Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2017) 25: 3607 – 3622
© TÜBİTAK
doi:10.3906/elk-1601-92

A new word-based compression model allowing compressed pattern matching

Halil Nusret BULUŞ[1,*], **Aydın CARUS**[2], **Altan MESUT**[2]
[1]Department of Computer Engineering, Çorlu Engineering Faculty, Namık Kemal University, Çorlu, Tekirdağ, Turkey
[2]Department of Computer Engineering, Faculty of Engineering, Trakya University, Edirne, Turkey

Received: 11.01.2016 • **Accepted/Published Online:** 20.06.2017 • **Final Version:** 05.10.2017

Abstract: In this study a new semistatic data compression model that has a fast coding process and that allows compressed pattern matching is introduced. The name of the proposed model is chosen as tagged word-based compression algorithm (TWBCA) since it has a word-based coding and word-based compressed matching algorithm. The model has two phases. In the first phase a dictionary is constructed by adding a phrase, paying attention to word boundaries, and in the second phase compression is done by using codewords of phrases in this dictionary. The first byte of the codeword determines whether the word is compressed or not. By paying attention to this rule, the CPM process can be conducted as word based. In addition, the proposed method makes it possible to also search for the group of consecutively compressed words. Any of the previous pattern matching algorithms can be chosen to use in compressed pattern matching as a black box. The duration of the CPM process is always less than the duration of the same process on the texts coded by Gzip tool. While matching longer patterns, compressed pattern matching takes more time on the texts coded by compress and end-tagged dense code (ETDC). However, searching shorter patterns takes less time on texts coded by our approach than the texts compressed with compress. Besides this, the compression ratio of our algorithm has a better performance against ETDC only on a file that has been written in Turkish. The compression performance of TWBCA is stable and does not vary over 6% on different text files.

Key words: Compression, pattern matching, compressed pattern matching, semistatic model

1. Introduction

Although data compression was an old notion, the entropy concept was first introduced in the work by Claude E. Shannon named *A Mathematical Theory of Communication* [1] in 1948. Besides this, Shannon introduced the mathematical background of coding theory. In the information theory area of computer science, data compression or source coding is storing information by using fewer bits. Nowadays, data compression is used in operations such as increasing the data transfer ratio or reducing the occupied storage space. Thus, many lossless and lossy data compression methods were developed. In lossless data compression, data are coded and when they are decoded the original data are acquired [2]. However, in lossy compression, data are compressed with some acceptable loss in original data. When decoding is done the original data are acquired with some acceptable loss [3].

Pattern matching algorithms have a significant role in the fields that information retrieval is used in. In programming methods used in computer science fields like system and software design, pattern matching

*Correspondence: nbulus@nku.edu.tr

algorithms are widely used as well. Furthermore, pattern matching algorithms are used in the fields like search engines, molecular biology, and image recognition systems.

Pattern matching without decoding the compressed data or with partial decoding on compressed data is called compressed pattern matching (CPM).

The CPM problem was formally defined by Amir and Benson [4]. Let c be a given compressor, $\sigma = s_1 \ldots s_u$ be a text string of length u over alphabet $\sum \{a_1, \ldots, a_q\}$, $c(P)$ be the compressed form of the pattern $P = p_1 \ldots p_m$, and $\sigma . c = t_1 \ldots t_n$ be the compressed form of σ of length $n \leq u$.

Input: Compressed text $\sigma . c = t_1 \ldots t_n$ and compressed pattern $c(P)$

Output: All locations in σ where pattern P occurs [5,6]

Let the text $\sigma . c$ be the compressed form of text T and pattern is P; the compressed pattern matching problem finds all occurrences of $c(P)$ in T by using only $\sigma . c$ and $c(P)$. In normal compressed pattern matching this process was taking $O\ (u + m)$ time, where $u = |T|$ and $m = |P|$. However, in an optimal CPM algorithm this process should take $O\,(n + m)$ time, where $n = |\sigma . c|$ [7].

The main goal of the pattern matching algorithms (whether they are compressed or not) is to find all the patterns in the text with minimum processing time and minimum additional memory usage. Pattern matching on compressed data is generally faster than on uncompressed data because it works on a smaller size. Many researchers in the literature have found inferences on time complexity rather than complexity of CPM in their works [4,5,7].

In this paper, the presented method allows existing algorithms to perform CPM without any major change. It makes the CPM process available with some small modifications on them.

In a dynamic compression algorithm, the compressed form of a word or a phrase varies in different positions of a file. This may lead to an improvement or a revision on CPM to update the codeword, which has been searched according to the position of the compressed text. As opposed to this, TWBCA has been designed as a semistatic model to avoid this update process.

Our approach has two passes on data to be compressed. In the first pass, the algorithm builds a new alphabet and a dictionary. In the second pass, the data are compressed via this dictionary and alphabet. Hence, this approach can be thought of as a semistatic method.

In section 2 of the paper, previous studies on CPM are given. Algorithm details in the compression and decompression phase and some definitions are given in section 3. Section 4 has the experimental results of the proposed algorithm in compression duration, decompression duration, compressed pattern matching duration, and compression rates.

2. Related works

This section gives a brief summary of compression algorithms to which CPM can be applied directly or with some little modifications to them. We also give brief descriptions of some new compression algorithms that have been designed for CPM.

While some works [7–9] deal with the problem of CPM over known compression algorithms, some other studies [10–17] present a new compression algorithm that is suitable for CPM.

LZ77 and LZ78 are the best-known dictionary-based compression algorithms, developed by Lempel and Ziv in 1977 [18] and 1978 [19]. Navarro and Raffinot presented a general method to search in texts that are compressed with LZ77 and LZ78 algorithms [7]. They also gave a hybrid compression technique that allows searching as fast as in LZ78 and as good to compress as LZ77 in the paper.

