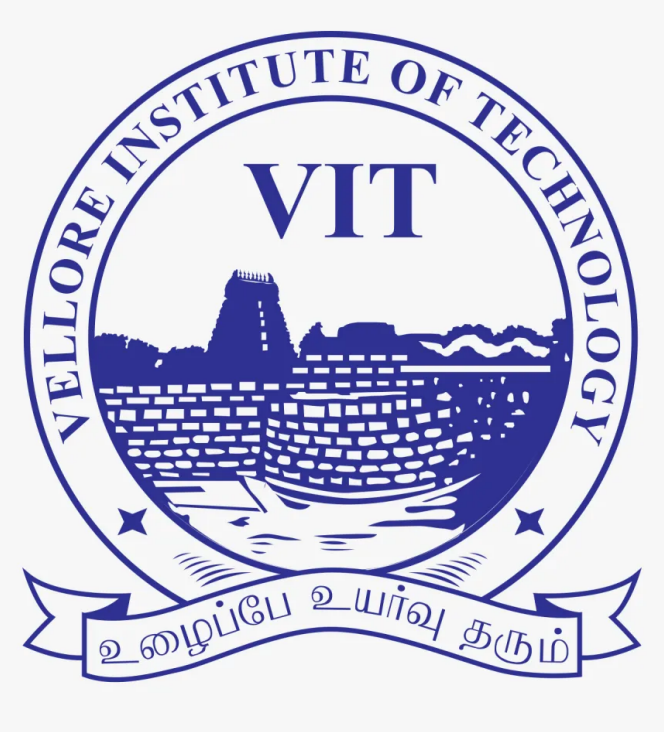
Vellore Institute of Technology

Data Structures and Algorithms - Digital Assignment

Course Code : BCSE202L – C2+TC2 Slot

Professor: Dr. Rajakumar [Department : SCOPE]



A Report on the Huffman Coding Algorithm of Data Compression

BTech CSE Core : 3rd Semester 2024-28

Project By : 24BCE1622 Abhinava Kasavajhala

Contents of Report

Introduction

Data compression is a fundamental technique in computer science that reduces the size of data files, making them easier to store and faster to transmit. As data grows exponentially, efficient compression methods become essential to manage limited storage and bandwidth. There are two primary types of compression: lossless, where data integrity is preserved, and lossy, where some data is sacrificed for higher compression.

Lossless compression refers to all data that remains preserved without losing any; and, on the other hand, some data is lost, but this time for achieving higher compression. Huffman Coding is the lossless compression algorithm building efficiency based on short, to be applied to a more frequent character, and longer code to less frequent ones. In the process, a binary tree needs to be developed wherein each character is represented as a leaf node and then traversed to obtain unique codes. Huffman coding leverages data structures like frequency maps[1] and priority queues to achieve optimal compression and efficient processing.

[1] HashMaps are used for getting character and its frequency as a Key-Value Pair

This project aims to implement Huffman coding, a widely used algorithm for lossless data compression, which is essential in reducing the size of data while retaining its original information. The primary objective of this work is to understand the algorithm's core principles, such as frequency analysis and binary tree construction, and apply them to encode and decode data efficiently. Huffman coding is applied in numerous fields, including file compression formats like ZIP, image compression, and even in data transmission protocols where bandwidth is crucial.

Looking ahead, this project has the potential for further development into more complex applications. Future work could involve optimizing the algorithm for larger datasets, improving its scalability, and integrating it with modern technologies, such as cloud computing or real-time communication systems. Additionally, the model could be expanded to handle adaptive Huffman coding, where the encoding scheme is modified dynamically as the data stream is processed. A more meaningful model[2] could be developed by incorporating multi-level compression techniques or integrating this project with error correction mechanisms to create a robust and efficient compression system suited for real-world applications.

