R

DATA COMPRESSION TECHNIQUES
FOR
BANGLA TEXT

BY
S. M. HUMAYUN

A THESIS SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE IN ENGINEERING
(COMPUTER SCIENCE AND ENGINEERING)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DHAKA, BANGLADESH

AUGUST 1994

CERTIFICATE

This is to certify that this thesis work has been done by me under the guidance of Dr. M. Kaykobad and it has not been submitted elsewhere for the award of any degree or diploma.

Countersigned

(DR. M. KAYKOBAD)
Supervisor

Signature of the Candidate

(S. M. HUMAYUN)
Roll no:871802
BUET, Bangladesh.

DATA COMPRESSION TECHNIQUES FOR BANGLA TEXT

BY
S. M. Humayun

Accepted as satisfactory for partial fulfillment of the requirements for the degree of Master of Science in Engineering (Computer Science and Engineering), Bangladesh University of Engineering & Technology, Dhaka.

EXAMINERS

4/8/94

1. Dr. M. Kaykobad
Assistant Professor,
Department of Computer
Science & Engineering,
BUET, Dhaka.

Chairman and
Supervisor

৪/৮/৯৪

2. Prof. A.B.M. Siddique Hossain
Head, Dept. of CSE, BUET, Dhaka

Member

M. S. Alam 4/8/94

3. Prof. Md. Shamsul Alam
Dept. of CSE, BUET, Dhaka

Member

4. Prof. Shamsuddin Ahmed
Head, Dept. of EEE, ICTVTR

Member

ABSTRACT

In recent years Bangla has been being used in computers. For efficient use of this language in computers it is very important to be able to store texts economically so that in terms of both storage requirement and transmission cost it is competitive. In this study efforts have been made to obtain economical coding of Bangla texts using static and dynamic Huffman codes, arithmetic codes and other important coding techniques. Performances of various coding techniques in coding Bangla texts of different types and formats have been considered in terms of compression efficiency, coding and decoding times.

Our result shows that arithmetic coding with scaling symbol counts has outperformed all the remaining coding techniques for off-line coding on general texts in BSCII format in terms of coding efficiency. Compression efficiency for this algorithm varies between 24.80% - for 1kb file and 34.92% - for 200kb file. Although Vitter algorithm is the slowest in terms of coding and decoding times, it has been found best in terms of coding efficiency among all on-line coding algorithms having efficiency 28.40% for 1kb file and 34.84% for 200kb file of general BSCII format texts.

There is a significant variation of efficiency and coding and decoding times with respect to text formats. Non document BSCII format texts have been found to be the most efficient and fastest whereas document BNA format texts are the slowest and most inefficient.

Static Huffman coding techniques have been found faster in terms of coding and decoding times requiring roughly 16% time less than the arithmetic coding. Among the dynamic coding algorithms Vitter algorithm, being the slowest, takes roughly 28% time more than the fastest static Huffman algorithm.

ACKNOWLEDGEMENT

It is a matter of great pleasure for the author to acknowledge his profound gratitude to his supervisor, Dr. M. Kaykobad, Assistant Professor, Department of Computer Science and Engineering, BUET, for his advice, valuable guidance and constant encouragement throughout the progress of this work.

The author is indebted to Dr. A.B.M. Siddique Hossain, Professor and Head, Department of Computer Science and Engineering, BUET, for his inspiration to complete the work. He is grateful to Prof. Md. Shamsul Alam of the department of Computer Science and Engineering, BUET and Prof. Shamsuddin Ahmed of ICTVTR for serving as members of the board of examiners.

He wishes to express his thanks and deep sense of gratitude to Dr. Jeffrey Scott Vitter, Associate Professor, Department of Computer Science, Brown University, USA, for his kind help through providing papers and publications related to this work.

Thanks are due to Dr. S.P. Majumder, Asstt. Prof., Electrical & Electronic Engineering Department, BUET, Dr. N.C. Das, and Dr. A.K.M. Mohiuddin, Assistant Professor, BIT Dhaka, R. A. A. Abdullah and M.Shahidul Islam of the SAfeworks, Dhaka, Engr. Sahjahan Sikder, Computer Engineer, NERP, Gulshan, and Engr. S. M. Shah-Newaz, System Manager, Surface Water Modelling Center, Gulshan for their help during this work. He also thanks all others who helped the author directly or indirectly during this work.

CONTENTS

Pages

CHAPTER THREE: STATISTICAL LOSSLESS COMPRESSION TECHNIQUES

CHAPTER FOUR: BANGLA TEXT ANALYSIS

CHAPTER FIVE: IMPLEMENTATION OF ALGORITHMS

CHAPTER SIX: DESIGN OF EXPERIMENTS AND RESULTS

CHAPTER SEVEN: DISCUSSIONS AND RECOMMENDATIONS

LIST OF SYMBOLS AND ABBREVIATIONS

Symbol/ Abbreviation	Meaning
BSCII	- Bangla Standard Code for Information Interchange
FGK	- Fallar, Gallager and Knuth algorithm
STD	- Standard (BSCII) format non-document general text
QSTD	- Standard (BSCII) format non-document specific text
XFR	- Non-document format (Barna word processor) general text
QXFR	- Non-document format (Barna word processor) specific text
BNA	- Document format (Barna word processor) general text
QBNA	- Document format (Barna word processor) specific text

LIST OF GRAPHS

LIST OF TABLES

Number Title Pages

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

Information theory is usually thought of as science of sending information from here to there, i.e., transmission of information, but this is exactly the same as sending information from now to then, i.e., storing information. Both situations occur constantly when handling Bangla text as source of information. Processing Bangla text with computers is the same as representing, transmitting and transforming the text. The processing of Bangla text using computers and other modern information processing equipment grows explosively in different fields of applications, especially in the Desk Top Publication (DTP). Concurrent with this growth several problem areas have developed which can result in major but unnecessary economic expenditures. One of these is the capacity of disk storage. The method that can be employed to alleviate a portion of Bangla text storage and transfer problems is through the representation of the text by more efficient codes.

The field of text compression requires a variety of back grounds and theoretical analysis for its successful application. This field has grown with the field of information theory and coding. The study of text compression has grown considerably from its beginning in the work of Shannon[65] to the today's large scale research programs continuing in many centers and universities. The holding of special conferences on data compression shows that the demand of this field is tremendous.

1.1 Importance of Bangla Text Compression

If our language is to keep pace with the development of allied technologies it is very important to be able to use these technologies to its development. It can be noted with great pleasure that researchers of our country, specially those related to computing, understood the importance of computerizing Bangla and started the ground work in the early 80's. Khan[43] has identified 434 characters in the Bangla alphabet among which 302 are compound Byanjana Varnas. He has presented 222 compound byanjana varnas with 2 byanjana varnas, 74 with 3 byanjana varnas and 6 with 4 byanjana varnas. A schedule of Bengali characters and mechanism of generation of the Bangla Graphic Symbol (BGS) set has been suggested, most of them are represented by 2 or 3 bytes in computer processing. Bangla word processors and DTP software are also using multibyte representation of these compound characters. So there should be high redundancy in Bangla text.

News papers and publication industries using Bangla DTP need a huge disk storage to store their information. Transmission of Bangla news between different cities would need excessive transmission time. To use Bangla text economically in storage and transmission, Bangla characters should be coded efficiently. Efficient coding reduces redundancy if there is any. As we expect higher redundancy in Bangla text, efficient coding should reduce the storage requirement significantly.

1.2 Objectives of the Research

The objective of the present research is to obtain a set of efficient standard variable length codes for Bangla text by studying the characteristics of Bangla texts and its alphabet. This study is also to suggest efficient real time data compression techniques for Bangla text to facilitate economic storing and transmission of Bangla text. In this study various compression algorithms will be tested by sample Bangla texts of different types to find out their efficiency in data compression.

1.3 Literature Survey

The problems associated with the data compression cover so wide a range that they must be classified in some way before any approach can be made. One simplified approach of classification of data compression techniques has been given by Held[29]. He discussed the different types of compression methods and their benefits. Bookstein and Storer[6] have given another classification reflecting the current state of data compression research. Compression depends largely upon the efficient representation of the source alphabet. A number of codes have been discussed by Shannon[65] and Hamming[26]. Aronson[4] gives the null suppression algorithms, a special type of block coding technique used in IBM 3780 BISYNC transmission protocol. Rubin[62], Ruth and Kreutzer[63] present block code compression method that is a general work of Aronson.

Data can be compressed using variable length compression techniques whenever some data symbols are more likely to appear than others. Shannon showed that for the best possible compression code in the sense of minimum average code length, the output length contains a contribution of $-\lg(p)$ bits from the encoding of each symbol whose probability of occurrence is p. The term redundancy has been defined by Shannon as a property of the code. Huffman[35] first introduced a minimum redundancy method of source coding called "Huffman code". Norwood[56] proposed a recursive formula to count the number of different compact codes. Even and Lempel[14] also presented a similar recursive formula introducing a new concept of "proper word". Connel[10] derived a Huffman-Shannon-Fano (HSF) code by adopting a notion of Shannon-Fano code[16,65] and combining it with the Huffman code, where in the code symbols appear lexicographically. The HSF code is unique if symbol probabilities are specified, and it is compact code in the sense of Norwood and Even and Lempel.

Recently, Itai[37], Glassey and Karp[23] and Golumbic[24] have presented new perspectives on how the algorithm works and how it can be employed in new ways. Until then, all research has concentrated on two variations of the algorithm, which respectively minimize:

(i) the weighted path length, and

(ii) measures akin to tree height,

of the constructed tree.

Modern applications for weighted path length minimization include:

(1) construction of optimal search trees[24,34],

(2) merging of lists[49],

(3) minimization of absolute error bounds in the sum of positive numbers and relative error bounds in products[76],

(4) text file compression[62], and

(5) optimal checking for leaky pipelines and water pollution[23].

Applications of tree height minimization include the determination of the minimum execution time for fanning-in data and problems related to speed in parallel processing[24].

Parker[57] characterized a wide class of weight combination functions, the quasilinear functions, for which the Huffman algorithm produces an optimum tree under correspondingly wide classes of cost criteria. Application of information divergence to Huffman codes is given by Longo and Galasso[51]. D-ary Huffman codes and their optimality are given by Capocelli and Santis[8] and Cover[11]. The idea of local redundancy and lower bounds on the redundancy of Huffman codes is given by Yeung[84]. Geckinli[22] gives two corollaries for D-ary Huffman codes with condition for optimality of a block code. Ferguson and Rabinowitz[18] add synchronizing property to Huffman codes.

Huffman algorithms require two passes over the text. For file compression the extra disk accesses slow down the operation. Faller[15] and Gallager[20] independently proposed a one-pass scheme, that has been improved substantially by Knuth[44] for dynamic Huffman codes, usually known as FGK codes. Vitter[78-79] has analyzed the one-pas algorithm due to Faller, Gallager and Knuth and proposed a new algorithm. He also derived tight upper and lower bounds for the dynamic Huffman codes.

Langdon[47] introduces arithmetic coding techniques and Howard and Vitter[30-32] has given a tutorial and analyzed these coding techniques. Apiki[3] and Nelson[52-53] have discussed and implemented the arithmetic coding technique.

A general algorithm for the minimum-redundancy encoding of a discrete information source is proposed by Gauzzo[25]. The problems associated with modeling of compression techniques and their complexity are discussed by Rissanen and Langdon[60]. Window based coding techniques are given by Ziv and Lempel[86-87], Wetch[81] and Nelson[43]. Comparative studies on the commercial data compression tools are given by Byrd[7], Simon[68], Nichols[54-55].

Static Huffman codes for the Bangla alphabet have been presented by Humayun, Rahman and Kaykobad[36] on the basis of character frequencies and variable length codes have been suggested by Haque[27] on the basis of Bangla character sound statistics. Data compression techniques on digital Chinese character patterns have been studied by Ju, Jou and Tsay[40].

1.4 Organization of the Thesis

Chapter One introduces the area of current research work, and states the importance and objective of the work. A discussion on works related to the current one has also been presented. Essence of data compression has been given in chapter Two. In this chapter, compression systems, classification of compression techniques and variable length codes have been discussed in depth. Finally ideas of information contents, entropy and redundancy of text have also been given. In chapter Three, statistical lossless data compression techniques have been presented. Shannon-Fano coding, Static Huffman coding, Dynamic Huffman coding by Fallar, Gallager and Knuth, Optimal Dynamic Huffman coding by Vitter and Arithmetic coding techniques has been discussed in this chapter. Bangla text analysis has been given in chapter Four. Idea of n-gram statistics for Bangla text has also been given in this chapter. In chapter Five, implementation of compression algorithms has been discussed. Chapter Six has been devoted to the design of experiments and presentation of results and in chapter Seven over all discussion of results and recommendation of future work have been given.

CHAPTER TWO

ESSENCE OF DATA COMPRESSION

Data encoding is a process of mapping the representation of data from one group of symbols called source symbols to another, a more precise series of symbols called code symbols. The relationship between the source symbols and their corresponding code symbols is called a codeword. If the encoding is one-to-one then an inverse mapping exists and decoding refers to the reversing process. The two primary functions of data compression are as follows:

Storage: The storage capacity of mass storage device can be effectively increased with data compression. Crucial to many applications is the hardware or on-the-fly compressing software that can in real time intercept and compress the data on its way to the storage device and decompress it as it is needed. Data compression/decompression system with storage is shown in the Fig.2.1.

Communications: The bandwidth of a digital communication link can be effectively increased by compressing the data at the sending end and decompressing the data at the receiving end.

Here it is crucial that compression/decompression can be performed in real time. Fig.2.2 shows the computer communication link with data compression.

2.1 Compression Systems

Any data compression approach has a model that makes some assumptions about the data and events encoded. The decision to output a certain codeword for a certain source symbol or a set of symbols is based on the model. The encoder itself can be independent of the model. The model is simply a collection of source symbols and rules used to process source symbols and determine which codeword(s) to output. A program uses the model to define the probabilities for each symbol accurately and the encoder to produce an appropriate codeword based on those probabilities.

Modeling and encoding are two distinctly different things. Several different methods can be used to model the source string, all of which can use the same encoding process to produce their codewords. A simple statistical model used with Huffman coding is shown in Fig.2.3. The encoder using Huffman scheme would use the model that gave the raw probability of each symbol occurring anywhere in the input stream. A more sophisticated model might calculate the probability based on

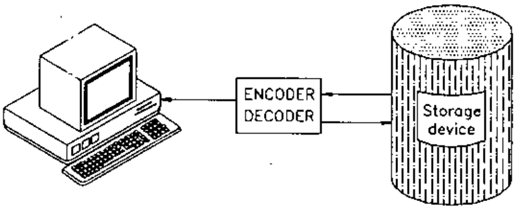
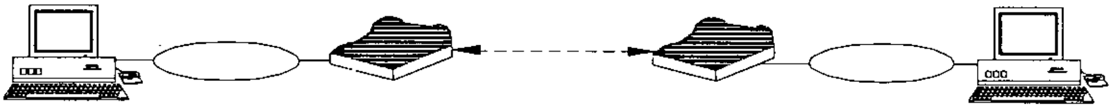


Fig. 2.1: Data compression/decression with strorage device



2.2: Digital communication link with dato compression.

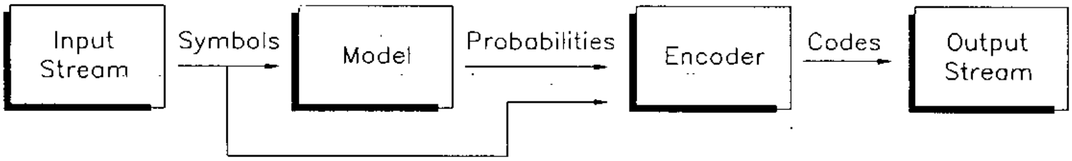


Fig. 2.3 A Statistical Model with a Huffman Encoder

the last 10 symbols in the input stream. Although both the models may use Huffman coding to produce their codewords, their compression ratios would probably be radically different.

A simple model is the memoryless model, where the source symbols themselves are encoded according to a single code. Another model is the first-order Markov model[47,82], which uses the previous symbol as the context for the current symbol. If the source message is a sentence in English language, and source symbol 'q' is the previous symbol, the model would expect the next symbol to be 'u'. The first-order Markov model is a dependent model in which there is a different expectation for each source symbol depending on the context. The context is a state governed by the past sequence of symbols. The purpose of a context is to provide a probability distribution for encoding (decoding) the next source symbol.

2.1.1 Components of a Compression System

The components of a compression system are

(a) the model structure,

(b) the statistical unit, and

(c) the encoder.

Model Structure: In practice, the model is a finite state machine that operates successively on each source symbol and determines the current event to be encoded and its context if it is a first-order Markov model. Often, each event is the source symbol itself, but the structure can define other events from which the source string could be reconstructed. For example, one could define an event such as the run length of succession of repeated symbols, i.e., the number of times the current symbol repeats itself.

Statistical Unit: This unit computes the relative frequency distribution used for each context. The computation may be performed beforehand, or may be performed during the encoding process, typically by a counting technique. For Huffman codes, the event statistics are predetermined by the length of the event's codeword.

Encoder: This unit accepts the events to be encoded and generates the code string.

The notions of the model structure and statistics are important because they completely determine the compression efficiency. In some complex systems, the compression problem is equivalent to the modeling problem.

2.2 Classification of Coding Methods

Based on the consideration of encoding error, data compression techniques can be divided into two major families.

(a) Lossy Compression, and

(b) Lossless Compression

2.2.1 Lossy Compression

Lossy data compression concedes a certain loss of accuracy in exchange for greatly increased compression. These processes are typically used for applications where there is a notion of fidelity associated with the data. Such applications often involve digitally sampled analog data (e.g., speech, still image, video, etc.) where it is only necessary that the decompressed data be acceptably close in quality to the original. By their very nature, these digitized representations of analog phenomena are not perfect to begin with, so the idea of output and input not matching exactly is a little more acceptable. Most lossy compression techniques can be adjusted to different quality levels, gaining higher accuracy in exchange for less effective compression. Until recently, lossy compression has been primarily implemented using dedicated hardware.

Lossy compression methods are primarily classified as:

(a) Scaler and vector quantization,

(b) Transform methods

(c) Fractal decomposition, and

(d) Temporal compensations.

Scaler and vector quantization: Vector quantization is the process of partitioning a body of data into disjoint ordered sets and replacing each vector by an index to a closest matching vector in a dictionary of vectors. For example, with image compression, vectors are typically subarrays of pixels in the range 2x2 to 8x8 and dictionaries (often called tables or codebooks) typically have sizes ranging from 256 to 64000 vectors. As another example, in the character recognition from a half-tone or fax data where vectors are arrays of bits that are positioned over the character positions and the codebook is the alphabet of characters that is being recognized. Larger vectors and dictionaries yield higher fidelity for a given amount of compression, but require greater computational resources. Scaler quantization is the special case where each vector consists of a single data element.

Transform Methods: A typical transform method takes a block of n input values and computes a new set of n values by applying a transform that has the effect of concentrating the important information non-uniformly in the new values. The new values are then scaler quantized, with less important values being either more coarsely quantized or discarded entirely. One of the most widely used transforms, and the basis of a number of data compression standards such as Joint Photographic Expert Group (JPEG) Compression, is the discrete cosine transform[2], but many others such as the Fourier, Walsh-Hadamard, Haar, Hartley, and wavelet transform have also been considered in the literature[28].

Fractal Decomposition: Fractals are recursively defined curves that can be specified by integer parameters. Data may be compressed by approximating it by a set of fractal curves. The approximation can range from lossless to very lossy. In fact, one of the most successful applications of fractals to date has been very high compression of images that can amount to replacing the image by a drawing of it.

Temporal Compensations: Digitized video is usually much more compressible than single images because there is typically a great similarity between successive frames. Hence, in addition to lossy compression of individual frames and lossless compression of the resulting data stream, displacement estimation algorithms that track groups of pixels that remain identical or acceptably close from one frame to the next can be crucial to achieve high degrees of compression. Other forms of temporal compensations include pan and zoom compensation, frame alignment and blending, and reversible dynamic range compression. Proposed standards for video compression are discussed in LeGall[48]; also, Sijstermans and van der Meer[67] discuss full motion video encoding.

Detail classifications and implementation of lossy compression techniques is done by Chowdhury[9].

2.2.2 Lossless Compression

Lossless data-compression techniques preserve all the information in the data so that it can be reconstructed without error. It consists of those techniques that guarantee to generate an exact duplicate of the input data stream after a compression/expansion cycle. This type of compression is typically used for applications where the loss of a single bit can change the meaning of the data. This type of compression is mandatory for transmission or storing computer programs, documents, numerical information, database records, spreadsheets etc. In these applications, the loss of even a single bit could be catastrophic.

On the basis of the codeword length, Lossless compression scheme can be classified as :

(a) Block or Fixed Length Coding, and

(b) Variable Length Coding.

Block Coding: In these schemes source messages are analyzed and a group of source symbols are replaced by a fixed length codeword or block of code symbols. Null Suppression, Bit Mapping and Run Length coding are common block coding techniques.

Variable Length Code: In these schemes the length of codewords for different source symbols are different depending on the statistics or any other property that comes from the model used by the coding scheme. Huffman code is an example of a variable length code.

On the basis of the mode of operation the coding schemes can be classified as :

(a) Substitution/Dictionary Based Encoding, and

(b) Statistical/Entropy Encoding.

Depending on the model, both statistical and dictionary based encodings can be (a) Static or (b) Dynamic/Adaptive.

Dictionary Based Encoding: Statistical schemes generally encode a single symbol at a time, reading it in, calculating a probability, then outputting a single codeword. Dictionary based scheme maintains a table of matching string (group of source symbols) called dictionary. The body of data is compressed by replacing substrings of the input message by the corresponding indices of these substrings in the dictionary; the indices are called pointers. Thus, the input is a stream of source symbols and the output is a stream of pointers, where most pointers specify strings of length greater than one. Similarly, the input to the decoder is a stream of pointers and output is a stream of source symbols. Various heuristic can be employed to recognize new strings to be added to the dictionary and to discard strings from the dictionary when more space is needed; such modifications allow the dictionary to adapt to changing characteristics of input message. A nice aspect of textual substitution methods is that commonly occurring substrings are more likely to be grown to longer strings in the dictionary, and the distribution of pointers is not needed, making textual substitution methods very practical to implement.

Textual substitution methods have been implemented by the following algorithms:

(a) LZ77 algorithms,
(b) LZ78 algorithms, and
(c) LZW algorithms.

All of the above methods are based on the important work of Lempel and Ziv[86-87] and Welch[81], who made substantial modifications. Reif and Storer[59] have developed massively parallel hardware for high speed textual substitution.

Statistical Compression Techniques: Statistical encoding schemes take advantage of the probabilities of occurrence of single source symbols and a group of symbols. In these schemes average codeword length relates to the probabilities of the source symbols. Statistical compression techniques are of the following types:

(a) Shannon-Fano Coding,
(b) Huffman Coding,
    Static Huffman Coding
    Dynamic Huffman Coding
        FGK algorithm,

Knuth algorithm,

Optimum One pass (Vitter) algorithm, and

(c) Arithmetic Coding

Static Arithmetic Coding

Adaptive Arithmetic Coding

All statistical encodings produce optimal variable length codewords.

2.3 Variable Length Codes

The codes in which the source symbols are encoded in different length (no. of digits) of code symbols are considered as variable length codes. These codes are becoming increasingly important as the costs of communication in distributed systems and external storage are beginning to dominate the costs for internal memory and computation. The advantage of a code in which the encoded source symbols are of variable length is that the code is more efficient in the sense that fewer digits (bits in the binary system) for representation of the same piece of information are required on the average, than do fixed length codes that require $\lceil \log(\mathbf{n}) \rceil$ bits per source symbols, where **n** is the source alphabet size. This can yield tremendous savings in communication system and file compression. Moreover, the buffering needed to support variable length coding is becoming inherent part of many systems. To accomplish this, the encoders need to know something about the statistics of the messages being encoded. If every symbol is as likely as every other one, then the block codes are about as efficient as any code can be. But if some symbols are more probable than others, then encoder can take advantage of this to make the most frequent symbols correspond to the shorter encodings, and the less frequent symbols correspond to the longer encodings, so that the message would take up less space. If the probabilities of the frequencies of occurrence of the individual symbols are sufficiently different, then variable-length encoding can be significantly more efficient than fixed length encoding.

However, variable-length codes bring with them a fundamental problem of identifying the codewords corresponding to the source symbols from the encoded stream of codewords. The decoder has to identify the beginning and end of the codeword.

To solve these problems, variable length codes must have the following important properties:

- Different codewords have different numbers of code symbols (bits).

- Codewords for source symbols with low probabilities have more bits, and codewords for source symbols with high probabilities have fewer bits.
- Though the codewords are of different code lengths, they can be uniquely decoded.
- The decoding should be instantaneous.

2.3.1 Unique Decoding

In general, the source alphabet has **q** symbols, $S = \{s_i \mid i = 1, 2, \dots, q\}$, and that the code's alphabet has **r** symbols, **r** for the radix of the system.

In variable-length coding, the codeword must be uniquely decodable, i.e., the encoded message must have a single, unique possible interpretation. Consider a code in which the source alphabet S has four symbols, and they are to be encoded in binary as follows:

$s_1 = 0$
$s_2 = 01$
$s_3 = 11$
$s_4 = 00$

The particular coded message 0011 could be decoded as one of these two source messages:

$$0011 = \{s_4, s_3\} \text{ or } \{s_1, s_1, s_3\}$$

Thus the code is not uniquely decodable. Unique decodability is usually highly desirable.

For unique decodability, no two codewords can be the same. Clearly, only if every distinct sequence of source symbols has a corresponding unique codeword sequence, then the code message is uniquely decodable. This is a necessary and sufficient condition.

2.3.2 Instantaneous Codes

Considering a source alphabet with four symbols, $S = \{s_i \mid i = 1, 2, \ldots, 4\}$, and coded in binary as in the following :

$s_1 = 0$
$s_2 = 10$
$s_3 = 110$
$s_4 = 111$

Now the coded message **0011011110** is to be decoded. Clearly the decoder would decode the message from left to right as the message appears to it. The decoder will emit source symbols $s_1 s_1 s_3 s_4 s_2$. To decode this message the decoder would set up a finite automaton, i.e., a decision tree shown in Fig.2.4. Starting in the initial state the first binary digit received will cause a branch, either to a terminal state s_1 if the digit is 0, or else to a second decision point if it is a 1. For the next binary digit this second branch would go to the terminal state s_2 if a 0 is received, and to a third decision point if it is a 1. The third would go to the terminal state s_3 if the third digit is a 0, and to the terminal state s_4 if it is a 1.

Each terminal state would, of course, emit its source symbol and then return control to the initial state. It is clear that each bit, i.e., code symbol of the received stream is examined only once, and that the terminal states of this tree are the four source symbols s_1, s_2, s_3 and s_4.

In this example, the decoding is instantaneous since when a complete codeword is received, the decoder immediately knows this, and does not have to look further before deciding what message symbol it received. No codeword of this code message is a prefix of any other codeword.

This shows the basic equivalence of the existence of the decoding tree and the instantaneous decodability; each implies the other. It is also clear that using the decoding tree means that each received codeword is looked at only once in the decoding process.

Now consider that the same source alphabet has been encoded in binary as in the following:

$s_1 = 0$
$s_2 = 01$
$s_3 = 011$
$s_4 = 111$

It is the previous codeword with the bits reversed. Some of these codewords are prefixes of other codewords; that is, they are the same as the beginning part of some other symbol.

Now consider the same coded message 0011011110 is to be decoded into its source symbols. If the decoder goes as the message received, it cannot decide whether it received the source symbol that corresponds to the prefix. In this example, the first code symbol 0 of this message corresponds to the source symbol s_1 and prefix of the source symbols s_2 and s_3. It can only be decoded by first going to the end and then identifying the symbols backward. This codeword is uniquely decodable but is not instantaneous because the decoder does not know when one codeword is over without looking further. The simplest way to decode messages in this particular code is always to start at the back end of the received message. This puts a severe burden on the storage and also causes a time delay.

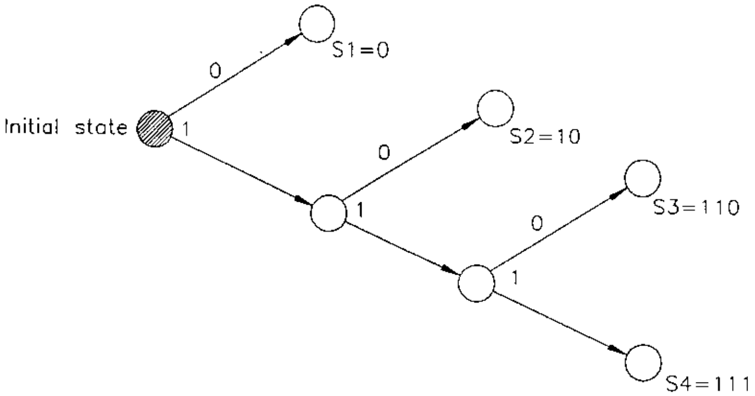


Fig. 2.4 : Decoding tree for 4 symbols with codeword set {0, 10, 110, 111}.

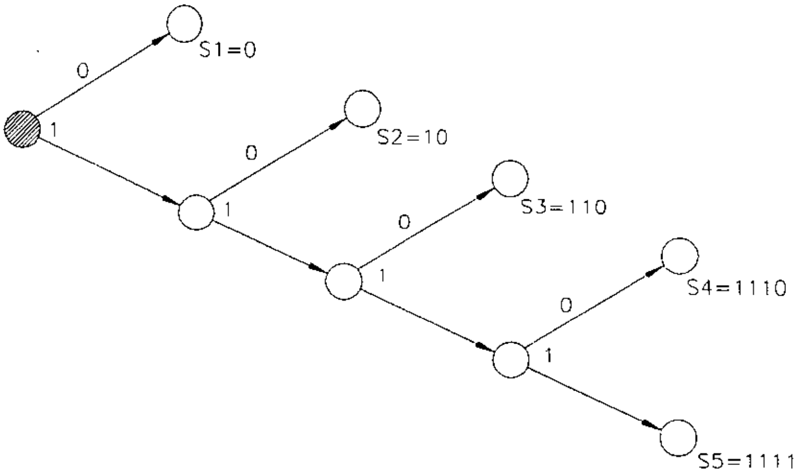


Fig. 2.5 : Decoding tree for 5 symbols with codeword set {0, 10, 110, 1110, 1111}.

In view of the existence of the decoding tree, it is clearly both necessary and sufficient that an instantaneous code has no code word s_i that is a prefix of another code word s_j.

2.3.3 Construction of Instantaneous Codes

It is clear that of all uniquely decodable codes, the instantaneous codes are preferable since they cost nothing extra to decode the coded message. To construct instantaneous codes, consider a source alphabet of five symbols $S = \{s_i \mid i = 1, \ldots, 5\}$ which is to be coded in binary alphabet. To get the instantaneous code the source symbols can be assigned the following prefix codes.

$s_1 = 0$

$s_2 = 10$

$s_3 = 110$

$s_4 = 1110$

$s_5 = 1111$

And the corresponding decoding tree is as in Fig.2.5.

In this construction the use of 0 for the first symbol reduced the number of possibilities available later. Instead of this, if the first two source symbols are encoded using two code symbols as $s_1 = 00$ and $s_2 = 01$ then s_3 can be encoded as $s_3 = 10$. There are two source symbols yet to encode, so s_4

cannot use $s_4 = 11$. Therefore, s_4 must be encoded by 110 leaving 111 for s_5. The complete codeword set is

$s_1 = 00$

$s_2 = 01$

$s_3 = 10$

$s_4 = 110$

$s_5 = 111$

This codeword set is clearly instantaneous since no codeword is a prefix of any other codeword, and the decoding tree is easily constructed as in Fig.2.6.

Which of these two codes is better, i.e., more efficient depends on the frequency of occurrence of the source symbols s_i.

2.3.4 The Kraft Inequality

The Kraft inequality gives the condition on the existence of instantaneous codes. It tells when the lengths of the code words permit forming an instantaneous code, but it does not discuss the code itself.

