Scalable Computing: Practice and Experience, ISSN 1895-1767, http://www.scpe.org
© 2021 SCPE. Volume 22, Issues 4, pp. 387–400, DOI 10.12694/scpe.v22i4.1870

A METHOD TO IMPROVE EXACT MATCHING RESULTS IN COMPRESSED TEXT USING PARALLEL WAVELET TREE

SHASHANK SRIVASTAV[*], PRADEEP KUMAR SINGH[†], AND DIVAKAR YADAV[‡]

Abstract. The process of searching on the World Wide Web (WWW) is increasing regularly, and users around the world also use it regularly. In WWW the size of the text corpus is constantly increasing at an exponential rate, so we need an efficient indexing algorithm that reduces both space and time during the search process. This paper proposes a new technique that utilizes Word-Based Tagging Coding compression which is implemented using Parallel Wavelet Tree, called WBTC_PWT. WBTC_PWT uses the word-based tagging coding encoding technique to reduce the space complexity of the index and uses a parallel wavelet tree which reduces the time it takes to construct indexes. This technique utilizes the features of compressed pattern matching to minimize search time complexity. In this technique, all the unique words present in the text corpus are divided into different levels according to the word frequency table and a different wavelet tree is made for each level in parallel. Compared to other existing search algorithms based on compressed text, the proposed WBTC_PWT search method is significantly faster and it reduces the chances of getting the false matching result.

Key words: Parallel computing, Wavelet Tree, Compressed Text Matching, Text Searching, Word-Based Tagged Coding, Compressed Indexing.

AMS subject classifications. 68W10

1. Introduction. In recent years, the production of a wide variety of devices such as powerful laptops, tablets, Wi-Fi TVs, various electronic gadgets and other mobile devices has been growing rapidly in the field of information technology. All these devices can connect directly to the Internet and create large amounts of data and search within the produced data [14][35]. This is an example of how we are living in an era of information explosion. If the size of the data is too large, it becomes a costly affair to store, process, retrieve and communicate data on such a large scale. Therefore, we need to do data compression to manage those huge data. Compression of data brings stability to the data and renders it in the minimum number of bit-space. The compression of data involves the process of data-encoding and data-decoding. This article uses the term data packing to refer to encoding and the term data un-packing to refer to decoding.

String matching is a method of identifying all possible pattern (string/substring) events from a large text corpus. String matching features are used in various applications for information retrieval, big data, text mining, plagiarism checking, DNA matching, and more. Compressed pattern matching (CPM) is known as a process of matching string in compressed data. CPM [2][20][27] is a process in which string matching is performed directly on compressed text without the need for decompression. Compared to other algorithms, CPM supported compression algorithms are considered more effective. During the decompression process, CPM saves time wastage and reduces search time. To save disk space, CPM algorithms are used and is also used to transfer a vast amount of information over a data network. CPM is introduced using the Lempel – Ziv – Welch (LZW) compression technique in [1] by Aamir et al. The CPM problem is to observe and reveal each event of P in T, in O(u + m) time, using only P and Z, the material T, where u is the length of compressed material Z and m is the length of an example pattern P. The upside of CPM is that instead of matching it to unpacked records, it directly matches the instance in the packed document and thus optimizes the search time. Later CPM is developed by M. Farch et al. [9] using the LZ77 compression process. For large text databases, compression based on the Huffman technique is not considered efficient, as one achieves a lower compression ratio using

[*]Department of Computer Science and Engineering, MMMUT Gorakhpur, India. (shashank07oct@gmail.com).
[†]Department of Computer Science and Engineering, MMMUT Gorakhpur, India.(topksingh@gmail.com).
[‡]Department of Computer Science and Engineering, NIT Hamirpur, India.(divakaryadav@nith.ac.in) .

the Huffman compression technique. On the other hand, LZW family compression techniques (LZW77, LZ78, etc.) produce a very good compression ratio, but the problem is that they cannot be considered efficient when searching for patterns directly in the packed content. Various techniques can be used to solve this problem. In [8][24], CPM is performed using straight-line software (SLP). The SLP uses a plot based on the structure of the sentence. The run-length encoding (RLE) approach suggested in [7] is used as an example for matching, where template matching was designed using Boyer Moore [3] and Knuth Morris Pratt (KMP) [22].

In [36], the authors introduce procedures for searching in packed data for Huffman's text. To fit the example material inside the compressed material, they used KMP measurements, but the correct match is not reliably distributed. One of the problems with this packing technique is false matching, as stated in [36] that the Huffman character packing technique is modified to handle words, and example patterns to execute CPMs. There are more problems of false matching in this situation. The word-based Huffman coding is said to be using bytes instead of bits [32]. In this approach, each specific word in a sample pattern is packed with a combination of bytes instead of bits. Words are packaged with either 128 bits ("tagging-Huffman packaging") or 256 bits ("plain-Huffman packaging"). For the first byte of each word-code of the tagged Huffman package, the 7 least significant bits are used for the Huffman packaging, and the most significant bit is used as the guard bit. Each guard bit used in the word-code is to distinguish its code from other word-codes and to mark the beginning of the code for each word. Thus, the use of this technique easily detects mismatch cases, and the use of bytes does not affect the efficiency of the packaging technique. This technique allows un-packaging of the content at any time and a pattern can also be searched effectively. The word-based Huffman packaging technique treats word-models without spaces.

The work depicted in [16][18] has implemented a new packaging technique, known as word-based tagging coding (WBTC), which enables to pack the contents partially and from any subjective position using the marked bit, enables the material to be easily unpacked. It also supports CPM and can quickly detect false matches. WBTC is a packaging technology that views the word as its basic unit of compression. At each level, each word present in the material has a fixed number of bits. As with other common packing techniques it often gives us false matches in the CPM process. WBTC can also suffer from false matching problems. In [17], the matched strategy uses a linear search on the packaged material but if the material is extensive then this process becomes an expensive one. WBTC codes are generally longer than most methods for packing.

An advanced data structure, the wavelet tree (WT), is used to represent and react to sequences. This data structure is space-efficient and can be used to construct indexes. It supports the rank operation, which detects the occurrence of the word and the select operation, which detects the position of the word and executes these operations within O (1) time. WT has been used previously in [15] and is known as a data structure that can be used as self-indexed, which can be constructed for characters or words, respectively. Various symbols are available on a WT leaf, which is either the character or the word. As seen in [25][28], WT can be used for text indexing as well as spatial search. The workings of WT indexing are explained in [34] using a set of documents on WWW.

