Text Compression Utility

Term Project Report Fall 2021

Analysis and Design of Algorithms

Department of Computer Science and Engineering

The American University in Cairo

Aref Mamdouh: 900192254

Ibrahim Soliman: 900192478

Tamer Osman: 900192103

Contents

1. Abstract

The text compression utility we have built has the ability to take in a file, compress it and decompress it. Regardless of this project being just a text compression utility, our team realized that this project can also be used as an encryption utility. Using the Huffman tree algorithm, we were able to compress files to much smaller sizes. Moreover, we used the encoding we gave the characters during decompression in order to get the original file even when the original file is deleted. Finally, the compression ratio and efficiency are calculated at output. Throughout this report we talk about the methodology of our project, the specifications of the algorithm we used (Huffman), Data specifications, experimental results, and our teams take on the entire project.

Keywords: Huffman, text compression, ASCII, binary encoding

2. Introduction

The purpose of this program is to compress text files using Huffman coding. Huffman coding is a Greedy lossless compression algorithm that follows several steps. First of all, the frequencies of all characters in the text file are calculated. Then, a huffman tree is created in order to encode each of those characters in binary. Subsequently, a compressed file is created which should consist of a number of bits much less than the original file. The reason for that is that each character is normally stored using 8 bits while the average bit length for Huffman variable length coding is less than that. Finally, two other files are outputted along with the compressed file and those are a table of code and a file called zeroes both of which will be elaborated on in the following sections of the report.

3. Problem Definition

The problem presented is that it is required to create a text compression utility for ASCII-encoded text using the Huffman coding algorithm. The text compression utility should be able to compress and decompress that file. The compression ratio and the efficiency should be calculated and shown in the output. To compress the file, we pass the file to be read and use the huffman tree, then we save the encoded text and the new encoded table in an output file. To decompress the file, the program returns the original text file. When running, the program takes three parameters which are whether the user wants to compress or decompress, the path of the file, and the path the user would like to place the compressed/decompressed file.

4. Methodology

In this project we pass a file and calculate the frequencies of each character,we then pass that to a function to create the Huffman tree. Furthermore, we take the new encoding of each character and place them in a map along with their characters, the key is the character and the value is the new encoding. We pass that map to the compress function, and then we read the file we want to compress again. It goes over each character, searches for the character in the map and then replaces the character with its new encoding. After we replace all the characters with their new encoding, we now have a string filled with 1's and 0's. We divide them into bits of 8 and then convert them to their following ASCII character. The last few bits are not always going to be 8 so we add the 0's needed by padding. In decompression, we take the file filled with characters and convert them to their equivalent ASCII encoding, we pass the lagging zeros to the function and so we remove the zeros, from the last 8 bits, we added during compression. In addition, since no character's encoding is the prefix of another, we start at the starting point of the binary code and increase in bits until we find one of the tests in the map that we created using a file that has the characters and their encodings. We have the encoding as the key and the character as the value. Anyways, we go bit by bit from the starting point until we find one of the encodings in the map and then we replace that encoding with its corresponding character. The last part is calculating the entropy and the average bit length (< L >). When we added all the characters and their frequencies in the map, we looped over that map and the frequencies to a variable called totalcount. We then looped over each character and multiplied the division of the frequency of that character and totalcount

(probability) by the length of the bits of that character. The entropy was the probability multiplied by the logarithm of the reciprocal of the probability.

Pseudo code of the calculation:

```
double sum = 0;
double entropy = 0;
itr2 = mapcount.begin(); //mapcount map has the key as the character and the
//value as the frequency
for (int i = 0 to i < mapcount.size()) {
        itr = charencode.find(itr2->first[0]); // charencode has the key as the
        //character and the code as the value
        int freq = itr2->second;
        int length = (itr->second).length();
        entropy += ((freq / totalcount) * log2(1.0 / (freq / totalcount)));
        sum += (length * (freq / totalcount));
        itr2++;
}
double compressionratio = sum / 8;
double efficiency = entropy / sum;
```

