Electronic Integrated Computer Algorithm Journal
Vol. 2, No. 1, October 2024, pp 1~8

ISSN: 3031-0350, DOI: 10.62123/enigma.v2i1.24

Comparative Analysis of Huffman Coding Implementations for Efficient Data Communication Using Greedy and Divide-and-Conquer Techniques

Kaleb Frye[1], Josh Ronhovde[1], Connor Stonestreet[1], Yousef Fazea[1*]

[1] *Department of Computer Sciences and Electrical Engineering, One John Marshall Dr.,* Huntington WV, USA 25755

DOI : 10.62123/enigma.v2i1.24

Received : May 16, 2024
Revised : August 11, 2024
Accepted : August 17, 2024

Keywords:
Data compression, Algorithm efficiency, Greedy, Divide and Conquer

ABSTRACT

Efficient data compression techniques are required to minimize storage and processing overhead due to modern systems' growing amount of data. Huffman Coding is a lossless compression technique that maintains data integrity by assigning shorter bit codes to characters appearing frequently, reducing size. Our analysis focuses on two implementation methodologies: greedy technique and divide and conquer. To find efficient solutions, divide-and-conquer algorithms partition problems into smaller components. In contrast, greedy algorithms strive to attain the utmost attainable result at each level. Our extensive investigation centers on the timing and space intricacies of diverse methodologies, enabling a comparative analysis that underscores their respective merits and drawbacks.

1. INTRODUCTION

As organizations accumulate more data, they will undoubtedly face the task of identifying effective data compression solutions [1][2]. Data compression aims to minimize data size without sacrificing accuracy. In recent years, this method has grown in significance across several domains [3][4][5]. The foundation of many deduction methods is the Huffman code [6][7][8]. To avoid altering the data storage method, lossless compression may be used to remove unused storage space [9]. To do this, codes are assigned to the letters that are entered, representing different instances. Hoffmann's code analysis is ensured to be correct by the use of techniques such as greedy programming and division and conquer [10]. The greedy algorithm aims to find the best solution for each problem by choosing the most favorable option in each subregion [11][12]. The Divide and conquer design technique separates a work into smaller portions, saving time. Each element or component is processed and finished separately, and then combined to generate the results [13][14]. This research examines the intricacy of space and time to evaluate the effectiveness of the two algorithms that have been developed. Huffman coding is often regarded as necessary for data compression [15][16][17][18]. Historically, this strategy has decreased profiles by correctly experimenting with various program parameters to enhance performance. Research indicates that Huffman coding has advantages and disadvantages [6][19][20][21]. Despite its computational intensity, particularly in tree creation, this strategy effectively decreases data [4]. The time complexity of O(n log n) is often enough, however, it may be enhanced [22]. Extensive testing of encryption and decryption methods with long paths is necessary because of the computational difficulties involved. In order to achieve efficient data compression, it is crucial to take into account the specific geographical and temporal boundaries of each technique. Our research strategy presents several challenges, including time constraints and the need for advanced techniques. An in-depth evaluation will be conducted on several software methodologies for Huffman coding. This paper proceeds as follows: Section 1 presents the introduction and related works. Section 2 presents the methodology used in this study. Sections 3 and 4 present the analysis of the time and space complexity. Section 5 presents the results and discussion and the conclusion is presented in Section 6.

2. METHODOLOGY

Many diverse companies and services have the potential to utilize Huffman Coding. Initially, it can be inferred that organizations such as WinZip, WinRAR, and 7-Zip are highly probable to employ Huffman coding as a means to facilitate the compression procedure, given that Huffman coding is an algorithm utilized for data compression. Huffman coding is utilized in image and video compression techniques, such as JPEG and MPEG, to facilitate the compression process. Telecommunication firms like AT&T and Verizon may employ Huffman coding to optimize data transmission and storage.

