Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Efficient Construction of a Compressed Index for Large Text Collections

Master Thesis

2024
Bc. Klára Sládečková

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Efficient Construction of a Compressed Index for Large Text Collections

Master Thesis

Study Programme:	Informatics
Field of Study:	Computer Science and Biology
Department:	Department of Computer Science
Supervisor:	doc. Mgr. Bronislava Brejová, PhD.
Consultant:	Andrej Baláž, MSc.

Bratislava, 2024
Bc. Klára Sládečková

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Klára Sládečková
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Efficient Construction of a Compressed Index for Large Text Collections
Efektívna konštrukcia komprimovaného indexu pre veľké zbierky textov

Anotácia: Indexovanie textu hrá kľúčovú úlohu v rôznych oblastiach, ako je bioinformatika, softvérová bezpečnosť a spracovanie prirodzeného jazyka, kde je nevyhnutné spracovávať veľké objemy textových údajov. Mnoho aplikácií sa spolieha na transformáciu Burrows-Wheeler na rýchle vyhľadávanie konkrétnych vzorov v texte. Pokrok v tejto oblasti viedol k vývoju Wheelerových grafov, zovšeobecnenej formy Burrows-Wheelerovej transformácie. Wheelerove grafy ponúkajú znížené pamäťové nároky indexu, ale ich optimálna konštrukcia je NP-ťažká. Výsledkom je, že na konštrukciu Wheelerovho grafu sa používajú heuristické algoritmy, čo ponecháva priestor na ďalšie vylepšenia.
Primárnym cieľom tejto práce je preskúmať, navrhnúť a implementovať nový algoritmus na vytváranie Wheelerových grafov. Navrhnutý algoritmus bude zameraný na prekonanie súčasných obmedzení tým, že umožní vytvorenie vnorených tunelov v rámci štruktúry Wheelerovho grafu. Študentka navrhne efektívny algoritmus schopný zostaviť Wheelerove grafy s vnorenými tunelmi, implementuje navrhnutý algoritmus pomocou vhodného programovacieho jazyka a vyhodnotí výkon algoritmu a kvalitu výsledných Wheelerových grafov na reálnych pangenomických dátach.

Vedúci: doc. Mgr. Bronislava Brejová, PhD.
Konzultant: Andrej Baláž
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 16.12.2022

Dátum schválenia: 16.12.2022 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Klára Sládečková
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Efficient Construction of a Compressed Index for Large Text Collections

Annotation: Text indexing plays a crucial role in various domains such as bioinformatics, software security, and natural language processing, where handling large volumes of textual data is essential. Many applications rely on the Burrows-Wheeler transform to quickly search for specific patterns within the text. Advancements in this field have led to the development of Wheeler graphs, a generalised form of the Burrows-Wheeler transform. Wheeler graphs offer reduced space requirements of the index, but their optimal construction is NP-hard. As a result, heuristic algorithms are used for Wheeler graph construction, leaving room for further improvements.
The primary objective of this thesis is to explore, design, and implement a novel algorithm for constructing Wheeler graphs. The proposed algorithm will aim to overcome current limitations by allowing the creation of nested tunnels within the Wheeler graph structure. The student will design an efficient algorithm capable of constructing Wheeler graphs with nested tunnels, implement the designed algorithm using appropriate programming language, and evaluate algorithm's performance and the quality of resulting Wheeler graphs on real pangenomic datasets.

Supervisor:	doc. Mgr. Bronislava Brejová, PhD.
Consultant:	Andrej Baláž
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	16.12.2022

Approved: 16.12.2022 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

Student Supervisor

Abstrakt

Wheelerove grafy sú dôležitou súčasťou komprimovaných textových indexov, menovane FM-indexu, prvýkrát opísaného v súvislosti s Burrows-Wheelerovou transformáciou. Výpočet Wheelerovho grafu pre textové kolekcie je ľahký, avšak dosiahnutie komprimovanej verzie Wheelerovho grafu je značne náročnejšie. V predošlých prácach bol tento problém riešený rôznymi pažravými algoritmami a heuristikami. Najlepšie z týchto prístupov však neuvažovali dôležitú výhodu komprimácie - tunelovanie kompenzovateľných kolízií. V tejto práci uvádzame inovatívny prístup, ktorý túto výhodu využíva, čím dosahuje lepšie výsledky v kompresii.

Kľúčové slová: Wheelerov graf, blok, tunelovanie, kolízia

Abstract

Wheeler graphs play an important role in compressed text indexing, namely FM-index, first described for the Burrows-Wheeler transform. Computation of Wheeler graphs for text collections is easy, however, achieving a compressed version of the Wheeler graph is considerably harder. In previous works, this problem was approached using different greedy algorithms and heuristics, which neglected a specific convenience of the compression - tunnelling of compensable collisions. We introduce a novel approach utilizing this convenience and thus providing better compression rates.

Keywords: Wheeler graph, block, tunneling, collision

Contents

List of Figures

Introduction

In 1994, David J. Wheeler and Michael Burrows introduced a novel text transformation, nowadays recognized as the Burrows-Wheeler transform (BWT). This transformation offers several advantages: it is reversible, can be indexed in a way that is space efficient and allows for fast traversal and, on top of that, possesses the trait of compact representation. Consequently, over the last years, BWT has become popular and indispensable across various domains, extending its utility from data compression to areas like sequence analysis and bioinformatics.

In 2017, Gagie et al. expanded on the BWT concept by introducing Wheeler graphs, providing a versatile framework for various BWT-based data structures. This development highlighted the adaptability and broad applicability of the BWT approach.

Given the popularity and effectiveness of the BWT in compression applications, numerous attempts have been undertaken to enhance the compressibility of BWT-based compressors. Many of these approaches involve applying additional text transformations to the BWT before employing entropy encoding techniques to achieve compression. In 2018, Uwe Baier introduced a compression technique known as "tunnelling," which focuses on compressing the BWT prior to subsequent transformations. Notably, tunnelling on Wheeler graphs preserves the structure's compatibility with efficient indexing without the need for decompression - a highly valued attribute, mainly when indexing enormously large datasets. This innovation is applicable to all index structures that can be explained in terms of Wheeler graphs.

This compression is mostly useful for repetitive data, such as pan-genomic data - our main focus. In the field of genetics, a genome is the complete genetic material of a single organism, and a pan-genome describes a collection of such genomes within one species. Therefore, these data are often highly repetitive, since the genomes within a pan-genome are usually very similar and even a single genome can be quite repetitive.

However, a year after introducing the tunnelling technique, U. Baier and K. Dede proved it to be NP-complete, in the general case. Several approaches were since proposed, aiming to find a reasonable compression in a short time. While most of these methods deprive themselves of a certain advantage in the compression, potentially leading to worse compression rates in order to minimize the time, our work takes a divergent approach. Our aim is to exploit this advantage in order to find more com-

pressed versions while increasing the time complexity minimally.

Chapter 1

Preliminaries

The purpose of this chapter is to introduce necessary concepts and outline the core challenges related to our study. In the first section, we present the concept of the Wheeler graphs (WG), a generalized structure of the Burrows-Wheeler transform. This structure supports a specific compression technique discussed in the second section of this chapter. This compression, however, poses a challenge as it is generally an NP-hard problem, further described in the third section of this chapter. In the final section of this chapter, we delve into various greedy algorithms and heuristics proposed in the past. Among these, the most efficient ones have sacrificed a convenient property of compression, explained in the third section. In this work, we introduce a novel heuristics capable of preserving this property.

Throughout this work, logarithms are assumed to be of base two. Additionally, let alphabet Σ denote a set of totally ordered characters with relation $\prec$ representing the order between two characters, such that the character $\$ \in \Sigma$ is considered the lowest ordered character in Σ. Each string S over the alphabet Σ is in this thesis null-terminated, meaning the character $\$ appears at the end of S and nowhere else in the string.

Let S be a string of length $n \in \mathbb{N}$ over the alphabet Σ, and i be an integer in the interval $[0, 1, ..., n-1]$. We denote:

- $S[i]$: the i-th character in S.
- $S[i..]$: the suffix of S starting at position i, defined as $S[i..] = S[i]S[i+1]...S[n-1]$.
- $S[..i]$: the prefix of S ending at position $i-1$, defined as $S[..i] = S[0]S[i]...S[i-1]$.
- $S \prec_{lex} S'$: S is lexicographically smaller than S' (similarly defined for $\succ_{lex}$).
- $|S| := n$: the length of the string S.

1.1 Wheeler Graph

The concept of Wheeler graphs was first introduced in 2017 by T. Gagie, G. Manzini, and J. Sirén [12]. The idea was based on the Burrows-Wheeler transform, a significant data structure in stringology used for compressed text representations. BWT's convenience is based on its ability to effectively perform queries directly on the compressed transformation. Such a space-efficient compressed index is highly beneficial in bioinformatics fields, where multiple genome files of large size need to be stored and queried. In order to generalize this data structure and its variants, Wheeler graphs were introduced.

Formally, a directed graph G is defined as a pair of sets (V, E), where V represents a finite set of vertices and $E \subseteq V \times V$ denotes a set of edges.

Definition 1.1.1 (Wheeler Graph)**.** Let $G = (V, E)$ be a directed, labelled graph, with the function $\lambda : E \longrightarrow \Sigma$ defining the label of each edge. We call the graph G a *Wheeler graph* if and only if there exists a total order of the vertices, satisfying the following conditions:

- Vertices with zero in-degree precede those with a non-empty set of incoming edges.
- For any $(u_1, v_1), (u_2, v_2) \in E$, if $\lambda((u_1, v_1)) \prec \lambda((u_2, v_2))$, then $v_1 < v_2$.
- For any $(u_1, v_1), (u_2, v_2) \in E$, if $\lambda((u_1, v_1)) = \lambda((u_2, v_2))$ and $u_1 < u_2$, then $v_1 \leq v_2$.

This order is called the *Wheeler order*.

The concept of the Wheeler order can be approached more straightforwardly as follows. When visually representing the graph with two copies of each node, sorted according to the Wheeler order, certain properties become apparent. The nodes with a zero in-degree form a consecutive interval at the beginning of the order, due to the first condition. The second condition ensures that the labels of incoming edges are arranged by the lexicographic order of the alphabet. Lastly, the third condition implies that edges sharing the same label do not intersect in this depiction of the Wheeler graph. All of these observations are depicted in Figure 1.1.

The authors [12] observed several convenient properties of this data structure. Firstly, it is possible to create a Wheeler graph from a text in linear time if the alphabet is constant (in contrast, it is an NP-hard problem to decide whether a graph has the Wheeler graph properties [13]). Secondly, Wheeler graphs can be compactly represented and traversed using no more than three arrays with additional data structures supporting efficient rank and select operations (these arrays then coincide with

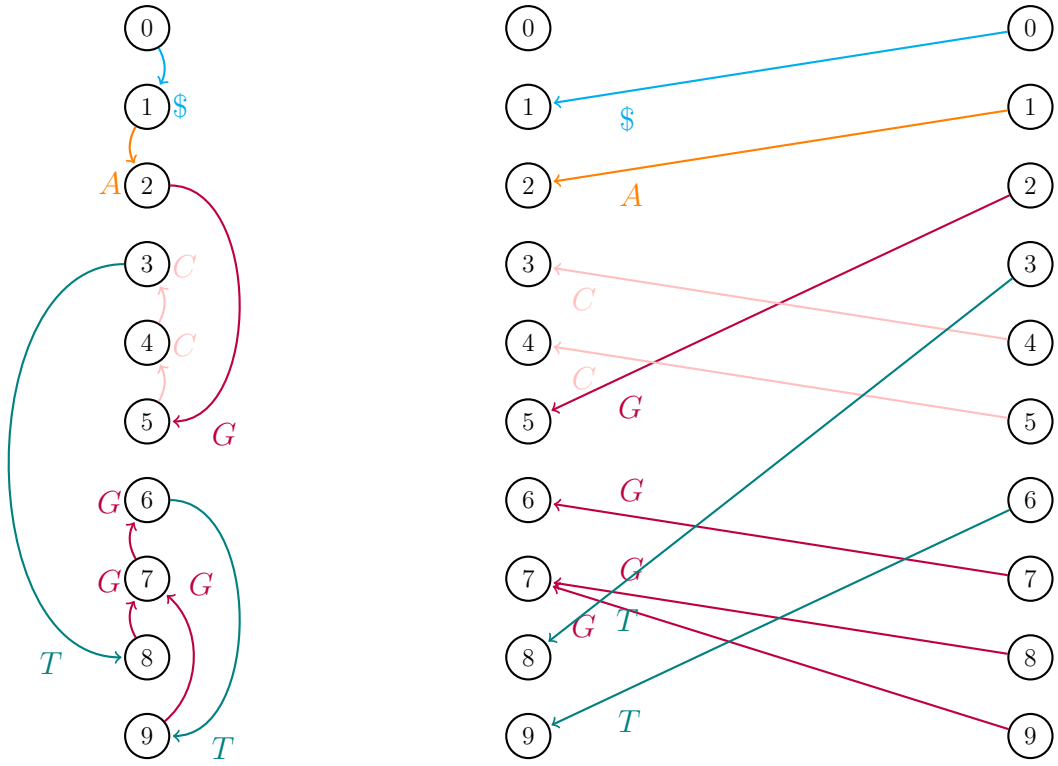


Figure 1.1: An example of a Wheeler graph. Left: Wheeler graph with nodes numbered in the Wheeler order. Right: The same graph with two copies of each node, where edges are going from the right copy of a node to the left one. Labels of incoming edges are ordered alphabetically and no two edges of the same label cross.

the output of many BWT variants). Thirdly, FM-index [11] (usually associated with BWT) can be applicable to Wheeler graphs as well, which leads to the generalization of the BWT concept while preserving its advantages. And, lastly, the property of Wheeler graphs called *path coherence*, useful for a specific compression that preserves all the previously mentioned properties, which we present in the next section.

Notably, any null-terminated string of length n can be represented as a graph with n nodes, with edges creating a cycle labelled by the reverse of the text, as shown in Figure 1.2. So starting from the node with an incoming edge labelled \$, by following the single path, the original text is created. While it is generally NP-hard to determine whether a graph satisfies the conditions of being a Wheeler graph, the process of finding a Wheeler order in a graph representing a text is relatively straightforward. The idea is inspired by the mentioned graphical observations of Wheeler graphs, which leads to computation of the *Burrows-Wheeler matrix* of a string.

Definition 1.1.2 (Burrows-Wheeler matrix)**.** Let S be a string over the alphabet Σ, with $|S| = n$. The *Burrows-Wheeler matrix* (BW matrix) M consists of lexicographically ordered cyclic rotations of the string S as its rows, i.e. two conditions for such a

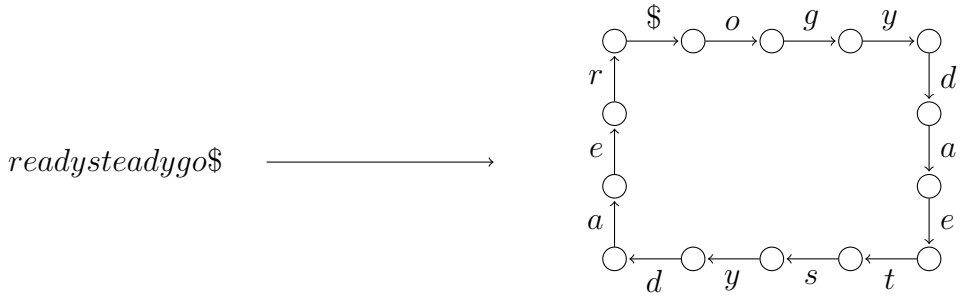


Figure 1.2: The process of constructing a graph from a text. The graph is a cycle and by following the path starting with the edge labelled \$, the reverse of the original string can be reconstructed, by concatenating the labels encountered on the path.

matrix must hold:

- For every $i \in \{0, ..., n-1\}$, there exists $j \in \{0, ..., n-1\}$ such that $M[i] = S[j..]S[..j]$
- For every $i \in \{1, ..., n-1\}$, $M[i-1] \prec_{lex} M[i]$.

We denote the last column of the matrix by L and its first column by F.

The first condition ensures that every row of the Burrows-Wheeler matrix represents a cyclic permutation of the original string. The second condition ensures that the rows are lexicographically ordered from top to bottom and are distinct. An illustration of the Burrows-Wheeler matrix is provided in 1.3.

