Journal of Computer Science

Original Research Paper

Huffman Based Code Generation Algorithms: Data Compression Perspectives

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

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

Article history
Received: 18-07-2018
Revised: 07-09-2018
Accepted: 06-12-2018

Corresponding Author:
Ahsan Habib
Department of Computer Science and Engineering,
Shahjalal University of Science and Technology, Sylhet,
Bangladesh
Email: ahabib-cse@sust.edu

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 speed 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 octanry 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

DOI: 10.3844/jcssp.2018.1599.1610

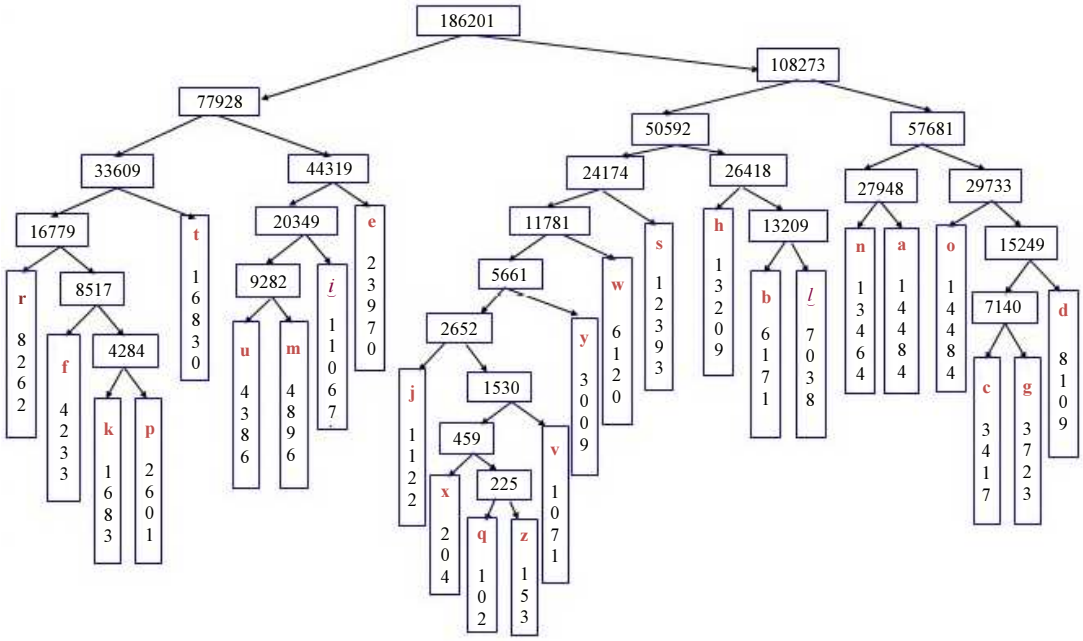

Fig. 1: Binary tree

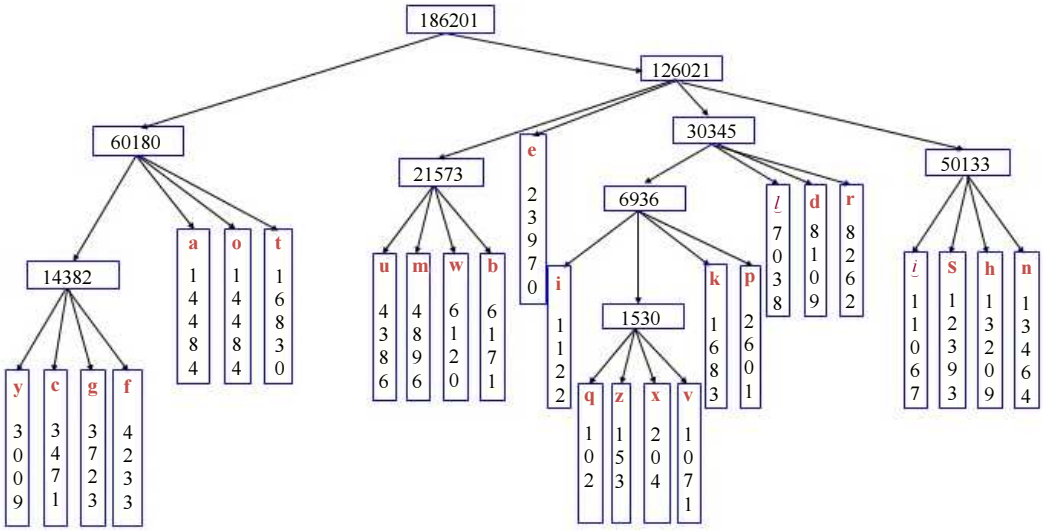

Fig. 2: Quaternary tree

Symbol	q	z	x	v	j	k	p	y	c	g	f	u	m	w	b	l	d	r	i	s	h	n	a	o	t	e
Frequency	102	153	204	1071	1122	1683	2601	3009	3417	3723	4233	4386	4896	6120	6171	7038	8109	8262	11067	12393	13209	13464	14484	14484	16830	23970