5. Specification of Algorithms to be used

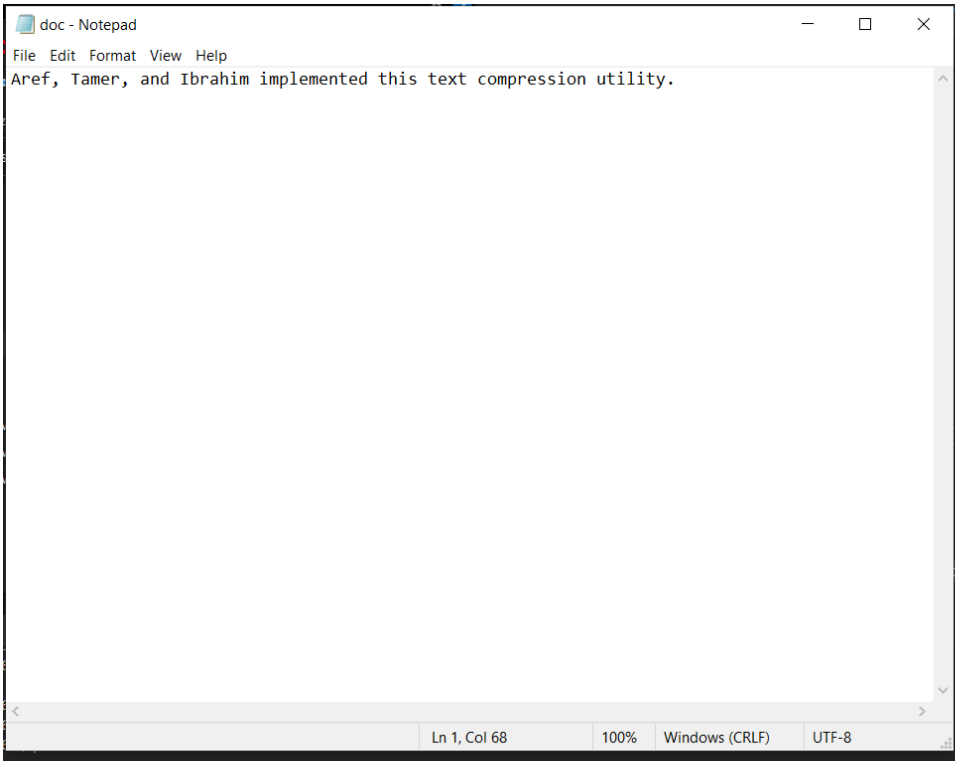
For this text compression utility, we used huffman coding greedy algorithm, which is a lossless data compression algorithm. We first extracted all distinct characters and their frequencies from the input file, and we implemented the algorithm by assigning variable-length codes to characters extracted from the input file. Each code assigned to a character could not be a prefix to another character code, as this would be needed to make sure that there is no ambiguity when decoding the created bitstream. We implemented the huffman coding algorithms by creating a leaf node for each distinct character then building a minimum heap of all leaf nodes. Then we would extract two nodes with the least frequencies from the heap. We would then create a new internal node with a frequency of the sum of the two leaf nodes, making the first extracted node the left child and the second extracted node the right child, and adding the new node to the heap again. By repeating these steps we would reach one node in the heap which meant the tree was finished. As each character had a frequency, the most frequent character got the smallest code and the least frequent character got the largest code.

