Incompressible String in Burrows-Wheeler Transform

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

Master of Technology

by
Kratika Jain
Roll No.: 13111027

under the guidance of
Prof. Satyadev Nandkumar

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

September, 2015

CERTIFICATE

It is certified that the work contained in this thesis entitled "*Incompressible String in Burrows-Wheeler Transform*" by Kratika Jain*(Roll No. 13111027)*, has been carried out under my supervision and this work has not been submitted elsewhere for a degree.

(Prof. Satyadev Nandkumar)
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur
Kanpur-208016

June, 2015

Abstract

Burrows-Wheeler Transform is a useful tool for many loss-less data compression applications. It permutes the string S in such a way that there are runs of similar characters in the output string BWT(S), which can be compressed better than original string S. For any length n and Σ different characters, there are Σ^n different strings possible. Since Burrows-Wheeler Transform is reversible, permutation function from S to BWT(S) has to be one-to-one.

Move-to-front transform replaces each character in the string S by its index of "recently used characters". So the long sequence of similar characters are replaced by zeros while the characters that appears after long time, replaced by larger value. MTF value of a string is sum of these indices. Since Burrows-Wheeler Transform tries to make runs of similar characters, MTF value of output string is smaller.

In our thesis we propose a polynomial time algorithm to pick a string S from all the different Σ^n strings whose Burrows-Wheeler Transform, BWT(S), has close to maximum MTF value that is possible for the length n. Such strings are in-compressible for Burrows-Wheeler Transform. The experiments we conducted indicate that it performs really well in practice. For 26 different characters and length up-to 50k the average over the ratio of actual MTF value and expected MTF value come out to be 0.999702934.

Acknowledgement

I would like to express my deepest gratitude to my supervisor Prof. Satyadev Nandkumar. I am fortunate to have an advisor who gave me freedom to explore new ideas and at the same time the guidance to recover the steps where my steps faltered. His valuable guidance, motivation and support helped me to overcome crisis situations and finish this M. Tech thesis. I take this opportunity to express gratitude to all of the Department faculty members and staff for their help and support.

I would like to thank my friends Ritika, Nidhi, Richa and Satyajit for their continuous motivation and support. I greatly value their friendship and I deeply appreciate their belief in me.

Lastly, I would like to thank my parents for their support, encouragement and belief, My sister Ankita Jain for her support in my tough times. This thesis would have not been possible without the love and patience of my family.

Kratika Jain

Contents

List of Figures

Chapter 1

Introduction

Data compression has become a very useful stream in the field of computer science. In Data compression we digitally store information on computers and transmit it over networks. Resources such as data storage space and transmission capacity are used wisely with the help of data compression. Data compression could be lossy or loss-less. In lossy compression, as some information is lost in the process of compressing data, it is not possible to gather back original data. While in loss-less compression, we can get back the original data, as no information is lost while compressing data. Depending on the data, loss-less or lossy compression can be applied. For the data such as images or videos where loss of data is acceptable, lossy compression can be applied. But for the text data, where each and every character is important, data loss is not acceptable. Burrows Wheeler Transform (BWT)[1], for text, helps transforming a message into a string which is more likely to be compress. Burrows Wheeler Transform (BWT) reorders the characters in the message to maximize the runs of repeated characters, and still recovering of original message is possible. it relies on the intuition, that most of the time the preceding character will be 't' or 'w' if the letter hen is found in English text. If grouping of all such preceding letters (t's and w's) is possible, then the data is more likely to be highly compressible. This transformation was originally discovered by David Wheeler in 1983, and was published by Michael Burrows and David Wheeler in 1994.[1]

1.1 Motivation

There are many different algorithms for loss-less data compression. Each algorithm has its own set of strings which are incompressible by that algorithm. Lempel–Ziv is an online compression algorithm and Champernowne sequence is an infinite sequence of 0's and 1's which is in-compressible by Lempel–Ziv.

As Burrows-Wheeler Transform, transforms a given string into such string which is more likely to be compressed and also it is a loss-less transformation, therefore there are strings whose transforms are incompressible. There are no such known explicit constructions for Burrows-Wheeler Transform. Since Burrows-Wheeler Transform is a block algorithm, the motivation was to find incompressible string, of any length n and in polynomial time, for this transform.

1.2 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 gives the brief description of related work. In Chapter 3 we have described all the necessary groundwork for the algorithm. Chapter 4 contains the algorithm. Chapter 5 presents the experimental evaluation. Future work is provided in Chapter 6.

Chapter 2

Related work

In this chapter we have described the algorithms for Burrows-Wheeler Transform, Reverse Burrows-Wheeler Transform and Move-to-front transform.

