

# The Burrows-Wheeler Algorithm

Daniel Schiller

August 5, 2012

## 1 Introduction

The Burrows-Wheeler Algorithm was published in the year 1994 by Michael Burrows and David Wheeler in the research report “A Block-sorting Lossless Data Compression Algorithm”. This research report is based on an unpublished work by David Wheeler from the year 1983. The Burrows-Wheeler Algorithm will used for data compression. The algorithm consists of several stages and these stages are performed successively. The input and output of each stages are blocks of arbitrary size, but the size must be specified at the beginning of the whole compression process. With the decompression of the algorithm the data which compressed by the algorithm can be returned to their original data. Meanwhile, there are countless different versions of the algorithm. In the further course we will be discuss about the original algorithm from the research report by Michael Burrows and David Wheeler from the year 1994.

## 2 Structure of the Burrows-Wheeler Algorithm



Figure 1: Structure of the Burrows-Wheeler Algorithm

The original algorithm consists of three stages. The first stage of the algorithm is the Burrows-Wheeler Transform. The aim of the first stage is to sort the characters of the input with the result that identical characters are close together, or better to sequences of identical characters. The second stage of the algorithm is the Move-To-Front Transform. In this stage, the characters of the input which are still in a local context get assigned a global index value. Thereby, the characters which occurs more frequently in the input get a lower index value as the less frequently characters. It must

be mentioned at this point that the first two stages of the algorithm don't compress the data, but the stages transformed the data with the result that they can be compressed better in the following third stage. The third stage of the algorithm is the Entropy Coding. In this stage the real compression of the data takes place. Normally, the Huffman coding or the arithmetic coding is used as entropy coder. Below we discuss more detailed about the several stages and their technique.

### 3 Burrows-Wheeler Transform

Remember that the aim of the Burrows-Wheeler Transform is to sort the characters of the input with the result that identical characters are close together. Now we look detailed at the technique of the Burrows-Wheeler Transform.

***Process steps:***

1. Order the input  $n$  ( $n$  is the length of the input) times among themselves, thereby rotate each row one character to the right compared to the previous row
2. Sort the rows lexicographical

*The output of this stage is the L-column (we call L-column for the last column and F-column for the first column of the sorted matrix) and the index value of the sorted matrix that contains the original input.*

Now we explain the procedure step by step with PANAMA as example input.

***Step 1:***

Index	F-column				L-column	
0	P	A	N	A	M	A
1	A	P	A	N	A	M
2	M	A	P	A	N	A
3	A	M	A	P	A	N
4	N	A	M	A	P	A
5	A	N	A	M	A	P

**Step 2:**

Index	F-column				L-column	
0	A	M	A	P	A	N
1	A	N	A	M	A	P
2	A	P	A	N	A	M
3	M	A	P	A	N	A
4	N	A	M	A	P	A
5	P	A	N	A	M	A

Output:  $\boxed{NPMAAA} \boxed{5}$

We see in our example that in the output the identical characters are close together as in the input. The output of these stage is the input for the next stage, the Move-To-Front Transform.

## 4 Move-To-Front Transform

As mentioned previously, in this stage the characters of the input get assigned a global index value. Therefore, we have in addition to the input a global list  $Y$ . Normally, the global list  $Y$  contains all characters of the ASCII-Code in ascending order. Now we look detailed at the technique of the Move-To-Front Transform.

**Process steps:**

1. Save the index value of the global list  $Y$  which contains the first character of the input
2. Move the saved character of the previous step in the global list on index position 0 and move all characters one position to the right which are located in the global list before the old position of the saved character
3. Repeat step 1 and 2 sequentially for the other characters of the input and use for all repetitions the modified global list from the previous repetition

*The output of this stage consists of all saved index positions and the index value of the sorted matrix from the Burrows-Wheeler Transform which contains the original input. This index value won't processed in the Move-To-Front Transform.*

Now we explain the procedure step by step with  $\boxed{NPMAAA} \boxed{5}$  (output of the example from the Burrows-Wheeler Transform) as example input. We

use  $Y = [A, M, N, P]$  as global list and don't a global list of all characters of the ASCII-Code, because the example is better to present and easier to understand with the smaller global list.

