IJCSE International Journal of Computer Sciences and Engineering Open Access
Survey Paper Vol.-7, Issue-3, March 2019 E-ISSN: 2347-2693

A Survey on Diverse Vision and Varied Application Zone of BWT Algorithm

S.Ranjitha[1*], L. Robert[2*]

[1,2]Department of Computer Science, Government Arts College, Coimbatore, Tamil Nadu, India

[*]*Corresponding Author: ranjithabiju@gmail.com, Tel.: +91 9751619450*

DOI: https://doi.org/10.26438/ijcse/v7i3.124135 | Available online at: www.ijcseonline.org

Accepted: 20/Mar/2019, Published: 31/Mar/2019

Abstract— In the world of information, the data has to be stored in a large amount where the compression plays a vital role. Compression is beneficial because it reduces the resources required to store and transmit data. The paper discusses the different vision on BWT transformation algorithm, BWT works on data in memory and files too big to process in one go. The first section concentrates on complete Burrow Wheeler Transformation algorithm compression and decompression mechanism. This paper also examines the various Modification on BWT. The primary objective of this study is investigating the different approaches using BWT transformation. The comparison and performance analysis of second step algorithm in BWT is highlighted in this survey. Various Search algorithm using BWT is discussed briefly. A brief on recent work using this algorithm in different application Suffix Array, Suffix Sort on small space. Therefore this paper examines comparative analysis performance in compression ratio is carried out on various techniques.

Keywords— *Move to Front, Frequency Count, Suffix Sort, Inversion Frequencies.*

I. INTRODUCTION

Data and compression is the word which is ubiquitous. All the text, images and data are profusely increasing in all the fields of the research area. Data compression is necessary to reduce the size of the resource required for storing and transmitting data. Data compression is encoding information using fewer bits than the original representation. There are different methods and techniques.

Data compression methods are Lossy and Lossless based upon the kind of data used. Lossy compression achieves better compression by losing some information. When the compressed stream is decompressed, the result is not identical to the original data stream. Such a method makes sense especially in compressing images, movies, or sounds Based upon the type of data different algorithm for compression are existing. In Lossless the compression is achieved without data loss and it is applied to text. Lossless data compression is based upon entropy type, Dictionary type and other types in which the most famous and popular compression BWT is used.

The main purpose of this article is to review the researches associated with BWT algorithm based upon various data compression, since BWT is the basis of many algorithm for compression and indexing of many large collection of strings. The paper highlight a survey on original BWT algorithm to its various modification on BWT with different applications areas and its approaches.

In this paper Section I contains the introduction about the data compression and its methods. Section II gives an introduction on BWT compression. Section III gives a detail on original BWT Algorithm with compression and decompression Algorithm using MTF coding and also its performance and result. Section IV explains modification on BWT with analysis on comparison and results of the experiments. Section V explores the second step algorithm in BWT where the MTF is replaced by WFC with result with different weight functions. Section VI describes Boyer Moore algorithm with tables and figures. Section VII contributes on BWT with suffix array, suffix sorting and different algorithm implemented based on these concepts and the last Section VIII concludes the survey of the paper with future directions on survey in the BWT area.

II. BACKGROUND

A. Introduction on BWT

BWT (Burrow Wheel Transformation) is a Block sort Lossless Compression. This concept was originally developed by Michael Burrow and D.J. wheeler in 1994. This method reaches compression within a percent or so of which can be accomplished by statistical modeling techniques, but at speeds similar to those of algorithms based on Lempel and Ziv's. This

Algorithm takes and processes a block of text as a single unit rather than taking the input sequentially and applies a reversible transformation to input to form a new block of text. The transformation tends to take the group characters together so that the possibility of finding a character close to another instance of the same character is increased significantly.

The Transformed Text can easily be compressed with fast locally-adaptive algorithms, such as move-to-front coding in combination with Huffman or arithmetic coding [1].

The "Calgary Corpus" is a random collection of 18 diverse files of text, program, binary, and conventionally limited to a subset of 14 files) that forms a de-facto standard for comparing lossless compressors. Results are usually mentioned as "bits per character" (bpc) or the final compressed size (in bits) divided by the input size (bytes or characters) for each file; the 14 values are then averaged to give an overall measure or figure of merit. A better compressor will provide a corpus average of about 3.0 bpc, while the best approach 2.1–2.2 bpc [2].

III. BWT ALGORITHM (ORIGINAL)

Briefly, This algorithm never process any input sequential but takes a lock of text as a single unit. The algorithm takes a string S of N characters and changes by forming the N rotations (cyclic shifts) of S, then lexicographically sorting them, and the last character from each rotation is extracted for further reference. A string L is formed from these characters, where the i^{th} character of L is the last character of the i^{th} sorted rotation. In addition to L, the algorithm computes the index I of the original string S in the sorted list of rotations. Remarkably, there is an efficient algorithm to compute the original string S given only L and I [3]

The rough idea is to encode a text in two passes. This transformation does not compress the text but it transforms it in such a way that it is easier to compress. The algorithm basic work by applying a reversible transformation block of the input text. It has two steps of transformation.

In the original version, the transformation was done using MTF to get a final Entropy Coding stage as depicted in Figure 1.

a) Compression Transformation.
b) Decompression Transformation

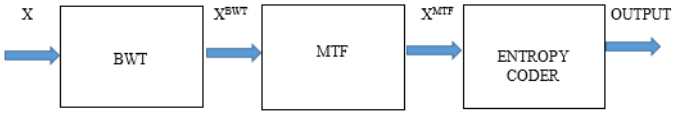

Figure 1: Burrow Wheel Compression Algorithm

Compression Algorithm :

Michael Burrow and D.J.Wheeler specified in the original BWT transformation give the easier compress closer to ideality of RLE.