2.1 Transformation of a message S to BWT(S)

We now describe the Burrows-Wheeler Transform.[1]
Let 'S' be the string we would like to transform. We append a special character $ at the end of the string that does not appear elsewhere in S and lexicographically prior to all characters in S. Let n be the length of the string i.e n=|S|.

Algorithm BWT(S) :

- Append $ sentinel character at the end of the string.
- Take all rotations of S and create a matrix A.
- Sort the matrix A Lexicographically.
- return the last character of every rotation i.e last column of A.

For example original message= banana
S=banana$. Now Take all rotation of S

b a n a n a $

$ b a n a n a

a $ b a n a n

n a $ b a n a

a n a $ b a n

n a n a $ b a

a n a n a $ b

Now sort them lexicographically,

$ b a n a n a

a $ b a n a n

a n a $ b a n

a n a n a $ b

b a n a n a $

n a $ b a n a

n a n a $ b a

So the BWT(S)= annb$aa

2.2 Retrieving message S from BWT(S)

We now describe the inverse transform.[2]

The cyclic shift of the rows of M is crucial to define the inverse BWT, which is based on two easy to prove observations :

- Given the i^{th} row of A , its last character L[i] precedes its first character F[i] in the original text S , namely S = . * * L[i]F[i]* *.
- Let L[i] = c and let r_i be the number of occurrences of c in the prefix of the last column L[1,i] . Let A [j] be the r_i^{th} row of A starting with c. The character

in the first column F corresponding to L[i]is located at F [j] .We call this LF-mapping (Last-to-First mapping) and set LF[i] = j

Algorithm :

1. Compute the array Count[1 . . Σ] storing in Count[i] the number of occurrences in string S of the characters {$, 1 , . . . ,Σ- 1). It is clear that the first occurrence of char ch in F will be at Count[ch]+ 1 position. (if any).

2. Define the LF-mapping LF[l . .. n+1] as follows:

   LF[i]= Count[L[i]] r_i , where r_i equals the number of times character L[i] occur up in L to the index i. (see observation (b) above).

3. Reconstruct S backward as follows:
   set s = 1 and S [n] = L[1](because A [1] = $ S) ; then, for each i = n - 1 , . . . , l do s = LF[s]and S[i] L[s].

2.3 Move to Front Algorithm

We describe the Move-To-Front transform which is used to compress BWT(S).[3] Move-to-front (MTF) is a reversible encoding technique in which each character in the data is replaced by its index in the list of 'recently used character'. So in the end encoded data will be the sequence of integers. In the list, symbols that recently appeared in the data will have the lower index. Since the BWT, transforms the message into runs of similar characters we apply MTF encoding before compressing the data finally. The algorithm was published in a paper [4] by Ryabko.

For example if we have lexicographically sorted list of english alphabet, which we use as a list of 'recently used character' then MTF encoding of massage 'mississippi' is as follows :

Iteration	**Encoding**	**Recently used character**
Mississippi	12	abcdefghijklmnopqrstuvwxyz
mIssisippi	12,9	mabcdefghijklnopqrstuvwxyz
miSsisippi	12,9,18	imabcdefghjklnopqrstuvwxyz
misSisippi	12,9,18,0	simabcdefghjklnopqrtuvwxyz
missIsippi	12,9,18,0,1	simabcdefghjklnopqrtuvwxyz
missiSsippi	12,9,18,0,1,1	ismabcdefghjklnopqrtuvwxyz
missisSippi	12,9,18,0,1,1,0	simabcdefghjklnopqrtuvwxyz
mississIppi	12,9,18,0,1,1,0,1	simabcdefghjklnopqrtuvwxyz
mississiPpi	12,9,18,0,1,1,0,1,16	ismabcdefghjklnopqrtuvwxyz
mississipPi	12,9,18,0,1,1,0,1,16,0	pismabcdefghjklnoqrtuvwxyz
mississippI	12,9,18,0,1,1,0,1,16,0,1	pismabcdefghjklnoqrtuvwxyz
mississippi	12,9,18,0,1,1,0,1,16,0,1	ipsmabcdefghjklnoqrtuvwxyz

It is easy to see that transform is reversible. Start with the lexicographically sorted list of English alphabet and in each iteration, replace the index with the character at that index in the list and update the list.

Chapter 3

Preliminaries

In this chapter we first state the problem statement, then discuss all the necessary ground work to formulate the algorithm.

3.1 Problem Statement

For Burrows-Wheeler Transform, given any length n, construct an incompressible string of that length.

