Iran Journal of Computer Science
https://doi.org/10.1007/s42044-019-00047-w

ORIGINAL ARTICLE

A dictionary-based text compression technique using quaternary code

Ahsan Habib[1] · M. Jahirul Islam[1] · Mohammad Shahidur Rahman[1]

Received: 24 March 2019 / Accepted: 28 August 2019

Abstract
Improving encoding and decoding time in compression technique is a great demand to modern users. In bit level compression technique, it requires more time to encode or decode every single bit when a binary code is used. In this research, we develop a dictionary-based compression technique where we use a quaternary tree instead of a binary tree for construction of Huffman codes. Firstly, we explore the properties of quaternary tree structure mathematically for construction of Huffman codes. We study the terminology of new tree structure thoroughly and prove the results. Secondly, after a statistical analysis of English language, we design a variable length dictionary based on quaternary codes. Thirdly, we develop the encoding and decoding algorithms for the proposed technique. We compare the performance of the proposed technique with the existing popular techniques. The proposed technique performs better than the existing techniques with respect to decompression speed while the space requirement increases insignificantly.

Keywords Algorithm analysis and design · Data compression · Graph theory · Huffman coding · Source coding · Tree data structures

1 Introduction

Data compression is an important research area not only for saving space but also for reducing query time. Data compression is popular for accumulating data and reducing download, upload and transfer time [1]. Compression and decompression speeds are a very important parameter in data compression techniques. It is monotonous when it requires more time to compress or decompress a file. It has been revealed in the contemporary research that some algorithms achieved more compression ratio by sacrificing the processing speed, whereas some others achieved more speed by sacrificing space. In many cases, authors mainly consider decompression speed, whereas compression speed is also an important performance measuring parameter.

Huffman coding is the most popular and widely used coding system, where compression is done by assigning a shorter code to a symbol with higher frequencies. This technique assigns a unique codeword for each symbol [2]. Huffman-based technique is used in many compression algorithms like Zip, PKZip, BZip2, and PNG. Multimedia codec such as JPEG and MP3 has a front-end model and quantization followed by Huffman coding. Almost all communications with and from the internet are at some points Huffman encoded. Almost all Huffman -based algorithms attempt to improve decoding speed; in most of the cases, they did not achieve very good compression speed.

The existing Huffman-based algorithms use binary code which slow the decoding speed. This paper proposes a new compression algorithm that makes use of a variation of the classic Huffman coding: quaternary Huffman coding. Using quaternary Huffman coding, each symbol is encoded into a quaternary code stream, instead of a binary bit stream. A quaternary code stream for Huffman coding requires a shorter Huffman tree, i.e., less depth. The potential benefit of a shorter Huffman tree is less traverse time, which improves both compression and decompression throughput.

In this research, we analyze the properties of quaternary Huffman tree and conclude that a quaternary Huffman tree is usually one-third of the height from a binary tree. In this research, we further implement a compression/decompression algorithm based on it.

✉ Ahsan Habib
ahabib-cse@sust.edu

M. Jahirul Islam
jahir-cse@sust.edu

Mohammad Shahidur Rahman
rahmanms@sust.edu

1 Department of Computer Science and Engineering, Shahjalal University of Science and Technology, Sylhet, Bangladesh

The datasets that used in this research experiments are some popular corpora like the Canterbury, the Enwik, the SUPara, and the Brown. The detail of the corpora can be found in Sect. 5.1. Both compression ratios and (de)compression throughput of the proposed technique are tested against four other compression algorithms: Lzham, bzip2, Zopfli, and Lzham. The test results show that while the proposed technique produces smaller compression ratios (not as good as others), it produces larger throughputs (better than others).

We organize this manuscript as follows. The literature review has been discussed in Sect. 2. Section 3 presents an overview of quaternary tree architecture. Section 4 discusses the implementation method and Sect. 5 analyzes the methods and results. Finally, Sect. 6 concludes the paper.

2 Literature review

There are two types of data compression: lossy and lossless. Lossy compression usually loses some bits during the decompression process; it also achieves more compression ratio than lossless compression. Lossy compression is effective when missing information does not affect the quality of decoded information. Image, voice and video data are compressed generally by lossy compression technique. Lossy compression can reduce the number of bits by a huge amount but it cannot reproduce the original quality [3]. On the contrary, lossless data compression technique is used for sensitive data as like text. Lossless compression technique ensures 100% reproduction of original data.

Lossless compression techniques are of two types: dictionary based and statistical based. Dictionary-based techniques typically use a variable length code for each symbol in the dictionary. Lampel-Ziv [4,5] and their variants [6,7] are the most well-known dictionary-based techniques. On the other hand, statistical-based compression techniques produce codeword based on the statistical occurrence of a symbol in a file.

Lossless text compression usually replaces an original symbol with a shorter symbol. The content of text in compression can be seen as a pattern of syllables or words [8,9]. Transformation of text to syllables or words is a difficult process because the number of syllables or words is undefined and is always a high number. Moreover, it is also required for all syllables or words to be transmitted to the decoder [10–12]. As a solution, bit level compression is used in text compression, where each character has a specific binary representation [13] which is called codes. Bit level compression occasionally ensures the maximum compression ratio.