In order to perform the BWT, we consider a string *S*, of length *N*, as if it actually contains N different strings, with each character in the original string being the start of a specific string that is N bytes long as shown in figure 2 and sort rotation.

Three steps for transforming

1) Form the N* N for an input string, by performing right shift one character to the end of the input string or cyclic rotating (left).
2) Sort the matrix in lexicographic order.
3) Extract the Last column of the matrix.

Here input string for transformation is considered as given below Eg: S=BANANA as shown in Fig 1

B A N A N A

Figure 2: A sample data set

B	A	N	A	N	A	[1]
A	N	A	N	A	B	[2]
N	A	N	A	B	A	[3]
A	N	A	B	A	N	[4]
N	A	B	A	N	A	[5]
A	B	A	N	A	N	[6]

Figure 3: The set of strings associated with the buffer

Considering the sample data set of Figure2, a cyclic rotation i.e Shifting is performed until the length of the input string. Here the length of the string is 6 so the shifting is performed till the 6 rows of the matrix.

The next step in the BWT is to perform a lexicographical sort on the set of input strings. That is, we want to order the strings using a fixed comparison function.

[1]	A	B	A	N	A	N
[2]	A	N	A	B	A	N
[3]	A	N	A	N	A	B
[4]	B	A	N	A	N	A
[5]	N	A	B	A	N	A
[6]	N	A	N	A	B	A

Figure 4: Lexicographical sort

We will be getting a lexicographical sort as shown in Figure3

[1]	A	B	A	N	A	**N**
[2]	A	N	A	B	A	**N**
[3]	A	N	A	N	A	**B**
[4]	B	A	N	A	N	A
[5]	N	A	B	A	N	**A**
[6]	N	A	N	A	B	**A**

Figure 5: Transformed Text Matrix

From the above transform, let L is easily obtained by taking The transpose of the last column of M together with the primary index. Eg: L=NNBAAA, INDEX = 4.

The transformed text using BWT is the last column in the sorted list, together with the row number where the original string ends up. The Benefits of taking the last column of the transformed string is because it has got a special symbol clustering using which recovering all the table entirely which other columns cannot give. The transformed text is more amenable to subsequent compression algorithms. The MTF algorithm is applied to encode the transformed text.

Algorithm1: *MOVE TO FRONT CODING:*

***Move to Front* (*MTF*)** transform is a data encoding technique designed (typically a stream of bytes) to improve the performance of *entropy encoding* techniques of compression. *MTF* each byte value is encoded by its index in a list, which changes over the course of the algorithm
This algorithm encodes the output (L, I) of the compression algorithm, where L is a string of length N and I is an index. It encodes L using a move-to-front algorithm

Step 1.1): The step applies to move to front techniques to encodes each of the characters in L to the individual characters and defines a vector of integers R[0];:::; R[N-1], which are the codes for the characters L[0];:::;L[N-1].
Step 1.2): Initialize a list Q of characters to contain each character in the alphabet X exactly once. For each i D 0;:::; N1 in turn, set R[i] to the number of characters preceding character L[i] in the list Q, then move character L[i] to the front of Q.
Taking Q = [‘A’,’B’,’N’] initially, and L =‘NNBAAA’, compute the vector R: (2 0 1 2 0 0). Apply on elements of R, anyone Huffman or arithmetic coding, treating each element as a separate token to be coded. Any coding method can be applied to the elements of R as long as the decompressor can achieve the inverse operation. [3]

Algorithm2: *MOVE-TO-FRONT DECODING*
This algorithm is the inverse of Algorithm1. It calculates the pair (L, I) from the pair. (OUT, I). We assume that the initial value of the list Q used in step1.1 is available to the decompressor and that the coding scheme used in step1.2 has an inverse operation.

Step 2.1): Decode
Decode the coded stream OUT using the inverse of the coding scheme used in step1.2. The result is a vector R of N integers. In the above example, R is: (2 0 1 2 0 0).

Step 2.2): Inverse move-to-front coding
The aim is here to analyze and compute a string L of N characters, given the move-to-front codes R [0],.R [N-1].

Initialize a list Q of characters to contain the characters of the alphabet X in the same order as in step 1.1. For each i D 0,…..,N-1 in turn, set L[i] to be the character at position R[i] in list Y (numbering from 0), then move that character to the front of Q. The resulting string L is the last column of matrix M of Algorithm1. The output of this algorithm is the pair (L, I), which is the input to the decompression Algorithm given below.
Taking Q = [‘A’,’B’,’N’] initially as Algorithm1, and to calculate the string L =‘NNBAAA’.

Decompression Algorithm
Based upon the compression Algorithm it’s possible to know primary index, 4, we know, L[4], i.e. it is the first character to retrieve backwardly.
Here we consider the Reverse of string S as S’, we should remember that the complete matrix is not available to the decompressor; only the strings S, L, and the index I (from the input) are needed by this step. [3]
The best thing about BWT is that it is not only lossless but also reversible
We define the matrix S’ formed by rotating each row of S one character to the right, so for each i = 0…..N-1, and each j = 0…..N-1,

S’ [i, j] =S [i,(j-1)/mod N]

	[1]	[2]	[3]	[4]	[5]	[6]
[1]	A	B	A	N	A	N
[2]	A	N	A	B	A	N
[3]	A	N	A	N	A	B
[4]	**B**	**A**	**N**	**A**	**N**	**A**
[5]	N	A	B	A	N	A
[6]	N	A	N	A	B	A

Figure 6: S’ Inverse BWT

Based upon the transformation using alphabetically sort for each index of the String S, the actual string in the index 4 we get it using reverse BWT.

Performance Implementation

Michael Burrow and Wheeler have given a performance implementation of their algorithm with 14 commonly used files of Calgary corpus which indicates that this algorithm does well on non-text inputs as well as text inputs. The compression expression as output is expressed here in the Table1 is in bits per input character.