Theorem: A necessary and sufficient condition for the existence of an instantaneous code S of q symbols $S = \{s_i \mid i = 1, \ldots, q\}$ with encoded words of lengths $l_1 \le l_2 \le l_3 \le \ldots \le l_q$ is

$$\sum_{i=1}^{q} \frac{1}{r^{l_i}} \le 1$$

where r is the radix (number of symbols) of the alphabet of the code symbols.

This inequality is asserting that the encoder cannot have too many short codewords. Most of the l_i must be reasonably large.

It is easy to prove the Kraft inequality from the decoding tree, whose existence follows from the instantaneous decodability. Here the proof is given by induction. The decoding tree is given in Fig.2.7. For simplicity consider first the binary case. For a tree whose maximum height is 1, the decoding tree contains one or two branches of height 1. Thus the inequality should have either $\frac{1}{2} \le 1$ for one symbol or $\frac{1}{2} + \frac{1}{2} \le 1$ for two symbols.

Now assume that the Kraft inequality is true for all trees of height less than n. Now given a tree of maximum height n, the

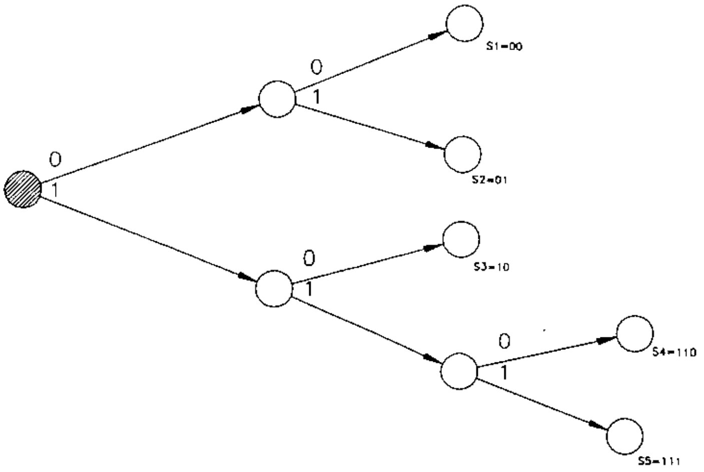
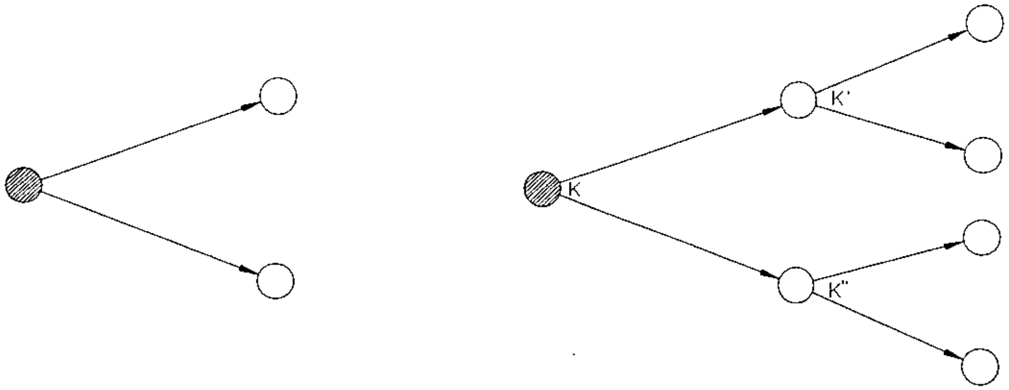


Fig. 2.6 : Decoding tree for 5 symbols with codeword set {00, 01, 10, 110, 111}.



(a) true for tree of length 1

(b) Assume true for length n−1

(c) Tree of length n

$1/2K' + 1/2K'' = K$

Fig. 2.7 Proof of Kraft inequality

first node leads to a pair of subtrees of height at most $n - 1$, for which the inequalities $K' \le 1$ and $K'' \le 1$, where K' and K'' are the values of their respective sums. Each height l_i in a subtree is increased by 1 when the subtree is joined to the main tree, so an extra factor of $\frac{1}{2}$ appears. Therefore, the inequalities will be

$$\tfrac{1}{2} K' + \tfrac{1}{2} K'' \le 1$$

For radix r instead of binary, there are at most r branches at each node-at most r subtrees each with an extra factor of 1/r when joined to the main tree. Again the theorem is true.

A moment's inspection shows that if every terminal node of the tree is a code word, then $K = 1$. It is only when some terminal nodes are not used that the inequality occurs. But if any terminal node is not used for a binary code alphabet, the preceding decision is wasted and that corresponding digit can be removed from every symbol that passes through this node in its decoding. Thus if the inequality holds, the code is inefficient, and how to correct this is immediately evident for binary trees. Thus $K = 1$ for binary trees with all the terminals used. It is only for radix $r > 2$ that it is reasonable to have unused terminals and hence a K less than 1. Since the theorem gives a condition on the heights only, the main use is in questions of the existence of a code with a given set of lengths.

Again the theorem refers to the existence of such a code, and does not refer to a particular code. A particular code may obey the Kraft inequality and still not be instantaneous, but there will exist codes that have the same l_i and are instantaneous.

If the lengths of four source symbols in binary are 1, 3, 3, 3, the Kraft sum will be 1/2 + 3/8 = 7/8, and an instantaneous code with those lengths is possible. One of the words of length 3 could be shortened to 2 bits. But if the lengths were 1, 2, 2, 3, the sum would be 1/2 + 2(1/4) + 1/8 = 9/8, and such an instantaneous code could not exist. In the above two examples, only the code word lengths are given since this is what matters in the theorem, not the actual code words.

2.3.5 Shortened Block Codes

In the fixed length codes, i.e., block codes if there are exactly 2^m code words in a binary system (r^m in a radix r system), exactly m digits could be used to represent each symbol. But there does not have an exact power of the radix but still need the maximum length to be as short as possible. To see what can happen, consider the case of five symbols. Of the eight binary symbols listed in the next page the encoder can drop any three.

000

001

010

011

100

101

110

111

If 001, 011, and 101, dropped, then three branches of the decoding tree can be shortened and still have instantaneous decodability. The codewords would be

$s_1 = 00$

$s_2 = 01$

$s_3 = 10$

$s_4 = 110$

$s_5 = 111$

and corresponding decoding tree is given in Fig.2.8.

Instead of this choice, the decoder can drop 001, 010, and 011 and shorten only one branch of the tree to the code as in Fig.2.9 and the codewords are

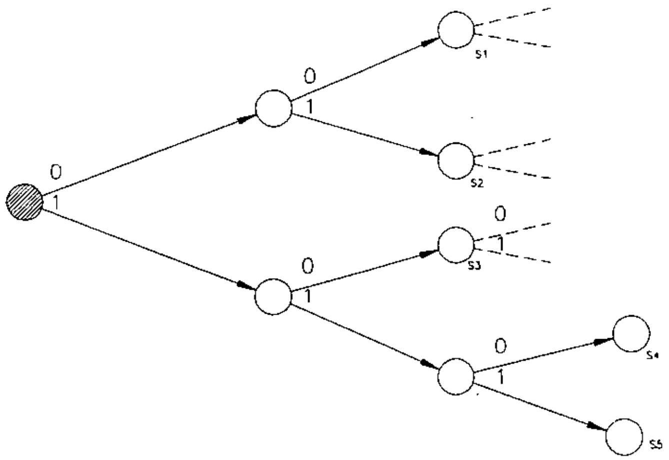


Fig. 2.8 : Decoding tree for shortened block codes after dropping {001, 011 and 101}.

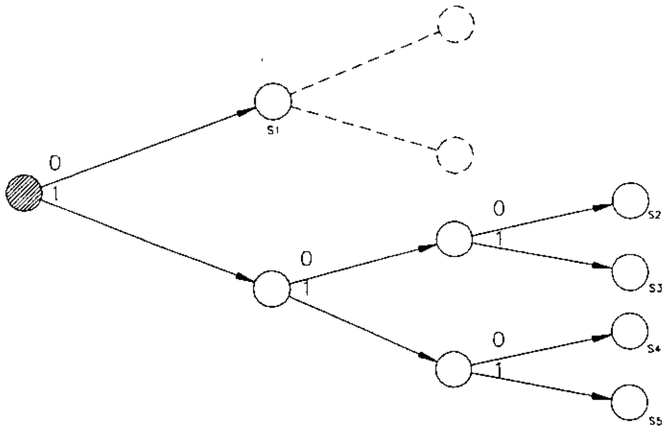


Fig. 2.9 : Decoding tree for shortened block codes after dropping {001, 010 and 011}.

$s_1 = 0$

$s_2 = 100$

$s_3 = 101$

$s_4 = 110$

$s_5 = 111$

In both cases there are now no unused terminals and therefore $K = 1$. These codes are called shortened block codes; they are essentially block codes with small modifications.

2.3.6 The McMillan Inequality

The Kraft inequality applies to instantaneous codes, which are a special case of uniquely decodable codes. McMillan showed that the same inequality applies to uniquely decodable codes. The underlying idea of the proof of the necessity is that very high powers of a number greater than 1 grow rapidly. If it is possible to bound tightly this growth, then it is clear that the number is not greater than 1. The proof of the sufficiency follows from the fact that it can be done for instantaneous codes which are special cases of uniquely decodable codes.

The necessity part of the proof begins by taking the nth power of the Kraft expression

$$\left[\sum_{i=1}^{q} \frac{1}{r^{l_i}}\right]^{n} \equiv K^{n}$$

If it is expanded, the left-hand terms will be a sum of many terms having various powers: the exponents running from **n**, the lowest possible power, to **nl**, the highest, where l is the length of the longest symbol. Thus the expression becomes

$$K^{n} = \sum_{k=n}^{nl} \frac{N_k}{r^{k}}$$

where $\mathbf{N_k}$ is the number of code symbols (of radix r) of length **k**. Since the code is uniquely decodable, $\mathbf{N_k}$ cannot be greater than r^k, which is the number of distinct sequences of length **k** in the code alphabet of radix **r**. Therefore, the bound is

$$K^{n} \leq \sum_{k=n}^{nl} \frac{r^{k}}{r^{k}} = nl - n + 1 < nl$$

The **+1** comes from the fact that both end terms in the sum are counted. This is the inequality for decodable codes, since for any x > 1 a sufficiently large **n** makes the number x^n > **nl**. But **n** can be chosen to be a very large value, and it follows, therefore, that the number **K** (the Kraft sum) must be ≤ **1**.

From this it is clear that there is very little to gain from avoiding instantaneously decodable codes and using the more general uniquely decodable codes; both have to satisfy the same Kraft inequality on the lengths of the encoded symbols.

2.3.7 Information Contents

Suppose that the encoder has the source alphabet $S = \{s_1, s_2, s_3, \ldots, s_q\}$, of q symbols with corresponding probabilities $P = \{p_1, p_2, p_3, \ldots, p_q\}$, such that $p_i = p(s_i)$. When the encoder receives one of these symbols, how much information does it get? For example, if $p_1 = 1$ (and all the other $p_i = 0$), then there is no surprise, no information, since it is already known what the message must be. On the other hand, if the probabilities are all very different, then when a symbol with a low probability arrives, the encoder feels more surprised, gets more information, than when a symbol with a higher probability arrives. Thus information is somewhat inversely related to the probability of occurrence.

So there should be a function I(p), which measures the amount of information - surprise, uncertainty - in the occurrence of a source symbol of probability p. The function I(p) must obey the following three assumptions:

(i) $I(p) \geq 0$ (a real non-negative measure).

(ii) $I(p_1 p_2) = I(p_1) + I(p_2)$ for independent source symbols.

(iii) I(p) is a continuous function of p.

The second of these conditions is known as the Cauchy functional equation for the function I(p), meaning that it serves to define I(p). If p_1 and p_2 are both the same number p, not necessarily the same event, then

$$I(p^2) = I(p) + I(p) = 2I(p)$$

Now if $p_1 = p$ and $p_2 = p^2$, then

$$I(p^3) = I(p) + 2I(p) = 3I(p)$$

and in general

$$I(p^n) = nI(p)$$

That is, the standard law of exponents for positive integers applies to the function I(p). Following this, the function can be adapted for the usual exponent extension to fractional values. Assume

$$p^n = y, \quad p = y^{1/n}$$

and hence

$$I(y) = nI(y^{1/n})$$

or, after some further manipulation,

$$I(y^{m/n}) = \frac{m}{n}I(y)$$

Thus for the rational numbers the function I(p) obeys the same formula as the log function.

The third assumption of continuity allows the function to extend this to all numbers ($0 \le p \le 1$), rational or irrational. Thus

$$I(p) = k \log(p)$$

for some constant k and some base of the log system. From the first assumption it is natural to pick the constant k as -1, and finally, the function will be

$$I(p) = -\log(p) = \log(1/p)$$

for some base of the log system. It is convenient to use the base 2 logs; the resulting unit of information is called a bit.

2.3.8 Entropy

In practice we are often more interested in the average information conveyed in some source symbol than in the specific information in each source symbol. Since p_i is the probability of getting the information $I(s_i)$, then the average information for each symbol s_i,

$$p_i I(s_i) = p_i \log_2(1/p_i)$$

From this it follows that on the average, over the whole source alphabet, the information will be

$$\sum_{i=1}^{q} p_i \log_2 \frac{1}{p_i}$$

For radix **r**,

$$H_r(S) = \sum_{i=1}^{q} p_i \log_r\left(\frac{1}{p_i}\right)$$

It has become a convention to label this important quantity as $H_r(S)$ and call it the entropy function for a distribution when all that is considered are the probabilities p_i of the symbols s_i. Corresponding to each distribution $P = (p_1, p_2, \ldots, p_q)$ of symbols s_i, there is a single number called the entropy and labelled **H(S)**. This is analogous to the usual idea of an average of a distribution - the average is a single number which summarizes the distribution. The entropy H(S) is the weighted average of the logs of the reciprocals of the probabilities of the distribution. The entropy is a single measure of a distribution; it is the average information of the alphabet S.

2.3.9 Entropy and Coding

There exists a fundamental relationship between the average code length L_{av} and the entropy $H_r(S)$. Given any instantaneous code it has some definite codeword lengths l_i represented in some radix r. From the Kraft inequality,

$$K=\sum_{i=1}^{q}\left(\frac{1}{r^{l_i}}\right)\leq 1$$

Assume a numbers $\mathbf{Q_i}$ (pseudo probabilities):

$$Q_i=\frac{r^{-l_i}}{K}$$

where,

$$\sum_{i=1}^{q} Q_i=1$$

The $\mathbf{Q_i}$ may be regarded as a probability distribution. Therefore, the fundamental Gribbs inequality

$$\sum_{i=1}^{q} p_i \log_2\left(\frac{Q_i}{p_i}\right)\leq 0$$

Upon expanding the log term into a sum of logs, one term leads to the entropy function,

$$H_2(S)=\sum_{i=1}^{q} p_i \log_2\left(\frac{1}{p_i}\right)\leq \sum_{i=1}^{q} p_i \log_2\left(\frac{1}{Q_i}\right)$$

$$\leq \sum_{i=1}^{q} p_i(\log_2 K-\log_2 r^{-l_i})$$

$$\leq \log_2 K + \sum_{i=1}^{q} p_i l_i \log_2 r$$

By Kraft inequality $\mathbf{K \leq 1}$, so that $\mathbf{\log_2 K \leq 0}$. Dropping this term can only strengthen the inequality. Therefore,

$$H_2(S) \leq \sum_{i=1}^{q} (p_i l_i) \log_2 r = L_{av} \log_2 r$$

or,

$$H_r(S) \leq L$$

where L is the average code length,

$$L = \sum_{i=1}^{q} p_i l_i$$

This is the fundamental result that the entropy supplies a lower bound on the average code length L for any instantaneous decodable system. By the McMillan inequality, it also supplies to any uniquely decodable system.

For efficient binary codes K = 1 and we have $\mathbf{\log_2 K = 0}$. Therefore, the inequality occurs in binary case only

$$p_i \neq Q_i = 2^{-l_i}$$

Huffman coding approaches the entropy in some probability distribution.

2.3.10 Redundancy

Redundancy is an important concept in information theory, particularly in connection with language. It is the presence of more code symbols in a message than is strictly necessary. For example, in a binary coding of two source symbols A and B, can be coded as 000 and B as 111, instead of A = 0 and B = 1. This gives some protection against errors, since one binary error in three digits could be tolerated. Redundancy is defined as

$$\text{Redundancy} = \frac{\text{maximum entropy} - \text{actual entropy}}{\text{maximum entropy}}$$

Spoken and written languages usually have high redundancy, permitting them to be understood in the presence of noise or errors.

CHAPTER THREE

STATISTICAL LOSSLESS COMPRESSION TECHNIQUES

A Lossless technique of data compression always produces the decompressed file that is identical to the original without losing even in a single bit. Most of the Lossless techniques implemented in software use statistics of the source symbols. In this chapter theoretical development of commonly used Lossless compression techniques are discussed.

3.1 Shannon-Fano Coding

The first well-known method of efficient variable length coding technique is known as Shannon-Fano coding. Claude Shannon at Bell Labs and R. M. Fano at M.I.T. developed this method nearly simultaneously. It depends on simply knowing the probability of each source symbol appearance in the message.

3.1.1 Conditions for variable length coding

The basic requirements for the variable length encoding of the source message are:

(a) No two source symbols will consist of identical arrangements of coding digits.

(b) The coded message will be instantaneously decodable, that is, the source message will be encoded so that no additional information is necessary to specify where a coded source symbol begins and ends once the starting point of the sequence of a source symbol is known in the encoded message.

(c) Though the codes are of different bit lengths, they can be uniquely decoded.

The kth prefix of a codeword is the first **k** code symbols (bits for binary code) of that codeword. Therefore, the condition (b) could be restated as: No message shall be coded in such a way that any one of its codeword is a prefix of another codeword, or that any of its prefixes are used elsewhere as a codeword.

For optimal coding, the length of codeword for a given source symbol can never be less than that of a more probable source symbol.

Assume a source alphabet $S = \{s_i \mid i = 1, 2, \ldots, q\}$, have probability $P = \{p_i \mid i = 1, 2, \ldots, q\}$ in a particular message and length set of codewords on the code alphabet

$X = \{x_i \mid i = 1, 2, \ldots, r\}$ be $L = \{l_i \mid i = 1, 2, \ldots, q\}$. Then for optimal codeword set,

$$p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_q, \quad \text{and}$$
$$l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_q$$

If both of the following conditions do not hold, then the code is not optimal in the sense that there could have a shorter average length by rearranging the codeword representation of the source alphabet.

Suppose that for some source symbols s_m & s_n (for $m > n$) we have both conditions,

$$p_m > p_n \quad \text{and} \quad l_m > l_n.$$

In computing the average length, the expression has two terms

$$p_m l_m + p_n l_n \qquad \ldots . \quad (i)$$

By interchanging s_m and s_n with their codewords corresponding terms could be

$$p_m l_n + p_n l_m \qquad \ldots . \quad (ii)$$

Subtracting the (i) from (ii), the change due to this rearrangement be (ii) - (i) :

$$p_m(l_n - l_m) + p_n(l_m - l_n) = (p_m - p_n)(l_n - l_m) < 0$$

From the forgoing assumptions this is a negative number; the average code length will decrease if the codewords for s_m and s_n are interchanged. Therefore, both of the above two running inequalities must hold.

Any source symbols coding technique following the above conditions will produce compact instantaneous codes.

3.1.2 Special Case of Variable Length Codes

There is an interesting special case[75], in which the symbol probabilities p_i are of the form $p_i = (\frac{1}{2})^{a_i}$, where a_i is integral. Perfect coding can be obtained by setting the lengths of codewords $l_i = a_i$. If the four symbols s_1, s_2, s_3, s_4 have probabilities 1/2, 1/4, 1/8, 1/8 respectively. Then using $p_i = \left(\frac{1}{2}\right)^{a_i}$,

$$l_1 = 1,\ l_2 = 2,\ l_3 = l_4 = 3.$$

So a suitable code would be

$$s_1 = 0,\ s = 10,\ s_3 = 110,\ s_4 = 111.$$

The average length of the codewords is $L = \Sigma p_i l_i = 1.75$ and the source entropy is $H = -\Sigma p_i \log(p_i) = 1.75$ bits per symbols numerically equal to the average length. This is not always possible.

3.1.3 Boundaries of Shannon-Fano Codes

Given the source symbols s_1, s_2, ..., s_q and their corresponding probabilities p_1, p_2, ..., p_q, then for each p_i there is an integer l_i such that

$$\log_r\left(\frac{1}{p_i}\right) \leq l_i < \log_r\left(\frac{1}{p_i}\right) + 1 \qquad (3.1)$$

since the two extreme values just span a unit length. Removing the logs, the inequality would be

$$\left(\frac{1}{p_i}\right) \leq r^{l_i} < \left(\frac{r}{p_i}\right)$$

Taking the reciprocal of each term, we obtain

$$p_i \geq \left(\frac{1}{p_i}\right) > \frac{p_i}{r}$$

Since $\Sigma\ p_i = 1$, when we sum this inequality, we get

$$1 \geq \sum_{i=1}^{q} \left(\frac{1}{r^{l_i}}\right) > \frac{1}{r}$$

which gives the Kraft inequality. Therefore, there is an instantaneous decodable code having these Shannon-Fano lengths.

To get the entropy of the distribution of p_i, we multiply the inequality (3.1) by p_i and sum:

$$H_r(s) = \sum_{i=1}^{q} p_i \log_r \frac{1}{p_i} \leq \sum_{i=1}^{q} p_i l_i \leq H_r(S) + 1$$

In terms of the average length L_{av} of the code, we have

$$H_r(S) \leq L_{av} < H_r(S) + 1$$

Thus, for Shannon-Fano coding we again have the entropy as a lower bound on the average length of the code. It is also part of the upper bound.

3.1.4 Shannon-Fano Algorithm

Shannon-Fano Coding technique produces instantaneous decodable codewords. This technique builds a decoding/encoding tree known as Shannon-Fano tree[53] and it can be built by following a simple algorithm. If a list of source symbols with a corresponding list of probabilities or frequency counts is given in the sorted form on the frequency counts, the tree is built by the following simple algorithm.

1. Form a node with the total frequency count, this node is the root of the tree.
2. Divide the list of symbols into two parts, with the total frequency counts of the upper half being as close to the total of the bottom half as possible.
3. The weight of upper half of the list is assigned to the right child of the previous node, and the lower half to the left.

4. Recursively apply the same procedure to each half, subdividing groups and adding to preceding nodes until each symbol has become a leaf on the tree.

The step by step procedures (as an example) of building a Shannon-Fano tree and codewords is shown in Fig.3.1 from a symbol set of five symbols and their frequency counts as in the Table 3.1.

Table 3.1: Five symbols and frequency counts for Shannon-Fano Coding.

Symbol	Count
s_1	1
s_2	1
s_3	2
s_4	2
s_5	3

In the Fig.3.1(a) putting a dividing line between symbols s_3 and s_4 assign a count of 5 to the lower group and 4 to the upper, the closest to exactly half. This means that s_4 and s_5 will be in the right of the root and each having a code that starts with a bit 1, and s_1, s_2 and s_3 are all in the left child of the root and going to start with a 0.

Subsequently, the lower half of the table gets a new division between s_4 and s_5 as in Fig.3.1(b) which puts s_4 on a leaf with

Symbol	Count	
s_1	1	
s_2	1	
s_3	2	
		Division 1
s_4	2	
s_5	3	

(a)

Symbol	Count	
s_1	1	
s_2	1	
s_3	2	
		Division 1
s_4	2	
		Division 2
s_5	3	

(b)

Symbol	Count	
s_1	1	
s_2	1	
		Division 3
s_4	2	
		Division 1
s_4	2	
		Division 2
s_5	3	

(c)

Symbol	Count	
s_1	1	
		Division 4
s_2	1	
		Division 3
s_3	2	
		Division 1
s_4	2	
		Division 2
s_5	3	

(d)

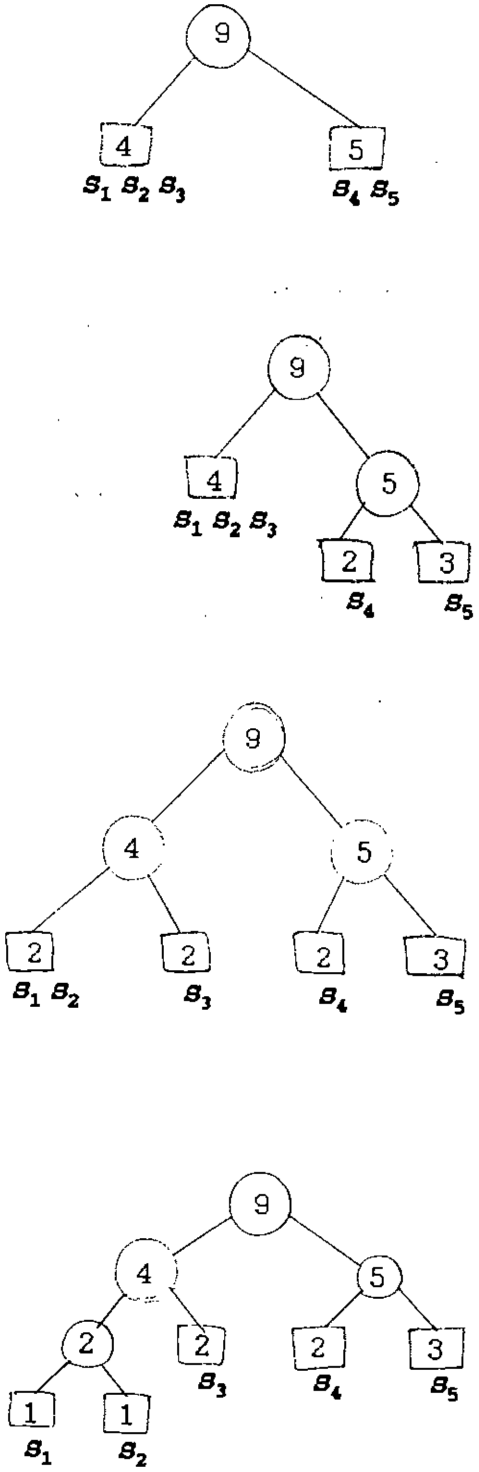


Fig.3.1: Step-by-step procedures of Shannon-Fano tree development.

code 10 and s_5 in a leaf with code **11**. After four divisions the final tree is built and gets the codeword set for the given symbol set. In the final codeword set, the three symbols with the highest frequencies have all been assigned 2-bit codes, and two symbols with lower counts have 3-bit codes.

3.2 Static **Huffman Coding**

Huffman coding is a statistical data-compression technique. It produces variable length codes for the source symbols. In this technique, both encoding and decoding are done by following an automata, i.e., a decision tree popularly known as Huffman tree. Its employment reduces the average code length used to represent the symbols of the source alphabet.

3.2.1 **Restrictions for Optimal Coding**

In addition to the conditions of variable length coding discussed previously, optimum coding would have the following restrictions.

Consider the maximum codeword length for the previous codeword set is l_q. If there is only one of such length, since the code is instantaneous, then any shorter codeword of length (l_q - 1) or shorter is not a prefix of the maximum-length codeword. Therefore, the last part of the longest codeword could be dropped with no loss of information in decoding. Thus at least two longest symbols must have the same length, and because of the running inequalities, they must be the two least probable.

Imagine an optimum coding in which no two of the source symbols coded with length l_n have identical prefixes of order (l_n - 1). Since an optimum coding has been assumed, then none of these codewords of length l_n can have codewords or prefixes of any order that correspond to other codewords. It would then be possible to drop the last digit of all of this group of codewords and thereby reduce the average code length. Therefore, in an optimum coding, it is necessary that at least two (and no more than r, the number of code symbols) of the codewords with length l_n have identical prefixes of order (l_n - 1).

Assume that there exists a source symbol s_n with a codeword c_n of length l_n and a combination of r different code symbols of length less than l_q which is not a prefix of a codeword. Then this combination of code symbols could be used as a codeword for s_n replacing the codeword c_n for the nth symbol with a consequent reduction of the average code length. Therefore, all possible sequences of $(l_n - 1)$ code symbols must be used either as a codeword or must have one of their prefixes used as codewords.

For a codeword set to be optimum for a particular source alphabet the above restrictions must be maintained.
The restrictions for an optimum coding are summarized below:

(a) No two source symbols will consist of identical arrangements of code symbols.

(b) No source symbol shall be coded in such a way that its codeword is a prefix of any other codeword, or that any of its prefixes are used elsewhere as a codeword.

(c) For $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_q$,

$$l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_{q-1} = l_q$$

(d) At least two and not more than **r**, (**r** is the number of code symbols) of codewords with length l have codes that are alike except for their final code symbol.

(e) Each possible sequence of $(l - 1)$ code symbols must be used either as a codeword or as a prefix of another codeword.

3.2.2 Binary Huffman Coding

In the binary Huffman coding, previously mentioned source alphabet $S = \{s_1, s_2, s_3, \ldots, s_q\}$, with corresponding probabilities $P = \{p_1, p_2, p_3, \ldots, p_q\}$ and length of the codeword $L = \{l_1, l_2, l_3, \ldots, l_q\}$ will be encoded into the code alphabet $X = \{0, 1\}$.

From the necessary restrictions for optimum codes, we show the development of the Huffman coding procedure.

Restriction (c) makes it necessary that the two least probable source symbols have codewords of equal length. Restriction (d) places the requirement that, for binary Huffman coding r is equal to two, therefore, be only two of the source symbols with coded length l_q are identical except for their last bits. The final bits of these two codewords will be one of the two binary digits 0 and 1. It will be necessary to assign these two codewords to the qth and the (q-1)th source symbols since at this point it is not known whether or not other codewords of length l_q exist. Once this has been done, these two source symbols are equivalent to a single composite source symbol. Its codeword (as yet undetermined) will be the common prefixes of order $(l_q - 1)$ of these two source symbols. Its probability will be the sum of the probabilities of the two source symbols from which it was created. The source alphabet containing this composite source symbol in the place of its two component source symbols will be called the first auxiliary source alphabet.

This newly created source alphabet contains one less source symbols than the original. Its symbols should be rearranged if necessary so that the source symbols are again ordered according to their probabilities. It may be considered exactly as the original source alphabet was. The codeword for each of the two least probable source symbols in this new source alphabet are required to be identical except in their final code symbols; 0 and 1 are assigned to these code symbols, one for each of the two source symbols. Each new auxiliary source alphabet contains one less source symbols than the preceding source alphabet. Each auxiliary source alphabet represents the original source alphabet with full use made of the accumulated necessary coding requirements.

The procedure is applied again and again until the number of source symbols in the most recently formed auxiliary source alphabet is reduced to two. One of each of the binary digits is assigned to each of these two composite source symbols. These source symbols are then combined to form a single composite source symbol with probability unity, and the coding is completed.

From the above discussion, Huffman coding scheme is a process of reduction of source symbols. At each stage 2 least probable symbols (**r** for r-ary code alphabet) are reduced to one symbol. Reversing the reduction process form the codeword for the symbols. The least probable symbol gets the longest codeword. Reversing process of reduction is called splitting procedure.

3.2.3 **Basic Machine for Huffman tree construction**

In binary tree construction problem[57] one is given a set of **n** leaves having corresponding weights $W = \{w_i \mid i = 1, 2, \ldots, n\}$. The weights need not to be normalized so that their sum comes out to be unity; we require only that they be non-negative and given, for convenience, sorted by index: $w_1 \leq w_2 \leq \ldots \leq w_n$. Construction of a binary tree on these leaves is then effected by **(n-1)** merges of pairs of available nodes. Each node in the pair is marked available, having as its weight some function **F** of the weights of its sons. Leaves are initially all marked available. Each internal node defines the root of a binary subtree of the constructed tree, which implies that tree construction can be defined inductively in terms of forests in the obvious way. The construction begins with a forest of n one-node trees and repeatedly reduces the number of trees by 1 via root merge operations until only one tree is left.

Weight Space U: A weight space of a weighted tree construction problem is a connected interval of the nonnegative reals R_+. All weights in the tree are elements of U.

Weight Combination Function $F:U^2 \rightarrow U$: A weight combination function $F:U^2 \rightarrow U$ is any symmetric function that is closed as a binary operator on U. F is used to produce the weight of internal nodes generated by merge operation in tree construction which is shown in Fig. 3.2.