Huffman algorithmic program is very popular and widely used lossless compression tool among statistical and bit level text compression techniques. In 1952, DA Huffman presented his most cited and important data compression algorithm [2], where the codeword is assigned to each symbol in such a way that no two symbols have the same codeword and the starting and ending point of a symbol can be easily recognized without requiring any additional information. This is also known as prefix-free coding.

After Huffman algorithm was proposed, in addition to compressing text data, the algorithm was proved efficient in image and video compression as well [14]. After studying the sibling property and level of trees, the authors claim that the codeword length of both Huffman and Shannon–Fano has similar interpretation [15]. The limitation of the research is that an error of a few hundred bits in the length of a typical record and the entropies are of the order of 2^{80} bits. In another research, the connection between the self-information of a symbol and its codeword length in a Huffman code is investigated [16]. The limitation of the study is that it is hard to attribute comparable significance to the self-information of an individual symbol.

In [17], Fenwick argued that the Huffman codes cannot ameliorate the code efficiency all the time and when moving from the lower extension to the higher, the performance is always nose diving. Instead of a most familiar fixed-to-variable code, a variable-to-variable code was also used to implement Huffman algorithm [18]. By transforming a basic Huffman tree to a recursive one, and then using it to decode more than one symbol at a time was a very interesting technique proposed by Lin et al. [19]. The proposed approach required large memory and only suits for test data compression problems. His primary aim was to improve the efficiency of a Huffman tree.

Google Inc. introduces a new compression technique known as Zopfli [20], which is currently one of the best available compression techniques. Internally Zopfli also uses Huffman coding as a basis of its compression technique. Though Zopfli achieves the highest compression ratio, it is only designed for the browser; decompression is not required.

Different types of compression algorithms and coding systems used in compression algorithms have manifested in the above discussion. It is observed that the length of the Huffman code affects the compression ratio and decoding speed. Therefore, we produce code in a new fashion. In this research, we introduce a quaternary tree to produce more efficient and optimal code to ensure the maximum decoding speed. The quaternary tree is a four ary tree or a tree with at most four children. The detailed architecture of the quaternary tree is discussed in the next section. The codeword generated by the quaternary tree technique is more optimal because it is produced from a less height tree. We produce a dictionary based on the quaternary codes. This new dictionary ensures the enhancement of compression and decompression speed.

3 Quaternary tree architecture

The tree data structure is very important to store data in the memory and retrieve data from the memory. A tree can be structured in many ways. Currently, the binary tree structure is mostly used in Computer Science. The required space to store data in memory and decoding time for retrieving those data from the memory depend on the structure of the tree. The weighted path length, the height of tree, eccentricity, diameter, center, radius, number of internal nodes are some performance measuring mathematical parameters. In this research, we exploit a quaternary tree (4 ary tree or a tree with at most 4 children) to construct the Huffman code. We choose quaternary tree instead of a binary tree (2 ary tree or a tree with at most 2 children) to produce the codeword.

A tree T is called quaternary tree if a vertex of T has at most four children named LEFT, LEFT-MID, RIGHT-MID, RIGHT. When every internal vertex of a tree T has all four children, the tree T is called a full quaternary tree. An ordered rooted tree T always has ordered children at its every internal vertex. Ordered rooted trees are designed in such a way that the children of each internal vertex are revealed in order from left to right. The length of the paths in the tree affects the running time of an algorithm. Consider T is a tree with n external nodes, and assume all of the external nodes are assigned a nonnegative weight. The external weighted path length P of the tree T is defined as the sum of the weighted path lengths of individual nodes, then

$$P = f_1l_1 + f_2l_2 + \cdots + f_nl_n \tag{1}$$

where f_i and l_i stand for the frequency and path length of the external node N_i.

3.1 Minimization of time

For the construction of Huffman codes of all the nodes, it is required to traverse the whole tree. Since the traversing time X of a tree depends on its weighted path length P. Thus, we have

$$X \propto P \Longrightarrow X \propto \sum_{i=1}^{n} f_il_i \tag{2}$$

The traversing time of a quaternary tree is always less than that of a binary tree, since $P_\mathrm{q} < P_\mathrm{b}$, where P_q and P_b are the weighted path length of quaternary and binary tree representation. We summarize the above result in the following remarks.

3.2 Remarks

For any distributions, in case of a Huffman tree

(i) The minimum height of a quaternary tree is less than that of a binary tree.
(ii) The maximum height of a quaternary tree is less than that of a binary tree. For big n the first one is almost one third of the later.
(iii) The weighted path length in case of a quaternary tree is less than that of a binary tree.

Proof Consider a complete tree in case of both binary and quaternary representation. And also consider a canonical tree for binary and quaternary representation as shown in Figs. 1 and 2 respectively. We know that the height of a tree is minimum when it is complete and maximum when it is canonical.

