TEXT FILE COMPRESSION USING HUFFMAN CODING

NUR SYAZWANI BINTI ZAINON

UNIVERSITI TEKNIKAL MALAYSIA MELAKA

TEXT FILE COMPRESSION USING HUFFMAN CODING

NUR SYAZWANI BINTI ZAINON

This report is submitted in partial fulfilment of the requirements for the degree of Bachelor of Computer Engineering Technology (Computer Systems) with Honours

Faculty of Electronics and Computer Technology and Engineering

Universiti Teknikal Malaysia Melaka

2025

UNIVERSITI TEKNIKAL MALAYSIA MELAKA
FAKULTI TEKNOLOGI DAN KEJURUTERAAN ELEKTRONIK DAN KOMPUTER

BORANG PENGESAHAN STATUS LAPORAN
PROJEK SARJANA MUDA II

Tajuk Projek	:	TEXT FILE COMPRESSION USING HUFFMAN CODING
Sesi Pengajian	:	2024/2025

Saya NUR SYAZWANI BINTI ZAINON mengaku membenarkan laporan Projek Sarjana Muda ini disimpan di Perpustakaan dengan syarat-syarat kegunaan seperti berikut:

1. Laporan adalah hakmilik Universiti Teknikal Malaysia Melaka.
2. Perpustakaan dibenarkan membuat salinan untuk tujuan pengajian sahaja.
3. Perpustakaan dibenarkan membuat salinan laporan ini sebagai bahan pertukaran antara institusi pengajian tinggi.
4. Sila tandakan (✓):

☐	**SULIT***	(Mengandungi maklumat yang berdarjah keselamatan atau kepentingan Malaysia seperti yang termaktub di dalam AKTA RAHSIA RASMI 1972)
☐	**TERHAD***	(Mengandungi maklumat terhad yang telah ditentukan oleh organisasi/badan di mana penyelidikan dijalankan.
☒	**TIDAK TERHAD**	

Disahkan oleh:

TANDATANGAN PENULIS) (COP DAN TANDATANGAN PENYELIA)

Alamat Tetap: ....
....
....

Tarikh : 12 Januari 2025 Tarikh : 12 Januari 2025

*CATATAN: Jika laporan ini SULIT atau TERHAD, sila lampirkan surat daripada pihak berkuasa/organisasi berkenaan dengan menyatakan sekali tempoh laporan ini perlu dikelaskan sebagai SULIT atau TERHAD.

DECLARATION

I declare that this project report entitled "Text File Compression Using Huffman Coding" is the result of my own research except as cited in the references. The project report has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Signature	:	
Student Name	:	NUR SYAZWANI BINTI ZAINON
Date	:	12 January 2025

APPROVAL

I hereby declare that I have checked this project report and in my opinion, this project report is adequate in terms of scope and quality for the award of the degree of Bachelor of Computer Engineering Technology (Computer Systems) with Honours.

Signature	:	
Supervisor Name	:	TS. DR. ROSTAM AFFENDI BIN HAMZAH
Date	:	12 January 2025

Signature	:	
Co-Supervisor Name (if any)	:	
Date	:	

DEDICATION

This project is dedicated to my beloved parents, who have always helped and motivated me throughout my academic career. Your unfailing confidence in my skills has served as a constant source of inspiration and motivation for me.

To my friends and classmates, thank you for your companionship, collaboration, and for making this journey enjoyable and educational. The times we spent together and your support have been priceless.

To my supervisor and mentors, I sincerely appreciate your commitment to my growth and the effort you have spent on teaching and guiding me. Because my academic and professional goals have been greatly shaped by the advice, wisdom, and support of you all.

Finally, I dedicate this project to all the final year students who work hard and overcome obstacles in order to realize their goals. I hope that my effort will demonstrate the value of endurance, hard work, and dedication.

ABSTRACT

With the amount of data growing exponentially in the digital era, text file efficiency in transmission and storage is a major challenge. This research investigates the application of Huffman Coding, a well-known lossless data compression technology, for text file compression. The goal is to create a method that keeps the original information intact while drastically reducing the size of text files. Huffman Coding assigns variable-length binary codes to characters based on their frequencies, which ensures that more often occurring characters have shorter codes and less frequently occurring characters have longer codes. By utilizing the unique prefix rule, which forbids any code from becoming a prefix of another, this technique minimizes the overall file size and removes any potential for ambiguous decoding. The project is to create a Huffman Tree using a min-heap data structure, analyse the frequency of characters in a text file, and create Huffman codes for every character. The text file's characters are then changed using these codes, producing a compressed output that uses less storage space. Significant file size reductions, data integrity preservation, and real-world examples of the useful uses of Huffman Coding are among the anticipated results. In addition to improving data transmission speed and storage efficiency, this research seeks to give a greater grasp of the theoretical and practical elements of lossless text file compression. The system's performance will be evaluated through extensive testing and assessment, opening the door for potential future improvements in text compression methods. This study emphasizes how crucial it is to handle data well and provides a solid answer to the problems associated with organizing massive amounts of text data.

ABSTRAK

Dengan jumlah data yang semakin bertambah pesat di dalam era digital ini, kecekapan pemindahan dan penyimpanan fail teks menjadi cabaran utama. Penyelidikan ini adalah untuk menyiasat aplikasi Pengekodan Huffman, teknologi pemampatan data tanpa kehilangan yang terkenal, untuk pemampatan fail teks. Matlamatnya adalah untuk mencipta kaedah yang mengekalkan maklumat asal sambil mengurangkan saiz fail teks secara drastik. Pengekodan Huffman memberikan kod binari panjang berubah kepada watak berdasarkan kekerapan mereka, yang memastikan bahawa watak yang lebih kerap muncul mempunyai kod yang lebih pendek dan watak yang kurang kerap muncul mempunyai kod yang lebih panjang. Dengan menggunakan peraturan awalan unik, yang melarang mana-mana kod daripada menjadi awalan kepada kod yang lain, teknik ini meminimumkan saiz fail keseluruhan dan menghapuskan sebarang potensi kekeliruan dalam penyahkodan. Projek ini adalah untuk mencipta Pokok Huffman menggunakan struktur data min-heap, menganalisis kekerapan watak dalam fail teks, dan mencipta kod Huffman untuk setiap watak. Watak dalam fail teks kemudian diubah menggunakan kod ini, menghasilkan output mampat yang menggunakan ruang penyimpanan yang lebih sedikit. Pengurangan saiz fail yang ketara, pemeliharaan integriti data, dan contoh dunia nyata mengenai penggunaan berguna Pengekodan Huffman adalah antara hasil yang dijangka. Selain meningkatkan kelajuan pemindahan data dan kecekapan penyimpanan, penyelidikan ini bertujuan untuk memberikan pemahaman yang lebih mendalam tentang aspek teori dan praktikal pemampatan fail teks tanpa kehilangan. Prestasi sistem akan dinilai melalui ujian dan penilaian yang meluas, membuka peluang untuk penambahbaikan masa hadapan dalam kaedah pemampatan teks. Kajian ini menekankan betapa pentingnya pengurusan data yang baik dan menyediakan penyelesaian kukuh kepada masalah yang berkaitan dengan menguruskan sejumlah besar data teks.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my supervisor, **TS. DR. ROSTAM AFFENDI BIN HAMZAH** for his precious guidance, words of wisdom and patience throughout this project.

I am also indebted to University Technical Malaysia Melaka (UTeM) for the financial support which enables me to accomplish the project. Not forgetting my fellow colleague, for the willingness of sharing their thoughts and ideas regarding the project.

My highest appreciation goes to my parents, and family members for their love and prayer during the period of my study.

Finally, I would like to thank all the staff at the **FACULTY OF ELECTRONICS AND COMPUTER TECHNOLOGY AND ENGINEERING**, fellow colleagues and classmates, the Faculty members, as well as other individuals who are not listed here for being co-operative and helpful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

0	-	Binary digit zero
1	-	Binary digit one

LIST OF ABBREVIATIONS

HTML	-	HyperText Markup Language
CSS	-	Cascading Style Sheets
GIF	-	Graphics Interchange Format
JPEG	-	Joint Photographic Experts Group
MP3	-	MPEG Audio Layer III
MPEG	-	Moving Picture Experts Group
RLE	-	Run-Length Encoding
LZW	-	Lempel-Ziv-Welch
PC	-	Personal Computer
ZIP	-	Zip file format for data compression

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Background

Text files may take up a lot of storage space. Examples include HTML, JavaScript, CSS, and plain text (.txt). Compressing these files is essential to enhance storage efficiency and facilitate quicker transmission over networks. Huffman Coding is a useful technique that ensures the best possible compression by assigns variable-length binary codes to characters based on their frequencies [1].

1.2 Societal and Technological Impact

There are several significant advantages of using Huffman Coding for text file compression for both society and technology.

I. With compressed text files, it will take up less space, which means our computers and servers use less storage. This helps reduce the energy consumption of data centres, which is good for the environment [1].

II. Because they consume less bandwidth, smaller files will be transmitted over the internet more quickly. This outcome will be leads to faster data transfer which improve the efficiency and smoothness of online operations.

III. File size reduction will decrease the cost of storage and bandwidth. For example businesses and organizations can make a financial savings on data storage and internet usage, freeing up funds for other crucial investments.

IV. Compressed text files enable larger text file handling applications to operate more quickly and fluidly. Better user experiences and more effective software and website functioning are the outcomes of this.

V. As a lossless compression technique, Huffman Coding shrinks files without erasing any data. This is essential to guaranteeing the complete restoration of the original data, which is essentials for accuracy and security.

1.3 Problem Statement

As the amount of digital information keeps growing rapidly, handling and sending vast amounts of text data has become a major difficulty. Text files, including HTML, JavaScript, CSS, and plain text (.txt), are used in both personal and professional environments. These files can take up a lot of storage space and bandwidth, which increases expenses, slows down data transmission, and causes inefficiencies in data handling and storage.

The primary issue is that uncompressed text files are inefficient. They take up too much space and could decrease data access and transfer rates. This inefficiency is especially troublesome for enterprises and organizations that manage massive databases and need quick, dependable access to information.

Huffman Coding is a way for solving this problem by compressing text files to make them smaller while retaining all contents. It works by assigning shorter codes to commonly used characters and longer codes to uncommon characters, reducing the total file size.

The aim is to create a system that efficiently compresses text files using Huffman Coding. This system has to drastically decrease file sizes, minimize storage and transmission costs, and enhance data handling, all while preserving the original data.

1.4 Project Objective

The purpose of this project is to create a system for compressing text files using Huffman Coding. The objectives of this system are:

a) To utilize the Huffman Coding algorithm for text file compression.

b) To reduce the amount of storage required for bin files by making them smaller, hence saving space on PCs and servers.

c) To verify the performance of the algorithm.

1.5 Scope of Project

This project will provide a system to compress text files using Huffman Coding, concentrating on the following main areas:

a) The project will deal with standard text files such as HTML, JavaScript, CSS, and plain text (.txt).

b) Use the Huffman Coding technique to reduce text file size by assigning shorter codes to frequently used characters and longer codes to less frequent ones.

c) Create a user-friendly tool or system that can:

    I. Read text files and record how frequently each character appears.

    II. Create a Huffman Tree that generates codes for each character.

d) Test the program on various text files to observe how effectively it compresses them.

e) To determine efficacy, compare the sizes of the original and compressed files.

f) Create a simple interface where users can pick files, compress them, and save the compressed versions.

g) Provide explicit instructions about how to utilize the tool.

h) Include a report on the tool's performance and result of it.

i) The project will only compress text files, not decompress those or deal with other file formats such as photos or videos. The utility can handle files of typical size, but it may not perform effectively with really big files or real-time compression.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter provides an overview of the current research and knowledge on text file compression, specifically emphasizing Huffman Coding. The objective is to comprehend the existing body of knowledge and findings in this field. This review aims to enhance our comprehension of the theory and use of Huffman Coding by examining multiple research, algorithms, and real-world implementations.

Text file compression is crucial due to the perpetual expansion of digital data. Utilizing efficient compression techniques aids in conserving store capacity, minimizing transmission duration, and enhancing data administration. Huffman Coding is a widely used method of lossless compression that is renowned for its efficient data compression capabilities while preserving all information [1]. In a study by Andysah Putera Utama Siahaan, the Huffman Text Compression Technique was detailed, explaining its efficiency in reducing file sizes without data loss [2].

Initially, we will examine the historical progression of data compression techniques, illustrating their evolution throughout the years [2] [3]. A notable contribution in this area is the study by D. Jasmine Shoba and Dr. S. Sivakumar, which provides a comprehensive analysis of Huffman Coding algorithms and their applications in data compression [4]. This study enhances our understanding of the theoretical foundations and practical implementations of Huffman Coding.

Next, we shall elucidate the functioning of Huffman Coding, encompassing its fundamental concepts and the methods for its implementation. In addition, we will do a comparative analysis of Huffman Coding in relation to various compression algorithms in order to emphasize its advantages and disadvantages [5].

In addition, we will examine several implementations of Huffman Coding in fields such as web development, software engineering, and data transfer. Through the examination of case studies and research, we will observe the practical applications of Huffman Coding in real-world scenarios [1].

The purpose of this chapter is to provide a concise overview of the existing knowledge on text file compression using Huffman Coding. Additionally, it will identify any deficiencies in the existing research and propose potential avenues for future investigation. This assessment will establish a strong basis for the development and execution of our proprietary text file compression system.

2.2 The historical development of data compression techniques

Advancements in data compression strategies have seen substantial development over time, propelled by the necessity to effectively store and transport data. Initial techniques were rather uncomplicated, although they established the groundwork for more advanced algorithms such as Huffman Coding. Gaining an understanding of these advancements helps to position Huffman Coding in the broader perspective of the progression of data compression [4] [6] [7].

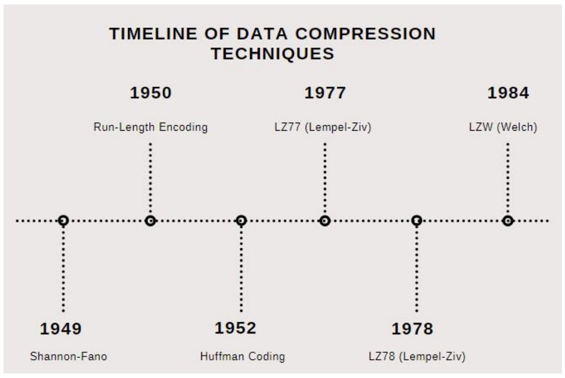


Figure 1: Timeline of Data Compression Techniques

Primitive Methods

I. Run-Length Encoding (RLE) is a data compression technique.

RLE, or Run-Length Encoding, is a straightforward technique that compresses data by substituting repeated characters with a single character and its corresponding count. As an illustration, the string "AAAABBBCCDAA" is transformed into "4A3B2C1D2A" [5] [4].

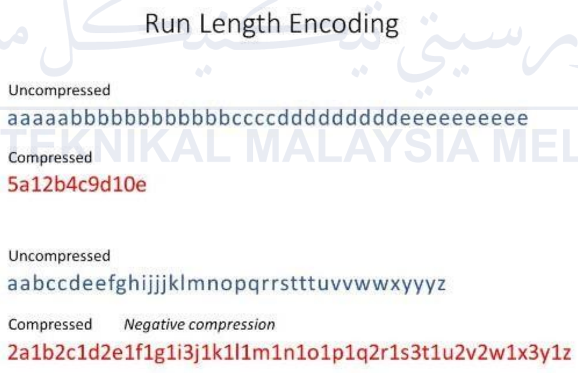

Run Length Encoding
Uncompressed
aaaaabbbbbbbbbbbbccccdddddddddeeeeeeeeee
Compressed
5a12b4c9d10e
Uncompressed
aabccdeefghijjjklmnopqrrstttuvvwwxyyyz
Compressed
Negative compression
2a1b2c1d2e1f1g1i3j1k1l1m1n1o1p1q2r1s3t1u2v2w1x3y1z

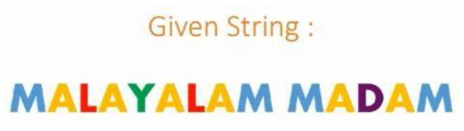
Figure 2: Example of Run-Length Encoding

Applications: It is effective for data sets that have a high number of recurring elements, such as basic photographs.

Drawbacks: The Run-Length Encoding (RLE) method is not efficient for text files that have limited repetition, as it may not significantly decrease the file size.

