Journal of Computer Science

Original Research Paper

Quaternary Tree Structure as a Novel Method for Huffman Coding Tree

[1]Ahsan Habib, [2]Mohammed Jahirul Islam and [2]Mohammad Shahidur Rahman

[1]Institute of Information and Communication Technology, Shahjalal University of Science and Technology, Sylhet, Bangladesh
[2]Department of Computer Science and Engineering, Shahjalal University of Science and Technology, Sylhet, Bangladesh

Article history
Received: 09-03-2023
Revised: 25-05-2023
Accepted: 29-05-2023

Corresponding Author:
Ahsan Habib
Institute of Information and Communication Technology, Shahjalal University of Science and Technology, Sylhet, Bangladesh
Email: ahabib-iict@sust.edu

Abstract: The binary code used by the current Huffman-based techniques slows down decoding speed. The quaternary Huffman coding method is a fresh take on the traditional Huffman coding that is suggested in this study. Each symbol is encoded into a quaternary code rather than a binary code using the quaternary Huffman coding technique. For Huffman coding, a quaternary code stream necessitates a shallower Huffman tree. Less traverse time could be a benefit of a shorter Huffman tree, with the enhancement of both encoding and decoding speed. The main objectives of this research are to verify the feasibility of using quaternary code in terms of both decoding speed and memory usage mathematically, to develop a new code generation technique called quaternary Huffman coding, and to analyze the performance of the proposed system in terms of both storage and decoding time. In this study, we first quantitatively investigate the quaternary tree structure's features for building Huffman codes. We thoroughly investigate the new tree structure's nomenclature and substantiate the findings. It is found in the research that the quaternary code performs better than binary code in terms of decoding speed in data compression and decompression techniques.

Keywords: Binary Tree, Quaternary Tree, Tree Structure, Huffman Coding, Data Compression, Lossless Compression

Introduction

When it comes to statistical and bit-level text compression techniques, the Huffman coding technique is used in various sectors of lossless compression area. After the Huffman algorithm (Huffman, 1952) was proposed, it was demonstrated that, besides text data compression, the approach was also effective in compressing images and videos (Kuo-Liang, 1997). The authors assert that both Shanon-Fano and Huffman's codeword lengths have a comparable interpretation after examining the sibling property and level of trees (Schack, 1994). Another study looks into the relationship between a symbol's self-information and the length of its codeword in a Huffman code (Katona and Nemetz, 1976). The study's shortcoming is that it can be challenging to give an individual symbol's self-information a similar level of significance.

Fenwick has shown in his study that the traditional Huffman code generation technique cannot always improve code efficiency and that performance always plummets when one moves from the lower extension to the higher (Fenwick, 1995). Huffman method was implemented using variable-to-variable code in addition to the more popular fixed-to-variable code (Kavousianos *et al*., 2008). Lin *et al*. (2012) proposed a highly intriguing method for decoding several symbols at once by converting a simple Huffman tree to a recursive one. The suggested method only worked for test data compression issues and consumed a lot of memory. His main goal was to increase a Huffman tree's effectiveness.

It has been found that The Huffman code's size has an impact on both the speed of decoding and the compression ratio. In this study, the recently used quaternary technique to create more effective and ideal codes that guarantee the fastest possible decoding is verified. The quaternary tree has four ary levels or at most four children. This study suggests using quaternary Huffman coding, a variant of traditional Huffman coding for improving the decoding speed in the tree structure. With the quaternary Huffman coding technique, each symbol is encoded into a new coding technique that is called quaternary code. The quaternary Huffman tree's properties were evaluated in a

different study by the same authors as this one and we came to the conclusion that its height is normally one-third that of a binary tree (Habib and Rahman, 2017).

Literature Survey

The property of a prefix code is that it is a prefix of the code for another symbol. The Huffman coding algorithm creates the best prefix codes, which are a family of codes with variable codeword lengths, from a collection of probabilities (Huffman, 2005). Huffman codes' prefix property guarantees that, despite their variable length, they can still be accurately decoded. David Huffman created the Huffman coding algorithm in 1950 as a student at MIT in an information theory class and is the algorithm's creator (Huffman, 1952).

A semi-adaptive (or semi-static) Huffman coding is necessary since it has to know the frequencies of each alphabetic symbol. Moreover, the compressed output must be saved together with the Huffman tree that contains the Huffman codes for symbols (or simply frequencies of symbols, which can be used to generate the Huffman tree). This data is typically included in the header of a compressed file since the decoder requires it.

When the probability of the input symbols changes, the semi-adaptive Huffman coding is ineffective. Using the Huffman algorithm to construct the tree and produce prefix codes after encoding each symbol from the input is incredibly inefficient. Faller introduced a modified version of the Huffman algorithm in 1973 (Faller, 1973), which deals with this issue. Modern terminology refers to this algorithm as an adaptive Huffman coding.

A new method of creating a Huffman tree is presented by the adaptive Huffman algorithm, which also provides the sibling attribute. This algorithm begins with either a standard distribution or an empty tree. It incorporates new symbols that are currently absent from the tree as special control symbols to the tree. After encoding each symbol from the input, the adaptive Huffman method allows modification of the Huffman tree. This enables dynamic changes in Huffman codes in response to variations in symbol probabilities. Naturally, the Huffman tree for the encoder and decoder must remain the same.

In most cases, the adaptive Huffman algorithm generates more efficient code than the semi-adaptive Huffman method. Moreover, the compressed output eliminates the requirement to keep the Huffman tree along with the Huffman codes for symbols. Nevertheless, the adaptive Huffman technique is slower than the semi-adaptive variant since it needs to update the Huffman tree after encoding each symbol. Moreover, the effectiveness of compression is poor early in the coding process or for little files.

Because different symbols have different probabilities of occurring, the semi-adaptive and adaptive Huffman coding reduces redundancy in a dataset. Shorter codes are used for symbols with higher likelihoods of occurrence, whereas longer codes are used for symbols with lower likelihoods. However, in actual applications, arithmetic coding, which is covered in the next paragraph, frequently takes the place of adaptive Huffman coding because it is simpler and more efficient.