(i) A complete k-ary tree with n nodes has height equal to log_kn which is minimum. Then $l_\mathrm{q} = log_4n = \frac{1}{2}log_2n = \frac{1}{2}l_\mathrm{b}$, where l_q is the height in case of a quaternary tree and l_b for a binary tree.
(ii) In case of a canonical binary tree with n nodes, we consider 2 blocks of levels as shown in Fig. 1. There is exactly 1 node in each level except the terminal level which consists of 2 nodes. Let m be the number of levels with only 1 node.
Then, $n = m + 2$ and $l_\mathrm{b} = m + 1$
Then, $l_\mathrm{b} = n - 2 + 1 = n - 1$

$$\Longrightarrow l_\mathrm{b} = n - 1 \tag{3}$$

In the case of a canonical quaternary tree with n nodes, we consider 3 blocks of levels as shown in Fig. 2. The terminal level has exactly 4 nodes, the starting level has $r(1 \le r \le 3)$ nodes and all the intermediate levels must have exactly 3 nodes. If m is the number of intermediate levels, then
$n = r + 3m + 4,\ 1 \le r \le 3$ and
$l_\mathrm{q} = 1 + m + 1$
Then $m = \dfrac{(n-r-4)}{3},\ 1 \le r \le 3$
$\therefore l_\mathrm{q} = 1 + \dfrac{n-r-4}{3} + 1,\ 1 \le r \le 3$

$$\Longrightarrow l_\mathrm{q} = 1 + \frac{n-r-2}{3},\ 1 \le r \le 3 \tag{4}$$

Then, from (3) and (4), we have $l_\mathrm{q} = \dfrac{n-r+2}{3} < n - 1 = l_\mathrm{b},\ 1 \le r \le 3 \Longrightarrow l_\mathrm{q} \approx \dfrac{1}{3}l_\mathrm{b}$ (for big n)
iii) For frequencies $f_i,\ i = 1, 2, 3, \ldots, n,$ for the best case, by part i), we have
$\sum f_il_\mathrm{q} = \dfrac{1}{2}\sum f_il_\mathrm{b}$
$\Longrightarrow T_\mathrm{q} = \dfrac{1}{2}T_\mathrm{b}$ [using (2)]

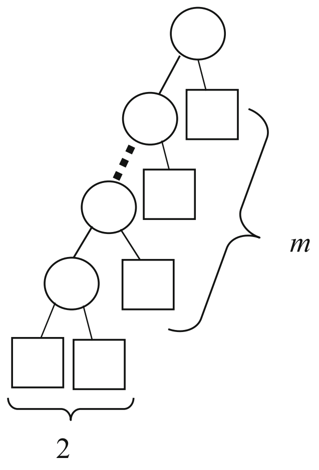

Fig. 1 Canonical binary tree

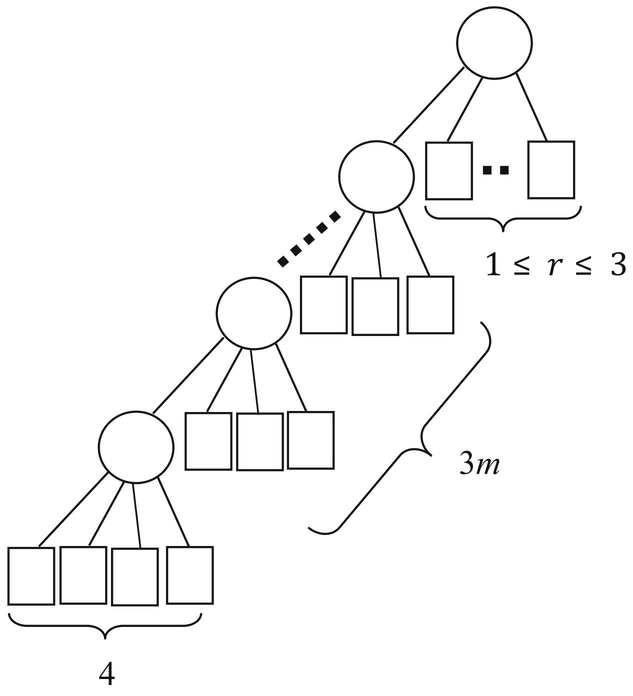

Fig. 2 Canonical quaternary tree

For the worst case, by part ii), we have $\sum f_i l_\text{q} \approx \frac{1}{3} \sum f_i l_\text{b}(\text{ for big } n)$ $\Longrightarrow T_\text{q} \approx \frac{1}{3} T_\text{b}$ [using (2)] In summary, it is revealed that the traversing time of a quaternary tree is at least one-third of the traversing time of a binary tree for big n whereas it is half of a binary technique in the best case.

3.3 Space requirement

During space minimization, maximum performance is achieved when a tree is complete. Because a complete tree requires less decoding time than an incomplete tree keeping the same space requirement. The required space (compressed) depends on the average codeword length. The average codeword length S can be defined as

$$S = \frac{\sum_{i=1}^{n} l_i f_i}{\sum_{i=1}^{n} f_i} \tag{5}$$

where n is the number of distinct symbols As mentioned earlier, the length of the codeword depends on the path length. Since $l_i \propto \alpha_i$, therefore (5) can be rewritten as

$$S = \frac{K \sum_{i=1}^{n} \alpha_i f_i}{\sum_{i=1}^{n} f_i}, \quad l_i = \alpha_i.K$$

where K is the arity and α_i is the height constant. Again, for a particular symbol, when the frequency f_i is increased, the corresponding height α_i is decreased. Thus, we can also write,

$$\alpha_i \propto \frac{1}{f_i}$$

When $K = 1$, then it is a binary tree and it requires a minimum 1 bit to represent a symbol. When $K = 2$ then it is a quaternary tree and it requires at least 2 bits to represent a symbol. Apparently, the space requirement is increased for a quaternary tree. However, when the value of K increases, the value of α_i decreases accordingly. It is obvious that a higher number of K produce a tree with a lower height. Hence, the space requirement for higher values of K does not increase linearly with the increment of K.