3.2 Move-To-Front Pattern

The MTF value of a string is maximum if every character to be moved to the front is always selected from the end of the alphabet array and if string length is much longer than the number of alphabet, then same pattern will repeat again over the string. BWT algorithm tries to make a run of similar characters so as to reduce the MTF value.

For example if we have alphabet array a,b,c, we will assign value 1 to a, 2 to b and 3 to c. To generate string of higher MTF value of length 9, we will move last character to front and assign new values to each character of alphabet array c=1,a=2,b=3. In this case pattern would be "cba" and generated string of length 9 will be cbacbacba. The MTF value of above string is 27 and no other string will have higher MTF value.

In our algorithm, based on this pattern we generate the string. But the generated string might not always be valid BWT output of any string. For example the above string of length 9 ,with a sentinel character $, 'cbacbacba$' has the highest MTF value for that length but it is not the valid BWT of any string.

So we try to get valid BWT string from generated string with minimum possible changes in pattern, which ensures close to maximum MTF value.

3.3 Rules of string permutation in BWT

In this section, we give necessary and sufficient condition to determine whether a string S is BWT of some string.

Permutation of a string is the rearrangement of the elements of the string. As a function mapping, each element listed in the left column mapped into a single element from the same set listed at the right. To be a permutation, the mapping should be one-to-one and onto.

Now each permutation σ of a set A, partition the set into equivalence classes with the following relation,

$$\forall a, b \in A, \text{ a related to b iff } b = \sigma^n(a) \text{ for some } n \in Z$$

Equivalence classes determined by the above equivalence relation are the orbits of σ. Let S' be some string and S be the first column of all sorted rotations of S'; and σ be the some permutation of S. BWT permutes a string in such a way that there are runs of similar characters in the output string. To make sure this permutation is reversible, we have following lemmas.

Lemma 1 : σ can be a valid BWT of some string S' if σ has only one cycle.

Proof : Any character in S is taken from the some rotation of the string. Let the first character in the row in which 'ch' occurs in the last column, be denoted by 'f'. Then 'ch' immediately precedes 'f' in the original string. So if we have the valid BWT and we start with 1st character of S i.e sentinel character, and take its preceding

character and then go to that character in S and take its preceding character and continue like that, we will get the original string in n steps.

If there are no repeated characters in the string we can find the preceding character unambiguously. But if there are repeated characters, then we need to find which occurrence of such a character precedes our chosen character. For this we need the next lemma.

Lemma 2 : The i^{th} occurrence of character c in last column is the same text character as the i^{th} occurrence of c in the first column.

Proof : Take some string with repetitive characters where number in the subscript part represent the i^{th} occurrence of character in string. Lets call it the rank of the character.

String $S = a_0b_0a_1c_0b_1a_2c_1a_3\$$

Now take a matrix A, rows as all the string rotation of S and sort them lexicographically. In our case A will be

$$\begin{array}{cccccccccc}
\$ & a_0 & b_0 & a_1 & c_0 & b_1 & a_2 & c_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & c_0 & b_1 & a_2 & c_1 \\
a_0 & b_0 & a_1 & c_0 & b_1 & a_2 & c_1 & a_3 & \$ \\
a_2 & c_1 & a_3 & \$ & a_0 & b_0 & a_1 & c_0 & b_1 \\
a_1 & c_0 & b_1 & a_2 & c_1 & a_3 & \$ & a_0 & b_0 \\
b_1 & a_2 & c_1 & a_3 & \$ & a_0 & b_0 & a_1 & c_0 \\
b_0 & a_1 & c_0 & b_1 & a_2 & c_1 & a_3 & \$ & a_0 \\
c_1 & a_3 & \$ & a_0 & b_0 & a_1 & c_0 & b_1 & a_2 \\
c_0 & b_1 & a_2 & c_1 & a_3 & \$ & a_0 & b_0 & a_1
\end{array}$$

Take a new matrix A' by rotating every row of A one more time. In our case A' will be,

$$
\begin{array}{ccccccccc}
a_3 & \$ & a_0 & b_0 & a_1 & c_0 & b_1 & a_2 & c_1 \\
c_1 & a_3 & \$ & a_0 & b_0 & a_1 & c_0 & b_1 & a_2 \\
\$ & a_0 & b_0 & a_1 & c_0 & b_1 & a_2 & c_1 & a_3 \\
b_1 & a_2 & c_1 & a_3 & \$ & a_0 & b_0 & a_1 & c_0 \\
b_0 & a_1 & c_0 & b_1 & a_2 & c_1 & a_3 & \$ & a_0 \\
c_0 & b_1 & a_2 & c_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & c_0 & b_1 & a_2 & c_1 & a_3 & \$ \\
a_2 & c_1 & a_3 & \$ & a_0 & b_0 & a_1 & c_0 & b_1 \\
a_1 & c_0 & b_1 & a_2 & c_1 & a_3 & \$ & a_0 & b_0
\end{array}
$$

Now pick any character from a,b,c and compare its rank in A to A'. The rank comes in same order in both the matrix. It is because in A', rows are bounded with respect to the first position and sorted starting at the second position. And as the last column of A is equal to the first column on A', this lemma holds.

