Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Tunnel planning of Burrows-Wheeler transform by employing integer linear programming

Bachelor Thesis

2022
Klára Sládečková

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Tunnel planning of Burrows-Wheeler transform by employing integer linear programming

Bachelor Thesis

Study Programme:	Bioinformatics
Field of Study:	Computer Science and Biology
Department:	Department of Applied Informatics
Supervisor:	Andrej Baláž, MSc.

Bratislava, 2022
Klára Sládečková

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Klára Sládečková
Študijný program:	bioinformatika (Medziodborové štúdium, bakalársky I. st., denná forma)
Študijné odbory:	informatika biológia
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Tunnel planning of Burrows-Wheeler transform by employing integer linear programming
Plánovanie tunelov v Burrows-Wheelerovej transformácií pomocou celočíselného lineárneho programovania

Anotácia: Burrows-Wheerová transformácia (BWT) je metóda, ktorá sa využíva na zvýšenie kompresibility textových reťazcov. V bioinformatike plní dôležitú úlohu pri uchovávaní veľkých genomických dát. Navyše je dôležitou súčasťou FM-indexu využívaného na rýchle vyhľadávanie v biologických sekvenciách. Wheelerov graf je dátová štruktúra generalizujúca BWT. Táto dátová štruktúra umožňuje kompaktnejšiu reprezentáciu genomických dát a zároveň umožňuje indexovať viacero sekvencií naraz. Jeden z možných spôsobov ako vytvoriť Wheelerov graf je tunelovaním BWT. Baier et al. (2019) dokázali, že optimálne tunelovanie BWT je NP-úplný problém a navrhli heuristické riešenie. Cieľom práce bude preskúmať ďalšie možnosti riešenia problému tunelovania BWT.

Cieľ: Cieľom práce je navrhnúť nový spôsob riešenia tunelovania BWT. Jedným z možných spôsobov je formulovať problém ako inštanciu celočíselného lineárneho programovania a vyriešiť túto inštanciu pomocou voľne dostupných ILP solverov. Úlohou študenta bude naštudovať problematiku vytvárania Wheelerových grafov pomocou tunelovania BWT a navrhnúť spôsob akým pretransformovať inštanciu tunelovania BWT na inštanciu celočíselného lineárneho programovania. Následne študent naprogramuje nástroj, ktorý vytvorí Wheelerov graf s využitím voľne dostupného ILP solvera a porovná vytvorený nástroj s nástrojom vytvoreným v práci Baier et al. (2019).

Vedúci:	MSc. Andrej Baláž
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 15.10.2021

Dátum schválenia: 22.10.2021

garant študijného programu

.....

študent

.....

vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Klára Sládečková
Study programme:	Bioinformatics (Joint degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science Biology
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Tunnel planning of Burrows-Wheeler transform by employing integer linear programming

Annotation: Burrows-Wheeler transform (BWT) is a method used to increase the compressibility of strings. In bioinformatics, it fulfils a significant role in storing big genomics data. Furthermore, it is a crucial part of an FM-index, used for a quick search in biological sequences. Wheeler graph is a data structure generalizing BWT. This data structure enables a more compact representation of genomic data and indexing of more biological sequences simultaneously. One of the possibilities for how to create a Wheeler graph is by tunnelling BWT. Baier et al. (2019) proved the BWT tunnelling to be an NP-complete problem and proposed a heuristic solution. The goal of this work will be to find out other possibilities for the BWT tunnelling problem.

Aim: The work aims to propose a new solution to the BWT tunnelling problem. One of the possibilities is to formulate the problem as an instance of integer linear programming and solve this instance using the freely available ILP solvers. The student's tasks will be to study the problem of generating Wheeler graphs using the BWT tunnelling and to propose a transformation from the instance of BWT tunnelling into the instance of integer linear programming. Subsequently, the student will program a tool generating a Wheeler graph by employing a freely available ILP solver and compare the created tool with the program introduced in Baier et al. (2019).

Supervisor:	Master of Science Andrej Baláž
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	15.10.2021
Approved:	22.10.2021

Guarantor of Study Programme

.....
Student

.....
Supervisor

Abstrakt

Burrows-Wheelerová transformácia slúži na bezstratovú transformáciu vstupného textu do podoby, ktorá je ďalej schopná kompresie. Reverzibilná technika kompresie zvaná tunelovanie je následne aplikovaná na Burrows-Wheelerovú transformáciu, čo vedie k skomprimovanej Burrows-Wheelerovej transformácii, ktorá je aj naďalej bezstratovo zvratná. Optimálne Tunelovanie je NP-ťažký problém. V našej práci predkladáme novú metódu riešiacu tento problém v polynomiálnom čase, a to použitím celočíselného lineárneho programovania.

Kľúčové slová: Burrows-Wheelerová transformácia, blok, tunelovanie, celočíselné lineárne programovanie

Abstract

The Burrows-Wheeler transform's purpose is to losslessly transform the input text data into a form with supervenient compressible properties. A reversible compression technique called tunneling is then applied to the Burrows-Wheeler transform, leading to a compressed Burrows-Wheeler transform that is still invertible. However, the tunneling itself is an NP-hard problem. Therefore we propose a novel heuristic for solving this problem in polynomial time, and that is using an Integer Linear Programming solver.

Keywords: Burrows-Wheeler transform, block, tunneling, integer linear programming

Contents

List of Figures

Introduction

One of the urgent problems in the field of Bioinformatics pertains to the efficient management of enormously large text data, such as DNA, RNA, or protein sequences, whose storage requires space of tens of Gigabytes. Operating with such data in the original form would be too expensive both in space and time. A representation of a string called Burrows-Wheeler transform is used for achieving effectiveness. It is a well-known lossless data transformation, with several compelling properties, making it essential to FM-indexing, a method useful for fast computation of complex text operations. Furthermore, this text transformation has a high potential for compression. Tunneling is a new method used for such a compression, providing a decompressible data structure with all the properties of the Burrows-Wheeler transform preserved. However, tunneling is generally an NP-hard problem.

The tunneling strategy is performed on so-called blocks, which are repetitions of the same pattern in the text. The idea behind this compression method is that only the pattern itself and occurrences of it in the text need to be remembered, leading to the minimization of the storage requirements.

However, these blocks may overlay each other and thus cause the tunneled transformation to be irreversible, which we would like to prevent. The problem of the optimal choice of blocks that would not overlay in an unwelcome way and would provide the best compression lies in the NP class. Several heuristical methods running in polynomial time were recently presented, yet all of them were restricted to "run-based" blocks only. Such blocks do not overlap, making the tunneling easier to handle, but the compression rate is not ideal. Therefore, we decided to consider all blocks and solve the problem by reducing it to the well-known NP-complete problem - integer linear programming.

Chapter 1

Preliminaries

The purpose of this chapter is to describe the major parts of the Burrows-Wheeler transform and the tunneling method.

Throughout this work, every logarithm is meant to be of base two. Also, let Σ be a set of totally ordered elements (alphabet of letters) with relation $\prec$ describing the order between two elements, such that the character $\$ \in \Sigma$ is the lowest ordered character in Σ. Every string S over the alphabet Σ (concatenation of elements from Σ) is in this thesis null-terminated, meaning the character $ occurs in S exactly once, and that is at its end.

Let S be a string of length $n \in \mathbb{N}$ over alphabet Σ and i be an integer in the interval $[1, 2, ..., n]$. We denote by

- $S[i]$ the i-th character in S
- S_i the suffix of S starting at position i, i.e. $S_i = S[i]S[i+1]...S[n]$
- $S \prec_{lex} S'$ if S is lexicographically smaller string than the string S' (analogously for $\succ_{lex}$ and $=_{lex}$).
- $|S| := n$ the length of the string S, i.e. the number of elements in S

1.1 Suffix array

First, we describe several data structures such as the suffix array, which will later be essential to building, as well as computing, other data structures and their peculiarities.