**Step 1:**

Input: NPMAAA

$Y = [A, M, \underline{N}, P] \Rightarrow$  Save index position 2

**Step 2:**

$Y = [A, M, \underline{N}, P] \Rightarrow Y = [\underline{N}, A, M, P]$

**Step 3:**

$Y = [N, A, M, \underline{P}] \Rightarrow$  Save index position 3  $\Rightarrow Y = [\underline{P}, N, A, M]$

$Y = [P, N, A, \underline{M}] \Rightarrow$  Save index position 3  $\Rightarrow Y = [\underline{M}, P, N, A]$

$Y = [M, P, N, \underline{A}] \Rightarrow$  Save index position 3  $\Rightarrow Y = [\underline{A}, M, P, N]$

$Y = [\underline{A}, M, P, N] \Rightarrow$  Save index position 0  $\Rightarrow Y = [\underline{A}, M, P, N]$

$Y = [\underline{A}, M, P, N] \Rightarrow$  Save index position 0  $\Rightarrow Y = [\underline{A}, M, P, N]$

Output: 

233300	5
--------	---

The output of this stage is the input for the next stage, the Entropy Coding. We see in this example that characters which occurs more frequently in the input get a lower index as the less frequently characters. The Move-To-Front Stage transform sequences of identical characters with the length  $m$  to sequences of zeros with the length  $m-1$ . But the question is, which profit gives us the sequences of zeros and which profit gives us the Move-To-Front Transform? Below we present with the Zero Run-Length Coding a simple compression method. We must mention at this point that in the praxis the compression methods are more complex, but this simple method is enough, to show what profit gives us the Move-To-Front Transform.

## 5 Zero Run-Length Coding

The Zero Run-Length Coding coded sequences of zeros to shorter strings. Now we explain the technique of the Zero Run-Length Coding step by step

233300 (output of the example from the Move-To-Front Transform) as example input.

**Example:**

1. Increase all characters of the input which are greater as 0 by 1

Input: 233300  $\Rightarrow$  344400

2. Coding the sequences of zeros with string combinations of 0 and 1

We use at this point an example coding table.

Number of zeros	Coding string
1	0
2	1
3	00
4	01
5	10
6	11

344400  $\Rightarrow$  Output: 34441

We see in our example that the length of the output is one character smaller as the length of the input. For a larger input we can more profit from the Zero Run-Length Coding, because for example, we can coded 6 zeros with only 2 characters. At this point we see the profit of the Move-To-Front Transform. Below we don't look detailed on the Entropy Coding. Now we speak about the decompression of the Burrows-Wheeler Algorithm and the techniques of his several stages.

## 6 Decompression and Move-To-Front Backtransform

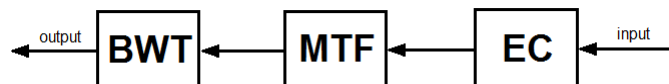


Figure 2: Decompression of the Burrows-Wheeler Algorithm

During the decompression of the data the stages of the algorithm will be run in the reverse order compared to the compression. Below we don't

talk about the Entropy Coding of the decompression, but we look at the technique of the Move-To-Front Backtransform. The technique of the Move-To-Front Backtransform is analog to the Move-To-Front Transform of the compression with the different that we have index values instead of characters as input. In addition to the input we use the same global list  $Y$  as also the Move-To-Front Transform of the compression.

***Process steps:***

1. *Save the character which is located in the global list  $Y$  on the first index value of the input*
2. *Move the saved character of the previous step in the global list on index position 0 and move all characters one position to the right which are located in the global list before the old position of the saved character*
3. *Repeat step 1 and 2 sequentially for the other index values of the input and use for all repetitions the modified global list from the previous repetition*

*The output of this stage consists of all saved characters and the index value of the sorted matrix from the Burrows-Wheeler Transform which contains the original input. This index value won't processed in the Move-To-Front Backtransform.*

We use  $\boxed{233300} \boxed{5}$  (output from the example of the Move-To-Front Transform) as example input and the output for this example is  $\boxed{NPMAAA} \boxed{5}$ . The output of this stage is the input for the Burrows-Wheeler Backtransform.

## 7 Burrows-Wheeler Backtransform

The Burrows-Wheeler Backtransform is the most complicated method of the Burrows-Wheeler Algorithm. If we remember back, we know that the input of the Burrows-Wheeler Backtransform is the L-column and the index value of the sorted matrix from the Burrows-Wheeler Transform which contains the original input. Now we look detailed of the technique from the Burrows-Wheeler Backtransform.

**Process steps:**

1. Sort the L-column of the input alphabetical  $\Rightarrow$  F-column
2. Save the character which is located in the F-column on the input index
3. Search index value  $i$ , whereby,  $i$  is the index value of the character in the L-column which is identical to the saved character of the previous step and this character has the same number of identical characters with lower index values in the L-column as the saved character of the previous step in the F-column
4. Save the character which is located in the F-column on the index value  $i$
5. Repeat step 3 and 4 and break off, when index value  $i$  is equal to the input index

The output of this stage consists of all saved characters.