May sometimes be the Huffman coding employed alone as a compression technique. It is typically applied as the final coding technique in lossless data compression. In conjunction with, for instance, the LZSS method (used in gzip, PKZip, ARJ, and LHArc), BWT-based algorithms, JPEG (Wallace, 1991), and MPEG compression, Run-Length Encoding and Move-To-Front coding (Skibinski, 2006), Huffman compression is utilized. According to trial analysis and results of another research based on manual mathematical and statistical calculations, 4-ary/MQ delivers high compression results with a very quick procedure, therefore using it to reduce data on local storage media, hosting/cloud, and bandwidth has numerous advantages (Hidayat *et al*., 2022). In another research, a compression method combines the information-rearranging Lempel-Ziv-Welch (LZW) algorithm and the Huffman encoding method, both of which are lossless compression techniques. Hoffman and LZW approaches are combined in this way that a binary information arrangement is employed, resulting in an information mapping that is fully unique for each piece of information while yet simplifying the mapping, making this article stand out (Mohammadi *et al*., 2022).

The discussion above has shown the variety of algorithms and coding schemes used in compression techniques. Because we used these models as the foundation for many architectures, we have covered the Huffman architecture in great detail. It has been noted that the Huffman code's length affects both the speed of decoding and the compression ratio. The quaternary tree technique produces a code word from a less tall tree, making it more optimum. As a result, the code could be written differently.

Materials and Methods

When storing and retrieving data from memory, the tree data structure is very important. There are numerous methods to structure a tree. The binary tree structure is currently mostly used during strong and retrieving data. The tree's structure determines how much memory is needed to store data and how long it takes to decode them before accessing them. Some mathematical metrics for measuring performance include the weighted path length, the height of the tree, and the number of internal nodes that are studied in this research. In this research, we also go through how quaternary trees can be used in place of binary trees to speed up Huffman code decoding.

Variable-length binary Huffman coding typically makes it challenging to strike a compromise between speed

and memory use. Here, a quaternary tree is employed to generate an ideal code word that accelerates the search process. The specifics of a few additional tree architectures that could provide the code word for data compression are also covered in this section. This section describes the structure and algorithm of binary and quaternary trees.

If each internal vertex of a rooted tree has exactly m children, the rooted tree is said to be an m-ary tree. If every internal vertex of the tree has exactly m offspring, the tree is said to be a full m-ary tree. A binary tree is an m-ary tree when m = 2. When the children of each internal vertex are ordered, the rooted tree is said to be ordered. The offspring of each internal vertex are displayed in order from left to right on ordered rooted trees. If an internal vertex of an ordered binary tree (often referred to as merely a binary tree) has two children, the first child is referred to as the left child and the second child as the right child. The left subtree (or right subtree, resp.) of a vertex is the tree that is rooted at the left child (or right child, resp.) of this vertex.

The variation of using a quaternary tree instead of a binary tree to produce the code for using different compression algorithms is the main purpose of this research. Although both ternary and quaternary trees will have the same level of storage complexity, ternary trees will have more internal nodes and be taller. Due to this, ternary trees take longer to traverse than quaternary trees.

In the current work, modified Huffman coding is used to produce dictionary code-word for data compression purposes. The traditional Huffman tree has at best two subtrees and produces a single bit for a single level traversing whereas the quaternary tree has at best four subtrees and it requires at least two bit for a single level of traversing. Figures 3-4 show how binary and quaternary codes are generated respectively.

A tree *T* is a connected undirected acyclic graph. It has vertices $V = \{v_0, v_1, ..., v_{n-1}\}$ and a set of edges $E = \{e_0, e_2, ..., e_{n-1}\}$. If *u* is *v* 's parent, *v* is referred to as *u* 's child. Siblings are children having the same parent. A vertex of a tree is referred to as a leaf if it has no offspring. There are always one or more children of an internal vertex. The distinctive node *R*, often known as the root of *T*, is found in the tree *T*. A tree *T* is referred to as a binary tree if each vertex has two offspring or fewer. If a tree *T* has at best four children with the names LEFT, LEFT-MID, RIGHT-MID, and RIGHT, it is referred to as a quaternary tree. A tree *T* is referred to as a full quaternary tree when each of its internal vertex's four offspring is present. The binary and quaternary tree structures are depicted in (Figs. 1-2) respectively.

A well-rooted tree in order at every internal vertex, *T* has an ordered child. In ordered rooted trees, the children of each internal vertex are exposed in the order of appearance from left to right. The optimal path between a root and a leaf has always been measured as the height of the tree's root, or *T*. (OCAML, 2014). The distance between the two leaf nodes that makes up a tree's diameter.

The time it takes for an algorithm to run depends on how long the pathways are in the tree. Assume that *T* is a tree with *n* exterior nodes and that each of them has a nonnegative weight. According to Lipschutz (2011), the weighted path lengths of the sum of each node make up the external weighted path length *P* of the tree *T*:

$$P = f_1 l_1 + f_2 l_2 + ... + f_n l_n$$

where,

f_i and L_i stand for the external node's frequency and path length N_i.