The LZW (Lempel–Ziv–Welch) algorithm is based on LZ78 and was presented by Welch in 1984 [20]. Navarro and Tarhio proposed a new CPM method that can be used over texts that are compressed with LZW-based Unix Compress [8].

Byte pair encoding (BPE), which was described by Philip Gage in 1994 [21], is a simple algorithm that uses digram coding recursively. BPE's compression speed is somewhat slower than LZW's, but its decompression is faster. Although BPE cannot become a popular algorithm, Shibata et al. have shown the advantage of BPE from the point of view of CPM [9].

End-tagged dense code (ETDC) and (s,c)-dense code (SCDC) are word-based semistatic compression methods that were proposed by Brisaboa et al. in 2003 [10,11]. Because they use a word-based strategy, both of them are suitable for CPM especially when searching words in a compressed text. Dynamic ETDC (DETDC) and dynamic SCDC (DSCDC), which were developed a few years later, are better than semistatic ones in terms of compression speed, but they are worse in terms of decompression speed [12]. The subsequently developed dynamic lightweight ETDC (DLETDC) and dynamic lightweight SCDC (DLSCDC) are able to perform decompression as fast as semistatic dense codes [13]. Although the compression speed of dynamic lightweight dense codes is worse than that of dynamic dense codes, it is still better than that of semistatic dense codes. On the other hand, dynamic lightweight dense codes are not better than dynamic dense codes, and dynamic dense codes are not better than semistatic dense codes in terms of compression ratio.

Carus and Mesut created a compression algorithm that uses a semistatic model and a dictionary that contains the most frequently used trigrams [14]. The main dictionary consists of several subdictionaries that contain the most frequently used trigrams after a particular digram. The algorithm selects the corresponding subdictionary by looking at the last encoded digram. They also showed that word-based compression methods like dense codes are not efficient for agglutinative languages like Finnish and Turkish, because the number of different words in these languages is more than those of other languages.

In the study by Horspool and Cormack in 1992 [15], the idea was using words in text as a dictionary entry. This approach was applied to the LZW compression algorithm to make it word based. The main difference of our approach from this work is in our study the alphabet and the generated tries are composed of characters, not words.

In the study by Zhang et al. in 2004 [16], the LZW compression algorithm has been converted to an algorithm that performs compression in two passes similar to the algorithm in this paper. However, our algorithm has a separator mechanism at the end of words and so it is a word-based algorithm and, apart from this disparity, our algorithm uses two-byte-long codewords, but in LZW and [16] this codeword length is not a definite value; it is parametric. Moreover, our algorithm has differences in pattern matching phase from [16]. The first difference is using one of the codeword bytes as a tag value instead of writing additional tag values on the output text. In standard LZW or LZ78 the compressed form of a phrase is changing during the compression phase since new entries are made while reading a phrase. In our approach only one compressed form of a phrase (a word with a separator) is used in the compression phase. This form changes neither at the beginning nor at the end of the file. Our compression schema is a two-pass schema and compression is performed in the second pass by using the output of the first pass, and so only the control of byte numbers is done instead of doing a partial decompression.

de Moura et al. presented an efficient compression technique for natural language texts that allows fast and flexible searching of words and phrases [17]. Our approach has similarities with this work, like both of them have byte-based coding and word-based pattern matching. However, our work is not a variant of Huffman coding as it is in [17].

3. Algorithm details

Our algorithm uses a text factorization as LZ78 does. The main rule of factorization in LZ78 and variants of LZ78 is constructing a prefix tree. Let $\sum$ be a finite alphabet and an element of $\sum^*$ is called a string. The LZ78-factorization of a string S is the factorization $f_1 ... f_m$ of S, where each LZ78-factor $f_i \in \sum + (1 \leq i \leq m)$ is the longest prefix of $f_1 ... f_m$, such as $f_i \in \{f_j c | 1 \leq j < i c \epsilon \sum\} \cup \sum$ [22]. According to this definition our algorithm and LZ78 construct the dictionary in the same way. The disparities of our algorithm from LZ78 are found in the output codeword. The output codeword of LZ78 has two parameters. The first parameter of the LZ78 output is a dictionary pointer and the second parameter is a character added to the longest prefix. However, our algorithm has an output codeword that includes only a dictionary pointer. In that sense it acts like LZW. Besides this, our algorithm has a one-byte-long codeword for uncompressed characters. Moreover, as a disparity from LZ78, our algorithm has a dictionary including all characters in the text in the initial case.

The dictionary construction, new code assignments of the characters, and the compression phase of the proposed algorithm are explained in this section.

In the first pass of the algorithm, the modeler performs two important tasks. The first of these tasks is constructing a new alphabet that is composed of all different characters used in the data and the second is constructing a dictionary by using appropriate strings. In the following section, these two tasks are defined separately but in practice these tasks are taken simultaneously in the first pass.

As seen in Figure 1 the modeler of the method constructs the dictionary based on original text in the first pass (Phase I). In the compression phase (Phase II) codewords of phrases in the original text are found in the dictionary ignoring the word boundaries. In the decompression phase (dashed line) a one- or two-byte-long phrase depending on the tag value is read from the compressed text as dictionary index. The string in the node of the dictionary is written as a decompressed phrase.

