

**DICTIONARY- BASED TEXT COMPRESSION TECHNIQUE
USING QUATERNARY CODE**

By
Ahsan Habib

**A DISSERTATION SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY**



**Department of Computer Science and Engineering
SHAHJALAL UNIVERSITY OF SCIENCE AND TECHNOLOGY
JULY 2019**

**DICTIONARY- BASED TEXT COMPRESSION TECHNIQUE
USING QUATERNARY CODE**

By

Ahsan Habib

Supervisor

Professor Dr M Shahidur Rahman
Department of Computer Science and Engineering

Co-Supervisor

Professor Dr M Jahirul Islam
Department of Computer Science and Engineering



**Department of Computer Science and Engineering
SHAHJALAL UNIVERSITY OF SCIENCE AND TECHNOLOGY**

JULY 2019

**DICTIONARY- BASED TEXT COMPRESSION TECHNIQUE
USING QUATERNARY CODE**

**A dissertation submitted to the Department of Computer
Science and Engineering of Shahjalal University of Science and
Technology in partial fulfillment of the requirements for the
degree of Doctor of Philosophy**

Ahsan Habib

JULY 2019

Approval Page

DICTIONARY- BASED TEXT COMPRESSION TECHNIQUE USING QUATERNARY CODE

By
Ahsan Habib

2019

Signature of the Supervisor:

Professor Dr. M Shahidur Rahman
Department of Computer Science and Engineering
Shahjalal University of Science and Technology
Sylhet, Bangladesh

Signature of the Co-Supervisor:

Professor Dr. M Jahirul Islam
Head, Department of Computer Science and Engineering
Shahjalal University of Science and Technology
Sylhet, Bangladesh

Thesis Examiners' Panel

1. Professor Dr. M Shahidur Rahman Department of Computer Science and Engineering, Shahjalal University of Science and Technology, Sylhet, Bangladesh	Member (Supervisor)
2. Professor Dr. Abu Sayed Md. Latiful Hoque Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh	Member (Category: Home- Outside SUST)
3. Professor Dr. Sudipta Roy Department of Computer Science and Engineering and Dean, TSSOT, Assam University, Silchar, Assam, India	Member (Category: Foreign)

Candidate's Declaration

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma, does not contain any unlawful statements.

Signature:

Ahsan Habib

PhD Fellow

Registration number : 2013361001

Department of Computer Science and Engineering

Shahjalal University of Science and Technology, Sylhet-3114

Bangladesh.

**Dedicated to my parents,
teachers
and family members**

Acknowledgement

First of all, I would like to thank Almighty **Allah SubhanahuWaTa'ala** for giving me opportunity, determination and strength to do my research. His continuous grace and mercy was with me throughout my life and ever more during the tenure of my research.

I want to express my sincere gratitude to my supervisor, Professor Dr. M Shahidur Rahman, Department of Computer Science and Engineering, Shahjalal University of Science and Technology (SUST), Sylhet, Bangladesh for making the research as an art of exploiting new ideas and technologies on top of the existing knowledge in the field of the data compression. He provides me moral courage and excellent guideline that make it possible to complete the work. His profound knowledge and expertise in this field provide me many opportunities to learn new things to build my carrier as a researcher.

I also want to express my sincere gratitude to my co-supervisor, Professor Dr. M Jahirul Islam, Head, Department of Computer Science and Engineering, Shahjalal University of Science and Technology (SUST), Sylhet, Bangladesh. It has been an honor to be his first Ph.D. student. He has taught me, both consciously and un-consciously, how good experimental data compression is done. I appreciate all his contributions of time and ideas to make my Ph.D. experience productive and stimulating. The joy and enthusiasm he has for his research was contagious and motivational for me, even during tough times in the Ph.D. pursuit.

I also express my gratitude to Professor Dr. Mohammad Reza Selim, former Head of the Department of Computer Science and Engineering, SUST, Sylhet for providing me enough lab facilities to make necessary experiments of my research. I am also grateful to my colleagues and relatives for encouraging me to continue my research.

I also thank Professor Abdullah Al Mumin for inspirational discussions with me regarding the research experiments. I am particularly indebted to Mr Md Salah Uddin, Assistant Professor, Department of Mathematics, SUST. He helped me to analyze the mathematical structure of quaternary tree.

I gratefully acknowledge the funding sources that made my Ph.D. work possible. I was funded by the ICT Division, Ministry of Posts, Telecommunications and Information Technology, People's Republic of Bangladesh to conduct this research work.

Lastly, I would like to thank my family for all their love and encouragement. For my parents who raised me with a love of science and supported me in all my pursuits. And most of all for my loving, supportive, encouraging, and patient wife whose faithful support during the final stages of this Ph.D. is so appreciated.

Abstract

Improving encoding and decoding time in compression technique is a great demand to modern users. In bit level compression technique, it requires more time to encode or decode every single bit when a binary code is used. The existing Huffman based algorithms use binary code which slow the decoding speed. This research proposes a new compression algorithm that makes use of a variation of the classic Huffman coding: quaternary Huffman coding. Using quaternary Huffman coding, each symbol is encoded into a quaternary code stream, instead of a binary bit stream. A quaternary code stream for Huffman coding requires a shorter Huffman tree, i.e., less depth. The potential benefit of a shorter Huffman tree is less traverse time, which improves both compression and decompression throughput. In this research, we analyze the properties of quaternary Huffman tree and conclude that a quaternary Huffman tree is usually one-third of the height from a binary tree.

In this research, we develop a dictionary-based compression technique where we use a quaternary tree instead of a binary tree for construction of Human codes. Firstly, we explore the properties of quaternary tree structure mathematically for construction of Human codes. We study the terminology of new tree structure thoroughly and prove the results. Secondly, after a statistical analysis of English language; we design a variable length dictionary based on quaternary codes. Thirdly, we develop the encoding and decoding algorithms for the proposed technique. We compare the performance of the proposed technique with the existing popular techniques. The proposed technique performs better than the existing techniques with respect to decompression speed while the space requirement increases insignificantly.

Table of Contents

Approval Page	iv
Candidate's Declaration	v
Acknowledgment	vii
Abstract	ix
Table of Content	x
List of Figures	xiii
List of Tables	xv
List of Algorithms	xvi
Chapter 1: Introduction.....	1
1.1 Background and present state of the problem.....	1
1.2 Problem Definition	3
1.3 Objective with specific aims and possible outcome	4
1.4 Outcome of Methodology / Experimental Design	4
1.5 Organization of the Thesis.....	5
Chapter 2: Literature Survey.....	7
2.1 Lossy Compression Methods	8
2.1.1 Quantization.....	10
2.1.2 Predictive Coding.....	13
2.1.3 Transform Coding.....	14
2.1.4 Subband/Wavelet Coding.....	15
2.1.5 Analysis-Synthesis Schemes.....	16
2.1.6 Video Compression.....	17
2.2 Loss-Less Compression Methods	21
2.2.1 Information and Entropy	22

2.2.2	Coding	22
2.2.3	Sources with Memory.....	28
2.2.4	Predictive Coding.....	30
2.2.5	Dictionary-Based Methods.....	33
2.2.6	Statistical Encoding.....	36
2.2.7	Lightweight Compression Methods	37
2.2.8	Heavyweight Compression Methods	37
2.3	Compression on Database Processing	38
2.4	State-of-the-art discussion	38
2.5	Summary.....	39
Chapter 3: Quaternary Tree Structure.....		40
3.1	Tree	40
3.2	Quaternary Tree Architecture	41
3.2.1.	Minimization of time using quaternary tree.....	43
3.2.2	Remarks.....	44
3.2.4.	Space Requirement.....	47
3.3	Code Generation Techniques.....	48
3.3.1.	Huffman Codes to Binary Data.....	48
3.3.2.	Huffman Codes to Quaternary Data.....	50
3.3.3	Huffman Codes to Octanary Data	51
3.3.4	Huffman Codes to Hexanary Data	52
3.3.5.	Comparison among trees.....	55
3.4	Code generation algorithm.....	55
3.4.1	Encoding Algorithm.....	55
3.4.2	Decoding Algorithm.....	61

3.4.3 Why Choosing Quaternary tree to generate dictionary code	67
3.5. Summary	68
Chapter 4: Implementation	69
4.1 Dictionary Generation Algorithm	70
4.2 Encoding Algorithm	72
4.3 Decoding Algorithm	73
4.4 Summary	76
Chapter 5: Result and Discussion	77
5. Performance analysis	77
5.1. Methods	77
5.2. Test Corpora	78
5.3. Results.....	78
5.4 The reason for choosing quaternary code dictionary.....	83
5.5 Summary.....	91
Chapter 6: Conclusion and Future Research	92
5.1 Fundamental Contributions of the Thesis.....	92
5.2 Future Research	93
Bibliography.....	94
Publications	101

List of Figures

Figure 1: Quantizer with alphabet size of four.....	10
Figure 2: Samples of a signal.	11
Figure 3: Two-dimensional view of an eight-level scalar quantizer.....	11
Figure 4: Two-dimensional view of an eight-level scalar quantizer.....	12
Figure 5: Block diagram of a DPCM system.	13
Figure 6: The sensin image coded at 0.25 bits per pixel using JPEG.	14
Figure 7: A two-band subband coding scheme.	15
Figure 8: Speech synthesis model used by LPC-10.	16
Figure 9: Block diagram of the ITU-T H.261 video compression algorithm....	17
Figure 10: Possible arrangement for a GOP.....	20
Figure 11: Example of building Huffman code tree.....	25
Figure 12: Histograms of pixel values in the sensin image and of the differences between neighboring pixels.	29
Figure 13: Labeling of pixels in the neighborhood of a pixel to be encoded....	30
Figure 14: Cyclically shifted versions of the original sequence and lexicographically order set of the cyclical shifts.	32
Figure 15: Illustration for LZ77.	34
Figure 16: Tree construction	41
Figure 17: Binary tree with 16 nodes	42
Figure 18: Quaternary tree with 16 nodes	43
Figure 19: Complete binary tree.....	44
Figure 20: Complete quaternary tree.....	44
Figure 21: A complete quaternary tree.....	45
Figure 22: A canonical quaternary tree.	45

Figure 23: Construction of Binary Huffman Tree.....	49
Figure 24: Construction of Quaternary Huffman Tree.....	50
Figure 25: Construction of Octanary Huffman Tree.....	52
Figure 26: Construction of Quaternary Huffman Tree.....	53
Figure 27: Frequency distribution of Canterbury, Brown and SUPara corpora	70
Figure 28: Dictionaries.....	71
Figure 29: Compression ratio versus decompression speed for Enwik corpus	79
Figure 30: Compression ratio vs decompression speed for Canterbury corpus	80
Figure 31: Compression ratio vs decompression speed for SUPara corpus.....	81
Figure 32: Compression ratio versus decompression speed for Brown corpus	82
Figure 33: Time-space graph for Enwik corpus.....	86
Figure 34: Time-space graph for Canturbury corpus	87
Figure 35: Time-space graph for “Bht_bn”	88
Figure 36: Time-space graph for “NLP07”	89
Figure 37: Time-space graph for “Demo”.....	90

List of Tables

Table 1: Codeword generated by Huffman based algorithms	54
Table 2: Comparison of different tree structures for <i>Luke 5</i> data.....	55
Table 3: Performance analysis for Enwik corpus.....	79
Table 4: Performance analysis for Canterbury corpus.	80
Table 5: Performance analysis for SUPara corpus.	81
Table 6: Performance analysis for Brown corpus.	82
Table 7: Data set.....	84
Table 8: The compression ratio and compression-decompression speed for the Enwik Corpus	84
Table 9: The compression ratio and compression-decompression speed for the Canterbury corpus.	85
Table 10: Time-space data for Enwik corpus.....	85
Table 11: Time-space data for Enwik corpus.....	86
Table 12: Time-space data for Bht_bn file.....	87
Table 13: Time-space data for NLP07 file	88
Table 14: Time-space data for Demo file.....	89

List of Algorithms

Algorithm 1: Encoding of Quaternary Huffman Tree.....	56
Algorithm 2: Encoding of Octanary Huffman Tree.....	58
Algorithm 3: Encoding of Hexanary Huffman Tree	60
Algorithm 4: Decoding of Quaternary Huffman Tree	63
Algorithm 5: Decoding of Octanary Huffman Tree.....	64
Algorithm 6: Decoding of Hexanary Huffman Tree.....	66
Algorithm 7: Encoding Algorithm.....	72
Algorithm 8: Decoding Algorithm.....	74

Chapter 1

Introduction

Data compression is an important research area not only for saving space but also for reducing query time. Data compression is popular for accumulating data and reducing download, upload and transfer time (Khuri & Hsu, 2000). Compression and decompression speeds are very important parameter in data compression techniques. It is boring when it requires more time to compress or decompress a file. It has been revealed in the contemporary research that some algorithms achieved more compression ratio by sacrificing the processing speed, whereas some others achieved more speed by sacrificing space. In many cases, authors mainly consider decompression speed, whereas compression speed is also an important performance measuring parameter.

Huffman coding is the most popular and widely used coding system, where compression is done by assigning a shorter code to a symbol with higher frequencies. This technique assigns a unique codeword for each symbol (Huffman, 1952). Huffman based technique is used in many compression algorithms like Zip, PKzip, BZip2, and PNG. Multimedia codec such as JPEG and MP3 have a front end model and quantization followed by Huffman coding. Almost all communications with and from the internet are at some points Huffman encoded. Almost all Huffman based algorithms attempt to improve decoding speed; in most of the cases they did not achieve very good compression speed.

1.1 Background and present state of the problem

There are two types of data compression: lossy and lossless. Lossy compression usually loses some bits during decompression process; it also achieves more compression ratio than lossless compression. Lossy compression is effective when missing information does not affect the quality of decoded information. Image,

voice and video data are compressed generally by lossy compression technique. Lossy compression can reduce the number of bits by a huge amount but it cannot reproduce the original quality (Carus & Mesut, 2010). On the other hand, lossless data compression technique is used for sensitive data as like text. Lossless compression technique ensures 100% reproduction of original data.

Lossless compression techniques are of two types : dictionary based and statistical based. Dictionary based techniques typically use a variable length code for each symbol in the dictionary. Lempel-Ziv (Ziv & Lempel, 1977; Ziv & Lempel, 1978) and their variants (Storer & Szymanski, 1982; Welch, 1984) are most well-known dictionary-based techniques. On the other hand, statistical-based compression techniques produce codeword based on the statistical occurrence of a symbol in a file. Lossless text compression usually replaces an original symbol with a shorter symbol. The content of text in compression can be seen as a pattern of syllables or words (Moffat & Isal, 2005; L'ansk'y & Zemlička, 2005). Transformation of text to syllables or words is a difficult process because the number of syllables or words is undefined and is always a high number. Moreover, it is also required for all syllables or words to be transmitted to the decoder (Adiego & de la Fuente, 2006; Dvorsky *et al.*, 1999; L'ansk'y & Zemlička, 2006). As a solution, bit level compression is used in text compression, where each character has a specific binary representation (Al-Bahadili & Rababa, 2007) which is called codes. Bit level compression occasionally ensures the maximum compression ratio.

Huffman algorithmic program is very popular and widely used lossless compression tool among statistical and bit level text compression techniques. In 1952, DA Huffman presented his most cited and important data compression algorithm (Huffman, 1952), where codeword is assigned to each symbol in such a way that no two symbols have the same codeword and the starting and ending point of a symbol can be easily recognized without requiring any additional information. This is also known as prefix-free coding.

After Huffman algorithm was proposed, in addition to compressing text data, the algorithm was proved efficient in image and video compression as well (Chung,

1997). After studying the sibling property and level of trees, authors claim that the codeword length of both Huffman and Shannon-Fano has similar interpretation (Schack, 1994). The limitation of the research is that an error of a few hundred bits in the length of a typical record and the entropies are of the order of 2^{80} bits. In another research the connection between the self information of a symbol and its codeword length in a Huffman code is investigated (Katona & Nemetz, 1978). The limitation of the study is that it is hard to attribute comparable significance to the self information of an individual symbol.

Fenwick argued that, the Huffman codes cannot ameliorate the code efficiency all the time and when moving from the lower extension to the higher, the performance is always nose-diving (Fenwick, 1995). Instead of most familiar fixed-to-variable code, variable-to-variable code was also used to implement Huffman algorithm (Kavousianos *et al.*, 2008). By transforming a basic Huffman tree to a recursive one, and then using it to decode more than one symbol at a time was a very interesting technique proposed by Lin et al. (2012). The proposed approach required large memory and only suits for test data compression problems. His primary aim was to improve the efficiency of a Huffman tree.

Google Inc. introduces a new compression technique known as Zopfli (Alakuijala & Vandevenne, 2018), which is currently one of the best available compression techniques. Internally Zopfli also uses Huffman coding as a basis of its compression technique. Though Zopfli achieves highest compression speed but it is only designed for browser; where decompression is not required.

1.2 Problem Definition

It is observed that the length of Huffman code affect the compression ratio and decoding speed. Therefore, we produce coding system in a new fashion. In this research, we introduce quaternary tree to produce more efficient and optimal code to ensure the maximum decoding speed. The quaternary tree is a 4 ary tree or a tree with at most 4 children. This research proposes a new compression algorithm that makes use of a variation of the classic Huffman coding: quaternary Huffman

coding. Using quaternary Huffman coding, each symbol is encoded into a quaternary code stream, instead of a binary bit stream. A quaternary code stream for Huffman coding requires a shorter Huffman tree, i.e., less depth. The potential benefit of a shorter Huffman tree is less traverse time, which could improve both compression and decompression throughput. Experimental results show that, the decompression speed of the proposed technique is better than widely used existing techniques, whereas the space requirement is slightly increased.

1.3 Objective with specific aims and possible outcome

The objectives of the research are to:

- a) develop a new dictionary based on quaternary code,
- b) develop an encoding algorithm by using quaternary code based dictionary,
- c) develop an algorithm to decode the compressed file, and
- d) analyze the performance of the proposed system in terms of both storage and decoding time.

1.4 Outcome of Methodology / Experimental Design

Quaternary tree or 4-ary tree is a tree in which each node has 0 to 4 children (labeled as LEFT child, LEFT MID child, RIGHT MID child, RIGHT child). Here for constructing codes for quaternary Huffman tree we use 00 for left child, 01 for left mid child, 10 for right mid child, and 11 for right child. Experimental design will be carried out using following steps:

Step 1: Study the Significance of Using Quaternary Tree – The different mathematical parameters are discussed, and the comparison of Binary tree with Quaternary structured are shown.

Step 2: Develop Dictionary Generation Algorithm – After study of English language; a new static dictionary is created using quaternary code which is produced

by using quaternary tree.

Step 3: Develop Encoding Algorithm – Using the static dictionary data may be encoded simply by replacing each symbol with its code.

Step 4: Develop Decoding Algorithm – During the decoding process the dictionary are divided into four normalized dictionaries. Encoded data is decoded using these dictionaries.

Step 5: Performance Analysis - The objective of the experimental work is to verify the applicability and feasibility of the proposed technique. The experimental evaluation has been performed with real data. The experimental results are compared with popular existing techniques. Our target is to justify query time and the storage requirements in comparison with regular Huffman based techniques and other widely used techniques.

1.5 Organization of the Thesis

In chapter 2, a survey of the research in compression methods and query processing in information systems is presented. We have developed a new compression technique using a new dictionary based on quaternary Huffman code. The Huffman principles and tree structure are discussed.

Chapter 3 presents the overview of our proposed architecture, Quaternary tree architecture. The different mathematical parameters of Quaternary tree are also discussed. A remark is also proved in this section. The Binary, Octanary, and Hexanary tree structures are also discussed in this chapter. The theoretical performance is also compared.

Chapter 4 gives the overview of implementation technique. Firstly, a dictionary generation technique is explained. Secondly, the encoding algorithm is described, and finally, decoding algorithm of the proposed technique is explained.

Chapter 5 describes the experimental work that has been carried out. Results obtained are thoroughly discussed.

Chapter 6 presents conclusions and suggestions for future work.

Chapter 2

Literature Survey

Data compression involves the development of a compact representation of information. Most representations of information contain large amounts of redundancy. Redundancy can exist in various forms. It may exist in the form of correlation: spatially close pixels in an image are generally also close in value. The redundancy might be due to context: the number of possibilities for a particular letter in a piece of English text is drastically reduced if the previous letter is a q . It can be probabilistic in nature: the letter e is much more likely to occur in a piece of English text than the letter q . It can be a result of how the information-bearing sequence was generated: voiced speech has a periodic structure. Or, the redundancy can be a function of the user of the information: when looking at an image we cannot see above a certain spatial frequency; therefore, the high-frequency information is redundant for this application. Redundancy is defined by the *Merriam-Webster Dictionary* as “the part of the message that can be eliminated without the loss of essential information.” Therefore, one aspect of data compression is redundancy removal. Characterization of redundancy involves some form of modeling. Hence, this step in the compression process is also known as modeling. For historical reasons another name applied to this process is decorrelation.

After the redundancy removal process the information needs to be encoded into a binary representation. At this stage we make use of the fact that if the information is represented using a particular alphabet some letters may occur with higher probability than others. In the coding step we use shorter code words to represent letters that occur more frequently, thus lowering the average number of bits required to represent each letter.

Compression in all its forms exploits structure, or redundancy, in the data to achieve a compact representation. The design of a compression algorithm involves understanding the types of redundancy present in the data and then developing strategies for exploiting these redundancies to obtain a compact representation of the data. People have come up with many ingenious ways of characterizing and using the different types of redundancies present in different kinds of technologies from the telegraph to the cellular phone and digital movies.

One way of classifying compression schemes is by the model used to characterize the redundancy. However, more popularly, compression schemes are divided into two main groups: lossless compression and lossy compression. Lossless compression preserves all the information in the data being compressed, and the reconstruction is identical to the original data. In lossy compression some of the information contained in the original data is irretrievably lost. The loss in information is, in some sense, a payment for achieving higher levels of compression. We begin our examination of data compression schemes by first looking at lossless compression techniques.

Based on how the input data is treated during compression, we can categorize the compression techniques as lightweight or heavyweight scheme. Lightweight scheme compresses a sequence of values. Heavyweight scheme compresses a sequence of bytes. This scheme is based on patterns found in the data, ignoring the boundaries between values, and treating the input data as an array of bytes.

2.1 Lossy Compression Methods

All real world measurement of audio-visual data inherently contains a certain amount of noise. If the compression method includes a small amount of additional noise, no harm is done. Compression techniques that result in this sort of degradation are called lossy. This phenomenon is important because lossy compression techniques can give greater compression ratio over the lossless methods. The higher the compression ratio, the more noise added to the

data. Lossy compression is advantageous for image, voice and video data because the additional noise has little effect on the user's perception. JPEG (Joint Photographic Expert Group) and MPEG (Moving Picture Expert Group) are standards for compression of image, voice and video data using lossy compression methods.

The requirement that no information be lost in the compression process puts a limit on the amount of compression we can obtain. The lowest number of bits per sample is the entropy of the source. This is a quantity over which we generally have no control. In many applications this requirement of no loss is excessive. For example, there is high-frequency information in an image which cannot be perceived by the human visual system. It makes no sense to preserve this information for images that are destined for human consumption. Similarly, when we listen to sampled speech we cannot perceive the exact numerical value of each sample. Therefore, it makes no sense to expend coding resources to preserve the exact value of each speech sample. In short, there are numerous applications in which the preservation of all information present in the source output is not necessary. For these applications we relax the requirement that the reconstructed signal be identical to the original. This allows us to create compression schemes that can provide a much higher level of compression. However, it should be kept in mind that we generally pay for higher compression by increased information loss. Therefore, we measure the performance of the compression system using two metrics. We measure the amount of compression as before; however, we also measure the amount of distortion introduced by the loss of information. The measure of distortion is generally some variant of the mean squared error. If at all possible, it is more useful to let the application define the distortion.

In this section we describe a number of compression techniques that allow loss of information, hence the name lossy compression. We begin with a look at quantization which, in one way or another, is at the heart of all lossy compression schemes.

2.1.1 Quantization

Quantization is the process of representing the output of a source with a large (possibly infinite) alphabet with a small alphabet. It is a many-to-one mapping and therefore irreversible. Quantization can be performed on a sample-by-sample basis or it can be performed on a group of samples. The former is called scalar quantization and the latter is called vector quantization. We look at each in turn.

2.1.1.1 Scalar Quantization

Let us, for the moment, assume that the output alphabet of the source is all or some portion of the real number line. Thus, the size of the source alphabet is infinite. We would like to represent the source output using a finite number of code words M . The quantizer consists of two processes: an encoding process that maps the output of the source into one of the M code words, and a decoding process that maps each code word into a reconstruction value. The encoding process can be viewed as a partition of the source alphabet, while the decoding process consists of obtaining representation values for each partition. An example for a quantizer with an alphabet size of four is shown in Figure 1.

