Adaptive Numeral System (ANS) and its Applications in Data Compression

Omaar Ayman Dadoch

Department of Electronic and Electrical Engineering

Brunel University London

May 2024

Abstract

This research explores innovative methodologies for advancing lossless data compression by developing adaptive numeral systems to calculate and reduce binary data representations effectively. Building on the foundational theories of Shannon and Kolmogorov, the study introduces the Adaptive Numeral System (ANS), Improved Adaptive Numeral System (IANS), and Modified Adaptive Numeral System (MANS), novel approaches for adaptively calculating binary values. These systems can be shown to exhibit the unique capability of compressing each segment iteratively, thereby reducing the overall data size progressively and form the basis for an iterative and progressive approach to data compression.

To leverage these adaptive numeral systems, the study presents the Data Extraction (DE) technique, a compression framework that uses MANS to perform conversions from binary values into more compact representations. DE achieves significant compression rates, demonstrating competitive performance compared to traditional methods like Huffman coding, particularly in its fully decodable state before binary conversion. Furthermore, the research addresses challenges in identifying flag locations within segmented data, proposing a range of solutions to enhance compression efficiency and reliability.

The combined contributions of ANS, IANS, MANS, and DE represent a significant advancement in the field of lossless compression, particularly in their ability to process already compressed data and transform non-prefix codes into prefix codes. These advancements hold substantial promise for applications in areas such as medical imaging, digital media, machine learning, artificial intelligence, embedded systems, and the Internet of Things.

Acknowledgement

I extend my heartfelt appreciation to my previous supervisor, Professor Abdul Sadka, for his invaluable guidance, unwavering support, and constant encouragement throughout the majority of my PhD research journey.

I am sincerely grateful to my current supervisor, Dr. Ruiheng Wu, for his ongoing support and valuable insights into my research.

I also wish to thank Dr. Nikolaos Boulgouris, my second supervisor, for providing constructive feedback that has challenged me to broaden my horizons.

I am deeply thankful to QA for their financial support, which played a crucial role in making my research endeavours possible. Special acknowledgement goes to Ben Sweetman for his encouragement and support.

A heartfelt appreciation is extended to Professor Sirkku Aaltonen for her invaluable guidance and unwavering support throughout my academic journey. Her insightful advice and meticulous review of my thesis greatly enriched its quality.

I would like to express my profound gratitude to Professor Gerald Porter for his support and insightful advice. Additionally, I extend my gratitude to my brother Anas Dadoch for his engaging discussions on my research topics, and to my friend Barz Surchi for the countless insightful conversations we've had regarding my research endeavours.

Finally, I am indebted to my wife, Jessica and our daughters, Sophia and Matilda, for their unwavering support and understanding throughout this journey.

Table of contents

List of Figures

List of Tables

List of Symbols

ANS	Adaptive Numeral System
D	The encoder/Decoder
b	The binary bit of a random source
S	Random binary string
IANS	Improved Adaptive Numeral System
A	Natural number, representing 1's from a binary number
B	Natural number, representing 0's from a binary number
NO	Number of Operations
DE	Data Extraction
MANS	Modified Adaptive Numeral System
DE-FI	Data Extraction - Flag Information
DE-FTF	Data Extraction - Flag to Flag
C_l	The length of C in bits (the selected numbers of bits to compress)
$C_x = \sum_{i=0}^{i=c_l+1} 2^i b_i$	The decimal value of C, where b is the binary bit
F_l	The flag length
$F_{Sg_x} = C_x$	The flag location for each segment
$S(sg_x)_l = C_x$	Length of segment x
$S(SGs) = \frac{S_l - S(sg_x)_l}{C_l - F_l}$	Number of Segments in S
$Sg_l = 2^{C_l}$	The maximum segment length
Sg_l / F_l	The number of flags the segment can hold
$S(sg_x) = \sum_{i=0}^{i=S(sg_x)_{l-1}} 2^i b_i$	The decimal value of a segment S_{sg_x}

Chapter 1

1.1. Introduction

This section discusses the increasing demand for higher data compression ratios, driven by the rapidly growing volume of digital data. It outlines the limitations of existing compression methods, which are often constrained by the binary system and the theoretical limits defined by Shannon's Entropy. The section also introduces a novel approach involving an adaptive numeral system, which aims to restructure binary-encoded data into iterative sequences, potentially unlocking new levels of compression efficiency. Additionally, it highlights key research questions regarding the feasibility, compatibility, and computational viability of this new method, setting the stage for the study's proposed innovations in lossless data compression.

1.2. Background and Motivation

The need for a higher ratio of data compression is increasing. This has become clear from the figures, published, for example, by International Business Machines (IBM). According to IBM [1], the amount of data generated every day was more than 2.5 exabytes in 2012, and it is forecasted to reach 180 zettabytes (ZB) by 2025 [2]. The majority of the generated data has been transmitted from one point to another, where a method of compression has been applied to either store it or transmit it. Although there are many methods of compression designed around statistics, redundancy, probability, combinations, permutations, or arithmetics with various ratios of data compression, these are not sufficient for dealing with the amount of data used daily. The paradigms designed so far are based on the binary system, which is one way of translating occurrences of permutations and, therefore, restricted to the theoretical limit of compressibility, the so-called Shannon's Entropy, which gives the minimum number of bits per symbol needed to encode information in binary form.

Most lossless compression aims to eliminate the redundancy in information [3] and to try to find a new way of representing information as efficiently as possible. Still, it cannot go beyond the theoretical limit (entropy) [4]. If, however, the encoded information were transformed into a different form, it might unlock the limit of compression. In other words, it might be possible to calculate the occurrences of data represented in binary form in sequences and then compress them into a reasonable size where each compressed sequence could be used to calculate the next sequence. This would mean that the final sequence would be the result of the information contained in it, and the form of these calculations would be reversible and reducible.

This study aims to improve the lossless compression of information by the application of new methods of calculating the occurrences of data represented by a binary system and also finding appropriate algorithms to reduce the results of these calculations. This improvement will open up an entirely new perspective on lossless compression. At the same time, it will take into account the complexity, memory size, and speed of the encoding and decoding of data. The assumptive hypothesis in this study posits the feasibility of calculating occurrences of data represented in binary form within sequences while compressing them into a manageable size. Each compressed sequence is envisioned to serve as a basis for calculating the subsequent sequence. Encoding each sequence is anticipated to further compress the preceding segment, resulting in the number of encoded segments aligning with the frequency of compression applied to the initial segment. This approach aims to enhance lossless compression of information through the design of a novel adaptive numeral system for data compression. Testing the efficiency of the new model will involve addressing the following set of questions:

1) Would the new numeral system calculate a compressed data stream from the binary numeral system?

    Method: This will be tested by applying the method to a data stream that needs to be compressed. The outcome of the previous data streams will be used to encode a new data stream, and the process will be reversed to check if the method is recursive.

2) Could any additional method of compression be applied to the new numeral system to further increase the compression ratio?

    Method: Other compression algorithms, such as Huffman, BWT, RLE and Arithmetic coding, will be applied to the new method to evaluate its effectiveness.

3) Would the new algorithm be computable?

Method: The algorithm will be developed using a Java application to assess its computability.

4) Would the redundancy limit of information change with the application of the new adaptive numeral system?

In total, the thesis comprises eight chapters, structured to provide a clear progression from foundational concepts to advanced methodologies and comparative analyses. The organisation reflects the study's evolution, addressing both theoretical underpinnings and practical applications of compression techniques. The report on the study continues as follows:

Chapter 2: discusses earlier research that forms the framework for modern compression methods. It examines foundational theories, such as Information Theory and Algorithmic Information, and contemporary techniques like Huffman Coding, Arithmetic Coding, and Asymmetric Numeral Systems. This chapter provides the groundwork upon which the current study is built, illustrating how the developments rely on prior work. The chapter highlight challenges that the subsequent chapters will address, especially the development of numeral systems for more efficient compression methods. This sets the stage for Chapter 3, where the focus shifts to the development and application of Adaptive Numeral Systems.

Chapter 3: Introduces the concept of numeral systems, particularly the Adaptive Numeral System (ANS) and Improved Adaptive Numeral System (IANS). These systems are designed to prepare data for compression by encoding and segmenting it iteratively. The chapter highlights the challenges of encoding data in a manner that allows for efficient compression. The discussion of ANS and IANS transitions into Chapter 4, where the emphasis shifts towards applying these systems for data compression and exploring the effectiveness of iterative compression methods based on these numeral systems.

Chapter 4: Builds upon the theoretical foundations set in Chapter 3. It discusses how the ANS and IANS can be used for iterative compression, where each segment's result is used to encode the subsequent segment. The chapter explores the limitations of initial ANS-based compression methods when applied to IANS, leading to the development of a conditional compression method. This modification takes advantage of the unique properties of IANS, such as segment initialisation and structural differences. The transition into Chapter 5 is marked by a focus on practical applications, specifically the challenges of data extraction (DE) and how these concepts can be used to improve compression techniques in real-world scenarios.

Chapter 5: Introduce Data Extraction (DE) and address its practical challenges. The chapter builds on the concepts discussed in Chapter 4, where iterative compression techniques were explored. It introduces the Flag approach which includes three solutions to overcome the identified challenges of DE, specifically in relation to data representation and encoding. The chapter emphasises the application of these systems in compressing data, with particular attention paid to MANS (Modified Adaptive Numeral System) as a flagging system solution. The transition into Chapter 6 involves a more in-depth exploration of how MANS specifically enable the DE process.

Chapter 6: The focus of this chapter is on the Modified Adaptive Numeral System (MANS) and its application to address specific challenges encountered in the Data Extraction (DE) technique. Building upon the issues identified in Chapter 5, this chapter demonstrates how MANS, with its distinctive characteristics, enables the implementation of necessary flagging solutions within DE. The DE method relies on a numeral and flag system capable of supporting these solutions, and MANS fulfills this requirement by efficiently encoding and decoding data while facilitating segment flagging during the compression process. The chapter explores the encoding and decoding processes of MANS, explaining how its properties are leveraged to enhance the DE technique. This sets the stage for Chapter 7, where the three proposed solutions introduced in Chapter 5 are tested, and a detailed comparative analysis of various compression methods, including DE with MANS, is presented.

Chapter 7: Provides a detailed comparative analysis of the DE technique proposed in Chapter 5, based on MANS, and evaluates its performance in relation to other established compression methods. This chapter offers a deeper understanding of the results obtained from Solution I and Solution II using MANS, with a thorough examination of how these solutions address the challenges of DE. However, Solution III is not as robust and does not offer significant comparative results, thus requiring less in-depth analysis. The transition to Chapter 8 concludes the study, summarising the findings, providing conclusions, and suggesting potential directions for future research.

Chapter 8: Draws together the findings from the previous chapters and offers a comprehensive conclusion. It highlights the contributions made by the study to the field of lossless data compression, specifically through the development and application of ANS, IANS, and MANS. The chapter also reflects on the strengths and limitations of the methods discussed, including the results from the DE technique and its applications. Additionally, Chapter 8 proposes areas for future research based on the gaps identified in the study and the outcomes of the comparative analyses conducted in Chapter 7.

1.3. Summary

This section explored the increasing need for more efficient data compression methods due to the rapid growth in digital data. It examined the limitations of existing compression techniques, particularly those based on the binary system and Shannon's Entropy, which define the theoretical limits of compressibility. The section introduced a novel approach involving an adaptive numeral system, aimed at restructuring data into iterative sequences to potentially surpass these limits. Key research questions were outlined, focusing on the feasibility, computational requirements, and integration of this novel method with existing compression algorithms.

Chapter 2

2.1. Introduction

This chapter provides an overview of earlier work on lossless compression, establishing the theoretical foundation upon which modern techniques are built. It begins with a discussion of Information Theory and Algorithmic Information, highlighting fundamental principles such as entropy and redundancy that govern data compression. The chapter then explores contemporary lossless compression techniques, covering widely used methods such as Huffman Coding, Run-Length Encoding (RLE), Burrows-Wheeler Transform (BWT), Fibonacci Code, Arithmetic Coding, Integer Arithmetic Coding, and Asymmetric Numeral Systems (ANS). Each method is examined in terms of its underlying principles, efficiency, and applications, setting the stage for the development of novel compression paradigms in subsequent chapters.

2.2. Earlier Work on Lossless Compression

In general, earlier work on lossless compression shows that there are several perspectives that can be applied to the increase of data compression, but all have certain reservations. Moreover, an effective method for lossless data compression can be evaluated in a number of ways. These include, for example, the analysis of the compression ratio and complexity of the applied algorithm, which, in turn, affects the processing power and speed required to implement the algorithm. In addition, lossless data compression techniques appear to have varying compression ratios, which depend on data patterns and/or the type of data (e.g. image, video, sound and text), and they frequently encounter serious challenges such as the expansion of data, long encoding and decoding speed, error propagation and library growth. Overall, these techniques aim to compress data by removing redundancy from the original data, but this can only be done to a certain point as the compressed data has little or no redundancy [4].

Compression can, in principle, be achieved by using one of the two classes, either lossless or lossy compression. In lossy compression, some information is discarded and cannot be retrieved, while in lossless compression, information can be reconstructed without any loss. Lossless compression generally includes three types. In the entropy type, a probability module is used as a measure of the amount of compression that can be obtained to help design and implement an effective algorithm for data compression. The dictionary type operates by searching for matches between the text that needs to be compressed and a set of strings that are stored in a dictionary. In other types, for example, counting sequences of repeated data or arranging sequences of data in a lexicographic order allows other algorithms to take advantage of the run.

In the present chapter, the approaches and techniques of data compression will be discussed in the following subsections: Information theory and algorithmic information and Current lossless compression methods, which will be divided into different approaches based on chronology.

2.3. Information Theory and Algorithmic Information

Claude Elwood Shannon outlined the theoretical limits of data compression in 1948 by defining the entropy of a binary system as the minimum number of bits per symbol needed to encode information in binary form. This formula has become known as Shannon's Entropy:

$$H(s) =- \sum_{i=1}^{n} pi\, Log_2(pi) \qquad (2.1)$$

where $H(s)$ is the entropy of a set of probabilities $p1, p2...pn$[5]. The equation 2.1 is applicable to the source distribution determined by the compression method, enabling comparison with the encoded information to ascertain the degree of available compression. Some researchers have, however, argued that Shannon's theory has theoretical difficulties when real-world data is involved [6]. For example, if the present document were to require compression, the use of the relative frequency approach would mean that the probability of each letter or word could be calculated and the estimated entropy obtained. This would be the approximate amount of compression that could be achieved. However, the letters and words examined in the present document are established and unchanging, as opposed to being subject to probability, which would make the estimation of entropy futile. Yet again, if the algorithm had been designed for the current document, it would apply only to this particular document and not necessarily to any other document. In consequence, basing an algorithm on calculating the probabilities for the English alphabet would be more useful, but maybe not as efficient for other documents or text.

The understanding of entropy currently assumes the existence of an abstract source, and it is still useful as it provides the limit of compression it can achieve [6]. This limit appears to depend on how information is interpreted. For instance, when considering x as a set of increasing numbers [0,1,2,3,4,5,6,7] that require compression, where the probability of each number is equal, Shannon's theory suggests that the minimum number of bits required to encode each symbol is three bits. This would result in 24 bits. However, the information contained in x shows that it is a linearly increasing sequence of ones. By using the delta coding approach, the differences between consecutive values in x can be transmitted as [0,1,1,1,1,1,1], resulting in an entropy of $H(x) = 1$. [8].

Algorithmic information theory, exemplified by Kolmogorov complexity [9] sidesteps the theoretical quandary of Shannon's entropy by adopting a distinct viewpoint on information. Kolmogorov complexity assesses information by examining a data string's length in binary bits, identifying patterns within it, and thereby enabling a shorter description of it. More precisely, the Kolmogorov complexity $K(x)$ of a sequence x is the size of the program that includes all the input that might be needed to generate x. This means that if x is a sequence of repeated data, for example, all zeros, the program will simply be a print statement in a loop. However, if x is a random sequence without a pattern, the only program that could generate it would contain the sequence itself. A typical example is the Minimum Description Length principle, introduced by Jorma Rissanen in 1978 and known as MDL, which compresses the data by describing any regularity in a set of data in fewer symbols.[10][11]

This shows that redundancy of information can be removed by either encoding information by using the relative frequency approach, which is limited to the entropy, or by using a method of looking at the duplication in the encoded data, which will, in turn, be limited to the number of duplication in the data stream.

2.4. Contemporary Lossless Compression Techniques

Given that all lossless compression relies on the binary system, a comprehensive review of earlier research will illuminate the techniques employed in compressing data, which can then potentially be adapted to the new numeral system with appropriate modifications. If none of the existing methods are applicable, this research will present techniques for developing a novel compression method. This subsection encompasses an in-depth examination of Huffman codes, Run Length Encoding, Burrows-Wheeler Transform, and Fibonacci code, along with further exploration of Arithmetic coding. Additionally, recent advancements in faster and more efficient arithmetic coding, such as An Efficient Adaptive Binary Arithmetic Coder Based on Logarithmic Domain, are discussed. This coder is a versatile compressor that integrates various lossless transforms to store non-media files such as text, source code, serialised data, and other binary content. will also address Asymmetric numeral systems, which are precise entropy coding algorithms exhibiting encoding speeds comparable to Huffman coding. Moreover, they possess the capability to approach optimal entropy levels arbitrarily, a characteristic shared with Arithmetic coding. Examples of such compression algorithms include GZIP by Jean-loup Gailly and Mark Adler [12], BZIP2 by Julian Seward [13], and LZMA by Abraham Lempel, Jacob Ziv, and Sergey Markov [14].

The next section will delve into Information theory and algorithmic information, setting the foundation for understanding the subsequent discussion on current lossless compression methods, which will serve as the cornerstone of the research.

Since Shannon and Kolmogorov's work, most of the work done on lossless data compression has focused on creating new ways of applying modern data compression theory, designed around statistics, probability and combinations and/or arithmetics. These paradigms have employed a variety of different compression techniques and include, for example

- Identifying the data that occurs the most frequently and encoding the shortest binary number to represent the information. An example of this is Huffman coding [15].
- Counting sequences of repeated data as is done in Run-Length Coding [16].
- Arranging the sequences of data in lexicographic order by using the Burrows-Wheeler transform to allow other algorithms, such as RLE, to take advantage of this run [17].
- Using adaptive dictionary-based techniques which store a sequence of data and use them if the same sequence occurs again, and, if it does not, the sequence is stored in the dictionary [18], for example, LZ77 and LZ78 by Abraham Lempel and Jacob Ziv [19] [20].

- Generating a unique identifier by using the numbers in the unit interval (0,1) to distinguish a sequence of symbols to be encoded such as Arithmetics [21][22][23][24][25].
- A general-purpose compressor which combines various lossless transforms to save non-media files such as text, source code, serialised data, and other binary content [26]. Examples of this include GZIP [12], BZIP2 [13], LZMA [14].
- Asymmetric Numeral Systems generate a list of natural numbers in a table based on the frequencies of the symbols. The symbols are processed by reading the discrete integers representing the symbols and the discrete integer is transited to the row number ready to process the next symbol J. Duda 2009 [27].

In what follows, some these paradigms will be discussed in more detail

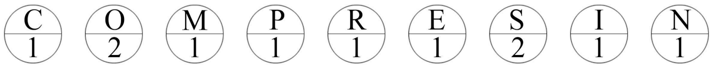
2.4.1. Huffman Coding

Huffman coding has been, and remains, a very popular method in data compression and one of the most well-known paradigms in information theory. It was published in 1952 by David Huffman [15], who originally designed it for text compression. It has since been modified in many ways and used to improve the ratio of compression [28]. The method has been used partially as the back end of GZIP [12], ZLIB [29] and JPEG [30].

The Huffman coding algorithm stands out as a groundbreaking variable-length coding technique. Unlike employing a uniform fixed-length code, such as 8-bit extended ASCII, for each symbol, it generates codewords based on a set of probabilities. It assigns shorter codewords to symbols with higher occurrence rates, while less frequent symbols receive longer codewords. As a result, the total number of bits needed for representation is substantially decreased for a source comprising symbols of varying frequencies.

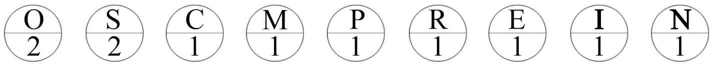
The application of the method starts by determining the frequency of the symbols and listing them in either descending or ascending order according to the associated probability distribution. Then the symbols are listed to form a tree with leaves, and combining two symbols repeatedly to form a new subtree. This is first done with the lowest probability symbols and continues until all the symbols are included in the tree. The lowest probability symbols will be combined and placed in the subtree in an order which is based on the total frequency of the combined letters, while each node on the tree will be assigned either $\{1\}$ to the left branch and $\{0\}$ to the right branch or vice versa, which will create the prefix code. For example, when using Huffman coding to encode the word “COMPRESSION”, the steps in Figure 2.1 show how the tree is built from the leaves to the root. Figure 2.1(a) illustrates the first step of listing the frequency in each letter.

Figure 2.1(a)



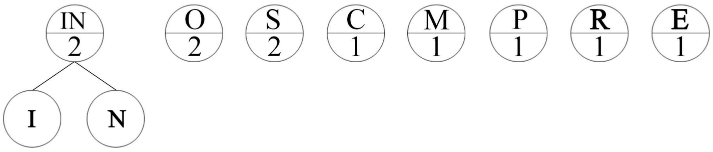
The subsequent step entails reordering the letters based on their respective frequencies. In the example illustrated in Figure 2.1(b), the rearrangement will follow a descending order. The last two items, which have the minimum frequencies, will be combined and placed in the subtree.

Figure 2.1(b)



The least and last frequent symbols on the list are I and N, which have been combined in Figure 2.1(c). The frequencies of the symbols are added together, and, in this example, placed in the first position to maintain the sorted order of the list.

Figure 2.1(c)



The symbols are then repeatedly combined to complete the tree as shown in Figure 2.1(d) to Figure 2.1.

Figure 2.1(d)

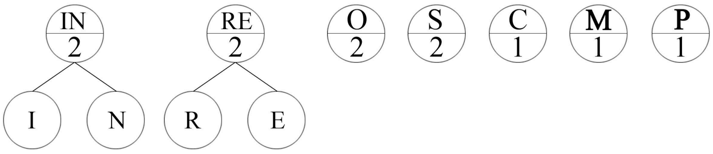


Figure 2.1(e)

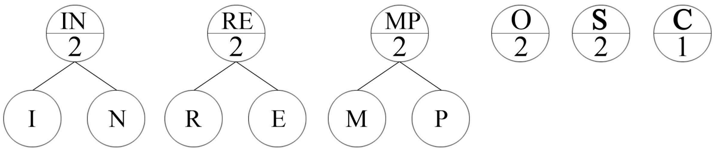


Figure 2.1(f)

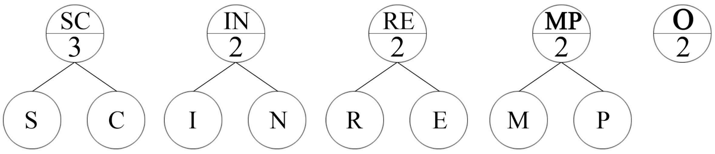


Figure 2.1(g)

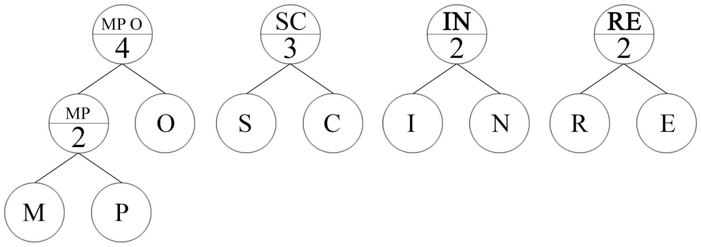


Figure 2.1(h)

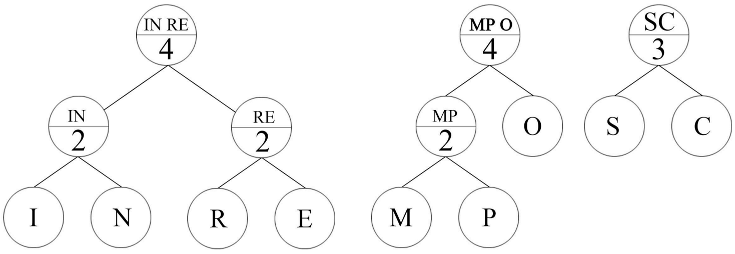
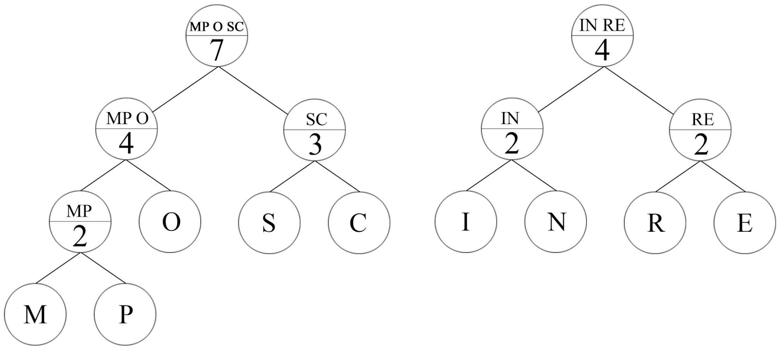
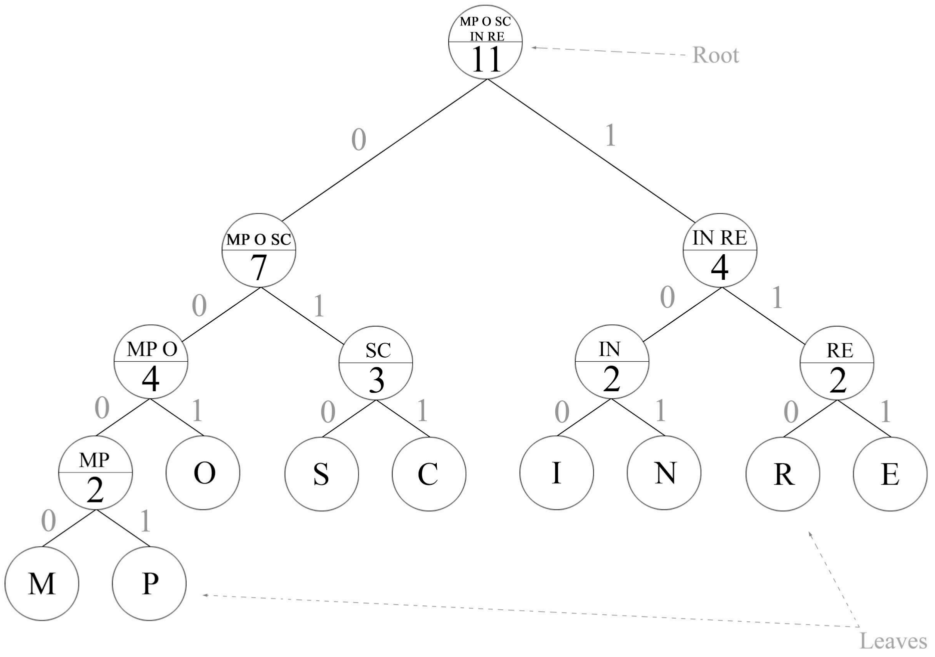


Figure 2.1(i)



Once the final two subtree symbols are combined, the 1 and 0 are assigned to branches, by inserting 1 to the left branch and 0 to the right or vice versa as shown in Figure 2.1.

Figure 2.1: Building a Huffman tree



The weighted binary tree completes the Huffman tree. As shown in Figure 2.1, each symbol has been constructed with prefix code that can be derived by collecting the 1's or 0's of each branch to generate the required symbol. Table 2.1 shows the codes for each letter obtained from the Huffman tree constructed from Figure 2.1. This gives the Huffman code for the word "COMPRESSION" as 011 001 0000 0001 110 111 010 010 100 001 101.

Table 2.1: Huffman code for word "Compression"

Letter	Probability	Codeword
C	0.09	011
O	0.18	001
M	0.09	0000
P	0.09	0001
R	0.09	110
E	0.09	111
S	0.18	010
I	0.09	100
N	0.09	101

Decompression can be done by following the branch from the root that leads to each symbol. The decoder sequentially reads the individual bits, either 1's or 0's, starting from the root and progressing downwards, as illustrated in Figure 2.1. If the symbol is 1, the decoder advances one step to the right subtree branch; conversely, if it's a 0, the decoder moves one step to the left. This iterative process continues until the decoder reaches a leaf node, where it decodes the symbol. Subsequently, the decoder restarts from the root and repeats this process until it reaches the end of the encoded message. The entropy for the word "COMPRESSION" is: $H =-\left(\frac{1}{11}\right) log_2\left(\frac{1}{11}\right) + \left(\frac{2}{11}\right) log_2\left(\frac{2}{11}\right) + \left(\frac{1}{11}\right) log_2\left(\frac{1}{11}\right) + \left(\frac{1}{11}\right) log_2\left(\frac{1}{11}\right) + \left(\frac{1}{11}\right) log_2\left(\frac{1}{11}\right) + \left(\frac{1}{11}\right) log_2\left(\frac{1}{11}\right) + \left(\frac{2}{11}\right) log_2\left(\frac{2}{11}\right) + \left(\frac{1}{11}\right) log_2\left(\frac{1}{11}\right) + \left(\frac{1}{11}\right) log_2\left(\frac{1}{11}\right) \approx 3.095$ bits/symbol.

The length L for this code, as per Table 2.1, is calculated as L = 3 + 3 + 4 + 4 + 3 + 3 + 3 + 3 + 3 + 3 + 3 = 35. Given that the number of symbols in the word "COMPRESSION" is $n = 11$, the average code length L is approximately:

$$L_{Avg} = \frac{35}{11} \approx 3.182 \text{ bits/symbol}$$

The efficiency of this code (redundancy) is measured by the difference between the entropy and code length, which in this case is: 3.182 - 3.095 = 0.087 bits/symbol. This corresponds to approximately 0.023% of the entropy. Therefore, the example (the word 'COMPRESSION') requires 0.023% more bits than the minimum of required bits to be encoded using Huffman code. However, if the frequency of a letter is very high compared with the other letters, the Huffman code can become inefficient as regards to the entropy. For example, if a source from the alphabet $A = \{a_1, a_2, a_3, a_4\}$ is used, that has the probability model: $P(a_1) = 0.9$, $P(a_2) = 0.06$, $P(a_3) = 0.03$, $P(a_4) = 0.01$. The entropy will be 0.6 bits/symbol. Table 2.2 shows the application of the Huffman code for this source:

Table 2.2: Huffman code for Source *A*

A	Probability	Codeword
a_1	0.9	1
a_2	0.06	01
a_3	0.03	000
a_4	0.01	001

The average length of this code is 1.14 bits/symbol and it has a redundancy of 0.54 bits/symbol. This means that using this sequence will require 0.54 more bits per symbol than the minimum calculated by the entropy.

There are many ways, such as the minimum variance Huffman codes [31] for combining the symbols into a frequency table. Huffman codes can be arranged by placing the letters with the combined lowest probabilities as high as possible on the list.
For example, if there are symbols a_1, a_2, a_3, a_4 with probabilities 0.6, 0.1, 0.2, 0.1 respectively. P_{a4} will be combined with P_{a3}, and since the probability of the combined symbols is $P_{a4} + P_{a3} = 0.3$, $(P_{a4} + P_{a3})$, it will be listed before a_2, resulting in the following order a_1, (a_4, a_3), a_2 . This leads to different binary trees and different codeword lengths for each letter. The redundancy measure will, however, remain the same when compared with the standard Huffman code.

When the Huffman code is obtained, the decoder requires the same model of the encoded symbols. The model can be sent to the decoder, but it will result in costly memory usage in terms of storing the model of encoded symbols. Also, when the statistics indicate wide variation, the coded tree will result in a significant amount of encoded data. Canonical Huffman codes [32] tackle these challenges by transmitting the codeword lengths of Huffman trees to the decoder, thereby minimising the data required for transmission or storage. Nonetheless, this approach entails constructing the Huffman tree and encoding/decoding the data according to its structure. Although the concept is straightforward, efficiently building the tree and managing edge cases can introduce complexity. Ongoing research aims to optimise Huffman coding by introducing a k-bit delay [33].

Hashimoto and Iwata demonstrate the potential for optimality using a class of 1-bit delay decodable codes with a finite number of code tables. They used a code-tuple model for a time-variant encoder and defined the class of k-bit delay decodable code-tuples. They established that Huffman code can achieve optimal average codeword length within the class of 1-bit delay decodable code-tuples, under the constraint that the encoder's code table selection depends only on the current symbol, independent of past symbols. Future work aims to explore if relaxing this constraint can result in shorter average codeword lengths compared to Huffman coding [34].

2.4.2. Run-Length Encoding

Similar to Huffman coding, Run-Length Encoding (RLE), introduced by Solomon W. Golomb in 1966 [35], is another common technique used in data compression. While RLE simplifies data by substituting repeated symbols with a count, Huffman coding assigns variable-length codes to symbols based on their frequencies. Both methods aim to decrease data size, with RLE being more straightforward but less versatile compared to Huffman coding. RLE focuses on consecutive symbols and substitutes them with a single symbol along with the count of the consecutive run. For example, the message sequence aaaabbccc could be transformed to (a,4),(b,2), (c,3), and once it has been transformed, a probability coder such as the Huffman code can be used to code both the symbols and the number of the run for each letter. This method is very useful for data sets that occur in succession, whereas it is not effective if the message sequence has a small run of symbols. Research on Run-Length Encoding (RLE) continues to address challenges associated with datasets containing small runs of symbols. Xutan Peng, Yi Zhang, Dejia Peng, and Jiafa Zhu [36] employ combinatorics to quantify RLE's space savings potential based on input distribution. Leveraging this understanding, they propose an algorithm that automatically identifies suitable symbols for RLE encoding, selectively applying RLE to these symbols while storing others directly. Through experiments on real-world datasets, they demonstrate that their approach maintains RLE's efficiency advantage and can effectively mitigate cases where RLE encoding results in increased space compared to the input data. However, the analysis was limited to the context of independent and identically distributed data.

2.4.3. Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) was developed by D. J. Wheeler in 1983, but it was not published until 1994 [37][38]. The BWT algorithm does not compress data, but, instead, it's been used as a structure for other compression techniques. BWT makes use of a reversible transformation to align the contexts of data and to arrange them in lexicographic order, making it easy to use by other compression algorithms, such as the Run-Length encoding [35], front coding, and Intelligent Dictionary Based Encoding [39].

The algorithm requires the entire sequence input for its arrangement. An illustrative example below demonstrates how the BWT algorithm can be applied to the word 'DATA', which has 4 characters, thus $n = 4$. The process starts with shifting the first symbol, in this case, D, to the left, after which it will be at the end of the word, 'ATAD'. The shifting will be repeated $n - 1$ times, which will result in $n \times n$ matrix as shown in Table 2.3.

Table 2.3: Burrows-Wheeler Transform for word DATA

3.1					3.2	**F**			**L**
	D	A	T	A		A	D	A	T
	A	T	A	D		A	T	A	D
	T	A	D	A		D	A	T	A
	A	D	A	T		T	A	D	A

The second step is to sort the rows in lexicographic order and assign F for the first column and L for the last, as shown in Table 2.3. The first letter of the word must be noted, or alternatively, the $ sign can be inserted to identify the beginning of the string. The order of the repeated letters in column L will match the order in column F. For example, the first A in column L will correspond to the first A in column F. Column L will be arranged and used to decode the data string to its original form.

Decoding of L can recover the original word in the following steps:

i. Finding the string F by sorting L in lexicographic order.
ii. Starting from the first noted letter D in Column L, the second letter of the word can be retrieved by looking at the corresponding letter in column F.
iii. A new letter in column F will indicate what letter needs to be looked for next in column L. The order of each letter in column L matches the order of letters in column F.

The algorithm serves as a foundation for various compression methods, such as the bzip compressor, and proves effective when coupled with compression techniques like Run-Length-Encoding. However, BWT necessitates the complete sequence to be accessible prior to the encoding process. BWT recently gained popularity in the field of genetics and bioinformatics due to its versatility, offering solutions for data compression, sequence analysis, haplotype inference, and genome assembly. Its widespread adoption is driven by its ability to address key challenges in managing and analysing complex genetic data [40] [41][42].

2.4.4. Fibonacci Code

The Fibonacci numbers and their associated golden ratio, typically denoted by φ, are widely recognised mathematical concepts. These numbers have been found in applications and are observed across various disciplines, including nature, music, market trading, number theory, physics, quantum mechanics, cryptography, and data compression [43]. The first two Fibonacci numbers are defined as $F_0 = F_1 = 1$, initiating a sequence of numbers where each subsequent integer is the sum of the two preceding integers, for example, $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$, resulting in the sequence {1, 1, 2, 3, 5, 8, 13, ...}. These are known as Fibonacci numbers of order 2. Over time, numerous codes have been developed using the Fibonacci numbers as a foundation, as elucidated by Salomon [44], among others.

One of the best-known representations is Zeckendorf's theorem [45], which represents any integer as sums of Fibonacci numbers, while at the same time avoiding the sum of any two consecutive Fibonacci numbers. The paradigm is referred to as Zeckendorf representation $Z(N)$, and it results in a code which does not contain adjacent 1-bits. This characteristic has been used to act as a termination flag for each encoded integer to differentiate, unambiguously, the code words from a long stream of codes [46]. This is achieved by omitting the first bit as F_1 which is, then, followed by the least significant bit. For example, the number 19 can be represented using Fibonacci numbers as 13+3+2+1=19 {100111} and 13+5+1=19 = {101001}, where the latter is Zeckendorf representation as it avoids the sum of any two consecutive Fibonacci numbers. Additional {1} will be inserted next to the least significant bit as a terminational flag of the code word, Z(19) = {1010011}. Thus, each encoded integer, which uses the Zeckendorf representation, can be decoded uniquely by appending the prefixed adjacent {1} to the end of each code word.

The Fibonacci codes are recognised for their efficacy in compressing a limited set of integers [47], offering reliability in data communication applications with minimal error susceptibility [48]. Consequently, when employed as a compression method for transmitting unbonded strings [49], the decoding process can be expedited through the use of algorithms based on finite automaton. Coupled with a precomputed mapping table of automaton reduction using Fibonacci shift operation, as demonstrated by Walder, Krátky, et al. [50], this approach enhances efficiency. The Fibonacci codes are considered a straightforward and practical alternative to Huffman codes, as demonstrated by Przywarski, Grabowski, et el [51].

2.4.5. Arithmetic Coding

Modern Arithmetic coding was developed by Pasco [52] in 1976, and it attracted a great deal of interest. It did not, however, take long before its major drawback, the problem of finite precision, was discovered. At the same time, however, the problem was solved by Rissanen [53], and Pasco's paradigm was then further developed by many researchers to provide a practical algorithm. One of the best-known algorithms has been developed by Rissanen and Langdon at IBM in 1979 [54].

Arithmetic coding is still an increasingly popular method for generating variable-length codes. It generates a unique identifier, or a tag, to represent the entire message with the corresponding binary number. This identifier is based on the cumulative probability of the message. In this respect, it differs from Huffman's coding, which separates the input symbols and generates a code to identify them. Arithmetic coding locates the position of the original message in a subinterval between 0 and 1, after which, the binary code can be generated to represent the message. For example, assume that the following source alphabet $A = \{a_1, a_2, a_3\}$ has the probabilities of $P(a_1) = 0.7$, $P(a_2) = 0.06$ and $P(a_3) = 0.24$, each symbol a_i can be tagged with a unique value $\overline{T}_X(a_i)$, using the following equation:

$$\overline{T}_X(a_i) = \sum_{k=1}^{i-1} P(X = k) + \frac{1}{2}P(X = i) \qquad (2.2)$$

$$= F_x(i - 1) + \frac{1}{2}P(X = i)$$

Determining the tag for a_1 follows from the equation (2.2) and becomes:

$$\overline{T}_X(a_1) = P(X = 0) + 0.5 \,.\, P(X = 1) = 0 + 0.35 = 0.35$$

Similarly, finding the tag for a_2 takes place as follows:

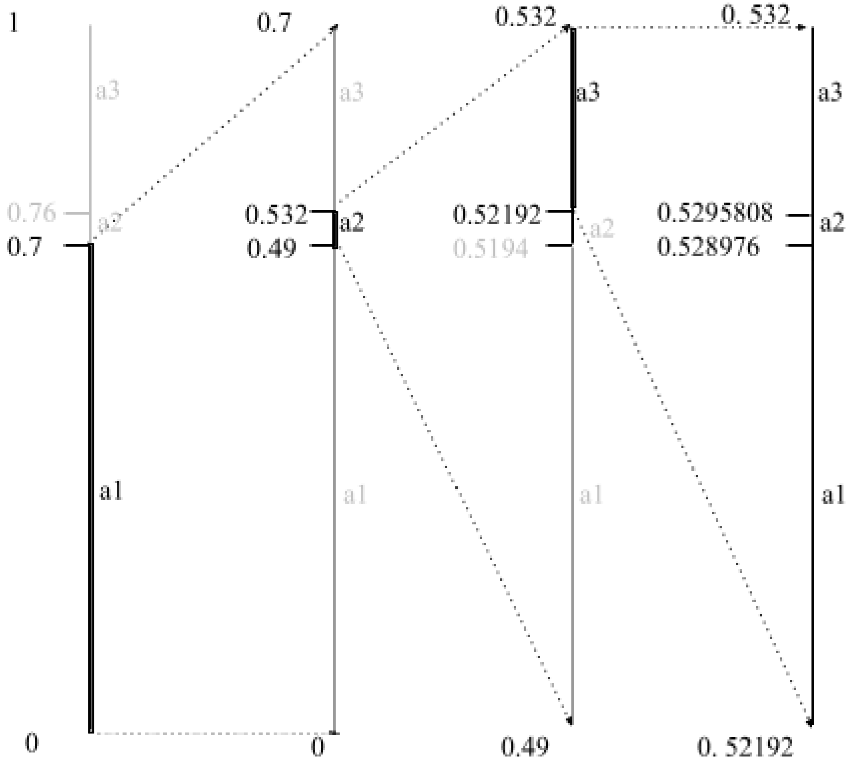
$$\overline{T}_X(a_2) = P(X = 1) + 0.5 \,.\, P(X = 2) = 0.7 + 0.03 = 0.73$$

And finding the tag for a_3:

$$\overline{T}_X(a_3) = \sum_{k=1}^{a_2} P(X = k) + 0.5 \,.\, P(X = 3) = 0.76 + 0.12 = 0.88$$

To tag the full sequence a_1, a_2, a_3 the symbols will be subdivided between [0, 1), based on their probabilities, as shown in Figure 2.2.

Figure 2.2: Subdivision of the interval [0,1) based on the probabilities of the source alphabet *A*



Since the first symbol of the sequence is a_1, the tag lies in the interval [0, 0.7), which will, then, get rescaled based on the same probability as in the source alphabet A, yielding the subintervals [0, 0.49) for a_1, [0.49, 0.532) for a_2 and [0.532, 0.7) for a_3. The second symbol is a_2, which lies in the subinterval [0.49, 0.532), and, by using the same method, the interval gets rescaled based on the same probability as in the source alphabet *A*, yielding, in this case, the further subintervals [0.49, 0.5194) for a_1, [0.5194, 0.52192) for a_2 and [0.52192, 0.532) for a_3. The third and final symbol is a_3, as shown in Figure 2.2. Each symbol can be located by using the subinterval generated by the symbol probability, while the interval value for each symbol is disjoint from all other intervals. Therefore, any value within the last interval [0.52192 and 0.532) can be used as a tag. Some techniques use the lower limit and others the midpoint of the interval, $\overline{T}_A$ = (0.52192+0.532)/2 = 0.52696.

The tag for sequence A can be found by using the following recursive algorithm:

$$l^{(n)} = l^{n-1} + \left(u^{n-1} - l^{n-1}\right)fx\,(x_n - 1) \tag{2.3}$$
$$u^{(n)} = l^{n-1} + \left(u^{n-1} - l^{n-1}\right)fx\left(x_n\right) \tag{2.4}$$

where $l^{(n)}$ is the lower limit, $u^{(n)}$ the upper limit of the tag interval, and x_n is the value of a random variable.

Since each subinterval contains the succeeding interval, the value of the subintervals decreases and the sequence gets longer, requiring, therefore, higher arbitrary precision. This will cause a problem when a system with finite-pre processes a tag that has higher precision than the system. In addition, the encoder has to encode the full message before transmitting it, as each symbol in the message is encoded from the succeeding interval. Both issues are, however, resolved by re-scaling the intervals to avoid getting smaller values and transmitting the bits that correspond to the intervals, which are regularly using the following equations:

$$\{0,\ 0.5\} \rightarrow [\,0,\ 1); \qquad E_1(x) = 2x \tag{2.5}$$
$$\{0.5,\ 1\} \rightarrow [\,0,\ 1); \qquad E_2(x) = 2(x - 0.5) \tag{2.6}$$
$$\{0.25,\ 0.75\} \rightarrow [\,0,\ 1); \quad E_3(x) = 2(x - 0.25) \tag{2.7}$$

Encoding the sequence with re-scaling is done incrementally by calculating the lower and upper limits of the interval. If the results of both the lower and upper intervals are confined within the interval [0, 0.5), the encoder will generate 0 and rascal to the full interval [0,1) by using the equation *2.5*. If, on the other hand, the results are confined within the interval [0.5, 1), the encoder will generate 1 and rascal to the full interval [0,1) by using the equation *2.6*. Otherwise, if the results are confined within the interval [0.25, 0.75) and do not satisfy equation 2.5 or 2.6, the encoder will double the tag interval, subtract it by 0.25 and, finally, rescale to the full interval [0,1) by using the equation 2.7. The encoder will record the number of equation 2.7 operations and transmit it as 0's when the next interval is confined within equation 2.6, or the encoder will transmit the number of uses as 1's when the next interval is confined within equation 2.5. The last results of $l^{(i)}$ and $u^{(i)}$ are used to generate the binary tag for the next element. The receiver will be informed by transmitting any value from the last interval, followed by the remaining bits, which are required by the word length of the system in use.

The decoding process works backwards with additional steps. It requires the word length and the code table of the message where the encoded message is located. It then initialises $l^{(n)}$ and $u^{(n)}$ and rescales using the same equations 2.5, 2.6 and 2.7 to extract the values of the encoded message. If the upper and lower values are contained in the upper half, the decoder shifts the MSB of the encoded message out of the receive buffer, moves the next bit in, and updates the tag using the equation 2.6. During the shifting process, the value of the tag will change, and it needs to be compared with the tag intervals.

If the values fall between the range of the tag intervals in the code table, it will generate the element of the message within the located interval.

If the tag value is contained between the upper and lower halves, it will be compared to the tag interval of the symbol in the source alphabet A to locate the symbol and rescale it again.

If the upper and lower values are contained in the lower half of the intervals, the same process will apply. The decoder needs to shift the MSB of the received encoded message out of the receive buffer, move the next bit in, and update the tag using the equation 2.5. During the shifting process, the value of the tag will change, and it needs to be compared with the tag intervals.

If the values fall between the tag intervals, the element of the message will be generated within the located interval.

2.4.6. Integer Arithmetic Coding

The arithmetic coding can be implemented on floating-point as discussed above. It can, however, be also implemented on integers. The integer implementation requires the word length m to be set up for both the encoder and decoder, while the generated values will be mapped into the intervals that have the size of 2^m binary words instead of the interval [0, 1). The word length must be large enough to have sufficient values to represent the upper and lower limits of all intervals. Instead of probabilities, it uses the number of occurrences of the symbols that are listed in a cumulative count (*CumCount*) table. In this way, a fractional representation of the probabilities can be avoided. The following equations 2.8, 2.9, 2.10 and 2.11 illustrate the process:

$$F_X(k) = \frac{\sum_{i=1}^{k} n_i}{Total\ Count} \qquad (2.8)$$

$$CumCount(k) = \sum_{i=1}^{k} n_i \qquad (2.9)$$

$$l^{(n)} = l^{(n-1)} + \left\lfloor \frac{u^{(n-1)} - l^{(n-1} + 1 \times CumCount(x_n - 1)}{TotalCount} \right\rfloor \qquad (2.10)$$

$$u^{(n)} = l^{(n-1)} + \left\lfloor \frac{u^{(n-1)} - l^{(n-1} + 1 \times CumCount(x_n)}{TotalCount} \right\rfloor - 1 \qquad (2.11)$$

The equations are very similar to the floating-point equations, in which in equation 2.8, n_i is defined as the number of times the symbol i occurs in the *TotalCount* $F_X(k)$ and where x_n is the symbol that needs to be encoded.

The encoding process starts with the selection of the word length m and populating the endpoint values of the intervals into the CumCount table according to equation 2.9. In equation 2.10, $l^{(n)}$ represents the lower half of the interval, and equation 2.11 represents the upper half of the interval $u^{(n)}$.

If the MSB is 1, the tag is contained in the upper half, or if the MSB is 0, the tag will be located in the lower half. When equation 2.5 and 2.6 have been mapped, the MSB will be shifted and 1 added into the integer code for $u^{(n)}$ and 0 for $l^{(n)}$. The mapping of 2.7 will be needed when the tag interval is located in the middle of the interval, that is, when the second MSB of $u^{(n)}$ is 0 and the MSB of $l^{(n)}$ is 1. In this case, the second MSB for both $l^{(n)}$ and $u^{(n)}$ will be complemented and shifted to the left, while 1 will be shifted in the LSB for $u^{(n)}$ and 0 for $l^{(n)}$. The number of times that 2.7 is used, will be recorded.

The decoding process starts when the tag t is received from the encoder by using the same *CumCount* table and the same word length m, which has been based on the word length during the initialisation process. It sets the lower limit as 0's and the upper limit as 1's. For example, if the word length is 4, then $l = (0000)_2 = 0$ and $u = (1111)_2 = 15$. The tag value is then computed using the following equation:

$$\left\lfloor \frac{(t - l + 1) \times TotalCount - 1}{u - l + 1} \right\rfloor \qquad (2.12)$$

The result will be compared to the Cumulative Count table, while the output of the symbol lies within the range of the result and, thus, updates the lower and upper limits. If the updated result satisfies equation 2.5 or 2.6, the decoding proceeds with the use of $l^{(n)}$ and $u^{(n)}$ equations and a comparison of the results with the Count table to obtain the second symbol. However, if the result satisfies equation 2.7, the MSB will be shifted out for $l^{(n)}$, $u^{(n)}$ and t, the new MSB will be complemented, and 1 shifted in the LSB for $u^{(n)}$ and 0 for $l^{(n)}$. Finally, the new LSB will be inserted for t from the tag. The decoder will stop when all the bits in t have been processed [55][56].

Arithmetic coding is becoming a very popular compression paradigm, and it is widely applied in the coding standards of pictures, such as Advanced Video Coding (H.264/AVC), High-Efficiency Video Coding (H265/HEVC), Joint Photographic Experts Group (JPEG2000) and Joint Bi-level Image Experts Group (JBIG) [57][58].

The reasons for its popularity include its effectiveness, flexibility, and ability to achieve good compression results. Moreover, it addresses the high-frequency difference between symbols, a disadvantage in Huffman coding, and encodes the entire message into a single code. There are, however, two main challenges in applying arithmetic coding. Firstly, the probability model is, in some cases, unknown, and secondly, each proceeding interval requires multiplication operations, which cause delays in information processing. Adaptive arithmetic coding [25] can solve the former problem by using a counter that calculates each symbol after encoding it and updating the cumulative count table accordingly. This means that the decoder updates the table after each decoding.

Substantial research endeavours have been undertaken to tackle the latter issue of processing delays and achieve swifter and more efficient arithmetic coding. The measures have included the use of a table coder and avoiding multiplications by restricting the range of intervals and using lookup tables that locate an approximate estimate instead of multiplications. Alternatively, compression times can be reduced by diminishing sequence lengths, the number of distinct symbols, and the symbol probabilities that influence arithmetic coding performance. Compression times also diminish as symbol probabilities approach 1 [59].

2.4.7. Asymmetric Numeral Systems

Asymmetric numeral systems are accurate entropy coding algorithms that have a similar encoding speed to Huffman coding and they can, arbitrarily, get close to optimal entropy. In this, they are similar to Arithmetic coding. Asymmetric numeral systems were introduced by Jarek Duda [60][61][62], and they include two main versions. The range Asymmetric Numeral Systems (rANS) and table variants (tANS). For tANS, a list of natural numbers can be generated in the table, which is based on the frequencies of the symbols. The table consists of a list of state numbers x, assigned based on symbol probabilities. Each state number is linked to a symbol s, ensuring proportional representation of each symbol in the table.

The state numbers represent different encoding states in the table. Each state is associated with a specific symbol. During encoding, the current state is used to determine the next symbol, and the updated state is selected based on predefined mappings in the table.

When the symbols can be selected from the state space x using the discrete integers, the state space must fit within a bit-length of $log_2(x)$. Since the probability of a symbol s can be defined, the size of s can also be defined as $log_2(1/p_s)$. Consequently, $x' = log_2(x) + log_2(1/p_s) = log_2(x/p_s)$ bits, and hence, the approximation of x' enables the selection of any s that defines the symbol. The table is created by processing the symbols in order based on their probabilities, starting from the left. The highest probability symbol will be inserted in the first column, the second-highest probability symbol will be inserted into the next column and so on. The values for each state x that represent the symbol can be determined by dividing the state number by the probability of each symbol x/p_s, while the results are rounded either up or down, depending on whether the value has been used before.

For example, if the symbols S, B, E have the probabilities $P(S) = 0.2, P(B) = 0.35, P(E) = 0.45$, and the word BEE requires encoding, the table is created by assigning symbols in decreasing probability order: E first, B next, and S last. Each symbol is assigned state values in increasing order, ensuring uniqueness (i.e., no value should occur twice). Furthermore, each state value representing a symbol must be greater than x.

The number of state values assigned to each symbol is approximately equal to the total number of states multiplied by the symbol's probability. The values representing each symbol in the table are determined by dividing the state number by the symbol's probability, rounding appropriately to ensure uniqueness.

For example, to compute the first state assignment for symbol S, the state number is divided by $P(S)$, $x_1/p(S)$, that is, $1/0.2 = 5$, and to generate the value for (1, B) the probability of B is 0.35, that is, $1/0.35 \approx 2.857$. In this case, the result is rounded either up or down, making sure that rounding the value results in a number that has not been used before. Consequently, dividing the state number by the selected value should yield a result approximately equal to the symbol's probability.

Table 2.4 shows the values generated for the symbols E, B and S which correspond to approximately the probability of the symbols [63].

Table 2.4: tANS for symbols E, B, S

State (x)	E	B	S
1	2	3	5
2	4	6	10
3	7	8	15
4	9	11	20
5	12	14	25
6	13	17	30
7	16	21	
8	18	22	
9	19	26	
10	23	28	
11	24		
12	27		
13	29		
14	31		

As the table has now been constructed, the word BEE can be encoded, and the coding equation can be defined as:

$$C(x, s) \longrightarrow x' \qquad (2.13)$$

Starting with x_1 in Table 2.4 the value representing the first letter B, is found at (1, B)= 3. The next state to be processed is x'_3 for the next symbol E, the table entry (3, E) = 7. Following the same process, the obtained value indicates x'_7 in the state column, leading to the third symbol E, which is encoded as (7,E) = 16, Thus, the values that represent the encoded word BEE are (3, 7, 16).

The decoding process works by using the table, and it is defined as:

$$D(x')\longrightarrow(x, s). \qquad (2.14)$$

Only the final value (16) is required to start the decoding process. The process commences by locating the value 16 from the table, which corresponds to (x'_7, E). Thus, the first decoded symbol is E, and the next state is 7. The state 7 corresponds to (x'_3, E), where E is decoded from x'_3, and the next state becomes 3. Looking up 3 in the table reveals that it corresponds to (x_1, B). The letter B will be decoded and the state number x_1 indicates the end of the message. The word EEB has been decoded, and it is the inverse of the encoded message BEE. The method outlined above illustrates the use of tANS in the encoding and decoding process, and this can be used in compression [63].

Compression begins by encoding values starting from the highest numerical value assigned to each symbol in Table 2.4. The value obtained from encoding the first symbol is then used to determine the corresponding state in column x. As shown in Table 2.4, the highest values representing each symbol exceed the maximum state value, which is x_{14}. For example, 30 represents S, 28 represents B, and 31 represents E. Since the first encoded value always exceeds the maximum state value, the highest available value in the state column x, which corresponds to the next symbol to be encoded, is chosen.

