness measure attest (see [93] for a recent survey). However, for string collections, the parameter r is not well defined, given that different methods on the same input collection can give different r 's, and even the same method on the same collection, if presented in a different order, can give very different r 's (up to a multiplicative factor of over 31 on real data [30]). We argue that this parameter should be standardized, to r_{opt} , the minimum number of runs among all possible transforms on a given string collection.

We round off the paper with a section about applications in bioinformatics that make use of string collections. These are highly repetitive data sets that are the ideal object for data structures based on the BWT. We give an overview of the different areas and problems using highly repetitive sequence collections, which are at the center of today's bioinformatics research.

1.1 Overview of paper

The paper is organized as follows. We provide necessary background and terminology in Section 2. In Section 3, we give the precise definitions and examples for the five BWT variants, distinguishing between EBWT-based variants (Section 3.1) and concatenation-based variants (Section 3.2). We study their combinatorial properties in Section 4, followed by a section on reducing the number of runs of the BWT (Section 5). This section also contains experimental results on real data, showing how much the number of runs can be reduced. Section 6 contains an overview of areas in bioinformatics where string collections play an important role. We close with a summary and some suggestions in Section 7.

The classification of BWT variants for string collections presented in this paper is based largely on that contained in [32, 33] but with some additions and modifications. In particular, Section 3, containing the definitions, has been extended, and the classification has been further developed¹. All examples contained in this paper are new. The combinatorial properties of the BWT variants of Section 4 are partially from [32, 33], with updated presentation; Section 4.3, on dynamicity of the BWT variants, has been newly added and Section 4.4, on the output order of the BWT variants, generalizes results presented in [32]. Finally, Section 5 contains mostly results from [30, 14], while the survey of bioinformatics applications contained in Section 6 is not contained elsewhere.

¹ We now group the BWT variants in EBWT-based versus concatenation-based, and we subsume *colexBWT* and *optBWT* under multidollarEBWT and multidollarBWT.

2 Basics

Let $\Sigma = \{a_1, a_2, \dots, a_\sigma\}$ be a finite ordered alphabet Σ with $a_1 < a_2 < \dots < a_\sigma$. A string (or text) $T = T[1..n]$ is a sequence of symbols $T[1] \cdots T[n]$ over the alphabet Σ . We denote by ϵ the empty string and by $|T|$ the length of string T . For $1 \leq i \leq n$, the substring $T[1..i]$ is called the i th *prefix* of T and $T[i..n]$ the i th *suffix* of T . The string $T^{\text{rev}} = T[n] \cdots T[1]$ is the *reverse* string of T . Let $<_{\text{lex}}$ denote the standard lexicographic order. The *colexicographic* order (sometimes also referred to as *reverse lexicographic order*) is defined as $S <_{\text{colex}} T$ if and only if $S^{\text{rev}} <_{\text{lex}} T^{\text{rev}}$.

Given a string T , the string $T^k = TT \cdots T$ is the string obtained by concatenating k copies of T . A string T is called *primitive* if $T = U^k$ implies $T = U$ and $k = 1$. For every string T , there exists a unique primitive string U and a unique integer k such that $T = U^k$. The string U is called $\text{root}(T)$ and k is called $\text{exp}(T)$; thus, $T = \text{root}(T)^{\text{exp}(T)}$.

For a string T , we denote by $T^\omega = TT \cdots$ the infinite string obtained by concatenating an infinite number of copies of T . The *omega-order* on Σ^* is defined as follows: $S \prec_\omega T$ if (a) $S^\omega <_{\text{lex}} T^\omega$, or (b) $S^\omega = T^\omega$ and $\text{exp}(S) < \text{exp}(T)$ (note that $S^\omega = T^\omega$ implies $\text{root}(S) = \text{root}(T)$). It can be verified that the ω -order relation coincides with the lexicographic order if neither of the two strings is a proper prefix of the other. Otherwise, the two orders can be different. For instance, $\text{CGA} <_{\text{lex}} \text{CGACC}$ but $\text{CGACC} \prec_\omega \text{CGA}$.

Given string T , the *alphabet of T* is defined as $\text{alph}(T) = \{T_i \mid 1 \leq i \leq |T|\}$. A *run* in a string T is a maximal substring U such that $\text{alph}(U)$ is a singleton. Given a string T , we denote by $\rho(T)$ the number of runs of T , e.g., $\rho(\text{baaabbcc}) = 4$. Moreover, we denote by $r(T)$ the number of runs of the BWT of T (defined below).

The string S is a *conjugate* (or *cyclic rotation*, or simply *rotation*) of the string T if $S = T[i..n]T[1..i-1]$, for some $i \in \{1, \dots, n\}$. In this case, S is also called the i th conjugate of T , denoted $\text{conj}_i(T)$. The *conjugate array* CA of a string T of length n is the permutation of $\{1, \dots, n\}$ such that $\text{CA}[j] = i$ if $\text{conj}_i(T)$ is the j th conjugate of T with respect to the lexicographic order, with ties broken according to string order, i.e. if $\text{CA}[j] = i$ and $\text{CA}[j'] = i'$ for some $j < j'$, then either $\text{conj}_i(T) <_{\text{lex}} \text{conj}_{i'}(T)$, or $\text{conj}_i(T) = \text{conj}_{i'}(T)$ and $i < i'$. If T is a primitive string, a conjugate S of T is called *Lyndon rotation* of T , denoted as $\text{Lynd}(T)$, if it is lexicographically the smallest among all conjugates of T .

To define the *suffix array* SA, it is common to assume that the string has an end-of-string symbol $\$$ appended at the end, such that $\$$ is smaller than all other characters and does not occur elsewhere in the string. Then, the SA is a permutation of the indices $\{1, 2, \dots, n+1\}$ such that $\text{SA}[j] = i$ if suffix $T[i..n+1]$ is the j th suffix of all suffixes in lexicographic order. Note that in this case, there can be no ties, since two distinct suffixes necessarily have different lengths. Since the string $T\$$ has exactly one occurrence of $\$$, the conjugate array and the suffix array of $T\$$ coincide.

The *Burrows-Wheeler Transform* (BWT) [23] is a reversible transformation that permutes the characters of the input string. Given a string T , the *BW-matrix* of T is defined as the matrix whose rows are the conjugates of T , given in lexicographic order. We denote by $\text{BWT}(T)$ the last column of the matrix. The transformation also outputs an integer i , which is the position of T within the matrix. It follows from the definition that if a string S is a conjugate of T , then $\text{BWT}(S) = \text{BWT}(T)$. The index i is fundamental for recovering T from $\text{BWT}(T)$, otherwise the original string can be recovered only up to conjugates. When the reconstruction of the original string is not relevant, the index is omitted.

The *standard permutation* of a word $S = S[1..n]$ over Σ is the permutation π_S of $\{1, 2, \dots, n\}$ such that $\pi_S(i) < \pi_S(j)$ if and only if $S[i] < S[j]$ or $S[i] = S[j]$ and $i < j$. The standard permutation π_S of a string S is also called LF-mapping if $S = \text{BWT}(T)$, for some

string T . It is known that T is a primitive word if and only if π_S is a cycle of length n . In this case, given S , π_S and an index i , the i th conjugate U of T in lexicographic order can be recovered as follows: $U[n] = S[i]$ and $U[n-j] = S[\pi_S^j(i)]$, for $j = 1, \dots, n-1$. The conjugate $Lynd(T)$ can be recovered with $i = 1$.

The permutation $BWT(T)$ can be computed from the conjugate array CA , since for all $j = 1, \dots, n$, $BWT(T)[j] = T[CA[j] - 1]$, assuming that $T[0] = T[n]$. Alternatively, if the input string T terminates with an end-of-string symbol $\$$, $BWT(T\$)$ can be computed from the suffix array SA . Moreover, recovering the first row of the BW-matrix will yield $\$T$, because $\$$ is smaller than all other symbols, and since $\$$ occurs exactly once, unique recovery is ensured also in this case.

Appending an end-of-string symbol to the string T has traditionally allowed $BWT(T\$)$ to be computed using linear-time algorithms for suffix array construction. However, it is possible to compute $BWT(T)$ in linear time without appending any end-of-string symbol $\$$ to T . This can be done by computing the suffix array of $U = Lynd(\text{root}(T))$ [58] and then obtaining $BWT(T) = BWT(U^k)$, where $k = \exp(T)$, from $BWT(U)$, applying a simple relationship between the two transforms: $BWT(U^k)$ can be obtained by concatenating k copies of each character of $BWT(U)$ [85]. Alternatively, it is also possible to compute $BWT(T)$ directly in linear time with the `cais` algorithm, which is based on induced sorting [18].

Now let $\mathcal{M} = \{T_1, T_2, \dots, T_m\}$ be a multiset of strings (often referred to as a *string collection*), with $|T_d| = n_d$ for $1 \leq d \leq m$. We denote by $|\mathcal{M}| = m$ the number of strings in $\mathcal{M}$ and by $||\mathcal{M}|| = \sum_{d=1}^m n_d$ their total length. Further, we will denote by $\mathcal{M}_\$ = \{T_1\$, T_2\$, \dots, T_m\$ \}$ the set of strings with end-of-string markers appended to all strings in $\mathcal{M}$, of total length $m + \sum_{d=1}^m n_d$.

The *generalized conjugate array* GCA of the collection $\mathcal{M} = \{T_1, T_2, \dots, T_m\}$ is an array of total length $||\mathcal{M}||$, where $GCA[j] = (d, i)$ if $conj_i(T_d)$ is the j th string in the ω -sorted list of the conjugates of the strings in $\mathcal{M}$, with ties broken first w.r.t. the index of the string (in case of identical strings), and then w.r.t. the position in the string (in case of identical conjugates of the same string).

The *generalized suffix array* GSA of the collection $\mathcal{M}_\$$ is an array of length $||\mathcal{M}_\$|| = m + \sum_{d=1}^m n_d$, containing pairs of integers (d, i) , corresponding to the lexicographically sorted suffixes of the strings in $\mathcal{M}_\$$. In particular, $GSA[j] = (d, i)$ is the pair corresponding to the j th smallest suffix of the strings in $\mathcal{M}_\$$, with ties broken according to $\$1 < \$2 < \dots < \$m$.

3 BWT variants for string collections

For string collections, several variants of the BWT have been proposed. Each of these variants aims to extend the features of the BWT for single strings to collections of strings. In this section, we review the most significant variants, which form the foundation of the most widely used tools for indexing and compressing string collections based on the BWT. We use the classification introduced in [32, 33], and in addition group these variants into two families, depending on the strategy used to process the strings of the collection.² The first family includes transformations that explicitly or implicitly apply the Extended BWT (EBWT) introduced in [84] to the multiset of strings, to which end-of-string symbols may or may not have been appended. The second family, on the other hand, consists of those

² In difference to [32, 33], here we include a new separator-based BWT variant, the multidollarEBWT, which was previously subsumed under multidollarBWT.

transformations that involve concatenating the strings of the collection, and then applying the traditional BWT to the concatenated string. We will see that these BWT variants, while preserving the general functionality of the BWT, can produce significantly different outputs.

Let us denote by $\mathcal{M} = \{T_1, T_2, \dots, T_m\}$ the input string collection, i.e., a multiset of m strings, with $|T_d| = n_d$ for $1 \leq d \leq m$. Even though the input is a *multiset* of strings, in real applications, this input is always given in some order. As we will see, the input order plays an important role in the resulting transform for several of the BWT variants we will review. For this reason, in slight abuse of notation, whenever the definition depends on the input order, we will treat $\mathcal{M}$ in the following as an ordered collection and will write accordingly $\mathcal{M} = (T_1, T_2, \dots, T_m)$. When it does not play a role, in the sense that the output is always the same whatever the input order, in those cases we will stick to the multiset notation.

In Table 1 we list, for each BWT variant, the tools that compute it (up to renaming of dollars).

3.1 EBWT-based transforms

In this section, we present the three EBWT-based transforms: EBWT, dolEBWT, and mdlEBWT. See Table 2 for an example.

Extended BWT. The *Extended Burrows–Wheeler Transform* [84] (EBWT) is a generalization of the BWT to a multiset of strings. Given the collection of strings $\mathcal{M} = \{T_1, \dots, T_m\}$, $\text{EBWT}(\mathcal{M})$ is a permutation of the characters of the strings in $\mathcal{M}$, obtained by concatenating the last characters of all conjugates of the strings of $\mathcal{M}$, listed according to the ω -order. Note that EBWT is a transform on the string collections treated as multisets. Indeed, the output $\text{EBWT}(\mathcal{M})$ is independent of the order in which the strings appear in the collection $\mathcal{M}$. For this reason, we denote its input as a multiset.

Similarly to BWT, the permutation $\text{EBWT}(\mathcal{M})$ can be computed from the generalized conjugate array of $\mathcal{M}$ in linear time, since

$$\text{EBWT}(\mathcal{M})[i] = \begin{cases} T_d[j-1] & \text{if GCA}[i] = (d, j) \text{ with } j > 1, \\ T_d[n_d] & \text{if GCA}[i] = (d, j) \text{ with } j = 1. \end{cases} \quad (1)$$

The transformation also outputs the m -tuple $I = (i_1, \dots, i_m)$ listing the positions of the strings of the collection within this ω -sorted list. The m -tuple $I = (i_1, \dots, i_m)$ is used to recover the multiset $\mathcal{M}$ from $\text{EBWT}(\mathcal{M})$. In fact, if π_U is the standard permutation of the word $U = \text{EBWT}(\mathcal{M})$, then π_U decomposes into m disjoint cycles $\pi_1, \pi_2, \dots, \pi_m$ such that the index i_d appears in the cycle π_d . The strings of the collection $\mathcal{M}$ are recovered as follows: for every $d = 1, \dots, m$, $T_d[l_d - t] = \text{EBWT}[\pi_d^t[i_d]]$, where l_d is the length of the cycle π_d and t ranges from 0 to $l_d - 1$. Moreover, if the set of indices is considered in increasing order, then the strings of the collection are recovered in ω -order. When the reconstruction of the original multiset is not relevant, the m -tuple is omitted.

► **Example 1.** Given the collection $\mathcal{M} = \{\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA}\}$, the output of EBWT is the string GGGCTACTCACCTCTAGCG together with the indices (9, 10, 12, 16, 18, 21), as shown in Table 2. The LF-mapping decomposes into the cycles (1 13 9 7 6) (2 14 10) (3 15 20 12) (4 5 18) (8 19 16 11) and (17 21). Starting from the indices in the set 9, 10, 12, 16, 18 and 21, the strings CGACC, CGA, CTGA, GTCC, TCA and TG are reconstructed.

Dollar-EBWT. This transformation, here denoted dolEBWT , uses a single end-of-string symbol appended to each string in the collection. More precisely, given the collection $\mathcal{M} = \{T_1, \dots, T_m\}$,

$$\text{dolEBWT}(\mathcal{M}) = \text{EBWT}(\{T_d\$ \mid d = 1, \dots, m\}).$$

Note that in this case, to recover the set of strings in the collection, it is sufficient to know the positions of the $\$$ -symbols in $\text{dolEBWT}(\mathcal{M})$. Since none of the strings in the collection is a prefix of another, the ω -order coincides with the lexicographic order. For this reason, starting from the positions of the $\$$ symbols in increasing order, the strings of $\mathcal{M}$ can be recovered in lexicographic order.

Similarly to EBWT, the permutation $\text{dolEBWT}(\mathcal{M})$ can be computed from the generalized conjugate array GCA of the multiset $\{T_d\$ \mid T_d \in \mathcal{M}\}$ in linear time, since

$$\text{dolEBWT}(\mathcal{M})[i] = \begin{cases} T_d[j-1] & \text{if GCA}[i] = (d, j) \text{ with } j > 1, \\ \$ & \text{if GCA}[i] = (d, j) \text{ with } j = 1. \end{cases} \quad (2)$$

► **Example 2.** Let us consider the collection $\mathcal{M} = \{\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA}\}$, then $\text{dolEBWT}(\mathcal{M}) = \text{ACACAGGGCGCCTAT}\$ \$ \$ \$ \text{TCTC}\$ \$ \$ \text{G}\$ \text{C}$, as shown in Table 2. The LF-mapping decomposes into 6 cycles (1 7 20 16) (2 11 14 10 22 17) (3 8 21 27 18) (4 12 15 25 23) (5 9 13 24) and (6 19 26).

Multidollar-EBWT. This transformation, here denoted mdolEBWT , uses distinct end-of-string symbols appended to the strings in the collection. Given the collection $\mathcal{M} = (T_1, \dots, T_m)$ and the end-of-strings symbols $\$1 < \$2 < \dots < \$m$,

$$\text{mdolEBWT}(\mathcal{M}) = \text{EBWT}(\{T_d\$d \mid d = 1, \dots, m\}).$$

Note that, unlike the previous transformations, the output of $\text{mdolEBWT}(\mathcal{M})$ is influenced by the order in which the strings appear in the collection, because the distinct end-of-string symbols result in different orders among the strings in the multiset. (For more on this, see Section 3.) Further, the position of each $\$d$ allows for the reconstruction of the corresponding string T_d of the collection.

Since end-marker symbols are appended to each string in the collection, $\text{mdolEBWT}(\mathcal{M})$ can be constructed using the lexicographic order of the suffixes of the strings in $\mathcal{M}$ as follows:

$$\text{mdolEBWT}(\mathcal{M})[i] = \begin{cases} T_d\$d[(j-1)] & \text{if GSA}[i] = (d, j) \text{ with } j > 1 \\ \$d & \text{if GSA}[i] = (d, j) \text{ with } j = 1 \end{cases} \quad (3)$$

► **Example 3.** Let us consider the collection $\mathcal{M} = (\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA})$. Then $\text{mdolEBWT}(\mathcal{M}) = \text{AGCACAGCGGCCTTA}\$6\$5\$1\text{TTCC}\$3\$4\text{G}\$2\text{C}$, as shown in Table 2. The LF-mapping decomposes into 6 cycles (1 7 20 27 18) (2 19 26) (3 11 14 25 23) (4 8 13 24) (5 12 15 10 22 17) and (6 9 21 16). For $\mathcal{M}' = (\text{TCA}, \text{CTGA}, \text{CGA}, \text{TG}, \text{GTCC}, \text{CGACC})$, where the same strings are given in another order, $\text{mdolEBWT}(\mathcal{M}') = \text{AAAGCCCGGCCTTA}\$3\$6\$2\text{TTCC}\$5\$1\text{G}\$4\text{C}$. For more details, see Section 5.

3.2 Concatenation-based transforms

In this section, we present the concatenation-based transforms, mdolBWT and concatBWT . See Table 3 for an example.

■ **Table 2** EBWT-based BWT variants: from left to right, we show the extended BWT (EBWT), the dollar-EBWT, and the multidollar-EBWT (using different string orderings) of the string collection $\mathcal{M} = (\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA})$. We use different colors to identify the characters preceding different identical suffixes. See Sections 3.1 and 5 for more details.

				dollar-EBWT				multidollar-EBWT					
				i	GCA	rotation		opt	plus	colex	input	rotation	GSA
				1	(6,4)	$\$CGA$	A	A	A	A	A	$\$1CTGA$	(1,5)
				2	(5,6)	$\$CGACC$	C	A	A	A	G	$\$2TG$	(2,3)
				3	(1,5)	$\$CTGA$	A	A	A	A	C	$\$3GTCC$	(3,5)
				4	(3,5)	$\$GTCC$	C	G	C	C	A	$\$4TCA$	(4,4)
				5	(4,4)	$\$TCA$	A	C	C	C	C	$\$5CGACC$	(5,6)
				6	(2,3)	$\$TG$	G	C	G	G	A	$\$6CGA$	(6,4)
				7	(6,3)	$A\$CG$	G	C	G	C	G	$A\$1CTG$	(1,4)
				8	(1,4)	$A\$CTG$	G	G	G	G	C	$A\$4TC$	(4,3)
				9	(4,3)	$A\$TC$	C	G	C	G	G	$A\$6CG$	(6,3)
				10	(5,3)	$ACC\$CG$	G	G	G	G	G	$ACC\$5CG$	(5,3)
				11	(5,5)	$C\$CGAC$	C	C	C	C	C	$C\$3GTC$	(3,4)
				12	(3,4)	$C\$GTC$	C	C	C	C	C	$C\$5CGAC$	(5,5)
				13	(4,2)	$CA\$T$	T	T	T	T	T	$CA\$4T$	(4,2)
				14	(5,4)	$CC\$CGA$	A	T	T	A	T	$CC\$3GT$	(3,3)
				15	(3,3)	$CC\$GT$	T	A	A	T	A	$CC\$5CGA$	(5,4)
				16	(6,1)	$CGA\$$	$\$$	$\$3$	$\$2$	$\$2$	$\$6$	$CGA\$6$	(6,1)
				17	(5,1)	$CGACC\$$	$\$$	$\$6$	$\$5$	$\$4$	$\$5$	$CGACC\$5$	(5,1)
				18	(1,1)	$CTGA\$$	$\$$	$\$2$	$\$1$	$\$3$	$\$1$	$CTGA\$1$	(1,1)
				19	(2,2)	$G\$T$	T	T	T	T	T	$G\$2T$	(2,2)
				20	(6,2)	$GA\$C$	C	T	T	C	T	$GA\$1CT$	(1,3)
				21	(1,3)	$GA\$CT$	T	C	C	T	C	$GA\$6C$	(6,2)
				22	(5,2)	$GACC\$C$	C	C	C	C	C	$GACC\$5C$	(5,2)
				23	(3,1)	$GTCC\$$	$\$$	$\$5$	$\$4$	$\$5$	$\$3$	$GTCC\$3$	(3,1)
				24	(4,1)	$TCA\$$	$\$$	$\$1$	$\$3$	$\$1$	$\$4$	$TCA\$4$	(4,1)
				25	(3,2)	$TCC\$G$	G	G	G	G	G	$TCC\$3G$	(3,2)
				26	(2,1)	$TG\$$	$\$$	$\$4$	$\$6$	$\$6$	$\$2$	$TG\$2$	(2,1)
				27	(1,2)	$TGA\$C$	C	C	C	C	C	$TGA\$1C$	(1,2)

EBWT			
i	GCA	rotation	
1	(5,3)	ACCCG	G
2	(6,3)	ACG	G
3	(1,4)	ACTG	G
4	(4,3)	ATC	C
5	(4,2)	CAT	T
6	(5,4)	CCCGA	A
7	(5,5)	CCGAC	C
8	(3,3)	CCGT	T
9	(5,1)	CGACC	C
10	(6,1)	CGA	A
11	(3,4)	CGTC	C
12	(1,1)	CTGA	A
13	(5,2)	GACCC	C
14	(6,2)	GAC	C
15	(1,3)	GACT	T
16	(3,1)	GTCC	C
17	(2,2)	GT	T
18	(4,1)	TCA	A
19	(3,2)	TCCG	G
20	(1,2)	TGAC	C
21	(2,1)	TG	G

Multidollar-BWT. In this transformation, denoted mdolBWT, given the string collection $\mathcal{M} = (T_1, \dots, T_m)$, the strings are concatenated in the order in which they appear in the m -tuple, using $\$1 < \$2 < \dots < \$m$ as separators, and then the BWT is applied to the concatenated string. More formally,

$$\text{mdolBWT}(T_1, \dots, T_m) = \text{BWT}(T_1\$1T_2\$2 \cdots T_m\$m).$$

Unlike the EBWT-based transforms, if π_L is the standard permutation of the word $L = \text{dolEBWT}(\mathcal{M})$, then π_L consists of a single cycle. Each string T_d in the collection can be reconstructed, using the LF mapping, starting from the symbol $\$d$ and terminating the reconstruction when the symbol $\$_{d-1}$ (for $d > 1$) or $\$m$ (for $d = 1$) is reached.

The order in which the strings appear in the collection can significantly affect the output, giving rise to specific transforms. See Section 5 for more details.

Note that sometimes the separators are added only implicitly when concatenating the strings, for example by maintaining a bitvector that marks the beginning of a new string. Moreover, some implementations of mdolBWT compute the BWT after appending an additional end-of-string symbol $\#$, which is smaller than all the characters of the alphabet

and the dollar symbols, to the concatenation of the strings. The resulting output differs from mdolBWT only in that the symbol $\$m$ is added in the first position, and the symbol $\#$ replaces $\$m$. The mdolBWT($\mathcal{M}$) is constructed using the suffix array SA of the concatenated string $T = T_1\$1T_2\$2 \cdots T_m\$m$ as follows:

$$\text{mdolBWT}(\mathcal{M})[i] = \begin{cases} T[\text{SA}[i] - 1] & \text{if } \text{SA}[i] > 1, \\ \$m & \text{if } \text{SA}[i] = 1. \end{cases} \quad (4)$$

► **Example 4.** Let us consider the collection $\mathcal{M} = (\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA})$. Then $\text{mdolBWT}(\mathcal{M}) = \text{AGCACAGCGGCCTTA}\$5\$4\$6\text{TTCC}\$2\$3\text{G}\$1\text{C}$, as shown in Table 3. The LF-mapping consists of a single cycle, which is (1 7 20 27 18 6 9 21 16 5 12 15 10 22 17 4 8 13 24 3 11 14 25 23 2 19 26). If $\mathcal{M}''$ contains the same strings in colexicographic order, i.e., $\mathcal{M}'' = (\text{TCA}, \text{CGA}, \text{CTGA}, \text{CGACC}, \text{GTCC}, \text{TG})$, then $\text{mdolBWT}(\mathcal{M}'') = \text{AAACCGCGGCCTAT}\$1\$3\$2\text{TCTC}\$4\$6\text{G}\$5\text{C}$. If the end-of-string symbol $\#$ is appended to the concatenated strings, then the output of the transformation is $\$6\text{AGCACAGCGGCCTTA}\$5\$4\#\text{TTCC}\$2\$3\text{G}\$1\text{C}$.

Concatenated-BWT. This transformation, denoted concatBWT , concatenates all the strings into a single string using a single end-of-string symbol $\$$ as separator and appending one additional character, the final end-of-string symbol, here denoted by $\#$, which is smaller than all other characters. Then the classic BWT is applied to this string. This is the method implicitly applied when using a tool for classic BWT construction, so probably this is the most commonly used method. Given the collection $\mathcal{M} = (T_1, T_2, \dots, T_m)$,

$$\text{concatBWT}(T_1, \dots, T_m) = \text{BWT}(T_1\$T_2\$ \cdots T_m\#\#).$$

The concatBWT can be computed using the SA of the concatenation of the input strings, similarly to Eq. (4), as follows:

$$\text{concatBWT}(\mathcal{M})[i] = \begin{cases} T[\text{SA}[i] - 1] & \text{if } \text{SA}[i] > 1, \\ \# & \text{if } \text{SA}[i] = 1. \end{cases} \quad (5)$$

Because of the extra end-of-string symbol, the additional suffix starting with $\#$ is the smallest, and therefore, we have an additional dollar in the first position of the transform. To facilitate the comparison with the other transforms, we denote by *adapted concatBWT*, the transform obtained by removing the first symbol from concatBWT and replacing the $\#$ by $\$$.

► **Example 5.** Let us consider the collection $\mathcal{M} = (\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA})$. Then $\text{concatBWT}(\mathcal{M}) = \$ACAGCAGCGGCCTAT\$\#\text{TCTC}\$\$G\$C$, as shown in Table 3. The LF-mapping consists of a single cycle, which is (1 2 8 21 17 3 12 15 11 23 18 4 9 14 25 6 13 16 26 24 5 20 27 7 10 22 28 19). The adapted concatBWT is $\text{ACAGCAGCGGCCTAT}\$\#\text{TCTC}\$\$G\$C$.

Note that the $\#$ -symbol at the end of the string is a necessary addition, as without it, the LF-mapping, and thus, backward search, may not work correctly. This is because in that case, a suffix of the concatenated string may be a proper prefix of another suffix. This is the BWT variant treated in [29, 80].

3.3 A short literature overview of BWT variants

Until a few years ago, all methods and algorithms extending the Burrows-Wheeler Transform (BWT) to string collections were considered equivalent. In the previous subsections, we described different types of approaches and the specific characteristics of each BWT variant. In this subsection, we survey the main papers where the BWT variants have been treated, often using different algorithmic strategies. The corresponding tools are listed in Table 1.

■ **Table 3** Concatenation-based BWT variants: the multidollar-BWT (left) and the concat-BWT (right) of the string collection $\mathcal{M} = (\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA})$. We use different colors to identify the characters preceding different identical suffixes. See Section 3.2 for more details.

i	multidollar-BWT			concatenated-BWT			i
	SA	rotation			rotation	SA	
				\$	#	28	1
1	5	$\$1\text{TG}\$2\text{GTCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	A	A	$\$ \#$	27	2
2	8	$\$2\text{GTCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	G	C	$\$ \text{CGA} \#$	23	3
3	13	$\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	C	A	$\$ \text{CGACC} \#$	17	4
4	17	$\$4\text{CGACC}\$5\text{CGA}\$6$	A	G	$\$ \text{GTCC} \#$	8	5
5	23	$\$5\text{CGA}\$6$	C	C	$\$ \text{TCA} \#$	13	6
6	27	$\$6$	A	A	$\$ \text{TG} \#$	5	7
7	4	$\text{A}\$1\text{TG}\$2\text{GTCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	G	G	$\text{A} \#$	26	8
8	16	$\text{A}\$4\text{CGACC}\$5\text{CGA}\$6$	C	C	$\text{A} \# \text{CGACC} \#$	16	9
9	26	$\text{A}\$6$	G	G	$\text{A} \# \text{TG} \# \text{GTCC} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	4	10
10	20	$\text{ACC}\$5\text{CGA}\$6$	G	G	$\text{ACC} \# \text{CGA} \#$	20	11
11	12	$\text{C}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	C	C	$\text{C} \# \text{CGA} \#$	22	12
12	22	$\text{C}\$5\text{CGA}\$6$	C	C	$\text{C} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	12	13
13	15	$\text{CA}\$4\text{CGACC}\$5\text{CGA}\$6$	T	T	$\text{CA} \# \text{CGACC} \# \text{CGA} \#$	15	14
14	11	$\text{CC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	T	A	$\text{CC} \# \text{CGA} \#$	21	15
15	21	$\text{CC}\$5\text{CGA}\$6$	A	T	$\text{CC} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	11	16
16	24	$\text{CGA}\$6$	$\$5$	$\$$	$\text{CGA} \#$	24	17
17	18	$\text{CGACC}\$5\text{CGA}\$6$	$\$4$	$\$$	$\text{CGACC} \# \text{CGA} \#$	18	18
18	1	$\text{CTGA}\$1\text{TG}\$2\text{GTCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	$\$6$	#	$\text{CTGA} \# \text{TG} \# \text{GTCC} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	1	19
19	7	$\text{G}\$2\text{GTCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	T	T	$\text{G} \# \text{GTCC} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	7	20
20	3	$\text{GA}\$1\text{TG}\$2\text{GTCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	T	C	$\text{GA} \#$	25	21
21	25	$\text{GA}\$6$	C	T	$\text{GA} \# \text{TG} \# \text{GTCC} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	3	22
22	19	$\text{GACC}\$5\text{CGA}\$6$	C	C	$\text{GACC} \# \text{CGA} \#$	19	23
23	9	$\text{GTCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	$\$2$	$\$$	$\text{GTCC} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	9	24
24	14	$\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	$\$3$	$\$$	$\text{TCA} \# \text{CGACC} \# \text{CGA} \#$	14	25
25	10	$\text{TCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	G	G	$\text{TCC} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	10	26
26	6	$\text{TG}\$2\text{GTCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	$\$1$	$\$$	$\text{TG} \# \text{GTCC} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	6	27
27	2	$\text{TGA}\$1\text{TG}\$2\text{GTCC}\$3\text{TCA}\$4\text{CGACC}\$5\text{CGA}\$6$	C	C	$\text{TGA} \# \text{TG} \# \text{GTCC} \# \text{TCA} \# \text{CGACC} \# \text{CGA} \#$	2	28

Linear-time algorithms for the computation of the original EBWT variant were recently presented in [18, 96, 5]. In particular, in [18], an adaptation of the well-known Suffix Array Induced Sorting (SAIS) algorithm of Nong et al. [94] is used. In [5], the computation is done via computing the Bijective BWT (BBWT) of a string which is obtained from the input multiset. In [96], a non-recursive suffix array construction is used.

The dolEBWT variant is studied in [41, 66]. In [41], the computation is based on the grammar compression of the text. In [66], a merging strategy is used to compute the BWT of a collection by successively merging the BWTs of smaller subsets. The mdolEBWT and mdolBWT variants, although based on different constructions – via the EBWT and via concatenation, respectively – produce essentially the same output string when the indices associated with the dollar symbols are ignored. Indeed, their output strings coincide up to the renaming of the dollar symbols, as detailed in Section 4. Notably, the LF-mapping associated with mdolEBWT decomposes into a number of cycles equal to the number of input strings, whereas for mdolBWT it forms a single cycle. However, the two transforms are easily related: one can be obtained from the other by shifting by one the indices associated with the dollar symbols, as stated in Lemma 6. Therefore, any tool computing one of these

two BWT variants can be easily adapted to compute the other. For this reason, in Table 1 where the indices associated to dollar symbols are ignored, we grouped together all tools that compute mdolEBWT and/or mdolBWT.

More in detail, the computation of mdolEBWT is explicitly treated in [6, 74, 112, 82, 46, 17, 81, 16, 42, 75]. Different algorithmic strategies are described in these papers: some algorithms implicitly sort the suffixes or build the generalized suffix array incrementally; others compute the transform by merging partial BWTs obtained from subsets of the collection; and some – such as the approach described in [81] – adopt a hybrid strategy, constructing the GSA starting from a single concatenated string and then adapting it to the collection setting. Instead, the mdolBWT variant is explicitly used in [9] to address the problem of finding the optimal order of concatenation of the strings, aiming at minimizing the number of runs in the resulting BWT.

The concatBWT variant, which uses a single dollar symbol as a separator and appends a final hash character, is adopted in [22, 97, 86]. This is also the variant implicitly computed by tools originally designed for single-string BWT construction.

4 Combinatorial properties of the different BWT variants

In this section, we study combinatorial properties of the five BWT variants introduced in Section 3. We give an overview in Table 4. Full proofs for Sections 4.1 and 4.2 can be found in the Supplementary Material of [33]. Section 4.3 is new, and Section 4.4 is a generalization of the respective results in [32].

First notice that all BWT variants except the EBWT append an end-of-string symbol (resp. separator-symbol) at the end of each input string. For this reason, we will in the following refer to these BWT variants as *separator-based* BWT variants.

Table 4 Overview of some properties of the BWT variants considered in this paper on the string collection $\mathcal{M} = (\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA})$. The colors in the example BWTs correspond to SAP intervals in separator-based variants, while the same characters are highlighted in the EBWT for showing their positions, see Section 4.2. Dynamicity is w.r.t. removal and addition of strings to the collection, see Section 4.3. The order of shared suffixes is discussed in Section 4.4.

BWT variant	example	order of shared suffixes	input order dependence	dynam- icity
<i>EBWT-based</i>				
EBWT($\mathcal{M}$)	GGGCTACTCACACCTCTAGCG	omega-order	no	yes
dolEBWT($\mathcal{M}$)	ACACAGGGCGCCTAT\$\$\$TCTC\$\$\$G\$C	lexicographic order	no	yes
mdolEBWT($\mathcal{M}$)	AGCACAGCGGCCTTA\$\$_6\$\$_5\$\$_1\$TCC\$\$_3\$\$_4\$G\$\$_2\$C	input order	yes	yes
<i>concatenation-based</i>				
mdolBWT($\mathcal{M}$)	AGCACAGCGGCCTTA\$\$_5\$\$_4\$\$_6\$TCC\$\$_2\$\$_3\$G\$\$_1\$C	input order	yes	yes
concatBWT($\mathcal{M}$)	\$ACAGCAGCGGCCTAT\$\$\$#TCTC\$\$\$G\$C	lex. order of subseq. string in input	yes	no

4.1 Differences between EBWT and separator-based variants

In all separator-based BWT variants, the m -length prefix of the resulting transform consists of a permutation of the last characters of the input strings T_d , since the smallest rotations are those starting with the separators. In the EBWT, on the other hand, these m characters appear interspersed with the other characters (see Table 4). In general, adding separators introduces a distinction, not present in the EBWT, between occurrences of a substring as

suffix and otherwise: in the EBWT, the left-context of a substring appears according to the omega-order of the corresponding rotations, while in the separator-based variants, the occurrences that are suffixes appear in one block at the start.

To demonstrate this, consider the substring **CC** in our running example. As can be seen in Table 2, **CC** occurs twice as suffix and once not as a suffix (in this case, this is a circular occurrence). The two preceding characters of the suffix occurrences, **A** and **T**, form one block in all separator-based BWTs, while in the EBWT, they are intermixed with the preceding **C** of the non-suffix occurrence. A similar observation can be made about the two occurrences of **C** as suffix, whose preceding characters (both **C**) appear in once block in the separator-based BWT variants, while they are intermixed with all other occurrences of **C** in the EBWT.

Moreover, observe that adding separator-symbols changes the order of the strings. In particular, appending **\$**'s to the strings has the consequence that no string is a proper prefix of another, and therefore, the omega-order and the lexicographic order on these strings coincide. This is true whether the same separator-symbol or different symbols are used. An example is given in Figure 1, where we also show the difference between $\text{mdolEBWT}(\mathcal{M})$ and $\text{mdolBWT}(\mathcal{M})$.

EBWT($\{\text{GTC}, \text{GT}\}$)		EBWT($\{\text{GTC}\$, \text{GT}\$}\}$)		EBWT($\{\text{GTC}\$, \text{GT}\$}_1, \text{GT}\$}_2\}$)		BWT($\text{GTC}\$, \text{GT}\$}_1, \text{GT}\$}_2$)	
CGT	T	\$ GT	T	\$ ₁	C	\$ ₁ GT \$ ₂ GTC	C
GTC	C	\$ GTC	C	\$ ₂	T	\$ ₂ GTC \$ ₁ GT	T
GT	T	C \$ GT	T	C \$ ₁	T	C \$ ₁ GT \$ ₂ GT	T
TCG	G	GT \$	\$	GT \$ ₂	\$ ₂	GT \$ ₂ GTC \$ ₁	\$ ₁
TG	G	GTC \$	\$	GTC \$ ₁	\$ ₁	GTC \$ ₁ GT \$ ₂	\$ ₂
		T \$ G	G	T \$ ₂	G	T \$ ₂ GTC \$ ₁ G	G
		TC \$ G	G	TC \$ ₁	G	TC \$ ₁ GT \$ ₂ G	G
EBWT($\mathcal{M}$)		dolEBWT($\mathcal{M}$)		mdolEBWT($\mathcal{M}$)		mdolBWT($\mathcal{M}$)	

■ **Figure 1** The EBWT of the three sets $\{\text{GTC}, \text{GT}\}$, $\{\text{GTC}\$, \text{GT}\$}\}$, and $\{\text{GTC}\$, \text{GT}\$}_1, \text{GT}\$}_2\}$, and the BWT of the concatenated string $\text{GTC}\$, \text{GT}\$}_1, \text{GT}\$}_2$, corresponding to EBWT, dolEBWT, mdolEBWT, and mdolBWT of the same input. As can be seen, the result is different in each case. See Section 4.1 for a discussion.

Let us denote by $\doteq$ that two transforms are equal up to renaming the dollar symbols. The following lemma states that the transformation mdolBWT and mdolEBWT produce the same string, up to renaming dollars, and that dolEBWT is equivalent to first lexicographically sorting the input strings, and then applying either the mdolBWT or the mdolEBWT .

► **Lemma 6.** *Let $\mathcal{M} = (T_1, T_2, \dots, T_m)$ be a string collection. Then*

1. $\text{mdolBWT}(\mathcal{M}) \doteq \text{mdolEBWT}(\mathcal{M})$. Moreover, $\$d$ in $\text{mdolEBWT}(\mathcal{M})$ is replaced by $\$_{d-1}$ in $\text{mdolBWT}(\mathcal{M})$, and $\$1$ by $\$m$. Formally, $\text{mdolBWT}(\mathcal{M}) = \text{EBWT}(\{T_d\$_{d-1} \mid d = 1, \dots, m, \text{ with } T_0 = T_m\})$.
2. $\text{dolEBWT}(\mathcal{M}) \doteq \text{mdolBWT}(\text{lex}(\mathcal{M})) \doteq \text{mdolEBWT}(\text{lex}(\mathcal{M}))$, where $\text{lex}(\mathcal{M})$ denotes the lexicographic order of the strings in $\mathcal{M}$.

In the following section, we will concentrate on separator-based BWT variants only and show how they differ from each other.

4.2 Interesting intervals and SAP-intervals

Let us call a string U a *shared suffix* if there exist at least two strings $T_d, T_{d'} \in \mathcal{M}$, $d \neq d'$, such that U is a suffix of both T_d and $T_{d'}$. Let b denote the lexicographic rank of the smallest rotation beginning with $U\$$ (where $\$$ now stands for *any* separator-symbol) and e that of the

largest. In other words, $[b, e]$ is the SA-interval of $U\$$. We refer to $[b, e]$ as an *SAP-interval*, following [39]³. If there exist $d \neq d'$ such that U is a suffix of both T_d and $T_{d'}$ and the preceding character in T_d is distinct from the preceding character in $T_{d'}$, then $[b, e]$ is called an *interesting interval* [32, 33].

The next lemma follows from the fact that no two distinct substrings ending in $\$$ can be one the prefix of the other.

► **Lemma 7.** *Any two distinct interesting intervals are disjoint. Moreover, any two distinct SAP-intervals are disjoint.*

Interesting intervals allow us to locate precisely how and where two separator-based BWT variants of the same multiset can differ, which we state formally in the next proposition. For the proof, see the Supplemental Material of [33].

► **Proposition 8.** *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 sets of the positions of the dollars in L_1 resp. L_2 . If $\mathcal{I}_1 \neq \mathcal{I}_2$ then there exist $d \neq d'$ such that T_d is a proper suffix of $T_{d'}$.*

Thus, the differences between two separator-based BWT variants of the same multiset can be found only in interesting intervals (Property 1). To see Property 2, consider the following example:

► **Example 9.** Consider the mdolEBWT of the same multiset $\{\text{GCA}, \text{CA}\}$, given in two different orders: $\text{mdolEBWT}(\text{GCA}, \text{CA}) = \text{AACCG}\$2\$1$, while $\text{mdolEBWT}(\text{CA}, \text{GCA}) = \text{AACC}\$1\text{G}\$2$. This is because the interesting interval $[5, 6]$ associated with the shared suffix CA contains a dollar and a G , which change place according to the input order.

Proposition 8 implies that the differences between two separator-based BWT variants of the same multiset can be fully explained based on what rule is used to break ties for shared suffixes. We will see in Section 4.4 how the different BWT variants determine this tie-breaking rule.

4.3 Dynamicity of BWT variants

In this subsection, we study the dynamicity of the five BWT variants presented: what happens to the transforms if one or more of the strings are removed from (or, symmetrically, added to) the collection? Are the transforms robust to these operations? In other words, is the relative order of characters preserved when strings are added or deleted? These characteristics are particularly important for adaptive indexing structures, as they could allow efficient updates when the input changes dynamically.

We will see that all except the concatBWT are robust w.r.t. removal or addition of strings.

We recall a string S is a *subsequence* of a string T if S can be obtained from T by deleting zero or more characters without changing the order of the remaining characters. More formally, given the string $T = T[1..|T|]$, a string $S = S[1..|S|]$ is a subsequence of T , denoted $S \sqsubseteq T$, if there exists a sequence of integers $i_1, i_2, \dots, i_{|S|}$, such that $1 \leq i_1 < i_2 < \dots < i_{|S|} \leq |T|$ and $S[j] = T[i_j]$ for every $j = 1, \dots, |S|$.

³ SAP stands for “same as previous”

The following lemma follows from the fact that, for $T_d \in \mathcal{M}$, any two cyclic rotations (resp. suffixes) of string T_d preserve their relative lexicographic order when they are considered in the ω -sorted (resp. lexicographically sorted) list of all cyclic rotations (resp. suffixes) of the string collection $\mathcal{M}$.

► **Lemma 10.** *Given a multiset $\mathcal{M}$, for every string $T_d \in \mathcal{M}$,*

1. $\text{BWT}(T_d) \subseteq \text{EBWT}(\mathcal{M})$,
2. $\text{BWT}(T_d\$) \subseteq \text{dolEBWT}(\mathcal{M})$,
3. $\text{BWT}(T_d\$) \subseteq \text{mdolEBWT}(\mathcal{M})$, up to renaming the character \$,
4. $\text{BWT}(T_d\$) \subseteq \text{mdolBWT}(\mathcal{M})$, up to renaming the character \$,
5. $\text{BWT}(T_d\$) \subseteq \text{concatBWT}(\mathcal{M})$, up to renaming the character \$.

When subsets of the string collection are considered, rather than a single string, the following proposition shows that Lemma 10 can be generalized for EBWT and dolEBWT.

► **Proposition 11.** *Let $\mathcal{M}$ be a multiset of strings and let $\mathcal{S} \subseteq \mathcal{M}$. Then:*

1. $\text{EBWT}(\mathcal{S}) \subseteq \text{EBWT}(\mathcal{M})$, and
2. $\text{dolEBWT}(\mathcal{S}) \subseteq \text{dolEBWT}(\mathcal{M})$.

In order to study the behavior of the input order dependent BWT variants when sub-collections of strings are considered, we use the following definition. Given an ordered collection $\mathcal{M} = (T_1, T_2, \dots, T_m)$, the ordered collection $\mathcal{S} = (S_1, S_2, \dots, S_p)$ is a *sub-collection* of $\mathcal{M}$ (denoted $\mathcal{S} \preceq \mathcal{M}$) if there exists a sequence of integers $i_1, i_2, \dots, i_p$, such that $1 \leq i_1 < i_2 < \dots < i_p \leq m$ and $S_j = T_{i_j}$ for every $j = 1, \dots, p$.

In the following proposition, we consider mdolEBWT and mdolBWT. In both of these transformations, the relative order of two identical suffixes is determined by the order of the distinct dollar symbols. Therefore, if the sub-collection of strings maintains the same relative order, then the output of each transformation applied to the sub-collection is a subsequence of the output obtained with the entire string collection.

► **Proposition 12.** *Let $\mathcal{M}$ be an ordered string collection and let $\mathcal{S} \preceq \mathcal{M}$. Then:*

1. $\text{mdolEBWT}(\mathcal{S}) \subseteq \text{mdolEBWT}(\mathcal{M})$, up to renaming the dollar symbols used for the strings in $\mathcal{S}$, and
2. $\text{mdolBWT}(\mathcal{S}) \subseteq \text{mdolBWT}(\mathcal{M})$, up to renaming the dollar symbols used for the strings in $\mathcal{S}$.

The previous two propositions show that all the considered BWT-variants can be used as basis of dynamic compressed indexing structures, since they allow the removal of a string from the collection simply by deleting the characters of the string from the output of the transformation on the full string collection. Similarly, they enable the use of the merge of the transforms on sub-collections to obtain the output of a collection containing all the strings.

The following example shows that, on the other hand, the BWT-variant we refer to as concatBWT, i.e., concatenating the input strings using the same dollar-symbol to separate them, does not satisfy this property.

► **Example 13.** Let us consider the collection $\mathcal{M} = (\text{CCA}, \text{ACA}, \text{TCA})$. Then $\text{concatBWT}(\mathcal{M}) = \$AAACCC\$TCA\#\$$. When the sub-collection $\mathcal{S} = (\text{CCA}, \text{ACA})$ is considered, then $\text{concatBWT}(\mathcal{S}) = \$AACCC\$AC\#$, which is not a subsequence of $\text{concatBWT}(\mathcal{M})$.

4.4 Input order and output order

It is clear that the first m characters of a separator-based BWT variant constitute an SAP-interval, namely the one corresponding to $U = \epsilon$. Since this m -length prefix contains every string T_d exactly once, it determines a permutation of the input strings. This is the permutation of the indices $1 \leq d \leq m$ found in the m first entries of GCA, i.e., in $\text{GCA}[1..m]$.⁴ This permutation then determines the order in which the respective suffixes appear in the interesting intervals, which, by Proposition 8, determines the final transform.

As we have a multiset in input, the relative order of equal strings $T_d = T_{d'}$, where $d \neq d'$, does not matter. We can thus use a meta-string to model the input and output permutations of $\mathcal{M}$, as follows. Let $m' \leq m$ denote the number of distinct strings in $\mathcal{M}$. We define a meta-string t of length m , with characters from an ordered alphabet of size m' , to model the *input permutation* of $\mathcal{M} = (T_1, T_2, \dots, T_m)$, by replacing every string T_d by a meta-character according to its lexicographic rank among all distinct input strings. For example, for $\mathcal{M} = (\text{ACA}, \text{TGA}, \text{ACA}, \text{GAA}, \text{TGA}, \text{TGA})$, we get $t = \text{acabcc}$. Now the m -length prefix of the BWT, corresponding to the *output permutation*, can be seen as another meta-string, which is, in fact, a permutation of t . As this string depends on the BWT variant employed, each of these variants can be viewed as a string transform, acting on the input meta-string t . We will denote these transforms as π_{dolE} , π_{mdolE} , π_{mdol} , and π_{concat} , respectively.

► **Example 14.** Let $\mathcal{M} = (\text{ACA}, \text{TGA}, \text{ACA}, \text{GAA}, \text{TGA}, \text{TGA})$. Then $t = \text{acabcc}$ and $\pi_{\text{mdolE}}(t) = \pi_{\text{mdol}}(t) = \text{acabcc}$, $\pi_{\text{dolE}}(t) = \text{aabccc}$, and $\pi_{\text{concat}}(t) = \text{ccacab}$.

■ **Figure 2** The first m rotations or suffixes and the output permutations for all separator-based BWTs of $\mathcal{M} = (\text{ACA}, \text{TGA}, \text{ACA}, \text{GAA}, \text{TGA}, \text{TGA})$.

mdolEBWT	π_{mdolE}	mdolBWT	π_{mdol}	dolEBWT	π_{dolE}	concatBWT	π_{concat}
\$ ₁ ACA	a	\$ ₁ TGA\$ ₂ ACA..	a	\$ACA	a	\$#	c
\$ ₂ TGA	c	\$ ₂ ACA\$ ₃ GAA..	c	\$ACA	a	\$ACA\$GAA\$. .	c
\$ ₃ ACA	a	\$ ₃ GAA\$ ₄ TGA..	a	\$GAA	b	\$GAA\$TGA\$. .	a
\$ ₄ GAA	b	\$ ₄ TGA\$ ₅ TGA..	b	\$TGA	c	\$TGA\$#	c
\$ ₅ TGA	c	\$ ₅ TGA\$ ₆	c	\$TGA	c	\$TGA\$ACA\$. .	a
\$ ₆ TGA	c	\$ ₆	c	\$TGA	c	\$TGA\$TGA\$#	b

It is easy to see that both $\pi_{\text{mdolE}}(t)$ and $\pi_{\text{mdol}}(t)$ are equal to t , since the dollar-symbols are ordered according to the input order. For the dolEBWT, the rank of $\$T_d$ equals the lexicographic rank of T_d among all input strings (Lemma 6), i.e., $\pi_{\text{dolE}}(t)$ is the lexicographically sorted permutation of the input string t . The situation is more complex for the concatBWT. Since # is the smallest character and it is in the final position of the concatenated string $T_1\$T_2\$ \dots T_m\$\#$, the last string T_m of the input will be the first, while for the other strings T_d , the lexicographic rank of *the following string* T_{d+1} decides the order. Moreover, if T_d and $T_{d'}$ are followed by two equal strings $T_{d+1} = T_{d'+1}$, then the lexicographic order of the strings following the following strings will break the tie, and so on. In other words, π_{concat} is essentially the BWT of t . (We thank Massimiliano Rossi for this insight.)

⁴ Or, equivalently, in the document array DA.

► **Lemma 15.** *Let t be the input order meta-string of $\mathcal{M} = (T_1, \dots, T_m)$. Then $\pi_{\text{concat}}(t) = \text{BWT}^*(t)$, where for a string u , $\text{BWT}^*(u)$ is defined as $\text{BWT}(u\$)$ from which the end-of-string symbol $\$$ has been removed.*

► **Example 16.** Continuing Example 14, $\text{BWT}(t\$) = \text{BWT}(\text{acabcc}\$) = \text{cc}\$ \text{acab}$, and therefore, $\text{BWT}^*(t) = \text{ccacab}$.

Lemma 15 has important consequences for the BWT variant concatBWT . First, it means that in most cases, the transforms mdolBWT and concatBWT will produce different outputs, given the same input. This is because $\pi_{\text{concat}}(t)[1] = t[m] = \pi_{\text{mdol}}(t)[m]$, and therefore, if the strings in $\mathcal{M}$ are all distinct, then for any input order, the output permutations induced by mdolBWT and concatBWT are distinct: in this case, $t[1] \neq t[m]$, and thus $\pi_{\text{mdol}}(t) \neq \pi_{\text{concat}}(t)$ holds for all t . This means that, in whatever order the strings are given, on most string sets the resulting transforms mdolBWT and concatBWT will differ.

Another consequence is that concatBWT cannot produce all possible transforms. Already for $m = 3$, on an input collection of three distinct strings, the mapping corresponding to concatBWT is not surjective, since the mapping π_{concat} for $m' = m = 3$ is as follows: $\text{abc} \mapsto \text{cab}$, $\text{acb} \mapsto \text{bca}$, $\text{cab} \mapsto \text{bca}$, $\text{bac} \mapsto \text{cba}$, $\text{bca} \mapsto \text{acb}$, and $\text{cba} \mapsto \text{abc}$. In particular, no t maps to bac .

It has been shown experimentally that more than half of binary and ternary strings of length between 10 and 20 do not lie in the image of the function BWT^* , with the percentage of those not in the image increasing with increasing length [59]. These results seem to indicate that, in fact, the majority of multi-permutations cannot be produced by concatBWT (for a concrete example, see Section 5).

On the other hand, it follows from Lemmas 6 and 15 that every separator-based transform can be simulated by both mdolBWT and mdolEBWT , in the sense that there exists an input order of $\mathcal{M}$ such that the resulting mdolBWT and mdolEBWT will be equal to the transform, up to possibly renaming dollars. We summarize:

► **Theorem 17.** *Given a string collection $\mathcal{M} = (T_1, T_2, \dots, T_m)$ and L the output of a separator-based BWT variant on $\mathcal{M}$, there exists a permutation τ of $\mathcal{M}$ such that*

$$L \hat{=} \text{mdolEBWT}(T_{\tau(1)}, T_{\tau(2)}, \dots, T_{\tau(m)}) \hat{=} \text{mdolBWT}(T_{\tau(1)}, T_{\tau(2)}, \dots, T_{\tau(m)}).$$

We will use the insight of Theorem 17 in the next section.

5 Reducing the number of runs

Recall that $L \hat{=} L'$ if the L and L' are equal up to renaming the dollar symbols. In Section 4, we showed that all separator-based BWT variants can be simulated by mdolEBWT or by mdolBWT (Thm. 17): if L is the output of one of these variants, then the input collection can be permuted in such a way that mdolEBWT and mdolBWT will produce a transform which differs from L only in the denomination of the dollar signs. This justifies the following definition.

Given a string collection $\mathcal{M}$, let $\mathfrak{S}_{\mathcal{M}}$ denote the string family obtained by applying mdolEBWT to all possible orders of the strings in $\mathcal{M}$. Formally, for $\mathcal{M} = (T_1, \dots, T_m)$,

$$\mathfrak{S}_{\mathcal{M}} = \{\text{mdolEBWT}(\{T_{\tau(d)}\$d \mid 1 \leq d \leq m\}) \mid \tau \text{ is permutation of } \{1, \dots, m\}\}.$$

This string family coincides with the one introduced in [14] (albeit there given with a different definition).

In this section, we focus on reducing the number of runs of the BWT, commonly referred to as r . This is a fundamental parameter, since the BWT is primarily used for compressed text indexes, such as the FM-index [48], the r -index [54], or the extended r -index [19, 20], whose storage space depends on r . Moreover, the value n/r , the average runlength of the BWT, is increasingly being used as a parameter describing the input data. This is because it is well known that the more repetitive the input data, the fewer runs the BWT tends to have.

Therefore, we are interested in the following:

► **Definition 18.** *Given a string collection $\mathcal{M}$, we let $r_{\text{opt}}(\mathcal{M}) = \min\{\rho(L) \mid L \in \mathfrak{S}_{\mathcal{M}}\}$, and call a string $L \in \mathfrak{S}_{\mathcal{M}}$ optimal if $\rho(L) = r_{\text{opt}}(\mathcal{M})$. When $\mathcal{M}$ is clear from the context, we just write r_{opt} for $r_{\text{opt}}(\mathcal{M})$.*

As we saw in Section 4.2, any differences between two transforms L and L' on the same collection $\mathcal{M}$, in whatever order, necessarily lie within interesting intervals. This ignores the naming of the separators, which is justified, given that no tool outputs the transform with *different dollar-symbols*, even if it computes the mdolBWT or mdolEBWT, which means that internally it handles the dollar-symbols as different symbols.⁵ Thus, for the sake of compression, it is appropriate to treat the output independently of the names of the dollar-symbols.

Bentley et al. [9] gave a linear-time algorithm for minimizing the number of runs but they gave no implementation. An implementation called `optimalBWT` [99] was presented by Cenzato et al. [30] which computes an optimal transform L . To understand how the algorithm works, let us consider an interesting interval $[b, e]$. It is clear that within this interval, the number of runs can be minimized if all occurrences of same characters are grouped together. Therefore, if there are k distinct characters in $L[b..e]$, then k is the minimum number of runs achievable in this interval. Now let's say we have grouped all characters into k runs and $L[b..e] = a_1^{r_1} a_2^{r_2} \dots a_k^{r_k}$. If the previous character $L[b-1] \neq a_1$ then it may be possible to reduce the number of runs by swapping some other run, say $a_i^{r_i}$ with the first run $a_1^{r_1}$, namely if $L[b-1] = a_i$. Similarly, one may be able to reduce the number of runs by swapping some other run with the last one $a_k^{r_k}$. The problem is that this choice may not be unique, and when there are consecutive interesting intervals then it may not be possible to decide until these interesting intervals end. Bentley et al. give an algorithm for making these choices, modeling the problem as what they refer to as a *tuple ordering problem*, which can be turned into a shortest path problem on a DAG and solved optimally in linear time.

Now consider our toy example $\mathcal{M} = (\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA})$. In Table 5, we give different transforms from $\mathfrak{S}_{\mathcal{M}}$, with the number of runs between 19 (top row, input order as given above), and 14 (bottom row), which is the minimum r_{opt} . In the middle rows we give two heuristics, which we explain next.

Recall that the colex order (also referred to as *reverse lexicographic order* (rlo)) is defined as $S <_{\text{colex}} T$ if $S^{\text{rev}} <_{\text{lex}} T^{\text{rev}}$. It is easy to see that if the input strings are in colex order, then every interesting interval will have the minimum number of runs. On our example, the colex results in 18 runs, one less than the input order. In particular, several existing tools (BCR [6, 7], `ropeBWT2` [107, 74], `ropeBWT3` [108, 75]) have the option to output the BWT with respect to the colex order. This was already shown in [39] to significantly reduce the number of runs.

⁵ Some tools are able to output the indices corresponding to the dollars, such as BCR and BEETL [6, 7, 8], but this is done in a separate file, or the indices can be extracted from the DA or the GSA, such as eGAP [46, 45].

■ **Table 5** For the input collection $\mathcal{M} = (\text{CTGA}, \text{TG}, \text{GTCC}, \text{TCA}, \text{CGACC}, \text{CGA})$, we show the resulting transform for the input order, the optimal order, and two heuristics (details see text). Note that in the count of the number of runs, all dollars are considered equal.

	transform	number of runs r
mdolEBWT($\mathcal{M}$)	AGCACAGCGGCCTTA\$ ₆ \$ ₅ \$ ₁ TTCC\$ ₃ \$ ₄ G\$ ₂ C	19
colexBWT($\mathcal{M}$)	AAACCGCGGCCTAT\$ ₂ \$ ₄ \$ ₃ TTC\$ ₅ \$ ₁ G\$ ₆ C	18
plusBWT($\mathcal{M}$)	AAACCGGGCGCCTTA\$ ₂ \$ ₅ \$ ₁ TTCC\$ ₄ \$ ₃ G\$ ₆ C	15
optBWT($\mathcal{M}$)	AAAGCCGGGCCTTA\$ ₃ \$ ₆ \$ ₂ TTCC\$ ₅ \$ ₁ G\$ ₄ C	14

Several other heuristics were introduced in [14], all based on the idea of grouping all occurrences in interesting intervals into one single run and possibly swapping the runs within the interesting interval. The one that performed best was plusBWT, which during the construction greedily puts as first run the character immediately before the interesting interval and as last run the character immediately after, if these characters are present. In Table 7, we report experimental results, over benchmarks that are described in Table 6, that show that both of these heuristics get very close to the optBWT. Interestingly, even randomly permuting the runs within interesting intervals results in a slight improvement over the colex order. For details, see [14, 6].

What is also remarkable is the improvement that can be achieved by the optimalBWT as compared to the input order, namely up to a multiplicative factor of over 31, on real biological data [30, 14], see Table 7. In other words, a resulting data structure built on top of the BWT would be far smaller if the optBWT was used, rather than just any BWT transform. Indeed, in [31], first experimental results on the size of the resulting r -index on real datasets are presented.

Another interesting aspect is the following: in order to reduce the number of runs, a necessary step is grouping all equal characters together within the interesting intervals. The larger the interesting intervals, the fewer the fraction of permutations of the string collection that do this. This implies that the probability that some arbitrary input order – as given by the order in which the collection comes, e.g. as downloaded from a database – will group these characters together is very small. This explains why the potential gain from applying the optBWT (or, indeed, one of the heuristics given above) is highest on very large sets of very similar short strings: many suffixes will be shared by many input strings, which produces long interesting intervals. (See [32, 33] for a more detailed analysis.)

Now consider the following example, which shows that with the concatBWT, it can be impossible to minimize the number of runs of the BWT. In other words, on some data sets, whatever the input order, concatBWT($\mathcal{M}$) will always have more runs than necessary.

► **Example 19.** Let $\mathcal{M} = \{\text{ACA}, \text{TGA}, \text{GAA}\}$, then $t = \text{acb}$, and the output order corresponding to the colexicographic order of the strings is bac . As it happens, this order yields the minimum number of runs among all strings in $\mathfrak{S}_{\mathcal{M}}$, with $\text{colexBWT}(\mathcal{M}) = \text{optBWT}(\mathcal{M}) = \text{AAAACGG\$AT\$\$}$, which has 7 runs. However, as we saw in Section 4.4, the string bac does not lie in the image of π_{concat} , so no permutation of the strings in $\mathcal{M}$ will yield this order for concatBWT. In particular, the $\text{colexBWT}(\mathcal{M}) = \text{AAAACGG\$AT\$\$}$ has 7 runs, while all concatBWTs have at least 8: $\text{AAAGACG\$AT\$\$}$, $\text{AAACGAG\$AT\$\$}$, $\text{AAAAGCG\$AT\$\$}$, $\text{AAAGCAG\$AT\$\$}$, $\text{AAACAGG\$AT\$\$}$. (Here we are using the adapted variant of the concatBWT, for better comparison.)

■ **Table 6** Features of the datasets included in our experiments. From left to right, we report the dataset code and description, the BWT length, the maximum string length, the number of sequences, and the ratio between the BWT length and the number of runs of the optBWT.

Dataset	Description	BWT length	Max length	Number of sequences	n/r_{opt}
SRR7494928-30	<i>Epstein Barr Virus</i>	3,225,480,720	101	9,648,932	79.25
ERR732065-70	<i>HIV-virus</i>	1,345,713,812	150	8,912,012	116.62
SRR12038540	<i>SARS-CoV-2 RBD</i>	1,690,229,250	50	33,141,750	113.71
ERR022075_1	<i>E. Coli str. K-12</i>	2,294,730,100	100	22,720,100	32.23
SRR059298	<i>Deformed wing virus</i>	2,455,299,082	72	33,634,234	50.75
SRR065389-90	<i>C. Elegans</i>	14,095,870,474	100	139,563,074	15.30
SRR2990914_1	<i>Sindibis virus</i>	15,957,722,119	36	431,289,787	151.62
ERR1019034	<i>H. Sapiens</i>	123,506,926,658	100	1,222,840,858	11.37
<i>pdb_seqres</i>	<i>proteins</i>	241,121,574	16,181	865,773	14.33

■ **Table 7** Number of runs of the mdolEBWT (resp. mdolBWT) with sequences presented in input order (inputBWT) and in two heuristic orders, colexBWT and plusBWT, compared to the optBWT for all datasets in Table 6. In the rightmost column we report the ratio between the number of runs of the inputBWT (r_{input}) and the optBWT (r_{opt}). Results reported from [14].

Dataset	inputBWT	Heuristic string orders		optBWT	r_{input}/r_{opt}
		colexBWT	plusBWT		
SRR7494928-30	254,663,327	41,730,649	41,372,530	40,700,607	6.26
ERR732065-70	48,727,709	11,941,093	11,766,827	11,539,661	4.22
SRR12038540	209,136,502	17,026,009	15,226,766	14,864,523	14.07
ERR022075_1	259,821,570	75,846,202	74,529,428	71,203,469	3.65
SRR059298	249,873,376	50,495,777	49,619,150	48,376,632	5.17
SRR065389-90	2,251,887,226	968,098,124	954,489,749	921,561,895	2.44
SRR2990914_1	3,313,966,937	109,772,697	108,466,351	105,250,120	31.49
ERR1019034	23,084,021,291	11,312,737,256	11,179,873,104	10,860,229,434	2.13
<i>pdb_seqres</i>	17,971,532	16,862,960	16,848,496	16,829,629	1.07

6 BWT for string collections in bioinformatics

Recent advances in DNA and RNA sequencing technologies led to the explosion of the amount of *in silico* genomic data to be assembled, investigated, stored and compared. In this final section we will overview the applications in bioinformatics that benefit from the efficiency of BWT-based indexing of string collections.

Both DNA and RNA sequences can be seen as strings over an alphabet of size four: the set of nucleotides that compose them. Furthermore, as we will see below, in all the applications we will mention, sequences are highly repetitive and the different strings of the collection encode redundant data: an ideal target for BWT based tools that are specifically designed for string collections.

When the string collection consists of *raw sequencing data*, that is, a large amount of fragments that directly result from the sequencing process, the most requested tasks can be:

Assembling. Size and accuracy details have changed across the time and also change from a technology to another, but all sequencing machines are not able to process molecules of arbitrary size. As a consequence, the typical procedure is to: first, create a redundant number of copies of the sequence while still *in vitro*; second, cut (this can be done using specific enzymes) the sequences into small enough fragments randomly enough to ensure

that distinct copies are cut at different sites; third, perform the actual sequencing that turn the sequence into *in silico* data; fourth, exploiting the redundancy of the original copies and hence fragments overlap, build a layout for the fragments that enables to reconstruct the original sequence. This last task is named *Fragment Assembly*, and its efficient solution played a crucial role in the historical human genome project. For decades the problem of fragment assembly has been the *Holy Grail* of algorithmic tasks for bioinformatics, having to deal with millions of DNA fragments, each of a few hundreds of length, that had to be jointly analysed, looking for local similarities among them [111].

Mapping. When the newly sequenced genome belongs to species for which a fully assembled *reference* genome already exists, then one does not need to perform a *de novo* assembly out of the fragments, as the reference can be exploited. In this case, the strings of the collection are, in general, not as many as in the case of the assembly task, because redundancy is not required as much as there. However, a huge amount of strings must be mapped onto the reference genome, dealing with sequencing errors, repetitions inside the genome that make the mapping loci ambiguous, as well as the physiological differences among the query genome and the reference [71, 76].

Haplotype Matching. A Haplotype is a set of genetic markers or DNA variations that are inherited together from a single parent. Haplotype Matching asks for finding long matches between sequences within a given large collection of aligned genetic sequences (that are, indeed, the haplotypes). The task is relevant because these long matches are candidates to be regions that are identical by descent (IBD) from a common ancestor. Even for this task BWT based methods have been designed: the positional BWT [44, 113, 114, 110] for linear strings, and more recently the GBWT for Pangenome Graphs [109].

Assembly free (and reference free) variant calling. Variant calling is the fundamental task that has to be performed with a genome, and it consists of detecting individual traits and genetic diversity of the query genome to discover genetic based diseases, perform personalised medicine, as well as to understand evolution (when variants are annotated at population scale). When a reference genome is not available, this task is performed directly on raw data without a preliminary assembling phase, and hence this requires the investigation and comparison of (pairs of) large collections of strings [104, 102, 103, 70].

When, instead, the string collections are multiple genomes, or several RNA isoforms, or belong to possibly many different genomes, we have a (possibly large) set of strings that are as long as billions of letters. In these cases, the downstream analyses tasks are different, and they include:

Comparative genomics and variant calling. Here two or more (whole) genomes must be compared, one of them possibly being *the* reference. Like for the above mentioned tasks, criticalities arise because of the highly repetitive nature of genomes, and by the different nature of variants that one needs to detect: the so called *point mutations* or the *SNPs* that involve single letters, or the short *INDELs* that are the insertion or the deletion of a short fragment, up to the fragment level mutations: the deletions or insertion or duplication of a possibly large DNA fragment, or the translocation (that is, its position in the genome changes), or the copy number variations (fragments that are commonly present in multiple copies show a different number of them in different individuals). Moreover, when a DNA mutation event involves a whole fragment, there is in general a 50% probability that a copy is *inverted*, that is it occurs in the reversed direction. The comparison tasks are very complex and cannot be made by means of global alignments due to the fragment level mutations. Hence efficient indices of the strings are a basic toolkit for the various possible algorithmic strategies [20, 44].

Transcriptomics. With the so called new generation sequencing technologies, also *RNA-Seq* became a common task, that is to move to *in silico* also RNA the sequences. In particular, mRNA (messenger RNA) isoforms are very interesting to investigate as they are RNA fragments that encode the information that leads a coding part of the DNA (that is, a gene) to the synthesis of a protein. The transcription of this coding part of DNA into RNA, called *gene expression*, is a very interesting topic for molecular biologists and genetists because the same gene can express in different ways according to the cell it is (say, in the liver rather than in the skin), or according to external conditions (say, in a skin cell whether it is cold or it is being burned). The set of RNA isoforms that are in a certain cell under specific conditions is called the *transcriptome*, and this is a sort of a picture of gene expression. Investigating or comparing transcriptome(s) requires counting the abundance of each kind, being each of them a different concatenation of a subset of the fragments of the gene: again several strings, and again highly redundant, and hence a perfect target for a string collection index based on the BWT [119, 18, 20].

Metagenomics. Metagenomics is the (comparative) study of genetic material found in environmental samples (say, into the soil, or in sea water, or in an animal's gut), with the purpose of detecting and investigating (the distribution of) genomes of different species of microorganisms such as bacteria, viruses, and fungi. The ultimate goals of metagenomic is the study of microbial diversity, and the interactions: both mutual and with the ecosystem it belongs to. This requires various tasks that include profiling, assembling, comparing and classifying data consisting of a huge amount of fragments belonging to different individuals and different species. As such, the redundancy will be less than in the other applications; however, indexing these string collections with BWT is still fundamental to make efficient compression and analyses [67, 64, 14, 62, 63, 40].

Pangenomics. Using a single genome as *the* reference clearly introduces a bias for the discovery of genomic variations. Therefore, in the literature there came a new challenge that simultaneously deals with, and exploits, the widespread availability of sequencing data: using a *pangenome* rather than a genome. In [118], a pangenome was defined as “any collection of genomic sequences to be analyzed jointly or to be used as a reference”. In contrast to a *linear* reference, a pangenome reference aims to compactly represent the variation within a population by encoding the commonalities and differences among the underlying sequences. In the literature there are many suggestions for pangenome representations [77, 4, 28, 106, 55, 100, 79, 13, 12, 92, 26, 72] whose construction may benefit from efficient indexing of data as a starting step [81, 46, 53, 89, 98, 110, 18, 109, 35]. Some of these representations support efficient and accurate analyses tools such as comparison [1, 2, 51, 52, 50] or alignments [90, 11, 91, 10, 3, 36], that could be sped up with string collections indices. However, by now none of these graph structures has been acknowledged as the standard yet. The computational challenges posed by pangenomes often result in a trade-off between the efficiency and accuracy of the methods and the information content of the chosen representation. In this view, a valid option for certain tasks turns out to be to just index and investigate the collection of genomes, disregarding loci and alignment information and rather choosing speed and low memory usage [21, 68, 22, 44].

7 Conclusion

In this paper, we studied different methods for constructing the BWT of string collections (as opposed to individual strings). We showed that the methods in use are non-equivalent, in the sense that they produce different transforms, which differ not only in the denomination

of separator-symbols (“dollars”). We analyzed precisely where and how these differences occur (“interesting intervals”). The differences between the transforms extend to the number of runs r of the BWT, a central parameter in compressed text indexing.

We showed that several of these methods are input-order dependent, and that this can be exploited to significantly reduce the number of runs: by up to a factor of over 31, on real biological data. This is the multiplicative gain of the optBWT as compared to computing the transform on the collection in the order it comes in. We also saw that several simple heuristics come close to this optimum.

On the downside, we saw that one of the most common methods, which we refer to as concatBWT (concatenating the input strings and separating them with the same dollar), has several undesirable properties. First, it cannot produce all possible transforms, and can therefore in general not be used as a method for run minimization. Second, it is not robust with respect to addition or removal of strings from the collection (dynamicity). Finally, it is input-order dependent but in a fairly opaque manner (its order being in essence the BWT of the input order), which makes it difficult to control the output in any way.

Based on our findings, we make the following suggestions:

1. We advise to exercise care when using the method we term concatBWT, due to the issues listed above. It is a good method because it is simple and efficient, but the user (or programmer) should be aware of its shortcomings.
2. We believe that the definition of the number of runs of a string collection should be standardized to r_{opt} , the minimum number of runs among all possible separator-based transforms. This number is also fairly easy to approximate with heuristics, the simplest of which is colexBWT.

References

- 1 Mai Alzamel, Lorraine A.K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundam. Informaticae*, 175(1-4):41–58, 2020. doi:10.3233/FI-2020-1947.
- 2 Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Degenerate string comparison and applications. In *18th International Conference on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPICs*, pages 21:1–21:14, 2018. doi:10.4230/LIPICs.WABI.2018.21.
- 3 Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster online elastic degenerate string matching. In *29th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 105 of *LIPICs*, pages 9:1–9:10, 2018. doi:10.4230/LIPICs.CPM.2018.9.
- 4 Jasmijn A. Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Nat. Comput.*, 21(1):81–108, 2022. doi:10.1007/s11047-022-09882-6.
- 5 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing and indexing the bijective and extended Burrows-Wheeler transform. *Inf. Comput.*, 297:105153, 2024. doi:10.1016/J.IC.2024.105153.
- 6 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134–148, 2013. doi:10.1016/j.tcs.2012.02.002.
- 7 BCR_LCP_GSA. URL: https://github.com/giovannarosone/BCR_LCP_GSA.
- 8 BEETL. URL: <https://github.com/BEETL/BEETL>.
- 9 Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the Complexity of BWT-Runs Minimization via Alphabet Reordering. In *28th Annual European Symposium on*

- Algorithms (ESA)*, volume 173 of *LIPICs*, pages 15:1–15:13, 2020. doi:10.4230/LIPICs.ESA.2020.15.
- 10 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In *46th International Colloquium on Automata, Languages, and Programming, (ICALP)*, volume 132 of *LIPICs*, pages 21:1–21:15, 2019. doi:10.4230/LIPICs.ICALP.2019.21.
 - 11 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Elastic-degenerate string matching via fast matrix multiplication. *SIAM J. Comput.*, 51(3):549–576, 2022. doi:10.1137/20M1368033.
 - 12 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In *24th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 10508 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2017. doi:10.1007/978-3-319-67428-5_7.
 - 13 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, 2020. doi:10.1016/J.TCS.2019.08.012.
 - 14 Gianmarco Bertola, Anthony J. Cox, Veronica Guerrini, and Giovanna Rosone. A Class of Heuristics for Reducing the Number of BWT-Runs in the String Ordering Problem. In *35th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 296 of *LIPICs*, pages 7:1–7:15, 2024. doi:10.4230/LIPICs.CPM.2024.7.
 - 15 BigBWT. URL: <https://github.com/alshai/Big-BWT>.
 - 16 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. doi:10.1016/j.tcs.2020.11.041.
 - 17 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. doi:10.1089/cmb.2018.0230.
 - 18 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the Original eBWT Faster, Simpler, and with Less Memory. In *28th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 12944 of *Lecture Notes in Computer Science*, pages 129–142. Springer, 2021. doi:10.1007/978-3-030-86692-1_11.
 - 19 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. *r*-indexing the eBWT. In *28th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 12944 of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2021. doi:10.1007/978-3-030-86692-1_1.
 - 20 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. *r*-indexing the eBWT. *Inf. Comput.*, 298:105155, 2024. doi:10.1016/J.IC.2024.105155.
 - 21 Christina Boucher, Travis Gagie, Tomohiro I, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: streamed matching statistics with multi-genome references. In *31st Data Compression Conference (DCC)*, pages 193–202. IEEE, 2021. doi:10.1109/DCC50243.2021.00027.
 - 22 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. doi:10.1186/S13015-019-0148-5.
 - 23 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
 - 24 bwt-lcp-parallel. URL: <https://github.com/AlgoLab/bwt-lcp-parallel>.
 - 25 bwt-lcp-parallel. URL: <https://github.com/AlgoLab/bwt-lcp-em/>.

- 26 Thomas B  chler and Enno Ohlebusch. An improved encoding of genetic variation in a Burrows–Wheeler transform. *Bioinform.*, 36(5):1413–1419, 2019. doi:10.1093/bioinformatics/btz782.
- 27 cais. URL: <https://github.com/davidecenzato/cais>.
- 28 Vincenzo Carletti, Pasquale Foggia, Erik Garrison, Luca Greco, Pierluigi Ritrovato, and Mario Vento. Graph-based representations for supporting genome data analysis and visualization: Opportunities and challenges. In *12th IAPR-TC-15 International Workshop Graph-Based Representations in Pattern Recognition (GbRPR)*, pages 237–246, 2019. doi:10.1007/978-3-030-20081-7_23.
- 29 Bastien Cazaux and Eric Rivals. Linking BWT and XBW via Aho-Corasick automaton: Applications to run-length encoding. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 128 of *LIPICs*, pages 24:1–24:20, 2019. doi:10.4230/LIPICs.CPM.2019.24.
- 30 Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipt  k, and Giovanna Rosone. Computing the optimal BWT of very large string collections. In *Data Compression Conference (DCC)*, pages 71–80. IEEE, 2023. doi:10.1109/DCC55655.2023.00015.
- 31 Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipt  k, and Giovanna Rosone. Computing the optimal BWT and r -index of very large read collections. Submitted, 2025.
- 32 Davide Cenzato and Zsuzsanna Lipt  k. A theoretical and experimental analysis of BWT variants for string collections. In *33rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 223 of *LIPICs*, pages 25:1–25:18, 2022. doi:10.4230/LIPICs.CPM.2022.25.
- 33 Davide Cenzato and Zsuzsanna Lipt  k. A survey of BWT variants for string collections. *Bioinform.*, page bt  e333, 2024. doi:10.1093/bioinformatics/bt  e333.
- 34 Ananyo Choudhury, Shaun Aron, and Laura R. Botigu   et al. High-depth African genomes inform human migration and health. *Nature*, 586:741–748, 2020. doi:10.1038/s41586-020-2859-7.
- 35 Lapo Cioni, Veronica Guerrini, and Giovanna Rosone. The Burrows-Wheeler Transform of an Elastic-Degenerate String. In *25th Italian Conference on Theoretical Computer Science (ICTCS)*, volume 3811 of *CEUR Workshop Proceedings*, pages 66–80, 2024. URL: <https://ceur-ws.org/Vol-3811/paper240.pdf>.
- 36 Aleksander Cislak, Szymon Grabowski, and Jan Holub. SOPanG: online text searching over a pan-genome. *Bioinform.*, 34(24):4290–4292, 2018. doi:10.1093/BIOINFORMATICS/BTY506.
- 37 CMS-BWT. URL: <https://github.com/fmasillo/CMS-BWT>.
- 38 The COVID-19 Genomics UK (COG-UK) consortium. An integrated national scale SARS-CoV-2 genomic surveillance network. *The Lancet Microbe*, 1(3):e99–e100, 2020. doi:10.1016/S2666-5247(20)30054-9.
- 39 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. doi:10.1093/bioinformatics/bts173.
- 40 Kim Daehwan, Li Song, Florian P. Breitwieser, and Steven L. Salzberg. Centrifuge: rapid and sensitive classification of metagenomic sequences. *Genome Research*, 26(12):1721–1729, 2016. doi:10.1101/gr.210641.116.
- 41 Diego D  az-Dom  nguez and Gonzalo Navarro. Efficient construction of the extended BWT from grammar-compressed DNA sequencing reads. *CoRR*, abs/2102.03961, 2021. doi:10.48550/arXiv.2102.03961.
- 42 Diego D  az-Dom  nguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. *Inf. Comput.*, 294:105088, 2023. doi:10.1016/j.ic.2023.105088.
- 43 Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, and Miles J. Efron et al. Big data: Astronomical or genomics? *PLoS Biol*, 13(7):e1002195, 2015. doi:10.1371/journal.pbio.1002195.

- 44 Richard Durbin. Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinform.*, 30(9):1266–1272, 2014. doi:10.1093/BIOINFORMATICS/ BTU014.
- 45 eGAP. URL: <https://github.com/felipelouza/egap>.
- 46 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. doi:10.1186/S13015-019-0140-0.
- 47 eGSA. URL: <https://github.com/felipelouza/egsa>.
- 48 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005. doi:10.1145/1082036.1082039.
- 49 G2BWT. URL: https://bitbucket.org/DiegoDiazDominguez/lms_grammar/src/bwt_imp2.
- 50 Esteban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string comparison. *Inf. Comput.*, 304:105296, May 2025. doi:10.1016/j.ic.2025.105296.
- 51 Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 259 of *LIPICs*, pages 11:1–11:20, 2023. doi:10.4230/LIPICS.CPM.2023.11.
- 52 Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Pangenome Comparison via ED Strings. *Frontiers in Bioinformatics*, 4:11:1–11:20, 2024. doi:10.3389/fbinf.2024.1397036.
- 53 Travis Gagie, Garance Gourdel, and Giovanni Manzini. Compressing and indexing aligned readsets. In *21st International Conference on Algorithms in Bioinformatics (WABI)*, volume 201 of *LIPICs*, pages 13:1–13:21, 2021. doi:10.4230/LIPICS.WABI.2021.13.
- 54 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1), 2020. doi:10.1145/3375890.
- 55 Erik Garrison, Andrea Guarracino, Simon Heumos, Flavia Villani, Zhigui Bao, Lorenzo Tattini, Jörg Hagmann, Sebastian Vorbrugg, Santiago Marco-Sola, Christian Kubica, David G. Ashbrook, Kaisa Thorell, Rachel L. Rusholme-Pilcher, Gianni Liti, Emilio Rudbeck, Agnieszka A. Golicz, Sven Nahnsen, Zuyu Yang, Moses Njagi Mwaniki, Franklin L. Nobrega, Yi Wu, Hao Chen, Joep de Ligt, Peter H. Sudmant, Sanwen Huang, Detlef Weigel, Nicole Soranzo, Vincenza Colonna, Robert W. Williams, and Pjotr Prins. Building pangenome graphs. *Nature Methods*, 21:2008–2012, 2024. doi:10.1038/s41592-024-02430-3.
- 56 Genome 10K Community of Scientists. A proposal to obtain whole-genome sequence for 10,000 vertebrate species. *J. Hered.*, 100:659–674, 2009. doi:10.1093/jhered/esp086.
- 57 Ira M. Gessel and Christophe Reutenauer. Counting permutations with given cycle structure and descent set. *J. Comb. Theory A*, 64(2):189–215, 1993. doi:10.1016/0097-3165(93)90095-P.
- 58 Raffaele Giancarlo, Antonio Restivo, and Marinella Sciortino. From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. *Theor. Comput. Sci.*, 387:236–248, 2007. doi:10.1016/J.TCS.2007.07.019.
- 59 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 *47th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 12607 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2021. doi:10.1007/978-3-030-67731-2_18.
- 60 grlBWT. URL: <https://github.com/ddiazdom/grlBWT>.
- 61 gsufsort. URL: <https://github.com/felipelouza/gsufsort>.
- 62 Veronica Guerrini, Alessio Conte, Roberto Grossi, Gianni Liti, Giovanna Rosone, and Lorenzo Tattini. phyBWT2: phylogeny reconstruction via eBWT positional clustering. *Algorithms Mol. Biol.*, 18(1):11, 2023. doi:10.1186/S13015-023-00232-4.

- 63 Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone. Metagenomic analysis through the extended Burrows-Wheeler transform. *BMC Bioinform.*, 21-S(8):299, 2020. doi:10.1186/S12859-020-03628-W.
- 64 Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone. Parallel lossy compression for large FASTQ files. In *Biomedical Engineering Systems and Technologies*, pages 97–120, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-38854-5_6.
- 65 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CB09780511574931.
- 66 James Holt and Leonard McMillan. Merging of multi-string bwts with applications. *Bioinform.*, 30(24):3524–3531, 2014. doi:10.1093/bioinformatics/btu584.
- 67 Lilian Janin, Giovanna Rosone, and Anthony J. Cox. Adaptive reference-free compression of sequence quality scores. *Bioinform.*, 30(1):24–30, 2014. doi:10.1093/BIOINFORMATICS/BTT257.
- 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. doi:10.1089/CMB.2019.0309.
- 69 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012. doi:10.1038/nmeth.1923.
- 70 Ben Langmead, Michael C. Schatz, Jimmy Lin, Mihai Pop, and Steven L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10:R134, 2009. doi:10.1186/gb-2009-10-11-r134.
- 71 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. doi:10.1186/gb-2009-10-3-r25.
- 72 Brice Letcher, Martin Hunt, and Zamin Iqbal. Gramtools enables multiscale variation analysis with genome graphs. *Genome Biology*, 22, 2021. doi:10.1186/s13059-021-02474-0.
- 73 1FGSACA. URL: <https://gitlab.com/qwerzuiop/lfgsaca>.
- 74 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinform.*, 30(22):3274–3275, 2014. doi:10.1093/BIOINFORMATICS/BTU541.
- 75 Heng Li. BWT construction and search at the terabase scale. *Bioinform.*, 40(12):btae717, 2024. doi:10.1093/bioinformatics/btae717.
- 76 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinform.*, 26(5):589–595, 2010. doi:10.1093/bioinformatics/btp698.
- 77 Heng Li, Xiaowen Feng, and Chong Chu. The Design and Construction of Reference Pangenome Graphs with Minigraph. *Genome Biology*, 21(265), 2020. doi:10.1186/s13059-020-02168-z.
- 78 Ruiqiang Li et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20(2):265–272, 2010. doi:10.1101/gr.097261.109.
- 79 Wen-Wei Liao et al. A draft human pangenome reference. *Nature*, 617(7960):312–324, 2023.
- 80 Bo Liu, Dixian Zhu, and Yadong Wang. deBWT: parallel construction of Burrows-Wheeler Transform for large collection of genomes with de Bruijn-branch encoding. *Bioinform.*, 32(12):174–182, 2016. doi:10.1093/BIOINFORMATICS/BTW266.
- 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, 2020. doi:10.1186/s13015-020-00177-y.
- 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. doi:10.1186/s13015-017-0117-9.
- 83 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J of Comput.*, 12(1):40–66, 2005.
- 84 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.

- 85 Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows-Wheeler transform and Sturmian words. *Inf. Process. Lett.*, 86(5):241–246, 2003. doi:10.1016/S0020-0190(02)00512-4.
- 86 Francesco Masillo. Matching statistics speed up BWT construction. In *31st Annual European Symposium on Algorithms (ESA)*, volume 274 of *LIPIcs*, pages 83:1–83:15, 2023. doi:10.4230/LIPICS.ESA.2023.83.
- 87 Merge-BWT. URL: <https://github.com/jltsiren/bwt-merge>.
- 88 msbwt. URL: <https://github.com/holtjma/msbwt>.
- 89 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. doi:10.1089/CMB.2019.0316.
- 90 Njagi Moses Mwaniki, Erik Garrison, and Nadia Pisanti. Fast exact string to d-texts alignments. In *16th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSPEC)*, pages 70–79. SCITEPRESS, 2023. doi:10.5220/0011666900003414.
- 91 Njagi Moses Mwaniki and Nadia Pisanti. Optimal sequence alignment to ED-strings. In *18th International Symposium on Bioinformatics Research and Applications (ISBRA)*, volume 13760 of *Lecture Notes in Computer Science*, pages 204–216. Springer, 2022. doi:10.1007/978-3-031-23198-8_19.
- 92 Joong Chae Na, Hyunjoon Kim, Seunghwan Min, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment with gaps. *Theor. Comput. Sci.*, 710:148–157, 2018. doi:10.1016/j.tcs.2017.02.020.
- 93 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.
- 94 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011. doi:10.1109/TC.2010.188.
- 95 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013. URL: <http://www.oldenbusch-verlag.de/>.
- 96 Jannik Olbrich, Enno Ohlebusch, and Thomas Böhler. Generic non-recursive suffix array construction. *ACM Trans. Algorithms*, 20(2), 2024. doi:10.1145/3641854.
- 97 Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big BWTs. In *33rd Data Compression Conference (DCC)*, pages 62–70, 2023. doi:10.1109/DCC55655.2023.00014.
- 98 Marco Oliva, Massimiliano Rossi, Jouni Sirén, Giovanni Manzini, Tamer Kahveci, Travis Gagie, and Christina Boucher. Efficiently merging *r*-indexes. In *31st Data Compression Conference (DCC)*, pages 203–212. IEEE, 2021. doi:10.1109/DCC50243.2021.00028.
- 99 optimalBWT. URL: <https://github.com/davidecenzato/optimalBWT>.
- 100 Benedict Paten, Adam M. Novak, Jordan M. Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome Research*, 27(5):665–676, 2017. doi:10.1101/gr.214155.116.
- 101 PFP-eBWT. URL: <https://github.com/davidecenzato/PFP-eBWT>.
- 102 Nicola Prezza, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. Detecting Mutations by eBWT. In *18th International Conference on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPIcs*, pages 3:1–3:15, 2018. doi:10.4230/LIPICS.WABI.2018.3.
- 103 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. doi:10.1186/S13015-019-0137-8.
- 104 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. doi:10.1186/S12859-020-03586-3.
- 105 r-pfbwt. URL: <https://github.com/marco-oliva/r-pfbwt>.

- 106 Goran Rakocevic et al. Fast and accurate genomic analyses using genome graphs. *Nature Genetics*, 51:354–362, 2019. doi:10.1038/s41588-018-0316-4.
- 107 ropeBWT2. URL: <https://github.com/lh3/ropebwt2>.
- 108 ropeBWT3. URL: <https://github.com/lh3/ropebwt3>.
- 109 Ahsan Sanaullah, Seba Villalobos, Degui Zhi, and Shaojie Zhang. Haplotype matching with GBWT for pangenome graphs. *bioRxiv*, 2025. doi:10.1101/2025.02.03.634410.
- 110 Pramesh Shakya, Ardalan Naseri, Degui Zhi, and Shaojie Zhang. mcPBWT: Space-Efficient Multi-column PBWT Scanning Algorithm for Composite Haplotype Matching. In *11th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, volume 13254 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2021. doi:10.1007/978-3-031-17531-2_10.
- 111 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinform.*, 26(12):367–373, 2010. doi:10.1093/BIOINFORMATICS/BTQ217.
- 112 Jouni Sirén. Burrows-Wheeler Transform for terabases. In *26th Data Compression Conference (DCC)*, pages 211–220, 2016. doi:10.1109/DCC.2016.17.
- 113 Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. In *18th International Conference on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPICs*, pages 4:1–4:13, 2018. doi:10.4230/LIPICs.WABI.2018.4.
- 114 Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinform.*, 36(2):400–407, 2020. doi:10.1093/BIOINFORMATICS/BTZ575.
- 115 Chen Sun et al. RPAN: rice pan-genome browser for 3000 rice genomes. *Nucleic Acids Res*, 45(2):597–605, 2017. doi:10.1093/nar/gkw958.
- 116 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015. doi:10.1038/nature15393.
- 117 The 1001 Genomes Consortium. Epigenomic Diversity in a Global Collection of Arabidopsis thaliana Accessions. *Cell*, 166(2):492–505, 2016. doi:10.1016/j.cell.2016.06.044.
- 118 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings Bioinform.*, 19(1):118–135, 2018. doi:10.1093/bib/bbw089.
- 119 Cole Trapnell, Lior Pachter, and Steven L. Salzberg. TopHat: discovering splice junctions with RNA-Seq. *Bioinform.*, 25(9):1105–1111, 2009. doi:10.1093/BIOINFORMATICS/BTP120.
- 120 Clare Turnbull et al. The 100,000 genomes project: bringing whole genome sequencing to the NHS. *Br. Med. J.*, 361, 2018. doi:10.1136/bmj.k1687.

Wavelet Tree, Part II: Text Indexing

Paolo Ferragina ✉ 

Department L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy
Department of Computer Science, University of Pisa, Italy

Raffaele Giancarlo ✉ 

Department of Mathematics and Computer Science, University of Palermo, Italy

Roberto Grossi ✉ 

Department of Computer Science, University of Pisa, Italy

Giovanna Rosone ✉ 

Department of Computer Science, University of Pisa, Italy

Rossano Venturini ✉ 

Department of Computer Science, University of Pisa, Italy

Jeffrey Scott Vitter ✉ 

Department of Computer Science, Tulane University, New Orleans, LA, USA
Department of Computer and Information Science, The University of Mississippi, MS, USA

Abstract

The *Wavelet Tree* data structure introduced in Grossi, Gupta, and Vitter [11] is a space-efficient technique for rank and select queries that generalizes from binary symbols to an arbitrary multisymbol alphabet. Over the last two decades, it has become a pivotal tool in modern full-text indexing and data compression because of its properties and capabilities in compressing and indexing data, with many applications to information retrieval, genome analysis, data mining, and web search. In this paper, we survey the fascinating history and impact of Wavelet Trees; no doubt many more developments are yet to come. Our survey borrows some content from the authors' earlier works.

This paper is divided into two parts: The first part gives a brief history of Wavelet Trees, including its varieties and practical implementations, which appears in the Festschrift dedicated to Roberto Grossi [4]; the second part (this one) deals with Wavelet Tree-based text indexing and is included in the Festschrift dedicated to Giovanni Manzini.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Data compression

Keywords and phrases Wavelet tree, data compression, text indexing, compressed suffix array, Burrows-Wheeler transform, rank and select

Digital Object Identifier 10.4230/OASICS.Manzini.2025.4

Acknowledgements The authors would like to thank Hongwei Huo for helpful comments and figures.

1 Introduction

The field of compressed full-text indexing [24] involves the design of data structures (aka, indexes) that support fast substring matching using small amounts of space. For a text string $\mathcal{T}[1, n]$ over an arbitrary alphabet Σ of size σ and a pattern $\mathcal{P}[1, m]$, the goal of text indexing is to preprocess $\mathcal{T}$ using succinct space so that queries like the following can be quickly answered: (1) count the number *occ* of occurrences of $\mathcal{P}$ in $\mathcal{T}$; (2) locate the *occ* positions in $\mathcal{T}$ where $\mathcal{P}$ occurs; and (3) starting at text position *start*, extract the length- ℓ substring $\mathcal{T}[\text{start}, \text{start} + \ell - 1]$.

A main goal is to create an index whose size is roughly equal to the size of the text in compressed format, with search performance comparable to the well-known indexes on uncompressed text, such as suffix trees and suffix arrays. Some compressed data structures



© Paolo Ferragina, Raffaele Giancarlo, Roberto Grossi, Giovanna Rosone, Rossano Venturini, and Jeffrey Scott Vitter;
licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 4; pp. 4:1–4:10



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

are in addition *self-indexes* in that the data structure encapsulates the original text and, thus, can quickly recreate any portion of it. As a result, the original text is not needed, it can be discarded and replaced by only the (self-)index.

Most compressed indexing techniques developed in the last two decades make use of the powerful *Wavelet Tree* data structure, developed by Grossi, Gupta, and Vitter [11], with many applications to information retrieval, genome analysis, data mining, and web search. The Wavelet Tree supports fast rank and select queries on any string from an arbitrary alphabet. As such, it provides a space-efficient extension of the space-efficient rank-select data structures for binary sequences [26, 24] to the case of general alphabets of arbitrary size.

A Wavelet Tree represents any string from a general multisymbol alphabet as a hierarchical set of binary strings. It has the following key property for purposes of compression: If each such binary string is encoded according to its information-theoretic minimum size, then the original string is compressed to its information-theoretic minimum size.

In this paper we discuss how Wavelet Trees can be used to construct an efficient self-index for text $\mathcal{T}$ using space related to the higher-order entropy of $\mathcal{T}$. The two main data structures to which Wavelet Trees are applied for that purpose are the compressed suffix array (CSA) [12, 28, 11] and the FM-index [6, 7].

2 Preliminaries

We refer the reader to PART I, which is included in the Festschrift dedicated to Roberto Grossi [4], for the notation used in this paper, the history and background of the ubiquitous Wavelet Tree, and the key definitions of 0th order entropy and higher-order entropy.

The suffix array SA gives the positions of all the suffixes of $\mathcal{T}$ in lexicographical order. We use $SA[0, n-1]$ to denote the suffix array (SA) of $\mathcal{T}$. We use $SA^{-1}[0, n-1]$ to denote the inverse suffix array, where SA^{-1} is a permutation of suffixes of $\mathcal{T}$ in lexicographical order, such that $SA^{-1}[SA[i]] = i$. Table 1 provides a running example.

The key notion of *neighbor function* Ψ [12, 28] is illustrated in Table 1 and formally defined as follows: Given a suffix s of $\mathcal{T}$ and its index i in the suffix array (i.e., $s = \langle \mathcal{T}[SA[i]], \mathcal{T}[SA[i]+1 \bmod n], \dots \rangle$), let s' be the suffix (a.k.a. neighbour) formed by removing the first symbol of s . The value of the neighbor function $\Psi[i]$ is the index i' of s' in the suffix array. More formally,

$$\Psi(i) = SA^{-1}[(SA[i] + 1) \bmod n].$$

Let us designate by $\mathcal{T}_x$ (as in PART I) the substring of $\mathcal{T}$ associated with an individual context x of k symbols. Wavelet Trees are a natural and elegant way to compress each substring $\mathcal{T}_x$ so that the code length per symbol is the 0th-order entropy of $\mathcal{T}_x$. It thus follows by Definition 3 in PART I of k th-order entropy that the cumulative encoding of all the substrings $\mathcal{T}_x$ achieves k th-order entropy of the full string $\mathcal{T}$. We elaborate on this key idea in Section 4.

3 Text Indexing

The two main paradigms used for modern text indexing are compressed suffix arrays (CSA) [12, 28, 11] and FM-index [6, 7]. We use the notation $\mathcal{T} = \mathcal{T}[0, n-1]$ to denote a text of n symbols drawn from an alphabet Σ of size σ . Both the CSA and FM-Index make use of the suffix array concept.

■ **Table 1** For text $\mathcal{T} = \text{tcaaaatatatgcaacatatagtagattgtat\#}$ shown in the first column, the subsequent columns show the suffix array SA , the inverse suffix array, the neighbor function Ψ , and the LF function of the BWT. The second-to-last column shows the suffixes in sorted order; each suffix starts at the symbol under F (first) and ends at the symbol L (last). The string of symbols under L is the BWT of $\mathcal{T}$. The last column $\mathcal{B}$ is the bit array for the root node of the Wavelet Tree of L ; symbols $\#$, a , and c (for the left subtree) are designated by a 0, and symbols g and t (for the right subtree) are designated by a 1. The Wavelet Tree of L is pictured in Figure 1.

i	$\mathcal{T}$	SA	SA^{-1}	Ψ	LF	F	L	$\mathcal{B}$
0	t	35	31	31	23	# t c a ...a t		1
1	c	2	16	2	16	a a a a ...t c		0
2	a	3	1	4	1	a a a t ...c a		0
3	a	13	2	5	17	a a c a ...g c		0
4	a	4	4	11	2	a a t a ...a a		0
5	a	14	11	18	3	a c a t ...c a		0
6	t	26	28	19	24	a g a t ...t t		1
7	a	20	12	22	25	a g t a ...a t		1
8	t	33	29	23	26	a t # t ...g t		1
9	a	18	13	25	27	a t a g ...a t		1
10	t	16	32	27	18	a t a t ...a c		0
11	g	5	20	28	4	a t a t ...a a		0
12	c	7	17	29	28	a t a t ...a t		1
13	a	9	3	32	29	a t g c ...a t		1
14	a	23	5	34	30	a t t a ...g t		1
15	c	28	18	35	19	a t t g ...a g		1
16	a	1	10	1	31	c a a a ...# t		1
17	t	12	27	3	20	c a a c ...t g		1
18	a	15	9	10	5	c a t a ...a a		0
19	t	27	25	15	6	g a t t ...t a		0
20	a	11	7	17	32	g c a a ...a t		1
21	g	31	22	26	33	g t a t ...t t		1
22	t	21	30	30	7	g t a t ...t a		0
23	a	34	14	0	8	t # t c ...t a		0
24	t	25	34	6	34	t a g a ...a t		1
25	t	19	24	7	9	t a g t ...t a		0
26	a	32	6	8	21	t a t # ...t g		1
27	g	17	19	9	10	t a t a ...c a		0
28	a	6	15	12	11	t a t a ...a a		0
29	t	8	35	13	12	t a t g ...t a		0
30	t	22	33	14	22	t a t t ...a g		1
31	g	0	21	16	0	t c a a ...t #		0
32	t	10	26	20	13	t g c a ...t a		0
33	a	30	8	21	35	t g t a ...a t		1
34	t	24	23	24	14	t t a g ...t a		0
35	#	29	0	33	15	t t g t ...g a		0

3.1 Compressed Suffix Array (CSA)

The CSA represents the text string $\mathcal{T}$ by encoding the *neighbor function* Ψ [12, 28]. The sequence of Ψ values suffices to recreate the original text $\mathcal{T}$, using some supporting data structures. For example, if we know the symbol that appears at the beginning of the i th suffix in lexicographic order, we can easily determine the neighboring symbol in the text (at position $SA[i] + 1$ in the text) by computing $j = \Psi(i)$. For each $\omega \in \Sigma$, if we keep the count of the number of symbols in $\mathcal{T}$ smaller than or equal to ω in lexicographic order, then the

desired symbol is simply the first symbol in lexicographic order whose count is $\geq j$. We can then continue taking steps forward from one neighbor to the next to recover additional symbols.

To encode Ψ , we conceptually partition Ψ into context lists according to each symbol $\omega \in \Sigma$. The context ω of $\Psi(i)$ is defined as the symbol $\mathcal{T}[SA[i]]$ that begins the suffix starting in position $SA[i]$. We denote a j -context as a length- j string. Suppose we have two suffixes at indices i and i' (where $i < i'$) in the suffix array that both start with the same symbol (context) ω . Then if we remove the first symbol ω from each suffix, the two suffixes that remain must maintain the same relative ordering in the suffix array:

► **Property 1.** *The neighbor function Ψ is a piecewise increasing sequence.*

We show the neighbor function Ψ for text $\mathcal{T} = \text{tcaaaatatatgcaacatatagattagattgtat\#}$ (see Table 1). The suffixes starting with the symbol **a** span indices $[1, 15]$ of the suffix array, and their Ψ values form an increasing sequence $\langle 2, 4, 5, 11, 18, 19, 22, 23, 25, 27, 28, 29, 32, 34, 35 \rangle$, which we call **a**'s context list. For a given k (to be specified later), we further partition each ω 's context list into sublists $\langle \omega, x \rangle$ according to k -context $x \in \Sigma^k$. Here k -context x of Ψ denotes the k -symbol prefix of the suffix starting at position $SA[\Psi(i)]$ and its preceding symbol is $SA[i]$. For the text $\mathcal{T}$ shown in Table 1, the symbol $SA[0] = \#$ precedes the 3-context **tca** of the suffix starting at $SA[\Psi(0)] = SA[31]$. As another example, the 2-context **at** associated with indices $[8, 15]$ further decomposes **a**'s context list into the sublist $\langle 23, 25, 27, 28, 29, 32, 34, 35 \rangle$. The entries in each sublist also form an increasing sequence.

The next property is key for efficient coding of the neighbor function Ψ :

► **Property 2.** *For any k -context $x \in \Sigma^k$, all the entries in sublists $\langle \omega, x \rangle$ for all nonempty contexts $\omega \in \Sigma$ form an index range of contiguous values in the suffix array SA . Moreover, all k -contexts x' that precede x in lexicographic order create an index range immediately prior to the index range of x in the suffix array.*

For coding purposes, this property allows us to normalize each entry $\Psi(i)$ in each sublist $\langle \omega, x \rangle$ from 1 to $total(x)$, where $total(x)$ is the number of suffixes whose first $k+1$ symbols are of the form yx for any $y \in \Sigma$. For example in Table 1, for $x = \text{t}$, the union of the sublists $\langle \omega, x \rangle$ for all ω is the contiguous range $[23, 35]$ of 13 values.

In Section 4 we will see how the ability of Wavelet Trees to encode individual sublists with 0th-order entropy results in an cumulative higher-order compression for the full text $\mathcal{T}$.

3.2 FM-Index

The FM-index involves accessing the Burrows-Wheeler transform (BWT [1]), which appears as the L column in the example in Table 1. Adjacent entries in the L column tend to share similar statistics regarding which symbols follow because the symbols that follow them in the text are adjacent in lexicographic order. As an example, $L[8, 15]$ is the substring of L formed by all symbols of $\mathcal{T}$ that precede the 2-context **at**, as you can observe in Table 1.

The close relation between the CSA and BWT follows from the fact that the neighbor function Ψ is the inverse of the so-called LF function of the BWT (which maps the index of a symbol in the L column to the index where it appears when rotated forward in the F column). In other words, $\Psi(i) = j$ iff $LF(j) = i$, as can be seen from Table 1.

Building LF is straightforward for symbols that occur only once, as it is the case of $\#$ in our running example of Table 1. But computing LF efficiently is no longer trivial regarding recurring symbols. Nonetheless, it can be solved in optimal $\mathcal{O}(n)$ time thanks to a clever algorithm that uses an auxiliary vector C of size σ and is shown in Algorithm 1. For the

Algorithm 1 Constructing the LF-mapping from column L .

```

1: for  $i = 0, 1, \dots, n - 1$  do
2:    $C[L[i]]++$ ;
3:  $temp = 0, sum = 0$ ;
4: for  $i = 0, 1, \dots, \sigma$  do
5:    $temp = C[i]$ ;
6:    $C[i] = sum$ ;
7:    $sum += temp$ ;
8: for  $i = 0, 1, \dots, n - 1$  do
9:    $LF[i] = C[L[i]]$ ;
10:   $C[L[i]]++$ ;
  
```

sake of description, we assume that array C is indexed by a *symbol* rather than by an integer. The first for-loop in Algorithm 1 computes, for each symbol c , the number n_c of its occurrences in L , and thus it sets $C[c] = n_c$. Then, the second for-loop turns these symbol-wise occurrences into a cumulative sum, namely $C[c] = \sum_{x < c} n_x$. We notice that $C[c]$ gives the first position in F where the symbol c occurs. Therefore, before the last for-loop starts, $C[c]$ is the landing position in F of the first c in L (we thus know the LF -mapping for the first occurrence of every alphabet symbol). Finally, the last for-loop scans the column L , and whenever it encounters the symbol $L[i] = c$, it sets $LF[i] = C[c]$ and then increments $C[c]$. So the algorithm maintains the invariant that $LF[i] = \sum_{x < c} n_x + k$, after that k occurrences of c in L have been processed. The time complexity of such computation is $\mathcal{O}(n)$.

