Huffman Encoding: Greedy Algorithm

Varun Kumar

July 8, 2025

Contents

1 Introduction

Huffman Encoding is a lossless data compression algorithm that assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. It minimizes the total number of bits used for encoding, based on character frequency.

Key Concept: Use a greedy approach by always combining the two least frequent characters to build the optimal prefix code tree.

2 Problem Statement

Given a set of characters along with their frequencies, build a binary tree such that:

- Each leaf node represents a character.
- The path from root to leaf represents its binary code.
- The total cost (weighted path length) is minimized.

3 Approach: Greedy Algorithm using Min-Heap

- Insert all characters and frequencies into a min-heap.
- While more than one node remains:
  - Extract two nodes with the lowest frequencies.
  - Create a new internal node with these two as children.
  - Frequency of new node = sum of both.
  - Insert the new node back into the heap.
- The remaining node is the root of the Huffman Tree.

3.1 Example

Characters and Frequencies: A:5, B:9, C:12, D:13, E:16, F:45

```
Step 1: Merge A(5) + B(9) = 14
Step 2: Merge C(12) + D(13) = 25
Step 3: Merge 14 + E(16) = 30
Step 4: Merge 25 + 30 = 55
Step 5: Merge 45 + 55 = 100
```

Total Cost (Weighted Path Length) = 224

3.2 Prefix Property

Huffman codes satisfy the prefix property — no code is a prefix of another. This ensures unambiguous decoding.

3.3 Why Greedy Works

- **Greedy choice property:** Combining lowest frequency nodes locally reduces the global cost.
- **Optimal substructure:** Huffman's solution is built from smaller optimal solutions.

3.4 GATE 2006 Huffman Coding Problem

Question

The characters a to h have frequencies based on the first 8 Fibonacci numbers as follows:

`a: 1, b: 1, c: 2, d: 3, e: 5, f: 8, g: 13, h: 21`

A Huffman code is used to represent the characters. What is the sequence of characters corresponding to the following code?

`110111100111010`

Options:

A. `fdheg`

B. `ecgdf`

C. `dchfg`

D. `fehdg`

Solution

First, we construct the Huffman Tree using the frequencies:

`a:1, b:1, c:2, d:3, e:5, f:8, g:13, h:21`

Steps:

1. Combine a(1) + b(1) $\rightarrow$ [ab]:2
2. Combine + c(2) $\rightarrow$ [abc]:4
3. Combine d(3) + $\rightarrow$ [abcd]:7
4. Combine e(5) + $\rightarrow$ [abcde]:12
5. Combine f(8) + $\rightarrow$ [abcdef]:20
6. Combine g(13) + $\rightarrow$ [abcdefg]:33
7. Combine h(21) + $\rightarrow$ root = 54

Now, assign codes:

- h = 0

abcdefg = 1

  - g = 10

abcdef = 11

    - f = 110

abcde = 111

      - e = 1110

abcd = 1111

      - d = 11110

abc = 11111

      - c = 111110

ab = 111111

      - a = 1111110
      - b = 1111111

Decoding the string `110111100111010`:

1. 110 → f
2. 11110 → d
3. 0 → h
4. 1110 → e
5. 10 → g

Decoded String: `fdheg`

Correct Answer:

$\boxed{\textbf{A. fdheg}}$

3.5 GATE 2017 Huffman Coding Problem

Question

A message is made up entirely of characters from the set $X = \{P, Q, R, S, T\}$. The table of probabilities for each of the characters is shown below:

Character	**Probability**
P	0.22
Q	0.34
R	0.17
S	0.19
T	0.08
Total	**1.00**

If a message of 100 characters over X is encoded using Huffman coding, then the expected length of the encoded message in bits is ________.

Solution

We follow the standard Huffman coding algorithm:

1. List the characters with their probabilities.
2. Construct a min-heap and combine the two lowest probabilities repeatedly until one tree remains.
3. Assign codes based on tree depth and compute expected length.