Definition 1.1.1 (Suffix array)**.** Let S be a string of length n over alphabet Σ. The permutation of integers in the range [1, n] SA is called *suffix array* if it satisfies the condition $S_{SA[1]} \prec_{lex} ... \prec_{lex} S_{SA[n]}$, that is for $SA[i] = k$ if $S_{SA[k]}$ is lexicographically the i-th suffix.

The suffix array is a popular space-efficient data structure in the field of sequence analysis and provides good time complexity for any pattern searching. The suffix array for a string of length n can be built in $O(n)$ time [11], and searching for a pattern of length m can be done in $O(m \log n + occ)$ time, where occ is the number of occurrences of the pattern in the text. Moreover, if a suffix array is coupled with the information about the longest common prefix of two adjacent suffixes, the pattern searching can be done in $O(m + \log n + occ)$ time [9]. Although this is a highly convenient property of the suffix array, for our purposes, the suffix array plays a crucial part in computing other structures thanks to other features, as we will later see.

1.2 Burrows-Wheeler transform

Definition 1.2.1 (Burrows-Wheeler transform)**.** Let SA be the suffix array of a string S with length n. The *Burrows-Wheeler transform* (BWT) of S is a string L of length n defined as $L[i] := S[SA[i] - 1]$ for $SA[i] > 1$ and $L[i] := \$$ if $SA[i] = 1$. Similarly, we define the F-column of S as the string F, where $F[i] := S[SA[i]]$ for $1 \leq i \leq n$, which can also be obtained by sorting the string L.

We call a *run* a length-maximal continuous substring in a BWT consisting of one character only.

Simply said, the Burrows-Wheeler transform L is the last column of a matrix consisting of lexicographically ordered cyclic permutations of string S as its rows. The first column of such a matrix would then correspond to the string F. An example of strings L and F, as well as the suffix array, can be found in Figure 1.1.

Since both strings L and F are defined over alphabet Σ and not over integers $\{1, ..., n\}$, it is hard to say which position belongs to a character in L and which in F. Therefore a method called LF-mapping was developed, which helps us to identify and map each character in L to its corresponding position in F [2]. To present this method, we first have to explain several notations used for the definition.

We use the notation $\texttt{select}_S(c, i)$ to describe the position of the i-th occurrence of character c in S, $\texttt{rank}_S(c, i)$ to denote the number of occurrences of character c in the substring $S[1]...S[i]$ and $\texttt{C}_S[c]$ to denote the number of characters ordered lower than c, that is, $\texttt{C}_S[c] := |\{i \in \{1, ..., n\} \mid S[i] \prec c\}|$.

Definition 1.2.2 (LF-mapping)**.** Let S be a string of length n over alphabet Σ and let SA and L be its suffix array and its BWT, respectively. The *LF-mapping* LF is a permutation of integers $1, ..., n$ such that $LF[i] = \texttt{C}_L[L[i]] + \texttt{rank}_L(L[i], i)$ for any $i \in \{1, ..., n\}$.

The attentive reader might notice a specific feature, which this definition relies upon. In both strings L and F the orders of precedence of equal characters are identical.

i	SA[i]	LCS[i]	$S[0]...S[SA[i]-1]$	$S_{SA[i]}$	$L[i]$	$F[i]$
0	13	0	readysteadygo	$	o	$
1	8	0	readyste	adygo$	e	a
2	2	1	re	adysteadygo$	e	a
3	9	0	readystea	dygo$	a	d
4	3	2	rea	dysteadygo$	a	d
5	7	0	readyst	eadygo$	t	e
6	1	0	r	eadysteadygo$	r	e
7	11	0	readysteady	go$	y	g
8	12	0	readysteadyg	o$	g	o
9	0	0		readysteadygo$	$	r
10	5	0	ready	steadygo$	y	s
11	6	0	readys	teadygo$	s	t
12	10	0	readystead	ygo$	d	y
13	4	3	read	ysteadygo$	d	y

Figure 1.1: Suffix array SA, LCS-array LCS, Burrows-Wheeler transfrom L, F-column F and prefixes preceding a suffix of string $S = readysteadygo\$$.

This is due to the fact that L points to suffixes that are shifted one character leftwards from those corresponding to F. Consider two positions i and j in L, such that $i < j$ and $L[i] = L[j]$. Since $i < j$, $S_i \prec_{lex} S_j$ which results in $L[i]S_i \prec_{lex} L[j]S_j$, as $L[i] = L[j]$. And that means that the corresponding position to $L[i]$ in F is smaller than the corresponding position to $L[j]$ in F, which implies the uniformity of the orders of precedence.

An important property of the Burrows-Wheeler transform is its invertibility, i.e. having information about a BWT we are able to shape the original string S. LF-mapping is crucial for the reconstruction of the original text from a BWT as it allows us to walk through L in reverse text order, commonly known as "backward-step". We start at the very end of the original text - the character $ or, more precisely, we start at the position in L where this character occurs. Then we use LF-mapping to direct us to the position at which this character is located in F. At this position in L, the previous character of the original text can be found. In this manner, the reverse of the initial string can be rebuilt in linear time, see Figure 1.4 for a better understanding.

This invertible permutation of characters of a string tends to show certain properties, making it highly compressible. Our next concern will be the keystone of such properties called a block (or a prefix interval [4]).

1.3 Block

The purpose of this section is to define a block in a BWT. As mentioned in the previous section, this structure is crucial to BWT compression. Nevertheless, we first have to define the longest common suffix array, which is essential to the computation of blocks in a BWT.

Definition 1.3.1 (Longest Common Suffix Array)**.** Let S be a string of length n and SA be its suffix array. The array LCS of length n is called the *longest common suffix array* (LCS-array) if it satisfies the following condition:

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

.

To put it another way, the LCS-array describes the length of the longest identical strings strictly preceding two adjacent suffixes in the suffix array, i.e. the length of the longest common suffixes of prefixes foregoing two adjacent suffixes in the suffix array, see also Figure 1.1. Having said that, let us finally define a block in BWT - the base element of the BWT compression method called tunneling.

Definition 1.3.2 (Block)**.** Let S be a string over alphabet Σ and L be its BWT. *Block* B in the BWT L is a matrix consisting of consecutive substrings $s_0,\ s_1,\ ...,\ s_k$ of S as its rows such that for $l := |s_0|$ the following conditions hold:

- $\forall i \in \{0,1,...,k\} \quad |s_i| = l$
- $s_0 \ =_{lex} \ s_1 \ =_{lex} \ ... \ =_{lex} \ s_k$
- $\forall i \in \{0,1,...,l\} \quad s_0[i]\ s_1[i]\ ...\ s_k[i]$ is part of a run in L.

We call $h_B := k$ the height of the block and $w_B := l$ the width of block B.

One can imagine a block B as a submatrix of the matrix of all suffixes of a string S being lexicographically ordered, such that every column in B consists of exactly one element. This is easily imagined, yet not quite right. We require blocks to have a certain characteristic, and that is the ability to be tunneled, which we later explain. In order to have this property, each column must be preserved in BWT of S. That is why we will search for blocks using the longest common suffixes array and not the longest common prefixes array [9]. The block can indeed be imagined as a submatrix with all rows equivalent. The supermatrix would not correspond to the matrix of all suffixes of S though, but to the matrix that agrees with the LCS-array of S. Nonetheless, these

i	SA[i]	LCS[i]	$S[0]...S[SA[i]-1]$	$S_{SA[i]}$	$L[i]$	$F[i]$
⋮	⋮	⋮	⋮	⋮	⋮	⋮
5	7	0	readyst	eadygoyho$	t	e
6	1	0	r	eadysteadygoyho$	r	e
⋮	⋮	⋮	⋮	⋮	⋮	⋮
14	10	0	readystead	ygoyho$	d	y
15	13	0	readysteadygo	yho$	o	y
16	4	0	read	ysteadygo$	d	y

Figure 1.2: The difference between a block in LCS-array and a block in LCP-array shown on a string $S = readysteadygoyho\$$. In contrast with Figure 1.1, the sequences *eady* are no more considered as a block according to definition 1.3.2.

blocks can also be found in the matrix of lexicographically ordered suffixes, therefore we can define the starting position of the block, i.e. its top left corner, by the position in the suffix array.

