Huffman Coding (Data Compression)

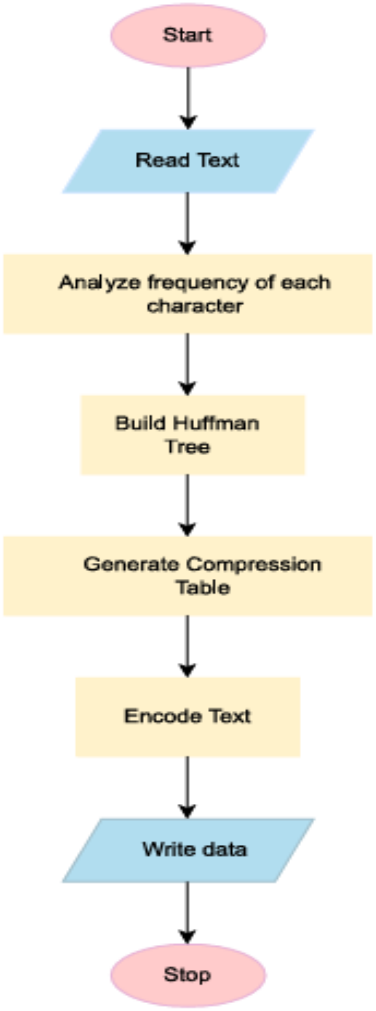
Overview

This project implements a **Huffman Coding based text compressor and decompressor**. Huffman Coding is a popular lossless data compression technique that assigns **variable-length binary codes** to characters based on their frequency of occurrence. Characters that appear more frequently are assigned **shorter codes**, leading to compression.

The system offers three main functionalities:

- Data compression
- Data decompression
- Character-based searching (both frequency and Huffman code lookup)

Algorithm



Focus

The project focuses on implementing an efficient text compression and decompression system using the Huffman coding algorithm. The primary objective is to reduce the size of user-provided text input while ensuring lossless compression, making it possible to perfectly reconstruct the original text.

This project was developed not only to demonstrate practical A**lgorithms** concepts but also to analyze algorithmic efficiency, aligning with the objectives of the **Analysis of Algorithm** course.

Key Features

The system offers three main functionalities:

1. **Text Compression (Encoding)**
    - Calculates frequency of each character
    - Builds Huffman tree
    - Generates binary codes
    - Encodes text into binary string
2. **Text Decompression (Decoding)**
    - Uses Huffman tree to decode binary string
    - Reconstructs original text
3. **Character-Based Searching**
    - Search frequency of a specific character
    - Search Huffman code of a specific character

Algorithms Applied

- Huffman Coding Algorithm

  Frequency Counting*:*

  Counts occurrences of characters in text

  Tree Construction*:*

  Uses min-heap (priority queue) to build tree

  Code Generation*:*

  Recursively generates binary codes for character.

- Search Algorithm

  Simple *unordered_map* search to find frequency and code of a character.

Menu Options

In this project, the algorithm provides different menu options to understand the logic and functionality behind this algorithm.

```
         *********    HUFFMAN CODING MENU   **********


1. Show Character Frequencies
2. Show Huffman Codes
3. Show Original vs Compressed Size
4. Show Encoded String
5. Show Decoded String
6. Find Frequency and Code of Specific Character
7. Exit
```

Example Usage

1) **To compress a data**

```
Choose an option: 4

--- Encoded String ---
00111001111101000011011000111010111001101001011
```

2) **To decompress a data**

```
Choose an option: 5

--- Decoded String ---
IlovePakistan
```

3) **To find character**

```
Choose an option: 6

Enter character to search for: a

Frequency of 'a': 2
Huffman Code of 'a': 100
```

Code Structure

- **Node Class**:

  Represents node of Huffman Tree (stores char, freq, left, right)

- **Compare Class**:

  Comparator for priority queue (min-heap)

- **buildCodes() Function**:

  Recursively generates Huffman codes

- **freeTree() Function**:

  Deletes Huffman tree and frees memory

- **Main() Function**:

  Manages text input, builds Huffman Tree, handles user menu

Main Data Structures Used

- unordered_map<char, int>: Stores character frequencies.
- unordered_map<char, string>: Stores Huffman codes for each character.
- priority_queue: Manages node ordering during tree construction.
- Binary Tree: Huffman Tree (Stores structure for encoding/decoding).

Performance Analysis

Time Complexity

1. Frequency Counting: **O(n)**

   - You go through the entire input string once to count character frequencies.
   - One pass = **O(n)**

2. Building Priority Queue: **O(k log k)**

   - You insert k items into a min-heap.
   - Each insert takes log k time.
   - Total = k × log k = O(k log k)

3. Tree Construction: **O(k log k)**

   - Initially K nodes
   - Two nodes popped: O(log k) each
   - One new node pushed: O(log k)
   - Total = O(k log k)

4. Code Generation (DFS): **O(k)**

   Traverse the Huffman tree using **DFS** to assign a binary code to each of the **k unique characters**.

5. Encoding: **O(n)**

   Replace each character in the input text (of length **n**) with its Huffman code using the map.

6. Decoding: **O(n)**

   Traverse the encoded bit string from left to right; follow the Huffman tree until a leaf is reached

7. Searching (map lookup): **O(1)** average case

   Looking up any character's frequency or Huffman code in a map takes **constant time** on average.

Space Complexity

- Frequency Map: **O(k)**
- Huffman Codes Map: **O(k)**
- Huffman Tree: **O(k)** nodes
- Encoded String: Up to **O(n)** bits

Where **n** = length of input text, **k** = number of unique characters

Overall Performance:

Time Complexity: $O(n + k \log k) \longrightarrow O(k \log k)$

Space Complexity: $O(n + k) \longrightarrow O(m)$

Why Huffman Coding is Efficient?

- **Shorter codes for frequent characters:** reduces total bits required.
- **Prefix-free property**: no ambiguity in decoding.
- **Optimal for character frequency distribution**: achieves near-best compression for given text.
- **Lossless compression**: original text is perfectly reconstructed

Conclusion

This project effectively demonstrates how **Huffman Coding** compresses text by leveraging character frequency patterns. It applies **tree-based algorithms**, **priority queues**, and **hash maps** to achieve efficient compression, fast encoding/decoding, and user-friendly interaction. The project highlights important algorithms concepts in a practical application

Limitations

This system has following limitations:

- ✓ Huffman coding may not provide significant compression on very small or **uniform datasets** where character frequencies are nearly equal.
- ✓ The system works only for text. It cannot compress **images, audio, or other file** types.
- ✓ The project is console-based. It does not have a **graphical interface** for easier use.