Example :

$$S' = baca\$ \; S = \$aabc(12345)$$

$$\sigma = c\$baa(51432)$$

$$1 \to 5 \to 2 \to 1 \text{ and } 3 \to 4 \to 3$$

we have 2 orbits so σ can not be BWT of any string. (Although σ is a derangement.) Now let,

$$\sigma = acba\$(35421)$$

$$1 \to 3 \to 4 \to 2 \to 5 \to 1$$

Even though σ has only one orbit it can not be the last column of S or BWT of S' because rank of character 'a' is different in S' and σ. (described below in Lemma 2)

Chapter 4

Algorithm

In this chapter we will describe the algorithm to generate a 'n' length incompressible string for Burrows-Wheeler Transform.

4.1 Description of the Algorithm :

1. Algorithm takes n and Σ as a input , where n is length of the string to be generated and Σ is number of distinct alphabet excluding sentinel character. It outputs the String of length n having close to maximum MTF of length n.

2. In the next step, with the pattern $\{\Sigma, \Sigma-1, \Sigma-2, .., 1\}$ we generate new string of length n which helps us to find the position of alphabet for the output string. This is done in algorithm 4.2.1 .

3. Because of the pattern , we know that every alphabet will occur at least $\frac{n}{\Sigma}$ times. And last (Alphabetically larger) $n \ mod \ \Sigma$ alphabet will occur one more time.

4. We take an array *position* with $|\Sigma|$ many entries and the entries of this array indicates the first occurrence position that different alphabets will get when all string characters are sorted lexicographically. Entries of this array at index i and i+1 will be the range of i^{th} alphabet in the first column. This is done in algorithm 4.2.2 .

5. Now, in our generated string of step 2, from left to right, at the place of Σ we assign it the minimum available value (the value which is not assigned before) from the range of Σ^{th} alphabet and at the place of $\Sigma - 1$ we assign it the minimum available value from the range of $\Sigma - 1^{th}$ alphabet and so on. This is done in algorithm 4.2.3 .

6. In the next step, starting from the index $i = 1$ we go to char_arr[i] and assign char_arr[i] to i, until we come back to $i = 1$. If the process did not continue for the length n that means we do not have one orbit as explained in lemma 1 of the previous chapter. So we make the necessary swap until we get the only one orbit. This is done in algorithm 4.2.4

7. In our generated string at each index we have an integer entry. Each entry lies in the range of some alphabet x, replace the entry with the alphabet x. output string will be the inverse BWT of our generated string. This is done in algorithm 4.2.5

4.2 Algorithm :

Input : n : length of string, Σ : No of distinct characters excluding \$.

Output : text: String of characters.

Algorithm 4.2.1

Require:
position : Array of length n

```
1: for i = 1 to Σ + 1 do
2:     position[i] ← Σ + 1 − i
3: end for
4: for i = Σ + 2 to n do
5:     for j = Σ to 1 do
6:         position[i] ← j
7:         i ← i + 1
8:     end for
9: end for
```

Algorithm 4.2.2

Require:
count : Array of length Σ

```
1:  C_1 ← (n−1)/Σ
2:  C_2 ← n − 1 mod Σ
3:  T_1 ← 1
4:  count[0] ← 1
5:  count[1] ← 2
6:  if C_2 == 0 then
7:      for i = 2 to Σ do
8:          count[i] ← 2 + C_1 * (i − 1)
9:      end for
10: else
11:     for i = 2 to Σ − C_2 + 1 do
12:         count[i] ← 2 + C_1 * (i − 1)
13:     end for
14:     if C_2 > 1 then
15:         for i = Σ − C_2 + 2 to Σ do
16:             count[i] ← 2 + C_1 * (i − 1) + T_1
17:             T_1 ← T_1 + 1
18:         end for
19:     end if
20: end if
```