Fig. 3: Frequency distribution of Luke5

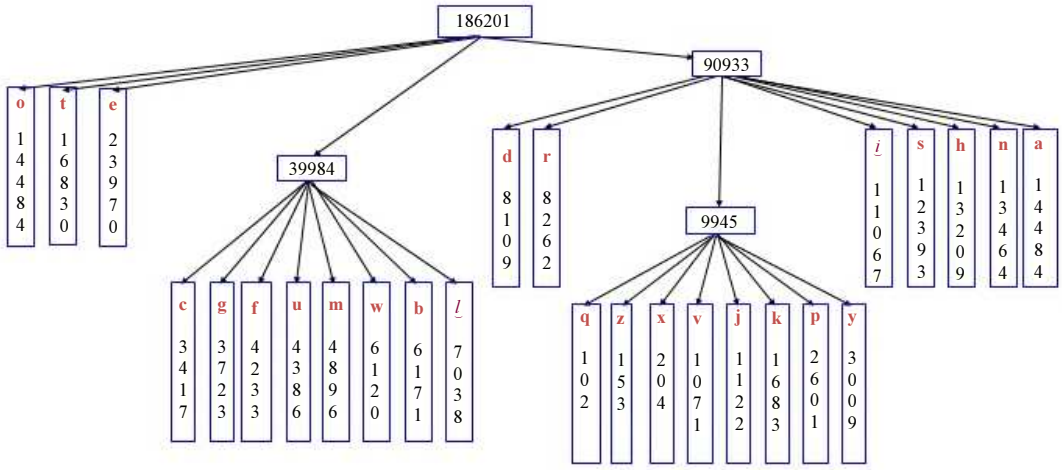

Fig. 4: Octanary tree

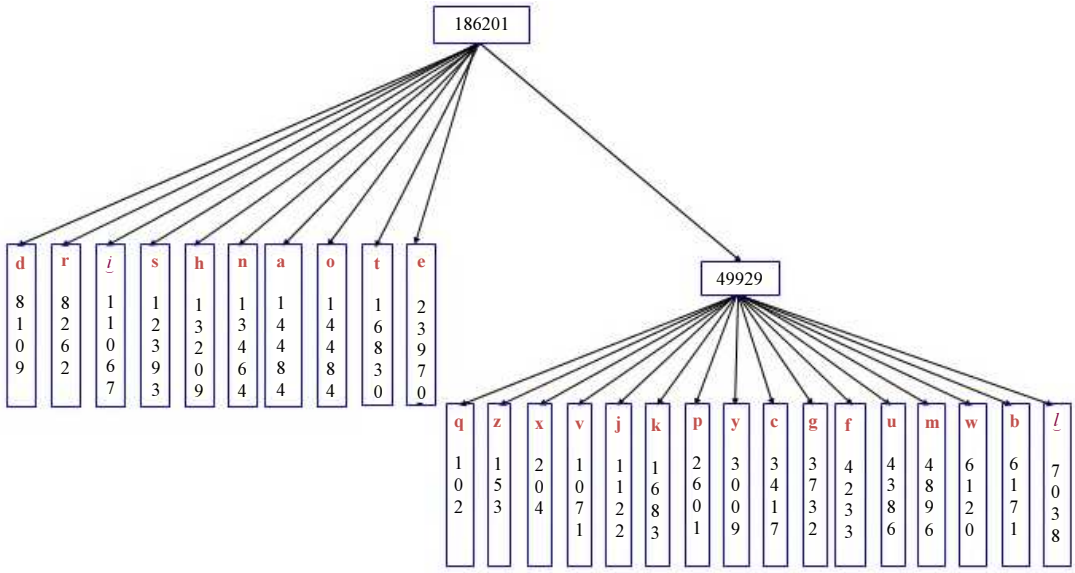

Fig. 5: Hexanary tree

Table 1: Comparison of different tree structures

Parameter	Binary	Quaternary	Octanry	Hexanary
Level of tree	10	5	3	2
Number of internal node	25	9	4	4
Total number of nodes	51	35	30	28
Weighted path length	784023	497301	327063	236130

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 Octanry 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 *1* 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 Octanry Huffman Tree

```
      O- HUFFMAN (C)
1.      Q ← C
2.      n ← |Q|
3.      i ← n
4.      WHILE  i > 1
5.        allocate a new node z
6.        left1[z] ← r ← EXTRACT-MIN(Q)
7.        left2[z] ← s← EXTRACT-MIN(Q)
8.        IF i = 2
9.          f [z] ← f[r] + f[s]
10.       ELSE IF i =3
11.         left3 [z] ←  t ← EXTRACT-MIN(Q)
12.         f [z] ← f[r] + f[s] + f[t]
13.       ELSE IF i =4
14.         left3 [z] ←  t ← EXTRACT-MIN(Q)
15.         left4 [z] ←  u ← EXTRACT-MIN(Q)
16.         f [z] ← f[r] + f[s] + f[t] + f[u]
17.       ELSE IF i =5
18.         left3 [z] ←  t ← EXTRACT-MIN(Q)
19.         left4 [z] ←  u ← EXTRACT-MIN(Q)
20.         right1[z] ←  v ← EXTRACT-MIN(Q)
21.         f [z] ← f[r] + f[s] + f[t] + f[u] + f[v]
22.       ELSE IF i =6
23.         left3 [z] ←  t ← EXTRACT-MIN(Q)
24.         left4 [z] ←  u ← EXTRACT-MIN(Q)
25.         right1[z] ←  v ← EXTRACT-MIN(Q)
26.         right2[z] ←  w ← EXTRACT-MIN(Q)