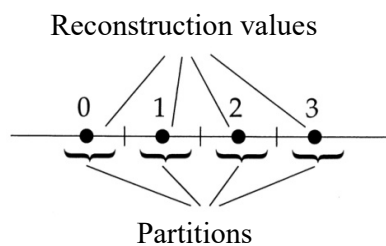


Figure 1 Quantizer with alphabet size of four.

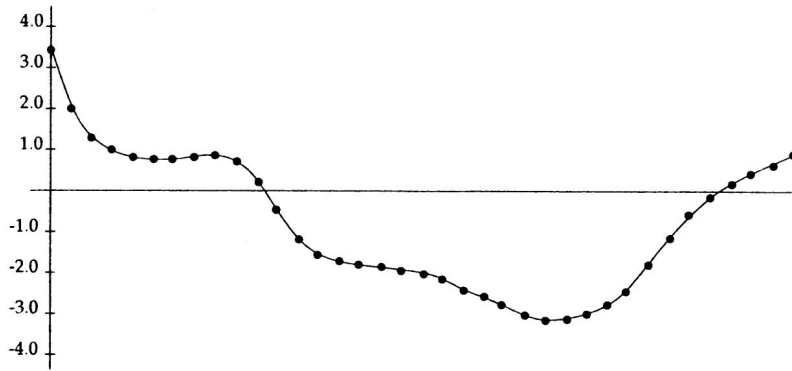


Figure 2 Samples of a signal.

2.1.1.2. Vector Quantization

The idea of representing groups of samples rather than individual samples has been present since Shannon's original papers (Shannon, 1948). There are several advantages to representing sequences. Consider the samples of the signal shown in Figure 2. The values vary approximately between -4 and 4. We could quantize these samples with an eight-level scalar quantizer with $\Delta=1$. So the reconstruction values would be $\{\pm\frac{1}{2}, \pm\frac{3}{2}, \pm\frac{5}{2}, \pm\frac{7}{2}\}$.

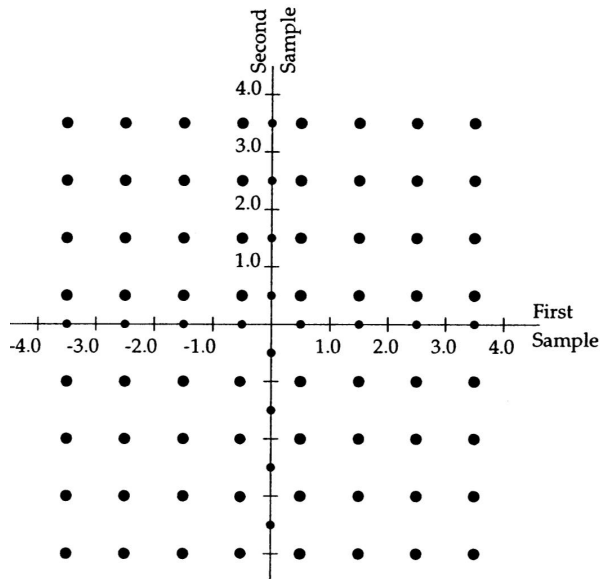


Figure 3 Two-dimensional view of an eight-level scalar quantizer.

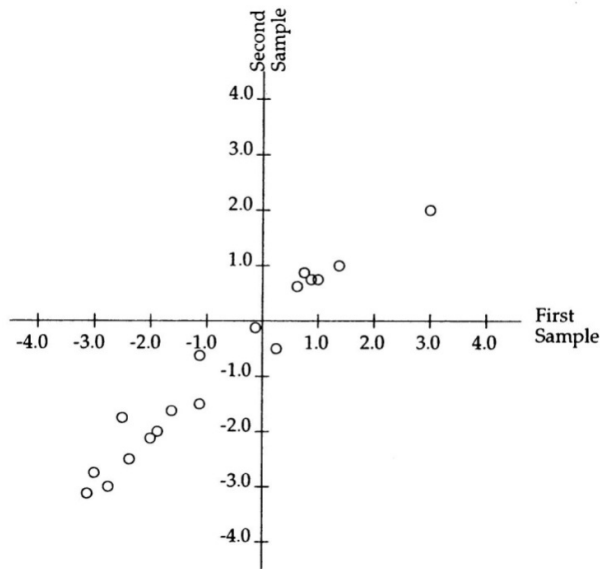


Figure 4 Two-dimensional view of an eight-level scalar quantizer.

If we were to use a fixed length code we would need three bits to represent the eight quantizer outputs. If we look at the output of the quantizer in pairs, we get 64 possible reconstruction values. These are represented by the larger filled circles in Figure 3. However, if we plot the samples of the signal as pairs (as in Figure 4) we see that the samples are clustered along a line in the first and third quadrants. This is due to the fact that there is a high degree of correlation between neighboring samples, which means that in two dimensions the samples will cluster around the $y = x$ line. Looking from a two dimensional point of view, it makes much more sense to place all the 64 output points of the quantizer close to the $y = x$ line. For a fixed length encoding, we would need six bits to represent the 64 different quantizer outputs. As each quantizer output is a representation of two samples, we would end up with three bits per sample.

Therefore, for the same number of bits, we would get a more accurate representation of the input and, therefore, incur less distortion. We pay for this decrease in distortion in several different ways. The first is through an increase in the complexity of the encoder. The scalar quantizer has a very simple encoder. In the case of the two dimensional quantizer, we need to block the input samples into “vectors” and then compare them against all the possible

quantizer output values. For three bits per sample and two dimensions this translates to 64 possible compares. However, for the same number of bits and a block size, or vector dimension, of 10, the number of quantizer outputs would be $2^{3 \times 10}$ which is 1,073,741,824! As it generally requires a large block size to get the full advantage of a vector quantizer, this means that the rate at which a vector quantizer (VQ) operates (i.e., bits per sample) is usually quite low.

2.1.2 Predictive Coding

If we have a sequence with sample values that vary slowly as in the signal shown in Figure 2, knowledge of the previous samples gives us a lot of information about the current sample. This knowledge can be used in a number of different ways. One of the earliest attempts at exploiting this redundancy was in development of differential pulse code modulation (DPCM) (Cutler, 1952). A version of DPCM is the algorithm used in the International Telecommunication Union (ITU) standard G.726 for speech coding.

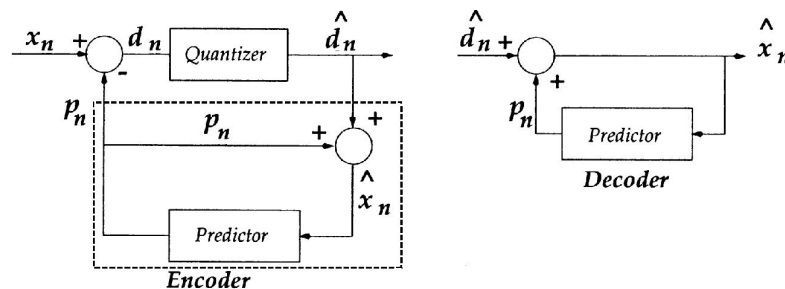


Figure 5 Block diagram of a DPCM system.

The DPCM system consists of two blocks as shown in Figure 5. The function of the predictor is to obtain an estimate of the current sample based on the reconstructed values of the past sample. The difference between this estimate, or prediction, and the actual value is quantized, encoded, and transmitted to the receiver. The decoder generates an estimate identical to the encoder, which is then added on to generate the reconstructed value. The requirement that the prediction algorithm use only the reconstructed values is to ensure that the prediction at both the encoder and the decoder are identical. The reconstructed

values used by the predictor, and the prediction algorithm, are dependent on the nature of the data being encoded. For example, for speech coding the predictor often uses the immediate past several values of the sequence, along with a sample that is a pitch period away, to form the prediction. In image compression the predictor may use the same set of pixels used by the JPEG-LS algorithm to form the prediction.



Figure 6 The sensin image coded at 0.25 bits per pixel using JPEG.

2.1.3 Transform Coding

Transform coding first became popular in the early 1970s as a way of performing vector quantization (Huang & Schultheiss, 1963). Transform coding consists of three steps. The data to be compressed is divided into blocks, and the data in each block is transformed to a set of coefficients. The transform is selected to compact most of the energy into as few coefficients as possible. The coefficients are then quantized with different quantizers for each coefficient. Finally, the quantizer labels are encoded.

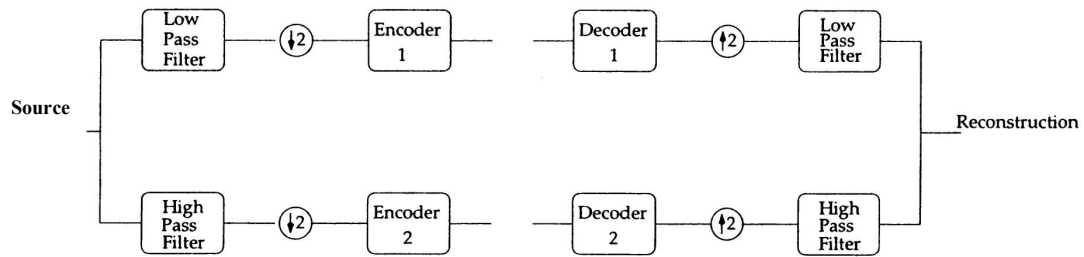


Figure 7 A two-band subband coding scheme.

2.1.4 Subband/Wavelet Coding

Transform coding at low rates tends to give the reconstructed image a blocky appearance. The image in Figure 6 has been coded at 0.25 bits per pixel using the JPEG algorithm. The blocky appearance is clearly apparent. This has led to the increasing popularity of subband and wavelet-based schemes. The implementation for both subband and wavelet-based schemes is similar. The input is filtered through a bank of filters, called the *analysis filterbank*. The filters cover the entire frequency range of the signal. As the bandwidth of each filter is only a fraction of the bandwidth of the original signal, the Nyquist criterion dictates that the number of samples required at the output of the filter be less than the number of samples per second required at the input of the filter. The output of the filters is subsampled or decimated and encoded. The decimated output values are quantized, and the quantization labels are encoded. At the decoder, after the received samples are decoded they are upsampled by the insertion of zeros between the received samples and filtered using a bank of reconstruction filters. A two-band subband coding scheme is shown in Figure 7. The major components of the design of subband coding schemes are the selection of the filters and the encoding method used for the subbands. In order to determine the latter, it may be necessary to allocate a predetermined bit budget between the various bands.

Notice that in the system shown in Figure 7 if the filters are not ideal filters then at least one of the two analysis filters will have a bandwidth greater than half

the bandwidth of the source output. If the source is initially sampled at the Nyquist rate, then when we subsample by two that particular filter output will effectively be sampled at less the Nyquist rate, thus introducing aliasing. One of the objectives of the design of the analysis and synthesis filterbanks is to remove the effect of aliasing.

2.1.5 Analysis-Synthesis Schemes

When possible, one of the most effective means of compression is to transmit instructions on how to reconstruct the source rather than transmitting the source samples. In order to do this we should have a fairly good idea about how the source samples were generated. One particular source for which this is true is human speech.

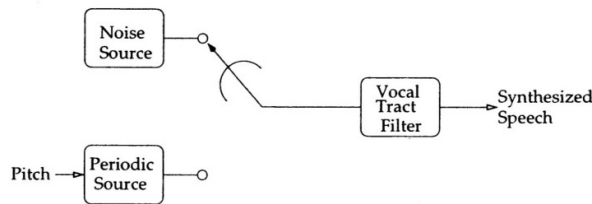


Figure 8 Speech synthesis model used by LPC-10.

Human speech can be modeled as the output of a linear filter which is excited by either white noise or a periodic input or a combination of the two. One of the earliest modern compression algorithms made use of this fact to provide a very high compression of speech. The technique, known as linear predictive coding, has its best known embodiment in the (now outdated) U.S. Government standard LPC-10. Some of the basic aspects of this standard are still alive, albeit in modified form in today's standards.

The LPC-10 standard assumes a model of speech pictured in Figure 8. The speech is divided into frames. Each frame is classified as voiced or unvoiced. For the voiced speech the pitch period for the speech sample is extracted. The parameters of the vocal tract filter are also extracted and quantized. All this information is sent to the decoder. The decoder synthesizes the speech samples

y_n as

$$y_n = \sum_{i=0}^M b_i y_{y-i} + G\epsilon_n$$

where b_i is the coefficient of the vocal tract filter. The input to the filter, the sequence $\{\epsilon_n\}$, is either the output of a noise generator or a periodic pulse train, where the period of the pulse train is the pitch period.

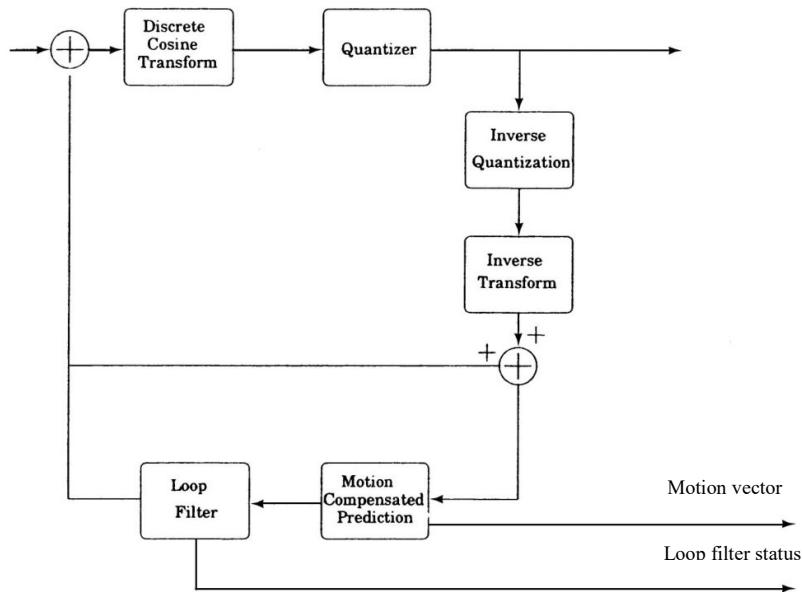


Figure 9 Block diagram of the ITU-T H.261 video compression algorithm.

2.1.6 Video Compression

Currently, the source that requires the most resources in terms of bits and, therefore, has benefitted the most from compression is video. We can think of video as a sequence of images. With this view video compression becomes repetitive image compression and we can compress each frame separately. This is the point of view adopted by M-JPEG, or motion JPEG, in which each frame

is compressed using the JPEG algorithm.

However, we know that in most video sequences there is a substantial amount of correlation between frames. It is much more efficient to send differences between the frames rather than the frames themselves. This idea is the basis for several international standards in video compression. In the following we briefly look at some of the compression algorithms used in these standards. Note that the standards contain much more than just the compression algorithms.

The ITU H.261 and its descendant ITU H.263 are international standards developed by the ITU, which is a part of the United Nations organization. A block diagram of the H.261 video coding algorithm is shown in Figure 9. The image is divided into blocks of size 8X8. The previous frame is used to predict the values of the pixels in the block being encoded. As the objects in each frame may have been offset from the previous frame, the block in the identical location is not always used. Instead the block of size 8X8 in the previous frame which is closest to the block being encoded in the current frame is used as the predicted value. In order to reduce computations the search area for the closest match is restricted to lie within a prescribed region around the location of the block being encoded. This form of prediction is known as *motion compensated prediction*. The offset of the block used for prediction from the block being encoded is referred to as the *motion vector* and is transmitted to the decoder. The loop filter is used to prevent sharp transitions in the previous frame from generating high frequency components in the difference.

The difference is encoded using transform coding. The DCT is used followed by uniform quantization. The DC coefficient is quantized using a scalar quantizer with a step size of 8. The other coefficients are quantized with 1 of 31 other quantizers, all of which are midread quantizers, with step sizes between 2 and 62. The selection of the quantizer depends in part on the availability of transmission resources. If higher compression is needed (fewer bits available), a larger step size is selected. If less compression is acceptable, a smaller step size

is selected. The quantization labels are scanned in a zigzag fashion and encoded in a manner similar to (though not the same as) JPEG.

The coding algorithm for ITU-T H.263 is similar to that used for H.261 with some improvements. The improvements include:

- Better motion compensated prediction
- Better coding
- Increased number of formats
- Increased error resilience

There are a number of other improvements that are not essential for the compression algorithm. As mentioned before, there are two versions of H.263. As the earlier version is a subset of the latter one; we only describe the later version.

The prediction is enhanced in H.263 in a number of ways. By interpolating between neighboring pixels in the frame being searched, a “larger image” is created. This essentially allows motion compensation displacement of half pixel steps rather than integer number of pixels. There is an unrestricted motion vector mode that allows references to areas outside the picture, where the outside areas are generated by duplicating the pixels at the image boundaries. Finally, in H.263 the prediction can be generated by a frame that is not the previous frame. An independent segment decoding mode allows the frame to be broken into segments where each segment can be decoded independently. This prevents error propagation and also allows for greater control over quality of regions in the reconstruction. The H.263 standard also allows for bidirectional prediction.

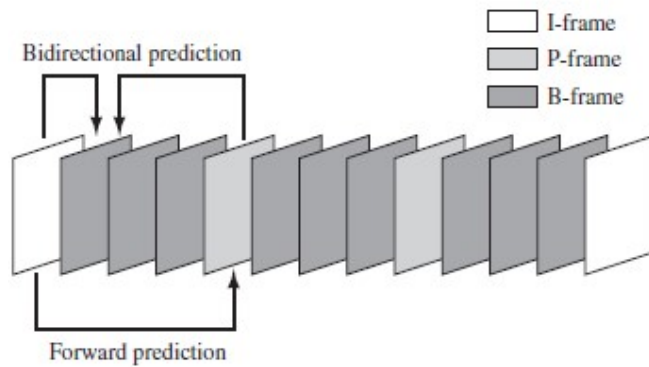


Figure 10 Possible arrangement for a GOP.

As the H.261 algorithm was designed for video conferencing, there was no consideration given to the need for random access. The MPEG standards incorporate this need by requiring that at fixed intervals a frame of an image be encoded without reference to past frames. The MPEG-1 standard defines three different kinds of frames: *I* frames, *P* frames, and *B* frames. An *I* frame is coded without reference to previous frames, that is, no use is made of prediction from previous frames. The use of the *I* frames allows random access. If such frames did not exist in the video sequence, then to view any given frame we would have to decompress all previous frames, as the reconstruction of each frame would be dependent on the prediction from previous frames. The use of periodic *I* frames is also necessary if the standard is to be used for compressing television programming. A viewer should have the ability to turn on the television (and the MPEG decoder) at more or less any time during the broadcast. If there are periodic *I* frames available, then the decoder can start decoding from the first *I* frame. Without the redundancy removal provided via prediction the compression obtained with *I* frames is less than would have been possible if prediction had been used.

The *P* frame is similar to frames in the H.261 standard in that it is generated based on prediction from previous reconstructed frames. The similarity is closer to H.263, as the MPEG-1 standard allows for half pixel shifts during motion compensated prediction.

The *B* frame was introduced in the MPEG-1 standard to offset the loss of compression efficiency occasioned by the *I* frames. The *B* frame is generated using prediction from the previous *P* or *I* frame and the nearest future *P* or *I* frame. This results in extremely good prediction and a high level of compression. The *B* frames are not used to predict other frames, therefore, the *B* frames can tolerate more error. This also permits higher levels of compression.

The various frames are organized together in a *group of pictures* (GOP). A GOP is the smallest random access unit in the video sequence. Therefore, it has to contain at least one *I* frame. Furthermore, the first *I* frame in a GOP is either the first frame of the GOP or is preceded by *B* frames which use motion compensated prediction only from this *I* frame. A possible GOP is shown in Figure 10. Notice that in order to reconstruct frames 2, 3, and 4, which are *B* frames, we need to have the *I* and *P* frames. Therefore, the order in which these frames are transmitted is different from the order in which they are displayed.

The MPEG-2 standard extends the MPEG-1 standard to higher bit rates, bigger picture sizes, and interlaced frames. Where MPEG-1 allows half pixel displacements, the MPEG-2 standard requires half pixel displacements for motion compensation. Furthermore, the MPEG-2 standard contains several additional modes of prediction. A full description of any of these standards is well beyond the scope of this research. For details the readers are referred to the standards ISO/IEC IS 11172, 13818, and 14496 and books Gibson *et al.* (1998), Mitchell *et al.* (1997), and Sayood (2000).

2.2 Loss-Less Compression Methods

Lossless compression involves finding a representation which will exactly represent the source. There should be no loss of information, and the decompressed, or reconstructed, sequence should be identical to the original sequence. The requirement that there be no loss of information puts a limit on how much compression we can get. We can get some idea about this limit by looking at some concepts from information theory.

2.2.1 Information and Entropy

In one sense it is impossible to denote anyone as the parent of data compression: people have been finding compact ways of representing information since the dawn of recorded history. One could argue that the drawings on the cave walls are representations of a significant amount of information and therefore qualify as a form of data compression. Significantly less controversial would be the characterizing of the Morse code as a form of data compression. Samuel Morse took advantage of the fact that certain letters such as *e* and *a* occur more frequently in the English language than *q* or *z* to assign shorter code words to the more frequently occurring letters. This results in lower average transmission time per letter. The first person to put data compression on a sound theoretical footing was Claude E. Shannon (1948). He developed a quantitative notion of information that formed the basis of a mathematical representation of the communication process.

The creation of a binary representation for the source output, or the generation of a *code* for a source, is the topic of the next section.

2.2.2 Coding

In the second example of the previous section the way we obtained an efficient representation was to use fewer bits to represent letters that occurred more frequently - the same idea that Samuel Morse had. It is a simple idea, but in order to use it we need an algorithm for systematically generating variable length code words for a given source. David Huffman (1952) created such an algorithm for a class project. We describe this algorithm below. Another coding algorithm which is fast gaining popularity is the arithmetic coding algorithm. We will also describe the arithmetic coding algorithm in this section.

2.2.2.1 Huffman Coding

Huffman began with two rather obvious conditions on the code and then added

a third that allowed for the construction of the code. The conditions were:

1. The codes corresponding to the higher probability letters could not be longer than the code words associated with the lower probability letters.
2. The two lowest probability letters had to have code words of the same length.

He added a third condition to generate a practical compression scheme.

3. The two lowest probability letters have codes that are identical except for the last bit.

It is easy to visualize these conditions if we think of the code words as the path through a binary tree: a zero bit denoting a left branch and a one bit denoting a right branch. The two lowest probability letters would then be the two leaves of a node at the lowest level of the tree. We can consider the parent node of these leaves as a letter in a reduced alphabet which is obtained as a combination of the two lowest probability symbols. The probability of the letter corresponding to this node would be the sum of the probabilities of the two individual letters. Now, we can find the two lowest probability symbols of the reduced alphabet and treat them as the two leaves of a common node. We can continue in this fashion until we have completed the tree. We can see this process best through an example (Skibiński, 2006).

2.2.2.1.1 Semi-adaptive Huffman coding

A *prefix code* has a property that code for no symbol is a prefix of the code for another symbol. The Huffman coding algorithm generates, from a set of probabilities, optimal prefix codes, which belongs to a family of codes with a variable codeword length. Prefix property of Huffman codes assures us that they can be correctly decoded despite being variable length. The Huffman coding algorithm (Huffman, 1952) is named after its inventor, David Huffman, who developed this algorithm as a student in a class on information theory at MIT in 1950 (Huffman code, 2005).

The Huffman algorithm builds a prefix code on the binary alphabet $\{0,1\}$, which corresponds to a binary tree in which each inner node has a left and a right child, labeled '0' and '1' respectively. Leaves of the code tree are labeled by the input symbols. Each node has a weight, which is the frequency of the symbol's appearance. A path from a root of tree to each leaf corresponds to Huffman code for each symbol. Picture 4) on Figure 11 presents binary tree created by Huffman algorithm. For example, the symbol 'd' corresponds to Huffman code '001' (left, left, right).

The Huffman algorithm is simple and can be described in terms of creating a Huffman code tree. The procedure for building this tree is following:

- a) Start with a list of free nodes, where each node corresponds to each symbol.
- b) Select two free nodes with the lowest frequency from the list.
- c) Created a parent node for two nodes found in b) with a frequency equal to the sum of the two child nodes.
- d) Remove the two nodes found in b) from the list of free nodes, and add the parent node created in c) to the list.
- e) Repeat the process starting from b) until only a single tree remains.

Figure 11 illustrates an example of building the Huffman tree. The algorithm starts with a list of nodes 'a', 'b', 'c', and 'd', with frequencies 4, 1, 3, and 2 respectively, what one can see on the picture 1). In the next step, the algorithm selects two free nodes with the lowest frequency, this is, 'b' and 'd'. Then, it creates a parent node for these nodes, with cumulative frequency 3, what illustrates the picture 2). In the following step, there are free nodes with frequency 3, 3, and 4, so the algorithm selects two first nodes, what can be observed on picture 3). In the last step, there are two free nodes, which have a frequency 6 and 4. They are joined into a root of tree, which has cumulative frequency equal to 10.

