Huffman Coding

2005001 - Anik Saha
2005006 - Kowshik Saha
2005023 - Jaber Ahmed Deedar

Section: A1
Group: 1

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

March 9, 2024

Contents

List of Figures

List of Tables

1 Introduction

Huffman coding is an efficient method of compressing data without losing information. It provides an efficient, unambiguous code by analyzing the frequencies that certain symbols appear in a message. The algorithm is named after David A. Huffman, who developed it while he was a Sc.D student at MIT..

2 Historical Background

Strategies for efficient compression techniques have long been searched for by scientists and mathematicians. One of the earliest compression techniques dates back to the telegraph era in the mid-19th century. Morse code, developed by Samuel Morse and Alfred Vail in the 1830s and 1840s, encoded text characters into sequences of dots and dashes, effectively compressing textual information for transmission over telegraph wires.

In 1951, Professor Robert M. Fano assigned a term paper on the problem of finding the most efficient binary encoding scheme. In his paper, David A. Huffman developed an algorithm where he assigned shorter codes to the most frequently occurring characters and longer codes to the less frequently occurring characters, thus employing a variable-length encoding system.

David A. Huffman Robert M. Fano Claude Shannon

Figure 1: Pioneers in developing compression techniques

In doing so, Huffman outdid Fano, who alongside Claude Shannon developed a similar method. Huffman's bottom-up approach turned out to be more optimal than the top-down approach of Shannon and Fano.

3 Applications

Huffman coding is used in multifaceted applications where data compression is necessary. Some of those applications are:

- **File Compression:** Huffman coding is widely utilized in file compression algorithms like ZIP, gzip, and zlib. Inferring the approximate distribution of characters over a large file, Huffman's algorithm assigns variable-length codes to encode the file, thereby reducing the size to a significant extent.
- **Text Compression:** Huffman coding finds applications in compressing textual data such as ASCII text and HTML documents. By reducing the size of text files, it optimizes storage space and transmission bandwidth.
- **Image Compression:** In image compression standards like JPEG and PNG, Huffman coding is employed as part of entropy encoding to reduce redundancy in image data. JPEG uses Huffman coding with Discrete Cosine Transform to compress data efficiently while maintaining image quality.
- **Audio Compression:** In audio compression formats like MP3 and AAC, Huffman coding is used to encode frequency deviations. Psychoacoustic modelling helps discard less significant portions of an audio, thereby reducing the size and preserving the perceived quality.
- **Network Communication:** Reducing the size of data packets transmitted over networks is crucial for optimzed data transfer and bandwidth utilization in network communication. Huffman coding reduces the size of data packets, leading to faster transmission times and reduced network congestion.

4 The Problem Definition

Efficient digital communication requires converting any type of information into binary strings. Since resource is limited, we must find out a way of conversion such that the message requires least possible bandwidth to transmit.

A computaionally specific definition of the problem is as follows.

Find an encoding scheme that requires minimum bits on average to encode a message.

However, merely encoding a message is of no use unless it is computationally feasible to decode it in a reasonable amount of time and space. Therefore, the scheme must also have unique decodability. To ensure unique decodability, the encoding must be prefix-free. In other words, if b_1 and b_2 are the binary codes for the characters c_1 and c_2 respectively, it is required that, b_1 is not a prefix of b_2 and b_2 is not a prefix of b_1.

5 A Naive Solution

Say, for instance, we forget the requirement of optimality and design the simplest possible encoding scheme. We assign random binary strings of a fixed length to each of the characters. Note that, assigning distinct codes of the same length automatically ensures unique decodability.

Let's assume, the alphabet consists of the letters A, B, C, D, and E. Since there are 5 characters, we need at least 3 bits to uniquely encode each of them. So, we may randomly assign 3-bit binary strings from **000** to **100**

Doing so, we obtain the following table.

Letter	**Binary Code**
A	000
B	001
C	010
D	011
E	100

Table 1: Binary codes for naive encoding scheme

Now, we may encode any string into a binary message with the help of the table.