27.         f [z] ← f[r] + f[s] + f[t] + f[u] + f[v] + f[w]
28.       ELSE IF i =7
29.         left3 [z] ←  t ← EXTRACT-MIN(Q)
30.         left4 [z] ←  u ← EXTRACT-MIN(Q)
31.         right1[z] ←  v ← EXTRACT-MIN(Q)
32.         right2[z] ←  w ← EXTRACT-MIN(Q)
33.         right3[z] ←  x ← EXTRACT-MIN(Q)
34.       f [z] ← f[r] + f[s] + f[t] + f[u] + f[v] + f[w] + f[x]
35.       ELSE
36.         left3 [z] ←  t ← EXTRACT-MIN(Q)
37.         left4 [z] ←  u ← EXTRACT-MIN(Q)
38.         right1[z] ←  v ← EXTRACT-MIN(Q)
39.         right2[z] ←  w ← EXTRACT-MIN(Q)
40.         right3[z] ←  x ← EXTRACT-MIN(Q)
41.         right4[z] ←  y ← EXTRACT-MIN(Q)
42.       f [z] ← f[r] + f[s] + f[t] + f[u] + f[v] + f[w] + f[x] + f[y]
43.       END IF
44.       INSERT(Q, z)
45.       i ← |Q|
46.   END WHILE
47.   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*, *LEFT3*, *LEFT4*, *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*, *LEFT3*, *LEFT4*, *RIGHT1*, *RIGHT2*, *RIGHT3*, *RIGHT4* 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 *1* 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 *121*, 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* 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* 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* 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* 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*, *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 *16* we extract sixteen 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 queue, *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