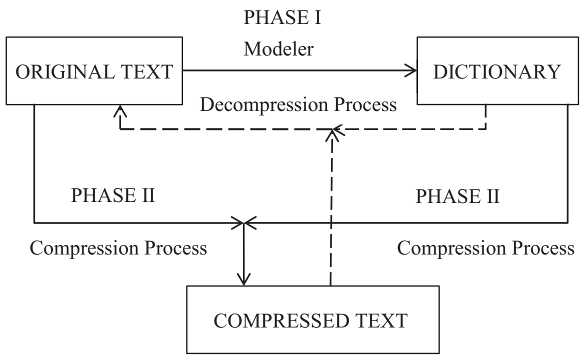

Figure 1. A general schema for two phases of the proposed model.

3.1. Alphabet construction

On a string $\sigma = s_1 ... s_u$ of length u over the alphabet $\sum = \{a_1, \ldots, a_q\}$, for every member of the alphabet $\sum = \{a_1, \ldots, a_q\}$, instead of using standard ASCII codes, a new code assignment is used. The assigned codes should take place in $[0, q-1]$ for $q \geq 0$ sequentially. Thus, the codes in $[q, 255]$ can be used with the consecutive byte for strings in the dictionary. Algorithm 1 and Figure 2 represent the steps of constructing alphabet phase of the algorithm.

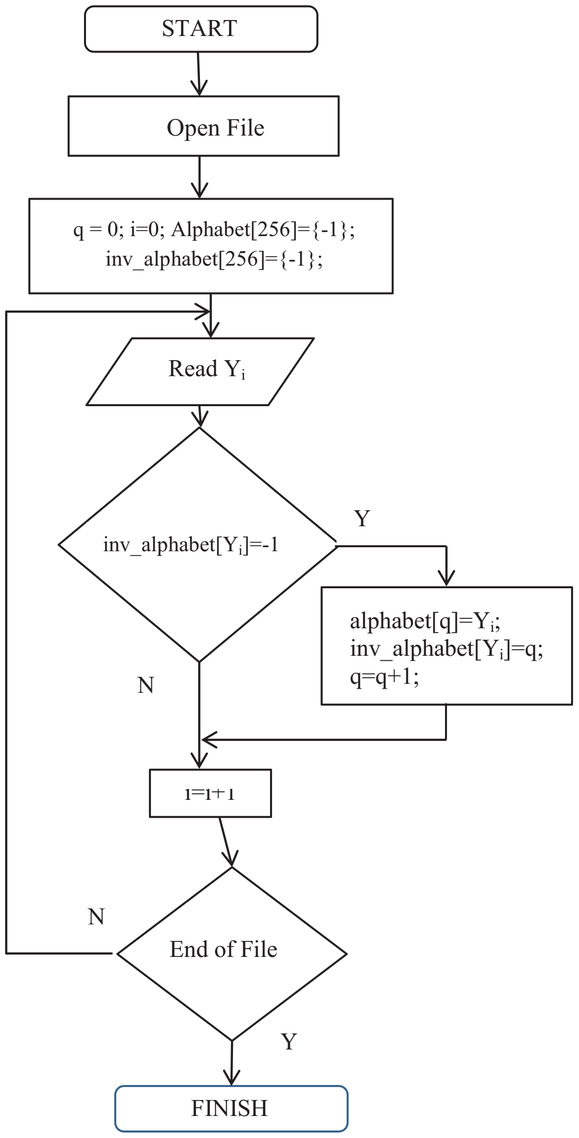

Figure 2. Assigning new code values to characters in data.

Constructing a new alphabet task includes assigning a new code for each different character in one byte range and it allows unassigned values to be used for dictionary entries. This process gives some space in one byte range for dictionary entries.

As seen in Algorithm 1, number of different characters in the text has an important role in dictionary editing. Generally, a natural language text file has approximately 100 different characters (see Table 1). Therefore, we have nearly 155 empty nodes left to store phrases in the dictionary. Although not a common situation, if a text has 256 different characters, then there will be no space for any dictionary entry. This would cause no compression in this algorithm.

Example 1 *Text: abracadabra_abracadabra (_ indicates the space character)*

Algorithm 1 Assigning new code values to characters in data

Input: Text to be compressed

Output: New code values for each member of the alphabet $\sum = \{a_1, \ldots, a_q\}$

```
1: Start initialization process
2: $q = 0$, $i = 0$, $Alphabet[256] = \{-1\}$, $inv\_alphabet[256] = \{-1\}$
3: While not EOF do
4:     Read character $Y_i$ from file
5:     If $inv\_alphabet[Y_i] = \{-1\}$ then
6:         $alphabet[q] = Y_i$
7:         $inv\_alphabet[Y_i] = q$
8:         $q = q + 1$
9:     End if
10:    $i = i + 1$
11: End While
12: End
```

Table 1. Number of different characters in each file used in calculating compression ratio.

File name	Number of different characters	Total number of characters (file size)
dickens.txt	101	10192446
Eng.txt	102	15866246
odtu.txt	126	15828594
bible.txt	64	4445255
britannica.txt	96	1945731
world192.txt	95	2473400

The new alphabet and inverse alphabet is shown respectively below after the text had been read. In initial state it is known that all nodes of the two arrays are empty [6].

alphabet[0] = a, alphabet[1] = b, alphabet[2] = r, alphabet[3] = c, alphabet[4] = d, alphabet[5] = sp;

inv_alphabet[32] = 5, inv_alphabet[97] = 0, inv_alphabet[98] = 1, inv_alphabet[99] = 3, inv_alphabet[100] = 4, inv_alphabet[114] = 2.

3.2. Dictionary construction

Let ω be the word that ends with a delimiter; i is the index of the last read character from word ω. The longest $\omega[p, i]$ prefix is found in the dictionary, where $3 \leq i \leq |\omega|$ and $p = 1$. If $i = |\omega|$ then no addition to the dictionary is made and a new word ω is read from the text. If $i \leq |\omega|$ then K_j character where $j = i + 1$ is attached to $\omega[p, i]$ prefix and string $\omega[p, i]\,K_j$ is added to the dictionary. Then $\omega \leftarrow \omega[j + 1, |\omega|]$ is done and the process is started as if ω is a new word. The algorithm that uses the idea of adding a dictionary entry by attaching a character to a prefix in dictionary is LZW [20]. However, as a difference our algorithm makes a word-based addition to the dictionary.

