


REVIEW

Open Access



Balancing decoding speed and memory usage for Huffman codes using quaternary tree

Ahsan Habib*  and Mohammad Shahidur Rahman

*Correspondence:
ahabib-cse@sust.edu
Shahjalal University
of Science and Technology,
Sylhet, Bangladesh

Abstract

In this paper, we focus on the use of quaternary tree instead of binary tree to speed up the decoding time for Huffman codes. It is usually difficult to achieve a balance between speed and memory usage using variable-length binary Huffman code. Quaternary tree is used here to produce optimal codeword that speeds up the way of searching. We analyzed the performance of our algorithms with the Huffman-based techniques in terms of decoding speed and compression ratio. The proposed decoding algorithm outperforms the Huffman-based techniques in terms of speed while the compression performance remains almost same.

Keywords: Binary tree, Encoding and decoding, Huffman tree, Quaternary tree, Data compression

Background

Huffman (1952) presented a coding system for data compression at I.R.E conference in 1952 and informed that no two messages will consist of same coding arrangement and the codes will be produced in such a way that no additional arrangement is required to specify where a code begins and ends once the starting point is known. Since that time Huffman coding is not only popular in data compression but also image and video compression (Chung 1997). Schack (1994) described in his paper that codeword lengths of both Huffman and Shannon–Fano have similar interpretation. Katona and Nemetz (1978) investigated the connection between self-information of a source symbols and its codeword length.

In another research, Hashemian (1995) introduced a new compression technique with the clustering algorithm. In this new type of algorithm, he claimed that it required minimum storage whereas the speed for searching of symbol will be high. He also conducted experiment on video data and found his method very efficient. Chung (1997) introduced an array-based data structure for Huffman tree where the memory requirement is $3n - 2$. He also proposed a fast decoding algorithm for this structure and claimed that the memory size can be reduced from $3n - 2$ to $2n - 3$, where n is the number of symbols. To attain more decoding speed with compact memory size, Chen et al. (1999) presented a fast decoding algorithm with $O(\log n)$ time and $\lceil \frac{3n}{2} \rceil + \lceil (\frac{n}{2}) \log n \rceil + 1$ memory space.

Banetley et al. (1986) introduced a new compression technique that is quite close to Huffman technique with some implementation advantages; it requires one-pass over the data to be compressed. Sharma (2010) and Kodituwakku and Amarasinghe (2011) have presented that Huffman-based technique produces optimal and compact code. However, the decoding speed of this technique is relatively slow. Bahadili and Hussain (2010) presented a new bit level adaptive data compression technique based on ACW algorithm, which is shown to perform better than many widely used compression algorithms in terms of compression ratio. Hermassi et al. (2010) showed how a symbol can be coded by more than one codeword having the same length. Chowdhury et al. (2002) presented a new decoding technique of self-styled static Huffman code, where they showed a very efficient representation of Huffman header. In paper, Suri and Goel (2011) focused on the use of ternary tree, where a new one-pass algorithm for decoding adapting Huffman codes is implemented.

Fenwick (1995) in his research showed that the Huffman codes do not improve the code efficiency at all time. It shows that the performance is always declining when moving to the lower extension to higher extension. Szpankowski (2011) and Baer (2006) explained the minimum expected length of fixed-to-variable lossless compression without prefix constraint. Huffman principle, which is well known for fixed-to-variable code, is used in Kavousianos (2008) as a variable-to-variable code. A new technique for online compression in networks has been presented by Vitter (1987) in his paper. Habib et al. (2013) introduced Huffman code in the field of database compression. Gallager (1978) explained four properties of Huffman codes—sibling property, upper bound property, codeword length property and symbol frequency property. He also proposed an adaptive approach of Huffman coding. Lampel and Ziv (1977) and Welch (1984) described a coding technique for any kind of source symbol. Lin et al. (2012) worked on the efficiency of Huffman decoding, where authors first transform the basic Huffman tree to recursive Huffman tree, and then the recursive Huffman algorithm decodes more than one symbol at a time. In this way, it achieves more decoding speed. Google Inc. recently released a compression tool named Zopfli (Alakuijala and Vandevenne 2013) and claimed that Zopfli yields the best compression ratio.

In summary, it is revealed in the literature that using binary Huffman code it is difficult to achieve a balance between speed and memory usage. In this paper, we focus on the use of quaternary tree instead of binary tree that speeds up decoding time. Here, we employ two algorithms for encoding and decoding quaternary Huffman codes for the implementation of our proposed technique. When compared with the Huffman-based techniques, the proposed decoding algorithm exhibits excellent performance in terms of speed while the compression performance remains almost same. In this way, the proposed technique offers a way to balance between the decoding time and memory usage. We have organized the paper as follows. In “[Quaternary tree architecture](#)” section, traditional binary Huffman decoding technique in data management systems is presented. The overview of our proposed architecture with encoding and decoding techniques is also presented in this section. The implementation technique has been described in “[Implementation](#)” section. The experimental results have been thoroughly discussed in “[Result and discussion](#)” section and finally “[Conclusion](#)” section concludes the paper.