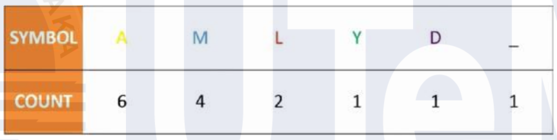
II. The Shannon-Fano coding technique:

The method, devised by Claude Shannon and Robert Fano, involves assigning variable-length codes to characters based on their frequency of occurrence. Characters that appear often are assigned shorter codes [3] [6].


Given String :

MALAYALAM MADAM

Figure 3: String for Shanon Fano example



SYMBOL	A	M	L	Y	D	_
COUNT	6	4	2	1	1	1

Figure 4: Summary of the string

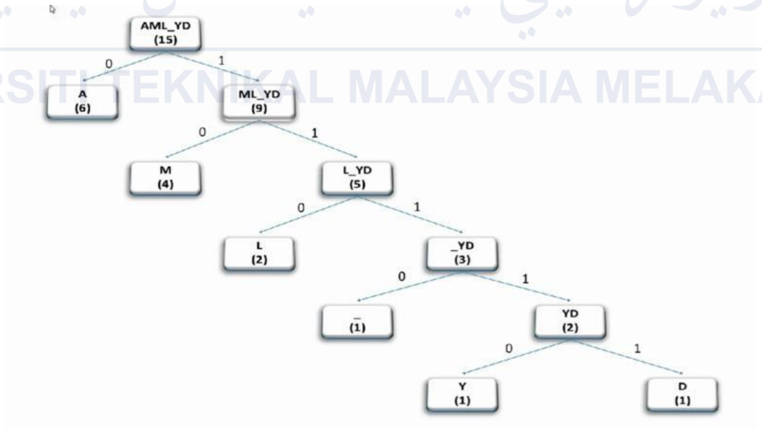


Figure 5: Shannon-Fano Tree

The process: the process will be involves sorting characters based on their frequency and then creating a binary tree to assign codes.

Drawbacks: Although it surpasses certain previous techniques, it is not as efficient as Huffman Coding.

2.3 Huffman Coding

Huffman Coding, devised by David A. Huffman in 1952, represented a significant breakthrough. Huffman's technique achieves optimality by generating the most efficient variable-length codes for a given set of character frequencies. The Huffman Coding method constructs a binary tree in which each character is assigned a code according to its frequency. Characters that appear often are assigned shorter codes [2] [4].

The process of Huffman Coding involves the following steps:

a) Determine the frequency of occurrence for each individual character in the given text.

b) Place characters and their frequencies into a priority queue.

c) Merge the two characters with the lowest frequencies into a new node and continue this process until only one node remains, which will serve as the root of the tree.

d) Encode characters with binary codes according to their position in the tree.

e) Subsequent advancements following the Huffman coding technique.

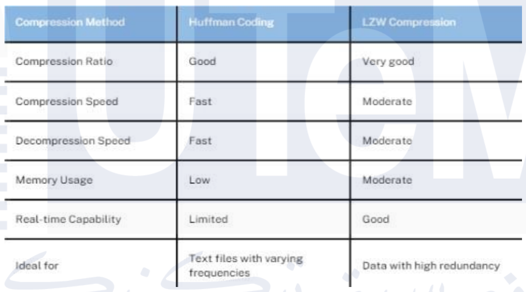
   I. Lempel-Ziv (LZ) algorithms:

   The LZ77 and LZ78 algorithms, developed by Abraham Lempel and Jacob Ziv, employ a dictionary to substitute recurring patterns within the data. Extensively utilized in compression utilities and file formats such as ZIP and GIF [4].

II. The Lempel-Ziv-Welch (LZW) algorithm.

The concept refers to an enhanced iteration of LZ78, which was created by Terry Welch. Utilized in file formats such as GIF. The LZW method constructs a dictionary of sequences based on the input data and substitutes sequences with individual dictionary references [6].

Table 1: Huffman vs LZW



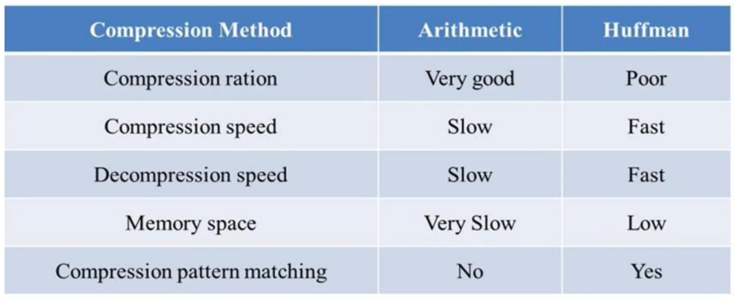
Compression Method	Huffman Coding	LZW Compression
Compression Ratio	Good	Very good
Compression Speed	Fast	Moderate
Decompression Speed	Fast	Moderate
Memory Usage	Low	Moderate
Real-time Capability	Limited	Good
Ideal for	Text files with varying frequencies	Data with high redundancy

III. Arithmetic Coding

Arithmetic Coding is a concept that replaces predefined codes with ranges of integers to represent whole messages. This can result in higher compression ratios. Utilized in scenarios that require optimal compression efficiency, particularly in multimedia applications [3].

Arithmetic coding has the advantage of increased efficiency for specific data types. Also can enables the representation of probabilities that are not whole numbers, resulting in improved compression. The disadvantages of this method is more intricate and less efficient compared to Huffman Coding and also requires meticulous execution to prevent numerical complications [3].

Table 2: Comparison of Huffman vs Arithmetic



Compression Method	Arithmetic	Huffman
Compression ration	Very good	Poor
Compression speed	Slow	Fast
Decompression speed	Slow	Fast
Memory space	Very Slow	Low
Compression pattern matching	No	Yes

2.4 Introduction to the Core Principles and Execution of Huffman Coding

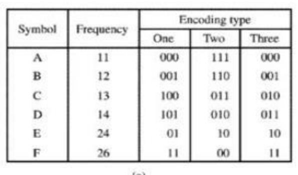
Huffman Coding is a prevalent algorithm utilized for the compression of data without any loss. David A. Huffman introduced it in 1952 as a component of his term thesis during his time as a Ph.D. student at MIT. This technique employs shorter codes for frequently occurring characters and longer codes for less often occurring ones, resulting in a reduction in data size [2] [8].

2.4.1 Basic Principles

a) Variable-Length Coding:

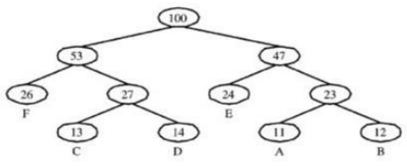
Huffman Coding assigns variable lengths of codes to characters dependent on their frequency of occurrence. More commonly occurring characters are assigned shorter codes, while less commonly occurring characters are assigned longer codes [2].

For instance, in a text with many occurrences of the letter 'e', it could be represented by a concise code such as '01', whereas a less common character like 'z' might be assigned a longer code such as '1101'.



Symbol	Frequency	Encoding type		
		One	Two	Three
A	11	000	111	000
B	12	001	110	001
C	13	100	011	010
D	14	101	010	011
E	24	01	10	10
F	26	11	00	11

(a)



(b)

Figure 6: Example Huffman Tree

b) Prefix-Free Codes:

Huffman Coding guarantees that no code is a prefix of another code. This implies that every code can be distinctly identifiable during the decoding process [3]. For example, if '01' is designated as a code, no other code will commence with '01', hence guaranteeing unambiguous decoding [9].

The process of Huffman coding involves the following steps:

I. Calculation of frequency:

First, determine the frequency of occurrence for each individual character in the given text. Second, generate a comprehensive inventory of every individual character and its corresponding frequency [5] as shown in the figure below.

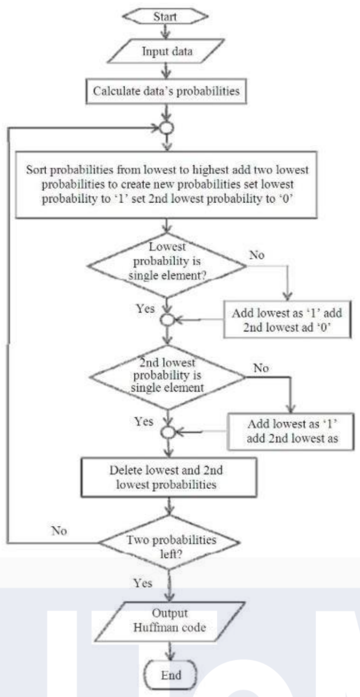

Start
Input data
Calculate data's probabilities
Sort probabilities from lowest to highest add two lowest probabilities to create new probabilities set lowest probability to '1' set 2nd lowest probability to '0'
Lowest probability is single element?
No
Yes
Add lowest as '1' add 2nd lowest ad '0'
2nd lowest probability is single element
No
Yes
Add lowest as '1' add 2nd lowest as
Delete lowest and 2nd lowest probabilities
No
Two probabilities left?
Yes
Output Huffman code
End

Figure 7: Huffman encoder flow chart

II. Creating a Priority Queue:

Insert the characters and their frequencies into a priority queue, which is a specialised data structure that arranges elements in ascending order based on their values. This facilitates the quick identification and utilization of the least common characters [3].

III. Construction of Huffman Tree:

Merge the two characters with the lowest frequencies to create a new node with their combined frequency. Continue iterating this procedure, merging nodes, until there is just a single node left, which will be the root. A tree is created in which the binary code for each character is determined by its route from the root to its leaf node [5].

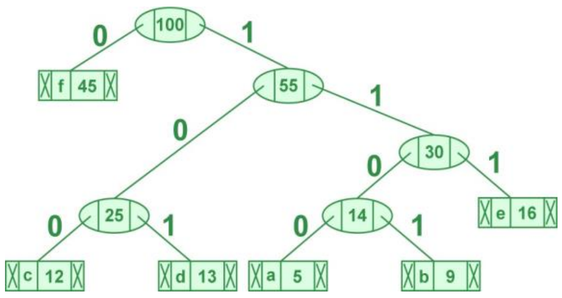


Figure 8: Huffman Tree Construction Example

IV. Generation of code:

Allocate binary codes by traversing the tree from the root to each leaf node. The label '0' is assigned to each left edge, while the label '1' is assigned to each right edge. As an illustration, the route to reach the letter 'e' could be described as left-left-right, resulting in a code such as '001' [3].

V. Data encoding:

Convert each character in the text into its respective binary representation. This process generates a condensed rendition of the text, occupying a smaller amount of storage capacity [2].

2.4.2 Benefits and Constraints

a) Benefits:

Huffman Coding is highly efficient in compressing file sizes for the characters it examines [2].

I. The approach is easily comprehensible and may be executed without difficulty [5].

II. Lossless compression is a method of reducing the size of data without sacrificing any information, allowing for the complete and accurate reconstruction of the original data [3].

b) Constraints:

The approach necessitates two iterations over the data—one to calculate frequencies and another to encode the text [5].

I. Not necessarily the optimal choice because in certain scenarios, other techniques such as arithmetic coding may achieve superior data compression [3].

II. If the data undergoes frequent changes, a static Huffman tree may not be as optimal. Adaptive Huffman coding can provide assistance, albeit it entails greater complexity [5].

2.4.3 Real-World uses of Huffman Coding

Huffman Coding is widely employed in numerous applications due to its effectiveness and user-friendly nature:

I. WinRAR and 7-Zip utilise Huffman Coding in their file compression algorithms [5].

II. JPEG images employ Huffman Coding to compress specific sections of the image data [3].

III. Minimising the data volume transmitted via the internet conserves bandwidth and accelerates transmission [2].

2.5 Comparative Analysis of Compression Algorithms

This section provides a comparison between Huffman Coding and other widely used compression methods, aiming to emphasize the advantages and disadvantages of Huffman Coding. Gaining a comprehensive understanding of these contrasts will offer a more distinct understanding of the reasons and circumstances in which Huffman Coding is employed, as well as its respective benefits and constraints.

a) Lempel-Ziv-Welch (LZW) is a method used for compressing data.

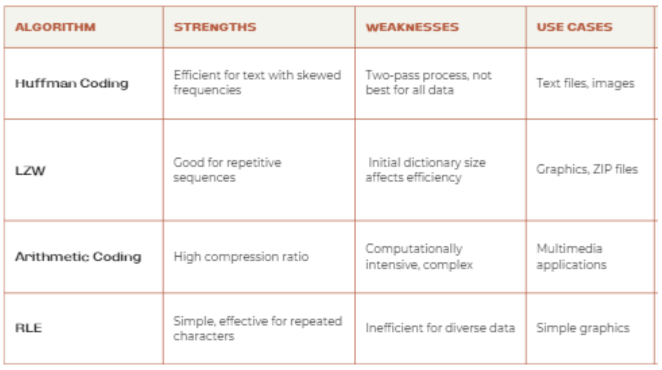
LZW is a compression technique that employs a dictionary to replace sequences of characters with shorter codes. When sequences are encountered, they are added to a dictionary. When a repetitive sequence occurs, it is replaced with a reference to the appropriate term in the dictionary [7].

I. Benefits: The LZW technique is particularly effective in compressing data that has a large number of repetitive sequences. It is often used in graphical file formats such as GIF and in compression software like ZIP.

II. Constraints: LZW may not exhibit the same level of efficiency as Huffman Coding when applied to text material that lacks significant redundancy. Additionally, the initial size of the dictionary can impact the effectiveness of compression.

The LZW algorithm compresses a file that contains repeating text, such as "ABABABABAB", by identifying the recurring sequence "AB" and replacing it with shorter codes.

Table 3: Comparison Table of Compression Algorithms



ALGORITHM	STRENGTHS	WEAKNESSES	USE CASES
Huffman Coding	Efficient for text with skewed frequencies	Two-pass process, not best for all data	Text files, images
LZW	Good for repetitive sequences	Initial dictionary size affects efficiency	Graphics, ZIP files
Arithmetic Coding	High compression ratio	Computationally intensive, complex	Multimedia applications
RLE	Simple, effective for repeated characters	Inefficient for diverse data	Simple graphics

b) Arithmetic coding.

Arithmetic Coding distinguishes itself from Huffman Coding by encoding complete messages as intervals of values ranging from 0 to 1. Instead than allocating predetermined binary codes to each individual character, it converts a sequence of characters into a singular integer within a specified range [3] [10]. Arithmetic Coding has the advantage of achieving superior compression ratios compared to Huffman Coding, especially for material that has intricate probability distributions. It effectively manages fractional probabilities, resulting in improved compression for specific data types [3]. Arithmetic Coding exhibits certain weaknesses, namely its higher computational intensity and complexity compared to Huffman Coding. The intricacy of this can result in slower encoding and decoding procedures and necessitates meticulous execution to prevent any precise issues [3]. Arithmetic Coding is a method that can generate a more concise representation of a text with different character frequencies. It achieves this by employing ranges instead of preset codes.

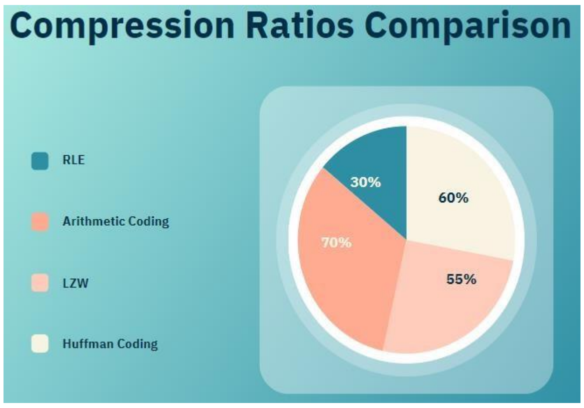


Figure 9: Compression Ratios Comparison

c) Run-Length Encoding (RLE) is a data compression technique.

I. Benefits: RLE, or Run-Length Encoding, is a straightforward compression technique that decreases the data size by substituting consecutive instances of the same character with a single character followed by the number of times it repeats. RLE is highly efficient for data that contains extended sequences of repeated characters, such as monochrome photographs or uncomplicated graphic files [5].

II. Constraints: The Run Length Encoding (RLE) method is not effective for text data or files with considerable variability, as it does not result in a significant reduction in file size for such data [5].

In an image containing a series of adjacent black pixels, the Run-Length Encoding (RLE) method would compress the sequence "00000000" to "8_0". In this compressed representation, "8" represents the count of consecutive occurrences and "0" represents the repeated character.

