EAI Endorsed Transactions
on Scalable Information Systems
Research Article EAI.EU

An approach for fast compressed text matching and to avoid false matching using WBTC and wavelet tree

Shashank Srivastav[1], P. K. Singh[1] and Divakar Yadav[2]*

[1] Madan Mohan Malaviya University of Technology, Gorakhpur, Uttar Pradesh, India
[2] National Institute of Technology, Hamirpur, Himachal Pradesh, India

Abstract

Text matching is a process of finding the frequency of occurrences of text pattern in a corpus. It's very costly to store, process, and retrieve a vast volume of text data. In this paper, we present a method to keep the massive text corpus in lesser memory space by using text compression and to retrieve the results by matching directly on this compressed corpus without decompression using compressed pattern matching (CPM). The proposed approach also helps to minimize the time taken to perform matching without compromising the false matching results. We used word-based tagged coding to perform text compression and Wavelet Trees for representing the compressed text in memory. The proposed Text Matching in Compressed text using Parallel Wavelet Tree (TMC_PWT) method is quite fast in comparison to other existing text matching algorithms that support CPM. In the context of CPM, the proposed method provides a good compression ratio and does not suffer from the problem of false matching.

Keywords: Modern Information Retrieval, Wavelet Tree, Word-Based Tagged Code, Compressed Pattern Matching.

Received on 31 May 2020, accepted on 30 September 2020, published on 23 October 2020

doi: 10.4108/eai.23-10-2020.166717

*Corresponding author. Email: dsy99@rediffmail.com

1. Introduction

The process of finding all the possible occurrences of a pattern (string/substring) inside a huge text content is popularly known as string matching. The various application areas of information retrieval that utilize the features of string matching are big data, text mining, plagiarism checking, etc.

In the last couple of years, we've got seen heaps of growth in data Technology (IT) in terms of powerful devices like laptops, smartphones, wi-fi television, and some other electronic hand-held gazettes. All these devices are capable of connecting to the internet and can produce a lot of data itself. The huge amount of data is produced by the internet [1] using these devices. As we know, when the size of data becomes too large, then storage, processing, retrieval, and communication of such huge data become a costly affair. So, to manage this, one needs to do data compression. Data compression is finding redundancy in data to represent it in less space. Data compression includes encoding and decoding of data content. Many compression techniques are there for achieving text compression, but retrieving any text matching results on compressed data becomes quite an uphill task. In modern information retrieval, knowing the number of occurrences of any query pattern is more important rather the position where the match occurs as in DNA sequence matching. Here, encoding refers to the packing of data, whereas decoding refers to the unpacking of data.

The problem with data compression is that it is not suitable for matching text directly because contents are not in its original form, so every time decompression is needed to perform matching. One solution to this problem is known as Compressed pattern matching (CPM) [2], [3], [4]. CPM is the process where matching of text directly performed on the compressed text without the need for decompression. There are very few compression techniques that support

CPM with or without getting a false match. The compression algorithm that supports CPM is considered more efficient in comparison to other algorithms. CPM saves time wasted during the decompression process and thus minimizes the matching time. CPM was used to save the disk space and is also useful in sending a huge amount of data over a communication network. The upsides of CPM are that it matches an example text in the packed document straightforwardly as opposed to matching it in the unpacked document, and it also minimizes match time as well as optimizes the compression ratio.

The CPM was first presented by Amir et al. in [5] using the Lempel-Ziv-Welch (LZW) packing technique. They expressed that given a content T, packed content Z of length u, and an example P of length m, the CPM is seeking and discovering each of the events of P in T by utilizing only P and Z will require O(u+m) time. Later, M. Farach et al. [6] developed CPM by utilizing the LZ77 packing technique. Compression based on Huffman techniques is not suitable for large textual databases because using the Huffman packing technique, one gets an abysmal compression ratio and sometimes false matches too. In contrast, LZW family packing techniques (LZW77, LZ78, etc.) produces an excellent compression ratio. But the problem is that the LZW packing techniques are not efficient in matching a pattern directly in the packed text. CPM utilizing the straight-line program (SLP) was examined in [7], [8]. The SLP utilizes a sentence structure-based pressure plot. Example matching based on utilizing the Run Length Encoding technique (RLE) proposed in [9], where example matching was contrived utilizing Knuth Morris Pratt (KMP) [10] and Boyer Moore [11] calculations. In [12], the authors created procedures for looking at Huffman packed records. They utilized the KMP calculation for matching an example inside the packed content, yet it isn't delivering the right match dependably. False matching is one of the issues with the Huffman packing technique, as discussed in [13] when the Huffman packing technique for characters was adjusted to deal with the words. In [14], the Huffman code for words was adopted by utilizing byte rather than bits. In this technique, a space-less word model is used and each unique word of the example pattern was packed using an arrangement of entire bytes rather than bits. The packing of each word was done using either 128 bits (refers to "tagged Huffman packing") or 256 bits (refers to "plain Huffman packing"). In every byte of the tagged Huffman packing, 7 bits are utilized for the Huffman packing, and the remaining 1 bit was utilized as the sign bit. The sign bit was used to show that the code of the word starts from here. Thus, using this technique, cases of false matching are easily identified, and by utilizing byte rather than bits doesn't affect the performance of the packing technique. Using this technique, unpacking of the content can begin anytime, and from any position.