1.1. Motivation, Contribution and Organization of paper. In WWW the size of the text corpus is increasing at an exponential rate, so an efficient algorithm is needed that can reduce the time taken during the search process. Due to the large data size, it uses a lot of memory space in the device. So, we need to do data compression to solve the space problem. CPM is one of the ways to reduce data space and provide search options. Many research articles have been studied to address the CPM problem, but both compression and search methods can be further improved. Several CPM solution algorithms have been developed and are given in [2] [12] [15] [18] [17] [16] [19] [20] [21] [8] [36]. The algorithms are given in [12] [15] [21] [20] work efficiently in search but do not provide a good compression ratio. On the other hand, algorithms given in [18] [16] [17] [19] [36] provides a good compression ratio but fail to provide correct matches. Thus, an algorithm should be proposed that provides a decent compression ratio and an accurate search procedure without getting incorrect results.

This paper proposes a word-based compression as well as word-based indexing of text content. As regards the number of bits, the word compression technique provides much better results than the compression technique for characters given in [32], so we choose word-based compression and create indexes using WT. The main contributions of this paper are as follows:

Table 2.1: Huffman word codes for content T

S. No.	**Words**	**Frequencies**	**Huffman word-Codes**
0	Indian	3	1
1	a	2	011
2	good	2	010
3	is	1	0011
4	for	1	0010
5	all	1	0001
6	always	1	0000

- Study of existing word-based compression techniques with examples and study about the possibility of mismatch results in CPM.
- An indexing approach is proposed with the help of WBTC and WT which efficiently solves the CPM problem with no mismatch results.
- Compares the results of the proposed method with the other word-based compression techniques used to solve the problem of CPM.
- The proposed approach can handle both single text patterns and multiple text patterns very efficiently.

Other parts of the paper are organized as follows. The basics and related functions are described in Section 2. Section 3 contains a description of the WBTC_PWT technique. Experimental analysis and demonstration are described in Section 4. Finally, Section 5 describes the conclusions of the proposed algorithm and its future work.

2. Basics and Related Work. In this section, we only focus on the word-based packing technique that supports CPM so that we can search for any query text directly in the packed file. CPM supported word-based techniques include Huffman Packing, WBTC Packing and WT. Here we describe the Huffman packing and WBTC packing methods used exclusively for words and illustrate with examples of how codes are assigned to words in these techniques. Here we also understand WT and how to use it to efficiently search for words in packaged content.

2.1. Huffman Technique of Word-Packing. The Huffman technique uses a clever method to generate packing variable length codes. It is naive to the number of words that are most visible in corpus material. If the frequency of a word increases, we use the least number of bits in the code of that word and vice versa. So, we use this packaging method to reduce the size of the content. In [32], insight into the design of the Huffman word coding is presented. Precedent 1 illustrates the strategy used in Huffman word coding.

Precedent 1. Let's take any material like T = ”a good Indian is always a good Indian for all Indian”. The above sentence is formed from a collection of ‘a’, ‘always’, ‘all’, ‘for’, ‘good’, ‘Indian’, ‘is’. We use the Huffman word-based compression technique to derive the code for each word as presented in Table 2.1. For packing purposes, we only expect a space-less content format. When the word ends with space in a space-less content format, no changes are made to the word's code and for another word that ends with a separator (such as colon, semicolon, comma, etc.), then the word and the separator both are coded separately. Compressed content T' of content T is encoded using Table 1 and is represented as T' = 011 010 1 0011 0000 011 010 1 0010 0001 1, that requires 31 bits. If the same material is compressed using Huffman character packing, so it needs 139 bits to represent T in compressed form. This precedent reflects a huge improvement in bit requirement by using the Huffman word packing on the Huffman character packing.

Depending on the number of bits used for packing, the Huffman packing is divided into two parts - binary and plane. Each word is packaged using 128 bits or 256 bits according to the byte-oriented Huffman packaging technique. When using 128 bits, it is called binary Huffman and when using 256 bits, it is called plain Huffman. As [32] suggested, byte-oriented Huffman packaging technology uses bytes instead of bits without violating the efficiency of packing and promotes unpacking faster than binary Huffman code. We consider only seven lower bits for each byte in a packing made by binary Huffman-tagging, and these 7 bits are used for packing. In each byte, code the most significant bit (MSB) with the following rule: MSB should be 1 for the first byte

Table 2.2: WBTC Code for content T

Indexes	**Unique-Words**	**Frequency**	**Word-Codes**
0	Indian	3	01
1	a	2	10
2	good	2	0001
3	is	1	0010
4	for	1	1101
5	all	1	1110
6	always	1	000001

of the word-code and 0 for the remaining bytes. In this method by this rule, we mark the beginning of each word in the packaged content, which encourages the example text to be properly searched within the packaged material. The plane Huffman-tagging packing is also like the binary Huffman-tagging packing, except that it allocates 256 bits to a word-code. Tagged-Huffman's packing may lose some of its data due to the extra bit used to identify the beginning of a word, so we prefer to use plain Huffman packing because it has a greater number of bits.

2.2. Word-Based Tagging Code (WBTC). WBTC is an efficient tool for compression developed by [18][16]. They have built compression techniques used for dynamic datasets. In this technique, the term has been considered an effective compressed unit rather than a character. This preserves each highlight of sub-optimal code with optimal compression ratios. The risk of mismatch is also low, and it is possible to see the pattern directly in the compressed text. Precedent 2 indicates the coding methodology. Here are the steps that have been taken during the coding process:

Step 1: For m=1, the first 2m unique terms of a text corpus are given a pair of bits as '10' and '01'. (level-1)
Step 2: In each code created in the previous step, we add prefix pairs '11' and '00' and code the next group of 2m words. (m=2) (level-2)
Step 3: Utilizing steps above one can normalize the encoding method by adding prefix pairs 11 and 00 of each code created in the last level, we code the 2m words in the next level. (level-m).
Step 4: Steps 1–3 is used repeatedly until the encoding of all the words are performed.