2.1 Huffman Code

The Huffman Mathematical Model can be presented as follows: The construction algorithm can be formally formalized and decomposed into multiple sequential parts. Frequency analysis can be defined as:

$$f(s) = count\ of\ \ s\ in\ S \tag{1}$$

*Corresponding Author Email: fazeaalnades@marshall.edu

Let *S* be the original string consisting of characters *s*, and let *f(s)* represent the frequency of character s in string *S*. The frequency table is created in the following manner:

$$F = \{(s, f(s)) \mid s \in S\} \tag{2}$$

where *F* is the frequency table, a set of tuples where each tuple consists of a character *s* and its frequency *f(s)*. The initialization of the priority queue is provided as:

$$Q = \text{PrinorityQueue}(F) \tag{3}$$

Using the elements from *F* as an initialization, we create a priority queue *Q*. The elements in *Q* are sorted in ascending order of frequency. The factors that decide when a node in a Huffman tree is created are:

$$n_s = (s, f(s)) \tag{4}$$

The frequency of the character *s* is represented by the Huffman Tree node *ns*. If there are many nodes in the priority queue, then. You can think of this loop as the building of a tree:

$$\begin{aligned} &while\,|Q|>1: \\ &n_a, n_b = Q.\text{pop}(), Q.\text{pop}() \\ &nc = (null, f(n_a) + f(n_b)) \end{aligned} \tag{5}$$

The two lowest-frequency nodes, n_a and n_b, are eliminated from *Q*. The frequencies of the newly created node n_c, which is the parent of the nodes n_a and n_b, are equal to the total of those of the two nodes. The root assignment is determined by the equation:

$$\text{root=Q.peek()} \tag{6}$$

The last node *Q* is transformed into the root of the Huffman Tree. To assign Huffman codes to each leaf node, traverse the tree:

$$Encode(n, p) = \begin{cases} C(s) = p \quad \text{if n is a leaf node} \\ \text{Encode}(n_l, p + "0") \\ \text{Encode}(n_r, p + "1") \end{cases} \tag{7}$$

n represents a node in the Huffman Tree, and n_1 and nr are the left and right children of *n*, respectively. *P* is the path string accumulated from the root to the node *n*. *C* is a dictionary mapping characters to their respective Huffman codes. The output of the construction is expressed as:

$$E = \bigoplus_{s_i \in S} C(s_i) \tag{8}$$

E is the encoded string constructed by concatenating the Huffman codes $C(s_i)$ for each character s_i in the original *S*. $\oplus$ denotes concatenation of string. The pseudocode implementation of Huffman codes is displayed as follows:

Algorithm 1: Huffman Tree Construction

```
Initialize frequencies as new HashMap
for each character in originalString do
    Increment character frequency in frequencies
Initialize priorityQueue as new PriorityQueue
for each character, frequency in frequencies do
    Create new HuffmanNode and add to priorityQueue
while size of priorityQueue > 1 do
    Remove two nodes with lowest frequency
    Create new node with sum of frequencies
    Add new node to priorityQueue
root ← peek of priorityQueue
```

```
Initialize huffmanCodes as new HashMap
Encode(root, "", huffmanCodes)
Initialize encodedString as new StringBuilder
for each character in originalString do
    Append Huffman code to encodedString
if root is a leaf then
    Output ← repeat root.character for its frequency
else
    Decode using root and encodedString
end procedure
```

2.2 Divide and Conquer

This algorithm aims to recursively divide a string into two halves until each substring is of length 1, updating a frequency map for each character.

$$\text{if } |S| = 1 \tag{9}$$

Then update the frequency map. $|S|$ denotes the length of the string S. The frequency map F is updated such that $F|c| = F|c| + 1$ for the character c in string S. The recursive case is expressed as follows:

$$\begin{aligned} &S = S_{left} + S_{right} \\ &S_{left} = S[0,\text{middle} - 1], S_{right} = S[\text{middle},\text{end}] \\ &\text{middle} = \left\lfloor \frac{|S|}{2} \right\rfloor \end{aligned} \tag{10}$$

