File Zipper: Text File Compression Using Huffman Trees

Noor Elshahidi 900230000 & Bassel Shoeib 900231376

1 Abstract

The project report on file compression through huffman coding,it minimizes text files through the building of Huffman trees according to the frequency with which characters occur.It then employs the optimal prefix codes.It compacts files and decompresses them with no need for additional space for the Huffman tree since the tree is stored in the compressed file.Tests indicate that it can substantially minimize english text files in size with no alteration in their original appearance.

Keywords: Huffman coding, data compression,file zipper,prefix codes,lossless compression

2 Introduction

Data compression is increasingly essential in contemporary computing because digital data grows rapidly. It conserves memory and makes data transmission faster through elimination of unnecessary information. Macinkovic et al.(2022) stated "Technique in carrying out data compression is an important point in technologcal developments." Lossy and lossless algorithms are the two principal methods for data compression. In text files, where the data should be reconstructed back,lossles compression should be employed.

David huffman developed Huffman coding in 1952.It is simple data compression that doesnt destroy any information. It provides codes of varying lengths in relation to the frequency at which symbols occur in the data.It provides short codes for symbols that occur frequently and longer codes for those symbols that rarely occur.According to moffat (2019),Huffman coding is an essential tool in computer science as it is simple and useful.

2.1 Problem definition

The primary issue this project addresses is the question of how to compress and expand text files in an intelligent manner without losing any information.Text files contain many repeating pieces of information,with certain characters being used more frequently than others.Traditional encoding schemes such as ascii

utilize the same size for every character regardless of frequency,which is not optimal as per information theory.

Our project implements a file compression system that:

1. analyzes character frequencies in text files
2. Constructs optimal Huffman trees based on these frequencies
3. Generates variable length prefix code for each character
4. compresses files using these codes
5. Allows decompression back to the original text
6. Embeds the Huffman tree within the compressed file, eliminating the need for external tree storage

3 Methodology

The project employs Huffman coding, an algorithm that constructs the best prefix codes on the basis of the frequency with which characters occur in a text file.The basic premise,as detailed in khan and mohsin (2022),is that frequent characters will be given short codes,whereas infrequent characters will be assigned longer codes.

3.1 The huffman algorithm

The Huffman coding process follows these key steps:

- **Frequency counting**
  Character frequencies are tracked in an array of 256 counters.Reading the file in binary mode ensures all byte values are handle
- **Tree construction**
  A custom MinHeap structure supports insertion and extraction of huffmannode pointers.Merging two nodes creates a new internal node with combined frequency until a single root remains.
- **Code generation**
  Recursively traverse the tree: appending '0'for a left edge and '1' for a right edge.Leaf nodes map characters to codes
- **Tree serialization**
  Use a pre order traversal:write a 1 bit followed by the byte if leaf, or a 0 bit then recurse into children.
- **Bit I/O**
  The BitWriter buffers bits into bytes,flushing when full. BitReader reads bits sequentially from bytes.

This approach produces the optimal prefix free codes, such that no code is the prefix of any other code.It will simplify and make the decoding process clearer."Huffman encoding,in particular,is capable of shrinking files to less than 70% of their original file size". according to komal(2024).

3.2 Implementation approach

Our implementation uses object oriented C++ and consists of several integrated components:

1. Read the input file and count the frequency of each byte value
2. build a min heap of nodes keyed by frequency and iteratively merge the two smallest nodes to form the huffman tree.
3. generate a code map by traversing the tree, assigning binary codes to each leaf
4. Write a header containing the original file size and the serialized tree using a pre order traversal, marking leaves and internal nodes
5. Encode the input data by writing corresponding bits for each symbol.
6. For decompression,read the header,reconstruct the tree, and decode bit by bit to recover the original symbols

We store the huffman tree structure in the compressed file.We dont need to store the tree elsewhere.When we decompress the file, we construct the tree from the saved form.

4 Specification of Algorithms Used

4.1 Frequency Counting

The frequency counting algorithm scans the input file once,incrementing counters for each character encountered:

```
void FrequencyCounter::count(const char* filename) {
    std::ifstream in(filename, std::ios::binary);
    if (!in) throw std::runtime_error("Failed␣to␣open␣file␣for␣reading");

    unsigned char ch;
    while (in.read(reinterpret_cast<char*>(&ch), 1)) {
        if (freqTable[ch].frequency == 0) {
            uniqueCount++;
        }
        freqTable[ch].character = ch;
        freqTable[ch].frequency++;
    }
}
```

This approach has O(n) time complexity,where n is the file size in bytes.