Precedent 2. We again use content T = "a good Indian is always a good Indian for all Indian". We perform the coding of all unique words in the above material T using the WBTC packing technique, and the assigned codes are shown in Table 2.2. Now compressed content T' is given as T' = 10 0001 01 0010 000001 10 0001 01 1101 1110 01. Here we see that only 36 bits are required to represent the contents of T by the WBTC technique. As seen earlier in precedent 1, the Huffman-word packaging requires 31 bits to represent this content T. WBTC entropy is higher than Huffman word packing for smaller corpus, but the chances of false matching are higher in Huffman word packing compared to WBTC. So, WBTC is an efficient and powerful packaging technique for compressing large volumes of text data.

Searching stage: Inside the compressed content, the search request for the word 'W' is completed with the steps below:

Step 1: We first see the word-code C of W that the WBTC coding method assigns.
Step 2: Word-code C is now used to execute CPM by the linear search.

The bit-pairs '01' and '10' are used as a marker to detect the end of the word-code. For the rest of the words in the next level, the concatenation of bits '11' or '00' is used as a prefix. WBTC codes are prefix-free, so we can easily identify the beginning and end of each unique word-code. Therefore, the chance of false matching is very low. Assume that sub-string g is postfix of the following string h, e.g. h = fg in which $f\epsilon \sum * \ and \ |g| \leq |h|$. For instance, assume h is given in space less model as h = bcadcaabcacba and we pick g = abcacba from h. Since g is postfix of string h, thus it can be shown as g!h. Assume now that x and y are the word-codes provided to h and g given as x = 1100110001 and y = 0001. Here, y is the postfix of x. Thus, it can be deduced that

Table 2.3: Words with their corresponding codes

Words	**Word-Codes**
good	45 41
Indian	23 25 18
always	25 18
is	41 23 25

the WBTC packing codes are not free of prefixes and thus it is very important to verify whether the match is correct or not. We must confirm whether the match is correct or not. Suppose the word is located at the 'i' position in the packed material. To check this, we must scan the bit pairs before this location 'i'. The match is true if we find 10/01 for the last two bits. The same if we find that the last two bits are 11/00, then the match is not correct. In this way, total match results and fake matches can be obtained quickly. For example, consider searching for the query word Q = 'Indian' in the content T above, in the packaged content T', first we can find that the WBTC word-code for Q is '01'. After that, in the compressed material T', we are looking for the word-code 01 directly in T' = 10 0001 01 0010 000001 10 0001 01 1101 1110 01

We find that the word 'Indian' (word-code = 01) is mixed in packed content T' at three places. The other word-code '01' matches like '0001', '000001', '0001' and '1101' are invalid matches because they do not consist of either '01' or '10' as consecutive bit-pairs before matching positions. Here we see that by using WBTC compression we can directly search into packaged content without the need for decompression and WBTC provided codes have less chance of a faulty matching, as shown in [26].

2.3. False Matching in CPM. The codes provided by the different packaging methods are free of prefixes. The compression ratio may be okay, but the main objective of these approaches is not to perform a search directly within the packaged content, as it may result in an invalid match. Precedent 3 addresses how to obtain false matching [32] [12] [19] in the CPM method.

Precedent 3. Suppose the content consists of words that are 'Indian', 'good', 'always' and 'is' and Table 2.3 shows the codes assigned to these words using some packing technique like Huffman. We have used numbers in the place of bits to better understand the problem. Now suppose a query content T and its compressed content T' is given as follows:

T: 'good Indian'
T': 45 41 23 25 18

We examine that the words 'always' and 'is' do not appear in the content T yet their word-code (25 18) and (41 23 25) are in T', indicating that 'always' and 'is' might match in T', but in reality, they do not appear in T, and when that happens, we conclude that this is an invalid match.

2.4. Wavelet Tree (WT). WT [15], a data structure with space efficiency which represents a series of queries and answers them properly. For representing sequences, rearranging elements, a point grid, and others, a WT can be utilized. We can retrieve any content at any time by retaining the content's index [5]. Authors implemented different WT forms in [4][28] and they completed their work and performed different types of tasks on WTs. The function of WT can be understood by performing the following functions: the number of events (to count the number of a symbol present in the source material), position (in the source material, find the correct position of any symbol), and Show (in the source material, show the symbol's status). All important tasks are implemented with these simple bit-map operations such as rank and select. In [6], we briefly addressed the efficiency of rank and select operations. For any given bitmap sequence M, $rank_c(M,i)$ = the frequency of symbol c up to i^{th} location in M[1..n] and $select_c(M,i)$ = index of the i^{th} case of symbol c in M[1...n]. For e.g. bitmap is M = 1010100101011, then $rank_0(M,10) = 5$ and $select_1(M,3) = 5$. The compressibility of the content and its advantages are established in [10][29]. We can also create WTs with compressed content. You will find various letter/word symbols on the WT leaf. Insights are given in [15] on the creation of WT. A new indexing model with the properties of the WT has been developed. The greatest advantage of constructing

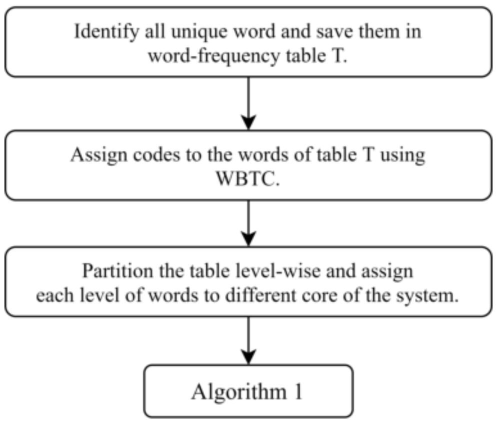

Identify all unique word and save them in word-frequency table T.
Assign codes to the words of table T using WBTC.
Partition the table level-wise and assign each level of words to different core of the system.
Algorithm 1

Fig. 2.1: Pre-processing before Construction of PWT.

indexes using the data structure of the WT is that it is much less complex than other data structures, such as the B tree and the B^+ tree. The downside to WT is that it takes a lot of time to create it. Various studies have recommended several construction paradigms. The WT is constructed sequentially or parallelly, and each has its problems. We choose a parallel WT structure to reduce the development time of WT.