```
      H- HUFFMAN (C)
1.    Q ← C
2.    n ← |Q|
3.    i ← n
4.    WHILE  I > 1
5.      allocate a new node z
6.      left1[z] ← j ← EXTRACT-MIN(Q)
7.      left2[z] ← k← EXTRACT-MIN(Q)
8.      IF i = 2
9.        f [z] ← f[j] + f[k]
10.     ELSE IF i =3
11.       left3 [z] ←  l ← EXTRACT-MIN(Q)
12.       f [z] ← f[j] + f[k] + f[l]
13.     ELSE IF i =4
14.       left3 [z] ←  l ← EXTRACT-MIN(Q)
15.       left4 [z] ←  m ← EXTRACT-MIN(Q)
16.       f [z] ← f[j] + f[k] + f[l] + f[m]
17.     ELSE IF i =5
18.       left3 [z] ←  l ← EXTRACT-MIN(Q)
19.       left4 [z] ←  m ← EXTRACT-MIN(Q)
20.       left5[z] ←  n ← EXTRACT-MIN(Q)
21.       f [z] ← f[j] + f[k] + f[l] + f[m] + f[n]
22.     ELSE IF i =6
23.       left3 [z] ←  l ← EXTRACT-MIN(Q)
24.       left4 [z] ←  m ← EXTRACT-MIN(Q)
25.       left5[z] ←  n ← EXTRACT-MIN(Q)
26.       left6[z] ←  o ← EXTRACT-MIN(Q)
27.       f [z] ← f[j] + f[k] + f[l] + f[m] + f[n] + f[o]
28.     .
104.    .
105.    ELSE
106.      left3 [z] ←  j ← EXTRACT-MIN(Q)
107.      left4 [z] ←  k ← EXTRACT-MIN(Q)
108.      left5 [z] ←  l ← EXTRACT-MIN(Q)
109.      left6 [z] ←  m ← EXTRACT-MIN(Q)
110.      left7 [z] ←  n ← EXTRACT-MIN(Q)
111.      left8 [z] ←  o ← EXTRACT-MIN(Q)
112.      right1 [z] ←  p ← EXTRACT-MIN(Q)
113.      right2 [z] ←  q ← EXTRACT-MIN(Q)
114.      right3[z] ←  r ← EXTRACT-MIN(Q)
115.      right4[z] ←  s ← EXTRACT-MIN(Q)
116.      right5[z] ←  t ← EXTRACT-MIN(Q)
117.      right6[z] ←  u ← EXTRACT-MIN(Q)
118.      right7[z] ←  v ← EXTRACT-MIN(Q)
119.      right8[z] ←  w ← EXTRACT-MIN(Q)
120.      f [z] ← f[j] + f[k] + f[l] + f[m] + f[n] + f[o] + f[p] +..+ f[y]
121.    END IF
122.    INSERT(Q, z)
123.    i ← |Q|
124.  END WHILE
125.  RETURN 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 (*000* bit means go *LEFT1*, *001* bit means go *LEFT2*, *010* bit means go *LEFT3,* etc in case of the Octanary tree; *0000* bit means go *LEFT1*, *0001* bit means go *LEFT2*, *0010* 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 Octanry Huffman Tree