S is recursively divided into two halves S_{left} and S_{right}. The function Map Frequency is called recursively in both halves.

```
Algorithm 2: Divide and Conquer Frequency Mapping
if length of inputString == 1 then
    Update frequency map
else
    middle ← length of inputString / 2
    leftString ← inputString[0, middle-1]
    rightString ← inputString[middle, end]
    MapFrequency(leftString)
    MapFrequency(rightString)
end procedure
```

2.3 Merge for Sorting by Frequency

This algorithm combines two sorted lists (sorted by frequency) into one list, preserving the order based on frequency. The initialization is done in the following manner:

$$i, j, k = 0, 0, 0 \tag{11}$$

Then, the merging process is expressed as follows:

$$\begin{aligned} &\text{while } i < |L| \text{ and } j < |R|: \\ &\text{if } F\left[L[i]\right] > F\left[R[j]\right] \text{ then } M\left[k++\right] = L[i++] \\ &\text{else } M\left[k++\right] = R\left[j++\right] \end{aligned} \tag{12}$$

L and R are two halves that are being combined, resulting in M being the merged outcome. The function $F[x]$ denotes the frequency of the element x. Adding the remaining elements:

$$\begin{aligned} &\text{while } i < |L|: M[k++] = L[i++] \\ &\text{while } i < |R|: M[k++] = R[i++] \end{aligned} \tag{13}$$

Algorithm 3: Merge for Sorting by Frequency

```
Initialize i, j, k ← 0
while i < Left.length and j < Right.length do
    if Left[i] frequency > Right[j] frequency then
        mergedString[k] ← Left[i]
        i ← i + 1
    else
        mergedString[k] ← Right[j]
        j ← j + 1
    k ← k + 1
while i < Left.length do
    mergedString[k] ← Left[i]
    i ← i + 1
    k ← k + 1
while j < Right.length do
    mergedString[k] ← Right[j]
    j ← j + 1
    k ← k + 1
end procedure
```

2.4 Modified Merge Sort for Frequency Sorting

The initial case is provided by:

$$\text{if } |S| = 1 \text{ return } S \tag{14}$$

The concept of recursive sorting is defined as follows:

$$\begin{aligned} \text{middle} &= \left\lfloor \frac{|S|}{2} \right\rfloor \\ S_{left} &= \text{FrequencySort}[S[0,\text{middle} - 1]) \\ S_{right} &= \text{FrequencySort}(S[\text{middle},\text{end}]) \end{aligned} \tag{15}$$

$$\text{return Merge}(S_{left}, S_{right}, S) \tag{16}$$

The Merge function is utilized to combine the two sorted halves.

Algorithm 4: Modified Merge Sort for Frequency Sorting

```
if length of string == 1 then
    return string
else
    middle ← length of string / 2
    leftString ← string[0, middle-1]
    rightString ← string[middle, end]
    FrequencySort(leftString)
    FrequencySort(rightString)
    return Merge(leftString, rightString, string)
end procedure
```

2.5 Greedy Technique

Initially, we compute the frequency of every individual character in the given input string.

$$f(s) = \text{count of } s \text{ in } S \tag{17}$$

Next, create the frequency table:

$$F = \{(s, f(s)) \mid s \in S\} \tag{18}$$

initialize a priority queue to store Huffman nodes, prioritizing nodes with lower frequencies. A priority queue implemented using Huffman code is provided:

$$\begin{aligned} &Q = \text{priorityQueue}(F, \text{by } f(s)) \\ &\text{Each element in } Q \text{ is Huffman code } n_s = (s,\ f(s)) \end{aligned} \tag{19}$$

create the Huffman Tree:

$$\begin{aligned} &\text{while} \mid Q \mid > 1: \\ &\text{let } n_{left} = Q.\text{remove}(), nright = Q.\text{remove}() \\ &n_{new} = (null,\ f(n_{left}) + f(n_{right}), n_{left}, n_{right}) \\ &Q.add(n_{new}) \end{aligned} \tag{20}$$