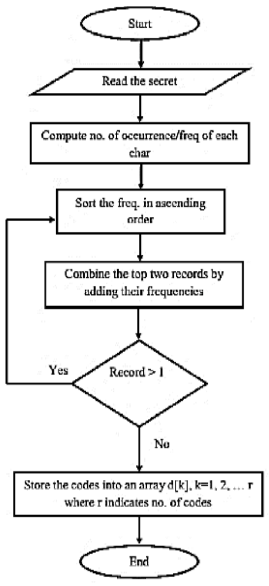

Start
Read the secret
Compute no. of occurrence/freq of each char
Sort the freq. in ascending order
Combine the top two records by adding their frequencies
Record > 1
Yes
No
Store the codes into an array d[k], k=1, 2, ... r where r indicates no. of codes
End

Figure 10: flowchart of Huffman coding

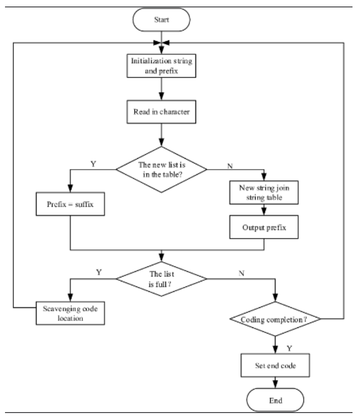


Figure 11: flowchart of LZW

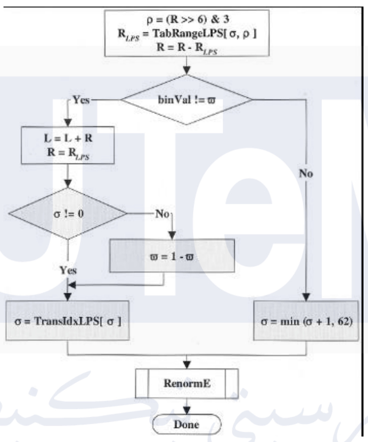


Figure 12: flowchart of Arithmetic coding

Summary of Comparison:

Huffman Coding is particularly effective for text files that have imbalanced frequency distributions, resulting in great efficiency. LZW is particularly efficient for files containing repetitive sequences, whereas Arithmetic Coding provides superior compression for data with intricate probabilities. Run-length encoding (RLE) is particularly well-suited for data that contains numerous sequences of repeated characters.

Huffman Coding achieves a harmonious equilibrium between simplicity and efficiency. The LZW and RLE algorithms are generally uncomplicated, although they may not always provide optimal compression. Arithmetic Coding, although it has the potential to be more efficient, is characterised by increased complexity and decreased execution speed.

Huffman Coding is extensively employed in various applications such as text compression, web data transmission, and image formats like JPEG. The LZW algorithm is widely employed in the compression of GIF pictures and ZIP files. Arithmetic Coding is utilized in multimedia compression, particularly in situations when achieving high efficiency is of utmost importance. Run-length encoding (RLE) is commonly employed in basic graphics and certain picture formats.

2.6 Real-world Applications of Huffman Coding

The efficiency and ease of implementation of Huffman Coding make it a widely utilized technique in numerous applications. This section will examine several practical applications of Huffman Coding, showcasing its efficacy and utility in diverse fields.

a) File compression tools:

File compression applications, like WinRAR and 7-Zip, utilize Huffman Coding as part of their compression algorithms to reduce file sizes, hence improving storage and transfer efficiency. These software employ compression methods to encode commonly appearing characters with shorter codes, leading to a decrease in the overall size of the files.

WinRAR use Huffman Coding, along with other techniques, to achieve remarkable compression ratios. WinRAR is compatible with several file formats and is widely used for creating compressed archives, which allow for easy storing and sharing. 7-Zip is a widely used file compression tool that incorporates Huffman Coding as part of its LZMA (Lempel-Ziv-Markov chain algorithm) compression method. This application is widely recognized for its outstanding compression ratios and comprehensive support for numerous file formats, making it an extremely adaptable choice for all file compression needs [5].

b) Image Compression:

Huffman Coding is an essential element of the JPEG image format, used to efficiently compress image data. During the process of JPEG compression, the image is divided into blocks, and each block is subjected to a transformation and quantization. Afterwards, Huffman Coding is utilized to compress the quantized values, effectively reducing the file size without affecting the image quality. JPEG compression, created by the Joint Photographic Experts Group, employs Huffman Coding to encode the quantized coefficients derived from applying a discrete cosine transform (DCT) to the image blocks. This approach efficiently reduces the amount of data required to represent the image, resulting in smaller file sizes without compromising on quality [3].

c) Transmission of Data:

Reducing the quantity of data sent over the internet is essential for preserving bandwidth and accelerating transmission speeds. Huffman Coding achieves this by compressing text-based data, such as HTML, CSS, and JavaScript files, commonly utilized in web development, hence reducing their size [11]. Huffman Coding is employed to compress text files during the transmission of web pages over the internet, hence minimizing the volume of data that needs to be transmitted. As a consequence, this leads to quicker loading speeds for web pages and enhances the overall user experience. Web browsers and servers can utilize Huffman Coding to compress HTTP headers and content, hence enhancing the efficiency of data transfer [2].

d) Applications of multimedia:

Huffman Coding is utilized in diverse multimedia applications, including audio and video compression. Huffman Coding facilitates the effective storage and transmission of multimedia content by lowering the size of audio and video files. The MP3 audio format employs Huffman Coding as a component of its compression strategy for encoding audio data. This process aids in diminishing the file size of audio files without substantially impacting the sound quality, hence facilitating the storage and sharing of music and other audio content [11]. MPEG Video Compression utilizes Huffman Coding to compress remaining data following motion compensation and transform coding. This minimizes the data needed to depict the video, facilitating effective storage and streaming [3].

e) Compression of documents:

Huffman Coding is employed in document compression formats such as PDF and PostScript to effectively diminish the file size of textual and graphical content. PDF files commonly employ Huffman Coding to compress text and graphics, thereby reducing file size and facilitating sharing. This is particularly crucial for extensive documents that require transmission over the internet or storage on devices with restricted storage capacity. PostScript files utilize Huffman Coding to compress both text and graphics, similar to PDF files. Compressing files enhances the efficiency of printing and sharing documents by reducing the amount of storage space and bandwidth they use [5].

f) Communication protocols used in computer networks:

Huffman Coding is employed in diverse network protocols to enhance data transmission efficiency and minimize delay. Huffman Coding enhances the efficiency of data communication over networks by compressing protocol headers and payloads. The HTTP/2 protocol utilizes Huffman Coding to compress HTTP headers, facilitating the delivery of web pages and other internet resources. This decreases the dimensions of the headers, leading to enhanced data transmission speed and improved web performance. QUIC, which stands for Quick UDP Internet Connections, is a network protocol developed by Google. It use Huffman Coding to compress the headers of the protocol. This aids in decreasing latency and enhancing the efficiency of online applications [3].

2.7 Existing Areas of Research that Require Further Investigation and Present Difficulties

Although Huffman Coding is widely used and effective, there are still research gaps and problems that must be solved in order to enhance its performance and broaden its usefulness. This section delineates the existing deficiencies and proposes prospective avenues for future investigation.

a) Real-Time Compression refers to the process of reducing the size of data in real-time, without any delay or latency.

Huffman Coding often necessitates performing two passes over the data—one to compute the frequencies of each letter and another to encode the text. This can be suboptimal for real-time applications where data is constantly generated and requires on- the-fly compression.

Although static Huffman Coding is effective for fixed datasets, it is not as appropriate for streaming data that undergoes dynamic changes [3].

Future research should focus on the development of adaptive Huffman coding algorithms that can efficiently handle real-time data compression. Adaptive Huffman Coding dynamically modifies the Huffman tree while processing fresh data, enabling real-time compression. Research can concentrate on optimizing these methods to minimize computing burden and enhance performance [12].

b) Integration with Other Techniques:

Although Huffman Coding is generally efficient for many data formats, it may not consistently achieve the highest compression ratios when compared to alternative methods such as arithmetic coding or dictionary-based algorithms [5].

Presently, certain hybrid compression algorithms amalgamate Huffman Coding with other methods, yet there is still scope for enhancing efficiency and compression ratio [3]. Future research should focus on investigating hybrid models that combine the advantages of several compression approaches in order to enhance overall performance. By combining Huffman Coding with Lempel-Ziv (LZ) algorithms or arithmetic coding, it is possible to achieve improved compression ratios and faster processing times. Research should prioritize the identification of optimal combinations for different types of data and the enhancement of the integration of these methods [16].

c) Optimization of energy usage

The computational intricacy of Huffman Coding can result in increased energy use, particularly in large-scale data centres where efficient power usage is crucial [5]. Although Huffman Coding is economical in terms of computing, the requirement for frequent tree updates and encoding can lead to higher energy consumption, especially when dealing with huge datasets [3].

Exploring methods to enhance the algorithm's energy economy without compromising compression effectiveness. Possible approaches include the development of more streamlined data structures for the Huffman tree, the optimisation of algorithms for traversing the tree, or the utilisation of hardware accelerators to enhance the efficiency of compression activities. Further research could investigate adaptive methods that optimise the trade-off between energy usage and compression performance, taking into account the specific workload.

2.7.1 Scalability

Scalability refers to the ability of a system or process to handle an increasing amount of work or data without sacrificing performance or efficiency.

With the ongoing increase in data volumes, the ability of Huffman Coding to handle larger amounts of data becomes more crucial. The method should possess the capability to efficiently process big datasets without experiencing substantial performance deterioration [3].

Presently, Huffman Coding is capable of managing datasets of moderate size. However, its efficiency may decline as the quantity of the data increases [5].

Future research should focus on the development of scalable Huffman coding algorithms capable of efficiently processing extremely huge datasets. This may entail utilizing parallel processing techniques, distributed computing frameworks, or enhanced data structures that decrease the temporal complexity of constructing and navigating the Huffman tree.

Research might potentially prioritize the optimization of the method for particular categories of extensive datasets, such as big data applications or high-resolution multimedia information.

2.7.2 Compression Efficiency across Varied Data Types:

Huffman Coding is particularly efficient for text and some forms of binary data, but its effectiveness may vary for other types of data that have distinct statistical characteristics [5].

Although Huffman Coding is commonly employed, it may not always deliver the most efficient compression for data that has highly variable or non-uniform distributions [3]. Exploring methods to enhance the efficiency of Huffman Coding for various types of data. One possibility is to create tailored Huffman trees that take into account the unique attributes of the data. Another option is to combine Huffman Coding with machine learning models to forecast the most effective coding strategies for certain datasets. Further research could investigate context-aware compression techniques that dynamically adjust the Huffman tree in response to evolving patterns in the input.

2.7.3 Ensuring the safety and strength of the system:

It is of utmost importance to guarantee the security and resilience of compressed data, particularly in situations when the preservation of data integrity and confidentiality are of vital significance [3].

The current state is that Huffman Coding alone does not offer security, but it can be combined with encryption and error-correction methods to improve data protection [5]. Future research should focus on the development of compression techniques that are both secure and robust. These schemes should integrate Huffman Coding with encryption algorithms and error-correction codes. This may entail developing integrated frameworks that guarantee the compactness and security of compressed data, protecting it from unauthorized access and transmission faults. Research might also prioritize evaluating the influence of these integrated strategies on compression efficiency and performance.

2.8 The Application of Colour in User Interface Design

2.8.1 Colour Psychology in User Interface Design

Colours are essential in user interface (UI) design as they affect users' visual perception, focus, and emotional reactions. Elliot and Maier (2014) conducted a study demonstrating that particular colours can elicit distinct psychological responses. Green is frequently linked to balance, harmony, and stability, whereas more vibrant shades like neon green evoke a futuristic and contemporary aesthetic. This hue is commonly employed in technology to highlight essential components such as buttons or significant icons. [13]

2.8.2 Effectiveness of Bright Colours in UI/UX

Witzel and Gegenfurtner (2018) discovered that vibrant colours, such as neon green, improve the visibility of screen elements, especially in low-light environments or when immediate prominence is required. Neon green is recognised for its significant contrast with dark backgrounds, rendering it an exceptional option for enhancing readability and visual clarity in applications like websites and interactive systems. [14]

2.8.3 Implementation of Neon Green in Technology

In HTML and CSS, neon green is commonly represented by the colour code HEX #39FF14. The W3C's official documentation (2023) emphasises that CSS offers versatility in colour manipulation, rendering it an effective instrument for personalising user interface designs. Moreover, this hue is frequently employed in interactive initiatives to augment user experience. [15]

2.9 Conclusion

The literature review has presented a thorough examination of text file compression, specifically emphasizing Huffman Coding. Through a review of its historical progression, core concepts, practical applications, comparison with other algorithms, and existing research gaps, we have acquired a comprehensive comprehension of the importance of Huffman Coding and its function in data compression.

2.9.1 Essential Highlights Summary

a) The historical development of data compression techniques: We examined the progression of data compression strategies, emphasizing the developments from basic approaches such as Run-Length Encoding and Shannon-Fano Coding to more intricate algorithms like Huffman Coding. The inception of Huffman Coding by David A. Huffman in 1952 represented a noteworthy breakthrough in the realm of data compression, offering an ideal technique for producing variable-length codes based on the frequencies of characters [2].

b) Introduction to the Core Principles and Execution of Huffman Coding: Huffman Coding allocates shorter codes to frequently encountered characters and longer codes to less commonly encountered ones, guaranteeing efficient data compression without any loss of information. The procedure entails computing the frequencies of characters, establishing a priority queue, constructing a Huffman tree, producing codes, and encoding the data [4].

c) Comparative Analysis of Compression Algorithms: An analysis of Huffman Coding in comparison to other compression methods such as LZW, Arithmetic Coding, and Run- Length Encoding has revealed its advantages and disadvantages. Huffman

Coding is very effective for text files with imbalanced frequency distributions but may not consistently reach the optimal compression ratios compared to alternative approaches in some situations [4] [3].

d) Real-world Applications of Huffman Coding: We examined many practical implementations of Huffman Coding, encompassing its utilization in file compression utilities like WinRAR and 7-Zip, picture compression formats like JPEG, data transmission for web pages, and multimedia applications such as MP3 and MPEG. These implementations showcase the adaptability and efficacy of Huffman Coding in diminishing data size and enhancing storage and transmission efficiency [3].

e) Existing Areas of Research that Require Further Investigation and Present Difficulties: Although Huffman Coding is widely used, it is also accompanied with various obstacles and limits. These requirements encompass real-time compression, integration of Huffman Coding with other methods, enhancement of energy efficiency, scalability, compression efficiency for various data kinds, and guaranteeing security and robustness. It is essential to address these areas of research that need attention in order to progress Huffman Coding and broaden its usefulness in different fields [4] [3].

2.9.2 Importance of Addressing Research Gaps:

a) Real-Time Compression: Enhancing the capabilities of Huffman Coding to efficiently compress real-time data would increase its suitability for applications that involve continuous data generation and require immediate processing.

b) Integration with Other Techniques: Investigating hybrid models that integrate Huffman Coding with other compression methods can result in enhanced overall performance and increased compression ratios, hence improving effectiveness across a broader spectrum of data types.
c) Optimization of energy usage: Efficiently optimizing Huffman Coding is crucial for power-sensitive environments such as large-scale data centers, where energy conservation is of utmost importance. This will contribute to enhancing the sustainability and cost-efficiency of data compression.
d) Scalability: Improving the scalability of Huffman Coding will allow it to effectively manage the increasing amounts of data, assuring its continued usefulness and applicability for large datasets and big data applications.
e) Compression Efficiency across Varied Data Types: Enhancing the efficiency of Huffman Coding for diverse data types will broaden its applicability, rendering it a more adaptable tool for a wide range of data compression jobs.
f) Ensuring the safety and strength of the system: By combining Huffman Coding with encryption and error-correction techniques, we may create compression schemes that are both secure and reliable. These schemes will guarantee that compressed data is compact and safeguarded against unauthorized access and transmission mistakes.