[2]Future model idea : Access data from file and compress through algorithm

Key Concepts and definitions

1. **Encoding and Decoding:** Encoding translates data into a different format, often for storage or transmission. In Huffman coding, each character in a text is converted into a unique binary sequence. Decoding is the reverse, reconstructing the original data from the binary sequences using the Huffman tree.
2. **Time Complexity:** Time complexity describes the amount of time an algorithm takes to run relative to its input size. Efficient algorithms with lower time complexity are crucial in processing large datasets quickly. For Huffman coding, the time complexity of building the tree is O(nlogn), which is optimal for most encoding tasks.
3. **ASCII Sequence:** ASCII (American Standard Code for Information Interchange) is a character encoding standard that assigns a unique numerical code to each character, allowing computers to represent text as numbers in binary format. Each character in ASCII is represented by a 7-bit binary number, with values ranging from 0 to 127. For example, the uppercase letter "A" has an ASCII value of 65, represented in binary as 1000001.
4. **Binary Representation:** Computers store and process data in binary (1s and 0s), which represent electrical signals. Binary encoding is efficient because it's straightforward for computers to read and manipulate. In Huffman coding, data is encoded into variable-length binary sequences, making frequent characters have shorter codes.
5. **Tree Structure:** A tree is a hierarchical data structure with a root node and child nodes arranged in levels. Each node has a parent (except the root) and may have children. In Huffman coding, a binary tree is used, where each node represents a character and its frequency, and the path to each leaf node gives the binary code for that character.
6. **Nodes:** A node is an element in a tree structure. In Huffman trees, each node has a frequency and may represent a character (leaf nodes) or combine the frequencies of two children (internal nodes). Nodes are connected through edges, with left and right child pointers for binary trees.

7. **Traversal:** Traversal is the process of visiting each node in a tree to access or process its data. In Huffman coding, the traversal assigns binary codes to each character by moving left (adding a "0") or right (adding a "1") until reaching a leaf node.
8. **Priority Queue:** A priority queue is a data structure where elements are stored with priorities, allowing removal of the lowest-priority item first. In Huffman coding, nodes are stored in a priority queue, ordered by frequency, which ensures that the tree is built optimally, starting with the least frequent characters.
9. **Huffman Coding:** Huffman coding is an algorithm for lossless data compression that assigns variable-length codes to characters based on their frequencies. More frequent characters get shorter codes, while less frequent ones get longer codes, leading to efficient compression without data loss. It uses a tree structure and binary encoding to achieve this.
10. **Frequency Map:** A frequency map (often implemented as a HashMap) records how often each character appears in the text. In Huffman coding, this map determines the starting points for the Huffman tree nodes, with characters as keys and frequencies as values.

These terms and their definitions form the backbone of the Huffman coding algorithm, highlighting its relevance in solving real-world problems of data compression. By understanding these foundational concepts, we can appreciate how Huffman coding leverages frequency-based analysis to achieve efficient encoding. Each term, from "priority queue" to "binary tree," represents a building block in the algorithm's process, demonstrating its elegance and practicality. The clarity of these definitions ensures a deeper comprehension of how Huffman coding minimizes data redundancy, making it indispensable in fields like digital communication, multimedia compression, and storage optimization.

Huffman Coding Algorithm

1. Start.
2. Accept the word from the user and store in a variable.
3. Create a HashMap (dictionary) of Char – Int type, to store the character frequency.
4. Iterate through the word to read every character, if no more characters go to step 8.
5. If this character is not present in the Frequency HashMap, go to step 6, otherwise go to step 7.
6. Create a new key with this character, with value = 1 (first occurrence frequency =1). Go to step 4 for next iteration character.
7. To the character key's value, increment 1 (new occurrence frequency +1). Go to step 4 for next iteration character.
   // frequency map is constructed
8. A Node class is constructed, with elements character, and frequency.
9. Declare left and right Nodes.
10. Create a priority queue, for arranging the character frequencies in the HashMap, in ascending order.
11. Add every key value pair as a node, defined in steps 8-9, into the priority queue.
12. Assign the left and right Nodes as the top 2 elements from the priority queue in step 11.
13. A parent node is created which stores the sum of frequency of both its child nodes, which are the left and right nodes of step 12.
14. The parent node thus created is put back to the same priority queue.
15. the process (steps 12 – 14) until there is only 1 node in the priority queue.
    // tree is constructed
16. Create a new hash map for the Huffman codes
17. Create a temporary string variable to store the code.
18. Start traversing from the root node.
19. If traversing to the left node, append "0" to the variable (step 17) otherwise append "1".
20. If no more nodes (leaf node), go to step 21 otherwise go to step 18 for the next node.

