File compression/decompression using Huffman Algorithm.
A MINOR PROJECT REPORT

Submitted by

Arsh Bhatia **RA2011031010053**
Aditi Mishra **RA2011031010041**

Under the guidance of

Dr. V.R Balasaraswathi **Assistant Professor**

In partial fulfillment for the award of the degree

of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

S.R.M. Nagar, Kattankulathur, Chengalpattu District

Session-2022

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

BONAFIDE CERTIFICATE

Certified that the minor project report titled **"File compression/ decompression using Huffman coding"** is the bonafide work of **"ADITI MISHRA (RA2011031010041)"** who carried out the minor project work under my supervision. Certified further, that to the best ofmy knowledge the work reported herein does not form any other project report or dissertation onthe basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Dr. V.R Balasaraswathi

Submitted for the university examination held on_____at

SRM Institute of Science and Technology, Kattankulathur-603 203.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

TABLE OF CONTENTS

ABSTRACT

Lossless text data compression is an important field as it significantly reduces storage requirements and communication costs. In this work, the focus is directed mainly to different file compression coding techniques and comparisons between them. Some memory-efficient encoding schemes are analyzed and implemented in this work. They are Shannon Fano Coding, Huffman Coding, Repeated Huffman Coding and Run-Length coding. A new algorithm "Modified Run-Length Coding" is also proposed and compared with the other algorithms. These analyses show how these coding techniques work, how much compression is possible for these coding techniques, the amount of memory needed for each technique an , comparison between these techniques to find out which technique is better in what conditions. It is observed from the experiments that the repeated Huffman Coding sha ows a higher compression ratio. Besides, the proposed Modified run-length coding shows a higher performance than the conventional one.

CONTRIBUTION TABLE:

Name	Reg no.	Contribution
Arsh Bhatia	**RA2011031010053**	**Implementation in cpp, Output, Explanation of algorithm, Final editing**
Aditi Mishra	**RA2011031010041**	**Problem Statement, Time and space complexity analysis, Design Technique used**
Arsh And Aditi	**RA2011031010053&41**	**Abstract, Approach for problem Algorithm for the problem, Content, Conclusion**

General Terms

Information Theory, Algorithms.

Keywords

Data compression; Lossless compression; Encoding; Compression Ratio; Code length; Standard deviation

INTRODUCTION

In computer science and information theory, text compression is the process of encoding texts using fewer bits or symbols than an original representation, by using specific encoding techniques. Text data compression is useful because it helps to reduce the consumption of expensive resources, such as hard disk space or transmission bandwidth. But the problem is that the decompression must be needed for further utilization and this extra processing may be unfavorable to some applications. The objectives of this work are efficient representation and implementation of some text data compression algorithm, proposing a new compression method named "Modified Run-Length Coding", computation of some important compression factors for each of these algorithms, comparison between different encoding techniques, and improvement of performance of different data compression techniques and selecting a suitable encoding technique for real life system. This paper is organized as follows: Section 2 discusses the related works for data compression. Section 3 explains the different algorithms of data compression techniques. Section 3 also analyzes the experimental results with proper reasons.
(HFLC), and Fixed-length code (FLC), with entropy 4.719, 4.855, 5.014, and 6.889 respectively. A similar work is found in [12] which analyzed Huffman algorithm and compared it with other common compression techniques like Arithmetic, LZW and Run Length Encoding on the basis of their use in different applications and their advantages and concluded that arithmetic coding is very efficient for more frequently occurring sequences of pixels with fewer bits and reduces the file size dramatically. RLE is simple to implement and fast to execute. LZW algorithm is better to use for TIFF, GIF and Textual Files and is easy to implement, fast and lossless algorithm whereas Huffman algorithm is used in JPEG compression which produces optimal and compact code but relatively slow. Shannon Fano coding, Huffman coding, Adaptive Huffman coding, RLE, Arithmetic coding, LZ77, LZ78 and LZW were tested using the Calgary corpus in [4]. In the Statistical compression techniques, Arithmetic coding technique outperformed the rest with some identifiable improvements. LZB outperformed LZ77, LZSS and LZH to show a marked compression. LZ78 and LZW were outperformed in their average BPC by LZFG. The entropy was calculated in [8] on the same English text file for Shanon Fano coding, Huffman Encoding, Run- Length Encoding (RLE), Lempel-Ziv-Welch (LZW). The compression ratio was almost same for the Shannon Fano and Huffman coding and 54.7% space could be saved by those two algorithms. The compression ratio of Run length encoding and Lempel-Ziv-Welch algorithms was low as compared with the Huffman and Shannon Fano algorithms and it concluded that, Huffman encoding algorithm is the best result for the text files. Another comparison based work was [13] which discussed Run Length based codes like Golomb code, Frequency Directed run length code (FDR), Extended FDR, Modified FDR, Shifted Alternate FDR, and OLEL coding methodology; Huffman coding; Shannon Fano coding; Lempel-Ziv-Welch coding; Arithmetic coding; Universal coding like Elias Gamma code, Elias Delta code, Elias Omega code and proposed double compression using Huffman code technique to reduce the test data volume and area even further. First, the data was compressed by any one of the run length based codes like Golomb code, FDR code, EFDR, MFDR, SAFDR, and OLEL coding and then from the compressed

