Ahsan Habib, M. Jahirul Islam and M. Shahidur Rahman

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

Abstract: This article proposes two dynamic Huffman based code generation algorithms, namely Octanary and Hexanary algorithm, for data compression. Faster encoding and decoding process is very important in data compression area. We propose tribit-based (Octanary) and quadbit-based (Hexanary) algorithm and compare the performance with the existing widely used single bit (Binary) and recently introduced dibit (Quaternary) algorithms. The decoding algorithms for the proposed techniques have also been described. After assessing all the results, it is found that the Octanary and the Hexanary techniques perform better than the existing techniques in terms of encoding and decoding speed.

Keywords: Binary Tree, Quaternary Tree, Octanary Tree, Hexanary Tree, Huffman Principle, Decoding Technique, Encoding Technique, Tree Data Structure, Data Compression

Introduction

Huffman coding (Huffman, 1952) is very popular in data compression area. Nowadays it is used in data compression for wireless and sensor networks (Săcăleanu et al., 2011; Renugadevi and Darisini, 2013), data mining (Oswald and Sivaselvan, 2018; Oswald et al., 2015). It is also found efficient for data compression in low resource systems (Radhakrishnan et al., 2016; Matai et al., 2014; Wang and Lin, 2016). The use of Huffman code in word-based text compression is also very common (Sinaga, 2015). Huffman principle produces optimal code using a Binary tree where the most frequent codewords are smaller in length. However, Huffman principle does not produce a balanced tree (Rajput, 2018). For this reason, it requires more memory to store longer codeword, and thus it also requires more time to decode those codewords from the memory. In this paper, we first review traditional Huffman algorithm and newly introduced Quaternary Huffman algorithm. Then we introduce Octanary and Hexanary tree for construction of Huffman codes. Octanary and Hexanary structure makes the underlying tree more balanced. The tree construction and decoding algorithms for both techniques have been developed. The codeword efficiency of Binary, Quaternary, Octanary and Hexanary structure have been compared. The compression ratio and speed are also compared for different methods using these coding systems. It is found that the compression and decompression speed of the proposed techniques are better than the others. To summarize, the proposed techniques may be suitable for offline data compression applications where encoding and decoding speed is more important with less constraint on space.

Related Works

In 1952, David Huffman introduced an algorithm (Huffman, 1952) which produced optimal code for data compression system. Huffman code is produced using Binary tree technique, where more frequent symbols produce shorter codeword length and less frequent symbols produce longer codeword. Later on, so many popular algorithms and applications have been developed based on Binary Huffman coding technique. Saradashri et al. (2015) explained in his book that Huffman code could also be static or dynamic. Chen et al. (Chen et al., 1999) introduced a method to speed up the process and reduced memory of Huffman tree. A tree clustering algorithm is introduced in (Hashemian, 1995) to avoid high sparsity of the tree. In this research, the author reduced the header size dramatically. Vitter (1987), the author introduced a new method where it required less memory than the conventional Huffman technique. Chung (1997) also introduced a memory-efficient array structure to represent the Huffman tree. In some other researches codeword length of Huffman code also investigated. Katona and Nemetz (1978) investigated the connection...
between self-information of a source letter and its codeword length. A recursive Huffman algorithm is introduced in (Lin et al., 2012), where a tree is transformed into a recursive Huffman tree and it decoded more than one symbol at a time. The decoding process starts by reading a file bit by bit in all of the above techniques. Recently, we introduced a code generation technique based on Quaternary (dibit) Huffman tree (Habib and Rahman, 2017) to produce Huffman codes. In this research, better encoding and decoding speed is achieved by sacrificing an insignificant amount of space, where it is also found that searching two bits at a time speeds up the overall processing speed than searching a single bit. This motivated us to search three or four bits at a time. In this connection, the Octanary algorithm is introduced to produce three bit based Huffman code, whereas the Hexanary algorithm is introduced to produce four bit based Huffman code. The proposed algorithms improved the Huffman decoding time compared with the existing Huffman algorithms.