Table 1: Results of compressing fourteen files of the Calgary Compression Corpus.

File	Size (bytes)	CPU time/s compress	CPU time/s decompress	Compressed size (bytes)	bits/ char
bib	111261	1.6	0.3	28750	2.07
book1	768771	14.4	2.5	238989	2.49
book2	610856	10.9	1.8	162612	2.13
geo	102400	1.9	0.6	56974	4.45
news	377109	6.5	1.2	122175	2.59
obj1	21504	0.4	0.1	10694	3.98
obj2	246814	4.1	0.8	81337	2.64
paper1	53161	0.7	0.1	16965	2.55
paper2	82199	1.1	0.2	25832	2.51
pic	513216	5.4	1.2	53562	0.83
progc	39611	0.6	0.1	12786	2.58
progl	71646	1.1	0.2	16131	1.8
progp	49379	0.8	0.1	11043	1.79
trans	93695	1.6	0.2	18383	1.57
Total	3141622	51.1	9.4	856233	-

Comparison with another compression

The performance shown here is the comparison of compression algorithm based CPU time/s for compress and decompress with three different programmers considering the same 14 files of the corpus with each individual with each algorithm and the total of the result is done, bits per character values are the means of the values for the individual files. The metric was considered for easy comparison.

The Comparison shown in Table2 is

- Compress is version 4.2.3 of the LZW based tool.
- Gzip is version 1.2.4 of Gailly's LZ77 based tool.
- Alg-C/D is Algorithms Compression and Decompression with back end Huffman coder.
- comp-2 is Nelson's comp-2 coder, limited to a fourth-order model.

Table 2: Comparison with other compression algorithms

Programme	Total CPU time/s		Total compressed	Mean
	compress	decompress	size (bytes)	bits/char
compress	9.6	5.2	1246286	3.63
gzip	42.6	4.9	1024887	2.71
Alg-C/D	51.1	9.4	856233	2.43
comp-2	603.2	614.1	848885	2.47

The authors concluded that their compression achieves good statistical modelers which is much closer in speed to coders based on the algorithms of Lempel and Ziv. Like Lempel and Ziv's algorithms, this algorithm decompresses faster than it compresses. [3]

IV. MODIFICATION ON BWT

Balkenhol and Kurtz [4] showed the improved result on the modification on BWT algorithm, based on the context tree model, and the specific statistical properties of the data were considered for the output of the BWT. The authors presented and contributed the six important properties, three of which have not been defined elsewhere which in turn improve the coding efficiency and also highlighted to compute the BWT with low complexity in time and space, using suffix trees in two different representations.

For increasing the efficiency of the BW algorithm they showed some difference from the original paper in Alphabet Encoding, modification of MTF, Grouping of symbols.

BWT does not change the entropy of the source but this Algorithm is attractive for data compression algorithm due to its low complexity.

Based on Context Tree Model (CT-Model) the following properties y^n transformed text correspond to the real input x^n.

Property 1: y^n is the sequence of independent symbols over with variable probabilities of occurrence. It resembles an infinite memory of the source generating, although the original CT-source has a restricted depth of memory.

Property 2: y^n consists of "good" and "bad" fragments corresponding to good and bad contexts, respectively. Inside the same fragment, the possibilities of occurrence of symbols almost not change (in fact, it can be an unnoticeable concatenation of "close" fragments), but this can change essentially between two fragments.

Property 3: The statistics of the fragments (i.e. the sets of different symbols in the fragments) are different.

Property 4: The fragment with a number of different symbols usually decreases with the actual length of the corresponding context (specifically, the good fragments consist of repetitions of one symbol mostly and that is one of the reasons for using MTF and run length coding as proposed in the original paper). Therefore the method of multialphabet coding, which allows adapting to an unknown subset of symbols in the fragments, should be used; MTF and grouping of numbers.

Property 5: The longer the common prefix of two contexts (the "closer" they are), the smaller the difference of the sets of symbols generated at any of these contexts. This is one more reason for applying MTF.

Property 6: At any given CT-source with an increasing message length at the output, the number of fragments slightly increases (because for a short message length some subsequence's x^n are empty). Thus the (average) length of the fragments grows almost linearly with the growing message length.

The first 3 properties are as mentioned in original BWT but the next three properties 4, 5, 6 [4] were the contributions by the authors [Balkenhol and Kurtz].

For the input, sequences are consistently based upon these 6 properties of corresponds to CT Models.

They showed the data compression program using the suffix tree-based method to compute the BWT of x^n. Suffix trees are constructed using the algorithm of McCreight [4]. Two different representations of the suffix tree were implemented:

1st representation showed edges of suffix tree to be stored in a linked list. Using the space reduction techniques an average case of 10n bytes of space is required, this gives better improvement than the original which gives 19n bytes on average.

2nd representation stores the edges of the suffix tree in a hash table. This table implements a function mapping consisting of a node and a symbol. The hash table is implemented using an open addressing technique with double hashing to resolve collisions. [4]

Comparisons and Results of the Experiment

The authors did two experiments in that the 1st program was compared with other program using the files of Calgary Corpus.as shown in Table 3 i.e the comparison rate of the switching method of VW98 of Volf and Willems, CTW (Context Tree Weighting with PPMDE- Predication by Partial Matching), PPMDE (improved version of PPM), gzip, BW94 developed by Burrows and Wheeler, of the program F96 by Fenwick, of the program BK98 developed by Balkenhol and Kurtz, and of the program BKS98.

The program VW98, PPMDE, and CTW have a better compression ratio than another program, but computational resources of these programs are more. If this restricts to the programs with the same in space and time then their program shows the best compression rates for most files.

