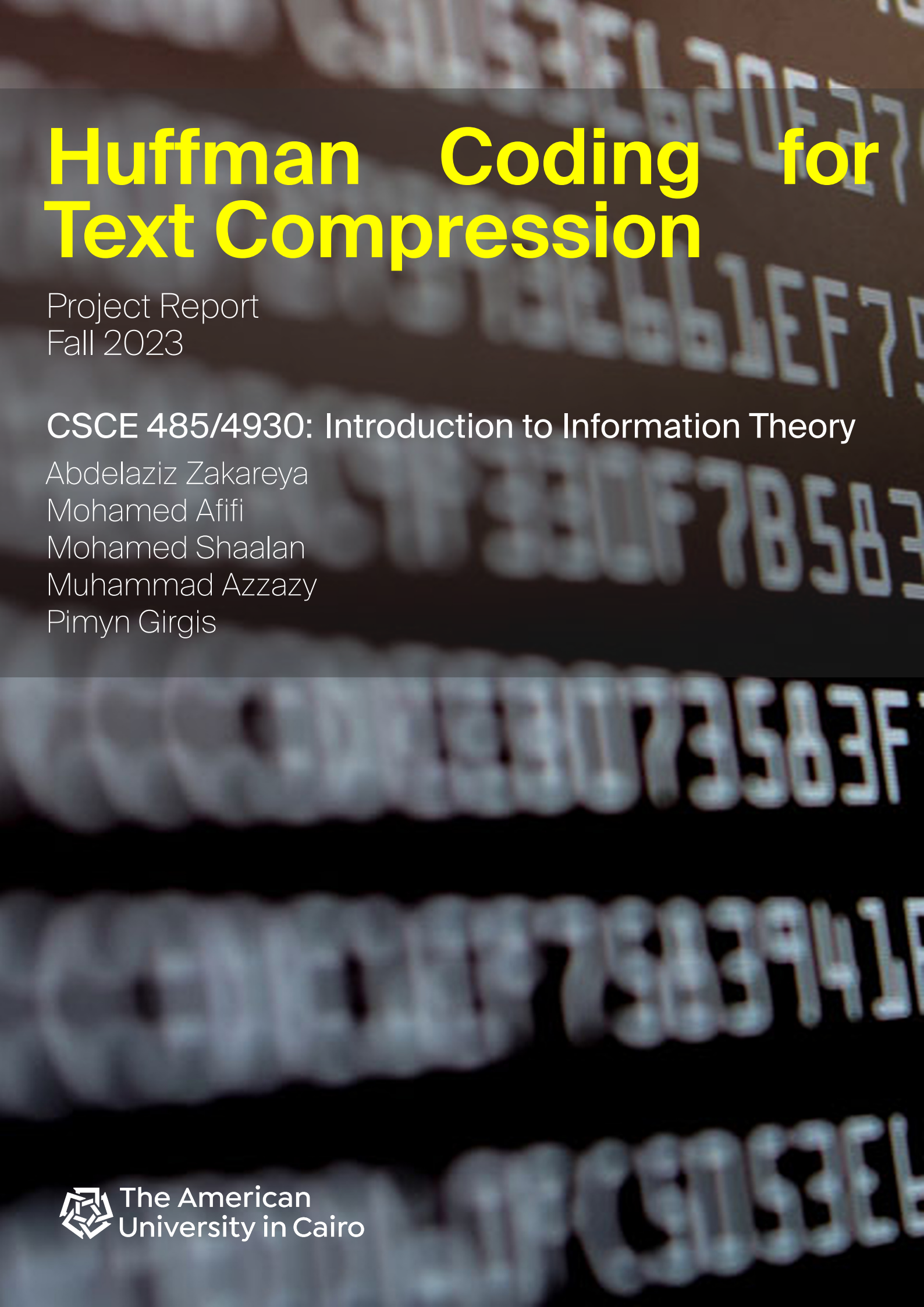


Huffman Coding for Text Compression

Project Report
Fall 2023

CSCE 485/4930: Introduction to Information Theory

Abdelaziz Zakareya
Mohamed Afifi
Mohamed Shaalan
Muhammad Azzazy
Pimyn Girgis

The American University in Cairo

Huffman Coding for Text Compression

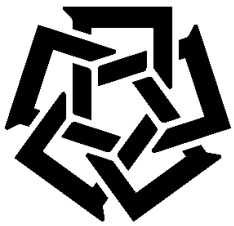
Project Report
Fall 2023

by

Abdelaziz Zakareya
Mohamed Afifi
Mohamed Shaalan
Muhammad Azzazy
Pimyn Girgis

Student Name	Student Number
Afifi	900191017
Azzazy	900202821
Girgis	900213066
Shaalan	900201539
Zakareya	900203361