Now we explain the procedure step by step with  $\boxed{NPMAAA} \boxed{5}$  (output of the example from the Move-To-Front Backtransform) as example input.

**Step 1:**

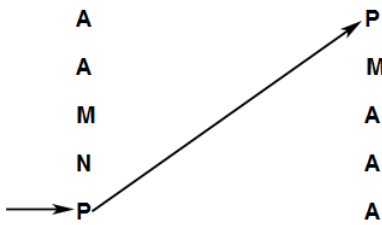
<b>Save order</b>	<b>Index</b>	<b>F-column</b>	<b>L-column</b>
-	0	A	N
-	1	A	P
-	2	A	M
-	3	M	A
-	4	N	A
-	5	P	A

*Step 2:*

<u>Save order</u>	<u>Index</u>	<u>F-column</u>	<u>L-column</u>
-	0	A	N
-	1	A	P
-	2	A	M
-	3	M	A
-	4	N	A
1	5	→ P	A

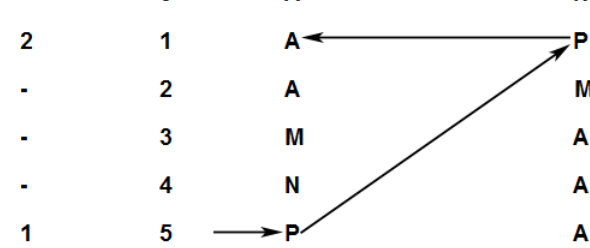
*Step 3:*

<u>Save order</u>	<u>Index</u>	<u>F-column</u>	<u>L-column</u>
-	0	A	N
-	1	A	P
-	2	A	M
-	3	M	A
-	4	N	A
1	5	→ P	A



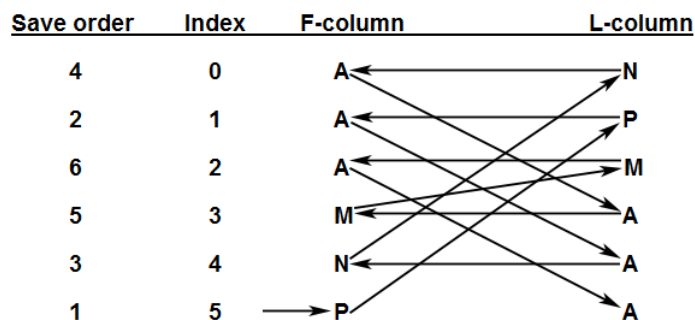
*Step 4:*

<u>Save order</u>	<u>Index</u>	<u>F-column</u>	<u>L-column</u>
-	0	A	N
2	1	A	P
-	2	A	M
-	3	M	A
-	4	N	A
1	5	→ P	A





*Step 5:*



Output: PANAMA (original input)

We see on this example that the technique works correctly. Now let's explain, why this technique does the right. To begin of the Burrows-Wheeler Backtransform we save the character which is in the F-column on the input index. It's clear that this character is the first character of the original input. In the following step we search in the L-column the identical character of the last save character from the F-column. We must be mentioned at this point that the characters of the L-column are the direct ancestor characters of the characters from the F-column. Therefore, we know that the character in the F-column, with the same index value as the search character of the L-column, is the next character of the original input. At last we repeat the method steps, to find the other characters of the original input. We know that the rows are in lexicographical order, therefore the order of identical characters from the F-column is determined by the following characters of the rows. And the order of the identical characters from the L-column is determined by the characters of the F-column and the following characters of the rows. Therefore, we can assume that the order of the identical characters in both columns are same, because the same characters in both columns decided about the order of the identical characters.

## 8 The secret of the good compression

In this article we use small examples, because the several techniques of the algorithm is with this example better to present and easier to understand, but in the praxis we compressed strings with a length of 900.000 characters. Take for example, the German language. In this language exists words which will used more frequently, for example, the word "und". If we would compress a German text, through the rotations of the input from the Burrows-Wheeler Transform, we would have each character of the input in

the F-column. Thereby, a lot of rows begin with the string "nd". Through the lexicographical order of the rows in the Burrows-Wheeler Transform, we would have sequences of the character "u" in the L-column. Now we know, why the algorithm compressed well. General, the repeated arise of the same string in a text provides a good order of the L-column and therefore also, a good compression of the complete text.