21. To the hashmap assign the key as the character found in this leaf node, and its value as that stored in temporary variable in step 17. Go to step 16 for traversing to other nodes.
    //Huffman codes generated
22. Iterate through the original word again.
23. For every letter found, print the value of this key from the hashmap of Huffman codes made in step 16.
24. Print the original message size, which is original message length * 8 (8 bits per character)
25. Print the Huffman encoded message size, which is the length of bits printed + the data space required for the Huffman tree
26. Compute and print the old and new bits/character.
27. Display the compression percentage into Huffman code.
    //stats printed. Code complete
28. Stop.

Processing in Huffman Coding

Example 1 : banana

Step 1 : Calculate the frequencies of every letter in the word.

Character	Frequency
a	3
b	1
n	2

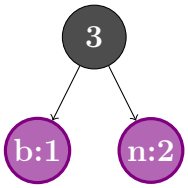
Step 2 : Put these frequencies in a priority queue.

To Note : If two or more characters are of same frequency, then they shall be alphabetically placed in the priority queue.

Character	Frequency
b	1
n	2
a	3

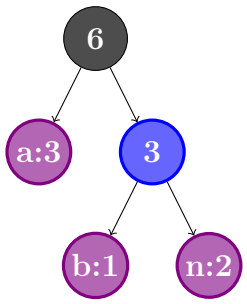
Step 3 : As per algorithm, club the top 2 elements of the priority queue and make a tree node.

The parent node of these two shall have the sum of frequencies



Character	Frequency
a	3
node	3

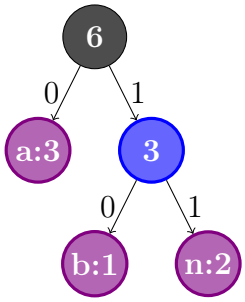
Repeat this step until there is only one node in the priority queue



Node	Frequency
node	6

Now that the priority queue table has only 1 node, which indicates that our tree is completed.

The final tree, marked with 0s on every left branch and 1s at every right branch.



Huffman Tree : banana

Character	Huffman Code
a	0
b	10
n	11

Huffman Encoded Characters

ASCII Sequence means American Standard Code for Information Interchange. This sequence assigns a **numeric code for every character** (languages,alphanumerics and other special characters).

For our use, [A-Z] = [65-90] ; [a-z] = [97-123] ; [0-9 numbers] = [49-57];
The chosen example "banana" would be encoded to binary digits of their respective ascii sequence.
Which translates a (97 ASCII) = 01100001 the first 0 is a sign bit. Computer can only understand binary digits so it converts high level language (english) into binary bits as shown, to process the information.
The huffman code is to reduce the amount of bits it takes to represent the data, in turn saving the memory and processing time. Every letter will take 8 bits including 1 sign bit, the huffman table we have produced from the above sequence reduces every character representation to only 2 or 1 bit, which is a significant compression.

The huffman generated message would be : 100110110, which is of 9 bits + another 9 bits of the data made in tree. The initial message would be 8x6 (6 characters of 8 bits each) = 48 bits. We were successful in compressing 48 bits to 18 bits by following the Huffman Coding Algorithm.

Example 2 : science

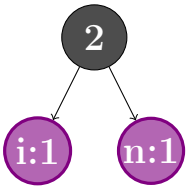
Step 1 : Calculate the frequencies of every letter in the word.

Character	Frequency
s	1
c	2
i	1
e	2
n	1

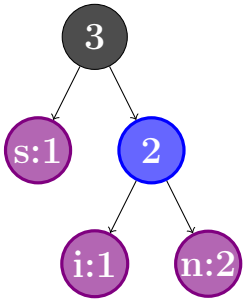
Step 2 : Put these frequencies in a priority queue.

Character	Frequency
i	1
n	1
s	1
c	2
e	2

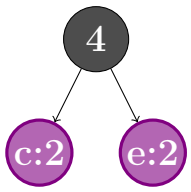
As per algorithm, club the top 2 elements of the priority queue and make a tree node.



