UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

ZAKLJUČNA NALOGA
(FINAL PROJECT PAPER)

TRANSFORMACIJA BURROWS-WHEELERJA
(BURROWS-WHEELER TRANSFORM)

DOROTEJA VUJINOVIĆ

2021

ZAKLJUČNA NALOGA (FINAL PROJECT PAPER)

VUJINOVIĆ

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga
(Final project paper)

Transformacija Burrows-Wheelerja

(Burrows-Wheeler Transform)

Ime in priimek: Doroteja Vujinović
Študijski program: Bioinformatika
Mentor: prof. dr. Andrej Brodnik

Koper, september 2021

Ključna dokumentacijska informacija

Ime in PRIIMEK: Doroteja VUJINOVIĆ

Naslov zaključne naloge: Transformacija Burrows-Wheelerja

Kraj: Koper

Leto: 2021

Število listov: 42 Število slik: 8 Število tabel: 3

Število referenc: 18

Mentor: prof. dr. Andrej Brodnik

Ključne besede: Transformacija Burrows-Wheelerja, BWT, besedilo, ujemanje vzorcev, pripona, algoritem

Izvleček:

Najprej uvajamo pojem Burrows-Wheelerjeva transformacija, skupaj z zapisom in metodo analize, uporabljeno v celotni diplomski nalogi. Podrobno predstavljamo, kako deluje BWT, in analiziramo njegovo računsko kompleksnost. Nadalje predstavljamo podatkovne strukture, kot sta drevo končnic in matrika pripon, ter koncept ujemanja vzorcev in njihov odnos z BWT. Na koncu si oglejmo nekaj najpomembnejših aplikacij BWT v bioinformatiki in računalniški biologiji.

Key document information

Name and SURNAME: Doroteja VUJINOVIĆ

Title of the final project paper: Burrows-Wheeler Transform

Place: Koper

Year: 2021

Number of pages: 42 Number of figures: 8 Number of tables: 3

Number of references: 18

Mentor: Prof. Andrej Brodnik, PhD

Keywords: Burrows-Wheeler Transform, BWT, text, pattern matching, suffix, algorithm

Abstract:

First, we introduce the notion of Burrows-Wheeler Transform, along with notation and method of analysis used throughout the thesis. We present in detail how does the BWT work, and analyze its computational complexity. Continuing, we present data structures such as suffix tree and suffix array, as well as the concept of pattern matching and their relationship with BWT. Finally, we take a look at some of the most important applications of BWT in bioinformatics and computational biology.

ACKNOWLEDGEMENTS

I would like to express my very great appreciation to my mentor and professor Andrej Brodnik, firstly for his adaptiveness and readiness to help me finish my studies during the COVID-19 pandemic, secondly, for his support, patience, encouragement and guidance during the writing of this final project paper.
Also, I would like to thank my family for being my biggest support from when I was young, up to now.

LIST OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVATIONS

i.e.	that is
et al.	and others

1 INTRODUCTION

The Burrows-Wheeler Transform (BWT)[1, Chap. 1] uses the idea of muddling (permuting) the letters in a document to make it easier to find a compact representation and to perform other kinds of processing. What is amazing about the BWT is that it makes it very easy to find the unique correct permutation very quickly. For example, for the following line from Hamlet's famous soliloquy:

"To be or not to be: that is the question, whether tis nobler in the mind to suffer the slings and arrows of outrageous fortune."

we get the transformed text:

"sdoosrtesrsefeeoe:nsrrtdn,r h onnhbhhbglfhuhnofu anttttw mltt bs ioaiui Tttn i fne r eoeetraoguiwi e ao es e. urqstoo o"

Notice that many characters in the transformed text appear in runs, or very close to previous occurrences. This clustering of characters makes compression very easy. The point is that the transform makes the encoding task a lot simpler, and importantly, can give compression that is comparable with the very best lossless compression methods, i.e methods that allow the original data to be perfectly reconstructed from the compressed data.

BWT is useful for a lot more than compression because it contains an implicit sorted index of the input string. In this thesis we will review some of its other uses especially for pattern-matching and full-text indexing, which leads to applications in bioinformatics.

The Burrows-Wheeler Transform method is often referred to as "block sorting", because it takes a block of text and permutes it. The main disadvantage of the block-wise approach is that it cannot process text character by character; it must read in a block (typically tens of kilobytes) and then compress it. This is not a limitation for most purposes, but it does rule out some applications that need to process data on-the-fly as it arrives. Another important point is that the text can be sorted.

1.1 Genesis of the Burrows-Wheeler Transform

One of the last 20th century breakthrough in general-purpose lossless compression methods was Burrows and Wheeler's enigmatic transform, the BWT. David Wheeler had come up with the transform as early as 1978, but it wasn't until 1994 that, with the help of Mike Burrows, the idea was turned into a practical data compression method. The research report involved rearranging the characters in a text before encoding and then arranging them back in their original order in the decoder. The fact that the original can be re-created at all is somewhat astonishing, and their early work took some time to receive the recognition it deserved. By the late 1990s, researchers began to realize that the BWT approach might be useful for more than just compressing text. Because BWT happens to

sort the text in alphabetical order, the permuted text has the benefit to act as a kind of dictionary for the original text. Traditionally an index and the compressed text would be stored separately, even though they contain effectively the same information. The BWT is an intermediate representation that is halfway between text and index; the original text can be efficiently reconstructed, yet sorted lists are suitable for binary searching, giving very fast searching for arbitrary fragments in the text. [1, Chap. 1.2]

1.2 Notation, definitions and structure of the thesis

First we will define some terms and notation that will be used in the thesis. Throughout the paper we will be coding a string T of n characters, $T[1 \ldots n]$ ($n \geq 1$), containing n characters over an alphabet $T[i] \in \Sigma = \{\sigma_1, \sigma_2, \ldots \sigma_{|\Sigma|}\}$ with size $|\Sigma|$. The array $R[1...n]$ will represent an array of references to rotated strings in the input text T. Furthermore, a string $P[1 \ldots m]$ ($1 \leq m \leq n$), where $P[j] \in \Sigma$. Finally, the compression of text which first transforms text using the BWT we call with a generic term BCompress.

Analyzing an algorithm means predicting the resources that the algorithm requires. Generally, we measure the time needed per operation (time complexity) and the amount of space we need to store the data structure (space complexity). We shall assume random-access machine (RAM) model of computation as our implementation technology.

There are different methods to perform the analysis. In this thesis we will use worst-case analysis in the comparison model measuring the number of comparisons needed (which is in linear relation with operations in a data structure). Worst case [14, Chap. 2.2] analysis looks at the worst-case running time and provides an upper bound for time in all cases. We give these bounds on performance based on asymptotic notation [14, Chap. 3.1]. Specifically, O-notation For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}.$$

We measure space complexity by counting the number of registers, in the RAM model [14, Chap. 2.2], needed to store the data structure. For the (uniform-cost) RAM model we assume that one register (space in memory) is needed to store one value from our data structure.

Section 2 continues to introduce Burrows-Wheeler Transform, and describes its relationship suffix trees and suffix arrays.

In Section 3 we perform computational complexity analysis of the BCompress, and we do the encoding of the BWT output created in Section 2.

Details about pattern matching and its relationship with BWT are described in Section 4.

Some of the applications of BWT in bioinformatics and computational biology can be found in Section 5. For instance, given the strong relationship between the BWT and data

structures such as suffix trees and suffix arrays, it can be expected that the BWT can be used in the identification and analysis of repetitions. Finding repeats can be viewed as a variant of the pattern matching problem.

2 HOW DOES BURROWS-WHEELER TRANSFORM WORK

Let us first look in detail at how the Burrows-Wheeler Transform is implemented in practice. We will see how to perform the encoding and decoding in *O(n)* space, and *O(n*log*n)* time. Using a few techniques, the time can be reduced to *O(n)*. [1, Chap. 2]

2.1 The forward Burrows-Wheeler Transform

The forward transform essentially involves sorting all rotations of the input string, which clusters together characters that occur in similar contexts. Table 1.a shows the matrix *A* containing all rotations that would occur if the transform is given *T* = `mississippi` as the input. Table 1.b shows the result of sorting *A*, which we will refer to as A_s. The last column of A_s (usually referred to as *L*) is the Burrows-Wheeler Transform of the input.

Table 1: Matrices *A* and A_s

A=			$=A_s$
	mississippi	**imississipp**	
	ississippim	**ippimississ**	
	ssissippimi	**issippimiss**	
	sissippimis	**ississippim**	
	issippimiss	**mississippi**	
	ssippimissi	**pimississip**	
	sippimissis	**ppimississi**	
	ippimississ	**sippimissis**	
	ppimississi	**sissippimis**	
	pimississip	**ssippimissi**	
	imississipp	**ssissippimi**	
	(a)	(b)	

Rather than use $O(n^2)$ space as suggested by Table 1, we can use array *R*[1...*n*]. Initially *R*[*i*] is simply set to *i* for each *i* from 1 to *n* to represent the unsorted list. It is then sorted using the substring beginning at *T*[*R*[*i*]] as the comparison key. Figure 1 shows the result of sorting: for example, position 11 is the first rotated string in lexical order (`imiss…`), followed by position 8 (`ippim..`) and position 5 (`issip…`).