Tree Cost Function $G:U^n \rightarrow R$: A tree cost function $G:U^n \rightarrow R$ for all trees having n internal nodes is any symmetric mapping of U^n into the real numbers R. For such a tree T, the cost of T will be $G(W_i \mid i = 1,2,\ldots,n)$, i.e., the value of G applied to the internal node weights of T.

Huffman algorithm for binary tree construction is now simple to state: To build the Huffman tree given a weight combination function F, merge at each step the two available nodes of smallest weight with ties resolved arbitrarily, until only one node is available.

In the use of file compression application, weight combination function $\mathbf{F(x,y) = x + y}$, and tree cost function G = sum with $U = \mathbf{R}_+$, which is $\Sigma w_j(T) l_j(T)$ and is called the weighted path length of $\mathbf{T}$.

Different choice of two available nodes of smallest weight, if their is a tie, form different Huffman trees and different codeword for the same source alphabet, but the average code length remains same. An example of reduction process and formation of the codeword using the above weight combination function and tree cost function is shown in the following:

Assume a source alphabet of 5 symbols $S = \{s_1, s_2, s_3, s_4, s_5\}$. A message is to be encoded into binary code alphabet $X = \{0, 1\}$. The frequency count, i.e., the weight of the source symbols of the message is $\mathbf{W} = \{20, 10, 10, 5, 5\}$ with corresponding probabilities $P = \{0.4, 0.2, 0.2, 0.1, 0.1\}$, The reduction process and the formation of codeword are shown in Table 3.2 and 3.3.

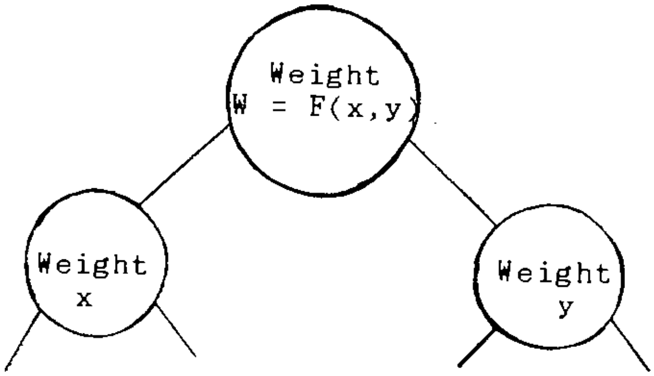


Fig.3.2: Weight combination function F(x,y).

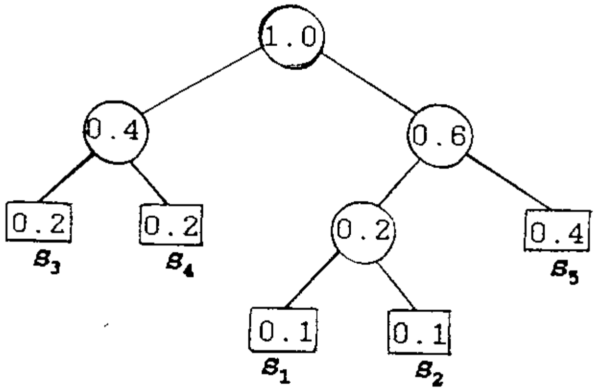


Fig.3.3: Static Huffman tree for symbols set $S = \{ s_i \mid i = 1, 2, \ldots, 5\}$ with probability set $P = \{0.1, 0.1, 0.2, 0.2, 0.4\}$.

This process forms a tree which is called Huffman tree. The source symbols with their weights are labeled at each node of the tree. The correspondence between the tree and the codeword is simply to represent the path from the root to each external node as a string of 0' and 1's, where 0 represents to a left branch and 1 represents to a right branch. An external node at level *l* corresponds in this way to a string in code alphabet of length *l*. The Huffman tree for the above example is shown in the Fig.3.3.

The tree formation steps from the above example is shown in Fig.3.4. In this example, the tie of selecting the smallest weighted available nodes in each merging step is solved by inserting the new parent node into the list of available nodes at the far end of the block of nodes with equal weights. In this process produced codeword set has the minimum variance of codeword length. Different choices of the smallest weighted available node form different Huffman trees and corresponding codeword set, when there is a tie, is also shown in Fig.3.5 for the above example.

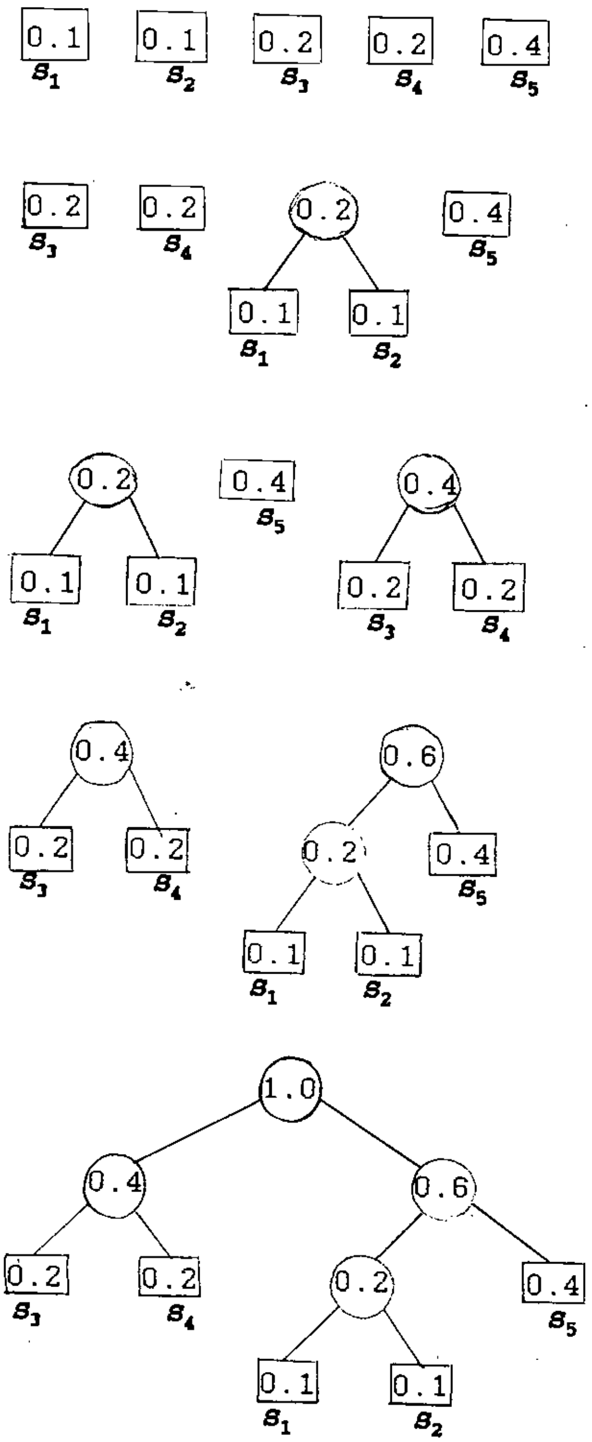
A prefix code is a set of string in which no string is a proper prefix of another. A minimal prefix code is a prefix code such that, if α is a proper prefix of some string in the set, then $\alpha 0$ is either in the set or a proper prefix of some string in the set, and so is $\alpha 1$. Binary trees with n external nodes are in one-to-one correspondence with sets of n string on {0, 1} that form a minimal prefix code. Binary Huffman tree is an optimum minimal prefix code.

Table 3.2 : Reduction process of the probabilities.

S	P	Probabilities of auxiliary source symbols			
s_1	0.1	0.2	0.2	0.4	1.0
s_2	0.1	0.2	0.4	0.6	
s_3	0.2	0.2	0.4		
s_4	0.2	0.4			
s_5	0.4				
	Original	First	Reductions Second	Third	

Table 3.3: Splitting Process (formation of codeword)

S	P	Auxiliary codewords			
s_1	100 ← 0.1	100 → 0.2	00 0.2	10 0.4	0 → 1.0
s_2	101 ← 0.1	101 0.2	01 → 0.4	11 → 0.6	1
s_3	00 ← 0.2	00 0.2	10 0.4	0	
s_4	01 ← 0.2	01 0.4	11		
s_5	11 ← 0.4	11			
	Final Codes	Third	Splitting Second	First	



Probability distribution of symbols

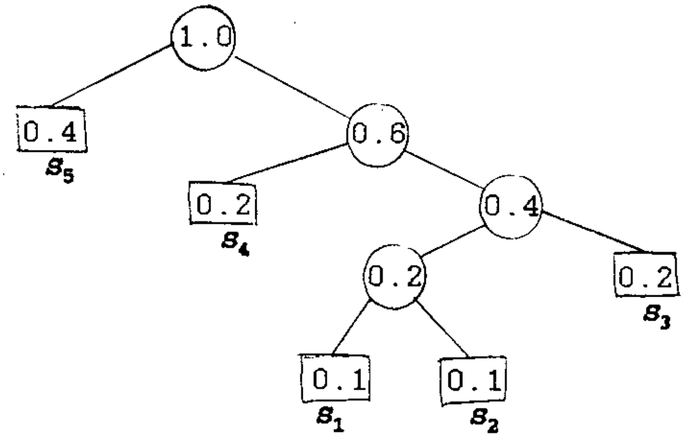
Reduction step 1

Reduction step 2

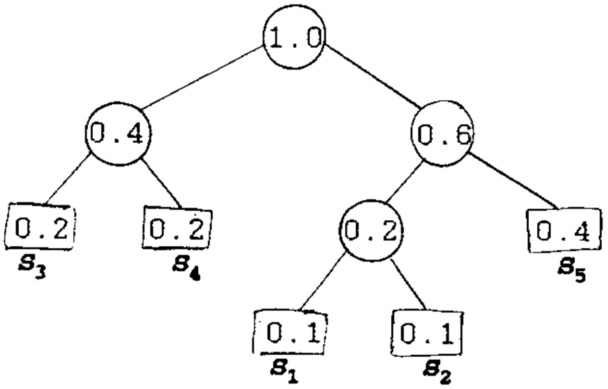
Reduction step 3

Final tree

Fig.3.4: Step-by-step procedures of static Huffman tree construction.



(a) New node is at the front of the block of nodes with same weight.
Codeword set = {1100, 1101, 111, 10, 0},
$\sum w_i l_i = 2.2$, $\sum l_i = 14$, $\max\{l_i\} = 4$.



(b) New node is at the end of the block of nodes with same weight.
Codeword set = {100, 101, 00, 01, 11},
$\sum w_i l_i = 2.2$, $\sum l_i = 12$, $\max\{l_i\} = 3$.

Fig.3.5: Different Huffman trees for the same set of probabilities P = {0.1, 0.1, 0.2, 0.2, 0.4}.

As weight set **W** of **n** source symbols forms a Huffman tree which is a binary tree with **n** external node and **(n-1)** internal nodes, where external nodes are labeled with the weight set **W** in some order. Huffman has minimum value of weighted path length over all such binary trees.

The average length of the encoded message over as alphabet of q symbols is

$$Lav = \sum_{i=1}^{q} p_i l_i$$

where

p_i = the probability of the ith source symbol, and
l_i = the length of its coded symbol.

3.2.4 General Huffman Tree

Optimum coding of the source symbols with code alphabet of radix **r** can be done by modification of binary Huffman coding scheme. To satisfy the restriction (e) of the optimum coding scheme, always combine **r** symbols into a single symbol in each reduction stage, with the probability that is the sum of the probabilities of the individual symbols. Thus merging of **r** symbols reduces the number of symbols by **(r-1)**. Therefore, if $\mathbf{n_l}$ is the number of symbols in the first auxiliary source symbols, then $(\mathbf{n_l}-1)/(\mathbf{r}-1)$ must be an integer. However, $\mathbf{n_l}$ =

$n - n_0 + 1$, where n_0 is the number of least probable symbols to combine at the first reduction stage. Therefore, n_0 must be such that $(n-n_0)/(r-1)$ is an integer and $2 \leq n_0 \leq r$. Reduction process for code alphabet $X = \{0,1,2,3\}$ for the previous example is shown in the Table 3.4 and corresponding tree is shown in the Fig.3.6.

3.2.5 Data Structure of r-ary Tree

The data structure of Huffman codes is a directed tree in which each branch represents a code symbol and each terminal node a codeword. The terminal nodes are all occupied in the case of binary codes, but in the case of r-ary codes, some terminal nodes with the longest paths may be empty The number of empty nodes is less than or equal to $(r - 2)$, and dummy symbols with probability zero are assigned to them.

The information source is defined by a pair (S,P) of source symbols $S = \{s_i \mid i = 1, 2, \ldots, q\}$ and a set of probabilities $P = \{p_i \mid i = 1, 2, \ldots, q\}$ $(p_1 \geq p_2 \geq \ldots \geq p_q)$. The code alphabet is decoded by $X = \{x_i \mid i = 1, 2, \ldots, r\}$. In general, S and X may be any set of symbols, but their homomorphic images $S^* = \{1, 2, \ldots, q\}$ and $X^* = \{1, 2, \ldots, r\}$ is used to treat the data structure more conveniently.

Table 3.4: Huffman coding procedure for D-ary (D = 4).

Merge Probabilities				Code Lengths L(i)	Code-words
Original P	Probabilities of Auxiliary source symbols				
			1.00		
		0.40			
0.22	0.22	0.22		1	1
0.20	0.20	0.20		1	2
0.18	0.18	0.18		1	3
0.15	0.15			2	00
0.10	0.10			2	01
0.08	0.08			2	02
	0.07				
0.05				3	030
0.02				3	031

The tree structure of r-ary Huffman codes can be completely specified by a two-dimensional array

$$\left\lceil \frac{(q-1)}{(r-1)} \right\rceil x\, r$$

where $\lceil A \rceil$ denotes the smallest integer greater than or equal to $|A|$.

Consider a directed tree of an r-ary Huffman code and assign an ordinal number i (i = 1, 2, ..., α) to each non-terminal node and a code symbol j (j = 1, 2, ..., r) to each branch in an orderly manner as depicted in Fig.3.7. To each terminal node is assigned a negative number **k** (k = -1, -2, ..., -q) whose absolute value corresponds to a source symbol. A two-dimensional $\alpha \times r$ array $M(i, j)$ ($1 \le i \le \alpha$, $1 \le j \le r$) can be formed such that i corresponds to a non-terminal node and j to a code symbol on each branch. The number of non-terminal nodes α will be evaluated later. Each element of $M(i,j)$ is determined by the following rule. If a code symbol j is assigned to a branch which combines the ith non-terminal node and the kth node, then $M(i,j) = k$ where if k is positive (negative), the kth node, is a non-terminal (terminal) node. By applying this rule to all pairs (i,j), $M(i,j)$ can be determined. The number of non-terminal nodes α is obtained as follows. Since α is equal to the number of reduction times and $(r-1)$ nodes are reduced at a time in constructing the Huffman code, α is the smallest integer which satisfies the inequality $\alpha(r - 1) + 1 \ge q$. Hence $\alpha = \lceil (q-1)/(r-1) \rceil$. Finally, as the number of dummy symbols is

$$N_d = (r - 1)\left\lceil \frac{(q - 1)}{(r - 1)} \right\rceil + 1 - q \le r - 2,$$

the elements $M(\alpha,j)$ ($r - N_d + 1 \le j \le r$) are all empty, but for convenience these empty elements are filled with dummy symbols $-(q + 1), -(q + 2), \ldots, -(q + N_d)$, respectively. It is noted here that the necessary and sufficient storage capacity is given by

$$N = \left\lceil \frac{(q-1)}{(r-1)} \right\rceil r - N_d = \left\lceil \frac{r(q-1)}{(r-1)} \right\rceil \leq 2(q-1)$$

where each storage location consists of $\lceil \log_2 q \rceil$ bits. The inversion of the data structure is accomplished as follows. First the number

$$l(i,j) = k - 1, \quad \text{if } k \text{ is positive, or}$$

$$l(i,j) = \left\lceil \frac{(q-1)}{(r-1)} \right\rceil - k - 1, \quad \text{if } k \text{ is negative,} \quad \ldots \quad (a)$$

is associated to the element $M(i,j) = k$ of the array. Then the numbers computed using the equation

$$M^{-1}(l) = 2^{\lceil \log_2 r \rceil} x \, (i-1) + (j-1)$$

and ordered them according to increasing $l(i,j)$ in a one dimensional array. The ordinal numbers

$$l \left(1 \leq l \leq \left\lceil \frac{(q-1)}{(r-1)} \right\rceil - 1 \right)$$

correspond to the non-terminal nodes and

$$l \left(\left\lceil \frac{(q-1)}{(r-1)} \right\rceil \leq l \leq \left\lceil \frac{r(q-1)}{(r-1)} \right\rceil \right)$$

to the source symbols. It is interesting to notice that the necessary and sufficient storage capacity for the inverted data structure $M^{-1}(l)$ is also equal to N given previously.

As an example, consider a small source $Y = (S, P)$ with a six symbols **S**, its homomorphic image $S^{\ddagger}$, and a set of six symbol probabilities **P** given in the Table 3.5. Then using the above scheme of Huffman tree representation, a binary Huffman code over $X^{\ddagger} = \{1, 2\}$ can be constructed and given in Table 3.5. The data structure and its inversion can be determined as shown in the Table 3.6 and 3.7.

Table 3.5 : An example of Huffman Code

S	$S^{\ddagger}$	P	C
s_1	1	0.4	2
s_2	2	0.3	11
s_3	3	0.1	122
s_4	4	0.1	1211
s_5	5	0.06	12121
s_6	6	0.04	12122

Table 3.6 : Data Structure M(i,j)

	j	
i	1	2
1	2	-1
2	-2	3
3	4	-3
4	-4	5
5	-5	-6

3.2.6 Decoding Automata

The automata[72,74] concept is applied to decoding the Huffman code. The input to the automaton is a semi-infinite sequence of code symbols, and the output is a semi-infinite sequence of source symbols. The automaton reads each input sequence of input symbols makes up a codeword, and if it does, produces a source symbol corresponding to the codeword.

Table 3.7: Inverted Data Structure $M^{-1}(l)$

l	$M^{-1}(l)$
1	0
2	3
3	4
4	7
5	1
6	2
7	5
8	6
9	8
10	9

The data structure **M(i, j)** specifies the next state **k** when the present state is i, the input is j, and $M(i,j) = k$ is positive. Then the automaton changes the state to k, and the process is repeated. If $M(i,j) = k$ is negative, the truncated input sequence is accepted as a codeword and the source symbol $-k \in S^{\ddagger}$ is produced. The automaton then changes the state to the initial state 1. The decoding automaton can be summarized as follow:

Huffman codes can be decoded by a finite-state automaton[72]

$A = \langle Q, \Sigma, \delta, q_0, F \rangle$, with output $k \in S^{\ddagger}$,

where

$\Sigma = X^{\ddagger}$, an input-alphabet,

$F = \{-k;\ k \in S^{\ddagger}\}$, a set of final states,

$Q = \{i;\ 1 \le i \le \lceil (q - 1)/(r - 1) \rceil \} \cup F$, a set of sates,

$q_0 = \{1\}$, an initial state,

$\delta : Q \times \Sigma \to Q$, a state transition function.

The mapping δ means that if $\delta(i,j) = M(i,j) = k > 0$, then k shows the next state, and if $\delta(i,j) = M(i,j) = k < 0$, then $-k \in F$ shows the decoded source symbol. The automaton then goes into the initial state q_0.

A decoding automaton for the binary Huffman code in Table 3.5 is given by

$A = \langle Q, \Sigma, \delta, q_0, F \rangle$

where

$\Sigma = \{1, 2\}$,

$F = \{-1, -2, -3, -4, -5, -6\}$,

$Q = \{1, 2, 3, 4, 5\} \cup \{-1, -2, -3, -4, -5, -6\}$,

$q_0 = \{1\}$, and

$\delta(i,j) = M(i,j) = k > 0$ is the data structure in Table 3.6. The performance of the mapping δ is subject to the same remark as in the statement of the finite state automaton.

3.2.7 Encoding Automata

In general, Huffman encoding can be performed using a table in which the address number corresponds to a source symbol and the content to a codeword. Unfortunately, however, due to the variable length property of this code, it is difficult to read a codeword from a fixed-length storage without extra information, such as its codeword length.

Here, using an inverted data structure of Huffman code, an encoding automaton is introduced to overcome this difficulty. The encoding procedure is as follows. First, prepare an inverted data structure $M^{-1}(l)$ $(1 \le l \le \lceil r(q - 1)/(r - 1) \rceil)$ and a push-down stack whose depth is not less than the maximum codeword length. The input to the automaton is a source symbol -k $\in S^{\ddagger}$, which specifies the initial state l of $M^{-1}(l)$, i.e., a certain address number calculated from k by the equation (a). Next, if the content of $M^{-1}(l)$ for some l is $2^{\lceil \log_2 r \rceil} \times i + (j-1)$, then push down symbol j, jump to address i, and repeat the process. When i = 0, push down symbol j and then pop up the sequence of code symbols in the stack, which makes up the codeword corresponding to the input source symbol. This concept of encoding automata is summarized as follows.

The encoding of Huffman codes is accomplished by a semiautonomous finite sequential machine.

$$B = \langle Q, \Sigma, Z, \delta, w \rangle$$

where

$\Sigma = \{l; \lceil (q-1)/(r-1) \rceil \leq l \leq \lceil r(q-1)/(r-1) \rceil\}$, an input alphabet,

$Q = \{l; 1 \leq l \leq \lceil (q-1)/(r-1) \rceil - 1\} \cup \Sigma$, a set of sates.

$Z = X^{\ddagger} = \{j; 1 \leq j \leq r\}$, an output alphabet,

$\delta : \delta(l) = i$, a next-state function,

$w : w(l) = j$, an output function,

$M^{-1}(l) = 2^{\lceil \log_2 r \rceil} \times i\,(j-1)$, and by a push-down stack[45] whose depth is not less than the maximum codeword length.

By semiautonomous we mean that the sequential machine is autonomous until a codeword is completed. The storage capacity and the average number of steps for encoding a source symbol are equal to those of the decoding automaton.

An encoding automaton of the binary Huffman code in Table 3.6 and 3.7 is a five-tuple

$$B = \langle Q, \Sigma, Z, \delta, w \rangle$$

where

$\Sigma = \{5, 6, 7, 8, 9, 10\}$,

$Q = \{1, 2, 3, 4\} \cup \{5, 6, 7, 8, 9, 10\}$

$Z = \{1, 2\}$

$\delta : \delta(1) = i$

$w : w(1) = j$

$M^{-1}(1) = 2 \times i + (j - 1)$

and by a push-down stack whose depth is not less than five.

3.3 Dynamic Huffman Coding

In static Huffman method of variable length coding, the encoder makes two passes over the message. In first pass, it collects the weights, i.e., the frequency counts of the source symbols of the message and then constructs the Huffman tree. The second pass encodes the source symbols into the code symbols to make the codewords based on the static Huffman tree structure. Both tree structure information and codewords have to be given to the decoder to decode to the original message. First the decoder makes the static Huffman tree from the tree structure information, then decodes the message. This causes extra disk accesses slowing down the algorithm for file compression application and causes delay when used for network communication.

In the dynamic coding, the coding is based on a dynamically varying Huffman tree instead of a single static tree. The

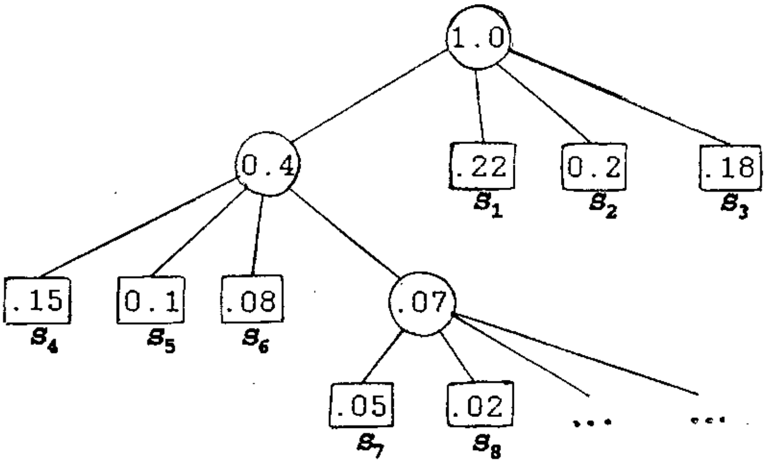


Fig.3.6: r-ary Huffman tree (r = 4) with symbol set given in Table 3.5.

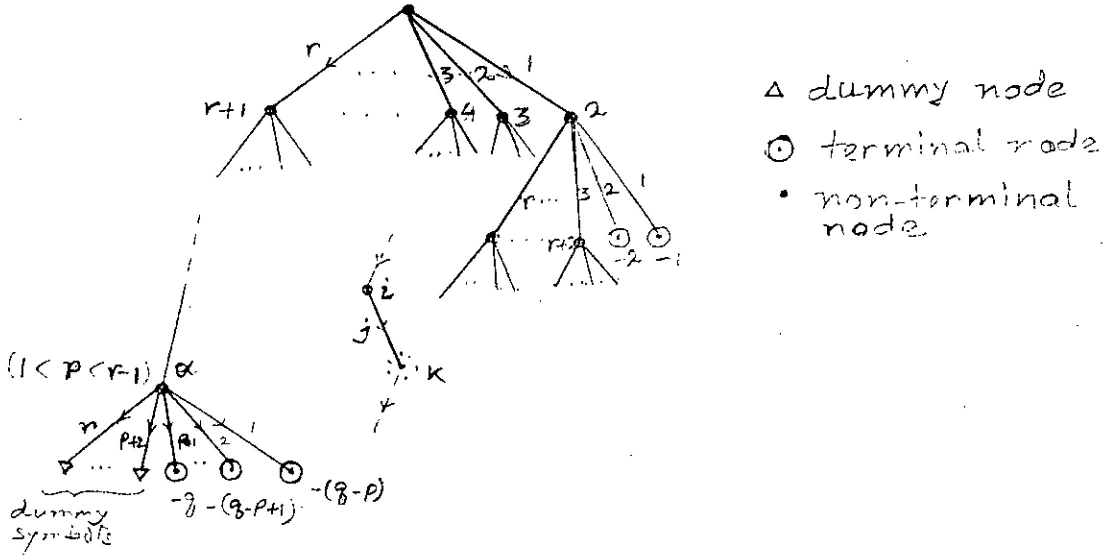
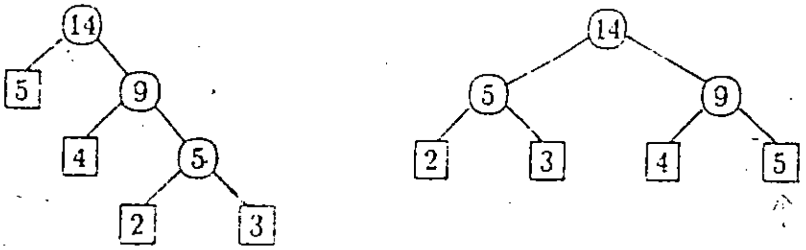


Fig.3.7: Typical directed tree of r-ary Huffman code.



(a) (b)

Fig.3.8: Different Huffman trees for weights {2, 3, 4, 5}.

(t + 1)st symbol of the message is encoded to its codeword based on the Huffman tree constructed for the weights of the previously processed portion of the message M_t of t symbols and learns the frequency of symbols of the message. The encoder encodes the (t + 1)st symbol in the message by a sequence of 0's and 1's that specify the path from the root to the leaf corresponding to the (t + 1)st symbol and makes dynamically varying prefix codes, then update the tree for the new frequency counts of the message M_{t+1}.

3.3.1 Strategy for Dynamic Huffman Coding

The Huffman algorithm combines the two smallest weights w_i and w_j, replacing them by their sum $w_i + w_j$ and repeats this process until only one weight is left. For example, given the leaf weights (2,3,4,5), the first step combines 2 + 3 = 5 and the remaining weights are (4,5,5). The next step combines 4 + 5 = 9, and then 5 and 9 are combined to form 14. There is some ambiguity about the nodes with equal weights, in this example, nodes with weight 5. Depending upon the selection, these procedures might form two different trees as in Fig.3.8. However, if the given weight 5 of the leaf node increase to 6, then the tree of Fig.3.8(a) is better, while if the weight 2 increase to 3 the tree of Fig.3.8(b) is better. A procedure for updating Huffman trees dynamically must, therefore, be able to convert from each of these possibilities to the other.

The weight combination process of Huffman algorithm with **n** leaf nodes leads to a nondecreasing sequence of node weights $U = (u_i \mid i = 1, 2, \ldots, 2n-1)$ for the internal and external nodes, and this sequence is the same for all Huffman trees on the given leaf weights $W = (w_i \mid i = 1, 2, \ldots, n)$. The n - 1 internal nodes of each Huffman tree that correspond to a particular sequence U have the weights $(u_1 + u_2, u_3 + u_4, \ldots, u_{2n-3} + u_{2n-2})$.

3.3.2 Sibling Property

A binary tree with **n** leaves of nonnegative weight is a **Huffman** tree if and only if

(i) the **n** leaves have nonnegative weights $W = (w_i, \mid i = 1, 2, \ldots, n)$, and the weight of each internal node is the sum of the weights of its children; and

(ii) the nodes can be numbered in nondecreasing order by weight, so that nodes $2j - 1$ and $2j$ are siblings, for $1 \leq j \leq n - 1$, and their common parent node is higher in the numbering.

The node numbering corresponds to the order in which the nodes are combined by Huffman algorithm.

Suppose that the message $M_t = m_1 m_2 m_3, \ldots, m_t$ has already been processed. The next source symbol m_{t+1} is encoded and decoded using a Huffman tree for M_t. The main difficulty is how to modify this tree quickly to get a Huffman tree for M_{t+1}. An example of the modification process is shown in the Fig.3.9, for the case $t = 32$, m_{t+1} = 'b'. Fig.3.9(a) shows the current status for message M_t, it is not good enough to simply increase by 1 the weights of m_{t+1}'s leaf and its ancestors, because the resulting tree will not be a Huffman tree, as it may violate the sibling property. The node will no longer be numbered in nondecreasing order by weight; node 4 will have weight 6 but node 5 will still have weight 5. Such a tree could, therefore, not be constructed by Huffman algorithm.

The solution can most easily be described as a two-phase process. In the first phase, the algorithm transforms the given into another Huffman tree for M_t, as shown in Fig.3.9(b). The incrementing process described above can be applied to this tree successfully in second phase and can be obtained a Huffman tree for M_{t+1} as in Fig.3.9(c).

3.3.3 Condition for Satisfying Sibling Properties

Given a tree satisfying sibling properties, with a sequence of nodes leading from some external node of weight w_i to the root be $Q_i = (q_i \mid j = 0, 1, 2, \ldots, l$; l be the path length of node w_i). If w_i is replaced by $(w_i + 1)$, then each of the weight $U_i = (u_i \mid j = 0,1,\ldots,l)$ must be increased by unity; the resulting tree will still satisfy sibling property provided that we had

$$u_i < u_{i+1} \qquad \text{for} \qquad 0 \leq j < l, \qquad (3.2)$$

in the original tree. Thus, the same tree will be optimum both for w_i and $w_i + 1$, whenever condition (3.2) holds.

If all leaf weights are positive, then it is always possible to transform a given Huffman tree into another one that satisfies the condition (3.2), by interchanging subtrees of equal weight by the following procedure:

First let i'_0 be maximum such that $u_i = u_{i'}$, and when i'_k has been defined let i'_{k+1} be maximum such that $u_{i_{k+1}}$ has the weight of the parent of q_{i_k}. If $i'_k = 2n - 1$, however, let $l' = k$ and terminate the construction. Now the tree can be permuted by interchanging the subtree rooted at q_{i_0} with the subtree rooted at q_{i_0}, Then interchanging the subtree at the parent of q_{i_0} with the subtree rooted at q_{i_1}, and so on. The final tree will satisfy the sibling properties, where the path from the character $a_{i_{l+1}}$ with weight w_i to the root is $q_{i_0}, q_{i_1}, \ldots, q_{i_l}$, and where $u_{i_j} < u_{i_{j+1}}$, for $0 \le j \le l'$. It is clear that $i_j \le i'_j$ for $0 \le j \le l'$ hence $l' \le l$; in other words, at most l interchanges are necessary to obtain a Huffman tree satisfying (3.2).

