Huffman Coding

Introduction

Huffman Coding is a *lossless data compression algorithm*.
Huffman Coding is a data compression algorithm used to make files smaller without losing any information.

It's based on a simple but powerful idea: not all characters appear equally often, so we can save space by giving shorter codes to common symbols and longer codes to rare ones. This makes the encoded message smaller without losing any information.

It is widely used in ZIP, JPEG, MP3, and other formats that require efficient data storage or transmission.

Main Idea

Imagine you have a text message where the letter "e" appears very often, while "z" appears only once.
Instead of using the same number of bits for every letter (like in ASCII, where every character uses 8 bits), Huffman Coding gives:

- shorter binary codes to frequent letters (like "e" → 10)
- longer binary codes to rare letters (like "z" → 110011)

This makes the total size of the message smaller overall.

How It Works

1. Count the frequency of each symbol in the data.
   Example: A:5, B:9, C:12, D:13, E:16, F:45

2. Build a binary tree:
    a. Combine the two least frequent symbols into a small tree.
    b. Repeat until there's one big tree containing all symbols.

3. Assign binary codes:
    a. Going left = 0, right = 1 in the tree.
    b. The path from root to each symbol becomes its unique binary code.
4. Encode the message by replacing each symbol with its code.

Example:

Text: “ABACA”
Frequencies:
A = 3, B = 1, C = 1

The Huffman codes might look like:
A = 0, B = 10, C = 11

Encoded message: A B A C A → 0 10 0 11 0 = 010110

That’s 6 bits instead of 15 bits, if each letter had 3 bits.

Why It’s Useful

- Reduces file sizes for text, images, and videos.
- Used in many formats: ZIP, JPEG, MP3.
- It’s lossless, meaning when you decode, you get exactly the original data back.

Mini project (google colab):

1. Input Text

What we do: The user enters or defines a text string, for example:

```
text = "hello huffman"
```

Why: We need some data to compress - this text will be analyzed to find character frequencies and to build Huffman codes.

2. Count Character Frequency

What we do: Count how many times each character appears in the text.
Example: h: 2 e: 1 l: 2 o: 1 : 1 u: 1 f: 2 m: 1 a: 1 n: 1

How in Python:

```
from collections import Counter

freq = Counter(text)
```

Why: The more often a symbol appears, the shorter its Huffman code will be. This is the core idea behind the algorithm.

3. Build Huffman Tree

What we do: Create a binary tree, where:

- The leaves are symbols (characters),
- The weight of each node is the symbol's frequency,
- We repeatedly combine the two smallest nodes into a new one.

How: We use a priority queue (heapq), which always returns the smallest element.

Algorithm:

1. Each character → a separate node with its frequency.
2. Take the two nodes with the smallest frequencies and combine them.
3. Repeat until only one node (the root) remains.

Why: This way, frequent symbols end up closer to the root → they get shorter binary codes.

4. Assign Binary Codes

What we do: Traverse the tree:

- Moving left = 0
- Moving right = 1

The path from the root to each leaf gives a unique binary code.

Example:

Symbol	Code
h	00
e	010
l	11
o	011
...	...

Why: Each symbol now has its own unique binary code, and no code is the prefix of another which ensures correct decoding.

5. Encode the Text

What we do: Replace each character in the text with its binary code.
Example: text = "hello" , encoded = "00101111011"

Why: This is the compressed version of the text. It's shorter than the original, especially when some letters repeat often.

6. Calculate Size Before and After

What we do??? Compare:

- Original size in bits → for example, 8 bits × number of characters
- Compressed size → number of bits in the encoded string

Example: Before: 120 bits After: 70 bits Saved: 41.6%

Why: This shows how effective Huffman Coding is at reducing data size.

How It Can Connect to AI

In the AI phase, Huffman Coding could be applied to:

- Compress training data (e.g., text datasets).
- Optimize memory usage for storing weights or embeddings.
- Preprocess text before sending it to an AI model.

It's not AI itself, but it's a supporting algorithm that helps AI systems run faster and handle data more efficiently.