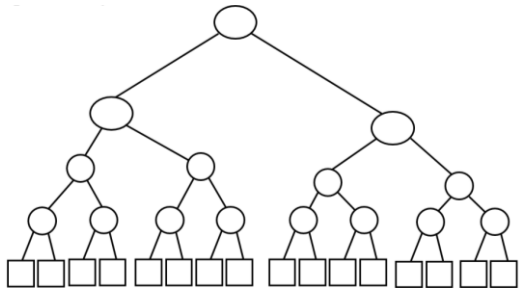

Fig. 1: Binary tree with 16 nodes

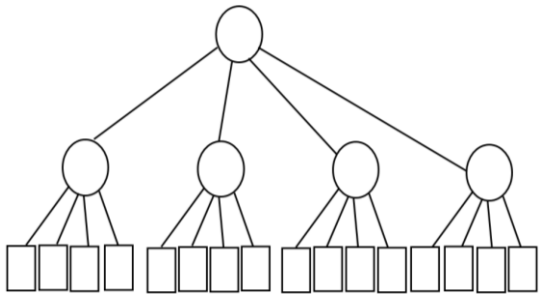

Fig. 2: Quaternary tree with 16 nodes

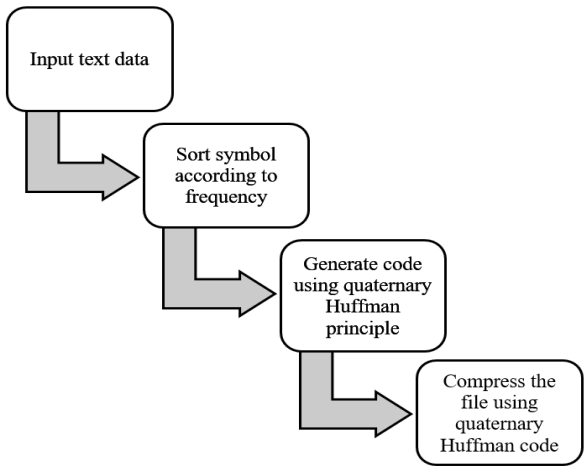

Fig. 3: Sequence of operation

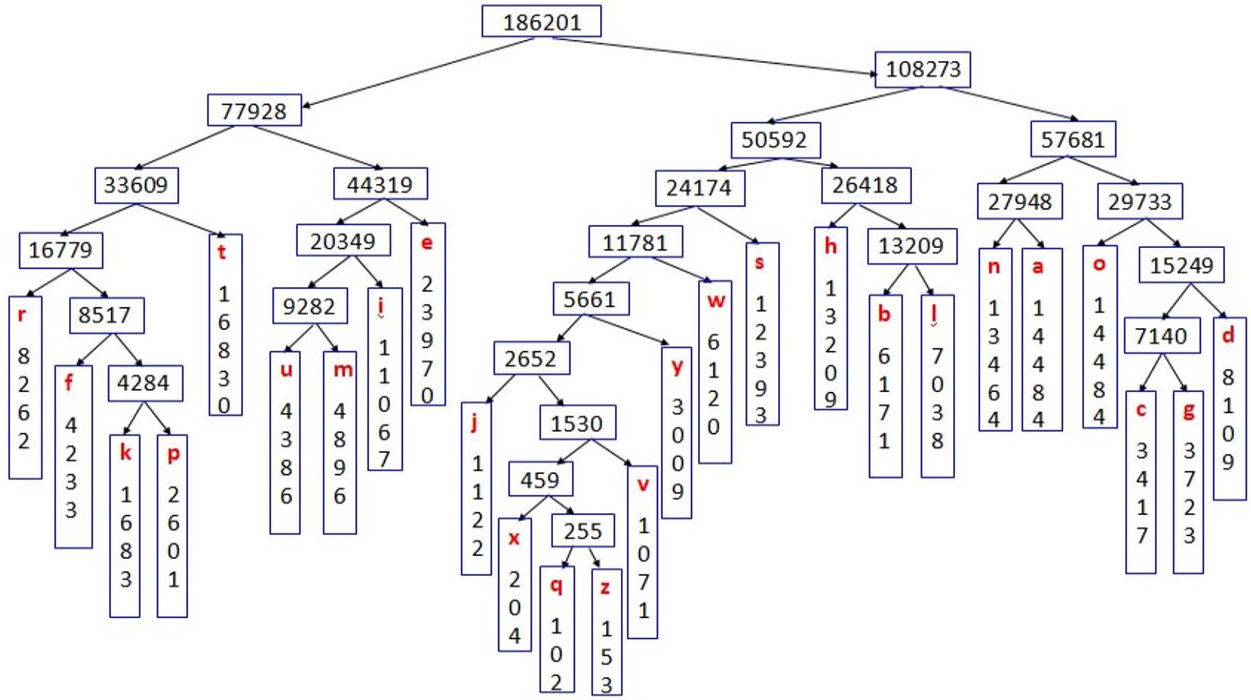

Fig. 4: Binary Huffman tree for Luke 5 data

Table 1: Comparison of different tree structures for Luke 5 data

Parameter	Binary	Quaternary
Number of levels	10	5
Number of internal nodes	25	9
Total number of nodes	51	35
Weighted path length	784023	497301

The sequence of operation is shown in Fig. 3. In this research first of all a text file is taken as input, then a frequency counter algorithm is used to count the frequency of each symbol of that input file. By using this frequency, the quaternary Huffman algorithm is used to generate the quaternary code for each symbol (Habib and Rahman, 2017). By using quaternary code, the file has been compressed using a compression algorithm (Habib *et al*., 2020).

When a dibit is searched instead of a single bit from the entire bit stream in the file, the dibit search technique performs better than a single bit.

Code Generation Technique

Verifying the applicability of binary and quaternary Huffman-based algorithms is the goal of this research. By giving up a negligible amount of space, the quaternary approach allows for faster encoding and decoding speed, proving that searching two bits simultaneously is faster than searching a single bit. The binary and quaternary Huffman-based tree structures are shown in (Figs. 4-5) respectively in this section for Luke 5 data. The fifth chapter in the New Testament of the Christian Bible is titled Luke 5. The chapter continues to discuss Jesus' teaching and healing ministry while also relating the selection of his first followers (Luke 5, 2019).

Why Choose Quaternary Tree to Generate Compression Code

The code generation techniques for both cases have been explained in detail by Habib *et al*. (2018). A binary heap is used to implement the quaternary tree. Priority queues are used with a heap data structure.

For each distinct character in the process, a leaf node is first created and then a minimum heap of all leaf nodes is built. Two nodes in the minimum heap are compared using the frequency field value. At first, the root is the least prevalent character. Second, we take the two nodes from the minimum heap with the lowest frequency. Finally, we add a new internal node whose frequency is the same as the sum of the frequencies of the previous two nodes. Create the second extracted node as the right child of the first extracted node and the other extracted node as the left child. To the minimum heap, add this node. Lastly, we repeat steps two and three until there is only one node left in the heap. The tree is finished when just the root node remains. The comparison between binary and quaternary technique for different parameters are shown in Table 1.