Step 1: Sort and Build the Huffman Tree

- Initial Frequencies:
  - $T = 0.08$
  - $R = 0.17$
  - $S = 0.19$
  - $P = 0.22$

- $Q = 0.34$

- Combine $T(0.08) + R(0.17) = 0.25$
- Combine $S(0.19) + P(0.22) = 0.41$
- Combine $0.25 + Q(0.34) = 0.59$
- Combine $0.41 + 0.59 = 1.00$

Step 2: Assign Huffman Codes

- Let's assume the following encoding from the tree:
  - $Q = 11$ (length = 2)
  - $T = 000$ (length = 3)
  - $R = 001$ (length = 3)
  - $S = 10$ (length = 2)
  - $P = 01$ (length = 2)

Step 3: Expected Length

$$E(L) = \sum(\text{probability} \times \text{code length})$$
$$= (0.22 \times 2) + (0.34 \times 2) + (0.17 \times 3) + (0.19 \times 2) + (0.08 \times 3)$$
$$= 0.44 + 0.68 + 0.51 + 0.38 + 0.24 = 2.25 \text{ bits per character}$$

Step 4: Total Bits for 100 Characters

$$\text{Expected Total Length} = 100 \times 2.25 = \boxed{225} \text{ bits}$$

Final Answer:

$\boxed{\textbf{225 bits}}$

4 Theoretical Logic Behind Huffman Encoding Code Implementation

Problem Insight

By assigning shorter codes to frequent characters and longer codes to infrequent ones, we minimize the total number of bits required for encoding.

Key Ideas

- Treat characters and frequencies as leaf nodes.
- Build tree bottom-up using a priority queue (min-heap).
- Traverse the tree to get prefix codes.

Time and Space Complexity

- **Time:** $O(n \log n)$
- **Space:** $O(n)$ for storing the heap and tree

4.1 Algorithm and Pseudocode

Algorithm 1 Huffman Encoding

```
procedure HUFFMANENCODE(charFreq)
    Create a min-heap H from all characters with their frequencies
    while H.size > 1 do
        a ← H.extractMin()
        b ← H.extractMin()
        Create new node with frequency a.freq + b.freq
        Set a and b as children of new node
        Insert new node into H
    end while
    return Root of the Huffman Tree
end procedure
```

4.2 Python Implementation

```python
import heapq
from collections import defaultdict

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

def huffman_encoding(char_freq):
    heap = [Node(c, f) for c, f in char_freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

def generate_codes(node, prefix="", code_map={}):
    if node:
        if node.char:
            code_map[node.char] = prefix
        generate_codes(node.left, prefix + "0", code_map)
        generate_codes(node.right, prefix + "1", code_map)
    return code_map

# Example usage
freq = {'A':5, 'B':9, 'C':12, 'D':13, 'E':16, 'F':45}
root = huffman_encoding(freq)
codes = generate_codes(root)
for char, code in codes.items():
    print(f"{char}: {code}")
```

Listing 1: Huffman Encoding in Python

4.3 C++ Implementation

```cpp
#include <iostream>
#include <queue>
#include <unordered_map>
using namespace std;

struct Node {
    char ch;
    int freq;
    Node *left, *right;
    Node(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
};

struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->freq > b->freq;
    }
};

void generateCodes(Node* root, string code, unordered_map<char, string>& codes) {
    if (!root) return;
    if (root->ch) codes[root->ch] = code;
    generateCodes(root->left, code + "0", codes);
    generateCodes(root->right, code + "1", codes);
}

int main() {
    unordered_map<char, int> freq = {{'A',5}, {'B',9}, {'C',12}, {'D',13}, {'E',16}, {'F',45}};
    priority_queue<Node*, vector<Node*>, Compare> pq;

    for (auto& pair : freq)
        pq.push(new Node(pair.first, pair.second));

    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();
        Node* merged = new Node('\0', left->freq + right->freq);
        merged->left = left;
        merged->right = right;
        pq.push(merged);
    }
```

```

    unordered_map<char, string> codes;
    generateCodes(pq.top(), "", codes);

    for (auto& pair : codes)
        cout << pair.first << ": " << pair.second << endl;

    return 0;
}
```

Listing 2: Huffman Encoding in C++

5 Applications

- **File Compression:** ZIP, GZIP, and other archival formats.
- **Multimedia Compression:** JPEG, MP3, MPEG.
- **Data Transmission:** Efficient coding over networks.

6 Conclusion

Huffman Encoding is an efficient and optimal solution for data compression where symbols with higher frequencies are assigned shorter binary codes. It leverages a greedy strategy and prefix properties to achieve lossless and minimal-bit encoding.