Like the neighbor function Ψ in the CSA, the BWT transform suffices to recreate the original text $\mathcal{T}$. To do so, it uses a backward scan supported by the LF -mapping in $\mathcal{O}(n)$ time and space. The algorithm starts from the last symbol of $\mathcal{T}$, which occurs at $L[0]$; and then it proceeds by moving one symbol at a time to the left in $\mathcal{T}$, deploying the properties of the LF -mapping: if we know the symbol in the F column at index i , then the preceding symbol in the text is the symbol in the L column on the same row i . Given this observation, the backward reconstruction of $\mathcal{T}$ first maps the current symbol occurring in L , say $L[i]$, to its corresponding copy in F , hence $j = LF[i]$; and then takes the symbol $L[j]$ that is ensured by the properties of the cyclic rotations of the rows of the BWT to precede $F[j] = L[i]$ in $\mathcal{T}$ (see above). This double step, which returns the algorithmic focus on L , allows to move one symbol leftward in $\mathcal{T}$. Hence, repeating this for n steps, we can reconstruct the original input string $\mathcal{T}$ in $\mathcal{O}(n)$ time. As far as the construction of the BWT is concerned, we mention that it can be done by building the suffix array data structure via a plethora of time and space-efficient algorithms that now abound in the literature (see e.g. [17, 16, 22]) or directly via several elegant approaches (see e.g. [14, 25, 18, 23]).

But how useful is it to permute $\mathcal{T}$ via the BW-Transform? If we read the first column F of Table 1, we notice a sequence of possibly long runs of equal symbols. It is clear that by compressing that sequence with a simple RLE-compressor (namely, the one that substitutes each run of equal symbols x^h with a pair $\langle x, h \rangle$) we would get a high compression ratio. But F does not allow us to return to the original string $\mathcal{T}$ because all strings consisting of 15 **a**, 3 **c**, 4 **g**, and 13 **t** symbols would obtain the same F . It can be proved [1] that the only column allowing to return to $\mathcal{T}$, for all possible $\mathcal{T}$, is properly L . That column could be highly compressible because it satisfies the so-called locally homogeneous property, namely, that substrings of L consist of a few distinct symbols. The reason comes from the fact that if we take a string (context) of length k (as in the Definition 3 in PART I of k th-order entropy), say $x \in \Sigma^k$, the n_x symbols $\mathcal{T}_x$ immediately preceding each occurrence of x in $\mathcal{T}$ occur

contiguously in L . The nice issue here is that many real sources (they are called Markovian) do exist that generate data sequences, other than texts, that can be turned to be locally homogeneous via the Burrows-Wheeler Transform and thus can be highly compressed by simpler compressors.

This basic principle is deployed by the design of the well-known **bzip** tool (now¹ arrived to version 3), whose algorithmic pipeline we briefly sketch: Let us for a moment focus on two simple algorithms, namely, the *Move-To-Front* (shortly, MTF) and the *Run-Length Encoding* (shortly RLE, mentioned above). The former maps symbols into integers, and the latter maps runs of equal symbols into pairs. We observe that RLE is a compressor indeed, because the output sequence may be reduced in length in the presence of long runs of equal symbols; while MTF is a transform that can be turned into a compressor by encoding the integers via proper variable-length encoders. In general, the compression performance of those algorithms is very poor; while BWT turns them magically to be very effective if combined together: first apply MTF on L , then input the result on RLE, and finally encode its output with a statistical coder like Huffman or arithmetic coding. That's it².

Manzini [21] showed that this, and other pipelines based upon the BWT, can represent a text string $\mathcal{T}$ of n symbols using space proportional to n times its k th-order empirical entropy $\mathcal{H}_k(\mathcal{T})$, with guarantees stronger than the ones ensured by Lempel-Ziv-based compressors (such as gzip). Later analysis using Wavelet Trees reduced the constant factor in front of the $n\mathcal{H}_k(\mathcal{T})$ term to 1 [11, 8]. Manzini's result generated increased interest in the BWT transform, whose impact on data compression, and not just compressed text indexing, was highlighted by [5] as a sort of *compression booster*.

4 Compression and Boosting to Higher-Order Entropy

The original motivation for Wavelet Trees was to extend the RRR method [26] from binary sequences to sequences composed of symbols from general alphabets in order to achieve 0th-order entropy. That capability is important because if a text $\mathcal{T}$'s Ψ neighbor function values [12, 11, 27] (in the case of compressed suffix arrays) and the BWT transform (L column) [7, 24] (in the case of the FM-index) are decomposed into substrings based upon $\mathcal{T}$'s contexts of length k , then we can use a Wavelet Tree to compress each of those substrings to its 0th-order entropy (plus lower-order terms). The key point is that the resulting cumulative encoding will achieve a compression equal to the k th-order entropy of $\mathcal{T}$ (plus lower-order terms).

This notion of achieving a high-order entropy based upon individual encodings of 0th-order entropy (later called “boosting” in [3, 5]) is inherent in Definition 3 in PART I of k th-order entropy, in which $n\mathcal{H}_k(\mathcal{T})$ is expressed as a sum over all k -contexts x of the terms $n_x\mathcal{H}_0(\mathcal{T}_x)$. Grossi, Gupta, and Vitter used Wavelet Trees to provide the first demonstration of the asymptotically optimal space bound $\sim n\mathcal{H}_k(\mathcal{T})$ (i.e., with leading coefficient 1) for compressed suffix arrays and for the Burrows-Wheeler transform [11].

In order to better illustrate this powerful idea, let us consider a generic 0-order statistical compressor $\mathcal{C}_0$ whose performance, in bits per symbol, over a string s is bounded by $\mathcal{H}_0(s) + f(|s|)$. We notice that the function $f(|s|) = 2/(|s|)$ is the one achieved by arithmetic coding and it is $f(|s|) = 1$ for Huffman coding. In order to turn $\mathcal{C}_0$ into an effective k th-order compressor $\mathcal{C}_k$, for a fixed $k > 0$, we can compute the Burrows-Wheeler Transform of the

¹ See <https://en.wikipedia.org/wiki/Bzip2> and <https://github.com/kspalaiologos/bzip3>

² For a scholarly discussion of the BWT and the FM-index, the reader is directed to the book [2].

input string $\mathcal{T}$, take all possible length- k substrings x of $\mathcal{T}$, and then partition the column L into the substrings $\mathcal{T}_x$ (each one formed by the last symbols of the rows prefixed by x). Therefore, the final step is just to compress each $\mathcal{T}_x$ by $\mathcal{C}_0$, and concatenate the output bit sequences by alphabetically increasing x .

As far as the compression performance in bits per symbol is concerned, we easily derive that it can be bounded as $(1/n) \sum_{x \in \Sigma^k} |\mathcal{T}_x| (\mathcal{H}_0(\mathcal{T}_x) + f(|\mathcal{T}_x|)) = \mathcal{H}_k(\mathcal{T}) + \sum_{x \in \Sigma^k} |\mathcal{T}_x| (f(|\mathcal{T}_x|)/n)$ bits, where we have applied Definition 3 in PART I of $\mathcal{H}_k(\mathcal{T})$ to the summation of the $\mathcal{H}_0(\mathcal{T}_x)$. Now, if $\mathcal{C}_0$ is the arithmetic coder, the previous bound is $\mathcal{H}_k(\mathcal{T}) + \mathcal{O}(\sigma^k/n)$ bits, since $f(|s|) = 2/|s|$. In [5] the authors showed other stronger upper bounds to the performance of $\mathcal{C}_k$ and, more importantly, that one actually does not need to fix k in advance, since there is a so-called Compression Booster that identifies in optimal linear time a partition of L that achieves a compression ratio that is at least as good as the one obtained by $\mathcal{C}_k$, up to an additive lower-order term for any possible k in the allowable range. The algorithm is a surprisingly simple and elegant greedy-based algorithm; we refer the interested reader to that paper.

4.1 Using a Single Wavelet Tree with Implicit Partitions of Context

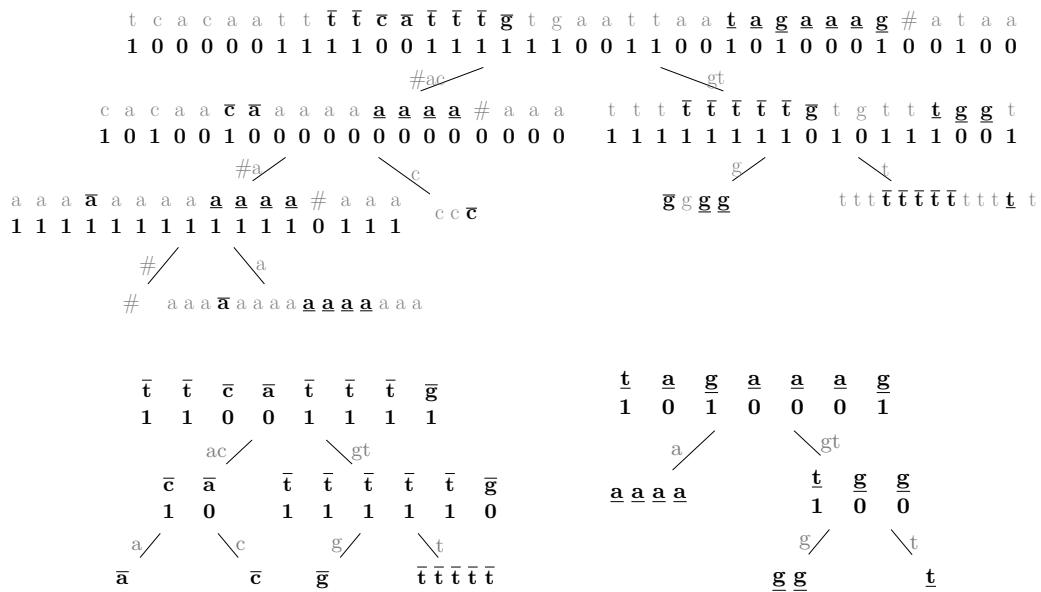
In the previous section, we decomposed the string being compressed (such as the neighbour sequence Ψ of the CSA or the L column in the FM Index) into contexts. Each individual context $\mathcal{T}_x$ was then encoded to its 0th-order entropy, and by Definition 3 in PART I of k th-order entropy, the sum of the resulting individual $n_x \mathcal{H}_0(\mathcal{T}_x)$ terms resulted in a total of $n \mathcal{H}_k(\mathcal{T})$ bits (the boosting effect). In some cases, the results hold for all k in the allowed range $k < \alpha \log_\sigma n$ for some constant $0 < \alpha < 1$.

Foschini et al. [9] showed how order k th-order entropy can be achieved by using a single Wavelet Tree without explicit partitioning into k -contexts. Intuitively, there are two reasons:

- One reason is that the individual Wavelet Trees that would be constructed for each context can be found within the single Wavelet Tree constructed on the entire BWT string L , as illustrated in Figure 1.
- The other reason follows from using a method like run-length encoding (RLE) or gap encoding (Gap) (in conjunction with a prefix coder such as the γ or δ code to encode the lengths) in order to encode the bit arrays of the nodes of the single Wavelet Tree. On the plus side, there is thus no need for data structures to keep track of the statistics of many individual contexts. On the negative side, the coding method RLE or Gap does not know the k -context boundaries. However, Foschini et al. [9] showed that RLE can adapt to the statistics for each k -context it encounters, and the positives make up for the negatives. By superimposing hypothetical k -context boundaries on the string for purposes of analysis, the resulting RLE implicitly encodes each k -context in roughly twice 0th-order space, and by Definition 3 in PART I of k th-order entropy, the resulting overall encoding achieves $2n \mathcal{H}_k(\mathcal{T}_k)$ bits leading space term. In effect, it achieves implicit boosting.

In implicit boosting, there is no need to specify k in advance; the compression results hold for all k in the allowed range $k < \alpha \log_\sigma n$, $0 < \alpha < 1$. Foschini et al. [9] used RLE encoding with γ encoding, but other coding methods could be used, possibly switching dynamically among several methods (along with the bits that indicate the choices made) [3, 13, 15].

Gog et al. [10] compress the BWT by using the RRR method [26] to encode the Wavelet Trees of fixed-size blocks of the BWT rather than using partitions based upon higher-order contexts; they showed that the resulting compression achieves the $n \mathcal{H}_k(\mathcal{T})$ leading space term. Mäkinen and Navarro [19, 20] showed the surprising result that RRR [26] applied



■ **Figure 1** The top row shows a single Wavelet Tree for the BWT L of $\mathcal{T}$ as in Table 1, and the bottom row shows separate Wavelet Trees for each of the context blocks ttcatttg, highlighted as overlined letters, and tagaag, highlighted as underlined letters, of L corresponding to 2-contexts **at** and **ta**, respectively.

to a single Wavelet Tree of the entire BWT sequence or Ψ sequence (i.e., without any partitioning) also achieves the $n \mathcal{H}_k(\mathcal{T}_k)$ bits leading space term, for a similar reason as in [9] described above; careful analysis in [19, 20] showed that the subblocks formed by [26] within each Wavelet Tree node implicitly encode each k th-order context in 0th-order entropy space plus lower-order terms. Grossi, Vitter, and Xu [13] present interesting experiments on the practical performance of these approaches.

References

- 1 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm, 1994.
- 2 Paolo Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.
- 3 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of Wavelet Trees. *Information and Computation*, 207(8):849–866, 2009. doi:10.1016/j.ic.2008.12.010.
- 4 Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, Giovanna Rosone, Rossano Venturini, and Jeffrey Scott Vitter. Wavelet Tree, Part I: A Brief History. appearing in the Festschrift dedicated to Roberto Grossi.
- 5 Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Marinella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, July 2005. doi:10.1145/1082036.1082043.
- 6 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS’00)*, pages 390–398, Washington, DC, November 2000. IEEE Computer Society. doi:10.1109/SFCS.2000.892127.
- 7 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
- 8 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):Article 20, 2007. An earlier version appeared in *Proceedings of the 11th Symposium on String Processing and Information Retrieval (SPIRE)*, 150–160, October 2004.

- 9 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. This work combined earlier versions from *SIAM Symposium on Discrete Algorithms (SODA)*, January 2004, and “Fast compression with a static model in high-order entropy,” *Data Compression Conference (DCC)*, March 2004. doi:10.1145/1198513.1198521.
- 10 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in FM-Indexes: Theory and practice. *Algorithmica*, 81:1370–1391, 2019. doi:10.1007/S00453-018-0475-9.
- 11 Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA’03)*, pages 841–850, New York, NY, January 2003. Association for Computing Machinery.
- 12 Roberto Grossi and Jeffrey S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. An earlier version appeared in *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC’00)*, May 2000, 397–406. doi:10.1137/S0097539702402354.
- 13 Roberto Grossi, Jeffrey S. Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *2011 First International Conference on Data Compression, Communications and Processing*, pages 210–221, June 2011. doi:10.1109/CCP.2011.16.
- 14 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.*, 38(6):2162–2178, 2009. doi:10.1137/070685373.
- 15 Hongwei Huo, Zongtao He, Pengfei Liu, and Jeffrey S. Vitter. FM-Adaptive: A practical data-aware FM-index, 2025. submitted.
- 16 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.
- 17 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2):126–142, 2005. doi:10.1016/J.JDA.2004.08.019.
- 18 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 Lenore J. Cowen, editor, *Research in Computational Molecular Biology - 23rd Annual International Conference, RECOMB 2019, Washington, DC, USA, May 5-8, 2019, Proceedings*, volume 11467 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2019. doi:10.1007/978-3-030-17083-7_10.
- 19 Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE’07)*, pages 229–241, Heidelberg, Germany, 2007. Springer-Verlag Berlin. doi:10.1007/978-3-540-75530-2_21.
- 20 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008.
- 21 Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001. An earlier version appeared in *Proceedings of the 10th Symposium on Discrete Algorithms (SODA ’99)*, January 1999, 669–677. doi:10.1145/382780.382782.
- 22 Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 698–710. Springer, 2002. doi:10.1007/3-540-45749-6_61.
- 23 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Fast compressed self-indexes with deterministic linear-time construction. *Algorithmica*, 82(2):316–337, 2020. A conference version of this paper appeared in *Proc. ISAAC 2017*. doi:10.1007/S00453-019-00637-X.

- 24 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.
- 25 Daisuke Okanohara and Kunihiro Sadakane. A linear-time burrows-wheeler transform using induced sorting. In Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö, editors, *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009, Proceedings*, volume 5721 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2009. doi:10.1007/978-3-642-03784-9_9.
- 26 Rajeev Raman, Venkatesh Raman, and S.Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 2007. doi:10.1145/1290672.1290680.
- 27 Kunihiro Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.
- 28 Kunihiro Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of the 11th Symposium on Algorithms and Computation (ISAAC'00)*, pages 410–421, Banff Alberta, Canada, August 2000. Springer-Verlag Berlin, Heidelberg.

FM-Adaptive: A Practical Data-Aware FM-Index

Hongwei Huo¹ 

Department of Computer Science, Xidian University, Xi'an, China

Zongtao He 

Department of Computer Science, Xidian University, Xi'an, China

Pengfei Liu 

Department of Computer Science, Xidian University, Xi'an, China

Jeffrey Scott Vitter¹ 

Department of Computer Science, Tulane University, New Orleans, LA, USA
the University of Mississippi, MS, USA

Abstract

The FM-index provides an important solution for efficient retrieval and search in textual big data. Its variants have been widely used in many fields including information retrieval, genome analysis, and web searching. In this paper, we propose improvements via a new compressed representation of the wavelet tree of the Burrows-Wheeler transform of the input text, which incorporates the gap γ -encoding. Our theoretical analysis shows that the new index, called **FM-Adaptive**, achieves asymptotic space optimality within a factor of 2 in the leading term, but it has a better compression and faster retrieval in practice than the competitive optimal compression boosting used in previous FM-indexes. We present a practical improved locate algorithm that provides substantially faster locating time based upon memoization, which takes advantage of the overlapping subproblems property. We design the lookup table for accelerated decoding to support fast pattern matching in a text. Extensive experiments demonstrate that FM-Adaptive provides faster query performance, often by a considerable amount, and/or comparable or better compression than other state-of-the-art FM-index methods.

2012 ACM Subject Classification Information systems → Information retrieval; Theory of computation → Design and analysis of algorithms; Theory of computation → Data structures design and analysis; Theory of computation → Data compression; Theory of computation → Pattern matching

Keywords and phrases Text indexing, Burrows-Wheeler transform, Compressed wavelet trees, Entropy-compressed, Compressed data structures

Digital Object Identifier 10.4230/OASICS.Manzini.2025.5

Supplementary Material *Software (Source Code)*: <https://doi.org/10.24433/CO.7967727.v1> [30]

Funding This work was supported in part by the National Natural Science Foundation of China under Grant No. 62272358.

Acknowledgements We would like to thank Simon Gog for sharing code.

1 Introduction

Massive data sets are being produced at unprecedented rates from sources like IoT, ultra-high-throughput next-generation sequencing, autonomous driving, digital universe, and social networks. A large part of the data consists of text in the form of a sequence of symbols representing not only natural language, but also multimedia streams, biological sequences, and myriad forms of other media. A full-text index is a data structure that stores a text string

¹ corresponding author



© Hongwei Huo, Zongtao He, Pengfei Liu, and Jeffrey Scott Vitter;
licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday.
Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 5; pp. 5:1–5:23



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in preprocessed form so that it can support fast string matching queries. The best-known full-text indexes are the suffix tree [45, 54], and the suffix array [43], which support pattern matching queries in optimal or almost-optimal time. However, for a text $\mathcal{T} = \mathcal{T}[0, n-1]$ consisting of n symbols drawn from an alphabet Σ of size σ , these data structures require $\Omega(n \log n)$ bits in the standard unit cost RAM, which is larger than $n \log \sigma$ bits, which is the size of the input text, by a multiplicative factor of $\Omega(\log_\sigma n)$.² For example, using suffix trees and suffix arrays, the full-text indexing requires approximately 36GB of memory in the most optimized implementations for short read mapping for a mammalian genome [4]. Thus indexing a text in space-efficient way while supporting efficient pattern matching queries is a challenging problem.

The field of compressed or succinct data structures attempts to build data structures whose space is provably close to the size of the data in compressed format and that still provide fast query functionality. Theoretical breakthroughs in the late 1990s led to the development of a new generation of space-efficient indexes. In particular, the compressed suffix array (CSA) [22, 52, 23, 53] and the FM-index [10, 21, 11] (based upon the Burrows-Wheeler transform (BWT) [3, 44]) provide the fundamentals of how to work with text efficiently in compressed format. Much subsequent work has focused on making compressed indexes fast and space-efficient in practice in order to handle a variety of big data applications, such as sequencing data consisting of billions of short reads.

1.1 Related Work

The FM-index [10, 11, 13, 12, 15, 24, 18, 29, 31] and the compressed suffix array (CSA) [22, 21, 23, 52, 53, 15, 42, 32, 19, 17] are space-efficient text indexes whose query times are proportional to the query pattern size plus the product of the output size and a small polylog function of n . The former maintains a succinct representation of the BWT and the latter maintains a succinct representation of the neighbor function Ψ [22, 23]. Both are self-indexes in that they represent the original text, and thus it can be discarded.

Grossi and Vitter [22, 23] and Sadakane [52] introduced the compressed suffix array and the neighbor function Ψ . Ferragina and Manzini [10, 11] designed the original entropy-compressed FM-index based upon the Burrows-Wheeler transform (BWT) [3, 44] and the mapping function LF . The mapping function LF and the neighbor function Ψ are inverses of one other.

Grossi, Gupta and Vitter [21] introduced the elegant data structure known as the wavelet tree, which has since become ubiquitous in text indexing. Using the wavelet tree, they were the first to establish a CSA self-index that provably achieves the asymptotically optimal space bound with leading coefficient 1 (i.e., $\sim \mathcal{H}_k(\mathcal{T})$ bits, the k th order entropy of $\mathcal{T}$, as defined subsequently in Definition 2). Their analysis of the wavelet tree also applied to the FM-index and was the first to show asymptotic optimality as well for the FM-index [21, 12, 13, 40, 41]. The original space bound derived for the BWT had a leading term of $5n \mathcal{H}_k(\mathcal{T})$ [10], which was improved to the asymptotically optimal $n \mathcal{H}_k(\mathcal{T})$ [21, 12] using wavelet trees.

Subsequently to the early theoretical breakthroughs [22, 10, 21, 11, 23], there have been numerous improvements. Simpler implementations for the CSA and the FM-index achieve high-order compression without explicit partitioning into separate k -contexts and thus using a single wavelet tree for the entire text $\mathcal{T}$ [15, 40, 41, 14]. (Here “ k -context” denotes a length- k prefix in $\mathcal{T}$ of a symbol $\mathcal{T}[SA[i]]$, for some $k \geq 0$, where SA is the suffix array of $\mathcal{T}$.)

² All logarithms in this paper that do not have an explicit base listed are in base 2.

We refer the reader to the nice survey of Navarro and Mäkinen [49], Navarro's book [48] and other references in the literature [40, 41, 18, 25, 20, 37, 28, 27, 33, 50, 46, 16, 47, 5, 38, 31, 17, 32, 1, 34, 2, 42, 31].

For example, Foschini et al. [15] encoded the single wavelet tree of the BWT sequence using run-length encoding and Elias γ code to encode the runs; if the k -contexts are hypothetically overlaid onto the BWT sequence, the encoding of each run length adapts implicitly to the frequency statistics of the current k -context(s), thus achieving zero-order compression, and thus by Definition 2 of k th-order entropy, they encode the overall node in k th-order entropy space. We use that same idea in this paper. Mäkinen and Navarro [40, 41] did a careful analysis to show the surprising result that applying RRR [51] to a single Wavelet Tree of the entire BWT sequence without any partitioning achieves the $n \mathcal{H}_k(\mathcal{T})$ bits leading space term, for a similar reason as in [15]; the subblocks formed by [51] implicitly encode each k th-order context in roughly zero-order entropy space. Interesting experiments on the practical performance of these RLE and RRR approaches and related methods appear in [24]. Gog et al. [18] used a fixed-block boosting technique and the RRR method [51] to implement the FM-index in asymptotically optimal entropy-compressed space with an extra cost of $o(n \log \sigma)$ bits. Mäkinen et al. [42] proposed a CSA-based index for highly repetitive data collections.

These references also mention many practical applications. An example is a special-purpose BWT-based compressed index for biological FASTQ data that exploits specific characteristics of next-generation sequence data [31].

1.2 Our Results

Developing space-efficient entropy-compressed self-indexes that achieve fast query performance both in theory and practice has been a challenging problem. In this paper, we propose FM-Adaptive, a fast data-aware FM-index applicable for a wide range of text strings with different alphabet sizes. The key is a new representation of the wavelet tree of the BWT, with new tradeoffs between space occupancy and search time. We propose several auxiliary data structures to support fast access to BWT.

Using gap γ code instead of Elias-Fano (EF) code [9, 7] used in [31] for DNA sequences, we deduce a new space bound for general texts while the space bound derived by using the EF code in [31] holds only when the entropy is $o(1)$. In addition, the compressed index in [31] is geared towards FASTQ data, which have a specific format, and its queries are not identical to the queries on the general-purpose compressed indexes in this paper. Our index has an additional efficiency in that it allows the gap γ code and the run-length γ code to share the same lookup table as in [28], though the tables for the gap γ code and for the run-length γ code which we introduce in Section 3.1 for encoding methods have different meanings, thus avoiding the need to store the lookup table R_{EF} [31] for Elias-Fano code.

Ferragina, Giancarlo, and Manzini [14] discussed the run-length code and the gap code for the implementations of the Wavelet Tree. For a bit string $\mathcal{B}$ of length n , they can represent $\mathcal{B}$ in $|Rle(\mathcal{B})| \leq \max(2a, b + 2a/3)|\mathcal{B}|\mathcal{H}_0(\mathcal{B}) + 3b + 1 = 4n\mathcal{H}_0(\mathcal{B}) + 4$ bits (Lemma 3.3) using the run-length γ code and in $|Ge(\mathcal{B})| \leq \max(a, b)|\mathcal{B}|\mathcal{H}_0(\mathcal{B}) + a \log |\mathcal{B}| + b + 2 = 2n\mathcal{H}_0(\mathcal{B}) + 2 \log n + 3$ bits (Lemma 3.4) using the gap γ code for $a = 2$ and $b = 1$. Instead, we can represent $\mathcal{B}$ in $2n\mathcal{H}_0(\mathcal{B}) + 2 \log n + 2$ bits [24, 28] using the run-length γ code and in $2n\mathcal{H}_0(\mathcal{B})$ bits in Lemma 3 using the gap γ code, which avoids any second order terms.

Ferragina et al. [14] addressed the problem of compressing the BWT L , but not the indexing mechanics of how to provide fast random access to L how to decode L . In this paper, we address compression as well as the problem of how to index the data. We designed several

efficient auxiliary data structures and the lookup table for fast access and decoding to support efficient pattern matching in a text. We also present a new locating algorithm that substantially improves the locating time in practice.

For input text $\mathcal{T}$ of n symbols, we summarize below the novel contributions we make in this paper:

- (1) We present a new compressed representation for the wavelet tree of the BWT, called FM-Adaptive, that incorporates the gap γ -encoding. Using this, we can implement FM-Adaptive in $2n\mathcal{H}_k + o(n \log \sigma)$ bits, for $k \leq \alpha \log_\sigma n - 1$ and any fixed constant $0 < \alpha < 1$, where $\mathcal{H}_k$ denotes the k th-order empirical entropy of $\mathcal{T}$. We can construct FM-Adaptive in $\mathcal{O}(n \log \sigma)$ time using $\mathcal{O}(n \log n)$ bits of space. In addition, for a bit string $\mathcal{B}$ of length n , we can represent $\mathcal{B}$ in $2n\mathcal{H}_0(\mathcal{B})$ bits using the gap γ code, which avoids and improves any second order terms in space bounds in [14, 24], where $\mathcal{H}_0(\mathcal{B})$ is zero-order empirical entropy of $\mathcal{B}$, defined in Definition 2.
- (2) We present an improved algorithm that provides substantially faster performance in practice for locating patterns in a text, based upon memoization, which takes advantage of the overlapping-subproblems property.
- (3) We design table R_γ (see Section 3.4) for accelerating the decoding of the gap γ -encoded blocks to support fast pattern matching in a text.
- (4) Given any pattern $\mathcal{P}$ of m symbols encoded in $m \log \sigma$ bits, using FM-Adaptive, we can count the number of occurrences of $\mathcal{P}$ in $\mathcal{T}$ in $\mathcal{O}(m \log \sigma)$ time, we can locate all occ occurrences in $\mathcal{O}(occ \log^{1+\epsilon} n)$ additional time, and we can retrieve a text substring $\mathcal{T}[start, start + \ell - 1]$ in $\mathcal{O}(\ell \log \sigma + \log^{1+\epsilon} n)$ time.
- (5) Extensive experiments demonstrate that FM-Adaptive generally provides faster query performance, often by a considerable amount, and comparable or better compression than other state-of-the-art FM-index methods. The source code is available online [30].

1.3 Organization of the Paper

In Section 3 we introduce a new compression structure for wavelet trees. The search and locate functions are described and analyzed in Section 4. We report the experimental results in Section 5 and give final comments in Section 6.

2 Preliminaries

In this section, we define some necessary concepts and provide a brief description of the BWT.

2.1 Problem Formalization

► **Definition 1** (Text indexing problem). *Let $\mathcal{T} = \mathcal{T}[0, n - 1]$ be a text string of length n over an alphabet Σ of size σ and let $\mathcal{P}$ be a query pattern. The text indexing problem is to represent $\mathcal{T}$ in as small space as possible while efficiently supporting the following query operations:*

- *count($\mathcal{P}$): returns the number of occurrences of $\mathcal{P}$ in $\mathcal{T}$.*
- *locate($\mathcal{P}$): reports the positions where $\mathcal{P}$ occurs in $\mathcal{T}$.*
- *extract($start, \ell$): retrieves substring $\mathcal{T}[start, start + \ell - 1]$.*

2.2 Empirical Entropy

► **Definition 2** (empirical entropy). Let $\mathcal{T}$ be a text string of length n over an alphabet $\Sigma = \{0, 1, \dots, \sigma - 1\}$ of size σ . The zero-order empirical entropy of $\mathcal{T}$ is defined as

$$\mathcal{H}_0 = \mathcal{H}_0(\mathcal{T}) = \frac{1}{n} \sum_{x \in \Sigma} n_x \log \frac{n}{n_x}$$

where n_x is the number of occurrences in $\mathcal{T}$ of symbol $x \in \Sigma$. The k th-order empirical entropy [44] of $\mathcal{T}$ is defined as

$$\mathcal{H}_k = \mathcal{H}_k(\mathcal{T}) = \frac{1}{n} \sum_{\omega \in \Sigma^k} n_\omega \mathcal{H}_0(\mathcal{T}_\omega)$$

where $\omega \in \Sigma^k$ designates a string of length k and $\mathcal{T}_\omega$ denotes the string of length n_ω formed by taking the symbol immediately preceding each occurrence of ω in $\mathcal{T}$ and concatenating the symbols together.

2.3 The Burrows-Wheeler Transform and FM-index

Let $\mathcal{T} = \mathcal{T}[0, n - 1]$ be a text string consisting of n symbols from alphabet Σ of size σ , and let SA denote the suffix array of $\mathcal{T}$. The BWT L of $\mathcal{T}$ is defined as $L[i] = \mathcal{T}[SA[i] - 1 \bmod n]$. As shown in Table 1, the BWT of $\mathcal{T}$ is the string L formed by sorting the suffixes of $\mathcal{T}$ in lexicographical order, choosing the preceding symbol for each suffix, and concatenating those preceding symbols. (When the suffix is the entire string $\mathcal{T}$, we use the symbol $\#$ as its preceding symbol.)

The Burrows-Wheeler transform is invertible. We can retrieve $\mathcal{T}$ from its BWT L by backward search [10] using the mapping function LF : The value $LF(i)$ is the lexicographical rank of the suffix with prefix $L[i]$, namely, $F[LF[i]] = L[i]$, where F is the first string of the BWT.

We can compute $LF(i)$ by $LF(i) = \mathcal{C}(L[i]) + \text{Occ}(L[i], i)$, where $\text{Occ}(L[i], i)$ is the number of occurrences of symbol $L[i]$ in $L[0, i]$, and $\mathcal{C}[L[i]]$ is the number of occurrences of symbols in $\mathcal{T}$ that are smaller than $L[i]$. Table 1 shows the SA and BWT L of the string $\mathcal{T} = \text{tcaaaatatatgcaacatatagtagattgtat\#}$ in which for each i we show the first four symbols (starting with $F[i]$) and the last symbol (namely, $L[i]$) of conceptual suffixes of $\mathcal{T}$ in lexicographical order. The mapping function LF and the neighbor function Ψ are inverses of one another: $LF(\Psi(i)) = \Psi(LF(i))$, as shown in Table 1.

2.4 The Wavelet Tree

Let S denote a text string of length n over an alphabet $\Sigma = \{0, 1, \dots, \sigma - 1\}$. We define the wavelet tree (WT) [21] of S with σ leaves labelled by the symbols of the alphabet Σ as follows: For each node v in WT, let Σ_v be the subset of symbols in the subtree rooted at v , and let S_v be the substring of S consisting of all the symbols of Σ_v . For each internal node v in WT, let B_v be a bit string with the same length as S_v defined as $B_v[i] = 0$ if $S_v[i]$ is in the left subtree of v ; $B_v[i] = 1$ if $S_v[i]$ is in the right subtree of v . Such a bit string representation happens recursively at each internal node; the substring at the internal node consists of the characters dispatched from the parent node, with their order in the text string preserved. The collective size of the bit strings at any given level of the tree is bounded by n . With proper encoding, the wavelet tree representation of S achieves the zero-order entropy space bound [21].

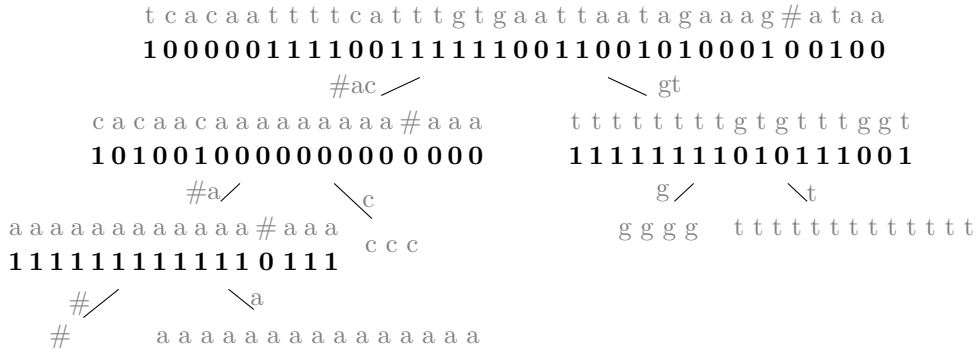
■ **Table 1** The SA and BWT L of $\mathcal{T} = \text{tcaaaatatatgcaacatatagtagattgtat}\#$. The column $\mathcal{B}$ is the bit string for the root node of the wavelet tree of L .

i	$\mathcal{T}$	SA	SA^{-1}	Ψ	LF	F	L	$\mathcal{B}$
0	t	35	31	31	23	# t c a ... a t	1	
1	c	2	16	2	16	a a a a ... t c	0	
2	a	3	1	4	1	a a a t ... c a	0	
3	a	13	2	5	17	a a c a ... g c	0	
4	a	4	4	11	2	a a t a ... a a	0	
5	a	14	11	18	3	a c a t ... c a	0	
6	t	26	28	19	24	a g a t ... t t	1	
7	a	20	12	22	25	a g t a ... a t	1	
8	t	33	29	23	26	a t # t ... g t	1	
9	a	18	13	25	27	a t a g ... a t	1	
10	t	16	32	27	18	a t a t ... a c	0	
11	g	5	20	28	4	a t a t ... a a	0	
12	c	7	17	29	28	a t a t ... a t	1	
13	a	9	3	32	29	a t g c ... a t	1	
14	a	23	5	34	30	a t t a ... g t	1	
15	c	28	18	35	19	a t t g ... a g	1	
16	a	1	10	1	31	c a a a ... #t	1	
17	t	12	27	3	20	c a a c ... t g	1	
18	a	15	9	10	5	c a t a ... a a	0	
19	t	27	25	15	6	g a t t ... t a	0	
20	a	11	7	17	32	g c a a ... a t	1	
21	g	31	22	26	33	g t a t ... t t	1	
22	t	21	30	30	7	g t a t ... t a	0	
23	a	34	14	0	8	t # t c ... t a	0	
24	t	25	34	6	34	t a g a ... a t	1	
25	t	19	24	7	9	t a g t ... t a	0	
26	a	32	6	8	21	t a t # ... t g	1	
27	g	17	19	9	10	t a t a ... c a	0	
28	a	6	15	12	11	t a t a ... a a	0	
29	t	8	35	13	12	t a t g ... t a	0	
30	t	22	33	14	22	t a t t ... a g	1	
31	g	0	21	16	0	t c a a ... t #	0	
32	t	10	26	20	13	t g c a ... t a	0	
33	a	30	8	21	35	t g t a ... a t	1	
34	t	24	23	24	14	t t a g ... t a	0	
35	#	29	0	33	15	t t g t ... g a	0	

The powerful *wavelet tree* data structure [21, 15, 24] reduces the problem of compressing a string over a finite alphabet to the problem of compressing a set of bit (i.e., binary) strings. It is an elegant and versatile data structure that allows efficient access, rank, and select queries on a text string $\mathcal{T}$ of n symbols from a multisymbol alphabet:

- $\text{access}(\mathcal{T}, i)$: returns the symbol $\mathcal{T}[i]$.
- $\text{rank}_c(\mathcal{T}, i)$: returns the number of occurrences of symbol c in $\mathcal{T}[0, i]$ for any $0 \leq i \leq n-1$ and $c \in \Sigma$.
- $\text{select}_c(\mathcal{T}, j)$: returns the position in L of the j th occurrence of symbol c for any $1 \leq j \leq n$.

A key result of Grossi et al. [21] is that if each bit string $\mathcal{B}_i$ of the internal nodes of the wavelet tree is compressed to zero-order entropy, then the cumulative encoding is a zero-order entropy encoding of $\mathcal{T}$:



■ **Figure 1** The balanced wavelet tree representation for the BWT L of $\mathcal{T}$ shown in Table 1.

► **Lemma 1** ([21]). $\sum_{i=1}^t |\mathcal{B}_i| \mathcal{H}_0(\mathcal{B}_i) = n \mathcal{H}_0(\mathcal{T})$, where t denotes the number of internal nodes of the wavelet tree.

With the extra cost for auxiliary data structures, the total space occupied by a wavelet tree is $n \mathcal{H}_0(\mathcal{T}) + o(n \log \sigma)$ bits of space [21, 15, 24].

We use a single wavelet tree to represent the BWT L , as first done in Foschini et al. [15]. Mäkinen and Navarro [39] used a single wavelet tree to represent the S string of the run-length encoding of the FM-index (i.e., the run heads). Figure 1 shows the balanced wavelet tree for the example $L = \text{tcacaatttttcatttgtgaattaatagaaag\#ataa}$ from Table 1. There are various ways [15] to form the wavelet tree, such as using a Huffman criterion [26]. Each leaf of the wavelet tree corresponds to a distinct symbol. The column labeled $\mathcal{B}$ in Table 1 is the root bit string of the wavelet tree in Figure 1. A value of 0 (resp., 1) in $\mathcal{B}$ means that the corresponding symbol in L lies in the left (resp., right) subtree. The bit string for the roots of each subtree are defined recursively.

Using the wavelet tree, we can turn each $\text{Occ}(L[i], i)$ computation into one $\text{rank}_c()$ computation on the wavelet tree.

3 Wavelet Tree Compression

In this section, we introduce a new compressed representation for bit strings of wavelet trees, which incorporates the gap γ -encoding for general text strings. In Sections 3–5, we show that this representation provides excellent query performance both in theory and practice.

3.1 Encoding Methods

We let $\mathcal{B}$ denote the bit string of length n . Let r be the number of occurrences of the least frequent bit in $\mathcal{B}$ such that $1 \leq r \leq n/2$. Let $1 \leq p_1 < p_2 < \dots < p_{r-1} < p_r \leq n$ denote the increasing sequence of positions of the least frequent bit in $\mathcal{B}$. We assume for convenience $p_0 = 0$. Let $g_j = p_j - p_{j-1}$ denote the gap sequence of pairwise differences of neighboring values of the position increasing sequence of the least frequent bit in $\mathcal{B}$ for $j = 1, 2, \dots, r$. We represent each gap using Elias γ code [8]. The resulting gap γ -encoding of $\mathcal{B}$ is the bit string $\gamma_{\text{gap}}(\mathcal{B}) = \gamma(g_1)\gamma(g_2) \dots \gamma(g_r)$. Obviously, $\sum_{j=1}^r g_j = p_r \leq n$.

On the other hand, we can view $\mathcal{B}$ as a sequence of maximal runs of identical bits $\mathcal{B} = b_1^{\ell_1} b_2^{\ell_2} \dots b_s^{\ell_s}$ for some $s \leq n$, where b_i and $b_{i+1} \in \{0, 1\}$ and $b_i \neq b_{i+1}$ for $1 \leq i < s$. We can represent the length of each run using Elias γ code [8]. The resulting run-length γ encoding of $\mathcal{B}$ is the bit string $\gamma_{\text{rl}}(\mathcal{B}) = \gamma(\ell_1)\gamma(\ell_2) \dots \gamma(\ell_s)$.

3.2 Structure

The starting point of the structure of the FM-Adaptive algorithm follows from [31], which we incorporate into the following discussion. We start the summary by introducing the compression structure to represent bit strings of the nodes of the wavelet tree. We then give an example to show how the newly introduced gap γ code works, which is the key to improve general-purpose performance.

To enable effective compression on the bit strings of the wavelet tree, we partition the bit strings of the wavelet tree nodes into blocks and categorize the blocks into three basic types:

1. blocks consisting only of all 0s or of all 1s;
2. blocks having relatively long runs of 0s or of 1s; and
3. blocks having a random-like sequence of 0s and 1s.

For each block, we choose one of the following four compression methods to minimize its coding length: All0/All1, GapG0/GapG1, RLG0/RLG1, and Plain.

Specifically, for the block of type 1 consisting only of all 0s or all 1s, we use the All0/All1 code to encode it; that is, no additional bits are needed to store it. For the block of type 2 having relatively long 0-runs or 1-runs, we use either the GapG code or the RLG code to encode it. Here GapG means that we apply the gap γ code (see Section 3.1) to encode the block. RLG means that we apply the run-length γ code (see Section 3.1) to encode the block. For the blocks of type 3 having a random-like sequence of 0s and 1s, we keep the block unchanged, denoted as Plain. This partitioning is combined with the mixed encoding to represent the wavelet tree nodes. The compressed wavelet trees (CWT) are the ones whose bit strings of nodes are partition-based and mixed-encoded.

Let $\mathcal{B}$ denote a bit string of an internal node of a wavelet tree. We use following three steps to obtain a succinct representation of $\mathcal{B}$:

1. Partition $\mathcal{B}$ into blocks $\mathcal{B}^j$ of size b , except possibly the last one.
2. Combine a/b contiguous blocks to form a superblock of size a .
3. Apply the mixed encoding to encode each block. We build the encoded sequence $\mathcal{S}$ by concatenating the encoded blocks.

We also maintain five extra structures to support fast access to $\mathcal{B}$: *SBrank*, *Brank*, *SB*, *B*, and *M*, where *SBrank* stores the number of 1s in $\mathcal{B}$ preceding the current superblock; *Brank* stores the number of 1s in $\mathcal{B}$ preceding the current block relative to the beginning of its enclosing superblock; *SB* stores the number of bits in $\mathcal{S}$ preceding the current superblock; *B* stores the number of bits in $\mathcal{S}$ preceding the current block relative to the beginning of its enclosing superblock; and *M* indicates the encoding method used in a block.

Table 2 shows the compression structure for the root bit string $\mathcal{B} = 100000111100111111001100101000100100$ of the wavelet tree of the BWT for the example string $\mathcal{T}$ from Table 1, where $n = 36$, $b = 6$, and $a = 18$. The figure shows four encodings: All1, GapG, RLG, and Plain. For purposes of illustrating all four methods, the RLG code is used to encode block 4. But in the actual algorithm, GapG would instead be selected to encode block 4, since it would result in a shorter encoding.

$$\text{rank}_1(\mathcal{B}, i) = \text{SBrank}[\lfloor i/a \rfloor] + \text{Brank}[\lfloor i/b \rfloor] + \text{lrank}(\mathcal{S}, \text{offset}, M[\lfloor i/b \rfloor]) \quad (1)$$

Using encoding sequence $\mathcal{S}$ of $\mathcal{B}$ and auxiliary structures *SBrank*, *Brank*, *SB*, *B*, and *M*, we can compute $\text{rank}_1(\mathcal{B}, i)$ by Equation (1). The *lrank* operation performs All0/All1, GapG0/GapG1, RLG0/RLG1, or Plain decodings depending upon $M[\lfloor i/b \rfloor]$ to return the number of 1s up to *offset* within block $\lfloor i/b \rfloor$ and *offset* = $i \bmod b$. The starting decoding position on $\mathcal{S}$ is $\text{SB}[\lfloor i/a \rfloor] + B[\lfloor i/b \rfloor]$. We can access $\mathcal{B}[i]$ in a way similar to $\text{rank}_1(\mathcal{B}, i)$.

■ **Table 2** The succinct index structure for the wavelet tree root bit string $\mathcal{B}$ using mixed encoding of All1, GapG, RLG and Plain, where $n = 36$, $b = 6$, and $a = 18$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
$\mathcal{B}$	1	0	0	0	0	0	1	1	1	1	0	0	1	1	1	1	1	1	0	0	1	1	0	0	1	0	1	0	0	0	1	0	0	1	0	0
M	GapG1					Plain					All1					RLG0					GapG1					GapG1										
S	1					1 1 1 1 0 0										0 1 0 0 1 0 0 1 0					1 0 1 0					1 0 1 1										
B	0					1					7					0					9					13										
SB	0																		7																	
$Brank$	0					1					5					0					2					4										
$SBrank$	0																		11																	

► **Lemma 2.** Using table R_γ of size w from Section 3.4, we can compute $\text{rank}_{L[i]}(\text{CWT}, i)$ in $\mathcal{O}((b/w) \log \sigma)$ time, for block size b .

3.3 Construction

In this section, we describe the construction of the compressed wavelet tree CWT. The CWT consists of compressed bit strings of the wavelet tree nodes. We describe the compression in Algorithm 3 in Appendix A, consisting of the encoded sequence $\mathcal{S}$ and five auxiliary structures $SBrank$, $Brank$, SB , B , and M . It is simple to see we can construct the compressed wavelet tree in $\mathcal{O}(n \log \sigma)$ time.

► **Theorem 1.** Given the wavelet tree of the BWT of text $\mathcal{T}$, we can construct the CWT in $\mathcal{O}(n \log \sigma)$ time.

3.4 Accelerating the Rank Computation

In order to accelerate the $\text{rank}_1(\mathcal{B}, i)$ computation for a gap γ -encoded block (GapG), we design table R_γ for the gap γ -encoding for every possible chunk w of $(\log n)/2$ bits. Using R_γ , we can process each $\Theta(\log n)$ bits of a γ -encoded sequence in constant time. Every *index* of R_γ , corresponding to a bit string of w bits, contains three components: R_1 , R_2 , and R_3 :

- R_1 stores the total number of decoded entries in the *index*. It is the number of decoded gaps.
- R_2 stores the cumulative sum of decoded digits (gap values) in the *index*, which corresponds to the maximum value of decoded positions.
- R_3 stores the total number of decoded bits in the *index*.

The function $\text{GapG1rank}()$ in Algorithm 4 in Appendix A implements the $\text{brank}()$ operation using R_γ for the GapG1-encoded block. The bit operator $\ll$ shifts the bits in the argument to the left by the designated number of positions.

3.5 Space Usage

We let $\mathcal{B}_i$ of length n_i denote the bit string of internal node i of the wavelet tree of the BWT L for $i = 1, 2, \dots, t = \sigma - 1$, and let n_i denote the number of bits contained in $\mathcal{B}_i$. We let r be the number of occurrences of the least frequent bit in $\mathcal{B}_i$ such that $1 \leq r \leq n_i/2$. Applying the gap γ code to $\mathcal{B}_i$, we get the resulting bit string $\gamma_{\text{gap}}(\mathcal{B}_i) = \gamma(g_1)\gamma(g_2) \cdots \gamma(g_r)$, whose space bound is given in Lemma 3; it represents an improved space bound over the ones in [14, 24]:

► **Lemma 3.** $|\gamma_{gap}(\mathcal{B}_i)| \leq 2n_i \mathcal{H}_0(\mathcal{B}_i)$.

Proof. By the definition of $\gamma_{gap}(\mathcal{B}_i)$, we have

$$\begin{aligned} |\gamma_{gap}(\mathcal{B}_i)| &= \sum_{j=1}^r |\gamma(g_j)| = \sum_{j=1}^r (2\lfloor \log(g_j) \rfloor + 1) \leq 2r \log \left(\frac{\sum_{j=1}^r g_j}{r} \right) + r \\ &= 2r \log \left(\frac{p_r}{r} \right) + r \leq 2r \log \left(\frac{n_i}{r} \right) + 2(n_i - r) \log \left(\frac{n_i}{n_i - r} \right) = 2n_i \mathcal{H}_0(\mathcal{B}_i) \end{aligned}$$

The second equality is due to Elias γ code [8]. The first inequality is due to Jensen's inequality, and the next line follows from $\sum_{j=1}^r g_j = p_r$ and $p_r \leq n_i$ (see Section 3.1) and $r \leq (n_i - r) \log(n_i/(n_i - r))$ when $r \leq n_i/2$. The last step is due to Definition 2 of zero-order empirical entropy. ◀

We partition $\mathcal{B}_i$ into blocks $\mathcal{B}_i^j$ of size b for $1 \leq j \leq \lceil n_i/b \rceil$, and represent each block $\mathcal{B}_i^j$ as a gap sequence $g_1, g_2, \dots, g_{r_j}$ of positions of the least frequent bit of $\mathcal{B}_i^j$ for some $1 \leq r_j \leq b/2$.

We compress each block $\mathcal{B}_i^j$ by choosing one of four compression methods that minimizes the encoding length $h(\mathcal{B}_i^j)$: All0/All1, GapG0/GapG1, RLG0/RLG1 and Plain. Specifically, $h(\mathcal{B}_i^j) = \min\{|\alpha(\mathcal{B}_i^j)|, |\gamma_{gap}(\mathcal{B}_i^j)|, |\gamma_{rl}(\mathcal{B}_i^j)|, |p(\mathcal{B}_i^j)|\}$, where $|\cdot|$ denotes the bit string length of the encoding.

► **Theorem 2.** *Given the BWT L of $\mathcal{T}$ of length n over an alphabet of size σ , we can represent L using the compressed wavelet tree in $2n\mathcal{H}_k(\mathcal{T}) + o(n \log \sigma)$ bits of space for any k such that $k \leq \alpha \log_\sigma n - 1$ and any fixed constant $0 < \alpha < 1$, where $\mathcal{H}_k(\mathcal{T})$ denotes the k th-order empirical entropy of $\mathcal{T}$.*

Theorem 2 follows by the analysis of Foschini et al. [15] and Huo et al. [28]. The key idea of [15] was to *implicitly* consider the k -contexts ω for purpose of analysis. The gap γ encoding adapts to each k -context as it is encountered, with some space inefficiency because gaps can span context blocks. However, the extra space is matched by the space savings achieved by using a single wavelet tree, which avoids all the space overhead that would be needed if there were an individual wavelet tree for each k -context.

Figure 2 shows bit string $\mathcal{B}_1$ of the root node of the wavelet tree for the BWT L of $\mathcal{T}$ from Table 1 and Figure 1. The second row is a hypothetical partition of L by k -contexts ω for $k = 2$, and each part is called a context block L_ω . Let k -context ω of $L[i]$ denote the length- k prefix in $\mathcal{T}$ of $\mathcal{T}[SA[i]]$. For each k -context ω , the context block L_ω of L can be formed conceptually by choosing the symbol in $\mathcal{T}$ preceding each occurrence of the k -context ω and concatenating those preceding symbols. The third row is a partition of $\mathcal{B}_1$ by contexts, and each part is called a context bit block X_ω^1 . The partition for L is the same as the partition for $\mathcal{B}_1$ according to contexts. The fourth row is a partition of $\mathcal{B}_1$ with fixed length 5. The red region (see Figure 2) comprises context bit block 11001111, corresponding to context block `ttcatttg` (a substring of L) for context `at`.

Each k -context block remains contiguous in the internal nodes of the wavelet tree, as shown in red in Figure 2 for the 2-context `at`. By adapting Lemmas 3 and 1, the terms $\log g_j$ for a given k -context ω over all the wavelet tree nodes sum up to $|L_\omega| \mathcal{H}_0(L_\omega)$ for that context, and by Definition 2 for k th-order entropy, summing over all the k -contexts ω yields $n\mathcal{H}_k$. The analysis of [15, 28] considers how and whether the various hypothetical k -contexts overlap with the blocks and superblocks, which incurs some extra space costs. The resulting code length is stated in Theorem 2.

ω	#t	aa	ac	ag	at	ca	ga	gc	gt	t#	ta	tc	tg	tt
L	t	c a c a	a t t	t t c a t t t g	t g a	a t	t a	a t a g a a a g	#	a t	a a			
$\mathcal{B}_1$	1	0 0 0 0	0	1 1	1 1 0 0 1 1 1	1 1 0	0	1	1 0	0	1 0 1 0 0 0 1	0	0 1	0 0
		$\mathcal{B}_1^1$		$\mathcal{B}_1^2$	$\mathcal{B}_1^3$	$\mathcal{B}_1^4$		$\mathcal{B}_1^5$		$\mathcal{B}_1^6$		$\mathcal{B}_1^7$		$\mathcal{B}_1^8$

■ **Figure 2** Hypothetical partition of $\mathcal{B}_1$ and L by k -context ω , for $k = 2$. The portion of $\mathcal{B}_1$ for 2-context $\omega = \text{at}$ is shaded in red.

4 Pattern Matching

In this section, we use FM-Adaptive to implement three types of string matching queries: count, locate and substring extract. We put the count query in Appendix B.

4.1 Locate Query

We sample suffix array SA in the same manner as in Huo et al. [32], and we denote by SA_s the SA sampling, where s is the SA sampling step size. We use the conceptual bit array D of length n to record the indices corresponding to the sampled values in SA_s . The locate algorithm without memoization is given in Appendix A.

► **Theorem 3.** *Let $s = \log^{1+\epsilon} n / \log \sigma$ for $\epsilon > 0$ be the step size for the suffix array sampling. For the given index range in SA of pattern $\mathcal{P}$, we can answer a locate query and find the occ occurrences of $\mathcal{P}$ using the CWT of the BWT L of $\mathcal{T}$ in $\mathcal{O}(occ \log^{1+\epsilon} n)$ time using $2n\mathcal{H}_k(\mathcal{T}) + o(n \log \sigma)$ bits of space for any k such that $k \leq \alpha \log_\sigma n - 1$ and any constant $0 < \alpha < 1$, where $\mathcal{H}_k(\mathcal{T})$ denotes the k th-order empirical entropy of $\mathcal{T}$.*

Proof. Basic operations $D[i]$ and $\text{rank}_1()$ on the bit string take constant time. Each computation of $LF[i]$ using R_γ takes $\mathcal{O}((b/w) \log \sigma)$ time by Lemma 2. We can find a SA sample in at most s steps, which takes $\mathcal{O}(s(b/w) \log \sigma)$ time. Substituting values for $b = \mathcal{O}(\log n)$, $s = \log^{1+\epsilon} n / \log \sigma$ for $\epsilon > 0$, and $w = (\log n)/2$, we get the time bound $\mathcal{O}(\log^{1+\epsilon} n)$. Thus the locate algorithm runs in $\mathcal{O}(occ \log^{1+\epsilon} n)$ time, where occ is the number of occurrences of $\mathcal{P}$ in $\mathcal{T}$.

There are n/s samplings of SA and each needs $\log(n/s)$ bits. So the space required by the sampling suffix array SA_s is $(n/s) \log(n/s) = o(n \log \sigma)$ bits for $s = \log^{1+\epsilon} n / \log \sigma$ for $\epsilon > 0$. We can store the bit array D in $o(n)$ bits and support $\text{rank}_1(D, \cdot)$ and $\text{select}_1(D, \cdot)$ in $\mathcal{O}(1)$ time [35, 6]. We can store R_γ in $o(n)$ bits for $w = (\log n)/2$. The space required by the index structure is given in Theorem 2.

Summing the space required by the CWT, R_γ , SA_s and D , we obtain the space bound of $2n\mathcal{H}_k(\mathcal{T}) + o(n \log \sigma)$ bits on the locate query. ◀

Now we consider the practical improvement of the locate query. The improvement is based upon memoization, which takes advantage of the overlapping subproblems property. Assume that we are given the index range $[l, r]$ in SA of the suffixes with prefix $\mathcal{P}$. According to our sampling method on the suffix array, for each $j \in [l, r]$, we can walk at most s steps on LF to reach a suffix array sampling. For any $i \neq j$ and $i \in [l, r]$, we could also perform the same process. The key point is that the process of finding a suffix array sampling for some $j \in [l, r]$ may include the process of finding a suffix array sampling for some $i \in [l, r]$. That is, i and j may share the same suffix sampling. If we know the suffix position $p(i)$ of i and the number of steps (say, $\#steps$) to walk on LF starting at $j \in [l, r]$ to $i \in [l, r]$, we can obtain the suffix position $p(j)$ of j by $p(j) = p(i) + \#steps$. This could allow us to reduce the number of steps walking on LF and thus speed up the locating process.

Algorithm 1 $\text{fastLocate}(\mathcal{P}, l, r)$.

```

1   $\text{pos}[0, r-l] \leftarrow [-1 \dots -1]$ 
2   $\text{dist}[0, r-l] \leftarrow [0 \dots 0]$ 
3   $\text{pred}[0, r-l] \leftarrow [-1 \dots -1]$ 
4  for  $j \leftarrow l$  to  $r$  do
5       $\text{flag} \leftarrow 0, i \leftarrow j, \text{step} \leftarrow 0$ 
6      while  $D[i] \neq 1$  do
7           $i \leftarrow LF[i]$ 
8           $\text{step} \leftarrow \text{step} + 1$ 
9          if  $l \leq i \leq r$  then
10              $\text{dist}[j-l] \leftarrow \text{step}$ 
11              $\text{pred}[i-l] \leftarrow j$ 
12              $\text{flag} \leftarrow 1$ 
13             break
14      if  $\text{flag} = 0$  then
15           $i \leftarrow \text{rank}_1(D, i) - 1$ 
16           $\text{pos}[j-l] \leftarrow (SA_s[i] \times s + \text{step}) \bmod n$ 
17 for  $i \leftarrow l$  to  $r$  do
18     if  $\text{pos}[i-l] \neq -1$  then
19          $q \leftarrow i$ 
20         while  $\text{pred}[q-l] \neq -1$  do
21              $\text{res} \leftarrow \text{pos}[q-l], j \leftarrow \text{pred}[q-l]$ 
22              $\text{step} \leftarrow \text{dist}[j-l]$ 
23              $\text{pos}[j-l] \leftarrow \text{res} + \text{step}$ 
24              $\text{pred}[q-l] \leftarrow -1, q \leftarrow j$ 
25 return  $\text{pos}$ 

```

Algorithm 2 $\text{extract}(\text{start}, \ell)$.

```

1   $i \leftarrow \text{transform}(\text{start}, \ell)$ 
2   $\text{substr} \leftarrow \text{retrieve}(i, \ell)$ 
3  return  $\text{substr}$ 
   procedure  $\text{transform}(\text{start}, \ell)$ 
4   $\text{end} \leftarrow \text{start} + \ell$ 
5   $k \leftarrow SA_s^{-1}[\lceil \text{end}/s \rceil]$ 
6   $i \leftarrow \text{select}_1(D, k)$ 
7   $\text{step} \leftarrow (s - (\text{end} \bmod s)) \bmod s$ 
8  for  $j \leftarrow 1$  to  $\text{step}$  do
9       $i \leftarrow LF[i]$ 
10 return  $i$ 
   procedure  $\text{retrieve}(i, \ell)$ 
11 for  $j \leftarrow 1$  to  $\ell$  do
12      $\text{substr}[\ell-j] \leftarrow L[i]$ 
13      $i \leftarrow LF[i]$ 
14 return  $\text{substr}$ 

```

We use structure $\text{pos}[]$ to keep the suffix positions for each $j \in [l, r]$. We compute the values of $\text{pos}[]$ in two ways:

1. by walking on LF only using SA_s ; and
 2. using the computed value of $\text{pos}[]$ and data structures $\text{pred}[]$ and $\text{dist}[]$ described below.
- The array pred has length $r-l+1$; $\text{pred}[i-l]$ records the j such that $i = LF^k(j) \in [l, r]$ for the first time. dist is an array of length $r-l+1$; $\text{dist}[j-l]$ is the number of steps using LF to walk from j to i .

fastLocate in Algorithm 1 gives the pseudocode of the practical improvement on the locate query. We initialize all $\text{pos}[], \text{dist}[],$ and $\text{pred}[]$ entries to $-1, 0,$ and -1 , respectively.

4.2 Substring Extract Query

In this section, we consider how to retrieve text substrings $\mathcal{T}[\text{start}, \text{start} + \ell - 1]$ using the CWT of L and SA_s^{-1} . We sample inverse suffix array SA^{-1} in the same manner as in Huo et al. [32]. We denote by SA_s^{-1} the SA^{-1} sampling.

The workflow to retrieve a substring using the CWT of L and SA_s^{-1} is that we first transform a given position into its rank i in SA and then retrieve $\mathcal{T}[\text{start}, \text{start} + \ell - 1]$ by walking on LF . The first step is done by the $\text{transform}()$ procedure and the second step by the $\text{retrieve}()$ procedure. Both procedures are given in Algorithm 2.

► **Theorem 4.** *Given position start and length ℓ , we can answer a substring extract query using the CWT of the BWT L of $\mathcal{T}$ in $\mathcal{O}(\ell \log \sigma + \log^{1+\epsilon} n)$ time using $2n\mathcal{H}_k(\mathcal{T}) + o(n \log \sigma)$ bits of space for any k such that $k \leq \alpha \log_\sigma n - 1$ and any constant $0 < \alpha < 1$, and $\epsilon > 0$, where $\mathcal{H}_k(\mathcal{T})$ denotes the k th-order empirical entropy of $\mathcal{T}$.*

■ **Table 3** Statistics and distribution for the tested data sets.

No	dataset	$n/2^{20}$	σ	n/r
1	DNA	386	16	1.65
2	proteins	1130	27	2.67
3	english	2108	239	3.35
4	sources	202	230	4.4
5	para	410	5	27.45
6	influenza	148	15	51.21
7	w.leaders	45	89	81.89
8	kernel	247	160	92.41
9	hg38	2998	5	1.75
10	NA12877R10	10240	5	5.67

Proof. The time complexity of the substring extract algorithm is determined by the total running time of the *transform* procedure and the *retrieve* procedure. Each computation of $LF[i]$ using R_γ of size w takes $\mathcal{O}((b/w) \log \sigma)$ time and the maximum walking length of the **for** loop is s in *transform*. Consequently, the *transform* procedure runs in $\mathcal{O}(s(b/w) \log \sigma)$ time, and the *retrieve* procedure runs in $\mathcal{O}((b/w) \log \sigma \ell)$ time.

By summing the two parts, substituting the values of $b = \mathcal{O}(\log n)$, $s = \log^{1+\epsilon} n / \log \sigma$, and $w = (\log n)/2$, we get the running time bound of the *extract* algorithm: $\mathcal{O}(s(b/w) \log \sigma) + \mathcal{O}(\ell(b/w) \log \sigma) = \mathcal{O}(\ell \log \sigma + \log^{1+\epsilon} n)$ in the worst case.

The space required by the extract query is the same as that for the locate query, which is $2n\mathcal{H}_k(\mathcal{T}) + o(n \log \sigma)$ bits. ◀

5 Experiments and Analysis

5.1 Experimental Setting, Datasets, and Measure

In this section we describe experiments performed using the environment described in [32]. We used C++ to implement the algorithms and constructed the suffix array with parameter 64 using Mori's fast lightweight suffix-sorting library.³

Table 3 summarizes some statistical characteristics of the data sets we used for the experiments. The data sets consist of four datasets from the *Pizza&Chili Corpus*⁴ (datasets 1–4), the highly repetitive data sets from the *Repetitive Corpus*⁵ (datasets 5–8), the human genome (called hg38 at UCSC) from UCSC⁶ (dataset 9), and NA12877R10 (dataset 10), formed by extracting 10 gigabytes of reads from NA12877_1, downloaded from EBI.⁷ Datasets 1–4 and 9–10 are nonrepetitive data sets, and datasets 5–8 are highly repetitive data sets. The expression n/r denotes the average run length in the BWT L , and σ is the alphabet size of the input data. For hg38, we exclude the query on pattern $NN \dots N$.

In our experiments, we examine the following state-of-the-art algorithms for their space usage and query time:

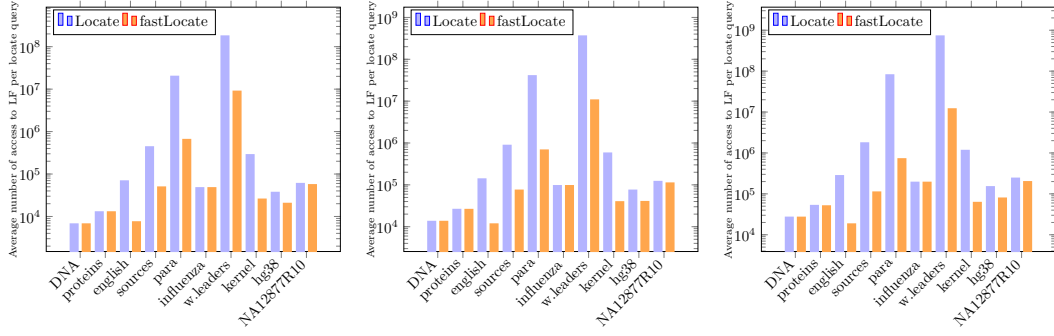
³ github.com/y-256/libdivsufsort/

⁴ pizzachili.dcc.uchile.cl/texts.html

⁵ pizzachili.dcc.uchile.cl/repocorpus.html

⁶ hgdownload.cse.ucsc.edu/goldenPath/hg38

⁷ www.ebi.ac.uk/ena/browser/view/ERR194146



■ **Figure 3** The average number of access to LF for each locate query before (in blue) and after (in orange) the memoization is used for $s = 64, 128, 256$ for the data sets shown in Table 3.

1. AF-Index: the implementation⁸ of alphabet-friendly FM-index [13, 12], which combines an existing compression boosting technique with the wavelet tree data structure. We used its latest version af-index_v2.1.
2. FMI-Hybrid: the recent implementation⁹ of the FM-index [18], which uses the fixed-block boosting technique [36], in which bitvectors by default are implemented by hybrid encoding [37].
3. RL-CSA: the implementation¹⁰ of the compressed suffix array [42] that uses run-length encoding and has been optimized for highly repetitive data. We used its current version of May 2016.
4. GeCSA: the recent implementation¹¹ of the compressed suffix array [32], in which a new run-length δ code is introduced for the gap sequence of Ψ [22, 23].
5. FM-Adaptive: our method described in this paper.

5.2 Improvement of the Locate Query by Memoization

In this section, we make an experimental comparison of the locate query before and after the improvement. The key point of the memoization improvement is to remember the computed suffix positions for some $j \in [l, r]$ and then use these computed positions and other auxiliary information to determine the suffix positions of the remaining j so as to reduce the number of accesses to LF and thus speed up the locate query.

In the experiments, we perform 500 locate queries and calculate the average number of access to LF for each locate query. Figure 3 shows the average number of access to LF for each locate query before (in blue) and after (in orange) the improvement by memoization.

As can be seen in Figure 3, the memoization of some computed suffix positions generally improves the locate query time substantially. The average number of accesses to LF is decreased by 88.69–99.11 percent on english, sources, para, w.leaders, and kernel, by 44.88–47.31 percent on hg38, by 6.29–17.82 percent on NA12877R10, by 0.05–2.2 percent on proteins, and by 0.01–0.08 percent on DNA, and by 0.0–0.01 percent on influenza for all s .

Figure 3 shows substantial reductions in LF invocations compared with the locate algorithm without memoization shown in Appendix A for most tested data. This memoization did not work on the datasets DNA, proteins, and influenza because there are few short patterns occurring frequently, which make $i = LF^k(j) \in [l, r]$ rarely satisfied.

⁸ pizzachili.dcc.uchile.cl/indexes

⁹ github.com/simongog/sdsl-lite

¹⁰ jlttsiren.kapsi.fi/rlcsa

¹¹ codeocean.com/capsule/3554560/tree/v1

■ **Table 4** The average pattern locating time T in milliseconds for three locate algorithms, for $s = 32$ (top), 64 (middle) and 128 (bottom).

Dataset	T_{ps}	T_{vs}	T_{fast}	T_{ps}/T_{fast}	T_{vs}/T_{fast}
DNA	3.71	2.18	2.31	1.6	0.94
proteins	12.21	7.30	4.76	2.56	1.53
english	54.67	25.35	3.54	15.44	7.16
sources	204.92	106.87	25.08	8.17	4.26
para	11180.81	7231.52	378.66	29.53	19.10
influenza	23.59	16.97	7.61	3.10	2.23
w.leaders	96419.82	67735.21	2979.77	32.36	22.73
kernel	163.02	96.56	8.37	19.48	11.54
DNA	7.41	4.58	4.94	1.5	0.93
proteins	27.77	15.01	8.88	3.13	1.69
english	109.92	60.72	6.16	17.84	9.86
sources	450.83	249.35	40.75	11.06	6.12
para	22379.9	14544.4	381.15	58.72	41.42
influenza	64.28	35.23	14.92	4.31	2.36
w.leaders	210913	144593	3657.32	57.67	39.53
kernel	342.34	200.46	13.24	25.86	15.14
DNA	17.80	10.39	10.30	1.73	1
proteins	54.95	32.15	20.44	2.69	1.57
english	212.27	136.36	10.39	20.43	13.12
sources	846.47	501.62	66.71	12.69	7.52
para	45335.7	28915.1	402.99	112.50	71.75
influenza	107.14	69.17	30.0	3.57	2.31
w.leaders	461394.47	293427	4446.15	103.77	65.99
kernel	725.13	422.53	21.99	32.97	19.21

To show that the improvement of the locate query is largely due to the memoization rather than the suffix array sampling method, we made additional experiments on three locate algorithms on the *Pizza&Chili Corpus*, as well as the *Repetitive Corpus*, which we show in Table 4.

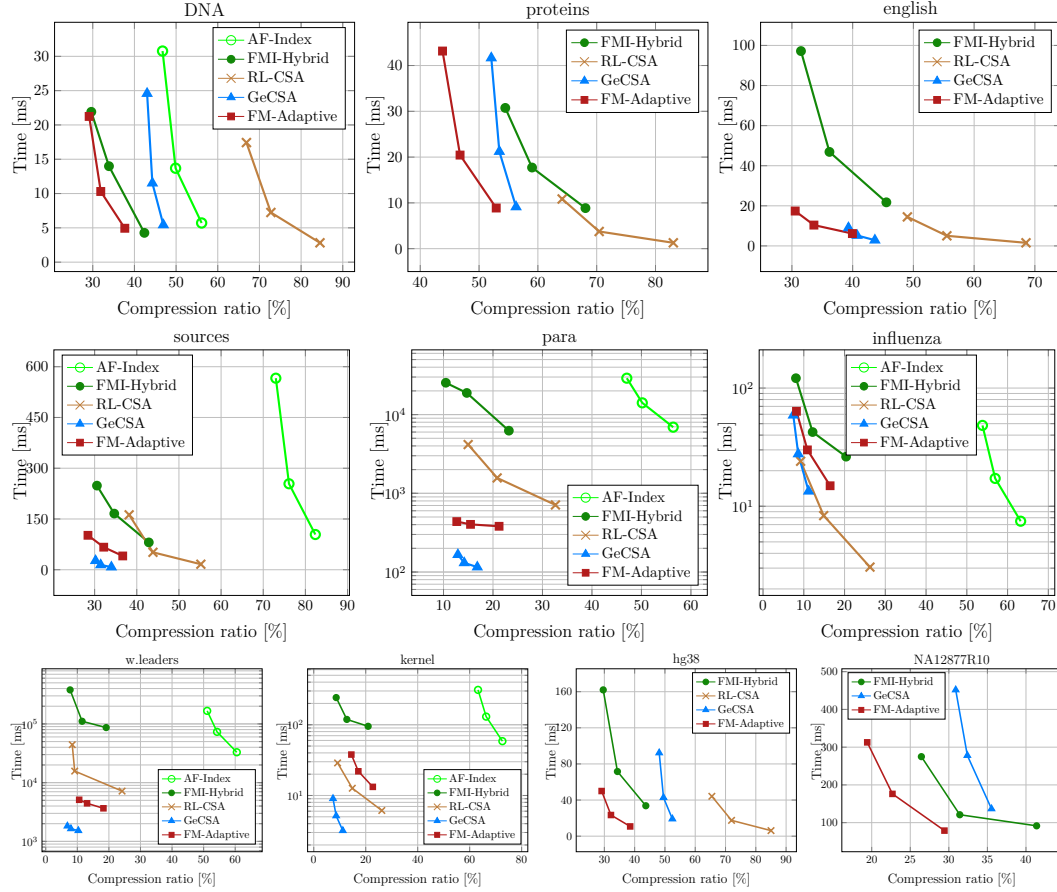
We denote the locate algorithm with position sampling in suffix array in [28] as *locate15*, the locate algorithm without memoization with suffix array value sampling (Algorithm 5 in Appendix A) as *locate*, and the locate algorithm with memoization with suffix array value sampling (Algorithm 1) as *fastLocate*. We randomly select 1000 patterns with length 20 from the input text, and average the total time over the 1000 locate queries on each of the three locate algorithms, denoted as T_{ps} for *locate15*, T_{vs} for *locate*, and T_{fast} for *fastLocate* in milliseconds, shown in Table 4. We take block size $b = 256$ for the three locate algorithms. It can be seen that *fastLocate* provides generally several times to orders of magnitude faster than *locate15* and *locate* for different suffix array sampling step size $s = 32, 64$ and 128.

5.3 Performance Evaluation

In the following sections we compare the performance of our proposed algorithms with the state-of-the-art indexing methods described above, based upon the performance criteria of compression ratio, locate query time, and extract query time. We use the following parameters for each index:

- AF-Index: $b = 32 * 8$, and $s = 64, 128$, and 256 ;
- FMI-Hybrid: $b = 256$, and $s = 32, 64$, and 128 ;
- RL-CSA: $b = 64$, $s = 32, 64$, and 128 ;
- GeCSA: $b = 256$, and $s = 128, 256$, and 512 ;
- FM-Adaptive: $b = 256$, and $s = 64, 128$, and 256 .

The other parameters used are the default parameters. The tested method used is the same as those in [32]. The compression ratio is defined as the ratio of the index size with the original size of the input text. The parameters in our index are $a = 16b$, $d = s$, and $w = 16$.



■ **Figure 4** Performance comparison of five indexing methods on the *locate* query. The measures used are compression ratio in percentage and average time T per query in milliseconds on the 10 data sets described in Section 5.1.

5.3.1 Locate Query

Figure 4 shows the compression ratio and time for the *locate* query of the compared indexing methods. For the tested data of NA12877R10, RL-CSA fails to build the index, since it limits the size of the input collection to less than 4 gigabytes. AF-Index fails to build the index on proteins, english, hg38, and NA12877R10 with input size larger than 1 gigabyte.

As can be seen in Figure 4, FM-Adaptive provides the best compression on the nonrepetitive datasets shown in Table 3. FM-Adaptive is typically several to dozens of times faster than FMI-Hybrid in query time. Specifically, for the nonrepetitive datasets 1–4 and 9–10, when minimizing compression ratio, FM-Adaptive gives the best compression and is 1.03–5.58 times

faster than FMI-Hybrid in query time in four out of six nonrepetitive datasets. Speedwise, when minimizing query time, RL-CSA shows the fastest query time in four out of six nonrepetitive datasets but at a cost of higher space usage. FM-Adaptive is 1.98–3.53 times faster than FMI-Hybrid in query time in four of six nonrepetitive datasets. FMI-Hybrid is essentially the same as and a bit faster than FM-Adaptive in query time on proteins and DNA, but it uses 15.17 and 4.7 percentage points more space.

For the repetitive datasets 5–8, when minimizing compression ratio, GeCSA gives the best compression on influenza, w.leaders, and kernel with the exception of FMI-Hybrid on para. FMI-Hybrid shows the best compression ratio on para, but it is 152.77 times slower than GeCSA. FM-Adaptive is 74.33, 58.03, 6.39, and 1.90 times faster with comparable compression than FMI-Hybrid on w.leaders, para, kernel, and influenza. Speedwise, when minimizing query time, GeCSA shows the fastest query time on para, w.leaders, and kernel about 2.4–59.93 times faster than the other three compared indexing methods with the exception of RL-CSA on influenza, but RL-CSA uses 15.06 percentage points more space than GeCSA. FM-Adaptive is 23.67, 16.38, 7.18, and 1.76 times faster with comparable compression than FMI-Hybrid on w.leaders, para, kernel, and influenza. AF-Index generally uses substantially more space on the tested data than the other compared indexing methods.

Memoization is a main reason for the improved locating query time for FM-Adaptive. Another reason is due to the different suffix array sampling. As can be seen in Figure 4, FM-Adaptive provides faster query performance by a considerable amount than the other compared FM-index methods on the tested data sets, except for DNA, proteins and influenza for which memoization barely works according to Figure 3. At the same time, FM-Adaptive provides a comparable or better compression.

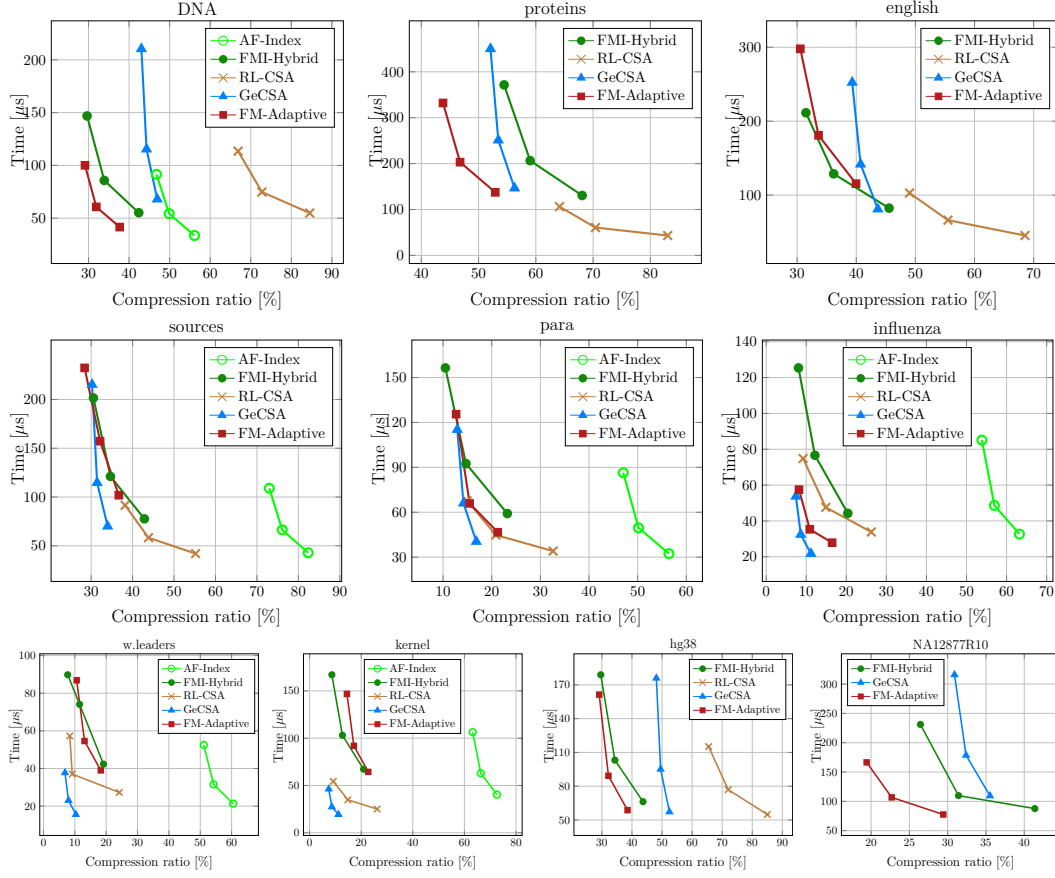
5.3.2 Substring Extract Query

The extract query uses the same index as the locate query, so it has the same space usage as the locate query. In this section we focus upon its time performance. Figure 5 shows the compression ratio and time for the extract query for the compared indexing methods.

Figure 5 shows that RL-CSA, GeCSA, AF-Index, and FM-Adaptive give the best extract time on four, three, two and one of the 10 tested datasets in Table 3. As can be seen in Figure 5, GeCSA shows good extract performance on repetitive data. GeCSA uses less space and is generally many times faster than the other indexing methods, with the exception of AF-Index on para. AF-Index is 1.44 times faster than GeCSA on para but uses 35.19 percentage points more space. The query time of FM-series indexing methods can be affected by the alphabet size σ of input text. When σ is small, such as DNA, hg38 and NA12877R10, they have a better or competitive time. FM-Adaptive is faster than FMI-Hybrid in query time in eight out of 10 datasets with the exception of FMI-Hybrid on proteins and english. FMI-Hybrid is 1.05 and 1.4 times faster than FM-Adaptive on proteins and english but uses 15.17 and 5.59 percentage points more space.

6 Conclusion

We propose improvements via a new compressed representation of the wavelet tree of the Burrows-Wheeler transform of the input text, which incorporates the gap γ -encoding. Our new index achieves asymptotic space optimality within a factor of 2 in the leading term, but it has a better compression and faster retrieval in practice than the competitive optimal compression boosting used in previous FM-indexes. We present a practical improved locate algorithm that provides substantially faster locating time.



■ **Figure 5** Performance comparison of five indexing methods on the *extract* query. The measures used are compression ratio in percentage and average time T per query in microseconds on the 10 data sets described in Section 5.1.

An interesting open problem is to improve our high-order entropy-compressed text self-index to achieve optimal space and query time both in theory and in practice. Another interesting goal is to adapt our index to compress and index the human genome and reads for the analysis of ultra-high-throughput next-generation sequencing (NGS) data, while supporting approximate matching queries.

References

- 1 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-indexing the eBWT. *Information and Computation*, 298:105155, 2024. doi:10.1016/J.IC.2024.105155.
- 2 Nathaniel K. Brown, Travis Gagie, Giovanni Manzini, Gonzalo Navarro, and Marinella Sciortino. Faster run-length compressed suffix arrays. In Alessio Conte, Andrea Marin, Giovanna Rosone, and Jeffrey S. Vitter, editors, *From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday*. OASIcs, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025.
- 3 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm, 1994.
- 4 Stefan Canzar and Steven L. Salzberg. Short read mapping: An algorithmic tour. *Proceedings of the IEEE*, 105(3):436–458, 2017. doi:10.1109/JPROC.2015.2455551.

- 5 Xiaoyang Chen, Hongwei Huo, Jun Huan, Jeffrey Scott Vitter, Weiguo Zheng, and Lei Zou. MSQ-Index: A succinct index for fast graph similarity search. *IEEE Transactions on Knowledge and Data Engineering*, 33(6):2654–2668, 2021. doi:10.1109/TKDE.2019.2954527.
- 6 David Richard Clark. *Compact PAT trees*. PhD thesis, University of Waterloo, Waterloo, Canada, 1996.
- 7 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. Assoc. Comput. Mach.*, 21:246–260, 1974. doi:10.1145/321812.321820.
- 8 Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. doi:10.1109/TIT.1975.1055349.
- 9 Robert M. Fano. On the number of bits required to implement an associative memory. Computer Structures Group, MIT, Cambridge, MA, 1971.
- 10 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00)*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.
- 11 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
- 12 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In *Proceedings of the 11th International Symposium on String Processing and Information Retrieval (SPIRE'04)*, pages 150–160, 2004. doi:10.1007/978-3-540-30213-1_23.
- 13 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):Article 20, 2007.
- 14 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. The myriad virtues of Wavelet Trees. *Information and Computation*, 207(8):849–866, 2009. doi:10.1016/J.IC.2008.12.010.
- 15 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. Shorter versions appear in *Proceedings of the 15th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA '04)*, New Orleans, LA, January 2004, 636–645, and in “Fast compression with a static model in high-order entropy,” *Proceedings of IEEE Data Compression Conference*, Snowbird, Utah, March 2004, 62–71. doi:10.1145/1198513.1198521.
- 16 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):Article 2, 2020.
- 17 Raffaele Giancarlo, Giovanni Manzini, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. A new class of string transformations for compressed text indexing. *Information and Computation*, 294:105068, 2022. doi:10.1016/J.IC.2023.105068.
- 18 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in FM-indexes: Theory and practice. *Algorithmica*, 81:1370–1391, 2019. doi:10.1007/S00453-018-0475-9.
- 19 Simon Gog, Gonzalo Navarro, and Matthias Petri. Improved and extended locating functionality on compressed suffix arrays. *Journal of Discrete Algorithms*, 32:53–63, 2015. doi:10.1016/J.JDA.2015.01.006.
- 20 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014. doi:10.1002/SPE.2198.
- 21 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 841–850, 2003. URL: <http://dl.acm.org/citation.cfm?id=644108.644250>.
- 22 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC'00)*, pages 397–406, 2000.

- 23 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.
- 24 Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *Proceedings of the 1st International Conference on Data Compression, Communications and Processing*, pages 210–221, 2011. doi:10.1109/CCP.2011.16.
- 25 Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compression, indexing, and retrieval for massive string data. In *Proceedings of the 21st annual conference on Combinatorial pattern matching (CPM'10)*, pages 260–274, 2010. doi:10.1007/978-3-642-13509-5_24.
- 26 David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- 27 Hongwei Huo, Longgang Chen, Jeffrey Scott Vitter, and Yakov Nekrich. A practical implementation of compressed suffix arrays with applications to self-indexing. In *Proceedings of the Data Compression Conference (DCC'14)*, pages 292–301, 2014. doi:10.1109/DCC.2014.49.
- 28 Hongwei Huo, Longgang Chen, Heng Zhao, Jeffrey Scott Vitter, Yakov Nekrich, and Qiang Yu. A Data-aware FM-index. In *Proceedings of the 17th Workshop on Algorithm Engineering and Experiments (ALENEX'15)*, pages 10–23, 2015. doi:10.1137/1.9781611973754.2.
- 29 Hongwei Huo, Xiaoyang Chen, Xu Guo, and Jeffrey Scott Vitter. Efficient compression and indexing for highly repetitive DNA sequence collections. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 18(6):2394–2408, 2021. doi:10.1109/TCBB.2020.2968323.
- 30 Hongwei Huo, Zongtao He, Pengfei Liu, and Jeffrey Scott Vitter. FM-Adaptive: A data-aware FM-index [source code], January 2022. doi:10.24433/C0.7967727.v1.
- 31 Hongwei Huo, Pengfei Liu, Chenhui Wang, Hongbo Jiang, and Jeffrey Scott Vitter. CIndex: Compressed indexes for fast retrieval of FASTQ files. *Bioinformatics*, 38(2):335–343, 2022. doi:10.1093/BIOINFORMATICS/BTAB655.
- 32 Hongwei Huo, Peng Long, and Jeffrey Scott Vitter. Practical high-order entropy-compressed text self-indexing. *IEEE Transactions on Knowledge and Data Engineering*, 35(3):2943–2960, 2023. doi:10.1109/TKDE.2021.3114401.
- 33 Hongwei Huo, Zhigang Sun, Shuangjiang Li, Jeffrey Scott Vitter, and et al.. CS2A: A compressed suffix array-based method for short read alignment. In *Proceedings of the Data Compression Conference (DCC'16)*, pages 271–278, 2016.
- 34 Hongwei Huo, Yongze Yu, Zongtao He, and Jeffrey Scott Vitter. Indexing labeled property multidigraphs in entropy space, with applications. In *Proceedings of the 41st IEEE International Conference on Data Engineering (ICDE'25)*, pages 2478–2492, 2025.
- 35 Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS'89)*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.
- 36 Juha Kärkkäinen and Simon J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proceedings of the 18th International Symposium on String Processing and Information Retrieval (SPIRE'11)*, pages 174–184, 2011. doi:10.1007/978-3-642-24583-1_18.
- 37 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *DCC*, pages 302–311, 2014. doi:10.1109/DCC.2014.87.
- 38 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. *Information and Computation*, 285(Part B):104818, 2022. doi:10.1016/J.IC.2021.104818.
- 39 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 45–56, 2005. LNCS 3537. doi:10.1007/11496656_5.
- 40 Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE'07)*, pages 229–241, 2007. doi:10.1007/978-3-540-75530-2_21.

- 41 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008.
- 42 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. doi:10.1089/CMB.2009.0169.
- 43 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 44 Giovanni Manzini. An analysis of the burrows-wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001. doi:10.1145/382780.382782.
- 45 Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.
- 46 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Fast compressed self-indexes with deterministic linear-time construction. *Algorithmica*, 82(2):316–337, 2020. A conference version of this paper appeared in Proc. ISAAC 2017. doi:10.1007/S00453-019-00637-X.
- 47 Gonzalo Navarro. Indexing highly repetitive string collections, part ii: Compressed indexes. *ACM Computing Surveys*, 54(2):Article 26, 2021.
- 48 Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, New York, USA, September 2016.
- 49 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.
- 50 Gonzalo Navarro and Nicola Prezza. Universal compressed text indexing. *Theoretical Computer Science*, 762:41–50, 2019. doi:10.1016/J.TCS.2018.09.007.
- 51 Rajeev Raman, Venkatesh Raman, and S.Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 2007. doi:10.1145/1290672.1290680.
- 52 Kunihiro Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of the 11th Symposium on Algorithms and Computation (ISAAC'00)*, pages 410–421, 2000. doi:10.1007/3-540-40996-3_35.
- 53 Kunihiro Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003. doi:10.1016/S0196-6774(03)00087-7.
- 54 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

A Algorithms

Algorithm 3 compress($\mathcal{B}$).

$\mathcal{B}$ denotes a node bit string of the wavelet tree.

```

1 Initialize  $l_1, l_2, k_1, k_2, n_1, n_2$  be 0
2  $S \leftarrow \emptyset$ 
3 for  $i \leftarrow 0$  to  $n - 1$  step  $b$  do
4   if  $i \bmod a = 0$  then
5      $l_2 \leftarrow l_1, n_2 \leftarrow n_1$ 
6      $SB[k_2] \leftarrow l_2$ 
7      $SBrank[k_2] \leftarrow n_2$ 
8      $k_2 \leftarrow k_2 + 1$ 
9    $B[k_1] \leftarrow l_1 - l_2, Brank[k_1] \leftarrow n_1 - n_2$ 
10   $temparr \leftarrow \mathcal{B}[i..i + b - 1]$ 
11  if  $isAllone(temparr)$  then
12     $besttype \leftarrow All1, bestlen \leftarrow 0$ 
13     $num \leftarrow b$ 
14  else if  $isAllzero(temparr)$  then
15     $besttype \leftarrow All0, bestlen \leftarrow 0$ 
16     $num \leftarrow 0$ 
17  else
18     $encodedSize(temparr, rlglen, gapglen, plainlen)$ 
19     $besttype \leftarrow minlertype\{rlg, gapg, plain\}$ 
20     $bestlen \leftarrow minlen\{rlglen, gapglen, plainlen\}$ 
21     $num \leftarrow numberofone(temparr)$ 
22  if  $besttype = rlg$  then
23     $encodeRLG(S, temparr)$ 
24  if  $besttype = gapg$  then
25     $encodeGapG(S, temparr)$ 
26  if  $besttype = plain$  then
27     $encodePlain(S, temparr)$ 
28   $l_1 \leftarrow l_1 + bestlen, n_1 \leftarrow n_1 + num$ 
29   $M[k_1] \leftarrow besttype$ 
30   $k_1 \leftarrow k_1 + 1$ 
31 return  $S, B, SB, M, Brank, SBrank$ 

```

Algorithm 4 GapG1rank($S, offset$).

```

1  $num \leftarrow 0, gapsum \leftarrow 0, pos \leftarrow 0$ 
2  $bitsum \leftarrow offset + 1$ 
3 while  $gapsum < bitsum$  and  $pos < |S|$  do
4    $getbitslen \leftarrow \min\{16, |S| - pos\}$ 
5    $index \leftarrow getbits(S, pos, getbitslen)$ 
6   if  $getbitslen < 16$  then
7      $index \leftarrow index \ll (16 - getbitslen)$ 
8    $gapsum \leftarrow gapsum + R_\gamma[index].R_2$ 
9    $pos \leftarrow pos + R_\gamma[index].R_3$ 
10   $num \leftarrow num + R_\gamma[index].R_1$ 
11 if  $gapsum = bitsum$  then
12   return  $num$ 
13 if  $gapsum > bitsum$  then
14    $gapsum \leftarrow gapsum - R_\gamma[index].R_2$ 
15    $pos \leftarrow pos - R_\gamma[index].R_3$ 
16    $num \leftarrow num - R_\gamma[index].R_1$ 
17 while  $gapsum < bitsum$  and  $pos < |S|$  do
18    $value \leftarrow gammaDecode(S[pos])$ 
19    $pos \leftarrow pos + codeLen(value)$ 
20    $gapsum \leftarrow gapsum + value$ 
21    $num \leftarrow num + 1$ 
22 if  $gapsum > bitsum$  then
23    $num \leftarrow num - 1$ 
24 return  $num$ 

```

Algorithm 5 locate($\mathcal{P}, l, r$).

```

1  $ans[0, r - l] \leftarrow [0 \dots 0]$ 
2 for  $i \leftarrow l$  to  $r$  do
3    $ans[i - l] \leftarrow GetPos(i)$ 
4 return  $ans$ 
   procedure  $GetPos(i)$ 
1    $step \leftarrow 0$ 
2   while  $D[i] \neq 1$  do
3      $i \leftarrow LF[i]$ 
4      $step \leftarrow step + 1$ 
5    $i \leftarrow rank_1(D, i) - 1$ 
6    $pos \leftarrow (SA_s[i] \times s + step) \bmod n$ 
7   return  $pos$ 

```

B

 Count Query

Given a pattern string $\mathcal{P}[0, m-1]$ of m symbols. Each symbol in $\mathcal{P}$ and $\mathcal{T}$ belongs to a fixed alphabet Σ of size $\sigma \leq n$. An occurrence of the pattern at position i of $\mathcal{T}$ means that the substring $\mathcal{T}[i, i+m-1]$ is equal to $\mathcal{P}$. The *count* algorithm given below reports the range $[l, r]$ of the first and the last occurrences of $\mathcal{P}$ in SA in which all the suffixes are prefixed by $\mathcal{P}$ using $\mathcal{C}$ and CWT.

■ **Algorithm 6** $\text{count}(\mathcal{P})$.

```

1  $c \leftarrow \mathcal{P}[m-1]$ 
2  $l \leftarrow \mathcal{C}[c], r \leftarrow \mathcal{C}[c+1] - 1$ 
3 for  $i \leftarrow m-2$  downto 0 do
4    $c \leftarrow \mathcal{P}[i]$ 
5    $l \leftarrow \mathcal{C}[c] + \text{rank}_c(L, l-1)$ 
6    $r \leftarrow \mathcal{C}[c] + \text{rank}_c(L, r) - 1$ 
7   if  $l > r$  then
8     return 0
9 return  $[l, r]$ 

```

During the searching for $\mathcal{P}$, the algorithm maintains the following invariant: Let $SA[l, r]$ denote the range of suffixes with prefix $\mathcal{P}[i, m-1]$. If $L[k]$ and $L[k']$ are the first and the last occurrences of symbol $\mathcal{P}[i-1]$ in range $L[l, r]$, then $SA[LF(k), LF(k')]$ is the range of suffixes with prefix $\mathcal{P}[i-1, m-1]$. Basic operation LF takes $\mathcal{O}(\log \sigma)$ time on the compressed wavelet tree for $b = \mathcal{O}(\log n)$ and $w = (\log n)/2$ by Lemma 2. As a *count* query does two LF operations per symbol of $\mathcal{P}$, it runs in $\mathcal{O}(m \log \sigma)$ time. (Some more complicated implementations such as that in [13] of wavelet trees make use of multiway branching to shorten the height of the tree and reduce traversal time, but we use the simpler binary branching, which works well in practice.)

► **Theorem 5.** *Let $\mathcal{P}$ be a query pattern of length m . We can answer a count query of $\mathcal{P}$ using the CWT of the BWT L of $\mathcal{T}$ in $\mathcal{O}(m \log \sigma)$ time using $2n\mathcal{H}_k(\mathcal{T}) + o(n \log \sigma)$ bits of space for any k such that $k \leq \alpha \log_\sigma n - 1$ and any constant $0 < \alpha < 1$, where $\mathcal{H}_k(\mathcal{T})$ denotes the k th-order empirical entropy of $\mathcal{T}$.*

Optimizing the Performance of the FM-Index for Large-Scale Data

Eddie Ferro   

Department of Computer and Information Science and Engineering, Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Christina Boucher¹  

Department of Computer and Information Science and Engineering, Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Abstract

The FM-index is a fundamental data structure used in bioinformatics to efficiently search for strings and index genomes. However, the FM-index can pose computational challenges, particularly in the context of large-scale genomic datasets, due to the complexity of its underlying components and data encodings. In this paper, we present a comprehensive review of efficient variants of the FM-index and the encoding strategies used to improve performance. We examine hardware-accelerated techniques, such as memory-efficient data layouts and cache-aware structures, as well as software-level innovations, including algorithmic refinements and compact representations. The reviewed work demonstrates substantial gains in both speed and scalability, making methods that use the FM-index more practical for high-throughput genomic applications. By analyzing the trade-offs and design choices of these variants, we highlight how combining hardware-aware and software-centric strategies enables more efficient FM-index construction and usage across a range of bioinformatics tasks.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases FM-Index Acceleration, Run-Length Encoding, Suffix Array Optimization, Burrows-Wheeler Transform, Efficient Backward Search

Digital Object Identifier 10.4230/OASICS.Manzini.2025.6

Acknowledgements I want to thank Travis Gagie for helpful suggestions.

1 Introduction

The introduction of the FM-index in 2000 by Ferragina and Manzini [14] was a novel concept that sought to combine the compression of word-based indices with the efficiency in searching for full-text indices. This revolutionary data structure, built on the Burrows-Wheeler Transform (BWT), offered a compact representation of text while maintaining efficient support for pattern matching queries. Using the properties of the BWT and the Suffix Array (SA), the FM-index enabled search operations, such as locating and counting occurrences of a pattern, to be performed in linear time with respect to the pattern length in a technique called backward search. Furthermore, the FM-index achieved space savings by compressing text data proportional to the empirical entropy of the input, making it an ideal tool for applications in data compression, bioinformatics, and information retrieval.

Despite its advantages, the traditional FM-index encountered significant performance bottlenecks when applied to large-scale highly repetitive datasets. These datasets, common in fields like bioinformatics and genomics, often feature long stretches of repeated or similar sequences that exacerbate the trade-offs between time efficiency and memory usage. The primary challenge lay in balancing the need for compact data representation with the ability

¹ Corresponding author



© Eddie Ferro and Christina Boucher;
licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 6; pp. 6:1–6:21

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to perform search operations efficiently. The original FM-index construction [14], while theoretically robust, struggled with memory overhead when applied to repetitive data due to its reliance on suffix arrays and wavelet trees, which scale poorly in such scenarios. Additionally, the cache-unfriendly random-memory accesses required to the BWT during backward search steps further limited its practical efficiency on modern hardware.

These limitations spurred a wave of research focused on enhancing the FM-index by introducing computational techniques that leverage the repetitive nature of input data to optimize storage and query performance. For example, the Run-Length FM-index (RLFM) introduced by Mäkinen and Navarro [26, 22] applied run-length encoding to the Burrows-Wheeler Transform (RLBWT), significantly reducing space requirements in repetitive datasets while maintaining efficient backward searches. Similarly, Grammar Compression Induced by Suffix Sorting (FIGISS) [10] employed grammar-based compression to accelerate backward search operations, particularly for long patterns, by replacing substrings with nonterminal symbols derived from suffix sorting. The Prefix-Free Parsing FM-index (PFP-FM) [17], in contrast, utilized a parsing technique to divide text into phrases, allowing search operations to process larger segments of the pattern at each step, thereby reducing the number of backward search iterations needed.

These advancements collectively aim to address the inherent space-time trade-offs in FM-index operations, offering tailored solutions for various use cases and dataset types. Each method innovatively adapts the FM-index framework to better suit the demands of large-scale repetitive data, ensuring its continued relevance and efficiency in modern applications.

This review provides an in-depth exploration of these advancements, focusing on the theoretical innovations and practical implementations that have significantly improved the FM-index's performance. By analyzing the methods used to compress repetitive data, accelerate backward search operations, and enhance scalability, this paper aims to present a comprehensive overview of the current state of FM-index technology and its applications in handling large and repetitive datasets.

2 Preliminaries

Throughout this section, we will discuss key data structures and algorithms used for string processing, focusing on their foundational definitions and properties. These concepts are vital for understanding the time and space efficiency of various text indexing methods, especially in the context of large datasets. Starting with basic string definitions, we will build up to more advanced structures such as suffix trees, arrays, and compression techniques, highlighting their roles in efficient pattern matching and data compression.

2.1 Basic Definitions

A string T is a finite sequence of symbols $T = T[1..n] = T[1] \cdots T[n]$ over an alphabet $\Sigma = \{c_1, \dots, c_\sigma\}$ whose symbols can be unambiguously ordered. We denote ε as the empty string, $|T|$ as the length of T , σ as the size of Σ , and the rotation of T is a transformation where the characters are circularly shifted so that the first character becomes the last character of the rotated string. A bitvector B is defined as a finite sequence $B = B[1..n] = B[1] \cdots B[n]$ over the alphabet $\Sigma = \{0, 1\}$. Unlike standard arrays, which typically require $\Theta(n \log \sigma)$ bits to store n elements on an alphabet of size σ , a bitvector uses only n bits, making it a highly space-efficient representation when working over binary data.

We define the substring of $T[i] \cdots T[j]$ of T as $T[i..j]$ starting at position i and ending in position j , with $T[i..j] = \varepsilon$ if $i > j$. For a string T and $1 \leq i \leq n$, $T[1..i]$ is called the i -th prefix of T , and $T[i..n]$ is called the i -th suffix of T . We call a prefix $T[1..i]$ of T a *proper prefix* if $1 \leq i < n$. Similarly, we call a suffix $T[i..n]$ of T a *proper suffix* if $1 < i \leq n$. Given a set of strings $\mathcal{S}$, $\mathcal{S}$ is *prefix-free* if no string in $\mathcal{S}$ is a prefix of another string in $\mathcal{S}$.

We use $\prec$ to denote the lexicographic ordering of two strings, which is defined as follows. For two strings $T_2[1..m]$ and $T_1[1..n]$, $T_2 \prec T_1$ if T_2 is a proper prefix of T_1 , or there exists an index $1 \leq i \leq \min(n, m)$ such that $T_2[1..i-1] = T_1[1..i-1]$ and $T_2[i] < T_1[i]$. We define the co-lexicographic order of two strings, T_1 and T_2 , as the lexicographic order obtained by reading the strings from right to left. We assume that the string T is terminated by a special symbol $\$$ that is lexicographically smaller than any other symbol in Σ , unless otherwise specified. This provides the technical convenience of allowing all rotations of a string and all suffixes of the same string to have coinciding lexicographic ordering.

We define a **count** query and a **locate** query for some pattern P as follows. **count**(P) returns the number of occurrences of the pattern in T . **locate**(P) returns all positions in T where this pattern occurs. These queries can be answered without access to the text by using an index of the text.

Throughout this survey, the time and space complexities will be specified in the RAM model of computation, where programs are run on a random-access memory where words of $w = \log n$ bits are accessed and manipulated in constant time. Therefore, we will generally measure space in words as well. Moreover, in this model, we assume that all ordinary arithmetic and logical operations are performed in constant time.

2.2 Rank and Select

For a string T of length n , a symbol $c \in \Sigma$, and a position i in T , the rank function $T.\text{rank}_c(i)$, is defined as the number of occurrences of c in $T[1..i]$. Given an integer i , the select function $T.\text{select}_c(i)$, returns the position of the i -th occurrence of c in T , if it exists.

For a bitvector B , we will instead define the rank function $B.\text{rank}(i)$, as the number of occurrences of 1 in $B[1..i]$, and the select function, $B.\text{select}(i)$, as the position in B of the i -th occurrence of 1. We note that for a bitvector we automatically assume that the character of interest is 1.

2.3 The Suffix Trie and Suffix Array

The *suffix tree* [39, 24, 1] is a compressed trie storing the suffixes of a string T , where paths with a single child are compressed into a single node with the edge labeled by the concatenation of the compressed edge symbols. As there are n suffixes and no suffix can be repeated twice, there are exactly n leaves. Furthermore, since there are no 1-degree nodes and n leaves, the suffix tree can be stored in the $O(n)$ space. Assuming that T is defined by a constant-size alphabet or an alphabet consisting of integers in a polynomial range, it can also be built in $O(n)$ time [39, 24, 38, 12].

The *suffix array* [23] (denoted as **SA**) was introduced to reduce the space consumption of the suffix tree. For a given string T of length n , the **SA** is the array $\text{SA} = [1..n]$ defined such that $\text{SA}[i]$ is the text position of the i -th suffix of T in lexicographical order. The theoretical space required by the **SA** is still $O(n)$, but only a single integer is stored for each suffix, so the practical space is much smaller than that of the suffix tree. Another benefit of **SA** is that it is possible to build without having to build the suffix tree in $O(n)$ time [19].

Both the suffix tree and the suffix array can be used to match a pattern $P = [1..m]$ to all occurrences in T . In the suffix tree, the process involves traversing the nodes from the root symbols matching P to T , eventually ending at some node or on the edge between two nodes. All paths that follow from this point represent a suffix that starts with the pattern P and as such the pattern has been matched. This is done in $O(m)$ time assuming that the child node that extends the path can be located in constant time through the use of hashing or some $O(1)$ lookup data structure. Given the suffix array of T , the query pattern can be found by performing a binary search over the suffix array. The binary search takes $O(m \log n)$ time and returns an interval in the suffix array that represents all occurrences of the query pattern in T . A binary search can be used because the suffixes are sorted lexicographically, so suffixes that start with the same prefix are grouped together. This bound can be reduced to $O(m + \log n)$ with the use of additional data structures [23].

We now define the inverse of SA as *inverse suffix array* (ISA). Given a suffix array of T , SA, the corresponding inverse suffix array, ISA, is defined as $ISA[i] = j$ if and only if $SA[j] = i$. In other words, this structure returns the position of a suffix in the SA given its starting position in the text. Furthermore, many applications, such as FM-index, that make use of SA do not store the entire SA and instead store a sample of SA values, which is called the *sampled SA*.

