# BWT TUNNELING

Universität Ulm
Institut für Theoretische Informatik

Leiter: Prof. Dr. Uwe Schöning

DISSERTATION

zur Erlangung des Doktorgrades **Dr. rer. nat.** der Fakultät
für Ingenieurswissenschaften, Informatik und Psychologie
der Universität Ulm

vorgelegt von
**Uwe Baier** aus **Ulm**
Ulm, 2020

Uwe Baier
*BWT Tunneling*

Amtierender Dekan:
Prof. Dr. Maurits Ortmanns

Gutachter:
Prof. Dr. Enno Ohlebusch
Prof. Dr. Travis Gagie
Prof. Dr. Jacobo Torán

Tag der Prüfung:
14.12.2020

# ABSTRACT

The Burrows-Wheeler-Transform (BWT) is a well-known reversible text transformation invented by David J. Wheeler around 1978 and published by Mike Burrows and David J. Wheeler in 1994. The key idea of the transform is to sort all suffixes of a given string S in lexicographic order and to concatenate all (cyclically) preceding characters of the sorted suffixes, resulting in a new string L. It is well-known that S can be reconstructed solely from string L, and that L has some intriguing properties making it very helpful for lossless data compression as well as sequence analysis.

One such property comes from the observation that, in real-world texts, similar contexts are succeeded (but also preceded) by similar characters. For example, in English text, the letter "q" is commonly followed by the letter "u". Moving over to the BWT, the lexicographically sorted suffixes serve as a kind of context, and thus the BWT gathers preceding characters of similar contexts. As a result, the BWT string L typically contains long repetitions of the same character, making it highly compressible using, e.g., run-length encoding.

The observation that similar contexts are preceded by similar characters can naturally be extended to the observation that similar contexts are preceded by similar strings, e.g. the string "puter" typically is preceded by "com". A new technique called tunneling is able to exploit this observation, leading to a significant reduction of the size of BWT-based data structures, and will therefore be the main topic of this thesis.

In addition to the pure theory behind tunneling, applications of the technique to other BWT fields are shown: in the case of lossless data compression, the technique allows for the building of a BWT-based data compressor achieving compression rates competitive or even superior to those of today's best lossless data compressors. In the field of sequence analysis, tunneling allows for the representation of some string-based data structures at an unprecedented level of succinctness.

# PUBLICATIONS

The following list shows publications that contain the main ideas of this thesis.

[Bai18] Uwe Baier. “On Undetected Redundancy in the Burrows-Wheeler Transform.” In: *Annual Symposium on Combinatorial Pattern Matching*. CPM ’18. 2018, 3:1–3:15.

[BD19] Uwe Baier and Kadir Dede. “BWT Tunnel Planning is Hard But Manageable.” In: *Proceedings of the 2019 Data Compression Conference*. DCC ’19. © 2019 IEEE. 2019, pp. 142–151.

[Bai+20] Uwe Baier, Thomas Büchler, Enno Ohlebusch, and Pascal Weber. “Edge minimization in de Bruijn graphs.” In: *Proceedings of the 2020 Data Compression Conference*. DCC ’20. © 2020 IEEE. 2020, pp. 223–232.

[OSB18] Enno Ohlebusch, Stefan Stauß, and Uwe Baier. “Trickier XBWT Tricks.” In: *String Processing and Information Retrieval*. SPIRE ’18. 2018, pp. 325–333.

Additionally, the author of this thesis was involved in the creation of the following publications during his time at Ulm University:

[Bai16] Uwe Baier. “Linear-time Suffix Sorting - A New Approach for Suffix Array Construction.” In: *Annual Symposium on Combinatorial Pattern Matching*. CPM ’16. 2016, 23:1–23:12.

[BBO15] Uwe Baier, Timo Beller, and Enno Ohlebusch. “Parallel Construction of Succinct Representations of Suffix Tree Topologies.” In: *String Processing and Information Retrieval*. SPIRE ’15. 2015, pp. 234–245.

[BBO16] Uwe Baier, Timo Beller, and Enno Ohlebusch. “Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform.” In: *Bioinformatics* 32.4 (2016), pp. 497–504.

[BBO17] Uwe Baier, Timo Beller, and Enno Ohlebusch. “Space-Efficient Parallel Construction of Succinct Representations of Suffix Tree Topologies.” In: *Journal of Experimental Algorithmics* 22.1 (2017), 1.1:1–1.1:26.

The following list shows student projects which were supervised by the author of this thesis and influenced this work:

[Arn19] Lisa Arnold. “Patternsuche in einer getunnelten BWT.” Bachelor’s thesis. Ulm University, 2019.

[Ded18] Kadir Dede. “Blockwahl beim Tunneln von Burrows Wheeler Transformationen.” Elaboration of Project Algorithm Engineering 2018 (draft by Uwe Baier). 2018.

[Koc19] Matthias Koch. “Getunnelte komprimierte DeBruijn Graphen.” Bachelor’s thesis. Ulm University, 2019.

[Rät19] Caroline Räther. “Heuristic for Tunneled BWT Block Choice.” Elaboration of Project Algorithm Engineering 2018 (draft by Uwe Baier). 2019.

[RHRH20] Sebastian Reyes Häusler and Valentin Reyes Häusler. “Getunnelte eXtended Burrows Wheeler Transformation.” Elaboration of Project Algorithm Engineering 2019 (draft by Uwe Baier). 2020.

[Sab20] Christian Sabisch. “On the Complexity of BWT Tunnel Planning.” Master’s thesis. Ulm University, 2020.

[Web20] Pascal Weber. “Kantenminimierung in DeBruijn Graphen.” Elaboration of Project Algorithm Engineering 2019 (draft by Uwe Baier). 2020.

# ACKNOWLEDGMENTS

First of all, I want to thank Prof. Dr. Enno Ohlebusch who gave me the opportunity to write this work. It can absolutely not be taken for granted that a Ph. D. student works on his own ideas for most of the time, but Prof. Dr. Enno Ohlebusch enabled me to do so. I hereby want to thank him for the confidence, encouragement and support he gave to me during my whole period as a Ph. D. student.

Next, I want to thank Prof. Dr. Jacobo Torán and Prof. Dr. Uwe Schöning for all of the support they gave to me. It is clearly not self-evident that a student can knock on the door of a professor to ask a question and to receive an answer immediately. However, both Prof. Dr. Jacobo Torán and Prof. Dr. Uwe Schöning did so for me, and even stopped their current work to discuss solution approaches when the issues could not be answered immediately.

I am grateful for the nice and uncomplicated contact support of Prof. Dr. Travis Gagie. During the SPIRE'18 conference, it has been said that he "sweated blood" to organize the conference. Nonetheless, it was him which contacted me during the conference actually, and afterwards, he always had an open ear to my questions and proposals. Moreover, I want to thank Prof. Dr. Travis Gagie for the very good ideas related to tunneling–they clearly influenced this work.

Another thanks goes to Prof. Dr. Hans Kestler and to Prof. Dr. Martin Bossert. After the first presentation of the dissertation project, both motivated me to write this thesis as presented, and gave useful hints to other application fields.

Thanks to all of my former colleagues Dr. Oliver Gableske, Dr. Simon Straub, Dr. Patrick Scharpfenecker, Dr. Dominikus Krüger, Dr. Timo Beller, Dr. Helmut Sedding, Dr. Gunnar Völkl, Jan-Hendrik Lorenz, Bogdan Adrian Dina, Julian Nickerl, Florian Wörz and Thomas Büchler. All of those people helped me to take things not too serious, but also were proficient and very obliging conversation partners with excellent tips. Moreover, thanks to our secretaries Waltraud Fromm and Christiane Halder-Schnell for all of the support regarding bureaucracy, as well as for being trustful contact persons when difficult situations occurred.

I am deeply grateful for all of the students that were involved in student projects related to tunneling. Namely, I want to thank Abdul-Kadir Dede, Caroline Räther, Lisa Arnold, Matthias Koch, Pascal Weber, Sebastian Reyes Häusler, Valentin Reyes

Häusler and Christian Sabisch. Despite the non-trivial projects, all of them decided to carry out the projects. I explicitly want to to state that all of the projects had influence on this work. I hope that noone regrets his decision for a tunneling project, and wish them the best for their future.

Special thanks go to my language readers Kate Wolf, Matthias Gerber and Ralf Wörner. They gave me useful tips to compensate my weaknesses in English grammar and spelling, making this thesis more understandable and therethrough accessible to a broader group of persons.

Finally, I want to thank my parents, my brother and sister-in-law as well as my friends Dominik Walter, Michael Schmid, Patrick Mack and Christoph Braunsteffer. Not only that they motivated me when necessary, they also helped me to keep a clear head during the COVID-19 pandemic.

Uwe Baier
Ulm, July 23, 2020

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ALGORITHMS

# 1 INTRODUCTION

Data compression is an indispensable component of modern data communication and storage. When given some data, the task of data compression is to compute a representation of the data which needs less space than the plain data itself. Additionally, the representation must be decompressible, i.e. it must be possible to compute the original data, or data which is similar to the original data.

One can distinguish two types of data compression: lossless and lossy. Lossy data compression typically appears within human-readable media like audio, images and video. In these cases, it is not important to precisely recover the original data. For recognition, the human senses do not need the original data in detail as long as the recovered data is close enough to the original data. To give a concrete example, say we want to compress an image with a resolution of e.g. $720 \cdot 480$ pixels. Then, during decompression, it could be the case that the color values of some pixels differ from that of the pixels in the original image. The human eye is unlikely to recognize a difference between both images given that only a few pixels are modified.

The loss in the quality allows for the compression of the image. Lossy compression is essential if the data is too big to be processed normally. The following example from a lecture slide of Henning Fernau [Fer13, lecture 1, slide 5] underlines this "need" for data compression. Say we want to record a video with a resolution of $720 \cdot 480$ pixels. We use 2 bytes per pixel to encode the color values, and want to record 30 frames per second. The video should have a length of 2 hours. Then, the size of the uncompressed video can be computed as follows:

$$\begin{aligned}
720 \cdot 480 \text{ pixels} &= 345600 \text{ pixels per image (ppi)} \\
345600 \text{ ppi} \cdot 2 \text{ bytes per pixel} &= 691200 \text{ bytes per image (Bpi)} \\
691200 \text{ Bpi} \cdot 30 \text{ frames per second} &= 20736000 \text{ bytes per second (Bps)} \\
20736000 \text{ Bps} \cdot 7200 \text{ seconds} &= 142383 \text{ MB} \approx 139 \text{ GB.}
\end{aligned}$$

Currently, we would consider this quality very low, as the full HDTV standard uses a resolution of $1920 \cdot 1080$ pixels per frame and 3.75 bytes per pixel [Shi11]. With the HDTV resolution, the video size would increase by the factor $\frac{1920 \cdot 1080}{720 \cdot 480} \cdot \frac{3.75}{2} = 11.25$, so that the final size would be about 1563.75 GB, not including audio. Thus, one would need a 2 TB hard drive to store the video. Additionally, to play the video, a data process rate of $\frac{1563.75 \text{ GB}}{7200 \text{ s}} \approx 1800$ Mbit/s would be needed. In the first quarter of 2017, the average peak connection speed of the internet in Germany was about 65.6 Mbit/s [Bel17b]. Thus, streaming services would have been an illusion before 2017, and still would not be available in 2020.

An other type of data compression is lossless data compression. Lossless data compression compresses data and fully recovers the original data. It should be noted that it is not always possible to compress data in a lossless way. If this would be possible, one could recursively compress the compressed representant and thereby obtain a representation using only 1 bit. This would imply that any message would consist of only 1 bit of information content and could not reflect the situation when the underlying alphabet has a size of more than two characters. Lossless data compression is used for pure data transmission in, e.g., download servers or for archiving files. Particular interest is given to this type of compression with the new age of “big data”.

For example, the goal of the 1000 Genomes Project [Res15] was to sequence and assemble 1000 human genomes. Each human genome is made-up of an alphabet of 4 different base pairs and consists of about 3 billion base pairs. Using a naive approach and encoding each base pair with 2 bits, a human genome requires about 715 MB of space. The 1000 genomes project was successfully finished in 2015, producing about 715 GB of data, not including additional labels. In 2019, another genome project called the 100000 Genomes project [Gen] had finished data collection. As human genomes are 99.8% the same, a huge potential of data compression exists.

The age of “big data” also inspired a new research trend: computation on compressed data. The idea is that one does not decompress data before it is processed. Instead, the processing works directly on the compressed data. This saves the time and memory needed to decompress the data. In some cases, computation on compressed data is even faster than the same computation on the raw data.

Finally, data compression offers an economic potential. With increasing internet usage, the amount of data transmission and computation increases. This leads to more energy costs for both transmission and computation, see Figure 1.1. Although modern computer systems are becoming more and more energy efficient, the

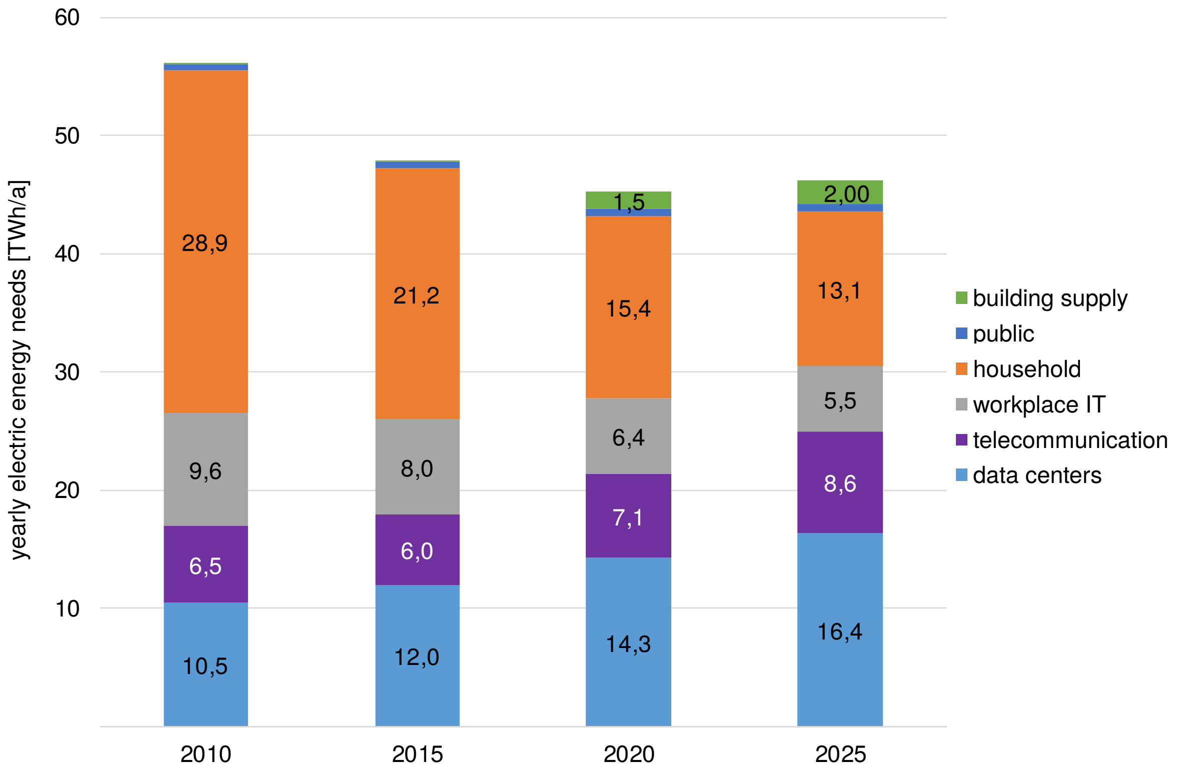


**Figure 1.1:** Prediction of the yearly electric energy needs of computer and communications engineering in Germany, measured in terawatt hours. The prediction comes from a study on behalf of the "Bundesministeriums für Wirtschaft und Energie" which was presented in 2015 [Sto+15]. In 2015, computer and communications engineering required about 9.3 percentage of the entire German electric power consumption. The (translated) image comes from the brief description of the study which is available at `https://www.bmwi.de/Redaktion/DE/Downloads/E/entwicklung-des-ikt-bedingten-strombedarfs-in-deutschland-kurzfassung.pdf` (last visited June 2020) with allowance for publication by the first author of the study, Dr. Lutz Stobbe.

amount of energy required in data centers and data transmission is still increasing. Consequently, developing better compression methods can help to save resources.

Lossless data compression has been introduced by the "father of information theory", Claude E. Shannon in 1948 [Sha48]. Since then, a lot of different methods have been introduced, e.g. source encoding methods from David A. Huffman [Huf52], dictionary-based approaches from Abraham Lempel and Jacob Ziv [ZL77; ZL78], context mixing methods using gradient descent or neuronal networks from Matthew V. Mahoney [Mah05] or text permutation methods from Michael Burrows and David J. Wheeler [BW94].

| $i$ | $\mathsf{L}[i]$ | $S[\mathsf{SA}[i]..\mathsf{SA}[i]+60]$ |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| 426 | `i` | `st also der Wahlspruch der Aufklärung.` |
| 427 | `i` | `st das Unvermögen, sich seines Verstandes ohne Leitung eines` |
| 428 | `i` | `st der Ausgang des Menschen aus seiner selbst verschuldeten U` |
| 429 | `i` | `st diese Unmündigkeit, wenn die Ursache derselben nicht am Ma` |
| 430 | `b` | `st verschuldeten Unmündigkeit. Unmündigkeit ist das Unvermöge` |
| 431 | `r` | `standes ohne Leitung eines anderen zu bedienen. Selbstverschu` |
| 432 | `r` | `standes zu bedienen! ist also der Wahlspruch der Aufklärung.` |
| 433 | `r` | `standes, sondern der Entschließung und des Mutes liegt, sich` |
| 434 | `b` | `stverschuldet ist diese Unmündigkeit, wenn die Ursache dersel` |
| ⋮ | ⋮ | ⋮ |

**Figure 1.2:** Excerpt from the BWT and the sorted suffixes from the first paragraph of the essay "Beantwortung der Frage: Was ist Aufklärung?" from Immanuel Kant [Kan84]. Each suffix prefixed by `sta` is preceded by the character `r`, suffixes prefixed by `st` strongly tend to be preceded by the characters `i`, `r` and `b`.

In this thesis, we will present advantages on lossless data compression that uses the Burrows-Wheeler transform (BWT). In Chapter 2, basics of the BWT, data compression and sequence analysis will be discussed. To show the working principle of the BWT, we give an example in Figure 1.2. The BWT is constructed by first sorting all suffixes of a given text lexicographically. Afterwards, the BWT can be obtained by concatenating the (cyclically) preceding characters of the sorted suffixes.

As Figure 1.2 shows, this has the effect of character clustering in the BWT, making the BWT compressible. Methods to compress this clustered string will be discussed in Chapter 2. Furthermore, this chapter explains how the transformation can be inverted and why the BWT is so useful in both data compression and sequence analysis.

We use this as basis to discuss a technique called tunneling in Chapter 3. Tunneling has been invented by the author of this thesis. It allows one to reduce the length of a BWT when lexicographically adjacent suffixes are preceded by identical strings.

The key idea is to view a BWT as a Wheeler graph. Wheeler graphs are generalizations of a normal BWT and can be understood as a graphical representation of a BWT. Each node contains one outgoing edge labeled with the BWT character of the entry pointing to the node that is reached by a backward step (see Chapter 2 and 3). The idea of tunneling will then be to fuse parallel equally-labeled paths (a so-called prefix interval) into one single path, see Figure 1.3.

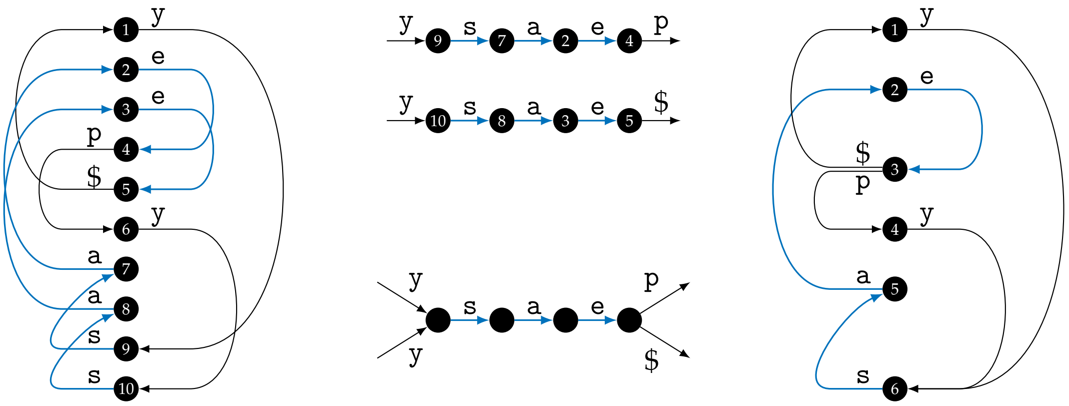


**Figure 1.3:** Wheeler Graph of BWT `yeep$yaass` (left) with prefix interval $(9,7,2,4),(10,8,3,5)$ colored blue. Tunneling of the prefix interval inside the graph (center), tunneled Wheeler graph (right). This image was already published in [BD19] © 2019 IEEE.

A consequence of the fusions is that the number of edges in the Wheeler graph is reduced. This also reduces the length of the succinct representation of the graph.

In addition to the presentation of tunneling, Chapter 3 introduces a theory of overlapping prefix intervals. Moreover, the chapter gives complexity results on the hardness of tunnel planning, showing that tunnel planning is a hard problem.

Chapter 4 relies on tunneling to improve the compression rates of BWT-based data compressors. To this end, the special class of length-maximal run-terminated prefix intervals is introduced. This special class is characterized by fast prefix interval computation. Moreover, the succinct representation of a tunneled Wheeler graph can be compacted.

The remaining contents of Chapter 4 present how BWT-based compressors can be enhanced with tunneling. First, a cost model is introduced which allows one to rate prefix intervals according to their influence on the compression rate when being tunneled. This cost model then allows one to develop heuristics for the problem of deciding which prefix intervals should be tunneled.

Chapter 4 also contains experimental results on the impact of tunneling to the compression rates of BWT-based compressors. Two BWT-based compressors are enhanced with tunneling. As the experiments show, tunneling is able to improve the compression rates of the compressors by about 9% on average. The best results of tunneling are achieved when the data being compressed is large or very repetitive, see Figure 1.4. As a result, tunneling makes BWT-based compressors competitive to state-of-the-art compressors.

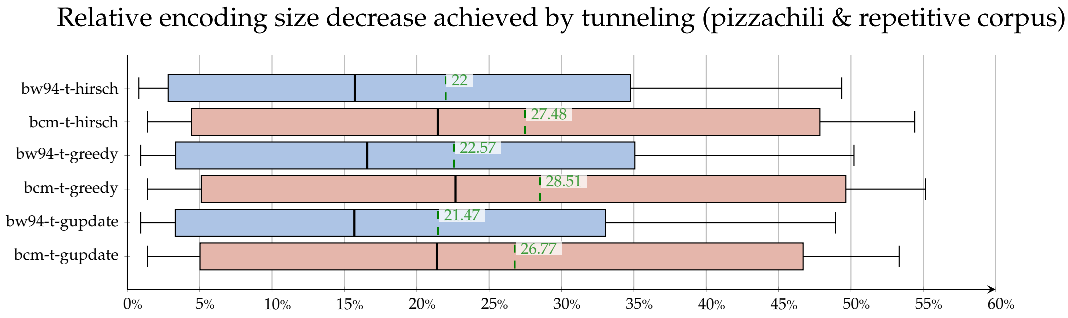


**Figure 1.4:** Boxplot of the relative encoding size decrease achieved by tunneling. The boxplot uses the results of files from the pizzachili and repetitive text corpus, see Chapter A. The blue boxes indicate tunneling in conjunction with the bw94 compressor, red boxes indicate tunneling in conjunction with bcm. The box plots consist of minimum (left whisker), lower quartil, median and upper quartil (boxes) and maximum (right whisker). The green lines indicate the average encoding size decrease. A similar image was already published in [BD19] © 2019 IEEE.

Chapter 5 presents tunneling applications in sequence analysis. First, the chapter shows an approach to find a non-overlapping prefix interval collection using de Bruijn graphs. A de Bruijn graph is a graphical representation of a string obtained by representing each length-$k$ substring of the original sequence as a node. Two nodes are connected when their node labels overlap by $k-1$ characters in the original string. This produces a multi-graph as two length-$k$ substrings can overlap multiple times in a string.

Then, a new compression scheme of these graphs is shown. Every time when a node $x$ is the only predecessor of a node $y$, and $y$ is the only successor of $x$, all edges between $x$ and $y$ can be fused to just one edge. This reduces the amount of edges in the de Bruijn graph and induces a tunneling strategy where the length of the tunneled BWT can be reduced by the same amount.

To minimize the length of tunneled BWTs in this way, the de Bruijn graph edge minimization problem is formulated. The problem asks for the order $k$ such that the edge-reduced de Bruijn graph has the minimal amount of edges, see Figure 1.5. We also provide an efficient algorithm to solve the problem. As experiments show, it is possible to reduce the amount of edges by about 80% in repetitive files. This allows for good tunneling compression of text indices when the data is repetitive.

The second part of Chapter 5 addresses tries. A trie is a tree-based data structure to represent a dictionary of strings. In 2005, Ferragina et al. presented a succinct representation of tries using the BWT, called extended BWT [Fer+05]. This XBWT is the smallest trie representation to date. In Chapter 5, we will present fast XBWT

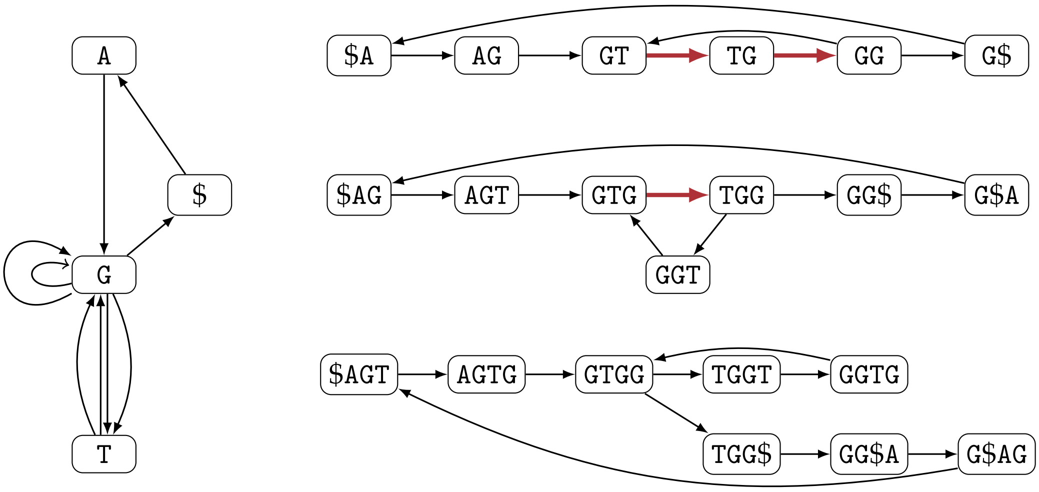


**Figure 1.5:** Edge-reduced de Bruijn graphs with order $k = 1$ (left), $k = 2$ (top right), $k = 3$ (middle right) and $k = 4$ (bottom right) of the string $S =$ AGTGGTGG$. Fused edges are indicated by red arrows, the de Bruijn graph with order $k = 2$ contains the least edges, namely $9 - 2 = 7$ edges. A similar image was already published in [Bai+20] © 2020 IEEE.

construction algorithms. Moreover, by using succinct counters [CAB19] for the construction, we were able to decrease the required memory peak during construction while maintaining the same construction speed.

Afterwards, it is shown how an XBWT can be tunneled. As an XBWT can be expressed as a Wheeler graph, it is not surprising that trie tunneling is possible. However, a key component of tries, the so called failure links, can be retained with the special tunneling strategy of de Bruijn graphs. Therefore, a tunneled XBWT is suitable to perform efficient multi-pattern searches using the Aho-Corasick algorithm

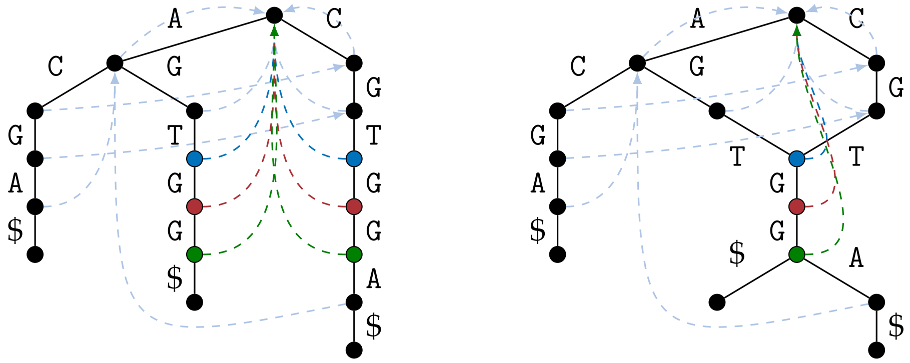


**Figure 1.6:** Trie (left) and tunneled trie (right) of the strings $S_1 =$ ACGA$, $S_2 =$ CGTGGA$ and $S_3 =$ AGTGG$. Both tries are illustrations of a normal and a tunneled XBWT that include failure links.

[AC75]. This is useful for real-world applications like Intrusion Detection Systems which are used to classify incoming messages into malicious and harmless ones. Typically, these systems contain a large dictionary of suspicious strings, and try to classify a message depending on the occurrences of these suspicious strings. As the dictionary can get very large, both fast multi-pattern searches as well as dictionary compression are favored [Nor06].

Trie tunneling can be visualized as a conversion from a tree structure to a directed word graph structure, see Figure 1.6. This makes the tunneling of an XBWT interesting in theory. Moreover, XBWT tunneling is able to decrease the XBWT size by up to 60% in the best cases while leaving the multi-pattern search speed almost unchanged.

Finally, in Chapter 6, we will give concluding remarks, state some open problems and make suggestions for future work.

# 2 PRINCIPLES

This chapter contains basic notations and definitions used in this thesis. We start with some basic notations about strings and lexicographic order. Throughout this thesis, any interval $[i, j]$ is meant to be an interval over the natural numbers, every logarithm is of base 2, and indices start with 1, except when stated differently.

A string $S$ is a finite sequence of letters from an ordered alphabet $\Sigma$ . We call $S$ null-terminated if $S$ ends with the lowest ordered character $\$ \in \Sigma$ occurring only once at the end of $S$, denote by $\sigma$ the size of the alphabet $\Sigma$ and by $n := |S|$ the length of the string $S$. The empty string with length 0 is denoted by $\varepsilon$.

Let $S$ and $T$ be two strings of length $n$ and $m$, let $i, j \in [1, n]$ be two integers and let $c \in \Sigma$ be some character. We denote by

- $S[i]$ the $i$-th character in $S$.
- $S[i..j]$ the substring of $S$ starting at the $i$-th and ending at the $j$-th position. Additionally, we define $S[i..j] = \varepsilon$ if $i > j$.
- $S[i..n]$ the $i$-th suffix of $S$.
- $S <_{\mathsf{lex}} T$ if $S$ is lexicographically smaller than $T$, that is, $S$ is a proper prefix of $T$ ($n < m$ and $S = T[1..m]$) or there exists a $k \in [1, \min\{n, m\}]$ with $S[1..k-1] = T[1..k-1]$ and $S[k] < T[k]$.
- $\mathsf{rank}_S(c, i)$ the number of occurrences of character $c$ in the string $S[1..i]$, that is, $\mathsf{rank}_S(c, i) := |\{\ k \in [1, i] \mid S[k] = c\ \}|$.
- $\mathsf{select}_S(c, i)$ the position of the $i$-th occurrence of character $c$ in $S$ ($i \leq \mathsf{rank}_S(c, n)$), that is, $\mathsf{select}_S(c, i) := \min\{\ k \in [1, n] \mid \mathsf{rank}_S(c, k) \geq i\ \}$.
- $\mathsf{C}_S[c]$ the number of characters which are lexicographically smaller than $c$ in $S$, that is, $\mathsf{C}_S[c] := |\{\ k \in [1, n] \mid S[k] < c\ \}|$.
- $S^R := S[n]S[n-1]..S[1]$ the reverse of the string $S$.
- $c^i := \underbrace{c..c}_{i \text{ times}}$ the $i$-fold repetition of the character $c$.

## 2.1 SUFFIX ARRAY AND BURROWS-WHEELER-TRANSFORM

The suffix array is a very popular data structure in the field of sequence analysis. Presented in 1993 by Manber & Myers [MM93], the suffix array is a compact (although sometimes limited) alternative to the suffix tree [Wei73]. The suffix array can, however, be enhanced to offer the same functionality as a normal suffix tree, using less space [AKO04].

The suffix array can be obtained by sorting all suffixes of a given string lexicographically. The suffix array then stores the start positions of these sorted suffixes in an array.

**Definition 2.1** (Suffix array)**.** Let $S$ be a string of length $n$. The suffix array $\mathsf{SA}$ of $S$ is a permutation of integers in the range $[1, n]$ such that

$$S[\mathsf{SA}[1]..n] <_{\mathsf{lex}} S[\mathsf{SA}[2]..n] <_{\mathsf{lex}} \cdots <_{\mathsf{lex}} S[\mathsf{SA}[n]..n].$$

An example of a suffix array can be found in Figure 2.1. Suffix array construction in general is a non-trivial task. There are several linear-time algorithms for suffix array construction [Kim+03; KS03; KA03; HSS03; Na05; Bai16], but for practical applications, several non-linear-time algorithms [Mor03; FK17] have turned out to be much faster.

A popular application of the suffix array is given in the so-called exact string matching problem. Given a string $S$ of length $n$ and a pattern $P$ of length $m$, one

| $i$ | $\mathsf{SA}[i]$ | $S[\mathsf{SA}[i]..n]$ |
|---|---|---|
| 1 | 10 | $ |
| 2 | 7 | asy$ |
| 3 | 2 | asypeasy$ |
| 4 | 6 | easy$ |
| 5 | 1 | easypeasy$ |
| 6 | 5 | peasy$ |
| 7 | 8 | sy$ |
| 8 | 3 | sypeasy$ |
| 9 | 9 | y$ |
| 10 | 4 | ypeasy$ |

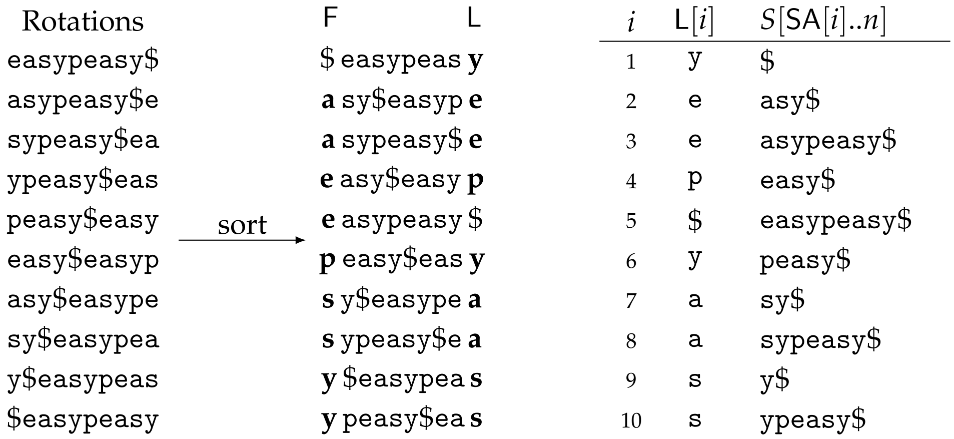
**Figure 2.1:** Suffix array of the null-terminated string $S$ = easypeasy$. The blue-colored rectangle in the suffixes indicates the eas-interval which starts at index 4 and ends at index 5. Thus, the string eas occurs at the positions $\mathsf{SA}[4] = 6$ and $\mathsf{SA}[5] = 1$ in $S$.

wants to find all positions $i \in [1, n - m + 1]$ where the pattern $P$ is contained in $S$, that is, $S[i..i + m - 1] = P$. There exist several algorithms to solve the problem. A popular one is the Knuth-Morris-Pratt algorithm [KMP77], able to determine all of these positions in a run-time of $O(n + m)$. Given a suffix array and the text $S$, by performing a modified binary search on the suffix array, it is possible to determine all suffixes being prefixed by $P$ in $O(m \log n)$ time, see [MM93] for details. Because of the lexicographic sorting, all of the suffixes form an interval in the suffix array. This interval, called an $\omega$-interval in general, spans over all lexicographically sorted suffixes which are prefixed by the string $\omega$, see Figure 2.1 for an example. Now, given a $P$-interval $[i, j]$, we can enumerate all positions where pattern $P$ occurs by printing the values $\mathsf{SA}[i], \mathsf{SA}[i + 1], \ldots, \mathsf{SA}[j]$. Thus, if a suffix array is given, the exact string matching problem requires $O(m \log n + occ)$ time, where $occ$ is the number of occurrences of $P$ in $S$. Using some additional precomputed information, the run-time can be improved to $O(m + \log n + occ)$, see [Ohl13] for more details. This is clearly superior to the run-time of the Knuth-Morris-Pratt algorithm, but one should note that the time for the suffix array construction, as well as space for the suffix array, is not included. Therefore, in the case of a single string search, the Knuth-Morris-Pratt algorithm is still preferable to the suffix array approach.

### 2.1.1 *Burrows-Wheeler-Transform*

The Burrows-Wheeler-Transform is a famous text transformation and also the foundation of this thesis. Originally presented by Mike Burrows and David J. Wheeler in 1994 for the purpose of data compression [BW94], the transformation became very popular in sequence analysis. Examples of usage include the data compression program `bzip2` [Sew96], a very efficient text index named FM-index [FM05] or the program `BWA` which is used to align DNA-subsequences against a reference genome [LD10]. Due to the importance of the Burrows-Wheeler-Transform (BWT) for this work, we will review the application of the BWT in data compression in Section 2.2 and its application in text indexing in Section 2.3.

The idea of the BWT is closely related to the idea of a suffix array: given a string $S$ of length $n$, one sorts the cyclic rotations of $S$ lexicographically. The BWT is then obtained by collecting the last character of each sorted cyclic rotation from top to bottom.



| Rotations | | F | | L |
|---|---|---|---|---|
| easypeasy$ | | $ | easypeas | y |
| asypeasy$e | | a | sy$easyp | e |
| sypeasy$ea | | a | sypeasy$ | e |
| ypeasy$eas | | e | asy$easy | p |
| peasy$easy | sort | e | asypeasy | $ |
| easy$easyp | | p | easy$eas | y |
| asy$easype | | s | y$easype | a |
| sy$easypea | | s | ypeasy$e | a |
| y$easypeas | | y | $easypea | s |
| $easypeasy | | y | peasy$ea | s |

| $i$ | L[$i$] | $S$[SA[$i$]..$n$] |
|---|---|---|
| 1 | y | $ |
| 2 | e | asy$ |
| 3 | e | asypeasy$ |
| 4 | p | easy$ |
| 5 | $ | easypeasy$ |
| 6 | y | peasy$ |
| 7 | a | sy$ |
| 8 | a | sypeasy$ |
| 9 | s | y$ |
| 10 | s | ypeasy$ |

**Figure 2.2:** BWT construction of the string $S =$ easypeasy\$ (left), correspondence of the BWT and the cyclically preceding characters of sorted suffixes (right). The F column corresponds to the first characters of the sorted suffixes, that is, $\mathsf{F}[i] = S[\mathsf{SA}[i]]$.

**Definition 2.2** (Burrows-Wheeler-Transform)**.** Let $S$ be a string of length $n$. The Burrows-Wheeler-Transform of $S$ can be constructed as follows:

1. Build a conceptional matrix $M'$ whose rows correspond to the cyclic rotations $S[1..n], S[2..n]S[1], \ldots, S[n]S[1..n-1]$ of $S$.

2. Compute the matrix $M$ by sorting the rows of $M'$ lexicographically.

3. The last column of $M$, denoted by L, corresponds to the BWT. Furthermore, the first column of $M$ is denoted by F.

An example of a BWT construction is shown in Figure 2.2. Definition 2.2 does not directly give the full impression about the BWT. A simpler explanation may be as follows: given the sorted rotations matrix $M$, the BWT consists of the cyclically preceding characters of the rows of $M$. This holds true because characters in the last column of $M$ are the cyclically preceding characters of the corresponding rows. To clarify this point, we want to give a connection between a BWT and the suffix array.

**Lemma 2.3.** *Let $S$ be a null-terminated string of length $n$, let* SA *be its suffix array and* L *be its BWT. For each position $i \in [1, n]$, the following correspondence between* SA *and* L *holds:*

$$\mathsf{L}[i] = \begin{cases} S[\mathsf{SA}[i] - 1] & \textit{, if } \mathsf{SA}[i] \neq 1 \\ \$ & \textit{, else} \end{cases}$$

*Proof.* Because $S$ is null-terminated, no suffix can be a proper prefix of another suffix in $S$. Therefore, the lexicographic order of suffixes corresponds to the lexicographic order of cyclic rotations of $S$. The proof is concluded by the fact that the last character of a cyclic rotation corresponds to the cyclically preceding character of the rotation, and \$ is the last character in $S$. □

By interpreting Lemma 2.3, it is clear that the BWT consists of the cyclically preceding characters of sorted suffixes, at least for null-terminated strings.[1] However, commonly strings are null-terminated (or we might just append such a character), so normally we can assume that Lemma 2.3 holds true. Additionally, Lemma 2.3 gives us an efficient procedure to compute a BWT: first, compute the suffix array. Then, scan the suffix array from left to right, and apply the case distinction from the lemma to determine the BWT entry. This results in a simple $O(n)$ BWT construction algorithm.

Given a BWT, it is unclear how the original string $S$ can be reconstructed. To this end, our next goal is to introduce the so-called LF-mapping, allowing one to jump from a sorted rotation to the rotation which is shifted one position rightwards.

**Definition 2.4** (LF-mapping)**.** Let $S$ be a string of length $n$, let $M$ be the BWT matrix of $S$ with the first column $\mathsf{F}$ and the last column $\mathsf{L}$. The LF-mapping is a permutation in the range $[1, n]$ defined as follows:

$$\mathsf{LF}[i] := \mathsf{C_L}[\mathsf{L}[i]] + \mathsf{rank_L}(\mathsf{L}[i], i)$$

We write $\mathsf{LF}^x[i]$ for the x-fold application of LF, i.e. $\mathsf{LF}^x[i] := \underbrace{\mathsf{LF}[\mathsf{LF}[\cdots\mathsf{LF}[i]}_{x \text{ times}}\cdots]]$, define $\mathsf{LF}^0[i] := i$ and the inverse[2] of LF as $\mathsf{LF}^{-1}$.
Analogously, we write $\mathsf{LF}^{-x}[i]$ for the x-fold application of the inverse of LF.

As indicated above, the LF-mapping allows one to navigate from a rotation to the rotation obtained by shifting the rotation one position rightwards. Analogously, in the case of a null-terminated string $S$, the LF-mapping leads from a suffix to the suffix which is one character longer. Therefore, following the LF-mapping one step typically is also called a backward step. As an example, consider position 8 in Figure 2.2 with the suffix $S[\mathsf{SA}[8]..10] = \texttt{sypeasy\$}$. Following the LF-mapping then leads to position $\mathsf{LF}[8] = 3$ with $S[\mathsf{SA}[\mathsf{LF}[8]]..10] = \texttt{asypeasy\$}$. The following lemma clarifies this connection.

[1] In non-null-terminated strings, this need not to be true. It is left to the reader to verify the inequality using the string $S = \texttt{baa}$. In case of non-null-terminated strings, the BWT is enhanced with the row index of the original string $S$ in the matrix $M$. This index is called the BWT index.

[2] The inverse LF-mapping is also referred to as $\psi$-array.

**Lemma 2.5.** *Let $S$ be a string of length $n$, let $M$ be the BWT matrix of $S$ with LF-mapping* $\mathsf{LF}$ *and let $i \in [1, n]$ be an integer. Let $r = M_i$ be the $i$-th row of $M$. Then, the matrix row $M_{\mathsf{LF}[i]}$ corresponds to the rotation that is obtained by shifting $r$ to the right, that is,*

$$M_{\mathsf{LF}[i]} = r[n]r[1..n-1]$$

*Additionally, in the case that $S$ is null-terminated, the following connection of* $\mathsf{LF}$ *and the suffix array* $\mathsf{SA}$ *holds:*

$$\mathsf{SA}[\mathsf{LF}[i]] = \begin{cases} \mathsf{SA}[i] - 1 & \textit{, if } \mathsf{SA}[i] \neq 1 \\ n & \textit{, else} \end{cases}$$

*Proof.* Because $\mathsf{L}$ is the last column of $M$, by the definition of the LF-mapping, $\mathsf{C_L}[\mathsf{L}[i]] \leq \mathsf{LF}[i] < \mathsf{C_L}[\mathsf{L}[i] + 1]$ must hold. Consequently, as $\mathsf{F}$ corresponds to the sorted characters of the underlying string $S$, $M_{\mathsf{LF}[i]}[1] = \mathsf{L}[i] = r[n]$ must hold. As $M$ contains the lexicographically sorted rows of $S$, starting from the row $M_i$, $M$ contains exactly $\mathsf{rank_L}(\mathsf{L}[i], i) - 1$ rows which are lexicographically smaller than $r$ and end with the character $\mathsf{L}[i]$. When we shift each such row one character rightwards, the lexicographic order of the new rows reflects the lexicographic order of the unshifted rows: As each such row ends with the same character $\mathsf{L}[i]$, removing this last character does not change the lexicographic order. Afterwards, appending the same character $\mathsf{L}[i]$ at the front of each shortened row also will not change this lexicographic order. Therefore, the row $M_{\mathsf{LF}[i]}$ indicates the row obtained by shifting the lexicographically $\mathsf{rank_L}(\mathsf{L}[i], i)$-th row of $M$ that ends with character $\mathsf{L}[i]$ one position rightwards, which indicates row $r$. Therefore, $M_{\mathsf{LF}[i]} = r[n]r[1..n-1]$ must hold.

The connection between $\mathsf{SA}$ and $\mathsf{LF}$ in case of a null-terminated string $S$ then automatically follows by the equivalence of sorted rotations in $M$ and sorted suffixes in $\mathsf{SA}$. □

The definition of the LF-mapping as the sum of smaller characters and identical characters positioned above seems a bit cryptic. However, there is a much simpler explanation of the mapping, also giving the mapping its name. The LF-mapping describes a correspondence between the characters in $\mathsf{L}$ and $\mathsf{F}$, which is as follows: the $k$-th occurrence of a character $c$ in $\mathsf{L}$ corresponds to the $k$-th occurrence of character $c$ in $\mathsf{F}$. For example, have a look at the suffix y$ at position 9 in the suffix array from

Figure 2.2. The preceding character $\mathsf{L}[9] = \mathtt{s}$ is the first occurrence of an $\mathtt{s}$ in $\mathsf{L}$ and corresponds to the first $\mathtt{s}$ in the $\mathsf{F}$-column (or equivalently, the leftmost character of each sorted suffix), which in our example is given by position 7. Consequently, the suffix at position 7 is the left-extension of $\mathtt{y\$}$ with the preceding character $\mathtt{s}$.

### 2.1.2 *Retransformation*

So far, we have introduced the BWT and the LF-mapping. The attentive reader might have noticed that the BWT is a text transformation, but details about a retransformation are currently missing. To this end, we will provide an algorithm which is able to do such a retransformation. The basic idea is to traverse the BWT using the LF-mapping and thereby pick up the characters in the BWT. As the LF-mapping corresponds to a backward step, this will produce the reverse of the original string, which can then be reversed to yield the original string back. Algorithm 2.1 shows the full retransform procedure.

We will briefly explain some details about Algorithm 2.1. The first part of the algorithm (lines 1–8) computes the $\mathsf{C}$-array of $\mathsf{L}$ and is a combination of counting occurrences and performing a prefix sum computation. The second part (lines 9–12) computes the LF-mapping by scanning the BWT from left to right and incrementing

```
Data: BWT L of a null-terminated string $S$ of length $n$ over alphabet $\Sigma$.
Result: String $S$.

                                                        // compute C-array
 1 let C be an array of size $\sigma$ initialized with zeros
 2 for $i \leftarrow 1$ to $n$ do
 3     $\mathsf{C}[\mathsf{L}[i]] \leftarrow \mathsf{C}[\mathsf{L}[i]] + 1$
 4 $sum \leftarrow 0$
 5 for $i \leftarrow 1$ to $\sigma$ do
 6     $cnt \leftarrow \mathsf{C}[i]$
 7     $\mathsf{C}[i] \leftarrow sum$
 8     $sum \leftarrow sum + cnt$
                                                        // compute LF-mapping
 9 let LF be an array of size $n$
10 for $i \leftarrow 1$ to $n$ do
11     $\mathsf{C}[\mathsf{L}[i]] \leftarrow \mathsf{C}[\mathsf{L}[i]] + 1$
12     $\mathsf{LF}[i] \leftarrow \mathsf{C}[\mathsf{L}[i]]$
                                                        // retransform BWT
13 let $S$ be a string of length $n$
14 $S[n] \leftarrow \$$
15 $j \leftarrow 1$
16 for $i \leftarrow n - 1$ to 1 do
17     $S[i] = \mathsf{L}[j]$
18     $j \leftarrow \mathsf{LF}[j]$
```

**Algorithm 2.1:** Retransformation of a BWT.

the values of the C-array before setting an entry of $\mathsf{LF}[i]$. This ensures that $\mathsf{LF}[i]$ is set to $\mathsf{C_L}[\mathsf{L}[i]] + \mathsf{rank_L}(\mathsf{L}[i], i)$ because of the incrementation of entries in the C-array. Finally, the third part (lines 13–18) retransforms the BWT as discussed above, by filling the string $S$ from back to front. The algorithm obviously has a run-time of $O(n)$.

Note that Algorithm 2.1 requires the original string $S$ to be null-terminated. The reason is that one needs a starting point for the retransformation (line 15). In the case of null-terminated strings it is clear that the suffix \$ is the lexicographically smallest and also shortest suffix. Therefore the uppermost position determines the start position of the backward steps. In the case that the original string is not null-terminated, removing line 14 of Algorithm 2.1 and performing $n$ backward steps will not suffice to reconstruct the original string: the algorithm would produce any rotation of the original string, but not necessarily the correct original string. To this end, in the case of non-null-terminated strings, an additional position (the BWT index) must be stored to ensure that the backward steps start at the correct position. Thus, in the case of non-null-terminated strings, the variable $j$ is set to the BWT index in line 15.

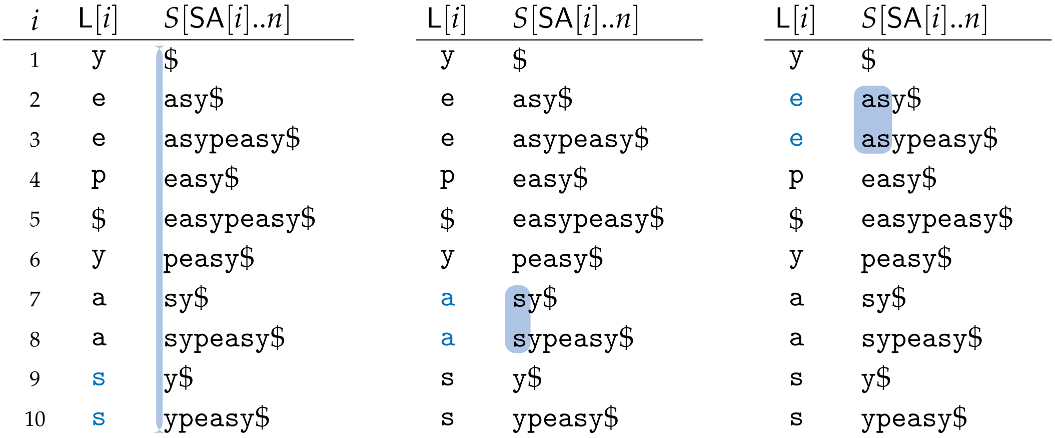
### 2.1.3 *Backward search*

The BWT is also useful for the before mentioned exact string matching problem which asks for all occurrences of a pattern $P$ in a string $S$. The key idea is that, given a $\omega$-interval $[i, j]$ in the suffix array and a character $c$, the $c\omega$-interval $[\tilde{i}, \tilde{j}]$ can be computed as follows:

$$\tilde{i} \leftarrow \mathsf{C_L}[c] + \mathsf{rank_L}(c, i-1) + 1 \qquad\qquad \tilde{j} \leftarrow \mathsf{C_L}[c] + \mathsf{rank_L}(c, j)$$

After computing the new boundaries, $\tilde{i}$ points to the lexicographically smallest suffix which is prefixed by $c\omega$, while $\tilde{j}$ points to the lexicographically largest suffix prefixed by $c\omega$. See Figure 2.3 for an example.

The string matching problem for a pattern $P$ of length $m$ can now be solved as follows: start with the $\varepsilon$-interval $[1, n]$ and iteratively refine the interval to the $P[m]$-interval, the $P[m-1..m]$-interval and so on, until the $P$-interval is known. Then, using the suffix array similar to the approach from Page 10, all occurrences of the pattern $P$ can be enumerated by printing all suffix array entries in the $P$-interval. A full description of one such backward search is given in Algorithm 2.2.



| $i$ | $\mathsf{L}[i]$ | $S[\mathsf{SA}[i]..n]$ | $\mathsf{L}[i]$ | $S[\mathsf{SA}[i]..n]$ | $\mathsf{L}[i]$ | $S[\mathsf{SA}[i]..n]$ |
|---|---|---|---|---|---|---|
| 1 | y | $ | y | $ | y | $ |
| 2 | e | asy$ | e | asy$ | e | asy$ |
| 3 | e | asypeasy$ | e | asypeasy$ | e | asypeasy$ |
| 4 | p | easy$ | p | easy$ | p | easy$ |
| 5 | $ | easypeasy$ | $ | easypeasy$ | $ | easypeasy$ |
| 6 | y | peasy$ | y | peasy$ | y | peasy$ |
| 7 | a | sy$ | a | sy$ | a | sy$ |
| 8 | a | sypeasy$ | a | sypeasy$ | a | sypeasy$ |
| 9 | s | y$ | s | y$ | s | y$ |
| 10 | s | ypeasy$ | s | ypeasy$ | s | ypeasy$ |

**Figure 2.3:** Illustration of a BWT backward search with pattern $P = \mathtt{eas}$. The $\varepsilon$-interval forms the start of the search (left). After one step, The interval is refined to the s-interval (middle). In the second step, the interval further is refined to the as-interval (right). In the final step (which is not depicted here), the eas-interval $[4,5]$ is determined.

Depending on the implementation of the rank-queries, the run-time of Algorithm 2.2 is bounded by $O(m \cdot \mathit{time}(\mathsf{rank}) + \mathit{occ})$, where $m$ is the length of the pattern and $\mathit{occ}$ is the number of occurrences from $P$ in $S$. Paolo Ferragina and Giovanni Manzini were the inventors of the BWT-based backward search [FM05] and used a couple of different structures to answer rank-queries efficiently. A possibility would be to use a $\sigma \cdot n$-sized table with precomputed rank values for each character and position. This results in an optimal $O(m + \mathit{occ})$ algorithm for solving the exact string matching problem, excluding the computation of the BWT, SA and the lookup table. The drawback of such a table is its large space consumption. We will later present

```
Data: BWT L of a string $S$ of length $n$ with C-array $C_L$, suffix array SA of $S$, pattern $P$ of length $m$.
Result: All positions $i \in [1, n-m+1]$ such that $S[i..i+m-1] = P$.

$i \leftarrow 1$
$j \leftarrow n$
for $k \leftarrow m$ to 1 do
    $c \leftarrow P[k]$
    $i \leftarrow C_L[c] + \mathsf{rank}_L(c, i-1) + 1$
    $j \leftarrow C_L[c] + \mathsf{rank}_L(c, j)$
                                                  // stop if no candidates are left
    if $i > j$ then
        return
print SA[$i$], SA[$i+1$], ..., SA[$j$]
```

**Algorithm 2.2:** Backward search for solving the exact string matching problem using a BWT and a suffix array.

another data structure called wavelet tree [GGV03], answering rank-queries and a couple of other operations in $O(\log \sigma)$ run-time and low space consumption.

Summing this section up, we introduced the BWT as a reversible text transformation which is very useful for string operations such as exact pattern matching. Note though that we have only touched the surface of possible BWT applications. For example, the BWT also offers possibilities for approximative pattern matching [LD10], data compression [BW94] or the succinct representation of string-based data structures [GMS17]. The next section shows the historically oldest application of the BWT in the field of data compression.

## 2.2 BLOCK-SORTING COMPRESSION

After the introduction of the BWT, this section will show the first application for which the BWT was developed: data compression. This knowledge is necessary to enhance BWT-based compressors with the tunneling technique, as it will be shown in Chapter 4. We will introduce some selected concepts from lossless data compression, starting with the historically oldest one: source encoding.

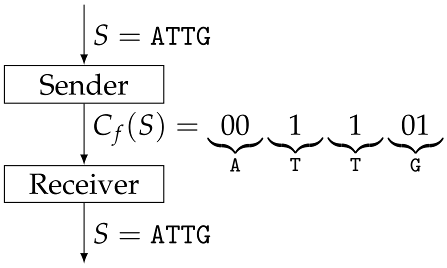
### 2.2.1 *Source encoding*

The typical setting of source encoding is as follows: a sender wants to transmit a sequence $S$ of length $n$ over an alphabet $\Sigma$ to a receiver using as few bits as possible due to a slow transmission line. Both sender and receiver have knowledge about the character frequencies in the sequence $S$, that is, the function

$$f : \Sigma \to \mathbb{N} \quad \text{with} \quad f(c) := |\{\, i \in [1, n] \mid S[i] = c \,\}|$$

is known to both sender and receiver. The task of source encoding is to develop a code $C_f$ such that the binary representation $C_f(S)$ of $S$ is as short as possible but still allows the reconstruction of $S$. Expressed differently, the code $C_f$ must be invertible. Figure 2.4 shows an example of such a setting.

There have been numerous attempts to develop optimal codes based on the relative frequencies of characters in the underlying sequence. The first of such codes, although not optimal, are known as Shannon coding [Sha48] and Shannon-Fano


$S = \texttt{ATTG}$
Sender
$C_f(S) = \underbrace{00}_{\texttt{A}}\underbrace{1}_{\texttt{T}}\underbrace{1}_{\texttt{T}}\underbrace{01}_{\texttt{G}}$
Receiver
$S = \texttt{ATTG}$

| string $S$ | $f(S)$ | $C_f(S)$ |
|---|---|---|
| A | 1 | 00 |
| G | 1 | 01 |
| T | 2 | 1 |
| $c_1 \cdots c_n$ | | $C_f(c_1) \cdots C_f(c_n)$ |

**Figure 2.4:** Source encoding illustration. The left-hand side shows the transmission of an encoded sequence. The right-hand side shows a dictionary of strings, frequencies and their codes. The string $S$ can be reconstructed using $C_f(S)$ because no code word of a character is a prefix of another code word. Encoding the sequence as shown above requires 6 bits, while a normal transmission would require 8 bits (2 bits per character). In general, characters with high frequency should be encoded using short code words, while rare characters should be encoded using longer code words.

coding [Fan49]. Later on, optimal codes were developed and are known as the very popular Huffman coding [Huf52] as well as Arithmetic coding [RL79].

The term "optimal" here means that it is not possible to transmit the sequence $S$ with less bits than produced by the given code. To specify the optimality of a code more precisely, we will next present a lower bound on the amount of information which has to be transmitted to ensure that the source can be reconstructed. The concept is called entropy and was developed by Claude E. Shannon in his famous publication "A Mathematical Theory of Communication" [Sha48].

**Definition 2.6** (Entropy)**.** Let $S$ be a string of length $n$ over an alphabet $\Sigma$ and character frequency function $f$. The entropy $H(S)$ is a lower bound for the average number of bits required to transmit a character of $S$ using any code $C_f$, and is defined as follows:

$$H(S) := \frac{1}{n} \cdot \left( \log_2(n!) - \sum_{c \in \Sigma} \log_2(f(c)!) \right)$$

**Theorem 2.7.** *Let $S$ be a string of length $n$ with alphabet $\Sigma$ and character frequency function $f$. Then, any invertible code $C_f$ which allows the reconstruction of $S$ from $C_f(S)$ using only the frequency function $f$ must satisfy*

$$|C_f(S)| \geq n \cdot H(S).$$

*Proof.* Suppose there exists a code $C_f$ with $|C_f(S)| < n \cdot H(S)$. As $C_f$ uses only the frequency function $f$ to reconstruct the string $S$, but has no further information about the order of the characters, any string $\tilde{S} \neq S$ with equal character frequency function $f$ must use the same amount of bits to be encoded, that is, $|C_f(S)| = |C_f(\tilde{S})|$. Thus, such a code is able to represent less than

$$2^{n \cdot H(S)} = 2^{\log_2(n!) - \sum_{c \in \Sigma} \log_2(f(c)!)} = 2^{\log_2(n!) - \log_2(\prod_{c \in \Sigma} f(c))} = \frac{n!}{\prod_{c \in \Sigma} f(c)!}$$

different strings with character frequency function $f$.

Given a character frequency function $f$, strings with frequency function $f$ may be generated as follows: distribute the $f(c_1)$ occurrences of the first character over $n$ free places in $S$ (there are $\binom{n}{f(c_1)}$ possibilities to do so without producing two equal strings). Then, distribute the $f(c_2)$ occurrences of the second character over the remaining $n - f(c_1)$ free places in $S$ ($\binom{n-f(c_1)}{f(c_2)}$ possibilities), and so on. So overall, we are able to produce

$$\begin{aligned}
&\binom{n}{f(c_1)} \cdot \binom{n - f(c_1)}{f(c_2)} \cdot \ldots \cdot \binom{n - f(c_1) - \cdots - f(c_{\sigma-1})}{f(c_\sigma)} \\
=&\frac{n!}{f(c_1)!(n - f(c_1))!} \cdot \frac{(n - f(c_1))!}{f(c_2)!(n - f(c_1) - f(c_2))!} \cdot \ldots \cdot \frac{(n - \sum_{i=1}^{\sigma-1} f(c_i))!}{f(c_\sigma)!(n - \sum_{i=1}^{\sigma} f(c_i))!} \\
=&\frac{n!}{f(c_1)! \cdot f(c_2)! \cdot \ldots f(c_\sigma)!}
\end{aligned}$$

of such strings. As the code $C_f$ is not able to represent that many strings, a string $S$ must exist which cannot be reproduced using $C_f$. Therefore, $C_f$ is not invertible, which is a contradiction. □

In the original publication [Sha48], Shannon considered "information sources" rather than finite sequences. In this setting, character frequencies are replaced by character probabilities, so the length of the sequence to be encoded is unknown. As we will show, both concepts coincide if we assume that the sequence is "large enough".

**Remark 2.8.** Let $p_1, \ldots, p_\sigma$ be the probabilities of occurrences of characters from an alphabet $\Sigma$ in a sequence $S$. The entropy of the information source then is defined as

$$H(p_1, \ldots, p_\sigma) := \sum_{i=1}^{\sigma} p_i \cdot \log_2 \left( \frac{1}{p_i} \right).$$

The connection between the entropy of an information source and the entropy of a sequence then is given as follows: For characters $c_1, \ldots, c_\sigma$ of a sequence $S$ with length $n$ and character frequency function $f$, define $p_i := \frac{f(c_i)}{n}$. Then, using the special form $\log_2(n!) = n \log_2(n) - n \log_2(e) + O(\log_2(n))$ of Stirling's formula [Sti30] and the identity $\sum_{c \in \Sigma} f(c) = n$, the entropy of a sequence can be written as follows:

$$\begin{aligned}
H(S) &= \frac{1}{n} \cdot \left( \log_2(n!) - \sum_{c \in \Sigma} \log_2(f(c)!) \right) \\
&= \frac{1}{n} \cdot (n \log_2(n) - n \log_2(e) + O(\log_2(n)) \\
&\quad - \sum_{c \in \Sigma} f(c) \log_2(f(c)) - f(c) \log_2(e) + O(\log_2(f(c)))) \\
&= \sum_{c \in \Sigma} \frac{f(c)}{n} \log_2 \left( \frac{n}{f(c)} \right) + O\left( \frac{\log_2(n)}{n} \right) - \sum_{c \in \Sigma} O\left( \frac{\log_2(f(c))}{n} \right) \\
&= H(p_1, \ldots, p_\sigma) + O\left( \frac{\log_2(n)}{n} \right) - \sum_{c \in \Sigma} O\left( \frac{\log_2(f(c))}{n} \right)
\end{aligned}$$

For a large $n$, the error terms $O(\frac{\log_2(n)}{n})$ and $\sum_{c \in \Sigma} O(\frac{\log_2(f(c))}{n})$ converge to zero, so for a large $n$, we can assume $H(S) = H(p_1, \ldots, p_\sigma)$. On the other hand, the entropy of an information source is not always a lower bound for the average number of bits per character needed to transmit a sequence $S$ from a sender to a receiver.

Consider as a counter-example strings with frequency function $f(\mathtt{A}) = 1$ and $f(\mathtt{B}) = 1$. The entropy of the information source is then given by $H(\frac{1}{2}, \frac{1}{2}) = 2 \cdot \frac{1}{2} \cdot \log_2(2) = 1$. On the other hand, as the set of strings with frequency function $f$ is given precisely by `AB` and `BA`, the sender only needs to send a code for the first character (1 bit), because it is automatically clear that the second character is the opposite of the first one. Thus it is possible to design a code such that the transmitted bits per character equals $\frac{1}{2}$, which is below the lower bound of the entropy of the information source.

In the following, we will assume that a sequence can be encoded optimally. Expressed differently, we assume that a code $C$ exists which has an average per-symbol encoding length identical to the entropy. Although this is a hypothetical assumption[3], some methods exist that are very close to this goal, see e.g. [Huf52] or [RL79].

[3] Real-world codes encode a sequence with a discrete number of bits while the information content $|S| \cdot H(S)$ of a string $S$ need not to be a natural number.

An important rule of thumb for entropy is that the entropy is small when the underlying character frequency function is skew. Skewness of a frequency function here means that a small number of characters occur much more frequently than the remaining characters do. Intuitively, if a frequency function is dominated by just a few characters, it makes sense to encode these characters with short code words, while rare characters should be encoded using longer code words, see Figure 2.4. This intuition is covered by the entropy: consider the extreme case where the alphabet of the underlying sequence consists of only one letter. In that case, $H(S) = \frac{1}{n} \cdot (\log_2(n!) - log_2(n!)) = 0$, so we need zero bits to transmit $S$. This makes sense because $S$ can be reconstructed using only the frequency function $f$ by repeating the single character according to its frequency. As a different case, consider a sequence that consists of two characters occurring with equal frequency $\frac{n}{2}$. In this case, the entropy is $H(S) = \frac{1}{n} \cdot \left(\log_2(n!) - 2 \cdot \log_2((\frac{n}{2})!)\right) \approx \frac{1}{n} \cdot \left(n \log_2(n) - n \log_2(\frac{n}{2})\right) = 1$, which indicates that each character must be transmitted using 1 bit, and so coincides with the trivial code for sequences over an alphabet size of two.

### 2.2.2 *Run-length encoding*

Source coding is a compression technique that uses the character frequencies to compress sequences. The compression rate of source encoding is independent of the order in which the characters occur. We will next present a compression technique which instead ignores character frequencies and makes use of the order in which characters occur. More precisely, think of a heavily clustered sequence, that is, a sequence where an occurrence of a character $c$ strongly tends to be succeeded by the same character $c$. An easy way to compress such a string is, given a subsequence consisting of e.g. 1000 times character `A`, replace the sequence by a concatenation of the letter `A` and the number 1000, known as run-length encoding.

```
rle($S$) ← $\varepsilon$
foreach run $[i, j] \in \mathcal{R}$ in ascending order do
    append $S[i]$ to rle($S$)
    for $k \leftarrow 1$ to $|\mathrm{bin}_H(j - i + 1)|$ do
        if $\mathrm{bin}_H(j - i + 1)[k] = 1$ then
            append $1_{\mathrm{rle}}$ to $S$
        else
            append $0_{\mathrm{rle}}$ to $S$
```

**Algorithm 2.3:** Run-length encoding computation of a string $S$.

**Definition 2.9** (Run-length encoding)**.** Let $S$ be a string of length $n$ over an alphabet $\Sigma$. A run of $S$ is a non-empty interval $[i, j] \subseteq [1, n]$ such that

- $i = 1$ or $S[i] \neq S[i-1]$.
- $j = n$ or $S[j] \neq S[j+1]$.
- $S[i] = S[i+1] = \ldots = S[j]$.

Denote by $\mathcal{R} := \{\ [i, j] \subseteq [1, n] \ \mid \ [i, j] \text{ is a run of } S\ \}$ the set of all runs of $S$. Denote by $\mathsf{bin}(i)$ the string corresponding to the binary representation of $i$ with the highest order bit on the left and the lowest ordered bit on the right. Furthermore, denote by $\mathsf{bin}_H(i)$ the headless binary representation of $i$, that is, $\mathsf{bin}_H(i) := \mathsf{bin}(i)[2..|\mathsf{bin}(i)|]$.

The run-length encoded string $\mathsf{rle}(S)$ is generated using Algorithm 2.3. The $c$-run-length encoded string $\mathsf{rle}_c(S)$ is generated by replacing each $[i, j]$-run with $S[i] = c$ in $S$ with a string over the alphabet $\Sigma_{\mathsf{rle}} = \{\mathsf{0}_{\mathsf{rle}}, \mathsf{1}_{\mathsf{rle}}\}$ corresponding to the bit string $\mathsf{bin}_H(j - i + 2)$.

The inverting of a run-length encoded string is straightforward: scan the string for pairs of a character and a headless binary representation of some integer $i$, and append the character $i$-times to $S$. An example of a run-length encoded string can be found in Figure 2.5.

### 2.2.3 *Move-to-front transform*

Another method used to compress sequences is given by the move-to-front transform [Rya80]. The idea of the transform is to replace a character $S[i]$ by the number of distinct characters seen since the last occurrence of the character $S[i]$.

**Definition 2.10** (Move-to-front transform)**.** Let $S$ be a sequence of length $n$ over alphabet $\Sigma$. The move-to-front transform $\mathsf{mtf}(S)$ is a string of length $n$ which is generated using Algorithm 2.4.

```
let $L$ be a list of characters from $\Sigma$ sorted by their first occurrence in $S$
for $i \leftarrow 1$ to $n$ do
    $\mathsf{mtf}(S)[i] \leftarrow$ position of $S[i]$ in the list $L$
    move the character $S[i]$ to the front of the list $L$
```

**Algorithm 2.4:** Move-to-front transformation.

| $i$ | $\mathsf{bin}(i)$ | $\mathsf{bin}_H(i)$ | $S[i]$ | | $\mathsf{rle}(S)[i]$ | $\mathsf{mtf}(S)[i]$ | | $\mathsf{rle}_1(\mathsf{mtf}(S))[i]$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | $\varepsilon$ | A | $\mathsf{bin}_H(1) = \varepsilon$ | A | 1 | $\mathsf{bin}_H(1+1) = 0$ | $1_{\mathsf{rle}}$ |
| 2 | 10 | 0 | G | $\mathsf{bin}_H(3) = 1$ | G | 2 | | 2 |
| 3 | 11 | 1 | G | | $1_{\mathsf{rle}}$ | 1 | $\mathsf{bin}_H(2+1) = 1$ | $1_{\mathsf{rle}}$ |
| 4 | 100 | 00 | G | | A | 1 | | 2 |
| 5 | 101 | 01 | A | | T | 2 | | 3 |
| 6 | 110 | 10 | T | $\mathsf{bin}_H(2) = 0$ | $0_{\mathsf{rle}}$ | 3 | | $0_{\mathsf{rle}}$ |
| 7 | 111 | 11 | T | | | 1 | | |

**Figure 2.5:** Run-length encoding, move-to-front transform and combination of both methods for the string $S = \texttt{AGGGATT}$.

The move-to-front transform can easily be inverted when the original list of characters (the list from line 1 of Algorithm 2.4) is transmitted. Starting with the same list, for $i = 1$ to $n$, set $S[i]$ to the $\mathsf{mtf}(S)[i]$-th character in L, before moving that character to the front of $L$.

The attentive reader might have noticed that the transform itself performs no compression at all: neither the size of the alphabet, nor the length of the transform differs from that of the original sequence. The purpose of the move-to-front transform is something different: it transforms locally skew character frequencies into globally skew character frequencies, which is useful for source encoding.

To see what this means, consider the following example: A sequence $S$ of length $n$ contains four characters A, C, G and T, all with character frequency $\frac{n}{4}$. The entropy of such a sequence can be approximated with $H(S) \approx 2$ bits. Now, assume that the first half of this sequence contains only the characters A and C, while the second half contains only the characters G and T. If we apply the transform to the sequence, the first half of $\mathsf{mtf}(S)$ will contain only the values 1 and 2 (A and B stay in front of list $L$), while the second half of $\mathsf{mtf}(S)$ contains one 3, one 4 (moving the characters G and T to front of $L$) and the values 1 and 2 for all remaining entries. This means that the overall sequence contains $n - 2$ times a 1 or 2, one 3 and one 4. Assuming that the 1's and 2's occur with same frequency, the entropy can be approximated with $H(\mathsf{mtf}(S)) \approx 1$, so we reduced the source encoding size by about half.

This effect typically occurs on sequences with locally skew character frequencies: once a subsequence with skew character frequencies is reached, the dominating characters are moved to the front of the list $L$ and remain at the front until the end of the sequence. Therefore, $\mathsf{mtf}(S)$ mainly consists of small numbers, resulting in a globally skew character frequency.

As depicted in Figure 2.5, run-length encoding and the move-to-front transform can be combined in the form of $\mathsf{rle}_1(\mathsf{mtf}(S))$. This combines the strength of both methods, as

- the move-to-front transform converts local to global skewness.
- $\mathsf{rle}_1$ transforms former runs in the sequence to a run-length encoded form and thereby reduces the size of the sequence.

This combination makes sense when the underlying sequence is both heavily clustered and also has locally skew character frequencies. This idea, in a slightly different way, comes from the inventors of the BWT, Michael Burrows and David J. Wheeler, themselves:

> Although simple Huffman and arithmetic coders do well in step M2, a more complex coding scheme can do slightly better. This is because the probability of a certain value at a given point in the vector R depends to a certain extent on the immediately preceding value. In practice, the most important effect is that zeroes tend to occur in runs in R. We can take advantage of this effect by representing each run of zeroes by a code indicating the length of the run. A second set of Huffman codes can be used for values immediately following a run of zeroes, since the next value cannot be another run of zeroes. [BW94, p. 13]

### 2.2.4 *Block-sorting compression*

So far, we have presented the compression methods source coding, run-length encoding and move-to-front transform. This is sufficient to present the basic scheme of "block-sorting compression" [BW94]. Given a string $S$, the first step is to compute the BWT $\mathsf{L}$ of $S$. Afterwards, we apply the move-to-front transform and run-length encoding of ones to $\mathsf{L}$. As final step, the string $\mathsf{rle}_1(\mathsf{mtf}(\mathsf{L}))$ is encoded using source encoding. The full process chain is depicted in Figure 2.6. The method is called

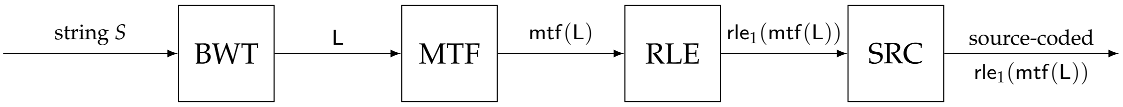

string $S$
BWT
L
MTF
mtf(L)
RLE
$\mathsf{rle}_1(\mathsf{mtf}(\mathsf{L}))$
SRC
source-coded
$\mathsf{rle}_1(\mathsf{mtf}(\mathsf{L}))$

**Figure 2.6:** Process chain of block-sorting compression.

| $i$ | $\mathsf{L}[i]$ | $S[\mathsf{SA}[i]..\mathsf{SA}[i]+60]$ |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| 426 | `i` | `st also der Wahlspruch der Aufklärung.` |
| 427 | `i` | `st das Unvermögen, sich seines Verstandes ohne Leitung eines` |
| 428 | `i` | `st der Ausgang des Menschen aus seiner selbst verschuldeten U` |
| 429 | `i` | `st diese Unmündigkeit, wenn die Ursache derselben nicht am Ma` |
| 430 | `b` | `st verschuldeten Unmündigkeit. Unmündigkeit ist das Unvermöge` |
| 431 | `r` | `standes ohne Leitung eines anderen zu bedienen. Selbstverschu` |
| 432 | `r` | `standes zu bedienen! ist also der Wahlspruch der Aufklärung.` |
| 433 | `r` | `standes, sondern der Entschließung und des Mutes liegt, sich` |
| 434 | `b` | `stverschuldet ist diese Unmündigkeit, wenn die Ursache dersel` |
| ⋮ | ⋮ | ⋮ |

**Figure 2.7:** Excerpt from the BWT and the sorted suffixes from the first paragraph of the essay "Beantwortung der Frage: Was ist Aufklärung?" from Immanuel Kant [Kan84]. Each suffix prefixed by `sta` is preceded by the character `r`, suffixes prefixed by `st` strongly tend to be preceded by the characters `i`, `r` and `b`.

"block-sorting" because it processes a sequence block-wise, mainly because of performance reasons. As each step is reversible, block-sorting compression is a lossless data compression method.

We now want to explain why block-sorting compression works well in practice. First of all, for a normal text, the BWT tends to be highly clustered and also contains a locally skew character distribution: in a normal text, similar contexts normally tend to be succeeded (but also preceded) by similar characters. As the lexicographic sorting of text rotations brings those similar contexts together, small portions of the BWT tend to contain similar characters, see Figure 2.7 for an example of this effect. This results in a high clustering and also a locally skew character frequency.

The move-to-front transform then converts the locally skew character frequency to a globally skew character frequency. Run-length encoding of ones then compresses the string because of the high clustering of the BWT. Finally, as the run-length encoded move-to-front transform of the BWT contains a skew character frequency, source encoding is able to compress this string even more.

In Section 4.4 we will see that block-sorting compression is a competitive method for lossless data compression, although other compressors exist which work better on different kinds of data. The main problem of block-sorting compression is performance: suffix array construction is a very expensive operation, resulting in slow compression speed. Decompression speed also is a bit slow, mainly because the backward steps required for BWT retransformation jump in a pseudo-random fashion in the BWT, resulting in bad caching properties and therefore long access times on modern computer architectures. To this end, despite the very popular data compressor `bzip2` [Sew96], block-sorting compression lost relevance in lossless data compression. However, there have already been papers that address those issues [FK17; KKP12], so hopefully, this very elegant data compressor will reenter the big stage of lossless data compression one day.

## 2.3 WAVELET TREES, RANK, SELECT AND BALANCED PARENTHESES

In this section, we present data structures which allow one to perform operations of sequence analysis using a BWT. These data structures are essential for sequence analysis, and will be used intensively when tunneling is applied to data structures from sequence analysis in Chapter 5. As a reminder, the basic operation necessary to perform e.g. exact string matching is the backward step (Definition 2.4), that is, for a character $c$ and some integer $i \in [1, n]$, compute

$$\mathsf{C}_\mathsf{L}[c] + \mathsf{rank}_\mathsf{L}(c, i).$$

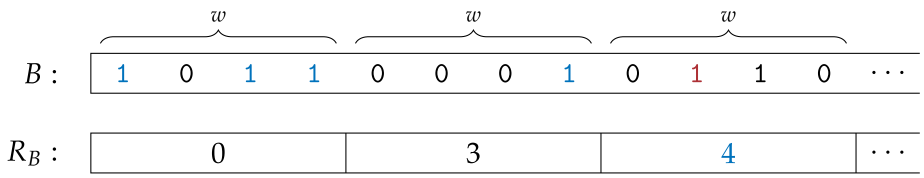
While it is easy to compute $\mathsf{C}_\mathsf{L}[c]$ using a preprocessing step, answering $\mathsf{rank}$-queries is not that straightforward. A trivial solution would be to set up a two-dimensional table $R$ such that $R[c][i] = \mathsf{rank}_\mathsf{L}(c, i)$. This table would require $\sigma \cdot n \cdot \log_2(n)$ bits of space, which is way too much for practical applications.

### 2.3.1 *Rank*

To present a more space-saving alternative for $\mathsf{rank}$-computation, we start with the simple case in which the alphabet $\Sigma$ consists of the two letters `0` and `1`. This implies that we work on bit-vectors, so in the following, we denote by $B$ a bit-vector of size $n$. The first such approach was presented in [Jac89], we herein will present a similar approach.

If the bit-vector $B$ is not too long, the special instruction `popcount` [War13, pp. 81–96] can accomplish the task of $\mathsf{rank}$-queries. Let $w$ be the word width of the operating system (typical values of $w$ are 32 for 32-bit systems or 64 for 64-bit systems). The instruction `popcount` for an integer word $x$ is defined as follows:

$$\mathtt{popcount}(x) := \mathsf{rank}_{\mathsf{bin}(x)}(\mathtt{1}, |\mathsf{bin}(x)|)$$



$$\mathsf{rank}_B(1,10) = R_B[1 + 10 \operatorname{div} 4] + \mathsf{rank}_{B[9..12]}(1, 10-8) = R_B[3] + \mathsf{rank}_{0110}(1,2)$$

**Figure 2.8:** Example of a two-way rank computation with word width $w = 4$.

To keep it simple, `popcount` computes the number of bits set to 1 in an integer word $x$. Using `popcount`, a $\mathsf{rank}_B(1,i)$-query for bit-vectors $B$ with $n \leq w$ can be computed as follows: put $B$ into an integer word $x$ and set the bits $B[i+1..n]$ to zero by shifting the value $x$ exactly $n - i$ positions to the right. Then, applying `popcount` yields the desired result, as can be seen in the following example:

$$\mathsf{rank}_{0110}(1,2) = \mathtt{popcount}(0110_2 \operatorname{div} 2^2) = \mathtt{popcount}(01_2) = 1$$

So, in the case that $n \leq w$, we have an $O(1)$-time solution to solve $\mathsf{rank}_B(1,i)$ queries. In the case $n > w$, we can precompute a table $R_B$ of size $\frac{n}{w}$, defined as follows:

$$R_B[d] := \mathsf{rank}_B(1, (d-1) \cdot w)$$

If each entry uses $w$ bits (if $\log_2(n) > w$ would hold, one would not be able to address all bits of $B$), the table requires $\frac{n}{w} \cdot w = n$ bits. To answer $\mathsf{rank}_B(1,i)$-queries, we first identify the block in which $i$ is, that is, we compute $d := 1 + (i-1) \operatorname{div} w$. Afterwards, the rank can be computed by adding the number of ones before the $j$-th block using table $R_B$ and the number of ones in the corresponding prefix of the block:

$$\mathsf{rank}_B(1,i) = R_B[d] + \mathsf{rank}_{B[(d-1)\cdot w+1..d\cdot w]}(1, i-(d-1)\cdot w)$$

The second `rank`-query then can be answered using the `popcount` instruction from above. An example of this 2-way computation is depicted in Figure 2.8. Thus, using $n$ bits and an $O(n)$ preprocessing step, we are able to answer $\mathsf{rank}_B(1,i)$-queries in $O(1)$ time. This also clearly gives an $O(1)$ time solution for $\mathsf{rank}_B(0,i)$-queries, as $\mathsf{rank}_B(0,i) = i - \mathsf{rank}_B(1,i)$ holds.