For an alphabet of k symbols procedure of building the Huffman tree requires $k-1$ steps as complete binary tree with k leaves has $k-1$ inner nodes, and each step creates one inner node. If we use a heap as the list of free nodes,

we can select nodes with the lowest frequency in $O(\log_2 k)$ time and the algorithm will run in $O(k \log_2 k)$ time.

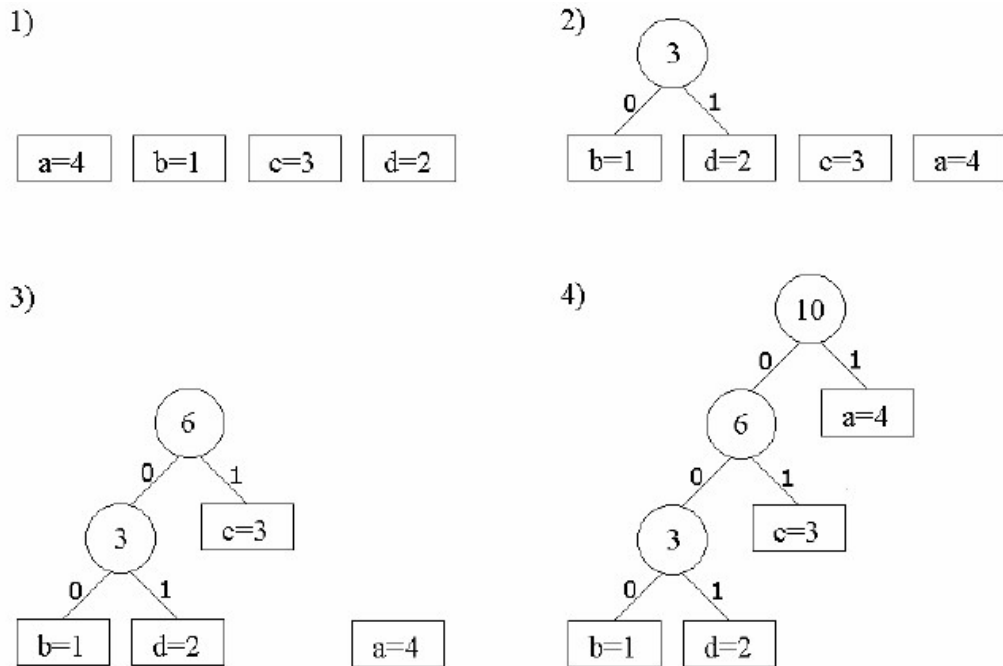


Figure 11 Example of building Huffman code tree

After building the Huffman tree, the algorithm creates a prefix code for each symbol from the alphabet by traversing the binary tree from the root to the node, which corresponds to the symbol. It records 0 for a left branch and 1 for a right branch. The Huffman algorithm, using the Huffman tree from Figure 11, assigns codes '1', '000', '01', and '001' to symbols 'a', 'b', 'c', and 'd' respectively.

Having Huffman codes for our model, we can encode, for example, the string 'acbdd'. It is encoded using $1 + 2 + 3 + 3 + 3 = 12$ bits, what gives 2.4 bits/char. From previous subsection, we know that entropy for this model is equal to 1.84644 bits/char. Ineffectiveness comes from the fact that usually symbol x cannot be encoded in exactly $-\log_2 P(x)$ bits, unless $P(x)$ is a

negative power of 2. In other words, the entropy for most symbols is usually a non-integer, but the Huffman coding uses codes of integer length.

Above presented algorithm is called an *semi-adaptive* (or an *semi-static*) Huffman coding as it requires knowledge of frequencies for each symbol from alphabet. Moreover, the Huffman tree with the Huffman codes for symbols (or just frequencies of symbols, which can be used to create the Huffman tree) must be stored together with the compressed output. This information is needed by the decoder and it is usually placed in the header of a compressed file.

2.2.2.1.2 Adaptive Huffman coding

The semi-adaptive Huffman coding is not suitable to situations when probabilities of the input symbols are changing. It is very ineffective to use Huffman algorithm for building the tree and generating prefix codes after encoding each symbol from the input. In 1973 Faller has presented modified version of Huffman algorithm (Faller, 1973), which manages with this problem. This algorithm is nowadays known as an *adaptive* Huffman coding.

The adaptive Huffman algorithm presents a different approach to building a Huffman tree, which introduces a concept known as the sibling property. This algorithm starts with an empty tree or a standard distribution. It adds to the tree a special control symbols identifying new symbols, which currently are not a part of the tree. The adaptive Huffman algorithm allows modifying the Huffman tree after encoding each symbol from the input. In this way, Huffman codes can be dynamically changed according to a change of probabilities of the symbols. Of course, the encoder and decoder must maintain the same Huffman tree.

Usually the adaptive Huffman algorithm produces code that is more effective than the semi-adaptive Huffman code. There is also no need to store the Huffman tree with the Huffman codes for symbols with the compressed output. On the other hand, adaptive Huffman algorithm has to update the Huffman tree after encoding each symbol, therefore is slower than semi-adaptive version.

Moreover, the compression effectiveness at the beginning of the coding or for small files is low.

The semi-adaptive and the adaptive Huffman coding decrease redundancy in a data by the fact that distinct symbols have distinct probabilities of occurrence. Symbols with higher probabilities of occurrence have assigned shorter codes, while symbols with lower probabilities are encoded with longer codes. In practical applications, however, the adaptive Huffman coding is often replaced by easier and more effective arithmetic coding, described in the next subsection.

The Huffman coding is rarely used as an independent compression method. It is usually used as the coding method in the last stage of lossless data compression. Huffman compression is used in a connection with, for example, LZSS algorithm (used in `gzip`, `PKZip`, `ARJ`, and `LHArc`), BWT-based algorithms, JPEG (Wallace, 1991) and MPEG compression, Run- Length Encoding, Move-To-Front coding (Skibiński, 2006).

2.2.2.2. Arithmetic Coding

Practical arithmetic coding came into existence in 1976 through the work of Risannen (1976) and Pasco (1976). However, the basic ideas of arithmetic coding have their origins in the original work of Shannon (1948). Arithmetic coding relies on the fact that there are an uncountably infinite number of numbers between 0 and 1 (or any other nonzero interval on the real number line). Therefore, we can assign a unique number from the unit interval to any sequence of symbols from a finite alphabet. We can then encode the entire sequence with this single number which acts as a label or tag for this sequence. In other words, this number is a code for this sequence: a binary representation of this number is a binary code for this sequence. Because this tag is unique, in theory, given the tag the decoder can reconstruct the entire sequence. In order to implement this idea we need a mapping from the sequence to the tag and an inverse mapping from the tag to the sequence.

2.2.3 Sources with Memory

Throughout our discussion we have been assuming that each symbol in a sequence occurs independently of the previous sequence. In other words, there is no *memory* in the sequence. Most sequences of interest to us are not made up of independently occurring symbols. In such cases an assumption of independence would give us an incorrect estimate of the probability of occurrence of that symbol, thus leading to an inefficient code. By taking account of the dependencies in the sequence we can significantly improve the amount of compression available.

Consider the letter u in a piece of English text. The frequency of occurrence of the letter u occurring in a piece of English text is approximately 0.018. If we had to encode the letter u and we used this value as our estimate of the probability of the letter, we would need approximately $\lceil \log_2 \frac{1}{0.018} \rceil = 6$ bits. However, if we knew the previous letter was q our estimate of the probability of the letter u would be substantially more than 0.018 and thus encoding u would require fewer bits. Another way of looking at this is by noting that if we have an accurate estimate of the probability of occurrence of particular symbols we can obtain a much more efficient code. If we know the preceding letter(s) we can obtain a more accurate estimate of the probabilities of occurrence of the letters that follow.

Similarly, if we look at pixel values in a natural image and assume that the pixels occur independently, then our estimate of the probability of the values that each pixel could take on would be approximately the same (and small). If we took into account the values of the neighboring pixels, the probability that the pixel under consideration would have a similar value would be quite high. If, in fact, the pixel had a value close to its neighbors, encoding it in this context would require many fewer bits than encoding it independent of its neighbors (Memon & Sayood, 1995).

2.2.3.1 Prediction with Partial Match

In general, the larger the size of a context, the higher the probability that we can accurately predict the symbol to be encoded. Suppose we are encoding the fourth letter of the sequence *their*. The probability of the letter *i* is approximately 0.053. It is not substantially changed when we use the single letter context *e*, as *e* is often followed by most letters of the alphabet. In just this paragraph it has been followed by *d, i, l, n, q, r, t,* and *x*. If we increase the context to two letters *he*, the probability of *i* following *he* increases. It increases even more when we go to the three-letter context *the*. Thus, the larger the context, the better off we usually are. However, all the contexts and the probabilities associated with the contexts have to be available to both the encoder and the decoder.

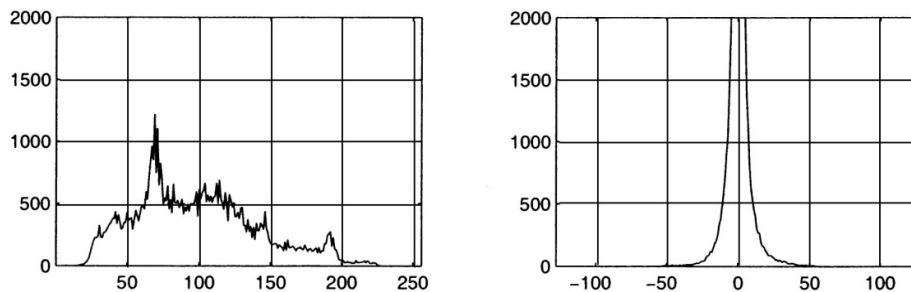


Figure 12 Histograms of pixel values in the sensin image and of the differences between neighboring pixels.

The number of contexts increases exponentially with the size of the context. This puts a limit on the size of the context we can use. Furthermore, there is the matter of obtaining the probabilities relative to the contexts. The most efficient approach is to obtain the frequency of occurrence in each context from the past history of the sequence. If the history is not very large, or the symbol is an infrequent one, it is quite possible that the symbol to be encoded has not occurred in this particular context.

2.2.4 Predictive Coding

If the data we are attempting to compress consists of numerical values, such as images, using context-based approaches directly can be problematic. There are several reasons for this. Most context-based schemes exploit exact reoccurrence of patterns. Images are usually acquired using sensors that have a small amount of noise. While this noise may not be perceptible, it is sufficient to reduce the occurrence of exact repetitions of patterns. A simple alternative to using the context approach is to generate a prediction for the value to be encoded and encode the prediction error. If there is considerable dependence among the values with high probability, the prediction will be close to the actual value and the prediction error will be a small number. We can encode this small number, which as it occurs with high probability will require fewer bits to encode. An example of this is shown in Figure 12. The plot on the left of Figure 12 is the histogram of the pixel values of the *sensin* image, while the plot on the right is the histogram of the differences between neighboring pixels. We can see that the small differences occur much more frequently and therefore can be encoded using fewer bits. As opposed to this, the actual pixel values have a much more uniform distribution.

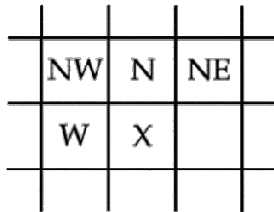


Figure 13 Labeling of pixels in the neighborhood of a pixel to be encoded.

Because of the strong correlation between pixels in a neighborhood, predictive coding has been highly effective for image compression. It is used in the current state-of-the-art algorithm CALIC (Wu & Memon, 1996) and forms the basis of JPEG-LS, which is the standard for lossless image compression.

2.2.4.1 JPEG-LS

The JPEG-LS standard uses a two-stage prediction scheme followed by a context-based coder to encode the difference between the pixel value and the prediction. For a given pixel the prediction is generated in the following manner. Suppose we have a pixel with four neighboring pixels as shown in Figure 13. The initial prediction X is obtained as

```
if  $NW \geq \max(W,N)$ 
 $X = \max(W,N)$ 
else
{
    if  $NW \leq \min(W,N)$ 
     $X = \min(W,N)$ 
    else
     $X = W + N - NW$ 
}
```

This prediction is then refined by using an estimate of how much the prediction differed from the actual value in similar situations in the past. The “situation” is characterized by an activity measure which is obtained using the differences of the pixels in the neighborhood. The differences $NE - N$, $N - NW$, and $NW - W$ are compared against thresholds which can be defined by the user and a number between 0 and 364 and a *SIGN* parameter which takes on the values +1 and -1. The sign parameter is used to decide whether the correction should be added or subtracted from the original prediction. The difference between the pixel value and its prediction is mapped into the range of values occupied by the pixel and encoded using Golomb codes. Details can be found at Sayood (2000) and Weinberger *et al* (1998).

2.2.4.2 Burrows-Wheeler Transform

The Burrows-Wheeler transform (BWT) uses the memory in a sequence in a somewhat different manner than either of the two techniques described previously. In this technique the entire sequence to be encoded is read in and all possible cyclic shifts of the sequence are generated. These are then sorted in lexicographic order. The last letter of each sorted cyclically shifted sequence is then transmitted along with the location of the original sequence in the sorted list. The sorting results in long sequences of identical letters in the transmitted sequence. This structure can be used to provide a very efficient representation of the transmitted sequence. The easiest way to understand the BWT algorithm is through an example.

R	I	N	T	I	N	T	I	N	I	N	R	I	N	T	I	N	T
N	R	I	N	T	I	N	T	I	I	N	T	I	N	R	I	N	T
I	N	R	I	N	T	I	N	T	I	N	T	I	N	T	I	N	R
T	I	N	R	I	N	T	I	N	N	R	I	N	T	I	N	T	I
N	T	I	N	R	I	N	T	I	N	T	I	N	R	I	N	T	I
I	N	T	I	N	R	I	N	T	N	T	I	N	T	I	N	R	I
T	I	N	T	I	N	R	I	N	R	I	N	T	I	N	R	I	N
N	T	I	N	T	I	N	R	I	T	I	N	R	I	N	T	I	N
I	N	T	I	N	T	I	N	R	T	I	N	T	I	N	R	I	N

Figure 14 Cyclically shifted versions of the original sequence and lexicographically order set of the cyclical shifts.

EXAMPLE: BWT

Suppose we wish to encode the sequence *RINTINTIN*. We first obtain all cyclical shifts of this sequence and then sort them in lexicographic order as shown in Figure 14.

We transmit the string consisting of the last letters in the lexicographically ordered set and the position of the original string in the lexicographically ordered set. For this example the transmitted sequence would be the string *TTRIINN* and the index 7. Notice that the string to be transmitted contains relatively long strings of the same letter. If our example string had been realistically long, the runs of identical letters would have been correspondingly long. Such a string is easy to compress. The method of choice for BWT is

move-to-front coding (Burrows & Wheeler, 1994; Nelson, 1996; Sayood, 2000).

Once the string and the index have been received, we decode them by working backwards. We know the last letter in the string is its seventh element which is N . To find the letter before N we need to generate the string containing the first letters of the lexicographically ordered set. This can be easily obtained as $IIINNRTT$ by lexicographically ordering the received sequence. We will refer to this sequence of first letters as the \mathcal{F} sequence and the sequence of last letters as the \mathcal{L} sequence. Note that given a particular string and its cyclic shift, the last letter of the string becomes the first letter of the cyclically shifted version, and the last letter of the cyclically shifted version is the letter prior to the last letter in the original sequence. Armed with this fact and the knowledge of where the original sequence was in the lexicographically ordered set, we can decode the received sequence. We know the original sequence was seventh in the lexicographically ordered set; therefore, the last letter of the sequence has to be N . This is the first N in \mathcal{L} . Looking in \mathcal{F} we see that the first N appears in the fourth location. The fourth element in \mathcal{L} is I . Therefore, the letter preceding N in our decoded sequence is I . This I is the first I in \mathcal{L} ; therefore, we look for the first I in \mathcal{F} . The first I occurs in the first location. The letter in the first location in \mathcal{L} is T . Therefore, the decoded sequence becomes TIN . Continuing in this fashion we can decode the entire sequence.

The BWT is the basis for the compression utilities *bzip* and *bzip2*. The more details on BWT compression can be found in Burrows and Wheeler (1994), Nelson and Gailly (1996), and Sayood (2000).

2.2.5 Dictionary-Based Methods

In dictionary based compression methods, the encoder operates on-line, inferring its dictionary of available phrases from previous parts of the message and adjusting its dictionary after the transmission of each phrase. This allows the

dictionary to be transmitted implicitly, since the decoder simultaneously makes similar adjustment to its dictionary after receiving each phrase. The Lempel-Ziv families of compression methods (Ziv & Lempel, 1977) (Ziv & Lempel, 1978) (Welch, 1984) are of this type and are used in many file storage and archiving systems. These methods perform better than the character-based methods in terms of speed and space. The main drawback of these methods for database applications is the locality of reference. The encoded data using the initial part of the dictionary is not the same as the encoded data using the later part of the dictionary. Therefore the compressed data is not directly addressable in these methods (McGregor *et al.*, 1998). These techniques require decompression all, or a large amount of the data even if only a small part of that data is required. Alternatively, a complete dictionary is created in advance using the full message. The dictionary is included explicitly as part of the compressed message. This scheme is highly efficient for decompression, and the compressed data can be searched directly (Larson & Moffat, 2000).

One of the earliest forms of compression is to create a dictionary of commonly occurring patterns which is available to both the encoder and the decoder. When this pattern occurs in a sequence to be encoded it can be replaced by the index of its entry in the dictionary. A problem with this approach is that a dictionary that works well for one sequence may not work well for another sequence. In 1977 and 1978, Jacob Ziv and Abraham Lempel (1977, 1978) described two different ways of making the dictionary adapt to the sequence being encoded. These two approaches form the basis for many of the popular compression programs of today.

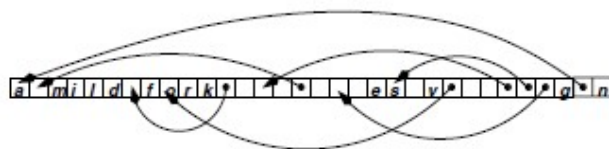


Figure 15 Illustration for LZ77.

2.2.5.1 LZ77

The 1977 algorithm commonly referred to as LZ77 uses the past of the sequence as the dictionary. Whenever a pattern recurs within a predetermined window, it is replaced by a pointer to the beginning of its previous occurrence and the length of the pattern. Consider the following (admittedly weird) sequence:

a mild fork for miles vorkosigan

We repeat this sentence in Figure 15 with recurrences replaced by pointers. If there were sufficient recurrences of long patterns, we can see how this might result in a compressed representation. In Figure 15 we can see that the “compressed” representation consists of both pointers and fragments of text that have not been encountered previously. We need to inform the decoder when a group of bits is to be interpreted as a pointer and when it is to be interpreted as text. There are a number of different ways in which this can be done. The original LZ77 algorithm used a sequence of triples $\langle o, l, c \rangle$ to encode the source output, where o is the offset or distance to the previous recurrence, l is the length of the pattern, and c corresponds to the character following the recurrence. In this approach when a symbol was encountered for the first time (in the encoding window) the values for o and l were set to 0. Storer and Szymanski (1982) suggested using a one bit flag to differentiate between pointers and symbols which had not occurred previously in the coding window. In their variant, commonly known as LZSS, the one bit flag is followed by either a pair $\langle o, l \rangle$ or the code word of the new symbol. There are a number of variants of both the LZ77 and the LZSS algorithms which are the basis for several popularly used compression utilities. Included among them are *gzip* and *PNG* (portable network graphics).

2.2.5.2 LZ78

The 1978 algorithm requires actually building a dynamic dictionary based on the past symbols in the sequence. The sequence is encoded using pairs of code

words $\langle i, c \rangle$, where i is the index to a pattern in the dictionary and c is the code word of the character following the current symbol. The most popular variant of the LZ78 algorithm is the LZW algorithm developed by Terry Welch (1984). In this variation the dictionary initially contains all the individual letters of the source alphabet. The encoder operates on the sequence by collecting the symbols in the sequence into a pattern p until such time as the addition of another symbol will result in a pattern which is not contained in the dictionary. The index to p is then transmitted, the pattern p concatenated with is added as the next entry in the dictionary, and a new pattern p is begun with as the first symbol.

Variants of the LZW algorithm are used in the UNIX *compress* command as part of the graphic interchange format (GIF) and for the modem protocol v. 42bis.

2.2.6 Statistical Encoding

Statistical encoding uses the probabilities of occurrence of each character and each group of characters, assigns short codes to frequently occurring characters or groups of characters while assigns longer codes to less frequently encountered characters or groups of characters (Held & Marshel, 1996). The widely used statistical compression methods are Huffman (Huffman, 1952) and Shannon-Fano (Fano, 1949; Shannon, 1948) encoding. These methods are static and require a prior knowledge of the probability of occurrence of each character in the input string. Performance degrades if the frequency of occurrences changes. Static methods require at least two passes: one pass to determine the probability of occurrences of the input alphabet and the other pass to encode the string. To maintain the efficiency of the resulting code obtained by compressing data, adaptive or dynamic compression schemes have been developed by many researchers (Vitter, 1987; Gallager, 1978).

2.2.7 Lightweight Compression Methods

Lightweight compression techniques work on the basis of some relationship between values, such as when a particular value occurs often, or if we encounter long runs of repeated values. Run length encoding (Golomb, 1966), delta encoding and dictionary encoding (Roth & Van Horn, 1993; Graefe & Shapiro, 1991; Chen *et al.*, 2001) are some examples of lightweight compression techniques. In a lightweight compression technique, the compression algorithm is simple and fast. The compression and decompression time is more important than the amount of compression.

2.2.8 Heavyweight Compression Methods

LZO (Lempel Ziv Oberhummer) (Oberhummer, 2002) is a modification of the original Lempel Ziv (Ziv & Lempel, 1977) dictionary coding algorithm. In this research author works by replacing byte patterns with tokens. Each time the algorithm recognizes a new pattern, it outputs the pattern and then it adds it to a dictionary. The next time it encounters that pattern, it outputs this token from the table. The first 256 tokens are assigned to possible values of a single byte. Subsequent tokens are assigned to larger patterns.

Details on the particular algorithm modifications added by LZO are undocumented, although the LZO code is highly optimized and hard to decipher. LZO is heavily optimized for decompression speed. It provides the following features:

- Decomposition is simple and very fast.
- Requires no memory for decomposition.
- Compression is fast.
- The algorithm is thread safe.
- The algorithm is loss-less.

2.3 Compression on Database Processing

Compression has now become an essential part of many large information systems where large amount of data is processed, stored or transferred. This data may be of any type e.g., voice, video, text, tables etc. No single compression technique is suitable for all types of data. Lossy compression is appropriate for voice or video data where as loss-less compression is suitable for other data types. Cormack (1985) has used a modified Huffman code (Huffman, 1952) for the IBM IMS database system. Westmann et al. (2000) has developed a lightweight compression method based on LZW (Welch, 1984) for relational databases. Moffat et al. (1992) use the Run Length Encoding (Golomb, 1966) method for a parameterized compression technique for sparse bitmaps of a digital library.

2.4 State-of-the-art discussion

Huffman coding is the most popular and widely used coding system, where compression is done by assigning a shorter code to a symbol with higher frequencies. This technique assigns a unique codeword for each symbol (Huffman, 1952). Huffman based technique is used in many compression algorithms like Zip, PKZip, BZip2, and PNG. Multimedia codec such as JPEG and MP3 have a front end model and quantization followed by Huffman coding. Almost all communications with and from the internet are at some points Huffman encoded. Almost all Huffman based algorithms attempt to improve decoding speed; in most of the cases, they did not achieve very good compression speed.

The existing Huffman based algorithms use binary code which slow the decoding speed. This research proposes a new compression algorithm that makes use of a variation of the classic Huffman coding: quaternary Huffman coding. Using quaternary Huffman coding, each symbol is encoded into a quaternary code stream, instead of a binary bit stream. A quaternary code stream for Huffman coding requires a shorter Huffman tree, i.e., less depth. The

potential benefit of a shorter Huffman tree is less traverse time, which improves both compression and decompression throughput. In this research, we analyze the properties of quaternary Huffman tree and conclude that a quaternary Huffman tree is usually one-third of the height from a binary tree. In this research, we further implement a compression/decompression algorithm based on it.

2.5 Summary

Different types of compression algorithms and coding systems used in compression algorithms have manifested in the above discussion. We have thoroughly discussed the Huffman architecture because we taken these model as the basis of our architecture. It is observed that the length of the Huffman code affect the compression ratio and decoding speed. Therefore, we produce code in a new fashion. In this research, we developed a dictionary-based bit level lossless compression technique. We produce the dictionary by using a special version of Huffman principle called quaternary Huffman principle which is explained in next chapter. The codeword generated by the quaternary tree technique is more optimal because it is produced from a less height tree. We produce a dictionary based on the quaternary codes. This new dictionary ensures the enhancement of compression and decompression speed.