In [15], [16], a new packing technique introduced as a word-based tagged code (WBTC), which permits partial packing of content and supports quick unpacking of the content from any subjective position by utilizing marking bit. It also supports CPM and can identify false matching easily. WBTC is a packing technique that considers 'word' as its integral unit for compression. It assigns an even number of bits to each unique word present in the content. The codes provided by WBTC is usually longer than other packing technique, but as other popular packing technique sometimes gives you false matches. While matching, the strategies in [16] utilize linear matching over the packed content, which turns out to be expensive issues, when content size is huge.

Wavelet Tree (WT) is an advanced data structure that is used to represent sequences and can answer queries on it. WT is a very space-efficient data structure used for creating indexes. WT is first used by [17] and is a self-indexed data structure that can be built for characters as well as for words. The symbols that are represented by WT, either it is character or words, are available at the leaf of a WT. WT can be utilized in web indexing [18], document indexing [19] as well as in geographical indexing [20], [21].

Motivation: In literature, many pieces of research have been done to solve the problem of CPM, still there is room to improve the efficiency of compression as well as searching methodology. Many algorithms have been developed to solve the CPM problem such as [2], [4], [12], [13], [15], [16], [17], [22], [35], [36]. Some algorithm [16], [17] works very efficiently in the searching process but fails to provide good compression ratio. On the other side, some algorithm [13], [14], [15], [22], provide good compression ratio but they sometimes suffer from the problem of false matching. So, there is a need to propose an algorithm that provides good compression ratio as well as an efficient searching methodology without getting false matching results.

Contribution: In this paper, we propose an efficient algorithm to access the compressed data with the help of WBTC and WT. We present an approach that decomposes the whole WT into different numbers of small WTs, based on the length of words in the text corpus. The construction of WTs are performed using the TMC_PWT algorithm by utilizing the multicore architecture of computers. The main purpose of doing the decomposition of a complete WT into several WTs is to load only those WTs into the main memory that are useful in the matching process. The proposed method of matching is quite fast as compared to other existed matching algorithms and does not suffer from the problem of false matching of text in the process of CPM.

This paper is systematized in the following sections. Section 2 is presented for a better understanding of related works. Section 3 deals with the proposed algorithm. Section 4 presents experimental work and results. At last, conclusion and future work are given in section 5.

2. Related Work

In this paper, we utilize the WBTC packing technique that supports CPM and allow us to match any query text directly into the compressed file without getting a false match. The popular packing techniques that support CPM is Huffman coding. Here we discuss various types of Huffman coding

(characters/words) and WBTC packing, and examples are used to show how the codes are assigned to word/characters.

2.1. Huffman Packing Technique for Text

The Huffman packing utilizes a greedy methodology to create variable-length code. It is greedy towards the number of occurrences of the symbol in the source content. The symbols could be characters or separators. In this packing, we follow the methodology that higher the number of occurrences of any symbol, lower the number of bits is used to code that symbol and vice-versa. The insights regarding the development of the Huffman code is given in [22]. Precedent 1 represents the Huffman code for content.

Precedent 1: We take the content: T = "this person is young, the way an actual young person is young" over the letters in order set {, , a, c, e, g, h, i, l, n, o, p, r, s, t, u, w, y}. We always consider the space-less content model for packing purposes. In the space-less content model, if any symbol ends with space, then that symbol is packed as it is, and the remaining other symbols that end with separator, then the symbol and the separator both are packed separately.

Table 1. Huffman code for each character

Symbol	Frequency	Huffman Code
n	6	000
o	5	0011
s	5	0010
a	4	0110
u	4	0101
y	4	0100
e	3	1010
g	3	1001
i	3	1000
t	3	0111
h	2	1101
p	2	1100
r	2	1011
c	1	11110
l	1	11101
w	1	11100
,	1	11111

Table 1 presents the Huffman packing of all the unique symbols. Utilizing character-based Huffman packing technique, the all number of bits necessary to represent the above content T (50 symbols) is 198 bits. In contrast, the same content T takes 50 * 8 = 400 bits using ASCII (8-bit code). It demonstrates the benefit of using the Huffman code in place of the ASCII code. The average compression ratio is increased by 22% when words are utilized as the fundamental components of vocabulary instead of symbols. Thus, the word-based Huffman packing technique acquires the preferred ratio of compression over the character-based Huffman packing technique. Table 2 presents the word-based Huffman packing that utilizes a space-less word model for the same given content T. All the coding steps are similar for Huffman code for words as they used in Huffman compression for characters. Still, the basic component of coding is words instead of characters. Precedent 2 clarifies the Huffman compression for words.