The value obtained from the first symbol will be converted to a binary representation. Since this value exceeds the maximum state value for the next symbol, the binary representation is right-shifted, outputting one bit at a time. After each shift, the resulting value is checked to determine if it still exceeds x. This process continues until the value becomes smaller than or equal to x. Once this condition is met, the adjusted binary value serves as an index for retrieving the next state and symbol, allowing the encoding process to proceed. For example, in encoding the word BEES, the first letter B corresponds to the maximum index value 28 which is represented in binary as $\{11100\}$. The next symbol is E, and the previous state value (28) is used to locate its corresponding state in column x. However, Since state x_{28}. is not available in the table, the highest available state containing E is selected specifically, (x_{14}, E). Since 28 exceeds 14, the state value is right-shifted by one bit, outputting {**0**}, until it becomes equal to or less than 14. After shifting the first bit, the value outcomes {11100} = 14 which matches x_{14}. This results in the next state symbol pair: (x_{14}, E) = 31, with 31 represented in binary as {11111}.

Proceeding, the third symbol *E* as (x'_{31}, E). However, since 31 is not present in the current state column, the highest state column containing the next symbol E, which is x'_{14}, is selected. Given that 31 exceeds 14, the state is shifted right by one bit to the output stream from the value 31, {1111}→{**1**}. Upon shifting the initial bit, the result is {1111} = 15, still exceeding 14. Consequently, another shift is performed, {111}→{**1**}, ultimately yielding (x'_7, E) = 16 represented in binary {10000}.

Next, the fourth symbol *S* is processed, resulting in (x_{16}, S). Since 16 is absent in the state column, the highest state column containing the subsequent symbol *S*, which is x_6, is selected. Since 16 exceeds 6, the state is shifted right one bit from {10000} to {1000}→{**0**}. Subsequent shifting leads to {1000} = 8, which still exceeds 6. Hence, another right shift is performed {100}→{**0**}, ultimately resulting in (x'_4, S) = 20 represented in binary as {10100}. Conclusively, the word BEES is fully encoded as {10100}, with the shifting output {**01100**}, resulting in a total bit string of 10 bits. It's noteworthy that the original length of the binary number is consistently 5 bits before each bit shift.

Decompression works by inverting the compression process, giving the symbol frequencies, the shift output bit string {**01100**}, and the encoded string {10100}. The table can be regenerated by using the given symbol frequency and the encoded string {10100} = 20, which will locate the corresponding symbol from Table 2.4, that is (4, S). The symbol *S* and the number x'_4 are output, this gives the binary value {100}, which is 3 bits. Next, 2 more bits are appended from the shifting output string, starting from the least significant bit, this gives {10000} = 16. The remaining bits from the shifting string are {**011**}. The value 16 corresponds to (x'_7, E), in the index table, so *E* is output, along with the state x'_7. This gives the binary value {111}, which is 3 bits. Then, 2 more bits are appended from the shifting string, resulting in {11111}=31. The remaining bit from the shifting string is {**0**}.the value 31 in the index column corresponds to (x'_{14}, E). *E* is output, and the state number 14 from the state column is {1110} in binary, which is 4 bits. Finally, the last bit from the shifting string is appended, resulting in {1110}=28, which corresponds to (x'_{10}, B). B is output, and since there are no more bits remaining in the shifting string, the decoding process ends. The decoded message is SEEB, which is the inverse of the word BEES [63].

Asymmetric Numeral Systems (ANS) are increasingly attracting attention. For instance, they have been used by companies like Facebook [65][66], in hardware architectures [67], and in information security applications [68]. Since the introduction of Asymmetric Numeral Systems, significant research has been conducted to explore their potential applications. Yokoo and Shimizu [69] analyzed the range Asymmetric Numeral System (rANS) and argued that directly making the length of state intervals proportional to symbol probability does not lead to optimal performance. Instead, they proposed an improved source approximation method for rANS to enhance compression efficiency. They also introduced a formula for source approximation that reduces compression loss caused by the asymmetry in the range variant.

In a separate study, Yokoo and Dubé [70] presented a method for constructing nearly optimal symbol distributions for the stream variant of tANS by leveraging sorting. This approach uses the stationary probabilities of symbols in the stream variant to achieve fast and efficient average code lengths.

The range variant of Asymmetric Numeral Systems outperforms Arithmetic Coding in implementation [62]. While Arithmetic Coding requires calculating multiple intervals, each defined by two variables (upper and lower limits), rANS manages a single discrete value at a time, corresponding to the symbol being encoded. This makes ANS more similar in speed to Huffman coding [62], while maintaining the efficiency of Arithmetic Coding. These advancements highlight the current state of research in entropy-based lossless data compression [44].

Lossless data compression techniques are essential for reducing storage space and transmission bandwidth required for digital data. Various coding methods have been developed to achieve this goal, each with its own advantages and drawbacks [64]. A promising approach lies in the development of an adaptive numeral system that dynamically adjusts to data characteristics without relying on prior knowledge of symbol frequencies, remaining resilient to highly variable data sets [60]. This differs from Huffman coding [15], which requires prior knowledge of symbol frequencies to function effectively. In contrast to Run-Length Encoding (RLE) [16], which is more limited in scope, the proposed approach aims to be applicable across all data types. It also avoids the computational complexity often associated with Arithmetic Coding [54] and does not demand the significant resources required by Asymmetric Numeral Systems [62]. Implementing such a system would involve continuously analyzing data streams and adapting encoding schemes based on observed patterns. This adaptive mechanism could enable efficient sequential data compression across diverse data types [79].

2.5. Summary

This chapter reviewed the key theoretical concepts and established compression techniques that have shaped the field of lossless data compression. It discussed the role of Information Theory in defining compression limits and examined a range of classical and modern algorithms, highlighting their strengths and limitations. The insights gained from this chapter provide a critical foundation for exploring alternative compression methods, particularly those based on adaptive numeral systems, which will be addressed in later chapters.

Chapter 3

3.1. Introduction

This chapter explores the Adaptive Numeral System (ANS) and its extended variant, the Improved Adaptive Numeral System (IANS), as potential numeral systems for data compression. Given that digital data is fundamentally represented using numeral systems, ANS and IANS were investigated as alternative approaches to encoding binary streams. The chapter details their encoding and decoding processes, examining how they structure data and whether they offer advantages in terms of efficiency, complexity, and memory usage. By analysing how these numeral systems manipulate binary sequences, this chapter aims to assess their suitability for data compression and their broader implications for digital storage and transmission.

3.2. Symbol Representation Variations in Binary Encoding Processes

The primary aim of this research has been to devise a numeral system capable of calculating a dataset in a manner conducive to developing a compression method. This method aims to reduce the outcomes of these calculations, employing them, in turn, to encode the subsequent dataset and compress it iteratively. Consequently, this approach enables the initial dataset to undergo compression multiple times, provided that new datasets are integrated into the compression chain. Moreover, as additional datasets are appended to the chain, further compression of the entire dataset becomes achievable.

This process unfolds as follows: the initial dataset undergoes compression based on the number of datasets in the chain. Subsequently, the second dataset is compressed by the number of datasets that succeed it in the chain. This pattern continues iteratively.

In what follows, the investigation of a number of methods that can represent a binary stream by applying to it a novel adaptive numeral system that has the potential to compress data, while at the same time taking into account the complexity, memory size and speed of the encoding and decoding processes.

All the digital data is based on the idea of using a numeral system to send a unique number to represent something like an image, sound, letters of the alphabet, or even a three-dimensional (3D) film. The binary system is one of the highly efficient numeral systems for symbolising a large number, especially, when almost all digital communication channels and device processes use this system [71]. This makes it the main numeral system in the digital era. The binary numeral system is the main communication technique when a digital device is actively sending and receiving electrical pulses that represent a binary stream that has been encoded to represent some object. It requires, however, the sending of a large number of bits to represent, for example, an image or a book.

As discussed in Chapter 2, various techniques and lossless compression algorithms have been designed and implemented to reduce the amount of data [6]. Also, techniques of achieving a higher compression ratio are constantly being researched as it is becoming increasingly important. The techniques are, however, not limited to the use of the binary system for applying compression, as only the results need to be in binary form. The fundamental idea of a code can, therefore, be defined as the encoding of one set of symbols with another set of symbols. The code can be designed to have a fixed length, known as *block codes*, or variable length, known as *variable-size codes*. It can also be designed as static or adaptive code, known as *dynamic code* [61].

In the field of lossless data compression, a great deal of research has concentrated on removing redundancy when encoding information [7]. This has been done by using probabilities to find the most frequently occurring symbols to be encoded with fewer bits and the least frequently occurring symbols with longer bits. However, the natural form of information consists of binary symbols in which each bit can be one of two integers 1 or 0, any x bit of information will have 2^x combinations, and every n in any set of data will have one of the 2^x combinations [72]. In consequence, if there is a method to identify the order within a combination from a data set, the question can be asked if it will have the potential to increase the redundancy in the information.

Since the binary system consists of two symbols 1 or 0, the representation of these symbols can take different forms during the encoding process. These can include forms, such as higher or lower, on or off, greater or less and even x and y. If a probability model is applied to the binary system, it can consist of only 1 number, either the probability of 1 or 0. The probability of the second symbol is simply one minus the probability of the first symbol, and it will, thus, have a more accurate representation of the probability compared with other models that represent different contexts [73]. In what follows, a novel recursive method of calculating an unbound bit string (the most significant bit could be 1 or 0) will be outlined.

3.3. Adaptive Numeral System (ANS)

The Adaptive Numeral System (ANS) is a novel recursive method of calculating an unbound bit string where the most significant bit could be 1 or 0. A list of non-negative integers will be generated, in which the last two integers of the calculated results will contain the given order in the data string and their use can be used to regenerate the original data set. The purpose of this is to generate an x which can be compared with x_{-1} to identify the change in the given data stream. For example, if $x > x_{-1}$, it indicates a repeated bit in the data stream, and if $x > x_{-1}$, a switch is identified from 1 to 0 or vice-versa. The encoding and decoding processes will be explained in the next two subsections.

3.3.1. Encoding Process using the ANS

The process of encoding involves the inspection of a string of data, starting either from the MSB or the least significant bit (LSB) and then encoding 1 when the first bit is either 1 or 0. If the second bit is the same as the previous bit, the last two generated integers will be added. There may also be a switch from 1 to 0 or vice-versa, In which case, the last two integers will be subtracted and the results added to the previous integer. The method can be defined more specifically as:

$$D_{i<2} = 1 \quad \text{when } s_1 = 1 \text{ or } s_1 = 0 \qquad (3.1)$$

$$D_{i>1} = D_{i-1} + D_{i-2} \quad \text{when } s_i = s_{i-1} \qquad (3.2)$$

$$D_{i>1} = D_{i-1} - D_{i-2} \quad \text{when } s_i \neq s_{i-1} \qquad (3.3)$$

$$D_i = D_{i-1} + D_{i-2}$$

In the above specification, D is the encoder and s is the binary bit of a data source. An example will illustrate this further. Consider the binary string $S = \{00111\}$, the encoder D can start from either MSB or LSB. Assuming the encoder starts from the MSB, it reads the first-bit s_1, and initialises the encoder D_0 and D_1 to 1, where D_1 is the first encoded bit. Once the first bit is in place, the encoder moves to the second bit and starts encoding the given binary stream. The second bit is equal to the first bit that satisfies the equation 3.2, $D_2 = 1 + 1 = 2$, whereas the third bit $s_3 = 0$ is not equal to the second bit, while satisfying the equation 3.3, $D_3=2-1=1$, $D_4=1+2=3$. When a change in the bits has been detected and equation 3.3 has been applied, equation 3.2 will be used after equation 3.3 before moving to the next bit.

The reason is that after using equation 3.3, it may result in $D_i < D_{i-1}$. If the next bit changes again in the binary stream, equation 3.2 will be applied, and it may result in a negative integer, which will lead to the loss of the comparative representation of the binary set when further bits are calculated. For the fourth, fifth and sixth bits, from s_4 to s_6 = 1, they satisfy equation 3.2, and, thus, D_5 = 1+3 =4, D_6 = 3+4 =7, and D_6 = 4+7 =11. The last two integers are 7 and 11 respectively. Table 3.1 illustrates this example, but it is important to note that when equation 3.2 is used, it will result in $D_i \geq D_{i-1}$ while when equation 3.3 is used, it will result in $D_i < D_{i-1}$.

Table 3.1: Encoding the binary stream *S*

	MSB						LSB
Source (*S*)	0	0	X	X	1	1	1
Encoder	1	1+1	2-1	2+1	1+3	3+4	4+7
Results (D_i)	1	2	1	3	4	7	11

Now the binary string S has been encoded with the following integers (D_6 = 7, D_7 = 11), which can be used to regenerate the original binary stream from the LSB (as the encoder started with the MSB) by comparing them with each other. If the last value is bigger or equal to the previous integer, it indicates a repeated bit, and $s_i = s_{i-1}$ should be decoded. Otherwise if $D_i < D_{i-1}$, it indicates a switch between 0 and 1 or 1 and 0, and the previous bit, if any, needs to be discarded.

One crucial observation emerges from the aforementioned analysis. The final two integers generated during decoding can potentially represent either bit, without any discernible indicator for the decoder to determine its initial bit. The starting bit could have been either 1 or 0, rendering it impossible for the decoder to ascertain the initial value of the LSB. For instance, if the decoder is designed to start decoding with 1 for the example in Table 3.1, it would decode the original bit stream. Conversely, if the decoder is designed to commence with 0, it would result in the decoded string being the inverse of the original bit stream. To circumvent this issue, subtracting the last two generated integers could be employed to ensure $D_i<D_{i-1}$, signifying the LSB ends with 0, while maintaining the last integer of D_i unchanged when the LSB ends with 1, as $D_i<D_{i-1}$. In the aforementioned example, given its termination with 1, no alteration is necessary. Moreover, since $D_7 > D_6$, it indicates the starting bit is 1. The next section will cover the decoding process.

3.3.2. Decoding Process using the ANS

The decoding process works backwards. Since the encoding started at the MSB, the decoding will start from LSB, and the decoder can be defined with the following equations:

$$s_i = 1, D_{i-2} = D_i - D_{i-1} \quad \text{when } D_i \geq D_{i-1} \tag{3.4}$$

$$s_i = 0, s_i \notin S, s_{i-1} \notin S, D_{i-2} = D_{i-1} - D_i \quad \text{when } D_i < D_{i-1} \tag{3.5}$$

Assuming the last two integers were derived from the preceding example, decoding the integers (D_i = 11, D_{i-1} = 7) requires initial verification of whether $D_i \geq D_{i-1}$ or $D_i < D_{i-1}$. In this instance, 11 is greater than 7, indicating adherence to the decoder equation (3.4) for regenerating the binary stream from the LSB. Consequently, s_7 = 1, and D_5 is calculated as D_7 - D_6, resulting in D_5 = 11 - 7 = 4.

Table 3.1(a): Decoding the first bit for ($D_i = 11, D_{i-1} = 7$)

	MSB		LSB
Binary String (S)		1	1
Decoder	11-7		
Results (D)	4	7	11

The next integer is 7 and, as it is greater than 4, equation (3.4) will be applied with $s_6 = 1$, $D_4 = D_6 - D_5 = 7 - 4 = 3$.

Table 3.1(b): Decoding the second bit for ($D_i = 11, D_{i-1} = 7$)

	MSB			LSB
Binary String (*S*)			1	1
Decoder	7-4			
Results (*D*)	3	4	7	11

The next integer is 4, and, as it is greater than 3, equation (3.4) will be applied with $s_5 = 1$ and $D_3 = D_5 - D_4 = 4 - 3 = 1$.

Table 3.1(c): Decoding the third bit for ($D_i = 11, D_{i-1} = 7$)

	MSB				LSB
Binary String (*S*)			1	1	1
Decoder	4-3				
Results (*D*)	1	3	4	7	11

The next integer is 3 and, as it is greater than 1, equation (3.4) will be applied with $s_4 = 1$ and $D_2 = D_4 - D_3 = 3 - 1 = 2$.

Table 3.1(d): Decoding the fourth bit for ($D_i = 11, D_{i-1} = 7$)

	MSB					LSB
Binary String (*S*)			1	1	1	1
Decoder	3-1					
Results (*D*)	2	1	3	4	7	11

The next integer is 1 and, as it is less than 2, equation (3.5) will be applied with $s_3 = 0$ and $D_1 = D_2 - D_3 = 2 - 1 = 1$, while the last two calculated bits will be removed.

Table 3.1(e): Decoding the fourth bit for ($D_i = 11, D_{i-1} = 7$)

	MSB						LSB
Binary String (*S*)			X	X	1	1	1
Decoder	2-1						
Results (*D*)	1	2	1	3	4	7	11

The next integer is 2 and, as it is greater than 1, equation (3.4) will be applied with $s_2 = 0$ and $D_0 = D_3 - D_2 = 2 - 1 = 1$.

Table 3.1(f): Decoding the fourth bit for ($D_i = 11, D_{i-1} = 7$)

	MSB							LSB
Binary String (S)			0	X	X	1	1	1
Decoder	2-1							
Results (D)	1	1	2	1	3	4	7	11

The next integer is 1 and, as it is equal to 1, equation (3.4) will be applied with $s_1 = 0$ and $D_{-1} = D_3 - D_2 = 1 - 1 = 1$.

Table 3.2: Decoding ($D_i = 11, D_{i-1} = 7$)

	MSB								LSB
Binary String (S)			0	0	X	X	1	1	1
Decoder	1-1								
Results (D)	0	1	1	2	1	3	4	7	11

The final bit of the string has been regenerated from the two integers 7 and 11, resulting in the encoded message 00XX111, where X is ignored as defined in the decoder, in equation (3.5).

3.3.3. Analysis and observations of the ANS

The method outlined above has been tested on the complete set of 1-byte combinations, revealing its unique decodability when dataset combinations share equal length. This distinctiveness is notably apparent in the combination and sequence of the last two integers generated, distinguishing them from others. Illustrated in Table 3.3 is the encoding process for the entire spectrum of 4-bit binary string combinations, with grey numbers denoting transitions between 1 and 0, or vice versa.

Table 3.3: Encoding the combinations of a 4-bit string

												Encoder Results	
Bit string	MSB										LSB	D_{i-1}	D_i
0000	1	2	3	5	2							5	2
0001	1	2	3	1	4	5						4	5
0010	1	2	1	3	4	1	5	6	1			6	1
0011	1	2	1	3	4	7						4	7
0100	1	0	1	1	0	1	1	2	1			2	1
0101	1	0	1	1	0	1	1	0	1	1		1	1
0110	1	0	1	1	2	1	3	4	1			4	1
0111	1	0	1	1	2	3						2	3
1000	1	0	1	1	2	3	1					3	1
1001	1	0	1	1	2	1	3	4				3	4
1010	1	0	1	1	0	1	1	0	1	1	0	1	0
1011	1	0	1	1	0	1	1	2				1	2
1100	1	2	1	3	4	7	3					7	3
1101	1	2	1	3	4	1	5	6				5	6
1110	1	2	3	1	4	5	1					5	1
1111	1	2	3	5								3	5

The final two integers of a 4-bit binary string will generate a range of values from 0 to 7 that can be converted back to binary symbols for transmission. When, however, the decoder regenerates the binary stream, there is no flag to determine when the decoder halts. For example, if the binary stream {1010} is encoded, the encoder will generate the final two integers as 1 and 1, respectively, as shown in Table 3.3. When decoding these integers using the decoder equations, the decoder will regenerate an infinite string of symbols of 011's, resulting in an infinite bit stream of 1's and 0's. Still, there are different ways of resolving this problem, depending on how the method is used:

i. By transmitting the total number of bits of the binary source to determine when the decoder will halt.

ii. Applying the method on a fixed bit length. For example, if the method is applied on a 4 bits string of a message length that consists of 8 bits of 1's, it will generate two pairs of integers 3 and 5 twice respectively. The decoder will regenerate 4 bits of information and halt for each pair. This is illustrated in Figure 3.1.

Figure 3.1: Solution II, halting after generating 4 bits of information

Bit String	Halt	1	1	1	1	Halt	1	1	1	1
Generated Values	Halt	1	2	3	5	Halt	1	2	3	5

This solution can also be applied to the total message length to generate one pair of values. Using this method will avoid sending any further information to the decoder.

iii. By initialising the first two integers of the encoder as 1 and 2 will cause an increase in the final two integers that are generated. Using the same example of the binary stream 1010, the final two integers will increase from 1 and 1 to 9 and 1, respectively, as shown in Table 3.4. This will allow the decoding of the integers until it has generated 2 and then 1, respectively, which will flag the halt of the decoding process. Using this solution will avoid the application of the encoder equation (3.5) because the second value of the decoder will be initialised to be greater than the first value $D_2 \geq D_1$.

Table 3.4: Encoding the binary stream 1010 by initialising the encoder to 1 and 2

	Initialisation		MSB										LSB
Source			1			0			1			0	
Encoder			2+1	3-2	1+4	4+1	5-4	1+5	6+1	7-6	1+7	8+1	9-8
Results	1	2	3	1	4	5	1	6	7	1	8	9	1

The method, which has been discussed above, has many useful properties. It was designed to be applicable to binary symbols, and in turn, it would be adaptive to various types of code like fixed and variable-length codes. Using a fixed-length code with the range of the 4-bit string between 0 and 7, each integer can be encoded to have the weight of 3 bits by simply converting each value that has been generated back to the binary symbols. A variable-length code can be achieved by combining the last two integers and converting them to binary symbols. For example, by giving a bit string 0000, the method will generate the last two integers as 5 and 2, while the two integers will be merged together to form the decimal value of 52, and the conversion of the number back to a binary will be 110100. Table 3.5 shows the variable-length code for the combined ANS values.

Table 3.5: Variable-length code of the integers from Table 3.3

Bit String	Generated integers	Binary
0000	52	110100
0001	45	101101
0010	61	111101
0011	47	101111
0100	21	10101
0101	11	1011
0110	41	101001
0111	23	10111
1000	31	11111
1001	34	100010
1010	10	1010
1011	12	1100
1100	73	1001001
1101	56	111000
1110	51	110011
1111	35	100011

The code could also be variable-length static, and the integers can be found by either calculating the probabilities of the given binary source, that is $1 = 54$ and $0 = 34$, P_1 = 0.6136 and P_0 = 1 - 0.6136 = 0.3864, or by generating the integers from the 4-bit string, as shown in Table 3.3. These would have the following probabilities: (0) = 0.031, (1) = 0.281, (2) = 0.125, (3) = 0.156, (4) = 0.125, (5) = 0.156, (6) = 0.063, (7) = 0.063, and the last step would be to assign the value with the highest probability to the shortest codeword and the values with the least probabilities to the longest ones.

The approach of generating two integers from a given binary source allows the information of the given source to be contained within the difference between the two integers and not the actual integers. In consequence, there is a potential for the generated integers to be reduced and an additional data set can be calculated, starting from the reduced values. This, in turn, compresses the data by generating lower integer values as long as the original values can be retrieved. The algorithm of this method shares one property with arithmetic coding: it is possible to encode an entire file as a sequence of binary symbols into dual decimal numbers [25]. Compression can be applied in an interval without affecting the encoding process: for example, if a compression method x is created to reduce the generated integers and if two symbols H and I have the following codewords $H = \{0111\}$ and $I = \{1010\}$, the encoder will generate 2 and 3 respectively for "H", as shown in Table 3.6.

Table 3.6: Encoding *"H"* using ANS

							Encoder integers	
	MSB					LSB	D_{i-1}	D_i
Bit string "H"	0			1	1	1		
Generated Values	1	0	1	1	2	3	2	3

If the method x reduces each value by 2, then this will result in $H = 0,$ 1 respectively. Further mapping can be applied to encode "I" using the reduced integers of "H" as shown in Table 3.7.

Table 3.7: Encoding *"I"* from the compressed results of *"H"* using ANS

														Encoder Results	
	Results of *"H"* after reduction		MSB										LSB	D_{i-1}	D_i
Bit string B			1			0			1			0			
Generated Values for *"I"* using the results from *A= 0, 1*	0	1	1	0	1	1	0	1	1	0	1	1	0	1	0

This will increase the amount of data input and reduce the values of the encoded integers. Furthermore, the ANS method can be applied to the output of many compression algorithms, such as Huffman [15], BWT [40], RLE [35], and arithmetic coding [25].

The ANS encodes binary symbols efficiently by using two operations for repeated bits and four operations for bit changes, reducing complexity and simplifying both encoding and decoding. It is computable, robust, and fast, without requiring prior knowledge of the source distribution. It has one drawback though, which needs to be avoided. The method is not reliable if even a single error occurs in the encoded integers, as this will result in generating the wrong data stream when decoding. One of its advantages is, however, that it can be applied to any compressed data set. Low-value integers are generated when multiple changes occur in the data string, but the encoded values increase rapidly when the value of the same bits occurs repeatedly in the data string. It can be used to encode blocks of data or to encode the full data stream if appropriate compression can be applied during the encoding process. To encode a 4-bit string as a fixed-length code will require the last two integers, that is 6 bits of information, which is 50% more than the original 4 bits string. Hence the reduction of integers is essential during the encoding process. Promising results are shown in Table 3.7 where hypothetical reduction has been applied, generating the values 1 and 0, respectively, representing 8 bits of information.

The Fibonacci sequence has been incorporated into this study within Section 2.4.4. This inclusion is based on the observation that the adaptive numeral system exhibits similarities to the Fibonacci sequence, specifically in cases where a run of 0's or 1's occurs without any alternation. Notably, the values of the last two integers align with Fibonacci numbers. For example, in Table 3.3, the method generated 3 and 5, respectively for the binary value 1111. These values are equivalent to Fibonacci numbers 3 and 4, respectively. If the next binary number is 1, the method will generate 8 for the final value, which is the 5th value of the Fibonacci sequence. However, if a change occurs from 1 to 0 in the ANS, the generated values will no longer be equal to Fibonacci numbers.

The generated integers, which represent the binary numbers, form a unique pair of integers. There are, however, no restrictions to the size of the binary string the method can encode and the values it can take. The binary representation of some of these values can be infinitely long, but in that case, the method may not be efficient unless a method of reduction is introduced. However, it remains uncertain whether a compression method can be developed by leveraging the unique properties of ANS, which generates two integers from a given binary source, allowing the information to be represented within the difference between the two integers rather than the integers themselves. This concept will be further explored in Chapters 4 and 5.

3.4. Improved Adaptive Numeral System (IANS)

The ANS generates values ranging from 0 to 7 to encode 4 bits of information. It becomes apparent that reducing the outcome values of the ANS is crucial to enhancing its adaptability and efficiency. To address this, additional studies have been conducted with the goal of reducing outcome values and simplifying the encoding and decoding process of the ANS. This research has resulted in the enhancement of the ANS, culminating in a refined process termed the Improved Adaptive Numeral Systems (IANS). The IANS has the potential to reduce the results generated with the ANS. This is done by changing the initialisation of the ANS and the calculation steps to generate the two values D_i and D_{i-1}. Similar to the ANS, but with better results, the IANS will still generate a list of non-negative integers, in which the last two integers of the calculated results will contain the order in the data string. Moreover, these last two integers can be used to regenerate the original data set. The IANS will generate two natural numbers, A and B. A will represent the 1's while B will represent the 0's from the data string, and these representations will be used throughout this thesis. However, there are no restrictions on representing the 0's with A and 1's with B. The IANS reduces the generated results by adding repeatedly A to B if a reoccurring binary string (1) has been detected and vice versa, the results will be repeated, adding B to A if a reoccurring binary string (0) has been detected. The same operation will be applied if there is a change from 0 to 1: A will be updated by adding B to A, while B will remain unchanged. However, if there is a change from 1 to 0, B will be updated by adding A to B, and A will remain unchanged.

3.4.1. Encoding Process using the IANS

The encoding process with IANS is similar to that with the ANS with a few modifications. It involves the inspection of a string of data S, starting either from the MSB or the least significant bit (LSB). The encoding process starts by initialising two variables $A = 0$ and $B = 1$, where A will represent the 1's and B will represent the 0's. It then reads the first bit s_1 from string S, and if the bit is 1, then A will be updated by adding B to A. However, when the first bit from string S is 0, B will be updated by adding A to B. If there is a switch from 1 to 0, or vice-versa in string S, the same process will be applied. The method can be illustrated with the following equations:

$$A_0 = 0,\ B_0 = 1 \tag{3.6}$$

$$D_i = D_{i-1} + \sum_{s_i = s_{i-1}}^{s_i \neq s_{i-1}} D_c \tag{3.7}$$

When $s_i = 0$ *then* $D_i = B_i$ and $D_{i-1} = A_{i-1}$ and $D_c = A$. When $s_i = 1$ then $D_i = A_i$ and $D_{i-1} = B_{i-1}$ and $D_c = B$. Here, s_i represents the binary state, and D_c denotes the outcome of the most recent change detected in the binary sequence. Equation (3.7) can be simplified as follows:

$$\begin{aligned} B_i &= A_{i-1} + B_{i-1} \qquad \text{when } s_i = 0 \\ A_i &= A_{i-1} \end{aligned} \tag{3.8}$$

$$A_i = B_{i-1} + A_{i-1} \quad \text{when } s_i = 1 \qquad (3.9)$$
$$B_i = B_{i-1}$$

Both A and B encode the information from string S and generate a string of natural numbers, while the last number of A and B will hold the information of string S. For example, if the binary string $S = \{0110\}$, the encoder can start from either MSB or LSB. If the encoder starts from the MSB, this initialises the encoder $A_0 = 0$ and $B_0 = 1$. The encoder starts constructing the given binary stream and reads the first-bit s_1, which is 0 that satisfies the equation (3.8), then B will get updated to $B_1 = A_0 + B_0 = 1$, and after that, B_0 is shifted, resulting in $B_1 = B_0$. Once the first bit is in place, the process moves to the second bit. The second bit equals 1, $s_2 = 1$, which satisfies equation (3.9). In this case, A will get updated to $A_2 = B_1 + A_1 = 1$, and B_1 will be shifted, resulting in $B_2 = B_1$. Moving to the third bit, it equals 1, $s_3 = 1$, and again, that satisfies equation (3.9). Hence A will get updated to $A_3 = B_2 + A_2 = 2$, and B_1 will be shifted, resulting in $B_3 = B_2$. The fourth and final bit is 0, that is, $s_4 = 0$, and, therefore, B will get updated as $B_4 = A_3 + B_3 = 3$. A_3 will be shifted, resulting in $A_4 = A_3$. The binary string S is now encoded with $A = 2$ and $B = 3$. Table 3.8 illustrates this example.

Table 3.8: The IANS - Encoding the binary string *S*

Initialisation	0	1	1	0	Results
A = 0	0	1+0=1	1+1=2	2	A = 2
B = 1	0+1=1	1	1	2+1=3	B = 3

3.4.2. Decoding Process using the IANS

The results of the IANS encoding can be used to regenerate the original binary stream from the LSB (as the encoder started with the MSB) by comparing A and B with each other. If $A \geq B$, the decoder will generate 1, and B will be subtracted from A. However, if $A < B$, then the decoder will generate 0, and A will be subtracted from B. The decoder can be defined as:

$$s_i = 1, A_{i-1} = A_i - B_i, B_{i-1} = B_i \quad \text{when } A \geq B \qquad (3.10)$$

$$s_i = 0, B_{i-1} = B_i - A_i, A_{i-1} = A_i \quad \text{when } A < B \qquad (3.11)$$

Using the example from the encoding process presented in Table 3.8, the decoding process starts with comparing A and B. The results obtained are $A = 2$ and $B = 3$, that is, B is greater than A, and, therefore, equation (3.11) will be used to regenerate the binary stream from LSB, that is, $s_4 = 0$, $B_3 = 3 - 2 = 1$. $A_3 = A_4 = 2$.

Table 3.9(a): The IANS - Decoding s_4 of the binary string S

	MSB			LSB
Binary String S	s_1	s_2	s_3	$s_4 = 0$
A			2	2
B			3-2=1	3

The first step generated the fourth bit $s_4 = 0$ and resulted in $A = 2$ and $B = 1$, consequently, the same method will be applied to the next step in comparing A and B. The comparison shows that A is greater than B, thus satisfying equation (3.10) and resulting in $s_3 = 1$, $A_2 = 2 - 1 = 1$, $B_2 = B_3 = 1$.

Table 3.9(b): The IANS - Decoding s_3 of the binary string S

	MSB			LSB
Binary String S	s_1	s_2	$s_3 = 1$	$s_4 = 0$
A		2-1=1	2	2
B		1	3-2=1	3

The third bit $s_3 = 1$ has been obtained, and the values have been updated for $A = 1$ and $B = 1$. Moving to the third step, A is equal to B, hence satisfying equation (3.10), in which $s_2 = 1$, $A_1 = 1 - 1 = 0$, $B_1 = B_2 = 1$.

Table 3.9(c): The IANS - Decoding s_2 of the binary string S

	MSB			LSB
Binary String S	s_1	$s_2 = 1$	$s_3 = 1$	$s_4 = 0$
A	0	2-1=1	2	2
B	1	1	3-2=1	3

The second bit has been generated as $s_2 = 1$, while $A = 0$ and $B = 1$, and moving to the final step, it is observed that A is smaller than B, which satisfies equation (3.11), that is, $s_1 = 0$, $B_0 = 1 - 0 = 1$, $A_0 = A_1 = 0$.

Table 3.9: The IANS - Decoding the binary string *S*

	MSB			LSB
Binary String *S*	$s_1 = 0$	$s_2 = 1$	$s_3 = 1$	$s_4 = 0$
A = 0	0	2-1=1	2	2
B = 1	1	1	3-2=1	3

The string *S* has been decoded using the IANS and generated the output $\{0110\}$.

3.4.3. Analysis and observations of the IANS

The Improved Adaptive Numeral System (IANS) has been tested to encode the combinations of 16-bits of information, and the testing has shown that the combined results of *A* and *B* are uniquely decodable. Table 3.10 illustrates the process of encoding all of the combinations of a 4-bit binary string.

Table 3.10: Encoding the combinations of 4-bit string using IANS

		0	**0**	**0**	**0**	
0000	A=0					A=0
	B=1	1	1	1	1	B=1
		0	**0**	**0**	**1**	
0001	A=0				1	A=1
	B=1	1	1	1		B=1
		0	**0**	**1**	**0**	
0010	A=0			1		A=1
	B=1	1	1		2	B=2
		0	**0**	**1**	**1**	
0011	A=0			1	2	A=2
	B=1	1	1			B=1
		0	**1**	**0**	**0**	
0100	A=0		1			A=1
	B=1	1		2	3	B=3
		0	**1**	**0**	**1**	
0101	A=0		1		3	A=3
	B=1	1		2		B=2
		0	**1**	**1**	**0**	
0110	A=0		1	2		A=2
	B=1	1			3	B=3
		0	**1**	**1**	**1**	
0111	A=0		1	2	3	A=3
	B=1	1				B=1

		1	**0**	**0**	**0**	
1000	A=0	1				A=1
	B=1		2	3	4	B=4
		1	**0**	**0**	**1**	
1001	A=0	1			4	A=4
	B=1		2	3		B=3
		1	**0**	**1**	**0**	
1010	A=0	1		3		A=3
	B=1		2		5	B=5
		1	**0**	**1**	**1**	
1011	A=0	1		3	5	A=5
	B=1		2			B=2
		1	**1**	**0**	**0**	
1100	A=0	1	2			A=2
	B=1			3	5	B=5
		1	**1**	**0**	**1**	
1101	A=0	1	2		5	A=5
	B=1			3		B=3
		1	**1**	**1**	**0**	
1110	A=0	1	2	3		A=3
	B=1				4	B=4
		1	**1**	**1**	**1**	
1111	A=0	1	2	3	4	A=4
	B=1					B=1

The last encoded results of A and B generated values between 0 and 5, and these are used to decode a 4-bit binary string. The initialisation and process of the IANS reduced the generated values when compared to those of the ANS. However, the number of bits encoded is required when initialising the IANS to 0 and 1 as there is no flag to determine when to stop the decoding process. For example, if the number of the encoded bits is unknown and the decoder receives the IANS values $A = 1$ and $B = 2$, the decoder will decode the first two bits $\{10\}$. At this stage, $A = 0$ and $B = 1$, following equation (3.11) of the IANS decoding algorithm, $B > A$, that is, $B = 1 - 0 = 1$. This indicates that the next generated bit will be 0, and the results for $A$and B will remain the same ($A = 0$ and $B = 1$). At this point, the decoder will continue to generate an infinite number of 0's, while A and B will remain 0 and 1 respectively. When the number of the encoded bits is unknown, there are different ways to resolve the problem, for example by:

I. transmitting the total number of bits of the binary source to determine when the decoder will halt;

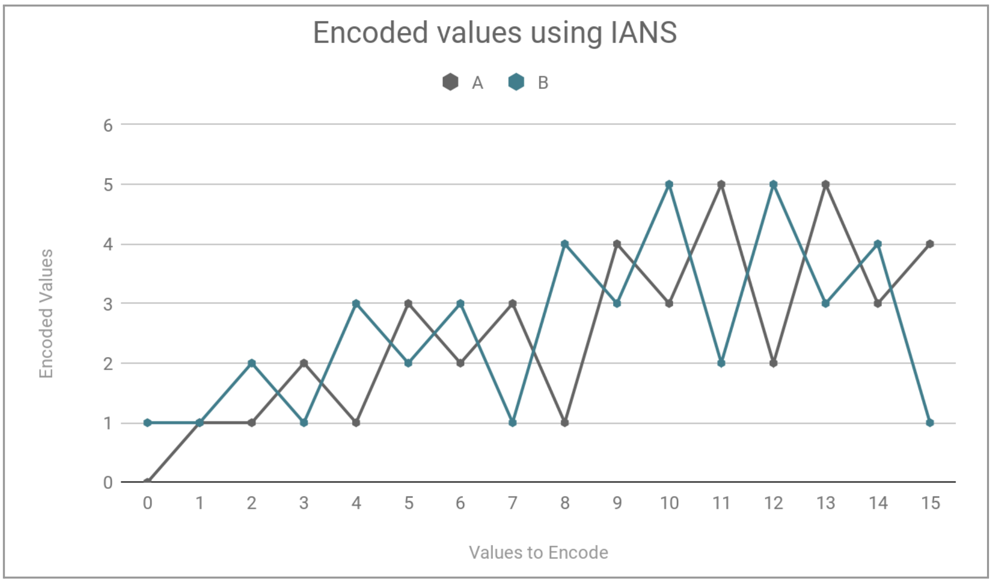
II. initialising $A = 1$ and $B = 1$, which will cause the results in A and B to increase by a maximum value of 5 and 8, respectively for a string consisting of 4 bits of information, compared with the initialisation of 0 and 1 where the maximum results of A and B are 5 and 3, respectively;

III. applying the method to a fixed bit length. For example, if the method is applied to encode a message that consists of 8 bits, the message can be divided into two segments that consist of 4 bits, and for each segment, the IANS will generate the values for A and B. This will enable the decoder to halt after generating the fourth bit of each segment.

The Improved Adaptive Numeral System has the same properties as the ANS, discussed in Section 3.3.3. However, it has turned out to be more efficient when compared to the ANS. The number of operations required to encode 1 bit of information using the IANS is two operations: the first one is checking the binary source, and the second operation is adding A to B or vice versa. The ANS requires 2 operations to encode 1 repeated bit of information, and it needs four operations to encode the change in bits from 1 to 0 or vice versa. The number of operations required for decoding in the ANS matches the number of operations needed for encoding, and the same applies to the IANS. This makes the IANS twice as fast when encoding and decoding the change in data from 1 to 0 or vice versa.

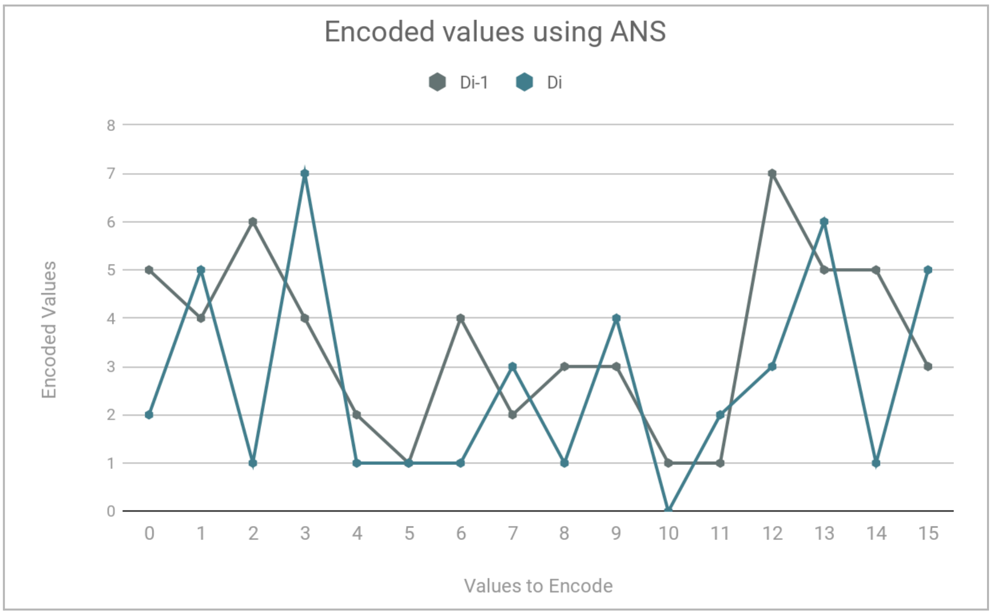
Furthermore, the generated values by the IANS when encoding the combinations of 4 bits of information are between 0 and 5. The number 0 has the minimum encoded values $A = 0$ and $B = 1$, while the numbers 10 and 13 have the maximum encoded values of $A = 3$ and $B = 5$ for the value 10 and $A = 5$ and $B = 3$ for the value 13, as shown in Figure 3.2.

Figure 3.2: IANS - Encoding 4-bits of information



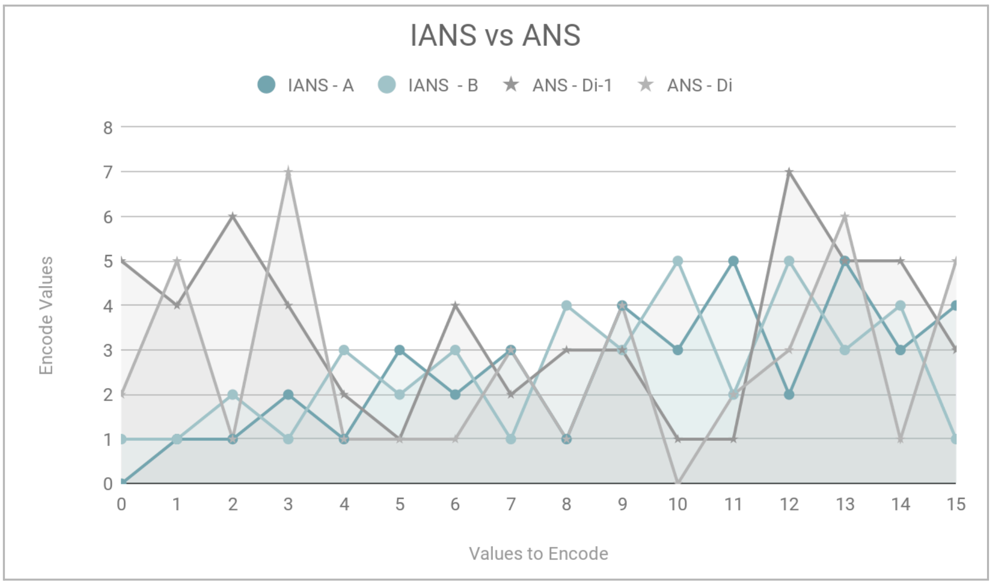
The IANS updates either A or B when encoding information, while in the decoding process, it compares the two values A and B to determine the bit to generate. This property enables the use of the generated numbers multiple times for both A and B. For example, in Figure 3.2 to encode number 2, the encoded values for $A = 1$ and $B = 2$, and to encode number 3, the IANS encodes $A = 2$ and $B = 1$. Both values 1 and 2 have been used to generate two different numbers. This is also occurring in the following encoded values: (5, 6), (4, 7), (8, 15), (9, 14), (10, 13), (11, 12). Compared with the ANS, this improves the efficiency of the IANS when generating the values for A and B. The generated values for the ANS to encode the combinations for 4 bits of information are between 0 and 7, while the number 10 has the minimum encoded values $D_{i-1} = 1$ and $D_i = 0$, and number 3 has the maximum encoded values of $D_{i-1} = 4$ and 7 as shown in Figure 3.3.

Figure 3.3: ANS - Encoding 4-bits of information



For 4-bits of information, the ANS maximum encoded values are two digits higher than the maximum encoded values for the IANS as shown in Figure 3.4. The ANS maximum encoded values for the combination for 8-bits of information are 76 and 33, whereas the maximum encoded values for IANS are 34 and 21.

Figure 3.4: ANS vs IANS - Encoding 4-bits of information



The benchmark of converting the values back to binary numbers will be the maximum values for both methods multiplied by two, as each method requires two variables to be decoded. The maximum value for 8-bits of information generated using the ANS is 76, therefore, Log_2 (76) ✕ 2 = 12.5 bits, while the IANS is 34, and, therefore, Log_2 (34) ✕ 2 = *10.18* bits, which is 2.33 bits less when using the IANS. Determining the number of bits required to encode each value for both the ANS and IANS has been considered as $\lceil Log_2$(maximum generated value)$\rceil$✕2. Figure 3.5 shows the number of bits required to encode a set of values using the ANS, the IANS, Fibonacci and direct binary systems.

Figure 3.5: ANS vs IANS vs Fibonacci vs Direct Binary - Bits required to encode a set of values

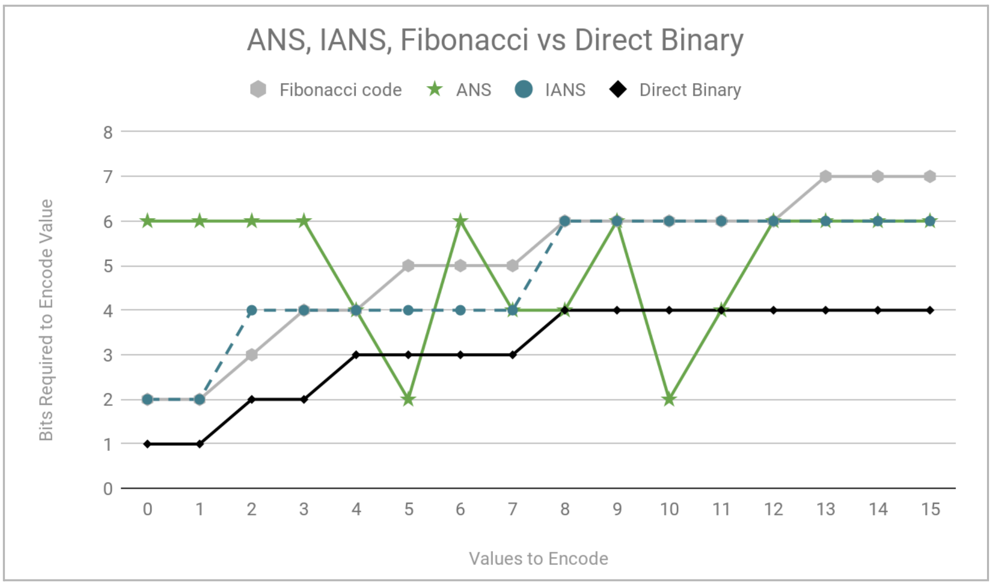


Figure 3.5 shows that the IANS compared with the ANS uses fewer bits to encode {0, 1, 2, 3, 6}, the same number of bits to encode {4, 7, 9, 12, 13, 14, 15}, and more bits to encode {5, 8, 10, 11}. Compared with Fibonacci, the IANS uses fewer bits to encode {5, 6, 7, 13, 14, 15}, the same number of bits to encode {0, 1, 3, 4, 8, 9, 10, 11, 12}, and one extra bit to encode {2}. Compared to direct binary encoding, the IANS requires an additional one to two bits. This contrasts with the ANS, as the IANS maintains symmetry with direct binary and Fibonacci encoding methods. By generating two integers from a given binary source, the IANS encapsulates the original source's information within the difference between these integers, rather than the integers themselves. This design enables the potential reduction of the generated integers, facilitating the computation of additional datasets based on the reduced values. This unique feature positions the IANS as advantageous for data compression.

3.5. Summary

This chapter examined the Adaptive Numeral System (ANS) and its derived variant, the Improved Adaptive Numeral System (IANS), as potential numeral systems for data compression. Both systems were explored in terms of their encoding and decoding processes. While ANS and IANS demonstrated structured approaches to representing binary streams, the investigation provided insights into how numeral systems can influence data representation and efficiency in encoding methods. These findings serve as a foundation for further exploration into optimising numeral systems for practical compression applications.

Chapter 4

4.1. Introduction

Building on the findings from Chapter 3, this chapter explores the application of Improved Adaptive Numeral System (IANS) in data compression. The IANS was identified as a more promising approach due to its structured encoding process and ability to initialise succeeding segments using prior results. The chapter first examines compression and decompression techniques using the IANS, assessing whether it can achieve meaningful data reduction. As part of this investigation, various modifications were made to adapt existing ANS-based compression methods to the IANS framework. These efforts led to the identification of a conditional compression method, which relies on two key properties of the IANS: segment-based encoding dependencies and differences between generated integer values. Additionally, the chapter introduces an alternative approach called The Leading Bit, a method derived from the IANS framework. This technique explores another potential way to optimise numeral system-based data encoding for compression purposes. The findings from these investigations provide further insight into the structural challenges and possibilities of applying numeral systems to data compression.

4.2. The Application of the Adaptive Numeral Systems in Data Compression

Sections 3.3 and 3.4 have demonstrated that occurrences of data represented in binary form can be calculated in segments, as assumed in the preliminary hypothesis. The second part of the hypothesis assumes that each segment can be compressed into a reasonable size where each compressed sequence can be used to calculate the next sequence. This will be discussed in the present section.

Recognising the superior efficiency of the IANS, the focus of compression studies shifted towards it. Modifications were made to ensure compatibility, leading to the discovery of a conditional compression method. This method leverages two key properties of the IANS: the ability to use encoded segment results to initialise succeeding segments, and the existence of a difference between the two generated integers used for encoding information. The following section will delve into the conditional compression method using the IANS and its implications, including the development of another compression method referred to as The Leading Bit.

4.3. The IANS

4.3.1. Compression using the IANS

The binary numeral system consists of two symbols $\{1 \text{ and } 0\}$, and the number of bits that are used to encode an x set of symbols is determined by the number of different symbols, which are available in x. Each succeeding bit doubles the number of symbols that can be encoded. Subsequently, each succeeding bit has two possibilities, that is, either $\{1\}$ or $\{0\}$ [72]. This means that, if the compression method divides the message into two segments to process each segment individually, the number of bits in both segments will have fewer combinations than the total number of bits in the entire message. This will simplify the encoding and decoding processes.

Assuming that a set of symbols x consists of one byte. The message can be divided into multiple segments, and the IANS maps the first segment and generates values for A and B. The aim is to reduce their values and use the results to initialise the second segment of the IANS, which can then be mapped. As illustrated in Section 3.3.3, Table 3.7, this process is achievable using the IANS. The conditional compression method can be applied to a segment that consists of 2 bits of information where the process requires 2 steps to complete the encoding for each segment:

1. The first segment will map the first two bits of the message using the IANS, generating the values for A and B. For two bits of information, the results of the mapping using the IANS will be between 0 and 2.
2. The results of the IANS will be subtracted from A and B, or vice versa, until reaching zero. When the last operation from the IANS mapping is applied to B, the result of the first subtraction will be placed on A, the next subtraction results will be placed on B, and so on. Overall, the operation will subtract the highest from the lowest value until 0

is reached. For two bits of information, the maximum Number of Operations (NO) to reach 0 is 3 as the values for two bits of information for {10} and {11} in IANS are $A = 1, B = 2$ and $A = 2$, $B = 1$ respectively. Post-subtractions, the result in A or B, will be either 1 or 0. The NO to reach 0 is either 1, 2 or 3.

The number of operations can then be encoded as a binary number, which can be used later for compression purposes. However, 2 bits are required to encode 1, 2, or 3. To limit the encoded NO to 1 bit, the NO can be increased from 1 to 2. Looking into this in more detail, while encoding the binary values {10} and {11} using the IANS, it can be seen that the number of subtractions to reach 0 is 3, whereas {00} requires two operations. In order to reach 0, the value {01} requires one operation, resulting in $A = 1$ and $B = 0$. Therefore, the NO of {01} can be increased by one to make the NO = 2, which is equivalent to the NO for {00}. This will give the same results of {00}, that is $A = 1$ and $B = 0$. The results can then be switched to make $A = 0$ and $B = 1$ in order to identify them from the results of the message encoded {01} or {00}. At this point, the 2 operations can be encoded with the binary symbol {0} and 3 operations as {1}. Table 4.1 illustrates this in more detail:

Table 4.1: Encoding and subtracting the combination of 2 bits using IANS

Encoding 2 bits of information using IANS				Subtracting the generated values			NO
Binary bits	0	0	Results				2
A = 0			A = 0	1			
B = 1	1	1	B = 1		0		
Binary bits	0	1					2
A = 0		1	A = 1	↶	0		
B = 1	1		B = 1	1	↷		
Binary bits	1	0					3
A = 0	1		A = 1	1		0	
B = 1		2	B = 2		1		
Binary bits	1	1					3
A = 0	1	2	A = 2		1		
B = 1			B = 1	1		0	

Increasing the NO for {01} and switching the results to make $A = 0$ and $B = 1$ makes the generated information identifiable when compared to the results generated for {00}, {10}, and {11}. The NO of {01} after modifications is equal to the NO for {00}, while the final results of the reduction for $\{00\}$ are $A = 1$ and $B = 0$. Consequently, the end results of the subtractions after modifications of $\{01\}$ are $A = 0$ and $B = 1$. Notice that the difference between $\{00\}$ and $\{01\}$ lies in the values in A and B, while the NO remains the same. In other words, when the NO is 2 and $A = 0$, it is known that $B = 1$, and the output is $\{01\}$, which means that the values of A and B can be switched back. Furthermore, when the NO is 2 and $B = 0$ and $A = 1$, the output is $\{00\}$. The same applies to $\{10\}$ and $\{11\}$. Table 4.2 summarises these properties.

Table 4.2: Encoding A and B and the NO.

Encoding A and B	
Values of A and B	Encoded A and B
$A = 0$ $B = 1$	0
$A = 1$ $B = 0$	1

Encoding the NO	
NO	Encoded NO
2	0
3	1

This will complete the encoding process of the first segment. The order of $A = 0$ and $B = 1$ or vice versa from the encoded segment can be used to initialise the next segment. The encoded NO can then be passed into the next segment and encoded further. Since the NO is either 3 or 2, this can be encoded into one bit of information. The next bit of the message can, then, be inserted next to the NO bit, making the next segment ready for processing. At this point, the mapping process using the IANS will start again, following the same steps. However, instead of encoding 2 bits from the message, the IANS will encode the already encoded NO and the next bit of the message. This process will be repeated until the message is concluded. The reduction of A and B can give either A= 0 and $B = 1$ or $A = 1$ and $B = 0$, which can be encoded using the binary system. For example, $A = 1$, $B = 0$ can be encoded to the binary symbol $\{1\}$ and $A = 0$, $B = 1$ with the binary symbol $\{0\}$. Once the last segment encodes the last bit of the message, it will be ready to be transmitted to the decoder. Only two bits are required to decode the message, the encoded NO and the encoded results of the reduction.

The following example illustrates this further. Assuming that the binary string $S = \{1001\}$. The encoder initialises $A = 0$ and $B = 1$ and can start from either MSB or LSB. In this case, the encoder starts from the MSB, reads the first two-bits s_1 and s_2, and encodes them using the IANS as shown in Table 4.3.

Table 4.3: The first segment: encoding the first two bits from the message S using IANS

Initialisation	The first bit of the message s_1	The second bit of the message s_2	Results
	1	0	
A = 0	1+0 = 1		A = 1
B = 1		1+1 = 2	B = 2

At the second step, the results will be subtracted from each other until 0 is reached, which will result in $A = 2 - 1 = 1$ then $B = 2 - 1 = 1$, and finally $A = 1 - 1 = 0$.

Table 4.4: Reducing the generated values of segment 1.