While encoding, each of the letters in the string needs 3 bits to encode. Therefore, if the length of a message = n, the expected length of the binary code = $3n$

Average number of bits required per character = $\frac{3n}{n} = 3$

6 Huffman's Algorithm

6.1 The Idea

David A. Huffman, in 1952, came up with a different idea for tackling the problem and it was proven by means of information theory that the output of his algorithm is guaranteed to be optimal.

As we have already seen, the naive algorithm randomly assigns binary codes of same length for each character of the alphabet. Therefore, if there are c characters in the alphabet, the minimum length of each of the codes is $log_2 c$. Huffman, however, looked into the fact that assigning shorter codes for characters that occur more frequently in the language helps decrease the average length of the encoded message, thereby reducing the cost of the transfer.

Based on the observation, Huffman proposed a greedy algorithm which builds a tree in bottom-up order by maintaining a priority queue of characters based on their relative frequencies. This tree, once built, can be used to encode or decode any string of the language.

6.2 Steps

1. Maintain a min-heap of nodes for characters with their relative frequencies as weights.
2. Pop two nodes with minimum weights
3. Merge them into a single node whose weight will be the summation of the previous nodes
4. Push the newly created node back into the min-heap
5. Repeat until the min-heap contains only the root of the tree
6. Use the Huffman tree to encode any message

6.3 Pseudocode

Algorithm 1: Huffman's Algorithm

```
1  Function Huffman(f[1..n])
2      Q ← priority queue containing all characters with their frequencies from f
3      for i ← 1 to n − 1 do
4          allocate a new node z
5          z.left ← x ← Extract-Min(Q)
6          z.right ← y ← Extract-Min(Q)
7          z.freq ← x.freq + y.freq
8          Insert(Q, z)
9      end
10     return the root of the tree
11 end
```

6.4 Properties

- **Greedy Choice Property:** At each step, we choose the two **least probable** symbols and merge them into a single node with a combined probability.
- **Optimal Substructure:** The optimal solution to the entire problem can be constructed from optimal solutions to its subproblems. When we merge two nodes and treat it just like leaf nodes subsequently, we essentially rely on this property.

6.5 The Intuition

The reason why the greedy approach gurantees optimality in the problem lies in the fact that the algorithm always places more-frequent characters closer to the root of the tree reducing the distance of these nodes from root and thereby decreasing the length of the binary encoding for them.

Since we always pop two nodes with minimum frequencies from the priority queue and merge them to form a new node, it is ensured that, less-frequent characters are placed farther from the root and vice versa. Furthermore, as we will explore later, the binary encoding of a character is derived bit by bit while traversing the tree from root to the desired character in a top-down manner. Therefore, the distance of a character from the root in the tree directly correlates with the length of the binary encoding.

The tree structure is necessary for ensuring a prefix-free encoding. It is evident that, the path from the root to a certain leaf can never be the prefix of the path to a different leaf of a binary tree must not have any common prefix with each other because at some internal node of the tree, a branching must have separated them and therefore there must be at least one additional character in both of the paths after the common prefix.

7 The Huffman Tree

In Huffman encoding, symbols are represented by nodes in a binary tree structure which is named the Huffman Tree. It is built using a bottom-up greedy approach.

Nodes of a Huffman Tree are associated with frequency values that represent the sum of the the frequencies of the characters in the corresponding subtree. Edges are used to decide which bit to append to the binary encoding of a character at the corresponding step.

7.1 Properties

- Symbols are represented by **leaves** in a binary tree structure.
- Each symbol in the tree has a **unique path** from the root to its corresponding leaf node.
- The path consists of left (0) and right (1) branches, **uniquely** identifying each symbol.
- Ensures **prefix-free** encoding.

7.2 Construction of Huffman Tree

Given the characters of a language with their relative frequencies, we can construct a Huffman Tree following the steps mentioned in the algorithm. Let us go through a simulation demonstrating the process.

