Huffman Compression Algorithm

Mohamed darwish

202201273

Abstract

This report presents a comprehensive implementation of the Huffman compression algorithm, supporting both static and adaptive coding techniques. By assigning variable-length binary codes based on symbol frequency, Huffman coding minimizes entropy and reduces file size. The system features modular design, visualizations of tree dynamics, and flexibility for real-time and offline applications. Experimental results confirm the efficiency of both approaches in reducing data redundancy.

Keywords: Huffman Coding, Entropy Compression, Adaptive Huffman, Static Huffman, Binary Trees, Real-Time Compression

---

I. Introduction

Huffman coding is a cornerstone technique in lossless data compression. It exploits the frequency distribution of symbols to generate an optimal prefix code, where more frequent characters are assigned shorter binary codes. This implementation supports two variants:

- **Static Huffman Coding**: Builds the tree once using precomputed frequencies.
- **Adaptive Huffman Coding**: Dynamically adjusts the tree during runtime, eliminating the need for a separate frequency analysis phase.

This dual-mode support provides a versatile framework suitable for both file-based and streaming compression scenarios.

II. System Architecture

The project is built with modularity and visualization in mind, divided into the following components:

```
Main
├── FileHandler
├── HuffmanEncoder / HuffmanDecoder
├── HuffmanTree
└── TreeVisualizer
```

- **Main**: Manages user input, mode selection, and program flow.
- **FileHandler**: Reads and writes both plain text and binary data.
- **HuffmanEncoder / Decoder**: Handles encoding and decoding logic for both static and adaptive modes.
- **HuffmanTree**: Constructs and manages the binary tree, frequency maps, and code tables.
- **TreeVisualizer**: Renders dynamic tree diagrams during encoding.

III. Algorithm Workflow

Compression Workflow

1. Load input text from file.
2. Generate a frequency map (static) or initialize an empty tree (adaptive).
3. Construct or update the Huffman tree.
4. Generate binary codes.
5. Encode and write compressed data.

Decompression Workflow

1. Load encoded binary stream.

2. Rebuild the Huffman tree (static) or adapt it during decoding.
3. Translate the bitstream to original characters.
4. Output the decompressed text.

IV. Adaptive Huffman Approach

Adaptive Huffman coding updates the tree as each character is processed. This approach is ideal for real-time compression where prior statistics are unavailable.

Key features include:

- **NYT Node (Not Yet Transmitted)**: Placeholder for symbols not yet encountered.
- **Dynamic Weight Adjustments**: Nodes update based on character frequency.
- **Tree Rebalancing**: Maintains the sibling property through swaps.
- **Single-Pass Processing**: No initial frequency map required.

V. Tree Visualization

To aid in understanding the compression process, the system graphically renders the Huffman tree:

- **Nodes**: Color-coded to indicate weights.
- **Edges**: Labeled with traversal direction (0/1).
- **Updates**: Visualize tree evolution during adaptive encoding.

Example: Adaptive Huffman Tree after encoding “aab”

```
      W:3
       /  \
    NYT   b(W:1)
          /   \
       a(W:2)
```

VI. Evaluation and Results

Test Case Summary

Input Text	Bits Used	Entropy	Sample Encoding Pattern
mississippi	99	2.52	011011010001000...100
aaaaabbbbccccc	93	1.58	a:0, b:10, c:11

Compression performance improves with increased frequency skew, highlighting the entropy-reduction strengths of Huffman coding.

VII. Static vs. Adaptive Huffman Comparison

Feature	Static Huffman	Adaptive Huffman
Tree Behavior	Pre-built (fixed)	Dynamic (incremental)
Ideal Use Case	Known input stats	Streaming, real-time
Encoding Speed	Faster (once-built)	Slower (ongoing update)
Memory Usage	Higher	Lower (on-the-fly)

Each approach serves different performance and use-case priorities.

VIII. Conclusion and Future Work

The implemented Huffman compression system successfully reduces file sizes while maintaining full data integrity. Its support for both static and adaptive modes, along with real-time visualization, makes it both practical and educational. Future extensions may include:

- Bit-level streaming support
- Integration with hybrid compression pipelines
- Enhanced GUI-based visualization for deeper analysis