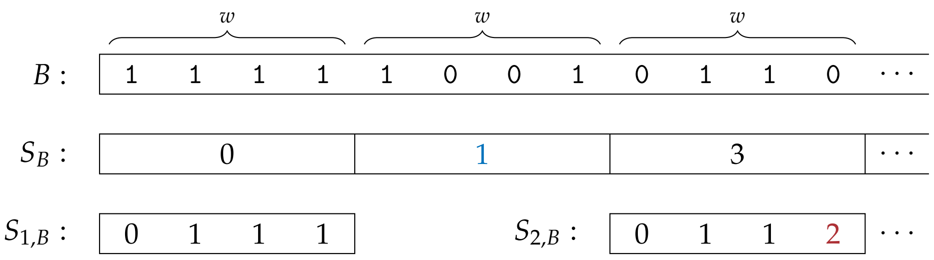
By a two-level division of the bit-vector $B$ into superblocks and blocks, it is possible to reduce the overhead of additional data structures for rank-queries to $\frac{1}{4} \cdot n$, see [Vig08] for more details.

### 2.3.2 *Select*

Another very useful operation is the select-function in bit-vectors. Analogously to `popcount`, there exists a function answering select-queries in an integer word $x$ very quickly [Vig08].

Suppose we do know the number $d$ of the $w$-block $B[(d-1) \cdot w + 1..d \cdot w]$ in which the $i$-th set bit in $B$ is. Then, using the select-query on integer words and the $R_B$-array, a $\mathsf{select}_B(1, i)$-query can be answered with the formula $\mathsf{select}_B(1, i) = (d-1) \cdot w + \mathsf{select}_{[(d-1) \cdot w+1..d \cdot w]}(1, i - R_B[d])$. In order to find the $w$-Block of the $i$-th set bit, as $R_B$ is monotonically increasing, a binary search can be used [CN09b]. This results in an $O(\log(\frac{n}{w}))$-solution for solving select-queries.

To improve the run-time of select-queries, one thus has to improve the run-time of finding the $w$-block in which the considered 1-bit can be found. This can be done using a two-level approach shown in Figure 2.9. First, we store the block of each $w$-th set bit in an array $S_B[d] := 1 + (\mathsf{select}_B(1, (d-1) \cdot w) - 1) \operatorname{div} w$. This array requires at most $\frac{n}{w} \cdot \log_2(\frac{n}{w})$ bits of space. Now, to find the block of the intermediate 1 bits, for each $S_B[d]$-entry, we store a pointer to another array $S_{d,B}[i] := 1 + (\mathsf{select}_B(1, (d-1) \cdot w + (i-1)) - 1) \operatorname{div} w - S_B[d]$ which for each intermediate set bit between the $(d-1) \cdot w$-th and the $d \cdot w$-th set bit stores the offset to the block of the $(d-1) \cdot w$-th set bit. The pointers require at most $\frac{n}{w} \cdot w = n$ bits, while the additional arrays need at most $\log_2(S_B[d] - S_B[d+1] + 1) + 1$ bits of space per entry because we



$$1 + (\mathsf{select}_B(1, 7) - 1) \operatorname{div} w = S_B[1 + (7-1) \operatorname{div} 4] + S_{1+(7-1) \operatorname{div} 4, B}[1 + 7 \bmod 4] = 3$$

**Figure 2.9:** Example of a two level block select structure with word width $w = 4$. The example shows that the 7-th set bit can be found in the third word block.

know that the offset must be a value with the upper bound $S_B[d] - S_B[d+1] + 1$. By concatenating the $S_{d,B}$-array into a bigger bit-vector $\tilde{S}$ (pointers are then replaced by starting positions of the different arrays in $\tilde{S}$), using Jensen's inequality [Jen06], the bit-vector $\tilde{S}$ requires at most

$$\begin{aligned} & w \cdot \sum_{d=1}^{n/w} \log_2(S_B[d] - S_B[d+1] + 1) + 1 \\ \leq\ & n + \frac{n}{w} \cdot \log_2 \left( \frac{\sum_{d=1}^{n/w} S_B[d] - S_B[d+1] + 1}{\frac{n}{w}} \right) \\ \leq\ & n + \frac{n}{w} \cdot \log_2 \left( \frac{n + \frac{n}{w}}{\frac{n}{w}} \right) \leq n + \frac{n}{w} \cdot \log_2(w+1) \leq 2 \cdot n \text{ bits.} \end{aligned}$$

Thus, overall we need at most $4 \cdot n$ bits of space to indicate the $w$-block of the $i$-th set bit using the following procedure: we access the $S_B[1 + (i-1) \operatorname{div} w]$-entry and follow the pointer into the bit-vector $\tilde{S}$. As the pointers $P$ are starting positions in $\tilde{S}$ and each array $S_{d,B}$ contains exactly $w$ entries, the bit width of an entry in $S_{d,B}$ is given by $\frac{P[2+(i-1) \operatorname{div} w] - P[1+(i-1) \operatorname{div} w]}{w}$. The block of the $i$-th set bit can be computed using $S_B[1 + (i-1) \operatorname{div} w] + S_{1+(i-1) \operatorname{div} w, B}[1 + i \bmod w]$. Then, using the select-query on integer words and the $R_B$-array, we can compute $\mathsf{select}_B(1, i)$.

Obviously, it is harder to design data structures for fast select-queries than for fast rank-queries. Also, in practice, rank-queries turn out to be faster than select-queries [Vig08]. We once more refer to [Vig08] for a more space-saving implementation of select-query data structures but now want to move on to rank- and select-queries on strings.

### 2.3.3 *Wavelet trees*

A simple way to introduce rank- and select-queries on a string $S$ of length $n$ is given by $\sigma$ bit-vectors of size $n$, where each bit-vector $B_c$ for a certain character $c \in \Sigma$ indicates if a position $i$ in $S$ contains the character $c$, that is, $B_c[i] = 1 \Leftrightarrow S[i] = c$. By adding rank and select support to each such bit-vector, we are able to compute rank- and select-queries fast using $\mathsf{rank}_S(c, i) = \mathsf{rank}_{B_c}(1, i)$ and $\mathsf{select}_S(c, i) = \mathsf{select}_{B_c}(1, i)$.

The problem with this solution is that it requires more than $\sigma \cdot n$ bits, which is too much for big data. Instead, we present a tree-based structure called wavelet tree to accomplish the same task. Wavelet trees were developed by Grossi et al. [GGV03], and since then have turned out to be extremely flexible and very useful in many applications, see [Nav14] for more information.

**Definition 2.11** (Wavelet tree)**.** Let $S$ be a string of length $n$ over alphabet $\Sigma$. Let $A \subseteq \Sigma$ be a subset of the alphabet $\Sigma$. We define by $S_A$ the string of length $\sum_{c \in A} \mathsf{rank}_S(c, n)$ obtained by removing all characters $c \notin A$ from string $S$, that is,

$$S_A[i] := S[\min\{\, j \in [1, n] \mid \sum_{c \in A} \mathsf{rank}_S(c, j) = i \,\}].$$

A tree decomposition of the alphabet $\Sigma$ is a binary tree $T = (V, E)$ with label function $\lambda : V \to \{\, A \mid A \subseteq \Sigma \,\}$ fulfilling the following conditions:

- $T$ has exactly $\sigma$ leaves whose disjoint union forms the alphabet $\Sigma$, that is, $\dot{\bigcup}_{\substack{v \in V \\ v \text{ is a leaf}}} \lambda(v) = \Sigma$.
- Each inner node $v \in V$ has exactly two children $u_l$ and $u_r$. Additionally, for each inner node $v$ with children $u_l$ and $u_r$, $\lambda(v) = \lambda(u_l) \,\dot{\cup}\, \lambda(u_r)$ holds.

Now, let $T = (V, E)$ be a tree decomposition of the alphabet $\Sigma$. A ($T$-shaped) wavelet tree $W(T, S) = (V, E)$ consists of the tree $T$ and $|V| - \sigma$ bit-vectors. For each inner node $v \in V$ with left child $u_l$ and right child $u_r$ the bit-vector $B_v$ of length $|S_{\lambda(v)}|$ is defined as follows:

$$B_v[i] := \begin{cases} 0 & \text{, if } S_{\lambda(v)}[i] \in \lambda(u_l) \\ 1 & \text{, if } S_{\lambda(v)}[i] \in \lambda(u_r) \end{cases}$$

An example of a wavelet tree can be found in Figure 2.10. Basically, a wavelet tree at each inner node splits a sequence in two subsequences. Each subsequence consists of the characters contained in the corresponding node of the alphabet decomposition tree. Moreover, a wavelet tree does not directly encode the sequence. Instead it encodes the sequence using a bit-vector $B_v$ which indicates if a symbol belongs to the left subtree ($B_v[i] = 0$) or to the right subtree ($B_v[i] = 1$).

The size of a wavelet tree strongly depends on the shape of the alphabet tree decomposition. For example, if each inner node of the tree decomposition splits its label (the remaining alphabet) into two almost equal sub-alphabets, each path from the root to a leaf has almost the same length. This special shaped wavelet tree is called a balanced wavelet tree, whose size can be approximated with $n \cdot \log \sigma$ bits.

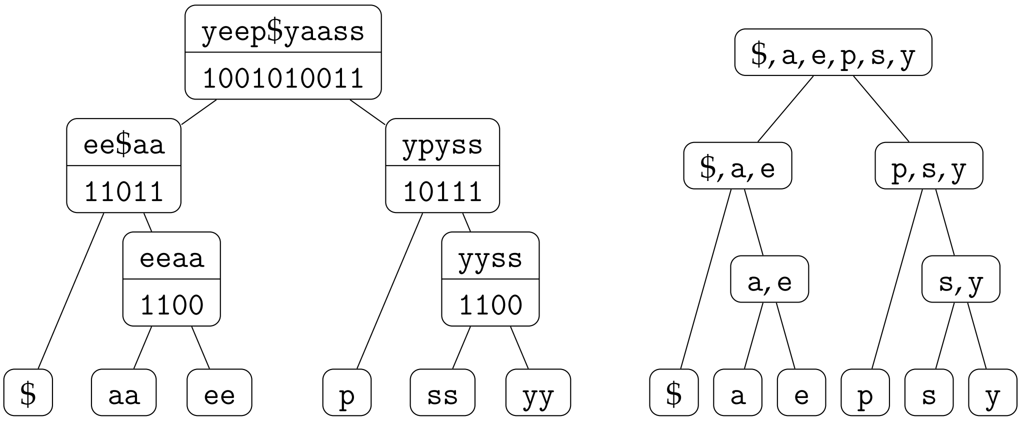


**Figure 2.10:** Wavelet tree of the string $S = \texttt{yeep\$yaass}$ (left) as well as the underlying tree decomposition of the alphabet $\Sigma = \{\$, \texttt{a}, \texttt{e}, \texttt{p}, \texttt{s}, \texttt{y}\}$ (right).

Therefore, a balanced wavelet tree has a size almost equal to the naive representation of the string $S$. As another example, if the tree decomposition is given by a Huffman tree [Huf52], we call the wavelet tree Huffman-shaped. As Huffman codes are optimal, the size of a Huffman-shaped wavelet tree can be approximated with $H(S) \cdot n$ bits, where $H(S)$ is the entropy of $S$.

By initializing rank- and select-support on each bit-vector in a wavelet tree[4], the tree enables efficient computation of rank-, select- and also access-queries[5] which will be described next.

*Answering* $\mathsf{rank}_S(c, i)$ *- queries*

The idea of answering rank-queries is based on the following observation. Let $v$ by an inner node of a wavelet tree with left child $u_l$ and right child $u_r$ such that $c \in \lambda(v)$ holds. Then,

$$\mathsf{rank}_{S_{\lambda(v)}}(c, i) = \begin{cases} \mathsf{rank}_{S_{\lambda(u_l)}}(c, \mathsf{rank}_{B_v}(0, i)) & \text{, if } c \in \lambda(u_l) \\ \mathsf{rank}_{S_{\lambda(u_l)}}(c, \mathsf{rank}_{B_v}(1, i)) & \text{, if } c \in \lambda(u_r) \end{cases} .$$

Thus, to answer rank-queries, one starts at the root node of the wavelet tree, and at each inner node checks if the character $c$ belongs to the left or right child. Then, one follows the edge to the appropriate child node and replaces $i$ with the value

[4] In practice, all bit-vectors are concatenated, so only one rank- and select-support is necessary. To access each bit-vector individually, the inner nodes in the tree decomposition then contain the start positions of the individual sequences in the common bit-vector.

[5] An access-query asks for the symbol at position $i$, i.e. $\mathsf{access}_S(i) := S[i]$.

$\mathsf{rank}_{B_v}(0,i)$ or $\mathsf{rank}_{B_v}(1,i)$ respectively. By repeating this procedure, once a leaf is reached, the desired rank-answer is given by the value of the variable $i$.

*Answering* $\mathsf{select}_S(c,i)$ *- queries*

Similar to rank-queries, there is an analogous observation for select-queries: Let $v$ by an inner node of a wavelet tree with left child $u_l$ and right child $u_r$ such that $c \in \lambda(v)$ holds. Then,

$$\mathsf{select}_{S_{\lambda(v)}}(c,i) = \begin{cases} \mathsf{select}_{B_v}(0, \mathsf{select}_{S_{\lambda(u_l)}}(c,i)) & \text{, if } c \in \lambda(u_l) \\ \mathsf{select}_{B_v}(1, \mathsf{select}_{S_{\lambda(ur)}}(c,i)) & \text{, if } c \in \lambda(u_r) \end{cases}.$$

In contrast to the top-down traversal in rank-queries, one starts at the leaf of the character $c$. Then, each step consists of an update of the value of $i$ to $\mathsf{select}_{B_v}(0,i)$ if the current node is a left child of the parent node $v$, or to $\mathsf{select}_{B_v}(1,i)$ if the current node is a right child of the parent node $v$. This induces a path from the leaf of character $c$ to the root node, and the desired answer of the select-query is given in the variable $i$.

*Answering* $\mathsf{access}_S(i)$ *- queries*

Once more, for access-queries, a recursive relation is given as follows: Let $v$ be an inner node of a wavelet tree with left child $u_l$ and right child $u_r$ such that $S[i] \in \lambda(v)$ holds. Then,

$$\mathsf{access}_{S_{\lambda(v)}}(i) = \begin{cases} \mathsf{access}_{S_{\lambda(u_l)}}(\mathsf{rank}_{B_v}(0,i)) & \text{, if } B_v[i] = 0 \\ \mathsf{access}_{S_{\lambda(ur)}}(\mathsf{rank}_{B_v}(1,i)) & \text{, if } B_v[i] = 1 \end{cases}.$$

Therefore, to answer access-queries, one starts at the root node of the tree. In each step, the variable $i$ is updated to $\mathsf{rank}_{B_v}(B_v[i],i)$. Afterwards, we follow the edge to the left child if $B_v[i] = 0$ holds or to the right child if $B_v[i] = 1$ holds. Once a leaf $v$ is reached, the desired character is found.

Because a wavelet tree supports rank-, select- and access-queries, it can be seen as a replacement of a traditional string. Once the wavelet tree of a string is set up, it is no longer necessary to keep the original string because the tree itself contains all necessary data. The run-time of the operations strongly depends on the length of the paths from the root node to the leaves. For example, in case of a balanced wavelet tree, each such path has length $\log \sigma$, and as each of the above described operations takes $O(1)$ processing time per inner node, all operations can be performed in $O(\log \sigma)$ time. In case of a Huffman-shaped wavelet tree, the timings may vary depending on how often a character occurs in the original sequence.

Aside from rank-, select- and access-queries, wavelet trees offer a plethora of other operations, see e.g. [Nav14] for an exhaustive list. We want to introduce one more operation which will turn out to be extremely useful in the following applications: the intervalsymbols-function. The function is kind of an extended rank-operation. For a given range $[i, j]$, $\mathsf{intervalsymbols}(i, j)$ computes the alphabet of $S[i..j]$ as well as the ranks of each character at position $i - 1$ and the ranks at position $j$. The function is defined as follows:

$$\mathsf{intervalsymbols}_S(i, j) := \{ \langle c, \mathsf{rank}_S(c, i-1), \mathsf{rank}_S(c, j) \rangle \mid c \in \Sigma \text{ occurs in } S[i..j] \}$$

The usefulness of the intervalsymbols-function can be seen when a BWT $\mathsf{L}$ of a string $S$ is encoded in a wavelet tree. Let $\omega$ be a substring of $S$, and let $[i, j]$ be the $\omega$-interval in $\mathsf{L}$. Calling intervalsymbols with the range $[i, j]$ then returns all preceding characters of $\omega$ in the original string. Furthermore, by modifying each triple $\langle c, lb, rb \rangle$ to $\langle c, \mathsf{C_L}[c] + lb + 1, \mathsf{C_L}[c] + rb \rangle$, we precisely obtain all $c\omega$-intervals contained in the original string. We therefore introduce another function getIntervals which automates this process and is shown in Algorithm 2.5.

The getIntervals-function can be used to enumerate all k-mer intervals[6] of $\mathsf{L}$: starting with the interval $[1, n]$, the first call of getIntervals generates all 1-mer intervals. If we repeat calling getIntervals for each 1-mer interval, all 2-mer intervals are obtained. Thus, by repeating this procedure, all k-mer intervals of a certain length $k$ can be obtained. This k-mer enumeration was presented in [BBO12], see also [Ohl13, pp. 323–328] for more details. The run-time of the intervalsymbols-function and thus of the getIntervals-function on a balanced wavelet tree is bound by $O(z \cdot \max\{\frac{\sigma}{z}, \log \sigma\})$ where $z$ is the number of distinct symbols in $S[i..j]$, see e.g. [Ohl13, p. 316].

The combination of a BWT and a wavelet tree turns out to be very powerful. For example, given a pattern $P$ of length $m$, the $P$-interval can be computed in $O(m \cdot \log \sigma)$-time. In conjunction with the suffix array, this allows one to perform exact string matching in $O(m \cdot \log \sigma + occ)$ time, see Algorithm 2.2. The combination of wavelet tree and BWT is often referred to as FM-index [FM05], despite the fact that

[6] A k-mer of a string $S$ is a subsequence of length $k$.

```
Data: BWT L of a string $S$ of length $n$ encoded as a wavelet tree, C-array $\mathsf{C_L}$, $\omega$-interval $[i, j]$.
Result: A set $M$ containing all $c\omega$-intervals, that is,
        $M := \{ \langle c, [lb, rb] \rangle \mid S$ contains $c\omega$ and $[lb, rb]$ is the range of the $c\omega$-interval in L $\}$.
$M \leftarrow \emptyset$
$\tilde{M} \leftarrow \mathsf{intervalsymbols_L}(i, j)$
foreach $\langle c, [lb, rb] \rangle \in \tilde{M}$ do
    $M \leftarrow M \cup \{ \langle c, [\mathsf{C_L}[c] + lb + 1, \mathsf{C_L}[c] + rb] \rangle \}$
return $M$
```

**Algorithm 2.5:** Implementation of the getIntervals function. A different implementation of getIntervals can be found in [Ohl13, p. 316].

its inventors used a different data structure to perform rank-queries. An FM-index is typically enriched with suffix array samples, allowing not only the performance of a fast backward search but also fast access to the current position in the underlying text. As the topics in this thesis have no necessity to access suffix array entries, we understand an FM-index as a BWT encoded in a balanced wavelet tree, enriched with the $\mathsf{C_L}$-array.

### 2.3.4 *Balanced parentheses sequences*

Balanced parentheses sequences are a special sort of sequences built over a two-symbol alphabet $\Sigma = \{(\,,)\}$. As the name suggests, these sequences correspond to parentheses sequences in which the number of opening parentheses corresponds to the number of closing parentheses. Moreover, the parentheses are "correct", that is, for any prefix of such a sequence, the number of opening parentheses must be greater or equal to the number of closing parentheses.

**Definition 2.12** (Balanced parentheses sequence)**.** A balanced parentheses sequence (BPS) is a sequence which is produced by the following inductive rules:

- the empty sequence $\varepsilon$ is a BPS.
- if $P$ is a BPS, then the sequence $(P)$ is a BPS.
- if $P$ and $Q$ are BPS, then the concatenation $PQ$ is a BPS.

An example of a balanced parentheses sequence is given in Figure 2.11. Those sequences are typically represented by a bit-vector, where a 1 corresponds to an opening and a 0 corresponds to a closing parenthesis. BPS are useful to e.g. represent the topology of a tree: starting at the root node of the tree, at each node write an opening parenthesis, write the BPS of all children from left to right, and then write

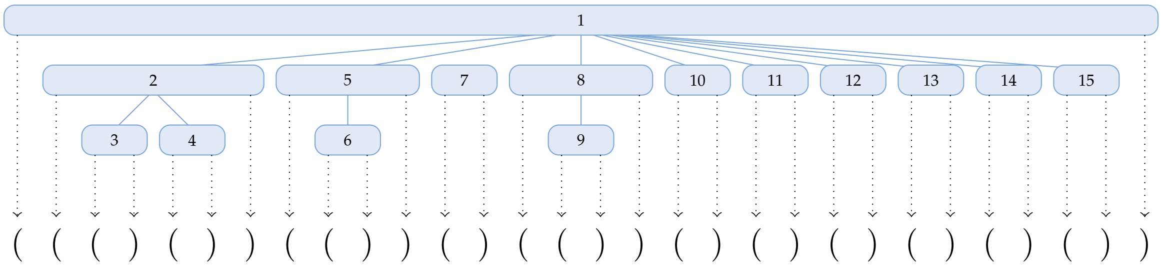


**Figure 2.11:** A tree and its tree topology represented using a balanced parentheses sequence. The nodes in the tree are numbered according to a pre-order traversal of the tree.

a closing parenthesis. By this construction, the opening and closing parentheses of a node enclose the BPS of its children. For more information on tree topology representations using BPS, see e.g. [Jac89] or [BBO17].

We now want to describe some basic operations on BPS. The first such operation for an opening parenthesis returns the position of its corresponding closing parenthesis.

**Definition 2.13** (findclose)**.** Let $P$ be a BPS of length $n$, and let $i \in [1, n]$ be an integer such that $P[i] = ''(''$ holds. The findclose-function is defined as follows:

$$\mathsf{findclose}_P(i) := \min\{\, j \in [i+1, n] \;\mid\; B[i..j] \text{ is a BPS } \}$$

As indicated above, to allow parent navigation in a BPS of a tree topology, for a given opening parenthesis at position $i$ one has to find the opening parenthesis $j < i$ such that the opening parenthesis at $j$ and its corresponding closing parenthesis enclose $i$ as close as possible. This special function is called the enclose-function, and is defined as follows:

**Definition 2.14** (enclose)**.** Let $P$ be a BPS of length $n$, and let $i \in [1, n]$ be an integer such that $P[i] = ''(''$. The enclose-function then is defined as follows:

$$\mathsf{enclose}_P(i) := (\max\{\, j \in [1, i-1] \;\mid\; P[j] = ''('' \text{ and } \mathsf{findclose}_P(j) > i\} \cup \{0\})$$

Similar to rank- and select-queries, it is possible to set up support structures requiring less than $n$ bits such that findclose- and enclose-queries can be computed in constant time. However, those support structures are considerably more complex

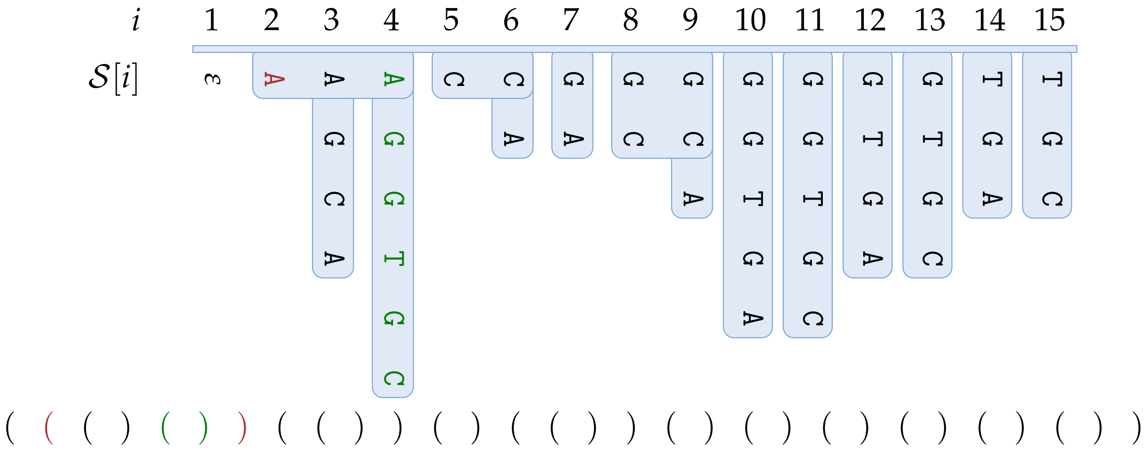


**Figure 2.12:** A list $\mathcal{S}$ of lexicographically sorted strings as well as the BPS of the prefix tree of the strings. Blue boxes in the strings indicate which string is a prefix of another, and thereby forms the prefix tree shape (see also Figure 2.11). To find the longest prefix of the 4-th string (colored green), one determines the 4-th opening parenthesis in the BPS, searches for the enclosing parentheses (red), and determines the rank of the opening parenthesis, so $\mathcal{S}[2]$ is the longest prefix of $\mathcal{S}[4]$.

and are beyond the scope of this thesis. We refer to [Jac89; Gea+06; SN10] for more information.

Instead, we want to describe a simple application of BPS and the enclose-method. Suppose we have a list $\mathcal{S}$ which contains lexicographically sorted strings. Given an integer $i$, we want to find the index $j$ such that $\mathcal{S}[j]$ is the longest prefix of $\mathcal{S}[i]$. Because the relation "is prefix of" is transitive, it forms the shape of a tree, as can be seen by comparing the Figures 2.11 and 2.12. By representing this tree topology with a balanced parenthesis sequence $P$, the index $j$ of the longest prefix of a string $\mathcal{S}[i]$ can be found using

$$j \leftarrow \mathsf{rank}_P(\text{"("}, \mathsf{enclose}_P(\mathsf{select}_P(\text{"("}, i))).$$

An explanation of the formula is given as follows: by selecting the $i$-th opening parenthesis in $P$, we enter the opening parenthesis of the $i$-th node, where the nodes of the tree are numbered according to a pre-order traversal of the tree (see Figure 2.11). By determining the position of the surrounding parentheses, we implicitly navigate to the parent node of the current node in the tree. Then, by ranking the number of open parentheses, we obtain the label of the current node, which then corresponds to the id of the longest prefix.

## 2.4 WHEELER GRAPHS

Wheeler graphs are a special class of graphs which are closely related to the BWT. First described by Gagie et al. [GMS17], the main purpose of Wheeler graphs is to describe other string data structures in the form of a special sort of graphs, allowing these data structures to be represented in a very compact form. In Chapter 3, we will use Wheeler graphs as a foundation of tunneling. There have been some results on the hardness of Wheeler graph recognition [GT19], but here we will mainly focus on applications of Wheeler graphs.

Different from the original definition of Wheeler graphs [GMS17], we herein present a slightly modified definition of Wheeler graphs, mainly because this definition is closer to the LF-mapping.

**Definition 2.15** (Wheeler graph)**.** Let $G = (V, E)$ be a directed multigraph such that each node in $G$ has in- and outdegree of at least 1, and let $\lambda : E \to \Sigma$ be a function asserting a label (character) to each edge. $G$ is a Wheeler graph if there is an ordering $\prec$ of the nodes such for any pair of edges $e = (u, v)$ and $e' = (u', v')$ the following monotonicity properties hold:

$$\begin{aligned} \lambda(e) < \lambda(e') \quad &\Rightarrow \quad v \preceq v', \\ \lambda(e) = \lambda(e') \text{ and } u \prec u' \quad &\Rightarrow \quad v \preceq v'. \end{aligned}$$

The following changes of Definition 2.15 in comparison to the classical Wheeler graph definition have been made:

- In the original definition, the graph was not allowed to be a multigraph, that is, no multiple edges between two nodes were allowed. On Page 59, we will see that it is useful to define Wheeler graphs as multigraphs.
- Definition 2.15 requires each node in the graph to have in- and outdegree of at least one. We added this condition because of the characteristics of the LF-mapping as a cyclic tour through the BWT, and in Section 5.2 will see that this condition brings a deep connection to another string based data structure.
- The original first monotonicity property required $v \prec v'$ to ensure that the incoming edges of any node $v$ all carry the same label. In the herein presented topics, we do not need such a condition, and therefore allow nodes with differently labeled incoming edges.

| $i$ | $\mathsf{L}[i]$ | $S[\mathsf{SA}[i]..n]$ |
|---|---|---|
| 1 | y | $ |
| 2 | e | asy$ |
| 3 | e | asypeasy$ |
| 4 | p | easy$ |
| 5 | $ | easypeasy$ |
| 6 | y | peasy$ |
| 7 | a | sy$ |
| 8 | a | sypeasy$ |
| 9 | s | y$ |
| 10 | s | ypeasy$ |

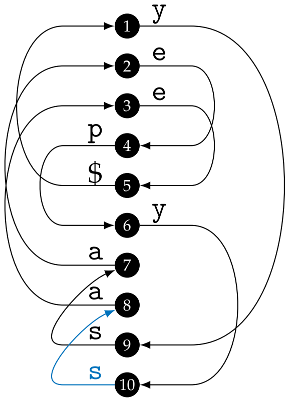


**Figure 2.13:** Suffix array and BWT (left) as well as BWT Wheeler graph (right) of the string $S =$ easypeasy$. The $i$-th node in the Wheeler graph corresponds to the suffix $S[\mathsf{SA}[i]..n]$, each outgoing edge of the $i$-th node in the graph carries the label $\mathsf{L}[i]$ and leads to the node labeled $\mathsf{LF}[i]$. As an example, the LF-mapping and L-entry at position 10 as well as the corresponding edge in the graph are colored blue. Parts of this image were already published in [BD19] © 2019 IEEE.

One should of course note that the recognition hardness results of Wheeler graphs [GT19] might not hold true for the modified definition, but as explained above, we want to focus on the applications and the operating principle of Wheeler graphs within this thesis.

Wheeler graphs can be used to represent a couple of string data structures, for example, tries [Fre60], de Bruijn graphs [DeB46; Iqb+12], wavelet trees [GGV03], or the BWT itself; see [GMS17] for a more exhaustive list. We will review some of these data structures and their Wheeler graph representations before giving more detail on Wheeler graphs.

We start with the simplest data structure that can be represented as a Wheeler graph: the BWT. Let $\mathsf{L}$ be a BWT of length $n$ with LF-mapping $\mathsf{LF}$. To represent $\mathsf{L}$ as a Wheeler graph, we create exactly $n$ nodes which are labeled from 1 up to $n$. Afterwards, for each node labeled with $i$, we create an outgoing edge labeled $\mathsf{L}[i]$ which points to the node with label $\mathsf{LF}[i]$.

**Corollary 2.16.** *Let* $\mathsf{L}$ *be a BWT of length $n$ with LF-mapping* $\mathsf{LF}$. *Let $G = ([1,n], E)$ be a graph whose edges are defined as follows:*

$$(i,j) \in E \;:\Leftrightarrow\; \mathsf{LF}[i] = j \quad \textit{with label function} \quad \lambda((i,j)) := \mathsf{L}[i]$$

*Then, using the node order $i \prec j \;:\Leftrightarrow\; i < j$, $G$ is a Wheeler graph.*

*Proof.* $G$ certainly is a graph where each node has in- and outdegree of at least one. Let $M$ be the BWT matrix of $\mathsf{L}$, and let $e = (i, \mathsf{LF}[i])$ and $e' = (j, \mathsf{LF}[j])$ be two edges of $G$.

If $\lambda(e) < \lambda(e')$ holds, $\mathsf{L}[i] < \mathsf{L}[j]$ is implied, and therefore, the matrix row $M_{\mathsf{LF}[i]} = \mathsf{L}[i]M_i[1..n-1]$ is lexicographically smaller than the row $M_{\mathsf{LF}[j]} = \mathsf{L}[j]M_j[1..n-1]$. As the rows of $M$ are sorted lexicographically, $\mathsf{LF}[i] < \mathsf{LF}[j]$ is implied.

If $\lambda(e) = \lambda(e')$ and $i < j$ holds, $\mathsf{L}[i] = \mathsf{L}[j]$ as well as $M_i \leq_{\mathsf{lex}} M_j$ holds. Consequently, for the rows $M_{\mathsf{LF}[i]} = \mathsf{L}[i]M_i[1..n-1]$ and $M_{\mathsf{LF}[j]} = \mathsf{L}[j]M_j[1..n-1]$ the relation $M_{\mathsf{LF}[i]} \leq_{\mathsf{lex}} M_{\mathsf{LF}[j]}$ must hold. Because of the lexicographic order of the rows in $M$, this implies $\mathsf{LF}[i] \leq \mathsf{LF}[j]$. □

### 2.4.1 *De Bruijn graphs*

Another data structure that can be represented as a Wheeler graph is given by de Bruijn graphs. De Bruijn graphs were originally invented to solve combinatorial problems [DeB46]. Nowadays, de Bruijn graphs are also used in bioinformatics, for example for genome assembly [IW95] or to express variations between different strings [Iqb+12]. In this thesis, a special class of string-related de Bruijn graphs are considered. Given a string $S$ of length $n$ and an order $k \in [1, n]$, the nodes of the graph correspond to the set of k-mers of the cyclic string of $S$. Each time two k-mers $x$ and $y$ overlap by $k-1$ characters in the cyclic string of $S$, an edge from $x$ to $y$ is drawn. The following definition was already used in [Bai+20].

**Definition 2.17** (De Bruijn graph)**.** Let $S$ be a string of length $n$ and $k \in [1, n]$. If we concatenate the k-mer prefix of $S$ to $S$ itself, then we obtain the string $Z_k(S) := S[1..n]S[1..k]$, which we call the *k*-cyclic string of $S$. Furthermore, the set of all k-mers of $Z_k(S)$ is

$$\mathcal{K} := \{\ Z_k(S)[i..i+k-1] \ \mid\ i \in [1, n]\ \}.$$

The de Bruijn graph $G_k(S) = (\mathcal{K}, E)$ of order $k$ is a directed multigraph, where the multiset of edges is defined as (the superscript $m$ denotes the multiplicity of an edge):

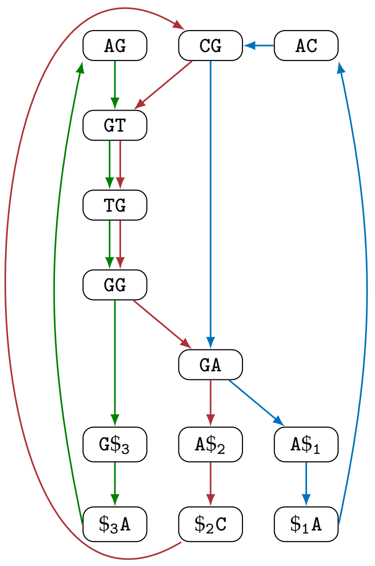
$$E := \{\ (x[1..k], x[2..k+1])^m \ \mid\ x \in \Sigma^{k+1} \text{ occurs exactly } m \text{ times in } Z_k(S)\}.$$

Multiple de Bruijn graphs $G_k(S_1) = (\mathcal{K}_1, E_1), \dots, G_k(S_m) = (\mathcal{K}_m, E_m)$ of the same order $k$ can be combined to a de Bruijn graph $G_k(S_1, \dots, S_m) = (\mathcal{K}, E)$ by merging the k-mer sets and the set of edges, that is, $\mathcal{K} := \bigcup_{i=1}^{m} \mathcal{K}_i$ and $E := \biguplus_{i=1}^{m} E_i$.

An example of a de Bruijn graph can be found in Figure 2.14. One should note that, independent of the order $k$, each de Bruijn graph $G_k(S)$ contains exactly $n$ edges. To represent a de Bruijn graph as a Wheeler graph, one performs the following steps:

1. label each edge $(x, y)$ with the character $x[1]$.
2. reverse each edge, that is, replace an edge $(x, y)$ with the edge $(y, x)$.
3. set up a list of lexicographically sorted k-mers.
4. replace the label of each node with the lexicographic rank of the corresponding k-mer in the list.

Similar to Corollary 2.16, it is easy to show that the arising graph is a Wheeler graph. The main problem of the construction is that the direction of edges is reversed. The reversal is necessary because a BWT (as well as a Wheeler graph) is traversed



| $k$-mer | lex. rank |
|---|---|
| $1A | 1 |
| $2C | 2 |
| $3A | 3 |
| A$1 | 4 |
| A$2 | 5 |
| AC | 6 |
| AG | 7 |
| CG | 8 |
| G$3 | 9 |
| GA | 10 |
| GG | 11 |
| GT | 12 |
| TG | 13 |

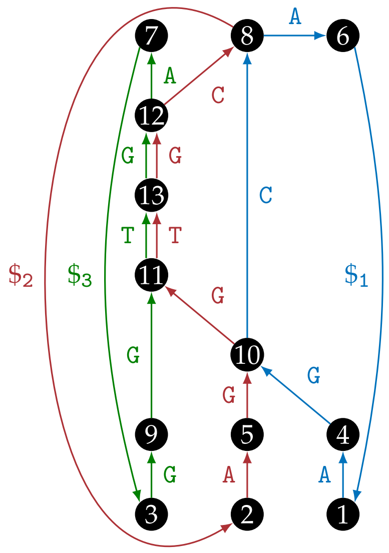


**Figure 2.14:** Combined de Bruijn graph of order $k = 2$ for the strings $S_1 =$ ACGA\$, $S_2 =$ CGTGGA$\$_2$ and $S_3 =$ AGTGG$\$_3$ (left), list of k-mers and lexicographic ranks of $S_1$, $S_2$, $S_3$ (middle), corresponding Wheeler graph with reversed edges and edge labels (right). The Wheeler graph is made by replacing the k-mers in the nodes of the de Bruijn graph by their lexicographic ranks, reversing edges, and labeling each edge with the first character of the k-mer it originated from.

using backward steps, while a de Bruijn graph is traversed using a “forward navigation”. The problem can easily be fixed by considering the de Bruijn graph of the reversed string(s): there is a one to one correspondence between k-mers and reversed k-mers, so the undirected de Bruijn graph of a string $S$ and the undirected de Bruijn graph of the reversed string $S$ are isomorphic. In the directed case, the graphs still are isomorphic except that the edges are reversed. Now, considering the Wheeler graph of the reversed de Bruijn graph, one notes that the edges are reversed once again and therefore correctly represent the edges in the original de Bruijn graph.

### 2.4.2 *Tries*

Another popular string data structure we want to present here is given by a tree-like dictionary of multiple strings, typically referred to as a trie [Fre60]. Tries are used in many applications. For example, the “Swiss army knife” of sequence analysis, the suffix tree [Wei73], is a special form of a trie built from all suffixes of a given string $S$. Tries can be enhanced with “failure links” which are useful for finding all occurrences of a set of patterns in a given string $S$ [AC75]. We will introduce failure links later in Section 5.4.

**Definition 2.18** (Trie)**.** Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a set of null-terminated strings. The trie $\mathcal{T}(S_1, \ldots, S_m) = (V, E)$ with label function $\lambda : E \to \Sigma$ is a directed rooted tree satisfying the following conditions:

- No two outgoing edges of a node carry the same label: if $e = (u, v)$ and $e' = (u, v')$ are two edges in the graph and $v \neq v'$, then $\lambda(e) \neq \lambda(e')$ must hold.

- There is a one to one correspondence between leaves of the trie and strings of the set $\mathcal{S}$: $S \in \mathcal{S}$ if and only if a path $p = (u_1, \ldots, u_{|S|+1})$ from the root node to a leaf of $\mathcal{T}(S_1, .., S_m)$ exists such that $S = \lambda((u_1, u_2)) \cdots \lambda((u_{|S|}, u_{|S|+1}))$.

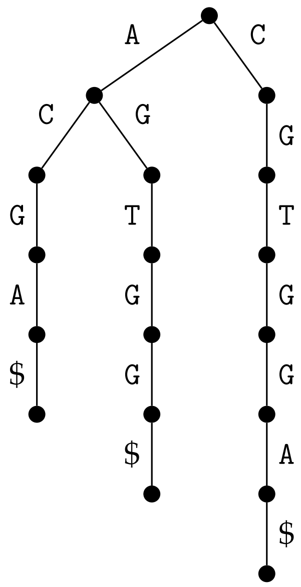
Let $u \in V$ be a node in the trie, and let $u_1, u_2, \ldots, u_i$ be the nodes visited on the unique path from the root to $u$. We denote by $\lambda(u) := \lambda((u_1, u_2)) \cdots \lambda((u_{i-1}, u_i))$ the node label of the node $u$, and define $\lambda(u) := \varepsilon$ in the case that $u$ is the root.

An example of a trie can be found in Figure 2.15. The representation of a trie using a Wheeler graph is quite similar to the representation of de Bruijn graphs, but has two major differences: instead of reversing the edges, one reverses the node labels.

Second, because the herein defined Wheeler graphs have to be cyclic, but tries are acyclic, a little workaround is needed to make tries cyclic. The full procedure is as follows:

1. set up a list of lexicographically sorted reversed node labels of all inner nodes.
2. label each inner node with the lexicographic rank of its reversed node label.
3. redirect the endpoints of edges pointing to leaves to the root node.
4. remove all isolated nodes (former leaves).

Using this little trick, a trie can be made cyclic and therefore can be represented as a Wheeler graph. Despite the modified representation of tries as Wheeler graphs, we once more refer to Corollary 2.16 and also [GMS17] that the above conversion produces a Wheeler graph–the main arguments still stay the same, and the redirection of edges from leaves to the root node do not change the order because each edge is labeled with the smallest character $\$ \in \Sigma$ and points to the lexicographically smallest node.



| rev. node label | lex. rank |
|---|---|
| $\varepsilon$ | 1 |
| A | 2 |
| AGCA | 3 |
| AGGTGC | 4 |
| C | 5 |
| CA | 6 |
| GA | 7 |
| GC | 8 |
| GCA | 9 |
| GGTGA | 10 |
| GGTGC | 11 |
| GTGA | 12 |
| GTGC | 13 |
| TGA | 14 |
| TGC | 15 |

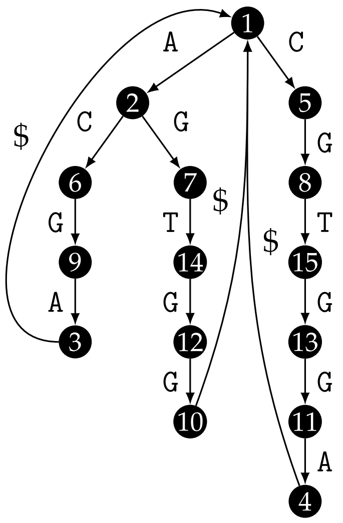


**Figure 2.15:** Trie of the strings $S_1 = \mathtt{ACGA\$}$, $S_2 = \mathtt{CGTGGA\$_2}$ and $S_3 = \mathtt{AGTGG\$_3}$ (left), list of reversed inner node labels and lexicographic ranks (middle), corresponding Wheeler graph (right). The Wheeler graph arises by labeling each node with the lexicographic rank of its reversed label, removing the leaves and moving the endpoints of edges pointing to former leaves to the root node.

### 2.4.3 *Succinct graph representation*

After seeing some examples of Wheeler graphs, the final topic of this section will be how Wheeler graphs can be represented succinctly. To this end, a string $\mathsf{L}$ behaving similar to a classical BWT is used. Additionally, two bit-vectors are used to represent the in- and outdegree of the nodes in the graph. Before going into more detail, we want to give the concrete definition of such a succinct representation.

**Definition 2.19** (Succinct Wheeler graph representation)**.** Let $G = (V, E)$ be a Wheeler graph with $n$ nodes and $m$ edges, label function $\lambda : E \to \Sigma$ and node order $u_1 \prec \cdots \prec u_n$. Denote by $\mathsf{C}_{\mathsf{out}}(u_i)$ the number of outgoing edges of nodes which are strictly smaller than $u_i$, that is, $\mathsf{C}_{\mathsf{out}}(u_i) := \sum_{j=1}^{i-1} \mathsf{deg}_{\mathsf{out}}(u_j)$.

A succinct Wheeler graph representation of $G$ is given by a string $\mathsf{L}$ of size $m$ and two bit-vectors $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ of size $m + 1$ fulfilling the following conditions:

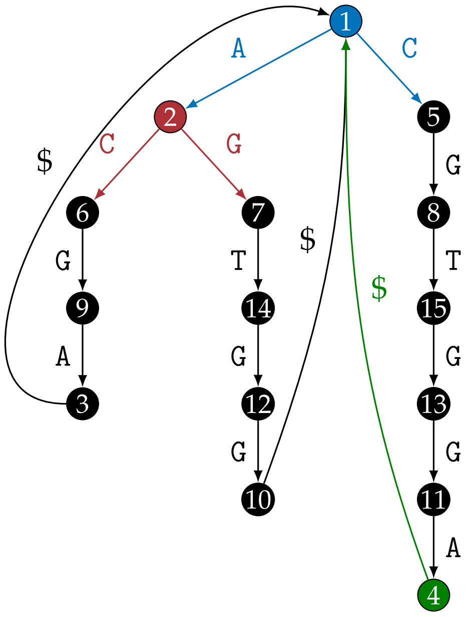
- For any node $u_i$, the substring $\mathsf{L}[\mathsf{C}_{\mathsf{out}}(u_i) + 1..\mathsf{C}_{\mathsf{out}}(u_i) + \mathsf{deg}_{\mathsf{out}}(u_i)]$ contains the labels of all outgoing edges of $u_i$, i.e. the following multisets are equal: $\{\mathsf{L}[\mathsf{C}_{\mathsf{out}}(u_i) + 1], \ldots, \mathsf{L}[\mathsf{C}_{\mathsf{out}}(u_i) + \mathsf{deg}_{\mathsf{out}}(u_i)]\} = \{\ \lambda((u_i, w))\ \mid\ (u_i, w) \in E\ \}$.

- The bit-vector $D_{\mathsf{out}}$ contains the reversed unary encoded out-degrees of all nodes in the underlying node order and is terminated by a $\mathtt{1}$, that is, $D_{\mathsf{out}} = \mathtt{1}\mathtt{0}^{\mathsf{deg}_{\mathsf{out}}(u_1)-1}\mathtt{1}\mathtt{0}^{\mathsf{deg}_{\mathsf{out}}(u_2)-1} \ldots \mathtt{1}\mathtt{0}^{\mathsf{deg}_{\mathsf{out}}(u_n)-1}\mathtt{1}$.

- The bit-vector $D_{\mathsf{in}}$ contains the reversed unary encoded in-degrees of all nodes in the underlying node order and is terminated by a $\mathtt{1}$, that is, $D_{\mathsf{out}} = \mathtt{1}\mathtt{0}^{\mathsf{deg}_{\mathsf{in}}(u_1)-1}\mathtt{1}\mathtt{0}^{\mathsf{deg}_{\mathsf{in}}(u_2)-1} \ldots \mathtt{1}\mathtt{0}^{\mathsf{deg}_{\mathsf{in}}(u_n)-1}\mathtt{1}$.

The string $\mathsf{F}$ denotes the string obtained by sorting the characters of $\mathsf{L}$.

At first glance, Definition 2.19 looks a bit complicated, so we want to explain the definition in a constructive way. To build the string $\mathsf{L}$, one traverses all the nodes of the graph in the underlying order while appending the labels of outgoing edges to $\mathsf{L}$. Similarly, to build the bit-vectors $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$, one traverses all nodes in the underlying order and appends the strings $\mathtt{1}\mathtt{0}^{\mathsf{deg}_{\mathsf{out}}(u)-1}$ respectively $\mathtt{1}\mathtt{0}^{\mathsf{deg}_{\mathsf{in}}(u)-1}$ to the bit-vectors, before appending a terminating 1-bit at the end. An example of a succinct representation of a Wheeler graph can be found in Figure 2.16. Note that the succinct representation of a Wheeler graph for a normal BWT (see Figure 2.13) consists of the normal BWT as well as two bit-vectors filled only with ones. Because



| $i$ | L[$i$] | $D_{\mathsf{out}}[i]$ | $D_{\mathsf{in}}[i]$ | F[$i$] |
|---|---|---|---|---|
| 1 | A | 1 | 1 | $ |
| 2 | C | 0 | 0 | $ |
| 3 | G | 1 | 0 | $ |
| 4 | C | 0 | 1 | A |
| 5 | $ | 1 | 1 | A |
| 6 | $ | 1 | 1 | A |
| 7 | G | 1 | 1 | C |
| 8 | G | 1 | 1 | C |
| 9 | T | 1 | 1 | G |
| 10 | T | 1 | 1 | G |
| 11 | A | 1 | 1 | G |
| 12 | $ | 1 | 1 | G |
| 13 | A | 1 | 1 | G |
| 14 | G | 1 | 1 | G |
| 15 | G | 1 | 1 | G |
| 16 | G | 1 | 1 | T |
| 17 | G | 1 | 1 | T |
| 18 |  | 1 | 1 |  |

**Figure 2.16:** Succinct representation of a Wheeler graph. The nodes and edges in the graph correspond to colored entries in L, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$. Colored entries in F correspond to the endpoints of the colored edges in the graph.

those additional bit-vectors in this case do not represent any further information, one typically omits them when dealing with a normal BWT.

We want to note that Definition 2.19 differs from other definitions on succinct representations of Wheeler graphs like the one in [GMS17]. In other definitions, nodes with an in- or outdegree of zero are allowed. To represent such zero-degrees, one has to encode them using the string `1`, so a degree of 1 has to be represented using `10`, a degree of 2 with `100`, and so on. Since we defined Wheeler graphs to be graphs where each node has an in- and outdegree of at least 1, we do not need to encode a zero-degree and therefore are able to save one bit per degree in the representation. This reduces the size of the bit-vectors from $n + m$ to $m$, excluding the termination bit.

The representation is edge-based: a position $i \in [1, m]$ in L specifies the starting point of an outgoing edge, while a position $i \in [1, m]$ in F specifies the endpoint of an incoming edge. The bit-vectors $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ serve as a mapping between edges and nodes, as each 1-bit in the vectors corresponds to a node in the graph. Unsurprisingly, if $i \in [1, m]$ is the starting point of an edge in L, the endpoint of the

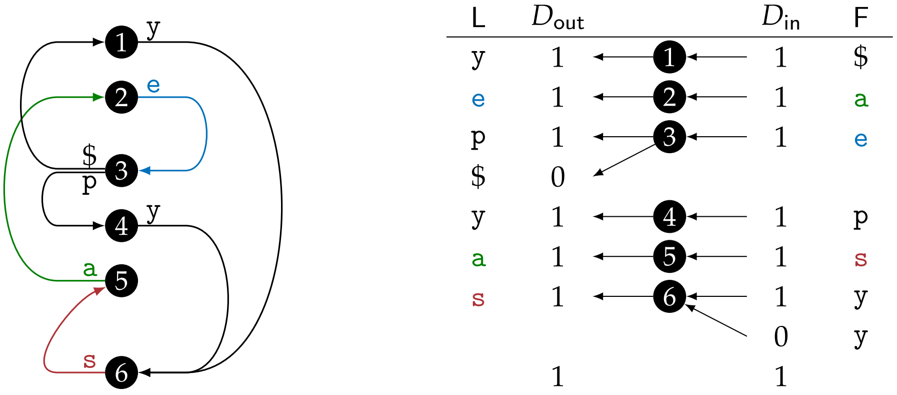


**Figure 2.17:** Connection between L and $D_{\text{out}}$ as well as F and $D_{\text{in}}$. The left-hand side shows a Wheeler graph, the right-hand side a succinct representation of the graph. On the right it is shown that $D_{\text{in}}$ maps the endpoints of edges to nodes, while $D_{\text{out}}$ maps nodes to the starting points of edges. Parts of this image were already published in [BD19] © 2019 IEEE.

edge in F can be found using a BWT backward step, that is, the endpoint can be computed using $\mathsf{C}_{\mathsf{L}}[i] + \mathsf{rank}_{\mathsf{L}}(\mathsf{L}[i], i)$. Moreover, if $i \in [1, m]$ is the starting point of an edge in L, its start node can be computed using $\mathsf{rank}_{D_{\text{out}}}(1, i)$. Similarly, the node to which the endpoint of an edge $i \in [1, m]$ in F is connected can be computed using $\mathsf{rank}_{D_{\text{in}}}(1, i)$. To put this differently, the bit-vector $D_{\text{out}}$ can be used to map nodes to the starting points of edges in L, while the bit-vector $D_{\text{in}}$ is used to map endpoints of edges in F to connected nodes, see also Figures 2.16 and 2.17. We are now ready to give a navigation theorem in Wheeler graphs.

**Theorem 2.20.** *Let $G = (V, E)$ be a Wheeler graph with node order $u_1 \prec \cdots \prec u_n$, and let* L*, $D_{\text{out}}$ and $D_{\text{in}}$ be a succinct representation of $G$. Furthermore, let $(u_i, u_j) \in E$ be an edge with label $c := \lambda((u_i, u_j))$, and define $\mathsf{C}_{\text{in}}(u_j) := \sum_{k=1}^{j-1} \mathsf{deg}_{\text{in}}(u_k)$.*

*(i) The string $\mathsf{L}[\mathsf{select}_{D_{\text{out}}}(1, i)..\mathsf{select}_{D_{\text{out}}}(1, i+1) - 1]$ contains precisely all labels of outgoing edges of the node $u_i$.*

*(ii) There exists an integer $k \in [\mathsf{select}_{D_{\text{out}}}(1, i)..\mathsf{select}_{D_{\text{out}}}(1, i+1) - 1]$ such that $\mathsf{L}[k] = c$ and $\mathsf{C}_{\mathsf{L}}[c] + \mathsf{rank}_{\mathsf{L}}(c, k) \in [\mathsf{C}_{\text{in}}(u_j) + 1, \mathsf{C}_{\text{in}}(u_j) + \mathsf{deg}_{\text{in}}(u_j)]$.*

*(iii) For any $k \in [\mathsf{C}_{\text{in}}(u_j) + 1, \mathsf{C}_{\text{in}}(u_j) + \mathsf{deg}_{\text{in}}(u_j)]$, $\mathsf{rank}_{D_{\text{in}}}(1, k) = j$ holds.*

*Proof.*

(i) By the definition of $D_{\text{out}}$ in Definition 2.19, using the notation from the same definition, $\mathsf{select}_{D_{\text{out}}}(1, i) = \mathsf{C}_{\mathsf{D}_{\text{out}}}(u_i) + 1$ and $\mathsf{select}_{D_{\text{out}}}(1, i+1) - 1 = \mathsf{C}_{\mathsf{D}_{\text{out}}}(u_i) + \mathsf{deg}_{\text{out}}(u_i)$ must hold. The statement then directly follows from Definition 2.19.

(ii) Define $[lb, rb] := [\mathsf{select}_{D_{\mathsf{out}}}(1, i), \mathsf{select}_{D_{\mathsf{out}}}(1, i+1) - 1]$. The statement can be shown by the following two inequalities:

$$\mathsf{C_L}[c] + \mathsf{rank_L}(c, lb - 1) \leq \mathsf{C_{in}}(u_j) + \mathsf{deg_{in}}(u_j) \tag{2.1}$$

$$\mathsf{C_L}[c] + \mathsf{rank_L}(c, rb) > \mathsf{C_{in}}(u_j) \tag{2.2}$$

For Inequality (2.1), we repeat the properties from Definition 2.15:

$$\begin{aligned} \lambda(e) < \lambda(e') \quad &\Rightarrow \quad v \preceq v', \\ \lambda(e) = \lambda(e') \text{ and } u \prec u' \quad &\Rightarrow \quad v \preceq v'. \end{aligned}$$

The value $\mathsf{C_L}[c]$ identifies the number of edges in $G$ with a lexicographically smaller label than $c$, so by the first monotonicity property, those edges must lead to nodes $v \in V$ with $v \preceq u_j$. Analogously, the value $\mathsf{rank_L}(c, lb - 1)$ includes the number of edges with label $c$ whose origin nodes $u \in V$ fulfill $u \prec u_i$ (the labels of outgoing edges in $\mathsf{L}$ are ordered according to the node order in $G$). By the second monotonicity property, these edges also lead to nodes $v \in V$ with $v \preceq u_j$. As the number of edges pointing to nodes $v \in V$ with $v \preceq u_j$ is bound by the value $\mathsf{C_{in}}(u_j) + \mathsf{deg_{in}}(u_j)$, we obtain $\mathsf{C_L}[c] + \mathsf{rank_L}(c, lb - 1) \leq \mathsf{C_{in}}(u_j) + \mathsf{deg_{in}}(u_j)$.

For Inequality (2.2), the properties in reversed implication order are as follows:

$$\begin{aligned} v \prec v' \quad &\Rightarrow \quad \lambda(e) \leq \lambda(e'), \\ v \prec v' \quad &\Rightarrow \quad (\lambda(e) \neq \lambda(e')) \vee (u \preceq u'). \end{aligned}$$

According to these properties, edges $(u, v)$ with $v \prec u_j$ can be split in edges $(u, v)$ with $\lambda((u, v)) < c$ and edges $(u, v)$ with $\lambda((u, v)) = c$ and $u \preceq u_i$. The number of edges $e$ with $\lambda(e) < c$ is clearly given by $\mathsf{C_L}[c]$. The number of edges $(u, v)$ with $\lambda((u, v)) = c$ and $u \preceq u_i$ is strictly smaller than $\mathsf{rank_L}(c, rb)$ because their edge labels must be contained in $\mathsf{L}[1..rb]$, and $\mathsf{rank_L}(c, rb)$ includes at least one additional edge label $c$ from the edge $(u_i, u_j)$. Combining both observations then gives the inequality $\mathsf{C_{in}}(u_j) < \mathsf{C_L}[c] + \mathsf{rank_L}(c, rb)$, which concludes the statement.

(iii) This directly follows from Definition 2.19 as the vector $D_{\mathsf{in}}$ is the concatenation of the reverse unary encoded indegrees of nodes sorted by the node ordering of the graph.

□

Theorem 2.20 can be applied as follows: say, we are at a node with rank $i$, and want to navigate to the node with rank $j$ using an edge $(i, j)$ with label $\lambda((i, j)) = c$. First, according to statement (i), we can navigate to the labels of outgoing edges using

$$[lb, rb] \leftarrow [\mathsf{select}_{D_{\mathsf{out}}}(1, i), \mathsf{select}_{D_{\mathsf{out}}}(1, i+1) - 1].$$

We might check if an outgoing edge labeled $c$ exists by checking $\mathsf{rank}_{\mathsf{L}}(c, lb - 1)$ for inequality with $\mathsf{rank}_{\mathsf{L}}(c, rb)$. Now, we follow the edge using $k \leftarrow \mathsf{C}_{\mathsf{L}}[c] + \mathsf{rank}_{\mathsf{L}}(c, rb)$ by statement (ii). Finally, we can find the rank of our target node using $j \leftarrow \mathsf{rank}_{D_{\mathsf{in}}}(1, k)$ by statement (iii). This allows us to traverse the Wheeler graph by using only the components $\mathsf{L}$, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$. As another example, say we are at the starting point of an edge $i$ in $\mathsf{L}$ and $D_{\mathsf{out}}$. To find the target node, one can apply the following formula:

$$j \leftarrow \mathsf{rank}_{D_{\mathsf{in}}}(1, \mathsf{C}_{\mathsf{L}}[\mathsf{L}[i]] + \mathsf{rank}_{\mathsf{L}}(\mathsf{L}[i], i))$$

Not shown here, but straight forward to show is that the node of an edge $i$ in $\mathsf{L}$ and $D_{\mathsf{out}}$ can be found using $\mathsf{rank}_{D_{\mathsf{out}}}(1, i)$. Similarly, the labels of the incoming edges of a node with rank $i$ can be found in the substring $\mathsf{F}[\mathsf{select}_{D_{\mathsf{in}}}(1, i)..\mathsf{select}_{D_{\mathsf{in}}}(1, i+1) - 1]$.

In one last point, we want to note that a BWT can also be traversed with forward instead of backward steps: without giving explicit proof, the connection $\mathsf{LF}^{-1}[i] = \mathsf{select}_{\mathsf{L}}(\mathsf{F}[i], i - \mathsf{C}_{\mathsf{F}}[\mathsf{F}[i]])$ holds. Therefore, a Wheeler graph can be traversed in both directions, but in typical applications, the backward step is preferred to the forward step as rank-queries can be executed faster than select-queries.

# 3 TUNNELING THEORY

In this chapter we will present a technique called "tunneling". Tunneling was introduced by the author of this thesis for the purpose of data compression [Bai18], but since then has been extended to the field of sequence analysis [Ala+19]. The technique uses Wheeler graphs as a base, but to give the reader more insight on the technique, definitions and examples will also be given in the form of Burrows-Wheeler transformed strings.

Before the technique is described, we want to recapitulate why the Burrows-Wheeler transform (BWT) is so useful for data compression. Suppose we have a long English text containing the word `computer` a couple thousand times. Given that the text does not contain too "creative" words, every occurrence of the string `puter` will be preceded by the character `m`, every occurrence of the string `mputer` will be preceded by the character `o`, and every occurrence of `omputer` will be preceded by the character `c`.

All suffixes prefixed by `puter` will be adjacent to each other (they build an $\omega$-interval, see Page 11). A similar observation holds true for suffixes prefixed by `mputer` and `omputer`. As the BWT consists of the (cyclically) preceding characters of the sorted suffixes of a text, within e.g. the `puter`-interval, the BWT will contain only the character `m`. Similarly, this will happen for the `mputer`-interval with character `o` and the `omputer`-interval with character `c`.

This clustering effect is the reason why the BWT is so useful for data compression. Clustered data typically is highly compressible using e.g. run-length encoding, see Definition 2.9. Summarizing the observations from the running example, suffixes that are prefixed by `puter` are always preceded by the string `com`. As one can imagine, this is just a natural extension of the observations from above and also tends to occur often. Now, to come back to the technique of tunneling, tunneling offers a way to fuse the BWT characters in the `puter`- , `mputer`- and `omputer`-intervals to just one character. Expressed differently, tunneling fuses all occurrences of `com` that are followed by `puter` to just one occurrence of `com`.

As we will see later, this helps to improve the compressibility of a BWT. The starting point of the technique is given by prefix intervals. Prefix intervals describe the idea that in our running example `puter` always is preceded by `com`.

## 3.1 PREFIX INTERVALS

In this section, we want to define the meaning of "identical substrings preceding identical strings" in form of prefix intervals.[1] Prefix intervals are related to suffix intervals, as prefix intervals describe intervals where prefixes preceding the sorted suffixes share a common suffix. We want to give the broader definition using Wheeler graphs first, which was originally introduced in [Ala+19] and [BD19].

**Definition 3.1** (Prefix interval)**.** Let $G = (V, E)$ be a Wheeler graph with label function $\lambda$. A prefix interval is a collection of $h \geq 2$ paths $p_1 = (v_{1,1}, v_{1,2}, \cdots, v_{1,w}), \cdots, p_h = (v_{h,1}, v_{h,2}, \cdots, v_{h,w})$ of equal length $w \geq 2$, fulfilling the following properties:

1. Each node $v_{i,j}$ has in- and out-degree one for all $1 \leq i \leq h$ and $1 \leq j \leq w$.

2. The paths are all "parallel", that is, $v_{i+1,j}$ is the immediate successor of $v_{i,j}$ in terms of the Wheeler graph node order for all $1 \leq i < h$ and $1 \leq j \leq w$.

3. Each path is "equally-labeled", that is, $\lambda((v_{i,j}, v_{i,j+1})) = \lambda((v_{i',j}, v_{i',j+1}))$ for all $1 \leq i, i' \leq h$ and $1 \leq j < w$.

The string length of a prefix interval is defined by $w - 1$, its height by $h$.

An example of a prefix interval can be found in Figure 3.1. The definition seems a bit unclear as there is no connection between prefix intervals and prefixes preceding sorted suffixes and sharing a common suffix. Therefore, we give the following corollary "translating" Definition 3.1 into the world of the normal BWT.

**Corollary 3.2.** *Let $S$ be a null-terminated string of length $n$ with suffix array* SA*, LF-mapping* LF *and BWT* L *with Wheeler graph $G$. Furthermore let $\langle w, [i, j] \rangle$ be a pair of a width $w \geq 2$ and an interval $[i, j] \subseteq [1, n]$ with $j > i$. Then the following statements are equivalent:*

*1.* $S[\mathsf{SA}[i] - w + 1..\mathsf{SA}[i] - 1] = \cdots = S[\mathsf{SA}[j] - w + 1..\mathsf{SA}[j] - 1]$.

[1] In other publications, prefix intervals are called blocks, but we decided to call them prefix intervals due to a better understandability of the concept.

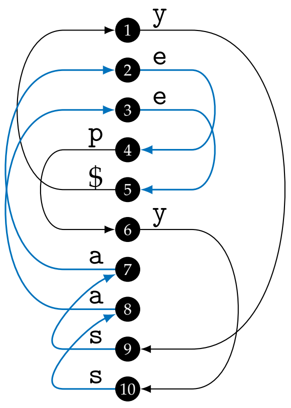
2. $\mathsf{L}[\mathsf{LF}^x[i]] = \mathsf{L}[\mathsf{LF}^x[i+1]] = \cdots = \mathsf{L}[\mathsf{LF}^x[j]]$ *for all* $0 \le x < w-1$.

3. $(\mathsf{LF}^{w-1}[i], \mathsf{LF}^{w-2}[i], \ldots, i), \ldots, (\mathsf{LF}^{w-1}[j], \mathsf{LF}^{w-2}[j], \ldots, j)$ *is a prefix interval of* $G$.

*Proof.* 1 $\Leftrightarrow$ 2:
Suppose $S[\mathsf{SA}[i] - w + 1..\mathsf{SA}[i] - 1] = \cdots = S[\mathsf{SA}[j] - w + 1..\mathsf{SA}[j] - 1]$. Then $S[\mathsf{SA}[i] - 1] = \cdots = S[\mathsf{SA}[j] - 1]$ holds iff $\mathsf{L}[i] = \cdots = \mathsf{L}[j]$ because of the definition of the BWT and the fact that $S[\mathsf{SA}[i]..n], \ldots, S[\mathsf{SA}[j]..n]$ are lexicographically adjacent suffixes. The equivalence of the remaining columns $\mathsf{L}[\mathsf{LF}^x[i]] = \cdots = \mathsf{L}[\mathsf{LF}^x[j]]$ and $S[\mathsf{SA}[i] - 1 - x] = S[\mathsf{SA}[j] - 1 - x]$ follows analogously using the fact that the LF-mapping corresponds to a backward step in the normal string.

2 $\Leftrightarrow$ 3:
Follows by the equivalence of the LF-mapping and outgoing edges in the Wheeler graph as well as the equally labeled columns in the parallel paths and the BWT, see Corollary 2.16. □

The correspondence between prefix intervals in a Wheeler graph and a BWT is illustrated in Figure 3.1. Our next target is to show how prefix intervals can be computed efficiently. We therefore make use of the longest common suffix array which was introduced in [KKP12].



| $i$ | $\mathsf{SA}[i]$ | $\mathsf{LCS}[i]$ | $S[1..\mathsf{SA}[i]-1]$ | $S[\mathsf{SA}[i]..n]$ |
|---|---|---|---|---|
| 1 | 10 | 0 | easypeasy | $ |
| 2 | 7 | 0 | easype | asy$ |
| 3 | 2 | 1 | e | asypeasy$ |
| 4 | 6 | 0 | easyp | easy$ |
| 5 | 1 | 0 | $\varepsilon$ | easypeasy$ |
| 6 | 5 | 0 | easy | peasy$ |
| 7 | 8 | 0 | easypea | sy$ |
| 8 | 3 | 2 | ea | sypeasy$ |
| 9 | 9 | 0 | easypeas | y$ |
| 10 | 4 | 3 | eas | ypeasy$ |

**Figure 3.1:** Wheeler graph (left) as well as suffix array, longest common suffix array, prefixes and sorted suffixes (right) of the string $S$ = easypeasy$. The Wheeler graph contains the prefix interval $(9,7,2,4), (10,8,3,5)$ colored blue. The corresponding prefix eas is marked blue (rows 9 and 10). Additionally, the prefixes ea (rows 7 and 8) and e (rows 2 and 3) are marked blue to show that the columns of the prefix interval correspond to consecutive letters in the BWT, namely s to rows 9-10, a to rows 7-8 and e to rows 2-3. Parts of this image were already published in [BD19] © 2019 IEEE.

**Definition 3.3** (LCS array)**.** Let $S$ be a null-terminated string of length $n$ with suffix array SA. The longest common suffix array (LCS-array) is an array of length $n$ defined as follows:

$$\mathsf{LCS}[i] := \begin{cases} 0 & \text{, if } i = 0. \\ \max\{\, l \in [0,n] \mid S[\mathsf{SA}[i]-l..\mathsf{SA}[i]-1] = \\ \qquad\qquad S[\mathsf{SA}[i-1]-l..\mathsf{SA}[i-1]-1]\,\} & \text{, else.} \end{cases}$$

In other words, the LCS-array describes the length of the longest common suffix between two prefixes preceding lexicographically adjacent suffixes. Figure 3.1 shows an example of an LCS-array; the following lemma formulated in [KKP12] can be used for efficient computation of the array.

**Lemma 3.4.** *Let $S$ be a null-terminated string of length $n$ with BWT $\mathsf{L}$ and LF-mapping $\mathsf{LF}$. Then, for $i > 1$ the $\mathsf{LCS}$-array can be recursively computed as follows:*

$$\mathsf{LCS}[i] = \begin{cases} 0 & \text{, } \textit{if } \mathsf{L}[i] \neq \mathsf{L}[i-1]. \\ \mathsf{LCS}[\mathsf{LF}[i]]+1 & \text{, } \textit{if } \mathsf{L}[i] = \mathsf{L}[i-1]. \end{cases}$$

*Proof.* Suppose $\mathsf{L}[i] \neq \mathsf{L}[i-1]$, then certainly $\mathsf{LCS}[i] = 0$. Now suppose $\mathsf{L}[i] = \mathsf{L}[i-1]$. Then $\mathsf{LCS}[i]$ must be greater than zero, as the prefixes share at least one character. Because of $\mathsf{L}[i] = \mathsf{L}[i-1]$, $\mathsf{LF}[i] = \mathsf{LF}[i-1]+1$ must hold, so following the LF-mapping leads to the suffixes with one more character. More precisely, $\mathsf{SA}[\mathsf{LF}[i-1]] = \mathsf{SA}[i-1]-1$. Because the LF-mapping works with backward steps, if $\mathsf{LCS}[\mathsf{LF}[i]] = l$ holds, then $\mathsf{LCS}[i] = l+1$ must hold too. □

The approach for an efficient computation of the LCS-array is thus as follows: We scan the text from front to back using the inverse LF-mapping $\mathsf{LF}^{-1}$ and meanwhile keep track of a counter $l$. Each time we encounter a position $j$ with $\mathsf{L}[j] = \mathsf{L}[j-1]$, we increment the counter whilst we set the counter $l$ to zero if $\mathsf{L}[j] \neq \mathsf{L}[j-1]$. Due to Lemma 3.4, the value $\mathsf{LCS}[j]$ is given by the current counter value $l$, and the algorithm requires $O(n)$ time.

The enumeration of all left-maximal and height-maximal prefix intervals[2] is possible using a stack-based approach on the LCS-array. The approach is similar to the

[2] Left-maximality means that a prefix interval cannot be extended to the left. Height-maximality means that it is not possible to add another row to the interval.

```
Data: BWT L and inverse LF-mapping $\mathsf{LF}^{-1}$ of a null-terminated string $S$ of length $n$.
Result: Left-maximal height-maximal prefix intervals in the form $\langle w, [i, j] \rangle$ where $w$ is the width and $[i, j]$ are the
        boundaries of the rightmost column of the prefix interval.

                                                                  // compute LCS-array
1  create an array LCS of size $n$
2  $\mathsf{LCS}[1] \leftarrow 0$
3  $j \leftarrow \mathsf{LF}^{-1}[1]$
4  $l \leftarrow 0$
5  for $i \leftarrow 2$ to $n$ do
6      if $\mathsf{L}[j] = \mathsf{L}[j-1]$ then
7          $l \leftarrow l + 1$
8      else
9          $l \leftarrow 0$
10     $\mathsf{LCS}[j] \leftarrow l$
11     $j \leftarrow \mathsf{LF}^{-1}[j]$

                                  // enumerate left-maximal height-maximal prefix intervals
12 initialize an empty stack $s$
13 push element $\langle 1, 0 \rangle$ on the stack $s$
14 for $i \leftarrow 2$ to $n$ do
15     $\langle lb, l \rangle \leftarrow$ top of stack $s$       // $l$ is the string width of the prefix interval
16     while $l > \mathsf{LCS}[i]$ do
17         pop topmost element of stack $s$
18         report prefix interval $\langle l + 1, [lb, i - 1] \rangle$
19         $\langle lb, l \rangle \leftarrow$ top of stack $s$
20     if $l < \mathsf{LCS}[i]$ then
21         push element $\langle i - 1, \mathsf{LCS}[i] \rangle$ on the stack $s$

                                                      // pop remaining elements from stack
22 while stack s has more than one element do
23     $\langle lb, l \rangle \leftarrow$ top of stack $s$
24     report prefix interval $\langle l + 1, [lb, n] \rangle$
25     pop topmost element of stack $s$
```

**Algorithm 3.1:** Enumeration of all left-maximal height-maximal prefix intervals. Lines 1–11 compute the LCS-array using Lemma 3.4, lines 12–25 use a stack-based approach to enumerate the intervals similar to LCP-intervals [Kas+01].

enumeration of LCP intervals as described in [Kas+01]. The key idea is to scan the LCS from top to bottom, meanwhile pushing the left boundary and string width of prefix intervals onto a stack. Once the right boundary of a prefix interval is reached, the interval is popped from the stack.

To be more precise, the left boundary of a prefix interval is detected if the string width of the topmost prefix interval on the stack is smaller than the current $\mathsf{LCS}[i]$-value. This means that a new prefix interval with left boundary $i-1$ and string width $\mathsf{LCS}[i]$ has been detected. The end of a prefix interval can be detected if its string width is smaller than the current $\mathsf{LCS}[i]$-value. This then means that all LCS-values between the left boundary and $i-1$ are equal or greater than the current string width, but the interval cannot be height-extended. The stack maintains the invariant that the topmost prefix interval on the stack has a bigger string width than all other prefix intervals on the stack. Therefore, all prefix intervals ending at a certain position can be popped from the stack using a loop. As this procedure pushes at most $n$ intervals onto the stack, at most $n$ elements are popped from the stack. This implies an $O(n)$ run-time.

We refer to [Kas+01] for more details on the interval enumeration using the LCS-array. Instead, we want to represent a combination of both LCS computation and prefix interval enumeration given in Algorithm 3.1 requiring $O(n)$ run-time.

## 3.2 TUNNELING

After introducing prefix intervals, we now want to present the tunneling technique. As described in the beginning of this chapter, the idea is to fuse identical strings preceding consecutive lexicographically sorted suffixes. We first want to give a definition of tunneling, similar to the ones published in [Bai18] and [BD19].[3]

**Definition 3.5** (Tunneling)**.** Let $G = (V, E)$ be a Wheeler graph with label function $\lambda$, and let $p_1 = (v_{1,1}, \cdots, v_{1,w}), \cdots, p_h = (v_{h,1}, \cdots, v_{h,w})$ be the node-disjoint paths of a prefix interval of $G$ with width $w$ and height $h$.

The process of tunneling the prefix interval is defined as fusing the nodes of each column of the prefix interval as well as the edges between two adjacent columns. Define $F := \{v_{2,1}, \dots, v_{2,w}, \dots, v_{h,1}, \dots, v_{h,w}\}$, then the tunneled Wheeler graph $\tilde{G} = (\tilde{V}, \tilde{E})$ with label function $\tilde{\lambda}$ is defined as

$$\begin{aligned}
\tilde{V} :=&\ V \setminus F, \\
\tilde{E} :=&\ E \setminus ((F \times V) \cup (V \times F)) \\
&\cup \{\, (u, v_{1,1}) \mid u \in V, (u, v_{1,j}) \in E \text{ for some } 1 \leq j \leq h \,\} \\
&\cup \{\, (v_{w,1}, v) \mid v \in V, (v_{w,j}, v) \in E \text{ for some } 1 \leq j \leq h \,\}, \\
\tilde{\lambda}(u, v) :=&\ \begin{cases} \lambda(u, v_{1,j}) & \text{, if } v = v_{1,1} \text{ and } (u, v_{1,j}) \in E. \\ \lambda(v_{w,j}, v) & \text{, if } u = v_{w,1} \text{ and } (v_{w,j}, v) \in E. \\ \lambda(u, v) & \text{, else.} \end{cases}
\end{aligned}$$

[3] Definition 3.5 differs from the published definitions as the original definitions required that nodes from the leftmost and rightmost column of a prefix interval must not be fused. This limiting requirement is unnecessary, allowing tunnels to be longer.

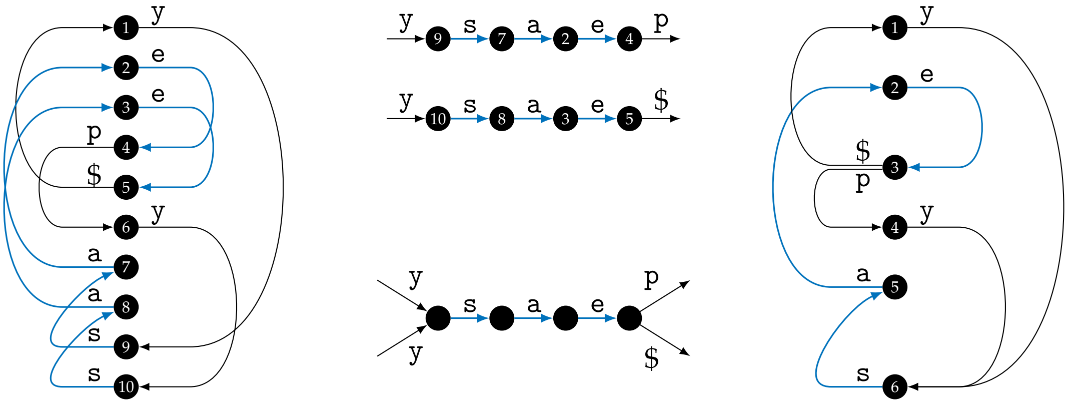


**Figure 3.2:** Wheeler Graph of BWT yeep$yaass (left) with prefix interval $(9,7,2,4),(10,8,3,5)$ colored blue. Tunneling of the prefix interval inside the graph (center), tunneled Wheeler graph (right). The numbering of nodes illustrates the underlying Wheeler graph order in both graphs. This image was already published in [BD19] © 2019 IEEE.

An example of a tunneled Wheeler graph can be found in Figure 3.2. The attentive reader may have noticed that tunneling requires the supplementary condition of node-disjoint paths in a prefix interval. The simple, albeit incomplete, answer is that tunneling would cut out the common subpaths of node-sharing paths, and thus would produce a shortened but differently shaped Wheeler graph. To give a complete answer (which is done in the next section), we need a better understanding of the effect of tunneling, which is given in the following theorem.

**Theorem 3.6.** *Let $G = (V, E)$ be a Wheeler graph with label function $\lambda$, and let $p_1 = (v_{1,1}, \dots, v_{1,w}), \dots, p_h = (v_{h,1}, \dots, v_{h,w})$ be the node-disjoint paths of a prefix interval of $G$ with width $w$ and height $h$.*

*Let $\tilde{G} = (\tilde{V}, \tilde{E})$ be the tunneled Wheeler graph with label function $\tilde{\lambda}$ emerging by tunneling the prefix interval $p_1, \dots, p_h$. Then $\tilde{G}$ is a Wheeler graph.*

*Proof.* We repeat the conditions for two edges $e = (u, v)$ and $e' = (u', v')$ in a Wheeler graph, in an equivalent reversed implication order, see Definition 2.15.

$$\begin{aligned} v' \succ v \quad &\Rightarrow \quad \lambda(e') \geq \lambda(e), \\ v' \succ v \quad &\Rightarrow \quad (\lambda(e') \neq \lambda(e)) \vee (u' \succeq u). \end{aligned}$$

Define the node ordering $\tilde{\prec}$ of $\tilde{G}$ to be equal to the node ordering $\prec$ of $G$, except that the nodes $F := \{v_{2,1}, \dots, v_{2,w}, \dots, v_{h,1}, \dots, v_{h,w}\}$ are removed from the ordering. $\tilde{G}$ is created from $G$ by removing nodes of $F$ and edges in $F$, as well as moving the endpoints of incoming edges from $v_{j,1}$ to $v_{1,1}$ and the starting points of outgoing edges from $v_{j,w}$ to $v_{1,w}$ for all $1 < j \leq h$.

For all edges neither incident to $v_{1,1}$ nor adjacent to $v_{1,w}$ the relative node ordering $\tilde{\preceq}$ as well as the edge labels remain unchanged. Therefore it suffices to show that the conditions are fulfilled for edge pairs involving incoming edges of $v_{1,1}$ and outgoing edges of $v_{1,w}$.

Let $e = (u, v_{1,1})$ be any incoming edge of $v_{1,1}$ and let $e' = (u', v')$ be any other edge of $\tilde{E}$. W.l.o.g. assume $v' \tilde{\succ} v_{1,1}$, and let $v_{j,1}$ be the node such that $(u, v_{j,1}) \in E$. Because the nodes $v_{1,1}, \ldots, v_{h,1}$ are adjacent in terms of the node order $\prec$ in $G$ and the node order $\tilde{\preceq}$ is derived from $\prec$ by, in addition to other removals, removing the nodes $v_{2,1}, \ldots, v_{h,1}$, the node order $v' \succ v_{j,1} \succeq v_{1,1}$ holds. As the edges $(u, v_{1,1})$ and $(u, v_{j,1})$ carry the same label and have identical predecessors, the necessary conditions between $e$ and $e'$ are inherited from the conditions between $(u, v_{j,1})$ and $e'$.

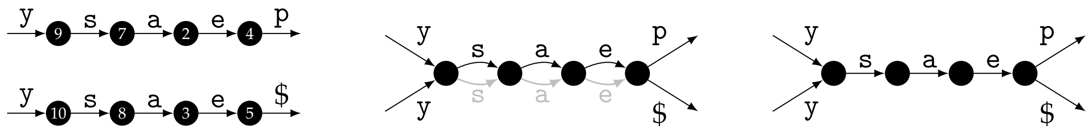
In the special case that $\lambda(e) = \lambda(e')$ and $u' = v_{1,w}$ holds, let $v_{k,w}$ be the unique node such that $(v_{k,w}, v') \in E$ holds. W.l.o.g. assume $v' \tilde{\succ} v_{1,1}$. Then, analogously to the discussion in the last paragraph, one can show that $u' \tilde{\succeq} u$ must hold because $v_{k,w} \succeq u$ holds in the old graph $G$. This ensures the necessary conditions between incoming edges of $v_{1,1}$ and outgoing edges of $v_{1,w}$.