There are bunches of hypothetical work on the parallel construction of the WT. [13] performed the creation of a parallel WT for the first time. The authors suggest the use of a maximum of $log\ \sigma$ processors to construct WT using two linear-time algorithms of $O(n)$ in parallel. The complexities of the solution are $O(n)$ for depth and $O(\sigma\ log\ n)$ for work. In 2015, A new algorithm for the creation of parallel WT has been introduced by [30]. In this work the construction of the WT is done level by level, requiring $O(log\ n)$ time for depth per level and $O(n)$ time for work. If there is σ number of levels in the WT, the complexities of the solution are given as $O(log\ n\ log\ \sigma)$ time for depth and $O(n\ log\ \sigma)$ time for work. [23] recommended a different strategy, which is a more space-efficient algorithm than the work of (Shun, 2015) but achieved the same complexity and limitation. The only difference is that Shun has used a 40-core processor to implement his algorithm, while J. Labeit has used a 64-core processor for his algorithm. An additional improvement is achieved by the recursive implementation of the algorithm rather than done using level by level. In [31], the author used parallel integer sorting methods to improve the work [30] proposed by himself and reduced his work up to $O(\sigma + log\ n)$ for depth and $O(n \left\lceil log\ \sigma/\sqrt{(log\ \sigma)} \right\rceil)$ for work. The fastest sequential and parallel construction of the WT has been introduced in [11]. The text size n has been divided into $\theta(n/p)$ and it has been allocated to each p-core of the system. However, it takes $O(n\ log\ \sigma)$ work and $O(n)$ time to construct the WT parallelly and it also needs $4\sigma \lceil log\ n \rceil$ in bits separately for input and output operation.

This segment reflects on how data packing is achieved and what are the issues with compression. The value of using the word as the essential compressive element in place of characters has been shown. Here we see that the word packing of Huffman is fast, and it requires less memory in the number of bits. WBTC packing is also a good method and more efficient than the Huffman methods of packing. Both Huffman and WBTC packing may give false matching results when we use it with the CPM quick-search mechanism. We also see the different WT construction approaches as we use WT to execute our WBTC packing and create the tree with multi-core computer architectures in parallel.

3. The WBTC_PWT Approach. We already understand that the data size is growing nearly every day, so it is very challenging to find any text faster. Therefore we need a data-set size reduction algorithm, which will take up less space for storing data, and will also allow a quick search. For this, we parallelize WT using the multi-core architecture of the computer and we use the WBTC packing technique with the help of WT. The idea is to divide our data-set into several unrelated sets by dividing the word-frequency table into several levels. Each level of words is assigned to a specific core of the multiprocessor computer, and each core forms a level of parallel WT. At the end of the algorithm, we logically add each small WT to create the final WT of the entire content.

Algorithm 1: Fast creation of WT

```
Input: Suppose there is an ‘X’ level in word-frequency tables.
  In each small partition table T,
    Unique-words $\longrightarrow$ $uw_1$, $uw_2$, $uw_3$ ………… $uw_n$.
    Word-codes $\longrightarrow$ $wc_1$, $wc_2$, $wc_3$ ………… $wc_n$.
Result: ‘X’ number of WTs.
Method:
 1: Par-For r $\longleftarrow$ 0 to X-1 do (assign each small table partitions to a core)
 2: For s $\longleftarrow$ 1 to n do (at each core)
 3: $P_{s,r}$ $\longleftarrow$ first prefix pair of word-code $wc_s$;
 4: Insert $A_{s,r}$ in Root($T_r$) of WT;
 5: Present_Node $\longleftarrow$ Root($T_r$);
 6: t $\longleftarrow$ 2;
 7: While (until word-code $wc_s$ is not empty) do
 8: $P_{s,t}$ $\longleftarrow$ next prefix pair bits of $wc_s$ ;
 9: If ($P_{s,t-1}$ == “00” ) then
10: Present_Node $\longleftarrow$ L-child (Present_Node);
11: Elseif ($P_{s,t-1}$ == “11” ) then
12: Present_Node $\longleftarrow$ R-child (Present_Node);
13: Else
14: Insert $P_{s,t}$ into Present_Node;
15: End If
16: t $\longleftarrow$ t+1;
17: End While
18: End For
19: End Par-For
```

Every WT is combined level by level starting from the root node in such a way that all the root nodes of small WTs have to be included in the final WT such as bitmaps of all the root nodes are combined to form the root node’s final bitmap and based on the prefix pair of bits, we then determine left and right subtree of the root of the final WT. This process is repeated for all the other nodes as well. To construct a WT S, the entire corpus is divided by first dividing the word-frequency table level-wise and then assigning each partition to a single core using a parallel for loop. The fastest sequential WT creation method given in [11] is applied to each core, and different WTs are formed for each level of words parallelly. We perform pre-processing on the text corpus, as shown in Fig 2.1, and use Algorithm 1 to build WT faster.

After the completion of Algorithm 1, each small WTs are combined in such a way that roots of all small WTs are combined using their prefix pair code-words to form the root of final WT, next L-child of root for all the small WTs are combined to form the L-child of final WT, and the same process is repeated for the R-child too. This process is followed for all the levels present in the small WT until each node of all small WTs does not merge into the final WT.

Pattern search: Whenever a search request comes for a pattern, then we perform the pre-processing for finding the WBTC code of the query word or pattern. If the word is not found in the word frequency table, we can conclude that it does not appear in the source content T, and there is no need for further processing. Once we find the WBTC code of all the words present in the query pattern from the word-frequency table, we load the final WT into the computer’s primary memory. Now we use Algorithm 2 to find the position of the query pattern in the compressed text corpus.

