Term Project: File Compression using Huffman Coding in C++

Yahia Kilany

Department of Computer Science and Engineering, AUC

CSCE2211:Applied Data Structures, Section:02

Dr.Amr Gonied

May 17, 2024

Abstract

This project implements Huffman Coding in C++ for file compression and decompression. Huffman Coding is an efficient, lossless data compression algorithm. This project demonstrates the implementation of the algorithm, including the creation of the Huffman Tree, the generation of Huffman codes, and the processes of compressing and decompressing text files. The program ensures efficient storage and retrieval of data, reducing the size of input files and enabling decompression back to the original content.

Keywords: Huffman coding, data compression, min-heap, binary tree, lossless compression, frequency map

Introduction

Huffman Coding is a widely used method of lossless data compression. It is particularly effective for reducing the size of files without losing any information(GeeksforGeeks, 2023). This project aims to implement Huffman Coding in C++ to compress and decompress text files, demonstrating the algorithm's effectiveness and efficiency.

Problem Definition

With the increasing amount of data generated and stored, efficient data compression techniques are crucial. Huffman Coding addresses the need for effective data compression by assigning shorter codes to more frequently occurring characters and longer codes to less frequent characters, thereby minimizing the overall size of the data.

Implementation of Huffman tree

This section details the implementation of the Huffman Tree and the associated data structures and methods used in the Huffman Coding program. The program is built using a combination of structs and classes to efficiently handle file compression and decompression. Key components include the MinHeapNode struct, which represents nodes in the Huffman Tree, and the HuffmanCoding class, which encapsulates the core logic and methods for both compression and decompression processes.

“MinHeapNode” Struct

The MinHeapNode Struct which represents a single node in the Huffman binary Tree. Each node contains information about a character and its frequency in the input file, as well as pointers to its

left and right children. This structure is fundamental in building the Huffman Tree and generating Huffman codes.

“HuffmanCoding” class

The” HuffmanCoding”(see Appendix A for declarations) class is central to the implementation. It implements the methods for file compression and decompression. The class contains the “MinHeap” struct which is a vector used during the construction of the Huffman Tree. Initially, nodes for each character and its frequency are added to the heap. The two nodes with the smallest frequencies are repeatedly extracted and combined into a new node, which is then inserted back into the heap. This process continues until a single node remains in the heap, representing the root of the Huffman Tree. Moreover, It contains several methods that preform operations on the “MinHeap” struct as listed below(see Appendix B for definitions).

Public Methods

- compressFile: Compresses the specified input file and writes the compressed data to the specified output file.
- decompressFile: Decompresses the specified input file and writes the decompressed data to the specified output file.

Private Methods

- buildHuffmanTree: Builds the Huffman tree based on character frequencies.
- generateHuffmanCodes: Generates Huffman codes for each character by traversing the Huffman tree.
- createAndBuildMinHeap: Creates and builds a min-heap from the character frequency map.

- minHeapify: Maintains the min-heap property by rearranging elements.
- extractMin: Extracts the minimum frequency node from the min-heap.
- insertMinHeap: Inserts a new node into the min-heap.
- buildMinHeap: Builds a min-heap from an unordered array of nodes.
- swapMinHeapNode: Swaps two min-heap nodes.

How the Program Runs

The program executes the Huffman Coding algorithm through a sequence of well-defined steps to compress and decompress files. Below is a detailed description of the program's execution flow, including the key processes involved in both compression and decompression.

Compression Process

1. Reading the Input File: The program begins the input file and calculates the frequency of each character storing it in an unordered map.

2. Building the Min-Heap: Using the frequency map, the program constructs a min-heap. Each node in the min-heap represents a character and its frequency. The min-heap ensures that the node with the smallest frequency is always at the root.

3. Constructing the Huffman Tree: The program repeatedly extracts the two nodes with the smallest frequencies from the min-heap and combines them into a new node. This new node's frequency is the sum of the two extracted nodes' frequencies. The new node is then inserted back into the min-heap. This process continues until only one node remains in the heap, representing the root of the Huffman Tree.

4. Generating Huffman Codes: The program traverses the Huffman Tree to generate binary codes for each character. These codes are stored in an unordered map, where each character is associated with its corresponding Huffman code.