Chapter 3

Quaternary Tree Structure

The tree data structure is very important to store data in the memory and retrieve data from the memory. A tree can be structured in many ways. Currently, the binary tree structure is mostly used in Computer Science. The required space to store data in memory and decoding time for retrieving those data from the memory depend on the structure of the tree. The weighted path length, the height of tree, eccentricity, diameter, center, radius, number of internal nodes are some performance measuring mathematical parameters. In this chapter, we describe the use of quaternary tree instead of binary tree to speed up the decoding time for Huffman codes. It is usually difficult to achieve a balance between speed and memory usage using variable-length binary Huffman code. Quaternary tree is used here to produce optimal codeword that speeds up the way of searching. This chapter also describes the details of some other tree structures that could generate the codeword for data compression. The structure, algorithm of different tree is described in this chapter.

3.1 Tree

A tree is a connected undirected graph with no simple circuits. A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root. If v is a vertex in T other than the root, the parent of v is the unique vertex u such that there is a directed edge from u to v . If u is the parent of v , v is called a child of u . Vertices with the same parent are called siblings.

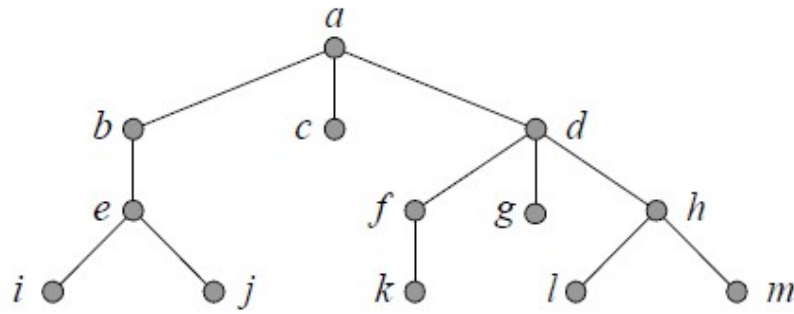


Figure 16 Tree construction

The ancestors of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root. The descendants of a vertex v are those vertices that have v as an ancestor. A vertex of a tree is called a leaf if it has no children. Vertices that have children are called internal vertices. If a is a vertex in a tree, the subtree with a as its root is the subgraph of the tree consisting of a and its descendants and all edges incident to these descendants.

A root tree is called an m -ary tree if every internal vertex has no more than m children. The tree is called a full m -ary tree if every internal vertex has exactly m children. An m -ary tree with $m = 2$ is called a binary tree. An ordered rooted tree is a rooted tree where the children of each internal vertex are ordered. Ordered rooted trees are drawn so that the children of each internal vertex are shown in order from left to right. In an ordered binary tree (usually called just a binary tree), if an internal vertex has two children, the first child is called the left child and the second child is called the right child. The tree rooted at the left child (or right child, resp.) of a vertex is called the left subtree (or right subtree, resp.) of this vertex.

3.2 Quaternary Tree Architecture

In this research we exploit quaternary tree to construct a dictionary. We choose quaternary tree (4-ary tree or a tree with at most 4 children) instead of binary

tree (2 ary tree or a tree with at most 2 children). Although each of them (ternary and quaternary) will have same storage complexity but ternary tree will have more internal nodes and more height than quaternary tree. For this reason, ternary tree requires more traversing time than quaternary tree.

Connected undirected acyclic graph is called a tree T . It has a set of edges $E = \{e_0, e_2, \dots, e_{n-1}\}$ and vertices $V = \{v_0, v_1, \dots, v_{n-1}\}$. If u is the parent of v , v is called a child of u . Children with the same parent are called siblings. If a vertex of a tree has no children then it is called a leaf. An internal vertex always has one or more children. The tree T contains a distinguished node R , called root of T . A tree T is called binary tree if each vertex has at most two children. A tree T is called quaternary tree if a vertex of T has at most four children named LEFT, LEFT-MID, RIGHT-MID, RIGHT. When every internal vertex of a tree T has all four children the tree T is called a full quaternary tree.

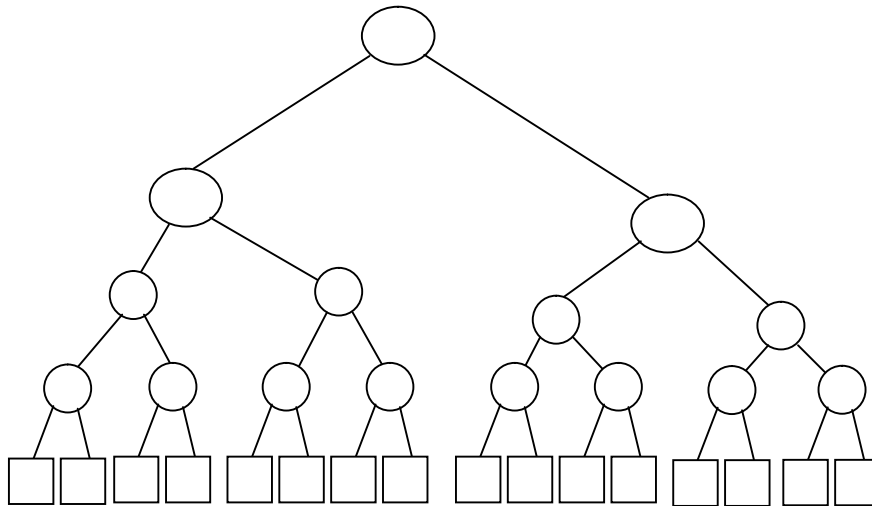


Figure 17 Binary tree with 16 nodes

An ordered rooted tree T always has ordered children of its every internal vertex. Ordered rooted trees are designed in such a way that the children of each internal vertex are revealed in order from left to right. The height of a tree T is the height of the root which is the best ever path between the root and a leaf

(OCAML, 2014). The diameter of a tree is the longest path between two leaf nodes of a tree.

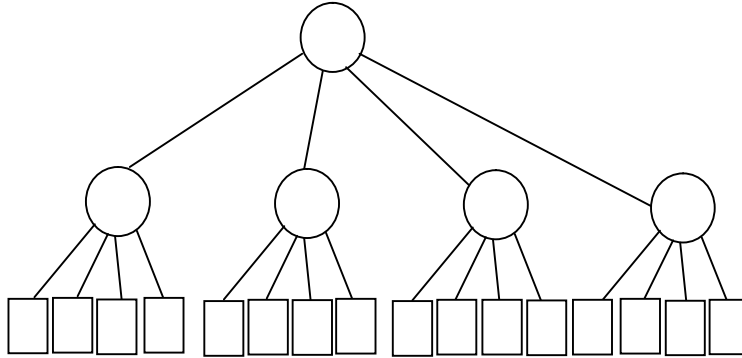


Figure 18 Quaternary tree with 16 nodes

The length of the paths in the tree affects the running time of an algorithm. Consider T is a tree with n external nodes, and assume all of the external nodes are assigned a nonnegative weight. The external weighted path length P of the tree T is defined as the sum of the weighted path lengths of individual nodes (Lipschutz, 2011), then

$$P = f_1 l_1 + f_2 l_2 + \dots + f_n l_n \dots \dots \dots (1)$$

where f_i and L_i stand for the frequency and path length of the external node N_i .

3.2.1. Minimization of time using quaternary tree

For the construction of Huffman codes of all the nodes, it is required to traverse the whole tree. Since the traversing time X of a tree depends on its weighted path length P . Thus we have,

$$X \propto P$$

Then from equation (1), we have

$$\Rightarrow X \propto \sum_{i=1}^n f_i l_i \dots \dots \dots (2)$$

The traversing time of a quaternary tree is always less than that of a binary tree, since $P_q < P_b$, where P_q and P_b are the weighted path length of quaternary and

binary tree representation. We summarize the above result in the following remarks.

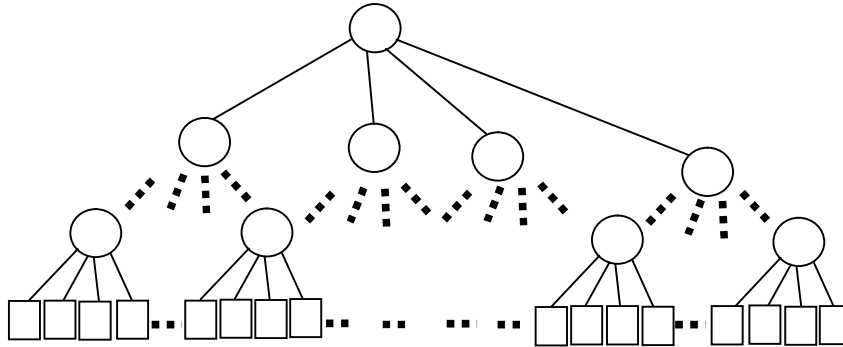


Figure 19 Complete binary tree

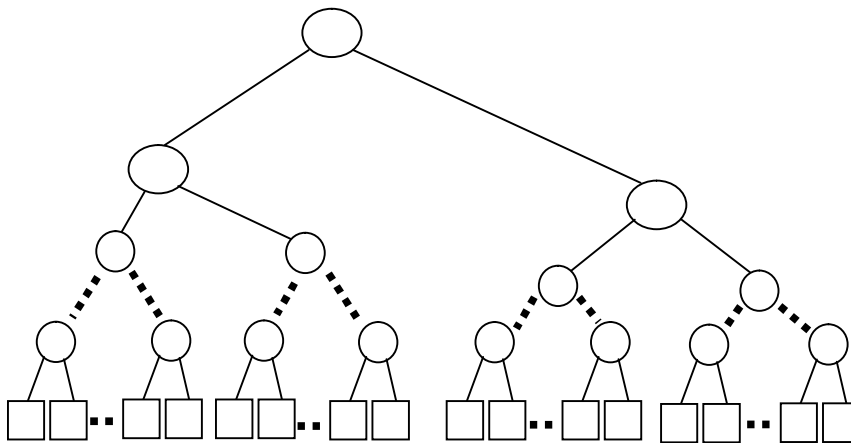


Figure 20 Complete quaternary tree

3.2.2 Remarks

For any distributions, in case of a Huffman tree

- i) *The minimum height of quaternary tree is less than that of binary tree.*
- ii) *The maximum height of quaternary tree is less than that of binary tree. For big n the first one is almost one third of the latter.*
- iii) *The weighted path length in case of quaternary tree is less than that of binary tree.*

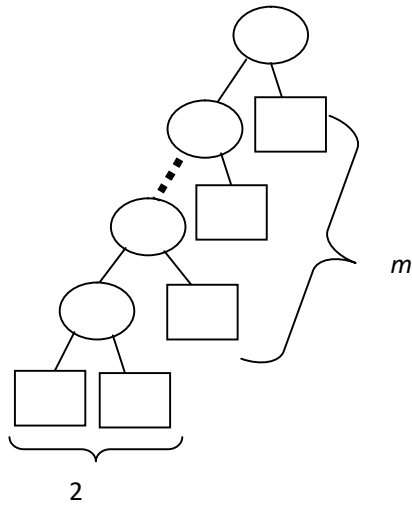


Figure 21 A complete quaternary tree.

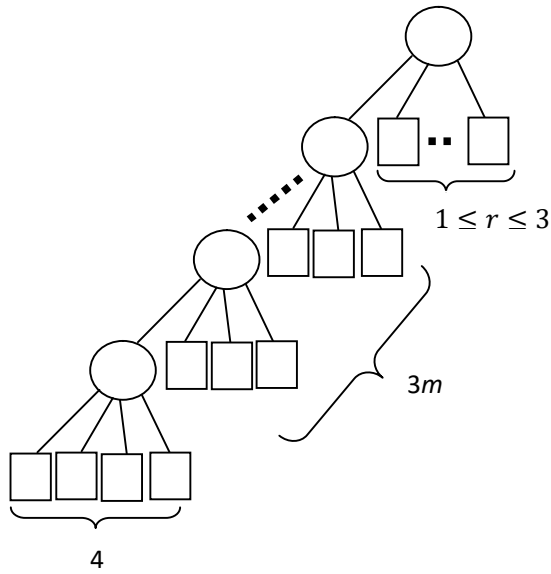


Figure 22 A canonical quaternary tree.

Proof: Consider a complete tree in case of both binary and quaternary representation as shown in Figure 19 and Figure 20 respectively. And also consider a canonical tree for binary and quaternary representation as shown in Figure 21 and Figure 22 respectively. We know that the height of a tree is minimum when it is complete and maximum when it is canonical.

i) A complete k -ary tree with n nodes has height equal to $\log_k n$ which is minimum as shown in Figure 19 and Figure 20 for binary and quaternary tree respectively. Then $l_q = \log_4 n = \frac{1}{2} \log_2 n = \frac{1}{2} l_b$, where l_q is the height in case of a quaternary tree and l_b for binary tree.

ii) In case of a canonical binary tree with n nodes, we consider 2 blocks of levels as shown in Figure 21. There is exactly 1 node in each level except the terminal level which consists of 2 nodes. Let m be the number of levels with only 1 node. Then

$$n = m + 2 \text{ and } l_b = m + 1$$

$$\text{Then } l_b = n - 2 + 1 = n - 1$$

$$\Rightarrow l_b = n - 1 \dots\dots\dots (3)$$

In case of canonical quaternary tree with n nodes, we consider 3 blocks of levels as shown in Figure 22. The terminal level has exactly 4 nodes, the starting level has r ($1 \leq r \leq 3$) nodes and all the intermediate levels must have exactly 3 nodes. If m is the number of intermediate levels, then

$$n = r + 3m + 4, 1 \leq r \leq 3 \text{ and}$$

$$l_q = 1 + m + 1$$

$$\text{Then } m = \frac{n-r-4}{3}, 1 \leq r \leq 3$$

$$\therefore l_q = 1 + \frac{n-r-4}{3} + 1, 1 \leq r \leq 3$$

$$\Rightarrow l_q = \frac{n-r+2}{3}, 1 \leq r \leq 3 \dots\dots\dots (4)$$

Then from equation (3) and equation (4), we have

$$l_q = \frac{n-r+2}{3} < n - 1 = l_b, 1 \leq r \leq 3$$

$$\Rightarrow l_q \approx \frac{1}{3} l_b \text{ (for big } n)$$

iii) For frequencies $f_i, i = 1, 2, 3, \dots, n$, for the best case, by part (i), we have

$$\sum f_i l_q = \frac{1}{2} \sum f_i l_b$$

$$\Rightarrow T_q = \frac{1}{2} T_b \quad [\text{Using equation (2)}]$$

For the worst case, by part (ii), we have

$$\sum f_i l_q \approx \frac{1}{3} \sum f_i l_b \quad (\text{for big } n)$$

$$\Rightarrow T_q \approx \frac{1}{3} T_b \quad [\text{Using equation (2)}]$$

In summary, it is revealed that the traversing time of a quaternary tree can be approximated as one-third of the traversing time of a binary tree for big n whereas it is half of a binary technique in the best case.

3.2.4. Space Requirement for quaternary tree

The best space minimization is achieved when a tree is complete. Because a complete tree requires less decoding time than an incomplete tree keeping the same space requirement. The required space (compressed) depends on the average codeword length. The average codeword length S can be defined as:

$$S = \frac{\sum_{i=1}^n l_i f_i}{\sum_{i=1}^n f_i}, \dots\dots\dots (5)$$

where n is the number of distinct symbols

As mentioned earlier, the length of codeword depends on the path length. Since $l_i \propto \alpha_i$, therefore equation (5) can be rewritten as,

$$S = \frac{K \sum_{i=1}^n \alpha_i f_i}{\sum_{i=1}^n f_i}, \quad L_i = \alpha_i \cdot K$$

where K = arity and α_i = height constant

Again, for a particular symbol, when the frequency f_i is increased, the corresponding height α_i is decreased. Thus we can also write,

$$\alpha_i \propto \frac{1}{f_i}$$

When $K=1$, then it is a binary tree and it requires a minimum 1 bit to represent a symbol. When $K=2$ then it is a quaternary tree and it requires at least 2 bits to represent a symbol. Apparently, the space requirement is increased for a quaternary tree. However, when the value of K increases, the value of α_i decreases accordingly. It is obvious that a higher number of K produces a tree with a lower height. Hence the space requirement for higher values of K does not increase linearly with the increment of K .

3.3 Code Generation Techniques

The objective of this section is to verify the applicability of several Huffman based algorithms. Better encoding and decoding speed is achieved by sacrificing an insignificant amount of space in quaternary technique, indicating that searching two bits at a time speed up the overall processing speed than searching a single bit. This motivated us to search three or four bits at a time. In this connection, the Octanary algorithm is introduced to produce three bit based Huffman code, whereas the Hexanary algorithm is introduced to produce four bit based Huffman code. In this section, we explain the different Huffman based tree structure and their implementation techniques.

3.3.1. Huffman Codes to Binary Data

Huffman's scheme uses a table of frequency to produce codeword for each symbol. This table consists of every symbol of entire document and its respective frequency is arranged in ascending order. According to the frequency of distinct symbol, each symbol has a variable length bit string and all the bit string is distinct. The Table 1 shows the variable length codeword for different symbols of the "Luke 5". Luke 5 is the fifth chapter of the Gospel of Luke in the New Testament of the Christian Bible. The chapter relates the recruitment of Jesus' first disciples and continues to describe Jesus's teaching and healing ministry. Early criticism by the religious authorities is encountered (Luke 5, 2019).

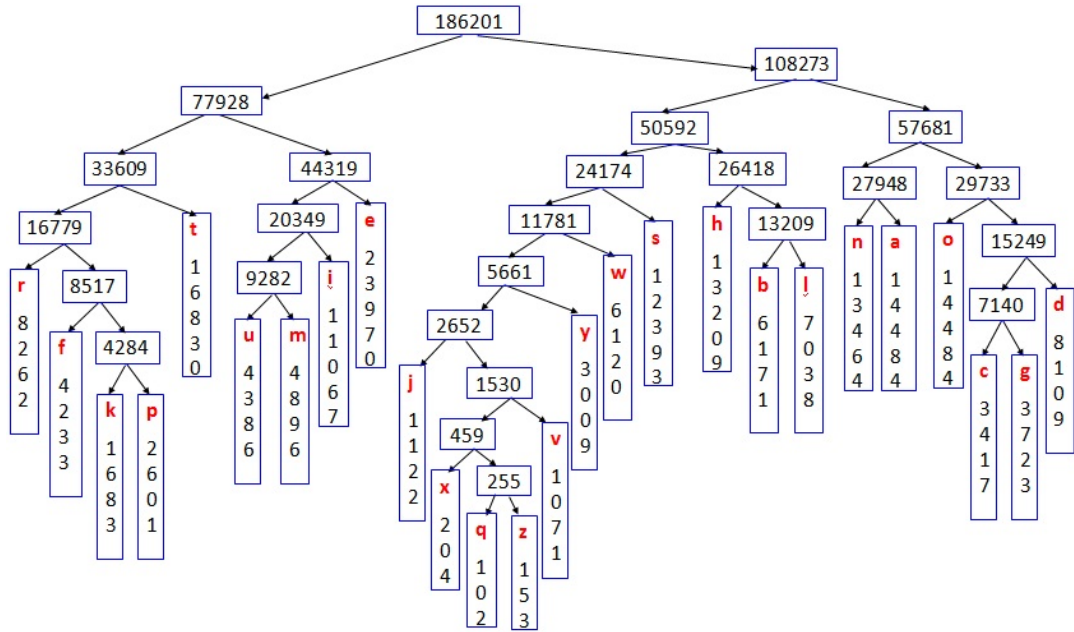


Figure 23 Construction of Binary Huffman Tree.

Consider a set of source symbols $S = \{s_0, s_1, \dots, s_{n-1}\} = \{e, t, \dots, q\}$ with frequencies $W = \{w_0, w_1, \dots, w_{n-1}\}$ for $w_0 \geq w_1 \geq \dots \geq w_{n-1}$, where the symbol s_i has frequency w_i and n is the number of symbols. The codeword c_i , $0 \leq i \leq n - 1$, for symbol s_i can be calculated by traversing the path from root to the symbol s_i , when goes to left it writes '0' and when goes to right it writes '1'. If the level of the root is zero then the codeword length can be determined as the level of s_i . The traversing time of a tree depends on its weighted path length $\sum w_i l_i$, which is expected to be minimum. The Huffman tree for the source symbols $\{s_0, s_1, \dots, s_{26}\}$ with the frequencies $\{23970, 16830, \dots, 102\}$, respectively, for the above example is shown in the Figure 23. The codeword set $C\{c_0, c_1, \dots, c_{26}\}$ is derived as $\{011, 001, \dots, 1000001010\}$, respectively, is shown in Table 1.

We than need to assign a variable length bit string to each character that unambiguously represents that character. This means that the encoding for each character must have a unique prefix. The Table 1 shows the variable length codewords for different symbols of the file *luke 5*.

3.3.2. Huffman Codes to Quaternary Data

As described earlier, quaternary tree or 4-ary tree is a tree in which each node has 0 to 4 children (labeled as LEFT child, LEFT MID child, RIGHT MID child, RIGHT child). Here for constructing codes for quaternary Huffman tree we use 00 for left child, 01 for left mid child, 10 for right mid child, and 11 for right child.

The process of the construction of a quaternary tree is described below:

- Listing all possible symbols with their probabilities;
- Finding the four symbols with the smallest probabilities;
- Replacing these by a single set containing all four symbols, and the probability of the parent is the sum of the individual probabilities.
- Repeating the procedure until it has one node.

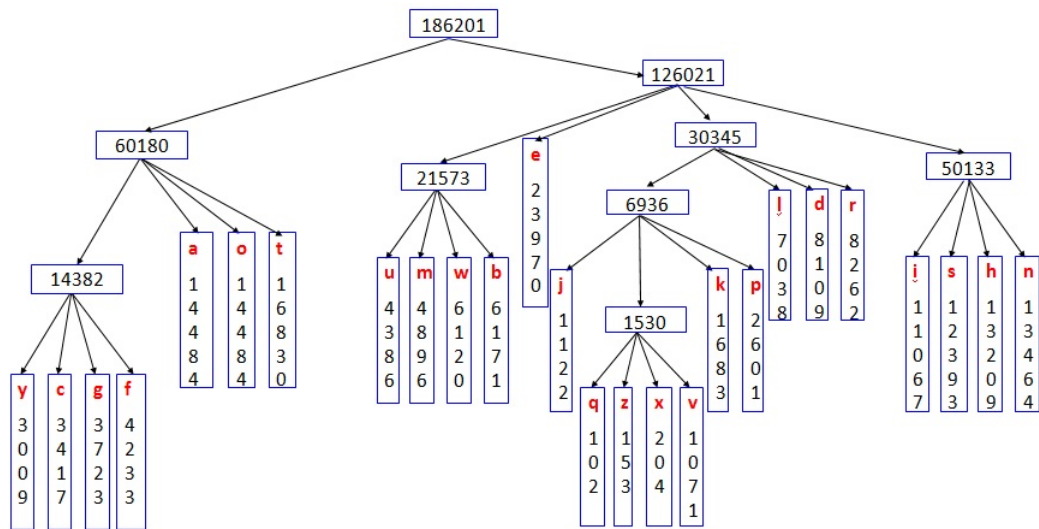


Figure 24 Construction of Quaternary Huffman Tree.

Consider a set of source symbols $S = \{s_0, s_1, \dots, s_{n-1}\} = \{e, t, \dots, q\}$ with frequencies $W = \{w_0, w_1, \dots, w_{n-1}\}$ for $w_0 \geq w_1 \geq \dots \geq w_{n-1}$, where the symbol s_i has frequency w_i and n is the number of symbols. The codeword c_i , $0 \leq i \leq n - 1$, for symbol s_i can be calculated by traversing the path from root

to the symbol s_i , when goes to left it writes ‘00’, when goes to left mid writes ‘01’, when goes to right mid writes ‘10’ and when goes to right writes ‘11’. The codeword length of a symbol can simply be calculated as the level of s_i . We know that the traversing time of a tree depends on its weighted path length $\sum w_i l_i$, and it is expected to be minimum. The quaternary Huffman tree for the source symbols $\{s_0, s_1, \dots, s_{26}\}$ with the frequencies $\{23970, 16830, \dots, 102\}$, respectively, for the above example (“Luke 5”) is shown in the Figure 24. The codeword set $C\{c_0, c_1, \dots, c_{26}\}$ is derived as $\{0101, 0011, \dots, 0110000100\}$, respectively, which is shown in Table 1.

3.3.3 Huffman Codes to Octanary Data

Octanary tree or 8-ary tree is a tree in which each node has 0 to 8 children (labeled as LEFT1 child, LEFT2 child, LEFT3 child, LEFT4 child, RIGHT1 child, RIGHT2 child, RIGHT3 child, RIGHT4 child). Here for constructing codes for octanary Huffman tree we use 000 for left1 child, 001 for left2 child, 010 for left3 child, 011 for left4 child, 100 right1 child, 101 for right2 child, 110 for right3 child, and 111 for right4 child. Generation of Huffman codes for a set of symbols is based on the probability of occurrence of the source symbols.