2.4 Burrows-Wheeler Transform and LF-mapping

The Burrows-Wheeler Transform (BWT[1..n]) [8] of a string $T[1..n]$ is a reversible permutation of the characters of the string. This means that it is possible to retrieve the original string, T , from its BWT. Let us denote the matrix of all sorted rotations of T as M_{BWT} , and F and L as the *First* and *Last* column of this matrix, respectively. By definition, L is the BWT of T and F contains all the characters in T in lexicographic order. If we were to only consider for each row of M_{BWT} from the first character of each rotation to the special character $\$$ (the last character in T), we would have the suffixes of the original input in lexicographic order. That is exactly what is stored by the SA and this insight allows for the BWT to be constructed from the SA. More formally, this is defined as $BWT[i] = T[n] = \$$ if $SA[i] = 1$ and $BWT[i] = T[SA[i] - 1]$ otherwise. More precisely, the BWT is storing the characters that occur before each suffix in lexicographic order. The BWT can be built directly in $O(n)$ time and $O(n \log \sigma)$ bits of space [25], or built from the SA of T using the formulas above.

A key property of the BWT is that the i -th occurrence of a character in L and the i -th occurrence of the same character in F are the same character in T . This property comes from the fact that the BWT is constructed by lexicographically ordered cyclic rotations. If we consider the row of the first occurrence of a character c in L , the character, say c' , in F in that same row is the character that follows c in the original input. Since this is the first occurrence of c , this character, c' , is also the lexicographically smallest character that follows any occurrence of c . During the sorting of F all c occurrences are grouped together and their order is determined by what follows them and since c' is the lexicographically smallest, the first occurrence of c in L is the first occurrence of c in F . Next, we denote $C[c]$ as the number of suffixes starting with a character smaller than c , which can be derived from F by counting the number of occurrences of each character. We now define the LF-mapping of the BWT as follows:

► **Definition 1** ([14]). *Given the BWT of a string T and the array C , the LF-mapping function is $LF(i) = C[c] + BWT.rank_c(i)$, where $c = BWT[i]$.*

The LF-mapping function relates the row of the i -th occurrence of a character in L to the row of the i -th occurrence of the same character in F and makes it possible to reconstruct the string T from its BWT. To reconstruct T , we set an iterator $t = 1$ and $T[n] = \text{BWT}[1]$ and for each $i = n - 1, \dots, 1$ do $t = \text{LF}(t)$ and $T[i] = \text{BWT}[t]$. The way this works is that the LF-mapping is used to find the row of M_{BWT} where F contains the same character as $L[t] = \text{BWT}[t]$ and since BWT stores the character that occurs before the character in F in T , then BWT in this row gives us the next character in reconstruction. The LF-mapping is essential in the FM-index and all related indexes, as will be shown.

2.5 FM-Index and Backwards Search

The FM-index by Ferragina and Manzini [14] is a compressed full-text index and the first self-index, an index that also replaces the text entirely. Formally, it makes use of the BWT and the SA to efficiently support **count** and **locate** queries, essential queries in pattern matching. The BWT of the input string supports **count** queries through the use of LF-mapping in a technique called backward search [14]. Backward search is formally defined as follows.

► **Definition 2** ([14]). *Given a pattern $P = [1..p]$, the backward search consists of p steps that follow the same procedure: at the i -th step, sp stores the position of the first row of M_{BWT} prefixed by $P[i, p]$ while ep stores the position of the last row of M_{BWT} prefixed by $P[i, p]$. To advance from i to $i - 1$, we use LF-mapping on sp and ep as follows $sp = C[c] + \text{BWT.rank}_c(sp - 1) + 1$ and $ep = C[c] + \text{BWT.rank}_c(ep)$. At the final step, if $ep < sp$, then the pattern does not occur in the input text; Otherwise, they represent the range of rows of M_{BWT} prefixed by P .*

Using the result of the backward search, a **count** query can be answered simply by providing the size of the range, or $ep - sp + 1$. Note that none of the steps of the backward search rely on the complete BWT matrix M_{BWT} , it is sufficient to retain only the BWT, which is commonly stored in compressed form, and the array $C[1..\sigma]$. It is also easy to see that the *backward search* algorithm takes $O(|P|)$ time, assuming that each $\text{LF}(i)$ computation can be performed in constant time. Consequently, a **count** query can be answered by the FM-index in $O(|P|)$ time under this assumption [14, 13]. The SA of the input string enables all occurrences to be efficiently located. A range in BWT directly corresponds to a range in SA, which means that locating the occurrences would be trivial with the full SA. However, in FM-index only a **sampld** SA is stored. We formally define the process of locating occurrences with a **sampld** SA as follows.

► **Definition 3** ([14]). *Given an interval $[s_p, e_p]$, locating occurrences consists of $occ = e_p - s_p + 1$ steps that all have the following invariant: at the i -th step, where $x = s_p + i$, if $\text{SA}[x]$ exists then return that value. Otherwise perform $x = \text{LF}(x)$ a total of v times until $\text{SA}[x]$ exists and then return $\text{SA}[x] + v$.*

The FM-index stores SA values at regular positions in the text T . Specifically, for some sampling parameter, s , each position $\text{SA}[i]$ is kept if $\text{SA}[i] \bmod s \equiv 0$. As a result, locating each of the occ occurrences requires traversing up to s LF-mapping steps, yielding a total time complexity of $O(occ \cdot s)$. In the original implementation, $s = \log^{1+\epsilon}(n)$ for some $\epsilon > 0$. The FM-index occupies $\frac{5nH_K}{\log n} + O(\frac{n \log \sigma}{\log n})$ words of space.

2.6 Wavelet Trees

The *wavelet tree* is a data structure by Grossi et al. [16] that can be used to store a string T over an alphabet Σ of size σ achieving $\frac{n \log \sigma}{\log n} + o(\frac{n \log \sigma}{\log n})$ words of space and $O(\log \sigma)$ time rank and select operations over the input text. To achieve these bounds, we assume

that the underlying bitvectors of the wavelet tree support constant-time rank and select. A wavelet tree is made up of $\log \sigma$ bitvectors organized in a balanced binary tree structure. The bitvector at the root, B_{root} , is the same size as the input T and is defined by choosing a character $v \in \Sigma$, usually the median character in lexicographic order, such that $B_{root}[i] = 0$ if $T[i] < v$ and $B_{root}[i] = 1$ otherwise. The positions marked as 0 form the subsequence that is used for the left child and the positions marked as 1 form the subsequence for the right child. The bitvectors for each of these children are created in the same way as the root, keeping in mind that each child's subsequence is on half the alphabet of the parent sequence. This partitioning is performed recursively until the bitvector of a node consists entirely of 0s or 1s, which denotes that this node is a leaf. Answering the query $T.\text{rank}_c(i)$ is done by traversing the tree and computing rank on the bitvector at each level. We initialize $pos_0 = i$ and continue down the left subtree with $pos_{j+1} = B.\text{rank}_0(pos_j)$ if $c < v$ or down the right subtree with $pos_{j+1} = B.\text{rank}_1(pos_j)$ otherwise. The query stops when a leaf is reached and the result is pos_h , where h is the height of the tree. The query $T.\text{select}_c(i)$ is answered by first identifying the leaf node for c , which can be done using a rank query. Then, the tree is traversed from the leaf node to the root through select queries on every node's bitvector. We initialize $pos_h = i$ and if the current node is the left child of its parent, we traverse with $pos_j = B.\text{select}_0(pos_{j+1})$; otherwise, we traverse with $pos_j = B.\text{select}_1(pos_{j+1})$. The result of the query is then pos_0 , which we get once we reach the root.

2.7 Prefix-Free Parsing

Prefix-Free Parsing (PFP) is a technique created by Boucher et al. [5] that compresses and replaces the input before constructing the index. This allows large data sets, such as pangenomes, to be indexed without storing the full input, reducing practical memory requirements. PFP takes as input a string T of length n , and two parameters, which we denote w and p . It produces a parse and a dictionary of T , denoted as P and D , respectively, which can be used to reconstruct the original input. The dictionary consists of unique overlapping phrases stored in their lexicographic ordering, and the parse gives the order in which these phrases appear in the input. The Prefix-Free in the name of the algorithm is because the output has the property that none of the suffixes of length greater than w of the phrases in D is a prefix of any other. This concept is formalized in the following lemma. We refer to PFP of T as $\text{PFP}(T)$, consisting of the dictionary D and the parse P .

► **Lemma 4** ([5]). *If we have a string T and $\text{PFP}(T)$, then the set S of distinct proper phrase suffixes of length at least w of the phrases in D is a prefix-free set.*

In order to construct these data structures, PFP first appends w copies of $\$$ – a special character that is lexicographically smaller than any element in Σ – to both the beginning and end of T . For the sake of explanation, we consider the string $T' = \$^w T \$^w$. Next, we define the set of trigger strings E , which determine how to parse T' . Given a parameter p , we construct the set of trigger strings by computing the Karp-Rabin hash, $H_p(t)$, of substrings of length w by sliding a window over T' . E is then the set of substrings $t = T'[i..i + w - 1]$, where $H_p(t) \equiv 0$ or $t = \$^w$. Then, using E , we can scan through T' , store every substring that starts and ends with a trigger string in the dictionary, and then only keep the unique phrases. The parse is then a string of integers, where each integer is a position in the dictionary, and the string indicates the order in which these phrases appear in T' . Note that by definition of the trigger strings and by appending w copies of $\$$, that T' starts and ends with a trigger string. PFP can be used as a preprocessing step to build data structures such as the BWT, SA, and LCP.

3 Optimization via Sampling and Run-Length Encoding

3.1 Run-Length Encoding the FM-Index

Mäkinen and Navarro introduced a major improvement in the space requirements of the FM-index in 2004 [26, 22] with the RLFM index. The key idea behind the RLFM index is to exploit the structure of the run-length encoded Burrows–Wheeler Transform (RLBWT). The RLBWT is a data structure that encodes the BWT in $O(r)$ words, where r is the number of runs in the BWT, and supports $LF(i)$ queries in $O(\log \log_w(\sigma + n/r))$ time. Here, a run in a string is defined as a maximal sequence of consecutive identical characters; thus, the number of runs corresponds to the number of such maximal contiguous blocks in the BWT. For example, if $BWT = \$aaacgggttacc$, then this BWT has seven runs. The construction of the BWT matrix groups similar prefixes together, indicating shared context in the input. This common context leads to the repeating of the same letter for several rows in BWT and is especially true in highly repetitive data where the number of runs, r , can be significantly less than the number of characters in the BWT. The RLBWT can be constructed indirectly or directly. The indirect construction of the RLBWT, which is the type of construction done in the RLFM index, is done by first constructing the BWT and then run-length encoding it. This can be done $O(n)$ time and $O(|PFP|)$ space [5] or $O(n)$ time and $O(\frac{n \log \sigma}{\log n})$ space [3, 25]. Direct construction of the RLBWT can be done in optimal time ($O(n + r \log r)$ time) and BWT-runs bounded space ($O(r)$ space) [29].

Taking advantage of the compressibility of BWT through run-length encoding, the RLFM index achieves substantial space savings while maintaining efficient `count` query performance. Mäkinen and Navarro run-length encode the BWT by replacing it with a bitvector, B , of length the size of the input that contains a 1 at the start of every run in the input, and a string, S , which lists all the characters that make up each run in order. Consider the earlier example where $BWT = \$aaacgggttacc$. This sequence is replaced by two components: a bitvector $B = 1100110010110$, where each 1 denotes the start of a new run, and a string $S = \$acgtac$, which stores the distinct characters corresponding to each run in the order of appearance in BWT. The authors demonstrate that the LF-mapping can still be computed using only these representations. To support this, a new bitvector B' is derived by performing a stable sort of the runs encoded in B , where the sorting is based on the lexicographic order of the associated character of each run in S , and stability ensures that the original relative order among runs of the same character is preserved. For the given example, this results in $B' = 1100111010010$. Additionally, instead of storing S directly, a bitvector, S_c , is stored for each character c storing a 1 at every index where c occurs in S . In our example, we would then have the following bitmaps: $S_{\$} = 1000000$, $S_a = 0100010$, $S_c = 0010001$, $S_g = 0001000$, and $S_t = 0000100$. Mäkinen and Navarro compute the LF-mapping in the RLFM index by presenting an equivalent equation to $C[c] + BWT.rank_c(i)$ that uses B , B' , and the S_c bitmaps. We denote that C_S is the same as the array C , but over the string S rather than the input. The authors present the following lemma.

► **Lemma 5** ([22]). *Given S and B' for some BWT for any $c \in \Sigma$:*

$$C[c] = B'.select(C_S[c] + 1) - 1$$

To prove this lemma, the authors establish that since $C_S[c]$ gives the number of runs of characters smaller than c and B' contains the runs sorted lexicographically, then $B'.select(C_S[c] + 1)$ is the position in B' corresponding to the first run of c . It follows that $B'.select(C_S[c] + 1) - 1$ is the sum of the run lengths of all characters smaller than c , which is equivalent to the number of characters smaller than c in BWT, or $C[c]$. The authors used this to then give the equation for the LF-mapping in RLBWT as follows.

► **Lemma 6** ([22]). *Given S , B' , and B of some RLBWT. For any $c \in \Sigma$ and position $1 \leq i \leq n$ that is the last position of a run:*

$$C[c] + \text{BWT.rank}_c(i) = B'.\text{select}(C_S[c] + 1 + S_c.\text{rank}(B.\text{rank}(i))) - 1$$

The proof starts with the fact that $B.\text{rank}(i)$ gives the position of the run that finishes at i in S , which means that $S_c.\text{rank}(B.\text{rank}(i))$ is the number of runs of c in the BWT from 1 to i . Therefore, $C_S[c] + 1 + S_c.\text{rank}(B.\text{rank}(i)) - 1$ points to a position in B' within the runs of c or to the character immediately after, which means that $B'.\text{select}(C_S[c] + 1 + S_c.\text{rank}(B.\text{rank}(i))) - 1$ is the starting position for the runs of c plus the sum of the run lengths of c in $\text{BWT}[1..i]$. This is the same as $C[c] + S_c.\text{rank}(B.\text{rank}(i))$. If $\text{BWT}[i] \neq c$, then this is the LF-mapping function. Otherwise, it is $B'.\text{select}(C_S[c] + 1 + S_c.\text{rank}(B.\text{rank}(i))) + i - B.\text{select}(B.\text{rank}(i))$, because the position we are looking for is within the run c , so the added term of $i - B.\text{select}(B.\text{rank}(i))$ gives how far into the run i is.

Mäkinen and Navarro presented an improved FM-index that when performing solely **count** queries required less space than the FM-index and achieved the same query time. The authors tested their implementation against the implementation of the FM-index, the original SA, the Compressed Suffix Array (CSA) [36], and the LZ self-index [27] using random patterns from the index text. Their index required 0.63 of the original input size, which is less than every competing method except for CSA with tuned parameters. In terms of **count** query time, RLFM was only slower than the implementation of the FM-index, requiring around 0.1 msec to **count** occurrences for the longest pattern length compared to around 0.05 msec. In order to answer **locate** queries, RLFM uses the same SA sampling techniques as the FM-index, which requires more than $O(r)$ words of space. The RLFM with **locate** therefore requires more space than CSA and less than the FM-index, but had comparable **locate** query times across all pattern lengths.

3.2 Sampling the Suffix Array: the r -index

Although **count** queries could be performed in $O(r)$ space with the RLFM index, it was not until the development of the r -index index by Gagie et al. [15] that **locate** queries could also be supported within $O(r)$ words of space. This is because the SA values are needed to answer a **locate** query, and storing the full SA requires $O(n)$ space. Even when sampling the SA values at a constant rate, as was done in the original FM-index, the space required would be linearly proportional to the input size. Policriti and Prezza's Toehold Lemma [35, 34] enabled SA values to be sampled once per run while still supporting **locate** queries. The lemma is as follows.

► **Lemma 7** ([15]). *With $O(r)$ -words and given a pattern $P[1..m]$, the interval $\text{BWT}[s_p..e_p]$ of all occurrences of P in the input T and the value $\text{SA}[p]$ of some position p in the interval $[s_p, e_p]$ can be found in $O(\log \log(n/r))$ time.*

The Toehold Lemma states that by storing a predecessor data structure in $O(r)$ words containing SA values at the start and end of each run in the BWT, it is possible to maintain a “toehold” – that is, an SA value for at least one position within the current BWT interval – at every step of the backward search. To prove this, assume that a pattern P , so far matched with $i + 1..m$, the interval $\text{BWT}[s_p, e_p]$ has been found, and $\text{SA}[p]$ is known for some index p in the BWT interval. If the next character that matches $P[i]$ is the same as $\text{BWT}[p]$, then the value of the SA for some index p' in the new interval $[s'_p, e'_p]$ can be found by setting $p' = \text{LF}(p)$ and $\text{SA}[p'] = \text{SA}[p] - 1$. If instead $\text{BWT}[p] \neq P[i]$ and P occurs in the text,

then there must exist an index u such that $s_p \leq u \leq e_p$ and $\text{BWT}[u] = P[i]$, as well as at least one index in $[s_p, e_p]$ for which BWT differs from $P[i]$. This implies that the next interval intersects a run boundary and, therefore, a sampled SA value is stored for this position. Finding this SA value requires a predecessor query to the earlier defined predecessor structure that can be completed in $O(\log \log(n/r))$ time. Therefore, the lemma is proven using induction. Improvements to the Toehold lemma by Bannai et al. [2] show that the SA only needs to be sampled at run starts for the proof to remain valid, which helps improve practical performance but the $O(r)$ words of space bound remains.

In their paper, Gagie et al. show that knowing a position in the BWT and its SA value (through Lemma 7), it is possible to compute the SA values of neighboring rows in $O(\log \log(n/r))$ time. To do this, the authors used the definition of the ϕ -array of Kärkkäinen et al. [18].

► **Definition 8.** *Given a string S , SA of S , and ISA of S , we define the function ϕ as follows, $\phi(i) = \text{SA}[\text{ISA}[i] - 1]$ if $\text{ISA}[i] \neq 1$, and $\phi(i) = \text{SA}[n]$ otherwise. This allows us to move to the SA value before the current one via ϕ . We define ϕ^{-1} symmetrically, allowing us to move to the next SA value from the current one.*

The presented lemma states that by storing an $O(r)$ space predecessor data structure it is possible to evaluate ϕ and ϕ^{-1} in $O(\log \log(n/r))$ time. The predecessor data structure stores for every position i in the text corresponding to a start of the run the positions in the text (the SA values) of the preceding and following position of the position i 's corresponding position in BWT, q . Essentially, the predecessor data structure, N , can be defined as follows for every i : $N[i] = \langle \text{SA}[q - 1], \text{SA}[q + 1] \rangle$ for $i = \text{SA}[q] - 1$. To then find the SA values neighboring for a given position, k , in the text (equivalently position p in the BWT), first a predecessor query is done to find the stored position i that starts the run that contains k to get the entry $N[i] = \langle \text{SA}[q - 1], \text{SA}[q + 1] \rangle$. The neighboring values are defined as $\text{SA}[p - 1] = \text{SA}[q - 1] + k - i$ and $\text{SA}[p + 1] = \text{SA}[q + 1] + k - i$. The proof for this relies on the fact that contiguous positions in the BWT that are in the same run tend to remain contiguous after applying LF-mapping. This lemma shows that if a `count` query on a pattern returns a BWT interval and a SA value is known for a position in this interval, it is possible to identify the SA values for every position in the interval. Lemma 7 guarantees that a position in the BWT interval is always known. This allows for `locate` queries to be answered with queries to $O(r)$ space predecessor data structures in $O(\log \log(n/r))$ time. Gagie et al. also establish that it is possible to increase the size of index in order to achieve optimal time `count` and `locate` queries.

The r -index combines of the SA and BWT samples sampled at run boundaries requiring $O(r)$ words of space, and supporting `count` queries in $O(m \log \log(n/r))$ time and `locate` queries in $O((m + \text{occ}) \log \log(n/r))$ time, where m is the length of the query pattern and occ is the number of occurrences of the pattern in the input. In their experiments, r -index outperforms all competing implementations of the FM-index with respect to space-time trade-offs, taking at most the same amount of space and being magnitudes faster in `locate` queries. Compared to indexes based on Lempel-Ziv compression or grammars, r -index uses more space but remains significantly faster for `locate` queries.

3.3 Sampling the Suffix Array: the sr-index

The subsampled r -index, or sr -index, was developed by Cobas et al. [9] to reduce the space requirement of the r -index, which can be large compared to Lempel-Ziv-based indexes when r is large. The main principle of sr -index is to reduce redundancy by removing run

samples that are within a text distance of at most s from another sample. Here, a run sample refers to a value stored for the start of a run in the BWT, whether this is a bit in a bitvector or a SA value. Doing this effectively reduces the space required by r -index from $O(r)$ to $O(\min(r, n/s))$. Subsampling is performed by iterating over each run sample t_i for $i = 2, \dots, r - 1$ and checking whether the distance between the run samples t_{i+1} and t_{i-1} is at most s – that is, whether $t_{i+1} - t_{i-1} \leq s$. If the condition is met, then the run sample t_i is removed from the data structures in the r -index that stores information about it. In this way, all run samples are removed so that if the text were segmented into blocks of size $s + 1$, there would be no more than two run samples in each block. The sr -index additionally introduces a bitvector of size $O(r)$ bits, marking each removed run sample with a 1. The space overhead of this bitvector is negligible compared to the savings gained by removing entries from the r -index data structures, as it is stored in bits rather than full words. Traditionally, the r -index relies on storing SA values at every run boundary to support `locate` queries. As a result, removing some of these run samples requires a modification to the way `locate` queries are performed. In particular, the process of maintaining a toehold SA value for every interval and the process of finding the SA values of all positions in a final BWT interval changes. For example, computing the SA value for the last position of a run is trivial if the run sample for the run was not removed by sub-sampling. However, in the case where the sample was removed, then in order to find the SA for the run boundary the LF-mapping must be applied recursively until a run sample that was not removed is found. The number of steps needed to find a sample is bounded by s , as this is guaranteed by the subsampling process.

The sr -index takes $O(m + s)$ time to complete the backward search of a pattern of length m , rather than the expected $O(m \cdot s)$ time. The authors managed to do this by taking advantage of the fact that the only SA values needed are the values of the final interval of the search. To do this, the authors keep track of the last time that the next character in the backward search did not match the character at the position in the BWT of the currently known SA value, i.e., $\text{BWT}[p] \neq P[i]$ as shown in Subsection 3.2. Then, the final SA value can be computed by subtracting the number of LF steps that have occurred since this mismatch from the SA value for this mismatched interval. The remaining step is to find the SA value for the mismatched interval, which can be located in at most s LF steps, resulting in a query time of $O(m + s)$. To `locate` all occurrences, the process involves breaking the final interval into maximal runs and using a single LF-mapping per run to calculate the LF-mapping for every position within the run, as the results are contiguous for positions in the same run. From the subsampling process, all runs will be completed in at most s steps, at which point the SA values can be found by applying ϕ to the LF-mapping results found earlier, given the location of all occurrences. The `locate` query therefore takes $O((m + s \cdot \text{occ}) \log \log(n/r))$ time.

As the sampling distance increases, Cobas et al. showed that the space required by sr -index reduces but the query times jump to at least three times the r -index query times. The authors also compared the sr -index with Lempel-Ziv-based indexes. For most data sets, the Lempel-Ziv indexes required less space than sr -index except for a few cases where sr -index took less space when using the largest sampling distance. For every data set, sr -index was significantly faster in querying than the Lempel-Ziv indexes.

3.4 Sampling the Suffix Array: Access to the Suffix Array via ϕ^{-1} -forest

The r -index achieves significant space savings by leveraging ϕ and ϕ^{-1} to minimize the number of SA values that must be stored. However, in practice, this method of accessing SA is slow because it requires the SA value to be computed for every position between the

desired position and the nearest run boundary position. This led to the development of the ϕ^{-1} forest by Boucher et al. [6] that aims to improve the time required to access the SA by providing random access to the SA.

The ϕ^{-1} forest, which works analogously for ϕ , exploits the iterative nature of ϕ^{-1} to compute random access to SA. The primary idea behind this auxiliary data structure relates to the toehold lemma 7. In particular, $\text{SA}[i+1] - \text{SA}[i]$, or the difference in offsets of two consecutive lexicographical suffixes, does not change as we iteratively set $j = i$, when $\text{SA}[j] = \text{SA}[i] - 1$ and $\text{BWT}[i] = \text{BWT}[i+1]$ for the current j . Consequently, given any text position, it is possible to compute successive values of ϕ^{-1} by iterating over indices and applying the known differences between consecutive SA values, until reaching a run boundary where $\text{BWT}[i] \neq \text{BWT}[i+1]$. Furthermore, when we have a list of SA values at run ends, $\mathcal{E}$, and at run starts, $\mathcal{S}$, ϕ^{-1} can be easily determined for each value in $\mathcal{E}$, since each occurs before some value in $\mathcal{S}$. Using these concepts, ϕ^{-1} can be redefined as follows.

► **Definition 9.** *Given a text position $i \in [1..n]$ and a predecessor structure pred , let $s = \text{pred}_{\mathcal{E}}(i) \in \mathcal{E}$ and assume that $\phi^{-1}(s)$ is known. Then, $\phi^{-1}(i) = \phi^{-1}(s) + (i - s)$.*

Using this idea, the ϕ^{-1} -graph is constructed with the nodes being the values of $\mathcal{E}$ and edges between nodes occurring when one node is the predecessor of the run following another node (i.e., $\overline{v_i v_j}$ exists when $\text{pred}_{\mathcal{E}}(\mathcal{S}[i+1]) = \mathcal{E}[j]$). Each edge is also labeled with a cost, c , and a limit, l , which represent the distances from the positions where a change in predecessor occurs. Each cost is defined by the number of steps the start of the next run takes from its predecessor, or $\mathcal{S}[i+1] - \text{pred}_{\mathcal{E}}(i+1)$, and each limit is defined by the distance from the next run, or $\text{succ}_{\mathcal{E}}(\mathcal{E}[i] + 1) - \mathcal{E}[i]$. An important property of this graph is that every node has a single outgoing edge because the predecessor function is unique, which means that there is no ambiguity in the path to follow when traversing the graph. Boucher et al. show that by following this graph and utilizing the cost and limit values along the edges between nodes on the path, it is possible to determine ϕ^{-1} for any given position in the text. Assume that we are given a SA value, $\text{SA}[s]$, then with the ϕ^{-1} graph, a total of m iterations of ϕ^{-1} can be calculated by traversing a m -length path starting from v_i , where $v_i = \mathcal{E}[i] = \text{pred}_{\mathcal{E}}(\text{SA}[s])$. The formula is as follows: $\phi^{-m}(\text{SA}[s]) = \mathcal{S}[i_m + 1] = \mathcal{E}[i_m] + \sum_{k=1}^{m-1} c_{i_k} + c_0$, where $c_0 = \text{SA}[s] - \mathcal{E}[i]$, as long as the requirement is met that $\sum_{k=1}^{j-1} c_{i_k} + c_0 < l_j$ from $j = 2, \dots, m-1$, or the sum of costs is less than the limit at every step of the path. This condition determines the length m of the path and also maintains the necessary condition that every node on the path is the predecessor of the SA value found by the previous node.

The ϕ^{-1} -graph is then used to construct the ϕ^{-1} -forest, which allows iterations of ϕ^{-1} to be skipped during the process of finding a missing SA sample in a `locate` query. To do this, long non-overlapping paths from the ϕ^{-1} -graph are extracted and a balanced binary search tree is built on each path such that given $\text{SA}[s]$ it is easy to find the largest index in the path for which $\phi^{-m}(\text{SA}[s])$ can be computed following the limit requirement laid out above. In the balanced binary search tree, each node stores a cost and limit value c and l , respectively. The leaves of the tree are the edges in the path that the tree represents, and therefore their values are easily determined. The internal nodes then represent their cost as the sum of the cost of their two children and their limit as the minimum between the limit of the left child and the limit of the right child minus the cost of the left child, or $\min(l_u, l_v - c_u)$. The tree can then be queried with a given SA sample, $\text{SA}[s]$, by first finding the predecessor $\mathcal{E}[i] = \text{pred}_{\mathcal{E}}(\text{SA}[s])$ to compute $c_0 = \text{SA}[s] - \mathcal{E}[i]$ and determine the leaf v that corresponds to the edge e_{ij} , or the outgoing edge of $\mathcal{E}[i]$. The goal of the query is to find the largest k such that $\sum_{x=j}^{m-1} c_{i_x} + c_0$ for all $m = j, \dots, k-1$. The algorithm proceeds in two phases: the

first involves traversing the tree from v_i and the second involves descending the tree to a leaf node that represents k . This *fast-forward* query can also be trivially extended to include a constraint, d , such that the result of the query is the largest $k \leq d$. This constraint is important when considering realistic applications in which a specific value within a certain distance from $\text{SA}[s]$ is desired.

The ϕ^{-1} -forest is composed of several of these binary trees, hence its name. To query the forest, with a given SA position, i , the first step is again to determine the predecessor at position, j , and then set $d = i - j$ and $c_0 = \text{SA}[j] - \text{pred}_{\mathcal{E}}(\text{SA}[j])$ and $v = e_j$. The trees in the ϕ^{-1} forest are then searched to find one that contains v . If none is found, ϕ^{-1} is computed normally, v is updated to the new node, d is decremented by one, and the trees are searched again for the new v . If v is found in a tree, then the tree is queried as described above and from the result v is updated to the new node, d is subtracted from the number of edges traversed in the query, and the trees are searched again for the new v . This process continues until $d = 0$. In this way, $\text{SA}[i]$ can be computed in $O((i - j)(\log \log(n/r) + \log r))$ time with $O(r)$ additional space to the r -index.

Boucher et al. used their ϕ^{-1} -forest to augment the r -index and evaluated its performance against traditional r -index in both the Pizza&Chili repetitive corpus [33] and data sets comprising an increasing number of Chromosome 19 sequences. Their results showed comparable SA access times to the traditional r -index on the Chromosome 19 datasets, and up to a sixfold speedup on the Pizza&Chili datasets. Although this improvement came at the cost of slightly increased memory usage compared to r -index, it marked the first method to improve the calculation of Φ .

4 Acceleration of FM-index via Phrase-Based Techniques

Hong et al. [17] point out that the FM-index can process a `count` query for a pattern P of length m in $O(m)$ time with close to m random accesses to the wavelet trees built on the BWT. In practice, these random accesses tend to be a bottleneck compared to actually using the results of `count` queries since each random access can result in a cache miss due to the poor cache locality of the underlying data structures. The idea was then to reduce the number of random accesses by using a word-based approach where multiple characters are grouped together in the text and in the query pattern. By doing this, the cost of a single backward search step would result in the matching of multiple characters. This reduction in backwards search steps does not resolve the issue of poor cache locality but attempts to mitigate it.

4.1 Grammar Compression for Accelerating FM-Index Backward Search: FIGISS

Deng et al. [10] introduced FM-Indexing Grammars Induced by Suffix Sorting (FIGISS), which aims to accelerate the FM-index by enhancing the backward search step using grammar compression. FIGISS uses a context-free grammar which we briefly define as follows. A context-free grammar is a formal system of production rules that recursively replace terminal and nonterminal symbols with a nonterminal symbol. This is formalized as grammar $G = (V, \Sigma, R, S)$, where V is the set of nonterminal symbols, Σ is a set of terminal symbols, R is the set of production rules that take the form $A \rightarrow \alpha$ with $A \in V$ and $\alpha \in (\Sigma \cup V)^*$, and S is the first rule $S \in V$. Their method uses Grammar Compression by Induced Suffix Sorting, known as GCIS [32], which creates a leftmost S -type suffixes (LMS) factorization of an input text T using the linear-time SAIS algorithm [31]. This algorithm works by first

padding the left and right of the input with characters that are lexicographically smaller than all the other characters in the string. Then, it labels each suffix of the input as an S -type suffix or an L -type suffix. This is done by creating a string the same size as the input, iterating through every position i of the input, and adding an S to the string if the suffix starting at that position is lexicographically smaller than the suffix starting at the following position, $i + 1$, or by adding an L otherwise. Suffixes that start at $T[0]$ and $T[n + 1]$ are also labeled as S -type suffixes. Afterwards, all left-most S -type (LMS) suffixes are defined as every S -type suffix that occurs right after an L -type suffix, where the suffix starting at $T[0]$ is also considered an LMS suffix. The LMS factorization is formed by taking each substring of the created string that starts at one LMS suffix and ends right before the next LMS suffix. Note that the first and last positions of the created string are not included in any substring since they corresponded to the padding characters. Then, each LMS substring is replaced with a nonterminal symbol producing the grammar that is used by FIGGIS.

To perform pattern matching, FIGGIS constructs the grammar, $\mathcal{G}_T$, on the input text T , then constructs the grammar, $\mathcal{G}_P$, on the query pattern P , and finally matches P to T by matching their nonterminals. When constructing $\mathcal{G}_P$, the same nonterminal as $\mathcal{G}_T$ is used when the right-hand side of an LMS substring in $T^{(1)}$, the LMS factorization of T , matches the right-hand side of an LMS substring in $P^{(1)}$, the LMS factorization of P . The idea of pattern matching in FIGGIS is built on finding a *core*, which is a maximal length substring in $P^{(1)}$ that is contained in $T^{(1)}$ for every occurrence of P in T . Note that the first and last nonterminals $P^{(1)}$ cannot be part of the *core* because the substrings those nonterminals represent do not appear in the same form in $T^{(1)}$. To find this *core*, FIGGIS constructs a BWT on $T^{(1)}$, represented as a wavelet tree that is constructed using the colexicographic ranking of the nonterminals. Then, a backward search of $p - 2$ steps is done, where p is the number of nonterminals in $P^{(1)}$. The resulting interval represents all occurrences of $P^{(1)}[1..p - 1]$ to be found in $T^{(1)}$, which corresponds to all occurrences of $P[1..|P| - 1]$ in T . To then find all occurrences including the last nonterminal, $P^{(1)}[p]$, the backward search is started from the intervals defined by any nonterminal of $T^{(1)}$ that contains the LMS substring represented by $P^{(1)}[p]$ as a prefix rather than starting with the entire range of the BWT. In practice, Deng et al. limit the maximum possible size of any LMS substring with a parameter to improve performance and include a generalized suffix tree on the right-hand side of nonterminals to support pattern matching for queries shorter than the maximum length parameter.

Deng et al. evaluated their implementation with different aggregation parameters against RLFM using datasets from the Pizza&Chili repetitive corpus, a data set of 15 Chromosome 19 sequences and a data set of randomly generated strings. FIGGIS requires less space than RLFM for every data set, and for every parameter tested, the space decreases as the parameter increases. When answering `count` queries, FIGGIS is faster than RLFM for some aggregation parameters when the pattern length is particularly long, but otherwise it is slower than RLFM. Finally, FIGGIS requires less total memory during index construction and less final index space but takes significantly more time to construct.

4.2 Prefix-Free Parsing for Enhanced FM-Index Performance: PFP-FM

One of the main limitations of FIGGIS was that a single round of LMS factorization led to short phrases that only helped improve the efficiency of `count` queries for particularly long patterns. This is because short phrases do not provide significant speedups from the reduction of backwards search steps compared to the increased overhead required unless the query pattern is long. PFP-FM, an accelerated FM-index, developed by Hong et al. [17]

similarly accelerates the `count` query of the FM-index, but allows for acceleration across a wider range of pattern lengths. This is because PFP-FM uses Prefix-Free Parsing (PFP) which uses user-defined parameters to change the length of the phrases that are created. This allows for longer phrases, which provides a greater reduction in the number of backwards search steps than shorter phrases offer. A detailed description of how PFP works can be found in the preliminaries section 2.7.

The index constructed by PFP-FM consists of the FM-index of T , the FM-index of the parse $\mathcal{P}$, a bitvector $\mathcal{B}$, a hash function h , and a mapping $\mathcal{M}$. $\mathcal{B}$ has the same length as the BWT and stores a 1 for every position for which a rotation of the input starting at that position is prefixed by a trigger string. The hash function h hashes the phrases, and the map $\mathcal{M}$ is used to map the hash of each phrase to the lexicographic rank of the phrase. Hong et al. note that with a sufficiently large hash function, it is possible to store $\mathcal{M}$ in space proportional to the number of phrases while maintaining a negligible probability of a hash collision. This allows the hash function and the map to effectively replace the dictionary $\mathcal{D}$ of the input. Therefore, in order to check if a phrase occurs in the dictionary of the input, the phrase is first hashed and then the map $\mathcal{M}$ is queried with the phrase hash. A null return from the map means that the phrase does not occur in the input.

To perform a `count` query, PFP-FM first parses the pattern P , the same way as the input was parsed using the same rolling hash function. The main difference is that the pattern does not have characters appended to it and is not assumed to be circular, which means that the pattern will not necessarily start or end with a trigger string. Therefore, when P is parsed, there will be up to three resulting components: a prefix that ends at the first trigger string in P , a sequence of complete phrases that are in P , and a suffix that starts at the last trigger string in P . First, if there is a suffix component, it is determined whether that suffix occurs in T by performing a backward search for it in the FM-index of T . If no BWT range is identified, or if a BWT range is identified and the parsed P consists solely of the suffix, then the query process is considered complete. Otherwise, the next step is to determine whether the sequence of phrases in P occurs in T . To do this, the phrases are each hashed with h and the hashes are placed on the map $\mathcal{M}$, which returns NULL if the phrase is not in the input, and their phrase ID if it is. If a NULL value is returned for any phrase, then that phrase and, by extension, P do not occur in T , and the query is done. Otherwise, the sequence of phrases will be encoded in a sequence of phrase IDs. PFP-FM then performs a backward search for this sequence of phrase IDs in the FM-index of $\mathcal{P}$, which means that every backward search step matches a full phrase rather than a single character. The primary acceleration achieved by PFP-FM arises from this mechanism. Continuing with the backward search process, if a BWT range is found and the parsing of P does not have a prefix, the query is complete. Otherwise, the final step is to find the prefix in the parsing of P in T through a backward search in the FM-index of T . In practice, Hong et al. demonstrate that the PFP parameters, w and p , can be adjusted to systematically optimize the performance gains achieved by PFP-FM, as these parameters directly influence the average phrase length.

The authors performed experiments that compared PFP-FM with FIGISS [10], the standard RLFM[26, 22], RLCSA[37], Bowtie[21], and Bowtie2[20] on increasing size SARS-CoV-2 datasets and the human genome GRCH38. In the SARS-CoV-2 datasets, Hong et al. report that PFP-FM requires less cumulative time to complete 1,000 `count` queries on patterns of increasing length than all other competing methods. PFP-FM completes queries at least 451% faster than the second-fastest method, FIGISS. The results were the same on the GRCH38 data set, where PFP-FM was between 3.9 and 7.1 times faster than RLCSA, the second fastest method. This improvement comes at the cost of an increase in index size and memory required during construction.

5 Acceleration via Improving Locality

As shown above, the FM-index and the r -index require several different data structures to support the **count** and **locate** queries. These data structures can become quite large as the size of the input increases, and each must be accessed in succession during query processing. It is difficult to keep all the necessary memory addresses from all different data structures cached, which results in a frequent number of cache misses. This in turn results in slower queries with higher variability. The problem then is to improve the locality of the data in the data structures, such that all the necessary information for a query can be cached together.

5.1 Move data structure

Nishimoto and Tabei [30] introduced the move data structure in 2021. The primary contribution of their paper was to introduce, *OptBWTR*, which is the first string index capable of performing **locate**, **count**, and decompression in $O(r)$ words of space and optimal time. It also supports extract queries in optimal time with $O(r + b)$ words of space, where b is some sampling parameter, and prefix search in optimal time with $O(r' + d)$ words of space, where r' is the number of runs in the RLBWT of the input and d is the size of the set of strings used in the prefix search. This outperforms the r -index, which can attain $O(r)$ words of space or optimal query times, but not both simultaneously. Additionally, the r -index does not support anything other than **locate** and **count** queries. The index of Nishimoto and Tabei achieves this result through the use of the move data structure, which enables the LF and ϕ^{-1} functions – both essential for **count** and **locate** queries – to be computed in $O(r)$ words of space and constant time. The previous best space-time tradeoff possible for these functions was in the r -index, which takes $O(m \log \log(n/r))$ time for **count** queries and $O((m + occ) \log \log(n/r))$ time for **locate** queries with $O(r)$ words of space[15].

The primary idea behind the move data structure is that of disjoint interval sequences and **move** queries. A disjoint interval sequence is a way to turn a function's domain and range into a set of non-overlapping input intervals and a set of non-overlapping output intervals. A **move** query takes a position i from the function's domain and the input interval that contains this position, x , and returns an i' and x' such that i' is a new position in the function's domain and x' is the input interval that contains this new position. The way in which i' is determined is by computing $i' = q_x + (i - p_x)$, where p_x and q_x are the start of the x -th input and output interval, respectively. This defines a bijective function that is related in its form to both the LF and the ϕ^{-1} functions. This is significant, as Nishimoto and Tabei demonstrate that dividing the interval sequences to obtain a balanced interval sequence allows **move** queries to be answered in constant time when the move structure is built on such sequences. Since LF and ϕ^{-1} have the same form as the bijective function defined by a disjoint interval sequence and a **move** query, then by building a move structure on the balanced interval sequence of these functions, it is possible to compute them in constant time.

A move data structure is composed of two arrays, **pair** and **index**. The **pair** stores all pairs from the original interval sequence. In the move data structure for the LF function, these pairs are defined as the BWT positions of all run heads and the LF mapping of these positions. The **index** array stores the index of the interval that contains the second value of the i -th pair. In terms of the LF move structure, this stores the index of the BWT run that contains the LF mapping of the i -th run head. For simplicity, we will now assume that the move structure is a table for which at every position i all the information that would be contained in both arrays is stored. A **move** query on the LF move data structure of some

position i known to be in the input interval x is as follows. First, the LF-mapping of i , let us say i' , is determined using the information from the x -th entry of the table by computing $i' = q_x + (i - p_x)$. Then, a linear search is conducted on the table to find the input interval containing i' , let us say x' , by starting at the input interval that contains the LF-mapping of the start of the x -th run, which is stored by the table. The final step is to return (i', x') for use in future `move` queries that would be necessary in any other backward search. The second step seems like it could be linear in the number of input intervals that need to be searched, but Nishimoto and Tabei established that the number of intervals searched will be constant because the move structure is constructed on a balanced interval sequence.

5.2 RLBWT Tricks

In their paper, RLBWT Tricks, Brown et al. [7] implement the move structure defined by Nishimoto and Tabei, which had yet to be implemented, and refine their approach for practical applications.

First, Brown et al. modified the move table introduced by Nishimoto and Tabei by replacing absolute indices with run-relative start indices. As a result, the table representation requires one fewer column of integers. This change preserves the ability to compute the LF-mapping and ϕ efficiently, but requires an additional sparse bitvector over BWT to mark run starts. This bitvector is used to convert run/offset pairs into absolute text positions at the end of the backward search. Each row in the table is composed of: the character of the run, the length of the run, the index of the run, the LF-mapping of the position at the start of the run, the index of the run that contains the LF-mapping, and the offset into that run it occurs. The main modification that Brown et al. do to the move data structure is to compress the table. Several different methods are presented, implemented, and compared to achieve this. Primarily, the table is partitioned into blocks of fixed size B , allowing each block to be easily loaded into the cache and taking advantage of the inherent locality of the tables for speed-ups. The fixed sizing specifically has the added benefit of positions within the blocks being mapped to easily. Additionally, by using uncompressed bitvectors as a wavelet tree to store the run characters in the block rank, and select queries can be done fast. The authors also augment each block with the position of the first run before and after the current block for each character to allow for quick transitions into other blocks while scanning.

To compress the run-length and offset columns, the authors use directly addressable codes (DACs), which provide a hierarchical encoding of integers [28]. Three different methods are presented to compress the LF-mapping columns. For two of these methods, the property that the LF-mapping values for multiple runs of the same character in a block form a non-decreasing subsequence is exploited. This enables the creation of sublists for each character, which can be represented using partial differences. One method stores these partial differences using DACs and samples the list of differences. The second method represents the partial differences as an uncompressed bitvector, where the number of 0's before each 1 encodes a difference value. For example, to retrieve the fifth value in the difference list, a select query is performed on the bitvector to `locate` the fifth 1. The third method omits partial differences entirely and instead samples the list of differences while applying linear interpolation to recover missing values. Rather than using a single bitvector to represent run head indices, an approach more suitable for large alphabets, a separate uncompressed bitvector is maintained for each character. These per-character bitvectors support faster rank and select queries compared to wavelet trees, at the cost of increased space per character, which is acceptable for data such as DNA that has a small alphabet. Without having implemented the second

half of the Nishimoto and Tabei paper, their implementations rely on sequential scans of the table, which takes $\Omega(r)$ time in the worst case. To combat against large sequential scans, the authors additionally implemented a way to reduce the total number of runs by splitting large runs by some maximum run length during construction.

The implementation developed by Brown et al. is designed to efficiently support `count` queries. Multiple variations of their approach are evaluated, both internally and in comparison to established alternatives such as *r*-index and RLCSA. The evaluation is carried out on datasets of increasing size, specifically Chromosome 19 and SARS-CoV-2, using randomly selected substrings of varying lengths as query input. Among the tested methods, the variant that employs a bitvector to encode the partial differences list of the LF-mapping values – referred to as *lookup-bv* – demonstrates superior performance relative to all other approaches. For shorter query lengths, the standard *lookup-bv* implementation exhibits the best performance, while for longer queries, a variant that integrates *lookup-bv* with run-splitting techniques proves to be the most effective. In all data sets, the best performing variant of Brown et al. implementation achieves execution speeds comparable to or exceeding those of *r*-index, although with higher space consumption. Although it is slower than RLCSA, it compensates for it by requiring less space. Overall, the approach proposed by Brown et al. offers a balanced trade-off between time and space efficiency, positioning it between the two competing methods.

5.3 Movi: A fast and cache-efficient full-text pangenome index

Movi, developed by Zakeri et al. [40], is the first full-text pangenome index to utilize Nishimoto and Tabei's move data structure and builds on the work of Brown et al. as well. Movi takes advantage of the locality of reference of the move data structure to reduce the frequency of cache misses and employs a parallelization strategy to hide the latency of any cache misses that do occur. During LF-mapping steps, the move table stores the index of the run that contains the LF-mapping of the current run head. This other run is usually in a different block of the table, and, as such, accessing it results in a cache miss as the new block is retrieved. To hide this, Zakeri et al. use a memory prefetch command and work on multiple `count` queries at the same time. In this way, while one of the `count` queries is waiting for a block to be loaded into the cache, work is still being done on another `count` query, and the latency is not felt. Hiding this latency is crucial as the jump to a run in another block can occur every iteration of the backward search.

During the backward search, we would normally want to use the LF-mapping of the current BWT range indices to find the next BWT range indices. In the move table, we instead use the stored value that indicates the index of the run that contains the LF-mapping of the current run head. However, the BWT index in the current backward search step does not always correspond to a run head, so the run that contains the LF-mapping of the current run head may not contain the LF-mapping of the current BWT index. In order to find the appropriate run, a sequential search through the runs in the table is necessary. Movi provides two different methods for handling the search. The first is a sequential search through the table that checks runs of characters that match the character of the LF-mapping we are looking for to try and find the run that contains this LF-mapping. The second is a constant-time method in which pointers are stored for each character that point to the indices of runs of the character. The locality of reference allows for the sequential search to be practically inexpensive, especially when compared to a cache failure, but the constant-time method is the faster of the two, as the sequential search still has an unpredictable number of memory accesses. However, the constant-time method does have the added cost of being needed to store the pointers.

The authors compare the backward search capabilities of Movi's two methods against the r -index using 10 million simulated 150 bp reads of the seven bacterial species in the Zymo community. The results show that both Movi methods were around 16 times faster than the r -index in completing the query, but required three times as much space in the default method (using sequential search) and 11 times as much space in the constant method (storing pointers).

5.4 Move-r: Optimizing the r -index

Move-r is a static text index created by Bertram et al. [4] that incorporates the move data structure described in 5.1 with the r -index described in Subsection 3.2. This allows the Move-r index to be practically faster than other r -index implementations with a small increase in the space requirement. This improvement comes from the ability of the Move-r index to use the move data structure to compute LF and ϕ^{-1} in $O(1)$ through move queries.

One of the primary contributions is the inclusion of an algorithm to balance a disjoint interval sequence on which the move data structure is built. Having a balanced disjoint interval sequence is essential to the proof that allows `move` queries to be answered in constant time. The proposed algorithm works by using two balanced search trees (BST), one for the input disjoint intervals and one for output disjoint intervals. It iterates through every output interval in order and determines whether it is an a -heavy interval by checking if it contains $\geq 2a$ input intervals, where a is some balancing parameter. If it does, then the output interval and related input interval are split, producing an a -balanced interval and another interval that is not necessarily balanced. The algorithm repeats this process until there are only a -balanced intervals in the binary search trees. Each step takes $O(\log k)$ time for BST updates, where k is the number of pairs of intervals in the disjoint interval sequence, and $O(a)$ time to determine if an interval is a -heavy. The authors show that there are at most $\frac{k}{a-1}$ splits, so their algorithm takes $O(k + \frac{k}{a} \log k)$ time and $O(1)$ additional space. The algorithm of Nishimoto and Tabei for constructing the balanced disjoint interval sequence takes $O(k \log k)$ time and $O(k)$ space. The authors also provide a practically optimized `locate` algorithm with an asymptotically worse runtime compared to the `locate` algorithm provided by Nishimoto and Tabei [30]. Their algorithm saves up to $3m$ cache misses, where m is the length of the query pattern, by precomputing and storing the nearest SA value for each run head, which allows constant time lookup during the query rather than accessing multiple different data structures in each iteration of the backward search as is done by *OptBWTR*.

Move-r is compared against the r -index [15], online-rlbwt [2], rcomp [29], lookup-bv [7], and block-rlbwt [11] using Pizza&Chili [33], concatenated samples of chr19, concatenated samples of the SARS-CoV-2 genome, and other repetitive texts. During construction of the index, block-rlbwt tended to take the least amount of time and the least amount of peak memory consumption, but move-r was 0.9-2 times as fast as the other methods, while requiring 0.33-1 times as much peak memory usage. For count queries, block-rlbwt offered the best trade-off between query throughput and index size, and move-r consistently had the highest query throughput but was the third largest index. For locate queries, move-r was 0.8-2.5 times as large and 2-35 times as fast.

6 Future Work

Despite the extensive work that has been done on the FM-index and the field of compressed full-text indexes, there are still future directions to be explored to improve further. One such area is the move data structure by Nishimoto and Tabei [30] and their full-text index.

Current research [7, 40, 4] has led to the implementation of the move data structure, which has been shown to be quicker than the r -index but requires more space. There is still room for improvement than can bring the space requirements closer to the r -index while maintaining the practical query times. Nishimoto and Tabei also provide the theoretical framework for a backward search algorithm and a full-text index based on the move data structure, which has yet to be implemented. Another avenue of work is to dynamically update the move data structure so that new datasets can be added dynamically. Currently, when a new string is being added to the move data structure, it has to be recreated entirely, unlike other indexes that can dynamically change to include new inputs.

Another direction of improvement relates to the ϕ and ϕ^{-1} functions. Traditionally, using these functions requires an iteration for every position between the desired SA value position and the known SA value position. The ϕ^{-1} -forest [6] instead allows some random access to SA using a forest built on the ϕ^{-1} values. The results showed significantly faster query times on small repetitive texts and comparable query times on larger repetitive texts, such as the human genome, at the cost of increased space. Future work in this area could look to improve the random access to ϕ and ϕ^{-1} so that the increased space requirements are reduced and that the performance on SA queries improves over the r -index on larger datasets.

7 Conclusion

In conclusion, the FM-index has profoundly influenced the fields of computational biology, data compression, and text retrieval since its introduction, offering a powerful tool for efficient and compact string indexing. Although the classical FM-index provides an elegant balance between space and time efficiency, the rapid growth in data size and complexity has highlighted its limitations, particularly with highly repetitive and large-scale datasets. This has spurred significant advancements, including the development of run-length encoding, grammar-based compression, and prefix-free parsing techniques, which enhance the FM-index's scalability and performance.

Despite these achievements, the field continues to face challenges that warrant further exploration. Bridging the gap between theoretical innovations and practical implementations remains critical, as is optimizing the balance between memory consumption and query efficiency. Emerging areas such as dynamic FM-indexes, hardware acceleration, and cross-domain applications demonstrate the ongoing evolution of this foundational data structure. As data volumes continue to grow exponentially, the importance of efficient self-indexes, like the FM-index, will only increase. Building upon the innovations reviewed in this paper, researchers have the opportunity to further enhance the capabilities of FM-index variants, addressing contemporary challenges while opening new avenues for applications across diverse domains.

References

- 1 Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer, 1985.
- 2 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r -index. *Theoretical Computer Science*, 812:96–108, April 2020. doi:10.1016/j.tcs.2019.08.005.
- 3 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Transactions on Algorithms (TALG)*, 16(2):1–54, 2020. doi:10.1145/3381417.

- 4 Nico Bertram, Johannes Fischer, and Lukas Nalbach. Move-r: optimizing the r-index. In *22nd International Symposium on Experimental Algorithms (SEA 2024)*, pages 1–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.
- 5 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Molecular Biology*, 14(1):13:1–13:15, 2019. doi:10.1186/S13015-019-0148-5.
- 6 Christina Boucher, Dominik Köppl, Herman Perera, and Massimiliano Rossi. Accessing the suffix array via ϕ -1-forest. In *International Symposium on String Processing and Information Retrieval*, pages 86–98. Springer, 2022. doi:10.1007/978-3-031-20643-6_7.
- 7 Nathaniel K Brown, Travis Gagie, and Massimiliano Rossi. Rlbwt tricks. *LIPICs: Leibniz international proceedings in informatics*, 233:16, 2022.
- 8 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, May 1994.
- 9 Dustin Cobas, Travis Gagie, and Gonzalo Navarro. A fast and small subsampled r-index. *arXiv preprint arXiv:2103.15329*, 2021. arXiv:2103.15329.
- 10 J. J. Deng, W. K. Hon, D. Köppl, and K. Sadakane. Fm-indexing grammars induced by suffix sorting for long patterns. In *Proceedings of the Data Compression Conference (DCC)*, pages 63–72. IEEE, 2022. doi:10.1109/DCC52660.2022.00014.
- 11 Diego Díaz-Domínguez, Saska Dönges, Simon J Puglisi, and Leena Salmela. Simple runs-bounded fm-index designs are fast. In *21st International Symposium on Experimental Algorithms (SEA 2023)*, pages 7–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.
- 12 Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM (JACM)*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.
- 13 P Ferragina and G Manzini. Indexing Compressed Text. *Journal of the ACM*, 52:552–581, 2005. doi:10.1145/1082036.1082039.
- 14 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *the Proceedings of the Foundations of Computer Science (FOCS)*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.
- 15 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *Journal of the ACM*, 67(1):1–54, 2020. doi:10.1145/3375890.
- 16 Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, et al. High-order entropy-compressed text indexes. In *SODA*, volume 3, pages 841–850, 2003.
- 17 Aaron Hong, Marco Oliva, Dominik Köppl, Hideo Bannai, Christina Boucher, and Travis Gagie. Pfp-fm: an accelerated fm-index. *Algorithms for Molecular Biology*, 19(1):15, 2024. doi:10.1186/S13015-024-00260-8.
- 18 Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi. Permuted longest-common-prefix array. In *Proceedings of the Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2009. doi:10.1007/978-3-642-02441-2_17.
- 19 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.
- 20 B Langmead and S L Salzberg. Fast gapped-read alignment with {B}owtie 2. *Nature Methods*, 9(4):357–359, 2012.
- 21 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(3):R25–R25, 2009. Publisher: BioMed Central. doi:10.1186/gb-2009-10-3-r25.
- 22 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56. Springer, 2005. doi:10.1007/11496656_5.

- 23 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 24 Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.
- 25 J Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 408–424. SIAM, 2017. doi:10.1137/1.9781611974782.26.
- 26 V Mäkinen and G Navarro. Run-length FM-index. *the Proceedings of the DIMACS Workshop: The Burrows-Wheeler Transform: Ten Years Later*, pages 17–19, 2004.
- 27 Gonzalo Navarro. Indexing text using the Ziv–Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004. doi:10.1016/S1570-8667(03)00066-2.
- 28 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.
- 29 Takaaki Nishimoto, Shunsuke Kanda, and Yasuo Tabei. An optimal-time rlbwt construction in bwt-runs bounded space. *arXiv preprint arXiv:2202.07885*, 2022. arXiv:2202.07885.
- 30 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on bwt-runs compressed indexes. *arXiv preprint arXiv:2006.05104*, 2020.
- 31 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE transactions on computers*, 60(10):1471–1484, 2010. doi:10.1109/TC.2010.188.
- 32 Daniel Saad Nogueira Nunes, Felipe Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. A grammar compression algorithm based on induced suffix sorting. In *2018 Data Compression Conference*, pages 42–51. IEEE, 2018. doi:10.1109/DCC.2018.00012.
- 33 Pizza & Chili repetitive corpus. Available at <http://pizzachili.dcc.uchile.cl/rep corpus.html>. Accessed 16 April 2020.
- 34 Alberto Policriti and Nicola Prezza. Lz77 computation based on the run-length encoded bwt. *Algorithmica*, 80(7):1986–2011, 2018. doi:10.1007/S00453-017-0327-Z.
- 35 Nicola Prezza. Compressed computation for text indexing, 2017.
- 36 Kunihiro Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *International symposium on algorithms and computation*, pages 410–421. Springer, 2000. doi:10.1007/3-540-40996-3_35.
- 37 J Sirén. Compressed suffix arrays for massive data. In *International Symposium on String Processing and Information Retrieval*, pages 63–74, 2009.
- 38 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.
- 39 Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11. IEEE, 1973. doi:10.1109/SWAT.1973.13.
- 40 Mohsen Zakeri, Nathaniel K Brown, Omar Y Ahmed, Travis Gagie, and Ben Langmead. Movi: a fast and cache-efficient full-text pangenome index. *iScience*, 27(12), 2024.

Algorithms for Computing Very Large BWTs: a Short Survey

Diego Díaz-Domínguez 

Department of Computer Science, University of Helsinki, Finland

Lavinia Egidi 

Computer Science Institute, DiSIT, University of Eastern Piedmont, Vercelli, Italy

Veronica Guerrini 

Department of Computer Science, University of Pisa, Italy

Felipe A. Louza 

Faculty of Electrical Engineering, Federal University of Uberlândia, Brazil

Giovanna Rosone¹ 

Department of Computer Science, University of Pisa, Italy

Abstract

The Burrows-Wheeler Transform (BWT) is a fundamental string transformation that, although initially introduced for data compression, has been extensively utilized across various domains, including text indexing and pattern matching within large datasets. Although the BWT construction is linear, the constants make the task impractical for large datasets, and as highlighted by Ferragina et al. [26], “to use it, one must first build it!”. Thus, the construction of the BWT remains a significant challenge. For these reasons, during the past three decades there has been a succession of new algorithms for its construction using techniques that work in external memory or that use text compression. In this survey, we revise some of the most important advancements and tools presented in the past years for computing large BWTs exploiting external memory or text compression approaches without using additional information about the data.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Burrows-Wheeler transform, Extended Burrows-Wheeler transform, external memory, text compression, longest common prefix

Digital Object Identifier 10.4230/OASICS.Manzini.2025.7

Funding *Diego Díaz-Domínguez*: Supported by the European Union’s Horizon Europe research and innovation programme under grant agreement No 101060011.

Veronica Guerrini: Supported by the Next Generation EU PNRR MUR M4 C2 Inv 1.5 project ECS00000017 – “THE – Tuscany Health Ecosystem” – Spoke 6 “Precision medicine & personalized healthcare” CUP I53C22000780001.

Felipe A. Louza: Supported by Brazilian agencies CNPq (grants 408314/2023-0 and 311128/2025-4) and FAPEMIG (grant APQ-01217-22).

Giovanna Rosone: Partially supported by the Next Generation EU PNRR MUR M4 C2 Inv 1.5 project ECS00000017 – “THE – Tuscany Health Ecosystem” – Spoke 6 “Precision medicine & personalized healthcare” CUP I53C22000780001, by the MUR PRIN 2022YRB97K PINC and by project “Hub multidisciplinare e interregionale di ricerca e sperimentazione clinica per il contrasto alle pandemie e all’antibioticoresistenza (PAN-HUB)” funded by the Italian Ministry of Health (POS 2014–2020, project ID: T4-AN-07, CUP: I53C22001300001).

¹ corresponding author



1 Introduction

The term “big data” has become widely used, encompassing a variety of data types. Genomic data is one of the most prominent forms of big data, due to its size, complexity, and impact [65]. The advancements in high-throughput sequencing technologies have resulted in an unprecedented expansion of genomic databases, which are expected to soon reach sizes of hundreds of terabytes. The scale and complexity of these datasets present significant challenges in terms of data storage, access, and processing.

One of the primary tasks when working with such large datasets is the construction of efficient indexes, which facilitate fast data retrieval and enable a variety of applications in genomics research. Until the early 2000s, the standard solutions for this task were the suffix array or the suffix tree. However, these approaches quickly become intractable as the text grows. Ferragina and Manzini solved this limitation with their FM-index [27], a succinct self-index² with suffix array functionality. Then, Lam *et al.* [41] noticed that the FM-index is efficient in performing pattern matching of short sequences with few mismatches over large genomes. This functionality led to the creation of popular bioinformatic tools such as **bwa** [46] and **bowtie** [42] based on the FM-index, and since then its use has been extended to other applications such as *de novo* assembly [62], variant detection [43, 60] and sequence compression [35, 61, 33, 31].

The main component of the FM-index is the Burrows-Wheeler transform [12], a string transformation introduced in the Data Compression field by Michael Burrows and David Wheeler in 1994. It is a reversible transformation that rearranges the symbols of a string w over an alphabet Σ , possibly increasing the runs of identical symbols. This structure enables compressing the original string using simple algorithms, such as Run-Length Encoding (RLE) followed by Move-to-Front (MTF) and Huffman coding (as employed in **bzip2** compressor), see [1] for further details.

Since the sequencing process generates a collection of sequences, each representing a read, it introduced challenges in how to represent, process, and analyze these sequence collections. Applying the BWT to each sequence individually could not capture the features among the different sequences, and hence, there emerged a necessity to extend the BWT to sets of strings. Moreover, the need to analyze multiple genomes together has recently emerged to study the complete genetic diversity within populations [18]. Here, we consider the extension of the Burrows-Wheeler transform to a string collection, in which a distinct end-marker symbol is appended to each string (see [14, 15] for a survey about the different BWT variants). When an end-marker symbol is appended to each string, there are no strings in the collection that are prefixed by other strings. Such a transform applied to a string collection is also known in the literature as multi-string BWT³ [23, 8]. Over the years, a multitude of strategies and tools have been introduced to build the BWT and the multi-string BWT, whose efficiency may be dictated by particular needs (e.g. input alphabet, redundant data, available resources, knowledge of a reference string).

In-memory algorithms to compute standard single-string BWTs are linear but do not scale to large inputs with multiple strings as, in those cases, they might not be able to fit everything in RAM (e.g., SAIS [56] and its BWT variant [57]). A possible solution is to keep large portions of data on disk and bring them back to RAM and caches whenever the BWT algorithm requires it. This strategy reduces the space usage in fast memory levels

² A data structure that encodes a text and an index for it in space proportional to the text size

³ In [15], the same transform is called Multidollar-EBWT.

considerably, but comes with a cost: While accessing caches requires nanoseconds in most modern computer architectures, accessing data from disks (aka I/O) requires milliseconds. This difference is the so-called I/O bottleneck, which amounts to a factor of 10^{5-6} [34, 25], and makes it necessary to adapt in-memory algorithms to minimize I/Os. An alternative solution is to keep the data in compressed form to fit more information in less space. This technique reduces the number of I/Os but requires extra time to compress and decompress. In addition, it is necessary to adapt the algorithm to make it compression-aware. In practice, disk-based and compression-aware algorithms are not incompatible and are often combined in the literature.

In this survey, we explore BWT construction algorithms that address the scalability problem by using techniques that work in external memory or by exploiting text compression approaches. We have selected some of the most representative algorithms from these two categories. Those that work in external memory (Section 3) must address the challenge of making the best possible use of the available RAM, minimizing and optimizing operations on external memory, avoiding random disk accesses. Optimal I/O volume when working with a set of strings of total size N is $\mathcal{O}(N)$ bytes which is never achieved to the best of our knowledge. The ordering of suffixes necessary to compute BWTs and suffix arrays is in general carried out with bucket sort algorithms that write data to disk in sequential order, and it is often implemented in a virtual way, ordering indices instead of the actual structures. Some of the algorithms approach the problem by computing partial BWTs, which could be either BWTs of substrings or BWTs of subcollections or BWTs of suffixes of fixed lengths or up to fixed lengths.

We first present **bwt-disk** [26] (Section 3.1), an external memory approach that works on a single input string. We then present **BCR** and **BCRext** [5] (Section 3.2), (semi-)external algorithms, which, although not competitive for collections of long strings stand out due to their simplicity, making them a good choice for sets of strings over any alphabet with limited lengths and for building BWTs with a reduced, possibly minimum, number of runs [13, 15]. In Section 3.3, we present **eGSA** [49], which works best with an amount of RAM significantly larger than the input size and it is extremely efficient, but requires a comparatively large amount of disk space, even though it has good I/O costs. In Section 3.4, we present **eGap** [23], which slightly improves on the I/O costs of **eGSA** and is faster when less RAM is available, also thanks to a specific heuristic. Finally, in Section 3.5, we present **BWT+LCP** [8], which has the same asymptotic complexity as **eGap** with no heuristic and fixed size alphabet.

In Section 4, we consider algorithms of the second class that leverage compression techniques that exploit the high repetitiveness of the input. This way more of the input fits in memory as close as possible to the CPU for faster processing, and repetitions in the input are eliminated. The risk, though, is introducing a significant overhead when strings must be decompressed for access. The art in designing these algorithms is finding compression schemes whose structure actually enhances the BWT computation process instead of hindering it.

We present in Section 4.1 **Big-BWT** [11], which takes as input a single string and performs a pre-processing step that encodes the input building a dictionary and a parse (*prefix-free parsing*). This strategy for building the BWT works well when the dictionary and the parse generated are much smaller than the input string. Finally, in Section 4.2, we present **grlBWT** [22], which joins the two worlds as an algorithm that uses both external memory and compression techniques. It maintains data in compressed form throughout the computation to reduce space usage and repetitions.

2 Preliminaries

Let Σ be a finite ordered alphabet of size σ . A *string* $S = S[1..n]$ over Σ of length $|S| = n$ is a sequence of n characters (or symbols) from Σ , where $S[i]$ denotes the i th character of S , and $S[i..j]$ denotes the substring $S[i] \cdots S[j]$, for $i \leq j$. By convention $S[i..j] = \epsilon$ if $i > j$, where ϵ is the empty string. Any substring of S of the form $S[1..i]$, $1 \leq i \leq n$, is called *prefix*, and any substring of the form $S[i..n]$, $1 \leq i \leq n$, is called *suffix*. Let S be a string of length $n - 1$ over Σ and $\$$ be a special character, which is not in Σ and is lexicographically smaller than any other character in Σ . Throughout this paper, we consider S with the special character (referred to as *end-marker*) appended to it, such that $S[n] = \$$.

For any two strings S and T , we define the *lexicographic order* $S \prec T$ if S is a proper prefix of T , or if there exists an index $1 \leq i \leq \min\{|S|, |T|\}$ such that $S[i] < T[i]$ and for all $j < i$, $S[j] = T[j]$.

We consider $\mathcal{S}$ be a collection of k strings, $\mathcal{S} = \{S_1, \dots, S_k\}$, where $S_j[1..n_j - 1]$ is a string over Σ and $S_j[n_j]$ is the end-marker $\$j$, for any j . The total number of characters in $\mathcal{S}$ is $N = \sum_{j=1}^k n_j$. By definition, the end-markers $\$j$ are not appearing elsewhere in $S_1, \dots, S_k$ and are smaller than any other character in $S_1, \dots, S_k$. In addition, we assume that the end-markers $\$j$ are sorted according to the input strings⁴, i.e. $\$i < \$j$, if $i < j$.

We use the disk model of computation, which has two levels: a small but fast memory of M bits (i.e., RAM) and a slow but unbounded memory (i.e., disk). Given an input of size n , a procedure runs in RAM using words of $\Omega(\log n)$ bits that can be manipulated in constant time. Additionally, the procedure can transfer B bits between memories (I/O operation). We express the space complexity in bits and the I/O cost as the number of data transfers. Additionally, we express the logarithms of base two as $\log$.

Suffix array and LCP. Let S be a string of length $n - 1$ over Σ and $S[n] = \$$. The *suffix array* sa of S [50] stores the permutation of $\{1, \dots, n\}$ listing the starting positions of the suffixes of S in increasing lexicographic order, i.e., $S[\text{sa}[i]..n] \prec S[\text{sa}[i + 1]..n]$, for all $1 \leq i < n$.

The *generalized suffix array* (GSA) of the collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ is the array of N pairs of integers (t, j) , corresponding to the lexicographically sorted suffixes $S_j[t, n_j]$, where $1 \leq t \leq n_j$ and $1 \leq j \leq k$.

The *longest common prefix* array lcp stores the length of the longest common prefix (LCP) between lexicographically consecutive suffixes, i.e., $\text{lcp}[i] = \text{LCP}(S[\text{sa}[i - 1]..n], S[\text{sa}[i]..n])$, for all $1 < i \leq n$, and $\text{lcp}[1] = 0$ for convenience.

Burrows-Wheeler Transform (BWT) and Multi-string BWT. The *Burrows-Wheeler Transform* (BWT) [12] is a reversible text transformation that given as input a string S produces a permutation of its characters: the output string bwt is obtained by concatenating the characters that circularly precede the lexicographically sorted suffixes of S , i.e.

$$\text{bwt}[i] = \begin{cases} S[\text{sa}[i] - 1] & \text{if } \text{sa}[i] > 1, \\ S[n] & \text{if } \text{sa}[i] = 1. \end{cases}$$

The reversibility of the BWT is guaranteed by the following two properties [12]. Let F be the string formed by lexicographically sorting the characters of S ,

- for all $i = 1 \dots n$, the character $F[i]$ circularly follows the character $\text{bwt}[i]$ in the string S ;

⁴ In order not to increase the alphabet size, typically tools use the same character in the BWT output string, such as $\$$, to represent all distinct end-markers. Each end-marker's index can be stored separately.

- for each $c \in \Sigma$, the k th occurrence of c in $\mathbf{bwt}$ corresponds to the k th occurrence of c in F . In particular, if $\mathbf{bwt}[i] = c$, then the position j in F corresponding to that c -occurrence is given by $C[\mathbf{bwt}[i]] + \text{rank}(\mathbf{bwt}[i], i)$, where $C[c]$ is the total number of characters in S that are smaller than c and $\text{rank}(c, i)$ is the number of occurrences of character c in the substring $\mathbf{bwt}[1..i]$.

The function described above that maps character occurrences from $\mathbf{bwt}$ to F is known as *LF-mapping* [27]. The LF-mapping yields a permutation of the positions $1, 2, \dots, n$ which forms a single cycle over all positions.

► **Example 1.** Given the string $S = \text{CATGATGATA\$}$ of length 11, the BWT output string is $\mathbf{bwt} = \text{ATGGC\$TTAAA}$. The LF-mapping yields the following permutation

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 9 & 7 & 8 & 6 & 1 & 10 & 11 & 3 & 4 & 5 \end{pmatrix}.$$

Historically, the BWT was extended to a collection of strings by Mantaci *et al.* [51] (EBWT), where cyclic rotations of the input strings are sorted⁵ without appending any end-marker to the input strings. Here, we consider the generalization that append to each input string in the collection a different end-marker and computes the transformed string by lexicographically sorting the suffixes of the input strings, as done for instance by [5, 24, 22]. Therefore, we define the BWT extended to a collection of strings $\mathbf{bwt}(\mathcal{S})$, *multi-string BWT*, as a permutation of the N characters in $S_1, \dots, S_k$ obtained by concatenating the characters circularly preceding the lexicographically sorted suffixes of $S_1, \dots, S_k$.

Since each S_j is terminated by an end-marker symbol, the $\mathbf{bwt}(\mathcal{S})$ can be defined in terms of the generalized suffix array (GSA). In particular, $\mathbf{gsa}[q] = (t, j)$ if the q -th smallest suffix of the strings in $\mathcal{S}$ is $S_j[t, n_j]$. So, for $i = 1, \dots, N$,

$$\mathbf{bwt}(\mathcal{S})[i] = \begin{cases} S_j[(t-1)] & \text{if } \mathbf{gsa}[i] = (t, j) \text{ with } t > 1, \\ \$_j & \text{if } \mathbf{gsa}[i] = (t, j) \text{ with } t = 1. \end{cases} \quad (1)$$

Equivalently, one can define $\mathbf{bwt}(\mathcal{S})$ in terms of suffix array $\mathbf{sa}_{1\dots k}$ of the concatenated string $S_1 \dots S_n$ (see [23, 24], where it is denoted $\mathbf{bwt}_{1\dots k}$) i.e.

$$\mathbf{bwt}(\mathcal{S})[i] = \begin{cases} S_j[\mathbf{sa}_{1\dots k}[i] - \sum_{h=1}^{j-1} n_h - 1] & \text{if } \sum_{h=1}^{j-1} n_h + 1 < \mathbf{sa}_{1\dots k}[i] \leq \sum_{h=1}^j n_h, \\ S_j[n_j] & \text{if } \mathbf{sa}_{1\dots k}[i] = \sum_{h=1}^{j-1} n_h + 1. \end{cases} \quad (2)$$

Note that both definitions in Equations (1) and (2), as well as the EBWT, are such that each input string cycles on itself. In fact, the two definitions above are equivalent, but we chose to present both of them because they imply different constructions. In presenting each algorithm we will refer to the definition that inspired the algorithm's strategy.

Finally, the multi-string BWT is reversible by definition: one can reconstruct the collection with the strings sorted according to their original order given the fact that the end-markers in $\mathbf{bwt}(\mathcal{S})$ are distinct [5, 6]. In addition, differently from the BWT applied to a single string, the LF-mapping associated with the multi-string BWT defines a permutation of the positions 1 through N that can be decomposed into k cycles, one for each string in the collection.

⁵ In this case, one needs to use the ω -order defined in [51].

► **Example 2.** Let $\mathcal{S}$ be the collection $\{AGCGT\$_1, TCAAC\$_2, CGCAA\$_3\}$. The multi-string BWT according to Equations (1) and (2) is given by $\text{bwt}(\mathcal{S}) = TCAACCA\$_1AGT\$_3GCACG\$_2$. The LF-mapping yields the following permutation

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 17 & 9 & 4 & 5 & 10 & 11 & 6 & 1 & 7 & 14 & 18 & 3 & 15 & 12 & 8 & 13 & 16 & 2 \end{pmatrix},$$

which can be decomposed into 3 disjoint cycles: $(1\ 17\ 16\ 13\ 15\ 8)\ (2\ 9\ 7\ 6\ 11\ 18)\ (3\ 4\ 5\ 10\ 14\ 12)$.

Induced suffix sorting (ISS). Induced suffix sorting (ISS) [40, 37, 56] is a recursive technique that deduces the lexicographical order of suffixes in S starting from a subset of partially sorted suffixes. One of the most popular variants of this idea is SAIS [56], a linear-time algorithm to calculate the suffix array [56, 48, 55, 47] and the BWT [57, 3, 9, 22].

Each recursive level i of SAIS receives a string $S^i[1..n^i]$ ($S^i = S$ and $\Sigma^i = \Sigma$ when $i = 1$) and returns the suffix array sa^i of S^i . The algorithm classifies the positions of S^i into two types: *L-type* or *S-type*. A character $S^i[j]$ is *L-type* if $S^i[j] > S^i[j+1]$ or if $S^i[j] = S^i[j+1]$ and $S^i[j+1]$ is L-type. On the other hand, $S^i[j]$ is *S-type* if $S^i[j] < S^i[i+1]$ or if $S^i[i] = S^i[j+1]$ and $S^i[i+1]$ is S-type. By definition, $S[n] = \$$ is S-type. Furthermore, $S^i[j]$ is *LMS-type* (leftmost S-type) if $S^i[j]$ is S-type and $S^i[j-1]$ is L-type. A substring $S^i[j..j']$ is an *LMS substring* if $S^i[j]$ and $S^i[j']$ are LMS-type.

Every suffix $S^i[j..n]$ whose starting position $S^i[j]$ is LMS-type is also a local minimum $S[j-1..n] \succ S^i[j..n] \prec S^i[j+1..n]$. SAIS exploits this fact and performs three linear-time scans of S^i to induce from $S^i[j]$ a partial sort of the suffixes starting within $S^i[j'..j-1]$, $S^i[j']$ being the preceding LMS-type symbol. In this survey, this process is called *ISS pass*.

For any position $j' \in [1..n]$, let $\text{nextLMS}(j') > j'$ be the smallest integer such that $S^i[\text{nextLMS}(j')]$ is of LMS type, or $\text{nextLMS}(j') = n^i$ if $j' = n^i$ or there are no LMS-type positions after j' . Additionally, let $\mathcal{T}$ be the set of strings over Σ^i that label the sequences $S^i[j'..\text{nextLMS}(j')]$. For two strings $X[1..n_x], Y[1..n_y] \in \mathcal{T}$, their LMS order $\prec_{\text{LMS}}$ is

$$X \prec_{\text{LMS}} Y \iff \begin{cases} n_y < n_x \text{ and } X[1..n_y] = Y, \\ \text{or} \\ X \prec Y. \end{cases} \quad (3)$$

Let $\mathcal{O}_X \subset [1..n]$ be the set of integers such that each $j \in \mathcal{O}_X$ corresponds to some $S^i[j..\text{nextLMS}(j)] = X$. The relation $X \prec_{\text{LMS}} Y$ implies that for any pair $j \in \mathcal{O}_X$ and $j' \in \mathcal{O}_Y$, it holds $S^i[j..n^i] \prec S^i[j'..n^i]$. The ISS pass places the suffixes of S^i starting at positions in $\mathcal{O}_X$ contiguously in the r th range $\text{sa}[s_X..e_X]$ of sa , where r is the $\prec_{\text{LMS}}$ rank of X among the elements of $\mathcal{S}$. Thus, the suffixes in $\text{sa}^i[s_X..e_X]$ are lexicographically smaller than the suffixes in $\text{sa}^i[s_Y..e_Y]$ if $X \prec_{\text{LMS}} Y$. However, this sorting is partial because the relative order of the elements within $\text{sa}^i[s_X..e_X]$ and $\text{sa}^i[s_Y..e_Y]$ is still unknown.

SAIS now replaces the LMS substrings in S^i by their $\prec_{\text{LMS}}$ ranks to create a new string $S^{i+1}[1..n^{i+1}]$, with $n^{i+1} \leq n^i/2$, which passes as input to the next recursive level $i+1$ to calculate sa^{i+1} . When the algorithm returns to level i from level $i+1$, it performs another ISS pass. However, this time it uses the information in sa^{i+1} as the starting point for the induction, and thus the output of the pass becomes sa^i . If $i > 1$, SAIS returns sa^i to the previous level sa^{i-1} , otherwise it finishes.

Each new S^{i+1} is at most half the size of S^i . Consequently, the total cost of the algorithm is $\mathcal{O}(n)$ time and $\mathcal{O}(n \log n)$ bits of working space. SAIS can also be easily adapted to compute the generalized suffix array gsa of a string collection $\mathcal{S}$ [48].

Karp-Rabin fingerprint. The *Karp-Rabin (KR) fingerprints* method [38] is a rolling hash technique that associates strings with integers. The KR fingerprints for all the substrings of length w of a string S of length n can be computed in $\mathcal{O}(n)$ time.

Run-length encoding. A *run* in a string S is a maximal substring consisting of repetitions of only one character, i.e. c^e for some $c \in \Sigma$ and $e \geq 1$. The *run-length encoding (RLE)* of a string S encodes any run c^e in S by a pair (c, e) .

Grammar compression. This technique consists of computing a small context-free grammar that generates only the string $S[1..n]$ [39, 16]. A grammar $\mathcal{G} = (\Sigma, V, \mathcal{R}, I)$ is a tuple where Σ is the alphabet of S , V is a set of nonterminal symbols, $\mathcal{R}$ is a relation $V \times (\Sigma \cup V)^*$ describing the set of rewriting rules, and $I \in V$ is the start symbol. Each nonterminal $b \in V$ has a replacement $X[1..x] \in (\Sigma \cup V)^*$ encoded by a rule $b \rightarrow X[1..x] \in \mathcal{R}$. Given two strings $w_a = A[1..q] \cdot b \cdot B[1..b]$, $w_b = A[1..a] \cdot X[1..x] \cdot B[1..b] \in (\Sigma \cup V)^*$, w_b rewrites w_a (denoted $w_a \Rightarrow w_b$) if $b \rightarrow X[1..x]$ exists in $\mathcal{R}$. The *expansion* $\text{exp}(b) \in \Sigma^*$ of $b \in V$ is the string resulting from successive rewrites $w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_k$ with $w_1 = b$ and $w_k = \text{exp}(b)$. Expanding the start symbol I produces $\text{exp}(I) = S[1..n]$. Grammar compression algorithms create one rewriting rule $b \rightarrow X[1..x]$ for every nonterminal $b \in V$, so there is one possible sequence of rewrites that starts in b and ends in $\text{exp}(b)$.

3 Disk-based strategies

When the data to be indexed exceeds the capacity of internal memory, external-memory algorithms become necessary to handle inputs efficiently. These algorithms are specifically designed to minimize random access and optimize disk I/O operations, as accessing external storage is significantly slower than accessing RAM. They typically rely on techniques such as sequential scans, buffering, and external sorting to process data in manageable chunks while reducing disk seek times. Efficient external-memory algorithms are crucial for indexing massive datasets where internal-memory approaches would be impractical due to space constraints.

3.1 bwt-disk [Ferragina et al., 2012]

The algorithm **bwt-disk**⁶ [26] is an external memory algorithm that constructs the BWT for a (single) input string S of length n , without first building the suffix array for the complete string. It operates using only M words of RAM and n bits of disk space, in addition to the disk space needed to store the input and the output. Moreover, all data on disk is accessed exclusively through sequential scans.

The algorithm partitions the input string $S[1..n]$ into consecutive blocks of length $m = \Theta(M)$, such that $S = S^{[n/m]} S^{[n/m]-1} \dots S^2 S^1$. Each block S^i represents a substring of S with length m , except possibly the last substring $S^{[n/m]}$, which may be shorter if n is not a multiple of m . The **bwt**(S) is computed through $\lceil n/m \rceil$ passes, one for each block, which are processed backwards, from S^1 to $S^{[n/m]}$. The key idea is that, during each pass $h+1$, only the characters of S^{h+1} need to be inserted into **bwt**($S^h \dots S^2 S^1$) to obtain **bwt**($S^{h+1} \dots S^2 S^1$), and the relative order of suffixes from $S^h \dots S^2 S^1$ remains unchanged.

⁶ An implementation is available at <https://people.unipmn.it/manzini/bwtdisk/>

In Example 1, suppose $S = CATGATGATA\$$ is partitioned into blocks of length $m = 4$. Then, we have $S^1 = ATA\$$, $S^2 = GATG$ and $S^3 = CAT$. Initially, $\text{bwt}(S^1) = AT\$A$ is computed in internal memory, and $\text{bwt}(S^2S^1)$ will be obtained by sequentially inserting each character of S^2 into $\text{bwt}(S^1)$.

For each pass $h + 1$ (for $h = 1, 2, \dots, \lceil n/m \rceil - 1$), the algorithm loads from disk $S^{h+1}S^h$ into a string buffer $s[1..2m]$. Also, an auxiliary bit array, called $\mathbf{gt}$, is assumed to be stored on disk, indicating whether each suffix in $S^h \dots S^2S^1$ is greater than the suffix $S^h \dots S^2S^1$ itself. The part of $\mathbf{gt}$ referring to the suffixes starting in S^h is loaded to the bit array $\mathbf{gt}_h[1..m]$ in internal memory.

Continuing from Example 1, at pass $h + 1 = 2$, we have the string buffer $s = S^2S^1 = GATGATGATA\$$ and $\mathbf{gt}_1 = [0, 1, 0, 0]$.

Then, the pass $h + 1$ proceeds through the following four steps.

In *Step 1*, the suffix array $\mathbf{sa}[1..m]$ containing the lexicographic ordering of the suffixes of $S[1..n]$ starting in block S^{h+1} (extending up to the last character $S[n]$) is computed in internal memory, without accessing any character of $S^{h-1} \dots S^2S^1$ on disk. To do that, the suffixes are compared using their prefixes stored in $s[1..2m]$, whenever the comparison of two suffixes exceeds position $s[m + 1]$, the algorithm uses the corresponding value in $\mathbf{gt}_h$ to determine their relative order. For example, given two suffixes starting at positions i and j of S^{h+1} , with $i < j$, if $s[i..m]$ is lexicographically smaller (or greater) than $s[j..j + m - i]$, then the suffix starting at i is smaller (or greater) than the suffix starting at j . Otherwise, if $s[i..m] = s[j..j + m - i]$, their relative order is determined by the corresponding bit stored in $\mathbf{gt}_h[j - i + 1]$.

In Example 1, at pass $h + 1 = 2$, the resulting suffix array for S^2 , $\mathbf{sa} = [2, 4, 1, 3]$, is computed using only the string buffer. Therefore, the suffix ordering in S^2S^1 is: $ATGS^1, GS^1, GATGS^1, TGS^1$.

In *Step 2*, the array $\mathbf{bwt}_{int}[1..m]$ is computed given $\mathbf{sa}[1..m]$. For each $i = 1, \dots, m$, the value of $\mathbf{bwt}_{int}[i]$ is set to $S^{h+1}[\mathbf{sa}[i] - 1]$ if $\mathbf{sa}[i] \neq 1$; otherwise, $\mathbf{bwt}_{int}[i]$ is assigned the special character $\#$, which does not appear in S . Note that $\mathbf{bwt}_{int}[1..m]$ is not the actual BWT of the substring S^{h+1} , as the suffixes of S^{h+1} extend up to the last character $S[n]$ and their relative order may differ.

Continuing from Example 1, at pass $h + 1 = 2$, $\mathbf{bwt}_{int} = GT\#A$ is computed from $\mathbf{sa}[1..4]$ and $S^2 = GATG$.

In *Step 3*, the algorithm computes a counter array, $\mathbf{gap}[1..m]$, which gives in $\mathbf{gap}[j]$ how many suffixes of $S^h \dots S^2S^1$ lie lexicographically between the suffixes in positions $\mathbf{sa}[j - 1]$ and $\mathbf{sa}[j]$ from the current block S^{h+1} . This counter array is computed in $\mathcal{O}(n)$ time with a single scan over $S^h \dots S^2S^1$ using n extra bits of $\mathbf{gt}$ and an $o(m)$ -bit data structure supporting constant time rank queries over $\mathbf{bwt}_{int}$. At *Step 3*, the algorithm also computes the content of $\mathbf{gt}_{h+1}$ for the next pass as a by-product of the computation of $\mathbf{gap}$.

Again, in Example 1, at pass $h + 1 = 2$, we have $\mathbf{gap} = [3, 0, 0, 1]$, since three suffixes from S^1 (namely $\$, A\$, and ATA\$$) are smaller than the first suffix of S^2 , and the suffix $TA\$$ from S^1 is smaller than only the last suffix of S^2 .

Finally, in *Step 4*, the algorithm merges $\mathbf{bwt}(S^h \dots S^2S^1)$ with $\mathbf{bwt}_{int}$ to create for the next pass $\mathbf{bwt}(S^{h+1}S^h \dots S^2S^1)$. To do that I/O-efficiently, the algorithm uses the counter array $\mathbf{gap}$, such that, for $i = 1, \dots, m$, $\mathbf{gap}[i]$ characters from $\mathbf{bwt}(S^h \dots S^2S^1)$ are copied to $\mathbf{bwt}(S^{h+1}S^h \dots S^2S^1)$ followed by the value $\mathbf{bwt}_{int}[i + 1]$. Whenever a character $\$$ or $\#$ is retrieved from $\mathbf{bwt}(S^h \dots S^2S^1)$ during this process, it is replaced by the last symbol $S^{h+1}[m]$ in $\mathbf{bwt}(S^{h+1}S^h \dots S^2S^1)$.

In Example 1, at the end of pass $h+1 = 2$, $\text{gap}[1] = 3$, then the first three values of $\text{bwt}(S^1)$ are retrieved (namely $AT\$$) and copied to $\text{bwt}(S^2S^1)$ as ATG directly to disk. Then, the value in bwt_{int} (namely G) is copied to $\text{bwt}(S^2S^1)$, together with the next two values in bwt_{int} (namely T and $\#$) as $\text{gap}[2] = 0$ and $\text{gap}[3] = 0$, resulting in $ATGGT\#$. Finally, $\text{gap}[4] = 1$ and the last value from $\text{bwt}(S^1)$ (namely A) is copied to $\text{bwt}(S^2S^1)$ and the last value from bwt_{int} (namely A) is copied as well, yielding the final result: $\text{bwt}(S^2S^1) = ATGGT\#AA$.

Theoretical costs. The running time of bwt-disk is $\mathcal{O}(n^2/M)$, for a (single) string $S[1..n]$, from a constant alphabet, using $M \lceil \log n \rceil$ bits of RAM. The algorithm executes $\mathcal{O}(n/M)$ passes over $\Theta(n)$ bits of disk data, using n bits of disk working space. The total number of I/Os is $\mathcal{O}(n^2/(MB \log n))$.

3.2 BCR and BCReXt [Bauer et al., 2013]

In this section we describe the core of a strategy for constructing the BWT of a string collection $\mathcal{S} = \{S_1, \dots, S_k\}$ according to the definition in Equation (1), which considers the strings in a circular way so that the symbol preceding the suffix $S_j[1..n_j]$ is the end-marker of the string S_j (i.e., $\$j$). The heart of this strategy is common to two algorithms, named BCR and BCReXt, introduced in [5] and part of the BEETL library⁷. BCR works in (semi-)external memory, whereas BCReXt works in external memory.

The common strategy does not require linearizing the strings in $\mathcal{S}$ by concatenating them and does not require explicitly computing the GSA. Rather, it builds the BWT incrementally in m steps, where m is the length of the longest string in $\mathcal{S}$ (including the end-markers) by exploiting the basic properties of the BWT and the LF-mapping. The basic idea is to left-align (or, equivalently, right-align) all the strings in $\mathcal{S}$ and, by scanning all the strings in $\mathcal{S}$ right-to-left at the same time, to incrementally construct the partial BWT of suffixes of $\mathcal{S}$ with length at most $h+1 \leq m$, with $h = 0, \dots, m-1$.

For simplicity, in this description, we assume that all the strings have the same length m and they are right-aligned. So, at each iteration only suffixes of the same length are considered. Both algorithms build the $\text{bwt}(\mathcal{S})$ in m steps. At each step $h = 0, \dots, m-1$, they build a partial BWT, denoted $\text{bwt}_h(\mathcal{S})$, containing the BWT of suffixes in $\mathcal{S}$ of length up to $h+1$.

Since we consider distinct end-markers and $\$i < \$j$ if $i < j$, then it is easy to verify that $\text{bwt}_0(\mathcal{S})$ is obtained by concatenating the last symbol of each string S_j in the same order as the strings appear in the collection: $\text{bwt}_0(\mathcal{S}) = S_1[n_1-1]S_2[n_2-1] \dots S_k[n_k-1]$.

For each step $h = 1, \dots, m-2$, both algorithms simulate the insertion of all suffixes of length $h+1$ in the list of sorted suffixes, i.e., insert all symbols (circularly) preceding the suffixes of length $h+1$ at the correct positions into $\text{bwt}_{h-1}(\mathcal{S})$ in order to obtain $\text{bwt}_h(\mathcal{S})$.

In the last step $h = m-1$, the k end-markers are inserted at the correct positions into $\text{bwt}_{m-2}(\mathcal{S})$ and $\text{bwt}_{m-1}(\mathcal{S}) = \text{bwt}(\mathcal{S})$ is obtained.

At the iteration $h > 0$, let p_{j-1} be the position of the symbol $c = S_j[n_j-h]$ in $\text{bwt}_{h-1}(\mathcal{S})$. To insert the symbol $S_j[n_j-h-1]$ preceding the suffix $S_j[n_j-h, n_j]$ in $\text{bwt}_{h-1}(\mathcal{S})$, we compute the position p_j using the LF-mapping (we omit details for space reasons): $p_j = C[c] + \text{rank}(c, p_{j-1})$, where $C[c]$ is the number of symbols lexicographically smaller than c in $\text{bwt}_{h-1}(\mathcal{S})$, $\text{rank}(c, p_{j-1})$ is the function that returns the number of occurrences of c in the prefix $\text{bwt}_{h-1}(\mathcal{S})[1, p_{j-1}]$.

⁷ <https://github.com/BEETL/BEETL>

By using Example 2, the base case of BCR/BCRext consists in concatenating all symbols preceding the suffixes $\$1, \$2, \$3$ obtaining $\text{bwt}_0(\mathcal{S}) = TCA$ (because the partial sorted suffixes is: $[\$1, \$2, \$3]$ since $\$1 < \$2 < \$3$). Then, they simulate the insertion of the $T\$1, C\$2, A\$3$ suffixes into the partial sorted suffix list by inserting the symbols G, A, A preceding the $T\$1, C\$2, A\$3$ respectively suffixes into bwt_0 , resulting in $\text{bwt}_1 = TCAAAG$. The process continues by simulating the insertion of the suffixes $GT\$1, AC\$2, AA\$3$ into the sorted suffix list, then inserting the symbols C, A, C preceding the $GT\$1, AC\$2, AA\$3$ suffixes in order to obtain $\text{bwt}_2(\mathcal{S})$. And so on. In the last step, they simulate the inserting of the suffixes $AGCGT\$1, TCAAC\$2, CGCAA\$3$ and insert the symbols $\$1, \$2, \$3$ in the partial BWT $\text{bwt}_{m-1}(\mathcal{S}) = \text{bwt}(\mathcal{S})$.

If the strings have different lengths, an end-marker is inserted when the longest suffix of a string in $\mathcal{S}$ is reached, and then no further symbols of that string will be inserted in the following iterations.

Note that, in practice, neither algorithm needs to explicitly append end-marker symbols to the strings, since they will only be inserted last and the symbols concatenated in bwt_0 follow the input order. Moreover, to avoid exponential growth of the alphabet, both algorithms write the same end-marker symbol in the BWT output string, but they can store and output the index associated with each end-marker in a separate file. This is particularly important for decoding. An interested reader can refer to [5, 6].

In order to reduce the I/O operations, both algorithms split each partial BWT $\text{bwt}_{h-1}(\mathcal{S})$ in $\sigma + 1$ files $B_{h-1}(t)$, where $B_{h-1}(0)$ contains the symbols preceding the suffixes consisting of only the end-marker symbols and $B_{h-1}(t)$ contains the symbols preceding the lexicographically sorted suffixes of $\mathcal{S}$ of length at most $h - 1$ and starting with $c_t \in \Sigma$.

The difference between the two algorithms lies in the data structures they employ, and therefore in the strategy they adopt to insert symbols incrementally.

In fact, BCR needs a data structure to keep track of each symbol to be inserted, the index of the sequence to which it belongs, and the position where each symbol was inserted in the previous iteration. Once the positions at which the new symbols are to be inserted have been updated, BCR needs to sort this information so that, at iteration h , it can insert the symbols into each of the B_{h-1} files sequentially.

BCRext computes and updates the positions where new symbols are to be inserted in a similar way to BCR, but it keeps this information in auxiliary files and the strings themselves are sorted externally by using the least-significant-digit radix sort. So it uses a negligible workspace. At the start of iteration h , the suffixes of length $h - 1$ of $\mathcal{S}$ are assumed to be ordered, this ordering being partitioned into external files $F_{h-1}(1), \dots, F_{h-1}(\sigma)$ according to the first symbols of the suffixes of length $h - 1$. Files $P_{h-1}(1), \dots, P_{h-1}(\sigma)$ are such that $P_{h-1}(s)$ contains the positions of the suffixes in $B_{h-1}(s)$, ordered the same way.

In [20], the authors show that BCR can be augmented in order to compute the LCP array and the generalized suffix array.

Moreover, in both algorithms the collection $\mathcal{S}$ is considered ordered by using the input order, but one can modify them so that one can use a different ordering for the strings in $\mathcal{S}$ to reduce the number of runs in the output of the BWT (see [19, 44, 6, 13, 14]).

Theoretical costs. BCR is performed in $\mathcal{O}((k + \sigma^2) \log N)$ bits of memory, with a worst-case time complexity of $\mathcal{O}(m(N + \text{sort}(k)))$, where $\text{sort}(k)$ is the time taken to sort k integers and $\mathcal{O}(Nm \log(\sigma))$ I/O (bits). Whereas BCRext uses $\mathcal{O}(\sigma^2 \log(N))$ bits of memory, with a worst-case time complexity of $\mathcal{O}(kN)$ and $\mathcal{O}(Nm \log(\sigma) + N \log(N))$ I/O (bits).

3.3 eGSA [Louza et al., 2017]

The algorithm eGSA⁸ [49] works in external memory and computes the (generalized) BWT for a string collection $\mathcal{S} = \{S_1, \dots, S_k\}$ of total length N according to the definition in Equation (1), with each string $S_j \in \mathcal{S}$ having the same end-marker $\$$ for computational efficiency. The algorithm works in two phases, as follows.

Phase 1. The collection $\mathcal{S}$ is partitioned into sub-collections $\mathcal{S}^1, \mathcal{S}^2, \dots, \mathcal{S}^m$ based on the available memory, such that each $\mathcal{S}^x$ has size less than $\lfloor M/4 \rfloor$, where M denotes the number of available words in RAM. For each sub-collection $\mathcal{S}^x$, the arrays $\mathbf{gsa}^x$ and $\mathbf{lcp}^x$ are computed in internal memory and stored to disk. The authors used gSACA-K [48] combined with Φ -algorithm [36]. Together, these algorithms run in linear. As each value $\mathbf{gsa}^x[i] = (t, j)$ and $\mathbf{lcp}^x[i]$ are written to disk, the corresponding $\mathbf{bwt}^x(\mathcal{S}^x)[i]$ and a substring of length p of the suffix $S_j[t, n_j]$, denoted as $\mathbf{pre}^x[i]$, are computed and written to disk. In particular, $\mathbf{pre}^x[i]$ gives a prefix of $S_j[t, n_j]$ when combined with previous values and it is used to reduce external memory accesses in Phase 2.

In Example 2, suppose the input collection $\mathcal{S}$ is partitioned into two sub-collections $\mathcal{S}^1 = \{AGCGT\$1, TCAAC\$2\}$ and $\mathcal{S}^2 = \{CGCAA\$3\}$, with prefix size $p = 2$. First, $\mathbf{gsa}^1 = [(1, 5), (2, 5), (2, 2), (2, 3), (1, 0), (2, 4), (2, 1), (1, 2), (1, 1), (1, 3), (1, 4), (2, 0)]$ and $\mathbf{lcp}^1 = [0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1]$ are computed in internal memory. While each pair $\mathbf{gsa}^1[i]$ and $\mathbf{lcp}^1[i]$ is written to disk, the corresponding values in $\mathbf{bwt}^1(\mathcal{S}^1) = TCCA\$ATGACG\$$ and $\mathbf{pre}^1 = [\$ \$, \$ \$, AA, AC, AG, C \$, CA, CG, GC, GT, T \$, TC]$ are obtained and stored on disk as well. Next, $\mathbf{gsa}^2 = [(3, 5), (3, 4), (3, 3), (3, 2), (3, 0), (3, 1)]$ and $\mathbf{lcp}^2 = [0, 0, 1, 0, 1, 0]$ are computed in internal memory, and the values of $\mathbf{bwt}^2(\mathcal{S}^2) = AACG\$C$ and $\mathbf{pre}^2 = [\$ \$, A \$, A \$, CA, GC, GC]$ are obtained and stored on disk in the same way.

Phase 2. The computed arrays of $\mathcal{S}^1, \mathcal{S}^2, \dots, \mathcal{S}^m$ are merged with the help of a binary heap and internal memory buffers designed to reduce string comparisons on the disk. Let $\mathbf{gesa}^x = \langle \mathbf{gsa}^x, \mathbf{lcp}^x, \mathbf{bwt}^x, \mathbf{pre}^x \rangle$ be the computed arrays of $\mathcal{S}^x$. Each $\mathbf{gesa}^x$ is split into r^x blocks of b elements, except possibly the last block, ensuring that one block of each $\mathbf{gesa}^1, \mathbf{gesa}^2, \dots, \mathbf{gesa}^m$ can simultaneously fit internal memory. Initially, the first block of each $\mathcal{S}^x$ is loaded from disk to a buffer $B^x[1, b]$. The heading element of each B^x is inserted into a binary min-heap H . For $i = 1, 2, \dots, N$, the smallest suffix, say from B^x , is removed from H , and the value of $\mathbf{bwt}(\mathcal{S})[i]$ is written to an output buffer O of size d . Whenever O is full, it is written to the disk. Then, the next element from the same block buffer B^x is inserted to the heap. Whenever a buffer B^x is empty, the next corresponding block in $\mathbf{gesa}^x$ is loaded from disk to B^x .

In Example 2, considering the block size $b = 3$, we have the following initial blocks in the buffers $B^1 = [\langle (1, 5), 0, T, \$ \$ \rangle, \langle (2, 5), 0, C, \$ \$ \rangle, \langle (2, 2), 0, C, AA \rangle]$ and $B^2 = [\langle (3, 5), 0, A, \$ \$ \rangle, \langle (3, 4), 0, A, A \$ \rangle, \langle (3, 3), 1, C, A \$ \rangle]$. We assume there is enough internal memory to hold B^1 and B^2 simultaneously. The heading elements $\langle (1, 5), 0, T, \$ \$ \rangle$ and $\langle (3, 5), 0, A, \$ \$ \rangle$ are inserted into the heap H .

Note, however, that each heap comparison during Phase 2 may require accessing strings in disk randomly. To reduce disk accesses, the following strategies are presented.

⁸ An implementation is available at <https://github.com/felipelouza/egsa>

For each sub-collection $\mathcal{S}^x$ a string buffer W^x of maximum size s is used to store a prefix of the heading suffix from B^x in internal memory. Therefore, whenever a suffix from $\mathcal{S}^x$, say $\text{gesa}^x[i]$ with $\text{gsa}^x[i] = (t, j)$, is inserted into H , if $\text{lcp}[i] = 0$ then $W^x = \text{pre}^x[1, p] \cdot \#$, otherwise, $W^x = W^x[1, h_i] \cdot \text{pre}^x[1, p] \cdot \#$, where $h_i = \min(\text{lcp}^x[i], h_{i-1} + p)$ and $\#$ is an end-of-buffer marker. In this way, a starting portion of $S_j[t, n_j]$ is already loaded into W^x when it is compared with other suffixes in H . If $\#$ is reached, S_j is accessed in disk. Also, if $h_i + p > s$ the prefix is truncated and the next substrings are accessed from disk.

Continuing from Example 2, when the elements compared in H are $\langle(2, 2), 0, C, AA\rangle$ from B^1 and $\langle(3, 4), 0, A, A\$ \rangle$ from B^2 , their string buffers $W^1 = AA\#$ and $W^2 = A\#\#$ provide enough information to decide that the smallest suffix comes from B^2 (without accessing the disk) and $\text{bwt}(\mathcal{S})[i] = A$ is added to the output buffer O . The next element from B^2 , $\langle(3, 3), 1, C, A\$ \rangle$, is inserted into the heap H . The prefix W^2 is updated by considering that the current element from B^2 shared one symbol in common with the previous one, resulting in $W^2 = AA\#\#$.

Another strategy is the usage of the LCP between nodes in H to avoid string comparisons in heap insertions. Let X from B^x be the root of H and nodes Y and Z its children, if the LCP between X and the next element from B^x is larger than the LCP between X and Y , and the LCP between X and Z , the next smallest value of H comes from buffer B^x . Also, if the LCP between X and the next element from B^x is smaller or equal to the LCP between X and Y , the comparison between their corresponding suffixes can start from the minimum LCP value. The LCP values in H are updated as nodes are swapped and can be used to speed up suffix comparisons as well.

In Example 2, again when the elements $\langle(2, 2), 0, C, AA\rangle$ from B^1 and $\langle(3, 4), 0, A, A\$ \rangle$ from B^2 are compared in H , the LCP between the corresponding nodes in H is computed and stored. Subsequently, the node representing B^2 is removed from H , and the element $\langle(3, 3), 1, C, A\$ \rangle$ is inserted. Since the LCP between the new element from B^2 and the previous one is equal to 1 and the LCP between the corresponding nodes in H is also equal to 1, the next comparison in H between the elements from B^1 and B^2 may start from the second symbol of W^1 and W^2 . In particular, this suffix comparison will require accessing the disk, as W^1 is a prefix of W^2 .

Finally, eGSA also induces suffixes as each $\text{gesa}^x[i]$ is removed from H , with $\text{gsa}^x[i] = (t, j)$. Whenever $\text{bwt}^x[i] > S_j[t]$ the suffix $S_j[t - 1, n_j]$ will be induced. To do that, the value x is inserted into a (queue) induced buffer I_c , with $S_j[t - 1] = \text{bwt}^x[i] = c$. Each induced buffer I_c , with $c \in \Sigma$, is written to disk as it gets full. Then, when the first suffix starting with c is the smallest value in H , the induced values in disk and in I_c are used to decide which heading element of the blocks is the smallest in H and send it to the output buffer, with no string comparison. As these values are sent to O the corresponding nodes in H are consumed.

In Example 2, when the element $\langle(1, 5), 0, T, \$\$ \rangle$ from B^1 is the smallest in H , the suffix $S_1[4, 5] = T\$$ will be induced. To do that, $x = 1$ is inserted into the induced buffer I_T . Then, when the first suffix starting with T becomes the smallest in H , the heading element $\langle(1, 4), 0, G, T\$ \rangle$ from B^1 is send directly to the output buffer, without string comparison.

Theoretical costs. The running time of eGSA is $\mathcal{O}((N \log N/M) \text{maxlcp})$, where maxlcp is length of the longest common prefix between suffixes of the input strings. The total number of I/Os is $\mathcal{O}((N \log N/M) \max |S^i|)$, where $\max |S^i|$ is the length of the longest string in the collection.

3.4 eGap [Egidi et al., 2019]

The algorithm **eGap**⁹ [23] is an external memory algorithm for the computation of the BWT of a collection of strings $\mathcal{S} = \{S_1, \dots, S_k\}$, with $S_i \neq S_j$ for $i \neq j$. The algorithm works in two phases, first using the available RAM to compute in memory the BWTs of subcollections of $\mathcal{S}$, and then merging the results into the final BWT.

Phase 1. **eGap** uses the optimal linear time algorithm **gSACA-K** [48] for the first phase; the size of the subcollections is determined based on the available memory, taken in input. The **gSACA-K** algorithm is used to compute the suffix array for subcollections, from which, by Equation (2), the multi-string $\text{bwt}(\mathcal{S})$ is obtained.

Phase 2. The second phase is based on the **gap** algorithm [24], in turn derived from an earlier algorithm by Holt and McMillan [32]. The ideas from **gap** are used to merge the BWTs of the subcollections.

For simplicity of exposition, we describe here the algorithm when merging single-string BWTs, but the same algorithm can merge multi-string BWTs. The latter is important because it allows to adopt a multi-stage strategy in which the BWTs computed in Phase 1 are merged in successive rounds.

The merging phase does not need to access the original strings but requires only their BWTs $\text{bwt}_{S_1}, \dots, \text{bwt}_{S_k}$. It works iteratively sorting the bwt_{S_λ} symbols according to prefixes of increasing lengths of their contexts: after iteration h , they are sorted according to the length- h prefixes of their contexts. The original BWTs are not explicitly sorted during the iterations, but the algorithm produces a recipe for performing the merge: the final iteration yields a length n , k valued vector Z such that $Z[i] = \lambda$ iff the i -th entry of $\text{bwt}(\mathcal{S})$ is from bwt_{S_λ} ($i = 1, \dots, N$; $\lambda = 1, \dots, k$).