Precedent 2: We take the same content: T = "this person is young, the way an actual young person is young". The arrangement of the above sentence is done from the set {‘this', 'person', 'is', 'young', 'the', 'way', 'an', 'actual', ','}. By utilizing the method of Huffman compression for the word, we acquired codes for each unique word, as appeared in Table 2.

Table 2. Huffman code for each word

Words	Frequencies	Huffman Code
young	3	11
Is	2	10
person	2	011
An	1	0101
the	1	0100
way	1	0011
this	1	0010
actual	1	0001
,	1	0000

By using Table 2, the encoded content T′ = 0010 011 10 11 0000 0100 0011 0101 0001 11 011 10 11 requires 40 bits, though content compression in Precedent 1 utilizes 198 bits, which demonstrates an uncommon improvement in terms of quantity of bits requirement.

2.2. Byte-Oriented Huffman Packing

In this packing technique, each word is packed using either 128 bits (refers to "tagged Huffman packing") or 256 bits (refers to "plain Huffman packing"). As indicated by [14], this packing technique utilizes bytes rather than bits without compromising with the packing performance, yet makes unpacking a lot quicker than the binary Huffman code. In tagged Huffman packing, for every byte, they consider just seven lower bits, and these seven bits are utilized for packing purposes. The most significant bit (MSB) of every byte is packed using the following rule: codeword's first byte should have MSB as 1, and the rest following bytes have MSB as 0. The MSB bits are used for marking the beginning of each word in this packed content, and this allows direct matching of an example text into the packed content. The working of plain Huffman packing is the same as tagged Huffman packing, but it assigns 256 bits to the codewords of each word. Tagged Huffman code may lose some of its data because of an additional marking bit used for identifying the beginning of a word; that's why we use plane Huffman packing.

2.3. False Matching in Huffman code

As we know, codes generated using Huffman packing are prefix-free. This quality of Huffman coding accelerates the packing procedure. Still, this method is not primarily used to perform direct matching inside the packed content because there are chances of false matching [14], [22], [35]. How false matching occurs in the process of CPM is clarified in Precedent 3.

Precedent 3: From Table 3, comprising of four words that are 'young', 'person', 'is' and 'actual' with their codes (generated by some other compression method).

Table 3: Words with their corresponding codes

Word	Frequencies	Code
young	3	55 120
person	2	44 50 98
is	2	50 98
actual	1	120 44 50

Assume the content T is: 'young person'. Its packed content T′ is 55 120 44 50 98. As shown in compressed content T′, we analyse that the word 'is' and 'actual' isn't present in the first content T yet their codeword (50 98) and (120 44 50) are found in T, it represents that word 'is' and 'actual' may be matched in the packed content T′, but they are not present in T, and if it happens then, we say it's a false match.

2.4. Word-Based Tagged Code

It's an effective compression technique created for a dynamic dataset in 2008 by A. Gupta and S. Agarwal [15], [16]. In this technique, the word is considered an essential unit for compression. It keeps every one of the highlights of the tagged sub-optimal code while keeping up the great compression ratio. It also does not suffer from the problem of false matching and provides direct matching of patterns in the compressed text. Following are the steps utilized in the coding process:

Step 1: For m=1, we assign a pair of bits as 10 and 01 to the first 2m unique words of an input text corpus. (level 1)

Step 2: The next group of 2m (m=2) unique words are then coded by adding prefix pair of bits 11 and 00, to every code generated in the previous step. (level 2)

Step 3: Using the above steps, we generalize the coding process as: for the next level, the remaining 2m unique words are going to code by adding prefix pair of bits 11 and 00 to every code generated in the previous step. (level m)

Step 4: Step 1– 3 are run again and again until every one of the words is encoded. Precedent 4 shows the coding strategy.

Precedent 4: Let the given content T = "this person is young, the way an actual young person is young". We apply the WBTC packing technique to the above content T, to find the compressed content T′. Table 4 shows the codes assigned to all the unique words. The of the content T is as per the following:

T′ = 000001 0001 10 01 001101 1101 1110 0010 000010 01 0001 10 01

WBTC takes 48 bits for representing the above content T. As we see in precedent 2, the same content T is represented by 40 bits using Huffman compression for words, but that Huffman compression suffers from the problem of false matching as shown in precedent 3, whereas WBTC doesn't. So, WBTC is an effective and efficient compression technique to compress a large amount of data.

Table 4: Word-based Tagged Code for each word

Words	Frequencies	WBTC Codes
young	3	01
is	2	10
person	2	0001
an	1	0010
the	1	1101
way	1	1110
this	1	000001
actual	1	000010
,	1	001101

Matching stage: Matching request for word W inside the compressed content has fulfilled by utilizing the following steps:

Step 1: First, we find the code C of W, which is assigned by the WBTC coding process.

Step 2: Now, using code C, we perform CPM.