6. Data Specifications

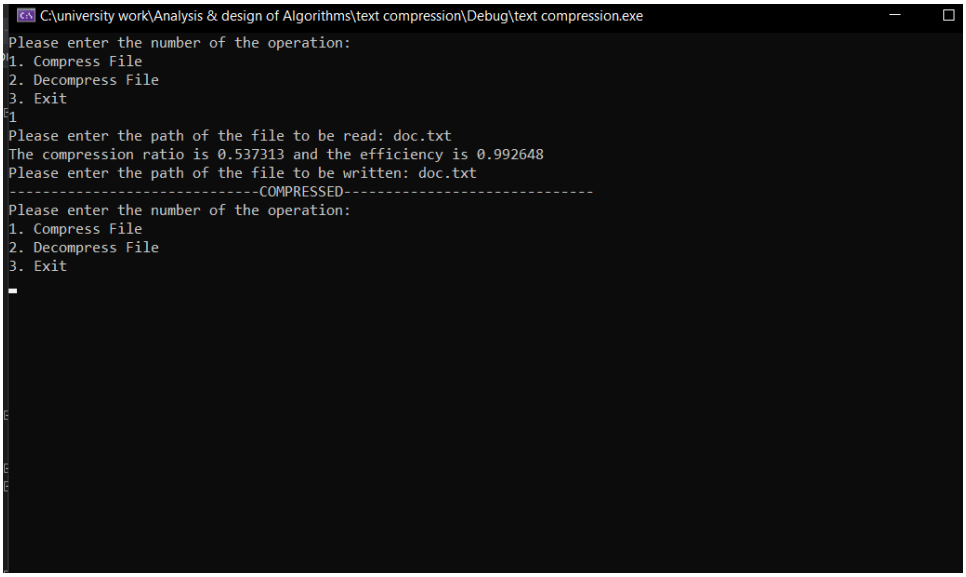
- Text file to be read (during compression)
- Compressed file (during decompression)
- File with the new encoding of characters (during decompression)
- File with the padded zeros (during decompression)