Quaternary tree architecture

The main contribution of this research is to implement a new lossless Huffman-based compression technique. The implementation of the algorithms has been explained with some mathematical foundations. Finally, implemented algorithms have been tested using real data.

Tree construction

Huffman codes to binary data

Huffman’s scheme uses a table of frequency to produce codeword for each symbol (Wikipedia short history of Huffman coding 2011). This table consists of every symbol of entire document and its respective frequency is arranged in ascending order. According to the frequency of distinct symbol, each symbol has a variable-length bit string and all the bit strings are distinct. Table 1 shows the variable-length codeword for different symbols of the sentence “This is an example of quaternary Huffman tree.”

Consider a set of source symbols $S = \{s_0, s_1, \dots, s_{n-1}\} = \{\text{Space}, a, \dots, y, .\}$ with frequencies $W = \{w_0, w_1, \dots, w_{n-1}\}$ for $w_0 \geq w_1 \geq \dots \geq w_{n-1}$, where the symbol s_i has frequency w_i and n is the number of symbols. The codeword $c_i, 0 \leq i \leq n - 1$, for symbol s_i can be calculated by traversing the path from root to the symbol s_i , when goes to left it writes ‘0’ and when goes to right it writes ‘1’. If the level of the root is zero, then the codeword length can be determined as the level of s_i . The traversing time of a tree depends on its weighted path length $\sum w_i l_i$, which is expected to be minimum. The Huffman tree for the source symbols $\{s_0, s_1, \dots, s_{18}\}$ with the frequencies $\{8, 6, \dots, 1\}$, respectively, for the above example is shown in Fig. 1. The codeword set $C\{c_0, c_1, \dots, c_{18}\}$ is derived as $\{000, 010, \dots, 11101\}$, respectively, is shown in Table 1.

Table 1 Codeword generation using binary Huffman principle

Character	Frequency	Code
Space	8	000
A	6	010
E	5	101
T	3	1000
N	3	1001
F	3	0110
R	3	0111
H	2	1101
I	2	00110
S	2	00111
M	2	00100
U	2	00101
X	1	11001
P	1	110000
L	1	110001
O	1	11110
Q	1	11111
Y	1	11100
.	1	11101

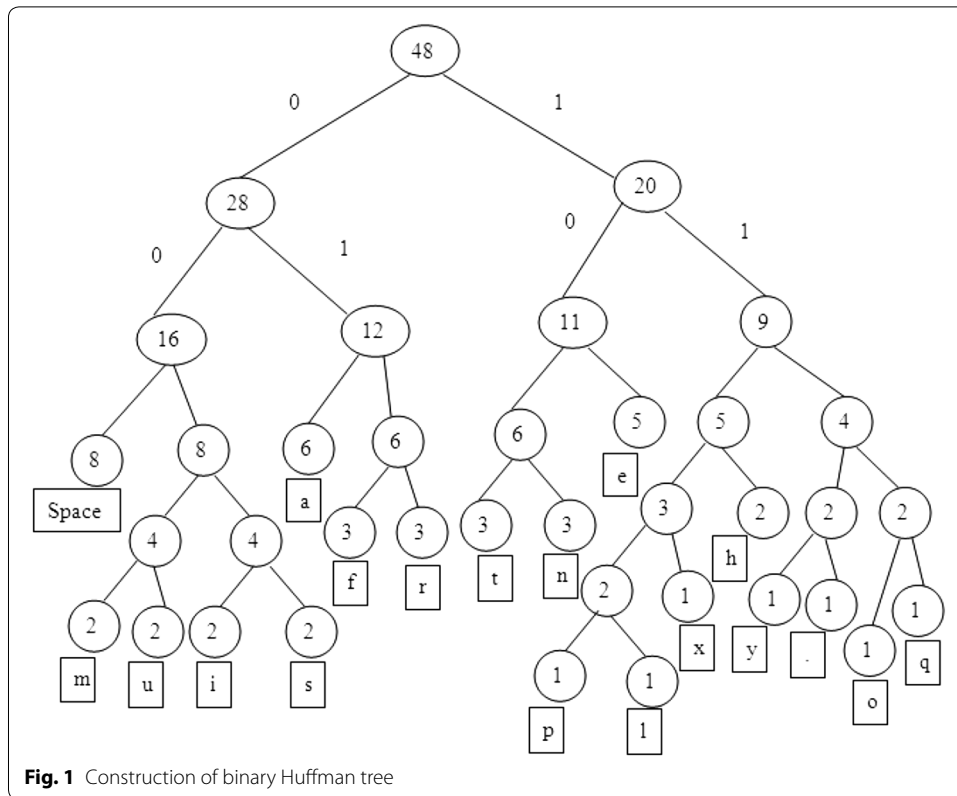


Fig. 1 Construction of binary Huffman tree

Huffman codes to quaternary data

Quaternary tree or 4-ary tree is a tree in which each node has 0–4 children (labeled as LEFT child, LEFT MID child, RIGHT MID child, RIGHT child). Here for constructing codes for quaternary Huffman tree, we use 00 for left child, 01 for left-mid child, 10 for right-mid child, and 11 for right child.