More formally, let $Z^{(h)}$ be the approximation of array Z after iteration h , with $Z^{(0)} = 1^{n_1} 2^{n_2} \dots k^{n_k}$. Then the following property holds:

Property Z. For $\lambda_1, \lambda_2 \in \{1, \dots, k\}$, $\lambda_1 < \lambda_2$, and $i = 1, \dots, n_{\lambda_1}$ and $j = 1, \dots, n_{\lambda_2}$ the i -th λ_1 precedes the j -th λ_2 in $Z^{(h)}$ iff $S_{\lambda_1}[\text{sa}_{\lambda_1}[i], \text{sa}_{\lambda_1}[i] + h - 1] \preceq S_{\lambda_2}[\text{sa}_{\lambda_2}[j], \text{sa}_{\lambda_2}[j] + h - 1]$. ◀

For instance, assume that we are merging the three BWTs of the strings of Example 2. The BWTs are $\text{bwt}_{S_1} = T\$1GACG$, $\text{bwt}_{S_2} = CCAAT\$2$ and $\text{bwt}_{S_3} = AACG\$3C$. Then, $Z^{(2)} = [1, 2, 3, 3, 2, 3, 2, 1, 2, 2, 3, 1, 3, 1, 3, 1, 1, 2]$ which corresponds to a partially merged **bwt** $TCAACCA\$1ATGG\$3ACCG\$2$, where the suffixes are ordered only by their length two prefixes. For instance, the twelfth and thirteenth character are $G\$3$ as opposed to $\$3G$ in the final $\text{bwt}(\mathcal{S})$ since their suffixes both start by CG and G comes from bwt_{S_2} whereas $\$3$ is from bwt_{S_3} .

Array $Z^{(h)}$ is obtained through a sequential scan of $Z^{(h-1)}$ together with $\text{bwt}_{S_1}, \dots, \text{bwt}_{S_k}$. The array $Z^{(h)}$ is partitioned into σ buckets, one for each character of the alphabet. This is achieved through the array $C[1, \sigma]$, where $C[c]$ ¹⁰ is defined as the number of occurrences of characters smaller than c in $\text{bwt}(\mathcal{S})$. Therefore $C[c]$ marks the beginning of the c -bucket in $Z^{(h)}$, whose first entry is $Z^{(h)}[C[c]]$. When scanning $Z^{(h-1)}$, if $Z^{(h-1)}[i]$ is λ , the algorithm looks at the next character in bwt_{S_λ} : if it is c , then it stores λ in the next free position of

⁹ An implementation is available at <https://github.com/felipelouza/egap/>

¹⁰ In [23] the array C is denoted F .

bucket c in $Z^{(h)}$. So, continuing the example above, the character C bucket of $Z^{(3)}$ will be 22331. Notice that now the twelfth character will be S_3 since its suffix is prefixed by CGC and the thirteenth character will be G , since its suffix starts with CGT .

From Property Z, it follows that $Z^{(h)}$ can be logically partitioned in blocks corresponding to sets of $\text{bwt}(S)$ symbols whose contexts share the same length- h prefixes. An additional bit array B keeps track of these blocks: $B[i] \neq 0$ iff a new block starts at $Z^{(h)}[i]$. When all entries of array B are non zero, $Z^{(h)}$ will not change in subsequent iterations and it is the required array Z . In our example, at the last iteration $Z = [1, 2, 3, 3, 3, 2, 2, 1, 2, 3, 2, 3, 1, 1, 1, 2]$.

Then, a sequential scan of Z and $\text{bwt}_{S_1}, \dots, \text{bwt}_{S_k}$ allows to write the characters of $\text{bwt}(S)$ in sequential order. Back to our example: $\text{bwt}(S)[0]$ is the first from $\text{bwt}_{S_{Z[0]}} = \text{bwt}_{S_1}$, so it's T ; the second is the first from $\text{bwt}_{S_{Z[1]}} = \text{bwt}_{S_2}$ (it's C); then we have an A from $\text{bwt}_{S_{Z[2]}} = \text{bwt}_{S_3}$; and then the second character from bwt_{S_3} (again an A) since $Z[3] = 3$.

The k input BWTs are read from disk and never moved to memory.

Phase 2 of **eGap** actually only uses two copies of array Z to store $Z^{(h)}$ for all values of h : namely, Z^{old} and Z^{new} , whose roles are swapped at each iteration. They can be maintained in external memory, since access to each bucket of Z^{old} and Z^{new} is sequential.

In [24, Lemma 5], it is proven that if $B[i]$ is set to 1 at iteration h , then $Z^{(g)}[i]$ will not change any more at any iteration $g \geq h + 2$. Since the roles of Z^{old} and Z^{new} are swapped, starting at iteration g , processing $Z^{\text{old}}[i]$ to compute $Z^{\text{new}}[i]$ will be useless. So we define an array B_x to keep track of entries that will not change again: if at iteration h the algorithm finds that $B[i] = 1$, then $B_x[i]$ will be set to 1 as well. Any sequence of 1s in B_x defines an *irrelevant range* that can be skipped in the subsequent processing. So, at each iteration, irrelevant ranges are sequentially read from file to skip them, and new irrelevant ranges are sequentially written to file. Since maintaining and skipping ranges has a cost, only ranges of a significant size (according to a configurable parameter) are considered.

Computing the LCP. The **eGap** algorithm can also compute the LCP of the input collection. This is done in two steps: during Phase 2, if at iteration h , $B[i] = 1$ and $B_x[i] = 0$, a pair $\langle i, h - 2 \rangle$ is written to file F_{h-2} to record that $\text{lcp}_{1..k}[i] = h - 2$ (then $B_x[i]$ is set to 1). Notice that in file F_{h-2} all entries are ordered according to their first components. In an additional third phase of the algorithm, the entries from all temporary files F_h are merged using a standard external memory multiway merge.

Theoretical costs. We analyze costs assuming a fixed size alphabet. Phase 1 of the algorithm runs in total time $\mathcal{O}(N)$, splitting the input into subcollections of size $\approx M/9$, if M is the available RAM.

Without skipping the irrelevant ranges, Phase 2 would require maxlcp (the maximum LCP value) sequential scans of $\mathcal{O}(N)$ items. By skipping irrelevant ranges, the overall amount of data *directly* read/written by the algorithm is $\mathcal{O}(N \text{ avelcp})$ items where avelcp is the arithmetic average of the entries in the LCP array. On the other hand, skipping irrelevant ranges causes an overhead in terms of blocks; so the overall cost of Phase 2 can be bound above by $\mathcal{O}(N \text{ maxlcp})$ sequential I/Os.

RAM usage is flexible. In Phase 1, the larger the memory used, the faster the algorithm. Phase 2 uses a negligible amount of RAM, but also a semi-external version that uses $\mathcal{O}(N)$ bytes of RAM was implemented.

The overall cost of Phase 3, to sort the LCP entries, is $\mathcal{O}(N \log_K \text{ maxlcp})$ sequential I/Os, since it requires $\mathcal{O}(\lceil \log_K \text{ maxlcp} \rceil)$ rounds (where K is a configurable parameter), each one merging K LCP files by sequentially reading and writing $\mathcal{O}(N)$ bytes of data.

3.5 BWT+LCP [Bonizzoni et al., 2021]

The BWT+LCP algorithm [8], implemented in a prototype called `bwt-lcp-em`¹¹, has similarities with BCR/BCRext [5] (Section 3.2) and uses ideas from [32], but proposes a different strategy.

The paper [8] uses a specific definition of suffixes that, contrary to the definition adopted in this survey, does not include the end-marker when counting the length of a suffix. So a suffix can have length 0 and in this case it is not the empty string but it consists of the end-marker. We chose here to use a notation that agrees with this choice, for the sake of the reader who wants to delve into the details of [8]. To avoid confusion, in the following we will call these suffixes *no\$-suffixes*.

The algorithm works in two distinct phases.

It first computes partial multi-string BWTs, and then merges the BWTs to the final BWT, making largely use of external memory.

The partial BWT¹² $\text{bwt}_{=j}$ only takes into account no\$-suffixes of length j . Notice the difference from the partial BWTs bwt_j of BCR (Section 3.2) that are relative to all suffixes of lengths up to j . Also notice the difference from eGap (Section 3.4) where the partial BWTs computed in the first phase are complete BWTs of subcollections.

Phase 1. The partial BWTs are computed with the support of positional representations of the input strings: these are arrays T_i such that element $T_i[j]$ is the i -th character from the end of string S_j , *without* the end-marker \$. The end-marker is added as a first character to each string. So T_0 contains all last characters of the strings.

The first partial BWT $\text{bwt}_{=0}$ contains the concatenation of the last characters of each string, since it is the BWT of length 0 no\$-suffixes (i.e. suffixes consisting only of the end-markers), and so is identical to T_0 .

To produce $\text{bwt}_{=j}$ from $\text{bwt}_{=j-1}$, for each $j = 1, \dots, \ell$ (where ℓ is the maximum length of the strings), a radix sort is used, ordering the strings from the rightmost character. In this phase array T_j must be kept in main memory since it is accessed in random order. The sorting is implemented using arrays N_i ($i = 1, \dots, \ell$). At the beginning of the j -th iteration, $N_j[i] = h$ if $\text{bwt}_{=j+1}[i]$ must be read from string S_h . As N_j is scanned, characters from the required strings are read: if character c is read from string S_h , it is appended to $\text{bwt}_{=j}$ which is under construction and h is appended to the bucket for character c . The concatenation of all buckets, ordered according to the alphabetical ordering of the characters, is the new N_{j+1} . The array N_0 is trivially initialized as $N_0 = \mathbf{12} \cdots \mathbf{k}$.

Let's consider again the strings of Example 2. To show how Phase 1 works, let's see how $\text{bwt}_{=4}$ is obtained from $N_3 = 231$ and T_4 . Notice that $T_4 = ATC$ since it contains the 5th character from the end of each string, not counting the end marker. According to N_3 , the first character must be read from string 2, so the first character of $\text{bwt}_{=4}$ is T . At the same time, the first index in the T bucket is 2. The second character must come from S_3 (it is C , and bucket C gets a 3), and the third from S_1 (it is an A and bucket A gets a 1). So $\text{bwt}_{=4} = TCA$ and $N_4 = 132$, as the concatenation of the buckets for A , C and T in alphabetical order.

¹¹ <https://github.com/AlgoLab/bwt-lcp-em>

¹² In [8] the partial BWTs are denoted bwt_j , and the maximum length is k . We use here the “=” in $\text{bwt}_{=j}$ to explicitly distinguish these partial BWTs from the partial BWTs bwt_j computed in BCR.

Phase 2. When all partial BWTs $\text{bwt}_{=j}$ have been built, they are merged to the final BWT. Notice that the ordering of each partial BWT will be preserved in the merge, since length- j no\$-suffixes are already correctly ordered relative to each other.

The merge is first carried out virtually building an array I that describes how the merge must be carried out. Specifically $I[j] = h$ if the j -th element of the final BWT is the character preceding a length- h suffix. Therefore, to construct the output BWT, element j will be taken from $\text{bwt}_{=h}$.

To build I , the no\$-suffixes of different lengths must be ordered. I is approximated in successive iterations, and the no\$-suffixes are ordered starting from their last characters and using a radix sort algorithm.

Precisely, at iteration j , an approximation I_j is computed from I_{j-1} . The initial I_0 trivially proposes to concatenate all the partial BWTs in increasing order of the suffix lengths, that is $I_0 = \mathbf{0}^{|\text{bwt}_{=0}|} \mathbf{1}^{|\text{bwt}_{=1}|} \dots \ell^{|\text{bwt}_{=\ell}|}$.

Then, at iteration j , I_j is scanned sequentially, and for each i , if $I_j[i] = h$ then the next character c is read from $\text{bwt}_{=h}$. If it is not the end-marker, the length $h + 1$ is appended to the c -bucket. The \$-bucket is fixed as a sequence of 0s, since the suffixes starting with the end-markers are no\$-suffixes of length 0. Then I_{j+1} is the concatenation of all buckets, in alphabetical order of the characters.

Let us see this in our running example: the partial bwts are: $\text{bwt}_{=0} = TCA$, $\text{bwt}_{=1} = AAG$, $\text{bwt}_{=2} = CAC$, $\text{bwt}_{=3} = CGG$, $\text{bwt}_{=4} = TCA$ and $\text{bwt}_{=5} = \$_1\$_3\$_2$. Scanning $I_0 = \mathbf{0}^3 \mathbf{1}^3 \mathbf{2}^3 \mathbf{3}^3 \mathbf{4}^3 \mathbf{5}^3$, since $I_0[0] = 0$, the first character is from $\text{bwt}_{=0} = TCA$. Then the character is T , and so bucket T gets as first index $I_0[0] + 1 = 1$. The second character is again from $\text{bwt}_{=0}$, since $I_0[1] = 0$; it is a C and thus bucket C gets as first index $I_0[1] + 1 = 1$. All buckets are filled continuing in this fashion, except that when an end marker is encountered, it is disregarded: so, for instance, nothing is done for $I_0[15] = 5$, since the first character of $\text{bwt}_{=5}$ is $\$_1$. Finally, the concatenation of all buckets in alphabetical order yields $I_1 = [0, 0, 0, 1, 2, 2, 3, 5, 1, 3, 3, 4, 5, 2, 4, 4, 1, 5]$

From the final I , $\text{bwt}(S)$ is computed as it is done in **eGap** using the array Z , except that here the partial bwts $\text{bwt}_{=j}$ are used. In our example, $I = [0, 0, 0, 1, 2, 3, 2, 5, 1, 3, 4, 5, 3, 4, 4, 2, 1, 5]$. The first three characters of $\text{bwt}(S)$ are TCA from $\text{bwt}_{=0}$, since I starts with three 0s. Then we have an A which is the first character of $\text{bwt}_{=1}$, since $I[3] = 1$, and so on.

Computing the LCP. As implied by the name, the BWT+LCP algorithm [8] can also compute the LCP array of the input sequence collection. This is done during the second phase: as the array I is computed, at each iteration, the algorithm keeps track of the longest prefixes locally encountered at each location, by inspecting the corresponding BWT characters.

Theoretical costs. For a fixed size alphabet, the algorithm runs in $\mathcal{O}(N \text{maxlcp})$ time, where maxlcp is the maximum LCP value, when the alphabet is constant and memory addresses can be stored in a single memory word. It uses $\mathcal{O}(k + M + \log \text{maxlcp})$ main memory (where M is the maximum length of the input strings) and requires $\mathcal{O}(N \text{maxlcp})$ I/Os.

4 Strategies exploiting compressibility

Another way to scale the computation of large BWTs is to apply a lightweight compression scheme on top of the text to produce a small data representation from which we derive the BWT. This strategy has several advantages: (i) the compressed representation fits smaller

memories that are closer to the CPU, so the transmission of data is more efficient. We need fewer bits of satellite information on top of the text (i.e., less RAM and disk usage), and (ii) the deduplication of strings via compression avoids redundant BWT computations. Overall, if we use the correct tools, this approach reduces both time and space. However, it also has some pitfalls. A greedy compression scheme (like Lempel-Ziv or RePair) greatly reduces space usage, but also imposes considerable overhead, overcoming the whole purpose of compressing. Moreover, the compact data representation we produce must follow some structure that facilitates the computation of the BWT, otherwise decompressing the strings every time we require to operate over them makes things more inefficient.

In this section, we present algorithms that exploit the repetitiveness of the text to compute the BWT.

4.1 Big-BWT [Boucher et al., 2019]

Boucher *et al.* [11] introduced a pre-processing technique, called *prefix-free parsing*, to compute the BWT for a (single) highly repetitive input string S of length n in $\mathcal{O}(n)$ time. The rationale behind this technique is to apply a simple compression scheme as to exploit the string repetitiveness, and then to compute the BWT from the encoded string. The pre-processing step generates a dictionary and a parse of the input string, and it is effective if both the dictionary and the parse together are much smaller than the original input string. In this case, indeed, the BWT computation executed in internal memory is more resource-efficient. The Big-BWT tool¹³ implements this technique.

Experiments showed that the size of the parse is typically the most demanding component for very large and repetitive inputs. Thus, in order to reduce the memory requirements, *recursive prefix-free parsing* has been recently introduced in [58], where the prefix-free parse is applied to the parse generated by prefix-free parsing the input string.

In the following we outline the key phases of the BWT computation via prefix-free parsing and highlight the properties of this technique.

Parsing phase. The main idea of the parsing phase is to divide the input string S into overlapping phrases of variable length that will constitute the dictionary. More precisely, the string considered is $\#S\$^w$, where S is the input string of length n over Σ , $\#$ and $\$$ are end-markers, and $w \geq 1$ is a fixed parameter.

In order to parse $\#S\$^w$, let $E \subseteq \Sigma^w$ be a set of strings of fixed length w over Σ (called *trigger strings*) augmented with $\#$ and $\$^w$. The *dictionary* $\mathcal{D}$ comprises all the strings d such that (i) d is a substring of $\#S\$^w$, (ii) exactly one prefix of d is in E , (iii) exactly one suffix of d is in E , (iv) no other substring of d is in E . The sequence of dictionary phrase occurrences that form $\#S\$^w$ is the *parse* $\mathcal{P}$, in which each phrase occurrence is encoded by a meta-character given by the lexicographic rank of that phrase in $\mathcal{D}$. Then, $\text{PFP}(S) = (\mathcal{D}, \mathcal{P})$, where $\mathcal{D}$ is lexicographically sorted and $\mathcal{P}$ is composed of characters that correspond to positions in $\mathcal{D}$. For instance, let S be the string in Example 1, $w = 2$, and trigger strings given by AT , $\#$ and $\$^2$. Then, $\mathcal{D} = \{\#CAT, ATA\$, ATGAT\}$. The parse of $\#S\$^2$ is $\#CAT ATGAT ATGAT ATA\$,$ which results in $\mathcal{P} = [0, 2, 2, 1]$, when using the lexicographic rank to identify phrases in $\mathcal{D}$.

In Big-BWT [11], $\mathcal{D}$ is iteratively built through a one-pass over $\#S\$^w$ using a Karp-Rabin hash function and a parameter p . The trigger strings are implicitly found by passing a sliding window of length w : wherever the KR fingerprint of the content of the current window is 0

¹³<https://gitlab.com/manzai/Big-BWT>

modulo p , a trigger string is found. Then, the current phrase terminates at the end of the window, and it is added to the dictionary (recording also the phrase frequency), and then, the next phrase starts at the beginning of the current window. Thus, the input string is decomposed into overlapping variable-length phrases, each starting and ending with a trigger string of length w . After sorting the dictionary $\mathcal{D}$, the final parse $\mathcal{P}$ is generated.

The main property of the prefix-free parsing procedure is that none of the suffixes of length greater than w of the phrases in $\mathcal{D}$ is a prefix of any other [11, Lemma 1]. This is a property fundamental for the next phase when building the BWT of S .

Building the BWT of the input string. To construct the BWT of S , we follow the definition that involves appending the end-marker $\$$ to S and then lexicographically sorting its suffixes.

Any suffix of $S\$$ corresponds to a unique suffix of $\#S\$^w$ of length greater than w , i.e., the suffix $y\$$ of $S\$$ is mapped to the suffix $y\$^w$ of $\#S\$^w$, if y is not empty, otherwise $\$$ is mapped to $\#S\$^w$. Then, the permutation that lexicographically sorts the suffixes of $\#S\$^w$ of length greater than w , also lexicographically sorts the corresponding suffixes of $S\$$ [11, Lemma 5].

Moreover, any suffix x of $\#S\$^w$ has exactly one prefix s that is a *phrase suffix*, i.e., s is a suffix of a phrase in the dictionary $\mathcal{D}$, [11, Lemma 3]. This implies that the order among phrase suffixes can be carried over to the suffixes of $\#S\$^w$. Indeed, given two suffixes x and x' of $\#S\$^w$ (of length greater than w) and their unique corresponding prefixes s and s' that are phrase suffixes, $s \prec s'$ implies $x \prec x'$. However, if $s = s'$, some information about $\mathcal{D}$ and $\mathcal{P}$ must be taken into account.

More precisely, one can think of constructing the BWT string in two-passes according to the list of lexicographically sorted suffixes (of length greater than w) of phrases in $\mathcal{D}$, where each suffix is considered a number of times equal to its phrase frequency. In a first pass, one builds the sequence of characters preceding any suffix s of $\mathcal{D}$ that is either a proper suffix of only one phrase dictionary $d \in \mathcal{D}$ or is an element of $\mathcal{D}$ that occurs once in $\mathcal{P}$. Both cases have no ambiguity, since the preceding character is uniquely determined. In a second pass, one deals with the sequence of characters preceding suffixes that are either elements of $\mathcal{D}$ occurring more than once in $\mathcal{P}$ or proper suffixes of different dictionary phrases. In this second instance, one needs the list of lexicographically sorted suffixes of $\mathcal{P}$ to put characters in sequence according to their correct order [11, Lemma 7]. Considering the string S of Example 1 and the PFP(S) defined above, the list of lexicographically sorted suffixes of $\mathcal{D}$ of length greater than $w = 2$ is $\{\#CAT(1), A\$(1), ATA\$(1), ATGAT(2), CAT(1), GAT(2), TA\$(1), TGAT(2)\}$, where for each suffix, its phrase frequency is reported within brackets. Then, in the first pass, the sequence of characters of $\text{bwt}(S)$ built is $ATG - -\$TTAAA$, where dashes left at positions 4 and 5 correspond to $\{C, G\}$, which are the characters in S preceding the two occurrences of the phrase dictionary $ATGAT$. Finally, the $\text{bwt}(S) = ATGGC\$TTAAA$ is completed considering the lexicographic order of the suffixes of $\mathcal{P} = [0, 2, 2, 1]$ starting with 2, which is the meta-character corresponding to $ATGAT$.

In Big-BWT, the BWT of $\mathcal{P}$ is performed in linear time using the suffix array construction algorithm SACA-K [54]. However, instead of the string $\text{BWT}(\mathcal{P})$, it stores an *inverted list* that associates to each dictionary phrase d the list of positions in the $\text{BWT}(\mathcal{P})$ where d occurs. The inverted list is a format more suitable for the next step, which consists in lexicographically sorting the suffixes in $\mathcal{D}$, and then by scanning them, placing the corresponding characters in the BWT string. The sorting of the phrase suffixes is accomplished by applying the gSACA-K [48] algorithm.

Theoretical costs. The BWT computation for an input string S of length n uses $\mathcal{O}(|\mathcal{D}| + |\mathcal{P}|)$ space, where $\text{PFP}(S) = (\mathcal{D}, \mathcal{P})$, $|\mathcal{D}|$ is the sum of the length of the dictionary phrases, and $|\mathcal{P}|$ the number of elements in $\mathcal{P}$, and it takes $\mathcal{O}(n)$ time when working in internal memory.

4.2 grlBWT [Díaz-Domínguez and Navarro, 2023]

Díaz-Domínguez and Navarro presented in [22] an external memory approach that computes the BWT of a collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ of k strings and N symbols according to Equation (1). Their method, called **grlBWT**¹⁴, builds on SAIS (Section 2) but maintains data in compressed form to reduce RAM usage and redundant calculations. We refer the reader to Section 2 for the notation used in this section.

Each recursive level i of **grlBWT** receives a collection $\mathcal{S}^i$ and returns the BWT L^i of $\mathcal{S}^i$ as defined in Equation (1). To do so, it computes the set $\mathcal{F}$ of strings labelling the LMS substrings of $\mathcal{S}^i$ and uses this set to fill in a preliminary BWT $pr(L^i)$. The output is preliminary because it has some incomplete areas that could not be inferred from $\mathcal{F}$. The algorithm then produces a grammar $\mathcal{G}^i$ that generates strings in $\mathcal{F}$ to finish $pr(L^i)$ later. Subsequently, **grlBWT** replaces the LMS substrings of $\mathcal{S}^i$ by their corresponding nonterminals in $\mathcal{G}^i$ to create another collection $\mathcal{S}^{i+1}$ that passes as input to the recursive level $i+1$. The base case of the recursion is when all strings in $\mathcal{S}^i$ have length one, in which case the BWT is the input itself. When **grlBWT** returns to level i , it combines the information from $\mathcal{G}^i$ and L^{i+1} to fill the incomplete areas of $pr(L^i)$ and thus produce L^i .

In practice, **grlBWT** is an iterative method with two phases, the *parsing phase* and the *induction phase*. The former simulates the descend of the recursive levels and the second simulates the ascend.

Parsing phase. During each parsing round i , **grlBWT** *parses* $\mathcal{S}^i$ to compute $\mathcal{F}$ and then applies ISS over $\mathcal{F}$ to *simulate* the partially sorted gsa^i of $\mathcal{S}^i$ that SAIS obtains in an ISS pass. Recall from Section 2 that the ISS pass partitions gsa^i such that each block $\text{gsa}^i[s_X..e_X]$ contains the suffixes $S_u[j..n_u]$ of $\mathcal{S}^i$ prefixed by the same $X = S_u[j..nextLMS(j)]$. The algorithm builds $pr(L^i)$ using this partition and the fact that each equal-symbol run $L^i[s_X..e_X] = (c, e_X - s_X + 1)$ does not need further processing of $\text{gsa}^i[s_X..e_X]$.

The parsing round i starts by calculating $\mathcal{F}$ along with an array $W[1..|\mathcal{F}|]$ that stores in $W[u]$ the number of times $F_u \in \mathcal{F}$ appears as an LMS substring in $\mathcal{S}^i$. Subsequently, **grlBWT** runs ISS over $\mathcal{F}$ to obtain an array A^i where the suffixes of $\mathcal{F}$ are arranged in $\prec_{LMS}$ order (Equation (3)), breaking ties for equal suffixes arbitrarily. Let $A^i[q] = (j, u)$ be the q -th smallest suffix $F_u[j..n_u]$ of $\mathcal{F}$ in $\prec_{LMS}$ order. The algorithm scans A^i from left to right to visit every block $A^i[y_X..z_X]$ of suffixes in $\mathcal{F}$ with length > 1 labelled $X = F_u[j..n_u]$ and uses the information in W to compute the pair (s_X, e_X) for the block $\text{gsa}^i[s_X..e_X]$ storing the suffixes in $\mathcal{S}^i$ labelled $X = S_{u'}[j'..nextLMS(j')] = F_u[j..n_u]$, with $S_{u'} \in \mathcal{S}^i$. The algorithm skips each suffix $X = F_u[j..n_u]$ of length one because it overlaps in $\mathcal{S}^i$ the first symbol of some $F_{u'} \neq F_u \in \mathcal{F}$. When **grlBWT** reaches $A^i[y_X..z_X]$, it visits the symbol $F_u[j-1]$ that precedes $X = F_u[j..n_u]$ in each $A^i[q] = (j, u)$, with $q \in [y_X..z_X]$. If X is always a proper suffix in $\mathcal{F}$ and is preceded by the same symbol c , **grlBWT** appends a run $(c, \ell = e_X - s_X + 1)$ to $pr(L^i)$. If X is a proper suffix, but is preceded by different symbols, then it appends an incomplete block $(\#, \ell)$ instead, where $\# \notin \Sigma^i$ is a special symbol. The third option is that X matches a full string $F_u[1..n_u] \in \mathcal{F}$. In that case, **grlBWT** appends $(*, \ell)$ to $pr(L^i)$, where $* \notin \Sigma^i$ is another special symbol that indicates another type of incomplete block.

¹⁴ An implementation is available at <https://github.com/ddiazdom/grlBWT>

The next step is to build a grammar $\mathcal{G}^i = (\Sigma^i, V, \mathcal{R}, I)$ (Section 2) that generates the set $\mathcal{P}$ of different strings X producing incomplete blocks in $pr(L^i)$. Thus, for a phrase $X[1..x] \in \mathcal{P}$, grLBWT creates the rule $b \rightarrow X[1..x]$, where b is the $\prec_{LMS}$ order of X in $\mathcal{P}$. In addition, it replaces $X[1..x]$ on the right-hand side with $b \rightarrow X[j_1 - 1]b_1$, where b_1 is the $\prec_{LMS}$ order of the longest suffix $X[j_1..x]$ which is also a phrase in $\mathcal{P}$. Recursively, $b_1 \rightarrow X[j_2 - 1]b_2$ stores the second longest suffix $X[j_2..x]$ that also occurs in $\mathcal{P}$. This structure repeats o times to cover the o proper suffixes of X in $\mathcal{P}$, and the last two symbols $X[x - 1..x]$ form the rule $b_{o+1} \rightarrow X[x - 1]\epsilon$, where $\epsilon \notin \Sigma^i$ denotes the empty symbol. Let $expseq(b)$ be the sequence of pairs $(c_1, b_1), (c_2, b_2), \dots, (c_o, b_o), (c_{o+1}, \epsilon)$ in the recursive rewrites $b \Rightarrow c_1 b_1 \Rightarrow c_1 \cdot c_2 b_2 \Rightarrow \dots \Rightarrow c_1 c_2 c_{o+1}$ of $exp(b)$. Notice that every (c_u, b_u) encodes the nonterminal $b_u \in V$ expanding to the u -th suffix $X[j_u..x] \in \mathcal{P}$ of X (from left to right) and the symbol $c_u = X[j_u - 1] \in \Sigma^i$ that precedes it. Additionally, (c_{o+1}, ϵ) stores the rightmost symbol $c_{o+1} = X[x - 1]$ of X that does not overlap the following LMS substring in $\mathcal{S}^i$.

To depict the steps of the first parsing round, let us consider again the collection $\mathcal{S} = \{AGC\underline{GT}\$1, TCA\underline{AAC}\$2, CGC\underline{AA}\$3\}$ of Example 2. The underlined symbols are LMS-type positions. After computing the LMS-substrings from $\mathcal{S}^1 = \mathcal{S}$, the method obtains the set of phrases $\mathcal{F} = \{AGC, CGT\$1, TCA, AAC\$2, CGCAA\$3\}$ and the corresponding array $W[1..5] = 1, 1, 1, 1, 1$ with their frequencies in $\mathcal{S}^1$. Then, it uses ISS to compute A^1 and $pr(L^1)$ from $\mathcal{F}$ to produce:

$$\begin{array}{rcccccccccccccccccccc}
 & & \$1 & \$2 & \$3 & A\$3 & AA\$3 & AAC\$2 & AC\$2 & AGC & C\$2 & CAA\$3 & CA & CGCAA\$3 & CGT\$1 & GC AA\$3 & GC & GT\$1 & T\$1 & TCA \\
 A^1 & = & 4 & 4 & 6 & 5 & 4 & 1 & 2 & 1 & 3 & 3 & 3 & 4 & 4 & 2 & 2 & 2 & 3 & 1 \\
 & & 2 & 4 & 5 & 5 & 5 & 4 & 4 & 1 & 4 & 5 & 3 & 4 & 1 & 5 & 1 & 2 & 2 & 3
 \end{array}$$

$$pr(L^1) = \# \# \# A C * A * A G T * * C A C G *$$

Recall that $A^1[q] = (j, u)$ refers to the j -th suffix of the phrase u in $\mathcal{F}$ (from left to right). The partition of A^1 according to the suffixes of $\mathcal{F}$ becomes $A^1[1..3], A^1[4], A^1[5], \dots, A^1[18]$. After block $A^1[1..3]$, all the blocks have length one because their corresponding suffixes (in grey above) are unique in $\mathcal{F}$. Also note that the method regards $\$1 = \$2 = \$3 = \$$ as the same symbol. In $pr(L^1)$, the first block $pr(L^1)[1..3] = (\#, 3)$ is incomplete because the method cannot infer the relative order of the occurrences of $\$$ in $\mathcal{S}^1$ from A^1 and $\mathcal{F}$. Moreover, the blocks $pr(L^1)[6]$, $pr(L^1)[8]$, $pr(L^1)[12]$, $pr(L^1)[13]$, and $pr(L^1)[18]$ are also incomplete (with symbol $*$) because their corresponding suffixes match full phrases of $\mathcal{F}$. The next step is to construct the grammar $\mathcal{G}^1$ from the strings that produce incomplete blocks in $pr(L^i)$ (i.e., $\mathcal{P}$). The original rules are $\mathcal{R} = \{1 \rightarrow \$, 2 \rightarrow AAC\$2, 3 \rightarrow AGC, 4 \rightarrow CGCAA\$3, 5 \rightarrow CGT\$1, 6 \rightarrow TCA\}$. However, the method only keeps in the right-hand sides of $\mathcal{R}$ the proper suffixes associated with incomplete blocks, so the rules become $\mathcal{R} = \{1 \rightarrow \$\epsilon, 2 \rightarrow C1, 3 \rightarrow G\epsilon, 4 \rightarrow A1, 5 \rightarrow T1, 6 \rightarrow C\epsilon\}$, with ϵ being the null symbol. Finally, the parsing round stores $pr(L)^1$ and $\mathcal{G}^1$, computes the new collection $\mathcal{S}^2 = \{35, 62, 4\}$ and continues to the next parsing round.

Induction phase. Every iteration i induces the order of symbols in the incomplete blocks of $pr(L^i)$ using $\mathcal{G}^i$ and L^{i+1} . The block $pr(L^i)[s_X..e_X] = (*, \ell)$ is associated with a string $X \in \mathcal{P}$ that only labels LMS substrings of $\mathcal{S}^i$, and if $pr(L^i)[s_X..e_X]$ is the b -th incomplete run of

$pr(L^i)$, then $b \in \Sigma^{i+1}$ is the nonterminal assigned to X in $\mathcal{G}^i$. Consider the partition of L^{i+1} according to their buckets in the fully sorted version of $\mathbf{gsa}^i$. Each block $L^{i+1}[s_b..e_b]$ stores the symbols that precede the suffixes in $\mathcal{S}^{i+1}$ prefixed by b (i.e., those in $\mathbf{gsa}^{i+1}[s_b..e_b]$). There is a map from $\mathbf{gsa}^{i+1}[s_b..e_b]$ to the range $\mathbf{gsa}^i[s_X..e_X]$ with the suffixes $S_u[j..n_u]$ of $\mathcal{S}^i$ prefixed by $S_u[j..nextLMS(j)] = X$. Consequently, filling in $pr(L^i)[s_X..e_X]$ reduces to scanning $L^{i+1}[s_b..e_b]$ from left to right, and for each $L^{i+1}[q]$ in $L^{i+1}[s_b..e_b]$, run $expseq(L^{i+1}[q]) = (c_1, b_1), (c_2, b_2), \dots, (c_{o+1}, \epsilon)$ and append c_{o+1} to $pr(L^i)[s_X..e_X]$.

On the other hand, each $pr(L^i)[s_Y..e_Y] = (\#, \ell)$ is associated with a string $Y \in \mathcal{P}$ that occurs as a proper suffix in different LMS substrings of $\mathcal{S}^i$, say $X[1..x]$ and $X'[1..x']$, and is preceded by different symbols. That is, $Y = X[j..x] = X'[j'..x']$ and $X[j-1] \neq X'[j'-1] \in \Sigma^i$. If Y has $\prec_{LMS}$ order b_u in $\mathcal{P}$, then $pr(L^i)[s_Y..e_Y]$ is the b_u -th incomplete run of $pr(L^i)$ and ℓ is the number of times Y occurs as a suffix in the LMS substring of $\mathcal{S}^i$.

The encoding of $\mathcal{G}^i$ is convenient to fill $pr(L^i)[s_Y..e_Y] = (\#, \ell)$ because $expseq(L^{i+1}[q]) = (c_1, b_1), (c_2, b_2), \dots, (c_{o+1}, \epsilon)$ tells that $c_1 \in \Sigma^i$ goes to the incomplete run number b_1 , $c_2 \in \Sigma^i$ goes to the incomplete run number b_2 , and so on. Thus, a symbol c_u associated with b_u in $expseq(L^{i+1}[q])$ goes to $L^{i+1}[s_Y..e_Y]$ after the symbols associated with b_u in $expseq(L^{i+1}[1]), expseq(L^{i+1}[2]), \dots, expseq(L^{i+1}[q-1])$. In other words, it is possible to fill all incomplete blocks $(\#, \ell)$ of $pr(L^i)$ in one linear decompression of L^{i+1} . Another important observation is that a run $L^{i+1}[q..q'] = (b, \ell')$ produces ℓ' copies of $expseq(b)$. Thus, `grlBWT` calls $expseq(b)$ once and copies the content ℓ' times in each incomplete block in $expseq(b)$.

The induction round i works as follows: Create an array P with m buckets, where m is the number of incomplete blocks of $pr(L^i)$ labelled $\#$. Then, scan L^{i+1} from left to right and, for each run (b, ℓ') in L^{i+1} , execute $expseq(b) = (c_1, b_1), (c_2, b_2), \dots, (c_{o+1}, \epsilon)$ and append ℓ' copies of every symbol c_u in the bucket b_u of P . In addition, replace (b, ℓ') by (c_{o+1}, ℓ') in L^{i+1} . Next, perform a sorted merge of P , L^{i+1} , and $pr(L^i)$ (observe that the three arrays over the alphabet $\Sigma^i \cup \{*, \#\}$), replacing each $(\#, \ell)$ of $pr(L^i)$ with the next ℓ symbols of P , and each $(*, \ell)$ with the next ℓ symbols of L^{i+1} .

Let us complete the preliminary BWT $pr(L^1)$ of the parsing round 1 from our current example using the multi-string BWT $L^2 = 65234$ and the grammar $\mathcal{G}^1$. The rules of this grammar are $\mathcal{R} = \{1 \rightarrow \$\epsilon, 2 \rightarrow C1, 3 \rightarrow G\epsilon, 4 \rightarrow A1, 5 \rightarrow T1, 6 \rightarrow C\epsilon\}$. Observe that the alphabet of L^2 is a subset of the symbols on the left-hand side of $\mathcal{R}$. The method first creates the array P with one bucket as $pr(L^i)$ contains one block labelled $(\#, 3)$. The length of P is three because the cumulative length of the blocks in $pr(L^1)$ labelled $\#$ is three. The method scans L^2 from left to right to compute $expseq$ on the symbol of each run, and uses the output to fill the buckets of P . The combined information of $expseq(5) = T1 \Rightarrow T\$ \epsilon$, $expseq(2) = C1 \Rightarrow C\$1$ and $expseq(4) = A1 \Rightarrow A\$ \epsilon$ yields $P = TCA$. In turn, this information updates $pr(L^1)$ to $pr(L^1) = \underline{TCA}AC * A * AGT * * CACG*$, where the underlined symbols represent the replacement of $(\#, 3)$ by P . Note that $expseq(6)$ and $expseq(3)$ do not produce symbols for P because their suffixes do not produce incomplete blocks $(\#, \ell)$. The method also updates each $L^2[j]$ with the rightmost symbol in $expseq(L^2[j])$ that belongs to $\Sigma = \{ \$, A, C, G, T \}$, so L^2 becomes $C\$ \$ G \$$. Finally, the method merges L^2 and $pr(L^1)$ to produce the final multi-string BWT $L^1 = \underline{TCA}AC \underline{CA} \$ \underline{AGT} \$ \underline{GCACG} \$$ of $\mathcal{S}$ in Example 2. In this case, the underlined symbols originally belonged to L^2 .

Theoretical costs. Under the RAM model, `grlBWT` runs in $\mathcal{O}(N + k \log n_{max})$ time and uses $\mathcal{O}((N + k \log n_{max}) \log N)$ bits of working memory, where n_{max} is the longest string in $\mathcal{S}^i$. However, this cost is reached only with adversarial inputs, with practical scenarios (e.g., $n_{max} = \mathcal{O}(N/k)$) running in $\mathcal{O}(N)$ time. On the other hand, `grlBWT` is a semi-external

approach that keeps $\mathcal{S}^i$, $pr(L^i)$, and L^i on disk and accesses them linearly, thus requiring $\mathcal{O}((N \log N)/B)$ I/Os. On the other hand, the RAM usage depends on the size of $\mathcal{F}$. This set is small when $\mathcal{S}$ is repetitive, but there is no known theoretical bound for its size.

5 Future directions

There are several promising directions for future research. First, the efficient computation of auxiliary components in BWT-based compressed self indexes. Examples of such components are the $2r$ suffix array samples of the r -index [28], the samples of the subsampled r -index [17], or the LCP array samples of thresholds [2]. An important challenge in these examples is how to compute the data structures without building full versions of `sa` and `lcp`.

From a systems perspective, the design of disk-based algorithms optimized for Solid State Drives (SSDs) could lead to significant speedups, taking advantage of SSD-specific features such as high random access speeds and internal parallelism. Moreover, the acceleration of BWT construction via GPU architectures could offer significant performance improvements for massive datasets (see, for instance, the BWT-based compressor `libbssc`¹⁵ that uses NVIDIA GPU acceleration).

On the algorithmic side, the Prefix-Free Parsing (PFP) strategy should be further explored and adapted for the case of string collections. To our knowledge, the only PFP variant that works on collections is PFP-eBWT [10] that computes the extended BWT (EBWT) by Mantaci *et al.* [51]. On the other hand, designing new lightweight schemes for text compression that are compatible with the construction of the BWT could enhance the scalability for massive datasets. This idea is particularly relevant for `grlBWT`, where the compression of the input is the main bottleneck. Recently, Díaz-Domínguez *et al.* [21] proposed a parallel grammar algorithm that handles terabytes of data efficiently, and whose scheme shares some similarities with `grlBWT`. Adapting this technique could achieve considerable reductions in space and time.

Finally, there is growing interest in designing indexing¹⁶ data structures to encode BWTs with large alphabets (see [29, 4, 64] and references therein). The standard solution to this problem is the wavelet tree [30], which represents the BWT as a binary tree of height $\mathcal{O}(\log \sigma)$ and with σ leaves, σ being the BWT's alphabet size. However, when σ is large, the $\mathcal{O}(\log \sigma)$ factor that results from navigating the tree becomes a considerable overhead. Furthermore, the $\mathcal{O}(\sigma w)$ bits required to store the tree also make the wavelet tree impractical in this scenario. In this regard, Nishimoto *et al.* recently proposed a data structure [53] that encodes the BWT in $\mathcal{O}(r \log n)$ bits and reduces the $\mathcal{O}(\log \sigma)$ penalty to $\mathcal{O}(1)$ regardless of the alphabet. However, this solution is space-efficient only if $r \ll n$, which occurs when the text encoded by the BWT is highly repetitive. In practical scenarios where repetitive collections also have some noise, this solution grows rapidly in size [7].

6 Final Remarks

In this short survey, we highlight key algorithms for computing large BWTs by leveraging external memory or text compression techniques with no additional information about the data. It appears as a first step towards a more in-depth comparison of the corresponding tools, potentially including experimental results to highlight the differences among them.

¹⁵ <https://github.com/IlyaGrebnev/libbssc>

¹⁶ A representation encoding the BWT and that supports LF-mapping and backward search queries

■ **Table 1** Theoretical complexities of algorithms. For Big-BWT, we assume that the data structures fit into internal memory. We consider an (single) input string of length n for Big-BWT and bwt-disk, and a collection $\mathcal{S}$ of k strings with total length N for the other algorithms. Let σ be the alphabet size, m the length of the longest string in $\mathcal{S}$, maxlcp the maximum LCP value, M the number of words available in RAM, and B the number of consecutive words on the disk.

Algorithm	Time Complexity	Memory Usage	I/O Complexity
Big-BWT [11]	$\mathcal{O}(n)$	$\mathcal{O}(\mathcal{D} + \mathcal{P})$	-
bwt-disk [26]	$\mathcal{O}(n^2/M)$	M words	$\mathcal{O}(n^2/(MB \log n))$
BCR [5]	$\mathcal{O}(m(N + \text{sort}(k)))$	$\mathcal{O}(k + \sigma^2)$ words	$\mathcal{O}(Nm \log \sigma)$
BCRext [5]	$\mathcal{O}(kN)$	$\mathcal{O}(\sigma^2)$ words	$\mathcal{O}(Nm \log \sigma + N \log N)$
eGSA [49]	$\mathcal{O}((N \log N/M) \cdot \text{maxlcp})$	M words	$\mathcal{O}((N \log N/M) \cdot m)$
eGap [23]	$\mathcal{O}(N \cdot \text{maxlcp})$	M words	$\mathcal{O}(N \cdot \text{maxlcp})$
BWT+LCP [8]	$\mathcal{O}(N \cdot \text{maxlcp})$	$\mathcal{O}(k + m + \log \text{maxlcp})$	$\mathcal{O}(N \cdot \text{maxlcp})$
griBWT [22]	$\mathcal{O}(N + k \log m)$	$\mathcal{O}((N + k \log m) \log N)$ bits	$\mathcal{O}((N \log N)/B)$

However, a direct comparison may not be entirely fair, given the differences in the intended use cases of the tools. Some are designed for a single string, others for string collections, and some work in external memory (semi- or fully), while others in internal memory. Additionally, some tools are prototypes, whereas others are engineered solutions. Table 1 summarizes the complexities of the algorithms at the basis of these tools.

In particular, bwt-disk was the first tool for constructing the BWT for a single string in external memory, without the need to first compute the suffix array. It optimizes disk access by using only sequential scans, which enables it to fully leverage modern caching systems. Additionally, bwt-disk stores all input, output, and intermediate files in compressed form, which allows it to use less total disk space than the uncompressed input for real-world datasets.

Specifically, BCR/BCRext was developed to build the BWT for huge collections of short strings over any alphabet. The experimental results in [5] showed that BCR is capable of computing the BWT for 1 billion strings, each 100 characters long (approximately 93 GB of data), using only 13 GB of RAM, while BCRext requires a negligible amount of RAM. Moreover, BCR can output auxiliary data structures (such as the Document Array, Generalized Suffix Array and Longest Common Prefix).

eGSA was the first external memory algorithm for constructing the generalized suffix arrays. It can also output the BWT for collections of strings with different sizes. The experimental results in [49] showed that eGSA tool can index up to 24 GB of data using only 2GB of RAM. One disadvantage of the eGSA is the large amount of disk working space. Also, it is observed in [49] that eGSA's running time degrades when the RAM is restricted to the input size.

eGap is very flexible about the use of RAM. The first phase adapts to the available memory, but also the second one has been implemented in an alternative semi-external version. Moreover, the algorithm can output additional information (such as the Document Array) to solve in a single sequential scan of the inputs three well known problems on collections of sequences: maximal repeats, all pairs suffix-prefix overlaps and succinct de Bruijn graphs.

BWT+LCP splits the problem of computing BWTs and LCPs into subproblems in a different way from other proposals, as we explained in Section 3.5, and it shows good performance in experiments.

In the context of tools that employ compression techniques, Big-BWT has been demonstrated to reduce the workspace when computing the BWT of highly repetitive inputs. The experiments in [11] showed that the prefix-free parsing procedure implemented in Big-BWT

allows to produce a dictionary $\mathcal{D}$ and a parse $\mathcal{P}$ that are considerably smaller than the input and can fit in internal memory, even in cases where the input is of considerable size. For example, on a dataset of 10,000 *Salmonella* genomes, the dictionary and the parse together took only about 7 GB, compared to the 50 GB of uncompressed input.

The current implementation of `grlBWT` can handle high volumes of repetitive data efficiently. In experiments in [22], the tool could process 400 human genomes (1.2 TB) in 41 hours, with a memory peak of 183 GB. The performance decreased in non-repetitive data (i.e., collections of DNA sequencing reads), although it remained one of the most efficient methods. In all inputs, the main bottleneck was text compression during the parsing phase, which took more than 90% of the running time. Currently, `grlBWT` only outputs the multi-string BWT. The construction of other data structures such as `lcp` or `sa` is not yet supported.

A comparison with almost all disk-based methods is available in [23]. More comparisons can be found in the original papers, such as in [22].

Nevertheless, there are a number of other tools in the literature, some of which are optimized to work in internal memory and/or on specific alphabet (such as, for instance, the DNA alphabet). For example, the recent `ropebwt3` [45] constructs the BWT of large DNA sequence sets and enables efficient searching via the FM-index. It builds on the result reported in [26], which has seen multiple implementations [63, 59]. It uses `libsais`¹⁷ to construct a partial multi-string BWT from a sequence subset, and then merges it into an existing BWT. The tool `ropebwt3`¹⁸ is particularly optimized for highly redundant data, such as pangenomes or high-coverage sequencing reads. It encodes the BWT using run-length encoding and a dynamic B^+ -tree (see also [44]). Another recent proposal [52] (called CMS-BWT¹⁹) introduces a method that takes as input a string collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ and a reference string R , and computes the BWT of the (single) concatenated string $S_1\$S_2\$ \dots S_k\$$ by exploiting the high repetitiveness of data through the compressed matching statistics introduced in [47].

References

- 1 Donald Adjero, Timothy Bell, and Amar Mukherjee. *The Burrows–Wheeler transform: data compression, suffix arrays, and pattern matching*. Springer, Boston, MA, 2008.
- 2 Hideo Bannai, Travis Gagie, et al. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020. doi:10.1016/J.TCS.2019.08.005.
- 3 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piątkowski. Constructing the Bijective and the Extended Burrows–Wheeler Transform in Linear Time. In *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 7:1–7:16, 2021. doi:10.4230/LIPIcs.CPM.2021.7.
- 4 Jérémy Barbay, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. 21st International Symposium on Algorithms and Computation (ISAAC)*, pages 315–326, 2010. doi:10.1007/978-3-642-17514-5_27.
- 5 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483(0):134–148, 2013. doi:10.1016/j.tcs.2012.02.002.
- 6 Gianmarco Bertola, Anthony J. Cox, Veronica Guerrini, and Giovanna Rosone. A Class of Heuristics for Reducing the Number of BWT-Runs in the String Ordering Problem. In *Proc. 35th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 7:1–7:15, 2024. doi:10.4230/LIPIcs.CPM.2024.7.

¹⁷Source code: <https://github.com/IlyaGrebnev/libsais>

¹⁸<https://github.com/lh3/ropebwt3>

¹⁹<https://github.com/fmasillo/CMS-BWT>

- 7 Nico Bertram, Johannes Fischer, and Lukas Nalbach. Move-r: optimizing the r-index. In *Proc. 22nd International Symposium on Experimental Algorithms (SEA)*, pages 1:1–1:19, 2024. doi:10.4230/LIPICS.SEA.2024.1.
- 8 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Computing the multi-string BWT and LCP array in external memory. *Theoretical Computer Science*, 862:42–58, 2021. doi:10.1016/j.tcs.2020.11.041.
- 9 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 129–142, 2021. doi:10.1007/978-3-030-86692-1_11.
- 10 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proceedings of 28th International Symposium in String Processing and Information Retrieval SPIRE 2021*, volume 12944 of *Lecture Notes in Computer Science*, pages 129–142, 2021. doi:10.1007/978-3-030-86692-1_11.
- 11 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14(1):13:1–13:15, 2019. doi:10.1186/S13015-019-0148-5.
- 12 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- 13 Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT of very large string collections. In *Proc. 33rd Data Compression Conference (DCC)*, pages 71–80, 2023. doi:10.1109/DCC55655.2023.00015.
- 14 Davide Cenzato and Zsuzsanna Lipták. A survey of BWT variants for string collections. *Bioinformatics*, 40(7):btac333, 2024. doi:10.1093/bioinformatics/btac333.
- 15 Davide Cenzato, Zsuzsanna Lipták, Nadia Pisanti, Giovanna Rosone, and Marinella Sciortino. BWT for string collections. submitted to Festschrift’s honoree Giovanni Manzini.
- 16 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.
- 17 Dustin Cobas, Travis Gagie, and Gonzalo Navarro. A fast and small subsampled r-index. In *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, page article 13, 2021.
- 18 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2016. doi:10.1093/bib/bbw089.
- 19 Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012. doi:10.1093/bioinformatics/bts173.
- 20 Anthony J. Cox, Fabio Garofalo, Giovanna Rosone, and Marinella Sciortino. Lightweight LCP construction for very large collections of strings. *Journal of Discrete Algorithms*, 37:17–33, 2016. doi:10.1016/J.JDA.2016.03.003.
- 21 Diego Diaz-Dominguez. Efficient terabyte-scale text compression via stable local consistency and parallel grammar processing. *arXiv preprint arXiv:2411.12439*, 2024. to appear in Proc. 23rd Symposium on Experimental Algorithms (SEA 2025).
- 22 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. *Information and Computation*, 294:105088, 2023. doi:10.1016/j.ic.2023.105088.
- 23 Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):6:1–6:15, 2019. doi:10.1186/S13015-019-0140-0.
- 24 Lavinia Egidi and Giovanni Manzini. Lightweight merging of compressed indices based on BWT variants. *Theoretical Computer Science*, 812:214–229, 2020. doi:10.1016/j.tcs.2019.11.001.

- 25 Paolo Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.
- 26 Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012. doi:10.1007/S00453-011-9535-0.
- 27 Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Information Sciences*, 135(1):13–28, 2001. doi:10.1016/S0020-0255(01)00098-6.
- 28 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM (JACM)*, 67(1):1–54, 2020. doi:10.1145/3375890.
- 29 Alexander Golynski. Rank/select operations on large alphabets: a tool for text. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 122, page 368, 2006.
- 30 Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, et al. High-order entropy-compressed text indexes. In *Proc. 14th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, volume 3, pages 841–850, 2003.
- 31 Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone. Parallel lossy compression for large FASTQ files. In *16th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC)*, pages 97–120, 2023. doi:10.1007/978-3-031-38854-5_6.
- 32 James Holt and Leonard McMillan. Merging of multi-string BWTs with applications. *Bioinformatics*, 30(24):3524–3531, 2014. doi:10.1093/bioinformatics/btu584.
- 33 Hongwei Huo, Pengfei Liu, Chenhui Wang, Hongbo Jiang, and Jeffrey Scott Vitter. CIndex: compressed indexes for fast retrieval of FASTQ files. *Bioinformatics*, 38(2):335–343, 2021. doi:10.1093/bioinformatics/btab655.
- 34 Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.
- 35 Lilian Janin, Giovanna Rosone, and Anthony J. Cox. Adaptive reference-free compression of sequence quality scores. *Bioinformatics*, 30(1):24–30, 2014. doi:10.1093/bioinformatics/btt257.
- 36 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2009. doi:10.1007/978-3-642-02441-2_17.
- 37 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.
- 38 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.
- 39 John C. Kieffer and En Hui Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000. doi:10.1109/18.841160.
- 40 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005. doi:10.1016/j.jda.2004.08.002.
- 41 Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, Chi-Kwong Wong, and Siu-Ming Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008. doi:10.1093/bioinformatics/btn032.
- 42 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10:1–10, 2009. doi:10.1186/gb-2009-10-3-r25.
- 43 Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 2012. doi:10.1093/bioinformatics/bts280.
- 44 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014. doi:10.1093/bioinformatics/btu541.
- 45 Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40(12):btae717, 2024. doi:10.1093/bioinformatics/btae717.

- 46 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. doi:10.1093/bioinformatics/btp698.
- 47 Zsuzsanna Lipták, Francesco Masillo, and Simon J Puglisi. Suffix sorting via matching statistics. *Algorithms for Molecular Biology*, 19(1):11, 2024. doi:10.1186/s13015-023-00245-z.
- 48 Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, 678:22–39, 2017. doi:10.1016/J.TCS.2017.03.039.
- 49 Felipe A. Louza, Guilherme P. Telles, Steve Hoffmann, and Cristina Dutra de Aguiar Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms for Molecular Biology*, 12(1):26:1–26:16, 2017. doi:10.1186/S13015-017-0117-9.
- 50 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computation.*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 51 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007. doi:10.1016/j.tcs.2007.07.014.
- 52 Francesco Masillo. Matching Statistics Speed up BWT Construction. In *Proc. 31st Annual European Symposium on Algorithms (ESA 2023)*, pages 83:1–83:15, 2023. doi:10.4230/LIPIcs.ESA.2023.83.
- 53 Takaaki Nishimoto and Yasuo Tabei. Optimal-Time Queries on BWT-Runs Compressed Indexes. In *Proc. 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 198, pages 101:1–101:15, 2021. doi:10.4230/LIPICS.ICALP.2021.101.
- 54 Ge Nong. Practical linear-time $o(1)$ -workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, 31(3):15:1–15:15, 2013. doi:10.1145/2493175.2493180.
- 55 Ge Nong, Wai Hong Chan, Sheng Qing Hu, and Yi Wu. Induced sorting suffixes in external memory. *ACM Transactions on Information Systems (TOIS)*, 33(3):1–15, 2015. doi:10.1145/2699665.
- 56 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011. doi:10.1109/TC.2010.188.
- 57 Daisuke Okanohara and Kunihiko Sadakane. A linear-time Burrows–Wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 90–101, 2009. doi:10.1007/978-3-642-03784-9_9.
- 58 Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big bwts. In *2023 Data Compression Conference (DCC)*, pages 62–70, 2023. doi:10.1109/DCC55655.2023.00014.
- 59 Marco Oliva, Massimiliano Rossi, Jouni Sirén, Giovanni Manzini, Tamer Kahveci, Travis Gagie, and Christina Boucher. Efficiently merging r-indexes. In *Proc. 31st Data Compression Conference (DCC)*, pages 203–212, 2021. doi:10.1109/DCC50243.2021.00028.
- 60 Nicola Prezza, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. Variable-order reference-free variant discovery with the Burrows–Wheeler Transform. *BMC Bioinformatics*, 21-S(8):260, 2020. doi:10.1186/S12859-020-03586-3.
- 61 Yoshihiro Shibuya and Matteo Comin. Better quality score compression through sequence-based quality smoothing. *BMC Bioinformatics*, 20-S(9):302:1–302:11, 2019. doi:10.1186/s12859-019-2883-5.
- 62 Jared T. Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–56, 2012. doi:10.1101/gr.126953.111.
- 63 Jouni Sirén. Burrows–Wheeler Transform for Terabases. In *Proc. 16th Data Compression Conference (DCC)*, pages 211–220, 2016. doi:10.1109/DCC.2016.17.
- 64 Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020. doi:10.1093/BIOINFORMATICS/BTZ575.

- 65 Clare Turnbull, Richard H. Scott, Ellen Thomas, Louise Jones, Nirupa Murugaesu, Freya Boardman Pretty, Dina Halai, Emma Baple, Clare Craig, Angela Hamblin, Shirley Henderson, Christine Patch, Amanda O'Neill, Andrew Devereau, Katherine Smith, Antonio Rueda Martin, Alona Sosinsky, Ellen M. McDonagh, Razvan Sultana, Michael Mueller, Damian Smedley, Adam Toms, Lisa Dinh, Tom Fowler, Mark Bale, Tim Hubbard, Augusto Rendon, Sue Hill, and Mark J. Caulfield. The 100 000 genomes project: bringing whole genome sequencing to the nhs. *BMJ*, 361, 2018. doi:10.1136/bmj.k1687.

A Taxonomy of LCP-Array Construction Algorithms

Johannes Fischer 

TU Dortmund, Germany

Enno Ohlebusch¹ 

Ulm University, Germany

Abstract

The combination of the suffix array and the LCP-array can be used to solve many string processing problems efficiently. We review some of the most important sequential LCP-array construction algorithms in random access memory.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases longest common prefix array, suffix array, Burrows-Wheeler transform

Digital Object Identifier 10.4230/OASICS.Manzini.2025.8

1 Introduction

In a landmark paper, Weiner [43] presented the concept of suffix trees and devised the first linear-time construction algorithm. For several decades, the suffix tree was *the* data structure in stringology, because it can be used to efficiently solve a “myriad” of string processing problems [4]. For instance, with the help of the suffix tree, exact pattern matching can be done in $O(m)$ time (assuming a constant alphabet size), where m is the length of the pattern searched for. It should be pointed out that the search time is independent of the text length n .

The suffix array was devised by Manber and Myers [30] and independently by Gonnet et al. [19] under the name PAT array. Ten years later, it was shown independently and contemporaneously by Kärkkäinen and Sanders [23], Kim et al. [26], Ko and Aluru [27], and Hon et al. [20] that a direct linear-time construction of the suffix array is possible.

The LCP-array first appeared in the seminal paper of Manber and Myers [30], where it was called *Hgt* array. The authors presented an $O(n \log n)$ time LCP-array construction algorithm (LACA for short) and showed that the augmentation of classical binary search with the LCP-array enables exact pattern matching on the suffix array in $O(m + \log n)$ time. The first linear-time LACA was devised by Kasai et al. [25]. They also demonstrated the importance of the LCP-array by presenting a linear-time algorithm that simulates the bottom-up traversal of a suffix tree with a suffix array *and* the LCP-array. Their work was taken one step further by Abouelhoda et al. [1], who showed that suffix trees can be completely replaced with *enhanced* suffix arrays. An enhanced suffix array is the combination of the suffix array with augmenting data structures, which depend on the application at hand. For example, exact pattern matching can be done in $O(m)$ time with the suffix array, the LCP-array, and constant-time range minimum queries [16] on the LCP-array [1] – again assuming constant alphabet size; for non-constant alphabet sizes σ one can extend this idea to achieve $O(m \log \sigma)$ running time [15]. Algorithms on enhanced suffix arrays are not only more space efficient than those on suffix trees, but they are also faster and easier to implement. Since the LCP-array is of utmost importance in this context, it has been studied extensively.

¹ Corresponding author



© Johannes Fischer and Enno Ohlebusch;
licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini’s 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 8; pp. 8:1–8:17

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ **Table 1** Suffix array, LCP-array, and BWT of the text $S = \text{mississippi\$}$.

k	SA	LCP	BWT	$S_{\text{SA}[k]}$
1	12	$\perp$	i	$\$$
2	11	0	p	$i\$$
3	8	1	s	$ippi\$$
4	5	1	s	$issippi\$$
5	2	4	m	$ississippi\$$
6	1	0	$\$$	$mississippi\$$
7	10	0	p	$pi\$$
8	9	1	i	$ppi\$$
9	7	0	s	$sippi\$$
10	4	2	s	$sissippi\$$
11	6	1	i	$ssippi\$$
12	3	3	i	$ssissippi\$$

2 Preliminaries

Let S be a text of length n on an ordered alphabet Σ of constant size σ . We assume that S has the sentinel $\$ \in \Sigma$ at the end (and nowhere else), which is smaller than any other character. For $1 \leq i \leq n$, $S[i]$ denotes the *character at position i* in S . For $i \leq j$, $S[i, j]$ denotes the *substring* of S starting at position i and ending at position j . Furthermore, S_i denotes the i -th suffix $S[i, n]$ of S .

The *suffix array* SA of the text S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of S , that is, it satisfies $S_{\text{SA}[1]} < S_{\text{SA}[2]} < \dots < S_{\text{SA}[n]}$. The suffix array can be built in linear time; we refer to the overview article of [37] for suffix array construction algorithms and to [35] for recent developments.

The *inverse suffix array* ISA is an array of size n such that for any i with $1 \leq i \leq n$ the equality $\text{ISA}[\text{SA}[i]] = i$ holds. Obviously, it takes only linear time to invert the suffix array.

The LCP-array contains the lengths of longest common prefixes between consecutive suffixes in SA . Formally, the LCP-array is an array such that $\text{LCP}[1] = \perp$ and $\text{LCP}[i] = \text{lcp}(S_{\text{SA}[i-1]}, S_{\text{SA}[i]})$ for $2 \leq i \leq n$, where $\text{lcp}(u, v)$ denotes the length of the longest common prefix between two strings u and v . Here the value $\text{LCP}[1]$ is undefined, but – depending on the application – it may make sense to define it as 0 or -1 . Table 1 shows the LCP-array of the string $S = \text{mississippi\$}$.

The Burrows-Wheeler transform [9] converts S into the string $\text{BWT}[1, n]$ defined by $\text{BWT}[i] = S[\text{SA}[i] - 1]$ for all i with $\text{SA}[i] \neq 1$ and $\text{BWT}[i] = \$$ otherwise. Given the suffix array, this transformation takes only linear time.

The ψ -array is an array of size n such that $\psi[1] = \text{ISA}[1]$ and $\psi[i] = \text{ISA}[\text{SA}[i] + 1]$ for all i with $2 \leq i \leq n$. That is, $\psi[i]$ is the index at which the suffix $S_{\text{SA}[i]+1}$ occurs in the suffix array. The ψ -array is the inverse of the LF -array, which can easily be computed in linear time from the BWT; see e.g. [34, Section 7.2.2].

The C -array is an array of size σ . For $c \in \Sigma$, the entry $C[c]$ is defined as follows: if we consider all characters in Σ that are smaller than c , then $C[c]$ is the overall number of their occurrences in S .

■ **Algorithm 1** This procedure computes the LCP-array by Kasai et al.'s algorithm [25].

```

Input:  $S$ ,  $SA$ ,  $ISA$ 
1  $\ell \leftarrow 0$ 
2 for  $i \leftarrow 1$  to  $n$  do
3    $j \leftarrow ISA[i]$ 
4   if  $j > 1$  then
5      $k \leftarrow SA[j-1]$       /*  $S_k$  precedes  $S_i$  in  $SA$  */
6   while  $S[k+\ell] = S[i+\ell]$  do
7      $\ell \leftarrow \ell + 1$ 
8    $LCP[j] \leftarrow \ell$ 
9   if  $\ell > 0$  then
10     $\ell \leftarrow \ell - 1$ 

```

3 The First Linear Time LACA

We first present Kasai et al.'s LACA [25], which computes the LCP-array from the string S , its suffix array SA , and its inverse suffix array ISA ; see Algorithm 1. It starts with the longest suffix $S_i = S_1$ of S (so $i = 1$), computes $j = ISA[i]$ and then $LCP[j]$ by a left-to-right comparison of the characters in $S_i = S_{SA[j]}$ and its immediately preceding suffix $S_k = S_{SA[j-1]}$ in the suffix array. The same is done for the other suffixes S_i of S by incrementing i successively. As we shall see, some character comparisons can safely be skipped.

In our example, Algorithm 1 first compares $S_1 = \text{mississippi\$}$ with $S_2 = \text{ississippi\$}$ and sets $LCP[j] = LCP[6] = 0$. In the next iteration of the for-loop, it compares $S_2 = \text{ississippi\$}$ with $S_5 = \text{issippi\$}$ and sets $LCP[j] = LCP[5] = 4$. In the following iterations of the for-loop the underlined characters in $S = \text{mississippi\$}$ will be skipped in further comparisons of adjacent suffixes. For example, in the third iteration, Algorithm 1 must compare $S_3 = \text{ssissippi\$}$ with $S_6 = \text{ssippi\$}$ and the first three characters will be skipped in the comparison. To prove the correctness of Algorithm 1, we must show that if $\ell > 1$ characters match in iteration i of the for-loop, then at least $\ell - 1$ characters match in iteration $i + 1$. So suppose that the longest common prefix of S_i and its preceding suffix S_k is $c\omega$, where c is a character and ω is a string of length $\ell - 1 \geq 1$. In the next iteration, Algorithm 1 compares S_{i+1} with its preceding suffix, which is not necessarily S_{k+1} as in the example above. However, since $S_k = cwu$ is lexicographically smaller than $S_i = cwv$, it follows that $S_{k+1} = \omega u$ is lexicographically smaller than $S_{i+1} = \omega v$. Moreover, all the suffixes in between S_{k+1} and S_{i+1} must share the prefix ω because the suffixes are ordered lexicographically. In particular, S_{i+1} and its preceding suffix have ω as a common prefix. Therefore, $\ell - 1$ characters (namely ω) can be skipped in the comparison of S_{i+1} and its preceding suffix. If S_{k+1} immediately precedes S_{i+1} in the suffix array, then we can directly conclude that their longest common prefix is ω ; otherwise further character comparisons are required to determine their longest common prefix. Obviously, S_{k+1} precedes S_{i+1} if $BWT[j'] = BWT[j' - 1]$, where $j' = ISA[i + 1]$. This result is summarized in the next lemma, which will be important later. The first proof of Lemma 1 can be found in [25]. A value $LCP[j]$ is called *reducible* if $BWT[j] = BWT[j - 1]$; otherwise it is called *irreducible*.

■ **Algorithm 2** This procedure computes the LCP-array by the algorithm of Kasai et al. [25] with Manzini's [31] improvement. Initially the array LCP stores ψ . During the course of computation, the initial values in the LCP-array are overwritten by the correct values.

```

Input:  $S, SA, LCP = \psi$ 
1  $j \leftarrow LCP[1]$       /*  $LCP[1] = \psi[1] = ISA[1]$  */
2  $\ell \leftarrow 0$ 
3 for  $i \leftarrow 1$  to  $n$  do
4    $next \leftarrow LCP[j]$       /*  $LCP[j] = \psi[j] = ISA[i+1]$  */
5   if  $j > 1$  then
6      $k \leftarrow SA[j-1]$       /*  $S_k$  precedes  $S_i$  in  $SA$  */
7   while  $S[k+\ell] = S[i+\ell]$  do
8      $\ell \leftarrow \ell + 1$ 
9    $LCP[j] \leftarrow \ell$ 
10  if  $\ell > 0$  then
11     $\ell \leftarrow \ell - 1$ 
12   $j \leftarrow next$ 

```

► **Lemma 1.** Suppose that $LCP[j]$ is reducible, where $j = ISA[i]$. Then we have $LCP[j] = LCP[ISA[i-1]] - 1$.

It will now be shown that Algorithm 1 constructs the LCP-array in $O(n)$ time. We use an amortized analysis to show that the while-loop is executed at most $2n$ times. It is readily verified that this implies the linear run-time. Each comparison in the while-loop ends with a mismatch, so there are $n - 1$ mismatches (redundant character comparisons) in total. If a position p in S is involved in a match, then this particular occurrence of character $S[p]$ will be skipped in further suffix comparisons. More precisely, if we have $S[k+\ell] = S[i+\ell]$ in the while-loop, then $p = i + \ell$ will not appear on the right-hand-side of a character comparison again. So there are at most n matches. To illustrate the argument, the positions in our example text that are involved in a match are marked by underlining the respective character in $S = \text{mississippi}\$$. Since every position is marked at most once (because it is skipped in further comparisons), it is clear that the overall number of character comparisons is $n - 1$ (mismatches) plus the number of underlined characters.

In Algorithm 1, the text S as well as the arrays SA , ISA , and LCP should be randomly accessible. Under the common assumption that an entry in SA takes 4 bytes, each of the arrays SA , ISA , and LCP requires $4n$ bytes. Consequently, for an 8 bit alphabet, Algorithm 1 needs $13n$ bytes of working memory.

In the context of compressed suffix trees, Mäkinen [29, Proposition 3.2] first showed that the LCP-array can be constructed with less working space. Manzini [31] observed that the additional array ISA is superfluous because it is possible to store the required information about ISA in the LCP-array itself. If the LCP-array initially stores the ψ -array, then the cell $LCP[j]$ contains the value $ISA[i+1]$ in iteration i of the for-loop of Algorithm 2. That is, variable $next$ gets the value $ISA[i+1]$ in line 4 and $LCP[j]$ can safely be overwritten in line 9. The correctness of Algorithm 2 follows directly from this fact. Algorithm 2 requires $9n$ bytes of working memory because it needs random access to the text and both input arrays.

Manzini [31] proposed another LACA, which saves even more space by overwriting the suffix array. This LACA is based on Lemma 1: if $LCP[j]$ is reducible, then one can immediately set $LCP[j] \leftarrow \ell$ because the character comparisons in lines 7-8 of Algorithm 2 are superfluous. It follows as a consequence that in this case the assignment $k \leftarrow SA[j-1]$ is

i	1	2	3	4	5	6	7	8	9	10	11	12
$S[i]$	m	i	s	s	i	s	s	i	p	p	i	$\$$
$\Phi[i]$	2	5	6	7	8	4	9	11	10	1	12	0
PLCP[i]	0	4	3	2	1	1	0	1	1	0	0	$\perp$

■ **Figure 1** Φ -array and PLCP-array of $S = \text{mississippi}\$$; cf. Table 1.

superfluous, too. In other words, the value $\text{SA}[j - 1]$ is required only if $\text{LCP}[j]$ is irreducible. In a preprocessing phase, Manzini's second LACA determines the values in the suffix array SA that are actually required and stores them in an auxiliary array. The auxiliary array is then used instead of SA and SA can safely be overwritten. However, this LACA is of limited value in practice because in most applications the suffix array is needed as well.

4 The Φ -Algorithm

Kärkkäinen, Manzini, and Puglisi [22] proposed a variant of Kasai et al.'s algorithm, which first computes a permuted LCP-array (the PLCP-array) with the help of the so-called Φ -array and then derives the LCP-array from the PLCP-array. They called it " Φ -algorithm" because it uses the Φ -array, which "is in some way symmetric to the ψ array."

The Φ -array and the PLCP-array, respectively, are defined as follows: For all i with $1 \leq i \leq n$ let

$$\Phi[i] = \begin{cases} \text{SA}[\text{ISA}[i] - 1] & \text{if } \text{ISA}[i] \neq 1 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \text{PLCP}[i] = \begin{cases} \text{lcp}(S_i, S_{\Phi[i]}) & \text{if } \Phi[i] \neq 0 \\ \perp & \text{otherwise} \end{cases}$$

Fig. 1 shows our example. So in the suffix array SA , the suffix S_i is immediately preceded by the suffix $S_{\Phi[i]}$ unless $\Phi[i] = 0$. Note that we have $\Phi[n] = 0$ because we assume that S is terminated by $\$$. For $j \neq 1$, we have

$$\text{PLCP}[\text{SA}[j]] = \text{lcp}(S_{\text{SA}[j]}, S_{\Phi[\text{SA}[j]]}) = \text{lcp}(S_{\text{SA}[j]}, S_{\text{SA}[j-1]}) = \text{LCP}[j] \quad (1)$$

In case $j = 1$, Equation (1) holds also true: $\text{LCP}[1] = \perp$ and $\text{PLCP}[\text{SA}[1]] = \text{PLCP}[n] = \perp$ because $\Phi[n] = 0$. Thus, the PLCP-array is a permutation of the LCP-array. The difference between the two arrays is that the lcp-values occur in text order (position order) in the PLCP-array, whereas they occur in suffix array order (lexicographic order) in the LCP-array. The pseudocode of the Φ -algorithm is shown in Algorithm 3. Its properties (correctness, linear run-time) can be shown as in the analysis of Algorithm 1.

Experimental comparisons of Algorithm 1 and the Φ -algorithm can be found in [22, 18]. The bottom line is that the Φ -algorithm is faster because of better memory locality: it merely needs sequential access to the Φ -array and the PLCP-array in its second for-loop. Consequently, these arrays can be streamed and the number of possible cache misses is limited to $3n$: n in the computation of the Φ -array, n in line 6,² and n in the conversion of the PLCP-array into the LCP-array (in virtually all applications lcp-values are required to be in suffix array order). By contrast, there can be up to $4n$ cache misses in Algorithm 1 (including the computation of ISA).

² Remark: If $\text{PLCP}[i]$ is reducible, then $\Phi[i] = \Phi[i - 1] + 1$ and hence $S[\Phi[i]]$ is cached.

■ **Algorithm 3** The Φ -algorithm of Kärkkäinen et al. [22].

```

Input:  $S, SA$ 
1 for  $i \leftarrow 2$  to  $n$  do
2    $\Phi[SA[i]] \leftarrow SA[i - 1]$ 
3  $\ell \leftarrow 0$ 
4 for  $i \leftarrow 1$  to  $n - 1$  do
5    $k \leftarrow \Phi[i]$  /*  $S_k$  precedes  $S_i$  in  $SA$  */
6   while  $S[k + \ell] = S[i + \ell]$  do
7      $\ell \leftarrow \ell + 1$ 
8    $PLCP[i] \leftarrow \ell$ 
9   if  $\ell > 0$  then
10     $\ell \leftarrow \ell - 1$ 
11 for  $i \leftarrow 2$  to  $n$  do
12    $LCP[i] \leftarrow PLCP[SA[i]]$ 

```

If we assume that the arrays SA , Φ , $PLCP$, and LCP are kept in working memory, then Algorithm 3 requires $17n$ bytes RAM. However, the algorithm merely needs random access to the Φ -array in the first for-loop, to the text in the second for-loop, and to the $PLCP$ -array in the third for-loop. All other arrays can be streamed from or to disk. Thus, $9n$ bytes of RAM are sufficient.

Kärkkäinen et al. [22] presented another algorithm for computing the $PLCP$ -array based on irreducible $PLCP$ -values. We call a value $PLCP[i]$ *reducible* if $S[i - 1] = S[\Phi[i] - 1]$; otherwise it is called *irreducible*. The proof of the next lemma is similar to that of Lemma 1.

► **Lemma 2.** *If $PLCP[i]$ is reducible, then $PLCP[i] = PLCP[i - 1] - 1$.*

The above-mentioned algorithm first determines all irreducible $PLCP$ -values in a brute-force manner: each $PLCP$ -value is computed by comparing the respective suffixes from the very beginning. Afterwards, it uses Lemma 2 to compute the remaining reducible $PLCP$ -values. The authors showed that the sum of all irreducible $PLCP$ -values is at most $2n \log n$. Hence, the overall run-time is $O(n \log n)$. An experimental comparison of this algorithm and the Φ -algorithm showed that the Φ -algorithm is faster in practice; see [22].

5 Computing the LCP-Array along with SA

The LCP-array can also be computed while sorting the suffixes lexicographically, which we show here for two well-known suffix array construction algorithms DC3 [24] and SAIS [33]. The adaptations to the LCP-array computation were described in the original article for DC3, and by Bingmann et al. for SAIS [8]. Intuitively, it seems clear that LCP-array computation during suffix array construction should be possible, as any suffix sorting algorithm must, at least implicitly, determine the lcp-values between lexicographically adjacent suffixes to determine their order, for otherwise it could not make the right decisions on the order of suffixes.

5.1 A Unified Perspective

We first look at the common features that both algorithms share, which form the basis for the LCP-computation. We assume knowledge about DC3 and SAIS, as a full exposition of those algorithms would take too much space in this article. We also leave out some details of

LCP-array construction and refer the readers to the original articles at various points; the focus here is more on a common exposition of how the computations of lcp-values can be woven into suffix sorting algorithms.

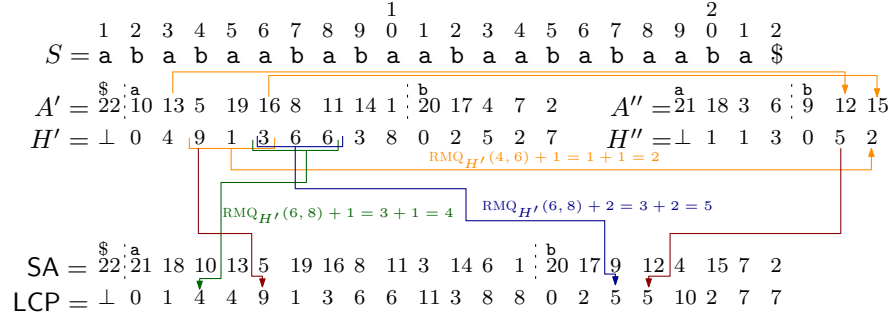
Both DC3 and SAIS are recursive algorithms, and for sorting all the suffixes of a text $S[1, n]$ they first sort a *subset* of $n' \leq n/\Omega(1)$ suffixes recursively, stored in an array $A'[1, n']$ such that $S_{A'[i]} < S_{A'[i+1]}$ for all $1 \leq i < n'$. From A' , they use different mechanisms to compute the full suffix array $SA[1, n]$. In DC3, A' contains all suffixes at positions $i \not\equiv 0 \pmod{3}$ (see top of Fig. 2), whereas in SAIS, A' contains those positions i with $S_{i-1} > S_i < S_{i+1}$ (called LMS-suffixes for *leftmost smallest suffixes*; see top of Fig. 3 with LMS-suffixes indicated by a star). In both algorithms, we talk about “ c -buckets” for an arbitrary character $c \in \Sigma$, by which we mean the subarray consisting of the suffixes starting with $c \in \Sigma$ as in counting or bucket sort – in the figures those buckets are depicted by dotted lines.

By induction, we can assume that we also get the LCP-array H' for all the lexicographically adjacent suffixes in A' from the recursion: $H'[i] = \text{lcp}(S_{A'[i]}, S_{A'[i-1]})$. The details of how to compute H' differ between the two algorithms and are a bit tedious, but are not the core of the algorithmic ideas. The reason why this is not trivial is that because A' has been computed recursively from the suffix array of a reduced text S' (each character in S' representing several characters in S – 3 in DC3 (hence the name), and a non-constant number in SAIS), the lcp-values need to be *scaled* to actually represent the number of characters of shared prefixes in S , and not S' . We assume here that this scaling step has already been performed – see again Fig. 2 for DC3 and Fig. 3 for SAIS, where, in the latter case, both arrays A' and H' are already stored in the final arrays SA and LCP , as the algorithm works in-place (skip the gray values in LCP for this moment, as they will be written in the course of the algorithm that follows).

Then both SAIS and DC3 compute the suffix array SA from A' , and the algorithms can be enhanced to compute the full LCP-array from H' . Both algorithms finally place suffixes into SA in a left-to-right manner (for SAIS, this happens in actually *two* passes, the second one right-to-left but symmetrically). Say the algorithm just wrote $SA[x] \leftarrow i$ and now needs to compute the lcp-value h between S_i and its lexicographic predecessor S_j at position $x - 1$ in SA ; the result h will have to be stored at $LCP[x] \leftarrow h$. As the values in SA are filled from left to right, the value $SA[x - 1] = j$ has been set in a previous iteration of this loop. (In SAIS, this left-to-right filling is per bucket, but the principle remains the same, as only suffixes in the same c -bucket have an lcp-value > 0 .)

The easy case is if both i and j were neighbors in A' ; then we know their lcp-values and we can just copy the lcp-value from H' . Consider, for example, the placement of suffix $i = 5$ and $j = 13$ at positions $x = 6$ and $x - 1 = 5$ in Fig. 2: those suffixes are also neighbors in A' at positions 4 and 3, respectively, and hence the lcp-value $H'[4] = 9$ can be copied to $LCP[6] \leftarrow 9$ (left red arrow). The same situation might occur in SAIS, e.g. in Fig. 3 suffix $i = 9$ is placed to the right of $j = 6$, whose lcp-value of 2 can just be copied from H' .

The more difficult case is when i and j were not neighbors in A' . Here, the techniques differ between the algorithms, but they share some common ideas. We first compare the two *characters* $S[i]$ and $S[j]$ (if in DC3 $i + 1 \equiv 0 \pmod{3}$, we also compare $S[i + 1]$ and $S[j + 1]$). If the 1 or 2 compared characters are all the same (otherwise the lcp-value h has been determined), it is clear that the value $h = \text{lcp}(S_i, S_j)$ must be one more than h' , where $h' = \text{lcp}(S_{i+1}, S_{j+1})$ (possibly two more than $h'' = \text{lcp}(S_{i+2}, S_{j+2})$ in DC3, see above). Now h' (or h'') can be computed by a *range minimum query* on H' , which gives the minimum value between two specified array indices. More formally, the range minimum



■ **Figure 2** Example of LCP-array computation with the DC3 suffix sorting algorithm.

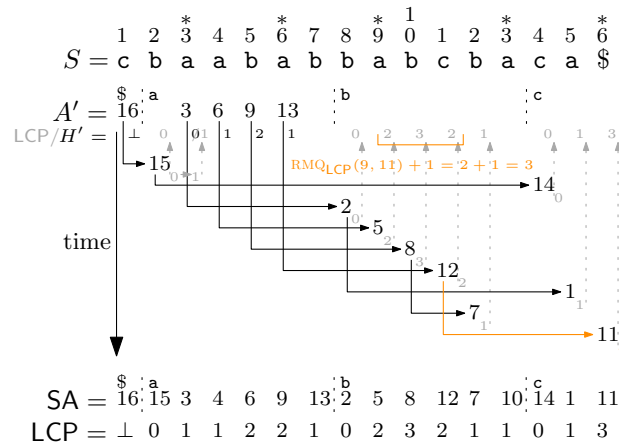
query (RMQ) problem on a general array $X[1, m]$ is to find a data structure such that later, for two specified indices ℓ, r with $1 \leq \ell \leq r \leq m$, the minimum in $X[\ell, r]$ can be returned efficiently, in symbols: $RMQ_X(\ell, r) = \min\{X[k] : \ell \leq k \leq r\}$. For these queries, linear preprocessing schemes exist that can answer the queries in constant time [16]. Armed with this tool, in DC3 we can simply compute $h' = RMQ_{H'}(A'^{-1}[i + 1] + 1, A'^{-1}[j + 1])$ (or $h'' = RMQ_{H'}(A'^{-1}[i + 2] + 1, A'^{-1}[j + 2])$), where A'^{-1} is the inverse of A' , defined by $A'[A'^{-1}[p]] = p$ for all values p actually stored in $A'[1, n']$ – note that for all the positions p queried we have $p \not\equiv 0 \pmod{3}$; this is also why the distinction between h' and h'' has been made.

For example, in Fig. 2, when writing $SA[4] \leftarrow 10$ to the right of $SA[3] = 18$, we have $A'^{-1}[11] = 8$ and $A'^{-1}[19] = 5$ and hence compute $h' = RMQ_{H'}(6, 8) = 3$ and thus finally write $LCP[4] = 3 + 1$ (green color). When writing $SA[17] \leftarrow 9$ to the right of $SA[16] = 17$, we compute $h'' = RMQ_{H'}(6, 8) = 3$ and thus finally write $LCP[17] = 3 + 2$ (blue color).

The justification of this approach is the following: Let $Y[1, m]$ be an arbitrary array storing strings in lexicographically sorted order, and let $X[2, m]$ be an array such that $X[i]$ stores the lcp-value between strings $Y[i]$ and $Y[i - 1]$ for all $2 \leq i \leq m$. Then due to the lexicographic sortedness of Y , the lcp-value between two arbitrary strings $Y[z]$ and $Y[y]$ is the minimum value from $X[z + 1], \dots, X[y]$ (assuming $z < y$), in symbols: $\text{lcp}(Y[z], Y[y]) = RMQ_X(z + 1, y)$. In our case, we exploit this fact with $Y[i] = S_{A'[i]}$ and $X[i] = H'[i]$.

In SAIS, the situation is different, as there is (most likely) no constant K such that $i + K$ and $j + K$ are both in A' . Instead, we make use of the fact that (in the left-to-right scan) both suffixes S_{i+1} and S_{j+1} are lexicographically smaller than S_i and S_j , respectively, and are thus already present in SA (say at positions z with $SA[z] = i + 1$ and $y < z$ with $SA[y] = j + 1$), including the lcp-values with their respective lexicographic predecessors and all lcp-values in between (the entire subarray $LCP[y, z]$ has already been written). So $h' = RMQ_{LCP}(y + 1, z)$ can be easily computed in the same manner as in DC3. Look at the computation of the values $SA[16]$ and $LCP[16]$ in Fig. 3 (orange color): We write $SA[16] \leftarrow 11$ when the scanning of SA reaches position $z = 11$ with $SA[11] = 12$. As $j = SA[15] = 1$, which was written when at position $y = 8$, we need to set $LCP[16]$ to the lcp-value of suffix S_{11} and its lexicographic predecessor S_1 . The values $LCP[8, 11]$ are all known (gray values in H' below the b -bucket); hence we can perform an RMQ from positions 9 to 11 in LCP , which gives a 2. We thus set $LCP[16] \leftarrow 2 + 1 = 3$.

Commenting on the differences between the RMQs in DC3 and SAIS, we can say that in DC3 the RMQs are performed on the static array H' (for which optimal preprocessing schemes exist), whereas for SAIS the array LCP on which RMQs are performed grows semi-dynamically from the left to the right. We comment below how this can be done efficiently.



■ **Figure 3** Example of LCP-array computation with the SAIS suffix sorting algorithm. Note that this only shows the left-to-right scan – in a subsequent right-to-left scan the missing suffixes 10 and 4 including the affected lcp-values would also be considered; the final result is shown at the bottom.

5.2 DC3

The above description of DC3 skipped one important detail of its inner workings: when it returns from the recursion, before it finally computes the full suffix array **SA**, it first needs to compute an array A'' containing the lexicographic order of the suffixes starting at positions $i \equiv 0 \pmod{3}$. This is done by a left-to-right scan through A' : for $z = 1, \dots, n' \approx \frac{2n}{3}$ in ascending order, if $A'[z] = i + 1 \equiv 1 \pmod{3}$, place i to the first free place p in A'' 's c -bucket for $c = S[i]$. (By “first free position” we mean the first position not already written in that bucket, as in classical counting- or bucket-sort algorithms.)

Having computed A'' from A' , we now explain how to compute array H'' storing the lcp-values of lexicographically adjacent suffixes in A'' . When placing i to $A''[p]$, let $A''[p-1] = j$ be its immediate predecessor in A'' in the c -bucket (first suffixes in a bucket are simple as their lcp-values are 0). Since both S_{i+1} and S_{j+1} are suffixes in A' , we can compute $H''[p] = 1 + h''$, where $h'' = \text{RMQ}_{H'}(A'^{-1}[j] + 1, A'^{-1}[i])$. Note that this is another usage of the RMQ data structure on H' that needs to be precomputed anyway for the later steps.

For example, in Fig. 2, when we write the final value $A''[7] \leftarrow 15$ next to $A''[6] = 12$, we have $A'^{-1}[13] = 3$ and $A'^{-1}[16] = 6$, and hence compute $H''[7]$ as $\text{RMQ}_{H'}(4, 6) + 1 = 1 + 1 = 2$ (orange color).

The rest of the algorithm works exactly as explained above – with the addition that if two suffixes from A'' are placed adjacently into **SA**, we can look up their lcp-value directly in H'' . In more detail, the final computation of **SA** can be viewed as merging the arrays A' and A'' , as they are both sorted lexicographically and, together, contain all the suffixes from S . Hence, if in this merging the value i is written to $\text{SA}[x]$ and we already know $\text{SA}[x-1] = j$ with both $i, j \equiv 0 \pmod{3}$, their lcp-value can be directly retrieved from H'' and written to **LCP**, as i and j are neighbors in A'' . See Fig. 2 for an example, where the value $\text{SA}[18] = 12$ is written right after $\text{SA}[17] = 9$, both of which come from A'' at adjacent positions 5 and 6 with $H''[6] = 5$ (right red arrow). We note here that neither A'' nor H'' need be materialized, but can be computed on the fly during the merging process.

5.3 SAIS

We now give more details about the LCP-array computation during the SAIS-algorithm. Let us first consider the *scaling* of the lcp-values to compute H' . As a reminder, the suffixes in A' (those positions i with $S_{i-1} > S_i < S_{i+1}$) are called *LMS-suffixes*, including the sentinel character $\$$ at the end. The recursion works by computing the suffix array of a reduced text with meta-characters ranging from one LMS-suffix to the next. In our example, the reduced text is $T' = [\text{aaba}][\text{abba}][\text{abcba}][\text{aba}\$][\$]$ (with square brackets indicating the meta-characters), and the recursion stops immediately with $A' = [16, 3, 6, 9, 13]$, as all meta-characters are different. Now in order to see the issue with LCP-computation, look at the suffixes S_6 and S_9 from A' : in A' , their order was determined from the order of the meta-characters $S[6, 9] = [\text{abba}]$ and $S[9, 13] = [\text{abcba}]$, where the first one is smaller, and the lcp-value of the suffixes – in terms of meta-characters – is 0. But the meta-characters *themselves* share a common prefix, **ab** in this case. Hence, the correct lcp-value for suffixes S_6 and S_9 is 2. Here, we do not give the full details of how the array H' is actually computed correctly, but refer the reader to [8, Lemma 3.1] for the correct way to do it, and to [14, Sect. 3.3] for the way how to compute this efficiently.

For what follows, we also need to classify the other suffixes: if $S_i > S_{i+1}$, then S_i is called *L-suffix* (for “larger”); if $S_{i-1} < S_i < S_{i+1}$, then S_i is called *S-suffix* (for “smaller”). The left-to-right pass (as shown in the example in Fig. 3) actually sorts the L-suffixes from the LMS-suffixes (whose order has been computed recursively), whereas the subsequent right-to-left pass sorts the S-suffixes into the already sorted LMS-suffixes. In a c -bucket in SA for a character $c \in \Sigma$, all the L-suffixes come first, followed by a mix of S- and LMS-suffixes. E.g., at the bottom of Fig. 3 one can see that in the **a**-bucket the only L-suffix S_{15} comes first, followed by the LMS-suffix S_3 , an S-suffix S_4 , followed by three more LMS-suffixes S_3 , S_9 , S_{13} in this order.

A further detail is the computation of the lcp-value between the last L-suffix in a bucket and its lexicographic successor (which is either S or LMS). In Fig. 3, this happens in the **a**-bucket, where the L-suffix S_{15} is placed right before the LMS-suffix S_3 , which had an lcp-value of 0 in H' (stored at $\text{LCP}[3]$) because it was the lexicographically first suffix starting with an **a** among those in A' . However, with the placement of $\text{SA}[2] \leftarrow 15$, the value $\text{LCP}[3]$ is not 0 anymore, but needs to be updated to 1 (second gray arrow from the left). It can be shown [8, Lemma 3.2] that in the c -bucket only the character c , possibly repeated many times, can contribute to the lcp-value of those suffixes. Hence, a *naive* LCP-computation per bucket suffices to compute these lcp-values, as any character from S can contribute at most once to this naive computation (in its corresponding bucket).

The last issue is that the RMQs needed for LCP-computation are performed on an array LCP where values are constantly being written to, no data structures are known for constant-time range minimum queries for dynamic arrays. However, the updates occur in a regular manner, namely from left to right per bucket in the left-to-right scan (and again, symmetrically from right to left in the right-to-left scan). We sketch here the solution for such *semi-dynamic RMQs* [8, Sect. 3.2] based on *LRM-trees* [5, Def. 1] – also known under the slightly misleading name *2d-Min-Heaps* [16, Def. 5.3]. We maintain an LRM-tree T_c for each bucket c , which initially contains only the LMS-suffixes of that bucket with their respective lcp-values (as computed in the recursive call). When a new L-suffix along with lcp-value h is written into its c -bucket, we climb up the rightmost path of T_c until we find an element x whose corresponding LCP-entry is strictly smaller than h (T_c has an artificial root holding lcp-value $-\infty$, which guarantees that such an element always exists). The new element is then added as x ’s new rightmost leaf; an easy amortized argument shows that

■ **Table 2** BWT of $S = \text{mississippi}\$$ and its partially filled LCP-array.

i	LCP	BWT	$S_{\text{SA}[i]}$
1	$\perp$	i	$\$$
2	0 $\nparallel$	p	$i\$$
3	$\perp$	s	$ippi\$$
4	$\perp$	s	$issippi\$$
5	$\perp$	m	$ississippi\$$
6	0 $\nparallel$	$\$$	$mississippi\$$
7	0 $\nparallel$	p	$pi\$$
8	$\perp$	i	$ppi\$$
9	0 $\nparallel$	s	$sippi\$$
10	$\perp$	s	$ssissippi\$$
11	$\perp$	i	$ssippi\$$
12	$\perp$	i	$ssissippi\$$
13	-1		

this results in overall linear time. T_c is stored with a data structure for constant-time lowest common ancestor queries (LCAs) that supports dynamic leaf additions in $O(1)$ worst-case time [10]. Then the minimum in any range in the processed portion of the c -bucket can be found in $O(1)$ time [16, Lemma 5.5] by LCA-queries on T_c .

Interestingly, this solution for RMQs on semi-dynamic arrays is the only scenario known to the authors where LRM-trees are actually advantageous over the better known Cartesian Trees [42, Sect. 3.1], as only in LRM-trees appending elements to the array results in pure leaf additions to the tree. In a Cartesian Tree, appending a single new element might result in more complicated relinking operations on the tree topology (be it in constant time), and no data structure for constant-time LCAs is known in fully dynamic trees. A final caveat is that one would probably not want to implement this approach due to the complicated data structure for dynamic LCAs – see [8, Sect. 3.2] for alternatives that work better in practice.

6 Computing the LCP-array from the BWT

Next, we will present a LACA that is based on the BWT of S [7]. We start with some prerequisites.

Let ω be a substring of S . The ω -interval is the interval $[i, j]$ such that ω is a common prefix of $S_{\text{SA}[i]}, S_{\text{SA}[i+1]}, \dots, S_{\text{SA}[j]}$, but neither of $S_{\text{SA}[i-1]}$ nor of $S_{\text{SA}[j+1]}$. For example, in Table 2 one can see that the *miss*-interval is $[6, 6]$, while the *iss*-interval is $[4, 5]$. Since the empty string ε is a common prefix of all suffixes of S , the ε -interval is $[1, n]$. The LACA in Algorithm 4 is based on the procedure *getIntervals*, which has the following functionality: For an ω -interval $[i, j]$, the call *getIntervals* $([i, j])$ returns the list of all $c\omega$ -intervals, where $c \in \Sigma$ and $c\omega$ is a substring of S . In our example, *getIntervals* applied to the ε -interval $[1, 12]$ generates the $\$$ -interval $[1, 1]$, the i -interval $[2, 5]$, the m -interval $[6, 6]$, the p -interval $[7, 8]$ and the s -interval $[9, 12]$. Beller et al. [7] describe an implementation of the procedure *getIntervals* that is based on the wavelet tree of the BWT of S . A call to *getIntervals* $([i, j])$ takes $O(k \log \sigma)$ time if it returns a list with k intervals (so each interval can be generated in $O(\log \sigma)$ time).

■ **Algorithm 4** Computation of the LCP-array based on the BWT [7].

Input: wavelet tree of the BWT

```

1 initialize the array LCP[1, n + 1] by LCP[i] ← ⊥ for all i
2 ℓ ← −1
3 initialize an empty queue Q
4 enqueue(Q, [1, n])      /* add the ε-interval to Q */
5 size ← 1                /* initial size of Q */
6 while Q is not empty do
7   if size = 0 then
8     ℓ ← ℓ + 1
9     size ← |Q|          /* current size of Q */
10  [lb, rb] ← dequeue(Q)
11  size ← size − 1
12  if LCP[rb + 1] = ⊥ then
13    LCP[rb + 1] ← ℓ      /* case 1 */
14    list ← getIntervals([lb, rb])
15    foreach [i, j] in list do
16      enqueue(Q, [i, j])
17  else
18    nothing to do        /* case 2 */

```

Algorithm 4 shows how the LCP-array of a string S can be obtained solely based on the procedure *getIntervals*. The algorithm maintains a queue Q , which initially contains the ε -interval $[1, n]$. Moreover, the variable ℓ stores the current lcp-value (initially, $\ell = -1$) and $size$ memorizes how many elements there are in Q that correspond to the current lcp-value (initially, $size = 1$). Algorithm 4 computes lcp-values in increasing order (first the artificial entry $LCP[n + 1] = -1$, then the 0 entries, and so on) as follows: whenever it dequeues an element from Q , say the ω -interval $[lb, rb]$ where $|\omega| = \ell + 1$, it tests whether $LCP[rb + 1] = \perp$. If so, it assigns ℓ to $LCP[rb + 1]$, generates all non-empty $c\omega$ -intervals (where $c \in \Sigma$) and adds them to (the back of) Q . Otherwise, it does nothing.

Let us illustrate the algorithm by computing the artificial -1 entry and all 0 entries in the LCP-array of our example in Table 2. Initially, $\ell = -1$ and the queue Q contains the ε -interval $[1, 12]$. Consequently, the first interval that is pulled from the queue is the ε -interval $[1, 12]$ and $size$ is decreased by one. Since $LCP[12 + 1] = \perp$, case 1 in Algorithm 4 applies. Thus, $LCP[13]$ is set to $\ell = -1$ and *getIntervals* $([1, 12])$ generates the $\$$ -interval $[1, 1]$, the i -interval $[2, 5]$, the m -interval $[6, 6]$, the p -interval $[7, 8]$ and the s -interval $[9, 12]$. These intervals are put into the queue Q . In the second iteration of the while-loop, we have $size = 0$. Therefore, $\ell = -1$ is increased by one and the variable $size$ gets the new value $|Q|$. At that point, the $size$ of Q is 5; so five intervals correspond to the new lcp-value $\ell = 0$. The second interval that is pulled from the queue is the $\$$ -interval $[1, 1]$ and $size$ is decreased by one. Since $LCP[1 + 1] = \perp$, case 1 in Algorithm 4 applies. Thus, $LCP[2]$ is set to $\ell = 0$ and the $i\$$ -interval $[2, 2]$, which is the only interval in the list returned by *getIntervals* $([1, 1])$, is added to the queue. Next, the i -interval $[2, 5]$ is dequeued (and $size$ is decreased by one). Again, case 1 applies because $LCP[5 + 1] = \perp$. So $LCP[6]$ is set to $\ell = 0$, *getIntervals* $([2, 5])$ returns the list $[[6, 6], [7, 7], [9, 10]]$, and the intervals in the list are added to the queue Q . When the m -interval $[6, 6]$ is dequeued, $size$ is decreased by one, $LCP[7]$ is set to $\ell = 0$, but no new interval is added to the queue (observe that *getIntervals* does not generate the $\$m$ -interval

because Sm is not a substring of S). Then the p -interval is dequeued, $size$ is decreased by one, $LCP[9]$ is set to 0, and the intervals $[3, 3]$ and $[8, 8]$ (the ip - and the pp -interval) are enqueued. Finally, the s -interval $[9, 12]$ is pulled from the queue. Again, $size$ is decreased by one; it now has the value 0. Because $LCP[12 + 1] = -1$, case 2 in Algorithm 4 applies. In the next iteration of the while-loop, ℓ is increased by one and $size$ is set to the current size 6 of Q . At that point in time, the six elements in Q are the following intervals that correspond to the lcp-value $\ell = 1$:

$$[2, 2]_{i\$}, [6, 6]_{mi}, [7, 7]_{pi}, [9, 10]_{si}, [3, 3]_{ip}, [8, 8]_{pp}$$

where the notation $[lb, rb]_\omega$ indicates that the interval $[lb, rb]$ is the ω -interval. The reader is invited to compute all 1 entries in the LCP-array by executing the algorithm by hand.

► **Theorem 3.** *Algorithm 4 correctly computes the LCP-array.*

Proof. We proceed by induction on ℓ . After the first iteration of the while-loop (which leads to the artificial entry $LCP[n + 1] = -1$), we have $\ell = 0$ and the queue Q contains the c -interval $[lb, rb]$ for every character $c = \Sigma[k]$, where $lb = C[c] + 1$ and $rb = C[d]$ with $d = \Sigma[k + 1]$. The algorithm sets $LCP[rb + 1] = 0$ unless $rb = n$. This is certainly correct because the suffix $S_{SA[rb]}$ starts with the character c and the suffix $S_{SA[rb+1]}$ starts with the character d . Clearly, the LCP-array contains all entries with value 0. Let $\ell > 0$. By the inductive hypothesis, we may assume that Algorithm 4 has correctly computed all lcp-values $< \ell$. Immediately after the value of ℓ was incremented in line 8 of Algorithm 4, the queue Q solely contains intervals that correspond to the lcp-value ℓ (i.e., maximal intervals in which the suffixes share a common prefix of length $\ell + 1$). Let the ωa -interval $[lb, rb]$ be in Q , where $|\omega| = \ell$ and $a \in \Sigma$. If $LCP[rb + 1] = \perp$, then we know from the induction hypothesis that $LCP[rb + 1] \geq \ell$, i.e., ω is a common prefix of the suffixes $S_{SA[rb]}$ and $S_{SA[rb+1]}$. On the other hand, ωa is a prefix of $S_{SA[rb]}$ but not of $S_{SA[rb+1]}$. Consequently, ω is the longest common prefix of $S_{SA[rb]}$ and $S_{SA[rb+1]}$. Hence Algorithm 4 assigns the correct value ℓ to $LCP[rb + 1]$.

We still have to prove that all entries of the LCP-array with value ℓ are really set. So let k , $0 \leq k < n$, be an index with $LCP[k + 1] = \ell$. Since $\ell > 0$, the longest common prefix of $S_{SA[k]}$ and $S_{SA[k+1]}$ can be written as $c\omega$, where $c \in \Sigma$, $\omega \in \Sigma^*$, and $|\omega| = \ell - 1$. Let $c\omega a$ be the length $\ell + 1$ prefix of $S_{SA[k]}$ and $c\omega d$ be the length $\ell + 1$ prefix of $S_{SA[k+1]}$, where $a \neq d$. Because ωa is a prefix of $S_{SA[k]+1}$ and ωd is a prefix of $S_{SA[k+1]+1}$, it follows that ω is the longest common prefix of $S_{SA[k]+1}$ and $S_{SA[k+1]+1}$. Let $[i, j]$ be the ω -interval, p be the index with $SA[p] = SA[k] + 1$, and q be the index with $SA[q] = SA[k + 1] + 1$. Clearly, $i \leq p < q \leq j$. Let t , $p < t \leq q$, be the smallest index at which the corresponding suffix does not start with ωa ; see Fig. 4. Consequently, $LCP[t] = |\omega| = \ell - 1$. According to the inductive hypothesis, Algorithm 4 assigns the value $\ell - 1$ to $LCP[t]$. Therefore, *getIntervals* is called with the ωa -interval $[s, t - 1]$ for some $s \leq t$. Since ωa is a prefix of $S_{SA[p]}$ and $BWT[p] = c$, it follows that the $c\omega a$ -interval, say $[lb, rb]$, is not empty. Moreover, $rb = k$ because $BWT[r] \neq c$ for all $p < r < q$. Thus, $[lb, k]$ is in the *list* returned by *getIntervals* $([s, t - 1])$. Hence it is added to the queue Q . At some point in time, $[lb, k]$ will be removed from Q and $LCP[k + 1]$ will be set to ℓ . ◀

► **Theorem 4.** *Algorithm 4 has a worst-case time complexity of $O(n \log \sigma)$.*

Proof. We use an amortized analysis to prove that each of the cases 1 and 2 can occur at most n times. Case 1 occurs as often as an entry of the LCP-array is filled, and this happens exactly n times. It remains for us to analyze how often case 2 can occur. We claim that

i	$\text{LCP}[i]$	$\text{BWT}[i]$	$S_{\text{SA}[i]}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
p		c	$\omega a \dots$
$\vdots$			$\omega a \dots$
t	$\ell - 1$		$\omega b \dots$
$\vdots$			
q		c	$\omega d \dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k			$c \omega a \dots$
$k + 1$	ℓ		$c \omega d \dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

■ **Figure 4** Correctness of Algorithm 4.

for a fixed position j , $1 \leq j \leq n$, there is at most one substring $\omega = S[i, j]$ ending at j for which the ω -interval $[lb, rb]$ belongs to case 2. If i is the largest position with $\omega = S[i, j]$ such that the ω -interval $[lb, rb]$ belongs to case 2, then none of the left-extensions of ω is generated. More precisely, none of the ω' -intervals, where $\omega' = S[i', j]$ with $1 \leq i' < i$, will be enqueued. This proves the claim. As there are only n possibilities for j , it follows that case 2 occurs at most n times. In summary, the procedure *getIntervals* can create at most $2n$ intervals because every interval belongs to exactly one case. Each interval can be generated in $O(\log \sigma)$ time, so the run-time of Algorithm 4 is $O(n \log \sigma)$. ◀

It is shown in [7] that Algorithm 4 can be implemented in such a way that it needs only $4n$ bits for the storage of the intervals in the queue. Furthermore, $n \log \sigma + o(n \log \sigma)$ bits are required for the wavelet tree and $4n$ bytes are needed for the LCP-array itself. Under the assumption that $\log \sigma = 8$, the algorithm requires $5.5n$ bytes plus $o(n)$ bits of RAM in total.