Table3: Compression rates for Calgary Corpus (in bits/bytes)

file	*Length*		*VW 98*	*CTW*	*PPM DE*	*Gzip*	*BW 94*	*F96*	*BK98*	*BKS 98*
bib	111261	81	1.71	1.8	1.84	2.51	2.02	1.95	1.94	1.93
book1	768771	81	2.15	2.2	2.3	3.25	2.48	2.39	2.31	2.33
book2	610856	96	1.82	1.9	1.96	2.7	2.1	2.04	2	2
geo	102400	256	4.53	4.5	4.73	5.34	4.73	4.5	4.49	4.27
news	377109	98	2.21	2.3	2.35	3.06	2.56	2.5	2.49	2.47
obj1	21504	256	3.61	3.7	3.72	3.84	3.88	3.87	3.87	3.79
obj2	246814	256	2.25	2.3	2.39	2.63	2.53	2.46	2.46	2.47
paper1	53161	95	2.15	2.3	2.31	2.79	2.52	2.46	2.45	2.44
paper2	82199	91	2.14	2.2	2.3	2.89	2.5	2.41	2.38	2.39
pic	513216	159	0.76	0.8	0.81	0.82	0.79	0.77	0.74	0.75
progc	39611	92	2.2	2.3	2.35	2.68	2.54	2.49	2.5	2.47
progl	71646	87	1.48	1.6	1.66	1.8	1.75	1.72	1.71	1.7
progp	49379	89	1.46	1.6	1.67	1.81	1.74	1.7	1.7	1.69
trans	93695	99	1.26	1.3	1.44	1.61	1.52	1.5	1.48	1.47
	3141622		2.12	2.2	2.27	2.7	2.4	2.34	2.32	2.3

The second experiment was implemented on the Canterbury Corpus (including the large files e.coli, bible.txt, and world192.txt). Therefore, Table 4 shows the compression rates of gzip, PPM, bzip2, szip, BK98, and finally of BKS98. The point to be noted that all these programs, except for gzip and PPM, are based on the BWT. BKS98 gave the best compression rate on all these files. [4]

Table 4: Compression rate for the Canterbury Corpus. [4]

File	*Length*	*M*	*Gzip*	*ppm*	*bred*	*bzip2*	*Szip*	*BK98*	*BKS*
alice29	152089		2.85	2.31	2.55	2.27	2.25	2.23	2.21
ptt5	513216	159	0.82	0.99	0.82	0.78	0.82	0.74	0.75
Fields	11150	90	2.24	2.11	2.17	2.18	2.19	2.11	2.09
Kennedy	1029744	256	1.63	1.08	1.21	1.01	0.84	0.9	0.92
Sum	38240	255	2.67	2.68	2.77	2.7	2.7	2.62	2.57
lcet10	426754	84	2.71	2.19	2.47	2.02	2	1.97	1.96
plrabn12	481861	81	3.23	2.48	2.89	2.42	2.38	2.36	2.35
Cp	24603	86	2.59	2.38	2.5	2.48	2.44	2.43	2.42
grammar	3721	76	2.65	2.43	2.69	2.79	2.6	2.55	2.54
xargs.1	4227	74	3.31	3	3.26	3.33	3.25	3.11	3.12
Asyoulik	125179	68	3.12	2.53	2.84	2.53	2.51	2.49	2.48
e.coli	4638690	4	2.24	2.03	2.16	2.16	2.07	2.04	2
Bible	4047392	63	2.33	1.66	2.09	1.67	1.62	1.63	1.62
world192	2473400	94	2.33	1.66	2.24	1.58	1.6	1.56	1.54
	13970266		2.48	2.11	2.33	2.14	2.09	2.05	2.04

V. EXPLORING THE SECOND STEP OF THE BWT

Sebastian Deorowicz [5] published a paper in which discussed many of the replacement from the original version using MTF transform which is the second step in transformation. He compared different compression and yield a new better algorithm ratio from the original.

Several previous works have been done on the Second phase of the BWTA. Some of them are based on the observation that the problem is similar to the List Update Problem (LUP).

Exploring of Modification of MTF

1) Fenwick and Schindler [5] explored the suggestion of Burrow and Wheeler that refraining from moving the current character to the very first position may be sometimes useful, but they failed to obtain better compression results.

2) Balkenhol, Kurtz, and Shtarkov proposed a modification called MTF-1 which improves the compression ratio. Their only modification to the MTF algorithm is that only the symbols from the second position in the list L are moved to the first position. The symbols from higher positions are moved to the second position. [5]

3) Balkenhol and Shtarkov proposed an additional more modification to the MTF-1, the symbols from the second position are moved to the beginning of the list L only when the previous transformed symbol was at the first position which was called as MTF-2. [5]

4) One of the best algorithm for List update Problem is Time Stamp was theoretically analyzed by Albers and Mitzenmacher. The authors showed that theoretically, the Timestamp is better than then MTF.[6]

5) Arnavut and Magliveras [5] proposed a completely new approach called Inversion Frequencies (IF) Algorithm to the problem of transforming the sequence x^{bwt} to a form that can be improved compressed by an entropy coder. The IF algorithm does not solve the LUP. A sequence of x^{if} over an alphabet of integers from the range [0, n−1] is formed. For each character a_j from the alphabet, the algorithm scans the sequence x^{bwt}. When the first occurrence of the character a_j is found it outputs its position in the sequence x^{bwt}. For further occurrences of the character a_j the IF outputs an integer which is the number of characters greater than aj that occurred since the last request to the character a_j. The sequence x^{if} is not sufficient to recover the sequence x^{bwt} correctly. We also have to know the number of occurrences of each character from the alphabet in the sequence x^{bwt}. This disadvantage is especially important for short sequences.

Replacement of MTF with WFC

Sebastian Deorowicz [5] introduced The Weighted Frequency Count algorithm (WFC) by replacing the Move to Front stage (MTF) within the Burrows-Wheeler Compression Algorithm and is a representative of a List Update Algorithm (LUS) just like MTF.
Weight Frequency Count Algorithm can be reflected as a generalization of the well-known Frequency Count (FC).)