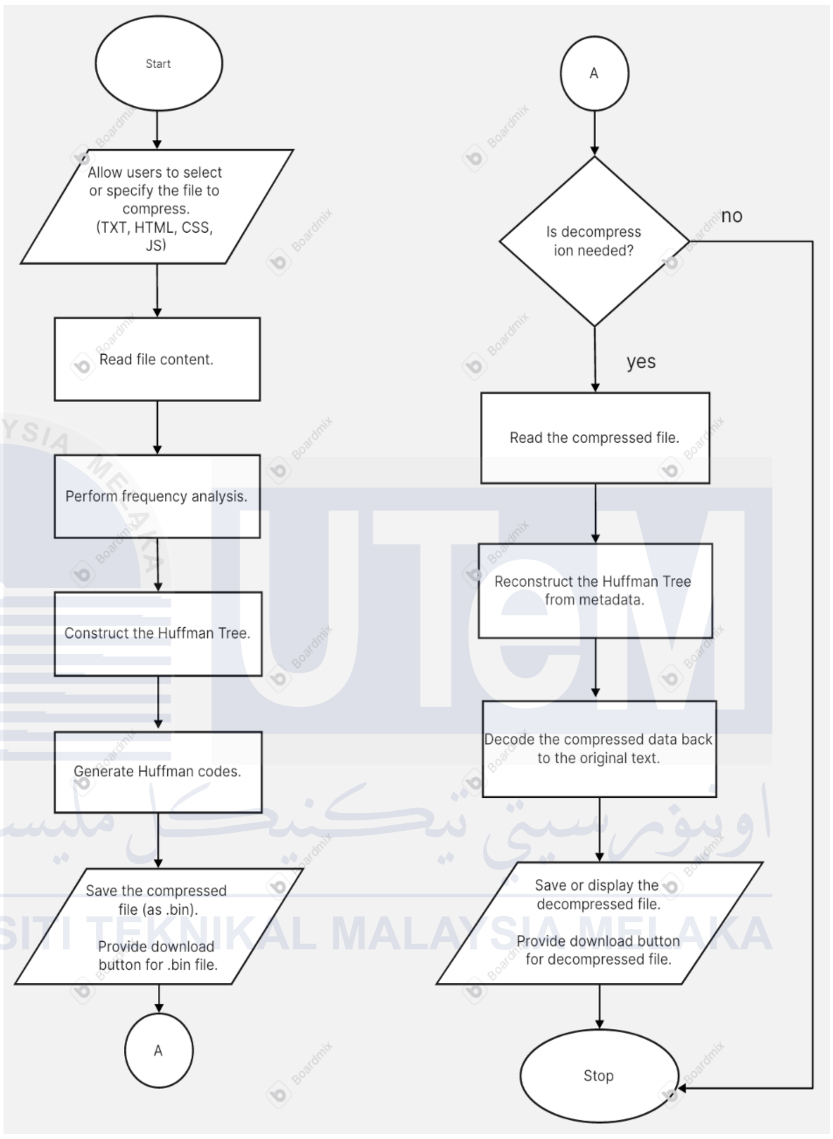
CHAPTER 3

METHODOLOGY

3.1 Introduction

This chapter presents the methodologies used to achieve the project's goals. The methodology involves the construction of a Huffman Coding mechanism for compressing text files, the development of the required software, and the evaluation of the system's efficiency.

3.2 Project Flowchart



Start
Allow users to select or specify the file to compress. (TXT, HTML, CSS, JS)
Read file content.
Perform frequency analysis.
Construct the Huffman Tree.
Generate Huffman codes.
Save the compressed file (as .bin). Provide download button for .bin file.
A
A
Is decompression needed?
no
yes
Read the compressed file.
Reconstruct the Huffman Tree from metadata.
Decode the compressed data back to the original text.
Save or display the decompressed file. Provide download button for decompressed file.
Stop

Figure 13: Project Flowchart

3.3 Visual Studio

Microsoft Visual Studio is one of the most popular integrated development environments, and it can support many languages and many different platforms. Visual Studio has a whole set of development tools for coding, debugging, and project management; hence, it is ideal for making and managing big software projects.

Visual Studio has been chosen for the implementation of this project because it possesses many superior attributes, including a strong debugging system that allows the easy combination of C++ and Python code, besides handling a few related projects under one solution.These capabilities were essential for implementing and testing the Huffman Coding algorithms efficiently.

This project utilised Visual Studio to:

a) To develop the compression and decompression algorithms in C++, respectively called HuffCompressProject and HuffDecompressProject.

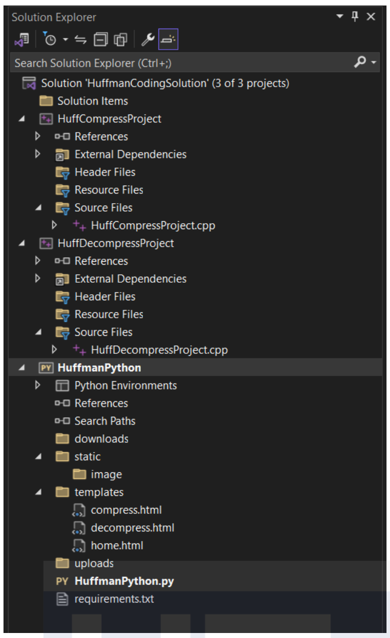
b) To implement a Python-based web interface-HuffmanPython project-which can easily be used by a user for compressing and decompressing files.

c) Oversee project dependencies and configurations to guarantee seamless compilation and execution of the system.

Figure 14: Visual Studio

3.3.1 Project Structure in Visual Studio

The Huffman Coding project was executed in Microsoft Visual Studio, utilising a solution architecture comprising three distinct projects. Each project was formulated to tackle specific functionalities within the system, encompassing compression, decompression, and web-based integration. I designated this project in Visual Studio as HuffmanCodingSolution.



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'HuffmanCodingSolution' (3 of 3 projects)
  Solution Items
  HuffCompressProject
    References
    External Dependencies
    Header Files
    Resource Files
    Source Files
      HuffCompressProject.cpp
  HuffDecompressProject
    References
    External Dependencies
    Header Files
    Resource Files
    Source Files
      HuffDecompressProject.cpp
  HuffmanPython
    Python Environments
    References
    Search Paths
    downloads
    static
      image
    templates
      compress.html
      decompress.html
      home.html
    uploads
    HuffmanPython.py
    requirements.txt
```

Figure 15: Structure of Visual Studio

a) HuffCompressProject.

This project executes the fundamental Huffman compression algorithm. The source file `HuffCompressProject.cpp` conducts frequency analysis, constructs the Huffman Tree, and encodes the input text into binary format. The resultant compressed data is recorded in a `.bin` file.

b) HuffDecompressProject.

This project contains the decompression algorithm, implemented in the source file `HuffDecompressProject.cpp`. The project includes the decompression algorithm, implemented in the source file `HuffDecompressProject.cpp`. The system reads the compressed `.bin` file, reconstructs the Huffman Tree utilising metadata, and decodes the binary data back into the original text.

c) HuffmanPython.

This project incorporates Python functionality to deliver an intuitive web interface for file compression and decompression.

The elements comprise:

I. 'HuffmanPython.py': The primary script that oversees the compression and decompression operations.

II. Templates ('compress.html', 'decompress.html', and 'home.html'): HTML documents designed for displaying web pages that facilitate user interaction with the system.

III. Folders:

'uploads': Stores user-uploaded files for compression or decompression. 'downloads': Contains the compressed .bin files and the decompressed files. 'static': Holds supporting assets, including an image that illustrates the project's workflow.

3.3.2 HuffCompressProject

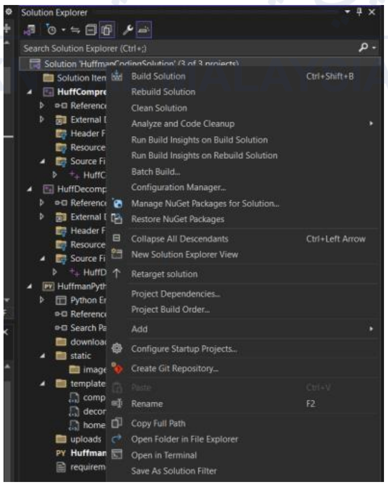


Figure 16: Solution Explorer

a) Ensure the file you want to compress is located in the uploads directory of the project. This is necessary for the program to access the file correctly.

b) Right-click on the solution name, ‘HuffmanCodingSolution (3 of 3 projects)’, in the Visual Studio Solution Explorer.

c) Select ‘Open in Terminal’ to open a terminal window at the project’s root directory.

d) Once the terminal shows the project path (e.g.,PSC:\Users\Dell\source\repos\HuffmanCodingSolution>), type the following command to navigate to the debug folder:

:- cd x64\debug

e) Use the following command to execute the HuffCompressProject.exe file. Replace the input and output file paths as needed:

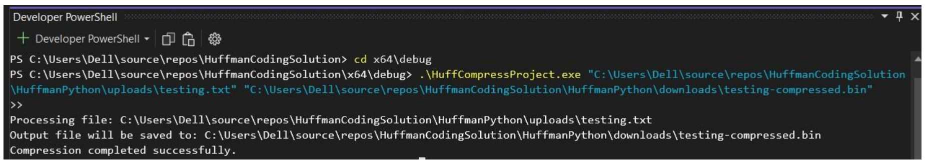
.\HuffCompressProject.exe

"C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\uploads\testing.txt"

"C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\downloads\testing-compressed.bin"

I. Explanation:

The first path ("C:\...\uploads\testing.txt") specifies the input file to be compressed. The second path ("C:\...\downloads\testing-compressed.bin") specifies the output location for the compressed .bin file.



```
PS C:\Users\Dell\source\repos\HuffmanCodingSolution> cd x64\debug
PS C:\Users\Dell\source\repos\HuffmanCodingSolution\x64\debug> .\HuffCompressProject.exe "C:\Users\Dell\source\repos\HuffmanCodingSolution
\HuffmanPython\uploads\testing.txt" "C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\downloads\testing-compressed.bin"
>>
Processing file: C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\uploads\testing.txt
Output file will be saved to: C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\downloads\testing-compressed.bin
Compression completed successfully.
```

Figure 17: HuffCompressProject

3.3.3 HuffDecompressProject

a) Ensure that the testing-compressed.bin file is located in the downloads folder of the project before running the program.

b) Right-click on the solution name ‘HuffmanCodingSolution (3 of 3 projects)’ in the Visual Studio Solution Explorer.

c) Select ‘Open in Terminal’ to open a terminal window in the project directory.

d) Right click on the Solution ‘HuffmanCodingSolution’(3 of 3 projects) and click open in terminal.

e) In the terminal, type the following command to navigate to the debug folder where the executable files are stored:

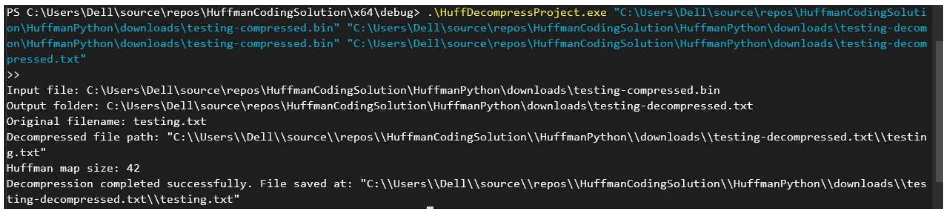
:- cd x64\debug

f) Execute the HuffDecompressProject.exe file with the appropriate input and output paths. Use the following command:

.\HuffDecompressProject.exe "C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\downloads\testing-compressed.bin" "C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\downloads\testing-decompressed.txt"

I. Explanation:

The first path ("C:\...\testing-compressed.bin") specifies the input file to decompress. The second path ("C:\...\testing-decompressed.txt") specifies the output location where the decompressed file will be saved.



```
PS C:\Users\Dell\source\repos\HuffmanCodingSolution\x64\debug> .\HuffDecompressProject.exe "C:\Users\Dell\source\repos\HuffmanCodingSoluti
on\HuffmanPython\downloads\testing-compressed.bin" "C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\downloads\testing-decom
on\HuffmanPython\downloads\testing-compressed.bin" "C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\downloads\testing-decom
pressed.txt"
>>
Input file: C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\downloads\testing-compressed.bin
Output folder: C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\downloads\testing-decompressed.txt
Original filename: testing.txt
Decompressed file path: "C:\\Users\\Dell\\source\\repos\\HuffmanCodingSolution\\HuffmanPython\\downloads\\testing-decompressed.txt\\testin
g.txt"
Huffman map size: 42
Decompression completed successfully. File saved at: "C:\\Users\\Dell\\source\\repos\\HuffmanCodingSolution\\HuffmanPython\\downloads\\tes
ting-decompressed.txt\\testing.txt"
```

Figure 18: HuffDecompressProject

3.3.4 HuffmanPython

To run this HuffmanPython section:

a) Right click on the HuffmanPython project in your IDE and select ‘set as startup project’.

b) After HuffmanPython project was set as startup project, right click again and click ‘open in terminal’.

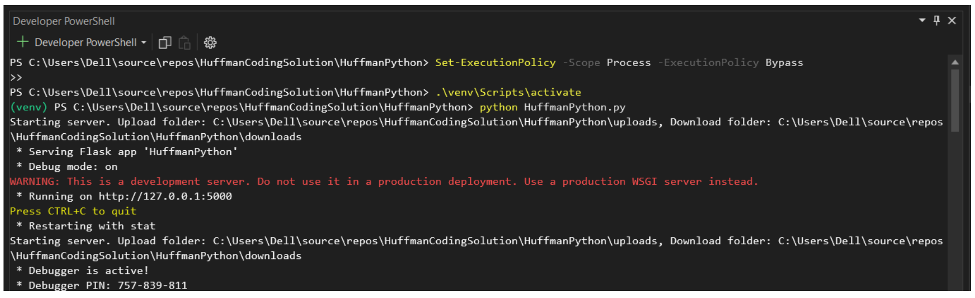
c) Once the terminal was open type the following sentence as shown in the figure below:

I. In the terminal, activate the virtual environment by typing:

‘Scripts\activate’

II. Run the application by typing:

‘python HuffmanPython.py’