The BW matrix of a string offers several conveniences, as shown throughout the work. Most notable is its last column L, which represents the Wheeler order of nodes of the graph representing the string. This column is known as the Burrows-Wheeler transform (BWT), first described in 1994 by M. Burrows and D. J. Wheeler [9], and later expanded to the concept of Wheeler graphs. Since we operate on biological data, which are represented as text, we focus mainly on this class of Wheeler graphs throughout the work.

It is worth noting that the F column of the BW matrix is simply the sorted version of the L column and, therefore can be derived from the BWT. Moreover, with just the BWT of a string S, its original form can be reconstructed, and consequently the entire BW matrix. This reconstruction process relies on a technique called LF-mapping, which enables the identification and mapping of each position of a character in L to its corresponding position in F [1]. To present this method, we first introduce some notation.

We employ the notation $\texttt{select}_S(c, i)$ to denote the position of the i-th occurrence of character c in string S, $\texttt{rank}_S(c, i)$ to represent the number of occurrences of character

c in the prefix $S[..i]$, and $\mathtt{C}_S[c]$ to signify the number of characters in S ordered lower than c, defined as $\mathtt{C}_S[c] := |\{i \in \{0, ..., n-1\} | S[i] \prec c\}|$.

Definition 1.1.3 (LF-mapping [3])**.** Let S be a string over the alphabet Σ, with $|S| = n$, and L be its BWT. The *LF-mapping* LF is a permutation of integers $0, ..., n-1$ such that for any $i \in \{0, ..., n-1\}$, $LF[i] = \mathtt{C}_L[L[i]] + \mathtt{rank}_L(L[i], i)$. We denote by LF^{-1} the inverse of this permutation.

In other words, for each $i \in \{0, ..., n-1\}$, $LF[i]$ gives the position of the ocurrance of character $L[i]$ in column F. A noteworthy aspect underlying this definition is the inherent ordering consistency between equal characters in both strings L and F. This arises from the fact that L references to suffixes shifted one character to the left compared to those represented by F. For instance, consider BW matrix M and two positions i and j in L where $i < j$ and $L[i] = L[j]$. Since $i < j$, it follows that $M[i] \prec_{lex} M[j]$, resulting in $M[LF[i]] \prec_{lex} M[LF[j]]$ due to the equality $L[i] = L[j]$ (note that $M[LF[k]] = L[k]M[k][0]...M[k][n-2]$). Consequently, the corresponding position to $L[i]$ in F is smaller than the corresponding position to $L[j]$ in F, thereby ensuring uniformity in precedence order. An example of this observation is depicted in Figure 1.4, where, on the left, no grey edges going from equally labelled characters cross. This property of the BWT is known as *path coherence* in terms of the Wheeler graph, explained in the next section.

Given that the rows of the BW matrix represent cyclic permutations of the initial string, the position of any character in column F is the position of the preceding character (in the sense of the initial string) in column L. Therefore, the LF-mapping serves as a fundamental tool for identifying the position in the L column where the preceding character appears. Consequently, by initiating from the position of the character \$ in L and following the LF-mapping, the reverse of the initial string can be computed. This implies a great feature of the BWT (and WG) - its invertibility. In the subsequent section, we explore another crucial property of the Wheeler graph: its capacity for compression while preserving convenient characteristics such as invertibility and efficient indexing.

1.2 Tunneling

The rationale behind our focus on the Wheeler graph lies in its ability to cluster the vertices with equally labelled incoming and outgoing edges. This characteristic - called *path coherence* [12] - was the base idea for the compression technique applicable to BWTs (and later to Wheeler graphs) called tunnelling. The path coherence is presented in the WG by grouped occurrence of nodes in the Wheeler order, when the nodes are reached by equally labelled paths from nodes that form a consecutive range in the

Wheeler order themself. Therefore, remembering only the size of the group and the labels of the path is sufficient, and thus the storage requirements of the Wheeler graph are reduced. Given the repetitive nature of biological data and the frequent need for their indexing, this approach is quite significant in bioinformatics, especially concerning the pan-genomic sequences.

Before delving into the tunnelling method, we first introduce the definition of a block within a Wheeler graph, for the block serves as the fundamental unit in this compression technique.

Definition 1.2.1 (Block)**.** Let $G = (V, E)$ be a Wheeler graph with label function λ and $w, h \in \mathbb{N},\ w \geq 4,\ h \geq 2$. Furthermore, let $P = (v_{0,0}, v_{1,0}, ..., v_{w-1,0}), ..., (v_{0,h-1}, v_{1,h-1}, ..., v_{w-1,h-1})$ be a set of acyclic paths in the graph. We call the subset $(v_{1,0}, ..., v_{w-1,0}), ..., (v_{1,h-1}, ..., v_{w-1,h-1}) \subseteq P$ a *block* of height h and width $w - 2$ (number of edges in one path) if the following conditions are fulfilled:

- all nodes in P except $v_{0,0}, ..., v_{0,h-1}$ have in-degree equal to one and all nodes except $v_{w-1,0}, ..., v_{w-1,h-1}$ have out-degreee equal to one
- the paths are parallel, that is $\forall i \in \{1, ..., w-1\}, \forall j \in \{0, ..., h-2\}$ the node $v_{i,j+1}$ is the immediate successor of the node $v_{i,j}$ in terms of the Wheeler order
- $\forall i \in \{0, ..., w-2\}, \forall j \in \{0, ..., h-2\}\ \ \lambda((v_{i,j}, v_{i+1,j})) = \lambda((v_{i,j+1}, v_{i+1,j+1}))$ (the paths are equally labeled)

  We call each path of the block a *row*, and the set of edges $\{(v_{i,0}, v_{i+1,0}), ..., (v_{i,h-1}, v_{i+1,h-1})\}$ is called *column* of the block, for each $i \in \{1, ..., w-2\}$. If $i = 1$ we say the column is *right-most*, and, analogously, if $i = w - 2$ we say the column is *left-most*.

Notably, the definition requires that the incoming edges to the first vertices within the block paths must be equally labelled, regardless of whether the originating nodes of these edges form a consecutive sequence in the Wheeler order. This condition distinguishes our definition from previously published ones [7, 2], where the incoming labels of the nodes in the right-most column are permitted to vary. This alternation of the definition was inspired by Baier's de Bruijn graph edge minimization heuristics [5], which is able to tunnel such blocks.

In the context of BW matrices, the block is simply a rectangle in the BW matrix with identical rows (ensuring the third condition), whose last column is aligned with the L column (ensuring the second condition). The first condition is held inherently, as the nodes in the graph have exactly one incoming and one outgoing edge. Our additional condition ensures that the characters in F column at positions of the alignment of the block to the L column are labelled the same. Notably, all blocks according to

the original definition are also computed upon this definition, only mapped to the L-column shifted one-column to the right, if they do not fit the alternative definition. Thereby, the extra condition ensures enhancing compression opportunities, without compromising any essential criteria for the blocks - their compression ability.

We say the block is *height maximal* (resp. *width maximal*) if it cannot be extended to height (resp. width) while preserving the block conditions, and *maximal* if it is both height and width maximal. An example of a maximal block within the WG and BW matrix is depicted in Figure 1.3.

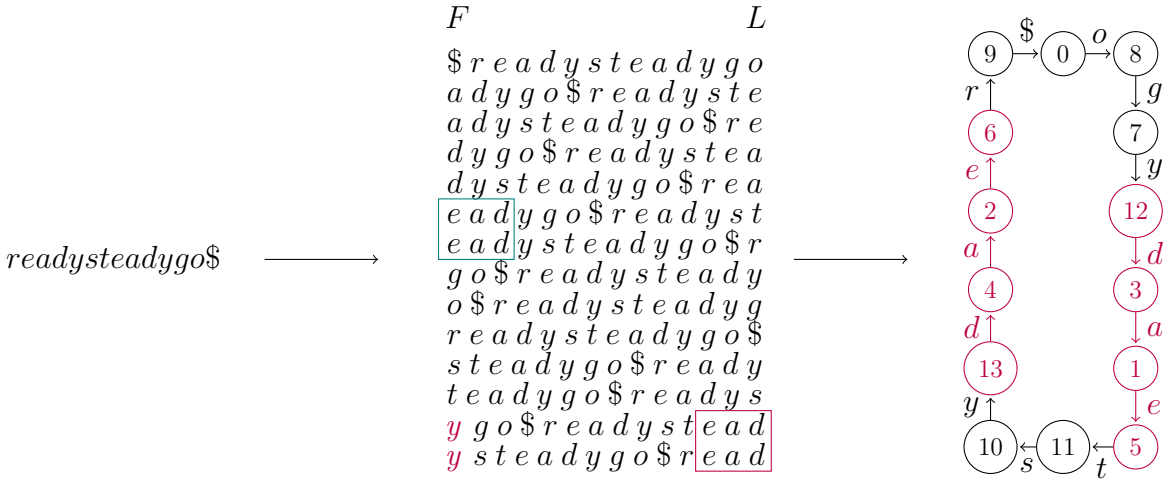


Figure 1.3: In the middle is the Burrows-Wheeler matrix for string S = *readysteadygo*\$. The rightmost column is the BWT L of the initial string, and the leftmost column denotes the F-column. Cyclic permutations of the initial string are ordered lexicographically, creating the BW matrix. Above, one maximal block is depicted aligned to both L and F columns. Our additional condition is depicted by the red-marked y in the F column, with the same block marked in red. The right-most picture depicts the WG created from the original string. The nodes are marked by the Wheeler order, achieved from the BW matrix. The same block is marked by red color.

Note that rectangles of blocks occur in the BW matrix more than once. Their appearance may be aligned to the F column as well since the rows are equal and strings preceding them form a continual interval in the BW matrix. However, this does not hold conversely (as shown in the Figure, where a greater rectangle of equal rows is aligned to the F column), therefore we restrict the definition to blocks that can be aligned to the L column solely.

The following definition underlines the significance of a block in a Wheeler graph in terms of compression.

Definition 1.2.2 (Tunneling [2])**.** Let $G = (V, E)$ be a Wheeler graph with a label function λ, and let $(v_{0,0}, v_{1,0}, ..., v_{w-1,0}), ..., (v_{0,h}, v_{1,h}, ..., v_{w-1,h-1})$ be the paths of a block

b of graph G. The tunnelling process on b is defined as merging the set of nodes $v_{i,0}, ..., v_{i,h-1}$ into a new node v_i for $0 < i < w - 1$, where the new edge labels are inherited from the original (equal) edge labels. The newly generated graph is called a *tunnelled Wheeler graph*.

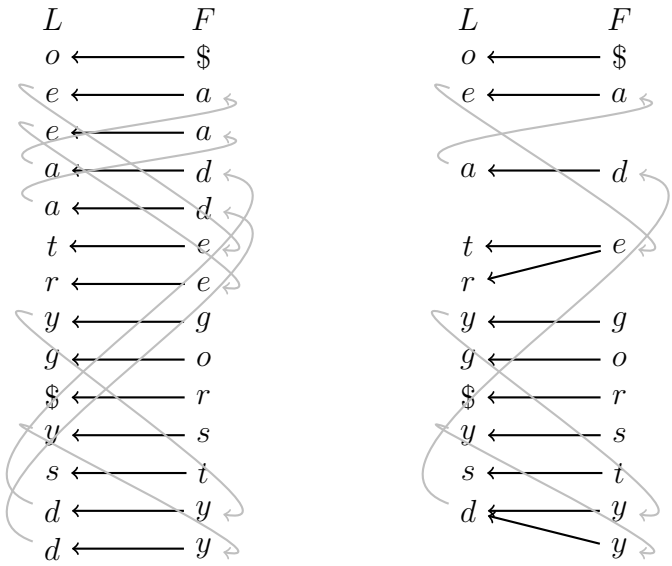


Figure 1.4: The process of tunneling. Above, the block from Figure 1.3 of string *readysteadygo*\$ is tunnelled. Arrows coloured grey show parts of the LF-mapping.

An example of the tunnelling process is illustrated in Figure 1.4. It was proven that a tunnelled Wheeler graph remains a Wheeler graph [3]. Hence, the process of tunnelling could be iteratively repeated, leading to increased compression benefits with each iteration. However, the blocks must be carefully and deliberately selected, a task more complex than initially apparent, as we discuss in the next section.

Notably, tunnelling of a bigger block is generally more beneficial than tunnelling of a smaller block, as the compression increases with the block size. For a block b, we define its *profit* by the difference in Wheeler graph size (i.e. number of edges) before and after tunnelling, denoted as $|G| - |\tilde{G}|$, where $|G|$ represents the size of the graph before tunnelling and $|\tilde{G}|$ represents the size of the graph after tunnelling block b. Analogously, we define the profit of a set of blocks B, as the difference $|G| - |\tilde{G}|$.

The profit of a single block b with height h and width w can be calculated as $w \cdot (h - 1)$, as the first row remains preserved in the $\tilde{G}$. Note that the profit would change to $(w-1) \cdot (h-1)$ in some cases, if not of our additional condition in the block definition. Such a case is illustrated in the Figure 1.3. Under the original definition, the same block would still be computed, however, during the tunnelling process, the right-most column would remain uncontracted, due to the lack of information about the nodes accessible from this column.

The computation of the profit of a set of blocks is, in contrast, not so straightforward, as the blocks may share some nodes and edges. Even though the nodes and edges may be present in multiple blocks, they can be removed only once, and therefore the profit of a set of blocks may differ from the sum of profits of the blocks within the set. Moreover, not every block collision is suitable for tunnelling, which brings us to the Block choice problem.

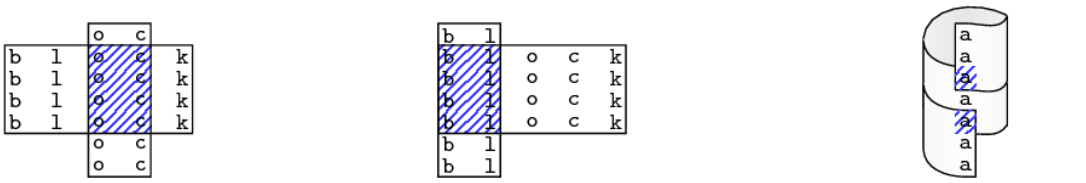
1.3 The Block choice problem

As mentioned in the preceding section, we permit blocks to overlap. Based on the outcome of tunnelling, we distinguish two types of such overlappings [3]. Tunnelling overlapping blocks may sometimes lead to a non-reversible Wheeler graph, a scenario we aim to avoid.

Definition 1.3.1 (Colliding blocks)**.** Let $G = (V, E)$ be a Wheeler graph with a label function λ. Blocks b and b' of G are *colliding* if there exists node $v \in V$ such that $v \in b$ and $v \in b'$. Furthermore, let b_{in} and b_{out} be two colliding blocks in G, with b_{in} being higher than b_{out}. We denote the collision of blocks b_{in} and b_{out} as *compensable* if the following conditions are met:

- The rightmost and leftmost columns of block b_{out} do not intersect with b_{in}
- At least one row of b_{in} does not intersect with b_{out}.
- The intersection forms a block of width $w_{b_{in}}$ and height $h_{b_{out}}$, where $w_{b_{in}}$ is the width of block b_{in} and $h_{b_{out}}$ is the height of block b_{out}.

If $b_{in} = b_{out}$ and v is present in multiple locations of the block, we call the collision a *self-collision*. If at least one of these conditions is not met, or if the collision is a self-collision, we classify the collision as *critical*.



a) compensable collision **(b)** critical collision **(c)** critical self-collision

Figure 1.5: Visualization of all types of block collisions. Shared regions of blocks are marked with blue diagonal stripes. A similar image was already published in [3].

Figure 1.5 illustrates examples of these collision types. This categorization of collisions is intended to determine which blocks are suitable for tunnelling. When tunnelling

a critical collision, the reversibility of the tunnelled Wheeler graph is compromised, resulting in an ambiguous compressed version, as will be explained throughout the work. On the contrary, compensably colliding blocks can be tunnelled without violating this valuable property, as shown in Figure 1.6. Hence, we introduce the following problem.