The array *R* directly indexes the characters in *T* corresponding to the first column of A_s (referred to as *F* in BWT literature). The last column of A_s (referred to as *L*) is the output of BWT and can be read off as *T*[*R*[*i*]-1], where *i* ranges from 1 to *n*. In this case the transformed text is *L* = `pssimipssii.` We also need to transmit an index *a* to indicate to the decoder which row of A_s contains the original string *T*. In this case the index *a* = 5 is included.

The transform is completed using just *O(n)* space (for *R*). The time taken is *O(n)* for the creation of the array *R*, plus the time needed for sorting *O(nlogn)*. [1, Chap. 2.1]

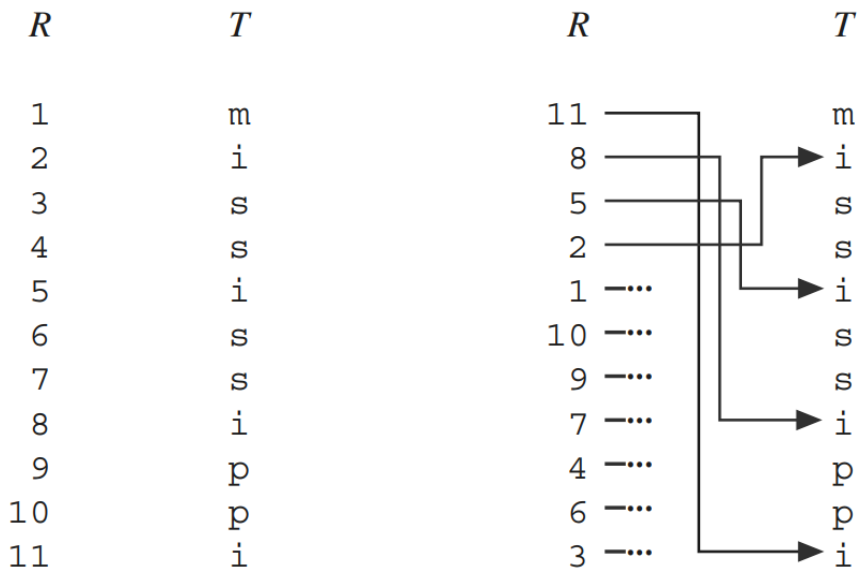

Figure 1: The *R* array used to sort the sample file `mississippi`

2.2 The reverse Burrows-Wheeler Transform

The reverse transform - taking a BWT permuted text and reconstructing the original input *T* - is somewhat more difficult to implement than the forward transform, but it can still be done in *O(n)* time and space if care is taken.
We will use the decoding of the string `mississippi` as the running example. Figure 2 shows the matrix A_s for this example, with columns *F* and *L* labeled. A_s is not sorted explicitly in practice, but we shall use it in meantime to illustrate how decoding can be done. The decoder can determine *F* simply by sorting *L*, since it contains exactly the same characters, just in a different order — each column of A_s contains the same set of characters because the rows are all the rotations of the original string. In fact, *F* need not be stored, as it can be generated implicitly by counting how often each character appears in *L*.
Looking at A_s helps us to see the information that is needed to perform the decoding.
Given just *F* and *L*, the key step is determining which character should come after a particular character in *F*. Consider, for example, the two rows ending with a `p` (rows 1 and 6). Because of the rotation, the order of these two rows is determined by the characters that come after the respective occurrences of `p` in *T* (`imi`... and `pim`... respectively). Thus the first occurrence of `p` in *L* corresponds to the first occurrence of `p` in *F*, and likewise with the second occurrence. This permits us to work through the text backwards: if we have just decoded the second `p` in *L*, then it must correspond to the one in row 7 of *F*. Looking at row 7, the *L* column tells us that the `p` was preceded by an `i`. In turn, because this is the second `i` in *L*, it must correspond to the second `i` in *F*, which is in row 2. We carry on traversing the *L* and *F* arrays in this way until the whole string is decoded in reverse.

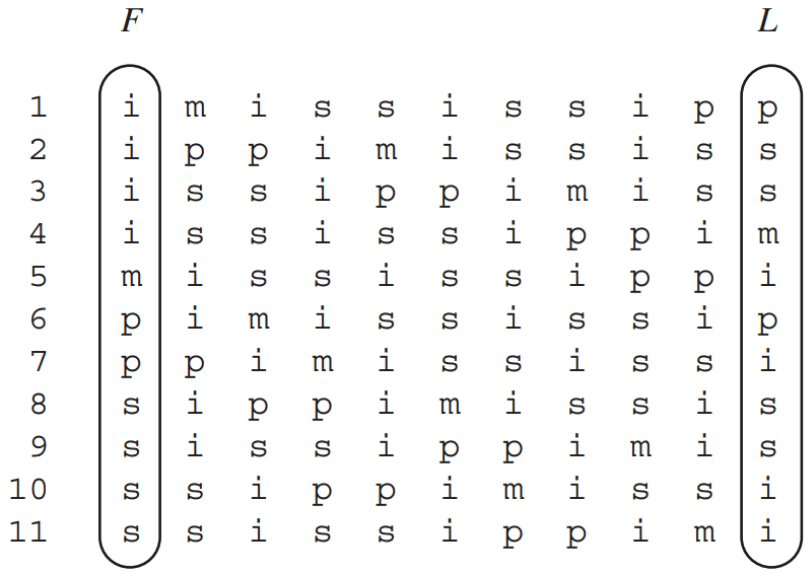

Figure 2: The array A_s for `mississipi`; F and L are the first and last columns respectively

In practice the decoder never reconstructs A_s or F in full, but implicitly creates indexes to represent enough of its structure to decode the original string. L is stored explicitly (the decoder just reads the input and stores it in L), but F is stored implicitly to save space and to efficiently provide the kind of information needed during decoding.
Figure 3 shows three auxiliary arrays that are useful for decoding. $K[c]$ is simply a count of how many times each character c occurs in F, which is easily determined by counting the characters in L. $M[c]$ locates the first position of character c in the array F, so K and M together effectively store the information in F. $C[i]$ stores the number of times the character $L[i]$ occurs in L earlier than position i; for example the last character in L is `i`, and `i` occurs 3 times in the earlier part of L. These three arrays make it easy to traverse the input in reverse.

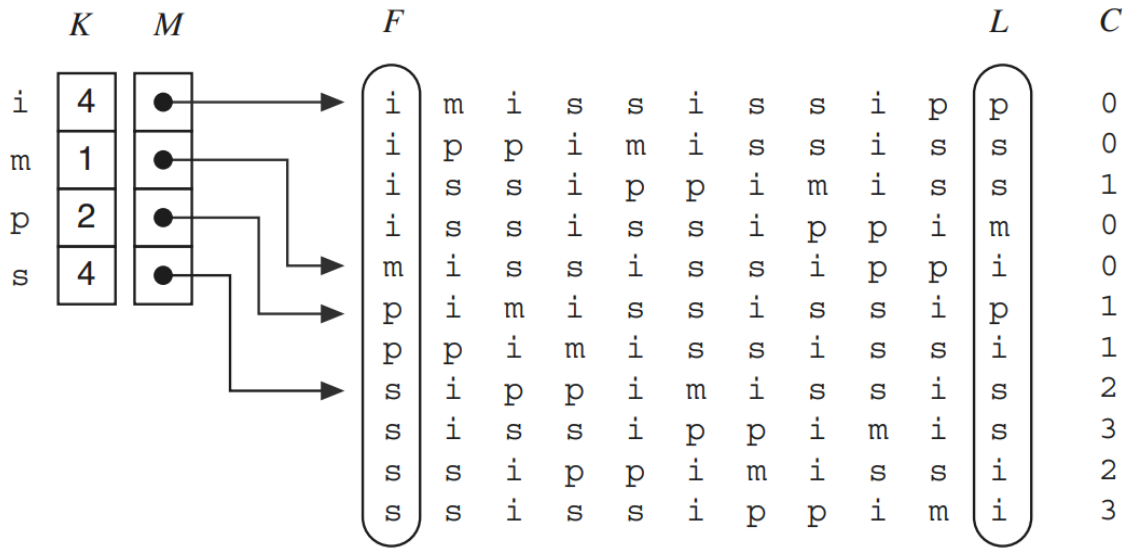

Figure 3: The array (A_s) that is implicitly reconstructed to decode the string `pssmipissii`

Algorithm 1 shows how the input (transformed text *L* and starting index *a*) is used to construct these three arrays, which are then used to produce *Q*, the decoded text. The first step is simply to count the characters into *K* by going through transformed text *L*, shown in lines 1 to 7 of the algorithm. At the same time, it is convenient to construct *C* by recording the value of *K* before each increment. The array *M* is then constructed in lines 8 to 12 by accumulating the values in *C*. We now have sufficient structures to decode the text in reverse, which happens in lines 13 to 17.