Formulation of the FC algorithm in an alternative way is shown in this paper and gave to each character a_j appearing

prior to the i^{th} position in the sequence x^{bwt} a sum and sort in descending order the list L according to the value of $W_i(a_j)$.

$$W_i(a_j) = \sum_{\substack{1 \le p \le i \\ a_j = x_p}} 1$$

The authors reformulated the above formula by introducing the weight function w instead of summing 1's for all characters that are summed from its relative position in the sequence x^{bwt}.

The formula is as highlighted below

$$W_i(a_j) = \sum_{\substack{1 \le p < i \\ a_j = x_p}} w(i - p).$$

If two characters have the same value $W_i(.)$, then to find relative order using the values $W_{i-1}(.)$, $W_{i-2}(.)$ and so on, until the counters are different. On completion, they define $W_0(a_j) = -j$. The algorithm outputting the position of processed characters in the list L and maintaining the list in the described above is called the Weighted Frequency Count.

A relation of WFC with a Context Tree sources is also been discussed since the sequence of x^{bwt} is a concatenation of Context Tree Component.

Comparison of MTF and WFC

In contrast to MTF, the WFC algorithm takes former symbol distribution into account by using a weighting scheme for former symbol occurrences. The WFC stage needs no additional overhead like the Inversion Frequencies (IF) stage and which leads to good compression rates in the BWT field. The main disadvantage of the WFC stage is the time consumption for the calculation of the weighting scheme.

Comparison of Result of different Weight Function

Examination of many weight function w can be done using the given below approaches as shown in Fig 7

$$W_1(t) = \begin{cases} 1 & \text{, for } t=1 \\ 0 & \text{, for } t > 1 \end{cases}$$

$$W_2(t) = \begin{cases} q^t & \text{, for } t=1 \\ 0 & \text{, for } t > 1 \end{cases}$$

$$W_3(t) = \begin{cases} 1/p*t & \text{, for } 1 < t \le t_{max} \\ 0 & \text{, for } t > t_{max} \end{cases}$$

$$W_4(t) = \begin{cases} 1 & \text{, for } t=1 \\ p*t & \text{, for } 1 < t \le t_{max} \\ 0 & \text{, for } t > t_{max} \end{cases}$$

$$W_5(t) = \begin{cases} 1 & \text{, for } t=1 \\ p*t^{q} & \text{, for } 1 < t \le t_{max} \\ 0 & \text{, for } t > t_{max} \end{cases}$$

$$W_6(t) = \begin{cases} 1 & \text{, for } t=1 \\ 1/p*t & \text{, for } 1 < t \le 64 \\ 1/2*p*t & \text{, for } 64 < t \le 256 \\ 1/4*p*t & \text{, for } 256 < t \le 1024 \\ 1/8*p*t & \text{, for } 1024 < t \le t_{max} \\ 0 & \text{, for } t > t_{max} \end{cases}$$

Figure 7: Examined Weight Function [5]

Based on the weight function the author has given the best set of parameters and achieved the best overall result for the w_6 function as shown in Table 5.

Table 5: Comparison result of the different weight function

File	Size	w1	w2 q = 0.7	w3 p = 4	w4 p = 4	w5 p = 0.5, q = −1.25	w6 p = 4	w6q p = 4
Bib	111261	1.915	1.916	1.969	1.916	1.899	1.896	1.896
book1	768771	2.344	2.311	2.28	2.283	2.279	2.273	2.274
book2	610856	1.999	1.98	1.999	1.973	1.962	1.959	1.958
geo	102400	4.235	4.229	4.115	4.121	4.146	4.15	4.152
news	377109	2.464	2.461	2.464	2.415	2.41	2.409	2.409
obj1	21504	3.766	3.757	3.724	3.695	3.695	3.697	3.695
obj2	246814	2.439	2.448	2.492	2.432	2.416	2.413	2.414
paper1	53161	2.42	2.422	2.488	2.424	2.405	2.403	2.403
paper2	82199	2.382	2.37	2.405	2.364	2.351	2.347	2.347
Pic	513216	0.761	0.741	0.703	0.706	0.716	0.718	0.717
progc	39611	2.455	2.461	2.521	2.451	2.431	2.431	2.431
progl	71646	1.684	1.697	1.769	1.682	1.672	1.67	1.67
progp	49379	1.667	1.69	1.787	1.69	1.673	1.672	1.672
trans	93695	1.45	1.483	1.611	1.466	1.456	1.45	1.452
Avg	3141622	2.284	2.283	2.309	2.258	2.251	2.249	2.249

The author has shown the Comparison based on second step modification of MTF ie. MTF-1, MTF-2, TS (0), DC (Distance Coding), IF and WFC is shown in Table 6 with files from Calgary Corpus. The best result is obtained by WFC and further improvement can be done. [5].