The process of the construction of a octanary tree is described below:

- Listing all possible symbols with their probabilities;
- Finding the eight symbols with the smallest probabilities;
- Replacing these by a single set containing all eight symbols, whose probability is the sum of the individual probabilities.
- Repeating until the list contains single member.

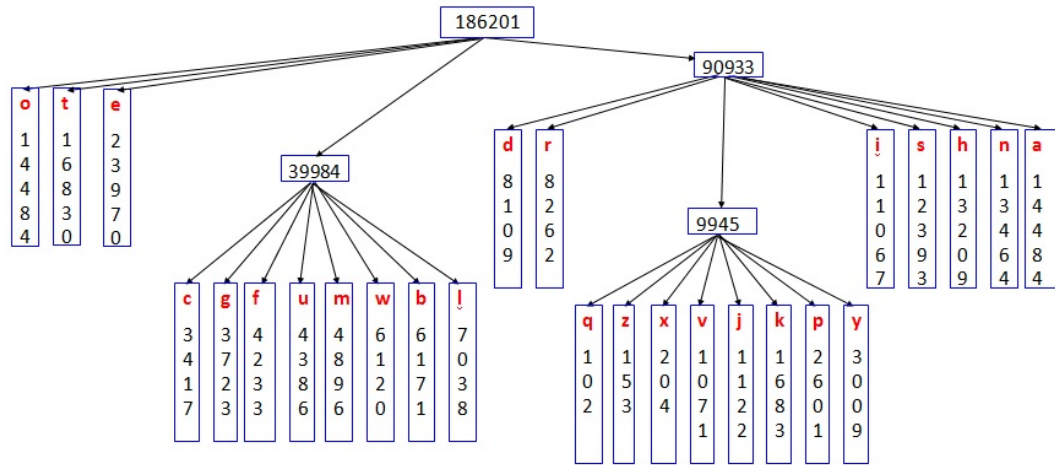


Figure 25 Construction of Octanary Huffman Tree.

Consider a set of source symbols $S = \{s_0, s_1, \dots, s_{n-1}\} = \{e, t, \dots, q\}$ with frequencies $W = \{w_0, w_1, \dots, w_{n-1}\}$ for $w_0 \geq w_1 \geq \dots \geq w_{n-1}$, where the symbol s_i has frequency w_i and n is the number of symbols. The codeword c_i , $0 \leq i \leq n - 1$, for symbol s_i can be calculated by traversing the path from root to the symbol s_i , when goes to most left branch it writes ‘000’, when goes to just after the left branch writes ‘001’, when goes to third branch from the left writes ‘010’ and in this way the most right branch writes ‘111’. The codeword length of a symbol can simply be calculated as the level of s_i . We know that the traversing time of a tree depends on its weighted path length $\sum w_i l_i$, and it is expected to be minimum. The quaternary Huffman tree for the source symbols $\{s_0, s_1, \dots, s_{26}\}$ with the frequencies $\{23970, 16830, \dots, 102\}$, respectively, for the above example (“Luke 5”) is shown in the Figure 25. The codeword set $C\{c_0, c_1, \dots, c_{26}\}$ is derived as $\{010, 001, \dots, 100010000\}$, respectively, which is shown in Table 1.

3.3.4 Huffman Codes to Hexanary Data

Hexanary tree or 16-ary tree is a tree in which each node has 0 to 16 children (labeled as LEFT1 child, LEFT2 child, LEFT3 child, LEFT4 child, LEFT5 child, LEFT6 child, LEFT7 child, LEFT8 child, RIGHT1 child, RIGHT2 child, RIGHT3 child, RIGHT4 child, RIGHT5 child, RIGHT6 child, RIGHT7 child,

RIGHT8 child). Here for constructing codes for hexanary Huffman tree we use 0000 for left1 child, 0001 for left2 child, 0010 for left3 child, 0011 for left4 child, 0100 for left5 child, 0101 for left6 child, 0110 for left7 child, 0111 for left8 child, 1000 right1 child, 1001 for right2 child, 1010 for right3 child, 1011 for right4 child, 1100 for right5 child, 1101 for right6 child, 1110 for right7 child, and 1111 for right8 child. Generation of Huffman codes for a set of symbols is based on the probability of occurrence of the source symbols.

The process of the construction of a hexanary tree is described below:

- Listing all possible symbols with their probabilities;
- Finding the sixteen symbols with the smallest probabilities;
- Replacing these by a single set containing all sixteen symbols, whose probability is the sum of the individual probabilities.
- Repeating until the list contains single member.

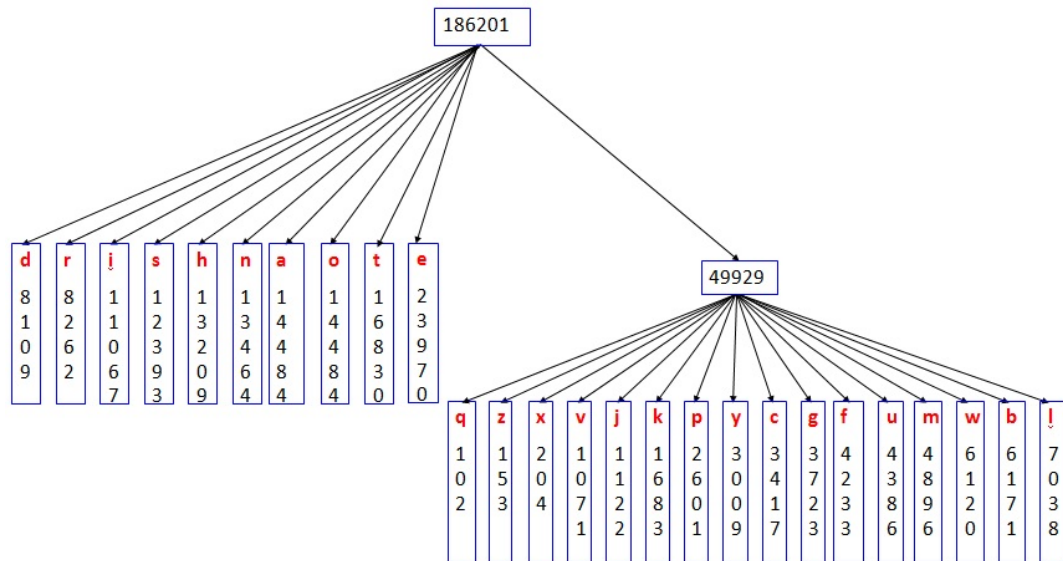


Figure 26 Construction of Hexanary Huffman Tree.

Consider a set of source symbols $S = \{s_0, s_1, \dots, s_{n-1}\} = \{e, t, \dots, q\}$ with frequencies $W = \{w_0, w_1, \dots, w_{n-1}\}$ for $w_0 \geq w_1 \geq \dots \geq w_{n-1}$, where the symbol s_i has frequency w_i and n is the number of symbols. The codeword c_i ,

$0 \leq i \leq n - 1$, for symbol s_i can be calculated by traversing the path from root to the symbol s_i , where the most left branch is corresponding to ‘0000’, just after the left branch is corresponding to ‘0001’, the third branch from left is corresponding to ‘0010’ and in this way the most right branch is corresponding to ‘1111’. The codeword length of a symbol can simply be calculated as the level of s_i . We know that the traversing time of a tree depends on its weighted path length $\sum w_i l_i$, and it is expected to be minimum. The quaternary Huffman tree for the source symbols $\{s_0, s_1, \dots, s_{26}\}$ with the frequencies $\{23970, 16830, \dots, 102\}$, respectively, for the above example (“Luke 5”) is shown in the Figure 26. The codeword set $C\{c_0, c_1, \dots, c_{26}\}$ is derived as $\{1001, 1000, 0111, \dots, 10100000\}$, respectively, which is shown in Table 1.

Table 1 Codeword generated by Huffman based algorithms

Symbol	Frequency	Binary	Quaternary	Octanary	Hexanary
q	102	1000001010	0110000100	100010000	10100000
z	153	1000001011	0110000101	100010001	10100001
x	204	100000100	0110000110	100010010	10100010
v	1071	10000011	0110000111	100010011	10100011
j	1122	1000000	01100000	100010100	10100100
k	1683	000110	01100010	100010101	10100101
p	2601	000111	01100011	100010110	10100110
y	3009	100001	000000	100010111	10100111
c	3417	111100	000001	011000	10101000
g	3723	111101	000010	011001	10101001
f	4233	00010	000011	011010	10101010
u	4386	01000	010000	011011	10101011
m	4896	01001	010001	011100	10101100
w	6120	10001	010010	011101	10101101
b	6171	10100	010011	011110	10101110
l	7038	10101	011001	011111	10101111
d	8109	11111	011010	100000	0000
r	8262	0000	011011	100001	0001
i	11067	0101	011100	100011	0010
s	12393	1001	011101	100100	0011
h	13209	1011	011110	100101	0100
n	13464	1100	011111	100110	0101
a	14484	1101	0001	100111	0110
o	14484	1110	0010	000	0111
t	16830	001	0011	001	1000
e	23970	011	0101	010	1001

3.3.5. Comparison among trees

Table 2 Comparison of different tree structures for *Luke 5* data

Parameter	Binary	Quaternary	Octanry	Hexanary
Number of levels	10	5	3	2
Number of internal nodes	25	9	4	4
Total number of nodes	51	35	30	28
Weighted path length	784023	497301	327063	236130

3.4 Code generation algorithm

To construct Huffman tree, distinct symbols and its frequency are necessary. The tree construction algorithm for the traditional Binary technique is explained in (Cormen *et al.*, 1989). The newly constructed Quaternary, Octanry and Hexanary code generation algorithms are illustrated in this section. There are two types of algorithms:

- Encoding Algorithm
- Decoding Algorithm

3.4.1 Encoding Algorithm

3.4.1.1 Encoding of Quaternary Tree

Encoding is a two-pass problem. The first-pass is to determine the frequencies of letters. We use this information to create the quaternary Huffman tree. We have used a dictionary to store the frequencies of the symbols. When a quaternary Huffman code has been generated, the symbol will be replaced by the code.

Algorithm 1 Encoding of Quaternary Huffman Tree

```
Q- HUFFMAN (C)
1.      Q ← C
2.      n ← |Q|
3.      i ← n
4.      WHILE i > 1
5.          allocate a new node z
6.          left[z] ← v ← EXTRACT-MIN(Q)
7.          left-mid[z] ← w ← EXTRACT-MIN(Q)
8.          IF i = 2
9.              f [z] ← f[v] + f[w]
10.         ELSE IF i =3
11.             right-mid [z] ← x ← EXTRACT-MIN(Q)
12.             f [z] ← f[v] + f[w] + f[x]
13.         ELSE
14.             right-mid [z] ← x ← EXTRACT-MIN(Q)
15.             right [z] ← y ← EXTRACT-MIN(Q)
16.             f [z] ← f[v] + f[w] + f[x] + f[y]
17.         END IF
18.         INSERT(Q, z)
19.         i ← |Q|
20.     END WHILE
21.     RETURN EXTRACT-MIN(Q)
```

In algorithm 1, in line 1 we assign the un-ordered nodes C in the queue Q and later we take the count of nodes in Q and assigning it to n . We assign the value of n to a new variable i . In line 4, we start iterating all the nodes in queue to build the quaternary tree until the count of i is greater than 1 which means there are nodes still left to be added to the parent. In line 5, a new tree node, z is allocated. This node will be the parent node of the least frequent nodes. In line 6, we extract the least frequent node from the queue Q and assign it as a left child of the parent node z . The EXTRACT-MIN (Q) function returns the least frequent node from the queue and removes it from the queue as well. In line 7, we take the next least frequent node from the queue and assign it as a left mid child of the parent z .

From line 8 to 17, we check the value of i or the number of nodes left in the queue Q . If i equals 2, the frequency of the parent node z , $f[z]$ will be the summation of the frequency of node v , $f[v]$ and the frequency of node w , $f[w]$. Likewise, for i is equal to 3 we extract another least frequent node from the queue and add it as a child and add its frequency to the parent node. For i is

greater than 3, we extract two least frequent node and add them as right mid and right child of the parent z and add their frequency to the parent z as well. In line 18, we insert the new parent node z into the queue, Q . In line 19, we take the count of the queue, Q and assign it to i again. The loop continues until a single node is left in the queue. Finally we return the last and single node from the queue Q as a quaternary Huffman tree.

3.4.1.2 Encoding of Octanary Tree

The encoding algorithm for Octanary Huffman tree is shown in algorithm 2. In line 1 we assign the un-ordered nodes, C in the Queue, Q and later we take the count of nodes in Q and assign it to n . We declare a variable i and assign the value of n to it.

In algorithm 2, in line 4, we start iterating all the nodes in the queue to build the Octanary tree until the count of i is greater than 1 which means there are nodes still left to be added to the parent. In line 5, a new tree node, z is allocated. This node will be the parent node of the least frequent nodes. In line 6, we extract the least frequent node from the queue Q and assign it as a *LEFT1* child of the parent node z . The purpose of the *EXTRACT-MIN* (Q) function is to return the least frequent node from the queue. It also removes least frequent node from the queue. In line 7, we take the next least frequent node from the queue and assign it as a *LEFT2* child of the parent z .

From line 8 to 43, we check the value of i or the number of nodes left in the queue Q . If i equals 2, the frequency of the parent node z , $f[z]$ will be the summation of the frequency of node r , $f[r]$ and the frequency of node s , $f[s]$. When i equals 3, we extract another least frequent node from the queue and add it as *LEFT1*, *LEFT2*, *LEFT3* child and add its frequency to the parent node. Likewise, for i equals 4 we extract four least frequent node from the queue and add it as *LEFT1*, *LEFT2*, *LEFT3*, *LEFT4* child and add its frequency to the parent node.

Algorithm 2 Encoding of Octanry Huffman Tree

```

O- HUFFMAN (C)
1.  Q ← C
2.  n ← |Q|
3.  i ← n
4.  WHILE I > 1
5.      allocate a new node z
6.      left1[z] ← r ← EXTRACT-MIN(Q)
7.      left2[z] ← s ← EXTRACT-MIN(Q)
8.      IF i = 2
9.          f[z] ← f[r] + f[s]
10.     ELSE IF i = 3
11.         left3 [z] ← t ← EXTRACT-MIN(Q)
12.         f[z] ← f[r] + f[s] + f[t]
13.     ELSE IF i = 4
14.         left3 [z] ← t ← EXTRACT-MIN(Q)
15.         left4 [z] ← u ← EXTRACT-MIN(Q)
16.         f[z] ← f[r] + f[s] + f[t] + f[u]
17.     ELSE IF i = 5
18.         left3 [z] ← t ← EXTRACT-MIN(Q)
19.         left4 [z] ← u ← EXTRACT-MIN(Q)
20.         right1[z] ← v ← EXTRACT-MIN(Q)
21.         f[z] ← f[r] + f[s] + f[t] + f[u] + f[v]
22.     ELSE IF i = 6
23.         left3 [z] ← t ← EXTRACT-MIN(Q)
24.         left4 [z] ← u ← EXTRACT-MIN(Q)
25.         right1[z] ← v ← EXTRACT-MIN(Q)
26.         right2[z] ← w ← EXTRACT-MIN(Q)
27.         f[z] ← f[r] + f[s] + f[t] + f[u] + f[v] + f[w]
28.     ELSE IF i = 7
29.         left3 [z] ← t ← EXTRACT-MIN(Q)
30.         left4 [z] ← u ← EXTRACT-MIN(Q)
31.         right1[z] ← v ← EXTRACT-MIN(Q)
32.         right2[z] ← w ← EXTRACT-MIN(Q)
33.         right3[z] ← x ← EXTRACT-MIN(Q)
34.         f[z] ← f[r] + f[s] + f[t] + f[u] + f[v] + f[w] + f[x]
35.     ELSE
36.         left3 [z] ← t ← EXTRACT-MIN(Q)
37.         left4 [z] ← u ← EXTRACT-MIN(Q)
38.         right1[z] ← v ← EXTRACT-MIN(Q)
39.         right2[z] ← w ← EXTRACT-MIN(Q)
40.         right3[z] ← x ← EXTRACT-MIN(Q)
41.         right4[z] ← y ← EXTRACT-MIN(Q)
42.         f[z] ← f[r] + f[s] + f[t] + f[u] + f[v] + f[w] + f[x] + f[y]
43.     END IF
44.     INSERT(Q, z)
45.     i ← |Q|
46. END WHILE
47. RETURN EXTRACT-MIN(Q)

```

For i equals 5 we extract five least frequent node from the queue and add it as *LEFT1*, *LEFT2*, *LEFT3*, *LEFT4*, *RIGHT1* child and add its frequency to the parent node. When i equals 6, we extract six least frequent node from the

queue and add it as *LEFT1*, *LEFT2*, *LEFT3*, *LEFT4*, *RIGHT1*, *RIGHT2* child and add its frequency to the parent node.

For *i* equals 7 we extract seven least frequent node from the queue and add it as *LEFT1*, *LEFT2*, *LEFT3*, *LEFT4*, *RIGHT1*, *RIGHT2*, *RIGHT3* child and add its frequency to the parent node. Likewise, when *i* equals 8 we extract eight least frequent node from the queue and add it as *LEFT1*, *LEFT2*, *LEFT3*, *LEFT4*, *RIGHT1*, *RIGHT2*, *RIGHT3*, *RIGHT4* child and add its frequency to the parent node. In line 44, we insert the new parent node *z* into the Queue, *Q*. In line 45, we take the count of the queue, *Q* and assign it to *i* again. And, the loop continues until a single node left in the queue. Finally, the last and single node from the queue *Q* is returned as an Octanary Huffman tree.

3.4.1.3 Encoding of Hexanary Tree

In algorithm 3, in line 1 we assign the un-ordered nodes, *C* in the Queue, *Q* and later we take the count of nodes in *Q* and assigning it to *n*. We declare a variable *i* and assign the value of *n* to it.

In line 4, we start iterating all the nodes in the queue to build the Hexanary tree until the count of *i* is greater than 1 which means there are nodes still left to be added to the parent. In line 5, a new tree node, *z* is allocated. This node will be the parent node of the least frequent nodes. In line 6, we extract the least frequent node from the queue *Q* and assign it as a *LEFT1* child of the parent node *z*. The purpose of the *EXTRACT-MIN (Q)* function is to return the least frequent node from the queue. It also removes least frequent node from the queue. In line 7, we take the next least frequent node from the queue and assign it as a *LEFT2* child of the parent *z*. From line 8 to 121, we check the value of *i* or the number of nodes left in the queue *Q*. If *i* equals 2, the frequency of the parent node *z*, $f[z]$ will be the summation of the frequency of node *j*, $f[j]$ and the frequency of node *k*, $f[k]$.

Algorithm 3 Encoding of Hexanary Huffman Tree

```
H- HUFFMAN (C)
1.   Q ← C
2.   n ← |Q|
3.   i ← n
4.   WHILE I > 1
5.     allocate a new node z
6.     left1[z] ← j ← EXTRACT-MIN(Q)
7.     left2[z] ← k ← EXTRACT-MIN(Q)
8.     IF i = 2
9.       f[z] ← f[j] + f[k]
10.    ELSE IF i = 3
11.      left3 [z] ← l ← EXTRACT-MIN(Q)
12.      f[z] ← f[j] + f[k] + f[l]
13.    ELSE IF i = 4
14.      left3 [z] ← l ← EXTRACT-MIN(Q)
15.      left4 [z] ← m ← EXTRACT-MIN(Q)
16.      f[z] ← f[j] + f[k] + f[l] + f[m]
17.    ELSE IF i = 5
18.      left3 [z] ← l ← EXTRACT-MIN(Q)
19.      left4 [z] ← m ← EXTRACT-MIN(Q)
20.      left5[z] ← n ← EXTRACT-MIN(Q)
21.      f[z] ← f[j] + f[k] + f[l] + f[m] + f[n]
22.    ELSE IF i = 6
23.      left3 [z] ← l ← EXTRACT-MIN(Q)
24.      left4 [z] ← m ← EXTRACT-MIN(Q)
25.      left5[z] ← n ← EXTRACT-MIN(Q)
26.      left6[z] ← o ← EXTRACT-MIN(Q)
27.      f[z] ← f[j] + f[k] + f[l] + f[m] + f[n] + f[o]
28.    .
29.    .
30.    .
31.    .
32.    .
33.    .
34.    .
35.    .
36.    .
37.    .
38.    .
39.    .
40.    .
41.    .
42.    .
43.    .
44.    .
45.    .
46.    .
47.    .
48.    .
49.    .
50.    .
51.    .
52.    .
53.    .
54.    .
55.    .
56.    .
57.    .
58.    .
59.    .
60.    .
61.    .
62.    .
63.    .
64.    .
65.    .
66.    .
67.    .
68.    .
69.    .
70.    .
71.    .
72.    .
73.    .
74.    .
75.    .
76.    .
77.    .
78.    .
79.    .
80.    .
81.    .
82.    .
83.    .
84.    .
85.    .
86.    .
87.    .
88.    .
89.    .
90.    .
91.    .
92.    .
93.    .
94.    .
95.    .
96.    .
97.    .
98.    .
99.    .
100.   .
101.   .
102.   .
103.   .
104.   .
105.   ELSE
106.     left3 [z] ← j ← EXTRACT-MIN(Q)
107.     left4 [z] ← k ← EXTRACT-MIN(Q)
108.     left5 [z] ← l ← EXTRACT-MIN(Q)
109.     left6 [z] ← m ← EXTRACT-MIN(Q)
110.     left7 [z] ← n ← EXTRACT-MIN(Q)
111.     left8 [z] ← o ← EXTRACT-MIN(Q)
112.     right1 [z] ← p ← EXTRACT-MIN(Q)
113.     right2 [z] ← q ← EXTRACT-MIN(Q)
114.     right3[z] ← r ← EXTRACT-MIN(Q)
115.     right4[z] ← s ← EXTRACT-MIN(Q)
116.     right5[z] ← t ← EXTRACT-MIN(Q)
117.     right6[z] ← u ← EXTRACT-MIN(Q)
118.     right7[z] ← v ← EXTRACT-MIN(Q)
119.     right8[z] ← w ← EXTRACT-MIN(Q)
120.     f[z] ← f[j] + f[k] + f[l] + f[m] + f[n] + f[o] + f[p] + .. + f[y]
121.   END IF
122.   INSERT(Q, z)
123.   i ← |Q|
124. END WHILE
125. RETURN EXTRACT-MIN(Q)
```

For i equals 3 we extract another least frequent node from the queue and add it as $LEFT1, LEFT2, LEFT3$ child and add its frequency to the parent node. Likewise, for i equals 4 we extract four least frequent node from the queue and add it as $LEFT1, LEFT2, LEFT3, LEFT4$ child and add its frequency to the parent node. For i equals 5 we extract five least frequent node from the queue and add it as $LEFT1, LEFT2, LEFT3, LEFT4, LEFT5$ child and add its frequency to the parent node. For i equals 6 we extract six least frequent node from the queue and add it as $LEFT1, LEFT2, LEFT3, LEFT4, LEFT5, LEFT6$ child and add its frequency to the parent node. When i equals 7, we extract seven least frequent node from the queue and add it as $LEFT1, LEFT2, LEFT3, LEFT4, LEFT5, LEFT6, LEFT7$ child and add its frequency to the parent node. Likewise, for i equals 8 we extract eight least frequent node from the queue and add it as $LEFT1, LEFT2, LEFT3, LEFT4, LEFT5, LEFT6, LEFT7, LEFT8$ child and add its frequency to the parent node. The process will be continued and for i equals 16 we extract sixteen least frequent node from the queue and add it as $LEFT1, LEFT2, LEFT3, LEFT4, LEFT5, LEFT6, LEFT7, LEFT8, RIGHT1, RIGHT2, RIGHT3, RIGHT4, RIGHT5, RIGHT6, RIGHT7, RIGHT8$ child and add its frequency to the parent node. In line 122, we insert the new parent node z into the Queue, Q . In line 123, we take the count of the queue, Q and assign it to i again. And, the loop continues until a single node left in the queue. Finally, we return the last and single node from the queue Q as a Hexanary Huffman tree.

3.4.2 Decoding Algorithm

This is a one pass algorithm. First, we open the encoded file and read the frequency data out of it. We create the Binary, or the Quaternary, or the Octanary, or Hexanary Huffman tree based on that information. Data is read out of the file and search the tree to find the correct symbol (00 bit means go $LEFT$, 01 means go $LEFT MID$, 10 means go $RIGHT MID$ and 11 means go $RIGHT$ in case of quaternary tree, 000 bit means go $LEFT1$, 001 bit means go $LEFT2$, 010 bit means go $LEFT3$, etc in case of the Octanary tree; 0000 bit

means go *LEFT1*, 0001 bit means go *LEFT2*, 0010 bit means go *LEFT3*, etc in case of the Hexanary tree). If we know the Octanary or Hexanary Huffman code for some encoded data, decoding may be accomplished by reading the encoded data three or four bit at a time. Once the bits read match a code for a symbol, write out the symbol and start collecting bits again. The newly constructed Octanary and Hexanary tree decoding techniques are explained below.