Bits (10/01) are utilized as a flag to spot the finish of a code word. Because of this, the concatenation of bits (11/00) to form the code of next level words becomes prefix-free that is no two words in the vocabulary have the same start and end pair of codes. Therefore, codes provided by WBTC are free from false matching. Suppose substring p is postfix of string r (represented as p!r) such as r = qp where $q \in \Sigma^*$ and $|p| \leq |r|$. For instance, suppose r is given as r = bccdaaabcaca, and from r, we pick substring p = bcaca. Since p is postfix of string r, then it is represented as p!r. Now suppose c1 and c2 are the codes assigned to r and p respectively as c1 = 0011001101 and c2 = 1101. Here, c2 is the postfix of c1. So, we can deduct that codes assigned by WBTC packing are not postfix free, so that we must confirm that the found match is a genuine match or not and we can verify the same as follows:

Assume we have found a matching of any word at position "i" in the packing content. To verify it, we must check the successive pair of bits before this i^{th} position. If we found that the past two bits are 10/01, then the match is genuine. Just the same, if we found that the past two bits are 11/00, then the match isn't genuine. Identification of the

total number of matches and false matches are possible. For instance, consider we match the position of pattern P = 'young'. First, we find its WBTC code that is 01. Next, we match the code 01 directly in the compressed content T′ such as:

T′ = 000001 0001 10 01 001101 1101 1110 0010 000010 01 0001 10 01.

As we find that the pattern 'young' (code = 01) matches at three spots in compressed content T′. The other matches with code 01, such as 000001 0001 001101 1101, are false matches because their successive pair of bits are neither 01 nor 10. We choose WBTC coding to develop our algorithm because codes provided by WBTC for each unique word is either end with 01 or 10 and that's why ending pair of bits of any word is never occur at the start of any other word and thus when these codes are implemented using WT then there is very less or no chance of getting false matching results in the process of CPM [23].

2.5. Wavelet Tree

WT [17], is a space-efficient data structure, that represents a sequence and answers queries on it. A WT provides supports for the representation of sequences, reordering of elements, a grid of points, etc. By keeping up the index of content [24], we can recover any content at any point. In [25], [26], authors have presented various types of WT, their execution, and different type of tasks have been performed on it. By using the operations of WT, we can calculate: the number of occurrences (how many times any symbol present in the source content), position (match the exact position of any symbol), show (show the position of the symbol). These essential tasks are performed using two fundamental bitmap operations, such as rank and select. The execution of rank and select operations are briefly discussed in [27]. For any given sequence P (bit-maps), $rank_x(P, i)$ = returns the frequency of symbol x in P [1..i] and $select_x(P, i)$ = returns the index of ith event of symbol x in P. For instance, suppose bitmap is P = 101100110001100, then $rank_1(P, 12) = 6$ and $select_0(P, 6) = 3$.

As given in [28], [29] shows the advantage of compressibility of the content. Various symbols of letters are available at a leaf of a wavelet tree. Insights concerning the construction of WT are given in [17]. WT is a very versatile data structure that can solve the problems in the area of string processing very efficiently using its rank/select operations. WT. There are various advantages and disadvantages of using the WT [26] in indexing. One of the most important advantages of using WT is that its space complexity is O(log log n) and once the tree is generated then the operations like rank/select are performed in O(1) time. The disadvantage of using a WT is that it takes too much time to construct the tree. There are various construction paradigms proposed by different researchers. The construction of the WT is done either sequentially or parallelly, and each one of them has its very own troubles. To overcome this disadvantage of the WT, we try to build the tree parallelly.

There exist bunches of theoretical work with regards to WT-construction. The first introduction of parallelism in the construction of a wavelet tree is given by Jose Fuentes-Sepulveda et al. [30] in 2014. In this work, the authors proposed two linear O(n) time parallel algorithms for the most expensive operation on WT construction using log σ processors. The complexity of the above is O(σ log n) for work and O(n) for depth. In 2015, J. Shun [31] proposed a new parallel WT construction algorithm. They construct the tree level by level that requires O(n) work and O(log n) depth per level. If there are log σ levels in the WT, then the complexity of the above is O(n log σ) for work and O(log n log σ) for depth. Another approach proposed by J. Labeit et al. [32] in 2016 presents a more space-efficient algorithm in comparison to Julian Shun [31] work but achieve the same bounds and complexity as he did. J. Shun implements his algorithm on 40 cores, whereas J. Labeit implemented his algorithm on 64 cores. Further optimization is done by executing the algorithm recursively instead of strictly level-by-level, as done in [31]. In J. Shun [33] improved his work proposed in 2015 by using parallel integer sorting methods and minimizes his work up to $O(n \lceil \log \sigma / \sqrt{(\log \sigma)} \rceil)$ and depth to O(σ + log n). In 2018, Johannes Fischer et al. [34] proposed the fastest sequential as well as the fastest parallel construction of the WT. They divided the text size n into Θ(n/p) and assigned it to each of the p cores of the multicore computer system. Thus, this parallelization of WT computes the WT in O(n) time with O(n log σ) work requiring $4\sigma \lceil \log n \rceil$ bits of extra space in addition to the input and output.