Instructor:	Professor Amr Goneid
Teaching Assistant:	Mr. Mohamed Ibrahim
Project Deadline:	11thDecember, 2023
School:	School of Sciences and Engineering, AUC
Department:	Computer Science and Engineering Department



Abstract

The purpose of this project is to comprehend and implement the Huffman coding algorithm as an effective mechanism for text file compression. The report includes a brief historical perspective on Huffman coding. Additionally, it incorporates the definition of the issue of text file compression using Huffman coding. The project report includes methodologies for text file compression using Huffman coding and the Huffman coding algorithm. The algorithm and data specifications are incorporated into this report. It also contains some experimental results associated with the Huffman coding algorithm. An analysis and critique of the algorithms to be used are included in the report. *Keywords: Huffman coding, Static coding, Text file compression, Lossless compression, Entropy, Coding efficiency, Compression ratio*

Contents

1

Introduction

"The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point." –Claude Shannon

Today, Huffman coding is the best and most crucial binary coding. In 1951, David Huffman and Robert Fano attended a conference about effective binary coding. In the beginning, it appeared that David Huffman could not handle his topic. However, Huffman thought about using binary trees for his coding that he constructed from the leaves to the root. He proved that this is the coding with the highest efficiency [1]. In 1952, David Huffman designed a greedy algorithm for optimal coding in binary notation. Huffman coding is a method of compressing data in a lossless fashion by analyzing the frequency of various values and giving shorter codes to more frequent values. It utilizes optimal merge patterns to build a binary tree. In a Huffman tree, the leafs are the symbols. The paths from the root to the leaves may not be the same. This implies that Huffman codes have varying lengths. Symbols with higher frequencies reside in leaves closer to the root. This indicates that symbols with higher frequencies have shorter code lengths. Symbols with lower frequencies reside in leaves farther from the root. This means that symbols with lower frequencies have longer code lengths. Data compression is concerned with diminishing the quantity of space required to store data or decreasing the time taken to transmit data. The data size is diminished by discarding extra information. The aim of compression is to put a source in digital representation with the fewest possible number of bits while simultaneously satisfying the condition of reconstruction of the source. If it is possible to precisely reconstruct the source from the compressed form, then the data compression is said to be lossless. When the source data is vital, lossless data compression is used [2].

2

Problem Definition

The goal of this project is to analyze and implement the Huffman coding algorithm as an efficient technique for text file compression. While classical compression algorithms exist, the aim of this project is understanding the concepts of Huffman coding and assessing its effectiveness in diminishing the size of textual data. Some debugging had to be done to guarantee the that the Huffman coding algorithm is working properly. Originally, when the input text file was empty, a run-time error was generated. Additionally, the presence of whitespace characters such as the carriage return, line feed, horizontal and vertical tabs, form feed, and space in the input text file produced erroneous results following the decompression of the compressed binary file.

3

Methodology

The goal of this project is to use Huffman coding to compress text files. We used the Python programming language to implement text file compression using Huffman coding. The project is implemented in a single source file called main.py.

An auxiliary function implements a binary to decimal converter. It takes a binary string as parameter and returns an integer value which is the decimal value after conversion.

Another helper function implements the reverse - a decimal to binary converter. It takes a decimal value as an input and returns a binary string which is the binary value following conversion.

There is a function that implements writing a binary byte to an output file. This function takes two parameters: a string which represents the byte to be written and a reference to the output file stream. The function that implements the binary to decimal conversion is called first to convert the input binary string to a decimal value. Then, the function writes the decimal value as a binary byte to the provided output file stream.