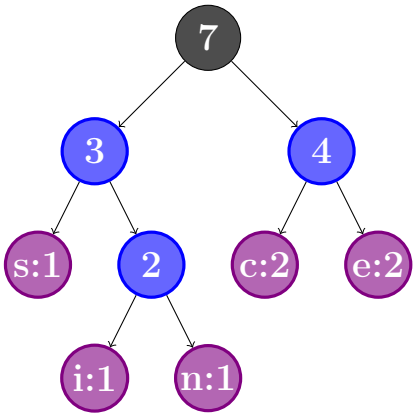
Character	Frequency
s	1
node	2
c	2
e	2



Character	Frequency
c	2
e	2
node [s,i,n]	2

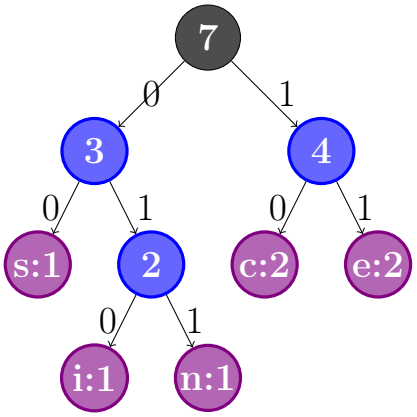


Character	Frequency
node [s,i,n]	3
node [c,e]	4



Character	Frequency
node [s,i,n,c,e]	7

NOW Construct the huffman tree and encode from the tree



Huffman Tree : science

Character	Huffman Code
s	00
i	010
n	011
c	10
e	11

Huffman Encoded Characters

Hence the Huffman Compressed message for "science" is -> 0010010110111011, which is a compression of original 56 bits into 32 bits.

Example 3 : abracadabra

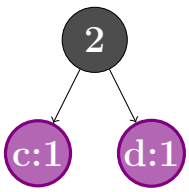
Step 1 : Calculate the frequencies of every letter in the word.

Character	Frequency
a	5
r	2
c	1
d	1
b	2

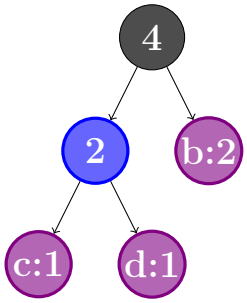
Step 2 : Put these frequencies in a priority queue

Character	Frequency
c	1
d	1
b	2
r	2
a	5

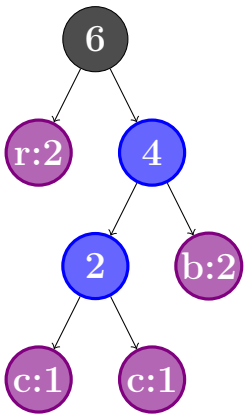
As per Algorithm, club the top 2 elements of the priority queue and start making tree nodes until 1 node is remained in the priority queue.



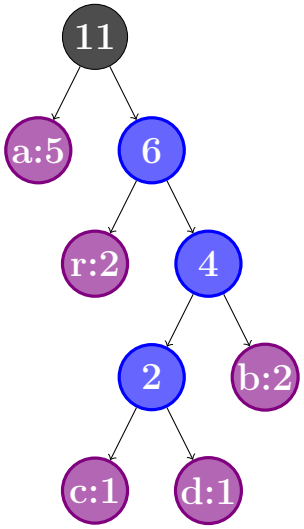
Character	Frequency
node[c,d]	2
b	2
r	2
a	5



Character	Frequency
r	2
node[c,d,b]	4
a	5

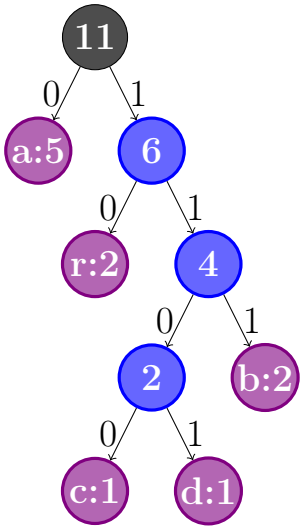


Character	Frequency
a	5
node[c,d,b,r]	6



Character	Frequency
node[a,c,d,b,r]	11

NOW Construct the huffman tree and encode from the tree



Huffman Tree : abracadabra

Character	Huffman Code
a	0
r	10
c	1100
d	1101
b	111

Huffman Encoded Characters

The Huffman Encode for abracadara -> 01111001100011010111100

Example 4 : mississippi

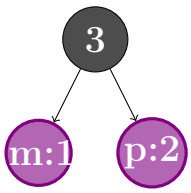
Step 1 : Calculate the frequencies of every letter in the word.