We organize the paper as follows. In section “Tree Structure”, traditional Binary, Quaternary, Octanary and Hexanary tree structures in data management system are presented. In section “Implementation”, the proposed encoding and decoding algorithm of Octanary and Hexanary techniques have been presented. Section “Result and Discussion” discusses the experimental results. Finally, Section “Conclusion” concludes the paper.

Tree Structure

Binary and Quaternary Tree

A rooted tree $T$ is called an $m$-ary tree if every internal vertex has no more than $m$ children. The tree is called a full $m$-ary tree if every internal vertex has exactly $m$ children. An $m$-ary tree with $m = 2$ is called a Binary tree. In a Binary tree, if an internal vertex has two children, the first child is called the LEFT child and the second child is called the RIGHT child (Adamchik, 2009). The Binary tree structure is thoroughly discussed in (Huffman, 1952). A tree with $m = 4$ is called a Quaternary tree, which has at most four children, the first child is called LEFT child, the second child is called LEFT-MID child, the third child is called RIGHT-MID child and the fourth child is called RIGHT child. The detail of Quaternary tree structures is explained in (Habib and Rahman, 2017). The Binary and Quaternary tree structures for Luke 5 (Luke 5, 2018) are shown in Fig. 1 and 2, respectively. Luke 5 is the fifth chapter of the Gospel of Luke in the New Testament of the Christian Bible. The chapter relates the recruitment of Jesus’ first disciples and continues to describe Jesus’ teaching and healing ministry (Luke 5, 2018). The frequency distribution of Luke 5 is shown in Fig. 3.

Octanary Tree

Octanary tree or 8-ary tree is a tree in which each node has 0 to 8 children (labeled as LEFT1 child, LEFT2 child, LEFT3 child, LEFT4 child, RIGHT1 child, RIGHT2 child, RIGHT3 child, RIGHT4 child). Here for constructing codes for Octanary Huffman tree, we use 000 for a LEFT1 child, 001 for a LEFT2 child, 010 for a LEFT3 child, 011 for a LEFT4 child, 100 for a RIGHT1 child, 101 for a RIGHT2 child, 110 for a RIGHT3 child and 111 for a RIGHT4 child.

The process of the construction of an Octanary tree is described below:

- List all possible symbols with their probabilities;
- Find the eight symbols with the smallest probabilities
- Replace these by a single set containing all eight symbols, whose probability is the sum of the individual probabilities
- Repeat until the list contains single member
- The octanary tree structure for Luke 5 data is shown in Fig. 4.

Hexanary Tree

Hexanary tree or 16-ary tree is a tree in which each node has 0 to 16 children (labeled as LEFT1 child, LEFT2 child, LEFT3 child, LEFT4 child, LEFT5 child, LEFT6 child, LEFT7 child, LEFT8 child, RIGHT1 child, RIGHT2 child, RIGHT3 child, RIGHT4 child, RIGHT5 child, RIGHT6 child, RIGHT7 child, RIGHT8 child). Here for constructing codes for Hexanary Huffman tree we use 0000 for LEFT1 child, 0001 for LEFT2 child, 0010 for LEFT3 child, 0011 for LEFT4 child, 0100 for LEFT5 child, 0101 for LEFT6 child, 0110 for LEFT7 child, 0111 for LEFT8 child, 1000 for RIGHT1 child, 1001 for RIGHT2 child, 1010 for RIGHT3 child, 1011 for RIGHT4 child, 1100 for RIGHT5 child, 1101 for RIGHT6 child, 1110 for RIGHT7 child and 1111 for RIGHT8 child.

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

- List all possible symbols with their probabilities
- Find the sixteen symbols with the smallest probabilities
- Replace these by a single set containing all sixteen symbols, whose probability is the sum of the individual probabilities
- Repeat until the list contains single member
- The Hexanary tree structure for Luke 5 data is shown in Fig. 5.
Fig. 1: Binary tree

Fig. 2: Quaternary tree

Fig. 3: Frequency distribution of Luke5
Fig. 4: Octanary tree

Fig. 5: Hexanary tree

Table 1: Comparison of different tree structures

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Binary</th>
<th>Quaternary</th>
<th>Octanary</th>
<th>Hexanary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level of tree</td>
<td>10</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Number of internal node</td>
<td>25</td>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total number of nodes</td>
<td>51</td>
<td>35</td>
<td>30</td>
<td>28</td>
</tr>
<tr>
<td>Weighted path length</td>
<td>784023</td>
<td>497301</td>
<td>327063</td>
<td>236130</td>
</tr>
</tbody>
</table>