data, another compression was made by Huffman code. Double compression using Huffman code had compression ratio of 50.8%. Better results were achieved for the data sets with redundant data. The execution times, compression ratio and efficiency of compression methods in a client-server distributed environment using four compression algorithms: Huffman algorithm, Shannon Fano algorithm, Lempel-Ziv algorithm and Run-Length Encoding algorithm were analyzed in [9]. The data from a client was distributed to multiple processors/servers, subsequently compressed by the servers at remote locations, and sent back to the client. Simgrid Framework was used and results showed that the LZ algorithm attains better efficiency/scalability and compression ratio but it worked slower than other algorithms. Huffman coding, LZW coding, LZW based Huffman coding and Huffman based LZW were compared for multiple and single compression in [6]. It showed that Huffman based LZW Encoding can Compresses data more than all other three cases, when in average case the Huffman based LZW compression ratio is 4.41, where other maximum average compression ratio is 4.17 in case of LZW compression. The Huffman based LZW compression is better in some of the cases than LZW Compression.

RELATED WORKS

In 1949, C. Shannon and R. Fano devised a systematic way to assign code words based on probabilities of blocks called the Shannon Fano Coding. An optimal method for this was found by D. Huffman in 1951 which is known as the Huffman Algorithm [1]. Huffman Algorithm is being used for compression since then. A research in [7] showed that text data compression using Shannon Fano algorithm has a same effectiveness with Huffman algorithm when all character in string are repeated and when the statement is short and just one character in the statement is repeated, but the Shannon Fano algorithm is more effective than Huffman algorithm when the data has a long statement and data text have more combination character in statement or in string or word. A variety of data compression methods spanning almost forty years of research, from the work of Shannon, Fano and Huffman in the late 40’s to a technique developed in 1986 was surveyed in [10]. The compression ratio, compression time and decompression time for the Run Length Encoding (RLE) Algorithm, Huffman Encoding Algorithm, Shannon Fano Algorithm, Adaptive Huffman Encoding Algorithm, Arithmetic Encoding Algorithm and Lempel Zev Welch (LZW) Algorithm using random text files were compared in [2]. It showed that, the compression time increased as file size increased. For Run Length encoding it was a constant value and not affected by the file size. Compression times were average values for two Static Huffman approaches, and times of Shannon Fano approach were smaller than the other algorithm. The LZW Algorithm worked well for only small files. Compression times of Adaptive Huffman algorithm were the highest. Decompression times of all the algorithms were less than 500000 milliseconds except the Adaptive Huffman Algorithm and LZW. The compression ratio were similar except the Run Length coding and for small sized files, LZW gave the best results. The comparison between RLE, Huffman, Arithmetic Encoding, LZ-77, LZW and LZH (first LZ applied, then Huffman) on random .doc, .txt, .bmp, .tif, .gif, and .jpg files is shown in [3]. It showed that, LZW and Huffman gave nearly same results when used for compressing text files. Using LZH compression to compress a text file gave an improved compression ratio than the others. Different methods of data compression algorithms such as: LZW, Huffman, Fixedlength code (FLC), and Huffman after using Fixed-length code (HFLC) on English text files were studied in [12] in terms of compression Size, Ratio, Time (Speed), and Entropy. LZW was the best algorithm in all of the compression scales, then Huffman, Huffman after using Fixed length code