Assume, a certain language consists of 5 characters A, B, C, D, and E. Let us define the relative frequency of a certain character as the number of times it appears in a sample text divided by the total number of characters in the same. Say, for instance, we observe that, in a large text of 1 million characters, A appears 170 thousand times. Therefore, relative frequency of $A = \frac{170000}{1000000} = 0.17$

In a similar manner, we tabulate the relative frequencies of each of the characters.

Letter	**Relative Frequency**
A	0.17
B	0.35
C	0.17
D	0.15
E	0.16

Table 2: Relative frequencies of characters

Step 1: First we push all the characters to the min-heap with their relative frequencies as weights.

Here, the minimum weighted two nodes are $D(0.15)$ and $E(0.16)$.

Step 2: Upon merging $D(0.15)$ and $E(0.16)$, we obtain an internal node with weight 0.31.

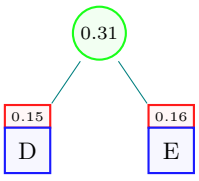

Figure 2: State of the tree after first merge

We push the newly created node back into the min-heap.

Now the minimum weighted two nodes turn out to be $A(0.17)$ and $C(0.17)$.

Step 3: After merging $A(0.17)$ and $C(0.17)$, we obtain a new internal node with weight 0.34.

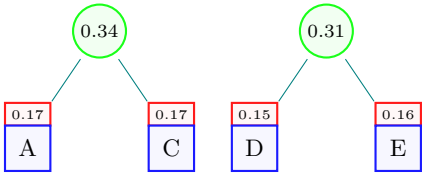

Figure 3: State of the tree after second merge

We again push the newly created internal node back into the min-heap.

After this step, the nodes to be popped from the min-heap are the two internal nodes with weights 0.34 and 0.31. These nodes will now be merged.

Step 4: Merging the internal nodes with weights 0.34 and 0.31, a new internal node with weight 0.65 is created.

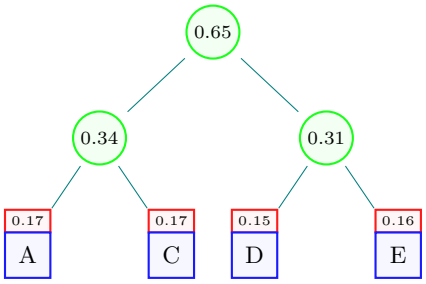

Figure 4: State of the tree after third merge

After we push this node back, the min-heap reaches the following state.

The only two remaining nodes will be merged.

Step 5: After performing the final merge, we obtain the complete huffman tree. Note that, the weight associated with the root of the tree turns out to be 1.00, just as expected.

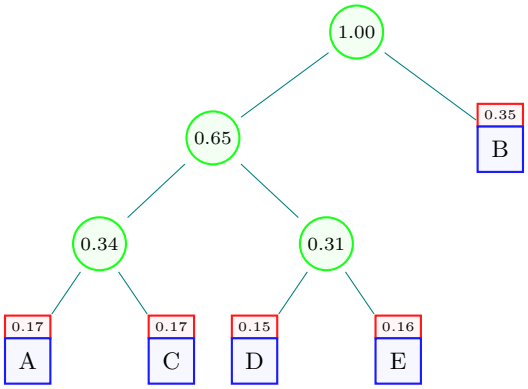

Figure 5: Obtained Huffman Tree

7.3 Interpretation of Huffman Tree

Now let us understand how to interpret binary encodings from the Huffman Tree.

Let's say we want to find the binary encoding of the letter D.

First we need to find a path from the root to the leaf containing D. For every left branch, we append a 0 and for every right branch, we append a 1 to the binary encoding.

The traversal is depicted in the following figure.