Results from segment 1	Reducing results from segment 1			Results
A = 1	2-1 = 1		1-1 = 0	A = 0
B = 2		2-1 = 1		B = 1

The final results are $A = 0$ and $B = 1$. This concludes the first segment. The results of the first segment will be used to initialise the second segment. The NO to reach 0, is 3, and that will be encoded as $\{1\}$. The encoded NO from the first segment will be inserted as the first bit of the next second segment to be encoded using the IANS. The third bit from the message S can now be inserted next to the bit that represents the NO from the previous segment. This will result in $\{10\}$ ready to be encoded using the IANS.

Table 4.5: The second segment, setting up the third bit from the message S using the reduced values of segment 1

Reduction Results from segment 1	NO	The next bit of the message s_3	Results
	1	0	
A = 0			
B = 1			

Repeating the same process, the NO of segment 1 and the third bit of the message will be encoded using the IANS.

Table 4.6: The second segment, encoding the third bit from the message S using IANS.

Reduction Results from segment 1	NO	The next bit of message s_3	Results
	1	0	
A = 0	1+0 = 1		A = 1
B = 1		1+1 = 2	B = 2

After the IANS is used to encode segment 2, the results will be subtracted from each other until 0 is reached, and that will result in $A = 2 - 1 = 1$ then $B = 2 - 1 = 1$, and finally $A = 1 - 1 = 0$.

Table 4.7: Reducing the generated values of segment 2.

Results from segment 2	Reducing results from segment 2			Results
A = 1	2-1 = 1		1-1 = 0	A = 0
B = 2		2-1 = 1		B = 1

The final results are $A = 0$ and $B = 1$. This concludes the second segment. The results will be used to initialise the third segment. The NO to reach 0, is 3, which will be encoded as {1}. The encoded NO will be inserted as the first bit for the next segment to be encoded using the IANS. The fourth bit from the message S can now be inserted next to the bit that represents the NO from the previous segment. This will result in {11} ready to be encoded using the IANS.

Table 4.8: The third segment, setting up the fourth bit from the message *S* using the reduced values of segment 2

Reduction Results from segment 2	NO	The next bit of the message s_4	Results
	1	1	
A = 0			
B = 1			

After setting up the values for the NO and the fourth bit, the message will be encoded using the IANS.

Table 4.9: The third segment, encoding the fourth bit from the message *S* using IANS.

Reduction Results from segment 2	NO	The next bit of message s_4	Results
	1	1	
A = 0	1+0 = 1	1+1 = 2	A = 2
B = 1			B = 1

The results will be subtracted from each other until 0 is reached, which will result in $B = 2 - 1 = 1$ then $A = 2 - 1 = 1$ and, finally, $B = 1 - 1 = 0$.

Table 4.10: Reducing the generated values of segment 3.

Results from segment 3	Reducing results from segment 3			Results
A = 2		1-1 = 1		A = 1
B = 1	2-1 = 1		1-1 = 0	B = 0

As shown in Table 4.10, the final segment has been encoded and compressed, resulting in *A = 1* and *B = 0*, The NO to reach 0, is 3, encoded as {1}. This will represent the first bit of the encoded message that will be transmitted. Moreover, the results of the reduction are *A = 1* and *B = 0* encoded as {1}. This will represent the second bit of the encoded message to be transmitted. The encoded message {11} can now be transmitted to the decoder. Since there is no flag to determine the end of the message, the length of the message will be required (see Section 3.4.3 for the discussion about flagging the end of the message).

4.3.2. De-compression using the IANS

The de-compression process works backwards. Once the decoder receives the encoded message that consists of two bits, it decodes the information from the LSB when the encoding starts at the MSB. The IANS compression method utilises each segment (a part of the first encoded segment) to encode the NO from the previous segment as well as one bit of information from the message S. In addition, the reduced values of the previous segment are used to initialise the next segment. Consequently, the first step of decoding the message can be defined by the NO and the encoded values of A and B, as shown in Table 4.2. The decoder receives two bits of information: the first bit represents the NO and the second bit represents the values in A and B. When the first bit is {0}, it represents two operations: the decoder will add the values of A and B once, and the results will be subtracted from the previous value. When the first received bit is {1}, that represents three operations and reverses the process of the encoder. The decoder will, therefore, add the values of A and B twice, and the results will be subtracted from the previous value. At this point, the results will be the IANS values of the encoded message.

Following the decoding process explained in sub-section 3.2.2, the IANS will decode the message and the initialisation of the last segment will be retrieved. This is the same as the values of A and B of the previous segment. The second bit received by the decoder can be {0} or $\{1\}$, When the decoder receives $\{0\}$, then $A = 1$ and $B = 0$, if it, however, receives $\{1\}$, then $A = 0$ and $B = 1$. Consequently, when the decoder receives $\{00\}$ it outputs $\{01\}$, and if the decoder receives $\{01\}$, then the output will be $\{00\}$. Furthermore, if the decoder received $\{10\}$, it will output $\{10\}$, and, finally, if the decoder received $\{11\}$, then the decoder outputs $\{11\}$. This is illustrated in Table 4.11.

Table 4.11: Decoding

Received message at the decoder		Generate		
Encoded NO	Encoded A and B	NO	Value position	Decode
0	0	2	$B = 0$	01
0	1	2	$A = 0$	00
1	0	3	$A = 0$	10
1	1	3	$B = 0$	11

Once the first two bits are decoded, the results will consist of the values A and B, and the IANS decoder will use these values to decode 2 bits of information. The first bit of the output is the last bit of the encoded message S, while the second bit of the output represents the NO of the previous segment. Decoding the message S will illustrate this further. Assume that the decoder received the message (as in the previous example) $\{11\}$, the first bit is $\{1\}$ which represents the NO. From Table 4.11, it can be seen that it is equal to 3 operations, and the decoding will be the reverse of the encoding calculations, that is, adding the values of A and B twice, and subtracting the results from the previous value. The second bit from the received message is also $\{1\}$, that is 0, and therefore, $A = 1$. This is because the encoder repeatedly subtracted the values of A and B to reach 0 and, in consequence, the value prior to 0 will always be equal to 1 as shown in Table 4.1. The decoder will then add the values twice, starting from the lowest value. Since the last generated value is $B = 0$, then $B = 1 + 0 = 1$ and $A = 1 + 1 = 2$, subtracting the results of A from B, will result in $B = 2 - 1 = 1$. This generates the values of the IANS ready to be decoded to generate the initialisation of the next segment.

Table 4.12: Segment 3, generating the IANS values from the received message $\{11\}$.

Received message		NO - Reverse process of the encoder			Results
11	A = 1		1+1 = 2		A = 2
	B = 0	1+0 = 1		2-1 = 1	B = 1

The values $A = 2$ and $B = 1$ have now been decoded by segment 3, and they will be decoded further by using the IANS. Since the algorithm uses two bits for each segment, the decoding will stop after the second bit has been extracted. Following the IANS decoding process explained in Section 3.4.2, the decoding starts with comparing A and B. In this case A is greater than $B,$ equation 4.1 will be used to decode the binary stream from LSB.

$$s_i = 1, A_{i-1} = A_i - B_i, B_{i-1} = B_i \quad \text{when } A \geq B \qquad (4.1)$$

$$s_i = 0, B_{i-1} = B_i - A_i, A_{i-1} = A_i \quad \text{when } A < B \qquad (4.2)$$

This will result in, $b_2 = 1$, $A = 2 - 1 = 1$, resulting in $A = 1$ and $B = 1$, which will again satisfy equation 4.1. Therefore, $b_1 = 1$, $A = 1 - 1 = 0$, resulting in $A = 0$ and $B = 1$, represents the starting values of the next segment. The message which is being decoded is {11}: the first bit of this message is the last bit of the encoded message S, while the second bit represents the NO of the next segment.

Table 4.13: Segment3, decoding using the IANS.

Results from segment 3	s_4	NO	Results	Decoded message
	$b_1 = 1$	$b_2 = 1$		
A = 2	2-1 = 1	1-1 = 0	A = 0	S={1}
B = 1			B = 1	

When the last bit of the message {1} has been decoded, the NO and the values for A and B have been generated. The first step of the decoding will be applied again; the NO is {1} as shown in Table 4.11, and it is equal to 3 operations. The decoded values are $A = 0$ and $B = 1$, and, therefore, the values of A and B will be added twice, and the results will be subtracted from the previous value. Since the lowest value is $A = 0$, the calculation will start with A, that is, $A = 1 + 0 = 1$ and $B = 1 + 1 = 2$, then the results of B will be subtracted from A, resulting in $A = 2 - 1 = 1$. This generates the values of the IANS ready to be decoded to generate the initialisation of yet another segment.

Table 4.14: Segment 2, generating the IANS values from NO, *A* and *B* of segment 3.

NO	Values of *A* and *B*	NO - Reverse process of the encoder			Results
1	A = 0	1+0 = 1		2-1 = 1	A = 1
	B = 1		1+1 = 2		B = 2

The values $A = 1$ and $B = 2$ have now been decoded by segment 2, and it is now ready to be further decoded using the IANS. Starting with the comparison of A and B, and with B being greater than A, equation (4.2) will be used to decode the binary stream $b_2 = 0$, $A = 2 - 1 = 1$, resulting in $A = 1$ and $B = 1$, which will satisfy equation 4.1. Therefore, $b_1 = 1$ and $A = 1 - 1 = 0$, resulting in $A = 0$ and $B = 1$, which represents the starting values of the next segment. The decoded message is $\{01\}$, the first bit is the next bit of the encoded message, and the second bit represents the NO of the next segment.

Table 4.15: Segment 2, decoding using the IANS.

Results from segment 3	s_3	NO	Results	Decoded message
	$b_1 = 0$	$b_2 = 1$		
A = 1		1-1 = 0	A = 0	S={01}
B = 2	2-1 = 1		B = 1	

The next bit of the message $\{0\}$, the NO is $\{1\}$ and the values for $A = 0$ and $B = 1$ have been decoded from segment 3. Since the length of the message is 4 bits and the first two bits have been decoded, the decoding process will decode one more segment to further generate two bits. The last two bits of the decoded message will be the first two bits of the message. Further decoding will be applied. Since the lowest value is $A = 0$, the calculation will start with A, $A = 1 + 0 = 1$ and $B = 1 + 1 = 2$, then subtract the results of B from A and resulting in $A = 2 - 1 = 1$. This generates the values of the IANS ready to decode the last two bits of the message S.

Table 4.16: Segment 1, generating the IANS values from NO, *A* and *B* of segment 2.

NO	Values of *A* and *B*	NO - Reverse process of the encoder			Results
1	A = 0	1+0 = 1		2-1 = 1	A = 1
	B = 1		1+1 = 2		B = 2

The values $A = 1$ and $B = 2$ have now been decoded from segment 1, and by decoding the segment further by using the IANS. The first two bits of the message S can be retrieved. B is greater than A, which satisfies equation 4.2, that is, $b_2 = 0$, $A = 2 - 1 = 1$, resulting in $A = 1$ and $B = 1$. Therefore, $b_1 = 1$, and $A = 1 - 1 = 0$, which will give the values of $A = 0$ and $B = 1$. The decoded message is $\{01\}$, the first bit of the decoded message is the second bit s_2 of the message S, and the second decoded bit is the first bit s_1 of the message S. The decoding process will stop and the message $\{1001\}$ is now fully decoded from the transmitted message $\{11\}$.

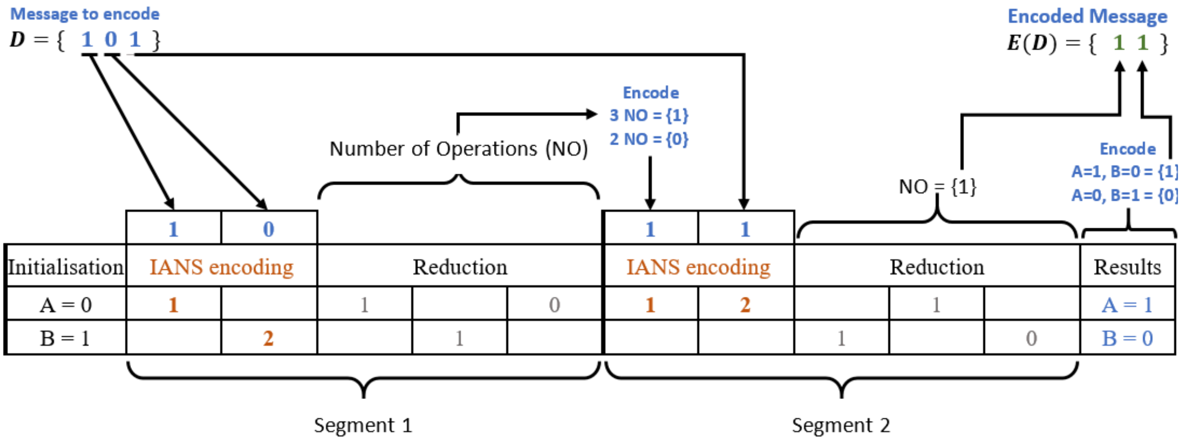
Table 4.17: Segment 1, decoding with the IANS.

Results from segment 3	s_2	s_1	Results	Decoded message
	$b_1 = 0$	$b_2 = 1$		
A = 1		1-1 = 0	A = 0	S={1001}
B = 2	2-1 = 1		B = 1	

4.3.3. Observations

The conditional compression method, using the IANS, outlined in Sections 4.3.1 and 4.3.2, encoded the given message S by using multiple segments where each segment consisted of 2 bits and required two steps to encode 2 bits of information. The first step consisted of encoding the first 2 bits of the given message S using the IANS, and the second step consisted of reducing the results. The output of the second step was used as the initialisation of the next segment, while the encoded NO of the previous segment, which consisted of one bit, was outputted as one bit of information along with the next bit of the message S. These two bits were formed together in preparation to be encoded in the next segment. The process and the steps of compression are illustrated in Figure 4.1.

Figure 4.1: Compression steps using the IANS.



From Figure 4.1 the information required to decode the message is located in the last segment results for A and B, the encoded NO of the last segment, and the number of encoded bits or the number of segments. As the compression method processes one bit at a time of the input data string, it is effective when the input consists of a minimum of three bits, extending to an infinite number of bits. Despite the number of bits processed, the final output consists of only two bits as seen in Figure 4.1. When the input is bigger than two bits, the compression occurs from reusing the results of the reduced A and B in the initialisation of the next segment.

When each segment has an input of two bits and an output of 2 bits, the output bit, which represents the values of A and B, is reused to initialise the second segment. This reduces the output of the previous encoded segment by 1 bit, and, hence, allows one bit from the message S to be encoded into the next segment, while the second bit of the next segment holds information about the previous segment.

The process of reusing the value of the previous segment is proven to be possible using the IANS, whereas this will not be possible if a direct binary system has been used. However, when the compression method has been applied to the total combination of two bits, the results of A and B consist of two possibilities, $A = 0$ and $B = 1$ or $A = 1$ and $B = 0$, as seen in Figure 4.2.

Figure 4.2: The IANS reduction results when the initialisation for A=0 and B=1.

Encoding 2 bits using IANS			Reduction			Results
	0	0				
$A = 0$			1			$A = 1$
$B = 1$	1	1		0		$B = 0$
	0	1				
$A = 0$		1	↶	0		$A = 0$
$B = 1$	1		1	↷		$B = 1$
	1	0				
$A = 0$	1		1		0	$A = 0$
$B = 1$		2		1		$B = 1$
	1	1				
$A = 0$	1	2		1		$A = 1$
$B = 1$			1		0	$B = 0$

When the reduction results of the first segment are $A = 0$ and $B = 1$, the second segment can be encoded with the IANS as seen in Figure 4.2 since the reduction results are equal to the initialisation values. However, when the results are $A = 1$ and $B = 0$, the reduction results will be changed to the inverse of the initialisations with $A = 0$ and $B = 1$. This is shown in Figure 4.3.

Figure 4.3: The IANS reduction results when the initialisation for A=1 and B=0.

Encoding 2 bits using IANS			Reduction			Results
	0	0				
$A = 1$			1		0	$A = 0$
$B = 0$	1	2		1		$B = 1$
	0	1				
$A = 1$		2		1		$A = 1$
$B = 0$	1		1		0	$B = 0$
	1	0				
$A = 1$	1		1	↶		$A = 1$
$B = 0$		1	↷	0		$B = 0$
	1	1				
$A = 1$	1	1		0		$A = 0$
$B = 0$			1			$B = 1$

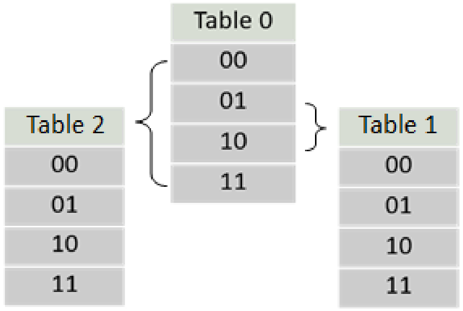
Also, the NO in Figure 4.2 is the inverse of the NO in Figure 4.3. That is, the NO for $\{00\}$ and $\{01\}$ is two in Figure 4.2, while the NO for $\{00\}$ and $\{01\}$ is three in Figure 4.3. This also applies to $\{10\}$ and $\{11\}$. Consequently, if the result of any segment (a part of the last segment) has the opposite initialisation of some other segment during the encoding process, the decoder cannot determine whether the results of the decoded segment are related to the results of either in Figure 4.2 or Figure 4.3. In addition, and apart from the last segment, if the results of the reduction of A and B for all other segments are equal, the method will result in a high compression ratio as can be seen in the previous examples. It may be possible to create an algorithm to determine the initialisation of A and B for each segment to solve this problem. In consequence, further study and analysis have been carried out, which branched into a novel method that is based on the *Run length encoding* covered in Section 4.4. The study of compression using the IANS will continue in Chapter 5.

4.4. The Leading Bit

In order to create an algorithm which would determine the initialisation of A and B for each segment and, thus, solve the problem of whether their reduction applies to all other segments, the following analysis and the development of a system using the Leading bit has been taken up. This has not only drawn on the earlier work on the ANS and its improved version, the IANS, but also, and mainly, focused on a novel method based on *the Run Length Encoding* (RLE). Figure 4.2 and Figure 4.3 show that each generated value has a unique combination of NO and the order of A and B when two bits of information are being encoded. In contrast, in order to simplify the process, both the NO and the order of A and B can be replaced by a unique number. Each number leads to a table that consists of the same information. The first bit of a given message can be used as the leading bit to encode the rest of the message.

From what has become clear from the earlier attempt of compression by using the IANS in Section 4.3, instead of passing the NO to the next segment, it has been replaced by linking the combinations of bits that start with $\{0\}$ to one table and the combinations of bits that start with $\{1\}$ to another table. Each table can be tagged as Table 1 and Table 2, as shown in Figure 4.4.

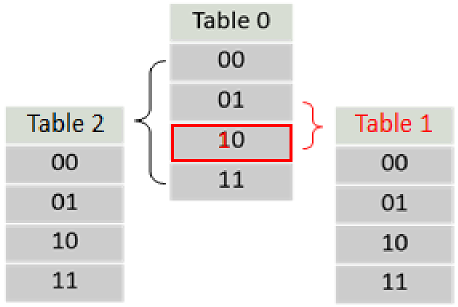
Figure 4.4: Replacing the NO and the order of *A* and *B* with Table 1 and Table 2



Starting from Table 0, if the message consists of a combination of two bits and the first bit is {0}, there are two possibilities that begin with 0. If the second bit of the message is {0}, then {00} will lead to Table 2, while if the second bit is {1}, then {01} leads to Table 1. Furthermore, if the message starts with {10}, then it will lead to Table 1 and {11} will lead to Table 2. The first bit of the message is considered the leading bit for both the encoding and decoding processes. In the encoding of the third bit according to either Table 1 or Table 2, the first bit of the message will be selected again, and this is, then, followed by the third bit of the message. Each table can be represented with one bit, while the encoded message consists of two bits, which will result in three bits that are required to decode the message. For example:

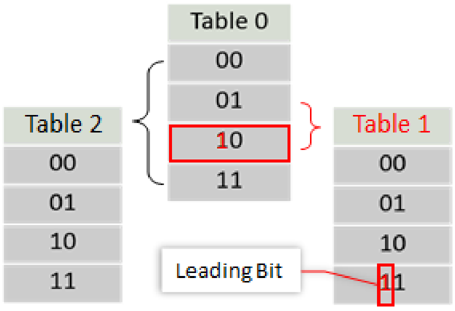
- if the message {101} needs to be encoded, the first two bits are {1} and {0}.
- the leading bit is, therefore {1} and the second bit {0}, which will be directed to Table 1 as shown in Figure 4.5.

Figure 4.5: Encoding the first two bits of the message {101}by using the leading bit



- from Table 1, the leading bit is selected from the message {1} and the third bit of the message is inserted in the Table. This will give {11}, as seen in Figure 4.6.

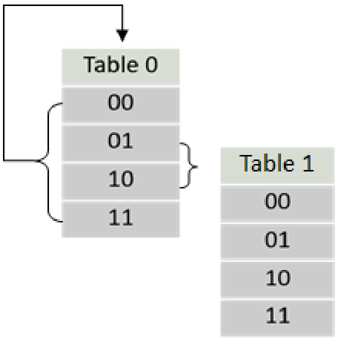
Figure 4.6: Encoding the first bit of Table 2 as the leading bit and following this with the third bit of the message {101}



- encoding Table 1 with {1} and Table 2 with {0}, and then
- emitting the last two bits of the encoded message along with the number of the table, so, in this case, the last two bits are {11} and Table 1 gives {1}, resulting in {111}

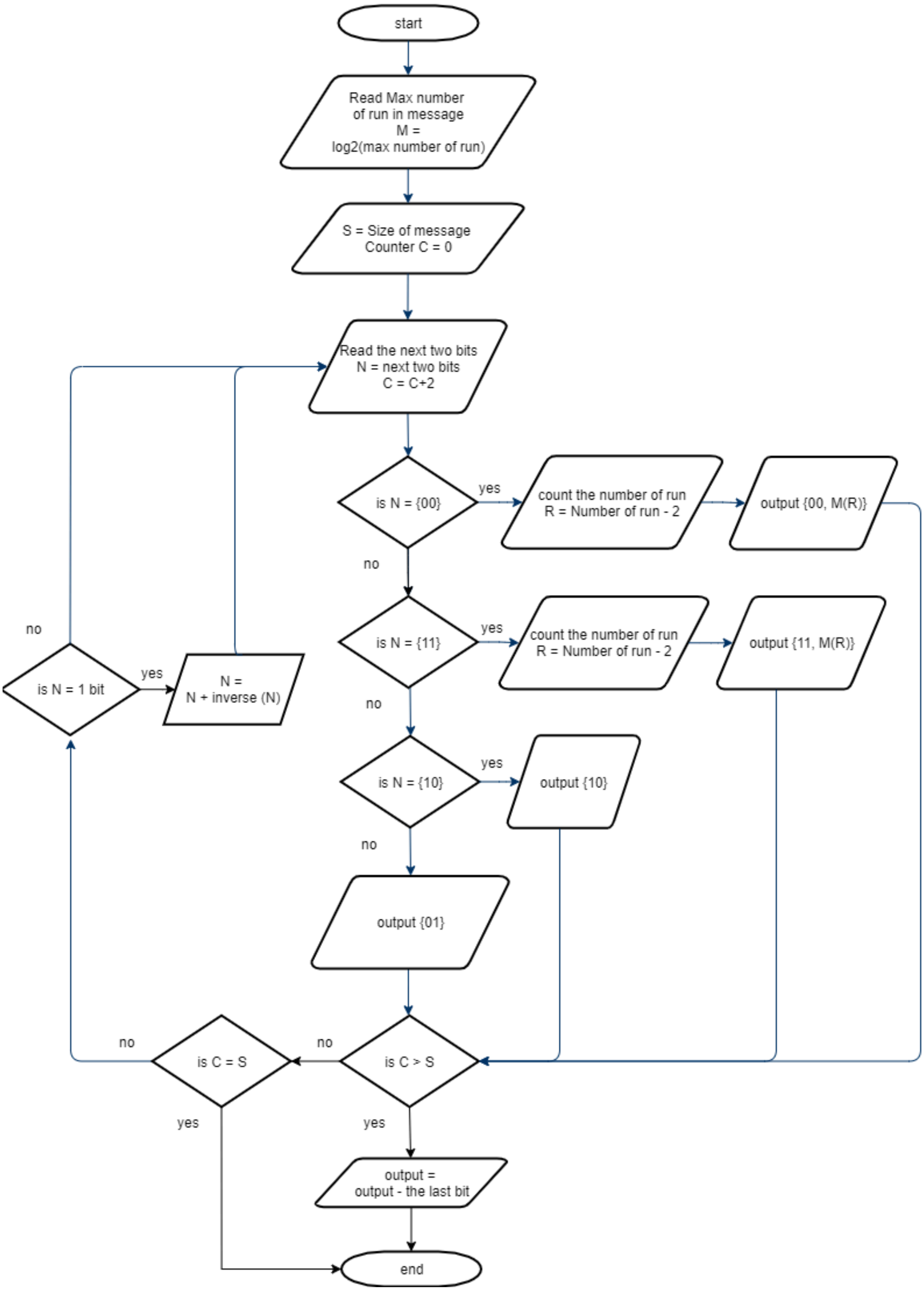
If, however, the direction is changed to lead to Table 2 and is fed back to Table 0, any message that consists of a repeated bit will lead back to Table 0 until the repeated bit in the message changes, which will, then, lead to Table 1. However, the number of repeated bits is required to determine the number of loops occurring in order to encode the message. If the message starts with 10 or 01, it will lead to Table 1, and the three bits of information can be encoded as illustrated in Figure 4.7.

Figure 4.7: Feedback process of a repeated bit



The method is suitable for data that contain multiple runs of zeros or ones as they loop back to Table 0, while Table 1 is used to indicate the change in data (exit from the loop). Therefore {11} can be assigned to indicate a run of ones from Table 1 and {00} as a run of zeros. In this case, the number of runs of the repeated bit can be included, {11}, while {00} will be used as a flag to indicate a read for the length of repeated bits post of the flag. Similarly, {01} can be assigned to indicate a message with two bits that contain {01} and the same for {10}. This will make the method very similar to Run Length Encoding but with the added advantage of detecting a change in data without adding the cost of extra bits. This is important, in particular, with messages containing a run of ones with few zeros in between, or a run of zeros with few ones in between that RLE, which in some cases will result in negative compression. Figure 4.8 illustrates the Leading Bit process in more detail.

Figure 4.8: Leading Bit process



start
Read Max number of run in message M = log2(max number of run)
S = Size of message Counter C = 0
Read the next two bits N = next two bits C = C+2
is N = {00}
yes
count the number of run R = Number of run - 2
output {00, M(R)}
no
is N = {11}
yes
count the number of run R = Number of run - 2
output {11, M(R)}
no
is N = {10}
yes
output {10}
no
output {01}
is N = 1 bit
no
yes
N = N + inverse (N)
is C = S
no
yes
is C > S
no
yes
output = output - the last bit
end

For example, if the message *b*={1010 1111 1111 1111 1111}, which contains 20 bits, is encoded using the leading bit method, the output is {10}{10}{11,14}, where the maximum run is 16 bits; then 2 bits can be used to identify the run of {1}, resulting in a maximum run of 14 bits, and, finally, 4 bits can be used to represent the run. Similar to RLE, when converting the run to binaries, it will give {10}{10}{11,1110}, that is 10 bits. Comparing the method to RLE, the maximum run of the message is 16 bits, while using RLE will require 4 bits to represent each run, which will result in {0001}{0001}{0001}{0001}{1111}, that is, 20 bits.

4.5. Summary

This chapter explored the Improved Adaptive Numeral System (IANS) as a potential approach to data compression, highlighting its structural advantages over the traditional binary system. A conditional compression method was identified, demonstrating that the IANS can use previous segment results to initialise subsequent ones, confirming the hypothesis. Additionally, an alternative approach, The Leading Bit, was introduced as another attempt to enhance compression using numeral system-based encoding. However, the IANS compression method was found to be conditional, depending on the structure of the input data string, and did not apply universally to all data combinations. Moreover, The Leading Bit method failed to achieve substantial compression. Despite these limitations, the experiments provided valuable insights into numeral system-based data encoding and revealed structural constraints that affect their viability for practical compression applications.

Chapter 5

5.1. Introduction

Building upon the findings of Chapter 4, this chapter introduces a novel approach to data compression known as Data Extraction (DE). Unlike previous methods that relied only on structural properties of numeral systems, DE introduces a mechanism where the number of bits to compress is controlled and correlated with the length of the data string. This is achieved by segmenting the input data based on a selected number of bits to compress, which in turn determines the number and maximum length of segments. The key advantage of this approach is that each segment retains information necessary to reconstruct the previous segment, facilitating incremental processing. A critical aspect of DE is the flagging system, which plays a fundamental role in segment identification and ensures proper encoding and decoding. The chapter explores multiple flagging order solutions and discusses how different numeral systems and flag systems impact the effectiveness of DE.

A key challenge in implementing DE is the potential for data overhead introduced during encoding, which must be managed to maintain compression efficiency. The Zeckendorf representation, a numeral system that represents numbers as sums of Fibonacci numbers, is examined as a potential flagging approach. However, due to its inherent structural constraints, such as ambiguous termination flags, it is found to be unsuitable for Data Extraction. Instead, the ANS and IANS numeral systems are considered as alternatives that can be modified to support a more reliable flagging system while avoiding the challenges of Zeckendorf's representation.

5.2. Data Extraction (DE)

Following the results obtained from previous studies in data compression where the Improved Adaptive Numeral System (IANS) was used, further research has led to a novel method of compression in which the number of bits required to compress can be controlled and correlated with the length of the data string. This is possible when the input data string is divided into segments where the selected number of bits to compress determines the total number of segments and the maximum length for all segments. These characteristics allow the incremental processing and identification of the length of each segment, which is based on the value of the selected number of bits to compress. Consequently, each segment will have the information to retrieve the data of the previous segment.

The process includes the following steps:

Firstly, the input data string *S*, is divided into segments, that is, *S(SGs)*, while the number of bits to compress can be selected. This is denoted by C_l for all segments. The total number of segments to process can be determined when the length of *S* is known, as follows:

$$S(SGs) = \frac{S_l - S(sg_x)_l}{C_l - F_l} \qquad (5.1)$$

The length of each segment $S(sg_x)_l$ is, then, determined by the value of *C*, denoted by C_x:

$$S(sg_x)_l = C_x \qquad (5.2)$$

Since C_x determines the length of $S(sg_x)_l$, the selected number of bits to be compressed must be smaller, or equal, to the size of the processor cache to process the data. The value *C* of each segment $S(sg_x)_l$ can be determined by the following equation:

$$S(sg_x)_l = C_x = \sum_{i=0}^{i=c_l+1} 2^i b_i \qquad (5.3)$$

where b is the binary bit in *C*. Once the value of $S(sg_x)_l$ has been determined, the value of the segment $S(sg_x)$ can be obtained by using the following equation:

$$S(sg_x) = \sum_{i=0}^{i=S(sg_x)_{l-1}} 2^i b_i \qquad (5.4)$$

When $S(sg_x)_l$ is known, the compressed C_x can be obtained from the selected C_l. It can be concluded from the above that by using the Data Extraction algorithm, data compression can be obtained when the number of bits to compress for each segment is identified from the input data string if the following conditions are met:

1. the length of each segment is known, and
2. the cost of identifying the segment length is less than the specified number of bits to compress C_l.

5.2.1. Flags

The conditions required to achieve the compression listed above in the previous section, can be achieved by flagging the end of each segment with a flag code to identify the length of the segment. The compression efficiency is, then, directly related to the length of C_l and the flag code length. If the length of $C_l = x$, the flag F must be at least 1 bit smaller than C_l, F = C_l - 1 to ensure that at least 1 bit of information can be encoded for every segment. For example, if C_l = 5 bits, then the Flag must be $F \leq 4$ bits. The maximum number of bits in each segment, when C = 5 bits, will be 2^5 = 32 bits. Each segment will be shifted left by 5 bits, allowing at least 1 bit of information to be encoded, and a maximum of 4 bits can be used to flag the end of the string for each segment. However, the size of the string will fit multiple flags, more precisely, the maximum number of bits in each segment can fit approximately six flags as 32/5 = 6.4 flags. Therefore, if the same flag code is used, it will result in an undefined order of six identical flags in each segment with the same code, resulting in data corruption. Therefore, the identification of either the order of the flag or six different flags are required to determine the end of each segment. The flag order is explored in more detail in Section 5.2.2. The following example will explore how Data Extraction can be used to encode information and the challenges associated with flagging the end of each segment.

If the message **E** = {1010 1000 0110 1111 0010}, is encoded using $C_l = 4$ and $F_l = 2$, then the number of bits to be compressed per segment is $C_l - F_l = 2$, the following steps show how the encoder will process the information:

The first four bits, that is C_l = *4*, are given the value of C_1 = *10* in the message E, which contains 20 bits. Since $S(sg_x)_l = C_x$, then the first segment is $S(sg_1)_l = C_1 = 10$. The 10 bits of the message following C_l are read, and a flag is placed after the 10th bit of the first segment:

$$\mathbf{e1} = \{\ \overbrace{1\ 0\ 1\ 0}^{\mathrm{C}=10}\ \overbrace{\underbrace{1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1}_{S(sg_1)}}^{S(sg_1)_l\,=\,C}\ \mathrm{F(1)}\ 1\ 1\ 0\ 0\ 1\ 0\ \}$$

The second segment e2 will be shifted by 4 bits, and it will start from the 5th bit of the message E. Since C_2 = 8 the, flag will be placed on the 9th bit of $S(sg_2)$:

$$\mathbf{e2} = \{\ \overbrace{1\ 0\ 0\ 0}^{\mathrm{C}=8}\ \overbrace{\underbrace{0\ 1\ 1\ 0\ 1\ 1\ \mathrm{F(1)}\ 1\ 1}_{S(sg_2)}}^{S(sg_2)_l\,=\,C}\ \mathrm{F(2)}\ 0\ 0\ 1\ 0\ \}$$

The third segment e3 will be shifted again by 4 bits and flagged at the 7th bit of the $S(sg_3)$:

$$\mathbf{e3} = \{\ \overbrace{0\ 1\ 1\ 0}^{\mathrm{C}=6}\ \overbrace{\underbrace{1\ 1\ \mathrm{F(1)}\ 1\ 1\ \mathrm{F(2)}\ 0\ 0}_{S(sg_3)}}^{S(sg_3)_l\,=\,C}\ \mathrm{F(3)}\ 1\ 0\ \}$$

The fourth segment e4 will be shifted by 4 bits and flagged at the 16th bit of the $S(sg_4)$:

$$\mathbf{e4} = \{\ \overbrace{1\ 1\ \mathrm{F(1)}\ 1\ 1}^{\mathrm{C}=15}\ \overbrace{\mathrm{F(2)}\ 0\ 0\ \mathrm{F(3)}\ 1\ 0}^{S(sg_4)_l\,\neq\,C}\ \}$$

As e4 contains C_{e4} = 15 and the length of e4 is 4 bits, excluding C_l, there are no more bits to fit this size of the last segment. Therefore, $S(sg_3)$ = { 1 1 F(1) 1 1 F(2) 0 0 F(3) 1 0 } can be transmitted along with the size of compression selected C_l = 4, and the receiver can decode the message to retrieve the original 20 bits from the compressed 8 bits of $S(sg_3)$ (excluding the flags).

As $S(sg_x)_l = C_x$, the decoding process involves tallying the bit count in the string from the least significant bit (LSB) until a flag is detected for each segment, where the total bit count in the segment equals C_x.

$$S(sg_3) = \{\ \underbrace{1\ 1\ F(1)\ 1\ 1\ F(2)\ 0\ 0}_{S(sg_3)_l = 6}\ F(3)\ 1\ 0\}$$

Since the length of $S(sg_3)_l = 6$ and the length of C is given $C_l = 4$, the value of the first C_l can be obtained by counting the number of bits from the first flag of the segment and converting the number to a decimal value in a 4-bit string $C_l = \{0110\}$.

$$\mathbf{e3} = \{\ \overbrace{0\ \ 1\ \ 1\ \ 0}^{C = S(sg_3)_l}\ \ \overbrace{1\ \ 1\ \ F(1)\ \ 1\ \ 1\ \ F(2)\ \ 0\ \ 0}^{S(sg_3)_l\ =\ 6}\ \ F(3)\ \ 1\ \ 0\ \}$$

Once the e3 is obtained, Flag 3 will be removed from the string. The next flag is located on the 9th bit of the string, $S(sg_2)_l = 8$, therefore $C_2 = \{1000\}$.

$$\mathbf{e2} = \{\ \overbrace{1\ \ 0\ \ 0\ \ 0}^{C = S(sg_2)_l}\ \ \overbrace{0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ F(1)\ \ 1\ \ 1}^{S(sg_2)_l\ =\ 8}\ \ F(2)\ \ 0\ \ 0\ \ 1\ \ 0\ \}$$

As e2 is obtained, Flag 2 will be removed and the next flag will be located on the 11th bit of the string, $S(sg_1)_l = 10$, therefore $C_l = \{1010\}$, resulting in:

$$\mathbf{e1} = \{\ \overbrace{1\ \ 0\ \ 1\ \ 0}^{C = 10}\ \ \overbrace{1\ \ 0\ \ 0\ \ 0\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1}^{S(sg_1)_l\ =\ C}\ \ F(1)\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 0\ \}$$

As there are no more flags, the decoder will stop with the final decoded message: **E** = {1010 1000 0110 1111 0010}

The preceding example illustrates the successful encoding and decoding of DE when the flag maintains order after the C_l shift, provided that the data to compress in each segment exceeds the previous flag $C_x > F_{x-1}$. However, if $C_x < F_{x-1}$, this arrangement leads to flag disorder. For instance, when C_{e3} = 3, F(e3) will appear before flag F(e2), the arrangement will lead in flag disorder. This disorder is characterised by F(e1), starting from the least significant bit (MSB), followed by F(e3), and then F(e2).

$$\mathbf{e3} = \{\ \overbrace{0\ \ 0\ \ 1\ \ 1}^{C\,=\,3}\ \ \underbrace{1\ \ 1\ \ F(1)\ \ 1}_{3\text{ bits}}\ \ F(3)\ \ 1\ \ F(2)\ \ 1\ \ 0\ \ 1\ \ 0\ \}$$

The next section will proceed to explore the ways to avoid the flag disorder, the significance of avoiding it, and the possible solutions to solve it in more detail.

5.2.2. Flags order

The example in the section above shows that the order of the flags depends on C_x for each segment, and that the order is required to compress the data. When the flags are not in order, the decoder will not be able to identify which flag relates to each segment unless the order of the flags is known. If a false flag is selected, it will result in data corruption. In the following discussion, three solutions are proposed to address this issue and explore their limitations. This problem could be resolved by applying one of the following approaches:

1. **Flags Information:** Initialising the number of bits in the flag where $F_l \leq C_l$ - 2 and then introducing a string of data to determine the order of the flags. When the highest flag number occurs first from the LSB to MSB, 1 is transmitted. Otherwise, 0 is transmitted for each flag that does not correspond to the flag related to the segment. For example, in the model at the end of the previous section, the flag of e3 is the second flag from the LSB, therefore 0 will be transmitted, to indicate that the first flag (from the LSB) is not related to the current segment, and 0 is followed by 1 to indicate that the second flag is related to the current segment. The flag for e2 is placed first from the LSB, so 1 will be transmitted. Likewise, the flag for e1 is set first from the LSB, and 1 will be transmitted. The flag string will be 0111. This solution costs at least 1 bit for each segment when the flag is placed first from the

LSB. It can lead to data expansion in some segments if the multiple flags are not in order.

2. **Flag to flag:** The second solution would be the counting of the number of bits between the flags instead of counting the number of bits from C_l. For example, if C = 3 for the 3rd segment (e3), instead of counting 3 bits after C, it will start from flag 2, extending it to 3 bits and placing the flag after the 3rd bits as follows:

$$\mathbf{e3} = \{\ \overbrace{0\ \ 0\ \ 1\ \ 1}^{C = 3}\ \ 1\ \ 1\ \ F(1)\ \ 1\ \ 1\ \ F(2)\ \ \underbrace{1\ \ 0\ \ 1}_{3\ \text{bits}}\ \ F(3)\ \ 0\ \}$$

This will ensure that the first flag from the LSB is always the flag which is related to the corresponding segment, and the value of C will be equal to the number of bits between the first two flags from the LSB of each segment. This solution provides compression, equivalent to the number of bits in C - F_l for each segment, without any additional cost. However, it changes the additional data cost presented in Solution I to the cost of increasing the number of bits to be processed for each segment. It also increases the processing power for both the encoder and decoder. For example, if C = 16 of e3, flag 3 will be placed after 16 bits from flag 2 with total bits of e3 = 20 bits. Then if the next segment e4 contains C_{e4} = 16 bits, flag 4 will be placed in the 33rd bit of e4, leading to a longer data string that needs to be processed for each $S(sg)_x$.

3. **Flag to flag with flag information:** A third solution uses the combination of the proposed solutions one and two, by limiting the increase in the size of data to be processed as well as the number of bits to be transmitted. It is achieved by subtracting the value of C from the number of bits to reach the previous segment flag (F_{i-1}). If the answer is equal to or greater than the number of bits needed to reach F_{i-1}, the bits from the start of the segment will be counted and the flag placed accordingly. Transmitting 0 to indicate that value C is equal to the number of bits from the start of the segment, excluding C. Alternatively, when the value of C is less

than the number of bits to reach the previous flag, $C < F_{i-1}$. The bits that are equivalent to C will be counted from F_{i-1} and place F_i accordingly. Then 1 is transmitted to indicate that value C is equal to the number of bits between F_{i-1} and F_i. If the example e3 is used, where $C = 3$ from Solution II above, there will be 4 bits to reach F_2, which is less than C, 1 will be transmitted to indicate the count from F_2. Therefore, the flag for e3 will be placed after three bits from flag 2.

$$\mathbf{e3} = \{\ \overbrace{0\ \ 0\ \ 1\ \ 1}^{C = 3}\ \ 1\ \ 1\ \ F(1)\ \ 1\ \ 1\ \ F(2)\ \ \underbrace{1\ \ 0\ \ 1}_{3\ \text{bits}}\ \ F(3)\ \ 0\ \}$$

Compared to Solution I, Solution II will limit the data transmission to only 1 bit, and for the selected $C_l = 4$, the flag size needs to be the maximum 2 bits to obtain average compressions of 1 bit for every segment. Alternatively, the flag size can be increased with the cost of increasing C_l, which, in turn, will cause an increase in both the length and the possible number of flags for each segment. Chapter 7 will offer an in-depth analysis of the three proposed solutions, thoroughly examining the trade-offs, advantages, and implications associated with each approach. This will allow for a comprehensive understanding of their relative effectiveness and applicability. The next section will shift focus to explore the flagging system for Data Extraction outlining its functionality, advantages, and potential challenges.

5.2.3. Flag systems

The previous section explored three solutions to resolve the flagging order problem in Data Extraction. However, Data Extraction will require a particular numeral system or flag system that has the characteristics to support the flagging solutions. These solutions may be changed or updated depending on the flag system structure. This section will discuss the possible flag systems that can be used for Data Extraction and which is expected to result in data overhead in the encoding process. However, as the Data Extraction method has a high compression volume during the encoding process, the data overhead can be controlled and managed by ensuring that C_l in every segment in Data Extraction compresses at least 1 bit from each segment, taking into consideration the length of each segment.

One of the well-known methods of flagging systems is Zeckendorf's representation, which represents any integer as sums of Fibonacci numbers. The method generates a code that does not contain two consecutive {1}, which would act as a termination flag for each encoded integer to differentiate the code words unambiguously from a long stream of codes [45]. This characteristic can be used to identify the length of the segment in the Data Extraction method. It will require converting the full input data string to Zeckendorf's representation and using the termination flags to identify the end of each segment and the use of Data Extraction instead of flagging the encoded integers. Furthermore, as Zeckendorf's representation uses two consecutive 1's to flag the end of the segment, if the flag falls behind, or after, a bit with the value = 1, this will generate three consecutive {1}. Therefore, the flag cannot be identified as it can be either the first two {1} or the last two {1}. For example, given the Zeckendorf representation of 57 as Z = {1000 0000 10}, if a flag is encoded at the 9th bit using Data Extraction (DE), the resulting flagged sequence becomes Z(DE) = {1000 0000 1110}. During decoding, the system recognises a flag within the repeated {11} sequence but cannot determine whether it starts at the 9th or 10th bit, leading to uncertainty in segment length and potential data corruption. These challenges make the method unsuitable for the Data Extraction method. However, the ANS and IANS have properties that can be modified to use a flagging system similar to Zeckendorf's representation but avoid the associated challenges to Zeckendorf's representation. This will be explored in the next Chapter.

5.3. Summary

This chapter investigated Data Extraction (DE) as a potential compression technique, focusing on the segmentation of input data based on a predefined number of bits to compress. By establishing a relationship between segment length and compression parameters, DE allows for incremental processing and ensures that each segment retains information necessary for reconstructing previous segments. The chapter explored the role of flags, their order, and different flag systems in managing segment structure and influencing the compression process. Three flagging order solutions were examined, but their effectiveness depends on the numeral system used. While Zeckendorf's representation was initially considered due to its unique termination flag property, it was found to be unsuitable due to potential ambiguity in identifying segment lengths, leading to decoding errors. However, the ANS and IANS numeral systems demonstrated properties that could be adapted to provide a more robust flagging system for DE. The findings highlight that while Data Extraction has a high compression potential, its success depends on minimising data overhead and ensuring that the selected flagging system supports accurate segment identification without introducing ambiguity.

Chapter 6

6.1. Introduction

This chapter introduce the Modified Adaptive Numeral System (MANS), an enhancement of the Improved Adaptive Numeral System (IANS) designed specifically for Data Extraction. The primary goal of MANS is to efficiently determine segment lengths using a flagging mechanism inspired by Zeckendorf's representation. In MANS, the numeral A signifies transitions between bits (from 1 to 0 or vice versa), while B is used to count repeated bits. This ensures a structured approach to encoding, where consecutive occurrences of A serve as flags to determine segment lengths. MANS overcomes issues in traditional numeral systems by preventing ambiguities arising from consecutive flags and ensuring data integrity during transitions. The system is structured to guarantee that a transition in data will always be followed by at least one encoding in B, ensuring consistency. The chapter further defines MANS mathematically and provides a detailed example demonstrating its encoding process. Additionally, the chapter discusses how MANS expands bit length based on the number of transitions in the data and presents experimental results illustrating its efficiency compared to direct binary and Fibonacci coding.

6.2. Modified Adaptive Numeral System for Data Extraction

To identify the length of the Data Extraction segment, the Adaptive Numeral System (ANS) and the Improved Adaptive Numeral System (IANS) hold characteristics (discussed in Sections 3.3 and 3.4) that can be modified to use the flagging system similar to Zeckendorf's representation. Since A of the IANS represents the 1's and B represents the 0's, A can be modified to identify a switch of the data from 1 to 0 ,or vice versa, represented by 1s, while B can be used to count the occurrences of the repeated bits, represented by 0s. The modified numeral system will avoid the sum of any two consecutive bits in A, which can be used as a flagging system to identify the number of bits in each segment for Data extraction.

It will also avoid the uncertainty of bits related to the flag when three consecutive bits occur during the flagging process. Since the modification of the IANS uses A to represent the switch in the data and not for encoding it, this will ensure that any switch in the data from 1 to 0 or vice versa will always end with encoding B at least once, ensuring that any consecutive occurrence in A will be the flag to determine the value of the segment for the Data Extraction. The Modified Adaptive Numeral System (MANS) can thus be defined by the following equation:

1. $A_0 = 1,\ B_0 = 1 \qquad \mathit{When} \quad s_1 = \{1\} \ \mathit{or} \ s_1 = \{0\}$ (6.1)
   $B_1 = A_0 + B_0,\ A_1 = A_0$
2. $B_{i>1} = A_{i-1} + B_{i-1},\ A_i = A_{i-1} \qquad \mathit{When} \quad s_i = s_{i-1}$ (6.2)
3. $A_{i>1} = B_{i-1} + A_{i-1},\ B_i = B_{i-1} \qquad \mathit{When} \quad s_i \neq s_{i-1}$ (6.3)
   $B_i = A_{i-1} + B_{i-1},\ A_i = A_{i-1}$

The following example shows how MANS encodes information and avoids the occurrence of two consecutive bits. If the message **S** = {1000 1110 0111 1100}, is encoded. Starting from the MSB and using the above equations, it will result in the following:

S	1		0	0	0		1	1	1		0	0		1	1	1	1	1		0	0
MANS(A)		3				14				67			254						1711		
MANS(B)	2		5	8	11		25	39	53		120	187		441	695	949	1203	1457		3168	4879
S(MANS)	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0

The encoding has resulted in A=1711 and B=4879. The values in A and B can also be converted to binary, such as each switch in A can be converted to {1} and every repeated bit in B to {0} resulting in S(MANS)={0100 0100 0100 1000 0010 0}, which will ensure that no consecutive ones occur in the string.

The converted string will enable the process to flag any string of data by using MANS with the {11}. The only scenario where three consecutive ones occur, is when the flag falls before a switch in data, represented in A as each switch in A is always followed by the encoding of B at least once. Therefore, the count of each segment must not include the known switch in A and the first bit from the LSB in any three consecutive bits, as the first bit will always be related to the switch in A.

For example, if $C = 4$ in the example above, the number of repeated bits is counted, which, in this case, is four zeros in the converted string, that is, the fifth bit of S(MANS) and place the flag on the sixth and seventh bit, resulting in S(MANS) = {0100 011100 0100 1000 0010 0} If the next $C = 5$, then after counting five zeros and ignoring the 1's, that is, the ninth bit, the flag will fall on the tenth and eleventh bits, S(MANS) = {0100 0111 0110 0100 1000 0010 0}. The MANS system will ensure that there is always a zero between each flag, which avoids any uncertainty in reading the flag from the data. However, since the MANS is unassigned when the decoder receives either the last two values in A and B or the converted binary string, there is no indication which would identify if the data string from the LSB starts with the $\{1\}$ or $\{0\}$. Hence, MANS values can be assigned by identifying the last value with the cost of adding an additional bit at the end of the encoded string.

If the data ends with 1, MANS will encode A at the end of the encoded string S(MANS) to indicate that the string ends with $\{1\}$. Alternatively, if the string ends with $\{0\}$, the encoder will not change as B will always be encoded at least once at the end of the encoding process, which will indicate that the data starts with $\{0\}$.

The decoding process of MANS requires either the last two decimal values of A and B or the encoded binary string S(MANS). When using the decimal values of A and B, the following equations can be used to decode the information starting from the LSB:

$$s_i = s_{i-1}, \quad B_i = B_{i-1} - A_{i-1}, \quad A_i = A_{i-1} \quad When \quad (B_{i-1} - A_{i-1}) > A_{i-1} \tag{6.4}$$

$$s_i \neq s_{i-1}, \quad A_i = A_{i-1} - B_{i-1}, \quad B_i = B_{i-1} \quad When \quad (B_{i-1} - A_{i-1}) < A_{i-1} \tag{6.5}$$

6.2.1. Analysis and observations of the MANS

As shown in the previous example, MANS provides the characteristics required to identify the segment length, which is a suitable candidate for Data Extraction. MANS will generate additional information to meet the requirements, and this section will cover the testing and analysis of the MANS.

Assume that every switch denoted by W corresponds to the original data string denoted by S. This means that for every W, $S(MANS)$ will generate one additional bit. To assign the value of MANS that identify if the string starts with $\{1\}$ or $\{0\}$, another bit will be added at the end of the string when S ends with $\{1\}$. Therefore the length of MANS can be defined as:

$$S(MANS)_l = S_l + W_{Total} + 1 \qquad (6.6)$$

When the data in S contains a run of ones, the overhead in $S = S_l + 1$, and an additional bit is required to identify the starting bit of the string. However, when S contains a run of zeros, there will be no overhead in $S(MANS)_l$ as there is no data switch, which makes

$$S(MANS)_l = S_l \qquad (6.7)$$

However, when S contains one or more switches, then

$$S(MANS)_l = S_l + W_{Total} \qquad (6.8)$$

as the number of switches in S_l increases, so does the overhead. The maximum overhead in $S(MANS)_l$ occurs when S contains a run of zeros and ones, respectively, This makes $S(MANS)_l$=$S_l \times 2$. Therefore, it can be concluded that:

$$MIN(S(MANS)_l) = S_l \qquad (6.9)$$

$$MAX(S(MANS)_l) = S_l \times 2 \qquad (6.10)$$

Given that the original data string, S, contains 16 bits with five switches from 1 to 0 and vice versa. When converted to $S(MANS)$, the total number of bits will increase to 21 bits. The Modified Adaptive Numeral System has been tested to encode the combinations of up to 50 Mbits of information. The results vary in length as they depend on the number of switches in each binary value, as each switch will cost 1 bit. Table 6.1 illustrates the combinations of a 4-bit binary string of both assigned and unassigned MANS.

Table 6.1: Encoding combinations of 4-bit string using MANS

Decimal	Binary	MANS	Unassigned MANS	Fib
0	0	0	0	0
1	1	01	0	11
2	10	010	010	011
3	11	001	00	0011
4	100	0100	0100	1011
5	101	010101	01010	00011
6	110	0010	0010	10011
7	111	0001	000	01011
8	1000	01000	01000	000011
9	1001	0100101	010010	100011
10	1010	0101010	0101010	010011
11	1011	0101001	010100	001011
12	1100	00100	00100	101011
13	1101	0010101	001010	0000011
14	1110	00010	00010	1000011
15	1111	00001	0000	0100011

Table 6.1 includes also comparisons of the MANS against the well-established Fibonacci numbers and direct binary numbers as is illustrated in Figure 6.1 down below.

Figure 6.1: Direct Binary vs Fibonacci code vs assigned MANS - Bits required to encode a set of values

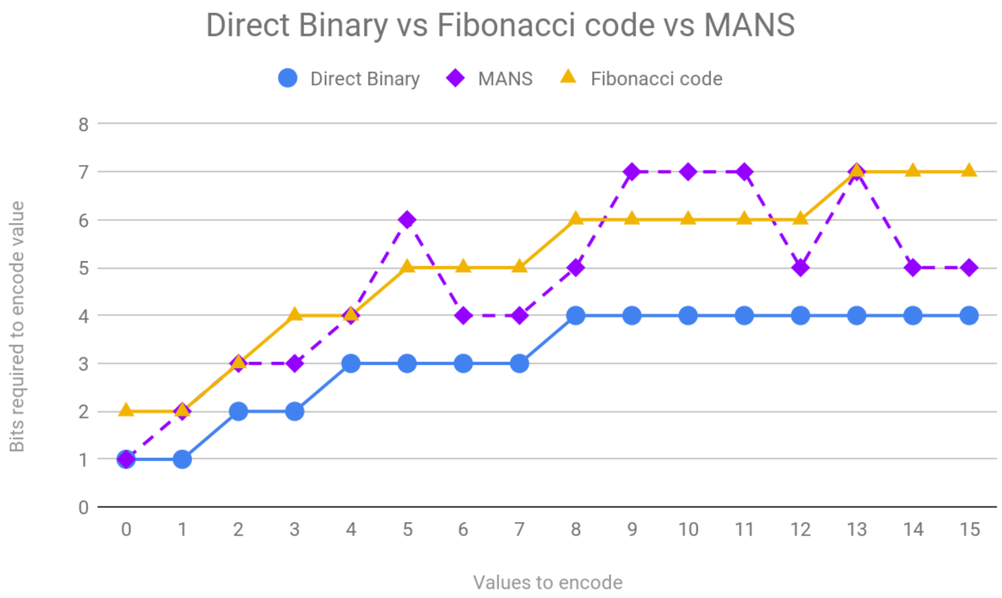
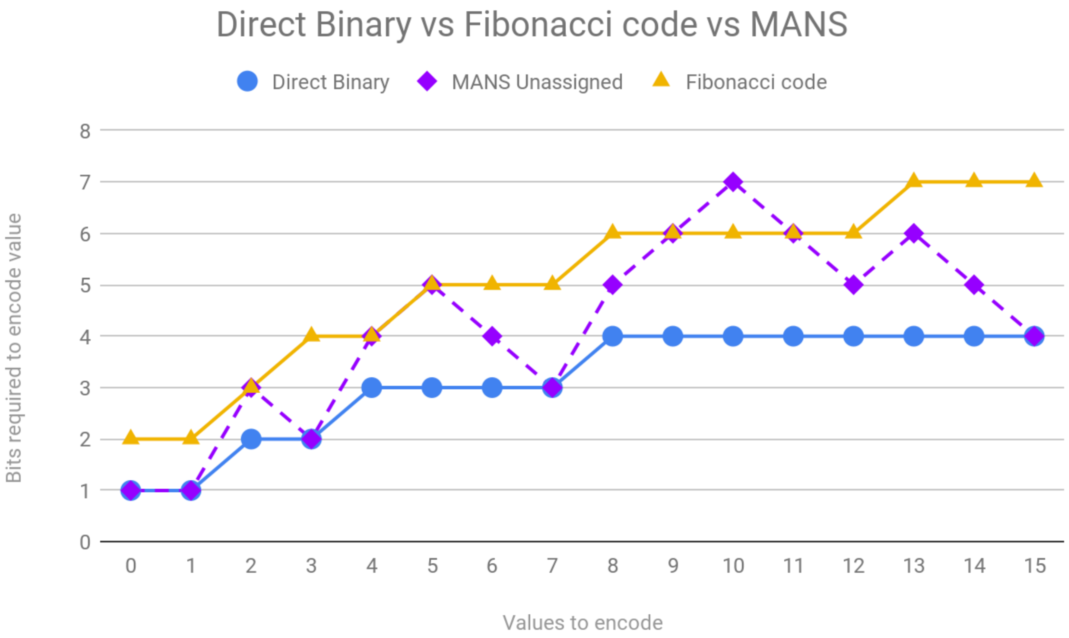


Figure 6.1 shows that 1 value out of 16 from assigned MANS, when compared with direct binary, uses the same number of bits, that is {0}, Ten values out of 16 use one additional bit,

that is, {1, 2, 3, 4, 6, 7, 8, 12, 14, 15} and 5 out of 16 values use 3 additional bits, that is, {5, 9, 10, 11, 13}. However, comparing MANS to the Fibonacci code, MANS shows that 6 values use 1 less bit, that is, {0, 3, 6, 7, 8, 12}, and 2 values use 2 fewer bits, that is, {14, 15}, 4 values use the same number of bits {1, 2, 4, 13}, and, finally, 4 values use 1 more bit {5, 9, 10, 11}.