In essence, Algorithm 4 enumerates all substring-intervals in a breadth-first manner and Beller et al. [7] demonstrated how this can be done space efficiently. Belazzougui [6] showed that the enumeration can also be done space-efficiently in a depth-first manner. Prezza and Rosone [36] analyzed the space consumption of the two alternatives and came to the conclusion that Beller et al.'s algorithm is more space-efficient on large alphabets whereas Belazzougui's algorithm is more space-efficient on small alphabets. Prezza and Rosone designed an algorithm which is the combination of the two alternatives: if the alphabet size σ is larger than $\frac{\sqrt{n}}{\log^2 n}$, they apply the algorithm of Beller et al. and otherwise they use Belazzougui's algorithm to compute the LCP-array. The resulting algorithm fills the LCP-array in $O(n \log \sigma)$ time, using only $o(n \log \sigma)$ bits of working space on top of the BWT and the LCP-array [36, Theorem 3].

7 Final Remarks

We have only scratched the surface of LCP-array construction and reviewed some classic algorithms, but we did not cover results on the efficient storage of lcp-values, such as sampled [41] or compressed [13, 39] LCP-arrays. There are also variants of LCP-arrays, such as for labeled graphs [3], spectra [2], or Wheeler DFAs [11], or with mismatches [32], which were all not covered here. We also did not look at more advanced models of computation, like parallel shared or distributed memory [17, 40] or external memory [8, 21]. Finally, LCP-arrays for

collections of strings have also been considered [12, 28]. All citations in this subsection should only be seen as a point of entry for further literature research, as they are far from exhaustive. We close this review article by mentioning that for large repetitive data sets the currently best practical algorithm is based on *prefix free parsing* [38, Lemma 2].

References

- 1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.
- 2 Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Longest common prefix arrays for succinct k-spectra. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2023. doi:10.1007/978-3-031-43980-3_1.
- 3 Jarno N. Alanko, Davide Cenzato, Nicola Cotumaccio, Sung-Hwan Kim, Giovanni Manzini, and Nicola Prezza. Computing the LCP array of a labeled graph. In Shunsuke Inenaga and Simon J. Puglisi, editors, *35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, June 25-27, 2024, Fukuoka, Japan*, volume 296 of *LIPICs*, pages 1:1–1:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICs.CPM.2024.1.
- 4 Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer, 1985.
- 5 Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theor. Comput. Sci.*, 459:26–41, 2012. doi:10.1016/J.TCS.2012.08.010.
- 6 D. Belazzougui. Linear time construction of compressed text indices in compact space. In *Proc. 46th Annual ACM Symposium on Theory of Computing*, pages 148–193, 2014.
- 7 Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. *Journal of Discrete Algorithms*, 18:22–31, 2013. doi:10.1016/J.JDA.2012.07.007.
- 8 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. doi:10.1145/2975593.
- 9 M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center, 1994.
- 10 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.
- 11 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on Wheeler DFAs. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2023, Snowbird, UT, USA, March 21-24, 2023*, pages 150–159. IEEE, 2023. doi:10.1109/DCC55655.2023.00023.
- 12 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. doi:10.1186/S13015-019-0140-0.
- 13 Johannes Fischer. Wee LCP. *Inf. Process. Lett.*, 110(8-9):317–320, 2010. doi:10.1016/J.IPL.2010.02.010.
- 14 Johannes Fischer. Inducing the LCP-array. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer, 2011. doi:10.1007/978-3-642-22300-6_32.
- 15 Johannes Fischer and Volker Heun. Finding range minima in the middle: Approximations and applications. *Math. Comput. Sci.*, 3(1):17–30, 2010. doi:10.1007/S11786-009-0007-8.

- 16 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.
- 17 Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In Jackie Kern and Jeffrey S. Vetter, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 16:1–16:10. ACM, 2015. doi:10.1145/2807591.2807609.
- 18 Simon Gog and Enno Ohlebusch. Compressed suffix trees: Efficient computation and storage of LCP-values. *Journal of Experimental Algorithmics*, 18:2.1, 2013. doi:10.1145/2444016.2461327.
- 19 Gaston H. Gonnet, Ricardo Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- 20 Wing-Kai Hon, Kunihiro Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 251–260, 2003. doi:10.1109/SFCS.2003.1238199.
- 21 Juha Kärkkäinen and Dominik Kempa. Better external memory LCP array construction. *ACM J. Exp. Algorithmics*, 24(1):1.3:1–1.3:27, 2019. doi:10.1145/3297723.
- 22 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching*, volume 5577 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2009. doi:10.1007/978-3-642-02441-2_17.
- 23 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003. doi:10.1007/3-540-45061-0_73.
- 24 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.
- 25 Toru Kasai, Hiroki Arimura, Gunho Lee, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001. doi:10.1007/3-540-48194-X_17.
- 26 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer-Verlag, 2003. doi:10.1007/3-540-44888-8_14.
- 27 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, 2003. doi:10.1007/3-540-44888-8_15.
- 28 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. doi:10.1186/S13015-017-0117-9.
- 29 Veli Mäkinen. Compact suffix array – a space-efficient full-text index. *Fundamenta Informaticae*, 56(1-2):191–210, 2003. URL: <http://content.iospress.com/articles/fundamenta-informaticae/fi56-1-2-12>.
- 30 Udi Manber and Gerne Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 31 Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 372–383. Springer, 2004. doi:10.1007/978-3-540-27810-8_32.
- 32 Giovanni Manzini. Longest common prefix with mismatches. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*,

- volume 9309 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2015. doi:10.1007/978-3-319-23826-5_29.
- 33 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. doi:10.1109/TC.2010.188.
 - 34 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.
 - 35 Jannik Olbrich, Enno Ohlebusch, and Thomas Büchler. Generic non-recursive suffix array construction. *ACM Transactions on Algorithms*, 20(2), 2024. doi:10.1145/3641854.
 - 36 Nicola Prezza and Giovanna Rosone. Space-efficient computation of the LCP array from the Burrows-Wheeler transform. In *Proc. 30th Annual Symposium on Combinatorial Pattern Matching*, volume 128 of *Leibniz International Proceedings in Informatics*, pages 7:1–7:18, 2019. doi:10.4230/LIPICS.CPM.2019.7.
 - 37 Simon J. Puglisi, Bill Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4, 2007.
 - 38 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. doi:10.1089/CMB.2021.0290.
 - 39 Kunihiro Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.
 - 40 Julian Shun. Fast parallel computation of longest common prefixes. In Trish Damkroger and Jack J. Dongarra, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 387–398. IEEE Computer Society, 2014. doi:10.1109/SC.2014.37.
 - 41 Jouni Sirén. Sampled longest common prefix array. In Amihood Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, volume 6129 of *Lecture Notes in Computer Science*, pages 227–237. Springer, 2010. doi:10.1007/978-3-642-13509-5_21.
 - 42 Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980. doi:10.1145/358841.358852.
 - 43 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

Search Schemes for Approximate Pattern Matching: An Overview

Lore Depuydt 

Department of Information Technology, Ghent University - imec, Belgium

Jan Fostier 

Department of Information Technology, Ghent University - imec, Belgium

Simon Gottlieb 

FB Mathematik & Informatik, Freie Universität Berlin, Germany

Gregory Kucherov 

LIGM, CNRS, Université Gustave Eiffel, Marne-la-Vallée, France

Knut Reinert 

FB Mathematik & Informatik, Freie Universität Berlin, Germany

Luca Renders 

Department of Information Technology, Ghent University - imec, Belgium

Abstract

We provide a brief survey of results on solving the approximate pattern matching problem using search schemes, as introduced by Kucherov et al. (2016). We demonstrate that search schemes constitute a flexible and versatile tool that enable the specification of various search strategies, including several known filtering methods. We present approaches for designing efficient search schemes and for implementing them effectively. Finally, we conclude with experimental results comparing multiple search schemes on DNA sequencing data using the Columba software by Renders et al. (2021).

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases FM-index, bidirectional index, approximate pattern matching, search scheme

Digital Object Identifier 10.4230/OASICS.Manzini.2025.9

Supplementary Material *Software*: <https://github.com/biointec/columba>
archived at `swb:1:dir:3dc8bae72bcd433f3e2ff1cdf0016666a5b6e816`

Funding *Lore Depuydt*: Ph.D. Fellowship FR (1117322N), Research Foundation - Flanders (FWO).
Luca Renders: Ph.D. Fellowship SB (1SE7822N), Research Foundation - Flanders (FWO).

1 Introduction

Quickly locating a given string (pattern) in a long sequence is a canonical algorithmic problem with numerous practical applications. In bioinformatics, for example, an important task is to locate thousands of *reads* generated by DNA sequencing in a genomic sequence. In this example, as well as in most other practical settings, it is necessary to allow for a certain number of *errors*, i.e., differences between the pattern and its occurrence in the sequence. This leads to the problem of *approximate pattern matching* (APM) that we address in this paper. A common algorithmic formalization, which we follow in this paper, assumes that we are given an error threshold k and that we are looking for *all* occurrences of a given pattern up to k errors in a text. The error model is defined either by the Hamming distance, which considers only character substitutions, or by the edit (Levenshtein) distance, which also allows character insertions and deletions.



© Lore Depuydt, Jan Fostier, Simon Gottlieb, Gregory Kucherov, Knut Reinert, and Luca Renders;
licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 9; pp. 9:1–9:16

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Pattern matching is a problem that has been extensively studied. One class of solutions, relevant for most practical applications, is *indexed pattern matching*, where the text is preprocessed into an index data structure that supports efficient pattern search. Among such data structures, the *FM-index*, also known as the BWT-index, is particularly important. It was proposed by Ferragina and Manzini in 2000 [7]. Unlike other popular data structures such as *suffix trees* [43, 28, 40] and *suffix arrays* [26], the FM-index is a succinct data structure whose size (in bits) is asymptotically the same as that of the underlying sequence. Interestingly, the FM-index is built upon the *Burrows-Wheeler transform* [5], which is rooted in word combinatorics [8] and was initially proposed for text compression [27]. However, the idea proved extremely useful for indexing: the FM-index and its variants became integral components of many bioinformatics tools, such as Bowtie [19, 18], BWA and BWA-SW [20, 21], SOAP2 [22], Masai [37], Centrifuge [12], FMAAlign [23, 44], Centrifuger [38], or ProPhyle [3].

In contrast to exact pattern matching, which can be solved very efficiently, approximate pattern matching with k errors is a more challenging problem, as known algorithms are, in the worst case, exponential in k with respect to either time or space requirements [39]. Considerable effort has been made to develop practically efficient APM algorithms. One proposed heuristic is *seeding*, based on the observation that approximately matching fragments typically share exact patterns of certain size. The idea is then to use small exactly matching patterns, or their combinations, as *seeds* for potential enclosing approximate matches. Another perspective on this approach is *filtering* where seeds are viewed as indicators of potential match locations in the sequence, filtering out irrelevant regions and narrowing the search. To efficiently detect seeds, all patterns in the text that could serve as seeds are indexed in a data structure, typically a hash table. The most prominent example of such algorithms is the BLAST alignment method, which was a predominant tool in bioinformatics for many years [13]. While seeding is generally not used for exhaustive search, it can also be applied to APM in a non-trivial way through the concept of *spaced seeds* [4, 24, 6]. This approach allows for defining one or several seeds such that *any* pattern occurrence contains at least one of them. However, computing spaced seeds that ensure a lossless search is a challenging problem, even for the Hamming distance [14]. For the edit distance, this approach becomes even more difficult to implement effectively.

In this paper, we focus on a different approach that leverages efficient indexing of the text via an FM-index. Although the standard method of pattern search using an FM-index scans the pattern backward (right-to-left), extensions of the FM-index, known as the *bidirectional FM-index*, support searching in both directions [17, 35, 36, 2, 1, 30]. This feature can be advantageous for approximate pattern matching, as first demonstrated by Lam et al. [17] for restricted cases of one or two errors. The idea of combining forward and backward searches for APM was also explored in [19] and [20], although those works employed a separate index for each direction.

In general, the idea involves partitioning the pattern into $k + 1$ or more parts and breaking the search into independent searches, which can potentially run in parallel. Each search begins with one of the parts, allowing either no errors or a small number of errors. This approach drastically reduces the search space compared to the naive *backtracking* method where the pattern is searched in one direction while exploring all possible errors at each step. This idea is formalized through the concept of *search schemes* which we study in this paper. Search schemes constitute a flexible and versatile formalism for specifying search strategies. They can also be used to simulate some existing filtering methods, such as the one based on the pigeonhole principle, the suffix filter [10] or the 01*0 filter [42].

The goal of this paper is to provide an overview of search schemes, relate it to other methods, and present its current state-of-the-art. We will also survey some techniques for designing efficient search schemes. Finally, we provide some experimental results demonstrating and comparing the efficiency of different search schemes.

2 Definitions

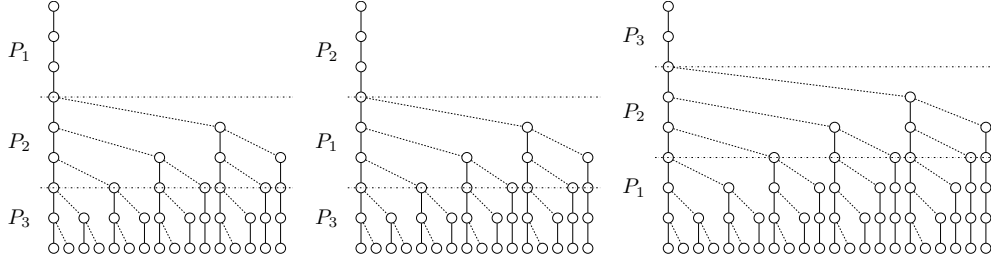
A string is denoted as $A = a_1 a_2 \dots a_{|A|}$ where $|A|$ represents its length, and a_i refers to the i -th character of the string. An infix (or substring) of A is denoted as $A_{i,j} = a_i a_{i+1} \dots a_j$. Given an error threshold k , the approximate pattern matching problem for a query pattern $P = p_1 p_2 \dots p_{|P|}$ and a text $T = t_1 t_2 \dots t_n$ is defined as finding the set of all substrings $\{T_{s_1, e_1}, T_{s_2, e_2}, \dots\}$ such that $d(T_{s_i, e_i}, P) \leq k$ for some distance measure $d(\cdot, \cdot)$. In this work, we consider either the Hamming distance d_{ham} or the edit distance d_{edit} . The Hamming distance $d_{\text{ham}}(X, Y)$ between two strings X and Y of equal length is the number of positions at which the corresponding symbols are different. The edit distance $d_{\text{edit}}(X, Y)$ between two strings X and Y is the minimum number of operations required to transform X into Y , where an operation is an insertion, deletion, or substitution of a single character.

A *bidirectional index* of a text is a data structure for pattern matching that supports incremental search of the pattern in both directions. This allows a pattern to be searched starting from any position, extending either to the left or right, and possibly alternating directions. More specifically, given an already matched substring M , the index supports extending that match to either Mc or cM , where c is a character.

Assume a query pattern P is divided into p non-overlapping parts $P = P_1 \dots P_p$. A *search* is a triplet $S = (\pi, L, U)$. Here, $\pi = (\pi[1], \dots, \pi[p])$ is a permutation of $\{1, \dots, p\}$ defining the *search order* in which these parts will be matched. To make it consistent with bidirectional indices, it is required that π fulfills the *connectivity property*: for each $i > 1$, π_i is either $\min_{j < i} \pi_j - 1$, or $\max_{j < i} \pi_j + 1$. $L = L[1] \dots L[p]$ and $U = U[1] \dots U[p]$ are non-negative integer sequences of length p that specify respectively lower and upper bounds on the cumulative number of errors when searching for consecutive parts in the order specified by π . Formally, L and U must satisfy $L[1] \leq \dots \leq L[p] \leq k$, $U[1] \leq \dots \leq U[p] \leq k$, and $L[i] \leq U[i]$ for all $i \leq p$. A *search scheme* $\mathcal{S}$ is a collection of searches $\mathcal{S} = \{S_1, S_2, \dots, S_{|\mathcal{S}|}\}$. For ease of reading, sequences π , L and U are written as strings without separators, for example search $((2, 3, 1), (0, 0, 0), (0, 1, 2))$ is written as $(231, 000, 012)$.

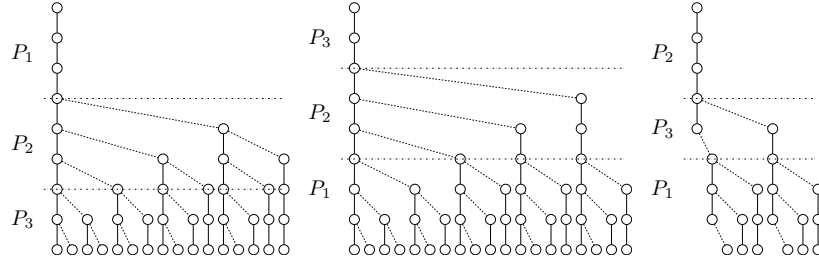
An *error configuration* $e = (e_1, \dots, e_p)$ is a distribution of at most k errors over the p parts, i.e. $\sum_{i=1}^p e_i \leq k$. A search $S = (\pi, L, U)$ *covers* e if $L[i] \leq \sum_{j \leq i} e_{\pi[j]} \leq U[i]$ for all i . A search scheme is called *lossless* if each possible error configuration is covered by at least one of its searches. Furthermore, if each error configuration is covered by only one search, it is called *non-redundant*. In the case of Hamming distance, a *lossless* and *non-redundant* search scheme will lead to each match being found exactly once. In contrast, in the case of edit distance, different error configurations can actually correspond to the same match.

An example of a search scheme is given in Fig. 1. The scheme contains three searches for a pattern of length 8 over a binary alphabet, with up to $k = 2$ errors under the Hamming distance. The pattern is partitioned into three parts of length 3, 3, 2 respectively. Note that scheme S_2 requires bidirectional search: after initially searching part P_2 , part P_1 is searched backward and then P_3 is searched forward. The scheme simulates the well-known pigeonhole principle for APM when the pattern is partitioned into $k + 1$ parts and an exact occurrence of each part is then extended into a potential match (see Section 3). Observe that the scheme is lossless but redundant.



■ **Figure 1** Pigeonhole principle expressed as a search scheme $\mathcal{S}_{\text{ph}} = \{S_1 = (123, 000, 022), S_2 = (213, 000, 022), S_3 = (321, 000, 022)\}$ with $k = 2$ errors, under the Hamming distance d_{ham} . The pattern $P = P_1P_2P_3$, defined over a binary alphabet, has length $|P| = 8$ and part lengths $|P_1| = 3$, $|P_2| = 3$, and $|P_3| = 2$. A vertical edge (solid line) represents a character match between the query pattern and the text, while a skewed edge (dotted line) denotes a mismatch. The horizontal lines delineate the parts of P .

Fig. 2 depicts another lossless search scheme for the search setting of Fig. 1. This scheme follows the algorithm of Lam et al. [17]. The third search S_{bi} covers error configuration $(1, 0, 1)$, the only one not covered by S_{fwd} and S_{bwd} . One can observe that $\mathcal{S}_{\text{Lam}}$ has a smaller search space than the search scheme from Fig. 1.



■ **Figure 2** A search scheme described by Kucherov et al. [15] simulating the search strategy proposed by Lam et al. [17]: $\mathcal{S}_{\text{Lam}} = \{S_{\text{fwd}} = (123, 000, 022), S_{\text{bwd}} = (321, 000, 012), S_{\text{bi}} = (231, 012, 012)\}$. The parameter setting is the same as in Fig. 1.

3 Search schemes

To gain intuition about what constitutes an efficient search scheme, we first consider naive backtracking as a baseline algorithm. Naive backtracking can be expressed as a search scheme with a single search $\mathcal{S}_{\text{bt}} = \{(1, 0, k)\}$, where pattern P is not partitioned ($p = 1$) and is matched in its entirety, with a lower bound of 0 and an upper bound of k errors. Using an index such as the FM-index, candidate occurrences O of the pattern P in the text T are incrementally enumerated character by character in a depth-first manner. For simplicity, we assume that O is spelled from left to right. The number of errors between a (partial) candidate occurrence O and (a prefix of) P is tracked using a dynamic programming matrix D , where each matrix element $D(i, j)$ represents $d(O_{1,i}, P_{1,j})$, i.e., the distance between the first i characters of O and the first j characters of P . As soon as all values in row $|O|$ of D exceed the threshold k , it is impossible to further extend O to match P within k errors. In this case, the algorithm backtracks to the most recent branching point with unexplored characters and continues the search from there. If P can be completely matched (i.e., $d(O, P) \leq k$), then O constitutes an approximate occurrence of P in T and its position(s) in T can be

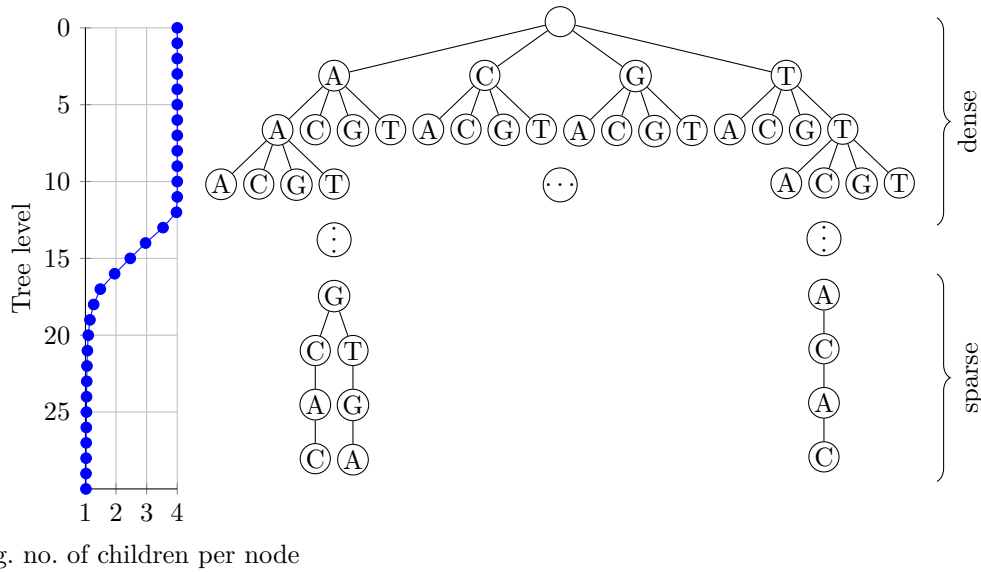


Figure 3 Left panel: average number of children per node at different levels in the substring trie of the human reference genome (GRCh38). Short sequences (length ≤ 12) can be extended by all four nucleotides. In contrast, longer sequences (length > 20) have a unique extension. Right panel: illustration of the corresponding substring trie which is dense near its root and becomes increasingly sparse at deeper levels.

reported. Note that in the case of the edit distance, maintaining a banded matrix D with $2k + 1$ entries per row suffices. In the case of the Hamming distance, it is sufficient to track only of the diagonal elements of D .

The naive backtracking algorithm induces a search tree where each node corresponds to a single character extension and, therefore, to a unique substring O of the text that is enumerated. Since the FM-index enables character extensions in constant time, the runtime of naive backtracking is proportional to the number of nodes in this search tree. The size of the search tree, i.e., the total number of strings O enumerated, is governed by two factors: (i) the string O should exist in T , and (ii) either $d(O, P) \leq k$ (and then O should be reported as an occurrence), or it must still be possible to further extend O to some O' , such that $d(O', P) \leq k$. Criterion (i) is controlled by the FM-index which spells out only candidate occurrences that exist in T . For (ii), the (banded) dynamic programming (DP) matrix D is used to track the distance between a (partial) candidate occurrence O and pattern P . Using naive backtracking, the search tree grows rapidly with the maximum number of allowed errors k and becomes impractically large even for modest values of k . This is because the region near the root of the search tree is densely branched. This is illustrated in Fig. 3 showing the trie of all substrings of the human reference genome. Each search enumerates a part of this trie, and a large number of short strings O are enumerated only to find out that the vast majority of them cannot be extended to an actual (approximate) occurrence of P in T . The reason that many short strings are enumerated is because short strings O are likely to be present in T and not (yet) exceed the threshold of k errors.

One way to avoid exploring the densely branched region near the root of the substring trie is to use the pigeonhole scheme $\mathcal{S}_{\text{ph}}$ illustrated in Fig. 1. If k errors are allowed, the pattern P is partitioned into $p = k + 1$ parts $P = P_1 \dots P_p$. Regardless of how the k errors are distributed across the parts, at least one part must be error-free. Consequently, each

approximate occurrence O of P with at most k errors must contain at least one such error-free part of P . The key idea is to first match this error-free part of P , and then to extend this match with the remaining parts of P using backtracking, allowing up to k errors in total. To identify all occurrences, this procedure must be performed p times, each time assuming a different part of P to be error-free. In each such search, the initial exact matching bypasses the densely branched upper levels of the substring trie, and the backtracking procedure is performed only in deeper levels which are more sparsely branched (nodes have fewer possible character extensions, see Fig. 3). This pigeonhole-based approach can be formally expressed as a search scheme with $p = k + 1$ searches: $\mathcal{S}_{\text{ph}} = \{S_1, \dots, S_{k+1}\}$, where

$$S_i = (i \dots p(i-1) \dots 1, 0 \dots 0, 0k \dots k) \quad \text{for } i = 1 \dots k+1. \quad (1)$$

Formula (1) expresses that during search S_i , part P_i is matched first with no errors allowed. Next, this seed is extended with parts $P_{i+1}, \dots, P_p$ to the right, followed with parts $P_{i-1}, \dots, P_1$ to the left, allowing up to k errors in total. Collectively, searches S_i of $\mathcal{S}_{\text{ph}}$ will identify all approximate occurrences O of P . However, the same occurrence may be reported by multiple searches. Indeed, any occurrence with more than one error-free part will be reported by multiple searches. The pigeonhole-based search has been applied in many earlier works (see e.g. [29]). Note that searches S_i with $i = 2 \dots p-1$ require *bidirectional* search that supports extending a partial occurrence O with a single character c either to the left (cO) or to the right (Oc).

Although conceptually simple, the pigeonhole search scheme can already outperform naive backtracking by a significant margin in practical applications. Yet, each search in the pigeonhole search scheme already admits the maximum number of k errors starting from the second part that is searched. Intuitively, performance could benefit from only gradually allowing more errors when additional parts are matched. Indeed, deeper levels in the substring trie are associated with fewer branches (see Fig. 3), so admitting more errors is best postponed as much as possible.

Kärkkäinen and Na [10] proposed a *suffix filter* that enhances the pigeonhole method. The pattern is still partitioned into $p = k + 1$ parts $P = P_1 \dots P_p$, but the search is now performed for *strongly matching* suffixes $P_i \dots P_p$. A suffix $P_i \dots P_p$ is said to strongly match a string $O = O_i \dots O_p$ (suffix of a potential approximate match $O = O_1 \dots O_p$) if for any j with $i \leq j \leq p$, the condition $d(P_i \dots P_j, O_i \dots O_j) \leq j - i$ holds. It was proven in [10] that this filter is lossless, meaning that for any error configuration, there always exists a strongly matching suffix. Note that the distance function *dist* can be either d_{ham} or d_{edit} .

It is easily seen that the suffix filter can be simulated by a search scheme that first searches forward for a strongly matching suffix and then extends it backward to the beginning of P . The idea of combining suffix filter with the search on the FM-index was exploited in [25, 41, 16] in application to DNA read alignment problems. Formally, the search scheme for suffix filter is defined by $\mathcal{S}_{\text{suf}} = \{S_1, \dots, S_{k+1}\}$, where

$$S_i = (i \dots p(i-1) \dots 1, 0 \dots 0, 01 \dots (p-i)k \dots k) \quad \text{for } i = 1 \dots k+1. \quad (2)$$

Another improvement of the pigeonhole-based approach was proposed by Vroland et al. [42]. Here, pattern P is partitioned into $p = k + 2$ parts instead of $k + 1$. This guarantees that any approximate occurrence O of P contains at least two error-free parts. Among these pairs, there exists one separated by a sequence of zero or more parts each containing exactly one error [42, Lemma 2]. This (variable-length) sequence of parts is referred to as a 01^*0 seed. Compared to the suffix filter, the seeding of the 01^*0 filter requires searching for parts with no more than one error, which contributes to its efficiency. Unfortunately, a direct

simulation of the 01^*0 filter with a search scheme requires $(k+2)(k+1)/2$ searches. However, as observed by Pockrandt [31], one can simulate a weaker version by searching for an exactly matching part P_i followed by part P_{i+1} matching with at most one error. This requires only $k+1$ searches, where the last $(k+1)$ -th search can be restricted to be followed by an exactly matching part P_{k+2} . In summary, this search scheme is defined by $\mathcal{S}_{01} = \{S_1, \dots, S_{k+1}\}$, where

$$S_i = \begin{cases} (i \dots p(i-1) \dots 1, 0 \dots 0, 01k \dots k) & \text{for } i = 1 \dots k \\ (i \dots p(i-1) \dots 1, 0 \dots 0, 00k \dots k) & \text{for } i = k+1 \end{cases} \quad (3)$$

Compared to the pigeonhole scheme (1) which uses $k+1$ parts, $\mathcal{S}_{01}$ uses $k+2$ parts which are thus slightly shorter, making the starting exact match of P_i a slightly weaker filter. However, the requirement that the following part is matched with at most one error outweighs this weakness in many practical scenarios, as will be shown below in Section 4.

In their seminal paper, Kucherov et al. [15] provided search schemes with $k+1$ and $k+2$ parts for up to $k=4$ errors. They differ from $\mathcal{S}_{ph}$, $\mathcal{S}_{suf}$ and $\mathcal{S}_{01}$ in several aspects. First, they often contain a larger number of searches. For example, for $k=4$ errors and $p=k+1=5$ parts, $\mathcal{S}_{Kuch} = \{S_1, \dots, S_8\}$ contains eight searches¹:

$$\begin{aligned} S_1 &= (12345, 00000, 02244); & S_2 &= (54321, 00000, 01344); \\ S_3 &= (21345, 01333, 01334); & S_4 &= (12345, 01333, 01334); \\ S_5 &= (43521, 00111, 01244); & S_6 &= (32145, 00113, 01244); \\ S_7 &= (21345, 01224, 01244); & S_8 &= (12345, 00334, 00444); \end{aligned} \quad (4)$$

Increasing the number of searches reduces the search space per search and enables a more gradual increase in the allowed number of errors as additional parts are added. Indeed, in search scheme (4), most searches allow the maximum number of $k=4$ errors only from the fourth part that is matched. Kucherov et al. [15] define the *critical string* of a search scheme $\mathcal{S}$ as the lexicographically maximal U -string of a search in $\mathcal{S}$. The critical U -string in $\mathcal{S}_{Kuch}$ (for $k=4$ errors and $p=5$ parts) is 02244 from search S_1 . This is more favorable than 04444, the critical U -string for the pigeonhole-based search scheme $\mathcal{S}_{ph}$. Second, the search schemes $\mathcal{S}_{Kuch}$ were the first to exploit the lower bound L . By carefully introducing a minimum number of errors on certain parts of each search, overlap between searches is minimized, reducing the search space and reducing the number of redundantly reported occurrences. Nevertheless, the searches in $\mathcal{S}_{Kuch}$ collectively still cover all error configurations and therefore maintain their lossless character. Third, the authors of [15] observed that the search pattern P does not necessarily need to be divided into equal-length parts. In fact, the part lengths can be chosen arbitrarily, provided that the same part lengths are used across all searches. Since search S_1 is associated with the critical U -string, it will likely correspond to the largest search space (and thus the longest runtime) if P is partitioned into equally sized parts. Therefore, it may be beneficial to slightly increase the size of part P_1 while slightly decreasing the sizes of the other parts. This adjustment reduces the search space for searches that match P_1 as their first, error-free part (i.e., searches S_1 , S_4 , and S_8), while increasing the search space for the remaining searches. Although some searches will become slower while others speed up, the overall effect is a net performance gain. A more formal justification of the benefit of uneven partitioning is given in [15], Section 3.2. Overall, the search schemes proposed in [15] proved to be very efficient for practical bioinformatics applications [34].

¹ Note that [15] has a different definition of lower bound, therefore those were adjusted to the definition we use in this paper. Furthermore, lower bounds in S_6 and S_7 were slightly corrected compared to [15].

3.1 Design of search schemes

Designing efficient (or even optimal) search schemes is a fundamental and difficult combinatorial optimization problem, due to a large number of degrees of freedom in designing a search scheme: the number of searches $|\mathcal{S}|$, the number of parts p , the size of each part $|P_i|$ ($i = 1 \dots p$), and the strings π_s , U_s and L_s for each search S_s ($s = 1 \dots |\mathcal{S}|$).

Kianfar et al. [11] were the first to tackle this problem in a systematic way for the Hamming distance. They proposed a mixed integer linear program (MILP) that minimizes the expected size of the search tree, i.e., the expected number of strings O enumerated. Assuming a fixed number of $|\mathcal{S}|$ searches, a fixed number of parts p parts and fixed part lengths $|P_i|$ ($i = 1 \dots p$), and a maximum number of allowed errors k , this objective translates into

$$\text{Minimize } \sum_{s=1}^{|\mathcal{S}|} \sum_{i=1}^p \sum_{j=1}^{|P_{\pi_s[i]}|} \sum_{d=L_s[i-1]}^{U_s[i]} n_{l,d}, \quad \text{where } l = j + \sum_{i'=1}^{i-1} |P_{\pi_s[i']}|. \quad (5)$$

Here, $n_{l,d}$ denotes the expected number of enumerated strings O of length l that have exactly d mismatches with respect to the corresponding part of search pattern P . The summations express that, during each search S_s , the number of errors d is constrained by $L_s[i-1] \leq d \leq U_s[i]$, and that parts of P are matched in the sequence specified by the permutation array π_s . Note that *during* the matching of part $P_{\pi_s[i]}$, the lower bound is $L_s[i-1]$, rather than $L_s[i]$, because the lower bound $L_s[i]$ applies only *after* part $P_{\pi_s[i]}$ has been fully matched. $L_s[0]$ is always 0. If the characters of the search text T and the search pattern P are assumed to be independently and uniformly drawn from the alphabet, $n_{l,d}$ can be efficiently computed for the Hamming distance (see [15, Section 3] for details). The MILP formulation imposes several constraints to ensure that the resulting search scheme $\mathcal{S}$ is valid. For example, the search scheme $\mathcal{S}$ should be lossless and the permutation array π should satisfy the connectivity property. Using the CPLEX solver on the MILP formulation, optimal triplets (π_s, L_s, U_s) are provided for each search S_s ($s = 1 \dots |\mathcal{S}|$) as output. Note that the value of $|\mathcal{S}|$, provided as input to the MILP solver, denotes an *upper bound* on the number of searches. If a resulting, MILP-optimal search scheme has $|\mathcal{S}^*| < |\mathcal{S}|$ searches, then $|\mathcal{S}| - |\mathcal{S}^*|$ *empty* searches are generated. The solver achieves this by setting $L_s[i] > U_s[i]$ for some i .

Using their MILP formulation, Kianfar et al. [11] were able to generate search schemes for up to $k = 4$ errors for the Hamming distance. For example, for $k = 4$ and $p = k + 1 = 5$ parts, the resulting search scheme $\mathcal{S}_{\text{Kian}}$ consists of three searches:

$$\begin{aligned} S_1 &= (12345, 00004, 03344); \\ S_2 &= (23451, 00000, 22334); \\ S_3 &= (54321, 00033, 00444); \end{aligned} \quad (6)$$

Note that search S_2 already allows two errors in the first part of P that is matched. This makes $\mathcal{S}_{\text{Kian}}$ less suitable for matching under the edit distance, as the dense region near the root of the search tree becomes very large.

More recently, Renders et al. [32] applied a MILP approach to generate search schemes, albeit with a different optimization function. The authors focus on the edit distance, where computing the expected number of enumerated strings O during a search is computationally more challenging. Additionally, they argue that the expected number of enumerated strings – computed under the assumption of uniformly random T and P – is a poor approximation of the actual number observed in real-world applications, such as read mapping against the

human reference genome. This discrepancy arises because the human reference genome has a complex repeat structure, in which certain patterns occur significantly more frequently than would be expected under a uniform randomness assumption.

Therefore, rather than trying to minimize the expected number of generated strings, the following objective function is used in their MIP formulation:

$$\text{Minimize } \sum_{s=1}^{|\mathcal{S}|} \sum_{i=1}^p (k+1)^{(p-i)} \cdot U_s[i] - \sum_{s=1}^{|\mathcal{S}|} \sum_{i=1}^p (p-i+1) \cdot L_s[i] + \sum_{q=1}^Q \sum_{s=1}^{|\mathcal{S}|} \lambda_{q,s}. \quad (7)$$

The first term of objective function (7) seeks to provide lexicographically small U -arrays. The weighting factors, $(k+1)^{(p-i)}$, are larger for smaller values of i , meaning that the number of allowed errors in U_s should increase only gradually as more parts of P are matched. The intuition is that allowing more errors should be deferred until the search tree becomes sparsely branched. The second term aims to maximize the L -arrays. In this case, the weighting factors are smaller, because the impact on runtime of the L -arrays is less pronounced compared to the U -arrays. Finally, the third component promotes minimizing the redundant reporting of occurrences by multiple searches. Here, $\lambda_{q,s}$ is a boolean variable indicating whether an error distribution q is covered by search s . Note that there are $Q = \binom{p+k}{k}$ possible ways to distribute at most k errors over p parts and that each error distribution should be covered by at least one search, i.e., $\sum_{s=1}^{|\mathcal{S}|} \lambda_{q,s} \geq 1$.

Assuming $p = k + 1$ parts and a search scheme $\mathcal{S}$ with exactly $|\mathcal{S}| = p$ searches, Renders et al. [33] provided MILP-optimal search schemes for up to $k = 7$ errors. These search schemes are referred to as *MinU* search schemes due to the primary focus on providing lexicographically small U -arrays. For reference, we provide three resulting optimal search schemes $\mathcal{S}_{\text{MinU,A}}$, $\mathcal{S}_{\text{MinU,B}}$, and $\mathcal{S}_{\text{MinU,C}}$ for $k = 4$ errors and $p = k + 1 = 5$ parts:

	$\mathcal{S}_{\text{MinU,A}}$	$\mathcal{S}_{\text{MinU,B}}$	$\mathcal{S}_{\text{MinU,C}}$	
S_1	(12345, 00222, 02244)	(12345, 01114, 01444)	(12345, 01114, 01444)	(8)
S_2	(23145, 00000, 01244)	(21345, 00003, 01444)	(21345, 00003, 01444)	
S_3	(32145, 01111, 01244)	(34521, 01111, 02244)	(34521, 01111, 01244)	
S_4	(45321, 00003, 01444)	(43521, 00000, 01244)	(43521, 00000, 01244)	
S_5	(54321, 01114, 01444)	(54321, 00222, 01244)	(54321, 00222, 02244)	

Next to these MILP-optimal search schemes for up to $k = 7$ errors, Renders et al. [32] provide search schemes for up to $k = 13$ errors that were generated using a greedy heuristic.

Note that despite this progress, the design of efficient search schemes remains an open research problem. First, due to the large number of variables involved, generating MILP-optimal search schemes becomes computationally challenging for larger values of k ; even for $k = 7$, the solving the optimization problem (7) requires many hours of computing time. Second, it is difficult to formulate the goal of minimizing the search space as an optimization criterion that is both accurate and easy to evaluate.

3.2 Dynamic selection of search schemes

For a fixed long text, such as a sequence of human genome, further optimizations of the search process are possible. The expected size of the search tree, and consequently the expected workload, associated with the optimal search schemes $\mathcal{S}_{\text{MinU,A}}$, $\mathcal{S}_{\text{MinU,B}}$, and $\mathcal{S}_{\text{MinU,C}}$ is identical. In other words, across a large number of (random) patterns P , and assuming equally sized parts P_i , the three search schemes should require the same amount of work. This can be verified by comparing their L - and U -arrays.

However, the workload for *individual* patterns P may vary significantly between the optimal search schemes. Recall that the critical U -string of a search scheme is the lexicographically largest U -string. Kucherov et al. [15] showed that in a search scheme with k errors and $p = k + 1$ parts, the minimal critical U -string is given by $02244 \dots kk$ when k is even, and $013355 \dots kk$ when k is odd [15, Theorem 1]. In $\mathcal{S}_{\text{MinU,A}}$, $\mathcal{S}_{\text{MinU,B}}$, and $\mathcal{S}_{\text{MinU,C}}$, searches S_1 , S_3 , and S_5 , respectively, have the critical U -string 02244 (indicated in boldface in (8)). As these searches already allow for two errors in the second part that is matched ($P_{\pi[2]}$), they are expected to require more work than the other searches in their respective search schemes. In other words, to minimize the workload associated with a search scheme, one should primarily focus on the search that contains the critical U -string.

These insights lead to the following heuristic for reducing workload. First, perform exact matching for parts P_i (for $i = 1, 3, 5$) individually. Next, compare their number of exact matches. If P_1 has the fewest exact matches, execute search scheme $\mathcal{S}_{\text{MinU,A}}$. Similarly, if P_3 has the fewest matches, execute search scheme $\mathcal{S}_{\text{MinU,B}}$. Otherwise, execute search scheme $\mathcal{S}_{\text{MinU,C}}$. The number of exact matches in P_i (for $i = 1, 3, 5$) is thus used as a proxy for the size of the search tree that must be explored during the search that contains the critical U -string. While this heuristic offers no strong guarantees regarding the workload of individual patterns, it performs very well *on average* across a large number of patterns. Additionally, the results from the exact matching of P_i (for $i = 1, 3, 5$) in the first step of the heuristic can be reused during the execution of the individual searches. Consequently, the heuristic incurs minimal overhead.

3.3 Dynamic partitioning of search patterns

Recall that search patterns P can be partitioned arbitrarily, as long as the same partitioning is consistently applied across all searches within a given search scheme. This flexibility allows for partitioning P in a way that balances the workload among searches.

In their original work, Kucherov et al. [15] proposed a dynamic programming algorithm to compute an optimal pattern partition for a given search scheme, under the assumption of a random pattern searched in a random text. They also provided some examples of uneven partitions outperforming even ones. They point out, however, that an optimal partition is very sensible to the choice of parameters such as the search scheme, number of parts but also the text length.

In practice, we are often interested to design an efficient search for a fixed text, such as a human genome. If all searches within a search scheme have the same expected workload—such as in the pigeonhole-based search scheme $\mathcal{S}_{\text{ph}}$ —it is advantageous to partition P so that each part P_i has approximately the same number of exact occurrences in the text. However, for a specific pattern P , an equal-sized partitioning does not necessarily achieve this balance, since certain parts P_i may occur more frequently within the search text T than others. In other words, an effective partitioning strategy should be tailored to the content of P itself.

To address this, Renders et al. [34] proposed a greedy heuristic that determines the part sizes dynamically to ensure a suitably balanced distribution of exact occurrences across the parts P_i . What constitutes a “suitably balanced” distribution depends on the search scheme. For symmetric search schemes like $\mathcal{S}_{\text{ph}}$, this means ensuring a uniform distribution of the number of occurrences of P_i . In general, balance is achieved when the number of occurrences of each P_i is inversely proportional to the expected workload of the searches that begin at P_i . The heuristic by Renders et al. operates by determining part sizes while performing exact matching for each P_i , introducing minimal overhead. However, it does not provide guarantees regarding the workload of individual patterns in the context of dynamic

Table 1 The average size of the search tree (lower = better) for different search schemes when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba, allowing for at most k errors in Hamming distance. Each value represents the average size for a single read and its reverse complement. A dash (‘-’) indicates no search scheme is available for that value of k . dnc = did not complete.

Search scheme	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
Naive backtracking	3 946	48 630	430 831	2 950 884	dnc	dnc	dnc
Pigeonhole [17]	439	1 151	3 605	10 851	29 956	74 175	165 197
Suffix filter [10]	439	954	2 275	5 153	10 902	21 589	40 707
01 [31]	516	1 115	2 749	6 510	14 363	30 098	61 651
Kucherov $k + 1$ [15]	439	902	1 715	4 260	-	-	-
Kucherov $k + 2$ [15]	516	1 156	2 131	6 020	-	-	-
Kianfar [11]	439	899	6 355	52 492	-	-	-
Man _{best} [31]	-	-	-	5 549	-	-	-
MinU [33]	439	902	1 715	3 506	6 595	10 394	16 371

search scheme selection. Nevertheless, it has been shown to yield performance improvements of approximately 30% on average. Notably, dynamic partitioning of search patterns and dynamic selection of search schemes are independent techniques and can be combined for further optimization.

4 Results

4.1 Dataset

In all benchmarks, all occurrences of search patterns were identified in both strands of the human reference genome (GRCh38) with up to k errors under the Hamming or edit distance metrics. Non-ACGT characters (e.g., ‘N’s) in the reference sequence were replaced with randomly chosen nucleotides. We sampled 1,000,000 Illumina NovaSeq 6000 reads (151 bp) from a whole-genome sequencing dataset (accession no. SRR13586123). Given our focus on approximate pattern matching, all reads were treated as single-end, ignoring paired-end read information. All results were obtained using Columba version 2.0, an efficient tool for approximate pattern matching using search schemes. The C++ source code of Columba is available at <https://github.com/biointec/columba> under AGPL-3.0 license.

4.2 Empirical evaluation of search schemes

We exemplify the practical benefit of using different search schemes using the mentioned data set. More extensive results can also be found in [9]. Tables 1 and 2 show the average size of the search tree to identify all approximate occurrences of an Illumina read in the human reference genome for different search schemes and different values of allowed errors k under the Hamming² and edit distance³, respectively. The search tree corresponds to the total number of (partial) candidate occurrence strings O that were generated during the execution of the search scheme. It evaluates the efficiency of a search scheme regardless of implementation characteristics of the software. Note that Tables 1 and 2 report the number of generated candidate occurrences O that actually exist in T .

² Command for Hamming distance:

```
./columba -a all -e [e] -K 0 -p uniform -i 0 -nD -m hamming -v -c [path/to/search/scheme]
```

³ Command for edit distance:

```
./columba -a all -e [e] -K 0 -p uniform -i 0 -nD -v -c [path/to/search/scheme]
```

■ **Table 2** The average size of the search tree (lower = better) for different search schemes when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba, allowing for at most k errors in edit distance. Each value represents the average size for a single read and its reverse complement. A dash (‘-’) indicates no search scheme is available for that value of k . dnc = did not complete.

Search scheme	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
Naive backtracking	7 248	160 736	2 419 153	24 728 576	dnc	dnc	dnc
Pigeonhole [17]	444	1 205	3 936	12 444	36 386	95 614	225 615
Suffix filter [10]	444	990	2 434	5 698	12 515	25 846	50 837
01 [31]	523	1 161	2 946	7 180	16 426	35 978	77 455
Kucherov $k + 1$ [15]	444	937	1 821	4 631	-	-	-
Kucherov $k + 2$ [15]	523	1 100	2 157	5 394	-	-	-
Kianfar [11]	444	934	10 344	166 809	-	-	-
Man _{best} [31]	-	-	-	6 233	-	-	-
MinU [33]	444	937	1 821	3 821	7 336	11 848	19 021

■ **Table 3** The runtime (in seconds, lower = better) for different search schemes when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba, allowing for at most k errors in Hamming distance. A dash (‘-’) indicates no search scheme is available for that value of k . The runtimes for Naive Backtracking are extrapolated from a dataset with only 10 000 reads. dnc = did not complete in 3 hours.

Search scheme	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
Naive Backtracking	328	3 613	32 354	235 843	dnc	dnc	dnc
Pigeonhole	102	223	585	1 681	4 611	dnc	dnc
Suffix filter	103	193	380	771	1575	3 051	5 581
01	105	208	443	947	2051	4 340	dnc
Kucherov $k + 1$	101	187	315	638	-	-	-
Kucherov $k + 2$	108	209	344	723	-	-	-
Kianfar	99	187	763	4 889	-	-	-
Man _{best}	-	-	-	914	-	-	-
MinU	98	190	311	577	1 000	1 573	2 334

Tables 1 and 2 include the search space for naive backtracking as a baseline scenario. Results for backtracking with $k > 4$ are omitted due to computational impracticality. The search scheme based on the pigeonhole principle already outperforms backtracking by a large margin. This emphasizes the necessity to avoid having to explore the densely branched region near the root of the substring trie. Exploiting the 01 principle further reduces this search space. However, custom-designed search schemes with stringent lower and upper bounds generally perform best, with a more significant difference for larger values of k . For example, for $k = 7$, the MinU search schemes have a $10\times$ and $4\times$ smaller search space than the pigeonhole principle-based and 01-based search schemes, respectively. Note that Tables 1 and 2 include Man_{best}, a search scheme designed by Pockrandt [31] for $k = 4$.

Tables 3 and 4 report the runtime to align 1 million Illumina reads to the human reference genome on a single core 64-core Intel® Xeon® E5-2698 v3 CPU, running at a base clock frequency of 2.30 GHz CPU. These values correlate to a large degree with Tables 1 and 2.

4.3 Effect of dynamic selection of search schemes

To ensure an unbiased comparison between search schemes, certain optimizations in Columba, such as dynamic selection of search schemes and dynamic partitioning of search patterns, were disabled in the previous comparison, as they may affect different search schemes in different ways. For example, exploiting dynamic selection of search schemes requires having equivalent alternatives to choose from.

■ **Table 4** The runtime (in seconds, lower = better) for different search schemes when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba, allowing for at most k errors in edit distance. A dash ('-') indicates no search scheme is available for that value of k . The runtimes for Naive Backtracking are extrapolated from a dataset with only 10 000 reads. dnc = did not complete in 3 hours.

Search scheme	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
Naive Backtracking	684	13 125	200 593	dnc	dnc	dnc	dnc
Pigeonhole	105	256	725	2 207	6 657	dnc	dnc
Suffix filter	102	231	479	1 044	2 251	4 741	9 687
01	112	243	550	1 271	2 920	6 648	dnc
Kucherov $k + 1$	105	219	405	877	-	-	-
Kucherov $k + 2$	110	239	442	975	-	-	-
Kianfar	113	215	1243	-	-	-	-
Man _{best}	-	-	-	1 210	-	-	-
MinU	104	217	392	762	1 398	2 281	3 618

■ **Table 5** The average size of the search space for different dynamic techniques when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba with the minU search schemes, allowing for at most k errors in edit distance. A dash ('-') indicates that dynamic selection is not useful for this value of k .

Heuristic	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
None	444	937	1 821	3 821	7 336	11 848	19 021
Dynamic Partitioning	340	742	1 589	3 791	7 174	11 679	18 333
Dynamic Selection	-	845	-	3 721	-	11 379	-
Both	-	708	-	3 458	-	10 779	-

Table 5 presents the average size of the search space for the same alignment task using the MinU search schemes and the edit distance metric under four scenarios: i) no heuristics applied; ii) using only dynamic partitioning of search patterns; iii) using only dynamic selection of search schemes (applicable only to even k); iv) both heuristics enabled. Note that the results in scenario i) are identical to those in Table 2 for the MinU search scheme.

The use of dynamic partitioning of search patterns adaptively determines the size of the search patterns so that their number of exact occurrences is balanced. Across all values of k , and for a large number of search patterns, this technique reduces the search space by 4% to 23% for this dataset. The largest relative reduction occurs for smaller values of k , where fewer but larger parts provide greater flexibility in adapting their sizes.

Enabling dynamic selection of search schemes allows Columba to choose between optimal search schemes based on the number of exact matches in their individual parts. This technique applies only to even values of k , where the critical U-string is 02244... kk . Again, this technique reduces the search space by a similar margin.

In case both techniques are combined, dynamic partitioning of the search pattern is applied first. Once a partitioning for the particular read has been fixed, the expected best-performing search scheme is selected. Combining both heuristics yields a reduction in search space between 9% ($k = 6$) and 24% ($k = 2$). Both heuristics are easy to implement and impose almost no overhead. These results demonstrate their effectiveness for practical applications.

References

- 1 Djamal Belazzougui and Fabio Cunial. Fully-functional bidirectional Burrows-Wheeler indexes and infinite-order de bruijn graphs. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual*

- Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPICs*, pages 10:1–10:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.CPM.2019.10.
- 2 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8125 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2013. doi:10.1007/978-3-642-40450-4_12.
 - 3 Karel Brinda, Leandro Lima, Simone Pignotti, Natalia Quinones-Olvera, Kamil Salikhov, Rayan Chikhi, Gregory Kucherov, Zamin Iqbal, and Michael Baym. Efficient and robust search of microbial genomes via phylogenetic compression. *bioRxiv:2023.04.15.536996*, 2024. to appear in *Nature Methods*. doi:10.1101/2023.04.15.536996.
 - 4 Stefan Burkhardt and Juha Kärkkäinen. Better filtering with gapped q-grams. In Amihood Amir and Gad M. Landau, editors, *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, volume 2089 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2001. doi:10.1007/3-540-48194-X_6.
 - 5 M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report 124, Digital Equipment Corporation, 1994.
 - 6 Martin Farach-Colton, Gad M. Landau, S. Cenk Sahinalp, and Dekel Tsur. Optimal spaced seeds for faster approximate string matching. *Journal of Computer and System Sciences*, 73(7):1035–1044, 2007. Bioinformatics III. doi:10.1016/j.jcss.2007.03.007.
 - 7 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st Symp. on Foundations of Computer Science (FOCS), Redondo Beach, California, USA*, pages 390–398. IEEE Computer Society, February 2000. doi:10.1109/SFCS.2000.892127.
 - 8 Ira M Gessel and Christophe Reutenauer. Counting permutations with given cycle structure and descent set. *Journal of Combinatorial Theory, Series A*, 64(2):189–215, 1993. doi:10.1016/0097-3165(93)90095-P.
 - 9 Simon Gene Gottlieb and Knut Reinert. Search schemes for approximate string matching. *NAR Genomics and Bioinformatics*, 7(1):lqaf025, March 2025. doi:10.1093/nargab/lqaf025.
 - 10 Juha Kärkkäinen and Joong Chae Na. Faster filters for approximate string matching. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007. doi:10.1137/1.9781611972870.8.
 - 11 Kiavash Kianfar, Christopher Pockrandt, Bahman Torkamandi, Haochen Luo, and Reinert Knut. Optimum search schemes for approximate string matching using bidirectional FM-index. In *RECOMB-Seq*, pages 1–13, March 2018. available at arxiv:1711.02035.
 - 12 Daehwan Kim, Li Song, Florian P Breitwieser, and Steven L Salzberg. Centrifuge: rapid and sensitive classification of metagenomic sequences. *Genome research*, 26(12):1721–1729, 2016.
 - 13 Gregory Kucherov. Evolution of biosequence search algorithms: a brief survey. *Bioinformatics*, 35(19):3547–3552, October 2019. doi:10.1093/bioinformatics/btz272.
 - 14 Gregory Kucherov, Laurent Noé, and Mikhail A. Roytberg. Multiseed lossless filtration. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 2(1):51–61, 2005. doi:10.1109/TCBB.2005.12.
 - 15 Gregory Kucherov, Kamil Salikhov, and Dekel Tsur. Approximate string matching using a bidirectional index. *Theoretical Computer Science*, 638:145–158, 2016. preliminary version in CPM’2014. doi:10.1016/J.TCS.2015.10.043.
 - 16 Gregory Kucherov and Dekel Tsur. Improved filters for the approximate suffix-prefix overlap problem. In Edleno Silva de Moura and Maxime Crochemore, editors, *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, volume 8799 of *Lecture Notes in Computer Science*, pages 139–148. Springer, 2014. doi:10.1007/978-3-319-11918-2_14.

- 17 Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon C. K. Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *2009 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2009, Washington, DC, USA, November 1-4, 2009, Proceedings*, pages 31–36. IEEE Computer Society, 2009. doi:10.1109/BIBM.2009.42.
- 18 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012.
- 19 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(3):R25, 2009.
- 20 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009. doi:10.1093/bioinformatics/btp324.
- 21 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. doi:10.1093/BIOINFORMATICS/BTP698.
- 22 Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, June 2009. doi:10.1093/bioinformatics/btp336.
- 23 Huan Liu, Quan Zou, and Yun Xu. A novel fast multiple nucleotide sequence alignment method based on FM-index. *Briefings in Bioinformatics*, 23(1):bbab519, 2022. doi:10.1093/BIB/BBAB519.
- 24 Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, March 2002. doi:10.1093/bioinformatics/18.3.440.
- 25 Veli Mäkinen, Niko Välimäki, Antti Laaksonen, and Riku Katainen. Unified view of backward backtracking in short read mapping. In Tapio Elomaa, Heikki Mannila, and Pekka Orponen, editors, *Algorithms and Applications: Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday*, volume 6060 of *Lecture Notes in Computer Science*, pages 182–195. Springer, 2010. doi:10.1007/978-3-642-12476-1_13.
- 26 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 27 Giovanni Manzini. Invited lecture: The Burrows–Wheeler transform: Theory and practice. In Mirosław Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS’99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*, volume 1672 of *Lecture Notes in Computer Science*, pages 34–47. Springer, 1999. doi:10.1007/3-540-48340-3_4.
- 28 Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.
- 29 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001. doi:10.1145/375360.375365.
- 30 Christopher Pockrandt, Marcel Ehrhardt, and Knut Reinert. EPR-dictionaries: A practical and fast data structure for constant time searches in unidirectional and bidirectional FM indices. In S. Cenik Sahinalp, editor, *Research in Computational Molecular Biology*, pages 190–206, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-56970-3_12.
- 31 Christopher Maximilian Pockrandt. *Approximate string matching: improving data structures and algorithms*. PhD thesis, Freie Universität Berlin, 2019.
- 32 Luca Renders, Lore Depuydt, Sven Rahmann, and Jan Fostier. Automated design of efficient search schemes for lossless approximate pattern matching. In Jian Ma, editor, *Research in Computational Molecular Biology - 28th Annual International Conference, RECOMB 2024, Cambridge, MA, USA, April 29 - May 2, 2024, Proceedings*, volume 14758 of *Lecture Notes in Computer Science*, pages 164–184. Springer, 2024. doi:10.1007/978-1-0716-3989-4_11.
- 33 Luca Renders, Lore Depuydt, Sven Rahmann, and Jan Fostier. Lossless approximate pattern matching: Automated design of efficient search schemes. *Journal of Computational Biology*, 31(10):975–989, 2024. doi:10.1089/cmb.2024.0664.

- 34 Luca Renders, Kathleen Marchal, and Jan Fostier. Dynamic partitioning of search patterns for approximate pattern matching using search schemes. *iScience*, 24(7):102687, 2021. doi:10.1016/j.isci.2021.102687.
- 35 Luís M. S. Russo, Gonzalo Navarro, Arlindo L. Oliveira, and Pedro Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009. doi:10.3390/A2031105.
- 36 Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012. doi:10.1016/J.IC.2011.03.007.
- 37 Enrico Siragusa, David Weese, and Knut Reinert. Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic acids research*, 41(7):e78–e78, 2013.
- 38 Li Song and Ben Langmead. Centrifuger: lossless compression of microbial genomes for efficient and accurate metagenomic sequence classification. *Genome Biology*, 25(1):106, 2024.
- 39 Wing-Kin Sung. Indexed approximate string matching. In *Encyclopedia of Algorithms*, pages 964–968. Springer, 2016. doi:10.1007/978-1-4939-2864-4_188.
- 40 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.
- 41 Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. *Inf. Comput.*, 213:49–58, 2012. doi:10.1016/J.IC.2012.02.002.
- 42 Christophe Vroland, Mikael Salson, Sébastien Bini, and Hélène Touzet. Approximate search of short patterns with high error rates using the 01*0 lossless seeds. *Journal of Discrete Algorithms*, 37:3–16, 2016. doi:10.1016/j.jda.2016.03.002.
- 43 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11. IEEE, 1973. doi:10.1109/SWAT.1973.13.
- 44 Pinglu Zhang, Huan Liu, Yanming Wei, Yixiao Zhai, Qinzong Tian, and Quan Zou. FM-Align2: a novel fast multiple nucleotide sequence alignment method for ultralong datasets. *Bioinformatics*, 40(1):btac014, 2024. doi:10.1093/BIOINFORMATICS/BTAE014.

Phasing Data from Genotype Queries via the μ -PBWT

Davide Cozzi  

Department of Computer Science, University of Milano-Bicocca, Italy

Paola Bonizzoni  

Department of Computer Science, University of Milano-Bicocca, Italy

Christina Boucher  

Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA

Ben Langmead  

Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

Yuri Pirola  

Department of Computer Science, University of Milano-Bicocca, Italy

Abstract

Genotype phasing – the process of reconstructing haplotypes from genotype data – is a fundamental problem in genomics with applications in ancestry inference, imputation, and disease association. Traditional phasing methods rely on statistical models or combinatorial approaches which can be computationally expensive, particularly when applied to large-scale reference panels.

In this paper, we present a first exploration of using the μ -PBWT (a run-length encoded Positional Burrows–Wheeler Transform) to solve the genotype phasing problem with a reference panel. Leveraging our previous results on positional substrings, we propose an approach that can explain a query genotype if the corresponding haplotype pair exists in the input panel. Moreover, our method is extended to cases where such a pair does not exist, even though some regions should remain unphased if they cannot be explicitly explained using the reference panel.

We implemented this method and compared it against Beagle, a state-of-the-art phasing tool, demonstrating that, in the absence of mutations and recombinations, our approach correctly identifies the haplotype pair that explains a genotype query while using seven times less memory than Beagle. However, we also observe that as mutation rates increase, the quality of the phasing decreases as a result of the growing difficulty of identifying consistent haplotype pairs in the presence of sequence variation.

These findings highlight the potential of μ -PBWT as an efficient alternative for genotype phasing, particularly in settings where computational resources are limited. The source code is publicly available at <https://github.com/dlclgold/muPBWT/tree/phase>.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Data structures design and analysis

Keywords and phrases Positional Burrows–Wheeler Transform, r-index, minimal position substring cover, set-maximal exact matches, genotype phasing

Digital Object Identifier 10.4230/OASICS.Manzini.2025.10

Supplementary Material

Software (Source code): <https://github.com/dlclgold/muPBWT/tree/phase>
archived at `swh:1:dir:8131138a04fdd3a78c074d3b5707c43694bc7914`

Funding P.B., Y.P., and D.C. have received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement PANGAIA No. 872539. P.B. is also supported by the grant MIUR 2022YRB97K, PINC, Pangenome Informatics: from Theory to Applications, funded by the EU, Next-Generation EU, Mission 4 and ITN ALPACA N.956229. C.B. was supported by NIH/NIAID (Grant No. R01AI14180) and NSF:SCH:INT (Grant No. 2013998). C.B. and B.L. were supported by NIH:NHGRI: (Grant No. R56HG013865).



© Davide Cozzi, Paola Bonizzoni, Christina Boucher, Ben Langmead, and Yuri Pirola; licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini’s 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 10; pp. 10:1–10:17

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Within the field of haplotype analysis, the Positional Burrows-Wheeler Transform (PBWT) [9] has emerged as a fundamental data structure for efficiently encoding and searching large-scale haplotype data. Its ability to rapidly detect long shared substrings between a query sequence and a reference panel makes it a crucial tool for various genomic applications, including haplotype imputation [19, 18], ancestral inference [13, 25], and genotype phasing [15]. One of the key advantages of PBWT is its efficient representation of haplotype panels, allowing fast and memory-efficient computations, particularly compared to traditional sequence alignment-based approaches. Over the years, PBWT has been extended and refined to support more complex operations [24, 26, 17, 21], making it an essential component of modern haplotype analysis pipelines.

Sanaullah et al. [22], who formulated the haplotype threading problem, the task of representing a query haplotype using a minimal set of substrings derived from a reference panel, introduced a significant development in PBWT-based haplotype analysis. To address this problem, they introduced the Minimal Positional Substring Cover (MPSC) problem, which seeks to identify the smallest collection of substrings that fully cover a query while maintaining positional consistency with the reference haplotypes. Their work proposed an efficient PBWT-based solution by modeling the solution space as a graph, enabling computationally efficient haplotype threading. However, their approach focused primarily on haplotypes rather than genotypes, leaving an open question as to how these methods could be extended to genotype phasing, which requires handling heterozygous loci and potential recombination events.

Building on this work, in 2024, Bonizzoni et al. [3] demonstrated that the MPSC problem could be solved in sublinear space, significantly improving its scalability for large reference panels. That approach was based on efficiently computing k -Set Maximal Exact Matches (SMEMs) via k -Matching Statistics (k -MS), a technique that enabled the identification of minimal positional substring covers with reduced memory requirements. This advancement allowed for an efficient solution to the MPSC problem and its variations proposed by Sanaullah et al. The ability to solve these problems in sublinear space is particularly valuable for large-scale genomic datasets, where traditional PBWT-based methods may become memory-intensive.

In this paper, we extend the results of Bonizzoni et al. [3] to the problem of genotype phasing, adapting the MPSC framework to work with genotype queries instead of haplotypes. The main idea that inspires this work is the observation that large reference panels likely include a haplotype pair that explains a genotype query. However, we need methods that can scale to such large panels. Here, we introduce a novel approach using μ -PBWT [8], a refined version of run-length encoded PBWT (RLPBWT) [2], to efficiently phase genotypes while optimizing for minimal recombination events. Our method is designed to reconstruct haplotype pairs that explain a given genotype by leveraging positional substrings (PSs) and complementary positional substrings (CPSs) extracted from a reference panel. Unlike previously proposed combinatorial methods that assume a perfect haplotype match [14, 4, 1], our approach can handle cases where recombinations are required to fully explain the genotype, making it more adaptable to real-world data. In addition, current state-of-the-art tools such as Beagle [5, 6] are based on the Li and Stephens Hidden Markov Model [16], while our approach is deterministic in the absence of mutation and recombination.

Our study focuses on two primary scenarios: (1) an ideal case where the reference panel contains two haplotypes that fully explain the genotype without requiring recombination, and (2) a more realistic setting where recombinations across haplotype segments of the panel

are needed due to the absence of a single haplotype pair that explains the genotype query. To address these challenges, we introduce a combinatorial approach that processes 2-positional substrings (2PSs) and identifies valid haplotype pairs that minimize recombination events. Furthermore, we implement and evaluate our method against Beagle [6, 5], a state-of-the-art statistical phasing tool. Our experimental results demonstrate that μ -PBWT achieves competitive accuracy while significantly reducing computational costs in scenarios with low mutation rates. On the other hand, introducing target mutations (and consequently recombinations not having the target parents in the reference panel) reduces the performance of our approach, in terms of switch error rate/mismatch rate. Our method requires little memory and can scale on very large reference panels, thanks to the μ -PBWT index. This makes μ -PBWT a promising alternative for applications where memory efficiency and processing speed are critical.

2 Related work

Due to the diploid nature of humans, chromosomes come in two copies, called haplotypes, each copy inherited from one of the two parents (see [4] for a survey of the main concepts). Sequencing an individual is a process that does not distinguish between the two haplotypes, as reads are fragments uniformly distributed over both chromosomal copies. Distinguishing the haplotypes from sequencing reads is biologically expensive, and thus computational methods are employed for this purpose.

A genotype is simply the conflated information of a pair of haplotypes and can be easily determined by sequencing. More precisely, a genotype is specified by a vector of allele pairs at each locus, where the pairs consist of homologous or heterozygous alleles. In a genotype, information on which heterozygous allele belongs to which haplotype is not available and must be determined.

Observe that each haplotype inherited from a parent is itself the result of a mosaic of the two parental haplotypes, due to *recombination* events, which are a typical evolutionary phenomenon occurring in human populations. Recombination events regulate Mendelian inheritance laws within a family trio and have been extensively studied in the context of combinatorial methods for the phasing or haplotyping of input genotype matrices [4].

From a combinatorial perspective, a haplotype at a set of biallelic loci, the genomic positions in which only two distinct alleles are observed in a population, is formalized as a binary string. In this representation, the i -th position of the string is assigned a 0 if the individual carries the reference allele at that locus (i.e. the allele matches the reference in that locus), and a 1 if the individual carries the alternative allele.

The *genotype* of an individual is the combined representation of the two haplotypes inherited from the two parents. Formally, a genotype is represented as a string on the alphabet $\{0, 1, 2\}$, where the i -th position is 0 if the individual has inherited the reference allele from both parents at the i -th locus, it is 1 if the individual has inherited the alternative allele from both parents at the i -th locus, or it is 2 if the individual, at the i -th locus, has inherited the reference allele from a parent and the alternative allele from the other parent. The positions where the genotype is 0 or 1 are called *homozygous*, while the others are called *heterozygous*. We say that a pair $\langle h_1, h_2 \rangle$ of w -length haplotypes explains a w -length genotype Q if and only if, for all $i \in [1, w]$ we have that $h_1[i] = h_2[i] = Q[i]$ if $Q[i] \neq 2$, or $h_1[i] \neq h_2[i]$ if $Q[i] = 2$. In simpler terms, a pair of haplotypes explains a genotype if they match the genotype at homozygous positions and differ at heterozygous positions.

Then, the *genotype phasing* problem aims at reconstructing an individual's haplotypes from their genotype. More specifically, determining the exact pair of haplotypes that correspond to the observed genotype. The problem is straightforward for homozygous positions, where both haplotypes match the genotype; however, at heterozygous positions, the phase is ambiguous because multiple pairs of haplotypes, differing only in the assignment of alleles to each parent, can equally explain the genotype. There is a vast literature on combinatorial methods for genotype phasing, mainly distinguished by how the problem is modeled and by the type of input data considered. For example, Gusfield proposed solving genotype phasing using the perfect phylogeny model [14, 1] when the input matrix consists of a collection of genotype vectors.

The availability of large panels of population haplotype data has shifted the focus of genotype phasing toward the use of statistical methods which, based on information from an input panel, can effectively explain genotype queries. An example of this is Beagle [5], which uses phased input data, that is, a known haplotype panel, to build a hidden Markov model specifically tailored to diploid data.

In this paper, genotype phasing is addressed through the so-called haplotype threading, which models the process of reconstructing a haplotype as a mosaic of haplotypes from a reference panel.

More precisely, haplotype threading aims to cover a haplotype query using segments from haplotypes in an input panel. The main idea is that a haplotype results from the accumulation of recombination events, combining chromosomal segments from various haplotypes in the panel. The Li and Stephens (LS) hidden Markov model (HMM) [16] produces a haplotype threading for a query haplotype using a panel of known haplotypes, requiring time that is linear in the size of the input panel. However, this running time becomes infeasible when the haplotype panel is large, and thus sublinear-time solutions have recently been investigated, mainly using the PBWT (Positional Burrows-Wheeler Transform) framework.

Moreover, modern haplotype panels typically exhibit much lower mismatch rates and switch error rates, two key measures commonly used in the probabilistic modeling of the LS framework. Observe that a zero mismatch rate between the query haplotype and the input panel corresponds to the situation where the query can be entirely covered by segments from the panel, with the number of such segments directly corresponding to the switch error. In this case, combinatorial approaches become particularly attractive, especially when they offer faster solutions than LS-based methods.

In particular, the solution proposed by Sanaullah et al. [20], the Minimum Positional Substring Cover (MPSC) with non-overlapping segments, corresponds exactly to haplotype threading under a zero mismatch error assumption. When overlapping segments are permitted, identifying the breakpoint between two matching positional substrings corresponds to determining the transition point between two distinct haplotypes in the panel.

Now, the input panel is not guaranteed to include the two haplotypes from which a query genotype may have been generated, nor do we know whether the two haplotypes of a parent are present in the panel when explaining a haplotype as the result of recombinations from that pair. This is why the mosaic of segments may involve multiple haplotypes of the panel.

The introduction of the PBWT has significantly improved the time efficiency of computational approaches to haplotype threading. The first major improvement concerns the time required to find coverage of a haplotype, which has been reduced from $O(N \cdot M)$ to $O(N)$, where N is the number of sites in a haplotype sequence [20]. This reduction in time complexity allows the methods to scale to a large number M of haplotype sequences, which was not feasible with the original LS-based method. Finally, since the memory required

by these approaches can be extremely prohibitive, a further improvement was proposed in [3], where a compressed data structure is introduced to store and manage the input panel using the μ -PBWT [8]. In particular, space efficiency is crucial for handling large cohorts of haplotype data, helping to reduce mismatch errors and switch errors. In this paper, we will assume an ideal future framework where the large amount of data allows for the assumption that the two haplotypes that explain a genotype are part of the input data.

3 Background

3.1 The Minimal Positional Substring Cover Problems

Throughout this paper, we define a string X on a finite, ordered alphabet $\Sigma = \{c_1, \dots, c_\sigma\}$ to be the concatenation of $|X| = w$ characters $X = X[1..w]$ from Σ . We denote the empty string as ε , the string that spans i through j as $X[i..j]$ (with $X[i..j] = \varepsilon$ if $i > j$), the i -th prefix of X as $X[1..i]$, and the i -th suffix as $X[i..|X|]$.

A *positional substring* of a string X , in short PS, is a triplet (i, j, X) with $1 \leq i, j \leq |X|$ and we say that the substring corresponding to (i, j, X) is $X[i..j]$. Two positional substrings (i, j, X) and (k, l, Y) are *equal* if and only if $i = k$, $j = l$, and $X[i..j] = Y[k..l]$. A positional substring (i, j, X) is *contained* in a string Y if and only if $X[i..j] = Y[i..j]$.

Given a set S of strings of length w (i.e., a panel), a positional substring (i, j, X) is a k -positional substring in S , in short k PS, if (i, j, X) is contained in at least k strings of S . A k -positional substring cover of a w -length string P by S is a set C of positional substrings of P such that: (i) each position $l \in [1, w]$ of P is covered by a $(i, j, X) \in C$ (i.e., $i \leq l \leq j$), (ii) each $(i, j, X) \in C$ is contained in P , and (iii) each $(i, j, X) \in C$ is contained in at least k distinct strings of S (i.e., they are k PS in S). The size of the cover is the number of elements in C , which we denote as $|C|$.

Given a set S of h strings of length w and a string P of length w , the k -Minimal Positional Substring Cover problem (k -MPSC) [20] seeks to compute, if it exists, a k -positional substring cover of P by S with the smallest size over all k -positional substring covers of P by S .

We note that the MPSC problem is the k -MPSC problem where k is equal to 1, i.e., each positional substring of the cover is contained in at least one string of the panel. It is easy to see that a solution to the problem exists if and only if for every i , with $1 \leq i \leq w$, the positional substrings (i, i, P) are contained in at least k distinct strings of S , i.e., they are k PS.

Given a panel S and a string P there can exist several distinct k -MPSC of the same size (hence, several solutions to the k -MPSC problem). Since the solution to k -MPSC can affect downstream applications, three variants have been proposed [20, 22] to constrain the solution.

Sanaullah et al. [22] defined the problem of computing Leftmost MPSC using 1-positional substring covers, while we generalize the method in [3] to compute Leftmost k -positional substring covers. In the problem definition, given a k -MPSC C , the i -th positional substring of C (for $1 \leq i \leq |C|$) refers to the i -th substring when listing all positional substrings of C in order of their starting positions.

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

For the purposes of this manuscript, we relax the problem of computing MPSC to compute a positional Substring Cover of P by S , which is not necessarily minimal, and we denote this solution as a PSC.

3.2 Solving the k -MPSC problem variants using PBWT

The best known algorithm for computing a k -MPSC requires $O(w)$ time [22], assuming that an index of the haplotype panel S is given as input. This algorithm processes the haplotype panel S column-wise from left to right, extending the matches of the query string P that are shared with at least k strings of the panel S . In each column, if the current match cannot be extended, then a new match is started from the current column. Optimality is ensured by the property of k -MPSC modularity [20, Lemma 2]. Matches of P with at least h strings in S are efficiently computed and extended using the Positional Burrows-Wheeler Transform (PBWT) of S . Hereon, we denote $h \cdot w$ as n , which will be used throughout this paper to bind the space and time complexity. The k -MPSC algorithm of Sanaullah et al. [20] requires $O(n)$ -space to ensure random access in constant time to the input panel and to the PBWT.

3.3 Positional Burrows–Wheeler Transform and μ -PBWT

In 2014, Durbin proposed the PBWT [9] as an efficient data structure to perform internal and external pattern matching on a set $S = \{S_1, \dots, S_h\}$ of h binary sequences of length w . The PBWT consists of two arrays per column j : (i) the *prefix array* PA_j and (ii) the *divergence array* DA_j .

Since the PBWT of column j is based on the co-lexicographic ordering of prefixes of S up to column $j - 1$, PA_j stores the permutation of the set of row indices, i.e., the set $\{1, \dots, h\}$. Formally, $\text{PA}_j[i] = k$ if and only if $S_k[1..j - 1]$ is the i -th element in this sorting in all row prefixes up to column $j - 1$. On the other hand, considering two consecutive prefixes in the reordering for column j , i.e., $S_{\text{PA}_j[i]}[1..j - 1]$ and $S_{\text{PA}_j[i-1]}[1..j - 1]$, $\text{DA}_j[i] = \ell$ iff ℓ is the length of the longest common suffix between those two prefixes. Note that, by definition, $\text{PA}_1 = \{1, \dots, h\}$ and $\text{DA}_1 = \{0, \dots, 0\}$.

Given the entire set of prefix arrays, we define the PBWT matrix $\text{PBWT}[1..h][1..w]$ as the matrix obtained by permuting each column of the input panel by the corresponding prefix array. Formally, denoting the j -th column of a matrix X by $\text{col}(X)_j$, each column of the PBWT matrix is defined as $\text{col}(\text{PBWT})_j[i] = \text{col}(M)_j[\text{PA}_j[i]]$ for all $i = 1..h$ and $j = 1..w$.

Additionally, to solve the pattern matching problem of a string against a panel, the PBWT is augmented with an FM-index [12] like data structure, used to follow a certain row in the permutations induced by the prefix arrays.

Regarding complexity, Durbin [9] proposed a set of algorithms that compute the entire set of prefix arrays and the entire set of divergence arrays in $O(hw)$ -time and $O(hw)$ -space.

Durbin highlighted the potential of a run-length compression of the PBWT in his original paper on this data structure. Later, Bonizzoni et al. [2] and Cozzi et al. [8] proposed various combinations of data structures to efficiently store and query a run-length encoded PBWT (RLPBWT).

In this paper, we consider the implementation of the RLPBWT of Cozzi et al. [8], referred to as μ -PBWT. For a detailed explanation of its definition and its application in addressing the problem of identifying set-maximal exact matches between a reference panel and an external query, we direct readers to the original paper. Next, we recall one of the key ideas behind the μ -PBWT algorithm: Matching Statistics.

► **Definition 1** (Matching Statistics in the PBWT). *Given a set S of h binary w -long sequences $S = \{S_1, \dots, S_h\}$ and a pattern $P[1..w]$, we define an array $MS[1..w]$ of (row, len) pairs as Matching Statistics of P with respect to S where, for each position $1 \leq j \leq w$:*

- $MS[j].\text{row}$ and P share a $MS[j].\text{len}$ -long match ending in j
- this match is left-maximal, i.e., $P[j - MS[j].\text{len}..j]$ does not match any sequence in S
- we have a mismatch iff $P[j]$ does not occur in column j and we represent this event as $MS[j].\text{row} = -$ and $MS[j].\text{len} = 0$

Using Matching Statistics, we can solve MPSC in $O(w)$ -time and $O(r)$ -space, where r represents the total number of runs in the PBWT [3]. We refer the reader to Cozzi et al. [8] and Bonizzoni et al. [3] for details about the algorithm used to compute Matching Statistics and SMEMs.

4 The genotype phasing problem via positional substrings

In this section, we extend the definition of positional substring cover to the case of a genotype and a panel S of strings. To accomplish this, we first define the concept of a *complementary positional substring* (CPS) for patterns P and P' . A CPS consists of two positional substrings (i, j, P) and (i, j, P') , which are bitwise complementary. For simplicity, a complementary positional substring will be simply identified by a triple (i, j, P) .

Building on this concept, we let Q be a w -long genotype query and S a set of w -long strings, then a PSC_g of genotype Q consists of a set C of positional substrings (i, j, Q) that occur in at least two rows of panel S , i.e., 2PS, and a CPS (k, l, S) of S , such that: (i) (k, l, Q) is a substring of Q of only 2's, and (ii) all positions t of Q are covered by C .

Next, we formally state the problem investigated in this paper. The core objective is to phase the query genotype Q by covering it with positional substrings (PSs) or complementary positional substrings (CPSs) derived from a panel of haplotypes S .

► **Problem 1.** Genotype-Positional-cover

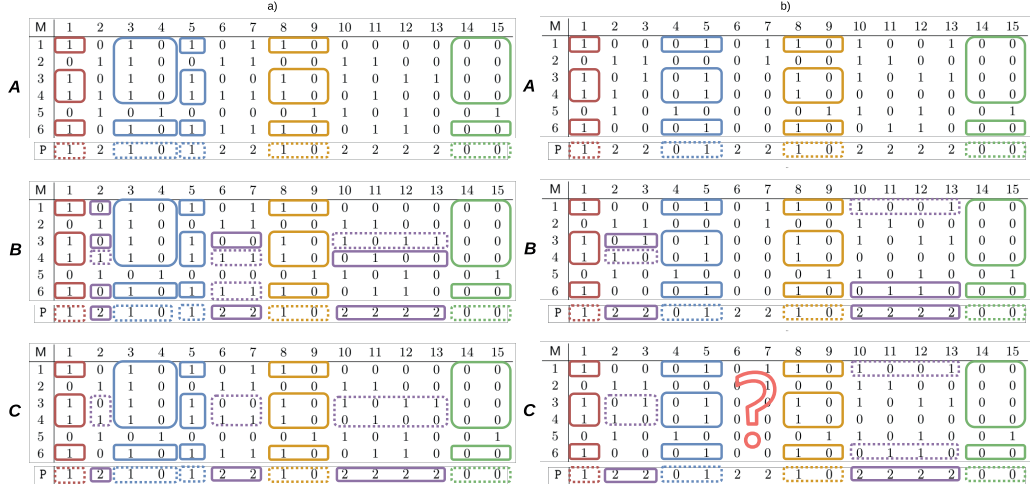
INPUT: a reference panel S of haplotypes and a genotype query Q ,

OUTPUT: a PSC_g for the query Q and the panel S .

In Problem 1, a natural optimization goal is to minimize ℓ , the size of the PSC_g . However, to further reduce the number of recombinations, we focus on minimizing the number of distinct pairs of haplotype sequences from S that contribute to the positional substrings used to cover Q . In other words, our goal is to minimize the number of 2-PSs and CPSs that originate from different haplotype pairs in S . We refer to this variant of Problem 1 as minimum recombination genotype-positional phasing. The problem has no recombinations if all 2-PSs and CPSs are derived from the same pair of haplotypes. This criterion will be incorporated into the heuristic method we propose to solve the genotype-positional phasing problem. The primary case we investigate in this paper is when the two haplotypes that resolve Q are already present in the panel S , which means that no recombinations occur. As we will discuss in Section 5, the case of no recombination has a straightforward and efficient solution. Finally, we recall that each 2PS is associated with a set of rows in the reference panel, and we will explicitly use this set to solve Problem 1.

5 Methods

In this section, we present our approach to solving Problem 1 using μ -PBWT. From this point forward, we make the simplifying assumption that each column in the reference panel contains at least two 0's and two 1's. This ensures that every homozygous position in the query genotype shares a 2PS with the reference panel, simplifying the presentation of our method. To effectively apply μ -PBWT, we first refine how it identifies matches and 2-positional substrings (2PSs) shared between the target genotype and the haplotype panel. Specifically, we introduce a constraint that limits the length of these 2PSs to at most two. The rationale behind this constraint will be explained at the end of this section, where we demonstrate how the use of 2-MPSCs affects the number of phased heterozygous sites.



■ **Figure 1** Example of solving the phasing problem a) assuming two haplotypes that fully explain the query genotype exist, i.e., in the absence of mutations and recombinations, and b) assuming this pair does not exist and we need to use different haplotype pairs for different genotype regions, i.e., with mutations and recombinations.

5.1 Computing 2PSs

First, we compute a set C of non-overlapping 2PSs of the genotype query with respect to the reference. The computation of 2PSs, which involves *at least* two rows in the reference panel, requires “resetting” the computation of two matching statistics each time $MS[i].len \geq 2$. This ensures that the computation restarts at $i + 1$, similar to handling a mismatch in column i . As a result, each entry in the matching statistics array now contains (likely not right-maximal) left-maximal matches with a length of at most two, involving at least two rows in the input panel. To prevent overlaps, we adopt a strategy similar to the one proposed in [3] for computing the leftmost MPSC set, thereby obtaining a set of non-overlapping 2PSs. Specifically, we modify that algorithm to incorporate 2-MSs, allowing us to skip matches shared by Q with only a single row in the panel, as well as positions corresponding to mismatches, which occur at heterozygous sites.

5.2 Solving the Haplotype Phasing Problem Without Recombinations

Now that we have computed the set C for query Q and reference panel S , we describe our approach for solving Problem 1. The key idea behind our approach is to analyze, for each iteration, two consecutive 2PSs of C to find possible haplotype pairs in S which can be used as anchors for phasing the heterozygous region of the query Q that lies between. In other words, given a heterozygous region in Q , we want to find a CPS that covers that region and shares at least a pair of rows with the previous 2PS and the next 2PS, i.e., the two anchors.

In practice, to solve a heterozygous region, we need to find two rows that complement each other bitwise. Given two consecutive 2PSs, we compute the intersection of the corresponding haplotype sets and use this intersection to extract the submatrix between them. Then, we naively find all the complementary haplotype pairs that contain the CPS explaining the substring of the genotype from the starting position of the first 2PS to the last position of the second 2PS. At each iteration, to avoid unnecessary computation and to consider previously computed pairs, we also need to store all the possible haplotype pairs which

explain the genotype query up to the current position. To formalize this, we denote S_{pair} as the current set of row pairs. In other words, at each iteration, instead of simply considering two consecutive 2PSs in C , we refine the possible pairs using the temporary result, that is, the set of pairs that explain the current prefix of the genotype query. In other words, the left anchor results from the filtering procedure. A similar filtering process is applied when two consecutive 2PSs occur without any heterozygous region between them.

To simplify the explanation of this approach, we consider the ideal scenario in which the reference panel contains the two haplotypes that fully explain the genotype query. We iterate through the query Q and the set C of 2PSs from left to right, storing haplotype pairs that are compatible with the query for a certain prefix. Note that, due to the assumption of haplotype existence, at each filtering step, there is always at least one pair within the current S_{pair} compatible with the pairs associated with the right 2PS, even if we have a heterozygous gap. In other words, the left anchor, which results from the filtering procedure, always consists of at least a row pair. To better understand this case, we refer to Figure 1.a. In Figure 1.a.A, we depict the 2PS set C shared between a reference panel M and a genotype query P (respectively S and Q in the previous discussion). Our iterative approach, which filters incorrect row pairs, is shown in Figure 1.a.B. In this example, at first, we have several row pairs after considering the first two 2PS: $S_{pair} = \{(1, 4), (3, 4), (4, 6)\}$. Note that, for example, we exclude row 2 since it cannot be used as a left anchor, having a mismatch at position 1. Continuing the iteration, we have two consecutive 2PSs, without any heterozygous gaps. The unique rows in S_{pair} need to be filtered by the rows associated with this new right 2PS but, in this case, each pair in S_{pair} is also present in the pairs set related to the right 2PS, so we do not need to delete any pair. We now need to consider the fourth 2PS as the right anchor. Note that row 1, in the substring that spans positions 6 to 7, does not complement any of the other rows in $\{3, 4, 6\}$, which are the remaining rows obtained from the previous step. After this iteration, we get a new set of possible pairs which explain the genotype in the pattern prefix up to position 9: $S_{pair} = \{(3, 4)\}$. Despite rows 3 and 6 sharing a complementary substring between positions 6 and 7, they are not considered because they are not present in the first pair set (sharing the same symbol in position 2). Finally, in the last iteration, the pair $(3, 4)$ is confirmed to be the only row pair that explains the genotype, being present in the set of pairs associated with the last 2PS and sharing complementary rows between positions 10 and 13. Figure 1.a.C summarizes the solution phasing for the given input.

5.3 Handling Recombination Cases

The scenario with zero recombinations occurs when the sample belongs to an individual already present in the panel or the analysis is restricted to a region without recombinations within the panel. However, in practical cases where sequencing errors occur or the reference panel has a limited number of haplotypes, the introduction of mutations and recombinations becomes necessary. It is important to note that a mutation can create a small recombination event around the mutation site. In this context, recombination refers to different segments of the genotype being explained by different haplotype pairs from the reference panel.

We now describe how our method works in this scenario. Again, we proceed to scan the set of 2PSs from the left to the right, by processing consecutive 2PSs using also the current pairs set S_{pair} , according to the approach described above whenever we follow two haplotypes that are present in the panel.

When the above assumption (i.e. the assumption of having zero recombinations) does not hold, we need to manage two consecutive 2PSs differently (filtered by S_{pair}). Specifically, we need to account for two possible cases. Recall that each adjacent 2PS is associated with

a list of row pairs, and we are considering the case where their intersection is empty. This leads to two possible scenarios: (1) there exist two row pairs – one from each 2PSs – that share at least one common haplotype but not both; or (2) no such two pairs exist. Recall that the row pairs associated with the 2PSs are filtered by considering the row pairs we have in S_{pair} . In the first scenario, only one haplotype recombines at this position between the two 2PSs, whereas in the second scenario, both haplotypes that explain the query undergo recombination at this position. In both cases, we store the temporary results up to the end of the first 2PS, then restart the iteration using all possible row pairs associated with the right 2PS. At this point, S_{pair} is stored and is updated to reflect the list of row pairs associated with the right 2PS.

In the second case, we encounter two consecutive 2PSs (after filtering by S_{pair}) separated by heterozygous regions that cannot be explained. This situation leads to two possible scenarios. First, a CPS covers the heterozygous region, but its associated list of row pairs does not overlap with any of the currently available pairs obtained from previous iterations. Second, the two anchor 2PSs do not share any associated row pairs in the reference panel. The latter case also involves any case in which there is no CPS in the heterozygous region. Both cases require different considerations when determining the compatible haplotype pairs to proceed.

In this situation, we can try to reset the current set S_{pair} by updating it with all the pairs associated with the left anchor, assuming a recombination. If we have pairs shared by this new anchor and the one related to the right 2PS that explain the underlying heterozygous region, we can continue to the next iteration using these pairs as S_{pair} after storing the previous S_{pair} in memory. On the other hand, if we do not have such pairs or simply if the CPS is not associated with any pair of rows in the panel, we skip those heterozygous sites and start the iteration again from the right 2PS. This implies that we need to update S_{pair} in the list of rows associated with the right 2PS but that these heterozygous sites will remain unphased. Observe that each time we need to reset the set of possible pairs and store the current S_{pair} , we add the last position in which it was consistent with the target genotype. In other words, each time we have a recombination, we store the current S_{pair} and the last position in which it explains a genotype substring.

After scanning the entire input genotype, we have a list of possible sets S_{pair} that cover various regions of the genotype, most likely all of them. Each genotype region can possibly be explained by more than one pair, so we must select the optimal haplotype pair. The selection is guided by an optimization function that aims to minimize the number of recombinant haplotypes. From a combinatorial perspective, we aim to find a list of haplotype pairs, one per region, that minimizes the number of pairs. This is the list of recombinant haplotypes that explain the input genotype. To compute this efficiently, we use a greedy algorithm to compute this list of pairs. Specifically, we sort all pairs by frequency and select the most frequent pair for each genotype region. In this way, for each genotype region, we get just one haplotype pair, which is the most common overall among the pairs that cover that region, i.e., the pair that covers more other regions. If some pairs have the same frequency, the selection is further weighted based on the pairs' choices made in previous regions to ensure consistency.

Figure 1.b illustrates the simplest case of recombination. In Figure 1.b.A, considering the first two 2PSs, we obtain $S_{pair} = \{(3,4)\}$ as the current set of pairs. However, this pair cannot explain the genotype at positions 6 and 7, even resetting the left anchor not having any CPS in that region, so this region remains unphased. Using the previously described approach, as shown in Figure 1.b.B, we restart from the next 2PS pair and obtain

a) 2-MPSC										
M	01	02	03	04	05	06	07	08	09	10
1	1	1	1	1	1	0	0	0	0	1
2	1	0	0	1	1	1	1	0	0	1
3	1	1	0	1	1	1	1	0	1	1
4	1	0	0	0	0	1	1	1	1	1
P	1	2	2	1	1	1	1	2	2	1

b) 2PSC										
M	01	02	03	04	05	06	07	08	09	10
1	1	1	1	1	1	0	0	0	0	1
2	1	0	0	1	1	1	1	0	0	1
3	1	1	0	1	1	1	1	0	1	1
4	1	0	0	0	0	1	1	1	1	1
P	1	2	2	1	1	1	1	2	2	1

Figure 2 Example for advantages of using 2PS and CPS against MPSC. In a) we show that using MPSC we should not be able to phase the two heterozygous regions between sites 2-3 and 8-9, having a recombination inside the MPSC at sites 5-6, while in b) we show that using 2PS and CPS, those regions are correctly phased.

$S_{pair} = \{(1,6)\}$, which explains the last genotype region. At this point, we also store in memory $S_{pair} = \{(3,4)\}$ and the last consistent position with that set: 5. At the end of the iteration, not having other possible pairs, we store $S_{pair} = \{(1,6)\}$ and position 15. This results in a recombination event between (3,4) and (1,6) even though we cannot infer the exact haplotype configuration at positions 6 and 7, i.e. those sites remain unphased.

5.4 Advantages of 2PSs against 2-MPSCs

After explaining our method, it becomes clear why we needed to avoid using classical 2-MPSC in favor of 2PS. As shown in Figure 2.a, if we rely on the use of 2-MPSC, we require less flexibility in how we detect anchors. In detail, we should miss some recombinations that occur in a position inside a 2-MPSC. In fact, using 2-MPSCs as in the example, we have that the two heterozygous regions cannot be explained by any row pair despite having a CPS which covers that substring. The first and second MPSC do not share any associated row pair, which explains the heterozygous region in between positions 2 and 3. The same happens when restarting the iteration from the entire second 2-MPSC and considering the last 2-MPSC: the heterozygous region between positions 8 and 9 cannot be explained in any way.

However, using 2PS, as in Figure 2.b, splitting the central MPSC into two 2PS, we can detect the pair $S_{pair} = \{1,2\}$ that explains the genotype up to position 5, including $P[2..3]$, and the pair $S_{pair} = \{2,4\}$ that explains the genotype from position 6 to the end of the genotype query, including $P[8..9]$.

5.5 Time and Space Complexity

For the time and space complexity of MS and the leftmost MPSC, refer to [2, 3, 8]. Although computing 2PSs and MPSC share the same time complexity, the key difference lies in the number of rows that share the same match. Since 2PSs generally involve smaller matches, more rows tend to share them, increasing the computational overhead.

Additionally, our approach has a time complexity that is quadratic in the number of possible haplotype pairs shared between the two anchors. In terms of extra space requirements, we store the input panel along with a vector of sparse bitvectors, enabling efficient reconstruction of submatrices. This avoids recomputing them using μ -PBWT in $O(h \log \rho)$, with ρ as the total number of runs in the PBWT. Furthermore, we must consider that storing all 2PSs and the current set of haplotype pairs can become memory intensive, having in the worst case a total number of pairs which is quadratic in the number of haplotypes.

■ **Table 1** Total number of heterozygous variation sites as the mutation rate varies.

Mutation rate	No. of het. sites
0 %	60 604
1 %	60 862
3 %	61 362
5 %	61 995
10 %	63 153
20 %	65 265

6 Results

In this section, we present some preliminary experimental results of our approach. Since this approach is a proof-of-concept, the current implementation is inefficient for most subtasks, from the extraction of complementary rows to the greedy algorithm used to select the best pairs of haplotypes. All of these steps could be further optimized in the future, both theoretically and practically.

To assess the applicability of our theoretical results, we integrate the genotype phasing algorithm into the μ -PBWT codebase and tested it in a simple genotype phasing scenario. For this experiment, we considered the HGSVC2 reference panel [7, 10] for human chromosome 2, with 68 haplotypes (34 samples) and 229 300 biallelic variation sites. We randomly selected a sample (*target sample*) from this reference panel as a query, combining its haplotypes into an unphased genotype target. The target sample we selected has 60,604 heterozygous sites ($\sim 26\%$ of the total number of variants on chromosome 2).

We used VCF as input format where 0|0 and 1|1 represent phased homozygous sites, 0|1 and 1|0 phased heterozygous sites, 0/0 and 1/1 unphased homozygous sites, and 0/1 unphased heterozygous sites.

We compared μ -PBWT with Beagle [5, 6], a state-of-the-art phasing tool that is also based on the use of a reference panel. We ran Beagle with default parameters, using the HapMap [23] chromosome 2 genetic map for the GRCh38 reference genome.

We ran our experiments on a machine equipped with an Intel Xeon CPU E5-4610 v2 (2.30GHz), 256 GB of RAM, 8 GB of swap, and Ubuntu 20.04.6 LTS 64-bit with kernel 5.15.0. We measured execution times and peak memory usage using the Unix `time` command.

Moreover, we simulate mutations in the target genotype by replacing the genotype at each position with probability ε . We considered five different values for ε : 1%, 3%, 5%, 10%, and 20%. We refer to ε as *mutation rate*. Table 1 summarizes the total number of heterozygous sites after the mutation step. As explained above, mutations cause small spurious recombinations at nearby sites. Thus, this small experiment allows us to test more realistic scenarios.

The accuracy of the results was evaluated by considering (i) the *switch error rate*, i.e., the rate of phase changes between the predicted haplotypes w.r.t. the ground truth haplotypes, (ii) the *mismatch rate*, i.e., the rate at which two switch errors occur at consecutive positions, and (iii) the percentage of heterozygous sites that have been phased. The three metrics were computed using the utility script publicly shared by the developers of HapCUT2 [11]¹, considering the two haplotypes of the target sample as ground truth.

¹ <https://github.com/vibansal/HapCUT2/tree/master/utilities>

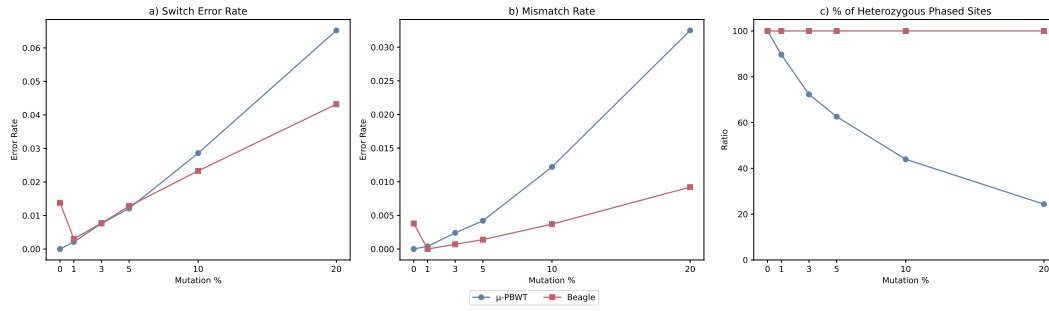


Figure 3 Various statistical results of μ -PBWT and Beagle at different genotype mutation rates. In a) we have the switch error rate, in b) the mismatch rate, and in c) the percentage of heterozygous sites phased over the total number of heterozygous sites.

Figure 3 presents the accuracy obtained by Beagle and μ -PBWT at different mutation rates (detailed results are available in Table 2). The results obtained in the scenario without mutations (i.e., with a mutation rate equal to 0%) are consistent with how the two tools approach the solution of the phasing problem. Beagle is a statistical approach that is not expected to find the exact solution even if the target genotype can be perfectly explained by two haplotypes in the reference panel. In contrast, μ -PBWT implements a combinatorial approach which is designed to always find a haplotype pair that explains the target, if such a pair exists in the reference panel. In fact, in this scenario μ -PBWT correctly identifies the two haplotypes of the target sample, so both *switch error rate* and *mismatch rate* are equal to 0.0%, while Beagle exhibits some fluctuations, with both metrics greater than zero.

As the mutation rate increases, the accuracy of μ -PBWT decreases dramatically. In fact, as we can see in Figure 3c, even with a mutation rate of 5%, μ -PBWT is able to phase only 62.6% of heterozygous sites. We recall that μ -PBWT does not phase heterozygous sites that are not explained by any haplotype pair that can be used as both a left and a right anchor. As a consequence, the higher the mutation rate, the larger the heterozygous genotype regions that do not have valid anchors and thus remain unphased. This fact further demonstrates the dependency of our approach on the data available in the reference panel, which benefits from panels including a large number of haplotypes. In addition to this strong dependency on the reference panel, our approach is unable to estimate haplotype pairs that differ from the ones obtained using the positional cover. Thus, around each mutation site, μ -PBWT will report pairs of haplotypes that differ from the ones of the target sample. However, a manual inspection of the results revealed that one of the two reported haplotypes often coincides with a haplotype of the target sample. However, these cases are handled better by Beagle, again due to its probabilistic approach, which seems to be more robust against these mutations and successfully phases each genotype site.

Figure 4 presents the computational resources used by the two approaches. The running times of the two approaches are comparable. However, the running time of μ -PBWT increases as the mutation rate increases, since, as expected, the number of haplotype switches that μ -PBWT has to compute (instead of simply continuing the match) also increases. Each time the algorithm cannot proceed with any pair, it has to start again, considering larger sets of possible pairs whose analysis increases both the overall execution time and RAM usage. Note that enlarging the number of reference samples would worsen performance in terms of indexing and storing the reference panel, but it would likely reduce the number of possible

10:14 Phasing Data from Genotype Queries via the μ -PBWT

■ **Table 2** Comparison of various genotype phasing accuracy metrics (such as the switch error rate and the mismatch rate) of Beagle and μ -PBWT as the mutation rate varies.

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

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

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

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

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

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

pairs to consider at each iteration since longer matches would become more likely. On the other hand, the running time of Beagle remains constant as the mutation rate increases since it does not depend on the number of haplotype switches in the reported solution.

As expected, μ -PBWT is very memory-efficient, requiring approximately 7 times less memory than Beagle. These results show that μ -PBWT could be run on low-end hardware or, on the public cloud, at a fraction of the cost of running Beagle. We argue that further refinements and improvements to the model and algorithm would not significantly impact memory requirements, since memory usage in genotype phasing approaches is often dominated by the space required to store the reference panel and μ -PBWT employs a very compact and efficient index representation of the panel.

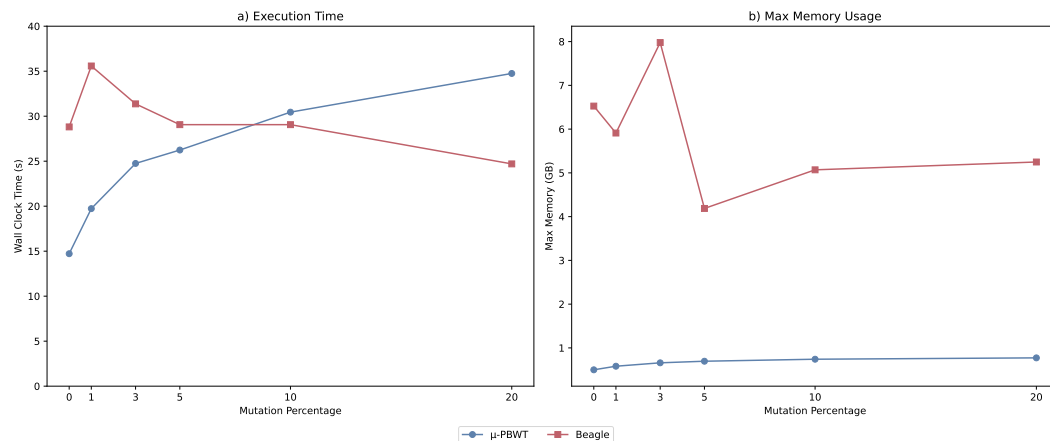


Figure 4 a) Running time and b) peak memory usage (right) of μ -PBWT and Beagle at different mutation rates.

7 Conclusion

In this paper, we present a proof of concept showing how μ -PBWT can be used to solve the genotype phasing problem. First, we outline the theoretical framework underlying our method, based on the foundations presented in [3]. We then introduce our combinatorial approach to tackling the phasing problem, considering both scenarios: (i) when a haplotype pair fully explains the target genotype and (ii) when multiple haplotype pairs are required to cover different genotype regions. Experimental results indicate that our approach is sensitive to the size of the reference panel, particularly when the mutation rate increases.

Improving the resolution of haplotype reconstruction in complex genomic regions remains an important direction for future work. In particular, anchoring heterozygous regions using flanking homozygous segments may enhance stability during phasing, while incorporating methods to detect recombination events within heterozygous regions could improve accuracy in the presence of structural variation and mutation.

These enhancements build on the current strengths of the proposed method, which, in mutation-free settings, reliably identifies the correct haplotype pair. Furthermore, consistent with the evaluations of μ -PBWT, the method achieves high efficiency with minimal memory requirements, making it suitable for execution on commodity hardware. Taken together, these characteristics provide a strong foundation for future comparisons with state-of-the-art phasing tools, particularly in real-world scenarios involving complex variant patterns and sequencing noise.

References

- 1 Paola Bonizzoni. A linear-time algorithm for the perfect phylogeny haplotype problem. *Algorithmica*, 48:267–285, 2007. doi:10.1007/S00453-007-0094-3.
- 2 Paola Bonizzoni, Christina Boucher, Davide Cozzi, Travis Gagie, Dominik Köppl, and Massimiliano Rossi. Data Structures for SMEM-Finding in the PBWT. In *International Symposium on String Processing and Information Retrieval*, pages 89–101. Springer, 2023. doi:10.1007/978-3-031-43980-3_8.
- 3 Paola Bonizzoni, Christina Boucher, Davide Cozzi, Travis Gagie, and Yuri Pirola. Solving the minimal positional substring cover problem in sublinear space. In *35th Annual Symposium*

- on *Combinatorial Pattern Matching (CPM 2024)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.
- 4 Paola Bonizzoni, Gianluca Della Vedova, Riccardo Dondi, and Jing Li. The haplotyping problem: an overview of computational models and solutions. *Journal of Computer Science and Technology*, 18:675–688, 2003. doi:10.1007/BF02945456.
 - 5 Brian L Browning, Xiaowen Tian, Ying Zhou, and Sharon R Browning. Fast two-stage phasing of large-scale sequence data. *The American Journal of Human Genetics*, 108(10):1880–1890, 2021.
 - 6 Brian L Browning, Ying Zhou, and Sharon R Browning. A one-penny imputed genome from next-generation reference panels. *The American Journal of Human Genetics*, 103(3):338–348, 2018.
 - 7 Mark JP Chaisson, Ashley D Sanders, Xuefang Zhao, Ankit Malhotra, David Porubsky, Tobias Rausch, Eugene J Gardner, Oscar L Rodriguez, Li Guo, Ryan L Collins, et al. Multi-platform discovery of haplotype-resolved structural variation in human genomes. *Nature communications*, 10(1):1784, 2019.
 - 8 Davide Cozzi, Massimiliano Rossi, Simone Rubinacci, Travis Gagie, Dominik Köppl, Christina Boucher, and Paola Bonizzoni. μ -PBWT: a lightweight r-indexing of the PBWT for storing and querying UK Biobank data. *Bioinformatics*, 39(9):btad552, 2023. doi:10.1093/BIOINFORMATICS/BTAD552.
 - 9 Richard Durbin. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 2014. doi:10.1093/BIOINFORMATICS/BTU014.
 - 10 Peter Ebert, Peter A Audano, Qihui Zhu, Bernardo Rodriguez-Martin, David Porubsky, Marc Jan Bonder, Arvis Sulovari, Jana Ebler, Weichen Zhou, Rebecca Serra Mari, et al. Haplotype-resolved diverse human genomes and integrated analysis of structural variation. *Science*, 372(6537):eabf7117, 2021.
 - 11 Peter Edge, Vineet Bafna, and Vikas Bansal. Hapcut2: robust and accurate haplotype assembly for diverse sequencing technologies. *Genome research*, 27(5):801–812, 2017.
 - 12 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000. doi:10.1109/SFCS.2000.892127.
 - 13 William A Freyman, Kimberly F McManus, Suyash S Shringarpure, Ethan M Jewett, Katarzyna Bryc, 23, Me Research Team, and Adam Auton. Fast and robust identity-by-descent inference with the templated positional burrows–wheeler transform. *Molecular Biology and Evolution*, 38(5):2131–2151, 2021.
 - 14 Dan Gusfield. Haplotyping as perfect phylogeny: conceptual framework and efficient solutions. In *Proceedings of the sixth annual international conference on Computational biology*, pages 166–175, 2002. doi:10.1145/565196.565218.
 - 15 Robin J Hofmeister, Diogo M Ribeiro, Simone Rubinacci, and Olivier Delaneau. Accurate rare variant phasing of whole-genome and whole-exome sequencing data in the uk biobank. *Nature genetics*, 55(7):1243–1249, 2023.
 - 16 Na Li and Matthew Stephens. Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. *Genetics*, 165(4):2213–2233, 2003.
 - 17 Ardalan Naseri, Erwin Holzhauser, Degui Zhi, and Shaojie Zhang. Efficient haplotype matching between a query and a panel for genealogical search. *Bioinformatics*, 35(14):i233–i241, 2019. doi:10.1093/BIOINFORMATICS/BTZ347.
 - 18 Simone Rubinacci, Olivier Delaneau, and Jonathan Marchini. Genotype Imputation using the Positional Burrows Wheeler Transform. *PLoS Genetics*, 16(11):e1009049, 2020.
 - 19 Simone Rubinacci, Robin J Hofmeister, Bárbara Sousa da Mota, and Olivier Delaneau. Imputation of low-coverage sequencing data from 150,119 uk biobank genomes. *Nature Genetics*, 55(7):1088–1090, 2023.