n_{left} and n_{right} are nodes with the lowest frequencies. n_{new} is a new node with its frequency being the sum of n_{left} and n_{right}. Root assignment as follows:

$$\text{root} = Q.\text{peek}() \tag{21}$$

The encoding process is as follows:

$$\text{encode}(n, c)\begin{cases} H(n.\text{character}) = c \text{ if } n \text{ is a leaf} \\ \text{Encode}(n.\text{leftChild}, c + \text{"0"}) \\ \text{Encode}(n.\text{rightChild}, c + \text{"1"}) \end{cases} \tag{22}$$

n symbolizes a node in the Huffman Tree. The code string *c* represents the sequence of code gathered from the root to the node *n*. *H* is a dictionary that associates characters with their respective Huffman codes.

Algorithm 5: Greedy Technique

```
Initialize frequencies as new HashMap
for each character in originalString do
    Increment character frequency in frequencies
Initialize priorityQueue as new PriorityQueue based on frequency
for each character, frequency in frequencies do
    Create new HuffmanNode(character, frequency) and add to priorityQueue
while size of priorityQueue > 1 do
    left ← priorityQueue.remove() (Node with lowest frequency)
    right ← priorityQueue.remove() (Node with next lowest frequency)
    sumFrequency ← left.frequency + right.frequency
    Create new node with no character, sumFrequency, left, right
    priorityQueue.add(new node)
root ← priorityQueue.peek() (The only node in the priorityQueue is the root of the Huffman Tree)
Initialize huffmanCodes as new HashMap
Encode(root, "", huffmanCodes)
end procedure
procedure Encode(node, code, huffmanCodes)
  if node is not null then
    if node is a leaf then
      if code is empty then
        huffmanCodes.put(node.character, "1") (Handle single character edge case)
      else
        huffmanCodes.put(node.character, code)
    Encode(node.leftChild, code + "0", huffmanCodes)
    Encode(node.rightChild, code + "1", huffmanCodes)
end procedure
```

3. TIME COMPLEXITY

When evaluating the temporal complexity of the Huffman coding algorithm, several steps contribute to the overall difficulty. The greedy implementation starts by creating a frequency table based on the input string. Inserting each of the *k* unique characters into the priority queue has a complexity of *O(log k)*. Thus, the total complexity of this phase is *O(k log k)*. Given that the value of *k* cannot exceed *n*, this aspect typically simplifies to *O(n log n)* in the worst-case situation, assuming that each character is unique. The Huffman tree is built iteratively by taking the two nodes with the lowest frequency from the queue and combining them into a new node, which is then inserted back into the queue. Every iteration of the technique takes logarithmic time, which is directly proportional to the queue size, and the process is repeated k-1 times. The result is a total complexity of O(k log k), which, for values of k close to n, is roughly equivalent to O(n log n). Assigning codes to individual characters involves encoding, which involves traversing the tree. When each letter is checked one at a time, this pass takes O(n) time. On the other hand, the decryption process has to be done for every piece of ciphertext. n letters are repeated n times, which gives the length of the encrypted string. This method takes O(n) time. Lastly, in the greedy Huffman coding method, it takes O(n log n) time to do the work needed to build the Huffman tree and process the priority queue.

Divide and conquer methods are similar, but they get trickier when you need to sort characters because the changed sequence order groups them by how often they appear. The sorting method takes O(n log n) time to run. Using this method to build Huffman trees one step at a time takes O(n log n) time. The amount of time needed for greedy encryption and decryption is the same. It takes O(n log n) time to run greedy Huffman algorithms and divide-and-conquer algorithms. The problem that still needs to be solved is making a Huffman tree, a sorting method, and a frequency table.