MANS is structurally designed to encode a large amount of data, mainly for the Data Extraction method. The odd values in Figure 6.1 include an additional bit at the end of the encoded message, which identifies the starting bit of the message as {1}. For demonstration and comparisons against other established numeral systems, the additional bit for the odd values has been added. For example, if the binary values {11, 1111, 111}, are encoded respectively, and if each message is treated separately, the assigned MANS output will read {001, 00001, 0001}. However, if the message has this specific order, then MANS can save an additional 2 bits compared with the results in Figure 6.1, by assigning the final bit only. This is because the message will encode the three values together and identify the last bit, such as {00 0000 000 1}, only once. The unassigned MANS is illustrated in Figure 6.2.

Figure 6.2: Direct Binary vs Fibonacci code vs Unassigned MANS - Bits required to encode a set of values



Given that the MANS value is assigned only once at the conclusion of the data, Figure 6.2 above depicts the bit count required to encode the unassigned MANS, showcasing its overall superior performance compared with the Fibonacci code.

Many tests on MANS have been completed on Text using ASCII characters via the JAVA application. A MANS encoder, decoder and verifier have been designed and implemented to test the system. One of the tests was used on the following message "*Testing the encoding and decoding steps of MANS!*" by converting each character to ASCII code, then converting the ASCII code to MANS. The total number of switches was calculated and compared with the initial analysis and tests. This proved successful. The encoded message was then sent to a decoder that retrieved the ASCII code for each letter, and the full message was retrieved. The results have been verified by checking each binary number and comparing the decoded message against the original message. The full code and output are shown in Appendix A.1. The next section, will cover the encoding process of Data Extraction using MANS.

6.2.2. Encoding Data Extractions using MANS

The following example will show how MANS can be used to encode information and the challenges associated with flagging the end of the string.

If the message **S** = {1000 1110 0111 1100} is encoded starting from the MSB and indicating the number of bits to be compressed as $C_l = 4$, the flag size $F_l = 2$ and the value $F = \{11\}$, then the first 4 bits of S = {1000} and after the conversion, $S(MANS) = \{01000\}$.

Status		Change				Change				Change			...
Data (S)	**1**		**0**	**0**	**0**		**1**	**1**	**1**		**0**	**0**	**…**
MANS (A)		**3**				**14**				**67**			**…**
MANS (B)	**2**		**5**	**8**	**11**		**25**	**34**	**53**		**120**	**187**	**…**
S(MANS)	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**…**

$C_l = 4$

Furthermore, the string, which has been generated, does not include two consecutive 1's. The segment flag will be possible now by encoding {11} in A to identify the length of the segment. The flagging system will use the number of bits, excluding any change in $S(sg_x)$, to count the number of bits for each flag that includes the bits which are related to previous flags from the value S. If the flag code falls before any change in A, $S(MANS)$ will generate three consecutive bits of {1}. When counting the number of bits in $S(sg_x)$, the change to reach the flag is excluded. The flag will always fall under one of the following conditions:

A. one bit after the change in A will result in at least 0 between the change and the flag S(MANS)={1011}, or
B. before the change in A, which will result in $S(MANS)$ = {111}, the third bit will always correspond to the change in A and the first two bits to the flag {11}.
C. If a flag or multiple flags fall after the previous flag that meets the condition in B, this will result in an odd number of {1}, which will indicate that the first bit from the LSB is the bit corresponding to the change in A, while the next two bits correspond to the first flag and the next two bits will correspond to the second flag and so on.

Since the selected C_1 = 4 bits, the first 4 bits of the $S(MANS)$ are used, C_1 = {0100} = 4 instead of S = {1000} to ensure that the compressed data can be retrieved during decoding. Any identifiable {1} related to the data switch is ignored. Consequently, the first flag (F1) of the first segment will start from the fifth bit after C_1 (from the MSB) of the first segment, $S(sg_1)$, and the data on the fifth bit will be shifted two bits to the right to accommodate flagging the segment. These are the number of bits used to flag the segment. Once the flag is placed for each segment, the $S(MANS)$ will be recalculated to include the flag's information:

$S(sg_1)$

Status		Change				Change				F1= 4		Change			...
Data (S)	1		0	0	0		1	1	1	1	1		0	0	...
MANS (A)		3				14				106	159	212			...
MANS (B)	2		5	8	11		25	39	53				265	477	...
S(MANS)	0	1	0	0	0	1	0	0	0	1	1	1	0	0	...

$C_1 = 4$

Now that the flag of the first segment is available, $S(sg_1)$ is shifted by C_1 = 4 bits, resulting in C_2 = {0100} = 4. Flag 2 (F2) will, therefore, be placed on the fifth bit after C_1 for the second segment:

$S(sg_2)$

Status		Change				F1= 4		Change		F2=4			Change		...
Data (S)	0		1	1	1	1	1		0	1	1	0		1	…
MANS (A)		3				14	25	36		108	180		432		…
MANS (B)	2		5	8	11				72			252		684	…
S(MANS)	0	1	0	0	0	1	1	1	0	1	1	0	1	0	…

$C_2 = 4$

Having obtained F2, the data is shifted by 4 bits to proceed to the third segment, resulting in C_3 = {0111} = 7. Consequently after disregarding the switches in the data, flag 3 (F3) will be placed after the eighth bit from C_1 for the third segment $S(sg_3)$:

$S(sg_3)$

Status		F1= 4		Change		F2=4			Change				F3=7		...
Data (S)	1	1	1		0	1	1	0		1	1	1	1	1	…
MANS (A)		3	5	7		16	25		59				270	481	…
MANS (B)	2				9			34		93	152	211			…
S(MANS)	0	1	1	1	0	1	1	0	1	0	0	0	1	1	…

$C_3 = 7$

The fourth segment will be shifted again by 4 bits with C_4 = {0110} = 6, and, therefore, flag 3 will be inserted after the seventh bit from C_1 for segment $S(sg_4)$:

$S(sg_4)$

Status		F2=4			Change				F3=7		F4 = 6				...
Data (S)	0	1	1	0		1	1	1	1	1	1	1	1	1	…
MANS (A)		3	5		12				55	98	141	184			…
MANS (B)	2			7		19	31	43					227	411	…
S(MANS)	0	1	1	0	1	0	0	0	1	1	1	1	0	0	…

$C_4 = 6$

Moving to the fifth segment $S(sg_5)$, shift 4 bits with $C_5 = \{1000\} = 8$, while Flag 5 (F5) will be placed after the last bit of the given data S on the ninth bit after C_1:

$S(sg_5)$

Status	Change				F3=7		F4 = 6				Change			F5 = 8	
Data (S)		1	1	1	1	1	1	1	1	1		0	0	1	1
MANS (A)	2				10	17	24	31			100			369	638
MANS (B)		3	5	7					38	69		169	269		
S(MANS)	1	0	0	0	1	1	1	1	0	0	1	0	0	1	1

$C_5 = 8$

Once again, 4 bits will be shifted for the sixth segment $S(sg_6)$, with the value of $C_6 = 13$:

$S(sg_6)$

Status	F3=7		F4 = 6				Change			F5 = 8		
Data (S)	1	1	1	1	1	1		0	0	1	1	
MANS (A)	2	3	4	5			16			59	102	End of (P)
MANS (B)					6	11		27	43			
S(MANS)	1	1	1	1	0	0	1	0	0	1	1	

$C_6 = 13$

Since there is no more data to process, $S(sg_6)$ will be discarded, and $S(sg_5)$=\{1111 0010 011\} will be the final segment to be transmitted.

The encoding of Data Extraction using MANS of **S** = \{1000 1110 0111 1100\} is now completed, and it has resulted in $S(sg_5)$=\{1111 0010 011\}. The next section will cover the decoding process of Data Extraction using MANS.

6.2.3. Decoding Data extractions using MANS

To illustrate the decoding process of Data Extraction when MANS is used, the example from the encoding section (encoded S) is used, where the final segment has been transmitted from the encoder.

The final segment which was generated, that is, $S(sg_5)$={1111 0010 011} will be the final segment to be transmitted. The decoder receives $S(sg_5)$, and since the length of C_l = 4 bits is used, the decoder will be designed to read the length of each segment and input the value in 4 bit-length to retrieve C. From the LSB, the decoder will scan the given string for the first two consecutive 1's that represent the flag of the first segment, and, once detected, the decoder will count the number of bits starting from the first detected flag until the end of the string. The total number of bits in the count will be the value of C, which has a length of four bits, C_l = 4. The decoder will then consider the following course of action:

1. Since the encoder discarded the 1's related to the change in data from the count, the decoder will dismiss any 1's detected between two 0's {010} from the count to flag the end of the string
2. Dismiss one bit, which is related to the change of the bit that falls after the flag when an odd consecutive bit, such as {111} or {11111} is detected.
3. After determining the value of C, the flag {11} will be removed from the string.

Using the encoded message as an example, the string $S(sg_5)$= {1111 0010 011} was received.

S(MANS)	1	1	1	1	0	0	1	0	0	1	1

The flag starts from the first two bits of the string, and the total number of bits from the detected flag is {11}. Moreover, {010} is detected in the string. Therefore one bit will be discarded from the count, and two bits related to the flag will be removed C = 11 - 2 - 1 = 8. Since the length of C_l is four bits, the results of C will be converted to a binary number with the length of 4 bits, C ={1000} as follows

S(MANS)	1	0	0	0	1	1	1	1	0	0	1	0	0	1	1

C = 8

After removing the first detected flag {11}, the next segment will results in the following:

S(MANS)	1	0	0	0	1	1	1	1	0	0	1	0	0

The next detected flag is, then, placed on the 7th and 8th bit, and as the segment after the flag (from the LSB) does not contain {010}, only two bits related to the flag are removed, resulting in C = 8 - 2 = 6, as shown in the string below:

S(MANS)	0	1	1	0	1	0	0	0	1	1	1	1	0	0	1	0	0

C = 6

Next, after removing the second detected flag, the following segment will result in

S(MANS)	0	1	1	0	1	0	0	0	1	1	0	0	1	0	0

The next detected flag {11} is placed on the 9th and 10th bit, while {010} is detected after the flag in the string. Therefore, one bit will be discarded from the count, and two bits related to the flag will be removed C = 10 - 2 - 1 = 7, resulting in C={0111}:

S(MANS)	0	1	1	1	0	1	1	0	1	0	0	0	1	1	0	0	1	0	0

C = 7

After removing the third detected flag, the next segment will be:

S(MANS)	0	1	1	1	0	1	1	0	1	0	0	0	0	0	1	0	0

The fourth flag {11} has been detected on the 6th and 7th bit, and also three consecutive {1} have been detected. Two bits related to the flag will, therefore, be removed, and one bit related to the three constitutive {1} will be discarded. This will result in $C = 7 - 2 - 1 = 4$:

S(MANS)	0	1	0	0	0	1	1	1	0	1	1	0	1	0	0	0	0	0	1	0	0

C = 4

After removing the fourth detected flag {11} the next segment will result in the following:

S(MANS)	0	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	1	0	0

When the fifth detected flag {11} is placed on the 7th and 8th bit, {010} and {111} are detected, two bits related to the switch are discarded from the count, while two bits that are related to the flag will be discarded, resulting in $C = 8 - 2 - 2 = 4$:

S(MANS)	0	1	0	0	0	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	1	0	0

C = 4

After removing the fifth detected flag, the next segment will result in the following:

S(MANS)	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0

As can be seen, the segment does not include any more flags, indicating the end of the string. The final string obtained is, thus, {0100 0100 0100 1000 0010 0}, which is the MANS representation of the original message.

Using MANS, it is, therefore, possible to convert the encoded string back to the original message. This is done using the following procedure. Since the 0's in MANS represent a repeated bit and 1's represent the switch in data, the decoding will start from the LSB. The first bit from the LSB in *S*(*MANS*) is {0}, indicating that the decoding begins with {0}, outputting {0} in S for every {0} in *S*(*MANS*) until {1} is detected. Once {1} is detected, the output will switch, in this case, to {1}, and outputting {1} for every {0} detected in *S*(*MANS*) until another {1} in *S*(*MANS*) is detected, which indicates the next switch. The process will continue to the end of the string as presented in the following table:

S(MANS)	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0
MANS(A)		3				14				67			254						1711		
MANS(B)	2		5	8	11		25	39	53		120	187		441	695	949	1203	1457		3168	4879
S	1		0	0	0		1	1	1		0	0		1	1	1	1	1		0	0

As can be seen from the table, the original data string has been obtained in **S** = {1000 1110 0111 1100} from $S(sg_5)$ ={1111 0010 011} with 11 bits of data, compared with 16 bits of the initial data string in S, thus achieving 5 bits of compression (31.25%). It can be concluded from this example that when the flags are in order, compression can be achieved using Data Extraction using the Modified Adaptive Numeral System. In this, only the last encoded segment is needed to retrieve the full message. However, if the flags are not in correct order, it will lead to data corruption. Unless the flag order is known, as C_x can be retrieved by counting the number of bits from the known flag. The Data Extraction method of compression is proven possible if the following two conditions are met:

1. a suitable numeral system is selected with the properties which enable Data Extraction, such as MANS, to flag the end of the string,
2. the order of the flags is known.

The first condition was met in the previous example, where MANS was applied to support the flagging system. The next Chapter will cover the second condition, that is, the flags order and explore the possible three solutions introduced in Section 5.2.2, associated with using MANS.

6.3. Summary

This chapter introduced MANS as a novel numeral system designed for effective data extraction by flagging segment lengths without ambiguity. The encoding process relies on A to denote bit transitions and B to count occurrences, ensuring that transitions are always marked and reducing uncertainty in segment identification. MANS introduces additional bits based on the number of transitions, and experimental analysis confirms its effectiveness in encoding large datasets. The chapter demonstrated that MANS adds one bit per switch in the data string and an extra bit if the sequence ends in 1. Testing on data sizes up to 50 Mbits showed variations in length due to transition counts, with comparative analysis against direct binary and Fibonacci coding highlighting MANS's efficiency. Structural optimisations, including flag positioning and bit assignment, allow MANS to reduce bit usage in certain cases, making it a suitable candidate for structured data extraction. The findings suggest that MANS is a viable approach for encoding large datasets while maintaining segment integrity and minimising encoding overhead.

Chapter 7

7.1. Introduction

This chapter provides a comprehensive analysis of the Flag Order System for Data Extraction introduced in Section 5.2.2, utilising the MANS numeral system. The primary objective is to test and evaluate different flagging solutions to enhance data compression efficiency. The study focuses on three key solutions: Flag Information (FI), Flag to Flag (F2F), and Flag to Flag with Flag Information. Each solution is implemented in Java, leveraging core libraries such as java.io.File, java.util.Arrays, and java.math.BigInteger. To ensure a fair and consistent evaluation, the proposed solutions are tested against the Canterbury Corpus, a well-established benchmark for lossless data compression.

Additionally, various file formats, including text, images, video, and audio, are analysed. Unlike other traditional algorithms such as pack and compress, the Data Extraction (DE) method aims to optimise redundancy removal using flag-based segmentation without relying on probabilistic models. The chapter explains in detail the implementation of the three proposed solutions covered in Section 5.2.2:

Flag Information (FI) - Solution I: Focuses on segmenting data by inserting flags at specific intervals, enabling effective compression.

Flag to Flag (F2F) - Solution II: Examines an alternative approach where flags are placed sequentially based on segment value, eliminating the need for explicit flag information storage. Based on the promising results produced in this solution, additional tests were conducted to evaluate its performance in terms of compression ratios, memory usage, and computational performance. These tests were necessary to better understand how the sequential flagging system affects the overall efficiency, particularly under varying data sizes and segment configurations.

Flag to Flag with Flag Information (Solution III): A solution combining aspects of the first two approaches.

This chapter aims to demonstrate the effectiveness of the DE algorithm in comparison to widely used lossless compression methods.

7.2. Flag order system for Data Extraction using MANS

This section provides a detailed explanation of the design and implementation of the Different Flag Systems Solutions compression algorithm, utilising the MANS numeral system. The tests were conducted on a macOS 14.6.1 (23G93) system. Commercial software was not used in this study; instead, standard implementations of gzip, bzip2, and Pack algorithms were utilised from the Canterbury Corpus results [74]. The algorithms for the proposed solutions in this section for Data extraction were developed entirely in Java, leveraging core libraries such as java.io.File, java.util.Arrays, and java.math.BigInteger. The implementation code for all the proposed solutions is included in Appendix A for reference.

The primary focus is on evaluating the performance of the Data Extraction (DE) algorithm. The comparison is conducted using published results from the Canterbury Corpus [74], a well-established benchmark for compression algorithm performance. Compression and decompression times, memory usage, and compression ratios for text files from the corpus were compared directly to these published results for algorithms such as gzip, bzip2, Pack, and others. Multimedia files (images, videos, and audio) were exclusively tested using the DE method, as other lossless algorithms are generally ineffective at further compressing these already-compressed files [75].

The use of published results from the Canterbury Corpus ensures consistency and comparability with previous studies, avoiding the need to re-implement other algorithms. This approach aligns with the study's focus on assessing the novelty and performance of the DE-F2F algorithm without diverting attention to recreating well-documented and widely-used algorithms.

7.2.1. Flag Information (FI) - Solution I:

The solutions to the flagging issue presented in Section 5.2.2 were tested on multiple files where an object-oriented programming language and software platform application (JAVA) were used to implement Solution I. The application processes a string of MANS data alongside the selection of *C*, which varies in length. Various lengths of C_l have been evaluated across different file sizes with flags denoting each processed segment positioned after *C*. Results from Data Extraction tests exhibit similarities across diverse file types as the initial data undergoes conversion to MANS prior to information processing. MANS transforms input data by inserting switches indicated by {1} for every transition from {1} to {0} and vice versa, while also converting {1} to {0} and retaining {0} unchanged. This ensures consistency in data handling during the Data Extraction process.

Table 7.1 illustrates the bit lengths of C_l which are utilised to flag the segments. When designing applications, the number of bits required to flag each segment is constrained by the count of {0} in the converted MANS string to prevent ambiguity in the bit count. After processing each segment of the input data string, *C* is removed upon the flag placement. Each flag incurs a cost of two bits, with a minimum cost of one bit for the flag information. Consequently, the data reduction commences once C_l reaches the length of at least four bits, enabling compression of maximum one bit of information per segment. Due to the alteration of input data through MANS application and after the insertion of C_l during the Data Extraction process, the data is segmented with flag insertion, consequently altering the redundancies within the input data. Therefore, Data Extraction (DE) using MANS has been evaluated with various file types, including those that are already compressed, such as Joint Photographic Experts Group (JPEG) files.

The Canterbury Corpus [74], which is a benchmark to enable researchers to evaluate lossless compression methods, has been used to test Data Extraction using MANS methods. The results of the tests are shown in Appendix B. Additionally, a range of file formats known as U.txt, which contains text, an image file called Garden.jpeg, a video file called Clip.mp4 and a sound file called DE.m4a have been used for the tests.

The files have been converted to Base 64 and then to a binary string. The string has been calculated and converted to MANS using the application designed to convert binary string to MANS, available in Appendix A.1. The Data Extraction application has been applied where the last encoded segment has been used to decode the full message. Finally, a string, called FlagsInfo, has been created to store the Flag Information (FI) for each segment, as proposed in Solution I in Section 5.2. The encoding and decoding codes and results of the application have been listed in Appendix A.2. The test results for (U.txt, Garden.jpeg, Clip.mp4 and DE.m4a) with variation of C_l are shown in Table 7.1, while the Canterbury Corpus results are listed in Appendix B.

Table 7.1: Tests results using Solution I.

C_l/Bits	Name	Type	File size/Bits	Number of switches/Bits	MANS/Bits	Number of processed segments	Data Extraction/Bits	FI/Bits	Total/Bits
4	U.txt	Text	303	157	460	208	44	401	445
5	U.txt	Text	303	157	460	131	67	348	415
6	U.txt	Text	303	157	460	85	120	307	427
7	U.txt	Text	303	157	460	46	230	195	425
8	U.txt	Text	303	157	460	10	400	42	442
4	Garden.jpeg	Image	1,048,472	517,654	1,566,126	783,047	34	1,595,227	1,595,261
5	Garden.jpeg	Image	1,048,472	517,654	1,566,126	522,024	57	1,439,795	1,439,852
6	Garden.jpeg	Image	1,048,472	517,654	1,566,126	391,505	110	1,552,741	1,552,851
7	Garden.jpeg	Image	1,048,472	517,654	1,566,126	313,183	216	1,894,225	1,894,441
8	Garden.jpeg	Image	1,048,472	517,654	1,566,126	260,951	426	2,524,306	2,524,732
4	Clip.mp4	Video	1,858,757	917,658	2,776,415	1,388,186	45	2,825,642	2,825,687
5	Clip.mp4	Video	1,858,757	917,658	2,776,415	925,451	65	2,555,584	2,555,649
6	Clip.mp4	Video	1,858,757	917,658	2,776,415	694,074	123	2,750,972	2,751,095
7	Clip.mp4	Video	1,858,757	917,658	2,776,415	555,239	225	3,361,418	3,361,643
8	Clip.mp4	Video	1,858,757	917,658	2,776,415	462,664	437	4,478,561	4,478,998
4	DE.m4a	Sound	669,797	326,000	995,797	497,876	47	1,009,663	1,009,710
5	DE.m4a	Sound	669,797	326,000	995,797	331,909	73	915,607	915,680
6	DE.m4a	Sound	669,797	326,000	995,797	248,919	125	983,086	983,211
7	DE.m4a	Sound	669,797	326,000	995,797	199,113	237	1,200,537	1,200,774
8	DE.m4a	Sound	669,797	326,000	995,797	165,892	451	1,600,458	1,600,909

The tests, illustrated in Table 7.1 above, show that Data Extraction using MANS has successfully reduced the file size of U.txt from 303 bits to 44 bits for the text file and from 1,048,472 bits to 34 bits for the image file, 1858757 bits to 45 bits for the video file, and from 669797 bits to 47 bits for the sound file when C_l = 4. However, the flag information string increased to ~32-52% compared with the initial file size. It decreased as the length of C_l increased. When C = 5, the flag information string increased to ~14-37% compared with the initial file size, generating better results when compared with the case when C_l = 4. The flag information string starts to increase when C_l = 6, 7 and 8. Since C varies in value for each segment, the flag location can be placed at the beginning or end of the segment. The lower the length of C_l, the more bits are generated for the flag information to identify the flag location.

The compression for each segment will start when $C_l \geq 4$ bits to allow one bit to be compressed for each segment as the cost of each flag for a segment is two bits, and the flag information will generate at least one bit. The results in Table 7.1 are also represented in Figures 7.1, 7.2, 7.3 and 7.4 with the parameters defined as follows:

1. Data Extraction/Bits: the final encoded segment that is required to decode the information
2. Flag info/Bits: the flag location for each flag when decoding the Data extraction
3. DE & Flag info/Bits: both the total number of bits of the generated Data Extraction and the flag information that are required to decode information
4. Number of Processed segments: the total number of segments processed to generate the final String of Data Extraction
5. File size/Bits: the original file size
6. MANS/Bits: the encoded original file using the Modified Adaptive Numeral System
7. The length of C/Bits (C_l): the number of bits selected to compress each segment.

Figure 7.1: Solution I - Data Extraction using flag information string (U.txt)

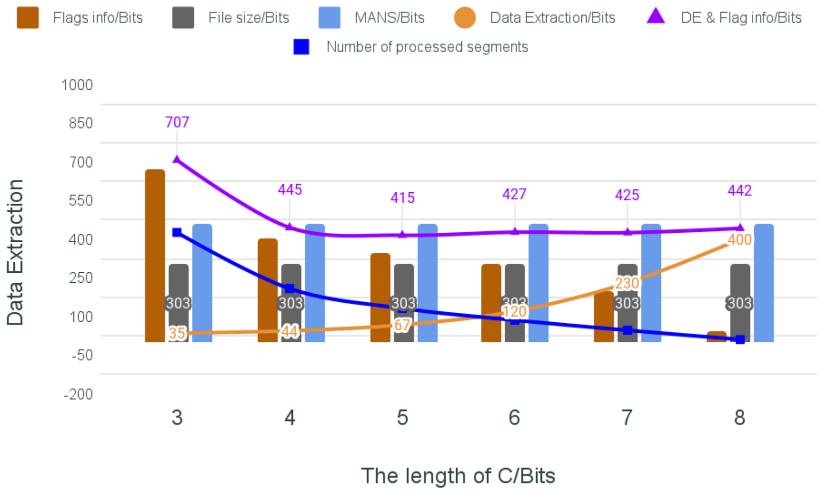


Figure 7.1 represents a small size text file which shows that when C_l is low, the last segment length that Data Extraction generates is low, while the number of bits generated for the flag information is the highest. The last segment length of Data Extraction increases exponentially while the flag information decreases when C_l increases.

Figure 7.2: Solution I - Data Extraction using flag information string (Garden.jpeg)

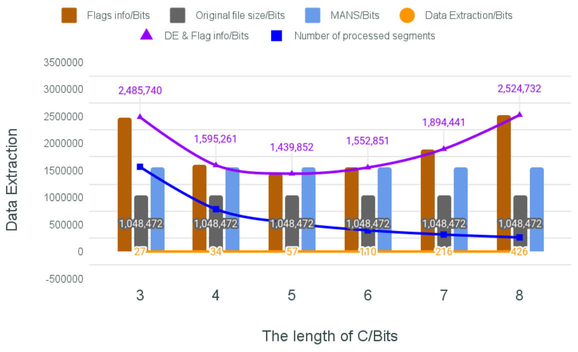


Figure 7.3: Solution I - Data Extraction using flag information string (Clip.mp4)

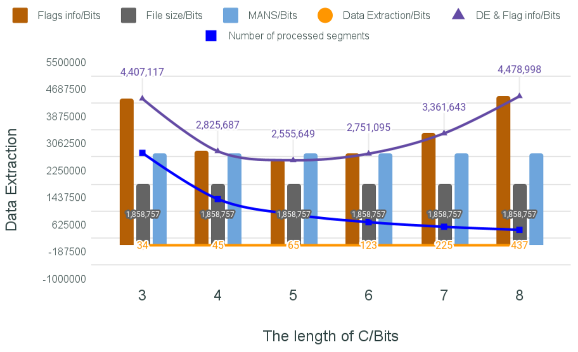
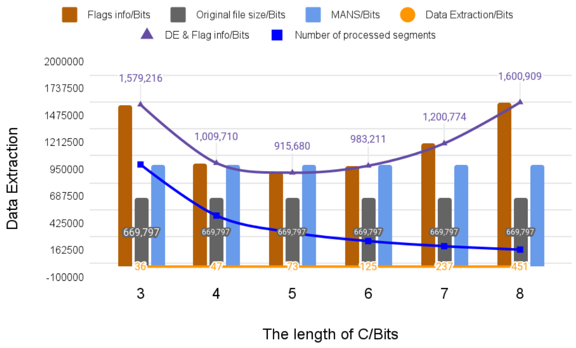


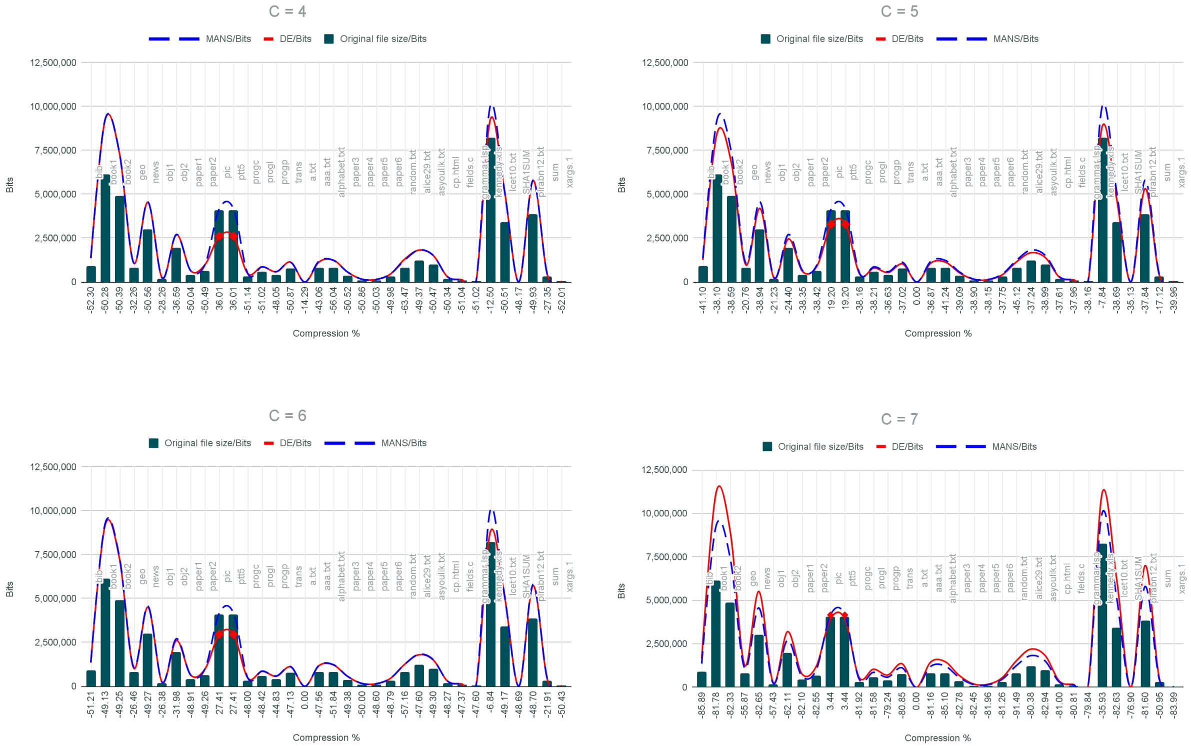
Figure 7.4: Solution I - Data Extraction using flag information string (DE.m4a)



The larger files in Figures 7.2, 7.3 and 7.4, which represent image, video and sound files, respectively and the Canterbury Corpus results in Appendix C, also show that when the length of C_l increases, the segment length increases accordingly, and, as Data Extraction uses the last processed segment to decode information, the number of bits to transmit the last segment will be equal to the value of C of the last segment. C_l increases with each bit and results in fewer flags being placed, fewer segments being processed, and more bits being compressed for each segment. However, the flag information shows an exponential decrease when $3 \leq C_l \leq 5$ and exponential increase when $C_l \ \geq 5$ bits. This is due to multiple flag locations not being at the beginning of the string from the (LSB) this will lead to data expansion when multiple flags are not in order as each flag not in order will cost 1 bit of information. This happens when the highest flag number occurs first from the LSB to MSB and the flag information stores $\{1\}$. Otherwise, it stores $\{0\}$ for each flag that does not correspond with the flag, which is related to the segment, leading to data expansion. An example of the flag information is available in the output section of Appendix A.1. However, when the length of C decreases, the number of segments to be processed increases but fewer bits need to be transmitted for the last segment. Here the flag location is significant and depends on the value of *C*.

Figure 7.5 shows MANS and Data Extraction results compared with the original files of the Canterbury Corpus tests (excluding the large files for visual purposes), where variations from 4 to 7 of the length of *C* have been applied.

Figure 7.5: Solution I - Data Extraction using flag information string (Canterbury Corpus using C = 4, 5, 6 and 7)



The Figures above show that compression occurred for two files, pic and ptt5. The compression percentage varied from ~3.4% to ~36%. The highest compression occurs when $C_l = 4$. Note the compression occurred when the conversion to MANS from the original file was minimal, approximately only 2% higher than the original file size. This is attributed to the low number of switches from {1} to {0} and vice versa in the data string of the original file, which consequently affect the number of bits generated in the flag information. To conclude, with consideration to the selected length of *C*, the lower the number of switches in the original file, the less information is generated in the flag information string and the better the results obtained from the Data Extraction using MANS.

The initial proposed solution in Section 5.2.2 necessitates both the last processed segment and the flag information. The total number of bits required to decode this information is less than the original file size for only pic and ptt5 files, while other file types experience data expansion across the length of C ranging from 4 to 7. This constraint limits the applicability of Data Extraction using MANS for the proposed Flag Information solution to black and white image file types, where it is most effective, particularly when the data to be compressed contains a high number of repeated sequences of 1s or 0s.

7.2.1.1. Flag Information - Analysis and future development

The Data Extraction compression method using MANS opens new perspectives for lossless data compression that can be explored further. After analysing the test results, it became evident that the variations in outcomes were influenced by the method which had been used to determine flag locations. This highlights the need for further analysis of flag location dependencies. These dependencies include:

1. C_x = The value of C for each segment
2. C_l = The length of C
3. $\mathrm{Sg}_l = 2^{\mathrm{Cl}}$ = The maximum segment length
4. F_l = The flag length
5. $F_{sgx} = C_x$ = The flag location for each segment
6. Sg_l/F_l = The number of flags the segment can hold

Since the segment length depends on the value of C, employing a small value for C_l restricts the location of flags to 2^{Cl} and results in high compression. However, when Solution I is used, each segment can accommodate multiple flags, and one of these flags, F_x, will pertain to the processed segment Sg_x.

The maximum number of flags the segment can hold is $2^{\mathrm{Cl}}/F_l$. Hence Flag Information (Solution I) resulted in high compression at the cost of a large number of bits for flag information. If all flags for all segments are assumed to be the first flag from the LSB, then the number of bits that the flag information will hold (FI), will equal the number of processed segments Sg_{Total} with the reservation of the segment length counts to flag it:

$$FI = Sg_{Total} = (MANS\ length\ - Sg_l) \times (\frac{F_l}{C_l}) \qquad (7.1)$$

If this assumption holds true, it implies that for a very small file size, C_l values of 4, 5, 6 and 7 depicted in Figure 7.1 - 7.4 will lead to compression. This is due to the reduced data generated by the processing segments in Data Extraction compared to the original file. Conversely, for large file sizes, all tests with $C_l > 3$ are anticipated to yield a notable compression percentage, ranging from ~25 to 75%. Therefore, if two bits are allocated to identify the flags for each segment, the compression percentage will range from ~12.5 to 37.5% for large files, and only $C_l = 6$ will yield slight compression for very small files. However, allocating additional bits to the flag information will decrease compression results accordingly. Hence, it is essential to limit the flag information used to identify the flag location for each segment to at least $FI \leq 2$, which can be increased as C_l increases, ensuring the method's effectiveness for a small file size to be at least 2^{Cl}. To reduce the amount of flag information, one approach is to modify the flagging system. In Solution I, as indicated by Table 7.2, segments with smaller C_l values exhibit a higher frequency occurrence of the first flag from the least significant bit (LSB). Therefore, rather than relying solely on flag information and prefixing data to the beginning of the string, implementing a probability module can effectively reduce the amount of information stored in the Flag Information (FI).

Table 7.2: Solution I - the occurrences of flags distance from *C* for U.txt

Flag location from LSB	Occurrences					
	$C_l = 3$	$C_l = 4$	$C_l = 5$	$C_l = 6$	$C_l = 7$	$C_l = 8$
1	286	110	51	27	11	3
2	67	45	32	15	6	2
3	47	28	14	7	10	1
4	21	17	13	7	5	0
5	4	5	8	8	6	1
6	0	4	6	7	1	3
7	1	0	4	8	0	0
8	0	0	1	4	0	0
9		0	3	1	3	1
10		0	0	1	2	0
11		0	0	0	2	0
12		0	0	1	0	0
13		0	0	0	0	0
14		0	0	0	0	0
15		0	0	0	1	0
16		0	0	0	1	0
⋮			⋮	⋮	⋮	⋮

As the flag location is dependent on the segment length and the segment length on *C*, Table 7.3 below shows that the possible compression yield per segment for the length of C_l ranges from 2 to 8. Since the segment length increases with the increased length of C_l, the number of bits in each segment will affect the total number of bits to send when Solution II is used. Therefore, the segment length can be explored by increasing C_l and reducing the segment length. Studies on the results obtained show that by increasing C_l to 8 bits and dividing it into two sections, each section will have the size of 4 bits, which will generate one flag per section. The total number of bits for both flags will be 4 bits, while compression will yield 4 bits per segment.

Table 7.3: Possible compression yield per segment.

C_l (bits)	C_x	$Sg_{l(MAX)}$ (bits)	F_l(bits)	F_{Sg_x}	Max Sg_l/F_l	Compression bits per segment
2	0 to 3	4	2	0 to 3	2	0
3	0 to 7	8	2	0 to 7	4	1
4	0 to 15	16	2	0 to 15	8	2
5	0 to 31	32	2	0 to 31	16	3
6	0 to 63	64	2	0 to 63	32	4
7	0 to 127	128	2	0 to 127	64	5
8	0 to 255	256	2	0 to 255	128	6
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Furthermore, considering each section will have C_l = 4, the segment length will be constrained to 16, 0's. allowing for the processing of 8 bits for each segment. Given that the flag order for each segment and section will vary depending on the value of *C*, 1 bit can be allocated in the flag information to store the flag order, as per the aforementioned assumption. Consequently, the cost for each segment will amount to 5 bits (4 for the flags and 1 for identifying the flag order). Affording 3 bits of compression for each segment. This introduces new challenges that warrant further exploration.

7.3. Flag to Flag (F2F) - Solution II:

This section provides a detailed explanation of the design of the DE-F2F algorithm, which leverages the MANS numeral system to achieve effective compression. It also includes an analysis of computational complexity and performance. The proposed DE-F2F is compared to traditional algorithms such as pack, compress, and others. The evaluation encompasses text files from the Canterbury Corpus and multimedia files (images, videos, and audio), with a focus on compression and decompression times, memory usage, and effectiveness across various data types. Unlike text files, multimedia files (images, videos, and audio) present a unique challenge in lossless compression. These files are often already compressed using specialised codecs like JPEG, MP3, or MP4, which are optimised for their respective data formats. This inherent compression limits the ability of traditional lossless algorithms, such as pack and compress, to achieve further significant compression. However, the DE F2F using the MANS algorithm was uniquely designed to explore potential redundancy in a non-probabilistic manner, even in pre-compressed files. As a result, multimedia files were tested exclusively using DE with MANS. Unlike other lossless compression methods, DE does not rely on any probabilistic model, as most traditional lossless compressors are generally ineffective at further compressing these already-compressed files without risking loss of information [75].

The results of this specialised analysis are presented to demonstrate the potential of DE MANS in scenarios where conventional algorithms fall short. This selective testing is justified by the primary aim of showcasing DE F2F's ability to uncover latent patterns and redundancies that traditional methods fail to exploit. By focusing exclusively on DE F2F for multimedia files, the analysis emphasises its capability to address unique challenges in compression, thereby avoiding redundancy in traditional methods.

Data Extraction Flag to Flag (DE-FTF) (see Section 5.2.2), has been designed and developed to test Solution II, (for more detail, see Appendix A.3). The flags are here placed after the last detected flag on the basis of the value of *C*. This solution ensures that the last flag always relates to the segment, and no flag information is required. Since the flags are placed after each other on the basis of *C* value (C_x), the length of each processed segment Sg_x becomes expanded by $Sg_x = C_x - C_l$. The total number of bits that can be compressed, i.e. (k), is defined by the number of processed segments and the length of $C - F_l$ with the reservation of the segment length counts to flag it. This can be given as

$$k = Sg_{total} \times (C_l - F_l) \qquad (7.2)$$

As expected in Section 5.2.2, tests show an increase in the length of each processed segment and fewer segments to process, as shown in Table 7.4. For example, when C_l = 4, Data Extraction resulted in an increase of ~22% to 26% compared to the original file.

Table 7.4: Test results using Solution II.

C_l/Bits	Name	Type	File size/Bits	Number of switches/Bits	MANS/Bits	Number of processed segments	Data Extraction/Bits	Total/Bits
3	U.txt	Text	303	157	460	66	395	395
4	U.txt	Text	303	157	460	43	372	372
5	U.txt	Text	303	157	460	41	400	400
6	U.txt	Text	303	157	460	11	416	416
7	U.txt	Text	303	157	460	5	445	445
8	U.txt	Text	303	157	460	0	460	460
3	Garden.jpeg	Image	1,048,472	517,654	1,566,126	114,711	1,326,706	1,326,706
4	Garden.jpeg	Image	1,048,472	517,654	1,566,126	128,290	1,288,308	1,288,308
5	Garden.jpeg	Image	1,048,472	517,654	1,566,126	145,586	1,330,656	1,330,656
6	Garden.jpeg	Image	1,048,472	517,654	1,566,126	162,158	1,395,646	1,395,646
7	Garden.jpeg	Image	1,048,472	517,654	1,566,126	22,375	1,454,256	1,454,256
8	Garden.jpeg	Image	1,048,472	517,654	1,566,126	11,490	1,497,192	1,497,192
3	Clip.mp4	Video	1,858,757	917,658	2,776,415	430,528	2,345,888	2,345,888
4	Clip.mp4	Video	1,858,757	917,658	2,776,415	252,629	2,271,159	2,271,159
5	Clip.mp4	Video	1,858,757	917,658	2,776,415	144,067	2,344,217	2,344,217
6	Clip.mp4	Video	1,858,757	917,658	2,776,415	78,815	2,461,159	2,461,159
7	Clip.mp4	Video	1,858,757	917,658	2,776,415	40,095	2,575,945	2,575,945
8	Clip.mp4	Video	1,858,757	917,658	2,776,415	20,405	2,653,991	2,653,991
3	DE.m4a	Sound	669,797	326,000	995,797	154,022	841,776	841,776
4	DE.m4a	Sound	669,797	326,000	995,797	89,831	816,137	816,137
5	DE.m4a	Sound	669,797	326,000	995,797	50,792	843,424	843,424
6	DE.m4a	Sound	669,797	326,000	995,797	27,840	884,441	884,441
7	DE.m4a	Sound	669,797	326,000	995,797	14,665	922,477	922,477
8	DE.m4a	Sound	669,797	326,000	995,797	7,540	950,563	950,563

Table 7.4 and Appendix D show the test results of the second solution discussed in Section 5.2. The same files used in the tests for Solution I are used for comparisons and analysis reasons. Results show that Data Extraction using MANS encoded the file size of U.txt from 303 bits to 395 bits for the text file and from 1,048,472 bits to 1,326,706 bits for the image file, 1,858,757 bits to 2,345,888 bits for the video file, and from 669,797 bits to 841,776 bits for the sound file when C_l = 3. On average, the results for C_l = 4 resulted in a ~22% increase compared to the

original file size. However, it dropped to ~50% compared to Solution I, as it decreased from the range between (~46% to 52%) to ~22%.

The results in Table 7.4 are also represented in Figures 7.6, 7.7, 7.8, 7.9 and the Canterbury Corpus tests in Appendix D are also represented in Appendix E with the same parameters defined in the previous section.

Figure 7.6: Solution II Data Extraction results for U.txt

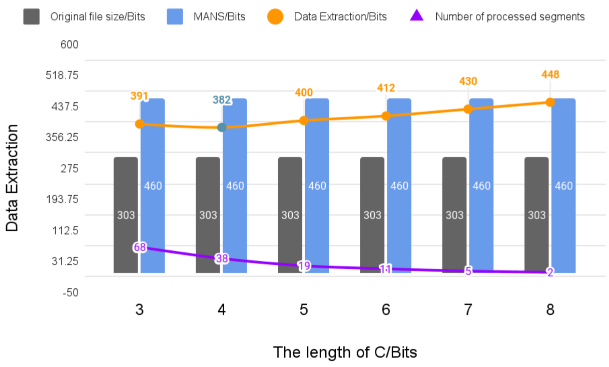


Figure 7.7: Solution II Data Extraction results for Garden.jpeg

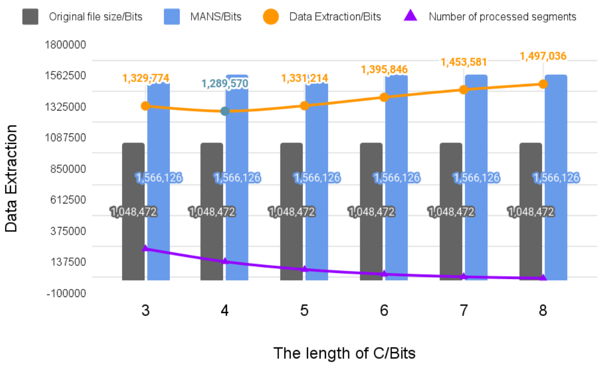


Figure 7.8: Solution II - Data Extraction results for Clip.mp4

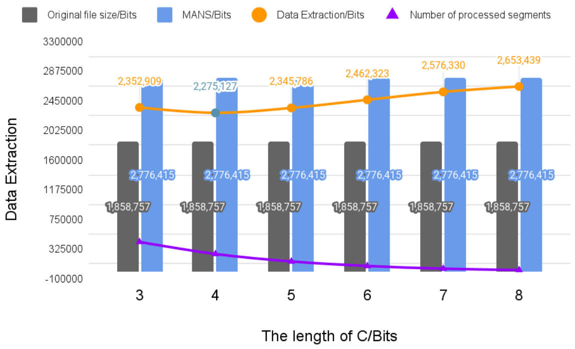
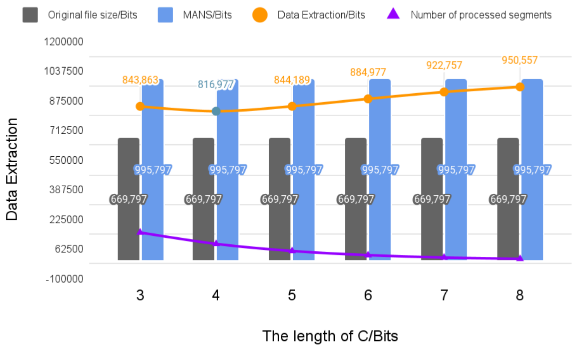


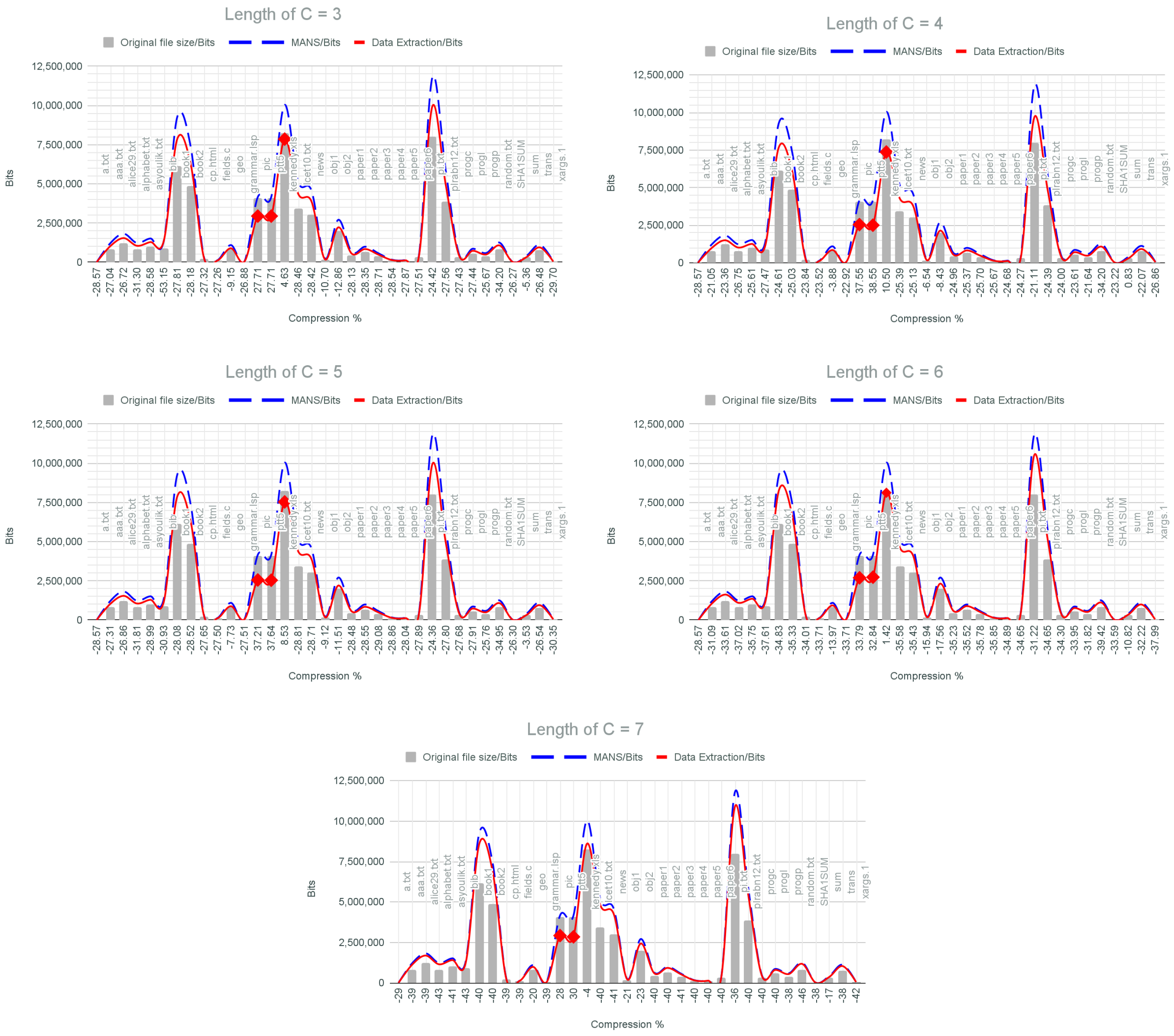
Figure 7.9: Solution II - Data Extraction results for DE.m4a



The Figures above show that all tested files follow the same pattern. The lowest generated results for Data Extraction occur when $C_l = 4$, and they increase slightly when $C_l = 3$. When $C_l > 4$ the results increase gradually, as the flags for each segment are inserted after the previous flag that is related to the previous segment. When $C_l = 3$, the maximum length of the segment, where the flag can be placed, is on the 8th {0} for each segment. At the same time, 3 bits related to C_l are removed from the string and two bits inserted to flag the segment.

This resulted in 1 bit of compression for each segment and multiple flags to be inserted for all segments. When C_l = 4, the maximum length of the segment to place the flag is 16 {0}. The count starts from C_l for the first segment, and then from the last flag that is related to the previous segment. C_l is removed, and 2 bits for the flag are added. Resulted in 2 bits compression per segment. For C_l > 4 the location of the flags in each segment expanded to a maximum of 32 {0}, which increased the size of each segment, resulting in a larger segment at the end of the process. And as the last segment is used to decode the data, compression results were reduced. The larger C_l, the larger the final encoded segment, as shown in Figures 7.6, 7.7, 7.8, 7.9 and Appendix E. MANS and Data Extraction results compared with the original files of the Canterbury Corpus tests (excluding the large files size for visual purposes) are illustrated in Figure 7.10, where variations from 3 to 7 of the length of C have been applied.

Figure 7.10: Data Extraction results for Solution II (Canterbury Corpus using C_l = 3, 4, 5, 6 and 7)



The figures above demonstrate the achieved compression for three files from the Canterbury Corpus tests. Data extraction resulted in reductions of approximately ~38% to 39% for both pic and ptt5 files, and ~11% for Kennedy.xls. The maximum compression is observed for all three files when C_l = 4. This approach, as opposed to Solution I, consistently leads to longer segments, thereby reducing the number of segments for processing and consequently decreasing the amount of data to compress. Nevertheless, it demonstrates superior performance compared to the first solution.

The experiments showcase the results obtained through the application of the Data Extraction technique on the MANS format following its conversion from binary. As a result, the outcomes of the Data Extraction are retained in the MANS format, with the option of reverting to binary format. However, it is crucial to address the positions of flags during the conversion process. Although the flag positions can be eliminated, they can instead be replaced by a non-prefix variable-length code. For example, the Binary string of the file, Binary(U.txt) =

{101010001100101011100110111010001101001011011100110011100100000011000010010000001110100011001010111100
0011101000010000000110011001101001011011000110010100100000011101010111001101101001011011100110011100100
000010011010100000101001110010100110010000000110000101101110011001000010000000100010001000101001011110}
= 303 bits

The MANS-encoded string for the U.txt file when using $C_l = 3$ is: MANS(U.txt) =

{0101010101000100100101010101000100100101000101010001001010100101010010100010010010001001010000000010
0100001010010100000001000101010001001001010101010000100001000101010000101000000100100100100100101010010
1010010100100010010010101010010100000001000101010101010001001001010010101001010100101000100100100100010
0101000000010100100101010101000001010101001000100101010100100100101000000100100001010100101000100100100
1010000101000000101000101000101000101010100101010001 0} = 460 bits

After implementing the Data Extraction method, the resulting output is: DE(MANS(U.txt))=

{001100100010111101001100101001100000100111001001001011010101001110101001010010110010110101101011101011 1
00101000011001011001010101010111100011100101101010010101001111101010010100110101101001001001101001111101
000001101010011100101110101010011001101010101011010001001110101010011100100101000110001110010001101010 1
11001010110010010110100101001100101110000001110100010100011101000101110101110010101000 10}= 395 bits

When converting $DE(MANS(U.txt))$ to binary, the resulting string will be in the form of a non-prefix variable-length code. Therefore, Binary(DE(MANS(U.txt)))=

{00 001110 100 00100 000011 0011001 10100 10110110 001 10 01 01 0010000 001 1101010 111 001 10110100 1011011 10 0110011 100 100000 0100 110 10100 00 01010 011100 10100 11001000 000 11000 010 1101 11001 100100 001 000000 10001000 10001 01 00101110} = 203 bits.

Following the conversion of DE(MANS) to binary, each flag is no longer explicitly present. The presence of flags is indicated by the termination of each non-prefix variable-length code, with a maximum code length of 8 bits for each segment, determined by C_l, which in this case is 3 (since $2^3 = 8$). While this approach effectively compresses the data, it is essential to notice that the non-prefix variable-length code, used in place of flags, necessitates additional steps to decode the information accurately. Therefore, an additional step is necessary to convert the non-prefix variable-length code into a prefix code, while carefully assessing potential overhead, to determine the length of each code (flag locations).

This procedure is essential for ensuring the efficient decoding of the Data Extraction (DE) output. Table 7.5 presents the results obtained from the second solution after converting DE to Binary, using the following equations for compression percentage and compression ratio:

$$\text{Compression Percentage} = (1 - \frac{MANS\ to\ Binary}{File\ size})\times 100 \tag{7.3}$$

$$\text{Compression Ratio} = \frac{File\ size}{MANS\ to\ Binary} \tag{7.4}$$

Table 7.5: Test results using Solution II post-conversion of DE to Binary.