An analogous discussion as above shows that the Wheeler graph edge conditions also are fulfilled for the outgoing edges of $v_{1,w}$. □

Similar versions of Theorem 3.6 appeared in [Bai18] for the BWT and in [Ala+19] for Wheeler graphs. The theorem confirms our assumption that tunneling in some way leaves the graph intact. Moreover, using the result of Theorem 3.6, we can iteratively apply Definition 3.5 to a Wheeler graph, allowing us to tunnel multiple prefix intervals of a Wheeler graph.

### 3.2.1 *Tunneled BWT computation*

The next goal will be to show an algorithm which is able to tunnel a normal BWT. A simple approach would be as follows: given a node-disjoint prefix interval $\langle w, [i, j] \rangle$, we start by fusing the nodes of the columns of the prefix interval, namely $[i, j], [\mathsf{LF}[i], \mathsf{LF}[j]], \ldots, [\mathsf{LF}^{w-1}[i], \mathsf{LF}^{w-1}[j]]$. This produces a multigraph which is quite similar to the desired graph, the difference being that the graph contains redundant



| $i$ | $\mathsf{L}[i]$ | $D_{\text{out}}[i]$ | $D_{\text{in}}[i]$ |
|---|---|---|---|
| 1 | y | 1 | 1 |
| 2 | e | 1 | 1 |
| 3 | e | 1 | 1 |
| 4 | p | 1 | 1 |
| 5 | $ | 1 | 1 |
| 6 | y | 1 | 1 |
| 7 | a | 1 | 1 |
| 8 | a | 1 | 1 |
| 9 | s | 1 | 1 |
| 10 | s | 1 | 1 |
| 11 | | 1 | 1 |

| $\mathsf{L}[i]$ | $D_{\text{out}}[i]$ | $D_{\text{in}}[i]$ |
|---|---|---|
| y | 1 | 1 |
| e | 1 | 1 |
| e | 0 | 0 |
| p | 1 | 1 |
| $ | 0 | 0 |
| y | 1 | 1 |
| a | 1 | 1 |
| a | 0 | 0 |
| s | 1 | 1 |
| s | 0 | 0 |
| | 1 | 1 |

| $\mathsf{L}[i]$ | $D_{\text{out}}[i]$ | $D_{\text{in}}[i]$ |
|---|---|---|
| y | 1 | 1 |
| e | 1 | 1 |
| p | 1 | 1 |
| $ | 0 | 1 |
| y | 1 | 1 |
| a | 1 | 1 |
| s | 1 | 0 |
| | 1 | 1 |

**Figure 3.3:** Systematic tunneling of the $\langle 4, [9, 10] \rangle$ prefix interval from Figure 3.2. The left column shows the prefix interval of the Wheeler graph as well as its succinct representation. The middle column shows the prefix interval and the succinct graph representation after fusing the nodes in the columns of the prefix interval, where redundant edges are colored gray in both the prefix interval and the succinct representation. The right column shows the prefix interval and the succinct representation after removing the redundant edges. Parts of this image were already published in [BD19] © 2019 IEEE.

equally-labeled edges, see Figure 3.3. Once we fused those nodes, we fuse the edges with common start and end node.

Translated to the BWT, this can be performed as follows: to fuse nodes, we set $D_{\text{in}}[\mathsf{LF}^k[i+1]..\mathsf{LF}^k[j]] = 0^{j-i}$ and $D_{\text{out}}[\mathsf{LF}^k[i+1]..\mathsf{LF}^k[j]] = 0^{j-i}$ for each column $k \in [0, w-1]$. Regarding the succinct representation of Wheeler graphs from Definition 2.19, this produces the multigraph as mentioned above. Afterwards, we have to remove redundant edges with a common start and end node, or put differently, reduce the in- and outdegrees of the affected nodes. More precisely, the outdegree of the first node must be reduced to one, the indegree of the last node must be reduced to one, and the out- and indegrees of all other nodes must be reduced to one. To fuse the edges, we have to remove the entries $[i+1, j]$ (first node) and $[\mathsf{LF}^k[i+1], \mathsf{LF}^k[j]]$ for $1 \leq k < w-1$ (inner nodes) from $D_{\text{out}}$ and also from $\mathsf{L}$. Additionally, the entries $[\mathsf{LF}^k[i+1], \mathsf{LF}^k[j]]$ for $1 \leq k < w-1$ (inner nodes) and $[\mathsf{LF}^{w-1}[i+1], \mathsf{LF}^{w-1}[j]]$ (last node) have to be removed from $D_{\text{in}}$. This could be done by e.g. marking the above

```
Data: BWT L and LF-mapping LF of a string $S$ of length $n$, set $I$ of prefix intervals such that any pair of paths from
      two prefix intervals are node-disjoint.
Result: Bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ containing markings of columns from prefix intervals in $I$.

create two bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ of size $n+1$ filled with ones
foreach $\langle w, [i,j] \rangle \in I$ do
    $h \leftarrow j - i + 1$
    $x \leftarrow i$
    for $k \leftarrow 0$ to $w - 2$ do
        $D_{\mathsf{in}}[x+1..x+h] \leftarrow 0^{h-1}$
        $x \leftarrow \mathsf{LF}[x]$
        $D_{\mathsf{out}}[x+1..x+h] \leftarrow 0^{h-1}$
```

**Algorithm 3.2:** Marking of columns from disjoint prefix intervals.

mentioned entries in two separate bit-vectors, and performing an additional top-to-bottom traversal of both bit-vectors whereby trimming $D_{\mathsf{out}}$, L and $D_{\mathsf{in}}$ appropriately, see Figure 3.3.

By reviewing the procedure from above, an optimization is possible as follows: instead of fusing nodes and marking edges to be removed separately, the steps can be merged by setting $D_{\mathsf{in}}[\mathsf{LF}^k[i+1]..\mathsf{LF}^k[j]] = 0^{j-i}$ and $D_{\mathsf{out}}[\mathsf{LF}^{k+1}[i+1]..\mathsf{LF}^{k+1}[j]] = 0^{j-i}$ for all $0 \leq k < w-1$. By doing so, the outdegrees for all nodes, except for the first, as well as the indegree for all nodes, except for the last, are correct. Additionally, the positions of zero markings in $D_{\mathsf{in}}$ and the entries to be removed from $D_{\mathsf{out}}$ and L are identical. The same holds true for zero markings in $D_{\mathsf{out}}$ and entries to be removed from $D_{\mathsf{in}}$. Thus, if we scan $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ from top to bottom after placing the markings, the desired Wheeler graph can be obtained by deciding whether to keep an entry depending on the zero-markings in both bit-vectors. Algorithms 3.2 and 3.3 thus show a way to tunnel multiple prefix intervals at once, given that the paths of all prefix intervals are disjoint.

An interesting special case of tunneling occurs when two adjacent prefix intervals are tunneled. Let $\langle w, [i,j] \rangle$ and $\langle \tilde{w}, [\tilde{i}, \tilde{j}] \rangle$ be two prefix intervals such that $\mathsf{LF}^w[k] \in [\tilde{i}, \tilde{j}]$ holds for any $k \in [i,j]$. Expressed differently, the first column of $\langle \tilde{w}, [\tilde{i}, \tilde{j}] \rangle$ directly follows the last column of $\langle w, [i,j] \rangle$. In this case, if the contact area $[\mathsf{LF}^w[i], \mathsf{LF}^w[j]] \cap [\tilde{i}, \tilde{j}]$ of both prefix intervals contains at least two nodes, tunneling both prefix intervals leads to a tunneled Wheeler graph with multi-edges between both tunnels, see Figure 3.4. This in a way explains why Wheeler graphs are defined to be multigraphs.

**Data:** BWT L of a string $S$ of length $n$, prefix interval marking bit-vectors $D_{\text{out}}$ and $D_{\text{in}}$ as created by Algorithm 3.2.
**Result:** Succinct representation $\tilde{\mathsf{L}}$, $D_{\text{in}}$ and $D_{\text{out}}$ of the tunneled Wheeler graph (or analogously the tunneled BWT).

```
1  let $\tilde{\mathsf{L}}$ be a string of size $\mathsf{rank}_{D_{\text{out}}}(1, n)$
2  $i \leftarrow 1$                          // output position in $D_{\text{out}}$ and $\tilde{\mathsf{L}}$
3  $j \leftarrow 1$                          // output position in $D_{\text{in}}$
4  for $k \leftarrow 1$ to $n$ do
5      if $D_{\text{in}}[k] = 1$ then
6          $D_{\text{out}}[i] \leftarrow D_{\text{out}}[k]$
7          $\tilde{\mathsf{L}}[i] \leftarrow \mathsf{L}[k]$
8          $i \leftarrow i + 1$
9      if $D_{\text{out}}[k] = 1$ then
10         $D_{\text{in}}[j] \leftarrow D_{\text{in}}[k]$
11         $j \leftarrow j + 1$
12 shorten $D_{\text{out}}$ to size $i$ and set $D_{\text{out}}[i] \leftarrow 1$
13 shorten $D_{\text{in}}$ to size $j$ and set $D_{\text{in}}[j] \leftarrow 1$
```

**Algorithm 3.3:** Tunneling of a BWT using the markings of prefix intervals from Algorithm 3.2. In the case that L can be overwritten, $\tilde{\mathsf{L}}$ can be replaced with L by removing line 1 and shortening L to size $i - 1$ before line 12.

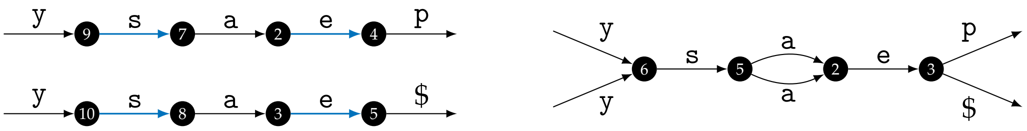


**Figure 3.4:** Tunneling of adjacent prefix intervals produces a multigraph. The left-hand side shows the prefix intervals $\langle 2, [9, 10] \rangle$ and $\langle 2, [2, 3] \rangle$ marked in blue for the running example. On the right, a snippet of the resulting tunneled Wheeler graph with multiple edges between nodes 5 and 2 is shown. The numbering of the nodes illustrates the underlying Wheeler graph order. Parts of this image were already published in [BD19] © 2019 IEEE.

## 3.2.2 *Backward steps*

Another important property of a tunneled Wheeler graph (or at least its succinct representation) is that a walk in the normal Wheeler graph can be emulated, giving that some order between the edges of the graph is present and is not modified by the process of tunneling. The key idea is that tunneling does not modify the structure of the graph, except for the fusion of parallel node-disjoint paths. During a walk, assume we detect the start of a tunnel and enter the tunnel at the $e$-th edge regarding the relative order of the incoming edges of the node. As the fused paths in the original prefix interval are equally labeled, we can follow the fused path and thereby obtain the equal path label. At the end of the tunnel, if we leave the tunnel at the $e$-th edge regarding the relative order, a walk through the corresponding $e$-th path in the original graph has been emulated. This holds true because in the original prefix interval, paths are parallel, so entering a prefix interval on the $e$-th row automatically

```
Data: Succinct representation L, $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ of a tunneled BWT as computed from Algorithm 3.3, index $i$ of an
      edge in $D_{\mathsf{out}}$, tunnel offset $e$.
Result: Index $i$ of the next edge in $D_{\mathsf{out}}$ and tunnel offset $e$ after an emulated backward step in the normal BWT.

function backwardstep($i, e$)
                          // follow $i$-th edge using a normal backward step and determine node rank
    $i \leftarrow \mathsf{C_L}[\mathsf{L}[i]] + \mathsf{rank_L}(\mathsf{L}[i], i)$
    $nr \leftarrow \mathsf{rank}_{D_{\mathsf{in}}}(1, i)$
                              // check if a tunnel starts and save offset to topmost entry edge
    if $D_{\mathsf{in}}[i] = 0$ or $D_{\mathsf{in}}[i+1] = 0$ then
        $e \leftarrow i - \mathsf{select}_{D_{\mathsf{in}}}(1, nr)$
                                                 // switch to outgoing edges of node $nr$
    $i \leftarrow \mathsf{select}_{D_{\mathsf{out}}}(1, nr)$
                         // check if a tunnel ends and use offset to jump to correct outgoing edge
    if $D_{\mathsf{out}}[i+1] = 0$ then
        $i \leftarrow i + e$
        $e \leftarrow 0$
    return $\langle i, e \rangle$
```

**Algorithm 3.4:** Backward step in a tunneled BWT.

implies that the prefix interval is left on the $e$-th row on this path. We now formulate an alternative way of a backward step for a tunneled BWT in Algorithm 3.4.

The algorithm performs a backward step using the two variables $i$ and $e$, where $i$ is the index of the next outgoing edge of the current node whilst $e$ stores the offset to the uppermost entry edge of a tunnel in the case that the tunnel is traversed. As shown in the algorithm, the start or end of a tunnel can be detected by checking the bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ for zeros. As a reminder, Definition 2.19 stated that following the $i$-th edge in the succinct representation of a Wheeler graph can be done using the formula

$$i \leftarrow \mathsf{select}_{D_{\mathsf{out}}}(1, \mathsf{rank}_{D_{\mathsf{in}}}(1, \mathsf{C_L}[i] + \mathsf{rank_L}(\mathsf{L}[i], i))).$$

Figure 3.5 shows how the emulation of a backward step in a tunneled BWT is performed. We now formulate a corollary, finishing this section by clarifying that a tunneled Wheeler graph in a sense is as powerful as a normal Wheeler graph. Similar statements as given by Corollary 3.7 appeared in [Bai18] and [Ala+19].

**Corollary 3.7.** *Let $S$ be a null-terminated string of length $n$. Let* L*, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ be the tunneled BWT emerging by tunneling a set $\mathcal{P}$ of prefix intervals in the BWT of $S$ such that any pair of parallel paths in the prefix intervals of $\mathcal{P}$ are node-disjoint.*

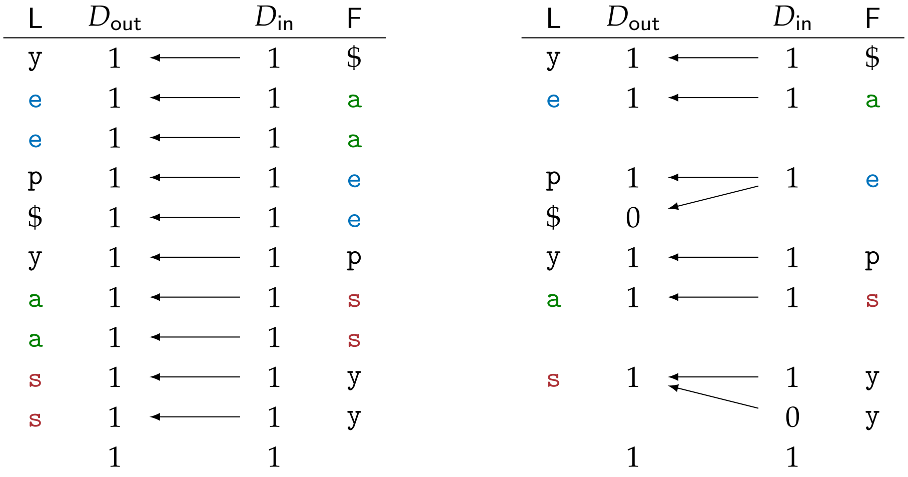
*Starting from the BWT index $i = \mathsf{select_L}(\$, 1)$ and $e = 0$ and repeating the steps*

1. *output character* $\mathsf{L}[i]$
2. *set* $\langle i, e \rangle \leftarrow \texttt{backwardstep}(i, e)$

*exactly $n$ times using the function* `backwardstep` *from Algorithm 3.4 yields the reversed string $S^R$ of the original string $S$.*

Corollary 3.7 shows that a tunneled BWT is able to reproduce the original string from which it was built. By encoding the components L, $D_{\text{out}}$ and $D_{\text{in}}$ using wavelet trees from Section 2.3, this reproduction requires $O(n \log \sigma)$ time.

Let us shortly recapitulate what tunneling means: tunneling is the fusion of parallel equally labeled paths in Wheeler graphs. As Theorem 3.6 shows, a tunneled Wheeler graph is still a Wheeler graph. Therefore, we are not only able to tunnel a normal BWT, but also other string data structures describable as a Wheeler graph. Moreover, if we can ensure that the succinct representation of a Wheeler graph has some edge ordering which is not changed by tunneling, edge navigation in the original Wheeler graph can be emulated in the tunneled Wheeler graph. The key idea is that if we enter a tunnel on the $e$-th incoming edge, leaving the tunnel at the $e$-th outgoing edge leads one back to the correct node in the original graph. This gives tunneling its name: once we enter a tunnel, we keep track of the path on which we entered it. When leaving the tunnel, we just head back to the original path and can proceed as usual.



| L | $D_{\text{out}}$ | $D_{\text{in}}$ | F |
|---|---|---|---|
| y | 1 | 1 | $ |
| e | 1 | 1 | a |
| e | 1 | 1 | a |
| p | 1 | 1 | e |
| $ | 1 | 1 | e |
| y | 1 | 1 | p |
| a | 1 | 1 | s |
| a | 1 | 1 | s |
| s | 1 | 1 | y |
| s | 1 | 1 | y |
|  | 1 | 1 |  |

| L | $D_{\text{out}}$ | $D_{\text{in}}$ | F |
|---|---|---|---|
| y | 1 | 1 | $ |
| e | 1 | 1 | a |
| p | 1 | 1 | e |
| $ | 0 |  |  |
| y | 1 | 1 | p |
| a | 1 | 1 | s |
| s | 1 | 1 | y |
|  |  | 0 | y |
|  | 1 | 1 |  |

**Figure 3.5:** Emulated backward step. The left side shows the succinct representation (including the F-column) of the normal Wheeler graph, while the right side shows the representation (also including F) of the tunneled graph of the running example. Performing four backward steps in the normal BWT beginning from the second y in L leads us to entry 5 with L[5] = $. Now suppose we perform four backward steps in the tunneled BWT starting from the second y in L. When entering the tunnel (first backward step), the offset $e = 1$ is saved. After 3 backward steps, the end of the tunnel is reached, and thus the offset $e = 1$ is added to the current position $i = 3$, thus pointing to entry 4 with L[4] = $.

## 3.3 OVERLAPPINGS

The last section introduced the technique of tunneling, requiring that a set of prefix intervals to be tunneled must be node-disjoint. Within this section, we will see that this restriction can be relaxed: if two prefix intervals overlay each other in a special way, it is still possible to tunnel both prefix intervals using iterative tunneling, see Figure 3.6 for an example.

**Definition 3.8.** Let $G = (V, E)$ be a Wheeler graph, let $p_1, \ldots, p_{h_P}$ be the node-disjoint paths of a prefix interval $P$ in $G$ with width $w_P$, and let $q_1, \ldots, q_{h_Q}$ be the node-disjoint paths of another prefix interval $Q$ in $G$ with width $w_Q$ such that w.l.o.g. $w_P \geq w_Q$.

We say that $P$ and $Q$ are overlayable if the graph $\tilde{G}$ emerging by tunneling $P$ in $G$ contains $Q$ as a prefix interval. In more detail, the set

$$\{q_1\} \cup \{\, q_i \;|\; 1 < i \leq h_Q \text{ such that } q_i \text{ is node-disjoint with the paths } p_2, \ldots, p_{h_P}\}$$

is a prefix interval in $\tilde{G}$.

Definition 3.8 states that two prefix intervals are overlayable if the shorter prefix interval remains in the graph after tunneling the longer one. It also makes sense to tunnel the longer before the shorter prefix interval: assume one would tunnel the shorter prefix interval in Figure 3.6 before the longer one. Then, the start- and endpoint of the fused path would no longer have in- respectively outdegree of one, and thus the longer prefix interval would no longer meet the conditions of a prefix interval from Definition 3.1.

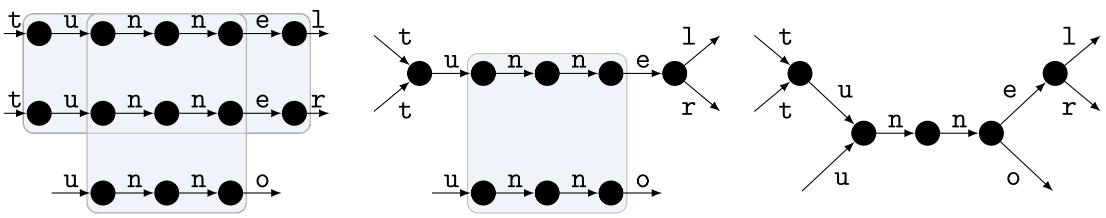


**Figure 3.6:** Iterative tunneling of two overlapping prefix intervals. The example is fictive but shows that the prefix intervals must "cross overlap" each other. According to Definition 3.5, correct iterative tunneling then can be performed by tunneling the longer before the shorter prefix interval. A similar version of this image was already published in [BD19] © 2019 IEEE.

A noteworthiness of Definition 3.8 is that the uppermost path $q_1$ of the shorter prefix interval must "survive" the tunneling process of the longer prefix interval, i.e. it must be part of the remaining prefix interval. Theoretically, it would be possible to remove this extra requirement, but there is a pragmatic reason why one should not do this. The uppermost path of a prefix interval is the "tunnel lane" when the interval is tunneled. The "tunnel lane" has the property that all backward steps performed in the tunnel are performed only in this lane. Therefore, ensuring that the tunnel lane remains intact ensures a kind of a tunnel planning security when a couple of prefix intervals are tunneled. Analogously, when thinking of tunnel planning in the real world, it takes less effort to modify the entries of a tunnel than replanning the tunnel lane: a replanned tunnel lane demands a few new planning steps, like new geological studies.

A weakness of Definition 3.8 is that it describes overlayable prefix intervals in an indirect way. Given two prefix intervals, one has to tunnel the higher prefix interval and check the remaining paths to see if the intervals are overlayable. The next theorem will eliminate this vagueness by formulating clear conditions of overlayable prefix intervals verifiable before any prefix interval is tunneled.

**Theorem 3.9.** *Let $G = (V, E)$ be a Wheeler graph, let $p_1, \dots, p_{h_P}$ be the node-disjoint paths of a prefix interval $P$ in $G$ with width $w_P$, and let $q_1, \dots, q_{h_Q}$ be the node-disjoint paths of another prefix interval $Q$ in $G$ with width $w_Q$ such that w.l.o.g. $w_P \geq w_Q$.*

*Then, $P$ and $Q$ are overlayable if and only if*

1. *$q_1$ and at least one other path of $Q$ are node-disjoint to $p_2, \dots, p_{h_P}$.*
2. *no path of $Q$ contains a start- or endpoint of a path from $P$.*

*Proof.*

"$\Leftarrow$": Tunneling the prefix interval $P$ removes the paths $p_2, \dots, p_{h_P}$. As no path of $Q$ contains a start- or endpoint of any path in $P$, each path of $Q$ is either a subpath of a path in $P$ (the nodes of such a path are fully removed by tunneling $P$ except for subpaths of $p_1$) or node-disjoint to all paths of $P$ (such a path "survives" tunneling $P$).

All nodes of the tunnel-surviving paths of $Q$ must have in- and outdegrees of exactly one, because tunneling modifies no node degrees except for the indegree of the starting point and the outdegree of the endpoint of $p_1$. Additionally, as tunneling does not modify the relative order of nodes or labels of edges (see Theorem 3.6), the tunnel-surviving paths of $Q$ all are parallel and equally labeled.

Finally, as $q_1$ and at least one other path of $Q$ are node-disjoint to $p_2, \ldots, p_{h_P}$, $q_1$ and at least one more path survives tunneling and thus induces a prefix interval in the tunneled graph.

"⇒": Suppose there are less than two paths being node-disjoint to $p_2, \ldots, p_{h_P}$. Then, less than two paths survive tunneling ($p_2, \ldots, p_{h_P}$ are removed) and thus do not fulfill the condition of a prefix interval which must contain at least two parallel paths.

Furthermore, suppose a path in $Q$ contains a start- or endpoint of a path from $P$. Let $q_i$ be the uppermost of such paths in $Q$. Then, $q_i$ must survive tunneling (i.e. it cannot share nodes with some path $p_j$ with $j > 1$), because otherwise, as the paths in $Q$ run in parallel and $q_1$ must be node-disjoint to paths of $P$ (see discussion above), there would be a path $q_k$ above $q_j$ ($k < j$) that contains a start- or endpoint of a path in $P$. This is a contradiction to the assumption that $q_i$ is the uppermost such path. Clearly, $q_i$ must contain the start- or endpoint of $p_1$, as it is the uppermost path. Now, as $q_i$ survives tunneling and the degree of the endpoints of $p_1$ are modified, after tunneling $P$, there is a node in $q_i$ with in- or outdegree of not exactly one, which contradicts the conditions of a prefix interval.

□

An alternative formulation of Theorem 3.9 is as follows: If two prefix intervals share nodes, both are overlayable if the shape of the overlap builds a cross ("cross overlay") such that the shorter prefix interval is higher and does not touch the endpoints of the longer interval, see Figure 3.6.

Before we proceed with algorithms for tunneling overlayable prefix intervals, we will briefly review the prerequisites of the tunneling process from Definition 3.5. One prerequisite of the definition requires that each pair of paths from a prefix interval must be node-disjoint. Using Theorem 3.9, we see that this prerequisite is reasonable. Given a prefix interval with node-disjoint paths, one can split the prefix interval vertically at any column and obtains two new prefix intervals which are overlayable. Now, given a prefix interval that contains a common node in two different paths, one may split the prefix interval vertically, anywhere between the columns where the common node is contained. The two emerging prefix intervals then will share

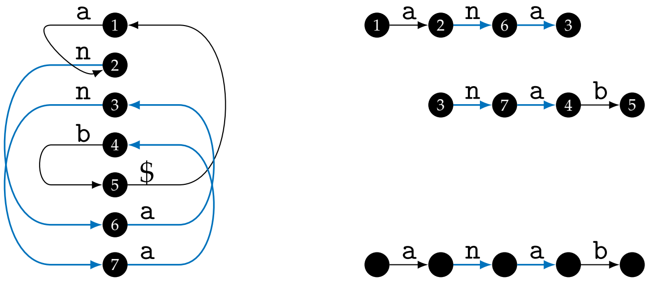


**Figure 3.7:** Tunneling of a self-overlapping prefix interval. The left side shows the Wheeler graph of the BWT of the string $S$ = banana$ with a self-overlapping prefix interval (see node 3). The middle shows the fusion of the parallel paths. As both paths contain the node 3, start and end of the prefix interval also are fused. The right side shows the created Wheeler graph, where one repetition of an has been removed by tunneling.

start- and endpoints, and thus are not overlayable, indicating that it is not reasonable to tunnel the self-overlapping prefix interval. Additionally, as Figure 3.7 shows, tunneling such an interval leads to a loss of the repeated paths. As a consequence, it is no longer possible to recover the original string using the tunneled Wheeler graph.

Next, we would like to discuss how the succinct representation of a tunneled Wheeler graph can be constructed when overlayable prefix intervals are tunneled. Also, we will show a modified approach to perform backward steps in the graph. The construction of a tunneled Wheeler graph using pairwise overlayable prefix intervals is quite straightforward. We run Algorithms 3.2 and 3.3 without any modification, except that the input prefix intervals are pairwise overlayable instead of pairwise node-disjoint.

Formulating a backward step in a tunneled Wheeler graph is a bit more complicated: it is possible to enter a tunnel inside of another tunnel, see e.g. Figure 3.6. However, as we know that the overlappings build a cross, we also know that if we detect the end of a tunnel, it must be the end of the last entered tunnel whose end was not found yet. Therefore, we can use a stack to store the tunnel entry offsets: when the end of a tunnel is detected, the corresponding tunnel entry offset can be found at the top of the stack. This allows one to match tunnel starts and ends appropriately. Algorithm 3.5 shows such a modified backward step, which is quite similar to the original backward step as presented in Algorithm 3.4 except for the use of a stack.

**Data:** Succinct representation $\mathsf{L}$, $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ of a tunneled BWT as computed from Algorithm 3.3 using pairwise overlayable prefix intervals, index $i$ of an edge in $D_{\mathsf{out}}$, stack $s$ with tunnel offsets.
**Result:** Index $i$ of the next edge in $D_{\mathsf{out}}$ and stack $s$ with tunnel offset after an emulated backward step in the normal BWT.

```
function backwardstep($i, s$)
                          // follow $i$-th edge using a normal backward step and determine node rank
    $i \leftarrow \mathsf{C}_\mathsf{L}[\mathsf{L}[i]] + \mathsf{rank}_\mathsf{L}(\mathsf{L}[i], i)$
    $nr \leftarrow \mathsf{rank}_{D_\mathsf{in}}(1, i)$
                          // check if a tunnel starts and save offset to topmost entry edge
    if $D_\mathsf{in}[i] = 0$ or $D_\mathsf{in}[i+1] = 0$ then
        $e \leftarrow i - \mathsf{select}_{D_\mathsf{in}}(1, nr)$
        push $e$ on the stack $s$
                          // switch to outgoing edges of node $nr$
    $i \leftarrow \mathsf{select}_{D_\mathsf{out}}(1, nr)$
                          // check if a tunnel ends and use offset to jump to correct outgoing edge
    if $D_\mathsf{out}[i+1] = 0$ then
        $e \leftarrow$ top of stack $s$
        $i \leftarrow i + e$
        pop topmost element of stack $s$
    return $\langle i, s \rangle$
```

**Algorithm 3.5:** Backward step in a tunneled BWT with tunnel overlappings.

**Corollary 3.10.** *Let $S$ be a null-terminated string of length $n$. Let $\mathsf{L}$, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ be the tunneled BWT emerging by tunneling a set $\mathcal{P}$ of prefix intervals in the BWT of $\tilde{S}$ such that any pair of prefix intervals of $\mathcal{P}$ are overlayable.*

*Starting from the BWT index $i = \mathsf{select}_\mathsf{L}(\$, 1)$ and an empty stack $s$ and repeating the steps*

1. *output character $\mathsf{L}[i]$*
2. *set $\langle i, s \rangle \leftarrow$ `backwardstep`$(i, s)$*

*exactly $n$ times using the function* `backwardstep` *from Algorithm 3.5 yields the reversed string $S^R$ of the original string $S$.*

## 3.4 HARDNESS OF TUNNEL PLANNING

As of now, we have seen what tunneling means, how prefix intervals can be computed, and under which conditions prefix intervals are allowed to overlap in order to be suitable for iterative tunneling. An important issue, however, is still missing: tunnel planning. Tunnel planning means, that given all prefix intervals of a BWT (or a Wheeler graph), one must decide which prefix intervals are worth being tunneled.

As we will see in the next chapter, it is not always worth it to tunnel all prefix intervals. Each tunneled prefix interval produces an extra zero-run in the bit-vectors

$D_{\text{out}}$ and $D_{\text{in}}$, and therefore reduces their compressibility. Seen differently, each tunnel produces some kind of cost, apart from the benefit that the length of L, $D_{\text{out}}$ and $D_{\text{in}}$ is decreased. Therefore, it is preferable to tunnel big prefix intervals instead of small ones, because their benefit-cost ratio is better than that of small ones. In some cases the benefit-cost ratio of prefix intervals is less than one, so it is not worth it to tunnel such a prefix interval. We will see such cases in the next chapter.

For now, we want to introduce two problems expressing the planning issue. The following problem was also stated in [BD19].

**Problem 3.11** (Wheeler graph prefix interval cover)**.** Let $G = (V, E)$ be a Wheeler graph, let $k$ be a positive integer and let $\mathcal{P}$ be the set of all prefix intervals contained in $G$.

The Wheeler graph prefix interval cover problem asks if a subset $P \subseteq \mathcal{P}$ of size $|P| \leq k$ exists such that each node $v \in V$ that is contained in a prefix interval of $\mathcal{P}$ also is contained in a prefix interval of $P$.

To put it simply, the Wheeler graph prefix interval cover problem asks if all prefix intervals in some Wheeler graph $G$ can be fully overlaid/covered by a set of at most $k$ prefix intervals. It therefore states the problem of achieving maximal benefit (all nodes contained in prefix intervals are covered by the set $P$) with the smallest possible costs. For our running example from Figure 3.1, the prefix interval of the string eas covers all other prefix intervals in the graph. A slight but more practice-oriented version of the problem is given by the following:

**Problem 3.12** (Wheeler graph prefix interval coverage)**.** Let $G = (V, E)$ be a Wheeler graph, let $k$ and $m$ be two positive integers and let $\mathcal{P}$ be the set of all prefix intervals contained in $G$.

The Wheeler graph prefix interval coverage problem asks if a subset $P \subseteq \mathcal{P}$ of size $|P| \leq k$ exists such that at least $m$ nodes $v \in V$ are contained in the prefix intervals of $P$.

The Wheeler graph prefix interval coverage problem is more practice-oriented because it enables us to ask for a certain benefit-cost ratio of prefix intervals to be tunneled. The ratio can be thought of as a fraction between the number of nodes $m$ and the number of tunneled prefix intervals $k$. The Problems 3.11 and 3.12 explicitly do not ask for an optimal tunnel planning. However, it is easy to see that optimal tunnel planning must be at least as difficult as solving the problems. For example, assume we would be able to find the minimal value of $k^*$ solving the Wheeler graph prefix interval cover problem. In this case, we could also implicitly solve the Wheeler graph prefix interval cover problem as follows:

1. compute the minimal value $k^*$ solving Problem 3.11.
2. for a given value $k$, Problem 3.11 is solvable if and only if $k \geq k^*$ holds.

Therefore, asking for the minimal value solving Problem 3.11 must be at least as hard as solving Problem 3.11 itself.

As we will see in this section, both problems are very difficult. More precisely, the problems turn out to be NP-complete. Therefore, tunnel planning turns out to be a difficult task. Before we give proofs on the hardness of the problems, we want to introduce the concepts behind the hardness of a problem, see e.g. [GJ90] for a more detailed description.

### 3.4.1 *Introduction to complexity classes*

First of all, we need some basic knowledge of complexity classes. The class P is a class that contains decision problems which are deterministically solvable in polynomial time. This means that a a problem-solving deterministic algorithm exists whose worst-case run-time is bound by a polynomial function that depends on the input size. For example, given a Wheeler graph $G$, the problem of determining if some paths $p_1, \ldots, p_h$ form a prefix interval in the graph belongs to P. One is able to describe an algorithm which takes the paths as input, checks the conditions of a prefix interval from Definition 3.1, and then returns true or false. As the worst-case run-time of such an algorithm is polynomial bounded by the sum of the lengths of all paths, and the algorithm works completely deterministic, the problem must be in P.

The class NP is a complexity class which contains decision problems which are non-deterministically solvable in polynomial time. This means that a problem-solving non-deterministic algorithm exists whose worst-case run-time is bounded by a polynomial function that depends on the input size. Non-deterministic algorithms seem a bit confusing at first glance, as those algorithms work like an oracle: during their execution, those algorithms guess some random bits in the right manner, and check that the result is appropriate. Typically, those algorithms work by a guess and check principle. For example, a non-deterministic algorithm for the Wheeler graph prefix interval cover problem can be formulated as follows:

1. Guess $n$ random paths.

2. Check if the paths form at most $k$ prefix intervals in the graph.

3. Check if all other prefix intervals in the graph are covered by the prefix intervals.

Surprisingly, due to the used non-determinism, this algorithm always gives the right answer to the Wheeler graph prefix interval cover problem. Also, each step requires at most polynomial-time, so the Wheeler graph prefix interval cover problem belongs to the class NP.

It is clear that P is a subclass of NP (P $\subseteq$ NP) as each deterministic algorithm is a non-deterministic algorithm which ignores the oracle function. Although strongly believed, it is not yet clear whether P is a proper subclass of NP (P $\subset$ NP). This issue is known as the currently unsolved "P vs NP Problem", and is one of seven Millennium problems [Cla00]. If we assume that P $\neq$ NP holds, problems from NP $\setminus$ P would not be solvable in polynomial time. The best thing that one could do in this case (besides some problem-specific heuristics) would be to simulate non-deterministic algorithms using deterministic ones in a brute force manner: try out all possible bit combinations of the random bits used in the non-deterministic algorithm, and check if at least one such combination survives the check-phase. This makes the worst-case run-time of such algorithms exponential and therefore is limited to only very small instances of the problem.

The hardest problems in NP are the so-called NP-complete problems. In the case that any of these problems can be solved in polynomial time, all problems in NP can be solved in polynomial time. Thus, P = NP would hold. The first such problem, the satisfiability problem, was introduced by Stephen A. Cook [Coo71], before Richard M. Karp presented another 21 such problems [Kar72]. The basic concept of NP-hardness is given by polynomial-time reductions. Assume we have two decision problems $A$ and $B$. A polynomial-time reduction from $A$ to $B$ is a function $f$ that in polynomial time can convert an input $x$ for problem A to an input $f(x)$ for problem $B$. Additionally, for each solution $x$ of problem $A$, $f(x)$ is a solution of problem $B$, and vice versa. More formally, by writing $x \in A$ for every solution $x$ of problem A, this can be expressed as

$$\begin{aligned} A \leq_{\mathsf{p}} B \;:\Leftrightarrow\; & \text{there exists a polynomial-time computable function } f \\ & \text{such that } x \in A \Leftrightarrow f(x) \in B \text{ for every possible input } x. \end{aligned}$$

The term $A \leq_{\mathrm{p}} B$ means that $A$ can be reduced to $B$ in polynomial time. Consequently, if $B$ is solvable in polynomial time, then so is $A$ by converting any input $x$ using the function $f$ and checking if $f(x) \in B$ in polynomial time. An NP-complete problem $A$ is characterized as a problem which

1. is contained in the class of NP-problems.
2. is at least as hard as all other problems in NP, that is, $L \leq_{\mathrm{p}} A$ for all problems $L$ contained in NP.

As described above, the second characterization ensures that all problems of NP can be solved in polynomial time once an NP-complete problem is solvable in polynomial time. As this characterization is hard to prove in practice, another simpler characterization is given as follows: let $A$ be the problem under consideration, and let $B$ be an NP-complete problem. Then, $A$ is NP-complete if

1. $A$ is contained in the class of NP-problems.
2. $A$ is at least as hard as $B$, that is, $B \leq_{\mathrm{p}} A$.

As sketched above in the comparison of the Wheeler graph prefix interval cover problem and the minimal value solving the Wheeler graph prefix interval cover problem, the second comparison ensures that $A$ is as hard as $B$. This also ensures that the second characterization of NP-complete problems is fulfilled: for any language $L$ from NP, we can convert the input to some input of $B$, and then to an input of $A$. As $L$ then is solvable iff $A$ is solvable, we obtain $L \leq_{\mathrm{p}} A$.

### 3.4.2 *NP-completeness of Wheeler graph prefix interval cover*

After this excursion on complexity classes and NP-completeness, we now want to return to our tunnel planning problems. To show that tunnel planning is a hard problem, we need to show that the problems 3.11 and 3.12 can be reduced to an NP-complete problem. To this end, we will use the following NP-complete problem which comes from [GJ90, p. 232].

**Problem 3.13** (Rectilinear picture rectangle cover)**.** Given a matrix $M \in \{0,1\}^{n \times n}$ of a rectilinear binary picture and a positive integer $k$, is there a collection of $k$ or fewer rectangles that cover precisely the 1's in $M$? Formally, does a collection $R_1 = [r_1^l, r_1^u] \times [c_1^l, c_1^u], \dots, R_k = [r_k^l, r_k^u] \times [c_k^l, c_k^u]$ of rectangles in matrix $M$ exist $(r_i^l, r_i^u, c_i^l, c_i^u \in [1,n]$ for all $i \in [1,k])$ such that

- for any $i \in [1, k]$, $R_i$ covers only 1's of $M$, that is, $|R_i| = \sum_{(\tilde{i},\tilde{j}) \in R_i} M_{\tilde{i},\tilde{j}}$.
- $R_1, \ldots, R_k$ cover all 1's in $M$, that is, $|\bigcup_{i=1}^{k} R_i| = \sum_{(\tilde{i},\tilde{j}) \in [1,n] \times [1,n]} M_{\tilde{i},\tilde{j}}$.

Figure 3.8 shows an illustration of the rectilinear picture rectangle cover problem. As noted already, Problem 3.13 is an NP-complete problem, although the history behind the NP-completeness is a bit strange: Masek showed that the problem is NP-complete [Mas78], but did not publish his result. Garey and Johnson also stated the problem as NP-complete [GJ90] without giving a proof, but Johnson in his "NP-completeness column" said that

> [...] Of those "unpublished manuscripts" and "personal communications" that have not yet seen the formal light of day, two in particular stand out. They were both originally cited in [G&J], and between them they seem to have garnered more enquiries than all the others combined, sending me off to the copier repeatedly to fulfill requests. One was the 1978 manuscript "Some NP-complete set covering problems," by William Masek, who was then at MIT but has since disappeared from the theory community.
>
> That paper contains the NP-completeness proofs for two problems of major importance in circuit and VLSI design. [...] The second, related NP-completeness result was for the problem RECTILINEAR PICTURE COMPRESSION ([SR251 in [G&J]): Given a rectilinear polygon R, possibly containing holes, and an integer K, can R be expressed as the (non-disjoint) union of K or fewer rectangles? [Joh87, p. 445]

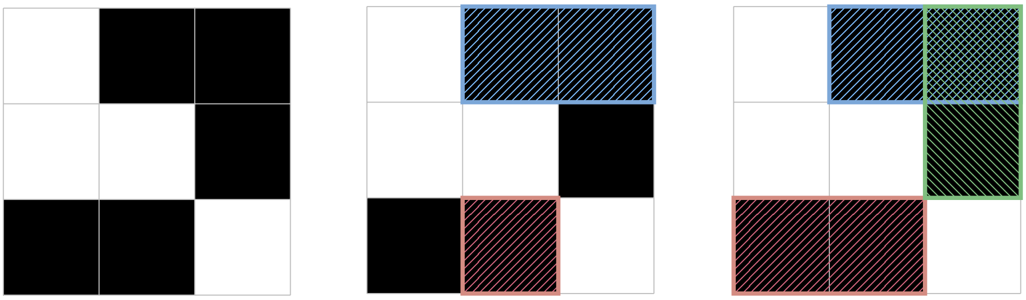


**Figure 3.8:** An instance of the rectilinear picture rectangle cover problem (left) where ones in the matrix are displayed by black pixels. The middle shows a possible coverage of black pixels with two rectangles, although not covering all pixels. The instance requires at least three rectangles to cover all black pixels, one of such coverings is shown on the right-hand side. A similar version of this image was already published in [BD19] © 2019 IEEE.

An indirect proof of the NP-completeness of the Rectilinear picture rectangle cover problem then appeared in [CR94]. The authors showed that 3-SAT can be reduced to the Rectilinear picture rectangle cover problem when the rectilinear picture contains no holes. In this context, a hole is an area of white pixels that are surrounded by black pixels. 3-SAT is a special version of the satisfiability problem which is NP-complete, see e.g. [Kar72] for more details. Solving the Rectilinear picture rectangle cover problem without holes is clearly a special case of the more general version stated in Problem 3.13. Therefore, Problem 3.13 must be NP-complete because it belongs to NP ("guess rectangles and check the cover") and is at least as hard as the same problem without holes.

Our next goal is to describe a certain kind of Wheeler graphs which are strongly related to rectilinear pictures. The following definitions and proofs were devised by the author of this thesis in [BD19].

**Definition 3.14** (Rectilinear picture Wheeler graph)**.** Let $M \in \{0,1\}^{n \times n}$ be the matrix representation of a rectilinear binary picture. Define $\tilde{M} \in \{0,1\}^{2 \cdot n \times 2 \cdot n}$ as a matrix generated by dividing each pixel of $M$ into four $2 \times 2$ pixels, that is,

$$\tilde{M}[i][j] := M[i+1 \operatorname{div} 2][j+1 \operatorname{div} 2].$$

The rectilinear picture Wheeler graph $RPG(M) = ([1, 2 \cdot n] \times [1, 2 \cdot n], E)$ is an edge-labeled graph whose edges are defined as follows:

$$\begin{aligned} E := {} & \underbrace{\{\, ((i,j),(i,j+1)) \;\mid\; i \in [1, 2 \cdot n], j \in [1, 2 \cdot n - 1] \,\}}_{\text{link each row from left to right}} \\ \cup {} & \underbrace{\{\, ((i, 2 \cdot n),(i+1,1)) \;\mid\; i \in [1, 2 \cdot n - 1] \,\} \cup \{\, ((2 \cdot n, 2 \cdot n),(1,1)) \,\}}_{\text{link the end of each row with the start of the (cyclically) next row}} \end{aligned}$$

The edge label function $\lambda$ of $RPG(M)$ is defined as follows:

$$\lambda((i,j),(i',j')) := \begin{cases} \lambda((i-1,j),(i-1,j+1)) & \text{, if } i > 1, i' = i, j' = j+1 \text{ and} \\ & \tilde{M}_{i,j} = \tilde{M}_{i,j+1} = \\ & \tilde{M}_{i-1,j} = \tilde{M}_{i-1,j+1} = 1 \\ j' \,\$\, i' & \text{, else} \end{cases}$$

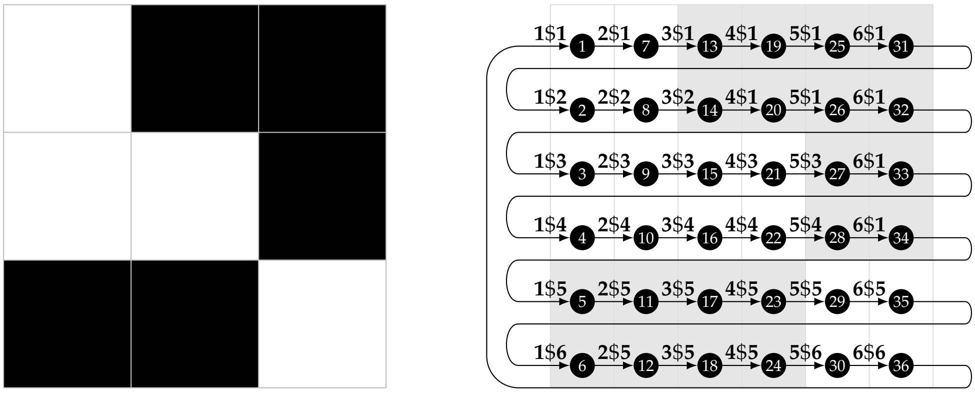


**Figure 3.9:** A rectilinear picture (left) and its corresponding rectilinear picture Wheeler graph (right). A similar version of this image was already published in [BD19] © 2019 IEEE.

Definition 3.14 can be explained as follows: Divide each pixel of the picture into four two-by-two minipixels. Then, place a node inside each of those minipixels. Link each node with its right neighbor node, and link the end node of each row with the start node of the (cyclically) next row. In general, an edge is labeled with the swapped and $-separated coordinates of the underlying minipixel of the destination node. In the special case that the minipixels of the source node and the node above as well as the minipixels of the destination node and the node above are colored black, label the edge with the label from the edge above. An example of a rectilinear picture Wheeler graph is given in Figure 3.9.

The special way in which the edges of the graph are labeled ensures that the graph is a Wheeler graph. In more detail, assume that each coordinate is a letter and that an edge label is smaller than another edge label if it is lexicographically smaller. We then obtain a Wheeler graph by ordering the nodes column-wise from left to right and ascending from top to bottom, as depicted in Figure 3.9.

**Lemma 3.15.** *Let $RPG(M) = ([1, 2 \cdot n] \times [1, 2 \cdot n], E)$ be the rectilinear picture Wheeler graph of some matrix representation $M \in \{0, 1\}^{n \times n}$ of a rectilinear binary picture. Then, using the alphabet order $i\$j < i'\$j' \;:\Leftrightarrow i < i'$ or $i = i'$ and $j < j'$ and the Wheeler graph node order $(i, j) \prec (i', j') \;:\Leftrightarrow\; j < j'$ or $j = j'$ and $i < i'$, $RPG(M)$ is a Wheeler graph.*

*Proof.* By the way edges are defined in Definition 3.14, it is clear that every node has an in- and outdegree of at least one.

Let $e = ((i, j), (\tilde{i}, \tilde{j}))$ and $e' = ((i', j'), (\tilde{i}', \tilde{j}'))$ be two edges of $RPG(M)$. Suppose $\lambda(e) < \lambda(e')$. As this means that $\lambda(e) \neq \lambda(e')$, the labels of $e$ and $e'$ have been produced by the second case of the label case distinction in Definition 3.14, so

$\lambda(e) = \tilde{j}\$\tilde{i}$ and $\lambda(e') = \tilde{j}'\$\tilde{i}'$. As $\lambda(e) < \lambda(e')$, $\tilde{j} < \tilde{j}'$ or $\tilde{j} = \tilde{j}'$ and $\tilde{i} < \tilde{i}'$ is implied, and thus, $(\tilde{i}, \tilde{j}) \preceq (\tilde{i}', \tilde{j}')$ must hold.

Now, suppose $(i, j) \prec (i', j')$ and $\lambda(e) = \lambda(e')$ holds. By the node ordering, this implies $i \leq i'$ and $j \leq j'$. Also, the label equality for at least one edge must have been produced by the first case of the label case distinction from Definition 3.14. Thus, $i = \tilde{i}$, $i' = \tilde{i}'$, $\tilde{j} = j + 1$ and $\tilde{j}' = j' + 1$ must hold. By substituting those identities in the inequalities $i \leq i'$ and $j \leq j'$, we obtain $\tilde{i} \leq \tilde{i}'$ and $\tilde{j} - 1 \leq \tilde{j}' - 1$, or equivalently, $\tilde{j} \leq \tilde{j}'$. This clearly implies $(\tilde{i}, \tilde{j}) \preceq (\tilde{i}', \tilde{j}')$. □

The special way in which the edges of a rectilinear picture Wheeler graph are defined ensures that all prefix intervals in the graph correspond to the black pixels of the underlying picture. A parallel path is labeled equally if and only if the four surrounding minipixels are black. Figure 3.10 shows this analogy.

**Theorem 3.16.** *The Wheeler graph prefix interval cover problem is NP-complete.*

*Proof.* We have already seen that the Wheeler graph prefix interval cover belongs to the complexity class NP. The NP-hardness is shown by a reduction from the rectilinear picture rectangle cover problem to our problem.

The reduction is given by converting some rectilinear binary picture $M \in \{0, 1\}^{n \times n}$ to a rectilinear picture Wheeler graph $RPG(M)$. By Lemma 3.15 it is ensured that $RPG(M)$ is a Wheeler graph. Moreover, without explicitly formulating an algorithm, it can be seen that the conversion from a picture $M$ to the Wheeler graph $RPG(M)$ can be done in polynomial time.

What remains to be shown is the satisfiability equivalence of the problems. Therefore, let $M$ be some rectilinear picture and let $k$ be any positive integer.

"$\Rightarrow$": By the construction of $RPG(M)$ it is ensured that two parallel paths are equally-labeled if and only if the underlying pixels of $M$ are colored black. Therefore, rectangles in $M$ that cover black pixels correspond to prefix intervals in $RPG(M)$. Thus, if the black pixels in $M$ can be covered by at most $k$ rectangles, then all nodes of prefix intervals in $RPG(M)$ can be covered by the $k$ corresponding prefix intervals.

"$\Leftarrow$": Suppose all nodes of prefix intervals in $RPG(M)$ can be covered by at most $k$ prefix intervals. The prefix intervals might not properly coincide with rectangles in $M$ because a prefix interval could e.g. start in the middle of a pixel in $M$ instead of starting at its boundaries. However, in such cases one can extend the prefix intervals such that start and end nodes correspond to the boundaries of rectangles in $M$. Therefore, if all nodes of prefix intervals in $RPG(M)$ can be covered by at most $k$ prefix intervals, then the nodes can also be covered by at most $k$ prefix intervals which coincide with rectangles in $M$ and cover all black pixels.

□

**Corollary 3.17.** *The Wheeler graph prefix interval coverage problem is NP-complete.*

*Proof.* Similar to the Wheeler graph prefix interval cover problem one can see that the coverage problem is in NP by guessing $k$ random prefix intervals and checking if the prefix intervals contain at least $m$ nodes.

Let $\tilde{m}$ be the number of nodes of all prefix intervals contained in some Wheeler graph $G$. This number can be computed in polynomial time, see e.g. Section 3.1. We reduce the cover problem to the coverage problem by asking if at most $k$ prefix intervals suffice to cover $\tilde{m}$ nodes in $G$. The reduction can be done in polynomial time and clearly ensures equal satisfiability. □

### 3.4.3 *Additional notes on tunnel planning complexity*

Because exact tunnel planning seems to be difficult, the next logical step is to look for approximate tunnel planning solutions. More precisely, in the case that all prefix

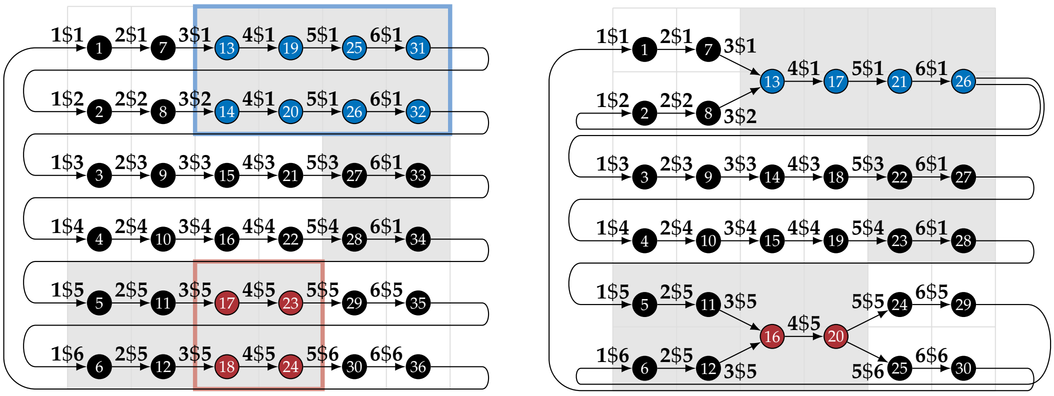


**Figure 3.10:** Analogy between the rectilinear picture rectangle cover problem and the Wheeler graph prefix interval cover problem on rectilinear picture Wheeler graphs. The left-hand side shows that any rectangle covering black pixels corresponds to a prefix interval in the graph. The right-hand side shows the tunneling of the marked prefix intervals in the Wheeler graph. A similar version of this image was already published in [BD19] © 2019 IEEE.

intervals can be covered with the optimum of $k^*$ prefix intervals, one could try to find a strategy ensuring that at most $c \cdot k^*$ prefix intervals are necessary to cover all prefix intervals, where $c$ is a small constant greater than 1. A greedy strategy for the rectilinear picture rectangle cover (picking the biggest rectangles in a greedy manner) results in $c = \log(n)$, so the approximation factor is input-sensitive and gets worse the bigger the instance is [HLL07].

Another way to get approximate results would be the use of a polynomial time approximation scheme (PTAS). Such a scheme requires an additional approximation error $\varepsilon > 0$ and produces a $c = (1 + \varepsilon)$ solution that requires $O(n^{\exp{(1/\varepsilon)}})$ runtime. Unfortunately, the rectilinear picture rectangle cover problem is MaxSNP-hard [BD97] and hence no PTAS for the problem exists unless P $\neq$ NP holds [Pap94]. We will provide a short proof showing that the Wheeler graph prefix interval cover problem is MaxSNP-hard, too. The result was first mentioned by the author of this thesis in [BD19].

We will not reveal the full theory behind the complexity class MaxSNP; see e.g. [Pap94] for a full description. Instead, we present a special approximation-preserving kind of reduction which is similar but a bit more complex than the reductions we have seen so far. Given some problems $A$ and $B$ with cost functions $c_A$ and $c_B$ as well as the optimal instance solutions $\mathsf{OPT}_A(x)$ and $\mathsf{OPT}_B(x)$, the problem $A$ can be $L$-reduced onto problem $B$ if two polynomial-time computable functions $f$ and $g$ exist such that

1. there exists a positive constant $\alpha$ such that for any problem instance $x$ from problem $A$, $\mathsf{OPT}_B(f(x)) \leq \alpha \cdot \mathsf{OPT}_A(x)$ holds.

2. there exists a positive constant $\beta$ such that for any problem instance $x$ from problem $A$ with solution $y$ of the problem instance $f(x)$,
$|\mathsf{OPT}_A(x) - c_A(g(y))| \leq \beta \cdot |\mathsf{OPT}_B(f(x)) - c_B(y)|$ holds.

We will now explain the components of an L-reduction by reducing the optimization version of the rectilinear picture rectangle cover problem to the optimization version of the Wheeler graph prefix interval cover problem[4] and thereby show that the Wheeler graph prefix interval cover problem is MaxSNP-hard. First of all, our cost functions are given by the number of rectangles/prefix intervals required to cover everything. The optimal values are given by the minimal number of rectangles/prefix

[4] Optimization versions here mean that we do not want to know if $k$ rectangles suffice to cover all black pixels. Instead, we want to use as few rectangles as possible to cover all black pixels, so a solution here is any choice of rectangles which cover all black pixels.

intervals needed to cover everything. The problem-translating function $f$ is given by rectilinear picture Wheeler graphs from Definition 3.14, so $f(x) = RPG(x)$. The solution-transforming function $g$ now converts any solution from $RPG(x)$ to a solution of $x$. In our case, given $k$ prefix intervals that cover all prefix intervals of $RPG(x)$, we can retranslate such a solution by determining rectangles of the minipixels underlying the nodes of a prefix interval. Then, by stretching those rectangles to the boundaries of the original pixels (similar as described in the proof of Theorem 3.16), we obtain a solution covering all black pixels in the original problem instance $x$. Both functions can clearly be computed in polynomial time.

For any rectilinear picture $x$, as proven in Theorem 3.16, any solution of rectangles covering black pixels corresponds to an equivalent solution of prefix intervals in $RPG(x)$ and vice versa. Therefore, the optimum solutions in $x$ and $RPG(x)$ are equivalent, so condition 1 of the L-reduction is satisfied using $\alpha = 1$. Moreover, as the solution-transforming function $g$ transforms $k$ prefix intervals to $k$ rectangles, we have $c_A(g(y)) = c_B(y)$ for any solution $y$ of the problem instance $RPG(x)$. Thus, condition 2 of L-reductions is satisfied using $\beta = 1$.

**Corollary 3.18.** *No PTAS for the Wheeler graph prefix interval cover problem exists unless P=NP.*

It could be possible that the Wheeler graph prefix interval cover problem has a constant factor approximation. We are not sure about such a result, but herein give the following quote:

> [...] In computational geometry, this problem received considerable attention in the past 25 years, in particular with respect to its complexity and approximability in a number of variants. Still, the intriguing main open question [5] is:
>
> > Is there a constant factor approximation algorithm for the rectangle cover problem?
>
> We do not answer this question now, but we offer a different and new kind of reply, which is “computationally, yes”. [...] [HLL07].

Because of the strong relation between prefix interval cover and rectangle cover, we suppose that it is not easy to design such an algorithm, but clearly state this as an open problem. We want to complete this section with some more remarks on the hardness of tunnel planning.

Alanko et al. [Ala+19] extended prefix intervals from parallel equally labeled paths to parallel equally labeled isomorphic subgraphs. In the case of tunnel planning, this special problem seems to have some similarity with the NP-complete subgraph isomorphism problem [GJ90, p. 202]. However, as the results from Theorem 3.16 and Corollary 3.17 hold for non-branching Wheeler graphs, they do so for more complex graphs like e.g. de Bruijn graphs or tries (Section 2.4). Thus, tunnel planning for more complex structured graphs will remain hard.

The attentive reader might have noticed that the Wheeler graph prefix interval cover problem does not precisely reflect the situation of tunnel planning. In Theorem 3.9 we have seen that overlapping prefix intervals can be tunneled only if they build a kind of "cross-overlay". According to the definition of the Wheeler graph prefix interval cover problem, the prefix intervals are allowed to overlap in any way. Thus, one would have to show that the rectilinear picture rectangle cover problem is NP-complete if the rectangles are allowed to overlap in the same "cross-overlay" shape. We strongly suppose that this special version of the problem is also NP-complete. The restriction seems to make the rectangle choice even harder, but we were not able to find this case in current literature and therefore state this as an open problem. Instead, in Chapter 4, we present a special restriction of prefix intervals, consisting of only overlayable prefix intervals. This restriction is very useful for data compression because the tunnel start and end markers can be represented succinctly.

In the case of non-overlapping prefix intervals it seems possible to find an efficient solution of the Wheeler graph prefix interval cover problem, as a polynomial time algorithm for the non-overlapping rectilinear picture rectangle cover problem exists [Oht82]. The problem with such an approach is that it is not straightforward to derive a picture from the prefix intervals in a Wheeler graph. For example, it is not clear how to assign the first nodes to some pixels. Also, the case of self-overlapping prefix intervals requires special handling, because this case would produce a kind of image tube instead of an image matrix. Finally, the dimension of the image is not directly clear: given a graph with $n$ nodes, in the simplest case, one could use an $n \times n$ binary matrix, but this would lead to an $\Omega(n^2)$ approach. In Chapter 5 we will instead present a special restriction of non-overlapping prefix intervals which can easily be computed and preserves some useful combinatorial properties of a BWT.

# 4 APPLICATION IN DATA COMPRESSION

The Burrows-Wheeler transform was invented for the purpose of data compression. 25 years after its first presentation in [BW94], Giovanni Manzini gave an inspiring talk about the BWT titled "The Past and the Future of an Unusual Compressor" during the Data Compression Conference 2019 [Man19]. Despite good compression properties, currently the only widely known BWT compressor is `bzip2` [Sew96].

The main problem of BWT based compressors is performance. The retransformation of a BWT requires "pseudo-random jumps" inside the string L using the LF-mapping, see Section 2.1.1. In contrast, most modern computer architectures use a memory cache hierarchy, making memory accesses faster the less distance there is between the previous and current RAM access [Dre07]. As a consequence, "pseudo-random jumping" inside the BWT has bad caching properties and makes BWT retransformation slow. There has been some research on improving the retransformation speed of a BWT [KKP12], but the speed is still slower than that of LZ77-based compressors.

The second performance problem of the BWT is its construction. Although a plethora of suffix array construction algorithms exist (see e.g. [PST07]), no algorithm has crystallized out itself as "the suffix array construction algorithm". It seems to be one of the hardest problems in computer science to devise a suffix array construction algorithm which

- uses almost no space (in the best case, it uses $O(1)$ space in RAM and writes the suffix array directly to disk).
- has a good worst-case run-time in theory.
- runs fast on current computer architectures.

There are some good candidates, see e.g. [Mor03; FK17; NZC09; Egi+19]. However, to be used in big data applications, more improvements would be desirable.

In this chapter, we will address a third issue which is related to BWT compressors: the compression rate itself. After the first good compression results of `bzip2`, researchers tried to improve the compression rate of BWT-based compressors with

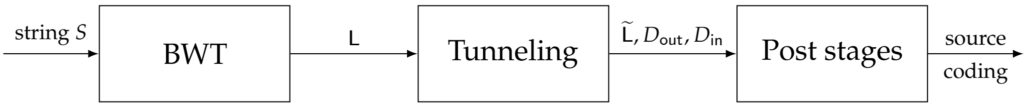


**Figure 4.1:** Integration of tunneling into the block-sorting compression chain, see also Figure 2.6. Tunneling works as an interim stage. Post stages are, for example, move-to-front transformation and source encoding, see Section 2.2.4.

varying degrees of success [Abe10; HMB06]. Nowadays, the compression rate of the oldest BWT compressor described by the inventors of the BWT themselves is still competitive to other BWT compressors developed over the past 26 years. Thus it is unlikely that the compression rate of BWT compressors can be improved much by encoding the BWT with a new mechanism.

Tunneling is a different mechanism, because it shortens a BWT but leaves its structure in the sense of backward steps intact. Also, tunneling does not recommend a special encoding procedure for the remaining BWT or the additional bit-vectors $D_{out}$ and $D_{in}$. Therefore, tunneling is a perfect interim stage for BWT compressors, see also Figure 4.1. In this chapter, we will present the necessary "engineering work" to include tunneling into the BWT compressor process chain. As we will see, this results in improved BWT compressors which are especially successful in compressing large and repetitive data. The chapter is based on the first publication about tunneling [Bai18] presented by the author of this thesis and further improvements made during supervised student projects [Ded18; Rät19].

## 4.1 RUN-TERMINATED PREFIX INTERVALS

The compressibility of a BWT mainly depends on the number of runs it contains. As a reminder, a run in a BWT is a length-maximal repetition of the same character, see Definition 2.9. In general, the smaller the number of runs in a BWT, the more compressible it is. There are numerous ways to encode a single run, see e.g. [Abe10]. One of the simplest and most popular ways to encode a run is given by the run-length encoding method shown in Section 2.2.2.

A successful integration of tunneling into the block-sorting compression chain requires one to find good "candidates" to be tunneled. More precisely, we have to find large prefix intervals which reduce the BWT length by a large amount. Furthermore, we should regard the compressibility of the additional BWT components $D_{in}$ and

$D_{\text{out}}$, as the additional encoding costs of these components may overcome the gain achieved by reducing the length of the BWT.

Unfortunately, finding "good candidates" turns out to be a difficult task when regarding all possible prefix intervals, see Section 3.4 about tunnel planning hardness. Moreover, we have to ensure that the candidates fulfill the special property of being overlayable. As a reminder, this property describes a special shape of overlapping prefix intervals and is required to invert a tunneled BWT, see Section 3.3.

In this chapter, we present a special restricted class of prefix intervals whose start column and end column coincide with runs in the underlying BWT. As we will see, this restriction allows us to compute the candidates efficiently and ensures the overlayability of the candidates. Furthermore, the length of the components $D_{\text{in}}$ and $D_{\text{out}}$ can be reduced to the number of runs in the BWT. As the number of runs is typically considerably less than the length of the BWT, this allows for a small encoding size of the additional components.

### 4.1.1 *Definition and properties*

We now want to give the definition of run-terminated prefix intervals. The idea of run-terminated prefix intervals was presented by the author of this thesis in [Bai18].

**Definition 4.1** (Run-terminated prefix intervals)**.** Let $S$ be a null-terminated string of length $n$ and let $\mathsf{L}$ be its BWT with LF-mapping $\mathsf{LF}$. Let $\mathcal{R}$ be the set of runs defined as $\mathcal{R} := \{\ [i,j] \subseteq [1,n] \ \mid\ [i,j] \text{ is a run of } \mathsf{L}\ \}$.

Let $\langle w, [i,j] \rangle$ be a prefix interval in $\mathsf{L}$. We call $\langle w, [i,j] \rangle$ a run-terminated prefix interval if and only if

- the start column coincides with a run, i.e. $[i,j] \in \mathcal{R}$.
- the end column coincides with a run, i.e. $[\mathsf{LF}^{w-1}[i], \mathsf{LF}^{w-1}[j]] \in \mathcal{R}$.

Additionally, for each run $[i,j] \in \mathcal{R}$, we define the tuple $\langle 1, [i,j] \rangle$ as a run-terminated prefix interval. A run-terminated prefix interval $\langle w, [i,j] \rangle$ is called length-maximal if it cannot be extended to the left or right. Formally, this means that no $x > 0$ exists such that $\langle w + x, [i,j] \rangle$ or $\langle w + x, [\mathsf{LF}^{-x}[i], \mathsf{LF}^{-x}[j]] \rangle$ is a run-terminated prefix interval.

As stated already, run-terminated prefix intervals have the property that their start- and end-columns coincide with runs of the underlying BWT. An example of run-terminated prefix intervals can be found in Figure 4.2. The definition also includes

| $i$ | $\mathsf{SA}[i]$ | run | $S[1..\mathsf{SA}[i]-1]$ | $S[\mathsf{SA}[i]..n]$ |
|---|---|---|---|---|
| 1 | 9 | $[1,3]$ | TCATCAGC | \$ |
| 2 | 6 | | TCATC | AGC\$ |
| 3 | 3 | | TC | ATCAGC\$ |
| 4 | 8 | $[4,4]$ | TCATCAG | C\$ |
| 5 | 5 | $[5,6]$ | TCAT | CAGC\$ |
| 6 | 2 | | T | CATCAGC\$ |
| 7 | 7 | $[7,8]$ | TCATCA | GC\$ |
| 8 | 4 | | TCA | TCAGC\$ |
| 9 | 1 | $[9,9]$ | $\varepsilon$ | TCATCAGC\$ |

**Figure 4.2:** Length-maximal run-terminated prefix intervals in the BWT of $S$ = TCATCAGC\$. The BWT contains the length-maximal run-terminated prefix intervals $P = \langle 3, [7,8] \rangle$ and $P' = \langle 1, [1,3] \rangle$. As a counter example, the prefix interval $\langle 4, [7,8] \rangle$ is not run-terminated, as its end-column $[\mathsf{LF}^3[7], \mathsf{LF}^3[8]] = [8,9]$ does not coincide with a run in the BWT.

each run as a special singleton run-terminated prefix interval. For tunneling, only the largest prefix intervals are subjects of interest. Therefore, Definition 4.1 includes conditions for the length-maximality of run-terminated prefix intervals.

Our first goal is to ensure that the conditions of tunneling from Definition 3.5 are fulfilled. To fulfill these conditions, the rows of a run-terminated prefix interval have to be node disjoint.

**Lemma 4.2.** *Let $S$ be a null-terminated string of length $n$ and let $\mathsf{L}$ be its BWT with LF-mapping $\mathsf{LF}$ and run set $\mathcal{R}$. Then, any run-terminated prefix interval $\langle w, [i,j] \rangle$ contained in $\mathsf{L}$ has disjoint rows. Define $\mathsf{row}_\mathsf{L}(y,x) := \{y, \mathsf{LF}^1[y], \ldots, \mathsf{LF}^{x-1}[y]\}$, then*

$$\mathsf{row}_\mathsf{L}(y,w) \cap \mathsf{row}_\mathsf{L}(y',w) = \emptyset \quad \textit{for all } i \le y < y' \le j.$$

*Proof.* Assume two rows $y < y'$ intersect, i.e. $\mathsf{row}_\mathsf{L}(y,w) \cap \mathsf{row}_\mathsf{L}(y',w)$. Because both rows are paths in the underlying Wheeler graph, the intersection must contain either $y$ or $y'$. W.l.o.g. assume that $y' \in \mathsf{row}_\mathsf{L}(y,w) \cap \mathsf{row}_\mathsf{L}(y',w)$ such that $\mathsf{LF}^x[y] = y'$. Because the rows belong to a run-terminated prefix interval, $\mathsf{L}[\mathsf{LF}^x[y] - y + i] = \ldots = \mathsf{L}[\mathsf{LF}^x[y] - y + j]$ must hold. Plugging the value $y' = \mathsf{LF}^x[y]$ in the equations shows that $\mathsf{L}[y' - y + i] = \ldots = \mathsf{L}[y' - y + j]$ must hold. Define $h := y' - y$, then $i \le y < y' \le j$ implies $h \in [1, j-i]$. Because $[i,j]$ is the start of the prefix interval, we can extend the above equality to $\mathsf{L}[i] = \ldots = \mathsf{L}[i+h] = \ldots = \mathsf{L}[j] = \ldots = \mathsf{L}[j+h]$. This means that $[i,j]$ is not a run because it can be extended to the bottom, so $\langle w, [i,j] \rangle$ is not a run-terminated prefix interval. □

Lemma 4.2 shows that it is possible to tunnel run-terminated prefix intervals. The next goal is to show that any selection of length-maximal run-terminated prefix intervals can be tunneled without causing problems.

**Lemma 4.3.** *Let $S$ be a null-terminated string of length $n$ and let $\mathsf{L}$ be its BWT with LF-mapping $\mathsf{LF}$. Let $\langle w, [i,j] \rangle$ and $\langle w', [i',j'] \rangle$ be two distinct length-maximal run-terminated prefix intervals. Then, $\langle w, [i,j] \rangle$ and $\langle w', [i',j'] \rangle$ are overlayable.*

*Proof.* Before we proof the theorem, we repeat the criteria for overlayability from Theorem 3.9. We also express the criteria in the form of BWT prefix interval notation instead of Wheeler graph prefix interval notation. For ease of expression, we use the definition $\mathsf{row}_\mathsf{L}(y,x) := \{y, \mathsf{LF}^1[y], \dots, \mathsf{LF}^{x-1}[y]\}$ from Lemma 4.2.

W.l.o.g. assume that $w \geq w'$. Then, $P = \langle w, [i,j] \rangle$ and $P' = \langle w', [i',j'] \rangle$ are overlayable if and only if

1. The uppermost row of $P'$ does not share an entry with any other row except for the uppermost row of $P$:
$$\mathsf{row}_\mathsf{L}(i', w') \cap \left( \bigcup_{k=i+1}^{j} \mathsf{row}_\mathsf{L}(k, w) \right) = \emptyset.$$
2. One more row excluding the uppermost row of $P'$ shares no entry with all rows except for the uppermost row of $P$. Formally, there exists a $k' \in [i'+1, j']$ such that
$$\mathsf{row}_\mathsf{L}(k', w') \cap \left( \bigcup_{k=i+1}^{j} \mathsf{row}_\mathsf{L}(k, w) \right) = \emptyset.$$
3. No row of $P'$ shares entries with the start- or end-column of $P$:
$$\left( \bigcup_{k'=i'}^{j'} \mathsf{row}_\mathsf{L}(k', w') \right) \cap \Big( [i,j] \cup [\mathsf{LF}^{w-1}[i], \mathsf{LF}^{w-1}[j]] \Big) = \emptyset.$$

Assume criteria 1 is false. Then there exists a $k \in [i+1, j]$ such that $\mathsf{row}_\mathsf{L}(i', w') \cap \mathsf{row}_\mathsf{L}(k, w) \neq \emptyset$. Because both rows are paths in the Wheeler graph of $\mathsf{L}$, the intersection must contain at least one of the entries $i'$ or $\mathsf{LF}^{w'-1}[i']$. Because $k$ refers to any row except of the uppermost row of $P$ and $P$ is a run-terminated prefix interval,

$\mathsf{L}[\mathsf{LF}^x[k-1]] = \mathsf{L}[\mathsf{LF}^x[k]]$ holds for all $x \in [0, w-1]$. Applying this result to one of the entries $i'$ or $\mathsf{LF}^{w'-1}[i']$ contained in the row intersection shows that either $\mathsf{L}[i'-1] = \mathsf{L}[i']$ or $\mathsf{L}[\mathsf{LF}^{w'-1}[i'-1]] = \mathsf{L}[\mathsf{LF}^{w'-1}[i']]$ must hold. As $i'$ is the uppermost row of $P'$, this implies that either $[i', j'] \notin \mathcal{R}$ or $[\mathsf{LF}^{w'-1}[i'], \mathsf{LF}^{w'-1}[j']] \notin \mathcal{R}$ because the corresponding run can be extended to the top. Thus, $P'$ is not a run-terminated prefix interval, which is a contradiction.

Now assume criteria 2 is wrong. Then, $\mathsf{row}_\mathsf{L}(i'+y, w') \cap \mathsf{row}_\mathsf{L}(i+y, w) \neq \emptyset$ holds for all $y \in [1, j'-i']$. As $P$ and $P'$ are prefix intervals with parallel paths, $\mathsf{row}_\mathsf{L}(i', w') \cap \mathsf{row}_\mathsf{L}(i, w) \neq \emptyset$ must hold too. We now distinguish two cases: in the first case, assume that $P$ is higher than $P'$, i.e. $j-i > j'-i'$. In the similar argument as above, we can see that at least one of the entries $j'$ or $\mathsf{LF}^{w'-1}[j']$ is contained in the row intersection $\mathsf{row}_\mathsf{L}(j', w') \cap \mathsf{row}_\mathsf{L}(i+j'-i', w)$. Analogously as above, this shows that either $[i', j'] \notin \mathcal{R}$ or $[\mathsf{LF}^{w'-1}[i'], \mathsf{LF}^{w'-1}[j']] \notin \mathcal{R}$ because the corresponding run can be extended to the bottom. Thus, $P'$ is not a run-terminated prefix interval, which is a contradiction. In the second case, assume that both prefix intervals have the same height, i.e. $j-i = j'-i'$. In this case, as $w \geq w'$ holds, $P$ is a left- or right-extension of $P'$. This implies a contradiction because $P'$ would then not be length-maximal.

Finally, assume criteria 3 is wrong. W.l.o.g. assume that the start-column of $P$ shares entries with the rows of $P'$. Let $x \in [0, w-1]$ such that $[i, j] \cap [\mathsf{LF}^x[i'], \mathsf{LF}^x[j']] \neq \emptyset$. As $P$ is run-terminated, $[i, j] \in \mathcal{R}$ must hold. This implies $[\mathsf{LF}^x[i'], \mathsf{LF}^x[j']] \subseteq [i, j]$, because otherwise, $P'$ would not be a prefix interval as it would contain different characters in its $x$-th column. First, assume that $[\mathsf{LF}^x[i'], \mathsf{LF}^x[j']] = [i, j]$. As $w \geq w'$ holds, this means that we can extend $P'$ to the run-terminated prefix interval $P$, so $P'$ is not length-maximal. Second, assume $[\mathsf{LF}^x[i'], \mathsf{LF}^x[j']] \subset [i, j]$. Because $w \geq w'$ holds, the last column of $P'$ must be contained within the $w'-x$-th column of $P$, i.e. $[\mathsf{LF}^{w'-1}[i'], \mathsf{LF}^{w'-1}[j']] \subset [\mathsf{LF}^{w'-x-1}[i], \mathsf{LF}^{w'-x-1}[j]]$. This implies that the end column of $P'$ does not coincide with a run, so $P'$ is not a run-terminated prefix interval. The case of shared entries in the end-column of $P$ follows analogously and also implies a contradiction. □

After ensuring that tunneling of run-terminated prefix intervals causes no problems or side-effects between prefix intervals, the last lemma of this section presents results about the dimensions of such intervals.

**Lemma 4.4.** *Let $S$ be a null-terminated string of length $n$ and let $\mathsf{L}$ be its BWT with LF-mapping $\mathsf{LF}$. Let $\mathcal{R}$ be the set of runs in $\mathsf{L}$, and let $\mathcal{RP}$ be the set of all length-maximal run-terminated prefix intervals in $\mathsf{L}$.*

- *The sum of heights of $\mathcal{RP}$ is bound by n: $\sum_{\langle w,[i,j]\rangle \in \mathcal{RP}}(j-i+1) \leq n$.*
- *The sum of widths of $\mathcal{RP}$ is bound by n: $\sum_{\langle w,[i,j]\rangle \in \mathcal{RP}} w \leq n$.*

*Proof.* We denote by LMRTPI a length-maximal run-terminated prefix interval. As LMRTPIs are length-maximal, at most one LMRTPI can start at a run. We thus obtain $\sum_{\langle w,[i,j]\rangle \in \mathcal{RP}}(j-i+1) \leq \sum_{\langle w,[i,j]\rangle \in \mathcal{R}}(j-i+1) = n$. For the sum of widths, we show that at most $h-1$ LMRTPIs can point through a run of height $h$. More precisely, for a run $[i,j] \in \mathcal{R}$, at most $j-i$ LMRTPIs $\langle w',[i',j']\rangle$ can exist such that $(\bigcup_{k'=i'}^{j'} \mathsf{row}_\mathsf{L}(k',w')) \cap [i,j] \neq \emptyset$. We will proof the result later, but beforehand show how it can be used to finish the proof.

Summing the weights of LMRTPIs is equal to counting the number of columns contained in all LMRTPIs. Each column of an LMRTPI is a subsequence of a run in the BWT. Therefore, counting how many LMRTPIs point through a run is equal to counting how many columns of LMRTPIs are subsequences of the run. Thus, counting the number of LMRTPIs that point through all runs of the BWT is equal to the sum of widths of all LMRTPIs. Using the upper bound of $j-i$ LMRTPIS pointing through a run $[i,j] \in \mathcal{R}$, we obtain

$$\sum_{\langle w,[i,j]\rangle \in \mathcal{RP}} w \leq \sum_{\langle w,[i,j]\rangle \in \mathcal{RP}} (j-i) \leq n.$$

Now, we will proof that at most $j-i$ LMRTPIs can point through a run $[i,j] \in \mathcal{R}$. The proof is done using an induction over the height $j-i+1$ of a run. Clearly, no LMRTPI points through a run with the height of 1, as prefix intervals require a minimal height of 2.

First, consider a run $[i,j]$ with the height $j-i+1=2$. In this case, the run may or may not belong to an LMRTPI. However, less than two LMRTPIs can point through the run, because otherwise, the LMRTPIs would not be length-maximal. Therefore, the number of LMRTPIs that point through the run is limited by $j-i=1$.

Now, consider a run $[i,j] \in \mathcal{R}$ with height $j-i+1>2$. Let $P$ be the set of (normal) prefix intervals that start in a run and end in $[i,j]$, i.e. $\langle w',[i',j']\rangle \in P \Leftrightarrow [i',j'] \in \mathcal{R}$ and $[\mathsf{LF}^{w'-1}[i'], \mathsf{LF}^{w'-1}[y']] \subseteq [i,j]$. Furthermore, let $\langle w_1,[i_1,j_1]\rangle, \ldots, \langle w_m,[i_m,j_m]\rangle \in P$ be the prefix intervals which are nearest to $[i,j]$, i.e. any prefix interval $\langle w',[i',j']\rangle \in P$ which overlaps with a prefix interval $\langle w_k,[i_k,j_k]\rangle$ satisfies $w' \geq w_k$.

All LMRTPIs not starting at $[i,j]$ but pointing through $[i,j]$ then also point through the runs $[i_1,j_1],\dots,[i_m,j_m]$. Because $\langle w_1,[i_1,j_1]\rangle,\dots,\langle w_m,[i_m,j_m]\rangle$ cannot overlap due to their definition, the following must hold:

$$\sum_{k=1}^{m}(j_k-i_k+1)\le(j-i+1). \tag{4.1}$$

Assume that $j_k-i_k+1<j-i+1$ holds for all $k\in[1,m]$. Then, by induction, we can see that at most $j_k-i_k$ LMRTPIs point through $[i_k,j_k]$. This means that at most $1+\sum_{k=1}^{m}(j_k-i_k)$ LMRTPIs point through $[i,j]$ (plus one because $[i,j]$ could be the start of an LMRTPI itself). In the case $m=1$, this number can be bound using $1+(j_1-i_1)\le j-i$. In the case $m>1$, we can use the inequality (4.1) and obtain $1+\sum_{k=1}^{m}(j_k-i_k)=1-m+\sum_{k=1}^{m}(j_k-i_k+1)\le 1-m+j-i+1\le j-i$.

Finally, assume that $j_k-i_k+1=j-i+1$ holds for all $k\in[1,m]$. Then clearly, $m=1$ holds because otherwise inequality (4.1) is wrong. This means that $[i,j]$ has only one predecessing run with the same height. Thus, $[i,j]$ cannot be the start of an LMRTPI, because otherwise, it could be extended to the right. The number of LMRTPIs pointing through $[i,j]$ then is identical to the number of LMRTPIs pointing through $[i_1,j_1]$. Repeatedly applying this argument until a run $[i',j']$ with multiple or smaller predecessing runs is reached then shows that the number of LMRTPIs pointing through $[i,j]$ is less than or equal to $j-i$. □

The Lemmas 4.2, 4.3 and 4.4 describe important properties of length-maximal run-terminated prefix intervals. We summarize the results in Corollary 4.5.