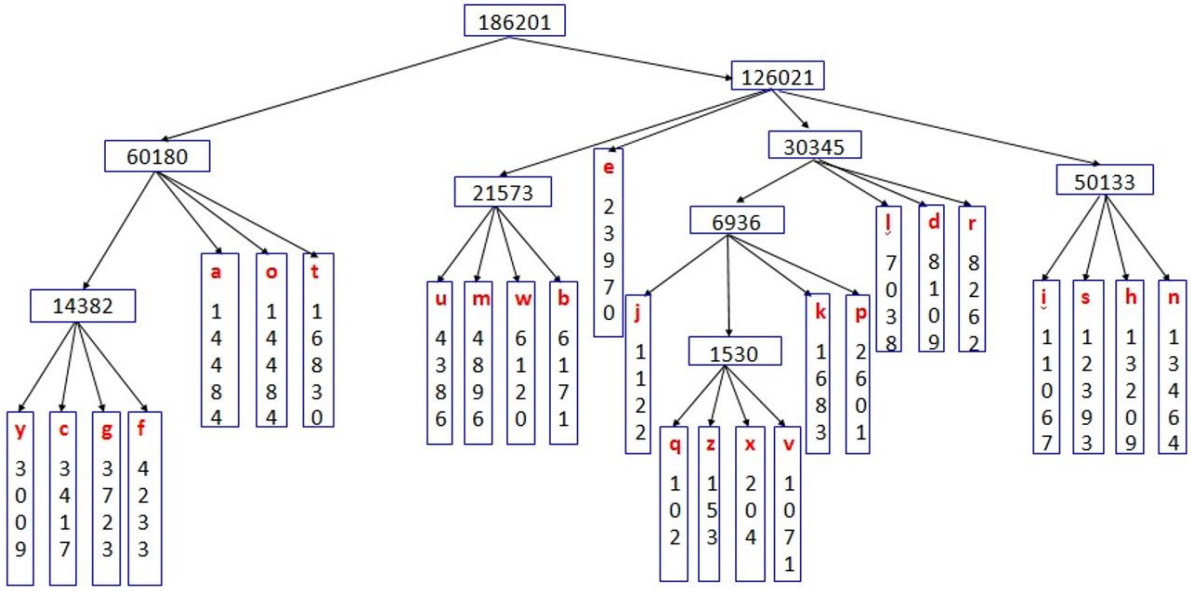

Fig. 5: Quaternary Huffman tree for Luke 5 data

Table 2: t-test result

Test	p-value	Statistical significance	Confidence interval
Paired t-test	0.0002	Extremely statistically significant	The mean of Binary height minus quaternary Height equals 2.0070; 95% confidence interval of this difference: From 1.2386 to 2.7754
Welch t-test	0.0126	Statistically significant	The mean of Binary height minus quaternary Height equals 2.0070; 95% confidence interval of this difference: From 0.5147 to 3.4993

Table 3: Analysis of binary and quaternary height data

	Tree height		
Technique	Mean	Standard deviation	Variation of coefficient
Binary	4.712	1.949	41.372
Quaternary	2.705	0.939	34.719

Table 4: Analysis of binary and quaternary code length data

	Code length		
Technique	Mean	Standard deviation	Variation of coefficient
Binary	4.712	1.949	41.372
Quaternary	5.409	1.877	34.697

Comparison Among Trees

Code Generation Algorithm

It takes specific symbols and their frequency to build a Huffman tree. The conventional Binary tree construction algorithm is described in (Cormen Thomas *et al*., 1989). The recently built Quaternary tree construction algorithm designed by the same authors of this research is shown in Algorithm 1 (Habib and Rahman, 2017). Quaternary Encoding and Decoding Methods have been extensively covered in the same research. The Quaternary Huffman Tree decoding algorithm has a search time of $0(n\log_4 n)$, whereas the classic Huffman-based approaches decoding algorithm has a search time of $(n\log_2 n)$. In Tables 2-4, it has been shown that the variation of the coefficient of binary and quaternary tree height is 41.372 and 34.719 respectively. The variation of the coefficient of binary and quaternary code length is 41.372 and 34.697 respectively, which indicates that the range of the height and code length of the quaternary tree technique is lesser than the binary tree.

Algorithm 1: Encoding of Quaternary Huffman Tree

```
        Q- HUFFMAN (C)
1. Q ← C
2. n ← |Q|
3. i ← n
WHILE i > 1
4.          allocate a new node z
5.          left[z] ← v ← EXTRACT-MIN(Q)
6.          left-mid[z] ← w ← EXTRACT-MIN(Q)
7.          IF i = 2
8.                  f [z] ← f[v] + f[w]
```

```
9.          ELSE IF i =3
10.                 right-mid [z] ← x ← EXTRACT-MIN(Q)
11.                 f [z] ← f[v] + f[w] + f[x]
12.         ELSE
13.                 right-mid [z] ← x ←
14.                 EXTRACT-MIN(Q)
15.                 right [z] ← y ← EXTRACT-
16.                 MIN(Q)
17.                 f [z] ← f[v] + f[w] + f[x] + f[y]
18.         END IF
19.         INSERT(Q, z)
20.         i ← |Q|
21. END WHILE
22. RETURN EXTRACT-MIN(Q)
```

Result

In this research, binary and quaternary Huffman-based algorithms are compared with many parameters. The Huffman principle calls for a shallower (i.e., shorter) Huffman tree to produce quaternary code streams. Less travel time from a shorter Huffman tree has the potential to increase throughput for both compression and decompression. Using the p test, which is displayed in Table 2, we tested the statistical significance between binary tree height and quaternary tree height.

Figures (6-11), it is explained how the height is statistically significant in between binary and quaternary techniques. We know that the code-word length is directly proportional to the height of the tree. In this analysis, Luke 5 data is used for generating the code word for both cases of binary and quaternary techniques. For both cases of every test, it is shown that the height of the quaternary tree is shorter than the binary tree for the same input data. This is also proved by the same authors as this in another research (Habib and Rahman, 2017).