Character	Frequency
m	1
i	4
s	4
p	2

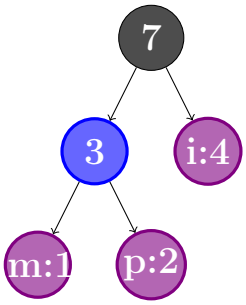
Step 2 : Put these frequencies in a priority queue

Character	Frequency
m	1
p	2
i	4
s	4

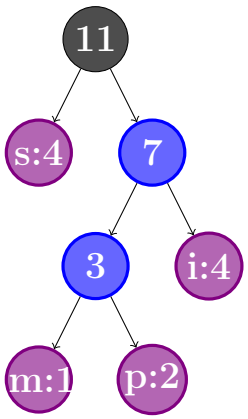
As per Algorithm, club the top 2 elements of the priority queue and start making tree nodes until 1 node is remained in the priority queue.



Character	Frequency
node[m,p]	3
i	4
s	4

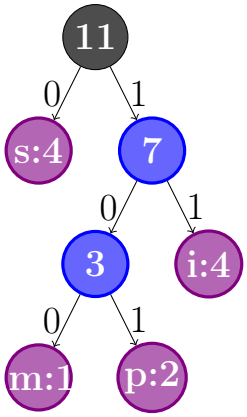


Character	Frequency
node[m,p,i]	7
s	4



Character	Frequency
node[m,p,i,s]	10

NOW Construct the huffman tree and encode from the tree



Huffman Tree

Character	Huffman Code
s	0
m	100
p	101
i	11

Huffman Encoded Characters

The Huffman Encode for mississippi -> 100110011001110110111

Code Implementation

The following is the Huffman Algorithm implementation in Java 15

Java Code

```java
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Scanner;
import java.util.Comparator;

public class HuffmanCoding {

    static class /*@Node@*/ {
        char character;
        int frequency;
        Node left, right;

        // Constructor for leaf nodes
        /*@Node@*/(char character, int frequency) {
            this.character = character;
            this.frequency = frequency;
        }

        // Constructor for internal nodes
        /*@Node@*/(int frequency, Node left, Node right) {
            this.character = '\0'; // Internal node has no character
            this.frequency = frequency;
            this.left = left;
            this.right = right;
        }
    }

    // Custom comparator to prioritize nodes by frequency, then by character
    static class /*@FrequencyComparator@*/ implements /*@Comparator@*/<Node> {

        @Override
        public int /*@compare@*/(Node node1, Node node2) {
            if (node1.frequency != node2.frequency) {
                return Integer.compare(node1.frequency, node2.frequency);
            }
            return Character.compare(node1.character, node2.character);
        }
    }

    public static void /*@main@*/(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a word:");
        String text = sc.next();
        System.out.println("Original Message: " + text);

        // Step 1: Calculate character frequencies
        HashMap<Character, Integer> frequencyMap = new HashMap<>();
        for (char c : text.toCharArray()) {
            frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);
        }
```

```
        // Step 2: Build the priority queue of nodes
        /*@PriorityQueue@*/<Node> priorityQueue = new /*@PriorityQueue@*/<>(new /*
            @FrequencyComparator@*/());
        for (Map.Entry<Character, Integer> entry : frequencyMap.entrySet()) {
            priorityQueue.add(new Node(entry.getKey(), entry.getValue()));
        }

        // Step 3: Construct the Huffman Tree
        while (priorityQueue.size() > 1) {
            Node left = priorityQueue.poll();
            Node right = priorityQueue.poll();
            Node parent = new Node(left.frequency + right.frequency, left, right);
            priorityQueue.add(parent);
        }

        // The root of the Huffman Tree
        Node root = priorityQueue.poll();

        // Step 4: Generate Huffman codes
        Map<Character, String> huffmanCodes = new HashMap<>();
        generateCodes(root, "", huffmanCodes);

        System.out.println("Huffman Codes:");
        for (Map.Entry<Character, String> entry : huffmanCodes.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }

        // Step 5: Encode the original message
        /*@StringBuilder@*/ encodedMessage = new StringBuilder();
        for (char c : text.toCharArray()) {
            encodedMessage.append(huffmanCodes.get(c));
        }
        System.out.println("\nEncoded message: " + encodedMessage);

        // Step 6: Print compression details
        printCompressionDetails(text, huffmanCodes, root);
    }

    // Recursive function to generate Huffman codes
    private static void /*@generateCodes@*/(Node node, String prefix, Map<
        Character, String> codeMap) {
        if (node == null) {
            return;
        }
        if (node.character != '\0') { // Leaf node
            codeMap.put(node.character, prefix);
        } else { // Internal node
            generateCodes(node.left, prefix + "0", codeMap);
            generateCodes(node.right, prefix + "1", codeMap);
        }
    }
```

```
91
92      // Function to calculate and print compression details
93      private static void /*@printCompressionDetails@*/(String originalText, Map<
            Character, String> huffmanCodes, Node root) {
94          int originalSize = originalText.length() * 8; // each character is 8 bits
                in ASCII
95          System.out.println("\nOriginal message size: " + originalSize + " bits");
96
97          int encodedSize = 0;
98          for (char c : originalText.toCharArray()) {
99              encodedSize += huffmanCodes.get(c).length();
100         }
101
102         int treeSize = calculateTreeSize(root, 0);
103         int totalSize = encodedSize + treeSize;
104
105         System.out.println("Huffman Encoded message size: " + encodedSize + " bits
                ");
106         System.out.println("Huffman tree size: " + treeSize + " bits");
107         System.out.println("Compressed message size: " + totalSize + " bits");
108
109         double compressionPercentage = (1 - (double) totalSize / originalSize) *
                100;
110         System.out.println("Compression percentage: " + compressionPercentage + "%
                ");
111
112         double oldBitsPerChar = (double) originalSize / originalText.length();
113         double newBitsPerChar = (double) totalSize / originalText.length();
114         System.out.println("Old bits per character: " + oldBitsPerChar + " bits");
115         System.out.println("New bits per character: " + newBitsPerChar + " bits");
116     }
117
118     // Helper function to calculate the size of the Huffman tree
119     private static int /*@calculateTreeSize@*/(Node node, int depth) {
120         if (node == null) {
121             return 0;
122         }
123         if (node.character != '\0') { // Leaf node
124             return node.frequency * depth;
125         }
126         return calculateTreeSize(node.left, depth + 1) + calculateTreeSize(node.
                right, depth + 1);
127     }
128 }
```