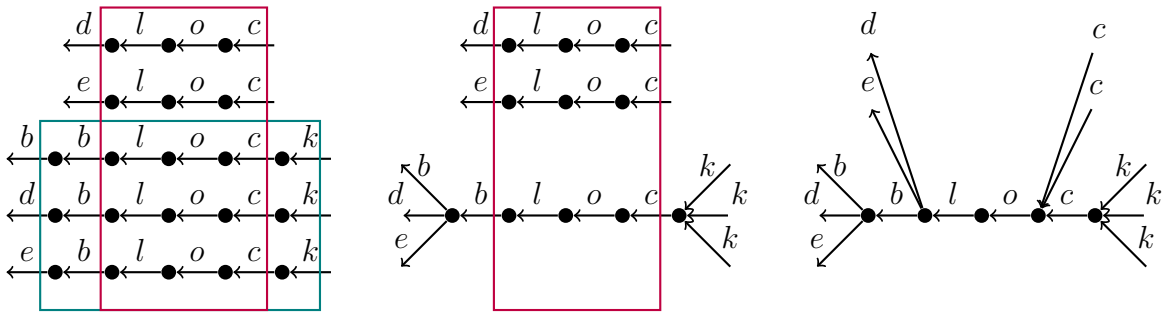


Figure 1.6: The process of iterative tunneling of a compensable collision. Left: two blocks colliding in a compensable manner. Middle: the wider block is tunnelled, the higher block is not yet tunnelled. Right: Both of the blocks are tunnelled, sequentially.

Definition 1.3.2 (The Block choice problem)**.** The *Block choice problem* entails determining an optimal set of blocks M from a Wheeler graph G which does not involve a critical collision, and the advantage of tunnelling is maximized, meaning that no other set of non-critically colliding blocks of G offers better compression rates (larger profit).

The Block choice problem has been proven to be NP-complete [7]. Several approaches aiming for a polynomial time solution have been proposed in the past, some of which are detailed in the next section.

1.4 Current state of the problem

In 2018, Baier introduced a greedy algorithm for selecting blocks [3], demonstrating a time complexity of $O(n \log|\mathcal{RB}|)$, where $\mathcal{RB}$ denotes the set of so-called "width-maximal run-blocks" and n signifies the length of the Burrows-Wheeler transform. This algorithm is based on the selection of a block from the set of width-maximal run-blocks, aiming to identify the block with the highest compression rate. Upon selecting a block, it is removed from the set and the algorithm undergoes iterations wherein it adjusts the profits of the remaining blocks, depending on their type of collision with the chosen block, thus ensuring that the relative information is considered in subsequent rounds. While this method serves as a foundational approach, Baier himself acknowledged many instances where the greedy strategy may not yield the optimal solution. Due to its complicated nature and resource-intensive demands, this approach is deemed somewhat impractical.

In another publication by Baier and Dede in 2019 [7], a simple heuristics was presented, outperforming existing solutions for the Tunnel-planning problem in terms of both resource requirements and compression rate. Utilizing Baier's concept of width-maximal run-blocks, the authors adapted the cost model to a single block. Tunnelling a block brings a profit to the compression, because of the number of edges removed from the BWT, but also a cost due to additional markings of the beginning and the end of the tunnel. They estimated the upper bound for the cost and the lower bound for the profit associated with tunnelling a block. By computing a minimal threshold based on statistics derived from the normal Burrows-Wheeler transform, they exclusively selected blocks with a score equal to or greater than this predefined threshold. In an environment devoid of collisions, this approach ensured that only blocks contributing to data compression were tunnelled. Despite its practical efficacy in minimizing resource requirements during encoding, the authors acknowledged that this estimation method may not be optimal in terms of compression.

In contrast to these approaches, a recent publication [8] has shifted focus towards addressing space complexity concerns when constructing large FM-indexes, particularly with respect to biological data, known for their high levels of repetition. The authors proposed a preprocessing algorithm termed *prefix-free parsing*, which operates on an input string S and efficiently generates two data structures called a dictionary D and parse P of S in a single pass. Leveraging this algorithm, the construction of the BWT of S from P and D requires a workspace proportional to their size and $O(|S|)$ time. Notably, in practical scenarios, especially when dealing with repetitive texts, the combined size of the parse and dictionary is substantially smaller than that of the original text S, allowing them to fit comfortably within internal memory even for sizable S. Consequently, this representation of the text facilitates quicker computation of the compressed version without necessitating frequent disk access.

Furthermore, the utility of the prefix-free parsing method extends to Wheeler graphs, as demonstrated by Baláž and Goga in 2022 [14]. Incorporating prefix-free parsing into the preprocessing algorithm for addressing the block choice problem accelerates the process, given the significantly reduced input size. Additionally, this approach offers the advantage of diminishing space complexity, which is particularly beneficial for handling collections of pangenomic sequences. Consequently, it enables the utilization of Wheeler graphs as pangenomic references for real-world pangenomic datasets. However, despite these advantages, this method did not significantly improve compression efficacy.

In his thesis [5], Uwe Baier introduced various tunnelling strategies, including the Hirsh strategy, the Greedy strategy without considering negative side effects, the Greedy strategy that takes negative side effects into account, and the tunnelling strategy utilizing de Bruijn graph edge minimization. The Hirsh strategy adopts a pragmatic

approach, focusing on tunnelling only profitable blocks, akin to the earlier 2018 work. The other greedy strategies build upon previous approaches with some enhancements. The last strategy is of the greatest interest.

The de Bruijn graph edge minimization strategy, initially proposed by Uwe Baier in 2020 [6], was specifically optimized for sequence analysis. This strategy addresses the de Bruijn graph edge minimization problem, which aims to find the order-k de Bruijn graph with the minimum edge count among all orders. The paper describes an efficient algorithm to solve this problem, establishing a direct connection between the edge minimization problem and the BWT tunnelling problem. Consequently, this strategy offers a means to minimize the length of a tunnelled BWT while preserving essential properties for sequence analysis. Notably, this state-of-the-art strategy operates in $O(n \log \sigma)$ time, where n represents the size of the graph and σ denotes the alphabet size. Given its superior time and compression capabilities, our approach will be compared solely to this strategy.

However, most of these approaches including the last one considered non-colliding blocks only, thereby limiting the potential compression. In contrast, our approach takes collisions into account and can generate sets of blocks containing compensable collisions among them, representing a significant advancement in solving the tunnelling problem.

Chapter 2

Description of the heuristics

In this chapter, we provide the theoretical foundation for our new approach to addressing the tunnelling problem. We explain the rationale behind our methodology and offer a high-level overview of the algorithm, with detailed explanations to follow in the next chapter.

As discussed previously, it is essential to ensure that the final set of blocks selected for tunnelling does not include critical collisions, as that would compromise the reversibility of the tunnelled Wheeler graph. To address this concern, the initial segment of this chapter analyses critical collisions and the time needed to find them aiming to find efficient heuristic solutions. In the next two sections, various approaches to resolving collisions between two critically colliding blocks are presented, accompanied by an assessment of their potential impact on the compression rate. Initially, the possible solutions are analysed theoretically, followed by validation through statistical analysis using both simulated and real-world data. The final part of this chapter presents an overview of the heuristics and illustrates its process on a complicated structure comprised of colliding blocks.

Since tunnelling compensable collisions does not affect the reversibility of the resulting Wheeler graph, we do not search for such collisions, i.e. the final set of blocks will be tunnelled without the additional information about compensable collisions within this set.

Before delving into the examination of the critical collisions, let us assume the set of maximal blocks B over a Wheeler graph G has been computed. Each block is represented as a triplet of numbers (w, s, e), where

- w denotes the width of the block,
- s signifies the first edge in the last column of the block,
- e denotes the last edge of the last column of the block.

In other words, s and e represent the start and end positions (in an inclusive range) of the block within the last column of the Burrows-Wheeler matrix. Such representation allows efficient storage and retrieval of the information about the block, enabling further analysis and processing of the Wheeler graph G with ease.

2.1 Critical collisions

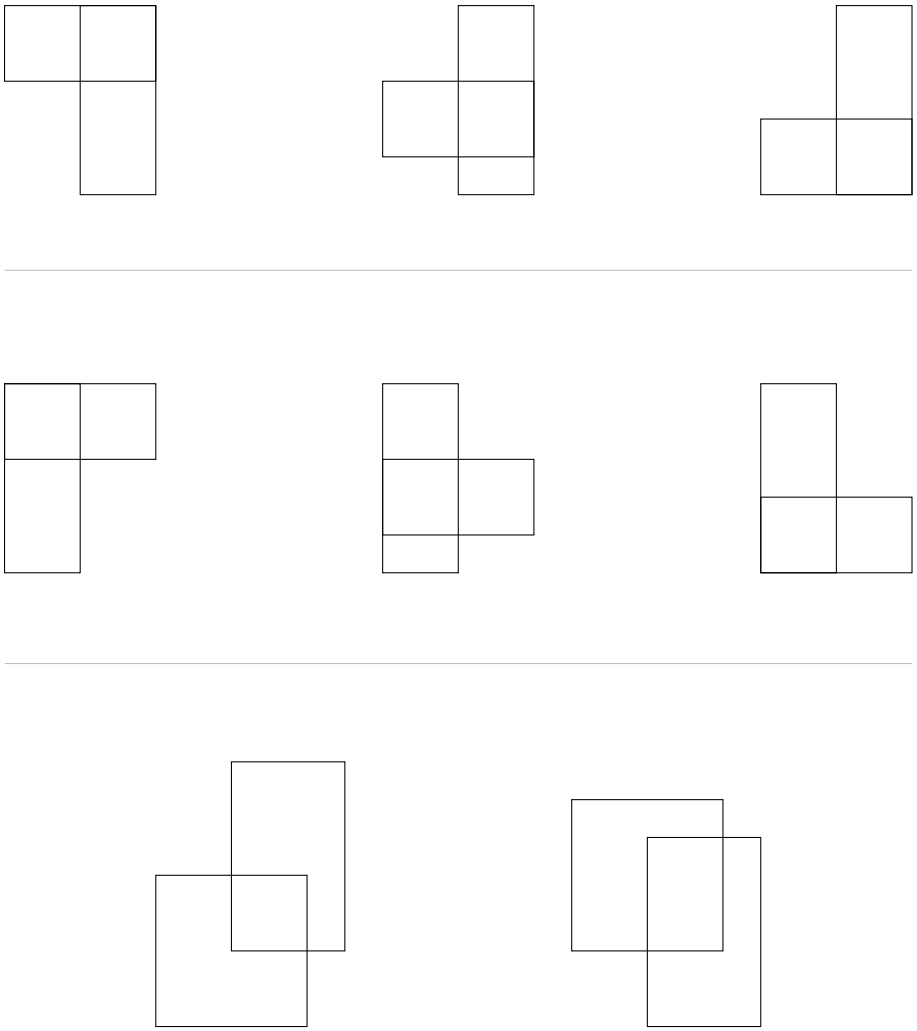


Figure 2.1: Possibilities of critical collisions for two blocks. The size and shape of blocks may differ. The grey division is based on the computation time of the collisions.

The full spectrum of critical collisions between two blocks is depicted in Figure 2.1. Given that the blocks are represented via their right-most columns, the computational complexity of determining the first three possibilities is $\mathcal{O}(N \log N + m)$, where N represents the number of maximal blocks in the precomputed set and m denotes the number of reported collisions. This is achieved by traversing the sorted sets of start and end positions, maintaining a memorized record of blocks overlapping the current

position (i.e. blocks whose end positions were not yet encountered but whose start positions were). Upon encountering a start position of a block, collisions between that block and others in the memorized set are reported, and the block is subsequently added to the set. Conversely, encountering an end position prompts the removal of the block from the set. The time complexity is the consequence of sorting the start and end positions and reporting all collisions. Notably, assessing the collision of this manner between two blocks is of constant time complexity, as it requires no more than comparing their start and end positions.

A similar strategy is applied to compute the next three categories of critical collisions after transposing the blocks to the left column of the BW matrix, given their alignment flexibility within the matrix. Mapping the blocks to the left-most column is done by applying the LF-mapping to the start position of the block a number of times equal to the block width, therefore it takes $\mathcal{O}(Nw_{max})$ time, where w_{max} represents the maximal width of the blocks. Consequently, the cumulative time complexity is $\mathcal{O}(N \log N + m + Nw_{max})$. Similarly to the previous case, the time complexity of identifying block collision in this manner remains constant when the blocks in question are mapped to the left column of the BW matrix.

However, the computational efficiency diminishes notably for the last two types of critical collisions, diverging from the straightforward assessment of the other collisions. Here, the computational time increases to $\Omega(Nw_{max} \log(Nw_{max}))$, due to the necessity for an examination across all block columns, rather than focusing on just one column as in the case with the other types of critical collisions. Furthermore, the determination of this type of collision between two blocks is not constant but depends on the width of the blocks. Each block column requires an inspection to accurately report the occurrence of the collisions since the blocks can overlap in many possible ways, where the number depends on their width.

Because of the variety of computation, we introduce terms for the three types of critical collision.

Definition 2.1.1 (Aligned and corner collision)**.** Let b_1 and b_2 be two critically-colliding blocks in the Wheeler graph G. We say these blocks create critical *right-aligned collision* (resp. critical *left-aligned collision*) if their right-most (resp. most left) columns overlap. If blocks b_1 and b_2 do not create an aligned collision, we say they create critical *corner collision*.

Figure 2.1 depicts in the first row right-aligned collisions, in the second row left-aligned collisions and in the last row corner collisions. The aim is to avoid computing the corner collisions, in order to achieve higher efficiency. The idea of our approach lies in examining the blocks involved in these collisions, as illustrated by grey stripes in Figure 2.2. From now on we refer to these blocks as the *inner blocks* of the cor-

ner collisions. Inner blocks inherently overlap with the initial blocks in terms of the aligned collisions. Therefore, resolving the right and left-aligned collisions in a manner that simultaneously addresses the corner collisions would lead to an algorithm free of additional computational overhead.

Notably, the maximal version of the inner blocks may be self-colliding or possess larger dimensions, with the higher block potentially being higher and the wider block being wider (and thus destroying aligned collision with the red block). To avoid complications, we assume the inner blocks can not be further extended. However, such assumptions will not undermine the fundamental concept of the heuristic approach, in contrast, addressing these speculations can be accomplished with relative ease, as will be demonstrated in the final section.

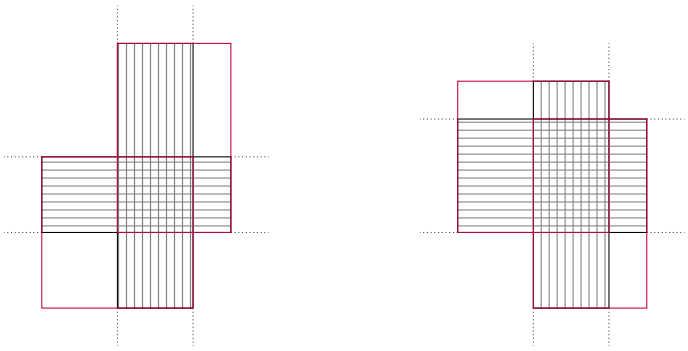


Figure 2.2: Structure of corner collisions. The corner collisions of the red blocks include two other blocks depicted with grey lines - the inner blocks. These blocks collide with both red blocks in a manner of aligned collisions. The grey blocks may extend beyond the indicated collision boundaries in a way the dotted lines imply.

In the next sections, we examine solutions addressing aligned collisions and evaluate their effect on corner collisions. Our objective is to find efficient strategies for resolving right-aligned and left-aligned collisions while minimizing the effect on the ultimate compression outcome, with a primary focus on the potential elimination of corner collisions.

2.1.1 Aligned collisions

In this section, three optimal strategies for addressing critical aligned collisions of two blocks are presented. The resulting profit of each strategy is compared with the profit gained by tunneling the entire collision, giving a lower bound of the optimal solution.

In the first chapter, we have shown the evaluation of the profit of a single block. Consider two height-maximal blocks in a Wheeler graph G, b_1 and b_2, with heights h_1 and h_2, and widths w_1 and w_2, respectively, where $h_1 > h_2$. Assuming these

blocks collide in an aligned manner, the profit derived from tunnelling such a collision is calculated as the sum of the profits of both blocks minus the intersecting positions, that were "tunnelled twice", i.e.:

$$\begin{aligned} & w_1 \cdot (h_1 - 1) \ + \ w_2 \cdot (h_2 - 1) \ - \ w_1 \cdot (h_2 - 1) \\ = \ & w_1 \cdot (h_1 - 1) + (h_2 - 1) \cdot (w_2 - w_1) \\ = \ & w_1 \cdot (h_1 - h_2) + w_2 \cdot (h_2 - 1). \end{aligned}$$

In the subsequent sections, we delve into how this profit fluctuates for right-aligned collisions based on different solution methodologies. Notably, the profit of critical aligned collisions remains the same for both left-aligned and right-aligned collisions, as the preserved number of the positions in the first rows within the tunnelled Wheeler graph G does not vary. Consequently, the following analysis primarily focuses on right-aligned collisions, given their symmetry with left-aligned collisions.