The construction in the preceding paragraph is the key to an efficient algorithm for maintaining optimal Huffman trees. In the previous example of Fig.3.9(a), the leave weight sequence of the processed portion of the message M_t is $W = (w_i \mid i = 1, 2, \ldots, 6) = (2, 3, 5, 5, 6, 11)$ and the label set $Q = (q_i \mid i = 1, 2, \ldots, 6) = (1, 2, \ldots, 6)$. Now the sequence of nodes from the leaf node of the symbol m_{t+1} = 'b' with weight 3 from the original Huffman tree be $Q_i = (2, 4, 7, 10, 11)$ and that of the transformed tree would be $Q_i' = (2, 5, 9, 11)$ and $l' = 3$. After selecting the nodes transformation of the tree for M_t beings from the leaf node m_{t+1} as the current node. Then repeatedly interchange the contents of the current node, including the subtree rooted there, with that of the largest numbered node of the same weight from Q_i', and make the parent of the latter node the new current node. The current node in Fig.3.9(a) is initially node 2. No interchange is possible, so its parent node 4 becomes the new current node. The content of nodes 4 and 5 are then interchanged, and node 8 becomes the new current node. Finally, the contents of nodes 8 and 9 are interchanged, and node 11 becomes the new current node. The first phase halts when the root is reached. The resulting tree is shown in Fig.3.9(b). Since each interchange operation has done on nodes of the same weights, the new tree will be a Huffman tree for the message M_t. In the second phase this tree turns into the desired Huffman tree for the message M_{t+1} by increasing the weights of m_{t+1}'s leaf and its ancestors by 1. The final tree is shown in the Fig.3.9(c). These two phases can be combined into one in the implementation in the algorithm.

3.3.4 Maintaining Symbols with zero-weights

If the number of distinct symbols processed so far is less than the total number of symbols in the alphabet, then a zero weight leaf node is present in the dynamic Huffman tree. In this situation there is a chance that a node and its parent both have same weight. If current (t + 1)st symbol is not in the message M_t, the 0-node is split to create a leaf node for it, as illustrated in the Fig.3.10. The (t+1)st symbol is encoded following the path from the root to the zero leaf node followed by an extra code that will represent the (t+1)th symbol. The representation of this (t+1)st symbol can be the ASCII code, but is even more appropriate to use minimal prefix code for the first appearance of the symbol in the message. When there are **m** symbols (a_1, a_2, ..., a_m) of weight zero, assume that $m = 2^e + r$, and $0 \le r < 2^e$. Symbol a_k is encoded as the (e + 1)-bit binary representation of k - 1, if $1 \le k \le 2r$, otherwise as the e-bit binary representation of (k - r - 1). For example, if m = 5' then e = 2, r = 1, and the encoding are:

a_1 -> 000, a_2 -> 001, a_3 -> 01, a_4 -> 10, a_5 -> 11.

This encoding is optimum when all letters have the weight e, for any e > 0.

3.3.5 Example of Dynamic Huffman Coding

The methods sketched above lead to a real-time algorithm for maintaining Huffman trees as the weights change. Here an example of dynamic Huffman coding is given showing the detailed constructions of the codes.

Before encoding or decoding begins, both encoder and decoder know only the size n of the alphabet being encoded. Assume there are just 27 symbols namely a, b, ..., z, and !; the last

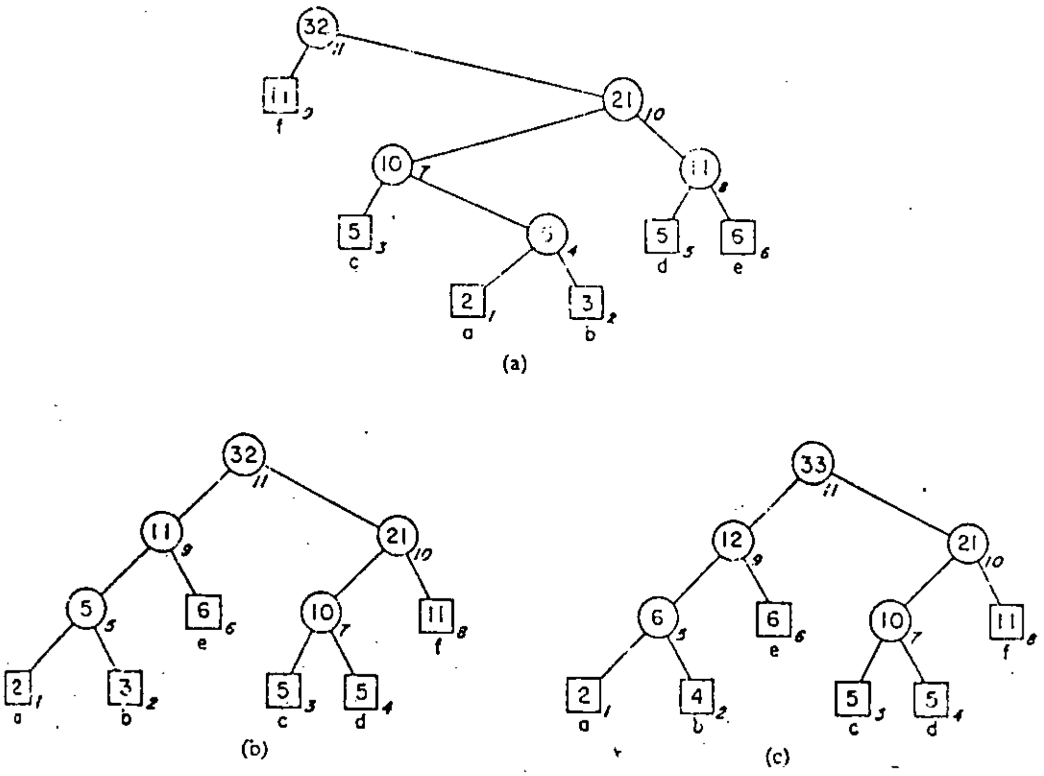


Fig.3.9: Basic idea of dynamic Huffman coding algorithm.

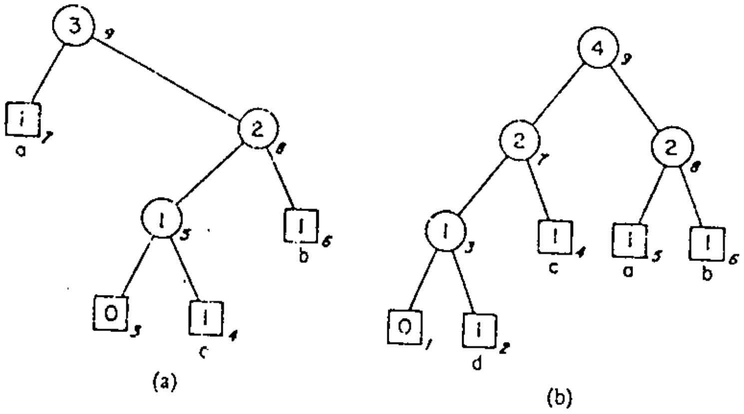


Fig.3.10: Dynamic Huffman algorithm operating on the message "abcd...", (a) The Huffman tree immediately before the fourth letter "**d**' is processed. The encoding for "**d**" is specified by the path to the 0-node, namely **100**, (b) After updating the tree.

symbol will be used only as the final character of the message. Since the adaptive encoding scheme seems to have a somewhat magic flavor, the encoder shall attempt to encode an arbitrary message **abracadabra!**.

Initially all weights are zero, so the first letter is encoded by the 0-weight scheme described at the end of section 2, where m = 27 = 2^4 + 11:

a -> 00000.

Now a has weight 1 and the other letters **(!, b,..., z)** have weight 0. In the list of remaining letters, ! has swapped places with **a** so that minimal changes need to be made to the data structure. The coding scheme at this point is represented by the tree in Fig.3.11(a), therefore the second letter of our message would be encoded simply as 1 if it were another **a**. However, it is a b, which comes out as an encoded 0-weight letter, prefixed by 0:

b ->000001

At this point the Huffman tree in shown by Fig.3.11(b). The third letter is, therefore, encoded as

r ->0010001

after which the Huffman tree has changed ,to Fig. 3.11(c), assuming that the algorithms of section 5 have been used. The encodings continue as

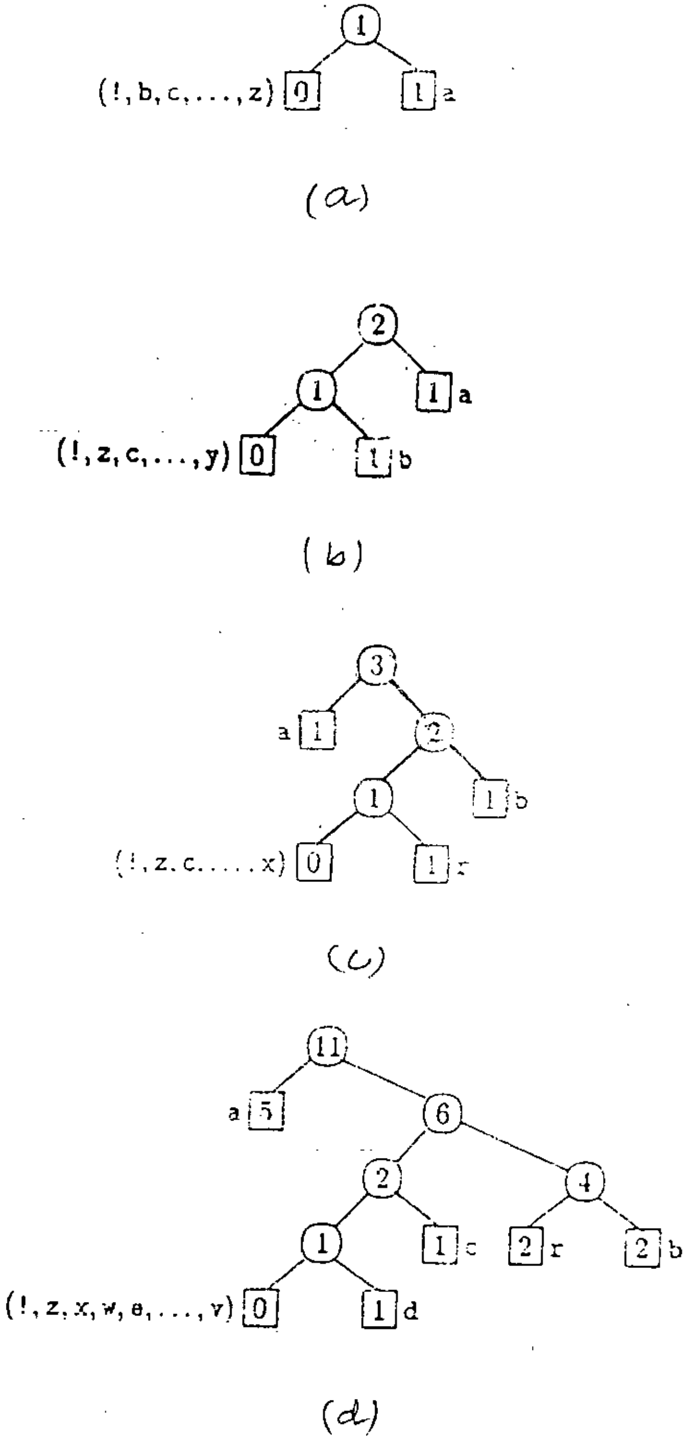


Fig.3.11: Example of Dynamic Huffman Algorithm with optimum 0-node encoding.

a -> 0

c -> 10000010

a -> 0

d -> 110000011

a -> 0

b -> 110

r -> 110

a -> 0

and by this time the tree has grown to Fig.3.11(d). The final symbol is now transmitted:

! -> 100000000.

If fixed length code (ASCII) is used to represent the new symbol preceding with code specifying the path from the root to the zero leaf node then the codes for the above symbols would be as in the followings:

a -> a,

b -> 0 b

r -> 00 r

a -> 0

c -> 100 c

a -> 0

d -> 1100 d

a -> 0

b -> 110

r -> 110

a -> 0

! -> 1000 !

Here the character itself is shown instead of the ASCII bit streams.

3.4 Optimum Dynamic Huffman Coding

Optimum Dynamic Huffman coding is a one pass Huffman algorithm designed and analyzed by Vitter[79-80]. He called it a Algorithm Λ. Binary tree produced by Huffman's algorithm minimizes the weighted external path length $\Sigma_j w_j l_j$ among all binary trees, but the binary Huffman tree produced by the this algorithm also minimizes $\Sigma_j l_j$ and $\max_j\{l_j\}$ so produces optimal coding uses fewer bits optimal coding possible in any one pass algorithm.

3.4.1 Types of Node Interchanges

During the update operation of the tree in FGK algorithm, the current node is to interchange with another node in the tree. There are many types of interchanges possible and identified as in the following:

↑: Interchange in which the current node moves up one level in the tree.

↓: Interchange in which the current node moves down one level in the tree.

→ : Interchange is in the same level.

↑↑: Interchange of the current node with another node that is two levels up in the tree.

Interchanges of type ← in which the current node is to move left in the same levels and of type ↓↓ in which the current node is to move down two level is not possible in dynamic Huffman coding. In Fig.3.9 node 8 and 9 are of type ↑, whereas that of nodes 4 and 5 are of type →

3.4.2 Motivating factors

There are two motivating factors to improve the dynamic Huffman coding as:

(1) The number of ↑'s should be bounded by some small number (in this case 1) during each call to tree update procedure.

(2) The dynamic Huffman tree should be constructed to minimize not only $\Sigma_j w_j l_j$, but also $\Sigma_j l_j$ and $\max_j\{l_j\}$, which intuitively has the effect of preventing a lengthy encoding of the next source symbol in the message.

3.4.3 Implicit Numbering

Implicit numbering is a method in which the nodes of the tree are numbered in increasing order by level. The nodes on one level are numbered lower than the nodes on the next higher level. Nodes on the same level are numbered in increasing order from left to right. In this scheme the node numbering corresponds to the visual representation of the tree.

The node numbering used by the Algorithm FGK does not always correspond to the implicit numbering. For example, the numbering of the nodes in Fig. 3.9 and 3.10 does agree with the implicit numbering, whereas the numbering in Fig. 3.12 is quite different.

With the implicit numbering, interchanges of type ↓ cannot occur. Also, if the node that moves up in an interchange of type ↑ is an internal node, then the node that moves down must be a leaf.

The above result is obvious from the definition of implicit numbering. Suppose that an interchange of type ↑ occurs between two internal nodes a and b, where a is the node that moves up one level. In the initial tree, since a and b are on different levels, it follows from the sibling property that both a and b must have two children each of exactly half their weight. During the previous execution of the loop in the tree update routine q is set to a's right child, which is the highest numbered node of its weight. But this contradicts the fact that b's children have the same weight and are numbered higher in the implicit numbering.

3.4.4 Invariant

The key to minimizing difference between coded message length in dynamic and static Huffman code is to make ↑'s impossible, except the first iteration of the while loop in the update routine. It can be done by the following invariant:

(3.3) *For each weight w, all leaves of weight w precede (in implicit numbering) all internal nodes of weight w.*

Any Huffman tree satisfying (3.3) also minimizes $\Sigma_j l_j$ and $\max_j\{l_j\}$[78].

If the invariant (3.3) is maintained, then interchanges of type ↑↑ are impossible, and the only possible interchanges of type ↑ must involve the moving up of a leaf.

This can be proved by contradiction. It established in the dynamic Huffman tree that no two nodes of the same weight can be two or more levels apart in the tree except the sibling of the 0-node. The effect of the invariant (3.3) is to allow consideration of the 0-node's sibling. Suppose p is the sibling of a 0-node with a weight w and another node of weight w two levels higher in the tree. By invariant, node q must be an internal node, since it follows p's parent, which also has weight w, in the implicit numbering, thus contradicting the sibling property. For the second assertion, suppose there is an interchange of type ↑ in which an internal node moves up one level. The node that moves down must be a leaf node. But this violates the invariant, since the leaf initially follows the internal node in the implicit numbering.

3.4.5 Maintaining Invariant

To maintain the invariant (3.3) the update procedure must keep separate blocks for internal and leaf nodes. Blocks are equivalence classes of nodes defined by v ≡ x if and only if nodes v and x have the same weight and are either both internal nodes or both leaves. The leader of a block is the highest numbered node in a block by the implicit numbering.

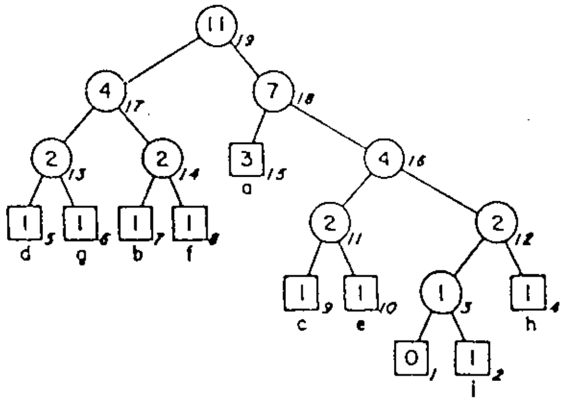


Fig.3.12: The Huffman tree formed by FGK algorithm after processing "**abcdefghiaa**".

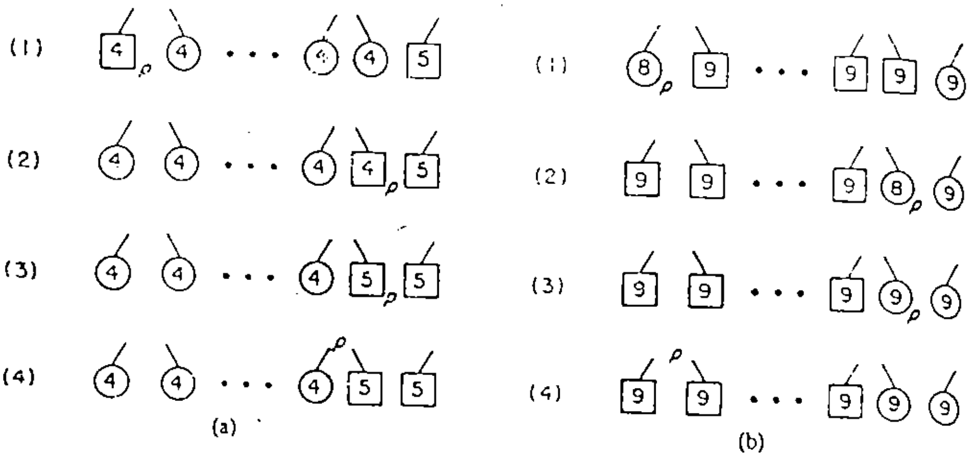

(a) (b)

Fig.3.13: Slide and increment operation of Vitter algorithm. All the nodes in a given block shift to the left one spot to make room for node **p**, which slides over the block to the right. (a) Node **p** is a leaf of weight 4. The internal nodes of weight 4 shift to the left. (b) Node **p** is an internal node of weight 8. The leaves of weight 9 shift to the left.

The blocks are linked together by increasing order of weight; a leaf block always precedes an internal block of the same weight. The main operations of the algorithm needed to maintain invariant (3.3) are the sliding and incrementing the current node. The possible situation is given in Fig.3.12.

3.5 Arithmetic Coding

Arithmetic coding is a statistical lossless data compression technique that encodes a source message by creating a code string which represents a fractional value on the number line between 0 and 1. It treats the code string as a magnitude which will be less than 1. The number of bits required to encode each symbol in a Huffman code is a whole number. But Shannon[65] showed that for best possible compressed code, in the sense of minimum average code length, the output length contains a contribution of $-\log_2 p$ bits from encoding of each symbol whose probability of occurrence is p. If there is an accurate model for probability of occurrence of each possible symbol at every point in a file, then arithmetic coding encodes the symbols that actually occurs; the number of bits used by arithmetic coding to encode a symbol with probability p is very nearly $-\log_2 p$, so the encoding is very nearly optimal for the given probability estimates. So arithmetic coding can be thought of as a generalization of Huffman coding in which probabilities are not constrained to be integral power of 2, and code lengths need not be integers. Arithmetic coding bypasses the idea of replacing an input symbol with a specific code. It replaces a stream of input symbols with a single floating-point number which is the merging of probabilities of symbols in the source message string. More bits are needed in the output number for longer, complex messages. This single fractional number is carefully constructed so that it may be uniquely decoded to create the exact stream of source symbols that went into its construction.

3.5.1 Initial View of Arithmetic Coding

Considering a four-symbol source alphabet S = {a, b, c, d} with their frequency of occurrence in a source message is W = {4, 2, 1, 1}. So the probability P = {1/2, 1/4, 1/8, 1/8}. The static binary Huffman tree is shown in the Fig.3.14. The source symbols in order of frequency of occurrence, with codeword is shown in the Table 3.8.

The encoding for the data string "a a b c" is 0 0 10 110. The codeword has a prefix property. Decoding is performed by a matching or comparison process starting with the first bit of the code string. For decoding code string 0010110, the first symbol is decoded as "a" as the code string begins with the codeword 0. The remaining code string is 010110 after removing the codeword 0. The second source symbol is similarly decoded as "a" leaving 10110, the only codeword starting with 10 is "b", so the code 110 is left for "c".

Table 3.8 Frequencies, Probabilities and Codewords for 4 symbols alphabets of an arbitrary message.

Symbol	Frequency Count	Probability		Codeword	Code Length	Cumulative Probability
		Decimal	Binary			
a	4	1/2	.1	0	1	.0
b	2	1/4	.01	10	2	.1
b	1	1/8	.001	110	3	.11
d	1	1/8	.001	111	3	.111

The Table 3.8 is described in terms of Huffman coding scheme. Arithmetic coding scheme can be viewed as a process of subdivision of the current interval with the aid of Table 3.8 and considering the following two points[47]:

Point 1: Each codeword (codepoint) is the sum of the probabilities of the preceding symbols.

Point 2: The width or size of the sub-interval to the right of each code point corresponds to the probability of the symbol.

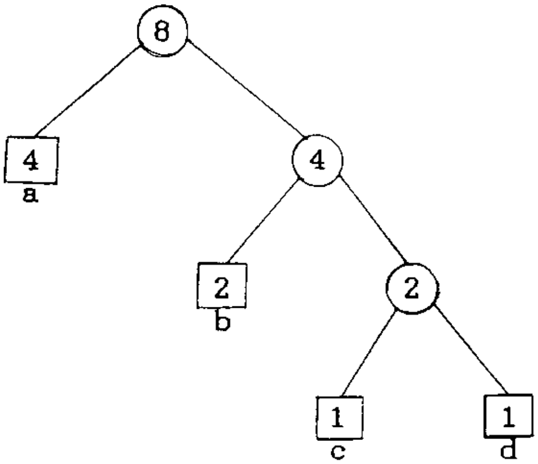
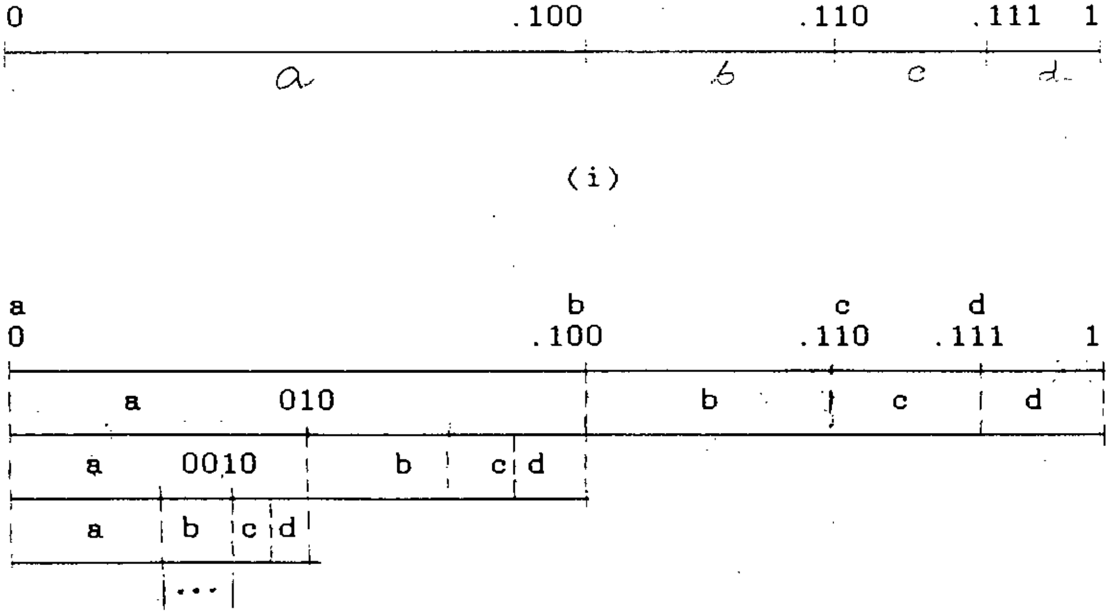


Fig.3.14 Static Huffman tree for source alphabet S = {a, b, c, d} with weight set W = {4, 2, 1, 1}.



(i)

(ii)

Fig.3.15 Code points of codewords of Table 3.8. (i) on unit interval, (ii) Successive subdivision of unit interval for data string "a a b ...".

From the Table 3.8, the codeword can be viewed as the binary fractional values (.0, .1, .11, .111) the these are the cumulative probabilities $P_i = \Sigma p_i$ of the source symbol s_i as stated in the point 1.

Now the codeword can be viewed as points i.e., code points on the number line from 0 to 1, or the unit interval, as shown in the Fig. 3.15. The four code points corresponding to the four source symbols divide the unit interval into four sub-intervals. Each sub-interval is identified by the symbol corresponding to its left most point. The interval for symbol "a" goes from .0 to .1, and for the symbol "c" goes from .11 to .111. The width of the subinterval to the right of each code point corresponds to the probability of the symbol as stated in the point 2. The codeword for the symbol "a". Initially the interval will be from 0 to 1 in the number line.

3.5.2 Basic Algorithm

The algorithm for encoding a source message using arithmetic coding works conceptually as follows:

1. The algorithm begins with a "current interval" [L,H) initialized to [0,1).

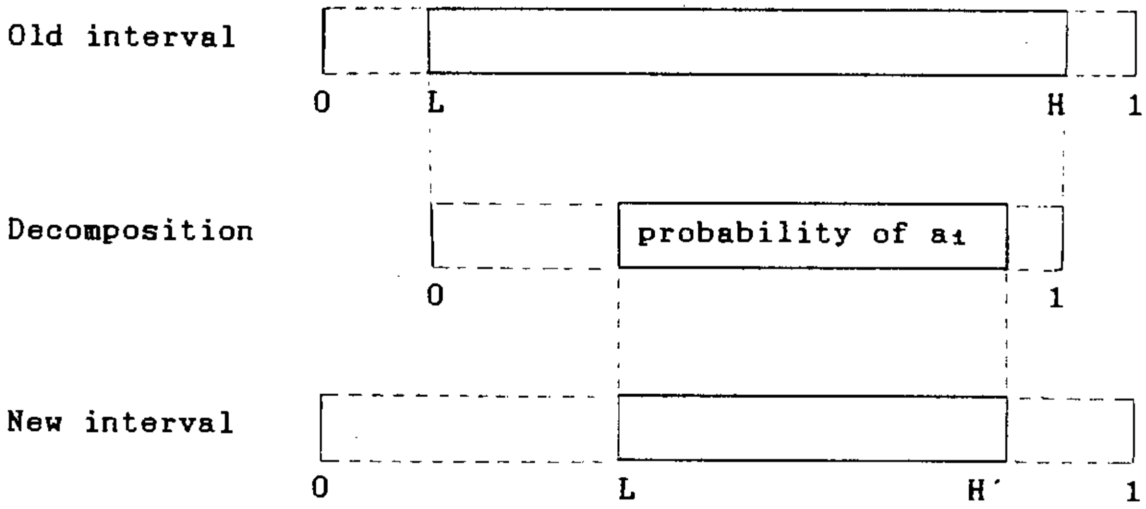


Fig.3.16 Subdivision of the current interval based on the probabilitiy of the input symbol a_i that occurs next.

2. For each symbol of the file, it performs two steps (Fig.3.16):

    (a) Subdivide the current interval into subintervals, one for each possible alphabet symbol. The size of a symbol's sub-interval is proportional to the estimated probability that the symbol will be the next symbol in the file, according to the model of the input.

    (b) Select the sub-interval corresponding to the symbol that actually occurs next in the file, and make it the new current interval.

3. Finally output enough bits to distinguish the final current interval from all other possible final intervals.

The length of the final subinterval is clearly equal to the product of the probabilities of the individual symbols, which is the probability p of the particular sequence of symbols in the source message. The final step uses almost exactly $-\lg_2(p)$ bits to distinguish the file from all other possible files. We need some mechanism to indicate the end of the file, either a special end-of-file symbol coded just once, or some external indication of the file's length.

In the step 2, the algorithm need to compute only the subinterval corresponding to the symbol a_i that actually occurs. To do this we need two cumulative probabilities,

$$P_C = \sum_{k=1}^{i-1} p_k \quad \text{and} \quad P_N = \sum_{k=1}^{i} p_k .$$

The new subinterval is $[L + P_C(H - L),\ L + P_N(H - L))$. The need to maintain and supply cumulative probabilities requires the model to have a complicated data structure; Moffat[61] investigates this problem, and concludes for a multi-symbol alphabet that move-to-front lists give a good balance between speed and simplicity.

3.5.3 Example of Arithmetic Coding

Here is an illustration of a non-adaptive arithmetic code, encoding the message "**CSE DEPTT.**". The message have a probability distribution given in Table 3.9.

Once the character probabilities are known, individual symbols need to be assigned a range along a probability line, normally 0 to 1. It does not matter which characters are assigned which segment of the range, as long as it is done in the same manner by both the encoder and decoder. The range of nine-character symbol set used in the example is also shown in Table 3.9.

Each character is assigned the portion of the 0 to 1 range that corresponds to its probability of appearance. The character owns everything up to but not including the higher number of the range. So the letter S in fact has the range 0.60 to 0.69999...

Table 3.9: Probability distribution and range of the symbols in message "CSE DEPTT."

Character	Probability	Range
space	1/10	[0.00, 0.10)
C	1/10	[0.10, 0.20)
D	1/10	[0.20, 0.30)
E	2/10	[0.30, 0.50)
P	1/10	[0.50, 0.60)
S	1/10	[0.60, 0.70)
T	2/10	[0.70, 0.90)
.	1/10	[0.90, 1.00)

The most significant portion of an arithmetic-coded message belongs to the first symbol C in the message **CSE DEPTT.**. To decode the first character properly, the final coded message has to be a number greater than or equal to 0.10 and less than 0.20. To encode this number, track the range it could fall in. After the first character is encoded, the low end for this range is 0.10 and the high end is 0.20. During the rest of the encoding process, each new symbol will further restrict the possible range of the output number. The next character to be encoded , the letter S, owns the range [0.60,0.70) in the new sub-range of [0.10, 0.20). So the new encoded number will fall somewhere in the 60th to 70th percentile of the currently established range. Applying this logic will further restrict our number to [0.16, 0.17). Following this process to its natural conclusion with the example message results in the Table 3.10.

So the final low value, .1630483504, will uniquely encode the message **CSE DEPTT.** using arithmetic coding scheme.

Given this encoding scheme, it is easy to see how the decoding process operates. Find the first symbol in the message by seeing which symbol owns the space our encoded message falls in. Since .1630483504 falls between .1 and .2, the first character must be C. Then remove the effect of C from the encoded number. Since we know the low and high ranges of C, remove their effects by reversing the process that put them in. First subtract the low value of C, giving .0630483504. Then divide by the width of the range of C ,i.e., .1. This gives a value of .630483504. Then calculate where that lands, which is in the range of the next letter, S. The decoding process for above encoded message is given in the Table 3.11.

Table 3.10 : Result of the Arithmetic coding of the message "CSE DEPTT."

New character	Low value	High value
	0.0	1.0
C	0.1	0.2
S	0.16	0.17
E	0.163	0.165
space	0.1630	0.1632
D	0.16304	0.16306
E	0.163046	0.163050
P	0.1630480	0.1630484
T	0.16304828	0.16304836
T	0.163048336	0.163048352
.	0.1630483504	0.1630483520

Table 3.11: Result of decoding process of the encoded message "CSE DEPTT."