3.4.2.1 Decoding of Quaternary Tree

Decoding is accomplished by reading the encoded data two bits at a time. When iterating the bit stream 00 bit pattern means go LEFT, 01 pattern means go LEFT MID, 10 pattern means go RIGHT MID and 11 pattern means go RIGHT in case of quaternary tree. When a bit pattern match with a symbol according to the header tree, replace the bit pattern with that symbol and the process is iterated until reached the last bit of the stream.

In the algorithm 4, in line 1, we assign the quaternary tree T in the local variable ln . Then we take the total count of bits in n from B . In line 3, we initialize a local variable i with 0 which will be used as a counter. In line 4, we started iterating all the bits in B . As it is a quaternary tree, we have at most four leaves for a parent node: left, left-mid, right-mid, right and 00, 01, 10, 11 represent these leaf nodes respectively. We take two bits at a time. $EXTRACT-BIT(B)$, returns a bit from the bit array B and removes it from B as well. In the line 5 and 6, local variable $b1$ and $b2$ is being assigned with two extracted bits from the bit array B .

From line 7 to line 15, we check the extracted bits to traverse the tree from the top. If the bits are 00 we take the left child of the parent ln and assign it to ln itself. For 01, we replace the parent ln with its left mid child, for 10 we replace it with its right mid child and for 11 we replace it with the right child. In line 16, we get the key of the replaced ln and assign it in k . Then, we check whether k has any value. If the k has any value we write the value of the k in

the output and update the ln with the quaternary tree T itself. In line 21 we increase the value of i by 2 and the loop gets continued and reads the next two bits.

Algorithm 4 Decoding of Quaternary Huffman Tree

```

DECODE (T, B)
1.      ln ← T
2.      n ← |B|
3.      i ← 0
4.      WHILE i < n
5.          b1 ← EXTRACT-BIT(B)
6.          b2 ← EXTRACT-BIT(B)
7.          IF b1 = 0 AND b2 = 0
8.              ln ← LEFT(ln)
9.          ELSE IF b1 = 0 AND b2 = 1
10.             ln ← LEFT-MID(ln)
11.          ELSE IF b1 = 1 AND b2 = 0
12.             ln ← RIGHT-MID(ln)
13.          ELSE
14.             ln ← RIGHT(ln)
15.          END IF
16.          k ← KEY(ln)
17.          IF k IS NOT NULL
18.              Output (k)
19.              ln ← T
20.          END IF
21.          i ← i + 2
22.      END WHILE

```

As discussed above, the search time for finding a source symbol using a quaternary Huffman technique is $O(\log_4 n)$ whereas for traditional Huffman based techniques decoding algorithm is $O(\log_2 n)$.

3.4.2.2 Decoding of Octanary Tree

In algorithm 5, in line 2, we assign the Octanary tree T in the local variable ln . After that the total count of bits in n from B is taken. In line 3, a local variable i with 0 is initialized which will be used as a counter.

In line 4, we start iterating all the bits in B . As it is an Octanary tree, we have at most eight leaves for a parent node: $LEFT1, LEFT2, LEFT3, LEFT4, RIGHT1, RIGHT2, RIGHT3, RIGHT4$ and 000, 001, 010, 011, 100, 101,

110, 111 represent these leaf nodes, respectively. So, we take three bits at a time. *EXTRACT-BIT(B)*, returns a bit from the bit array *B* and removes it from *B* as well. In line 5, 6 and 7, local variable *b1*, *b2* and *b3* are being assigned with three extracted bits from the bit array *B*.

Algorithm 5 Decoding of Octanary Huffman Tree

```

OH-DECODE (T, B)
1.   ln ← T
2.   n ← |B|
3.   i ← 0
4.   WHILE i < n
5.       b1 ← EXTRACT-BIT(B)
6.       b2 ← EXTRACT-BIT(B)
7.       b3 ← EXTRACT-BIT(B)
8.       IF b1 = 0 AND b2 = 0 AND b3=0
9.           ln ← LEFT1 (ln)
10.      ELSE b1 = 0 AND b2 = 0 AND b3=1
11.          ln ← LEFT2 (ln)
12.      ELSE b1 = 0 AND b2 = 1 AND b3=0
13.          ln ← LEFT3 (ln)
14.      ELSE b1 = 0 AND b2 = 1 AND b3=1
15.          ln ← LEFT4 (ln)
16.      ELSE b1 = 1 AND b2 = 0 AND b3=0
17.          ln ← RIGHT1 (ln)
18.      ELSE b1 = 1 AND b2 = 0 AND b3=1
19.          ln ← RIGHT2 (ln)
20.      ELSE b1 = 1 AND b2 = 1 AND b3=0
21.          ln ← RIGHT3 (ln)
22.      ELSE
23.          ln ← RIGHT4 (ln)
24.      END IF
25.      k ← KEY(ln)
26.      IF k IS NOT NULL
27.          Output (k)
28.          ln ← T
29.      END IF
30.      i ← i + 3
31.  END WHILE

```

From line 8 to line 24, we check the extracted bits to traverse the tree from the top. If the bits are 000 we take the *LEFT1* child of the parent *ln* and assign it to *ln* itself. For 001, we replace the parent *ln* with its *LEFT2* child, for 010 we replace it with its *LEFT3* child, for 011 we replace it with the *LEFT4* child, for 100 we replace it with its *RIGHT1* child, for 101 we replace it with its *RIGHT2* child, for 110 we replace it with its *RIGHT3* child

and for 111 we replace it with its *RIGHT4* child. In line 25, we get the key of the replaced *ln* and assign it in *k*.

Then, we check whether *k* has any value. If the *k* has any value we write the value of the *k* in the output and update the *ln* with the Hexanary tree *T* itself. In line 30 we increase the value of *i* by 3 and the loops get continued and read the next three bits. The search time for finding the source symbol using a Octanary Huffman Tree is $O(\log_8 n)$.

3.4.2.3 Decoding of Hexanary Tree

In algorithm 6, in line 1, we assign the Hexanary tree *T* in the local variable *ln*. After that the total count of bits in *n* from *B* is taken. In line 3, a local variable *i* with 0 is initialized which will be used as a counter. In line 4, we start iterating all the bits in *B*. As it is a Hexanary tree, we have at most sixteen leaves for a parent node: *LEFT1*, *LEFT2*, *LEFT3*, *LEFT4*, *LEFT5*, *LEFT6*, *LEFT7*, *LEFT8*, *RIGHT1*, *RIGHT2*, *RIGHT3*, *RIGHT4*, *RIGHT5*, *RIGHT6*, *RIGHT7*, *RIGHT8* and 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 represent these leaf nodes respectively. So, we take four bits at a time. *EXTRACT-BIT(B)*, returns a bit from the bit array *B* and removes it from *B* as well. In line 5, 6, 7 and 8, local variable *b1*, *b2*, *b3* and *b4* is being assigned with four extracted bits from the bit array *B*.

From line 9 to line 41, we check the extracted bits to traverse the tree from the top. If the bits are 0000 we take the *LEFT1* child of the parent *ln* and assign it to *ln* itself. For 0001, we replace the parent *ln* with its *LEFT2* child, for 0010 we replace it with its *LEFT3* child, for 0011 we replace it with the *LEFT4* child, for 0100 we replace the parent *ln* with its *LEFT5* child, for 0101 we replace it with its *LEFT6* child, for 0110 we replace it with the *LEFT7* child, for 0111 we replace it with its *LEFT8* child, for 1000 we replace it with its *RIGHT1* child, for 1001 we replace it with its *RIGHT2* child, for 1010 we replace it with its *RIGHT3* child, for 1011 we replace it with its *RIGHT4* child,

for 1100 we replace it with its *RIGHT5* child, for 1101 we replace it with its *RIGHT6* child, for 1110 we replace it with its *RIGHT7* child and for 1111 we replace it with its *RIGHT8* child. In line 42, we get the key of the replaced *ln* and assign it in *k*.

Algorithm 6 Decoding of Hexanary Huffman Tree

```

HH-DECODE (T, B)
1.  ln ← T
2.  n ← |B|
3.  i ← 0
4.  WHILE i < n
5.    b1 ← EXTRACT-BIT(B)
6.    b2 ← EXTRACT-BIT(B)
7.    b3 ← EXTRACT-BIT(B)
8.    b4 ← EXTRACT-BIT(B)
9.    IF b1 = 0 AND b2 = 0 AND b3=0 AND b4=0
10.     ln ← LEFT1 (ln)
11.   ELSE b1 = 0 AND b2 = 0 AND b3=0 AND b4=1
12.     ln ← LEFT2 (ln)
13.   .
14.   .
15.   .
16.   .
17.   .
18.   .
19.   .
20.   .
21.   .
22.   .
23.   .
24.   .
25.   IF b1 = 1 AND b2 = 0 AND b3=0 AND b4=0
26.     ln ← RIGHT1 (ln)
27.   ELSE b1 = 1 AND b2 = 0 AND b3=0 AND b4=1
28.     ln ← RIGHT2 (ln)
29.   .
30.   .
31.   .
32.   .
33.   .
34.   .
35.   .
36.   .
37.   .
38.   .
39.   ELSE
40.     ln ← RIGHT8 (ln)
41.   END IF
42.   k ← KEY(ln)
43.   IF k IS NOT NULL
44.     Output (k)
45.     ln ← T
46.   END IF
47.   i ← i + 4
48. END WHILE

```

Then, we check whether *k* has any value. If the *k* has any value we write the value of the *k* in the output and update the *ln* with the Hexanary tree *T* itself. In line 47 we increase the value of *i* by 4 and the loops get continued and read the next four bits.

Encoding and Decoding Techniques of Quaternary, Octanary and Hexanary techniques have been thoroughly discussed in this section. The search time for finding the source symbol using Quaternary, Octanary and Hexanary Huffman Tree is $O(\log_4 n)$, $O(\log_8 n)$ and $O(\log_{16} n)$, respectively, whereas for traditional Huffman based techniques decoding algorithm it is $O(\log_2 n)$. The codeword generated by each technique are shown in Table 1.

3.4.3 Why Choosing Quaternary tree to generate dictionary code

Three new Huffman based algorithms have been introduced in this chapter. The time-space trade-off for different Huffman based algorithms have been thoroughly discussed. Binary Huffman algorithm performs better for achieving more compression ratio. Quaternary Huffman algorithm is useful when a balance between time and space is required. However, Octanary and Hexanary Huffman algorithms perform superior to Binary and Quaternary algorithms in terms of decoding speed but the compression ratio is significantly decreased in both cases.

To make a balance between time and space we choose quaternary tree structure for producing dictionary codeword for our compression algorithm. To produce quaternary code stream using Huffman principle requires a shorter Huffman tree, i.e., less depth. The potential benefit of a shorter Huffman tree is less traverse time, which could improve both compression and decompression throughput.

In this chapter, the structure of quaternary tree is investigated along with other Huffman tree for construction of Huffman code. A number of parameters like height, diameter, and weighted path length of the tree are studied. A remark has also been proved to show that quaternary tree is better than binary tree for constructing of Huffman code. It is found that the traversing time of quaternary tree is approximately one third of that of binary tree in the worst case and exactly half in the best case.

3.5. Summary

In this thesis, we propose a new compression algorithm using quaternary Huffman coding that makes use of a variation of the classic Huffman coding. Using quaternary Huffman coding, each symbol is encoded into a quaternary code stream, instead of a binary bit stream. Here, at first the properties of quaternary tree structure for construction of Huffman codes are mathematically investigated. The terminology of new tree structure is explained thoroughly and the conjectures are proved as remarks. In parallel to quaternary tree structure we have also investigated octanary and hexanary tree structure. The encoding and decoding technique of all tree structures have been developed and explained carefully in this chapter.

Chapter 4

Implementation

This chapter describes the details implementation technique of the proposed text compression architecture. The dictionary generation, encoding, and decoding algorithms are explained in this chapter. The complexity of different algorithms is also analyzed in details in this chapter. The presence of characters in English text are also observed carefully from different corpora to create dictionary entry.

4 Implementation

We have implemented this research in three steps. First, we have produced a dictionary based on the quaternary code. Secondly, we have encoded data based on the newly produced dictionary. Thirdly, we have decoded data with four normalized dictionaries. For these three steps, three algorithms have been developed which are discussed in section 4.1.

Frequency of symbols of a language is very important for creating a dictionary. Huffman principle produces codeword for a symbol based on its frequency in the text. In this context, we have developed frequency counter software to calculate the frequency. We have used Brown (The Brown Corpus, 2018), Canterbury (Bell & Powel, 2000), English part of an English-Bengali parallel corpus SUPara (Mumin *et al.*, 2012), and Enwik (Mahoney, 2018) corpora to calculate the frequency. There are 87, 96, 106, and 205 distinct symbols in Canterbury, Brown, SUPara, and Enwik corpora respectively.

We require all the symbols available in the keyboard (small letters, capital letters, big key, punctuations, numbers, etc.), so that we can create a complete dictionary. We have tested the accuracy of our frequency counter with existing tools. It is found that our tool is working fine, the results are shown in Figure

27. It is obvious that the repetitions of English symbols in different corpora are almost the same. Moreover, it is easy to check that the n th character (from bottom to top) in the frequency distribution table occurs with the probability $\frac{1}{n^2}$ (approximately), for some constant $k < n$.

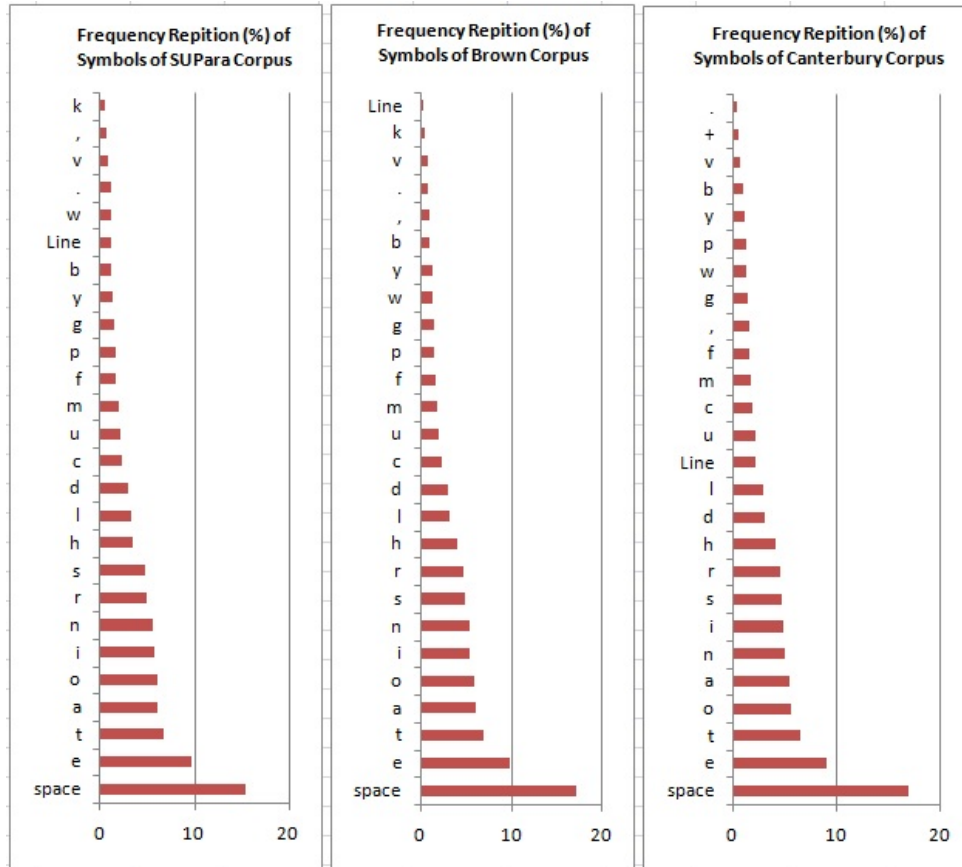


Figure 27 Frequency distribution of Canterbury, Brown and SUPara corpora.

4.1 Dictionary Generation Algorithm

We have produced a character based dictionary rather than traditional word-based or syllable based dictionary. Because the numbers of words or syllables are undefined; sometimes it may reach more than 100 millions of words for a monolingual corpus, whereas the number of characters or symbols is fixed. We have 205 symbols for the first version of the dictionary. The dictionary is produced using the principle of Huffman on quaternary tree architecture. The

dictionary produced by modified Huffman Algorithm is shown partly in Figure 28 (named Dic).

An algorithm is developed to generate quaternary Huffman codeword. The algorithm used for generating codeword is shown in Algorithm 1 (Habib & Rahman, 2017). If m and n is the total number of nodes in quaternary and binary tree respectively, then the run time complexity of the quaternary tree based algorithm is $O(m\log_4 n)$, whereas for Huffman based algorithm is $O(n\log_2 n)$. Here m is always less than n , because the total number of nodes in a quaternary tree is less than that of a binary tree for the same number of input symbols. For Enwik corpus, the average codeword length of the dictionary using quaternary code is 4.726, whereas the average codeword length of the dictionary using binary codes is 4.639. The weighted path length of the quaternary tree is 3501553, whereas the weighted path length of the binary tree is 6873872. The searching becomes faster when the petite quaternary tree is used.

The detail of this algorithm is already explained in chapter 3 section 3.4.1.1.

Dic		Dic 1		Dic 3		Dic 4																															
Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code																														
Space	00	Space	00	a	1010	e	1110																														
e	1110	<table border="1"> <thead> <tr> <th colspan="2">Dic 2</th> </tr> <tr> <th>Symbol</th> <th>Code</th> </tr> </thead> <tbody> <tr> <td>n</td> <td>0111</td> </tr> <tr> <td>r</td> <td>0101</td> </tr> <tr> <td>s</td> <td>0100</td> </tr> <tr> <td>y</td> <td>011010</td> </tr> <tr> <td>b</td> <td>011001</td> </tr> <tr> <td>Line</td> <td>011000</td> </tr> <tr> <td>S</td> <td>01101111</td> </tr> <tr> <td>I</td> <td>01101111</td> </tr> <tr> <td>T</td> <td>01101100</td> </tr> <tr> <td>O</td> <td>0110111011</td> </tr> <tr> <td>F</td> <td>0110111010</td> </tr> <tr> <td>?</td> <td>0110111001</td> </tr> <tr> <td>Tab</td> <td>0110111000</td> </tr> </tbody> </table>		Dic 2		Symbol	Code	n	0111	r	0101	s	0100	y	011010	b	011001	Line	011000	S	01101111	I	01101111	T	01101100	O	0110111011	F	0110111010	?	0110111001	Tab	0110111000	o	1001	t	1100
Dic 2																																					
Symbol	Code																																				
n	0111																																				
r	0101																																				
s	0100																																				
y	011010																																				
b	011001																																				
Line	011000																																				
S	01101111																																				
I	01101111																																				
T	01101100																																				
O	0110111011																																				
F	0110111010																																				
?	0110111001																																				
Tab	0110111000																																				
t	1100	i	1000	h	111110																																
a	1010	f	101110	l	111101																																
o	1001	p	101101	d	111100																																
i	1000	g	101100	c	110110																																
n	0111	k	10111110	u	110101																																
r	0101	G	1011111111	m	110100																																
s	0100	D	1011111110	w	11111111																																
h	111110)	1011111101	.	11111101																																
l	111101	H	1011111100	v	11011111																																
d	111100	(1011110111	,	11011101																																
c	110110	W	1011110110	A	1111111001																																
u	110101	E	1011110101	B	1111111000																																
m	110100	R	1011110100	C	1111110011																																
f	101101	q	1011110011	P	1111110001																																
p	101101	N	1011110010	l	1111110000																																
.																																
.																																
.																																

Figure 28 Dictionaries.

4.2 Encoding Algorithm

As a prerequisite of the encoding process, a static dictionary is created using quaternary Huffman algorithm. Every character of the text is replaced by its codeword from the dictionary. We have developed an algorithm to convert a string to a binary codeword is shown in Algorithm 7.

In the algorithm 7, in line 1, T is the input text file and in line 2, N is the length of text file T . In line 3, $BITSTRING$ is the string variable with a NULL value. In line 4, there is a loop. Here, each character of Text T matches with dictionary and stores its $CODEWORD$ into $BITSTRING$ variable with its previous value. The process continues until the last character with its $CODEWORD$ is added into $BITSTRING$. In line 7, $BITSTRING$ is converted into unsigned short integers and stored into a binary file.

Algorithm 7 Encoding Algorithm

ENCODE (Text File, Dictionary)	
1.	Set T=Text file
2.	Set N=Length of T
3.	Set BITSTRING=NULL
4.	FOR i=0 to N-1
5.	match T[i] with dictionary character
6.	Set BITSTRING = BITSTRING+CODEWORD of character
7.	END FOR
8.	Convert BITSTRING into unsigned short integers and store it into a binary file
9.	EXIT

The encoding algorithm is very simple and fast. An efficient linear search is used and its complexity is less than $O(n)$. Since the n th character in the frequency with the probability is $\frac{k}{n^2}$ (approximately), for some constant $k < n$, for linear search we can write,

$$\begin{aligned}
 C(n) &= 1 \cdot \frac{k}{n^2} + 2 \cdot \frac{k}{n^2} + \dots \dots \dots + n \cdot \frac{k}{n^2} \\
 &= (1 + 2 + \dots \dots \dots + n) \cdot \frac{k}{n^2} \\
 &= (1 + 2 + \dots \dots \dots + n) \cdot \frac{k}{n^2}
 \end{aligned}$$

$$\begin{aligned}
&= \frac{n(n+1)}{2} \cdot \frac{k}{n^2} \\
&= \frac{k}{2} \cdot \left(1 + \frac{1}{n}\right) \\
&\approx \frac{k}{2} < O(n)
\end{aligned}$$

This optimal result is achieved just because of the formation of the dictionary based on the quaternary Huffman principle. It is found that top 10 characters out of 205 characters occupy almost 70% of the text file and we have traversed top positions of the dictionary most of the time.

4.3 Decoding Algorithm

The dictionary produced by the modified Huffman algorithm is divided into four dictionaries for faster search which is shown partly in Figure 28 (named Dic_1, Dic_2, Dic_3 and Dic_4). The codeword started with 00, 01, 10, 11 takes place in Dictionary 1, Dictionary 2, Dictionary 3, and Dictionary 4 respectively.

In all the cases, top entries of dictionaries contain most frequent symbols which ensure faster search during the decoding process. Decoding is started by reading two bits at a time from the encoded file. All the quaternary codes are available in four normalized dictionaries. When a bit pattern matches with any codeword in the dictionary, the respective symbol is replaced with the codeword. The process is iterated until reaching the last two bits of the file. The decoding algorithm is shown in Algorithm 8.

In the algorithm 8, in line 1, the binary file is read and the unsigned short integer numbers are taken as input. In line 2, unsigned short integer numbers are converted into bits and are set into a variable named *BITSTRING*. In line 3, a new variable *N* is declared as the length of *BITSTRING* variable. In line 4, there is a loop to search for any one of the dictionaries. After extracting two bits, the searching process is started in a particular dictionary.

Algorithm 8 Decoding Algorithm

```
DECODE(Encoded Binary File, Dictionary)
1. Read the binary file and take input the unsigned short integers number
2. Set BITSTRING=convert unsigned short integers into bits
3. Set N=length of BITSTRING
4. FOR i=0 to N-1
5.     WORD← extract first 2 bit from BITSTRING
6.     IF WORD= '00' Dictionary_No←1
7.     ELSE IF WORD= '01' Dictionary_No←2
8.     ELSE IF WORD= '10' Dictionary_No←3
9.     ELSE IF WORD= '11' Dictionary_No←4
10.    END IF
11.    Set i=i+2
12.    WHILE(TRUE)
13.        SET k=0
14.        IF WORD.size>= (Dictionary_No)[0].CODEWORD length
15.            WHILE k < Length of (Dictionary_No)
16.                IF (Dictionary_No)[k].size of CODEWORD >WORD.size
17.                    Break the loop
18.                IF (Dictionary_No)[k].CODEWORD = WORD
19.                    Write the character into decoded file for that CODEWORD
                    from (Dictionary_No) and break the loop in 15
20.                    Set k=k+1
21.            END WHILE
22.            IF (Dictionary_No)[k].CODEWORD match word then break the loop in 12
23.            IF i+2 < N
24.                Set WORD= WORD+( BITSTRING[i+1] + BITSTRING[i+2])
25.                Set i=i+2
26.            ELSE
27.                Break the loop in 12
28.            END IF
29.        ELSE
30.            IF i+2 < N
31.                Set WORD= WORD+( BITSTRING[i+1] + BITSTRING[i+2])
32.                Set i=i+2
33.            ELSE
34.                Break the loop in 12
35.            END IF
36.        END IF
37.    END WHILE
38.    Set i=i-1
39. END FOR
40. EXIT
```

In line 11, loop iterator is increased by 2. In line 12, a loop is run until the expected character is found. In line 13, an index variable named k is initialized with 0. In line 14, we check the current $WORD$ length whether it exists in the dictionary. In line 15, there is a loop to search the whole dictionary. In line 16, if the $WORD$ size is greater than the existing $CODEWORD$ size then we break

the loop in line 15. In line 18, if the *WORD* matches with the dictionary *CODEWORD*, we write the character of respective *CODEWORD* into the decoded text and break the loop in line 15. In line 20, we increase the index variable by 1. In line 21, if the character is found then we break the loop in line 12. In line 22, if the iterator $i + 2$ is less than N then we update the *WORD* by adding next 2 bits from *BITSTRING*. In line 24, we increase the iterator i by 2. In line 26, we end the loop in line 12. In line 27, we continue the process line 22 to 26. In line 33, we decrease the iterator value i by 1.