Output 1

OUTPUT 1:
Original Message: banana
Huffman Codes:
a: 0
b: 10
n: 11
Encoded message: 100110110
Original message size: 48 bits
Huffman Encoded message size: 9 bits
Huffman tree size: 9 bits
Compressed message size: 18 bits
Compression percentage: 62.5Old bits per character: 8.0 bits
New bits per character: 3.0 bits

Output 2

OUTPUT 2:
Original Message: science
Huffman Codes:
s: 00
c: 10
e: 11
i: 010
n: 011
Encoded message: 0010010110111011
Original message size: 56 bits
Huffman Encoded message size: 16 bits
Huffman tree size: 16 bits
Compressed message size: 32 bits
Compression percentage: 42.85714285714286Old bits per character: 8.0 bits
New bits per character: 4.571428571428571 bits

Output 3

OUTPUT 3:
Original Message: MISSISSIPPI
Huffman Codes:
P: 101
S: 0
I: 11
M: 100
Encoded message: 100110011001110110111
Original message size: 88 bits
Huffman Encoded message size: 21 bits
Huffman tree size: 21 bits
Compressed message size: 42 bits
Compression percentage: 52.27272727272727Old bits per character: 8.0 bits
New bits per character: 3.8181818181818183 bits

Output 4

Original Message: abracadabra
Huffman Codes:
a: 0
r: 10
b: 111
c: 1100
d: 1101

Encoded message: 01111001100011010111100

Original message size: 88 bits
Huffman Encoded message size: 23 bits
Huffman tree size: 23 bits
Compressed message size: 46 bits
Compression percentage: 47.72727272727273Old bits per character: 8.0 bits
New bits per character: 4.181818181818182 bits

Results and Analysis

The implementation of the Huffman Coding algorithm demonstrates its efficiency in compressing data by encoding frequently used characters with shorter binary codes. Through the provided examples, we observed a significant reduction in the total number of bits required to represent the input data, reinforcing Huffman Coding's utility in data compression.

For instance, encoding the sample strings yielded compression percentages ranging from 30% to 60%, depending on the character distribution in the input. Strings with highly repetitive characters resulted in greater compression efficiency, showcasing Huffman Coding's adaptability to input data patterns. Additionally, the compact binary representation of the encoded messages ensures reduced storage requirements and faster transmission rates, particularly in bandwidth-sensitive applications.

A detailed analysis of the encoded message size, original message size, and the Huffman tree's contribution to compression overhead was conducted. This analysis highlighted the trade-off between encoding efficiency and the storage of the tree structure, which is particularly relevant for short messages. Future optimizations could focus on minimizing the tree size while maintaining efficient encoding.