Encoded number	Output character	Range
0.1630483504	C	[0.1, 0.2)
0.630483504	S	[0.6, 0.7)
0.30483504	E	[0.3, 0.5)
0.0241752	space	[0.0, 0.1)
0.241752	D	[0.2, 0.3)
0.41752	E	[0.3, 0.5)
0.5876	P	[0.5, 0.6)
0.876	T	[0.7, 0.9)
0.88	T	[0.7, 0.9)
0.9	.	[0.9, 1.0)
0.0		

In summary, the encoding process is simply one of narrowing the range of possible numbers with every new symbol. The new range is proportional to the predefined probability attached to that symbol. Decoding is the inverse procedure, in which the range is expanded proportionally to the probability of each symbol as it is extracted.

3.5.4 Implementation

The basic implementation of arithmetic coding described above has two major difficulties:

(i) The shrinking of the current interval requires the use of high precision arithmetic, and

(ii) No output is produced until the entire message has been read.

The most straight forward solution to both of these problems is to output each leading digit as soon it is known, and then multiply the length of the current interval by the radix of the base number, so that it reflects only unknown part of the final interval.

3.5.5 Incremental Transmission

Arithmetic coding is best accomplished using standard 16-bit and 32-bit integer math. Floating-point math is neither required nor helpful. Mechanisms for incremental transmission and a fixed precision arithmetic have been developed by Rubin[61-62], Rissanen and Langdon[60], and Guazzo[25]. In this mechanism, fixed-size integer state variables receive new digit at the lower end and shift them out at the high end, forming a single number that can be as long as the number of bits on the computer.

Earlier, we saw that the algorithm works by keeping track of a high and low number that brackets the range of the possible output number. When the algorithm first starts, the low value is set to .0 and the high value is set to 1. The first simplification made to work with integer math is to change the 1 to .9999... (or .1111... in binary). To store these numbers in integer registers, first justify them so the implied decimal point is on the left side of the word. Then load as much of the initial high and low values as will fit into the word size we are working with. Implementation with 16-bit unsigned math, the initial low value and high value will be 0 and 0xFFFF respectively. The high value continues with 0xFs and low value with 0s forever. For the previous example,

HIGH: 99999 implied digits => 9999999...

LOW : 00000 implied digits => 0000000...

The initial range between the low and high will be 100000, not 99999. This is because we assume the high has an infinite number of 9s added to it, so we need to increment the calculated difference. We then compute the new low and high values following the previous techniques. In the example, the new high value was 0.20, which gives a new value for high of 20000. Before storing the new value of high, we need to decrease it, again because of the implied digits appended to the integer value. So the new value of high is 19999. The calculation of the low value follows the same procedure, with a resulting new value of 10000. So the new high and low would be:

HIGH: 19999 (19999...)
LOW: 10000 (10000...)

At this point, the most significant digits of high and low match. Due to the nature of the algorithm, high and low can continue to grow closer together without quite ever matching. So once they match in the most significant digit, that digit will never change. So the that digit can output as first digit the encoded number. This is done by shifting both high and low left by one digit and shifting in a 9 in the least significant digit of high. The encoding process for the given message of the previous example is shown in Table 3.12.

After all the symbols are accounted for, two extra digits need to be shifted out of either from the high or low value to finish the output word. This is so that the decoder can properly track the input data. The final result of encoding of the message is .1630483504 by considering the next two digits from low value register. Part of the information about the data stream is still in the high and low registers, and that values should be passed to the decoder to use later.

Table 3.12: Arithmetic Encoding process of the message "CSE DEPTT." using incremental transmission.

Current values			Input symbol with interval	New values		Cumulative Output
High	Low	Range		High	low	
99999	00000	100000	C [0.1, 0.2)	19999	10000	.1
99999	00000	100000	S [0.6, 0.7)	69999	60000	.16
99999	00000	100000	E [0.3, 0.5)	49999	30000	
49999	30000	20000	sp [0.0, 0.1)	31999	30000	.163
19999	00000	20000	D [0.2, 0.3)	05999	04000	.1630
59999	40000	20000	E [0.3, 0.5)	49999	46000	.16304
99999	60000	40000	P [0.5, 0.6)	83999	80000	.163048
39999	00000	40000	T [0.7, 0.9)	35999	28000	
35999	28000	8000	T [0.7, 0.9)	35199	33600	.1630483
51999	36000	16000	. [0.9, 1.0)	51999	50400	.16304835

3.5.6 Under Flow Problem

Incremental transmission scheme works well in arithmetic coding. Enough accuracy is retained during the double-precision inter calculations to ensure that the message is accurately encoded. But there is potential for a loss of precision under certain circumstances. If the encoded word has a string of 0s or 9s in it, the high and low values will slowly converge on a value, but they may not see their most significant digits match immediately. High may be 700004 and low may be 699995. At this point, the calculated range will be only a single digit long, which means the output word will not have enough precision to be accurately encoded. Worse, after a few more iterations, high could be 70000 and low could be 69999.

At this point, the values are permanently stuck. The range between high and low has become so small that any iteration through another symbol will leave high and low at their same values. But since the most significant digits of both high and low are not equal, the algorithm cannot output the digit and shift. To overcome this underflow problem the algorithm should modify as

```
If the most significant digits of low and high match then
    shift it out
else  if the high and low are one apart and second most
significant digit of low is 0 and that of high is 9 then
            there is an underflow and take action.
```

When an underflow occurs, instead of shifting the most significant digit out of the word, the algorithm delete the second digits from high and low and shifts the rest of the digits left to fill the space. The most significant digit stays in place. Then it sets underflow counter to remember number of digits deleted. This process is shown bellow:

	Before	After
HIGH :	40344	43449
LOW :	39810	38100
Underflow:	0	1

After every recalculation, the algorithm checks for underflow digits again if the most significant digits do not match. If underflow digits are present, the algorithm shifts them out and increment the counter. When the most significant digits do finally converge to a single value, it outputs the value, and then outputs the underflow digits previously discarded. The underflow digit will all be 9s or 0s, depending on whether high and low converged on the higher or lower value.

3.5.7 Use of Integer Arithmetic

In practice, the arithmetic coding can be done by storing the current interval in sufficiently long integers rather than in floating point. Also frequency counts can be used to estimate symbol probabilities. The subdivision process involves selecting non overlapping intervals of at least 1 with lengths appropriately proportional to the total counts. To encode symbol a_i we need two cumulative counts C and N, and the sum T for all counts r as in the following:

$$C = \sum_{k=1}^{i-1} c_k , \quad N = \sum_{k=1}^{i} c_k \text{ and } T = \sum_{k=1}^{n} c_k ,$$

where n is the alphabet size. And

$$\left[L + \left\lfloor \frac{C(H-L)}{T} \right\rfloor,\ L + \left\lfloor \frac{N(H-L)}{T} \right\rfloor\right)$$

would be the new subinterval for integer arithmetic coding scheme.

Chapter Four

BANGLA TEXT ANALYSIS

The mother tongue of over 200 million people of Bangladesh, West Bengal and some parts of Assam is Bangla. After the independence of Bangladesh it has been decided to use Bangla as the official language and medium of instruction for schools, college and universities. In modern times transmission of information is totally dependent on modern technology. Specially without the use of telex, teleprinter, fax and computer no nation can advance. To use Bangla in all spheres of life it is important to be able to adapt Bangla for use in those technologies. The first Bangla typewriter 'Remington' was introduced to the market as early as in 1940's. Later Shahid Munir Chowdhury recommended some structural changes in the Bangla typewriter under the patronage of Bangla Unnayan Board. 'Munir Optima' came to the market in 1970's based on those recommendations. Although it was much more improved than the Remington typewriter Bangla Academy took a project on developing an electronic typewriter to remove various limitations. In the final report of the project it was recommended to use 138 key three layer keyboard with absolutely different layout. Afterwards, to modernize typewriter and use Bangla in computer National Computer

Committee, later renamed as National Computer Board, and other government organizations took different steps.

Parallel to government efforts various commercial organizations came forward to use Bangla in computers and designed their own keyboard layout. As a result several Bangla word processors like Onirbaan, Barna, Shahid Lipi, Basundhara, etc. were developed. To use Bangla in computers, typewriters and teleprinters, it was felt that standardization of both the keyboard layout and codes is important. With this aim Rahman[58] and later Khan[43] proposed a 131-key keyboard.

Considering the importance of standardization a task force consisting of 5 members headed by the Honorable Vice Chancellor of the Bangladesh University of Engineering and Technology was formed which very recently approved standardization of both the keyboard and codes of letters.

Now-a-days Bangla is being used in Desk Top Publishing in mass scale. This language is being used for transmission of information from one town to another using computers. In these applications for compound letters multibyte representation is being used, and as a result both the length of the transmitted code and redundancy have increased. Under these circumstances it has become very important to be able to compress Bangla text for improving efficiency of both storing and transmitting Bangla texts.

4.1 Character Frequencies of Bangla Text

For designing keyboard layout and code for Bangla character set, several statistics of the characters have been made. Khan[43], in 1986, made a survey of letter frequencies based on 16,090 no, of occurrence. Mr. Khan has also presented the survey report on the frequency of Bangla characters done by Das[12] and another survey of Das[13] with some correction of the frequency of space using Poison distribution as these surveys did not include space character. Another survey was made by Kuraisi[46] in 1988 and in his report he has given frequency of Bangla characters done by Saheed Munir Chowdhuri Bangla Unnayan Project (1965), Jamil Chowdhuri (1987), and Monoj Kumar Mitra (see, [46]). During the present work, another survey has been made on the frequency of occurrence of Bangla characters and Bangla akkharas based on number of occurrence from representative texts of various disciplines. These frequency distributions of occurrence of Bangla characters in single byte representation and akkharas in multibyte representation have been given in the Table 4.1 and 4.2 respectively.

Table 4.1(a): Frequency of Bangla characters in single byte representation in dictionary order.
Total characters 25024 and unique character 100.

Symbol Number	Symbol	Frequency		Symbol Number	Symbol	Frequency	
32	Space	3792	15.153	105	ষ	73	0.292
33	!	31	0.124	106	স	489	1.954
39	'	89	0.356	107	হ	913	3.649
44	,	264	1.055	108	ড়	13	0.052
45	-	73	0.292	109	ঢ়	5	0.020
46	.	2	0.008	110	য়	386	1.543
48	০	1	0.004	111	ৎ	27	0.108
49	১	1	0.004	112	ং	181	0.723
50	২	1	0.004	113	ঃ	8	0.032
52	৪	1	0.004	114	ঁ	23	0.092
54	৬	1	0.004	121	।	229	0.915
55	৭	1	0.004	127	[illegible]	113	0.452
56	৮	1	0.004	194	া	2826	11.293
59	;	29	0.116	195	ি	1274	5.091
63	?	31	0.124	196	ী	210	0.839
65	অ	163	0.651	197	ু	244	0.975
66	আ	331	1.323	198	ূ	93	0.372
67	ই	340	1.359	199	ৃ	88	0.352
68	ঈ	31	0.124	200	ে	1248	4.987
69	উ	74	0.296	201	ৈ	3	0.012
70	ঊ	2	0.008	202	ো	282	1.127
72	এ	178	0.711	203	ৌ	2	0.008
74	ও	176	0.703	204	[illegible]	15	0.060
76	ক	1083	4.328	205	[illegible]	1	0.004
77	খ	138	0.551	206	[illegible]	21	0.084
78	গ	157	0.627	207	[illegible]	1	0.004
79	ঘ	22	0.088	209	[illegible]	21	0.084
80	ঙ	21	0.084	210	[illegible]	15	0.060
81	চ	51	0.204	211	[illegible]	10	0.040
82	ছ	207	0.827	213	[illegible]	16	0.064
83	জ	158	0.631	214	[illegible]	46	0.184
84	ঝ	5	0.020	215	[illegible]	4	0.018
65	ঞ	4	0.016	216	[illegible]	4	0.016
86	ট	51	0.204	218	[illegible]	19	0.076
87	ঠ	9	0.036	219	[illegible]	133	0.531
90	ণ	122	0.486	220	[illegible]	12	0.048
91	ত	1168	4.660	221	[illegible]	27	0.108
92	থ	72	0.288	222	[illegible]	21	0.084
93	দ	511	2.042	223	[illegible]	25	0.100
94	ধ	68	0.272	224	[illegible]	33	0.132
95	ন	934	3.732	228	[illegible]	116	0.464
96	প	441	1.762	227	[illegible]	6	0.024
97	ফ	46	0.184	228	[illegible]	46	0.184
98	ব	699	2.793	229	[illegible]	218	0.871
99	ভ	67	0.268	230	[illegible]	225	0.899
100	ম	674	2.693	231	[illegible]	93	0.372
101	য	260	1.039	232	[illegible]	14	0.058
102	র	1628	6.506	233	[illegible]	62	0.248
103	ল	529	2.114	235	[illegible]	144	0.575
104	শ	208	0.831	252	[illegible]	2	0.008

Table 4.1(b): Frequency of Bangla characters in single byte representation in decending order of frequency. Total characters 25024 and unique character 100.

Symbol Number	Symbol	Frequency		Symbol Number	Symbol	Frequency	
32	Space	3792	15.153	233	্ল	62	0.248
194	্য	2826	11.293	81	চ	51	0.204
102	র	1626	6.506	86	ট	51	0.204
195	ি	1274	5.091	228	্ন	46	0.184
200	ে	1246	4.987	97	ফ	46	0.184
91	ত	1166	4.660	214	ৃ	46	0.184
76	ক	1083	4.328	224	্ক	33	0.132
95	ন	934	3.732	88	ক্ষ	31	0.124
107	ই	913	3.649	83	?	31	0.124
98	ব	699	2.793	33	!	31	0.124
100	এ	674	2.693	59	:	29	0.116
103	ল	529	2.114	111	ঃ	27	0.108
93	দ	511	2.042	221	্	27	0.108
106	প	489	1.954	223	ু	25	0.100
96	ম	441	1.762	114	ঁ	23	0.092
110	স	386	1.543	79	ঘ	22	0.088
67	ই	340	1.359	206	্ণ	21	0.084
66	আ	331	1.323	209	ঁ	21	0.084
202	ো	282	1.127	222	ৃ	21	0.084
44	,	264	1.055	80	ঙ	21	0.084
101	য	260	1.039	218	্	19	0.076
197	∝	244	0.975	213	্দ	16	0.064
121	।	229	0.915	210	্ন	15	0.060
230	ু	225	0.899	204	্ধ	15	0.060
229	ূ	218	0.871	232	্য	14	0.056
198	ী	210	0.839	108	ড	13	0.052
104	শ	208	0.831	220	্স	12	0.048
82	জ	207	0.827	211	্ড	10	0.040
112	ং	181	0.723	87	ঞ	9	0.036
72	এ	178	0.711	113	ঃ	8	0.032
74	ও	176	0.703	227	ৃ	6	0.024
65	অ	163	0.651	84	ঝ	5	0.020
83	হ	158	0.631	109	ঢ	5	0.020
78	গ	157	0.627	215	্	4	0.016
235	ষ	144	0.575	216	্ত	4	0.016
77	খ	138	0.551	85	ঞ	4	0.016
219	ৃ	133	0.531	201	(	3	0.012
90	ণ	122	0.488	70	ঊ	2	0.008
226	্র	116	0.464	203	ৌ	2	0.008
127	ৃ	113	0.452	46	.	2	0.008
231	র্	93	0.372	252	/	2	0.008
186	ৈ	93	0.372	205	্য	1	0.004
39	'	89	0.356	54	৬	1	0.004
199	<	88	0.352	207	্দ	1	0.004
69	উ	74	0.296	55	৭	1	0.004
45	-	73	0.292	56	৮	1	0.004
105	ষ	73	0.292	48	০	1	0.004
92	য	72	0.288	49	১	1	0.004
94	থ	68	0.272	50	২	1	0.004
99	ভ	67	0.268	52	৪	1	0.004

Table 4.2: Frequency of Bangla characters in multi-byte representation in decending decending order of frequency. Total characters 16534 and unique character 407.

	Frequency			Frequency			Frequency	
Akkhara	Count	(%)	Akkhara	Count	(%)	Akkhara	Count	(%)
	3790	22.922	লি	163	0.986	তু	72	0.435
র	742	4.488	লে	161	0.974	যো	70	0.423
ক	514	3.109	য	135	0.816	ষ্ট	70	0.423
হা	509	3.079	পা	126	0.762	মূ	69	0.417
ন	397	2.401	উ	126	0.762	চ	68	0.411
রা	361	2.183	ছে	121	0.732	হী	66	0.399
ব	355	2.147	বা	119	0.720	খ	65	0.393
ই	340	2.056	শ	116	0.702	ধা	56	0.339
আ	331	2.002	দো	111	0.671	ষ্ক	52	0.315
তা	326	1.972	জো	110	0.665	ন্ড	51	0.308
ম	286	1.730	লা	104	0.629	সি	50	0.302
ত	274	1.657	মি	100	0.605	ত্র	44	0.266
যা	245	1.482	ন্য	97	0.587	স্ম	40	0.242
রি	217	1.312	সা	97	0.587	বী	39	0.236
মা	217	1.312	সে	94	0.569	ম্ব	37	0.224
দে	217	1.312	যে	89	0.538	দু	34	0.206
হ	194	1.173	হি	88	0.532	ধ	32	0.194
কে	193	1.167	ট	85	0.514	শু	32	0.194
স	190	1.149	কো	84	0.508	দী	32	0.194
তো	189	1.143	জ্ঞা	82	0.496	হু	31	0.187
ং	181	1.095	হে	81	0.490	পূ	31	0.187
নি	179	1.083	দা	77	0.466	চা	29	0.175
এ	178	1.077	স্ত্রী	77	0.466	মী	28	0.169
ল	177	1.071	নু	75	0.454	জ্ঞী	28	0.169
ও	176	1.064	মু	74	0.446	খা	28	0.169

Table 4.2 (Continued).

Akkhara	Frequency Count	(%)	Akkhara	Frequency Count	(%)	Akkhara	Frequency Count	(%)
র্ণ	27	0.163	ষ্ট	17	0.103	'া	12	0.073
ষ	27	0.163	বৃ	17	0.103	হ্	12	0.073
না	26	0.157	ভী	17	0.103	হ্ন	12	0.073
কা	26	0.157	র্ব	17	0.103	পৃ	11	0.067
ভে	25	0.151	ত্য	16	0.097	হ্ম	11	0.067
দি	25	0.151	ৎ	16	0.097	স্থা	11	0.067
ঘা	25	0.151	শা	16	0.097	শি	11	0.067
বে	24	0.145	মূ	15	0.091	ঙ্গা	11	0.067
প্র	23	0.139	ক্ষ	15	0.091	ঝ	11	0.067
য	23	0.139	নে	15	0.091	ষি	10	0.060
র্ষ	23	0.139	ত্রে	14	0.085	লে	10	0.060
ণ	22	0.133	ভা	14	0.085	থি	10	0.060
ছি	22	0.133	শে	14	0.085	স্ম	10	0.060
হ্ন	21	0.127	গ্র	14	0.085	ষ্	10	0.060
ডি	21	0.127	ফি	13	0.079	কৃ	10	0.060
ন্দা	21	0.127	ব্য	13	0.079	ক্ষে	10	0.060
কি	20	0.121	ত্যা	13	0.079	ষ্ঠি	10	0.060
বি	20	0.121	ডো	13	0.079	প্রা	9	0.054
র্ব	20	0.121	শ্যা	12	0.073	ঘ	9	0.054
রে	20	0.121	ক্ষা	12	0.073	দ্বা	9	0.054
জ্ঞা	19	0.115	দ্	12	0.073	চ্	9	0.054
ষ্টি	19	0.115	চ্যা	12	0.073	ষ্ট	9	0.054
মে	19	0.115	বু	12	0.073	ম্র	9	0.054
লো	18	0.109	দ্বি	12	0.073	হৃ	9	0.054
শী	18	0.109	নো	12	0.073	গ্রী	9	0.054
শে	18	0.109	ব্রে	12	0.073	ত্মে	9	0.054
শ্রী	17	0.103	নী	12	0.073	ত্রু	9	0.054
গো	17	0.103	হ্ন	12	0.073	ত্র	9	0.054

Table 4.2 (Continued).

Akkhara	Frequency Count	(%)	Akkhara	Frequency Count	(%)	Akkhara	Frequency Count	(%)
ন্ড	9	0.054	টি	6	0.036	ষ্ট	5	0.030
বূ	9	0.054	প্রে	6	0.036	ত্রে	5	0.030
ন্ধ	9	0.054	ট্টি	6	0.036	স্তে	4	0.024
ষ্কি	8	0.048	দূ	6	0.036	ণ্ড	4	0.024
ক্ত	8	0.048	হৃ	6	0.036	ন্ধূ	4	0.024
ষ্ঠ	8	0.048	জি	6	0.036	ঙ্ক	4	0.024
নূ	8	0.048	ন্ত	6	0.036	স্য	4	0.024
ন্নি	8	0.048	ন্দূ	6	0.036	স্থ	4	0.024
ক্তি	8	0.048	ঘ	6	0.036	স্তা	4	0.024
মৃ	8	0.048	ঘো	6	0.036	ল্য	4	0.024
থ্যে	8	0.048	ন্যা	6	0.036	ঝা	4	0.024
•	8	0.048	ম্ব	6	0.036	দূ	4	0.024
থে	8	0.048	ক্ষে	6	0.036	ধ্ব	4	0.024
ধূ	8	0.048	ম্ম	6	0.036	ন্তা	4	0.024
ঝ	8	0.048	চ	6	0.036	খূ	4	0.024
হূ	7	0.042	ণা	5	0.030	ড়	4	0.024
দৃ	7	0.042	ন্টা	5	0.030	ন্ধে	4	0.024
সৃ	7	0.042	ত্ম	5	0.030	ন্ম	3	0.018
লে	7	0.042	ক্ক	5	0.030	দ্বে	3	0.018
ধা	7	0.042	হ্য	5	0.030	ষে	3	0.018
ঘা	7	0.042	ত্তি	5	0.030	ক্ষু	3	0.018
ষ্য	7	0.042	কী	5	0.030	ষ্ক	3	0.018
গৃ	7	0.042	ন্দ	5	0.030	ভ	3	0.018
শূ	7	0.042	ত্তা	5	0.030	ঞ্জা	3	0.018
ষ্টা	7	0.042	ন	5	0.030	ঙ্গা	3	0.018
বৃ	7	0.042	ম	5	0.030	গে	3	0.018
পূ	7	0.042	ক্ষি	5	0.030	অ্ন	3	0.018
ক্ষ	6	0.036	শ	5	0.030	ম্ম	3	0.018

Table 4.2 (Continued).

Akkhara	Frequency Count	(%)	Akkhara	Frequency Count	(%)	Akkhara	Frequency Count	(%)
ঃ	3	0.018	ঘী	2	0.012	ত্ত	2	0.012
শী	3	0.018	ঝি	2	0.012	ম্ম	2	0.012
ড়া	3	0.018	ঞ্জি	2	0.012	ঔ	2	0.012
স্ত্রী	3	0.018	ন্দে	2	0.012	ষি	2	0.012
শৃ	3	0.018	খ্যা	2	0.012	টা	2	0.012
মৃ	3	0.018	জ্ঞ	2	0.012	ক্ষু	2	0.012
ক্স	3	0.018	ণ্ড	2	0.012	খে	2	0.012
জে	3	0.018	ত্তু	2	0.012	ন্টি	2	0.012
ন্ধা	3	0.018	খে	2	0.012	ত্যো	2	0.012
ল্কি	3	0.018	শে	2	0.012	স্ম	2	0.012
ঘা	3	0.018	ব্যো	2	0.012	ভূ	2	0.012
ল্যা	3	0.018	ম্রি	2	0.012	ক্ক	2	0.012
ঙ্কে	3	0.018	মু	2	0.012	স্না	2	0.012
ঝা	3	0.018	ঙ্কে	2	0.012	ষ্ক	2	0.012
চি	3	0.018	ত্রে	2	0.012	দ্বা	2	0.012
ন্দ	3	0.018	বৈ	2	0.012	স্ক	2	0.012
শ্র	3	0.018	শ্চা	2	0.012	স্রি	2	0.012
ম্মা	3	0.018	পী	2	0.012	ঘৃ	2	0.012
ঊ	3	0.018	ঙ্ক	2	0.012	লি	1	0.006
ন্ধে	3	0.018	ন্দি	2	0.012	ম্ব	1	0.008
যা	3	0.018	শ্রী	2	0.012	ণ্ব	1	0.006
ভৃ	3	0.018	ষ্ক	2	0.012	স্নি	1	0.006
খ্য	3	0.018	ঙ্খি	2	0.012	ক্কা	1	0.006
লি	3	0.018	খ্যে	2	0.012	ষ্কা	1	0.006
পো	3	0.018	ঞ্চি	2	0.012	দ্দি	1	0.006
ঙ্কি	2	0.012	জী	2	0.012	হু	1	0.006
ম্মু	2	0.012	ঝা	2	0.012	ক্ষী	1	0.006
ক্কি	2	0.012	ক্মি	2	0.012	ঙ্ক	1	0.006

Table 4.2 (Continued).

Akkhara	Frequency Count	(%)	Akkhara	Frequency Count	(%)	Akkhara	Frequency Count	(%)
স্বা	1	0.006	ঝে	1	0.006	হূ	1	0.006
হ্ব	1	0.006	ক্ষা	1	0.006	বো	1	0.006
ব্রে	1	0.006	ক্ত	1	0.006	দ্যে	1	0.006
ন্ধা	1	0.006	ন্ধ	1	0.006	ত্রী	1	0.006
ক্ষি	1	0.006	সূ	1	0.006	ত্রি	1	0.006
য	1	0.006	শ্রী	1	0.006	জ্জ	1	0.006
ঙ্ক	1	0.006	ঙ্ক	1	0.006	হ্যা	1	0.006
ন্দা	1	0.006	চী	1	0.006	ক্ষে	1	0.006
শূ	1	0.006	ক্য	1	0.006	ক্ষি	1	0.006
দ্বি	1	0.006	ঙে	1	0.006	ন্দী	1	0.006
ক্ষে	1	0.006	ব্ধ	1	0.006	ঘু	1	0.006
ক্যা	1	0.006	শ্য	1	0.006	ব্রি	1	0.006
ক্তা	1	0.006	ক্ষা	1	0.006	ঘ্যা	1	0.006
স্র	1	0.006	দ্ধ	1	0.006	ক্ষী	1	0.006
হু	1	0.006	স্ব	1	0.006	ঝা	1	0.006
ব্ব	1	0.006	জী	1	0.006	ষূ	1	0.006
ঝ	1	0.006	চৃ	1	0.006	ঞু	1	0.006
ঙ্খ	1	0.006	স্মা	1	0.006	চে	1	0.006
ঠা	1	0.006	হী	1	0.006	স্ফ	1	0.006
ট্টা	1	0.006	মো	1	0.006	ড্রি	1	0.006
দ্দা	1	0.006	ঠ	1	0.006	দ্য	1	0.006
ধৈ	1	0.006	জে	1	0.006	ড্রে	1	0.006
শ্য	1	0.006	শ্তে	1	0.006	ষ্ট	1	0.006
ম্পৃ	1	0.006	ম্মূ	1	0.006	ঝ	1	0.006
ণী	1	0.006	ত্র	1	0.006	দ	1	0.006
ঢী	1	0.006	ষ্ক	1	0.006	স্রো	1	0.006
শা	1	0.006	জ্ঞা	1	0.006			

4.2 Redundancy in Bangla Text

Natural language is highly redundant and it would, therefore, be beneficial if some of this redundancy could be removed. Most data compression techniques are based on information theoretic concepts and take advantages of the statistical properties of natural language.

Symbols of Bangla alphabet are stored in secondary storage and processed in computer as pattern of binary digits. Redundancy exists when portion of these patterns are predictable and, therefore, carry little or no information. Redundancy typically exists in one of the following forms:

- Symbols have widely different probabilities, and
- Strong inter symbol influence exists over adjacent symbols.

Most of the word processors and DTP use multibyte representation of some or all compound symbols of the Bangla alphabet. So for the existence of a group of bytes for a single symbol, Bangla text is more redundant than any other language in which single byte representation is used for their symbols. Inter symbol influence is not only over the adjacent symbols but also over the whole group of symbols.

Suppose we imagine that a word of Bangla text is being transmitted through digital communication line. There is a choice of 46 symbols for the first letter providing about 5 bits; suppose it was a ঢ (89). The choice for the next letter is restricted, because of the construction of the Bangla language, to about 15 letters, providing about 4 bits; assume it was an া (194). There is only 5 choice for the next letter contributing 2 bits; say the letter is ক (76). Now the word could reasonably guessed to be say ঢাকা, ঢাকাই or ঢাকী The last few letters clearly provide very little information, because the probability of the next letters after the sequence ঢাক approaches unity.

Similarly, words are not equiprobable and intersymbol influence extends over groups of words, phrases, sentences etc. and as a result the information content of the language is much less than the ideal value for equiprobable and independent symbols.

4.3 n-Gram Statistics of Bangla Text

An n-gram is a string of n symbols occurring sufficiently frequently in a text to justify its being considered as a symbol in its own right in addition to the conventional symbols that comprise text. We have taken 26368-byte text as a representative text in BSCII format to generate a list of frequencies of all strings up to 8-character long. Multiple occurrences of spaces being reduced to a single occurrence. The symbols considered were all symbols except punctuation marks. Average length of the sentence and words are found from the representative text is given in the Table 4.3.

Yannakoudakis[83] give the n-gram statistics of the title from 31369 records in a 1975 British National Bibliography file. Suen[70] gives n-gram statistics from English words. But no such statistics is found for Bangla Text. A comparison between the top 25 bigrams to octagrams derived from the present study with single-byte and multibyte representation of the symbol is given in the Table 4.4 and 4.5 respectively. It can be shown from the Fig. 4.1 that as the length of the n-grams increases the maximum n-gram frequency decreases.

By assuming that text of any language can be generated by n-grams of symbols from an alphabet of K symbol types of that language, Shannon[66] estimated the nth order entropy as

$$H^n = -\sum_{j=1}^{K^n} p(x_j^n) \log_2 p(x_j^n)$$

where $p(x_j^n)$ is the probability of the jth n-gram. If we write the entropy of the nth symbol as H_n then Shannon has shown that $H_n = H^n - H^{(n-1)}$.

Shannon also defined the nth order redundancy as

$$R_n = 1 - \frac{H_n}{\log_2 K}$$

Table 4.3: Statistics of the representative Bangla BSCII file. Total characters : 25410, Total words : 4238 and Total sentences : 291.

(a) Words Size in Characters : (Average 4.94 characters/word).

Character per word	Number of words	Character per word	Number of words	Character per words	Number of words
1	138	7	440	13	5
2	302	8	194	14	2
3	754	9	150	15	2
4	813	10	83	16	2
5	648	11	27	17	1
6	660	12	17		

(b) Sentences Size in words:(Average 14.56 words/sentence).

Words per sentence	Number of sentence	Words per sentence	Number of sentence	Words per sentence	Number of sentences	Words per sentence	Number of sentences
1	1	12	12	23	7	36	3
2	7	13	16	24	3	37	3
3	6	14	19	25	2	39	2
4	7	15	11	26	7	40	1
5	23	16	7	27	4	42	1
6	16	17	12	28	2	46	1
7	14	18	7	31	1	47	1
8	14	19	7	32	1	48	1
9	23	20	7	33	4	53	1
10	11	21	3	34	1	63	1
11	13	22	6	35	1	74	1

(c) Sentences Size in Characters:
(Average 85.54 characters/sentence).