PROBLEM STATEMENT

Text File compression and decompression has always been a challenging problem in the last decade. The purpose is to develop more effective algorithm in order to reduce the computing time and utilize lower memory space.
Our project is to develop a text file compression/decompression tool using Huffman Coding, a lossless, bottom-up compression algorithm. Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code proceeds by means of Huffman coding Our algorithm can currently compress or decompress up to 50% of the size of any text file.

- **Save Storage Space**

  Compressing folders can prevent problems before they occur, and also save more space for youto store other valuable content, and there will be no lag when using the computer.

- **Archive Documents**

  Compression tools are also great for archiving old files. One can compress any number of files ofthe same type into a single file and add corresponding notes to them, which will simplify the filesystem.

- **Prevent Transmission Interruption**

  File compression increases data transfer speed The longer the file takes to send, the more likelythe transfer will be interrupted. The time required to transfer a compressed word file is one tenth of the time required to transfer the same uncompressed file, which will greatly reduce therisk of unexpected interruptions, and guarantee the work efficiency.

- **Ensure Data Security**

  File compression can also hide information. In addition, it is also a good choice to pack multiplefiles containing sensitive information with compression software and then add a password.

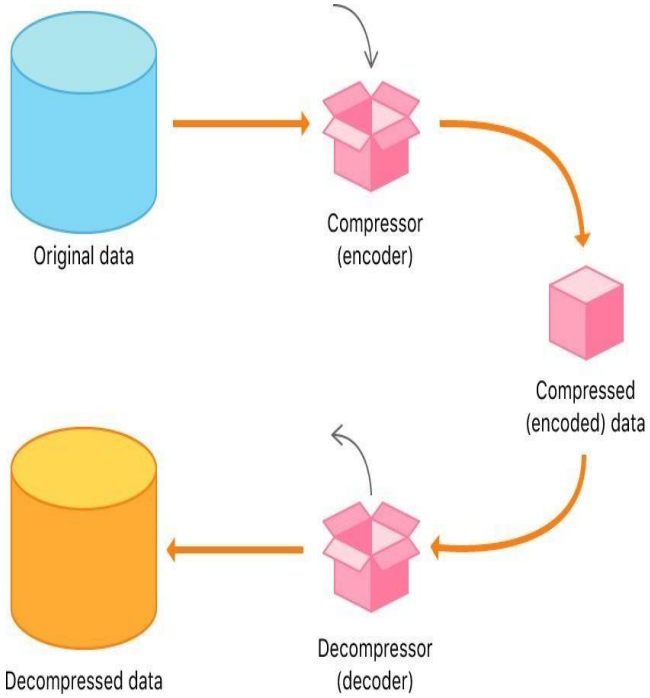
- **Meet Server Requirements**

  On some Internet servers, file compression is mandatory. The server operator may not allow thetransfer of uncompressed files. File compression is also useful when sending email attachments,which often have file size limitations.

BEST APPROACH –

- Greedy algorithm is the best approach for constructing the above-mentioned program because itgreedily searches for an optimal solution.

- The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it isa technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.



HUFFMAN ALGORITHM -

- Huffman coding is a lossless data compression algorithm. The idea is to assign variable-lengthcodes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix ofcode assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

- There are mainly two major parts in Huffman Coding
  I. Build a Huffman Tree from input characters.
  II. Traverse the Huffman Tree and assign codes to characters.

- Steps to build Huffman Tree

1. Input is an array of unique characters along with their frequency of occurrences andoutput is Huffman Tree.

2. Create a leaf node for each unique character and build a min heap of all leaf nodes (MinHeap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
3. Extract two nodes with the minimum frequency from the min heap.

4. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node asits right child. Add this node to the min heap.
5. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is theroot node and the tree is complete.
   Let us understand the algorithm with an example:

- Real-life applications of Huffman Encoding-

  - Huffman encoding is widely used in compression formats like GZIP, PKZIP (WinZip) and BZIP2.
  - Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precisedthe prefix codes)
  - Huffman encoding still dominates the compression industry since newer arithmetic andrange coding schemes are avoided due to their patent issues.

Implementation:

Project supports two functions:
1) Encode: Compresses input file passed.
2) Decode: Decompresses Huffman coded file passed back to its original file.

Encoding Source Code:

```
#include <iostream>
#include "huffman.hpp"
using namespace std;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cout << "Failed to detect Files";
        exit(1);
    }

    huffman f(argv[1], argv[2]);
    f.compress();
    cout << "Compressed successfully" << endl;

    return 0;
}
```

Decoding Source Code:

```
#include <iostream>
#include "huffman.hpp"
using namespace std;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cout << "Failed to detect Files";
        exit(1);
    }

    huffman f(argv[1], argv[2]);
    f.decompress();
    cout << "Decompressed successfully" << endl;

    return 0;
}
```

Huffman Code:

```
#include "huffman.hpp"
```

```
void huffman::createArr() {
    for (int i = 0; i < 128; i++) {
        arr.push_back(new Node());
        arr[i]->data = i;
        arr[i]->freq = 0;
    }
}

void huffman::traverse(Node* r, string str) {
    if (r->left == NULL && r->right == NULL) {
        r->code = str;
        return;
    }

    traverse(r->left, str + '0');
    traverse(r->right, str + '1');
}

int huffman::binToDec(string inStr) {
    int res = 0;
    for (auto c : inStr) {
        res = res * 2 + c - '0';
    }
    return res;
}

string huffman::decToBin(int inNum) {
    string temp = "", res = "";
    while (inNum > 0) {
        temp += (inNum % 2 + '0');
        inNum /= 2;
    }
    res.append(8 - temp.length(), '0');
    for (int i = temp.length() - 1; i >= 0; i--) {
        res += temp[i];
    }
    return res;
}

void huffman::buildTree(char a_code, string& path) {
    Node* curr = root;
    for (int i = 0; i < path.length(); i++) {
        if (path[i] == '0') {
            if (curr->left == NULL) {
                curr->left = new Node();
            }
            curr = curr->left;
```

```
        }
        else if (path[i] == '1') {
            if (curr->right == NULL) {
                curr->right = new Node();
            }
            curr = curr->right;
        }
    }
    curr->data = a_code;
}

void huffman::createMinHeap() {
    char  id;
    inFile.open(inFileName, ios::in);
    inFile.get(id);
    //Incrementing frequency of characters that appear in the input file
    while (!inFile.eof()) {
        arr[id]->freq++;
        inFile.get(id);
    }
    inFile.close();
    //Pushing the Nodes which appear in the file into the priority queue (Min Heap)
    for (int i = 0; i < 128; i++) {
        if (arr[i]->freq > 0) {
            minHeap.push(arr[i]);
        }
    }
}

void huffman::createTree() {
    //Creating Huffman Tree with the Min Heap created earlier
    Node *left, *right;
    priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
    while (tempPQ.size() != 1)
    {
        left = tempPQ.top();
        tempPQ.pop();

        right = tempPQ.top();
        tempPQ.pop();

        root = new Node();
        root->freq = left->freq + right->freq;

        root->left = left;
        root->right = right;
        tempPQ.push(root);
    }
```