Algorithm 3. Decoding of Octanary Huffman Tree

```
OH-DECODE (T, B)
1.      ln ← T
2.      n ← |B|
3.      i ← 0
4.      WHILE   i ← n
5.        b1 ← EXTRACT-BIT(B)
6.        b2 ← EXTRACT-BIT(B)
```

```
7.          b3 ← EXTRACT-BIT(B)
8.          IF  b1 = 0 AND b2 = 0 AND  b3=0
9.            ln ← LEFT1 (ln)
10.         ELSE b1 = 0 AND b2 = 0 AND  b3=1
11.           ln ← LEFT2 (ln)
12.         ELSE b1 = 0 AND b2 = 1 AND  b3=0
13.           ln ← LEFT3 (ln)
14.         ELSE b1 = 0 AND b2 = 1 AND  b3=1
15.           ln ← LEFT4 (ln)
16.         ELSE b1 = 1 AND b2 = 0 AND  b3=0
17.           ln ← RIGHT1 (ln)
18.         ELSE b1 = 1 AND b2 = 0 AND  b3=1
19.           ln ← RIGHT2 (ln)
20.         ELSE b1 = 1 AND b2 = 1 AND  b3=0
21.           ln ← RIGHT3 (ln)
22.         ELSE
23.           ln ← RIGHT4 (ln)
24.         END IF
25.           k ← KEY(ln)
26.         IF ← k IS NOT NULL
27.           Output (k)
28.           ln ← T
29.         END IF
30.         i ← i + 3
31.   END WHILE
```

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 an Octanary tree, we have at most eight leaves for a parent node: *LEFT1, LEFT2, LEFT3, LEFT4, RIGHT1, RIGHT2, RIGHT3, RIGHT4* and *000*, *001*, *010*, *011*, *100*, *101*, *110*, *111* represent these leaf nodes, respectively. So, we take three 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* and *7*, local variable *b1*, *b2 and b3* are being assigned with three 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 *000* we take the *LEFT1* child of the parent *ln* and assign it to *ln* itself. For *001*, we replace the parent *ln* with its *LEFT2* child, for *010* we replace it with its *LEFT3* child, for *011* we replace it with the *LEFT4* child, for *100* we replace it with its *RIGHT1* child, for *101* we replace it with its *RIGHT2* child, for *110* we replace it with its *RIGHT3* child and for *111* we replace it with its *RIGHT4* child. In line *25*, 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 *30* we increase the value of *i* by *3* and the loops get continued and read the next three bits.

Search time for finding the source symbol Octanary Huffman Tree is O($log_8 n$) whereas for Huffman based techniques decoding algorithm it is O($log_2 n$).

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, LEFT8, 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 *9* to line *41*, 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

```
      HH-DECODE (T, B)
1.      ln ← T
2.      n ← |B|
3.      i ← 0
4.      WHILE   i ← n
5.            b1 ← EXTRACT-BIT(B)
6.            b2 ← EXTRACT-BIT(B)
7.            b3 ← EXTRACT-BIT(B)
8.            b4 ← EXTRACT-BIT(B)
9.            IF  b1 = 0 AND b2 = 0 AND  b3=0 AND b4=0
10.             ln ← LEFT1 (ln)
11.           ELSE b1 = 0 AND b2 = 0 AND  b3=0 AND b4=1
12.             ln ← LEFT2 (ln)
13.       .
23.       .
25.       IF  b1 = 1 AND b2 = 0 AND  b3=0 AND b4=0
26.         ln ← RIGHT1 (ln)
27.       ELSE b1 = 1 AND b2 = 0 AND  b3=0 AND b4=1
28.         ln ← RIGHT2 (ln)
29.       .
37.       .
39.       ELSE
40.         ln ← RIGHT8 (ln)
41.       END IF
42.       k ← KEY(ln)
```

```
43.          IF ← k IS NOT NULL
44.            Output (k)
45.            ln ← T
46.          END IF
47.          i ← i + 4
48.    END WHILE
```

child, for *1011* we replace it with its *RIGHT4* child, for *1100* we replace it with its *RIGHT5* child, for *1101* we replace it with its *RIGHT6* child, for *1110* we replace it with its *RIGHT7* child and for *1111* we replace it with its *RIGHT8* child. 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($log_{16} n$), 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.