Algorithm 1 Reconstruction of the original text

BWT-Decode(L, a)

```
1:  for c ← 1 to |∑| do
2:      K[c] ← 0
3:  end for
4:  for i ← 1 to n do
5:      C[i] ← K[L[i]]
6:      K[L[i]] ← K[L[i]] + 1
7:  end for
8:  sum ← 1
9:  for c ← 1 to |∑| do
10:     M[c] ← sum
11:     sum ← sum + K[c]
12: end for
13: i ← a
14: for j ← n downto 1 do
15:     Q[j] ← L[i]
16:     i ← C[i] + M[L[i]]
17: end for
```

Reversing the BWT this way requires four arrays (*L*, *K*, *C* and *M*). *K* and *M* contain just $|\Sigma|$ entries (the characters are represented by integers from 1 to $|\Sigma|$) and are likely to be of negligible size; *L* and *C* contain *n* values, and hence use *O(n)* space. We would normally also have to allow for *Q*, which uses *O(n)* space to store the backwards string before it can be stored in the correct order. The time taken is also $O(n) + O(|\Sigma|)$, since the main work is in the two passes through the *n* input items — once to count them, and once to decode them. [1, Chap. 2.2]

2.2.1 Decode the string in its original order

It may be inconvenient that the output is generated backwards, and there are ways to address this. The simplest approach is to reverse the order of the string at encoding time. This should not take any extra time, since the whole string must be read into memory anyway — we simply fill the array T in reverse. If the transformed text is to be decoded

multiple times, it is possible to store one or more auxiliary arrays that enable us to traverse sections of the text at will. This can be useful for pattern matching because it allows segments of the original string to be read off when needed for matching, but still relates the data to the implicit sorted array A_s, which provides access to a sorted list of strings that are useful for searching.

As we can see from line 19 of Algorithm 1, the value $C[i] + M[L[i]]$ is the key to navigating through L to decode the original string. Instead of doing the decoding immediately (which was in lines 16 to 20 of Algorithm 1), an array V is created to store the navigation information, shown in Algorithm 2. This array can then be used to step backwards through the original characters. The values of V for our example are shown in Figure 4.

Algorithm 2 Creating the array V to allow for efficient future decoding of the input

Compute-Array-V(C, M, L)

```
1: i ← a
2: for j ← n downto 1 do
3:     V[i] ← C[i] + M[L[i]]
4:     i ← V[i]
5: end for
```

It is just as easy to create an auxiliary array that will decode the original text forwards rather than backwards. This array will be called *W*, and it identifies the position of the character in *L* that comes after the present one, compared with *V* , which gives the position that comes before. As for *V* , this new array is not essential for decoding, but it can be useful because it preserves access to the sorted structure of *L*, which can be exploited during pattern matching. Algorithm 3 shows how the *W* array can be created. Note that the array *M* that was created in Algorithm 1 is used, and that afterwards its contents are changed so they are no longer valid. Like *V*, the *W* array is created in just *O(n)* time. Figure 4 shows the values of *W* for our example.

	F										*L*	*V*	*W*
1	i	m	i	s	s	i	s	s	i	p	p	6	5
2	i	p	p	i	m	i	s	s	i	s	s	8	7
3	i	s	s	i	p	p	i	m	i	s	s	9	10
4	i	s	s	i	s	s	i	p	p	i	m	5	11
5	m	i	s	s	i	s	s	i	p	p	i	1	4
6	p	i	m	i	s	s	i	s	s	i	p	7	1
7	p	p	i	m	i	s	s	i	s	s	i	2	6
8	s	i	p	p	i	m	i	s	s	i	s	10	2
9	s	i	s	s	i	p	p	i	m	i	s	11	3
10	s	s	i	p	p	i	m	i	s	s	i	3	8
11	s	s	i	s	s	i	p	p	i	m	i	4	9

Figure 4: The auxiliary arrays *V* and *W* which can be used to decode the sample string

Algorithm 3 Creating the array W to allow for future decoding of the input

Compute-Array-W(M, L)

```
1: for i ← 1 to n do
2:     W[M[L[i]]] ← i
3:     M[L[i]] ← M[L[i]] + 1
4: end for
```

W can then be used to generate the original text in its correct order using the simple sequence shown in Algorithm 4.

Algorithm 4 Decoding the original text in its correct order using W

Decode-With-Array-W(W, L)

```
1: i ← a
2: for j ← 1 to n do
3:     Q[j] ← L[i]
4:     i ← W[i]
5: end for
```

If both forwards and backwards generation of the original text is needed, it is possible to create *V* and *W* in one pass as shown in Algorithm 5. [1, Chap. 2.2]

Algorithm 5 Creating both the V and W arrays in one pass

Compute-Arrays-V-And-W(M, L)

```
1: for i ← 1 to n do
2:     V[i] ← M[L[i]]
3:     W[M[L[i]]] ← i
4:     M[L[i]] ← M[L[i]]
5: end for
```

2.3 The BWT, Suffix Trees, and Suffix Arrays

The Burrows-Wheeler Transform has a very close relationship with suffix trees and suffix arrays — the array of indexes to the sorted array of substrings generated during the transform is essentially a suffix array, which in turn is a representation of the information in a suffix tree. As pointed out by Burrows and Wheeler in their original work [2], the problem of sorting the rotated matrices is the major bottleneck in performing the transformation.

To see the relationship [1, Chap. 5.1] between the BWT and these data structures, consider Table 2 and Figure 5, which show the list of suffixes, the suffix tree, and the BWT sorted rotation matrix for the sample string T = `mississippi`. In this example a `$` character is used to denote the end-of-string; it isn't strictly necessary for the BWT, but it does simplify the descriptions by marking the end of each suffix. The L array, the final BWT output, is given in the last column of the table. We have also included the corresponding BWT output character (given in parenthesis) at each leaf node in the suffix tree. Given that the suffix array records the index of the sorted suffixes of a string, it is easy to see the relationship between the BWT and suffix arrays.

Given that leaves are sorted, the BWT output can be obtained by a simple traversal of the leaf nodes in the suffix tree from left to right. Let l_i ($1 \leq i \leq n+1$) be the label of the i-th leaf node in the suffix tree (scanning left to right). Recall that l_i is the starting position of the i-th suffix in the input string T. Thus, at the i-th leaf node, with n as the length of T, we obtain the corresponding BWT output as follows:

$$L[i] = BWT[i] = \begin{cases} T[l_i - 1], & i \neq 1 \\ T[n+1] = \$, & otherwise \end{cases}$$

Table 2: The suffixes and the rotation matrix

Suffixes	ID	Sorted Suffixes	Sufix Array	Sorted Rotations (A_s matrix)	BWT Output (L)
mississippi$	1	$	12	$mississippi	i
ississippi$	2	i$	11	i$mississipp	p
ssissippi$	3	ippi$	8	ippi$mississ	s
sissippi$	4	issippi$	5	issippi$miss	s
issippi$	5	ississippi$	2	ississippi$m	m
ssippi$	6	mississippi$	1	mississippi$	$
sippi$	7	pi$	10	pi$mississip	p
ippi$	8	ppi$	9	ppi$mississi	i
ppi$	9	sippi$	7	sippi$missis	s
pi$	10	sissippi$	4	sissippi$mis	s
i$	11	ssippi$	6	ssippi$missi	i
$	12	ssissippi$	3	ssissippi$mi	i

In the Figure 5 the number at each leaf node in the suffix tree corresponds to the suffix ID, which indicates the starting position of the suffix in the original sequence *T*. The character in parenthesis at each leaf node corresponds to the BWT output character for the leaf. The label on each edge corresponds to a substring of *T*.

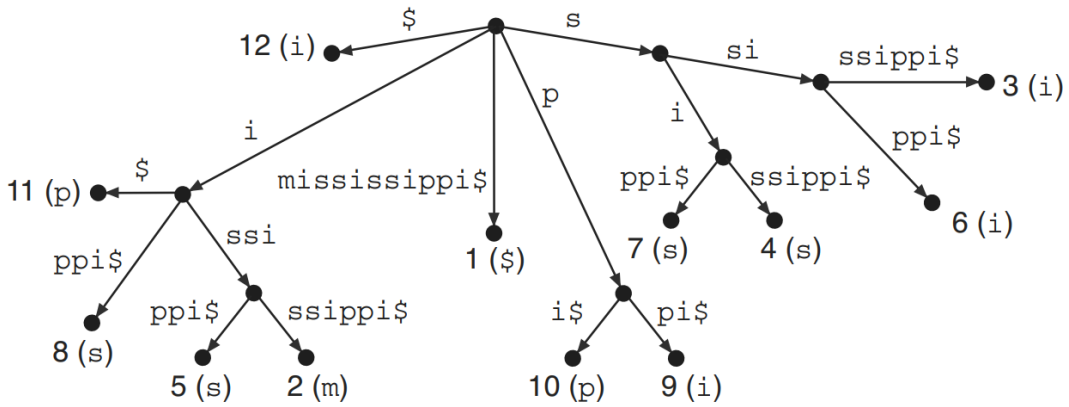

Figure 5: The suffix tree

If we ignore the characters after the special character `$` in the final results from the BWT rotation and permutation procedures, the sorted suffixes (Table 2, third column) correspond exactly to the sorted rotated matrix from the BWT (Table 2, fifth column).

