Huffman Coding: A Greedy Approach

Sumanth

June 18, 2025

Abstract

Huffman Coding is a **greedy algorithm** for lossless data compression, assigning variable-length prefix-free binary codes to characters based on their frequencies. By constructing an optimal Huffman Tree, it ensures shorter codes for more frequent characters. This document explores the algorithm's formulation, pseudocode, complexity analysis, practical applications, and implementation considerations, including encoding, decoding, and optimizations using a min-heap. Limitations and extensions are also discussed.

1 Introduction

Huffman Coding is a **lossless data compression** technique that generates variable-length binary codes for characters, with shorter codes assigned to more frequent characters. Using a greedy approach, it builds a binary tree (Huffman Tree) to produce **prefix-free codes**, ensuring no code is a prefix of another. This algorithm is foundational in compression formats like ZIP, JPEG, and MP3, optimizing storage and transmission efficiency.

2 Problem Formulation

Given a set of n unique characters and their frequencies $\{f_1, f_2, \ldots, f_n\}$, construct an optimal prefix-free binary code that minimizes the expected length of encoded messages. The output is a set of binary codes for each character, represented by paths in a Huffman Tree.

2.1 Key Idea

The algorithm greedily constructs a binary tree by repeatedly combining the two least frequent nodes (characters or subtrees) into a new node with their combined frequency. Codes are generated by traversing the tree, assigning `0` for left edges and `1` for right edges.

3 Huffman Coding Algorithm

The algorithm uses a min-heap to efficiently select nodes with the lowest frequencies. Below is the pseudocode:

3.1 Encoding and Decoding

- **Encoding**: Replace each character in the input string with its Huffman code, producing a compressed bitstream.

Algorithm 1 Huffman Coding Algorithm

Input: Set of n characters with frequencies $\{f_1, f_2, \ldots, f_n\}$
Output: Huffman codes for each character
Initialize min-heap Q with n leaf nodes, each containing a character and its frequency
for $i = 1$ to $n-1$ **do**
    $left \leftarrow Q$.extractMin()
    $right \leftarrow Q$.extractMin()
    Create new node z with z.freq $= left$.freq $+ right$.freq
    Set z.left $\leftarrow left$, z.right $\leftarrow right$
    Insert z into Q
end for
$root \leftarrow Q$.extractMin() ▷ Root of Huffman Tree
Procedure GenerateCodes($node, code$):
if $node$ is a leaf **then**
    Output $node$.char with $code$
else
    GenerateCodes($node$.left, $code$ + "0")
    GenerateCodes($node$.right, $code$ + "1")
end if
GenerateCodes($root$, "")
Return Huffman codes

- **Decoding**: Traverse the Huffman Tree from the root using the bitstream. On each `0`, move left; on `1`, move right. Output the character when reaching a leaf and reset to the root.

4 Complexity Analysis

- **Time Complexity**:
  - Building the Huffman Tree: $O(n \log n)$, as each heap operation (insert, extract-min) is $O(\log n)$, and there are $O(n)$ operations.
  - Encoding: $O(n)$, where n is the length of the input string, assuming codes are pre-computed.
  - Decoding: $O(n)$, where n is the length of the encoded bitstream.
- **Space Complexity**: $O(n)$ for the min-heap, tree nodes, and code storage, where n is the number of unique characters.

5 Practical Applications

Huffman Coding is used in:

- **Data Compression**: Tools like ZIP and GZIP for file compression.
- **Image Compression**: JPEG for reducing image file sizes.
- **Audio Compression**: MP3 for efficient audio storage.

- **Network Optimization**: Minimizing data transmission in communication protocols.

6 Example

Consider a string with characters and frequencies:

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

The Huffman Tree is constructed as follows:

- Combine nodes for `c` (1) and `d` (1): new node with frequency 2.
- Combine new node (2) and `e` (1): new node with frequency 3.
- Combine `b` (2) and new node (3): new node with frequency 5.
- Combine `a` (5) and new node (5): root with frequency 10.

Generated codes:

Character	a	b	c	d	e
Code	0	10	110	1110	1111

For input string `aab`, the encoded output is `0010` (0 for `a`, 0 for `a`, 10 for `b`).

7 Limitations and Extensions

- **Limitations**:
  - Ineffective for small files or uniform frequency distributions, as overhead (tree storage) may outweigh compression benefits.
  - Requires frequency analysis beforehand, adding preprocessing time.
- **Extensions**:
  - **Adaptive Huffman Coding**: Updates frequencies dynamically for streaming data.
  - **Canonical Huffman Codes**: Optimizes code storage for faster decoding.
  - **Hybrid Compression**: Combines with other techniques (e.g., Run-Length Encoding) for better compression ratios.

8 Implementation Considerations

- **Min-Heap**: Use a priority queue (e.g., `std::priority`$_q$*ueueinC++*)*forefficientnodeselection*,*ac*
- **`Tree Storage:`** `Store the Huffman Tree or codes in a compact form to minimize overhead in compressed files.`

- **Bit-Level I/O**: Handle encoding/decoding at the bit level to ensure true compression, using bit manipulation or libraries.
- **Frequency Analysis**: For large inputs, precompute frequencies accurately to avoid recomputation.
- **Numeric Stability**: Ensure frequency sums do not overflow for large inputs by using appropriate data types (e.g., long long).

9 References

- GeeksForGeeks: Huffman Coding
- Wikipedia: Huffman Coding
- Cormen, T. H., et al., *Introduction to Algorithms*, Chapter on Greedy Algorithms.