- 20 Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. Haplotype threading using the positional Burrows-Wheeler transform. In *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
- 21 Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. d-pbwt: dynamic positional burrows-wheeler transform. *Bioinformatics*, 37(16):2390–2397, 2021. doi:10.1093/BIOINFORMATICS/BTAB117.
- 22 Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. Minimal positional substring cover is a haplotype threading alternative to Li and Stephens Model. *Genome Research*, 33(7):1007–1014, 2023. doi:10.1101/gr.277673.123.
- 23 The International Hapmap 3 Consortium. Integrating common and rare genetic variation in diverse human populations. *Nature*, 467(7311):52–58, September 2010. doi:10.1038/nature09298.
- 24 Victor Wang, Ardalan Naseri, Shaojie Zhang, and Degui Zhi. Syllable-PBWT for space-efficient haplotype long-match query. *Bioinformatics*, 39(1):btac734, 2023. doi:10.1093/BIOINFORMATICS/BTAC734.
- 25 Yaoling Yang, Richard Durbin, Astrid KN Iversen, and Daniel J Lawson. Sparse haplotype-based fine-scale local ancestry inference at scale reveals recent selection on immune responses. *medRxiv*, pages 2024–03, 2024.
- 26 William Yue, Ardalan Naseri, Victor Wang, Pramesh Shakya, Shaojie Zhang, and Degui Zhi. P-smoother: efficient PBWT smoothing of large haplotype panels. *Bioinformatics Advances*, 2(1):vbac045, 2022.

Circular Dictionary Matching Using Extended BWT

Wing-Kai Hon ✉ 

National Tsing Hua University, Hsinchu, Taiwan

Rahul Shah ✉ 

Louisiana State University, Baton Rouge, LA, USA

Sharma V. Thankachan ✉ 

North Carolina State University, Raleigh, NC, USA

Abstract

The *dictionary matching problem* involves preprocessing a set of strings (patterns) into a data structure that efficiently identifies all occurrences of these patterns within a query string (text). In this work, we investigate a variation of this problem, termed *circular dictionary matching*, where the patterns are circular, meaning their cyclic shifts are also considered valid patterns. Such patterns naturally occur in areas such as bioinformatics and computational geometry. Based on the *extended Burrows-Wheeler Transformation* (eBWT), we design a space-efficient solution for this problem. Specifically, we show that a dictionary of d circular patterns of total length n can be indexed in $n \log \sigma + O(n + d \log n + \sigma \log n)$ bits of space and support circular dictionary matching on a query text T in $O((|T| + \text{occ}) \log n)$ time, where σ represents the size of the underlying alphabet and occ represents the output size.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String algorithms, Burrows-Wheeler transformation, suffix trees, succinct data structures

Digital Object Identifier 10.4230/OASIScs.Manzini.2025.11

Funding Supported by the US National Science Foundation (NSF) under Grant Numbers 2137057, 2434261 (R Shah) and 2315822 (S Thankachan).

Acknowledgements We thank all the anonymous reviewers (and Mano Prakash Parthasarathi) for their valuable feedback, which helped improve this paper. We also thank Travis Gagie for pointing out the independent work by Cotumaccio [25].

1 Introduction

Text indexing is a fundamental problem in computer science, where we are given a long string (the text) for preprocessing into a data structure (the index) that supports efficient substring matching. Specifically, when a shorter string (the pattern) is provided as a query, the goal is to find all occurrences of the pattern as a substring within the text. Data structures such as suffix trees and suffix arrays are commonly used for this task [67, 53]. However, a major drawback of suffix trees and suffix arrays is their significant space requirement. To address this issue, compact and space-efficient solutions have been developed [28, 41, 62]. Among these, FM-index [28], Compressed Suffix Arrays [41], and Compressed Suffix Trees [62] are particularly notable due to their historical significance. We refer to [57] for a comprehensive survey on this topic.

Dictionary matching is an orthogonal problem to text indexing. Here, we are given a set of patterns (the dictionary) and our aim is to design a data structure capable of finding all occurrences of these patterns as substrings within a query text. Solutions to this problem have applications in areas such as intrusion detection and bioinformatics, where they are used to identify known DNA or protein sequences in genomic data. The classic



© Wing-Kai Hon, Rahul Shah, and Sharma V. Thankachan;
licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 11; pp. 11:1–11:14

OpenAccess Series in Informatics



OASIS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

solution to this problem is the Aho-Corasick automaton [1], which efficiently matches multiple patterns simultaneously. However, as before, this data structure also suffers from a high space requirement. To address this issue, compact and space-efficient solutions have been proposed. The current best succinct-space result is due to Belazzougui [10], where an xBWT-like technique [27] is applied for the succinct encoding of the Aho-Corasick automaton. We refer the reader to [55, 59, 16] for follow-up work in this direction, including an entropy-compressed solution [45]. For an alternative solution based on sparse suffix trees, see [42]. A wide range of variations on this problem have been studied, including dynamic dictionary matching [3, 18, 46, 26, 65], online dictionary matching [35, 51, 6], dictionary matching in streaming model [36, 34, 37], dictionary matching with errors or gaps [5, 4, 24, 44, 47], internal dictionary matching [19], dictionary matching under parameterized or order-preserving models [52, 32, 33], 2D dictionary matching [2, 58], etc.

In this work, we investigate another variant of dictionary matching, termed *circular dictionary matching* [48]. Here the patterns in the given dictionary are circular, meaning their cyclic shifts are also considered valid patterns. For example, the set of cyclic shifts of `abcd` is `{abcd, bcda, cdab, dabc}`, whereas that of `abab` is `{abab, baba}`. Note that circular patterns arise naturally in applications in bioinformatics and computational geometry. For instance, the genomes of many viruses, such as the herpes simplex virus (HSV-1), exist as circular strings [64]. In computational geometry, polygons are often represented by listing the coordinates of their vertices in clockwise order. The problem of matching a circular pattern, or a collection of circular patterns, in a given text has been extensively studied from an algorithmic perspective [23, 22, 20, 7, 49, 9, 40, 21]. Our objective is to design a solution for the data-structural version of this problem.

► **Problem 1** (Circular Dictionary Matching [48]). *Given a set of d circular patterns $\mathcal{D} = \{P_1, P_2, \dots, P_d\}$ of total length n on an alphabet $\Sigma = [\sigma]$, design a data structure (called an index) that supports the following query efficiently:*

- *Input: A text T over Σ*
- *Output: All occurrences of all circular patterns in T . Specifically, every substring of T that corresponds to a cyclic shift of any pattern in $\mathcal{D}$; a substring can be denoted by the starting and ending position in the text. We use `occ` to denote the output size.*

Problem 1 is equivalent to standard dictionary matching on an expanded dictionary $\mathcal{D}'$, which includes all circular patterns in $\mathcal{D}$ along with their cyclic shifts. Therefore, one approach is to index $\mathcal{D}'$, which is clearly inefficient, as the sum of the sizes of all patterns in $\mathcal{D}'$ can be quadratic to that of $\mathcal{D}$ in the worst case. To that end, we presented a succinct space index of space $n \log \sigma + O(n + d \log n + \sigma \log n)$ bits and query complexity $O(|T| \log^2 n + \text{occ} \log n)$ in our earlier work [48], which we improve upon in this paper.

► **Theorem 2.** *For the circular dictionary matching problem, there exists an index requiring space $n \log \sigma + O(n + d \log n + \sigma \log n)$ bits and query time $O((|T| + \text{occ}) \log n)$. Here n denotes the total length of circular patterns in the dictionary $\mathcal{D}$, $d = |\mathcal{D}|$, σ denotes the size of the alphabet set, T denotes the text (and $|T|$ denotes its length) that comes as a query, and `occ` denotes the output size.*

Our new solution is based on a structure similar to the FM-index [28], which is built from the *extended Burrows-Wheeler Transformation* (eBWT) [54], an extension of the traditional Burrows-Wheeler Transformation (BWT) [15] that works over multiple strings [8, 13, 14, 17, 60]. We remark that the application of the eBWT to the space-efficient indexing of circular patterns is not new. For example, in recent work, Boucher et al. [14] showed how to

construct an eBWT-based index for a collection of strings with full cyclic pattern matching functionality in compressed space. This problem is distinct from the one we address in this work; nonetheless, the techniques employed in this paper are closely similar to those in their work.

► **Remark.** In work parallel to ours, Cotumaccio [25] independently achieved an index with similar space-time complexity; specifically $n \log \sigma(1 + o(1)) + O(n + d \log n)$ bits of space and $O((|T| + \text{occ}) \log n)$ query time.

2 Preliminaries

Let $\Sigma = \{0, 1, 2, \dots, \sigma - 1\}$ denote the underlying alphabet. For a string $S[1..s]$, we use $|S|$ to denote its length, $S[i..j]$ to denote its substring $S[i] \circ S[i+1] \circ \dots \circ S[j]$ when $i \leq j$ and an empty string when $i > j$, where $\circ$ denotes concatenation. In addition, $S[i..j] = S[i..j-1]$ and $S(i..j) = S[i+1..j]$. For an integer $k \geq 0$, S^k denotes an empty string if $k = 0$, and $S \circ S^{k-1}$ otherwise, while S^∞ denotes the concatenation of infinite copies of S . The *root* of a string S , $\text{root}(S)$ is defined as the shortest string R , such that $S = R^k$ for some integer k ; we call S *primitive* if $S = \text{root}(S)$. For example, **a**, **ab**, **ababa** are primitive, whereas **abab** = $(\text{ab})^2$ is not. The result below follows from Fine and Wilf's theorem [29] (also see Theorem 1 in [12]).

► **Lemma 3.** *Let X and Y be two distinct primitive strings. Then, the length of the longest common prefix of their infinite repetitions, X^∞ and Y^∞ , is at most $|X| + |Y| - \gcd(|X|, |Y|)$, where $\gcd(\cdot, \cdot)$ denotes the greatest common divisor. This implies that although X^∞ and Y^∞ are infinite in length, their lexicographic order can be established by comparing a bounded number of characters.*

Rank and Select Queries. For a character $\alpha \in \Sigma$, $\text{rank}_S(i, \alpha)$ denotes the number of occurrences of α in $S[1..i]$, $\text{select}_S(j, \alpha)$ denotes the location of j -th occurrence of α in S , and $\text{partialRank}_S(i) = \text{rank}_S(i, S[i])$. There exist different space-time trade-offs for supporting these operations. For example, a wavelet tree structure of space $s \log \sigma + o(s) + O(\sigma \log s)$ bits can support all three operations in $O(\log \sigma)$ time [39]. See [11, 38] for faster alternatives.

For our purpose, we use an $s \log \sigma + O(s + \sigma \log s)$ -bit structure, which requires $O(\log \log s)$ time for rank and only $O(1)$ time for others. The idea is to use the following result (indexable dictionaries) by Raman et al. [61]: a binary string $B[1..s]$ with f number of 1s can be represented in $f \log(s/f) + O(f)$ bits and support $\text{rank}_B(\cdot, \cdot)$ in $O(\log \log s)$ time, $\text{select}_B(j, 1)$ and $\text{partialRank}_B(i)$ in $O(1)$ time, where $\text{partialRank}_B(i) = \text{rank}_B(i, B[i])$ if $B[i] = 1$ and is an arbitrary number otherwise. Therefore, for each $\alpha \in \Sigma$, we maintain the following bit vectors as indexable dictionaries: $B_\alpha[1..s]$, where $B_\alpha[i] = 1$ iff $S[i] = \alpha$. Let f_α be the number of occurrences of α in S . Then the total space is $\sum_{\alpha \in \Sigma} f_\alpha \log(s/f_\alpha) + O(f_\alpha) = s \log \sigma + O(s)$ bits. To enable the operations on S , we employ the following connections: $\text{rank}_S(i, \alpha) = \text{rank}_{B_\alpha}(i, 1)$, $\text{select}_S(j, \alpha) = \text{select}_{B_\alpha}(j, 1)$, and $\text{partialRank}_S(i) = \text{partialRank}_{B_{S[i]}}(i)$.

Range Minimum Queries (RMQ). Let A be an array of numbers, we can design an $2|A| + o(|A|)$ bit structure that supports the following query [30] in constant time: input is a range $[a, b]$ and output is position $t \in [a, b]$, such that $A[t]$ is the smallest element in $A[a, b]$. For answering RMQ, we do not need to explicitly store A .

Tree Topology. The topology of an ordinal tree can be encoded in linear number of bits and support various tree operations in $O(1)$ time [63, 56]. The ones relevant to us are finding the parent of a node, the range of leaves (i.e., the first and last leaves) in the subtree of a node, and the lowest common ancestor (LCA) of two nodes. Here a node is referred to by its pre-order rank.

3 Extended Burrows-Wheeler Transformation and Related Structures

In this section, we formally define the *extended BWT*, along with data structures analogous to suffix trees and suffix arrays for circular strings, which we refer to as the *extended suffix tree* and *extended suffix array*. The definition of extended BWT is first proposed by Mantaci et al. [54]. For related concepts, different terminologies have been used in prior work; for example, the extended suffix array was referred to as the circular suffix array in [48, 43], and as the generalized conjugate array in recent work by Boucher et al. [14]. However, it is straightforward to observe that these structures are equivalent. As we will show, the succinct encoding of the extended suffix array follows naturally from the original ideas of the FM-index. However, the definition of the extended suffix tree, or the use of it to solve pattern matching problems, seem to be new. In the following, we will follow the terminologies in [48, 43] to define the extended BWT, as they align more naturally in defining (more importantly *compressing*) the extended suffix tree than those in the original paper [54]. Also, we remark that encoding certain components of the extended suffix tree requires non-trivial modifications of known results.

Let $\mathcal{Q} = \{Q_1, Q_2, Q_3, \dots, Q_r\}$ be a given set of r primitive strings, where $Q_i = Q_i[1..q_i]$ and $m = \sum_{i=1}^r q_i$. We call $Q_{i,k} = Q_i[k..q_i] \circ Q_i[1..k]$ the k -th *cyclic shift* of Q_i , where $k \in [1, q_i]$. Without loss of generality, we make an important assumption that no string in $\mathcal{Q}$ is a cyclic shift of another.¹ Next, we define the following sets:

$$\text{eSUFFIXES}(Q_i) = \{Q_{i,k}^\infty \mid k \in [1, q_i]\} \text{ and } \text{eSUFFIXES}(\mathcal{Q}) = \bigcup_{i=1}^r \text{eSUFFIXES}(Q_i)$$

Note that $\text{eSUFFIXES}(\mathcal{Q})$ is a collection of m strings, which are infinite in length, but pairwise distinct; therefore, their lexicographic order is well-defined. Moreover, by Fine and Wilf's theorem (see Lemma 3), the length of the longest common prefix between any two strings in $\text{eSUFFIXES}(\mathcal{Q})$ is at most $2m$.

The **Extended Suffix Tree** (denoted by eST) is a compacted trie of all strings in $\text{eSUFFIXES}(\mathcal{Q})$. It consists of m leaves, say $\ell_1, \ell_2, \dots, \ell_m$ in the left-to-right order, which are enumerated according to the lexicographic order of the infinite strings in $\text{eSUFFIXES}(\mathcal{Q})$. The number of internal nodes is at most $m - 1$. For any node u , $\text{str}(u)$ denotes the concatenation of edge labels on the path from the root to u and $\text{strlen}(u)$ denotes the length of $\text{str}(u)$. Note that $\text{strlen}(\cdot) < 2m$ for all internal nodes from Lemma 3. However, each edge connecting to a leaf has a label with an infinite length. This is unlike the standard suffix tree. Yet, the tree topology is well-defined, because the lexicographic order of strings in $\text{eSUFFIXES}(\mathcal{Q})$ is well defined². We remark that the idea of having leaf edges with

¹ As we will see, our goal is to efficiently represent all cyclic shifts of strings in the collection. If two strings are cyclic shifts of each other, they share the same set of shifts. Therefore, it's sufficient to store only one representative from each group of cyclically equivalent strings.

² Note that this remains true even when each infinite string in $\text{eSUFFIXES}(\mathcal{Q})$ is treated as a finite string by considering only its first $2m$ characters.

unspecified lengths is not new; e.g., see Ukkonen's suffix tree construction algorithm [66]. The *suffix range* of a string S is the maximal range $[sp, ep]$ where S is a prefix of $\text{str}(\ell_j)$ for all $j \in [sp, ep]$, and is NIL if no such j exists. The *range of leaves* of a node u is the suffix range of the string $\text{str}(u)$.

The **Extended Suffix Array** $\text{eSA}[1..m]$ is an array of pairs such that $\text{eSA}[j] = (i, k)$, where $\text{str}(\ell_j) = Q_{i,k}^\infty$, and the **Extended BWT** $\text{eBWT}[1..m]$ is a string, where $\text{eBWT}[j]$ is the last character of $Q_{i,k}$. We remark that the terminology used in the original definition of eBWT in [54] is slightly different (i.e., omega order), but there is no technical difference. Note that eST and eSA take $O(m \log m)$ bits, whereas eBWT takes only $m \log \sigma$ bits.

As an example, consider the set $\mathcal{Q} = \{Q_1, Q_2, Q_3, Q_4\}$, where $Q_1 = aab, Q_2 = ab, Q_3 = abb$ and $Q_4 = b$. Then eST consists of 9 leaves, such that

1. $\text{str}(\ell_1) = Q_{1,1}^\infty = aabaab \dots$
2. $\text{str}(\ell_2) = Q_{1,2}^\infty = abaaba \dots$
3. $\text{str}(\ell_3) = Q_{2,1}^\infty = ababab \dots$
4. $\text{str}(\ell_4) = Q_{3,1}^\infty = abbabb \dots$
5. $\text{str}(\ell_5) = Q_{1,3}^\infty = baabaa \dots$
6. $\text{str}(\ell_6) = Q_{2,2}^\infty = bababa \dots$
7. $\text{str}(\ell_7) = Q_{3,3}^\infty = babbab \dots$
8. $\text{str}(\ell_8) = Q_{3,2}^\infty = bbabba \dots$
9. $\text{str}(\ell_9) = Q_{4,1}^\infty = bbbbbb \dots$

The corresponding eSA[1..9] is (1, 1), (1, 2), (2, 1), (3, 1), (1, 3), (2, 2), (3, 3), (3, 2), (4, 1) and eBWT[1..9] is babbaabab.

3.1 Succinct Representation of eSA

We maintain eBWT in $m \log \sigma + O(m + \sigma \log m)$ bits to support **rank**, **select** and **partialRank** operations efficiently as described in Section 2. Similar to the FM-index, we now implement LF mapping and backward search [28].

Last-to-Front (LF) Mapping. Define $\text{eLF}(j) = j'$, where deleting the first character of $\text{str}(\ell_{j'})$ gives $\text{str}(\ell_j)$. The function $\text{eLF}(j)$ can also be described in terms of eBWT as $\text{eLF}(j) = \text{count}(\text{eBWT}[j]) + \text{partialRank}_{\text{eBWT}}(j)$. For any character $\alpha \in \Sigma$, $\text{count}(\alpha) = |\{i \in [1, m] \mid \text{eBWT}[i] < \alpha\}|$. For all $\alpha \in \Sigma$, this information can be stored explicitly in $O(\sigma \log m)$ bits, and the **partialRank** operation takes only $O(1)$ time. Therefore, the overall time for eLF operation is constant.

Backward Search. Given the suffix range $[sp, ep]$ of a string S , we can compute the suffix range $[sp', ep']$ of $\alpha \circ S$ for any $\alpha \in \Sigma$ in $O(\log \log m)$ time as follows. First compute the following

$$sp' = \text{count}(\alpha) + \text{rank}_{\text{eBWT}}(sp - 1, \alpha) + 1$$

$$ep' = \text{count}(\alpha) + \text{rank}_{\text{eBWT}}(ep, \alpha)$$

If $sp' \leq ep'$, then return $[sp', ep']$ as the suffix range of $\alpha \circ S$. Otherwise, conclude that the suffix range of $\alpha \circ S$ does not exist. The time complexity $O(\log \log m)$ comes from the time for rank operations.

To encode eSA, we replace it with a sparse eSA, where we store eSA values only at some sampled positions. Specifically, we store an entry (i, k) iff $k \bmod \Delta = 1$, where Δ be a parameter. This reduces the number of values stored to $O(d + m/\Delta)$, where d is the

number of indexed patterns; space is $O(\log m)$ bits per value. Suppose $\text{eSA}[j] = (i, k)$, then $\text{eSA}[\text{eLF}[j]] = (i, k - 1)$, $\text{eSA}[\text{eLF}[\text{eLF}[j]]] = (i, k - 2)$, etc., which means we can decode $\text{eSA}[j]$ (if not explicitly stored) by iteratively applying eLF (say h times) until we arrive at a positions where $\text{eSA}[\cdot] = (i, k')$ is stored. We then obtain $\text{eSA}[j] = (i, k' + h)$. The time complexity is $O(h)$ and $h < \Delta$. By fixing $\Delta = \lceil \log m \rceil$, we obtain the following result.

► **Lemma 4.** *There exists an $m \log \sigma + O(m + d \log m + \sigma \log m)$ -bit data structure that returns $\text{eSA}[j]$ for any given j in time $O(\log m)$.*

3.2 Succinct Representation of eST

We encode and maintain the topology of eST in $O(m)$ bits for supporting the relevant tree operations in $O(1)$ time. To encode the values of $\text{strlen}(\cdot)$, we modify the techniques introduced by Sadakane [62].

For all $i \in [1, r]$, define $\text{ePLCP}^i[1..q_i]$, where $\text{ePLCP}^i[k]$ is the length of the longest common prefix of $Q_{i,k}^\infty$ and the string in the set $\text{eSUFFIXES}(\mathcal{Q})$ that comes next in the lexicographic order (say $Q_{i',k'}^\infty$). Note that longest common prefix of $Q_{i,(k+1) \bmod q_i}^\infty$ and $Q_{i',(k'+1) \bmod q_{i'}}$ shares at least $\text{ePLCP}^i[k] - 1$ characters. So, $\text{ePLCP}^i[(k+1) \bmod q_i] \geq \text{ePLCP}^i[k] - 1$. By adding k on both sides and rearranging, we have

$$\text{ePLCP}^i[k \bmod q_i] + k \leq \text{ePLCP}^i[(k+1) \bmod q_i] + (k+1)$$

Letting $f_i(k) = \text{ePLCP}^i[k \bmod q_i] + k$, we have

$$f_i(1) \leq f_i(2) \leq \dots \leq f_i(|Q_i|) \leq f_i(|Q_i| + 1) = f_i(1) + |Q_i|$$

This means $f_i(k) : k \in [1, q_i]$ is non-decreasing and its range is $[f_i(1), f_i(1) + |Q_i|]$. We store $f_i(1)$ explicitly in $O(\log m)$ bits and use unary encoding for the rest. Specifically, for each i we store a binary string $B_i = 10^{f_i(2)-f_i(1)}10^{f_i(3)-f_i(2)}10^{f_i(4)-f_i(3)} \dots$ with $O(1)$ time rank/select supported [61]. To find $f_i(k)$, we simply count the number of 0s before k -th 1 and add $f_i(1)$. Subtracting k from this value gives $\text{ePLCP}^i[k]$. Space required for a fixed i is $O(|B_i| + \log m)$, where $|B_i| = O(q_i)$. Therefore, space over all i 's is $O(m + d \log m)$ bits.

We now present the last component for encoding $\text{strlen}(\cdot)$ values. Define array $\text{LCP}_e[1..m]$ as follows. Suppose $\text{eSA}[j] = (i, k)$, then $\text{LCP}_e[j] = \text{ePLCP}^i[k]$. We do not store LCP_e explicitly, but an RMQ structure over it in $2m + o(m)$ bits. To compute $\text{strlen}(u)$, find $[a, b]$, the range of in the subtree of u and the position $t \in [a, b]$ corresponding to the minimum in $\text{LCP}_e[a, b]$ using an RMQ. Then obtain $\text{eSA}[t] = (i', k')$ and $\text{strlen}(u) = \text{ePLCP}^{i'}[k']$. The time complexity (asymptotically) is equal to that of an eSA query.

► **Lemma 5.** *By associating an $O(m + d \log m)$ bit structure with eSA, we can find $\text{strlen}(\cdot)$ of any given node in time $O(\log m)$.*

This concludes our discussion of the main data structure. In the following section, we demonstrate how it can be used to solve the circular dictionary matching problem.

4 A Succinct Index for Circular Dictionary Matching

Traditional “non-circular” dictionary matching can be solved efficiently with a generalized suffix tree of the patterns, by successively finding the *locus* of each suffix of the query text T within the suffix tree, and then reporting the patterns that appear as the prefix of the corresponding suffix. For circular patterns, we show that this can be done analogously

with the extended suffix tree. Yet, there are some technical challenges. First, as patterns of different lengths can be represented by the same leaf, we need some care to verify if a pattern actually appears in a certain text position. Another, more difficult, challenge is to obtain the loci of the suffixes; a direct adaptation of the “non-circular” dictionary matching approach cannot bound the running time as desired. In the following, we will explain how such challenges can be solved.

Given a set of d circular patterns to index. We first make a critical step of replacing each pattern with its lexicographically smallest cyclic shift. Denote the resulting set by $\mathcal{D} = \{P_1, P_2, \dots, P_d\}$. We then collect the roots of all P_i 's and call it $\mathcal{Q} = \{Q_1, Q_2, \dots, Q_r\}$. The first step ensures that $\mathcal{Q}$ is invariant of the cyclic shifts of P_i 's. Let $n = \sum_i |P_i|$ and $m = \sum_i |Q_i|$. Note that $r \leq d$ and $m \leq n$.

4.1 The Data Structure

We construct and maintain the structures in Lemma 4 and Lemma 5 over the strings in $\mathcal{Q}$. Additional components are below.

1. We encode each pattern $P_i \in \mathcal{D}$ as a pair $(i', |P_i|)$, where $Q_{i'}$ is the root of P_i , equivalently $P_i = Q_{i'}^{|P_i|/|Q_{i'}|}$. For each $Q_{i'} \in \mathcal{Q}$, we maintain a list $\text{LIST}_{i'}$ of patterns (in encoded form) with $Q_{i'}$ being their root. Each list is sorted in the ascending order of the pattern lengths. Total space is $O(d \log n)$ bits. Therefore, given any (i', τ) , we can list all patterns with root $Q_{i'}$ and length $\leq \tau$ in optimal time.
2. We want to support the query $\text{REPORT}([a, b], \tau)$, which reports all patterns $P_i = (i', |P_i|)$ (i.e., in encoded form), where $|P_i| \leq \tau$ and $\text{eSA}[j] = (i', \cdot)$ for some $j \in [a, b]$. Define an array $\text{Length}[1, m]$, where $\text{Length}[j]$ is the length of the *shortest* pattern in $\text{LIST}_{i'}$, where $\text{eSA}[j] = (i', \cdot)$. We do not store this array, but an RMQ structure over it in $2m + o(m)$ bits. With that, we support the query using the following standard procedure.
 - When $a = b$, we decode i' , where $\text{eSA}[a] = (i', \cdot)$ and obtain the output from $\text{LIST}_{i'}$.
 - When $a \neq b$, we find the position $t \in [a, b]$ of the minimum element in $\text{Length}[a, b]$ using RMQ, and then make the query $\text{REPORT}([t, t], \tau)$. If it returns NIL (i.e., output is empty), we conclude that $\text{REPORT}([a, b], \tau)$ is also NIL. Otherwise, we continue our search for more answers in the remaining parts of the array, specifically in $[a, t)$ and $(t, b]$ using queries $\text{REPORT}([a, t], \tau)$ and $\text{REPORT}((t, b], \tau)$, recursively.

Let g be the output size. Then it can be observed that the original query is split into $O(1 + g)$ subqueries. Therefore, the time complexity is $O((1 + g) \log m)$.

3. We mark a node u in eST if it is the highest node (i.e., closest to root) for some $i \in [1, d]$, such that P_i or a cyclic shift of it is a prefix of $\text{str}(u)$. With each node, we associate a bit, indicating if it is marked or not. Next, we want to support the following query: given an arbitrary node u , list its marked ancestors. One could easily accomplish this by first finding all of its ancestors (via finding parent nodes iteratively, starting u) and then extracting those that are marked. But the time taken could be $\Theta(m)$ in the worst case. To bound the time in terms of the number of marked ancestors, we store the lowest marked ancestor (LMA) of the following sampled nodes explicitly: $\text{LCA}(\ell_{t \log m}, \ell_{(t+1) \log m})$ for all $t \in [1, m/\log m]$. This scheme requires $O(m)$ extra bits and guarantees that any path towards the root with $\log m$ nodes contains a sampled node. Therefore, to list all the marked ancestors of u , we traverse the path from u to root as before with the difference that when we are at a sampled node, we jump to its LMA (i.e., skip all nodes in between) and continue. All marked ancestors will be visited and the total number of nodes visited (and total time) is bounded by $O(\log m)$ times the number of marked ancestors.

Total space is $m \log \sigma + O(m + d \log n + \sigma \log m) \subseteq n \log \sigma + O(n + d \log n + \sigma \log n)$ bits.

4.2 Query Algorithm

We report all (x, i) , where $x \in [1, |T|]$ and (a *cyclic shift* of) $P_i \in \mathcal{D}$ occurs at position x in T , using the following steps.

1. Find the maximum L_x , such that there exists a node u_x where $T[x..x+L_x)$ is a prefix of $\text{str}(u_x)$ and u_x is the highest such node. Also, let $[sp_x, ep_x]$ be the range of leaves of u_x .
2. Report (x, i) , if a *cyclic shift* of $P_i \in \mathcal{D}$ is a prefix of $\text{str}(u_x)$ and its length is $\leq L_x$.

Note that the first step corresponds to finding the loci of $T[x..]$ in the extended suffix tree, for each x . For traditional suffix tree approach, we find the loci in *ascending* order of x . Here, we do so in the opposite manner, in *descending* order of x . Intuitively, this change allows us to replace the “downward” traversal of tree edges in the extended suffix tree to “upward” traversals, where the latter can be implemented more efficiently with our auxiliary data structures. We now show how to implement these two steps efficiently.

4.2.1 Details of Step 1

Initialize $x = |T|$, $L_{x+1} = 0$, and fix $[sp_{x+1}, ep_{x+1}] = [1, m]$. Then we compute (L_x, u_x, sp_x, ep_x) in the descending order of x using backward search as follows. We have two cases.

1. If $T[x]$ has an occurrence in $\text{eBWT}[sp_{x+1}, ep_{x+1}]$, then find the smallest number a and the largest number b in $[sp_{x+1}, ep_{x+1}]$, such that $\text{eBWT}[a] = \text{eBWT}[b] = T[x]$. Then obtain $sp_x = \text{eLF}[a]$, $ep_x = \text{eLF}[b]$, $u_x = \text{LCA}(\ell_{sp_x}, \ell_{ep_x})$ and $L_x = 1 + L_{x+1}$. The time complexity in this case is $O(\log \log m)$.
2. If $T[x]$ has no occurrence in $\text{eBWT}[sp_{x+1}, ep_{x+1}]$, then we find the lowest ancestor of u_{x+1} , say w , such that $T[x]$ has an occurrence in $\text{eBWT}[z, z']$, where $[z, z']$ is the range of leaves below w . Then find the smallest number a and the largest number b in $[z, z']$, such that $\text{eBWT}[a] = \text{eBWT}[b] = T[x]$. Then obtain $sp_x = \text{eLF}[a]$, $ep_x = \text{eLF}[b]$, $u_x = \text{LCA}(\ell_{sp_x}, \ell_{ep_x})$ and $L_x = \text{strlen}(u_x)$. To find w , we perform a linear search on the path from u_{x+1} to root until we find a node satisfying the required condition. This requires $O(h)$ rank queries, where $h = O(1 + L_{x+1} - L_x)$ is the number of nodes on the path from u_{x+1} to w . Therefore, the time for a particular value of x is $O(h \log \log m + \log m)$. When considering all values of x , we obtain the following total time complexity.³

$$\sum_{x=1}^{|T|} ((L_{x+1} - L_x) \log \log m + \log m) = (L_{|T|+1} - L_1) \log \log m + |T| \log m = O(|T| \log m).$$

In summary, the time for step 1 for all values of x combined is $O(|T| \log m)$.

4.2.2 Details of Step 2

Find $u_x^0 = u_x, u_x^1, u_x^2, \dots, u_x^f$, where u_x^h is the lowest marked ancestor of u_x^{h-1} for all $h \in [1, f]$ and f is the number of marked ancestors of u_x . Let $[sp^h, ep^h]$ be the range of leaves in the subtree of u_x^h . We make the following queries and collect their answers.

- $\text{REPORT}([sp^0, ep^0], L_x)$, which returns the P_i 's corresponding to (x, i) as the final output, where the leaves corresponding to them are below u_x^0 .

³ An alternative way to find w is to perform a binary search along the path from u_{x+1} to the root. This method requires a logarithmic number of rank queries and takes $O(\log m \log \log m)$ time for a particular value of x . However, it results in a higher overall time complexity compared to what we described above.

- For $h \in [1, f]$, we perform the following two queries: $\text{REPORT}([sp^h, sp^{h-1}], \text{strlen}(u_x^h))$ and $\text{REPORT}([ep^{h-1}, ep^h], \text{strlen}(u_x^h))$, which return the P_i 's corresponding to (x, i) as the final output, where the leaves corresponding to them are below u_x^h , but not below u_x^{h-1} (this avoids reporting the same answer multiple times). For a fixed x , the output size is at least f from the definition of marked nodes.

The total time of Step 2 is $O((|T| + \text{occ}) \log m)$, which thereby establishes the overall time complexity. This completes the proof of Theorem 2.

5 Discussion and Open Problems

We present a succinct index for the circular dictionary matching problem that supports queries efficiently in $O((|T| + \text{occ}) \log n)$ time. In our earlier work [48], we achieved a similar result under the assumption that all patterns are approximately the same length (i.e., $\Theta(n/d)$). There, this assumption was necessary for the efficient encoding of $\text{strlen}(\cdot)$ values of nodes in a suffix tree-like structure. Although we later managed to remove this restriction, it came at the cost of a higher query time of $O((|T| \log n + \text{occ}) \log n)$. The improved result in this paper builds on a novel use of the eBWT, combined with a careful adaptation of Sadakane's technique for encoding $\text{strlen}(\cdot)$ values of nodes in the eST, as described in section 3.2.

We conclude with a list of problems that remain open for future research.

1. In contrast to the best known succinct solution for the standard dictionary matching problem [10], which achieves a query time of $O(|T| + \text{occ})$, our solution has a query complexity of $O((|T| + \text{occ}) \log n)$ in the circular setting. Even the recent alternative solution by Cotumaccio [25] has the same query time. This raises an important and natural question: Can the query time be further improved while maintaining the same space bound? Although this does not seem immediate, we remark that a trade-off allowing faster query time might be possible by adapting techniques from [41]; for example, using $O(n \log \sigma)$ bits of space and achieving $O((|T| + \text{occ})(\log_\sigma^\epsilon n + \log \log n))$ query time, where $\epsilon > 0$ is an arbitrarily small constant.
2. Another important question is the efficient construction of our data structure. While the construction of the extended Burrows-Wheeler Transform is already a well-studied problem [8, 13] (also see [43]), the remaining challenge lies in designing the construction of the additional structures. While linear-space construction algorithms that require near-linear time appear achievable, achieving space efficiency (i.e., using small working space) and optimizing the polylogarithmic factors in time can be challenging. Additionally, introducing engineering solutions, possibly through heuristics, could lead to a practical approach, which would benefit from experimental analysis.
3. A *repetition-aware* index for circular dictionary matching would be desirable. While it is relatively straightforward to reduce the space of the extended Burrows-Wheeler Transform by applying run-length compression [14], encoding the remaining components (specifically, the RMQ structures in Section 4.1) in a repetition-aware manner remains challenging. Addressing this may require a careful adaptation of the techniques from [31] or from very recent work in [50].

References

- 1 Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
- 2 Amihood Amir and Martin Farach. Two-Dimensional Dictionary Matching. *Inf. Process. Lett.*, 44(5):233–239, 1992. doi:10.1016/0020-0190(92)90206-B.

- 3 Amihoud Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Kunsoo Park. Dynamic Dictionary Matching. *J. Comput. Syst. Sci.*, 49(2):208–222, 1994. doi:10.1016/S0022-0000(05)80047-9.
- 4 Amihoud Amir, Dmitry Kesselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text Indexing and Dictionary Matching with One Error. *J. Algorithms*, 37(2):309–325, 2000. doi:10.1006/JAGM.2000.1104.
- 5 Amihoud Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the Gap! – Online Dictionary Matching with One Gap. *Algorithmica*, 81(6):2123–2157, 2019. doi:10.1007/S00453-018-0526-2.
- 6 Amihoud Amir, Avivit Levy, Ely Porat, and B. Riva Shalom. Online Recognition of Dictionary with One Gap. *Inf. Comput.*, 275:104633, 2020. doi:10.1016/J.IC.2020.104633.
- 7 Tanver Athar, Carl Barton, Widmer Bland, Jia Gao, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Fast Circular Dictionary-Matching Algorithm. *Math. Struct. Comput. Sci.*, 27(2):143–156, 2017. doi:10.1017/S0960129515000134.
- 8 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing and Indexing the Bijective and Extended Burrows-Wheeler Transform. *Inf. Comput.*, 297:105153, 2024. doi:10.1016/J.IC.2024.105153.
- 9 Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Fast Algorithms for Approximate Circular String Matching. *Algorithms Mol. Biol.*, 9:9, 2014. doi:10.1186/1748-7188-9-9.
- 10 Djamal Belazzougui. Succinct Dictionary Matching with No Slowdown. In Amihoud Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, volume 6129 of *Lecture Notes in Computer Science*, pages 88–100. Springer, 2010. doi:10.1007/978-3-642-13509-5_9.
- 11 Djamal Belazzougui and Gonzalo Navarro. Optimal Lower and Upper Bounds for Representing Sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015. doi:10.1145/2629339.
- 12 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. doi:10.1142/S0129054114400309.
- 13 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the Original eBWT Faster, Simpler, and with Less Memory. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 129–142. Springer, 2021. doi:10.1007/978-3-030-86692-1_11.
- 14 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-Indexing the eBWT. *Inf. Comput.*, 298:105155, 2024. doi:10.1016/J.IC.2024.105155.
- 15 Michael Burrows and David J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. *SRC Research Report*, 124, 1994.
- 16 Bastien Cazaux and Eric Rivals. Linking BWT and XBW via Aho-Corasick Automaton: Applications to Run-Length Encoding. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPICs*, pages 24:1–24:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.CPM.2019.24.
- 17 Davide Cenzato and Zsuzsanna Lipták. A Theoretical and Experimental Analysis of BWT Variants for String Collections. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPICs*, pages 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.CPM.2022.25.
- 18 Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiro Sadakane. Dynamic Dictionary Matching and Compressed Suffix Trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada*,

- January 23–25, 2005, pages 13–22. SIAM, 2005. URL: <http://dl.acm.org/citation.cfm?id=1070432.1070436>.
- 19 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal Dictionary Matching. *Algorithmica*, 83(7):2142–2169, 2021. doi:10.1007/S00453-021-00821-Y.
 - 20 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Circular Pattern Matching with k Mismatches. In Leszek Antoni Gasieniec, Jesper Jansson, and Christos Levkopoulou, editors, *Fundamentals of Computation Theory - 22nd International Symposium, FCT 2019, Copenhagen, Denmark, August 12–14, 2019, Proceedings*, volume 11651 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2019. doi:10.1007/978-3-030-25027-0_15.
 - 21 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Circular Pattern Matching with k Mismatches. *J. Comput. Syst. Sci.*, 115:73–85, 2021. doi:10.1016/J.JCSS.2020.07.003.
 - 22 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P. Pissis, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate Circular Pattern Matching. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022, September 5–9, 2022, Berlin/Potsdam, Germany*, volume 244 of *LIPICs*, pages 35:1–35:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ESA.2022.35.
 - 23 Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate Circular Pattern Matching Under Edit Distance. In Olaf Beyersdorff, Mamadou Moustapha Kanté, Orna Kupferman, and Daniel Lokshtanov, editors, *41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024, March 12–14, 2024, Clermont-Ferrand, France*, volume 289 of *LIPICs*, pages 24:1–24:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICs.STACS.2024.24.
 - 24 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary Matching and Indexing with Errors and Don’t Cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13–16, 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.
 - 25 Nicola Cotumaccio. Improved circular dictionary matching. In Paola Bonizzoni and Veli Mäkinen, editors, *36th Annual Symposium on Combinatorial Pattern Matching, CPM 2025, June 17–19, 2025, Milan, Italy*, volume 331 of *LIPICs*, pages 18:1–18:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICs.CPM.2025.18.
 - 26 Guy Feigenblat, Ely Porat, and Ariel Shiftan. An Improved Query Time for Succinct Dynamic Dictionary Matching. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel A. Pevzner, editors, *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16–18, 2014. Proceedings*, volume 8486 of *Lecture Notes in Computer Science*, pages 120–129. Springer, 2014. doi:10.1007/978-3-319-07566-2_13.
 - 27 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring Labeled Trees for Optimal Succinctness, and Beyond. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23–25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 184–196. IEEE Computer Society, 2005. doi:10.1109/SFCS.2005.69.
 - 28 Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
 - 29 Nathan J Fine and Herbert S Wilf. Uniqueness Theorems for Periodic Functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.
 - 30 Johannes Fischer and Volker Heun. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

- 31 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. doi:10.1145/3375890.
- 32 Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Yilin Yang. A Framework for Designing Space-Efficient Dictionaries for Parameterized and Order-Preserving Matching. *Theor. Comput. Sci.*, 854:52–62, 2021. doi:10.1016/J.TCS.2020.11.036.
- 33 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pBWT: Achieving Succinct Data Structures for Parameterized Pattern Matching and Related Problems. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 397–407. SIAM, 2017. doi:10.1137/1.9781611974782.25.
- 34 Pawel Gawrychowski and Tatiana Starikovskaya. Streaming Dictionary Matching with Mismatches. *Algorithmica*, 84(4):896–916, 2022. doi:10.1007/S00453-021-00876-X.
- 35 Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Dynamic Dictionary Matching in the Online Model. In Zachary Friggstad, Jörg-Rüdiger Sack, and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, volume 11646 of *Lecture Notes in Computer Science*, pages 409–422. Springer, 2019. doi:10.1007/978-3-030-24766-9_30.
- 36 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Towards Optimal Approximate Streaming Pattern Matching by Matching Multiple Patterns in Multiple Streams. In Ioannis Chatzigiannakis, Christos Kaklamanis, Daniel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPICs*, pages 65:1–65:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.ICALP.2018.65.
- 37 Shay Golan and Ely Porat. Real-Time Streaming Multi-Pattern Search for Constant Alphabet. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPICs*, pages 41:1–41:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.ESA.2017.41.
- 38 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/Select Operations on Large Alphabets: A Tool for Text Indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 368–373. ACM Press, 2006. URL: <http://dl.acm.org/citation.cfm?id=1109557>. 1109599.
- 39 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 841–850. ACM/SIAM, 2003. URL: <http://dl.acm.org/citation.cfm?id=644108.644250>.
- 40 Roberto Grossi, Costas S. Iliopoulos, Robert Mercas, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, and Fatima Vayani. Circular Sequence Comparison: Algorithms and Applications. *Algorithms Mol. Biol.*, 11:12, 2016. doi:10.1186/S13015-016-0076-6.
- 41 Roberto Grossi and Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.
- 42 Wing-Kai Hon, Tsung-Han Ku, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, Sharma V. Thankachan, and Jeffrey Scott Vitter. Compressing Dictionary Matching Index via Sparsification Technique. *Algorithmica*, 72(2):515–538, 2015. doi:10.1007/S00453-013-9863-3.
- 43 Wing-Kai Hon, Tsung-Han Ku, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Efficient Algorithm for Circular Burrows-Wheeler Transform. In Juha Kärkkäinen and Jens Stoye, editors, *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*, volume 7354 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 2012. doi:10.1007/978-3-642-31265-6_21.

- 44 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Compressed Dictionary Matching with One Error. In James A. Storer and Michael W. Marcellin, editors, *2011 Data Compression Conference (DCC 2011), 29-31 March 2011, Snowbird, UT, USA*, pages 113–122. IEEE Computer Society, 2011. doi:10.1109/DCC.2011.18.
- 45 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster Compressed Dictionary Matching. *Theor. Comput. Sci.*, 475:113–119, 2013. doi:10.1016/J.TCS.2012.10.050.
- 46 Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Succinct Index for Dynamic Dictionary Matching. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 1034–1043. Springer, 2009. doi:10.1007/978-3-642-10631-6_104.
- 47 Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Sharma V. Thankachan, Hing-Fung Ting, and Yilin Yang. Dictionary Matching with a Bounded Gap in Pattern or in Text. *Algorithmica*, 80(2):698–713, 2018. doi:10.1007/S00453-017-0288-2.
- 48 Wing-Kai Hon, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Succinct Indexes for Circular Patterns. In Takao Asano, Shin-Ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, volume 7074 of *Lecture Notes in Computer Science*, pages 673–682. Springer, 2011. doi:10.1007/978-3-642-25591-5_69.
- 49 Costas S. Iliopoulos, Solon P. Pissis, and M. Sohel Rahman. Searching and Indexing Circular Patterns. In Mourad Elloumi, editor, *Algorithms for Next-Generation Sequencing Data, Techniques, Approaches, and Applications*, pages 77–90. Springer, 2017. doi:10.1007/978-3-319-59826-0_3.
- 50 Dominik Kempa and Tomasz Kociumaka. Collapsing the Hierarchy of Compressed Data Structures: Suffix Arrays in Optimal Compressed Space. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1877–1886. IEEE, 2023. doi:10.1109/FOCS57990.2023.00114.
- 51 Tsvi Kopelowitz, Ely Porat, and Yaron Rozen. Succinct Online Dictionary Matching with Improved Worst-Case Guarantees. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPICs*, pages 6:1–6:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.CPM.2016.6.
- 52 Avivit Levy and B. Riva Shalom. Online Parameterized Dictionary Matching with One Gap. *Theor. Comput. Sci.*, 845:208–229, 2020. doi:10.1016/J.TCS.2020.09.016.
- 53 Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 54 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An Extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.
- 55 Giovanni Manzini. XBWT Tricks. In Shunsuke Inenaga, Kunihiro Sadakane, and Tetsuya Sakai, editors, *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*, volume 9954 of *Lecture Notes in Computer Science*, pages 80–92, 2016. doi:10.1007/978-3-319-46049-9_8.
- 56 J. Ian Munro and Venkatesh Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM J. Comput.*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.
- 57 Gonzalo Navarro and Veli Mäkinen. Compressed Full-Text Indexes. *ACM Comput. Surv.*, 39(1):2, 2007. doi:10.1145/1216370.1216372.
- 58 Shoshana Neuburger and Dina Sokol. Succinct 2D Dictionary Matching. *Algorithmica*, 65(3):662–684, 2013. doi:10.1007/S00453-012-9615-9.
- 59 Enno Ohlebusch, Stefan Stauß, and Uwe Baier. Trickier XBWT Tricks. In Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, editors, *String Processing and*

- Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, volume 11147 of *Lecture Notes in Computer Science*, pages 325–333. Springer, 2018. doi:10.1007/978-3-030-00479-8_26.
- 60 Eric M. Osterkamp and Dominik Köppl. Extending the Parameterized Burrows-Wheeler Transform. In Ali Bilgin, James E. Fowler, Joan Serra-Sagristà, Yan Ye, and James A. Storer, editors, *Data Compression Conference, DCC 2024, Snowbird, UT, USA, March 19-22, 2024*, pages 143–152. IEEE, 2024. doi:10.1109/DCC58796.2024.00022.
 - 61 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct Indexable Dictionaries with Applications to Encoding k -ary Trees, Prefix Sums and Multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.
 - 62 Kunihiro Sadakane. Compressed Suffix Trees with Full Functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.
 - 63 Kunihiro Sadakane and Gonzalo Navarro. Fully-Functional Succinct Trees. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 134–149. SIAM, 2010. doi:10.1137/1.9781611973075.13.
 - 64 Blair L Strang and Nigel D Stow. Circularization of the Herpes Simplex Virus Type 1 Genome upon Lytic Infection. *Journal of Virology*, 79(19):12487–12494, 2005.
 - 65 Kazuya Tsuruta, Dominik Köppl, Shunsuke Kanda, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. c-trie++: A Dynamic Trie tailored for Fast Prefix Searches. *Inf. Comput.*, 285(Part):104794, 2022. doi:10.1016/J.IC.2021.104794.
 - 66 Esko Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.
 - 67 Peter Weiner. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

Wheeler Graphs and Wheeler Languages

Nicola Cotumaccio  

University of Helsinki, Finland

Giovanna D'Agostino  

University of Udine, Italy

Daniel Gibney  

University of Texas at Dallas, TX, USA

Alberto Policriti  

University of Udine, Italy

Nicola Prezza  

DAIS, Ca' Foscari University of Venice, Italy

Sharma V. Thankachan  

North Carolina State University, Raleigh, NC, USA

Abstract

Suffix sorting stands at the core of the most efficient solutions for indexed pattern matching: the suffix tree, the suffix array, compressed indexes based on the Burrows-Wheeler transform, and so on. In [Gagie, Manzini, Sirén, TCS 2017] this concept was extended to labeled graphs, obtaining the rich class of *Wheeler graphs*. This work opened a very fruitful line of research, ultimately generating results able to bridge the fields of compressed data structures, graph theory, and regular language theory. In a Wheeler graph, nodes are sorted according to the alphabetic order of their incoming labels, propagating this order through pairs of equally-labeled edges. This apparently-simple definition makes it possible to solve on Wheeler graphs problems (including, but not limited to: compression, subpath queries, NFA equivalence, determinization, minimization) that on general labeled graphs are extremely hard to solve, and induces a rich structure in the class of regular languages (*Wheeler languages*) recognized by automata whose state transition is a Wheeler graph. The goal of this survey is to provide a summary of (and intuitions behind) the results on Wheeler graphs that appeared in the literature since their introduction, in addition to a discussion of interesting problems that are still open in the field.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Theory of computation → Graph algorithms analysis; Theory of computation → Pattern matching; Theory of computation → Formal languages and automata theory

Keywords and phrases Wheeler languages, Wheeler graphs, pattern matching, indexing, compressed data structures

Digital Object Identifier 10.4230/OASICS.Manzini.2025.12

Funding *Nicola Cotumaccio*: Funded by the Helsinki Institute for Information Technology (HIIT). *Nicola Prezza*: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Sharma V. Thankachan: Partially supported by NSF under grants CCF-2316691 and CCF-2315822.

1 Notation

We refer the reader to [52] for basics on automata theory.



© Nicola Cotumaccio, Giovanna D'Agostino, Daniel Gibney, Alberto Policriti, Nicola Prezza, and Sharma V. Thankachan;

licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 12; pp. 12:1–12:28



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Sequences

A sequence (or string) $\alpha \in \Sigma^*$ is a concatenation of characters from a finite alphabet Σ of size σ . Indices start from 1: $\alpha[i]$, with $i \geq 1$ is the i -th character of the sequence. Notation $\alpha[i : j]$, with $j \geq i$, denotes $\alpha[i]\alpha[i+1]\dots\alpha[j]$.

1.2 Automata and Labeled Graphs

A *finite (nondeterministic) automaton* (NFA for brevity) is a tuple $\mathcal{A} = (Q, s, \Sigma, \delta, F)$ where Q is a finite, non empty set of states, s is the initial state, $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function, and $F \subseteq Q$ is the set of final states. We sometimes omit the alphabet Σ and simply write an NFA as a tuple $\mathcal{A} = (Q, s, \delta, F)$. Sometimes we shall describe the transition function δ using triples (edges), where (q, q', a) stands for $q' \in \delta(q, a)$.

A *semi-automaton* is an automaton that does not specify the initial state nor the set of final states. More formally, a semi-automaton is a triple $\mathcal{A} = (Q, \Sigma, \delta)$, where Q, Σ, δ are defined as above. Equivalently, one can view a semi-automaton as an edge-labeled graph $G = (Q, \Sigma, E)$ where $E = \{(q, q', a) \in Q \times Q \times \Sigma : q' \in \delta(q, a)\}$. When clear from the context, sometimes we will drop the alphabet Σ and simply write (Q, E) to denote a labeled graph. When talking about labeled graphs, we will refer to the elements of Q as *vertices* or *nodes*. Due to their equivalence, in this manuscript we will treat labeled graphs and semi-automata interchangeably. We denote with $|\delta|$ and $|Q|$ the number of transitions and states of a (semi-) automaton $\mathcal{A} = (Q, \Sigma, \delta)$, and with $|\mathcal{A}| = |Q| + |\delta|$ its total size. Similarly, $|G|$ denotes the size of graph $G = (Q, E)$, that is, the number $|Q|$ of its nodes plus the number $|E|$ of its edges.

As customary, we extend δ to operate on strings as follows: for all $q \in Q$, $a \in \Sigma$ and $\alpha \in \Sigma^*$

$$\delta(q, \varepsilon) = \{q\}, \quad \delta(q, \alpha a) = \bigcup_{v \in \delta(q, \alpha)} \delta(v, a).$$

The same definitions hold for semi-automata. Given a finite automaton $\mathcal{A}$, we denote by $\mathcal{L}(\mathcal{A}) = \{\alpha \in \Sigma^* \mid \delta(s, \alpha) \cap F \neq \emptyset\}$ the language accepted (or recognized) by $\mathcal{A}$. We say that two automata are *equivalent* if they accept the same language.

Given a NFA $\mathcal{A} = (Q, s, \Sigma, \delta, F)$ and any of its states $u \in Q$, we denote with symbol I_u the set of all strings labeling paths starting from the source of $\mathcal{A}$ and ending in u : $I_u = \{\alpha \in \Sigma^* : u \in \delta(s, \alpha)\}$.

A finite automaton is *deterministic* (for brevity, DFA) if $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$. In the deterministic case we consider the transition function as a (possibly partial) function $\delta : Q \times \Sigma \rightarrow Q$ (hence, our DFA do not need to be complete). When δ is not defined on a pair (q, a) we write $\delta(q, a) = \perp$.

Unless otherwise stated, we assume that every finite automaton is *trimmed*, that is, every state is reachable from the initial state and can reach at least one final state. This assumption is not restrictive: every automaton can be put in an equivalent trimmed form in linear time.

For trimmed automata the following hold:

- there is at most one state without incoming edges, namely s ;
- every string that can be read starting from s belongs to the set of prefixes, $\text{Pref}(\mathcal{L})$, of the language $\mathcal{L}$.

The languages accepted by automata form the class of *regular languages* and are closed under boolean operations (union, intersection, complementation), concatenation, and the Kleene star.

Let $\mathcal{A} = (Q, s, \delta, F)$ be a NFA and $q \in Q$. The *input language* of q , denoted by I_q , is the set of strings leaving s and entering q , that is, I_q is the language recognized by the automaton $(Q, s, \delta, \{q\})$. We will refer to I_q also as the *set of strings reaching* q . Similarly, the *output language* of q , denoted by O_q is the language recognized by the automaton (Q, q, δ, F) .

Given a regular language, there exists a unique, up to isomorphism, state-wise minimum DFA that recognizes such language. The states of this minimum DFA correspond to the classes of the Myhill-Nerode equivalence relation, defined as follows.

► **Definition 1** (Myhill-Nerode equivalence [48]). *Let $\mathcal{L} \subseteq \Sigma^*$ be a language. Given a string $\alpha \in \Sigma^*$, we define the right context of α as*

$$\alpha^{-1}\mathcal{L} = \{\gamma \in \Sigma^* \mid \alpha\gamma \in \mathcal{L}\},$$

and we denote by $\equiv_{\mathcal{L}}$ the Myhill-Nerode equivalence on $\text{Pref}(\mathcal{L})$ defined as

$$\alpha \equiv_{\mathcal{L}} \beta \iff \alpha^{-1}\mathcal{L} = \beta^{-1}\mathcal{L}.$$

If $\mathcal{L}$ is a regular language we will denote by $\mathcal{D}_{\mathcal{L}} = (Q, s, \delta, F)$ its minimum automaton, having as set of states Q the classes $[\alpha]_{\mathcal{L}}$ of the Myhill-Nerode equivalence, $s = [\varepsilon]_{\mathcal{L}}$, $\delta([\alpha]_{\mathcal{L}}, a) = [\alpha \cdot a]_{\mathcal{L}}$, and $F = \{[\alpha]_{\mathcal{L}}^c \mid \alpha \in \mathcal{L}\}$.

It is customary to define a version of the Myhill-Nerode equivalence relation on the states of a DFA by putting into the same class states u, v whose strings in I_u, I_v are Myhill-Nerode equivalent. More formally, one can define the following relation:

► **Definition 2** (Myhill-Nerode equivalence on DFA). *Let $\mathcal{D} = (Q, s, \delta, F)$ be a DFA. We denote by $\equiv_{\mathcal{D}}$ the Myhill-Nerode equivalence on Q defined as*

$$u \equiv_{\mathcal{D}} v \iff (\forall \alpha \in \Sigma^*)(\delta(u, \alpha) \in F \iff \delta(v, \alpha) \in F).$$

The minimum DFA for a regular language $\mathcal{L}$ can be equivalently obtained by collapsing $\equiv_{\mathcal{D}}$ -equivalent states of any DFA $\mathcal{D}$ recognizing $\mathcal{L}$. Hopcroft's algorithm [41] achieves this goal efficiently by computing $\equiv_{\mathcal{D}}$ in $O(|\mathcal{D}| \log |\mathcal{D}|)$ time.

1.3 Orders

We assume that there is a fixed total order $\preceq$ on the alphabet Σ . We extend $\preceq$ to strings in Σ^* *co-lexicographically*, that is, for $\alpha, \beta \in \Sigma^*$, we have $\alpha \preceq \beta$ if and only if either α is a suffix of β (denoted by $\alpha \dashv \beta$), or there exist $\alpha', \beta', \gamma \in \Sigma^*$ and $a, b \in \Sigma$, such that $\alpha = \alpha'a\gamma$ and $\beta = \beta'b\gamma$ and $a \prec b$.

If $(Z, \leq)$ is a partial order, we denote by $(Z, <)$ its corresponding strict partial order. Given a partial order $(Z, \leq)$ we say that a subset $I \subseteq Z$ is *convex* if, for any $x, y, z \in Z$ with $x < y < z$, if $x, z \in I$ then $y \in I$.

2 Wheeler Graphs

Gagie, Manzini, and Sirén in [35] showed a natural way to extend the co-lexicographic order of strings to labeled graphs. The idea is simple: let us focus on the class of labeled graphs for which there exists a total order of the nodes that is propagated along pairs of equally-labeled transitions, further requiring that pairs of nodes whose incoming labels differ are sorted consistently with the underlying total order of Σ . Due to the fact (as shown next) that these graphs generalize the celebrated *Burrows-Wheeler transform* [16] (*BWT* for short), the authors of [35] decided to call such a class *Wheeler graphs*. More formally:

► **Definition 3** (Wheeler Graph). A Wheeler graph is a labeled graph $G = (Q, \Sigma, E)$ endowed with a binary relation $<$ such that: $(Q, <)$ is a linear order where all nodes with in-degree equal to zero are smaller than nodes with in-degree strictly larger than zero, and for which the following two Wheeler properties (sometimes also called Wheeler axioms) are satisfied. Let $(u, u', a), (v, v', b) \in E$:

(W1) $a < b \rightarrow u' < v'$

(W2) $(a = b \wedge u < v \wedge u' \neq v') \rightarrow u' < v'$.

In this manuscript it will be useful to adapt the above definition to finite automata, in order to study the class of regular languages recognized by automata whose transition relation is a Wheeler graph.

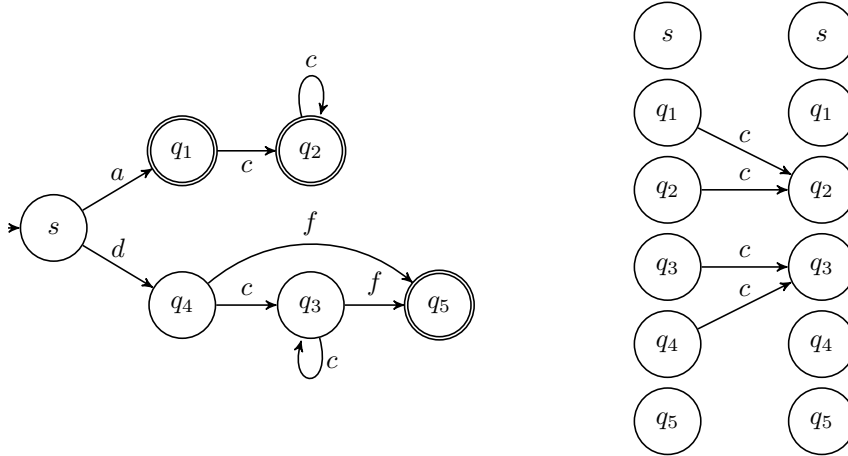
► **Definition 4** (Wheeler Automaton). A Wheeler automaton $\mathcal{A}$ (WNFA for brevity) is a NFA (Q, s, δ, F) endowed with a binary relation $<$ such that: $(Q, <)$ is a linear order having the initial state s as minimum, s has no in-going edges, and the following two Wheeler properties (or Wheeler axioms) are satisfied. Let $p' \in \delta(p, a)$ and $q' \in \delta(q, b)$:

(W1) $a < b \rightarrow p' < q'$

(W2) $(a = b \wedge p < q \wedge p' \neq q') \rightarrow p' < q'$.

A Wheeler DFA (WDFA) is a WNFA in which $|\delta(q, a)| \leq 1$, for all $q \in Q$ and $a \in \Sigma$.

An example of a WDFA is presented in Figure 1 (left). In the same figure (right) we show a very insightful representation of Wheeler automata, called the *bipartite representation*: we duplicate states in two columns following any Wheeler order of the automaton, and we draw edges from left to right (in Figure 1, for clarity only edges labeled with c are shown). In this representation, Wheeler Axiom W1 translates to the fact that nodes are sorted by increasing incoming letter, and Wheeler Axiom W2 to the fact that no same-letter edges cross. Most properties, algorithms, and data structures on Wheeler graphs can be easily understood by keeping in mind this representation.



■ **Figure 1** Left: A WDFA $\mathcal{D}$ recognizing the language $\mathcal{L}_d = ac^* \cup dc^*f$. The only order that makes $\mathcal{D}$ Wheeler is $s < q_1 < q_2 < q_3 < q_4 < q_5$. Right: Bipartite representation of the Wheeler automaton $\mathcal{D}$, showing only edges labeled with c for clarity. In this representation, Axiom W2 translates to the fact that no same-letter edges cross.

Of particular interest for this survey is the class of Wheeler languages:

► **Definition 5** ([3]). *A regular language is said to be Wheeler if there exists a WNFA recognizing it.*

As a matter of fact, nondeterminism is not important in Definition 5. As proved in [3] (we will come back to this in Section 4), WDFA and WNFA have the same expressive power: both such classes of finite automata recognize precisely the class of Wheeler languages.

The following considerations hold both for Wheeler graph and Wheeler automata, so we state them only for the latter class of combinatorial objects.

First, we observe that the automata for which there exists a total order satisfying Axiom W2 correspond to the class of totally-sortable *Ordered Automata*, studied by Shyr and Thierrin in 1974 [53]. In that article, the authors showed that the languages recognized by such automata are star-free; in particular, it follows that all Wheeler languages are star-free. As discussed in this manuscript, by imposing the additional constraint on nodes with in-degree equal to zero and the additional Axiom W1, the work [35] unveiled a restricted class of Ordered Automata (the Wheeler automata) possessing a large number of extremely interesting properties.

Also observe that a consequence of Wheeler Axiom (W1) is that $\mathcal{A}$ is *input-consistent*, that is all transitions entering a given state $q' \in Q$ must have the same label: if $q' \in \delta(q, a)$ and $q' \in \delta(p, b)$, then $a = b$. Therefore a function $\lambda : Q \rightarrow \Sigma$ that labels each state with the unique label of its incoming edges, can be introduced. For the initial state s , the only one without incoming edges, we set $\lambda(s) = \# \notin \Sigma$, stipulating that $\# \prec a$, for all $a \in \Sigma$. Observe that this simplifies the definition of Wheeler automata in that, by replacing a and b with $\lambda(p')$ and $\lambda(q')$ in Definition 4, we no longer need to require that states with in-degree equal to zero are smaller than states with in-degree strictly larger than zero. We decided to adopt the formulation of Definitions 3 and 4 since it is closer to the one originally defined in [35].

For a fixed-sized (constant) alphabet, requiring an automaton to be *input-consistent* is not computationally demanding. In fact, given an NFA $\mathcal{A} = (Q, s, \delta, F)$ we can build an equivalent, input-consistent one just by creating, for each state $q \in Q$, at most $|\Sigma|$ copies of q , that is, one for each different incoming label of q . This operation can be performed in $O(|Q| \cdot |\Sigma|)$ time.

3 Compact Data Structures on Wheeler Graphs

Wheeler graphs extend the suffix array [45], the Burrows-Wheeler Transform [16] and Ferragina and Manzini's FM-index [34] from strings to graphs. Following [35], we now show how to encode Wheeler graphs using a small number of bits. Consider the automaton in Figure 1. We store the following strings:

- The string **LAB** = *adccfcf* is obtained by concatenating the labels of all edges leaving s , the labels of all edges leaving q_1 , the labels of all edges leaving q_2 , and so on. If a node has multiple outgoing edges, the corresponding labels are sorted in alphabetic order.
- The string **OUT** = 10010101001001 is obtained by storing in unary the outdegree of s , the outdegree of q_1 , the outdegree of q_2 , and so on. The outdegrees are 2, 1, 1, 2, 2, 0.
- The string **IN** = 11010010010100 is obtained by storing in unary the indegree of s , the indegree of q_1 , the indegree of q_2 , and so on. The indegrees are 0, 1, 2, 2, 1, 2.
- The bit array **FIN** = 011001 remembers which states are final: s is not final, q_1 is final, q_2 is final, and so on.

Let $e = |\delta|$ be the number of transitions in the NFA, $n = |Q|$ be the number of nodes and $\sigma = |\Sigma|$ be the size of the alphabet. We can store **LAB** in $e \log \sigma$ bits, **OUT** in $e + n$ bits, **IN** in $e + n$ bits and **FIN** in n bits. It turns out that these sequences provide a compact encoding

of the Wheeler graphs that extends the Burrows-Wheeler transform from strings to Wheeler automata. Let us show how we can use these sequences to retrieve the automaton. By using LAB and OUT, we infer that s has two outgoing edges labeled a and d , q_1 has an outgoing edge labeled c , q_2 has an outgoing edge labeled c , q_3 has two outgoing edges labeled c and f , q_4 has two outgoing edges labeled c and f , and q_5 has no outgoing edges. Now we sort the character in LAB, obtaining $acccdf f$. From Wheeler property (W1) and IN we infer that s has no incoming edges, q_1 has an incoming edge labeled a , q_2 has two incoming edges both labeled c , q_3 has two incoming edges both labeled c , q_4 has an incoming edge labeled d , and q_5 has two incoming edges labeled f . By using the bipartite representation of Figure 1, we can retrieve the whole set of edges. For example, consider character c . We know that there are outgoing edges labeled c from q_1 , q_2 , q_3 , q_4 , and we know that there are incoming edges labeled c from q_2 (two edges) and q_3 (two edges), so since same-letter edges cannot cross we conclude that the edges labeled c are (q_1, q_2, c) , (q_2, q_2, c) , (q_3, q_3, c) , (q_4, q_3, c) . Lastly, we retrieve the set of all final states by using FIN.

Not only Wheeler graphs can be encoded in a small number of bits as described above, but they also support efficient pattern matching queries on the automaton's paths (equivalently, on the language recognized by the automaton). Consider the following pattern matching problems (where we assume, without loss of generality, that every state can reach a final state):

1. Given $\alpha \in \Sigma^*$, decide whether α is accepted by the automaton $\mathcal{A}$ (that is, decide, whether $\alpha \in \mathcal{L}(\mathcal{A})$).
2. Given $\alpha \in \Sigma^*$, decide whether α occurs on the paths of the automaton $\mathcal{A}$ (that is, decide whether there exist $\gamma_1, \gamma_2 \in \Sigma^*$ such that $\gamma_1 \alpha \gamma_2 \in \mathcal{L}(\mathcal{A})$).

For example, in Figure 1, the string $\alpha = cccf$ is not accepted by the automaton, but it occurs in the automaton (there is a path from q_4 to q_5 labeled with α).

The compressed representation of a Wheeler graph allow solving both problems efficiently. First, as observed in [35], from the bipartite representation of a graph we can deduce *path-coherence*: if we start from an interval of consecutive nodes in Wheeler order, we consider a character c , and we follow all edges labeled with c from those nodes, we end up in another (possibly empty) *interval* of consecutive nodes. For example, if we start from $\{q_1, q_2, q_3, q_4\}$ and we follow all edges labeled c , we end up in $\{q_2, q_3\}$.

This property implies that we can solve both pattern matching queries above listed in a linear number of steps. For the first problem, we start from the interval of states $\{s\}$, and for the second problem we start from the whole set of states Q . To update an interval by following the edges labeled with c , we augment the compressed representation of a Wheeler automaton with compressed data structures that support efficient rank/select queries, thus extending the FM-index from strings to Wheeler automata. This leads to the following result:

► **Theorem 6** (adapted from [35]). *Let $\mathcal{A}$ be a Wheeler NFA with e edges on an alphabet of size σ . Then, $\mathcal{A}$ can be encoded by using a data structure of $e \log \sigma (1 + o(1)) + O(e)$ bits that supports pattern matching queries in $O(m \log \log \sigma)$ time, where m is the length of the string to be matched.*

Wheeler graphs are therefore a very convenient class of graphs, because they support pattern matching in linear time (for constant alphabets) within compressed space, while on arbitrary graphs pattern matching cannot be solved in subquadratic time (assuming that the orthogonal vector hypothesis is true) [32]. At the same time, the class of Wheeler graphs subsumes all previous extensions of the Burrows-Wheeler transform and the FM-index, including labeled trees [33], circular strings [46, 40] and de Bruijn graphs [14]. Wheeler

graphs also inherit compressibility properties of the Burrows-Wheeler transform. For example, *tunneling*, a method for identifying blocks that capture some redundancy of the BWT [8], can be extended to Wheeler graphs [4].

Some more complicated pattern matching problems require more advanced data structures. A typical example is the problem of computing *matching statistics*: given an automaton $\mathcal{A}$, and a string α , determine for every $1 \leq i \leq |\alpha|$ the longest suffix α_i of $\alpha[1, i]$ that occurs in $\mathcal{A}$. For example, in Figure 1, if $\alpha = accfdf$, we have $\alpha_1 = a$, $\alpha_2 = ac$, $\alpha_3 = acc$, $\alpha_4 = cc$, $\alpha_5 = d$, $\alpha_6 = df$. On strings, this problem can be solved by exploiting the FM-index and the *longest common prefix array* [49]. In [19], the notion of longest common prefix array was extended to Wheeler DFA. The idea is to consider the smallest and the largest string reaching each state. For example, in Figure 1, the smallest string reaching q_2 is ac and the largest string reaching q_2 is $\dots ccccc$ (these strings can be read by starting from q_2 , following edges in a backward fashion until either reaching the source s or a loop, and finally reversing the obtained string). More generally, we proceed as follows. Let $\mathcal{A}$ be a Wheeler DFA with n nodes, in which the i -th state in the Wheeler order is q_i . We define $2n$ strings: γ_1 is the smallest string reaching q_1 , γ_2 is the largest string reaching q_1 , γ_3 is the smallest string reaching q_2 , γ_4 is the largest string reaching q_2 , and so on. We define the array $\text{LCS}[2, 2n]$ such that $\text{LCS}[i]$ is the length of the longest common suffix between γ_{i-1} and γ_i . Then, it can be shown that, for every $2 \leq i \leq 2n$, either $\text{LCS}[i]$ is infinite or its value is less than $2n$ [19, 5]. This implies that LCS can be stored using at most $O(n \log n)$ bits.

The LCS array of a Wheeler DFA supports computing matching statistics efficiently:

► **Theorem 7** ([19]). *Let $\mathcal{A}$ be a Wheeler DFA with n states. By storing the LCS array of $\mathcal{A}$ using $O(n \log n)$ bits, we can compute matching statistics in $O(m \log n)$ time, where m is the length of the string to be matched.*

Storing a Wheeler DFA requires only $e \log \sigma(1 + o(1)) + O(e)$ bits (Theorem 6), so storing the LCS array using $O(n \log n)$ bits may require significantly more space than storing the automaton itself, if the alphabet and the numbers of edges are small. Similarly to the string case, it is possible to design a sampling mechanism for the LCS array that allows better space-time trade-offs. The key idea is to store a range minimum query data structure on the LCS array (which only requires $O(n)$ bits) and sample $O(n/\log n)$ entries of the LCS array in such a way that, by solving range minimum queries, it is possible to retrieve each entry of the LCS array in $O(\log n)$ steps. This leads to the following result.

► **Theorem 8** ([26]). *Let $\mathcal{A}$ be a Wheeler DFA with n states. By storing a data structure of $O(n \log \log \sigma)$ bits, we can compute matching statistics in $O(m \log^2 n)$ time, where m is the length of the string to be matched.*

4 Wheeler Languages

Given the interesting properties of Wheeler automata discussed in the previous section, a natural language-theoretic question is: which languages are recognized by Wheeler DFA (WDFA for brevity) and Wheeler NFA (WNFA for brevity)? In this section we consider the class of *Wheeler Languages*, that is the class of languages recognized by a WNFA or, equivalently as we shall see in Theorem 10, by a WDFA.

A first observation is that the Wheeler *local* conditions of Definition 4 are equivalent, on DFA, to a more global condition expressed in terms of the sets I_q of words reaching a state q . If $\mathcal{D}$ is a DFA and $\prec$ is a (strict) order of the alphabet, we can define a (strict) partial

order $<_{\mathcal{D}}$ on its states by stipulating, for $q \neq q'$, that

$$q <_{\mathcal{D}} q' \Leftrightarrow \forall \alpha \in I_q \forall \beta \in I_{q'} (\alpha \prec \beta)$$

The (in general) partial order $<_{\mathcal{D}}$ indicates whether $\mathcal{D}$ is Wheeler in the following sense:

► **Lemma 9** ([25]). *Let $\mathcal{D}$ be an input-consistent DFA in which the initial state s has no in-going edges. Then*

$$\mathcal{D} \text{ is a WDFA if and only if the order } <_{\mathcal{D}} \text{ is total.} \quad (1)$$

Using the previous lemma we easily see that we can enlarge the class of WDFA to non input-consistent automata, without changing the class of recognized languages, by defining a WDFA as a DFA in which the initial state s has no in-going edges and the order $\mathcal{D}$ is total. In the following we will freely use the new enlarged class of WDFA.

Notice that if $<_{\mathcal{D}}$ is total, then we can decide whether $q <_{\mathcal{D}} q'$ by simply checking the relative co-lexicographical order of any two strings $\alpha \in I_q$ and $\beta \in I_{q'}$. An easy consequence of this is that, if $\mathcal{D}$ is a WDFA and $(\alpha_n)_{n \in \omega}$ is a co-lexicographically (strictly) monotone sequence of words in $\text{Pref}(\mathcal{L}(\mathcal{D}))$, then for some state q of $\mathcal{D}$, $(\alpha_n)_{n \in \omega}$ will eventually belong to I_q . This fact helps us to locate Wheeler languages among sub-regular languages by testing it against a very well-known class: the star-free languages. Languages in this class can be described by regular expressions constructed from the letters of the alphabet, the empty word, the empty set symbol, all boolean operators – including complementation – and concatenation but no Kleene star. We can indeed prove that a Wheeler language is always star-free. To see this we recall that a characterization of the star-free languages says that a language $\mathcal{L}$ is star-free if and only if there exists n such that for all $\alpha, \beta, \gamma \in \Sigma^*$ it holds:

$$\alpha\beta^n\gamma \in \mathcal{L} \Rightarrow \forall m \geq n \quad \alpha\beta^m\gamma \in \mathcal{L}. \quad (2)$$

Suppose now that a language $\mathcal{L}$ is Wheeler, that is, there exists a WDFA $\mathcal{D}$ recognizing $\mathcal{L}$. Consider a sequence of words of the form $(\alpha\beta^k)_{k \in \omega}$: this sequence is always monotone (increasing or decreasing depending on whether $\alpha \preceq \alpha\beta$ or $\alpha \succeq \alpha\beta$). Hence, by the Wheelerness of $\mathcal{D}$, there must exist a $\mathcal{D}$ -state q and $n \in \mathbb{N}$ such that, for all $m \geq n$, $\alpha\beta^m$ belongs to I_q . Condition (2) easily follows.

As a natural next step we present below a characterization of the class of Wheeler languages, both from the language as well as from the accepting-automata view points.

The former can be given by making an appeal to the celebrated Myhill-Nerode equivalence $\equiv_{\mathcal{L}}$. To this end we say that an equivalence relation $\equiv$ over an ordered set $(A, \leq)$ is *convex* if whenever $a \leq b \leq c$ and $a \equiv c$, we also have $b \equiv a$. Consider the following “convex refinement” $\equiv_{\mathcal{L}}^c$ of $\equiv_{\mathcal{L}}$. For all $\alpha, \beta \in \text{Pref}(\mathcal{L})$ we say that $\alpha \equiv_{\mathcal{L}}^c \beta$ if and only if

$$\text{end}(\alpha) = \text{end}(\beta) \wedge (\forall \gamma \in \text{Pref}(\mathcal{L})) (\min\{\alpha, \beta\} \preceq \gamma \preceq \max\{\alpha, \beta\} \rightarrow \gamma \equiv_{\mathcal{L}} \alpha),$$

where $\text{end}(\alpha)$ is the final character of α when $\alpha \neq \epsilon$, and ϵ otherwise.

The following theorem shows that the convex equivalence $\equiv_{\mathcal{L}}^c$ plays exactly the role of the Myhill-Nerode equivalence when referred to Wheeler Languages.

► **Theorem 10** (Myhill-Nerode for Wheeler Languages [2, 3]). *Given a language $\mathcal{L} \subseteq \Sigma^*$, the following are equivalent:*

1. $\mathcal{L}$ is a Wheeler language (i.e. $\mathcal{L}$ is recognized by a WNFA).
2. $\equiv_{\mathcal{L}}^c$ has finite index.

3. $\mathcal{L}$ is a union of classes of a convex, input-consistent, right invariant, finite index equivalence relation over $(\text{Pref}(\mathcal{L}), \prec)$.
4. $\mathcal{L}$ is recognized by a WDFA.

As a consequence of Theorem 10 we get an elegant further characterization of Wheeler languages by considering, again, monotone sequences. The condition expresses the fact that they turn out being eventually trapped in a single state, this time of the minimum DFA for the language:

► **Corollary 11** ([2, 3]). *A regular language $\mathcal{L}$ is Wheeler if and only if all monotone sequences in $(\text{Pref}(\mathcal{L}), \preceq)$ become eventually constant modulo $\equiv_{\mathcal{L}}$. In other words, for all sequences $(\alpha_i)_{i \in \omega}$ in $\text{Pref}(\mathcal{L})$ such that:*

$$\alpha_1 \preceq \alpha_2 \preceq \dots \preceq \alpha_i \preceq \dots \quad \text{or} \quad \alpha_1 \succeq \alpha_2 \succeq \dots \succeq \alpha_i \succeq \dots,$$

there exists an n such that $\alpha_h \equiv_{\mathcal{L}} \alpha_k$, for all $h, k \geq n$.

Proof Sketch: to see that the condition is sufficient we use Theorem 10 and suppose, by contraposition, $\equiv_{\mathcal{L}}^c$ has an infinite number of classes. Then, there is a single $\equiv_{\mathcal{L}}$ -class which is partitioned into an infinite number of $\equiv_{\mathcal{L}}^c$ -classes and we can single out a monotone sequence $(\beta_i)_{i \in \omega}$ (say, increasing) whose elements are pairwise non $\equiv_{\mathcal{L}}^c$ -equivalent words. By discarding at most a finite number of them, we may suppose that all β_i are not only $\equiv_{\mathcal{L}}$ -equivalent, but also end with the same letter. For any $i \in \omega$, since $\beta_i \not\equiv_{\mathcal{L}}^c \beta_{i+1}$, by definition of $\equiv_{\mathcal{L}}^c$, there must exist η_i with $\beta_i \prec \eta_i \prec \beta_{i+1}$ and $\eta_i \not\equiv_{\mathcal{L}} \beta_i$. But then the monotone sequence

$$\beta_1 \prec \eta_1 \prec \beta_2 \prec \eta_2 \prec \dots$$

is never trapped in a single $\equiv_{\mathcal{L}}$ class.

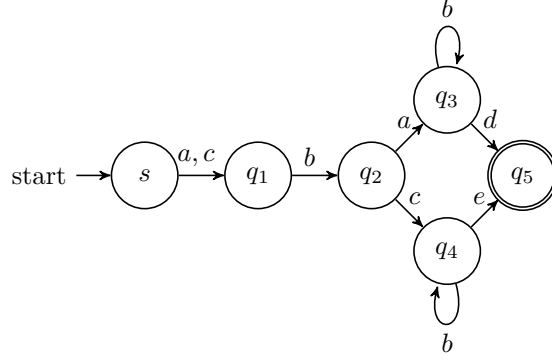
The condition is also necessary, since a monotone sequence gets trapped in any WDFA recognizing $\mathcal{L}$, it gets a fortiori trapped in the minimum DFA $\mathcal{D}_{\mathcal{L}}$ recognizing the language (even if $\mathcal{D}_{\mathcal{L}}$ is not Wheeler).

Using Corollary 11 one can show that Wheeler languages are closed under a few classical operations while other (even very simple) compositions of Wheeler languages do not maintain Wheelerness.

► **Lemma 12** ([3]). *The following hold:*

1. *Finite and co-finite languages are Wheeler.*
2. *The intersection of two Wheeler languages is Wheeler.*
3. *Wheeler languages are not closed under union, concatenation, Kleene star.*
4. *The union of a Wheeler language with a finite set is Wheeler.*
5. *The right-concatenation of a Wheeler language with a finite set is Wheeler.*

We now consider the problem of characterizing Wheeler languages using automata. For star-free languages we know that a combinatorial property of the minimum DFA – i.e. being *counter-free* – characterizes membership in that class and we can give a similar result also for the Wheeler class. However, let us first notice that being Wheeler for a language $\mathcal{L}$ does not necessarily imply Wheelerness for its minimum DFA $\mathcal{D}_{\mathcal{L}}$. Consider the DFA $\mathcal{D}$ depicted in Figure 2. If $\preceq$ is the alphabetical order on Σ , since $a \prec ab \prec ac$ and since $a, ac \in I_{q_1}$, $ab \in I_{q_2}$, it follows that states q_1, q_2 are $<_{\mathcal{D}}$ -incomparable. Hence $\mathcal{D}$ is not Wheeler, even though $\mathcal{L}(\mathcal{D})$ is Wheeler. To see this just consider the DFA $\mathcal{D}'$ obtained from $\mathcal{D}$ by duplicating state q_1 into $q_{1,a}$ and $q_{1,c}$ and by defining $\delta(s, a) = q_{1,a}$, $\delta(s, c) = q_{1,c}$, and $\delta(q_{1,a}, b) = \delta(q_{1,c}, b) = q_2$. In $\mathcal{D}'$ the order $<_{\mathcal{D}'}$ is total so that $\mathcal{D}'$ is a WDFA recognizing $\mathcal{L}(\mathcal{D})$.



■ **Figure 2** The depicted DFA $\mathcal{D}$ is the minimum DFA accepting the Wheeler language $\mathcal{L}(\mathcal{D})$. However, $\mathcal{D}$ is not Wheeler since states q_1, q_2 are incomparable with respect to $<_{\mathcal{D}}$.

Hence, in order to decide whether a language $\mathcal{L}$ is Wheeler we cannot simply rely on the Wheelerness of its minimum DFA $D_{\mathcal{L}}$. Yet, a simple combinatorial property of $<_{D_{\mathcal{L}}}$ characterizes Wheelerness. Even though there are Wheeler languages in which $<_{D_{\mathcal{L}}}$ is not total, in such cases $<_{D_{\mathcal{L}}}$ -incomparability must be limited to special kind of pairs. To illustrate this consider again the minimum DFA in Figure 2, this time with respect to the following order on the alphabet: $a \prec c \prec b \prec d \prec e$. The following theorem tells us that the language recognized is not Wheeler, because the two nodes q_3, q_4 are not only $<_{\mathcal{D}}$ -incomparable but are also infinitely *entangled*, since they are the starting point of two equally labeled cycles – this was not so using the alphabetical order since $q_3 <_{\mathcal{D}} q_4$ in that case.

Intuitively speaking, $\mathcal{L}$ is *not* Wheeler if and only if there is no *finite* splitting of the states of $\mathcal{D}_{\mathcal{L}}$ that can possibly make $\mathcal{D}_{\mathcal{L}}$ Wheeler.

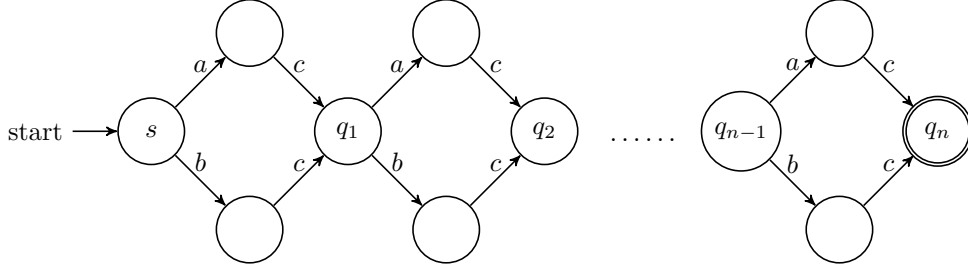
► **Theorem 13** ([3, 11]). *Let $\mathcal{L}$ be a regular language and let $D_{\mathcal{L}}$ be the minimum trimmed DFA accepting $\mathcal{L}$. Then, $\mathcal{L}$ is not Wheeler if and only if $D_{\mathcal{L}}$ contains two $<_{D_{\mathcal{L}}}$ -incomparable nodes $u \neq v$ and two equally labeled cycles, one starting from u and the other from v .*

As an easy corollary of the previous theorem we see at once that if the minimum trimmed DFA $D_{\mathcal{L}}$ accepting a regular language $\mathcal{L}$ contains three equally labeled cycles starting from three different states q_1, q_2, q_3 , then $\mathcal{L}$ cannot be Wheeler. Indeed, suppose w.l.o.g. that the cycles are labeled by the word γ and $\alpha_1 \prec \alpha_2 \prec \alpha_3$ are non empty words, different from γ , reaching q_1, q_2, q_3 , respectively. Then two out of the three among $\alpha_1, \alpha_2, \alpha_3$ must lie on the same side of γ , say $\alpha_1 \prec \alpha_2 \prec \gamma$. Then $\alpha_1\gamma \prec \alpha_2\gamma \prec \alpha_1\gamma^2$ with $\alpha_1\gamma, \alpha_1\gamma^2 \in I_{q_1}$ and $\alpha_2\gamma \in I_{q_2}$, showing that q_1, q_2 are $<_{D_{\mathcal{L}}}$ -incomparable and $\mathcal{L}$ is not Wheeler by Theorem 13.

A similar combinatorial condition on the minimum DFA will be used in the following to find efficient algorithms for analyzing the “level of Wheelerness” of regular languages.

Finally, we consider the problem of constructing the minimum WDFA for a Wheeler language. Consider, for any $n \in \mathbb{N}$, the language whose minimum (input-consistent) DFA D_n is depicted in Figure 3. This language is finite, hence it is Wheeler – in fact, for *any* order of the alphabet. However, if we consider the standard alphabetical order, D_n is not Wheeler, because states q_1, q_2 cannot be ordered by $<_{D_n}$ since $acac \prec bc \prec bcbc$ with $acac, bcbc \in I_{q_2}$, $bc \in I_{q_1}$. However, Theorem 10 says that every Wheeler language has a unique state-minimum WDFA, whose set of states is the set of equivalence classes of $\equiv_{\mathcal{L}}^c$, while initial and final states are defined as in the regular case. For the language in Figure 3 it can be proved that the state-minimum WDFA has a number of states which is exponential

in the number n of states in the minimum (input-consistent) DFA D_n (see [47] for further consideration on the relative size of minimum DFA and WDFA for the same language). We will discuss algorithmic solutions to the problem of computing such minimum WDFA in Section 7.4.



■ **Figure 3** The depicted DFA is accepting a Wheeler (finite) language but the minimum Wheeler DFA accepting the same language has size exponential in n .

4.1 The *rational* embedding

A different “reading” of the effect of co-lexicographic ordering on the collection of strings accepted by a given Wheeler automaton can be given mapping strings on rational numbers in between zero and one (see [47]). To this end it is sufficient to use characters as digits whose positional value is decreasing right-to-left. The connection that this view will provide is based on the fact that the co-lex ordering of strings turns out to be the (familiar) ordering of the rationals onto which they map.

Consider the following definition:

► **Definition 14 (The Rational Embedding of Σ^*).** *The Rational Embedding of Σ^* is the map $\mathfrak{q} : \Sigma^* \rightarrow \mathbb{Q}[0, 1)$ defined as follows. If $\sigma = |\Sigma|$, for any $\alpha = \alpha_1 \dots \alpha_m \in \Sigma^*$:*

$$\mathfrak{q}(\alpha) = \sum_{i=1}^m \alpha_i \cdot (\sigma + 2)^{-(m-i+1)}.$$

The function $\mathfrak{q}$ maps characters in digits and strings in rational numbers in $[0, 1)$ written in base $(|\Sigma| + 2)$, without using the smallest and the largest digit.

The mapping has the following elegant property:

$$\alpha \preceq \beta \text{ if and only if } \mathfrak{q}(\alpha) \leq \mathfrak{q}(\beta).$$

Letting $I_{\mathbb{Q}[0,1)}$ be the collection of non-empty convex sets of rationals in $\mathbb{Q}[0, 1)$:

$$I_{\mathbb{Q}[0,1)} = \{J \subseteq \mathbb{Q}[0, 1) \mid J \neq \emptyset \wedge (\forall a, c \in J)(\forall b \in \mathbb{Q})(a \leq b \leq c \Rightarrow b \in J)\},$$

we can map any collection of strings reaching a given state into a convex subset of rationals in $[0, 1)$.

► **Definition 15 (The Rational Embedding of an automaton).** *The Rational Embedding of $\mathcal{A} = (Q, s, \delta, F)$ is the map $I^{\mathfrak{q}} : Q \rightarrow I_{\mathbb{Q}[0,1)}$ defined as follows: for any $q \in Q$,*

$$I^{\mathfrak{q}}(q) = \bigcap \{J \in I_{\mathbb{Q}[0,1)} \mid (\forall \alpha \in I_q)(\mathfrak{q}(\alpha) \in J)\}.$$

In other words, $I^{\mathfrak{q}}(q)$ is the convex closure (convex hull) of $\{\mathfrak{q}(\alpha) \mid \alpha \in I_q\}$.

12:12 Wheeler Graphs and Wheeler Languages

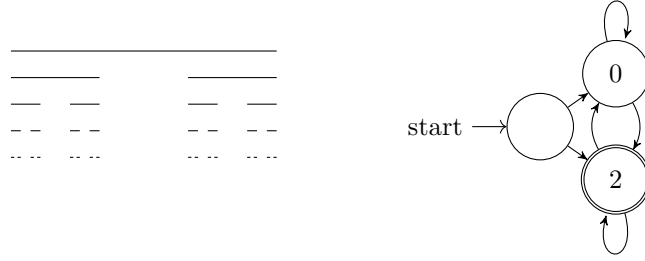
Denoting $I^q(q)$ by I_q^q we can restate Wheelerness as follows: for any pair of states $q_1, q_2 \in Q$, either the sup (right limit) of $I_{q_1}^q$ is smaller than the inf (left limit) of $I_{q_2}^q$, or vice versa.

Moreover, many further questions relating numbers and accepted strings arise. For example, are the inf and sup of I_q^q s always rational numbers? How many accumulation points can we observe by analyzing the embeddings of strings belonging to a regular language? Are accumulation points useful in recognizing Wheelerness?

The first of the above question has been tackled in [47], where the following has been proved:

► **Theorem 16.** *If $\mathcal{L} = \mathcal{L}(\mathcal{D})$, with $\mathcal{L}$ Wheeler and $\mathcal{D}$ either minimum or Wheeler, then for all $q \in Q$, we have that the inf and the sup of I_q^q are rationals.*

The above result was essentially already present in [38] where, however, no co-lexicographic ordering of strings was (explicitly) taken into account and where a real number was considered *defined* by an automaton when all its approximations are accepted by the automaton. Among other things, Hartmanis and Stearns observe that the measure zero Cantor set¹ can be defined by an automaton (see Figure 4).



■ **Figure 4** Cantor set and the automaton accepting it. Notice that the automaton on the right is trimmed: the underlying alphabet is $\Sigma = \{0, 1, 2\}$.

For distinct states $q, q' \in Q$, the presence of accumulation points reached by rationals in I_q^q and $I_{q'}^q$, respectively, can imply *non*-Wheelerness of a language (consider Figure 2 above and see Theorem 17 below). However, the mere presence of accumulation points can easily comply with Wheelerness: Cantor set, for example, can be accepted by a Wheeler automaton using Theorem 13 and the approach outlined here. Fixing $\Sigma = \{0, 1, 2, 3, 4\}$, the automaton $\mathcal{C}$ obtained from the one in Figure 4 replacing 0 by 1 and 2 by 3 and making both its non-starting states final, is Wheeler and complying with Definition 14. Strictly speaking, since we are not using the first and last digits of Σ , none of the rational embeddings of strings in $\mathcal{L}(\mathcal{C})$, that is $I^q(\mathcal{L}(\mathcal{C}))$, is an accumulation point², however $I^q(\mathcal{L}(\mathcal{C}))$ retains most of the properties of the classical Cantor set. More general *Cantor-like* sets are, for example, the rational embeddings of *reverse definite* languages, that is languages $\mathcal{L} = F \cup G\Sigma^*$, for finite $F, G \subseteq \Sigma^*$, or *strictly locally testable* languages, that is languages $\mathcal{L} = F \cup (H\Sigma^* \cap \Sigma^*K) \setminus \Sigma^*W\Sigma^*$ for finite $H, K, W, F \subseteq \Sigma^*$.

In general, the rational embedding of a Wheeler language can have many accumulation points – think of the trivial case of $\mathcal{L} = \Sigma^*$ – and the analysis of their position within different I_q^q s, can provide useful insights. Consider, for example, the following sufficient condition for non-Wheelerness.

¹ The celebrated Cantor set can be seen as the set of rationals in $[0, 1]$ written in base 3 without using the digit 1. Is an example of *perfect* set (all its points are accumulation points) that is also *nowhere dense* (the interior of its closure is empty).

² Hence, $I^q(\mathcal{L}(\mathcal{C}))$ is not perfect.

► **Theorem 17** ([47]). *If $\mathcal{D}$ is the minimum DFA accepting $\mathcal{L}$ and for distinct $q, q' \in Q$,*

- $x = \inf(I_q^{\mathcal{Q}}) = \inf(I_{q'}^{\mathcal{Q}})$ or $x = \sup(I_q^{\mathcal{Q}}) = \sup(I_{q'}^{\mathcal{Q}})$ and
- x is an accumulation point for both $I_q^{\mathcal{Q}}$ and $I_{q'}^{\mathcal{Q}}$,

then $\mathcal{L}$ is not Wheeler.

4.2 State Complexity

The *state complexity* (also known as *quotient complexity*) of a regular language is naturally defined as the number of states of the minimum DFA $\mathcal{D}_{\mathcal{L}}$ recognizing $\mathcal{L}$. State complexity is also used to measure the complexity of operations on regular languages: the state complexity of an operation is a function that, starting from the state complexities of the operands, associate the worst-case state complexity of the language resulting from the operation. For instance, on regular languages the state complexity of the intersection of $\mathcal{L}_1$ and $\mathcal{L}_2$ is mn , where m and n are the number of states of $\mathcal{D}_{\mathcal{L}_1}$ and $\mathcal{D}_{\mathcal{L}_2}$ respectively. This can be seen using the state-product construction for $\mathcal{D}_{\mathcal{L}_1}$ and $\mathcal{D}_{\mathcal{L}_2}$ and in [55] it is proved that this bound is tight. Moreover, as proved in [15], the same complexity is met for the intersection of star-free languages.

If we restrict to Wheeler languages, it is natural to define the Wheeler state complexity of a Wheeler language $\mathcal{L}$ as the number of states of the minimum WDFA $\mathcal{D}_{\mathcal{L}}^W$ recognizing $\mathcal{L}$. The following lemma shows that the convex property of a Wheeler DFA can be exploited to prove that the Wheeler state complexity of the intersection of Wheeler languages is significantly better than the state complexity of the intersection for regular or star-free languages.

► **Lemma 18** ([30]). *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two WDFA recognizing the languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. Then, the direct product $\mathcal{D} := \mathcal{D}_1 \times \mathcal{D}_2$ recognizing the language $\mathcal{L} := \mathcal{L}_1 \cap \mathcal{L}_2$ is Wheeler and, in its trimmed form, it has at most $|Q_1| + |Q_2| - 1$ states, where $|Q_i|$ is the cardinality of the set of states of $\mathcal{D}_i$.*

Here is an intuition for the proof of the previous lemma. The product of two DFA $\mathcal{D}_i = (Q_i, s_i, \delta_i, F_i)$ is the DFA $\mathcal{D}_1 \times \mathcal{D}_2 = (Q_1 \times Q_2, (s_1, s_2), \delta_1 \times \delta_2, F_1 \times F_2)$, where $\delta_1 \times \delta_2((q, r), \sigma) = (\delta_1(q, \sigma), \delta_2(r, \sigma))$. If the $\mathcal{D}_i$'s are Wheeler for $i = 1, 2$, say w.r.t. the Wheeler orders $(Q_i, \leq_i)$, then we can consider the colexicographic order on $Q_1 \times Q_2$:

$$(q_1, p_1) <_{\text{colex}} (q_2, p_2) \Leftrightarrow (p_1 <_2 p_2) \vee [(p_1 = p_2) \wedge (q_1 <_1 q_2)].$$

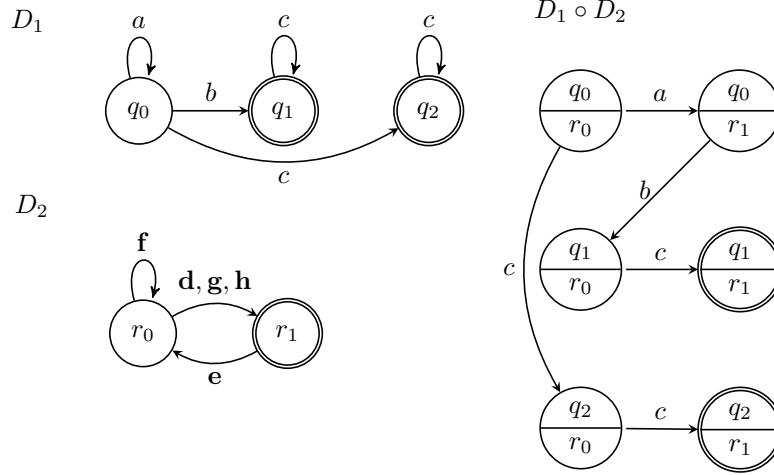
It is then possible to prove that the set of reachable states R in the product $\mathcal{D}_1 \times \mathcal{D}_2$ is such that if $(q_1, p_1) <_{\text{colex}} (q_2, p_2)$ with $(q_1, p_1), (q_2, p_2) \in R$ and $p_1 <_2 p_2$ then $q_1 \leq_1 q_2$ holds as well, and this allows to prove that $|R| \leq |Q_1| + |Q_2|$.

It is natural to ask whether there are more operations, beside intersection, preserving the Wheeler properties. As already mentioned, Wheeler DFA do not behave well when classic operations on DFA are concerned (booleans, concatenation, and Kleene star), so we have to look somewhere else. DFA being special cases of semi-groups, it is there where we can find an interesting operation, the so called *cascade product* of DFA's, and prove that it gets along well with Wheelerness. The cascade product is a generalization of the direct product, in which the parallelism of the computation between the two DFA in the product is substituted by a more hierarchical interaction. Indeed, the second DFA in the cascade does not only read the input string, but also read, moving a step later, the run that the first DFA makes on the input string. To do so, if the alphabet of the first DFA $\mathcal{D}_1 = (Q_1, s_1, \delta_1, F_1)$ is Σ , the alphabet of the second DFA must be $Q_1 \times \Sigma$. More formally, we define the *cascade product* as follows.

► **Definition 19** (Cascade product). Let $\mathcal{D}_1 = (Q_1, s_1, \delta_1, F_1 \times F_2)$ be a DFA over the alphabet Σ and let $\mathcal{D}_2 = (Q_2, Q_1 \times \Sigma, \delta_2, F_2)$ be a second DFA whose alphabet is the cartesian product of $Q_1 \times \Sigma$. The cascade product $\mathcal{D}_1 \circ \mathcal{D}_2 = (Q_1 \times Q_2, (s_1, s_2), \delta, F_1 \times F_2)$ is the automaton over the alphabet Σ with transition function defined by

$$\delta((q, r), a) = (\delta_1(q, a), \delta_2(r, (q, a))).$$

Figure 5 shows an example of cascade product between two automata.



■ **Figure 5** On the left the automata $\mathcal{D}_1$ and $\mathcal{D}_2$, where the $\mathcal{D}_2$ -transitions are: $\mathbf{d} = (q_0, a)$, $\mathbf{e} = (q_0, b)$, $\mathbf{f} = (q_0, c)$, $\mathbf{g} = (q_1, c)$, $\mathbf{h} = (q_2, c)$. On the right, the cascade product $\mathcal{D}_1 \circ \mathcal{D}_2$.

Notice that the direct product between automata can be seen as a particular case of the cascade product: the direct product $\mathcal{D}_1 \times \mathcal{D}_2$ is equal to the cascade product $\mathcal{D}_1 \circ \mathcal{D}'_2$, where $\mathcal{D}'_2$ is obtained from $\mathcal{D}_2$ by replacing each transition $\delta_2(r, a) = r'$ with $|Q_1|$ transitions $\delta'_2(r, (q, a)) = r'$, one for each $q \in Q_1$.

The cascade product plays a fundamental role in the theory of automata (or semi-groups): the celebrated Krohn-Rhodes Decomposition Theorem ([44]) is in fact for semi-groups the analogous of the Jordan-Holder Decomposition Theorem ([6]), allowing to decompose any semi-group (automaton) as a cascade product of “basic” semi-groups (automata).

It is possible to prove that Wheelerness gets along very well with the cascade product. Indeed the following generalization of Lemma 18 holds:

► **Lemma 20** ([28]). If $\mathcal{D}_1 = (Q_1, \Sigma, s_1, \delta_1)$ is Wheeler with respect to the orders $(\Sigma, <)$ and $(Q_1, <_1)$ and $\mathcal{D}_2 = (Q_2, Q_1 \times \Sigma, s_2, \delta_2)$ is Wheeler with respect to the co-lexicographic order on the set of pairs $Q_1 \times \Sigma$ and the order $(Q_2, <_2)$ then $\mathcal{D}_1 \circ \mathcal{D}_2$ (restricted to accessible states) is a WDFA with respect to $(\Sigma, <)$ and the co-lexicographic order over $Q_1 \times Q_2$ (restricted to accessible states).

Moreover, if $\mathcal{D}_1$ has n_1 states and $\mathcal{D}_2$ has n_2 states then the cascade product $\mathcal{D}_1 \circ \mathcal{D}_2$ has at most $n_1 + n_2 - 1$ states.

In Sec 7.1 we shall see that this good behavior of the Wheeler class is not restricted to the intersection/cascade operation, as we find a significant improvement in the determinization of Wheeler NFA as well.

5 Hardness and Lower Bounds

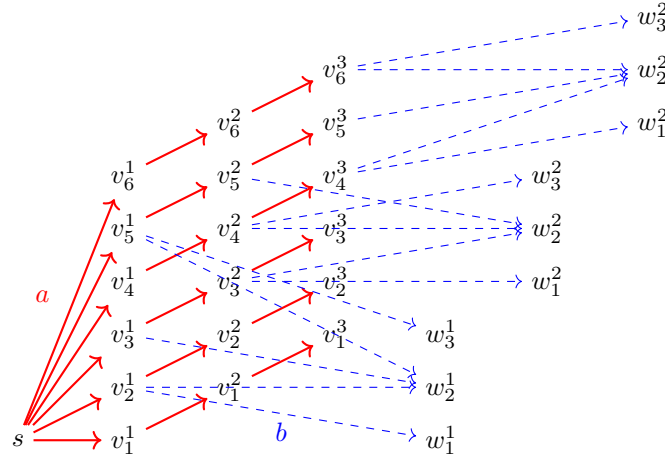
We now move to algorithmic problems on Wheeler graphs. We first discuss the computational complexity of the natural problem of determining if a given edge labeled graph is a Wheeler graph. The following hardness result by Gibney and Thankachan implies the computational intractability of this problem.

► **Theorem 21** ([36]). *Determining if a given labeled graph $G = (Q, \Sigma, E)$ is a Wheeler graph is NP-complete. This holds even when restricted to the class of graphs over binary edge label alphabets having a single source and with vertices having total degree (in-degree + out-degree) at most five.*

We describe a simple reduction that requires a source vertex with a large degree, but demonstrates many of the same techniques used to prove the full statement of Theorem 21. We first introduce the Betweenness Problem, proven NP-complete by Opatrny [50].

► **Problem 1** (Betweenness Problem). *Given an integer n and a set of triples $X \subseteq \{(a, b, c) \mid a, b, c \in \{1, 2, \dots, n\}\}$, determine if there exists a permutation π of $\{1, 2, \dots, n\}$ such that for every $(a, b, c) \in X$, either $\pi(a) < \pi(b) < \pi(c)$ or $\pi(c) < \pi(b) < \pi(a)$.*

We call the set of triples X *betweenness constraints*. For example, an instance of the Betweenness problem is $(n = 6, X = \{(3, 2, 5), (1, 5, 2), (4, 5, 6), (2, 6, 4)\})$. The permutation resulting in the ordering 1, 4, 5, 6, 2, 3 would satisfy every triple in X . The permutation resulting in the ordering 1, 2, 3, 4, 5, 6 would violate the triples $(3, 2, 5)$, $(1, 5, 2)$ and $(2, 6, 4)$.