Handling critically aligned collisions can result in three possibly optimal solutions, depending on the dimensions (width and height) of the blocks. A collision can be divided into two non-overlapping blocks either vertically or horizontally, or a compensable collision can be achieved by properly shortening the higher block. Illustrations showcasing these approaches are provided in Figures 2.3, 2.4, and 2.5.

Vertical division

The *vertical division* of two blocks colliding in terms of aligned collision comprises preserving the taller block while eliminating all positions shared with the taller block from the shorter, wider block.

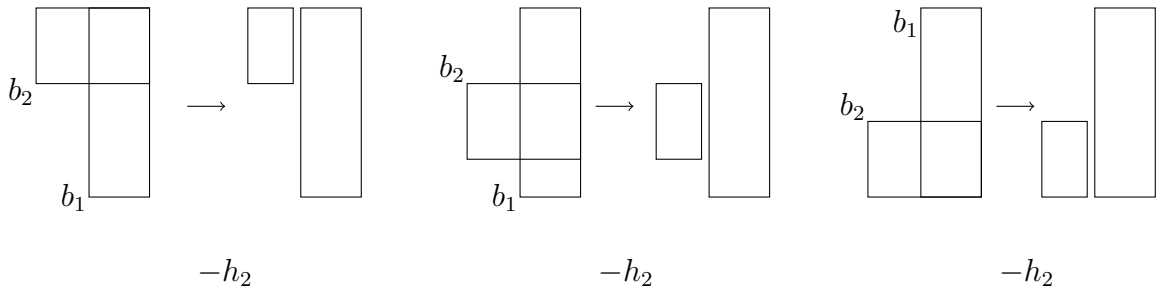


Figure 2.3: Vertical division depicted on all types of right-aligned critical collisions. The vertical division results in two non-colliding blocks. The change in profit made by the solutions is depicted below the solutions.

The profit derived from tunnelling vertically divided aligned collisions undergoes the same change for each type of right-aligned collision, as illustrated in Figure 2.3. As indicated, the right column of block b_2 remains in the tunnelled graph G. Consequently,

it is expected and indeed confirmed through computation, that the size of the tunnelled graph increases by the height h_2 of block b_2, compared to the size of the graph arising from tunnelling the whole collision.

Horizontal division

The *horizontal division* mirrors vertical division in principle but inverts the preservation process. Here, the wider block remains intact, while shared positions are removed from the taller block, potentially resulting in the taller block being fragmented into multiple blocks.

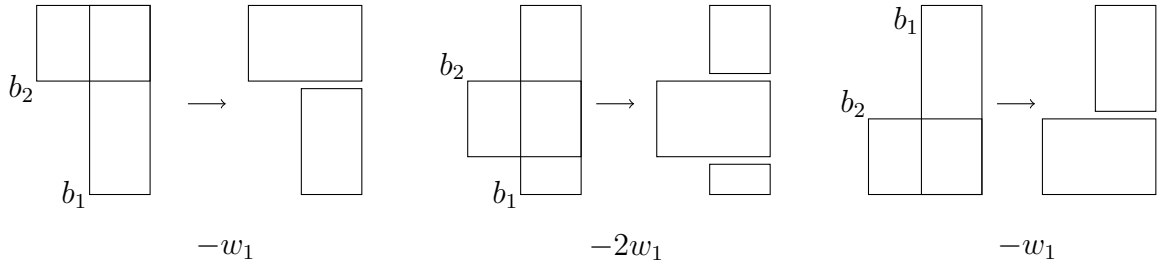


Figure 2.4: Horizontal division depicted on all types of right-aligned critical collisions. The horizontal division results in two or three non-colliding blocks. The change in profit made by the solutions is depicted below the solutions.

Unlike vertical division, the profit of this solution varies depending on the type of right-aligned (or left-aligned) collision. If the division yields two blocks, the profit decreases in the width of the taller block w_1. However, if the division generates three non-colliding blocks, the size of the tunnelled graph increases by $2w_1$.

Shortening

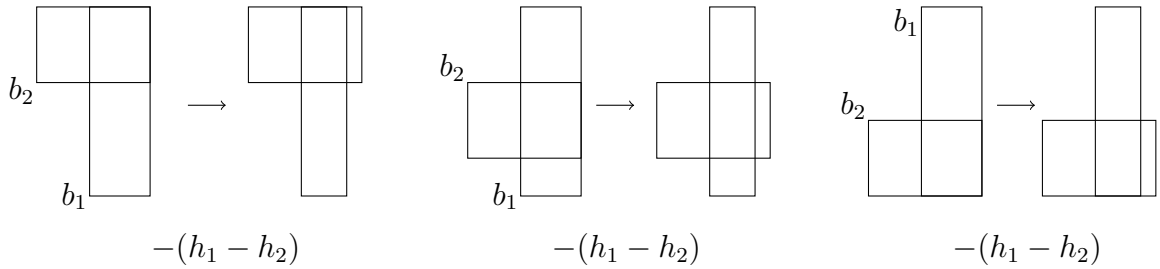


Figure 2.5: Shortening depicted on all types of right-aligned critical collisions. Shortening creates a compensable collision of blocks. The change in profit made by the solutions is depicted below the solutions.

The *shortening* involves the removal of one column from the taller block, either the right-most in right-aligned collisions or the left-most in left-aligned collisions. In contrast to divisions, shortening the taller block results in a compensable collision. Similar to vertical division, the profit derived from such a solution changes in the same way for each type of aligned collision. Notably, the size of the tunnelled graph increases by the difference between the heights of the blocks, $h_1 - h_2$.

Note that these solutions are optimal only when considering two colliding blocks. In scenarios involving larger colliding hierarchies, alternative solutions such as omitting an entire block may lead to an optimal set. This means our heuristics will approach the tunnelling problem with a limited perspective, yet this approach ultimately enhances the efficiency of the final algorithm.

2.1.2 Corner collisions

In this section, we show how corner collisions can be addressed through various combinations of solutions for aligned collisions. The outcome is contingent upon the selection of solution types for both right-aligned and left-aligned collisions, as well as the order in which they are addressed.

Notably, each combination necessarily involves at least one division. Attempting to resolve both right-aligned and left-aligned collisions solely through shortening would not solve the corner collisions, since the shortening has no effect on the blocks that create the corner collision.

Additionally, in order to resolve all corner collisions, consistent handling of either right-aligned or left-aligned collisions is required, either through vertical or horizontal division. Combining the approaches is counterproductive and would necessitate additional information about corner collisions, which we assume is unavailable and seek to avoid computing. The second type of aligned collisions may, in contrast, be addressed using a combination of solutions.

Let us consider two blocks that con the contraryreate corner collision $b1$ and b_2 with height h_1 and h_2, and widths w_1 and w_2, respectively. Furthermore, we denote h_3 the height of the shared area and w_3 the width of the shared area.

The possible combinations and their differences in the profit compared to tunnelling the whole collision are as follows:

- Horizontal division + horizontal division $\rightarrow (w_1 + w_2)$
- Horizontal division + vertical division $\rightarrow (w_1 + h_3)$ or $(w_2 + h_3)$ or $(h_1 + w_3)$ or $(h_2 + w_3)$
  .
- Horizontal division + shortening $\rightarrow (w_1 + h_2 - h_3)$ or $(w_2 + h_1 - h_3)$

- Vertical division + vertical division $\rightarrow (h_1 + h_2)$
- Vertical division + shortening $\rightarrow (2h_1 - h_3)$ or $(2h_2 - h_3)$

An example of the process when the combination of vertical and horizontal division is applied is depicted in Figure 2.6. The variability in the profit difference within the combinations arises from the various permutations of their application. It depends on which solution type is used for each aligned collision, and the order in which they are applied. These factors collectively contribute to the divergence in profitability considering one combination.

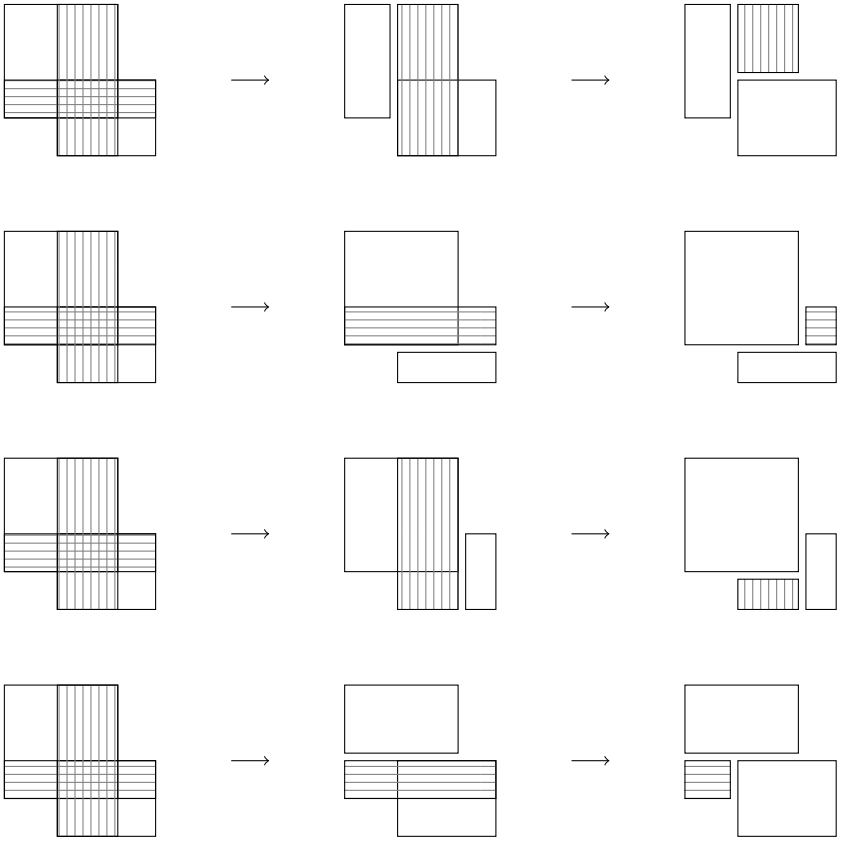


Figure 2.6: Solving corner collision with vertical and horizontal division. First picture: right-aligned collision solved first with vertical division, left-aligned collisions solved with horizontal division afterwards. Second picture: right-aligned collision solved first with horizontal division, left-aligned collisions solved with vertical division afterwards. Third picture: vertical division applied to left-aligned collisions first, then horizontal division used for left-aligned collisions. Fourth picture: left-aligned collision solved with horizontal division, and afterwards, right-aligned collision solved with vertical division.

Given the difference in profits, relying exclusively on vertical division for both aligned collisions is suboptimal, as an appropriate combination of vertical division and shortening can yield higher profits.

2.1.3 Statistics

In order to select the most suitable solution for aligned collisions (as corner collisions are addressed implicitly), we performed a comparative analysis of the solutions using both artificial and real data.

Initially, we measured the length of the tunnelled Wheeler graph without resolving any critical collisions. Such a graph has minimal size, however, is useless in practice, due to the tunnelling of critical collisions. Subsequently, all right-aligned collisions were solved using one of the solutions and the resulting increase in the size of the Wheeler graph was evaluated.

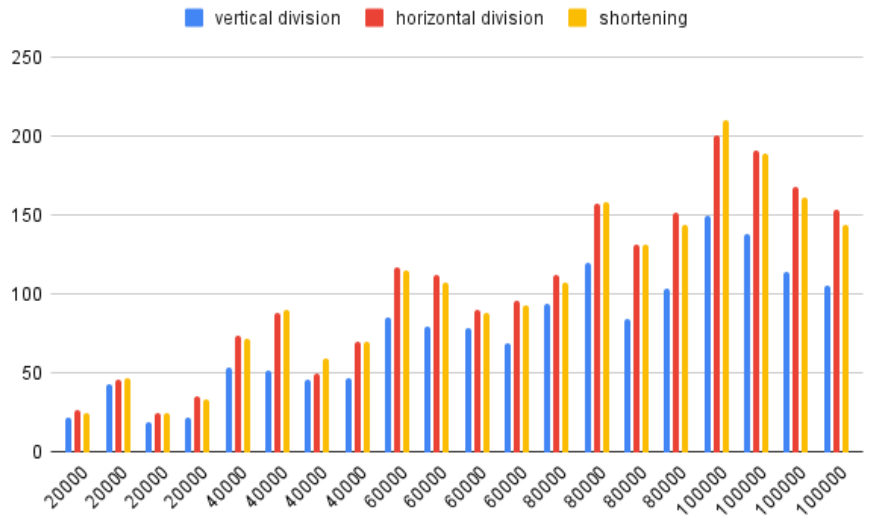


Figure 2.7: Comparision of vertical, and horizontal division and shortening on random data of various lengths. The x-axis depicts the length of the random sequences (with 4 samples for each value of length) and the y-axis shows the increase in length of the tunnelled Wheeler graph when solving the right-aligned collision with a specific solution.

For the first analysis, random sequences of varying lengths over the alphabet $\{a, c, g, t\}$ were generated. This analysis aimed to evaluate the general effectiveness of the solutions when applied to Wheeler graphs generated from sequences not coming from a specific distribution. The results are shown in Figure 2.7.

In the second analysis, we assessed the impact of the solutions on biological data consisting of a single sequence. Biological data inherently exhibit more repetition compared to random sequences, leading to slightly different outcomes compared to the previous analysis. The results are depicted in Figure 2.8.

Lastly, we compared the solutions using pan-genomic data. This analysis provides the most insightful results as this type of data is the primary focus of our research. The achievements of this analysis are shown in Figure 2.9.

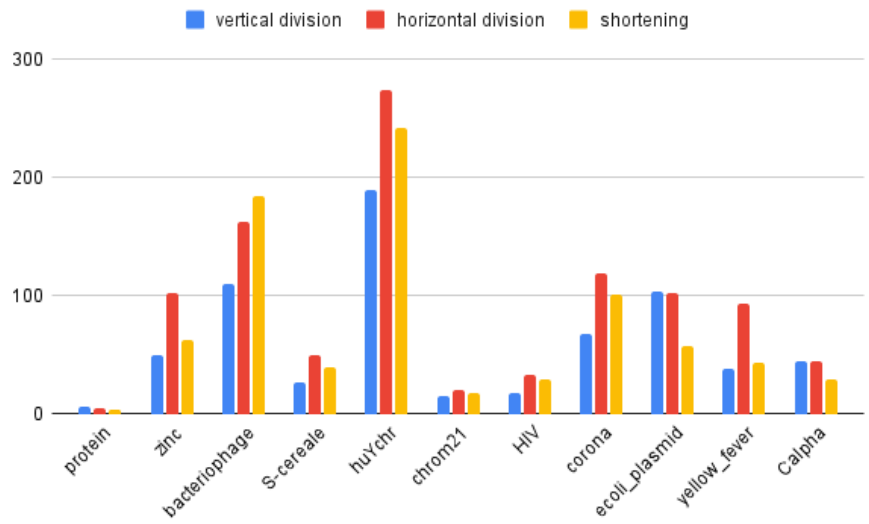


Figure 2.8: Comparision of vertical, horizontal division and shortening on biological data. The x-axis depicts the names of the data files and the y-axis shows the increase in length of the tunnelled Wheeler graph when solving the right-aligned collision with a specific solution.