The process of the construction of a quaternary tree is described below:

- List all possible symbols with their probabilities;
- Find the four symbols with the smallest probabilities;
- Replace these by a single set containing all four symbols, and the probability of the parent is the sum of the individual probabilities.
- Replicate the procedure until it has one node.

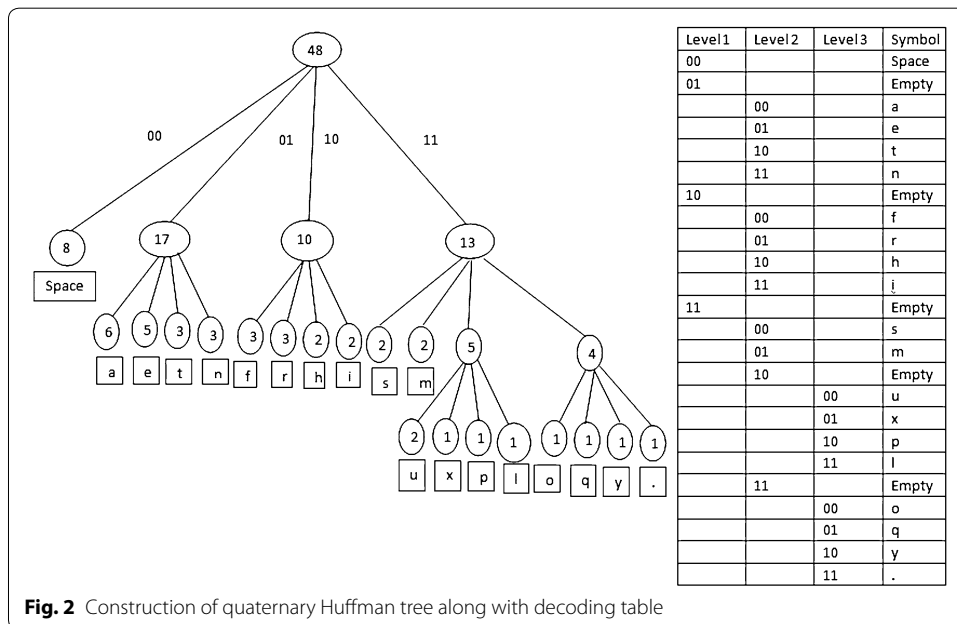
The code word generated using quaternary Huffman technique is shown in Table 2.

Consider a set of source symbols $S = \{s_0, s_1, \dots, s_{n-1}\} = \{\text{Space}, a, \dots, y, .\}$ with frequencies $W = \{w_0, w_1, \dots, w_{n-1}\}$ for $w_0 \geq w_1 \geq \dots \geq w_{n-1}$, where the symbol s_i has frequency w_i and n is the number of symbols. The codeword $c_i, 0 \leq i \leq n - 1$, for symbol s_i can be calculated by traversing the path from root to the symbol s_i , when goes to left it writes ‘00’, when goes to left mid writes ‘01’, when goes to right mid writes ‘10’ and when goes to right writes ‘11’. The codeword length of a symbol can simply be calculated as the level of s_i . We know that the traversing time of a tree depends on its weighted path length $\sum w_i l_i$, and it is expected to be minimum. The quaternary Huffman tree for

Table 2 Codeword generation using quaternary Huffman principle

Character	Frequency	Code
Space	8	00
A	6	0100
E	5	0101
T	3	0110
N	3	0111
F	3	1000
R	3	1001
H	2	1010
I	2	1011
S	2	1100
M	2	1101
U	2	111000
X	1	111001
P	1	111010
L	1	111011
O	1	111100
Q	1	111101
Y	1	111110
.	1	111111

the source symbols $\{s_0, s_1, \dots, s_{18}\}$ with the frequencies $\{8, 6, \dots, 1\}$, respectively, for the above example (“This is an example of quaternary Huffman tree.”) is shown in Fig. 2. The codeword set $C\{c_0, c_1, \dots, c_{18}\}$ is derived as $\{00, 0100, \dots, 111111\}$, respectively, which is shown in Table 2.



Comparison of binary and quaternary tree

Table 3 shows some comparisons with some mathematical parameters for the previous example.

Reduction of time using quaternary tree

Encoding and decoding time of a tree depends on the weighted path length of a tree. If n is the number of distinct character, L_i is code length of the i th character, and f_i is the frequency of the i th character, then we can write the required traversing time T as

$$T \propto \sum_{i=1}^n L_i f_i$$

$$T \propto K \sum_{i=1}^n \alpha_i f_i$$

where $L_i = \alpha_i \cdot K$, $\alpha_i \propto \frac{1}{f_i}$, $K = \text{arity} = 2$, for quaternary tree, and $\alpha_i = \text{height constant}$

Thus, the traversing time also depends on the height of the tree and frequency of different symbols. The height of a quaternary tree is always smaller than the height of a binary tree. For this reason, traversing time will be reduced for a petite tree.