For matching a codeword with its symbol, we have to visit any one of four dictionaries. Most of the cases we found the symbols on the top of the dictionary because most frequent character stays at the top position of the dictionary. The symbols are set in order of decreasing probability in the proposed dictionary. The probability of these symbols is geometrically distributed. For this case, the run time complexity is less than $O(n)$. If l equals half of the maximum code length, then for linear search we can write,

$$\begin{aligned}
C(n) &\leq \frac{1}{4} \cdot l \left(1 \cdot \frac{1}{n^2} + 2 \cdot \frac{1}{n^2} + \dots + n \cdot \frac{1}{n^2} \right) \\
&\leq \frac{l}{4} \cdot (1 + 2 + \dots + n) \cdot \frac{1}{n^2} \\
&\leq \frac{l}{4} \cdot (1 + 2 + \dots + n) \cdot \frac{1}{n^2} \\
&\leq \frac{l}{4} \cdot \frac{n(n+1)}{2} \cdot \frac{1}{n^2} \\
&\leq \frac{l}{4} \cdot \frac{n+1}{2n} \\
&\leq \frac{l \cdot (n+1)}{8n} \\
&\approx \frac{l}{8} < O(n) (\because l < n)
\end{aligned}$$

Encoding and decoding processes of the proposed technique have been thoroughly discussed in this section. The search time for finding a symbol in

decoding algorithm is $O\left(\frac{L(n+1)}{8n}\right)$, whereas for traditional Huffman based techniques complexity of decoding algorithm is $O(n \log_2 n)$.

4.4 Summary

In this chapter we have presented an attractive dictionary based text compression architecture using quaternary code. After a statistical analysis of the English language; we design a variable length dictionary based on quaternary codes. The dictionary generation algorithm explained in this chapter. The algorithms of encoding and decoding operations given in this chapter have been thoroughly discussed.

Chapter 5

Result and Discussion

The objective of the experimental work is to verify the applicability and feasibility of the proposed architecture. The experimental evaluation has been performed with real data. The experimental result is compared with many widely used popular compression algorithms. Our target was to develop a text compression technique and justify the storage requirements and query time in comparison with recent compression tools.

5. Performance analysis

5.1. Methods

The test computer we used is an Intel® Core™ i5 – 6500 CPU running at 3.20 GHz with 2 cores and 4 additional hyper threading contexts. We ran Ubuntu 14.04 LTS Operating system. All codecs were compiled using the same compiler, GCC 4.8.4. The amount of primary memory is 4 GB DDR4 type.

The versions of the following algorithms are tested in our experiment:

- Zopfli version 2015-09-01 (Zopfli Source Code, 2018), Google claims that it has the highest compression ratio,
- LZHAM, an advanced compression algorithm (LZHAM Source, 2018),
- bzip2 1.0.6 6-Sept-2010 (bzip2 1.0.6, 2010); an open source compression program,
- LZMA implementation in 7zip 9.20.1 (LZMA Source, 2018), is an algorithm used to perform lossless data compression, and

- TCQC - the proposed technique, a text compression technique using Quaternary Huffman based dictionary.

5.2. Test Corpora

The compression corpora we used in the testing are:

- the Canterbury, an ad hoc crawled web content corpus, 1285 files, 70611753 bytes total, and this corpus is a modified version of Calgary corpus which is designed to test the compression algorithms,
- the Enwik8, a single file corpus, the Enwik corpus is a 95:3 MB file with 205 distinct characters,
- the SUPara, an English - Bengali parallel corpus produced by Shahjalal University of Science and Technology (SUST), Bangladesh, and
- the Brown, a modern, computer readable, general corpus consists of one million words of American English texts.

We have measured the compression ratio, compression speed in (MB/S) and decompression speed in (MB/S) for selected algorithms and compression levels. The compression and decompression speed of each algorithm are measured with the same environment and the same compiler. Zopfli only compresses and does not decompress, for measuring decompression speed of Zopfli we used gzip-9 (Deutsch, 1996) as decompression which is compatible with Zopfli. Unlike other algorithms compared here, TCQC includes a static dictionary. It contains 205 characters of English Wikipedia. We can extend the static dictionary by a mechanism of transforms that slightly changes the characters in the dictionary. In averaging over the results of individual files and over the corpus we have considered five consecutive runs.

5.3. Results

Table 3, Table 4, Table 5 and Table 6 show the results for Enwik, Canterbury, SUPara, and Brown corpus respectively.

Table 3 Performance analysis for Enwik corpus.

Algorithm	Compression Ratio	Compression Speed (MB/s)	Decompression Speed (MB/s)
Zopfli-1.0.1	2.855	0.4471	154.1818
Lzham:1	3.335	1.7886	98.7711
Lzham:4	3.643	0.2981	106.7106
bzip2:1	3.007	9.1665	17.1004
bzip2:9	3.447	9.2411	16.8228
Lzma:1	3.106	7.3034	33.6457
Lzma:9	3.639	2.5637	39.8640
TCQC	2.525	1.7083	159.1843

We can see from Table 3 that the decompression speed of TCQC is faster than other algorithms for EnWik8 corpus. The experimental result shows that for Enwik corpus the decompression speed of TCQC is faster than all techniques. It is approximately 1:5, 8, and 4 times faster than Lzham, bzip2, and Lzma respectively which is shown in Table 3. The compression speed of bzip2:9 is the highest of all for the EnWik8 corpus. The compression ratio versus decompression speed is also shown in figure 29, where we have seen that the decompression speed of TCQC is the highest of all.

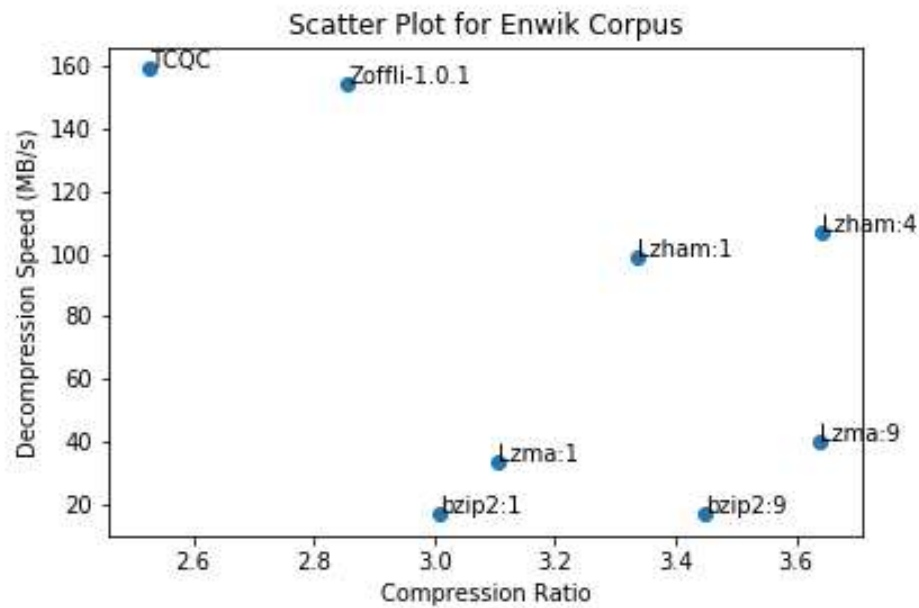


Figure 29 Compression ratio versus decompression speed for Enwik corpus

Table 4 Performance analysis for Canterbury corpus.

Algorithm	Compression Ratio	Compression Speed (MB/s)	Decompression Speed (MB/s)
Zopfli-1.0.1	3.580	0.1491	189.9373
Lzham:1	3.836	2.9064	64.4043
Lzham:4	3.952	0.3726	65.3482
bzip2:1	3.757	8.7939	22.4304
bzip2:9	3.869	8.9431	22.3194
Lzma:1	3.847	7.6015	38.8647
Lzma:9	4.240	2.9064	39.8085
TCQC	3.166	1.6671	80.2001

For Canterbury corpus, the Zopi has highest decompression speed and the TCQC is second highest but the compression speed of TCQC is better than Zopi which are shown in Table 4. In Table 4, for Canterbury corpus, the decompression speed of the proposed technique is approximately 1:25, 4, and 2 times faster than Lzham, bzip2, and Lzma respectively. The compression speed of lzma:1 is the highest of all for Canterbury corpus. From figure 30 we have seen that the decompression speed of TCQC is the second highest of all algorithms.

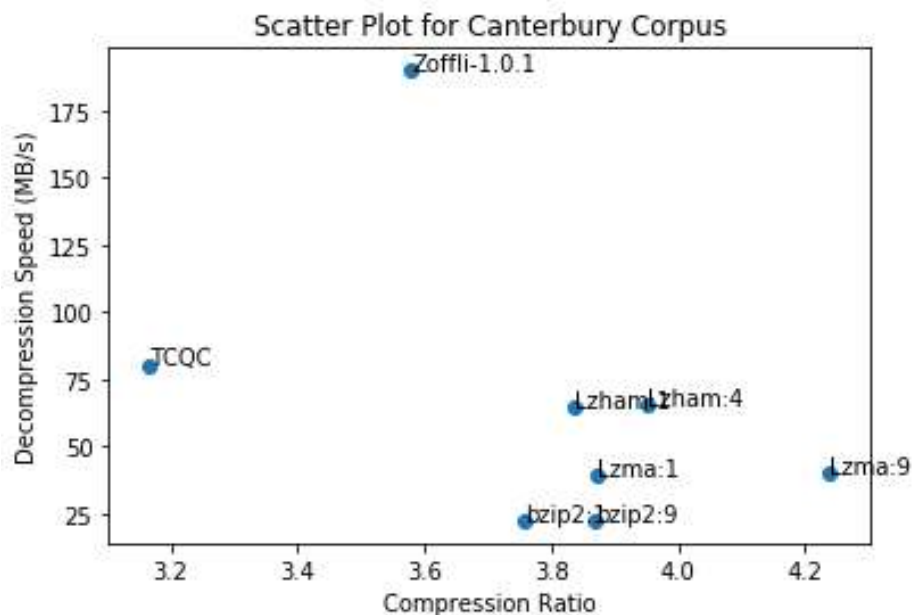


Figure 30 Compression ratio versus decompression speed for Canterbury corpus

Table 5 Performance analysis for SUPara corpus.

Algorithm	Compression Ratio	Compression Speed (MB/s)	Decompression Speed (MB/s)
Zopfli-1.0.1	3.393	0.2369	89.9035
Lzham:1	3.961	0.9476	70.2426
Lzham:4	4.326	0.1579	75.8871
bzip2:1	3.571	4.8564	12.1608
bzip2:9	4.093	4.8959	11.9634
Lzma:1	3.688	3.8693	16.1280
Lzma:9	4.389	1.3582	22.6404
TCQC	3.001	0.6732	62.8357

We also test the proposed technique using a bilingual SUPara corpus, the performance is also good comparing with bzip2 and lzma which is shown in Table 5. In Table 5, for SUPara corpus, the decompression speed of the proposed technique is approximately 5, and 6 times faster than bzip2, and Lzma respectively. The compression speed of bzip2:9 is the highest of all for this corpus. The compression ratio versus decompression speed of SUPara corpus is shown in figure 31.

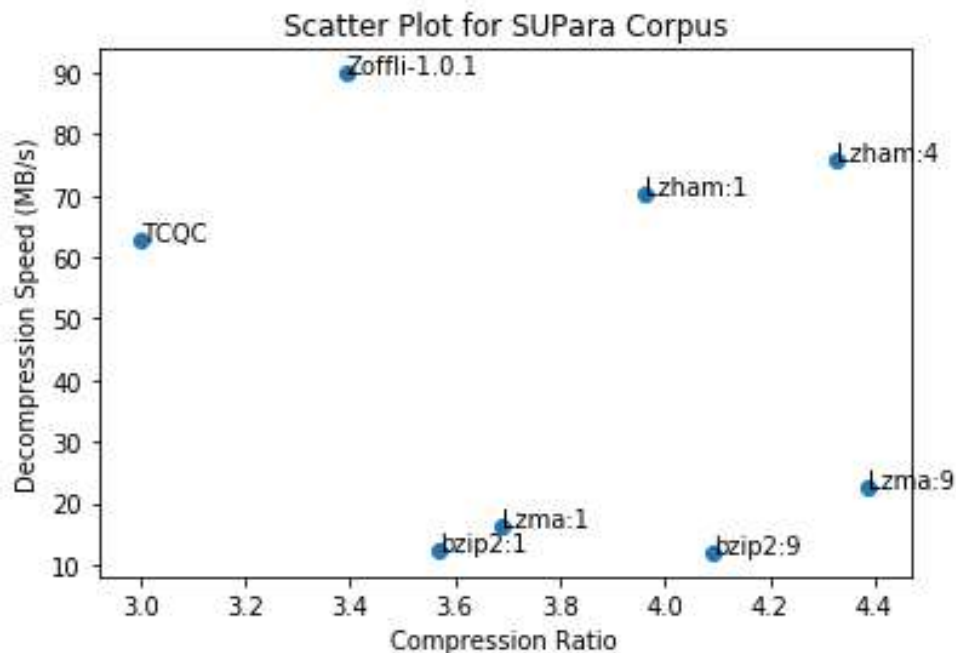


Figure 31 Compression ratio versus decompression speed for SUPara corpus

Table 6 Performance analysis for Brown corpus.

Algorithm	Compression Ratio	Compression Speed (MB/s)	Decompression Speed (MB/s)
Zopfli-1.0.1	2.707	0.2235	255.4521
Lzham:1	3.159	3.5026	93.6639
Lzham:4	3.451	0.1491	95.8847
bzip2:1	2.849	8.1977	29.0374
bzip2:9	3.265	8.2722	29.0374
Lzma:1	2.942	5.8874	55.7986
Lzma:9	3.501	3.2791	56.7424
TCQC	2.394	0.6353	178.5414

For Brown corpus, the Zopi has highest decompression speed and the TCQC is second highest but the compression speed of TCQC is better than Zopi which are shown in Table 6. In Table 6, for Brown corpus, the decompression speed of the proposed technique is approximately 2, 6, and 3 times faster than Lzham, bzip2, and Lzma respectively. The compression speed of bzip2:9 is the highest of all for this corpus. Though the compression ratio of TCQC is the lowest for Brown corpus, the decompression speed of TCQC is the second highest of all, which is shown in figure 32.

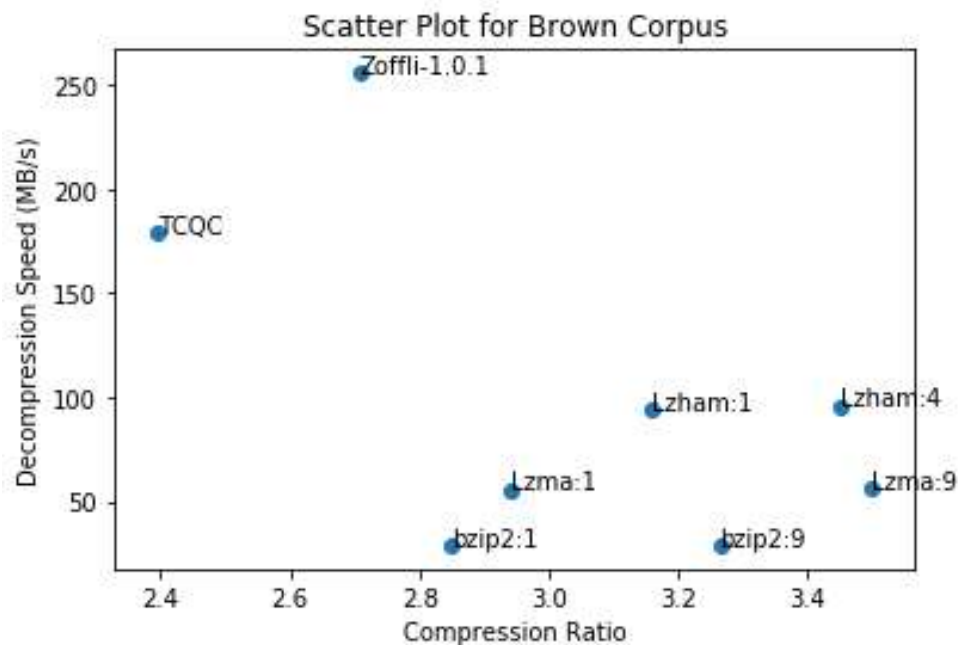


Figure 32 Compression ratio versus decompression speed for Brown corpus

From Table 4 and Table 6, we observe that the TCQC is little slower than only for Zopfli but the compression speed of TCQC is better than Zopfli for any corpora. Experimental results show that the decompression speed of the proposed technique is better than widely used existing techniques, whereas the space requirement is slightly increased.

The existing Huffman based algorithms use single bit (binary) code to store data in the memory, which slow the decoding speed. The proposed algorithm use dibit (quaternary) code to store data. Retrieving two bits at a time from the memory during decoding process speeds up the process. Moreover, if the tree is balanced, and the number of symbols is approximately 4^h , where h is the height of the tree, the proposed algorithm performs well.

5.4 The reason for choosing quaternary code dictionary

The objective of this experiment is to evaluate the performance of quaternary, octanary and hexanary Huffman based algorithms. We consider Zopfli (Alakuijala & Vandevenne, 2013; Alakuijala *et al.*, 2016) as a traditional (Binary) Huffman algorithm. Zopfli is one of the most successful compression algorithm released by Google Inc. Google claims that Zopfli has the highest compression ratio. We also compare the performance of the dibit based Quaternary algorithm and the proposed tribit based Octanary and quadbit based Hexanary Huffman algorithms. The dataset used in this experiment to verify the performance of different Huffman based algorithms are described in Table 7.

As shown in Table 8, it is observed that compression ratio is highest for Zopfli but the respective compression and decompression speed is very slow. The Zopfli requires over 400 sec whereas all other proposed techniques require less than 200 sec. For the Canterbury corpus, Zopfli requires over 13 sec whereas all other proposed techniques require less than 2 sec, which is shown in Table 9.

Table 7 Data set

S / L	File Name	Description	File Size	Distinct Symbol
1	Enwik8.txt	It has been developed as a large text compression benchmark, consisting of 100 million bytes of English Wikipedia	95.3 MB	156
2	Canterbury.txt	A compression corpus designed for lossless data compression, Improved version of Calgary corpus	2.67 MB	72
3	Demo.txt	This document demonstrates the ability of the calibre DOCX Input plugin to convert the various typographic features in a Microsoft Word	3.03 MB	81
4	Bht_Bn_Corpora.txt	A Bengali Corpus	641 KB	166
5	NLP07_National1.txt	A file from SUST Corpus	4.16 MB	109

Table 8 The compression ratio and compression-decompression speed for the Enwik Corpus

Algorithm	Space (MB)	Compression Enhancement W.R.T OF (%)	Time (S)	Time Enhancement W.R.T Zopfli(%)
Zopfli (Binary)	33.37	64.98	463.26	-
Quaternary	49.67	47.88	186.88	59.66
Octanary	61.06	35.93	187.82	59.46
Hexanary	59.73	37.32	174.58	62.31

Table 9 The compression ratio and compression-decompression speed for the Canterbury corpus.

Algorithm	Space (MB)	Compression Enhancement W.R.T OF (%)	Time (S)	Time Enhancement W.R.T Zopfli(%)
Zopfli (Binary)	0.64	76.07	13.36	-
Quaternary	1.71	35.85	1.37	89.78
Octanary	2.27	15.01	1.47	89
Hexanary	1.79	32.97	1.04	92.2

The performance of different algorithms is shown in Table 8 and 9 for Enwik (Mahoney, 2018) and Canterbury (Bell & Powel, 2000) corpora, respectively. From the both tables, it is shown that the valuation of two different parameter space and time are not same. In some cases saving space is more important and in some other cases speed (time) is important. To see a time-space relation at the same time, we normalize the data. If we divide every number by the largest number of the range, we will get every number in the range between 0 and 1. The data before and after normalization for Enwik corpus is shown in Table 10 and the time-space graph is shown in Figure 33.

Table 10 Time-space data for Enwik corpus

Before normalization			After normalization		
Algorithm	Space (MB)	Time (S)	Algorithm	Space (MB)	Time (S)
Zopfli (Binary)	33.37	463.26	Zopfli (Binary)	0.55	1
Quaternary	49.67	186.88	Quaternary	0.81	0.4
Octanary	61.06	187.82	Octanary	1	0.41
Hexanary	59.73	174.58	Hexanary	0.98	0.38

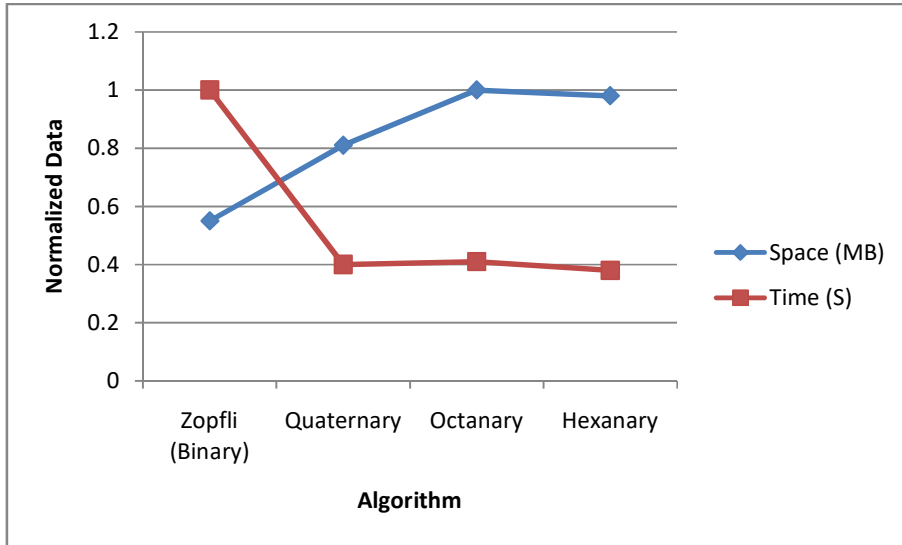


Figure 33 Time-space graph for Enwik corpus

From Figure 33, it has been shown that Zopfli requires maximum time whereas Quaternary, Octanary or Hexanary requires less time. In the Quaternary technique, it achieves almost 60% speed improvement with sacrificing 17% of space. For Octanary technique, it achieves almost 59% more speed with sacrificing 29% of space. For Hexanary technique, it achieves almost 62% more speed with sacrificing 44% of space.

Table 11 Time-space data for Enwik corpus

Before normalization			After normalization		
Algorithm	Space (MB)	Time (S)	Algorithm	Space (MB)	Time (S)
Zopfli (Binary)	0.64	13.36	Zopfli (Binary)	0.28	1
Quaternary	1.71	1.37	Quaternary	0.75	0.11
Octanary	2.27	1.47	Octanary	1	0.11
Hexanary	1.79	1.04	Hexanary	0.78	0.08

The data before and after normalization for Canterbury corpus is shown in Table 11 and the time-space graph is shown in Figure 34.