Sym-bol	Frequ-ency	Bin-ary	Quatern-ary	Octan-ary	Hexan-ary
q	102	1000001010	0110000100	100010000	10100000
z	153	1000001011	0110000101	100010001	10100001
x	204	100000100	0110000110	100010010	10100010
v	1071	10000011	0110000111	100010011	10100011
j	1122	1000000	01100000	100010100	10100100
k	1683	000110	01100010	100010101	10100101
p	2601	000111	01100011	100010110	10100110
y	3009	100001	000000	100010111	10100111
c	3417	111100	000001	011000	10101000
g	3723	111101	000010	011001	10101001
f	4233	00010	000011	011010	10101010
u	4386	01000	010000	011011	10101011
m	4896	01001	010001	011100	10101100
w	6120	10001	010010	011101	10101101
b	6171	10100	010011	011110	10101110
l	7038	10101	011001	011111	10101111
d	8109	11111	011010	100000	0000
r	8262	0000	011011	100001	0001
i	11067	0101	011100	100011	0010
s	12393	1001	011101	100100	0011
h	13209	1011	011110	100101	0100
n	13464	1100	011111	100110	0101
a	14484	1101	0001	100111	0110
o	14484	1110	0010	000	0111
t	16830	001	0011	001	1000
e	23970	011	0101	010	1001

Fig. 6: Codeword generated by different algorithms

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 amunt of primary memory is 4 GiB DDR4 type. We exeocute 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.