The structure of header tree for decoding is very simple for the proposed technique. According to Fig. 2, it does not require to store the entire codeword in the header tree for a symbol. The most frequent symbol is stored first in the header which confirms faster decoding. Moreover, retrieving two bits at a time during decoding process also speeds up the process. In the decoding phase, matching (two bits at a time) from encoded bit string with the header starts from level 1 in the header tree. If there is any symbol with codeword of length 2, then it will be found in level 1 in the header tree. Likewise, matching a symbol with codeword of length 4 both the level 1 and level 2 have to be searched. The simplicity of the header tree also contributes to speed up the decoding process.

Implementation

As mentioned earlier, in quaternary tree each node has 0–4 children (labeled as LEFT child, LEFT MID child, RIGHT MID child, and RIGHT child).

There are basically two components in quaternary Huffman coding:

- Quaternary Huffman encoding
- Quaternary Huffman decoding

Table 3 Comparison of binary and quaternary tree

Parameter	Binary tree	Quaternary tree
Level	6	3
Total node	37	25
Internal node	18	6
Weighted path length	190	97

Encoding algorithm

Encoding is a two-pass problem. The first pass is to determine the frequencies of letters. We use this information to create the quaternary Huffman tree. We have used a dictionary to store the frequencies of the symbols. When a quaternary Huffman code has been generated, the symbol will be replaced by the code. This is a modification of Huffman algorithm (Coreman et al. 2001).

Algorithm 1: Encoding of Quaternary Huffman Tree

```

Q- HUFFMAN (C)
1.      Q ← C
2.      n ← |Q|
3.      i ← n
4.      WHILE i > 1
5.          allocate a new node z

6.          left[z] ← v ← EXTRACT-MIN(Q)
7.          left-mid[z] ← w ← EXTRACT-MIN(Q)

8.          IF i = 2
9.              f [z] ← f[v] + f[w]
10.         ELSE IF i =3
11.             right-mid [z] ← x ← EXTRACT-MIN(Q)
12.             f [z] ← f[v] + f[w] + f[x]
13.         ELSE
14.             right-mid [z] ← x ← EXTRACT-MIN(Q)
15.             right [z] ← y ← EXTRACT-MIN(Q)
16.             f [z] ← f[v] + f[w] + f[x] + f[y]
17.         END IF

18.         INSERT(Q, z)

19.         i ← |Q|

20.     END WHILE

21.     RETURN EXTRACT-MIN(Q)

```

In line 1, we assign the unordered nodes, C in the queue, Q and later we take the count of nodes in Q and assign it to n . We assign the value of n to a new variable i . In line 4, we start iterating all the nodes in queue to build the quaternary tree until the count of i is greater than 1 which means that there are nodes still left to be added to the parent. In line 5, a new tree node, z is allocated. This node will be the parent node of the least frequent nodes. In line 6, we extract the least frequent node from the queue Q and assign it as a left child of the parent node z . The $\text{EXTRACT-MIN}(Q)$ function returns the least frequent node from the queue and removes it from the queue as well. In line 7, we take the next least frequent node from the queue and assign it as a left-mid child of the parent z .

From line 8 to 17, we check the value of i or the number of nodes left in the queue Q . If i equals 2, the frequency of the parent node z , $f[z]$ will be the summation of the frequency of node v , $f[v]$ and the frequency of node w , $f[w]$. Likewise, for i is equal to 3, we extract another least frequent node from the queue and add it as a child and add its frequency to the parent node. For i is greater than 3, we extract two least frequent nodes and add them as right-mid and right child of the parent z and add their frequency to the parent z as well. In line 18, we insert the new parent node z into the queue, Q . In line 19, we take the count of the queue, Q and assign it to i again. The loop continues until a single node is left in the queue. Finally, we return the last and single node from the queue Q as a quaternary Huffman tree.

Decoding algorithm

Decoding is accomplished by reading the encoded data two bits at a time. When iterating the bit stream 00 bit pattern means go LEFT, 01 pattern means go LEFT MID, 10 pattern means go RIGHT MID and 11 pattern means go RIGHT in case of quaternary tree. When a bit pattern matches with a symbol according to the header tree, replace the bit pattern with that symbol and the process is iterated until reached the last bit of the stream.

In the following algorithm 2 in line 1, we assign the quaternary tree T in the local variable ln . Then, we take the total count of bits in n from B . In line 3, we initialize a local variable i with 0 which will be used as a counter. In line 4, we started iterating all the bits in B . As it is a quaternary tree, we have at most four leaves for a parent node: left, left-mid, right-mid, right and 00, 01, 10, 11 represent these leaf nodes, respectively. We take two bits at a time. $\text{EXTRACT-BIT}(B)$ returns a bit from the bit array B and removes it from B as well. In lines 5 and 6, local variables $b1$ and $b2$ are being assigned with two extracted bits from the bit array B .