**Corollary 4.5.** *Let $S$ be a null-terminated string of length $n$, let $\mathsf{L}$ be its BWT and let $\mathcal{RP}$ be the set of all length-maximal run-terminated prefix intervals in $\mathsf{L}$.*

- *Each length-maximal run-terminated prefix interval consists of disjoint rows and therefore can be tunneled.*
- *Each subset $RP\subseteq\mathcal{RP}$ contains pairwise overlayable prefix intervals. As a result, each subset $RP\in\mathcal{RP}$ of prefix intervals can be tunneled.*
- *The sum of the heights of all prefix intervals in $\mathcal{RP}$ is less than or equal to $n$.*
- *The sum of the widths of all prefix intervals in $\mathcal{RP}$ is less than or equal to $n$.*

### 4.1.2 *Computation*

After showing some basic properties of length-maximal run-terminated prefix intervals, the next goal is to present an efficient algorithm for their computation. As it turns out, the property of run-termination allows one to do this computation relatively easily and also memory efficient. The results of this section come from the full version of the first paper which presented tunneling [Bai18], published by the author of this thesis.

Before an algorithm is described, we want to present a "toolbox" which eases the handling of runs and LF-mapping navigation in a BWT.

**Definition 4.6** (Run-LF support)**.** Let $S$ be a null-terminated string of length $n$, let $\mathsf{L}$ be its BWT with LF-mapping $\mathsf{LF}$, and let $\mathcal{R}$ be the set of runs in $\mathsf{L}$.

The Run-LF support of $\mathsf{L}$ consists of an array $\mathsf{LF}_{\mathcal{R}}$ of size $|\mathcal{R}|$ and a bit-vector $B_{\mathcal{R}}$ of length $n+1$ which are defined as follows:

- $B_{\mathcal{R}}[i..j+1] = 10^{j-i}1 \Leftrightarrow [i,j] \in \mathcal{R}$.
- $\mathsf{LF}_{\mathcal{R}}[k] := \mathsf{LF}[\mathsf{select}_{B_{\mathcal{R}}}(1,k)]$.

Examples of a Run-LF support from Definition 4.6 can be found in Figure 4.3. The bit-vector $B_{\mathcal{R}}$ can be used to get information about runs. For example, let $y \in [1,n]$ be a position and suppose we want to know the run $[i,j] \in \mathcal{R}$ which satisfies $y \in [i,j]$. We start by setting $k \leftarrow \mathsf{rank}_{B_{\mathcal{R}}}(1,y)$ to get the run identifier in which $y$ lies. Then, the run boundaries can be found using $i \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1,k)$ and $j \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1,k+1)-1$.

To support the LF-mapping, it suffices to store only the first LF-mapping of each run. In the case of a run $[i,j]$, $\mathsf{L}[i] = \cdots = \mathsf{L}[j]$ holds, so the LF-mapping runs "in

| $i$ | $k$ | $\mathsf{LF}[i]$ | $\mathsf{LF}_{\mathcal{R}}[k]$ | $B_{\mathcal{R}}[i]$ | $\mathsf{L}[i]$ | $S[\mathsf{SA}[i]..n]$ |
|---|---|---|---|---|---|---|
| 1 | 1 | 4 | 4 | 1 | C | $ |
| 2 |  | 5 |  | 0 | C | AGC$ |
| 3 |  | 6 |  | 0 | C | ATCAGC$ |
| 4 | 2 | 7 | 7 | 1 | G | C$ |
| 5 | 3 | 8 | 8 | 1 | T | CAGC$ |
| 6 |  | 9 |  | 0 | T | CATCAGC$ |
| 7 | 4 | 2 | 2 | 1 | A | GC$ |
| 8 |  | 3 |  | 0 | A | TCAGC$ |
| 9 | 5 | 1 | 1 | 1 | $ | TCATCAGC$ |
| 10 |  |  |  | 1 |  |  |

**Figure 4.3:** Run-LF support for the BWT $\mathsf{L}$ of the string $S =$ TCATCAGC$.

parallel". More precisely, $\mathsf{LF}[i+h] = \mathsf{LF}[i] + h$ holds for all $h \in [0, j-i]$. This means that we can compute the LF-mapping at a position $y$ by first computing the run identifier $k \leftarrow \mathsf{rank}_{B_\mathcal{R}}(1, y)$. Now, we can compute the offset $h$ to the uppermost entry of the run using $h \leftarrow y - \mathsf{select}_{B_\mathcal{R}}(1, k)$. The parallelism of the LF-mapping then ensures that $\mathsf{LF}[y] = \mathsf{LF}_\mathcal{R}[k] + h$.

The computation of a Run-LF support and the initialization of rank- and select-support on $B_\mathcal{R}$ can be done in $O(n)$ time and is straightforward. This support suffices to compute all length-maximal run-terminated prefix intervals in a BWT. For the computation we will use two additional arrays PE and RPE. The array PE is an abbreviation for prefix interval end: for each run $[i, j]$, let $\langle w, [i, j] \rangle$ be the left-maximal prefix interval starting at the run, meaning that $\langle w+1, [i, j] \rangle$ is no prefix interval. The array PE then stores the entry $\mathsf{LF}^w[i]$. More precisely,

$$\langle w, [i, j] \rangle \text{ is a left-maximal prefix interval and } [i, j] \in \mathcal{R} \Leftrightarrow \mathsf{PE}[\mathsf{rank}_{B_\mathcal{R}}(1, i)] = \mathsf{LF}^w[i].$$

Before we explain the array RPE, we will first explain an incremental algorithm to compute PE. The algorithm is implicitly contained in Algorithm 4.1. We start by setting $\mathsf{PE} \leftarrow \mathsf{LF}_\mathcal{R}$, because each run implies a prefix interval of length one.

Then, for each run $[i, j]$ with $k = \mathsf{rank}_{B_\mathcal{R}}(1, i)$, we search for the longest left-extension of a prefix interval. We do this by checking if $[\mathsf{PE}[k], \mathsf{PE}[k] + (j-i)] = [\mathsf{LF}^w[i], \mathsf{LF}^w[j]]$ is a subrange of a run $[i', j'] \in \mathcal{R}$, i.e. $[\mathsf{PE}[k], \mathsf{PE}[k] + (j-i)] \subseteq [i', j']$. In this case, a left extension is possible. Clearly, the prefix interval starting at $[i, j]$ can be extended by the same amount of columns that a left-maximal prefix interval starting in $[i', j']$ has. Therefore, we recursively search for the longest left-extension of a prefix interval starting at $[i', j']$.

Assume this longest extension has been found. We can then update the value $\mathsf{PE}[k]$ as follows: let $k' = \mathsf{rank}_{B_\mathcal{R}}(1, i')$ be the run identifier of $[i', j']$. Note that until now, $\mathsf{PE}[k] \in [i', j']$ holds. In the case that $\mathsf{PE}[k] = i$ holds, the topmost row of both prefix intervals coincide, so we can set $\mathsf{PE}[k] \leftarrow \mathsf{PE}[k']$. If this is not the case, as the rows in a prefix interval run in parallel, we need to add the offset between the topmost row of $[i, j]$ and the topmost row of $[i', j']$ to the value of $\mathsf{PE}[k']$. This ensures a valid modification of $\mathsf{PE}[k]$. Therefore, we can use the formula $\mathsf{PE}[k] \leftarrow \mathsf{PE}[k'] + (\mathsf{PE}[k] - i)$ to left-extend the prefix interval of $[i, j]$. Afterwards, we search for the next left-extension of $[i, j]$.

Algorithm 4.1 uses this as follows: it starts by iterating over all runs $[i, j]$ with height $j - i + 1 \geq 2$ (lines 4–7). Then, the run $k$ is pushed to a stack to allow recursive

```
Data: Run-LF support of the underlying BWT L with runs $\mathcal{R}$.
Result: Array RPE indicating the upper left corner of all length-maximal run-terminated prefix intervals.

1  initialize an empty stack $s$
2  let PE be a copy of $\mathsf{LF}_{\mathcal{R}}$
3  let RPE be a copy of $\mathsf{LF}_{\mathcal{R}}$
4  for $r \leftarrow 1$ to $|\mathcal{R}|$ do
5      $k \leftarrow r$
6      $i \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k)$
7      $j \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k+1) - 1$
8      push element $k$ on the stack $s$
9      repeat
10         $k' \leftarrow \mathsf{rank}_{B_{\mathcal{R}}}(1, \mathsf{PE}[k])$
11         if $k' = \mathsf{rank}_{B_{\mathcal{R}}}(1, \mathsf{PE}[k] + j - i)$ then          // $k$ can be left-extended
12             $k \leftarrow k'$
13             $i \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k)$
14             $j \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k+1) - 1$
15             push element $k$ on the stack $s$
16         else                                                                 // $k$ can not be left-extended
17             $k' = k$
18             $i' = i$
19             $j' = j$
20             pop topmost element of stack $s$                                  // pop element $k$
21             if stack $s$ is not empty then
22                 $k \leftarrow$ top of stack $s$
23                 $i \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k)$
24                 $j \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k+1) - 1$
25                 $\mathsf{PE}[k] \leftarrow \mathsf{PE}[k'] + (\mathsf{PE}[k] - i)$
26                 if $j - i = j' - i'$ then  // run-terminated prefix interval starting at $k$ can be extended
27                     $\mathsf{RPE}[k] \leftarrow \mathsf{RPE}[k']$
28                     $\mathsf{RPE}[k'] \leftarrow \bot$                              // $k'$ is not length-maximal
29     until stack $s$ is empty
30 return RPE
```

**Algorithm 4.1:** Computation of length-maximal run-terminated prefix intervals. A modified version of this algorithm has already been published in the full version of [Bai18], licensed under CC BY 3.0, http://creativecommons.org/licenses/by/3.0/.

left-extensions (line 8). In the case that the prefix interval starting at $k$ can be left-extended, the algorithm recursively proceeds with the run $k'$ (lines 11–15). In the case that the prefix interval starting at $k$ cannot be extended, $k$ is popped from the stack (lines 17–20). If the stack is not empty, the previous run $k$ is popped from the stack and the prefix interval end pointers PE are adapted (lines 25). The next iteration then checks if further left-extensions of the prefix interval starting at $k$ are possible.

Note that it is possible that a run $k$ is processed multiple times. This however causes no problems because a maximal left extension is found in the first processing. The later processings of the same run will then try to find additional left-extensions which have no effect on the value of $\mathsf{PE}[k]$.

We now come to the array RPE. Similar to the array PE, for each run $[i, j]$ it stores the value $\mathsf{LF}^{w}[i]$ in the case that a run-terminated length-maximal prefix interval

$\langle w, [i, j] \rangle$ exists. Note the subtle difference to PE: PE considers only left-maximality while RPE considers length-maximality. In the case that no such prefix interval exists, the array stores the value $\perp$ instead.

**Definition 4.7** (RPE array)**.** Let $S$ be a null-terminated string of length $n$ with BWT L and Run-LF support. Let $[i, j] \in \mathcal{R}$ be a run with run identifier $k = \mathsf{rank}_{B_{\mathcal{R}}}(1, i)$. Then,

$$\mathsf{RPE}[k] := \begin{cases} \mathsf{LF}^{w}[i] & \text{, if a length-maximal run-terminated prefix interval} \\ & \quad \langle w, [i, j] \rangle \text{ exists.} \\ \perp & \text{, else.} \end{cases}$$

The array RPE can be computed as a by-product of the computation of PE. One starts by setting $\mathsf{RPE}[k]$ to $\mathsf{LF}_{\mathcal{R}}[k]$ in Algorithm 4.1. This means that the value $\mathsf{RPE}[k]$ induces a run-terminated prefix interval of length one at each run, which is not necessarily length-maximal.

Now suppose a run $[i, j]$ with run identifier $k = \mathsf{rank}_{B_{\mathcal{R}}}(1, i)$ is placed onto the stack $s$ for the first time. The stack $s$ maintains following invariant: viewing the stack from top to bottom, the runs have descending heights. A run that is pushed onto $s$ has a height that is bigger or equal to the heights of all remaining runs on the stack. This is implied by lines 10–15 of Algorithm 4.1. The invariant implies that a run-terminated prefix interval starting at a run with identifier $k$ can be left-extended if and only if another run $[i', j']$ with equal height $j' - i' + 1 = j - i + 1$ is placed immediately above $k$ on the stack. Lines 10–15 ensure that a prefix interval from the run with identifier $k$ to the run $[i', j']$ with identifier $k'$ is spanned. The invariant itself ensures that $k'$ must be placed immediately above $k$ on the stack.

If we have this situation ($k'$ is immediately above $k$ on the stack, $k'$ is popped, both runs have the same height) we can extend the run-terminated prefix interval of $k$. Because $k'$ is popped, we can assume that $\mathsf{RPE}[k']$ induces a left-maximal run-terminated prefix interval starting at $k'$. Thus, by setting $\mathsf{RPE}[k] = \mathsf{RPE}[k']$, a left-maximal run-terminated prefix interval starting at $k$ is induced. Because $k'$ was pushed to the stack after $k$, $\mathsf{RPE}[k]$ induces a longer prefix interval than $\mathsf{RPE}[k']$. This means that no length-maximal run-terminated prefix interval can start at run $k'$. Therefore, we set $\mathsf{RPE}[k'] = \perp$ to match Definition 4.7 of the RPE-array. The whole left-extension process is shown in lines 26–28 of Algorithm 4.1.

In other words, let $\langle w, [i, j] \rangle$ be a length-maximal run-terminated prefix interval, and let $0 = x_1 < \ldots < x_m = w - 1$ be numbers such that $[\mathsf{LF}^{x_r}[i], \mathsf{LF}^{x_r}[j]] \in \mathcal{R}$ for all $r \in [1, m]$. The numbers $x_1, \ldots, x_m$ thus describe columns of the prefix interval that coincide with runs. Algorithm 4.1 then starts by left-extending the run-terminated prefix interval starting at $x_{m-1}$. This left-extension ends at $x_m$, and the run-terminated prefix interval starting at $x_m$ is not length-maximal, so the RPE-value is cleared. Afterwards, it left-extends the run-terminated prefix interval starting at $x_{m-2}$ and clears the RPE-value of $x_{m-1}$. Repeating this process shows that only the length-maximal run-terminated prefix interval starting at $x_1$ remains, all other left-extensions are cleared.

Note that it is still unproblematic if a run $k$ is pushed onto the stack multiple times. After the first time, $\mathsf{PE}[k]$ induces a left-maximal prefix interval starting at $k$. Also, $\mathsf{RPE}[k]$ induces a run-terminated left-maximal prefix interval starting at $k$. When $k$ is pushed onto the stack $s$ again, it might be the case that $\mathsf{RPE}[k]$ is set to $\perp$ because the prefix interval was not length-maximal. However, $\mathsf{PE}[k]$ still points to the end of the length-maximal prefix interval starting at $k$. Therefore, no other run $k'$ can be pushed onto the stack when $k$ is handled a second time. Therefore, the lines 21–28 will not be executed, and the value of $\mathsf{RPE}[k]$ remains unchanged.

We now come to the run-time of Algorithm 4.1. This run-time is determined by the overall number of executions of the inner loop from lines 9–29. Within each iteration of the loop, a run is either pushed to the stack or popped from the stack. The overall run-time thus equals two times the number of runs that are pushed to the stack during execution.

Each run $[i, j]$ with identifier $k$ is pushed on the stack once by line 8 of Algorithm 4.1. Additionally, $k$ can be pushed to $s$ by line 15. This can happen at most $j - i + 1$ times: Before the outer loop (lines 4–29), at most $j - i + 1$ PE-values point into $[i, j]$, i.e. $\mathsf{PE}[k'] \in [i, j]$ for some $k' \in [1, |\mathcal{R}|]$. The PE-values pointing into $[i, j]$ must be distinct because they are inherited from the LF-mapping. During the algorithm, two cases can happen: in the first case, a $\mathsf{PE}[k']$-value belongs to a run where a left-extension through $k$ is not possible. In this case, $k$ is not pushed onto the stack during the whole algorithm. In the second case, $\mathsf{PE}[k']$ belongs to a run where a left-extension through $k$ is possible. In this case, after $k'$ has been popped from the stack, $\mathsf{PE}[k']$ is updated in line 25 so $\mathsf{PE}[k'] \notin [i, j]$ holds afterwards. This means that the run $k'$ cannot cause a second push of $k$ onto the stack.

To summarize, each run $[i,j]$ is pushed at most $j-i+2$ times to $s$. The overall number of runs pushed to the stack can therefore be bound by $\sum_{[i,j]\in\mathcal{R}} j-i+2 = |\mathcal{R}| + \sum_{[i,j]\in\mathcal{R}} j-i+1 = |\mathcal{R}| + n$. The overall run-time thus is $O(|\mathcal{R}|+n) = O(n)$.

**Corollary 4.8.** *Let $S$ be a null-terminated string of length $n$ with BWT $\mathsf{L}$. The set $\mathcal{RP}$ of all length-maximal run-terminated prefix intervals can be computed in $O(n)$ time using the following steps:*

- *Computation of a Run-LF support for $\mathsf{L}$ in $O(n)$ time.*
- *Computation of the* RPE*-array with Algorithm 4.1 in $O(n)$ time.*

A summary of the results from this section is given in Corollary 4.8. Despite the non-trivial arguments for the correctness and worst-case run-time of Algorithm 4.1, the algorithm itself is easy enough to be implemented.

Regarding the integration of tunneling into the block-sorting compression chain, we have finished the first part: devising a restricted class of prefix intervals where

- any subset of the prefix intervals can be tunneled.
- efficient computation of the prefix intervals is possible.

Our next goal will be to show how the components $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ can be efficiently encoded.

### 4.1.3 *Tunnel encoding*

In Section 3.2 we have seen how a tunneled BWT can be encoded. A tunneled BWT requires two additional bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ which are used to encode the starts and the ends of the tunnels. A possible mechanism to tunnel a single prefix interval then works as follows: let $\langle w,[i,j]\rangle$ be a prefix interval of width $w \geq 2$ in a string $S$ of length $n$ with BWT $\mathsf{L}$.

1. initialize two bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ of size $n+1$ with ones.
2. initialize a bit-vector $M$ of size $n$ with zeros.
3. mark the tunnel start by setting $D_{\mathsf{in}}[i..j] = \mathsf{10}^{j-i}$.
4. mark the tunnel end by setting $D_{\mathsf{out}}[\mathsf{LF}^{w-1}[i]..\mathsf{LF}^{w-1}[j]] = \mathsf{10}^{j-i}$.

5. mark the entries to be removed by setting $M[\mathsf{LF}^x[i+1]..\mathsf{LF}^x[j]] = 1^{j-i}$ for all $x \in [0, w-2]$.

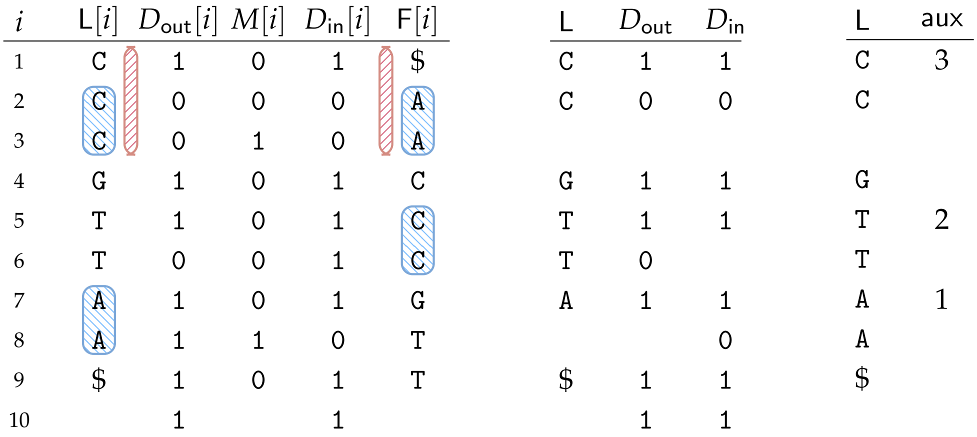
6. remove entries $\mathsf{L}[y]$, $D_{\text{out}}[y]$ and $D_{\text{in}}[\mathsf{LF}[y]]$ when $M[y] = 1$ holds.

An exemplaric execution of this mechanism is shown in Figure 4.4.

In the case of run-terminated prefix intervals, the start and end column of each prefix interval coincide with a run. For any run-terminated prefix interval $\langle w, [i,j] \rangle$, $[i,j] \in \mathcal{R}$ and $[\mathsf{LF}^{w-1}[i], \mathsf{LF}^{w-1}[j]] \in \mathcal{R}$ holds. Also, if only length-maximal run-terminated prefix intervals are tunneled, the prefix intervals are overlayable. This implies that a single run $[i,j] \in \mathcal{R}$ is either

- a normal run where prefix intervals can point through.
- the start of a run-terminated prefix interval.
- the end of a run-terminated prefix interval.
- both the start and end of a run-terminated prefix interval if the width of the prefix interval is one.

The idea for the encoding of a tunneled BWT when only length-maximal run-terminated prefix intervals are tunneled comes from [Bai18]. Instead of encoding the



| $i$ | $\mathsf{L}[i]$ | $D_{\text{out}}[i]$ | $M[i]$ | $D_{\text{in}}[i]$ | $\mathsf{F}[i]$ |
|---|---|---|---|---|---|
| 1 | C | 1 | 0 | 1 | $ |
| 2 | C | 0 | 0 | 0 | A |
| 3 | C | 0 | 1 | 0 | A |
| 4 | G | 1 | 0 | 1 | C |
| 5 | T | 1 | 0 | 1 | C |
| 6 | T | 0 | 0 | 1 | C |
| 7 | A | 1 | 0 | 1 | G |
| 8 | A | 1 | 1 | 0 | T |
| 9 | $ | 1 | 0 | 1 | T |
| 10 | | 1 | | 1 | |

| L | $D_{\text{out}}$ | $D_{\text{in}}$ |
|---|---|---|
| C | 1 | 1 |
| C | 0 | 0 |
| | | |
| G | 1 | 1 |
| T | 1 | 1 |
| T | 0 | |
| A | 1 | 1 |
| | | 0 |
| $ | 1 | 1 |
| | 1 | 1 |

| L | aux |
|---|---|
| C | 3 |
| C | |
| | |
| G | |
| T | 2 |
| T | |
| A | 1 |
| A | |
| $ | |

**Figure 4.4:** Encoding variants of a tunneled BWT with length-maximal run-terminated prefix intervals. The BWT and the prefix intervals $P = \langle 3, [7,8] \rangle$ and $P' = \langle 1, [1,3] \rangle$ are identical to those of Figure 4.2. The left-hand side shows the columns of the prefix intervals marked in L and F. Additionally, the components of the tunneling mechanism before entry removal are shown. The middle shows the emerging tunneled BWT. The right-hand side shows a special encoding where the information of a tunnel start and end is saved only for the runs of the BWT.

tunnel starts and ends in two bit-vectors $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$, the information is contained in a single vector aux of a maximum size of $|\mathcal{R}|$. The vector aux is a fusion of $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ which is trimmed to a maximum length of $|\mathcal{R}|$. The vector has an alphabet size of two, indicating which of the above cases pertain to a run. Information about the height of a tunnel start or end is acquired by measuring the height of the corresponding run in L. As a tunnel must then start or end at a run in L with a height greater than one, we can reduce the size of aux to $|\mathcal{R}_{h>1}|$, where

$$\mathcal{R}_{h>1} := \{\ [i,j] \in \mathcal{R}\ \mid\ j-i+1 \geq 2\ \}.$$

The set $\mathcal{R}_{h>1}$ indicates the number of runs in the tunneled version of L with a height greater than 1. A definition of aux is given by

$$\mathsf{aux}[k] := \begin{cases} 0 & \text{, if no tunnel starts or ends at the } k\text{-th run with a height greater one.} \\ 1 & \text{, if a tunnel starts at the } k\text{-th run with a height greater one.} \\ 2 & \text{, if a tunnel ends at the } k\text{-th run with a height greater one.} \\ 3 & \text{, if a tunnel starts and ends at the } k\text{-th run with a height greater one.} \end{cases}$$

Figure 4.4 shows an example of this special encoding. Regarding the “normal mechanism” of tunneling a prefix interval, the following changes have to be made:

- tunnel starts and ends are marked in a vector aux of length $|\mathcal{R}|$. After the tunneling process, the vector aux is trimmed to length $|\mathcal{R}_{h>1}|$.
- no entries at the start or end of a tunnel should be marked by the mechanism. This ensures that the height information of a tunnel is available.

Special attention should be given to the second item in the above list. In the case that two length-maximal run-terminated prefix intervals overlap, the wider prefix interval will mark entries in the start and end of the shorter prefix interval. While this is not a problem, it is important that no entries at the start or end of a prefix interval are marked by the marking routine of the prefix interval itself. Moreover, as run-terminated prefix intervals can overlap, we must avoid setting duplicate markings in $M$. This means that we must not set an entry $M[i] \leftarrow 1$ multiple times to ensure linear run-time.

Algorithm 4.2 shows an encoding mechanism for this special situation. The algorithm expects a modified version of the array RPE from Definition 4.7 as input.

**Data:** BWT L of a null-terminated string $S$ of length $n$, Run-LF support, RPE-array where all prefix intervals which should not be tunneled contain a $\perp$.
**Result:** Encoding of a tunneled BWT with the components L and aux.

```
initialize a bit-vector $M$ of size $n$ with zeros
initialize an array aux of size $|\mathcal{R}|$ with zeros
for $r \leftarrow 1$ to $|\mathcal{R}|$ do
    if $\mathsf{RPE}[r] \neq \perp$ then
        $k \leftarrow r$
        $i \leftarrow \mathsf{select}_{B_\mathcal{R}}(1, k)$
        $j \leftarrow \mathsf{select}_{B_\mathcal{R}}(1, k+1) - 1$
        $\mathsf{aux}[k] \leftarrow \mathsf{aux}[k] + 1$                    // set tunnel start marking
        $k' \leftarrow k$
        $i' \leftarrow \mathsf{LF}_\mathcal{R}[k]$
        while $i' \neq \mathsf{RPE}[k]$ do                                    // mark entries to be removed
            $M[i'+1..i'+j-i] \leftarrow 1^{j-i}$
            $k' \leftarrow \mathsf{rank}_{B_\mathcal{R}}(1, i')$
            $h \leftarrow i' - \mathsf{select}_{B_\mathcal{R}}(1, k')$
            if $\mathsf{RPE}[k'] = \perp$ then             // no prefix interval, use normal navigation
                $i' \leftarrow \mathsf{LF}_\mathcal{R}[k'] + h$
            else                                                // jump over prefix interval using RPE
                $i' \leftarrow \mathsf{RPE}[k'] + h$
        $\mathsf{aux}[k'] \leftarrow \mathsf{aux}[k'] + 2$                          // set tunnel end marking
        $\mathsf{RPE}[k'] \leftarrow i$                         // set a pointer from the last column to $i$

                       // correct markings of end-columns using pointers to start columns
for $k \leftarrow 1$ to $|\mathcal{R}|$ do
    if $\mathsf{aux}[k] \geq 2$ then
        $i \leftarrow \mathsf{RPE}[k]$
        $i' \leftarrow \mathsf{select}_{B_\mathcal{R}}(1, k)$
        $j' \leftarrow \mathsf{select}_{B_\mathcal{R}}(1, k+1) - 1$
        $M[i'..j'] \leftarrow M[i..i+j'-i']$

                                                                      // shorten L and aux
$p \leftarrow 1$                                                        // output position in L
$q \leftarrow 1$                                                      // output position in aux
$k \leftarrow 0$                                                                 // current run
$c \leftarrow \$$                                                    // character of current run
$h_1 \leftarrow 1$                                                // current run has height one
for $i \leftarrow 1$ to $n$ do
    if $M[i] = 0$ then
        $\mathsf{L}[p] \leftarrow \mathsf{L}[i]$
        $p \leftarrow p + 1$
        if $c \neq \mathsf{L}[i]$ then                                          // start of a new run
            $c \leftarrow \mathsf{L}[i]$
            $k \leftarrow k + 1$
            $h_1 \leftarrow 1$
        else if $h_1 = 1$ then // current run has height greater than one and is processed the first time
            $\mathsf{aux}[q] \leftarrow \mathsf{aux}[k]$
            $q \leftarrow q + 1$
            $h_1 \leftarrow 0$
trim L to size $p$
trim aux to size $q$
return $\langle \mathsf{L}, \mathsf{aux} \rangle$
```

**Algorithm 4.2:** Computation of a tunneled BWT encoding when only run-terminated length-maximal prefix intervals are considered.

Each length-maximal run-terminated prefix interval starting at a run with identifier $k$ which should not be tunneled must satisfy $\mathsf{RPE}[k] = \bot$. This means that we can select the prefix intervals to be tunneled by clearing entries in RPE.

The algorithm starts by marking entries in $M$ in the lines 3–20. More precisely, the algorithm marks entries in all columns of a prefix interval except for the start column. To avoid duplicate markings, the algorithm checks if prefix intervals overlap in lines 15–18. If this is the case, the algorithm uses RPE to jump over all remaining columns of the overlapping prefix interval. During this first loop, the algorithm also prepares the markings in aux (lines 8 and 19).

Moreover, the algorithm sets a pointer from the run coinciding with the last column of a prefix interval to the start of the prefix interval in line 20. The reason for this pointers is as follows: after lines 3–20, the algorithm has marked the end columns of all prefix intervals. This behavior is not desired. However, it could happen that entries in the start column of a prefix interval are marked because of overlappings. To this end, the lines 21 – 26 copy the markings from the start column to the end column of each interval. This ensures a desired marking even if prefix intervals overlap.

The remaining lines of the algorithm then remove all marked entries from L and shorten aux. The removal of marked entries uses only the variable $j$ and is performed in the lines 27, 32–35 and line 44. The shortening of aux is a bit more complicated: in the case that a new run in the shortened string L starts (line 36), a variable $h_1$ is set to one. When no new run starts and the variable $h_1$ is set to one, we know that a run with a height greater than one is processed for the first time (line 40). After copying the value of aux, setting $h_1 \leftarrow 0$ then ensures that no additional output is written to aux for the remaining characters of the current run.

Overall, Algorithm 4.2 requires $O(n)$ time to compute this specially encoded BWT, using Run-LF support and a subset of length-maximal run-terminated prefix intervals marked in the RPE-array.

For the sake of thoroughness, we also present a decoding algorithm transforming a tunneled BWT in Form of L and aux to a tunneled BWT using the components L, $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$. Similar to Algorithm 4.2, Algorithm 4.3 uses a couple of variables to indicate the current run with a height greater one. In the case that a new run starts (line 10), a variable $h_1$ is set to one. When the run has a height greater than one (line 18), the variable $k_{h>1}$ is incremented by $h_1$. Setting $h_1$ to zero afterwards ensures that $k_{h>1}$ points to the same run for all remaining characters of the run.

The difference between the tunneled BWT encoding in form of $\langle \mathsf{L}, \mathsf{aux} \rangle$ and the encoding in form of $\langle \mathsf{L}, D_{\mathsf{in}}, D_{\mathsf{out}} \rangle$ can be seen in Figure 4.4. Looking at the differences, the algorithm must handle two special cases. Let $k_{h>1}$ be the start of a tunnel which is not also the end of a tunnel, i.e. $\mathsf{aux}[k_{h>1}] = 1$. Then, the algorithm must trim the length of the run in L to one. Also, only one 1-bit in $D_{\mathsf{out}}$ corresponds to the start of the tunnel. This special case is handled by the lines 23–26 of Algorithm 4.3. The other special case occurs when $k_{h>1}$ is the end but not also the start of a prefix interval, i.e. $\mathsf{aux}[k_{h>1}] = 2$. In this case, only one 1-bit in $D_{\mathsf{in}}$ corresponds to the tunnel end. This special case is handled by the lines 27–29 of Algorithm 4.3.

The algorithm uses the special values of aux to facilitate case distinctions regarding bits to be set in $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ in lines 21–22. If the least significant bit in aux is set, $\mathsf{aux}[k_{h>1}] \in \{1, 3\}$ is implied, so a tunnel start is handled (line 21). If the most

```
Data: Tunneled BWT ⟨L, aux⟩ encoded by Algorithm 4.2.
Result: Tunneled BWT with the components L, $D_{out}$ and $D_{in}$.

1  $\tilde{n} \leftarrow$ size of L
2  let $D_{out}$ be a bit-vector of size $\tilde{n} + 1$
3  let $D_{in}$ be a bit-vector of size $\tilde{n} + 1$
4  $p \leftarrow 1$                      // current output position in L and $D_{out}$
5  $q \leftarrow 1$                      // current output position in $D_{in}$
6  $k_{h>1} \leftarrow 0$                // current run with height greater one
7  $c \leftarrow \$$                     // last character processed in L
8  $h_1 \leftarrow 1$                    // current run has height of one
9  for $i \leftarrow 1$ to $\tilde{n}$ do
10     if L[$i$] $\neq c$ then            // start of a new run
11         $c \leftarrow$ L[$i$]
12         $h_1 \leftarrow 1$
13         L[$p$] $\leftarrow$ L[$i$]
14         $D_{out}[p] \leftarrow 1$
15         $p \leftarrow p + 1$
16         $D_{in}[q] \leftarrow 1$
17         $q \leftarrow q + 1$
18     else
19         $k_{h>1} \leftarrow k_{h>1} + h_1$
20         $h_1 \leftarrow 0$
21         $d_{out} \leftarrow 1 -$ aux[$k_{h>1}$] div 2
22         $d_{in} \leftarrow 1 -$ aux[$k_{h>1}$] mod 2
23         if aux[$k_{h>1}$] $\neq 1$ then     // no tunnel start or both tunnel start and end
24             L[$p$] $\leftarrow c$
25             $D_{out}[p] \leftarrow d_{out}$
26             $p \leftarrow p + 1$
27         if aux[$k_{h>1}$] $\neq 2$ then     // no tunnel end or both tunnel start and end
28             $D_{in}[q] \leftarrow d_{in}$
29             $q \leftarrow q + 1$
30 trim L to size $p - 1$
31 trim $D_{out}$ to size $p$ and set $D_{out}[p] \leftarrow 1$
32 trim $D_{in}$ to size $q$ and set $D_{in}[q] \leftarrow 1$
33 return ⟨L, $D_{out}$, $D_{in}$⟩
```

**Algorithm 4.3:** Transformation of a tunneled BWT with components L and aux to a tunneled BWT with components L, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$.

significant bit in $\mathsf{aux}$ is set, $\mathsf{aux}[k_{h>1}] \geq 2$ holds, so a tunnel end is handled (line 22). The remaining transformation is straight forward. The algorithm requires linear time.

In summary, when only length-maximal run-terminated prefix intervals are tunneled, the additional components $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ can be reduced to a vector $\mathsf{aux}$ with size of $2 \cdot |\mathcal{R}_{h>1}|$ bits. The tunneling process itself can be performed in $O(n)$ time using Algorithm 4.2. To recover the original string from which the BWT was built, we can use Algorithm 4.3 to convert the tunneled BWT encoding to the "standardized" form. Afterwards, we can invert the tunneled BWT with the normal backward steps.

## 4.2 COST MODEL

The first "milestone" of an integration of tunneling into BWT-based data compressors has been made. We focused on length-maximal run-terminated prefix intervals, which allow overlapping prefix intervals. Also, these prefix intervals enable us to store the additional tunneling information in a vector using at most 2 bits per run of the underlying BWT.

We now address the final encoding of a tunneled BWT. In the case of a normal BWT, the `bzip2` compressor performs the following post BWT stages:

1. Transformation of the BWT $\mathsf{L}$ using move-to-front transform to $\mathsf{mtf}(\mathsf{L})$.
2. Application of run-length encoding to $\mathsf{mtf}(\mathsf{L})$ to get $\mathsf{rle}_1(\mathsf{mtf}(\mathsf{L}))$.
3. Source encoding of $\mathsf{rle}_1(\mathsf{mtf}(\mathsf{L}))$.

Details regarding the stages can be found in Section 2.2. A plethora of post BWT stages exist, see e.g. [Abe10]. However, almost all state-of-the-art BWT compressors include forms of run-length encoding and source encoding as stages in their post BWT process chain [Abe10]. Therefore, run-length encoding and source encoding can be seen as the key stages in making BWT compressors successful.

Coming back to the encoding of a tunneled BWT, the simplest way to integrate tunneling into a BWT data compressor is to apply all post BWT stages of the compressor to both components $\mathsf{L}$ and $\mathsf{aux}$, as shown in Figure 4.5. There are several reasons why this form of integration makes sense. First, the integration is easy and can be applied to almost all BWT compressors. Also, $\mathsf{L}$ and $\mathsf{aux}$ both come from the same source: the original BWT. As $\mathsf{L}$ is only a shortened version of the original BWT, compression of $\mathsf{L}$ should work well, and the same should hold true for $\mathsf{aux}$.

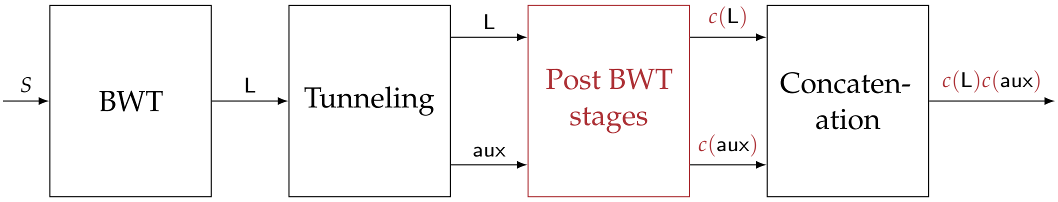


**Figure 4.5:** Integration of tunneling into a BWT-based compressor. Red items belong to the special Post BWT stages of the underlying compressor. Both L and aux are encoded using the same procedure.

This approach also offers a good worst-case compression: if we tunnel no prefix interval at all, L is identical to the original BWT. Therefore, the encoding of L has the same size as that of the original BWT. If no prefix interval is tunneled, aux contains only zeros. Almost all state-of-the-art BWT compressors include forms of run-length encoding. Thus, the overhead of the encoding of aux is about $O(\log |\mathcal{R}_{h>1}|)$ bits in this case. In summary, if no prefix interval is tunneled, about $O(\log |\mathcal{R}_{h>1}|)$ additional bits are required for tunneling integration. This number of additional bits is insignificant.

Another argument justifying that the post encoding stages of L and aux should be equal comes from the navigation in a tunneled BWT. In Section 3.2 we have seen that this navigation is possible using a wavelet tree of L and rank- and select-support for both $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$. Thinking about the operations of a wavelet tree, it is clear that a rank- and select-support of bit-vectors is nothing else than a wavelet tree of the bit-vector. To put this differently, if all components L, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ are encoded in wavelet trees, navigation in a tunneled BWT can be performed efficiently. This indicates that a similar encoding for all related tunneled BWT components enables a certain property of the tunneled BWT. In our case, we are interested in compression, so encoding both components L and aux with the same mechanism seems to be a good idea.

For the aforementioned reasons we decided to use the same compression for both L and aux. Additionally, we assume that the BWT post stages make use of some form of run-length encoding. This circumstance allows us to compute a "rating" of prefix intervals to be tunneled as follows. Each prefix interval removes a certain amount of characters from L when being tunneled. In a normal BWT, given a prefix interval $\langle w, [i, j] \rangle$ this number corresponds to $(w-1) \cdot (j-i)$. If we use run-length encoding, less characters are removed because each run $[i, j] \in \mathcal{R}$ is encoded using $1 + \lfloor \log_2(j-i+1) \rfloor$ characters. The rating of a prefix interval now consists of the number of characters it removes in a run-length encoded BWT. More precisely, let $\mathsf{L}$ be the normal BWT, and let $\tilde{\mathsf{L}}$ and $\mathsf{aux}$ be the tunneled BWT obtained by tunneling only the considered prefix interval. The rating of the prefix interval is then given by $|\mathsf{rle}(\mathsf{L})| - |\mathsf{rle}(\tilde{\mathsf{L}})|$.[1]

The rating of a prefix interval $\langle w, [i, j] \rangle$ can be computed by enumerating all columns of a prefix interval except for the start and end column. For each column, we determine the surrounding run $[i', j'] \in \mathcal{R}$ and subtract the length of the run-length encoded shortened run from the length of the run-length encoded normal run. In other words, when the prefix interval is tunneled, exactly $1 + \lfloor \log_2(j' - i' + 1) \rfloor - (1 + \lfloor \log_2(j' - i' + 1 - (j - i)) \rfloor)$ characters are removed from the run-length encoding of the run $[i', j']$. Thus, the rating consists of the sum of all removed characters. Note that we have to exclude the start- and end-column of a prefix interval because we use the special tunnel encoding where no characters from the start- and end-column are removed.

**Definition 4.9** (Length-maximal run-terminated prefix interval rating)**.** Let $\mathsf{L}$ be a BWT of a null-terminated string $S$ of length $n$. Let $\mathcal{R}$ be the runs in $\mathsf{L}$, and let $B_{\mathcal{R}}$ be the component of a Run-LF support. The rating array $\mathsf{RPTC}$ is an array of size $|\mathcal{R}|$ which is defined as follows:

- Let $k$ be an identifier of a run in $\mathsf{L}$ where a length-maximal run-terminated prefix interval $\langle w, [i, j] \rangle$ starts, i.e. $k = \mathsf{rank}_{B_{\mathcal{R}}}(1, i)$. Let $[i_0, j_0], \ldots, [i_{w-1}, j_{w-1}] \in \mathcal{R}$ be the runs where the prefix interval points through, i.e. $[\mathsf{LF}^x[i], \mathsf{LF}^x[j]] \subseteq [i_x, j_x]$ for all $x \in [0, w-1]$. Then,

$$\mathsf{RPTC}[k] := \sum_{x=1}^{w-2} \lfloor \log_2(j_x - i_x + 1) \rfloor - \lfloor \log_2(j_x - i_x + 1 - (j - i)) \rfloor.$$

- If $k$ identifies a run where no length-maximal run-terminated prefix interval starts, then $\mathsf{RPTC}[k] := 0$.

The computation of the rating array $\mathsf{RPTC}$ is shown in Algorithm 4.4. The algorithm uses the $\mathsf{RPE}$ - array from Definition 4.7 to indicate runs where a length-maximal run-terminated prefix interval starts. The correction term in line 14 is required because the algorithm also enumerates the end column of each prefix

[1] The computation of such a rating is not easy: a prefix interval could point through the same run multiple times. Instead, we use a simpler rating system by assuming that a prefix interval points through a run only once.

```
Data: Run-LF support, RPE-array of all length-maximal run-terminated prefix intervals.
Result: RPTC array, for each prefix interval storing the number of characters it removes when being tunneled in a
        run-length encoded BWT.

initialize an array RPTC of size $|\mathcal{R}|$ with zeros
for $r \leftarrow 1$ to $|\mathcal{R}|$ do
    if $\mathsf{RPE}[r] \neq \perp$ and $\mathsf{RPE}[r] \neq \mathsf{LF}_{\mathcal{R}}[r]$ then        // prefix intervals with width $\geq 2$
        $k \leftarrow r$
        $i \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k)$
        $j \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k+1) - 1$
        $x \leftarrow \mathsf{LF}_{\mathcal{R}}[k]$
        while $x \neq \mathsf{RPE}[k]$ do
            $k' \leftarrow \mathsf{rank}_{B_{\mathcal{R}}}(1, x)$
            $i' \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k')$
            $j' \leftarrow \mathsf{select}_{B_{\mathcal{R}}}(1, k'+1) - 1$
            $\mathsf{RPTC}[k] \leftarrow \mathsf{RPTC}[k] + \lfloor \log_2(j' - i' + 1) \rfloor - \lfloor \log_2(j' - i' + 1 - (j - i)) \rfloor$
            $x \leftarrow \mathsf{LF}_{\mathcal{R}}[k'] + (x - i')$
        $\mathsf{RPTC}[k] \leftarrow \mathsf{RPTC}[k] - \lfloor \log_2(j - i + 1) \rfloor$
return RPTC
```

**Algorithm 4.4:** Computation of the length-maximal run-terminated prefix interval rating array RPTC.

interval because RPE "points behind" this column. As the algorithm enumerates all columns of a prefix interval, the complexity of the algorithm is identical to the sum of widths of all length-maximal run-terminated prefix intervals. According to Lemma 4.4, this sum is bounded by $n$, so Algorithm 4.4 has a worst-case time complexity of $O(n)$. Note that the computation of the binary logarithm $\lfloor \log_2(x) \rfloor$ can be performed efficiently [War13]: most modern computer platforms include special processor instructions allowing to compute this logarithm efficiently.

When thinking about prefix intervals to be tunneled, Figure 4.6 shows that it is not a good idea to tunnel all length-maximal run-terminated prefix intervals. Despite the fact that this approach produces the shortest string L, the costs of encoding aux outweigh the benefits in many cases.

The reason for this is that the encoding size of aux increases when more prefix intervals are tunneled. If no prefix interval is tunneled, aux consists only of zeros, so aux contains only one run. When tunneling one prefix interval, aux contains two non-zero entries for the start and the end of the tunnel. This means that the number of runs in aux is increased from 1 to at least 3, or maximum 5. Therefore, tunneling of a prefix interval increases the number of runs in aux and thereby makes aux less compressible.

A possible strategy would be to tunnel e.g. only the best 10% of prefix intervals. Best prefix intervals here means that the considered prefix intervals have the highest rating of all possible prefix intervals. However, as Figure 4.6 shows, this approach

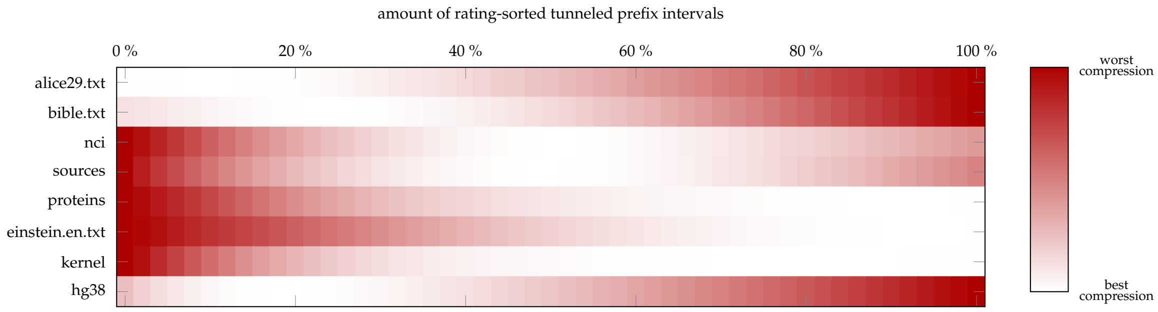


**Figure 4.6:** Compressibility of a tunneled BWT using block-sorting compression. The compressibility depends on the number of tunneled length-maximal run-terminated prefix intervals. The image shows only selected test files, an image with all test files is shown in Figure B.1. A similar image was already published in [BD19] © 2019 IEEE.

works in only a few cases. Also, the approach does not consider overlappings between prefix intervals, possibly reducing the number of characters that can be removed from L. Therefore, we need a more sophisticated approach for the successful integration of tunneling. We will present some strategies in Section 4.3. Beforehand, we need a cost model that allows for the comparison of tunneling benefits and costs.

### 4.2.1 *Definition*

The benefits and costs of tunneling strongly depend on the post BWT stages in the compression chain. A possibility would be to develop a special cost model for each possible compression chain. The problem with this approach is that tunneling integration must be adapted for each enhanced BWT compressor.

Instead, we will present the uniform cost model as presented in [Bai18] and [BD19]. The cost model is based on a run-length encoded and source encoded BWT. As discussed in the beginning of this section, all state-of-the-art BWT compressors make use of different forms of these BWT post stages.

Another difficulty of developing a cost model is that it must work a priori: decisions about tunneling have to be made on the basis of the cost model. Applying the cost model after prefix intervals have been tunneled is similar to a trial-and-error method that forces us to revert the tunneling in bad cases. This is impractical because the time-consuming tunneling operations have to be performed multiple times. However, an a priori cost model cannot reflect all effects of tunneling to a

BWT. For example, prefix intervals can overlap, reducing the number of removed characters. Also, the model cannot foresee the run-length encodings of all runs, making estimations about the character frequencies difficult.

A circumstance that simplifies the cost model is that the cost model must not precisely foresee how many bits the tunneling of a prefix interval saves and costs. In other words, there is no need to precisely determine the number of saved bits before tunneling. Instead, it is important that the cost model reflects the impact of tunneling on the compression rate. In this case, a good tunnel choice in the model will produce a good compression with the underlying compressor.

We thus introduce a couple of relaxations that make the definition of a cost model easier. Our first assumption is that the two run-characters $\mathtt{0}_{\mathsf{rle}}$ and $\mathtt{1}_{\mathsf{rle}}$ used to encode runs in run-length encoding occur with the same frequency. This makes the estimation about tunneling benefits possible. Let $\mathsf{L}$ be a BWT of size $n$, where

- $n_{\mathsf{rle}} := |\mathsf{rle}(\mathsf{L})|$ is the length of the run-length encoding.
- $rc := n_{\mathsf{rle}} - |\mathcal{R}|$ is the number of "run-characters" in $\mathsf{rle}(\mathsf{L})$.
- $f_{\mathsf{rle}}(c) := |\{\ [i,j] \in \mathcal{R} \ \mid\ S[i] = c\ \}$ is the frequency of the characters $c \in \Sigma$ in $\mathsf{rle}(\mathsf{L})$.

The size of a run-length encoded and source-encoded BWT can be approximated by

$$n_{\mathsf{rle}} \cdot H(\mathsf{rle}(\mathsf{L})) \approx \underbrace{\log_2(n_{\mathcal{R}}!)}_{\text{full information}} \underbrace{- \sum_{c\in\Sigma} \log_2(f_{\mathsf{rle}}(c)!)}_{\text{"non-run-characters"}} - \underbrace{2 \cdot \log_2((rc/2)!)}_{\text{run-characters}}.$$

Now, let $\langle \tilde{\mathsf{L}}, \mathsf{aux} \rangle$ be a tunneled BWT obtained by tunneling $\mathsf{L}$. The effect of tunneling $\mathsf{L}$ is that the runs in $\mathsf{L}$ are shortened. Therefore, the frequencies of characters at the start of a run do not change, i.e. $f_{\mathsf{rle}}$ is identical for both $\mathsf{L}$ and $\tilde{\mathsf{L}}$. Let

- $t$ be the number of length-maximal run-terminated prefix intervals that were tunneled.
- $tc := |\mathsf{rle}(\mathsf{L})| - |\mathsf{rle}(\tilde{\mathsf{L}})|$ be the number of characters removed from $\mathsf{L}$.

By using these variables, the benefit of tunneling can be approximated with

$$
\begin{align}
& n_{\mathsf{rle}} \cdot H(\mathsf{rle}(\mathsf{L})) - (n_{\mathsf{rle}} - tc) \cdot H(\mathsf{rle}(\tilde{\mathsf{L}})) \nonumber \\
& \quad \approx \log_2(n_{\mathsf{rle}}!) - \log_2((n_{\mathsf{rle}} - tc)!) + \sum_{c \in \Sigma} \log_2(f_{\mathsf{rle}}(c)!) - \sum_{c \in \Sigma} \log_2(f_{\mathsf{rle}}(c)!) \nonumber \\
& \qquad - 2 \cdot \log_2((rc/2)!) + 2 \cdot \log_2(((rc - tc)/2)!) \nonumber \\
& \quad = \log_2\left(\frac{n_{\mathsf{rle}}!}{(n_{\mathsf{rle}} - tc)!}\right) - 2 \cdot \log_2\left(\frac{(rc/2)!}{((rc - tc)/2)!}\right) \tag{4.2} \\
& \quad \approx \log_2\left((n_{\mathsf{rle}})^{tc}\right) - 2 \cdot \log_2(\Big((rc/2)^{tc/2}\Big) \tag{4.3} \\
& \quad = tc \cdot \log_2(n_{\mathsf{rle}}) - tc \cdot \log_2(rc) + tc \nonumber \\
& \quad = tc \cdot \left(1 + \log_2\left(n_{\mathsf{rle}}/rc\right)\right). \nonumber
\end{align}
$$

The transition from line (4.2) to line (4.3) is possible because we assume that the number of tunneled characters $tc$ is much smaller than the amount of run-characters $rc$. Therefore, we assume that $tc \ll rc$ and $tc \ll n_{\mathsf{rle}}$ hold. Dividing the faculty $n_{\mathsf{rle}}!$ by $(n_{\mathsf{rle}} - tc)!$ leaves only $tc$ factors over which all are very close to $n_{\mathsf{rle}}$. The same holds true for the content of the second logarithm in line (4.2).

Estimations about the costs of encoding the additional component aux are more difficult. We do not know where the tunnel start and end markings will be placed before starting the tunneling process. Our first assumption is that the markings are evenly distributed over aux. Furthermore, we assume that tunnel starts and ends are sparse, so the majority of entries in aux indicate "normal" runs. The reason is that we assume that only a couple of runs belong to the start or end of a length-maximal run-terminated prefix interval. Also, we assume that the number of "good candidates" to be tunneled is even smaller. As a consequence, we assume that no length-maximal run-terminated prefix interval of length one is tunneled, because it brings no benefit at all.

Assumptions about the sparseness of tunnel start and end markings in aux lead to the implication that only runs of zeros in aux exist, see also Figure 4.7. Those runs are interrupted by exactly $2 \cdot t$ markings for tunnel starts and ends, so we assume that aux contains $2 \cdot t + 1$ runs of zero-characters. Because we do not know where the tunnel markings in aux will be placed, we assume that the entries are distributed evenly, see Figure 4.7. This means that the average length of a zero-run is given by $\frac{r_{h>1} - 2 \cdot t}{2 \cdot t + 1}$ characters, where $r_{h>1} := |\mathcal{R}_{h>1}|$ is the number of runs with height greater than one. Run-length encoding will transform such a run to a new sequence of length

$$
\begin{array}{ll}
\begin{array}{l}
\quad\mathsf{aux} \\
\left.\begin{array}{c}0\\0\\\vdots\\0\end{array}\right\}\approx \frac{r_{h>1}-2\cdot t}{2\cdot t+1} \\
\quad 2 \\
\left.\begin{array}{c}0\\0\\\vdots\\0\end{array}\right\}\approx \frac{r_{h>1}-2\cdot t}{2\cdot t+1} \\
\quad 1 \\
\quad 0 \\
\quad 0 \\
\quad \vdots
\end{array}
&
\begin{array}{l}
\quad\mathsf{rle}(\mathsf{aux}) \\
\left.\begin{array}{c}0\\0_{\mathsf{rle}}\\\vdots\\1_{\mathsf{rle}}\end{array}\right\}\approx \log_2\left(\frac{r_{h>1}-2\cdot t}{2\cdot t+1}\right) \\
\quad 2 \\
\left.\begin{array}{c}0\\1_{\mathsf{rle}}\\\vdots\\0_{\mathsf{rle}}\end{array}\right\}\approx \log_2\left(\frac{r_{h>1}-2\cdot t}{2\cdot t+1}\right) \\
\quad 1 \\
\quad 0 \\
\quad 1_{\mathsf{rle}} \\
\quad \vdots
\end{array}
\end{array}
$$

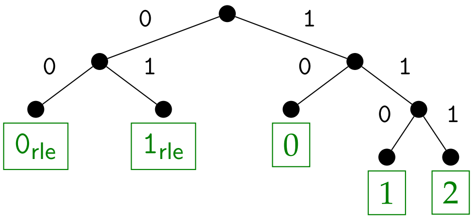
**Figure 4.7:** Expected structure of the aux-component (left-hand side) and expected run-length encoding of aux (right-hand side). The $2 \cdot t$ tunnel start and end markings are evenly distributed over aux. The average length of a zero-run in aux is $\frac{r_{h>1}-2\cdot t}{2\cdot t+1}$.

$\log_2(\frac{r_{h>1}-2\cdot t}{2\cdot t+1})$. This allows us to estimate the length of aux when using run-length encoding:

$$|\mathsf{rle}(\mathsf{aux})| \approx \underbrace{2\cdot t}_{\text{tunnel markings}} + \underbrace{(2\cdot t+1)\cdot\log_2\left(\frac{r_{h>1}-2\cdot t}{2\cdot t+1}\right)}_{\text{run-length encoded zero-runs}}.$$

To approximate the size of the run-length encoded and source encoded aux-component, we need information about the entropy $H(\mathsf{rle}(\mathsf{aux}))$. This is even harder to estimate because the character frequencies in rle(aux) depend on the length of rle(aux) which itself depends on the number of tunnels. We make the following additional assumptions:

- The fewest characters in aux are tunnel starts and ends, each occurring exactly $t$ times.
- rle(aux) contains about $2 \cdot t + 1$ zeros, which indicate the start of a zero-run.
- The most frequent characters are run-characters, i.e. characters that belong to the run-length encoding of a zero-run. About $(2\cdot t+1)\cdot\left(\log_2\left(\frac{r_{h>1}-2\cdot t}{2\cdot t+1}\right)-1\right)$ of these characters will occur in aux.



| category | character | code |
|---|---|---|
| run-length | $0_{\mathsf{rle}}$ | 00 |
| encoding | $1_{\mathsf{rle}}$ | 01 |
| normal run | 0 | 10 |
| tunnel | 1 | 110 |
| markings | 2 | 111 |

**Figure 4.8:** Huffman code for the characters in a run-length encoded aux-component. The least frequent characters are tunnel markings, the most frequent characters are run-length encoding characters.

By using the above numbers, we could compute an approximation of $H(\mathsf{rle}(\mathsf{aux}))$. However, we prefer a much simpler method here: Huffman coding [Huf52]. As mentioned in Section 2.2.1, Huffman coding is optimal. Therefore, $|\mathsf{rle}(\mathsf{aux})| \cdot H(\mathsf{rle}(\mathsf{aux}))$ is approximately the same as the size of the Huffman encoding of $\mathsf{rle}(\mathsf{aux})$. Regarding the above assumptions about character frequencies in $\mathsf{rle}(\mathsf{aux})$, a possible Huffman code for $\mathsf{rle}(\mathsf{aux})$ is shown in Figure 4.8.

With this code, an approximation of the size of the run-length encoded and source-encoded aux-component is given as follows:

$$\begin{aligned}
&|\mathsf{rle}(\mathsf{aux})| \cdot H(\mathsf{rle}(\mathsf{aux})) \\
&\approx \underbrace{2 \cdot t \cdot 3}_{\text{tunnel markings}} + \underbrace{(2 \cdot t + 1) \cdot 2}_{\text{zeros}} + \underbrace{(2 \cdot t + 1) \cdot \left( \log_2 \left( \frac{r_{h>1} - 2 \cdot t}{2 \cdot t + 1} \right) - 1 \right) \cdot 2}_{\text{run-length encoding characters}} \\
&= 10 \cdot t + 2 + (4 \cdot t + 2) \cdot \log_2 \left( \frac{r_{h>1} - 2 \cdot t}{2 \cdot t + 1} \right) - 4 \cdot t - 2 \\
&= 6 \cdot t + 4 \cdot (t + 0.5) \cdot \log_2 \left( \frac{r_{h>1} + 1}{2 \cdot t + 1} - 1 \right) \\
&= 6 \cdot (t + 0.5) - 3 + 4 \cdot (t + 0.5) \cdot \log_2 \left( \frac{r_{h>1} + 1}{2 \cdot t + 1} - 1 \right) \\
&= (t + 0.5) \cdot (6 + 4 \cdot \log_2 \left( \frac{r_{h>1} + 1}{2 \cdot t + 1} - 1 \right)) - O(1).
\end{aligned}$$

We sum up these results in the following definition.

**Definition 4.10** (Tunneling cost model)**.** Let $\mathsf{L}$ be a BWT of a null-terminated string $S$. Let $r$ be the number of runs in $\mathsf{L}$, and let $r_{h>1}$ be the number of runs in $\mathsf{L}$ with height greater one. Let $n_{\mathsf{rle}}$ be the length of the run-length encoding of $\mathsf{L}$, i.e. $n_{\mathsf{rle}} := |\mathsf{rle}(\mathsf{L})|$, and define $rc := n_{\mathsf{rle}} - r$.

The benefit of removing $tc$ run-characters from $\mathsf{L}$ is determined by the function

$$\mathsf{benefit}(tc) := tc \cdot \left(1 + \log_2\left(\frac{n_{\mathsf{rle}}}{rc}\right)\right).$$

The cost of tunneling $t$ length-maximal run-terminated prefix intervals is determined by the function

$$\mathsf{cost}(t) := (t + 0.5) \cdot \left(6 + 4 \cdot \log_2\left(\frac{r_{h>1} + 1}{2 \cdot t + 1} - 1\right)\right).$$

Definition 4.10 constitutes our cost model. As a reminder, the cost model is independent of the used BWT post compression stages. Values for $tc$ and $t$ can be determined by the RPTC array. However, a lot of unproven assumptions were made to set up the cost model. Thus, the next section will validate the cost model and show that the model is appropriate for tunneling decisions.

### 4.2.2 *Validation*

To validate the cost model, we set up the following experiment: we sorted prefix intervals to be tunneled according to their rating. Furthermore, we ignored prefix intervals with a zero rating. Then, we tunneled the highest rated $p$ percent of prefix intervals, where $p \in \{0, 2, 4, \ldots, 100\}$. After the tunneling process, we measured the amount of characters removed from the run-length encoding of the BWT. This corresponds to the variable $tc$ in the cost model. Also, we keep track of the number of tunnels $t$.

Then, for each file and its corresponding value of $p$, we computed a number between 0 and 1 which indicates the relative compressibility. A 0 means that the compressibility was best under all other $p$-values of that file, while a 1 indicates the worst compressibility. These numbers are similar to the color values of files in Figure 4.6.

The relative compressibility numbers of the cost model were computed using the difference between the theoretical benefit and cost for the file when tunneling $t$ prefix intervals that remove $tc$ characters. In the case of real compressors, we took the best and worst compression result for each file and scaled the other compression results of that file accordingly.

We then compared the discrepancy between cost model and compressor by subtracting the relative compressibility of the compressor from the relative compressibil-

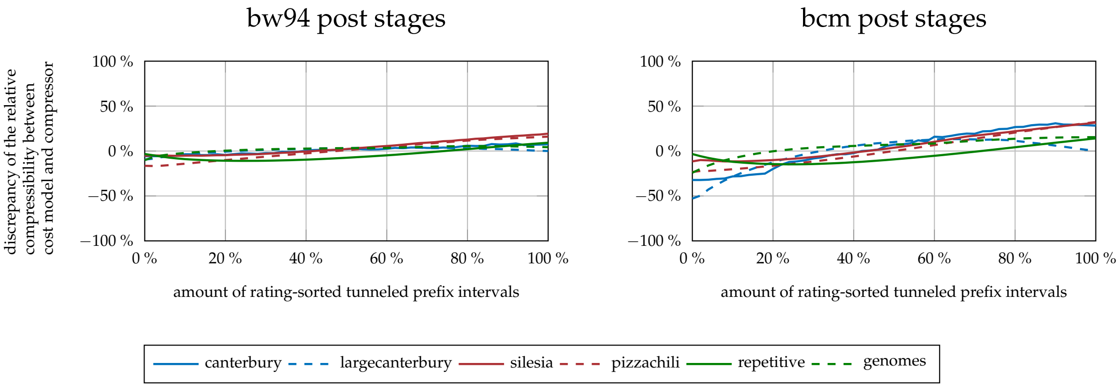


**Figure 4.9:** Discrepancy between estimated relative compressibility and real relative compressibility in conjunction with tunneling. The discrepancies are average values for all files of a text corpus. See Chapter A for more information about the test data. The cost model was tested on the two BWT post stages bw94 and bcm, see Section B.1. To limit the computation time, we truncated each test file after the first gigabyte of data.

ity in the cost model. We used two different bwt compressors: bw94 is the scheme as presented by Burrows and Wheeler in 1994 using move-to-front, run-length encoding and source encoding. Additionally, we used the post stages of another very good BWT compressor called bcm. More information about both post stages can be found in Section B.1. More details about the used test data can be found in Section B.1.

The idea behind the experiment is the following: in the case that the discrepancy between cost model and compressor is zero for all values of $p$, the cost model is ideal. It might not be the case that the cost model and compressor achieve the same benefits in bits in this case. However, the cost model then precisely describes the effects of tunneling onto the relative compression rate. This is exactly the desired behavior of the cost model.

The average discrepancies between cost model and compressor are shown in Figure 4.9. One can see that the discrepancy is not constantly zero, but very close to zero for $p \in [20, 80]$. Also it can be seen that the cost model fits better to the bw94 compressor than to the bcm compressor. In general, the discrepancy plots in Figure 4.9 describe monotone increasing functions.

This means that the relative compressibility was under-estimated by the cost model for $p \in [0, 20)$. The relative compressibility in the compressors was worse than the relative compressibility in the cost model. This results in negative values of discrepancy. The opposite holds true for $p \in (80, 100]$.

Regarding the cost model, it is very likely that the discrepancy is owed to the estimation of the component aux in the cost model. In the cost model, the encoding of aux is a worst-case encoding: we assumed that every run in aux has the same length, which produces a worst-case run-length encoding of aux. Moreover, we assumed that the global structure transformations used in the post stages have no effect on the encoding of aux. As indicated by Figure 4.9, these assumptions hold true for $p \in [20, 80]$. When the number of tunnels is smaller, aux tends to be better compressible, causing negative effects in the discrepancy. For $p \in (80, 100]$, the hypothesis of the character frequencies in aux no longer holds true, causing the source encoding of $\mathsf{rle}(\mathsf{aux})$ to grow. This explains the discrepancy for small and big values of $p$.

Nonetheless, the cost model has a good fit. Except for some outliers, the discrepancy ranges between $-13\%$ and 13% in the bw94 compressor, and between $-25\%$ and 25% for bcm. Figure 4.9 shows almost no kinks in the plots, so we can expect a steady and precise benefit-cost estimation. For all test files, the optimal value of $p$ lies between 0 and 80, or very close to $p = 80$, see Figure 4.6. For $p \in [0, 80]$, the cost model could be designated as conservative, because it typically under-estimates the compressibility gain of tunneling. However, the more conservative estimation ensures a good worst-case compression and allows realistic decisions for tunnel planning.

## 4.3 TUNNEL PLANNING STRATEGIES

The cost model is the second "milestone" of an integration of tunneling into BWT-based data compressors. We have seen that it is not worth it to tunnel all length-maximal run-terminated prefix intervals in most cases because each tunnel produces a certain cost. The final step that has to be made is to set up a tunnel strategy that considers both the benefits and costs of tunneling.

In the Chapter 3 it was mentioned that tunnel planning can be difficult. The problem we face here is something different: we do not want to cover all prefix intervals. Figure 4.6 clearly has shown that it is not always worth it to tunnel everything. Instead, we want to maximize the benefit of tunneling using the cost model from the last section.

In the last section we introduced the rating array RPTC, see Definition 4.9. The higher the rating of a prefix interval is, the more benefit can be achieved by tunneling the prefix interval. The cost of a tunnel is independent from the size of the underlying prefix interval, so choosing the candidates with the highest rating is a good idea in general. Indeed, all approaches presented in this section will make use of the rating array to find a good choice of prefix intervals to be tunneled. The problem lies elsewhere.

Length-maximal run-terminated prefix intervals can overlap. This means that the decision of tunneling one prefix interval can have negative side effects on the rating of other prefix intervals. More precisely, in the case of an overlapping, characters in the intersection of both prefix intervals cannot be removed twice. Therefore, the combined benefit of tunneling two prefix intervals can be smaller than the sum of benefits when tunneling the prefix intervals separately. An example of this situation is shown in Figure 4.10.

The situation is even more complicated because the possibility of positive side effects exists. In the case that two non-overlapping prefix intervals with identical height $h$ point through a run $k$ with height $2 \cdot h$, the situation is different. Tunneling only one prefix interval has the consequence that the run-length encoding of the run $k$ is shortened from $\log_2(2 \cdot h) = \log_2(h) + 1$ characters to $\log_2(2 \cdot h - (h-1)) \approx \log_2(h)$ characters. Tunneling only one prefix interval thus brings the benefit of removing one run-length character. When both prefix intervals are tunneled, the run-length encoding of $k$ is reduced to only $\log_2(2 \cdot h - 2 \cdot (h-1)) = \log_2(2) = 1$ character. This means that tunneling one prefix interval increases the rating of the other prefix interval. An example of this situation is shown in Figure 4.10.

In summary, the decision of tunneling a prefix interval can have both positive and negative side effects on the ratings of other prefix intervals. This makes the development of a strategy difficult. In this section, we will present basic and relatively easy tunnel planning strategies. The reason for this is that the strategies work well in practice, are easy to implement and are fast in terms of theoretical and practical run-time. We will include some aspects on the optimality of the strategies.

However, it could be possible that more sophisticated strategies can be developed, improving the compression rate even more. We strongly suppose that the tunnel planning problem with the cost model and side-effects is NP-complete. We clearly state this as open problems, but for now want to focus on simple strategies that are "good enough" and "easy enough" to show the potential of tunneling.

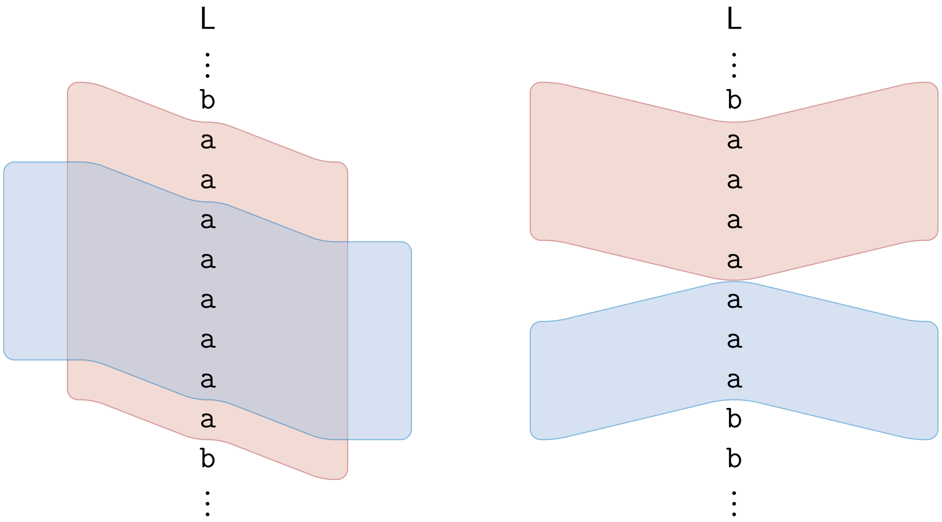


**Figure 4.10:** Illustration of side effects of tunneling on the rating of prefix intervals. Length-maximal run-terminated prefix intervals are indicated by the colored boxes. The left-hand side shows an example of a negative side-effect. Before the blue prefix interval is tunneled, the red prefix interval reduces the run-length encoding of the illustrated run from $1 + \log_2(8) = 4$ characters to just one character. When the blue prefix interval is tunneled, 4 characters are removed from the run. If the red prefix interval is tunneled afterwards, it reduces the run-length encoding from the run from $1 + \log_2(4) = 3$ characters to only one character. The right-hand side shows an example of a positive side effect. Before the blue prefix interval is tunneled, tunneling the red prefix interval "reduces" the run-length encoding from $1 + \lfloor \log_2(7) \rfloor = 3$ to $1 + \lfloor \log_2(4) \rfloor = 3$ characters. After the blue prefix interval is tunneled, the red prefix interval reduces the run-length encoding of the run from $1 + \lfloor \log_2(5) \rfloor = 3$ to $1 + \log_2(2) = 2$ characters.

### 4.3.1 *Hirsch strategy*

The first strategy we will present is a pragmatic approach: only prefix intervals that are profitable should be tunneled. This approach was invented by the author of this thesis for the use in a student project [Rät19], and is an extension of another strategy also presented by the author of this thesis [BD19]. A prefix interval is worth being tunneled if the expected benefit is bigger than the expected tunnel costs.

Let $tc$ be the number of characters that are removed by tunneling one certain prefix interval. Note that $tc$ is nothing more than an entry in the rating array RPTC, see Definition 4.9. Using our cost model from Definition 4.10, the benefit of tunneling this prefix interval then is $\mathsf{benefit}(tc)$. The cost of tunneling a prefix interval depends on the number $t$ of overall tunneled prefix intervals. The average cost per tunnel is given by $\mathsf{cost}_{\mathsf{avg}}(t) := \frac{\mathsf{cost}(t)}{t}$.

Thus it is profitable to tunnel the prefix interval if $\mathsf{benefit}(tc) \geq \mathsf{cost}_{\mathsf{avg}}(t)$ holds. Using the cost model and the definition $\varepsilon(t) := \frac{0.5}{t}$, the following inequalities are induced:

$$
\begin{aligned}
& & \mathsf{benefit}(tc) &\geq \mathsf{cost}_{\mathsf{avg}}(t) \\
\Leftrightarrow & & tc \cdot \left(1 + \log_2\left(\frac{n_{\mathsf{rle}}}{rc}\right)\right) &\geq \frac{t+0.5}{t} \cdot \left(6 + 4 \cdot \log_2\left(\frac{r_{h>1}+1}{2 \cdot t + 1} - 1\right)\right) \\
\Leftrightarrow & & tc \cdot \log_2\left(\frac{2 \cdot n_{\mathsf{rle}}}{rc}\right) &\geq (1 + \varepsilon(t)) \cdot \left(6 + 4 \cdot \log_2\left(\frac{r_{h>1}+1}{2 \cdot t + 1} - 1\right)\right) \\
\Leftrightarrow & & \frac{tc \cdot \log_2\left(\frac{2 \cdot n_{\mathsf{rle}}}{rc}\right)}{4 \cdot (1 + \varepsilon(t))} - 1.5 &\geq \log_2\left(\frac{r_{h>1}+1}{2 \cdot t + 1} - 1\right) \\
\Leftrightarrow & & 2^{\frac{tc}{4 \cdot (1+\varepsilon(t))} \cdot \log_2\left(\frac{2 \cdot n_{\mathsf{rle}}}{rc}\right) - 1.5} &\geq \frac{r_{h>1}+1}{2 \cdot t + 1} - 1 \\
\Leftrightarrow & & 2 \cdot t + 1 &\geq \frac{r_{h>1}+1}{2^{\frac{tc}{4 \cdot (1+\varepsilon(t))} \cdot \log_2\left(\frac{2 \cdot n_{\mathsf{rle}}}{rc}\right) - 1.5} + 1} \\
\Leftrightarrow & & t &\geq \frac{r_{h>1}+1}{2^{\frac{tc}{4 \cdot (1+\varepsilon(t))} \cdot \log_2\left(\frac{2 \cdot n_{\mathsf{rle}}}{rc}\right) - 0.5} + 2} - 0.5 \qquad (4.4)
\end{aligned}
$$

The interpretation of the last inequality in line (4.4) is as follows. The right-hand side contains only variables that are known before the considered prefix interval is tunneled. We ignore the $\varepsilon(t)$ term as it is very small and disappears for large values of $t$, i.e. $\varepsilon(t) = \frac{0.5}{t} \to 0$ for $t \to \infty$. The left-hand side describes the number of overall tunnels. Therefore, the benefits of tunneling the prefix interval starting at the run with identifier $k$ exceed the cost for the tunnel if at least

$$\mathsf{MT}[k] := \frac{r_{h>1}+1}{2^{\frac{\mathsf{RPTC}[k]}{4} \cdot \log_2\left(\frac{2 \cdot n_{\mathsf{rle}}}{rc}\right) - 0.5} + 2} - 0.5$$

prefix intervals in the whole BWT are tunneled. This makes sense because the average cost per tunnel decreases the more prefix intervals are tunneled, i.e. the function

$$\mathsf{cost}_{\mathsf{avg}}(t) = \frac{\mathsf{cost}(t)}{t} = \frac{t+0.5}{t} \cdot \left(6 + 4 \cdot \log_2\left(\frac{r_{h>1}+1}{2 \cdot t + 1} - 1\right)\right)$$

is a monotone decreasing function for $t \in [1, \frac{r_{h>1}}{2}]$. Thus, the average costs can be minimized by maximizing the number $t$ of tunnels. On the other hand, each tunnel starting at the run with identifier $k$ should be profitable. By the above inequalities, this is true if $\mathsf{MT}[k] \leq t$ holds. Bringing both conditions together, we are looking for a maximal set $T^*$ of run identifiers where $\mathsf{MT}[k] \leq |T^*|$ holds for all $k \in T^*$. The set can be determined as follows:

1. Compute a maximal number $t^*$ such that $|\{\, k \in [1, |\mathcal{R}|] \mid \mathsf{MT}[k] \leq t^* \,\}| \geq t^*$.
2. Find $T^*$ by filtering all values $\mathsf{MT}[k]$ satisfying $\mathsf{MT}[k] \leq t^*$.

The maximal number $t^*$ is similar to another well-known concept: the Hirsch index [Hir05]. The Hirsch index is a rating system for the productivity and work impact of a researcher. Let $Q$ be an array that stores the number of citations of a paper for all published papers of a researcher. The Hirsch index is the maximum number $h^*$ such that at least $h^*$ entries in $Q$ are greater than or equal to $h^*$. The difference between the Hirsch index and the number $t^*$ is subtle. The Hirsch index is looking for entries that are greater than or equal to $h^*$, while we are looking for entries that are smaller than or equal to $t^*$. Regarding the similarities and ignoring the subtle difference, this gives the tunnel planning strategy its name: the Hirsch strategy.

The problem we are facing here is how the maximum number $t^*$ can be computed. First of all, we can ignore entries where $\mathsf{MT}[k] \geq \frac{r_{h>1}+1}{\frac{1}{\sqrt{2}}+2}$ hold. If a prefix interval brings no benefit at all, $\mathsf{RPTC}[k] = 0$ holds. By plugging this value into the above definition of MT we obtain

$$\mathsf{MT}[k] = \frac{r_{h>1}+1}{2^{0-0.5}+2} - 0.5 < \frac{r_{h>1}+1}{\frac{1}{\sqrt{2}}+2}.$$

Therefore, it is never worth it to tunnel more than $\frac{r_{h>1}+1}{\frac{1}{\sqrt{2}}+2}$ prefix intervals. The idea of an efficient computation of $t^*$ is to set up a counting array $\mathsf{C}_{\mathsf{MT}}$ of size $\frac{r_{h>1}+1}{\frac{1}{\sqrt{2}}+2}$ which counts the number of entries in MT which are smaller than a certain amount. More precisely, $\mathsf{C}_{\mathsf{MT}}$ is defined as follows:

$$\mathsf{C}_{\mathsf{MT}}[i] := |\{\, k \in [1, |\mathcal{R}|] \mid \mathsf{MT}[k] \leq i-1\}|.$$

The maximal number $t^*$ then can be found at the largest index where $\mathsf{C}_{\mathsf{MT}}[t^*+1] \geq t^*$ is satisfied. This can be checked with a single left-to-right scan of $\mathsf{C}_{\mathsf{MT}}$.

Algorithm 4.5 gathers all the ideas of this strategy. Lines 1–4 compute the MT-array. The expression $\log_2\left(\frac{2\cdot n_{\mathsf{rle}}}{rc}\right)$ can be precomputed. The final expression then

```
Data: RPTC array, RPE-array, BWT statistics $n_{\mathrm{rle}}$, $rc$, $r := |\mathcal{R}|$ and $r_{h>1}$.
Result: Modified RPE-array where all prefix intervals which should not be tunneled contain a $\perp$.
                                                        // compute MT array
1  initialize an integer array MT of size $r$
2  for $k \leftarrow 1$ to $r$ do
3      $p \leftarrow \frac{\mathsf{RPTC}[k] \cdot \log_2\left(\frac{2 \cdot n_{\mathrm{rle}}}{rc}\right) - 2}{4}$
4      $\mathsf{MT}[k] \leftarrow (r_{h>1} + 1)/(2^p + 2) - 0.5$
                                        // compute counts of distinct MT values
5  initialize an array $\mathsf{C}_{\mathsf{MT}}$ of size $(r_{h>1} + 1)/(\frac{1}{\sqrt{2}} + 2)$ with zeros
6  for $k \leftarrow 1$ to $r$ do
7      if $\mathsf{MT}[k] < (r_{h>1} + 1)/(\frac{1}{\sqrt{2}} + 2)$ then
8          $\mathsf{C}_{\mathsf{MT}}[\mathsf{MT}[k] + 1] \leftarrow \mathsf{C}_{\mathsf{MT}}[\mathsf{MT}[k] + 1] + 1$
                              // compute cumulative counts and find maximal $t^*$
9  $t^* \leftarrow 0$
10 for $t \leftarrow 1$ to $(r_{h>1} + 1)/(\frac{1}{\sqrt{2}} + 2) - 1$ do
11     $\mathsf{C}_{\mathsf{MT}}[t + 1] \leftarrow \mathsf{C}_{\mathsf{MT}}[t + 1] + \mathsf{C}_{\mathsf{MT}}[t]$
12     if $\mathsf{C}_{\mathsf{MT}}[t + 1] \geq t$ then
13         $t^* \leftarrow t$
                                    // choose prefix intervals to be tunneled
14 for $k \leftarrow 1$ to $r$ do
15     if $\mathsf{MT}[k] > t^*$ then
16         $\mathsf{RPE}[k] \leftarrow \perp$
17 return RPE
```

**Algorithm 4.5:** Hirsch tunnel planning strategy. The result of this algorithm can be used to tunnel a BWT, see Algorithm 4.2.

can be computed by computing the exponent $p$ (line 3), rounding to the next integer and using bit-shifting in the left direction to compute the power with base two.

Lines 5–8 compute the counts of distinct MT-values that are less than $(r_{h>1} + 1)/(\frac{1}{\sqrt{2}} + 2)$. The entry $\mathsf{C}_{\mathsf{MT}}[\mathsf{MT}[k] + 1]$ is accessed to ensure that zero-values of MT are handled correctly. Lines 9–13 then compute the final $\mathsf{C}_{\mathsf{MT}}$ array and thereby search for the largest index $t$ satisfying $\mathsf{C}_{\mathsf{MT}}[t + 1] \geq t$. As indicated above, this is the desired maximal number $t^*$.

Finally, in the lines 14–16, the algorithm clears entries in RPE where it is not worth to tunnel the prefix interval. This result can be used to compute a tunneled BWT, see Algorithm 4.2. The algorithm has the worst-case run-time of $O(|\mathcal{R}|)$.

Let us shortly revisit the Hirsch strategy. The idea of the strategy is to choose only prefix intervals which are worth being tunneled. As the average costs decrease when more prefix intervals are tunneled, the strategy searches for a maximal subset of prefix intervals such that the benefit of tunneling each prefix interval is bigger than the average costs. The strategy will thus prefer big prefix intervals, i.e. the final choice consists of the best $p$ percentage of all prefix intervals regarding the rating array RPTC.

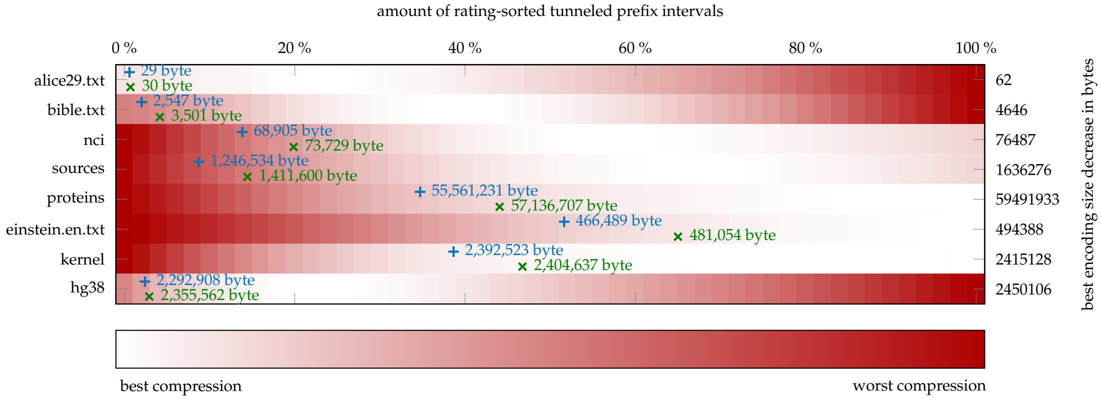


**Figure 4.11:** Optimality of the hirsch and greedy strategy in the bcm compressor. The amount of tunneled prefix intervals using the hirsch strategy is indicated with blue pluses. The amount of tunneled prefix intervals using the greedy strategy is indicated with green crosses. The best encoding size decrease compared to an compression without tunneling is shown on the right. The encoding size decrease of the hirsch and greedy strategy is shown right beside the pluses/crosses. An image including all test files can be found in Figure B.2. An image measuring the optimality with block-sorting compression can be found in Figure B.1. A similar image was already published in [BD19] © 2019 IEEE.