Algorithm 4.2.3

Require:
char_arr: Array of length n
count_copy: copy of count array

```
1:  last_pos ← position[n]
2:  count_last_pos ← count[last_pos]
3: for i = 1 to n do
4:     char_arr[i] ← count[position[i]]
5:     count[position[i]] ← count[position[i]] + 1
6:     if i == char_arr[i] then
7:         if i < n then
8:             swap char_arr[i] and char_arr[i − 1]
9:             swap position[i] and position[i − 1]
10:            for j = i + 1 to n do
11:                if position[j−1] ≡ position[i] & position[j] ≡ position[i−1] then
12:                    swap position[j − 1] and position[j]
13:                end if
14:            end for
15:        else if i == n then
16:            char_arr[i] = count_last_pos
17:            for j = 1 to n do
18:                if char_arr[i] > count_last_pos then
19:                    char_arr[i] = char_arr[i] + 1
20:                end if
21:            end for
22:        end if
23:    end if
24: end for
```

Algorithm 4.2.4

Require:
$mark$: Boolean array of length n

1: Initialize $mark$ array with false entries
2: $cycle\_length \leftarrow 1$
3: $cycle\_pos \leftarrow 1$
4: **for** $i = 1$ to n **do**
5: **if** $char\_arr[cycle\_pos] \equiv 1$ && $i < n$ **then**
6: $found$ =find first unmarked positions till sigma entries to the right of cycle_pos
7: **if** $found \equiv -1$ **then**
8: find first unmarked positions till Σ entries to the left of cycle_pos
9: **else**
10: swap $char\_arr[cycle\_pos]$ and $char\_arr[found]$
11: **end if**
12: **if** $found \equiv -1$ **then**
13: $umark$ = first unmarked postion from start of mark array
14: **if** $char\_arr[umark] \neq umark-1$ && $char\_arr[umark-1] \neq umark$ **then**
15: swap $char\_arr[umark]$ and $char\_arr[umark-1]$
16: **else if** $char\_arr[umark] \neq mark+1$ && $char\_arr[umark+1] \neq umark$ **then**
17: swap $char\_arr[umark]$ and $char\_arr[umark+1]$
18: **end if**
19: go to 1
20: **else**
21: swap $char\_arr[cycle\_pos]$ and $char\_arr[found]$
22: **end if**
23: **end if**
24: $mark[cycle\_pos]$ = true
25: $cycle\_pos = char\_arr[cycle\_pos]$
26: **end for**

Algorithm 4.2.5

1: Declare text as a string
2: range of a alphabet $i = copy\_count[i]$ to $copy\_count[i+1]$
3: **for** each entry in $char\_arr$ **do**
4: find in which alphabet range entry lies
5: append the found character to the text
6: **end for**
7: text = InverseBWT(text)

4.3 Time Complexity :

Time complexity of the algorithm is $O(\Sigma n^2)$

Explanation :

1. 1^{st} part of algorithm will take $O(n)$ time.

2. 2^{nd} part will take $O(\Sigma)$ time.

3. In the 3^{rd} part outer loop will run for $O(n)$ times and whenever the case $char\_arr == i$ occurs inner loop will run for $O(n)$ times.

   - **Case 1 :** when i < n

     If range of a alphabet is from a_l to a_r and w.l.g assume for a_i, $a_l \leq a_i < a_r$

     given, $char\_arr[i] == i == a_i$

     Now,

     $char\_arr[i + \Sigma] == a_{i+1}$

     $char\_arr[i + 2\Sigma] == a_{i+2}$

     So,

     $char\_arr[i + \Sigma] \neq i + 1$

     therefore for any a_x, $a_i < a_x \leq a_r$

     $char\_arr[x] == x == a_x$

     We have Σ such alphabet range.

   - **Case 2 :** when i== n

     Clearly this case could occur atmost 1 time.

   So 3^{rd} part will take $O(n(\Sigma + 1))$ time.

4. Since there are only $\frac{n}{2}$ cycles possible and after each time we go to step 1, two cycles merge together. So 4^{th} part will take $O(\Sigma n^2)$ time.