C_l/Bits	Name	Type	File size/Bits	Number of switches/ Bits	MANS/Bits	Number of processed segments	Data Extraction/ Bits	Number of flags	MANS to Binary/Bits (None prefix variable length code)	Compression%	Ratio
3	U.txt	Text	303	157	460	66	395	44	203	33.00%	1.49
4	U.txt	Text	303	157	460	43	372	29	210	30.69%	1.44
5	U.txt	Text	303	157	460	41	400	16	245	19.14%	1.24
6	U.txt	Text	303	157	460	11	416	10	264	12.87%	1.15
7	U.txt	Text	303	157	460	5	445	3	290	4.29%	1.04
8	U.txt	Text	303	157	460	0	460	0	303	0.00%	1.00
3	Garden.jpeg	Image	1,048,472	517,654	1,566,126	114,711	1,326,706	156,430	683786	34.78%	1.53
4	Garden.jpeg	Image	1,048,472	517,654	1,566,126	128,290	1,288,308	97,549	736,402	29.76%	1.42
5	Garden.jpeg	Image	1,048,472	517,654	1,566,126	145,586	1,330,656	60,870	812,895	22.47%	1.29
6	Garden.jpeg	Image	1,048,472	517,654	1,566,126	162,158	1,395,646	36,100	888,757	15.23%	1.18
7	Garden.jpeg	Image	1,048,472	517,654	1,566,126	22,375	1,454,256	20,219	948,092	9.57%	1.11
8	Garden.jpeg	Image	1,048,472	517,654	1,566,126	11,490	1,497,192	10,767	988,912	5.68%	1.06
3	Clip.mp4	Video	1,858,757	917,658	2,776,415	430,528	2,345,888	277,542	1,194,729	35.72%	1.56
4	Clip.mp4	Video	1,858,757	917,658	2,776,415	252,629	2,271,159	175,287	1,281,286	31.07%	1.45
5	Clip.mp4	Video	1,858,757	917,658	2,776,415	144,067	2,344,217	110,816	1,416,319	23.80%	1.31
6	Clip.mp4	Video	1,858,757	917,658	2,776,415	78,815	2,461,159	66,089	1,556,203	16.28%	1.19
7	Clip.mp4	Video	1,858,757	917,658	2,776,415	40,095	2,575,945	35,195	1,678,515	9.70%	1.11
8	Clip.mp4	Video	1,858,757	917,658	2,776,415	20,405	2,653,991	18,252	1,752,983	5.69%	1.06
3	DE.m4a	Sound	669,797	326,000	995,797	154,022	841,776	99,809	433,821	35.23%	1.54
4	DE.m4a	Sound	669,797	326,000	995,797	89,831	816,137	62,574	466,388	30.37%	1.44
5	DE.m4a	Sound	669,797	326,000	995,797	50,792	843,424	39,094	516,044	22.96%	1.30
6	DE.m4a	Sound	669,797	326,000	995,797	27,840	884,441	23439	564,344	15.74%	1.19
7	DE.m4a	Sound	669,797	326,000	995,797	14,665	922,477	13,157	603,368	9.92%	1.11
8	DE.m4a	Sound	669,797	326,000	995,797	7,540	950,563	7,056	630,261	5.90%	1.06

As can be seen in Table 7.5, the conversion from DE to binary involves extracting segments between each flag while discarding the flags, resulting in variable-length codes displayed in a non-prefix format. The findings clearly demonstrate a significant compression compared with the initial file size, ranging from approximately 6% to 36% for *C* values ranging from 3 to 8. Notably, lower *C* values correspond to higher compression rates across all file types.

The conversions from DE to Binary of the MANS results have been also subjected to testing and comparison with the original files from the Canterbury Corpus tests. These comparisons are listed in Appendix D and depicted in Figure 7.11 below.

Figure 7.11: Data Extraction Results in Variable-Length Codes for Solution II (Canterbury Corpus using C_l = 3, 4, 5, 6 and 7)

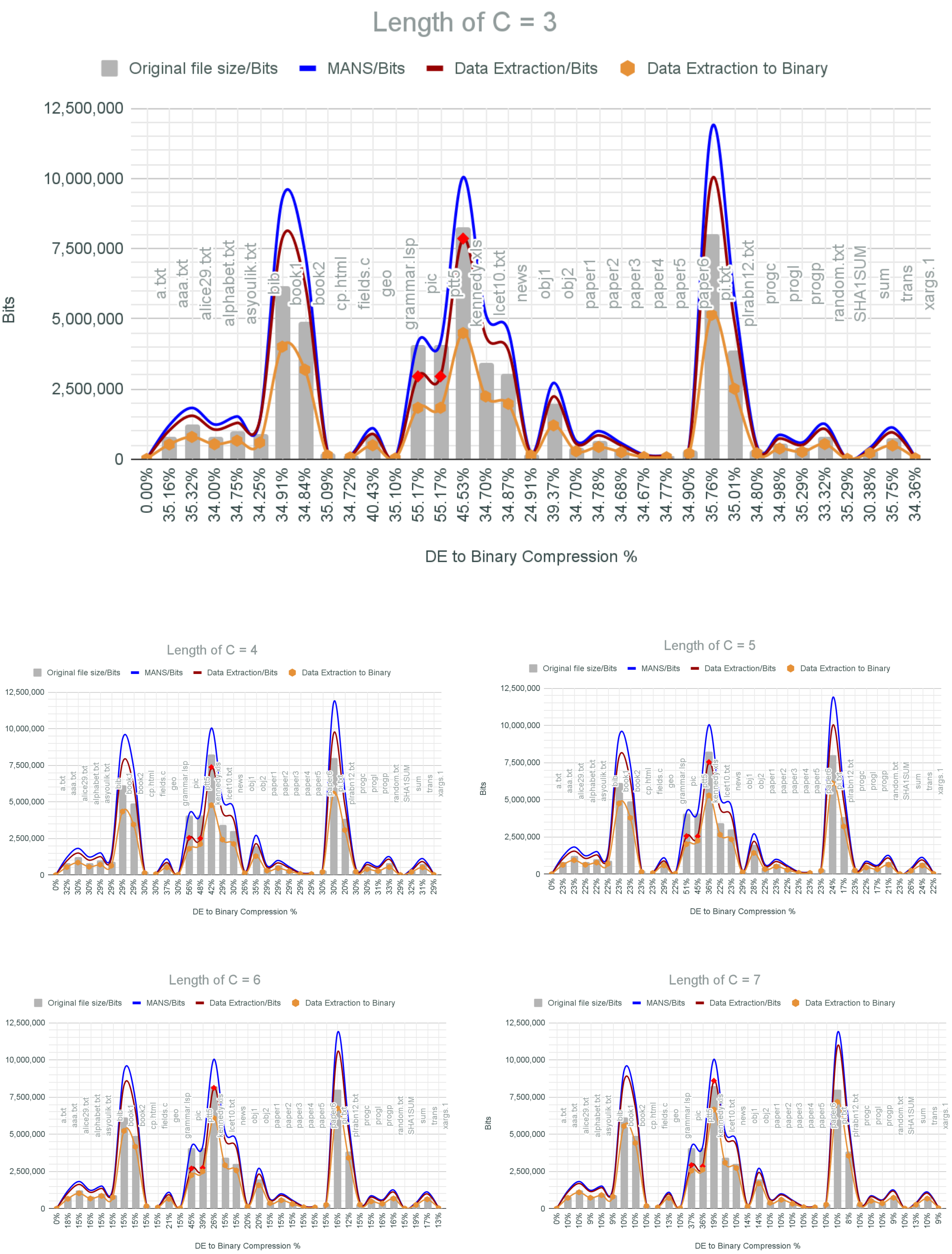


Figure 7.11 illustrates variations in the length of *C*, ranging from 3 to 7. The results demonstrate that compression, in the form of non-prefix variable-length codes, was achieved after converting from DE to Binary, ranging from approximately 25% to 55% when $C_l = 3$.

Figure 7.12: Data Extraction Results in Variable-Length Codes for Solution II (Kennedy.xls file using $C_l = 3, 4, 5, 6$ and 7)

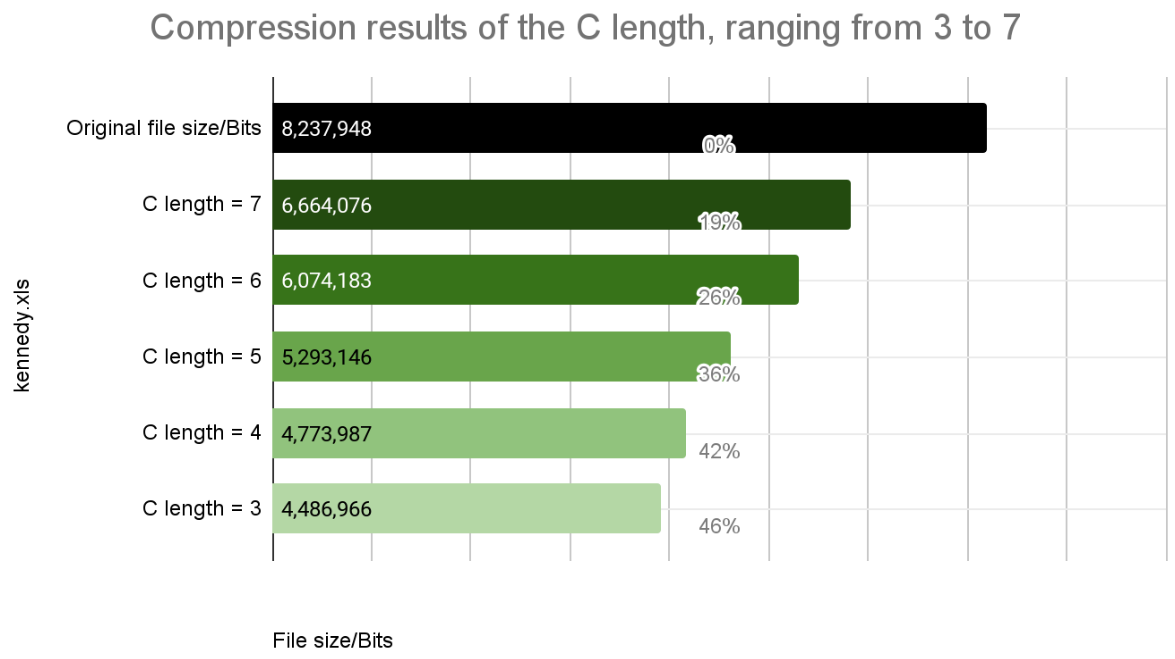


Figure 7.12 offers an additional insight, revealing an increase in compression variation as C_l decreases. This trend stems from a higher volume of processed segments being processed and eliminated, thus contributing to compression and flag generation. Subsequently, the segments between the flags are replaced by non-prefix variable-length codes.

7.3.1. Comparative analysis of Data Extraction - Flag to Flag with other compression methods

In this section, a comparative analysis of the Data Extraction-Flag to Flag (DE-FTF) compression method is conducted with other established compression techniques, including Huffman Coding, Lempel-Ziv-Welch (LZW) Compression, bzip, Gzip, Burrows-Wheeler Transform (BWT), Dynamic Markov Compression (DMC), and Prediction by Partial Matching (PPMC) The objective is to evaluate the effectiveness of DE-FTF in terms of compression ratio, computational complexity, and suitability for various types of data.

1. **Data Extraction-Flag to Flag (DE-FTF):** a binary-based compression method discussed in Section 7.3, is designed for universal application across various file formats and

efficiently compresses data by eliminating it from the beginning of any given binary string and replacing it with unique code, referred to as flags. This substitution is facilitated by MANS, as introduced in Chapter 6, which enables the insertion of flags within the dataset. DE-FTF using MANS prior to conversion into binary, demonstrates effective compression outcomes, especially with black and white images or files containing repetitive sequences of 0s or 1s. Upon conversion to binary, this approach yields a non-prefix variable-length code, which subsequently delivers notable compression ratios. However, further measures are required, which involve converting the non-prefix variable-length code into a prefix-code while meticulously evaluating potential overhead to ascertain the length of each code (flag locations).

DE-FTF employs a novel algorithm that leverages flag-based encoding to achieve high compression ratios in a form of none-prefix variable length code. Due to its utilisation of MANS, which structurally alters the data, DE-FTF can be used as an additional compression method on files that have already been compressed. It operates without needing extra memory for data processing. However, it mandates converting the data into MANS format, which in turn necessitates memory resources of up to double the size of the original file. DE-FTF exhibits adaptability across diverse data traits, excelling particularly with data featuring repetitive bit patterns. Its implementation offers flexibility, allowing users to opt for higher compression ratios at the expense of increased processing power, or alternatively, to prioritise lower compression ratios for reduced processing demands. This choice is determined by the length of *C*, as illustrated in Figure 7.12. DE-FTF may be susceptible to errors if the flag detection process fails or if the extracted flags are corrupted. Error-checking mechanisms can enhance its robustness.

2. **Huffman Coding:** an entropy coding technique that assigns variable-length codes to input symbols based on their frequencies. It is known for its simplicity and effectiveness in achieving near-optimal compression by assigning shorter codewords to more frequent symbols in the data. Huffman coding offers good compression ratios and is widely used in various applications due to its simplicity. Storing the Huffman tree and the codewords associated with each symbol requires memory. However, the memory overhead is relatively low compared with more complex algorithms like bzip, Gzip, BWT, PACK,

DMC, and PPMC, which may require additional memory for storing dictionaries, models, or other data structures. Huffman Coding performs well on data with predictable symbol frequencies, making it suitable for text-based data and certain types of structured data. The implementation involves constructing the Huffman tree and encoding/decoding the data based on the tree structure. While the concept is straightforward, the building of the tree and handling edge cases efficiently can add complexity. It is vulnerable to errors if the encoded data is corrupted or if the Huffman tree is lost during transmission. However, error-checking mechanisms can be incorporated to detect and mitigate errors [76][77].

3. **Lempel-Ziv-Welch (LZW) Compression:** is a dictionary-based algorithm that builds a dictionary of frequently occurring substrings in the input data. It achieves compression by replacing repeated substrings with shorter codes from the dictionary. It requires memory to store the compression dictionary which grows dynamically as new patterns are encountered [78]. The memory overhead can increase with the size and complexity of the input data. LZW adapts well to data with recurring patterns, making it effective for text-based data, images, and certain types of structured data and achieves competitive compression ratios. Implementing LZW Compression involves managing the compression dictionary and efficiently encoding/decoding patterns. The dynamic nature of the dictionary adds complexity to the implementation. LZW can be resilient to errors to some extent as long as the compression dictionary remains intact during transmission. However, errors in the dictionary or corruption of encoded data can lead to decoding errors[79][80][81].

4. **bzip:** is a popular data compression program that uses the Burrows-Wheeler Transform (BWT) and the Move-to-Front (MTF) algorithm followed by Huffman coding. bzip is adaptable to various data types and can handle both text-based data and binary files effectively, it achieves excellent compression ratios and is commonly used for compressing large files and archives, especially for text-based data and files with repetitive patterns. It may have higher memory requirements compared to straightforward compression methods like Huffman Coding, especially during the BWT stage, where the transformed data needs to be stored temporarily. Implementing bzip involves integrating multiple compression algorithms such as BWT, RLE, and Huffman Coding. Managing the memory requirements

and optimising the compression process add complexity to the implementation. bzip includes error-checking mechanisms to ensure data integrity during compression and decompression. However, errors in critical components like the BWT stage can affect decompression accuracy [37][82].

5. **Gzip**: is adaptable to various data types and can handle both text-based data and binary files effectively. It is a widely used file compression program that uses the DEFLATE algorithm, which combines LZ77 and Huffman coding. Gzip offers good compression ratios and is commonly used for compressing files on Unix-like operating systems. It may have moderate memory requirements, especially during the LZ77 sliding window compression stage where the history buffer needs to be maintained. Implementing Gzip involves integrating the DEFLATE algorithm, which combines LZ77 compression and Huffman Coding. Managing the sliding window and optimising the compression process can add complexity to the implementation. Similar to bzip, Gzip includes error-checking mechanisms to ensure data integrity during compression and decompression. However, errors in critical components like the LZ77 sliding window can affect decompression accuracy[83][84].

6. **Burrows-Wheeler Transform (BWT):** is a reversible transformation technique that reorders the characters in the input data to improve the compressibility of repetitive patterns. BWT is effective for compressing text data and is commonly used as a preprocessing step in conjunction with other compression algorithms. It is generally effective across a wide range of data types but may not perform as well on highly randomised or uniformly distributed data. It achieves moderate compression ratios by rearranging the input data to expose potential patterns for subsequent compression stages. BWT may have moderate memory requirements, especially during the construction of the Burrows-Wheeler matrix and the move-to-front steps; it offers rapid compression and decompression speeds compared to alternative techniques. In summary, BWT serves as a valuable tool in data compression, prioritising moderate compression ratios and efficient processing, making it suitable for a range of applications in the field [37].

7. **Dynamic Markov Compression (DMC):** a statistical data compression method that models the input data that uses a dynamic Markov model to predict the next symbol. This can lead to high compression ratios, especially for text and similar data with predictable patterns. DMC adapts well to changing input data and achieves good compression ratios for certain types of data. DMC's effectiveness relies heavily on the quality of the model and its ability to capture the underlying patterns in the data. It may perform exceptionally well on certain types of data with strong dependencies between symbols but may struggle with highly random or diverse data. DMC typically requires more memory compared to straightforward algorithms due to the need to store and update the state of the Markov model [71][85].

8. **Prediction by Partial Matching (PPMC):** similar to DMC, PPMC is a statistical compression method that relies on statistical modelling to predict symbols based on previous context and uses adaptive arithmetic coding. It achieves high compression ratios by exploiting patterns in the input data and adapting its encoding strategy based on previous symbols. It may excel in scenarios where long-range dependencies between symbols exist but may struggle with highly random or noisy data. PPMC typically requires more memory compared to straightforward algorithms due to the need to maintain and update the prediction models for different contexts [86][87].

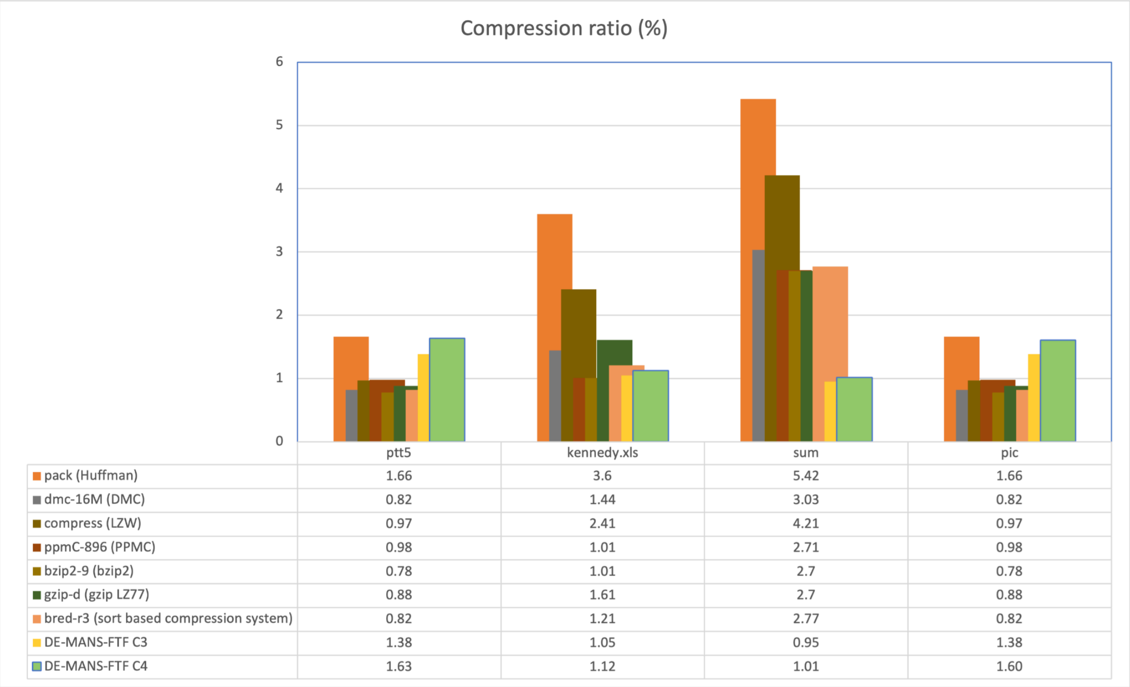
7.3.2. Compression results

The DE-FTF technique, employing a form of MANS before and after binary conversion, was evaluated on files from the Canterbury Corpus [74] and compared against the compression methods discussed, including Huffman, LZW, bzip, Gzip, BWT, DMC and PPMC. The DE using MANS format is entirely decodable. However, the conversion of DE to binary results in a non-prefix code, necessitating additional steps to convert it to a prefix code. Both aspects are thoroughly evaluated in this section.

DE-FTF (MANS):

The results of DE-FTF applied to files in MANS form are detailed in Appendix F, while compression outcomes are illustrated in Figure 7.13.

Figure 7.13: DE-FTF Results in MANS, (ptt5, Kennedy.xls, sum and pic files using $C_l = 3$ and 4)



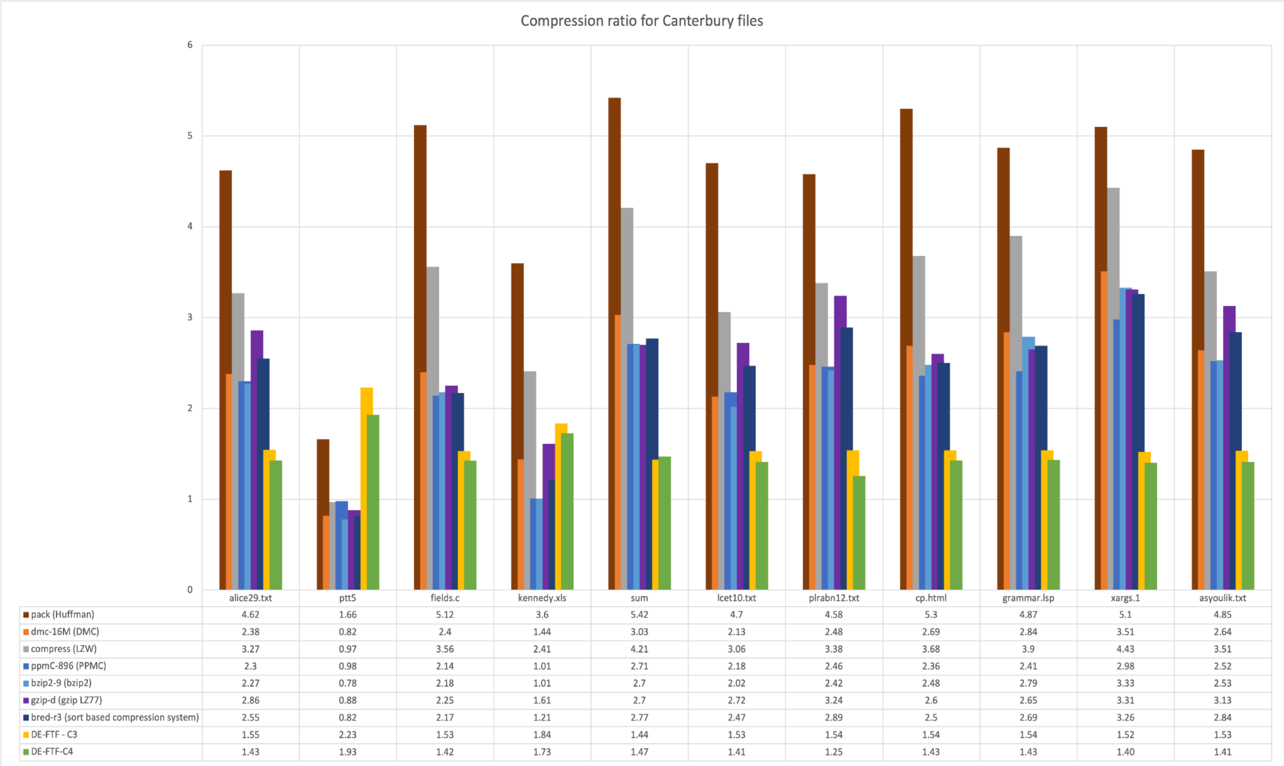
	ptt5	kennedy.xls	sum	pic
■ pack (Huffman)	1.66	3.6	5.42	1.66
■ dmc-16M (DMC)	0.82	1.44	3.03	0.82
■ compress (LZW)	0.97	2.41	4.21	0.97
■ ppmC-896 (PPMC)	0.98	1.01	2.71	0.98
■ bzip2-9 (bzip2)	0.78	1.01	2.7	0.78
■ gzip-d (gzip LZ77)	0.88	1.61	2.7	0.88
■ bred-r3 (sort based compression system)	0.82	1.21	2.77	0.82
■ DE-MANS-FTF C3	1.38	1.05	0.95	1.38
■ DE-MANS-FTF C4	1.63	1.12	1.01	1.60

Figure 7.13 presents the compression outcomes for the files from the Canterbury Corpus tests. When using $C_l = 3$, DE-FTF(MANS) achieved compression ratios of 1.38 for ptt5 and pic files, surpassing most compression methods apart from 1.66 achieved by Huffman. For the kennedy.xls file, DE-FTF(MANS) achieved a compression ratio of 1.12 compared with 3.6 for Huffman. However, it resulted in overhead for all other files. With $C_l = 4$, the results improved significantly, with compression ratios of 1.63, 1.12, 1.01, and 1.6 for ptt5, kennedy.xls, sum, and pic files, respectively, compared to 1.66, 3.6, 5.42, and 1.66 achieved by Huffman.

DE-FTF (MANS to Binary):

The outcomes of applying the Data Extraction technique on data in MANS format post its binary conversion were also examined and juxtaposed with the same methods of compression. This analysis revealed significant compression ratios. However, an additional step is necessary, that is, converting the non-prefix variable-length code into a prefix code, while carefully evaluating potential overhead, to determine the length of each code (flag locations). The data presented in Figures below demonstrates the results of the Canterbury Corpus folders [74] after converting DE-FTF back to binary as a non-prefix variable-length code.

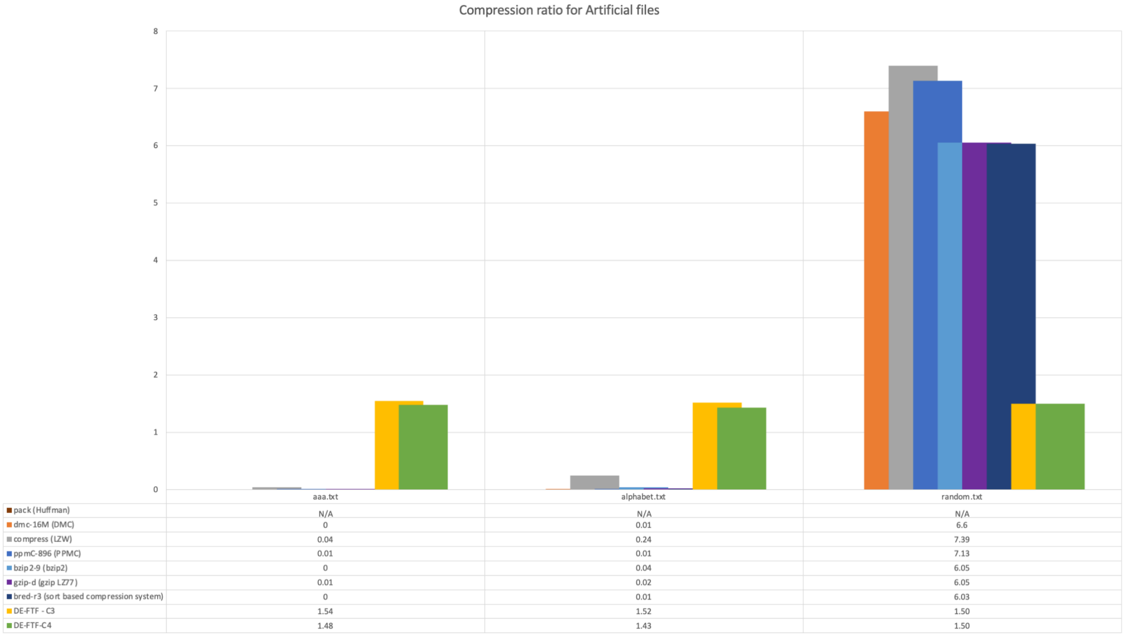
Figure 7.14: DE-FTF Results in MANS, (Canterbury Corpus, files in Canterbury folder, $C_l = 3$ and $C_l = 4$)



	alice29.txt	ptt5	fields.c	kennedy.xls	sum	lcet10.txt	plrabn12.txt	cp.html	grammar.lsp	xargs.1	asyoulik.txt
■ pack (Huffman)	4.62	1.66	5.12	3.6	5.42	4.7	4.58	5.3	4.87	5.1	4.85
■ dmc-16M (DMC)	2.38	0.82	2.4	1.44	3.03	2.13	2.48	2.69	2.84	3.51	2.64
■ compress (LZW)	3.27	0.97	3.56	2.41	4.21	3.06	3.38	3.68	3.9	4.43	3.51
■ ppmC-896 (PPMC)	2.3	0.98	2.14	1.01	2.71	2.18	2.46	2.36	2.41	2.98	2.52
■ bzip2-9 (bzip2)	2.27	0.78	2.18	1.01	2.7	2.02	2.42	2.48	2.79	3.33	2.53
■ gzip-d (gzip LZ77)	2.86	0.88	2.25	1.61	2.7	2.72	3.24	2.6	2.65	3.31	3.13
■ bred-r3 (sort based compression system)	2.55	0.82	2.17	1.21	2.77	2.47	2.89	2.5	2.69	3.26	2.84
■ DE-FTF - C3	1.55	2.23	1.53	1.84	1.44	1.53	1.54	1.54	1.54	1.52	1.53
■ DE-FTF-C4	1.43	1.93	1.42	1.73	1.47	1.41	1.25	1.43	1.43	1.40	1.41

The data depicted in Figure 7.14 illustrates the compression outcomes of the Canterbury folder, following the conversion of DE-FTF back to binary as a non-prefix variable-length code. It outperforms all other compression methods notably in the ptt5 file, achieving a compression ratio of 2.23 with $C_l = 3$, compared to Huffman's ratio of 1.66 and all other methods with a ratio below 1. DE-FTF-C_l = 3 also yields a higher compression ratio 1.84 for the Kennedy.xls file compared to DMC 1.44, PPMC 1.01, bzip 1.01, and gzip 1.61, although it falls short of Huffman's achievement of 3.6. However, for the remaining files, DE-FTT results in lower compression ratios ranging from 1.44 to 1.55 compared to the range of 2.02 to 5.42 achieved by other methods.

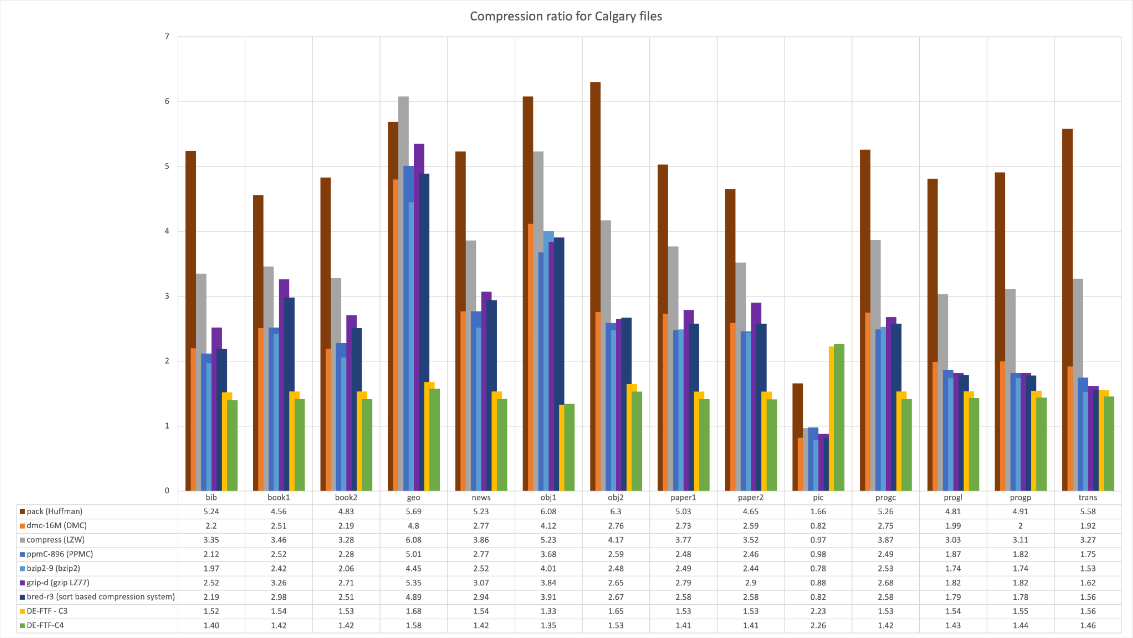
Figure 7.15: DE-FTF Results in MANS, (Canterbury Corpus, files in Artificial folder - $C_l = 3$ and $C_l = 4$)



	aaa.txt	alphabet.txt	random.txt
■ pack (Huffman)	N/A	N/A	N/A
■ dmc-16M (DMC)	0	0.01	6.6
■ compress (LZW)	0.04	0.24	7.39
■ ppmC-896 (PPMC)	0.01	0.01	7.13
■ bzip2-9 (bzip2)	0	0.04	6.05
■ gzip-d (gzip LZ77)	0.01	0.02	6.05
■ bred-r3 (sort based compression system)	0	0.01	6.03
■ DE-FTF - C3	1.54	1.52	1.50
■ DE-FTF-C4	1.48	1.43	1.50

DE-FTF C3 also demonstrated significant compression for the aaa.txt and alphabet.txt files within the Artificial folder, as illustrated in Figure 7.15, achieving compression ratios of 1.54 and 1.52, respectively. In comparison, other compression methods achieved ratios ranging from 0 to 0.24. However, DE-FTF exhibited a lower compression ratio of 1.5 for the random.txt file compared to other methods, which achieved ratios ranging from 6.03 to 7.39.

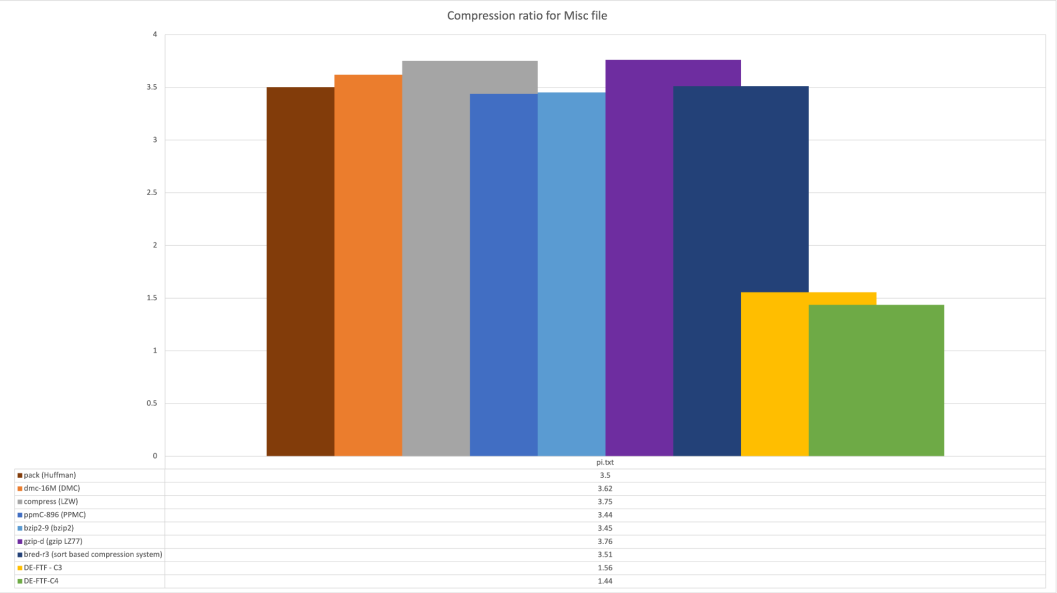
Figure 7.16: DE-FTF Results in MANS, (Canterbury Corpus, files in Calgary folder - $C_l = 3$ and $C_l = 4$)



	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	progl	progp	trans
■ pack (Huffman)	5.24	4.56	4.83	5.69	5.23	6.08	6.3	5.03	4.65	1.66	5.26	4.81	4.91	5.58
■ dmc-16M (DMC)	2.2	2.51	2.19	4.8	2.77	4.12	2.76	2.73	2.59	0.82	2.75	1.99	2	1.92
■ compress (LZW)	3.35	3.46	3.28	6.08	3.86	5.23	4.17	3.77	3.52	0.97	3.87	3.03	3.11	3.27
■ ppmC-896 (PPMC)	2.12	2.52	2.28	5.01	2.77	3.68	2.59	2.48	2.46	0.98	2.49	1.87	1.82	1.75
■ bzip2-9 (bzip2)	1.97	2.42	2.06	4.45	2.52	4.01	2.48	2.49	2.44	0.78	2.53	1.74	1.74	1.53
■ gzip-d (gzip LZ77)	2.52	3.26	2.71	5.35	3.07	3.84	2.65	2.79	2.9	0.88	2.68	1.82	1.82	1.62
■ bred-r3 (sort based compression system)	2.19	2.98	2.51	4.89	2.94	3.91	2.67	2.58	2.58	0.82	2.58	1.79	1.78	1.56
■ DE-FTF - C3	1.52	1.54	1.53	1.68	1.54	1.33	1.65	1.53	1.53	2.23	1.53	1.54	1.55	1.56
■ DE-FTF-C4	1.40	1.42	1.42	1.58	1.42	1.35	1.53	1.41	1.41	2.26	1.42	1.43	1.44	1.46

Figure 7.16 illustrates the performance of DE-FTF on the Pic file within the Calgary folder. It achieved superior compression ratios compared to other compression methods, reaching 2.23 for C_l = 3 and 2.26 for C_l = 4. This outperforms the ratio of 1.66 achieved by Huffman and ratios below 1 for all other methods. However, DE-FTF exhibited less impressive compression results for all other files, ranging from 1.33 to 1.68, compared to other methods with ratios ranging from 1.53 to 6.08.

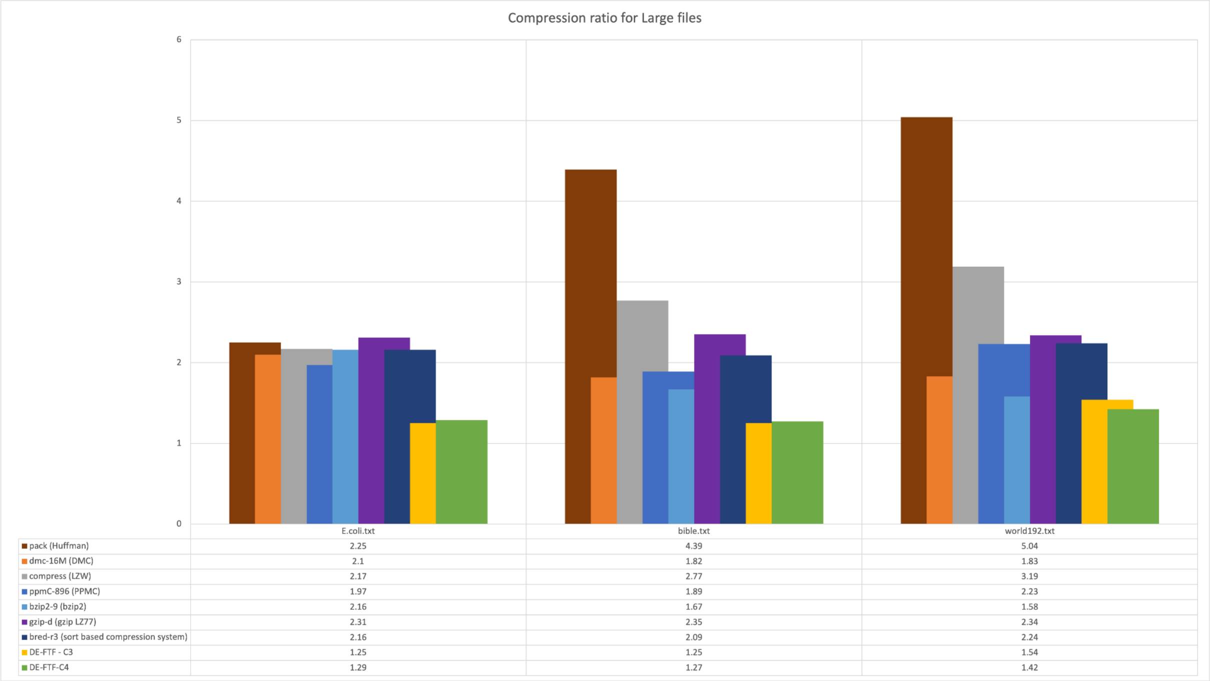
Figure 7.17: DE-FTF Results in MANS, (Canterbury Corpus, file in Misc folder - $C_l = 3$ and $C_l = 4$)



	pi.txt
■ pack (Huffman)	3.5
■ dmc-16M (DMC)	3.62
■ compress (LZW)	3.75
■ ppmC-896 (PPMC)	3.44
■ bzip2-9 (bzip2)	3.45
■ gzip-d (gzip LZ77)	3.76
■ bred-r3 (sort based compression system)	3.51
■ DE-FTF - C3	1.56
■ DE-FTF-C4	1.44

The gzip LZ77 compression method exhibited superior performance compared to all other compression techniques as per Figure 7.17, including DE-FTF, for the Misc file. It achieved an impressive compression ratio of 3.76, surpassing the ratio achieved by DE-FTF, which was 1.56.

Figure 7.18: DE-FTF Results in MANS, (Canterbury Corpus, files in Large folder - $C_l = 3$ and $C_l = 4$)



	E.coli.txt	bible.txt	world192.txt
■ pack (Huffman)	2.25	4.39	5.04
■ dmc-16M (DMC)	2.1	1.82	1.83
■ compress (LZW)	2.17	2.77	3.19
■ ppmC-896 (PPMC)	1.97	1.89	2.23
■ bzip2-9 (bzip2)	2.16	1.67	1.58
■ gzip-d (gzip LZ77)	2.31	2.35	2.34
■ bred-r3 (sort based compression system)	2.16	2.09	2.24
■ DE-FTF - C3	1.25	1.25	1.54
■ DE-FTF-C4	1.29	1.27	1.42

Figure 7.18 illustrates the outcomes for the larger files. DE-FTF demonstrated lower compression ratios compared to other methods, ranging from 1.25 to 1.54, in contrast to the ratios achieved by other methods, which ranged from 1.58 to 5.04.

The data presented in Figures 7.14 to 7.18 demonstrates that upon conversion of DE-FTF back to binary as a non-prefix variable-length code, it resulted in compression across all files, except for one instance involving the file "a.txt" in the Artificial folder, which contains only one letter and where the file size was smaller than C_x, leading to the DE method not being executed. This achievement includes a compression ratio of up to 2.26 for the "pic" file in the Calgary folder Figure 7.16, compared with 1.66 compression for Huffman coding. The lowest compression ratio achieved by DE-FTF is 1.25 for "E.coil.txt" and “bible.txt” files in the Large folder Figure 7.18, in contrast to Huffman, which achieved a compression ratio of 2.25 and 4.39 respectively. On average, DE-FTF achieved a compression ratio of 1.55 with a median of 1.54. These tests have been conducted with various lengths of *C*, ranging from 3 to 7, and the results are presented in Appendix D.

7.3.3. Compression analysis

The results demonstrate that DE-FTF outperforms all current compression methods on specific datasets (e.g., ptt5 and pic). However, while DE using MANS achieves a high compression ratio, traditional methods such as Huffman outperform DE on certain other datasets. This analysis evaluates the performance of DE with MANS and its overall impact on compression efficiency. Table 7.6 summarises the performance of Data Extraction (DE) using the Modified Adaptive Numeral System (MANS) across various datasets highlighting results before and after the application of MANS and binary conversion when C=4.

Table 7.6: Test results using flag to flag.

Original file			**MANS**			**Data Extraction - Flag to Flag- C=4**							
File Name	**Category**	**Original file size/bits**	**Number of switches/bits**	**MANS/bits**	**MANS Overhead %**	**Number of processed segments**	**Data Extraction/ bits**	**Compression in bits**	**Compression %**	**Number of flags (f)**	**DE to Binary**	**Compression after conversion**	**%**
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	50%	115,808	968,383	231,614	19%	78,436	541,008	258,991	32%
alice29.txt	English text	1,216,708	604,973	1,821,681	50%	160,401	1,500,881	320,800	18%	112,261	851,369	365,339	30%
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	54%	108,425	1,013,978	216,788	18%	75,939	560,326	239,673	30%
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	51%	129,202	1,257,857	258,402	17%	91,451	710,026	291,402	29%
bib	Bibliography (refer format)	890,086	473,065	1,363,151	53%	114,299	1,134,555	228,596	17%	81,400	634,308	255,778	29%
book1	Fiction book	6,150,166	3,116,671	9,266,837	51%	801,479	7,663,881	1,602,956	17%	565,357	4,336,553	1,813,613	29%
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	51%	635,253	6,109,871	1,270,504	17%	449,714	3,449,918	1,436,928	29%
cp.html	HTML source	196,822	98,909	295,731	50%	25,994	243,745	51,986	18%	18,207	137,983	58,839	30%
fields.c	C source	89,198	44,627	133,825	50%	11,823	110,181	23,644	18%	8,245	62,629	26,569	30%
geo	Geophysical data	819,199	277,687	1,096,886	34%	122,963	850,962	245,924	22%	78,177	519,193	300,006	37%
grammar.lsp	LISP source	29,766	14,837	44,603	50%	4,008	36,589	8,014	18%	2,790	20,747	9,019	30%
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**2%**	**809,980**	**2,542,664**	**1,619,958**	**39%**	**355,963**	**1,797,920**	**2,273,751**	**56%**
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**2%**	**830,363**	**2,501,898**	**1,660,724**	**40%**	**347,133**	**2,111,405**	**1,960,266**	**48%**
kennedy.xls	**Excel Spreadsheet**	**8,237,948**	**1,802,450**	**10,040,398**	**22%**	**1,333,775**	**7,372,850**	**2,667,548**	**27%**	**768,407**	**4,773,987**	**3,463,961**	**42%**
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	51%	441,538	4,280,695	88,3074	17%	312,456	2,418,554	995,474	29%
news	USENET batch file	3,016,870	1,546,897	4,563,767	51%	394,321	3,775,127	788,640	17%	278,724	2,125,169	891,701	30%
obj1	Object code for VAX	172,028	60,703	232,731	35%	24,726	183,281	49,450	21%	15,648	127,876	44,152	26%
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	37%	284,697	2,140,879	569,392	21%	180,708	1,289,885	684,604	35%
paper1	Technical paper	425,286	216,497	641,783	51%	55,172	531,441	110,342	17%	39,130	300,799	124,487	29%
paper2	Technical paper	657,590	336,453	994,043	51%	84,800	824,445	169,598	17%	60,115	465,871	191,719	29%
paper3	Technical paper	372,206	191,713	563,919	52%	48,029	467,863	96,056	17%	33,972	263,987	108,219	29%
paper4	Technical paper	106,286	54,559	160,845	51%	13,640	133,567	27,278	17%	9,681	75,517	30,769	29%
paper5	Technical paper	95,630	48,451	144,081	51%	12,426	119,231	24,850	17%	8,789	67,582	28,048	29%
paper6	Technical paper	304,838	153,543	458,381	50%	39,774	378,835	79,546	17%	28,075	214,901	89,937	30%
pi.txt	The first million digits of pi	7,999,998	3,800,828	11,800,826	48%	1,056,121	9,688,586	2,112,240	18%	735,451	5,571,236	2,428,762	30%
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	50%	502,386	4,795,079	1,004,770	17%	356,573	3,071,722	783,162	20%
progc	Source code in "C"	316,886	159,267	476,153	50%	41,609	392,937	83,216	17%	29,398	223,146	93,740	30%
progl	Source code in LISP	573,166	288,491	861,657	50%	76,580	708,499	153,158	18%	53,703	400,661	172,505	30%
progp	Source code in PASCAL	395,031	192,299	587,330	49%	53,416	480,500	106,830	18%	36,951	274,037	120,994	31%
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	57%	178,741	1,073,626	178,740	14%	119,244	533,436	266,563	33%
SHA1SUM	SPARC Executable	6,886	3,389	10,275	49%	896	8,485	1,790	17%	623	4,855	2,031	29%
sum	SPARC Executable	305,919	97,635	403,554	32%	50,091	303,374	100,180	25%	30,098	208,101	97,818	32%
trans	Transcript of terminal session	749,559	372,119	1,121,678	50%	103,337	915,006	206,672	18%	70,822	514,103	235,456	31%
xargs.1	GNU manual page	33,814	17,701	51,515	52%	4,310	42,897	8,618	17%	3,126	24,125	9,689	29%
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	50%	4,433,968	39,673,418	8,867,934	18%	3,074,735	25,438,004	6,941,131	21%
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	50%	5,364,511	44,935,257	10,729,020	19%	3,631,592	28,746,307	8,363,212	23%
world.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	50%	2,596,413	24,516,607	5,192,824	17%	1,824,494	13,898,401	5,888,797	30%

Table 7.6 includes the following metrics:

- *File name*
- *Category:* A description of the file type.
- *Original file size (Bits):* The file size before processing
- *Number of switches (Bits):* The number of transitions between {1} and {0} (and vice versa) identified in each file.
- *MANS/Bits:* The total number of bits outputted by MANS after converting the binary representation of each file from binary to MANS.
- *MANS Overhead %:* the overhead percentage of MANS compared to the original file, calculated as:

$$\text{Overhead (\%)} = \frac{(MANS\ Size - Original\ Size)}{Original\ Size} \times 100 \qquad (7.5)$$

- *Number of processed segments:* The number of segments processed by DE.
- *Data Extraction (Bits):* The bit length of the output produced by DE after application.
- *Compression in Bits:* The number of bits reduced relative to MANS.
- Compression %: The compression percentage relative to MAN, calculated as:

$$\text{Compression Percentage} = (1 - \frac{Compressed\ Size}{Original\ Size}) \times 100 \qquad (7.6)$$

- *Number of flags (f):* The number of flags generated during the DE application.
- *DE to Binary:* The total file size (in bits) after converting the DE output to binary.
- Compression %: The percentage of compression achieved after binary conversion, relative to the original file size, calculated as:

$$\text{Compression Percentage} = (1 - \frac{Compressed\ Size}{Original\ Size}) \times 100 \qquad (7.7)$$

The compression performance of the DE process shows significant reduction in the size of all datasets with compression ranging from ~21% to ~56% relative to the original file size. For example, pic file is reduced from 4,071,671 bits to 2,273,751 bits (~56% compression) and bible.txt is reduced from 32,379,135 bits to 25,438,004 bits (~21% compression).

- MANS encoding impact: MANS plays a pivotal role in identifying segment lengths and encoding switches (denoted as W_{Total}) present in the data, as discussed in Section 6.2.1. The overhead introduced by MANS significantly impacts the level of compression that DE can achieve.

- Low Overhead: For datasets like pic in Figure 7.17 and ptt5 in Figure 7.14, where the MANS overhead is ~2%, compression after the DE application is notably high (~48% to ~56%), surpassing traditional methods.
- High Overhead: When the MANS overhead exceeds ~50%, such as in bible.txt (~21% compression) and E.coli (~23% compression), the compression efficiency decreases substantially.

Table 7.7 demonstrates the inverse relationship between MANS overhead and DE compression efficiency:

Table 7.7: Correlation Between MANS Overhead and DE Compression

MANS Overhead	**DE Compression**	**DE Compression post conversion to binary**
2%	39% to 40%	48% to 56%
22%	27%	42%
32%-37%	21% to 25%	26% to 37%
48%-57%	17% to 19%	20% to 33%

- Data Extraction Encoding impact: The analysis shows that DE compression improves when MANS overhead is low but declines as overhead increases. This behaviour arises from the structure of MANS, which transforms input data by inserting switches (indicated by {1}) for every transition from {1} to {0} (and vice versa) while retaining {0} unchanged. The retention of {0} reduces the likelihood of overhead, while the frequent insertion of switches introduces additional bits. files that have low MANS overhead such as ptt5 and kennedy.xls and high MANS overhead such as and plrabn12.txt in Figure 7.14 have been analysed further by calculating the occurrences of {0} and {1}, for example, pic contains a total of {0}= 3,753,964 and {1}=317,707, with a total number of switches (W_{Total}) = 90,951, Consequently, the total number of bits after converting the file to MANS is calculated as:

$$pic(MANS)_l = pic_l + W_{Total} = 4,071,671 + 90,951 = 4,162,622\ bits$$

$$\text{Overhead(\%)}=\frac{(MANS\ Size - Original\ Size)}{Original\ Size}\times 100 = \frac{(4,162,622-4,071,671)}{4,071,671}\times 100 = 2.23\%$$

These results are confirmed in Table 7.7, were further validated through additional tests, highlighting the compression achievable with low versus high MANS overhead:

- A MANS string containing 2,048 bits of continuous {0} was processed. The DE reduced the string length to 1,208 bits (~41% compression).
- For a string with 2,048 bits of alternating {10}, the DE reduced the file size to 1,870 bits (~9% compression).

In contrast, datasets with high MANS overhead, such as bible.txt and E.coli, show limited compression (21% and 23%, respectively) due to the increased number of switches introduced by MANS, as shown in Figure 7.17. This highlights that DE-FTF, unlike traditional methods like Huffman coding, is more sensitive to input characteristics, which can limit its effectiveness in datasets with high MANS overhead. While other tested methods like Huffman coding relies on a frequency-based approach to optimise compression by assigning shorter codes to frequently occurring symbols, DE-FTF uses MANS to process segment lengths and encode transitions between data states. This fundamental difference in approach makes DE-FTF more adaptable to certain dataset structures but also more vulnerable to overhead when data characteristics are less favourable.

As shown in Figure 7.15, DE-FTF significantly outperforms traditional methods for datasets with low MANS overhead, achieving notable compression efficiency. This further validates the impact of input characteristics on DE-FTF's performance.

Overall, these results suggest that while DE-FTF is highly effective for datasets with low MANS overhead, its practical applicability is limited for datasets where the overhead is significant, underscoring the importance of input data characteristics in determining its efficiency.

7.3.4. Quantitative Analysis on Complexity and Space Usage

In this section, I provide a quantitative analysis of the computational complexity and space complexity of the proposed DE compression algorithm (DE F2F using MANS) compared to well-established algorithms such as gzip, bzip2, and others. The analysis focuses on the algorithm's performance in terms of compression time, decompression time, and memory usage. The DE-F2F algorithm was developed in Java, leveraging core libraries such as java.io.File, java.util.Arrays, and java.math.BigInteger. The implementation code for all the proposed solutions is included in Appendix A.3 for reference.

To ensure consistency and comparability with previous studies, the results are compared against published data from the Canterbury Corpus [74]. This use of published results avoids the need to re-implement other algorithms, allowing the study to focus on assessing the novelty and performance of the DE-F2F algorithm. This approach ensures that attention is not diverted to recreating well-documented and widely-used algorithms.

The results were obtained through a series of tests conducted on a variety of text-based files from the Canterbury Corpus. Additionally, this section includes an evaluation of the algorithm's space efficiency by quantifying its memory usage during both the compression and decompression phases.

Computational Complexity Analysis

The computational complexity of a compression algorithm is typically determined by its time complexity during both the compression and decompression stages. The DE compression algorithm was evaluated on the canterbury folder which included text files with varying sizes and contents. The compression time (in seconds) and decompression time were measured. The DE-F2F C4 algorithm leverages the MANS numeral system, offering a novel approach to data compression that prioritises computational efficiency.

Table 7.8 presents a comprehensive comparison between DE-F2F C4 and other established compression algorithms, focusing on computational complexity, speed, and consistency. The analysis uses experimental results derived from the Canterbury Corpus to highlight the strengths and limitations of DE-F2F C4 in diverse scenarios.