```
PS C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython> Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
>>
PS C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython> .\venv\Scripts\activate
(venv) PS C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython> python HuffmanPython.py
Starting server. Upload folder: C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\uploads, Download folder: C:\Users\Dell\source\repos
\HuffmanCodingSolution\HuffmanPython\downloads
 * Serving Flask app 'HuffmanPython'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
Starting server. Upload folder: C:\Users\Dell\source\repos\HuffmanCodingSolution\HuffmanPython\uploads, Download folder: C:\Users\Dell\source\repos
\HuffmanCodingSolution\HuffmanPython\downloads
 * Debugger is active!
 * Debugger PIN: 757-839-811
```

Figure 19: HuffmanPython.py

d) Copy the URL http://127.0.0.1:5000 from the terminal and paste it into your browser to access the application.

e) Then the website will display as shown below:

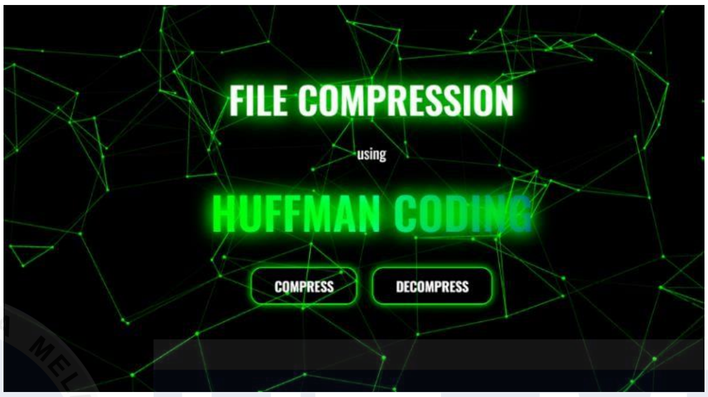


Figure 20: Homepage of the Huffman Coding website showing options for file compression and decompression

f) You can either start by compressing files and decompress them later, or you can decompress any .bin files first.

g) Press button Ctrl+C at your own pc if you want to stop the program from running.

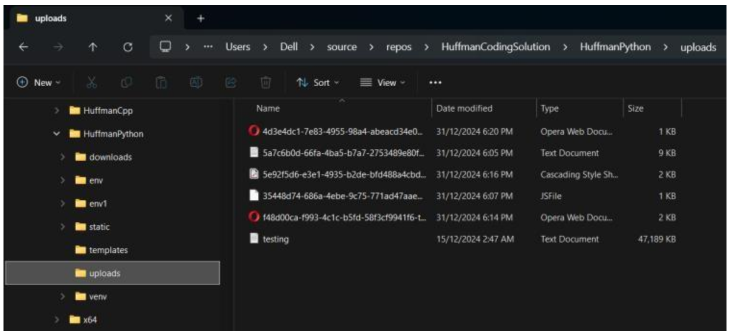
3.3.5 Folder Structure and Contents

a) Uploads folder.

I. The uploads folder contains the input files that are prepared for compression. These files are uploaded to the system before starting the compression process.

II. For instance, the file testing.txt is a text document stored here for testing the compression algorithm.

III. Figure 22 shows the contents of the uploads folder, including sample input files.



Name	Date modified	Type	Size
4d3e4dc1-7e83-4955-98a4-abeacd34e0...	31/12/2024 6:20 PM	Opera Web Docu...	1 KB
5a7c6b0d-66fa-4ba5-b7a7-2753489e80f...	31/12/2024 6:05 PM	Text Document	9 KB
5e92f5d6-e3e1-4935-b2de-bfd488a4cbd...	31/12/2024 6:16 PM	Cascading Style Sh...	2 KB
35448d74-686a-4ebe-9c75-771ad47aae...	31/12/2024 6:07 PM	JSFile	1 KB
f48d00ca-f993-4c1c-b5fd-58f3cf9941f6-t...	31/12/2024 6:14 PM	Opera Web Docu...	2 KB
testing	15/12/2024 2:47 AM	Text Document	47,189 KB

Figure 21: Compress Uploads

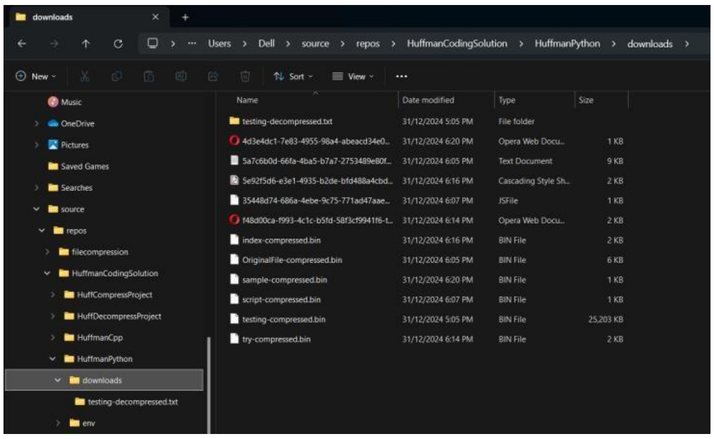
b) Downloads folder.

I. The downloads folder stores all output files generated by the system, including compressed .bin files and their corresponding decompressed text files.

II. For example:

The file testing-compressed.bin is the compressed output for the input file testing.txt. The file testing-decompressed.txt is generated after decompressing testing-compressed.bin.

III. Figure 23 provides an overview of the files in the downloads folder.



Name	Date modified	Type	Size
testing-decompressed.txt	31/12/2024 5:05 PM	File folder	
4d3e4dc1-7e83-4955-98a4-abeacd34e0...	31/12/2024 6:20 PM	Opera Web Docu...	1 KB
5a7c6b0d-66fa-4ba5-b7a7-2753489e80f...	31/12/2024 6:05 PM	Text Document	9 KB
5e92f5d6-e3e1-4935-b2de-bfd488a4cbd...	31/12/2024 6:16 PM	Cascading Style Sh...	2 KB
35448d74-686a-4ebe-9c75-771ad47aae...	31/12/2024 6:07 PM	JSFile	1 KB
f48d00ca-f993-4c1c-b5fd-58f3cf9941f6-t...	31/12/2024 6:14 PM	Opera Web Docu...	2 KB
index-compressed.bin	31/12/2024 6:16 PM	BIN File	2 KB
OriginalFile-compressed.bin	31/12/2024 6:05 PM	BIN File	6 KB
sample-compressed.bin	31/12/2024 6:20 PM	BIN File	1 KB
script-compressed.bin	31/12/2024 6:07 PM	BIN File	1 KB
testing-compressed.bin	31/12/2024 5:05 PM	BIN File	25,203 KB
try-compressed.bin	31/12/2024 6:14 PM	BIN File	2 KB

Figure 22: Decompress Uploads

3.3.6 Static folder

The static folder contains auxiliary files, such as stylesheets or images, that are essential for the visual presentation of the web-based user interface.

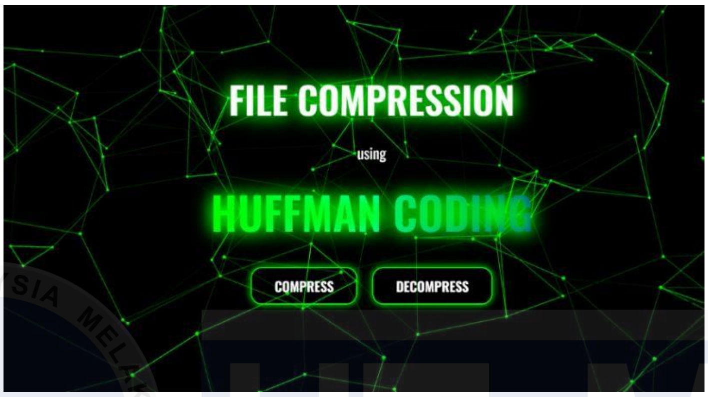


Figure 23: Homepage of The Project

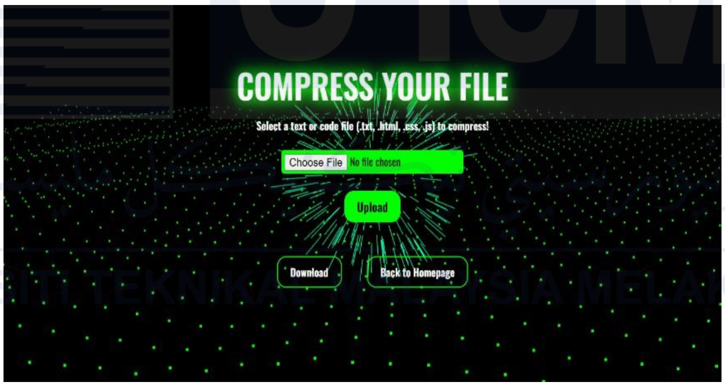
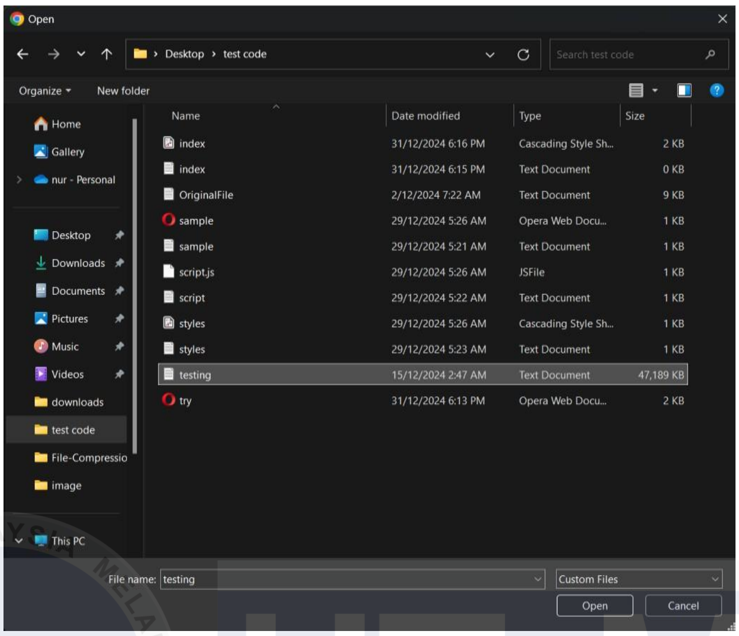


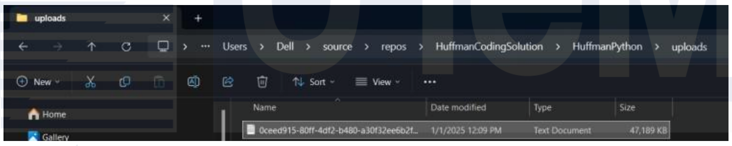
Figure 24: Compress Text File Page



Name	Date modified	Type	Size
index	31/12/2024 6:16 PM	Cascading Style Sh...	2 KB
index	31/12/2024 6:15 PM	Text Document	0 KB
OriginalFile	2/12/2024 7:22 AM	Text Document	9 KB
sample	29/12/2024 5:26 AM	Opera Web Docu...	1 KB
sample	29/12/2024 5:21 AM	Text Document	1 KB
script.js	29/12/2024 5:26 AM	JSFile	1 KB
script	29/12/2024 5:22 AM	Text Document	1 KB
styles	29/12/2024 5:26 AM	Cascading Style Sh...	1 KB
styles	29/12/2024 5:23 AM	Text Document	1 KB
testing	15/12/2024 2:47 AM	Text Document	47,189 KB
try	31/12/2024 6:13 PM	Opera Web Docu...	2 KB

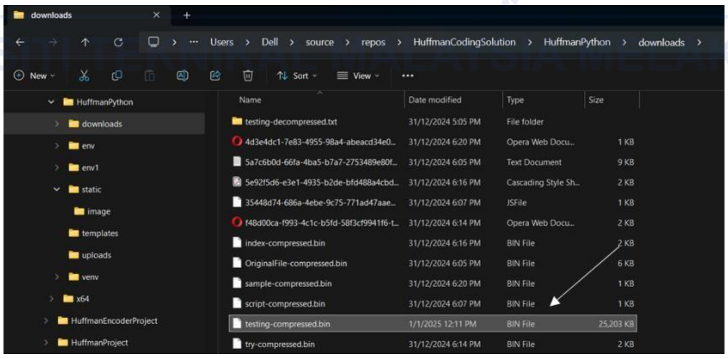
File name: testing

Figure 25: Upload testing.txt Into Uploads File With 47,189 KB


uploads
Users › Dell › source › repos › HuffmanCodingSolution › HuffmanPython › uploads
New
Sort
View
Home
Gallery

Name	Date modified	Type	Size
0ceed915-80ff-4df2-b480-a30f32ee6b2f...	1/1/2025 12:09 PM	Text Document	47,189 KB

Figure 26: Successfully Added testing.txt File Into Uploads File



Name	Date modified	Type	Size
testing-decompressed.txt	31/12/2024 5:05 PM	File folder	
4d3e4dc1-7e83-4955-98a4-abeacd34e0...	31/12/2024 6:20 PM	Opera Web Docu...	1 KB
5a7c6b0d-66fa-4ba5-b7a7-2753489e80f...	31/12/2024 6:05 PM	Text Document	9 KB
5e92f5d6-e3e1-4935-b2de-bfd488a4cbd...	31/12/2024 6:16 PM	Cascading Style Sh...	2 KB
35448d74-686a-4ebe-9c75-771ad47aae...	31/12/2024 6:07 PM	JSFile	1 KB
f48d00ca-f993-4c1c-b5fd-58f3cf9941f6-t...	31/12/2024 6:14 PM	Opera Web Docu...	2 KB
index-compressed.bin	31/12/2024 6:16 PM	BIN File	2 KB
OriginalFile-compressed.bin	31/12/2024 6:05 PM	BIN File	6 KB
sample-compressed.bin	31/12/2024 6:20 PM	BIN File	1 KB
script-compressed.bin	31/12/2024 6:07 PM	BIN File	1 KB
testing-compressed.bin	1/1/2025 12:11 PM	BIN File	25,203 KB
try-compressed.bin	31/12/2024 6:14 PM	BIN File	2 KB

Figure 27: bin File Was Successfully Created In Downloads File With 25,203 KB

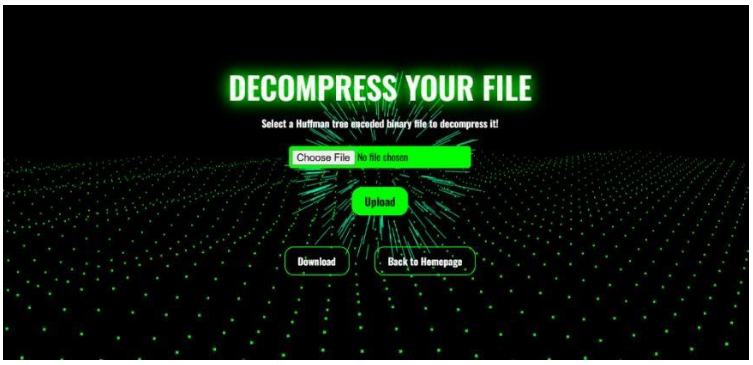
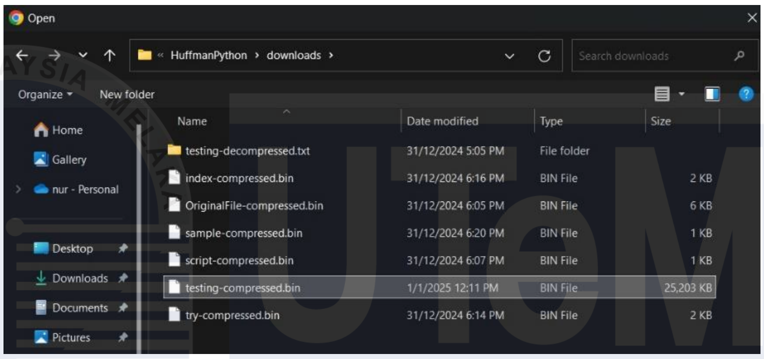
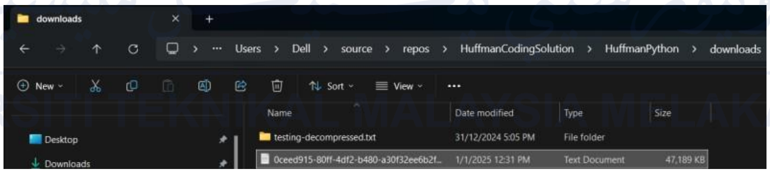


Figure 28: Decompress Text File Page



Name	Date modified	Type	Size
testing-decompressed.txt	31/12/2024 5:05 PM	File folder	
index-compressed.bin	31/12/2024 6:16 PM	BIN File	2 KB
OriginalFile-compressed.bin	31/12/2024 6:05 PM	BIN File	6 KB
sample-compressed.bin	31/12/2024 6:20 PM	BIN File	1 KB
script-compressed.bin	31/12/2024 6:07 PM	BIN File	1 KB
testing-compressed.bin	1/1/2025 12:11 PM	BIN File	25,203 KB
try-compressed.bin	31/12/2024 6:14 PM	BIN File	2 KB

Figure 29: Decompress-Upload testing-compressed.bin From Downloads File


downloads
Users › Dell › source › repos › HuffmanCodingSolution › HuffmanPython › downloads
New
Sort
View
Desktop
Downloads

Name	Date modified	Type	Size
testing-decompressed.txt	31/12/2024 5:05 PM	File folder	
0ceed915-80ff-4df2-b480-a30f32ee6b2f...	1/1/2025 12:31 PM	Text Document	47,189 KB

Figure 30: Successfully Decompress The File Back To Original Type File

3.4 Step for implementation

1. Step 1: Conduct frequency analysis.

The programme parses the input text file and computes the occurrence rate of each individual character.

The frequencies are recorded in a table or list, which acts as the foundation for constructing the Huffman Tree.

2. Step 2: Constructing the Huffman Tree

The characters and their frequency are added to a priority queue. The two characters with the lowest frequencies are iteratively removed and merged into a new node, which is subsequently inserted back into the priority queue. The procedure continues until only one node remains, which represents the root of the Huffman Tree.

3. Step 3: Creating Huffman Codes

The assignment of binary codes to each character is accomplished by traversing the HuffmanTree.

The left edges are assigned the label '0' while the right edges are assigned the label '1', guaranteeing that no code is a prefix of another.