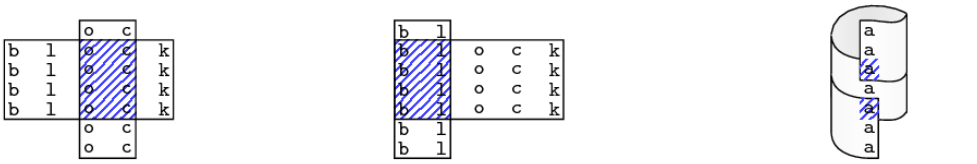
Figure 1.1 shows examples of three blocks, colored black in the fourth column. These blocks can also be found in the next column, which refers to the matrix of sorted suffixes. On the other hand, if the sequence *yho* was appended to the initial string, the block $4 - [5, 7)$ would not match the definition 1.3.2, yet it occurs in the matrix of ordered suffixes, see Figure 1.2.

Henceforth, we will consider only wide height-maximal and left-maximal blocks in terms of not being part of a higher or left-expandable block and having a width size of at least 2. We will unambiguously describe each such block by its starting position and by its width. Thus, we will address block B of width w_B and height h_B starting at position i as $w_B - [i, i + h_B)$.

Notably, we allow two blocks to intersect. As described later, this is a crucial part of the block tunneling problem, thence we define several properties of such intersections (collisions) of blocks.

Definition 1.3.3 (Colliding blocks)**.** Let S be a string of length n over alphabet Σ. Blocks B and B' of S are *colliding* if there exists $1 \leq i \leq n$ such that $S[i] \in B$ and $S[i] \in B'$. Moreover, let B_{IN} and B_{OUT} be two colliding blocks in S, with the block B_{IN} being higher than the block B_{OUT}. We call the collision of blocks B_{IN} and B_{OUT} *compensable* if the following conditions are fulfilled:

- The rightmost and the leftmost columns of block B_{OUT} do not intersect
- At least one row of B_{IN} is not shared



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

Figure 1.3: Visualization of all types of block collisions. Shared regions of blocks are marked with blue diagonal stripes. A similar image was already published in [2].

- The intersection area 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 the blocks are the same and $S[i]$ is in several places, we call the collision a *self-collision*. If at least one of these conditions is not fulfilled, or if the collision is a self-collision, we call the collision *critical*.

Figure 1.3 shows three different types of block collisions. An example of these kinds of collisions in the string $S = readysteadygo\$$ is as follows: blocks $2-[1,3)$ and $4-[5,7)$ form a compensable collision, whereas the block $4-[5,7]$ is critically colliding with the block $3-[5,7)$. Note that for a string $aa...a\$$ any block of height and width greater than one is self-colliding.

By definition, compensably colliding blocks always form a cross consisting of one wider (outer) block and one shorter but higher (inner) block. With that said, it is easy to spot the transitivity of particular compensable collisions. If blocks B_1 and B_2 and blocks B_2 and B_3 are compensably colliding with B_2 be once as the inner block and once as the outer block in these collisions, then the blocks B_1 and B_3 form a compensable collision as well. In this manner, compensably colliding blocks form a folding hierarchy, which will be useful for invertibility issues.

1.4 Tunneling

Definition 1.4.1 (Tunneling)**.** Let S be a string over alphabet Σ, L and F be its BWT and F-column, and B be one of its blocks. The process of *tunneling block* B is defined as follows:

- cross out all positions except those in the first row or the last column in L
- cross out all positions except those in the first row or the first column in F
- remove positions that were doubly crossed out.

The newly created block is called a tunneled block.

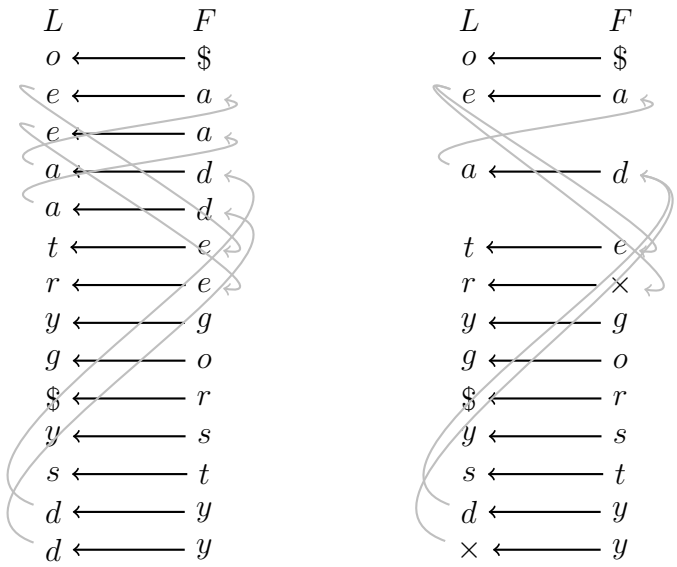


Figure 1.4: The process of tunneling. Above, block $4 - [5, 7)$ of string *readysteadygo*\$ is tunneled. The cross-outs are displayed by crosses and all doubly crossed positions are removed. Any arrows colored gray are related to the LF-mapping.

The idea is that for each block, we only have to remember the starting and ending positions of all rows, and the first row since all the other rows are the same. Every other position in the block can be safely removed. By tunneling a block of width greater than 2 we can shorten both arrays L and F by removing all positions of the block (and therefore edges between them) except those in the leftmost and rightmost columns and the uppermost row of the block, see Figure 1.4. Thus we can reduce the memory requirements for storing the BWT of string S.

It is clear now why we exact blocks to have the property of having each column preserved in the BWT. By tunneling a single column of a block, we merge all characters and replace them with a single one, which would not be possible if they were squandered in the BWT. Computing blocks using the LCS-array, we make sure that each column, starting with the rightmost one, is preserved in the BWT, and only blocks that can later be tunneled will be enumerated.

It is proven that a tunneled BWT is still invertible [2], i.e. the original string from which the BWT was constructed can be rebuilt. This suggests that BWT could be tunneled iteratively and thus would decrease storage requirements with each iteration. However, although the invertibility of BWT is not destroyed by single tunneling, it can be if several specific blocks are chosen to be tunneled. We will discuss this issue more deeply in the next chapter.

Chapter 2

Block choice problem

In this chapter, we focus on the block choice problem, its complexity, and previous solutions. The main part, though, is devoted to the polynomial reduction of the Block choice problem to the Integer Linear Programming problem.

The idea behind dividing collisions between blocks into two groups (compensable and critical) is as follows. Since the Burrows-Wheeler transform of a string S is invertible and by tunneling a single block we do not infract this property, we could iteratively tunnel the BWT of the string S and therefore minimize memory requirements by a vast amount. However, by tunneling two critically colliding blocks the invertibility is disturbed.

When reconstructing the original text from its tunneled BWT, one has to unambiguously determine what the next letter will be. To walk through a tunneled block correctly, all the beginnings and ends of the paths must be explicit, or in other words, they must not be tunneled. In addition, all elements of a column to-be-tunneled must have the same characteristics, that is, they must be equal and belong to the same set of blocks. Otherwise, we would lose the information about the difference, after replacing all of the elements with just one. These requirements then lead us to the definition of different types of collisions and also imply the problem when critically colliding blocks are tunneled at the same time. Thence, we define the following problem.