4. SPACE COMPLEXITY

The time complexity of Huffman coding is closely linked to space complexity because of how the data is structured. We use a hash map that takes *O(k)* amount of room to store the frequency value of each character in the string. In this case, k stands for the string's total amount of unique characters. Even though *K* can be equal to n, the number of letters in the alphabet is generally less than n. There must be *O(k)* gaps between the Huffman nodes in the top row, where k is the number of unique characters. It also stores the Huffman code that goes with each word as a separate hash. The amount of room needed to store these codes gets exponentially harder as *K* goes up. The amount of room that greedy technology needs is directly related to the number of unique characters. In the worst case, the space complexity can be cut down to *O(n)* if each character is unique. In real life, though, this is *O(k)*. It takes *O(k)* room to use the divide-and-conquer approach for frequency and Huffman codes because they share data structures. Unlike the greedy approach, the divide and conquer strategy often does not employ a priority queue during the construction of a Huffman tree. Instead, it constructs the tree directly from data partitions, resulting in significant space expenses caused by improper handling of repetitive stack calls. The majority of storage space requirements are still attributed to frequency maps and Huffman codes. Both approaches have a fundamental constraint on space, requiring *O(k)* storage for the Huffman code and frequency map. Assuming that the value of k is approximately equal to the value of n, both approaches have a total spatial complexity of *O(n)* in the worst case. The greedy approach employs a priority queue, but it does not fundamentally alter the broad classification of space complexity.

5. RESULTS AND DISCUSSION

When comparing the time and space complexities of implementing Huffman coding using the greedy strategy versus the divide and conquer approach, both methods exhibit the same time and space complications. Both methods exhibit a time complexity of *O (n log n)* and function linearly. All algorithms have the same space complexity, each necessitating *O(n)* space. However, the greedy technique demonstrates more computational efficiency in comparison to the divide-and-conquer approach, primarily because of the recursive tradeoffs. However, the main difference between the two was the difference in the levels of execution complexity. The greedy technique, inherent to Huffman coding, was notably easier to apply in contrast to the divide-and-conquer approach. Generally, despite the comparable time and space challenges of both approaches, it is recommended to prioritize the native approach due to its straightforward implementation. We performed multiple iterations of the algorithms using the greedy and divide and conquer methods, with different string lengths. Subsequently, we computed the mean outcomes for every string length. Both programs were tested using a standardized set of randomly generated strings. The compression ratio is a measure of the space saved as a percentage compared to the original size of the compressed string.

Table 1. Divide and Conquer Data

String Length	Time to Execute (ms)	Compression Ratio
10	0.6	0.575
25	1	0.47
50	1.2	0.382
100	2.8	0.32
500	7.4	0.299

1000	11	0.297
10000	48.6	0.296
50000	146.6	0.295

Table 2. Greedy Data

String Length	Time to Execute (ms)	Compression Ratio
10	1	0.575
25	1	0.475
50	1	0.3975
100	1	0.345
500	2	0.309
1000	3	0.304
10000	8	0.298
50000	9	0.297

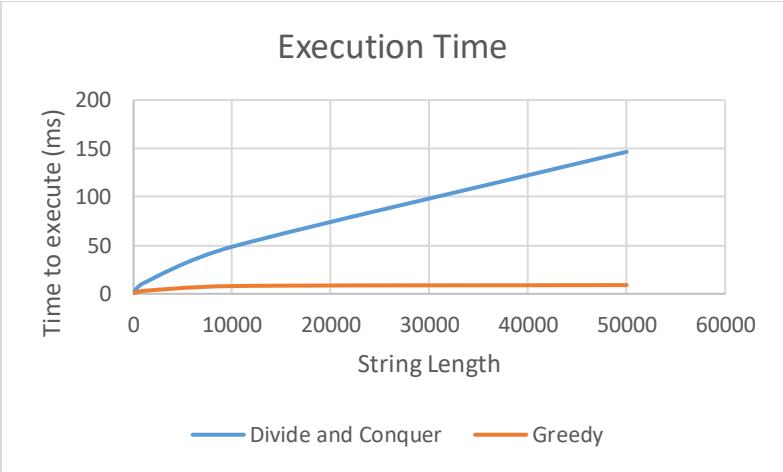

Figure 1. Time Complexity comparison between Divide and Conquer and Greedy Technique