The optimality of the strategy is shown in Figure 4.11. One can see that the strategy chooses an amount of prefix intervals with good compression. Also, the strategy is easily implementable. It is noticeable that the strategy chooses less than the optimal number of prefix intervals to be tunneled in almost all cases. This is owed to the conservative cost model, see Section 4.2.2.

The strategy however has some downsides. First, the strategy does not consider overlappings or positive side effects at all. Second, the strategy does not consider the advantages of a “pawn sacrifice”. This means that it can be beneficial to tunnel a prefix interval although its benefit does not outweigh its cost: in such a case, the average tunnel cost decreases. If the negative effect of the additional tunnel is smaller than the sum of the reduced costs of all other chosen prefix intervals, a “pawn sacrifice” would make sense. To this end, we will present another strategy which is able to compensate for these disadvantages. The strategy is already included in Figure 4.11: the greedy strategy.

### 4.3.2 *Greedy strategy*

A strategy which immediately suggests itself is a greedy strategy. Starting with $t = 0$, one chooses the prefix interval with the highest rating and checks if the benefit-cost difference is better than the current optimum. Repeating these commands until all prefix intervals have been chosen results in an optimal value $t^*$ of tunnels to be used. Additionally, during the choice of a prefix interval, we add the run identifier $k$ of the start of a prefix interval to an array SPI. The prefix intervals with the best benefit-cost difference can then be found in $\mathsf{SPI}[1..t^*]$.

The strategy is shown in Algorithm 4.6. The greedy approach is quite similar to the hirsch strategy, but allows the possibility of a "pawn sacrifice". This means that the final choice of prefix intervals to be tunneled can include prefix intervals where the tunnel benefit is less than the average tunnel cost. It is clear that the strategy is optimal when no side effects between prefix intervals occur. A disadvantage of this approach is that the prefix intervals have to be sorted according to their rating. By using a standard sorting algorithm or a heap [Wil64], this takes $O(|\mathcal{R}| \cdot \log |\mathcal{R}|)$

```
Data: RPTC array, RPE-array, BWT statistics $n_{\mathsf{rle}}$, $rc$, $r := |\mathcal{R}|$ and $r_{h>1}$, benefit function
      $\mathsf{benefit}(tc) := tc \cdot \left(1 + \log_2\left(\frac{n_{\mathsf{rle}}}{rc}\right)\right)$, cost function $\mathsf{cost}(t) := (t + 0.5) \cdot \left(6 + 4 \cdot \log_2\left(\frac{r_{h>1}+1}{2 \cdot t+1} - 1\right)\right)$.
Result: Modified RPE-array where all prefix intervals which should not be tunneled contain a $\perp$.

let SPI be an array of size $r$
initialize an empty heap $H$
for $k \leftarrow 1$ to $r$ do
    if $\mathsf{RPE}[k] \neq \perp$ then
        add $\langle k, \mathsf{RPTC}[k] \rangle$ to $H$

$t \leftarrow 0$
$tc \leftarrow 0$
$t^* \leftarrow 0$
$tc^* \leftarrow 0$
while $H$ is not empty do                                        // greedily choose prefix intervals
    let $\langle k, rptc \rangle$ be the pair with the highest rating $rptc$ in $H$
    remove $\langle k, rptc \rangle$ from $H$
    $t \leftarrow t + 1$
    $tc \leftarrow tc + rptc$
    if $\mathsf{benefit}(tc) - \mathsf{cost}(t) \geq \mathsf{benefit}(tc^*) - \mathsf{cost}(t^*)$ then
        $t^* \leftarrow t$
        $tc^* \leftarrow tc$
    $\mathsf{SPI}[t] \leftarrow k$

for $i \leftarrow t^* + 1$ to $t$ do                             // choose prefix intervals to be tunneled
    $k \leftarrow \mathsf{SPI}[i]$
    $\mathsf{RPE}[k] \leftarrow \perp$

return RPE
```

**Algorithm 4.6:** Greedy tunnel planning strategy. The result of this algorithm can be used to tunnel a BWT, see Algorithm 4.2.

time. Therefore, the greedy approach is slower than the hirsch strategy, which takes only $O(|\mathcal{R}|)$ time. Furthermore, the strategy does not consider side-effects between prefix intervals.

A refined greedy strategy allows one to consider negative side effects between prefix intervals. This refined greedy strategy is the first published tunnel planning strategy of all time, presented by the author of this thesis [Bai18]. The idea is to choose prefix intervals in a greedy manner, similar to Algorithm 4.6. The difference is that the ratings of prefix intervals are updated when negative side effects with the currently chosen prefix interval are detected.

The idea of the updating strategy is as follows: if only one prefix interval should be tunneled, the best one can do is to pick the prefix interval with the highest rating. Tunneling this prefix interval then causes negative side effects on other prefix intervals, so the rating has to be updated. Updating the ratings of overlapping prefix intervals then decreases the rating of some prefix intervals, so the first choice still is the best. The best choice of two prefix intervals to be tunneled then consists of the best choice of tunneling one prefix interval and the best choice under the updated remaining prefix intervals. In other words, the best choice of $t$ tunnels is the same as the best choice of $t-1$ tunnels, extended by the best-rated remaining prefix interval. Therefore, we assume that Bellman's principle of optimality [CN09a] is fulfilled.[2] This allows us to compute an optimal choice of exactly $t$ prefix intervals for all possible values of $t$, and compare the benefit-cost difference of each solution to find the optimal value $t^*$. Analogously to the normal greedy strategy from above, we can use an array SPI to store the sorted prefix intervals.

The greedy strategy that considers negative side effects between prefix intervals is shown in Algorithm 4.7. The algorithm requires an additional array length that stores the length of each length-maximal run-terminated prefix interval. Let $\langle w, [i, j] \rangle$ be a length-maximal run-terminated prefix interval that starts at the $k$-th run, i.e. $k = \mathsf{rank}_{B_{\mathcal{R}}}(1, i)$. Then, $\mathsf{length}[k] = w - 1$ holds. The length array can be obtained as a by-product of the rating array computation, see Algorithm 4.4.

The handling of negative side-effects is split into two cases. First, assume that the current chosen prefix interval $P$ points through another length-maximal run-terminated prefix interval $P'$. Such a situation is shown in Figure 4.12. This corresponds to the lines 20–24 of Algorithm 4.7. Now consider a run of height $h_r$ where both $P$ and $P'$ point through. Let $h$ and $h'$ be the heights of the prefix intervals $P$

[2] Note that Bellman's principle only holds true as long as no positive side effects occur.

```
Data: RPTC array, RPE-array, BWT statistics $n_{\mathsf{rle}}$, $rc$, $r := |\mathcal{R}|$ and $r_{h>1}$, benefit function
      $\mathsf{benefit}(tc) := tc \cdot \left(1 + \log_2\left(\frac{n_{\mathsf{rle}}}{rc}\right)\right)$, cost function $\mathsf{cost}(t) := (t + 0.5) \cdot \left(6 + 4 \cdot \log_2\left(\frac{r_{h>1}+1}{2 \cdot t+1} - 1\right)\right)$, length array
      length storing the length of each length-maximal prefix interval starting at the run with identifier $k$.
Result: Modified RPE-array where all prefix intervals which should not be tunneled contain a $\perp$.

let SPI be an array of size $r$
initialize an empty heap $H$
for $k \leftarrow 1$ to $r$ do
    if $\mathsf{RPE}[k] \neq \perp$ then
        add $\langle k, \mathsf{RPTC}[k] \rangle$ to $H$

$t \leftarrow 0$
$tc \leftarrow 0$
$t^* \leftarrow 0$
$tc^* \leftarrow 0$
while $H$ is not empty do                                        // greedily choose prefix intervals
    let $\langle k, rptc \rangle$ be the pair with the highest rating $rptc$ in $H$
    remove $\langle k, rptc \rangle$ from $H$
    $t \leftarrow t + 1$
    $tc \leftarrow tc + rptc$
    if $\mathsf{benefit}(tc) - \mathsf{cost}(t) \geq \mathsf{benefit}(tc^*) - \mathsf{cost}(t^*)$ then
        $t^* \leftarrow t$
        $tc^* \leftarrow tc$
    $\mathsf{SPI}[t] \leftarrow k$
    $\mathsf{RPTC}[k] \leftarrow 0$                    // perform rating updates of overlapping prefix intervals
    foreach run $k'$ where the length-maximal run-terminated prefix interval starting at run k points through do
        if $\mathsf{RPTC}[k'] > 0$ then
            remove $\langle k', \mathsf{RPTC}[k'] \rangle$ from the heap $H$
            $\mathsf{RPTC}[k'] \leftarrow \max\{\mathsf{RPTC}[k'] - \frac{\mathsf{length}[k']-2}{\mathsf{length}[k]-2} \cdot \mathsf{RPTC}[k], 0\}$
            add $\langle k', \mathsf{RPTC}[k'] \rangle$ to $H$

    foreach length-maximal run-terminated prefix interval starting at run $k'$ that points through the run $k$ do
        if $\mathsf{RPTC}[k'] > 0$ then
            remove $\langle k', \mathsf{RPTC}[k'] \rangle$ from the heap $H$
            $\mathsf{RPTC}[k'] \leftarrow \max\{\frac{\mathsf{length}[k']-\mathsf{length}[k]}{\mathsf{length}[k']-2} \cdot \mathsf{RPTC}[k'], 0\}$
            $\mathsf{length}[k'] \leftarrow \mathsf{length}[k'] - (\mathsf{length}[k] - 2)$
            add $\langle k', \mathsf{RPTC}[k'] \rangle$ to $H$

for $i \leftarrow t^* + 1$ to $t$ do                              // choose prefix intervals to be tunneled
    $k \leftarrow \mathsf{SPI}[i]$
    $\mathsf{RPE}[k] \leftarrow \perp$
return RPE
```

**Algorithm 4.7:** Greedy tunnel planning strategy with updates. The result of this algorithm can be used to tunnel a BWT, see Algorithm 4.2.

and $P'$. Without tunneling $P$, the benefit of tunneling $P'$ for this single column is approximately $\log_2(h_r) - \log_2(h_r - (h' - 1))$, see Definition 4.9. When $P$ is tunneled, the height of the run $r$ is reduced by $h - 1$ characters. Also, the height of the prefix

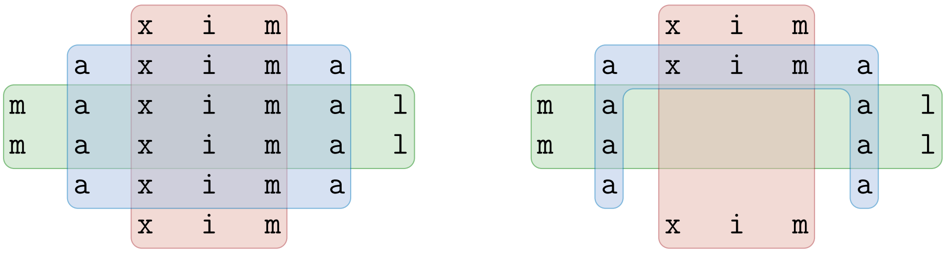


**Figure 4.12:** Effects of tunneling when overlapping prefix intervals are considered. The left-hand side shows an illustration of three overlapping length-maximal run-terminated prefix intervals. The right-hand side shows the situation when the blue prefix interval is tunneled. The height of the red prefix interval is reduced because the blue prefix interval points through. These cases are handled by the lines 20–24 of Algorithm 4.7. The length of the green prefix interval is reduced because it points through the blue prefix interval. These cases are handled by the lines 25–30 of Algorithm 4.7.

interval $P'$ is reduced by $h-1$ characters. Therefore, the new benefit of tunneling $P'$ for the same column is about

$$\begin{aligned}
&\log_2(h_r-(h-1))-\log_2(h_r-(h-1)-(h'-1-(h-1)))\\
&\quad=\log_2(h_r-(h-1))-\log_2(h_r-(h'-1))\\
&\quad=\underbrace{\log_2(h_r)-\log_2(h_r-(h'-1))}_{\text{benefit when tunneling only } P'}-\left(\underbrace{\log_2(h_r)-\log_2(h_r-(h-1))}_{\text{benefit when tunneling only } P}\right).
\end{aligned}$$

Therefore, the rating of $P'$ could be updated by subtracting the column-wise rating of $P$ from the column-wise rating of $P'$. Unfortunately, the rating is stored as a single number and not column-wise. To overcome this problem, the column-wise subtraction can be approximated by subtracting the length-reduced benefit of $P$ from $P'$:

$$\mathsf{RPTC}[k'] \leftarrow \mathsf{RPTC}[k'] - \frac{\mathsf{length}[k']-2}{\mathsf{length}[k]-2}\cdot\mathsf{RPTC}[k].$$

The subtraction of two from each length is done because a row in a prefix interval of length $l$ has $l$ labels, and in the special case of our tunnel encoding, no labels from the first and last column of the prefix interval are removed.

The computation of all affected runs (line 20) is straight forward: using the Run-LF support, we can iterate over all columns of the chosen prefix interval and find the surrounding runs, see Algorithm 4.4. According to Corollary 4.5, the sum of the lengths of all prefix intervals in $\mathcal{RP}$ is less than or equal to $n$. Therefore, the overall number of updates for the first kind of overlapping prefix intervals is limited by $n$. Each update requires updates in the heap, so the overall run-time for these updates is $O(n \log_2(r_{h>1}))$.

We now come to the second kind of updates: an overlapping prefix interval $P'$ that points through the chosen prefix interval $P$. Such updates are performed in the lines 25–30 of Algorithm 4.7. As can be seen in Figure 4.12, tunneling the prefix interval $P$ trims the height of $P'$ in the intersection area to one. Therefore, after tunneling $P$, tunneling $P'$ cannot produce any profit in the intersection area. To update the rating of $P'$, one could subtract the benefit in the intersection area from the whole benefit. However, as above, the only available benefit is the benefit of the whole prefix interval. Therefore, we decrease the benefit by multiplying the whole benefit by the reduced-length ratio:

$$\begin{aligned}\mathsf{RPTC}[k'] &\leftarrow \frac{(\mathsf{length}[k'] - 2) - (\mathsf{length}[k] - 2)}{\mathsf{length}[k'] - 2} \cdot \mathsf{RPTC}[k'] \\ &= \frac{\mathsf{length}[k'] - \mathsf{length}[k]}{\mathsf{length}[k'] - 2} \cdot \mathsf{RPTC}[k'].\end{aligned}$$

The subtraction of two from each length has the same meaning as explained above. Algorithm 4.7 also updates the length of the overlapping prefix interval $P'$ in line 29. The reason is that the prefix interval $P'$ was "virtually" shortened by the length of $P$ (minus two).

Enumerating all length-maximal prefix intervals that point through a run $k$ (line 25 of Algorithm 4.7) can be done using a graph. For each run $k$, we store references to all run-terminated prefix intervals that point through $k$. Such a graph can be constructed by enumerating all columns of all prefix intervals and adding the identifier of the interval to the run surrounding the current column. According to Corollary 4.5, this takes $O(n)$ time, and the graph has at most $n$ edges. Thus, at most $n$ updates of this kind are performed, so the run-time for all updates of the second kind is $O(n \log_2(r_{h>1}))$.

To summarize, the greedy strategy that considers negative side effects has a worst-case time complexity of $O(n \log_2(r_{h>1}))$. It should be noted that the side effect updates are approximated and do not precisely reflect the situation.

## 4.4 EXPERIMENTAL RESULTS

We implemented the aforementioned strategies using the cost model from Section 4.2. The implementation is publicly available [Bai20]. The following strategies were used:

- hirsch: Hirsch strategy from Section 4.3.1.
- greedy: Greedy strategy ignoring negative side effects, Algorithm 4.6.
- gupdate: Greedy strategy that considers negative side effects, Algorithm 4.7.
- debruijn: Tunneling strategy that uses de Bruijn graph edge minimization, Chapter 5.

The last strategy will be described in the next chapter and is optimized for the purpose of sequence analysis. The main difference is that it tunnels non-overlapping prefix intervals, and that it is not possible to reduce the length of the components $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$. This is a significant difference to the tunneled BWT representation when only length-maximal run-terminated prefix intervals are tunneled. The latter allows for the reduction of the size of aux to the number of runs, which typically is considerably smaller than the length of the string itself.

We enhanced the following BWT compressors with tunneling:

- bw94: block-sorting compression as proposed by Burrows and Wheeler [BW94], see Section 2.2.4.
- bcm: one of the best open-source available BWT compressor to date [Mur16].

Additionally, we included the following compressors that are using other compression paradigms:

- xz: one of the best LZ77-based compressors to date. It uses the Lempel-Ziv-Markov Algorithm [Col10].
- zpaq: one of the best context-mixing compressors to date [Mah09].

More information about the compressors, the used computer platform and the full benchmark results can be found in Chapter B. We tested the performance of the compressors with 44 test files coming from 6 different text corpora:

- canterbury: 11 small files between 4 KB and 1 MB.

- largecanterbury: 3 small files between 2.3 and 4.4 MB.
- silesia: 12 medium-sized files between 5 MB and 49 MB.
- pizzachili: 6 large files between 53 MB and 2108 MB.
- repetitive: 9 large and highly repetitive files between 45 MB and 440 MB.
- gnomes: 3 reference genomes with file sizes between 2500 MB and 2900 MB.

A full description of each corpus and detailed test data statistics can be found in Chapter A. Figure 4.13 shows the impact of tunneling on the compression rates of bw94 and bcm. For all strategies except of the debruijn, the encoding size is reduced by an average of 8% in the case of bw94 and by an average of 10% for bcm. In the case of the debruijn strategy, tunneling increases the encoding size by an average of 22% for bw94 and 20.5% for bcm. This clearly shows that tunneling has a positive effect on compression rates when length-maximal run-terminated prefix intervals with the special tunneled BWT encoding are used. Also, the less desirable results of the debruijn strategy show that tunneling has to be engineered carefully to achieve improvements in data compression.

Going more into the details of Figure 4.13, one can see that tunneling has only minor influence on the compression of most test files. This holds especially true for

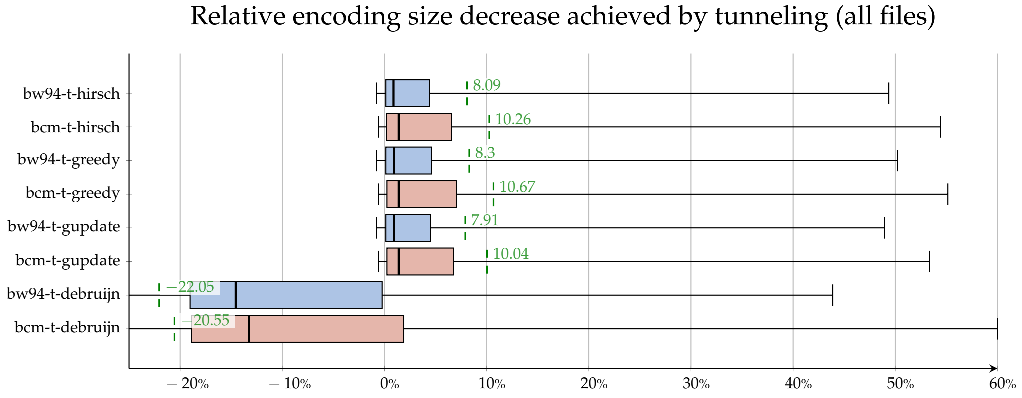


**Figure 4.13:** Boxplot of the relative encoding size decrease achieved by tunneling. The boxplot uses the results of all test files described in Chapter A. The blue boxes indicate tunneling in conjunction with the bw94 compressor, red boxes indicate tunneling in conjunction with bcm. The box plots consist of minimum (left whisker), lower quartil, median and upper quartil (boxes) and maximum (right whisker). The green lines indicate the average encoding size decrease. Full information about encoding size decreases for each compressor and each file can be found in Table B.7. A similar image was already published in [BD19] © 2019 IEEE.

the small and medium-sized files from the canterbury corpus, the largecanterbury corpus and the silesia corpus. Furthermore, the influence of tunneling on compression rates is small when the underlying data is difficult to compress, which for example is the case for reference genomes. The best results of tunneling are achieved when the data is highly repetitive. In the case of text collection files in the pizzachili corpus or repetitive files from the repetitive corpus, tunneling is able to reduce the encoding size by up to 55%.

To this end, Figure 4.14 shows the same boxplot for files from the pizzachili corpus and repetitive corpus. Tunneling shows its full potential for these files: the average encoding size reduction is approximately 22% for bw94 and 27% for bcm. Because the debruijn strategy achieves poor results, we omitted the debruijn strategy in this plot. More information about the improvements of tunneling on the compression rates can be found in Section B.2.

Next, we want to compare the influence of the tunneling strategy on the compression results. As Figures 4.13 and 4.14 show, the greedy strategy achieves the best results while the gupdate strategy achieves the worst. However, the differences range around 2%, showing that all strategies work well. Surprisingly, as the comparatively bad results from the gupdate strategy show, it is not worth it to consider negative side effects between prefix intervals. As the results from the greedy and gupdate strategy differ, the BWTs must contain overlapping prefix intervals. However, the bad results of the gupdate strategy imply that negative and positive side effects cancel each other out.

Regarding resource requirements, the average encoding speed of bcm is decreased by 35% for the hirsch strategy, by 47% for the greedy strategy and by 48% for the gupdate strategy. In the case of the bw94 compressor, the average encoding speed is

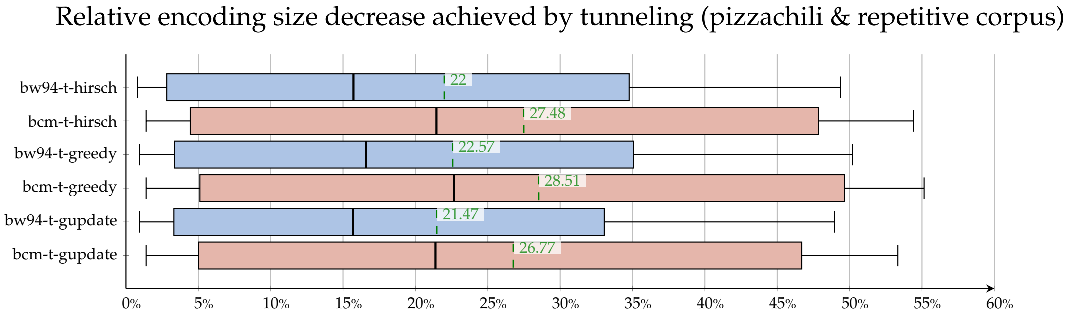


**Figure 4.14:** Boxplot of the relative encoding size decrease achieved by tunneling. The boxplot uses the results of files from the pizzachili and repetitive text corpus, see Chapter A. A similar image was already published in [BD19] © 2019 IEEE.

decreased by 45% for the hirsch strategy, and by 53% for both the greedy and the gupdate strategy. The decoding speed remains identical for almost all files. In the case of files with good compression rates, the compression speed was improved by a small amount, see Section B.1.

A critical resource required for compression is memory. In this case, a normal BWT compressor requires about 5 times the input size for suffix array construction using divsufsort [Mor03]. The hirsch strategy increases this amount by 30% on average, the greedy strategy by 42% and the gupdate strategy by 76%. The reason is that the hirsch strategy needs the additional $C_{MT}$-array, greedy requires the bigger SPI array and gupdate requires an additional overlapping graph. For decompression, again, the memory peak remains identical or decreases when good compression was achieved. Full data about memory peaks and speeds can be found in Section B.1. Taking the resource requirements into account, our BWT compressor of choice is the bcm compressor with the hirsch tunneling strategy.

Table 4.1 shows a comparison of the BWT-cased compressors with xz and zpaq for selected files. A table containing all compression information can be found on Page 207. In general, zpaq works best for small files from the canterbury and largecanterbury corpus. Results for medium-sized files from the silesia corpus are mixed, but bcm-t-greedy, xz and zpaq dominate the other compressors. In the case of large files from the pizzachili corpus, both bcm-t-greedy and zpaq offer the best compression. Very repetitive files should be compressed with xz or zpaq, but bcm-

| Text corpus | File | bw94 | bw94-t-hirsch | bw94-t-greedy | bw94-t-gupdate | bcm | bcm-t-hirsch | bcm-t-greedy | bcm-t-gupdate | xz | zpaq |
|---|---|---|---|---|---|---|---|---|---|---|---|
| canterbury | alice29.txt | 2.354 | 2.354 | 2.354 | 2.354 | 2.130 | 2.129 | 2.129 | 2.129 | 2.552 | **2.032** |
| | sum | 2.779 | 2.657 | 2.651 | 2.654 | 2.557 | 2.389 | 2.377 | 2.384 | **1.987** | 2.186 |
| largecanterbury | bible.txt | 1.663 | 1.660 | 1.659 | 1.659 | 1.440 | 1.435 | 1.433 | 1.434 | 1.750 | **1.391** |
| silesia | nci | 0.339 | 0.327 | 0.327 | 0.327 | 0.292 | 0.276 | **0.274** | 0.275 | 0.345 | 0.362 |
| | x-ray | 4.244 | 4.244 | 4.244 | 4.244 | **3.452** | 3.452 | 3.452 | 3.452 | 4.239 | 3.560 |
| pizzachili | proteins | 2.289 | 1.972 | 1.965 | 1.977 | 2.331 | 1.912 | **1.901** | 1.920 | 2.222 | 2.609 |
| | english | 1.711 | 1.470 | 1.469 | 1.471 | 1.478 | 1.184 | **1.183** | 1.186 | 1.985 | 1.683 |
| repetitive | cere | 0.237 | 0.120 | 0.118 | 0.121 | 0.238 | 0.118 | 0.115 | 0.119 | **0.087** | 1.771 |
| | world-leaders | 0.121 | 0.100 | 0.098 | 0.100 | 0.126 | 0.097 | 0.094 | 0.097 | **0.088** | 0.093 |
| genomes | hg38 | 1.754 | 1.723 | 1.723 | 1.723 | 1.645 | 1.608 | **1.607** | 1.608 | 1.716 | 1.826 |

**Table 4.1:** Compression results in bits per symbol for selected files. The best compression result of each file is printed in bold. A full list of compression results for all files can be found in Table B.2.

t-greedy offers a good and robust compression which is always close to the best compression result. Finally, for whole reference genomes, bcm-t-greedy offers the best compression.

In our opinion, xz is the best compressor for the typical application in a download server. In the case of a download server application, decompression must be fast and resource-saving because client systems have limited computational resources. The compression using xz is resource-intensive and slow, but the decompression speed is very high, see Table B.4. The memory peak of xz during decompression is the best among all tested decompressors, underlining the strength of xz. Furthermore, xz offers competitive compression rates which accelerate the transmission time when the bandwidth is limited.

The zpaq compressor offers very good compression for some, but not all files. The strength of zpaq lies in the context-mixing technique, which allows one to adapt the compressor to a specific data source. As a consequence, other versions of context-mixing compressors still dominate the ongoing hutter prize compression competition nowadays [HMB06]. The problem of zpaq is that it is a rather slow compressor.

BWT-based compressors always had the problem that their decompression speed was too slow to be competitive. Moreover, the memory peak during decompression is very high when the underlying string is long. We did not increase the decompression speed by much, so BWT-based compressors still remain a niche product. However, our goal was to improve the compressibility of a BWT, which in our opinion can be seen as accomplished. This shows that it is worth it to further invest research on faster decompression methods.

In contrast to xz and zpaq, BWT-based compressors do not only transport a string. Instead, they transport a BWT, which is a significant additional value when the underlying data should be analyzed. In Section 2.1.1 we have seen that a BWT encoded in a wavelet tree is suitable to do efficient pattern search. Moreover, a BWT allows approximative pattern search and a lot more indexing operations. BWT compression thus should be seen as index transport rather than as pure data transport. We have seen that tunneling achieves good compression results when the indexes are big and the underlying data is repetitive, which is the typical case of applications of indexing. Therefore, this chapter can be seen as a “transport protocol” of BWT-based indices. In the next chapter we will see that tunneling is more than a pure “transport protocol” of indices: it is a full compression scheme of such indices.

# 5 APPLICATION IN SEQUENCE ANALYSIS

In the last chapter we have seen that tunneling works well in the area of data compression. What this chapter proposes is to show how tunneling can be applied to the area of sequence analysis. One of the most popular problems in sequence analysis is the exact string matching problem: given a string $S$ of length $n$ and a pattern $P$ of length $m$, one wants to find all occurrences of the pattern $P$ in the sequence $S$, see also Page 10.

Many efficient solutions for problems in sequence analysis require additional data structures to work fast. Examples of such data structures are given by rank- and select support, wavelet trees or balanced parentheses, see Section 2.3. As the sequences to be analyzed can get very large[1], these data structures have to be designed very carefully to ensure that they fit in memory and offer acceptable performance. This has lead to the research trend called "succinct data structures": the target is to design data structures whose size is close to the information-theoretic lower bound of the underlying problem while still offering high performance.

This means that the worst-case run-time of an operation is sub-linear. For example, if one wants to answer $\mathsf{rank}_S(c, i)$ queries, one could scan the underlying string $S$ from position 1 to $i$ and accumulate the number of $c$'s. This solution requires no additional space at all, but has a worst-case run-time of $O(n)$, which is clearly not sub-linear. A further idea could be to precompute each $\mathsf{rank}_S(c, i)$-value and store it in a two-dimensional table. This results in an $O(1)$ worst-case run-time, but requires $\sigma \cdot n \cdot \log(n)$ bits of space, which is much too big for practical applications. A balanced wavelet tree instead requires about $n \cdot \log(\sigma)$ bits of space and has a worst-case run-time of $O(\log \sigma)$, which can be seen as a better time-space tradeoff. In general, a common phenomenon is that the run-time is increased when less space is used: more space means that more information can be precomputed.

Moving over to tunneling, tunneling decreases the size of a BWT but requires two additional bit-vectors and a more complex backward step for the special tunnel handling. Despite that the asymptotic worst-case run-time of a backward step does not change, the expectation is that the real-world run-times increase because of

[1] A human genome has roughly about 3 billion base pairs.

the more complex characteristics of a backward step in a tunneled BWT. Therefore, tunneling shifts a given time-space tradeoff in the direction of less space (depending on the input) and more run-time.

In this chapter, we will present an approach to compute a special sort of non-overlapping prefix intervals. Alanko et al. have shown that such a non-overlapping prefix interval collection can be used to build a tunneled FM-index [Ala+19]. Instead, we will present a succinct representation of tries, the so-called extended BWT [Fer+05]. We will use the prefix interval collection to tunnel these tries. This results in a trie representation with an unprecedented level of succinctness.

## 5.1 DE BRUIJN GRAPH EDGE REDUCTION AND TUNNELING

As previously mentioned, tunneling only non-overlapping prefix intervals is beneficial for sequence analysis. The main reason is that navigational operations tend to be easier because there is no need for a stack saving the different tunnel entry points. Non-overlapping prefix intervals can be used to obtain a tunneled FM-index with a special text sampling scheme [Ala+19] but also to obtain a tunneled trie preserving useful combinatorial properties. This will be introduced in Section 5.4.

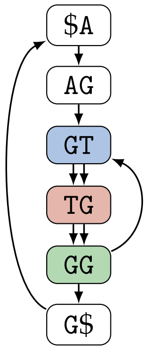
In Section 3.4 we have seen that tunnel planning is hard in the case of overlapping prefix intervals but did not give an affirmative answer to the open problem of non-overlapping tunnel planning, as stated by Alanko et al.

> How to find the optimal blocks that minimize space? [Ala+19]

We will not directly solve the problem here but give another approach which is quite similar and preserves combinatorial properties which will be very useful in Section 5.4. First of all, we want to repeat the connection between de Bruijn graphs on sequences (Definition 2.17) and the BWT.

Let $S$ be a string of length $n$, and let $k \in [1, n]$ be some integer. Every k-mer $x \in \mathcal{K}$ from the $k$-cyclic string $Z_k(S)$ then corresponds to a unique $x$-interval in the sorted rotations of $S$. Let $y \in \mathcal{K}$ be any preceding node of $x$, i.e. $y[2..k] = x[1..k-1]$. Then, the $y$-interval can be entered by performing a backward search using the $x$-interval and the character $c = y[1]$. Figure 5.1 shows the connection between de Bruijn graphs and a BWT matrix.

Moreover, there exists a deep connection between a special sort of paths in a de Bruijn graph and prefix intervals in a BWT matrix, which is presented next.



| $i$ | SA[$i$] | L[$i$] | $S$[SA[$i$]..$n$]$S$[1..SA[$i$] − 1] |
|---|---|---|---|
| 1 | 9 | G | $AGTGGTGG |
| 2 | 1 | $ | AGTGGTGG$ |
| 3 | 8 | G | G$AGTGGTG |
| 4 | 7 | T | GG$AGTGGT |
| 5 | 4 | T | GGTGG$AGT |
| 6 | 5 | G | GTGG$AGTG |
| 7 | 2 | A | GTGGTGG$A |
| 8 | 6 | G | TGG$AGTGG |
| 9 | 3 | G | TGGTGG$AG |

**Figure 5.1:** Connection between the de Bruijn graph of order $k = 2$ (left) and the BWT matrix (right) of the string $S =$ AGTGGTGG$. The colored k-mer nodes on the left correspond to k-mer-intervals in the sorted rotations on the right. A non-forking path in $G_2(S)$ is given by (GT, TG, GG). The corresponding 2-mer prefix interval in the BWT matrix is given by $\langle 3, [8, 9] \rangle$. Parts of this image were already published in [Bai+20] © 2020 IEEE.

**Definition 5.1** (Non-forking path)**.** Let $S$ be a string of length $n$ with de Bruijn graph $G_k(S) = (\mathcal{K}, E)$ for some $k \in [1, n]$. Let $p = (x_1, x_2, \ldots, x_l)$ be a sequence of nodes. We say that $p$ is a non-forking path in $G_k(S)$ if and only if for all $i \in [1, l-1]$

- $x_{i+1}$ is the only successor of $x_i$ in $G_k(S)$.
- $x_i$ is the only predecessor of $x_{i+1}$ in $G_k(S)$.
- $x_i$ has an outdegree greater than one.

We call $p$ length-maximal if it is no proper subpath of any other non-forking path.

**Definition 5.2** (k-mer prefix interval)**.** Let $S$ be a string of length $n$ with BWT L, let $k \in [1, n]$ be an integer. A prefix interval $\langle w, [i, j] \rangle$ is a k-mer prefix interval if and only if each column $[\mathsf{LF}^x[i], \mathsf{LF}^x[j]]$ $(0 \leq x < w)$ of the prefix interval corresponds to a k-mer interval in the BWT matrix of $S$.

Examples of non-forking paths and k-mer prefix intervals can be found in Figure 5.1. In Figure 5.1, it can be seen that a one to one correspondence between the non-forking path and the k-mer prefix interval exists. The following theorem demonstrates this correspondence and in a similar form was presented by the author of this thesis in [Bai+20].

**Theorem 5.3.** *Let $S$ be a string of length $n$ with de Bruijn graph $G_k(S) = (\mathcal{K}, E)$ for some $k \in [1, n]$, and let* L *be the BWT of $S$.*

*(i) There is a one to one correspondence between non-forking paths in $G_k(S)$ and k-mer prefix intervals in* $\mathsf{L}$.

*(ii) k-mer prefix intervals corresponding to length-maximal non-forking paths are non-overlapping.*

*Proof.*

(i) Let $p$ be a non-forking path in $G_k(S)$, and let $x$ and $y$ be two adjacent nodes in $p$ with $x$-interval $[i_x, j_x]$ and $y$-interval $[i_y, j_y]$ in the BWT matrix. W.l.o.g. let $x$ be the only predecessor of $y$ and let $y$ be the only successor of $x$. Because $x$ is the only predecessor of $y$, $\mathsf{L}[i_y] = \cdots = \mathsf{L}[j_y]$ and $[\mathsf{LF}[i_y], \mathsf{LF}[j_y]] \subseteq [i_x, j_x]$ must hold. Additionally, as $y$ is the only successor of $x$, $[\mathsf{LF}[i_y], \mathsf{LF}[i_x]] = [i_x, i_y]$ must hold. Therefore, the nodes of $x$ correspond to k-mer intervals in the BWT matrix with equal height and interval-wise equal BWT entries, where the LF-mapping of any entry in a k-mer interval leads to the k-mer interval of the preceding k-mer. This suffices to show that $\langle l, [i_{x_l}, j_{x_l}] \rangle$ is a k-mer prefix interval.

Now, let $\langle w, [i, j] \rangle$ be a k-mer prefix interval in the BWT matrix. Because each column of $\langle w, [i, j] \rangle$ corresponds to a k-mer interval in the BWT matrix, we can map each column $[\mathsf{LF}^l[i], \mathsf{LF}^l[j]]$ $(0 \leq l < w)$ to some node $x_l$ in $G_k(S)$. The sequence $(x_{w-1}, \ldots, x_0)$ now is a non-forking path in $G_k(S)$: For any two adjacent nodes $x$ and $y$ in the sequence, $[\mathsf{LF}[i_y], \mathsf{LF}[j_y]] = [i_x, j_x]$ holds true by the definition of a prefix interval, so $x$ is the only predecessor of $y$ and $y$ is the only successor of $x$.

(ii) A node cannot be contained in two different length-maximal non-forking paths: this would imply that the paths are not length-maximal. Therefore, the k-mer intervals corresponding to both the nodes of the paths and the columns of the associated prefix intervals must be disjoint, so the associated prefix intervals are non-overlapping.

□

Theorem 5.3 shows that non-forking length-maximal paths correspond to non-overlapping prefix intervals. We will extend this connection by defining edge-reduced de Bruijn graphs similar to [Bai+20]. These graphs are strongly related to tunneled BWTs.

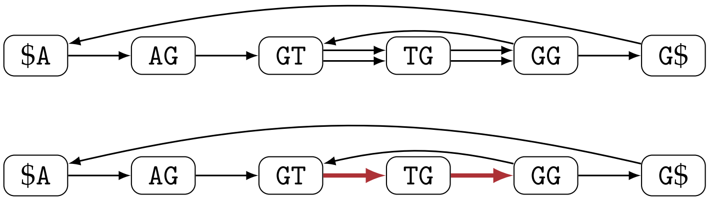


**Figure 5.2:** Normal and edge-reduced de Bruijn graphs with order $k = 2$ of the string $S =$ AGTGGTGG$. Fused edges are indicated by red arrows. Parts of this image were already published in [Bai+20] © 2020 IEEE.

**Definition 5.4** (Edge-reduced de Bruijn graph)**.** Let $S$ be a string of length $n$, let $k \in [1, n]$ be an integer and let $G_k(S) = (\mathcal{K}, E)$ be the de Bruijn graph of $S$. Denote by

$$F := \{ (x,y)^m \in E \mid (x,y) \text{ is a non-forking path } \}$$

the set of fusible edges. The edge-reduced de Bruijn graph $\tilde{G}_k(S) = (\mathcal{K}, \tilde{E})$ of $G_k(S)$ is a graph where the multiplicity of all fusible edges is reduced to one, that is,

$$\tilde{E} := (E \setminus F) \uplus \{ (x,y)^1 \mid (x,y)^m \in F \}.$$

Definition 5.4 describes a special compression scheme of non-forking paths in de Bruijn graphs: the number of edges is reduced. Another more common compression scheme merges the nodes of a non-forking path instead. This reduces the number of nodes, see e.g. [MLS14; BBO16] for more information.

When thinking about edge reductions, one notes that each multi-edge of a non-forking path is fused to just one edge. Similarly, when tunneling a prefix interval, the BWT entries between columns of a prefix interval are reduced to just one entry. The following theorem, presented in [Bai+20], states this connection more precisely. It shows that the reduction of edges in a de Bruijn graph analogously reduces the size of a BWT by tunneling the associated k-mer prefix intervals.

**Theorem 5.5.** *Let $S$ be a string of length $n$, let $k \in [1, n]$ be an integer and let $G_k(S) = (\mathcal{K}, E)$ be the de Bruijn graph of $S$. Let $\tilde{G}_k(S) = (\mathcal{K}, \tilde{E})$ be the edge-reduced de Bruijn graph of $G_k(S)$, and let $\tilde{\mathsf{L}}$, $\mathsf{D}_{\mathsf{out}}$, $\mathsf{D}_{\mathsf{in}}$ be a tunneled BWT obtained by tunneling all k-mer prefix intervals associated to length-maximal non-forking paths in $G_k(S)$. Then, the number of edges in $\tilde{G}_k(S)$ is equivalent to the size of the tunneled BWT, that is, $|\tilde{E}| = |\tilde{\mathsf{L}}|$.*

*Proof.* Let $p = (x_1, \ldots, x_l)$ be a length-maximal non-forking path in $G_k(S)$, and let $\langle l, [i,j] \rangle$ be its corresponding k-mer prefix interval according to Theorem 5.3. As each edge in $p$ from a node $x$ to its successor $y$ corresponds to a text position where the k-mers overlap, there must be $j - i + 1$ edges between $x$ and $y$. Thus, edge-reducing $p$ removes exactly $(l - 1) \cdot (j - i + 1)$ edges from $G_k(S)$. Analogously, tunneling $\langle l, [i, j] \rangle$ shortens the BWT L by exactly $(l - 1) \cdot (j - i + 1)$ characters. $G_k(S)$ initially consists of $n$ edges and L initially consists of $n$ entries. As the set $F$ of fusible edges from Definition 5.4 corresponds to all edges that are contained in length-maximal non-forking paths, the amount of remaining edges resp. entries after edge reduction resp. tunneling is equal. □

An example for the analogy between edge reductions and tunneling can be seen in Figure 5.3. The next goal will be to show how a tunneled BWT associated with an edge-reduced graph can be computed. First of all, given some $k \in [1, n]$, we need a way to efficiently store and access the k-mer intervals in the BWT matrix. We use a bit-vector $B$ of size $n + 1$ for this purpose: each left boundary of a k-mer interval will be marked with a 1. Furthermore, we set $B[n + 1]$ to 1. This implicitly gives us information about the end of an interval, as the start of the lexicographically next interval indicates the end of the current interval.

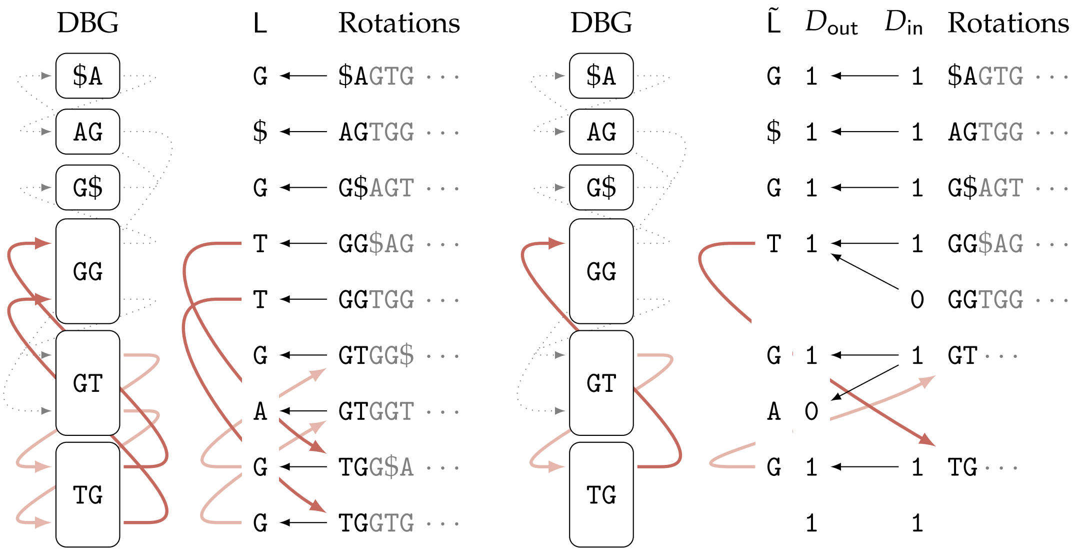


**Figure 5.3:** Analogy between an edge-reduced de Bruijn graph $\tilde{G}_2(S)$ and a tunneled BWT obtained by tunneling all 2-mer prefix intervals associated to length-maximal non-forking paths for the string $S =$ AGTGGTGG\$. The left-hand side shows that each edge in the de Bruijn graph $G_2(S)$ corresponds to an LF-mapping in the BWT, the right-hand side shows that this correspondence remains when edge-reducing the graph resp. tunneling length-maximal 2-mer prefix intervals. A similar image was already published in [Bai+20] © 2020 IEEE.

```
Data: BWT L in form of an FM-index for a string $S$ of length $n$, C-array $\mathsf{C_L}$, integer $k \in [1, n]$.
Result: A bit-vector $B$ of size $n+1$ such that $B[i] = 1$ iff $i$ is the left boundary of a k-mer interval or $i = n+1$.

1  initialize a bit-vector $B$ of size $n+1$ filled with zeros
2  initialize an empty queue $Q$
3  push element $[1, n]$ on $Q$
                                                  // compute k-mer intervals
4  for $l \leftarrow 1$ to $k$ do
5      for $q \leftarrow |Q| - 1$ to 0 do
6          $[i, j] \leftarrow$ first element of $Q$
7          pop first element of $Q$
8          $M \leftarrow \mathsf{getIntervals_L}(i, j)$
9          foreach $\langle c, [lb, rb] \rangle \in M$ do
10             push $[lb, rb]$ on the back of $Q$
                                      // mark left boundaries of the k-mer intervals
11 foreach $[i, j] \in Q$ do
12     $B[i] \leftarrow 1$
13 $B[n+1] \leftarrow 1$
14 return $B$
```

**Algorithm 5.1:** Computation of the k-mer interval boundaries bit-vector. The k-mer enumeration is adapted from an algorithm in [Ohl13, p. 324].

For the computation of the k-mer intervals, we make use of the getIntervals-function of wavelet trees. This allows us to enumerate all required k-mer intervals, and afterwards set the corresponding bits. Algorithm 5.1 shows a way to compute the k-mer interval boundaries.

The first part of Algorithm 5.1 computes k-mer intervals in increasing order: after each iteration of the outer loop, $Q$ contains all $l$-mer intervals where $l$ is the loop variable. This is ensured by the handling and removal of the frontmost interval in $Q$ (line 6) and the enqueuing of new intervals at the back of $Q$ (line 10). In the worst-case, each iteration of the outer for-loop (lines 4–10) creates $\min\{\sigma^l, n\}$ new $l$-mer intervals. Therefore, the worst-case time complexity of Algorithm 5.1 is $O(\sum_{l=1}^{k} \min\{\sigma^l, n\} \cdot \log(\sigma)) = O(k \cdot n \cdot \log(\sigma))$. This can only be accepted for small values of $k$. In the next section, we will see a different approach which has a better worst-case time complexity. For now, we want to focus on the computation of a tunneled BWT.

Given the k-mer interval boundaries in a bit-vector $B$, in the full version of [Bai+20], it has been shown how the bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ which are necessary for tunneling can be computed. We start by setting both $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ to copies of $B$. Then, for each k-mer interval $[i, j]$ we detect if $[i, j]$ and $[\mathsf{LF}[i], \mathsf{LF}[i]]$ are columns of a k-mer prefix interval (and thus also of a length-maximal k-mer prefix interval). According to the proof of Theorem 5.3, this can be done by checking two conditions: first, one checks if $\mathsf{L}[i] = \cdots = \mathsf{L}[j]$ holds. This implies that the de Bruijn graph node of the $[i, j]$-interval

```
Data: BWT L in form of an FM-index for some string $S$ of length $n$, markings $10^*1$ of potential prefix interval starts
      in $D_{\mathsf{in}}$ and potential prefix interval ends in $D_{\mathsf{out}}$.
Result: Markings of valid prefix intervals in $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$.
                                                        // process possible prefix interval starts
$i \leftarrow 1$
for $j \leftarrow 1$ to $n$ do
    if $D_{\mathsf{in}}[j+1] = 1$ then
        $M \leftarrow \mathsf{getIntervals}_{\mathsf{L}}(i, j)$
        $\langle c, [lb, rb] \rangle \leftarrow$ first element of $M$
                                        // check if interval contains multiple characters ($|M| > 1$)
            // and if it can be associated to a prefix interval end ($D_{\mathsf{out}}[\mathsf{LF}[i]..\mathsf{LF}[j]+1] = 10^{j-i}1$).
        if $|M| > 1$ or $D_{\mathsf{out}}[lb..rb+1] \neq 10^{j-i}1$ then
            $D_{\mathsf{in}}[i..j] \leftarrow 1^{j-i+1}$
            foreach $\langle c', [lb', rb'] \rangle \in M$ do
                $D_{\mathsf{out}}[lb'..rb'] \leftarrow 1^{rb'-lb'+1}$
        $i \leftarrow j+1$
```

**Algorithm 5.2:** Unmarking of invalid prefix intervals. The algorithm treats each $10^*1$ sequence in $D_{\mathsf{in}}$ as a potential start of a prefix interval. Potential ends of prefix intervals are indicated by a $10^*1$ sequence in $D_{\mathsf{out}}$. If the start of a potential prefix interval contains multiple different characters in L or cannot be associated to an end of a prefix interval, the prefix interval markings in $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ are unmarked. The algorithm produces a k-mer prefix interval marking if the parameters $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ are set to the k-mer boundary bit-vector $B$. The results of this algorithm can be used to obtain a tunneled BWT using Algorithm 3.3. A similar algorithm was already published in the full version of [Bai+20] © 2020 IEEE.

has only one predecessor. Second, one checks if $B[\mathsf{LF}[i]..\mathsf{LF}[j]+1] = 10^{j-i}1$ holds. This, together with the first condition, ensures that $[\mathsf{LF}[i], \mathsf{LF}[j]]$ is a k-mer interval of a de Bruijn graph node with only one successor.

If both conditions are true, we know that $[i, j]$ and $[\mathsf{LF}[i], \mathsf{LF}[j]]$ are k-mer intervals that belong to two adjacent nodes, where one node is the only predecessor and the other node is the only successor of each other. The nodes then belong to a non-forking path, and thus the intervals are columns of a length-maximal k-mer prefix interval. To obtain the bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$, analogously to the optimized tunneling of a set of prefix intervals in Algorithm 3.2, we would have to set $D_{\mathsf{in}}[i..j] = 10^{j-i}$ and $D_{\mathsf{in}}[\mathsf{LF}[i]..\mathsf{LF}[j]] = 10^{j-i}$. As $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ are copies of bit-vector $B$, this is already the case: both intervals are k-mer intervals, so we don't have to do anything. Instead, if the conditions are not met, we have to set $D_{\mathsf{in}}[i..j] = 1^{j-i+1}$ and $D_{\mathsf{out}}[\mathsf{LF}[i]] = \cdots = D_{\mathsf{out}}[\mathsf{LF}[j]] = 1$ because the entries $i, i+1, \ldots, j$ and $\mathsf{LF}[i], \ldots, \mathsf{LF}[j]$ do not belong to a k-mer prefix interval.

Executing the above procedure for all k-mer intervals in the BWT matrix then produces two bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ which are similar to the bit-vectors produced

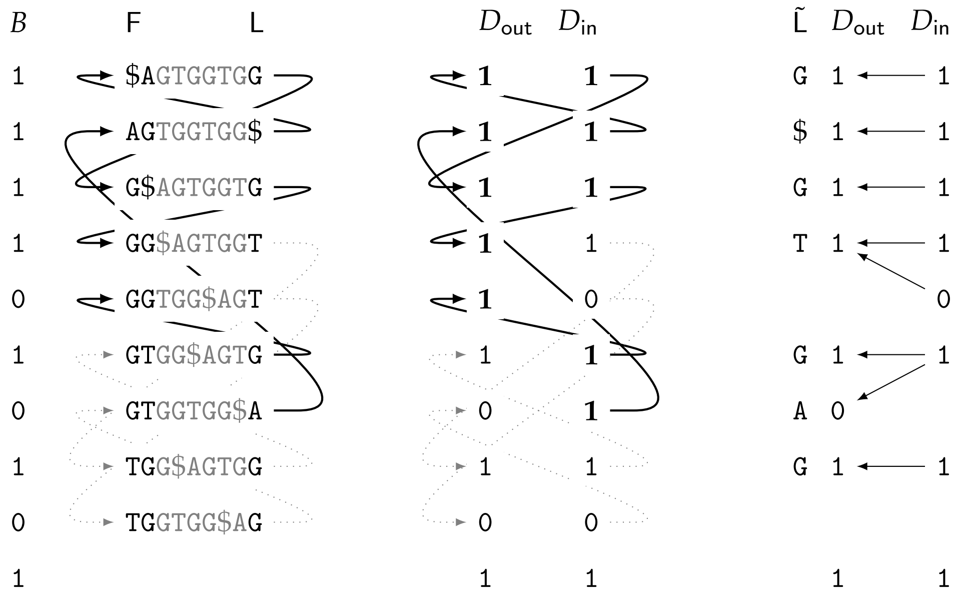


**Figure 5.4:** Tunneling of all length-maximal k-mer prefix intervals using the node boundary bit-vector $B$. The left-hand side shows the bit-vector $B$ and all LF-mappings that do not belong to a length-maximal prefix interval. The middle part shows the results of Algorithm 5.2 after execution with parameters $D_{\text{out}} = B$ and $D_{\text{in}} = B$. The algorithm sets entries to 1 that do not belong to a length-maximal k-mer prefix interval. The right-hand side shows the final tunneled BWT after applying Algorithm 3.3 to the result in the middle. Parts of this image were already published in [Bai+20] © 2020 IEEE.

by Algorithm 3.2. By applying Algorithm 3.3 on the result, the demanded tunneled BWT can be obtained.

The operating principle of Algorithm 5.2 is shown in Figure 5.4. The k-mer intervals are detected using the markings in the bit-vector $D_{\text{in}}$. It is important to use $D_{\text{in}}$ for the k-mer interval detection: if one would use $D_{\text{out}}$ to do so, side effects with line 9 could occur, so $D_{\text{out}}$ may not represents the node boundaries properly. Additionally, to speed up the algorithm, required LF-mapping entries are computed together in line 9. The worst-case run-time of Algorithm 5.2 is $O(n \log(\sigma))$, so given the k-mer node boundaries, a tunneled BWT can be obtained using $O(n \log(\sigma))$ time.

## 5.2 DE BRUIJN GRAPH EDGE MINIMIZATION

In the last section, we have seen that edge-reduced de Bruijn graphs and tunneled BWTs related to length-maximal k-mer prefix intervals are deeply connected. Thus, to produce a small tunneled BWT one needs to minimize edges in a de Bruijn graph. The size of an edge-reduced de Bruijn graph is fixed, so for a single order $k$, we cannot minimize anything. However, the amount of edges may vary depending on the order of the de Bruijn graph, see Figure 5.5. Thus we want to know the best order $k^*$ such that the edge-reduced de Bruijn graph $\tilde{G}_{k^*}(S)$ of a string $S$ has the minimal amount of edges.

**Problem 5.6** (De Bruijn graph edge minimization)**.** Given a string $S$ with length $n$, find the smallest order $k^* \in [1, n]$ s.t. the edge-reduced de Bruijn graph $\tilde{G}_{k^*}(S) = (\mathcal{K}^*, E^*)$ contains the minimal amount of edges:

$$k^* = \operatorname*{argmin}_{k \in [1,n]}\{ \sum_{(x,y)^m \in E} m \mid \tilde{G}_k(S) = (\mathcal{K}, E) \text{ is the order } k \text{ edge-reduced DBG of } S \}$$

Problem 5.6 has first been stated in [Bai+20]. A naive solution of the problem works as follows: Given a BWT L in form of a wavelet tree, we can enumerate all k-mer intervals with increasing order $k$ (similar to Algorithm 5.1) and thereby measure the amount of edges for each order. The measurement of edges can be done by an idea similar to the one in Algorithm 5.2: for each k-mer interval, we mark the left boundary in a bit-vector $B$. Afterwards, we enumerate the k-mer intervals and check if a k-mer interval $[i, j]$ has only one preceding character in L, as well as checking

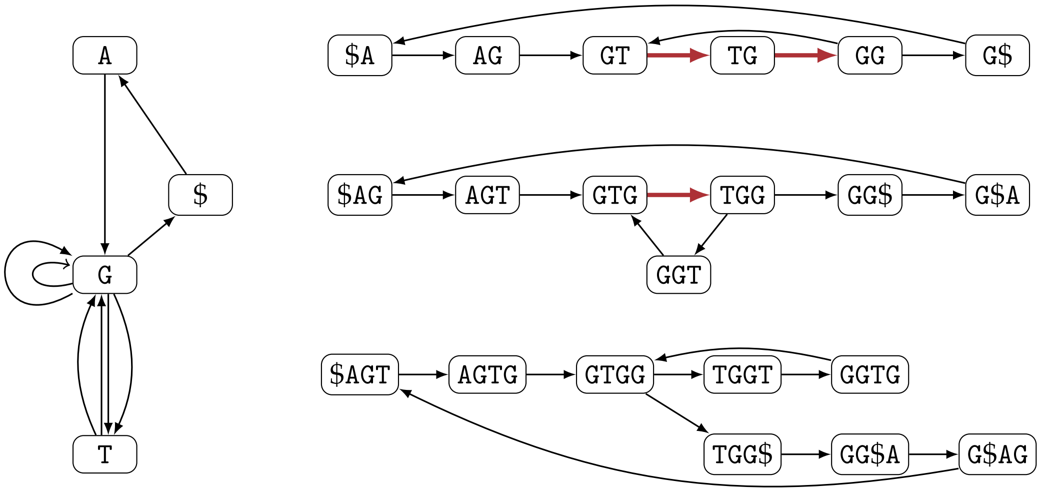


**Figure 5.5:** Edge-reduced de Bruijn graphs with order $k = 1$ (left), $k = 2$ (top right), $k = 3$ (middle right) and $k = 4$ (bottom right) of the string $S =$ AGTGGTGG\$. Fused edges are indicated with red arrows, the de Bruijn graph with order $k = 2$ contains the least edges, namely $9 - 2 = 7$ edges. A similar image was already published in [Bai+20] © 2020 IEEE.

```
Data: BWT L in form of an FM-index for a string $S$ of length $n$.
Result: Order $k^*$ such that the edge-reduced de Bruijn graph $\tilde{G}_{k^*}(S)$ has the minimal amount of edges.

1  initialize a bit-vector $B$ of size $n+1$ filled with zeros
2  initialize an empty queue $Q$
3  push element $[1,n]$ on $Q$
4  $B[1] \leftarrow 1$          $B[n+1] \leftarrow 1$
5  $k^* \leftarrow 1$           $m^* \leftarrow n$
6  for $k \leftarrow 0$ to $n$ do
7      $m \leftarrow n$
8      for $q \leftarrow |Q| - 1$ to $0$ do
9          $[i,j] \leftarrow$ first element of $Q$
10         pop first element of $Q$
11         $M \leftarrow \mathsf{getIntervals}_{\mathsf{L}}(i,j)$
                                                  // check for fusible edges
12         if $\langle c, lb, rb \rangle$ is the only element in $M$ and $B[lb] = 1$ and $B[rb+1] = 1$ then
13             $m \leftarrow m - (rb - lb)$
                                                  // store new nodes
14         foreach $\langle c, [lb, rb] \rangle \in M$ do
15             push $[lb, rb]$ on the back of $Q$
               // check if next order has too many nodes
16             if $|Q| - q \geq m^*$ then
17                 return $k^*$
                                                  // check for new minimum
18     if $m < m^*$ then
19         $k^* \leftarrow k$
20         $m^* \leftarrow m$
                                                  // mark node boundaries
21     foreach $[lb, rb] \in Q$ do
22         $B[rb+1] \leftarrow 1$
23 return $k^*$
```

**Algorithm 5.3:** Naive de Bruijn graph edge minimization algorithm. A similar algorithm was already published in the full version of [Bai+20] © 2020 IEEE.

if the bits $B[\mathsf{LF}[i]]$ and $B[\mathsf{LF}[j]+1]$ are set. As a k-mer interval gets partitioned into (possibly multiple) other intervals when the order is increased, it is not necessary to clear the set bits. However, we have to set additional bits when an interval is partitioned into multiple new intervals.

Algorithm 5.3 gathers these ideas. The enumeration of k-mer intervals is done in lines 8–11 and 14–15. At the beginning of each iteration of the outer for loop, the number of edges of the graph $\tilde{G}_k(S)$ is set to $n$. Then, during the enumeration of the intervals, the lines 12–13 reduce this number when fusible edges in the graph are detected. The "best order" graph is updated in lines 18–20 ($m^*$ is the number of nodes of the "best order" graph). Finally, the marking of node boundaries for the next order is performed in the lines 21–22.

The algorithm contains an optimization speeding up the worst-case run-time: In the lines 16–17, it is checked if the queue $Q$ contains more than $m^*$ new k-mer intervals that were created during the current iteration. In this case, the algorithm terminates immediately, which can be explained as follows: if $Q$ contains more than $m^*$ new k-mer intervals, the graph of the current order $k$ contains more than $m^*$ distinct multi-edges $(x, y)^m$. Even when all distinct multi-edges can be fused, this means that the edge-reduced graph $\tilde{G}_k(S)$ contains at least $m^*$ edges, so no new minimum is built in the lines 18–20. Additionally, the graph $\tilde{G}_{k+1}(S)$ contains at least $m^*$ nodes. As the number of nodes increases for increasing $k$, this means that all of the graphs $\tilde{G}_{k+1}(S), \dots, \tilde{G}_n(S)$ contain at least $m^*$ nodes. As each node must have at least one outgoing edge, all of those graphs must have at least $m^*$ edges, so we don't need to inspect them anymore.

The run-time of Algorithm 5.3 can be determined by counting the overall number of k-mer intervals that are pushed to the queue $Q$ in line 15. The run-time per element consists of generation (line 11, 14 and 15), marking of the left boundary (line 21–22) and processing (line 9 and 10). While the run-time for both the later mentioned stages is constant, the generation requires $O(\log \sigma)$ time, so a single element in $Q$ takes $O(\log \sigma)$ time using a balanced wavelet tree. Lines 16–17 ensure that $Q$ contains at most $m^*$ k-mer intervals in a fixed order $k$. In the worst case, for each order k, the edge-reduced de Bruijn graph has the same amount of edges and nodes (for example the string $S = \mathtt{A}^n$), so Algorithm 5.3 requires at most $O(n \cdot m^* \cdot \log(\sigma))$ time. The worst-case run-time of Algorithm 5.3 can only be accepted if $m^*$ is a very small number, i.e. $m^* = O(\log n)$. This is very unlikely, as it would mean that the "best order" graph would consist of only $O(\log n)$ nodes.

To improve the run-time of the algorithm, one could assume that the dependence between the order of a graph and its number of edges is a convex function. In this case, one could test for each order $k$ if the number of edges has decreased and stop when the number increases. This seems to make sense: for small values of k, only few nodes exist and so few forking paths exist. Then, after processing the best order, the number of edges could increase because the graph contains more nodes with less edges between each other, so fusing a path does not reduce the overall edge count that much. In many cases this is true. Unfortunately, as Figure 5.6 shows there are cases in which the dependency contains multiple local minima, e.g. for the files `nci` or `influenza`.

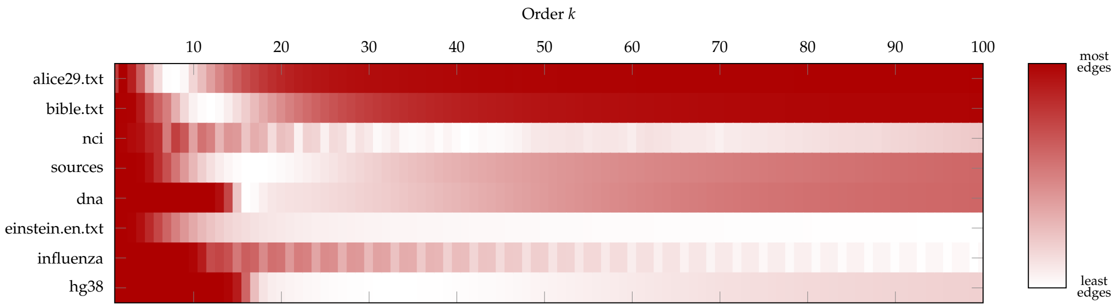


**Figure 5.6:** Dependence between the order $k$ and the number of edges in an edge-reduced de Bruijn graph for selected files from the test data set (see Chapter C). The best order for most files is typically in the range of $[5, 41]$ and varies the bigger and more repetitive the files get. A similar image was already published in the full version of [Bai+20] © 2020 IEEE.

### 5.2.1 *Incremental algorithm*

Another idea which was presented in [Bai+20] is to define an incremental algorithm which updates the edge count every time the order is increased. We herein will present this algorithm, but beforehand need some knowledge about "node evolution". Node evolution describes the way in which nodes of a de Bruijn graph change when increasing its order.

For any node in a de Bruijn graph, there exist three different kinds of "evolution steps". The first one is its birth, that is, the node is initially created. An example of such a birth is given by the node T in the de Bruijn graph of order 1 in Figure 5.5. The second kind of evolution step is aging, which means that the label of a node is growing during order increase, but the node itself never disappears. In Figure 5.5, the node T ages to TG ($k = 2$) and then to TGG ($k = 3$). The final evolution step of a node is its decease, which means that the node is segmented into new born nodes. In Figure 5.5, the node TGG deceases after the order $k = 3$ as it is split into the nodes TGGT and TGG\$. The full evolutionary process of the running example can be seen in Figure 5.7.

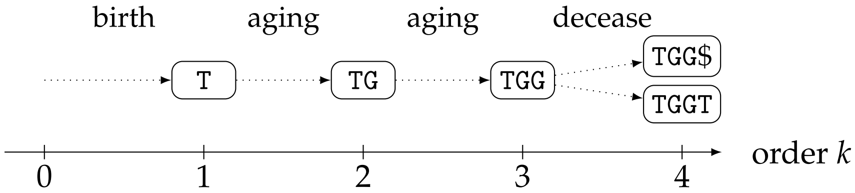


**Figure 5.7:** Node evolution steps of the node T from Figure 5.5.

Besides the trivial observation that the decease of a node means the birth of new nodes, one can see that a node deceases if and only if it has multiple successors in the current order. Let $x$ be a node with only one successor $y$. If the order of the graph is increased, the label of $x$ is extended by the last character of $y$ because of the overlapping of $x$ and $y$ by $k-1$ characters. If a node $x$ has multiple successors $y_1, \ldots, y_d$, it inherits the characters $y_1[k], \ldots, y_d[k]$. As those characters must be distinct, new nodes $xy_1[k], \ldots, xy_d[k]$ are set up in the next higher order, which indicates the decease of node $x$. See Figure 5.8 for an example.

Before we discuss the connections between edges and node evolution, we show a central result: all successors of a deceasing node must have been born immediately before the current order.

**Remark 5.7.** Let $k$ be an order of a de Bruijn graph, and let $x$ be a node with multiple successors $y_1, \ldots, y_d$. Then, the nodes $y_1, \ldots, y_d$ must have been born between order $k$ and $k-1$.

*Proof.* Because the nodes $y_1, \ldots, y_d$ are successors of $x$, they all must share the prefix $x[2..k]$. Therefore, at order $k-1$ there must have been a node with label $x[2..k]$ which deceased between order $k-1$ and $k$. The decease of this node then induces the birth of the nodes $y_1, \ldots, y_d$, so $y_1, \ldots, y_d$ must have been born between order $k$ and $k-1$. □

The first consequence of Remark 5.7 is that newly born nodes of an order $k$ induce all the newly born nodes of order $k+1$. Algorithmically, we can use this as follows: say we have a queue $Q$ in which all new nodes of order $k$ are maintained. Additionally, we need a bit-vector $B$ of size $n$ which contains the markings of node boundaries from order $k$, similar to Algorithm 5.3. New nodes of order $k+1$ can then be obtained by inspecting all $c\omega$-intervals $[lb, rb]$ of all nodes $[i, j] \in Q$: If either $B[lb]$ or $B[rb+1]$ is set to zero, the $[lb, rb]$-interval corresponds to a new node, so we can put it to the back of queue $Q$ for the next iteration, similar to lines 9–11 and 14–15 of Algorithm 5.3. Moreover, to obtain the node boundaries of order $k+1$ in the bit-vector $B$, we can enumerate all new nodes and mark their right boundary, see lines 21–22 of Algorithm 5.3.

The second consequence of Remark 5.7 deals with inherited edge fusions. Let $y$ be a node in the de Bruijn graph of order $k$ which aged between order $k-1$ and $k$. Then, its order-$k$ predecessors $x_1, \ldots, x_d$ remain unchanged (i.e. they are also aging) during the whole of its remaining lifetime. This must hold true because otherwise, new

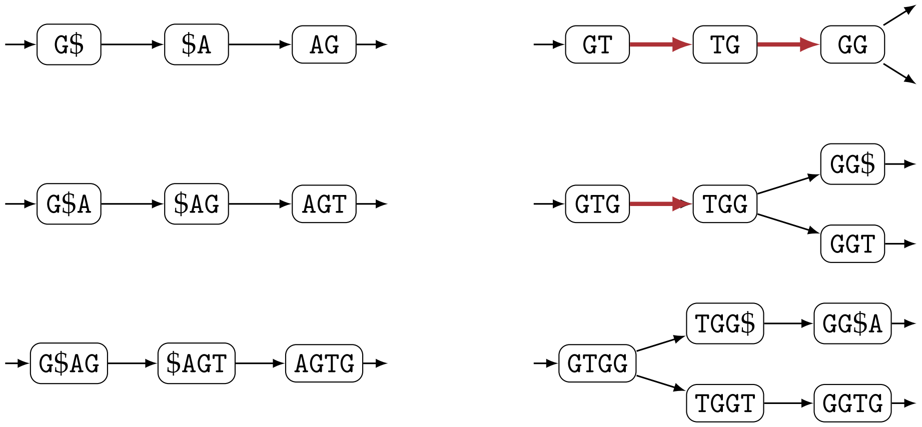


**Figure 5.8:** Node evolution during increase of the de Bruijn graph order. The left-hand side shows an extract of the graphs with order $k = 2, 3$ and 4 from Figure 5.5: as long as a node has only a single successor, increasing the order only causes the node labels to grow. The right-hand side shows an other extract of the graphs with order $k = 2, 3$ and 4 from Figure 5.5: once a node has multiple successors, it is split into new nodes in the next higher order.

nodes would be created with the existence of an aging successor, which contradicts Remark 5.7. Therefore, neither the predecessors of node $x$ nor the successors of the nodes $x_1, \ldots, x_d$ change, implying that fused edges as well as normal multi-edges are inherited in such cases. This allows us to update the edge count of an edge-reduced de Bruijn graph while increasing the order by inspecting only a couple of nodes, namely new nodes and deceasing nodes.

For a new node, two cases for edge fusions are possible. In the first case, the new node $y$ has only one predecessor $x$, and this predecessor $x$ has only the successor $y$. As this is the definition of a fusible path, we decrease the edge count by the the size of the interval minus one, see also lines 12–13 of Algorithm 5.3. In the second case, a new node $y$ has only one predecessor $x$, but this predecessor has multiple successors and thus deceases after the current order. If $y$ ages after the current order, according to the above discussion, we know that a new child of $x$ would be the only predecessor of $y$ and $y$ would be the only successor of the new child of $x$. We therefore treat $y$ as aging, although we don't know if it will be aging, and possibly revert the edge fusion later. Also, we store the amount of fused edges in a separate variable, which after the current order is used to reduce the edge count of the next order. To detect fused edges we mark each target of a fused edge in a bit-vector $F$.

The handling of deceased nodes uses the bit-vector $F$ to increase the edge count again. According to Remark 5.7, we know that a deceasing node is fully replaced by

```
Data: BWT $\mathsf{L}$ in form of an FM-index for a string $S$ of length $n$.
Result: Order $k^*$ such that the edge-reduced de Bruijn graph $\tilde{G}_{k^*}(S)$ has the minimal amount of edges.

initialize a bit-vector $B$ of size $n+1$ filled with zeros
initialize a bit-vector $F$ of size $n$ filled with zeros
initialize an empty queue $Q$
push element $[1,n]$ on $Q$
$B[1] \leftarrow 1$            $B[n+1] \leftarrow 1$
$k^* \leftarrow 1$             $m^* \leftarrow n$
$n_{\tilde{G}} \leftarrow 1$   $m \leftarrow n$
for $k \leftarrow 0$ to $n$ do
                                        // possibly fusible edges in next higher order
    $fusible \leftarrow 0$
    for $q \leftarrow |Q| - 1$ to $0$ do
        $[i,j] \leftarrow$ first element of $Q$
        pop first element of $Q$
        $M \leftarrow \mathsf{getIntervals}_{\mathsf{L}}(i,j)$
                                        // check for fusible edges
        if $\langle c, lb, rb \rangle$ is the only element in $M$ then
            if $B[lb] = 1$ and $B[rb+1] = 1$ then
                $m \leftarrow m - (rb - lb)$
            else
                $fusible \leftarrow fusible + (rb - lb)$
            $F[rb] \leftarrow 1$
                                        // store new nodes
        foreach $\langle c, [lb, rb] \rangle \in M$ do
            if $B[lb] = 0$ or $B[rb+1] = 0$ then
                push $[lb, rb]$ on the back of $Q$
                // check if next order has too many nodes
                if $B[rb+1] = 0$ then
                    $n_{\tilde{G}} \leftarrow n_{\tilde{G}} + 1$
                    if $n_{\tilde{G}} \geq m^*$ then
                        return $k^*$
                                        // check for new minimum
    if $m < m^*$ then
        $k^* \leftarrow k$
        $m^* \leftarrow m$
                                        // establish possible fusions of next order and revert old fusions
    $m \leftarrow m - fusible$
    foreach $[lb, rb] \in Q$ do
        if $F[rb] = 1$ then
            // find left bound of old k-mer interval
            $last \leftarrow \mathsf{select}_B(1, \mathsf{rank}_B(1, lb))$
            $m \leftarrow m + (rb - last)$
            $F[rb] = 0$
                                        // mark node boundaries
    foreach $[lb, rb] \in Q$ do
        $B[rb+1] = 1$
return $k^*$
```

**Algorithm 5.4:** Incremental de Bruijn graph edge minimization algorithm. A similar algorithm was already published in the full version of [Bai+20] © 2020 IEEE.

new nodes. Therefore we can use the new nodes in the queue $Q$ to check markings in the bit-vector $F$, determine the node boundaries of the deceasing node and increase the edge count again. Within this step we can also revert falsely assumed fusions of the next order, as the target node of such a falsely assumed fusion was marked in the bit-vector $F$.

The incremental (but not yet efficient) procedure is shown in Algorithm 5.4. We want to describe some more technical details now but also show why the algorithm is not yet efficient. As the queue $Q$ no longer contains all nodes of the current order, a variable $n_{\tilde{G}}$ now stores the number of nodes in the current graph. Moreover, as the edge count is updated after every iteration, a variable $m$ stores the current amount of edges in the graph.

Line 9 uses a variable $fusible$ to sum up possible fusions of the next higher order. The lines 11–13 generate new nodes, similar to lines 9–11 of Algorithm 5.3. The detection of fusible edges then works similar to the one in Algorithm 5.3, except in the special case that edges can be fused in the next higher order (line 18). Each fusion, or possible fusion, is marked in $F$ by setting the right boundary of the edge target node to 1 (line 19).

Lines 20–26 are used to determine the node count and newly born nodes in the next higher order. As already indicated, if $B[lb] = 0$ or $B[rb + 1] = 0$ holds, a node of the current order deceases and a new node is born (lines 21–22). Because a deceased node is replaced by a couple of new nodes, the node count $n_{\tilde{G}}$ is increased for all except of the lexicographically greatest new node, as this new node can be seen as a replacement of the old one (lines 23–24). The termination of the algorithm in line 26 in case of too many nodes is similar to that of Algorithm 5.3.

At line 27, it is checked if a new minimum of edges is discovered. Then, lines 30–35 are used to establish possible fusions of the next higher order to revert fusions of deceasing nodes. Similar to the increase of the number of nodes in lines 23–26, fused edges are removed only once per deceasing node using the lexicographically largest new replacing node (line 32). For the determination of the left boundary of the deceasing node using the bit-vector $B$ (line 32), the edge count is increased and the marking of fused edges in the bit-vector $F$ are cleared. This also reverses falsely fused paths as their target node was marked in line 19 of the algorithm. Finally, the algorithm marks all right node boundaries of new nodes in lines 36–37 and thereby updates the bit-vector $B$ for the next higher order.

The main problem of Algorithm 5.4 is the determination of the left boundaries of deceasing nodes (line 33). We cannot use fast rank- and select-queries from Section 2.3 because the bit-vector $B$ changes in every iteration. Rebuilding such support structures after each iteration would end up in a worst-case run-time of $O(n^2 \log \sigma)$ which would not be an improvement.

We will fix this issue soon but beforehand want to discuss the worst-case run-time of the algorithm under the assumption that line 33 can be executed in constant time. Apart from the $O(n)$ bit-vector initializations in lines 1 and 2, the worst-case run-time of Algorithm 5.4 depends on the number $z$ of generated intervals using the getIntervals-function in line 13 during the whole algorithm execution. The run-time of the remaining inner loops then depends only on this number (lines 20–26, $O(z)$) or to a subset of the generated intervals (lines 31–35 as well as lines 36–37, $O(z)$). From the preliminaries (Algorithm 2.5) we know that one call of the getIntervals-function takes $O(z' \log \sigma)$ time where $z'$ is the size of the output set, so the overall run-time of the outer for loop in lines 8–37 is limited by $O(z \log \sigma)$.

To find an upper bound for the value of $z$, we divide the generated intervals into three categories. The first category consists of newly born nodes which are not the lexicographically largest replacing nodes, i.e. the conditions from line 21 and 23 are both true. Every time such a node is created, the node count number $n_{\tilde{G}}$ is incremented, but as the algorithm terminates as soon as $n_{\tilde{G}} \geq m^*$ holds, the number of such intervals is given by $m^*$.

The second category of intervals consists of newly born nodes which are also the lexicographically largest replacing nodes. This means that the condition in line 21 is true, while the condition in line 23 is false. Each such node is a direct replacement of a deceasing node, so the number of such nodes is identical to the number of deceased nodes throughout the execution of the algorithm. It is useful to think about a "family tree" of nodes during execution: each node in the family tree corresponds to the lifetime of a node during the algorithm execution. In such a family tree, each inner node has at least two children because a deceased node is split into multiple new nodes. Also, the family tree has exactly $m^*$ children as the algorithm terminates as soon as $m^*$ nodes exist. The number of deceased nodes is thus identical to the number of inner nodes in this tree, which can be bound by $m^* - 1$. Therefore, the second category of intervals is also bound by $m^*$.

The third category of intervals consists of intervals which do not correspond to newly born nodes, i.e. the condition in line 21 is false. This means that an aging node has a new node as a successor, but the successor does not cause the aging node to decease. It is possible that an aging node has multiple new successors without being deceased. To count the number of intervals from the third category, we introduce a concept called "aging predecessorship". Suppose a new node was born at the

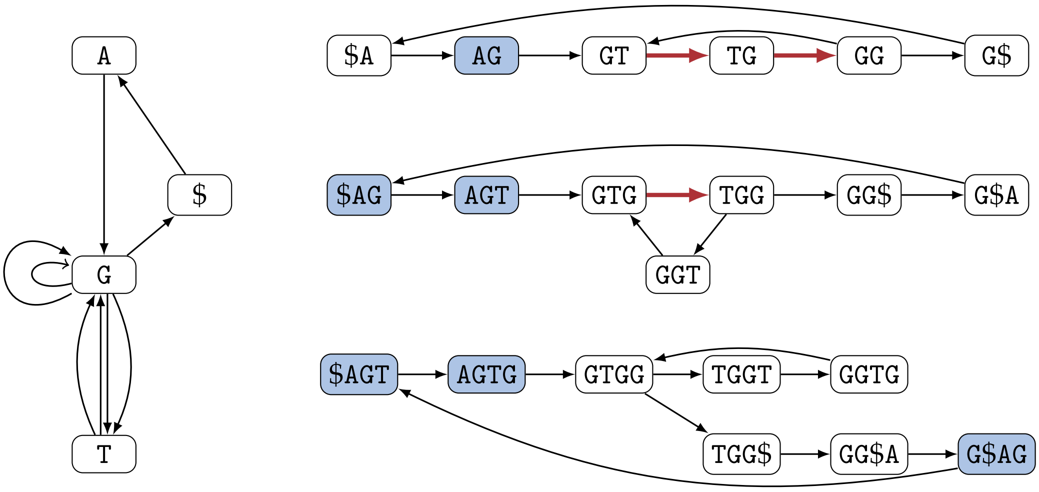


**Figure 5.9:** Aging predecessorship (blue nodes) of the A-node during aging. After the first aging, the predecessorship consists of the node itself. Then, every time the node ages, the predecessorship is extended by the predecessors of the newest members. A similar image was already published in [Bai+20] © 2020 IEEE.

order $k-1$, and then aged to the order $k$. In this case, the aging predecessorship of the node is given by the node itself. Then, if the node ages again during order $k$ and $k+1$, all of its predecessors must age too, because otherwise, we would have a contradiction to Remark 5.7. We thus add the predecessors of the node to its aging predecessorship. If the node ages again, for the same reason as before, the predecessors of the predecessors of the node must age, so we add the predecessors of the predecessors of the node to its aging predecessorship in order $k+1$, and so on.

Figure 5.9 shows an example of such an aging predecessorship. We now want to discuss some of its properties. First, when increasing the order, the aging predecessorship of a node is increased by at least one node because of the inductive application of Remark 5.7. Second, all of the nodes in the predecessorship have only aging successors, as they would decease otherwise. This also implies that nodes in the predecessorship must be non-related nodes in the family tree of nodes. Moreover, at some fixed order $k$, the predecessorship of an older node is either disjoint or a superset of a predecessorship of a younger node.

Using the aging predecessorship, we now distribute the number of cases of the third category as follows: each time the case occurs for a node, we count the case towards any of the newest members of the aging predecessorship of the node. As the predecessorship of a node grows during order increase, and this case can happen only once per order for a certain node, there can be no node in the predecessorship to which this case applies twice. When an old node deceases, the predecessorship of all of its former predecessors increases again, so if the case applies to some of those predecessors, the same case distribution strategy ensures that no node in the family tree receives a case twice. In summary, the intervals of the third category can be distributed to nodes of the family tree such that no case applies to node twice. This means that the intervals of the third category are bounded by $2m^*$.

To summarize, the number of generated intervals of all three categories is given by $O(m^*)$. We shortly sum up the current result in the following corollary.