Definition 2.0.1 (Block choice problem)**.** The *Block Choice Problem* is defined as finding an optimal set of blocks M of a BWT L such that after iteratively tunneling this set of blocks the invertibility of L will be preserved, i.e. no two blocks in M are critically colliding, and the advantage of tunneling is maximized, i.e. no other set of non-critically colliding blocks of L provides better compression rates.

Considering the example in Figure 1.1, the optimal choice of blocks is the set $\{4 - [5, 7)\}$. All other left-maximal and width-maximal blocks are in critical collision with this one and-or do not bring greater benefit to the compression when tunneled.

2.1 Block choice complexity

The set of all blocks of a BWT can be enumerated in polynomial time [4]. Besides the block enumeration, we want to compute the optimal choice of blocks for tunneling. Within this section, we discuss the complexity of making such a choice.

It is easy to see that this problem is in the NP class. The computation of blocks takes polynomial time, and the choice of blocks can be left to the nondeterminism of the Nondeterministic Turing Machine. Note that when no collisions of blocks are allowed, a polynomial-time algorithm for this problem exists. Despite this, the problem is, in general, NP-hard. Uwe Bayer proved this by reducing the Rectilinear picture Rectangle Cover problem (which is itself an NP-hard problem) to the Block choice problem [5].

It is useful to think of blocks as rectangles covering the positions in BWT. Our mission is to pick up rectangles that do not overlap in an inexpedient way and cover as many positions as possible. This concept also gives us an idea. Each tunneled block improves compression by the number of edges it removes but causes extra costs for saving important information about the start and end positions of the tunnel. Therefore, only blocks with gain higher than cost should be considered.

2.2 Related works

Tunnel planning is generally NP-hard, thus several works proposed polynomial algorithms that solve this problem approximately.

In one such work [2], Uwe Baier proposed a greedy algorithm with time complexity $O(n \log |\mathcal{RB}|)$, where $\mathcal{RB}$ is a set of so-called "width-maximal run-blocks" and n is the length of the Burrows-Wheeler transform. This algorithm is based on the idea that given a set of width-maximal run-blocks blocks, we want to pick up the one with the maximal compression rate. After picking such a block, the algorithm modifies the initial set of blocks, and also prices of all blocks compensably colliding with the chosen block, so that the correct information would be considered in the next round. After that, this process is repeated until the set of blocks is empty. Nevertheless, the author himself stated, that this algorithm sometimes does not find the best solution and he gave a clear example of a situation when this greedy strategy is not optimal, although he also stated that such situations do not occur often in practice, so they can be neglected. This approach is a nice baseline, though too complicated and resource-expensive.

Another publication by Uwe Baier and Kadir Dede [5] presented a simple heuristic that outperformed already existing solutions for the Tunnel-planning problem both in resource requirements and compression rate. They used Baier's cost model of width-maximal run-blocks and adapted the cost model to a single block. Afterward, they

estimated the pragmatic upper bound for the cost of a tunnel and the close lower bound for the gain of tunneling a block. Given that they computed a minimal threshold by the statistics of the normal BWT and chose only blocks whose score was greater or equal to this fixed threshold. In a collision-free environment, this ensured that only blocks that bring gain in the sense of data compression will be tunneled. Although the authors themselves stated that this estimation is very practical in the sense of resource requirements during encoding, they also mentioned it is not completely optimal in the sense of compression.

In his thesis [4], Uwe Bayer presented several strategies for tunneling such as the Hirsh strategy, the Greedy strategy ignoring negative side effects as well as the Greedy strategy that considers negative side effects, and the tunneling strategy that uses de Bruijn graph edge minimization. The first strategy is a pragmatic approach based on the idea that only profitable blocks should be tunneled, similar to the first-mentioned work. The other greedy strategies refer to the previously mentioned work, with a little upgrade. The last tunneling strategy was optimized for the purpose of sequence analysis. The main difference is that it tunnels non-overlapping blocks only, which can be used to obtain a tunneled FM-index with a special text sampling scheme [1] and to obtain a tunneled trie preserving useful combinatorial properties. This strategy vastly differs from the one we present, yet, we decided to use the available code for our purposes. We will also make use of the algorithm in the last chapter, where we compare its result with ours.

All of these solutions were limited by focusing on width-maximal run-blocks alone. These blocks are defined similarly as in 1.3.2 except the start and end column must coincide with a run in BWT, not just be part of it. Any collision within these blocks is then always compensable, which makes the tunneling easier to handle. Our implementation, on the contrary, considers every reasonable left-maximal and height-maximal block and thus may find solutions that never even came into question in mentioned works. Uwe Baier himself stated [2]

> It also would be nice to get rid of the restriction of run-based blocks; ... .

so this restriction-riddance is definitely a step forward.

2.3 Reduction to ILP

The Integer Linear Programming (ILP) problem is one of the most famous NP-complete problems, meaning it belongs to the NP class and any other problem in the NP class can be algorithmically reduced to this one in polynomial time. After this, one is left with an instance of the ILP problem and tries to find the optimal solution for it. Although the ILP problem is in NP, which practically means there exists no polynomial

algorithm that would solve this problem optimally, considering its fame, there exists a great amount of approximal ILP solvers working in polynomial time. One of such freely available solvers is Gurobi [8], which we decided to use.

Firstly, we have to clarify which information should be considered in the reduction or, in other words, what is relevant to the final outcome. Obviously, all of the computed non-self-colliding blocks should be involved in the ILP instance, as well as their prices. One might also notice that the price of tunneling two compensably colliding blocks is not equal to the sum of prices of separately tunneling these blocks, so we have to make sure that the final cost of tunneling is correct. Lastly, no two critically colliding blocks should be together in the result, thus there must be constraints in the instance providing this.

Having all reasonable (not self-colliding and large enough) blocks and their prices computed, our goal is to maximize the sum $p_0x_0 + p_1x_1 + ... + p_kx_k$, where $k \in \mathbb{N}$ is the number of the considered blocks and $p_0,\ p_1,\ ...,\ p_k$ are the prices of the corresponding blocks. Binary variables $x_0,\ x_1,\ ...,\ x_k$ would indicate the presence of each block in the final tunneling process, i.e. $x_i = 1$ if the i-th block is in the block choice set and $x_i = 0$ otherwise.

To make sure that the price of the tunneling is computed correctly, we could use the inclusion-exclusion principle as there could possibly be a very large hierarchy of compensably colliding blocks. This is a straightforward attitude, yet not very simple for the computation since one has to enumerate the prices of all regions created by the overlaps. Therefore we will approach this issue differently, which will result in computing prices of shared regions for each pair of compensably colliding blocks only. Furthermore, this approach uses fewer variables and is more clear.

Having a folding hierarchy of compensably colliding blocks being tunneled, it is easy to notice that by subtracting the price of a common region of two closest blocks (blocks that form compensable collision such that no other block that is compensably colliding with both of them covers all the shared positions) from the sum of prices of all blocks, we get the right price. Consider the hierarchy in Figure 2.1 and function $p : w - (a, b] \longrightarrow \mathbb{R}$, which assigns a block its price. We want to compute the price of regions covered by a pattern. Instead of computing the sum $p(a) + p(c) + p(d) - p(a \cap c) - p(a \cap d) - p(c \cap d) + p(a \cap c \cap d)$ (as we would if the inclusion-exclusion principle was applied) we could only calculate the sum $p(a) + p(c) + p(d) - p(a \cap c) - p(c \cap d)$. Thus, we reduce the number of calls of function p from $\sum_{i=1}^{t} \binom{t}{i} = 2^t - 1$ to $t + \binom{t}{2} = t(t+1)/2$ in general (before we know the block choice set), where $t \in \mathbb{N}$ is the number of blocks in a compensable collisions hierarchy. In that manner, we create additional variables for all pairs of blocks that form a compensable collision.