Therefore, given A_T , the suffix array of the input string T, the BWT output can be computed as follows:

$$L[i] = \begin{cases} T[A_T[i] - 1], & A_T[i] \neq 1 \\ T[n+1] = \$, otherwise \end{cases}$$

2.3.1 Construction of a suffix tree

Construction of the suffix tree for a string is not difficult [1, Chap. 4.1.2]. A simple algorithm that accomplishes this task for any given string is given in Algorithm 6.

Algorithm 6 Simple suffix tree construction algorithm

Simple-Suffix-Tree-Algorithm(T)

```
1: Create the root node, with empty string
2: for i ← 1 to n do
3:     Traverse current tree from the root
4:     Match symbols in the edge label one-by-one with symbols in the current
   suffix, T_i
5:     if a mismatch occurs then
6:         Split the edge at the position of mismatch to create a new node, if
   need be
7:         Insert suffix T_i into the suffix tree at the position of mismatch
8:     end if
9: end for
```

The above algorithm, although simple to implement, unfortunately requires construction time that is proportional to the square of n, the length of the string. This $O(n^2)$ complexity may not pose a problem for short sequences with a few characters. However, for most practical applications of suffix trees, such as in whole-genome sequence analysis with input strings that could have billions of characters, more efficient approaches are required. Ukkonen's algorithm [5] is popular mainly because it is easy to understand and implement, and also because it has a relatively small memory footprint. Complete details can be found in the original paper by Ukkonen [5].

Given a string $T = t_1t_2 \ldots t_n$, Ukkonen's algorithm is based on an observation that the suffixes of a string $T^i = t_1t_2 \ldots t_i$ (i.e. the i-th prefix of T) can be obtained from the suffixes of $T^{i-1} = t_1t_2 \ldots t_{i-1}$ by appending t_i at the end of each suffix of T^{i-1}, and by adding the empty suffix. Therefore, the suffix tree of $T = t_1t_2 \ldots t_n = T^n$ can be constructed using a left to right scan, by first building the suffix tree for T^0, the empty string, expanding this to obtain the suffix tree for T^1, and continuing in this way until we build the suffix tree of $T = T^n$ from that of T^{n-1}. This incremental construction and the left-to-right scanning ability

also imply that the algorithm can build the suffix tree piece-by-piece as a new character is received, without having the entire input string available at the beginning.

3 ANALYSIS OF BCOMPRESS

The Burrows-Wheeler Transform does not change the size of the file that has been transformed, but merely rearranges it so that it will be easier to represent it compactly. It then needs to be coded using a second phase which we will refer to as the "Local to Global Transform" (LGT). The local to global transform (LGT) converts the local structure induced by the BWT to a global structure that can be exploited by standard entropy encoding algorithms [1, Chap. 5].
Entropy coding is process in which characters are encoded to bits based on an estimated probability distribution for how likely each character is. The aim is to represent characters in as few bits as possible, and the limit of this is dictated by the entropy. The BCompress often use entropy coding as their final stage. In entropy coding, the representation of a character is based on some estimated probability of that character occurring. The next character to be coded will be drawn from a probability distribution that is typically estimated based on previous observations [1, Chap 3.1].

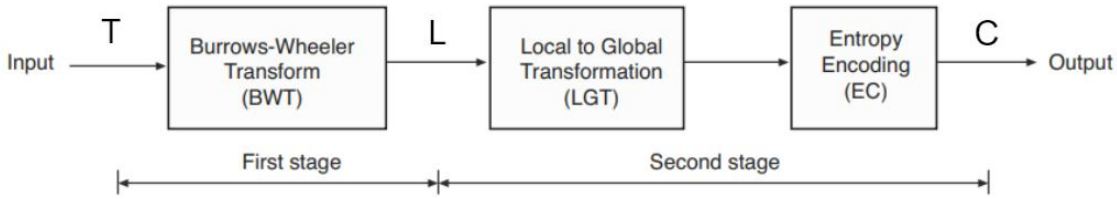

Figure 6: Stages in the BCompress

3.1 Computational complexity

To analyze the overall computational complexity of using the BCompress, we consider the two major stages. The first stage is concerned with the transformation (permutation and sorting) that produces the *L* array (described in Section 2). The second stage concerns the subsequent stage of possibly transforming the local structures in the BWT output into a global structure (for instance, using some recency ranking scheme) and a final entropy encoding using a given variable length code. Thus, stage two corresponds to LGT and EC in the general compression framework. For some applications of the BWT compression such as in pattern matching, only the first stage is needed, while applications such as biological sequence comparison and phylogeny in bioinformatics may not require the entropy encoding stage.
From Section 2, we can observe that given *L*, the reconstruction of the original sequence *T* requires the construction of *F*, the array of first characters, and other count arrays. This can be computed in linear time *O(n)*. Hence, the forward BWT and inverse BWT (BWT first stage) can each be performed in linear time and linear space in the worst case.

3.1.1 BCompress second stage – coding the transformed text

The second stage of BCompress invloves the encoding of the BWT output – the *L* array and the index value, *a* (number of row of A_s containing original string *T*).
Traditionally BCompressors use a "Move to Front" (MTF) list, which essentially ranks characters based on how recently they have occurred. This is done by keeping a list with one entry for each character in the alphabet, and a character is moved to the front of the list each time it is coded, thereby increasing its rank (and decreasing the corresponding number of bits that will be used to code it next time). For our example we will again use `mississippi`. We know from Section 2 that for the Burrows-Wheeler Transform of it, we get the text *L* =`pssmipissii`. The MTF list that is used for coding our example is shown in Table 3, assuming only four characters in the alphabet and starting in alphabetical order Σ = {`i, m, p, s`}. The character at the front (left end) of the list is numbered 0, so the first character to be coded (`p`) has a rank of 2. After it has been coded, it is then moved up to rank 0 (the move-to-front step), which in this case happens to be unfortunate because every other character in our small alphabet will be encoded before the next `p`. Next, the `s` is coded as rank 3, and then moved up to rank 0. This time it works out well, because the next character is also an `s`, and is represented by rank 0. The decoder maintains the same list, and after each character is decoded it makes the same updates to the list, so it always has the up-to-date ranking for each character. [1, Chap. 3.3]

Table 3: The MTF ranks for the characters in the BWT transformed text *L* = `pssmipissii`

MTF list	Rank
`im(p)s`	2
`pim(s)`	3
`(s)pim`	0
`spi(m)`	3
`msp(i)`	3
`ims(p)`	3
`p(i)ms`	1
`ipm(s)`	3
`(s)ipm`	0
`s(i)pm`	1
`(i)spm`	0

Local to global structure transformation (LGT) algorithms [1, Chap. 5.2.2], such as MTF require only a single pass over the BWT output. Consider the *i*-th step in the MTF algorithm.

Let $\sigma = L[i]$, and let the position of σ in the current alphabet list be p_σ. At the *i*-th step, the MTF algorithm records p_σ on the output stream, and then moves σ to the head of the alphabet list. Thus, the algorithm requires $O(n)$ time to process all the characters in the input sequence *L*. The only additional space required is an $O(|\Sigma|)$ space to maintain the alphabets, where Σ is the character alphabet. With a fixed alphabet, the time will be linear with respect to *n*, the length of the input string. Similarly, the entropy encoding stage can be implemented in linear time and linear space for a given input sequence. The decoding process essentially reverses the encoding steps, and subsequent processing of the output using the inverse LGT. Thus, decoding can equally be performed in $O(n)$ time and space in the worst case.

3.2 BWT context clustering property

An important characteristic of the Burrows-Wheeler Transform is that it supports finding characters that have similar contexts in the input string and cluster them together in the output stream [1, Chap. 5.3]. For a given sequence, $T = t_1 t_2 \ldots t_n$, the context of character t_i is defined by the characters that immediately precede t_i in *T*. This is sometimes called the preceding context (or left context), and is defined by the substring $t_{i-1} t_{i-2} \ldots t_1$. Thus, the context of t_i is defined by reversing the prefix T^{i-1}. With the BWT, context clustering is performed based on the succeeding context (also called right context, or forward context), defined by the characters that immediately succeed t_i in *T*. Given the cyclic rotation of the input string by the BWT, the forward context for ti is therefore essentially defined by the string $T_{i+1} * T^{i-1}$, where the character $*$ denotes concatenation. When the special end of string terminator $ is used, the forward context will be determined only by the suffix T_{i+1}.
With the suffix sorting stage of the BWT, suffixes that are similar in the original sequence will be placed together in the sorted matrix of rotations of the BWT.
From the relationship between the BWT and suffix trees and suffix arrays it therefore follows that the BWT output characters in the same region of the output array *L* are likely to have similar following suffixes, that is, similar forward contexts in the original sequence. Thus, the output stream can be partitioned into different segments based on the similarity in the character contexts.