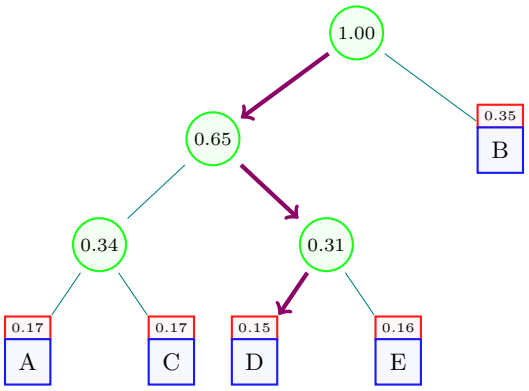

Figure 6: Interpretation of Huffman Tree for the character D

Since, we followed left(0), right(1) and left(0) branch respectively, the binary encoding of $D = 010$

In a similar manner, we tabulate the binary codes for all the characters.

Letter	**Binary Code**
A	000
B	1
C	001
D	010
E	011

Table 3: Binary codes interpreted from the Huffman Tree

7.4 Comparison

In our naive encoding scheme described earlier, the average length of the binary encoding of a character 3. Now let's calculate the same for the above table obtained from the Huffman Tree.

Since expected value is additive, we obtain,

Average Length $= 0.17 \times 3 + 0.35 \times 1 + 0.17 \times 3 + 0.15 \times 3 + 0.16 \times 3 = 2.3$

Now we calculate the compression ratio,

Compression ratio $= \frac{3-2.3}{3} \times 100\% =$ **23.33 %**

8 Encoding

Encoding refers to the process of converting a string into a binary message under a certain scheme or strategy. As we have seen earlier, the Huffman Tree helps us determine the binary codes for each of the characters of a language. Encoding a string is fairly straightforward. We just need to scan the input string from left to right and upon encountering a character, wee need to look up its code and append it to the binary message.

8.1 An Example

For convenience, the previously obtained table is again presented here.

Letter	Binary Code
A	000
B	1
C	001
D	010
E	011

Let us assume we want to encode the string $DECADE$ with Huffman Encoding following the above mentioned table. The following figure demonstrates how we obtain the encoded message.

D E C A D E

010 011 001 000 010 011

Thus we obtain the binary encoding of the string $DECADE$ = **010011001000010011**

8.2 Efficiency of Encoding

According to Information Theory,

$$\text{Entropy}, H = -\sum_{i=1}^{n} p_i \cdot \log_2(p_i)$$

Here p_i denotes the probability of the occurrence of the ith symbol. For our word $DECADE$, we calculate entropy as follows-

$$
\begin{aligned}
H = &- (0.17 \cdot \log_2(0.17) \\
&+ 0.35 \cdot \log_2(0.35) \\
&+ 0.17 \cdot \log_2(0.17) \\
&+ 0.15 \cdot \log_2(0.15) \\
&+ 0.16 \cdot \log_2(0.16)) = 2.23
\end{aligned}
$$

We know,

$$
\text{Efficiency} = \frac{\text{Entropy}}{\text{Number of bits per symbol}} \times 100\%
$$

Encoding of our word "DECADE" is - 010 011 001 000 010 011

Number of bits per symbol = $\frac{18}{6} = 3$

$$
\text{Efficiency} = \frac{2.23}{3} \times 100\% = \mathbf{74.33\%}
$$

9 Decoding

Since we have the binary codes for all the characters, one simple strategy for decoding would be to perform a reverse-lookup for prefixes of the binary message until we find a matching character, and thereby decoding the string letter by letter. This approach, however is neither computationally efficient not conceptually interesting.

Instead, we may just resort to the Huffman Tree. The traversal starts from the root and proceeds in a top-down manner. Just as we interpreted edges of the tree previously, upon having a **0**, we turn **left** and upon having a **1**, we turn **right** until we reach a leaf. When we reach a leaf, the corresponding character is appended to the decoded message and thus the string is decoded character by character.

9.1 An Example

Assume we need to decode the binary message **1011011**. The steps proceed as follows.

Step 1: First we encounter a **1** and upon turning right, we reach a leaf containing **B**. Thus we add **B** to the decoded string.