4.2 Huffman Tree Construction

The huffman tree is constructed using a min heap priority queue:

1. For each character with non zero frequency,create a leaf node
2. insert all leaf nods into the min heap
3. While the heap contains more than one node:
    - extract the two nodes with lowest frequencies
    - create a new internal node with these as children
    - set the new nodes frequency to the sum of its childrens frequencies
    - snsert the new node back into the heap
4. The last node remaining in the heap is the root of the huffman tree

This algorithm has O(n log n) time complexity,where n is the number of unique characters.

4.3 Code Generation

Codes are generated by traversing the hufman tree from root to leaf:

```
void HuffmanTree::buildCodeMap(HuffmanNode* node, std::string code) {
    if (!node) return;
    if (node->isLeaf() && code.empty()) {
        codeMap[static_cast<unsigned char>(node->data)] = "0";
        return;
    }
    if (node->isLeaf()) {
        codeMap[static_cast<unsigned char>(node->data)] = code;
        return;
    }
    buildCodeMap(node->left, code + "0");
    buildCodeMap(node->right, code + "1");
}
```

this recursive approach assigns a ”0” for left branches and ”1” for right branches, building the code as the tree is traversed

4.4 Tree serialization and deserialization

a critical aspect of our implementation is storing the huffman tree with in the compressed file:

```
void HuffmanTree::serializeTree(BitWriter& writer) {
    std::function<void(HuffmanNode*)> serialize = [&](HuffmanNode* node) {
        if (node->isLeaf()) {
            writer.writeBit(true);
```

```
            writer.writeByte(node->data);
        } else {
            writer.writeBit(false);
            serialize(node->left);
            serialize(node->right);
        }
    };
    serialize(root);
}
```

It employs pre order serialization. It tags the leaf nodes with the bit "1" and then the character value.It tags the internal nodes with the bit "0".It gets reversed during decompression to reconstruct the tree.

5 Data specifications

5.1 Input data

the program can handle any ascii text file,but its effectiveness can vary depending on the characters.The version I currently use has been tried with:

1. english text files (essays,articles,books)
2. Source code files (C++, java, python)
3. Data files (csv,json, xml)

5.2 Compressed file format

The compressed file has the following structure:

1. **header** (4 bytes):Original file size in bytes
2. **serialized tree**: Variable length representation of the huffman tree
3. **compressed data**:The encoded bit stream

This format ensures that all information needed for decompression is contained within a single file.

6 Experimental results

6.1 Compression performance

we wanted to test how good our huffman coding is, and we tried it on varying text files of varying sizes.The findings indicate that our approach is efficient in decreasing the sizes of the files without losing any information.

Table 1: Compression Results for Test Files

File	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio
file1.txt	768	489	0.637
file2.txt	856	548	0.640
file3.txt	1594	890	0.558

the compression ratio is $\frac{\text{Compressed Size}}{\text{Original Size}}$, and smaller values have better compression.from Table 1, our implementation achieved compression ratios of 55.8% to 64.0%, i.e.,the compressed files were 36% to 44% smaller than the originals.

Several observations can be made from these results:

- the compression efficiency varies depending on the file content and its entropy
- larger files (like file3.txt)tend to achieve better compression ratios due to more stable symbol distributions
- The results are consistent with theoretical expectations for huffman coding on english text

These test results confirm that our huffman coding implementation is successfully compressing files without the loss of any of the original information.That compression ratios vary between files shows how the algorithms performance depends on the specific characteristics of the input data

Based on similar implementations and a komal (2024) study,huffman encoding typicaly compresses text files between 30-70%. Comparative experimental studies on huffman coding versus arithmetic coding by hamza et al.(2019) revealed that huffman achieved good compression for wav audio files, reflecting its suitability for diverse data formats.

The performance of huffman coding is very dependent upon the character frequency distribution. Files with highly skewed distributions(where some characters occur much more frequently than other characters) give rise to greater compression

6.2 Comparison with other algorithms

As praja et al (2022) state, in a comparison of huffman coding and lempel ziv Welch (lzw) algorithm, "LZW algorithm was superior to huffmans algorithm in .txt file type compression and .csv,the average space savings produced were 63.85% and 77.56%".It can be seen that although huffman coding is efficient, dictionary based methods such as lzw can achieve greater compression for specific file formats.