5. Encoding the Input File: The program reads the input file again and replaces each character with its corresponding Huffman code to create an encoded string. To ensure the encoded data can be written as binary, padding is added to make its length a multiple of 8.
6. Writing the Compressed Output: The encoded string is divided into 8-bit chunks, converted to their respective binary representations, and written to the output file. Additionally, the Huffman codes are stored in the output file to facilitate decompression.

Decompression Process

1. Reading the Compressed File: The program reads the Huffman codes and the encoded data from the compressed file. The Huffman codes are used to reconstruct the Huffman Tree.

2. Reconstructing the Huffman Tree:Using the Huffman codes, the program rebuilds the Huffman Tree. Each code guides the placement of characters in the tree.

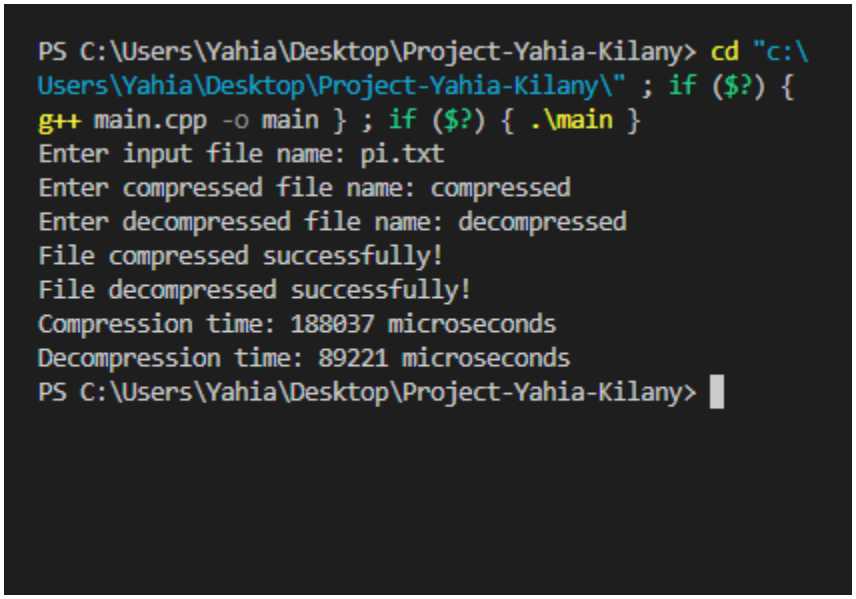
3. Decoding the Compressed Data:The program reads the binary data from the compressed file and uses the reconstructed Huffman Tree to decode the data back into the original

characters. Each bit in the binary data guides the traversal of the tree until a leaf node (character) is reached.

4. Writing the Decompressed Output: The decoded characters are written to the specified output file, reconstructing the original input file.

Example Run

Before running the program the user has to put the file that he wants to compress into the same folder as the root directory of the program.



The user is prompted to enter the names of the input file, the compressed output file, and the decompressed output file. The program then outputs a confirmation message for the compression, the decompression, and the time it took to complete the two processes and it creates the compressed and decompressed files in the root directory of the program.

Analysis and Critique

The Huffman Coding algorithm effectively reduces the file size by approximately 58% in the test case. The time complexity of building the Huffman Tree and generating codes is O(n log n), making it efficient for large datasets. However, the compression ratio depends on the frequency distribution of the input data. Files with uniformly distributed characters may not achieve significant compression.

Experimental Results

These tests were made with files sourced from *The Canterbury Corpus*.

Input: pi.txt (size:977 KB)

Output: compressed (size: 415 KB)

Input: bible.txt (3953 KB)

Output: compressed (2168 KB)

Conclusions

Huffman Coding is an effective method for lossless data compression. The implemented C++ program successfully compresses and decompresses text files, demonstrating the algorithm's practicality and efficiency. Future work could involve optimizing the code further and exploring adaptive Huffman Coding for streaming data.