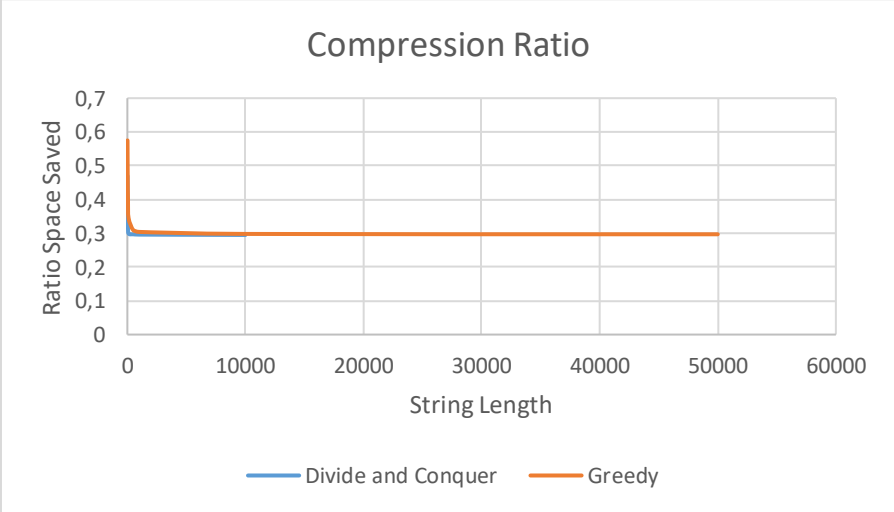

Figure 2. Comparison Ratio between Divide and Conquer and Greedy Technique

According to the trends depicted in these graphs, greedy and divide-and-conquer algorithms exhibit similar performance for small input sizes. Nevertheless, when dealing with substantial input sizes, the greedy method will outperform other approaches. The execution time graph reveals that the line representing the divide and conquer approach exhibits a resemblance to a linear graph, but the greedy method demonstrates a closer resemblance to a logarithmic graph. When considering the compression ratio, we observe that the amount of space saved drops to a fixed number as the size of the input grows. The reason behind this is that we

conducted tests on our programs utilizing a collection of randomly generated strings, resulting in the normalization of the distribution of repeated characters and thus increasing the size of the strings.

6. CONCLUSION

The research project effectively devised and executed Huffman coding by integrating the greedy and divide-and-conquer approaches. We quantified the temporal and spatial efficiencies attained by these techniques. Our research shows that both methods effectively compress data, but the greedy technique is more time efficient because it is less complex and computationally costly. Nevertheless, the divide-and-conquer strategy significantly improved the adaptability and ease of implementing intricate systems, albeit with a modest trade-off in time efficiency. The study emphasizes the necessity of discovering effective algorithmic techniques for data compression to improve efficiency and user experience. Our research has provided us with a comprehensive understanding of many data compression strategies, including but not limited to Huffman coding. Possessing this knowledge is crucial for future projects that necessitate sophisticated data compression techniques.