Table 7.8: Compression speed results (sorted by increasing average compression time)

Method	DE-FTF C4	pack	compress	gzip-d	bzip2-9	ppmC-896	bred-r3	dmc-16M
alice29.txt	**0.5**	0.27	0.41	1.35	2.52	2.51	4.71	9.49
ptt5	**0.97**	0.2	0.22	0.58	0.83	2.09	1.48	8.73
fields.c	**0.07**	1.14	1.56	1.38	4.79	3.9	5.56	12.06
kennedy.xls	**2.2**	0.21	0.33	1.15	3.47	6.14	7.56	8.84
sum	**0.14**	0.5	0.71	1.54	3.22	6.48	6.08	9.5
lcet10.txt	**1.2**	0.22	0.48	1.33	2.37	2.22	4.57	9.34
plrabn12.txt	**1.39**	0.23	0.46	1.8	2.61	2.28	4.76	9.45
cp.html	**0.12**	0.62	0.97	1.27	3.37	3.94	5.23	10.08
grammar.lsp	**0.03**	2.77	4.32	4.16	8.08	6	9.59	18.43
xargs.1	**0.03**	2.48	3.86	2.88	10.15	7.11	8.8	17.46
asyoulik.txt	**0.44**	0.27	0.48	1.5	2.38	2.61	4.77	9.2
Average	**0.64**	0.81	1.25	1.72	3.98	4.12	5.74	11.14
S.D.	**0.71**	0.94	1.45	0.98	2.75	1.95	2.24	3.48

DE-F2F C4 exhibits the lowest average compression time (0.64 seconds), outperforming traditional and high-compression methods such as:

- pack (0.81 seconds) and compress (1.25 seconds): Conventional algorithms prioritising simplicity and speed.
- bzip2-9 (3.98 seconds) and dmc-16M (11.14 seconds): Algorithms optimised for high compression ratios but with significant computational overhead.

The algorithm demonstrates consistent performance across file sizes, with minimal variance, confirming its suitability for time-critical compression tasks. the Standard Deviation (S.D.) was calculated using the following equation:

$$S.D. = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2} \qquad (7.8)$$

Where x_i are the individual values, $\bar{x}$ is the mean (average) of the values and N is the total number of values. DE-F2F C4 has a Standard Deviation of 0.71, indicating a consistent computational complexity across various file types. Comparatively, high-compression methods, such as dmc-16M (S.D. = 3.48) and bzip2-9 (S.D. = 2.75), exhibit higher variability, underscoring their sensitivity to input file characteristics. For smaller files, such as grammar.lsp and xargs.1, DE-F2F C4 outperforms all algorithms with compression times as low as 0.03 seconds. In contrast, algorithms optimised for compression ratios (e.g., bzip2-9 and ppmC-896) incur significant computational overhead. On larger files, such as plrabn12.txt and lcet10.txt, DE-F2F C4 continues to demonstrate efficiency, maintaining compression times under 1.4 seconds. This consistency highlights its computational scalability compared to slower methods like ppmC-896 (9.45 seconds for plrabn12.txt).

The decompression efficiency of a compression algorithm is critical for real-world applications where fast data retrieval is essential. Table 7.9 presents a comprehensive comparison between DE-F2F C4 and other well-established methods, including gzip-d, bzip2-9, ppmC-896, and dmc-16M, using experimental results derived from the Canterbury Corpus [74].

Table 7.9: De-compression speed results (sorted by increasing average compression time)

Method	**gzip-d**	**compress**	**pack**	**DE-F2F C4**	**bzip2-9**	**bred-r3**	**ppmC-896**	**dmc-16M**
alice29.txt	0.2	0.3	0.38	**0.87**	0.85	1.13	2.98	9.48
ptt5	0.1	0.17	0.16	**2.05**	0.29	0.5	2.58	8.76
fields.c	1.18	1.2	1.02	**0.11**	1.67	1.86	4.47	12.08
kennedy.xls	0.12	0.17	0.28	**3.47**	0.68	2.62	7.47	9
sum	0.44	0.44	0.58	**0.32**	1.08	2.29	7.45	9.46
lcet10.txt	0.16	0.23	0.35	**1.9**	0.98	1.06	2.69	9.53
plrabn12.txt	0.17	0.24	0.34	**2.17**	1.17	1.14	2.69	9.71
cp.html	0.57	0.78	0.79	**0.2**	1.13	1.67	4.64	10.6
grammar.lsp	3.1	3.55	3.32	**0.07**	3.42	3.98	6.51	17.68
xargs.1	2.6	2.6	3.07	**0.75**	2.98	3.78	7.3	16.55
asyoulik.txt	0.22	0.4	0.39	**0.79**	0.88	1.23	3.07	9.42
Average	0.81	0.92	0.97	**1.15**	1.38	1.93	4.71	11.12
S.D.	1.06	1.13	1.13	**1.10**	0.97	1.13	2.09	3.11

Table 7.9 illustrates that DE-F2F C4 exhibited an average decompression time of 1.15 seconds, positioning it between faster, low-overhead methods such as gzip-d (0.81 seconds) and compress (0.92 seconds), and more computationally intensive algorithms like ppmC-896 (4.71 seconds) and dmc-16M (11.12 seconds). While traditional algorithms like pack (0.97 seconds) and compress (0.92 seconds) generally provide faster decompression speeds, they sacrifice compression efficiency for speed. DE-F2F C4, on the other hand, strikes a balance, providing reasonable decompression times without compromising too much on compression effectiveness. For small files such as alice29.txt (0.87 seconds for DE-F2F C4), DE-F2F C4 performed competitively with faster methods like gzip-d (0.2 seconds) and compress (0.3 seconds). However, on larger files like plrabn12.txt (2.17 seconds) and lcet10.txt (1.9 seconds), DE-F2F C4 demonstrated slightly slower decompression times compared to gzip-d (0.17 seconds for plrabn12.txt and 0.16 seconds for lcet10.txt), but still outperformed more complex methods such as dmc-16M (9.71 and 9.53 seconds, respectively).

The standard deviation of 1.10 seconds further underscores the consistency of DE-F2F C4 across different file types, surpassing high-compression methods like dmc-16M (3.11 seconds) that tend to exhibit more variation in decompression times.

Space Complexity Analysis

The space complexity of the DE-F2F algorithm was evaluated based on its memory usage during both the compression and decompression phases. This evaluation involved converting files from the Canterbury Corpus to binary format, encoding each file using the DE-F2F method, and then decoding the files to verify accuracy and assess memory demands. The analysis captures memory requirements in the following key metrics:

- Initial Memory Usage: Memory consumed by the algorithm before the compression process begins.
- Encoding Memory Usage: Peak memory usage during the compression phase.
- Decoding Memory Usage: Peak memory usage during the decompression phase.
- Compression Memory Footprint: Difference between encoding memory usage and initial memory usage.

- Decompression Memory Footprint: Difference between decoding memory usage and initial memory usage.

Memory usage was measured in megabytes (MB) for each file, and the results are summarised in Table 7.10 below:

Table 7.10: Memory usage for DE F2F C=4

Method	Initial (MB)	Encoding (MB)	Decoding (MB)	Compression (MB)	Decompression (MB)
alice29.txt	69	228	156	159	87
ptt5	94	376	270	282	176
fields.c	9	25	30	16	21
kennedy.xls	53	962	619	909	566
sum	4	94	78	90	74
lcet10.txt	89	432	412	343	323
plrabn12.txt	55	598	526	543	471
cp.html	15	38	77	23	62
grammar.lsp	8	16	16	8	8
xargs.1	8	18	20	10	12
asyoulik.txt	46	182	100	136	54
Average	40.91	269.91	209.45	229.00	168.55
S.D.	33.92	301.04	216.29	281.83	196.33

Table 7.10 shows that the average initial memory usage across all files is 40.91 MB, reflecting the base memory footprint of the DE-F2F algorithm. while the average encoding memory usage is 269.91 MB, indicating the peak memory demand during compression. The average decoding memory usage is 209.45 MB, slightly lower than encoding but still substantial. On average, the compression phase consumes 229 MB of additional memory, while decompression requires 168.55 MB beyond the initial memory usage.

The standard deviation for encoding memory usage (301.04 MB) and decoding memory usage (216.29 MB) highlights significant variability depending on file size and complexity. Similarly, the standard deviation of memory used during compression (281.83 MB) and decompression (196.33 MB) reflects variation in algorithmic demands across different datasets. Large files such as kennedy.xls and lcet10.txt, require significantly higher memory during both compression and decompression due to their size and structure. For instance, kennedy.xls exhibits the highest memory usage, with 962 MB during encoding and 619 MB during decoding. While small files such as grammar.lsp and xargs.1, demonstrate minimal memory requirements, with encoding and decoding memory usage ranging between 16 MB and 20 MB. Average files such as alice29.txt and asyoulik.txt show consistent memory usage, making them representative of typical computational scenarios.

The memory requirements for decompression are generally lower than those for compression, with the exception of a few anomalies (e.g., cp.html, where decoding demands slightly higher memory). This behaviour suggests that the MANS-based DE-F2F algorithm optimises memory efficiency during the decompression phase.

The DE-F2F algorithm demonstrates competitive space efficiency, with relatively low initial memory requirements and scalable encoding/decoding performance. However, large files, such as kennedy.xls, can significantly increase memory usage, highlighting the importance of optimising the algorithm for high-memory scenarios. This analysis confirms that DE-F2F C4, with its use of the MANS, offers a viable balance between memory usage and computational speed, making it suitable for various data compression applications.

7.4. Flag to flag with flag information (Solution III):

The third proposed solution, discussed in Section 5.2.2 and detailed in Appendix A.4, integrates elements from the first and second solutions. This method initiates by computing the value of *C* from the beginning of the string, positioning the first flag accordingly. Subsequently, the system calculates the difference between the value of *C* and the number of bits required to reach the previous segment flag (F_{i-1}). If this difference equals or exceeds the number of bits to reach F_{i-1}, the system counts the bits from the start of the segment and positions the flag accordingly.

In this scenario, a transmission of {0} indicates that the value of *C* equals the number of bits from the start of the segment, excluding *C*. Conversely, if the value of *C* is less than the number of bits to reach the previous flag ($C < F_{i-1}$), the system counts the bits equivalent to *C* from F_{i-1} and positions F_i accordingly. Subsequently, a transmission of {1} signifies that the value of *C* equals the number of bits between F_{i-1} and F_i. The reintroduction of flag information serves to identify the flag source, thereby constraining the expansion of data size for each segment under processing, as well as limiting the number of bits to be transmitted. Detailed results can be found in Table 7.11.

Table 7.11: Test results using Solution III.

C_l/Bits	Name	Type	File size/Bits	Number of switches/Bits	MANS/Bits	Number of processed segments	Data Extraction/Bits	Flags info/Bits	Total/Bits
3	U.txt	Text	303	157	460	71	388	10	398
4	U.txt	Text	303	157	460	43	372	7	379
5	U.txt	Text	303	157	460	24	385	2	387
6	U.txt	Text	303	157	460	13	404	9	413
7	U.txt	Text	303	157	460	7	420	2	422
3	Garden.jpeg	Image	1,048,472	517,654	1,566,126	236,190	1,329,935	24	1,329,959
4	Garden.jpeg	Image	1,048,472	517,654	1,566,126	138,186	1,289,752	4	1,289,756
5	Garden.jpeg	Image	1,048,472	517,654	1,566,126	78,319	1,331,166	9	1,331,175
6	Garden.jpeg	Image	1,048,472	517,654	1,566,126	42,638	1,395,570	2	1,395,572
7	Garden.jpeg	Image	1,048,472	517,654	1,566,126	22,511	1,453,566	2	1,453,568
3	Clip.mp4	Video	1,858,757	917,658	2,776,415	423,430	2,352,984	7	2,352,991
4	Clip.mp4	Video	1,858,757	917,658	2,776,415	250,924	2,274,565	4	2,274,569
5	Clip.mp4	Video	1,858,757	917,658	2,776,415	143,671	2,345,399	5	2,345,404
6	Clip.mp4	Video	1,858,757	917,658	2,776,415	78,607	2,461,983	4	2,461,987
7	Clip.mp4	Video	1,858,757	917,658	2,776,415	40,053	2,576,145	4	2,576,149
3	DE.m4a	Sound	669,797	326,000	995,797	151,586	844,210	9	844,219
4	DE.m4a	Sound	669,797	326,000	995,797	89,328	817,139	4	817,143
5	DE.m4a	Sound	669,797	326,000	995,797	50,660	843,814	5	843,819
6	DE.m4a	Sound	669,797	326,000	995,797	27,822	884,505	4	884,509
7	DE.m4a	Sound	669,797	326,000	995,797	14,647	922,557	4	922,561

Table 7.11 presents the results obtained from testing the third solution. It was observed that as the encoder processes the first few segments, the string size exceeds the maximum value of C, which results in the flag always being placed at the end of the string (the first flag from the least significant bit of the string). A {1} flag is used to denote that the flag pertains to the current segment, while {0} flags indicate that the flag is unrelated to the segment, continuing until a {1} flag is encountered. The results in Table 7.11 closely align with those of Solution II in Table 7.5, with variations ranging from 0.0001% to 0.04% when . For example, in the U.txt file, the total number of generated bits is 398, which is fewer than in Solution II before conversion to binary, where the total is 395 bits.

Moreover, the tests reveal that the flag information string results in a repetition of {1} bits. For instance, with $C_l = 3$ and a maximum value of $C_x = 7$, if the first flag location falls on the 7th {0} of the string, and the next segment's C value is x, the flag will be positioned after the previous flag by the value $7 + x$, extending the segment length. The final flag will be situated after counting the {0} bits in the segment equal to $7 + x$. Subsequently, for subsequent segments, where the maximum C value of 7 is consistently smaller than the last flag location, repeated bits occur in the flag information.

Examining the flag information for the tests in Table 7.11 reveals that with $C_l = 3$, the string contains: {00101001001111111111111111111111111111111111111111111111111111111111111}. This string exhibits a repeated sequence of {1}s after processing the 10th segment, as the last flag location follows the 7th {0} of the segment.

Given that each segment compresses a minimum of 3 bits while incurring a cost of 2 bits for the subsequent flag location, it follows that the C value will consistently be smaller than the last flag location. Consequently, if a segment contains multiple flags equal to or exceeding the maximum value of C, denoted as $F_x \geq 2^{Cl}$, then the final flag of the segment pertains to the processed segment.

As a result, the identification of the flag source in the flag information string can cease immediately after reaching this threshold. During decoding, Solution II can be applied when $F_x \geq 2^{Cl}$, whereas Solution I, in conjunction with the flag information, can be utilised when $F_x < 2^{Cl}$. In the aforementioned scenario, the flag information is condensed to 10 bits, as opposed to the original 71 bits, resulting in a net saving of 61 bits. Hence, the third solution effectively employs the first solution until the flag location equals or exceeds the maximum value of C. Subsequently, Solution II is utilised until the string's conclusion. Figure 7.19 visually depict Table 7.11 and underscore the similarities in results compared to Solution II.

Figure 7.19: Solution III, Data Extraction results for U.txt, Garden.jpeg, Clip.mp4 and DE.m4a using solutions one and two

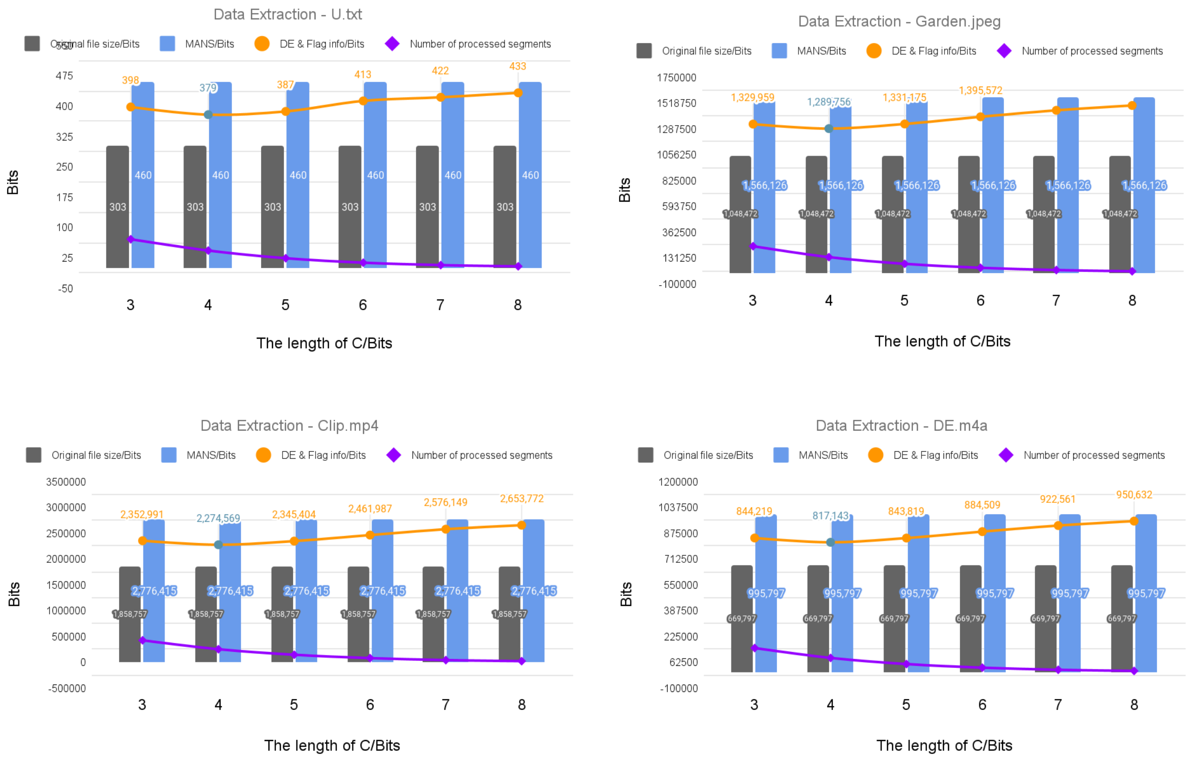


Figure 7.19 demonstrates that the third solution outperforms Solution I and achieves comparable results to Solution II when operating on MANS-formatted data.

Several factors influence compression performance. Solution I (Data Extraction) yielded significant compression results but required a sizable data string to identify flag locations. Solution II avoided using the flag location string but increased segment length. In contrast, the third solution combines aspects of both solutions: it employs Solution I until $F_x \geq 2^{Cl}$ is reached, thereafter switching to Solution II until the end of the input data string. indicates that Solution I processes a small number of segments, while Solution II handles a larger number. Consequently, Solution III produces results nearly identical to Solution II.

7.3. Summary

The findings presented in this chapter demonstrate that the Flag Order System, when applied using the MANS numeral system, offers a novel approach to data compression. Key takeaways include: Flag Information (FI) - Solution I: This approach allows for high compression efficiency but incurs additional bit overhead due to flag storage requirements. The placement of flags within the data stream significantly impacts the overall compression ratio. Flag to Flag (F2F) - Solution II: By eliminating the need for explicit flag information, this method achieves improved compression efficiency, particularly in scenarios where data segments can be effectively grouped based on their structural properties. Flag to Flag with Flag Information (Solution III): This model combain the previous two solutions, optimising compression while maintaining flexibility in flag placement. Testing on text files from the Canterbury Corpus confirms that Solution II (F2F) algorithm capability to compete with traditional methods like pack and compress. Furthermore, exclusive tests on multimedia files reveal that DE, unlike conventional compressors, is able to uncover redundancies even in already-compressed formats. Overall, the results underscore the potential of the MANS-based F2F-DE approach as a viable alternative for lossless data compression, particularly in environments where traditional algorithms struggle to achieve further data reduction [75].

Chapter 8

This chapter summarises the key findings of the research, which focuses on advancing lossless data compression through innovative methodologies for calculating data occurrences in binary sequences and developing algorithms to streamline these calculations. The work builds upon foundational theories by Shannon and Kolmogorov while introducing fresh paradigms that consider factors like complexity, memory size, and encoding/decoding speed. Central to the contributions of this research are the development of the Adaptive Numeral System (ANS), Improved Adaptive Numeral System (IANS), and Modified Adaptive Numeral System (MANS), along with the introduction of the Data Extraction (DE) technique. The research culminates in the creation of DE-FTF, a method that demonstrates exceptional compression efficiency, particularly with already compressed files, and surpasses traditional methods in various cases. While the findings of this research provide significant advancements in the field, several promising directions for future study have been identified in Section 8.2. These avenues of further research are aimed at refining the methods developed and expanding their applicability across different domains.

8.2. Conclusions

In conclusion, the pursuit of this research has been twofold: firstly, to augment the efficacy of lossless data compression by innovating and exploring novel methodologies for calculating data occurrences in binary sequences, and secondly, to devise appropriate algorithms to streamline these calculations. This progression has led to the introduction of a fresh paradigm in lossless compression which takes into consideration crucial factors such as complexity, memory size, and encoding/decoding speed.

Building upon seminal works by Shannon and Kolmogorov, this research has evaluated their contributions and limitations within the broader context of the field. Benchmarking against prior studies has provided deeper insights into existing methodologies, particularly in compression techniques.

The research journey has commenced with the creation of the Adaptive Numeral System (ANS), a novel numeral system designed to calculate binary values adaptively. Subsequent iterations have led to the development of the Improved Adaptive Numeral System (IANS), which has demonstrated superior efficiency and symmetry compared with its predecessor. Notably, the ANS and IANS possess the unique capability to compress each segment successively, thereby reducing the overall data size iteratively.

Further refinement of the IANS has culminated in the exploration of conditional compression methods, resulting eventually in the creation of the Data Extraction (DE) technique. Leveraging a Modified Adaptive Numeral System (MANS), DE exemplified high-yield compression potential although encountering challenges in identifying flag locations within segmented data. Solutions such as Flag Information (FI), Flag to Flag (FTF), and FI using FTF were proposed and analysed to address this flagging challenge, and varying degrees of efficacy were observed.

In a comparative analysis with conventional methods like Huffman coding, DE-FTF achieved comparable compression rates ranging from 38% to 39% for selected files before conversion to binary. This illustrates the method's effectiveness in achieving competitive compression outcomes. Particularly noteworthy is its performance post-conversion to binary as a non-prefix variable-length code, surpassing other compression methods for specific files. With compression rates reaching up to 56% across all file types, DE-FTF excels when dealing with files containing repeated bits, showcasing its superiority over conventional methods such as Huffman coding, LZW, bzip, Gzip, and others. Additionally, DE-FTF exhibits the capability of being applied to already compressed files, due to the alteration of input data through MANS application and, after the insertion of C_l during the Data Extraction process, the data was segmented with flag insertion, consequently altering the redundancies within the input data. Therefore, Data Extraction (DE) using MANS was evaluated with various file types, including those that were already compressed, such as Joint Photographic Experts Group (JPEG) files. This enabled further compression ranging from 33% up to 35%.

Despite facing challenges and constraints, DE-FTF demonstrates significant advancements in compression efficiency, notably due to its adaptability to already compressed files. These findings suggest a promising direction for future research in the field of lossless data compression.

In summary, this research contributes significantly to the advancement of novel compression methodologies, primarily through the development of the DE-FTF using the MANS technique. The findings highlight DE-FTF's potential, especially post-binary conversion, to attain exceptional compression results, surpassing conventional methods for certain files. Moving forward, future investigations could focus on converting non-prefix to prefix codes, as well as incorporating the $\{1\}$ related to the switch in MANS into the calculations of C for DE-FTF. Additionally, further exploration of DE-FI, as discussed in Section 7.1.1, presents opportunities to enhance compression techniques by reducing the flag information string size or integrating the information within the encoded DE string. These endeavours aim to enhance the efficiency and versatility of lossless data compression techniques, which play a pivotal role in advancing innovative communication methods applicable to emerging fields such as medical imaging, digital media, artificial intelligence, embedded systems, and the Internet of Things.

Original Contributions:

1. The introduction of ANS, IANS, and MANS as novel numeral systems, with superior adaptability and efficiency compared to traditional systems.
2. The development of lossless compression method, DE, which demonstrates exceptional compression rates, particularly in its fully decodable state before binary conversion.
3. The creation of DE-FTF, a technique capable of surpassing conventional methods in specific scenarios, including already compressed files.
4. Proposals for flag-based solutions, which significantly enhance segmentation and compression outcomes.

8.2. Future Directions:

This research lays the foundation for several promising avenues of further investigation:

Refinement of DE-FTF: Future work can focus on optimising the flag information string size and integrating flag data directly into the encoded DE strings to reduce the associated overhead.

Non-Prefix to Prefix Code Conversion: An in-depth theoretical exploration of converting non-prefix codes to prefix codes within the MANS framework could enhance encoding efficiency and contribute to the broader field of lossless data compression.

Further Enhancements to MANS: The exceptional performance of DE-F2F driven by the MANS numeral system highlights its computational efficiency across diverse datasets. further enhancement can focus on incorporation of the {1} Switch in MANS into the calculations of C for DE-FTF could improve the algorithm's performance. Additionally, enhancing DE-FI, as discussed in Section 7.1.1, may lead to further improvements in compression efficiency. Future research could focus on:

- Improving compression ratios by refining the MANS framework.
- Adapting MANS for parallel processing to reduce compression times.
- Expanding applications through comparative analysis of multimedia and non-textual datasets to broaden the framework's applicability.

Expansion into Emerging Fields: As compression needs evolve, there is significant potential for applying DE-FTF to emerging fields such as medical imaging, digital media, artificial intelligence, embedded systems, and the Internet of Things (IoT), where efficient data compression methods are critical.

Optimising Memory Usage: A key direction for future work involves reducing peak memory usage during the encoding phase, particularly for large datasets, and investigating the impact of distributed memory allocation on space efficiency.

Minimising Memory Overhead: Investigating compression techniques that minimise memory overhead, while maintaining or improving speed and compression ratios, will be crucial for applications in resource-constrained environments.

These future directions aim to build upon the findings of this research, expanding its scope and applicability while enhancing the overall performance and efficiency of lossless data compression techniques.

References

[1] Retrieved April 9, 2017, from https://www.bbc.co.uk/news/business-26383058

[2] Taylor, P. (2023, November 16). Worldwide data created. Statista. Retrieved March 25, 2024, from https://www.statista.com/statistics/871513/worldwide-data-created/

[3] Salomon, D. (2004). Data compression: The complete reference (4th ed.). Springer (pp. 2-4).

[4] Sayood, K. (2017). Introduction to data compression (5th ed.). Morgan Kaufmann; Elsevier (pp. 222-223).

[5] Shannon, C. E. (1948). A mathematical theory of communication. The Bell System Technical Journal, 27 (pp. 10-15).

[6] Sayood, K. (2017). Introduction to data compression (5th ed.). Morgan Kaufmann; Elsevier (pp. 34).

[7] Sayood, K. (2017). Introduction to data compression (5th ed.). Morgan Kaufmann; Elsevier (pp. 22).

[8] McAnlis, C., & Haecky, A. (2016). Understanding compression: Data compression for modern developers. O'Reilly Media (pp. 24-25).

[9] Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. Problems of Information Transmission, 1(1), (pp. 1-7). https://doi.org/10.1134/S0032946007000126

[10] Sayood, K. (2017). Introduction to data compression (5th ed.). Morgan Kaufmann; Elsevier (pp. 34-36).

[11] Mondol, T., & Brown, D. G. (2021, September). Incorporating algorithmic information theory into fundamental concepts of computational creativity. In ICCC (pp. 173-181).

[12] Gailly, J.-L., & Adler, M. (1993). Zlib compressed data format specification version 3.3. Retrieved from https://www.zlib.net/zlib_tech.html

[13] Seward, J. (1996). bzip2: A block-sorting file compressor - Algorithm version 0.9. Retrieved from https://web.archive.org/web/20060206200439/http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.html

[14] Zhao, Y., & Li, J. (2014). Implementation of the LZMA compression algorithm on FPGA. Electronics Letters, 50(19), (pp. 1370–1372). https://doi.org/10.1049/el.2014.1734

[15] Huffman, D. A. (1951). A method for the construction of minimum redundancy codes. Proceedings of the IRE, 40, (pp. 1098–1101).

[16] Sayood, K. (2017). Introduction to data compression (5th ed.). Morgan Kaufmann; Elsevier (pp. 205).

[17] McAnlis, C., & Haecky, A. (2016). Understanding compression: Data compression for modern developers. O'Reilly Media (pp. 126-133).

[18] Sayood, K. (2017). Introduction to data compression (5th ed.). Morgan Kaufmann; Elsevier (pp. 134-146).

[19] Lempel, A., & Ziv, J. (1977). A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3), (pp. 337-343). https://doi.org/10.1109/TIT.1977.1055714

[20] Lempel, A., & Ziv, J. (1978). Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory, 24(5), (pp. 530-536). https://doi.org/10.1109/TIT.1978.1055934

[21] Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic coding for data compression. Communications of the ACM, 30(6), (pp. 520-540).

[22] Rissanen, J. (1979). Arithmetic coding as number representation. Acta Polytechnica Scandinavica, Mathematics, Computing and Management in Engineering Series, 31(8), (pp. 29-39).

[23] Langdon, G. G. (1984). An introduction to arithmetic coding. IBM Journal of Research

and Development, 28(2), 135-149. https://doi.org/10.1147/rd.282.0135

[24] Howard, P. G., & Vitter, J. S. (1994). Arithmetic coding for data compression. Proceedings of the IEEE, 82(6), (pp. 857-865). https://doi.org/10.1109/5.286189

[25] Rissanen, J. J. (1976). Generalized Kraft inequality and arithmetic coding. IBM Journal of Research and Development. https://doi.org/10.1147/rd.203.0198

[26] Pavlov, I. (2018). LZMA SDK (Software Development Kit) (Version 18.05) [Software]. Available from https://www.7-zip.org/sdk.html

[27] Duda, J. (2009). Asymmetric numeral systems. Retrieved from http://arxiv.org/abs/0902.0271

[28] Sayood, K. (2017). Introduction to data compression (5th ed.). Morgan Kaufmann; Elsevier (pp. 59-74).

[29] ZLIB compressed data format specification version 3.3. (1996). Retrieved from https://www.ietf.org/rfc/rfc1950.txt

[30] Wallace, G. K. (1991). The JPEG still picture compression standard. Communications of the ACM, 34(4) (pp. 31-44).

[31] Sayood, K. (2017). Introduction to data compression (5th ed.). Morgan Kaufmann; Elsevier (pp. 45-50).

[32] Sayood, K. (2017). Introduction to data compression (5th ed.). Morgan Kaufmann; Elsevier (pp. 47-50).

[33] Hashimoto, K., & Iwata, K.-i. (2021). On the optimality of binary AIFV codes with two code trees. In 2021 IEEE International Symposium on Information Theory (ISIT) (pp. 3173-3178). IEEE. https://doi.org/10.1109/ISIT45174.2021.9518105

[34] Hashimoto, K., & Iwata, K.-i. (2022). Optimality of Huffman code in the class of 1-bit delay decodable codes. IEEE Journal on Selected Areas in Information Theory, 3(4), 616-625. https://doi.org/10.1109/JSAIT.2022.3230745

[35] Golomb, S. W. (1966). Run-length encodings. IEEE Transactions on Information

Theory, 12 (pp. 399–401).

[36] Peng, X., Zhang, Y., Peng, D., & Zhu, J. (2023). Selective run-length encoding. arXiv preprint. Retrieved from https://arxiv.org/abs/2312.17024

[37] Burrows, M., & Wheeler, D. J. (1994). A block-sorting data compression algorithm. Technical Report SRC 124, Digital Systems Research Center.

[38] Begum, M. B., Deepa, N., Uddin, M., Kaluri, R., Abdelhaq, M., & Alsaqour, R. (2023). An efficient and secure compression technique for data protection using Burrows-Wheeler transform algorithm. Heliyon, 9(6), e17602. https://doi.org/10.1016/j.heliyon.2023.e17602

[39] Shanmugasundaram, S., & Lourdusamy, R. (2011). Text preprocessing using enhanced intelligent dictionary-based encoding (EIDBE). In *2011 3rd International Conference on Electronics Computer Technology (ICECT)* (pp. 451-455). Kanyakumari, India: IEEE. https://doi.org/10.1109/ICECTECH.2011.5941833

[40] Wertenbroek, R., Xenarios, I., Thoma, Y., & Delaneau, O. (2023). Exploiting parallelization in positional Burrows-Wheeler transform (PBWT) algorithms for efficient haplotype matching and compression. *Bioinformatics Advances*, 3(1), vbad021. https://doi.org/10.1093/bioadv/vbad021

[41] Demongeot, J., Gardes, J., Maldivi, C., Boisset, D., Boufama, K., & Touzouti, I. (2023). Genomic phylogeny using the Maxwell™ classifier based on Burrows–Wheeler transform. *Computation*, 11, (pp. 158). https://doi.org/10.3390/computation11080158

[42] Durbin, R. (2014). Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinformatics*, 30(9), (pp. 1266-1272). https://doi.org/10.1093/bioinformatics/btu014

[43] Sinha, S. (2017, September). The Fibonacci numbers and its amazing applications, *6*(9), (pp. 7-14).

[44] Salomon, D. (2007). *Variable-length codes for data compression*. Springer-Verlag.

[45] Zeckendorf, E. (1972). Representation of natural numbers by a sum of Fibonacci numbers or Lucas numbers. *Fibonacci Quarterly*, 10(3), (pp. 245-250).

[46] Fenwick, P. (2002, August). Variable-length integer codes based on the Goldbach conjecture, and other additive codes. *48*, (pp. 2413).

[47] Lelewer, D. A., & Hirschberg, D. S. (1987). Data compression. *ACM Computing Surveys*, 19, (pp. 261–296).

[48] Fraenkel, A. S., & Klein, S. T. (1996). Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64, (pp. 31–55).

[49] Apostolico, A., & Fraenkel, A. S. (1987). Robust transmission of unbounded strings using Fibonacci representations. *IEEE Transactions on Information Theory*, 33, (pp. 238–245).

[50] Walder, J., Krátký, M., Bača, R., Platoš, J., & Snašel, V. (2010). Fast decoding algorithms for variable-length codes. *Information Science*. Submitted.

[51] Przywarski, R., Grabowski, S., Navarro, G., & Salinger, A. (2006). FM-KZ: An even simpler alphabet-independent FM-index. In *Proceedings of the Prague Stringology Conference* (pp. 226–239). Czech Technical University, Prague.

[52] Pasco, R. (1976). *Source coding algorithms for fast data compression* (PhD thesis, Stanford University).

[53] Rissanen, J. J. (1976). Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3), (pp. 198–203).

[54] Rissanen, J. J., & Langdon, G. G. (1979). Arithmetic coding. *IBM Journal of Research and Development*, 23(2) (pp. 149–162).

[55] Sayood, K. (2017). *Introduction to data compression* (5th ed.). Morgan Kaufmann Publishers; Elsevier (pp. 89-116).

[56] Said, A. (2023). Introduction to arithmetic coding – theory and practice. *arXiv preprint*. https://arxiv.org/abs/2302.00819

[57] Yu, Q., Yu, W., Yang, P., Zheng, J., Zheng, X., & YunHe, (2015). An efficient adaptive binary arithmetic coder based on logarithmic domain. *IEEE Transactions on Image Processing*, 24(11).

[58] Rabbani, M. (2002). JPEG2000: Image compression fundamentals, standards and practice. *Journal of Electronic Imaging*, 11(2). https://doi.org/10.1117/1.1469618

[59] Kim, H. J. (2010). A fast implementation of arithmetic coding. In *2010 12th International Asia-Pacific Web Conference* (pp. 419-423). Busan, South Korea. https://doi.org/10.1109/APWeb.2010.76

[60] Duda, J. (2009). Asymmetric numeral systems. *arXiv preprint*. https://arxiv.org/abs/0902.0271

[61] Duda, J., et al. (2015). The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *2015 Picture Coding Symposium, PCS 2015 - with 2015 Packet Video Workshop, PV 2015 - Proceedings* (pp. 65–69). IEEE. https://doi.org/10.1109/PCS.2015.7170048

[62] Duda, J. (2013). Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding (pp. 1-24). *arXiv preprint*. https://arxiv.org/abs/1311.2540

[63] Townsend, J. (2020). A tutorial on the range variant of asymmetric numeral systems. *arXiv preprint*. https://arxiv.org/abs/2001.09186

[64] McAnlis, C., & Haecky, A. (2016). Understanding compression: Data compression for modern developers (1st ed.). O'Reilly Media.

[65] Collet, Y., & Turner, C. (2016). Smaller and faster data compression with Zstandard. *Facebook Code Blog*. https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/

[66] Torres, M. M., et al. (2020). High-throughput variable-to-fixed entropy codec using

selective, stochastic code forests. *IEEE Access*, 8, (pp. 81283–81297). https://doi.org/10.1109/ACCESS.2020.2992993

[67] Najmabadi, S. M., et al. (2019). An architecture for asymmetric numeral systems entropy decoder – A comparison with a canonical Huffman decoder. *Journal of Signal Processing Systems*, 91(7) (pp. 805–817). https://doi.org/10.1007/s11265-018-1421-4

[68] Wang, N., Qin, C., & Lin, S.-J. (2023). A novel reversible data hiding scheme based on asymmetric numeral systems. *arXiv preprint*. https://arxiv.org/abs/2307.08190

[69] Yokoo, H., & Shimizu, T. (2019). Probability approximation in asymmetric numeral systems. In *Proceedings of the 2018 International Symposium on Information Theory and Its Applications* (pp. 638–642). IEICE. https://doi.org/10.23919/ISITA.2018.8664207

[70] Dube, D., & Yokoo, H. (2019). Fast construction of almost optimal symbol distributions for asymmetric numeral systems. *IEEE International Symposium on Information Theory - Proceedings* (pp. 1682–1686). IEEE. https://doi.org/10.1109/ISIT.2019.8849430

[71] Talli, P., Pase, F., Chiariotti, F., Zanella, A., & Zorzi, M. (2024). Effective communication with dynamic feature compression. *arXiv preprint*. https://arxiv.org/abs/2401.16236

[72] Lee, H., & Yang, T. (2020). The role of binary systems in data compression. Journal of Data Science, 12(3), 45-58.

[73] Brown, P., & Zhou, L. (2019). Advanced methods for encoding binary data. Springer.

[74] Powell, M. (2001, November 20). *Corpus Canterbury*. https://corpus.canterbury.ac.nz/index.html

[75] Fitriya, L. A., Purboyo, T. W., & Prasasti, A. L. (2017). A review of data compression techniques. *International Journal of Applied Engineering Research*, 12(19), 8956–8963.

[76] Moffat, A. (2019). Huffman coding. *ACM Computing Surveys*, 52(4), 1–35.

[77] Varshney, A., Suneetha, K., & Yadav, D. K. (2024). Analyzing the performance of different compactor techniques in data compression & source coding. In *2024 International Conference on Optimization Computing and Wireless Communication (ICOCWC)*. Debre Tabor, Ethiopia.

[78] Gopinath, A., & Ravisankar, M. (2020). Comparison of lossless data compression techniques. In *2020 International Conference on Inventive Computation Technologies (ICICT)* (pp. 628-633). Coimbatore, India: IEEE. https://doi.org/10.1109/ICICT48043.2020.9112516

[79] Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression.

[80] Gao, L., Zhang, B., Jiang, T., & He, Z. (2022). Application of a lossless compression algorithm based on Burrows-Wheeler transform for large data sets. *Journal of Computational Biology*, 19(7), 1100–1107.

[81] Peñaranda, C., Reaño, C., & Silla, F. (2024). Hybrid-Smash: A heterogeneous CPU-GPU compression library. doi: 10.1109/ACCESS.2024.3371253

[82] Awan, F. S., Zhang, N., Motgi, N., Iqbal, R. T., & Mukherjee, A. (2001, March). LIPT: A reversible lossless text transform to improve compression performance. In *Data Compression Conference* (p. 481). Snowbird, UT, USA.

[83] Shcherbakov, I., Weis, C., & Wehn, N. (2012, May). A high-performance FPGA-based implementation of the LZSS compression algorithm. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (pp. 449–453). IEEE. https://doi.org/10.1109/IPDPSW.2012.66

[84] Sayood, K. (2017). *Introduction to data compression* (5th ed.). Morgan Kaufmann Publishers; Elsevier (pp. 134 - 145).

[85] Bunton, S. (1995). The structure of dynamic Markov compression (DMC). In *Proceedings of the DCC '95 Data Compression Conference* (pp. 72–81). Snowbird, UT, USA.

[86] Feregrino, C. (2003). High performance PPMC compression algorithm. In *Proceedings of the Fourth Mexican International Conference on Computer Science, ENC 2003* (pp. 135–142). Tlaxcala, Mexico.

[87] Razilov, V., Wittig, R., Matúš, E., et al. (2024). Access interval prediction by partial matching for tightly coupled memory systems. *International Journal of Parallel Programming*, 52, 3–19. https://doi.org/10.1007/s10766-024-00764-1

Appendices

Appendix A

1. Text to ASCII to MANS.

```
public class Text2Ascii {
// converting text to ASCII code
public static String convertStringToBinary(String input) {
    StringBuilder result = new StringBuilder();
    char[] chars = input.toCharArray();
    for (char aChar : chars) {
      result.append(
          String.format("%8s", Integer.toBinaryString(aChar))   // char -> int, auto-cast
              .replaceAll(" ", "0")                  // zero pads
      );}
    return result.toString();
  }
//removing any space or separators from string of binary
public static String prettyBinary(String binary, int blockSize, String separator) {
    List<String> result = new ArrayList<>();
    int index = 0;
    while (index < binary.length()) {
      result.add(binary.substring(index, Math.min(index + blockSize, binary.length())));
      index += blockSize;}
    return result.stream().collect(Collectors.joining(separator));}
 //converting a string of binary numbers to characters then ASCII (decimal) then text.
public static String BinaryToAscii(String input) {
    StringBuilder sb = new StringBuilder();
    String str = "";
    String text = "";
        char[] chars = input.replaceAll("\\s", "").toCharArray();
        for (int j = 0; j < chars.length; j+=8) {
          int idx = 0;
          int sum = 0;
          //for each bit in reverse
          for (int i = 7; i>= 0; i--) {
            if (chars[i+j] == '1') {
                sum += 1 << idx;}
            idx++;}
```

```
                    sb.append(Character.toChars(sum));
                str = Character.toString((char)sum);
                text=text+str;}return text;}
//Converting binary string to Modified adaptive numeral system
public class BIN2MANS {
                public static String ConvertBin2Mans (String BinInput) {
//input data
                StringBuilder MansResult = new StringBuilder();
                char[] BinChars = BinInput.toCharArray();
// Initialising MANS
                 int j = 0;
                 int s=0;
                 int[] pMans = new int[BinInput.length()*2]; // creating a string with a maximum length of 2Xthe input data length
// Converting to MANS
                for (int i=0;i<BinInput.length();i++, j++){
                    if(i>0) {
                     if(BinChars[i]==BinChars[i-1]){  // if repeated bit detected B[i]=0
                     pMans[j]=0;
                     } else { // if switch detected, A[i]=1 B[i+1]=0
                     pMans[j]=1; // A[i] = 1
                     pMans[j+1]=0; // B[i+1]=0
                     j++;
                     s++;
                     } else {
                     pMans[i]=0;}}
                      System.out.println("Total number of switches = "+s);
                      int [] Pmans = new int [j]; // copying the correct length of the converted MANS string to Pmans
                            for (int a=0;a<j;a++) {
                            Pmans[a]=pMans[a];
                            MansResult.append(Pmans[a]);}
                            return MansResult.toString();}}


public class test {
                public static void main(String[] args) {
                        String input = "Testing the encoding and decoding steps of MANS!";
                        String result = Text2Ascii.convertStringToBinary(input);
                        System.out.println("Original data string is: "+ result.length() +"\n"+result);
                        String MansResult = BIN2MANS.ConvertBin2Mans(result);
                        System.out.println("MANS data is= "+MansResult.length() +"\n"+MansResult);
                        System.out.println("****** Decoding starts here *****");
                        String BinResult = MANS2BIN.ConvertMans2Bin(MansResult);
```

```
            System.out.println("Orginal data after decoding MANS is= "+BinResult.length()+"\n"+BinResult);
            System.out.println("Verifying the results...");
            char [] OriginalBin = result.toCharArray();
            char [] Mans2Bin = BinResult.toCharArray();
            boolean DataCheck=true;
            for (int i =0; i<result.length();i++) {
                if (OriginalBin[i]!=Mans2Bin[i]) {
                System.out.println("Error in bit number " + Mans2Bin[i]);
                DataCheck=false;}}
        if (DataCheck = true) {System.out.println("Successfully decoded, decoded MANS data match the original data");
        }else {System.out.println("unsuccessfully MANS decoded data do not match");}
            String Ascii = Text2Ascii.BinaryToAscii(BinResult);
            System.out.println(Ascii);}}
```

Output:

```
Original data string is: 384
0101010001100101011100110111010001101001011011100110011100100000011101000110100001100101001000000110010
10110111001100011011011110110010001101001011011100110011100100000011000010110111001100100001000000110010
00110010101100011011011110110010001101001011011100110011100100000011100110111010001100101011100000111001
1001000000110111101100110001000000100110101000001010011100101001100100001
Total number of switches = 197
MANS data is= 581
010101010101000100100101010101000100100101000101010001001010100101010010100010010010010001001010000001
000101010001001010100001001001010101001010000001001001010101010010100010010010001001010010100001010010
010100010010101001010100101000100100100100010010100000010010000101010010100010010010010100001010000001
001001010001001001010101010010001001010010100001010010010100010010101001010100101000100100100100010010
100000010001001001010001010100010010010101010100010000010001001001001010000001001010000101001001001000
10100000010100100101010101000001010101001000100101010100100100101000010
****** Decoding starts here *****
Orginal data after decoding MANS is= 384
0101010001100101011100110111010001101001011011100110011100100000011101000110100001100101001000000110010
10110111001100011011011110110010001101001011011100110011100100000011000010110111001100100001000000110010
00110010101100011011011110110010001101001011011100110011100100000011100110111010001100101011100000111001
1001000000110111101100110001000000100110101000001010011100101001100100001
Verifying the results...
Successfully decoded, decoded MANS data match the original data
Testing the encoding and decoding steps of MANS!
```

2. Flag Information, Solution I.

```
//Main Class that calls all other classes
import java.io.File;
public class Test2 {
    public static void main(String[] args) throws Exception  {
// converting the Base64 file to Binary
        File path = new File("/Users/od/Desktop/U.txt");
        String s = File2ByteArray.method(path);
        System.out.println(s);
        String f = Base64ToBinary.B642B(s);
        System.out.println(f);
        System.out.println("the total bit of the original file = "+f.length());
//Converting Binary file to MANS
        String r = BIN2MANS.ConvertBin2Mans(f);
        System.out.println("MANS output = \n"+r);
//Using Data Extraction to compress MANS using Flag Information solution
        String[] t = DataExtraction.FI(r, 3); // C=3
        System.out.println("Segment encoded = "+t[0]+"\n"+"Flag Information = "+t[1]);
//Decoding Data Extraction to MANS using Flag Information
        String[] d = DecoderDateExtraction.DEFI(t, 4);
        String ary = Arrays.toString(d).replaceAll("[, \\[\\]]", "");
        System.out.println(ary);
//Converting MANS to Binary
        String e = MANS2BIN.ConvertMans2Bin(ary);
        System.out.println("\n"+e);
//Converting Binary to Base64
        String g = Binary2Base64.B2B64(e);
        System.out.println(g);
//Writing file
        String h = Base642File.Base64ToFile(g, "/Users/od/Desktop/U2.txt");}}


import java.util.Arrays;
//Data Extraction, bits counted is only 0, 2 bits flag with addtional string to identify the flag order, the additional flag string consist of 1 and 0. 1 indicate the first flag from LSB while 0 indicate the wrong flag.
public class DataExtraction {
    public static String[] FI(String MANSInput, int com) {
        // Initialising DE
        int EndOfData = 0; // initialising the end of data
        int segNo = 0; // the segment number of the data processed using Data Extraction
        int segCheck=0;
        int PsegNo = 0; // initialising the length of the previous segment
        int j = MANSInput.length(); // the length of the MANS data received.
        int lim1 = (int) Math.pow(2, com);
        int lim2=0;
        int shift=0;
        int[][] Pmans = new int[j][]; // copying the correct length of the flagged string to Pmans
        int[][] Fpmans = new int[j][];// processing the MANS data to flag the value of C
        int[][] c = new int[j][com]; // identifying the length of C
        StringBuffer flagInfo = new StringBuffer(); // the flags information stored for each segment
        int count = com - 1; // initialising variable count to calculate the value of C in decimal
        int Cdec = 0; // defining the decimal value of C
        int po = 0; // defining the power to convert binary to decimal
        int sgRun = 0; // sgRun to identify the bit number in the data string and to shift data before
                                                // placing the flag, start count after C
        int f = 0; // initialising the flag location for each segment
        int flagNo = 0; // initialising the flag order in each segment
        int totalFlags = 0; // initialising the total flags in each segment
        char[] BinChars = MANSInput.toCharArray();// copying data from MANSInput
// Data Extraction
        while (EndOfData == 0) { //Keep processing until the end of data
            Pmans[segNo] = new int[j]; //setting the right length of data of the current segment
        if (segNo != 0) { // if this is not the first segment
        for (int a = 0; a < j; a++) { // loop until the end of the current segment length
        if(a< PsegNo) { // if the current bit of the segment < then previous segment length
        if(segCheck==1) {// copy the previous segment to new one
            Pmans[segNo][a] = Fpmans[segNo-1][a];
        }else {//
            Pmans[segNo][a] = Fpmans[segNo][a];}
        }else { // if the current bit of the segment > then previous segment length
            Pmans[segNo][a] = Pmans[0][a+shift]; }}//copy the remaining bits from the initial string
        } else {// if this is the first segment
            Fpmans[segNo] = new int[j]; //setting the right length of data of the current segment (reusing
array for memory)
// If this is the first segment, copy the data from MANSInput to Pmans and Fpmans
        for (int a = 0; a < MANSInput.length(); a++) {
        if (BinChars[a] == 48) {
            Pmans[segNo][a] = 0;
            Fpmans[segNo][a] = 0;
        } else {
            Pmans[segNo][a] = 1;
            Fpmans[segNo][a] = 1;}}}
            Fpmans[segNo] = new int[j];//setting the right length of data of the current segment (reusing array for memory )
//counting the first c bits from the MSB (flipping the first C bits) and covert it to its decimal value
            j=0;
            lim2=0;
        for (int b = 0; b < c[segNo].length; b++) {
            c[segNo][b] = Pmans[segNo][b]; // C(PMAS) obtained (binary from MSB)
            if (c[segNo][b] == 1) {
                po = (int) Math.pow(2, count);
                Cdec = Cdec + po; // C(PMANS) obtained decimal value of C (from LSB)
                count--;
        } else {
            count--;}}
//flagging the segments when bit = C
        for (int d = c[segNo].length; d < Pmans[segNo].length; d++){ // start counting after C until the end of Pmans
            Fpmans[segNo][sgRun] = Pmans[segNo][d];
        if (Fpmans[segNo][sgRun] == 1) {
        if (sgRun > 1) {// Count the total flags detected in the segment
        if (Fpmans[segNo][sgRun - 1] == 1 & Fpmans[segNo][sgRun - 2] == 0) { // Total flags in a segment including flags after C
            totalFlags++;}}
        if(j<lim1) {
          lim2++;}}
        else if (Fpmans[segNo][sgRun] == 0) {
        if (f == Cdec) { // if the flag = C value, flag the current bit
          sgRun++;
          Fpmans[segNo][sgRun] = 1;
          sgRun++;
          Fpmans[segNo][sgRun] = 1;
          totalFlags++; // Increment the total flags
          flagNo = totalFlags;// current segment flag number = flagNo
          lim2++;//counting the number of available bits in the current segment}
        if(j<lim1) {
          lim2++;
          j++;}
        if(f!=-1) {
          f++; // increment the number of bit to read, to compare it with C
        }}
          sgRun++; // first segment is processed, move to the next segment
        }
        if (totalFlags - flagNo == 0) { //check the flag position of the segment
```

```
                    flagInfo.append(1); //store the flag order (1st flag from LSB)
                } else {
                for (int i = 0; i < totalFlags - flagNo + 1; i++) {
                if (i > 0) {
                    flagInfo.append(0); //store the number of flags to skip
                } else {
                    flagInfo.append(1); //store the flag order
                }}}
                if(f==-1) {
                    EndOfData=1;}
//reseting variables
                f = 0;
                shift=shift+(com-2);
                count = com - 1;
                po = 0;
                Cdec = 0;
                sgRun = 0;
        if (segCheck >0) {
                segCheck++;}
                PsegNo=Fpmans[segNo].length-(com-2);
        if(segNo==0) {
                segNo++;
                segCheck=1;
        }
                flagNo = 0;
                totalFlags = 0;
// resizing the next segment length
        if(segNo !=0) {
        for (int a = 0; a < lim2; a++) {
        if(Pmans[0].length>lim2+shift) {//checking if the end of the initial data string length is reached
        if(j<lim1+2) {//if the end of the initial data string not reached, resizing the next segment length
        if(Pmans[0][lim2+shift]==0) {//count the number of {0} in the current segment
                j++;
                lim2++;
        }else {//count the number of {1} in the current segment
                lim2++;}
        }}else {//if the end of the initial data string is reached, resizing the next segment to include the last bits from the
initial data string and initialise the end of the encoder
        a=lim2;
        f=-1;}}
        j=lim2; }}
        String res [] =new String[2];
        res[0]=Arrays.toString(Pmans[segNo]);
        res[1]=flagInfo.toString();
        System.out.println("Orginal MANS data string length is " + MANSInput.length());
        System.out.println("DE = " + Arrays.toString(Pmans[segNo])+" = " + Pmans[segNo].length);
        System.out.println("Segment No to send " + (segCheck) + ", Seg length " + Pmans[segNo].length+ " Total bits of the
flagInfo " + flagInfo.length() + " Total DE bits to send = "+ (flagInfo.length() + (Pmans[segNo].length)));
        return (res);
        }

import java.util.Arrays;
public class DecoderDateExtraction {
        public static String[] DEFI(String[] DEFI, int com) {
// Initialising DE
        int[] F = new int [DEFI[1].length()];
        int[] c= new int [com];
        int b=0; // initialising variable to count the flag location
        int count=0;
        int fLoc = 0;
        int FC = 0; // Flag Count (FC) from DE string
        int FCB=0; // Flag Count Bits (FCB), counting the run of {1} for each flag
        int FCBM = 0; // Flag Count Bit Merge (FCBM), merging the number of {1} to a flag in each flag run
        char[] FI = DEFI[1].toCharArray();// flag information string
        char[] MANSChars = DEFI[0].replaceAll("[, \\[\\]]", "").toCharArray();
        int[] DE = new int [MANSChars.length]; // remove brackets comma and pace from string and converting the string
to Char
        int[] DE2 = new int [MANSChars.length]; // remove brackets comma and pace from string and converting the string
to Char
// sorting all the flags locations
        for(int i = FI.length-1 ; i>-1 ;i--) {
        if(FI[i]==48) {
                fLoc++;
        }else {
                fLoc++;
                F[count]=fLoc;
                count++;
                fLoc=0;
        }
        }
        for (int a = 0; a < MANSChars.length; a++) {
        if (MANSChars[a] == 48) {
                DE[a] = 0;
                DE2[a] = 0;
```

```
    } else {
        DE[a] = 1;
        DE2[a] = 1;
    }
    }
    int TF [] = new int[count];// resizing to the actual flag location length
    for(int i=0;i<count;i++) {
        TF[i]=F[i];
    }
        count=0;
// Decoding DE
    for (int i=DE.length-1; i>-1; i--) {// determining the number of flags in the segment
    if(DE[i]==0) {
        count++;
    }else if(i!=0&&DE[i-1]==1 && DE[i+1]==0){
    for (int a=i;a>-1;a--) {
    if(DE[a]==1) {
        FCB++;
    }else {
        a=-1;}}}
        FCBM = FCB/2;
    if (FCB > 2) {
        FCBM=1;}
        FC=FC+FCBM;
    if(FC>=TF[b]) {
        count = 0;
    for (int r=i; r>-1; r--) {// counting the number of {0} after the flag location to determine C
    if(DE[r]==0) {
        count++; }}
    for (int d =0, e=0; d<DE.length-1 ; d++,e++) {
    if(d==i-1) { // if the location of the flag = the reading bit
        d=d+1; // if yes, move additional bit to avoid copying the detected flag
System.out.println("i= "+i+" d= "+d);
        e--;// reset e to previous value to copy the right bit after the detected flag
    }else {
        DE2[e]=DE[d];// copying the data to new string, without the detected flag
    }}
    if (count ==0) {
        count++;}
        String SC = Integer.toBinaryString(count-1);
    for(int g=SC.length();g<com;g++) {
        SC=0+SC;}
```

```
                String[] ary = SC.split("");
        for(int g=0;g<com;g++) {
                c[g]=Integer.parseInt(ary[g]);}
        DE = new int[DE.length+com-2];
        for (int g=0;g<DE.length;g++){
        while (g<c.length) {
                DE[g]=c[g];
                g++;}
                DE[g]=DE2[g-c.length];}
                DE2 = new int[DE.length+com-2];
                i=DE.length;
                FC=0;
                FCB=0;
        if(TF.length==b+1) { //exit the loop if there is no more flags
                i=-1;
        }else {
                b++; // move to the next flag check
        }
        }
                FCB=0;
                FCBM=0;
                count=0;}
                String[] aryDE = new String [DE.length];
        for(int g=0;g<DE.length;g++) {
                aryDE[g]=String.valueOf(DE[g]);}
        return aryDE;}}


public class MANS2BIN {
public static String ConvertMans2Bin (String MansResults) {
//input data
        StringBuilder BinResult = new StringBuilder();
        char[] BinChars = MansResults.toCharArray();
// Initialising Bin
        int initial = 1;
        int j = 0;
        int[] Mans = new int[MansResults.length()];
// Converting to Bin
        if(BinChars[0]=='1') {
          initial =1;}
        for (int i=0;i<BinChars.length;i++){
        if(BinChars[i]=='0'){
```

```
                        Mans[j]=initial;
                        BinResult.append(initial);
                        j++;
                } else { // if switch detected, A[i]=1 B[i+1]=0
                if(initial == 0) {
                        initial = 1;
                }else {
                        initial = 0;}}}
                return BinResult.toString();}}
```

```
import java.math.BigInteger;
import java.util.Base64;
//send string of binary number with no spaces between the code to convert it to byte array then Base64
public class Binary2Base64 {
        static String B2B64 (String BInput) {
        byte[] Str2Byte = new BigInteger(BInput, 2).toByteArray();
        String encoded = new String(Base64.getEncoder().encodeToString(Str2Byte));
        return encoded;}}
```

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Base64;
// writing file from Base 64
public class Base642File {
        public static String Base64ToFile (String data, String path) throws IOException{
        byte[] d = Base64.getDecoder().decode(data);
        try (OutputStream stream = new FileOutputStream(path)) {
          stream.write(d);
          stream.close();}
        return path;}}
```

Output

Encoding DE

```
VGVzdGluZyBhIHRleHQgZmlsZSB1c2luZyBNQU5TIGFuZCBERS4=
10101000110010101110011011101000110100101101110011001110010000000110000100100000001110100011001010111100
01110100001000000110011001101001011011000110010100100000001110101011100110110100101101110011001110010000
00100110101000001010011100101001100100000011000010110111001100100001000000100010001000101001001110
the total bit of the original file = 303
Total number of switches = 157
MANS output =
01010101010001001001010101010001001001010001010100010010101001010100101000100100100100010010100000000100
100001010010100000001000101010001001001010101010000100001000101010000101000000100100100100100101010100101
```

```
01001010010001001001010101001010000000100010101010101000100100101001010100101010010100010010010010001001
10100000001010010010101010100000101010100100010010101010010010010100000001001000010101001010001001001001
0100001010000000101000101000101000101010100101010010
Original MANS data string length is 460
DE = [1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0] = 35
Segment No to send 426, Seg length 35 Total bits of the flagInfo 672 Total DE bits to send = 707
Segment encoded = [1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0]
Flag Information =
1110110110111110100110110011001011001111110001111011111010011001110110111011011110111111011111100100111011
1101001110101101110001110011000111110011111001110011001001101100011011111110001110001111011111000100101111
0011101111000100011100111011100101101110111000110011100011101110001111100110011101111111001100111011011111
1001111111111101001100110010110111011110111111111000111110011001111100011111111001110111110111000011001110
0111001111011101110110011111011000111111100110001001111101111101111111100011110110010110111000111011011101
1001011100111110110000110010101111110110001001011011110111000111111100111101101100110010111011100001110011
01011101111011100001111111000100111000000
```

Decoding DE

```
DE=
0101010101000100100101010101000100100101000101010001001010100101010010100010010010001001010000000100
10000101001010000000100010101000100100101010101000010000100010101000010100000001001001001001001010100101
01001010010001001001010101001010000000100010101010101000100100101001010100101010010100010010010010001001
10100000001010010010101010100000101010100100010010101010010010010100000001001000010101001010001001001001
0100001010000000101000101000101000101010100101010010
MANS to Binary
1010100011001010111001101110100011010010110111001100111001000000110000100100000011101000110010101111000
01110100001000000110011001101001011011000110010100100000011101010111001101101001011011100110011100100000
001001101010000010100111001010011001000000110000101101110011001000001000000010001000100010100101110
Bainary to Base 64
VGVzdGluZyBhIHRleHQgZmlsZSB1c2luZyBNQU5TIGFuZCBERS4=
```

3. Flag to Flag, Solution II.