The dictionary has 65,536 nodes since the output is two bytes long. As additions to dictionary are made according to the size of the alphabet and the maximum number of elements in the alphabet can be 256, each dictionary node has 256 internal nodes.

The point to notice in addition to the dictionary is the situation that $|\omega| > 2$. Since the output is a two-byte-long string, it does not give any positive compression ratio to add the strings that have two characters or less.

Addition to the dictionary starts with three-character phrases. Thus the first two characters are tagged as a digram in the array $digram\,[256]\,[256]$. Then the dictionary index is calculated by using the codes of two characters in a new alphabet. A hash function without any collision is used to calculate the index value. Tagging of a digram is made tagging when the digram is first read. There is no more tagging afterwards.

Example 2 *Text:"abracadabra_"*

Tagged Digrams: "a_", "ab", "br", "ca", "da", "ra"

When the string "abracadabra" is read in the text the second time, the following steps are made. "ab" is the first digram and "a" and "b" take place in the new alphabet. The "ab" digram has been read before since it was tagged. The consecutive character "r" is attached to the digram "ab". Now the "abr" string is searched in the dictionary. To search in the dictionary a hash value of the first two characters has to be calculated by (1).

$$hash\,(\omega_1\,,\omega_2) = alphabet[inv\_alphabet[\omega_1]]\;\times\;256\;+\;alphabet[inv\_alphabet[\omega_2]] \tag{1}$$

This hash value determines the dictionary index to be looked for.

$$index = hash\,(\omega_1\,,\omega_2) \tag{2}$$

After calculating the index value, the node of the character that follows the digram in the indexed dictionary node is searched as to whether it is empty or not. If it is empty, a dictionary adding value named dictionary index that has a value of 65535 as initial state is written to this node. Then 1 is subtracted from the dictionary index value. If the node is not empty this means a dictionary index has been written before; the value in the node is accepted as a new index value. A new character is read from the word and processes are started again with the new index value [6].

Example 3 *Assume that the word "abracadabra" is read a second time in the text.*

The digram "ab" is read. It is seen that "a" and "b" have a new code value and the digram "ab" has been tagged. Now the following character "r" is read and the hash value of "ab" is calculated.

$$hash\,(a\,,\,b) = 0\;\times\;256\;+\;1\;=\;1$$

This value is the index of the dictionary node to be searched. In the dictionary node, if the internal node of character "r" has a value −1, it means the internal node is empty and the string "abr" does not take place in the dictionary. If it is empty the dictionary index (initial value is 65535) is written, and the dictionary index is decreased by 1. In each insertion to the dictionary, this value is decreased by 1.

$$dictionary\,[hash\,(a\,,\,b)]\,[inv\_alphabet[r]]\;=\;65535 \tag{3}$$

After the insertion of the string "abr" to the dictionary, it is removed from the word "abracadabra". The substring "acadabra" is accepted as a new word. The digram "ac" is tagged since it has not been tagged and then it is removed from the word. Also the following digram "ad" is not tagged either. It is tagged and removed. Then the string "abra" remains in the word.

The digram "ab" is tagged. The hash value is calculated from the characters. The internal node "r" of the dictionary node, which has an index number equal to the hash value, now has a value 65535 (3). It means the internal node is not empty. Then the index number is changed to 65535. The following character "a" is read and the internal node "a" in dictionary node numbered 65535 is taken. It is empty and the value 65534 is written there.

dictionary [65535] [*inv_alphabet*[*a*]] = 65534. Algorithm 2 and Figure 3 represent the steps of the dictionary insertion phase of the algorithm.