Implementation

Code Generation (Encoding) Algorithm

To construct Huffman tree, distinct symbols and its frequency are necessary. The tree construction algorithm for the traditional Binary technique is explained in (Cormen et al., 1989). The newly constructed Quaternary technique is explained in (Habib and Rahman, 2017). In this section, newly constructed Octanary and Hexanary tree generation algorithms are illustrated.

Encoding of Octanary Huffman Tree

The encoding algorithm for Octanary Huffman tree is shown in algorithm 1. In line 1 we assign the un-ordered nodes, C in the Queue, Q and later we take the count of nodes in Q and assign it to n. We declare a variable i and assign the value of n to it.
In line 4, we start iterating all the nodes in the queue to build the Octanary tree until the count of \( i \) is greater than \( j \) which means 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 \( LEFT1 \) child of the parent node \( z \). The purpose of the \( EXTRACT-MIN (Q) \) function is to return the least frequent node from the queue. It also removes least frequent node from the queue. In line 7, we take the next least frequent node from the queue and assign it as a \( LEFT2 \) child of the parent \( z \).

From line 8 to 43, we check the value of \( i \) or the number of nodes left in the queue \( Q \). If \( i \) is equal to exactly 2, the frequency of the parent node \( z, f[z] \) will be the summation of the frequency of node \( r, f[r] \) and the frequency of node \( s, f[s] \). For \( i \) is equal to 3 we extract another least frequent node from the queue and add it as \( LEFT1, LEFT2, LEFT3 \) child and add its frequency to the parent node. Likewise, for \( i \) is equal to 4 we extract four least frequent node from the queue and add it as \( LEFT1, LEFT2, LEFT3, LEFT4 \) child and add its frequency to the parent node. For \( i \) is equal to 5 we extract five least frequent node from the queue and add it as \( LEFT1, LEFT2, LEFT3, LEFT4, RIGHT1 \) child and add its frequency to the parent node. For \( i \) is equal to 6 we extract six least frequent node from the queue and add it as \( LEFT1, LEFT2, LEFT3, LEFT4, RIGHT1, RIGHT2 \) child and add its frequency to the parent node.