To better understand the proposed strategy, we consider the same previously used textual content T given as ”a good Indian is always a good Indian for all Indian”. We now perform pre-processing according to the steps shown in Fig 2.1 and apply Algorithm 1. Each unique word that is in T is stored in the word-frequency table according to their frequency and divides the word-frequency table into levels. We now provide the WBTC code

Algorithm 2: Fast Matching

Input: Word-code ‘wc’ of each query words.
Result: The exact position of each word and its occurrence in the packed text corpus.
Method:

```
1: For each unique wc do
2: $B_0 \longleftarrow$ Root (T); (T is the root of Final WT)
3: $P_0 \longleftarrow$ find the first prefix pair from word-code wc;
4: r $\longleftarrow$ 0;
5: While ($P_r$ != “01” and $P_r$ != “10”) do
6: If ($P_r$ == “00”) then
7: $B_{r+1} \longleftarrow$ L-child ($B_r$);
8: Else
9: $B_{r+1} \longleftarrow$ R-child ($B_r$);
10: End-If
11: r $\longleftarrow$ r+1;
12: $P_r \longleftarrow$ find the $r^{th}$ prefix pair bits from wc;
13: End While
14: $N_{occ} \longleftarrow Rank_{P_r}$ (Br, $|Br|$);
15: For k $\longleftarrow$ 1 to $N_{occ}$ do
16: pos $\longleftarrow Select_{P_r}$ ($B_r$, k);
17: level $\longleftarrow$ r;
18: While ($B_{level}$ != Root(T)) do
19: level $\longleftarrow$ level – 1;
20: $B_{level} \longleftarrow$ Parent ($B_{level+1}$);
21: If ($B_{level+1}$ == L-child ($B_{level}$)) then
22: $P_{level} \longleftarrow$ 00;
23: Else
24: $P_{level} \longleftarrow$ 11;
25: End If
26: pos $\longleftarrow Select_{P_{level}}$ ($B_{level}$, pos);
27: End While
28: display $k^{th}$ occurrence of the word at pos;
29: End For
```

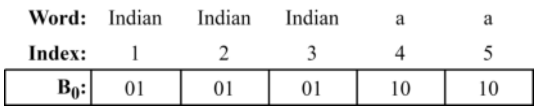
Table 3.1: WBTC codes for all the words level by level

Levels	Words in each level	Indexes	Words	Frequencies	WBTC Codes
Level 1	2^1 words	1	Indian	3	01
		2	a	2	10
Level 2	2^2 words	3	good	2	0001
		4	is	1	0010
		5	for	1	1101
		6	all	1	1110
Level 3	2^3 words	7	always	1	000001

for the words present at all levels, as shown in Table 3.1. The example pattern has three levels, and all three levels are assigned to multi-core systems for constructing three different WTs, as shown in Fig 3.1, Fig 3.2 and Fig 3.3.

All three WTs are combined to form the Final-WT because this Final-WT is used for matching a query pattern directly on the compressed text. After the WBTC packing, the example content T may look like T’ = 10 0001 01 0010 000001 10 0001 01 1101 1110 01, and its Final WT is shown in Fig 3.4.

Matching and searching: Using the WBTC_PWT approach we search if the word ‘always’ is found or not



Word:	Indian	Indian	Indian	a	a
Index:	1	2	3	4	5
B_0:	01	01	01	10	10

Fig. 3.1: WT for Level 1.

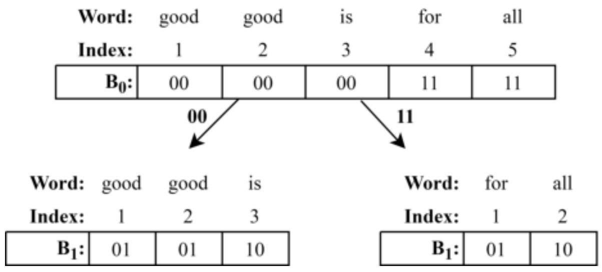


Fig. 3.2: WT for Level 2.

in the textual material T, and if it is found in T then how often and what is its correct position. To search for the word 'always' one must first get its code from the WBTC frequency table and then implement Algorithm 2. The WBTC code of the word 'always' is '000001', so we start searching from the root bitmap B_0 of the final WT. The first prefix pair of bits is '00', so we should move towards the left subtree of the final WT and reach node B_1. The next pair of bits is '00', so we next move towards the left subtree of bitmap B_1 and reach node B_2. The next pair of bits is '01' that is the flag bit, so we search bitmap B_2 for the number of occurrences of the word 'always'. If the code '01' is not found in B_2 then, we say the word 'always' is not present in the compressed content, but since it is found so we perform a rank operation at B_2 to know the frequency of the word 'always' as $Rank_{01}(B_2, |B_2|) = 1$, that implies the word 'always' appears only one time in the content T.

To give the exact position of the word 'always' we perform the select operation and traverse the final WT in reverse order. Since the frequency of occurrence of the word is 1, so there should be only one position where the word 'always' is to be found in T. The select operation is performed to give the exact position of the word 'always'. So, we calculate $Select_{01}(B_2, 1) = 1$. Now we move to the inverse direction towards its parent node and again run the select operation with the outputs of the previous select operation such as $Select_{00}(B_1, 1) = 3$. Now again we move to the inverse direction towards its parent node and again run the select operation with the outputs of the previous select operation such as $Select_{00}(B_0, 3) = 5$. Since B_0 is the root node of the final WT thus, we can correctly say that the query word 'always' is found exactly at position 5 from the starting position in the source content T.

4. Experimental Setup and Results. To do this experiment, the Intel(R) Xeon(R) CPU E3-1245-v3 with 12 Gigabytes of RAM has been selected and all our algorithms have been implemented and demonstrated with Ubuntu 18.04 LTS. The programming language used is C++ with OpenMP, and the GCC compiler is used to construct all the codes. The research is tested on a small, self-made word document and consequences are compared with existing algorithms like Huffman word packing, WT, and WBTC as these algorithms have been used for word-based compressed pattern matching. The steps for different sample words of different lengths have been executed periodically, and we have followed the average time value of the algorithms to be represented in the result. There are two algorithms for our solution, the first is for wavelet building, and the second algorithm is sample word matching. Thus, both the build-time and the matching time are the cumulative time of the methodology. With $O(n\ log\ \sigma)$ time for work and an additional $4\sigma\ \lceil log\ n \rceil$ bits of space, the construction of parallel WT is achieved in $O(n)$ time. WBTC encoding takes $O(n\ log_2\ n)$ if the source content includes n number of words. It takes $O(n)$ time to fulfil the word matching request using a WT. Therefore, our solution takes the cumulative of $O(n\ log_2\ n)$ time to design the index and matching sample query.