Another function implements the computation of the probability table. This function takes a single input which is the path. An unordered map is utilized to store the probabilities indexed by their corresponding characters. A check is done to ensure that there are no errors when opening the file for reading. Then, the function reads every character from the input file sequentially and increments the count for every character in the probability map. Moreover, the function increments a variable to track the total number of characters. The function then iterates over the unordered map and divides each number of characters by the value of the variable that represents the total number of characters. This effectively calculates the probability of every character. The file is then closed and the unordered map is returned.

Another function implements getting the binary codes from the Huffman tree. This function has five inputs: a pointer to a node, a character indicating whether the node is a right node or a left node, a code representing what was generated so far, and an unordered map storing the Huffman code for every leaf node. The stopping condition is when the input node is null. The code is updated by appending a 0 for a left child and 1 for a right child. The function then calls itself recursively for the left child with the new code. A leaf node indicates a symbol in the Huffman tree. A leaf node's code and data are stored in the unordered map. The function finally calls itself recursively for the right child with new code.

The encoding function takes two parameters: the codes as an unordered map and the path of the output text file. An input file stream is opened to read the original text file. An output file stream is opened to write the binary data to a binary file. If the input file fails to open, an error message is output to the console. An outer loop reads the characters of the input text file sequentially. The inner loop adds the Huffman code of the character to a string and checks if the string has reached a length of 8 bits. If the string has reached a length of 8 bits, it is written to the output file by calling the function which writes a byte to a file. If there are remaining bits in the final byte, the binary string is padded until it reaches a length of 1 byte. This is done using the Huffman code representing the space character. The input and output streams are then closed indicating the end of the encoding process.

The decoding function does the reverse of the encoding function. It has the responsibility of decoding a compressed binary file using a given Huffman coding table. It has four parameters: an unordered map which represents the Huffman coding table, a string which represents the path to the compressed binary file, and a string representing the path to the decompressed text file. Some file handling is done to open two file

streams: one for reading from the compressed binary file and the other for writing to the decompressed text file. If the input stream is not open, it prints an error message. The decoding function reads the input binary file one byte at a time into an integer variable. The integer variable is converted to its binary representation by calling the auxiliary function which performs that task. Then, it loops over each digit of the binary representation and adds it to a string variable. If that code is found in the unordered map representing the Huffman table, the corresponding character is written to the output text file and the string is reset. Finally, the input and output streams are closed marking the end of the decoding process.

A function that generates statistics associated with Huffman coding calculates the coding efficiency and compression ratio. This function takes a pair of unordered maps as input and returns a vector of floating point numbers. The variables representing the entropy and the code length are initialized to zero. The unordered map for the probabilities is iterated over. Then, the entropy is calculated using the probabilities. The code length is obtained by multiplying the code length by the probability. The coding efficiency is calculated by dividing the entropy by the average code length. The code-to-message ratio (CMR) is obtained by dividing the code length by the number of bits per byte. A vector which contains the coding efficiency and the CMR is returned.

Another function reads a table from a file and fills an unordered map with associations between binary strings and symbols. The input file stream is opened. Then, it checks whether the input file opens without errors. Each line is read sequentially and a string stream is created for each line to obtain the tokens. In the first case, if the token is the first one, the character data is extracted. In the second case, if it is the second token, the token is associated with the character data in the unordered map. The file is closed after reading indicating that the process is complete.

The most important function is the one that implements the Huffman coding algorithm. The function has three parameters: the path to the input file for compression or the output file for decompression, the path of the output file for compression, and a character indicating whether the process is compression or decompression. The function first checks if the character parameter implies compression has to be performed. The function that gets the probabilities is invoked to compute the probabilities of each character in the input file. A priority queue is declared, and each character with the corresponding probability is added as a leaf to the priority queue. The Huffman tree is built by iteratively merging a pair of nodes with the lowest probabilities until only the root is left. A variable is used to store the resulting Huffman tree. The function that obtains the codes is invoked to produce the Huffman codes for every symbol. If the process is decompression, the Huffman codes are read into the unordered map. The function that performs decoding is called to execute decompression and the output is written to the text file. In the case of compression, the coding efficiency and the compression ratio are printed to the console.