Let $\{p_0, q_0\},\ \{p_1, q_1\},\ ...,\ \{p_m, q_m\}$ be all the $m \in \mathbb{N}$ pairs of compensably colliding blocks. For each such pair, we compute the difference between the sum of the blocks'

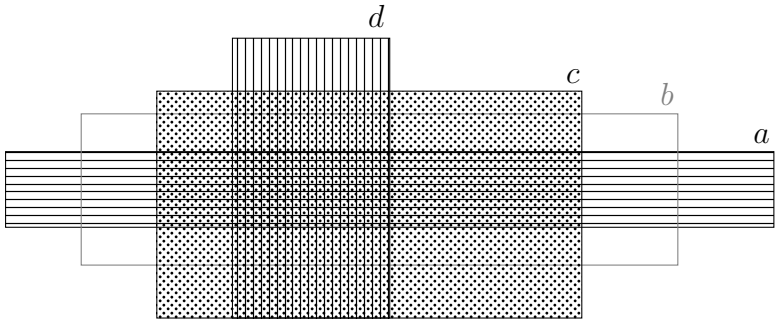


Figure 2.1: Hierarchy of compensably colliding blocks. To-be-tunneled blocks are colored black. Gray-colored blocks are not part of the block choice set.

prices, and the real price of tunneling both blocks. We will address these differences as $shp_0,\ shp_1,\ ...,\ shp_m$. With new binary variables $y_0,\ y_1,\ ...,\ y_m$, we alter the main sum into the sum $p_0x_0 + p_1x_1 + ... + p_kx_k - shp_0y_0 - shp_1y_1 - ... - shp_my_m$. This is the final version of the principal function.

Now, in order to ensure the precision of this tunneling sum, we have to make sure that for every $i \in \{0, 1, ..., m\}$ y_i is set correctly, that is $y_i = 1$ if and only if both blocks addressed by this shared region (p_i and q_i) are chosen to be tunneled, and no other block between them (block that forms a compensable collision with both of them and overlays the intersecting area) is chosen. Formally, we want the statement

$$x_{p_i} + x_{q_i} + \sum_{j=1}^{l}(1 - x_{z_j^i}) \geq l + 2 \iff y_i \geq 1 \tag{2.1}$$

to be always true, where $\{z_1^i, z_2^i, ..., z_l^i\}$ is the set of blocks that are between blocks p_i and q_i. That is possible by adding the following constraints to the ILP instance:

- $\forall i \in \{0, 1, ..., m\} \quad x_{p_i} + x_{q_i} + \sum_{j=1}^{l}(1 - x_{z_j^i}) - (l + 2) < (l + 2) \cdot y_i$
- $\forall i \in \{0, 1, ..., m\} \quad x_{p_i} + x_{q_i} + \sum_{j=1}^{l}(1 - x_{z_j^i}) \geq (l + 2) \cdot y_i$

The first condition ensures the implication from right to left, and the other constraint holds the opposite direction. Imagine that blocks p_i and q_i are chosen to be tunneled, and no other block between them is selected for the tunneling. The left side of the first inequation would therefore equal zero, pushing the right side to be at least one. That would result in the variable y_i being set to one. Now consider variable y_i to be set to one in the solution. The right side of the second inequation would be equal to $l + 2$. The left side of the second condition gains values in the range $[0, l + 2]$, and the highest value, which is, in this case, the only acceptable one, is obtained if and only if the variables x_{p_i}, x_{q_i} are set to one, and the other variables $x_{z_1^i}, x_{z_2^i}, ..., x_{z_l^i}$ are set to zero.

Note that the condition holds even for $l = 0$, which would mean that there is no block in between. In that case, we require y_i be one if x_{p_i} and x_{q_i} are asserted to one, and zero in the other case. And exactly that is handled by both of the conditions.

The final task is to avoid tunneling two blocks that form a critical collision. Let $n \in \mathbb{N}$ be the number of all critical collisions and $\{r_0, s_0\},\ \{r_1, s_1\},\ \ldots\ ,\ \{r_n, s_n\}$ be all the pairs of blocks that are critically colliding. By adding the constraints

$$\forall i \in \{0, 1, ..., n\} \quad x_{r_i} + x_{s_i} \leq 1$$

we ensure that the final tunneled BWT is invertible.

The size of the ILP instance is clearly dependent on the number of blocks. For k blocks, the main function can expand to the size $O(k^2)$. The number of constraints for critical collisions is also quadratically bounded as there only exist $O(k^2)$ possible pairs of blocks. Each of these constraints can be coded to the size $O(\log k)$, so altogether, the constraints take no more than $O(k^2 \log k)$ space. On the other hand, the size of constraints addressing compensable collisions is $O(k^3)$. The number of constraints remains $O(k^2)$ but the size of one constraint can rise to $O(k)$. All in all, the size of the ILP instance is polynomially bounded, cubically to be exact.

The instance created this way can be afterward solved by an ILP solver, and a set of to-be-tunneled blocks is obtained. This set of blocks would correspond to the set of variables assigned with the value one by the ILP solver. In the next chapter, we focus on the practical implementation of this reduction and the follow-up tunneling, as well as the final inversion of a tunneled BWT.

Chapter 3

Practical Implementation

In this chapter, we show the main parts of our algorithm and explain its correctness. We also give notice of the time complexity and the positive and negative sides of this specific computation.

The algorithm consists of four main sections. The previous chapter gave us an idea about the implementation of the second section, which is devoted to the reduction itself. However, in order to determine the blocks that should be tunneled, several pieces of information need to be computed. This will be handled in the first part of our algorithm. After the reduction, the tunneling process follows, which the third section of the algorithm is in charge of. These three sub algorithms together deal with the tunneled BWT construction itself, and their output fulfills one's desires. Nonetheless, for other needs, as well as for testing the correctness of the previous parts of the algorithm, a program taking care of the inversion of a tunneled BWT is introduced in the last, fourth section.

3.1 Basic parts computation

What information is relevant to the reduction? The answer to this question can be found in the last chapter. We need to compute all non-self-colliding blocks and their prices (in the sense of compression), as well as all pairs of colliding blocks and the types of collisions. In addition, for compensable collisions, the price of the shared area needs to be evaluated.

Everything essential for the second part of the algorithm can be evaluated using the precomputed suffix array SA, BWT L, and LF-mapping LF. Several works deal with the efficient and easily implemented ways of enumerating these arrays [11, 6, 4], thus we will not take any concern with it.

3.1.1 Block computation

Previously, we defined a block using the LCS array. This suggests we compute and use the LCS array for the block evaluation in the practical world as well.

The light-headed computation of the LCS array would take $O(n^2)$ time, where n is the size of the original string and, consequently, the size of the suffix array of that string. For each pair of adjacent suffixes in the suffix array, we would compute the length of the longest common suffix of the strings that precede them, which takes $O(n)$ time in general. Doing this for each of the n pairs, we end up with the quadratic time in the worst case.

However, by observing a certain property of the LCS array, we may reduce the computation time, and thus calculate the LCS array effectively. The key fact to observe is as follows:

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

where LF is the LF-mapping and L is the BWT. This equation then holds for every $i > 1$, whereas $LCS[0] = 0$. With this insight, the LCS array can be recursively computed in linear time [4].

Using a stack-based approach, Algorithm 1 enumerates all left-maximal and height-maximal blocks in a BWT. This computation is very similar to the computation of all prefix intervals in Bayer's thesis [4], though slightly different.

The computation of prefix intervals indeed results in enumerating left-maximal blocks, but not height-maximal in each case. Consider the sequence 0, 5, 5, 5, 4, 4, 4 occurring in the LCS-array. Bayer's program would mark two blocks, one with a width of 5 and a height of 3 and the other with a width of 4 and a height of 3. This is not quite right, as the second block can be extended in height and be of size 4×6. The lines $10 - 12$ in Algorithm 1 ensure that truly height-maximal and left-maximal blocks will figure in the result.