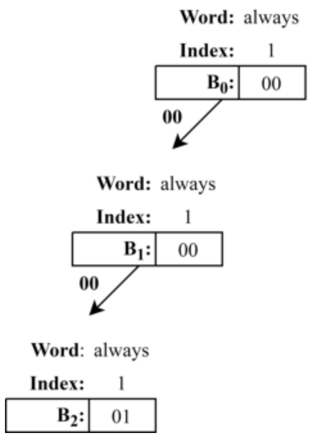


Fig. 3.3: WT for Level 3.

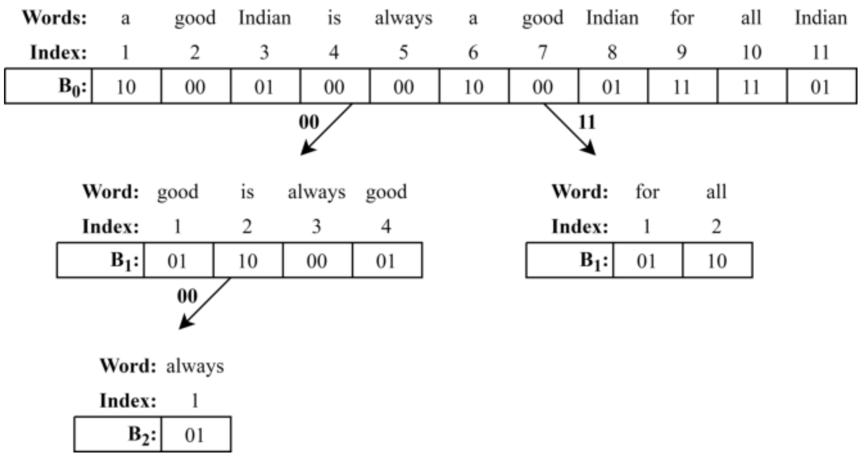

Words: a good Indian is always a good Indian for all Indian
Index: 1 2 3 4 5 6 7 8 9 10 11
B0: 10 00 01 00 00 10 00 01 11 11 01
00
11
Word: good is always good
Index: 1 2 3 4
B1: 01 10 00 01
Word: for all
Index: 1 2
B1: 01 10
00
Word: always
Index: 1
B2: 01

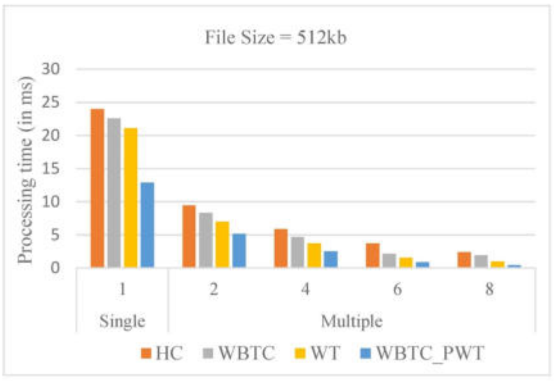
Fig. 3.4: Final WT for the content T.

In our approach, some additional memory is required in parallel construction and in creating the final index by merging all the smaller WTs into the final WT. The indexes created by our algorithm are more space-efficient than the indexes created by previous existing algorithms. The main advantage of using a WT data structure to create indexes is that its space complexity is much lower than other data structures such as B tree and B^+ tree. The other advantage of WT is that each node stores a pair of bits, so this will minimize the overall height of the WT compared to the Huffman tree for words. The rank and select operations of the WT always help to avoid mismatches and advantageously, all these operations are completed in $O(1)$ time. The build time is the only disadvantage of a WT, so we try to reduce it by using a parallel construction approach. Our proposed method matches the words better than other algorithms by comparing the processing time of the proposed method with different prevalent compressed matching algorithms for words like Huffman Word-Coding (HC), WBTC and WT. Table 4.1 and Fig 4.1display the processing time of all algorithms by varying the file size with a fixed alphabet size. Fig 4.1 displays the processing time of HC, WBTC, WT, and WBTC_PWT with a fixed alphabet length of 256. Furthermore, the processing time is also increasing as the file size increases as displayed in Table 4.1.

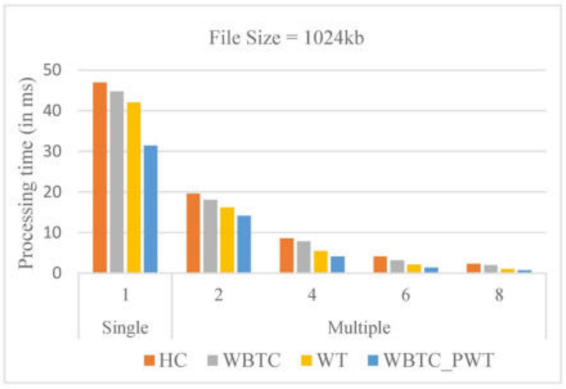
From Table 4.1 we see, for all word patterns and the file size 512 KB, the HC algorithm runs for about 45.353 seconds, the WBTC algorithm runs for about 39.584 seconds, the WT algorithm runs for about 34.320 seconds,

Table 4.1: Processing time (in seconds) of algorithms (for fixed alphabet size = 256)

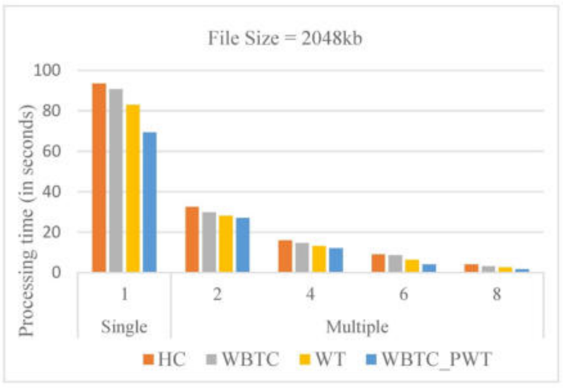
Words in Pattern		File size = 512KB				File size = 1024KB				File size = 2048KB			
		HC	**WBTC**	**WT**	**WBTC_PWT**	**HC**	**WBTC**	**WT**	**WBTC_PWT**	**HC**	**WBTC**	**WT**	**WBTC_PWT**
Single	1	23.965	22.546	21.089	12.876	46.947	44.763	41.984	31.380	93.531	90.735	82.967	69.386
Multiple	2	9.432	8.321	6.978	5.129	19.621	18.057	16.245	14.096	32.607	29.841	28.073	27.007
	4	5.854	4.675	3.723	2.531	8.639	7.847	5.397	4.062	15.832	14.546	13.094	11.998
	6	3.729	2.145	1.541	0.859	4.067	3.185	2.074	1.354	9.045	8.598	6.332	4.098
	8	2.373	1.897	0.989	0.431	2.273	1.945	1.023	0.687	4.023	3.105	2.639	1.576