4

Specification of Algorithms to be used

In plain English, the Huffman coding algorithm works as follows:

- Each symbol is stored in a parentless node of a binary tree (BT).
- The symbols are inserted with their corresponding probabilities in a min heap on probabilities.
- Loop
  - The lowest two probabilities are discarded from the heap.
  - The symbols with these lowest probabilities are combined to create a new symbol. The probability of this new symbol is the sum of the probabilities of the individual symbols.
  - The symbols are then stored in a parentless node with two children.
  - The probability of the symbols with the lowest probabilities and its symbol are inserted into the heap.

  until all symbols are combined into a single final symbol.
- The path from the root node to each symbol (leaf) is traced to form the binary string for the symbol. A zero is concatenated for a left branch and a one is concatenated for a right branch.
- The code of the symbol is returned.

In summary, the Huffman coding algorithm works as follows:

1. Sort the symbols in decreasing order of probabilities.
2. Repeat
   - Merge the lowest two symbols into a virtual symbol.
   - Rearrange in decreasing order.

   until all symbols are merged into one.

We obtain a tree where the leaves are symbols, and the internal nodes are virtual symbols (merges). Trace from the root to leaf (symbol) to get the code. A left is represented by a zero and a right by a one. The process of extracting the minimum frequency from the priority queue occurs $2 * (n - 1)$ times, and the complexity is $O(logn)$. Therefore, the time complexity of Huffman coding is $O(nlogn)$.

5

Data Specifications

For compression, the input data file is an ordinary text file with American Standard Code for Information Interchange (ASCII) characters. For decompression, the input data file is a binary file.

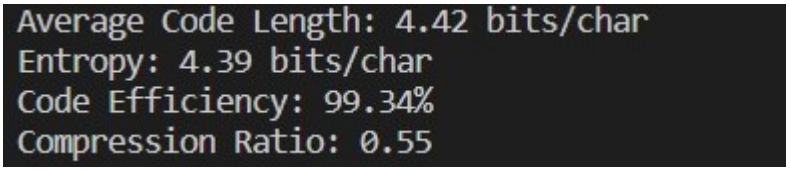
The following are the most important data structures used in the implementation of the Huffman coding algorithm:

1. HeapNode class for representing a node in the Huffman tree. Each node has the following data:
    - char: for storing the character read from the file.
    - freq: the frequency of that character
    - left: pointer to the left child node of the current node
    - right: pointer to the right child node of the current node
2. Heap: array (list) of heap nodes, used primarily for keeping track of the Huffman tree from the initial step till the final step after merging all nodes.
3. Frequency: dictionary storing each character of the text file along with its corresponding frequency
4. Codes: dictionary storing each character of the text file along with its corresponding Huffman code
5. Reverse_mapping: dictionary maps the Huffman-generate code for each character of the text file with its corresponding character for decoding.

6

Experimental Results

We tested the following input example:
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.



The average code length associated with this example is approximately 4.42 bits per symbol (bps), and the entropy of the input example is about 4.39 bps. The coding efficiency of the input example is 99.34%, and the compression ratio is 0.55. Additionally, the compressed binary file was obtained without errors. The table of input codes was also obtained correctly where a code is assigned to each character in the input text file.

7

Analysis and Critique

The output results show that the compressed file size is approximately half the size of the input text file. Additionally, the average code length is very close to the entropy. Thus, the coding efficiency is approaching 100%. In this particular case, Huffman coding is an excellent algorithm for text file compression. Huffman coding is an example of static coding where two passes are required. One pass is required to calculate the probabilities of the symbols and to determine the mapping. Another pass is needed for encoding the symbols. Additionally, Huffman coding achieves the entropy if and only if the probabilities of the symbols are powers of two. When utilizing Huffman coding with bit strings, no compression is accomplished even though the entropy is low. Huffman coding produces a coding efficiency of about 19% for binary strings which is effectively no compression. In Huffman coding, the probability distribution of the source must be established. Huffman coding is static because the code has to be recalculated if any symbol probability is altered. The Huffman encoder and decoder must be cognizant of the encoding tree. There are other methods for compression which are not static. In Run Length Encoding (RLE), for example, we do not calculate the probabilities at all. In RLE, we achieved a good compression ratio if there is a stream of zeros and ones. RLE can be used to encode anything and not necessarily binary data. However, sometimes RLE is not efficient due to the fluctuations of ones and zeros. Other dynamic coding methods include the Lempel-Ziv (LZ) and Lempel-Ziv-Welch (LZW) coding methods. These adaptive ource coding methods are capable of compressing binary strings unlike Huffman coding.