**Algorithm 1. Encoding of Octanary Huffman Tree**

O-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. left1[z] \( \leftarrow r \leftarrow EXTRACT-MIN(Q) \)
7. left2[z] \( \leftarrow s \leftarrow EXTRACT-MIN(Q) \)
8. \( \text{IF } i = 2 \)
9. \( f[z] \leftarrow f[r] + f[s] \)
10. **ELSE IF** \( i = 3 \)
11. left3[z] \( \leftarrow t \leftarrow EXTRACT-MIN(Q) \)
12. \( f[z] \leftarrow f[r] + f[s] + f[t] \)
13. **ELSE IF** \( i = 4 \)
14. left3[z] \( \leftarrow t \leftarrow EXTRACT-MIN(Q) \)
15. left4[z] \( \leftarrow u \leftarrow EXTRACT-MIN(Q) \)
16. \( f[z] \leftarrow f[r] + f[s] + f[t] + f[u] \)
17. **ELSE IF** \( i = 5 \)
18. left3[z] \( \leftarrow t \leftarrow EXTRACT-MIN(Q) \)
19. left3[z] \( \leftarrow u \leftarrow EXTRACT-MIN(Q) \)
20. right1[z] \( \leftarrow v \leftarrow EXTRACT-MIN(Q) \)
21. \( f[z] \leftarrow f[r] + f[s] + f[t] + f[u] + f[v] \)
22. **ELSE IF** \( i = 6 \)
23. left3[z] \( \leftarrow t \leftarrow EXTRACT-MIN(Q) \)
24. left4[z] \( \leftarrow u \leftarrow EXTRACT-MIN(Q) \)
25. right1[z] \( \leftarrow v \leftarrow EXTRACT-MIN(Q) \)
26. right2[z] \( \leftarrow w \leftarrow EXTRACT-MIN(Q) \)
27. \( f[z] \leftarrow f[r] + f[s] + f[t] + f[u] + f[v] + f[w] \)
28. **ELSE** IF \( i = 7 \)
29. left3[z] \( \leftarrow t \leftarrow EXTRACT-MIN(Q) \)
30. left4[z] \( \leftarrow u \leftarrow EXTRACT-MIN(Q) \)
31. right1[z] \( \leftarrow v \leftarrow EXTRACT-MIN(Q) \)
32. right2[z] \( \leftarrow w \leftarrow EXTRACT-MIN(Q) \)
33. right3[z] \( \leftarrow x \leftarrow EXTRACT-MIN(Q) \)
34. \( f[z] \leftarrow f[r] + f[s] + f[t] + f[u] + f[v] + f[w] + f[x] \)
35. **ELSE**
36. left3[z] \( \leftarrow t \leftarrow EXTRACT-MIN(Q) \)
37. left4[z] \( \leftarrow u \leftarrow EXTRACT-MIN(Q) \)
38. right1[z] \( \leftarrow v \leftarrow EXTRACT-MIN(Q) \)
39. right2[z] \( \leftarrow w \leftarrow EXTRACT-MIN(Q) \)
40. right3[z] \( \leftarrow x \leftarrow EXTRACT-MIN(Q) \)
41. right4[z] \( \leftarrow y \leftarrow EXTRACT-MIN(Q) \)
43. **END IF**
44. **END WHILE**
45. **END WHILE**
46. **RETURN** \( EXTRACT-MIN(Q) \)

For \( i \) is equal to 7 we extract seven least frequent node from the queue and add it as \( LEFT1, LEFT2, RIGHT1, RIGHT2, RIGHT3 \) child and add its frequency to the parent node. Likewise, for \( i \) is equal to 8 we extract eight least frequent node from the queue and add it as \( LEFT1, LEFT2, RIGHT1, RIGHT2, RIGHT3, LEFT4 \) child and add its frequency to the parent node. In line 44, we insert the new parent node \( z \) into the Queue, \( Q \). In line 45, we take the count of the queue, \( Q \) and assign it to \( i \) again. And, the loop continues until a single node left in the queue. Finally, the last and single node from the queue \( Q \) is returned as an Octanary Huffman tree.

**Encoding of Hexanary Huffman Tree**

In line 1 we are assigning the un-ordered nodes, \( C \) in the Queue, \( Q \) and later we are taking the count of nodes in \( Q \) and assigning it to \( n \). We declare a variable \( i \) and assign the value of \( n \) to it. In line 4, we start iterating all the nodes in the queue to build the Hexanary tree until the count of \( i \) is greater than \( l \) which means 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 \( LEFT1 \) child of the parent node \( z \). The purpose of the \( EXTRACT-MIN (Q) \) function is to return the least frequent node from the queue. It also removes least frequent node from the queue. In line 7, we take the next least frequent node from the queue and assign it as a \( LEFT2 \) child of the parent \( z \).