This section focused on how data compression is performed and what is the problem we face when we do compression. Here we see that Huffman code is fast, and it takes less memory in terms of numbers of bit, but it gives false matching results when we do perform CPM. We also see that how WBTC avoids false matching so we implement this compression with the help of a WT and further minimize the time of constructing a WT by using the multicore architecture of the computer. We created an algorithm to minimize the time complexity further and provide faster matching of textual data.

3. Proposed Algorithm and Analysis

As we all know that the size of data is increasing day by day, so the task of matching for any text in a faster way becomes quite challenging. That's why we need an algorithm that can minimize the size of the database, so it takes less space in the storage, and that can also perform faster matching. We use WBTC compression with the combination of WT and construct WT parallelly using the multicore architecture of the computer.

According to various dictionaries average length of a meaningful word is about 15 characters. Hence, we divided the list of unique words from the text corpus into 16 disjoint parts, such as 1st part contains words with a length of one character, 2nd part contains words with a length of two characters, and so on till the 15th part. The 16th part contains

all the words with length greater than fifteen, and because there are very few meaningful words in English with a length of above 15 characters, thus 16^{th} part is rarely taking part in matching. We achieve faster access by constructing separate WTs for each group of words such as for words with length one we create wavelet tree WT1, for words with length two WT2, and so on till WT16 parallelly.

Faster Construction: The constructions of wavelet trees are done parallelly using the multicore architecture of the computer. First, we have already divided the whole text corpus into 16 disjoint parts as discussed above, and second, each disjoint part is assigned to a single core of the multicore system using the parallel for loops. At each core, the fastest sequential wavelet tree construction algorithm is applied, and separate WTs for each part of the corpus is formed. Each part of the corpus is different and independent of each other, so the output of one core does not affect the output of other cores. Figure 1 shows the pre-processing steps of the faster construction of WTs.

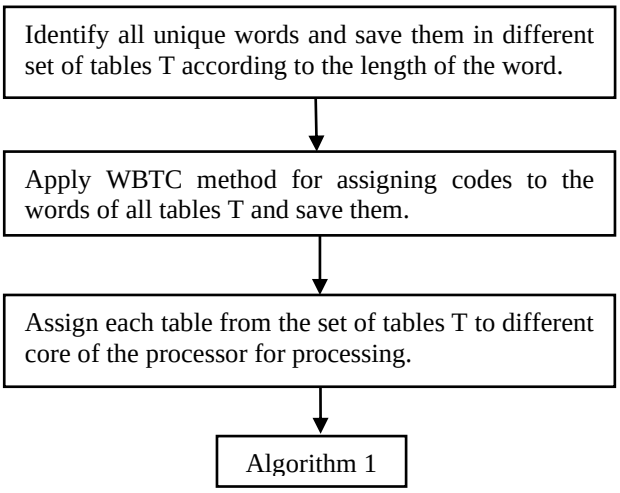

Figure 1. Pre-processing before Construction of WTs parallelly

Algorithm 1: Faster construction

Input:
Total of P term frequency tables. (maximum value of P is 16)
In each table T in P,
Text-words – t1, t2, t3 ……. tn
Code-words – ct1, ct2, ct3 …… ctn

Output:
Total of P wavelet trees

Method:
par-for x = 0 to P-1 do (parallel loop for accessing each table)
for y = 1 to n do (loop executes at each core)
$A_{y,x}$ = find first prefix pair of code-word ct_y
insert $A_{y,x}$ in the root of wavelet tree T_x
curr_node = root (T_x)
z = 2
while (code-word ct_y is not empty) do
$A_{y,z}$ = find the next prefix pair bits from ct_y
If ($A_{y,z-1}$ ={00})
curr_node=left-child (curr_node)
elseif ($A_{y,z-1}$ = {11}) then
curr_node = right-child (curr_node)
else
insert $A_{y,z}$ into curr_node
z = z+1
end while
end for
end par-for.

Faster Matching: Whenever a matching request comes for any query word, then we count the query word's length and load the corresponding WT into the main memory instead of loading the whole WT as previously done by all other matching algorithms. Figure 2 shows the pre-processing steps of faster matching.