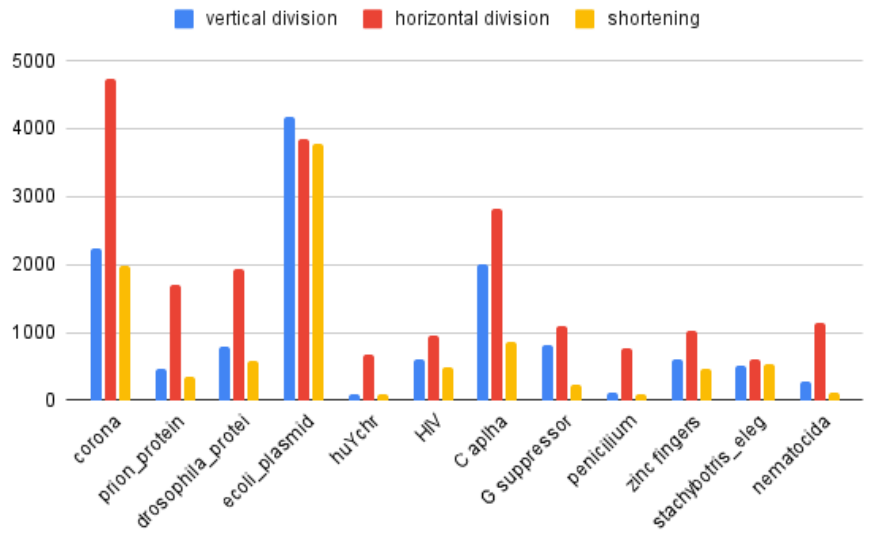


Figure 2.9: Comparision of vertical, horizontal division and shortening on pangenomic data. The x-axis depicts the names of the data files and the y-axis shows the increase in length of the tunnelled Wheeler graph when solving the right-aligned collision with a specific solution.

2.1.4 Selected combination

Given our focus on pangenomic sequences, the choice of combinations is affected by the latest statistics mainly. Analysis of the pangenomic dataset indicates that shortening appears as the most favourable solution type. To incorporate it into a combination,

the second type of solution must involve a division of any nature.

Upon observation, horizontal division generally exhibits lower success rates compared to vertical division. This holds for the pangenomic data as well as for any biological data or random sequences. The cause of this observation has been revealed by the theoretical analysis. The profit of the horizontal division fluctuates depending on the unique pattern of the collision, and its loss can potentially be twice as significant, unlike for the other types of solutions.

Ultimately, the chosen combination for the heuristic approach is vertical division paired with shortening. Indeed, the first two statistics suggest that the selected combination for the heuristics should yield favourable outcomes not only on pangenomic data but also on any text data.

While exploring the performance of the heuristics on non-biological data could provide valuable insights into its performance, we currently choose not to pursue such analysis. Nonetheless, we will compare its effectiveness on both biological non-pangenomic and pangenomic data in the last chapters to evaluate its versatility and applicability across different domains.

2.2 Heuristics overhead

The heuristics can be divided into four main steps. Initially, all maximal blocks, including self-colliding ones, are enumerated. Subsequently, they are sorted according to their start and end positions in the BWT, and right-aligned collisions are addressed through vertical division. Notably, this process may repeat, as one division can create a new right-aligned collision that necessitates resolution. By the end of this step, no two blocks overlap in any part of their right columns, and all critical corner collisions are thus effectively eliminated from the block set. In the third step, left-aligned collisions are handled by shortening the taller block. Similar to the previous step, this process must take into account the potential emergence of new left-aligned collisions.

The removal of self-colliding blocks from the set is also a crucial step in the process, and it must be performed at the appropriate time rather than at the beginning. To understand why timing is essential, we can reflect on the previous notion regarding the resolution of corner collisions. To effectively solve corner collisions, it is necessary to include the inner blocks within the collisions. However, these inner blocks may themselves be self-colliding. If all self-colliding blocks were removed from the set too early in the process, there would be no guarantee of the resolution of all corner collisions. Therefore, this step must be delayed. However, postponing the process too long could lead to worse compression results, as the self-colliding blocks could cause unnecessary reduction of other blocks. Therefore, all self-colliding blocks are removed from the set

between the second and third steps of the algorithm. Consequently, before handling left-aligned collisions, the set is free of corner collisions, and after the shortening, all critical collisions are entirely resolved within the block set, paving the way for the tunnelling process.

Although it is mentioned that all corner collisions are eliminated after the fourth step, this is not entirely accurate yet. It holds for corner collisions that overlap in at least two columns, considering each maximal block has a width of at least two. However, resolving corner collisions overlapping solely in one column presents a challenge.

To address this issue, before the second step of the algorithm, collisions in the first and last columns of the blocks must be computed, if any, and added to the block set as blocks of width one. This ensures that one-column overlaps are addressed as well. This computation is efficient, requiring $\mathcal{O}(n)$ time if the blocks are aligned to both the last and first columns of the BW matrix and are sorted. By merging intervals occupied by the blocks in both columns into non-overlapping intervals in both columns and comparing them with intervals from the other column, the one-column corner collision can be identified. Whenever two intervals overlap, the entire interval merged from the overlapping intervals is added to the block set as a block of width one. Thus, the one-column corner collisions can be solved by the heuristics, as the block of width one divides the colliding blocks into non-overlapping blocks in the second step. Afterwards, this block is removed from the set. Detailed explanation and pseudo-code of this process are explained in the next chapter.

Previously, we discussed the possibility that the inner blocks within a corner collision may have dimensions larger than what the collision indicates. However, these larger dimensions have no effect on the heuristics and its correctness. The increased height of the higher block has no impact on the process and the result remains similar. If the wider block is of the smallest width, it will be removed from the set of blocks in the second step, where right-aligned collisions are handled. If it is wider to the right, it will remain in the set, and in the third step, one of the blocks within the corner collision is shortened. If it is stretched to the left, the second step ensures that only the non-overlapping area of the block remains. If the wider block is prolonged in both ways, the block collides with each block within the collision in a manner of compensable collision, therefore it remains untouchable.

The entire heuristics process is illustrated in Figure 2.10. The first image depicts all maximal blocks within one fictive input, with blocks involved in corner collisions depicted with grey stripes and one-column blocks highlighted in red. The second image showcases the blocks after addressing all right-aligned collisions with vertical division. Notably, no corner collisions are present. The third image displays the set after shortening is applied to all left-aligned collisions. This block set contains only compensable collisions, thus enabling safe tunnelling.

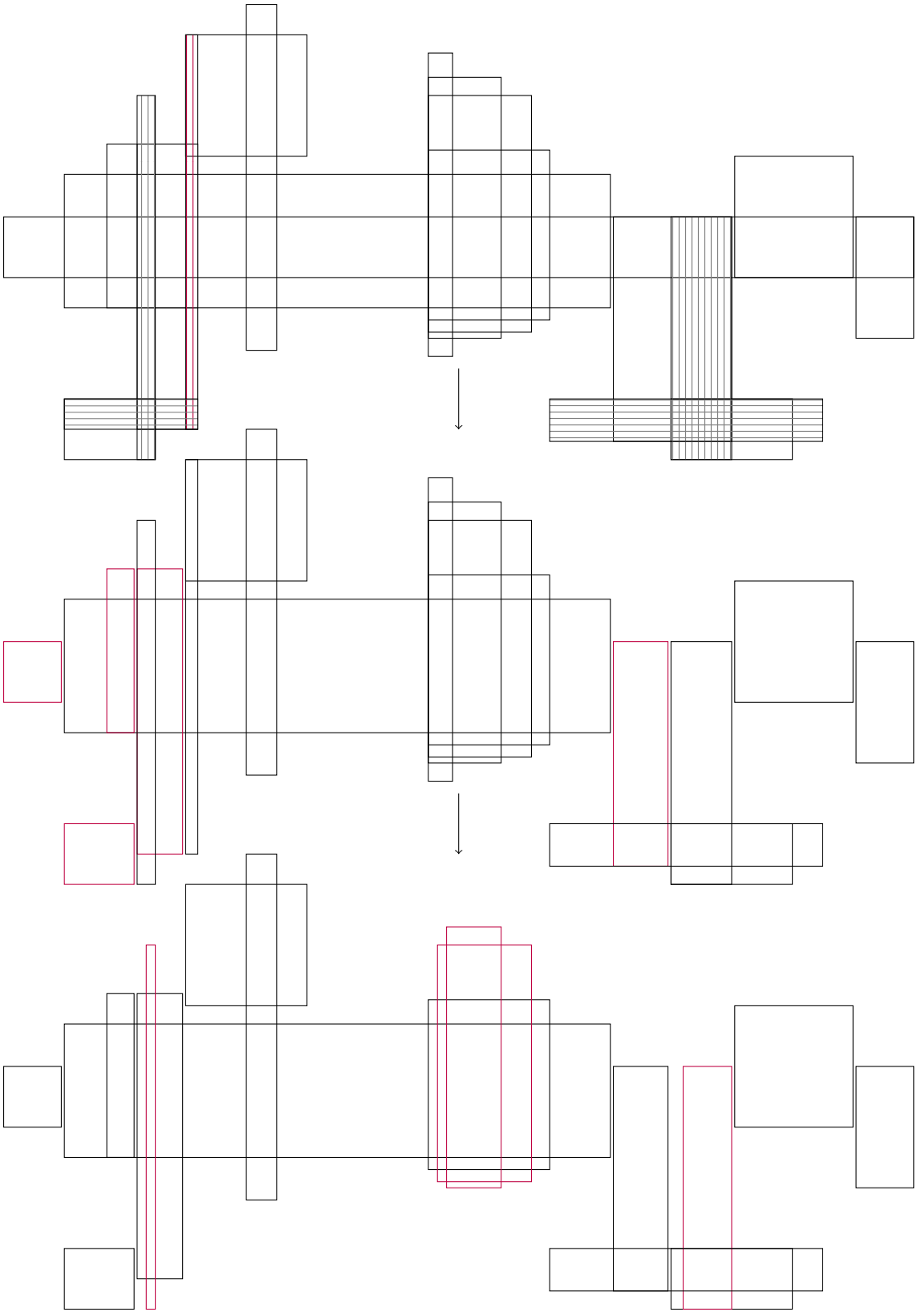


Figure 2.10: Process of the heuristics shown on a hierarchy of colliding blocks. The steps illustrate the process of the heuristic and its final output. Firstly, all maximal blocks are computed. The inner blocks within corner collisions are depicted with grey lines. Secondly, the one-width inner blocks in corner collisions are computed, depicted with red lines. After that, the right-aligned critical collisions are vertically divided. In the last step, the left-aligned collisions are handled in a manner of shortening. All changes in steps are depicted in red colour.

Chapter 3

Implementation

In this chapter, we delve into the details of the heuristics and provide pseudo-code for the main components of the heuristics. The code was originally implemented in Rust and can be found at `https://gitlab.com/wheeler_graphs/wglib`.

The chapter is divided into five main sections. The first part explains how the maximal blocks are enumerated. To enhance the understanding, we provide an overview of the calculation, however, we will not delve into deep details, as the process has been extensively covered in existing literature [5, 16, 10]. In addition to these blocks, the inner blocks of width one are necessary to be computed, to ensure all right-aligned collisions are addressed in the following part. Both of these parts run in linear time with the size of the graph.

The next section handles right-aligned collisions. Collisions of this type are solved using vertical division. After handling the right-aligned collisions, all self-colliding blocks and blocks of width one are removed from the set in this section. This computation requires $\mathcal{O}(N \log N w_{max})$, where N is the number of input blocks and w_{max} denotes the maximal width of the blocks.

The third part of this chapter focuses on resolving left-aligned collisions using shortening. All critical collisions are ultimately removed from the set of blocks, making the result set a feasible solution. The time complexity of the algorithm in this section is also $\mathcal{O}(N \log N w_{max})$.

In the fourth stage, the tunnelling is briefly explained. This topic has been covered by previous works [6, 2, 5], therefore the aim is to introduce just a simple overview, instead of diving deeply into the problematics. This part takes $\mathcal{O}(N w_{max} h_{max})$ time, where h_{max} is the maximal height of the final blocks. Since the tunnelling process is not part of the heuristics, this computation is not included in the overall time complexity of the heuristics.

The last section is devoted to the linear reconstruction of the tunnelled Wheeler graph, an essential part of the decompression. This part may illustrate the process of

traversing the tunnelled graph and extracting information from it. It also may shed light on the problem behind tunnelling two critically colliding blocks. Whilst many other useful queries can be efficiently performed on the tunnelled Wheeler graph, we will not display or implement them in this work, as they have been explored in previous research [11, 13].

3.1 Enumeration of the blocks

This section is dedicated to the enumeration of all maximal blocks, accompanied by one-width overlappings. The second computation necessitates the first one for an efficient implementation, thus we explain the enumerations in this order.

3.1.1 Maximal blocks

To efficiently enumerate all maximal blocks with a width and height of at least two, we employ the Longest Common Suffix (LCS) array [5, 15]. This array encapsulates the information about the longest common suffix of two adjacent rows within the BW matrix, hence inheriting its nomenclature. We cease from showing the computation of this array, since comprehensive discussions regarding its efficient implementation are available in existing literature [5]. Notably, our implementation includes these methodologies to ensure computational efficacy.

Due to the slightly modified definition of the blocks, a different version of the LCS array will be employed, where the values will take into account the first column of the BW matrix as well. An example of the original and modified version of the LCS array can be found in Figure 3.1. The computation of the alternative version varies from the original only slightly. When determining the value of the LCS array at a certain position, the value is changed according to the letters at the same positions in the F-column. If they are equal, the value is incremented, otherwise it is set to zero.

Using a stack-based approach, the process of enumerating all maximal blocks, as depicted in Algorithm 1, takes $\mathcal{O}(n)$ time, for a Wheeler graph of size n. This algorithm traverses the nodes of the Wheeler graph in the Wheeler order while checking the LCS array. A possible start of a block of a certain width (which is given by the LCS array) is stacked when a wider part of the LCS array is encountered. Conversely, should the potential block width exceed the current position in the LCS array, the block has to end here. If the potential block satisfies minimal size properties and is right maximal, it can be reported. The right maximality of the block is proved using inverse LF-mapping. If the positions to the right of the right-most column do not form a continuous part in the Wheeler order, with all incoming edges of the nodes labelled equally, then the block is right-maximal and can be reported. Otherwise, it is omitted, since it can be

$readysteadyro\$ \longrightarrow$

F	L	LCS	$\widetilde{LCS}$
$\$\ r\ e\ a\ d\ y\ s\ t\ e\ a\ d\ y\ r\ o$		0	0
$a\ d\ y\ r\ o\ \$\ r\ e\ a\ d\ y\ s\ t\ e$		0	0
$a\ d\ y\ s\ t\ e\ a\ d\ y\ r\ o\ \$\ r\ e$		1	2
$d\ y\ r\ o\ \$\ r\ e\ a\ d\ y\ s\ t\ e\ a$		0	0
$d\ y\ s\ t\ e\ a\ d\ y\ r\ o\ \$\ r\ e\ a$		2	3
$e\ a\ d\ y\ r\ o\ \$\ r\ e\ a\ d\ y\ s\ t$		0	0
$e\ a\ d\ y\ s\ t\ e\ a\ d\ y\ r\ o\ \$\ r$		0	1
$o\ \$\ r\ e\ a\ d\ y\ s\ t\ e\ a\ d\ y\ r$		1	0
$r\ e\ a\ d\ y\ s\ t\ e\ a\ d\ y\ r\ o\ \$$		0	0
$r\ o\ \$\ r\ e\ a\ d\ y\ s\ t\ e\ a\ d\ y$		0	1
$s\ t\ e\ a\ d\ y\ r\ o\ \$\ r\ e\ a\ d\ y$		4	0
$t\ e\ a\ d\ y\ r\ o\ \$\ r\ e\ a\ d\ y\ s$		0	0
$y\ r\ o\ \$\ r\ e\ a\ d\ y\ s\ t\ e\ a\ d$		0	0
$y\ s\ t\ e\ a\ d\ y\ r\ o\ \$\ r\ e\ a\ d$		3	4

Figure 3.1: Original LCS-array and the modified version $\widetilde{LCS}$ shown on a wheeler matrix for string $S = readysteadygo\$$. The modified version is one unit larger than the original when the rows are equivalent in the F column. If not, it is equivalent to zero.

extended by at least one column to the right.

3.1.2 One-column overlappings

Once the set of all maximal blocks has been computed, we advance to identify one-column corner collisions. These collisions indicate an overlap between the first column of one block and the last column of another block. In order to efficiently identify and report such overlaps, we establish two sets of intervals.

The first set represents intervals in the last column of the Burrows-Wheeler matrix occupied by the blocks while the second set determines the occupied intervals in the first column of the BW matrix. The construction of these sets involves similar, yet slightly different methodologies.