From line 8 to 43, we check the value of \( i \) or the number of nodes left in the queue \( Q \). If \( i \) is equal to exactly 2, the frequency of the parent node \( z, f[z] \) will
be the summation of the frequency of node $j$, $f[j]$ and the frequency of node $k$, $f[k]$. For $i$ is equal to 3 we extract another least frequent node from the queue and add it as $LEFT1$, $LEFT2$, $LEFT3$ child and add its frequency to the parent node. Likewise, for $i$ is equal to 4 we extract four least frequent node from the queue and add it as $LEFT1$, $LEFT2$, $LEFT3$, $LEFT4$ and $LEFT5$ child and add its frequency to the parent node. For $i$ is equal to 5 we extract five least frequent node from the queue and add it as $LEFT1$, $LEFT2$, $LEFT3$, $LEFT4$, $LEFT5$, $LEFT6$ child and add its frequency to the parent node. For $i$ is equal to 6 we extract six least frequent node from the queue and add it as $LEFT1$, $LEFT2$, $LEFT3$, $LEFT4$, $LEFT5$, $LEFT6$, $LEFT7$ child and add its frequency to the parent node. For $i$ is equal to 7 we extract seven least frequent node from the queue and add it as $LEFT1$, $LEFT2$, $LEFT3$, $LEFT4$, $LEFT5$, $LEFT6$, $LEFT7$, $LEFT8$ child and add its frequency to the parent node. The process will be continued and for $i$ is equal to 8 we extract eight least frequent node from the queue and add it as $LEFT1$, $LEFT2$, $LEFT3$, $LEFT4$, $LEFT5$, $LEFT6$, $LEFT7$, $LEFT8$, $RIGHT1$, $RIGHT2$, $RIGHT3$, $RIGHT4$, $RIGHT5$, $RIGHT6$, $RIGHT7$, $RIGHT8$ child and add its frequency to the parent node. In line 122, we insert the new parent node $z$ into the Queue, $Q$. In line 123, we take the count of the node $Q$ and assign it to $i$ again. And, the loop continues until a single node left in the queue. Finally, we return the last and single node from the queue $Q$ as a Hexanary Huffman tree.

**Algorithm 2. Encoding of Hexanary Huffman Tree**

1. $Q \leftarrow C$
2. $n \leftarrow |Q|
3. $i \leftarrow n$
4. **WHILE** $1 > 1$
5. allocate a new node $z$
6. left1$[z] \leftarrow j \leftarrow \text{EXTRACT-MIN}(Q)$
7. left2$[z] \leftarrow k \leftarrow \text{EXTRACT-MIN}(Q)$
8. **IF** $i = 2$
9. $f[z] \leftarrow f[j] + f[k]$
10. **ELSE** **IF** $i = 3$
11. left3$[z] \leftarrow l \leftarrow \text{EXTRACT-MIN}(Q)$
12. $f[z] \leftarrow f[j] + f[k] + f[l]$
13. **ELSE** **IF** $i = 4$
14. left4$[z] \leftarrow m \leftarrow \text{EXTRACT-MIN}(Q)$
15. $f[z] \leftarrow f[j] + f[k] + f[l] + f[m]$
16. **ELSE** **IF** $i = 5$
17. left5$[z] \leftarrow n \leftarrow \text{EXTRACT-MIN}(Q)$
18. $f[z] \leftarrow f[j] + f[k] + f[l] + f[m] + f[n]$
19. **ELSE** **IF** $i = 6$
20. left6$[z] \leftarrow o \leftarrow \text{EXTRACT-MIN}(Q)$
22. **ELSE** **IF** $i = 7$
23. left7$[z] \leftarrow l \leftarrow \text{EXTRACT-MIN}(Q)$
25. **ELSE** **IF** $i = 8$
26. left8$[z] \leftarrow o \leftarrow \text{EXTRACT-MIN}(Q)$
28. **END IF**
29. **END WHILE**
30. **RETURN** $\text{EXTRACT-MIN}(Q)$

**Decoding Algorithm**

This is a one pass algorithm. First, open the encoded file and read the frequency data out of it. Create the Octanary or Hexanary Huffman tree base on that information. Read data out of the file and search the tree to find the correct character to decode. Decode bit means go $LEFT1$, $001$ bit means go $LEFT2$, $010$ bit means go $LEFT3$, etc in case of the Octanary tree; $0001$ bit means go $LEFT1$, $0011$ bit means go $LEFT2$, $0101$ bit means go $LEFT3$, etc in case of the Hexanary tree). If we know the Octanary or Hexanary Huffman code for some encoded data, decoding may be accomplished by reading the encoded data three or four bit at a time. Once the bits read match a code for a symbol, write out the symbol and start collecting bits again. The newly constructed Octanary and Hexanary tree decoding techniques are explained below.