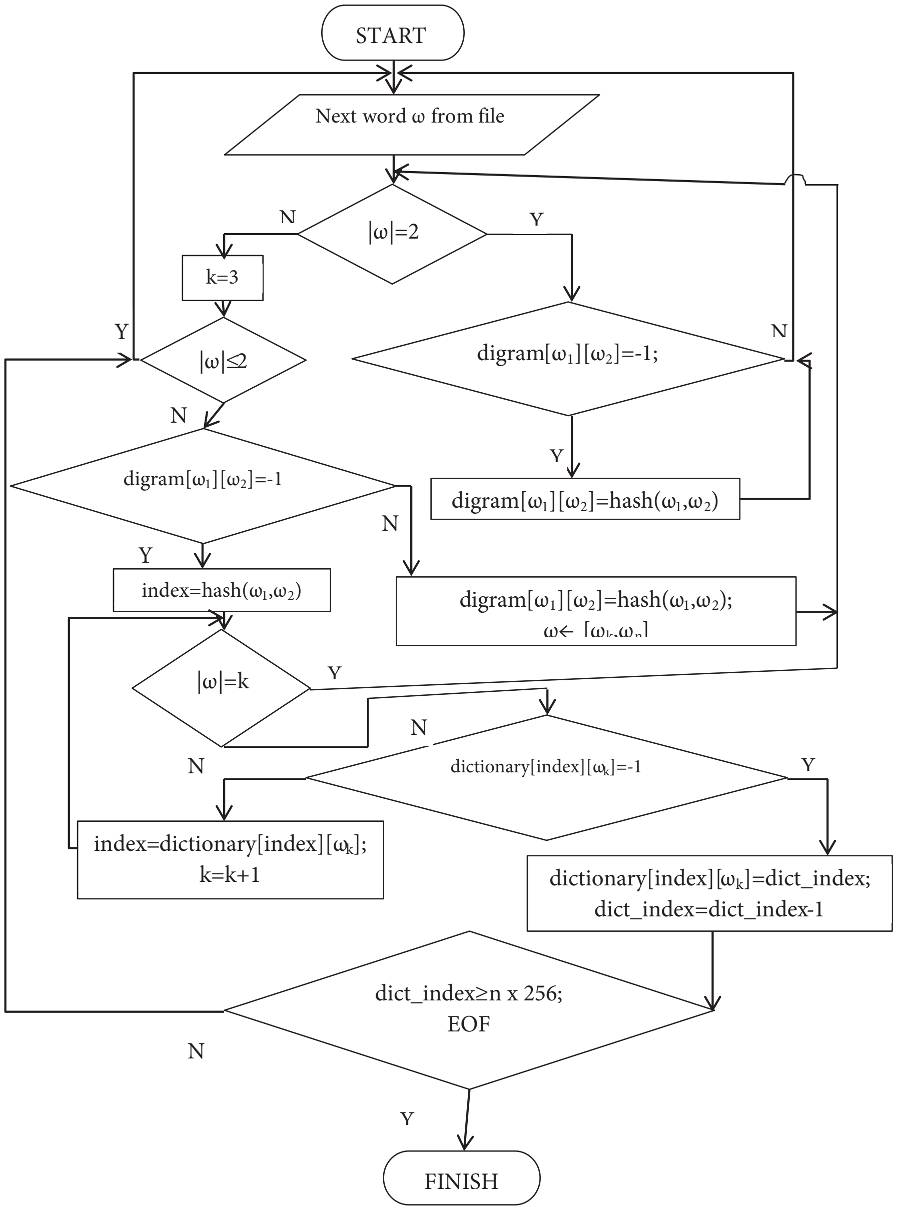
START
Next word ω from file
|ω|=2
N
Y
k=3
|ω|≤2
digram[ω1][ω2]=-1;
digram[ω1][ω2]=hash(ω1,ω2)
digram[ω1][ω2]=-1
index=hash(ω1,ω2)
digram[ω1][ω2]=hash(ω1,ω2); ω← [ωk..ωn]
|ω|=k
dictionary[index][ωk]=-1
index=dictionary[index][ωk]; k=k+1
dictionary[index][ωk]=dict_index; dict_index=dict_index-1
dict_index≥n x 256; EOF
FINISH

Figure 3. Dictionary insertion.

Algorithm 2 Dictionary Insertion

Input: Text to be read to construct dictionary, n to be the number of different characters in T.

Output: Dictionary used in compression

```
1: Start
2: While not EOF or $dict\_index \geq n \times 256$ do
3:   Read next word $\omega$ from file
4:   If $|\omega| = 2$ then
5:       If $digram[\omega_1][\omega_2] = -1$ then
6:          $digram[\omega_1][\omega_2] = hash(\omega_1, \omega_2)$
7:       End if
8:       go to step 2
9:   Else
10:          $k = 3$
11:          If $|\omega| \leq 2$ then
12:              go to step 2
13:            Else
14:              If $digram[\omega_1][\omega_2] = -1$ then
15:                 $digram[\omega_1][\omega_2] = hash(\omega_1\ \omega_2)$
16:                 $\omega \leftarrow [\omega_k, \omega_n]$
17:                     go to step 4
18:                  Else
19:                     $index = hash(\omega_1\ \omega_2)$
20:                     If $|\omega| = k$ then
21:                        go to step 4
22:                     Else
23:                          If $dictionary[index][\omega_k] = -1$ then
24:                             $dictionary[index][\omega_k] = dict\_index$
25:                             $dict\_index = dict\_index - 1$
26:                        Else
27:                             $index = dictionary[index][\omega_k]$
28:                             $k = k + 1$
29:                        go to step 20
30:                  End if
31:              End if
32:          End if
33:      End if
34:  End While
35: End
```

3.3. Compression phase

In the second pass or in other words in compression phase, there is no need to read the data in a word-based manner since the dictionary has been constructed in a word-based manner. The compression phase is quite simple. A character is read and attached to the string until the string has no dictionary node. When an empty node is found, the compressed output is the two-byte code of the value of the last dictionary index that the string has [6]. Algorithm 3 and Figure 4 represent the compression phase of the algorithm.

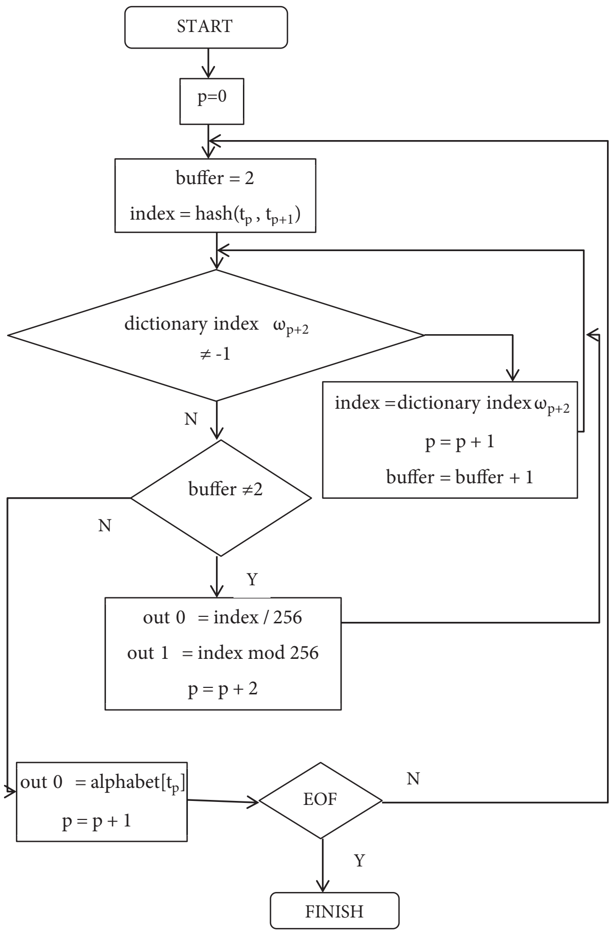

Figure 4. Compression phase.