Algorithm 2: Decoding of Quaternary Huffman Tree

```

DECODE (T, B)
1.       $ln \leftarrow T$ 
2.       $n \leftarrow |B|$ 
3.       $i \leftarrow 0$ 
4.      WHILE  $i \leftarrow n$ 

5.           $b1 \leftarrow \text{EXTRACT-BIT}(B)$ 
6.           $b2 \leftarrow \text{EXTRACT-BIT}(B)$ 

7.          IF  $b1 = 0$  AND  $b2 = 0$ 
8.               $ln \leftarrow \text{LEFT}(ln)$ 
9.          ELSE IF  $b1 = 0$  AND  $b2 = 1$ 
10.              $ln \leftarrow \text{LEFT-MID}(ln)$ 
11.          ELSE IF  $b1 = 1$  AND  $b2 = 0$ 
12.              $ln \leftarrow \text{RIGHT-MID}(ln)$ 
13.          ELSE
14.              $ln \leftarrow \text{RIGHT}(ln)$ 
15.          END IF

16.           $k \leftarrow \text{KEY}(ln)$ 
17.          IF  $k$  IS NOT NULL
18.              Output ( $k$ )
19.               $ln \leftarrow T$ 
20.          END IF
21.           $i \leftarrow i + 2$ 
22.      END WHILE

```

From line 7 to line 15, we check the extracted bits to traverse the tree from the top. If the bits are 00, we take the left child of the parent ln and assign it to ln itself. For 01, we replace the parent ln with its left-mid child, for 10 we replace it with its right-mid child and for 11 we replace it with the right child. In line 16, we get the key of the replaced ln and assign it in k . Then, we check whether k has any value. If the k has any value, we write the value of the k in the output and update the ln with the quaternary tree T itself. In line 21, we increase the value of i by 2 and the loop gets continued and reads the next two bits.

This section discusses the encoding and decoding technique of a quaternary Huffman architecture. The search time for finding a source symbol using quaternary Huffman algorithm is $O(\log_4 n)$, whereas for Huffman-based algorithm it is $O(\log_2 n)$.

Results and discussion

To verify the applicability and feasibility of the proposed quaternary-based technique, experimental evaluation has been performed on real data. The experimental results are compared with regular Huffman-based techniques. Our target was to justify query time and the storage requirements in comparison with regular Huffman-based techniques.

Experimental environment

Each query has been executed five times and the average execution time has been counted. The experiments are conducted on a machine with following specifications:

Operating System (Ubuntu):

Release 12.04(precise) 32-bit

Kernel Linux 3.2.0-26-generic-pae

GNOME 3.4.1

Hardware:

Memory: 2.8 GiB

Processor: Intel® Pentium® Dual CPU T3400 @ 2.16Hz x 2

Compiler:

Mono JIT compiler version 2.10.8.1 (Debian 2.10.8.1-1ubuntu2.1)

Language:

C#

IDE:

MonoDevelop v2.8.6.3

Data set

We have used four real text files as data set. The first two files are the source code of our implemented programs, which we do not wish to share as it is still unpublished. The other two files used for evaluation are readily available online (The famous lgpl 2.1 license. <https://www.gnu.org/licenses/lgpl-2.1.txt>; The transcript of the movie matrix. <http://thematrixtruth.remotewindowlight.com/>). The description of the datasets is given in Table 4.

Table 4 Data set

S/L	File name	Description	File size (bytes)
1	Quaternary-source.txt	The source code of the quaternary Huffman implementation	9861
2	Quaternary-license.txt	The license file of the quaternary Huffman implementation	18,651
3	Lgpl-2.1.txt	The famous lgpl 2.1 license	27,032
4	The-matrix-transcript.txt	The transcript of the movie matrix	46,836

Decoding performance

To measure the decoding performance, we used the dataset on both regular and quaternary Huffman techniques. We consider three techniques as regular Huffman-based techniques (Chung 1997; Hashemian 1995; Chowdhury et al. 2002) and the performance of all three techniques is almost same considering next integer number. We used the *StopWatch* Class under *System.Diagnostic* of Mono framework to calculate the time required. Stopwatch provides a set of methods and properties that can be used to accurately measure elapsed time. The obtained results are described in Table 5. In all cases, we took the average output of at least five runs.

Four source files of different file size have been used altogether to measure the performance. In Table 5, it has been observed that for each case, quaternary Huffman technique is more than 50% faster than the regular Huffman-based techniques in case of decoding time.