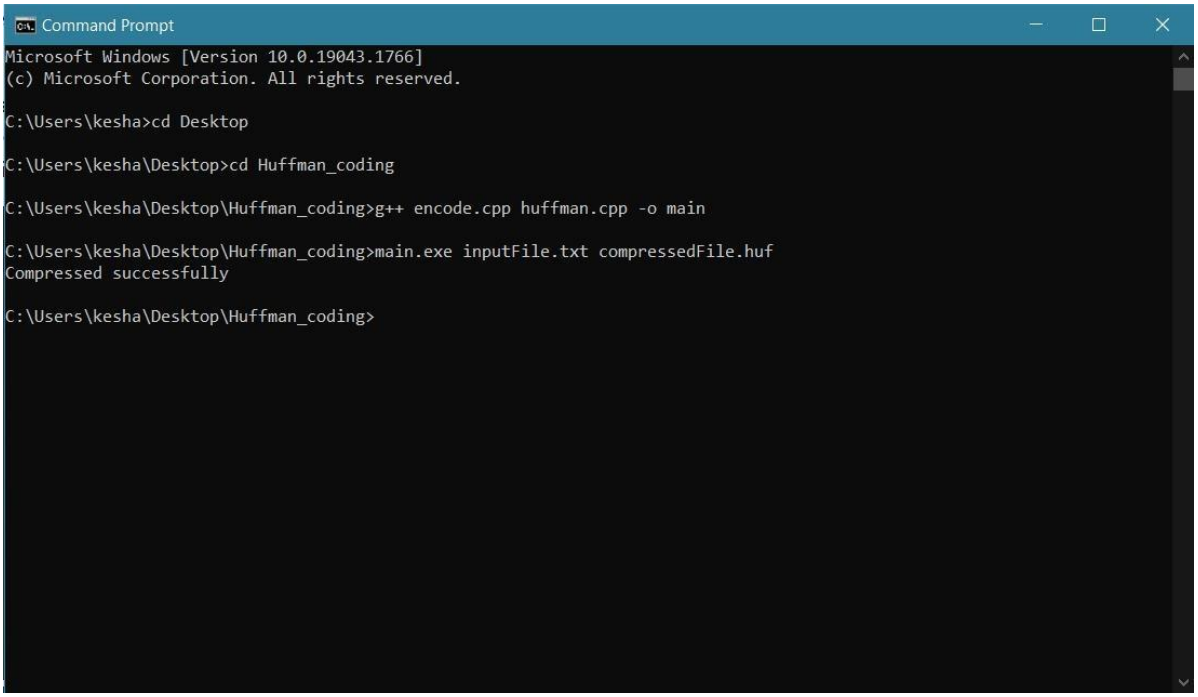
```
}

void huffman::createCodes() {
    //Traversing the Huffman Tree and assigning specific codes to each character
    traverse(root, "");
}

void huffman::saveEncodedFile() {
    //Saving encoded (.huf) file
    inFile.open(inFileName, ios::in);
    outFile.open(outFileName, ios::out | ios::binary);
    string in = "";
    string s = "";
    char id;

    //Saving the meta data (huffman tree)
    in += (char)minHeap.size();
    priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
    while (!tempPQ.empty()) {
        Node* curr = tempPQ.top();
        in += curr->data;
        //Saving 16 decimal values representing code of curr->data
        s.assign(127 - curr->code.length(), '0');
        s += '1';
        s += curr->code;
        //Saving decimal values of every 8-bit binary code
        in += (char)binToDec(s.substr(0, 8));
        for (int i = 0; i < 15; i++) {
            s = s.substr(8);
            in += (char)binToDec(s.substr(0, 8));
        }
        tempPQ.pop();
    }
    s.clear();

    //Saving codes of every charachter appearing in the input file
    inFile.get(id);
    while (!inFile.eof()) {
        s += arr[id]->code;
        //Saving decimal values of every 8-bit binary code
        while (s.length() > 8) {
            in += (char)binToDec(s.substr(0, 8));
            s = s.substr(8);
        }
        inFile.get(id);
    }

    //Finally if bits remaining are less than 8, append 0's
```