The first set utilizes the precomputed blocks without additional information about their widths. Conversely, the second set is generated by mapping each block to the first column of the BW matrix. This mapping is achieved by applying the LF-mapping to the start position of the block a number of times equivalent to the block width, determining its position in the first column. The end position of the block is computed using the height of the block.

Once we obtain these sets of intervals, we advance to sort them prioritly according to their start positions, incrementally and secondarily in a decremental order according to their end positions. After that, the intervals are merged by traversing the sets separately and comparing adjacent intervals. During this process, we track the lowest

Algorithm 1: Algorithm enumerating all maximal blocks as triplets of width, start position and end position in the bwt.

Data: LCS array LCS of size n

Result: All maximal blocks in the form (w, s, e), where w is the width of the block and s, e are the start and end position of the interval of the right-most column.

```
1  begin
2      initialize an empty stack s
3      push (1, 0) on s
4      for i ⟵ 1 to n do
5          (b, w) ← top of stack s
6          while w > LCS[i] do                    // end of block of width w
7              pop topmost element of s
8
9              block ⟵ (w, b, i − 1)
10             if block is right-maximal and w > 1 and i − b > 1 then
11                 report block
12
13             (b′, w′) ⟵ top of stack s
14             if LCS[i] > 1 and LCS[i] < w′ then
15                 push (b, LCS[i]) on s   // assurance of the height-maximality
16
17             (b, w) ⟵ top of stack s
18         if w < LCS[i] then          // possible start of a block of width w
19             push {i − 1, LCS[i]} on s
20
               // make sure to report all possible block saved in the stack
21     while s is not empty do
22         (b, w) ⟵ top of stack s
23         block ⟵ (w, b, n − 1)
24         if block is right-maximal and w > 1 and n − b > 1 then
25             report block
26         pop topmost element of s
```

start and highest end positions covering two adjacent overlapping intervals. When encountering a non-overlapping adjacent pair, a new interval is pushed to the result set using the remembered start and end positions. After that, these positions are updated according to the newly encountered interval.

After merging the interval sets, they are compared in order to identify any overlapping intervals. Such overlaps indicate the presence of blocks creating corner collisions with one-column overlapping. These overlappings are computed by traversing both interval sets, and comparing potentially overlapping intervals. If one interval starts after the end of the other, the index in the set of the foregoing interval is increased. However, if the intervals overlap, we create a new block of width one with its start position set to the minimum of the start positions of the intervals and its end position set to the maximum of the end positions. This newly formed block is then added to the set of blocks, and the index in the set of the interval that started earlier is incremented. As the traversal holds only once for both sets, the algorithm runs in linear time, assuming the mapping of the blocks has been precomputed. If not, the time complexity is $\mathcal{O}(Nw_{max})$, where N is the number of maximal blocks and w_{max} denotes the maximal width of the maximal blocks.

3.2 Right-aligned collisions

In this section, we address two issues within the heuristics, in order to achieve a corner collisions-free block set. The first challenge covers handling right-aligned collisions in the manner of vertical division, and the second part focuses on eliminating self-colliding blocks. These computations are independent and are thus split into separate subsections for clarity and coherence.

3.2.1 Vertical division

It becomes evident that resolving one right-aligned collision may introduce another right-aligned collision, leading to a cyclical process. Such a scenario is illustrated by the widest block in Figure 2.10. Furthermore, in instances where multiple right-aligned collisions exist, as depicted by the left-aligned collisions in the middle of Figure 2.10, an inefficient algorithm would proceed in several cycles to resolve them. This inefficiency arises from the inappropriate order of addressing the right-aligned collisions. For instance, prematurely dividing the collision of the highest and the second-highest block would result in the third-highest block being initially compared with the highest block and then, in a subsequent cycle, with the block created from the division, even though the same result could be achieved by one comparison of the third and second highest blocks in the beginning. As the hierarchy of collisions grows, so does the

inefficiency of such an algorithm.

To circumvent this inefficiency, we introduce the concept of a *block tree*. The block tree is based on the observation that height-maximal blocks form nested intervals in both the last and first columns of the BW matrix.

Theorem 1 (Nested intervals of height-maximal blocks)**.** Let B be the set of height-maximal blocks in the Wheeler graph G. Then the start and end positions of the blocks in the last column of the Burrows-Wheeler matrix form nested intervals.

To prove the theorem, let us assume all blocks are maximal in height but there exists a non-nested right-aligned collision. If the collision is not nested, then the thinner block can be extended to height and, therefore, is not height-maximal. This observation forms the basis of our approach however, before delving into its practical implications, we first present the underlying idea.

Definition 3.2.1 (Block tree)**.** Let B be the set of height-maximal blocks. Let T be a tree, where each node belongs to a block except for the root, any block node in its last-column interval obtains the intervals of its children, and the sum of the node depth (length of the shortest path to the root) is maximized. We call such tree a *block tree*.

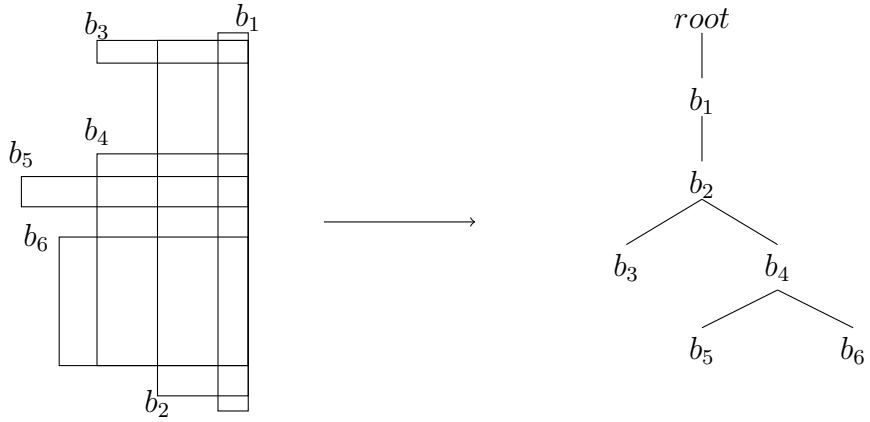


Figure 3.2: Block tree of a group of blocks. Left: group of height-maximal blocks colliding in an aligned manner. Right: block tree od this structure. Each child is entirely overlaped by its parent, in the sense of their right-most columns.

An example of a block tree for a group of aligned blocks is depicted in Figure 3.2. To ensure the blocks are still height-maximal after one round of the vertical divisions, it is necessary to merge aligned colliding blocks of the same width. The importance of merging aligned blocks of the same width to ensure height maximality is shown in Figure 3.3.

The introduction of the block tree offers an efficient solution to resolving right-aligned collisions. To minimize the number of cycles required by the process, it is

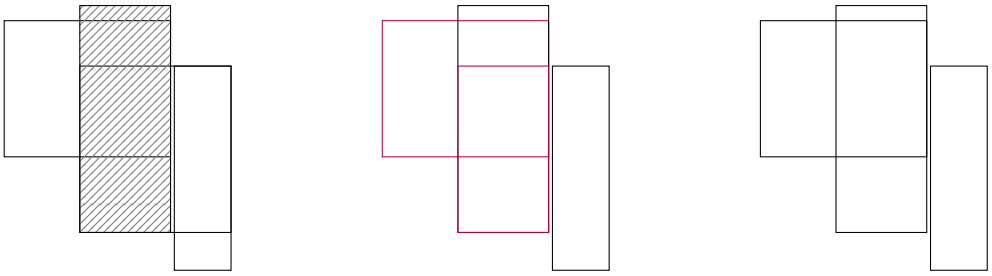


Figure 3.3: Reason of the merging for height-maximality insurance. In the first picture, all blocks are height maximal. The grey block indicates the inner block of the corner collision. The second picture shows the blocks after one round of vertical division. Red blocks do not form nested intervals, and the thinner block is not height-maximal. After merging aligned blocks of the same width, as shown in the last picture, all blocks are height-maximal and form nested intervals.

necessary to select the proper set of aligned collisions to be addressed. Specifically, if a shorter block collides with multiple higher blocks, it is most efficient to remove the positions shared with the widest higher block it overlaps. Deviating from this approach would result in unnecessary repetition of the algorithm. Hence, it is crucial to compute the largest area that needs to be removed from the shorter block to resolve the right-aligned collisions effectively. This information can be evaluated for each block by utilizing the block tree, as it corresponds to the width of the parent block, with the root block having zero width.

Another convenient feature of the block tree is its straightforward implementation. By sorting the blocks in increasing order of start positions and decreasing order of end positions, with priority given to start positions in the ordering, traversing such a sorted set aligns with traversing the block tree depth-first. Consequently, the algorithm can traverse the sorted set while keeping track of the ancestors. For each block, the width of its parent is chosen to create a thinner block. Subsequently, all blocks with a non-root parent are removed from the set, and the new blocks formed by their vertical division are added to a temporary set. After the traversal of the block tree, the temporary set of blocks is sorted and merged with the leftover set. This process is iterated until right-aligned collisions are no longer present. The number of iterations is bounded by the maximal width of blocks w_{max}. Therefore the overall time complexity of the vertical division algorithm is $\mathcal{O}(w_{max} N \log N)$, for N blocks. The number of repetitions of the algorithm is bounded by w_{max}, alternation of the blocks can happen at most $N w_{max}$ times and in each repetition, the blocks are sorted, which indicates the overall complexity. The pseudo-code for the algorithm is depicted in Algorithm 2.

Algorithm 2: Algorithm solving right-aligned collisions in the block set by vertical division.

Data: Set of blocks B as triplets (width, start position, end position), all mapped to the last column of the Wheeler matrix

Result: Set of blocks reduced from right-aligned collisions.

```
1  begin
2      do
3          changed ⟵ False
4          initialize an empty vector tmp
5          initialize an empty stack s
6
7          foreach b ∈ B do
8              t ⟵ top of stack s
                                  // update the path to the root
9              while s is not empty and t.end < b.start do
10                 pop topmost element of s
11                 t ⟵ top of stack s
12             if s is not empty then  // if block has non-root parent change it
13                 changed ⟵ True
14                 t ⟵ top of stack s
15                 new_start ⟵ b.start
16                 for i ⟵ 0 to t.width do                    // divide the block
17                     new_start ⟵ LF[new_start]
18                 push (b.width − t.width, new_start, new_start + (b.end − b.start)))
                    on top of tmp
19                 remove b from the B
20             push b on top of s
21         sort tmp
22         B ⟵ merge(tmp, B)
23     while changed
24     return B
```

3.2.2 Self-colliding blocks

After resolving the right-aligned collisions, the blocks of width one and self-colliding blocks are no longer important, thus their removal from the set is in place. While these blocks could be removed at the end of the process, reducing them as soon as possible is beneficial in two ways. Firstly, the amount of input data for the next computations is reduced, and so is the computation time. Secondly, retaining such blocks in the set potentially compromises the compression results, as some may lead to an unreasonable shortening of valuable blocks.

Identifying self-colliding blocks is straightforward: a block is self-colliding if and only if there exist nodes in its first row, that are closer in the Wheeler order than the height of the block. Given that the columns of the block form intervals of length equal to the height of the block, the absence of such a pair indicates that the block cannot overlap with itself. Thus, to select self-colliding blocks from the set, we sort the positions in their first row, obtained using the LF-mapping, and compare adjacent pairs. This operation runs in $\mathcal{O}(Nw_{max} \log w_{max})$ time, where N indicates the number of blocks in the set and w_{max} denotes the maximal width of the blocks in the set.

3.3 Left-aligned collisions

The left-aligned collisions are solved by employing the shortening technique, wherein the higher block is shortened by one column. Firstly, all blocks resulting from the previous reduction are mapped to the first column of the BW matrix, so any collisions of this type are easily detected. After that, the process closely resembles the one solving the right-aligned collisions, with a slight difference.

In this scenario, we also use the block tree to determine the maximum number of columns to be removed from the blocks, thereby minimizing the number of algorithm repetitions. In contrast to the right-aligned collisions, this number corresponds to the height of the node in the block tree, which is the length of the longest path to any of its descendants. Consequently, the height of the node represents the maximal minimum of columns to be removed from the block during the algorithm. Computing this maximum is particularly crucial for algorithmic efficiency, as it enables the removal of columns in a single step rather than multiple steps. In contrast with traversing the block tree for right-aligned collisions, here the removal values are set after searching the whole subtree of the node, not when the node is first encountered.

When the block is to be shortened, it is removed from the original set, and its altered form is added to a temporary set. This transformation is achieved using inverse LF-mapping, as the block is intended to be mapped to the first column of the BW matrix. Following the traversal of the block tree, the temporary set is sorted and merged with

the leftover set, akin to the previous process. This iterative process continues until no left-aligned collisions (and consequently, no critical collisions) remain. Similarly to the right-aligned collisions, the time complexity of handling the left-aligned collisions is $\mathcal{O}(w_{max}N \log N)$. The detailed algorithmic steps are depicted in Algorithm 3.

3.4 Tunneling

After obtaining the final set of blocks earmarked for tunnelling by the heuristics, we proceed to the final phase - compression. This process is held in two stages.

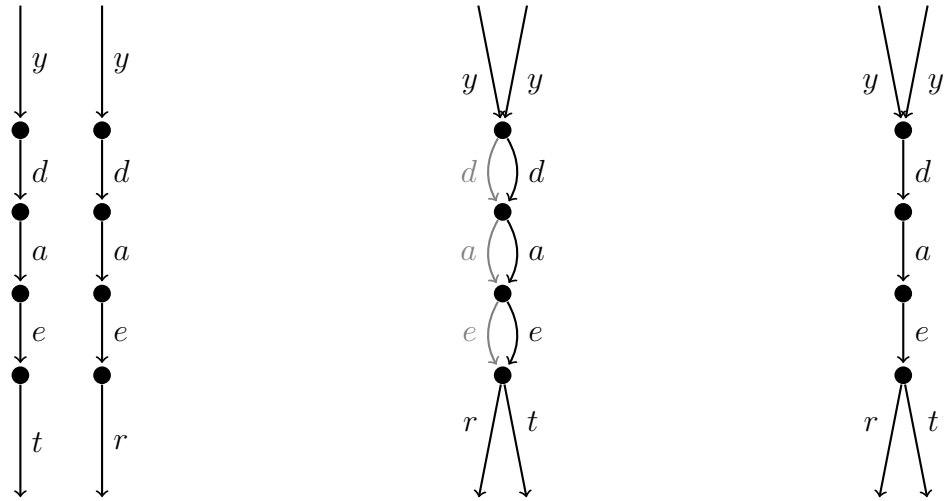


Figure 3.4: The process of tunneling in a Wheeler graph. The leftmost picture shows part of the initial Wheeler graph. In the middle, a multi-graph is depicted after the fusion of inner nodes. All redundant edges are coloured grey. On the right, after the removal of redundant paths, a tunnelled part of the Wheeler graph is shown.

Firstly, all positions in the blocks are marked depending on their positioning in the blocks. This represents fusing all inner nodes of the blocks, thereby yielding a multigraph. Secondly, all marked positions (i.e., positions that are neither the start nor the end of a tunnel) are removed. This step means eliminating redundant edges between inner nodes, resulting in a graph free from multi-edges. The two-stage process of tunnelling is depicted in Figure 3.4.

It is worth noting that tunnelling the blocks sequentially might end up in complex computation. The positions of the blocks in the tunnelled Wheeler graph may change as the graph shortens, hereby the sequential algorithm would necessitate remapping the blocks after each tunnelling. Therefore, we opt to tunnel the blocks in parallel. This entails executing the first stage for all blocks simultaneously, followed by the

Algorithm 3: Algorithm solving left-aligned collisions in the block set by shortening method.