7. Experimental Results

- doc.txt to be compressed

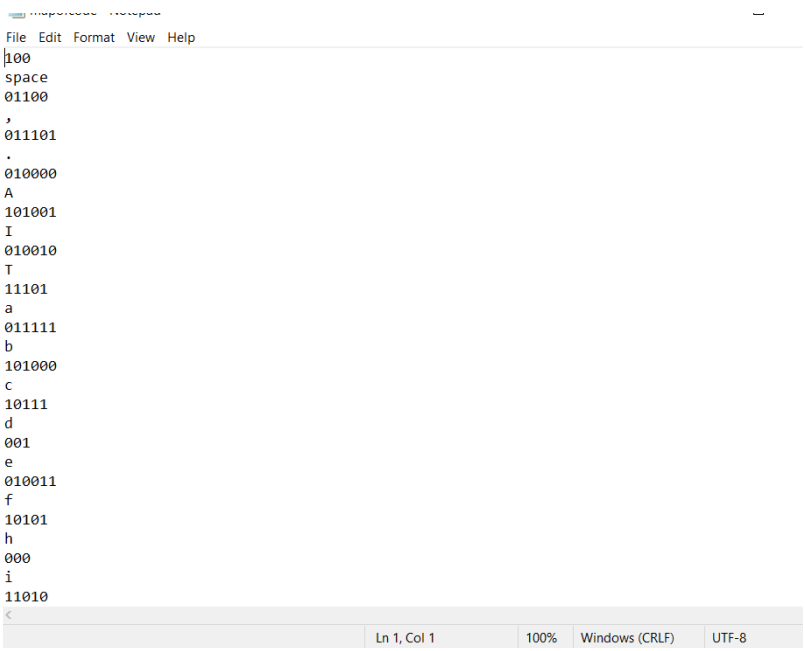


- Compression



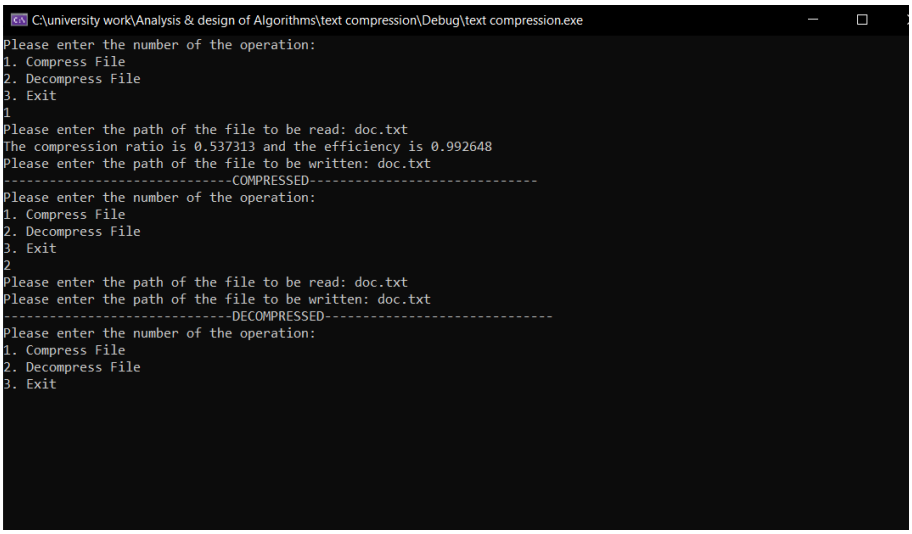
To show that we do not need the original file I compressed the original into the same file, so now we do not have the original, just the compressed file. The compression ratio and efficiency are also in the output.

This is the output of the compressed file.



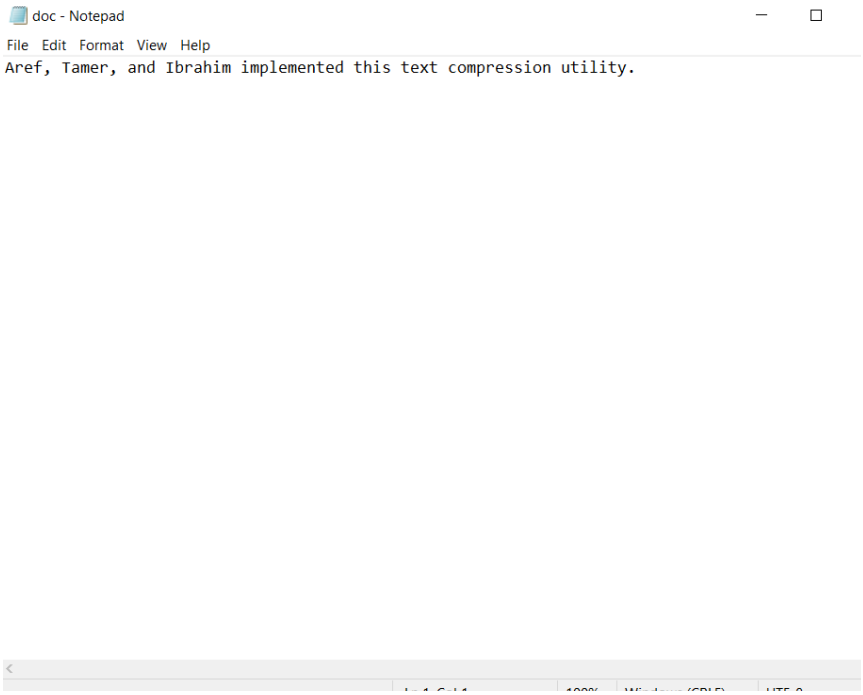
This is the file that has the characters and their new encodings.

- Decompression



Now I have written the compressed file back into the same file.