## 9 Memory Complexity

We have seen that in the theory the Burrows-Wheeler Algorithm compressed very well, but, when we using a matrix for the steps of the Burrows-Wheeler Transform, it would mean for the compression of 900.000 characters that we have an effort of memory of  $900\text{KB} \cdot 900$  (three-digit gigabyte range). This is too much for use in practice. For this reason, we use a suffix array for the steps of the Burrows-Wheeler Transform. A suffix array saves the suffixes of a string S in lexicographical order. It is of course no problem to save the rotations of the string S instead of the suffixes of the string S. Now we show an example to understand the notation of a suffix array.

**Example:**

S	=	P	A	N	A	M	A
		0	1	2	3	4	5

For the string S our suffix array A with the length 6 (length of the string S) have the following entrys:

$A[0] = 3, A[1] = 1, A[2] = 5, A[3] = 4, A[4] = 2$  and  $A[5] = 0$ .

A array entry  $A[i] = j$  of the suffix array A means that the rotation which begins with the position j of the string S is located in the lexicographical order on position i.

Through the use of a suffix array for the Burrows-Wheeler Transform, we obtain a linear memory complexity which is for the performance of the today's computer a satisfactory result. But, what is the time complexity to constructing a suffix array? At this point it should be said that for example, with the Skew-Algorithm from Kärkkäinen and Sanders from the year 2003[6], it exists a algorithm that construct a suffix array in linear time complexity.

## 10 Bzip2

Bzip2 has been developed by Julian Seward and is a free compression program which use the Burrows-Wheeler Algorithm with the huffman coding as Entropy Coding. In the following we compare the compression rate of

bzip2 with gzip (used Deflate as compression method) and 7z (used LZMA as compression method) as test data we use the files book1 (english novel), progc (source code in C) and obj1 (binary) of the Calgary Corpus[8].

**Compressible:** (in byte)

file	original size	bzip2	gzip	7z
progc	39611	<b>12560</b>	12993	12605
book1	768771	<b>232595</b>	300699	261064
obj1	21504	10795	10247	<b>9472</b>

We see that bzip2 offers for the files progc and book1 the best compression rates, this can be explained by the fact that the english roman and the C source code contains words which are used more frequently. For example, the word "the" in book1 or the command "else" in progc. We see, but also that bzip2 for the file obj1 the worst compression rate offers, this can be explained by the fact that a binary file doesn't have the relationships between the characters which the Burrows-Wheeler Algorithm needs to achieve a good compression rate.

## 11 Conclusion

We have presented with the Burrows-Wheeler Algorithm a Stagebase- Block-sorting algorithm that is used for the lossless data compression. We looked at the structure and the several stages of compression and the decompression, in which the stages are run in reverse order compared to the compression. We presented with bzip2 a data compression program which is based on the Burrows-Wheeler Algorithm. We have show on theoretical basis and in practice by comparing the various data compression programs that the Burrows-Wheeler Algorithm is very good especially by the compression of text-based files. But in this comparison was also clear that the Burrows-Wheeler Algorithm is worse for non-text-based files compared to other common data compression methods.

## References

- [1] M. Burrows and D.J. Wheeler: "A Block-sorting Lossless Data Compression Algorithm", Digital Systems Research Center, Research Report 124, 1994.
- [2] J. Abel: "Grundlagen des Burrows-Wheeler-Kompressionsalgorithmus", Informatik - Forschung und Entwicklung, Springer Verlag, Vol. 18, Nr. 2, 2004.

- [3] J. Abel: "Verlustlose Datenkompression auf Grundlage der Burrows-Wheeler-Transformation", Praxis der Informationsverarbeitung und Kommunikation, Vol. 26, Nr.3, 2003.
- [4] Fei Nan, Don Adjeroh: "An Algorithm for suffix sorting and its applications", IEEE Computational systems bioinformatics conference, 2006.
- [5] Sebastian Deorowicz: "Improvements to Burrows-Wheeler compression algorithm", Software: Practice and Experience, Volume 30, Issue 13, 2000.
- [6] J. Kärkkäinen and P. Sanders: "Simple Linear Work Suffix Array Construction", Lecture Notes in Computer Science, Vol. 2719, 943-955, 2003.
- [7] U. Manber and G. Myers: "Suffix Arrays: a new method for on-line string searching", SIAM J. COMPUT, Vol. 22, No. 5, 935-948, 1993.
- [8] Bell, T.C., Witten, I.H., Cleary, J.G.: Calgary Corpus: "Modeling for text compression", Computing Surveys 21(4), 557-591, 1989.