After the block evaluation, a filtration of too small and self-colliding blocks is necessary. Too small blocks mark either by being lower than two or having a width of size one at most. On the other hand, self-colliding blocks are those that cover certain positions in the BWT more than once. This actually means that the starting positions of two rows of the block are no further from each other in the original string than the width of the block. Having all starting positions of the block rows presorted, it is easy to see that this examination of a block takes $O(h)$ time, where h is the height of the block.

Algorithm 1: Algorithm enumerating all left-maximal and height-maximal blocks.

Data: LCS array LCS of size n

Result: All maximal blocks in form $w - [a, b)$, where w is the width of the block and $[a, b)$ is the half-closed 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              report block w+1 −[b, i)
9
10             {b′, w′} ⟵ top of stack s
11             if LCS[i] > 1 and LCS[i] < w′ then
12                 push {b, LCS[i]} on s
13
14             {b, w} ⟵ top of stack s
15         if w < LCS[i] then          // possible start of a block of width w
16             push {i − 1, LCS[i]} on s
17
                   // make sure to report all possible block saved in the stack
18     while s is not empty do
19         {b, w} ⟵ top of stack s
20         report block w+1 −[b, n)
21         pop topmost element of s
```

3.1.2 Collisions computation

Our next concern required for the ILP reduction is the detection of all the pairs of blocks that overlap each other. For faster result achievement, we will merge the computation of critical and compensable collisions into one double cycle.

We point out that two blocks are colliding if and only if one of the rows' starting positions of one block is covered by the other block as well. This note can help us achieve the collision verdict more quickly, as we will not have to compare all possible pairs of positions in the blocks. Furthermore, for the compensable collision decision-making, we have to check the first condition alone. The second and third conditions are irrelevant since we consider left-maximal and height-maximal blocks only. When a collision between these blocks fulfills the first condition, the inner block must be implicitly higher than the outer one. The same applies to the third condition.

Algorithm 2: Algorithm indicating the type of collision between two blocks. By notation $v + x$, where v is a vector and x a number, we mean a new vector v' containing all elements of the vector v but increased by x.

Data: block A as a pair $\{w_A, \{a_A, b_A\}\}$ and block B as a pair $\{w_B, \{a_B, b_B\}\}$, where the first elements are the widths of the blocks and the second elements are the intervals of the first columns

Result: the type of collision of these blocks

```
1  begin
2      v_A ⟵ {SA[a_A], SA[a_A + 1], ...., SA[b_A − 1]}
3      v_B ⟵ {SA[a_B], SA[a_B + 1], ...., SA[B_A − 1]}
                    // check if the first column of the wider block is shared
4      if (w_A < w_B and element of v_B is in A) or (w_A ≥ w_B and element of v_A is in
        B) then
5          report critical collision
                                  // check if none of the first columns is shared
6      if (w_A < w_B and not element of v_A in B) or (w_A >= w_B and not element of
        v_B in A) then
7          report no collision
                        // check if the last column of the wider block is shared
8      if (w_A > w_B and not element of v_A + (w_A − 1) in B) or (w_A < w_B and not
        element of v_B + (w_B − 1) in A) then
9          report compensable collision
10     report critical collison
```

The Algorithm 2 shows the pseudo-code for the identification of the type of collision between two blocks. The algorithm relies on other functions, which decide whether a

position in a vector is covered by a block. These functions could be possibly improved in time by employing the binary search, provided that the input structures are ordered. Therefore, when indicating all collisions and comparing all possible pairs of blocks, for each block, we would first sort the starting positions of the block rows and then compare the block with others, using binary search in the mentioned function. The time complexity of the collisions indication would then result in $O(k^2 h \log h)$, where k is the number of all height-maximal and left-maximal non-self-colliding blocks and h is the maximal height of blocks, i.e. $h = max\{h_B \mid \exists a\ \exists w\ w - [a, a + h_B)$ is a block$\}$. The presorting of the first positions is also useful for the filtration part, when casting out the self-colliding blocks, therefore by engaging these functions, the time is spared.

Another improvement would be as follows. In the first chapter, we mentioned the transitivity of certain compensable collisions, so in some cases, the collision indication computation is not necessary. Looking at the compensable collisions as a graph, where vertices correspond to the blocks and all determined compensable collisions are marked as edges from the inner block to the outer one, an easy way of the collision indication occurs. Having two blocks $B1$ and $B2$, we would first ask if there exists a path from block $B1$ to block $B2$ in this graph. If the answer was positive, no further computation would be needed and we would report a compensable collision. Looking for a path between two vertices in a graph would use one of the well-known methods - Depth-first search or Breadth-first search. However, this improvement would not generally result in a better time, thus we decided to omit it.

3.1.3 Computation of prices

To establish the price of a block, i.e. how much its tunneled form benefits the overall compression, we could use Bayer's cost model [2]. This estimation of the benefit and the tax of a block is based on the run-length-encoding method [10], which is widely used within the state-of-art compressors. The gross benefit of a block is given by

$$\text{gross benefit} \ := \ n \ \log\left(\frac{n}{n-tc}\right) - rc \ \log\left(\frac{rc}{rc-tc}\right) + tc \ \left(1 + \log\left(\frac{n-tc}{rc-tc}\right)\right)$$

where $n := |rlencode(L)|$ is the length of the run-length-encoded BWT L, rc is the number of run-characters in $rlencode(L)$ (all characters except for the run-heads, i.e. n minus the number of runs) and tc is equal to the number of run-characters that are removed during the block tunneling.

In addition, Bayer defines a tax in bits for the encoding of aux, which serves for storing the starts and ends of tunnels. As we want to compute a precise price of each block, the estimated tax for an average block is of no use to us. Moreover, using the gross-benefit-based approach led to worse results than the following simple attitude.

In section 2.1 we mentioned the relation between blocks and rectangles covering positions in the BWT. When deciding over two blocks which one to tunnel, we should pick the one covering a greater number of positions. So a block provides better compression rates than the other, if by tunneling it, more information is "lost", i.e. higher number of positions is removed. To put it more simply, block $w_A - [a_A, b_A)$ is more convenient than block $w_B - [a_B, b_B)$ if $(w_A - 1) \cdot (b_A - a_A - 1) > (w_B - 1) \cdot (b_B - a_B - 1)$. These numbers would correspond to the numbers of positions each block removes from the BWT when being tunneled since it is necessary to remember only the first row and the side columns.

3.2 The reduction in practice

As we have already mentioned, for the ILP instance solving we operate with the Gurobi solver. The input file for the Gurobi solver is required to contain three main sections: the main sum with the goal, the constraints, and the boundaries of the variables.

The goal is clear - to maximize the compression score. The main sum can be easily enumerated, and as for the boundaries, all variables we use are binary. What might not be obvious is the Gurobi format of constraints, especially those related to the intersection area variables.

Let $\{p_0, q_0\},\ \{p_1, q_1\},\ ...,\ \{p_m, q_m\}$ be the block pairs that form a compensable collision and $y_0,\ y_1,\ ...,\ y_m$ be the variables that indicate the closest pair of compensably colliding blocks being tunneled. For each variable y_i the following constraints would be added.

Let $x_{z_1^i},\ x_{z_2^i},\ ...,\ x_{z_l^i}$ be the variables for such blocks, that for all $j \in \{1, 2, ..., l\}$ x_{p_i} and $x_{z_j^i}$ as well as x_{q_i} and $x_{z_j^i}$ form a compensable collision and $min(w_{p_i}, w_{q_i}) < w_{z_j^i} < max(w_{p_i}, w_{q_i})$, where $w_{p_i}, w_{q_i}, w_{z_j^i}$ are the widths of the blocks p_i, q_i, z_j^i, respectively. The constraint

$$x_{p_i} + x_{q_i} - (\sum_{j=1}^{l} x_{z_j^i}) - (l + 2) \cdot y_i \leq 1$$

assures the correctness of the case when the variable y_i is set to zero. Similarly, we ensure that the variable y_i is set to one in the right situations by adding the following constraint

$$x_{p_i} + x_{q_i} - (\sum_{j=1}^{l} x_{z_j^i}) - (l + 2) \cdot y_i \geq -l.$$

In this manner, the required equivalence 2.1 for compensable collisions is secured and the correct price will be evaluated.

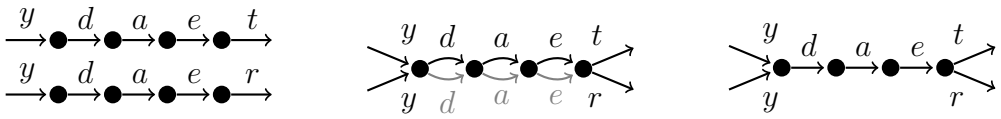


Figure 3.1: The process of tunneling in a Wheeler graph. The leftmost picture shows part of the initial Wheeler graph. In the middle, a multi-graph is depicted after the fusion of inner nodes. All redundant edges are colored gray. On the right, after the removal of redundant paths, tunneled part of the Wheeler graph is shown.

3.3 The tunneling process

The output of the Gurobi solver contains, among other information, the values of all binary variables that took part in the ILP instance. We will tunnel only those blocks, whose corresponding variables were assessed to one. The order of precedence in which these blocks should be tunneled plays no relevant role, thus we will show how to tunnel a single block. Each other block can be tunneled similarly and, as a consequence, in parallel.

The process of compression is held in two stages. Firstly, all to-be-tunneled positions are marked. Given a Wheeler graph [7], this would mean fusing all inner nodes and thus producing a multigraph. Secondly, all the double-marked positions (positions that are not a start or an end of the tunnel) are removed. Regarding the Wheeler graph, this would mean the removal of all redundant edges between the inner nodes, generating a multi-edges-free graph, see also Figure 3.1.

For the tunnel marking, we will use arrays D_{IN} and D_{OUT} of length $n+1$, where n is the length of the BWT L. In the beginning, these arrays contain ones only. We mark the tunnel of a block by setting all positions that are to-be-tunneled to zero. These positions correspond to all the positions covered by the block, which take no part in the leftmost, rightmost column, or the uppermost row. This is done for both arrays D_{IN} and D_{OUT}. In addition, we set to zeros the positions of the leftmost column except for the first position in the array D_{OUT} and all the positions in the rightmost column except for the first in D_{IN}, see the first part of Algorithm 3.

To put it another way, these arrays denote the indegrees and outdegrees of the nodes if we considered a Wheeler graph [4]. At first, all nodes have the indegree and the outdegree equal to one. When marking a block for the tunneling, we merge the inner columns into one node. As a consequence, the indegrees of all nodes except for the ones in the leftmost column and the outdegrees of all nodes except for the ones in the rightmost column are equal to the height of the block. After this fusion, we get a multigraph ready for the final compression, where all redundant paths with a common start and end node are removed.

Algorithm 3: Computation of tunneled BWT L'. First, the to-be-tunneled position are marked in the bit-vectors, then thay are removed from D_{IN}, D_{OUT} and BWT L.

Data: BWT L of size n, set of to be tunneled blocks I and LF-mapping LF
Result: tunneled BWT L' and corresponding arrays D_{IN} and D_{OUT}