Table 6: Comparison of different second step algorithms for the files from Calgary Corpus [5]

File	Size	MTF	MTF-1	MTF-2	TS(0)	IF	DC	BS99	WFC
Bib	111261	1.915	1.904	1.904	2.012	1.963	1.931	1.91	1.896
book1	768771	2.344	2.317	2.304	2.308	2.239	2.241	2.27	2.274
book2	610856	1.999	1.983	1.976	2.027	1.964	1.938	1.96	1.958
Geo	102400	4.235	4.221	4.22	4.186	4.19	4.51	4.16	4.152
news	377109	2.464	2.45	2.449	2.586	2.459	2.397	2.42	2.409
obj1	21504	3.766	3.737	3.74	3.9	3.889	3.969	3.73	3.695
obj2	246814	2.439	2.427	2.429	2.637	2.548	2.451	2.45	2.414
paper1	53161	2.42	2.411	2.411	2.588	2.454	2.407	2.41	2.403
paper2	82199	2.382	2.369	2.364	2.458	2.366	2.343	2.36	2.347
Pic	513216	0.761	0.741	0.737	0.732	0.706	0.717	0.72	0.717
progc	39611	2.455	2.446	2.45	2.643	2.5	2.473	2.45	2.431
progl	71646	1.684	1.678	1.681	1.851	1.747	1.692	1.68	1.67
progp	49379	1.667	1.665	1.67	1.887	1.745	1.705	1.68	1.672
trans	93695	1.45	1.448	1.452	1.704	1.557	1.473	1.46	1.452
Avg	3141622	2.284	2.271	2.27	2.394	2.309	2.303	2.26	2.249

VI. BOYER –MOORE WITH BWT TEXT

Andrew Firth, Tim Bell, Amar Mukherjee, and Don Adjeroh [6] compared the various search algorithm using BWT using the index and non indexed based algorithm in which they conferred about the Boyer-Moore algorithm with BWT text.

Boyer Moore Algorithm

The Boyer Moore algorithm is used for string pattern matching and it is classified into non-index based algorithm. The Boyer-Moore algorithm (Boyer & Moore 1977) for searching an ordinary text file is considered as one of the most efficient pattern matching algorithms due to the ability to recognizing and skip the certain areas of the text where no match would be possible [6]. It scans the query pattern from right to left, making comparisons with characters in the text. When a mismatch is found, the maximum of two pre-computed functions, called the *good-suffix rule and the bad-character rule* is used to determine how far to shift the pattern before beginning the next set of comparisons. This shifts the pattern along the text from left to right, without missing possible matches, until the required patterns have been located or the end of the text is reached. The good-suffix rule is used when a suffix of P has already been matched to a substring of T, but the next comparison results in a mismatch [6].

Algorithm for Compressed Domain Search for BWT

Boyer Moore Algorithm to be used in modified compressed domain search for BWT Compression is achieved by decoding parts of the text.

The paper refers to the pattern matching problem in terms of searching for a pattern P of length m in a text T of length n. The input alphabet will be referred to as Σ; similarly, |Σ| will denote the size of the alphabet.

The algorithm given requires access to the BWT text which constructs certain arrays which have been found using BWT algorithm.

Algorithm1.1: Reconstruct the original text

```
BWT-DECODE (L, index)
 1 for i ← 0 to 255 do
 2     K[i] ← 0
 3 end for
 4
 5 for i ← 1 to n do
 6     C[i] ← K [L[i]]
 7     K [L[i]] ← K [L[i]] + 1
 8 end for
 9
10 sum ← 1
11 for ch ← 0 to 255 do
12     M[ch] ← sum
13     sum ← sum + K[ch]
14 end for
```


15
16 i ← *index*
17 for *j* ← *n* **downto** 1 **do**
18 *T*[j] ← *L*[*i*]
19 *i* ← *C*[*i*] + *M* [*L*[*i*]]
20 end **for**

Algorithm 1.2 Construct the BWT transform arrays [6]

BUILD-TRANSFORM-ARRAYS (*L, M*)
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*]] + 1
5 end **for**

Algorithm 1.3: Reconstruction of Original Text from left to right using W Array [7]

BWT-DE CODE0 (L, W, index)
1 i ← index
2 for j ← 1 to n do
3 i ← W [i]
4 T [j] ← L[i]
5 end for

Algorithm 1.4: Construction of auxiliary arrays [7]

BUILD-AUXIL IARY-ARRAYS (W, index)
1 i ← index
2 for j ← 1 to n do
3 H r[j] ← i
4 I[i] ← j
5 i ← W [i]
6 end for

The Summary of an array is been given below in Fig 5, which is conferred from the BWT transformation Algorithm 1.1, 1.2, 1.3, 1.4 and which is further used in Boyre Moore Search Algorithm 2.

Table 7: Summary of Important array used in the study [7]

Array Type	Array	Size	Algorithm	Description first used
Basic arrays	*T*	N	2.1	Original Text sequence L
		n	2.1	of Last characters
	F	n	3.1	Array of First characters
	P	m	3.1	The search Pattern
Counting Arrays	*C*	|∑|	2.1	*C*[*i*] = # of occurrence of *L*[*i*] in *L*[1…*i*− 1]
	K	|∑|	2.1	K[i] = # of occurrence of i in L (or T).
	M	|∑|	2.1	Cumulative counts of the values in K
Transform Arrays	*V*	N	2.2	One-to-one mapping between L and F . Used to construct text in reverse order.
	W	n	2.2	One-to-one mapping between L and F . Used to construct text without reverse.
Auxiliary arrays	*Hr*	N	2.4	One to one mappings between F & T
	I	n	2.4	Inverse of Hr

Algorithm 2: Boyer Moore for BWT Text [7]

COMPRESSED-DOMAIN-BOYE R-MOORE -SEARCH (P, F, and H r)
1 COMPUT E -GOOD-SUFFIX (P)
2 COMPUT E -BAD-CHARACT E R (P)
3 k ← 1
4 while k ≤ n − m + 1 do
5 i ← m
6 while i > 0 and P[i] = F [H r [k + i − 1]] do
7 i ← i − 1
8 end whilea
9 if i = 0 then
10 # Report a match beginning at position k − 1
11 k ← k + <shift proposed by the good-suffix rule>
12 else
13 sG ← <shift proposed by the good-suffix rule>
14 sB ← <shift proposed by the extended bad-character rule>
15 k ← k + MAX (sG, sB)
16 end if
17 end while

The above algorithm requires access to the text in the correct order, thus after a file has undergone the Burrows-Wheeler Transform, an ordinary Boyer-Moore search is no longer possible without full decompression first. Therefore the result shows that , Using shift heuristics, it is able to avoid making comparisons with some parts of the text and it can produce in the best case a sub-linear performance of O(n m), although on average comparison it requires O(m+n) and in the worst case deteriorates to O(mn) time complexity.[7]

VII. BWT WITH SUFFIX SORTING

To understand the BWT with suffix sorting we need to understand the concept of Suffix Array, Suffix Sorting and study of the different algorithm implemented based on these concepts, which is highlighted below.