8

Conclusions

We can state that our program is efficient in terms of compressing text files. Huffman coding suffers from some issues because it is a static form of coding where two passes are necessary to complete the compression process. Additionally, it fails in compressing binary strings. However, adaptive source coding methods such as RLE, LZ, and LZW coding methods do not depend on the probability distributions of the source. These adaptive coding methods are capable of achieving compression for binary strings. One of the possible future goals of this project might include investigating how well Huffman coding performs in contrast to other text file compression methods such as the dynamic coding methods mentioned earlier. This can be done by computing and analyzing certain metrics such as the compression ratio and the coding efficiency.

References

[1] O. Manz. *Well packed - not a bit too much: Compression of digital data explained in an understandable way*. Weisbaden, Germany: Springer Fachmedien Wiesbaden GmbH, 2021.

[2] R. Lourdusamy S. Shanmugasundaram. "A Comparative Study Of Text Compression Algorithms". In: *International Journal of Wisdom Based Computing* (2011).

Acknowledgements

In the end, we thank God for helping us to produce this project. We are grateful for the efforts of Professor Amr Goneid for his clear and concise instructions about the project and its topic. Furthermore, we appreciate the support of his graduate teaching assistant, Eng. Mohamed, and his recommendations concerning the project.

Appendix

```
1
2  import heapq
3  import os
4  from collections import defaultdict, Counter
5  import math
6  input_codes_path = "input_codes.txt"
7
8  class HuffmanCoding:
9      # Initializing the data structures
10     def __init__(self):
11         self.heap = []
12         self.frequency = {}
13         self.codes = {}
14         self.reverse_mapping = {}
15
16     # HeapNode Class for Huffman Coding Algorithm
17     class HeapNode:
18         def __init__(self, char, freq):
19             self.char = char
20             self.freq = freq
21             self.left = None
22             self.right = None
23
24         def __lt__(self, other):
25             return self.freq < other.freq
26
27         def __eq__(self, other):
28             if not other:
29                 return False
30             if not isinstance(other, self._class_):
31                 return False
32             return self.freq == other.freq
33
34     # Main function used for compressing text files
35     def compress(self, input_path):
36         filename, file_extension = os.path.splitext(input_path)
37         output_path = filename + ".bin"
38
39         with open(input_path, 'r+') as file, open(output_path, 'wb') as output:
40             text = file.read()
41
42             # Handling in case the text file is empty
43             if not text:
44                 output.write(b'\n')
45                 print("Input text is empty. Nothing to compress.")
46                 return output_path
47
48             frequency = self.make_frequency_dict(text)
49             self.make_heap(frequency)
50             self.merge_nodes()
51             self.make_codes()
52
53             encoded_text = self.get_encoded_text(text)
54             padded_encoded_text = self.pad_encoded_text(encoded_text)
55             b = self.get_byte_array(padded_encoded_text)
56
57             # Write the binary compressed file
58             output.write(bytes(b))
59
60             # Output the input table for decompression
61             self.output_codes(input_codes_path)
62
```

```
            # Calculate and print the metrics
            avg_code_length, entropy, efficiency, compression_ratio = self.calculate_metrics(text)
            print(f"Average Code Length: {avg_code_length:.2f} bits/char")
            print(f"Entropy: {entropy:.2f} bits/char")
            print(f"Code Efficiency: {efficiency:.2%}")
            print(f"Compression Ratio: {compression_ratio:.2f}")

        return output_path

    # Extract the frequency of each character in the text
    def make_frequency_dict(self, text):
        self.frequency = Counter(text)
        return self.frequency

    # Inserting Nodes in the heap
    def make_heap(self, frequency):
        for key in frequency:
            node = self.HeapNode(key, frequency[key])
            heapq.heappush(self.heap, node)

    # Merging Nodes
    def merge_nodes(self):
        while len(self.heap) > 1:
            node1 = heapq.heappop(self.heap)
            node2 = heapq.heappop(self.heap)
            merged = self.HeapNode(None, node1.freq + node2.freq)
            merged.left = node1
            merged.right = node2
            heapq.heappush(self.heap, merged)

    # Extract the codes from the Huffman Tree
    def make_codes(self):
        root = heapq.heappop(self.heap)
        current_code = ""
        self.make_codes_helper(root, current_code)

    # Recursive helper function to extract codes from the tree
    def make_codes_helper(self, root, current_code):
        if root is None:
            return

        if root.char is not None:
            self.codes[root.char] = current_code
            self.reverse_mapping[current_code] = root.char

        self.make_codes_helper(root.left, current_code + "0")
        self.make_codes_helper(root.right, current_code + "1")

    # Get the binary encoded text
    def get_encoded_text(self, text):
        encoded_text = ""
        for character in text:
            encoded_text += self.codes[character]
        return encoded_text

    # Pad encoded text
    def pad_encoded_text(self, encoded_text):
        extra_padding = 8 - len(encoded_text) % 8
        for i in range(extra_padding):
            encoded_text += "0"
        padded_info = "{0:08b}".format(extra_padding)
        encoded_text = padded_info + encoded_text
        return encoded_text

    # Get byte array to write the binary compmressed file
    def get_byte_array(self, padded_encoded_text):
        if len(padded_encoded_text) % 8 != 0:
            print("Encoded text not padded properly")
            exit(0)