In Fig. 3, it has been shown that as file size increases, the quaternary Huffman (*line with diamond shape dot*) technique is performing consistently better than the regular Huffman (*line with square dot*)-based techniques. In some cases depending on the

Table 5 Decoding performance of the proposed method and regular Huffman-based Technique

S/N	Source file	File size (bytes)	Time (ms)		Enhancement rate over regular binary $((RH - QH) * 100)/RH$
			Quaternary Huffman (QH)	Regular Huffman-based techniques (RH) (Chowdhury et al. 2002)	
1	Quaternary-source.txt	9861	3	7	57.14
2	Quaternary-license.txt	18,651	6	12	50.00
3	Lgpl-2.1.txt	27,032	7	16	56.25
4	The-matrix-transcript.txt	46,836	12	27	55.56

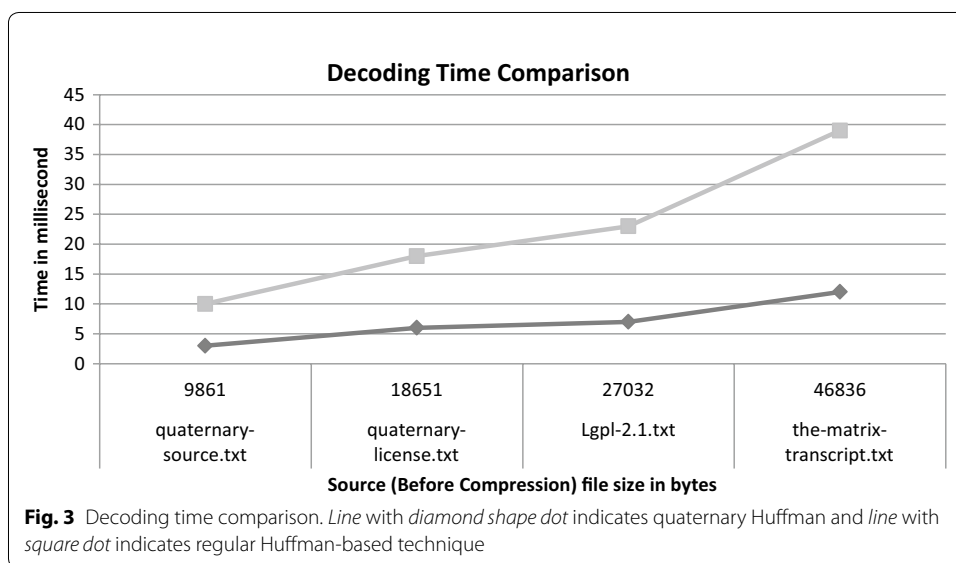


Fig. 3 Decoding time comparison. *Line with diamond shape dot* indicates quaternary Huffman and *line with square dot* indicates regular Huffman-based technique

relative frequencies of the symbols in a file, it is more than two times faster than regular Huffman-based techniques.

To measure the memory usage, we used the dataset on both regular and quaternary Huffman techniques. The method described in Chen et al. (1999) is used for comparison with the proposed method. Table 6 illustrates the compression rate between two techniques. It has been shown that the quaternary technique compresses the original file at an average rate of 32%, whereas the regular Huffman-based technique compresses at an average rate of 39%. Regular Huffman-based technique compresses little better than the proposed quaternary technique, this is just because of quaternary technique produced larger codeword. In some cases, the compression rate is almost equal for both techniques.

The comparison of the compression performance of both techniques using the original file is also shown in Fig. 4 [ash color column indicates original file size, black column indicates regular Huffman-based technique (Chen et al. 1999) and texture column indicates quaternary Huffman technique].

Performance test with reknown corpus and recent Huffman-based techniques

We compare the performance of the proposed technique with Zopfli (Alakuijala and Vandevenne 2013), WinZip (2016) and PKZip (2016) algorithms. Google claims that

Table 6 Compression performance of the proposed technique and regular Huffman-based technique

Source file	Space (byte)			Enhancement rate (Quaternary) $((OS - QH) * 100)/OS$	Enhancement rate (regular) $((OS - RH) * 100)/OS$
	Original size (OS)	Quaternary Huffman (QH)	Huffman-based technique (RH) (Chen et al. 1999)		
Quaternary-source.txt	9861	6958	6347	29.44	35.64
Quaternary-license.txt	18,651	13,520	10,930	27.51	41.40
Lgpl-2.1.txt	27,032	16,042	15,840	40.66	41.40
The-matrix-transcript.txt	46,836	30,909	27,816	34.01	40.61