- **Suffix Array**

A Suffix array is a sorted array of all suffixes of a given string**.** It is similar to a Suffix Tree which is compressed trie of all suffixes of the given text. Any suffix tree based

algorithm and the algorithm that uses a suffix array enhanced with additional information, both answers the same problem in the same time complexity.
A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree, both can be constructed from each other in linear time. Suffix Arrays has advantages over suffix trees which includes improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality.

Table 7: Example for Suffix Array (Wiki)

Let the given string be "banana".		
0 banana		5 a
1 anana	Sort the Suffixes	3 ana
2 nana	---------->	1 anana
3 ana	alphabetically	0 banana
4 na		4 na
5 a		2 nana
So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}		

- **Fastest Suffix Sorting[8]**

The authors N. Jesper Larssona, Kunihiko Sadakaneb [8] discussed Suffix sorting which is defined as the problem of lexicographically ordering all the suffixes of a string. The suffixes are represented as a list of integers denoting their starting positions. In the paper "Fastest Suffix Sorting", Suffix Sorting has at least two important applications:
a) The first one is the construction of a suffix array (also known as PAT array), a data structure for pattern matching that supports some of the operations of a suffix tree, generally slower than the suffix tree but requiring less space. When additional space is allocated to supply a bucket array or a longest common prefix array, the time complexity of basic operations closely approaches those of the suffix tree.
b) Another application is in data compression. The Burrows-Wheeler Transform is a transformation which facilitates compression based on repetition of string which shows better performance.
Suffix sorting is a in the Burrow Wheeler Transformation gives a computational problem and an efficient sorting method is essential for any implementation of this compression scheme.

Comparative Analysis and Conclusion: The authors have done a study of alternative approaches of an algorithm based on suffix sorting and analysis of time complexity of the algorithm.
The author has highlighted a suffix sorting algorithm which has good worst-case time complexity and actual running time using memory with reasonable size.

In theory, the suffix sorting can be done in linear time by building a suffix tree and gaining the sorted order from its leaves but it involves significant overhead mostly in space requirements which makes it too expensive to use it alone.

Here the author considered a string X = x0x1 . . . xn of n + 1 symbols, where the first n symbols include the actual input string and x_n = $ is a unique sentinel symbol. It was considered to look $, which may or may not be denoted as an actual
The symbol in the implementation, as having a value below all other symbols. By S_i, for $0 \leq i \leq n$, It's denoted with the suffix of X beginning in position i. Thus, $S_0 = X$, and S_n = $ is the first suffix in lexicographic order.
The output of suffix sorting is a permutation of the S_i, contained in an integer array *I*. In the course of the algorithm, *I* hold all integers in the range $[0, n]$. Finally, these numbers are arranged in an order corresponding to lexicographic suffix order, i.e., $S_{I[i-1]}$ lexicographically precedes $S_{I[i]}$ for all $i \in [1, n-1]$. The final content of *I* is referred to *as* the *sorted suffix array*. In practical terms, the suffix sorting means sorting the Integer I according to the corresponding suffixes.
Among all suffix sorting algorithms with worst-case O (nlogn) time complexity, the author claims the fastest and the most space-efficient. Though there exist faster algorithms than this for many inputs, they do not have good worst-case time complexity, with the result that their performance will decrease for some inputs. [8]

- **Fast BWT in small space by block wise suffix sorting [9]**

The Burrow Wheel Transformation is text reversible method that has the central role in some of the best data compression method. The transformed text is easier to compress using a simple and fast method. Computing time and space is significantly more for BWT than the other steps of the compression.

- Another application of BWT is the construction of compressed full-text indexes, which support fast substring searching on the text while taking little more space than the compressed text. Some compressed indexes are directly based on BWT (for example) while others can be efficiently constructed from the BWT. It gives more bottleneck for compressed indexes in computing the BWT
- Usually, the BWT is computed from the suffix array (SA), the lexicographically sorted array of all the suffixes of the text. Computing BWT from SA is simple and fast, and a lot of effort has been spent in developing fast and space-efficient algorithms for constructing the suffix array, i.e., for sorting the set of all suffixes. However, all such algorithms need to store the suffix array, which can be much larger than the text or the BWT.

- The suffix array needs Ω(nlogn) bits of space while the text and the BWT can be encoded using O(nlogσ) bits for an alphabet size σ. In practice, the size of the suffix array is usually at least 4n bytes while the text and the BWT typically need only n bytes each, and sometimes even less, for example, 2n bits each for a DNA sequence.

In this paper, the author gets rid of the storing the full suffix array and the suffix array as small piece or block at a time and compute the corresponding block of BWT and remove the SA block so no need of space for full suffix array.

- **Algorithm Outline**

The usual way to compute the BWT is to first construct the suffix array (SA) and then to use Eq. (1) to compute the BWT.

$$BWT[i]=\begin{cases} T[SA[i]-1] & \text{if } SA[i]\neq 0 \\ \$ & \text{if } SA[i]=0 \end{cases} \quad \text{----------(1)}$$

Here $ is a special character that is distinct from (and usually considered to be smaller than) all other characters. The definition is same to the description of common BWT as the last column in a matrix, whose rows are the rotations (cyclic shifts, conjugates) of T [0, n) $ in lexicographical order.