The compression phase is started after the modeler constructs arrays and dictionary. The same string "abracadabra_abracadabra" is compressed through the following steps.

- The digram "ab" is read. It is a tagged digram and so the hash value is calculated.
- The following character "r" is searched in the dictionary that has an index calculated by the hash function.
- Node is not empty. It has value 65535. Thus this value is a new index value to look for.
- The value 65535 expresses the string "abr". The following character "a" is looked for in this index.
- The node 0 of the index 65535 ("abra") has a value 65534. New index value is 65534 from now on. The following character "c" is read.

Algorithm 3 Compression Phase

Input: T be the text to be compressed

Output: Compressed text

```
1: Start initialization process $p = 0$
2: While not EOF do
3:    $buffer = 2$
4: $index = hash(t_p\ t_{p+1})$
5:    If not $dictionary[index][\omega_{p+2}] = -1$ then
6:    $index = dictionary[index][\omega_{p+2}]$
7:    $p = p + 1$
8:    $buffer = buffer + 1$
9:    go to step 5
10:   Else
11:      If not $buffer = 2$
12:      $out[0] = index / 256$
13:      $out[1] = index\, mod\, 256$
14:      $p = p + 2$
15:      go to step 4
16:   Else
17:      $out[0] = alphabet[t_p]$
18:      $p = p + 1$
19: End if
20:   End if
21: End While
22: End
```

- The code value in new alphabet for "c" is 3. The node 3 of the index 65534 has value –1, which means it is empty. Thus the string "abrac" has no entry in the dictionary. The last dictionary index 65534 is coded in two bytes as an output.
- The string "abra" (65534) is removed from the text but the character "c" is not.
- The following character "a" is read and attached to the character "c". Now the processes start from calculating the hash value of "ca".
- "ca" is a tagged digram and the hash value is:

$$hash\,(c, a) = 3 \times 256 + 0$$

- The following character "d" has a code value 4 in the new alphabet. When the node 4 of index 768 of the dictionary is looked up, it is seen that the node has –1 value. Thus the string "cad" is not coded as output.
- The code value 3 for the character "c" is coded as an output and "c" is removed from text.
- These operations are performed for the whole text to achieve the compression process.

The text is compressed by the end of the second pass. The length of the dictionary, dictionary entries, and new alphabet are written into the compressed text.

In this study a structure similar to de Moura's work [17] in 2000 named spaceless word model is used. It is assumed in [17] that each word is followed by a space character, and so in the coding phase these space characters can be ignored. If a word is followed by a separator character, word and character are coded separately. In our study, each word that has different separators is treated as a different word, but the common parts of them take place only once in the dictionary.

In our approach, the first byte value of the output is greater than the number of different characters used in text. This gives us a tag value to perform pattern matching. In addition, our approach is a word-based approach and so the name of the approach is tagged word-based compression algorithm (TWBCA) [6].

4. Searching in compressed text

Storing or transferring data in compressed form has some issues to notice. One of them is the necessity of decompressing the metadata/dictionary to process. Especially text data retrieval systems have to allow users to search patterns in text. This issue is an example for pattern matching. For a particular string pair $\{(X(i)\ ,\ Y(i))\}$, if it is possible, an r value that makes $X(r) = Y(r)$ is found [23].

For a native description of the issue, an example for linear pattern matching: Let the pattern and the text be respectively

$$X = x_1 x_2 \ldots x_n \qquad\qquad x_i \in \{0,1\}$$

$$Y = y_1 y_2 \ldots y_m \qquad\qquad y_i \in \{0,1\}$$

and

$Y(i) = y_i y_{i+1} \ldots y_{i+n-1}$. If $X = Y(i)$ exists, a matching occurs [23].

Let m be the compressed text, ω be a word from uncompressed text, n be the number of different characters in the text, and *alphabet*[] be the new code list of characters in the text. Thus the maximum index of the list is $(n - 1)$.

Also let c_j be an uncompressed character in compressed text in position j, where $0 \le j \le |m|$, and f_i is a digram code of a compressed string in position i, where $0 \le i \le |m|$. The code f_i is a digram or in other words it is composed of two bytes. The values of these bytes would be respectively $n \le f_{i0} \le 255$ and $0 \le f_{i1} \le 255$; in addition, c_j would be $0 \le c_j \le n-1$.

From the expressions above, any c_j may have a same value with any f_{i1}, but f_{i1} code may have different values than c_j in the range of $[n\,,\,255]$. Moreover, f_{i1} code may have the same values with f_{i0} in the range of $[n\,,\,255]$. However, there are no common values between c_j and f_{i0}. Hence this is the control mechanism for false matching of the pattern.

Therefore, a combination of a compressed form of a word ω_z in the position z and compressed form of ω_{z-1} may be a compressed form of word ω_p.

Here is an example: assume n = 100 and the words of the string "the real world" respectively have the compressed codes

the_ : 130 65, real_ : 192 127, world : 95 208 160

In this situation, the string "the real world" has a compressed output "130 65 192 127 95 208 160". A pattern "world" would be coded as "95 208 160" and easily found in 5th position.