The algorithm for decoding the compressed data is also explained in another research by the same author and also compared with so many existing techniques like bizip2, LZMA, LZHAM, and Zopfli by using some popular data sets (Habib *et al*., 2020). In the experiment, it was shown that the proposed decompression algorithm performs better than many other popular techniques. The performance matrix and time-space complexity are also explained in that research. The main purpose of this research is to verify the applicability of the code generation technique using a quaternary tree.

Analysis of quaternary and binary tree height with the increasing number of symbols with different probability distribution e.g., uniform frequency distribution, skewing frequency distribution is given in (Figs. 6-9). A comparison between tree heights and code lengths between the binary tree and quaternary tree is also shown in (Figs. 10 and 11) respectively.

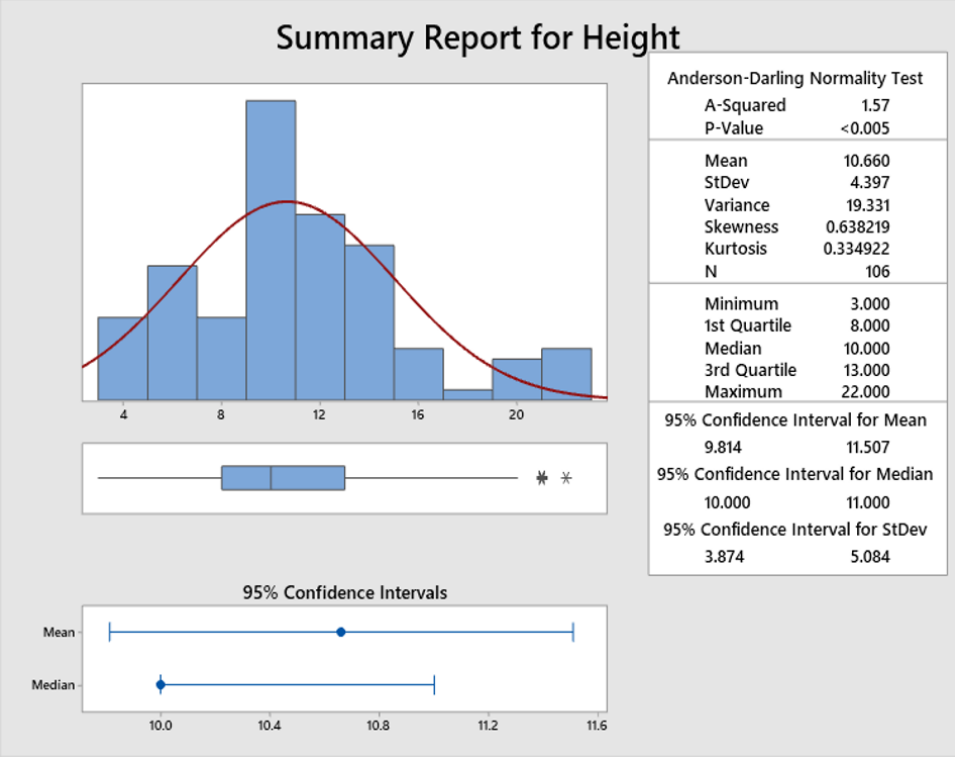

Fig. 6: Anderson-Darling normality test for binary height

DOI: 10.3844/jcssp.2023.1132.1142

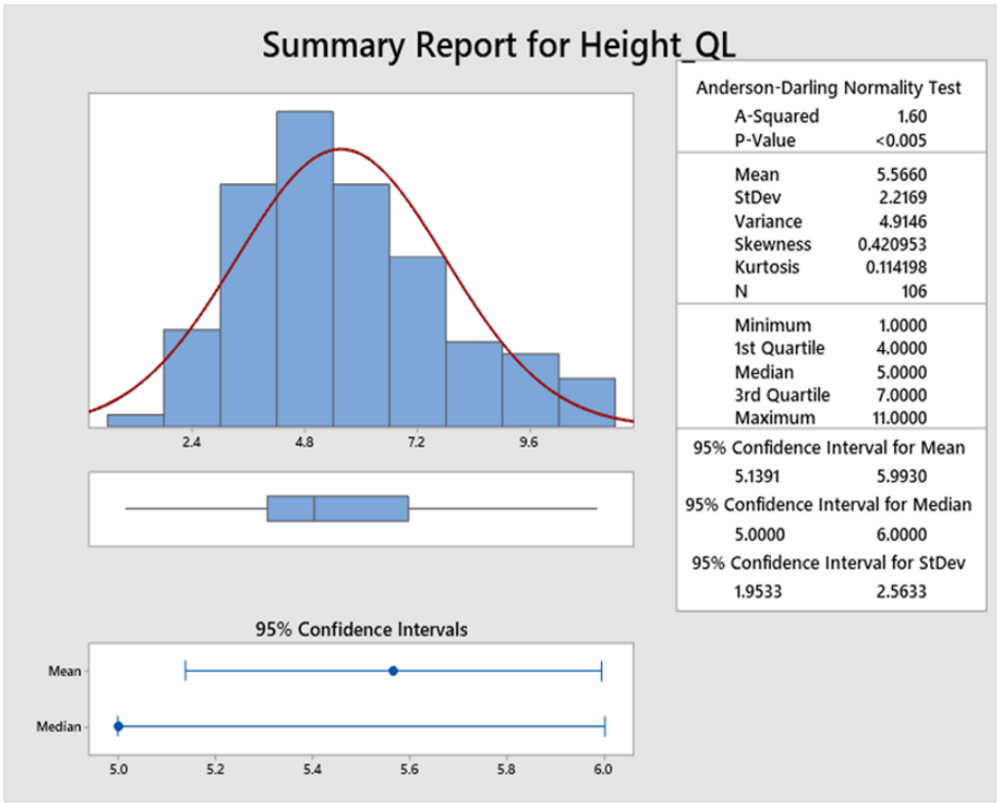

Fig. 7: Anderson-Darling normality test for quaternary height