6.3 Processing time

The algorithm's time complexity is dominated by the creation of the initial tree (O(n log n)), hence it is efficient even for large files.Compression and decom-

pression both have linear time complexity in terms of input file size

7 Analysis and Critique

7.1 Strengths

1. **Optimal prefix codes**:huffman coding guarantees optimal prefix-free codes for a given character frequency distribution
2. **Lossless compression**: The algorithm preserves all information, making it suitable for text files where data integrity is essential
3. **Self contained implementation**:By embedding the tree structure in the compressed file, our implementation eliminates the need for external tree storage, making file sharing simpler
4. **Efficient implementation**:The use of a min-heap for tree construction ensures O(n log n) time complexity

7.2 Limitations

1. **Static encoding**:Our implementation uses static huffman coding, which analyzes the entire file before compression.This requires two passes through the data and doesnt adapt to changing character distributions within a file.
2. **Compression ratio ceiling**: As noted by moffat (2019), huffman coding is constrained by the need to use an integral number of bits per symbol, limiting compression efficiency.
3. **Suboptimal for smal files**:The overhead of storing the huffman tree can outweigh the compression benefits for very small text files
4. **Memory usage**: Building and storing the huffman tree and code table requires memory proportional to the number of unique characters

7.3 Potential improvements

1. **Adaptive huffman coding**:Implementing an adaptive approach that updates the tree as characters are processed would eliminate the need for two passes and potentially improve compression for files with varying character distributions.
2. **Combining with other techniques**: As suggested by comparative studies, combining Huffman coding with dictionary based methods could improve compression ratios.
3. **Length limited codes**:Implementing length limited huffman codes could improve decoding speed at a slight cost to compression efficiency.

4. **Parallel processing**: For very large files, parallel processing of different file segments could improve performance.

8 Conclusions

This project is a successful implementation of a file compression system based on Huffman coding.It is effective at compressing text files and returns them perfectly intact when decompressed. The huffman tree is included in the implementation within the compressed file.This eliminates the requirement for external tree storage, and the system is self sufficient.

Our experiments demonstrate that huffman coding performs effectively at text compression.Comparison with other techniques suggests that its combination with dictionary techniques may enhance compression.Future research may be directed towards experiments with such combined techniques and variations of huffman coding

The project reflects the good and the bad points of huffman coding as discussed by moffat (2019):it remains very efficient even if it is outdated, while newer algorithms perform better in some environments.Nevertheless,its efficiency,simplicity,and being the best available option lead it to being an excelent choice for the greater part of text compression applications.

9 Acknowledgements

We thank Dr Ashraf and Dr Dina for their guidance, and feedback during development and testingfor their valuable feedback during the implementation and testing phases.

10 References

D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," Proceedings of the IRE, vol. 40, no. 9, pp. 1098–1101, 1952.

GeeksforGeeks. (2025, April 22). Huffman Coding — Greedy algo3. GeeksforGeeks. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. MIT Press, 2009.

Hamza, S. A., Hasnaoui, M., Radi, B., & Benahmed, M. (2019). Comparison of Lossless Compression Schemes for WAV Audio Data 16-Bit Between Huffman and Coding Arithmetic.

Khan, R. M., & Mohsin, H. M. (2022). Analisa Kompresi File Teks Menggunakan Algortima Huffman.

Komal, M. (2024). Huffman Tree Compression and Lempel-Ziv Coding Using Java.

Mačinković, D., Gajinović, J., & Stracenski, R. (2022). Application of Huffman Algorithm and Unary Codes for Text File Compression.

Moffat, A. (2019). Huffman Coding..

Praja, Y. A., Karhendri, D. H., & Purwari, A. (2022). Comparison of Huffman Algorithm and Lempel Ziv Welch Algorithm in Text File Compression.

Human, D. S., Reaves, C. B., & Smith, J. R. (2008). A Generic Top-Down Dynamic-Programming Approach to Prefix-Free Coding. *arXiv*.

Yeung, D. Y. (2007). Reserved-Length Prefix Coding. *arXiv*.

11 Appendix: Listing of Implementation Code

The complete source code for this project is in our github repo:

https://github.com/Bassel-Shoeib/File-zipper.git