```
1  begin
2      create vectors D_IN and D_OUT of size n + 1 filled with ones
           // mark all positions for tunneling in the bit-vectors D_IN and
        D_OUT
3      foreach {w, {a, b}} ∈ I do
4          x ⟵ a
5          h ⟵ b − a
6          for i ⟵ 0 to w − 1 do
7              for j ⟵ 1 to h do
8                  D_OUT[x + j] ⟵ 0
9              x ⟵ LF[x]
10             for j ⟵ 1 to h do
11                 D_IN[x + j] ← 0
12
           // remove all the positions that are marked in both bit-vectors
13     initialize empty string L'
14     p, q ⟵ 0
15     for i ⟵ 1 to n do
16         if D_IN = 1 then
17             L' ⟵ L' + L[i]
18             D_OUT[p] ⟵ D_OUT[i]
19             p ⟵ p + 1
20         if D_OUT = 1 then
21             D_IN[q] ⟵ D_IN[i]
22             q ⟵ q + 1
23     D_OUT[p], D_IN[q] ⟵ 1
24     resize D_OUT to size p + 1 and D_IN to size q + 1
```

This process can be done simultaneously for any set of blocks that does not involve a critical collision. After marking the tunnels in the bit-vectors D_{IN} and D_{OUT}, the tunneled BWT can be constructed as follows.

In order to compress the BWT L and obtain a tunneled BWT $\tilde{L}$, we have to reduce arrays D_{IN} and D_{OUT} of the affected nodes and store only essential positions in $\tilde{L}$. In other words, the outdegrees of all nodes except for the last one, and the indegrees of all nodes except for the first one must be reduced to one. This can be reached by the top-to-bottom traversal of both arrays whereby trimming D_{IN}, D_{OUT}, and L appropriately, as presented in Bayer's thesis [4].

Note that the positions of zero marking in D_{OUT} and the entries to be removed from D_{IN} are identical. The same holds for the positions marked zero in D_{IN} and the entries to be removed from D_{OUT} and L as well. Therefore, by scanning these bit-arrays from top to bottom after emplacing the markings, the desired tunneled BWT can be obtained by deciding whether to keep an entry depending on the zero markings in both arrays, see the second part of Algorithm 3.

3.4 The reversion

In this section, we introduce the strategy for the reconstruction of the original text from a tunneled BWT.

The reverse of the primary string of length n can be rebuilt in $O(n)$ time using the `backward-step` function. This function navigates us through the tunneled BWT, visiting the previous character of the initial string or, in other words, one step backward.

The process of the reverse walk in a tunneled BWT is similar to the one in an unmodified BWT, except we have to remember the offsets to the uppermost row when entering the tunnel so that when leaving, we jump to the correct position in the BWT. In cases when no collisions are allowed, only one additional number would be necessary to remember - the current offset. However, when considering overlapping blocks, we may enter another tunnel before leaving the first and thus have to store a possibly large number of offset values. As a consequence, when leaving a tunnel, we have to choose from all saved offsets. This would be a problem if critically colliding blocks were tunneled, whereas handling compensable collisions is effortless. By definition, two compensably colliding blocks form a cross, and thus the starts and ends of the tunnels form a well-parenthesized expression, i.e. when a tunnel end is encountered, it would belong to the tunnel that was entered last and never left. Therefore, a stack-based approach is a sufficient and easily implemented choice.

Using the `backward-step` function as described in Algorithm 4, the reverse of the original string can be built as follows. Let n be the length of the original string S, L

i	$L[i]$	D_{OUT} D_{IN}	$F[i]$
0	o	1 ← 1	\$
1	e	1 ← 1	a
2	e	1 ← 1	a
3	a	1 ← 1	d
4	a	1 ← 1	d
5	t	1 ← 1	e
6	r	1 ← 1	e
7	y	1 ← 1	g
8	g	1 ← 1	o
9	\$	1 ← 1	r
10	y	1 ← 1	s
11	s	1 ← 1	t
12	d	1 ← 1	y
13	d	1 ← 1	y
14		1 ← 1	

$L[i]$	D_{OUT} D_{IN}	$F[i]$
o	1 ← 1	\$
e	1 ← 1	a
a	1 ← 1	d
t	1 ← 1	e
r	0	
y	1 ← 1	g
g	1 ← 1	o
\$	1 ← 1	r
y	1 ← 1	s
s	1 ← 1	t
d	1 ← 1	y
	0	y
	1 ← 1	

Figure 3.2: Inverse walk in the tunneled BWT L. Above, BWT L, and arrays F, D_{IN} and D_{OUT} are captured before and after tunneling block $4 - [5, 7)$. Starting from position 9, using the LF-mapping, the inverse walk leads to building the original string. When encountering the end of the tunnel (zeros in the D_{IN} array), an offset is saved and then used when the beginning of the tunnel (zeros in array D_{OUT} is reached.)

be its tunneled BWT, s an empty set, and i the position in L where the character \$ occurs. Then, by repeating the commands

1. output $L[i]$
2. $\{i, s\} \longleftarrow$ `backward-step`(i, s)

n-times, we produce S in the reversed order.

Figure 3.2 shows the initial BWT L, arrays D_{IN} and D_{OUT} for string *readysteadygo*\$, and the final tunneled version of each. The arrows between the bit arrays indicate the reverse walk through the tunneled BWT, implicitly showing how Algorithm 4 works.

Algorithm 4: Backward-step function for computing the reversed original string from a tunneled BWT. Similar algorithm was published in [4].

Data: Succinct representation of a tunneled BWT as computed in algorithm 3 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 ⟵ C_{L'}[L'[i]] + rank_{L'}(L'[i], i)          // follow i-th edge
3      nr ⟵ 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 ⟵ i − select_{D_IN}(1, nr)
7          push o on s
8
9      i ⟵ 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 ⟵ top of s
13         i ⟵ i + o
14         pop topmost element of s
15     return {i, s}
```