**Decoding of Octanary Huffman Tree**

**Algorithm 3. Decoding of Octanary Huffman Tree**

1. $ln \leftarrow T$
2. $n \leftarrow |B|
3. $i \leftarrow 0$
4. **WHILE** $i < n$
5. $bl \leftarrow \text{EXTRACT-BIT}(B)$
6. $b2 \leftarrow \text{EXTRACT-BIT}(B)$
In line 1, we assign the Octanary tree \( T \) in the local variable \( ln \). After that the total count of bits in \( n \) from \( B \) is taken. In line 3, a local variable \( i \) with 0 is initialized which will be used as a counter. In line 4, we start iterating all the bits in \( B \). As it is a Hexanary tree, we have at most sixteen leaves for a parent node: \( LEFT1, LEFT2, LEFT3, LEFT4, LEFT5, LEFT6, LEFT7, RIGHT1, RIGHT2, RIGHT3, RIGHT4, RIGHT5, RIGHT6, RIGHT7, RIGHT8 \) and 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 represent these leaf nodes respectively. So, we take four bits at a time. \( EXTRACT-BIT(B) \), returns a bit from the bit array \( B \) and removes it from \( B \) as well. In line 5, 6, 7 and 8, local variable \( b1, b2, b3 \) and \( b4 \) is being assigned with four extracted bits from the bit array \( B \).

From line 8 to line 24, we check the extracted bits to traverse the tree from the top. If the bits are 0000 we take the \( LEFT1 \) child of the parent \( ln \) and assign it to \( ln \) itself. For 0001, we replace the parent \( ln \) with its \( LEFT2 \) child, for 0010 we replace it with its \( LEFT3 \) child, for 0100 we replace the parent \( ln \) with its \( LEFT5 \) child, for 0101 we replace it with its \( LEFT6 \) child, for 0110 we replace it with the \( LEFT7 \) child, for 0111 we replace it with its \( LEFT8 \) child, for 1000 we replace it with its \( RIGHT1 \) child, for 1001 we replace it with its \( RIGHT2 \) child, for 1010 we replace it with its \( RIGHT3 \).

Algorithm 4. Decoding of Hexanary Huffman Tree

Decoding of Hexanary Huffman Tree

In line 1, we assign the Hexanary tree \( T \) in the local variable \( ln \). After that the total count of bits in \( n \) from \( B \) is taken. In line 3, a local variable \( i \) with 0 is initialized which will be used as a counter. In line 4, we start iterating all the bits in \( B \). As it is a Hexanary tree, we have at most sixteen leaves for a parent node: \( LEFT1, LEFT2, LEFT3, LEFT4, LEFT5, LEFT6, LEFT7, RIGHT1, RIGHT2, RIGHT3, RIGHT4, RIGHT5, RIGHT6, RIGHT7, RIGHT8 \) and 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 represent these leaf nodes respectively. So, we take four bits at a time. \( EXTRACT-BIT(B) \), returns a bit from the bit array \( B \) and removes it from \( B \) as well. In line 5, 6, 7 and 8, local variable \( b1, b2, b3 \) and \( b4 \) is being assigned with four extracted bits from the bit array \( B \).

From line 8 to line 24, we check the extracted bits to traverse the tree from the top. If the bits are 0000 we take the \( LEFT1 \) child of the parent \( ln \) and assign it to \( ln \) itself. For 0001, we replace the parent \( ln \) with its \( LEFT2 \) child, for 0010 we replace it with its \( LEFT3 \) child, for 0011 we replace it with the \( LEFT4 \) child, for 0100 we replace the parent \( ln \) with its \( LEFT5 \) child, for 0101 we replace it with its \( LEFT6 \) child, for 0110 we replace it with the \( LEFT7 \) child, for 0111 we replace it with its \( LEFT8 \) child, for 1000 we replace it with its \( RIGHT1 \) child, for 1001 we replace it with its \( RIGHT2 \) child, for 1010 we replace it with its \( RIGHT3 \).

Algorithm 4. Decoding of Hexanary Huffman Tree

Decoding of Hexanary Huffman Tree
we write the value of the check whether key of the replaced we replace it with its continued and read the next four bits.

Hexanary techniques have been thoroughly discussed in line 42, 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 Hexanary tree \( T \) itself. In line 47 we increase the value of \( i \) by 4 and the loops get continued and read the next four bits.