4 EXACT AND APPROXIMATE PATTERN MATCHING

A fundamental operation on strings is determining whether a pattern of characters occurs as a substring in a larger string called the text, or as an approximate subsequence in the text. This problem has been investigated since the early 1960s, not only for its theoretical importance in computer science but because it has many applications in information processing and in particular in biological sciences. In recent years, string matching algorithms have been used as powerful tools in the study of genomics and proteomics, in finding genes and regulatory motifs, and in comparative genomics, gene expression analysis and molecular evolutionary theory.

Pattern matching and data compression are closely related, and at the same time they can work against each other. The process of compression removes redundancies in a text by replacing data with smaller and irregular bit patterns, which unfortunately also destroys the natural structure of the text and makes it harder to search for patterns and retrieve information.

We are focusing on pattern-matching, which involves processing the entire text looking for the pattern. It is also possible to construct various kinds of indexes to support the pattern matching. Particularly if multiple patterns are to be located, in which case the effort used in building the index can be amortized over the multiple searches. Adapting the Burrows-Wheeler Transform for this sort of application is particularly attractive because the compression process automatically provides a sorted index of the text, and thus we have a nice compromise between avoiding doing extra work to create an index, yet we can benefit from such information being available.[1, Chap. 7]

4.1 Exact Pattern Matching

The pattern matching problem is to determine whether pattern P occurs as a substring in text T. If P occurs in T then we also want to report the positions of P in T.

4.1.1 The Knuth-Morris-Pratt Algorithm

The most obvious approach for pattern matching is to compare the given pattern characters $P[1 \ldots m]$ with the first m characters of the text $T[1 \ldots m]$ stored in a buffer, and compare the corresponding pairs of characters. If all character pairs match, the algorithm reports that P has been found in T, and reports the text position where the first pair of characters matched. Whether or not a match is found, the pattern characters are then compared with the characters starting at the next position in T (i.e. $T[2 \ldots m+1]$) to see if every pair matches. This process is iterated $n - m + 1$ times, and then terminates.

Algorithm 7 Brute force pattern matching algorithm

```
Brute-Force-Pattern-Matching(P, T)
 1: i ← 1 /* reference to a character in pattern P */
 2: j ← 1 /* reference to a character in text T */
 3: while (i ≤ m and j ≤ n) do
 4:     if P[i] = T[j] then
 5:         i ← i + 1
 6:         j ← j + 1
 7:     else
 8:         j ← j − i + 2
 9:         i ← 1
10:     end if
11:     if i > m then
12:         report P found in T, beginning at position j − m
13:     end if
14: end while
```

Pseudocode for this method is given in Algorithm 7, and it takes *O(nm)* time.

An example of the worst case is when $T = \mathtt{a}^{n-1}\mathtt{b}$, and $P = \mathtt{a}^{m-1}\mathtt{b}$. For the first $n - m$ attempts the character comparison fails at the last (m-th) character position of the pattern after $m-1$ successful matches.

It arises because the algorithm does not use the information that it has already encountered during the partial match between the pattern and the text. When a mismatch occurs at the j-th input character of the text, the text reference does not necessarily have to be reset back to $j-i+2$ because the algorithm already "knows" the previous $i-1$ characters of the pattern P. More precisely, if we imagine placing the pattern under the text and sliding it right during the matching process, the brute force method blindly slides the pattern by just one character right whenever a mismatch occurs, but it may be possible to slide it further than that. What is the maximum amount of right shift that can be made without missing the occurrence of the pattern in the text? [1, Chap 7.1.1]

To determine the appropriate amount of right shift, we need to define the concept of a border. A border [1, Chap. 7.1.2] of a pattern $P[1 \ldots i]$ is any prefix of $P[1 \ldots i]$ that is equal to a proper suffix of $P[1 \ldots i]$. The longest border for $P[1 \ldots i]$ will be denoted as b_i with a length l_i. The sequence of l_i's for $i = 1$ to n is called the border array. If we add the constraint that $P[i+1] \neq P[l_i + 1]$, then the border will be denoted as B_i and its length will be denoted as Λ_i (Figure 7).

The amount of this shift depends only on the pattern P, and is independent of the text T. We can pre-compute and store in an array the values of L_i for each value of i ($1 \leq i \leq m$) for a given pattern P.

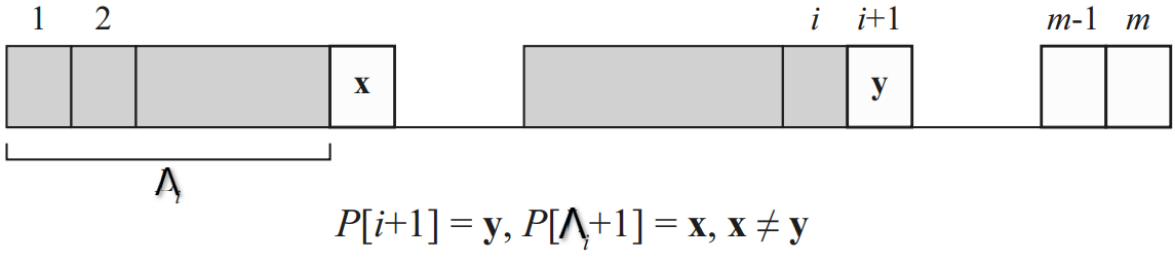

Figure 7: Definition of border B_i with length Λ_i

We formally define the failure function Φ for $1 \leq i \leq m + 1$ as $\Phi[i] = \Lambda_{i-1} + 1$, and define $\Lambda_0 = 0$. There are two special cases: if a mismatch occurs at position $i = 1$, set $\Phi[1] = 1$; and if the entire pattern matches and we are interested in continuing the search process to find all matches in the text, we right shift the text by $(m - L_m)$. This is done by setting $\Phi[m + 1] = \Lambda_m + 1$, given that Φ is precomputed. The computation of the failure function Φ takes $O(m)$ time. The KMP algorithm [6] is given in Algorithm 8.

Algorithm 8 KMP pattern matching algorithm

KMP-String-Matching(P, T)

```
1: pre-process pattern P to compute the failure functions:
2: Φ[j] = Λ_{j−1} + 1 for 1 ≤ j ≤ m + 1
3: p ← 1 /* a reference into the pattern P */
4: k ← 1 /* a reference into the text T */
5: /* The pattern is aligned at the left of the text */
6: while (k + (m − p) ≤ n) do /* T matches P so far */
7:     while P[p] = T[k] and p ≤ m do /* match continues */
8:         k ← k + 1
9:         p ← p + 1
10:    end while
11:    if p = m + 1 then
12:        report pattern P occurs at position k − m in the text T
13:    end if
14:    if p = 1 then
15: /* a special case when mismatch occurs at position 1 of P */
16:        k ← k + 1
17:    end if
18:    p ← Φ[p]
19: end while
```

In the KMP algorithm, when there is a mismatch between a pattern and a text character, the pattern is shifted right as specified by the failure function. But, failures may repeat and in the worst case this may happen $|\Sigma|$ times. Thus, in the worst case the KMP algorithm has complexity $O(m|\Sigma| + n)$. If we assume that $|\Sigma|$ is constant the complexity is $O(m+n)$.

4.1.2 Multiple pattern matching

A generalization of the pattern matching problem is to search for sets of patterns. Given a set of patterns $P = [P_1, P_2, \ldots, P_k]$ and an input text string T of length n, we need to determine whether some pattern P_i occurred as a substring in T. We will assume that the total length of the patterns is m, and the text is longer than the shortest pattern in the set P. We also assume that the patterns in the set are distinct. A straightforward approach to this problem is to apply KMP algorithm for each pattern in the set over the text T. This will take $O(m + kn)$ time in total.

Aho and Corasick [7] solved the multiple pattern matching problem using only $O(n+m)$ time. If the number of patterns k is large, this is a significant improvement. The idea of their algorithm is to generalize the KMP approach using a finite state machine. Aho and Corasick adopted an approach similar to KMP shifts, but a better known formulation using the digital trie, which contains all keywords. (see also Algorithm 1)

The construction of the tree takes $O(m)$ time. This is easy to see: the tree for P_1 is simply a path with edges labeled by the characters of P_1. Suppose we have the tree of the first i patterns in the tree. To add the pattern P_{i+1}, just trace the path using the letters in P_{i+1} as far as possible. At some point a distinct sibling edge has to be added, which will lead to a sub-path terminating in a terminal node representing the pattern. Thus the work involved is bounded by the length of the pattern P_{i+1}. Adding all the pattern lengths gives us m characters, so the total construction time is $O(m)$.