■ **Figure 6** The labeled graph constructed from the Betweenness Problem instance $(n = 6, X = \{(2, 3, 5), (3, 5, 4), (4, 5, 6)\})$. This graph is a Wheeler graph if and only if there exists a permutation of $\{1, 2, \dots, n\}$ satisfying all betweenness constraints.

Given an instance (n, X) of the Betweenness Problem we construct a labeled graph $G = (Q, \Sigma, E)$ as follows:

- We create a single source vertex s .
- We next create a set of n vertices, denoted v_i^1 for $i \in [1, n]$ and the labeled edges (s, v_i^1, a) for $i \in [1, n]$.
- Next, for $j \in [2, |X|]$, $i \in [1, n]$, we create the vertex set v_i^j and the labeled edges (v_i^{j-1}, v_i^j, a) .

12:16 Wheeler Graphs and Wheeler Languages

Observe that in the definition of a total vertex order $<$ for a Wheeler graph, the vertex s must be ordered first, followed by some ordering of $v_1^1, v_2^1, \dots, v_n^1$, immediately followed by the same order applied to $v_1^2, v_2^2, \dots, v_n^2$. In fact, the vertices described above must have an ordering of the form

$$s < v_{\pi(1)}^1 < \dots < v_{\pi(n)}^1 < v_{\pi(1)}^2 < \dots < v_{\pi(n)}^2 < \dots < v_{\pi(1)}^{|X|} < \dots < v_{\pi(n)}^{|X|}$$

for some permutation π . Otherwise, there would exist some i' and i such that $v_{i'}^{j-1} < v_i^{j-1}$ but $v_i^j < v_{i'}^j$, contradicting Wheeler graph axiom W2.

Next, we add the following betweenness constraint gadgets. We order X arbitrarily. Iterating through X , consider (a, b, c) as the j^{th} betweenness constraint. We add the new vertices w_1^j, w_2^j , and w_3^j and the labeled edges (v_a^j, w_1^j, b) , (v_a^j, w_2^j, b) , (v_b^j, w_2^j, b) , (v_c^j, w_2^j, b) , and (v_c^j, w_3^j, b) . See Figure 6 for an example with $n = 6$ and three betweenness constraint gadgets.

The key property enforced by the betweenness constraint gadget is as follows. If the j^{th} constraint is (a, b, c) , then, for the total order $<$ on the vertices to satisfy Wheeler Axiom W2, the vertex v_b^j must be ordered between v_a^j and v_c^j . For non-example, if $v_b^j < v_a^j < v_c^j$ we must have $w_2^j < w_1^j < w_3^j$ due to the edges (v_b^j, w_2^j, b) , (v_a^j, w_1^j, b) , and (v_c^j, w_3^j, b) . However, we also have the edges (v_a^j, w_1^j, b) and (v_c^j, w_2^j, b) where $v_a^j < v_c^j$ and $w_2^j < w_1^j$, violating Wheeler axiom W2. A similar argument holds in all cases where v_b^j is not ordered between v_a^j and v_c^j . Taking the bipartite view of the graph from Section 2, this is equivalent to the only ordering of the vertices allowing for non-crossing dashed edges labeled 2 being one where the v_b^j is positioned between v_a^j and v_c^j . See Figure 7. By drawing similar figures, one can confirm that the only possible orderings for the vertices in which blue dashed lines do not cross are $v_a^j < v_b^j < v_c^j, w_1^j < w_2^j < w_3^j$ and $v_c^j < v_b^j < v_a^j, w_3^j < w_2^j < w_1^j$.

We note here that if the single source restriction is lifted, then by making each v_i^1 for $i \in [1, n]$ a source vertex, the reduction implies NP-completeness for the case where the total degree of each vertex is at most three.

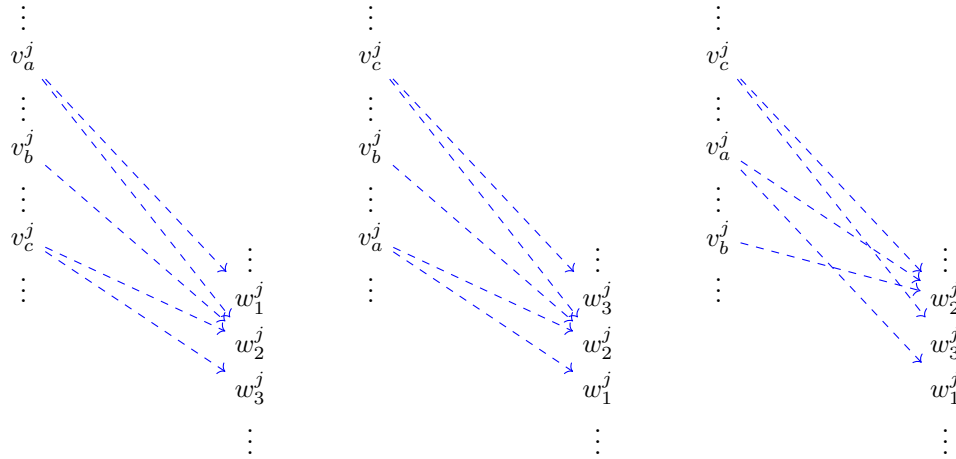
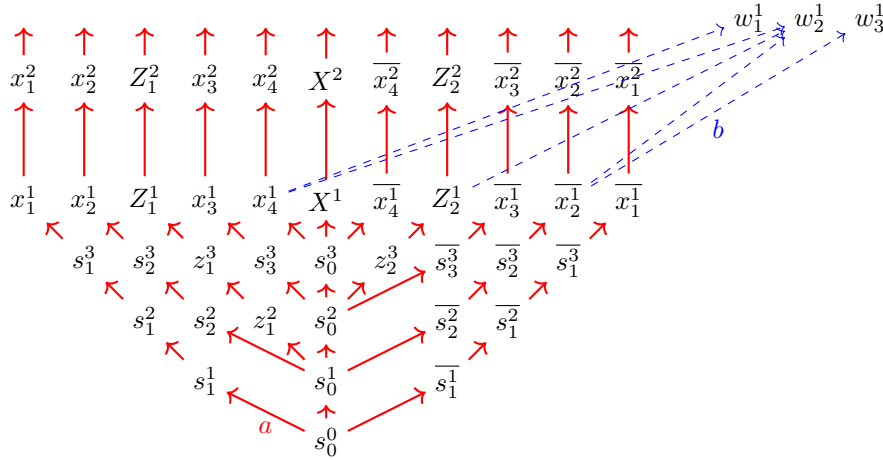


Figure 7 The orderings represented in the left and middle figures are the only vertex orderings that result in no blue dashed edges crossing. The right figure shows an invalid ordering that violates this condition.

We next discuss how the complete statement of Theorem 21 is obtained in the single source case. An examination of Opatrny's NP-completeness proof of the Betweenness Problem shows that not all permutations of $\{1, 2, \dots, n\}$ have to be under consideration for NP-hardness

of the Betweenness problem to hold. Only a subset of permutations allowing for pivoting of elements around some central value X , along with some potential placement of some additional elements have to be under consideration for the problem to be NP-hard. This observation guides a reduction from Not-All-Equal SAT (NAESAT) that utilizes a tree structure rather than a single source with degree n , which previously had allowed for all permutations of $v_1^1, \dots, v_n^1$. An example of this tree structure is shown in Figure 8.

In more detail, in the Not-All-Equal Boolean Satisfiability (NAESAT) Problem, one needs to find a satisfying Boolean assignment in which each clause has both a true literal and a false literal. Like in Opatrný's reduction from NAESAT to the Betweenness Problem, in the reduction from NAESAT to Wheeler graph recognition, each variable x_i is represented by two elements (or vertices) x_i and $\bar{x}_i$. Betweenness constraints force these to be in one of two orderings $x_i < X < \bar{x}_i$ or $\bar{x}_i < X < x_i$ corresponding to either a true or false assignment to x_i . Each clause requires one element (represented with a vertex) and two additional betweenness constraints. After constructing the tree structure, all betweenness constraints can be enforced in the same manner as in the reduction from the Betweenness Problem to Wheeler graph recognition. We refer the readers to [36] for details.



the assumption that $P \neq NP$. The proof is through a reduction from the Feedback Arc Set Problem, that is, given a directed graph $G = (V, E)$, determine a minimum cardinality subset of edges $E' \subset E$ such that $(V, E \setminus E')$ is a directed acyclic graph. This same reduction implies that, under the Unique Games Conjecture, it is NP-hard to find a solution to WGV that is within a constant factor of optimal for any constant $C > 0$ [37].

Further hardness results in this vein include work by D'Agostino et al. [30] that proves that deciding whether a language of an NFA is a Wheeler language is PSPACE-complete. The same authors demonstrate that the problem of determining whether there exists an alphabet order that causes a DFA to become Wheeler is NP-complete [29]. As a fine-grained complexity result, a quadratic lower bound condition on the Strong Exponential Time Hypothesis (SETH) for the problem of determining if a DFA recognizes a Wheeler language is provided by Becker et al. [11], along with an algorithm having a time complexity matching this lower bound. Section 7.3 provides more details on this last result.

6 Sorting Algorithms

As discussed in Section 5, the problems of (i) deciding whether a given NFA is a WNFA and (ii) finding a Wheeler order for a given WNFA are NP-complete. Subsequent research, however, has shown that this is not the end of the story: as we show next, problems (i) and (ii) can still be solved in polynomial time for a strict superclass of the DFA, and can actually be solved in polynomial time for all NFA by slightly changing the Wheeler order definition (i.e. moving to preorders, a solution which preserves the ability to index the WNFA).

6.1 Sorting WDFA and 2-WNFA

Alanko et al. in [2] showed that in the deterministic case (WDFA) the recognition and sorting problems are easy. This result is easily explained using Lemma 9 (Section 4). Intuitively, this Lemma states that a DFA $\mathcal{A}$ is Wheeler if and only if, for any pair of states $u \neq v$ of $\mathcal{A}$, all strings labeling paths from the source of $\mathcal{A}$ to u are co-lexicographically smaller than all strings labeling paths from the source of $\mathcal{A}$ to v (or the other way round). If this is the case, then one can sort the states of $\mathcal{A}$ according to any representative of I_u , for all $u \in Q$. In particular, one can build a spanning tree (rooted in the source) of $\mathcal{A}$ and sort it using existing algorithms for sorting labeled trees (for example, [33]); if $\mathcal{A}$ is Wheeler, then the resulting node order is a Wheeler order. If on the other hand, $\mathcal{A}$ is not Wheeler, then one can easily check in linear time that the resulting order does not satisfy Definition 4. This yields:

► **Theorem 22** ([2]). *Let $\mathcal{A}$ be a DFA. In $O(|\mathcal{A}|)$ time one can:*

1. *Decide whether $\mathcal{A}$ is a Wheeler DFA, and*
2. *If $\mathcal{A}$ is a Wheeler DFA, return its (unique) Wheeler order.*

Importantly, note that Lemma 9 does not hold for NFA: the nondeterministic case is, in fact, much more complicated.

Another interesting case covered in [2] is that of acyclic WDFA, which can be sorted *online* following any topological order of the edges (that is, after the i -th edge is received, the algorithm maintains internally the Wheeler order of all the nodes incident to the i edges received so far). The algorithm is a generalization of a folklore online algorithm for building the Burrows-Wheeler transform in an online fashion [39], and runs with $O(\log |\mathcal{A}|)$ delay per edge.

What about nondeterministic finite automata? While Gibney and Thankachan [36] established that the problem is NP-complete in general (i.e. NFA), a closer look reveals that every node of the labeled graph used in the reduction of [36] has at most five outgoing transitions labeled with the same character. Let us formalize this concept:

► **Definition 23** ([2]). *A d -NFA is a NFA such that, for every state u and every character $a \in \Sigma$, it holds $|\delta(u, a)| \leq d$.*

In other words, the work [36] established that 5-WNFA are NP-hard to recognize and sort. Note that 1-NFA correspond to DFA (recognizable and sortable in linear time, as discussed above), which leaves open the cases $2 \leq d \leq 4$. Surprisingly, [2] showed that also the case $d = 2$ admits a polynomial-time (quadratic) solution. The idea is to encode the candidate Wheeler order as a set of $|Q|^2$ boolean variables (each encoding the statement “ $u < v$ ” for every possible choice of $u, v \in Q$), and to enforce the Wheeler axioms of Definition 4 via a 2-SAT formula of size $O(|\mathcal{A}|^2)$ (solvable in time $O(|\mathcal{A}|^2)$). While Axioms 1-2 are easily expressible using only 2-SAT clauses, the tricky part turns out to be the totality requirement for the Wheeler order and, in particular, transitivity (which requires 3-SAT clauses in the general case). The core of the reduction shown in [2] is to prove that the Wheeler axioms, together with antisymmetry and totality of the order (also easily expressible via 2-SAT clauses), imply transitivity on 2-NFA. As a result, transitivity does not need to be enforced at all and a 2-SAT formula can be used to find a Wheeler order, if one exists.

2-SAT formulas can also be used to attack the sorting problem for the union of two Wheeler automata (see [31] for details).

6.2 Sorting Arbitrary WNFA: Preorders and Relations

As it turns out, the NP-completeness of the problem of recognizing and sorting WNFA with the Wheeler order defined in Definition 4, does not mean that such automata cannot be indexed for pattern matching queries in polynomial time. Another closer look at the NP-completeness reduction of Gibney and Thankachan [36] reveals that the labeled graph (in fact, a semiautomaton with a single source) of their reduction has another peculiarity: several distinct node pairs $u \neq v$ satisfy $I_u = I_v$, i.e. they are reached by exactly the same strings from the source. Observe that such pairs of nodes can safely be merged without affecting the strings that can be read on the graph’s paths; in particular, the answers to pattern matching existential queries (does a given query string α label any path in the graph?) are not affected by this transformation. One idea could therefore be to not take any decision on the relative order of such equivalent nodes, simply deeming them as being equivalent in the order. In other words, we may slightly change our goal: rather than looking for a *strict* Wheeler order, we may look for a *preorder*. As shown by Alanko et al. in [3], this road is indeed the right way to go. A particular case is the one where no equivalent states exist. In this case, Alanko et al. [3] proved:

► **Theorem 24** ([3]). *Let us call reduced a semi-automaton $\mathcal{A} = (Q, \Sigma, \delta)$ such that $I_u \neq I_v$ for all pairs of states $u, v \in Q$ with $u \neq v$. Then, in $O(|\delta| \cdot |Q|^2)$ time we can decide if $\mathcal{A}$ is Wheeler and, if so, compute a Wheeler order for it.*

The algorithm behind Theorem 24 was named *Forward algorithm* due to the fact that it is a partition refinement algorithm “propagating forward” (in the direction of the automaton’s transitions) Wheeler axiom W2. At any point of the algorithm’s execution, a total order $<$ is maintained among the classes of the current partition of the states. The algorithm starts by partitioning the automaton’s states into $|\Sigma|$ classes, according to the labels of

their incoming transitions (recall that we require *input-consistency*: all incoming transitions of a given state bear the same label); classes are sorted according to the total order of Σ . This step corresponds to enforcing Wheeler Axiom W1. At this point, the algorithm implements iteratively the following observation. Fix a given character a and a class C . Let moreover S (the *splitter*) be the smallest class (in the total order of the classes that we are maintaining) such that $C \cap \delta(S, a) \neq \emptyset$. Then, states in $C' = C \cap \delta(S, a)$ must precede those in $C'' = C \setminus \delta(S, a)$ in *any* Wheeler order for $\mathcal{A}$ (if $\mathcal{A}$ is Wheeler). Split C into C' and C'' and record the order $C' < C''$. A careful implementation of this observation results in a partition refinement algorithm – the Forward algorithm – satisfying Theorem 24.

As a matter of fact, the Forward algorithm achieves much more than what is stated in Theorem 24: it can sort *any* WNFA (actually, even a strict superclass of the WNFA) with a preorder that still allows indexing the NFA for pattern matching queries. First, it is worth noting that the partition output by the Forward algorithm is the *coarsest forward stable partition* of the NFA, i.e. the coarsest partition such that, for every pair of classes S, C and every $a \in \Sigma$, either $C \cap \delta(S, a) = \emptyset$ or $C \subseteq \delta(S, a)$ hold. Furthermore, the following properties, shown in [3] and later studied more in detail in [9, 13], hold:

► **Lemma 25** ([3]). *Let $\equiv$ be the equivalence relation induced by the coarsest forward-stable partition of $\mathcal{A}$'s states. Let moreover $\mathcal{A}_{/\equiv}$ denote the quotient automaton obtained by collapsing states of $\mathcal{A}$ being $\equiv$ -equivalent. Then, the following properties hold:*

1. *If $u \equiv v$, then $I_u = I_v$.*
2. *If $\mathcal{A}_{/\equiv}$ is Wheeler, then the Forward algorithm finds a Wheeler order for $\mathcal{A}_{/\equiv}$.*
3. *If $\mathcal{A}$ is Wheeler, then $\mathcal{A}_{/\equiv}$ is Wheeler.*
4. *The converse of property 3 does not hold: there exist non-Wheeler $\mathcal{A}$ such that $\mathcal{A}_{/\equiv}$ is Wheeler.*

Recalling that states such that $I_u = I_v$ can be safely collapsed for pattern matching purposes, the above Lemma states that any Wheeler NFA can be indexed for pattern matching queries in polynomial time, despite the fact that recognizing and sorting WNFA is an NP-complete problem. This apparently counterintuitive result is made possible by Property 4 of Lemma 25: there do exist non-Wheeler NFA $\mathcal{A}$ such that $\mathcal{A}_{/\equiv}$ is Wheeler (see [3] for a concrete example of such a NFA). In other words, the Forward algorithm cannot decide if the input NFA $\mathcal{A}$ is Wheeler, but if it is it can index an equivalent (Wheeler) automaton $\mathcal{A}_{/\equiv}$. What's more, there exist non-Wheeler NFA $\mathcal{A}$ that can still be indexed by the Forward algorithm, because they are such that $\mathcal{A}_{/\equiv}$ is Wheeler.

Another partition-refinement strategy for propagating forward the Wheeler Axiom W2 was later independently proposed in [18], although in that work the authors did not prove that their partition-refinement algorithm yields the coarsest forward-stable partition. In that paper, the goal was however slightly different than that of the previously-mentioned results: once obtained an ordered partition consistent with any Wheeler order of the input NFA, the algorithm proposed in [18] goes on to find in worst-case exponential time a total order also among states still belonging to the same partition's class (for which an order has not been established yet) by exhaustive search or by using a Satisfiability Modulo Theory (SMT) solver. The output of this tool is therefore a Wheeler order for the original NFA $\mathcal{A}$ (NP-complete to find), rather than a Wheeler order for an equivalent quotient automaton as done in [3, 9] (computable in polynomial time). As shown in [18], this heuristic is in fact very efficient since in practice the equivalence classes output by the first partition refinement step are small, thus even a brute-force (or solver-based) enumeration of the remaining permutations to be tested (those of equivalent nodes) is fast.

The cubic complexity of the Forward algorithm was finally reduced to nearly-linear $O(|\delta| \log |Q|)$ in [9], who proved:

► **Theorem 26** ([9]). *Let $\mathcal{A} = (Q, \Sigma, \delta)$ be a (semi-) NFA and $\equiv$ be the equivalence relation corresponding to the coarsest forward-stable partition of $\mathcal{A}$'s states. Then, in $O(|\delta| \log |Q|)$ time one can compute a total order among the states of $\mathcal{A}_{/\equiv}$ that is a valid Wheeler order for $\mathcal{A}_{/\equiv}$, if $\mathcal{A}_{/\equiv}$ is Wheeler.*

The idea behind the result in Theorem 26 is to use the “smallest-splitter” rule of the classic relational coarsest partition refinement algorithm of Paige and Tarjan [51], opportunely adapted to work with ordered partitions. Intuitively, consider a class C that has been split into the (totally-ordered) classes $C_1 < \dots < C_k$ using some splitter class S (while in the Forward algorithm each class is split into $k = 2$ sub-classes as mentioned above, the algorithm from [9] uses a three-way split and thus $k \leq 3$; the initial class Q is an exception since it is initially split into $k = |\Sigma| + 1$ classes, according to the states' incoming labels). In the next steps, we will choose the smallest between C_1 and C_k as splitter (Paige and Tarjan's algorithm, instead, chooses the smallest between C_1 and C_2). This choice guarantees that (i) we can still propagate Wheeler Axiom W2, and (ii) each state ultimately belongs to at most $\log |Q|$ splitters, leading to the claimed linearithmic complexity.

The NP-completeness of the problem of recognizing Wheeler NFA can also be attacked using a more general method based on *arbitrary relations* [20]. The key idea is that the antisymmetry and transitivity of a Wheeler order are not required to extend the Burrows-Wheeler transform and the FM-index to automata and so can be dropped. A “Wheeler relation” still satisfies axioms (W1) and (W2), but there may exist states u and v for which both $u < v$ and $v < u$ hold. The existence of a Wheeler relation can be checked in polynomial time [20], and this powerful approach can be extended to arbitrary automata (see Section 7.4).

7 Algorithms on Wheeler Automata and Languages

The wonders of Wheeler automata are not limited to their language-theoretic and indexing properties. In this section, we show that Wheeler automata and languages possess several additional remarkable properties that imply efficient algorithms for problems which are typically hard on arbitrary automata.

7.1 Determinization via Powerset Construction

The first such remarkable property is determinization. Consider a Wheeler NFA $\mathcal{A} = (Q, s, \Sigma, \delta, F)$. Consider moreover running the classic powerset construction algorithm on $\mathcal{A}$, yielding an equivalent DFA $\mathcal{D}$. A classic result is that $\mathcal{D}$ has at most $2^{|Q|}$ states when $\mathcal{A}$ is an arbitrary NFA. As shown in [2], in the Wheeler case the situation is radically different: if $\mathcal{A}$ is a WNFA then $\mathcal{D}$ is a WDFA with at most $2|Q| - 1 - |\Sigma|$ states. In other words, determinization of a WNFA maintains the Wheeler property and causes a blow-up in the number of states by a factor of (at most) 2, rather than exponential as in the general case. This result has several interesting implications. First of all, we deduce that WNFA and WDFA have the same expressive power: a language $\mathcal{L}$ is Wheeler if and only if it is recognized by a WNFA, if and only if it is recognized by a WDFA. Second, hard problems on NFA such as equivalence (do two given NFA recognize the same language?) and universality (does a given NFA recognize Σ^* ?) are easy on WNFA (via determinization). Third, in polynomial time we can compute a 2-approximation of the smallest WNFA for a Wheeler language: this

is just the smallest WDFA for the language (computable in polynomial time from any WDFA, as discussed in the following subsection). Note that all such problems are PSPACE-hard on general NFA.

We give an intuition over the above result. Denote by $\mathcal{L}$ the language recognized by the input WNFA. Consider the bipartite representation of the WNFA (see Figure 1, right; even though the figure refers to a WDFA, the following reasoning holds on WNFA as well). The figure should make it clear that, when following all edges labeled with a given character from states belonging to an *interval* (example: in the figure, consider following all edges labeled c from states $q_1 \dots q_4$), we reach another *interval* of states (in the figure, $q_2 \dots q_3$). Now, powerset construction can be equivalently described as the process of creating a DFA state corresponding to set $\delta(s, \alpha) \subseteq Q$, for every possible $\alpha \in \text{Pref}(\mathcal{L})$. Since the singleton set $\{s\}$ is an interval, the above consideration implies that all sets $\delta(s, \alpha)$ are intervals (on any Wheeler order for the WNFA). As there can be at most $(|Q| + 1)|Q|/2 \leq |Q|^2$ distinct intervals on any total order of Q , this already implies that powerset construction can create at most $|Q|^2$ DFA states (which is already a considerable improvement over the $2^{|Q|}$ states of the general case). A more accurate analysis allows to reduce this bound from quadratic to linear. The crucial property that can be proved is that, by Wheeler Axiom W2, no interval $\delta(s, \alpha)$ can be strictly contained inside another interval $\delta(s, \beta)$ for $\alpha, \beta \in \text{Pref}(\mathcal{L})$, except for the cases where $\delta(s, \alpha)$ is either a prefix or a suffix of $\delta(s, \beta)$ on the total order of Q . As it turns out, a family of intervals with this property over a total order with N elements can contain at most $2N - 1$ distinct intervals [2, Lem. 2.1]. Let N_a be the number of nodes with incoming label $a \in \Sigma$, and note that (by input-consistency) any interval $\delta(s, \alpha)$ contains only nodes whose incoming label is $\alpha[|\alpha|]$. We can therefore apply the above bound separately for every $a \in \Sigma$, and obtain that there are at most $\sum_a (2N_a - 1) = 2|Q| - |\Sigma|$ distinct intervals of the form $\delta(s, \alpha)$, for $\alpha \in \text{Pref}(\mathcal{L})$. The final bound $2|Q| - 1 - |\Sigma|$ comes from the observation that only interval $\delta(s, \epsilon) = \{s\}$ contains the source state s . To conclude, one can show that the following order is a Wheeler order among the states of the powerset DFA: for all strings $\alpha < \beta$, with $\alpha, \beta \in \text{Pref}(\mathcal{L})$, if $\delta(s, \alpha) \neq \delta(s, \beta)$ then we deem $\delta(s, \alpha) < \delta(s, \beta)$.

7.2 Linear-Time Minimization of WDFA

A classic algorithmic result in automata theory is that any DFA $\mathcal{D}$ can be minimized in $O(|\mathcal{D}| \log |\mathcal{D}|)$ time via Hopcroft's algorithm [41], outputting a minimum-size (i.e. minimum number of states) DFA $\mathcal{D}'$ equivalent to $\mathcal{D}$. In [2], the authors considered the analogous problem on the restricted space of Wheeler DFA:

► **Problem 2.** *Given a WDFA $\mathcal{D}$, compute the minimum-size equivalent WDFA $\mathcal{W}$.*

This problem can be solved by characterizing the smallest WDFA of a given Wheeler language, either in terms of the Myhill-Nerode equivalence relation [48] (see Theorem 10) or, equivalently, in terms a process merging states of any WDFA $\mathcal{D}$ recognizing it [2, 3]. The latter characterization, which we sketch below, yields an algorithm solving Problem 2 in $O(|\mathcal{D}| \log |\mathcal{D}|)$ time. Later, this time was improved by [1] to optimal $O(|\mathcal{D}|)$.

The authors of [2] proved the following:

► **Theorem 27.** *Let $\mathcal{D}$ be a WDFA. Consider the process of merging all maximal intervals of states $u_1 < \dots < u_t$ such that (i) $\lambda(u_1) = \dots = \lambda(u_t)$ and (ii) the states $u_1, \dots, u_t$ are Myhill-Nerode equivalent (see Definition 2). Then, the resulting automaton $\mathcal{W}$ is the smallest WDFA recognizing $\mathcal{L}(\mathcal{D})$.*

Theorem 27 can be turned into an algorithm by simply observing that the Myhill-Nerode equivalence classes of $\mathcal{D}$'s states can be computed in $O(|\mathcal{D}| \log |\mathcal{D}|)$ time with Hopcroft's algorithm, and the (unique) Wheeler order of $\mathcal{D}$'s states can be computed in linear time using the algorithms of Section 6.1. Why does this algorithm work? First, recalling the bipartite representation of Figure 1, observe that merging an interval of states with the same incoming letter preserves the Wheeler properties (in particular, merging such an interval of states cannot introduce same-letter arc crossings in the bipartite representation). As a result, the automaton $\mathcal{W}$ generated by this process is certainly a WNFA. Moreover, $\mathcal{W}$ is equivalent to $\mathcal{D}$, since we only merge Myhill-Nerode equivalent states. To show that $\mathcal{W}$ is actually deterministic, consider a collapsed interval of states $u_1 < \dots < u_t$, and let $a \in \Sigma$. As previously observed, the image $\delta(\{u_1, \dots, u_t\}, a) = \{v_1, \dots, v_t\}$ of those states through letter a , must also be an interval of states, all reached by letter a . Additionally, $v_1, \dots, v_t$ must be Myhill-Nerode equivalent, otherwise $u_1, \dots, u_t$ would not be Myhill-Nerode equivalent. We conclude that $v_1, \dots, v_t$ must be collapsed into a single state of the output $\mathcal{W}$ by the algorithm, which implies that $\mathcal{W}$ is deterministic. To prove minimality of $\mathcal{W}$, let $u < w < v$ be states of $\mathcal{D}$ sorted according to its Wheeler order $<$ (in particular, u and v are not adjacent in the order), such that u and v are Myhill-Nerode equivalent, but they are not Myhill-Nerode equivalent to w . Then, one can easily observe that merging u and v will violate a Wheeler Axiom (W1 if $\lambda(u) \neq \lambda(v)$, or W2 otherwise). It follows that, when minimizing $\mathcal{D}$ while maintaining the Wheeler properties, the best we can do is to collapse maximal intervals of states according to Theorem 27.

7.3 Recognizing Wheeler Languages

A very natural problem, considered for the first time in [3], is that of recognizing Wheeler languages:

► **Problem 3.** *Given a DFA $\mathcal{D}$, is the language $\mathcal{L}(\mathcal{D})$ Wheeler?*

The solution to Problem 3 provided in [3] is based on Corollary 11 (Section 4). Consider a minimum DFA $\mathcal{D} = (Q, s, \delta, F)$ recognizing a language $\mathcal{L}(\mathcal{D})$. Corollary 11 states that this language is Wheeler if and only if the image through $\delta(s, \cdot)$ of any co-lexicographically monotone sequence of strings $\alpha_1, \alpha_2, \dots$ (that is, this sequence is either increasing or decreasing in co-lexicographic order) belonging to $\text{Pref}(\mathcal{L})$, ultimately stabilizes in a single state of $\mathcal{D}$. In other words, there exists $N \in \mathbb{N}$ and $q \in Q$ such that, for all $i \geq N$, $\delta(s, \alpha_i) = q$. This characterization can be also understood in terms of Theorem 27: if every such sequence stabilizes, then the sequence of states obtained by mapping $\text{Pref}(\mathcal{L})$ (in co-lexicographic order) through $\delta(s, \cdot)$ is formed by a *finite* number of maximal intervals containing just one distinct state. Those intervals essentially correspond to the states of the minimum WDFA for the language. If, on the other hand, there exists such a non-stabilizing monotone sequence, then the sequence of states $\delta(s, \text{Pref}(\mathcal{L}))$ is formed by an *infinite* number of maximal intervals containing just one distinct state, thus no WDFA for the language can exist since, by Theorem 27, we can only merge Myhill-Nerode equivalent states being adjacent in co-lexicographic order.

Using this characterization, [3] devised a dynamic-programming polynomial-time algorithm solving Problem 3. Since the smallest WDFA for a Wheeler language $\mathcal{L}$ could be *exponentially larger* than the smallest DFA for $\mathcal{L}$ [3], the fact that Problem 3 can be solved in polynomial time at all is not trivial. Additionally, [30] showed that Problem 3 becomes PSPACE-complete when the input is an arbitrary NFA, rather than a DFA.

The polynomial-time solution of [3] is based on the observation that a non-stabilizing monotone sequence $\alpha_1, \alpha_2, \dots$ can be identified by searching for particular *cycles* in the smallest DFA for the language. More in detail, [3] proved that such a non-stabilizing monotone sequence exists if and only if we can identify two disjoint cycles C_1 and C_2 in the smallest DFA for the language, such that there exist two nodes u, v belonging to C_1 and C_2 , respectively, satisfying: (i) there exists a string γ such that $\delta(u, \gamma) = u$ and $\delta(v, \gamma) = v$, and (ii) there exist strings α, β with $\delta(s, \alpha) = u$ and $\delta(s, \beta) = v$ such that either $\alpha, \beta \prec \gamma$ or $\gamma \prec \alpha, \beta$ hold.

The work [3] showed that such three sequences α, β, γ (and corresponding cycles) can be identified via dynamic programming. The drawback of this solution was the degree of the polynomial running time: $\Omega(|\mathcal{D}|^{13})$. Later, this polynomial was reduced to $\Omega(|\mathcal{D}|^2)$ by [11], who also proved a conditional lower bound stating that, unless the Strong Exponential Time Hypothesis [42] (SETH) fails, Problem 3 cannot be solved in strongly sub-quadratic time. Thus, this work essentially settled the complexity of the problem. The intuition behind this result is the following. On the upper-bound side, [11] provided a simpler cycle-based characterization of Wheeler languages: a language $\mathcal{L}$ is Wheeler if and only if its minimum DFA $\mathcal{D}$ does not have two distinct states $u \neq v$ such that (i) the open intervals (in co-lexicographic order) $(\inf I_u, \sup I_u)$ and $(\inf I_v, \sup I_v)$ overlap, where $\inf X$ and $\sup X$ denote the infimum and supremum of set X , respectively, and (ii) $\delta(u, \alpha) = u$ and $\delta(v, \alpha) = v$ for some string α (i.e. u and v belong to two disjoint equally-labeled cycles). Condition (i) can be checked in $O(|\mathcal{D}|^2)$ time simultaneously on all state pairs u, v by computing $\inf I_u, \sup I_u$ for all states u using the partition-refinement algorithm of [9]. Condition (ii) can then be verified in $O(|\mathcal{D}|^2)$ time by checking if the square automaton $\mathcal{D} \times \mathcal{D}$ contains cycles involving state pairs (u, v) satisfying condition (i). On the lower-bound side, [11] observed that one can convert any instance of the *Orthogonal Vectors (OV) problem* (decide if there exist orthogonal vectors in two sets of binary vectors) into a (minimum) DFA $\mathcal{D}$ of the same asymptotic size of the OV instance having two disjoint equally-labeled cycles if and only if the original OV instance has two orthogonal vectors. By building $\mathcal{D}$ in such a way that condition (i) holds on all node pairs, one then obtains that there exist two orthogonal vectors in the OV instance if and only if the language of $\mathcal{D}$ is Wheeler. By a classic reduction [54], OV cannot be solved in strongly subquadratic time unless the Strong Exponential Time Hypothesis [42] (SETH) fails. As a result, Wheeler languages cannot be recognized in strongly subquadratic time (in the input DFA's size), unless SETH fails.

7.4 Other Algorithmic Results

We conclude the section by mentioning other interesting lines of algorithmic research related with Wheeler automata.

DFA to minimum WDFA

The following variant of Problem 2 was first considered in [2]:

► **Problem 4.** *Given a minimum DFA $\mathcal{D}$ recognizing a Wheeler language, compute the minimum-size equivalent WDFA $\mathcal{W}$.*

Since $\mathcal{W}$ can be exponentially-larger than $\mathcal{D}$ [3], Problem 4 cannot be solved in sub-exponential time. The goal is then to solve the problem in time as close as possible to the output size $|\mathcal{W}|$. Alanko et al. solved Problem 4 in near-optimal time $O(|\mathcal{W}| \log |\mathcal{W}|)$ in the particular case where $\mathcal{D}$ is acyclic. The idea is to process the transitions of $\mathcal{D}$ in any

topological order, inserting them in $\mathcal{W}$ (initially empty). Whenever inserting a transition in $\mathcal{W}$, if this breaks Wheeler Axiom W2, i.e. if this transition produces a crossing in the bipartite representation of $\mathcal{W}$ (see Figure 1), then a state of $\mathcal{W}$ is split into two states (partitioning the incoming transitions of the state into two parts) in order to remove the axiom violation. This operation is supported by encoding $\mathcal{W}$ with a dynamic data structure essentially storing the bipartite representation of $\mathcal{W}$. The $\log |\mathcal{W}|$ multiplicative factor in the running time comes from the query time of this dynamic structure. Later, [30] solved Problem 4 in the general case (i.e. $\mathcal{D}$ is any DFA recognizing a Wheeler language) in time $O(|\mathcal{D}|^3 |\mathcal{W}| \log |\mathcal{W}|)$.

Random WDFA generation

Another interesting algorithmic problem, considered in [10], is that of generating Wheeler DFA according to a uniform distribution over all Wheeler DFA with a given number of states, transitions, and alphabet size. Interestingly, the paper shows that the Wheeler properties allow solving the problem in expected linear time (in the output size) and *constant* working space, provided that the transitions of the WDFA are streamed to output following the Wheeler order of their destination states. Such a result is not known to be possible for arbitrary DFA.

Computing the union and complementation of Wheeler languages

In [31], the authors study the problem of computing, given two Wheeler automata, a Wheeler automaton recognizing the union of their languages, provided that the union itself is Wheeler (as mentioned in Section 4, Wheeler languages are not closed under union). Their solution runs in quadratic time and linear (succinct) space. An alternative solution to the problem was later provided by [17], by providing an algorithm for *completing* WDFA (when such a completion is possible at all), and then applying the classical DFA complementation/union algorithms working on complete DFA.

Generalization to arbitrary automata

To conclude, a fruitful line of research initiated by [25, 27] and continued in [20, 13, 9, 21, 43, 5, 22, 24, 12, 23] later showed that almost all the results on Wheeler automata described in this survey can be generalized to *arbitrary* automata by simply not requiring the Wheeler order of Definition 4 to be total. As shown in [25, 27], any automaton admits a *partial* order satisfying the axioms of Definition 4. Interestingly, the *width* p of such a partial order – that is, the size of its largest antichain (a parameter being equal to 1 on total orders, i.e. Wheeler automata) – parameterizes several problems on automata being hard in the general case. Examples include: pattern matching on graphs (See Section 3) can be performed in time proportional to p^2 per character in the input pattern, automata can be encoded using $O(\log p)$ bits per edge, and the powerset construction algorithm for turning NFA into equivalent DFA runs in time exponential in p (the width of the input NFA's order), rather than in the size of the input NFA (as a classical result shows).

References

- 1 Jarno Alanko, Nicola Cotumaccio, and Nicola Prezza. Linear-time minimization of wheeler dfas. In *2022 Data Compression Conference (DCC)*, pages 53–62, 2022. doi:10.1109/DCC52660.2022.00013.

- 2 Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 911–930. SIAM, 2020. doi:10.1137/1.9781611975994.55.
- 3 Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Wheeler languages. *Information and Computation*, 281:104820, 2021. doi:10.1016/J.IC.2021.104820.
- 4 Jarno Alanko, Travis Gagie, Gonzalo Navarro, and Louisa Seelbach Benkner. Tunneling on wheeler graphs. In *2019 Data Compression Conference (DCC)*, pages 122–131, 2019. doi:10.1109/DCC.2019.00020.
- 5 Jarno N. Alanko, Davide Cenzato, Nicola Cotumaccio, Sung-Hwan Kim, Giovanni Manzini, and Nicola Prezza. Computing the LCP Array of a Labeled Graph. In *35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024)*, volume 296 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:15, 2024. doi:10.4230/LIPIcs.CPM.2024.1.
- 6 Michael Aschbacher. *Finite Group Theory*. Cambridge studies in advanced mathematics. Cambridge University Press, 1986.
- 7 László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 171–183. ACM, 1983. doi:10.1145/800061.808746.
- 8 Uwe Baier. On Undetected Redundancy in the Burrows-Wheeler Transform. In *29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:15, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2018.3.
- 9 Ruben Becker, Manuel Cáceres, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Francisco Olivares, and Nicola Prezza. Sorting Finite Automata via Partition Refinement. In *31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2023.15.
- 10 Ruben Becker, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Riccardo Maso, and Nicola Prezza. Random Wheeler Automata. In *35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024)*, volume 296 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:15, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2024.5.
- 11 Ruben Becker, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Alberto Policriti, and Nicola Prezza. Optimal wheeler language recognition. In *String Processing and Information Retrieval*, pages 62–74, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-43980-3_6.
- 12 Ruben Becker, Nicola Cotumaccio, Sung-Hwan Kim, Nicola Prezza, and Carlo Tsoni. Encoding co-lex orders of finite-state automata in linear space. In *36th Annual Symposium on Combinatorial Pattern Matching*, page 17, 2025.
- 13 Ruben Becker, Sung-Hwan Kim, Nicola Prezza, and Carlo Tsoni. Indexing finite-state automata using forward-stable partitions. In *International Symposium on String Processing and Information Retrieval*, pages 26–40. Springer, 2024. doi:10.1007/978-3-031-72200-4_3.
- 14 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In *Algorithms in Bioinformatics*, pages 225–235, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-33122-0_18.
- 15 Janusz A. Brzozowski and Bo Liu. Quotient complexity of star-free languages. *Int. J. Found. Comput. Sci.*, 23(6):1261–1276, 2012. doi:10.1142/S0129054112400515.
- 16 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical report, DIGITAL SRC RESEARCH REPORT, 1994.
- 17 Giuseppa Castiglione, Antonio Restivo, et al. Completing wheeler automata. In *CEUR Workshop proceedings*, volume 3811, pages 120–132. CEUR-WS, 2024. URL: <https://ceur-ws.org/Vol-3811/paper060.pdf>.
- 18 Kuan-Hao Chao, Pei-Wei Chen, Sanjit A Seshia, and Ben Langmead. Wgt: Tools and algorithms for recognizing, visualizing, and generating wheeler graphs. *Iscience*, 26(8), 2023.

- 19 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on wheeler dfas. In *2023 Data Compression Conference (DCC)*, pages 150–159. IEEE, 2023. doi:10.1109/DCC55655.2023.00023.
- 20 Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In *2022 Data Compression Conference (DCC)*, pages 272–281. IEEE, 2022.
- 21 Nicola Cotumaccio. Prefix sorting dfas: A recursive algorithm. In *34th International Symposium on Algorithms and Computation (ISAAC 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.
- 22 Nicola Cotumaccio. A Myhill-Nerode Theorem for Generalized Automata, with Applications to Pattern Matching and Compression. In *41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)*, volume 289 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2024.26.
- 23 Nicola Cotumaccio. Fast pattern matching with epsilon transitions. In *International Workshop on Combinatorial Algorithms*, pages 242–255. Springer, 2025.
- 24 Nicola Cotumaccio. Improved circular dictionary matching. In *36th Annual Symposium on Combinatorial Pattern Matching (CPM 2025)*, pages 18–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025.
- 25 Nicola Cotumaccio, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages-part i. *Journal of the ACM*, 70(4):1–73, 2023. doi:10.1145/3607471.
- 26 Nicola Cotumaccio, Travis Gagie, Dominik Köppl, and Nicola Prezza. Space-time trade-offs for the lcp array of wheeler dfas. In *String Processing and Information Retrieval: 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26–28, 2023, Proceedings*, pages 143–156, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-43980-3_12.
- 27 Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2585–2599. SIAM, 2021. doi:10.1137/1.9781611976465.153.
- 28 Giovanna D’Agostino, Luca Geatti, Davide Martincigh, and Alberto Policriti. Cascade products and wheeler automata. *Theor. Comput. Sci.*, 1013:114754, 2024. doi:10.1016/J.TCS.2024.114754.
- 29 Giovanna D’Agostino, Davide Martincigh, and Alberto Policriti. Ordering regular languages: a danger zone. In *Proceedings of the 22nd Italian Conference on Theoretical Computer Science, Bologna, Italy, September 13-15, 2021*, volume 3072 of *CEUR Workshop Proceedings*, pages 46–69. CEUR-WS.org, 2021. URL: <https://ceur-ws.org/Vol-3072/paper5.pdf>.
- 30 Giovanna D’Agostino, Davide Martincigh, and Alberto Policriti. Ordering regular languages and automata: Complexity. *Theoretical Computer Science*, 949:113709, 2023. doi:10.1016/j.tcs.2023.113709.
- 31 Lavinia Egidi, Felipe A Louza, and Giovanni Manzini. Space efficient merging of de bruijn graphs and wheeler graphs. *Algorithmica*, 84(3):639–669, 2022. doi:10.1007/S00453-021-00855-2.
- 32 Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Trans. Algorithms*, 19(3), April 2023. doi:10.1145/3588334.
- 33 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), November 2009. doi:10.1145/1613676.1613680.
- 34 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005. doi:10.1145/1082036.1082039.
- 35 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theoretical computer science*, 698:67–78, 2017. doi:10.1016/J.TCS.2017.06.016.

- 36 Daniel Gibney and Sharma V Thankachan. On the complexity of recognizing wheeler graphs. *Algorithmica*, 84(3):784–814, 2022. doi:10.1007/S00453-021-00917-5.
- 37 Venkatesan Guruswami, Rajsekar Manokaran, and Prasad Raghavendra. Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 573–582. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.51.
- 38 Julius Hartmanis and Richard E. Stearns. Sets of numbers defined by finite automata. *The American Mathematical Monthly*, 74(5), 1967.
- 39 Wing-Kai Hon, Tak-Wah Lam, Kunihiro Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48:23–36, 2007. doi:10.1007/S00453-006-1228-8.
- 40 Wing-Kai Hon, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Succinct indexes for circular patterns. In *Algorithms and Computation*, pages 673–682, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25591-5_69.
- 41 John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.
- 42 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.
- 43 Sung-Hwan Kim, Francisco Olivares, and Nicola Prezza. Faster Prefix-Sorting Algorithms for Deterministic Finite Automata. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2023.16.
- 44 Kenneth Krohn and John Rhodes. Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–464, 1965.
- 45 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 46 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the burrows–wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007. The Burrows-Wheeler Transform. doi:10.1016/j.tcs.2007.07.014.
- 47 Giovanni Manzini, Alberto Policriti, Nicola Prezza, and Brian Riccardi. The rational construction of a wheeler DFA. In *35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, June 25-27, 2024, Fukuoka, Japan*, volume 296 of *LIPIcs*, pages 23:1–23:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.CPM.2024.23.
- 48 Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
- 49 Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *String Processing and Information Retrieval*, pages 347–358, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-16321-0_36.
- 50 Jaroslav Opatrny. Total ordering problem. *SIAM J. Comput.*, 8(1):111–114, 1979. doi:10.1137/0208008.
- 51 Robert Paige and Robert Endre Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987. doi:10.1137/0216062.
- 52 Jacques Sakarovitch. *Elements of automata theory*. Cambridge university press, 2009.
- 53 Huei Jang Shyr and Gabriel Thierrin. Ordered automata and associated languages. *Tamkang J. Math*, 5:9–20, 1974.
- 54 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.
- 55 Sheng Yu, Qingyu Zhuang, and Kai Salomaa. The state complexities of some basic operations on regular languages. *Theor. Comput. Sci.*, 125:315–328, 1994. doi:10.1016/0304-3975(92)00011-F.

Graph Indexing Beyond Wheeler Graphs

Jarno N. Alanko ✉ 

Department of Computer Science, University of Helsinki, Finland

Massimo Equi ✉ 

Department of Computer Science, Aalto University, Finland

Simon J. Puglisi ✉ 

Department of Computer Science, University of Helsinki, Finland

Kunihiko Sadakane ✉ 

Department of Mathematical Informatics, University of Tokyo, Japan

Elena Biagi ✉ 

Department of Computer Science, University of Helsinki, Finland

Veli Mäkinen ✉ 

Department of Computer Science, University of Helsinki, Finland

Nicola Rizzo ✉ 

Department of Computer Science, University of Helsinki, Finland

Jouni Sirén ✉ 

Genomics Institute, University of California, Santa Cruz, CA, USA

Abstract

After the discovery of the FM index, which linked the Burrows-Wheeler transform (BWT) to pattern matching on strings, several contemporaneous strands of research began on indexing more complex structures with the BWT, such as tries, finite languages, de Bruijn graphs, and aligned sequences. These directions can now be viewed as culminating in the theory of Wheeler Graphs, but sometimes they go beyond. This chapter reviews the significant body of “proto Wheeler Graph” indexes, many of which exploit characteristics of their specific case to outperform Wheeler graphs, especially in practice.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Theory of computation → Graph algorithms analysis; Theory of computation → Pattern matching; Theory of computation → Formal languages and automata theory

Keywords and phrases indexing, compression, compressed data structures, string algorithms, pattern matching

Digital Object Identifier 10.4230/OASICS.Manzini.2025.13

Funding *Jarno N. Alanko*: Funded by the Helsinki Institute for Information Technology (HIIT).

Elena Biagi: Funded by the Academy of Finland via grant 339070.

Massimo Equi: Partially funded by the Research Council of Finland, Grant 359104.

Veli Mäkinen: Partially funded by the European Union’s Horizon Europe research and innovation programme under grant agreement No 101060011 (TeamPerMed).

Simon J. Puglisi: Funded by the Academy of Finland via grant 339070.

Nicola Rizzo: Funded by the European Union’s Horizon Europe research and innovation programme under grant agreement No 101060011 (TeamPerMed).

Jouni Sirén: Funded by the National Human Genome Research Institute (NHGRI) under award numbers U01HG013748 and U41HG010972.

Acknowledgements We wish to thank Hideo Bannai for spotting a mistake in the original version of Definition 22.

1 Introduction

After the discovery of the FM index [22], designed for pattern matching on strings using the Burrows-Wheeler transform (BWT)[11], many groups explored BWT-inspired indexing of more complex structures, such as tries [21], finite languages [48, 49], de Bruijn graphs [10, 5], and aligned sequences [20, 43]. These directions can now be viewed as culminating in



© Jarno N. Alanko, Elena Biagi, Massimo Equi, Veli Mäkinen, Simon J. Puglisi, Nicola Rizzo, Kunihiko Sadakane, and Jouni Sirén;
licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini’s 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 13; pp. 13:1–13:29



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the theory of Wheeler Graphs, which is covered in the next chapter. Before that, however, in this chapter we review the significant body of “proto Wheeler Graph” indexes, many of which exploit characteristics of their specific case to outperform Wheeler graphs, especially in practice. While indeed many of the approaches reviewed (but not all) can be encapsulated in the general Wheeler Graph framework, the connection usually covers only the last indexing step. Construction of the object to be indexed is typically the most complex step and this is tailored for each approach. Especially these construction considerations go beyond the Wheeler Graph framework.

We start with key notions and index structures for strings and string collections. Before proceeding to general graphs, we extend the techniques to tries. Then we continue with de Bruijn graphs that represent overlaps of length k substrings of a string collection. The studied extensions of BWT are then shown to work for general graphs, albeit with the caveat of worst case exponential index size. This worst case behavior is then avoided by resorting back to de Bruijn graphs, but this time indexing paths of a graph rather than string collection. Finally, we explore an approach to directly build graphs from aligned sequences that have indexing properties similar to de Bruijn graphs, but without fixed parameter k for the substring length.

2 Notation

2.1 Strings

Let $S = S[1] \cdots S[n]$ be a string of length $|S|$ over an alphabet Σ of size $|\Sigma|$. If the string and the alphabet are clear from the context, we often denote their sizes as $n = |S|$ and $\sigma = |\Sigma|$. We write substrings as $S[i : j] = S[i] \cdots S[j]$ and call substrings of the form $S[: j] = S[1 : j]$ and $S[i :] = S[i : n]$ prefixes and suffixes, respectively. Let $\$ \in \Sigma$ be the smallest symbol in the alphabet. We often use it as a sentinel $S[n] = \$$ that occurs only at the end of the string.

Let S be a string of length n , let $i \in \mathbb{Z}$ be an offset in it, and let $c \in \Sigma$ be a symbol. We define $S.\text{rank}(i, c)$ as the number of occurrences of symbol c in the prefix $S[1 : i]$. For convenience, we define $S.\text{rank}(i, c) = 0$ when $i < 1$ and $S.\text{rank}(i, c) = S.\text{rank}(n, c)$ when $i > n$.

We define the inverse operation $S.\text{select}(i, c)$ as the position of the occurrence of c of rank i . Let $n_c = S.\text{rank}(n, c)$ be the total number of occurrences of symbol c in the string. For $1 \leq i \leq n_c$, we have $j = S.\text{select}(i, c)$ such that $S.\text{rank}(j, c) = i$ and $S.\text{rank}(j - 1, c) = i - 1$. We also define $S.\text{select}(0, c) = 0$ and $S.\text{select}(n_c + 1, c) = n + 1$ for convenience and leave the function undefined for the remaining values of i .

A *bitvector* is a data structure that stores a binary string and supports efficient rank/select queries. If we store the string as such, we can support both queries in $O(1)$ time and $n + o(n)$ bits of space [14, 37]. When the bitvector is sparse, we can achieve better space usage by using Elias–Fano encoding to store the positions with ones [31, 42]. If there are n_1 ones in a bitvector of length n , we can store it in $n_1 \log(n/n_1) + O(n_1)$ bits of space and support select queries in $O(1)$ time and rank queries in $O(\log(n/n_1))$ time.

2.2 String collections

Let $\mathcal{S} = (S_1, \dots, S_m)$ be an ordered collection of $|\mathcal{S}|$ strings of total length $\|\mathcal{S}\|$. We assume that each string S_i ends with a sentinel $\$_i$ and that $\$_i < \$_j$ iff $i < j$ to make the suffixes of the strings all distinct in lexicographic comparisons. However, we represent the sentinels using the same symbol $\$$ in the actual strings. When the collection is clear from the context,

we use $m = |\mathcal{S}|$ and $N = \|\mathcal{S}\|$ to denote the number of strings and the total length of the strings. When the collection consists of a single string S , we may consider the collection and the string to be the same.

► **Definition 1** (Multiple sequence alignment). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings over alphabet Σ , and let $- \notin \Sigma$ be a gap symbol. Let $\text{spell} : (\Sigma \cup \{-\})^* \rightarrow \Sigma^*$ be a function such that $\text{spell}(S)$ is string S with all occurrences of the gap symbol removed. A multiple sequence alignment of collection $\mathcal{S}$ is a matrix with m rows and n columns $\text{MSA}[1 : m, 1 : n] \in (\Sigma \cup \{-\})^{m \times n}$ such that for each row i , we have $S_i = \text{spell}(\text{MSA}[i, 1 : n])$.*

Let (i, j) and (i', j) be two positions in the same column. If we have $\text{MSA}[i, j] \neq -$ (if $\text{MSA}[i', j] \neq -$), let $S_i[x]$ ($S_{i'}[y]$) be the corresponding symbol. We interpret the alignment in the following way:

- If $\text{MSA}[i, j] = \text{MSA}[i', j] \neq -$, positions $S_i[x]$ and $S_{i'}[y]$ are a *match*.
- If $- \neq \text{MSA}[i, j] \neq \text{MSA}[i', j] \neq -$, positions $S_i[x]$ and $S_{i'}[y]$ are a *mismatch*.
- If $\text{MSA}[i, j] \neq -$ and $\text{MSA}[i', j] = -$, string S_i has an *insertion* relative to string $S_{i'}$ and string $S_{i'}$ has a *deletion* relative to string S_i at the position.

Recombinations let us build new strings by breaking existing strings in two at aligned positions and concatenating the prefix of one string with the suffix of another. Using the above notation, if positions $S_i[x]$ and $S_{i'}[y]$ are a match, we can recombine strings S_i and $S_{i'}$ at those positions as $S_i[:x-1]S_{i'}[y:]$ and $S_{i'}[:y-1]S_i[x:]$. This corresponds to switching between rows i and i' in column j of the multiple sequence alignment.

In later sections, we discuss data structures for indexing all possible recombinations of a string collection compatible with a given alignment. We formalize this as follows.

► **Definition 2** (Closure under recombinations). *Let $\mathcal{S}$ be a collection of strings and let $\sim$ be an equivalence relation for its suffixes such that if $S[j:] \sim S'[j']:$, we have $S[j] = S'[j']$. The closure of collection $\mathcal{S}$ under recombinations is a collection $\mathcal{S}^\sim$ such that:*

1. $\mathcal{S} \subseteq \mathcal{S}^\sim$
2. For any two strings $S, S' \in \mathcal{S}^\sim$ and positions j, j' such that $S[j:] \sim S'[j']:$, it holds that $S'' = S[:j-1]S'[j'] \in \mathcal{S}^\sim$.
3. Given a string $S'' \in \mathcal{S}^\sim$ as above, $S''[j''] \sim S[j'']:$ if $j'' < j$ and $S''[j''] \sim S'[j''-j+j']:$ if $j'' \geq j$.

According to rule 3, each suffix $S''[j'']:$ of the recombined string S'' inherits the alignment of the suffix starting from the corresponding position in the source strings S and S' .

2.3 Arrays over sorted suffixes

Let $\mathcal{S}$ be an ordered collection of strings. We define the (generalized) *suffix array* of collection $\mathcal{S}$ as an array of pointers $\text{SA}[1 : N]$ to the suffixes of the strings in lexicographic order. Each pointer $\text{SA}[x] = (i, j)$ refers to the suffix $S_i[j:]$ of string S_i . Given two pointers $\text{SA}[x] = (i, j)$ and $\text{SA}[x'] = (i', j')$, we have $x < x'$ iff $S_i[j:] < S_{i'}[j']:$ in lexicographic order. We therefore call x the *lexicographic rank* of suffix $S_i[j:]$ among the suffixes of collection $\mathcal{S}$. Note that for $1 \leq x \leq m$, we have $\text{SA}[x] = (x, |S_x|)$.

The multi-string *Burrows–Wheeler transform* (BWT) [11] of collection $\mathcal{S}$ is a string $\text{BWT}[1 : N]$ representing a permutation of the symbols in strings $S \in \mathcal{S}$. We define it using the suffix array. If $\text{SA}[x] = (i, j)$ and $j > 1$, the BWT stores the preceding symbol $\text{BWT}[x] = S_i[j-1]$. When $j = 1$, we set $\text{BWT}[x] = \$$.

In addition to the suffix array and the Burrows–Wheeler transform, we often define other arrays over the lexicographically sorted suffixes. The *longest common prefix (LCP) array* $\text{LCP}[1 : N]$ stores the length of the longest common prefix of two lexicographically adjacent suffixes. If $\text{SA}[x] = (i, j)$ and $\text{SA}[x - 1] = (i', j')$, we store the LCP of suffixes $S_i[j :]$ and $S_{i'}[j' :]$ in $\text{LCP}[x]$, with $\text{LCP}[1] = 0$.

2.4 FM-index

The Burrows–Wheeler transform can be inverted using a function called *LF-mapping* that maps the lexicographic rank of a suffix to the lexicographic rank of its cyclic predecessor. We define it in two parts. Let $\text{SA}[x] = (i, j)$. If $j > 1$, we define $\text{LF}(x) = y$ such that $\text{SA}[y] = (i, j - 1)$. We sometimes write this as $\text{SA}[\text{LF}(x)] = \text{SA}[x] - 1$. For $j = 1$, we define $\text{LF}(x) = y$ such that $\text{SA}[y] = (i, |S_i|)$.

We can compute the LF-mapping for most values of x using the BWT. Let $\text{C}[1 : \sigma]$ be an array such that $\text{C}[c]$ is the total number of occurrences of all symbols $c' < c$ in collection $\mathcal{S}$. Let $\text{SA}[x] = (i, j)$. If $j > 1$, which is equivalent to $\text{BWT}[x] \neq \$$, we have $\text{LF}(x) = \text{C}[\text{BWT}[x]] + \text{BWT.rank}(x, \text{BWT}[x])$. Here $\text{C}[\text{BWT}[x]]$ is the number of suffixes starting with a symbol smaller than $\text{BWT}[x] = S_i[j - 1]$ and $\text{BWT.rank}(x, \text{BWT}[x])$ is the lexicographic rank of suffix $S_i[j - 1 :]$ among the suffixes starting with symbol $\text{BWT}[x]$.

Inverse LF-mapping, denoted as Ψ , can be computed as $\Psi(x) = \text{LF}^{-1}(x)$ by identifying a symbol c such that $\text{C}[c] < x \leq \text{C}[c + 1]$ and then applying $\Psi(x) = \text{BWT.select}(x - \text{C}[c], c)$.

We can generalize LF-mapping for hypothetical suffixes. Let S be the lexicographic rank of string X among the suffixes of collection $\mathcal{S}$. That is, x is the number of suffixes Y of the collection such that $Y \leq X$. Then, given a symbol $c \in \Sigma$ with $c \neq \$$, we define the generalized LF-mapping as the lexicographic rank of string cX . We can compute it as $\text{LF}(x, c) = \text{C}[c] + \text{BWT.rank}(x, c)$.

Let $\text{find}(X) = [sp : ep]$ be the *lexicographic range* of suffixes starting with string X . That is, for each $x \in [sp : ep]$, we have $\text{SA}[x] = (i, j)$ such that $S_i[j : j + |X| - 1] = X$. Then for any symbol c , the range of suffixes starting with cX is $\text{find}(cX) = [\text{LF}(sp - 1, c) + 1 : \text{LF}(ep, c)]$. This backward search step forms the core of the *FM-index* [22]. Given a pattern P , we can determine the range of suffixes $\text{find}(P)$ starting with P with $O(|P|)$ rank operations on the BWT.

Another core operation is $\text{locate}(x) = \text{SA}[x] = (i, j)$, which converts the lexicographic rank of a suffix to its starting position. To compute it, the FM-index needs *suffix array samples* in addition to the BWT. Let $B[1 : N]$ be a bitvector that marks the sampled suffix array positions with 1, and let S be the subsequence of the suffix array corresponding to those positions. If $B[x] = 1$, we can compute $\text{locate}(x)$ as $S[B.\text{rank}(x, 1)]$. Otherwise we rely on iterated LF-mapping and the fact that $\text{SA}[\text{LF}(x)] = \text{SA}[x] - 1$ in the general case. If we need k steps to find the nearest sampled position, we have $\text{locate}(x) = \text{locate}(\text{LF}^k(x)) + k$. For this scheme to function, we need to sample $\text{SA}[x] = (i, j)$ whenever $j = 1$. If we sample $\text{SA}[x]$ every time $j \equiv 1 \pmod{s}$ for a parameter $s > 0$, we can compute $\text{locate}(x)$ with $O(s)$ rank operations.

2.5 Graphs

Let $G = (V, E)$ be a graph with a finite set of vertices V and a set of edges $E \subseteq V \times V$. We assume that the graph is directed. For any two vertices $u, v \in V$, with $u \neq v$, edges (u, v) and (v, u) are distinct. A path is a string $P[1 : n]$ of vertices such that $(P[i], P[i + 1]) \in E$ for $1 \leq i < n$. We call the graph acyclic if there is no path that visits a vertex $v \in V$ more than once.

A *vertex-labeled graph* is a tuple $G = (V, E, \Sigma, \ell)$, where (V, E) is a graph, Σ is an alphabet, and $\ell : V \rightarrow \Sigma$ is a function that assigns a label $\ell(v) \in \Sigma$ to each vertex $v \in V$. If P is a path, we write $\ell(P) = \ell(P[1]) \cdots \ell(P[n])$ to refer to the string formed by concatenating the vertex labels. We can similarly define *edge-labeled graphs* as tuples $G = (V, E, \Sigma, \ell)$, where $\ell : E \rightarrow \Sigma$ is a function that assigns labels to edges. Later, we extend the notions to vertices and edges labeled with non-empty strings.

► **Definition 3** (Alignment graph). *Let $\mathcal{S}$ be a collection of strings over alphabet Σ , and let $\sim$ be an equivalence relation for its suffixes such that if $S[j:] \sim S'[j']:$, we have $S[j] = S'[j']$. The alignment graph of collection $\mathcal{S}$ with respect to $\sim$ is a vertex-labeled graph $G^\sim = (V, E, \Sigma, \ell)$ such that:*

1. *The set of vertices V is the set of equivalence classes for relation $\sim$.*
2. *For any two vertices $u, v \in V$, we have $(u, v) \in E$ if and only if there is a string $S \in \mathcal{S}$ and a position j such that $S[j:] \in u$ and $S[j+1:] \in v$.*
3. *The label of vertex $v \in V$ is the shared first symbol $\ell(v) = S[j]$, where $S[j:] \in v$.*

Note that the set of labels $\ell(P)$ for paths P in the alignment graph is the set of substrings of strings $S \in \mathcal{S}^\sim$. If $\sim$ is an equivalence relation based on a multiple sequence alignment such that $S[j:] \sim S'[j']:$ when the corresponding positions match, the resulting alignment graph is always acyclic.

3 XBWT: Indexing Labeled Trees

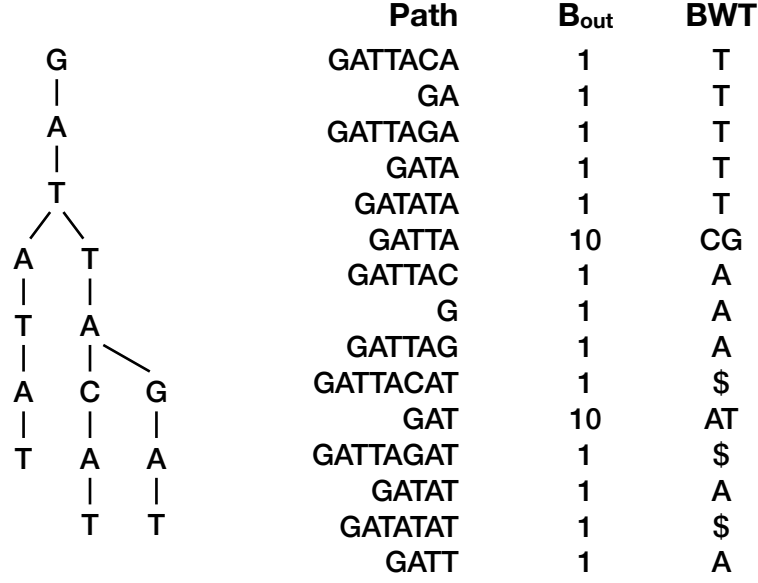
The Burrows–Wheeler transform was originally defined for a single string. Extending the definition to an ordered collection of strings is straightforward enough. But a collection of strings is still a linear structure, while paths in a graph can both diverge and converge. The first step towards using the BWT for graph indexing was the *extended Burrows–Wheeler transform (XBWT)* [21], which can index labeled trees. The key technique was using select queries on a bitvector for expanding a (co-)lexicographic range $[sp : ep]$ of vertices to a (co-)lexicographic range $[sp_{out} : ep_{out}]$ of outgoing edges.

In this section, we present a variant of the XBWT that is similar to the indexes for vertex-labeled graphs discussed in Sections 4 and 5. There are other fundamentally equivalent representations. The representations differ in their handling of leaf vertices and outdegrees, on whether they consider vertex-labeled or edge-labeled trees, and on whether the search proceeds from the root towards the leaves or the other way around. Some of them achieve better space bounds than what is presented here.

3.1 Data structure

A *vertex-labeled tree* with root $v_r \in V$ is a vertex-labeled graph $T = (V, E, \Sigma, \ell)$ such that the indegree of the root is 0 and the indegree of every other vertex $v \in V$ is 1, with a unique path P_v from the root to every vertex $v \in V$. Given an edge $(u, v) \in E$, we call vertex u the parent of vertex v and v a child of u . We call vertices with outdegree 0 the leaves of the tree.

XBWT uses vertices in place of suffixes. The vertices are sorted by the label $\ell(P_v)$ in colexicographic order (in lexicographic order by reverse path labels). For a given vertex $u \in V$, the BWT stores the concatenation of the labels $\ell(v)$ for each outgoing edge $(u, v) \in E$. We encode the outdegree d in unary as a one followed by $d - 1$ zeros and concatenate the outdegrees in the same order to form bitvector B_{out} . In order to handle the leaves, we add a technical vertex $v_\$ \notin V$ such that $\ell(v_\$) = \$$. For each leaf v , we encode an edge $(v, v_\$) \notin E$. As with strings, we assume that $\ell(v) \neq \$$ for every vertex $v \in V$. If



■ **Figure 1** XBWT for the trie of strings GATATAT, GATTACAT, and GATTAGAT. The XBWT consists of the bitvector B_{out} and the string BWT.

the colexicographic rank of vertex $v \in V$ is x , the labels of its children are stored in $BWT[B_{out}.select(x, 1) : B_{out}.select(x + 1, 1) - 1]$. See Figure 1 for an example. With an uncompressed wavelet tree [29] for the BWT and an uncompressed bitvector for B_{out} , the XBWT data structure requires $O(|V| \log \sigma)$ bits of space, not including the C array.

Let $find(X) = [sp : ep]$ be the colexicographic range of vertices $v \in V$ with path labels $\ell(P_v)$ ending with string X . Given a symbol $c \in \Sigma$, we want to find the range $find(Xc)$ of vertices with path labels ending with Xc . We first expand the range into a colexicographic range of outgoing edges $[sp_{out} : ep_{out}] = [B_{out}.select(sp, 1) : B_{out}.select(ep + 1, 1) - 1]$. Every edge $(u, v) \in E$ in that range is encoded with the label $\ell(v)$ of the child vertex, and the colexicographic rank of that vertex is based on path label $\ell(P_v) = \ell(P_u)\ell(v)$. Hence we can use a backward search step (Section 2.4) to get the range of children with label c : $find(Xc) = [LF(sp_{out} - 1, c) + 1 : LF(ep_{out}, c)]$. Note that the backward search step becomes a forward search step, as the XBWT is based on colexicographic order. Given a string X , we can therefore compute $find(X)$ using $O(|X|)$ rank operations on BWT and $O(|X|)$ select operations on B_{out} .

If we know the colexicographic rank x_r of the root v_r , we can match entire path labels instead of suffixes by starting the above search from range $[x_r : x_r]$. We can also use the XBWT for tree navigation. Given a vertex $v \in V$ with colexicographic rank x , the range of its children is $[B_{out}.select(x, 1) : B_{out}.select(x + 1, 1) - 1]$. The colexicographic rank of its i -th child is $LF(B_{out}.select(x, 1) + i - 1)$ and the rank of its parent is $B_{out}.rank(\Psi(x), 1)$.

3.2 Applications

The initial XBWT paper proposed using the structure as a fast and space-efficient representation for hierarchical documents, such as XML [21]. In such applications, it is important that a vertex can have multiple children with the same label (such as list items). When storing string collections (often called string dictionaries in this context), we can collapse the tree into

a trie, improving space usage. Compared to other representations for string collections, the XBWT is smaller but slower [36]. The slowness comes from symbol-to-symbol navigation and the poor memory locality of iterated LF-mapping. However, the same FM-index functionality enables substring searches not supported by faster representations based on hashing or front coding.

The Aho–Corasick algorithm can find the occurrences of multiple patterns $\mathcal{P}$ in a string S in linear time $O(|S| + \|\mathcal{P}\| + occ)$, where occ is the number of occurrences. It is based on a trie $T = (V, E, \Sigma, \ell)$ of the patterns augmented with failure links (suffix links). A failure link from vertex $v \in V$ points to a vertex $u \in V$ such that $\ell(P_u)$ is the longest proper suffix of $\ell(P_v)$ among all vertices $u \in V$. The XBWT can be augmented with a $2|V|(1 + o(1))$ -bit balanced parentheses representation to support failure links, making it a space-efficient Aho–Corasick automaton [35]. Here the relative slowness matters less, as the algorithm is based on symbol-to-symbol navigation anyway.

Because the XBWT represents a collection of strings, it can be built from other representations, such as the suffix array and the Burrows–Wheeler transform [35]. The link also works in the other direction: we can go from the XBWT directly to a run-length encoded multi-string BWT [12]. As a byproduct, we can efficiently find the order of the strings that minimizes the number of runs in the BWT [12]. Because the order of the strings determines how well the suffixes close to the end of the string can be run-length encoded, this can be valuable with highly repetitive collections of short strings.

4 SBWT: Indexing k -Spectra

The k -spectrum of a string S is the set of distinct k -mers occurring in S . In this section we give an overview of how k -spectra can be indexed for k -mer lookup queries via the Spectral Burrows–Wheeler transform (SBWT) [5]. Given a query k -mer, a k -mer lookup query returns the colexicographic rank of the k -mer in the k -spectrum. It is not difficult to see that a structure for k -mer lookup queries gives a convenient representation of a de Bruijn graph.

The SBWT is a distillation of the ideas found in the BOSS [10] and Wheeler graph data structures. It is indeed an evolution of the BOSS data structure, which is an indexed representation of the edge-centric de Bruijn graph. The SBWT differs from the BOSS being vertex-centric, and more general – the SBWT is a mathematical transformation which can be indexed in various ways, one of which closely resembles the BOSS data structure. In addition, the SBWT graph is also a Wheeler graph [25] covered in the next chapter of this book. This implies that the generic Wheeler graph index could be used to index the graph. However, that index requires storage of the sequence of indegrees and outdegrees of the vertices in the graph, whereas in the SBWT this is not required because every vertex apart from the vertex of the empty string has indegree of exactly 1, and the sequence of outdegrees is already included in the sizes of the sets of the outgoing edge label sets. Thus, the SBWT can be seen as a specialization of the Wheeler graph framework to k -spectra, taking advantage of the properties of the special case.

4.1 The Spectral Burrows–Wheeler Transform

In this section we define the Spectral Burrows–Wheeler transform, SBWT. We begin with few basic definitions:

► **Definition 4** (k -spectrum). *A string’s k -spectrum, denoted with $S_k(S)$, consists of all unique k -mers present in the string: $\{S[i : i + k - 1] \mid i = 1, \dots, |S| - k + 1\}$. The k -spectrum $S_k(S_1, \dots, S_m)$ of a set of strings $S_1, \dots, S_m$ is the union $\bigcup_{i=1}^m S_k(S_i)$.*

padded k -spectrum adds a minimal set of $\$$ -padded *dummy k -mers* to ensure that in the de Bruijn graph (to be defined shortly), every non-dummy k -mer has an incoming path of length at least k :

► **Definition 5** (Source vertex set). *Let $R = S_k(S_1, \dots, S_m)$ be a k -spectrum, the source vertex set R' is a subset of R containing k -mers Y such that $Y[1 : k - 1]$ is not a suffix of any k -mer in R .*

► **Definition 6** (Padded k -spectrum). *Let R be a k -spectrum of $\mathcal{S} = (S_1, \dots, S_m)$, and let $R' \subseteq R$ be the source vertex set of $\mathcal{S}$ for a given k . The padded k -spectrum is the set $S_k^+(S_1, \dots, S_m) = R \cup \{\$^k\} \cup \bigcup_{Y \in R'} \{\$^{k-i} Y[1 : i] \mid i = 1, \dots, k - 1\}$, where $\$$ is a special character of Σ , which is smaller than all other characters of the alphabet.*

The number of dummy vertices can be calculated with a simple formula. It is indeed equal to the number of internal vertices of $\mathcal{T}(R')$, where $\mathcal{T}(R')$ is the trie of the source vertex set of a set of strings.

For example, if we consider two strings $S_1 = \text{ACAGTG}$ and $S_2 = \text{ATCAGA}$, and $k = 3$, then $R = S_3(S_1, S_2) = \{\text{ACA}, \text{CAG}, \text{AGT}, \text{GTG}, \text{ATC}, \text{TCA}, \text{AGA}\}$, $R' = \{\text{ACA}, \text{ATC}\}$, and the padded k -spectrum $S_3^+(S_1, S_2) = S_3(S_1, S_2) \cup \{\$\$\$, \$\$A, \$AC, \$AT\}$.

► **Definition 7** (de Bruijn graph). *The de Bruijn graph of a set of strings $\mathcal{S} = (S_1, \dots, S_m)$ is an edge-labeled graph $G = (V, E, \Sigma, \ell, k)$, where:*

1. *The set of vertices V is the set of k -mers from the padded k -spectrum of $\mathcal{S}$, in symbols $V = S_k^+(S_1, \dots, S_m)$.*
2. *For any two vertices, k -mers, $u, v \in V$, we have $(u, v) \in E$ if and only if $u[2 : k] = v[1 : k - 1]$.*
3. *The label of an edge $e = (u, v) \in E$, denoted with $\ell(e)$ or $\ell(u, v)$, is the character $v[k]$. $\$$ never appears as an edge label.*

See figure 2 for an illustration of a de Bruijn graph.

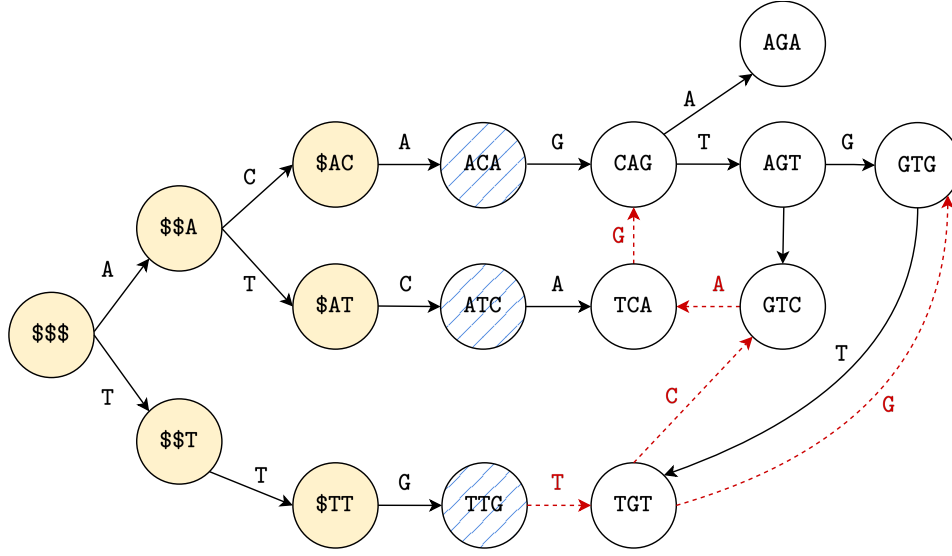
We can now define the Spectral BWT.

► **Definition 8** (Spectral Burrows-Wheeler transform (SBWT) [5]). *Let R^+ be a padded k -spectrum and let $X_1, \dots, X_{|R^+|}$ be the elements of R^+ in colexicographic order. The SBWT is the sequence of sets $A_1, \dots, A_{|R^+|}$ with $A_i \subseteq \Sigma$ such that $A_i = \emptyset$ if $i > 1$ and $X_i[2 : k] = X_{i-1}[2 : k]$, otherwise $A_i = \{c \in \Sigma \mid X_i[2k]c \in R^+\}$.*

Continuing the example above and adding a new sequence:

$\mathcal{S} = \{\text{ACAGTG}, \text{ATCAGA}, \text{TTGTCAGTGT}\}$. The colexicographically ordered list of $S_3^+(\mathcal{S})$ is: $\$\$\$, \$\$A, \text{ACA}, \text{TCA}, \text{AGA}, \$AC, \text{ATC}, \text{GTC}, \text{CAG}, \text{GTG}, \text{TTG}, \$\$T, \$AT, \text{AGT}, \text{TGT}, \$TT$, and the SBWT of $\mathcal{S}$ is the sequence of sets: $\{\text{A}, \text{T}\}, \{\text{C}, \text{T}\}, \{\text{G}\}, \emptyset, \emptyset, \{\text{A}\}, \{\text{A}\}, \emptyset, \{\text{A}, \text{T}\}, \{\text{T}\}, \emptyset, \{\text{T}\}, \{\text{C}\}, \{\text{C}, \text{G}\}, \emptyset, \{\text{G}\}$.

The sets in the SBWT represent the labels of outgoing edges in the vertex-centric de Bruijn graph, such that we only include outgoing edges from k -mers that have a different suffix of length $k - 1$ than the preceeding k -mer in the colexicographically sorted list. The padding of dollar-symbols is a technical detail required to make the SBWT work. It ensures that every k -mer of the k -spectrum has an incoming path of length at least k in the de Bruijn graph. Alternatively, the sequences of the set $\mathcal{S}$ on which the SBWT is built could be made cyclic, thus removing the need for the padded k -spectrum $S_k^+(\mathcal{S})$.



■ **Figure 2** The de Bruijn graph of $S = \{ACAGTG, ATCAGA, TTGTCAGTGT\}$, $k = 3$. The *source vertex set*, k -mers with no predecessor in S , is striped, and the solid-colored vertices represent the dummy vertex set. Dashed edges are pruned from the graph as the vertices they point to can be reached from other (solid) edges.

We now describe how to implement efficient k -mer membership queries on the spectrum of the input strings, using only the information encoded in the SBWT. It is helpful to think of the SBWT as encoding the set R^+ of k -mers as a list arranged in colexicographical order (as in the example above). In the context of k -mer lookup, the SBWT allows navigation of this ordered list via a search routine based on *subset rank queries*.

► **Definition 9** (Subset rank query). Let $X_1, \dots, X_n$ be a sequence of subsets of an alphabet $\Sigma = \{1, \dots, \sigma\}$. A *subset rank query* takes as input an index i and a character $c \in \Sigma$, and returns the number of subsets X_j with $j \leq i$ such that $c \in X_j$.

The search routine, introduced below, works by searching the k -mer character by character from left to right. It does so maintaining the interval of k -mers in the colexicographically-sorted list that are suffixed by the prefix which has been processed so far.

Since the k -mers in R^+ are colexicographically sorted in the SBWT, the subset of k -mers in R^+ that share a string α as a suffix are next to each other, and they are called the *colexicographic interval* of α . The colexicographic interval of a string α longer than k is defined as the range of k -mers in the sorted list of R^+ suffixed by the last k characters of α . Now let $[s : e]_\alpha$ denote the *colexicographic interval* of the string α , where s and e are respectively the colexicographic ranks of the smallest and largest k -mer in the SBWT having substring α as a suffix. Finding the interval $[s' : e']$ of αc , where c is a character from $\Sigma - \{\$$, is equivalent to following all edges labeled with c from the vertices in $[s : e]$. Due to the way the edges are defined, the end points of these edges are a contiguous range $[s' : e']$ such that s' is the destination of the first outgoing edge labeled c from $[s : e]$, and e' is the destination of the last one. Let $C[c]$ be the number of edge labels with label smaller than c in the graph. We now have the following formulas:

$$\begin{aligned} s' &= 1 + C[c] + \text{subsetrank}_c(s - 1) + 1 \\ e' &= 1 + C[c] + \text{subsetrank}_c(e) \end{aligned} \tag{1}$$

\$\$\$	\$\$A	ACA	TCA	AGA	\$AC	ATC	GTC	CAG	GTG	TTG	\$\$T	\$AT	AGT	TGT	\$TT
1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1
1	1	0	0	0	0	0	0	1	1	0	1	0	0	0	0

■ **Figure 3** MatrixSBWT of $\mathcal{S} = \{\text{ACAGTG}, \text{ATCAGA}, \text{TTGTCAGTGT}\}$ with $k = 3$. The dashed lines indicate borders of suffix groups. Two adjacent columns are in the same group if they share the same suffix of length $k - 1$. Bits may be moved horizontally inside a suffix group without affecting the k -mer set encoded in the matrix.

where subsetrank_c is a subset rank query on the SBWT sequence. The “+1” at the start of the formulas is to skip over the vertex of $\$^k$. The values $C[c]$ can be precomputed for all characters $c \in \Sigma$. By iterating these formulas k times, we have the k -mer search routine.

The simplest data structures for subset rank queries is the plain matrix. See Figure 3 for an example.

► **Definition 10** (Plain Matrix representation). *The plain matrix representation of the SBWT sequence is a $|\Sigma - \{\$\}| \times n$ binary matrix M , such that $M[i][j] = 1$ iff subset X_j includes the i^{th} character in the alphabet.*

The rows of the matrix M are indexed for constant time rank queries [37] (implementation from [28]). A single rank query on row i at index j in M would answer in constant time $\text{subsetrank}_c(j)$, where c is the i^{th} character of the alphabet. There exist alternative succinct data structures for subset rank, such as the *subset wavelet tree*, details of which can be found in [4, 5]. A concatenation of the sets indexed as a wavelet tree leads to something resembling the BOSS data structure. All k -mers of a longer string can be searched efficiently by adding the LCS array of the SBWT [3]. At the end of each single k -mer search, the LCS array allows to extend the colexicographic interval to the last $k - 1$ characters observed in the previous k -mer.

5 GCSA: Indexing Finite Languages

The *generalized compressed suffix array (GCSA)* [48, 49] is a generalization of the FM-index to graphs. It was originally designed for indexing acyclic vertex-labeled graphs (or equivalently finite languages) that arise from aligned DNA sequences, using techniques similar to the XBWT (Section 3).

GCSA can be built for any such graph, and it supports efficient queries in it, but the index can be exponentially larger than the graph. Some other BWT-based graph indexes, such as BWBBLE [32] and vBWT [34], take the opposite approach. By encoding the graph as a string, they make index construction fast and the index itself small. Queries can be slower than in GCSA, as they often require branching in the graph.

BOSS [10] represents de Bruijn graphs using a data structure equivalent to the GCSA. This equivalence led to a realization that the GCSA approach works with any graph that is sufficiently similar to a de Bruijn graph. Such graphs can recognize a strict subclass of regular languages [49]. GCSA2 [46] uses this realization in a practical way to index any vertex-labeled graph by approximating it with a high-order (pruned) de Bruijn graph.

GCSA requires that the vertices of the graph can be sorted unambiguously by the labels of maximal paths starting from them. This is a global property that is sometimes difficult to reason about. Wheeler graphs [25] replaced this requirement with an essentially equivalent local property. Vertices are sorted by their labels, and ties are broken by inheriting the order of the successor vertices. (The original formulation used edge-labeled graphs and inherited the order from the predecessors.) This alternate characterization made reasoning about Wheeler graphs easier in many situations.

5.1 Structural runs in the BWT

Consider a collection that consists of m identical copies of the same string S . Then for any position j , the suffixes starting at that position are consecutive in lexicographic order. Because the preceding symbol $S[j-1]$ (if $j > 1$; otherwise $S[n] = \$$) is also the same for every string, the suffixes are in the same run of symbol $S[j-1]$ in the BWT of the collection. Hence the BWT of a collection of m identical strings always has the same number of runs, regardless of the number of copies. We now consider what happens when we apply edits to a collection of identical strings.

► **Definition 11** (Aligned suffixes). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings, and let MSA be a multiple sequence alignment of the collection. For any two strings $S_i, S_{i'} \in \mathcal{S}$ and positions j, j' in them, suffixes $S_i[j:]$ and $S_{i'}[j':]$ are aligned if and only if positions $S_i[j]$ and $S_{i'}[j']$ are a match in MSA. We represent this with equivalence relation $\sim_0$*

► **Definition 12** (Structural BWT runs). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings, let $\sim_0$ be an alignment of the suffixes, and let BWT and SA be the Burrows–Wheeler transform and the suffix array of the collection. A structural BWT run is a maximal interval $[sp : ep]$ such that for any $x, x' \in [sp : ep]$:*

1. $S_i[j:] \sim_0 S_{i'}[j':]$, where $SA[x] = (i, j)$ and $SA[x'] = (i', j')$; and
2. $BWT[x] = BWT[x']$.

We use equivalence relation $\sim_0^b$ to denote that suffixes are in the same structural BWT run.

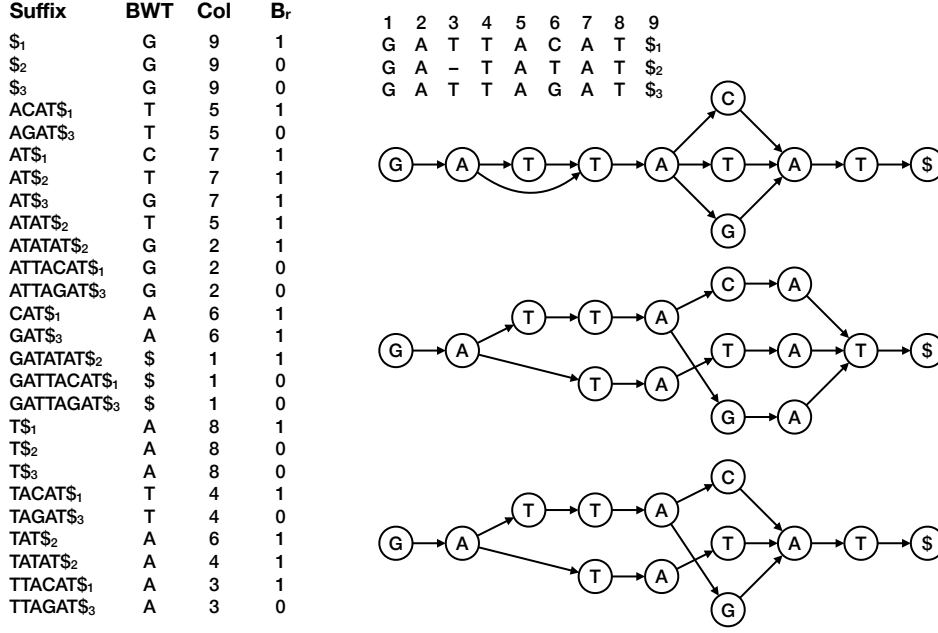
Let $\mathcal{S} = (S_1, \dots, S_m)$ be a *highly repetitive* collection of similar strings. We assume that there is a sequence of s edit operations that transforms a collection of m copies of string S into $\mathcal{S}$, with $s \ll mn$. We further assume that there is a multiple sequence alignment corresponding to the edit operations, where suffixes that have been derived from the same suffix of the original string S start with a match or a mismatch in the same column. Note that different sequences of edit operations leading to the same collection may correspond to different alignments.

Initially, there are n structural runs, one for each position in string S . Each edit operation that changes a substring $S_i[j : j']$ can affect the suffixes of S_i in three ways:

1. The symbol in the BWT can change for the suffixes starting from $S_i[j+1]$ to $S_i[j'+1]$.
2. Suffixes starting from $S_i[j-k]$ to $S_i[j']$, for some $k \geq 0$, may leave their current structural runs due to changes in lexicographic ranks.
3. New unaligned suffixes can be created and existing suffixes may be removed if substring $S_i[j : j']$ is replaced with a string of different length.

Every affected suffix can increase the number of structural runs by at most two: by becoming a new run and by splitting an existing run in two.

Mäkinen et al. [38] showed that if the original string S is random and we apply random edits, k is at most $O(\log_\sigma N)$ in the expected case. If we assume single-symbol edits, s edit operations increase the number of structural runs by at most $O(s \log_\sigma N)$ in the expected



■ **Figure 4** Structural runs in the BWT of strings $S_1 = \text{GATTACAT}\$, S_2 = \text{GATATAT}\$, and $S_3 = \text{GATTAGAT}\$$. The table lists the suffixes in lexicographic order, the BWT symbol for each suffix, and the starting position of the suffix in the MSA (also shown). Bitvector B_r marks positions at the start of each structural BWT run. The graphs from top to bottom are: $G^{\sim 0}$ representing the MSA; $G^{\sim b}$ representing structural BWT runs; and $G^{\sim s}$ representing structural SA runs (see Section 5.4).$

case. Because structural runs are (possibly non-maximal) BWT runs, there are at most $r + O(s \log_\sigma N)$ runs in the BWT of collection $\mathcal{S}$ in the expected case. Here r is the number of runs in the BWT of string S .

5.2 Indexing the graph of structural runs

Let $\mathcal{S}$ be a string collection and $\sim_0^b$ an equivalence relation that denotes that two suffixes are in the same structural BWT run. Let $G^{\sim b} = (V, E, \Sigma, \ell)$ be the alignment graph of the collection with respect to $\sim_0^b$.

Graphs like this have been used as space-efficient representations of the collection. Both SimDNA [33] and FM-index of alignment [40, 39] index the collection using a graph based on a specific type of alignment. The alignment alternates between common regions, where all strings are aligned, and non-common regions, where they diverge. BWT tunneling [7] also uses a similar graph to improve BWT-based data compression by not storing run lengths within unary paths. Our goal is instead to extend the FM-index of the collection to index all strings in the closure $\mathcal{S}^{\sim b}$ using the graph.

We could augment the FM-index with a bitvector $B_r[1 : N]$ that marks the start (or equivalently the end) of each structural run with ones. After each backward search step, we use some rank/select operations on B_r to extend the interval $[sp : ep]$ to the smallest interval $[sp' : ep']$ that contains it and consists of entire structural runs. If we continue the search from interval $[sp' : ep']$, we are effectively accepting recombinations within each vertex in the range. No matter which suffixes we used to reach a vertex, we can continue the search to the predecessor of any suffix in the vertex. See Figure 4 for an example.

5.3 GCSA structure

Because we discard the information on how we arrived to the current vertex, we no longer need to keep track of the exact BWT interval. Hence we choose the GCSA representation [48] (Figure 5) that corresponds more closely to the structure of the graph. We keep the vertices in the same order as the structural runs. Let $\text{rank}_V : V \rightarrow [1 : |V|]$ be a function that maps a vertex $v \in V$ to its lexicographic rank $\text{rank}_V(v)$. Lexicographic range $[sp : ep]$ now corresponds to vertices $v \in V$ such that $\text{rank}_V(v) \in [sp : ep]$.

The data structure consists of three components B_{in} , BWT, and B_{out} . We use them all for computing a backward search step from $\text{find}(X) = [sp : ep]$ to $\text{find}(cX) = [sp' : ep']$ for a string X and a symbol c .

1. Bitvector B_{in} encodes the indegree of each vertex in unary. Similar to using B_{out} in XBWT (Section 3.1), we map the range of vertices to a range of incoming edges as $[sp_{in} : ep_{in}] = [B_{in}.\text{select}(sp, 1) : B_{in}.\text{select}(ep + 1, 1) - 1]$.
2. The BWT range for vertex $v \in V$ is the concatenation of $\ell(u)$ for each vertex $u \in V$ with $(u, v) \in E$ in an arbitrary order. A backward search step with this BWT maps the range $[sp_{in} : ep_{in}]$ of incoming edges to a range $[sp_{out} : ep_{out}] = [\text{LF}(sp_{in} - 1, c) + 1 : \text{LF}(ep_{in}, c)]$ of outgoing edges, ordered by the predecessor vertex.
3. Bitvector B_{out} encodes the outdegree of each vertex in unary. With rank queries, we get the interval of the predecessors of the vertices in $[sp : ep]$ with label c as $[sp' : ep'] = [B_{out}.\text{rank}(sp_{out}, 1) : B_{out}.\text{rank}(ep_{out}, 1)]$.

Together these take $|E|(\log \sigma + 2)(1 + o(1))$ bits, assuming an uncompressed wavelet tree for the BWT and uncompressed bitvectors and excluding the C array. To handle the edge cases with source/sink vertices, we add technical edges connecting vertices with outdegree 0 to vertices with indegree 0.

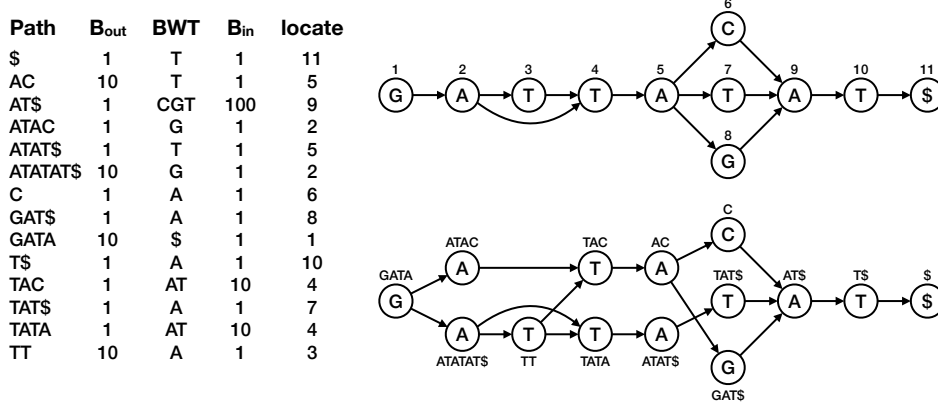
This data structure also facilitates graph navigation. For instance, if $v \in V$ is a vertex and $x = \text{rank}_V(v)$, we can navigate backward along a unary path from v . This is effectively the same as using $\text{LF}(x)$ to navigate from a suffix to its predecessor. The rank of the first incoming edge to v is $x_{in} = B_{in}.\text{select}(x, 1)$. If $B_{in}[x_{in} + 1] = 0$, vertex v has multiple predecessors and the path is no longer unary. With $x_{out} = \text{LF}(x_{in})$, we get the rank of the corresponding outgoing edge, and $B_{out}.\text{rank}(x_{out}, 1)$ is the rank of the predecessor vertex. Navigation like this is used in locate queries (Section 5.5).

The alignment graph based on structural runs allowed us to index the original strings as well as their recombinations within the runs. Our ultimate goal is allowing recombinations at any aligned position. In principle, we already have the necessary tools for it. Because $\sim_0^b$ is a refinement of $\sim_0$, we have $\mathcal{S}^{\sim_0} = (\mathcal{S}^{\sim_0})^{\sim_0^b}$. In other words, recombinations at aligned positions include recombinations at aligned positions within the same BWT run. We could therefore build the GCSA structure for the closure $\mathcal{S}^{\sim_0}$. The challenge is doing that without representing the closure explicitly.

5.4 GCSA construction

Before proceeding, we relax the definition of structural runs. The GCSA structure only requires that the equivalence classes of suffixes are lexicographic ranges that start with the same symbol. By not requiring that they also share the preceding symbol, we simplify the discussion and make the alignment graph $G^{\sim_0^s}$ smaller.

► **Definition 13** (Structural SA runs). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings, let $\sim_0$ be an alignment of the suffixes, and let SA be the suffix array of the collection. A structural SA run is a maximal interval $[sp : ep]$ such that for any $x, x' \in [sp : ep]$ with $\text{SA}[x] = (i, j)$ and $\text{SA}[x'] = (i', j')$, we have $S_i[j :] \sim_0 S_{i'}[j' :]$. We use equivalence relation $\sim_0^s$ to denote that suffixes are in the same structural SA run.*



■ **Figure 5** GCSA example for the string collection in Figure 4. We show the original graph (top) again with each vertex v annotated with $\text{id}(v)$ (see Section 5.5). The bottom graph is the final graph $G^{\sim s} = (V', E', \Sigma, \ell')$ for the closure, with each vertex $v' \in V'$ annotated with $\ell'(P_{v'})$. Paths $P_{v'}$ are chosen to be prefixes of the lexicographically smallest maximal path starting from the vertex. The table lists each vertex $v' \in V'$ identified by the label $\ell'(P_{v'})$, the corresponding parts of the GCSA structure, and the identifier $\text{id}(\text{locate}(v'))$ of the corresponding vertex in the original graph.

The algorithm we use for building GCSA [48, 49] is similar to the prefix-doubling approach to suffix array construction. We start from graph $G_1 = G^{\sim 0} = (V, E, \Sigma, \ell)$ and build a series of graphs $G_l = (V_l, E_l, \Sigma, \ell_l)$ for increasing path lengths l . If we consider the graphs finite automata, they all recognize the same language $\mathcal{S}^{\sim 0}$. There are two types of vertices $v \in V_l$:

1. The vertex corresponds to a path P_v of length $|P_v| = l$ starting from vertex $\text{locate}(v) = P_v[1] \in V$ in graph $G^{\sim 0}$. For every maximal path P starting from vertex v in graph G_l , we require that $\ell(P_v)$ is a prefix of $\ell_l(P)$.
2. The vertex corresponds to a half-open lexicographic range $[X_v : Y_v)$ of path labels. There is an arbitrary path P_v of length $|P_v| \leq l$ starting from vertex $\text{locate}(v) = P_v[1] \in V$ in graph $G^{\sim 0}$ such that $\ell(P_v) \in [X_v : Y_v)$. For every maximal path P starting from vertex v in graph G_l , we require that $\ell_l(P) \in [X_v : Y_v)$. For every other vertex $u \in V_l$ with $u \neq v$, we require that $\ell(P_u) \notin [X_v : Y_v)$ (if u is of type 1) or $[X_u : Y_u) \cap [X_v : Y_v) = \emptyset$ (if u is of type 2).

In both cases, we set $\ell_l(v) = \ell(\text{locate}(v))$.

If all vertices $v \in V_l$ are of type 2, we have a graph G_l that is equivalent to $G^{\sim 0}$, with vertices that can be sorted unambiguously by the labels of maximal paths starting from them. For each suffix $S_i[j :]$ of the closure $\mathcal{S}^{\sim 0}$, there is a vertex $v \in V_l$ such that $S_i[j :] \in [X_v : Y_v)$ and a path P starting from v with label $\ell_l(P) = S_i[j :]$. Therefore the vertices of G_l correspond to (possibly non-maximal) structural runs. If we merge them to form maximal runs, we get the graph $G^{\sim s}$. See Figure 5 for an example.

We derive graph G_{2l} from graph G_l with a *doubling* step followed by a *pruning* step. Let $u \in V_l$ be a type 1 vertex, and let $\text{rank}_l(u)$ be the lexicographic rank of $\ell(P_u)$ in the set of distinct path labels $\{\ell(P_v) \mid v \in V_l\}$. In the doubling step, we combine it with all vertices $v \in V_l$ such that $P_u P_v$ is a valid path. For every combination, we create a new vertex w with $P_w = P_u P_v$ and $\text{rank}_{2l}(w) = (\text{rank}_l(u), \text{rank}_l(v))$. The new vertex will be of the same type as vertex v . This requires that graph $G^{\sim 0}$ is reverse deterministic; see below for a discussion. If $u \in V_l$ is a type 2 vertex, we copy it as vertex $w \in V_{2l}$ with $\text{rank}_{2l}(w) = (\text{rank}_l(u), 0)$. The doubling step finishes by sorting the new vertices by their ranks and replacing the pairs of ranks with integer values.

In the pruning step, we merge lexicographic ranges of type 1 vertices into type 2 vertices. Let $V_{2l}^x = \{v \in V_{2l} \mid \text{rank}_{2l}(v) = x\}$ be the set of vertices with rank x . If all vertices $v \in V_{2l}^x$ share the same $\text{locate}(v)$ vertex, the shared path label uniquely defines the starting vertex. We can therefore merge them into a type 2 vertex. Similarly, if sets V_{2l}^x and V_{2l}^{x+1} also share the starting vertex, we can merge them into a single vertex.

If $S_{\max}$ is the longest string in $\mathcal{S}^{\circ}$, the prefix-doubling algorithm finishes after at most $\lceil \log |S_{\max}| \rceil$ doubling steps. In the worst case, the final graph $G^{\sim^{\circ}}$ can be exponentially larger than the graph G° we started from. However, if the distances between the positions of the original string S with edits in any string $S' \in \mathcal{S}$ are sufficiently high, graph $G^{\sim^{\circ}}$ will have $O(|V|)$ vertices and $O(|E|)$ edges in the expected case [49]. If we assume a random string and random edits, the phase transition to exponential growth occurs at average distance $O(\log_{\sigma} |S|)$.

We do not need to maintain the edges of the intermediate graphs. Determining them for the final graph $G^{\sim^{\circ}} = (V', E', \Sigma, \ell')$ is simple enough. Given a vertex $v' \in V'$ of the final graph and an edge $(u, \text{locate}(v')) \in E$ of the original graph, we know that there must be a corresponding edge $(u', v') \in E'$ such that $\text{locate}(u') = u$ in the final graph. We can therefore generate the edges as (u, v') and sort them by $(\ell(u), \ell'(P_{v'}))$. This also sorts them by $\ell'(P_{u'})$. By scanning the lists of vertices and edges in sorted order, we can then match edges (u, v') to source vertices u' by $\text{locate}(u') = u$.

This construction algorithm only works if the original graph G° is *reverse deterministic*. For every vertex $v \in V$, we require that each predecessor $u \in V$ with $(u, v) \in E$ has a distinct label $\ell(u)$. If two predecessors $u, u' \in V$ share a label, $\ell(uP) = \ell(u'P)$ for every path P starting from vertex v . The pruning step cannot merge them, because no extension of the path label can determine the starting vertex uniquely. On the other hand, if the graph is reverse deterministic and $v \in V$ is a type 2 vertex, $\ell(uP_v)$ always determines the unique predecessor u .

Determinizing a finite automaton takes exponential time in the worst case, because the states of the determinized automaton correspond to sets of states of the non-deterministic automaton. However, when the textbook algorithm was used to determinize the graph before GCSA construction, it was observed to be fast [49]. This was later explained using Wheeler graphs [16]. The exponent in the time complexity is not the total number of states but the maximal number of states that are not comparable in lexicographic order. Intuitively, the closer the graph is to a Wheeler graph, the faster the determinization is. If the graph is simple enough that the prefix-doubling algorithm only increases its size by a constant factor, it must already be close to being a Wheeler graph.

5.5 Locating the hits

Let $G^{\circ} = (V, E, \Sigma, \ell)$ be the original alignment graph and $G^{\sim^{\circ}} = (V', E', \Sigma, \ell')$ the final graph in GCSA construction. During the construction, we annotated each vertex $v \in V_l$ in each intermediate graph $G_l = (V_l, E_l, \Sigma, \ell_l)$ (including G° and $G^{\sim^{\circ}}$) with the corresponding vertex $\text{locate}(v) \in V$ of the original graph. As the notation implies, we want GCSA locate queries return positions in the original graph. While it would be easier to return positions in $G^{\sim^{\circ}}$ (we built GCSA for that graph, after all), it is only a technical construct. Positions in the original graph (or columns in the alignment) are more likely to be meaningful to the user.

A locate query in an FM-index relies on the fact that $\text{SA}[x] = \text{SA}[\text{LF}(x)] + 1$. If a position has not been sampled, we can derive the value from the predecessor. Let $\text{id} : V \rightarrow [1 : |V|]$ be a function that gives each vertex $v \in V$ a unique integer identifier. To apply the same idea with GCSA, we require that $\text{id}(v) = \text{id}(u) + 1$ for each edge $(u, v) \in E$ such that u is

the only predecessor of v and v is the only successor of u . If we sample $\text{id}(\text{locate}(v))$ for each vertex $v \in V'$ that has multiple predecessors or $\text{id}(\text{locate}(v)) \neq \text{id}(\text{locate}(u)) + 1$ for the only predecessor u , the locate algorithm works as before [48, 49]. (We also want to sample $\text{id}(\text{locate}(v))$ at regular intervals on unary paths to ensure query performance.)

This approach was originally designed for sampling the LCP array [45]. If $\text{BWT}[x]$ is not at the start of a run, $\text{LCP}[x] = \text{LCP}[\text{LF}(x)] - 1$. We therefore need to sample the irreducible values, while the rest can be derived from the predecessor. More generally, the approach can be understood as a sampled tag array [8]. We annotate each position in the FM-index with a tag, which can store arbitrary information related to the position. If most tags can be derived easily from the only predecessor, we can store the tag array space-efficiently and still support efficient queries.

5.6 Simplifying the graph

As mentioned earlier, there is a phase transition in GCSA construction. As the distances between positions with edits in the alignment get below a critical threshold, the number of vertices and edges in the final graph $G^{\sim^s}$ starts growing rapidly. This makes the GCSA impossible to build in practice for many graphs. One way to handle this is adding a context length requirement to the definition of alignment [48, 49].

► **Definition 14** (*k*-aligned suffixes). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings, let MSA be a multiple sequence alignment for the collection, and let $k \geq 0$ be a context length. For any two strings $S_i, S_{i'} \in \mathcal{S}$ and positions j, j' in them, suffixes $S_i[j :]$ and $S_{i'}[j' :]$ are *k*-aligned if and only if positions $S_i[j]$ and $S_{i'}[j']$ are a match in MSA and $S_i[j : j + k] = S_{i'}[j' : j' + k]$. We represent this with equivalence relation $\sim_k$.*

We similarly define structural SA runs based on *k*-aligned suffixes and denote them with equivalence relation $\sim_k^s$. Then we start GCSA construction from graph $G^{\sim^k}$ and use prefix-doubling to build graph $G^{\sim_k^s}$. In practice, context length *k* imposes a minimum distance between recombinations. Regions that are heavily branching in $G^{\sim^0}$ gradually become simpler in $G^{\sim^k}$ as *k* increases. That makes it more likely that as path length *l* grows, path labels become unique at a faster rate than the number of type 1 vertices grows.

5.7 Simplified data structure

The final graph $G^{\sim_k^s} = (V, E, \Sigma, \ell)$ is always reverse deterministic. To see that, consider any two edges $(u, v) \in E$ and $(u', v) \in E$ such that $\ell(u) = \ell(u')$. Because *u* and *u'* are type 2 vertices, it must be that $\ell(uP) = \ell(u)\ell(P) \in [X_u : Y_u]$ and $\ell(u'P) = \ell(u')\ell(P) \in [X_{u'} : Y_{u'}]$, where *P* is a maximal path starting from *v*. As the lexicographic ranges for two type 2 vertices cannot overlap, we have $u = u'$.