Encoding and Decoding Techniques of Octanary and Hexanary techniques have been thoroughly discussed in this section. The search time for finding the source symbol using Octanary and Hexanary Huffman Tree is \( O(\log_8 n) \) and \( O(\log_8 n) \), respectively, whereas for Huffman based techniques decoding algorithm it is \( O(\log_2 n) \). The codeword generated by each technique are shown in Fig. 6.

### Results and Discussion

The objective of this experiment is to evaluate the performance of several Huffman based algorithms. We consider Zopfli (Alakuijala and Vandevenne, 2013; Alakuijala et al., 2016) as a traditional (Binary) Huffman algorithm. Zopfli is one of the most successful compression algorithm released by Google Inc. Google claims that Zopfli has the highest compression ratio. We also compare the performance of the dibit based Quaternary algorithm and the proposed tribit based Octanary and quadbit based Hexanary Huffman algorithms. We run all algorithms in the same computer with Intel® Core™ i5 – 6500 CPU running at 3.20 GHz with 2 cores and 4 additional hyper threading contexts. We run Ubuntu14.04 LTS Operating system. All codecs were compiled using the same compiler, GCC 4.8.4. The amount of primary memory is 4 GiB DDR4 type. We execute every query five times and count average time. The dataset used in this experiment to verify the performance of different algorithms are described in Table 2.

As shown in Table 3, it is observed that compression ratio is highest for Zopfli but the respective compression and decompression speed is very slow. The Zopfli requires over 400 sec whereas all other proposed techniques require less than 200 sec.

For the Canterbury corpus, Zopfli requires over 13 sec whereas all other proposed techniques require less than 2 sec, which is shown in Table 4.

The performance of different algorithms is shown in Table 3 and 4 for Enwik (Mahoney, 2018) and Canterbury (Bell and Powel, 2000) corpora, respectively. From the both tables, it is shown that the valuation of two different parameter space and time are not same. In some cases saving space is more important and in some other cases speed (time) is important. To see a time-space relation at the same time, we normalize the data. If we divide every number by the largest number of the range, we will get every number in the range between 0 and 1. The data before and after normalization for Enwik corpus is shown in Table 5 and the time-space graph is shown in Fig. 7.

From Fig. 7, it has been shown that Zopfli requires maximum time whereas Quaternary, Octanary or Hexanary requires less time. In the Quaternary technique, it achieves almost 60% speed improvement with sacrificing 17% of space. For Octanary technique, it achieves almost 59% more speed with sacrificing 29% of space. From Fig. 8 in the performance of Canterbury corpus, it is shown that almost 90% speed improvement can be achieved by sacrificing 40% of space.
**Fig. 7:** Time-space requirement for Enwik corpus

**Fig. 8:** Normalized Time-Space requirement for Canterbury corpus

**Fig. 9:** Octanary performance for “Consultation-en”
Table 2: Data set

<table>
<thead>
<tr>
<th>S/L</th>
<th>File name</th>
<th>Description</th>
<th>File size</th>
<th>Distinct symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Enwik8.txt</td>
<td>It has been developed as a large text compression benchmark, consisting of 100 million bytes of English Wikipedia</td>
<td>95.3 MB</td>
<td>156</td>
</tr>
<tr>
<td>2</td>
<td>Canterbury.txt</td>
<td>A compression corpus designed for lossless data compression, Improved version of Calgary corpus</td>
<td>2.67 MB</td>
<td>72</td>
</tr>
<tr>
<td>3</td>
<td>consultation-document_en.pdf</td>
<td>Public Consultation on the review of the EU copyright rules</td>
<td>113 KB</td>
<td>92</td>
</tr>
</tbody>
</table>

Table 3: The compression ratio and compression-decompression speed for the Enwik Corpus

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Space (MB)</th>
<th>Compression enhancement with respect to original file (in %)</th>
<th>Time (S)</th>
<th>Time enhancement with respect to Zopfli (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zopfli (Binary)</td>
<td>33.37</td>
<td>64.98</td>
<td>463.26</td>
<td>-</td>
</tr>
<tr>
<td>Quaternary</td>
<td>49.67</td>
<td>47.88</td>
<td>186.88</td>
<td>59.66</td>
</tr>
<tr>
<td>Octanary</td>
<td>61.06</td>
<td>35.93</td>
<td>187.82</td>
<td>59.46</td>
</tr>
<tr>
<td>Hexanary</td>
<td>59.73</td>
<td>37.32</td>
<td>174.58</td>
<td>62.31</td>
</tr>
</tbody>
</table>