4 Implementation

We have implemented this research in three steps. First, we have produced a dictionary based on the quaternary code. Secondly, we have encoded data based on the newly produced dictionary. Thirdly, we have decoded data with four normalized dictionaries. For these three steps, three algorithms have been developed which are discussed in Sect. 4.1.

Letter frequency of a language is very important for creating a dictionary. Huffman principle produces codeword for a symbol based on its frequency in the text. In this context, we have developed frequency counter software to calculate the frequency. We have used Brown [21], Canterbury [22], English part of an English–Bengali parallel corpus SUPara [23], and Enwik [24] corpora to calculate the frequency. There are 87, 96, 106, and 205 distinct symbols in Canterbury, Brown, SUPara, and Enwik corpora, respectively.

We require all the symbols available in the keyboard (small letters, capital letters, big key, punctuations, numbers, etc.), so that we can create a complete dictionary. We have tested the accuracy of our frequency counter with existing tools. It is obvious that the repetitions of English symbols in different corpora are almost the same. Moreover, it is easy to check that the nth character (from bottom to top) in the frequency distribution table occurs with the probability $\frac{k}{n^2}$ (approximately), for some constant $k < n$.

4.1 Dictionary generation algorithm

We have produced a character-based dictionary rather than traditional word-based or syllable-based dictionary. Because the numbers of words or syllables are undefined; sometimes it may reach more than 100 millions of words for a monolingual corpus, whereas the number of characters or symbols is fixed. We have 205 symbols for the first version of the dictionary. The dictionary is produced using the principle of Huffman on quaternary tree architecture. An algorithm is developed to generate quaternary Huffman codeword. The algorithm used for generating codeword is shown in Algorithm 1 [25]. The run time complexity of the algorithm is $O(mlog_4n)$, whereas for Huffman-based algorithm is $O(nlog_2n)$. Here, m is always less than n, because the total number of nodes in a quaternary tree is less than that of a binary tree. For Enwik corpus, the average codeword length of the dictionary using quaternary code is 4.726, whereas the average codeword length of the dictionary using binary codes is 4.639. The weighted path length of the quaternary tree is 3501553, whereas the weighted path length of the binary tree is 6873872. The searching becomes faster when the petite quaternary tree is used.

Algorithm 1: Dictionary Generation Algorithm

```
    Q-Huffman (C)
1.  $Q \leftarrow C$
2.  $n \leftarrow |Q|$
3.  $i \leftarrow n$
4.  WHILE $i > 1$
5.      allocate a new node $z$
6.      left[$z$] $\leftarrow v \leftarrow$EXTRACT-MIN(Q)
7.      left-mid[$z$] $\leftarrow w \leftarrow$EXTRACT-MIN(Q)
8.      IF $i = 2$
9.          $f[z] \leftarrow f[v] + f[w]$
10.     ELSE IF $i = 3$
11.         right-mid[$z$] $\leftarrow x \leftarrow$EXTRACT-MIN(Q)
12.         $f[z] \leftarrow f[w] + f[x]$
13.     ELSE
14.         right-mid[$z$] $\leftarrow x \leftarrow$EXTRACT-MIN(Q)
15.         right[$z$] $\leftarrow y \leftarrow$EXTRACT-MIN(Q)
16.         $f[z] \leftarrow f[v] + f[w] + f[x] + f[y]$
17.     END IF
18.     INSERT(Q,z)
19.     $n \leftarrow |Q|$
20. END WHILE
21. RETURN EXTRACT-MIN(Q)
```

Details of the algorithm can be found in another research [25].

4.2 Encoding algorithm

As a prerequisite of the encoding process, a static dictionary is created using quaternary Huffman algorithm. Every character of the text is replaced by its codeword from the dictionary. We have developed an algorithm to convert a string to a binary codeword which is shown in Algorithm 2.

Algorithm 2: Encoding Algorithm

```
    ENCODE (Text File, Dictionary)
1.  Set $T$ = Text file
2.  Set $N$ = Length of $T$
3.  Set $BITSTRING = NULL$
4.      FOR $i = 0$ to $N - 1$
5.      match $T[i]$ with dictionary character
6.      $BITSTRING = BITSTRING + CODEWORD$
        of character
7.  END FOR
8.  Convert $BITSTRING$ into unsigned short integers and
        store it into a binary file
9.  EXIT
```

In the algorithm 2, in line 1, T is the input text file and in line 2, N is the length of text file T. In line 3, $BITSTRING$ is the string variable with a NULL value. In line 4, there is a loop. Here, each character of Text T matches with dictionary and stores its $CODEWORD$ into $BITSTRING$ variable with its previous value. The process continues until the last character with its $CODEWORD$ is added into $BITSTRING$. In line 7, $BITSTRING$ is converted into unsigned short integers and stored into a binary file. The encoding algorithm is very simple and fast. An efficient linear search is used and its complexity is less than $O(n)$. Since the nth character in the frequency with the probability is $\frac{k}{n^2}$ (approximately), for some constant $k < n$, for linear search we can write

$$\begin{aligned} C(n) &= 1 \cdot \frac{k}{n^2} + 2.\frac{k}{n^2} + \cdots + n \cdot \frac{k}{n^2} \\ &= (1 + 2 + \cdots + n) \cdot \frac{k}{n^2} \\ &= \frac{n(n+1)}{2} \cdot \frac{k}{n^2} \\ &= \frac{k}{2} \cdot \left(1 + \frac{1}{n}\right) \\ &\approx \frac{k}{2} < O(n) \end{aligned}$$

This optimal result is achieved just because of the formation of the dictionary based on the quaternary Huffman principle. It is found that top 10 characters out of 205 characters occupy almost 70% of the text file and we have traversed top positions of the dictionary most of the time.