Characters per sentence	Number of sentence	Characters per sentence	Number of sentence	Characters per sentence	Number of sentences	Characters per sentence	Number of sentences
7	1	51	6	88	1	145	1
8	3	52	2	90	4	147	1
11	1	53	2	92	6	150	1
12	2	54	2	93	2	152	1
14	2	55	1	94	1	153	1
16	2	56	3	95	1	155	1
19	2	57	3	97	1	157	1
20	3	58	4	98	2	158	1
21	3	59	4	99	3	161	2
22	2	60	1	101	1	162	1
23	3	61	5	102	3	169	1
24	3	62	2	103	1	171	1
26	1	65	4	104	2	172	1
27	3	66	4	105	2	177	1
29	5	67	3	106	3	185	2
30	5	68	5	108	2	188	1
31	2	69	2	110	2	190	2
32	8	70	2	112	1	201	1
33	2	71	2	113	1	208	1
34	1	72	1	114	1	212	1
35	2	73	3	115	1	216	1
37	3	74	4	116	3	221	1
38	1	75	3	119	2	223	1
39	4	76	3	121	2	230	1
40	2	77	3	124	2	231	1
41	3	78	1	127	1	236	1
42	3	79	4	128	1	254	1
43	6	80	3	132	2	261	1
44	2	81	1	133	1	286	1
45	1	82	3	135	2	297	1
46	4	83	1	136	1	299	1
47	2	84	2	137	1	368	1
48	1	85	1	142	2	370	1
49	1	86	2	143	2	465	1
50	2	87	3	144	5		

Table 4.6 and Graph 4.1 show natural characteristics of Bangla text as derived from the sample text.

Table 4.4(a): Percentage of occurrence of 25 1-5 grams for Bangla text with single byte representation.

1-grams		2-grams		3-grams		4-grams		5-grams	
	18.58	া=	3.60	াহা	3.28	াহার	1.88	াহারা	1.81
া	13.85	ে=	3.46	=কর	2.22	=করি	1.50	হারা=	1.63
র	7.98	াহ	3.33	রা=	2.00	হারা	1.49	=যাহা	1.12
ি	6.24	র=	3.21	রে=	1.98	ারা=	1.40	আল্লাহ	1.11
ে	6.12	হা	3.15	হার	1.72	িয়াছ	1.24	ল্লাহ=	1.10
ক	5.31	=ক	3.09	করি	1.40	এবং=	1.09	=এবং=	1.07
হ	4.48	ার	2.43	ারা	1.30	যাহা	1.08	করিয়া	0.96
ব	3.43	রা	2.24	িয়া	1.28	মরা=	1.04	িয়াছি	0.88
ম	3.30	কর	2.07	য়াছ	1.14	=এবং	0.90	=করিয়	0.86
ল	2.59	রে	1.75	কে=	1.13	ল্লাহ	0.89	রিয়াছ	0.86
স	2.40	=স	1.55	ার=	1.06	=যাহ	0.88	=আল্লা	0.74
য়	1.89	য়া	1.52	এবং	0.90	আল্লা	0.87	য়াছলি	0.73
ই	1.67	=আ	1.47	হা=	0.90	লাহ=	0.86	যাহা=	0.66
আ	1.62	=ব	1.38	মরা	0.89	য়াছি	0.80	গিকে=	0.64
য	1.27	মা	1.34	বং=	0.86	রিয়া	0.80	িয়াছে	0.60
্	1.20	রি	1.34	যাহ	0.85	করিয়	0.76	লাহ=	0.56
।	1.12	ি=	1.24	=হই	0.78	াহা=	0.76	যাহার	0.46
ু	1.10	কে	1.20	=যা	0.77	=আল্ল	0.61	াহার=	0.43
ী	1.07	=য	1.18	া=ক	0.74	াছলি	0.57	লাহর	0.41
শ	1.02	য়ি	1.02	াহ=	0.73	য়াছে	0.56	=করে=	0.37
ছ	1.01	াছ	0.91	=এব	0.71	=বলি	0.56	ারা=ব	0.36
ং	0.89	=এ	0.87	আল্ল	0.71	গিকে	0.54	=বললি	0.36
এ	0.87	=হ	0.87	লাহ	0.70	হার=	0.54	রণ=কর	0.34
ও	0.86	লি	0.87	=বল	0.70	=করে	0.52	াহর=	0.34
অ	0.80	=অ	0.84	ল্লা=	0.70	গকে=	0.50	=হইয়া	0.32
Total	20402		16148		12736		9989		7832
Unique	63		629		1730		2642		3057

Table 4.4(b): Percentage of occurrence of 25 6-8 grams for Bangla text with single byte representation.

6-grams		7-grams		8-grams	
াহারা=	2.05	=করিয়াছ	1.24	=করিয়াছি	0.86
আল্লাহ	1.38	=আল্লাহ	1.20	আল্লাহর=	0.72
=করিয়া	1.10	আল্লাহ=	0.88	=আল্লাহর	0.70
করিয়াছ	1.10	করিয়াছি	0.82	=আল্লাহ=	0.62
=আল্লাহ	0.95	আল্লাহর	0.67	করিয়াছিল	0.62
িয়াছিল	0.77	যাহারা=	0.63	=যাহারা=	0.59
=যাহা=	0.72	ল্লাহর=	0.57	=করিয়াছে	0.56
ল্লাহ=	0.72	াহারা=ব	0.52	াহারা=বল	0.51
রিয়াছি	0.64	রিয়াছিল	0.48	িয়াছিলাম	0.51
ল্লাহর	0.52	=যাহারা	0.46	বলিয়াছিল	0.45
যাহারা	0.51	করিয়াছে	0.46	=বলিয়াছি	0.40
হারা=ব	0.44	য়াছিলাম	0.44	=বিশ্বাস=	0.32
্লাহর=	0.44	হারা=বল	0.40	=রহিয়াছে	0.32
=যাহার	0.41	িয়াছিলা	0.40	রা=যাহা=	0.32
রিয়াছে	0.36	=বিশ্বাস	0.38	য়াছিলাম=	0.29
য়াছিলা	0.34	লিয়াছিল	0.38	া=আল্লাহ	0.29
াছিলাম	0.34	াহারা=ক	0.36	=গ্রহণ=কর	0.29
বিশ্বাস	0.34	বলিয়াছি	0.36	িশ্বাস=কর	0.29
=বলিয়া	0.33	রা যাহা	0.34	বিশ্বাস=ক	0.29
ারা=বল	0.31	=বলিয়াছ	0.31	হারা=বলি	0.27
িয়াছে=	0.31	রহিয়াছে	0.31	রিয়াছিলা	0.24
=বিশ্বা	0.29	াহারা=য	0.27	রহিয়াছে=	0.24
লিয়াছি	0.29	=রহিয়াছ	0.27	আল্লাহ=স	0.24
বলিয়াছ	0.28	বিশ্বাস=	0.27	ে=আল্লাহ	0.24
হারা=ক	0.28	া=যাহা=	0.25	হারা=বলে	0.24
Total	6108		4766		3739
Unique	3048		2820		2517

Table 4.5(a): Percentage of occurrence of 25 1-4 grams for Bangla text considering multibyte representation.

1-grams		2-grams		3-grams		4-grams	
	22.92	র=	3.38	=তাহা	1.56	এবং=	0.83
র	4.49	=ক	2.06	দের=	1.25	তাহারা=	0.72
ক	3.11	তাহা	1.81	=করি	1.06	=এবং	0.69
হা	3.08	রা=	1.67	হারা=	0.90	=তাহারা	0.68
ন	2.40	=আ	1.55	=এবং	0.81	তোমরা=	0.64
রা	2.18	=তা	1.50	তাহারা	0.78	তাহাদের	0.62
ব	2.15	ন=	1.46	=বং=	0.77	হাদের=	0.58
ই	2.06	দের	1.26	মরা=	0.73	মাদের=	0.53
আ	2.00	করি	1.16	=হই	0.70	=করিয়া	0.51
তা	1.97	হারা	0.97	=এব	0.63	=তাহাদে	0.49
ম	1.73	কে=	0.94	তোমরা	0.63	=তোমরা	0.46
ত	1.66	=এ	0.92	=যাহা	0.62	=আল্লাহ	0.43
য়া	1.48	=তো	0.89	হাদের	0.60	তোমাদের	0.40
রি	1.31	=অ	0.87	আল্লাহ	0.59	দিগকে=	0.38
মা	1.31	ং=	0.87	তাহাদে	0.57	=যাহা=	0.34
দে	1.31	ত=	0.86	করিয়া	0.53	র=প্রতি	0.33
হ	1.17	=হ	0.85	মাদের	0.53	আল্লাহ=	0.32
কে	1.17	ও=	0.84	=তোমা	0.51	র জন্য	0.32
স	1.15	=ব	0.81	=কর	0.51	=জন্য=	0.30
ত	1.14	=স	0.78	=তোম	0.42	=হইতে	0.30
ং	1.09	বং	0.78	=আল্লা	0.41	যখন=	0.30
নি	1.08	এব	0.76	=ও=	0.40	=তোমাদে	0.30
এ	1.08	হা=	0.75	খন=	0.40	করিয়াছি	0.30
ল	1.07	মরা	0.74	=বলি	0.39	হইতে=	0.30
ও	1.06	হই	0.71	দিগকে	0.38	=কোন=	0.27
Total	16534		15304		14193		13125
Unique	407		1979		4047		5943

Table 4.5(b): Percentage of occurrence of 20 5-6 grams for Bangla text considering multibyte representation.

	5-grams		6-grams	
	=এবং=	0.69	=তাহাদের=	0.51
	=তাহারা=	0.65	=তোমাদের=	0.30
	তাহাদের=	0.60	র=নিকট=	0.26
	=তাহাদের	0.53	দের=জন্য=	0.22
	=তোমরা=	0.46	তাহাদিগকে=	0.20
	তোমাদের=	0.39	=আল্লাহর=	0.19
	=তোমাদের	0.32	র=প্রতিপাল	0.18
	র=জন্য=	0.31	=তাহাদিগকে	0.17
	=হইতে=	0.30	=এবং=তাহা	0.14
	র=নিকট	0.27	=আমাদের=	0.14
	=করিয়াছি	0.26	=এবং=তা	0.13
	=নিকট=	0.26	তোমাদিগকে=	0.13
	আল্লাহর=	0.22	দের=নিকট	0.13
	দের=জন্য	0.22	=প্রতিপালক	0.12
	=যখন=	0.21	স্মরণ=কর	0.12
	=আল্লাহর	0.21	=ঈমান=আ	0.12
	হাদিগকে=	0.21	=করিয়াছেন	0.12
	তাহাদিগকে	0.20	=সুতরাং=	0.12
	আমাদের=	0.19	ং=তাহারা=	0.11
	=আল্লাহ=	0.19	=তোমাদিগকে	0.11
Total		12125		11232
Unique		7278		8044

Table 4.5(b): Percentage of occurrence of 20 7-8 grams for Bangla text considering multibyte representation.

7-grams		8-grams	
=তাহাদিগকে=	0.16	এবং=তাহারা=	0.09
দের=নিকট=	0.12	=নিকট=হইতে	0.09
র=প্রতিপালক	0.12	নিকট=হইতে=	0.09
=এবং=তাহা	0.12	তাহাদের=নিকট	0.09
প্রতিপালকের=	0.10	=আল্লাহর=নিকট	0.08
তোমাদিগকে=	0.10	ল্লাহর=নিকট=	0.08
=এবং=তাহারা	0.10	=প্রতিপালকের=	0.08
নিকট=হইতে	0.10	তাহাদের=জন্য=	0.08
মাদের জন্য=	0.09	আল্লাহর=নিক	0.07
বং=তাহারা=	0.09	এবং=তাহারা	0.07
=নিকট=হই	0.09	হাদের=নিকট=	0.07
হাদের=নিকট	0.09	র=নিকট=হই	0.07
কট=হইতে=	0.09	র=প্রতিপালকের	0.07
তাহাদের=নিক	0.08	যাহা=অবতীর্ণ	0.07
দের=প্রতিপাল	0.09	=তাহাদের=জন্য	0.06
=বিশ্বাস=করে	0.08	বে=না=এবং=	0.06
=প্রতিপালকের	0.08	যাহা=অবতীর্ণ=	0.06
=তোমার=প্রতি	0.08	লকের=নিকট=	0.06
=ঈমান=আনি	0.08	=কর=এবং=	0.06
=করিয়াছিলাম	0.08	=করিয়াছিলাম=	0.06
Total	10405		9639
Unique	8372		8312

4.3.1 n-Gram generation

The n-gram encoding technique represents an attempt to convert the normal hyperbolic distribution of single letters to a rectangular or equiprobable distribution of symbol groups (n-grams) by considering frequently occurring strings of up to **n** letters in length. Equiprobability increases entropy and hence decreases redundancy. As character strings of increasing length are considered, the number or variety of symbols increases but the greatest frequency decreases.

A sliding window of 8 characters length starting from the beginning of the representative text to the end moving along a character at a time. The n-grams produced were written to a file. We produce different files for different n-grams depending on the value of n. Then the n-grams are counted and written in another file and sorted in a third file. The n-gram can be produced and counted in a single pass using an n x m matrix, where m is the size of the alphabet. But it would require a huge amount of memory.

Table 4.6: Most frequent 25 Bangla words from a representable 26368 bytes BSCII text file. Total words = 4236, and unique words = 1238.

Words	Frequency	Frequency (%)
এবং	115	2.715
তাহারা	101	2.384
তোমরা	85	2.007
না	83	1.959
তাহাদের	81	1.912
কর	76	1.794
ও	70	1.653
যাহা	56	1.322
তোমাদের	51	1.204
আল্লাহ	48	1.133
করে	46	1.086
জন্য	43	1.015
হইতে	42	0.992
যখন	41	0.968
নিকট	38	0.897
কোন	38	0.897
যে	38	0.897
আমি	31	0.732
যাহারা	30	0.708
আল্লাহর	28	0.661
উহা	24	0.567
তাহাদিগকে	24	0.567
তাহার	24	0.567
তাহা	24	0.567
কি	24	0.567

Table 4.7(a): Natural Bangla Language Characteristics with multi-byte representation.

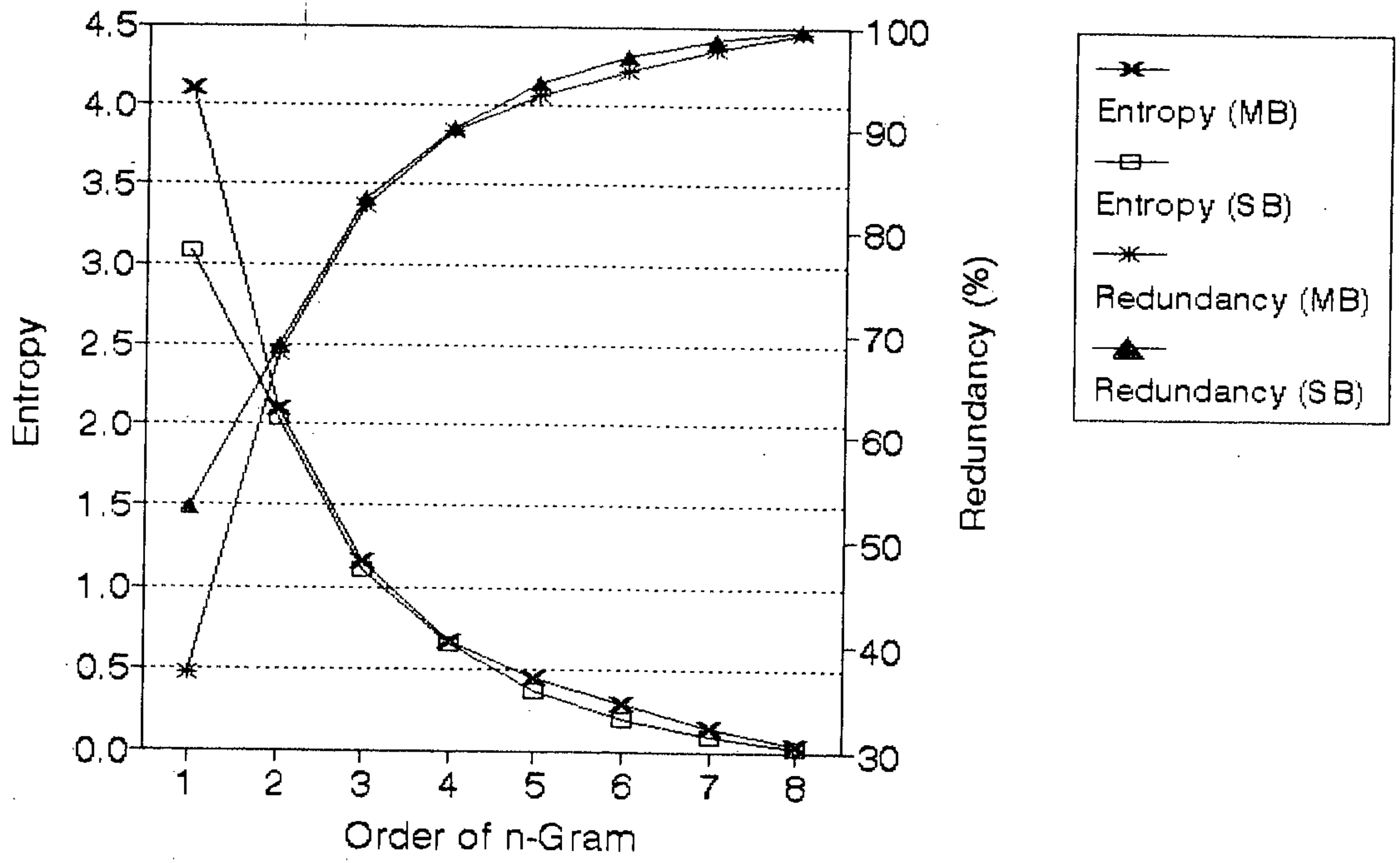
order n	Number of n-grams		H^n	H_n	%Redunduncy $\ln_2 94=6.55$
	Total	Unique			
1	16534	407	4.11	4.11	37.30
2	15304	1979	6.21	2.10	67.96
3	14193	4047	7.37	1.16	82.30
4	13125	5943	8.04	0.67	89.78
5	12125	7278	8.48	0.44	93.29
6	11232	8044	8.77	0.29	95.58
7	10405	8372	8.91	0.14	97.86
8	9639	8312	8.95	0.04	99.39

Table 4.7(b): Natural Bangla Language Characteristics with single-byte representation.

order n	Number of n-grams		H^n	H_n	%Redunduncy $\ln_2 94=6.55$
	Total	Unique			
1	20402	63	3.08	3.08	53.01
2	16148	629	5.12	2.04	68.88
3	12736	1730	6.23	1.11	83.07
4	9989	2642	6.88	0.65	90.08
5	7832	3057	7.24	0.36	94.51
6	6108	3048	7.43	0.19	97.10
7	4766	2820	7.52	0.09	98.63
8	3739	2517	7.54	0.02	99.69

Graph4.1: n-Gram Characteristics for Bangla Text

(MB for Multi-byte, SB for Single Byte)



Chapter Five

IMPLEMENTATION OF ALGORITHMS

In this thesis algorithms related to different data compression techniques have been implemented as Object Oriented Programming in Boarland C++. In addition, some utility routines have been developed for data analysis. Objects and their variations for all data compression techniques have been presented in this chapter.

5.1 Compression and Decompression

Compression and decompression techniques that we have been implemented can be classified as two general categories:

- a) Variable length coding
- b) Arithmetic coding

Variable length coding techniques are of two types :

1) Static variable length coding, and
2) Dynamic variable length coding

We have implemented two algorithms for static variable length coding:

- a) Shannon-Fano algorithm, and
- b) Huffman algorithm.

For dynamic variable length coding, the following algorithms have been considered:

a) FGK algorithm,
b) Knuth algorithm, and
c) Vitter algorithm.

Among the above variable length algorithm, Shannon-Fano, Huffman and FGK algorithms have been implemented both in scaled and unscaled symbol counts. Arithmetic coding algorithm has been considered only in static coding scheme with scaled symbol counts.

5.2 Classes for Data Compression

We have considered compression and decompression techniques in three classes. These three classes and their hierarchical relationship have been given below.

Among these three classes, *Utility* and *ComDecom* are common to all compression and decompression techniques. The class

Technique is different for different coding algorithms. The *Utility* and *ComDecom* classes are given in the following:

```
class Utility {
private:
        int dx1, dx2, dy1, dy2, dy3; // Co-ordinates on screen
protected:
        // Bit I/O data.
        FILE *file;
        unsigned char mask;
        int rack;
        long int ccount;
public:
        Utility(void) {dx1 = 14; dx2 = 28; dy1 = 2; dy2 = 3;  dy3 = 9;}

        // Common utility functions.
        void disp_scr(const  char *act, const char *fnt,  const char *fnc,
                const char *tech, const char *scaled, const char *model);
        char* getfname (const char *path);
        void usage(char *comm);
        void report (clock_t stime, clock_t etime,
                        long tcount, long scount, long ccount);
        void error (const int flag, const char *message);
        void outcode (const long scount, const long ccount);
        void outtcount (const long tcount);
        long int getccount(void);

        // Bit oriented I/O functions
        FILE *bfopen(const char *name, const char *mode);
        void fputb (int bit);
        void fputbs (unsigned long code, int count);
        int  fgetb  (void);
        unsigned long fgetbs (int bit_count);
        void bfclose  (void);
        void bfflush  (void);
        void fprintbs (FILE *file, unsigned int code, int bits);
};


class ComDecom : public Technique {
protected:
        char *tfspec, *cfspec;
        FILE *tfp, *cfp;
        long tcount, scount, ccount;
        clock_t stime, etime;
```

```
public:
      ComDecom (enum action_t act, int margc, char* margv[]);
      ~ComDecom (void){};
      void open_comp (void);  // Files open for compression
      void open_decomp (void);// Files open for decompression
      void compress (void);
      void decompress (void);
      void report (void);
};
```

Some functions of *Utility* class are used for common house keeping operations and others are used for bit oriented input and output operations. Functions of the class *ComDecom* are used to compress the text file to code file and to expand the compressed code file to the original text file. All these functions are given in the program *Util.CPP* and *ComDecom.CPP* respectively with proper documentation.

5.3 Members of ComDecom Class

The functions of coding and decoding of all techniques have some common routines as implemented in the *ComDecom* and *Utility* classes. Some of these are used in compression or decompression or in both. The data member and member functions in *ComDecom* class are common to all coding techniques. The *tfspec* and *cfspec* are the source and code file specifications respectively. The pointer variables *tfp* and *cfp* are file pointers for text and code file. Variables *tcount, scount* and *ccount* are size of the text file, the statistics of the text file to the code file and the actual code for the source message. The variables *scount* and *ccount* makes the code file size. These data are used to calculate coding efficiency of the algorithm. The variables *stime* and *etime* hold the starting and ending time of the compression or decompression function.

Common steps for all types of compression algorithms are

- Opening the input Bangla text file to read text symbols and an output codeword file to write binary bit streams.
- Call a function to compress the text file to a binary file.
- Report the compression ratio and time used.
- Close both files.

Similarly the common steps for all the types of decompression algorithms are:

- Opening the compressed binary file as input and a text file for the decoded Bangla text symbols as output file.
- Call a function to decompress the code file into a text file.
- Report the compression ratio and time used.
- Close both file.

The first steps of compression and decompression routines are implemented by *open_comp()* and *open_decomp()*. Third step is implemented by a *report()* for both compression and decompression program. This function gives the result of compression. In the second step, the member functions *compress()* and *decompress()* call the compression and decompression routine of the corresponding *Technique* class.

5.4 Members of Utility Class

Some of the member functions of Utility class are used for house keeping and others for unconventional bit oriented input/output operations.

Common House Keeping: The function *disp_scr()* gives a screen for progress report of compression or decompression. Variables dx1, dx2, dy1, dy2 and dy3 track the positions of these reports. Function *getfname()* give the true file name from the full file specification. Message for syntax error if any is traced by *usage()* while processing error message or any other message is traced by *error()*. The functions *outcode()*, *outtcount()* and *getccount()* are used to give and take the statistics of the files. The result of compression and decompression is reported by *report()*.

Bit oriented Input/Output: Data compression programs perform a lot of input/output that needs a single bit at a time. The standard C++ I/O library accommodates only I/O on even numbers of bytes boundaries. The library offers no help for single bit I/O at a time. The functions *bfopen()*, *bfclose()*, *fputb()*, *fgetb()*, *fputbs()* and *fgetbs()* are written to support this unconventional bit oriented I/O. Function *bfflush()* writes the remaining bit in the rack to the code file before leaving the compression routine. The function *fprintbs()* gives binary bit pattern of a given integer.

5.5 Static Variable Length Algorithms

Each of the static variable length techniques has to do the following steps:

1. Count symbol frequencies in the first pass of the source message,
2. Scaled symbol counts in the scaled version of the algorithm.
3. Save the counts (scaled counts for scaled version) for decoder.
4. Build the model for the encoder from the symbol counts.

5. Encode the source message in the second pass of the message to the code message.

Similarly the decoder must have the following steps in static variable length coding:

1. Retrieve the symbol counts saved by encoder.
2. Build the model for decoder from the symbol count.
3. Decode the coded message to the source message.

Steps 1 to 3 in the encoding algorithm and step 1 in the decoding algorithm are same for all static versions of coding and decoding algorithms. Step 4 in encoding and 2 in decoding are the same for a particular variable length coding technique. Step 5 in encoding and step 3 in decoding are the same for the two mentioned static variable-length coding algorithms.

Implementations of the class *Technique* for Shannon-Fano and Huffman algorithms with scaled and unscaled counts are almost similar. The data members which keep the weight field of the tree nodes and the member functions that count the symbols are different. The function *scale_counts()* is an additional member function for scaled count version of the implementation of the algorithms. This class with scaled symbol counts is given below:

```
typedef struct  tree_node {
      unsigned int count;
      int no;
      int left;
      int right;
      int parent;
} node_t;

typedef struct code {
      unsigned int code;
      int code_bits;
} CODE;


class Technique : public Utility {
private:
      unsigned long *counts;
      node_t *nodes;
      CODE *codes;
      int root_node;
protected:
      char tech[50], scaled[5], model [50];
public:
      Technique(void);
      int build_tree (void);
      void compress (FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
      void decompress(FILE *fi,FILE *fo, long *tc, long *sc, long *cc);

      long int count_bytes (FILE *fi);
      void scale_counts (void);
      long int output_counts  (FILE *fo);
      void tree2code (unsigned int code_so_far,int bits, int node);
      void text2code (FILE *fi);

      long int input_counts (FILE *fi);
      long code2text (FILE *fo);

      void fput_tree  (FILE *fo, int root, int depth);
      void fsave_tree (FILE *fi, FILE *fo);
      int index_codes (int *index);
      void fsave_code (FILE *fi, FILE *fo);
      unsigned int index_nodes (unsigned int *index);
};
```

Compression and Decompression Function: The function prototypes

```
void compress (FILE *fi,FILE *fo,long *tc,long *sc,long *cc);

void decompress (FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
```

of *Technique* class are common to all coding techniques but their implementations are different. Function *compress()* calls from the object of class *ComDecom* with the text and code file pointers *fi* and *fo*. This function compresses the source text file to the code file and return the statistics to the parent program. The statistics are (i) text file size, (ii) size of static tree statistics passed to the code file for static algorithms and (iii) actual code for the source message. These statistics can be used to find the coding efficiency of the algorithm. The function *decompress()* do the opposite action of *compress()*, i.e., this function actually decompresses the code file to the original text file.

Counting symbols: The member function *count_bytes()* counts the frequency of Bangla characters of the source text file. The frequency count for individual symbol is stored in a unsigned integer array element indexed by the symbol number.

Scaling Counts: The frequency counts of the symbols are stored in the unsigned long integer array. The file size of the compressed file can be reduced by scaling the counts to unsigned character which is single byte. This is done by the member function *scale_counts()* of the scaled version of the corresponding algorithm. This function finds the maximum value of the frequency counts and scales all the counts to an unsigned byte boundary.

Saving Frequency Counts: A copy of the static model identical to the one used by encoder must be passed to the decoder as a header of the code stream to decode correctly the code stream to the original source stream. The easiest way for the decoder is to get these as the entire model is passed in the header of the code stream. Many alternative methods that copy far less space in code stream can be possible. We have saved the frequency counts as a series of count-runs followed by a 0. A count-run consists of the value of the first symbol in the run, followed by the value of the last symbol in the run, followed by the counts (scaled counts) for all symbols in the run from first to last. This is repeated until each run has been stored in the header of the code stream. A typical source counts with corresponding saved list of runs is given in Table 5.1. The

Table 5.1 : Typical source symbol counts and list of runs of the output_counts() function.

1	0	26	0
2	0	27	0
3	0	28	0
4	0	29	0
5	7	30	2
6	3	31	4
7	0	32	4
8	9	33	0
9	2	34	0
10	0	35	0
11	0	36	0
12	0	37	0
13	0	38	0
14	0	39	0
15	0	40	0
16	0	41	0
17	1	42	0
18	7	43	0
19	0	44	0
20	4	45	0
21	23	46	9
22	0	47	0
23	0	48	0
24	0	49	0
25	0	50	0

(a) Source symbol counts

1st	5	First index
run	9	last index
	7	
	3	counts
	0	
	9	
	2	
2nd	17	first index
run	21	last index
	1	
	7	counts
	0	
	4	
	23	
3rd	30	first index
run	32	last index
	2	counts
	4	
	4	
4th	46	first index
run	46	last index
	9	counts
terminate	0	zero

(b) Header of code file

member function *output_counts()* saves the counts as a header of the code message.

Retrieve the Counts: The decoder retrieves the frequency counts from the header and builds the model. The member function *input_counts()* retrieves the counts to the weight field of the *nodes* array of the tree and returns the non-zero symbols.

Building Static Variable Length Coding Model: The model for static variable length coding techniques are the coding and decoding automata. These automata are the trees corresponding to the coding techniques. The models corresponding to the static variable length coding techniques, we have implemented, are the Huffman tree and Shannon-Fano tree. The tree corresponding to the technique is built from the (scaled) symbol frequency counts. The tree is implemented as an array of nodes. Each node consists of

- node number,
- weight of the node,
- parent index,
- left index,
- right index.

The node indexed by the symbol number is the leaf node corresponding to the symbol. If the total number of symbols of the alphabet including the end of message symbol is *CSIZ* then the number of node of the tree would be *NSIZ=2xCSIZ-1*. So the leaf node indexes would be from 1 to *CSIZ* and the internal node indexes would be from *(CSIZ + 1)* to *NSIZ*. The member function *build_tree()* builds the model. Shannon-Fano coding techniques need the nodes to be arranged in order of increasing weights of the nodes. So, *build_trees()* function needs another function *index_nodes()* that arranges the nodes in the required order by indexing the weight field.

Encoding Source Message: Encoding source message to code string with the static variable length coding scheme has two steps :

- converting the model, i.e., decoding binary tree to an array of codes, and
- encoding each symbol of the source message to the code string.

The functions *tree2code()* and *text2code()* do these two jobs. The function *tree2code()* traverses the tree and collects bit stream for the symbol corresponding to each leaf node and assigned to the element of the array of *CODE* structure. The function *text2code()* reads the text file and saves the codeword corresponding to the symbol.

Saving Static Trees Structure and Codes: The static Huffman and Shannon-Fano trees for a source message can be saved in the disk text file. Similarly the static codes for these two algorithms can be saved. Two classes *SaveCode* and *SaveTree* has been derived from *Technique* class. These classed are given here.

```
class SaveCode : public Technique {
private :
     FILE *fi, *fo;
     char finame[50], foname[50];
public:
     SaveCode (int margc, char *margv[]);
     void open(void);
     void svcode(void);
};

class SaveTree : public Technique {
private :
     FILE *fi, *fo;
     char finame[50], foname[50];
public:
     SaveTree (int margc, char *margv[]);
     void open(void);
     void svtree(void);
};
```

The member function *svcode()* of *SaveCode* class uses the member functions *index_codes()* and *fsave_code()* of the class *Technique* to save the codes. Similarly the member function *svtree()* of *SaveCode* class uses the member functions *fput_tree()* and *fsave_tree()* of the class *Technique* to save the tree corresponding to the algorithm. Function *open()* of class *SaveCode* and *SaveTree* simply open the source text file and corresponding output files for code or tree.

5.6 Dynamic Variable Length Coding

All dynamic algorithms encode the current symbol with the model built from the message just encoded, modify the model to adapt the effect of the current symbol. The coding and decoding algorithm for each of the dynamic variable length coding schemes are as follows:

Coding :

```
C1.  Initialize model;
C2.  while (not end of source text file) do
C3.       read a symbol from source text file;
C4.       Encode the symbol;
C5.       Update the model for the symbol;
C6.  end while;
```

Decoding:

```
D1.  Initialize model;
D2.  while (not end of code file) do
D3.       Decode code bit string to a symbol;
D4.       write the symbol to the text file;
D5.       Update the model for the symbol;
D6.  end while;
```

Functions implementing these steps are different for different dynamic variable length coding techniques but the steps C1 - D1 and C5 - D5 should be the same in compression and decompression for a definite coding technique. The member functions of the class *Technique* corresponding to the steps C1-D1 and C5-D5 are *initialize()* and *update()* while the steps C3 and D3 are *encode()* and *decode()*.