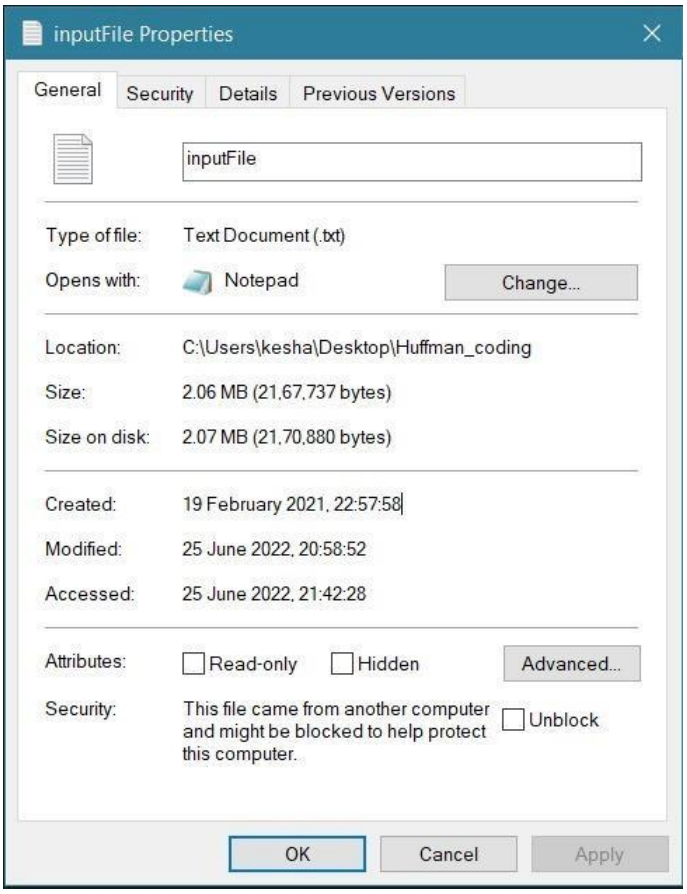
```
    int count = 8 - s.length();
    if (s.length() < 8) {
        s.append(count, '0');
    }
    in += (char)binToDec(s);
    //append count of appended 0's
    in += (char)count;

    //write the in string to the output file
    outFile.write(in.c_str(), in.size());
    inFile.close();
    outFile.close();
}

void huffman::saveDecodedFile() {
    inFile.open(inFileName, ios::in | ios::binary);
    outFile.open(outFileName, ios::out);
    unsigned char size;
    inFile.read(reinterpret_cast<char*>(&size), 1);
    //Reading count at the end of the file which is number of bits appended to make final value
8-bit
    inFile.seekg(-1, ios::end);
    char count0;
    inFile.read(&count0, 1);
    //Ignoring the meta data (huffman tree) (1 + 17 * size) and reading remaining file
    inFile.seekg(1 + 17 * size, ios::beg);

    vector<unsigned char> text;
    unsigned char textseg;
    inFile.read(reinterpret_cast<char*>(&textseg), 1);
    while (!inFile.eof()) {
        text.push_back(textseg);
        inFile.read(reinterpret_cast<char*>(&textseg), 1);
    }

    Node *curr = root;
    string path;
    for (int i = 0; i < text.size() - 1; i++) {
        //Converting decimal number to its equivalent 8-bit binary code
        path = decToBin(text[i]);
        if (i == text.size() - 2) {
            path = path.substr(0, 8 - count0);
        }
        //Traversing huffman tree and appending resultant data to the file
        for (int j = 0; j < path.size(); j++) {
            if (path[j] == '0') {
                curr = curr->left;
            }
```

```
            else {
                curr = curr->right;
            }

            if (curr->left == NULL && curr->right == NULL) {
                outFile.put(curr->data);
                curr = root;
            }
        }
    }
    inFile.close();
    outFile.close();
}

void huffman::getTree() {
    inFile.open(inFileName, ios::in | ios::binary);
    //Reading size of MinHeap
    unsigned char size;
    inFile.read(reinterpret_cast<char*>(&size), 1);
    root = new Node();
    //next size * (1 + 16) characters contain (char)data and (string)code[in decimal]
    for(int i = 0; i < size; i++) {
        char aCode;
        unsigned char hCodeC[16];
        inFile.read(&aCode, 1);
        inFile.read(reinterpret_cast<char*>(hCodeC), 16);
        //converting decimal characters into their binary equivalent to obtain code
        string hCodeStr = "";
        for (int i = 0; i < 16; i++) {
            hCodeStr += decToBin(hCodeC[i]);
        }
        //Removing padding by ignoring first (127 - curr->code.length()) '0's and next '1'
character
        int j = 0;
        while (hCodeStr[j] == '0') {
            j++;
        }
        hCodeStr = hCodeStr.substr(j+1);
        //Adding node with aCode data and hCodeStr string to the huffman tree
        buildTree(aCode, hCodeStr);
    }
    inFile.close();
}

void huffman::compress() {
    createMinHeap();
    createTree();
    createCodes();
```