4.3 Decoding algorithm

The dictionary is split into four dictionaries for faster search. The codeword started with 00, 01, 10, 11 takes place in $Dictionary\_1$, $Dictionary\_2$, $Dictionary\_3$, and $Dictionary\_4$, respectively.

In all the cases, top entries of dictionaries contain most frequent symbols which ensure faster search during the decoding process. Decoding is started by reading two bits at a time from the encoded file. All the quaternary codes are available in four normalized dictionaries. When a bit pattern matches with any codeword in the dictionary,the respective

Algorithm 3: Decoding Algorithm

```
     DECODE(Encoded Binary File, Dictionary)
1.   Read the binary file and take the unsigned short integer numbers as input
2.   Set $BITSTRING$=convert unsigned short integer numbers into bits
3.   Set $N = length$ of $BITSTRING$
4.   FOR $i = 0$ to $N - 1$
5.       $WORD \leftarrow$ extract first 2 bits from $BITSTRING$
6.       IF $WORD =$'00' $Dictionary\_No \leftarrow 1$
7.       ELSE IF $WORD =$'01' $Dictionary\_No \leftarrow 2$
8.       ELSE IF $WORD =$'10' $Dictionary\_No \leftarrow 3$
9.       ELSE IF $WORD =$'11' $Dictionary\_No \leftarrow 4$
10.      END IF
11.      Set $i = i + 2$
12.      WHILE(TRUE)
13.          SET $k = 0$
14.          IF $WORD.size >= (Dictionary\_No)[k].CODEWORD$ length
15.              WHILE $k <$ Length of $(Dictionary\_No)$
16.                  IF $(Dictionary\_No)[k].size$ of $CODEWORD > WORD.size$
17.                      Break the loop
18.                  IF $(Dictionary\_No)[k].CODEWORD = WORD$
19.                      Write the character into the decoded file for the $CODEWORD$
                         $(Dictionary\_No)$ and break the loop in line 15
20.                  Set $k = k + 1$
21.              END WHILE
22.              IF $(Dictionary\_No)[k].CODEWORD$ match with $WORD$, break the loop in line 12
23.                  IF $i + 2 < N$
24.                      Set $WORD = WORD + (BITSTRING[i + 1] + BITSTRING[i + 2])$
25.                      Set $i = i + 2$
26.                  ELSE
27.                      Break the loop in line 12
28.                  END IF
29.              ELSE
30.                  IF $i + 2 < N$
31.                      Set $WORD = WORD + (BITSTRING[i + 1] + BITSTRING[i + 2])$
32.                      Set $i = i + 2$
33.                  ELSE
34.                      Break the loop in line 12
35.              END IF
36.          END IF
37.      END WHILE
38.      SET $i = i - 1$
39.  END FOR
40.  EXIT
```

symbol is replaced with the codeword. The process is iterated until reaching the last two bits of the file. The decoding algorithm is shown in Algorithm 3.

In the algorithm 3, in line 1, the binary file is read and the unsigned short integer numbers are taken as input. In line 2, unsigned short integer numbers are converted into bits and are set into a variable named $BITSTRING$. In line 3, a new variable N is declared as the length of $BITSTRING$ variable. In line 4, there is a loop to search for any one of the dictionaries. After extracting two bits, the searching process is started in a particular dictionary. In line 11, loop iterator is increased by 2. In line 12, a loop is run until the expected character is found. In line 13, an index variable named k is initialized with 0. In line 14, we check the current $WORD$ length whether it exists in the dictionary. In line 15, there is a loop to search the whole dictionary. In line 16, if the $WORD$ size is greater than the existing $CODEWORD$ size then we break the loop in line 15. In line 18, if the $WORD$ matches with the dictionary $CODEWORD$, we write the character of respective $CODEWORD$ into the decoded text and break the loop in line 15. In line 20, we increase the index variable by 1. In line 21, if the character is found then we break the loop in line 12. In line 22, if the iterator $i + 2$ is less than N then we update the $WORD$ by adding next 2 bits from $BITSTRING$. In line 24, we increase the iterator i by 2. In line 26, we end the loop in line 12. In line 27, we continue the process line 22 to 26. In line 33, we decrease the iterator value i by 1.

For matching a codeword with its symbol, we have to visit any one of the four dictionaries. In most of the cases we found the symbols on the top of the dictionary because most frequent character stays at the top position of the dictionary. The symbols are set in order of decreasing probability in the proposed dictionary. The probability of these symbols is geometrically distributed. For this case, the run time complexity is less than $O(n)$. If l equals half of the maximum code length, then for linear search we can write

$$\begin{aligned}
C(n) &\leq \frac{1}{4} \cdot l\left(1 \cdot \frac{1}{n^2} + 2 \cdot \frac{1}{n^2} + \cdots + n \cdot \frac{1}{n^2}\right)\\
&\leq \frac{l}{4} \cdot (1 + 2 + \cdots + n) \cdot \frac{1}{n^2}\\
&\leq \frac{l}{4} \cdot \frac{n(n+1)}{2} \cdot \frac{1}{n^2}\\
&\leq \frac{l}{4} \cdot \frac{n+1}{2n}\\
&\leq \frac{l \cdot (n+1)}{8n}\\
&\approx \frac{l}{8} < O(n)(\because l < n)
\end{aligned}$$

Encoding and decoding processes of the proposed technique have been thoroughly discussed in this section. The search time for finding a symbol in decoding algorithm is $O(\frac{l.(n+1)}{8n})$, whereas for Huffman-based techniques complexity of decoding algorithm is $O(nlog_2 n)$.