**Corollary 5.8.** *Given a BWT* $\mathsf{L}$ *of a string* $S$ *of length* $n$ *in form of an FM-index, Algorithm 5.4 solves the de Bruijn graph edge minimization problem. Assuming that line 33 of Algorithm 5.4 can be implemented to run in* $O(1)$ *time, the algorithm requires*

- $O(n)$ *time for the initialization of the bit-vectors* $B$ *and* $F$*.*
- $O(m^* \log \sigma)$ *time for computing the best order* $k^*$*, where* $m^*$ *is the number of edges in the edge-reduced de Bruijn graph* $\tilde{G}_{k^*}(S)$*.*

### 5.2.2 *Efficient incremental algorithm*

As noted in Corollary 5.8, the main problem of our current incremental algorithm is finding the left boundary of a deceasing node in order to revert old fusions properly. A solution to this problem is to traverse the intervals in $Q$ in lexicographical order, i.e. by sorting the intervals in $Q$ according to their left boundary. When we access the interval of a deceasing node for the first time ($B[rb+1] = 0$ and $last = \perp$ for the enumerated $[lb, rb]$ interval), we set a variable $last$ to the left boundary $lb$, as this corresponds to the left boundary of a deceasing node. Then, if we access the interval for the last time ($B[rb+1] = 1$), the interval of the deceasing node is determined. To detect the left boundary of the next deceasing node, we set $last$ to $\perp$ again.

The enumeration of intervals from new nodes in lexicographical order can be performed similar to a top-down traversal of intervals in an lcp-interval-tree presented by Beller et al. [BBO12], see also [Ohl13, pp. 323–328]. The idea is to use $\sigma$ queues $Q_c$ instead of only one queue $Q$, where each queue $Q_c$ contains only intervals of nodes whose first character is $c$. Let us assume that the intervals in each queue $Q_c$ are lexicographically sorted. Trivially, this is true when only one interval is used at the beginning of Algorithm 5.5 (line 5). The enumeration of all characters $c \in \Sigma$

in ascending order (lines 13 and 37) and for each $c$ enumerating all intervals in the queue $Q_c$ from front to back (lines 15–16 and 38) then simulates the enumeration of all intervals in lexicographical order.

When generating new intervals for the next order, we have to ensure that the new intervals do not conflict with intervals of the current order, and are also lexicographically sorted. To avoid conflicts of new and old intervals in Algorithm 5.5, we use the same trick as in Algorithm 5.4: we store the size of each queue $Q_c$ in a variable $q_c$ (line 12) before enumerating only the first $q_c$ intervals in each queue $Q_c$ (line 14). As new intervals are pushed to the back of a queue $Q_c$ (line 26), this ensures a conflict-free interval handling. To ensure that the newly generated $c\omega$ - intervals are lexicographically sorted, we note that due to the enumeration of $w$-intervals in lexicographic order, a newly generated interval $c\omega$ is pushed to the back of $Q_c$ before another $c\tilde{\omega}$-interval iff $\omega$ is lexicographically smaller than $\tilde{\omega}$. This induces that the newly generated intervals are lexicographically sorted inside of each queue $Q_c$, completing the induction.

The previously mentioned trick of storing the left boundary of a deceasing node the first time it is entered (line 43), of not modifying the left boundary for all intermediate accesses (line 42) and of obtaining the deceasing node boundaries when the interval is accessed for the last time (lines 44–48) originally comes from Beller et al. [BBO12]. Furthermore, Algorithm 5.5 combines the establishment of new fusions, the removal of old fusions as well as the marking of node boundaries for the next higher order in the loop of the lines 37–48. Using those modifications, the main loop of Algorithm 5.5 requires $O(m^* \log \sigma)$ time.

Next, the blue marked lines of Algorithm 5.5 will be discussed. Without the blue lines, the algorithm solves the de Bruijn graph edge minimization problem already. The problem itself only asks for the best order $k^*$, but for the purpose of tunneling, we need a node boundary bit-vector $B$, see Algorithm 5.2. We could use the order $k^*$ to generate such a bit-vector using Algorithm 5.1, but this takes additional computational effort which can be avoided. Suppose we are in an order $k$ in Algorithm 5.5 in which a new minimum of edges is detected. Then, immediately before line 31, the used bit-vector $B$ corresponds to the required node boundary bit-vector. However, as the algorithm proceeds, additional bits are set in the bit-vector.

Obviously, the best order $k^*$ is found when the algorithm detects a new minimal amount of edges (line 31) for the last time during its execution. The related node boundary bit-vector can be retained by storing all positions where a new bit in $B$ is set (line 41) and by clearing all such bits in the loop in lines 49–50 afterwards. As

```
Data: BWT L in form of an FM-index for some string $S$ of length $n$.
Result: Order $k^*$ and node boundary bit-vector $B$ of the edge-reduced DBG $\tilde{G}_{k^*}(S)$ with minimal amount of edges.
initialize a bit-vector $B$ of size $n+1$ filled with zeros
initialize a bit-vector $F$ of size $n$ filled with zeros
initialize an external buffer buf
initialize $\sigma$ empty queues $Q_c$
push element $[1,n]$ on an arbitrary queue $Q_c$
$B[1] \leftarrow 1$          $B[n+1] \leftarrow 1$
$k^* \leftarrow 1$           $m^* \leftarrow n$
$n_{\tilde{G}} \leftarrow 1$           $m \leftarrow n$
for $k \leftarrow 0$ to $n$ do
    $fusible \leftarrow 0$
    foreach $c \in \Sigma$ in ascending order do
        $q_c \leftarrow |Q_c|$
    foreach $c \in \Sigma$ in ascending order do
        for $q \leftarrow q_c - 1$ to $0$ do
            $[i,j] \leftarrow$ first element of $Q_c$
            pop first element of $Q_c$
            $M \leftarrow \mathsf{getIntervals}_{\mathsf{L}}(i,j)$
            if $\langle \tilde{c}, lb, rb \rangle$ is the only element in $M$ then
                if $B[lb] = 1$ and $B[rb+1] = 1$ then
                    $m \leftarrow m - (rb - lb)$
                else
                    $fusible \leftarrow fusible + (rb - lb)$
                $F[rb] \leftarrow 1$
            foreach $\langle \tilde{c}, [lb, rb] \rangle \in M$ do
                if $B[lb] = 0$ or $B[rb+1] = 0$ then
                    push $[lb, rb]$ on the back of $Q_{\tilde{c}}$
                    if $B[rb+1] = 0$ then
                        $n_{\tilde{G}} \leftarrow n_{\tilde{G}} + 1$
                        if $n_{\tilde{G}} \geq m^*$ then
                            break outermost for-loop
    if $m < m^*$ then
        $k^* \leftarrow k$
        $m^* \leftarrow m$
        clear the external buffer buf
    $m \leftarrow m - fusible$
    $last \leftarrow \perp$
    foreach $c \in \Sigma$ in ascending order do
        foreach $[lb, rb] \in Q_c$ from front to back do
            if $B[rb+1] = 0$ then
                $B[rb+1] \leftarrow 1$
                add $rb+1$ to the external buffer buf
                if $last = \perp$ then
                    $last \leftarrow lb$
            else
                if $F[rb] = 1$ then
                    $m \leftarrow m + (rb - last)$
                    $F[rb] \leftarrow 0$
                $last \leftarrow \perp$
foreach $lb \in$ buf do
    $B[lb] \leftarrow 0$
return $\langle k^*, B \rangle$
```

**Algorithm 5.5:** Efficient incremental de Bruijn graph edge minimization algorithm. A similar algorithm was presented in [Bai+20], this version was adapted from [Web20].

we do not know how often a new minimal amount of edges is found, we clear the buffer after each new minimal amount (line 34) to ensure that only positions after the last optimum are stored in the buffer. The maximal number of positions stored in the buffer corresponds to the maximal number of new nodes in the graph, so the buffer will contain at most $2 \cdot m^*$ positions. This shows that the loop in line 49–50 requires $O(m^*)$ time. Furthermore, clearing the buffer in line 34 can be done in $O(1)$ time by setting the output pointer of the buffer to the first position again. Thus, we can retain the node boundary bit-vector $B$ of the best order $k^*$ without increasing the run-time of Algorithm 5.5.

**Corollary 5.9.** *Given a string $S$ of length $n$, the de Bruijn graph edge minimization problem can be solved in $O(n \log \sigma)$ time using the following steps:*

1. *Computation of the BWT $\mathsf{L}$ of $S$ in $O(n)$ time.*
2. *Construction of the FM-index (wavelet tree of $\mathsf{L}$) in $O(n \log \sigma)$ time.*
3. *Initialization of two bit-vectors $B$ and $F$ in $O(n)$ time (line 1 and 2 of Algorithm 5.5).*
4. *Execution of lines 3–51 of Algorithm 5.5 in $O(m^* \log \sigma)$, where $m^*$ is the number of edges in the edge-reduced de Bruijn graph with the minimal amount of edges.*

*Using the FM-index and the resulting node boundary bit-vector $B$ of Algorithm 5.5 as input of Algorithm 5.2, we can construct a length-minimal tunneled BWT of $S$ in terms of $k$-mer prefix intervals in $O(n \log \sigma)$ time.*

### 5.2.3 *Experimental results*

We implemented Algorithms 5.5 and 5.2 using `C++` and the `sdsl-lite` library [Gog07]. The combination of both algorithms enhanced with wavelet tree construction for the tunneled BWT can be seen as the construction of a tunneled FM-index. The programs are publicly available [Bai20], information about the used test data and the full benchmark results can be found in Appendix A and C.1.

We tested the algorithms on 44 files overall, categorized into 6 text corpora. The average result per corpus can be found in Figure 5.10. For small-sized files (canterbury and largecanterbury, < 5 MB) the edge-reduction rate is too small to compensate the additional overhead in the tunneled FM-index, that is, the two additional bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$. Files with medium and big size (silesia and pizzachili, 5 MB – 2 GB)

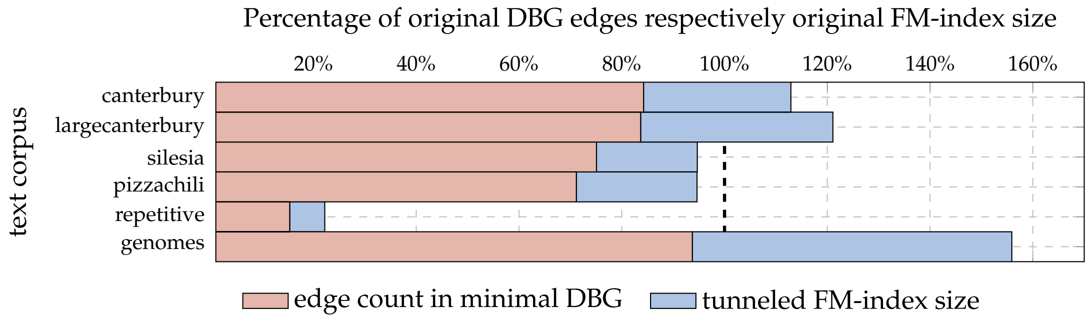


**Figure 5.10:** Average amount of reduced edges for minimal edge-reduced de Bruijn graphs as well as size of corresponding tunneled FM index compared to a normal FM-index. The values shown are averages over a whole text corpus; see Appendix A and C.1 for more details. A similar image was already published in the full version of [Bai+20] © 2020 IEEE.

achieve a good reduction rate, so the tunneled FM-index is favorable when being compared to a normal FM-index in terms of size. The best results are achieved for repetitive files, where the size of an FM-index could be reduced by 80 % on average using tunneling. Full genome sequences with about 3 GB achieved the worst result: because of the small alphabet size of 4, a character in an FM-index roughly requires 2 bits. Now, as the edge reduction rate in genomes is low and a tunneled FM-index requires 2 additional bits per character, the size of a tunneled FM-index almost doubles compared to a normal FM-index.

The construction of a tunneled FM-index requires about twice the amount of time needed to construct the normal FM-index. The memory peak during construction is dominated by the construction of the suffix array, see Appendix C.1.

## 5.3 TRIE REPRESENTATION USING THE EXTENDED BWT

This section will present a succinct representation of extended tries. We already introduced tries as tree-like dictionaries for multiple strings in the principles chapter, see Section 2.4. Within the same section, a way to represent tries using Wheeler graphs was also presented. This enables us to represent tries using the succinct representation of Wheeler graphs. However, to perform exact multi-string matching using the Aho-Corasick Algorithm [AC75], tries have to be extended by a few components, which is given in the following definition.

**Definition 5.10** (Extended trie)**.** Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a set of null-terminated strings. The extended trie $\mathcal{T}_{\text{ex}}(S_1, .., S_m) = (V, E)$ is a directed rooted tree with the following components:

- A label function $\lambda : E \to \Sigma$ assigning a label to each edge.
- A node label function $\lambda : V \to \Sigma^*$ assigning a unique label to each node.
- A failure link function $\phi : \{\ v \in V \ \mid\ v \text{ is no leaf and not the root }\ \} \to V$ linking an inner node to its longest proper suffix node.
- A record function $\rho : \{\ v \in V \ \mid\ v \text{ is a leaf }\} \to [1, m]$ assigning each leaf the number of its corresponding record.

The components fulfill the following conditions:

- No two outgoing edges of a node $u$ have the same label: let $u, v, w \in V$ be nodes with $(u, v), (u, w) \in E$. Then, $\lambda((u, v) \neq (u, w) \Leftrightarrow v \neq w$.
- Let $u_1, u_2, \ldots, u_i$ be the nodes visited on the unique path from the root node to a node $v \in V$. Then, $\lambda(v) := \lambda(u_1, u_2) \cdots \lambda(u_{i-1}, u_i)$.
- $\phi(u) := \text{argmax}_{v \in V}\{\ |\lambda(v)| \ \mid\ \lambda(v) \text{ is a proper suffix of } \lambda(u)\ \}$.
- A leaf $v \in V$ with $\rho(v) = i$ fulfills $\lambda(v) = S_i$.

An example of an extended trie can be found in Figure 5.11. The basic tree shape, label function $\lambda$ and node label function $\lambda$ are equivalent to normal tries as described in the principles, see Definition 2.18. It should be noted that the node label function $\lambda$ is implicitly given by traversing the unique path from the root node to a desired node $v$.

While the record function immediately seems to be useful because it maps a leaf to the corresponding record, the purpose of failure links is apparent only at second glance. Failure links were devised by Alfred V. Aho and Margaret J. Corasick [AC75] and are an analogy to the jump table of the Knuth Morris Pratt string matching algorithm [KMP77]. Suppose we have a string $S = \texttt{ACGTGGA}$ and want to find all occurrences of the strings from Figure 5.11. We start to match the three first characters of $S$ with the labels of nodes in the trie and end up in the leftmost node $v$ at the fourth level, i.e. $\lambda(v) = \texttt{ACG}$. When trying to match the next character we see that $S[4] = \texttt{T}$ and the outgoing edge with label $\texttt{A}$ are not equal, so we are no longer able

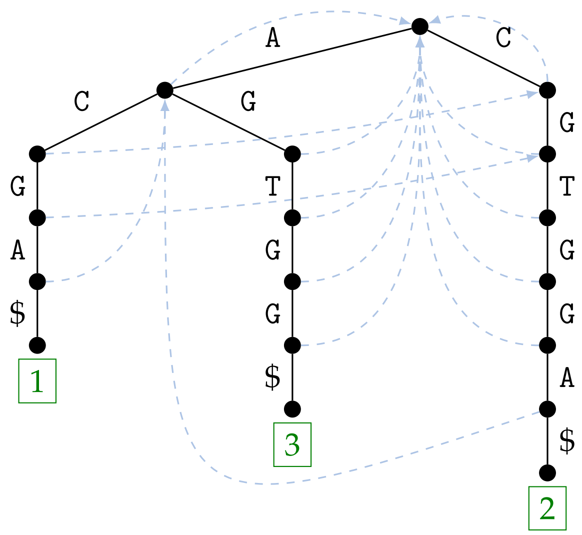


**Figure 5.11:** Extended trie $\mathcal{T}_{\text{ex}}(S_1, S_2, S_3)$ of the strings $S_1 = \texttt{ACGA\$}$, $S_2 = \texttt{CGTGGA\$}$ and $S_3 = \texttt{AGTGG\$}$. The dashed blue lines indicate failure links, green boxes indicate records of the corresponding leaves.

to extend the match. A naive approach would be to start at the second character of $S$ and the root of the trie again.

Instead, a more clever approach is to see that we can proceed with matching at the fourth character of $S$ against the node $v$ with $\lambda(v) = \texttt{CG}$. As the string ACG has already been matched, we do not need to match the characters CG again. Instead of this, we start at the node CG, and try to proceed with the matching. Generally speaking, for a given node $u$ we want to have a link to a node $v$ such that $\lambda(v)$ is the longest proper suffix of $\lambda(u)$. In the case of a mismatch (or failure), we follow the link and thereby prevent unnecessary matches. As the name already suggests, failure links fulfill exactly this task.

Let $S$ be a string of length $n$, and let $P_1, \ldots, P_m$ be non-null-terminated patterns such that no pattern is a substring of another pattern. Given an extended trie $\mathcal{T}_{\text{ex}}(P_1\$, \ldots, P_m\$)$, it is possible to solve the exact multi-string matching problem in $O(n)$ time, see Algorithm 5.6 and [Ohl13, pp. 27].

It is also possible to solve the exact multi-string pattern problem for arbitrary patterns $P_1, \ldots, P_m$ not fulfilling any other prerequisite. The basic idea is to replace lines 4–5 of Algorithm 5.6 with a loop that iterates over all nodes on the unique path of failure links from the current node to the root. On this path, each node that has an outgoing edge labeled with \$ corresponds to a match. The problem of such an approach is that the path of failure links from a node to the root can contain up to

```
Data: Extended trie $\mathcal{T}_{\text{ex}}(P_1\$,\dots,P_m\$)$ such that no non-null-terminated pattern $P_i$ is a substring of another pattern
      $P_j$, string $S$ of length $n$.
Result: All occurrences of the patterns $P_1,\dots,P_m$ in the string $S$.

$i \leftarrow 1$
$v \leftarrow$ root of $\mathcal{T}_{\text{ex}}(S_1,\dots,S_m)$
while $i \leq n$ do
    if there exists an edge $(v,v')$ with $\lambda((v,v')) = \$$ then
        report that pattern $P_{\rho(v')}$ occurs in $S$ ending at position $i$
    if there exists an edge $(v,v')$ with $\lambda((v,v')) = S[i]$ then
        $v \leftarrow v'$
        $i \leftarrow i+1$
    else
        if $v =$ root of $\mathcal{T}_{\text{ex}}(S_1,\dots,S_m)$ then
            $i \leftarrow i+1$
        else
            $v \leftarrow \phi(v)$
```

**Algorithm 5.6:** Aho-Corasick algorithm [AC75] for exact multi-string matching using an extended trie $\mathcal{T}_{\text{ex}}(P_1\$,\dots,P_m\$)$. The algorithm works correctly if no pattern $P_i$ is a substring of another pattern $P_j$. This algorithm is a modification of an algorithm in [Ohl13, p. 27].

$n$ nodes. Thus, the worst-case run-time increases to $O(n^2)$. In [AC75] and [Ohl13, pp. 27–31] two ways to accelerate this path traversal are presented. We will present an alternative acceleration later but for now keep things simple and concentrate on the restricted case.

The extended BWT (XBWT) is a succinct representation of extended tries. The XBWT was invented by Ferragina et al. [Fer+05], but a succinct representation of failure links was missing. Later on, Manzini added such a succinct representation of failure links to the XBWT and also presented efficient algorithms for XBWT construction [Man16]. We herein present a definition of an XBWT which additionally takes the record function into account.

**Definition 5.11** (Extended BWT)**.** Let $\mathcal{T}_{\text{ex}}(S_1,\dots,S_m) = (V,E)$ be an extended trie of the null-terminated strings $S_1,\dots,S_m$ with label function $\lambda$ and record function $\rho$. Furthermore, let $\pi : [1, |V| - m] \to \{ v \in V \mid v \text{ is no leaf} \}$ be a mapping function such that $\lambda(\pi(1))^R <_{\text{lex}} \cdots <_{\text{lex}} \lambda(\pi(|V| - m))^R$.

The extended BWT of $\mathcal{T}_{\text{ex}}(S_1,\dots,S_m)$ consists of the following components:

- A bit-vector $D_{\text{out}}$ saving the outdegrees of all inner nodes using reverse unary coding, that is, $D_{\text{out}} := 10^{\deg_{\text{out}}(\pi(1))-1}10^{\deg_{\text{out}}(\pi(2))-1}\dots10^{\deg_{\text{out}}(\pi(|V|-m))-1}1$.
- A string $\mathsf{L}_X$ storing the labels of outgoing edges of all inner nodes: for an inner node $u \in V$ with $\pi(i) = u$, the following holds true:

$\{ \mathsf{L}_X[k] \mid \mathsf{select}_{D_{\mathsf{out}}}(1,i) \leq k < \mathsf{select}_{D_{\mathsf{out}}}(1,i+1) \} = \{ \lambda((u,v)) \mid (u,v) \in E \}$.

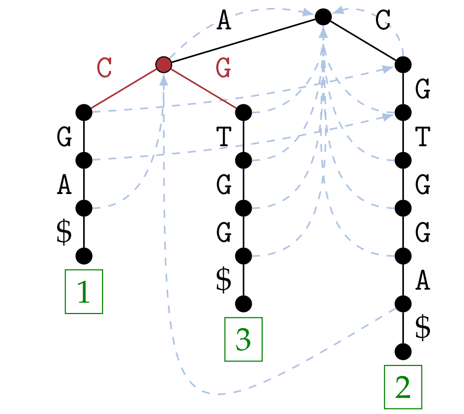
- An array $\mathsf{C}_X$ of size $\sigma$ (size of the alphabet of $S_1,\ldots,S_m$) supporting navigational operations: $\mathsf{C}_X[c] := \mathsf{C}_{\mathsf{L}_X}[c] - (m-1)$.
- A balanced parentheses sequence $P$ of length $2 \cdot (|V| - m)$ supporting failure links. The sequence $P$ fulfills the following condition: Let $i$ and $j$ be integers such that $\lambda(\pi(i))^R$ is a proper prefix of $\lambda(\pi(j))^R$, and define $i_o := \mathsf{select}_P(''('',i)$ as well as $j_o := \mathsf{select}_P(''('',j)$. Then, $P$ satisfies $i_o < j_o < \mathsf{findclose}_P(j_o) < \mathsf{findclose}_P(i_o)$.
- A record array $R$ mapping leaves to records. Let $v$ be a leaf and define $i := |\{ u \in V \mid u \text{ is a leaf with } \lambda(u)^R <_{\mathsf{lex}} \lambda(v)^R \}| + 1$ as the lexicographic rank of $v$ under all leaves. Then, $R[i] := \rho(v)$.

An example of the components of an XBWT can be found in Figure 5.12. First of all, the XBWT numbers nodes by the lexicographic rank of their reversed labels. Analogously to Section 2.4, as a BWT offers backward steps, the node labels are reversed to simulate forward steps. In our case a forward step is meant to be an edge navigation towards a child node.

The components $\mathsf{L}_X$ and $D_{\mathsf{out}}$ describe the outgoing edges of all nodes, similar to the components in the succinct representation of a Wheeler graph, see Definition 2.19. The difference to Wheeler graphs is that we do not use an additional bit-vector $D_{\mathsf{in}}$ because we declare that each node in the trie has an indegree of one. This means that the bit-vector $D_{\mathsf{in}}$ would be completely filled with ones, so a rank- and select-query is an identity operation: $\mathsf{select}_{1^n}(1,i) = i$ and $\mathsf{rank}_{1^n}(1,i) = i$. Thus it is possible to avoid the bit-vector, which saves space.

The declaration of a fixed indegree of one induces that the underlying Wheeler graph is not cyclic. On Page 2.15 we have seen that a trie can be made cyclic by moving the endpoints of edges labeled with \$ to the root node. Instead, for the XBWT, we will not follow any edge labeled with \$, as those edges only lead to leaves. We also have to adapt the C-array: virtually, only one edge points to the root node of the trie. Thus, the occurrence count of the \$ characters in $\mathsf{L}_X$ must be set to one.

Failure links are represented using a sequence of balanced parentheses. As we know that the labels of nodes in an XBWT are viewed in a reverse manner, a failure link can be simulated by finding the longest proper prefix label instead of the "normal" longest proper suffix label. In Section 2.3 it was shown that the operation of



| $c$ | \$ | A | C | G | T |
|---|---|---|---|---|---|
| $\mathsf{C}_X[c]$ | 0 | 1 | 4 | 6 | 13 |

| $i$ | $i_v$ | $i_\$$ | $\mathsf{L}_X[i]$ | $D_{\text{out}}[i]$ | $\lambda(\pi[i_v])^R$ | $R[i_\$]$ |
|---|---|---|---|---|---|---|
| 1 | 1 | | A | 1 | $\varepsilon$ | |
| 2 | | | C | 0 | | |
| 3 | 2 | | C | 1 | A | |
| 4 | | | G | 0 | | |
| 5 | 3 | 1 | \$ | 1 | AGCA | 1 |
| 6 | 4 | 2 | \$ | 1 | AGGTGC | 2 |
| 7 | 5 | | G | 1 | C | |
| 8 | 6 | | G | 1 | CA | |
| 9 | 7 | | T | 1 | GA | |
| 10 | 8 | | T | 1 | GC | |
| 11 | 9 | | A | 1 | GCA | |
| 12 | 10 | 3 | \$ | 1 | GGTGA | 3 |
| 13 | 11 | | A | 1 | GGTGC | |
| 14 | 12 | | G | 1 | GTGA | |
| 15 | 13 | | G | 1 | GTGC | |
| 16 | 14 | | G | 1 | TGA | |
| 17 | 15 | | G | 1 | TGC | |
| 18 | | | | 1 | | |

$$P = (\ (\ (\ )\ (\ )\ )\ (\ (\ )\ )\ (\ )\ (\ (\ )\ )\ (\ )\ (\ )\ (\ )\ (\ )\ (\ )\ (\ )\ )$$

**Figure 5.12:** Extended trie $\mathcal{T}_{\text{ex}}(S_1, S_2, S_3)$ of the strings $S_1$ = ACGA\$, $S_2$ = CGTGGA\$ and $S_3$ = AGTGG\$ (upper left corner) and components of an extended BWT. The red node and edges in the extended trie correspond to the red entries in $D_{\text{out}}$, $\mathsf{L}_X$ and $\lambda(\cdot)^R$. The red parentheses in $P$ also correspond to the red node.

finding the longest proper prefix of a string in a lexicographically sorted list can be performed efficiently using a balanced parentheses sequence. Thus, the sequence $P$ is a direct analogue to the succinct representation of the prefix tree shown in Figure 2.12.

The last component of the XBWT is the record array. It is a simple mapping from the \$ characters in $\mathsf{L}_X$ to the corresponding record numbers. Given that we detected some node $\pi[i]$ which contains an outgoing edge labeled \$, the corresponding record can be computed using the formula $j \leftarrow R[\mathsf{rank}_{\mathsf{L}_X}(\$, \mathsf{select}_{D_{\text{out}}}(1, i+1) - 1)]$.

To see how each component is used, we rephrased Algorithm 5.6 such that the Aho-Corasick algorithm works with the use of an XBWT. The result is shown in Algorithm 5.7. If we encode the string $\mathsf{L}_X$ with a balanced wavelet tree, add select support to the bit-vector $D_{\text{out}}$ as well as balanced parentheses support to the bit-vector $P$, Algorithm 5.7 has a worst-case run-time of $O(n \log \sigma)$.

Regarding the space consumption of representations of extended tries, one can see that the XBWT is a succinct representation. An extended trie $\mathcal{T}_{\text{ex}}(S_1, \ldots, S_m) = (V, E)$ for example can be represented by a special adjacency list. Each edge in the list then

```
Data: XBWT of an extended trie $\mathcal{T}_{\text{ex}}(P_1\$, \dots, P_m\$)$ such that no non-null-terminated pattern $P_i$ is a substring of
      another pattern $P_j$, string $S$ of length $n$. The XBWT consists of the components $D_{\text{out}}$, $\mathsf{L}_X$, $P$ and $R$.
Result: All occurrences of the patterns $P_1, \dots, P_m$ in the string $S$.

$i \leftarrow 1$
$i_v \leftarrow 1$
while $i \leq n$ do
    $lb \leftarrow \mathsf{select}_{D_{\text{out}}}(1, i_v)$
    $rb \leftarrow \mathsf{select}_{D_{\text{out}}}(1, i_v + 1) - 1$
                                        // check if node $\pi[i_v]$ has an outgoing edge labeled with $
    $r_{lb} \leftarrow \mathsf{rank}_{\mathsf{L}_X}(\$, lb - 1)$
    $r_{rb} \leftarrow \mathsf{rank}_{\mathsf{L}_X}(\$, rb)$
    if $r_{lb} < r_{rb}$ then
        report that pattern $R[r_{rb}]$ occurs in $S$ ending at position $i$
                                        // check if node $\pi[i_v]$ has an outgoing edge labeled with $S[i]$
    $r_{lb} \leftarrow \mathsf{rank}_{\mathsf{L}_X}(S[i], lb - 1)$
    $r_{rb} \leftarrow \mathsf{rank}_{\mathsf{L}_X}(S[i], rb)$
    if $r_{lb} < r_{rb}$ then
        $i_v \leftarrow \mathsf{C}_X[S[i]] + r_{rb}$
        $i \leftarrow i + 1$
    else
                                        // follow failure link if possible
        if $i_v = 1$ then
            $i \leftarrow i + 1$
        else
            $i_v \leftarrow \mathsf{rank}_P(''('', \mathsf{enclose}_P(\mathsf{select}_P(''('', i_v)))$
```

**Algorithm 5.7:** Aho-Corasick algorithm using an XBWT.

consists of the label of the edge as well as a pointer to the destination node. If an edge is labeled with \$, we can replace the pointer with the corresponding record number. Furthermore, failure links can be represented explicitly by using an additional array of pointers. These data structures require $|E| \cdot (\log \sigma + \log |V|)$ bits for the adjacency list and $(|V| - m)\log(|V| - m)$ bits for the failure links.

The XBWT instead requires $|E| \log \sigma + o(|E| \log \sigma)$ bits for the balanced wavelet tree, $(|E| + 1) + o(|E| + 1)$ bits for the bit-vector $D_{\text{out}}$ with select support and $m \log m$ bits for the record array $R$. To summarize, we can replace the adjacency list with data structures that require only $|E| \cdot (\log \sigma + 1) + o(|E| \cdot (\log \sigma + 1)) + m \log m + 1 + o(1)$ bits, which is clearly beneficial in terms of space. The explicit representation of failure links can be replaced by a bit-vector with balanced parentheses support, requiring only $2 \cdot (|V| - m) + o(|V| - m)$ bits of space. Moreover, it has been shown in practice that an XBWT among other string dictionary representations achieves the best compression to date [MP+16].

Before proceeding with the construction of an XBWT, we will present a way to remove the restriction from Algorithm 5.7 concerning the patterns. More precisely, we want to present an algorithm that lists all occurrences of patterns $P_1, \dots, P_m$ in a string $S$, even if a pattern is a proper substring of another pattern. These results have not been published yet and strengthen the good reputation of the XBWT as an extended trie representation.

**Remark 5.12** (Unrestricted Aho-Corasick algorithm with an XBWT)**.**
To report all occurrences of patterns $P_1, \ldots, P_m$ in a string $S$, Algorithm 5.7 would have to be extended as follows: at the start of each iteration of the outer while–loop, one would have to inspect the unique failure link path leading from the current node $v$ to the root node. Then, each node on this path having an outgoing edge labeled with \$ corresponds to the occurrence of a pattern, see e.g. [Ohl13, p. 28].

In an XBWT, let $i_v$ be an integer such that $\pi[i_v] = v$. The nodes $\pi[j_v]$ on the failure link path to the root corresponding to occurrences of patterns fulfill the following conditions:

1. $\lambda(\pi[j_v])^R$ is a prefix of $\lambda(\pi[i_v])^R$. Because reversed node labels in an XBWT are sorted lexicographically and the balanced parentheses sequence represents the prefix trie of the node labels, this is equivalent to check if $j_v \leq i_v$ and $\mathsf{findclose}_P(\mathsf{select}_P(''('', i_v)) \leq \mathsf{findclose}_P(\mathsf{select}_P(''('', j_v))$ holds.

2. A node $\pi[j_v]$ contains an outgoing edge labeled with \$, or equivalently, the string $\mathsf{L}_X[\mathsf{select}_{D_{\mathsf{out}}}(1, j_v)..\mathsf{select}_{D_{\mathsf{out}}}(1, j_v + 1) - 1]$ contains a \$ symbol.

The number of candidate nodes $cn$ fulfilling the "first half" of condition 1 ($j_v \leq i_v$) and condition 2 can be determined using $cn = \mathsf{rank}_{\mathsf{L}_X}(\$, \mathsf{select}_{D_{\mathsf{out}}}(1, i_v + 1) - 1)$. To filter out candidates fulfilling the "second half" of condition 1 we set up a conceptual array $PC$ of size $m$. The array contains the position of the closing parenthesis for each node which has an outgoing edge labeled \$, regarding the lexicographic order of reverse node labels.

$$PC[k] := \mathsf{findclose}_P(\mathsf{select}_P(''('', \mathsf{rank}_{D_{\mathsf{out}}}(1, \mathsf{select}_{\mathsf{L}_X}(\$, k))))$$

Given the number $cn$ from above and the position $cp = \mathsf{findclose}_P(\mathsf{select}_P(''('', i_v))$ of the closing parenthesis of node $\pi[i_v]$ in $P$, all entries $k \in [1, cn]$ in $PC$ satisfying $cp \leq PC[k]$ correspond to nodes that fulfill both conditions. Moreover, as entries in $PC$ and the record array $R$ are sorted according to the lexicographic order of the reversed node labels, the record of the match then is given by $R[k]$.

```
Data: XBWT of an extended trie $\mathcal{T}_{\mathsf{ex}}(P_1\$, \ldots, P_m\$)$, range maximum data structure of the array $PC$, index $i_v$ of the
      visited node in the Aho-Corasick algorithm, position $i$ in the string $S$.
Result: All occurrences of the patterns $P_1, \ldots, P_m$ in the string $S$ ending at the current position $i$.

$cn \leftarrow \mathsf{rank}_{\mathsf{L}_X}(\$, \mathsf{select}_{D_{\mathsf{out}}}(1, i_v + 1) - 1)$
$cp \leftarrow \mathsf{findclose}_P(\mathsf{select}_P(''('', i_v))$

let $Q$ be an empty queue of pairs
push $[1, cn]$ on $Q$
while $Q$ is not empty do
    let $[lb, rb]$ be the first element of $Q$
    remove first element of $Q$
    if $lb \leq rb$ then
        $k \leftarrow \mathsf{rmq}_{PC}(lb, rb)$
        if $\mathsf{findclose}_P(\mathsf{select}_P(''('', \mathsf{rank}_{D_{\mathsf{out}}}(1, \mathsf{select}_{\mathsf{L}_X}(\$, k)))) \geq cp$ then      // $PC[k] \geq cp$
            report that pattern $R[k]$ occurs in $S$ ending at position $i$
            push $[lb, k-1]$ to queue $Q$
            push $[k+1, rb]$ to queue $Q$
```

**Algorithm 5.8:** Reporting of all occurrences of patterns ending at position $i$ during the Aho-Corasick algorithm. The algorithm is a replacement of lines 6–9 of Algorithm 5.7 and reports all occurrences even if patterns are proper substrings of other patterns.

We thus need to determine all entries $k \in [1, cn]$ satisfying $PC[k] \geq cp$. This can be done efficiently with the help of range maximum queries. An $\mathsf{rmq}_{PC}(lb, rb)$-query computes the position of the largest value within the range $PC[lb..rb]$, that is,

$$\mathsf{rmq}_{PC}(lb, rb) := \operatorname*{argmax}_{k \in [lb, rb]} PC[k].$$

Such queries can be computed in $O(1)$ time by using special range maximum query data structures. The construction of these data structures requires linear time. Furthermore, these data structures are stand-alone which means that the underlying array is not needed to support the queries. Range maximum query data structures can be represented using only $2m + o(m)$ bits of space where $m$ is the size of the underlying array, see e.g. [Ohl13, pp. 275–276] for more details.

Coming back to the primary problem of determining all entries $k \in [1, cn]$ satisfying $PC[k] \geq cp$, we can use range maximum queries to solve the problem as follows: we start with an $\mathsf{rmq}_{PC}(1, cn)$-query and store the result in a variable $k$. If $PC[k] < cp$ holds, we know that all values in $PC[1..cn]$ must be smaller than $cp$, as $k$ was the index with the largest value. Otherwise, we report the occurrence and repeat the procedure with the intervals $[1, k-1]$ and $[k+1, cn]$. The recursion thus allows one to filter exactly the desired entries. Each reported occurrence causes at most 2 additional $\mathsf{rmq}$-queries. Let $occ$ be the number of occurrences of the patterns in $S$. Then, using the described procedure, at most $3 \cdot occ$ $\mathsf{rmq}$-queries are executed during the Aho-Corasick algorithm.

To save the space of the $PC$-array, we note that we can replace each access to the array with its above definition using only the existing components of an XBWT, reducing the additional space to $2m + o(m)$ bits for the range maximum query data structure. Algorithm 5.8 gathers all of the described ideas and is a replacement of lines 6–9 of Algorithm 5.7.

To summarize, we need an additional data structure using $2m + o(m)$ bits to solve the exact multi-string matching problem with an XBWT. The Aho-Corasick algorithm then requires $O((n + occ) \log \sigma)$ worst-case run-time where $occ$ is the number of occurrences of all patterns in the string $S$ of length $n$.

### 5.3.1 *XBWT Construction*

So far we have seen that the XBWT is a compact representation of extended tries which can be used in the Aho-Corasick algorithm. Of course, there are a plethora of other applications of tries as string dictionaries, like for example Natural Language processing, Web graphs or Bioinformatics, see [MP+16, pp. 74–75] for details and more applications.

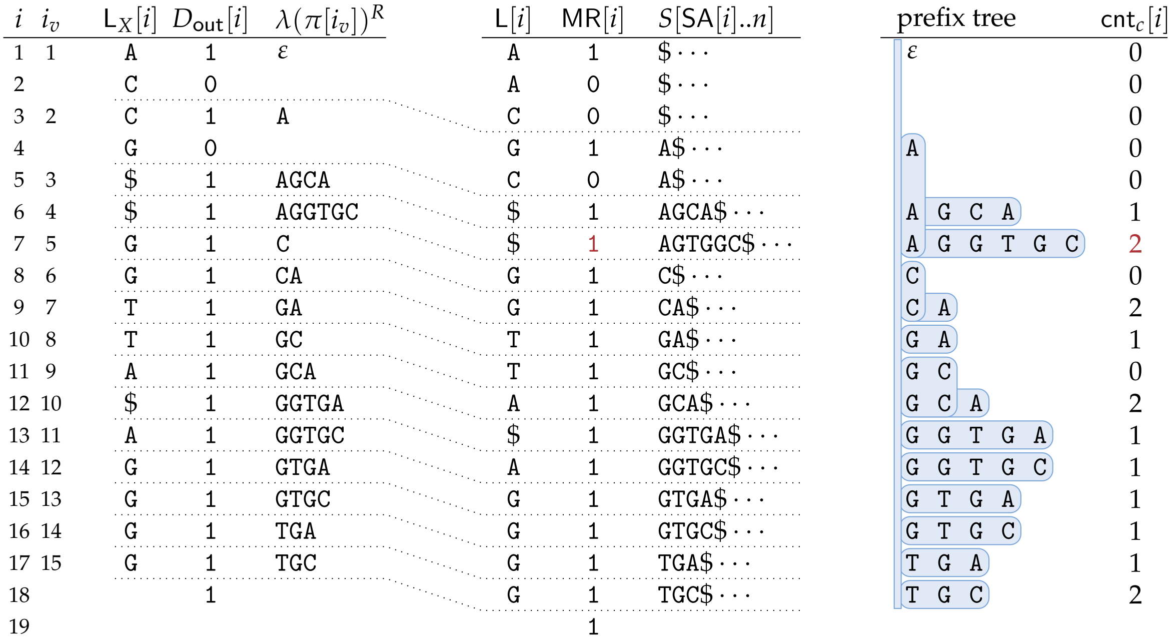
We now want to demonstrate how an XBWT can be constructed. Several approaches for XBWT construction varying in run-time and memory peak exist, see e.g. [Fer+05], [Man16] or [OSB18]. We will present only the currently fastest one, invented in our XBWT construction algorithm comparison paper [OSB18].

The basis of XBWT construction is formed by the BWT $\mathsf{L}$ of the string $S$ with length $n$ obtained by concatenating all reversed strings $S_1, \ldots, S_m$ and separating them with the character \$. More precisely, let $S_1, \ldots, S_m$ be the null-terminated strings of which the XBWT should be constructed. The string $S$ then is defined as

$$S := S_1[1..|S_1| - 1]^R \$ S_2[1..|S_2| - 1]^R \$ \cdots S_m[1..|S_m| - 1]^R \$.$$

The key idea is that the set of BWT characters $\{ \mathsf{L}[k] \mid k \in [i, j] \}$ in an $\omega\$$-interval $[i, j]$ of $\mathsf{L}$ corresponds to the set of labels of outgoing edges from a node $v$ with reverse node label $\lambda(v)^R = \omega$. For example, the set $\{ \mathsf{L}[k] \mid k \in [1, \mathsf{rank}_{\mathsf{L}}(\$, n)] \}$ directly corresponds to the labels of outgoing edges of the root node in the trie, see also Figure 5.13.

The first step in XBWT construction is to partition the sorted suffixes into $\omega\$$-intervals. The partition is indicated with a bit-vector $\mathsf{MR}$ of size $n + 1$ where each set bit at position $i$ means that a new $\omega\$$-interval starts at position $i$. Additionally, we set



| $i$ | $i_v$ | $\mathsf{L}_X[i]$ | $D_{\text{out}}[i]$ | $\lambda(\pi[i_v])^R$ | $\mathsf{L}[i]$ | $\mathsf{MR}[i]$ | $S[\mathsf{SA}[i]..n]$ | prefix tree | $\mathsf{cnt}_c[i]$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | A | 1 | $\varepsilon$ | A | 1 | $··· | $\varepsilon$ | 0 |
| 2 |  | C | 0 |  | A | 0 | $··· |  | 0 |
| 3 | 2 | C | 1 | A | C | 0 | $··· |  | 0 |
| 4 |  | G | 0 |  | G | 1 | A$··· | A | 0 |
| 5 | 3 | $ | 1 | AGCA | C | 0 | A$··· |  | 0 |
| 6 | 4 | $ | 1 | AGGTGC | $ | 1 | AGCA$··· | A G C A | 1 |
| 7 | 5 | G | 1 | C | $ | 1 | AGTGGC$··· | A G G T G C | 2 |
| 8 | 6 | G | 1 | CA | G | 1 | C$··· | C | 0 |
| 9 | 7 | T | 1 | GA | G | 1 | CA$··· | C A | 2 |
| 10 | 8 | T | 1 | GC | T | 1 | GA$··· | G A | 1 |
| 11 | 9 | A | 1 | GCA | T | 1 | GC$··· | G C | 0 |
| 12 | 10 | $ | 1 | GGTGA | A | 1 | GCA$··· | G C A | 2 |
| 13 | 11 | A | 1 | GGTGC | $ | 1 | GGTGA$··· | G G T G A | 1 |
| 14 | 12 | G | 1 | GTGA | A | 1 | GGTGC$··· | G G T G C | 1 |
| 15 | 13 | G | 1 | GTGC | G | 1 | GTGA$··· | G T G A | 1 |
| 16 | 14 | G | 1 | TGA | G | 1 | GTGC$··· | G T G C | 1 |
| 17 | 15 | G | 1 | TGC | G | 1 | TGA$··· | T G A | 1 |
| 18 |  |  | 1 |  | G | 1 | TGC$··· | T G C | 2 |
| 19 |  |  |  |  |  | 1 |  |  |  |

$$P = (\ (\ (\ )\ (\ )\ )\ (\ (\ )\ )\ (\ )\ (\ (\ )\ )\ (\ )\ (\ )\ (\ )\ (\ )\ (\ )\ (\ )\ )$$

**Figure 5.13:** Components of XBWT construction. The left-hand side shows selected components of an XBWT of the strings $S_1$ = ACGA$, $S_2$ = CGTGGA$ and $S_3$ = AGTGG$. The middle shows the BWT of the string $S$ = AGCA$AGGTGC$GGTGA$. The bit-vector MR partitions the suffix array into $\omega$\$-intervals. The correspondence between reversed node labels in the XBWT and $\omega$\$-intervals is indicated with dotted lines. The right-hand side shows the prefix tree of the reversed node labels as well as the counter array $\mathsf{cnt}_c$ indicating how many intervals end at each position. The red marked entries in MR and $\mathsf{cnt}_c$ correspond to the red marked parenthesis in the failure link sequence $P$.

a termination bit $\mathsf{MR}[n+1] = 1$. The bit-vector MR can be computed as follows: we enumerate the \$-interval, then all $c$\$-intervals, then all $\tilde{c}c$\$-intervals and so on, and stop the left-extension of an interval if the left-extending character is a \$. Meanwhile, we set $\mathsf{MR}[i] = 1$ for each enumerated interval and thus obtain the desired bit-vector, see Algorithm 5.9. We note that due to the observation from above, this emulates a traversal of all inner nodes of the trie.

For the computation of the balanced parentheses sequence $P$ we need information about the prefix tree of the reversed node labels, see Figure 5.13. A node in this prefix tree corresponds to an $\omega$-interval in the suffix array. The left boundary of such an $\omega$-interval corresponds to the left boundary of the $\omega$\$-interval, so the left boundaries of the desired $\omega$-intervals correspond to the positions of set bits in MR. Now let $[i, j]$ be an $\omega$-interval. The right boundaries of all $c\omega$-intervals then can be computed

```
Data: Null-terminated strings $S_1, \ldots, S_m$, BWT $\mathsf{L}$ in form of an FM-index of the string
      $S = S_1[1..|S_1|-1]^R \cdots S_m[1..|S_m|-1]^R\$$ of length $n$, C-array $\mathsf{C_L}$ of string $\mathsf{L}$.
Result: A bit-vector $\mathsf{MR}$ of size $n+1$ storing the left boundaries of all $\omega\$$-intervals in $S$, a counter array $\mathsf{cnt}_c$ of size
        $n$ storing the right boundaries of all $\omega$-intervals where the $\omega\$$-interval is not empty.

1  initialize a bit-vector $\mathsf{MR}$ of size $n+1$ filled with zeros
2  $\mathsf{MR}[n+1] \leftarrow 1$
3  initialize a counter array $\mathsf{cnt}_c$ of size $n$ initialized with zeros
4  initialize an empty queue $Q$
5  push element $\langle 1, m, n \rangle$ on $Q$
6  while $Q$ is not empty do
7      $\langle i_\$, j_\$, j \rangle \leftarrow$ first element of $Q$
8      pop first element of $Q$
                                   // mark left boundary of $\omega\$$-interval and right boundary of $\omega$-interval
9      $\mathsf{MR}[i_\$] \leftarrow 1$
10     $\mathsf{cnt}_c[j] \leftarrow \mathsf{cnt}_c[j] + 1$
                                                                        // enumerate $c\omega\$$-intervals
11     $M \leftarrow \mathsf{getIntervals_L}(i_\$, j_\$)$
12     foreach $\langle c, [lb_\$, rb_\$] \rangle \in M$ do
13         if $c \neq \$$ then
14             if $j_\$ = j$ then                                   // avoid unnecessary rank-queries
15                 $rb \leftarrow rb_\$$
16             else
17                 $rb \leftarrow \mathsf{C_L}[c] + \mathsf{rank_L}(c, j)$
18             push element $\langle lb_\$, rb_\$, rb \rangle$ to queue $Q$
19 return $\langle \mathsf{MR}, \mathsf{cnt}_c \rangle$
```

**Algorithm 5.9:** Construction of the bit-vector MR and the counter array $\mathsf{cnt}_c$. The algorithm is derived from a similar algorithm presented in [OSB18].

using $rb \leftarrow \mathsf{C_L}[c] + \mathsf{rank_L}(c, j)$. This means that we can integrate the enumeration of all desired $c\omega$-interval boundaries into the enumeration of $\omega\$$-intervals by keeping track of each right boundary. To store these boundaries, we use a counter array $\mathsf{cnt}_c$ of size $n$ and increment the counter each time we visit a right boundary, see Algorithm 5.9. As we will see later, the information in MR and $\mathsf{cnt}_c$ suffices to compute $P$.

The run-time of Algorithm 5.9 is determined by the number of triples which reside in the queue $Q$ during execution. Each new triple requires $O(\log \sigma)$ time for generation (lines 11, 12 and 17) and corresponds to a node in the trie. The overall run-time of Algorithm 5.9 is thus given by $O(n \log \sigma)$, because the trie cannot have more than $n$ nodes. This run-time also covers the initialization processes in lines 1–3. Note that the integer width of each entry in the counter-array $\mathsf{cnt}_c$ must be $\log n$ bits. In the worst case, the right boundaries of all intervals coincide. On Page 165 we will present a way to reduce the size of $\mathsf{cnt}_c$ but for now concentrate on the remaining XBWT construction.

To finish the "shape components" $\mathsf{L_X}$ and $D_{\mathrm{out}}$ of the XBWT, we must remove duplicate characters within each $\omega\$$-interval, as a node cannot have two outgoing edges with the same label. An easy way to remove duplicates is to make use of the

```
Data: Null-terminated strings $S_1, \dots, S_m$, BWT L in form of an FM-index of the string
      $S = S_1[1..|S_1|-1]^R \cdots S_m[1..|S_m|-1]^R\$$ of length $n$, bit-vector MR computed from Algorithm 5.9.
Result: XBWT components $\mathsf{L}_X$ and $D_{\text{out}}$.

initialize a string $\mathsf{L}_X$ of size $\mathsf{rank}_{\mathsf{MR}}(1,n) + m - 1$
$i \leftarrow 1$
$k \leftarrow 1$
for $j \leftarrow 1$ to $n$ do
    if $\mathsf{MR}[j+1] = 1$ then
        $M \leftarrow \mathsf{getIntervals}_{\mathsf{L}}(i,j)$
        foreach $\langle c, [lb, rb] \rangle \in M$ do
            $\mathsf{L}_X[k] \leftarrow c$
            $\mathsf{MR}[k] \leftarrow 0$
            $k \leftarrow k + 1$
        $\mathsf{MR}[k - |M|] \leftarrow 1$
        $i \leftarrow j + 1$
$\mathsf{MR}[k] \leftarrow 1$
trim MR to size $k$
return $\langle \mathsf{L}_X, \mathsf{MR} \rangle$
```

**Algorithm 5.10:** Construction of the XBWT components $\mathsf{L}_X$ and $D_{\text{out}}$. The algorithm is derived from a similar algorithm presented in [Man16].

getIntervals-function for each $\omega\$$-interval: the function enumerates the characters in the interval only once per character. The bit-vector $D_{\text{out}}$ can be computed by counting the number of remaining characters in each interval. Algorithm 5.10 shows how both operations can be combined. The bit-vector MR is overwritten with the final bit-vector $D_{\text{out}}$. The worst-case run-time of Algorithm 5.10 is bound by $O(n \log \sigma)$.

The idea for the construction of the balanced parentheses sequence was presented in [OSB18]. It originates from a similar construction of a balanced parentheses sequence used to represent the topology of a suffix tree, devised by Belazzougui [Bel14]:

> [...] Our key observation is that we can easily build a balanced parenthesis representation by enumerating the suffix array intervals. More precisely for every position in $[1..n]$, we associate two counters, one for open and the other for close parentheses implemented through two arrays of counters $C_o[1..n]$ and $C_c[1..n]$. Then given a suffix array interval $[i, j]$ we will simply increment the counters $C_o[i]$ and $C_c[j]$. Then we scan the counters $C_c$ and $C_o$ in parallel and for each $i$ from 1 to $n$, write $C_o[i]$ opening parentheses followed by $C_c[i]$ closing parentheses. It is easy to see that the constructed sequence is that of the balanced parentheses of the suffix tree. [...]

The construction of a balanced parentheses sequence for prefix trees works in a similar way. In our case, the left boundaries of all $\omega$-intervals are stored in the

```
Data: Bit-vector MR and counter array $\mathsf{cnt}_c$ computed from Algorithm 5.9.
Result: XBWT failure link component $P$.

initialize a parentheses sequence $P$ of size $2 \cdot \mathsf{rank}_{\mathsf{MR}}(1, n)$
$k \leftarrow 1$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $\mathsf{MR}[i]$ do
        $P[k] \leftarrow$ "("
        $k \leftarrow k + 1$
    for $j \leftarrow 1$ to $\mathsf{cnt}_c[i]$ do
        $P[k] \leftarrow$ ")"
        $k \leftarrow k + 1$
return $P$
```

**Algorithm 5.11:** Construction of the XBWT component $P$.

bit-vector MR, so we can treat MR similar to the $C_0$ array from above. The right boundaries of all $\omega$-intervals were counted in the $\mathsf{cnt}_c$ array, so we can analogously treat $\mathsf{cnt}_c$ as $C_c$. The construction of $P$ thus is rather simple and is shown in Algorithm 5.11. The worst-case run-time of the algorithm clearly is $O(n)$.

As Algorithm 5.10 overwrites the MR-array, one should run Algorithm 5.11 before executing Algorithm 5.10. The computation of the $\mathsf{C}_X$-array is straightforward by Definition 5.11. The last component which has to be constructed is the record array $R$ which assigns the record number to each corresponding \$ in $\mathsf{L}_X$.

An efficient construction algorithm of the record array has not been published until now, so we will give one here. The first ingredient is a mapping from the \$'s in L to \$'s in F. By the definition of the LF-mapping (see Page 13) we know that the $k$-th occurrence of a \$ in L corresponds to the $k$-th occurrence of a \$ in F. However, as we used multiple \$'s in our string, the first \$ in F must be handled differently. Unlike the other entries, this entry does not inherit the string context because the corresponding suffix consists only of the character \$. Let $d_c$ be the rank of the corresponding \$ in L, that is, $d_c = \mathsf{rank}_{\mathsf{L}}(\$, i)$ where $i$ is the BWT index, i.e. $\mathsf{SA}[i] = 1$. The mapping from \$'s in L to \$'s in F is thus as follows: the $d_c$-th \$ in L corresponds to the first \$ in F. For the remaining entries, the $k$-th \$ in L ignoring entry $d_c$ then corresponds to the $k+1$-th \$ in F.

This special mapping can be easily computed given the entry $d_c$. The entry $d_c$ itself is easily computable because typical BWT construction algorithms compute the BWT index $i$. Using an FM-index, $d_c$ can be computed using $d_c = \mathsf{rank}_{\mathsf{L}}(\$, i)$. Next, we want to map the \$'s in F to the records using the first $m$ entries of the suffix array. In an XBWT, navigation to a child is performed by a backward step. This means that each \$ in F corresponds to the nearest left occurrence of a \$ next to a record in $S$.

```
Data: Number $m$ of null-terminated strings $S_1, \ldots, S_m$, suffix array $\mathsf{SA}[1..m]$ of the string
      $S = S_1[1..|S_1|-1]^R \cdots S_m[1..|S_m|-1]^R\$$, rank $d_c$ of the cyclic $ in $\mathsf{L}$, i.e. $d_c = \mathsf{rank}_{\mathsf{L}}(\$, i)$ where $i$ is the index
      satisfying $\mathsf{SA}[i] = 1$.
Result: XBWT record array $R$.

                                     // set up a list of records and their nearest left $ in S
1  let $A$ be an array of pairs of size $m$
2  $j \leftarrow 2$                                 // position of current non-cyclic $ in F.
3  for $i \leftarrow 1$ to $m$ do
4      if $i = d_c$ then                          // special handling for cyclic entry
5          $A[i] \leftarrow \langle 0, i \rangle$
6      else
7          $A[i] \leftarrow \langle \mathsf{SA}[j], i \rangle$
8          $j \leftarrow j + 1$

                                                            // compute record array
9  sort $A$ by the first key of the pairs
10 let $R$ be an array of size $m$
11 for $i \leftarrow 1$ to $m$ do
12     $\langle sa, j \rangle \leftarrow A[i]$
13     $R[j] \leftarrow i$
14 return $R$
```

**Algorithm 5.12:** Computation of the XBWT record array $R$.

The nearest left occurrence of a \$ next to the first record in $S$ is at position $n$ because the string is viewed cyclically. The nearest left occurrence of a \$ next to the second record then can be found at the first occurrence of \$ in $S$. The nearest left \$ next to the third record is at the second occurrence of a \$ in $S$, and so on.

We now combine everything by setting up an array $A$ containing pairs $\langle \mathsf{SA}[j], i \rangle$ that map the \$'s in $\mathsf{L}$ to their position in the text. To handle the special cyclic case for the first record, we set $A[d_c] \leftarrow \langle 0, d_c \rangle$. Now, the array $A$ is sorted by the first key of the pairs. Afterwards, the entry $A[j]$ contains the $j-1$-th occurrence of a \$ in $S$, or analogously, $A[j]$ refers to the $j$-th record. Additionally, the second key stores the lexicographic rank of the suffix starting at the $j-1$-th occurrence of a \$ in $S$. We can thus reversely plug the position and the second key into the array $R$ and get our desired result.

The full record array construction can be found in Algorithm 5.12. Using a standard sorting algorithm [Wil64], Algorithm 5.12 requires $O(m \log m)$ time. If we assume that the suffix array SA of $S$ is stored in an external memory, we can save space by using an arrangement of the array $A$. The second key of the array $A$ can be represented using the array $R$, while the first key of the array can be represented using an array of integers with width $\log n$. After sorting, we copy the second key to the first key of $A$, so the array $R$ can be used again. Then, we only have to adapt line 8 such that the variable $j$ contains the first key of $A$. This means that Algorithm 5.12 needs $m \log n$ bits of additional working space.

**Corollary 5.13.** *Let $S_1, \dots, S_m$ be a set of null-terminated strings and let $S$ be a string of length $n$ defined as $S := S_1[1..|S_1|-1]^R\$ \cdots S_m[1..|S_m|-1]^R\$$. The XBWT of the extended trie $\mathcal{T}_{\mathsf{ex}}(S_1, \dots, S_m)$ can be constructed in $O(n \log \sigma + m \log m)$ time with the following steps:*

1. *Suffix array construction and FM-index construction of $S$ in $O(n \log \sigma)$ time.*
2. *Construction of $\mathsf{MR}$ and $\mathsf{cnt}_c$ with Algorithm 5.9 in $O(n \log \sigma)$ time. Both components require $n$ and $n \log n$ bits of space.*
3. *Construction of the balanced parentheses sequence $P$ with Algorithm 5.11 in $O(n)$ time using the arrays $\mathsf{cnt}_c$ and $\mathsf{MR}$.*
4. *Construction of the shape components $\mathsf{L}_X$ and $D_{\mathsf{out}}$ with Algorithm 5.10 in $O(n \log \sigma)$ time using the FM-index and $\mathsf{MR}$. The bit-vector $\mathsf{MR}$ can be overwritten with the $D_{\mathsf{out}}$ component during execution, so no additional space for $D_{\mathsf{out}}$ is needed.*
5. *Construction of the $\mathsf{C}_X$ array using $\mathsf{L}_X$ in $O(n)$ time.*
6. *Construction of the record array $R$ using Algorithm 5.12 and $\mathsf{SA}[1..m]$ in $O(m \log m)$ time and $m \log n$ bits of additional working space.*

Looking at Corollary 5.13, we note that the counter array $\mathsf{cnt}_c$ requires a lot of space. The counter array requires $n \log n$ bits of space for saving the information of at most $n$ right boundaries of intervals in the prefix tree. Additionally, the counter array is used solely for the construction of $P$ and is not overwritten during XBWT construction. We thus present a way to save working space during construction by using a succinct counter array as presented by Cunial et al. [CAB19].

**Remark 5.14** (Succinct counter array)**.** Suppose we get $n$ integer positions $i_1, \dots, i_n$ which are bounded by $n$ ($i_k \in [1, n]$) and want to count how often any position $i \in [1, n]$ was hit. More formally, we want to know the value $C(i) := |\{\, k \in [1, n] \mid i_k = i \,\}|$ for each $i \in [1, n]$. A normal counting array requires $n \cdot \log(n)$ bits because in the worst case all integer positions are equal, so each position must be able to cover the value $n$.

Instead, according to [CAB19], a succinct counter array can be designed as follows: create an array $c$ such that each entry uses only $b$ bits, and count positions similar to a conventional counting array. Once an entry is full, i.e. $c[i] = 2^b$, create an entry in a map $M$ which contains $i$ as a key and the count as value. If $i$ is hit again, increment the count in $M$. To read a count of a position $i$, if $c[i] < 2^b$ holds, use the value $c[i]$, otherwise access $M$ to get the correct count.

Using the map $M$, it is still possible to cover the value $n$, but it is unlikely that $M$ will contain many entries. Also, frequent positions reside in the cache, making accesses to $M$ efficient. To find an appropriate value for $b$ we regard the worst-case space consumption. In the worst case, $M$ can have $n/2^b$ entries. If $m$ is the space consumption of a single entry in $H$, the space consumption is

$$\underbrace{n \cdot b}_{c \text{ array}} + \underbrace{\frac{n}{2^b} \cdot m}_{\text{map } M} = n \cdot \left(b + \frac{m}{2^b}\right) \text{ bits.}$$

Depending on the implementation of the map $M$, a typical value for $m$ on 64-bit platforms could be $4 \cdot 64 = 256$: In the case of a binary tree, two integers for position and count as well as two pointers for the child nodes are required. In the case of a hashmap, we can set the initial number of buckets to $n/2^b$ and for each entry use two integers for position and count as well as two pointers for the doubly linking between elements with the same hashed key. Defining $f(b) := b + \frac{256}{2^b}$, we get the derivation $f'(b) = 1 - \frac{\log_e(2) \cdot 256}{2^b}$. As $f$ is a convex function, the minimum of $f$ corresponds to the root of $f'$, which is approximately given at $b \approx 7.47$. As $b$ must be an integer, taking into account that integer arrays with bit width 8 are common on most platforms, we set $b = 8$. Plugging the values $b = 8$ and $m = 256$ into the worst-case space consumption shows that this form of succinct counter array requires at most

$$n \cdot 8 + n \cdot \frac{256}{2^8} = 9 \cdot n \text{ bits.}$$

This is beneficial compared to a normal counting array when $\log_2(n) > 9$, or similarly, when $n > 512$ holds, which is true for all of the herein used data sets.

### 5.3.2 *Experimental results*

We implemented the described XBWT construction algorithm which we called XBWT fast. The implementation is publicly available [Bai20]. We compared the algorithm against the most memory-saving construct algorithm from [OSB18], which we called XBWT lightweight. XBWT fast and XBWT lightweight differ in the way the balanced parentheses sequence $P$ is computed. While XBWT fast computes the arrays MR and

$P$ in only one traversal of all $\omega\$$-intervals, XBWT lightweight uses one traversal for MR and another one for $P$. The two traversals allow for the reduction of the counter array size, see [OSB18] for more details.

We used specially prepared input data from Chapter A as input. Empty lines and lines with less than 10 characters were filtered out. Additionally, lines containing other lines as proper substring were filtered out, to ensure correctness of the Aho-Corasick algorithm. From such modified files we used only files with more than 1 MB of size and with more than 1000 lines to ensure a certain branching factor and size of the resulting trie. The lines of such files then were used as input strings for the trie. A full description of the generated test data can be found in Section C.2. Average timings and memory peaks during construction can be found in Figure 5.14.

The figure shows that XBWT fast requires about twice the time of the FM-index construction, while XBWT lightweight requires about four times the amount of the time of the FM-index construction. This is owed to the second traversal over all $\omega\$$-intervals, making the algorithm slower. Additionally it can be seen that the lightweight algorithm saves space, but only a very small amount. The best space-saving technique is the use of succinct counters, leaving the run-time of the algorithms almost unchanged.

At first glance it seems unnecessary to save space during construction because the memory peak is dominated by FM-index construction, but we note here that slower but more space-saving algorithms for FM-index construction exist. The dominating memory peak in FM-index construction comes from suffix array construction, for

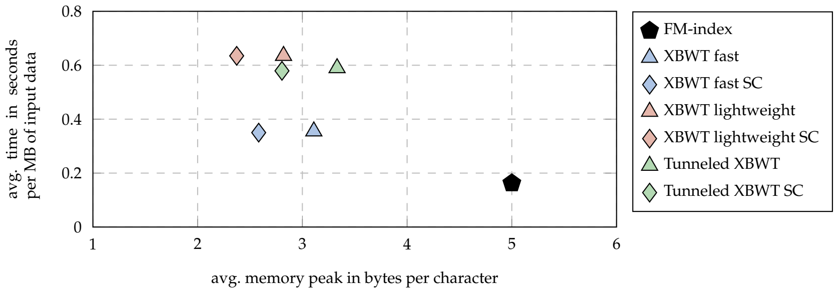


**Figure 5.14:** Average space-time tradeoff of different trie construction algorithms. The further down a mark in the graph is, the faster the algorithm runs. The more to the left a mark in the graph is, the less memory is required. The trie construction algorithms require the FM-index and the (externally saved) suffix array of the underlying test data as input. The SC variants refer to algorithms using succinct counters. The full benchmark results can be found in the Figures C.7 and C.8.

which slower but more memory-saving algorithms exist, see e.g. [Bel17a]. However, we used fast suffix array construction [Mor03] because we were only interested in the timings for pure XBWT construction.

Figure 5.14 clearly shows that the algorithm XBWT fast with the use of succinct counters offers the best space-time tradeoff, and is therefore the method of choice for XBWT construction. The figure also includes construction algorithms for the construction of a tunneled XBWT. It can be seen that these algorithms require a bit more memory and require about 80 % more time than the XBWT fast variants, but are still faster than the XBWT lightweight variants. The time penalty mainly comes from the additional execution of the de Bruijn graph edge minimization algorithm which is required for tunneling.

We will introduce trie tunneling in the next section to see how beneficial tunneling of an XBWT can be. More information about the size of an XBWT and the performance of the Aho-Corasick algorithm using an XBWT can be found in the Sections 5.4.4 and C.2.

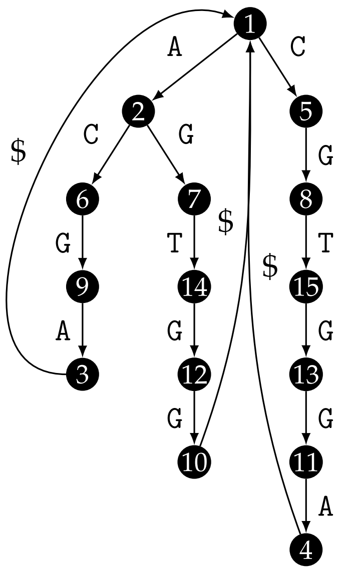
## 5.4 TRIE TUNNELING

The last part of this chapter combines our previous results about tunnels related to de Bruijn graph edge minimization (Section 5.1) and the succinct representation of extended tries (Section 5.3). The results of this chapter have not been published yet but were already presented in a student project supervised by the author of this thesis [RHRH20].

We will answer four main questions in this section:

- Is it possible to produce a tunneled XBWT (or TXBWT in short), and how can we traverse it?
- How can failure links be retained in a tunneled BWT?
- How can the components of a TXBWT be constructed efficiently?
- What are the gains and performance penalties of a TXBWT?

We will see that the special class of k-mer prefix intervals related to de Bruijn graph edge minimization is essential to get both concepts together and answer the questions from above appropriately.



| L | $D_{\text{out}}$ | $D_{\text{in}}$ | F |
|---|---|---|---|
| A | 1 | 1 | $ |
| C | 0 | 0 | $ |
| | | 0 | $ |
| C | 1 | 1 | A |
| G | 0 | | |
| $ | 1 | 1 | A |
| $ | 1 | 1 | A |
| G | 1 | 1 | C |
| G | 1 | 1 | C |
| T | 1 | 1 | G |
| T | 1 | 1 | G |
| A | 1 | 1 | G |
| $ | 1 | 1 | G |
| A | 1 | 1 | G |
| G | 1 | 1 | G |
| G | 1 | 1 | G |
| G | 1 | 1 | T |
| G | 1 | 1 | T |
| | 1 | 1 | |

| $c$ | $ | A | C | G | T |
|---|---|---|---|---|---|
| $\mathsf{C}_{\mathsf{L}}[c]$ | 0 | 3 | 6 | 8 | 15 |

| $\mathsf{L}_X$ | $D_{\text{out}}$ | $D_{\text{in}}$ | $\lambda(\cdot)^R$ |
|---|---|---|---|
| A | 1 | 1 | $\varepsilon$ |
| C | 0 | | |
| C | 1 | 1 | A |
| G | 0 | | |
| $ | 1 | 1 | AG··· |
| $ | 1 | 1 | AG··· |
| G | 1 | 1 | C |
| G | 1 | 1 | CA |
| T | 1 | 1 | GA |
| T | 1 | 1 | GC |
| A | 1 | 1 | GC··· |
| $ | 1 | 1 | GG··· |
| A | 1 | 1 | GG··· |
| G | 1 | 1 | GT··· |
| G | 1 | 1 | GT··· |
| G | 1 | 1 | TG··· |
| G | 1 | 1 | TG··· |
| | 1 | 1 | |

| $c$ | $ | A | C | G | T |
|---|---|---|---|---|---|
| $\mathsf{C}_X[c]$ | 0 | 1 | 4 | 6 | 13 |

**Figure 5.15:** Comparison of the shape components of an XBWT (right) and the succinct representation (middle) of a Wheeler graph representation (left) of the underlying trie. The trie is set up from the strings $S_1$ = ACGA$, $S_2$ = CGTGGA$ and $S_3$ = AGTGG$. The blue and red dashed lines indicate boundaries in the representations determined by the corresponding C-array. The $D_{\text{in}}$ bit-vector in the XBWT is grayed out because it is normally omitted as it consists only of ones.

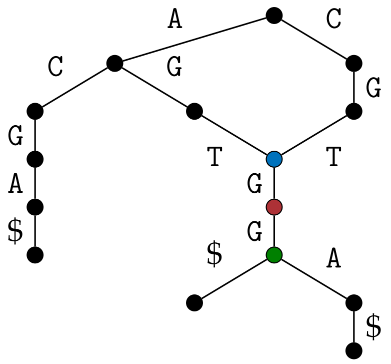
### 5.4.1 *Tunneled XBWT introduction and trie traversal*

We first will show that it is possible to tunnel XBWTs. In the principles about Wheeler graphs in Section 2.4.2 we have seen that tries can be formulated as Wheeler graphs. To do this, we removed leaves from the trie and moved the endpoints of incoming edges to the root node, see Figure 5.15. The consequence is that the root node has an indegree of $m$, where $m$ is the number of strings contained in the trie.

In an XBWT we have the convention that it is not allowed to follow edges labeled with the termination symbol $. This allows us to define the indegree of the root node to one, so the succinct indegree bit-vector $D_{\text{out}}$ consists of only ones and thus does not need to be represented explicitly. To perform correct trie navigation, we defined the cumulative character count array $\mathsf{C}_X[c]$ as $\mathsf{C}_{\mathsf{L}_X}[c] - (m-1)$ which allows us to ignore the $ characters in L, see also Figure 5.15.

Therefore, the XBWT can be seen as an optimized version of the succinct Wheeler graph representation of tries. Because we are not interested in following edges

| $\widetilde{\mathtt{L}}_X$ | $D_{\text{out}}$ | $D_{\text{in}}$ | $\lambda(\cdot)^R$ |
|---|---|---|---|
| A | 1 | 1 | $\varepsilon$ |
| C | 0 | | |
| C | 1 | 1 | A |
| G | 0 | | |
| $ | 1 | 1 | AGCA |
| $ | 1 | 1 | AGGTGC |
| G | 1 | 1 | C |
| G | 1 | 1 | CA |
| T | 1 | 1 | GA |
| T | 1 | 1 | GC |
| A | 1 | 1 | GCA |
| $ | 1 | 1 | GG··· |
| A | 0 | | |
| G | 1 | 1 | GT··· |
| | | | |
| G | 1 | 1 | TG··· |
| | | 0 | |
| | 1 | 1 | |



| $c$ | $ | A | C | G | T |
|---|---|---|---|---|---|
| $\widetilde{\mathtt{C}}_X[c]$ | 0 | 1 | 4 | 6 | 11 |

**Figure 5.16:** Tunneled XBWT shape components (left and bottom right) and an illustration of the associated word graph (top right). The tunneled XBWT was obtained by tunneling the $(14, 12, 10), (15, 13, 11)$ prefix interval in the Wheeler graph from Figure 5.15. The tunnel is marked in both the TXBWT and the associated word graph using blue, red and green. A similar image appeared in [RHRH20].

labeled with the termination character, we removed them from the representation. The remaining edges in an XBWT are isomorphic to the edges in the succinct Wheeler graph representation of tries, so a tunneled Wheeler graph also automatically implies a tunneled XBWT.

The effect of tunneling an XBWT can be seen in Figure 5.16. Tunneling converts the tree structure of a trie to a directed word graph, but still allows one to traverse the graph as if it is a tree. This is interesting in its own right, because the graph corresponding to the tunneled XBWT might contain loops. However, we now want to focus on navigation in a tunneled XBWT.

As a short reminder, tunneling fuses parallel equally labeled paths. The tunnel start is indicated by an indegree greater than one, represented in the bit-vector $D_{\text{in}}$. Analogously, the tunnel end is indicated by an outdegree greater than one represented in the bit-vector $D_{\text{out}}$. Note that the XBWT also contains "normal" branching nodes with outdegree greater than one. To distinguish between a normal branching node and the end of a tunnel, we thus have to save whether we navigate in a tunnel or not.

**Data:** Tunneled XBWT with components $\widetilde{L}_X$, $\widetilde{C}_X$, $D_{\text{out}}$ and $D_{\text{in}}$, node index $i_v$, uppermost tunnel entry edge $e_{\text{top}}$, tunnel entry edge $e_{\text{ent}}$, character $c$.
**Result:** Node index $j_v$, uppermost tunnel entry edge $e_{\text{top}}$, tunnel entry edge $e_{\text{ent}}$ in case that an edge from node $i_v$ to node $j_v$ labeled with $c$ exists.

```
function child($\langle i_v, e_{\text{top}}, e_{\text{ent}} \rangle, c$)
    $lb \leftarrow \text{select}_{D_{\text{out}}}(1, i_v)$
    $rb \leftarrow \text{select}_{D_{\text{out}}}(1, i_v + 1) - 1$
                                                  // check if the end of a tunnel is reached
    if $e_{ent} \neq \bot$ and $D_{\text{out}}[rb] = 0$ then
        $lb \leftarrow lb + (e_{\text{ent}} - e_{\text{top}})$          // jump to correct lane
        $rb \leftarrow lb$
        $e_{\text{top}} \leftarrow \bot$
        $e_{\text{ent}} \leftarrow \bot$
                                  // check if node $\pi[i_v]$ has an outgoing edge labeled with $c$
    $r_{lb} \leftarrow \text{rank}_{\widetilde{L}_X}(c, lb - 1)$
    $r_{rb} \leftarrow \text{rank}_{\widetilde{L}_X}(c, rb)$
    if $r_{lb} = r_{rb}$ then                              // no edge with character $c$ available
        return $\bot$
                                                                  // navigate to child node
    $e \leftarrow \widetilde{C}_X[c] + r_{rb}$
    $j_v \leftarrow \text{rank}_{D_{\text{in}}}(1, e)$
                                             // check if the start of a new tunnel is detected
    if $D_{\text{in}}[e] = 0$ then
        $e_{\text{top}} \leftarrow \text{select}_{D_{\text{in}}}(1, j_v)$
        $e_{\text{ent}} \leftarrow e$
    else if $D_{\text{in}}[e + 1] = 0$ then                // start of a tunnel at uppermost entry
        $e_{\text{top}} \leftarrow e$
        $e_{\text{ent}} \leftarrow e$
    return $\langle j_v, e_{\text{top}}, e_{\text{ent}} \rangle$
```

**Algorithm 5.13:** Child navigation in a tunneled XBWT with non-overlapping tunnels. A similar algorithm appeared in a student project [RHRH20].

Because tunneling fuses only parallel paths, there is no possibility for a branching node to be reached in the middle of a tunnel.[2] To indicate if a node with outdegree greater than one is the end of a tunnel or a branching node, we use two variables $e_{\text{top}}$ and $e_{\text{ent}}$. The first variable (if set) saves the position of the uppermost incoming edge of a tunnel in $D_{\text{in}}$ while the second variable saves the position of the incoming edge at which the tunnel was entered. When a node with outdegree greater than one is reached, we can distinguish between a tunnel end and a branching node by checking if variable $e_{\text{ent}}$ is set.

The remaining navigation is equivalent to the navigation in a tunneled BWT or an XBWT respectively, and can be found in Algorithm 5.13. Although it seems unnecessary to use two variables storing information about the tunnel entry edge,

[2] This also justifies why we defined prefix intervals as parallel paths and not as isomorphic subgraphs in Section 3.2. In the latter case, it could be possible that tunnel starts and ends are misinterpreted as normal branching nodes, leading to incorrect navigation.

we will see that this information is essential to support failure links in a TXBWT, which is the topic of the next section. The same section will also show why the tunneled prefix intervals in a trie should not overlap when failure links are to be supported.

### 5.4.2 *Failure link support*

Failure links are an essential tool to support an efficient multi-pattern search, see Algorithm 5.6. As a reminder, we repeat the definition of failure links as introduced in Section 5.3. Given an inner node $v$ of the trie with reversed node label $\lambda(v)^R$, the failure link $\phi(v)$ points to the unique node whose reversed node label is the longest proper prefix of $\lambda(v)^R$.

$$\phi(u) := \operatorname*{argmax}_{v \in V}\{\ |\lambda(v)|\ \mid\ \lambda(v)^R \text{ is a proper prefix of } \lambda(u)^R\ \}.$$

Tunneling fuses paths and thereby removes nodes from the graph. We face two problems when combining tunneling and failure links:

1. How can we handle failure links which originally pointed to a node that was removed by tunneling?
2. How can we emulate failure links starting at removed nodes?

We will see that both problems can be solved if tunnels are built only from the restricted class of k-mer prefix intervals. Again, as a reminder, k-mer prefix intervals have the property that the nodes of a column of the prefix interval share a common prefix $\omega$ of length k in their labels. Moreover, no other nodes outside of the column have $\omega$ as prefix of their labels. We refer to Definition 5.2 for more details.

Our first goal is to avoid failure links pointing to removed nodes. We therefore formulate the following isolation lemma that enables us to set up distinct failure link groups of nodes.

**Lemma 5.15.** *Let $\mathcal{T} = (V, E)$ be an extended trie, and let $\omega \neq \varepsilon$ be a string. Define $V_\omega$ as the set of nodes whose reversed node label share the common prefix $\omega$, i.e.*

$$V_\omega := \{\ v \in V\ \mid\ v \text{ is an inner node and } \omega \text{ is a prefix of } \lambda(v)^R\ \}.$$

*The set $V_\omega$ then fulfills the following properties:*

1. *No failure link outside of $V_\omega$ points into $V_\omega$.*
   *More precisely, for all inner nodes $v \in V \setminus V_\omega$ except of the root, $\phi(v) \notin V_\omega$ holds.*

2. *All failure links in $V_\omega$ pointing out of $V_\omega$ point to the same node.*
   *Let $u, v \in V_\omega$ be nodes with $\phi(u) \notin V_\omega$ and $\phi(v) \notin V_\omega$. Then, $\phi(u) = \phi(v)$ holds.*

*Proof.* By the definition of the set $V_\omega$, no inner node $v \in V \setminus V_\omega$ shares the prefix $\omega$ in its reversed node label $\lambda(v)^R$. Therefore, the failure link of such a node must point to a node whose reversed node label does not share the prefix $\omega$, or is a proper prefix of $\omega$. In both cases, the node $\phi(v)$ will not belong to the set $V_\omega$, proving Property 1.

Now suppose we have two nodes $u, v \in V_\omega$ with $\phi(u) \notin V_\omega$ and $\phi(v) \notin V_\omega$. Because failure links point to the longest proper prefix nodes, and both failure links point out of $V_\omega$, both reversed node labels $\lambda(\phi(u))^R$ and $\lambda(\phi(v))^R$ must be proper prefixes of $\omega$. As failure links point to the longest proper prefix nodes, $\lambda(\phi(u))^R = \lambda(\phi(v))^R$ and so $\phi(u) = \phi(v)$ must hold. This proves Property 2. □

As we have learned from Lemma 5.15, sets $V_\omega$ of nodes sharing an equal prefix $\omega$ in their reversed prefix label are ideal candidates for columns of prefix intervals. First, no failure link from outside points into the set, so removing nodes inside the set is no problem. Second, all failure links pointing out of the set are equal, so we simply need to retain the failure link of the node with the lexicographically smallest reversed node label and can use that for any other outgoing link. Finally, these nodes all share the prefix $\omega$ in their reversed labels, so they are adjacent to each other regarding the Wheeler graph order. An illustration of Lemma 5.15 and its consequences for fused paths in a trie is shown in Figure 5.17.

Of course, this holds true only if the considered sets $V_\omega$ are disjoint. In case that two sets intersect, failure links might point from one set to the other, making node removals difficult. This for example is the case when overlapping prefix intervals are tunneled, and justifies why only non-overlapping prefix intervals should be considered.

The easiest way to set up a list of candidates for columns of prefix intervals consists of grouping nodes whose reverse node label share a common prefix of some fixed length k. This ensures that the sets are disjoint. Nodes having a label with less than k characters are ignored in this scenario. However, as a trie is a branching data structure, the number of such nodes should be limited depending on the value of k. To find a good value for k, we remind that the de Bruijn graph edge minimization from Section 5.2 concerns a similar problem. We will see how both concepts can be

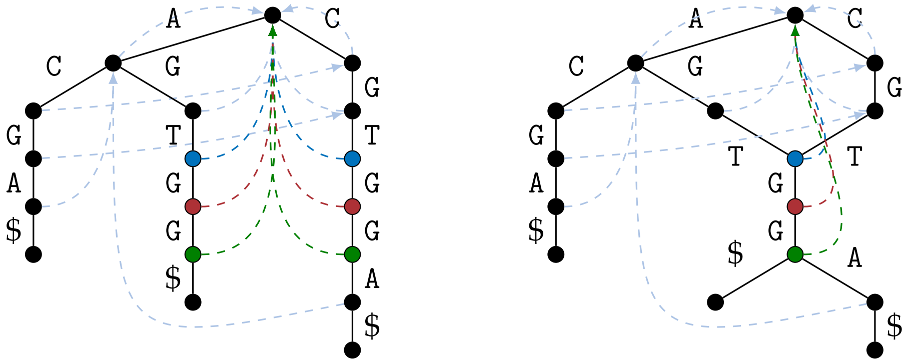


**Figure 5.17:** Illustration of the isolation Lemma 5.15. The left side shows a normal trie with the node sets $V_{\texttt{TG}}$, $V_{\texttt{GT}}$ and $V_{\texttt{GG}}$. No failure link points into the sets, and each failure link pointing out of a set points to the identical target node. This enables one to tunnel the underlying prefix interval and to keep only one failure link (right) unless no failure links in a set point into the same set.

brought together in Section 5.4.3, but for now want to come back to failure links in a tunneled XBWT.