(a)



(b)



(c)

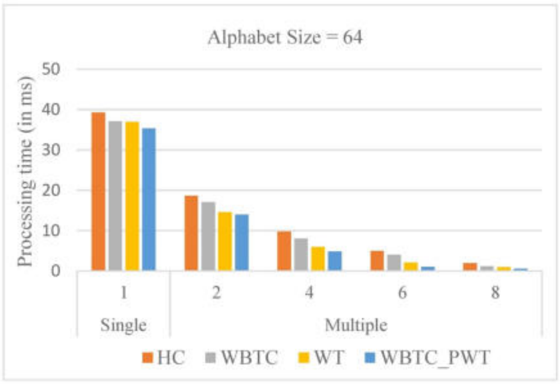
Fig. 4.1: (a), (b) and (c) shows the processing time of algorithms (for fixed alphabet size of 256)

and our suggested WBTC_PWT algorithm runs for about 21.826 seconds. Thus, our proposed algorithm shows an average increase of up to 51%, 44% and 36% relative to the HC algorithm, the WBTC algorithm and the WT algorithm, respectively. According to our estimates, the average improvement of our proposed WBTC_PWT algorithm over HC algorithm, WBTC algorithm and WT algorithm is 36%, 31% and 22% respectively for file size 1024Kb and 26%, 22% and 16% respectively for file size 2048Kb. Fig 4.1 also demonstrate that processing time decreases as the number of words in pattern increases as shown in Table 4.1.

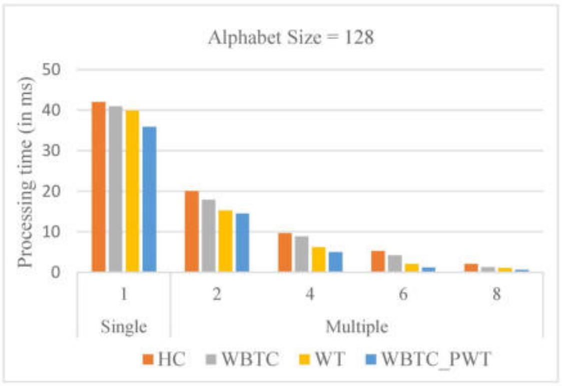
Table 4.2 and Fig 4.2 shows algorithm processing time while alphabet length varies, and file size is constant as 1024KB. Table 4.2 display that the algorithms processing time is decreasing as the alphabet size increases.

Table 4.2: Processing time (in seconds) of algorithms (for fixed file size = 1024KB)

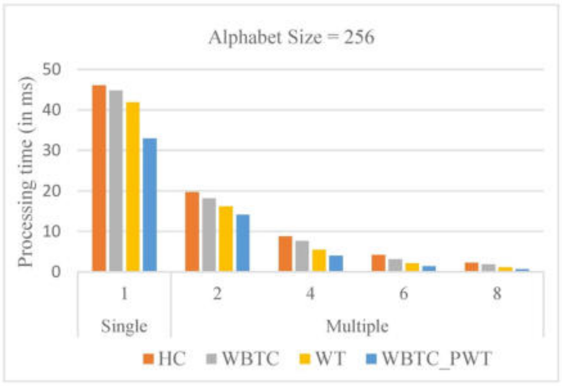
Words in Pattern		**Alphabet size = 64**				**Alphabet size = 128**				**Alphabet size = 256**			
		HC	**WBTC**	**WT**	**WBTC_PWT**	**HC**	**WBTC**	**WT**	**WBTC_PWT**	**HC**	**WBTC**	**WT**	**WBTC_PWT**
Single		39.280	37.084	36.932	35.371	42.042	40.993	39.897	35.902	46.140	44.823	41.871	32.984
Multiple	2	18.635	17.035	14.581	14.005	20.048	17.941	15.261	14.523	19.729	18.173	16.187	14.106
	4	9.734	8.047	5.932	4.842	9.642	8.903	6.186	4.943	8.833	7.683	5.431	4.007
	6	4.981	4.002	2.094	0.989	5.258	4.174	2.068	1.238	4.168	3.186	2.109	1.399
	8	1.994	1.179	0.928	0.599	2.043	1.278	1.048	0.641	2.275	1.890	1.098	0.698



(a)



(b)



(c)

Fig. 4.2: (a), (b) and (c) shows the processing time of algorithms (for fix file-size of 1024KB)

From Table 4.2 we see, for all word patterns and the alphabet size of 64, the HC algorithm runs for about 74.624 seconds, the WBTC algorithm runs for about 67.347 seconds, the WT algorithm runs for about 60.467 seconds, and our suggested WBTC_PWT algorithm runs for about 55.806 seconds. Thus, our proposed algorithm shows an average increase of up to 25%, 17% and 7% relative to the HC algorithm, the WBTC algorithm and the WT algorithm, respectively. According to our estimates, the average improvement of our proposed WBTC_PWT algorithm over HC algorithm, WBTC algorithm and WT algorithm is 27%, 21% and 12% respectively for the alphabet size 128 and 34%, 29% and 20% respectively for the alphabet size 256. Fig 4.2 demonstrate that

Table 4.3: Compression-Ratio of the Approaches

S.No.	**Approaches**	**Compression Ratio**
1.	HC	37.93
2.	WBTC	40.06
3.	WT	32.48
4.	WBTC_PWT	30.29

runtime increases as the size of the alphabet increases. The results show differences in processing time when we process a single pattern and multiple patterns in both cases. The proposed approach outperforms the other existing algorithms.

4.1. Compression Ratio. In our proposed approach WBTC_PWT, we need to store the packaged file as well as the word frequency table. This word frequency table is used to store all the unique words in the corpus along with their corresponding WBTC codes. For a large text file, the word frequency table size is also large which can directly impact the compression ratio. As stated by the heap rule [33], a text file of size 'w' in words will have the word frequency table size s = $O(w^\eta)$ for $0.3 < \eta < 0.8$. Therefore, for a large text file, storing the word frequency table must take the minimum size in memory. For the file size of 1024KB and alphabet size of 256, Table 4.3 shows the compression ratio of the approaches discussed here and clearly shows that on average WBTC_PWT takes a lower space than the other approaches.