REFERENCES

[1] A. Adadi, "A survey on data-efficient algorithms in big data era," *Journal of Big Data*, vol. 8, no. 1, p. 24, 2021.
[2] J. Reis and M. Housley, *Fundamentals of data engineering*. " O'Reilly Media, Inc.", 2022.
[3] F. S. Mahammad and V. M. Viswanatham, "Performance analysis of data compression algorithms for heterogeneous architecture through parallel approach," *The Journal of Supercomputing*, vol. 76, no. 4, pp. 2275-2288, 2020.
[4] G. Wu, F. Zhou, G. Ding, Q. Wu, and X.-Y. Li, "An efficient heterogeneous edge-cloud learning framework for spectrum data compression," *IEEE Transactions on Mobile Computing*, 2022.
[5] T. Choudhary, V. Mishra, A. Goswami, and J. Sarangapani, "A comprehensive survey on model compression and acceleration," *Artificial Intelligence Review*, vol. 53, pp. 5113-5155, 2020.
[6] J. Tian *et al.*, "Revisiting huffman coding: Toward extreme performance on modern gpu architectures," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021: IEEE, pp. 881-891.
[7] M. A. Rahman and M. Hamada, "Burrows–wheeler transform based lossless text compression using keys and Huffman coding," *Symmetry*, vol. 12, no. 10, p. 1654, 2020.
[8] Y. Zhou *et al.*, "47 Gbps 100 m ultra-high-speed free-space visible light tricolor laser communication system utilizing time domain hybrid Huffman coding," *Optics Express*, vol. 32, no. 14, pp. 24811-24825, 2024.
[9] C. Jaranilla and J. Choi, "Requirements and Trade-Offs of Compression Techniques in Key–Value Stores: A Survey," *Electronics*, vol. 12, no. 20, p. 4280, 2023.
[10] S. S. Mukherjee, P. Sarkar, and P. J. Bickel, "Two provably consistent divide-and-conquer clustering algorithms for large networks," *Proceedings of the National Academy of Sciences*, vol. 118, no. 44, p. e2100482118, 2021.
[11] S. Khuller, B. Raghavachari, and N. E. Young, "Greedy methods," in *Handbook of Approximation Algorithms and Metaheuristics*: Chapman and Hall/CRC, 2018, pp. 55-69.
[12] T. H. Alabi. "What is a greedy algorithm? examples of greedy algorithms. ." https://www.freecodecamp.org/news/greedy-algorithms/ (accessed 2023).
[13] R. Hannane, A. Elboushaki, and K. Afdel, "A divide-and-conquer strategy for facial landmark detection using dual-task CNN architecture," *Pattern Recognition*, vol. 107, p. 107504, 2020.
[14] A. Qaroush, A. Awad, A. Hanani, K. Mohammad, B. Jaber, and A. Hasheesh, "Learning-free, divide and conquer text-line extraction algorithm for printed Arabic text with diacritics," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 9, pp. 7699-7709, 2022.
[15] U. Jayasankar, V. Thirumal, and D. Ponnurangam, "A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications," *Journal of King Saud University-Computer and Information Sciences*, vol. 33, no. 2, pp. 119-140, 2021.
[16] R. Ranjan, "Canonical Huffman coding based image compression using wavelet," *Wireless Personal Communications*, vol. 117, no. 3, pp. 2193-2206, 2021.
[17] X. Liu, P. An, Y. Chen, and X. Huang, "An improved lossless image compression algorithm based on Huffman coding," *Multimedia Tools and Applications*, vol. 81, no. 4, pp. 4781-4795, 2022.
[18] B. A. Wijaya, S. Siboro, M. Brutu, and Y. K. Lase, "Application of huffman algorithm and unary codes for text file compression," *Sinkron: jurnal dan penelitian teknik informatika*, vol. 6, no. 3, pp. 1000-1007, 2022.
[19] R. R. Gajjala, S. Banchhor, A. M. Abdelmoniem, A. Dutta, M. Canini, and P. Kalnis, "Huffman coding based encoding techniques for fast distributed deep learning," in *Proceedings of the 1st Workshop on Distributed Machine Learning*, 2020, pp. 21-27.
[20] A. K. M Al-Qurabat, "A lightweight Huffman-based differential encoding lossless compression technique in IoT for smart agriculture," *International Journal of Computing and Digital System*, 2021.
[21] F. Prakash, V. Singh, and A. K. Saxena, "An Evaluation of Arithmetic and Huffman Coding in Data Compression & Source Coding," in *2024 International Conference on Optimization Computing and Wireless Communication (ICOCWC)*, 2024: IEEE, pp. 1-6.
[22] T. Hidayat, M. H. Zakaria, and A. N. C. Pee, "Increasing the Huffman generation code algorithm to equalize compression ratio and time in lossless 16-bit data archiving," *Multimedia Tools and Applications*, vol. 82, no. 16, pp. 24031-24068, 2023.