Chapter 4

Experimental results

This chapter's purpose is to show the differences in results of our and other algorithms launched on the same input data.

The comparison is realized against Uwe's program implemented as the tunneling strategy based on de Bruijn graph edge minimization [3]. For the most accurate results, we used as much as possible from this program, altering only the functions handling the tunneling strategy and the reversion. The final tunneled FM-index's structures are identical, differing in the contained data alone. Therefore, their size difference is a competent comparison technique.

Since this compression method is supposed to extend bioinformatical tools, all of the input data are of biological origin.

First, we would like to depict the main difference between the two algorithms on the string $S := easypeasybpeasyb\$$. The optimal choice of blocks would consist of one block covering the substrings *easy*, and the second block covering the sequences *peasyb*. The ILP reduction strategy would output tunneled BWT of size 10 (which is the optimal size), whereas the de Bruijn graph edge minimization strategy's result would be of size 12. This is due to the non-consideration of compensable collisions in the second method, which leads to significantly faster result achievement, but lower compression intensity. The very opposite applies to our algorithm.

Table 4.1 shows the results of both algorithms launched on different inputs. The first (black-colored) input file contains erythrocyte membrane protein sequence from Plasmodium sp. gorilla clade G2. After that, we launched both programs on the zinc finger protein 213 gene sequence originated in Homo sapiens, showing the main characteristics of the difference. Thirdly, the partial genome of Bacteriophage served as the input. The fourth input file consisted of four repetitions of the last 5000 bases of the 21. human chromosome. The purpose of this input file was to show the results of each algorithm when launched on highly repetitive data, encouraging it to compress the data to a great extent. Lastly, we randomly generated a file containing a large

Figure 4.1: Comparison of the ILP reduction (ILPR) strategy and the de Bruijn graph edge minimization (dBGEM) strategy launched on the same input data. The sizes are given in bytes, and the time is given in seconds (or otherwise, if stated explicitly). The size ratio is generated as the tunneled size divided by the initial size.

Input file	Initial size	Tunneled size	Size ratio	Time	Strategy
example.txt	18	10	0.56	0.1	ILPR
		12	0.67	0.1	dBGEM
protein.fasta	5109	5019	0.98	0.36	ILPR
		5076	0.99	0.06	dBGEM
zinc_fingers.fa	10345	9050	0.87	1111	ILPR
		10029	0.97	0.1	dBGEM
bacteriophage.fasta	34041	29796	0.88	7.5 hours	ILPR
		33343	0.98	0.1	dBGEM
chrom21_rep.fasta	20001	4431	0.22	15 hours	ILPR
		5022	0.25	0.06	dBGEM
repetitive.txt	3019	622	0.21	880	ILPR
		1881	0.62	0.06	dBGEM

hierarchy of compensable collisions. As the table shows, the results of ILPR algorithm are considerably superior to the dBGEM approach within this input. As regards the real data though, the compression rates are not so different.

As we have already pointed out, de Bruijn graph edge minimization does not consider overlapping blocks. Therefore, when launched on several repetitions of the same text (such as the chromosome file in table 4.1), the compression will be limited, and generally would not drop below the length of the repeated sequence alone, whereas this is not the case with our tunneling strategy, as shown in the table. For that reason, in the field of bioinformatics, where data alone are often highly repetitive, and sometimes several alignments need to be stored, our method represents a better option.

Conclusion

As shown in the last chapter, our algorithm provides better compression rates than Bayer's. However, it requires a vast amount of time compared to the de Bruijn graph edge minimization strategy. This inconvenient time consumption could be improved in several ways.

The part that takes the most time is the one handling collisions of blocks. This part's complexity depends on the number of blocks, thus, a simple approach suggests itself. When enumerating the blocks, we could consider only those with even width or those with width dividable by $k \in \mathbb{N}$. This update would, in some cases, lead to faster result-obtaining, but generally not in better time complexity and perhaps worse compression rate even.

In section 3.1.2 we mentioned another possible upgrade. When reaching a decision about a collision of two blocks, information about other collisions might be of use. This holds for situations when the two blocks compensably overlap another block, one from the inside and the other from the outside. This upgrade would, however, cost a lot of memory since one would have to remember all the possibly-useful information. Therefore, the time would not decrease in all cases, only those that contain large hierarchies of compensable collisions. In other cases would this requirement cost us additional time, thus we would not recommend it unless the input data are of the desired appearance. Nonetheless, if decided to be included, the graph would also find its use in the second most time-consuming part of the algorithm - constraint computation for the compensable collisions in the reduction part. In fact, regarding this part, the graph brings a major advantage to the time complexity, reducing it from $O(comp)$ to $O(g)$, where $comp$ is the number of all pairs of blocks that form a compensable collision, and g is the size (the number of vertices) of the largest component of the mentioned graph.

Our presented approach brings new light to the BWT-based compression. A significant improvement is based on the non-restriction of run-based blocks, enhancing the intensity of compression. However, this improvement considerably affected the time. Perhaps with the mentioned ideas and some more, this affection can be minimized, giving us motivation for further research on the topic.

Bibliography

[1] 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.

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

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

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

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

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

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

[8] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. available at `https://www.gurobi.com`.

[9] 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.

[10] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56. Springer, 2005.

[11] 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: Content of the electronic attachment

In the electronic attachment attached to the work can be found the source code as well as the input files mentioned in the last chapter. The source code can also be found at https://github.com/SladeckovaKlara/BWT-Tunneling-ILP.git.

The attachment contains two folders. The testdata folder covers all input files used in the last chapter. The seqana folder contains the program handling the construction of a tunneled FM-index and the program for inversion of a tunneled FM-index.

For full functionality, several additional downloads are necessary, see also files README.md. After the installation, check if all paths are set correctly. For instance, if the path in Make.helper to sdsl-lite is accurate, also if the prefix paths in files in lib/pkgconfig/ are set according to the environment. Moreover, if the Gurobi file is not installed directly in the BWT-Tunneling-ILP directory, please change the system call in tfm_index.hpp in line 457.

After the installation, simply call make to compile all programs. For usage information, execute the programs without parameters.