Data: Set of blocks B in the form of triplets, (width, start position, end position), mapped to the first column of the Wheeler matrix

Result: Set of blocks reduced from left-aligned collisions

```
1  begin
2      do
3          changed ⟵ False
4          initialize an empty vector tmp
5          initialize an empty stack s
6          initialize a vector of zeros shorten of length N
7          i ⟵ 0
8          while i < B.length do
9              i ⟵ i + 1
10             while B[i].start < B[i − 1].end do // check a path to descendant
11                 push i on stack s
12                 i ⟵ i + 1
13             depth ⟵ 0
14             t ⟵ top of stack s
                // go from bottom to top in the path and update the values of
               shortening
15             while B[i].start > B[t].end do
16                 shorten[t] ⟵ max(depth, shorten[t])
17                 depth ⟵ depth + 1
18                 pop topmost element of s
19                 t ⟵ top of stack s
20             for k ⟵ s.length to 0 do
21                 shorten[k] ⟵ max(depth, shorten[k])
22                 depth ⟵ depth + 1
23         for j ⟵ 0 to B.length do                    // update all blocks
24             if shorten[j] > 0 then
25                 changed ⟵ True
26                 new_block ⟵ B[j] shortened of shorten[j] columns
27                 push new_block on top of tmp
28                 remove B[j] from B
29         sort tmp
30         B ⟵ merge(tmp, B)
31     while changed
32     return B
```

eliminating phase. This parallel approach ensures a more streamlined and efficient compression process.

3.4.1 Compression of nodes

For the tunnel marking process, we utilize two bit-vectors, d_{in} and d_{out}, each of length $n + 1$, where n represents the length of the original Wheeler graph G. Initially, these vectors are comprised of positive bits only.

To mark a tunnel of a block, we traverse from its last column to its first using LF-mapping. During this process, all positions in each column, are set to negative bits in the bit vectors. However, the rightmost (last) column is exclusively marked in the bit vector d_{in}, while the leftmost (first) column is marked solely in the bit vector d_{out}. This inconsistency in the marking of the bit vectors indicates the beginning or end of a tunnel, depending on the context.

In essence, these arrays serve to denote the indegrees and outdegrees of the nodes in a Wheeler graph. Initially, all nodes possess an indegree and outdegree of one. When a block is marked for tunnelling, the inner column nodes are merged into one node. Consequently, the indegrees of all nodes, except those in the leftmost column, and the outdegrees of all nodes, except those in the rightmost column, become equal to the height of the block. This fusion results in a multigraph prepared for the final compression, where redundant paths with a common start and end node are eliminated. [5], see the first part of the Algorithm 4.

3.4.2 Compression of edges

Once the tunnels have been marked in the bit-vectors d_{in} and d_{out}, the construction of the tunnelled BWT can proceed. It is crucial to note that if any critical collision were present the marking stage would yield contradictory markings in the bit-vectors, leading the process to abort. Therefore, ensuring the absence of critical collisions is essential before proceeding with the tunnelling process.

To compress the Wheeler graph G and derive the tunnelled Wheeler graph $\tilde{G}$, it is necessary to reduce the arrays d_{in} and d_{out} of the redundant edges and retain only essential positions in $\tilde{G}$. In essence, the outdegrees of all nodes except the last one, and the indegrees of all nodes except the first one, must be reduced to one. This reduction is achieved through a traversal of both arrays, whereby d_{in}, d_{out}, and G are trimmed accordingly.

The positions marked zero in d_{out} correspond to the entries to be removed from d_{in}. Similarly, the positions marked zero in d_{in} correspond to the entries to be removed from d_{out}, as well as from G. Therefore, by scanning the marked bit-vectors, the desired tunnelled Wheeler graph can be obtained by determining whether to retain an entry

Algorithm 4: Computation of tunneled Wheeler graph $\tilde{G}$. First, the to-be-tunneled positions are marked in the bit-vectors d_{in} and d_{out}. After that they are removed from the bit-vectors d_{in}, d_{out} and $\tilde{G}$.

Data: Wheeler graph G of size n, set of to be tunneled blocks B and LF-mapping LF

Result: tunneled Wheeler graph $\tilde{G}$ and corresponding bit-vectors d_{in} and d_{out}

```
1  begin
2      initialize vectors d_in and d_out of size n + 1 with ones
           // mark all positions for tunneling in the bit-vectors d_in and d_out
3      foreach (w, s, e) ∈ B do
4          s' ⟵ s
5          h ⟵ e − s
6          for i ⟵ 0 to w − 1 do
7              for j ⟵ 1 to h − 1 do
8                  d_in[s' + j] ⟵ 0
9              s' ⟵ LF[s']
10             for j ⟵ 1 to h − 1 do
11                 d_out[s' + j] ⟵ 0
12
                   // remove all the positions according the markings in the
       bit-vectors
13     initialize empty vectors G̃, new_d_in, new_d_out
14     for i ⟵ 0 to n do
15         if d_in = 1 then
16             push G[i] on top of G̃
17             push d_out[i] on top of new_d_out
18         if d_out = 1 then
19             push d_in[i] on top of new_d_in
20     push 1 on top of new_d_in
21     push 1 on top of new_d_out
22     return G̃, new_d_in, new_d_out
```

i	$G[i]$	d_{out} d_{in}	$F[i]$
0	o	$1 \longleftarrow 1$	\$
1	e	$1 \longleftarrow 1$	a
2	e	$1 \longleftarrow 1$	a
3	a	$1 \longleftarrow 1$	d
4	a	$1 \longleftarrow 1$	d
5	t	$1 \longleftarrow 1$	e
6	r	$1 \longleftarrow 1$	e
7	y	$1 \longleftarrow 1$	g
8	g	$1 \longleftarrow 1$	o
9	\$	$1 \longleftarrow 1$	r
10	y	$1 \longleftarrow 1$	s
11	s	$1 \longleftarrow 1$	t
12	d	$1 \longleftarrow 1$	y
13	d	$1 \longleftarrow 1$	y
14		$1 \longleftarrow 1$	

$\tilde{G}[i]$	d_{out} d_{in}	$F[i]$
o	$1 \longleftarrow 1$	\$
e	$1 \longleftarrow 1$	a
a	$1 \longleftarrow 1$	d
t	$1 \longleftarrow 1$	e
r	0	
y	$1 \longleftarrow 1$	g
g	$1 \longleftarrow 1$	o
\$	$1 \longleftarrow 1$	r
y	$1 \longleftarrow 1$	s
s	$1 \longleftarrow 1$	t
d	$1 \longleftarrow 1$	y
	0	y
	$1 \longleftarrow 1$	

Figure 3.5: Inverse walk in the tunnelled Wheeler graph $\tilde{G}$ of string *readysteadygo*\$. Above, Wheeler graph G, its sorted form F and bit-vectors d_{in} and d_{out} are captured before and after tunneling block $(4, 5, 7)$. Starting from a position of \$, using LF-mapping, the inverse walk leads to building the original string. When encountering the end of the tunnel (marked in the d_{in} vector), an offset is saved and then used when the beginning of the tunnel (marked in the array d_{out}) is reached.

based on the zero markings in both arrays [5], see also the second part of the Algorithm 4.

3.5 Reconstruction

In this section, we elucidate the process of reconstructing the original string Wheeler graph from its tunnelled version. The process closely resembles the one described in Baier's thesis [5], though with a slight modification, due to possible tunnelling of compensable collisions.

To rebuild the reverse of the primary string of length n in $O(n)$ time, we employ the `backward-step` function. This function navigates through the tunnelled Wheeler graph, moving one step backwards to visit the previous edge (previous letter in the context of the strings) of the initial graph.

The process of the reverse walk in a tunnelled WG closely resembles that in an

unmodified Wheeler graph, with one notable difference: we need to remember the offsets to the uppermost row when entering a tunnel, ensuring that when exiting that tunnel, we jump to the correct position in the WG.

In scenarios where no collisions are permitted, only one additional number would be required to remember - the current offset. However, when overlapping blocks are considered, it is possible to enter another tunnel before leaving the first, necessitating the storage of a potentially large number of offset values. Consequently, when exiting a tunnel, we must appropriately select from all saved offsets. This would rise to a complex problem if critical collisions were considered, whereas dealing with compensable collisions is straightforward.

By definition, two compensably colliding blocks form a cross, resulting in the starts and ends of the tunnels forming a well-parenthesized expression. When encountering a tunnel end, it logically belongs to the tunnel most recently entered and not yet exited. Therefore, a stack-based approach proves sufficient and easily implemented. This approach ensures efficient management of offset values, facilitating the accurate reconstruction of the original Wheeler graph from the tunnelled graph.

Using the `backward-step` function as outlined in Algorithm 5, we can construct the reverse of the original graph as follows. Let n be the length of the original graph G, $\tilde{G}$ be its tunnelled version, s represent an empty set and i denote the position in G where the character \$ occurs. Then, by executing the following commands repeatedly for n times,

1. Output $\tilde{G}[i]$
2. $i, s \longleftarrow$ `backward-step`(i, s)

we generate G in reverse order.

Figure 3.5 illustrates the initial Wheeler graph G, arrays d_{in} and d_{out} for the string *readysteadygo*\$, and the final tunnelled version of each. The arrows between the bit arrays signify the reverse walk through the tunnelled BWT, implicitly demonstrating the functionality of Algorithm 5.

Algorithm 5: Backward-step function for computing the reversed original string from a tunneled BWT. Similar algorithm was published in [5].

Data: Succinct representation of a tunneled BWT as computed in algorithm 4 D_{IN}, D_{OUT} and L', index i of na edge in D_{OUT} and stack s with tunnel offsets

Result: Index i of the next edge in D_{OUT} and stack s with updated tunnel offsets

1 **Function** `backward-step`(i, s)`:`

2 $i \longleftarrow \mathtt{C}_{L'}[L'[i]] + \mathtt{rank}_{L'}(L'[i], i)$ `// follow` i`-th edge`

3 $nr \longleftarrow \mathtt{rank}_{D_{IN}}(1, i)$ `// determine node rank`

4

`// check if a tunnel starts and save offset to the uppermost entry edge`

5 **if** $D_{IN}[i] = 0$ ***or*** $D_{IN}[i+1] = 0$ **then**

6 $o \longleftarrow i - \mathtt{select}_{D_{IN}}(1, nr)$

7 push o on s

8

9 $i \longleftarrow \mathtt{select}_{D_{OUT}}(1, nr)$ `// swith to outgoing edges of node` nr

10

`// check for the end of a tunnel and jump to the right edge using saved offset`

11 **if** $D_{OUT}[i+1] = 0$ **then**

12 $o \longleftarrow$ top of s

13 $i \longleftarrow i + o$

14 pop topmost element of s

15 **return** $\{i, s\}$

Chapter 4

Experimental results

In this chapter, the proposed algorithm's time and compression performance are analyzed through experiments. The experiments were conducted using a computer equipped with an Intel Core i5-1035G1 processor and 16GB of RAM.

To achieve the results, a diverse set of biological sequences, including those of a pan-genomic nature, were employed as input data. The algorithm was executed on these inputs to evaluate its runtime and compression efficacy. Subsequently, these observations were compared with the outcomes obtained from the state-of-the-art de Bruijn graph edge minimization (dBGEM) algorithm by Baier [4], introduced in the initial chapter. It is noteworthy that while Baier's algorithm was coded in C++, our implementation was coded using the RUST programming language.

The expectation is that the heuristics employed in the proposed algorithm yields more compressed outcomes, due to its power to tunnel compensable collisions. However, considerations regarding time complexity arise. The overall time complexity of the heuristics is denoted as $\mathcal{O}(Nw_{max} \log N + n)$, where n is the size of the initial graph, N represents the number of maximal blocks, and w_{max} indicates the maximal width of the blocks. Although N and w_{max} are theoretically upper-bounded by the size of the initial graph (or string) n, in practical scenarios, they often exhibit significantly smaller magnitudes. On the contrary, the time complexity of Baier's dBGEM algorithm is $\mathcal{O}(n \log \sigma)$, where σ is the size of the alphabet.

To illustrate the compression power of the heuristics, consider the example sequence *easypeasybpeasyb*\$. This sequence comprises two (maximal) blocks: one of height three labelled *eas* and the other of height two labelled *peasy*. These blocks collide in a compensable manner, indicating that the optimal compressed version would be of length 10 $(18 - (5 + 6 - 3))$. Indeed, this length aligns with the output of our algorithm, whereas the dBGEM algorithm yields a result of size 12, as it fails to tunnel compensable collisions.

Further analysis is provided through a comparison of the algorithms on two distinct

types of biological data: first, non-pan-genomic, followed by pan-genomic. The data were sourced from `https://www.ncbi.nlm.nih.gov/`, a reputable repository in bioinformatics. These analyses aim to underscore the full potential of our novel approach. Some of the files can be found attached to the implementation, in a package named `data`.

Table 4.1 provides a comparison of the algorithms applied to biological non-pan-genomic data, however, two rows highlighted in grey represent non-biological data. Specifically, the `example.txt` file contains the sequence *easypeasybpeasyb*\$, while the `repetitive.txt` file comprises an artificially generated sequence over the alphabet $\{a, c, g, t\}$, containing numerous compensable collisions. This file serves to show the potential power of the heuristics, particularly when dealing with highly repetitive data.

The second Table 4.2 offers insights into the algorithm's performance on pan-genomic data. As expected, the heuristics demonstrates superior compression rates. However, in some instances, it requires more time than the dBGEM algorithm, as indicated by the red markings. The repetitiveness of pan-genomic data is noticeable, as in most cases, the size of the compressed version is at most 38% of the original size.

To examine the effect of increasing the number of sequences in pan-genomic data on compression and computational time, we conducted additional experiments. Employing three distinct types of data — coronavirus, yellow fever, and salmonella genomes — we executed both algorithms across varying numbers of input sequences to observe their outcomes.

For coronavirus genome data, comprising 10 to 100 sequences, we examined the compression results, as depicted in Figure 4.1, and watched computational time changes, shown in Figure 4.2. Notably, the largest gap in compressed output sizes was 37533 characters. Furthermore, Figure 4.2 indicates a linear trend in the time complexity of the heuristics.

The second two experiments illustrated in Figures 4.3 and 4.4 were conducted similarly to the previous ones, only using sequences of yellow fever virus. In this case, the gap between compressed versions is more significant, and the time complexity of the heuristics seems non-linear.

Lastly, we performed the analysis using salmonella sequences, as shown in Figure 4.5 and 4.6. One sequence of salmonella consists of approximately 3,000,000 characters, thus we were able to perform the algorithms on twenty sequences at most. Despite limitations allowing a maximum of twenty sequences, the heuristics required three hours for computation, whereas dBGEM was completed within one minute. Nevertheless, the time complexity of the heuristics appeared linear in this case.

Input file	Initial size	Tunneled size	Size ratio	Time	Strategy
example.txt	18	12	0.67	0.1	dBGEM
		10	0.56	0.001	heuristics
repetitive.txt	3019	1881	0.62	0.06	dBGEM
		1306	0.43	0.005	heuristics
protein.fasta	5109	5076	0.99	0.06	dBGEM
		4753	0.99	0.02	heuristics
zinc_fingers.fa	10345	10029	0.97	0.10	dBGEM
		8724	0.84	0.03	heuristics
bacteriophage.fasta	34041	33343	0.98	0.10	dBGEM
		28965	0.85	0.10	heuristics
S-cereale.fasta	6837	6288	0.92	0.06	dBGEM
		5359	0.78	0.02	heuristics
huYchr.fasta	3693	3518	0.95	0.06	dBGEM
		2979	0.81	0.01	heuristics
yellow_fever1.fasta	11128	10833	0.97	0.08	dBGEM
		9438	0.85	0.03	heuristics
HIV1.txt	28060	9400	0.33	0.08	dBGEM
		8026	0.29	0.07	heuristics
C_alpha1.fasta	16841	16348	0.97	0.09	dBGEM
		14345	0.85	0.03	heuristics
corona_virus1.fasta	30592	29895	0.98	0.11	dBGEM
		26105	0.85	0.09	heuristics
ecoli_plasmid1.fasta	33854	32891	0.97	0.11	dBGEM
		28635	0.85	0.08	heuristics
salmonicida1.fasta	20578	19893	0.97	0.10	dBGEM
		17318	0.84	0.06	heuristics

Table 4.1: Comparison of the heuristics and the de Bruijn graph edge minimization (dBGEM) strategy launched on the same input data of non-pan-genomic character. The sizes are given by lengths of the strings, and the time is given in seconds. The size ratio is generated as the tunnelled size divided by the initial size. The grey rows indicate non-biological data.