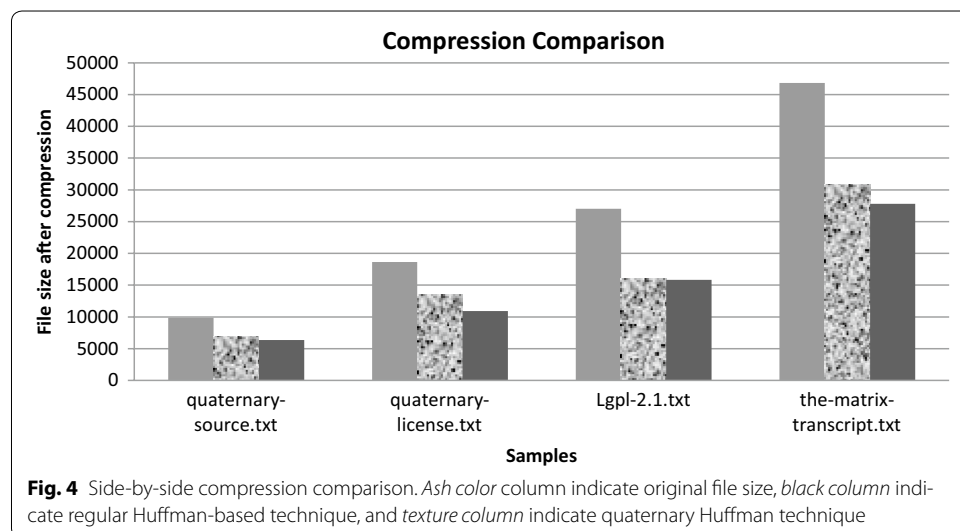


Fig. 4 Side-by-side compression comparison. Ash color indicate original file size, black column indicate regular Huffman-based technique, and texture column indicate quaternary Huffman technique

Zopfli produces the highest compression ratio for similar technique. Zopfli uses Huffman coding to replace each value with a string of bits. WinZip and PKZip are the most widely used recent Huffman-based compression tools. In all cases, we took the average output of five runs.

Table 7 shows the result of compression ratio and compression–decompression speed on the Enwik8 corpus. The Enwik8 corpus is a 95.3-MB file with 156 distinct characters. This corpus is prepared as a large text compression standard, which have 100 million bytes of English Wikipedia.

The result indicates that compression ratio is highest for Zopfli but the compression and decompression speed is very slow. The Zopfli requires over 400 s whereas all other techniques require less than 200 s. If we would compromise between time–space, and when speed is the main factor, then we may choose quaternary technique for this type of large corpus.

Table 8 shows the result of compression ratio and compression–decompression speed on the Canterbury corpus (The Canterbury Corpus. <http://corpus.canterbury.ac.nz/resources/cantrbry.zip>). The Canterbury corpus is 2.67-MB file with 72 distinct characters. This corpus is a modified version of Calgary corpus which is designed to test the compression algorithms.

If we observe the result, it has been shown that compression ratio is highest for Zopfli but its compression and decompression speed is very slow. The Zopfli requires over 13 s whereas all other techniques require less time.

In this section, we have analyzed both techniques thoroughly with different example in terms of time and space. For decoding speed, the proposed quaternary technique

Table 7 Comparison of the proposed technique with recent Huffman-based techniques for Enwik (The Enwik8 Corpus. <http://mattmahoney.net/dc/text.html> <http://mattmahoney.net/dc/enwik8.zip>) corpus

Method/algorithm	Space (MB)	Compression enhancement with respect to original file (%)	Compression–decompression time (s)	Time enhancement with respect to Zopfli (%)
Quaternary	49.67	47.88	186.88	59.66
WinZip	35.2	63.06	187.65	59.49
PKZip	34.5	63.80	195.21	57.86
Zopfli	33.37	64.98	463.26	–

Table 8 Comparison of the proposed technique with recent Huffman-based techniques for Canterbury corpus

Method/algorithm	Space (MB)	Compression enhancement with respect to original file (%)	Compression–decompression time (s)	Time enhancement with respect to Zopfli (%)
Quaternary	1.71	35.95	1.37	89.78
WinZip	0.71	73.40	5.61	46.471
PKZip	0.69	74.15	2.74	21.26
Zopfli	0.64	76.07	13.36	–

outperforms the regular Huffman-based techniques. On the other hand, the compression recital is almost similar for most of the files.

Conclusion

A new lossless compression technique based on Huffman principle is implemented in this paper. We introduced quaternary tree instead of binary tree in Huffman principle. We have shown that representation of Huffman code using quaternary tree is more beneficial than Huffman code using binary tree in terms of processing speed with an insignificant increase in required space. When speed is the main factor, then the quaternary tree based technique performs better than the binary tree based technique. Thus, the proposed technique provides a way to balance between the decoding time and memory usage.

Authors' contributions

The authors discussed the problem and the solutions proposed all together. Both authors participated in drafting and revising the final manuscript. Both authors read and approved the final manuscript.

Acknowledgements

Authors are grateful to ministry of posts, telecommunications and information technology, People's Republic of Bangladesh for their grant to do this research work. The authors would like to thank the anonymous experts for their valuable comments and suggestion for improving the quality of this research paper.

Competing interests

The authors declare that they have no competing interests.

Availability of data

The datasets supporting of this article are available online in the following link.