5. Step 6 of 5^{th} part will take $O(n)$ time, So this part will take $O(\Sigma n)$ time.

Chapter 5

Experiments and Results

In our previous chapter we described a five part algorithm to get a highly in-compressible string for Burrows-Wheeler Transform. We conducted experiments for length up-to 100000 and sigma varies from 3 to 26. We want to compare our string, that we get as a result of algorithm for some length n, to the most in-compressible string of Burrows-Wheeler Transform for the same length, but as for length n and Σ distinct alphabets there are Σ^n possible string, it is unrealistic to determine a string with maximum MTF by brute force method. And since there is no other algorithm, we compared our result with the theoretical upper bound of MTF for BWT. Maximum value of MTF for any string of length n and Σ distinct character is $n(\Sigma - 1)$ but since in any valid BWT, sentinel character comes only once therefore maximum possible MTF for any valid BWT will be,

$(2 * \Sigma) * (\Sigma - 1) + (n - 2 * \Sigma) * (\Sigma - 2)$

In graph 5.1 we have compared our results with expected MTF value when $\Sigma = 3$. For length up-to 1000, maximum difference between actual and expected MTF value is 76 at the length 949. In graph 5.2, Σ is 26 and maximum difference is 358 at the length 675.

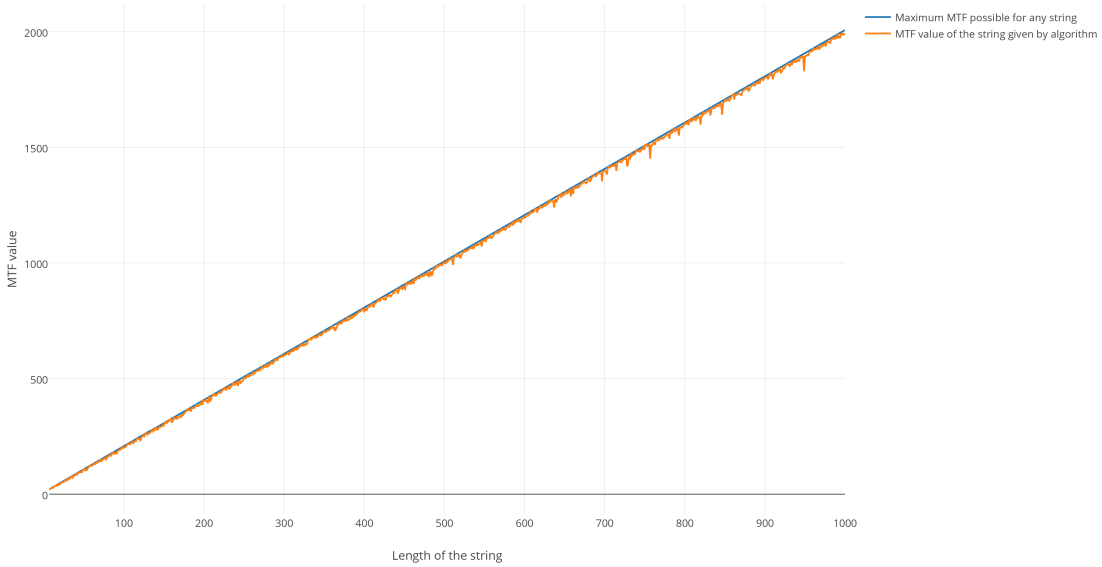

Figure 5.1: Comparison between maximum MTF for BWT with MTF value of the string, given by algorithm, where string is made of 3 distinct alphabets