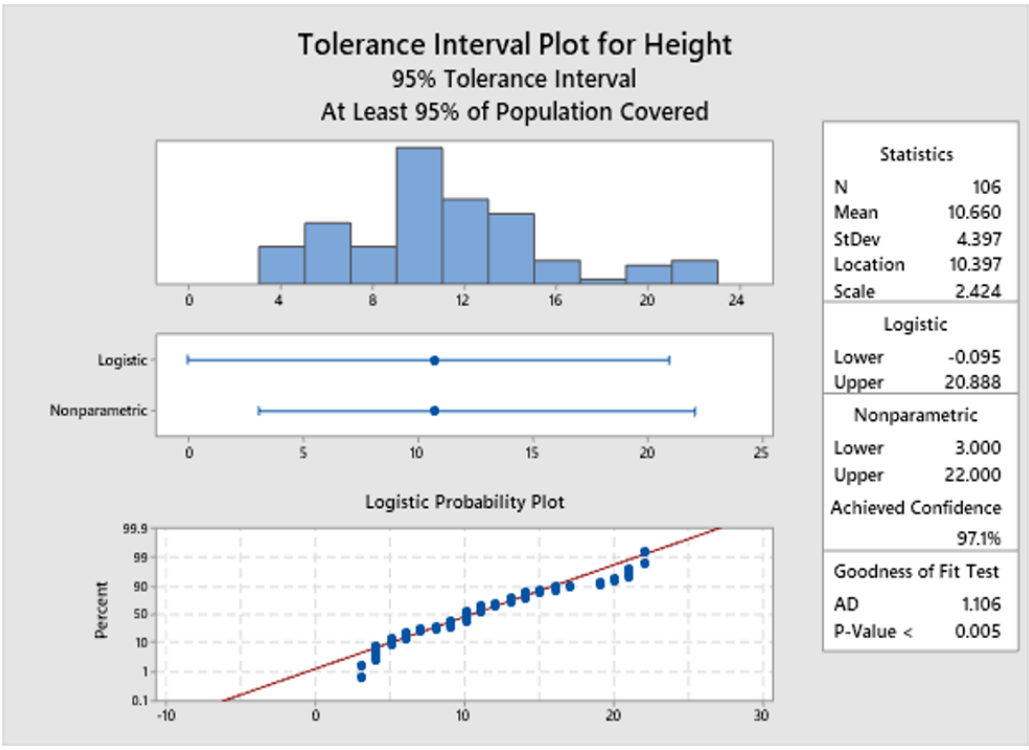

Fig. 8: Tolerance interval plot (logistic) binary height

DOI: 10.3844/jcssp.2023.1132.1142

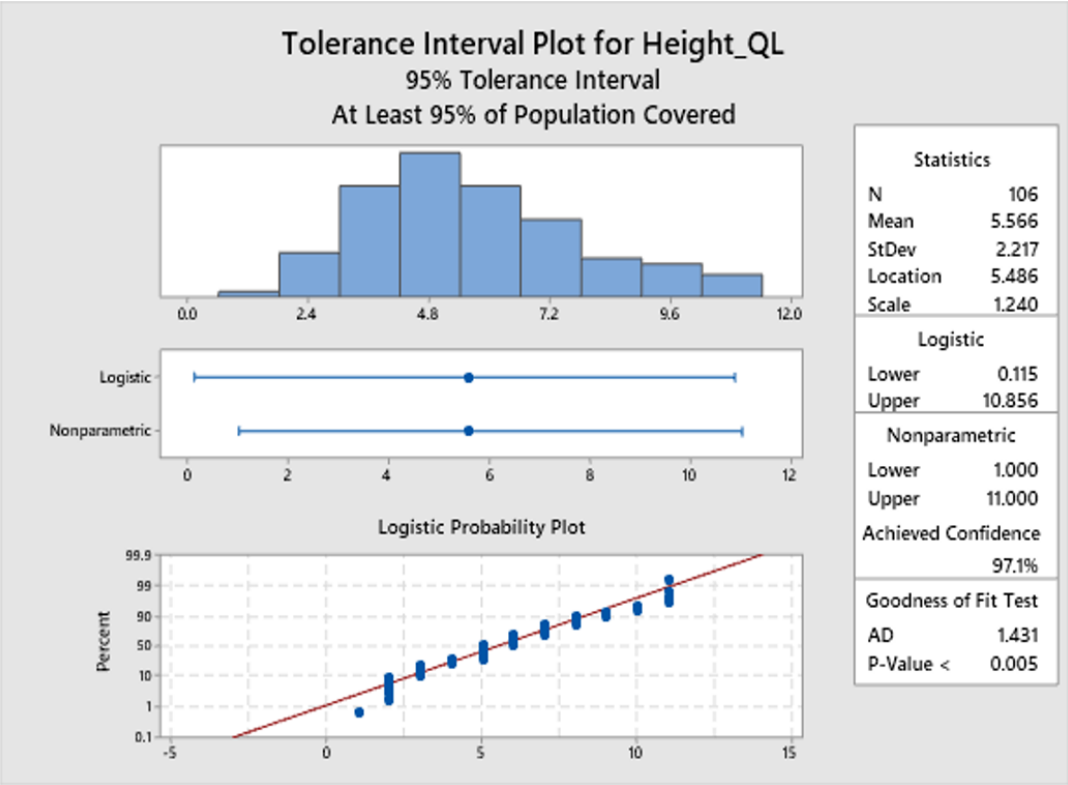

Fig. 9: Tolerance interval plot (logistic) quaternary height

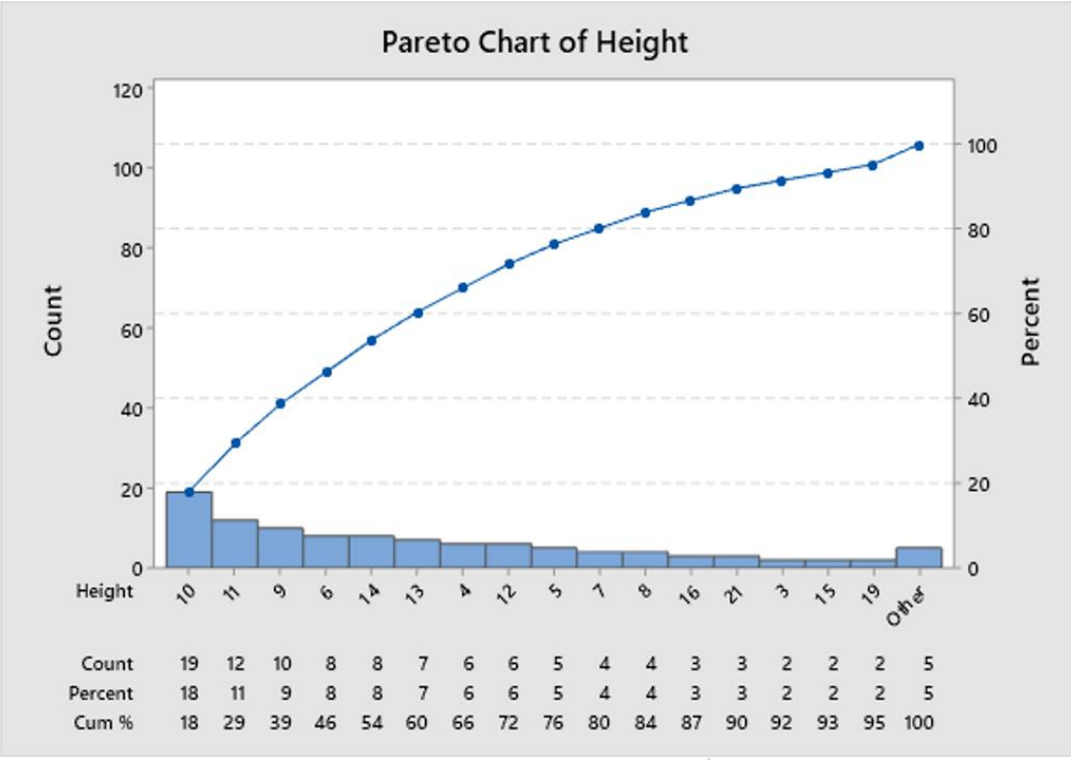

Fig. 10: Pareto chart for binary height

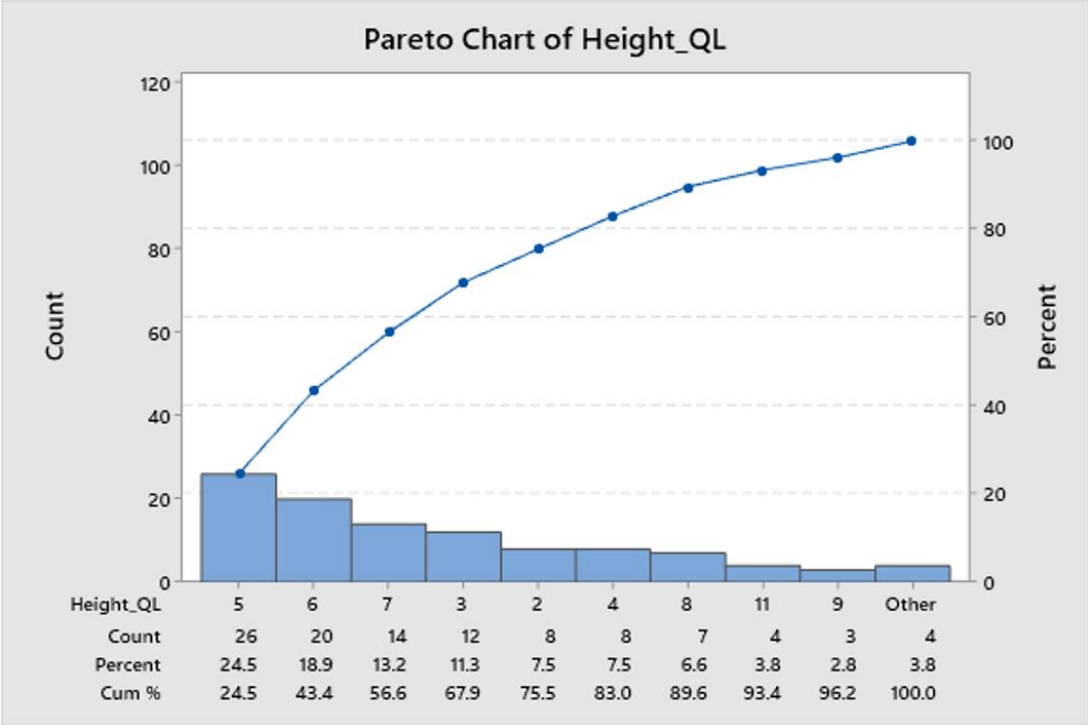

Fig. 11: Pareto chart for quaternary height

In this research, we used quaternary code instead of traditional binary code for data compression techniques. Before developing the code generation algorithm, the code-word length is verified statistically in this research. The Anderson-darling normality test is in (Figs. 6 and 7), the Tolerance interval plot is in (Figs. 8 and 9) and the Pareto chart in (Figs. 10 and 11) shows the comparison of tree height for both binary and quaternary techniques. In all cases, it is shown that the standard deviation and variation of the coefficient of the quaternary code generation technique are lower than the binary code generation technique. This means data is more clustered in the quaternary technique than in the binary technique. It indicates that the average code-word length of quaternary code will be lower than the binary code word.

We know that the code-word length is inversely proportional to the frequency that occurred in the input data. When the average code-word length will be minimum then the decoding speed will also be faster.

To build the Huffman code, the structure of the quaternary tree is examined in this section along with binary Huffman trees. The tree's height, diameter, and weighted path length are among the many criteria that are examined. It has also been demonstrated that quaternary trees are superior to binary trees for the construction of Huffman codes. In both the worst and best cases, it is discovered that the quaternary tree's traversing time is exactly half that of the binary tree shown by the same authors in their other research (Habib *et al*., 2020). In this study, quaternary Huffman coding is a variant of the traditional.

Huffman coding is used to suggest generating Huffman-based code for use in different compression algorithms. Here, the mathematical investigation begins with the characteristics of the quaternary tree structure for the creation of Huffman codes.