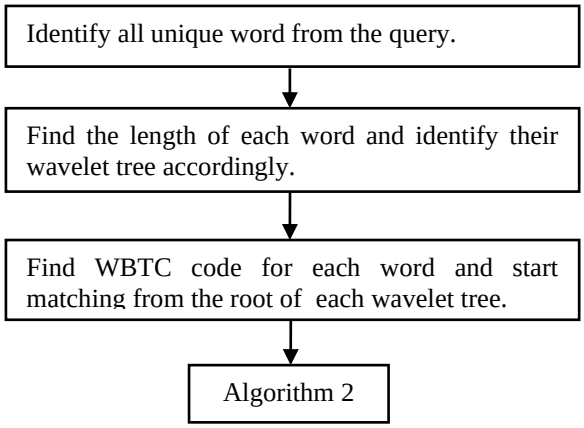

Figure 2. Pre-processing before Matching of query patterns

Algorithm 2: Faster word matching

Input:
code-words ct of each word present in the pattern

Output:
number of occurrences of the words N_{occ} in the corpus

Method:
for each word and their wavelet tree do
B_0 = root (t) // t is one wavelet tree
A_0 = find the first prefix pair from codeword ct
x = 0
while (A_x != {01} and A_x != {10}) do
if (A_x == {00}) then
B_{x+1} = left-child (B_x)
else
B_{x+1} = right-child (B_x)
x = x +1
A_x = find the x^{th} prefix pair bits from ct
end while
N_{occ} = $Rank_{A_x}$ (B_x, |B_x|)
end for

For a better understanding of our proposed approach, we consider the same text content T = "this person is young, the way an actual young person is young". Now we perform the pre-processing before applying algorithm 1 as shown in Figure 1. Each unique word is extracted by their length and stored in the word-frequency table with their WBTC codes.

So, in our case, we have a total of six word-frequency tables, as shown in Tables 5-10, and after applying our algorithm 1 we get six different wavelet trees, as shown in Tables 11-16. Each WTs are going to store on the secondary memory. Now suppose we must match whether the word 'young' is present in the source content T or not and if its present, then what is the frequency of it. To match the word 'young', we first perform pre-processing, as shown in Figure 2, and apply algorithm 2. The length of the word 'young' is five, and its corresponding WBTC code is 01, so we load the corresponding WT into the main memory and match only in the loaded WT instead in all other WTs. In this case, WT for word length of five will be loaded into the memory, and matching is performed from the root of the loaded wavelet tree. In this case, we find that the word is present in the WT by matching WBTC code for the word 'young'. To know the frequency of the word 'young' we further perform the rank operation on bitmap B_0 because the WBTC code is matched in B_0 bitmap as $Rank_{01}(B0, |B0|) = 3$, this means this word appears three times in the source content.

Table 5: Word-frequency table for length one

indexing	word	frequencies	WBTC code
0	,	1	01

Table 6: Word-frequency table for length two

indexing	word	frequencies	WBTC code
0	is	2	01
1	an	1	10

Table 7: Word-frequency table for length three

indexing	word	frequencies	WBTC code
0	the	1	01
1	way	1	10

Table 8: Word-frequency table for length four

indexing	word	frequencies	WBTC code
0	this	1	01

Table 9: Word-frequency table for length five

indexing	word	frequencies	WBTC code
0	young	3	01

Table 10: Word-frequency table for length six

indexing	word	frequencies	WBTC code
0	person	2	01
1	actual	1	10

Table 11: Wt for Table 5

Text:	,
Index:	0
B0:	01

Table 12: Wt for Table 6

Text:	is	an
Index:	0	1
B0:	01	10

Table 13: Wt for Table 7

Text:	the	way
Index:	0	1
B0:	01	10

Table 14: Wt for Table 8

Text:	this
Index:	0
B0:	01

Table 15: Wt for Table 9

Text:	young
Index:	0
B0:	01

Table 16: Wt for Table 10

Text:	person	actual
Index:	0	1
B0:	01	10

Table 17. Running times of algorithms (for fixed alphabet size = 128)

Words in Pattern	File size = 512kb				File size = 1024kb				File size = 2048kb			
	HC	WBTC	WT	TMC_PWT	HC	WBTC	WT	TMC_PWT	HC	WBTC	WT	TMC_PWT
2	13.96	12.54	11.08	10.87	26.97	24.73	21.94	17.30	53.51	50.75	48.97	45.36
4	9.43	8.32	6.97	5.12	19.61	18.07	16.25	14.06	32.67	29.81	28.03	27.07
6	5.85	4.67	3.72	2.53	8.63	7.84	5.39	4.06	15.82	14.56	13.04	11.98
8	3.72	2.14	1.54	0.85	4.06	3.88	2.07	1.35	9.04	8.59	6.33	4.09
10	2.07	1.89	0.98	0.43	2.87	2.14	1.42	0.68	4.02	3.10	2.63	1.57

Table 18. Running times of algorithms (for fixed file size = 1024kb)

Words in Pattern	Alphabet size = 64				Alphabet size = 128				Alphabet size = 256			
	HC	WBTC	WT	TMC_PWT	HC	WBTC	WT	TMC_PWT	HC	WBTC	WT	TMC_PWT
2	29.20	28.04	21.32	16.71	36.42	29.93	22.37	18.02	46.40	44.23	41.71	36.84
4	18.65	17.05	14.81	14.05	20.48	18.41	16.61	14.23	21.29	19.73	17.87	15.06
6	9.73	8.07	5.92	4.82	9.02	8.93	6.16	4.93	10.83	9.63	6.41	5.07
8	4.98	4.02	2.04	0.99	4.28	4.14	2.08	1.28	4.98	3.66	2.19	1.39
10	1.99	1.19	0.98	0.59	3.03	1.88	1.08	0.61	2.25	1.80	1.08	0.68