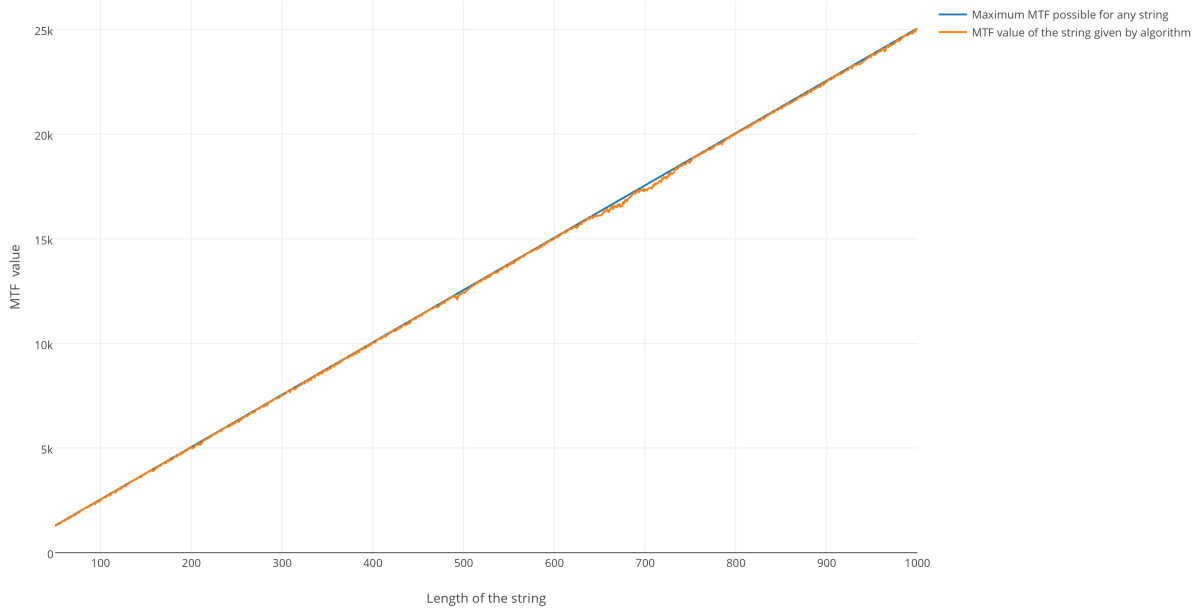

Figure 5.2: Comparison between maximum MTF for BWT with MTF value of the string, given by algorithm, where string is made of 26 distinct alphabets

We define α as ,

$$\alpha = \frac{\text{expected MTF-actual MTF}}{\text{length of the string}}$$

In the below graph, when the length is very small α tends to be higher but as the length of the string increase, value of alpha decrease. Which means the difference between the expected value and the actual value does not increases with the length. Except for the few peaks, difference almost remains constant.

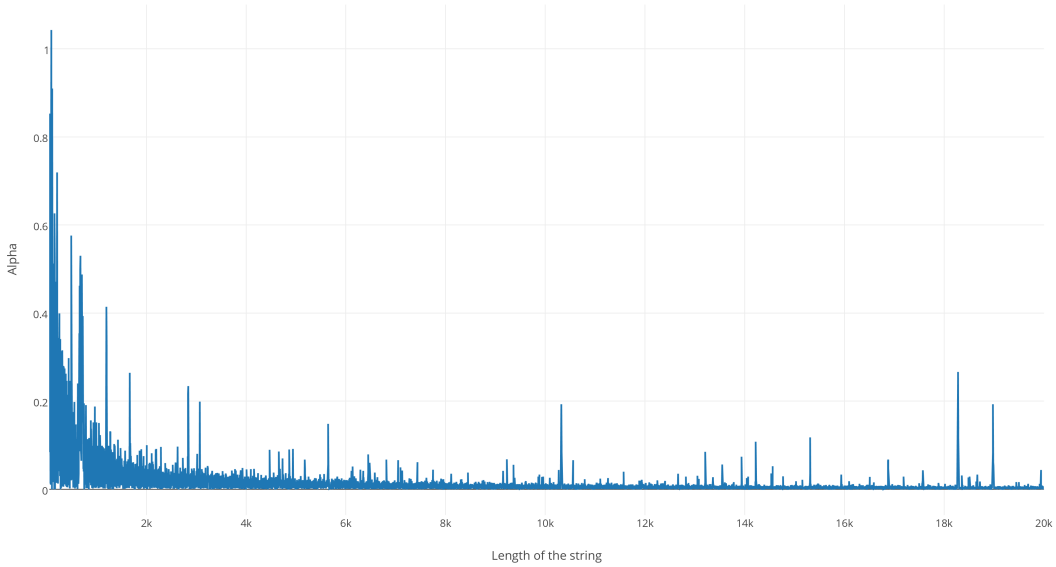

Figure 5.3: α is decreasing as the length of the string is increasing , where string is made of 26 distinct alphabets

If we only take points where difference is more than 100 and length of the string is large then this phenomena is more clearly visible.