References

The Canterbury Corpus. (n.d.). https://corpus.canterbury.ac.nz/descriptions/#cantrbry

GeeksforGeeks. (2023, September 11). *Huffman Coding Greedy Algo-3*. GeeksforGeeks. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Appendices

Appendix A

HuffmanCoding.h

```
#ifndef HUFFMAN_CODING_H
#define HUFFMAN_CODING_H
#include <cstdlib>
#include <iostream>
#include <vector>
#include <unordered_map>
#include <fstream>
#include <bitset>
#include <queue>
using namespace std;

// Huffman tree node
struct MinHeapNode {
    char data; // One of the input characters
    unsigned freq; // Frequency of the character
    MinHeapNode* left , * right; // Left and right child of this node
    MinHeapNode(char data, unsigned freq) : data(data), freq(freq),
left(nullptr), right(nullptr) {}
};

// Huffman Coding class
class HuffmanCoding {
public:
    void compressFile(const string& inputFile, const string& outputFile);
    void decompressFile(const string& inputFile, const string&
outputFile);

private:
    struct MinHeap {
        vector<MinHeapNode*> array; // Array of minheap node pointers
    };

    MinHeapNode* buildHuffmanTree(const unordered_map<char, unsigned>&
freqmap);
```

```
    void generateHuffmanCodes(MinHeapNode* root, string code,
unordered_map<char, string>& huffmanCodes);
    MinHeap* createAndBuildMinHeap(const unordered_map<char, unsigned>&
freqmap);
    void minHeapify(MinHeap* minHeap, int idx);
    MinHeapNode* extractMin(MinHeap* minHeap);
    void insertMinHeap(MinHeap* minHeap, MinHeapNode* minHeapNode);
    void buildMinHeap(MinHeap* minHeap);
    void swapMinHeapNode(MinHeapNode** a, MinHeapNode** b);
};
#include "HuffmanCoding.cpp" // Include the implementation file for
HuffmanCoding class
#endif // HUFFMAN_CODING_H
```