The implementations of class *Technique* for different dynamic coding algorithms are given in the following articles.

5.7 **FGK Algorithm**

It is an adaptive Huffman coding technique. Like any adaptive technique, both the encoder and decoder of the FGK algorithm start from same initial model and update the model for the symbol just processed. For FGK coding technique, the model is a binary tree, which must have properties that have in a Huffman tree with their weights in every node. The tree must follow the sibling properties and be maintained by implementing the FGK algorithm.

The FGK algorithm uses linked-list structures to represent the dynamically varying binary tree. The tree is an array of nodes. Each node of the tree represents (a) weight of the node, (b) parent node index and (c) child with lowest index. The correspondence between leaf node and the symbol is maintained by an array. Data structures and functions required for FGK algorithm are implemented in the following class.

```
class node_class  {
public:
      unsigned int weight;
      int parent;
      int child;      // child with lowest index
};

class Technique : public Utility {
private:
      node_class nodes [NSIZ]; // tree node array
      int leaf [SSIZ]; // leaf node corresponding symbol
      int free_node;   // next free node of the node array
      int l;           // tree rebuilding counter
protected:
       char tech[50], scaled[5], model [50];
public:
      Technique (void);
      void initialize(void);
      void encode(unsigned int c);
      int  decode(void);
      void update(int c);
      void rebuild (void);
      void swap_nodes(int i, int j);
      void add_node (int c);
      void compress  (FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
      void decompress(FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
};
```

Initialize the Model: Initially the model is tree with three nodes. One of the leaf node corresponds to the end of stream symbol and the other one represents all zero-weight symbols marked by ESC. The correspondence between character and leaf nodes is maintained by an array of integers that is also initialized.

Encoding Symbols: The leaf node corresponding to the symbol is found from the array data member *leaf[]*. The path of the tree from root to the leaf node corresponding to the symbol represents the codeword. The bit streams of the codeword is collected inversely from the leaf node to the root in an unsigned long integer. The symbol with zero weight is encoded by the leaf node corresponding to the *ESC* character and add the symbol to the model tree with the member function *add_node()*.

Decoding Code Stream: The decoder gets a bit from the coded message and branch left or right node of the tree depending on the value of the bit. If the node is a leaf the character corresponding to the leaf node outputs as source symbol. This is done by the member function *encode()* and similar for each of the dynamic algorithms.

Updating the Model: The most complicated part of the dynamic algorithms is updating the model which is done by the member function *update()*. This routine increases the weight of the leaf nodes corresponding to the symbol just processed and takes care of the side effect to maintaine the sibling property. In scaled version of the FGK algorithm the maximum value of the root node weight is maintained by rebuilding the tree by scaling the weights of all leaf nodes by the member function *rebuild()*. The updating routine finds the node with higher node number with equal weight to the current node weight and swap them and then increase the weight by 1. Node swapping is done by member function *swap_node()*.

5.8 Knuth Algorithm

In this algorithm, dynamic Huffman coding algorithm has been implemented as a group of array data structures. The class for this algorithm is given in the following.

```
class Technique : public Utility {
private:
      unsigned s[n], p[n+1], c[2*n], a[n+1], b[2*n], w[2*n], l[2*n];
      unsigned g[n*2], d[2*n], m, e, r, h, v, z, code, len, maxlen;
      unsigned testval, end;
protected:
      char tech[50], scaled[5], model[50];
public:
      Technique(void);
      void initialize(void);
      void exchange(unsigned q, unsigned t);
      void update(int k);
      void encode(int k);
      unsigned decode(void);
      void compress  (FILE *fi, FILE *fo, long *tc, long *sc,long *cc);
      void decompress(FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
};
```

Initialize Model: The model is initialized as a single zero weight leaf node. The function *initialize()* is use to initialize the tree.

Encoding Symbols: The leaf node corresponding to the correct symbol is found from the data structure. The bit stream representing the path from the leaf to the root is kept in a stack. Then codeword is the reverse order of this bit stream which can be written from the stack to the code file. The zero weight symbol is encoded as an optimal encoding scheme. The encoding algorithm is implemented in the function *encode()*.

Decoding Code Stream: Decoding procedure is easier than the encoding procedure. Starting from the root of the tree, follow a path to the leaf according to the bit getting from the bit stream and emit the symbols corresponding to the leaf. If the leaf node is a zero weight node then some of the next bit stream interprets the symbol. This is implemented in the function *decode()*.

Updating Tree: The update procedure is to adapt the Huffman tree for new counts of symbols. This can be done by maintaining the sibling properties. This function is also has to maintain the zero weight symbols. The sibling properties are maintained in two steps. In first step, it finds the highest numbered node with the same weight and interchange the current node with that node and in the second step, increment the weight of the node. The current node is the parent of the node in which weight is increased. The function *update()* is written to do this job by calling functions *exchange()*.

5.9 Vitter Algorithm

The steps to implement the Vitter algorithm, i.e., the optimal dynamic Huffman coding technique are similar to the Knuth algorithm. But the main difference is to maintain the invariant with implicit numbering scheme discussed previously in the algorithms. This is done by the updating and *slide_and_incrementing* routine. Initialization, encoding and decoding routines are implemented similar to the FGK algorithm, but maintained the node blocks and leader of the block. The class for the Vitter algorithm is given in the following.

```
class Technique : public Utility {
private:
      int m, r, e, z;
      int alpha[N1], rep [N1];
      int block[N2];
      long int weight[N2];
      int parent[N2],parity[N2],rtChild[N2],first[N2],last[N2];
      int prevBlock[N2], nextBlock[N2];
      int availBlock;
      int stack [N1];
      int q, leafToIncreament, bq, b, oldParent, oldParity;
      int slide, nbq, par, bpar;
protected:
      char tech[50], scaled[5], model[50];
public:
      Technique(void);
      void initialize(void);
      void update(int k);
      void encode(int k);
      unsigned decode(void);
      int  FindChild (int j, int parity);
      void InterchangeLeaves (int e1, int e2);
      void FindNode (int k);
      void SlideAndIncreament (void);
      void compress  (FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
      void decompress(FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
};
```

Updating Tree: Updating routine maintain the tree with a floating tree data structure. This tree use only right child and parent pointer to the leader of the block. A list of leaf and internal block of nodes are maintained using arrays of integer discussed in the algorithm section of the thesis. The main procedures used the updating routines are **slide_and_increment()**, **find_node()**, **find_child()**, and **interchange_leaves()**. Using these functions, update routine maintains the invariant (3.3).

Sliding node and Incrementing weight: The main routine which maintain invariant (3.3) with implicit numbering is the function **slide_and_increment()**. The current node is to be made a member of the block of node whose weight is one higher than the current node. So this node is interchanged with the leader of current block and is to move to the block whose weight is higher. This can cause the nodes implicit numbering to slide past the numberings of the nodes in the next block.

5.10 Arithmetic Coding

Arithmetic coding algorithm is implemented as static with scaled symbol counts. The implementation of the **Technique** class uses the almost similar member function for counting symbols, scaling symbol counts, saving scaled symbol counts and retrieve symbol ccounts as those of the scaled version of Shannon-Fano and Huffman algorithm. This class is given bellow.

```
class Technique : public Utility {
private:
        short int totals [258]; // range table.
        // range and code of the encoder.
        unsigned short int code;
        unsigned short int low;
        unsigned short int high;
        long underflow_bits;  // under flow count
        // range of the current symbol.
        unsigned short int low_count;
        unsigned short int high_count;
        unsigned short int scale;

        unsigned long *counts;
        unsigned char *scaled_counts;
protected:
        char tech[50], scaled[5], model [50];
public:
        Technique (void);
        void build_model (FILE *fi, FILE *fo, long *tc, long *sc);

        void scale_counts (void);
        void build_totals (void);

        long int count_bytes (FILE *fo);
        long int output_counts ( FILE *fo);
        long int input_counts (FILE *fi);

        void  int2symbol (int symbol);
        void  get_scale (void);
        int   symbol2int (int count);
        void  initialize_encoder (void);
        void  encode (void);
        void  flush_encoder (void);
        short int get_count (void);
        void  initialize_decoder (void);
        void  remove_symbol (void);
        void compress   (FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
        void decompress(FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
};
```

Compression: We implement the arithmetic coding algorithms as the integer arithmetics with incremental transmission method. The compression function performs three different jobs: (i) initialize the model and encoder, (ii) encoding and (iii) flushing the encoder.

The model is initialized with the member function **build_model()**. It counts the all symbols of the source text, scaled the counts and save the counts to the code file. Finally builds the range table for the coder. This functions use the function **count_bytes()**, **scale_counts()**, **output_counts()** and **build_totals()**. The encoder is initialized by function **initialize_encoder()** just by setting up the low and high value of the range and the under flow bit counter.

The compression function sets up loop to encode each symbol of the text file. It get a symbol, cover the symbol to the code rage by **int2symbol()**, and encode the symbol using **encode()**. Finally the arithmetic encoder is flushes to the code file using **flush_encoder()**.

Expanding: The model is rebuild from the symbol counts from the header of the code file. The decoding routing retrieve the symbol counts and build the range table using **input_counts()**. The decoder initializes with a word from the code message and the low and high range. This is done by *initialize_decoder()*. The expansion routine then sets up a loop and gets scale and counts of the symbol, convert the symbol range to the symbol (character) and remove the effect of the symbol from the range. These is done by *get_scale()*, *get_count()*, *symbol2int()* and *remove_symbol()* functions.

Chapter Six

DESIGN OF EXPERIMENTS AND RESULTS

6.1 Design of Experiments

In this study we would like to analyze performance of various coding algorithms applied to Bangla texts. We have selected text in the following formats:

(i) Document BNA format

(ii) Non-document XFR format

(iii) Non-document standard STD (BSCII) format

and for each format of the text input has been considered for general as well as special Bangla text. Files of the following different lengths (1000. 2000, ..., 10,000, 20,000, ..., 200,000) in bytes has been considered for ascertaining differences of efficiency of various algorithms. The algorithms considered for coding and decoding are Shannon-Fano, Huffman - static, Huffman - dynamic (Faller & Gallager, Knuth, Vitter) and Arithmetic coding. Most of these also have been run on both scaled and unscaled counts. As a measure of coding efficiency, coding and decoding times and also compression efficiencies have been considered. Observation on these quantities for different algorithms with general and specific types of texts stored in different formats has been made. Lengths of statistics and actual codes in compressed file also shown separately for static version of each of these algorithms.

6.2 Results

Results, obtained by applying different compression algorithms to different texts, have been presented in tables and graphs. Static Huffman Codes and Shannon-Fano codes for Bangla text are given in order of symbol numbers in Tables 6.1 and 6.2 respectively. The Huffman tree and Shannon-Fano tree corresponding to the codes are shown in Figs.6.1 and 6.2. The Tables 6.1 - 6.2 and Figs. 6.1 - 6.2 show that Huffman coding is better than Shannon-Fano coding. The tables showing the results of compression techniques are arranged in order of varying text formats and file types, with scaling or without scaling status, static or dynamic models and types of algorithms. For each file length, we show performance of various algorithms once for scaled counts and once for unscaled counts to compare performance of these algorithms on fixed length files.

Tables 6.3(a) to 6.3(c) show the compression efficiency, coding and decoding times of different algorithms for fixed length files for general and specific texts of different text formats. In Graphs 6.1 to 6.3, the compression efficiency,

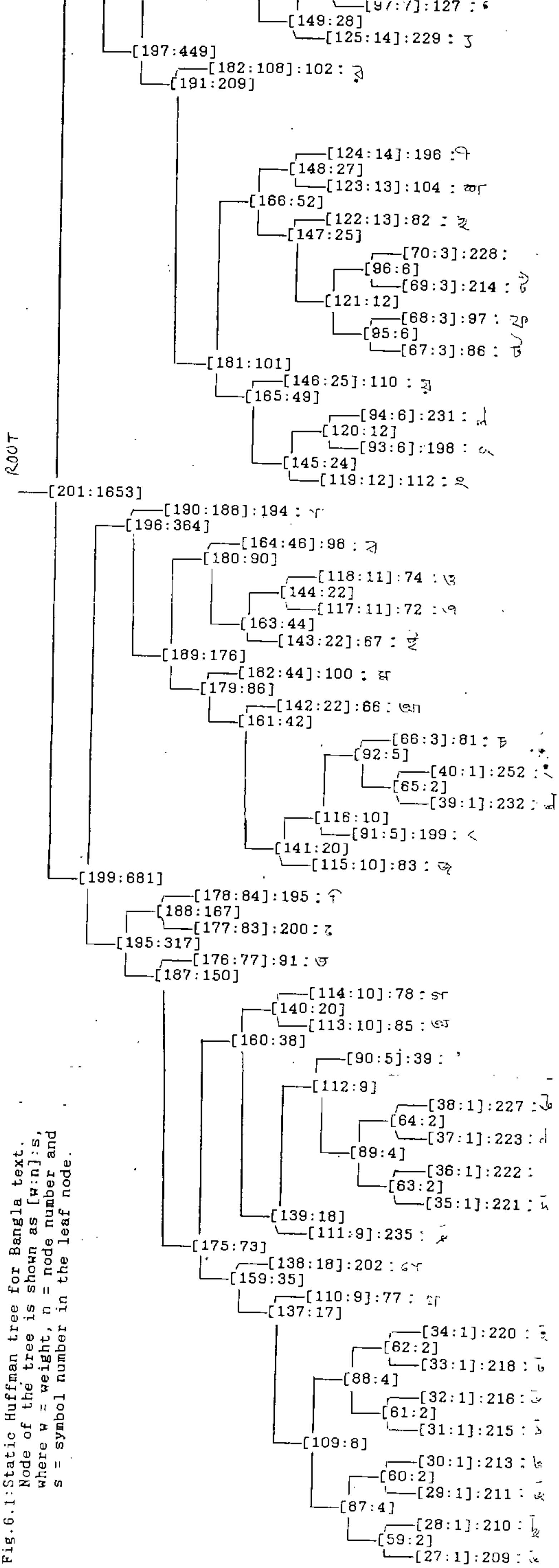
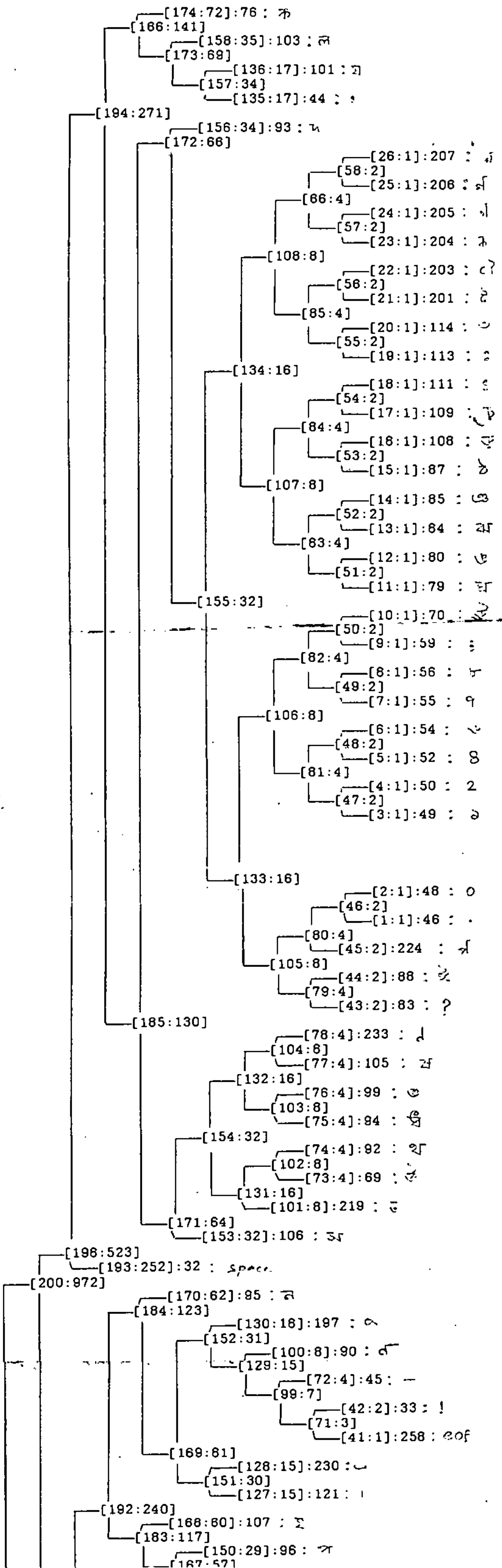
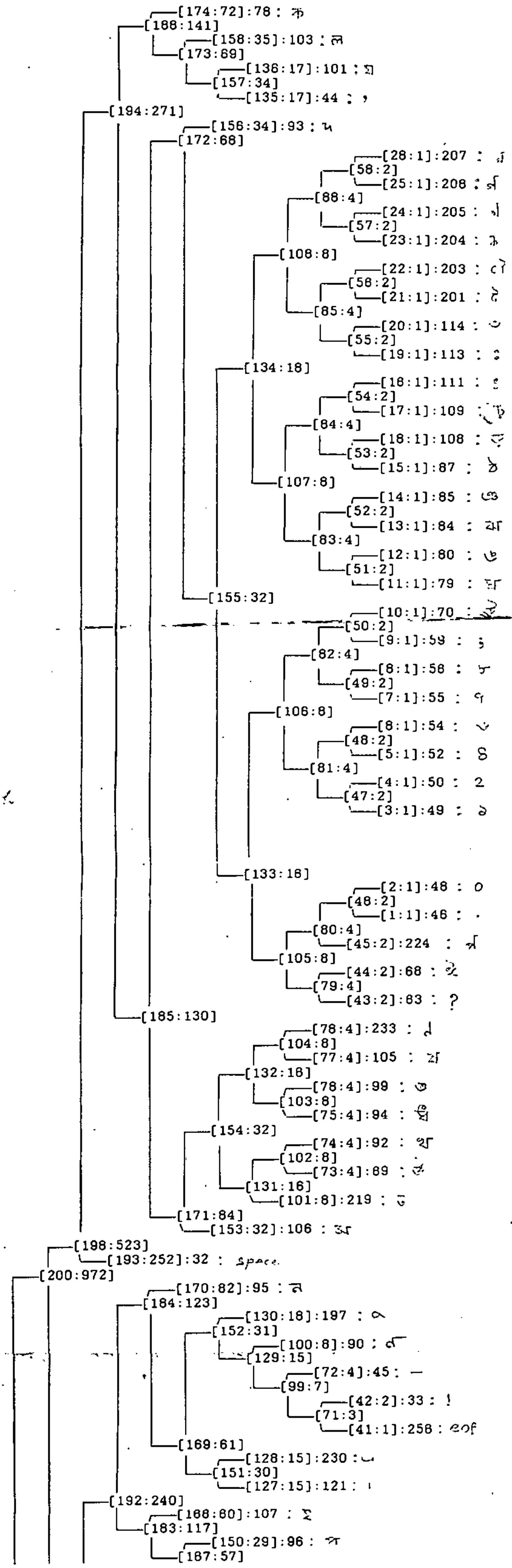
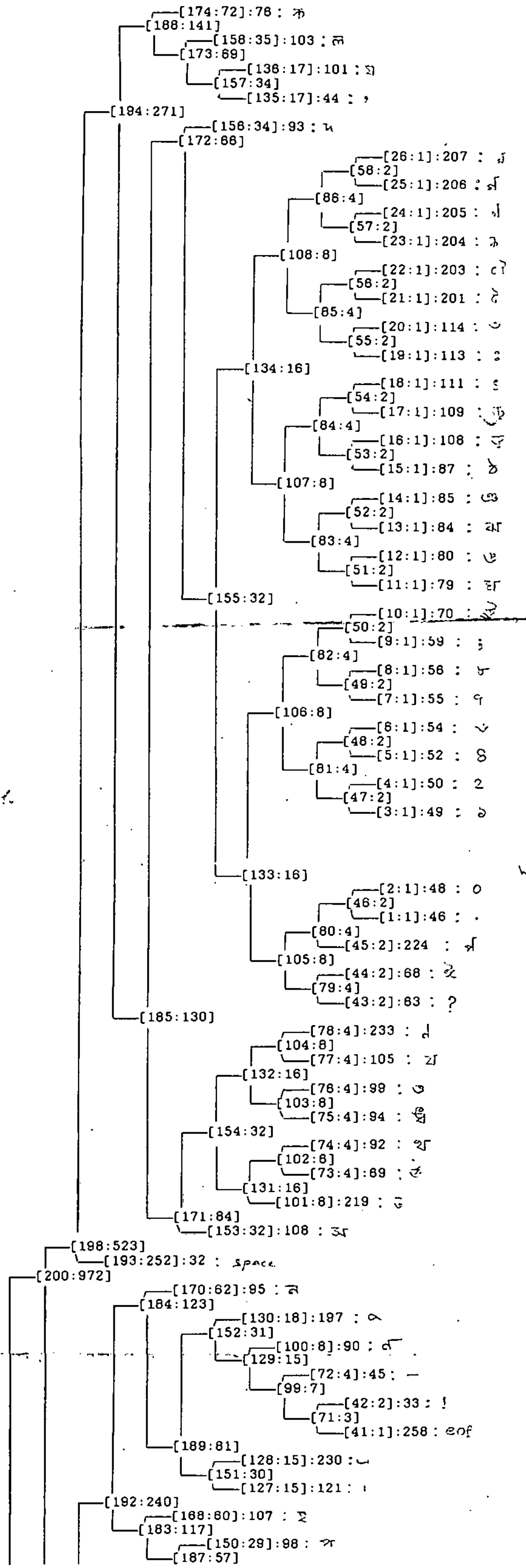


Fig.6.1:Static Huffman tree for Bangla text.
Node of the tree is shown as [w:n]:s,
where w = weight, n = node number and
s = symbol number in the leaf node.


[174:72]:76 : ক
[166:141]
[158:35]:103 : ল
[173:69]
[136:17]:101 :
[157:34]
[135:17]:44 :
[194:271]
[156:34]:93 : ন
[172:66]
[26:1]:207 :
[58:2]
[25:1]:206 :
[66:4]
[24:1]:205 :
[57:2]
[23:1]:204 :
[108:8]
[22:1]:203 :
[56:2]
[21:1]:201 :
[85:4]
[20:1]:114 :
[55:2]
[19:1]:113 :
[134:16]
[18:1]:111 :
[54:2]
[17:1]:109 :
[84:4]
[16:1]:108 :
[53:2]
[15:1]:87 :
[107:8]
[14:1]:85 :
[52:2]
[13:1]:84 :
[83:4]
[12:1]:80 :
[51:2]
[11:1]:79 :
[155:32]
[10:1]:70 :
[50:2]
[9:1]:59 :
[82:4]
[8:1]:56 :
[49:2]
[7:1]:55 : ৭
[106:8]
[6:1]:54 :
[48:2]
[5:1]:52 : ৪
[81:4]
[4:1]:50 : 2
[47:2]
[3:1]:49 :
[133:16]
[2:1]:48 : ০
[46:2]
[1:1]:46 : .
[80:4]
[45:2]:224 :
[105:8]
[44:2]:88 :
[79:4]
[43:2]:83 : ?
[185:130]
[78:4]:233 :
[104:8]
[77:4]:105 :
[132:16]
[76:4]:99 :
[103:8]
[75:4]:94 :
[154:32]
[74:4]:92 :
[102:8]
[73:4]:69 :
[131:16]
[101:8]:219 :
[171:64]
[153:32]:106 :
[196:523]
[193:252]:32 : space
[200:972]
[170:62]:85 :
[184:123]
[130:18]:197 :
[152:31]
[100:8]:90 :
[129:15]
[72:4]:45 : —
[99:7]
[42:2]:33 : !
[71:3]
[41:1]:258 : eof
[169:81]
[128:15]:230 :
[151:30]
[127:15]:121 : ।
[192:240]
[168:60]:107 :
[183:117]
[150:29]:96 :
[167:57]


[174:72]:78 : ক
[188:141]
[158:35]:103 : ল
[173:69]
[136:17]:101 : য
[157:34]
[135:17]:44 : ,
[194:271]
[156:34]:93 : ন
[172:68]
[28:1]:207 :
[58:2]
[25:1]:208 :
[88:4]
[24:1]:205 :
[57:2]
[23:1]:204 :
[108:8]
[22:1]:203 :
[56:2]
[21:1]:201 :
[85:4]
[20:1]:114 :
[55:2]
[19:1]:113 :
[134:18]
[18:1]:111 :
[54:2]
[17:1]:109 :
[84:4]
[16:1]:108 :
[53:2]
[15:1]:87 :
[107:8]
[14:1]:85 :
[52:2]
[13:1]:84 :
[83:4]
[12:1]:80 :
[51:2]
[11:1]:79 :
[155:32]
[10:1]:70 :
[50:2]
[9:1]:59 :
[82:4]
[8:1]:56 :
[49:2]
[7:1]:55 :
[106:8]
[6:1]:54 :
[48:2]
[5:1]:52 : 8
[81:4]
[4:1]:50 : 2
[47:2]
[3:1]:49 :
[133:16]
[2:1]:48 : 0
[46:2]
[1:1]:46 : .
[80:4]
[45:2]:224 :
[105:8]
[44:2]:68 :
[79:4]
[43:2]:63 : ?
[185:130]
[78:4]:233 :
[104:8]
[77:4]:105 :
[132:16]
[76:4]:99 :
[103:8]
[75:4]:94 :
[154:32]
[74:4]:92 :
[102:8]
[73:4]:89 :
[131:16]
[101:8]:219 :
[171:64]
[153:32]:106 :
[198:523]
[193:252]:32 : space
[200:972]
[170:82]:95 :
[184:123]
[130:18]:197 :
[152:31]
[100:8]:90 :
[129:15]
[72:4]:45 : —
[99:7]
[42:2]:33 : !
[71:3]
[41:1]:256 : eof
[169:61]
[128:15]:230 :
[151:30]
[127:15]:121 : ।
[192:240]
[168:60]:107 :
[183:117]
[150:29]:96 :
[167:57]


[174:72]:76 : ক
[188:141]
[158:35]:103 : ল
[173:69]
[136:17]:101 : য
[157:34]
[135:17]:44 : ,
[194:271]
[156:34]:93 : ন
[172:66]
[26:1]:207 :
[58:2]
[25:1]:206 :
[86:4]
[24:1]:205 :
[57:2]
[23:1]:204 :
[108:8]
[22:1]:203 :
[56:2]
[21:1]:201 :
[85:4]
[20:1]:114 :
[55:2]
[19:1]:113 : ঃ
[134:16]
[18:1]:111 :
[54:2]
[17:1]:109 :
[84:4]
[16:1]:108 :
[53:2]
[15:1]:87 : ঠ
[107:8]
[14:1]:85 : ঞ
[52:2]
[13:1]:84 :
[83:4]
[12:1]:80 :
[51:2]
[11:1]:79 :
[155:32]
[10:1]:70 :
[50:2]
[9:1]:59 : ;
[82:4]
[8:1]:56 : ৮
[49:2]
[7:1]:55 : ৭
[106:8]
[6:1]:54 : ৬
[48:2]
[5:1]:52 : ৪
[81:4]
[4:1]:50 : ২
[47:2]
[3:1]:49 : ১
[133:16]
[2:1]:48 : ০
[46:2]
[1:1]:46 : .
[80:4]
[45:2]:224 :
[105:8]
[44:2]:68 :
[79:4]
[43:2]:63 : ?
[185:130]
[78:4]:233 :
[104:8]
[77:4]:105 : য
[132:16]
[76:4]:99 :
[103:8]
[75:4]:94 :
[154:32]
[74:4]:92 : ধ
[102:8]
[73:4]:69 :
[131:16]
[101:8]:219 :
[171:84]
[153:32]:108 : অ
[198:523]
[193:252]:32 : space
[200:972]
[170:62]:95 : র
[184:123]
[130:18]:197 :
[152:31]
[100:8]:90 :
[129:15]
[72:4]:45 : —
[99:7]
[42:2]:33 : !
[71:3]
[41:1]:258 : eof
[189:81]
[128:15]:230 :
[151:30]
[127:15]:121 : ।
[192:240]
[168:60]:107 :
[183:117]
[150:29]:98 : প
[187:57]

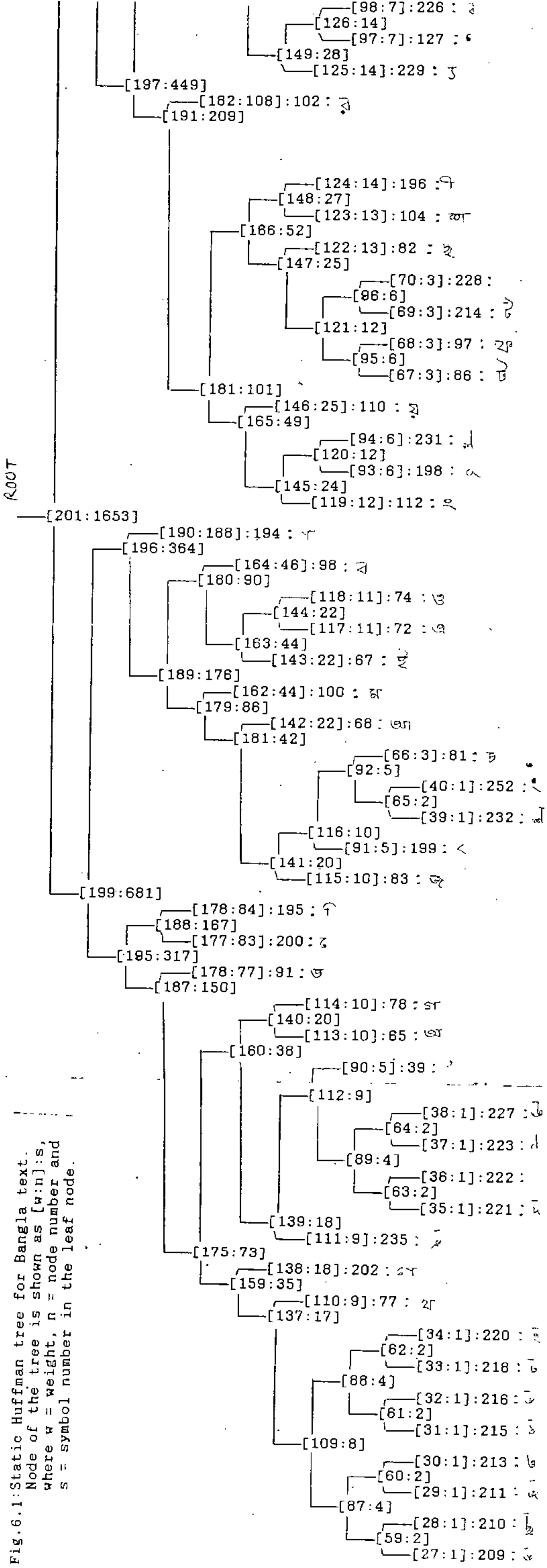
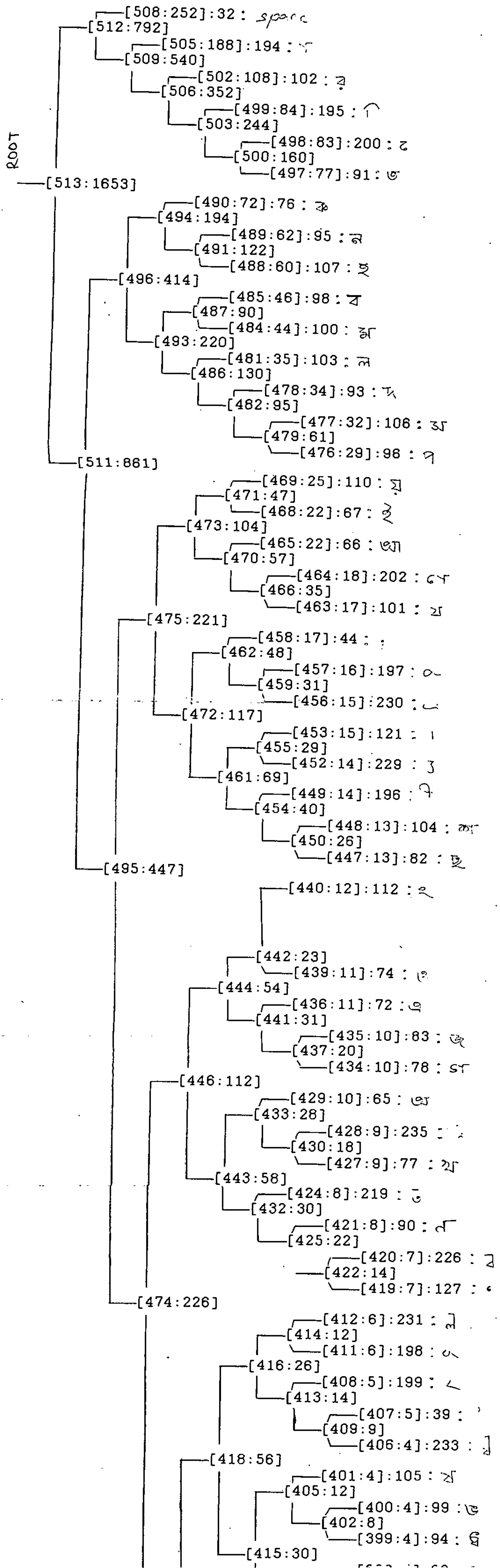


Fig.6.1:Static Huffman tree for Bangla text. Node of the tree is shown as [w:n]:s, where w = weight, n = node number and s = symbol number in the leaf node.