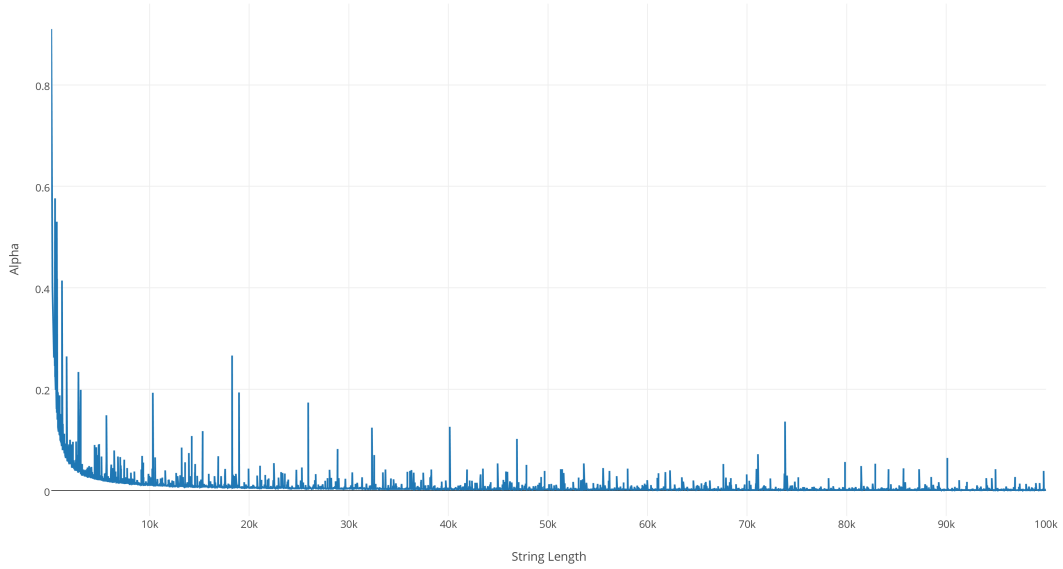

Figure 5.4: α taken at the points where the difference in expected and actual is more than 100 and string is made of 26 distinct alphabets

We define β as,

$$\beta = \frac{\text{actual MTF}}{\text{expected MTF}}$$

β shows how close the MTF value of the string, given by our algorithm, to the maximum possible MTF value for that length. For 26 different characters and length up-to 50k the average value of β come out to be 0.999702934 with the minimum of 0.959218 at length 95. In the below graph we can clearly see that as the length of the string increasing β is tends to 1.

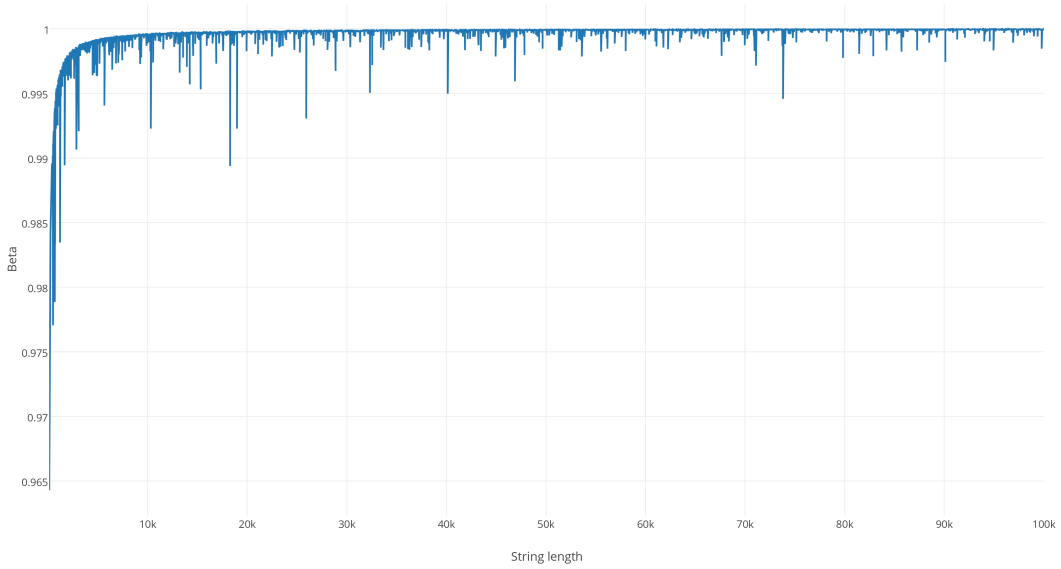

Figure 5.5: Ratio between the actual and expected MTF value where string is made of 26 distinct alphabets

Chapter 6

Future work

We leave open the question of constructing an infinite sequence ω of symbols from Σ such that all but finitely many prefixes of ω are incompressible according to the algorithm described in the thesis.

Recently, Salson, Lecroq, Léonard and Mochard [5] have presented a polynomial-time "Dynamic Burrows-Wheeler Transform" that converts BWT(S) to BWT(S') where S' is the modification of the string S. Using their algorithm, if we have an incompressible string of length n, then we can find an incompressible extension of length $n+1$ in polynomial time - thus the incompressible prefix of ω with length n can be found in time $O(|\Sigma|n^2)$.

Bibliography

[1] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm. digital equipment corporation (src research report 124) 1994."

[2] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pp. 390–398, IEEE, 2000.

[3] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, no. 4, pp. 320–330, 1986.

[4] B. Y. Ryabko and A. I. Pestunov, ""book stack" as a new statistical test for random numbers," *Problems of Information Transmission*, vol. 40, no. 1, pp. 66–71, 2004.

[5] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard, "Dynamic burrows-wheeler transform.," in *Stringology*, pp. 13–25, 2008.