4. Step 4: Text Encoding

The initial text is encoded utilizing the produced Huffman codes.

The encoded text and the Huffman Tree structure are stored in a compressed file.

3.5 Summary

This chapter delineated the systematic methodology employed to design, implement, and assess the Huffman Coding system for text file compression. The process commenced with a comprehensive overview of the project flowchart, outlining each phase in the system's development. Microsoft Visual Studio served as the principal integrated development environment, integrating C++ and Python capabilities to facilitate effective compression and decompression.

The methodology outlines the project's structure in Visual Studio, comprising three primary components:

a) HuffCompressProject: Implements a compression algorithm to encode text files into compact .bin files.
b) HuffDecompressProject: Reverts .bin files to their original text format.
c) HuffmanPython: Offers an intuitive web interface for file compression and decompression.

This chapter elucidated the project structure in Visual Studio, detailing folder organization and the functions of uploads and downloads directories in managing input and output files. The methodology for constructing Huffman Trees, allocating binary codes to characters, and producing compressed binary files was elucidated through sequential instructions and accompanying illustrations.

By adhering to this methodology, the Huffman Coding system guarantees substantial file size reduction while preserving data integrity. This project's methodology illustrates scalability for diverse file types and establishes a basis for effective data management.

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Introduction

This chapter delineates the results of the Huffman Coding system, illustrating its efficacy in compressing text files. The results demonstrate the diminishment of file sizes, the intricate formation of Huffman Trees, and the most efficient encoding of textual data. These results confirm the efficacy of Huffman Coding in attaining lossless compression, with possible applications in data transmission, storage optimization, and real-time processing.

4.2 Huffman tree result

The Huffman Tree is an essential element of the Huffman Coding algorithm, facilitating the creation of variable-length binary codes derived from character frequencies. This section illustrates the construction of the Huffman Tree utilizing the input string 'AEE ISSIG MSISS HTH' and elucidates each step with accompanying figures.

4.2.1 Explanation by figure

Suppose we had the following string of characters

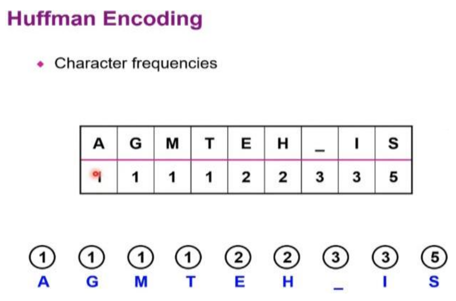
AEE ISSIG MSISS HTH

Figure 31: String of Characters

a) Step 1:

   I. Insert each character as separate nodes into a priority queue, arranged by frequency.

II. A priority queue is utilised to guarantee that the two nodes with the lowest frequencies are consistently merged first. This reduces the total length of the code.



A	G	M	T	E	H	_	I	S
1	1	1	1	2	2	3	3	5

Figure 32: Character Frequencies

b) Step 2:

I. Combine A and G (frequencies: 1 + 1 = 2).

II. Generate a new node with a frequency of 2 and reintegrate it into the queue.

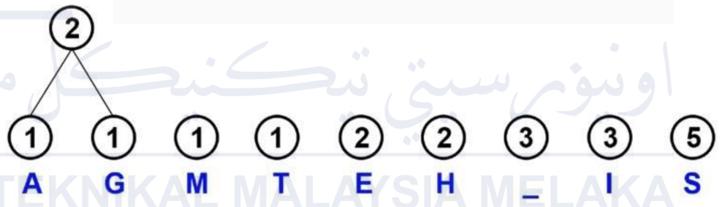


Figure 33: Combine A&G

c) Step 3:

I. Combine M and T (frequencies: 1 + 1 = 2).

II. Generate a new node with a frequency of 2 and reintegrate it into the queue.

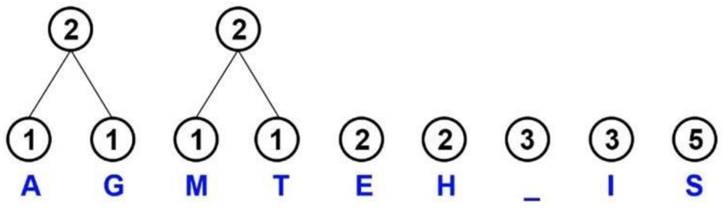


Figure 34: Combine M&T

d) Step 4:

I. Combine E and H (frequencies: 2 + 2 = 4).

II. Generate a new node with a frequency of 4 and reintegrate it into the queue.

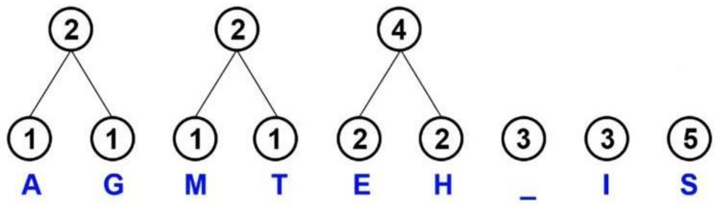


Figure 35: Combine E&H

e) Step 5:

I. Combine (A-G) and (M-T) (frequencies: 2 + 2 = 4).

II. Generate a new node with a frequency of 4 and reintegrate it into the queue.

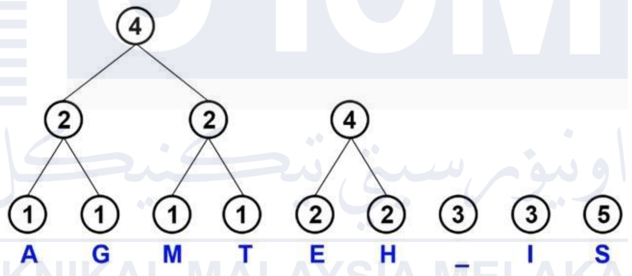


Figure 36: Combine 2&2

f) Step 6:

I. Combine (A-G) (M-T) and (E-H) (frequencies: 4 + 4 = 8).

II. Generate a new node with a frequency of 8 and reintegrate it into the queue.

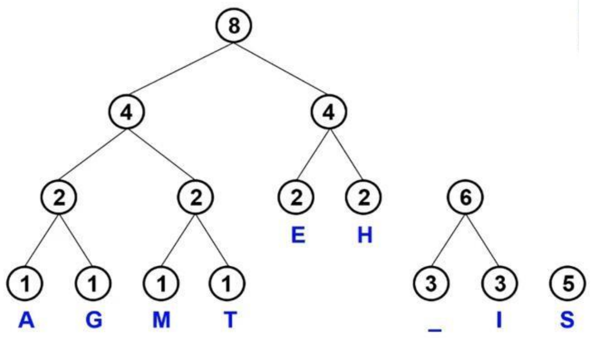


Figure 37: Combine 4&4

g) Step 7:

I. Combine (- and I) and (- -I) and (S) (frequencies: 6 + 5 = 11).

II. Generate a new node with a frequency of 11 and reintegrate it into the queue.

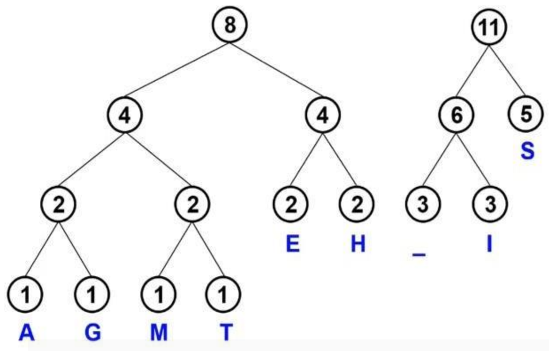


Figure 38: Combine 6&5

g) Step 8:

I. Combine (frequencies: 8 + 11 = 19).

II. Generate a new node with a frequency of 19 and reintegrate it into the queue.

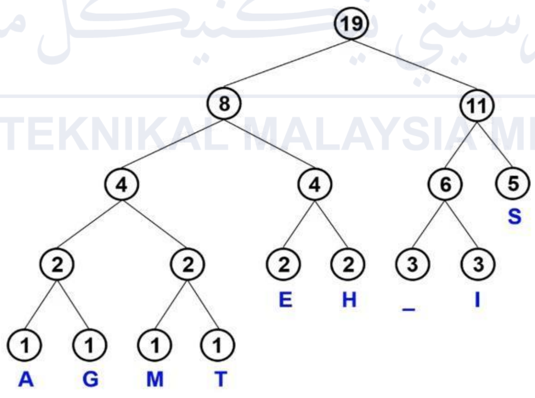


Figure 39: Combine 8&11

III. Note that the “19” was the root for this tree.

h) Step 9:

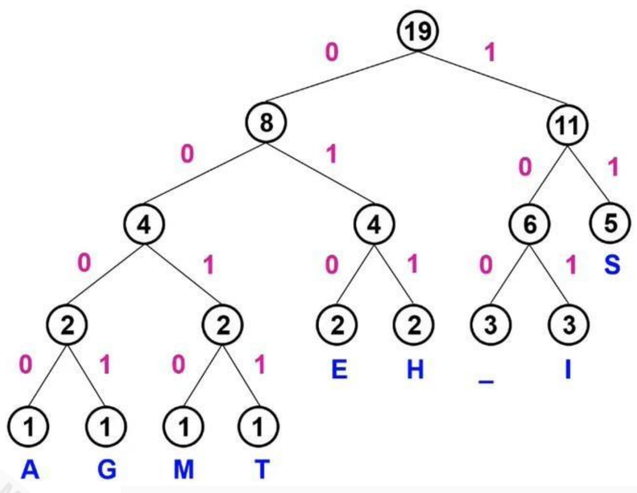
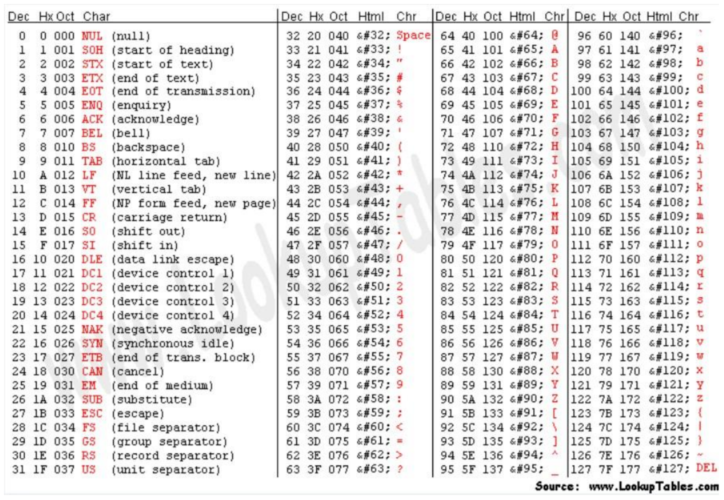


Figure 40: Assigned the left edges as ‘0’ & Assigned the right edges as '1’

4.2.2 Detail explanation for Huffman tree

1. Given the string was ‘go go gophers’.
2. Then convert the given characters into ASCII code.

Table 4: ASCII Code



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

3. Convert ASCII to binary.

a) Each ASCII code is then converted into an 8-bit binary representation:

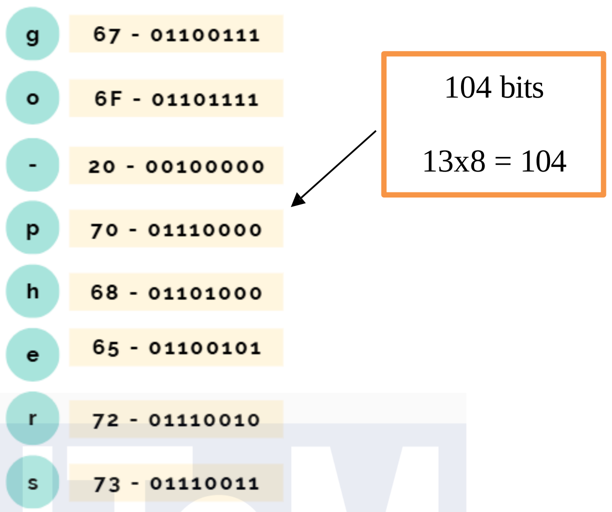


Figure 41: Hex to Binary

b) Count the frequency of each characters.

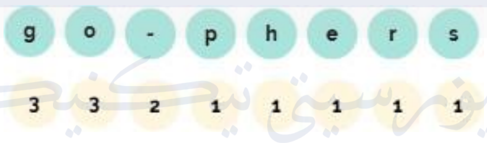


Figure 42: Frequency of Characters

c) Constructing the Huffman tree

I. Combined r & s

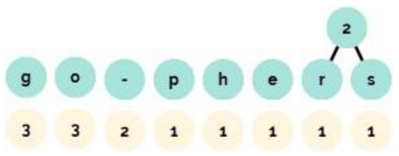


Figure 43: Step 1

II. Combined h & e

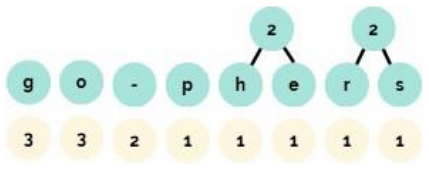


Figure 44: Step 2

III. Combined - & p

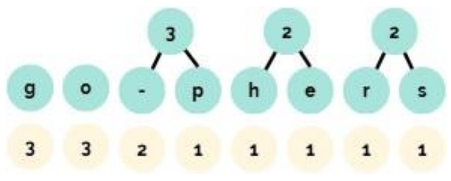


Figure 45: Step 3

IV. Combined 2 & 2

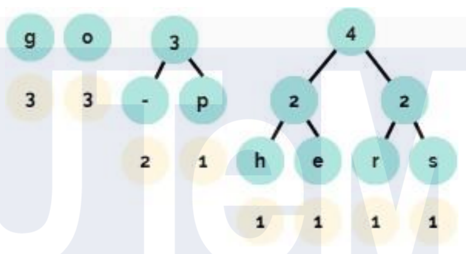


Figure 46: Step 4

V. Combined g & o

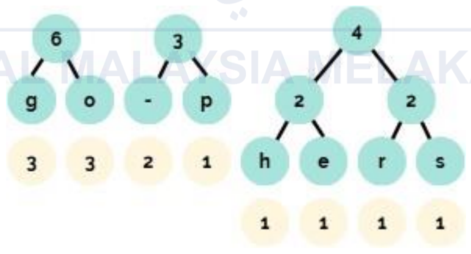


Figure 47: Step 5

VI. Combined 3 & 4

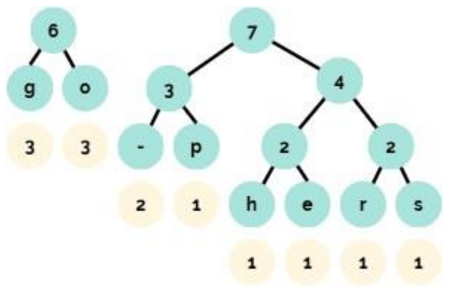


Figure 48: Step 6

VII. Combined 6 & 7

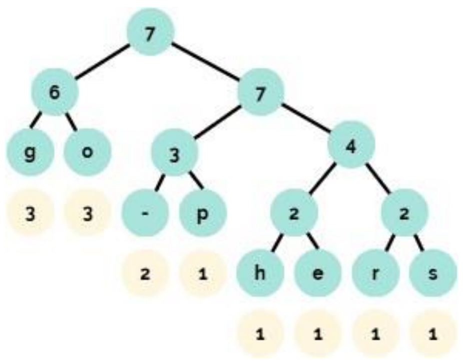


Figure 49: Step 7

d) Assign Binary Codes

I. Once the Huffman Tree is constructed:

Left branches are assigned 0.

Right branches are assigned 1.

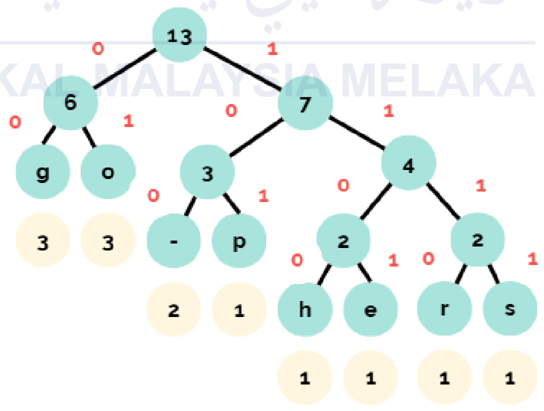
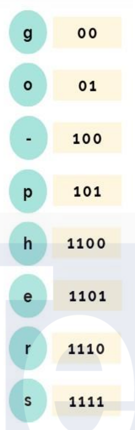


Figure 50: Step 8

e) Generate Binary Codes for Characters

I. The binary codes for the characters in the string "go go gophers" are as follows:



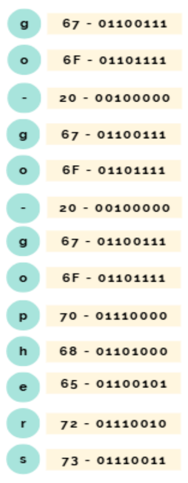
g	00
o	01
-	100
p	101
h	1100
e	1101
r	1110
s	1111

Figure 51: Step 9

II. The codes are generated from the Huffman Tree by tracing the path from the root to each character's leaf node.

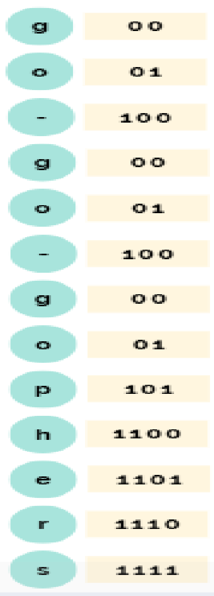
f) Compress the string

I. Original ASCII binary representation (13 characters × 8 bits each): 104 bits.



g	67 - 01100111
o	6F - 01101111
-	20 - 00100000
g	67 - 01100111
o	6F - 01101111
-	20 - 00100000
g	67 - 01100111
o	6F - 01101111
p	70 - 01110000
h	68 - 01101000
e	65 - 01100101
r	72 - 01110010
s	73 - 01110011

Figure 52: Original: 104 bits



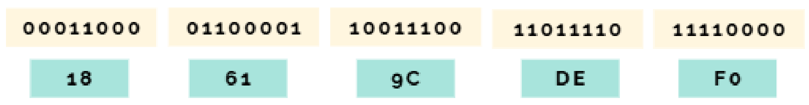
g	00
o	01
-	100
g	00
o	01
-	100
g	00
o	01
p	101
h	1100
e	1101
r	1110
s	1111

Figure 53: New: 37 bits

II. The encoded binary string after Huffman coding is significantly shorter than the original, demonstrating the efficiency of Huffman coding. With the binary bits: 37 bits.

g) Convert Binary to Hex (Preparation for Decompression code)

I. After encoding, the compressed binary string is divided into 8-bit segments (1 byte each).

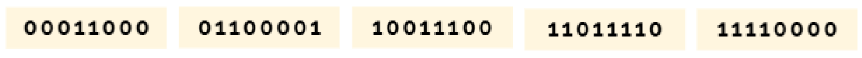


00011000	01100001	10011100	11011110	11110000
18	61	9C	DE	F0

Figure 54: From Binary to Hex

h) This code binary will be saved into .bin file.

I. The binary representation is stored in a compressed .bin file for storage or transmission.

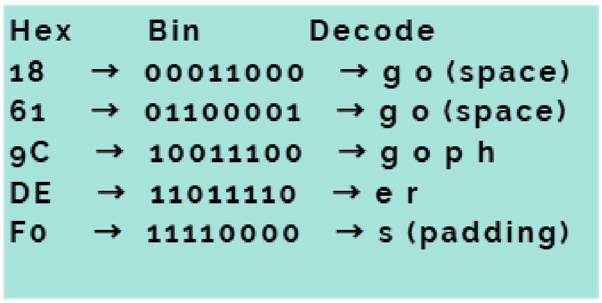


00011000	01100001	10011100	11011110	11110000

Figure 55: Binary In .bin File

i) Decompression (Reading the .bin File)

I. The .bin file is analyzed and decoded to recreate the original text.