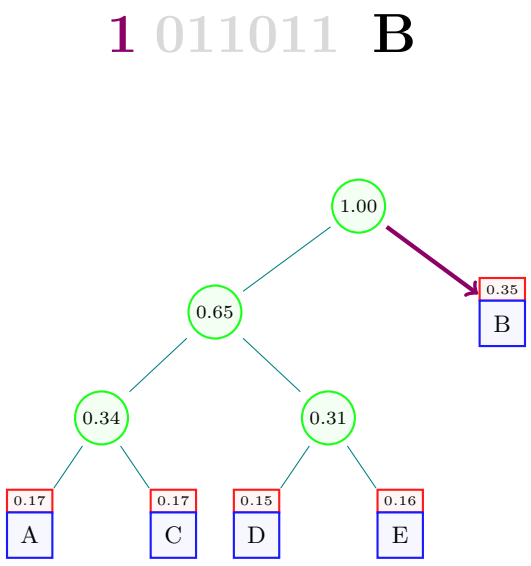

Figure 7: Traversal while decoding the first character

Step 2: Then we keep moving down the tree for three additional bits and reach a reach containing **E**. Thus, our decoded string becomes **BE** so far.

1 011 011 **B E**

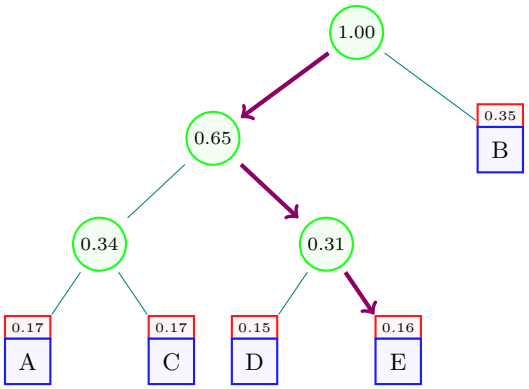

Figure 8: Traversal while decoding the second character

Step 3: The rest of the string is again **011**. So it will also be decoded as **E**.

1011 011 **BE E**

Thus, out decoded string turns out to be **BEE**.

10 Time Complexity Analysis

10.1 Building Frequency Table

To construct the Huffman Tree, we first need the relative frequency values of each of the characters. To do this, we may simply maintain a count-array while scanning a large input text from first to last.

Thus, if the number of symbols in the input = n, Time Complexity = **O(n)**

10.2 Building The Huffman Tree

Assume there are n distinct symbols in the language. Then the running time complexity of the construction of Huffman Tree is determined as follows.

1. **Constructing Initial Heap** Although a single insertion to a heap takes $O(logn)$ time, a heap can be constructed bottom-up in $O(n)$ time from the frequency table obtained earlier. Therefore,
   Time Complexity = **O(n)**
2. **Merging Nodes in the Heap** Every merge requires two extract-min operations on the min-heap. If there are n distinct symbols in the language, $O(n)$ merges are needed in total. Since an extract-min operation takes $O(logn)$ time,
   Time Complexity = **O(nlogn)**

10.3 Generating Huffman Codes

Once we have obtained the Huffman Tree, determining the huffman encodings of the characters needs traversing the tree in a top-down manner. This can be done with any tree-traversal algorithm like BFS or DFS.

Time Complexity = **O(n)**

10.4 Encoding a Message

While scanning the input message from left to right, for each character, we need to append the corresponding binary code to the encoded binary message. Therefore, if the length of the input text is n and the average length of the encoding of a character is k,

Time Complexity = **O(nk)**

10.5 Decoding a Message

Once the Huffman Tree has been built, decoding only requires traversing the tree one edge per input bit. Thus if the encoded binary string has n bits,

Time Complexity = **O(n)**

11 Conclusion

In conclusion, Huffman coding has been a fundamental stepping stone in the journey of data compression. It revolutionized the field by providing efficient and effective compression techniques based on symbol frequencies. Despite the emergence of more sophisticated compression algorithms, Huffman coding remains widely used in various applications.