```
import java.io.File;
public class Test2 {
            public static void main(String[] args) throws Exception  {
// converting the Base64 file to Binary
                        File path = new File("/Users/od/Desktop/U.txt");
                        String s = File2ByteArray.method(path);
                        System.out.println(s);
                        String f = Base64ToBinary.B642B(s);
                        System.out.println(f);
                        System.out.println("the total bit of the original file = "+f.length());
//converting Binary file to MANS
```

```
                String r = BIN2MANS.ConvertBin2Mans(f);
                System.out.println("MANS output = \n"+r);
//Data Extraction using F2F
                String t = DataExtraction.F2F(r, 3);
                System.out.println(t);}}
//Data Extraction, bits counted is only 0. 2 bits flag. flags placed according to the value of C. count start after the previous flag and after the first segment.
import java.util.Arrays;
public class DataExtraction {
public static String F2F(String MANSInput, int com){
//Data Extraction
//Initialising DE
int EndOfData = 0; //initialising the end of data
int segNo=0;// the segment number of the data processed using Data Extraction
int j = MANSInput.length(); // the length of the MANS data received.
int[][] Pmans = new int[j][]; // copying the correct length of the flagged string to Pmans
int[][] Fpmans = new int[j][];// processing the MANS data to flag the value of C
int [][] c = new int[j][com];// identifying the length of C by the user from 'com' variable
int count = com - 1; // initialising variable count to calculate the value of C in decimal
int PCdec=0; // initialising a variable to store the previous flag location
int Cdec= 0; //defining the value of C
int po=0; //defining the power to convert the C to decimal
int sgRun = 0; // sgRun to identify the bit number in the data string and to shift data before placing the flag, start count after C
int segCount = 0; //to count the number of segments to process
int f = -1; // f is to count the number of repeated bits, excluding switch
int Pf = 0; // defining the count of bits between previous flag and current flag
char[] BinChars = MANSInput.toCharArray();// copying data from MANSInput
// initialising segments
        while (EndOfData == 0) {//Keep processing until the end of data
          Pmans [segNo]= new int [j];//setting the right length of data of the current segment (reusing array for memory)
          if(segNo==0) {// If this is the first segment, copy the data from MANSInput to Pmans and Fpmans
           Fpmans [segNo]= new int [j];//setting the right length of data of the current segment (reusing array for memory)
           for (int a = 0; a < MANSInput.length(); a++) {
             if (BinChars[a] == 48) {
                Pmans[segNo][a] = 0;
                Fpmans[segNo][a] = 0;
                } else {
                Pmans[segNo][a] = 1;
                Fpmans[segNo][a] = 1;
                }}
             }else {// if this is not the first segment
                for (int a=0;a<j;a++) {// loop until the end of the current segment length
```

```
                 if(segCount==1) {
                 Pmans[segNo][a]=Fpmans[segNo-1][a];// copy the previous segment to new one
                }else {
                 if(Pmans[segNo].length>a) {
                 Pmans[segNo][a]=Fpmans[segNo][a];  // copy the previous segment to new one
                }}}
                  Fpmans [segNo]= new int [j];}
//value of C, counting the number of bits in C from the LSB and convert it to decimal
        for (int b = 0; b < c[segNo].length; b++) {
                c[segNo][b] = Pmans[segNo][b]; // C(PMAS) obtained (binary from MSB)
                if (c[segNo][b] == 1) {
                  po = (int) Math.pow(2, count);
                  Cdec = Cdec + po; // C(PMANS) obtained decimal value of C (from LSB)
                  count--;
                } else {
                  count--;
                   Pf++;
                }}
// counting the number of bits between flags
                if(segNo!=0) {
                Cdec = (PCdec - Pf) + Cdec+1;}
//flagging the segments
                for (int d=c[segNo].length;d<Pmans[segNo].length-1;d++)  //start counting after C until the end of Pmans
                {
                Fpmans[segNo][sgRun]=Pmans[segNo][d];
                if(Cdec==0) { // if C = 0 then flag the first two bits after C
                   if(segNo==0) {
                   Fpmans[segNo][sgRun]=1;
                   sgRun++;
                   Fpmans[segNo][sgRun]=1;
                   sgRun++;
                   Cdec=-1; // change the value of the flag after flagging it to exit the loop.
                   }
                 Fpmans[segNo][sgRun]=Pmans[segNo][d];
                }else if(Fpmans[segNo][sgRun]==0){  // if C>1 and there is no switch in this bit increment the flag count
                        f++;
                        if(f==Cdec) { // if the flag = C value, flag the current bit
                        sgRun++;
                        Fpmans[segNo][sgRun]=1;
                        sgRun++;
                        Fpmans[segNo][sgRun]=1; }}
                 sgRun++;
```

```
                    }
                    if(f<=Cdec&&f!=-1) { // When C not equal to the segment length exit
                    System.out.println("****** Segment "+segCount+" flag number is smaller then C, flag has not been placed, send priviouse segment ****** " +" Segment "+ (segCount-1)+" with length = "+Fpmans[segNo].length);
                    EndOfData=1;
                    }else {
                    System.out.println("\n"+"********** Segment "+segCount+ ": is "+Arrays.toString(Fpmans[segNo])+" **********" + "\n Segment length = "+Fpmans[segNo].length);
                    System.out.println("Flag Number after is "+ Pf);}
//reseting variables
                    f=-1;
                    count = com - 1;
                    po=0;
                    PCdec = Cdec;
                    Cdec = 0;
                    sgRun=0;
                    Pf=0;
                    j=j-(com-2);
                    segCount++;
                    if(segNo==0) {
                      segNo++;
                    }
                    System.out.println("Original MANS data string length is " + MANSInput.length());
                    System.out.println("DE = " + Arrays.toString(Pmans[segNo])+" = " + Pmans[segNo].length);
                    System.out.println("Segment No to send " + (segCount-1) + ", Seg length " + Pmans[segNo].length);
return Arrays.toString(Pmans[segNo]);
```

Output

```
VGVzdGluZyBhIHRleHQgZmlsZSB1c2luZyBNQU5TIGFuZCBERS4=
1010100011001010111001101110100011010010110111001100111001000000110000100100000011101000110010101111000
0111010000100000011001100110100101101100011001010010000001110101011100110110100101101110011001110010000
001001101010000010100111001010011001000000110000101101110011001000010000001000100010001010010111 0
the total bit of the original file = 303
Total number of switches = 157
MANS output =
0101010101000100100101010101000100100101000101010001001010100101010010100010010010010001001010000000100
1000010100101000000100010101000100100101010101000010000100010101000010100000010010010010010010101001 01
0100101001000100100101010100101000000100010101010101000100100101001010100101010010100010010010010001 00
101000000010100100101010101000000101010100100010010101010010010010100000001001000010101001010001001001001
01000010100000001010001010001010001010101001010100010
Original MANS data string length is 460
```

```
DE = [1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1,
0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0,
0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0,
1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1,
1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1,
1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0,
1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0,
1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0] = 391
Segment No to send 68, Seg length 391
```

4. Flag to Flag including Flag Information, Solution III.

```
// Data Extraction Solution III C to flag if C>previous flag. Flag to flag if C<Privouse flag
        public static String C2F2F(String MANSInput, int com){
                // Data Extraction
                //Initialising DE
                int EndOfData = 0; //initialising the end of data
                int segNo=0;
                int j = MANSInput.length(); // the length of the MANS data received.
                int [][] Pmans = new int [j][]; // copying the correct length of the converted MANS string to Pmans
                int [][] Fpmans = new int [j][];
                int [][] c = new int[j][com];
                int count = com - 1; // initialising variable count to calculate the value of C in decimal
                int PCdec=0;
                int Cdec= 0; //defining dec value of the first 4 bits
                int po=0;           //defining the power to convert the first 4 bits to decimal
                int sgRun = 0; // sgRun to shift data before placing flag, start count after C
                int segCount = 0;
                int f = -1; // f is to count the number of repeated bits, excluding switch
                int Pf = 0;
                StringBuffer flagInfo = new StringBuffer(); // the flags information stored for each segment
                char[] BinChars = MANSInput.toCharArray();// copying data from MANSInput
// initialising segments
        while (EndOfData == 0) {
                        Pmans [segNo]= new int [j];
                        if(segNo==0) {
                                Fpmans [segNo]= new int [j];
                                        for (int a = 0; a < MANSInput.length(); a++) {
                                                if (BinChars[a] == 48) {
                                                        Pmans[segNo][a] = 0;
                                                        Fpmans[segNo][a] = 0;
                                                } else {
```

```
                                  Pmans[segNo][a] = 1;
                                  Fpmans[segNo][a] = 1;
                             }}}else {
                    for (int a=0;a<j;a++) {
                         if(segCount==1) {
                    Pmans[segNo][a]=Fpmans[segNo-1][a];
                         }else {
                         if(Pmans[segNo].length>a) {
                         Pmans[segNo][a]=Fpmans[segNo][a]; }}}
                    Fpmans [segNo]= new int [j];}
//value of C, counting the number of bits from the LSB and converting C to decimal
                    for (int b = 0; b < c[segNo].length; b++) {
                         c[segNo][b] = Pmans[segNo][b]; // C(PMAS) obtained (binary from MSB)
                         if (c[segNo][b] == 1) {
                              po = (int) Math.pow(2, count);
                              Cdec = Cdec + po; // C(PMANS) obtained decimal value of C
(from LSB)
                              count--;
                         } else {
                              count--;
                         Pf++;}}
               System.out.println("C= "+Arrays.toString(c[segNo])+" = "+Cdec);
           // checking if the current flag > or < than the last flag
               if(segNo!=0 && PCdec-Pf>=Cdec) {
                    Cdec = (PCdec - Pf) + Cdec+1;
//                  System.out.println("Current flag < last flag C changed, (Last C "+Cdec+" - number of
0's in current C "+ Pf[segNo]+" + C  = " + Cdec+")");
                     flagInfo.append(1); //store the flag order (1st flag from LSB)
               }else{
                     flagInfo.append(0); //store the flag order (1st flag from LSB)}
//flagging the segments
               for (int d=c[segNo].length;d<Pmans[segNo].length-1;d++)  //start counting after C until the end
of Pmans
               {
                 Fpmans[segNo][sgRun]=Pmans[segNo][d];
                    if(Cdec==0) { // if C = 0 then flag the first two bits after C
                    if(segNo==0) {
                         Fpmans[segNo][sgRun]=1;
                    sgRun++;
                         Fpmans[segNo][sgRun]=1;
                    sgRun++;
                         Cdec=-1; // change the value of the flag after flagging it to exit the loop.
```

```
                                        }
                                  Fpmans[segNo][sgRun]=Pmans[segNo][d];
                                        }else if(Fpmans[segNo][sgRun]==0){   // if C>1 and there is no switch in this bit increment the flag count
                                              f++;

                                              if(f==Cdec) { // if the flag = C value, flag the current bit
                                              sgRun++;
                                                     Fpmans[segNo][sgRun]=1;
                                              sgRun++;
                                                     Fpmans[segNo][sgRun]=1; }}
                                sgRun++;}
                                if(f<=Cdec&&f!=-1) { // When C not equal to the segment length exit
                                       System.out.println("****** Segment "+segCount+" flag number is smaller then C, flag has not been placed, send priviouse segment ****** " +" Segment "+ (segCount-1)+" with length = "+Fpmans[segNo].length);
                                EndOfData=1;
                                }else {
                                       System.out.println("\n"+"********** Segment "+segCount+ ": is "+Arrays.toString(Fpmans[segNo])+" **********" + "\n Segment length = "+Fpmans[segNo].length);
                                System.out.println("Flag Number after is "+ Pf);
                                System.out.println("Flag Information: "+flagInfo+" = "+flagInfo.length()+ " bits"); }
                                f=-1;
                                count = com - 1;
                                po=0;
                                PCdec = Cdec;
                                Cdec = 0;
                                sgRun=0;
                                Pf=0;
                                j=j-(com-2);
                                segCount++;
                                if(segNo==0) {
                                  segNo++;}
                   return Arrays.toString(Pmans[segNo]);}
```

Output:

```
VGVzdGluZyBhIHRleHQgZmlsZSB1c2luZyBNQU5TIGFuZCBERS4=
1010100011001010111001101110100011010010110111001100111001000000110000100100000011101000110010101111000
0111010000100000011001100110100101101100011001010010000001110101011100110110100101101110011001110010000
001001101010000010100111001010011001000000110000101101110011001000010000001000100010001010100101110
the total bit of the original file = 303
Total number of switches = 157
MANS output =
0101010101000100100101010101000100100101000101010001001010100101010010100010010010001001010000000100
1000010100101000000100010101000100100101010101000010000100010101000010100000001001001001001010100101
0100101001000100100101010100101000000100010101010101000100100101001010100101010010100010010010001000100
1010000000101001001010101010100000101010100100010010101010010010010100000001001000010101001010001001001001
01000010100000001010001010001010001010101001010100010
Orginal MANS data string length is 460
Flag Information: 00101001001111111111111111111111111111111111111111111111111111111111111111 = 72 bits
Flag Information to send: 0010100100
DE = [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0,
1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0,
0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0,
1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1,
1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1,
0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1,
1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0,
0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0] = 388
Segment No to send 71, Seg length 388 Total bits of the flagInfo 72 Total DE bits to send = 460
```

Appendix B

Original File			MANS		C=4 - Solution I - Flag Information					
File Name	Category	Original file size/Bits	Number of switches/Bits	MANS/Bits	Number of processed segments	DE/Bits	Flags info/Bits	Total/Bits	Compression%	Ratio
a.txt	The letter 'a'	7	2	9	2	7	1	8	-14.29	0.88
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	599,978	43	1,144,406	1,144,449	-43.06	0.70
alice29.txt	English text	1,216,708	604,973	1,821,681	910,819	45	1,817,335	1,817,380	-49.37	0.67
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	615,363	42	1,248,312	1,248,354	-56.04	0.64
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	758,108	45	1,506,835	1,506,880	-50.47	0.66
bible.txt	Bibliography (refer format)	32,379,135	16,162,217	48,541,352	24,270,657	40	48,905,512	48,905,552	-51.04	0.66
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	24,270,657	40	48,905,512	48,905,552	-51.04	0.66
book1	Fiction book	6,150,166	3,116,671	9,266,837	4,633,396	47	9,242,301	9,242,348	-50.28	0.67
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	3,690,166	45	7,349,190	7,349,235	-50.39	0.66
cp.html	HTML source	196,822	98,909	295,731	147,848	37	295,858	295,895	-50.34	0.67
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	27,832,120	39	56,784,029	56,784,068	-53.02	0.65
fields.c	C source	89,198	44,627	133,825	66,894	39	134,686	134,725	-51.04	0.66
geo	Geophysical data	819,199	277,687	1,096,886	548,429	30	1,083,426	1,083,456	-32.26	0.76
grammar.lsp	LISP source	29,766	14,837	44,603	22,277	51	44,901	44,952	-51.02	0.66
kennedy.xls	Excel Spreadsheet	8,237,948	1,802,450	10,040,398	5,020,183	34	9,267,540	9,267,574	-12.50	0.89
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	2,581,863	45	5,138,339	5,138,384	-50.51	0.66
news	USENET batch file	3,016,870	1,546,897	4,563,767	2,281,862	45	4,542,195	4,542,240	-50.56	0.66
obj1	Object code for VAX	172,028	60,703	232,731	116,353	27	220,619	220,646	-28.26	0.78
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	1,355,112	49	2,696,858	2,696,907	-36.59	0.73
paper1	Technical paper	425,286	216,497	641,783	320,871	43	638,049	638,092	-50.04	0.67
paper2	Technical paper	657,590	336,453	994,043	497,000	45	989,541	989,586	-50.49	0.66
paper3	Technical paper	372,206	191,713	563,919	281,940	41	560,187	560,228	-50.52	0.66
paper4	Technical paper	106,286	54,559	160,845	80,402	43	160,305	160,348	-50.86	0.66
paper5	Technical paper	95,630	48,451	144,081	72,019	45	143,432	143,477	-50.03	0.67
paper6	Technical paper	304,838	153,543	458,381	229,171	41	457,161	457,202	-49.98	0.67
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**2,081,299**	**26**	**2,605,623**	**2,605,649**	**36.01**	**1.56**
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	2,899,904	43	5,779,446	5,779,489	-49.93	0.67
progc	Source code in "C"	316,886	159,267	476,153	238,057	41	478,914	478,955	-51.14	0.66
progl	Source code in LISP	573,166	288,491	861,657	430,807	45	865,572	865,617	-51.02	0.66
progp	Source code in PASCAL	395,031	192,299	587,330	293,643	46	584,815	584,861	-48.05	0.68
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**2,081,299**	**26**	**2,605,623**	**2,605,649**	**36.01**	**1.56**
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	626,164	40	1,307,693	1,307,733	-63.47	0.61
SHA1SUM	SPARC Executable	6,886	3,389	10,275	5,117	43	10,160	10,203	-48.17	0.67
sum	SPARC Executable	305,919	97,635	403,554	201,765	26	389,571	389,597	-27.35	0.79
trans	Transcript of terminal session	749,559	372,119	1,121,678	560,827	26	1,130,813	1,130,839	-50.87	0.66
world192.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	14,854,695	43	29,615,968	29,616,011	-49.67	0.67
xargs.1	GNU manual page	33,814	17,701	51,515	25,736	45	51,355	51,400	-52.01	0.66

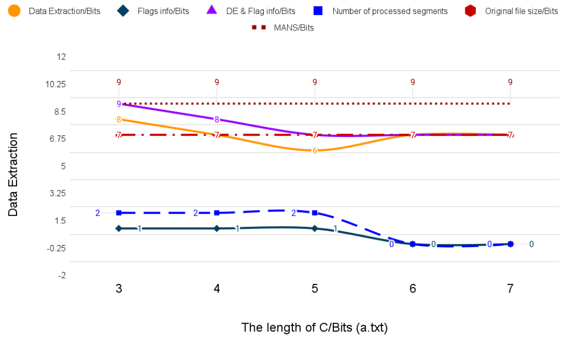
Original File			MANS		C=5 - Solution I - Flag Information					
File Name	**Category**	**File size/Bits**	**Number of switches/Bits**	**MANS/Bits**	**Number of processed segments**	**DE/Bits**	**Flags info/Bits**	**Total/Bits**	**Compression %**	**Ratio**
a.txt	The letter 'a'	7	2	9	2	6	1	7	0.00	1.00
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	399,977	69	1,094,870	1,094,939	-36.87	0.73
alice29.txt	English text	1,216,708	604,973	1,821,681	607,205	69	1,669,684	1,669,753	-37.24	0.73
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	410,233	70	1,129,870	1,129,940	-41.24	0.71
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	505,397	71	1,391,778	1,391,849	-38.99	0.72
bible.txt	Bibliography (refer format)	32,379,135	16,162,217	48,541,352	16,180,429	68	44,485,025	44,485,093	-37.39	0.73
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	16,180,429	68	44,485,025	44,485,093	-37.39	0.73
book1	Fiction book	6,150,166	3,116,671	9,266,837	3,088,923	71	8,493,095	8,493,166	-38.10	0.72
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	2,460,102	72	6,772,595	6,772,667	-38.59	0.72
cp.html	HTML source	196,822	98,909	295,731	98,556	66	270,771	270,837	-37.61	0.73
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	18,554,737	69	49,124,127	49,124,196	-32.38	0.76
fields.c	C source	89,198	44,627	133,825	44,588	64	122,996	123,060	-37.96	0.72
geo	Geophysical data	819,199	277,687	1,096,886	365,612	53	989,203	989,256	-20.76	0.83
grammar.lsp	LISP source	29,766	14,837	44,603	14,843	77	41,049	41,126	-38.16	0.72
kennedy.xls	Excel Spreadsheet	8,237,948	1,802,450	10,040,398	3,346,783	52	8,883,945	8,883,997	-7.84	0.93
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	1,721,234	70	4,734,867	4,734,937	-38.69	0.72
news	USENET batch file	3,016,870	1,546,897	4,563,767	1,521,234	68	4,191,644	4,191,712	-38.94	0.72
obj1	Object code for VAX	172,028	60,703	232,731	77,565	39	208,504	208,543	-21.23	0.82
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	903,399	77	2,456,268	2,456,345	-24.40	0.80
paper1	Technical paper	425,286	216,497	641,783	213,904	74	588,300	588,374	-38.35	0.72
paper2	Technical paper	657,590	336,453	994,043	331,324	74	910,181	910,255	-38.42	0.72
paper3	Technical paper	372,206	191,713	563,919	187,950	72	517,639	517,711	-39.09	0.72
paper4	Technical paper	106,286	54,559	160,845	53,592	72	147,560	147,632	-38.90	0.72
paper5	Technical paper	95,630	48,451	144,081	48,003	75	132,034	132,109	-38.15	0.72
paper6	Technical paper	304,838	153,543	458,381	152,774	62	419,859	419,921	-37.75	0.73
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**1,387,527**	**44**	**3,289,676**	**3,289,720**	**19.20**	**1.24**
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	1,933,260	72	5,313,683	5,313,755	-37.84	0.73
progc	Source code in "C"	316,886	159,267	476,153	158,695	71	437,736	437,807	-38.16	0.72
progl	Source code in LISP	573,166	288,491	861,657	287,196	72	792,077	792,149	-38.21	0.72
progp	Source code in PASCAL	395,031	192,299	587,330	195,754	71	539,660	539,731	-36.63	0.73
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**1,387,527**	**44**	**3,289,676**	**3,289,720**	**19.20**	**1.24**
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	417,433	70	1,160,923	1,160,993	-45.12	0.69
SHA1SUM	SPARC Executable	6,886	3,389	10,275	3,404	66	9,239	9,305	-35.13	0.74
sum	SPARC Executable	305,919	97,635	403,554	134,505	42	358,250	358,292	-17.12	0.85
trans	Transcript of terminal session	749,559	372,119	1,121,678	373,880	41	1,026,973	1,027,014	-37.02	0.73
world192.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	9,903,121	71	27,242,157	27,242,228	-37.68	0.73
xargs.1	GNU manual page	33,814	17,701	51,515	17,148	74	47,252	47,326	-39.96	0.71

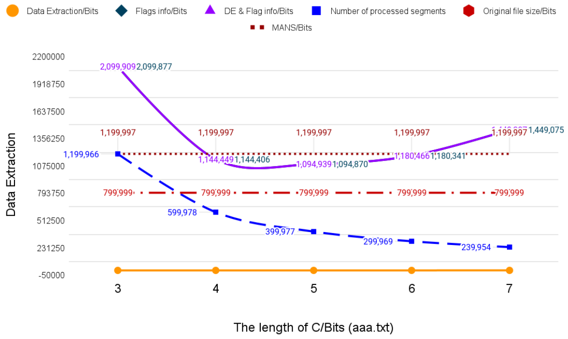
Original File			MANS		C=6 - Solution I - Flag Information					
File Name	**Category**	**File size/Bits**	**Number of switches/Bits**	**MANS/Bits**	**Number of processed segments**	**DE/Bits**	**Flags info/Bits**	**Total/Bits**	**Compression %**	**Ratio**
a.txt	The letter 'a'	7	2	9	0	7	0	7	0.00	1.00
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	299,969	125	1,180,341	1,180,466	-47.56	0.68
alice29.txt	English text	1,216,708	604,973	1,821,681	455,391	121	1,795,795	1,795,916	-47.60	0.68
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	307,662	122	1,214,584	1,214,706	-51.84	0.66
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	379,033	131	1,494,992	1,495,123	-49.30	0.67
bible.txt	Bibliography (refer format)	32,379,135	16,162,217	48,541,352	12,135,308	124	47,833,098	47,833,222	-47.73	0.68
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	12,135,308	124	47,833,098	47,833,222	-47.73	0.68
book1	Fiction book	6,150,166	3,116,671	9,266,837	2,316,678	129	9,171,829	9,171,958	-49.13	0.67
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	1,845,063	127	7,293,449	7,293,576	-49.25	0.67
cp.html	HTML source	196,822	98,909	295,731	73,905	115	291,711	291,826	-48.27	0.67
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	13,916,040	121	54,728,921	54,729,042	-47.48	0.68
fields.c	C source	89,198	44,627	133,825	33,427	121	131,330	131,451	-47.37	0.68
geo	Geophysical data	819,199	277,687	1,096,886	274,197	102	1,035,882	1,035,984	-26.46	0.79
grammar.lsp	LISP source	29,766	14,837	44,603	11,118	135	43,801	43,936	-47.60	0.68
kennedy.xls	Excel Spreadsheet	8,237,948	1,802,450	10,040,398	2,510,077	94	8,801,579	8,801,673	-6.84	0.94
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	1,290,910	133	5,092,667	5,092,800	-49.17	0.67
news	USENET batch file	3,016,870	1,546,897	4,563,767	1,140,911	127	4,503,305	4,503,432	-49.27	0.67
obj1	Object code for VAX	172,028	60,703	232,731	58,165	75	217,341	217,416	-26.38	0.79
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	677,536	131	2,605,719	2,605,850	-31.98	0.76
paper1	Technical paper	425,286	216,497	641,783	160,415	127	633,175	633,302	-48.91	0.67
paper2	Technical paper	657,590	336,453	994,043	248,480	127	981,384	981,511	-49.26	0.67
paper3	Technical paper	372,206	191,713	563,919	140,948	131	555,853	555,984	-49.38	0.67
paper4	Technical paper	106,286	54,559	160,845	40,180	129	159,305	159,434	-50.00	0.67
paper5	Technical paper	95,630	48,451	144,081	35,989	129	141,974	142,103	-48.60	0.67
paper6	Technical paper	304,838	153,543	458,381	114,565	125	453,440	453,565	-48.79	0.67
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**1,040,638**	**74**	**2,955,561**	**2,955,635**	**27.41**	**1.38**
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	1,449,930	133	5,732,241	5,732,374	-48.70	0.67
progc	Source code in "C"	316,886	159,267	476,153	119,007	129	468,862	468,991	-48.00	0.68
progl	Source code in LISP	573,166	288,491	861,657	215,384	125	850,586	850,711	-48.42	0.67
progp	Source code in PASCAL	395,031	192,299	587,330	146,801	130	572,007	572,137	-44.83	0.69
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**1,040,638**	**74**	**2,955,561**	**2,955,635**	**27.41**	**1.38**
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	313,060	130	1,257,159	1,257,289	-57.16	0.64
SHA1SUM	SPARC Executable	6,886	3,389	10,275	2,539	123	10,116	10,239	-48.69	0.67
sum	SPARC Executable	305,919	97,635	403,554	100,869	82	372,871	372,953	-21.91	0.82
trans	Transcript of terminal session	749,559	372,119	1,121,678	280,402	74	1,102,756	1,102,830	-47.13	0.68
world192.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	7,427,326	131	29,269,278	29,269,409	-47.92	0.68
xargs.1	GNU manual page	33,814	17,701	51,515	12,849	123	50,743	50,866	-50.43	0.66

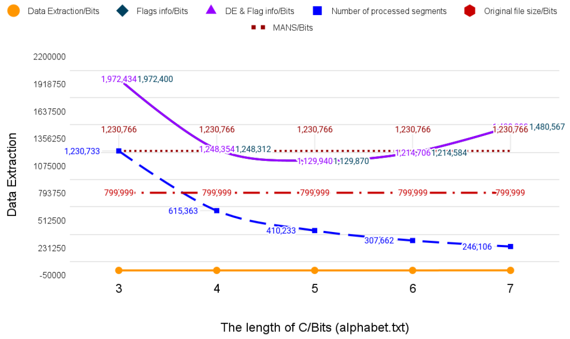
Original File			MANS		C=7 - Solution I - Flag Information						
File Name	**Category**	**File size/Bits**	**Number of switches/Bits**	**MANS/Bits**	**Number of processed segments**	**DE/Bits**	**Flags info/Bits**	**Total/Bits**	**Compression %**	**Ratio**
a.txt	The letter 'a'	7	2	9	0	7	0	7	0.00	1.00
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	239,954	232	1,449,075	1,449,307	-81.16	0.55
alice29.txt	English text	1,216,708	604,973	1,821,681	364,294	216	2,194,451	2,194,667	-80.38	0.55
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	246,106	241	1,480,567	1,480,808	-85.10	0.54
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	303,204	244	1,831,737	1,831,981	-82.94	0.55
bible.txt	Bibliography (refer format)	32,379,135	16,162,217	48,541,352	9,708,224	237	58,501,809	58,502,046	-80.68	0.55
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	9,708,224	237	58,501,809	58,502,046	-80.68	0.55
book1	Fiction book	6,150,166	3,116,671	9,266,837	1,853,321	237	11,179,607	11,179,844	-81.78	0.55
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	1,476,028	240	8,909,802	8,910,042	-82.33	0.55
cp.html	HTML source	196,822	98,909	295,731	59,102	226	356,016	356,242	-81.00	0.55
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	11,132,810	232	66,237,740	66,237,972	-78.49	0.56
fields.c	C source	89,198	44,627	133,825	26,720	230	161,053	161,283	-80.81	0.55
geo	Geophysical data	819,199	277,687	1,096,886	219,339	196	1,276,663	1,276,859	-55.87	0.64
grammar.lsp	LISP source	29,766	14,837	44,603	8,873	243	53,287	53,530	-79.84	0.56
kennedy.xls	Excel Spreadsheet	8,237,948	1,802,450	10,040,398	2,008,043	188	11,198,058	11,198,246	-35.93	0.74
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	1,032,706	244	6,234,671	6,234,915	-82.63	0.55
news	USENET batch file	3,016,870	1,546,897	4,563,767	912,708	232	5,510,086	5,510,318	-82.65	0.55
obj1	Object code for VAX	172,028	60,703	232,731	46,519	141	270,684	270,825	-57.43	0.64
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	542,006	246	3,200,664	3,200,910	-62.11	0.62
paper1	Technical paper	425,286	216,497	641,783	128,310	238	774,213	774,451	-82.10	0.55
paper2	Technical paper	657,590	336,453	994,043	198,762	238	1,200,210	1,200,448	-82.55	0.55
paper3	Technical paper	372,206	191,713	563,919	112,736	244	680,091	680,335	-82.78	0.55
paper4	Technical paper	106,286	54,559	160,845	32,121	245	193,673	193,918	-82.45	0.55
paper5	Technical paper	95,630	48,451	144,081	28,767	251	173,761	174,012	-81.96	0.55
paper6	Technical paper	304,838	153,543	458,381	91,630	236	552,319	552,555	-81.26	0.55
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**832,497**	**142**	**3,931,521**	**3,931,663**	**3.44**	**1.04**
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	1,159,923	239	7,000,115	7,000,354	-81.60	0.55
progc	Source code in "C"	316,886	159,267	476,153	95,183	243	576,240	576,483	-81.92	0.55
progl	Source code in LISP	573,166	288,491	861,657	172,284	242	1,040,531	1,040,773	-81.58	0.55
progp	Source code in PASCAL	395,031	192,299	587,330	117,417	250	707,814	708,064	-79.24	0.56
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**832,497**	**142**	**3,931,521**	**3,931,663**	**3.44**	**1.04**
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	250,428	231	1,531,701	1,531,932	-91.49	0.52
SHA1SUM	SPARC Executable	6,886	3,389	10,275	2,009	235	11,946	12,181	-76.90	0.57
sum	SPARC Executable	305,919	97,635	403,554	80,682	149	461,649	461,798	-50.95	0.66
trans	Transcript of terminal session	749,559	372,119	1,121,678	224,308	143	1,355,437	1,355,580	-80.85	0.55
world192.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	5,941,840	236	35,851,699	35,851,935	-81.19	0.55
xargs.1	GNU manual page	33,814	17,701	51,515	10,257	235	61,980	62,215	-83.99	0.54

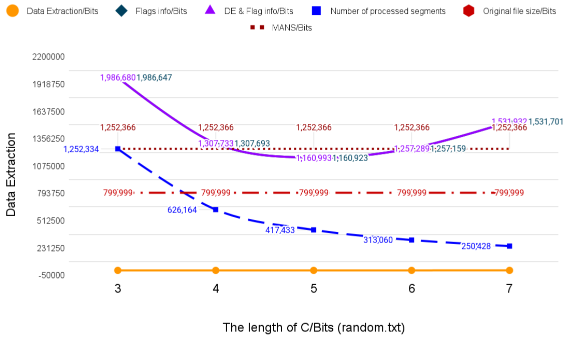
Appendix C

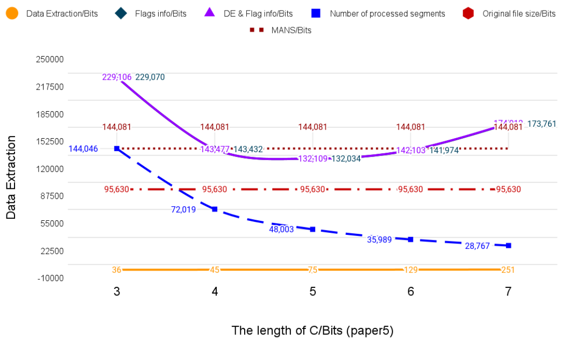
The Figures below show that all tested files for Solution I of the Canterbury Corpus follow the same pattern, the Number of processed segments decrease as C_l increases and results from Data Extraction and flag information increase as C_l increases from when $C_l \geq 5$. The lowest generated results for Data Extraction occur when $C_l = 4$ for all tested files. Compression occurs for both pic and ptt5 files by ~3.4% to ~26% across the variation of C.

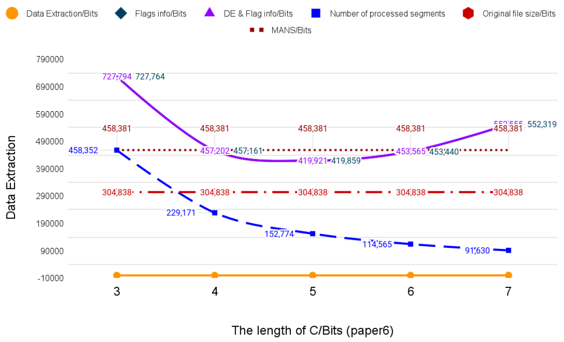


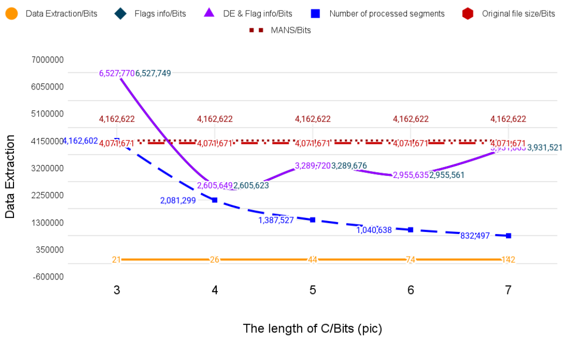


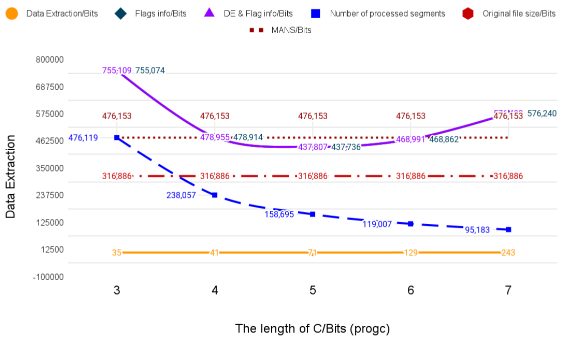


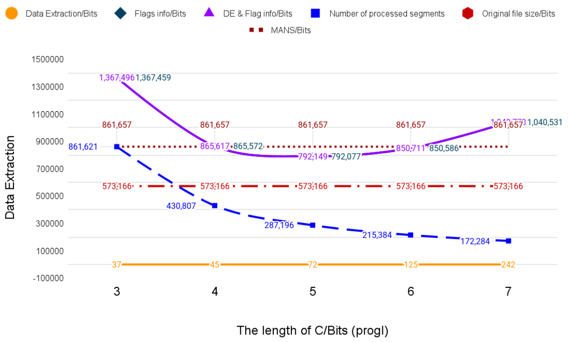


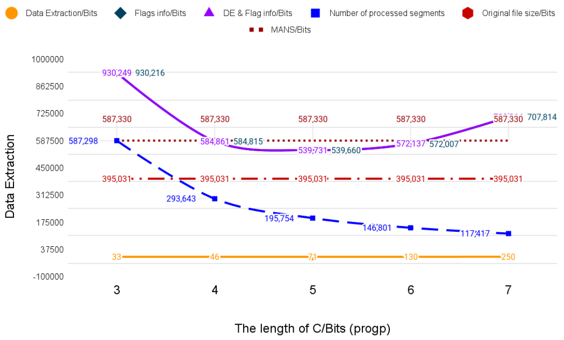

Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
250000
217500
185000
152500
120000
87500
55000
22500
-10000
229,106
229,070
144,081
144,081
144,081
144,081
144,081
173,761
144,046
143,477
143,432
132,109
132,034
142,103
141,974
95,630
95,630
95,630
95,630
95,630
72,019
48,003
35,989
28,767
36
45
75
129
251
3
4
5
6
7
The length of C/Bits (paper5)

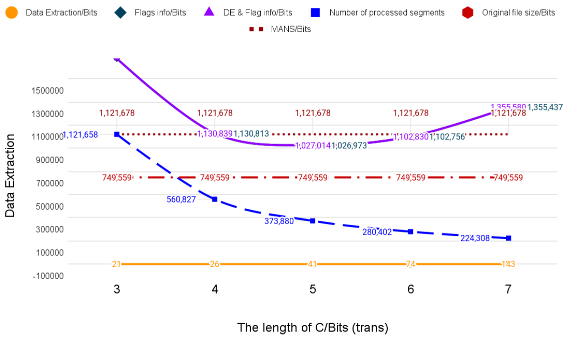

Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
790000
690000
590000
490000
390000
290000
190000
90000
-10000
727,794
727,764
458,381
458,381
458,381
458,381
458,381
552,319
458,352
457,202
457,161
419,921
419,859
453,565
453,440
304,838
304,838
304,838
304,838
304,838
229,171
152,774
114,565
91,630
3
4
5
6
7
The length of C/Bits (paper6)

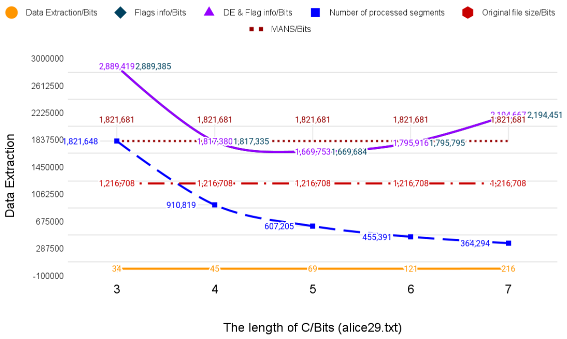

Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
7000000
6050000
5100000
4150000
3200000
2250000
1300000
350000
-600000
6,527,770
6,527,749
4,162,622
4,162,622
4,162,622
4,162,622
4,162,622
4,162,602
4,071,671
4,071,671
4,071,671
4,071,671
4,071,671
3,931,521
3,289,720
3,289,676
2,955,635
2,955,561
2,605,649
2,605,623
2,081,299
1,387,527
1,040,638
832,497
21
26
44
74
142
3
4
5
6
7
The length of C/Bits (pic)

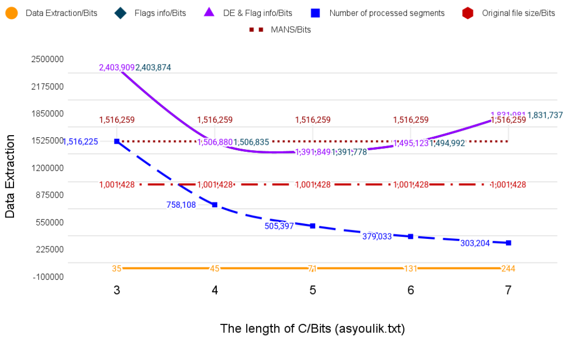

Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
800000
687500
575000
462500
350000
237500
125000
12500
-100000
755,109
755,074
476,153
476,153
476,153
476,153
476,153
576,240
476,119
478,955
478,914
437,807
437,736
468,991
468,862
316,886
316,886
316,886
316,886
316,886
238,057
158,695
119,007
95,183
35
41
71
129
243
3
4
5
6
7
The length of C/Bits (progc)

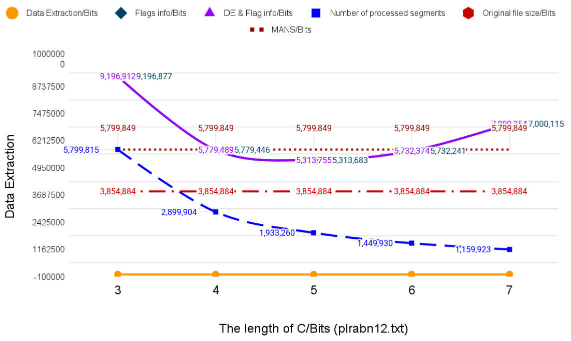

Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
1500000
1300000
1100000
900000
700000
500000
300000
100000
-100000
1,367,496
1,367,459
861,657
861,657
861,657
861,657
861,657
1,040,531
861,621
865,617
865,572
792,149
792,077
850,711
850,586
573,166
573,166
573,166
573,166
573,166
430,807
287,196
215,384
172,284
37
45
72
125
242
3
4
5
6
7
The length of C/Bits (progl)

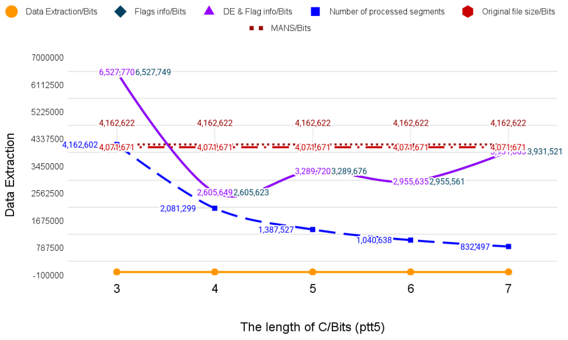

Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
1000000
862500
725000
587500
450000
312500
175000
37500
-100000
930,249
930,216
587,330
587,330
587,330
587,330
587,330
707,814
587,298
584,861
584,815
539,731
539,660
572,137
572,007
395,031
395,031
395,031
395,031
395,031
293,643
195,754
146,801
117,417
33
46
71
130
250
3
4
5
6
7
The length of C/Bits (progp)

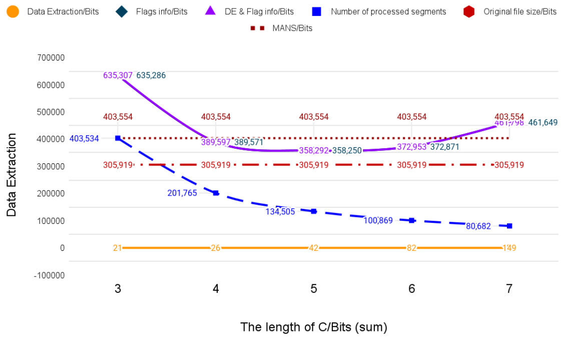

Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
1500000
1300000
1100000
900000
700000
500000
300000
100000
-100000
1,121,678
1,121,678
1,121,678
1,121,678
1,121,678
1,355,580
1,355,437
1,121,658
1,130,839
1,130,813
1,027,014
1,026,973
1,102,830
1,102,756
749,559
749,559
749,559
749,559
749,559
560,827
373,880
280,402
224,308
21
26
41
74
143
3
4
5
6
7
The length of C/Bits (trans)

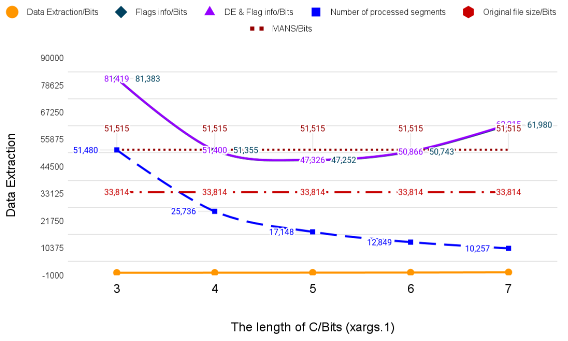

Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
3000000
2612500
2225000
1837500
1450000
1062500
675000
287500
-100000
2,889,419
2,889,385
1,821,681
1,821,681
1,821,681
1,821,681
1,821,681
2,194,451
1,821,648
1,817,380
1,817,335
1,669,753
1,669,684
1,795,916
1,795,795
1,216,708
1,216,708
1,216,708
1,216,708
1,216,708
910,819
607,205
455,391
364,294
34
45
69
121
216
3
4
5
6
7
The length of C/Bits (alice29.txt)

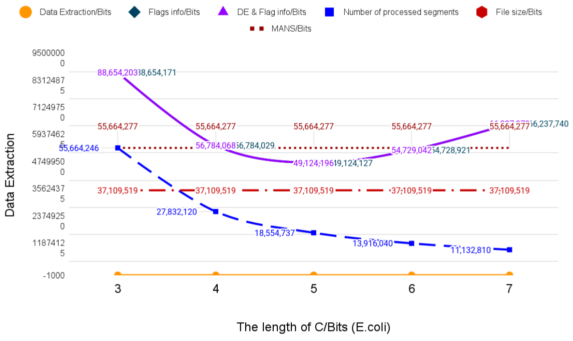

Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
2500000
2175000
1850000
1525000
1200000
875000
550000
225000
-100000
2,403,909
2,403,874
1,516,259
1,516,225
1,506,880
1,506,835
1,391,849
1,391,778
1,495,123
1,494,992
1,831,737
1,001,428
758,108
505,397
379,033
303,204
35
45
74
131
244
3
4
5
6
7
The length of C/Bits (asyoulik.txt)


Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
10000000
8737500
7475000
6212500
4950000
3687500
2425000
1162500
-100000
9,196,912
9,196,877
5,799,849
5,799,815
5,779,489
5,779,446
5,313,755
5,313,683
5,732,374
5,732,241
7,000,115
3,854,884
2,899,904
1,933,260
1,449,930
1,159,923
3
4
5
6
7
The length of C/Bits (plrabn12.txt)


Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
7000000
6112500
5225000
4337500
3450000
2562500
1675000
787500
-100000
6,527,770
6,527,749
4,162,622
4,162,602
4,071,671
3,289,720
3,289,676
2,605,649
2,605,623
2,955,635
2,955,561
3,931,521
2,081,299
1,387,527
1,040,638
832,497
3
4
5
6
7
The length of C/Bits (ptt5)


Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
700000
600000
500000
400000
300000
200000
100000
0
-100000
635,307
635,286
403,554
403,534
389,597
389,571
358,292
358,250
372,953
372,871
461,649
305,919
201,765
134,505
100,869
80,682
21
26
42
82
149
3
4
5
6
7
The length of C/Bits (sum)


Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
Original file size/Bits
MANS/Bits
Data Extraction
90000
78625
67250
55875
44500
33125
21750
10375
-1000
81,419
81,383
51,515
51,480
51,400
51,355
47,326
47,252
50,866
50,743
61,980
33,814
25,736
17,148
12,849
10,257
3
4
5
6
7
The length of C/Bits (xargs.1)


Data Extraction/Bits
Flags info/Bits
DE & Flag info/Bits
Number of processed segments
File size/Bits
MANS/Bits
Data Extraction
95000000
83124875
71249750
59374625
47499500
35624375
23749250
11874125
-1000
88,654,203
88,654,171
55,664,277
55,664,246
56,784,068
56,784,029
49,124,196
49,124,127
54,729,042
54,728,921
66,237,740
37,109,519
27,832,120
18,554,737
13,916,040
11,132,810
3
4
5
6
7
The length of C/Bits (E.coli)

Appendix D

Original file			MANS		Data Extraction - Solution II - Flag to Flag- C=3							
File Name	Category	Original file size/Bits	Number of switches/Bits	MANS/Bits	Number of processed segments	Data Extraction/ Bits	Compression before conversion	Compression %	Number of flags (f)	DE to Binary	Compression after conversion	%
a.txt	The letter 'a'	7	2	9	1	9	-2	-28.57	0	7	0	0.00
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	183,672	1,016,326	-216,327	-27.04	119,107	518,742	281,257	35.16
alice29.txt	English text	1,216,708	604,973	1,821,681	279,815	1,541,867	-325,159	-26.72	180,926	786,938	429,770	35.32
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	180,381	1,050,386	-250,387	-31.30	119,053	527,984	272,015	34.00
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	228,620	1,287,640	-286,212	-28.58	149,210	653,454	347,974	34.75
bib	Bibliography (refer format)	890,086	473,065	1,363,151	201,604	1,363,151	-473,065	-53.15	132,462	585,273	304,813	34.25
book1	Fiction book	6,150,166	3,116,671	9,266,837	1,406,231	7,860,607	-1,710,441	-27.81	915,225	4,003,131	2,147,035	34.91
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	1,116,273	6,264,103	-1,377,257	-28.18	727,703	3,184,225	1,702,621	34.84
cp.html	HTML source	196,822	98,909	295,731	45,136	250,596	-53,774	-27.32	29,333	127,752	69,070	35.09
fields.c	C source	89,198	44,627	133,825	20,316	113,510	-24,312	-27.26	13,223	58,229	30,969	34.72
geo	Geophysical data	819,199	277,687	1,096,886	202,760	894,127	-74,928	-9.15	120,934	488,016	331,183	40.43
grammar.lsp	LISP source	29,766	14,837	44,603	6,838	37,766	-8,000	-26.88	4,441	19,318	10,448	35.10
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**1,219,171**	**2,943,452**	**1,128,219**	**27.71**	**542,542**	**1,825,358**	**2,246,313**	**55.17**
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**1,219,171**	**2,943,452**	**1,128,219**	**27.71**	**573,670**	**2,505,396**	**1,566,275**	**55.17**
kennedy.xls	**Excel Spreadsheet**	**8,237,948**	**1,802,450**	**10,040,398**	**2,184,226**	**7,856,173**	**381,775**	**4.63**	**1,186,600**	**4,486,966**	**3,750,982**	**45.53**
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	778,227	4,385,543	-971,515	-28.46	508,078	2,229,430	1,184,598	34.70
news	USENET batch file	3,016,870	1,546,897	4,563,767	689,650	3,874,188	-857,318	-28.42	449,556	1,964,765	1,052,105	34.87
obj1	Object code for VAX	172,028	60,703	232,731	42,290	190,442	-18,414	-10.70	24,905	129,179	42,849	24.91
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	481,898	2,228,374	-253,885	-12.86	289,328	1,197,036	777,453	39.37
paper1	Technical paper	425,286	216,497	641,783	96,875	544,909	-119,623	-28.13	63,318	277,729	147,557	34.70
paper2	Technical paper	657,590	336,453	994,043	150,025	844,019	-186,429	-28.35	97,871	428,863	228,727	34.78
paper3	Technical paper	372,206	191,713	563,919	84,837	479,083	-106,877	-28.71	55,355	243,121	129,085	34.68
paper4	Technical paper	106,286	54,559	160,845	24,228	136,618	-30,332	-28.54	15,783	69,437	36,849	34.67
paper5	Technical paper	95,630	48,451	144,081	21,798	122,284	-26,654	-27.87	14,230	62,378	33,252	34.77
paper6	Technical paper	304,838	153,543	458,381	69,681	388,701	-83,863	-27.51	45,419	198,435	106,403	34.90
pi.txt	The first million digits of pi	7,999,998	3,800,828	11,800,826	11,800,826	9,953,360	-1,953,362	-24.42	1,186,834	5,138,891	2,861,107	35.76
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	882,612	4,917,238	-1,062,354	-27.56	573,670	2,505,396	1,349,488	35.01
progc	Source code in "C"	316,886	159,267	476,153	72,352	403,802	-86,916	-27.43	47,244	206,604	110,282	34.80
progl	Source code in LISP	573,166	288,491	861,657	131,207	730,451	-157,285	-27.44	85,531	372,668	200,498	34.98
progp	Source code in PASCAL	395,031	192,299	587,330	90,881	496,450	-101,419	-25.67	58,688	255,628	139,403	35.29
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	178,741	1,073,626	-273,627	-34.20	119,244	533,436	266,563	33.32
SHA1SUM	SPARC Executable	6,886	3,389	10,275	1,581	8,695	-1,809	-26.27	1,026	4,456	2,430	35.29
sum	SPARC Executable	305,919	97,635	403,554	81,237	322,318	-16,399	-5.36	45,516	212,991	92,928	30.38
trans	Transcript of terminal session	749,559	372,119	1,121,678	173,616	948,063	-198,504	-26.48	111,628	481,584	267,975	35.75
xargs.1	GNU manual page	33,814	17,701	51,515	7,658	43,858	-10,044	-29.70	5,070	22,197	11,617	34.36
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	7,435,023	41,106,330	-8,727,195	-26.95	4,844,342	25,826,847	6,552,288	20.24
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	8,496,073	47,168,205	-10,058,686	-27.11	5,537,354	29,633,262	7,476,257	20.15
world.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	4,537,172	25,172,260	-5,385,062	-27.21	2,945,115	12,842,324	6,944,874	35.10

Original file			MANS		Data Extraction - Solution II - Flag to Flag- C=4							
File Name	Category	Original file size/Bits	Number of switches/Bits	MANS/Bits	Number of processed segments	Data Extraction/ Bits	Compression before conversion	Compression %	Number of flags (f)	DE to Binary	Compression after conversion	%
a.txt	The letter 'a'	7	2	9	1	9	-2	-28.57	0	7	0	0%
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	115,808	968,383	-168,384	-21.05	78,436	541,008	258,991	32%
alice29.txt	English text	1,216,708	604,973	1,821,681	160,401	1,500,881	-284,173	-23.36	112,261	851,369	365,339	30%
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	108,425	1,013,978	-213,979	-26.75	75,939	560,326	239,673	30%
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	129,202	1,257,857	-256,429	-25.61	91,451	710,026	291,402	29%
bib	Bibliography (refer format)	890,086	473,065	1,363,151	114,299	1,134,555	-244,469	-27.47	81,400	634,308	255,778	29%
book1	Fiction book	6,150,166	3,116,671	9,266,837	801,479	7,663,881	-1,513,715	-24.61	565,357	4,336,553	1,813,613	29%
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	635,253	6,109,871	-1,223,025	-25.03	449,714	3,449,918	1,436,928	29%
cp.html	HTML source	196,822	98,909	295,731	25,994	243,745	-46,923	-23.84	18,207	137,983	58,839	30%
fields.c	C source	89,198	44,627	133,825	11,823	110,181	-20,983	-23.52	8,245	62,629	26,569	30%
geo	Geophysical data	819,199	277,687	1,096,886	122,963	850,962	-31,763	-3.88	78,177	519,193	300,006	37%
grammar.lsp	LISP source	29,766	14,837	44,603	4,008	36,589	-6,823	-22.92	2,790	20,747	9,019	30%
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**809,980**	**2,542,664**	**1,529,007**	**37.55**	**355,963**	**1,797,920**	**2,273,751**	**56%**
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**830,363**	**2,501,898**	**1,569,773**	**38.55**	**347,133**	**2,111,405**	**1,960,266**	**48%**
kennedy.xls	**Excel Spreadsheet**	**8,237,948**	**1,802,450**	**10,040,398**	**1,333,775**	**7,372,850**	**865,098**	**10.50**	**768,407**	**4,773,987**	**3,463,961**	**42%**
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	441,538	312,456	3,101,572	90.85	312,456	2,418,554	995,474	29%
news	USENET batch file	3,016,870	1,546,897	4,563,767	394,321	3,775,127	-758,257	-25.13	278,724	2,125,169	891,701	30%
obj1	Object code for VAX	172,028	60,703	232,731	24,726	183,281	-11,253	-6.54	15,648	127,876	44,152	26%
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	284,697	2,140,879	-166,390	-8.43	180,708	1,289,885	684,604	35%
paper1	Technical paper	425,286	216,497	641,783	55,172	531,441	-106,155	-24.96	39,130	300,799	124,487	29%
paper2	Technical paper	657,590	336,453	994,043	84,800	824,445	-166,855	-25.37	60,115	465,871	191,719	29%
paper3	Technical paper	372,206	191,713	563,919	48,029	467,863	-95,657	-25.70	33,972	263,987	108,219	29%
paper4	Technical paper	106,286	54,559	160,845	13,640	133,567	-27,281	-25.67	9,681	75,517	30,769	29%
paper5	Technical paper	95,630	48,451	144,081	12,426	119,231	-23,601	-24.68	8,789	67,582	28,048	29%
paper6	Technical paper	304,838	153,543	458,381	39,774	378,835	-73,997	-24.27	28,075	214,901	89,937	30%
pi.txt	The first million digits of pi	7,999,998	3,800,828	11,800,826	1,056,121	9,688,586	-1,688,588	-21.11	735,451	5,571,236	2,428,762	30%
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	502,386	4,795,079	-940,195	-24.39	356,573	3,071,722	783,162	20%
progc	Source code in "C"	316,886	159,267	476,153	41,609	392,937	-76,051	-24.00	29,398	223,146	93,740	30%
progl	Source code in LISP	573,166	288,491	861,657	76,580	708,499	-135,333	-23.61	53,703	400,661	172,505	30%
progp	Source code in PASCAL	395,031	192,299	587,330	53,416	480,500	-85,469	-21.64	36,951	274,037	120,994	31%
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	178,741	1,073,626	-273,627	-34.20	119,244	533,436	266,563	33%
SHA1SUM	SPARC Executable	6,886	3,389	10,275	896	8,485	-1,599	-23.22	623	4,855	2,031	29%
sum	**SPARC Executable**	**305,919**	**97,635**	**403,554**	**50,091**	**303,374**	**2,545**	**0.83**	**30,098**	**208,101**	**97,818**	**32%**
trans	Transcript of terminal session	749,559	372,119	1,121,678	103,337	915,006	-165,447	-22.07	70,822	514,103	235,456	31%
xargs.1	GNU manual page	33,814	17,701	51,515	4,310	42,897	-9,083	-26.86	3,126	24,125	9,689	29%
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	4,433,968	39,673,418	-7,294,283	-22.53	3,074,735	25,438,004	6,941,131	21%
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	5,364,511	44,935,257	-7,825,738	-21.09	3,631,592	28,746,307	8,363,212	23%
world.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	2,596,413	24,516,607	-4,729,409	-23.90	1,824,494	13,898,401	5,888,797	30%

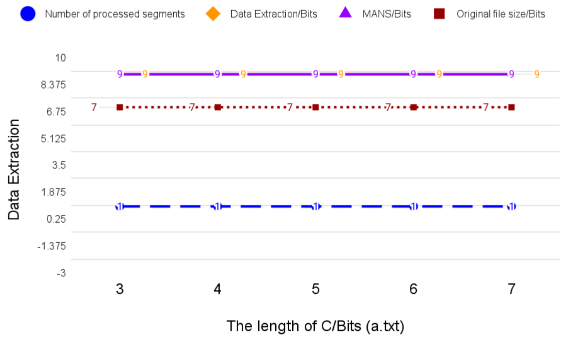
Original file			MANS		Data Extraction - Solution II - Flag to Flag- C=5							
File Name	Category	Original file size/Bits	Number of switches/Bits	MANS/Bits	Number of processed segments	Data Extraction/ Bits	Compression before conversion	Compression %	Number of flags (f)	DE to Binary	Compression after conversion	%
a.txt	The letter 'a'	7	2	9	1	9	-2	-28.57	0	7	0	0%
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	60,511	1,018,467	-218,468	-27.31	46,691	616,724	183,275	23%
alice29.txt	English text	1,216,708	604,973	1,821,681	92,741	1,543,461	-326,753	-26.86	71,320	934,329	282,379	23%
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	58,764	1,054,477	-254,478	-31.81	45,912	625,727	174,272	22%
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	74,845	1,291,727	-290,299	-28.99	57,986	776,645	224,783	22%
bib	Bibliography (refer format)	890,086	473,065	1,363,151	65,916	1,165,406	-275,320	-30.93	51,318	693,750	196,336	22%
book1	Fiction book	6,150,166	3,116,671	9,266,837	463,329	7,876,853	-1,726,687	-28.08	358,075	4,752,671	1,397,495	23%
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	366,607	6,280,557	-1,393,711	-28.52	2,833,964	3,783,911	1,102,935	23%
cp.html	HTML source	196,822	98,909	295,731	14,832	251,238	-54,416	-27.65	11,485	151,900	44,922	23%
fields.c	C source	89,198	44,627	133,825	6,700	113,728	-24,530	-27.50	5,152	69,060	20,138	23%
geo	Geophysical data	819,199	277,687	1,096,886	71,443	882,560	-63,361	-7.73	51,053	583,793	235,406	29%
grammar.lsp	LISP source	29,766	14,837	44,603	2,217	37,955	-8,189	-27.51	1,713	23,095	6,671	22%
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**535,345**	**2,556,590**	**1,515,081**	**37.21**	**254,721**	**2,012,494**	**2,059,177**	**51%**
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**541,188**	**2,539,061**	**1,532,610**	**37.64**	**248,563**	**2,249,920**	**1,821,751**	**45%**
kennedy.xls	**Excel Spreadsheet**	**8,237,948**	**1,802,450**	**10,040,398**	**834,916**	**7,535,653**	**702,295**	**8.53**	**534,877**	**5,293,146**	**2,944,802**	**36%**
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	255,382	4,397,626	-983,598	-28.81	197,716	2,646,898	767,130	22%
news	USENET batch file	3,016,870	1,546,897	4,563,767	226,899	3,883,073	-866,203	-28.71	175,651	2,333,505	683,365	23%
obj1	Object code for VAX	172,028	60,703	232,731	15,003	187,725	-15,697	-9.12	10,268	122,903	49,125	29%
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	169,536	2,201,666	-227,177	-11.51	119,618	1,421,454	553,035	28%
paper1	Technical paper	425,286	216,497	641,783	31,790	546,416	-121,130	-28.48	24,660	329,820	95,466	22%
paper2	Technical paper	657,590	336,453	994,043	49,569	845,339	-187,749	-28.55	38,373	508,375	149,215	23%
paper3	Technical paper	372,206	191,713	563,919	27,825	480,447	-108,241	-29.08	21,565	288,644	83,562	22%
paper4	Technical paper	106,286	54,559	160,845	7,962	136,962	-30,676	-28.86	6,197	82,365	23,921	23%
paper5	Technical paper	95,630	48,451	144,081	7,214	122,442	-26,812	-28.04	5,590	73,962	21,668	23%
paper6	Technical paper	304,838	153,543	458,381	22,839	389,867	-85,029	-27.89	17,702	236,067	68,771	23%
pi.txt	The first million digits of pi	7,999,998	3,800,828	11,800,826	617,309	9,948,902	-1,948,904	-24.36	470,969	6,106,167	1,893,831	24%
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	291,144	4,926,420	-1,071,536	-27.80	225,785	3,200,772	654,112	17%
progc	Source code in "C"	316,886	159,267	476,153	23,848	404,612	-87,726	-27.68	18,522	245,016	71,870	23%
progl	Source code in LISP	573,166	288,491	861,657	42,845	733,125	-159,959	-27.91	33,201	444,523	128,643	22%
progp	Source code in PASCAL	395,031	192,299	587,330	30,177	496,802	-101,771	-25.76	23,241	326,373	68,658	17%
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	57,589	1,079,602	-279,603	-34.95	45,480	631,581	168,418	21%
SHA1SUM	SPARC Executable	6,886	3,389	10,275	527	8,697	-1,811	-26.30	402	5,292	1,594	23%
sum	SPARC Executable	305,919	97,635	403,554	28,951	316,704	-10,785	-3.53	19,483	226,667	79,252	26%
trans	Transcript of terminal session	749,559	372,119	1,121,678	57,729	948,494	-198,935	-26.54	43,860	573,221	176,338	24%
xargs.1	GNU manual page	33,814	17,701	51,515	2,480	44,078	-10,264	-30.35	1,942	26,480	7,334	22%
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	2,456,550	41,171,705	-8,792,570	-27.16	1,899,407	26,825,980	5,553,155	17%
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	2,805,328	47,248,296	-10,138,777	-27.32	2,162,810	30,777,928	6,331,591	17%
world.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	1,496,969	25,218,527	-5,431,329	-27.45	115,431	15,258,879	4,528,319	23%

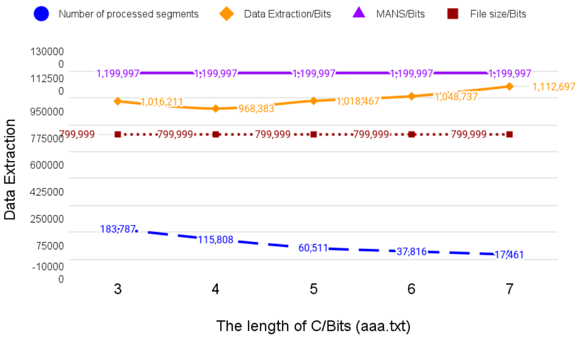
Original file			MANS		Data Extraction - Solution II - Flag to Flag- C=6							
File Name	Category	Original file size/Bits	Number of switches/Bits	MANS/Bits	Number of processed segments	Data Extraction/ Bits	Compression before conversion	Compression %	Number of flags (f)	DE to Binary	Compression after conversion	%
a.txt	The letter 'a'	7	2	9	1	9	-2	-28.57	0	7	0	0%
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	37,816	1,048,737	-248,738	-31.09	31,135	657,645	142,354	18%
alice29.txt	English text	1,216,708	604,973	1,821,681	48,998	1,625,693	-408,985	-33.61	41,561	1,029,963	186,745	15%
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	33,651	1,096,166	-296,167	-37.02	28,436	675,543	124,456	16%
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	39,194	1,359,487	-358,059	-35.75	33,426	853,860	147,568	15%
bib	Bibliography (refer format)	890,086	473,065	1,363,151	34,578	1,224,843	-334,757	-37.61	29,575	760,977	129,109	15%
book1	Fiction book	6,150,166	3,116,671	9,266,837	243,683	8,292,109	-2,141,943	-34.83	207,372	5,228,259	921,907	15%
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	191,803	6,613,167	-1,726,321	-35.33	163,479	4,163,634	723,212	15%
cp.html	HTML source	196,822	98,909	295,731	7,995	263,755	-66,933	-34.01	6,728	166,596	30,226	15%
fields.c	C source	89,198	44,627	133,825	3,641	119,265	-30,067	-33.71	3,057	75,475	13,723	15%
geo	Geophysical data	819,199	277,687	1,096,886	40,805	933,670	-114,471	-13.97	32,602	649,450	169,749	21%
grammar.lsp	LISP source	29,766	14,837	44,603	1,202	39,799	-10,033	-33.71	1,005	25,256	4,510	15%
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**366,678**	**2,695,914**	**1,375,757**	**33.79**	**206,385**	**2,246,154**	**1,825,517**	**45%**
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**357,033**	**2,734,494**	**1,337,177**	**32.84**	**192,142**	**2,502,169**	**1,569,502**	**39%**
kennedy.xls	**Excel Spreadsheet**	**8,237,948**	**1,802,450**	**10,040,398**	**479,900**	**8,120,802**	**117,146**	**1.42**	**352,865**	**6,074,183**	**2,163,765**	**26%**
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	133,765	4,628,713	-1,214,685	-35.58	113,374	2,910,581	503,447	15%
news	USENET batch file	3,016,870	1,546,897	4,563,767	119,537	4,085,623	-1,068,753	-35.43	101,724	2,565,873	450,997	15%
obj1	Object code for VAX	172,028	60,703	232,731	8,321	199,451	-27,423	-15.94	6,024	137,205	34,823	20%
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	97,285	2,321,135	-346,646	-17.56	76,851	1,570,148	404,341	20%
paper1	Technical paper	425,286	216,497	641,783	16,666	575,123	-149,837	-35.23	14,210	362,589	62,697	15%
paper2	Technical paper	657,590	336,453	994,043	25,717	891,179	-233,589	-35.52	21,947	560,586	97,004	15%
paper3	Technical paper	372,206	191,713	563,919	14,637	505,375	-133,169	-35.78	12,494	317,003	55,203	15%
paper4	Technical paper	106,286	54,559	160,845	4,114	144,393	-38,107	-35.85	3,516	90,800	15,486	15%
paper5	Technical paper	95,630	48,451	144,081	3,772	128,997	-33,367	-34.89	3,221	81,421	14,209	15%
paper6	Technical paper	304,838	153,543	458,381	11,978	410,473	-105,635	-34.65	10,239	259,574	45,264	15%
pi.txt	The first million digits of pi	7,999,998	3,800,828	11,800,826	325,891	10,497,266	-2,497,268	-31.22	274,829	6,743,662	1,256,336	16%
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	152,271	5,190,769	-1,335,885	-34.65	130,230	3,407,924	446,960	12%
progc	Source code in "C"	316,886	159,267	476,153	12,646	425,573	-108,687	-34.30	10,797	269,292	47,594	15%
progl	Source code in LISP	573,166	288,491	861,657	23,478	767,749	-194,583	-33.95	19,773	485,041	88,125	15%
progp	Source code in PASCAL	395,031	192,299	587,330	16,653	520,722	-125,691	-31.82	13,935	331,510	63,521	16%
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	34,262	1,115,322	-315,323	-39.42	28,901	675,647	124,352	16%
SHA1SUM	SPARC Executable	6,886	3,389	10,275	270	9,199	-2,313	-33.59	230	5,854	1,032	15%
sum	SPARC Executable	305,919	97,635	403,554	16,135	339,018	-33,099	-10.82	11,844	248,807	57,112	19%
trans	Transcript of terminal session	749,559	372,119	1,121,678	32,650	991,082	-241,523	-32.22	27,116	624,442	125,117	17%
xargs.1	GNU manual page	33,814	17,701	51,515	1,215	46,659	-12,845	-37.99	1,032	29,387	4,427	13%
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	1,392,275	42,972,256	-10,593,121	-32.72	1,165,651	28,267,229	4,111,906	13%
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	1,750,362	48,662,833	-11,553,314	-31.13	1,440,067	31,961,867	5,147,652	14%
world.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	792,425	26,539,735	-6,752,537	-34.13	672,187	16,781,422	3,005,776	15%

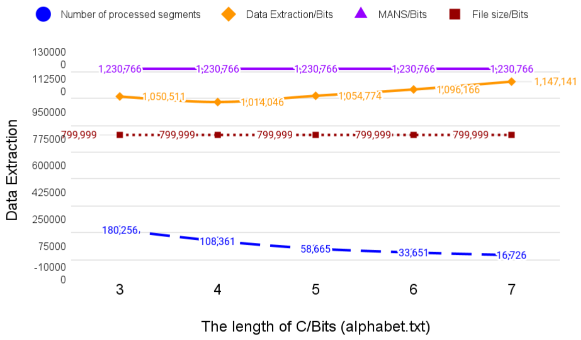
Original file			MANS		Data Extraction - Solution II - Flag to Flag- C=7							
File Name	Category	Original file size/Bits	Number of switches/Bits	MANS/Bits	Number of processed segments	Data Extraction/ Bits	Compression before conversion	Compression %	Number of flags (f)	DE to Binary	Compression after conversion	%
a.txt	The letter 'a'	7	2	9	1	9	-2	-29	0	7	0	0%
aaa.txt	The letter 'a', repeated 100,000 times.	799,999	399,998	1,199,997	17,461	1,112,697	-312,698	-39	15,721	720,837	79,162	10%
alice29.txt	English text	1,216,708	604,973	1,821,681	26,787	1,687,751	-471,043	-39	24,123	1,094,723	121,985	10%
alphabet.txt	Enough repetitions of the alphabet to fill 100,000 characters	799,999	430,767	1,230,766	16,726	1,147,141	-347,142	-43	15,170	725,921	725,921	9%
asyoulik.txt	Shakespeare	1,001,428	514,831	1,516,259	21,692	1,407,804	-406,376	-41	19,614	903,899	97,529	10%
bib	Bibliography (refer format)	890,086	473,065	1,363,151	18,819	1,269,061	-378,975	-43	17,017	806,319	83,767	9%
book1	Fiction book	6,150,166	3,116,671	9,266,837	133,430	8,599,692	-2,449,526	-40	120,306	5,547,956	602,210	10%
book2	Non-fiction book (troff format)	4,886,846	2,493,529	7,380,375	105,514	6,852,810	-1,965,964	-40	95,281	4,412,248	474,598	10%
cp.html	HTML source	196,822	98,909	295,731	4,282	274,326	-77,504	-39	3,866	177,421	19,401	10%
fields.c	C source	89,198	44,627	133,825	1,935	124,155	-34,957	-39	1,741	80,422	8,776	10%
geo	Geophysical data	819,199	277,687	1,096,886	22,004	986,871	-167,672	-20	19,088	709,041	110,158	13%
grammar.lsp	LISP source	29,766	14,837	44,603	651	41,353	-11,587	-39	585	26,836	2,930	10%
pic	**Black and white fax picture**	**4,071,671**	**90,951**	**4,162,622**	**248,199**	**2,921,632**	**1,150,039**	**28**	**157,128**	**2,554,835**	**1,516,836**	**37%**
ptt5	**CCITT test set**	**4,071,671**	**90,951**	**4,162,622**	**263,675**	**2,844,252**	**1,227,419**	**30**	**173,130**	**2,622,805**	**1,448,866**	**36%**
kennedy.xls	Excel Spreadsheet	8,237,948	1,802,450	10,040,398	287,966	8,600,573	-362,625	-4	231,988	6,664,076	1,573,872	19%
lcet10.txt	Technical writing	3,414,028	1,749,741	5,163,769	73,541	4,796,069	-1,382,041	-40	65,977	3,083,439	330,589	10%
news	USENET batch file	3,016,870	1,546,897	4,563,767	64,960	4,238,972	-1,222,102	-41	58,617	2,725,426	291,444	10%
obj1	Object code for VAX	172,028	60,703	232,731	4,894	208,266	-36,238	-21	3,508	147,142	24,886	14%
obj2	Object code for Apple Mac	1,974,489	735,782	2,710,271	55,848	2,431,036	-456,547	-23	49,600	1,689,894	284,595	14%
paper1	Technical paper	425,286	216,497	641,783	9,238	595,598	-170,312	-40	8,351	383,837	41,449	10%
paper2	Technical paper	657,590	336,453	994,043	14,055	923,773	-266,183	-40	12,725	594,384	63,206	10%
paper3	Technical paper	372,206	191,713	563,919	7,977	524,039	-151,833	-41	7,214	336,279	35,927	10%
paper4	Technical paper	106,286	54,559	160,845	2,317	149,265	-42,979	-40	2,095	95,873	10,413	10%
paper5	Technical paper	95,630	48,451	144,081	2,037	133,901	-38,271	-40	1,848	86,481	9,149	10%
paper6	Technical paper	304,838	153,543	458,381	6,509	425,841	-121,003	-40	5,873	275,600	29,238	10%
pi.txt	The first million digits of pi	7,999,998	3,800,828	11,800,826	179,788	10,901,891	-2,901,893	-36	161,274	7,171,958	828,040	10%
plrabn12.txt	Poetry	3,854,884	1,944,965	5,799,849	83,796	5,380,874	-1,525,990	-40	75,610	3,551,580	303,304	8%
progc	Source code in "C"	316,886	159,267	476,153	6,807	442,123	-125,237	-40	6,153	286,467	30,419	10%
progl	Source code in LISP	573,166	288,491	861,657	12,275	800,287	-227,121	-40	11,068	517,980	55,186	10%
progp	Source code in PASCAL	395,031	192,299	587,330	8,760	543,535	-148,504	-38	7,868	355,064	39,967	10%
random.txt	100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	799,999	452,367	1,252,366	16,309	1,170,826	-370,827	-46	14,852	728,980	71,019	9%
SHA1SUM	SPARC Executable	6,886	3,389	10,275	151	9,525	-2,639	-38	131	6,203	683	10%
sum	SPARC Executable	305,919	97,635	403,554	9,239	357,364	-51,445	-17	7,102	264,910	41,009	13%
trans	Transcript of terminal session	749,559	372,119	1,121,678	16,969	1,036,838	-287,279	-38	15,266	671,453	78,106	10%
xargs.1	GNU manual page	33,814	17,701	51,515	712	47,960	-14,146	-42	642	30,709	3,105	9%
bible.txt	The King James version of the bible	32,379,135	16,162,217	48,541,352	711,467	44,984,022	-12,604,887	-39	640,576	29,785,748	2,593,387	8%
E.coli	Complete genome of the E. Coli bacterium	37,109,519	18,554,758	55,664,277	810,660	51,610,982	-14,501,463	-39	730,357	34,163,869	2,945,650	8%
world.txt	The CIA world fact book	19,787,198	9,922,233	29,709,431	431,855	27,550,161	-7,762,963	-39	389,432	17,831,007	1,956,191	10%

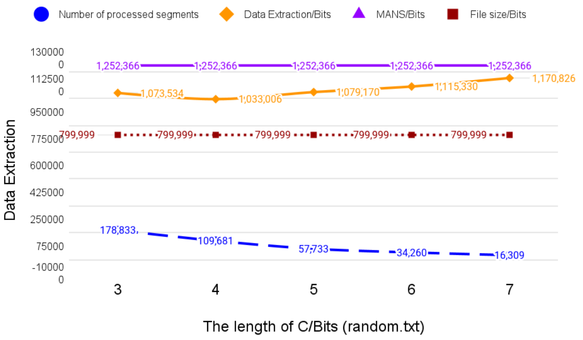
Appendix E

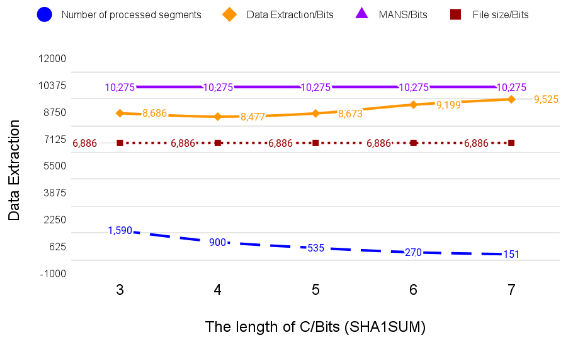
The Figures below show that all tested files for Solution II of the Canterbury Corpus follow the same pattern, the Number of processed segments decrease as C_l increases and results from Data Extraction increase as C_l increases from when $C_l \geq 5$. The lowest generated results for Data Extraction occur when $C_l = 4$ for all tested files. Compression occurs for both pic and ptt5 files by ~29% to ~38% across the variation of C, while Kennedy.xls compression results are from 1.5% to 11% the values 3, 4, 5 and 6 of C.

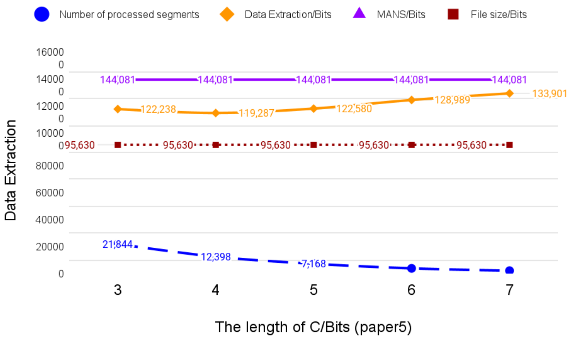


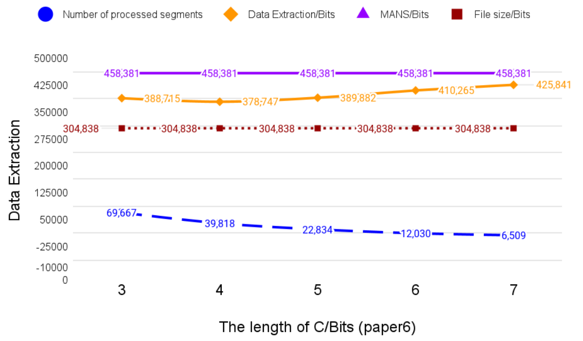




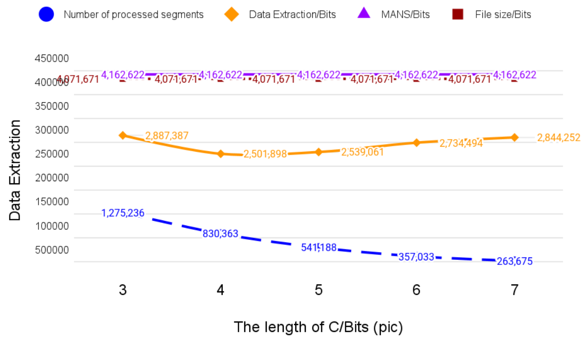




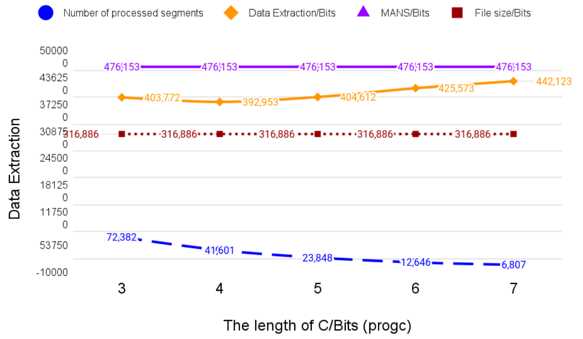




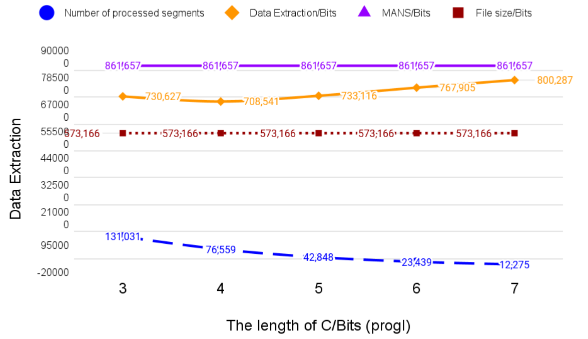
● Number of processed segments ◆ Data Extraction/Bits ▲ MANS/Bits ■ File size/Bits
Data Extraction
500000
425000
350000
275000
200000
125000
50000
-25000
-100000
458,381 458,381 458,381 458,381 458,381
388,715 378,747 389,882 410,265 425,841
304,838 304,838 304,838 304,838 304,838
69,667 39,818 22,834 12,030 6,509
3 4 5 6 7
The length of C/Bits (paper6)



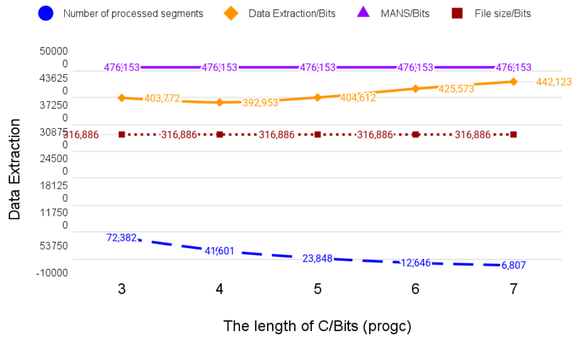
● Number of processed segments ◆ Data Extraction/Bits ▲ MANS/Bits ■ File size/Bits
Data Extraction
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
4,162,622 4,162,622 4,162,622 4,162,622 4,162,622
4,071,671 4,071,671 4,071,671 4,071,671 4,071,671
2,887,387 2,501,898 2,539,061 2,734,494 2,844,252
1,275,236 830,363 541,188 357,033 263,675
3 4 5 6 7
The length of C/Bits (pic)



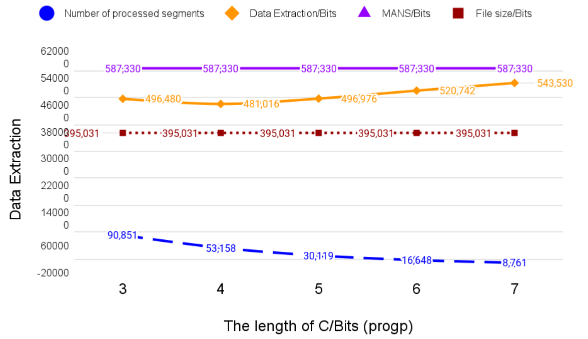
● Number of processed segments ◆ Data Extraction/Bits ▲ MANS/Bits ■ File size/Bits
Data Extraction
500000
436250
372500
308750
245000
181250
117500
53750
-10000
476,153 476,153 476,153 476,153 476,153
403,772 392,953 404,612 425,573 442,123
316,886 316,886 316,886 316,886 316,886
72,382 41,601 23,848 12,646 6,807
3 4 5 6 7
The length of C/Bits (progc)



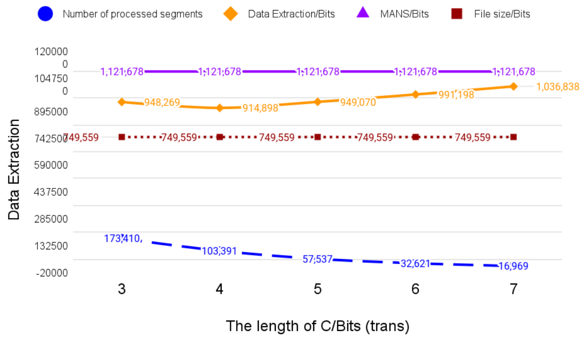
● Number of processed segments ◆ Data Extraction/Bits ▲ MANS/Bits ■ File size/Bits
Data Extraction
900000
785000
670000
555000
440000
325000
210000
95000
-20000
861,657 861,657 861,657 861,657 861,657
730,627 708,541 733,116 767,905 800,287
573,166 573,166 573,166 573,166 573,166
131,031 76,559 42,848 23,439 12,275
3 4 5 6 7
The length of C/Bits (progl)



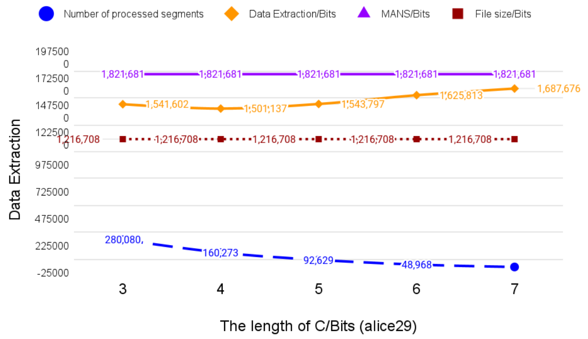
● Number of processed segments ◆ Data Extraction/Bits ▲ MANS/Bits ■ File size/Bits
Data Extraction
500000
436250
372500
308750
245000
181250
117500
53750
-10000
476,153 476,153 476,153 476,153 476,153
403,772 392,953 404,612 425,573 442,123
316,886 316,886 316,886 316,886 316,886
72,382 41,601 23,848 12,646 6,807
3 4 5 6 7
The length of C/Bits (progc)



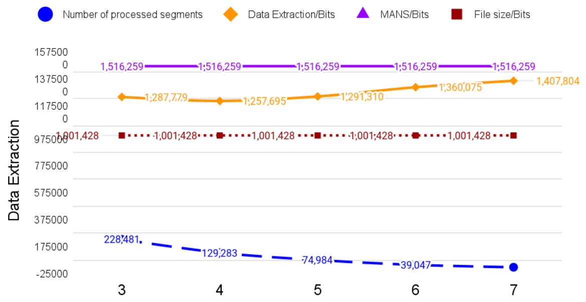
● Number of processed segments ◆ Data Extraction/Bits ▲ MANS/Bits ■ File size/Bits
Data Extraction
620000
540000
460000
380000
300000
220000
140000
60000
-20000
587,330 587,330 587,330 587,330 587,330
496,480 481,016 496,976 520,742 543,530
395,031 395,031 395,031 395,031 395,031
90,851 53,158 30,119 16,648 8,761
3 4 5 6 7
The length of C/Bits (progp)



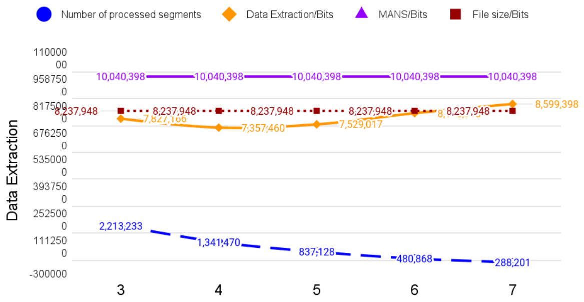
● Number of processed segments ◆ Data Extraction/Bits ▲ MANS/Bits ■ File size/Bits
Data Extraction
1200000
1047500
895000
742500
590000
437500
285000
132500
-20000
1,121,678 1,121,678 1,121,678 1,121,678 1,121,678
948,269 914,898 949,070 991,198 1,036,838
749,559 749,559 749,559 749,559 749,559
173,410 103,391 57,537 32,621 16,969
3 4 5 6 7
The length of C/Bits (trans)



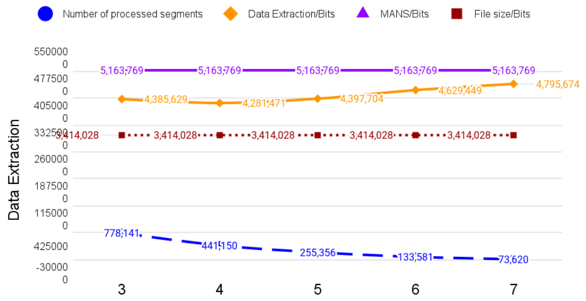
● Number of processed segments ◆ Data Extraction/Bits ▲ MANS/Bits ■ File size/Bits
Data Extraction
1975000
1725000
1475000
1225000
975000
725000
475000
225000
-25000
1,821,681 1,821,681 1,821,681 1,821,681 1,821,681
1,541,602 1,501,137 1,543,797 1,625,813 1,687,676
1,216,708 1,216,708 1,216,708 1,216,708 1,216,708
280,080 160,273 92,629 48,968
3 4 5 6 7
The length of C/Bits (alice29)



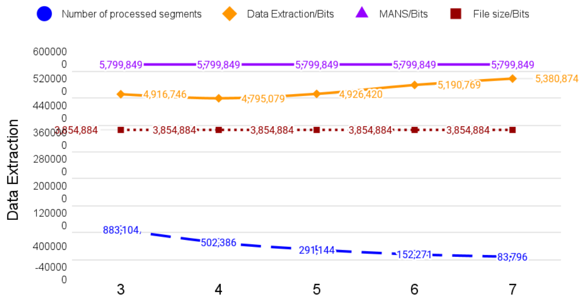
The length of C/Bits (asyoulik.txt)



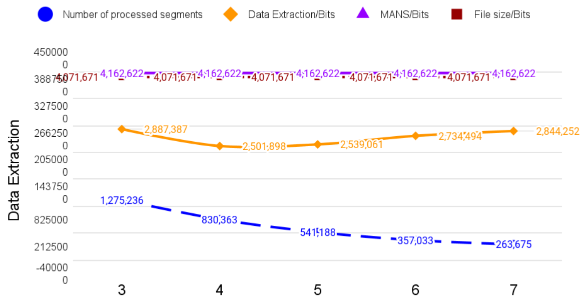
The length of C/Bits (kennedy.xls)



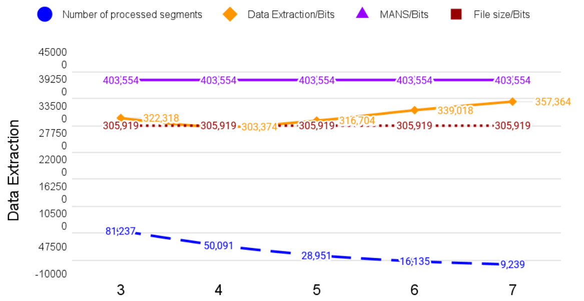
The length of C/Bits (lcet10.txt)



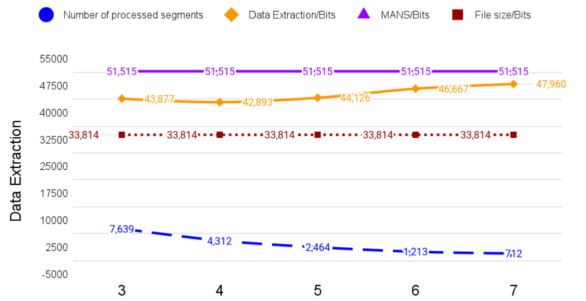
The length of C/Bits (plrabn12.txt)



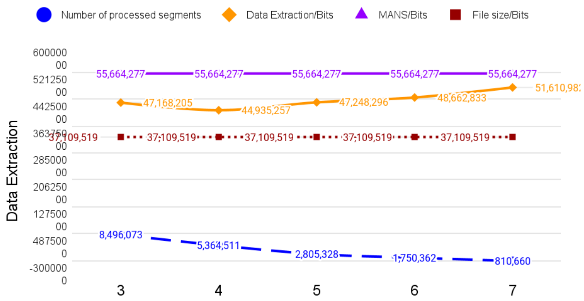
The length of C/Bits (ptt5)



The length of C/Bits (sum)



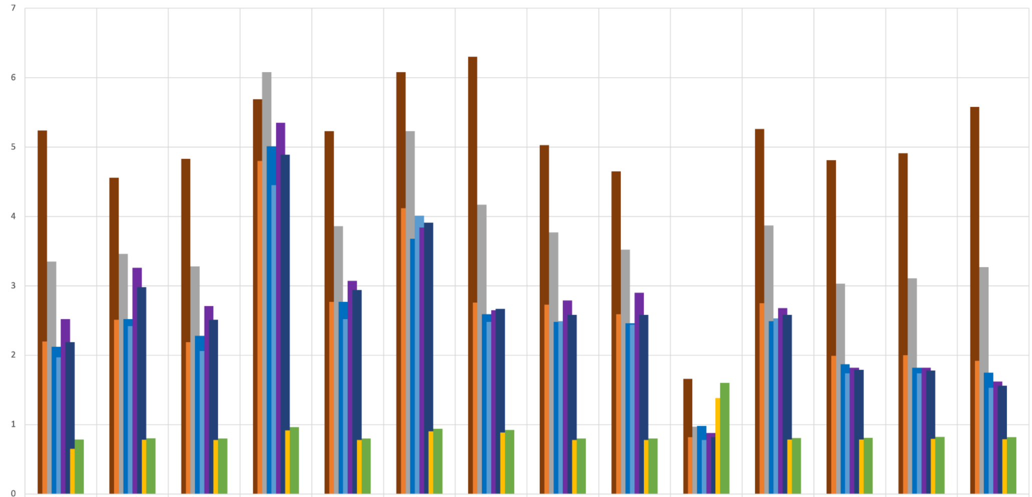
The length of C/Bits (xargs.1)



The length of C/Bits (E.coli)

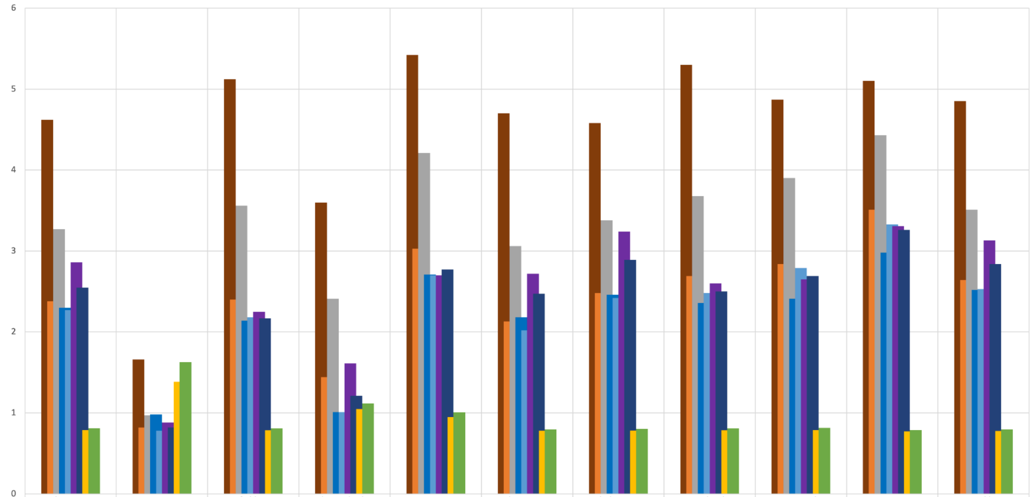
Appendix F

Compression ratio for Calgary files

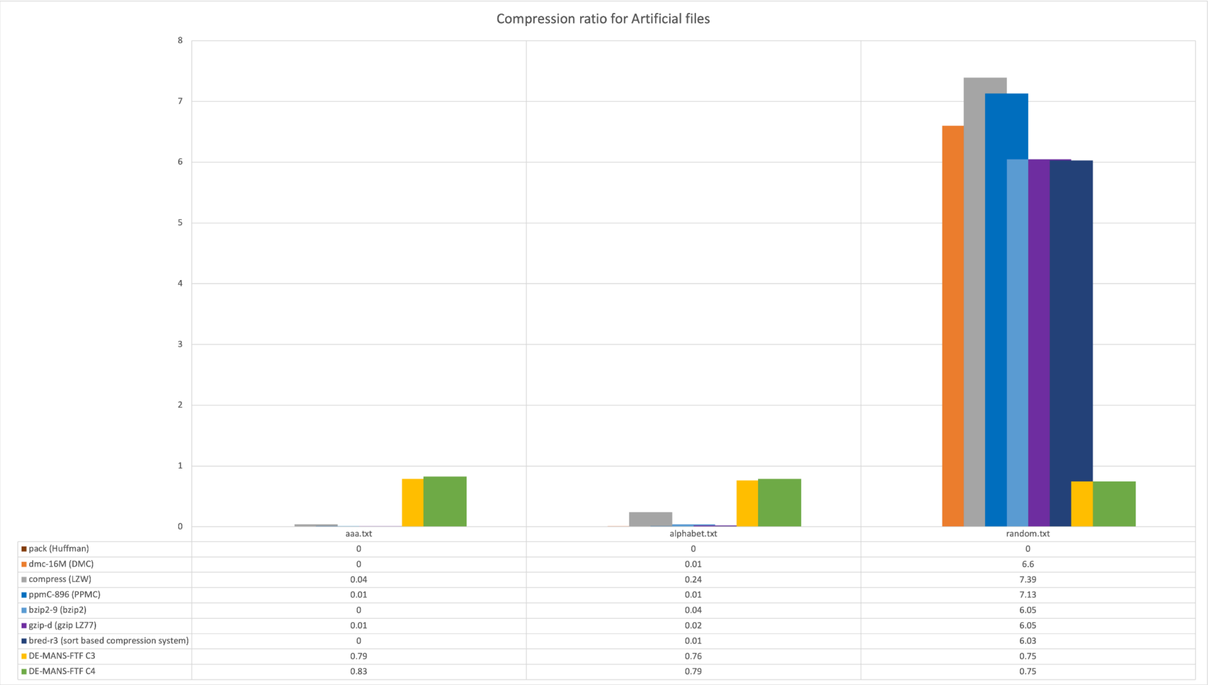


	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	progl	progp	trans
■ pack (Huffman)	5.24	4.56	4.83	5.69	5.23	6.08	6.3	5.03	4.65	1.66	5.26	4.81	4.91	5.58
■ dmc-16M (DMC)	2.2	2.51	2.19	4.8	2.77	4.12	2.76	2.73	2.59	0.82	2.75	1.99	2	1.92
■ compress (LZW)	3.35	3.46	3.28	6.08	3.86	5.23	4.17	3.77	3.52	0.97	3.87	3.03	3.11	3.27
■ ppmC-896 (PPMC)	2.12	2.52	2.28	5.01	2.77	3.68	2.59	2.48	2.46	0.98	2.49	1.87	1.82	1.75
■ bzip2-9 (bzip2)	1.97	2.42	2.06	4.45	2.52	4.01	2.48	2.49	2.44	0.78	2.53	1.74	1.74	1.53
■ gzip-d (gzip LZ77)	2.52	3.26	2.71	5.35	3.07	3.84	2.65	2.79	2.9	0.88	2.68	1.82	1.82	1.62
■ bred-r3 (sort based compression system)	2.19	2.98	2.51	4.89	2.94	3.91	2.67	2.58	2.58	0.82	2.58	1.79	1.78	1.56
■ DE-MANS-FTF C3	0.65	0.78	0.78	0.92	0.78	0.90	0.89	0.78	0.78	1.38	0.78	0.78	0.80	0.79
■ DE-MANS-FTF C4	0.78	0.80	0.80	0.96	0.80	0.94	0.92	0.80	0.80	1.60	0.81	0.81	0.82	0.82

Compression ratio for Canterbury files

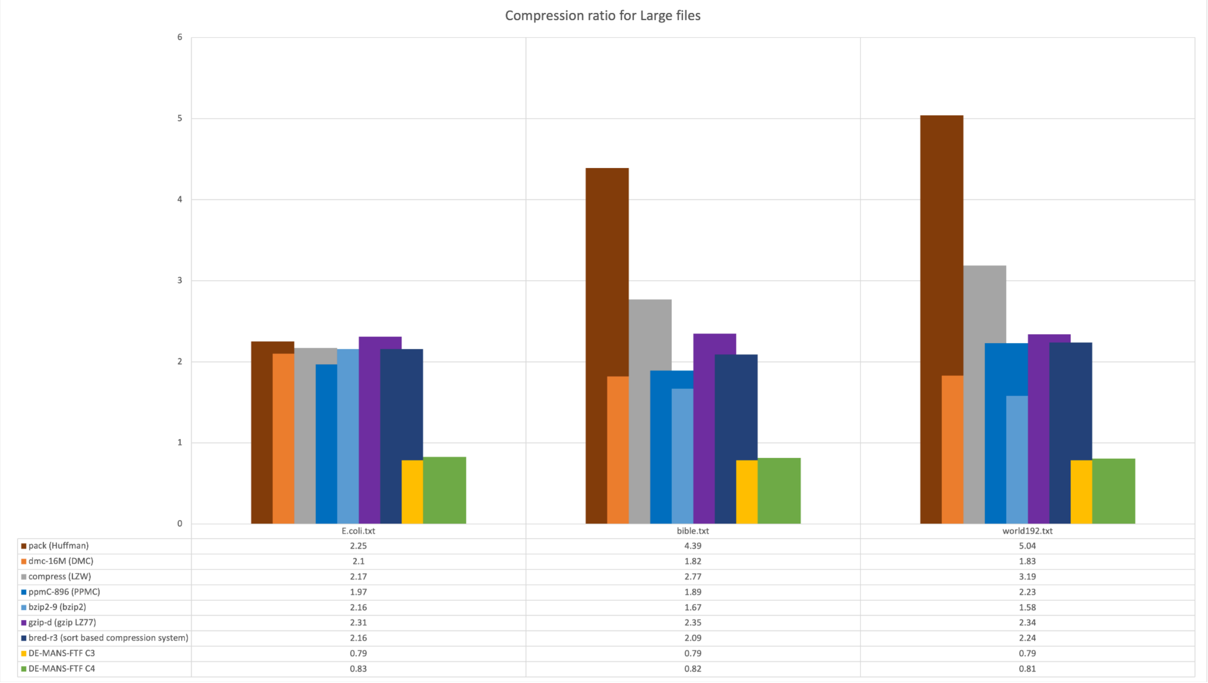


	alice29.txt	ptt5	fields.c	kennedy.xls	sum	lcet10.txt	plrabn12.txt	cp.html	grammar.lsp	xargs.1	asyoulik.txt
■ pack (Huffman)	4.62	1.66	5.12	3.6	5.42	4.7	4.58	5.3	4.87	5.1	4.85
■ dmc-16M (DMC)	2.38	0.82	2.4	1.44	3.03	2.13	2.48	2.69	2.84	3.51	2.64
■ compress (LZW)	3.27	0.97	3.56	2.41	4.21	3.06	3.38	3.68	3.9	4.43	3.51
■ ppmC-896 (PPMC)	2.3	0.98	2.14	1.01	2.71	2.18	2.46	2.36	2.41	2.98	2.52
■ bzip2-9 (bzip2)	2.27	0.78	2.18	1.01	2.7	2.02	2.42	2.48	2.79	3.33	2.53
■ gzip-d (gzip LZ77)	2.86	0.88	2.25	1.61	2.7	2.72	3.24	2.6	2.65	3.31	3.13
■ bred-r3 (sort based compression system)	2.55	0.82	2.17	1.21	2.77	2.47	2.89	2.5	2.69	3.26	2.84
■ DE-MANS-FTF C3	0.79	1.38	0.79	1.05	0.95	0.78	0.78	0.79	0.79	0.77	0.78
■ DE-MANS-FTF C4	0.81	1.63	0.81	1.12	1.01	0.80	0.80	0.81	0.81	0.79	0.80



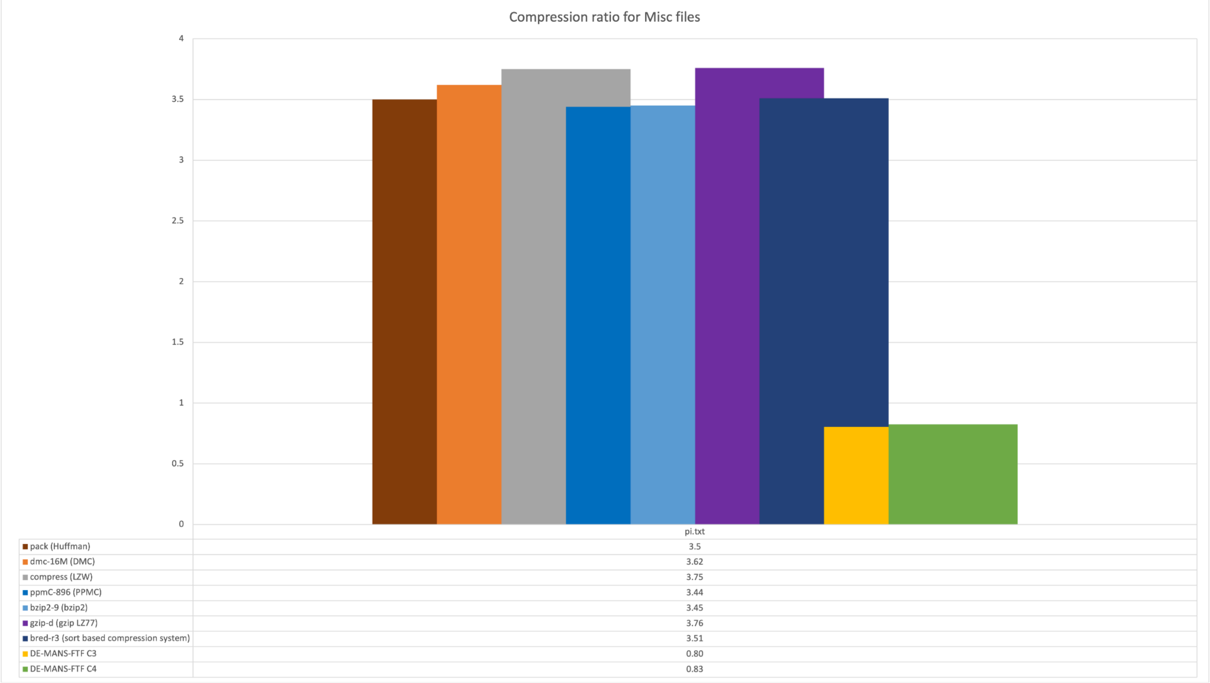
Compression ratio for Artificial files

	aaa.txt	alphabet.txt	random.txt
pack (Huffman)	0	0	0
dmc-16M (DMC)	0	0.01	6.6
compress (LZW)	0.04	0.24	7.39
ppmC-896 (PPMC)	0.01	0.01	7.13
bzip2-9 (bzip2)	0	0.04	6.05
gzip-d (gzip LZ77)	0.01	0.02	6.05
bred-r3 (sort based compression system)	0	0.01	6.03
DE-MANS-FTF C3	0.79	0.76	0.75
DE-MANS-FTF C4	0.83	0.79	0.75



Compression ratio for Large files

	E.coli.txt	bible.txt	world192.txt
pack (Huffman)	2.25	4.39	5.04
dmc-16M (DMC)	2.1	1.82	1.83
compress (LZW)	2.17	2.77	3.19
ppmC-896 (PPMC)	1.97	1.89	2.23
bzip2-9 (bzip2)	2.16	1.67	1.58
gzip-d (gzip LZ77)	2.31	2.35	2.34
bred-r3 (sort based compression system)	2.16	2.09	2.24
DE-MANS-FTF C3	0.79	0.79	0.79
DE-MANS-FTF C4	0.83	0.82	0.81



Compression ratio for Misc files

	pi.txt
■ pack (Huffman)	3.5
■ dmc-16M (DMC)	3.62
■ compress (LZW)	3.75
■ ppmC-896 (PPMC)	3.44
■ bzip2-9 (bzip2)	3.45
■ gzip-d (gzip LZ77)	3.76
■ bred-r3 (sort based compression system)	3.51
■ DE-MANS-FTF C3	0.80
■ DE-MANS-FTF C4	0.83