4. Experimental Setup and Results

For the experiment purpose, we choose 3.40 GHz Intel(R) Xeon(R) CPU E3-1245 processors with 4 GB of primary memory and we have executed all our algorithms on Ubuntu 18.04 LTS. The language used for the algorithms in C++ and all the codes are compiled using the GCC compiler. Our approach consists of two algorithms, first is related to the construction of the wavelet trees, and the second algorithm is related to matching of a query word. So, the overall running time of our approach includes construction time and matching time both. We have performed our experiment on a self-made text corpus and compare the results with previous algorithms that support CPM, such as Huffman coding, WBTC, and simple WT. We have performed our experiment on different sizes of the text corpus first by fixing the alphabet size and varying the file size, and second by fixing the file size and varying the alphabet size. The steps are implemented multiple times for different query words of varying length; we have taken the average value of running time of all algorithms. Table 17 used to show the comparison of methods when we fix the alphabet size and Table 18 shows the comparison of methods when we fix the file size. The performance of our proposed method TMC_PWT is better in comparison to other popular methods that support CPM shown using the running time of all algorithms. Whenever CPM is going to perform for any query then the words present in the query need not be in compressed form. In our proposed approach, when the number of words increases in the query, then we get an improvement in terms of matching time over other CPM methods. Table 19 shows the comparison of our proposed approach to different techniques that support CPM using various compression performance measures.

In our approach, we require some extra space to store various word-frequency tables. Our method combines the advantages of both WT and WBTC. The main advantage of applying a wavelet tree is that it avoids false matching, and it's rank operations are performed in O(1) time. The parallel construction of the wavelet tree is performed in O(n) time with O(n log σ) work and some extra space of $4\sigma \lceil \log n \rceil$ bits. The matching of the word takes O(1) time to complete. Thus, the proposed approach takes a total of O(n) time to perform the construction and the matching of the word.

Table 19: Comparison between methods that supports CPM

Compression Performance Measure				
Methods	**Ratio**	**Speed**	**Memory Needs**	**CPM support**
Huffman coding (HC)	Good	Medium	Low	Yes (with false match)
WBTC	Good	Fast	High	Yes (with false match)
Wavelet tree (WT)	Good	Very Fast	Low	Yes (without a false match)
TMC_PWT (Proposed)	Better	Very Fast	Medium	Yes (without a false match)

5. Conclusion and Future work

In this paper, a new methodology is proposed for matching the word in the text corpus. This paper presents an

improvement of matching time in comparison to other algorithms. The proposed algorithm saves disk space using WBTC compression, provide faster matching using WT and improves the system performance by minimizing the number of page-faults when performing matching using our proposed approach of dividing the indexes into several WTs.

We have utilized the WBTC packing technique with the help of a wavelet tree. We have minimized the construction time of WT, that is the main disadvantage of using WTs using parallel processing. We have partitioned the whole text corpus into 16 parts using the length of words in the corpus and construct a unique WT for each partition. The advantage of dividing whole text corpus and constructing several WTs based on the length of words is that in our approach we load only those indexes into the memory that are needed in the matching process instead of loading the whole indexes. This algorithm returns whether the query word is present in the corpus or not and what's the frequency of it. This saves a lot of time wasted during loading all the WTs from secondary memory into primary memory. Since the size of the main memory is less in comparison to the size of indexes saved in secondary memory; thus, our proposed approach will minimize the number of page faults and increase overall system performance. This approach takes less memory in the system and quite efficient in order of matching time of O(n).

In the future, we use Machine Learning (ML) multi-classification models to find unique words and their word frequency tables. We also use ML models for assigning the codes to all unique words and try to construct the wavelet tree because, by using ML, we can minimize the additional memory needs of our proposed algorithm.

References

[1] D. Yadav, A. Singh, and V. Jain. Search results optimization. *Communications in Computer and Information Science*. 2011, vol. 168 CCIS, pp. 325–334.

[2] R. Beal and D. Adjeroh. Compressed parameterized pattern matching. *Theor. Comput. Sci.*. 2016, vol. 609, pp. 129–142.

[3] S. P. Mishra, C. G. Singh, and R. Prasad. A review on compressed pattern matching. *Perspect. Sci.*. 2016, vol. 8, pp. 727–729.

[4] R. Khetan, S. Agarwal, and R. Prasad. An efficient approach towards compressed parameterized word matching using wavelet tree. *J. Inf. Optim. Sci.*. 2016, vol. 37, no. 2, pp. 285–301.

[5] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. Comput. Syst. Sci.*. 1996, vol. 52, no. 2, pp. 299–307.

[6] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica (New York)* 1998, vol. 20, no. 4, pp. 388–404.