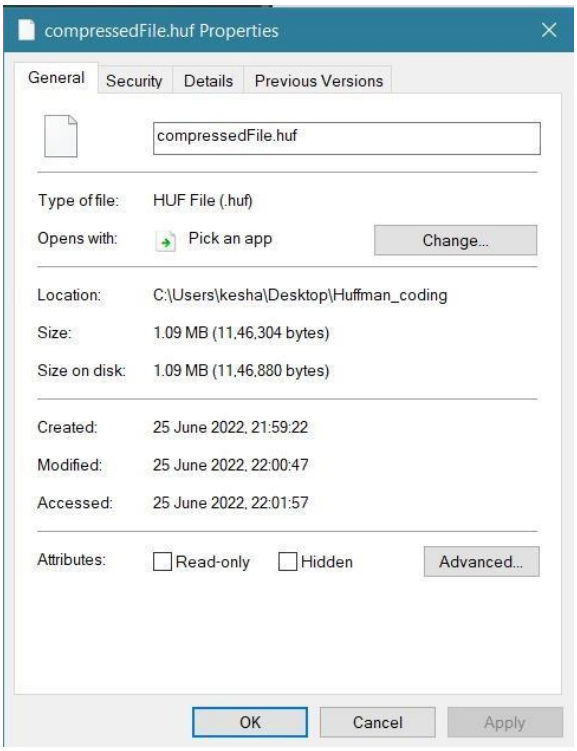
```
    saveEncodedFile();
}

void huffman::decompress() {
    getTree();
    saveDecodedFile();
}
```

Compression Output:



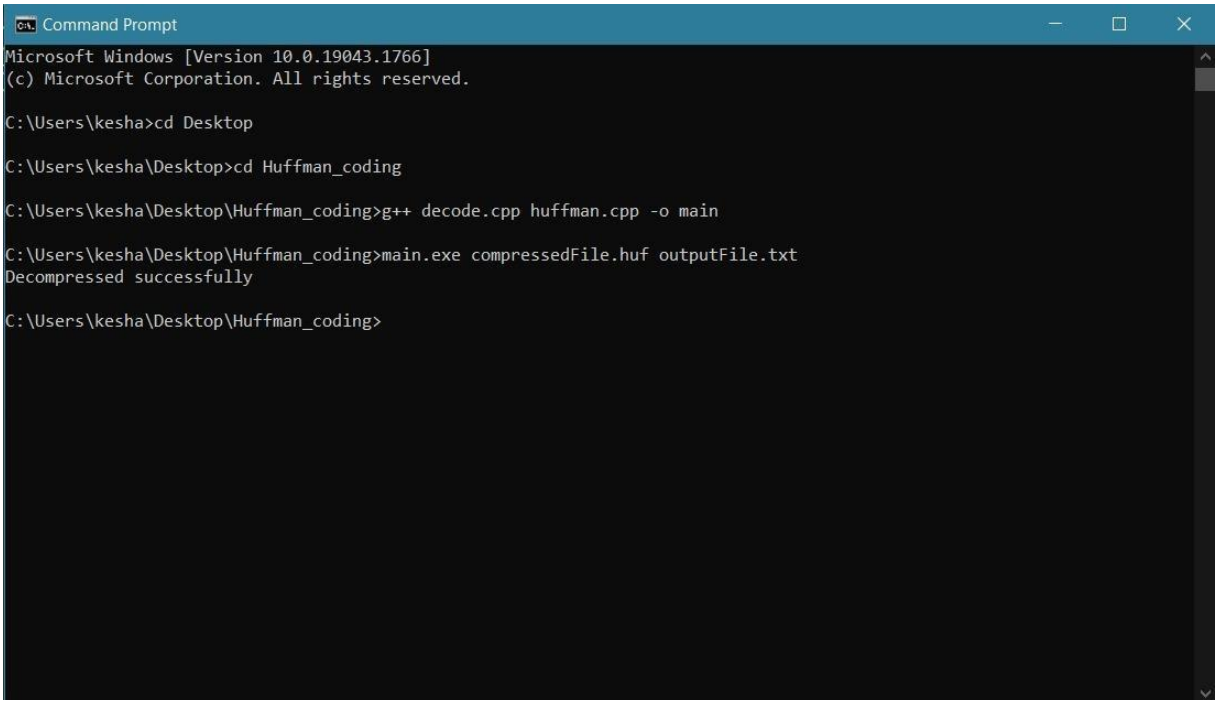


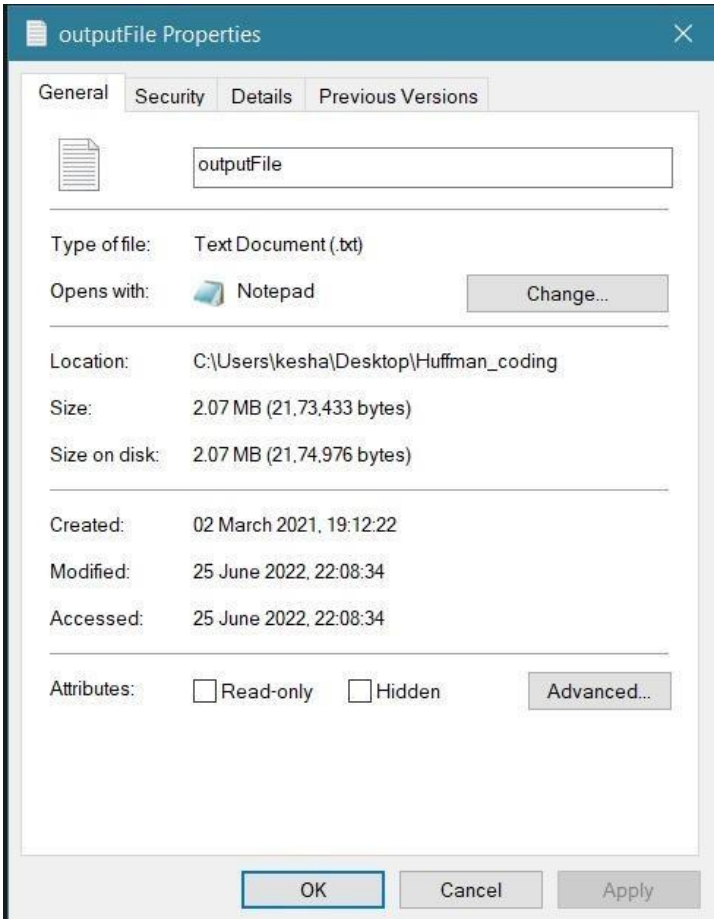
Normal File:



After Compression File:

Decompression Output:





Decompressed File:

Result:

This project is just an implementation of Huffman coding, it is not as efficient as the compression algorithm used currently to compress files.
Example: inputFile.txt (2.7MB) is compressed to compressedFile.huf (1.9MB) file and decompressed back to ouputFile.txt (2.7MB).

CONCLUSION

After computing and comparing the compression ratio, average code length, and standard deviation for Shannon Fano Coding, Huffman Coding, Repeated Huffman Coding, RunLength Coding, and Modified Run-Length Coding, an idea is generated about how much compression can be obtained by each technique. So, now the most effective algorithm can be used based on the input text file size, content type, available memory and execution time to get the best result. A new approach for data compression “Modified Run Length algorithm” is also proposed here and which gives a lot better compression than the existing Run-Length algorithm. Future works can be carried on an efficient and optimal coding technique using mixture of two or more coding techniques for image file, exe file, etc. to improve the compressionratio and reduce average code length.