5 Performance analysis

5.1 Methods

The test computer we used is an Intel®Core™i5—6500 CPU running at 3.20 GHz with 2 cores and 4 additional hyper threading contexts. We ran Ubuntu 14.04 LTS Operating system. All codecs were compiled using the same compiler, GCC 4.8.4. The amount of primary memory is 4 GB DDR4 type.

The versions of the following algorithms are tested in our experiment:

- Zopfli version 2015-09-01 [26], Google claims that it has the highest compression ratio,
- LZHAM, an advanced compression algorithm [27],
- bzip2 1.0.6 6-Sept-2010 [28]; an open source compression program,
- LZMA implementation in 7zip 9.20.1 [29], is an algorithm used to perform lossless data compression, and
- TCQC—the proposed technique, a text compression technique using Quaternary Huffman based dictionary.

The compression corpora we used in the testing are

- the Canterbury, an ad hoc crawled web content corpus, 1285 files, 70611753 bytes total, and this corpus is a modified version of the Calgary corpus which is designed to test the compression algorithms,
- the Enwik8, a single file corpus, the Enwik corpus is a 95.3 MB file with 205 distinct characters,
- the SUPara, an English–Bengali parallel corpus produced by Shahjalal University of Science and Technology (SUST), Bangladesh, and
- the Brown, a modern, computer readable, general corpus consists of one million words of American English texts.

We have measured the compression ratio, compression speed in megabytes/second (MB/S) and decompression speed in MB/S for selected algorithms and compression levels. The compression and decompression speed of each algorithm are measured with the same environment and the same compiler. Zopfli only compresses and does not decompress, for measuring decompression speed of Zopfli we used gzip-9 [30] as decompression which is compatible with Zopfli. Unlike other algorithms compared here, TCQC includes a static dictionary. It contains 205 characters of English Wikipedia. We

Table 1 Performance analysis for Enwik corpus

Algorithm	Compression ratio	Compression speed (MB/s)	Decompression speed (MB/s)
Zopfli1.0.1	2.855	0.4470	154.1818
Lzham:1	3.335	1.7886	98.7711
Lzham:4	3.643	0.2981	106.7106
bzip2:1	3.007	9.1665	17.1004
bzip2:9	3.447	9.2411	16.8228
Lzma:1	3.106	7.3034	33.6457
Lzma:9	3.639	2.56366	39.8640
TCQC	2.525	1.70826	159.1843

Table 2 Performance analysis for Canterbury corpus

Algorithm	Compression ratio	Compression speed (MB/s)	Decompression speed (MB/s)
Zopfli1.0.1	3.580	0.1491	189.9373
Lzham:1	3.836	2.9064	64.4043
Lzham:4	3.952	0.3726	65.3482
bzip2:1	3.757	8.7939	22.4304
bzip2:9	3.869	8.9431	22.3194
Lzma:1	3.847	7.6015	38.8647
Lzma:9	4.240	2.9064	39.80855
TCQC	3.166	1.6671	80.2001

Table 3 Performance analysis for SUPara corpus

Algorithm	Compression ratio	Compression speed (MB/s)	Decompression speed (MB/s)
Zopfli 1.0.1	3.393	0.2369	89.9035
Lzham:1	3.961	0.9476	70.2426
Lzham:4	4.326	0.1579	75.8871
bzip2:1	3.571	4.8564	12.1608
bzip2:9	4.093	4.8959	11.9634
Lzma:1	3.688	3.8693	16.1280
Lzma:9	4.389	1.3582	22.6404
TCQC	3.001	0.6732	62.8357

Table 4 Performance analysis for Brown corpus

Algorithm	Compression ratio	Compression speed (MB/s)	Decompression speed (MB/s)
Zopfli1.0.1	2.707	0.2235	255.4521
Lzham:1	3.159	3.5026	93.6639
Lzham:4	3.451	0.1491	95.8847
bzip2:1	2.849	8.1977	29.0374
bzip2:9	3.265	8.2722	29.0374
Lzma:1	2.942	5.8874	55.7986
Lzma:9	3.501	3.2791	56.7424
TCQC	2.394	0.6353	178.5414

can extend the static dictionary by a mechanism of transforms that slightly changes the characters in the dictionary. In averaging over the results of individual files and over the corpus, we have considered five consecutive runs.

5.2 Results

Tables 1, 2, 3 and 4 show the results for Enwik, Canterbury, SUPara, and Brown corpus, respectively. We can see from Table 1 that the decompression speed of TCQC is faster than other algorithms for EnWik8 corpus. The experimental result shows that for Enwik corpus the decompression speed of TCQC is faster than all techniques. It is approximately 1.5, 8, and 4 times faster than Lzham, bzip2, and Lzma, respectively, which is shown in Table 1. The compression speed of bzip2:9 is the highest of all for the EnWik8 corpus.