What is still missing is the handling of failure links of removed nodes that point to nodes within the same set $V_\omega$. Note that this is by no means unnecessary because at the end of a tunnel the labels of outgoing edges might differ depending on the visited path. We therefore formulate the following inheritance lemma.

**Lemma 5.16.** *Let $\mathcal{T} = (V, E)$ be an extended trie, let $\omega \neq \varepsilon$ be a string and $c$ be a character. Furthermore, let $V_\omega$ and $V_{c\omega}$ be two set of nodes as defined in Lemma 5.15 such that all outgoing edges from nodes in $V_\omega$ point to nodes in $V_{c\omega}$:*

$$V_{c\omega} = \{\, v \in V \;\mid\; u \in V_\omega, (u, v) \in E \,\}.$$

*Let $u \in V_\omega$ and $v \in V_{c\omega}$ be two nodes with $(u, v) \in E$.*

- *If $\phi(u) \in V_\omega$ holds, then $\phi(v) \in V_{c\omega}$ and $(\phi(u), \phi(v)) \in E$ is implied.*
- *If $\phi(u) \notin V_\omega$ holds, then $\phi(v) \notin V_{c\omega}$ is implied.*

*Proof.* The label length of the failure link node $\phi(v)$ cannot be greater than the label length of the failure link node $\phi(u)$ plus one. Otherwise, as $(u, v) \in E$ implies $\lambda(v)^R = c\lambda(u)^R$, the failure link of node $u$ would not point to the node with the longest reversed proper prefix label. So clearly, the string $c\lambda(\phi(u))^R$ is the longest possible reversed label of a failure link node $\phi(v)$ of $v$.

If $\phi(u) \in V_\omega$ holds, as all outgoing edges from nodes in $V_\omega$ point to nodes in $V_{c\omega}$, there exists a node having the reversed node label $c\lambda(\phi(u))^R$. This node clearly must be the failure link node of $v$, belongs to the set $V_{c\omega}$ and is also connected with $\phi(u)$ because it is the left-extension of $\lambda(\phi(u))^R$ with character $c$, so $(\phi(u), \phi(v)) \in E$ must hold.

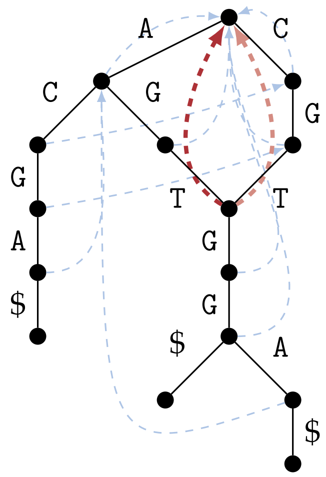
If $\phi(u) \notin V_{\omega}$ holds, there is no node in $V_{\omega}$ whose reversed label is a proper prefix of $\lambda(u)^R$. As all outgoing edges from $V_{\omega}$ point to nodes in $V_{c\omega}$, a node whose reversed label is a proper prefix of $\lambda(v)^R = c\lambda(u)^R$ cannot exist within the set $V_{c\omega}$. This holds true because all reversed labels of nodes in $V_{c\omega}$ are left-extension of reversed node labels from nodes in $V_{\omega}$ and clearly implies $\phi(v) \notin V_{c\omega}$. □

Inheritance Lemma 5.16 tells us that failure links are inherited inside of k-mer prefix intervals. More precisely, let $p_1 = (v_{1,1}, v_{1,2}, \ldots, v_{1,w})$ and $p_2 = (v_{2,1}, v_{2,2}, \ldots, v_{2,w})$ be two paths of such a prefix interval. If the first failure link of $p_1$ points into the same node set, i.e. $\phi(v_{1,1}) = v_{2,1}$, then the second failure link of $p_1$ points to the second node in the same path, i.e. $\phi(v_{2,1}) = v_{2,2}$. Applying the lemma repeatedly then shows that all failure links from $p_1$ point into $p_2$, i.e. $\phi(v_{1,i}) = v_{2,i}$ for all $i \in [1, w]$. For the other case, if the failure link of node $v_{1,1}$ points out of the k-mer prefix interval, then the failure links of the nodes $v_{1,2}, \ldots, v_{1,w}$ point out of the k-mer prefix interval.

The combination of Lemma 5.15 and 5.16 enables us to retain all failure links in a tunneled XBWT. For all nodes not belonging to a tunnel, the isolation lemma ensures that no failure links to removed nodes exist. Additionally, for each fused node in a tunnel, we keep the first failure link of the column. Note that this failure link must point out of the column, because it belongs to the node with the lexicographically smallest label. Because all failure links pointing out of a column point to the same node, this suffices to leave a tunnel correctly. We will call those links explicit.

To handle failure links inside prefix intervals, we keep all failure links of the start node of a tunnel and associate them to the incoming edges of the node. While the first such failure link is explicit, we call the remaining failure links implicit because they can point to nodes which are no longer present. We thus modify implicit failure such that they point to edges instead of nodes, namely the incoming edges of the original nodes.

The idea to emulate failure links inside of a tunnel now is as follows: when a tunnel is entered, the index of the incoming edge $e_{\text{ent}}$ and the index of the topmost incoming edge of the node $e_{\text{top}}$ is saved. In the case that we want to follow a failure link at some node in the tunnel, we distinguish between two cases. In the first case, we entered a tunnel at the topmost edge, so we simply follow the explicit failure link at this node. In the second case, we first follow the implicit failure link at edge $e_{\text{ent}}$



| $i_{\text{out}}$ | $i_{\text{in}}$ | $\widetilde{L}_X[i_{\text{out}}]$ | $D_{\text{out}}[i_{\text{out}}]$ | $D_{\text{in}}[i_{\text{in}}]$ | $\lambda(\pi[i_{\text{in}}])^R$ |
|---|---|---|---|---|---|
| 1 | 1 | A | 1 | 1 | $\varepsilon$ |
| 2 | | C | 0 | | |
| 3 | 2 | C | 1 | 1 | A |
| 4 | | G | 0 | | |
| 5 | 3 | $ | 1 | 1 | AGCA |
| 6 | 4 | $ | 1 | 1 | AGGTGC |
| 7 | 5 | G | 1 | 1 | C |
| 8 | 6 | G | 1 | 1 | CA |
| 9 | 7 | T | 1 | 1 | GA |
| 10 | 8 | T | 1 | 1 | GC |
| 11 | 9 | A | 1 | 1 | GCA |
| 12 | 10 | $ | 1 | 1 | GG··· |
| 13 | | A | 0 | | |
| 14 | 11 | G | 1 | 1 | GT··· |
| 15 | 12 | G | 1 | 1 | TGA |
| | 13 | | | 0 | TGC |
| 16 | 14 | | 1 | 1 | |

$$\widetilde{P} = ( ( ( ) ( ) ) ( ( ) ) ( ) ( ( ) ) ( ) ( ) ( ) ( ) )$$

**Figure 5.18:** Example of explicit and implicit failure links in a tunneled XBWT. Red arcs or entries in the trie or the components of the XBWT are explicit, because they belong to a node which is present in the trie. Red-white arcs or entries are implicit, because those components belong to nodes which were removed by tunneling but need to be kept in the XBWT to emulate failure links correctly.

and obtain an edge $e_{\mathsf{f}}$. If the edge $e_{\mathsf{f}}$ does point to the start node of the tunnel, we jump inside of the tunnel by setting $e_{\text{ent}} \leftarrow e_{\mathsf{f}}$. This is correct behavior because of the inheritance Lemma 5.16. If the edge $e_{\mathsf{f}}$ does not point to the start node of the tunnel, we follow the explicit failure link of the current node instead. This is correct because of the inheritance Lemma 5.16 and the isolation Lemma 5.15.

Let us translate this behavior to the tunneled XBWT. The balanced parentheses sequence $\widetilde{P}$ now contains an opening and closing parenthesis for each position in the bit-vector $D_{\text{in}}$. In case that $D_{\text{in}}[e] = 1$ holds, the failure link is explicit, otherwise implicit. $\widetilde{P}$ now is built from the prefix trie of both the explicit reversed node labels and implicit reversed node labels. Implicit here means that the nodes are no longer present because $D_{\text{in}}[e] = 0$ holds. This has the consequence that the failure links in $\widetilde{P}$ point to indices of incoming edges instead of nodes. In the case of an explicit failure link, the edge index $e_{\mathsf{f}}$ can easily be converted to the node index $i$ using $i \leftarrow \mathsf{rank}_{D_{\text{in}}}(1, e_{\mathsf{f}})$. Similarly, the uppermost entry edge of a node index $i$ can be found using $e \leftarrow \mathsf{select}_{D_{\text{in}}}(1, i)$. This allows us to handle both explicit and implicit failure links in the same manner, but we have to remember to convert edge identifiers to

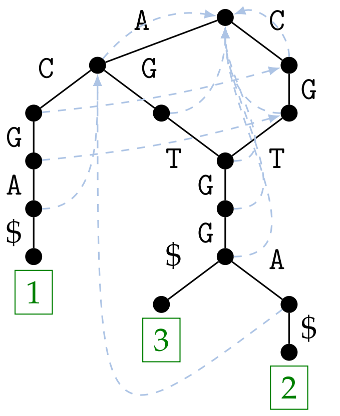
```
Data: Tunneled XBWT with components $\widetilde{L}_X$, $\widetilde{C}_X$, $D_{\text{out}}$, $D_{\text{in}}$ and $\widetilde{P}$, node index $i_v$, uppermost tunnel entry edge $e_{\text{top}}$,
      tunnel entry edge $e_{\text{ent}}$.
Result: Node index $j_v$, uppermost tunnel entry edge $e_{\text{top}}$, tunnel entry edge $e_{\text{ent}}$ of the (implicit or explicit) node
        where the failure link in the original trie points to.

function failure-link($\langle i_v, e_{\text{top}}, e_{\text{ent}} \rangle$)
                                             // check if we are in a tunnel at a non-uppermost path
    if $e_{\text{top}} \neq e_{\text{ent}}$ then
        $e_{\text{f}} \leftarrow \text{rank}_{\widetilde{P}}(''('', \text{enclose}_{\widetilde{P}}(\text{select}_{\widetilde{P}}(''('', e_{\text{ent}}))))$
        // check if implicit failure link points to a node at the start of the tunnel
        if $e_{\text{f}} \geq e_{\text{top}}$ then
            return $\langle i_v, e_{\text{top}}, e_{\text{f}} \rangle$                       // jump in the tunnel
                                             // follow explicit failure link at current node
    $e_{\text{top}} \leftarrow \text{select}_{D_{\text{in}}}(1, i_v)$
    $e_{\text{f}} \leftarrow \text{rank}_{\widetilde{P}}(''('', \text{enclose}_{\widetilde{P}}(\text{select}_{\widetilde{P}}(''('', e_{\text{top}}))))$
    $j_v \leftarrow \text{rank}_{D_{\text{in}}}(1, e_{\text{f}})$
    return $\langle j_v, \perp, \perp \rangle$
```

**Algorithm 5.14:** Failure link navigation in a tunneled XBWT with tunnels from k-mer prefix intervals. A similar algorithm appeared in a student project [RHRH20].

node identifiers in the case of explicit failure links. An example of implicit failure links, implicit node labels and the resulting balanced parentheses sequence $\widetilde{P}$ can be found in Figure 5.18. Algorithm 5.14 shows how failure links in a tunneled XBWT can be emulated.

We now want to repeat the mechanisms of failure links in a tunneled XBWT. In addition to explicit failure links for the nodes that have not been removed by tunneling, we also keep implicit failure links for the start nodes of all tunnels to emulate failure links within a tunnel. The failure links are no longer associated to nodes, but to incoming edges instead. This implies that the length of the balanced parentheses sequence $\widetilde{P}$ is reduced by tunneling. More precisely, the number of failure links in a tunneled XBWT is equal to the length of the bit-vector $D_{\text{in}}$, or put differently, to the length of the tunneled XBWT. For the sake of thoroughness, a full tunneled XBWT is depicted in Figure 5.19

It is essential to tunnel only non-overlapping prefix intervals. Otherwise, the isolation Lemma 5.15 does not hold. Moreover, it is essential to tunnel only prefix intervals where the column nodes set up a “prefix group”, i.e. the reversed node labels share a common prefix $\omega$ and no node outside of the group shares this prefix. This is essential for both the isolation Lemma 5.15 and the inheritance Lemma 5.16. The easiest way to fulfill both conditions is to tunnel k-mer prefix intervals. In Section 5.2 we have seen a way to find the best value of k such that the corresponding tunneled BWT has minimal length. Consequently, in the next section, we put all modules together and show how a tunneled XBWT can be constructed.



| $i_{out}$ | $i_{in}$ | $i_\$$ | $\widetilde{L}_X[i_{out}]$ | $D_{out}[i_{out}]$ | $D_{in}[i_{in}]$ | $\lambda(\pi[i_{in}])^R$ | $R[i_\$]$ |
|---|---|---|---|---|---|---|---|
| 1 | 1 | | A | 1 | 1 | $\varepsilon$ | |
| 2 | | | C | 0 | | | |
| 3 | 2 | | C | 1 | 1 | A | |
| 4 | | | G | 0 | | | |
| 5 | 3 | 1 | $ | 1 | 1 | AGCA | 1 |
| 6 | 4 | 2 | $ | 1 | 1 | AGGTGC | 2 |
| 7 | 5 | | G | 1 | 1 | C | |
| 8 | 6 | | G | 1 | 1 | CA | |
| 9 | 7 | | T | 1 | 1 | GA | |
| 10 | 8 | | T | 1 | 1 | GC | |
| 11 | 9 | | A | 1 | 1 | GCA | |
| 12 | 10 | 3 | $ | 1 | 1 | GG··· | 3 |
| 13 | | | A | 0 | | | |
| 14 | 11 | | G | 1 | 1 | GT··· | |
| 15 | 12 | | G | 1 | 1 | TG··· | |
| | 13 | | | | 0 | | |
| 16 | 14 | | | 1 | 1 | | |

| $c$ | $ | A | C | G | T |
|---|---|---|---|---|---|
| $\widetilde{C}_X[c]$ | 0 | 1 | 4 | 6 | 11 |

$$\widetilde{P} = (\ (\ (\ )\ (\ )\ )\ (\ (\ )\ )\ (\ )\ (\ (\ )\ )\ (\ )\ (\ )\ (\ )\ (\ )\ )$$

**Figure 5.19:** Full tunneled XBWT (right and bottom left) and its trie illustration (top left). A similar image appeared in [RHRH20].

### 5.4.3 *Construction*

The construction of a tunneled XBWT is mainly equivalent to the construction of a normal XBWT, except that nodes belonging to tunnels are removed. We will use the intermediate construction components MR and $\mathsf{cnt}_c$ from Section 5.3.1 as well as the k-mer prefix interval markings $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ from Section 5.1 for this purpose.

First of all, recall that the bit-vector MR partitions the suffix array of the string $S = S_1[1..|S_1| - 1]^R\$S_2[1..|S_2| - 1]^R\$ \cdots S_m[1..|S_m| - 1]^R\$$ into intervals where each interval corresponds to a node in the final XBWT. More precisely, the XBWT contains a node $v$ with reverse node label $\omega = \lambda(v)^R$ if and only if there exists a $\omega\$$-interval $[i, j]$ in the BWT L of $S$ where all suffixes share the common prefix $\omega\$$. In such a case, the bit-vector MR indicates the start and the end of the interval, i.e. $\mathsf{MR}[i..j+1] = 10^{j-i}1$. Figure 5.20 gives an example of the bit-vector MR of the running example.

The k-mer prefix interval marking bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ mark the start and end of a prefix interval. Let $[i, j]$ and $[i', j']$ be two k-mer intervals fulfilling the predecessor-successor relationship of a k-mer prefix interval, i.e. $[\mathsf{LF}[i], \mathsf{LF}[j]] = [i', j']$ and $\mathsf{L}[i] = \ldots = \mathsf{L}[j]$. Then $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ contain markings where a possible tunnel can start and end, that is, $D_{\mathsf{in}}[i..j] = D_{\mathsf{out}}[i'..j'] = 10^{j-i}$. An example of k-mer

| $i$ | $\mathsf{MR}[i]$ | $\mathsf{cnt}_c[i]$ | $\mathsf{L}[i]$ | $D_{\mathsf{out}}[i]$ | $D_{\mathsf{in}}[i]$ | rotations |
|---|---|---|---|---|---|---|
| 1 | 1 | 0 | A | 1 | 1 | `$A···` |
| 2 | 0 | 0 | A | 1 | 0 | `$A···` |
| 3 | 0 | 0 | C | 1 | 1 | `$C···` |
| 4 | 1 | 0 | G | 1 | 1 | `A$···` |
| 5 | 0 | 0 | C | 0 | 1 | `A$···` |
| 6 | 1 | 1 | $ | 1 | 1 | `AGCA$···` |
| 7 | 1 | 2 | $ | 1 | 1 | `AGTGGC$···` |
| 8 | 1 | 0 | G | 1 | 1 | `C$···` |
| 9 | 1 | 2 | G | 1 | 1 | `CA$···` |
| 10 | 1 | 1 | T | 1 | 1 | `GA$···` |
| 11 | 1 | 0 | T | 1 | 1 | `GC$···` |
| 12 | 1 | 2 | A | 1 | 1 | `GCA$···` |
| 13 | 1 | 1 | $ | 1 | 1 | `GGTGA$···` |
| 14 | 1 | 1 | A | 0 | 1 | `GGTGC$···` |
| 15 | 1 | 1 | G | 1 | 1 | `GTGA$···` |
| 16 | 1 | 1 | G | 0 | 0 | `GTGC$···` |
| 17 | 1 | 1 | G | 1 | 1 | `TGA$···` |
| 18 | 1 | 2 | G | 1 | 0 | `TGC$···` |
| 19 | 1 |  |  | 1 | 1 |  |

| $i_v$ | $\mathsf{L}_X[i]$ | $\lambda(\pi[i_v])^R$ |
|---|---|---|
| 1 | A | $\varepsilon$ |
|  | C |  |
| 2 | C | A |
|  | G |  |
| 3 | $ | AGCA |
| 4 | $ | AGGTGC |
| 5 | G | C |
| 6 | G | CA |
| 7 | T | GA |
| 8 | T | GC |
| 9 | A | GCA |
| 10 | $ | GGTGA |
| 11 | A | GGTGC |
| 12 | G | GTGA |
| 13 | G | GTGC |
| 14 | G | TGA |
| 15 | G | TGC |

**Figure 5.20:** Components required to construct a tunneled XBWT (left) and normal XBWT (right). The XBWT is constructed from the strings $S_1$ = `ACGA$`, $S_2$ = `CGTGGA$` and $S_3$ = `AGTGG$`. The left side shows the sorted rotations of $S$ = `AGCA$AGGTGC$GGTGA$` and contains the 2-mer prefix intervals $(6,1,4),(7,2,5)$ and $(17,15,13),(18,16,14)$. The blue prefix interval does not induce a prefix interval in the XBWT because the zero markings in $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ are placed at positions $i$ where $\mathsf{MR}[i] \neq 1$ holds. Additionally, the markings of the prefix interval $(6,1),(7,2)$ were removed by Algorithm 5.15 to ensure that no $-characters are removed by tunneling. The red prefix interval induces a prefix interval in the XBWT. The explicit failure link of the start of the interval starts at position 14, while the implicit failure link starts at position 15 in the XBWT.

intervals and markings in $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ can be found in Figure 5.20. In general, a zero in $D_{\mathsf{in}}$ indicates an edge that points to a node which is removed by tunneling. Analogously, a zero in $D_{\mathsf{out}}$ indicates an edge coming from a node which is removed by tunneling. In all other cases, the edges point to or come from nodes which "survive" the tunneling process.

The key observation is that entries $i$ in $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ where $\mathsf{MR}[i] = 1$ hold induce markings of k-mer prefix intervals in the XBWT. This means that the reduced bitvectors $D_{\mathsf{out}}[\mathsf{select}_{\mathsf{MR}}(1,j)]$ and $D_{\mathsf{in}}[\mathsf{select}_{\mathsf{MR}}(1,j)]$ with $j \in [1, \mathsf{rank}_{\mathsf{MR}}(1, |\mathsf{MR}|)]$ are suitable for tunneling an XBWT, see also Figure 5.20. To see this, we regard only k-mer intervals belonging to a prefix interval, i.e. the intervals contain zeros in either $D_{\mathsf{out}}$ or $D_{\mathsf{in}}$. Furthermore, we distinguish two cases, depending on the length of the node labels belonging to the intervals.

Let $\omega$ be a string with length $|\omega| = k$. The set $V_\omega$ of nodes whose reverse labels share the prefix $\omega$ can be associated to the $\omega$-interval in $\mathsf{L}$. The association of a left-extended node set $V_{c\omega}$ to the $c\omega$-interval in $\mathsf{L}$ follows analogously. In the case that the $\omega$- and $c\omega$-intervals are start- and end-columns of a k-mer prefix interval, then clearly, all outgoing edges from $V_\omega$ must be labeled with $c$ and point into $V_{c\omega}$. This means that a k-mer prefix interval in the XBWT is implied. Now, let $[i, j]$ be an $\omega$-interval contained in a k-mer prefix interval. Then, $D_{\mathsf{in}}[i] = \mathtt{1}$ and $\mathsf{MR}[i] = \mathtt{1}$ must hold because $i$ is the left boundary of the MR-interval associated to the node with the lexicographically smallest reversed label within $V_\omega$. For all remaining positions $l \in [i+1, j]$ with $\mathsf{MR}[l] = \mathtt{1}$, $D_{\mathsf{in}}[l] = \mathtt{0}$ holds because of the markings in the $\omega$-interval. This shows that the markings in the reduced bit-vector $D_{\mathsf{in}}$ are analogous to the normal markings. The same holds true for the reduced bit-vector $D_{\mathsf{out}}$.

The other case considers nodes $v$ whose node label $\lambda(v)$ is less than k characters long. In this case, because a BWT is viewed cyclically, we can associate multiple k-mer intervals $\omega\$x_1, \omega\$x_2, \ldots$ to the node. These k-mer intervals cannot be associated to any other node, because no node label contains the character \$. Also, these intervals span a consecutive range $[i, j]$ in $\mathsf{L}$ because of the lexicographical order. Within this range, $\mathsf{MR}[i..j] = \mathtt{10}^{j-i}$ must hold, because those entries belong to the node $v$. In case that any k-mer interval in the range is contained in a prefix interval, all except of the first entry in $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ are zero. This means that the range $[i, j]$ must start with a $\mathtt{1}$, so $D_{\mathsf{out}}[i] = D_{\mathsf{in}}[i] = \mathsf{MR}[i] = \mathtt{1}$ must hold. The node $v$ thus does not belong to a k-mer prefix interval in the XBWT. This is correct because its label has got less than k characters.

We now need to be aware of two special cases. First, tunneling of an XBWT must not remove edges that are labeled with \$. The reason is that the \$'s are required to associate a leaf in the trie to a record. Second, the nodes at a tunnel end must have an outdegree of exactly one. If this is not the case, tunneling could accidentally change the shape of the trie, because branching nodes are interpreted as tunnel ends.

To handle both cases, we make use of the prefix interval "unmarking" algorithm from Page 134. The algorithm requires an FM-index as well as two bit-vectors $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ which contain possible starts and ends of prefix intervals. A $\mathtt{10}^*\mathtt{1}$-sequence in $D_{\mathsf{in}}$ indicates a possible prefix interval start, a $\mathtt{10}^*\mathtt{1}$-sequence in $D_{\mathsf{out}}$ a possible prefix interval end. The algorithm then checks if the conditions of a prefix interval are met and if a possible prefix interval start can be associated to a prefix interval end. If this is not the case, the markings in $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ are cleared, resulting in a valid prefix interval marking.

```
Data: Strings $S_1, \dots, S_m$, BWT L in form of an FM-index of the string $S = S_1[1..|S_1|-1]^R \cdots S_m[1..|S_m|-1]^R\$$ of
      length $n$, bit-vector MR computed from Algorithm 5.9, k-mer interval boundary bit-vector $B$ computed from
      Algorithm 5.5.
Result: k-mer prefix interval markings $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ where no prefix interval containing a $ is marked and no
        conflicts between branching nodes in the trie and tunnel ends exist.

$D_{\mathsf{in}} \leftarrow B$
$D_{\mathsf{out}} \leftarrow B$
                                        // unmark possible prefix interval ends containing a $
for $i \leftarrow 1$ to $m$ do
    $D_{\mathsf{out}}[i] \leftarrow 1$
                                        // unmark possible prefix interval ends at branching nodes
                                        // computation can be integrated in MR-computation Algorithm 5.9
foreach MR-range $[i,j]$ with $\mathsf{MR}[i..j+1] = 10^{j-i}1$ and $|\mathsf{getIntervals_L}(i,j)| > 1$ do
    $D_{\mathsf{out}}[i+1] \leftarrow 1$
                                        // unmark invalid prefix interval markings
run Algorithm 5.2 with $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$
return $\langle D_{\mathsf{out}}, D_{\mathsf{in}} \rangle$
```

**Algorithm 5.15:** Computation of a k-mer prefix interval marking suitable for XBWT tunneling.

We can use this algorithm as follows: for possible prefix interval starts we use the k-mer interval boundaries bit-vector $B$ from Algorithm 5.5, analogously to the computation of k-mer prefix interval markings. For possible prefix interval ends, we use the bit-vector $B$ as base, but set additional bits that consider the special cases.

- By setting $D_{\mathsf{out}}[1..m] = 1^m$, we can ensure that no $ is contained in a prefix interval.

- Let $[i,j]$ be the MR-range of a branching node, i.e. $\mathsf{MR}[i..j+1] = 10^{j-i}1$ and $|\mathsf{getIntervals_L}(i,j)| > 1$. By setting $D_{\mathsf{out}}[i+1] = 1$, we can avoid conflicts between prefix interval ends and the branching node.

Because each prefix interval containing a $ must point through the range $[1,m]$, prefix intervals are interrupted by ones in $D_{\mathsf{out}}[1..m]$. In case of a branching node in an MR-range $[i,j]$, we note that $j - i \geq 1$. Setting $D_{\mathsf{out}}[i+1] = 1$ now has two effects: first, any k-mer prefix interval is interrupted. Second, as left-extensions of k-mer intervals coincide with ones in MR, we ensure that no additional prefix intervals are marked accidentally. Algorithm 5.15 shows how a valid prefix interval marking taking both special cases into consideration can be computed.

Using the knowledge about induced k-mer prefix intervals in an XBWT, a construction algorithm for the “shape components” $\widetilde{\mathsf{L}}$, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ can be formulated. The idea is to merge the XBWT “shape component” construction Algorithm 5.10 with the tunneling Algorithm 3.3.

**Data:** Strings $S_1, \ldots, S_m$, BWT $\mathsf{L}$ in form of an FM-index of the string $S = S_1[1..|S_1|-1]^R \cdots S_m[1..|S_m|-1]^R\$$ of length $n$, bit-vector $\mathsf{MR}$ computed from Algorithm 5.9, prefix interval markings $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ computed from Algorithm 5.15.
**Result:** Tunneled XBWT components $\widetilde{\mathsf{L}}_X$, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$.

```
initialize a string $\widetilde{\mathsf{L}}_X$ of size $\mathsf{rank}_{\mathsf{MR}}(1,n)+m-1$
$i \leftarrow 1$
$i_{\mathsf{out}} \leftarrow 1$                    // output position in $D_{\mathsf{out}}$ and $\widetilde{\mathsf{L}}_X$
$i_{\mathsf{in}} \leftarrow 1$                        // output position in $D_{\mathsf{in}}$
for $j \leftarrow 1$ to $n$ do
    if $\mathsf{MR}[j+1] = 1$ then
        $b \leftarrow D_{\mathsf{out}}[i]$
        if $D_{\mathsf{in}}[i] = 1$ then
            $M \leftarrow \mathsf{getIntervals}_{\mathsf{L}}(i,j)$
            foreach $\langle c, [lb, rb] \rangle \in M$ do
                $\widetilde{\mathsf{L}}_X[i_{\mathsf{out}}] \leftarrow c$
                $D_{\mathsf{out}}[i_{\mathsf{out}}] \leftarrow 0$
                $i_{\mathsf{out}} \leftarrow i_{\mathsf{out}} + 1$
            $D_{\mathsf{out}}[i_{\mathsf{out}} - |M|] \leftarrow b$
        if $b = 1$ then
            $D_{\mathsf{in}}[i_{\mathsf{in}}] \leftarrow D_{\mathsf{in}}[i]$
            $i_{\mathsf{in}} \leftarrow i_{\mathsf{in}} + 1$
        $i \leftarrow j+1$
trim $\widetilde{\mathsf{L}}_X$ to size $i_{\mathsf{out}} - 1$
trim $D_{\mathsf{out}}$ to size $i_{\mathsf{out}}$ and set $D_{\mathsf{out}}[i_{\mathsf{out}}] \leftarrow 1$
trim $D_{\mathsf{in}}$ to size $i_{\mathsf{in}}$ and set $D_{\mathsf{in}}[i_{\mathsf{in}}] \leftarrow 1$
return $\langle \widetilde{\mathsf{L}}_X, D_{\mathsf{out}}, D_{\mathsf{in}} \rangle$
```

**Algorithm 5.16:** Construction of the tunneled XBWT components $\widetilde{\mathsf{L}}_X$, $\widetilde{D_{\mathsf{out}}}$ and $\widetilde{D_{\mathsf{in}}}$ using the BWT $\mathsf{L}$ and the k-mer prefix interval markings $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$.

As a reminder, the tunneling algorithm requires the markings of prefix intervals in the bit-vector $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ and produces a tunneled BWT as follows. The algorithm scans the components $\mathsf{L}$, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ in a front-to-back scan.

- If $D_{\mathsf{in}}[i] = 1$ holds, the entries $\mathsf{L}[i]$ and $D_{\mathsf{out}}[i]$ are appended to $\widetilde{\mathsf{L}}$ and $\widetilde{D_{\mathsf{out}}}$.
- If $D_{\mathsf{out}}[i] = 1$ holds, the entry $D_{\mathsf{in}}[i]$ is appended to $\widetilde{D_{\mathsf{out}}}$.

Because $\widetilde{D_{\mathsf{in}}}$ and $\widetilde{D_{\mathsf{out}}}$ are shorter than $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$, $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ can be overwritten by $\widetilde{D_{\mathsf{in}}}$ and $\widetilde{D_{\mathsf{out}}}$ during this process.

The XBWT shape component construction algorithm requires the bit-vector $\mathsf{MR}$ and $\mathsf{L}$ as input and works as follows. In a front-to-back scan, each $\mathsf{MR}$-interval is enumerated. For each such interval $[i,j]$, the labels of outgoing edges of the corresponding node are determined using $M \leftarrow \mathsf{getIntervals}_{\mathsf{L}}(i,j)$. Then, the characters in the set $M$ are appended to $\mathsf{L}_X$. Additionally, the sequence $10^{|M|-1}$ is appended to a bit-vector $D_{\mathsf{out}}$.

To combine the algorithms, we first enumerate all $\mathsf{MR}$-intervals $[i,j]$ in a front-to-back scan. The labels of outgoing edges are determined using $M \leftarrow \mathsf{getIntervals}_{\mathsf{L}}(i,j)$.

- If $D_{\text{in}}[i] = 1$ holds, we append the characters in $M$ to $\widetilde{\mathsf{L}}_X$ and also append the bit $D_{\text{out}}[i]$ followed by the sequence $0^{|M|-1}$ to a bit-vector $\widetilde{D_{\text{out}}}$.
- If $D_{\text{out}}[i] = 1$ holds, we append the bit $D_{\text{in}}[i]$ to a bit-vector $\widetilde{D_{\text{in}}}$.

Analogous to the tunneling algorithm, $D_{\text{in}}$ and $D_{\text{out}}$ can be overwritten by $\widetilde{D_{\text{in}}}$ and $\widetilde{D_{\text{out}}}$ during the process. Algorithm 5.16 shows the construction in full detail, requiring $O(n \log \sigma)$ worst-case time.

We now come to the construction of the balanced parentheses sequence $\widetilde{P}$ used for failure link support. The original sequence $P$ for a normal XBWT is constructed with the support of $\mathsf{MR}$ and the counter array $\mathsf{cnt}_c$. Using a front-to-back scan, at each position $i$ exactly $\mathsf{MR}[i]$ opening and $\mathsf{cnt}_c[i]$ closing parentheses are appended to the sequence $P$. Note that the $i$-th opening parenthesis in $P$ refers to the $i$-th node in the XBWT, or equivalently, to the $i$-th set bit in $\mathsf{MR}$.

In a tunneled XBWT, we have to store both explicit and implicit failure links. Explicit failure links come from nodes which are not removed by tunneling. Let $D_{\text{in}}$ and $D_{\text{out}}$ be the prefix interval markings computed by Algorithm 5.15. Explicit failure links then refer to entries $i$ where $\mathsf{MR}[i] = D_{\text{out}}[i] = D_{\text{in}}[i] = 1$ holds. Implicit failure links are failure links from nodes at the start of a tunnel which are removed by tunneling. These links refer to entries $i$ where $\mathsf{MR}[i] = D_{\text{out}}[i] = 1$ hold (start of a tunnel), but $D_{\text{in}}[i] = 0$ holds because the node is removed later, see Figure 5.20.

Using this knowledge, an easy construction of $\widetilde{P}$ would be as follows. We use the original sequence of $P$ as a base. For each $i$-th set bit in $\mathsf{MR}$ we check if the bit in $D_{\text{out}}$ is cleared, i.e. $D_{\text{out}}[\mathsf{select}_{\mathsf{MR}}(1, i)] = 0$. In this case, the corresponding failure link is neither explicit nor implicit, so we remove the opening parenthesis $\mathsf{select}_P(''('', i)$ as well as the closing parenthesis $\mathsf{findclose}_P(\mathsf{select}_P(''('', i))$ from $P$. However, this approach has a problem: we first need to construct $P$, then initialize balanced parentheses support, construct $\widetilde{P}$ and then initialize balanced parentheses support once more for $\widetilde{P}$.

A direct approach is possible because of the isolation Lemma 5.15 from the last section. This lemma states that no failure link from outside can point into a set $V_\omega$ of nodes. This implies that the failure links pointing out of $V_\omega$ induce a balanced parentheses subsequence inside $P$. More precisely, all parentheses belonging to such failure links form a consecutive subsequence in $P$.

This allows us to simplify the construction of $\widetilde{P}$. Analogous to the normal construction, we use a front-to-back scan and the components $\mathsf{MR}$ and $\mathsf{cnt}_c$. In contrast to the normal construction, when we reach an entry $i$ with $\mathsf{MR}[i] = 1$ and $D_{\text{out}}[i] = 0$, we

```
Data: Bit-vector MR and counter array $\mathsf{cnt}_c$ computed from Algorithm 5.9, k-mer prefix interval marking $D_{\mathsf{out}}$
      computed in Algorithm 5.15.
Result: Tunneled XBWT failure link component $\widetilde{P}$.

initialize a parentheses sequence $P$ of size $2 \cdot \mathsf{rank}_{\mathsf{MR}}(1, n)$
$k \leftarrow 1$
$skip \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
                                      // write opening parenthesis in case of explicit or implicit failure link
    if $\mathsf{MR}[i] = 1$ then
        if $D_{\mathsf{out}}[i] = 1$ then
            $\widetilde{P}[k] \leftarrow$ ”(”
            $k \leftarrow k + 1$
        else
            $skip \leftarrow skip + 1$
                                                                               // write closing parentheses
    if $\mathsf{cnt}_c[i] > skip$ then
        for $j \leftarrow 1$ to $\mathsf{cnt}_c[i] - skip$ do
            $\widetilde{P}[k] \leftarrow$ ”)”
            $k \leftarrow k + 1$
        $skip \leftarrow 0$
    else
        $skip \leftarrow skip - \mathsf{cnt}_c[i]$
trim $\widetilde{P}$ to size $k$
return $\widetilde{P}$
```

**Algorithm 5.17:** Construction of the tunneled XBWT component $\widetilde{P}$ using the components MR and $\mathsf{cnt}_c$ computed from XBWT construction and the k-mer prefix interval marking $D_{\mathsf{out}}$. A similar algorithm appeared in a student project [RHRH20].

do not append an opening parentheses to $\widetilde{P}$. Instead, we increment a variable *skip*. This variable then is used to skip the appendage of the next *skip* closing parentheses. In case that $\mathsf{cnt}_c[i]$ is greater than *skip*, we append exactly $\mathsf{cnt}_c[i] - skip$ closing parentheses to $\widetilde{P}$ and set *skip* to zero. If $\mathsf{cnt}_c[i]$ is lower than or equal to the value of *skip*, we subtract $\mathsf{cnt}_c[i]$ from the variable *skip*. This ensures that only the topmost failure link of a fused column in a prefix interval remains in $\widetilde{P}$. The construction requires $O(n)$ worst-case run-time and is shown in Algorithm 5.17.

We want to note that the construction of $\widetilde{P}$ should be performed before the construction of the shape components $\widetilde{\mathsf{L}}_X$, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ because the construction of the shape components overwrites the prefix interval markings in $D_{\mathsf{out}}$. The remaining components $\widetilde{\mathsf{C}}_X$ and the record array $R$ can be constructed similarly to their construction in a normal XBWT.

**Corollary 5.17.** *Let $S_1, \ldots, S_m$ be a set of null-terminated strings and let $S$ be a string of length $n$ defined as $S := S_1[1..|S_1| - 1]^R\$ \cdots S_m[1..|S_m| - 1]^R\$$. A tunneled XBWT of the extended trie $\mathcal{T}_{\mathsf{ex}}(S_1, \ldots, S_m)$ can be constructed in $O(n \log \sigma + m \log m)$ time with the following steps:*

1. *Suffix array construction and FM-index construction of $S$ in $O(n \log \sigma)$ time.*
2. *Construction of* $\mathsf{MR}$ *and* $\mathsf{cnt}_c$ *with Algorithm 5.9 in $O(n \log \sigma)$ time.*
3. *Construction of the k-mer prefix interval markings $D_{\mathsf{in}}$ and $D_{\mathsf{out}}$ with the Algorithms 5.5, 5.2 and 5.15 in $O(n \log \sigma)$ time using* $\mathsf{MR}$ *and the FM-index of $S$.*
4. *Construction of the balanced parentheses sequence $\widetilde{P}$ with Algorithm 5.17 in $O(n)$ time using the arrays* $\mathsf{cnt}_c$ *and* $\mathsf{MR}$.
5. *Construction of the shape components $\widetilde{\mathsf{L}}_X$, $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$ with Algorithm 5.16 in $O(n \log \sigma)$ time using the FM-index,* $\mathsf{MR}$ *and the markings in $D_{\mathsf{out}}$ and $D_{\mathsf{in}}$.*
6. *Construction of the $\widetilde{\mathsf{C}}_X$ array using $\widetilde{\mathsf{L}}_X$ in $O(n)$ time.*
7. *Construction of the record array $R$ using Algorithm 5.12 and* $\mathsf{SA}[1..m]$ *in $O(m \log m)$ time.*

Unfortunately, we cannot make statements about the optimality of the tunneling strategy in tries. The reason is that we cannot tunnel all k-mer prefix intervals contained in the normal BWT. For nodes with a label length which is less than k, the trie itself compresses the input because of the tree structure. Furthermore, it is not possible to tunnel "cyclic" k-mer prefix intervals, that is, prefix intervals containing a \$. Finally we cannot tunnel prefix intervals ending at a branching node in the trie to avoid conflicts between tunnel ends and branching nodes. However, compared to tunneled FM-indices from Section 5.2.3, the overhead of tunneling consists only of the additional $D_{\mathsf{in}}$ component. Furthermore, the number of failure links is reduced by the number of removed edges in the trie, enabling compression within this component. In the next section, we will examine the compression using real-world data.

### 5.4.4 *Experimental results*

We have seen that the concepts of tunneling and trie representation using a BWT variant harmonize. Now it is time to see whether tunneling provides good compression results. Note that we have already shown results for tunneled XBWT construction in Section 5.3.2. For further information about construction performance, we refer to the Figures C.6 and C.7.

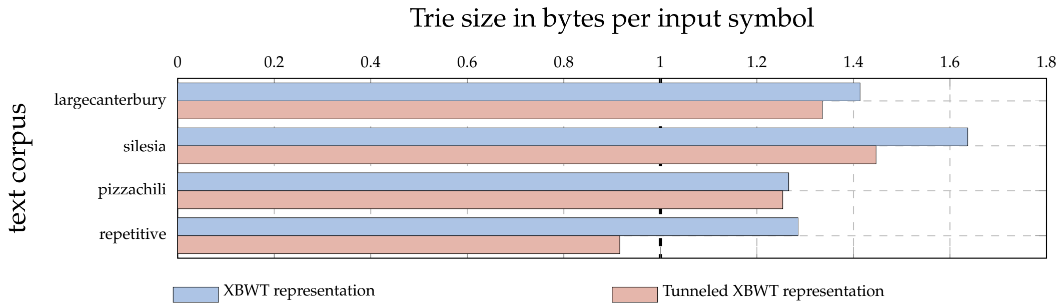


**Figure 5.21:** Average sizes of extended trie representations in bytes per symbol, grouped by text corpora. A trie size with less than 1 byte per character indicates that the trie compresses the input. Most representations do not compress the input because they contain additional components to support failure links and record mapping. The full underlying data can be found in Figure C.5.

We used the same input data as described in Section 5.3.2. The implementation of our algorithms is publicly available [Bai20]. For each file listed in Chapter A, we removed empty lines, lines containing a nullbyte and lines with less than 10 characters. Additionally, we removed lines containing other lines as proper substring to ensure correctness of the Aho-Corasick algorithm. From such modified files we used only files with more than 1 MB of size and more than 1000 lines to ensure a certain branching factor and size of the resulting trie. More details on the used input data and results for each file can be found in Section C.2.

Figure 5.21 shows the sizes of a normal XBWT compared to a tunneled XBWT. In general, tunneling reduces the representation size by an amount of approximately 10 %. The best compression results are achieved for repetitive data, i.e. collections of similar strings. In these cases, the average trie representation size could be reduced by about 1/3 using tunneling. However, we want to emphasize that tunneling does not always compress the representation. In some cases, the size of a tunneled XBWT is larger than the size of a normal XBWT. In accordance with Section 5.2.3, the number of reduced edges using the de Bruijn edge minimization is too small to overcome the space penalty of the additional component $D_{\text{out}}$. Those cases can be seen on Page 223.

We also compared the data throughput of a multi-pattern search using the Aho-Corasick algorithm [AC75] with the different representations. The method can be described as follows: for each test file, we constructed a XBWT and a TXBWT. The resulting tries then were used to find all occurrences of their contained strings within

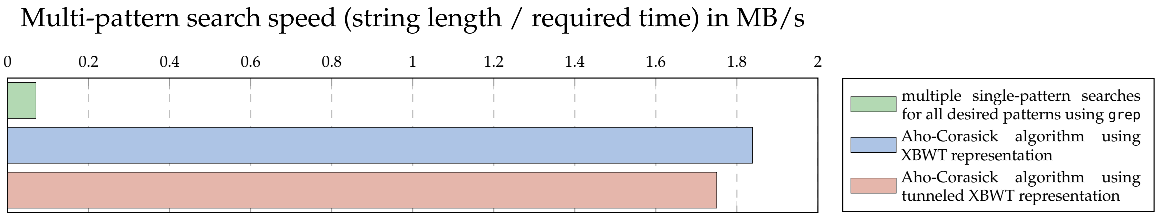


**Figure 5.22:** Average speed of the multi-pattern search using the Aho-Corasick algorithm compared to multiple single-pattern searches for all patterns using the linux command `grep`. The Aho-Corasick algorithm uses a priori constructed tries of the test files and searches for all occurrences of the test file lines within the same test file. The full underlying data can be found in Figure C.8.

the test file itself. We removed newline characters from the test file before the search, so the number of occurrences is higher than the number of strings in the trie.

Figure 5.22 shows the average data throughput in this multi-pattern search scenario. The multi-pattern search using a tunneled XBWT is about 5 % slower than one with a normal XBWT, mainly because the navigational and failure link operations are more complex. To compare the speed with other pattern search programs, we included the popular linux command `grep`.[3] We simulated a search for all patterns by executing `grep` once with the last line of each test file as pattern, and multiplied this time by the number of lines in the test file. This approximation was necessary because multiple `grep` executions with a large number of patterns would take an incredibly long time. We did not use the multi-pattern search feature of `grep` because the required memory exceeded our RAM size in the case of a large number of patterns.

Unsurprisingly, the multi-pattern search with XBWT variants is much faster than `grep` if the number of patterns is large. In the case that the number of patterns is low (e.g. less than 1000), it is sometimes beneficial to use `grep` because a single `grep` execution is very fast, and Figure 5.22 does not include the extra time of trie construction. More information about data throughput for each test file can be found in Figure C.8.

As a conclusion, we can say that tunneling an XBWT is beneficial if the underlying data is both big and repetitive. An XBWT is one of the most succinct representations of a dictionary of strings [MP+16], and tunneling allows one to compress this representation even more. The disadvantage in timings of navigational and failure link operations can be neglected because of the high compression rates. It might be

[3] https://www.gnu.org/software/grep/manual/grep.html

worth to develop specialized tunnel planning strategies for an XBWT. Presently, this work proposed the first approach to tunnel an XBWT. We did not find arguments for tunnel planning optimality yet, but state this as an open problem. Moreover, tunneling an XBWT is interesting in its own right, because it converts a trie into a word graph which fully emulates the trie. Therefore, we think that trie tunneling is a concept with high potential.

# 6 CONCLUSION

We proposed tunneling as a new compression scheme for the Burrows-Wheeler transform. Originally, we presented tunneling for the purpose of data compression [Bai18]. Then, Alanko et al. extended tunneling to Wheeler graphs [Ala+19]. This allows tunneling to be applied not only in pure data compression but also in fields of sequence analysis.

Our main contribution to the theory of tunneling (Chapter 3) is the proposal of criteria for overlappings between prefix intervals to be tunneled. Moreover, we presented complexity results to the Wheeler graph prefix interval cover problem. This shows that tunnel planning is difficult to be solved exactly. Additionally, it turns out that tunnel planning is difficult to be solved approximately.

It might be interesting to obtain more complexity results on tunnel planning. It would be especially interesting to know results when only overlayable prefix intervals are allowed to be tunneled. Also, it would be important to know more about the complexity situation for the restricted class of run-terminated length-maximal prefix intervals. We suppose that the planning problems remain NP-hard in these special situations, but gave no results and therefore state this as an open problem.

In the second part of this thesis, we presented tunneling in the field of data compression. We presented methods to trim the auxiliary tunnel information to a size relative to the number of runs, significantly reducing the tunneling costs. Using a cost model, we were able to devise heuristics which were able to select a good choice of prefix intervals to be tunneled. As a result, we were able to improve the compression rate of two state-of-the-art BWT compressors by about 9% on average and up to 55% in the best cases. It was shown that tunneling has the biggest compression impact on very big and repetitive files.

It may be possible to improve the compression rates even more. However, we experimentally showed that our tunnel strategies achieves results which are close to the optimum. Therefore, we assume that more sophisticated strategies will only show small improvements. The main problem of BWT-based compressors is decompression speed and decompression memory peak, which was not addressed in this thesis. However, we showed that the compressors are competitive to modern state-of-the-art compressors using tunneling.

An interesting application of run-terminated length-maximal prefix intervals could be given in the compression of run-length encoded wavelet trees [MN05]. During the data compression conference 2020 (which took place virtually for the first time), Dominik Köppl stated

> [...] I was wondering in the experiments whether this compressed FM-index can be put into relation with the run-length compressed FM-index, as the run-length compression is kind of similar (it needs also additional bit vectors). [Köp20]

Combining the special prefix interval class with the run-length encoded FM-index, it should be possible to reduce the length of the two additionally used bit-vectors. Associating this index with the new development of the r-index [GNP18] could end up in a very good compression for repetitive inputs.

Currently, the most promising application of tunneling can be found in the field of sequence analysis. We presented the de Bruijn graph edge minimization problem and showed a deep connection between the problem and tunneling of the special class of k-mer prefix intervals. We also presented an efficient algorithm to solve the problem. As experiments have shown, this allows one to reduce the BWT length by an average of 80% for repetitive inputs. Unfortunately, it has been shown that this approach is not suitable for pure data compression.

However, it has been shown that this tunneling strategy is suitable for the use in sequence analysis. We presented a BWT-based representation of tries, which among other representations offers the best dictionary compression rates to date [MP+16]. Our contribution to tries was the presentation of fast and memory-saving construction methods. Moreover, we presented algorithms to tunnel tries using the de Bruijn graph edge minimization approach. Though it is not surprising that trie tunneling is possible (tries can be expressed as Wheeler graphs [GMS17]), we showed that failure links can be retained in tunneled tries. A key to this result was the use of the special class of k-mer prefix intervals.

Failure links allow one to perform efficient multi-pattern matching with the Aho-Corasick algorithm [AC75]. We conducted experiments showing that the pattern search speed remains almost unchanged when using tunneled instead of normal tries. We were not able to achieve the same compression improvements as in the de Bruijn graph edge minimization problem. This is due to the circumstance that we had to modify the input data to ensure that the simple version of the Aho-Corasick algorithm works correctly. Within repetitive data, tunneled tries were around 30% smaller than the normal ones, underlining the good compression rates of tunneling on repetitive data. We also presented a way to extend BWT-based tries with a structure that allows one to execute the Aho-Corasick algorithm with arbitrary patterns. It could be interesting to implement and test this approach, but we refrained from doing so because we wanted to keep the implementation simple.

Recently, it has been shown that a multi-string BWT or context-tree based structures allows for good sequence predictions [Gue+15; Kti+19]. The basic idea is to use the multi-string BWT to predict probabilities for the next character by counting the character frequencies in the BWT within a $\omega$-interval. This can be done efficiently with the help of the intervalsymbols function of a wavelet tree. The context-tree structure can be represented by a sequence of balanced parentheses, see e.g. [Ohl13, Section 6.3].

This form of context-tree representation is quite similar to the BWT-based representation of tries. We have seen that trie tunneling leaves failure links intact, so it is very likely that it does the same for the context-tree structure. The de Bruijn graph edge minimization tunneling approach reduces edges if and only if a certain context is preceded by the same character. Removing this characters does not influence the relative probability of the character within the context. Therefore, we suppose that tunneling allows one to compress the "knowledge base" for sequence prediction.

It may be possible to use this compressed knowledge base for other data compression methods like prediction by partial matching [CW84] or context-tree weighting [WST95]. Both methods are known to offer good compression, but are limited by the memory consumption of their used context tree. Using adaptions of trie tunneling, it could be possible to use a large context tree for these methods. The problem of static BWT-based structures could be solved by computing pseudo-random substring samples of the string which cover e.g. about 20% of the string to be compressed. Creating a BWT-based knowledge base from these samples then allows one to compress the remaining 80% of the string with the original methods. However, we have not implemented such a method yet, but propose this as a future project.

In conclusion, we showed that tunneling is a useful method to compress data and to compress data structures. In the age of "big data", the need for compression methods of large and very repetitive data sets is increasing. As experiments show, this is the main strength of tunneling. Tunneling is also applicable in sequence analysis, which is useful when the data should be processed.

# BIBLIOGRAPHY

[Abe10] Jürgen Abel. "Post BWT Stages of the Burrows–Wheeler Compression Algorithm." In: *Software Practice and Experience* 40.9 (2010), pp. 751–777.

[AKO04] Mohamed I. Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. "Replacing suffix trees with enhanced suffix arrays." In: *Journal of Discrete Algorithms* 2.1 (2004), pp. 53–86.

[AC75] Alfred V. Aho and Margaret J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search." In: *Communications of the ACM* 18.6 (1975), pp. 333–340.

[Ala+19] Jarno Alanko, Travis Gagie, Gonzalo Navarro, and Louisa Seelbach Benkner. "Tunneling on Wheeler Graphs." In: *Proceedings of the 2019 Data Compression Conference*. DCC '19. 2019, pp. 122–131.

[Bai16] Uwe Baier. "Linear-time Suffix Sorting - A New Approach for Suffix Array Construction." In: *Annual Symposium on Combinatorial Pattern Matching*. CPM '16. 2016, 23:1–23:12.

[Bai18] Uwe Baier. "On Undetected Redundancy in the Burrows-Wheeler Transform." In: *Annual Symposium on Combinatorial Pattern Matching*. CPM '18. 2018, 3:1–3:15.

[Bai20] Uwe Baier. *BWT Tunneling API*. `https://github.com/waYne1337/BWT-Tunneling`. Last visited July 2020. 2020.

[BBO16] Uwe Baier, Timo Beller, and Enno Ohlebusch. "Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform." In: *Bioinformatics* 32.4 (2016), pp. 497–504.

[BBO17] Uwe Baier, Timo Beller, and Enno Ohlebusch. "Space-Efficient Parallel Construction of Succinct Representations of Suffix Tree Topologies." In: *Journal of Experimental Algorithmics* 22.1 (2017), 1.1:1–1.1:26.

[BD19] Uwe Baier and Kadir Dede. "BWT Tunnel Planning is Hard But Manageable." In: *Proceedings of the 2019 Data Compression Conference*. DCC '19. © 2019 IEEE. 2019, pp. 142–151.

[Bai+20] Uwe Baier, Thomas Büchler, Enno Ohlebusch, and Pascal Weber. "Edge minimization in de Bruijn graphs." In: *Proceedings of the 2020 Data Compression Conference*. DCC '20. © 2020 IEEE. 2020, pp. 223–232.

[Bel14] Djamal Belazzougui. "Linear time construction of compressed text indices in compact space." In: *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*. STOC '14. 2014, pp. 148–193.

[Bel17a] Timo Beller. "Space-efficient construction and applications of basic data structures in full text indexing." PhD thesis. University of Ulm, 2017.

[BBO12] Timo Beller, Katharina Berger, and Enno Ohlebusch. "Space-Efficient Computation of Maximal and Supermaximal Repeats in Genome Sequences." In: *Proceedings of the 19th International Symposium on String Processing and Information Retrieval*. SPIRE '12. 2012, pp. 99–110.

[Bel17b] David Belson. *State of the Internet Report Q1 2017*. Tech. rep. 1. akamai, 2017.

[BD97] Piotr Berman and Bhaskar DasGupta. "Complexities of Efficient Solutions of Rectilinear Polygon Cover Problems." In: *Algorithmica* 17.4 (1997), pp. 331–356.

[BW94] Michael Burrows and David J. Wheeler. *A block-sorting lossless data compression algorithm*. Tech. rep. 124. Digital Equipment Corporation, 1994.

[CN09a] Christopher Clapham and James Nicholson. *The Concise Oxford Dictionary of Mathematics*. 4th ed. Oxford University Press, 2009.

[CN09b] Francisco Claude and Gonzalo Navarro. "Practical Rank/Select Queries over Arbitrary Sequences." In: *Proceedings of the 2009 String Processing and Information Retrieval Conference*. SPIRE '09. 2009, pp. 176–187.

[Cla00] Clay Mathematics Institute. *Millenium Problems*. `http://www.claymath.org/millennium-problems`. Last visited December 2019. 2000.

[CW84] John G. Cleary and Ian H. Witten. "Data Compression Using Adaptive Coding and Partial String Matching." In: *IEEE Transactions on Communications* 32.4 (1984), pp. 396–402.

[Col10] Lasse Collin. *XZ Utils*. `https://tukaani.org/xz/`. Last visited June 2020. 2010.

[Coo71] Stephen A. Cook. “The Complexity of Theorem-proving Procedures.” In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. 1971, pp. 151–158.

[CR94] Joseph C. Culberson and Robert A. Reckhow. “Covering Polygons Is Hard.” In: *Journal of Algorithms* 17.1 (1994), pp. 2–44.

[CAB19] Fabio Cunial, Jarno Alanko, and Djamal Belazzougui. “A framework for space-efficient variable-order Markov models.” In: *Bioinformatics* 35.22 (2019), pp. 4607–4616.

[DeB46] Nicolas G. DeBruijn. “A Combinatorial Problem.” In: *Koninklijke Nederlandse Akademie V. Wetenschappe* 49 (1946), pp. 758–764.

[Ded18] Kadir Dede. “Blockwahl beim Tunneln von Burrows Wheeler Transformationen.” Elaboration of Project Algorithm Engineering 2018 (draft by Uwe Baier). 2018.

[Dre07] Ulrich Drepper. *What every programmer should know about memory*. `https://people.freebsd.org/~lstewart/articles/cpumemory.pdf`. Last visited April 2020. 2007.

[Egi+19] Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. “External memory BWT and LCP computation for sequence collections with applications.” In: *Algorithms for Molecular Biology* 14.6 (2019).

[Fan49] Robert M. Fano. *The Transmission of Information*. Tech. rep. 65. Massachusetts Institute of Technology, 1949.

[Fer13] Henning Fernau. *Lecture slides of the data compression course*. `https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/ss_2013/vorlesung01.pdf`. Last visited June 2020. 2013.

[FM05] Paolo Ferragina and Giovanni Manzini. “Indexing Compressed Text.” In: *Journal of the ACM* 52.4 (2005), pp. 552–581.

[Fer+05] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and Senthilmurugan Muthukrishnan. “Structuring Labeled Trees for Optimal Succinctness, and Beyond.” In: *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*. FOCS ’05. 2005, pp. 184–196.

[FK17] Johannes Fischer and Florian Kurpicz. "Dismantling DivSufSort." In: *Proceedings of the Prague Stringology Conference 2017*. PSC '17. 2017, pp. 62–76.

[Fre60] Edward Fredkin. "Trie Memory." In: *Communications of the ACM* 3.9 (1960), pp. 490–499.

[GMS17] Travis Gagie, Giovanni Manzini, and Jouni Sirén. "Wheeler graphs: A framework for BWT-based data structures." In: *Theoretical Computer Science* 698 (2017), pp. 67–78.

[GNP18] 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 '18. 2018, pp. 1459–1477.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[Gar06] Sachin Garg. *64-bit Range Coding and Arithmetic Coding*. `http://www.sachingarg.com/compression/entropy_coding/64bit`. Last visited June 2020. 2006.

[Gea+06] Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. "A simple optimal representation for balanced parentheses." In: *Theoretical Computer Science* 368.3 (2006), pp. 231–246.

[Gen] Genomics England. *The* 100, 000 *Genomes Project*.
`https://www.genomicsengland.co.uk/about-genomics-england/the-100000-genomes-project/`
. Last visited June 2020.

[GT19] Daniel Gibney and Sharma V. Thankachan. "On the Hardness and Inapproximability of Recognizing Wheeler Graphs." In: *27th Annual European Symposium on Algorithms*. ESA '19. 2019, 51:1–51:16.

[Gog07] Simon Gog. *SDSL lite*. `https://github.com/simongog/sdsl-lite`. Last visited February 2020. 2007.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. "High-order Entropy-compressed Text Indexes." In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '03. 2003, pp. 841–850.

[Gue+15] Ted Gueniche, Philippe Fournier-Viger, Rajeev Raman, and Vincent S. Tseng. "CPT+: Decreasing the Time/Space Complexity of the Compact Prediction Tree." In: *Advances in Knowledge Discovery and Data Mining - 19th Pacific-Asia Conference, PAKDD 2015, Ho Chi Minh City, Vietnam, May 19-22, 2015, Proceedings, Part II*. PAKDD '15. 2015, pp. 625–636.

[HLL07] Laura Heinrich-Litan and Marco E. Lübbecke. "Rectangle Covers Revisited Computationally." In: *Journal of Experimental Algorithmics* 11.2.6 (2007), pp. 55–66.

[Hir05] Jorge E. Hirsch. "An Index to Quantify An Individual's Scientific Research Output." In: *Proceedings of the National Academy of Sciences of the United States of America* 102.46 (2005), pp. 16569–16572.

[HSS03] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. "Breaking a Time-and-Space Barrier in Constructing Full-Text Indices." In: *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '03. 2003, pp. 251–260.

[Huf52] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes." In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[HMB06] Marcus Hutter, Matt Mahoney, and Jim Bowery. *The Hutter Prize*. http://prize.hutter1.net/. Benchmark results on http://mattmahoney.net/dc/text.html. 2006.

[IW95] Ramana M. Idury and Michael S. Waterman. "A New Algorithm for DNA Sequence Assembly." In: *Journal of Computational Biology* 2.2 (1995), pp. 291–306.

[Iqb+12] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. "De novo assembly and genotyping of variants using colored De Bruijn graphs." In: *Nature genetics* 44 (2012), pp. 226–32.

[Jac89] Guy Jacobson. "Space-efficient Static Trees and Graphs." In: *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*. SFCS '89. 1989, pp. 549–554.

[Jen06] Johan L. W. V. Jensen. "Sur les fonctions convexes et les inégalités entre les valeurs moyennes." In: *Acta Mathematica* 30.1 (1906), pp. 175–193.

[Joh87] David S. Johnson. "The NP-completeness column: An ongoing guide." In: *Journal of Algorithms* 8.3 (1987), pp. 438–448.

[Kan84] Immanuel Kant. "Beantwortung der Frage: Was ist Aufklärung?" In: *Berlinische Monatsschrift* (1784), pp. 481–494.

[KKP12] Juha Kärkkainen, Dominik Kempa, and Simon J. Puglisi. "Slashing the Time for BWT Inversion." In: *Proceedings of the 2012 Data Compression Conference*. DCC '12. 2012, pp. 99–108.

[KS03] Juha Kärkkäinen and Peter Sanders. "Simple Linear Work Suffix Array Construction." In: *Proceedings of the 30th International Conference on Automata, Languages and Programming*. ICALP '03. 2003, pp. 943–955.

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems." In: *Complexity of Computer Computations* 40 (1972), pp. 85–103.

[Kas+01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications." In: *Proceedings of the 12th Annual Conference on Combinatorial Pattern Matching*. CPM '01. 2001, pp. 181–192.

[Kim+03] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. "Linear-time Construction of Suffix Arrays." In: *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*. CPM '03. 2003, pp. 186–199.

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. "Fast Pattern Matching in Strings." In: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350.

[KA03] Pang Ko and Srinivas Aluru. "Space Efficient Linear Time Construction of Suffix Arrays." In: *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*. CPM '03. 2003, pp. 200–210.

[Köp20] Dominik Köppl. *DCC '20 Sigport video comment*. https://sigport.org/documents/edge-minimization-de-bruijn-graphs. Last visited June 2020. 2020.

[Kti+19] Rafael Ktistakis, Philippe Fournier-Viger, Simon J. Puglisi, and Rajeev Raman. "Succinct BWT-Based Sequence Prediction." In: *Database and Expert Systems Applications*. DEXA '19. 2019, pp. 91–101.

[LD10] Heng Li and Richard Durbin. "Fast and accurate long-read alignment with Burrows–Wheeler transform." In: *Bioinformatics* 26.5 (2010), pp. 589–595.

[Mah09] Matt Mahoney. *ZPAQ*. http://mattmahoney.net/dc/zpaq.html. Last visited June 2020. 2009.

[Mah05] Matthew V. Mahoney. *Adaptive Weighing of Context Models for Lossless Data Compression*. Tech. rep. CS-2005-16. Florida Institute of Technology, 2005.

[MN05] Veli Mäkinen and Gonzalo Navarro. "Succinct Suffix Arrays Based on Run-length Encoding." In: *Nordic Journal of Computing* 12.1 (2005), pp. 40–66.

[MM93] Udi Manber and Gene Myers. "Suffix Arrays: A New Method for On-Line String Searches." In: *SIAM Journal on Computing* 5 (1993), pp. 935–948.

[Man16] Giovanni Manzini. "XBWT Tricks." In: *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval*. SPIRE '16. 2016, pp. 80–92.

[Man19] Giovanni Manzini. *The Past and the Future of an Unusual Compressor*. https://www.cs.brandeis.edu/~dcc/Programs/Program2019InvitedPresentation.pdf . Invited presentation of the 2019 Data Compression Conference. 2019.

[MLS14] Shoshana Marcus, Hayan Lee, and Michael C. Schatz. "SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips." In: *Bioinformatics* 30.24 (2014), pp. 3476–3483.

[MP+16] Miguel A. Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. "Practical compressed string dictionaries." In: *Information Systems* 56 (2016), pp. 73–108.

[Mas78] William J. Masek. *Some NP-complete set covering problems*. Unpublished manuscript. 1978.

[Mor03] Yuta Mori. *divsufsort*. https://github.com/y-256/libdivsufsort. Last visited November 2019. 2003.

[Mur16] Ilya Muravyov. *BCM*. https://github.com/encode84/bcm. Last visited June 2020. 2016.

[Na05] Joong C. Na. “Linear-Time Construction of Compressed Suffix Arrays Using O(N Log N)-bit Working Space for Large Alphabets.” In: *Proceedings of the 16th Annual Conference on Combinatorial Pattern Matching*. CPM ’05. 2005, pp. 57–67.

[Nav14] Gonzalo Navarro. “Wavelet trees for all.” In: *Journal of Discrete Algorithms* 25 (2014), pp. 2–20.

[NZC09] Ge Nong, Sen Zhang, and Wai Hong Chan. “Linear Suffix Array Construction by Almost Pure Induced-Sorting.” In: *Proceedings of the 2009 Data Compression Conference*. DCC ’09. 2009, pp. 193–202.

[Nor06] Marc Norton. *Optimizing pattern matching for intrusion detection*. http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf. Last visited July 2020. 2006.

[Ohl13] Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

[OSB18] Enno Ohlebusch, Stefan Stauß, and Uwe Baier. “Trickier XBWT Tricks.” In: *String Processing and Information Retrieval*. SPIRE ’18. 2018, pp. 325–333.

[Oht82] Tatsuo Ohtsuki. “Minimum dissection of rectilinear regions.” In: *Proceedings 1982 IEEE Symposium on Circuits and Systems*. IEEE. 1982, pp. 1210–1213.

[Pap94] Christos H. Papadimitriou. *Computational Complexit*. Addison-Wesley, 1994.

[PST07] Simon J. Puglisi, William F. Smyth, and Andrew H. Turpin. “A Taxonomy of Suffix Array Construction Algorithms.” In: *ACM Computing Surveys* 39.2 (2007).

[Rät19] Caroline Räther. “Heuristic for Tunneled BWT Block Choice.” Elaboration of Project Algorithm Engineering 2018 (draft by Uwe Baier). 2019.

[Res15] The International Genome Sample Resource. *The* 1000 *Genomes Project*. https://www.internationalgenome.org/. Last visited June 2020. 2015.

[RHRH20] Sebastian Reyes Häusler and Valentin Reyes Häusler. "Getunnelte eXtended Burrows Wheeler Transformation." Elaboration of Project Algorithm Engineering 2019 (draft by Uwe Baier). 2020.

[RL79] Jorma J. Rissanen and Glen G. Langdon. "Arithmetic coding." In: *IBM Journal of Research and Development* 23 (1979), pp. 149–162.

[Rya80] B. Ya Ryabko. "Data compression by means of a "book stack"." In: *Problems of Information Transmission* 16 (1980), pp. 265–269.

[SN10] Kunihiko Sadakane and Gonzalo Navarro. "Fully-Functional Succinct Trees." In: *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '10. 2010, pp. 134–149.

[Sew96] Julian Seward. *bzip2 file compressor*. http://bzip.org/. Last visited November 2019. 1996.

[Sha48] Claude E. Shannon. "A Mathematical Theory of Communication." In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[Shi11] Andy Shipsides. *HD Formats: Bit Rate vs Bit Depth*. https://www.abelcine.com/articles/blog-and-knowledge/tutorials-and-guides/hd-formats-bit-rate-vs-bit-depth. Last visited June 2020. 2011.

[Sti30] James Stirling. *Methodus Differentialis sive Tractatus de Summatione et Interpolatione Serierum Infinitarum*. 1730.

[Sto+15] Lutz Stobbe, Marina Proske, Hannes Zedel, Ralph Hintemann, Jens Clausen, and Severin Beucker. *Entwicklung des IKT-bedingten Strombedarfs in Deutschland*. Studie im Auftrag des Bundesministeriums für Wirtschaft und Energie Projekt-Nr. 29/14. Fraunhofer IZM and Borderstep. 2015.

[Vig08] Sebastiano Vigna. "Broadword Implementation of Rank/Select Queries." In: *Proceedings of the 2008 International Workshop on Experimental and Efficient Algorithms*. WEA '08. 2008, pp. 154–168.

[War13] Henry S. Warren. *Hacker's Delight*. 2nd ed. Addison Wesley – Pearson Education, 2013.

[Web20] Pascal Weber. "Kantenminimierung in DeBruijn Graphen." Elaboration of Project Algorithm Engineering 2019 (draft by Uwe Baier). 2020.

[Wei73] Peter Weiner. “Linear pattern matching algorithms.” In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*. SWAT ’73. 1973, pp. 1–11.

[WST95] Frans M.J. Willems, Yuri M. Shtarkov, and Tjalling J. Tjalkens. “The context-tree weighting method: basic properties.” In: *IEEE Transactions on Information Theory* 41.3 (1995), pp. 653–664.

[Wil64] John William Joseph Williams. “Algorithm 232: Heapsort.” In: *Communications of the ACM* 7.6 (1964), pp. 347–349.

[ZL77] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression.” In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343.

[ZL78] Jacob Ziv and Abraham Lempel. “Compression of individual sequences via variable-rate coding.” In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536.

# A

# TEST DATA DESCRIPTION

The test data used in this thesis comes from 6 different publicly available text corpora, which are characterized next. More information can be found in Table A.1.

CANTERBURY CORPUS The Canterbury corpus is the historically oldest corpus set up in 1997. It contains 11 small-size files ranging between about 4 KB and 1 MB. The files are a mix of various text documents and binary sources.

LARGE CANTERBURY CORPUS The Large Canterbury Corpus is an extension of the Canterbury Corpus, consisting of 3 files with sizes between 2.3 and 4.4 MB. It can be seen as a modernization of the Canterbury Corpus, as the computational power of computer systems and therefore their ability to handle bigger data increased.

SILESIA CORPUS The Silesia Corpus again can be seen as a modernization of the both corpora from above. It consists of 12 files with about 5–49 MB, and also contains a few newer data sources such as medical images or database files.

PIZZA & CHILI CORPUS The Pizza & Chili Corpus is a relatively modern text corpus. It contains 6 files with e.g. audio data, DNA and protein sequences as well as source code and English text. The files are big (53–2108 MB) and not repetitive, so compressing these texts is difficult in terms of good compression rates.

REPETITIVE The Repetitive Corpus contains 9 big files with sizes between 45 and 440 MB. The texts are very repetitive, meaning that they contain a lot of long repetitions. This allows for good compression for all of these texts.

GENOMES Genomes contains full reference genomes of a human (`hg38`), a mouse (`mm10`) and a rat (`rn6`), collected from the UCSC Genome Browser. The files were converted from 2-bit-format to FASTA–format and all characters except of `A`,`C`,`C` and `T` were removed from the files. The file size range between 2500 and 2900 MB, and therefore are the biggest data sets used in our experiments.

| Text corpus | File | Size in MB | Alphabet size | Lines |
|---|---|---:|---:|---:|
| canterbury[1] | alice29.txt | 0.145 | 74 | 3,609 |
| | asyoulik.txt | 0.119 | 68 | 4,123 |
| | cp.html | 0.023 | 86 | 646 |
| | fields.c | 0.011 | 90 | 432 |
| | grammar.lsp | 0.004 | 76 | 95 |
| | kennedy.xls | 0.982 | 256 | 886 |
| | lcet10.txt | 0.407 | 84 | 7,520 |
| | plrabn12.txt | 0.460 | 81 | 10,700 |
| | ptt5 | 0.489 | 159 | 1 |
| | sum | 0.036 | 255 | 95 |
| | xargs.1 | 0.004 | 74 | 113 |
| largecanterbury[2] | bible.txt | 3.860 | 63 | 30,384 |
| | E.coli | 4.424 | 4 | 1 |
| | world192.txt | 2.359 | 94 | 65,120 |
| silesia[3] | dickens | 9.720 | 100 | 200,784 |
| | mozilla | 48.848 | 256 | 141,536 |
| | mr | 9.509 | 256 | 118,957 |
| | nci | 31.999 | 62 | 840,552 |
| | ooffice | 5.867 | 256 | 13,278 |
| | osdb | 9.618 | 256 | 29,564 |
| | reymont | 6.320 | 256 | 20,138 |
| | samba | 20.606 | 256 | 595,584 |
| | sao | 6.916 | 256 | 17,031 |
| | webster | 39.538 | 98 | 930,839 |
| | xml | 5.098 | 104 | 57,511 |
| | x-ray | 8.082 | 256 | 362,032 |
| pizzachili[4] | sources | 201.097 | 230 | 7,065,007 |
| | pitches | 53.246 | 133 | 84 |
| | proteins | 1,129.200 | 27 | 4,210,908 |
| | dna | 385.216 | 16 | 1,867 |
| | english | 2,108.000 | 239 | 43,894,766 |
| | dblp.xml | 282.417 | 97 | 7,619,950 |
| repetitive[5] | Escherichia-Coli | 107.468 | 15 | 1 |
| | cere | 439.917 | 5 | 1 |
| | coreutils | 195.772 | 236 | 6,443,719 |
| | einstein.de.txt | 88.461 | 117 | 545,357 |
| | einstein.en.txt | 445.963 | 139 | 2,379,632 |
| | influenza | 147.636 | 15 | 1 |
| | kernel | 246.011 | 160 | 9,550,967 |
| | para | 409.380 | 5 | 1 |
| | world-leaders | 44.792 | 89 | 103,041 |
| genomes[6] | hg38 | 2,908.070 | 4 | 1 |
| | mm10 | 2,529.890 | 4 | 1 |
| | rn6 | 2,603.400 | 4 | 1 |

[1] http://www.data-compression.info/Corpora/CanterburyCorpus/index.html
[2] http://www.data-compression.info/Corpora/CanterburyCorpus/index.html
[3] http://sun.aei.polsl.pl/ sdeor/index.php?page=silesia
[4] http://pizzachili.dcc.uchile.cl/texts.html
[5] http://pizzachili.dcc.uchile.cl/repcorpus.html
[6] http://hgdownload.soe.ucsc.edu/downloads.html

**Table A.1:** Statistics about the used test data.

# B

# DATA COMPRESSION BENCHMARK RESULTS

Experiments related to the data compression chapter were conducted with the aid of the `sdsl-lite`-library [Gog07]. The programs were written in `C++` and are publicly available [Bai20]. We used a computer equipped with two 16-core Intel Xeon E5-2698v3 processors and 256 GB of RAM because some compressors required more than 16 gigabytes of memory during compression. The computer used Ubuntu 18.04.3 LTS with 64 bit as operating system.

## B.1 COMPRESSOR BENCHMARK

We benchmarked various different lossless data compressors. A list of all compressors can be found in Table B.1. An overview of the used test data can be found in Chapter A.

Table B.2 shows the compression results. For small and medium-sized files, zpaq is the method of choice for good compression rates. For larger but not too repetitive files, bcm-t-greedy achieves the best results. The compressor xz serves best for very repetitive files.

The Tables B.3 and B.4 show the speed of compression and decompression. Typically, bw94 is the fastest compressor while xz is the fastest decompressor in almost all cases.