Input file	Initial size	# seq	Tunneled size	Size ratio	Time	Strategy
corona_virus.fasta	275303	9	58294	0.21	0.21	dBGEM
			40668	0.15	0.67	heuristic
prion_protein.fasta	15614	6	5881	0.38	0.10	dBGEM
			4622	0.30	0.04	heuristic
drosophila_protein.fasta	23134	9	9600	0.41	0.90	dBGEM
			7504	0.32	0.06	heuristic
ecoli_plasmid.fasta	190975	5	114653	0.60	0.30	dBGEM
			94965	0.50	0.55	heuristic
HIV.fasta	19706	7	9027	0.46	0.08	dBGEM
			6675	0.34	0.05	heuristic
Calpha.fasta	94119	7	42034	0.45	0.15	dBGEM
			35263	0.37	0.24	heuristic
G_suppressor.fasta	14048	7	5379	0.38	0.07	dBGEM
			4587	0.33	0.04	heuristic
penicilium.fasta	8640	11	3292	0.38	0.06	dBGEM
			2208	0.26	0.02	heuristic
zinc_fingers.fasta	22981	5	9305	0.40	0.08	dBGEM
			7551	0.33	0.06	heuristic
nematocida.fasta	39134	23	35735	0.91	0.13	dBGEM
			30249	0.77	0.10	heuristic
stachybotris.fasta	23348	30	10353	0.44	0.08	dBGEM
			7669	0.33	0.05	heuristic

Table 4.2: Comparison of the heuristics and the de Bruijn graph edge minimization (dBGEM) strategy launched on pan-genomic data. The sizes are given by the lengths of the strings, and the time is given in seconds. The size ratio is generated as the tunnelled size divided by the initial size. The column labelled #seq indicates the number of sequences present in the files. Worse results of the heuristics are depicted in red.

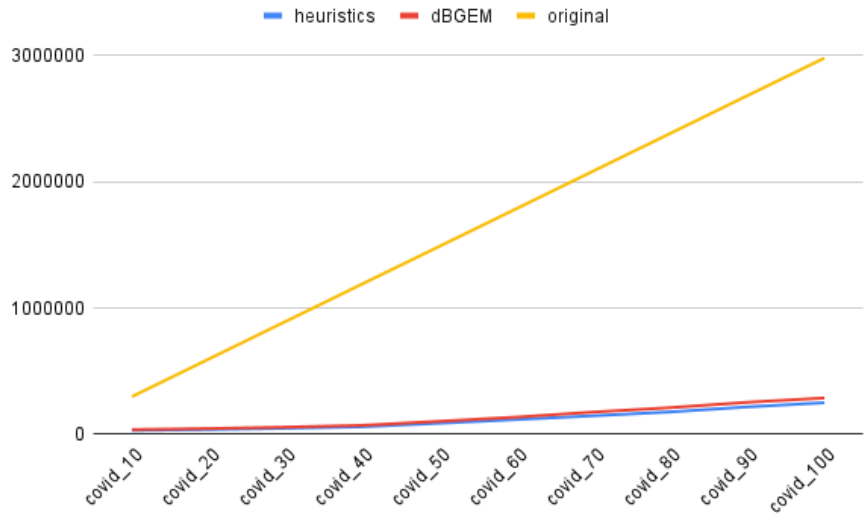


Figure 4.1: Compression rates of the heuristics and dBGEM algorithm on different numbers of input sequences of coronavirus. The x-axis depicts the number of sequences used for the input, and the y-axis depicts the length of either the tunnelled or the original version.

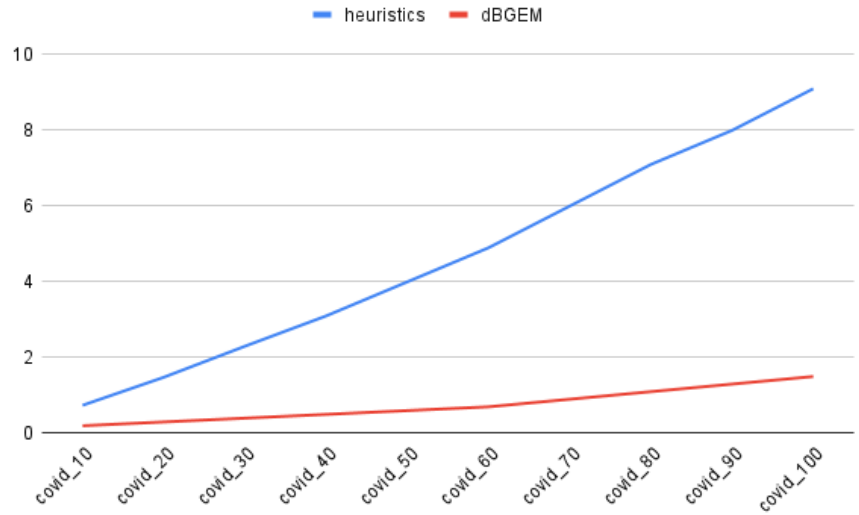


Figure 4.2: Computation time of the heuristics and dBGEM algorithm on different numbers of input sequences of coronavirus. The x-axis depicts the number of sequences used for the input, and the y-axis depicts the time in seconds required by the algorithms.

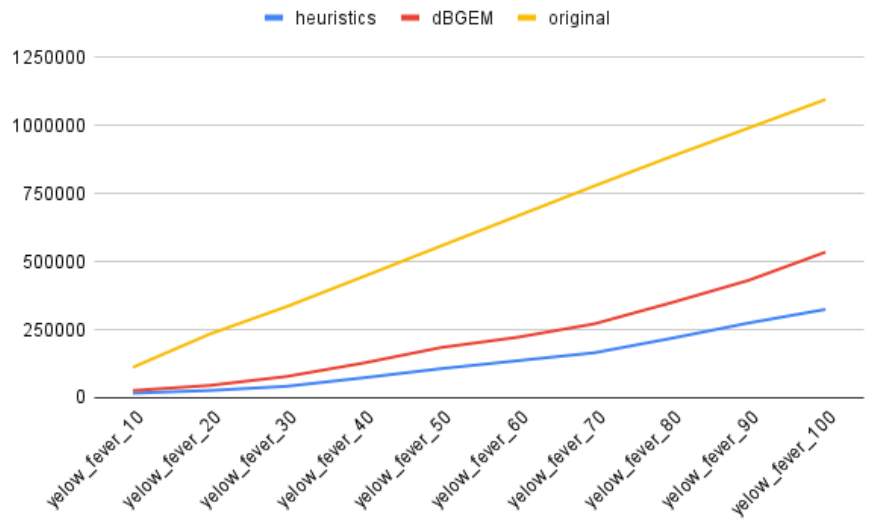


Figure 4.3: Compression rates of the heuristics and dBGEM algorithm on different numbers of input sequences of yellow fever virus. The x-axis depicts the number of sequences used for the input, and the y-axis depicts the length of either the tunnelled or the original version.

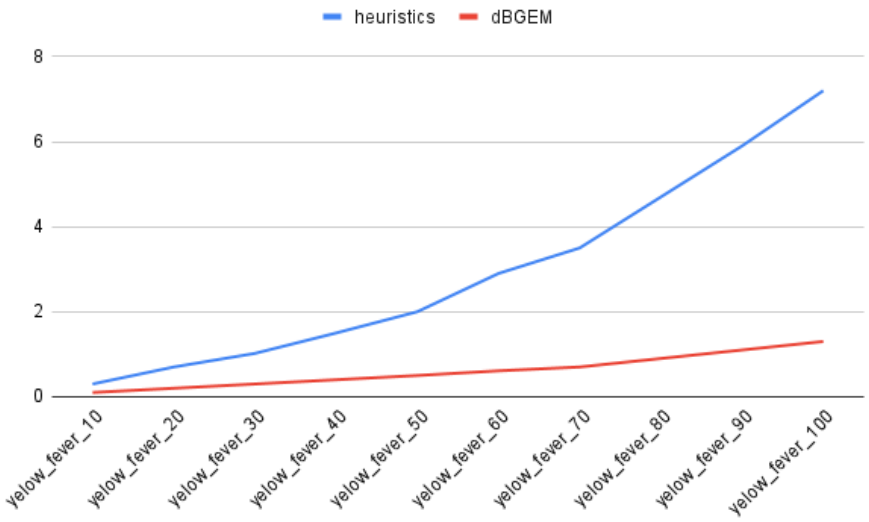


Figure 4.4: Computation time of the heuristics and dBGEM algorithm on different numbers of input sequences of yellow fever virus. The x-axis depicts the number of sequences used for the input, and the y-axis depicts the time in seconds required by the algorithms.

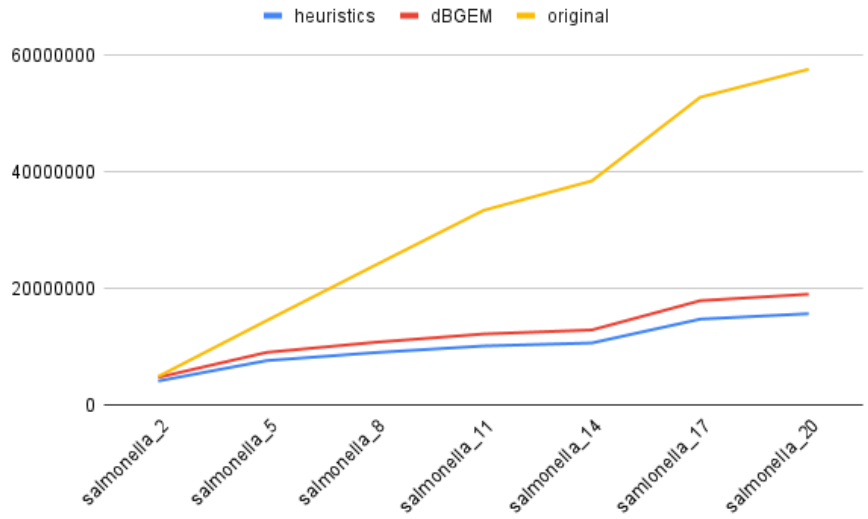


Figure 4.5: Compression rates of the heuristics and dBGEM algorithm on different numbers of input sequences of salmonella. The x-axis depicts the number of sequences used for the input, and the y-axis depicts the length of either the tunnelled or the original version.

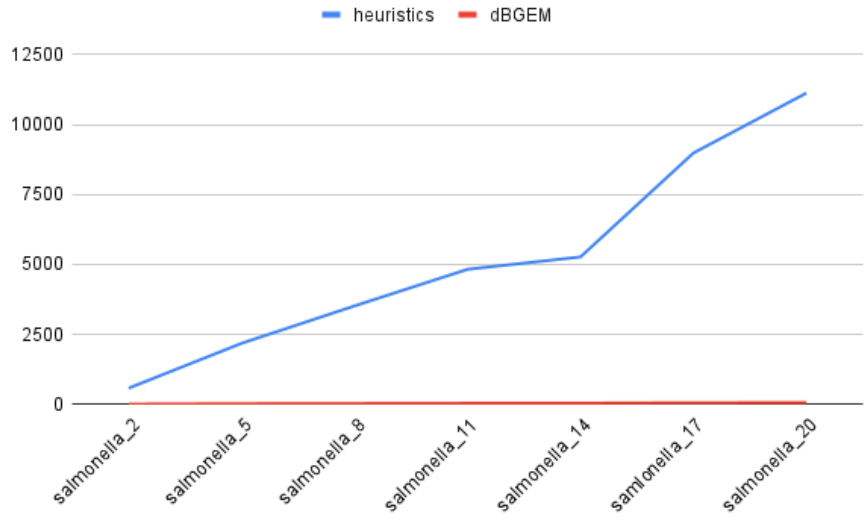


Figure 4.6: Computation time of the heuristics and dBGEM algorithm on different numbers of input sequences of salmonella. The x-axis depicts the number of sequences used for the input, and the y-axis depicts the time in seconds required by the algorithms.

Conclusion

This study introduced a novel algorithm for tunnelling the Burrows-Wheeler Transform, a procedure widely applicable in bioinformatics and other domains. As illustrated in the previous chapter, our algorithm consistently surpasses the state-of-the-art method in terms of compression rates. However, this enhancement is accompanied by an increase in computational time compared to Baier's algorithm. Addressing this time complexity challenge presents an opportunity for future improvement.

One potential approach involves leveraging the prefix-free parsing method discussed in the first chapter. This preprocessing technique condenses input data into a smaller representation while retaining some of the repetitive patterns. Consequently, the input size for tunnelling is often reduced by orders of magnitude, resulting in notable time savings during computation.

The second strategy entails setting a lower bound on the height of blocks. It's worth noting that limiting the width may compromise heuristic accuracy by omitting too thin inner blocks. However, constraining block height is safe (when the solutions do not involve horizontal division), as the valuable higher inner block is inherently at least as tall as the blocks generating the corner collision. By reducing the input size through this limitation, computation time will be correspondingly improved.

Another idea comprises alternating at least one of the aligned solutions with horizontal division. Horizontal division, if implemented properly, requires only one iteration, as all collisions can be addressed in one round, without the introduction of new aligned collisions. However, if this solution is paired with the shortening, i.e. resolution of all corner collisions depends on it, one-row overlappings between blocks are necessary to be computed. This computation is not as straightforward as the computation of one-column overlappings, as the rows are not represented by intervals. Thus, we recommend pairing this solution with vertical division, with the vertical division applied first.

All of these approaches, however, come at the expense of compression efficacy, as they may exclude valuable information necessary for constructing an optimal block set.

Conversely, enhancing compression rates may be achieved at the cost of increased computation time. As discussed in the theoretical analysis, various methods can be applied to address one of the aligned collisions. Opting for the most optimal solu-

tion for a certain scenario typically requires additional computation time, making the improvement suitable only for smaller instances where such precision is feasible.

In summary, for a large group of inputs, this approach outperforms the state-of-the-art algorithm in both time and compression rates. For larger instances, the increase in computation time should be carefully weighed against the potential benefits, although the time complexity of the heuristics appears to be close to linear. Nonetheless, employment of our heuristics is highly recommended, particularly in cases where compression rates take precedence over time considerations.

Bibliography

[1] Donald Adjeroh, Timothy Bell, and Amar Mukherjee. *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Science & Business Media, 2008.

[2] Jarno Alanko, Travis Gagie, Gonzalo Navarro, and Louisa Seelbach Benkner. Tunneling on wheeler graphs. In *2019 Data Compression Conference (DCC)*, pages 122–131. IEEE, 2019.

[3] Uwe Baier. On undetected redundancy in the burrows-wheeler transform. *arXiv preprint arXiv:1804.01937*, 2018.

[4] Uwe Baier. Bwt-tunneling. `https://github.com/waYne1337/BWT-Tunneling.git`, 2020.

[5] Uwe Baier. *BWT tunneling*. PhD thesis, Universität Ulm, 2021.

[6] Uwe Baier, Thomas Büchler, Enno Ohlebusch, and Pascal Weber. Edge minimization in de bruijn graphs. *CoRR*, abs/1911.00044, 2019.

[7] Uwe Baier and Kadir Dede. Bwt tunnel planning is hard but manageable. In *2019 Data Compression Conference (DCC)*, pages 142–151. IEEE, 2019.

[8] Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big bwts. *Algorithms for Molecular Biology*, 14(1):1–15, 2019.

[9] Wheeler Burrows. A block-sorting lossless data compression algorithm. *SRS Research Report*, 124, 1994.

[10] Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the bwt for repetitive text using string compression. *arXiv preprint arXiv:2204.05969*, 2022.

[11] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[12] Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theoretical computer science*, 698:67–78, 2017.

[13] Daniel Gibney and Sharma V Thankachan. On the hardness and inapproximability of recognizing wheeler graphs. *arXiv preprint arXiv:1902.01960*, 2019.

[14] Adrián Goga and Andrej Baláž. Prefix-free parsing for building large tunnelled wheeler graphs. *arXiv preprint arXiv:2206.15097*, 2022.

[15] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

[16] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 data compression conference*, pages 193–202. IEEE, 2009.

Appendix A

The electronic attachment accompanying this work includes the source code and some of the input files used in the final chapter. The source code is also accessible at `https://gitlab.com/wheeler_graphs/wglib` within the `src/lib.rs` package.

The attachment comprises two folders. The `testdata` folder covers some of the input files employed in the previous chapter, while the `src` folder contains the program responsible for constructing the BWT, implementing the heuristics for tunnelling the BWT, functions facilitating the reconstruction of the original text, and various tests.

Following the download, it is essential to verify that all paths are correctly configured.

To execute all tests within the program, simply invoke `cargo test`. To observe the results of the heuristics applied to the provided data, modify the input in the `lib.rs` file within the `test_heuristics` function, then execute
`cargo test test_heuristics -- --nocapture`.