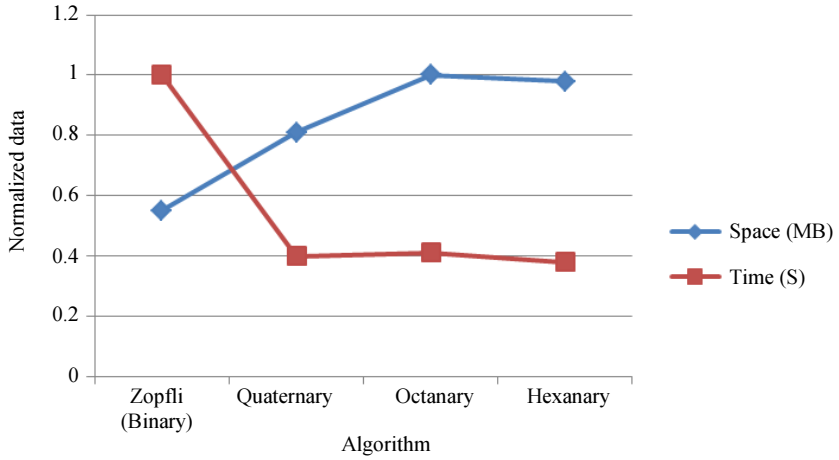

Fig. 7: Time-space requirement for Enwik corpus

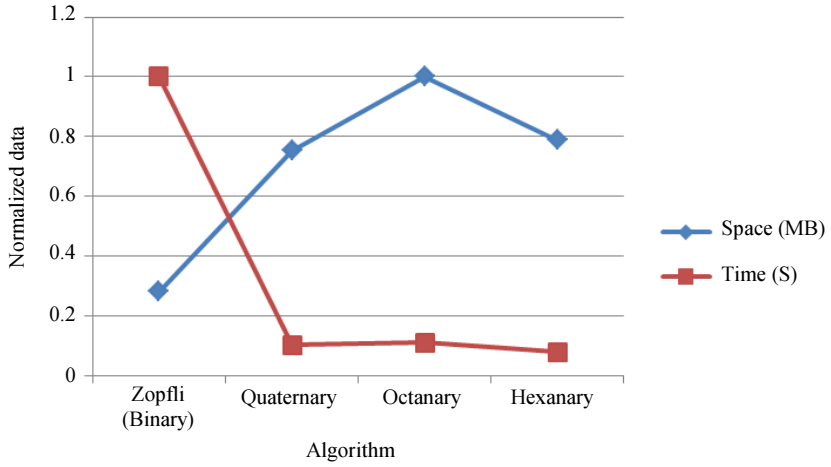

Fig. 8: Normalized Time-Space requirement for Canturbury corpus

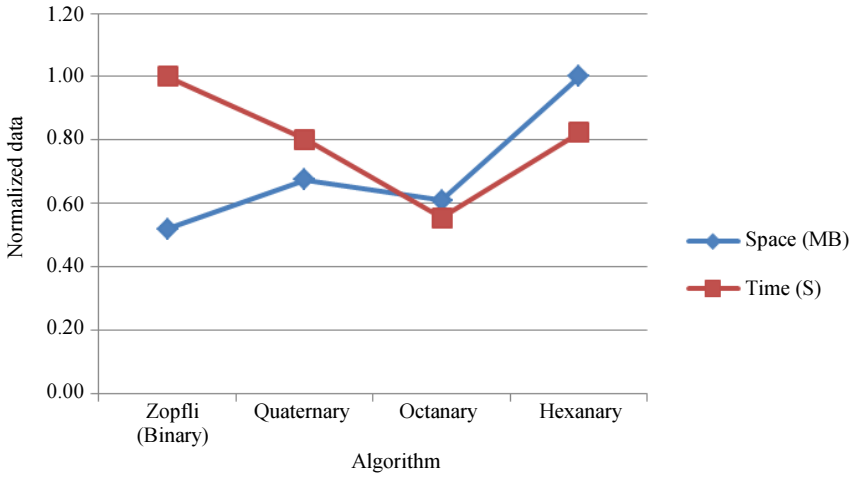

Fig. 9: Octanary performance for "Consultation-en"

Table 2: Data set

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

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

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

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

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

Table 5: Time-space data for Enwik corpus

Before normalization			After normalization		
Algorithm	Space (MB)	Time (S)	Algorithm	Space (MB)	Time (S)
Zopfli (Binary)	33.37	463.26	Zopfli (Binary)	0.55	1.00
Quaternary	49.67	186.88	Quaternary	0.81	0.40
Octanary	61.06	187.82	Octanary	1.00	0.41
Hexanary	59.73	174.58	Hexanary	0.98	0.38

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 approximatly 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-1)-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

Adamchik, V.S., 2009. Binary trees. https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html

Alakuijala, J., E. Kliuchnikov, Z. Szabadka and L. Vandevenne, 2016. Comparison of brotli, deflate, zopfli, LZMA, LZHAM and BZip2 compression algorithms. Internet Engineering Task Force.

Alakuijala, J. and L. Vandevenne, 2013. Data compression using zopfli. Google Inc.

Bell, T. and M. Powel, 2000. The Canterbury corpus. http://corpus.canterbury.ac.nz/resources/cantrbry.zip

Chen, H.C., Y.L. Wang and Y.F. Lan, 1999. A memory-efficient and fast Huffman decoding algorithm. Inform. Process. Lett., 69: 119-122. DOI: 10.1016/S0020-0190(99)00002-2

Chung, K.L., 1997. Efficient Huffman decoding. Inform. Process. Lett., 61: 97-99. DOI: 10.1016/S0020-0190(96)00204-9

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein, 1989. Introduction to Algorithms. 2nd Edn., The MIT Press, ISBN-10: 0-262-03293-7.

EC, 2013. Public consultation on the review of the EU copyright rules. European Commission.

Habib, A. and M.S. Rahman, 2017. Balancing decoding speed and memory usage for huffman codes using quaternary tree. Applied Informat., 4: 1-15. DOI: 10.1186/s40535-016-0032-z

Hashemian, R., 1995. Memory efficient and high-speed search Huffman coding. IEEE Trans. Comm., 43: 2576-2581.