5. Conclusion and Future work. This paper presents a method WBTC_PWT, for exact text matching in a compressed text corpus. We also tried to minimize the size of the data by performing data packing, so less memory is needed in the processing of data. This paper presents an improvement in matching time compared to other algorithms. As we know, whenever the size of the indexes is larger than the size of the main memory and a search request arrives for any query, these indexes must be loaded into the main memory. Since the size of the indexes is larger than the size of the main memory, it will cause a high number of page faults and thus affect the overall system throughput. We used the WBTC packing technique to minimize the size of the data at the initial stage and then create the indexes of the data with the help of the WT. The main advantage of creating indexes using a WT is that it takes up less memory than other indexing methods and it is free from getting a false match. The main drawback of the WT is its construction time. So, to minimize the construction time of the WT, we used parallel processing using the computer's multi-core architecture. Furthermore, we minimize the matching and search time by matching the query text directly to the compressed text content, without the need for decompression. This approach improves the overall throughput of the system.

In the future, we will attempt to create a WT using machine learning, by using our algorithm with various size of text datasets. We can also use computer classification models to find and measure the frequency of specific terms to allocate WBTC codes to all words. Using machine learning we can reduce the additional memory needs in constructing WT in the WBTC_PWT algorithm.

REFERENCES

[1] Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: Pattern matching in z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[2] Richard Beal and Donald Adjeroh. Compressed parameterized pattern matching. *Theoretical Computer Science*, 609:129–142, 2016.

[3] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[4] Nieves R Brisaboa, Yolanda Cillero, Antonio Farina, Susana Ladra, and Oscar Pedreira. A new approach for document indexing usingwavelet trees. In *18th International Workshop on Database and Expert Systems Applications (DEXA 2007)*, pages 69–73. IEEE, 2007.

[5] Nieves R Brisaboa, Antonio Farina, Susana Ladra, and Gonzalo Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval*, 15(6):527–557, 2012.

[6] Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *International Symposium on String Processing and Information Retrieval*, pages 176–187. Springer, 2008.

[7] Edleno Silva De Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Direct pattern matching on compressed text. In *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No. 98EX207)*, pages 90–95. IEEE, 1998.

[8] Edleno Silva De Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast searching on compressed text allowing errors. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 298–306, 1998.

[9] Martin Farach and Mikkel Thorup. String matching in lempel—ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[10] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):20–es, 2007.

[11] Johannes Fischer, Florian Kurpicz, and Marvin Löbel. Simple, fast and lightweight parallel wavelet tree construction. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 9–20. SIAM, 2018.

[12] Kimmo Fredriksson and Maxim Mozgovoy. Efficient parameterized string matching. *Information Processing Letters*, 100(3):91–96, 2006.

[13] José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. Efficient wavelet tree construction and querying for multicore architectures. In *International Symposium on Experimental Algorithms*, pages 150–161. Springer, 2014.

[14] Lars Gleim and Stefan Decker. Open challenges for the management and preservation of evolving data on the web. *MEPDaW@ ISWC*, 2020.

[15] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. 2003.

[16] A Gupta and S Agarwal. A scheme that facilitates searching and partial decompression of textual documents. *Int. J. Adv. Comput. Eng*, 1(2), 2008.

[17] Ashutosh Gupta and Suneeta Agarwal. A fast dynamic compression scheme for natural language texts. *Computers & Mathematics with Applications*, 60(12):3139–3151, 2010.

[18] Rahul Gupta, Ashutosh Gupta, and Suneeta Agarwal. A novel approach of data compression for dynamic data. In *2008 IEEE International Conference on System of Systems Engineering*, pages 1–6. IEEE, 2008.

[19] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[20] Radhika Khetan, Suneeta Agarwal, and Rajesh Prasad. An efficient approach towards compressed parameterized word matching using wavelet tree. *Journal of Information and Optimization Sciences*, 37(2):285–301, 2016.

[21] Shmuel T Klein and Dana Shapira. Compressed matching in dictionaries. *Algorithms*, 4(1):61–74, 2011.

[22] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[23] Julian Labeit, Julian Shun, and Guy E Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. *Journal of Discrete Algorithms*, 43:2–17, 2017.

[24] Yury Lifshits. Processing compressed texts: A tractability border. In *Annual Symposium on Combinatorial Pattern Matching*, pages 228–240. Springer, 2007.

[25] Christos Makris. Wavelet trees: A survey. *Computer Science and Information Systems*, 9(2):585–625, 2012.

[26] Surya Prakash Mishra, Rajesh Prasad, and Gurmit Singh. Fast pattern matching in compressed text using wavelet tree. *IETE Journal of Research*, 64(1):87–99, 2018.

[27] Surya Prakash Mishra, Col Gurmit Singh, and Rajesh Prasad. A review on compressed pattern matching. *Perspectives in Science*, 8:727–729, 2016.

[28] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[29] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2–es, 2007.

[30] Julian Shun. Parallel wavelet tree construction. In *2015 Data Compression Conference*, pages 63–72. IEEE, 2015.

[31] Julian Shun. Improved parallel construction of wavelet trees and rank/select structures. *Information and Computation*, 273:104516, 2020.

[32] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

[33] Dick C van Leijenhorst and Th P Van der Weide. A formal derivation of heaps' law. *Information Sciences*, 170(2-4):263–272, 2005.

[34] Arun Kumar Yadav, Divakar Yadav, and Rajesh Prasad. Efficient textual web retrieval using wavelet tree. *International Journal of Information Retrieval Research (IJIRR)*, 6(4):16–29, 2016.

[35] Divakar Yadav, Apeksha Singh, and Vinita Jain. Search results optimization. In *International Conference on Contemporary Computing*, pages 325–334. Springer, 2011.

[36] Nivio Ziviani, E Silva De Moura, Gonzalo Navarro, and Ricardo Baeza-Yates. Compression: A key for next-generation text retrieval systems. *Computer*, 33(11):37–44, 2000.

Edited by: Dana Petcu
Received: Apr 26, 2021
Accepted: Nov 16, 2021