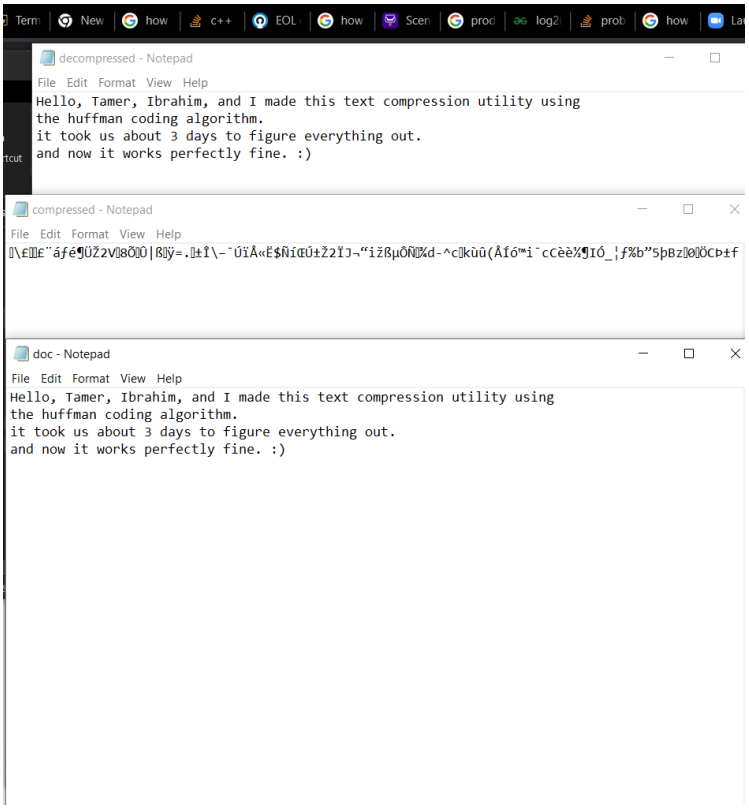
And the result:



And to show three different files at the same time, doc, compressed, and decompressed.

```
C:\university work\Analysis & design of Algorithms\text compression\Debug\text compression.exe
Please enter the number of the operation:
1. Compress File
2. Decompress File
3. Exit
1
Please enter the path of the file to be read: doc.txt
The compression ratio is 0.565541 and the efficiency is 0.992966
Please enter the path of the file to be written: compressed.txt
------------------------------COMPRESSED------------------------------
Please enter the number of the operation:
1. Compress File
2. Decompress File
3. Exit
2
Please enter the path of the file to be read: compressed.txt
Please enter the path of the file to be written: decompressed.txt
------------------------------DECOMPRESSED------------------------------
Please enter the number of the operation:
1. Compress File
2. Decompress File
3. Exit
```

The files:



8. Analysis and Critique

The output file after decompression is almost always half the size of the original file. Moreover, during decompression the output file is identical to the original file. The algorithm used allows for not just compression and decompression but it could also be used for the encryption of messages in another project. This project helped us implement algorithms we took in class, in real life problems. First, we used a priority queue (minimum heap) and then extracted its nodes one by one. Technically, this is the same as performing heap sort and has time complexity of O(nlogn) and space complexity of O(n). Also, we used 3 maps; to map each character to its probability in the code, to map each character to its binary encoding and to map each binary encoding for its relevant character (for decompression). To place the elements in the map, this has time complexity O(n) and search in the map has a time complexity of O(1). Recursing through the Huffman tree is basically traversing a binary tree, so it is a normal DFS that has time complexity of O(n + m). Since the number of edges that can originate from a node is limited to 2 in the case of a Binary Tree, the maximum number of total edges in a Binary Tree is n-1, where n is the total number of nodes. The complexity then becomes O(n + n-1), which is O(n). For calculating the efficiency, we used the log function which operates in time complexity of O(log n). Since the loop is constant, then the total complexity of this part is O(log n). For the decompression, we looped over the characters in the compressed file and checked if the appended binary code matched any code in the table. Since we used a hash for saving the table, then checking if the code exists takes O(1). So, the overall complexity of this part is represented by just a loop of constant operations, so it is O(n).

9. Conclusion

Our project, the text compression utility, took a great deal of research and understanding of the Huffman tree algorithm. Moreover, it took more understanding and brainstorming to know how to read the files and actually replace characters with their new encoding and vice versa without any complication such as missing characters or extra characters; or even reading the entire compressed text wrong and replacing them with entirely different characters. It was very interesting and prepared us for the upcoming projects we shall have in the future by triggering our problem solving skills which is key for a Computer Science major looking for a job.

10. Acknowledgements

This project was done by Aref Mamdouh, Tamer Osman, and Ibrahim Soliman.

Professor Amr Goneid helped us greatly by merely explaining the algorithms and answering all our questions. Moreover, our team had to do research to understand how to put the information we learned in class into code.