A brute force method to search a text for any of the patterns in the set is to start at each position in the text and trace a unique path from the root, matching text characters with the characters labeling the edges in the path until a failure occurs or a node representing a pattern is reached. For a keyword tree, a node representing a pattern may be reached more than once if some patterns in P are prefixes of other patterns in P. In general, to find all patterns for a given position in the text the tree traversal time is bounded by the length of the longest pattern which could be $O(m)$. Since there are n positions in the text, the brute force algorithm thus takes $O(nm)$ time in the worst case. The Aho-Corasick algorithm generalizes the KMP algorithm for a keyword tree and reduces this time to $O(n + m + \eta_{occ})$, where η_{occ} is the number of occurrences of the patterns in P in T.

String of characters on the path from the root to v will be denoted as $path(v)$. More formally, for the node v, define $lp(v)$ to be the length of the longest proper suffix of $path(v)$ that is a prefix pr_i of some pattern P_i in P. The unique node in the tree that has a path label pr_i will be denoted as $link(v)$. The ordered pair $(v, link(v))$ will be called a failure link. If $lp(v) = 0$, then $link(v)$ will be the root node; in this case it need not be stored, as nodes that do not have an explicit failure link will have an implicit failure link to the root node. Now, suppose we are tracing a text T on the digital tree to find patterns and come to node v after

matching with the character $T[c-1]$ of text, and there is a mismatch with character $T[c]$, where c points to a position in the text. Then, by the definition of the failure link $(v, link(v))$, it is guaranteed that the characters $T[c-lp(v). . . c-1]$ will match the characters of $path(link(v))$. We can then proceed to compare the character $T[c]$ with the next character after $path(link(v))$. This is a generalization of the notion of shifting the pattern in KMP algorithm.

We also need to resolve the situation where one pattern is a substring of another pattern in P. This can be taken care of easily by noting that if there is a failure link or a directed path of failure links from any node v in the tree to a terminal node corresponding to pattern P_i, then whenever the node v is reached during the tree traversal, one can conclude that the pattern P_i has occurred in the text ending at the current text position (c) when v was reached. Thus we can modify our algorithm by stating that if a node v is reached at $T[c]$, and if v is either a terminal node, a link or a directed path of links from v that leads to a terminal node, then a pattern must have occurred in the text ending in position c.

Algorithm 9 Aho-Corasick multiple pattern matching algorithm

Multiple-Pattern-Matching(T, $KeywordTree$)

```
1: c = 1 /*a reference into the text*/
2: v = root /* search begins at root node */
3: repeat
4:     while (v is root node and T[c] does not match with any character labeling outgoing edges (v, v') from v) do
5:         c ← c + 1
6:     end while
7: until character of outgoing edge (v, v') = T[c] or c > n
8: if c > n then
9:     report no pattern found and quit
10: end if
11: repeat
12:     while (T[c] matches the character labeling outgoing edges (v, v') from v) do
13:         if v' corresponds to a pattern P_i or there is a directed path of failure links from v' to a node associated with a pattern P_i then
14:             report P_i occurs in T ending at position c − 1
15:         end if
16:         v ← v'
17:         c ← c + 1
18:     end while
19:     v ← link(v)
20: until c > n
```

4.2 Approximate pattern matching

Until now we have assumed that patterns must match exactly with the text being searched. However, there are many applications where we want to find close matches, rather than require strict equality. Moreover, there are applications where we wish to apply a metric to the difference, and find matches that conform to some specified parameters. The need to consider approximation may arise in the context of errors which might occur in a text file because of inaccuracies in the data stored, or in the pattern being searched for; and it arises in the context of determining how similar two strings are.

Approximate pattern matching algorithms [1, Chap. 7.4] have been used as powerful tools in the study of genomics and proteomics; in finding genes, regulatory motifs, conserved sequences in DNA, sequence alignment and multiple sequence alignments.

4.2.1 Edit distance

Given two strings $S_1 = a_1a_2 \ldots a_m$ and $S_2 = b_1b_2 \ldots b_n$, the problem is to determine how similar are they. This can be made more precise by defining an integer parameter k and a distance function d and then stating the problem as finding all substrings S of S_2 such that $d(S_1, S) \leq k$. Let us first consider how to compute $d(S_1, S_2)$. A natural way to compare these two strings is to determine their edit distance (sometimes called the Levenshtein distance[15]), defined as the minimum number of editing operations that will transform one string to the other string. The two simplest such operations are insertion and deletion of a character, and a cost of 1 is usually associated with each such operation. A third operation is substitution or replacement of a character of one sequence by another character of the second sequence, as it happens in a mutation of a DNA sequence. The cost for this operation is also usually assumed to be 1, but in text editing operations this can be realized by one delete operation followed immediately by an insert operation in which case the cost should then be 2 for this operation. If two characters match, the cost is assumed to be 0. In general, arbitrary cost values can be defined for these operations depending on the application. The sequence of edit operations to transform S_1 to S_2 is called the edit transcript.

A dynamic programming formulation to compute the edit distance is as follows:
Define $d(i, j)$ to be the edit distance between the prefix strings $S_1[1 \ldots i]$ and $S_2[1 \ldots j]$. The basis equations are:

- $d(0, 0) = 0$ because no cost is involved in converting a null string to a null string,
- $d(i, 0) = i$ for $1 \leq i \leq m$ signifying that i deletion operations are needed to convert the prefix of S_1 to a null string, and
- $d(0, j) = j$ for $1 \leq j \leq n$ signifying that j insertion operations are needed to covert a null string to the prefix $S_2[1 \ldots j]$.

We can write the recurrence relation as

- $d(i, j) = min\{d(i - 1, j) + c_I, d(i, j - 1) + c_D, d(i - 1, j - 1) + c_S\}$,

where c_I, c_D, and c_S are costs for insertion, deletion, and substitiution, respectively, and are equal to 1, except for $c_S = 0$ if $a_i = b_j$.

Consider a minimum cost edit transcript for $d(i, j)$. If the last operation of this transcript is an insertion operation in S_2, then this corresponds to the term $d(i, j - 1) + 1$. This is characterized by a horizontal arrow in the $m + 1$ by $n + 1$ matrix M containing $d(i, j)$ values, from cell $M[i, j - 1]$ to cell $M[i, j]$. If the last operation of this transcript is a deletion operation in S_1, then the corresponding term is $d(i - 1, j) + 1$. It can be sybmolized by a vertical arrow in M from cell $M[i - 1, j]$ to cell $M[i, j]$. Match, or a substitution (i.e replacement) of $S_2[j]$ by $S_1[i]$ correspond to the term $d(i - 1, j - 1) + c(i, j)$ in the expression for $d(i, j)$. Both match and replacement operations are represented by a diagonal arrow from cell $M[i - 1, j - 1]$ to cell $M[i, j]$.

To compute the value at any $M[i, j]$, it is sufficient if we know the minimum edit distances of its north, north-west and west neighbors, and the pair of characters from the two sequences under consideration. We know how to compute the 0th row and the 0th column of the matrix from the basis equations defined above, and so we can compute the rest of the matrix one row at a time consecutively with increasing row indexes, or one column at a time consecutively with increasing column indexes. The edit distance of the two strings S_1 and S_2 is given by $d(m, n)$. The time complexity of the algorithm is $O(mn)$ since a matrix of size $(m + 1)(n + 1)$ has to be computed and each entry takes a constant amount of work (three additions, one comparison and a minimum operation). The space complexity is also $O(mn)$. In the case when we are interested only in the distance, not the sequence of operations, this can be reduced to $O(min\{n, m\})$, since we only need to keep information about the last column or last row in order to perform the required computation at any point.

An example illustrating this algorithm is shown in Figure 8 using the sequences S_1=`abccdab` and S_2=`babcabc`. Two possible alignments (corresponding to the squares enclosed by heavy lines and arrows indicating insert, delete or match operations) are also shown. The '-' stands for a gap created in a corresponding sequence. During construction of the table, back pointers to neighboring cells can be kept to trace the paths taken by the minimum edit distance computation. The forward pointers are drawn to show the operations performed.