Finally, Tables B.5 and B.6 list the required memory peak during compression and decompression. For compression, the smallest memory peak is required by the bw94 and bcm compressors for small files and zpaq for bigger files. For decompression, xz outperforms all other compressors in terms of memory peak. Some outliers in Tables B.5 and B.6 exist because in some cases the standard memory required by a process exceeds the size of the file to be compressed/decompressed.

| Compressor | Description |
|---|---|
| bw94 | Block-sorting compression as described by Burrows and Wheeler in 1994 [BW94]. Uses the Burrows-Wheeler transform offered by the divsufsort library [Mor03], a self-implemented move-to-front transform and an entropy coder from Sachin Garg [Gar06]. The input is processed block-wise with at most 1.5 GB per block. |
| bw94-t-hirsch | bw94 enhanced with the hirsch tunneling strategy, see Section 4.3.1. |
| bw94-t-greedy | bw94 enhanced with the greedy tunneling strategy, see Section 4.3.2. |
| bw94-t-gupdate | bw94 enhanced with the greedy tunneling strategy that considers negative side effects, see Section 4.3.2. |
| bcm | One of the best open source BWT compressors to date using run-length encoding and context mixing, developed by Ilya Muravyov [Mur16]. The input is processed block-wise with at most 1.5 GB per block. |
| bcm-t-hirsch | bcm enhanced with the hirsch tunneling strategy, see Section 4.3.1. |
| bcm-t-greedy | bcm enhanced with the greedy tunneling strategy, see Section 4.3.2. |
| bcm-t-gupdate | bcm enhanced with the greedy tunneling strategy that considers negative side effects, see Section 4.3.2. |
| xz | One of the best LZ77-based compressors to date using the Lempel-Ziv-Markov Algorithm [Col10]. |
| zpaq | One of the best context-mixing compressors to date [Mah09]. |

**Table B.1:** Used compressors in the compression benchmark.

| File | bw94 | bw94-t-hirsch | bw94-t-greedy | bw94-t-gupdate | bcm | bcm-t-hirsch | bcm-t-greedy | bcm-t-gupdate | xz | zpaq |
|---|---|---|---|---|---|---|---|---|---|---|
| alice29.txt | 2.354 | 2.354 | 2.354 | 2.354 | 2.130 | 2.129 | 2.129 | 2.129 | 2.552 | **2.032** |
| asyoulik.txt | 2.635 | 2.631 | 2.630 | 2.630 | 2.393 | 2.388 | 2.387 | 2.387 | 2.849 | **2.289** |
| cp.html | 2.505 | 2.462 | 2.462 | 2.461 | 2.428 | 2.361 | 2.361 | 2.362 | 2.488 | **2.215** |
| fields.c | 2.201 | 2.194 | 2.191 | 2.191 | 2.288 | 2.268 | 2.265 | 2.265 | 2.175 | **1.883** |
| grammar.lsp | 2.775 | 2.797 | 2.797 | 2.797 | 2.833 | 2.850 | 2.850 | 2.850 | 2.777 | **2.461** |
| kennedy.xls | 1.496 | 1.496 | 1.496 | 1.496 | 0.594 | 0.594 | 0.594 | 0.594 | 0.402 | **0.260** |
| lcet10.txt | 2.114 | 2.111 | 2.111 | 2.111 | 1.866 | 1.861 | 1.860 | 1.860 | 2.239 | **1.767** |
| plrabn12.txt | 2.542 | 2.541 | 2.541 | 2.541 | 2.235 | 2.234 | 2.234 | 2.234 | 2.746 | **2.181** |
| ptt5 | 0.832 | 0.832 | 0.832 | 0.832 | 0.704 | 0.704 | 0.704 | 0.704 | **0.621** | 0.693 |
| sum | 2.779 | 2.657 | 2.651 | 2.654 | 2.557 | 2.389 | 2.377 | 2.384 | **1.987** | 2.186 |
| xargs.1 | 3.315 | 3.336 | 3.336 | 3.336 | 3.366 | 3.382 | 3.382 | 3.382 | 3.429 | **3.047** |
| bible.txt | 1.663 | 1.660 | 1.659 | 1.659 | 1.440 | 1.435 | 1.433 | 1.434 | 1.750 | **1.391** |
| E.coli | 2.010 | 1.990 | 1.990 | 1.990 | 1.939 | 1.915 | **1.914** | 1.915 | 2.045 | 1.952 |
| world192.txt | 1.438 | 1.433 | 1.432 | 1.432 | 1.290 | 1.282 | 1.280 | 1.280 | 1.568 | **1.269** |
| dickens | 2.113 | 2.108 | 2.108 | 2.108 | 1.761 | 1.752 | **1.752** | 1.752 | 2.222 | 1.775 |
| mozilla | 2.939 | 2.913 | 2.912 | 2.912 | 2.495 | 2.454 | 2.451 | 2.452 | **2.089** | 2.120 |
| mr | 2.083 | 2.083 | 2.083 | 2.083 | **1.699** | 1.699 | 1.699 | 1.699 | 2.208 | 1.795 |
| nci | 0.339 | 0.327 | 0.327 | 0.327 | 0.292 | 0.276 | **0.274** | 0.275 | 0.345 | 0.362 |
| ooffice | 3.821 | 3.796 | 3.795 | 3.795 | 3.306 | 3.271 | 3.268 | 3.269 | 3.156 | **2.931** |
| osdb | 2.250 | 2.238 | 2.238 | 2.238 | 1.784 | 1.771 | **1.771** | 1.771 | 2.256 | 1.886 |
| reymont | 1.470 | 1.469 | 1.469 | 1.469 | 1.186 | 1.184 | **1.184** | 1.184 | 1.588 | 1.271 |
| samba | 1.805 | 1.747 | 1.744 | 1.747 | 1.488 | 1.418 | 1.413 | 1.418 | 1.384 | **1.196** |
| sao | 5.949 | 5.949 | 5.949 | 5.949 | 5.155 | 5.155 | 5.155 | 5.155 | **4.882** | 4.980 |
| webster | 1.481 | 1.479 | 1.479 | 1.479 | 1.239 | 1.236 | 1.236 | 1.236 | 1.614 | **1.209** |
| xml | 0.664 | 0.640 | 0.639 | 0.641 | 0.590 | 0.558 | 0.556 | 0.559 | 0.650 | **0.530** |
| x-ray | 4.244 | 4.244 | 4.244 | 4.244 | **3.452** | 3.452 | 3.452 | 3.452 | 4.239 | 3.560 |
| sources | 1.383 | 1.351 | 1.348 | 1.349 | 1.222 | 1.175 | 1.169 | 1.171 | 1.184 | **0.989** |
| pitches | 2.838 | 2.649 | 2.640 | 2.647 | 2.664 | 2.424 | 2.409 | 2.419 | 1.980 | **1.963** |
| proteins | 2.289 | 1.972 | 1.965 | 1.977 | 2.331 | 1.912 | **1.901** | 1.920 | 2.222 | 2.609 |
| dna | 1.829 | 1.807 | 1.806 | 1.807 | 1.720 | 1.696 | **1.696** | 1.696 | 1.778 | 1.859 |
| english | 1.711 | 1.470 | 1.469 | 1.471 | 1.478 | 1.184 | **1.183** | 1.186 | 1.985 | 1.683 |
| dblp.xml | 0.751 | 0.745 | 0.744 | 0.744 | 0.628 | 0.619 | 0.618 | 0.618 | 0.819 | **0.605** |
| Escherichia-Coli | 0.776 | 0.526 | 0.516 | 0.525 | 0.796 | 0.519 | 0.506 | 0.517 | **0.368** | 1.941 |
| cere | 0.237 | 0.120 | 0.118 | 0.121 | 0.238 | 0.118 | 0.115 | 0.119 | **0.087** | 1.771 |
| coreutils | 0.232 | 0.148 | 0.147 | 0.156 | 0.229 | 0.133 | **0.132** | 0.144 | 0.144 | 0.618 |
| einstein.de.txt | 0.015 | 0.010 | 0.010 | 0.010 | 0.022 | 0.011 | 0.010 | 0.011 | 0.008 | **0.007** |
| einstein.en.txt | 0.007 | 0.005 | 0.005 | 0.005 | 0.015 | 0.007 | 0.007 | 0.007 | 0.005 | **0.004** |
| influenza | 0.120 | 0.116 | 0.115 | 0.115 | 0.119 | 0.113 | 0.112 | 0.112 | **0.082** | 0.351 |
| kernel | 0.130 | 0.069 | 0.068 | 0.074 | 0.136 | 0.062 | 0.061 | 0.069 | 0.064 | **0.058** |
| para | 0.308 | 0.173 | 0.168 | 0.177 | 0.315 | 0.171 | 0.165 | 0.176 | **0.113** | 1.854 |
| world-leaders | 0.121 | 0.100 | 0.098 | 0.100 | 0.126 | 0.097 | 0.094 | 0.097 | **0.088** | 0.093 |
| hg38 | 1.754 | 1.723 | 1.723 | 1.723 | 1.645 | 1.608 | **1.607** | 1.608 | 1.716 | 1.826 |
| mm10 | 1.714 | 1.699 | 1.699 | 1.699 | 1.618 | 1.598 | **1.598** | 1.598 | 1.669 | 1.840 |
| rn6 | 1.787 | 1.726 | 1.725 | 1.726 | 1.695 | 1.628 | **1.628** | 1.628 | 1.705 | 1.819 |

**Table B.2:** Compression results in bits per symbol. The best compression result of each file is printed in bold.

| File | bw94 | bw94-t-hirsch | bw94-t-greedy | bw94-t-gupdate | bcm | bcm-t-hirsch | bcm-t-greedy | bcm-t-gupdate | xz | zpaq |
|---|---|---|---|---|---|---|---|---|---|---|
| alice29.txt | **6.90** | 4.67 | 2.84 | 3.53 | 6.90 | 3.53 | 2.37 | 2.84 | 1.79 | 0.48 |
| asyoulik.txt | **10.85** | 3.85 | 2.91 | 3.85 | 5.68 | 2.91 | 2.34 | 2.91 | 1.47 | 0.39 |
| cp.html | **23.46** | 2.13 | 2.13 | 2.13 | 23.46 | 2.13 | 2.13 | 2.13 | 0.75 | 0.17 |
| fields.c | 10.63 | 0.96 | **10.63** | 10.63 | 10.63 | 10.63 | 0.96 | 0.96 | 0.96 | 0.11 |
| grammar.lsp | 3.54 | 3.54 | 3.54 | 3.54 | 3.54 | 3.54 | 3.54 | 3.54 | **3.54** | 0.06 |
| kennedy.xls | **9.72** | 7.49 | 6.96 | 8.11 | 7.49 | 8.11 | 6.09 | 5.42 | 0.81 | 0.74 |
| lcet10.txt | **7.98** | 4.47 | 3.10 | 4.02 | 7.98 | 3.10 | 2.38 | 3.10 | 1.92 | 0.63 |
| plrabn12.txt | **11.20** | 4.13 | 3.04 | 4.13 | 7.53 | 3.04 | 2.53 | 3.04 | 1.98 | 0.62 |
| ptt5 | **23.30** | 11.93 | 9.59 | 9.59 | 8.02 | 5.37 | 4.84 | 4.84 | 1.22 | 0.72 |
| sum | 3.31 | **3.31** | 3.31 | 3.31 | 3.31 | 1.73 | 1.73 | 3.31 | 0.88 | 0.27 |
| xargs.1 | 4.03 | **4.03** | 4.03 | 4.03 | 4.03 | 4.03 | 4.03 | 4.03 | 0.36 | 0.05 |
| bible.txt | **10.69** | 5.75 | 4.53 | 5.13 | 6.88 | 4.33 | 3.47 | 4.05 | 2.22 | 0.72 |
| E.coli | **9.39** | 5.26 | 3.00 | 4.75 | 6.49 | 3.74 | 2.51 | 3.56 | 1.31 | 0.81 |
| world192.txt | **11.17** | 6.91 | 5.11 | 5.73 | 7.34 | 5.23 | 3.99 | 4.36 | 2.31 | 0.74 |
| dickens | **8.99** | 5.25 | 3.66 | 4.90 | 5.39 | 4.24 | 3.06 | 3.65 | 1.62 | 0.64 |
| mozilla | **8.39** | 4.71 | 3.10 | 3.84 | 6.64 | 4.06 | 2.68 | 3.16 | 1.97 | 0.65 |
| mr | **9.99** | 7.30 | 4.97 | 7.25 | 6.59 | 5.36 | 4.07 | 4.24 | 1.49 | 0.66 |
| nci | **12.94** | 8.64 | 7.82 | 6.17 | 6.73 | 5.62 | 4.90 | 4.37 | 1.45 | 0.84 |
| ooffice | **8.13** | 4.84 | 3.27 | 4.27 | 6.04 | 4.12 | 2.96 | 3.68 | 2.32 | 0.57 |
| osdb | **9.90** | 7.28 | 4.16 | 7.33 | 6.53 | 6.12 | 4.57 | 6.16 | 2.27 | 0.67 |
| reymont | **10.34** | 6.50 | 5.17 | 6.44 | 6.93 | 4.82 | 3.89 | 4.64 | 1.71 | 0.68 |
| samba | **9.36** | 4.87 | 3.73 | 2.83 | 7.00 | 3.81 | 3.20 | 2.82 | 1.69 | 0.69 |
| sao | 5.85 | 4.54 | 2.67 | 4.45 | **6.00** | 3.63 | 2.16 | 4.37 | 2.55 | 0.60 |
| webster | **8.63** | 5.52 | 3.96 | 5.02 | 5.57 | 3.71 | 3.25 | 3.79 | 1.44 | 0.73 |
| xml | **15.88** | 8.07 | 6.87 | 6.13 | 7.71 | 5.78 | 5.36 | 4.63 | 2.68 | 0.78 |
| x-ray | **7.08** | 5.49 | 3.36 | 5.45 | 5.80 | 4.64 | 3.02 | 4.58 | 2.62 | 0.63 |
| sources | **7.70** | 2.91 | 2.42 | 1.78 | 5.28 | 2.43 | 2.16 | 1.61 | 1.62 | 0.73 |
| pitches | **8.17** | 2.65 | 2.03 | 1.64 | 6.19 | 2.41 | 1.83 | 1.52 | 2.06 | 0.68 |
| proteins | **4.35** | 1.02 | 0.93 | 0.62 | 3.47 | 0.98 | 0.90 | 0.58 | 0.93 | 0.66 |
| dna | **4.71** | 2.65 | 1.87 | 2.22 | 3.67 | 2.19 | 1.66 | 1.91 | 0.66 | 0.79 |
| english | **4.11** | 0.75 | 0.67 | 0.37 | 3.39 | 0.70 | 0.63 | 0.37 | 1.01 | 0.73 |
| dblp.xml | **7.50** | 5.28 | 4.39 | 4.24 | 4.91 | 3.98 | 3.43 | 3.37 | 1.58 | 0.74 |
| Escherichia-Coli | **6.79** | 1.55 | 1.47 | 0.14 | 4.75 | 1.48 | 1.43 | 0.14 | 0.96 | 0.81 |
| cere | **6.18** | 2.44 | 2.38 | 0.16 | 4.20 | 2.30 | 2.23 | 0.16 | 1.08 | 0.82 |
| coreutils | **8.42** | 2.47 | 2.36 | 0.62 | 5.40 | 2.38 | 2.37 | 0.59 | 2.84 | 0.75 |
| einstein.de.txt | **8.10** | 7.18 | 7.06 | 6.92 | 5.23 | 6.62 | 6.56 | 5.54 | 7.07 | 0.74 |
| einstein.en.txt | 5.98 | 5.16 | 5.32 | 5.06 | 4.08 | 4.77 | 4.60 | 4.28 | **7.47** | 0.76 |
| influenza | **7.92** | 4.89 | 4.80 | 3.41 | 5.02 | 3.70 | 3.56 | 2.88 | 2.16 | 0.84 |
| kernel | **8.45** | 3.04 | 2.97 | 1.45 | 5.26 | 2.98 | 3.12 | 1.46 | 3.28 | 0.74 |
| para | **6.08** | 2.64 | 2.52 | 0.42 | 4.21 | 2.57 | 2.52 | 0.42 | 1.05 | 0.83 |
| world-leaders | **19.38** | 7.13 | 6.89 | 3.85 | 7.74 | 5.73 | 4.81 | 3.27 | 2.71 | 0.84 |
| hg38 | **4.05** | 2.13 | 1.52 | 1.65 | 3.31 | 1.81 | 1.39 | 1.54 | 0.64 | 0.79 |
| mm10 | **3.98** | 2.35 | 1.65 | 1.85 | 3.21 | 1.96 | 1.46 | 1.62 | 0.65 | 0.80 |
| rn6 | **4.08** | 1.74 | 1.34 | 1.22 | 3.29 | 1.54 | 1.22 | 1.15 | 0.65 | 0.80 |

**Table B.3:** Compression speed in MB per second. The fastest compression of each file is printed in bold.

| File | bw94 | bw94-t-hirsch | bw94-t-greedy | bw94-t-gupdate | bcm | bcm-t-hirsch | bcm-t-greedy | bcm-t-gupdate | xz | zpaq |
|---|---|---|---|---|---|---|---|---|---|---|
| alice29.txt | 3.53 | 3.53 | 3.53 | 3.53 | 3.53 | 4.67 | 3.53 | 4.67 | **13.18** | 0.53 |
| asyoulik.txt | 3.85 | 3.85 | 3.85 | 3.85 | 3.85 | 5.68 | 3.85 | 3.85 | **119.37** | 0.41 |
| cp.html | 2.13 | 23.46 | 23.46 | 2.13 | 2.13 | 23.46 | 23.46 | 2.13 | **23.46** | 0.21 |
| fields.c | 10.63 | 10.63 | 10.63 | 10.63 | 10.63 | 0.96 | 10.63 | 10.63 | **10.63** | 0.11 |
| grammar.lsp | 3.54 | 3.54 | 3.54 | 3.54 | 3.54 | 3.54 | 3.54 | 3.54 | **3.54** | 0.06 |
| kennedy.xls | 4.44 | 4.65 | 4.44 | 4.25 | 6.09 | 5.74 | 5.74 | 6.09 | **46.76** | 0.72 |
| lcet10.txt | 5.02 | 4.02 | 4.02 | 4.02 | 5.02 | 6.67 | 4.47 | 6.67 | **36.99** | 0.59 |
| plrabn12.txt | 3.50 | 3.50 | 3.50 | 3.50 | 4.54 | 4.13 | 4.13 | 4.13 | **21.88** | 0.62 |
| ptt5 | 6.89 | 6.04 | 6.04 | 6.04 | 6.89 | 8.02 | 6.04 | 5.37 | **44.49** | 0.70 |
| sum | 1.73 | 3.31 | 1.73 | 1.73 | 3.31 | 3.31 | 3.31 | 3.31 | **36.46** | 0.20 |
| xargs.1 | 4.03 | 4.03 | 4.03 | 4.03 | 4.03 | 4.03 | 4.03 | 4.03 | **4.03** | 0.06 |
| bible.txt | 7.70 | 6.88 | 7.26 | 7.13 | 6.75 | 6.42 | 6.42 | 6.42 | **47.65** | 0.61 |
| E.coli | 10.03 | 9.59 | 9.59 | 9.80 | 7.12 | 5.26 | 5.89 | 5.97 | **48.61** | 0.81 |
| world192.txt | 6.72 | 6.19 | 6.72 | 7.83 | 6.53 | 6.35 | 6.19 | 6.53 | **46.25** | 0.69 |
| dickens | 4.90 | 4.45 | 3.93 | 4.47 | 5.58 | 5.08 | 5.51 | 5.08 | **53.70** | 0.70 |
| mozilla | 2.84 | 2.73 | 2.82 | 2.79 | 5.80 | 4.96 | 5.25 | 5.06 | **49.29** | 0.65 |
| mr | 3.80 | 3.18 | 3.23 | 3.40 | 7.09 | 5.33 | 5.33 | 5.22 | **36.43** | 0.76 |
| nci | 10.45 | 7.78 | 7.82 | 8.01 | 5.52 | 5.35 | 5.46 | 5.54 | **151.65** | 0.82 |
| ooffice | 2.27 | 2.33 | 2.35 | 2.35 | 5.69 | 5.52 | 6.04 | 5.42 | **25.39** | 0.62 |
| osdb | 3.57 | 3.95 | 3.69 | 3.89 | 6.00 | 7.62 | 6.72 | 7.45 | **41.63** | 0.63 |
| reymont | 3.45 | 3.45 | 3.43 | 3.19 | 6.57 | 6.50 | 6.44 | 6.57 | **52.23** | 0.75 |
| samba | 3.93 | 4.23 | 4.21 | 4.19 | 6.79 | 7.33 | 7.68 | 7.57 | **68.45** | 0.73 |
| sao | 1.99 | 2.03 | 2.02 | 1.98 | 4.93 | 5.04 | 4.11 | 4.66 | **20.89** | 0.58 |
| webster | 5.45 | 4.60 | 4.42 | 4.54 | 5.15 | 4.78 | 4.64 | 4.67 | **77.37** | 0.71 |
| xml | 11.30 | 11.82 | 11.55 | 11.55 | 8.34 | 9.60 | 9.78 | 9.08 | **83.56** | 0.75 |
| x-ray | 2.47 | 2.26 | 2.44 | 2.44 | 5.93 | 5.68 | 5.72 | 5.72 | **23.02** | 0.63 |
| sources | 3.63 | 3.44 | 3.45 | 3.25 | 5.88 | 5.38 | 5.59 | 5.47 | **96.17** | 0.71 |
| pitches | 3.61 | 4.04 | 4.18 | 3.78 | 6.06 | 5.70 | 5.70 | 5.93 | **51.64** | 0.70 |
| proteins | 6.00 | 4.92 | 4.95 | 4.73 | 4.45 | 4.15 | 4.19 | 4.07 | **54.49** | 0.65 |
| dna | 5.45 | 4.18 | 4.19 | 4.11 | 4.42 | 3.38 | 3.28 | 3.29 | **78.27** | 0.79 |
| english | 2.35 | 2.39 | 2.30 | 2.29 | 4.34 | 3.61 | 3.63 | 3.46 | **75.93** | 0.70 |
| dblp.xml | 8.09 | 6.15 | 6.62 | 6.40 | 6.15 | 5.43 | 5.52 | 5.40 | **109.84** | 0.71 |
| Escherichia-Coli | 6.93 | 6.40 | 6.80 | 6.48 | 4.93 | 6.00 | 5.72 | 5.84 | **206.27** | 0.77 |
| cere | 7.42 | 6.91 | 6.98 | 6.99 | 4.93 | 6.26 | 6.55 | 6.03 | **382.20** | 0.80 |
| coreutils | 6.34 | 12.30 | 13.48 | 9.44 | 5.52 | 13.85 | 14.89 | 9.75 | **305.41** | 0.70 |
| einstein.de.txt | 15.65 | 18.12 | 19.78 | 18.08 | 6.84 | 14.84 | 16.34 | 13.42 | **352.43** | 0.76 |
| einstein.en.txt | 14.20 | 12.29 | 12.69 | 12.62 | 6.53 | 9.36 | 9.48 | 8.70 | **484.21** | 0.72 |
| influenza | 11.88 | 7.67 | 7.76 | 7.67 | 6.01 | 4.82 | 4.71 | 4.88 | **278.03** | 0.80 |
| kernel | 6.70 | 18.83 | 16.77 | 12.99 | 4.88 | 20.03 | 19.53 | 13.30 | **383.79** | 0.75 |
| para | 7.46 | 6.61 | 6.89 | 6.37 | 4.76 | 6.08 | 6.16 | 5.99 | **343.72** | 0.80 |
| world-leaders | 9.90 | 8.90 | 10.06 | 8.49 | 5.74 | 6.99 | 7.61 | 6.22 | **247.47** | 0.81 |
| hg38 | 5.85 | 3.85 | 3.65 | 3.72 | 4.41 | 2.76 | 2.84 | 2.97 | **78.99** | 0.76 |
| mm10 | 5.89 | 3.97 | 3.80 | 3.95 | 4.30 | 3.06 | 3.04 | 3.03 | **86.31** | 0.81 |
| rn6 | 5.96 | 3.89 | 3.59 | 3.82 | 4.43 | 3.04 | 3.08 | 3.12 | **82.75** | 0.79 |

**Table B.4:** Decompression speed in MB per second. The fastest decompression of each file is printed in bold.

| File | bw94 | bw94-t-hirsch | bw94-t-greedy | bw94-t-gupdate | bcm | bcm-t-hirsch | bcm-t-greedy | bcm-t-gupdate | xz | zpaq |
|---|---|---|---|---|---|---|---|---|---|---|
| alice29.txt | **236.35** | 268.66 | 278.36 | 302.92 | 239.79 | 279.65 | 287.19 | 301.20 | 1624.94 | 5040.30 |
| asyoulik.txt | 278.78 | 330.61 | 347.62 | 348.93 | **270.93** | 338.99 | 349.98 | 365.95 | 1903.58 | 6064.66 |
| cp.html | **1303.90** | 1342.52 | 1385.14 | 1390.47 | 1329.20 | 1415.77 | 1426.43 | 1442.41 | 7202.75 | 29654.08 |
| fields.c | **2850.66** | 2953.52 | 2944.71 | 2968.22 | 2991.73 | 3035.81 | 3050.50 | 3068.14 | 8070.03 | 47041.91 |
| grammar.lsp | **8260.24** | 8806.23 | 9035.19 | 8682.94 | 8709.36 | 8999.97 | 9184.90 | 8991.16 | 15622.26 | 89400.89 |
| kennedy.xls | 70.86 | 71.34 | **69.72** | 87.95 | 70.54 | 70.86 | 71.78 | 90.97 | 277.10 | 765.40 |
| lcet10.txt | **109.49** | 129.99 | 137.59 | 155.87 | 110.10 | 129.53 | 135.37 | 154.79 | 823.66 | 1833.14 |
| plrabn12.txt | **100.64** | 136.14 | 144.84 | 160.35 | 102.20 | 139.67 | 147.09 | 167.49 | 633.10 | 1591.95 |
| ptt5 | 101.90 | 99.73 | 101.77 | 101.19 | **98.96** | 100.56 | 100.30 | 102.34 | 731.19 | 1533.06 |
| sum | 874.04 | 905.74 | 905.74 | 903.17 | **840.62** | 904.03 | 931.45 | 944.30 | 5760.96 | 20319.75 |
| xargs.1 | **7217.17** | 7504.00 | 8000.13 | 7767.57 | 7612.53 | 7945.87 | 7961.37 | 8031.14 | 17302.62 | 99784.64 |
| bible.txt | **47.30** | 58.17 | 64.15 | 85.95 | 47.61 | 58.43 | 64.16 | 86.61 | 144.15 | 196.78 |
| E.coli | **46.45** | 106.91 | 120.38 | 144.65 | 46.74 | 107.27 | 120.54 | 144.70 | 77.45 | 63.69 |
| world192.txt | **51.94** | 53.04 | 57.04 | 78.00 | 52.51 | 52.67 | 57.05 | 78.27 | 265.94 | 324.32 |
| dickens | 43.12 | 67.06 | 75.79 | 100.21 | **43.07** | 67.37 | 75.78 | 100.24 | 116.62 | 84.59 |
| mozilla | 40.58 | 58.11 | 66.82 | 93.02 | 40.62 | 58.12 | 66.82 | 93.03 | 82.96 | **17.96** |
| mr | 43.22 | 55.70 | 62.97 | 86.25 | **42.89** | 55.71 | 62.88 | 86.28 | 122.31 | 86.36 |
| nci | 40.90 | 40.96 | 40.92 | 41.03 | 40.91 | 40.96 | 40.93 | 41.04 | 77.09 | **26.59** |
| ooffice | **44.78** | 80.75 | 91.47 | 114.89 | 44.88 | 81.15 | 91.35 | 115.32 | 163.38 | 135.74 |
| osdb | **42.82** | 52.23 | 58.95 | 82.24 | 42.89 | 52.29 | 59.10 | 82.54 | 127.51 | 85.92 |
| reymont | **44.61** | 50.73 | 56.11 | 78.16 | 44.90 | 50.91 | 56.45 | 78.51 | 153.72 | 126.42 |
| samba | **41.40** | 41.41 | 45.82 | 69.94 | 41.42 | 41.42 | 45.96 | 69.94 | 97.99 | 42.59 |
| sao | **43.96** | 110.04 | 127.09 | 152.01 | 44.09 | 110.36 | 127.25 | 151.77 | 149.20 | 116.18 |
| webster | 40.73 | 47.17 | 53.29 | 78.53 | 40.73 | 47.18 | 53.28 | 78.51 | 84.78 | **22.16** |
| xml | 45.56 | 45.78 | 45.48 | 50.12 | 45.62 | **45.45** | 45.57 | 50.46 | 170.19 | 154.93 |
| x-ray | **43.53** | 84.42 | 97.07 | 121.80 | 43.60 | 84.61 | 97.15 | 121.69 | 138.23 | 100.80 |
| sources | 40.14 | 40.14 | 43.12 | 70.20 | 40.14 | 40.14 | 43.12 | 70.20 | 26.87 | **4.36** |
| pitches | 40.56 | 63.78 | 72.74 | 99.23 | 40.54 | 63.80 | 72.75 | 99.23 | 82.06 | **16.44** |
| proteins | 40.02 | 57.16 | 66.61 | 97.25 | 40.02 | 57.16 | 66.61 | 97.25 | 4.46 | **0.74** |
| dna | 40.07 | 87.20 | 102.11 | 131.99 | 40.07 | 87.20 | 102.11 | 131.99 | 12.74 | **1.70** |
| english | 37.96 | 45.08 | 51.82 | 81.17 | 37.96 | 45.08 | 51.82 | 81.17 | 2.56 | **0.41** |
| dblp.xml | 40.10 | 40.10 | 40.10 | 56.19 | 40.10 | 40.10 | 40.10 | 56.18 | 19.13 | **3.10** |
| Escherichia-Coli | 40.26 | 40.26 | 40.26 | 53.38 | 40.27 | 40.26 | 40.26 | 53.36 | 45.68 | **6.17** |
| cere | 40.06 | 40.06 | 40.06 | 40.06 | 40.06 | 40.06 | 40.06 | 40.06 | 11.12 | **1.05** |
| coreutils | 40.14 | 40.14 | 40.14 | 40.14 | 40.14 | 40.14 | 40.14 | 40.14 | 27.60 | **4.47** |
| einstein.de.txt | 40.32 | 40.33 | 40.32 | 40.32 | 40.32 | 40.32 | 40.32 | 40.33 | 58.52 | **9.91** |
| einstein.en.txt | 40.06 | 40.06 | 40.06 | 40.06 | 40.06 | 40.06 | 40.06 | 40.06 | 11.94 | **1.96** |
| influenza | 40.19 | 40.18 | 40.19 | 40.19 | 40.18 | 40.18 | 40.18 | 40.19 | 33.21 | **4.39** |
| kernel | 40.11 | 40.11 | 40.11 | 40.11 | 40.11 | 40.11 | 40.11 | 40.11 | 21.95 | **3.56** |
| para | 40.07 | 40.07 | 40.07 | 40.07 | 40.07 | 40.07 | 40.07 | 40.07 | 11.95 | **1.18** |
| world-leaders | 40.66 | 40.69 | 40.66 | 40.69 | 40.67 | 40.65 | 40.65 | 40.67 | 82.26 | **19.46** |
| hg38 | 27.51 | 57.87 | 68.95 | 91.11 | 27.51 | 57.87 | 68.95 | 91.11 | 1.68 | **0.12** |
| mm10 | 31.63 | 67.06 | 79.95 | 105.43 | 31.63 | 67.06 | 79.95 | 105.43 | 1.93 | **0.14** |
| rn6 | 30.74 | 65.42 | 77.88 | 102.65 | 30.74 | 65.42 | 77.88 | 102.65 | 1.87 | **0.14** |

**Table B.5:** Compression memory peak in bits per symbol. The compression with the lowest memory peak is printed in bold.

| File | bw94 | bw94-t-hirsch | bw94-t-greedy | bw94-t-gupdate | bcm | bcm-t-hirsch | bcm-t-greedy | bcm-t-gupdate | xz | zpaq |
|---|---|---|---|---|---|---|---|---|---|---|
| alice29.txt | 223.20 | 217.17 | 217.17 | 219.11 | 222.99 | 241.52 | 239.58 | 233.55 | **173.43** | 5019.18 |
| asyoulik.txt | 257.31 | 267.00 | 271.19 | 273.28 | 280.35 | 270.66 | 276.42 | 272.76 | **207.84** | 6055.24 |
| cp.html | 1309.22 | 1313.22 | 1311.89 | 1267.94 | 1351.84 | 1322.54 | 1341.19 | 1375.82 | **1060.16** | 29540.87 |
| fields.c | 2841.85 | 2830.09 | 2809.52 | 2824.21 | 2962.34 | 2938.83 | 2882.99 | 2906.50 | **2392.21** | 46836.19 |
| grammar.lsp | 8383.53 | 8154.57 | 8330.69 | 8436.37 | 8594.88 | 8762.20 | 8506.82 | 8630.11 | **7221.11** | 88388.17 |
| kennedy.xls | 66.85 | 69.24 | 68.60 | 68.28 | 69.27 | 70.16 | 71.31 | 70.23 | **27.52** | 764.76 |
| lcet10.txt | 105.42 | 104.42 | 106.19 | 106.42 | 108.57 | 109.26 | 110.79 | 109.49 | **64.03** | 1823.93 |
| plrabn12.txt | 96.97 | 101.52 | 101.39 | 102.20 | 100.57 | 102.48 | 100.91 | 99.96 | **56.71** | 1588.68 |
| ptt5 | 95.32 | 96.15 | 94.87 | 96.79 | 95.64 | 99.09 | 102.09 | 100.36 | **53.18** | 1529.99 |
| sum | 810.63 | 884.32 | 835.48 | 841.47 | 876.61 | 868.04 | 860.33 | 868.90 | **712.94** | 20288.05 |
| xargs.1 | 7170.66 | 7286.94 | 7519.50 | 7379.97 | 7651.29 | 7612.53 | 7798.58 | 7550.51 | **6503.98** | 99521.07 |
| bible.txt | 46.81 | 48.34 | 48.32 | 48.33 | 47.45 | 49.04 | 49.02 | 48.95 | **12.88** | 196.11 |
| E.coli | 45.93 | 47.16 | 47.26 | 47.18 | 45.82 | 47.45 | 47.70 | 47.45 | **12.17** | 63.26 |
| world192.txt | 51.09 | 51.31 | 50.19 | 50.36 | 51.46 | 51.07 | 51.37 | 50.99 | **16.03** | 323.57 |
| dickens | 42.94 | 44.08 | 44.07 | 44.00 | 42.85 | 44.18 | 44.19 | 44.20 | **10.00** | 84.46 |
| mozilla | 40.55 | 39.97 | 39.62 | 39.95 | 40.55 | 39.98 | 39.66 | 40.00 | **8.39** | 17.91 |
| mr | 42.86 | 44.80 | 44.82 | 44.86 | 42.85 | 44.94 | 44.95 | 44.89 | **10.05** | 86.23 |
| nci | 40.85 | 37.25 | 36.55 | 36.98 | 40.83 | 37.28 | 36.61 | 37.01 | **8.59** | 26.55 |
| ooffice | 44.45 | 44.81 | 44.90 | 44.86 | 44.44 | 45.08 | 44.95 | 45.03 | **11.12** | 135.25 |
| osdb | 42.80 | 31.83 | 31.89 | 31.79 | 42.80 | 31.96 | 31.79 | 32.06 | **9.92** | 85.63 |
| reymont | 44.37 | 45.36 | 45.15 | 45.00 | 44.26 | 45.47 | 45.27 | 45.20 | **11.00** | 126.19 |
| samba | 41.29 | 34.69 | 34.04 | 34.94 | 41.28 | 34.77 | 34.13 | 34.99 | **8.90** | 42.46 |
| sao | 43.83 | 45.90 | 45.83 | 45.74 | 44.02 | 46.12 | 46.05 | 46.13 | **10.64** | 115.97 |
| webster | 40.66 | 42.06 | 42.05 | 42.06 | 40.67 | 42.07 | 42.11 | 42.06 | **8.48** | 22.11 |
| xml | 45.07 | 34.10 | 33.12 | 35.36 | 45.67 | 34.66 | 33.32 | 35.40 | **11.58** | 154.41 |
| x-ray | 43.28 | 45.38 | 45.33 | 45.23 | 43.37 | 45.46 | 45.62 | 45.51 | **10.34** | 100.44 |
| sources | 40.12 | 36.50 | 35.65 | 36.08 | 40.13 | 36.51 | 35.65 | 36.06 | **2.63** | 4.35 |
| pitches | 40.49 | 32.48 | 31.78 | 32.35 | 40.50 | 32.52 | 31.82 | 32.35 | **8.35** | 16.42 |
| proteins | 40.02 | 27.66 | 27.23 | 27.90 | 40.02 | 27.66 | 27.23 | 27.90 | **0.47** | 0.74 |
| dna | 40.06 | 41.39 | 41.38 | 41.38 | 40.07 | 41.40 | 41.38 | 41.38 | **1.37** | 1.70 |
| english | 37.96 | 27.87 | 27.82 | 27.94 | 37.96 | 27.87 | 27.82 | 27.94 | **0.25** | 0.41 |
| dblp.xml | 40.09 | 33.48 | 32.96 | 33.86 | 40.09 | 33.49 | 32.96 | 33.87 | **1.87** | 3.09 |
| Escherichia-Coli | 40.24 | 13.36 | 12.52 | 13.23 | 40.26 | 13.36 | 12.56 | 13.23 | **4.94** | 6.15 |
| cere | 40.06 | 8.24 | 7.85 | 8.43 | 40.06 | 8.25 | 7.86 | 8.43 | 1.20 | **1.04** |
| coreutils | 40.13 | 6.18 | 5.82 | 10.93 | 40.13 | 6.18 | 5.83 | 10.95 | **2.70** | 4.47 |
| einstein.de.txt | 40.29 | 8.01 | 6.94 | 10.41 | 40.31 | 8.02 | 6.97 | 10.41 | **6.00** | 9.88 |
| einstein.en.txt | 40.05 | 12.62 | 11.64 | 14.53 | 40.06 | 12.62 | 11.64 | 14.54 | **1.19** | 1.96 |
| influenza | 40.17 | 38.38 | 37.55 | 37.92 | 40.18 | 38.39 | 37.56 | 37.92 | **3.59** | 4.38 |
| kernel | 40.10 | 2.21 | **2.12** | 3.75 | 40.11 | 2.21 | 2.12 | 3.75 | 2.15 | 3.55 |
| para | 40.06 | 8.14 | 7.53 | 8.47 | 40.06 | 8.14 | 7.54 | 8.48 | 1.29 | **1.17** |
| world-leaders | 40.58 | 21.84 | 19.97 | 23.32 | 40.59 | 21.87 | 19.99 | 23.32 | **8.43** | 19.43 |
| hg38 | 27.51 | 28.38 | 28.36 | 28.37 | 27.51 | 28.38 | 28.36 | 28.37 | 0.18 | **0.12** |
| mm10 | 31.63 | 32.70 | 32.66 | 32.70 | 31.63 | 32.70 | 32.66 | 32.70 | 0.20 | **0.14** |
| rn6 | 30.73 | 30.76 | 30.74 | 30.76 | 30.73 | 30.76 | 30.73 | 30.76 | 0.20 | **0.14** |

**Table B.6:** Decompression memory peak in bits per symbol. The decompression with the lowest memory peak is printed in bold.

## B.2 TUNNELING IMPACT

To measure the impact of tunneling, we added some additional experiments. In the first experiment, we compared the compression rates of normal and tunnel-enhanced BWT compressors. The results can be found in Table B.7. The table shows that the greedy and hirsch tunnel strategies reduce the encoding size by the biggest amount. Both tunneling strategies reduce the encoding sizes in almost all cases.

Table B.7 shows that the greedy update strategy does not work that good, so it seems like positive and negative side effects between prefix intervals balance each other out. Moreover, we included another tunnel strategy using edge reduction in de Bruijn graphs as shown in Section 5.2. The strategy uses the same BWT post stages, but increases the encoding size in many cases. This shows that the edge minimization approach is not suitable for data compression. However, the approach has a big impact in the field of sequence analysis, which is shown in the next chapter.

The Figures B.1 and B.2 show the potential of tunneling and the optimality of the greedy and hirsch tunnel strategies using the post stages of the bw94 and bcm compressors. Albeit hirsch and greedy do choose less than the optimal amount of prefix intervals to be tunneled, the choice is good enough to reduce the encoding size by almost the optimum. The "conservative behavior" of the strategies is owed to the conservative cost model used to determine the best prefix intervals, see Section 4.2.2. Another cost model might improve the benefits of both strategies. However, the used cost model results in a benefit which is very close to the optimum, so the cost model is "good enough" to work well in practice.

Differences between Table B.7 and Figures B.1 and B.2 can be explained by the different methods used. Table B.7 uses the full test files and compares the full encodings. The Figures B.1 and B.2 instead use at most 1 GB prefixes of each test file to reduce the computational amount of the benchmark. Moreover, Figures B.1 and B.2 measure only the size of the encoded components L and aux and ignore additional fields such as original test file length or the BWT index.

In summary, the greedy tunnel strategy achieves the biggest benefits. The most resource-saving strategy is the hirsch strategy. Because the differences are subtle, our BWT compressor of choice is bcm with the hirsch tunneling strategy.

| File | bw94-t-hirsch | bw94-t-greedy | bw94-t-gupdate | bw94-t-debruijn | bcm-t-hirsch | bcm-t-greedy | bcm-t-gupdate | bcm-t-debruijn |
|---|---|---|---|---|---|---|---|---|
| alice29.txt | 0.00% | 0.00% | 0.00% | −18.99% | 0.05% | 0.05% | 0.05% | −17.75% |
| asyoulik.txt | 0.15% | 0.19% | 0.19% | −14.42% | 0.21% | 0.25% | 0.25% | −13.62% |
| cp.html | 1.72% | 1.72% | 1.76% | −10.10% | 2.76% | 2.76% | 2.72% | −7.37% |
| fields.c | 0.32% | 0.46% | 0.46% | −22.13% | 0.87% | 1.01% | 1.01% | −16.57% |
| grammar.lsp | −0.79% | −0.79% | −0.79% | −15.32% | −0.60% | −0.60% | −0.60% | −9.74% |
| kennedy.xls | 0.00% | 0.00% | 0.00% | −78.74% | 0.00% | 0.00% | 0.00% | −92.76% |
| lcet10.txt | 0.14% | 0.14% | 0.14% | −20.72% | 0.27% | 0.32% | 0.32% | −20.31% |
| plrabn12.txt | 0.04% | 0.04% | 0.04% | −15.97% | 0.04% | 0.04% | 0.04% | −15.93% |
| ptt5 | 0.00% | 0.00% | 0.00% | −478.37% | 0.00% | 0.00% | 0.00% | −532.81% |
| sum | 4.39% | 4.61% | 4.50% | −32.74% | 6.57% | 7.04% | 6.77% | −33.71% |
| xargs.1 | −0.63% | −0.63% | −0.63% | −13.73% | −0.48% | −0.48% | −0.48% | −9.42% |
| bible.txt | 0.18% | 0.24% | 0.24% | −26.70% | 0.35% | 0.49% | 0.42% | −27.22% |
| E.coli | 1.00% | 1.00% | 1.00% | −17.01% | 1.24% | 1.29% | 1.24% | −13.25% |
| world192.txt | 0.35% | 0.42% | 0.42% | −18.57% | 0.62% | 0.78% | 0.78% | −18.22% |
| dickens | 0.24% | 0.24% | 0.24% | −19.02% | 0.51% | 0.51% | 0.51% | −19.48% |
| mozilla | 0.89% | 0.92% | 0.92% | −33.55% | 1.64% | 1.76% | 1.72% | −35.99% |
| mr | 0.00% | 0.00% | 0.00% | −129.24% | 0.00% | 0.00% | 0.00% | −133.25% |
| nci | 3.54% | 3.54% | 3.54% | −22.42% | 5.48% | 6.17% | 5.82% | −22.60% |
| ooffice | 0.65% | 0.68% | 0.68% | −14.81% | 1.06% | 1.15% | 1.12% | −16.36% |
| osdb | 0.53% | 0.53% | 0.53% | −4.89% | 0.73% | 0.73% | 0.73% | −6.67% |
| reymont | 0.07% | 0.07% | 0.07% | −14.56% | 0.17% | 0.17% | 0.17% | −16.10% |
| samba | 3.21% | 3.38% | 3.21% | −16.62% | 4.71% | 5.04% | 4.71% | −16.73% |
| sao | 0.00% | 0.00% | 0.00% | −0.99% | 0.00% | 0.00% | 0.00% | −0.99% |
| webster | 0.14% | 0.14% | 0.14% | −18.64% | 0.24% | 0.24% | 0.24% | −18.89% |
| xml | 3.61% | 3.76% | 3.46% | −14.16% | 5.42% | 5.76% | 5.26% | −12.54% |
| x-ray | 0.00% | 0.00% | 0.00% | −10.23% | 0.00% | 0.00% | 0.00% | −10.26% |
| sources | 2.31% | 2.53% | 2.46% | −18.73% | 3.85% | 4.34% | 4.17% | −17.02% |
| pitches | 6.66% | 6.98% | 6.73% | 1.69% | 9.01% | 9.57% | 9.20% | 3.98% |
| proteins | 13.85% | 14.15% | 13.63% | 8.69% | 17.98% | 18.45% | 17.63% | 12.53% |
| dna | 1.20% | 1.26% | 1.20% | −14.93% | 1.39% | 1.39% | 1.39% | −12.21% |
| english | 14.09% | 14.14% | 14.03% | 10.75% | 19.89% | 19.96% | 19.76% | 16.51% |
| dblp.xml | 0.80% | 0.93% | 0.93% | −14.25% | 1.43% | 1.59% | 1.59% | −14.33% |
| Escherichia-Coli | 32.22% | 33.51% | 32.35% | 23.58% | 34.80% | 36.43% | 35.05% | 27.64% |
| cere | 49.37% | 50.21% | 48.95% | 43.88% | 50.42% | 51.68% | 50.00% | 45.80% |
| coreutils | 36.21% | 36.64% | 32.76% | 18.54% | 41.92% | 42.36% | 37.12% | 24.02% |
| einstein.de.txt | 33.33% | 33.33% | 33.33% | 20.00% | 50.00% | 54.55% | 50.00% | 45.46% |
| einstein.en.txt | 28.57% | 28.57% | 28.57% | 14.29% | 53.33% | 53.33% | 53.33% | 60.00% |
| influenza | 3.33% | 4.17% | 4.17% | −46.67% | 5.04% | 5.88% | 5.88% | −42.86% |
| kernel | 46.92% | 47.69% | 43.08% | 40.77% | 54.41% | 55.15% | 49.26% | 47.79% |
| para | 43.83% | 45.45% | 42.53% | 29.55% | 45.71% | 47.62% | 44.13% | 32.06% |
| world-leaders | 17.36% | 19.01% | 17.36% | −4.13% | 23.02% | 25.40% | 23.02% | 3.18% |
| hg38 | 1.77% | 1.77% | 1.77% | −0.23% | 2.25% | 2.31% | 2.25% | 0.67% |
| mm10 | 0.88% | 0.88% | 0.88% | −1.52% | 1.24% | 1.24% | 1.24% | −0.87% |
| rn6 | 3.41% | 3.47% | 3.41% | 1.01% | 3.95% | 3.95% | 3.95% | 1.89% |

**Table B.7:** Tunneling compression improvements in percentage. The numbers show the encoding size differences between the normal and the enhanced BWT compressor relative to the encoding size of the normal BWT compressor. A positive number indicates an improvement, a negative number indicates a deterioration.

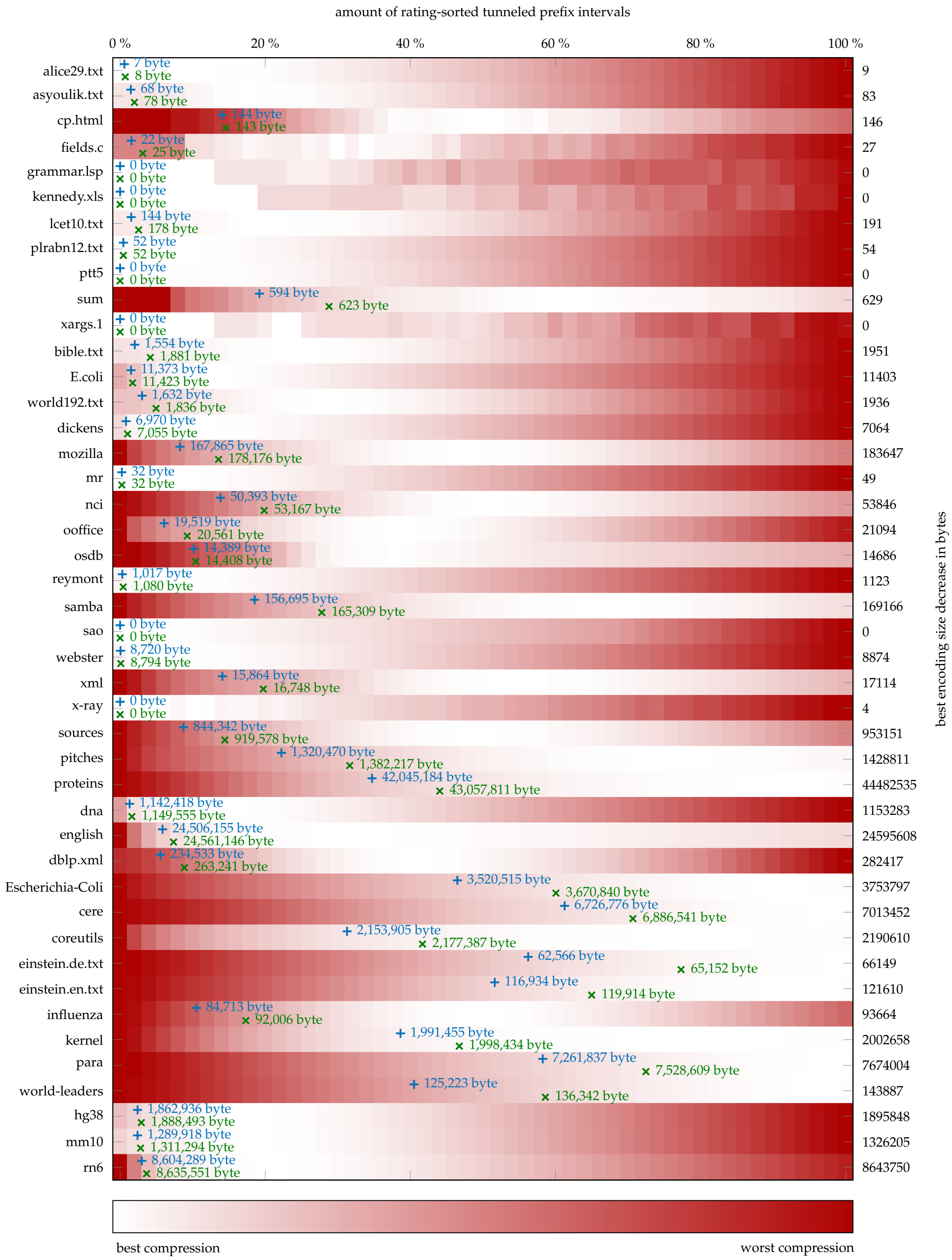


**Figure B.1:** Optimality of the hirsch and greedy strategy with the bw94 post stages. The amount of tunneled prefix intervals using the hirsch strategy is indicated with blue pluses. The amount of tunneled prefix intervals using the greedy strategy is indicated with green crosses. The best encoding size decrease compared to an compression without tunneling is shown on the right. The encoding size decrease of the hirsch and greedy strategy is shown right beside the pluses/crosses. The benchmark uses up to at most 1 GB of the prefix of a test file to reduce the computational amount.

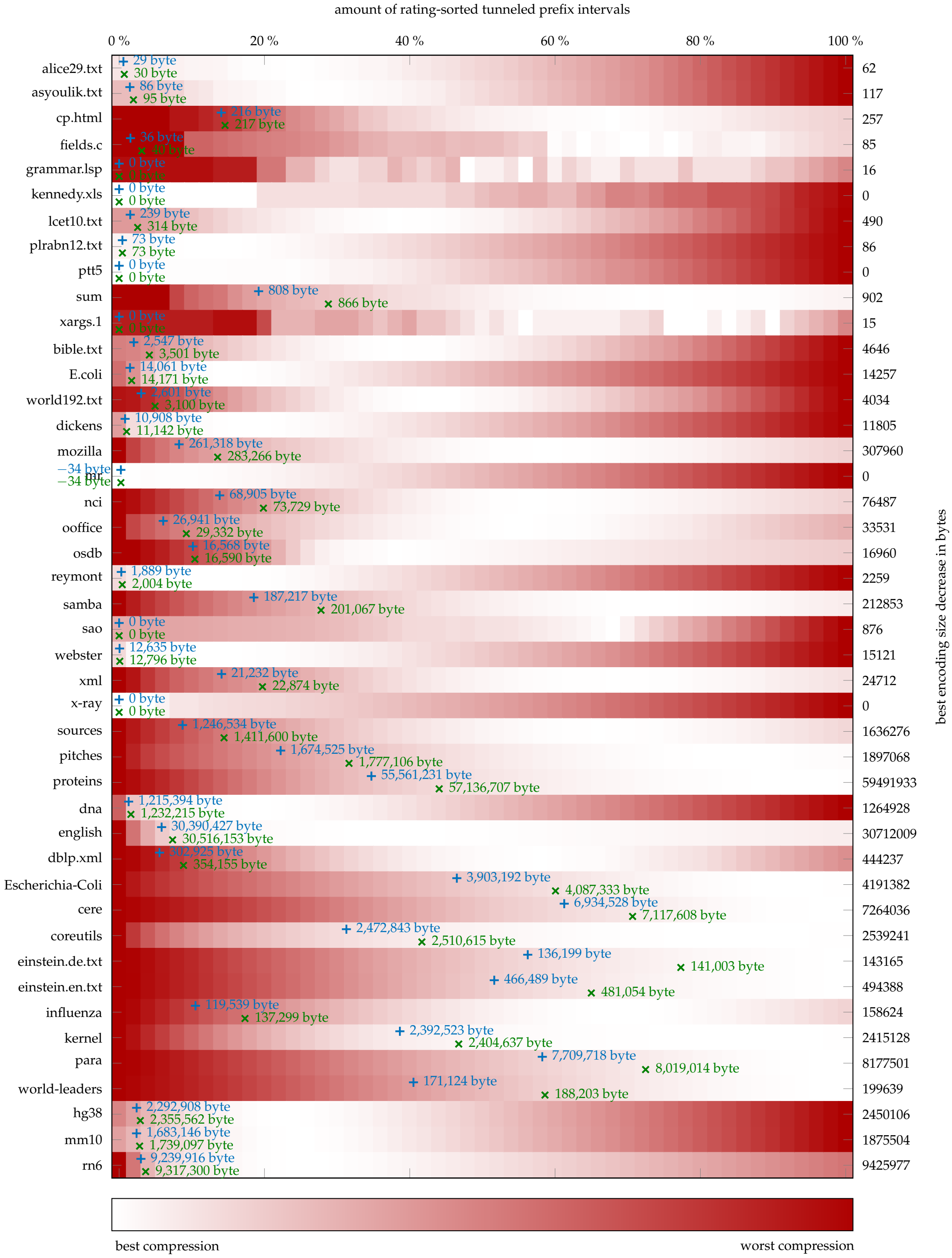


**Figure B.2:** Optimality of the hirsch and greedy strategy with the bcm post stages. The amount of tunneled prefix intervals using the hirsch strategy is indicated with blue pluses. The amount of tunneled prefix intervals using the greedy strategy is indicated with green crosses. The best encoding size decrease compared to an compression without tunneling is shown on the right. The encoding size decrease of the hirsch and greedy strategy is shown right beside the pluses/crosses. The benchmark uses up to at most 1 GB of the prefix of a test file to reduce the computational amount.

# C SEQUENCE ANALYSIS BENCHMARK RESULTS

Experiments related to the sequence analysis chapter 5 were conducted with the support of the `sdsl-lite`-library [Gog07]. The programs were written in `C++` and are publicly available [Bai20]. Because the experiments are very space- and time-consuming, we used a high performance computer equipped with two 16-core Intel Xeon E5-2698v3 processors and 256 GB of RAM. The computer used Ubuntu 18.04.3 LTS with 64 bit as operating system. We removed all nullbytes from the test data because of technical reasons in combination with the `sdsl-lite`-library.

## C.1 DE BRUIJN GRAPH EDGE REDUCTION

We implemented the algorithms presented in Sections 5.1 and 5.2. As a result, we obtain a construction algorithm for a tunneled FM-index by means of de Bruijn graph edge minimization.

Figure C.1 shows the dependence between the graph order k and the number of reduced edges. For most of the test data, the best order $k^*$ lies between 5 and 41, only very repetitive files have a higher best order.

The number of reduced edges using the best order $k^*$ can be found in Figure C.2. The figure furthermore shows the size of the tunneled FM-index compared to the normal one. The best results were achieved on repetitive data, the worst on very small files or non-repetitive DNA sequences.

The edge minimization algorithm on Page 5.5 typically requires the same time as FM-index construction, as can be seen on Figure C.3. The algorithm runs faster on repetitive files which is not surprising in a sense as the computationally intense steps have an output-sensitive worst-case time, see Corollary 5.9. The memory peaks during construction typically are $5n$ bytes for files with less than 2 GB and $9n$ bytes for files with more than 2 GB, see Figure C.4. This corresponds to the memory peak of suffix array construction using `divsufsort` [Mor03]. For cases in which the program requires more than $9n$ bytes the input size is so small that the memory overhead of the running process matters.

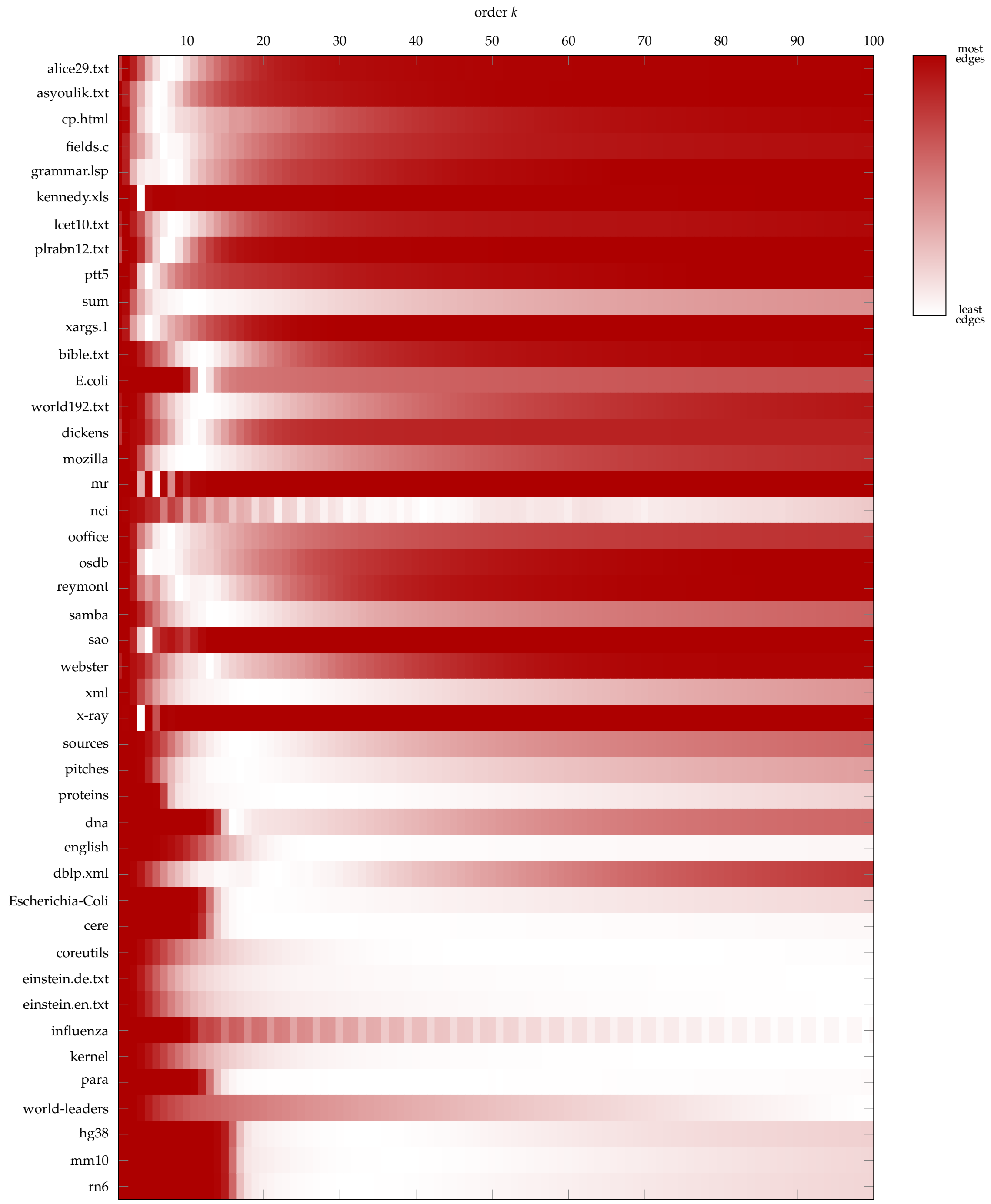


**Figure C.1:** Dependence between the order $k$ and the number of edges in an edge-reduced de Bruijn graph for all files from the test data set. A similar image was already published in the full version of [Bai+20] © 2020 IEEE.

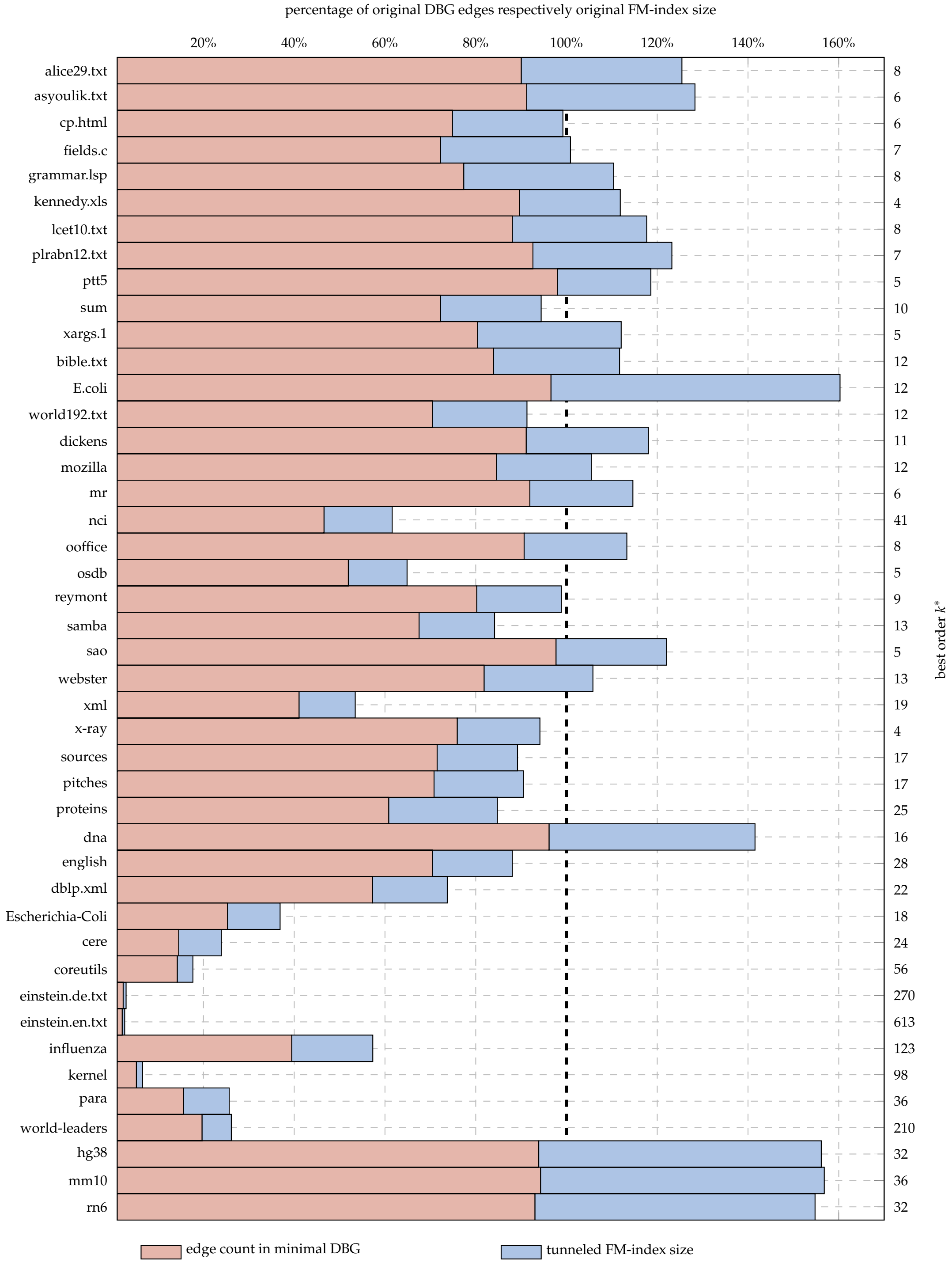


**Figure C.2:** Amount of reduced edges for minimal edge-reduced de Bruijn graphs as well as size of corresponding tunneled FM index compared to a normal FM-index. The overhead between the amount of reduced edges and the tunneled FM-index size comes from the two additional bit-vectors required in the tunneled FM index. A similar image was already published in the full version of [Bai+20] © 2020 IEEE.

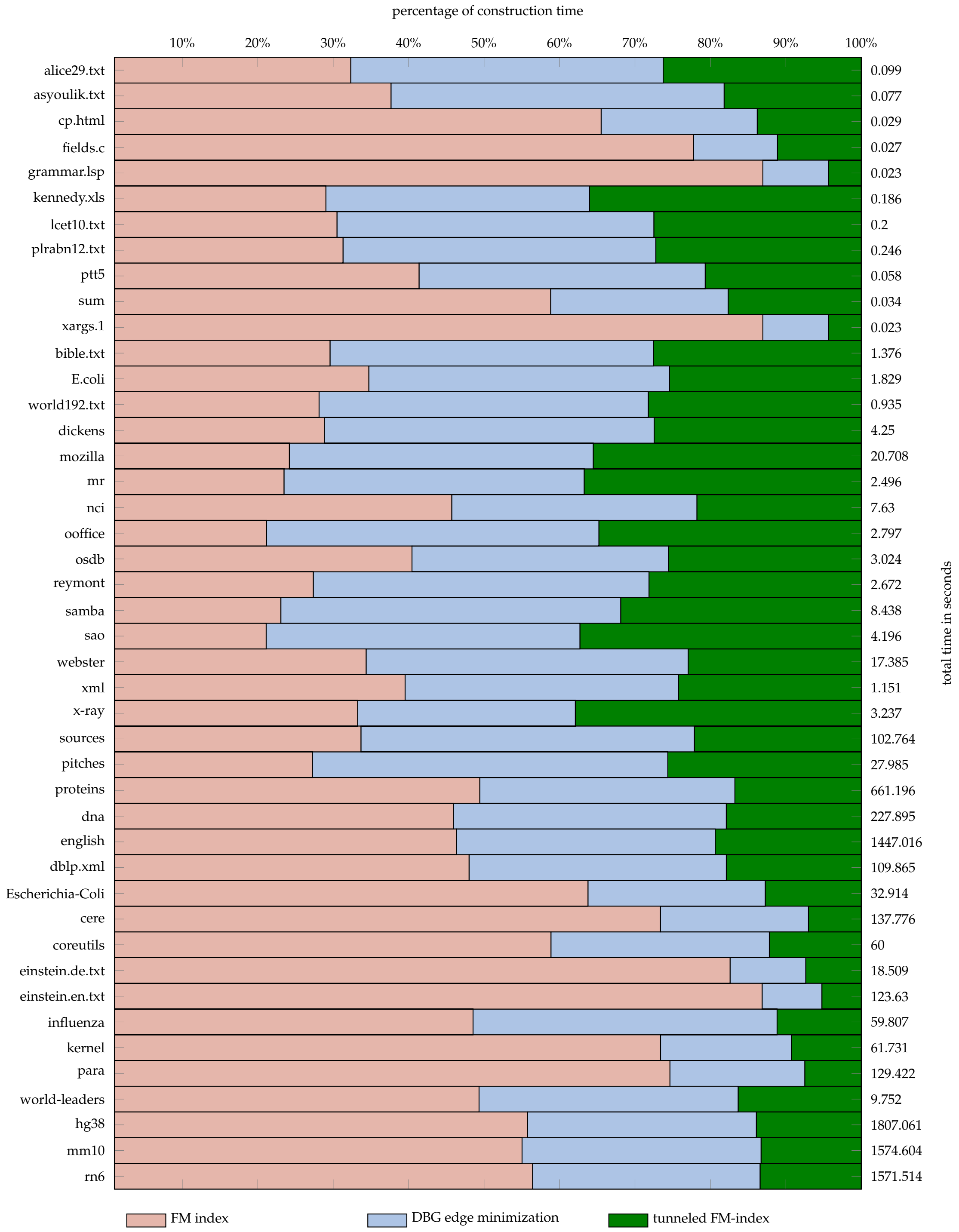


**Figure C.3:** Construction timings of tunneled FM-index construction broken down into FM-index construction, de Bruijn graph edge minimization and tunneled FM-index construction. The idea of this image comes from a student project [Web20].

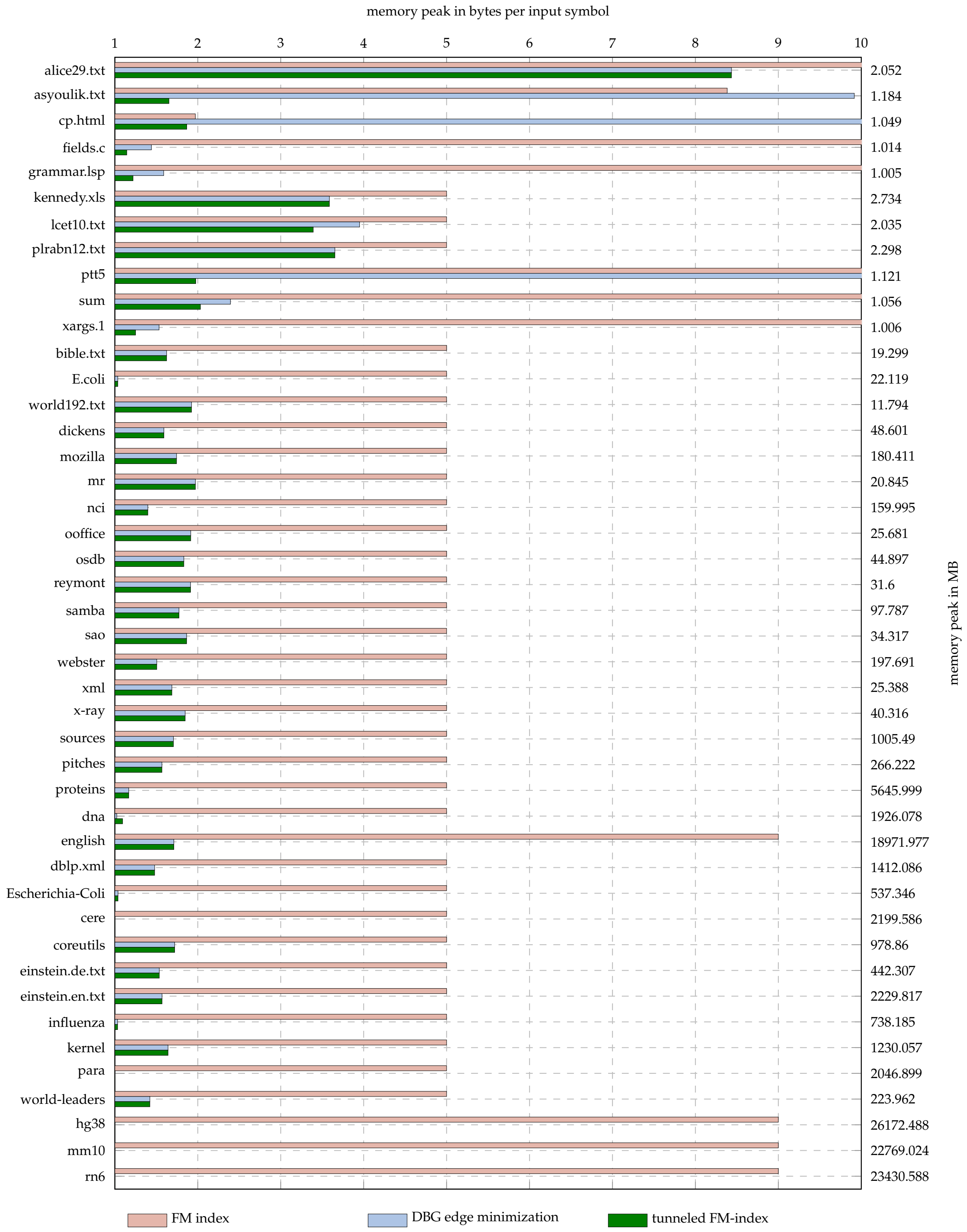


**Figure C.4:** Memory peak during tunneled FM index construction, measured in bytes per symbol of the input data. The idea of this image comes from a student project [Web20].

## C.2 TRIE RESULTS

This section presents results related to tries and trie tunneling, as described in the Sections 5.3 and 5.4. The test input comes from the data described in Chapter A. For each file, we removed all lines containing nullbytes and lines with less than 10 characters. To ensure correctness of the Aho-Corasick algorithm (see Algorithm 5.6), we also removed lines that contain any other line as a proper substring.

The test input now consists of all files listed in Chapter A which, after being modified as described above, have a size bigger than 1 MB and contain at least 1000 lines. The lines of each such file are used as input strings for trie construction. Both filter methods ensure a certain branching degree and size of the final trie. Table C.1 shows statistics about the prepared test data files.

| File | Size in MB | Number of strings |
|---|---:|---:|
| bible.txt | 3.841 | 30,105 |
| world192.txt | 1.826 | 32,650 |
| dickens | 9.345 | 157,038 |
| mozilla | 1.387 | 21,579 |
| nci | 10.594 | 178,289 |
| reymont | 6.103 | 6,058 |
| samba | 10.274 | 247,372 |
| webster | 30.598 | 491,661 |
| xml | 2.375 | 26,657 |
| x-ray | 3.199 | 23,172 |
| sources | 104.845 | 2,507,311 |
| proteins | 651.355 | 2,265,632 |
| dna | 383.493 | 1,864 |
| english | 988.750 | 16,709,728 |
| dblp.xml | 164.885 | 2,949,908 |
| coreutils | 10.746 | 230,982 |
| einstein.en.txt | 1.958 | 5,298 |
| kernel | 5.983 | 154,025 |
| world-leaders | 6.144 | 5,276 |

**Table C.1:** Statistics about the trie test data used for experiments. The files were generated as follows: first, all lines containing nullbytes or containing less than 10 characters were removed from the original file. Next, lines containing any other line as proper substring were removed. Resulting files with less than 1 MB or less than 1000 lines were filtered out. The lines of the resulting remaining files then were used as input strings for trie construction.

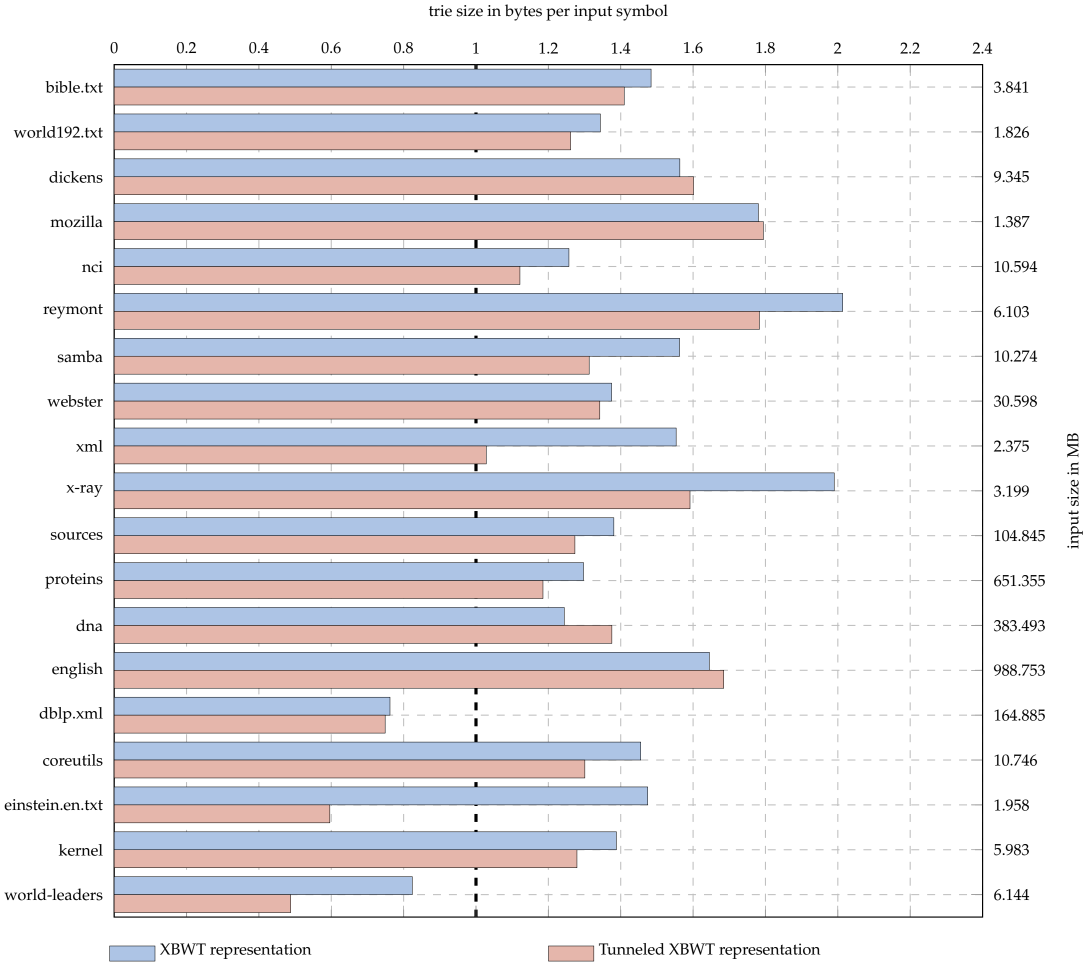


**Figure C.5:** Size of trie representations relative to the size of the sum of lengths of all input strings. A similar image already appeared in a student project [RHRH20].

The size of tunneled and normal tries using the XBWT representation can be found in Figure C.5. The size of a normal XBWT relative to the input length varies depending on the number of nodes in the trie. The tunneled variants offer compression about 10 % on average. Note that for some inputs like e.g. dna, the tunneled XBWT is bigger than the normal XBWT. This is in accordance with sizes of tunneled FM-indices (see Figure C.2): a tunneled BWT needs the additional component $D_{\mathsf{in}}$ which sometimes is bigger than the gain of tunneling. Again, best compression can be achieved on repetitive inputs.

Construction timings of the trie representations are shown in Figure C.6. This shows that the use of succinct counters in the algorithms (see Remark 5.14) has almost no influence on the run-time. The algorithms XBWT fast and XBWT lightweight come from [OSB18], algorithm XBWT fast also is described in Section 5.3. The

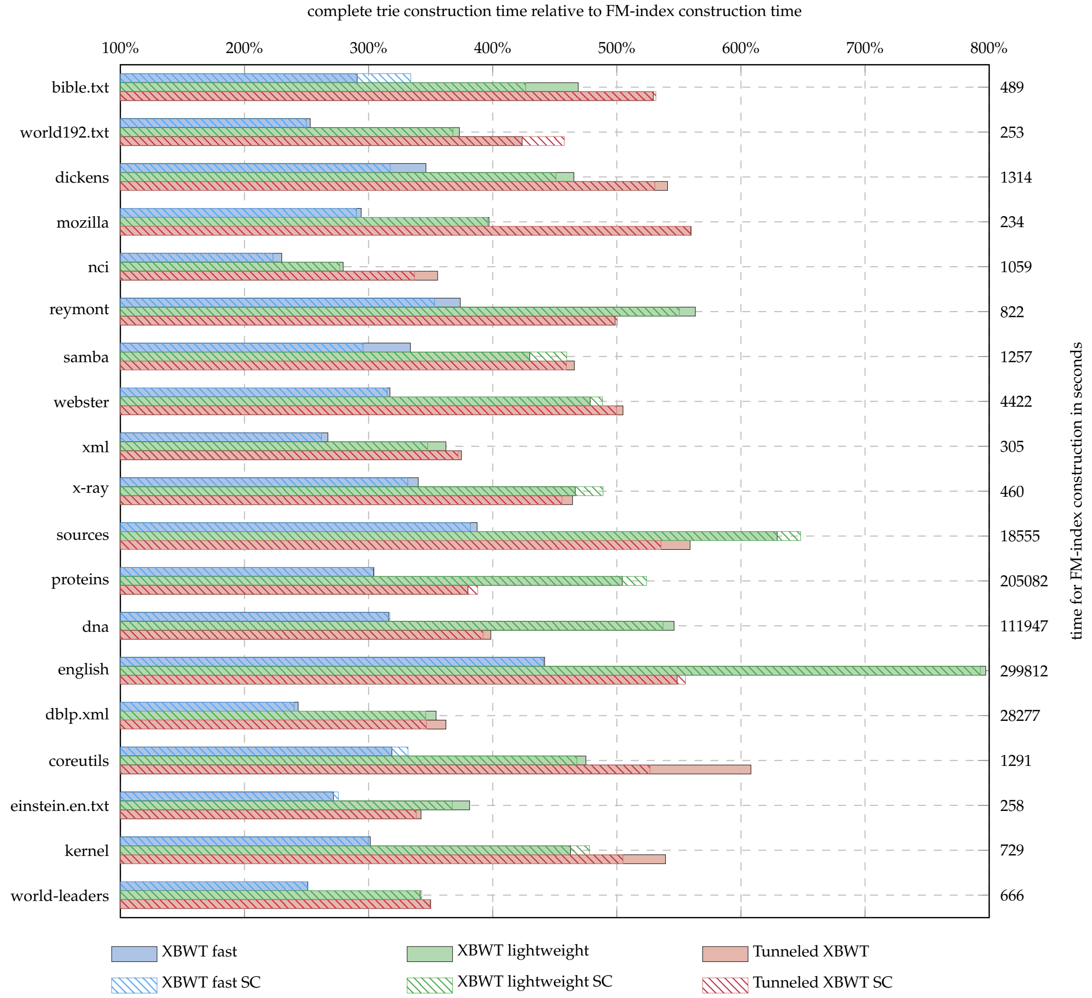


**Figure C.6:** Trie construction timings of different algorithms relative to FM-index construction. Algorithm variants using a succinct counter instead of a normal counter are indicated by the same color and a north west line pattern. Similar results on other test files already were presented in [OSB18] and [RHRH20].

tunneled XBWT construction algorithm has been described in Section 5.4 and is an extension of the XBWT fast algorithm.

Algorithm XBWT fast turns out to be the fastest construction algorithm under consideration. The lightweight XBWT construction algorithm requires roughly about twice as much time as the fast algorithm. This is owed to the double inspection of all nodes of the final trie, see [OSB18] for details. The TXBWT construction algorithm also requires roughly about twice the time of the fast algorithm. The reason is that the used de Bruijn graph edge minimization algorithm works very similar to a node inspection of the trie.

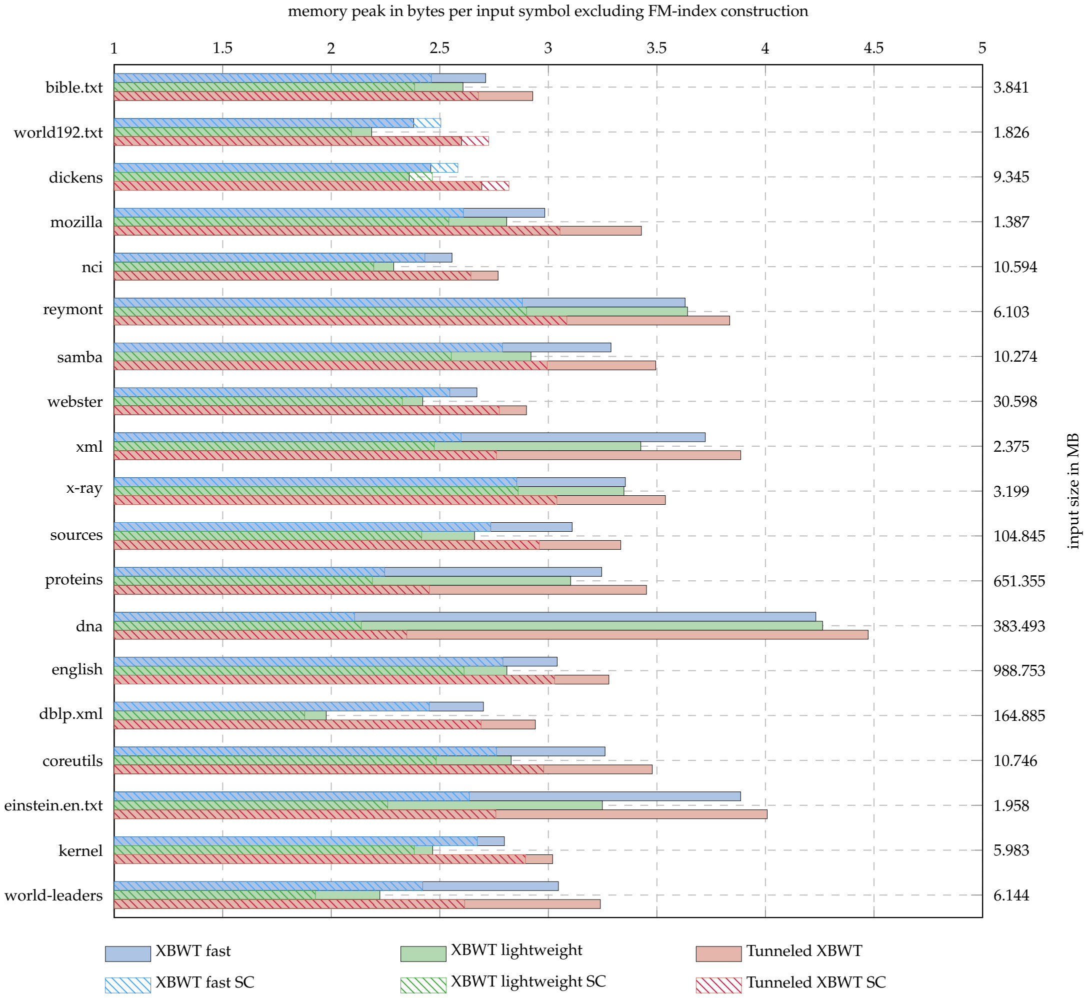


**Figure C.7:** Memory peak during trie construction and excluding FM-index construction. The peak is measured in bytes per symbol of the input data. Algorithm variants using a succinct counter instead of a normal counter are indicated by the same color and a north west line pattern. Similar results on other test files already were presented in [OSB18].

The memory peaks during trie construction (excluding FM-index construction) are shown in Figure C.7. The figure shows that the memory peak can be reduced most effective using succinct counters. The lightweight trie construction algorithm reduces the memory peak only in cases where the final trie has a small amount of nodes. This can also be seen by comparing Figure C.7 with Figure C.5. Tunneled XBWT construction requires about 10 % more memory peak than normal XBWT, mainly because of the additional component $D_{\text{in}}$.

We also compared the speed of the Aho-Corasick algorithm (see Algorithm 5.6) using the different trie representations. The tries were used to find all occurrences of

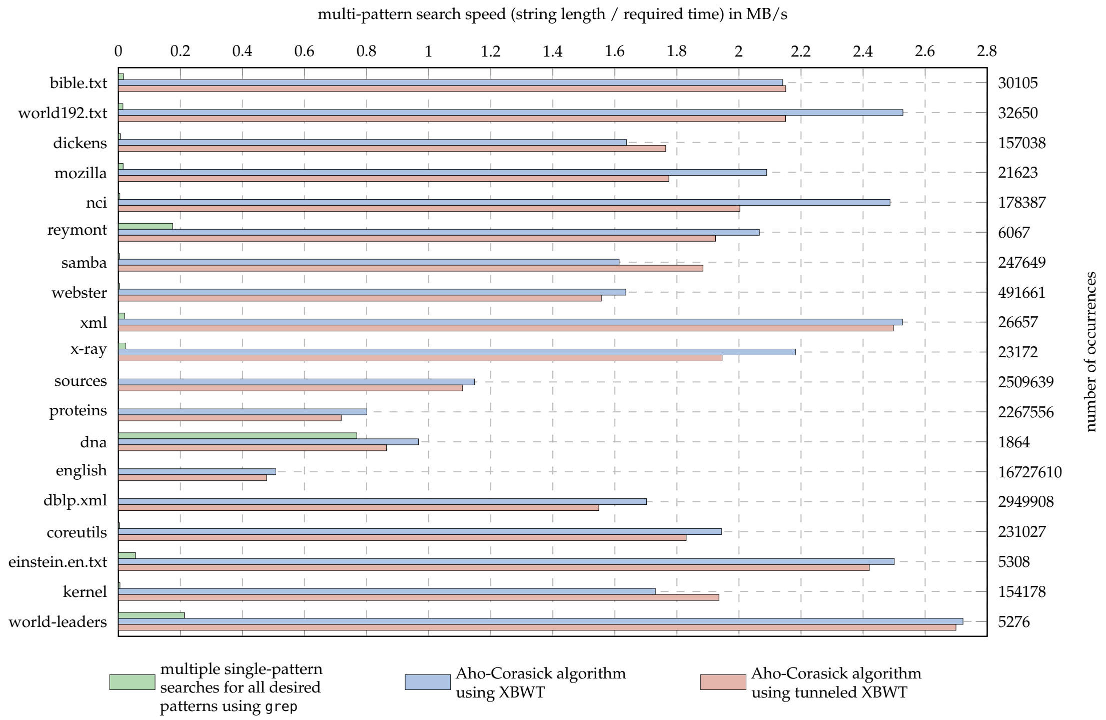


**Figure C.8:** Speed of the multi-pattern search using the Aho-Corasick algorithm compared to multiple single-pattern searches for all patterns using `grep`. The Aho-Corasick algorithm uses a priori constructed tries of the test files and searches for all occurrences of the test file lines within the same test file. The `grep` speed is estimated by executing `grep` with the last pattern of each file and multiplying the required time by the number of overall patterns.

the lines of the input files within the input files themselves. We removed all newline characters from the input files, so the amount of occurrences is generally larger than the number of strings in the trie.

For a better comparability, we measured the speed when using multiple single-pattern searches to find all occurrences. The used single-pattern search consists of the popular linux command `grep`[1] using one pattern per execution. We approximated the multi pattern search time of `grep` by executing one `grep` search with the last pattern in the test data and multiplying the time by the number of all patterns. This was necessary because the amount of time can get very large if the number of `grep` executions is big.

Unsurprisingly, Figure C.8 shows that it is beneficial to use multi-pattern search with an XBWT especially if the number of patterns is large. For a smaller amount of patterns, `grep` performance increases, see e.g. the test cases dna, reymont or world-

[1] https://www.gnu.org/software/grep/manual/grep.html

leaders in Figure C.8. Except for some outliers, the speed of multi-pattern search using a tunneled XBWT is a little bit slower than the speed using a normal XBWT. This is owed to the more complex navigation and failure link operations, but can be accepted because the speed decreases by only about 5 %.

# INDEX

# COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`