For each vertex $v \in V$ and symbol $c \in \Sigma$, there is at most one vertex $u \in V$ such that $(u, v) \in E$ and $\ell(u) = c$. We can therefore replace the indegree bitvector B_{in} and the BWT with a binary matrix [49]. For each symbol $c \in \Sigma$, let B_c be a bitvector of length $|V|$, with $B_c[\text{rank}_V(v)] = 1$ whenever vertex $v \in V$ has a predecessor $u \in V$ with $\ell(u) = c$. The first two parts of a backward search step become $[sp_{out}, ep_{out}] = [C[c] + B_c.\text{rank}(sp - 1, 1) + 1 : C[c] + B_c.\text{rank}(ep, 1)]$.

We replaced a select query on a bitvector and a rank query on a string with a single rank query on a bitvector. A backward search step now consists of four rank queries on bitvectors. Depending on the data structures used, this can be considerably faster than with the full structure. With uncompressed bitvectors, the size bound increases from $|E|(\log \sigma + 2)(1 + o(1))$

bits to $(\sigma|V| + |E|)(1 + o(1))$ bits, which can be significant with large alphabets. With Elias–Fano encoded bitvectors for B_c , we get $|E|(\log(\sigma|V|/|E|) + O(1))$ bits at the expense of somewhat slower queries. A practical hybrid approach [46] uses uncompressed bitvectors for common symbols and compressed bitvectors for rare ones.

SBWT [5] (Section 4.1) goes even further with a representation, where the rank queries on B_c and B_{out} can be done with a single rank query on a bitvector. This comes at the expense of some functionality. In particular, SBWT does not tell directly whether vertex $v \in V$ has a predecessor with symbol $c \in \Sigma$.

5.8 Graphs with cycles

BOSS [10] represents de Bruijn graphs with a data structure that is equivalent to the full GCSA structure (Section 5.3). While a de Bruijn graph often contains cycles, its vertices v are type 2 vertices, with $[X_v : Y_v) = \{S \in \Sigma^* \mid S_v \text{ is a prefix of } S\}$ as the interval. Here S_v is a string of length $|S_v| = k$ and k is the order of the de Bruijn graph. We could therefore use GCSA, including the more efficient simplified structure (Section 5.7), with de Bruijn graphs. The main difference is that while LF-mapping and backward search follow the edges in BOSS, they go against the direction of the edges in GCSA.

A natural question is which classes of graphs can be represented using GCSA, and what is the corresponding class of formal languages. Because vertex-labeled graphs can be understood as finite automata, the class of languages is a subclass of regular languages. Sirén et al. [49] showed that the class is a proper subclass: GCSA can represent the obvious automaton recognizing language $\{a, b\}^*$ but no automaton recognizing $\{a, b\}^* \cup \{a, c\}^*$. The characterization of type 2 vertices with lexicographic ranges suggests that the class of graphs admitting the GCSA representation is a generalization of de Bruijn graphs. The properties of this class were later investigated based on the Wheeler graph formalization [1, 2].

5.9 GCSA2

GCSA was designed for indexing pangenome graphs based on aligned DNA sequences. The construction algorithm assumes that the alignments can be represented as a multiple sequence alignment or an acyclic graph. Such models are appropriate for simple variants, such as substitutions, insertions, and deletions. Other forms of variation, such as rearrangements (same substrings appear in different order in different strings) and inversions (a substring is replaced with its reverse complement) can be represented better with cycles in the graph. Such graphs often do not have an equivalent Wheeler graph that can be indexed with GCSA.

GCSA2 [46] is a reimplement of GCSA for general pangenome graphs. Because the graph cannot be fully indexed, GCSA2 approximates it with an order- k de Bruijn graph. It matches patterns of length at most k correctly, while longer matches may be false positives combining partial matches in different parts of the original graph. The initial implementation used $k = 128$, which was chosen to make excessive growth of the number of vertices less likely. This was later increased to $k = 256$ to allow mapping perfectly matching 150 bp and 250 bp reads as a single hit [47].

Let $G = (V, E, \Sigma, \ell)$ be the original graph and $G_k = (V', E', \Sigma, \ell')$ be the graph after $\log k$ rounds of prefix-doubling (Section 5.4). We replace the final pruning step with a merging step that also merges all type 1 vertices $v \in V'$ sharing $\text{rank}_k(v)$, regardless of whether the $\text{locate}(v)$ vertices match. The resulting *pruned de Bruijn graph* is equivalent to an order- k de Bruijn graph but potentially much smaller. To handle some edge cases, we must ensure

that all type 2 vertices $v \in V'$ have intervals $[X_v : Y_v)$ defined by a shared prefix. We build GCSA2 for graph G_k . As a consequence of the final merge, $\text{locate}(v) \subseteq V$ is now a set of vertices in the original graph.

Pangenome graphs contain regions that are dense in variation. While GCSA2 construction stops prefix-doubling early to prevent the worst exponential growth, such regions must often be simplified to keep the size of the index manageable. Because the original GCSA considered the alignments and the graph computational artifacts, it could handle complex regions by changing the definition of alignment (Section 5.6). GCSA2 cannot do that, because the provided alignments are assumed to be biologically meaningful. Instead, a preprocessing step removes everything except the reference genome in regions where there are too many branching vertices in a short window [26]. There is also an option to avoid sequence loss by unfolding the original strings in the complex region while maintaining a mapping to the corresponding positions in the original graph. Variation is then replaced with a set of disjoint paths representing unaligned strings [47].

GCSA2 supports some additional functionality beyond that provided by FM-indexes. It builds a trie of path labels $\ell(P_v)$ for vertices $v \in V'$ and represents it using next/previous smaller value queries and range minimum queries over the LCP array [23]. The resulting suffix tree-like functionality is used for finding maximal exact matches between the pattern and the graph [41]. A similar approach was first used with variable-order de Bruijn graphs [9] and later generalized to Wheeler graphs [15]. There is also a data structure for reporting the number of distinct $\text{locate}(v)$ values for vertices $v \in V'$ in a range $[sp : ep]$. It is based on compressed versions [24] of Sadakane’s document counting structure [44].

6 Founder Graphs

In this section, we deal with graphs built from the segmentation of multiple sequence alignments (MSAs), which is a line of research originating from the study of *founder sequences* [50]. Similarly to the GCSA approach from Section 5, respecting the MSA positions (i.e. the columns) results in an acyclic graph spelling the input strings and their recombination. However, differently from GCSA, in these segmentation-induced graphs that we call founder graphs, the recombinations depend exclusively on the chosen segments.

Many approaches reviewed so far fix a k -mer length to avoid indexing exponentially many paths of a graph. While such fixed parameter yields practically efficient solutions, it is of great theoretical value to study more general approaches. Wheeler Graphs [25] have been proposed as such, but they appear too general; Wheeler Graph recognition is an NP-complete problem [27] and thus to use Wheeler graph machinery one first needs to limit to graph classes that somehow avoid this complexity. Founder graphs also have similar hardness issues: in Section 6.2 we present the negative results from [17] stating that matching in founder graphs is not easier than matching in general labeled graphs. However, this hardness can be avoided, as we present in Section 6.3 that suitable segmentations of MSAs respecting a specific uniqueness property induce graph classes that are easy to index for linear-time pattern matching. These graphs are not limited to a fixed k -mer length, but can handle node labels of arbitrary length (provided that they are unique) and arbitrary queries. After delving into more details, we continue discussing the connection to Wheeler Graphs in Sect. 6.3. We conclude this section with a simple extension of the uniqueness properties of de Bruijn graphs and founder graphs to overlap graphs.

6.1 Inducing graphs from multiple sequence alignments

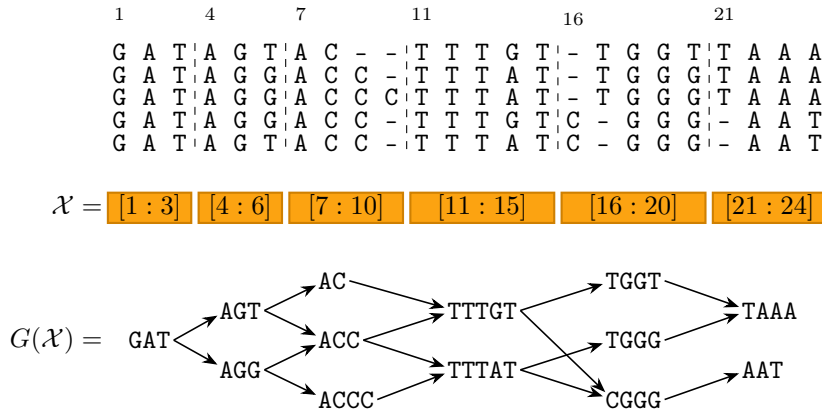
We define a *segmentation* $\mathcal{X}$ of $\text{MSA}[1 : m, 1 : n]$ as a partition of columns $[1 : n]$ in b segments $[1 = x_1 : y_1], [x_2 : y_2], \dots, [x_b : y_b = n]$ such that $y_j = x_{j+1} - 1$ for $j \in [1 : b - 1]$, and additionally we forbid any segmentation to spell the empty string in one of its segments, in symbols $\text{spell}(\text{MSA}[i, x_j : y_j]) \neq \varepsilon$ for all $i \in [1 : m], j \in [1 : b]$.

► **Definition 15** (elastic founder graph [20]). *Let $\text{MSA}[1 : m, 1 : n] \in (\Sigma \cup \{-\})^{m \times n}$ be a multiple sequence alignment. Given segmentation $\mathcal{X} = [x_1 : y_1], \dots, [x_b : y_b]$, the elastic founder graph (EFG) induced by $\mathcal{X}$ is a vertex-labeled graph $G = (V, E, \ell)$, where $\ell : V \rightarrow \Sigma^+$ assigns a non-empty label to every vertex, and the set of vertices V is partitioned into blocks $V_1, \dots, V_b$ such that $|\{\ell(v) : v \in V_k\}| = |V_k|$ (unique strings in a block) and for all $k \in [1 : b]$ we have that*

$$\{\ell(v) : v \in V_k\} = \{\text{spell}(\text{MSA}[i, x_k : y_k]) \mid 1 \leq i \leq m\}.$$

Moreover, it holds that $(v, w) \in E$ if and only if there exists $k \in [1 : b - 1]$ and $i \in [1 : m]$ such that $v \in V_k, w \in V_{k+1}$, and $\text{spell}(\text{MSA}[i, x_k : y_{k+1}]) = \ell(v) \cdot \ell(w)$.

If $E = \bigcup_{k=1}^{b-1} (V_k \times V_{k+1})$, that is all vertices in V_k are connected to all vertices in V_{k+1} for each $b \in [1..b]$, then the EFG is an elastic degenerate string (EDS) [30] with no empty string, and we can omit edge set E . Moreover, given an EFG, we define its EDS relaxation as the corresponding EDS over $V_1, \dots, V_b$.



■ **Figure 6** Example of $\text{MSA}[1 : 5, 1 : 24]$, segmentation $\mathcal{X}$, and elastic founder graph $G = (V, E, \ell)$ induced by $\mathcal{X}$.

See Figure 6. Given an elastic founder graph $G = (V, E, \ell)$, for each vertex $v \in V$ we denote as $\|v\|$ the vertex label length $|\ell(v)|$. We define height $H(G) = \max_{k=1}^b |V_k|$ as the maximum number of vertices in a block, and $L(G) = \max_{v \in V} \|v\|$ as its maximum vertex label length. We can extend ℓ to label paths in the graph: given a $u_1 u_k$ -path $P = u_1 \cdots u_k$, with $(u_i, u_{i+1}) \in E$ for $i \in [1..k - 1]$, we define $\ell(P) = \ell(u_1) \cdots \ell(u_k)$; for an edge $(u, v) \in E$, we call $\ell(uv)$ its *edge label*. Then, we say that some string $Q \in \Sigma^+$ occurs in G if there exists path P in G such that Q is a substring of $\ell(P)$. More specifically, we indicate with triple $(i, u_1 \cdots u_k, j)$ the *subpath* of P spelling Q starting from position $i \in [1 : \|u_1\|]$ in u_1 and ending in position $j \in [1 : \|u_k\|]$ in u_k , that is, $Q = \ell(u_1)[i : j] \ell(u_2 \cdots u_{k-1}) \ell(u_k)[1 : j]$. The *length* of subpath $(i, u_1 \cdots u_k, j)$ is then $|Q|$. We say that Q occurs *starting from the beginning* of u_1 if $i = 1$; Q occurs *starting inside* u_1 if $i \in [2 : \|u_1\|]$.

6.2 Hardness of matching EFGs

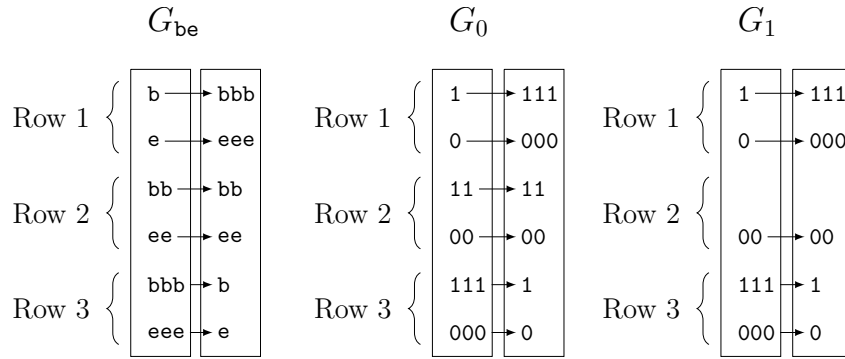
Starting from an MSA, we can construct EFG G . After this step, we would like to answer string matching queries for G , namely determining whether a given pattern string Q occurs in G . We will now show that, conditioned on some complexity hypotheses, this problem requires quadratic time complexity. This justifies the approach we will adopt in later sections, where we assume an additional property to hold in EFGs in order to break through this quadratic barrier.

The quadratic lower bound is obtained via fine-grained complexity techniques and is conditioned on the *Strong Exponential Time Hypothesis* (SETH) and the *Orthogonal Vectors Conjecture* (OVC). Since Williams [51] showed that SETH implies OVC, it suffices to show a subquadratic reduction from the *Orthogonal Vectors* (OV) problem to obtain this result. Let us first give a formal definition of OV.

► **Definition 16** (OV). *Let $X, Y \subseteq \{0, 1\}^d$ be two sets of n binary vectors of length d . Determine whether there exist $x \in X, y \in Y$ such that $x \cdot y = \sum_{i=1}^d x[i] \cdot y[i] = 0$.*

The notation $x[i] \cdot y[i]$ indicates the scalar product when used for two single entries of vectors x and y , while it refers to the dot product $x \cdot y$ when applied on the vectors themselves. Currently, it is conjectured that no algorithm can solve OV in truly subquadratic time.

► **Definition 17** (Orthogonal Vectors Conjecture (OVC)). *Let X, Y be the two sets of an OV instance, each containing n binary vectors of length d .¹ For any constant $\epsilon > 0$, no algorithm can solve OV in time $O(\text{poly}(d)n^{2-\epsilon})$, where poly is a polynomial function on d .*



■ **Figure 7** G_{be} , G_0 and G_1 . Each gadget is organized into three rows, each row encoding a different partitioning of the strings $bbbb$, $eeee$, 0000 , 1111 . This ensures that, when combining these gadgets into the final graph, edges can be controlled to go within the same row, or to the row below. Figure from [20].

Then, we can show that determining whether string Q occurs in graph G is at least as hard as solving OV, obtaining a quadratic conditional lower bound.

► **Theorem 18** ([20]). *For any constant $\epsilon > 0$, it is not possible to find a match for a query string Q in an EFG $G = (V, E, \ell)$ in either $O(|E|^{1-\epsilon} |Q|)$ or $O(|E| |Q|^{1-\epsilon})$ time, unless OVC fails. This holds even if restricted to an alphabet of size 4.*

¹ In this section, keeping in line with the usual notation in the OV problem, we use n to denote the size of X and Y , instead of the number of columns of the MSA.

We will provide an intuition on how to prove Theorem 18 by giving an idea of how to perform a subquadratic reduction from OV.

Starting from sets $X, Y \subseteq \{0, 1\}^d$ of n binary vectors of length d , the goal of the reduction is to build string Q and EFG G such that Q occurs in G if and only if there exist $x \in X, y \in Y$ such that $x \cdot y = 0$. Moreover, this reduction should be performed in not more than $O(n^{2-\epsilon} \text{poly}(d))$ time, for some constant $\epsilon > 0$.

Pattern. We start by constructing string Q from set of vectors X . We build Q by combining string gadgets $Q_1, \dots, Q_n$, one for each vector in X , plus some additional characters. For example, vector $x_i = 101$ results in string

$$Q_i = \text{bbbb } Q_{i,1} Q_{i,2} Q_{i,3} \text{eeee}, \text{ where } Q_{i,1} = 1111,$$

$$Q_{i,2} = 0000, \quad Q_{i,3} = 1111.$$

Full string Q is then the concatenation $Q = \text{bbbb}Q_1Q_2 \dots Q_n\text{eeee}$.

Graph. We build graph G combining together three different sub-graphs: G_L, G_M, G_R (for *left*, *middle* and *right*). Our final goal is to build a graph structured in three logical “rows”. We denote the three rows of G_M as G_{M1}, G_{M2}, G_{M3} , respectively. Sub-graphs G_L is able to match any prefix of pattern Q and is connected only to row G_{M1} , which can also match any prefix of Q . Symmetrically, sub-graphs G_R is able to match any suffix of pattern Q and it is connected only to row G_{M3} , which can also match any suffix of Q . Thus, the first and the third rows of G , along with subgraphs G_L and G_R (introduced to allow slack), can match any vector. The second row matches only sub-patterns encoding vectors that are orthogonal to the vectors of set Y . The key is to structure the graph such that the pattern is forced to utilize the second row to obtain a full match. We present the full structure of the graph in Figure 8, which shows the graph built on top of vector set $\{100, 011, 010\}$. In particular, G_M consists of n gadgets G_M^j , one for each vector $y_j \in Y$. The key elements of these sub-graphs are gadgets G_{be}, G_0 and G_1 (see Figure 7), which allow to stack together multiple instances of strings $b^4, e^4, 1^4, 0^4$. The overall structure mimics the one in [17, 19], except for the new idea from Figure 7 ([20]).

6.3 Indexable founder graphs

As we have seen, EFGs are not likely to support fast pattern matching without additional conditions. To alleviate this hardness, a uniqueness property has been introduced:

► **Definition 19** (semi-repeat-free indexability property [20]). *An EFG $G = (V, E, \ell)$ is semi-repeat-free or indexable (denoted as iEFG) if each subpath $(i, u_1 \dots u_p, j)$ spelling $\ell(v)$ for $v \in V_k$ is such that $i = 1$ and $u_1 \in V_k$, that is, $\ell(v)$ only occurs from the beginning of vertices in block V_k . The semi-repeat-free property is a generalization of the stronger repeat-free property, imposing that each $\ell(v)$ only occurs in G from the beginning of v .*

The simpler repeat-freeness condition requiring $\ell(v)$ to start only from the beginning of vertex v already enables fast pattern matching, but it is open if there is a fast algorithm to construct such EFG from the MSA [20].

Before proceeding to the construction of iEFGs, let us first gather some intuition why such graphs are useful. To fix ideas, assume the input MSA has no gaps —. Then semi-repeat-freeness matches with repeat-freeness, as the strings in each block are of the same length, and cannot be prefixes of one another. Consider searching for query string $Q[1 : m]$ in iEFG.

13:22 Graph Indexing Beyond Wheeler Graphs

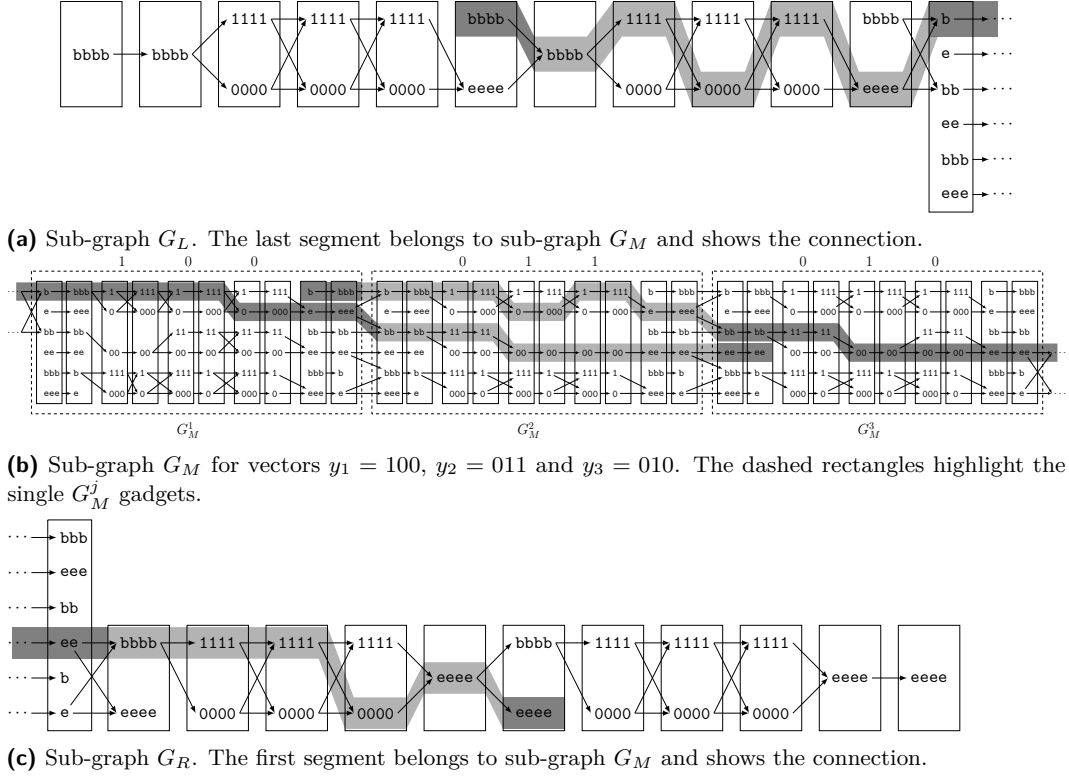


Figure 8 An example of graph G . To visualize the entire graph, watch the three sub-figures from top to bottom and from left to right. We also show two example occurrences of a query string Q constructed from $x_1 = 101$, $x_2 = 110$, $x_3 = 100$ (left-most), and from $x_1 = 101$, $x_2 = 100$, $x_3 = 110$ (right-most), respectively. The shade of gray changes from one Q_i to the next. Any such occurrence must pass through the middle row of G_M . Figure from [20].

An FM-index for the collection of edge labels (concatenation of adjacent vertex labels) can be used for checking if Q occurs inside a vertex or an edge. Longer occurrences must span at least one complete vertex label. One can build an Aho-Corasick automaton on the vertex labels to find all candidate occurrences. Consider vertex v with $\ell(v)$ occurring closest to the end of Q , say, at $Q[i : j]$. Consider a trie built on the set $F(v) = \{\ell(w) \mid (v, w) \in E\}$. We can check if $Q[j + 1..m]$ can be traced from the root of $F(v)$ to verify if $Q[i : j]$ occurs starting from the beginning of v . To verify if $Q[1 : i - 1]$ occurs ending at the end of vertex u s.t. $(u, v) \in E$, we can use tries $R(v) = \{\ell(u)^{-1} \mid (u, v) \in E\}$: We can scan $Q[1 : i - 1]$ from right to left in $R(v)$. If we reach a leaf, it corresponds to a unique vertex u due to the (semi-)repeat-free property. The scanning can continue from $R(u)$, until we have verified an occurrence for Q or detected that there was no occurrence starting from our candidate vertex v .

▷ **Claim 20.** If Q is found neither inside an edge label nor with the verification algorithm above, it has no occurrence in iEFG.

Proof. For contradiction, assume Q has an occurrence $(i', u_1 \dots u_k, j')$ in G . Since such occurrence is not found within an edge, it must span at least one vertex of G . Therefore the verification algorithm has started with some vertex v such that $Q[i : j] = \ell(v)$, but did not report this occurrence. That is, the substring $Q[i : j] = \ell(v)$ occurs starting somewhere else than in the beginning of v and thus our iEFG is not repeat-free. ◁

This tailored Aho-Corasick approach can be made more space-efficient by an *expanded backward search* mechanism [20] similar to one used for de Bruijn graphs in Section 4.1. Alternatively, one can reduce this case of repeat-free graphs induced from MSAs with no gaps to Wheeler Graphs [20].

It is important to note why none of the above approaches work if the MSA has gaps and our iEFG is semi-repeat-free: $\ell(v)$ is allowed to occur starting from another vertex v' from the same block. Due to this, straightforward extensions lead to false positive matches, and it is open if there is a reduction to Wheeler Graphs. However, there is a way to fix the Aho-Corasick approach: consider instead edges of the graph, as they have a uniqueness property analogous to the one above for vertices in the repeat-free case. One can basically repeat similar scanning of suffix and prefix of Q on indexes built on the set of edges, but there is a non-trivial detail in connecting the searches we omit here [43]:

► **Theorem 21** ([43]). *A semi-repeat-free EFG $G = (V, E, \ell)$ can be indexed in polynomial time into a data structure occupying $O(|D| \log |D|)$ bits of space, where $|D| = O(N \cdot H(G))$, $N = \sum_{v \in V} \|v\|$ is the total length of the vertex labels, and $H(G)$ is the height of G . Later, one can find out in $O(|Q|)$ time if a given query string Q occurs in G .*

Next, we will consider how to construct iEFGs. To fix ideas, we consider only the problem of maximizing the number of blocks in iEFG; there exist similar solutions for other objective functions like minimizing the maximum segment length and minimizing the maximum height of a block.

Recall from Definition 15 that an EFG is defined by a graph induced by a segmentation $\mathcal{X} = [x_1 : y_1], \dots, [x_b : y_b]$ of an MSA. We denote this graph $G(\mathcal{X})$.

► **Definition 22** ([20]). *Segment $[x : y]$ of $\text{MSA}[1 : m, 1 : n]$ is semi-repeat-free if for any $i, i' \in [1 : m]$ string $\text{spell}(\text{MSA}[i, x : y])$ occurs in gaps-removed row $\text{spell}(\text{MSA}[i', 1 : n])$ only at position $g(i', x)$, where $g(i', x)$ is equal to x minus the number of gaps in $\text{MSA}[i', 1 : x - 1]$.²*

► **Lemma 23** (Characterization of semi-repeat-free segments [20]). *The graph $G(\mathcal{X})$ induced from a segmentation $\mathcal{X}$ is semi-repeat-free if and only if all segments of $\mathcal{X}$ are semi-repeat-free.*

Consider you have found a semi-repeat-free segmentation until column x and want to extend it with a new semi-repeat-free segment into a longer segmentation. The following definition tells the minimum length of such extension.

► **Definition 24** (Minimal right extensions [20]). *Given $\text{MSA}[1 : m, 1 : n]$, for each $0 \leq x \leq n - 1$ we define value $f(x)$ as the smallest integer greater than x such that segment $[x + 1 : f(x)]$ is semi-repeat-free. If such integer does not exist, we define $f(x) = \infty$.*

To see how to compute these $f(x)$ values, let us for now assume our MSA is gapless. One can build a generalized suffix tree of the rows of the MSA. Consider the set of leaves of the suffix tree corresponding to suffixes of the rows starting at column $x + 1$. The goal is to find a set of suffix tree vertices that cover these leaves and only them. Among such sets, one should find the one that contains a vertex with the smallest string depth, as that will determine $f(x)$. This is called the *exclusive ancestor set* problem and can be solved in linear time in the number of leaves in the input set [43]. Updating the set of leaves when moving from column x to $x - 1$ can also be done in linear time, but with general MSAs, one needs to tailor the algorithm to take into gaps (e.g. in string depth), which is non-trivial but possible.

² The original definition considers mistakenly gaps in $\text{MSA}[i', 1 : x]$.

► **Theorem 25** ([43]). *Given $\text{MSA}[1 : m, 1 : n] \in \{\Sigma \cup \{-\}\}^{m \times n}$ with $\Sigma = [1 : \sigma]$ and $\sigma \leq mn$, we can compute the minimal right extensions $f(x)$ for $0 \leq x < n$ (Definition 24) in time $O(mn)$.*

► **Corollary 26** ([43]). *Given $\text{MSA}[1 : m, 1 : n] \in \{\Sigma \cup \{-\}\}^{m \times n}$ with $\Sigma = [1 : \sigma]$ and $\sigma \leq mn$, the construction of an optimal semi-repeat-free segmentation maximizing the number of blocks can be done in time $O(mn)$.*

Proof. Algorithm [20, Algorithm 1] by Equi et al. solves the problem in $O(n)$ time, assuming minimal right extensions given (Theorem 25). ◀

6.4 Repeat-free graphs: generalizing de Bruijn graphs with vertex uniqueness

We generalize iEFGs respecting the repeat-free property to overlap graphs respecting an equivalent and homonymous property. We show that graphs where long strings are constrained to have a unique starting location in the graph are a straightforward generalization of de Bruijn graphs, and are still indexable in polynomial time for linear-time pattern matching.

We define labeled graphs where each vertex is labeled with a non-empty string of variable length, and each directed edge specifies a suffix-prefix overlap between the connected vertices, as follows.

► **Definition 27** (Overlap graph). *An overlap graph is a quadruple $G = (V, E, \Sigma, \ell, \text{ov})$ where V is a finite set of vertices, $\ell: V \rightarrow \Sigma^+$ labels the vertices, $E \subseteq V \times V$ is the set of directed edges, and $\text{ov}: E \rightarrow \mathbb{N}$ defines the length of a (possibly empty) suffix-prefix overlap between the vertices of each edge $uv \in E$ such that the following holds: $0 \leq \text{ov}(uv) < \min(\|u\|, \|v\|)$ and*

$$\ell(u)[\|u\| - \text{ov}(uv) + 1 :] = \ell(v)[: \text{ov}(uv)].$$

Condition $\text{ov}(uv) < \min(\|u\|, \|v\|)$ imposes the overlap to be less than one full vertex.

See Figure 9. We extend ℓ to label paths of G as follows: given path $P[1 : k]$ ($(P[i], P[i+1]) \in E$ for all $i \in [1 : k-1]$), we define path label $\ell(P[1] \cdots P[k])$ as $\ell(u_1)$ if $k = 1$, otherwise it is equal to

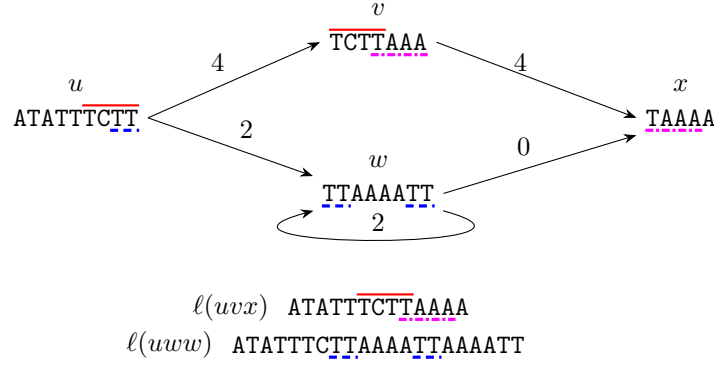
$$\ell(u_1)[\|u_1\| - \text{ov}(u_1 u_2) + 1 :] \cdots \ell(u_{k-1})[\|u_{k-1}\| - \text{ov}(u_{k-1} u_k) + 1 :] \cdot \ell(u_k),$$

that is, the string spelled by path P considers the overlaps between adjacent vertices only once.

6.4.1 Hardness of exact matching in overlap graphs

We say that a pattern $Q \in \Sigma^+$ occurs in an overlap graph G if Q is a substring of $\ell(P)$ for some path P in G . Then, it is immediate to see that the problem of matching a query in an overlap graph is a generalization of string matching in labeled graphs, since the latter are vertex-labeled graphs with overlap equal to 0 for all edges. Then, the conditional lower bounds of [19, 18] holds also for overlap graphs. However, two classes of graphs that bypass the conditional lower bounds by having a *uniqueness property* of the possible strings spelled in the graph are *de Bruijn graphs* and *iEFGs* we considered earlier.

► **Observation 28.** *Overlap graphs (Definition 27) are a generalization of EFGs (Definition 15), since the latter are directed graphs, the vertices are labeled with non-empty strings, and the edges do not present any overlap ($\text{ov}(uv) = 0$ for all $uv \in E$).*



■ **Figure 9** An overlap graph with four vertices u, v, w , and x , where the overlaps are represented above each edge. Note that in the spelling of edges and paths we do not repeat the overlapping substrings.

► **Observation 29.** *Overlap graphs (Definition 27) are a generalization of de Bruijn graphs (Definition 7) and compacted de Bruijn graphs (see e.g. [13]), since these can be represented as k -mers or unitigs connected by directed edges with a fixed overlap of $k - 1$ ($\text{ov}(uv) = k - 1$ for all $uv \in E$).*

► **Observation 30** (Uniqueness property of dBGs). *In a (compacted) de Bruijn Graph $G = (V, E, \Sigma, \ell, \text{ov})$, we have that for every possible k -mer $S \in \Sigma^k$, if S occurs in G then S is uniquely associated to a single node in G .*

6.4.2 Indexable overlap graphs

We study the natural extension of the repeat-free property (Definition 19) and the uniqueness property of de Bruijn Graphs (Observation 30) to overlap graphs, where the occurrences of patterns ignore overlaps between nodes. For this, we need to redefine the interpretation of occurrence positions, as an occurrence starting inside the overlapping part of an edge (u, v) can be seen to start from u or from v .

► **Definition 31.** *Let query Q have an occurrence $(i, u_1u_2 \cdots u_k, j)$ in an overlap graph G with $\|u_1\| - \text{ov}(u_1u_2) < i \leq \|u_1\|$. We say that such occurrence is redundant, as it can be written equivalently as $(i - (\|u_1\| - \text{ov}(u_1u_2)), u_2 \cdots u_k, j)$. Occurrences that are not redundant are called non-redundant.*

► **Definition 32.** *Query Q occurs in an overlap graph G starting inside a node if it has a non-redundant occurrence $(i, u_1, u_2, \dots, u_k, j)$ with $i > 1$. Query Q occurs in an overlap graph G starting from the beginning of a node if all its non-redundant occurrences are of the form $(1, u_1, u_2, \dots, u_k, j)$.*

► **Definition 33.** *We say that an overlap graph $G = (V, E, \Sigma, \ell, \text{ov})$ is repeat-free if each non-redundant occurrence $(i, u_1 \cdots u_k, j)$ of $\ell(u)$ for $u \in V$ is such that $i = 1$ and $u_1 = u$, that is, $\ell(u)$ occurs in G only starting from the beginning of u .*

Then, we argue that the *expanded backward search* algorithm [20] for repeat-free EFGs is correct also for overlap graphs, since it performs the following steps:

1. preprocess G by indexing string

$$T_{\text{edges}} = \# \prod_{(u,v) \in E} \ell(uv) \#$$

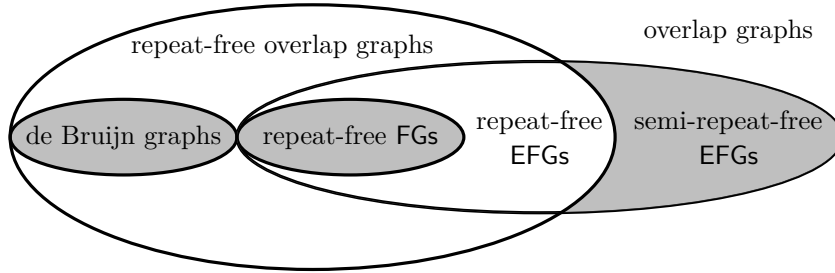
for pattern matching queries and bit arrays $B[1 : |T_{\text{edges}}| + 1]$, $E[1 : |T_{\text{edges}}| + 1]$ for rank and select queries, where B and E are defined such as $B[k] = 1$ and $E[r] = 1$ if and only if $[s : e]$ is the lexicographic interval of some node label $\ell(v)$, $v \in V$; then, given a non-empty lexicographical interval $[s : e]$ of string S in T_{edges} , we can answer in constant time whether $\ell(v)$ is prefix of S for some node $v \in V$;

2. perform backward search for Q in T_{edges} and, when the current lexicographical interval is fully contained in the lexicographical interval of some $\ell(v)$, *expand* the interval to that of $\ell(v)$ and continue the search.

The correctness follows from the repeat-free property and the lexicographic order: whenever we have read a substring $Q[i : j]$ of Q – initially, a suffix of Q – whose lexicographical interval is contained in that of some $\ell(v)$, it must be that $\ell(v)$ is a prefix of $Q[i : j]$ and by the repeat-free property all matches of $Q[i : j]$ in G start from the beginning of v . By expanding the lexicographic range to that of $\ell(v)$, we are discarding the characters $Q[i + |\ell(v)| : j]$ and we can continue with the search without losing any occurrence.

► **Theorem 34.** *A repeat-free overlap graph $G = (V, E, \Sigma, \ell, \text{ov})$ can be indexed in $O(|T_{\text{edges}}|) = O(LD)$ time, where $L = \max_{uv \in E} |\ell(uv)|$ and $D = \max_{v \in V} \max(\delta^+(v), \delta^-(v))$, where $\delta^+(v)$ and $\delta^-(v)$ are the indegree and outdegree of vertex v , to answer whether $Q \in \Sigma^+$ occurs in G in $O(|Q|)$ time.*

We leave it for future work how to construct such repeat-free overlap graphs, which is at least as hard as constructing repeat-free founder graphs from MSA with gaps, for which we do not have an efficient construction either.



■ **Figure 10** Hierarchy of vertex-labeled graph (with non-empty long labels) admitting a poly-time index for linear-time pattern matching. The classes of graphs with known linear-time construction algorithms are highlighted in gray. Note that semi-repeat-free EFGs are a superclass of repeat-free EFGs, that in turn contain repeat-free non-elastic founder graphs, and that the class of overlap graphs (Definition 27) contains all shown classes.

7 Conclusions

This chapter has reviewed BWT-inspired indexing of more complex structures, such as tries, finite languages, de Bruijn graphs, and aligned sequences. As we have seen, much can be gained by tailoring the BWT to these specific types of objects. The next chapter covers a more general indexing framework for labeled graphs, via the notion of Wheeler Graphs.

Even for the special cases we have covered above, many open problems remain. Although enormous progress has been made toward making these data structures practical – in many cases to the point that they are used routinely a variety of bioinformatics pipelines (e.g., the SBWT lies at the heart of the Themisto pseudoalignment software [6]) – further improvements

in performance practical performance are desirable. Construction algorithms in particular have room for improvement, and precisely how best to add parallelization to both construction and querying algorithms is currently under-explored.

References

- 1 Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proc. SODA 2020*, pages 911–930. SIAM, 2020. doi:10.1137/1.9781611975994.55.
- 2 Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Wheeler languages. *Information and Computation*, 281:104820, 2021. doi:10.1016/j.ic.2021.104820.
- 3 Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Longest common prefix arrays for succinct k-spectra. In *Proc. SPIRE*, LNCS 14240, pages 1–13. Springer, 2023. doi:10.1007/978-3-031-43980-3_1.
- 4 Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi. Subset wavelet trees. In *Proc. of the 21st International Symposium on Experimental Algorithms (SEA)*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SEA.2023.4.
- 5 Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Small searchable k-spectra via subset rank queries on the spectral Burrows-Wheeler transform. In *Proc. ACDA*, pages 225–236. SIAM, 2023. doi:10.1137/1.9781611977714.20.
- 6 Jarno N. Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J. Puglisi. Themisto: a scalable colored k -mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinform.*, 39(Supplement-1):260–269, 2023. doi:10.1093/BIOINFORMATICS/BTAD233.
- 7 Uwe Baier. On undetected redundancy in the Burrows-Wheeler transform. In *Proc. CPM 2018*, volume 105 of *LIPIcs*, pages 3:1–3:15, 2018. doi:10.4230/LIPIcs.CPM.2018.3.
- 8 Andrej Baláž, Travis Gagie, Adrián Goga, Simon Heumos, Gonzalo Navarro, Alessia Petescia, and Jouni Sirén. Wheeler maps. In *Proc. LATIN 2024*, volume 14578 of *LNCS*, pages 178–192. Springer, 2024. doi:10.1007/978-3-031-55598-5_12.
- 9 Christina Boucher, Alex Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In *Proc. DCC 2015*, pages 383–392. IEEE, 2015. doi:10.1109/DCC.2015.70.
- 10 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012. doi:10.1007/978-3-642-33122-0_18.
- 11 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994. URL: <http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html>.
- 12 Bastien Cazaux and Eric Rivals. Linking BWT and XBW via Aho-Corasick automaton: Applications to run-length encoding. In *Proc. CPM 2019*, volume 128 of *LIPIcs*, pages 24:1–24:20, 2019. doi:10.4230/LIPIcs.CPM.2019.24.
- 13 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinform.*, 32(12):201–208, 2016. doi:10.1093/BIOINFORMATICS/BTW279.
- 14 David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proc. SODA 1996*, pages 383–391. SIAM, 1996. doi:10.1145/313852.314087.
- 15 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on Wheeler DFAs. In *Proc. DCC 2023*, pages 150–159. IEEE, 2023. doi:10.1109/DCC55655.2023.00023.
- 16 Nicola Cotumaccio, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages - part I. *Journal of the ACM*, 70(4):1–73, 2023. doi:10.1145/3607471.

- 17 Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *LIPIcs*, pages 55:1–55:15, 2019. doi:10.4230/LIPICS.ICALP.2019.55.
- 18 Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. *Theor. Comput. Sci.*, 975:114128, 2023. doi:10.1016/J.TCS.2023.114128.
- 19 Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Trans. Algorithms*, 19(3):21:1–21:25, 2023. doi:10.1145/3588334.
- 20 Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing founder graphs. *Algorithmica*, 85(6):1586–1623, 2023. doi:10.1007/S00453-022-01007-W.
- 21 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):4, 2009. doi:10.1145/1613676.1613680.
- 22 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
- 23 Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009. doi:10.1016/j.tcs.2009.09.012.
- 24 Travis Gagie, Aleksi Hartikainen, Kalle Karhu, Juha Kärkkäinen, Gonzalo Navarro, Simon J. Puglisi, and Jouni Sirén. Document retrieval on repetitive string collections. *Information Retrieval Journal*, 20(3):253–291, 2017. doi:10.1007/s10791-017-9297-7.
- 25 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. doi:10.1016/J.TCS.2017.06.016.
- 26 Erik Garrison, Jouni Sirén, Adam M. Novak, Glenn Hickey, Jordan M. Eizenga, Eric T. Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F. Lin, Benedict Paten, and Richard Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36(9):875–879, 2018. doi:10.1038/nbt.4227.
- 27 Daniel Gibney and Sharma V. Thankachan. On the complexity of recognizing wheeler graphs. *Algorithmica*, 84(3):784–814, 2022. doi:10.1007/S00453-021-00917-5.
- 28 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exp.*, 44(11):1287–1314, 2014. doi:10.1002/SPE.2198.
- 29 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. SODA 2003*, pages 841–850. SIAM, 2003. URL: <http://dl.acm.org/citation.cfm?id=644108.644250>.
- 30 Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. On-line pattern matching on similar texts. In *Proc. CPM 2017*, volume 78 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.CPM.2017.9.
- 31 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.
- 32 Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013. doi:10.1093/bioinformatics/btt215.
- 33 Songbo Huang, T.W. Lam, W.K. Sung, S.L. Tam, and S.M. Yiu. Indexing similar DNA sequences. In *Proc. AAIM 2010*, volume 6124 of *LNCS*, pages 180–190. Springer, 2010. doi:10.1007/978-3-642-14355-7_19.
- 34 Sorina Maciucă, Carlos del Ojo Elias, Gil McVean, and Zamin Iqbal. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference.

- In *Proc. WABI 2016*, volume 9838 of *LNCS*, pages 222–233. Springer, 2016. doi:10.1007/978-3-319-43681-4_18.
- 35 Giovanni Manzini. XBWT tricks. In *Proc. SPIRE 2016*, volume 9954 of *LNCS*, pages 80–92. Springer, 2016. doi:10.1007/978-3-319-46049-9_8.
 - 36 Miguel A. Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. Practical compressed string dictionaries. *Information Systems*, 56:73–108, 2016. doi:10.1016/j.is.2015.08.008.
 - 37 J. Ian Munro. Tables. In *Proc. FSTTCS 1996*, volume 1180 of *LNCS*. Springer, 1996. doi:10.1007/3-540-62034-6_35.
 - 38 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010. doi:10.1089/cmb.2009.0169.
 - 39 Joong Chae Na, Hyunjoon Kim, Seunghwan Min, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment with gaps. *Theoretical Computer Science*, 710:148–157, 2018. doi:10.1016/j.tcs.2017.02.020.
 - 40 Joong Chae Na, Hyunjoon Kim, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment: A compressed index for similar strings. *Theoretical Computer Science*, 638:159–170, 2016. doi:10.1016/j.tcs.2015.08.008.
 - 41 Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proc. SPIRE 2010*, volume 6393 of *LNCS*, pages 347–358. Springer, 2010. doi:10.1007/978-3-642-16321-0_36.
 - 42 Daisuke Okanohara and Kunihiro Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX 2007*, pages 60–70. SIAM, 2007. doi:10.1137/1.9781611972870.6.
 - 43 Nicola Rizzo, Massimo Equi, Tuukka Norri, and Veli Mäkinen. Elastic founder graphs improved and enhanced. *Theor. Comput. Sci.*, 982:114269, 2024. doi:10.1016/J.TCS.2023.114269.
 - 44 Kunihiro Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007. doi:10.1016/j.jda.2006.03.011.
 - 45 Jouni Sirén. Sampled longest common prefix array. In *Proc. CPM 2010*, volume 6129 of *LNCS*, pages 227–237. Springer, 2010. doi:10.1007/978-3-642-13509-5_21.
 - 46 Jouni Sirén. Indexing variation graphs. In *Proc. ALENEX 2017*, pages 13–27. SIAM, 2017. doi:10.1137/1.9781611974768.2.
 - 47 Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020. doi:10.1093/bioinformatics/btz575.
 - 48 Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing finite language representation of population genotypes. In *Proc. WABI 2011*, volume 6833 of *LNCS*, pages 270–281. Springer, 2011. doi:10.1007/978-3-642-23038-7_23.
 - 49 Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014. doi:10.1109/TCBB.2013.2297101.
 - 50 Esko Ukkonen. Finding founder sequences from a set of recombinants. In Roderic Guigó and Dan Gusfield, editors, *Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2452 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 2002. doi:10.1007/3-540-45784-4_21.
 - 51 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/J.TCS.2005.09.023.

BWT Indexes for Optimal Joins in Graph Databases

Diego Arroyuelo 

Department of Computer Science, Escuela de Ingeniería, Pontificia Universidad Católica de Chile, Santiago, Chile

Millennium Institute for Foundational Research on Data (IMFD), Santiago, Chile

Gonzalo Navarro 

Department of Computer Science, University of Chile, Santiago, Chile

Millennium Institute for Foundational Research on Data (IMFD), Santiago, Chile

Abstract

Graph databases represent data as a labeled directed graph, where the labels refer to properties that connect the entities represented by their source and target vertices. Queries feature, most prominently, sets of edges where source, target, and/or label can be variables; each instantiation of the variables where all the edges occur in the graph is a solution to the query. Worst-case-optimal algorithms to solve those queries have been devised, but they pose significant space requirements. This overhead has hindered the adoption of worst-case-optimal algorithms in real systems. We show that a representation of the graph based on the extended BWT (eBWT), where each edge is seen as an independent string of length 3 (source, label, target) supports worst-case-optimal algorithms while using almost no extra space on top of the raw data. We then show how the idea is generalized to the relational model, where the strings can be longer than 3 and several eBWTs are needed to obtain worst-case optimality. The aim to minimize the amount of space in that case leads to consider novel eBWT variants, where columns other than the last can be chosen. Finally, we show how the same graph representation can be used to solve other typical queries, like finding graph paths that match regular expressions.

2012 ACM Subject Classification Theory of computation → Data structures and algorithms for data management; Theory of computation → Database query processing and optimization (theory); Theory of computation → Data compression

Keywords and phrases Graph databases, Ring index, extended BWT, compact data structures

Digital Object Identifier 10.4230/OASICS.Manzini.2025.14

Funding ANID, Chile, via Millennium Science Initiative Program – Code ICN17_002.

Gonzalo Navarro: Fondecyt Grant 1-230755.

1 Introduction

Graph databases (GDBs) have become increasingly popular [29, 24, 26], as they enable the storage of unstructured information repositories that emphasize the relations between entities in applications such as knowledge graphs, the semantic web, social networks, transaction networks, and communication networks, among others. A number of GDB management systems and prototypes have been introduced [15, 33, 34, 1, 2, 25, 30], along with models and query languages [3, 19, 23]. The efficient implementation of GDBs, however, still faces various challenges [8], such as the efficient implementation of the key algorithms to solve the most relevant queries on GDBs. To achieve the needed efficiency, GDBs require indexes that tend to demand a lot of memory. For example, a common type of query on GDBs are basic graph patterns, which search for subgraphs that are homomorphic to a given query graph. These types of queries can be translated into the relational join of many tables, which is known to be computationally demanding. Research on efficient algorithms to compute



© Diego Arroyuelo and Gonzalo Navarro;
licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 14; pp. 14:1–14:19

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

those joins has led to the concept of worst-case optimal (wco) join algorithms [10], which offer certain guarantees of running time in a worst-case scenario. These wco algorithms, however, generally demand strong indexing schemes, which entails a significant use of memory. Leapfrog Triejoin (LTJ) [35] is one of the most popular wco join algorithms, and has proven to be efficient in practice [35, 25]. However, its high memory consumption makes it less viable in situations of limited memory or when the graphs are excessively large. This high memory usage comes from the need to store indexes for 6 different “permutations” of the graph database (a concept that will be clear soon) in order to work properly.

With other coauthors, we introduced the Ring [7, 5] as an alternative to reduce the excessive memory usage of LTJ. The Ring is able to simulate the 6 permutations needed by LTJ (and the corresponding indexes) by storing only one of them. This allows implementing LTJ using little additional space on top of that used by the database itself, thereby opening the use of wco algorithms to many more applications. We originally conceived the Ring in terms of the BWT [13]. We presented it to a database audience aiming to simplify the stringology aspects as much as possible [7]. Our presentation was closer to that of the Permuterm index [18], which regards the strings as circular, and simplified it to the case of strings of length 3. In the journal version [5] we managed to get rid completely of the stringology concepts, which are difficult to grasp in other areas, and presented the Ring in terms of stable sortings.

In this paper we present the Ring to a stringology audience, and thus properly formalize its concepts in terms of the more elegant extended BWT (eBWT) [28]. The generalization of the ring to relational tables leads to the definition of eBWT variants that are new, to the best of our knowledge.

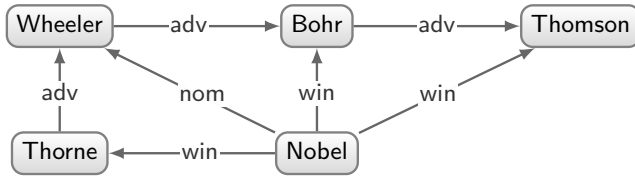
2 Basic Concepts of Graph Databases

2.1 Graph databases

A graph database is an edge-labeled graph modeled as a finite set of triples $G \subseteq \mathcal{U}^3$ where, for simplicity, $\mathcal{U}$ will represent a finite and totally ordered set of constants. We will stick to the Resource Description Framework (RDF) model [27], as it is simpler than others and sufficient to present the important concepts. In RDF, each triple $(s, p, o) \in G$ represents the directed edge $s \xrightarrow{p} o$, connecting vertex s to vertex o , with p being the edge label. The terms *subject*, *predicate*, and *object* are used to refer to s , p , and o , respectively. The number of edges (triples) in G is denoted by $N = |G|$. The alphabet for the vertices of graph G is defined as $\Sigma_V = \{s, o \mid (s, p, o) \in G\}$, while $\Sigma_L = \{p \mid (s, p, o) \in G\}$ represents the alphabet for the edge labels. We assume $\Sigma_V \cap \Sigma_L = \emptyset$, implying that no edge label p is used as a vertex in G . The domain of graph G is given by $\text{dom}(G) = \{s, p, o \mid (s, p, o) \in G\} \subseteq \mathcal{U}$, so we can safely assume $|\mathcal{U}| \leq 3N$. Figure 1 illustrates a sample graph, which will be used as our running example. The figure also presents a particular enumeration of the constants in $\text{dom}(G)$, as well as the corresponding set of graph triples.

2.1.1 Basic Graph Patterns

Let $\mathcal{V}$ denote an infinite set of variables such that $\mathcal{U} \cap \mathcal{V} = \emptyset$. The simplest way of querying an edge-labeled graph is through *triple patterns*, which are tuples $(s, p, o) \in (\mathcal{U} \cup \mathcal{V})^3$. The solutions to a triple-pattern query involve all potential assignments of the variables within the pattern to constants in $\text{dom}(G)$ such that the resulting triple exists in G . This allows for querying all graph edges (using variables s , p , and o), all edges with a vertex $s \in \text{dom}(G)$ as the subject (where s is a constant and p and o remain variables), all edges with label p (p is



Mapping:

1: Bohr	5: Nobel	(4, 6, 1)	(5, 8, 1)
2: Thomson	6: adv	(3, 6, 4)	(5, 8, 2)
3: Thorne	7: nom	(5, 8, 3)	(1, 6, 2)
4: Wheeler	8: win	(5, 7, 4)	

■ **Figure 1** An example graph database (above), a possible enumeration of its constants (below, left), and the corresponding triple representation (below, right).

constant while s and o are variables), and all objects related to subject s by label p (s and p are constant while o is a variable), among other queries. When viewing G as a ternary relation in the relational model, triple patterns are analogous to equality-based selections.

Subsequently, triple patterns can be used to formulate more complex queries, such as *basic graph patterns* (BGP). A BGP represents a finite set $Q \subseteq (\mathcal{U} \cup \mathcal{V})^3$ of triple patterns. Let $\mathbf{V}(Q)$ be the set of variables in a BGP Q . The *evaluation* of Q upon a graph G is defined as set of mappings $Q(G) = \{\mu : \mathbf{V}(Q) \rightarrow \text{dom}(G) \mid \mu(Q) \subseteq G\}$, referred to as *solutions*, where $\mu(G)$ denotes the result of substituting each variable $x \in \mathbf{V}(Q)$ in Q with $\mu(x)$. Some examples of BGPs on the graph of Figure 1 are as follows:

- $Q = \{(\text{Nobel}, \text{win}, x)\}$, for $x \in \mathbf{V}(Q)$, which looks for all Nobel Prize winners. In our example graph, $Q(G)$ contains the instantiations $\mu_1(x) = \text{Thorne}$, $\mu_2(x) = \text{Bohr}$, and $\mu_3(x) = \text{Thomson}$.
- $Q = \{(\text{Nobel}, \text{win}, x), (\text{Nobel}, \text{win}, y), (x, \text{adv}, y)\}$, for $x, y \in \mathbf{V}(Q)$, which looks for all pairs of values x and y such that x advised y and both x and y won the Nobel Prize. In our example graph, $Q(G)$ contains a single instantiation, namely $\mu(x) = \text{Bohr}$ and $\mu(y) = \text{Thomson}$.

2.1.2 Regular Path Queries

Apart from BGPs, the second major component of most graph query languages are the so-called *Regular Path Queries* (RPQs). An RPQ essentially specifies a regular expression on the alphabet Σ_L , and looks for the paths in the graph where the concatenation of the edge labels belongs to the language denoted by the regular expression. An RPQ may also fix the initial and/or the final nodes of the desired paths. The answer to the query are the extremes of all matching paths, in the form of pairs of nodes (s, o) . Additionally, the regular expression may use arrows in reverse order, by specifying a “reversed” version $\hat{p}$ of the symbols $p \in \Sigma_L$. This can be handled by adding an edge $o \xrightarrow{\hat{p}} s$ per graph edge $s \xrightarrow{p} o$, so we omit this feature in the sequel.

For instance, the RPQ $\text{Thorne} \text{adv}^+ x$, where x is a variable, looks for all academic descendants of Thorne in the graph of Figure 1 (the regular expression adv^+ denotes all the nonempty repetitions of the symbol adv). Its solutions are $\mu(x) = \text{Wheeler}$ (because

of the path $\text{Thorne} \xrightarrow{\text{adv}} \text{Wheeler}$, $\mu(x) = \text{Bohr}$ (because of $\text{Thorne} \xrightarrow{\text{adv}} \text{Wheeler} \xrightarrow{\text{adv}} \text{Bohr}$), and $\mu(x) = \text{Thompson}$ (because of $\text{Thorne} \xrightarrow{\text{adv}} \text{Wheeler} \xrightarrow{\text{adv}} \text{Bohr} \xrightarrow{\text{adv}} \text{Thompson}$). Similarly, $x \text{adv}^+ y$, for variables x and y , looks for all instantiations of these variables such that y is an academic descendant of x .

2.2 Worst-Case-Optimal joins

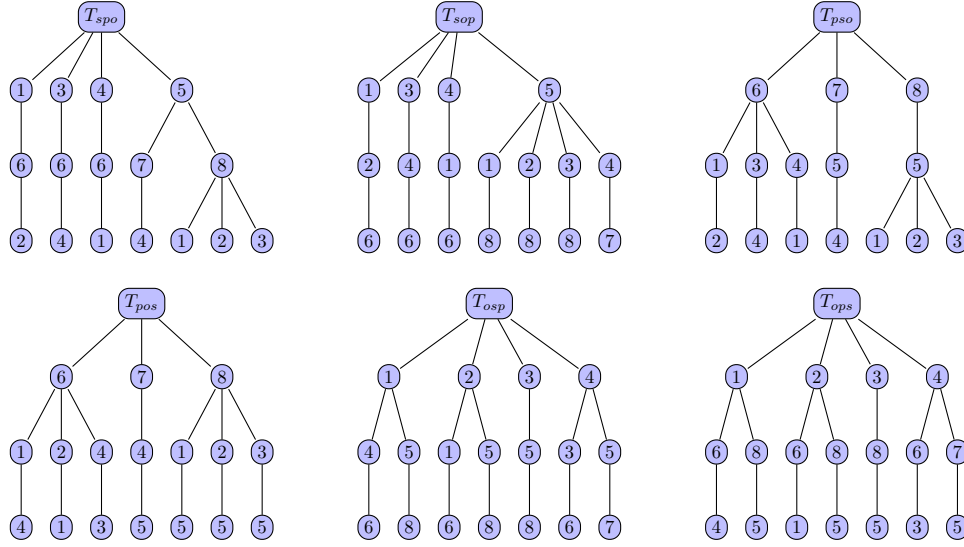
Joins are some of the most resource-intensive operations in relational algebra, making their efficient computation essential for the performance of database systems. A sloppy implementation might end up computing the whole cross product of the involved relations, wasting resources. For a join query Q and a database instance D , a join algorithm enumerates $Q(D)$, the solutions for Q over D . In order to formalize the study of efficient join algorithms, Atserias et al. [10] introduced the concept of the *AGM bound* (named after its authors). Consider a natural join query Q on a database instance D , which comprises a collection of relations. A natural join can be seen as the subset of the Cartesian product where the values in the common attributes coincide (and only one column per common attribute is preserved). The AGM bound of Q over D , denoted Q^* , is the highest possible number of tuples that can result from evaluating Q on any database instance D' that includes, for each relation R in D , a relation R' with the same attributes as R and such that $|R'| \leq |R|$.

A join algorithm is termed *worst-case optimal* (wco) if its running time is bounded by $O(Q^*)$, possibly multiplied by polylogs on N (the number of tuples in D) and factors independent of N (such as $|Q|$ or the number of attributes in D). Although originally conceptualized for relational databases, this idea has also been extended to include graph databases. Even though BGPs are more general than natural joins, it is still possible to apply the AGM bound by considering each triple pattern in a BGP as a relation comprising the triples that match its constants [25].

2.3 Leapfrog TrieJoin

Next, we outline the Leapfrog Triejoin algorithm [35] (LTJ, for short), which is arguably the most commonly used wco join algorithm in practice. In order to work properly, LTJ requires the graph database to be structured using tries. The idea is to represent each triple (graph edge) storing its components in a trie following a particular order (e.g., in *spo* order). Indeed, to achieve worst-case optimality, all $3! = 6$ possible permutations of the tuples need to be maintained, resulting in the storage of 6 tries (the rationale for this requirement will be explained subsequently). We will call T_{spo} , T_{sop} , T_{psp} , T_{pos} , T_{osp} , and T_{ops} these tries. Figure 2 shows the 6 tries corresponding to our running-example graph of Figure 1.

Let $Q = \{t_1, \dots, t_q\}$ be a BGP, where t_i is a triple pattern, for $1 \leq i \leq q$. Let $\mathbf{V}(Q) = \{x_1, \dots, x_v\}$ be the set of variables of Q . The distinctive strategy of LTJ is known as *variable elimination*. This method involves $v = |\mathbf{V}(Q)|$ iterations, each focusing on a single variable from $\mathbf{V}(Q)$ to be “eliminated” from the join process. The particular order in which variables are eliminated determines a total order $\langle x_{i_1}, \dots, x_{i_v} \rangle$ of $\mathbf{V}(Q)$, which is called a Variable Elimination Order (VEO, for short). Throughout this process, each triple pattern t_i is treated as a ternary relation that participates in the join. Once a VEO is established for Q , each triple pattern t_i will have a particular trie T_i associated with it. The trie T_i should first traverse the constants in t_i , and then traverse the variables in t_i in an order that is consistent with that of the VEO. This is why LTJ needs to store 6 tries. To illustrate this, consider the query $Q = \{(x, \text{adv}, y), (z, \text{nom}, x), (z, w, y)\}$, for $x, y, z, w \in \mathbf{V}(Q)$. If the VEO



■ **Figure 2** The 6 tries corresponding to our running-example graph.

is $\langle x, y, z, w \rangle$, then the first triple is associated with trie T_{psp} , the second with T_{pos} , and the third one with T_{osp} . If, instead, the VEO is $\langle z, y, x, w \rangle$, then the tries are T_{pos} , T_{psp} , T_{osp} . If some of these 6 tries were not stored, then some VEOs would become unmanageable.

The navigational operations that must be supported at any trie node v of a trie T are:

- $T.\text{root}()$: moves to the root node of trie T .
- $T.\text{child}(v, c)$: descends to the child of node v labeled c in trie T .
- $T.\text{leap}(v, c_0)$: yields the smallest symbol $c \geq c_0$ that labels a child of node v in trie T .

LTJ starts at the root of each T_i and descends by the children that correspond to the constants in triple t_i . Next, we proceed to the variable elimination phase, assuming a VEO $\langle x_{i_1}, \dots, x_{i_v} \rangle$. For $j = 1, \dots, v$, let $Q_j \subseteq Q$ be the set of triple patterns that contain variable x_{i_j} . Starting with the initial variable x_{i_1} in the VEO, algorithm LTJ first identifies all values $c \in \text{dom}(G)$ such that for every $t \in Q_1$, substituting x_{i_1} with c in t results in a non-empty evaluation of the modified triple pattern t over G (indicating that there are potential solutions to Q where x_{i_1} equals c). If the trie T related to t is consistent with the VEO, then its current node's children precisely represent the suitable values c for x_{i_1} .

Throughout the execution, we maintain a mapping μ with the solutions of Q . When a suitable value c is found for x_{i_1} , we bind x_{i_1} to c , resulting in $\mu = \{(x_{i_1} := c)\}$, and branch on this particular value c . In this branch, we go down by c in all the virtual tries T such that $t \in Q_1$. Next, the same process is applied to Q_2 , determining suitable values d for x_{i_2} , and extending the mapping to $\mu = \{(x_{i_1} := c), (x_{i_2} := d)\}$. This process is repeated for the remaining variables in the VEO. Once all variables have been bound in this way, μ effectively contains a solution for Q ; this occurs multiple times due to the branching for each binding of x_{i_1} , x_{i_2} , and so forth. When all bindings c for a variable x_{i_j} have been considered, LTJ backtracks and moves on to the next binding for Q_{j-1} . Upon completion of this process, the algorithm has reported all possible solutions for Q .

Determining the values c , d , and so forth, involves finding the intersection among the child nodes of the current nodes across all tries T_i , for each $t_i \in Q_j$. The algorithm LTJ carries out this intersection using the primitive $T_i.\text{leap}(v, c_0)$ to implement Barbay and Kenyon's

intersection algorithm [11]. Veldhuizen [35] proved that if the *leap* operation runs within polylogarithmic time, then LTJ is wco regardless of the VEO chosen, provided the tries have an appropriate attribute order.

We next illustrate how LTJ handles the query $Q = \{(x, \text{adv}, y), (z, \text{nom}, x), (z, w, y)\}$ using the VEO $\langle z, y, x, w \rangle$. The triple patterns in Q are associated with tries T_{pos} , T_{pso} , and T_{sop} , respectively. We set $\mu = \emptyset$ and begin by processing the constants **adv** and **nom**, identified as 6 and 7 in our example enumeration, descending to the respective nodes in T_{pos} and T_{pso} . We then handle the variables following the VEO. Starting with z in the second and third triple patterns of Q , we intersect the children at the current nodes of both tries (the node corresponding to string 7 in T_{pso} and the root of trie T_{sop}). The only common child is 5, so we descend to those nodes in both tries, setting $\mu = \{(z := 5)\}$. We now proceed to y , in the first and third triple patterns. We intersect the children of nodes for string 6 in T_{pos} and 5 in T_{sop} . The first intersection element is 1, leading us to descend in both tries and to update $\mu = \{(z := 5), (y := 1)\}$, and to proceed to x in the first and second triple patterns. We intersect the children of nodes corresponding to string 61 in T_{pos} and 75 in T_{pso} . Since 4 is a common value, we descend to the appropriate nodes and update $\mu = \{(z := 5), (y := 1), (x := 4)\}$. Lastly, for w , present only in the third triple pattern, no intersection is required: all children of the node for string 51 in T_{sop} are valid for w . The only value is 8, so we set $\mu = \{(z := 5), (y := 1), (x := 4), (w := 8)\}$. Having bound all variables, μ is a solution to Q . According to our enumeration, these values translate to $z := \text{Nobel}$, $y := \text{Bohr}$, $x := \text{Wheeler}$, $w := \text{win}$, resulting in the set of triples $\{(\text{Wheeler}, \text{adv}, \text{Bohr}), (\text{Nobel}, \text{nom}, \text{Wheeler}), (\text{Nobel}, \text{win}, \text{Bohr})\}$ in the graph. Finally we backtrack, removing w from μ and returning to variable x , and repeat the process until all bindings for the variables of Q are found. In this example, this was the only answer for Q .

3 The Ring Index

In the sequel we show how the Ring implements the functionality needed to implement the LTJ algorithm (see Section 2.3) over a different data representation: the Ring index [7, 5].

3.1 The circular strings model

While LTJ is classically presented as running on tries, we take a more abstract view and consider the database triples (s, p, o) as *circular strings* [12], where we regard all the strings spo , osp , and pos as the same. We formalize circular strings using the concept of rotation.

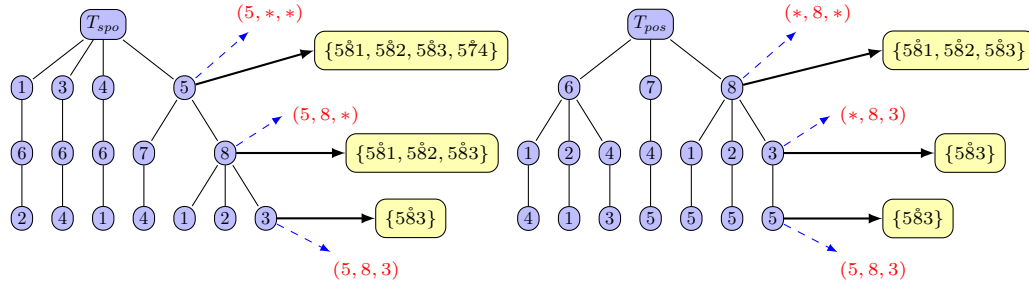
► **Definition 1.** A rotation of a string $S[1..n]$ is any $S[i+1..n]S[1..i]$, for $0 \leq i < n$.

► **Definition 2.** The circular string (or c-string) $\mathring{S}$ is defined as the set of the rotations of S .

In particular, $\mathring{spo} = \{spo, osp, pos\}$; note that $\mathring{spo} = \mathring{osp} = \mathring{pos}$. Note that, because the alphabet of the predicates is disjoint from that of subjects and objects, the three rotations of Def. 2 must be different in our case (because p is at a different position). Therefore, there is a one-to-one correspondence between triples and c-strings.

The graph database is then seen as a set of c-strings, one per triple, and we identify each trie node v with the set of c-strings corresponding to the triples that descend from v . Figure 3 shows the c-strings corresponding to three sample nodes of tries T_{spo} and T_{pos} .

For what follows, we will create two disjoint copies, Σ_s and Σ_o , of the alphabet Σ_V , in order to distinguish subjects from objects in c-strings. We will call $\Sigma_p = \Sigma_L$ the alphabet of predicates for consistency. Thus, triples will belong to $\Sigma_s \times \Sigma_p \times \Sigma_o$. Further, the three



■ **Figure 3** The c-strings corresponding to three nodes of tries T_{spo} and T_{pos} (see solid arrows). The corresponding triple patterns $t(v)$ (according to Definition 3) are also shown for these nodes (see dashed arrows).

sub-alphabets will form disjoint lexicographic ranges. In our running example, we separate the alphabets by adding $|\mathcal{U}|$ to the p components of each triple in the graph, while the o components are increased by $2|\mathcal{U}|$. Let us also redefine triple patterns.

► **Definition 3.** A triple pattern is an element of $(\Sigma_s \cup \{*\}) \times (\Sigma_p \cup \{*\}) \times (\Sigma_o \cup \{*\})$, where $*$ is a special symbol. We say that the elements $*$ are unbound and the others are bound. A triple (s, p, o) matches a triple pattern (s', p', o') iff $x = x'$ or $x' = *$ for every $x \in \{s, p, o\}$.

Every trie node v then corresponds univocally to a triple pattern $t(v)$, where the edges that lead from the root to v define the bound components of $t(v)$. The node v then represents all the database triples that match $t(v)$. Figure 3 also shows the triple patterns corresponding to three nodes of tries T_{spo} and T_{pos} .

For our purposes, a key property of the characterization as circular strings is the following.

► **Theorem 4.** The bound components of any triple pattern $t = (s', p', o')$ can be concatenated into a string $P(t)$ such that a triple (s, p, o) matches t iff $P(t)$ prefixes exactly one string of its c-string $s\hat{p}o$. The prefixed rotation of $s\hat{p}o$ is unique if $P(t) \neq \varepsilon$.

Proof. Let us consider all the possibilities for $|P(t)|$.

If (s, p, o) matches t and the latter has one bound component, then $P(t)$ is univocally s , p , or o , and it prefixes spo , pos , or osp , respectively. It prefixes only one of those because the triple components have disjoint alphabets. Conversely, if $|P(t)| = 1$ and $P(t)$ prefixes spo , pos , or osp , then (s, p, o) matches t .

If there are two bound components, then we can form $P(t) = sp$, $P(t) = po$, or $P(t) = os$, which prefixes spo , pos , or osp , respectively. Again, $P(t)$ prefixes exactly one of those. Conversely again, if $|P(t)| = 2$ and it prefixes spo , pos , or osp , then (s, p, o) matches t . Note that the other choices, $P(t) = ps$, $P(t) = so$, and $P(t) = op$, do not prefix any suffix.

If all the components are bound, then three choices of $P(t)$ prefix (and equal) a string of $s\hat{p}o$ for the only triple (s, p, o) that matches t . Finally, if there are no bound components, then $P(t) = \varepsilon$ prefixes every string, and conversely every triple matches t . ◀

Building on this result, it will suffice to represent tries T_{spo} , T_{osp} , and T_{pos} . Our plan is to identify every trie node v with a set of rotations, which except for the root will contain exactly one rotation of each c-string $s\hat{p}o$ whose triple (s, p, o) matches $t(v)$. By Theorem 4, the rotations are exactly those that start with a given string $P(t)$, so v descends from the root by $P(t)$. The way to efficiently represent that set of suffixes is considered next.

For example, consider the trie T_{pos} on the right of Figure 3. Its rightmost node v descending by 83 has $t(v) = (*, 8, 3)$ and $P(t) = 83$. The node represents the single c-string 583, and contains exactly one rotation of that c-string, 835, which starts with $P(t)$.

3.2 The extended Burrows-Wheeler Transform

The *extended BWT* (*eBWT*) [28, 14] is an extension of the Burrows-Wheeler Transform (BWT) [13] to a set of strings. It builds on the concept of *omega-order* [28] on strings, which is defined as follows.

► **Definition 5.** An infinite string S over alphabet Σ is a mapping $S : \mathbb{N}^+ \rightarrow \Sigma$. The prefix $S[1..i]$ of the infinite string S is the finite string $S[1]S[2] \cdots S[i]$. The suffix $X = S[i..]$ of the infinite string S is the infinite string such that $X[j] = S[i + j - 1]$.

► **Definition 6.** The ω -extension S^ω of a string S is the infinite string formed by concatenations of strings S , that is, $S^\omega[i] = S[1 + ((i - 1) \bmod |S|)]$.

► **Definition 7.** Given a string S , let $m \geq 1$ be the maximum value such that $S = U^m$ for some string U ; we then say that $\text{exp}(S) = m$ and $\text{root}(S) = U$. The omega-order, denoted $<_\omega$, between strings S and T is then defined as follows: $S <_\omega T$ iff

- $\text{root}(S) = \text{root}(T)$ and $\text{exp}(S) < \text{exp}(T)$, or
- $S^\omega < T^\omega$, where $<$ is the lexicographic order.

Note that, if S and T are not a proper prefix of the other, then $<_\omega$ coincides with the lexicographic order, but it can differ otherwise.

► **Definition 8.** Given a set of strings $\{S_1, \dots, S_n\}$, the *eBWT* is a permutation of the symbols in the strings S_i given by listing the rotations of all strings S_i in omega-order, and then concatenating the last symbols of each rotation.

We are now in position to define the Ring index.

► **Definition 9.** The Ring index of a set $\mathcal{D}$ of triples is the *eBWT* of the set $\mathcal{S} = \{spo, (s, p, o) \in \mathcal{D}\}$, after properly separating the alphabets into Σ_s , Σ_p , and Σ_o .

Because of the alphabet separations, it follows that the rotations X of all the c-strings spo have $\text{root}(X) = X$ and $\text{exp}(X) = 1$; therefore the omega-order boils down to the lexicographic order between their ω -extensions. Further, because of our alphabet separation, such order boils down in turn to the plain lexicographic order of the strings. The Ring's eBWT then lists all the strings of all c-strings spo in the graph, sorts them lexicographically, and collects their last symbols.

Figure 4 (left) shows all the spo strings corresponding to the edges of the graph from Figure 1. Since $|\mathcal{U}| = 8$ in this case, we sum 8 to the p component of each triple, and 16 to the o components, and thus $\Sigma_s = [1..8]$, $\Sigma_p = [9..16]$, and $\Sigma_o = [17..24]$. On the right, the triples are lexicographically sorted, obtaining the corresponding Ring index as the last column of the table.

The following theorem shows that we can identify each node v of the three chosen tries with a range of the eBWT of the Ring index.

► **Theorem 10.** There is a distinct interval in the Ring representing the triples that descend from each node v of T_{spo} , T_{osp} , and T_{pos} .

4	14	17	
14	17	4	
17	4	14	
3	14	20	
14	20	3	
20	3	14	
5	16	19	
16	19	5	
19	5	16	
5	15	20	
15	20	5	
20	5	15	
5	16	17	
16	17	5	
17	5	16	
5	16	18	
16	18	5	
18	5	16	
1	14	18	
14	18	1	
18	1	14	

$\xRightarrow{\text{sort}}$

1	14	18	
3	14	20	
4	14	17	
5	15	20	
5	16	17	
5	16	18	
5	16	19	
14	17	4	
14	18	1	
14	20	3	
15	20	5	
16	17	5	
16	18	5	
16	19	5	
17	4	14	
17	5	16	
18	1	14	
18	5	16	
19	5	16	
20	3	14	
20	5	15	

■ **Figure 4** On the left, the 3 rotations of each c-string $s\hat{p}o$ of our running example graph. On the right, the strings are lexicographically sorted. The last column is the corresponding eBWT. The outer red rectangle corresponds to the node v of T_{spo} with $t(v) = (5, *, *)$, and the inner one to $t(v) = (5, 16, *)$.

Proof. If v is the root, we can define its range as the whole eBWT. Otherwise, by Theorem 4, there is a string $P(t)$ for the triple pattern $t = t(v)$ that prefixes exactly one string of the c-string $s\hat{p}o$, or equivalently a rotation of spo , for each triple (s, p, o) matching t . By Def. 9, the three rotations of spo are listed in the Ring in lexicographic order, and thus those prefixed by $P(t)$ form a single range. Those ranges are unique once we define how $P(t)$ is formed. As seen in Theorem 4, there are multiple choices to form $P(t)$ only when $|P(t)| = 3$, that is, when t is a triple. In this case it corresponds to just one c-string $s\hat{p}o$ and the singleton range can be arbitrarily fixed to be that of any of its three rotations. ◀

Consider, for instance, the node corresponding to $t = (5, *, *)$ in T_{spo} of Figure 2, whose subtree includes the triples $(5, 7, 4)$, $(5, 8, 1)$, $(5, 8, 2)$, and $(5, 8, 3)$. This node is represented by the interval $[4..7]$ in the Ring shown in Figure 4 (right side), highlighted with the larger red box, which contains exactly the same triples mentioned before (with components p and o shifted accordingly, as we have already explained). On the other hand, interval $[5..7]$ (highlighted with the smaller red box) corresponds to the node for triple pattern $t = (5, 8, *)$ in the trie T_{spo} of Figure 2. Actually, notice that interval $[1..7]$ in this sample Ring corresponds to the root of trie T_{spo} , interval $[8..14]$ is associated with the root of trie T_{pos} , and interval $[15..21]$ represents the root of trie T_{osp} . Each subinterval within these regions corresponds to nodes in the respective tries.

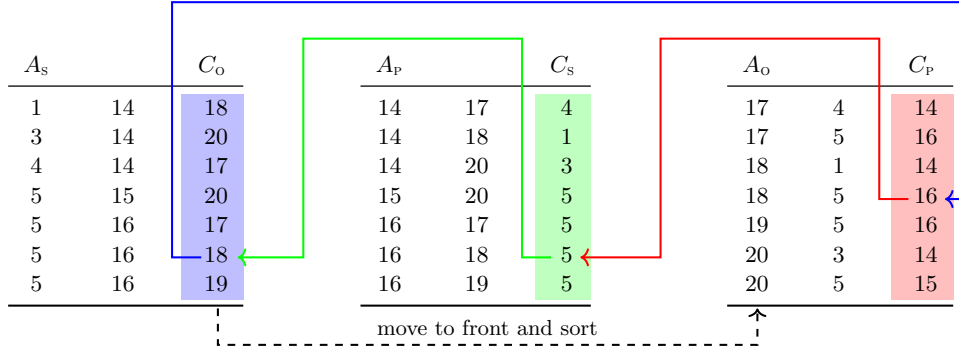
Roughly speaking, the Ring index then maps the trie operations to operations on the eBWT of the triples. In the sequel we show precisely how this is done.

3.3 The eBWT as a representation of the graph

The following definition will help us discuss how the eBWT works on the graph.

► **Definition 11.** Let $S = a_1a_2a_3 \cdots a_k$ be a string where $a_i \in \Sigma_i$ for all i . We then say that the order of S is the string $123 \cdots k$.

In our case, the order of spo is SPO, the order of pos is POS, and the order of osp is OSP. Because the three alphabets form lexicographic ranges, eBWT contains a range where all the listed rotations are of order SPO, and thus the eBWT contains only symbols of Σ_o , a



■ **Figure 5** The Ring index, formed by the three columns C_x , and how each column is obtained by moving some other column to the front and reordering (the dashed arrow shows the case SPO $\rightarrow$ OSP). We also show, with solid arrows, the process for retrieving a triple in 5,16,18 from the index.

second range where all the rotations are of order POS and the eBWT contains only symbols of Σ_s , and a third range where all the rotations are of order OSP and the eBWT contains only symbols of Σ_p . We store those three segments of the eBWT, respectively, as strings, called *columns*, $C_o[1..N]$, $C_s[1..N]$, and $C_p[1..N]$ (see the different ranges and colors in the last column in Figure 4), and say that their orders are SPO, POS, and OSP, respectively.

Let us first show that our representation allows accessing any desired triple (s, p, o) starting from its position in any of the columns C_x . For example, to retrieve the triple represented at $C_o[i]$, we know immediately that $o = C_o[i]$. Now, by the classic BWT invariant [13, 16] (which holds true in the eBWT [28, Prop. 10]), we find the entry $C_p[j]$ corresponding to $C_o[i]$ with

$$j = \text{LF}_o(i) = A_o[o] + \text{rank}_o(C_o, i), \quad (1)$$

where $A_x[c]$ counts the number of times symbols smaller than c occur in C_x , and $\text{rank}_c(X, i)$ counts the number of times c occurs in $X[1..i]$. With j we obtain $p = C_p[j]$, compute the position k of the triple in C_s with

$$k = \text{LF}_p(j) = A_p[p] + \text{rank}_p(C_p, j),$$

and complete the process with $s = C_s[k]$. The eBWT invariants imply that

$$i = \text{LF}_s(k) = A_s[s] + \text{rank}_s(C_s, k),$$

that is, we cycle on the string: the eBWT effectively regards the strings as circular and this permits retrieving the triples starting from any of their components.

See Figure 5 for an example of this process. In this case, we have $i = 6$, and hence $o = C_o[6] = 18$. So, $j = A_o[18] + \text{rank}_{18}(C_o, 6) = 2 + 2 = 4$. Then, $p = C_p[4] = 16$, so $k = A_p[16] + \text{rank}_{16}(C_p, 4) = 4 + 2 = 6$. Then, $s = C_s[6] = 5$. So, we get back to i since $i = A_s[5] + \text{rank}_5(C_s, 6) = 3 + 3 = 6$. The corresponding *spo* triple is (5, 16, 18).

The rationale of (say) Eq. (1) is the following. C_o corresponds to the order SPO and C_p to the order OSP. The second order is obtained from the first by stably reordering the tuples by their o component, in accordance with the lexicographic order. In a stable reordering of SPO, position $C_o[i] = o$ will be moved to position j such that (i) all the symbols less than o come before j (and those are counted in $A_o[o]$), and (ii) all the symbols o preceding the one at $C_o[i]$ also come before j (thus we add $\text{rank}_o(C_o, i)$). Figure 5 also shows one of those reorderings, from SPO to OSP.

Because the shifted symbols are stored in different strings C_x , the Ring remaps their alphabets back to $\Sigma_x = [1 \dots |\Sigma_x|]$. The nodes that are both subjects and objects use the same identifier in both Σ_s and Σ_o , in the range $[1 \dots |\Sigma_s \cap \Sigma_o|]$. Also, the Ring stores the arrays A_x using $N + o(N)$ bits, whereas function rank is computed in time $O(\log |\mathcal{U}|)$ using a wavelet tree [22] representation of the columns $C_x[1 \dots N]$, just like a typical FM-index implementation [17]. Overall, it represents a graph database of N triples over universe $\mathcal{U}$ using $3N \log |\mathcal{U}| + o(N \log |\mathcal{U}|)$ bits (logarithms are to the base 2), and can retrieve any triple in time $O(\log |\mathcal{U}|)$. This is almost the same space needed to store the $3N$ triple identifiers in plain form. What is striking is that, within this space, the Ring can implement all the operations required by the LTJ algorithm, as we see next.

3.4 Implementing the operations on the eBWT

Having separate columns, it is more convenient to identify the root of a trie starting with symbol of alphabet Σ_x as the whole range $C_x[1 \dots N]$. This implements operation root depending on the next triple component we will descend by.

Now assume we are in a range of the eBWT (that is, a range of C_s , C_p , or C_o) that corresponds to a trie node v . Recall that this means that the range is that of all the rotations that start with $P(t(v))$. Let us show how to simulate the trie operations child and leap on this representation.

The “easy” case

In the “easy” case, v is represented as $C_x[i \dots j]$ and the children of v belong to the alphabet Σ_x . We can then use an extension of the LF_x functions to compute the eBWT range corresponding to a desired child of v . For example, if we are at $v = \text{root}()$ and want to descend to $\text{child}(v, s)$, then we move from the range $C_s[1 \dots N]$, which represents v , to the range $C_o[i \dots j] = C_o[A_s[s] + 1 \dots A_s[s + 1]]$, which represents $u = \text{child}(v, s)$. This is the range of all the rotations starting with s . We can now descend to $\text{child}(u, o)$ with the famous *backward search* formula [16], which essentially computes the range of all values $\text{LF}_o(k)$ for all $i \leq k \leq j$ where $C_o[k] = o$:

$$\begin{aligned} i' &= A_o[o] + \text{rank}_o(C_o, i - 1) + 1, \\ j' &= A_o[o] + \text{rank}_o(C_o, j), \end{aligned} \tag{2}$$

so that now $C_p[i' \dots j']$ is the range of all the rotations starting with os , and represents $w = \text{child}(u, o)$. If we further descend to $\text{child}(w, p)$, the analogous formulas will boil down to a single entry of C_s corresponding to the trie leaf representing the triple (s, p, o) (in this case, using the order POS).

For instance, assume that we want to go down by $s = 5$ and then by $o = 19$. We start from the range $C_o[1 \dots 7]$, corresponding to $T_{spo}.\text{root}()$. Going down to the child labeled $s = 5$ corresponds to restricting the range to $C_o[A_s[5] + 1 \dots A_s[6]] = C_o[4 \dots 7]$. Then, descending by $o = 19$ from this node corresponds to moving to the range $C_p[i' \dots j']$, with $i' = A_o[19] + \text{rank}_{19}(C_o, 3) + 1 = 4 + 1 = 5$ and $j' = A_o[19] + \text{rank}_{19}(C_o, 7) = 4 + 1 = 5$. The range $C_p[5 \dots 5]$ then corresponds to the range of a node v in the trie T_{osp} , which represents the triple pattern $t(v) = (5, *, 19)$. Note that, although we do not represent the trie T_{sop} , we can find a node in some trie that corresponds to the same set of c-strings.

Operation $\text{leap}(v, c_0)$ is slightly more complex, and resorts to the geometric capabilities of the wavelet tree. In the “easy case” again, node v is represented by a range $C_x[i \dots j]$ and c_0 belongs to Σ_x ; for example we are in $C_o[i \dots j]$ and c_0 is an object. We then use the wavelet tree ability to compute, in $O(\log |\mathcal{U}|)$ time, the least value not smaller than c_0 in a range $[i \dots j]$ of the string C_x it represents [21].

The “hard” case

If v is the root and we descend by triple component x , we are always in the “easy” case, because we can use $C_x[1..N]$ as $\text{root}()$. It is also always the easy case if $t = t(v)$ has two bound components (or, equivalently, v has depth 2): given any length-2 prefix $P(t)$ of the eBWT rotations, their range is in C_x , where x is the unbound component of t . For example, if $P(t) = sp$, then its eBWT range is within C_o . The only “hard” cases occur when exactly one component is bound and we are forced to move “forward”, instead of “backward” as Eq. (2) does: if only s is bound in t (the cases of P and O are analogous), then the range of $P(t)$ is within order SPO , and then the representation of v is of the form $C_o[i..j]$. We can then descend by o using Eq. (2), but we cannot descend by p in this way. Rather, we must restrict the range $C_o[i..j]$ so that it goes from representing all the rotations starting with s to representing all the rotations starting with sp ; the order is still SPO .

To descend in this hard case, we reorder the string $P(t)$: we descend from the root $C_p[1..N]$ by p , and from the resulting interval $C_s[i'..j']$ of all the rotations starting with p , we descend by s to the interval $C_o[i''..j'']$ of the rotations starting with sp .

For instance, assume that we have instantiated $s = 5$, which corresponds to $C_o[4..7]$ in our example Ring. This range is depicted by the larger box in the Ring of Figure 4. Next, suppose that we instantiate $p = 16$ and want to go down in the corresponding trie. The resulting range is represented by the smaller box in Figure 4. Notice that in this case we remain at C_o (i.e., in trie T_{spo}) after descending with $p = 16$ (unlike the “easy” case, where we jump to a different trie after going down). However, using LF_p to go down by $p = 16$ is not feasible since we are currently in C_o . Instead, we descend from $C_p[1..7]$ to $C_s[A_p[16] + 1..A_p[17]] = C_s[5..7]$, and then proceed with $s = 5$ to obtain the desired range $C_o[i'..j']$, with $i' = A_s[5] + \text{rank}_5(C_s, 4) + 1 = 5$ and $j' = A_s[5] + \text{rank}_5(C_s, 7) = 7$.

Finally, we implement $\text{leap}(v, c_0)$ in the hard case as follows. Consider the same case above, where only s is bound, v is represented by $C_o[i..j]$, and $p_0 \in \Sigma_p$. We start from $C_s[A_p[p_0] + 1..N]$, which is the range of all the rotations starting with predicates $p \geq p_0$. We then map that range using an analogous of Eq. (2):

$$\begin{aligned} i' &= A_s[s] + \text{rank}_s(C_s, A_p[p_0]) + 1, \\ j' &= A_s[s + 1], \end{aligned}$$

and obtain $C_o[i'..j']$, the range of all the rotations starting with s and following with some $p \geq c_0$. The first element of that range, $C_o[i']$, is a triple with the desired minimum value $p \geq p_0$. We then obtain p by traversing the circular string as described in Section 3.3.

4 Rings in Higher Dimensions

The theory of worst-case-optimal join algorithms was initially developed for the relational model, and only then adapted to graph databases [25], where BGPs are modeled mainly as multijoins on a single table of three columns. The space problem is even more serious for tables with more than three attributes: a table with d attributes (which we will say to be of dimension d) needs $d!$ tries, each storing the equivalent of a copy of the database, in order to implement worst-case-optimal joins using LTJ. This, of course, makes the technique unaffordable even for modest amounts of attributes! A natural question is: can we extend the Ring to dimensions $d > 3$ so as to use space proportional to the raw data?

The answer is, unfortunately, no: the key Theorem 4 does not hold for higher dimensions. Already for $d = 4$, let $\{s, p, o, g\}$ be the attributes. No matter how we choose the c -strings (say, $sp\hat{o}g$), there are subsets of attributes for which $P(t)$ do not prefix any rotation of $sp\hat{o}g$ (say, $\{s, o\}$ or $\{p, g\}$).

It is not hard to see, however, that *two* c-strings, like $sp\hat{o}g$ and $s\hat{o}p\hat{g}$, suffice for Theorem 4 to hold on $d = 4$, in the sense that any string $P(t)$, in some convenient order, prefixes some rotation of $sp\hat{o}g$ or of $s\hat{o}p\hat{g}$. Compared to storing $4! = 24$ tries, storing just two eBWTs yields an enormous space reduction!

The Ring simulation of Section 3 can be extended to having two or more eBWTs, by jumping from one to another as needed. The following definition captures what is needed from the chosen set of c-strings for such a simulation to work.

► **Definition 12.** A string Y is a substring of a c-string $\hat{S}$ iff it is a substring of S^ω . If $|Y| \leq |S|$, this is equivalent to Y being a substring of some rotation of S .

► **Definition 13.** A set $\mathcal{S} = \{S_1, S_2, \dots\}$ of strings over alphabet $\Sigma_d = \{1, 2, \dots, d\}$ is complete for dimension d iff, for every $\mathcal{Y} \subset \Sigma_d$ and $x \in \Sigma_d \setminus \mathcal{Y}$, $\mathcal{Y}$ can be ordered to form a string Y such that Yx or xY are a substring of some $\hat{S}_i \in \mathcal{S}$.

The elements of $\mathcal{S}$ are strings of attributes, for example $\mathcal{S} = \{123\}$ is complete for $d = 3$ and $\mathcal{S} = \{1234, 1324\}$ is complete for $d = 4$. A Ring for dimension d is built on a complete set $\mathcal{S}$ of strings of length d . For each data tuple $(a_1, \dots, a_d)$ and each $S_i \in \mathcal{S}$, we extract the string $a_{S_i[1]} \dots a_{S_i[d]}$ (a permutation of the tuple). For example, with $d = 3$ and $\mathcal{S} = \{123\}$, we extract from the tuple (s, p, o) the string spo . With $d = 4$ and $\mathcal{S} = \{1234, 1324\}$, we extract from the tuple (s, p, o, g) the strings $spog$ and $sopg$. The Ring consists of one eBWT for each string S_i , which indexes the rotations of all the c-strings $a_{S_i[1]} \cdot \dots \cdot a_{S_i[d]}$.

Consider a trie node v representing a tuple pattern $t = t(v)$, now of d elements. If $\mathcal{S}$ is complete, then by definition it is possible to concatenate the bound elements of t in some order into a string $P(t)$ that prefixes all rotations in the eBWT of some $\hat{S}_i$. We note that there could be various ways to concatenate the bound attributes of t to form strings $P(t)$ so that they prefix rotations in different eBWTs.

We can then descend from v to $\text{child}(v, x)$, with $x \in \Sigma_x$, in two ways. The “easy” one, where v is represented by a range in C_x , uses the analogous of Eq. (2). The “hard” case discards the current range and recomputes the result from scratch: it forms $P(t(v))$ such that its order of attributes Y , preceded or followed by x , occurs in some $\hat{S}_i$ (this works when $\mathcal{S}$ is complete). It then retraverses the tries from the root following the path of $P(t(v))x$ or $xP(t(v))$, using the analogous of Eq. (2) at each step.

Suppose, with Rings for SPOG and SOPG, that we want to go down using s , then o , then p , and finally g . Initially, without further information, we may descend by s from the root of the trie T_{spog} , obtaining a range of the SPOG Ring. Then, when it comes to go down by o , we note that this is not possible on this Ring, so we move to the SOPG Ring: we start with o and continue by s using the “easy” case, reaching a node in the trie T_{sopg} (i.e., a range in the SOPG Ring). To proceed with p , we can remain in the same Ring, yet it corresponds to the “hard” case, because p is to the right of so in the order SOPG. This can be solved with “easy” cases, either by re-running p , o , and s on the same SOPG Ring, or using o , p , and s if we move to SPOG. In both cases, the final descent by g corresponds to an easy case.

To implement $\text{leap}(v, x_0)$, there are also two possibilities. In the easy case, we proceed on the wavelet tree exactly as explained (for the easy case) in Section 3. In the hard case, there are two subcases:

- We can form $P(t(v))$ whose order Y is such that xY occurs in some $\hat{S}_i$. In this case, we retrace the tries from the root to find Y , and finish as in the easy case. (Note that we were already in some range for the unbound attributes of $t(v)$, but not in one from where we could descend by x .)

- We can form $P(t(v))$ whose order Y is such that YX occurs in some $\dot{S}_i$. In this case, we start from the range $C_x[A_x[x_0] + 1, N]$ in the corresponding eBWT, and descend by $P(t(v))$ from it.

We then track the first cell in the resulting range, using LF, back to C_x to find the answer x . Because of the hard cases, we simulate each child or leap operation in time $O(d \log |\mathcal{U}|)$, which still yields a worst-case-optimal query algorithm, as d is independent of N .

Now that we know how to use multi-eBWT Rings to handle higher dimensions, the question is: how many eBWTs (i.e., strings in $\mathcal{S}$) are needed to be complete in dimension d ? Arroyuelo et al. [5] studied this problem, showing that 5 eBWTs (instead of $5! = 120$ tries) are necessary and sufficient for $d = 5$, and 7 eBWTs (instead of $6! = 720$ tries) are necessary and sufficient for $d = 6$. The optimal number for $d = 7$ is shown to be between 10 and 12, and for $d = 8$ is between 21 and 25 (they obtain various lower bounds). In asymptotic terms, they prove that the number of eBWTs needed is $O(2^d)$ and $\Omega(2^d/\sqrt{d})$. These results make worst-case-optimal algorithms much more feasible for relational databases.

5 Order Graphs

A way to visualize complete sets $\mathcal{S} = \{S_1, S_2, \dots\}$ is what are called *order graphs* [5].

► **Definition 14.** *An order graph of dimension d has, as nodes, all the permutations of $[1 \dots d]$, and has directed labeled edges of the form $Z \xrightarrow{x} xY$.*

In the order graphs that model complete sets of attribute strings, Z is always of the form YX . For example, the order graph for $\mathcal{S} = \{123\}$ consists of the cycle

$$123 \xrightarrow{3} 312 \xrightarrow{2} 231 \xrightarrow{1} 123;$$

recall the example in Figure 5, where we can see how one can move from the SPO order to OSP using LF_O , then to POS using LF_P , and then back to SPO using LF_S . The order graph for $\mathcal{S} = \{1234, 1324\}$ consists of the two cycles:

$$1234 \xrightarrow{4} 4123 \xrightarrow{3} 3412 \xrightarrow{2} 2341 \xrightarrow{1} 1234,$$

$$1324 \xrightarrow{4} 4132 \xrightarrow{2} 2413 \xrightarrow{3} 3241 \xrightarrow{1} 1324.$$

Each edge $e = YX \xrightarrow{x} xY$ of the order graph implies that the Ring stores a column C_e over the alphabet Σ_x , so that the function LF_e maps from order YX to order xY . That is, LF_e applied on a position of the source order leads to the corresponding position in the target order. Backward search, as in Eq. (2), also maps from the first to the second order. The cycles arise because the eBWT, by definition, always stores the *last* column of its rotations.

An alternative way to understand Def. 13 is that, for every Y (in some order) and x , there must exist a *directed path* labeled YX or xY in the order graph: we can go from the source to the target node using LF or backward search and obtain the desired cell or range in the order of target node. This leads to a definition of complete order graphs.

► **Definition 15.** *An order graph of dimension d is complete iff, for every $\mathcal{Y} \subset \Sigma_d = [1 \dots d]$ and $x \in \Sigma_d \setminus \mathcal{Y}$, $\mathcal{Y}$ can be ordered to form a string Y such that concatenating the labels over some directed path in the graph one obtains YX or xY .*

The definition of order graph, however, opens up new alternatives that are not possible under the eBWT framework, namely when Z is not of the form YX in Def. 14. Consider the list of the rotations listed in lexicographic order by the eBWT, and regard the possibility of

with trees possibly sprouting from the nodes [5]. That is, every node must have indegree 1. As it can be inferred from this discussion, no effective ways to build minimal complete sets of c-strings or order graphs for a dimension d exist; we have resorted to exhaustive search.

6 Regular Path Queries

Assume an RPQ such that the initial node s is fixed. A classic solution to this problem is to build the nondeterministic automaton of the regular expression and traverse the graph from s , in DFS or BFS order. As we traverse an edge $s \xrightarrow{p} o$, we feed the automaton with symbol p , and abort the branch if the automaton runs out of active states. We must also record the active automaton states with which we had already visited each graph node, to avoid repeating the same nodes with the same states (i.e., we deactivate the repeated states in this second visit). We report (s, o) for every node o where we arrive with a final automaton state activated.

The other cases are solved analogously: if only the final node o is fixed, we can reverse the regular expression the symbols (i.e. convert every p with $\hat{p}$ and vice versa). If both are fixed, we can start from either of those and stop as soon as the other node is found with a final state. If none is fixed, we start a search from every possible value of s .

6.1 Simulation with the Ring

This process can be simulated with the Ring without using additional data structures. In this case, it is easier to start assuming that only o is fixed, and derive the other cases analogously, so we use the reversed automaton. Our first step is to descend from the trie root by o , $v = \text{child}(\text{root}(), o)$, which is represented by the range $C_p[i..j] = C_p[A_o[o] + 1..A_o[o + 1]]$. We now analyze the automaton states to determine the symbols that would lead to active states. For each such symbol p , we descend to $u = \text{child}(v, p)$ using an analogous of Eq. (2), for predicate p instead of for object o . The resulting range, $C_s[i'..j']$, contains all the distinct sources s that lead to o by symbol p . Since there cannot be repetitions in this list, we extract each value of s and iterate from it as the new source node, that is, setting $o \leftarrow s$.

Note that we do not need the Ring component C_o for this simulation. Further, one can avoid duplicating the edges to support reversed symbols if one uses the “non-typical” case technique to handle them. Such a technique [32] was implemented over a bit-parallel simulation of the automaton, and shown to be very competitive in space and time with classic solutions. Further, it was shown that the used Ring components are equivalent to a known succinct representation of labeled graphs [31, Sec. 9.1.4]

6.2 Larger and smarter

An earlier Ring-based implementation [6], instead, does include the reversed edges, and thus it can always use the “typical” case formulas. This enables other optimizations that make the process faster. One possibility is that, instead of trying out the labels p that are relevant for the automaton, one can enumerate the distinct labels p that occur in $C_p[i..j]$, which could be less. The wavelet tree of C_p can do this enumeration in logarithmic time per resulting symbol [21]. Further, the wavelet tree can be enriched with information on the automaton states that are sources of symbols in ranges of Σ_L . Those can be intersected with the currently active states as we traverse the wavelet tree, so as to prune subtrees that cannot activate any state. This technique computes the intersection between the predicates that lead to active automaton states and those that leave from the current node, in optimal

time, that is, proportional to the alternation complexity of both sets [11]. An analogous technique on the wavelet tree of C_s avoids visiting sources of $C_s[i'..j']$ that have already been visited with all the currently active states.

Using even more space (but still several times less than classic solutions), one can implement even smarter traversals, in considerably less time [6]. For example, the regular expression can be cut at an edge with a label p that is infrequent in the graph. From each edge $s \xrightarrow{p} o$, we launch a backward traversal from s to nodes that activate the initial automaton state, and a forward traversal from o to nodes that activate the final automaton states, and report the Cartesian product of both sets.

7 Perspective

We believe that the most important take-away message for the stringology audience is that there are algorithmic and combinatorial problems in the database community, particularly (but not only) in graph databases, that can benefit from approaches rooted in stringology: the LTJ algorithm works on tries, the Ring structure builds on the eBWT, and RPQs are an extended form of regular expression matching. We have also seen how combinatorial problems arise, like finding lower bounds to the number of Rings needed in dimension d , finding optimal ones efficiently, or characterizing the shapes of optimal order graphs. There is also space for the use of compact data structures: the Ring is a non-redundant version of the six tries used by LTJ, and we have used wavelet trees to solve the more complex LTJ primitives, as well as to find better query plans [4]. Other remarkable uses of compact data structure are compressed quadrees to solve BGP on any dimension d [9] and ordinal tree representations to extend BGPs with topological queries [20].

On the other hand, the application of stringology techniques to databases bounces back with new challenges and ideas. The most intriguing one is probably the idea of permuting the eBWT matrix using an arbitrary column, not necessarily the last one. Could this technique bring meaningful functionalities in other scenarios?

References

- 1 Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems*, 42(4):20:1–20:44, 2017. doi:10.1145/3129246.
- 2 Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *The VLDB Journal*, 31(3):1–26, 2022. doi:10.1007/S00778-021-00711-3.
- 3 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017. doi:10.1145/3104031.
- 4 Diego Arroyuelo, Fabrizio Barisione, Antonio Fariña, Adrián Gómez-Brandón, and Gonzalo Navarro. New compressed indices for multijoins on graph databases. *CoRR*, 2408.00558, 2024. doi:10.48550/arXiv.2408.00558.
- 5 Diego Arroyuelo, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. The Ring: Worst-case optimal joins in graph databases using (almost) no extra space. *ACM Transactions on Database Systems*, 29(2):article 5, 2024.
- 6 Diego Arroyuelo, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. Optimizing RPQs over a compact graph representation. *The VLDB Journal*, 33:349–374, 2024. doi:10.1007/S00778-023-00811-2.

- 7 Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. Worst-case optimal graph joins in almost no space. In *Proc. 47th ACM International Conference on Management of Data (SIGMOD)*, pages 102–114, 2021. doi:10.1145/3448016.3457256.
- 8 Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, and Domagoj Vrgoc. Tackling challenges in implementing large-scale graph databases. *Communications of the ACM*, 67(8):40–44, 2024. doi:10.1145/3653314.
- 9 Diego Arroyuelo, Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. Optimal joins using compressed quadrees. *ACM Transactions on Database Systems*, 47(2):article 8, 2022.
- 10 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013. doi:10.1137/110859440.
- 11 Jérémy Barbay and Claire Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms*, 4(1):4:1–4:18, 2008. doi:10.1145/1328911.1328915.
- 12 Nieves Brisaboa, Ana Cerdeira-Pena, Guillermo de Bernardo, Antonio Fariña, and Gonzalo Navarro. Space/time-efficient rdf stores based on circular suffix sorting. *The Journal of Supercomputing*, 79:5643–5683, 2023. doi:10.1007/S11227-022-04890-W.
- 13 Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- 14 Davide Cenzato and Zsuzsanna Lipták. A theoretical and experimental analysis of BWT variants for string collections. In Hideo Bannai and Jan Holub, editors, *Proc. 33rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 25:1–25:18, 2022. doi:10.4230/LIPICS.CPM.2022.25.
- 15 Orri Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Engineering Bulletin*, 35(1):3–8, 2012. URL: <http://sites.computer.org/debull/A12mar/vicol.pdf>.
- 16 Paolo Ferragina and Giovanni Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
- 17 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.
- 18 Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):article 10, 2010.
- 19 Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proc. 44th ACM International Conference on Management of Data (SIGMOD)*, pages 1433–1445. ACM, 2018. doi:10.1145/3183713.3190657.
- 20 José Fuentes-Sepúlveda, Adrián Gómez-Brandón, Aidan Hogan, Ayleen Iribarra-Cortés, Gonzalo Navarro, and Juan Reutter. Worst-case-optimal joins on graphs with topological relations. In *Proc. 34th International World-Wide Web Conference (WWW)*, 2025. To appear.
- 21 Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012. doi:10.1016/J.TCS.2011.12.002.
- 22 Roberto Grossi, Ankur Gupta, and Jeff S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
- 23 Steve Harris, Andy Seaborne, and Eric Prud’hommeaux. SPARQL 1.1 query language. W3C recommendation, 2013. URL: <https://www.w3.org/TR/sparql11-query/>.
- 24 Aidan Hogan. *The Web of Data*. Springer, 2020. doi:10.1007/978-3-030-51580-5.
- 25 Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for SPARQL. In *Proc. 18th International Semantic Web Conference (ISWC)*, pages 258–275, 2019. doi:10.1007/978-3-030-30793-6_15.

- 26 Emil Eifrem Ian Robinson, Jim Webber. *Graph Databases (2nd Edition)*. O'Reilly Media, Inc., 2015.
- 27 Frank Manola and Eric Miller. RDF primer, 2004. W3C Recommendation.
- 28 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theoretical Computer Science*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.
- 29 Amine Mhedhbi, Amol Deshpande, and Semih Salihoglu. Modern techniques for querying graph-structured databases. *Foundations and Trends in Databases*, 14(2):72–185, 2024. doi:10.1561/19000000090.
- 30 Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endowment*, 12(11):1692–1704, 2019. doi:10.14778/3342263.3342643.
- 31 Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.
- 32 Gonzalo Navarro and Josefa Robert. Compressed graph representations for evaluating regular path queries. In *Proc. 31st International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 218–232, 2024. doi:10.1007/978-3-031-72200-4_17.
- 33 Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010. doi:10.1007/S00778-009-0165-Y.
- 34 Bryan B. Thompson, Mike Personick, and Martyn Cutcher. The bigdata® RDF graph database. In *Linked Data Management*, pages 193–237. Chapman and Hall/CRC, 2014.
- 35 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 96–106, 2014. doi:10.5441/002/ICDT.2014.13.