The famous lgpl 2.1 license, Accessed at <https://www.gnu.org/licenses/lgpl-2.1.txt>

The transcript of the movie Matrix. Accessed at <http://thematrixtruth.remoteviewinglight.com/>

The Enwik8 Corpus. Accessed at <http://mattmahoney.net/dc/text.html> <http://mattmahoney.net/dc/enwik8.zip>

The Canterbury Corpus. Accessed at <http://corpus.canterbury.ac.nz/resources/cantrbry.zip>

The WinZip compression tool, version 1.0.220.1, released by WinZip Computing, S.L., A Corel Company. Accessed at: <http://www.winzip.com/win/en/downwz.html>

The PKZip compression tool, version 14.40.0028, released by PKWARE Inc. Accessed at <https://www.pkware.com/pkzip>

Funding

All the funding provided by the Ministry of Posts, Telecommunications and Information Technology, People's Republic of Bangladesh [Order No: 56.00.0000.028.33.007.14 (part-1)-275, date: 11.05.2014; and Order No: 56.00.0000.028.33.025.14-115, date 10.05.2015]. The above funding gives the financial support for the designing of the study and conducting experiments.

Received: 13 December 2016 Accepted: 26 December 2016

Published online: 07 January 2017

References

- Alakuijala J, Vandevenne L (2013) Data compression using Zopfli. Google Inc. https://zopfli.googlecode.com/file/Data_compression_using_Zopfli.pdf
- Baer M (2006) A general framework for codes involving redundancy minimization. *IEEE Trans Inf Theory* 52:344–349
- Bahadili HA, Hussain SM (2010) A bit-level text compression scheme based on the ACW algorithm. *Int J Autom Comput* 7(1):123–131
- Benetley JL, Sleator DD, Tarjan RE, Wei VK (1986) A locally adaptive data compression scheme. *Commun ACM* 29(4):320–330
- Chen HC, Wang YL, Lan YF (1999) A memory-efficient and fast Huffman decoding algorithm. *Inform Process Lett* 69:119–122
- Chowdhury RA, Kykobad M, King I (2002) An efficient decoding technique for Huffman codes. *Info Process Lett* 81:305–308
- Chung KL (1997) Efficient Huffman decoding. *Inform Process Lett*. 61:97–99
- Coreman TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms. The MIT Press, England
- Fenwick PM (1995) Huffman code efficiencies for extensions of sources. *IEEE Trans Commun* 43:163–165
- Gallager RG (1978) Variations on a theme by Huffman. *IEEE Trans Inf Theory* 24(6):668–674
- Habib A, Hoque ASML, Hussain MR (2013) H-HIBASE: compression enhancement of HIBASE technique using Huffman coding. *J Comput* 8(5):1175–1183
- Hashemian R (1995) Memory efficient and high-speed search Huffman coding. *IEEE Trans Comm* 43(10):2576–2581

- Hermassi H, Rhouma R, Belghith S (2010) Joint compression and encryption using chaotically mutated Huffman trees. *Commun Nonlinear Sci Numer Simulat* 15:2987–2999
- Huffman DA (1952) A method for construction of minimum redundancy codes. *Proc IRE* 40(1952):1098–1101
- Katona GOH, Nemetz TOH (1978) Huffman codes and self information. *IEEE Trans Inform Theory* 22(3):337–340
- Kavousianos X (2008) Test-data compression based on variable-to-variable Huffman encoding with codeword reusability. *IEEE Trans Comput Aided Des Integr Circuits Syst* 27:1333–1338
- Kodituwakku SR, Amarasinghe US (2011) Comparison of lossless data compression algorithms for text data. *Indian J Comput Sci Eng* 1(4):416–426
- Lempel A, Ziv J (1977) A universal algorithm for sequential data compression. *IEEE Trans Inf Theory* 23:337–343
- Lin YK, Huang S-C, Yang CH (2012) A fast algorithm for Huffman decoding based on a recursion Huffman tree. *J Syst Softw* 85:974–980
- Schack R (1994) The length of a typical Huffman codeword. *IEEE Trans Inform Theory* 40(4):1246–1247
- Sharma M (2010) Compression Using Huffman Coding. *Int J Comput Sci Netw Secur* 10(5):133–141
- Suri PR, Goel M (2011) Ternary tree and memory-efficient Huffman decoding algorithm. *Int J Comput Sci Issues* 8(1):483–489
- Szpankowski W (2011) Minimum expected length of fixed-to-variable lossless compression without prefix constraints. *IEEE Trans Inf Theory* 57:4017–4025
- The PKZip compression tool, version 14.40.0028, released by PKWARE Inc., accessed at <https://www.pkware.com/pkzip>. Accessed 19 July 2016
- The WinZip compression tool, version 1.0.220.1, released by WinZip Computing, S.L., A Corel Company. <http://www.winzip.com/win/en/downwz.html>. Accessed 19 July 2016
- Vitter JS (1987) Design and analysis of dynamic Huffman code. *J ACM* 34(4):825–845
- Welch TA (1984) A technique for high-performance data compression. *IEEE Comput* 17(6):8–19
- Wikipedia short history of Huffman coding. http://en.wikipedia.org/wiki/Huffman_coding. Accessed 31 July 2011

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- ▶ Convenient online submission
- ▶ Rigorous peer review
- ▶ Immediate publication on acceptance
- ▶ Open access: articles freely available online
- ▶ High visibility within the field
- ▶ Retaining the copyright to your article

Submit your next manuscript at ▶ springeropen.com