For Canterbury corpus, the Zopfli has highest decompression speed and the TCQC is second highest but the compression speed of TCQC is better than Zopfli which are shown in Table 2. In Table 2, for Canterbury corpus, the decompression speed of the proposed technique is approximately 1.25, 4, and 2 times faster than Lzham, bzip2, and Lzma, respectively. The compression speed of lzma:1 is the highest of all for Canterbury corpus.

We also test the proposed technique using a bilingual SUPara corpus; the performance is also good compared with bzip2 and lzma which is shown in Table 3. In Table 3, for SUPara corpus, the decompression speed of the proposed technique is approximately 5, and 6 times faster than bzip2, and Lzma, respectively. The compression speed of bzip2:9 is the highest of all for this corpus.

For Brown corpus, the Zopfli has highest decompression speed and the TCQC is second highest but the compression speed of TCQC is better than Zopfli which are shown in Table 4. In Table 4, for Brown corpus, the decompression speed of the proposed technique is approximately 2, 6, and 3 times faster than Lzham, bzip2, and Lzma, respectively. The compression speed of bzip2:9 is the highest of all for this corpus.

From Tables 2 and 4, we observe that the TCQC is little slower than only for Zopfli but the compression speed of TCQC is better than Zopfli for any corpora. Experimental results show that the decompression speed of the proposed technique is better than widely used existing techniques, whereas the space requirement is slightly increased.

The existing Huffman-based algorithms use single bit (binary) code to store data in the memory, which slow the decoding speed. The proposed algorithm uses dibit (quaternary) code to store data. Retrieving two bits at a time from the memory during decoding process speeds up the process. Moreover, if the tree is balanced, and the number of symbols is approximately 4^h, where h is the height of the tree, the proposed algorithm performs well.

6 Conclusion

In this paper, the performance for construction of Huffman code using a quaternary tree is compared with that of using a binary tree. We find that the traversing time of a quaternary tree is approximately one-third of that of a binary tree in the worst case and exactly half in the best case. A variable length dictionary is created using a quaternary code-based Huffman algorithm. During the decoding process, it searches two bits at a time which ensures faster search compared to ordinary linear search. The run time complexity of the encoding and decoding process is $O(\frac{k \cdot (n+1)}{2n})$ and $O(\frac{l \cdot (n+1)}{8n})$, respectively. This method is simple to program and is time efficient. The experiment shows that the decompression speed of the proposed technique is better than the many existing techniques. As we use a quaternary tree to generate the dictionary, so whenever we use more symbols we will get more benefit from it. As Enwik8 corpus has the highest number of symbols, for this case, the performance of the proposed technique is better than any other techniques. The decompression performance of the proposed technique increases when the number of input symbol increases. For any multilingual text, when the number of symbols is higher and the decompression speed is the main factor, the proposed technique may be a better choice for text compression.

In the research, we observe that Huffman principle does not produce any balanced tree. The codeword generation from a balanced tree could improve the compression and decompression speed together. Research on balanced quaternary Huffman tree for production of dictionary codeword could also be an important topic for future research.

Acknowledgements The authors are grateful to Information and Communication Technology Division, Government of the People's Republic of Bangladesh for the grant to do this research work.

Author contributions The authors discussed the problem and the solutions proposed altogether. All authors participated in drafting and revising the final manuscript. All authors read and approved the final manuscript.

Funding All the funding provided by the Ministry of Posts, Telecommunications and Information Technology, People's Republic of Bangladesh [Order No: 56.00.0000.028.33.025.14-153, date: 08.06.2017]. The above funding gives financial support for the designing of the study and conducting experiments.

Availability of data The datasets and the source code of compression algorithms supporting of this article are available online in the following link.The Brown Corpus [Online]. Available: http://www.nltk.org/nltk_data/. Accessed 30 May 2018. The Canterbury Corpus [Online]. Available: http://corpus.canterbury.ac.nz/resources/cantrbry.zip. Accessed 30 May 2018. The Enwik8 Corpus [Online]. Available: http://mattmahoney.net/dc/text.html, http://mattmahoney.net/dc/enwik8.zip. Accessed 30 May 2018. The SUPara Corpus [Online]. Available: http://dx.doi.org/10.21227/gz0b-5p24. Zopfli Source Code

[Online]. Available: https://github.com/google/zopfli/commit/\penalty-\@M89cf773beef75d7f4d6d378debdf299378c3314e. Accessed 30 May 2018. LZHAM Source [Online]. Available: https://github.com/\penalty-\@Mrichgel999/lzham_codec. Accessed 30 May 2018. bzip2 Source: bzip2 1.0.6 6-Sept-2010 [Online]. Available: https://github.com/\penalty-\@Menthought/bzip2-1.0.6. LZMA Source: LZMA implementation in 7zip 9.20.1 [Online]. Available: LZMA SDK:https://www.7-zip.org/sdk.html.

Compliance with ethical standards

Conflict of interest The authors declare that they have no competing interests.