Appendix B

HuffmanCoding.cpp

```
#include "HuffmanCoding.h"
void HuffmanCoding::swapMinHeapNode(MinHeapNode** a, MinHeapNode** b) {
    MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

void HuffmanCoding::minHeapify(MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->array.size() && minHeap->array[left]->freq <
minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->array.size() && minHeap->array[right]->freq <
minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

MinHeapNode* HuffmanCoding::extractMin(MinHeap* minHeap) {
    MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->array.size() - 1];
    minHeap->array.pop_back();
    minHeapify(minHeap, 0);
    return temp;
}

void HuffmanCoding::insertMinHeap(MinHeap* minHeap, MinHeapNode*
minHeapNode) {
```

```
    minHeap->array.push_back(minHeapNode);
    int i = minHeap->array.size() - 1;

    while (i > 0 && minHeap->array[(i - 1) / 2]->freq >
minHeap->array[i]->freq) {
        swap(minHeap->array[i], minHeap->array[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void HuffmanCoding::buildMinHeap(MinHeap* minHeap) {
    int n = minHeap->array.size() - 1;
    for (int i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

HuffmanCoding::MinHeap* HuffmanCoding::createAndBuildMinHeap(const
unordered_map<char, unsigned>& freqmap) {
    MinHeap* minHeap = new MinHeap;
    for (const auto& pair : freqmap) {
        MinHeapNode* temp = new MinHeapNode(pair.first, pair.second);
        minHeap->array.push_back(temp);
    }
    buildMinHeap(minHeap);
    return minHeap;
}

MinHeapNode* HuffmanCoding::buildHuffmanTree(const unordered_map<char,
unsigned>& freqmap) {
    MinHeapNode* left, * right, * top;
    MinHeap* minHeap = createAndBuildMinHeap(freqmap);

    while (minHeap->array.size() != 1) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        top = new MinHeapNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
```

```
    }
    return extractMin(minHeap);
}

void HuffmanCoding::generateHuffmanCodes(MinHeapNode* root, string code,
unordered_map<char, string>& huffmanCodes) {
    if (!root)
        return;

    if (!root->left && !root->right) {
        huffmanCodes[root->data] = code;
        return;
    }

    generateHuffmanCodes(root->left, code + "0", huffmanCodes);
    generateHuffmanCodes(root->right, code + "1", huffmanCodes);
}

void HuffmanCoding::compressFile(const string& inputFile, const string&
outputFile) {
    ifstream inFile(inputFile, ios::binary);
    if (!inFile) {
        cerr << "Error opening input file: " << inputFile << endl;
        return;
    }
    unordered_map<char, unsigned> freqMap;
    char ch;
    while (inFile.get(ch)) {
        if (ch == 13) {
            ch = 10;
        }
        freqMap[ch]++;
    }
    inFile.close();

    MinHeapNode* root = buildHuffmanTree(freqMap);
    unordered_map<char, string> huffmanCodes;
    generateHuffmanCodes(root, "", huffmanCodes);

    ofstream outFile(outputFile, ios::binary);
```

```
    if (!outFile) {
        cerr << "Error opening output file: " << outputFile << endl;
        return;
    }
    for (const auto& pair : huffmanCodes) {
        outFile << pair.first << pair.second << '\n';
    }
    outFile << huffmanCodes.begin()->first << "\n";

    inFile.open(inputFile);
    if (!inFile) {
        cerr << "Error reopening input file: " << inputFile << endl;
        return;
    }
    string encodedText;
    while (inFile.get(ch)) {
        encodedText += huffmanCodes[ch];
    }

    int padding = (8 - encodedText.length() % 8) % 8;
    encodedText += string(padding, '0');

    for (size_t i = 0; i < encodedText.length(); i += 8) {
        bitset<8> bits(encodedText.substr(i, 8));
        char c = static_cast<char>(bits.to_ulong());
        outFile.put(c);
    }
    inFile.close();
    outFile.close();

    cout << "File compressed successfully!" << endl;
}

void HuffmanCoding::decompressFile(const string& inputFile, const string&
outputFile) {
    ifstream inFile(inputFile, ios::binary);
    if (!inFile) {
        cerr << "Error opening input file: " << inputFile << endl;
        return;
    }
```

```
    unordered_map<char, string> huffmanCodes;
    char ch;
    string code;
    string line;
    while (getline(inFile, line)) {
        ch = line[0];
        if (ch == 0 || ch == 10 || ch == 13) {
            ch = '\n';
            getline(inFile, line);
            code = line;
        } else {
            code = line.substr(1);
        }
        if (huffmanCodes.find(ch) != huffmanCodes.end()) {
            break;
        }
        huffmanCodes[ch] = code;
    }

    MinHeapNode* root = new MinHeapNode('$', 0);
    for (const auto& pair : huffmanCodes) {
        MinHeapNode* current = root;
        for (char c : pair.second) {
            if (c == '0') {
                if (!current->left) {
                    current->left = new MinHeapNode('$', 0);
                }
                current = current->left;
            } else if (c == '1') {
                if (!current->right) {
                    current->right = new MinHeapNode('$', 0);
                }
                current = current->right;
            }
        }
        current->data = pair.first;
    }

    ofstream outFile(outputFile);
    if (!outFile) {
```

```
        cerr << "Error opening output file: " << outputFile << endl;
        return;
    }

    MinHeapNode* current = root;
    while (inFile.get(ch)) {
        bitset<8> bits(ch);
        for (int i = 7; i >= 0; --i) {
            if (bits[i] == 0) {
                current = current->left;
            } else {
                current = current->right;
            }
            if (!current->left && !current->right) {
                outFile << current->data;
                current = root;
            }
        }
    }
    inFile.close();
    outFile.close();
    cout<<"File decompressed successfully!"<<endl;
}
```

Appendix C

main.cpp

```
#include "HuffmanCoding.h" // Include the header file for HuffmanCoding
class
#include <chrono>
using namespace std::chrono;
int main() {
    HuffmanCoding huffman;

    string inputFile, compressedFile, decompressedFile;

    // Get file names from user
    cout << "Enter input file name: ";
    cin >> inputFile;
    cout << "Enter compressed file name: ";
    cin >> compressedFile;
    cout << "Enter decompressed file name: ";
    cin >> decompressedFile;

    // Compress the input file and then decompress it
    auto compress_start = high_resolution_clock::now();
    huffman.compressFile(inputFile, compressedFile);
    auto compress_stop = high_resolution_clock::now();
    auto decompress_start = high_resolution_clock::now();
    huffman.decompressFile(compressedFile, decompressedFile);
    auto decompress_stop = high_resolution_clock::now();
    auto compress_duration = duration_cast<microseconds>(compress_stop -
compress_start);
    auto decompress_duration = duration_cast<microseconds>(decompress_stop
- decompress_start);
    cout << "Compression time: " << compress_duration.count() << "
microseconds" << endl;
    cout << "Decompression time: " << decompress_duration.count() << "
microseconds" << endl;

    return 0;
}
```