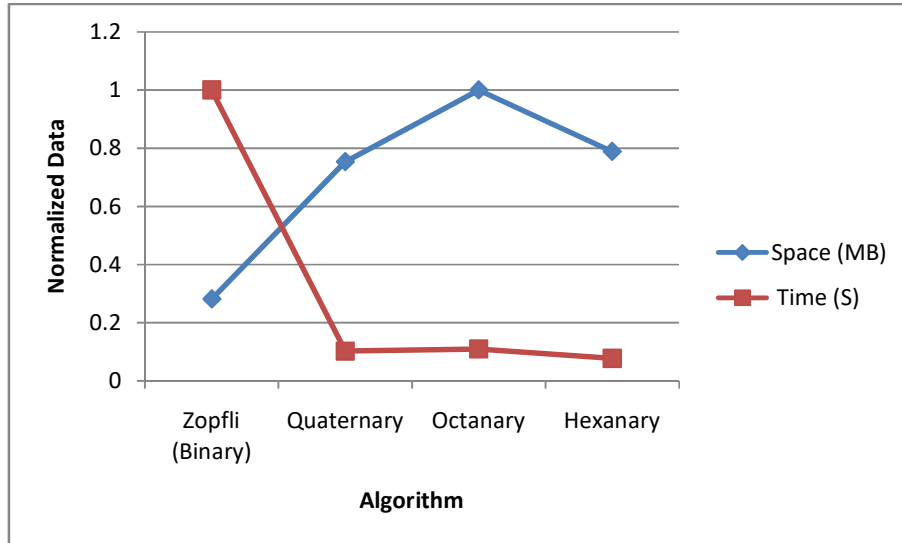


Figure 34 Time-space graph for Canturbury corpus

From Figure 34, it has been shown that Zopfli requires maximum time whereas Quaternary, Octanary or Hexanary requires less time. In the Quaternary technique, it achieves almost 90% speed improvement with sacrificing 167% of space. For Octanary technique, it achieves almost 90% more speed with sacrificing 254% of space. For Hexanary technique, it achieves almost 92% more speed with sacrificing 179% of space.

Table 12 Time-space data for Bht_bn file

Before normalization			After normalization		
Method	Space (MB)	Time (S)	Method	Space (MB)	Time (S)
Binary	0.31	5.69	Binary	0.81	1
Quaternary	0.32	3.13	Quaternary	0.84	0.55
Octanary	0.39	2.75	Octanary	1	0.48
Hexanary	0.38	2.14	Hexanary	0.99	0.38

The data before and after normalization for Bht_bn file is shown in Table 12 and the time-space graph is shown in Figure 35.

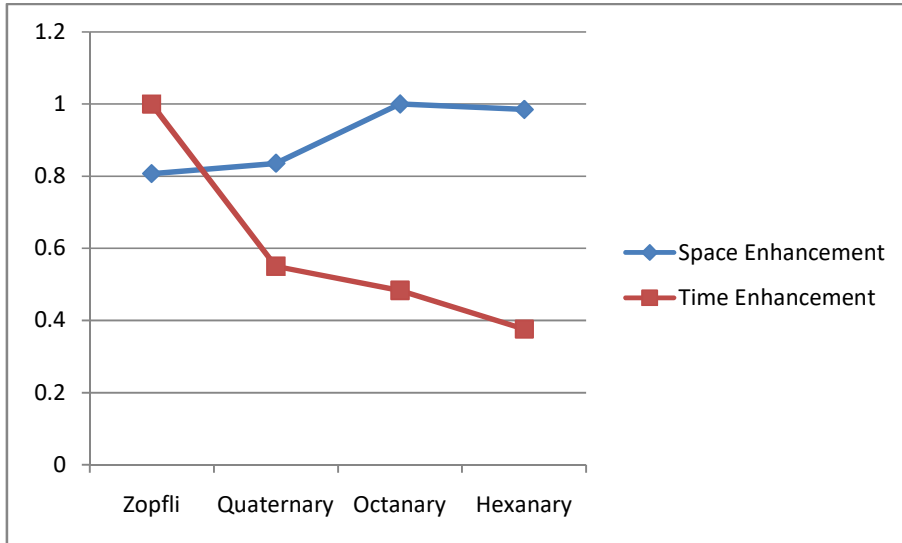


Figure 35 Time-space graph for “Bht_bn”

From Figure 35, it has been shown that Zopfli requires maximum time whereas Quaternary, Octanary or Hexanary requires less time. In the Quaternary technique, it achieves almost 44% speed improvement with sacrificing 4% of space. For Octanary technique, it achieves almost 52% more speed with sacrificing 24% of space. For Hexanary technique, it achieves almost 62% more speed with sacrificing 22% of space.

Table 13 Time-space data for NLP07 file

Before normalization			After normalization		
Method	Space (MB)	Time (S)	Method	Space (MB)	Time (S)
Binary	1.86	21.41	Binary	0.61	1
Quaternary	1.88	12.54	Quaternary	0.62	0.58
Octanary	2.45	13.26	Octanary	0.80	0.62
Hexanary	3.04	15.24	Hexanary	1	0.71

The data before and after normalization for NLP07 file is shown in Table 13 and the time-space graph is shown in Figure 36.

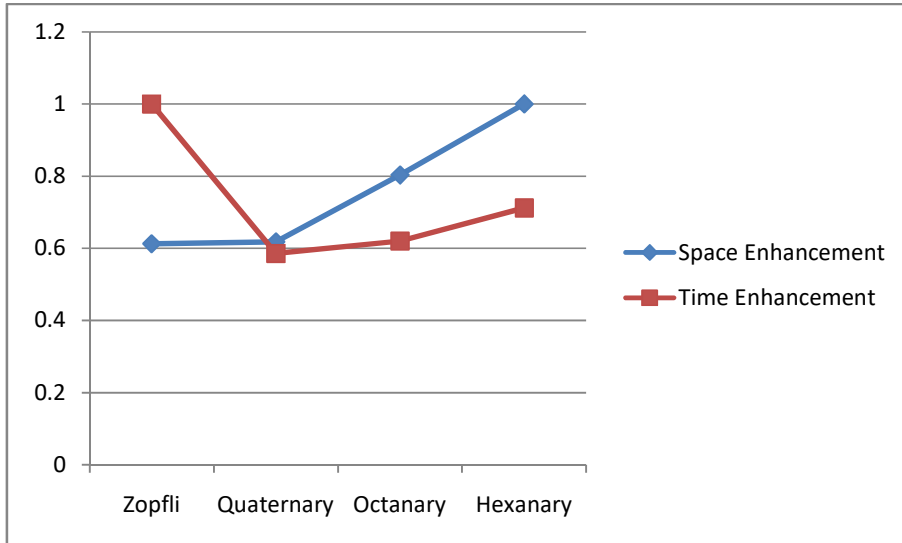


Figure 36 Time-space graph for “NLP07”

From Figure 36, it has been shown that Zopfli requires maximum time whereas Quaternary, Octanary or Hexanary requires less time. In the Quaternary technique, it achieves almost 41% speed improvement with sacrificing 1% of space. For Octanary technique, it achieves almost 38% more speed with sacrificing 24% of space. For Hexanary technique, it achieves almost 31% more speed with sacrificing 63% of space.

Table 14 Time-space data for Demo file

Before normalization			After normalization		
Method	Space (MB)	Time (S)	Method	Space (MB)	Time (S)
Binary	0.007	0.096	Binary	0.616	1
Quaternary	0.008	0.063	Quaternary	0.717	0.656
Octanary	0.010	0.075	Octanary	0.860	0.781
Hexanary	0.011	0.077	Hexanary	1	0.802

The data before and after normalization for Demo file is shown in Table 14 and the time-space graph is shown in Figure 37.

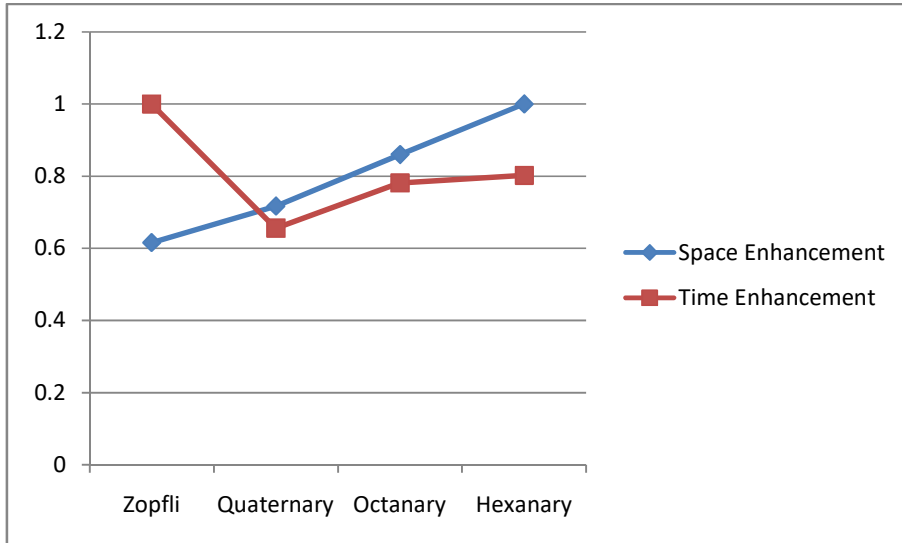


Figure 37 Time-space graph for “Demo”

From Figure 37, it has been shown that Zopfli requires maximum time whereas Quaternary, Octanary or Hexanary requires less time. In the Quaternary technique, it achieves almost 34% speed improvement with sacrificing 16% of space. For Octanary technique, it achieves almost 22% more speed with sacrificing 40% of space. For Hexanary technique, it achieves almost 20% more speed with sacrificing 62% of space.

Different Huffman based algorithms have been introduced in this section. The time-space trade-off for different Huffman based algorithms have been thoroughly discussed. Binary Huffman algorithm performs better for achieving more compression ratio. Quaternary Huffman algorithm is useful when a balance between time and space is required. However, if the tree is balanced, due to less tree-height Octanary and Hexanary Huffman algorithms perform superior to Binary and Quaternary algorithms in terms of speed, but both techniques sacrifices huge spaces. To make a balance between time and space quaternary technique is better than others algorithms which is shown in this section using various graphs.

In summary, it is revealed in the experiment that using binary Huffman code it is difficult to achieve a balance between speed and memory usage. In this research,

we focus on the use of quaternary tree instead of binary tree that speeds up decoding time with sacrificing a little space. When compared with the Huffman-based techniques, the proposed decoding algorithm exhibits excellent performance in terms of speed while the compression performance remains almost same. In this way, the proposed technique offers a way to balance between the decoding time and memory usage.

5.5 Summary

In this chapter, we have presented the experimental evaluation of the proposed text compression architecture. We evaluated the storage performance in comparison with most widely used compression techniques. The storage performance that is achieved in the proposed technique is almost parallel with the existing techniques. As we know Huffman algorithm generates an optimal tree, hence the compression will be optimized. Moreover high performance will be ensured as most repeated attribute values will get more weight and will be entered first in the dictionary i.e. domain dictionary values will be sorted in such a way that frequently occurred values will be accessed first then the rare values. The decoding speed that is achieved in the proposed technique is outperforms with many widely used existing techniques which is shown in this chapter.

We also verify the use of octanary and hexanary techniques along with binary and quaternary techniques. We observe from the experiment is that quaternary technique is the best choice to make a balance between time and space. For this reason we choose quaternary code based dictionary for our compression technique. In the experiment we also show that the proposed quaternary code based compression techniques perform well in terms of decoding speed whereas the storage space increase insignificantly.

Chapter 6

Conclusion and Future Research

A new lossless compression technique based on Huffman principle is implemented in this thesis. We introduce quaternary tree instead of binary tree in Huffman principle. We have shown that representation of Huffman code using quaternary tree is more beneficial than Huffman code using binary tree in terms of processing speed with an insignificant increase in required space. When speed is the main factor then the quaternary technique perform better than the binary technique. Thus the proposed technique provides a way to balance between the decoding time and memory usage.

5.1 Fundamental Contributions of the Thesis

- ❖ The main contribution of this research is to develop a text compression technique using quaternary code with better compression capability.
- ❖ As we use a quaternary tree to generate the dictionary, so whenever we use more symbols we will get more benefit from it. The dictionary can be used for any dictionary based text compression technique.
- ❖ The experiment shows that the decompression speed of the proposed technique is better than the many existing techniques.
- ❖ For any multilingual text, when the number of symbols is higher and the decompression speed is the main factor, the proposed technique is a better choice for text compression.
- ❖ Considerable storage reduction is achieved using the proposed architecture.

5.2 Future Research

This method is simple to program and is time efficient. The experiment shows that the decompression speed of the proposed technique is better than the many existing techniques. The future expansion of this research is to explore the following issues:

- ❖ The architecture can be used for database environment to achieve scalable performance for data warehouse application.
- ❖ We did not use any index. Techniques should be considered for indexing.
- ❖ In the research, we observe that Huffman principle does not produce any balanced tree. The codeword generation from a balanced tree could improve the compression and decompression speed together. Research on balanced quaternary Huffman tree for production of dictionary codeword could be also an important topic for future research.
- ❖ An adaptive algorithm on how to find the most suitable encoding algorithm for balancing speed and memory requirement could be an important topic for future research.

Bibliography

Adiego, J., & de la Feunte, P. (2006). On the use of words as source alphabet symbols in PPM. *Proceedings of Data Compression Conference*, (p. 435). Snowbird, UT, USA.

Alakuijala, J., & Vandevenue, L. (2013). *Data Compression Using Zopfli*. Retrieved 05 30, 2018, from https://zopfli.googlecode.com/files/Data_compression_using_Zopfli.pdf

Alakuijala, J., & Vandevenue, L. (2018). *Data Compression Using Zopfli*. Retrieved 5 30, 2019, from https://zopfli.googlecode.com/files/Data_compression_using_Zopfli.pdf

Alakuijala, J., Kliuchnikov, E., Szabadka, Z., & Vandevenue, L. (2016). *Comparison of brotli, deflate, zopfli, LZMA, LZHAM and BZip2 compression algorithms*. Internet Engineering Task Force.

Al-Bahadili, H., & Rababa, A. (2007). An adaptive bit-level text compression scheme based on the HCDC algorithm. *Proceedings of Mosharaka International Conference on Communications, Networking and Information Technology*, (pp. 51-56). Amman, Jordan.

Bell, T., & Powel, M. (2000). *The Canterbury corpus*. Retrieved 05 30, 2018, from <http://corpus.canterbury.ac.nz/resources/cantrbry.zip>

Burrows, M., & Wheeler, D. J. (1994). *A Block Sorting Data Compression Algorithm*. Digital Systems Research Center.

bzip2 1.0.6. (2010). Retrieved 05 10, 2018, from <https://github.com/enthought/bzip2-1.0.6>

Carus, A., & Mesut, A. (2010). Fast text compression using multiple static dictionaries. *Information Technology Journal*, 9 (5), 1013-1021.

- Chen, Z., Gehrke, J., & Korn, F. (2001). Query optimization in compressed database systems. [26] Chen, Z., Gehrke, J. and Korn, F., “Query optimization in compressed database systems”, 2001 ACM SIGMOD International Conference on Management of Data, ACM Press, (pp. 271–282).
- Chung, K. (1997). Efficient Huffman decoding. *Inform. Process. Lett.* , 61 (2), 97–99.
- Cormack, G. V. (1985). Data compression on a database system. *Communication of the ACM* , 28 (12), 1336–1342 .
- Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (1989). *Introduction to Algorithms* (2nd ed.). The MIT Press.
- Cutler, C. (1952). Differential Quantization for Television Signals. *U.S. Patent 2 605 361* .
- Deutsch, P. (1996). *RFC 1952 - GZIP file format specification, version 4.3* . Retrieved 05 10, 2018, from <http://www.ietf.org/rfc/rfc1952.txt>
- Dvorsky, J., Pokorny, J., & Snasel, V. (1999). Word-based compression methods for large text documents. *Proceedings of Data Compression Conference*, (p. 523). Snowbird, UT, USA.
- Faller, N. (1973). An adaptive system for data compression. *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, (pp. 593–597).
- Fano, R. (1949). *The transmission of information*. Research Laboratory for Electronics, MIT Technical Report.
- Fenwick, P. (1995). Huffman Code Efficiencies for Extensions of Sources. *IEEE Transactions on Communications* , 43 (2/3/4), 163-165.
- Gallager, R. (1978). Variations on a theme by Huffman. *IEEE Transaction on Information Theory* , 24 (6), 668–674.

- Gibson, J. D., Berger, T., Lookabaugh, T., Lindbergh, D., & Baker, R. (1998). *Digital Compression for Multimedia: Principles and Standards*. San Mateo, CA: Morgan Kaufmann.
- Golomb, S. W. (1966). Run-length encodings. *IEEE Transaction on Information Theory* , 12 (3), 399-401.
- Graefe, G., & Shapiro, L. (1991). Data compression and database performance. *ACM/IEEE-CS Symposium on Applied Computing*, (pp. 22-27).
- Habib, A., & Rahman, M. (2017). Balancing decoding speed and memory usage for Huffman codes using quaternary tree. *Applied Informatics* , 4 (5), 1-15.
- Held, G., & Marshel, T. R. (1996). *Data and Image Compression*. John Wiley and Sons Ltd, West Sussex, England.
- Helmer, S., Westmann, T., Kossmann, D., & Moerkotte, G. (2000). The implementation and performance of compressed databases. *SIGMOD Record* , 29 (3), 55–67.
- Huang, J.-Y., & Schultheiss, P. M. (1963). Block quantization of correlated gaussian random variables. *IEEE Trans. Communication Systems* , CS-11, 289–296.
- Huffman code*. (2005). Retrieved from Sandhills Publishing Company: <http://www.smartcomputing.com/editorial/dictionary/detail.asp?searchtype=1&DicID=17660&RefType=Encyclopedia>
- Huffman, D. (1952). A method for construction of minimum redundancy codes. *Proc. IRE*, (pp. 1090-1101).
- Katona, G., & Nemetz, T. (1978). Huffman codes and self-information. *IEEE Trans. Inform. Theory* , 22 (3), 337-340.
- Kavousianos, X., Kalligeros, E., & Nikolos, D. (2008). Test-Data Compression Based on Variable-to-Variable Huffman Encoding with Codeword Reusability.

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems , 27 (7), 1333-1338.

Khuri, S., & Hsu, H.-C. (2000). Tools for visualizing text compression algorithms. *Proceedings of SAC'00*, (pp. 119-123). Italy.

L'ansk'y, J., & Zemlička, M. (2006). Compression of a dictionary. *Proceedings of DATESO Workshop on Databases, Texts, Specifications and Objects*, (pp. 11-20). Desna, Czech Republic.

L'ansk'y, J., & Zemlička, M. (2005). Text compression : Syllables. *Proceedings of the Dateso - Workshop on Databases, Texts, Specifications and Objects*, (pp. 32-45). Desna, Czech Republic.

Larson, N. J., & Moffat, A. (2000). Off-line dictionary-based compression. [24] *Larson, N. J. and Moffat, A., "Off-line dictionary-based compression. Proceedings of the IEEE Data Compression Conference. Snowbird, Utah.*

Lin, Y.-K., Huang, S.-C., & Yang, C.-H. (2012). A fast algorithm for Huffman decoding based on a recursion Huffman tree. *The Journal of System and Software* , 85, 974-980.

Lipschutz, S. (2011). *Data Structures With C*. Tata McGraw-Hill Education.

Luke 5. (2019). Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Luke_5

LZHAM Source. (2018). Retrieved 05 10, 2018, from
https://github.com/richgel999/lzham_codec

LZMA Source. (2018). Retrieved 05 10, 2018, from LZMA implementation in 7zip 9.20.1: <https://www.7-zip.org/sdk.html>

Mahoney, M. (2018). *The Enwik8 corpus*. Retrieved 05 30, 2018, from
<http://mattmahoney.net/dc/enwik8.zip>

- McGregor, D., Cockshott, W. P., & Wilson, J. (1998). McGregHigh-performance operations using a compressed architecture. *The Computer Journal* , 41 (5), 283– 296.
- Memon, N. D., & Sayood, K. (1995). Lossless Image Compression: A Comparitive Study. *Proceedings SPIE Conference on Electronic Imaging*. SPIE.
- Mitchell, J. L., Pennebaker, W. B., Fogg, C. E., & LeGall, D. J. (1997). *MPEG Video Compression Standard*. London/New York: Chapman & Hall.
- Moffat, A., & Isal, R. (2005). Word-based text compression using the Burrows-wheeler transform. *Information Processing Management* , 41 (5), 1175-1192.
- Moffat, A., & Zobel, J. (1992). Parameterised compression for sparse bitmap. [28] Moffat, A. and Zobel, J., “Parameterised compressiProceedings of the 15 Annual International SIGIR, (pp. 274-285).
- Mumin, M., Shoeb, A., Selim, M., & Iqbal, M. (2012). SUPara : A Balanced English-Bengali Parallel Corpus. *SUST Journal of Science and Technology* , 16 (2), 46-51.
- Nelson, M. (1996). Data compression with the Burrows-Wheeler transform. *Dr. Dobbs Journal* .
- Nelson, M., & Gailly, J.-L. (1996). *The Data Compression Book*. California: M&T Books.
- Oberhummer, M. F. (2002). *LZO: A real-time data compression library*. Retrieved from <http://www.oberhumer.com/opensource/lzo/lzodoc.php>.
- OCAML. (2014). *Height, Depth and Level of a Tree*. Retrieved 07 09, 2019, from <http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree/>
- Pasco, R. (1976). *Source Coding Algorithms for Fast Data Compression*. Ph.D. thesis, Stanford University.

- Rissanen, J. J. (1976). Generalized Kraft inequality and arithmetic coding. *IBM J. Research and Development* , 20, 198–203.
- Roth, M. A., & Van Horn, S. J. (1993). Database compression. *SIGMOD Record* , 22 (3), 31–39.
- Sayood, K. (2000). *Introduction to Data Compression*. San Mateo, CA: Morgan Kaufman/Academic Press.
- Schack, R. (1994). The length of a typical Huffman codeword. *IEEE Trans. Inform. Theory* , 40 (4), 1246-1247.
- Shannon, C. (1948). A mathematical theory of communication. *Bell System Technical J.* , 27, 379–423,623–656.
- Skibiński, P. (2006). *Reversible data transforms that improve effectiveness of universal lossless data compression*. Wrocław: Doctor of Philosophy Dissertation, University of Wrocław.
- Storer, J., & Szymanski, T. (1982). Data compression via textual substitution. *Journal of the ACM* , 29 (4), 928-951.
- The Brown Corpus*. (2018). Retrieved 07 09, 2018, from <http://www.nltk.org/nltk data/>
- The Canterbury Corpus*. (2018). Retrieved 07 10, 2018, from <http://corpus.canterbury.ac.nz/resources/cantrbry.zip>.
- The Enwik8 Corpus*. (2018). Retrieved 07 10, 2019, from <http://mattmahoney.net/dc/enwik8.zip>
- Vitter, J. (1987). Design and analysis of dynamic Huffman code. *Journal of the ACM* , 34 (4), 825–845.
- Wallace, G. K. (1991). The JPEG still picture compression standard. *Communications of the ACM* , 34 (4), 30–44.

Weinberger, M., Seroussi, G., & Sapiro, G. (1998). *The LOCO-I Lossless Compression Algorithm: Principles and Standardization into JPEG-LS*. Hewlett-Packard Laboratory.

Welch, T. (1984). A technique for high-performance data compression. *Computer* , 17 (6), 8-19.

Wu, X., & Memon, N. D. (1996). CALIC—A context based adaptive lossless image coding scheme. *IEEE Trans. Communications* .

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE transaction Information Theory* , 23 (3), 337-343.

Ziv, J., & Lempel, A. (1978). Compression of individual sequence via variable-rate coding. *IEEE transaction Information Theory* , 24 (5), 530-536.

Zopfli Source Code. (2018). Retrieved 07 10, 2018, from <https://github.com/google/zopfli/commit/89cf773beef75d7f4d6d378debd299378c3314e>

Publications

1. Ahsan Habib, M. Jahirul Islam and M. Shahidur Rahman, “A Dictionary Based Text Compression Algorithm Using Quaternary Code,” Iran Journal of Computer Science, Springer, 2019. (Accepted)
2. Ahsan Habib, M. Jahirul Islam and M. Shahidur Rahman, “Huffman Based Code Generation Algorithms: Data Compression Perspectives,” Journal of Computer Science, 14 (12): 1599-1610. DOI: 10.3844/jcssp.2018.1599.1610, 2018.
3. Ahsan Habib, Muhammad Shahidur Rahman, “Balancing Decoding Speed and Memory Usage for Huffman Codes Using Quaternary Tree,” Applied Informatics, Springer, 4(5):1-15, DOI: 10.1186/s40535-016-0032-z, 2017.
4. Ahsan Habib, M I H Palash, A Ahmed Chowdhury, Kawser Ahmed, M Jahirul Islam, M Shahidur Rahman, “A Dynamic Huffman Based Short Message Service (SMS) Compression Technique,” NEUB Journal, Vol. 2, No. 1, pp 33-42, 2017.
5. Md. Mamun Hossain, Ahsan Habib, Muhammad Shahidur Rahman, “Transliteration based Bengali Text Compression Using Huffman Principle”, Proceedings of the International Conference on Informatics, Electronics & Vision (ICIEV), Bangladesh , ISBN 978-1-4799-5179-6; IEEE, May 2014.