References

1. Khuri, S., Hsu, H-C.: Tools for visualizing text compression algorithms. In: Proceedings of the 2000 ACM Symposium on Applied Computing (SAC'00), Como, Italy, March 2000, vol. 1, pp. 119–123 (2000)
2. Huffman, DA.: A method for construction of minimum redundancy codes. In: Proceedings of the IRE, Sep 1952, vol. 40, pp. 1090–1101 (1952)
3. Carus, A., Mesut, A.: Fast text compression using multiple static dictionaries. Inf. Technol. J. **9**(5), 1013–1021 (2010). https://doi.org/10.3923/itj.2010.1013.1021
4. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977). https://doi.org/10.1109/TIT.1977.1055714
5. Ziv, J., Lempel, A.: Compression of individual sequence via variable-rate coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978). https://doi.org/10.1109/TIT.1978.1055934
6. Storer, J.A., Szymanski, T.G.: Data compression via textual substitution. J. ACM **29**(4), 928–951 (1982). https://doi.org/10.1145/322344.322346
7. Welch, T.A.: A technique for high-performance data compression. Computer **17**(6), 8–19 (1984). https://doi.org/10.1109/MC.1984.1659158
8. Moffat, A., Isal, R.Y.K.: Word-based text compression using the Burrows–Wheeler transform. Inf. Process. Manag. **41**(5), 1175–1192 (2005). https://doi.org/10.1016/j.ipm.2004.08.009
9. L'ansk'y, J., Žemlička, M.: Text compression: syllables. In: Proceedings of the Dateso–Workshop on Databases, Texts, Specifications and Objects, Desna, Czech Republic, April 13–15, 2005, pp. 32–45 (2005)
10. Adiego, J., de la Feunte, P.: On the use of words as source alphabet symbols in PPM. In: Proceedings of Data Compression Conference, Snowbird, UT, USA, March 28–30, 2006, pp. 435 (2006)
11. Dvorsky, J., Pokorny, J., Snasel, V.: Word-based compression methods for large text documents. In: Proceedings of Data Compression Conference, Snowbird, UT, USA, March 29–31, 1999, pp. 523 (1999)
12. L'ansk'y, J., Žemlička, M.: Compression of a dictionary. In: Proceedings of DATESO Workshop on Databases, Texts, Specifications and Objects, Desna, Czech Republic, April 26–28, 2006, pp. 11–20 (2006)
13. Al-Bahadili, H., Rababa, A.: An adaptive bit-level text compression scheme based on the HCDC algorithm. In: Proceedings of Mosharaka International Conference on Communications, Networking and Information Technology, Amman, Jordan, Dec 6–8, 2007, pp. 51–56 (2007)
14. Chung, K.L.: Efficient Huffman decoding. Inf. Process. Lett. **61**(2), 97–99 (1997). https://doi.org/10.1016/S0020-0190(96)00204-9
15. Schack, R.: The length of a typical Huffman codeword. IEEE Trans. Inf. Theory **40**(4), 1246–1247 (1994). https://doi.org/10.1109/18.335944
16. Katona, G.O.H., Nemetz, T.O.H.: Huffman codes and self-information. IEEE Trans. Inf. Theory **22**(3), 337–340 (1978). https://doi.org/10.1109/TIT.1976.1055554
17. Fenwick, P.M.: Huffman code efficiencies for extensions of sources. IEEE Trans. Commun. **43**(2/3/4), 163–165 (1995)
18. Kavousianos, X., Kalligeros, E., Nikolos, D.: Test-data compression based on variable-to-variable huffman encoding with codeword reusability. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **27**(7), 1333–1338 (2008)
19. Lin, Y.-K., Huang, S.-C., Yang, C.-H.: A fast algorithm for Huffman decoding based on a recursion Huffman tree. J. Syst. Softw. **85**, 974–980 (2012)
20. Alakuijala, J., Vandevenne, L.: Data compression using Zopfli. Google Inc. [Online]. https://zopfli.googlecode.com/files/Data_compression_using_Zopfli.pdf. Accessed 30 May 2018 (2013)
21. The Brown Corpus [Online]. http://www.nltk.org/nltk_data/. Accessed 30 May (2018)
22. The Canterbury Corpus [Online]. http://corpus.canterbury.ac.nz/resources/cantrbry.zip. Accessed 30 May (2018)
23. Mumin, M.A.A., Shoeb, A.A.M., Selim, M.R., Iqbal, M.Z.: SUPara: a balanced english-bengali parallel corpus. SUST J. Sci. Technol. **16**(2), 46–51 (2012)
24. The Enwik8 Corpus [Online]. http://mattmahoney.net/dc/text.html http://mattmahoney.net/dc/enwik8.zip. Accessed 30 May 2018
25. Habib, A., Rahman, M.S.: Balancing decoding speed and memory usage for Huffman codes using quaternary tree. Appl. Inf. **4**, 5 (2017). https://doi.org/10.1186/s40535-016-0032-z
26. Zopfli Source Code [Online]. https://github.com/google/zopfli/commit/89cf773beef75d7f4d6d378debdf299378c3314e. Accessed 30 May 2018
27. LZHAM Source [Online]. https://github.com/richgel999/lzham_codec. Accessed 30 May (2018)
28. bzip2 Source: bzip2 1.0.6 6-Sept-2010 [Online]. https://github.com/enthought/bzip2-1.0.6. Accessed 30 May 2018
29. LZMA Source: LZMA implementation in 7zip 9.20.1 [Online]. LZMA SDK: https://www.7-zip.org/sdk.htm. Accessed 30 May 2018
30. Deutsch P (1996) RFC 1952—GZIP file format specification, version 4.3, May 1996. [Online]. http://www.ietf.org/rfc/rfc1952.txt. Accessed 30 May (2018)