Discussion

The purpose of the experimental effort is to confirm that the suggested architecture is practical and applicable. Binary and quaternary Huffman-based algorithms' time-space trade-off has been extensively discussed. For increasing the compression ratio, the binary Huffman method performs better. When there needs to be a balance between time and space, the quaternary Huffman algorithm is helpful. The quaternary technique, which is illustrated in this research using a variety of graphs, is superior to other algorithms for achieving a balance between time and space.

In conclusion, the experiment shows that it is challenging to strike a compromise between speed and memory utilization when utilizing binary Huffman coding. In this study, we emphasize the usage of quaternary trees rather than binary trees since they can decode data more quickly while taking up less space. The suggested decoding algorithm has superior speed performance when compared to Huffman-based algorithms, but the

compression performance is nearly unchanged. The suggested method provides a mechanism to balance memory usage and decoding time in this fashion.

The experiment shows that the optimum method for achieving a balance between time and space is the quaternary method. We further demonstrate in the experiment that the suggested quaternary code-based compression strategies exhibit good decoding speed performance with negligible storage space increase.

Conclusion

The advantages of using a quaternary tree structure compared to a binary tree structure for producing Huffman codes are explained in this research. We have demonstrated that, with a negligible increase in necessary space, the Huffman code can be represented more quickly using a quaternary tree than it can with a binary tree. The quaternary strategy outperforms the binary technique when speed is the primary consideration. Consequently, the suggested method offers a means of striking a compromise between decoding time and memory consumption. This procedure saves time and is easy to program. In this research a new method of code generation technique for data compression is verified and its performance is compared statistically with different parameters. It is shown that the new quaternary based code generation technique outperforms in terms of decoding speed.

Funding Information

We are grateful to the research center of Shahjalal University of science and technology, Sylhet, Bangladesh for providing funds to carry out this research.

Author's Contributions

Ahsan Habib: Contributed to the original conception and algorithm design of the research work, drafted the article, and produce the figures used in the manuscript.

Mohammed Jahirul Islam: Contributed to the conception and design of the research work, reviewed the manuscript, and gave final approval of the final version of the manuscript.

Mohammad Shahidur Rahman: Contributed to 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

Cormen Thomas, H., Leiserson Charles, E., & Rivest Ronald, L. (1989). Introduction to Algorithms. The MIT Press and McGraw-Hill Book Company, ISBN: 10-262-03141-8.

Faller, N. (1973). An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems and Computers* (pp. 593-597). https://cir.nii.ac.jp/crid/1572543026347275392

Fenwick, P. M. (1995). Huffman code efficiencies for extensions of sources. *IEEE Transactions on Communications*, *43*(2/3/4), 163-165. https://doi.org/10.1109/26.380027

Habib, A., & Rahman, M. S. (2017, December). Balancing decoding speed and memory usage for Huffman codes using quaternary tree. In *Applied Informatics* (Vol. 4, No. 1, pp. 1-15). Springer Open. https://doi.org/10.1186/s40535-016-0032-z

Habib, A., Islam, M. J., & Rahman, M. S. (2018). Huffman based code generation algorithms: Data compression perspectives. *J Comput Sci*, *14*(12), 1599-1610. https://doi.org/10.3844/jcssp.2018.1599.1610

Habib, A., Islam, M. J., & Rahman, M. S. (2020). A dictionary-based text compression technique using quaternary code. *Iran Journal of Computer Science*, *3*, 127-136. https://doi.org/10.1007/s42044-019-00047-w

Hidayat, T., Zakaria, M. H., & Pee, A. N. C. (2022). Increasing the Huffman generation code algorithm to equalize compression ratio and time in lossless 16-bit data archiving. *Multimedia Tools and Applications*, 1-38. https://doi.org/10.1007/s11042-022-14130-1

Kuo-Liang, C. (1997). Efficient Huffman decoding. *Information Processing Letters*, *61*(2), 97-99. https://doi.org/10.1016/S0020-0190(96)00204-9

Mohammadi, H., Ghaderzadeh, A., & Sheikh Ahmadi, A. (2022). A novel hybrid medical data compression using huffman coding and LZW in IoT. *IETE Journal of Research*, 1-15. https://doi.org/10.1080/03772063.2022.2052985

Huffman. (2005). Code Retrieved from Wikipedia, web link. https://en.wikipedia.org/wiki/Huffman_coding

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, *40*(9), 1098-1101. https://doi.org/10.1109/JRPROC.1952.273898

Katona, G., & Nemetz, O. (1976). Huffman codes and self-information. *IEEE Transactions on Information Theory*, *22*(3), 337-340. https://doi.org/10.1109/TIT.1976.1055554

Kavousianos, X., Kalligeros, E., & Nikolos, D. (2008). Test data compression based on variable-to-variable Huffman encoding with codeword reusability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *27*(7), 1333-1338. https://doi.org/10.1109/TCAD.2008.923100

Lin, Y. K., Huang, S. C., & Yang, C. H. (2012). A fast algorithm for Huffman decoding based on a recursion Huffman tree. *Journal of Systems and Software*, *85*(4), 974-980. https://doi.org/10.1016/j.jss.2011.11.1019

Lipschutz, S. (2011). *Data Structures with C (Sie) (Sos)*. McGraw-Hill Education (India) Pvt Limited. ISBN: 10-0070701989.

Luke 5. (2019). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Luke_5

OCAML. (2014). *Height, Depth and Level of a Tree*. Retrieve d 07 09, 2019, from. http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree

Schack, R. (1994). The length of a typical Huffman codeword. *IEEE Transactions on Information Theory*, *40*(4), 1246-1247. https://doi.org/10.1109/18.335944

Skibinski, P. (2006). Reversible data transforms that improve effectiveness of universal lossless data compression. *Doctor of Philosophy Dissertation*, *University of Wroclaw*. http://pskibinski.pl/papers/PhD_thesis.pdf

Wallace, G. K. (1991). The JPEG still picture compression standard. *Communications of the ACM*, *34*(4), 30-44. https://doi.org/10.1145/103085.103089