```
Hex     Bin          Decode
18   →  00011000   → g o (space)
61   →  01100001   → g o (space)
9C   →  10011100   → g o p h
DE   →  11011110   → e r
F0   →  11110000   → s (padding)
```

Figure 56: Step For Decoding

4.3 Binary Encoding and File Size Reduction

This section assesses the efficacy of the Huffman Coding system by contrasting file sizes prior to and subsequent to compression. The table below encapsulates the results, emphasizing the dimensions of input files, their associated compressed .bin files, and the decompressed outputs.

Table 5: File Size Comparison Before and After

INPUT FILES	SIZE	COMPRESS	BIN SIZES	DECOMPRESS	SIZE	COMPRESSION RATIO	COMPRESSION PERCENTAGE (REDUCTION)
main.css	5 KB	main-compressed.bin	3 KB	b2d5af35-1619-482f-885b-788f36288e5c-main.css	5 KB	$\frac{3}{5}X100 = 60\%$	100−60=40%
styles.css	4 KB	styles-compressed.bin	3 KB	c57e8738-c2cf-4909-9961-92bb3d8a509d-styles.css	4 KB	$\frac{3}{4}X100 = 75\%$	100−75=25%
app.js	6 KB	app-compressed.bin	4 KB	6e3b2d00-d9b3-415f-aa0f-82f5f0be45e4-app.js	6 KB	$\frac{4}{6}X100 = 66.67\%$	100−66.67=33.33%
script.js	5 KB	script-compressed.bin	4 KB	31fc7cd2-ff66-4147-9a43-d84f8e8f9515-script.js	5 KB	$\frac{4}{5}X100 = 80\%$	100−80=20%
ecommerce .html	6 KB	ecommerce-compressed.bin	4 KB	e7510b0b-4840-4ca6-a182-6c2a96b7d002-ecommerce.html	6 KB	$\frac{4}{6}X100 = 66.67\%$	100−66.67=33.33%
homepage .html	8 KB	homepage-compressed.bin	5 KB	695f2ea2-bbc1-4d4c-ba8f-0053315a2204-homepage.html	8 KB	$\frac{5}{8}X100 = 62.5\%$	100−62.5=37.5%
originalfile .txt	9 KB	originalfile-compressed.bin	6 KB	71b65343-e565-4d18-845d-5fc3cee952c5-originalfile.txt	9 KB	$\frac{6}{9}X100 = 66.67\%$	100−66.67=33.33%
testing.txt	47,189 KB	testing-compressed.bin	25,203 KB	f1fa1dc4-4deb-4eef-afec-4f2b4fc7b87a-testing.txt	47,189 KB	$\frac{25,203}{47,189}X100 = 53.4\%$	100−53.4=46.6%

The table demonstrates the efficacy of the Huffman Coding system in compressing files of diverse sizes and types.

a) The file testing.txt, originally 47,189 KB, was compressed to 25,203 KB. This indicates a compression ratio of roughly 46.6%.

b) Files of lesser size, such as styles.css (4 KB) and main.css (5 KB), exhibited minimal decreases in size (from 5 KB to 3 KB for main.css and from 4 KB to 3 KB for styles.css).

c) The decompressed sizes of all files correspond to the original sizes, thereby validating the lossless characteristics of the Huffman Coding system. This illustrates that the algorithm effectively preserves all original information throughout the compression and decompression processes.

d) The .bin files produced during compression are concise and effectively represent the input data through the binary encoding derived from the Huffman Tree. The file originalfile.txt was compressed from 9 KB to 6 KB while preserving its content integrity.

This is how to calculate compression ratio and compression percentage (Reduction):

$$\text{Compression Ratio} = \frac{\text{Compressed Size}}{\text{Original Size}}\ x\ 100$$

Compression Percentage (Reduction) = 100−Compression Ratio (%)

4.4 Summary

This chapter outlined the findings and assessment of the Huffman Coding system for compressing text files. The chapter commenced by demonstrating the sequential construction of the Huffman Tree, an essential element of the compression process.

Characters from the input string were systematically amalgamated using a priority queue according to their frequencies to construct a complete binary tree. The tree structure enabled the creation of variable-length binary codes, assigning shorter codes to more frequent characters to ensure optimal compression efficiency.

The binary encoding process was examined, emphasizing the decrease in file size relative to the original ASCII format. The results indicated the system's proficiency in compressing files, attaining a compression ratio of roughly 46.6% for larger text files. The decompression process was validated, confirming that the system precisely reconstructed the original text from the compressed .bin files. The results highlighted the lossless characteristics of Huffman Coding, establishing it as a dependable and efficient method for text file compression.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

5.1 Conclusion

The project effectively created a Huffman Coding-based system for compressing text files, resulting in substantial reductions in file size while maintaining data integrity. The system proved to be efficient, scalable, and reliable, rendering it appropriate for data storage and transmission applications. Huffman Coding optimizes storage and transmission costs by assigning shorter codes to frequently occurring characters, preserving the integrity of the original data. The results confirm the system's ability to manage diverse text file formats and highlight its potential for wider application.

5.2 Potential for Commercialization

The Huffman Coding-based text compression system provides an efficient and lossless method for handling text files, rendering it suitable for commercial applications across various industries.

Principal benefits comprise:

I. Huffman coding significantly reduces text file sizes, lowering storage costs and improving efficiency.

II. Compressed files require less bandwidth, making data transmission faster and cost- effective.

III. Ideal for embedded systems and IoT devices with limited memory, enabling efficient data management

5.3 Future Works

To enhance the system's functionalities, multiple areas for enhancement have been recognized:

I. Design adaptive Huffman algorithms to process real-time data streams for applications including live video feeds or sensor data.

II. Enhance the compression algorithm to reduce energy consumption, especially in large-scale data centres.

III. Optimize the system to effectively handle exceptionally large files or datasets, potentially employing parallel computing or distributed systems.

IV. Incorporate encryption and error-correction methods to guarantee secure and dependable data management during compression and decompression.

REFERENCES

[1] "Text File Compression And Decompression Using Huffman Coding," 10 May 2024. [Online]. Available: https://www.geeksforgeeks.org/text-file-compression-and-decompression-using-huffman-coding/.

[2] Andysah Putera Utama Siahaan, "Huffman Text Compression Technique," *SSRG International Journal of Computer Science and Engineering,* vol. 3, no. 8, 2016.

[3] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," 9 September 1952. [Online]. Available: https://ieeexplore.ieee.org/document/4051119.

[4] D.Jasmine Shoba, Dr.S.Sivakumar, "A Study on Data Compression Using Huffman Coding," *International Journal of Computer Science Trends and Technology,* vol. 5, no. 1, 2017.

[5] Ian H. Witten, Alistair Moffat, and Timothy C. Bell , "Managing Gigabytes:Compressing and Indexing documents and images," 1999.

[6] David Salomon, Data Compression: The Complete Reference, Springer Science & Business Media, 2007.

[7] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal,* vol. 27, no. 3, pp. 379-423, July 1948.

[8] Virendra Nikam, Sheetal Dhande, "A Historical Perspective on Approaches to Data Compression," *Mathematics and Computer Science,* vol. 8, no. 3, pp. 68-72, 2023.

[9] Timothy C. Bell, John G. Cleary, Ian H. Witten, Text Compression, the University of Michigan: Prentice Hall, 1990, 20 Nov 2007.

[10] JACOB ZIV, ABRAHAM LEMPEL, "Universal Algorithm for Sequential Data Compress," *IEEE TRANSACTIONS ON INFORMATION THEORY,* p. NO. 3, 1977.

[11] J. Ziv; A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory* , pp. 530 - 536, 1978.

[12] Thomas M. Cover, Joy A. Thomas, Elements of Information Theory, Wiley-Interscience, 7 April 2005.

[13] Andrew J Elliot 1, Markus A Maier, "Color psychology: effects of perceiving color on psychological functioning in humans," *Annual Review of Psychology,* vol. 65, pp. 95-120, 2014.

[14] Witzel, C.; Gegenfurtner, K. R, "Color Perception: Objects, Constancy, and Categories," *Annual Review of Vision Science,* vol. 4, no. 1, pp. 475-499, 2018.

[15] World Wide Web Consortium (W3C), "HTML Standard," World Wide Web Consortium (W3C), 16 January 2023. [Online]. Available: https://html.spec.whatwg.org/. [Accessed 16 January 2025].

[16] M. Burrows and D.J. Wheeler, "A Block-sorting Lossless," Systems Research Center, May 10, 1994.

[17] M. Nelson, and J. Gailly., The Data Compression Book, 2nd Edition, M&T Books, 1996.

[18] K Sayood , Introduction to data compression, Morgan Kaufmann, 2017.

[19] Welch, "A Technique for High-Performance Data Compression," *Computer,* pp. 8 - 19, 1984 .

[20] J. J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," *IBM Journal of Research and Development,* vol. 20, no. 3, pp. 198 - 203, May 1976.

[21] G. G. Langdon, "An Introduction to Arithmetic Coding," *IBM Journal of Research and Development,* vol. 28, no. 2, pp. 135 - 149, March 1984.

[22] Ian Witten.,Ian H., Neal, J. G., "Arithmetic Coding for Data Compression," January 1987.

[23] Donald Knuth , Art of Computer Programming, The: Sorting and Searching, Volume 3, Addison-Wesley Professional; 2nd edition, (April 24, 1998).

[24] Alistair Moffat , Andrew Turpin, Compression and Coding Algorithms, Springer New York, NY, 2002.

[25] "GeeksforGeeks," 10 may 2024. [Online]. Available: https://www.geeksforgeeks.org/text-file-compression-and-decompression-using-huffman-coding/.

[26] "Text File Compression And Decompression Using Huffman ..," 10 May 2024. [Online]. Available: https://www.geeksforgeeks.org/text-file-compression-and-decompression-using-huffman-coding/.

[27] GeeksforGeeks, "Huffman Coding in C," GeeksforGeeks, 19 July 2024. [Online]. Available: https://www.geeksforgeeks.org/huffman-coding-in-c/. [Accessed 16 January 2025].

[28] GeeksforGeeks, "Huffman Coding in C++," GeeksforGeeks, 14 July 2024. [Online]. Available: https://www.geeksforgeeks.org/huffman-coding-in-cpp/. [Accessed 16 January 2025].

[29] David A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE,* vol. 40, no. 9, pp. 1098 - 1101, 1952.

[30] Ian H. Witten, Radford M. Neal, John G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520-540, 1987.

[31] Computer Hope, "David Huffman," Computer Hope, 30 Desember 2019. [Online]. Available: https://www.computerhope.com/people/david_huffman.htm. [Accessed 16 January 2025].

[32] Wikipedia, "Huffman Codding," Wikipedia, 15 January 2025. [Online]. Available: https://en.wikipedia.org/wiki/Huffman_coding. [Accessed 16 January 2025].

[33] wscubetech, "Huffman Code: Example, Algorithm, Time Complexity," wscubetech, 16 January 2025. [Online]. Available: https://www.wscubetech.com/resources/dsa/huffman-code. [Accessed 16 January 2025].

[34] Trung Tran, "Huffman coding," UC Santa Cruz, 31 July 2024. [Online]. Available: https://www.ucsc.edu/timeline/huffman-coding/. [Accessed 16 January 2025].

[35] AmazingAlgorithms., "Huffman Coding," AmazingAlgorithms., 2025. [Online]. Available: https://amazingalgorithms.com/definitions/huffman-coding/. [Accessed 16 Jun 2025].

[36] GeeksforGeeks, "Huffman Coding | Greedy Algo-3," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/. [Accessed 16 January 2025].

[37] Neeraj Mishra, "Huffman Coding Algorithm With Example," thecrazyprogrammer, 2025. [Online]. Available: https://www.thecrazyprogrammer.com/2014/09/huffman-coding-algorithm-with-example.html. [Accessed 16 Jun 2025].

[38] Yuriy Georgiev, "Lossless compression - Huffman Coding & RLE," Yuriy Georgiev, 3 February 2024. [Online]. Available: https://yuriygeorgiev.com/2024/02/03/lossless-compression-huffman-coding-rle/. [Accessed 16 January 2025].

[39] Eva SongPaul CuffH. Vincent PoorH. Vincent Poor, "The Likelihood Encoder for Lossy Source Compression," *IEEE Transactions on Information Theory*, 2014.

[40] K. W. ROBERT SEDGEWICK, "DATA COMPRESSION," Princeton University, 2021. [Online]. Available: https://www.cs.princeton.edu/courses/archive/spring21/cos226/lectures/55DataCompression.pdf. [Accessed 16 January 2025].

APPENDICES

Appendix A Project Planning(Gantt Chart)

	J. **PERANCANGAN PROJEK** ***PROJECT PLANNING (GANTT CHART)***
	Senaraikan aktiviti-aktiviti yang berkaitan bagi projek yang dicadangkan dan nyatakan jangkamasa yang diperlukan bagi setiap aktiviti. *List all the relevant activities of the proposed project and mark the period of time that isneeded for each of the activities.*

		PSM II												
Aktiviti Projek *Project Activities*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Design initial UI, install necessary tools (e.g., Visual Studio), and implement the basic Huffman coding algorithm.	🟦													
Debug the Huffman coding algorithm and conduct edge-case testing.		🟦												
Research improvements for the Huffman algorithm and its application.			🟦											
Design and develop the website interface, including interactive functionalities.				🟦										
Optimize the Huffman coding algorithm for better compression results.					🟦									
Implement and test the decompression						🟦								

functionality.						🟦								
Troubleshoot file handling issues and refine error handling in compression and decompression.							🟦							
Validate the full workflow, including file compression and decompression.								🟦						
Set up Flask web application and integrate compression functionalities.									🟦					
Perform comprehensive testing and finalize the project.										🟦				
Perform comprehensive testing and finalize the project.											🟦			
Make preparation for presentation, do poster												🟦		
Make preparation for presentation, do poster													🟦	
Make any correction on thesis, based on discussion with supervisor														🟦

Appendix B Contents Language

HuffCompressProject.cpp

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <vector>
#include <algorithm>
#include <cstdint>
#include <cstdio>
#include <filesystem>

using namespace std;
namespace fs = std::filesystem;

unordered_map<char, string> hashmap;

struct huffnode {
    char character;
    unsigned int freq;
    huffnode* left;
    huffnode* right;
    ~huffnode() { delete left; delete right; } // Destructor for cleanup
};

huffnode* new_node(char c, unsigned int freq, huffnode* left = nullptr,
huffnode* right = nullptr) {
    return new huffnode{ c, freq, left, right };
}

void assign_codes(huffnode* root, string code) {
    if (!root->left && !root->right) {
        hashmap[root->character] = code;
        return;
    }
    if (root->left) assign_codes(root->left, code + "0");
    if (root->right) assign_codes(root->right, code + "1");
}

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cerr << "Usage: HuffCompressProject.exe <input file> <output
file>\n";
        return 1;
    }

    FILE* input;
    if (fopen_s(&input, argv[1], "rb") != 0) {
        cerr << "Error: Failed to open input file: " << argv[1] << "\n";
        return 2;
    }

    // Ensure the output directory exists
    fs::path output_path = argv[2];
    fs::create_directories(output_path.parent_path());
    FILE* output;
    if (fopen_s(&output, argv[2], "wb") != 0) {
        cerr << "Error: Failed to create output file at: " << argv[2] <<
"\n";
        fclose(input);
        return 3;
    }

    cout << "Processing file: " << argv[1] << "\n";
    cout << "Output file will be saved to: " << argv[2] << "\n";

    // Step 1: Frequency analysis
    unsigned int freq[256] = { 0 };
    char c;
    while (fread(&c, 1, 1, input)) {
        freq[(unsigned char)c]++;
    }
    rewind(input);

    // Step 2: Build Huffman Tree
    vector<huffnode*> nodes;
    for (int i = 0; i < 256; i++) {
        if (freq[i] > 0) {
            nodes.push_back(new_node((char)i, freq[i]));
        }
    }

    while (nodes.size() > 1) {
        sort(nodes.begin(), nodes.end(), [](huffnode* a, huffnode* b) {
return a->freq > b->freq; });
        huffnode* left = nodes.back(); nodes.pop_back();
        huffnode* right = nodes.back(); nodes.pop_back();
        nodes.push_back(new_node('\0', left->freq + right->freq, left,
right));
    }
    assign_codes(nodes[0], "");

    // Save the original filename and extension as metadata
    string original_filename = fs::path(argv[1]).filename().string();
    uint32_t filename_length = original_filename.size();
    fwrite(&filename_length, sizeof(uint32_t), 1, output); // Write filename
length
    fwrite(original_filename.c_str(), 1, filename_length, output); // Write
filename

    // Step 3: Write Huffman map
    uint32_t num_entries = hashmap.size();
    fwrite(&num_entries, sizeof(uint32_t), 1, output);
    for (const auto& pair : hashmap) {
        fwrite(&pair.first, 1, 1, output);
        uint32_t code_length = pair.second.size();
        fwrite(&code_length, sizeof(uint32_t), 1, output);

        unsigned char byte = 0;
        int bit_count = 0;
        for (char bit : pair.second) {
            byte = (byte << 1) | (bit - '0');
            bit_count++;
            if (bit_count == 8) {
                fwrite(&byte, 1, 1, output);
                byte = 0;
                bit_count = 0;
            }
        }
        if (bit_count > 0) {
            byte <<= (8 - bit_count);
            fwrite(&byte, 1, 1, output);
        }
    }

    // Step 4: Write encoded data
    unsigned char byte = 0;
    int bit_count = 0;
    while (fread(&c, 1, 1, input)) {
        string code = hashmap[c];
        for (char bit : code) {
            byte = (byte << 1) | (bit - '0');
            bit_count++;
            if (bit_count == 8) {
                fwrite(&byte, 1, 1, output);
                byte = 0;
                bit_count = 0;
            }
        }
    }
    if (bit_count > 0) {
        byte <<= (8 - bit_count);
        fwrite(&byte, 1, 1, output);
    }

    fclose(input);
    fclose(output);
    delete nodes[0]; // Clean up dynamically allocated memory

    cout << "Compression completed successfully.\n";
    return 0;
}
```