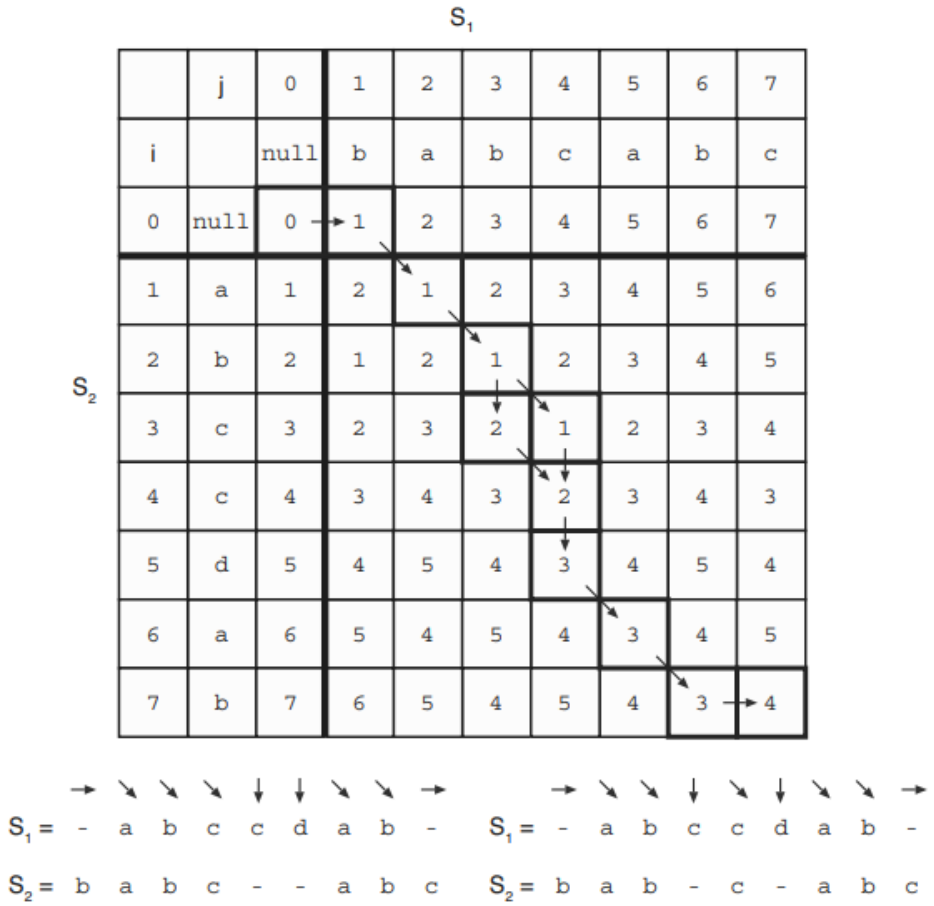

Figure 8: Computation of edit distances

In the context of biological applications, scientists are more interested in finding similarities than the difference between two sequences. This can be cast into another dynamic programming formulation by defining a score or value to each of the edit operations, giving the match operation a high score and giving other edit operations an appropriate low or negative score. The problem then is to compute the maximum score $v(i, j)$ between the two prefixes of S_1 and S_2. The similarity of the two strings is then expressed by the value $v(m, n)$. [1, Chap. 7.4.1]

4.2.2 Local similarity

An important variant of similarity search is local alignment or local similarity. Suppose we have two long DNA sequences in which there is a particularly interesting subsequence representing a gene that is common between the sequences. Doing a global alignment or similarity search will not be able to identify this because there may be a lot of dissimilarities in the rest of the sequence which yield a low value for similarity and a large edit distance, neither of which say anything about this interesting region. If the regions of highly similar local alignment are small, they can get lost in the context of global

alignment. The dynamic programming equations are very similar to the edit distance computation and similarity value $v(i, j)$ is given as:

$$v(i, j) = max\{\ 0,\ v(i-1, j) + c_I,\ v(i, j-1) + c_D,,$$
$$v(i-1, j-1) + c_S,\}$$

where c_I, c_D, and c_S,are the values assigned to the insertion, deletion and substitution operations, respectively. [1, Chap. 7.4.3]

5 APPLICATIONS OF BURROWS-WHEELER TRANSFORM IN BIOINFORMATICS AND COMPUTATIONAL BIOLOGY

The major application of the Burrows-Wheeler Transform has been for data compression and efficient storage. However, various researches on the Burrows-Wheeler Transform have shown the versatility of the BWT, and hence efforts are shifting from its traditional application in data compression to other areas of study.

Given the availability of complete genomes of various organisms, a major challenge is how to make some sense out of the growing mass of data. Computational methods have been brought to bear on this problem, and different algorithms have been proposed for various problems. One major characteristic of problems in this area is the huge size of data often involved. The human genome, for instance, contains about 3 billion base pairs, and there are organisms with genomes that are orders of magnitude larger. A suffix array would require 12 gigabytes of storage for the human genome, while the suffix tree may take as much as 5 times this amount. Developing efficient search algorithms for patterns of various forms in the genomic sequence thus represents an important problem.

Another important characteristic of genomic sequences is the relatively large amount of repetition often observed in such data. Hence, identification, grouping, and effective exploitation of the various types of repetition found in biological sequences is another challenge. Currently, most applications of the BWT in bioinformatics and computational biology mainly exploit the context clustering ability of the BWT[1, Chap 8.3]. For the rest of the chater we will look at analysis, compression, annotation, and pattern matching of DNA sequences that use BWT.

5.1 DNA sequence compression

Given the large data sizes involved in biological sequences, one way to deal with the increasing data sizes is by compressing the sequence (see Figure 5 and Section 3). The major problem is that these algorithms deal with the data as merely a sequence of characters, without exploiting the special nature of such sequences. In Adjeroh et al. (2002) [8], two methods were proposed for DNA sequence compression, based on the BWT. The basic idea was to exploit the different repetition structures observed in DNA sequences to compress them. Thus, repetition analysis was performed on the sequence based on the relationship between the BWT and suffix trees and suffix arrays. They proposed two vocabulary parsing schemes that use a repetition code for repeat types, to parse the input sequence. Here, vocabulary refers to the ensemble of repeat structures without reference to their specific locations in the sequence. In one scheme, each repeated substring is removed from the input sequence and moved to an external dictionary. The positions in the sequence where each repetition occurred, along with the corresponding repetition code, are

recorded in the dictionary. Thus there is no reference or pointer information in the original sequence.
Results in the paper showed that the introduction of repetition analysis and parsing in the BWT compression pipeline generally improves the compression result. In essence, the analysis and parsing stage further exposed the hidden regularities in the DNA sequence (such as reverse complements), which typically will not be discovered by compression algorithms. [1, Chap. 8.3.1]

5.2 Building BWTs for big databases using prefix free-parsing

High-throughput sequencing technologies have led to the growth of genomic databases. For many applications we want to build and store indexes of these databases but constructing such indexes is a challenge. Fortunately, many of these genomic databases are highly repetitive – a characteristic that can be exploited and enable the computation of the Burrows-Wheeler Transform (BWT). In the paper Boucher et al. (2019) [3], introduced a preprocessing algorithm, referred to as prefix-free parsing, to ease the computation of Burrows-Wheeler Transforms (BWTs) of genomic databases. Given a string S, it produces a dictionary D and a parse P of overlapping phrases such that BWT of S can be computed from D and P in time and workspace bounded in terms of their combined size $|\mathrm{PFP}(S)|$.
Their experiments show that D and P are significantly smaller than S in practice, and thus, can fit in a reasonable internal memory even when T is very large. Therefore, prefix-free parsing eases BWT construction, which is pertinent to many bioinformatics applications.
In the other paper[4], they considered PFP(S) as a data structure and showed how it can be augmented to support various queries, including BWT, quickly, still in $O(|\mathrm{PFP}(S)|)$ space.

5.3 Analysis of repetition structures and genome annotation

Repetition structures represent an important characteristic of genomic sequences. Long runs of tandem repeats and of randomly interspersed repeats are prominent features of DNA sequences. The family of Alu repeats (usually about 350 bases in length) is typical of short interspersed repeat sequences, referred to as SINEs — short interspersed nuclear elements[9]. These have been estimated to make up about 11% of the human genome[16]. There are also the long interspersed repeat sequences (LINEs — long interspersed nuclear elements) which are usually more than 6000 bases in length. In the human genome, the L1 family is the most common LINEs, with about 60,000 to 100,000 occurrences. There are also short repeats (sometimes called “random repeats”), attributed to the fact that typical sequences and genomes are orders of magnitude larger than the alphabet size (4 in this case).

It is known that the redundancy due to the repetition structures provides some form of stability for the genome. Tandem repeats in particular play a major role in various regulation mechanisms in the genome, such as in protein binding. Repetition structures have been implicated in various diseases and genetic disorders. For instance, the triplet repeats $(CTG)_n/(CAG)_n$ have been associated with Huntington's disease, while the hairpins formed in $(CGG)_n/(CCG)_n$ repeats have been linked to the Fragile-X mental retardation syndrome [10]. An important observation for computational analysis of such repetition structures is that, in every single case listed, the susceptibility to (or incidence of) the disease critically depends on the number of copies (that is, the copy exponents in the repeat), and how many times the triple repeat occurs with a given exponent. [1, Chap. 8.3.2]

Another issue in the analysis of genomic sequences is determining changes in the copy number of certain important repeating elements over time, perhaps in response to a drug or environmental changes. A special case of this problem is in detecting changes in the gene copy number between a normal genome and a mutant genome [11]. For such analysis, substrings in the genome sequence can be viewed as a word, and the major problem becomes that of performing a word count over the genome. When the word length is small, or we have only one or two words, the problem is easy and could be solved using direct methods. [1, Chap. 8.3.2] However, with increasing word lengths, or an increasing number of words, improved data structures and algorithms are needed. Using the BWT and related data structures, Healy et al. (2003) [12] developed a method for annotating any sequence, including the entire human genome, with the counts of its constituent words. Thus, the problem is turned into reporting counting queries for each pattern.