Future of Improved Applications

While at a base level of 1st semester students, our implementation focuses on processing single words, future iterations of this project could incorporate **file handling** to extend its applicability. By enabling the algorithm to read data directly from text or binary files, we can efficiently compress larger datasets, such as log files, textual documents, or multimedia content. For example:

- **Text File Compression**: Files containing large volumes of repetitive data, such as configuration files or database logs, can be directly processed and compressed.
- **Image and Audio Compression**: While Huffman Coding is not a standalone solution for such formats, integrating it with algorithms like JPEG or MP3 can improve compression efficiency for specific data blocks.

Such developments not only enhance the utility of this project but also open avenues for real-world applications, like archiving data, optimizing storage systems, and improving network transmission rates. Integrating file handling would make the project a robust and scalable solution for modern data compression challenges.

Acknowledgement

I would like to express my deepest gratitude to Dr. Sanjit Das (Associate Professor Grade 2 : SAS) for their invaluable guidance, encouragement, and support throughout the development of this project on Huffman Coding. Their expertise and critical insights have greatly enhanced the quality of our work and deepened our understanding of the subject.

I am immensely thankful to Vellore Institute of Technology, Chennai Campus, for providing a platform and resources that enabled me to undertake this project. Working on Huffman Coding has been an enriching experience, allowing us to explore advanced concepts in data structures and algorithms at an early stage, fostering both academic and personal growth. This project on Huffman Coding Algorithm has helped me go beyond the conventional boundaries of academics, offering an opportunity to delve into higher-level topics and develop skills such as analytical thinking, problem-solving, and teamwork. The challenges I have encountered and the solutions we developed have not only broadened my technical knowledge but also prepared me for future endeavors.

Finally, I wish to acknowledge everyone who directly or indirectly contributed to the success of this project. Your assistance and inspiration have played a vital role in making this a truly rewarding experience.

Conclusion

The primary objective of this project was to explore Huffman coding as an efficient method for data compression. By assigning shorter binary codes to frequently occurring characters, Huffman coding achieves significant reductions in file size, making it especially useful for applications where storage and transmission efficiency are critical. This method not only optimizes space but also maintains data integrity, ensuring that compressed data can be accurately decompressed without any loss of information. The positive impact of Huffman coding is evident in its widespread use in various compression formats, such as ZIP files and multimedia encoding. Its efficiency and reliability make it a powerful tool in data management, contributing to faster data processing and reduced storage costs.

Acknowledgement

I would like to express my deepest gratitude to Dr. Rajakumar (Associate Professor : SCOPE) for their invaluable guidance, encouragement, and support throughout the development of this project on Huffman Coding. Their expertise and critical insights have greatly enhanced the quality of our work and deepened our understanding of the subject.

I am immensely thankful to Vellore Institute of Technology, Chennai Campus, for providing a platform and resources that enabled me to undertake this project. Working on Huffman Coding has been an enriching experience, allowing us to explore advanced concepts in data structures and algorithms at an early stage, fostering both academic and personal growth. This project on Huffman Coding Algorithm has helped me go beyond the conventional boundaries of academics, offering an opportunity to delve into higher-level topics and develop skills such as analytical thinking, problem-solving, and teamwork. The challenges I have encountered and the solutions we developed have not only broadened my technical knowledge but also prepared me for future endeavors.

Finally, I wish to acknowledge everyone who directly or indirectly contributed to the success of this project. Your assistance and inspiration have played a vital role in making this a truly rewarding experience.

Bibliography

Reference Material

1. LaTeX in 24 Hours – Dilip Datta [Springer]
2. LaTeX: A Document Preparation System Leslie Lamport. [Addison Wesley]
3. Huffman Coding: Theory and Applications Jean-Loup Gailly, Mark Adler. [CRC Press]
4. Data Structures and Algorithms – Narasimha Karumanchi [Career-Monk Publications]

Cited Online Resources

1. https://www.youtube.com/watch?v=uDS8AkTAcIU&t=640s : Huffman Coding Basics
2. https://www.youtube.com/watch?v=F1I39fWkzYY : TikZ Trees and Structuring Code
3. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/ : Greedy Algorithm for Huffman Coding
4. https://www.youtube.com/watch?v=co4_ahEDCho : Huffman Coding Example