[7] Y. Lifshits. Processing Compressed Texts: A Tractability Border. *Combinatorial Pattern Matching*, LNCS, Berlin, Heidelberg: Springer Berlin Heidelberg. 2007, vol. 4580, pp. 228–240.

[8] E. S. de Moura, N. Ziviani, G. Navarro, and R. Baeza-Yates. Fast searching on compressed text allowing errors. *SIGIR Forum (ACM Spec. Interes. Gr. Inf. Retrieval)* 1998, pp. 298–306.

[9] T. Eilam-Tzoreff and U. Vishkin. Matching patterns in strings subject to multi-linear transformations. *Theor. Comput. Sci.*. 1988, vol. 60, no. 3, pp. 231–254.

[10] D. Knuth, J. Morris, Jr, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*. 1977, vol. 6, no. 2, pp. 323–350.

[11] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*. 1977, vol. 20, no. 10, pp. 762–772.

[12] A. Mukherjee. Compressed pattern-matching. *Proc. IEEE Data Compression Conf.*. 1994, no. March, pp-468.

[13] N. Ziviani, E. S. De Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *Proc. IEEE Comput.*. 2000, vol. 33, no. 11, pp. 37–44.

[14] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Syst.*. 2000, vol. 18, no. 2, pp. 113–139.

[15] S. Gupta, and A. Agarwal. A Scheme that Facilitates Searching and Partial Decompression of Textual Documents. *Int. J. Adv. Comput. Eng.*. 2008, vol. 1, no. ii, pp. 99–109.

[16] R. Gupta, A. Gupta, and S. Agarwal. A novel approach of data compression for dynamic data. *IEEE International Conference on System of Systems Engineering*. 2008, pp. 1–6.

[17] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*. 2003, pp. 841–850.

[18] A. K. Yadav, D. Yadav, and R. Prasad. Efficient Textual Web Retrieval using Wavelet Tree. *Int. J. Inf. Retr. Res.*. 2016, vol. 6, no. 4, pp. 16–29.

[19] A. K. Yadav, D. Yadav, and D. Rai. Efficient methods to generate inverted indexes for ir. *Advances in Intelligent Systems and Computing*. 2016, vol. 435, pp. 431–440.

[20] A. Yadav and D. Yadav. Wavelet tree based hybrid geo-textual indexing technique for geographical search. *Indian J. Sci. Technol.*. 2015, vol. 8, no. 33,

pp. 1–7.

[21] A. K. Yadav and D. Yadav. Wavelet tree based dual indexing technique for geographical search. *Int. Arab J. Inf. Technol.*. 2019, vol. 16, no. 4, pp. 624–632.

[22] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proc. IRE.*. 1952, vol. 40, no. 9, pp. 1098–1101.

[23] S. P. Mishra, R. Prasad, and G. Singh. Fast Pattern Matching in Compressed Text using Wavelet Tree. *IETE J. Res.*. 2018, vol. 64, no. 1, pp. 87–99.

[24] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Inf. Retr. Boston.*. 2012, vol. 15, no. 6, pp. 527–557.

[25] N. R. Brisaboa, Y. Cillero, A. Fariña, S. Ladra, and O. Pedreira. A new approach for document indexing using wavelet trees. *Proc. - Int. Work. Database Expert Syst. Appl. DEXA*. 2007, pp. 69–73.

[26] G. Navarro. Wavelet trees for all. *J. Discret. Algorithms*. 2014, vol. 25, pp. 2–20.

[27] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2008, vol. 5280 LNCS, pp. 176–187.

[28] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*. 2007, vol. 3, no. 2, pp. 1–25.

[29] G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *ACM Comp. Surv.*. 2007, vol. 39, pp. 1–61.

[30] J. Fuentes-Sepúlveda *et al.* Efficient Wavelet Tree Construction and Querying for Multicore Architectures. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*. 2014, vol. 1, pp. 150–161.

[31] J. Shun. Parallel Wavelet Tree Construction. *Data Compression Conf. Proc.*. 2015, vol. July, pp. 63–72.

[32] J. Labeit, J. Shun, and G. E. Blelloch. Parallel lightweight wavelet tree, suffix array and FM-index construction. *J. Discret. Algorithms*. 2017, vol. 43, pp. 2–17.

[33] J. Shun. Improved Parallel Construction of Wavelet Trees and Rank/Select Structures. *Data Compression Conf. Proc.*. 2017, vol. Part F1277, pp. 92–101.

[34] J. Fischer, F. Kurpicz, and M. Löbel. Simple, Fast and Lightweight Parallel Wavelet Tree Construction. *Proc. Twent. Work. Algorithm Eng. Exp.*. 2018, pp. 9–20.

[35] K. Fredriksson and M. Mozgovoy. Efficient parameterized string matching. *Inf. Process. Lett.*. 2006. vol. 100, no. 3, pp. 91–96.

[36] S. T. Klein, D. Shapira, E. S. De Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Compressed Matching in Dictionaries. *ACM Trans. Inf. Syst.*. 2011, vol. 18, no. 1, pp. 61–74.