At each position along the genome, annotation is performed in terms of the number of occurrences of the q-mer at this position in both the forward and the reverse directions, for different values of q. The result is a visualization of the annotation "terrain" along the entire genome, which provides a quick view of the structure of repeats within a localized region along the genome.

5.4 Distance measure between sequences and phylogeny

Mantaci et al. (2005, 2007) [17, 18] proposed an extension of the BWT that uses an ordering different from the lexicographic ordering. The new ordering allows the BWT to be extended to handle a multiset of strings (rather than just a single string which is a multiset of characters). In the same set of papers they showed how the extended BWT can be used to define a distance measure between sequences. Such a distance measure can thus be used to cluster species based on their similarity with respect to this measure. Given two input sequences, say S and T, the number of segments shared by S and T could be used as a measure of their similarity over evolution. Given the extended BWT, the extent to which

the two sequences share some segments can be captured by considering the extent of mixing between symbols from the two sequences in the output array after the transformation. Thus, even when large but similar segments of two genomes are shuffled within each genome, the distance measure can still capture their potential relatedness. This is important in other applications such as in the analysis of genome rearrangements. The distance measure is therefore simply given by the number of alternations between symbols from each sequence in the output of the extended BWT. Based on this Mantaci et al. (2007) constructed phylogenic trees between different species based on their mitochondria DNA. [1, Chap. 8.3.5]

6 CONCLUSION

Thoroughout this final project paper we have studied the BWT, one of the last 20th century breakthroughs in general-purpose lossless compression methods.

The transform is closely related to suffix tree, and suffix array. It can be implemented using suffix array and thus achiving linear time complexity. The remarkable thing about BWT is not that it generates more easily encoded output, but that it does this reversibly, allowing the original document to be re-generated.

Various researchers have shown that versatility of BWT, and shifted its traditional application in data compression to other areas of study. In this thesis we explore the view of transformed file as both text and an index, and look at some of the applications that exploit this.

We observed, that characters in the transformed text are clustered together. From the relatioship between BWT and suffix array and suffix tree, it follows that output text can be partitioned into different segments based on the similarity in the character context.

BWT transform has a promising future, as it is being applied in an environment of new data structures, more powerful computers with new models of computation, increasing amounts of data to be processed for storage, and pattern matching, and new theory to help us better understand how we can exploit a powerful technique that is based on simply muddling up the contents of a file.

7 DALJŠI POVZETEK V SLOVENSKEM JEZIKU

V diplomski nalogi smo najprej opisali Burrows-Whelerjevo preslikavo (BWT) kot enega zadnjih dosežkov v 20. stoletju v splošnem stiskanju brez izgub. BWT uporablja zamisel o permutiranju črk v besedilu, kar olajša izgradnjo kompaktne predstavitve in druge vrste obdelave. Za analizo uporabljamo primerjalni oziroma model RAM. Za zapis rezultatov uporabljamo asimptotičen zapis, zlasti *O*-zapis.

Nato preučimo, kako se BWT izvaja v praksi. Transformacija naprej uredi rotacije vhodnega niza kar ima za posledico združevanje znakov ki se pojavljajo v podobnih kontekstih. Obratna BWT omogoča ponovno izdelavo izvornega besedila iz permutiranega besedila BWT. Obratno transformacijo je nekoliko težje izvesti kot transformacijo naprej. BWT je zelo tesno povezana s priponskim drevesom in priponskim poljem. Problem razvrščanja rotiranih matrik je glavna težava pri izvedbi preslikave.

Pri analizi splošne računske zahtevnosti stiskanja upoštevamo dve glavni fazi. Prva faza je BWT preoblikovanje. Druga faza se nanaša na naslednjo stopnjo morebitnega preoblikovanja lokalnih struktur v izhodu BWT v globalno strukturo in končno entropijsko kodiranje z uporabo kode spremenljive dolžine. BWT naprej in obratna BWT (prva faza BCompress) se lahko izvedeta v linearnem času in linearnem prostoru v najslabšem primeru. LGT pretvori lokalno strukturo, ki jo ustvari BWT, v globalno strukturo, ki jo lahko uporabe algoritmi za entropijsko stiskanje. Entropijsko kodiranje je proces, v katerem je mogoče simbole kodirati na podlagi ocenjene porazdelitve verjetnosti, kako verjetni so posamezni simboli. Algoritmi za lokalno preoblikovanje globalne strukture (LGT) zahtevajo le en prehod preko izhoda BWT. Tako je lahko dekodiranje v najslabšem primeru prav tako izvedeno v $O(n)$ času in prostoru.

Problem ujemanja vzorcev je ugotoviti, ali se vzorec *P* pojavlja kot podniz v besedilu *T*. Eden najpomembnejših algoritmov za iskanje vzorcev je algoritem Knuth-Morris-Prath (KMP). Deluje na osnovi opažanja, da, ko pride do neujemanja, smo zbrali dovolj informacij, da bolje določimo, kje bi se lahko začelo naslednje ujemanje. Na ta način se izognemo ponovnemu preverjanju predhodno ujemajočih se znakov. Ujemanje več vzorcev je posplošitev problema ujemanja enega vzorca. Problem ujemanja več vzorcev sta rešila Aho in Corasick s pristopom, podobnim KMP in z uporabo številskega drevesa. Omenjamo tudi približno ujemanje vzorcev, kar je koristno, če bi radi našli približna ujemanja, namesto popolne enakosti.

Glede na razpoložljivost popolnih genomov različnih organizmov je velik izziv, kako iz vse večje množice podatkov narediti nekaj smislenega. Druga pomembna značilnost genomskih zaporedij je razmeroma velika stopnja ponovitev. Večina aplikacij BWT v bioinformatiki in računalniški biologiji izkorišča predvsem sposobnost kontekstnog bručenja BWT.

8 REFERENCES

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

[2] Burrows, Michael, and Wheeler, David. "A block-sorting lossless data compression algorithm." *Digital SRC Research Report*. 1994.

[3] Boucher, Christina, et al. "Prefix-free parsing for building big BWTs." *Algorithms for Molecular Biology* 14.1 (2019): 1-15.

[4] Boucher, Christina, et. al. ”PFP Data Structures.” *arXiv preprint arXiv:2006.11687* (2020).

[5] Ukkonen, Esko. "On-line construction of suffix trees." *Algorithmica* 14.3 (1995): 249-260.

[6] Knuth, Donald E., James H. Morris, Jr, and Vaughan R. Pratt. "Fast pattern matching in strings." *SIAM journal on computing* 6.2 (1977): 323-350.

[7] Aho, Alfred V., and Margaret J. Corasick. "Efficient string matching: an aid to bibliographic search." *Communications of the ACM* 18.6 (1975): 333-340.

[8] Adjeroh, Don, et al. "DNA sequence compression using the Burrows-Wheeler Transform." *Proceedings. IEEE Computer Society Bioinformatics Conference.* IEEE, 2002.

[9] Kramerov, Dimitri A., and Nikita S. Vassetzky. "Short retroposons in eukaryotic genomes." *International review of cytology* 247 (2005): 165-221.

[10] Lubin, Flora, et al. "Nutritional and lifestyle habits and water-fiber interaction in colorectal adenoma etiology." *Cancer Epidemiology and Prevention Biomarkers* 6.2 (1997): 79-85.

[11] Lucito, Robert, et al. "Representational oligonucleotide microarray analysis: a high-resolution method to detect genome copy number variation." *Genome research* 13.10 (2003): 2291-2305.

[12] Healy, John, et al. "Annotating large genomes with exact word matches." *Genome research* 13.10 (2003): 2306-231

[13] Smith, Temple F., and Michael S. Waterman. "Identification of common molecular subsequences." *Journal of molecular biology* 147.1 (1981): 195-197.

[14] Cormen, Thomas H., et al. *Introduction to algorithms, Third Edition*. 3rd. MIT Press, 2009

[15] Levenshtein, Vladimir I. "Binary codes capable of correcting deletions, insertions, and reversals." *Soviet physics doklady*. Vol. 10. No. 8. 1966.

[16] Bai, Xue, Feifei Li, and Zhihua Zhang. "A hypothetical model of trans-acting R-loops-mediated promoter-enhancer interactions by Alu elements." *Journal of Genetics and Genomics* (2021).

[17] Mantaci, Sabrina, et al. "An extension of the Burrows–Wheeler transform." *Theoretical Computer Science* 387.3 (2007): 298-312.

[18] Mantaci, Sabrina, et al. "An extension of the Burrows Wheeler transform and applications to sequence comparison and data compression." *Annual Symposium on Combinatorial Pattern Matching.* Springer, Berlin, Heidelberg, 2005