3rd Semester PROJECT

DSA

Syed Jaffar Raza Kazmi (57375)
Zain Ali (57113)

Faculty of Computing
Riphah International University

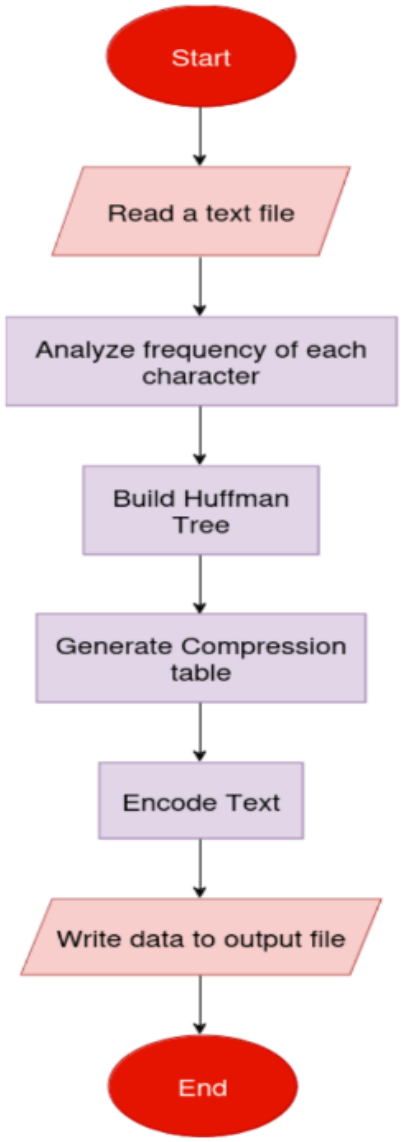
Table of Contents

Text File Compressor

Overview

This project implements a text file compression system using **Huffman coding**, a popular data compression technique that creates **variable-length codes** for characters based on their **frequency of occurrence**. More frequent characters get shorter codes, resulting in overall data compression. The system offers three main functionalities:

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



Focus

The Text File Compressor project focuses on implementing an efficient file compression system using Huffman coding algorithm. The primary objective is to **reduce the size of text** files while ensuring lossless compression, making it possible to perfectly reconstruct the original file. The project demonstrates practical application of various data structures and algorithms learned throughout the DSA Course Curriculum.

Key Features

1. Compression

i. **Frequency Analysis:**

Scans input file to count character occurrences

ii. **Huffman Tree Generation:**

Creates a binary tree based on frequencies

iii. **Code Generation:**

Produces variable-length binary codes for each character

iv. **Binary Output:**

Writes compressed data in binary format with header information

2. Decompression

i. **Header Reading**:

Extracts Huffman coding table from compressed file

ii. **Binary Decoding**:

Converts binary data back to original characters

iii. **File Recreation**:

Reconstructs original file from compressed data

3. Search Functionality

i. **Character Frequency Search**:

Linear search to find specific character frequencies

ii. **Huffman Code Search**:

Binary search implementation to find codes efficiently

iii. **Optimized Sorting**:

Custom QuickSort implementation for code organization

Version Control

Github : Click here

Algorithms Applied

1. Huffman Coding Algorithm

Frequency Counting

Scans the input file once to count character frequencies.

```
map<char, int> countFrequency(string fileName)…
```

Tree Construction

Uses a priority queue to repeatedly combine least frequent nodes
Creates unique codes for each character

```
Node *buildHuffmanTree(priority_queue<Node *, vector<Node *>, Compare> pq)…
```

Code Generation

Recursively traverses the Huffman tree
Assigns binary codes to characters based on their path from root to leaf

```
void generateHuffmanCodes(Node *node, string code, map<char, string> &huffmanCodes)
```

2. Search Algorithms

Linear Search

Used for single character frequency lookups

```
bool findFrequency(string fileName, char ch)…
```

Binary Search

Used for Huffman code lookup

```
string findHuffmanCode(map<char, string> &huffmanCodes, char ch)
```

3. Sorting Algorithms

Quicksort Algorithm

Used to sort Vector containing Huffman Codes for Binary Search.

```
void quickSort(vector<pair<char, string>>& arr, int low, int high) {
```

Menu Options

1. Compress File
- Enter source file name
- Enter destination file name for compressed data

2. Decompress File
- Enter compressed file name
- Enter destination file name for decompressed data

3. Search Character
- Choose between:
    - Finding character frequency
    - Looking up Huffman code
- Enter file name and character to search

4. Exit Program

```
********************Text File Compressor*****************

1. Compress File
2. Decompress File
3. Search Character
4. Exit

Enter your choice:
```

Example Usage

1. **To compress a file**

```
1. Compress File
2. Decompress File
3. Search Character
4. Exit

Enter your choice: 1
Enter the file name to be compressed: input.txt
Enter the file name to store the compressed data: compressed.bin

File compressed successfully!
```

2. **To decompress a file**

```
1. Compress File
2. Decompress File
3. Search Character
4. Exit

Enter your choice: 2
Enter the file name to to be Decompressed: compressed.bin
Enter the file name to store the Decompressed data: output.txt

Decompression complete. Data written to output.txt
```

3. **To search for character information**

```
1. Compress File
2. Decompress File
3. Search Character
4. Exit

Enter your choice: 3

Search Options:
1. Frequency of a character
2. Huffman code of a character
3. Exit
```

Code Structure

Key Classes

1. **Node:**

➔ Represents a node in the Huffman tree
➔ Stores character, its frequency, and pointers to left and right children
➔ Used to build the compression tree structure

2. **Compare:**

➔ Acts as a custom comparator for the priority queue used in building the Huffman tree.
➔ Creates a min-heap structure where lower frequencies have higher priority.
➔ Essential for building the Huffman tree from bottom up.

3. **HuffmanSorter:**

➔ Implements QuickSort algorithm specifically for Huffman codes
➔ Sorts character-code pairs for efficient searching
➔ Used in the binary search implementation for finding specific character codes

Main Data Structures Used

➔ map<char, int>: Stores character frequencies
➔ map<char, string>: Stores Huffman codes for each character
➔ priority_queue: Manages node ordering during tree construction
➔ vector<pair<char, string>>: Used for sorted storage of Huffman codes
➔ Binary Tree

Why Binary Trees?

➔ Variable-Length Coding
➔ Binary choices map naturally to bits (0/1)
➔ No-Prefix Property

Key Functions

- **countFrequency():** Counts character frequencies in input file
- **buildPriorityQueue():** Creates priority queue for Huffman tree construction
- **buildHuffmanTree():** Constructs the Huffman tree
- **generateHuffmanCodes():** Creates Huffman codes for each character
- **compressFile():** Performs file compression
- **decompressData():** Reconstructs original data from compressed file
- **findHuffmanCode():** Searches for specific character codes

Compressed File Structure

1. Number of Huffman codes
2. Huffman coding table (character:code pairs)
3. Compressed binary data

Performance

- Achieves variable compression ratios depending on input text patterns
- Efficient search operations using binary search
- Lossless compression guaranteeing exact data recovery

Acknowledgments

- Based on David Huffman's data compression algorithm
- Inspired by the need for efficient text file compression
- Developed as part of 3rd Semester's DSA project

Known Issues

- Limited to text file compression

Submitted By
Syed Jaffar Raza Kazmi (57375)
Zain Ali (57113)