Table 4: The compression ratio and compression-decompression speed for the Canterbury corpus

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Space (MB)</th>
<th>Compression enhancement with respect to original file (in %)</th>
<th>Time (S)</th>
<th>Time enhancement with respect to Zopfli (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zopfli (Binary)</td>
<td>0.64</td>
<td>76.07</td>
<td>13.36</td>
<td>-</td>
</tr>
<tr>
<td>Quaternary</td>
<td>1.71</td>
<td>35.85</td>
<td>1.37</td>
<td>89.78</td>
</tr>
<tr>
<td>Octanary</td>
<td>2.27</td>
<td>15.01</td>
<td>1.47</td>
<td>89.00</td>
</tr>
<tr>
<td>Hexanary</td>
<td>1.79</td>
<td>32.97</td>
<td>1.04</td>
<td>92.20</td>
</tr>
</tbody>
</table>

Table 5: Time-space data for Enwik corpus

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Space (MB)</th>
<th>Time (S)</th>
<th>After normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zopfli (Binary)</td>
<td>33.37</td>
<td>463.26</td>
<td>Zopfli (Binary) 0.55</td>
</tr>
<tr>
<td>Quaternary</td>
<td>49.67</td>
<td>186.88</td>
<td>Quaternary 0.81</td>
</tr>
<tr>
<td>Octanary</td>
<td>61.06</td>
<td>187.82</td>
<td>Octanary 1.00</td>
</tr>
<tr>
<td>Hexanary</td>
<td>59.73</td>
<td>174.58</td>
<td>Hexanary 0.98</td>
</tr>
</tbody>
</table>

It is not always true that Quaternary technique perform better than the other techniques. For Consultation-en (EC, 2013) documents, it has been observed that Octanary perform better than the other techniques. It is found that for both time and space Octanary achieved the best performance, which is shown in Fig. 9. When the number of symbol is approximately $8^h$ ($h$ is the height of the tree) then the Octanary performs better than the other techniques.

Conclusion

Two new Huffman based algorithms have been introduced in this article. The time-space trade-off for different Huffman based algorithms have been thoroughly discussed. Binary Huffman algorithm performs better for achieving more compression ratio. Quaternary Huffman algorithm is useful when a balance between time and space is required. However, if the tree is balanced, due to less tree-height Octanary and Hexanary Huffman algorithms perform superior to Binary and Quaternary algorithms. In all cases, optimal codeword is produced when the tree is balanced. Binary, Quaternary, Octanary and Hexanary algorithms perform best when the number of symbols is approximately $2^h$, $4^h$, $8^h$ and $16^h$, respectively, where $h$ is the height of the tree. An adaptive algorithm on how to find the most suitable encoding algorithm for balancing speed and memory requirement could be an important topic for future research.

Acknowledgment

We are grateful to ICT Division, Ministry of Posts, Telecommunications and Information Technology, People's Republic of Bangladesh for their grant to conduct this research work.

Funding Information

Fund is provided by the ICT Division, Ministry of Posts, Telecommunications and Information Technology, People's Republic of Bangladesh (Order No: 56.00.0000.028.33.007.14 (part-l)-275, date: 11.05.2014).

Author's Contributions

Ahsan Habib: Contributed in the original conception and algorithm design of the research work, drafted the article and produce the figures used in the manuscript.
M. Jahirul Islam: Contributed in the conception and design of the research work, reviewed the manuscript and gave final approval of the final version of the manuscript.

M. Shahidur Rahman: Contributed in the conception and design of the research work, reviewed the manuscript critically and gave final approval of the final version of the manuscript.

Ethics

This research manuscript is original and has not been published elsewhere. The corresponding author confirms that all of the other authors have read and approved the manuscript and there are no ethical issues involved.

References


