Huffman Compression Algorithm

Mohamed darwish

202201273

Abstract

This report outlines the implementation of the Huffman compression algorithm, a statistical, entropy-based approach to lossless data compression. Huffman coding assigns shorter binary codes to frequently occurring characters and longer codes to infrequent ones, resulting in efficient data representation. The implementation features a modular design for ease of maintenance and performance evaluation.

Keywords: Huffman coding, entropy encoding, variable-length codes, lossless compression, data structures

---

I. Overview

Huffman coding is a classical algorithm for optimal prefix coding based on character frequency. It constructs a binary tree in which the path from root to leaf determines the code for each character. This report presents a full-featured, modular implementation of the Huffman algorithm designed to compress and decompress text files while maximizing space efficiency.

II. Project Structure

The implementation is divided into independent, reusable modules:

```
Main
├── FileHandler
├── HuffmanEncoder
├── HuffmanDecoder
└── CompressedFile
```

III. Core Components

A. HuffmanEncoder (Compression)

The encoder follows a multi-step process:

- **Frequency Analysis**: Builds a frequency map of characters in the input text.
- **Tree Construction**: Constructs a binary Huffman tree using a priority queue, placing low-frequency characters deeper in the tree.
- **Code Generation**: Traverses the tree to assign prefix-free binary codes to each character.
- **Encoding**: Replaces each character in the input with its binary code and writes the encoded data to a compressed file.

B. HuffmanDecoder (Decompression)

Decompression involves:

- Reading the encoded binary stream and corresponding code table.
- Rebuilding the Huffman tree from the table.
- Decoding the bitstream by traversing the tree.
- Writing the recovered text to an output file.

IV. Compression and Decompression Workflow

A. Compression Workflow

1. Read input text.
2. Construct a character frequency map.

3. Build the Huffman tree.
4. Generate binary codes.
5. Encode text using these codes.
6. Output the binary stream and code table.

B. Decompression Workflow

1. Read compressed binary data and code table.
2. Reconstruct the Huffman tree.
3. Decode binary stream by traversing the tree.
4. Write the original text to output.

C. Code Format

Each character is replaced by a unique binary string, satisfying prefix-free properties. For example:

```
A: 011, B: 10, C: 11, D: 00, E: 010
```

This mapping ensures unambiguous decoding of any concatenated bit sequence.

V. Test Cases

Two representative cases used to validate the encoder and decoder are as follows:

Input Text	Code Mapping	Encoded Bits	Size bits	Entropy
BCCABBDDAECCBBAEDDCC	A:011, B:10, C:11, D:00, E:010	10111101110100000011010 11	120	2.2282
aaaaabbbbccccc	a:0, b:10, c:11	0000010101010111111111 1	93	1.5774

VI. Conclusion

This implementation of Huffman coding demonstrates effective data compression through entropy encoding. The structured modular approach enhances readability and extensibility. Future enhancements could focus on memory optimization for tree storage and support for binary file inputs. Additionally, hybrid models combining Huffman with dictionary-based methods may yield further compression gains in large-scale applications.