HuffDecompressProject.cpp

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <cstdint>
#include <cstdio>
#include <filesystem>
#include <vector>
#include <cassert>

using namespace std;
namespace fs = std::filesystem;

unordered_map<string, char> reverse_hashmap;

struct BitReader {
    FILE* file;
    uint8_t buffer;
    int bits_left;

    explicit BitReader(FILE* file) : file(file), buffer(0), bits_left(0) {}

    int read_bit() {
        if (bits_left == 0) {
            if (fread(&buffer, 1, 1, file) != 1) return -1; // End of file 
or error
            bits_left = 8;
        }
        return (buffer >> --bits_left) & 1;
    }
};

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cerr << "Usage: HuffDecompressProject.exe <input file> <output 
folder>\n";
        return 1;
    }

    // Debug: Print input and output paths
    cout << "Input file: " << argv[1] << "\n";
    cout << "Output folder: " << argv[2] << "\n";

    // Open the input file
    FILE* input;
    if (fopen_s(&input, argv[1], "rb") != 0) {
        cerr << "Error: Failed to open input file: " << argv[1] << "\n";
        return 2;
    }
    assert(input != nullptr);

    // Read the original filename length
    uint32_t filename_length;
    if (fread(&filename_length, sizeof(uint32_t), 1, input) != 1) {
        cerr << "Error: Failed to read filename length.\n";
        fclose(input);
        return 3;
    }

    // Read the original filename
    vector<char> original_filename(filename_length + 1);
    if (fread(original_filename.data(), 1, filename_length, input) != 
filename_length) {
        cerr << "Error: Failed to read original filename.\n";
        fclose(input);
        return 4;
    }
    original_filename[filename_length] = '\0';

    // Debug: Print original filename
    cout << "Original filename: " << original_filename.data() << "\n";

    // Create the decompressed file path
    fs::path decompressed_filepath = fs::path(argv[2]) / 
original_filename.data();
    fs::create_directories(decompressed_filepath.parent_path());

    // Debug: Print decompressed file path
    cout << "Decompressed file path: " << decompressed_filepath << "\n";

    // Open the decompressed output file
    FILE* decompressed_output;
    if (fopen_s(&decompressed_output, 
decompressed_filepath.string().c_str(), "wb") != 0) {
        cerr << "Error: Failed to create decompressed output file: " << 
decompressed_filepath << "\n";
        fclose(input);
        return 5;
    }
    assert(decompressed_output != nullptr);

    // Read the number of Huffman map entries
    uint32_t num_entries;
    if (fread(&num_entries, sizeof(uint32_t), 1, input) != 1) {
        cerr << "Error: Failed to read Huffman map entries.\n";
        fclose(input);
        fclose(decompressed_output);
        return 6;
    }

    // Read the Huffman map
    for (uint32_t i = 0; i < num_entries; i++) {
        char character;
        uint32_t code_length;
        if (fread(&character, 1, 1, input) != 1 || fread(&code_length, 
sizeof(uint32_t), 1, input) != 1) {
            cerr << "Error: Failed to read Huffman map entry.\n";
            fclose(input);
            fclose(decompressed_output);
            return 7;
        }

        string code;
        unsigned char byte = 0;
        int bits_read = 0;
        for (uint32_t j = 0; j < code_length; j++) {
            if (bits_read == 0) {
                if (fread(&byte, 1, 1, input) != 1) {
                    cerr << "Error: Failed to read Huffman code bits.\n";
                    fclose(input);
                    fclose(decompressed_output);
                    return 8;
                }
                bits_read = 8;
            }
            code.push_back(((byte >> --bits_read) & 1) ? '1' : '0');
        }
        reverse_hashmap[code] = character;
    }

    // Debug: Print Huffman map size
    cout << "Huffman map size: " << reverse_hashmap.size() << "\n";

    // Decode the compressed content
    BitReader reader(input);
    string current_code;
    int bit;
    while ((bit = reader.read_bit()) != -1) {
        current_code.push_back(bit ? '1' : '0');
        if (reverse_hashmap.find(current_code) != reverse_hashmap.end()) {
            char decoded_char = reverse_hashmap[current_code];
            fwrite(&decoded_char, 1, 1, decompressed_output);
            current_code.clear();
        }
    }

    // Check for leftover bits
    if (!current_code.empty()) {
        cerr << "Warning: Extra bits in the encoded file were ignored.\n";
    }

    // Close the files
    fclose(input);
    fclose(decompressed_output);

    // Reset permissions for the decompressed file
    fs::permissions(decompressed_filepath, fs::perms::owner_all |
fs::perms::group_all | fs::perms::others_all, fs::perm_options::add);

    cout << "Decompression completed successfully. File saved at: " <<
decompressed_filepath << "\n";
    return 0;
}
```

HuffmanPython.py

```
import os
import uuid
from flask import Flask, render_template, request, send_file
import subprocess

# Configure Application
app = Flask(__name__)

# Base directory of the project
BASE_DIR = os.path.abspath(os.path.dirname(__file__))

# Upload and Download folder paths
UPLOAD_FOLDER = os.path.join(BASE_DIR, "uploads")
DOWNLOAD_FOLDER = os.path.join(BASE_DIR, "downloads")

# Executable paths
COMPRESSION_EXE =
r"C:\Users\Dell\source\repos\HuffmanCodingSolution\x64\Debug\HuffCompressPro
ject.exe"
DECOMPRESSION_EXE =
r"C:\Users\Dell\source\repos\HuffmanCodingSolution\x64\Debug\HuffDecompressP
roject.exe"

# Create folders if they do not exist
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
os.makedirs(DOWNLOAD_FOLDER, exist_ok=True)

app.config["UPLOAD_FOLDER"] = UPLOAD_FOLDER
app.config["DOWNLOAD_FOLDER"] = DOWNLOAD_FOLDER

# Allowed file extensions
ALLOWED_EXTENSIONS = {'txt', 'bin', 'html', 'css', 'js'}

def allowed_file(filename):
    """Check if the file extension is allowed."""
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in
ALLOWED_EXTENSIONS


@app.route("/")
def home():
    """Render the home page."""
    return render_template("home.html")


@app.route("/compress", methods=["GET", "POST"])
def compress():
    """Handle file compression."""
    if request.method == "POST":
        uploaded_file = request.files.get("file")

        if uploaded_file and uploaded_file.filename and
allowed_file(uploaded_file.filename):
            original_filename = uploaded_file.filename
            filename = f"{uuid.uuid4()}-{original_filename}"
            save_path = os.path.join(app.config["UPLOAD_FOLDER"], filename)
            # Save uploaded file to uploads folder
            try:
                uploaded_file.save(save_path)
                print(f"File saved to: {save_path}")

                # Verify the file exists
                if os.path.exists(save_path):
                    print(f"Uploaded file exists at: {save_path}")
                else:
                    print(f"Uploaded file does not exist at: {save_path}")
                    return render_template("compress.html", check=-1,
message="File upload failed.")
            except Exception as e:
                print(f"Error saving file: {e}")
                return render_template("compress.html", check=-1,
message="File upload failed.")

            # Path for compressed file in downloads folder
            compressed_file = f"{os.path.splitext(original_filename)[0]}-
compressed.bin"
            compressed_path = os.path.join(app.config["DOWNLOAD_FOLDER"],
compressed_file)

            # Check if compression executable exists
            if not os.path.exists(COMPRESSION_EXE):
                print("Compression executable not found.")
                return render_template("compress.html", check=-1,
message="Compression executable not found.")

            # Run compression executable
            command = [COMPRESSION_EXE, save_path, compressed_path]
            print(f"Running command: {command}")
            try:
                result = subprocess.run(command, check=True,
capture_output=True, text=True)
                print(f"Subprocess output: {result.stdout}")
                print(f"Subprocess error: {result.stderr}")

                # Check if compressed file was successfully created
                if os.path.exists(compressed_path):
                    print("Compression successful.")
                    return render_template("compress.html", check=1,
filename=os.path.basename(compressed_path))
                else:
                    print("Compressed file not found.")
                    return render_template("compress.html", check=-1,
message="Compressed file not created.")
            except subprocess.CalledProcessError as e:
                print(f"Subprocess failed with error: {e}")
                return render_template("compress.html", check=-1,
message="Compression failed. Please try again.")
        else:
            print("Invalid file or no file selected.")
            return render_template("compress.html", check=-1,
message="Invalid or no file selected.")
    else:
        return render_template("compress.html", check=0)

@app.route("/decompress", methods=["GET", "POST"])
```

```
def decompress():
    """Handle file decompression."""
    if request.method == "POST":
        uploaded_file = request.files.get("file")

        if uploaded_file and uploaded_file.filename and
allowed_file(uploaded_file.filename):
            temp_filename = f"{uuid.uuid4()}-{uploaded_file.filename}"
            save_path = os.path.join(app.config["UPLOAD_FOLDER"],
temp_filename)

            try:
                uploaded_file.save(save_path)
                print(f"File saved to: {save_path}")
            except Exception as e:
                print(f"Error saving file: {e}")
                return render_template("decompress.html", check=-1,
message="File upload failed.")

            decompressed_file =
f"{os.path.splitext(uploaded_file.filename)[0]}-decompressed.txt"
            decompressed_path = os.path.join(app.config["DOWNLOAD_FOLDER"],
decompressed_file)

            # Check if decompression executable exists
            if not os.path.exists(DECOMPRESSION_EXE):
                print("Decompression executable not found.")
                return render_template("decompress.html", check=-1,
message="Decompression executable not found.")

            # Run decompression executable
            command = [DECOMPRESSION_EXE, save_path,
app.config["DOWNLOAD_FOLDER"]]
            print(f"Running command: {command}")
            print(f"Expected decompressed file path: {decompressed_path}")

            try:
                result = subprocess.run(command, check=True,
capture_output=True, text=True)
                print(f"Subprocess output: {result.stdout}")
                print(f"Subprocess error: {result.stderr}")

                # Additional logging to verify file paths and contents
                print("Files in the download directory after
decompression:")
                for root, dirs, files in
os.walk(app.config["DOWNLOAD_FOLDER"]):
                    for file in files:
                        print(f"Found file: {os.path.join(root, file)}")

                # Extract the actual decompressed file path from the
subprocess output if available
                decompressed_file_actual = None
                for line in result.stdout.splitlines():
                    if "Decompressed file path:" in line:
                        decompressed_file_actual = line.split("Decompressed
file path:")[-1].strip().strip('"')
                        print(f"Actual decompressed file path from output:
{decompressed_file_actual}")
                        break
                if decompressed_file_actual and
os.path.exists(decompressed_file_actual):
                    print("Decompression successful.")

                    # Remove uploaded .bin file after decompression
                    if os.path.exists(save_path):
                        os.remove(save_path)

                    os.chmod(decompressed_file_actual, 0o777)
                    return render_template("decompress.html", check=1,
filename=os.path.basename(decompressed_file_actual))
                else:
                    print("Decompressed file not found.")
                    return render_template("decompress.html", check=-1,
message="Decompressed file not created.")
            except subprocess.CalledProcessError as e:
                print(f"Subprocess failed with error: {e}")
                print(f"Command output: {e.stdout}")
                print(f"Error output: {e.stderr}")
                return render_template("decompress.html", check=-1,
message="Decompression failed. Please try again.")
        else:
            print("Invalid file or no file selected.")
            return render_template("decompress.html", check=-1,
message="Invalid or no file selected.")
    else:
        return render_template("decompress.html", check=0)

@app.route("/download/<filename>")
def download_file(filename):
    """Handle file download."""
    download_path = os.path.join(app.config["DOWNLOAD_FOLDER"], filename)

    if os.path.exists(download_path):
        return send_file(download_path, as_attachment=True)
    else:
        return render_template("error.html", message="File not found.")


if __name__ == "__main__":
    print(f"Starting server. Upload folder: {UPLOAD_FOLDER}, Download
folder: {DOWNLOAD_FOLDER}")
    app.run(debug=True)
```