```

```
        b = bytearray()
        for i in range(0, len(padded_encoded_text), 8):
            byte = padded_encoded_text[i:i+8]
            b.append(int(byte, 2))
        return b

    # Outputting each character of the text file along with its code to be used for decommpression
    def output_codes(self, output_path):
        with open(output_path, 'w') as file:
            for char, code in self.codes.items():
                if char == "\n" :
                    char = "\\n"
                elif char == " ":
                    char = "\\s"
                file.write(f"{char} {code}\n")

    # Calculating Compression Metrics
    def calculate_metrics(self, text):
        total_length = sum(len(self.codes[char]) * freq for char, freq in self.frequency.items())
        total_chars = sum(self.frequency.values())
        average_code_length = total_length / total_chars
        probabilities = [freq / total_chars for freq in self.frequency.values()]
        entropy = -sum(p * math.log2(p) for p in probabilities)
        efficiency = entropy / average_code_length if average_code_length else 0
        compression_ratio = average_code_length / 8

        return average_code_length, entropy, efficiency, compression_ratio

    # Main function for decompressing text files
    def decompress(self, input_path, input_codes_path):
        filename, file_extension = os.path.splitext(input_path)
        output_path = filename + "_decompressed" + ".txt"

        with open(input_path, 'rb') as file, open(output_path, 'w') as output:
            bit_string = ""

            byte = file.read(1)
            while byte:
                byte = ord(byte)
                bits = bin(byte)[2:].rjust(8, '0')
                bit_string += bits
                byte = file.read(1)
            encoded_text = self.remove_padding(bit_string)
            decompressed_text = self.decode_text(encoded_text, input_codes_path)
            output.write(decompressed_text)

        return output_path

    # Remove padding of the encoded text
    def remove_padding(self, padded_encoded_text):
        padded_info = padded_encoded_text[:8]
        extra_padding = int(padded_info, 2)

        padded_encoded_text = padded_encoded_text[8:]
        encoded_text = padded_encoded_text[:-1*extra_padding]

        return encoded_text

    # Decode text
    def decode_text(self, encoded_text, input_codes_path):
        current_code = ""
        decoded_text = ""

        with open(input_codes_path, "r") as file:
            for line in file:
                char, code = line.strip().split()
                self.reverse_mapping[code] = char

        for bit in encoded_text:
```

```
            current_code += bit
            if current_code in self.reverse_mapping:
                character = self.reverse_mapping[current_code]
                if character == "\\n":
                    character = "\n"
                elif character == "\\s":
                    character = " "
                decoded_text += character
                current_code = ""

        return decoded_text

path = "input.txt"
h = HuffmanCoding()
compressed_path = h.compress(path)
decompressed_path = h.decompress("input.bin", input_codes_path)
```