Huffman, D.A., 1952. A method for the construction of minimum-redundancy codes. Proc. IRE, 40: 1090-1101. DOI: 10.1109/JRPROC.1952.273898

Katona, G.O.H. and T.O.H. Nemetz, 1978. Huffman codes and self-information. IEEE Trans. Inform. Theory, 22: 337-340. DOI: 10.1109/TIT.1976.1055554

Lin, Y.K., S.C. Huang and C.H. Yang, 2012. A fast algorithm for Huffman decoding based on a recursion Huffman tree. J. Syst. Software, 85: 974-980. DOI: 10.1016/j.jss.2011.11.1019

Luke 5, 2018. Wikipedia. https://en.wikipedia.org/wiki/Luke_5

Mahoney, M., 2018. The Enwik8 corpus. http://mattmahoney.net/dc/text.html http://mattmahoney.net/dc/enwik8.zip

Matai, J., J.Y. Kim and R. Kastner, 2014. Energy efficient canonical Huffman encoding. Proceedings of the IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, Jun. 8-20, IEEE Xplore Press, Zurich, Switzerland, pp: 202-209. DOI: 10.1109/ASAP.2014.6868663

Oswald, C. and B. Sivaselvan, 2018. An optimal text compression algorithm based on frequent pattern mining. J. Ambient Intell. Human Comput., 9: 803-822. DOI: 10.1007/s12652-017-0540-2

Oswald, C., A.I. Ghosh and B. Sivaselvan, 2015. An efficient text compression algorithm-data mining perspective. Proceedings of the 3rd International Conference on Mining Intelligence and Knowledge Exploration, Dec. 09-11, Springer, Hyderabad, India, pp: 563-575. DOI: 10.1007/978-3-319-26832-3_53

Radhakrishnan, J., S. Sarayu, K.G. Kurain, D. Alluri, and R. Gandhiraj, 2016. Huffman coding and decoding using Android. Proceedings of the International Conference on Communication and Signal Processing, Apr. 6-8, IEEE Xplore Press, Melmaruvathur, India, pp: 0361-0365. DOI: 10.1109/ICCSP.2016.7754156

Rajput, K.K., 2018. Are Huffman trees balanced? https://www.quora.com/Are-Huffman-trees-balanced

Renugadevi, S. and P.S.N. Darisini, 2013. Huffman and Lempel-Ziv based data compression algorithms for wireless sensor networks. Proceedings of the International Conference on Pattern Recognition, Informatics and Mobile Engineering, Feb. 21-22, IEEE Xplore Press, Salem, India, pp: 461-463. DOI: 10.1109/ICPRIME.2013.6496521

Săcăleanu, S.I., R. Stoian, D. M. Ofrim, 2011. An adaptive Huffman algorithm for data compression in wireless sensor networks. Proceedings of the International Symposium on Signals, Circuits and Systems, Jun. 30-Jul. 1, IEEE Xplore Press, Lasi, Romania, pp: 1-4. DOI: 10.1109/ISSCS.2011.5978764

Saradashri, S., A. Arelakis, P. Stenstrom and D.A. Wood, 2015. A Primer on Compression in the Memory Hierarchy. 1st Edn., Morgan and Claypool Publishers, ISBN-10: 1627057048, pp: 86.

Sinaga, A., Adiwijaya and H. Nugroho, 2015. Development of word-based text compression algorithm for indonesian language document. Proceedings of the 3rd International Conference on Information and Communication Technology, May 27-29, IEEE Xplore Press, Nusa Dua, Bali, pp: 450-454. DOI: 10.1109/ICoICT.2015.7231466

Vitter, J.S., 1987. Design and analysis of dynamic Huffman code. J. ACM, 34: 825-845. DOI: 10.1145/31846.42227

Wang, W.J. and C.H. Lin, 2016. Code compression for embedded systems using separated dictionaries. IEEE Trans. Very Large Scale Integrat. Syst., 24: 266-275. DOI: 10.1109/TVLSI.2015.2394364