Let a different string, e.g., “compress” have a “127 95” value in compressed form. It can be seen in 4th position of the coded string above. It would match although it does not exist in the string “the real world”.

When a pattern matching occurs, the number of the bytes previously consecutive to the position whose value is in the range $[n, 255]$ is counted. If an odd number is found, it is obviously seen that a comparison with f_{i1} has been made since the code of any c_j cannot be equal to any f_{i0} code. This means a false match has been made. A routine that counts the previously consecutive bytes is added to pattern matching algorithms to overcome the false match problem.

5. Experimental results

The approaches used in comparison are ETDC, Compress, and Gzip. ETDC performs word-based compressed pattern matching since it performs word-based compression [10].

One of the compared compressors in our experiments is Compress, which is the standard compression utility of the UNIX operating system. The LZW compression algorithm is the basis of it. Software called Lzgrep is used to perform CPM on the files compressed with Compress [8].

In addition to these utilities, Gzip has been used for experiments. Gzip is a standard UNIX data compression utility as well. Lzgrep is also used for compressed pattern matching on files compressed with Gzip.

odtu.txt, which has been used in experiments in this section, is the work of Say et al. in 2002 [24].

The test files bible.txt and world192.txt are from the Canterbury Corpus, dickens is from the Silesia Corpus, and britannica.txt is from Project Gutenberg. The file named Eng.txt is the combination of the files “1musk12.txt”, “alice29.txt”, “asyoulik.txt”, “bible.txt”, and “dickens.txt”.

In Table 2, the averages of durations of 10 tries for different groups of patterns on 2 different files are given. The same pattern groups and the same text files were used for 4 different algorithms.

Table 2. The average compressed pattern matching durations of patterns grouped by number of characters they have (s).

	File name	ETDC	Lzgrep/Gzip	Lzgrep/Compress	TWBCA
15 Char. 75 Patterns	Eng.txt	0.0081	0.1688	0.0899	0.1300
	odtu.txt	0.0136	0.1770	0.1028	0.1329
12 Char. 100 Patterns	Eng.txt	0.0119	0.1709	0.0889	0.1135
	odtu.txt	0.0136	0.1786	0.1009	0.1419
9 Char. 100 Patterns	Eng.txt	0.0124	0.1736	0.0944	0.1283
	odtu.txt	0.0150	0.1785	0.1033	0.1328
6 Char. 100 Patterns	Eng.txt	0.0125	0.1754	0.1291	0.1274
	odtu.txt	0.0167	0.1813	0.1292	0.1239
3 Char. 100 Patterns	Eng.txt	0.0151	0.1803	0.2240	0.1424
	odtu.txt	0.0182	0.1855	0.2333	0.1428

In Table 2 it is seen that TWBCA has better results than Gzip for each pattern group. Besides this, TWBCA has better results than Compress for shorter (3–6 characters) pattern groups. However, in longer pattern groups TWBCA has worse results than Compress. ETDC has the best results for each pattern group.

In Table 3, the compression durations of ETDC, Gzip, and TWBCA for sample files are given. As seen in Table 2, TWBCA has minimum durations against the other two algorithms. This is important for processes that need fast compression.

Table 3. Compression and decompression durations (s).

File name	ETDC		Gzip		TWBCA	
	Comp.	Decomp.	Comp.	Decomp.	Comp.	Decomp.
Eng.txt	0.55	0.20	1.67	0.21	0.41	0.36
odtu.txt	0.79	0.22	1.56	0.20	0.52	0.37
dickens.txt	0.37	0.13	1.15	0.14	0.32	0.24

Table 3 shows the decompression durations of compressed files. As seen in Table 2, TWBCA has the worst results in the decompression process. The dictionary is written at the beginning of the compressed file in compressed form. Each dictionary entry is three bytes long but can express longer strings. These three-byte-long entries are made up with a hash function described in section 3. While decompressing the text, the first thing to do is to extract the dictionary. Each entry is referring to another in the compressed dictionary. Therefore an entry is read many times to get uncompressed strings that use that entry.

Table 4 gives the compression ratio of TWBCA on some texts that are used in testing the compression algorithms. Our approach has a 50% compression ratio on average. An important reason for this is having maximum 256^2 dictionary nodes since the codewords are two bytes long. Moreover, in a text that has n different characters, $n \times 256$ dictionary nodes remain empty according to the method. Furthermore, to add a z-character-long word into the dictionary, the word has to be read $z - 1$ times in the original text, and $z - 2$ prefix entries have been made until the whole word takes place in the dictionary. These reasons reduce the efficiency of the dictionary. On the other hand, as a disparity from other word-based methods, if a word that has not been placed in the dictionary is read, it does not remain in fully uncompressed form and is compressed by using the longest prefix. This gives us a gain in compression ratio. In addition, the experiments show us in Table 4 that our approach is independent of the language in which the text is written [6].

Table 4. Compression ratios of Compress, ETDC, and TWBCA (%).

File name	Original size (bytes)	Compress	ETDC	TWBCA
dickens.txt	10192446	39.32	34.40	49.91
Eng.txt	15866246	38.11	33.87	49.75
odtu.txt	15828594	42.72	50.67	49.04
bible.txt	4445255	30.98	28.85	47.37
britannica.txt	1945731	43.17	43.65	52.90
world192.txt	2473400	37.20	37.48	48.25

The compression ratios of Compress, ETDC, and TWBCA are given in Table 4. While calculating the compression ratio of ETDC, the size of the dictionary that is made for each separate file is added to the compressed file.