The algorithm uses Eq. (1), too, but the difference is that the SA is computed in smaller blocks. That is, for some $0=i_0 < i_1 < i_2 < \cdots < i_r = n+1$, At first the algorithm computes $SA[0,i_1)$ and uses it to compute $BWT[0,i_1)$, then it computes $SA[i_1,i_2)$ and BWT[i1,i2), and so on. The division of SA into blocks is determined by using a sample of suffixes as splitters. [9]

- **Experiments and Result**

The Algorithm implemented a program BWT that reads the text from a file and writes the BWT to another file. BWT is never stored in memory but is written directly to disk.
There is also a second program used for dnabwt for the four-letter DNA alphabet that stores the text using just two bits per character.
Here the author used two algorithms for comparison

a)The first algorithm is (MF) deep-shallow algorithm of Manzini and Ferragina [9]

b) The second one (BK) is the algorithm of Burkhardt and K¨arkk¨ainen [10].
The author used 6 text files for studying the running time in UNIX time command. The memory consumption is the total size of the process at its maximum as reported by the Unix top command.

Table 9: Runtime (in seconds) and memory footprint (in GB) of BWT construction algorithms [9]

	Text size=256 MB				Text Size=IGB	
Text	bwt	Dnabwt	MF	BK	bwt	Dnabwt
english	546	–	287	573	2746	-
random-64	511	–	241	605	2566	-
repeat-64	2994	–	43751	1372	13082	-
DNA	585	1 974	223	589	-	-
random-DNA	574	1 876	237	582	2898	8250
repeat-DNA	2986	12 619	70125	1323	12555	52668
Memory	0.46	0.23	1.3	1.5	1.8	0.90

The 1GB files are too large for MF and BK. The results show that bwt is quite competitive in speed. It is 2–3 times slower than MF for most texts but much faster on repetitive data while taking barely over one-third of the space. The times for bwt and BK are very similar because both spend most of their time in string sorting. The larger slow- down of bwt for repetitive data is probably due to the larger value of the parameter v. dnabwt is significantly slower than bwt but still fast enough for overnight computation of BWT for multi-gigabyte texts.
The author has presented an algorithm that can compute the BWT of text with very limited space and with speedy compression.

VIII. CONCLUSION AND FUTURE SCOPE

This paper has done an investigation on the original BWT algorithm and its mix match with different other algorithms, with slight modification. Also exploring the performance level and time complexity in space and speed. The Paper also highlights on the second stage of BWT with different variation and also their performance such as MTF1, MTF2, and IF. The survey also focused on the fast BWT transformation using a suffix array and with limited spaces. It can be concluded that many researchers have practiced on the BWT transformation using many different algorithms with satisfied performance. Thereforee the future extensions of this study can also be focused on the improvements on various search algorithms and genomic sequence with better compression ratio using BWT can be analysed.

REFERENCES

[1] David Salomon, "*Data Compression the Complete Reference*" 3rd Edition, Spring Verlag, New York, pp.777, 2004
[2] Peter Fenwick, "Burrows–Wheeler compression: Principles and reflections", Theoretical Computer Science 387 (2007) pp. 200–219, 2007 Elsevier.
[3] M. Burrows, D.J. Wheeler. *A Block-sorting Lossless Data Compression Algorithm*. DEC Systems Research Center Research Report 124, May 1994.
[4] Bernhard Balkenhol, Stefan Kurtz, Yuri M. Shtarkov, "Modifications of the Burrows and Wheeler Data Compression

Algorithm", In the Proceedings 1999 Data Compression Conference, IEEE Computer Society Washington, DC, USA pp.188-197, April 1999
[5] Sebastian Deorowicz, "*Second step algorithms in the Burrows-Wheeler compression algorithm*", November 22, 2001, Journal of Software—Practice and Experience, 2002; 32(2): pp. 99–111, 2002
[6] Tim Bell1, Matt Powell1, Amar Mukherjee, Don Adjeroh, "Searching BWT compressed text with the Boyer-Moore algorithm and binary search", Proceeding in 2002 Data compression Conference, Snowbird, UT, USA, pp. 112-121, 2002.
[7] Andrew Firth, Tim Bell, Amar Mukherjee, and Don Adjeroh "*Ä comparison of BWT Approaches to String Patterns Matching*", Journal of Software-Practice and Experience, Volume 35, Issue 13, pp. 1217-1258,10 November 2005.
[8] N. Jesper Larssona, Kunihiko Sadakaneb,∗ "*Faster suffix sorting*", Theoretical Computer Science 387 (2007) pp.258–272, 2007 Elsevier.
[9] Juha K¨arkk¨ainen∗," *Fast BWT in small space by blockwise suffix sorting* ", Theoretical Computer Science 387 (2007) pp. 249–257, 2007 Elsevier

Authors Profile

Mrs. S.Ranjitha pursed Bachelor of Computer Science from RVS College of Science & Commerce, Bharathiar University, Coimbatore. Master of Computer Application from IGNOU University in the year 2009. She received his M.Phil in the field Network Security and is currently pursuing Ph.D. and currently working as Assistant Professor in Department of Information Technology, CMS College of Science and Commerce, Coimbatore, Tamil Nadu India since 2007 onwards. She has 13 years of teaching experience.

Mr. L. Robert received his undergraduate and postgraduate degrees in Computer Science from St. Joseph's College, Tiruchirappalli, Tamil Nadu, India. He received his M.Phil and Ph.D.. Degrees in the field of Data Compression and Security from PSG College of Technology, Coimbatore, Tamil Nadu, India. He has been teaching Computer science for nearly 24 years. He is now working as Associate Professor in Computer Science, Government Arts College, Coimbatore, Tamil Nadu, India He has published more than 20 research papers in national, international journals and conferences in the area of data compression and data management. His other areas of interest include Compiler Design, Web Mining, Cloud Computing, and Big Data Analytics. He is a reviewer of many international journals including IJITWE.