The main difference of TWBCA and tagged Huffman [17] is in the phase of constructing the dictionary. TWBCA constructs the dictionary in a character-based way but regarding the word boundary. However, in tagged Huffman complete words are written into the dictionary by sorting them in descending order of frequency. Also in TWBCA codewords have constant length. In contrast, tagged Huffman has Huffman coding manner that is assigning shorter codes to the frequent ones and tagged Huffman has no tagging value like TWBCA to express uncompressed codes.

6. Conclusion

Two main properties make pattern matching effective. One of them is correct matching and the other one is fast matching. Before the 90s, pattern matching was performed only on uncompressed text. After that, many algorithms that perform fast and correct compressed pattern matching were developed or modifications were added to the previous compression algorithms to perform pattern matching. It is seen that most of the developed algorithms or methods are at the center of dictionary-based algorithms. Since our dictionary-based compression algorithm uses a semistatic model, it is efficient for CPM, because using a semistatic model ensures that a particular word is compressed all over the text in the same way. When you use a dynamic model, you may need to store extra information to do so (like in DLETDC). If you do not store such extra information, the compression ratio will be better (like in DETDC). However, the compression ratio of DETDC is still worse than its semistatic counterpart ETDC [12].

Our experiments have shown that our approach has an average compression ratio and average compressed pattern matching durations. Although TWBCA is slow at decompression, it is better than ETDC and Gzip in compression speed, which is one of the efficient algorithm properties.

References

[1] Shannon CE. A mathematical theory of communication. Bell System Technical Journal 1948; 27: 379-423.

[2] Crochemore M, Rytter W. Jewels of Stringology. Singapore: World Scientific Pub. Co. Inc., 2002.

[3] Mesut A. New methods in data compression. PhD, Trakya University, Edirne, Turkey, 2006.

[4] Amir A, Benson G. Efficient two-dimensional compressed matching. In: DCC'92 Data Compression Conference; 24–27 March 1992; Snow Bird, UT, USA: IEEE. pp. 279.

[5] Amir A, Benson G, Farach M. Let sleeping files lie: pattern matching in z-compressed files. Journal of Computer and System Sciences 1996; 52: 299-307.

[6] Buluş HN. Studying of usage of pattern matching algorithms in compressed text data and developing a new approach. PhD, Trakya University, Edirne, Turkey, 2010.

[7] Navarro G, Raffinot M. A general practical approach to pattern matching over Ziv – Lempel compressed text. In: CPM 99 Combinatorial pattern matching: 10th annual symposium; 22–24 July 1999; Warwick University, UK: pp. 14-36.

[8] Navarro G, Tarhio J. LZgrep: a Boyer–Moore string matching tool for Ziv–Lempel compressed text. Software—Practice & Experience Archive 2005; 35: 1107-1130.

[9] Shibata Y, Kida T, Fukamachi S, Takeda M, Shinohara A, Shinohara T, Arikawa S. Byte pair encoding: a text compression scheme that accelerates pattern matching. Technical Report DOI-TR-161; 1999; Dept. of Informatics, Kyushu University.

[10] Brisaboa NR, Iglesias EL, Navarro G, Parama JR. An efficient compression code for text databases. In: ECIR 2003 Advances in Information Retrieval, 25th European Conference on IR Research; 14–16 April 2003; Pisa, Italy. pp. 468-481.

[11] Brisaboa NR, Farinã A, Navarro G, Esteller MF. (S,C)-Dense Coding: an optimized compression code for natural language text databases. In: Symposium on String Processing and Information Retrieval; 8–10 October 2003; Manaus, Brazil. LNCS 2857, Berlin, Germany: Springer-Verlag. pp. 122-136.

[12] Brisaboa NR, Farinã A, Navarro G, Paramà JR. New adaptive compressors for natural language text. Software Pract Exper 2008; 38: 1429-1450.

[13] Brisaboa NR, Farinã A, Navarro G, Paramà JR. Dynamic lightweight text compression. ACM T Inform Syst 2010; 28: 1-32.

[14] Carus A, Mesut A. A new compression algorithm for fast text search. Turk J Elec Eng & Comp Sci 2016; 24: 4355-4367.

[15] Horspool RN, Cormack GV. Constructing word-based text compression algorithms. In: DCC'92 Data Compression Conference; 24–27 March 1992; Snow Bird, UT, USA: IEEE. pp. 62-71.

[16] Zhang N, Mukherjee A, Tao T, Bell T, Satya RV, Adjeroh D. A flexible compressed text retrieval system using a modified LZW algorithm. In: DCC 2005 Data Compression Conference; 29–31 March 2005; Snow Bird, UT, USA: IEEE. pp. 493.

[17] Moura ES, Navarro G, Ziviani N, Baeza-Yates R. Fast and flexible word searching on compressed text. ACM Transactions on Information Systems 2000; 18: 113-139.

[18] Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE T Inform Theory 1977; 23: 337-343.

[19] Ziv J, Lempel A. Compression of individual sequences via variable-rate coding. IEEE Inform Theory 1978; 24: 530-536.

[20] Welch TA. A technique for high-performance data compression. Journal Computer Archive 1984; 17: 8-19.

[21] Gage P. A new algorithm for data compression. C Users Journal 1994; 12: 23-28.

[22] Bannai H, Inenaga S, Takeda M. Efficient LZ78 factorization of grammar compressed text. In: 19th International Symposium, SPIRE 2012; 21–25 October, 2012; Cartagena de Indias, Colombia. pp. 86-98.

[23] Karp RM, Rabin MO. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 1987; 31: 249-260.

[24] Say B, Zeyrek D, Oflazer K, Ozge U. Development of a corpus and a treebank for present-day written Turkish. In: Eleventh International Conference of Turkish Linguistics; August 2002; Near East University, Gazimagusa, Cyprus. pp. 183-192.