ROOT
[513:1653]
[512:792]
[508:252]:32 : space
[509:540]
[505:188]:194 :
[506:352]
[502:108]:102 : র
[503:244]
[499:84]:195 : ি
[500:160]
[498:83]:200 :
[497:77]:91 : ত
[511:861]
[496:414]
[494:194]
[490:72]:76 : ক
[491:122]
[489:62]:95 : ন
[488:60]:107 : হ
[493:220]
[487:90]
[485:46]:98 : ব
[484:44]:100 : স
[486:130]
[481:35]:103 : ল
[482:95]
[478:34]:93 : দ
[479:61]
[477:32]:106 :
[476:29]:96 : প
[495:447]
[475:221]
[473:104]
[471:47]
[469:25]:110 : য়
[468:22]:67 : ই
[470:57]
[465:22]:66 : আ
[466:35]
[464:18]:202 :
[463:17]:101 : য
[472:117]
[462:48]
[458:17]:44 : ,
[459:31]
[457:16]:197 :
[456:15]:230 :
[461:69]
[455:29]
[453:15]:121 : ।
[452:14]:229 :
[454:40]
[449:14]:196 :
[450:26]
[448:13]:104 : কা
[447:13]:82 :
[474:226]
[446:112]
[444:54]
[442:23]
[440:12]:112 :
[439:11]:74 :
[441:31]
[436:11]:72 : এ
[437:20]
[435:10]:83 : জ
[434:10]:78 : গ
[443:58]
[433:28]
[429:10]:65 : অ
[430:18]
[428:9]:235 :
[427:9]:77 : থ
[432:30]
[424:8]:219 : ু
[425:22]
[421:8]:90 :
[422:14]
[420:7]:226 :
[419:7]:127 :
[418:56]
[416:26]
[414:12]
[412:6]:231 :
[411:6]:198 :
[413:14]
[408:5]:199 :
[409:9]
[407:5]:39 : '
[406:4]:233 :
[415:30]
[405:12]
[401:4]:105 : শ
[402:8]
[400:4]:99 : উ
[399:4]:94 :

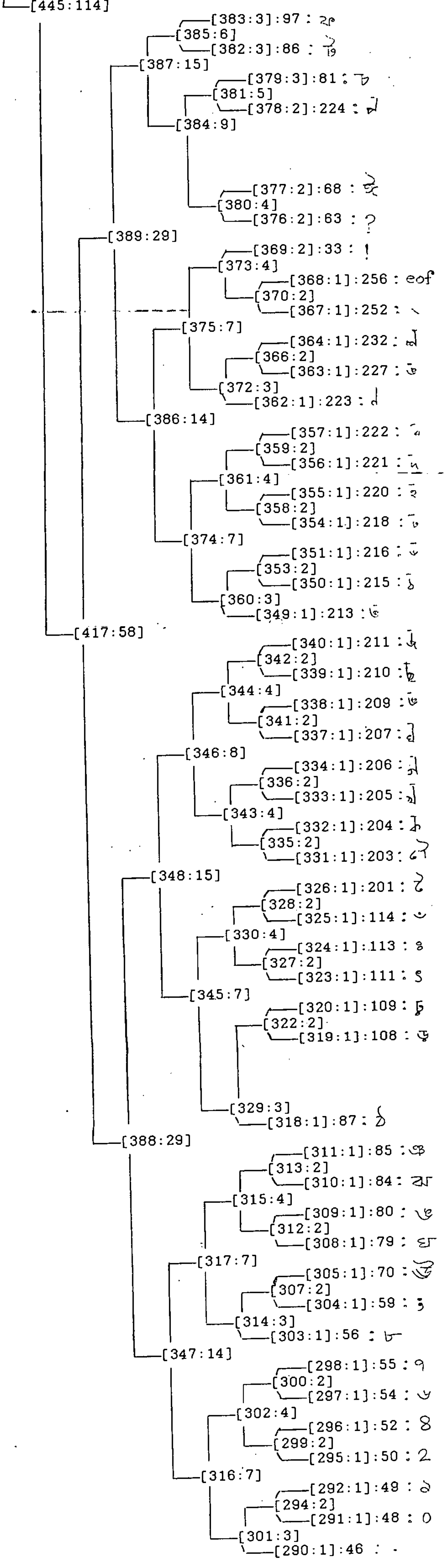


Fig.6.2:Static Shannon-Fano tree for Bangla text.
Node of the tree is shown as [w:n]:s,
where w = weight, n = node number and
s = symbol number in the leaf node.

Table 6.1(a) Static Huffman Codes in order of symbol number.

Symbols	Counts	Codes	Symbols	Counts	Codes
32	252	110	106 স	32	111000
33 !	2	1011010001	107 হ	60	10101
39 '	5	00001011	108 ড়	1	11101010101
44 ,	17	1111000	109 ঢ়	1	11101010110
45 -	4	101101001	110 য়	25	100001
46 .	1	11101000110	111 ৎ	1	11101010111
48 0	1	11101000111	112 ং	12	1000000
49 ১	1	11101001000	113 ঃ	1	11101011000
50 ২	1	11101001001	114 ঁ	1	11101011001
52 ৪	1	11101001010	121 ।	15	1011000
54 ৬	1	11101001011	127 [illegible]	7	10100010
55 ৭	1	11101001100	194 া	188	011
56 ৮	1	11101001101	195 ি	84	0011
59 ;	1	11101001110	196 ী	14	1000111
63 ?	2	1110100000	197 ু	16	1011011
65 অ	10	0000110	198 ূ	6	10000010
66 আ	22	010001	199 ৃ	5	01000010
67 ই	22	010100	200 ে	83	0010
68 ঈ	2	1110100001	201 ৈ	1	11101011010
69 উ	4	111001010	202 ো	18	000001
70 ঊ	1	11101001111	203 ৌ	1	11101011011
72 এ	11	0101010	204 [illegible]	1	11101011100
74 ও	11	0101011	205 [illegible]	1	11101011101
76 ক	72	11111	206 [illegible]	1	11101011110
77 খ	9	0000001	207 [illegible]	1	11101011111
78 গ	10	0000111	209 [illegible]	1	0000000000
79 ঘ	1	11101010000	210 [illegible]	1	0000000001
80 ঙ	1	11101010001	211 [illegible]	1	0000000010
81 চ	3	010000111	213 [illegible]	1	0000000011
82 ছ	13	1000101	214 [illegible]	3	100010010
83 জ	10	0100000	215 [illegible]	1	0000000100
84 ঝ	1	11101010010	216 [illegible]	1	0000000101
85 ঞ	1	11101010011	218 [illegible]	1	0000000110
86 ট	3	100010000	219 [illegible]	8	11100100
87 ঠ	1	11101010100	220 [illegible]	1	0000000111
90 ণ	8	10110101	221 [illegible]	1	0000101000
91 ত	77	0001	222 [illegible]	1	0000101001
92 থ	4	111001011	223 [illegible]	1	0000101010
93 দ	34	111011	224 [illegible]	2	1110100010
94 ধ	4	111001100	226 [illegible]	7	10100011
95 ন	62	10111	227 [illegible]	1	0000101011
96 প	29	101001	228 [illegible]	3	100010011
97 ফ	3	100010001	229 [illegible]	14	1010000
98 ব	46	01011	230 [illegible]	15	1011001
99 ভ	4	111001101	231 [illegible]	6	10000011
100 ম	44	01001	232 [illegible]	1	0100001100
101 য	17	1111001	233 [illegible]	4	111001111
102 র	108	1001	235 [illegible]	9	0000100
103 ল	35	111101	252 [illegible]	1	0100001101
104 শ	13	1000110	256 eof	1	1011010000
105 ষ	4	111001110			

Table 6.2(a) Static Shannon-Fano codes in order of symbol number.

Symbols		Counts	Codes	Symbols		Counts	Codes
32		252	11	106		32	0100001
33	!	2	0000010111	107		60	01100
39	'	5	000011001	108		1	00000010010
44	,	17	001011	109		1	00000010011
45	–	4	000010001	110		25	001111
46	.	1	0000000000	111		1	00000010100
48	০	1	00000000010	112		12	0001111
49	১	1	00000000011	113		1	00000010101
50	২	1	00000000100	114		1	00000010110
52	৪	1	00000000101	121		15	0010011
54	৬	1	00000000110	127		7	000100000
55	৭	1	00000000111	194		188	101
56	৮	1	0000000100	195		84	10001
59	;	1	00000001010	196		14	0010001
63	?	2	0000011000	197		16	0010101
65		10	0001011	198		6	00001110
66		22	001101	199		5	00001101
67		22	001110	200		83	100001
68		2	0000011001	201		1	00000010111
69		4	000010010	202		18	0011001
70		1	00000001011	203		1	00000011000
72		11	0001101	204		1	00000011001
74		11	0001110	205		1	00000011010
76		72	0111	206		1	00000011011
77		9	00010100	207		1	00000011100
78		10	00011000	209		1	00000011101
79		1	00000001100	210		1	00000011110
80		1	00000001101	211		1	00000011111
81		3	0000011011	213		1	0000010000
82		13	00100000	214		3	0000100000
83		10	00011001	215		1	00000100010
84		1	00000001110	216		1	00000100011
85		1	00000001111	218		1	00000100100
86		3	000001110	219		8	0001001
87		1	0000001000	220		1	00000100101
90		8	00010001	221		1	00000100110
91		77	100000	222		1	00000100111
92		4	000010011	223		1	0000010100
93		34	010001	224		2	0000011010
94		4	000010100	226		7	000100001
95		62	01101	227		1	00000101010
96		29	0100000	228		3	0000100001
97		3	000001111	229		14	0010010
98		46	01011	230		15	0010100
99		4	000010101	231		6	00001111
100		44	01010	232		1	00000101011
101		17	0011000	233		4	000011000
102		108	1001	235		9	00010101
103		35	01001	252		1	00000101100
104		13	00100001	256	eof	1	00000101101
105		4	00001011				

Table 6.3(a) : Coding efficiency (%) for fixed file size.

File Size	Coding Technique	Scaled Count	General Text			Specific Text		
			STD	XFR	BNA	STD	XFR	BNA
1000	Shannon-Fano	Yes	24.00	19.60	13.00	20.50	19.70	13.50
		No	-5.10	-15.00	-32.00	-13.40	-13.60	-28.50
	Huffman	Yes	24.70	20.70	13.60	22.00	21.00	14.10
		No	-4.40	-14.10	-31.40	-11.90	-12.30	-27.90
	FGK	Yes	28.70	27.80	22.00	27.20	27.30	21.10
		No	28.70	27.80	22.00	27.20	27.30	21.10
	Knuth	No	27.50	26.60	20.60	26.10	26.30	19.70
	Vitter	No	28.40	27.40	21.70	26.80	27.00	21.10
	Arithmetic	Yes	24.80	20.90	13.70	22.10	21.00	14.20
10000	Shannon-Fano	Yes	33.16	31.40	29.11	32.80	31.66	28.84
		No	29.61	26.76	23.46	28.96	26.61	23.06
	Huffman	Yes	33.70	32.42	29.93	33.98	33.02	29.68
		No	30.42	27.88	24.42	30.47	28.15	24.12
	FGK	Yes	34.07	33.22	30.67	34.41	33.99	30.49
		No	34.07	33.22	30.67	34.41	33.99	30.49
	Knuth	No	33.90	33.16	30.51	34.27	33.88	30.35
	Vitter	No	34.03	33.30	30.68	34.36	33.99	30.52
	Arithmetic	Yes	34.16	32.90	30.20	34.39	33.39	30.02
200000	Shannon-Fano	Yes	33.56	31.30	27.83	32.35	31.89	31.29
		No	33.65	31.23	30.97	32.23	31.76	30.97
	Huffman	Yes	34.56	32.67	30.17	33.98	33.51	32.12
		No	34.47	33.24	29.96	34.15	34.05	31.66
	FGK	Yes	34.96	33.86	31.23	34.30	34.09	32.90
		No	34.84	33.68	30.97	34.17	33.98	32.63
	Knuth	No	34.83	33.67	30.97	34.16	33.98	32.62
	Vitter	No	34.84	33.69	30.99	34.17	33.98	32.63
	Arithmetic	Yes	34.92	33.03	30.47	34.33	33.73	32.61

Note : STD = BSCII format, XFR = Non-document format and BNA = Document format.

Table 6.3(b) : Compression time (sec.) for fixed file size.

File Size	Coding Technique	Scaled Count	General Text			Specific Text		
			STD	XFR	BNA	STD	XFR	BNA
1000	Shannon-Fano	Yes	0.99	1.04	1.04	1.04	1.04	1.10
		No	1.04	1.04	1.15	1.10	1.04	1.15
	Huffman	Yes	0.99	0.99	1.10	1.10	1.10	1.15
		No	1.04	1.10	1.21	1.10	1.10	1.15
	FGK	Yes	1.04	1.04	1.10	1.04	1.04	1.10
		No	1.10	1.10	1.15	1.10	1.10	1.10
	Knuth	No	0.99	1.04	0.99	1.04	0.99	1.10
	Vitter	No	1.21	1.26	1.32	1.21	1.26	1.32
	Arithmetic	Yes	1.10	1.10	1.10	1.15	1.10	1.10
10000	Shannon-Fano	Yes	9.07	9.18	9.29	9.18	9.12	9.45
		No	9.23	9.40	9.67	9.29	9.12	9.62
	Huffman	Yes	9.07	9.18	9.51	9.12	9.12	9.51
		No	9.18	9.29	9.56	9.23	9.18	9.62
	FGK	Yes	10.00	10.16	10.38	10.16	10.11	10.60
		No	10.22	10.33	10.60	10.33	10.33	10.77
	Knuth	No	9.95	10.00	10.22	10.00	9.95	10.38
	Vitter	No	11.81	11.96	12.25	11.87	11.76	12.31
	Arithmetic	Yes	10.55	10.71	10.77	10.71	10.60	10.88
200000	Shannon-Fano	Yes	193.85	196.43	201.43	195.77	196.48	197.75
		No	197.09	199.62	198.68	195.71	196.32	198.68
	Huffman	Yes	193.57	195.49	199.29	193.02	193.24	196.48
		No	194.01	195.74	201.43	194.22	194.53	197.40
	FGK	Yes	212.53	215.27	219.89	214.95	215.55	218.13
		No	214.95	216.92	221.81	216.32	217.03	219.67
	Knuth	No	212.53	211.92	217.75	213.24	213.08	216.26
	Vitter	No	247.75	248.65	254.56	248.41	248.79	252.56
	Arithmetic	Yes	226.32	227.75	230.77	226.54	226.76	228.35

Note : STD = BSCII format, XFR = Non-document format and BNA = Document format.

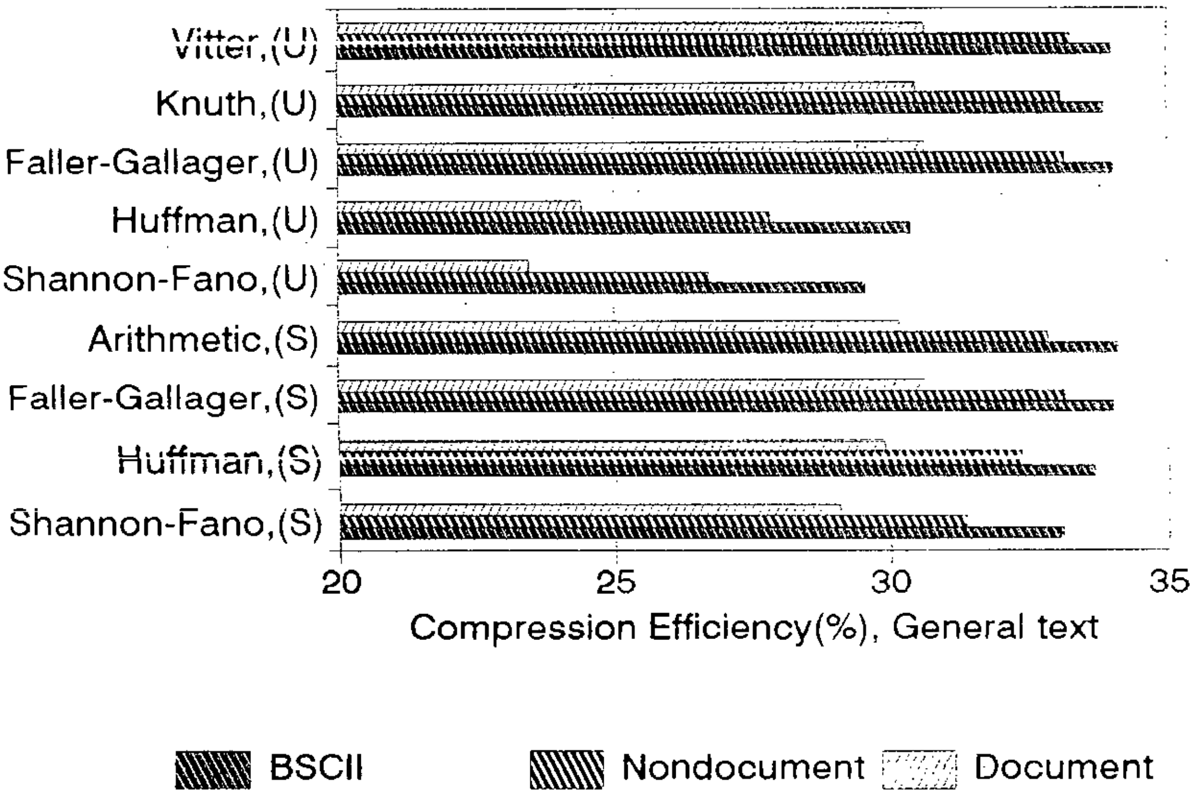
Table 6.3(c) : Decompression time (sec.) for fixed file size.

File Size	Coding Technique	Scaled Count	General Text			Specific Text		
			STD	XFR	BNA	STD	XFR	BNA
1000	Shannon-Fano	Yes	0.93	0.99	1.10	0.99	0.99	1.04
		No	0.99	1.04	1.10	1.04	1.04	1.15
	Huffman	Yes	0.99	1.04	1.10	1.04	0.99	1.10
		No	0.99	1.04	1.15	1.04	1.04	1.21
	FGK	Yes	1.04	1.04	1.10	1.04	0.99	1.10
		No	1.04	0.99	1.15	1.04	1.10	1.15
	Knuth	No	0.99	0.99	1.04	0.99	0.99	1.04
	Vitter	No	1.21	1.21	1.26	1.26	1.21	1.26
	Arithmetic	Yes	1.26	1.37	1.37	1.32	1.32	1.37
10000	Shannon-Fano	Yes	9.12	9.23	9.45	9.18	9.18	9.45
		No	9.18	9.34	9.56	9.23	9.34	9.62
	Huffman	Yes	9.07	9.18	9.4	9.01	9.12	9.45
		No	9.12	9.4	9.62	9.18	9.29	9.62
	FGK	Yes	9.73	9.95	10.05	9.73	9.78	10.05
		No	9.95	10.11	10.33	10.05	10	10.33
	Knuth	No	9.89	10	10.22	9.78	9.84	10.11
	Vitter	No	12.03	12.09	12.36	11.92	12.03	12.36
	Arithmetic	Yes	13.19	13.3	13.68	13.3	13.35	13.74
200000	Shannon-Fano	Yes	197.31	200.55	208.02	199.34	200.49	200.99
		No	197.80	201.37	194.80	201.04	201.26	194.89
	Huffman	Yes	195.05	197.86	202.36	195.22	196.10	198.08
		No	195.80	198.28	203.11	195.58	196.50	198.90
	FGK	Yes	208.24	210.60	215.77	208.85	209.67	212.31
		No	211.59	213.85	218.30	211.76	212.47	214.78
	Knuth	No	213.46	214.84	220.33	212.31	212.64	214.95
	Vitter	No	253.46	254.45	260.93	253.96	254.23	257.36
	Arithmetic	Yes	281.59	248.84	291.81	283.35	284.23	288.68

Note : STD = BSCII format, XFR = Non-document format and BNA = Document format.

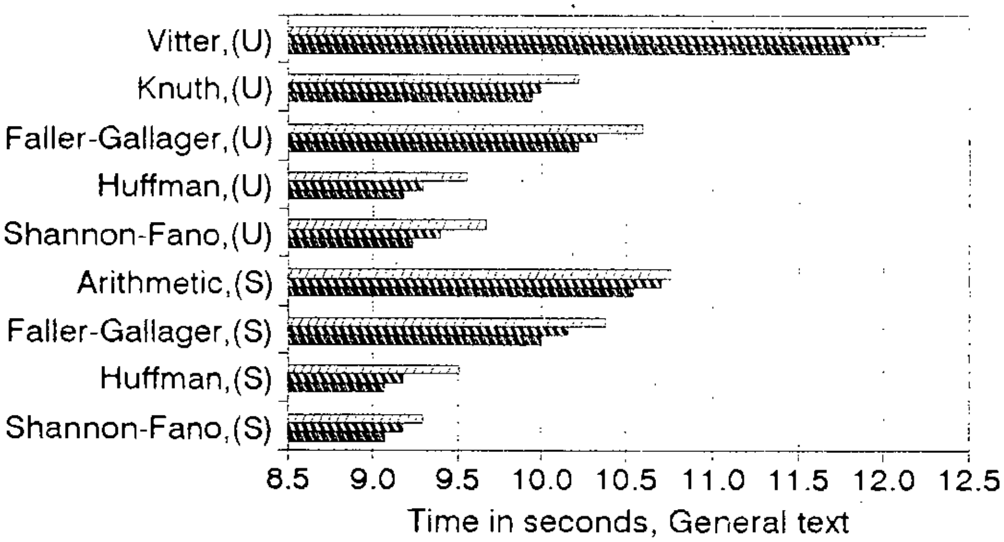
Graph 6.1: Compression Efficiency

(S for scaled and U for unscaled)



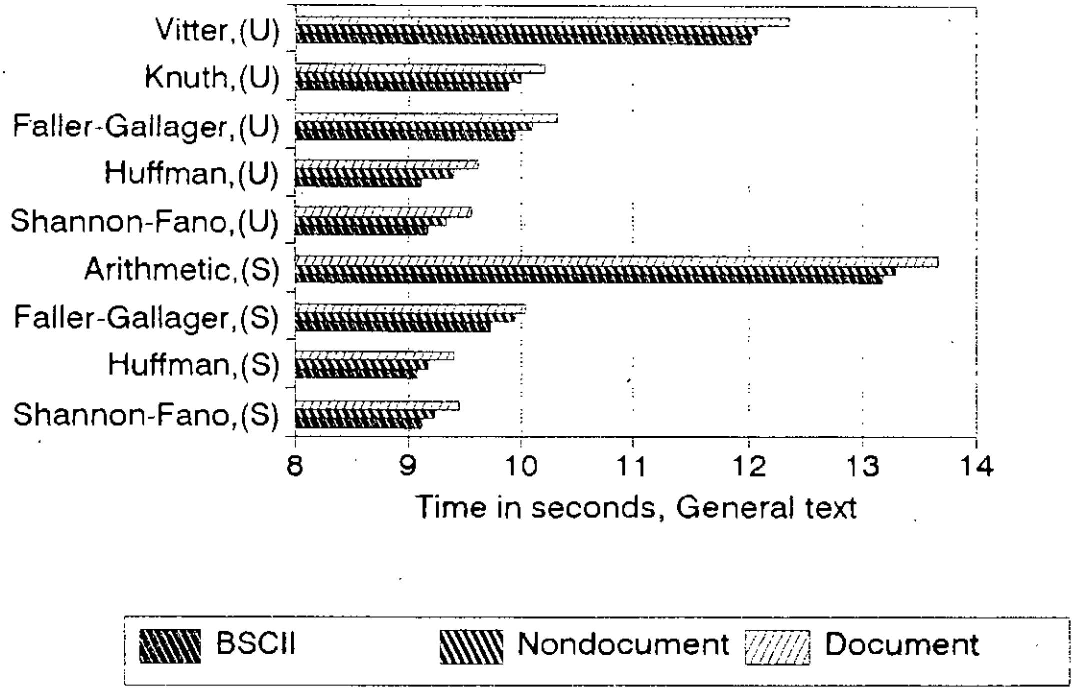
Graph 6.2: Compression Time

(S for scaled and U for unscaled)



Graph 6.3: Decompression Time

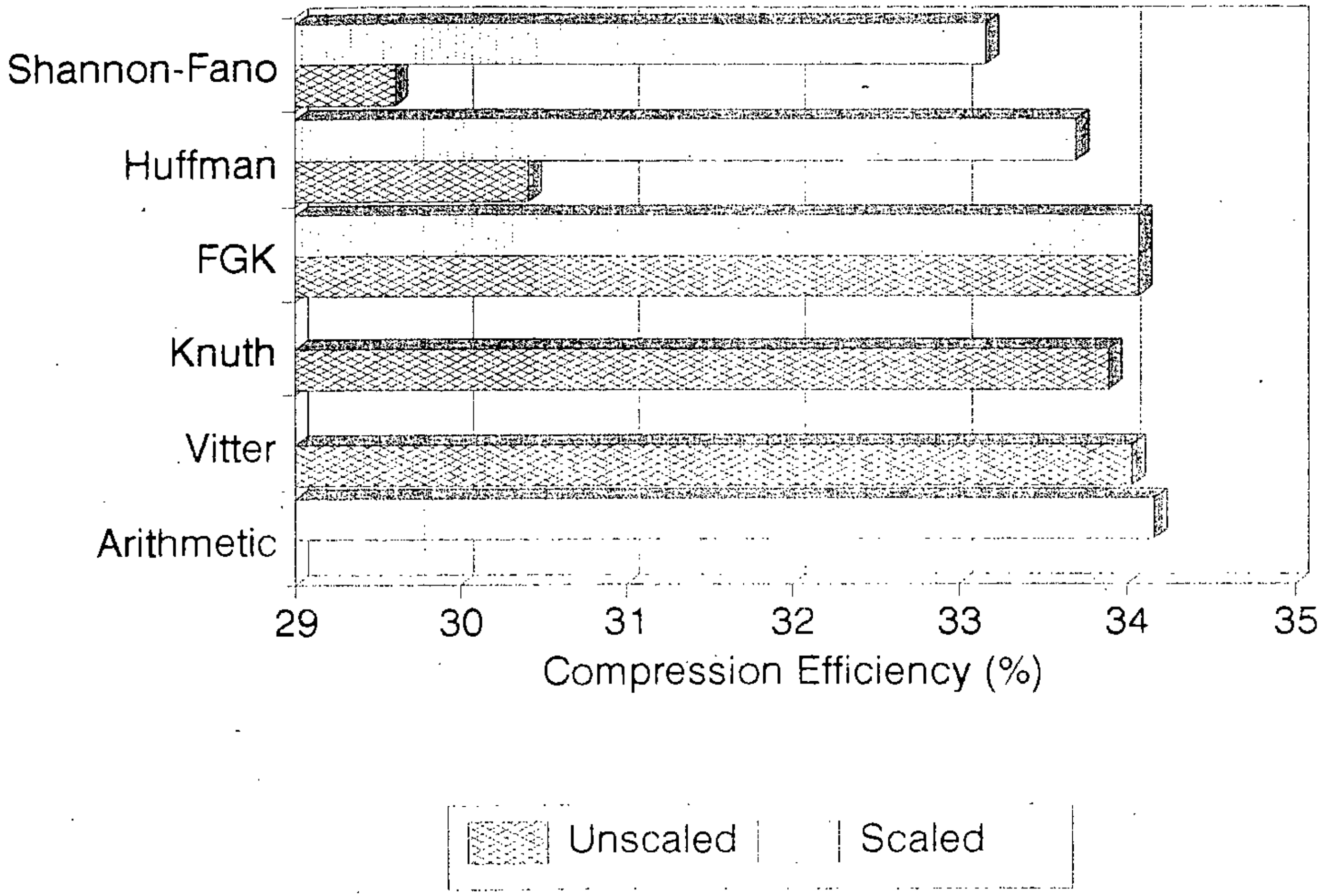
(S for scaled and U for unscaled)



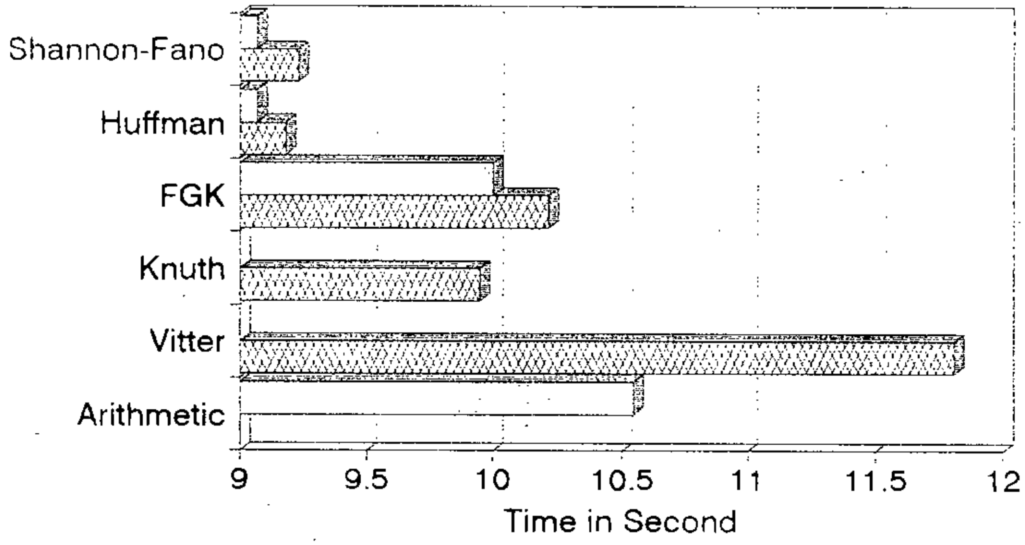
coding and decoding times for different coding techniques for different text formats on a fixed length general text are given. Compression efficiency of different algorithms for general text varies from 27.83% - for Shannon-Fano algorithm with scaled symbol count for general document (BNA) format texts, to 34.92% - for arithmetic coding with scaled symbol count for general standard BSCII (STD) format texts for 200000-byte file. Compression time varies from 193.24 sec. - for Huffman algorithm with scaled symbol count for non-document XFR format specific text, to 254.56 sec. - for Vitter algorithm without scaled count document BNA format general text for the same size text. Decompression time also varies from 195.05 sec. - for Huffman algorithm with scaled symbol count for standard non-document BSCII format general text, to 291.81 sec. - for arithmetic coding with scaled symbol count for non-document format general text for the same size file.

Effects of scaling the symbol counts of various coding techniques for standard non-document BSCII format general text are shown in Graphs 6.4 to 6.6. Shannon-Fano, Huffman and FGK algorithm have been tested for both scaled and unscaled symbol counts. In the Table 6.3(a) and Graph 6.4, coding efficiencies of all algorithms have been found better by

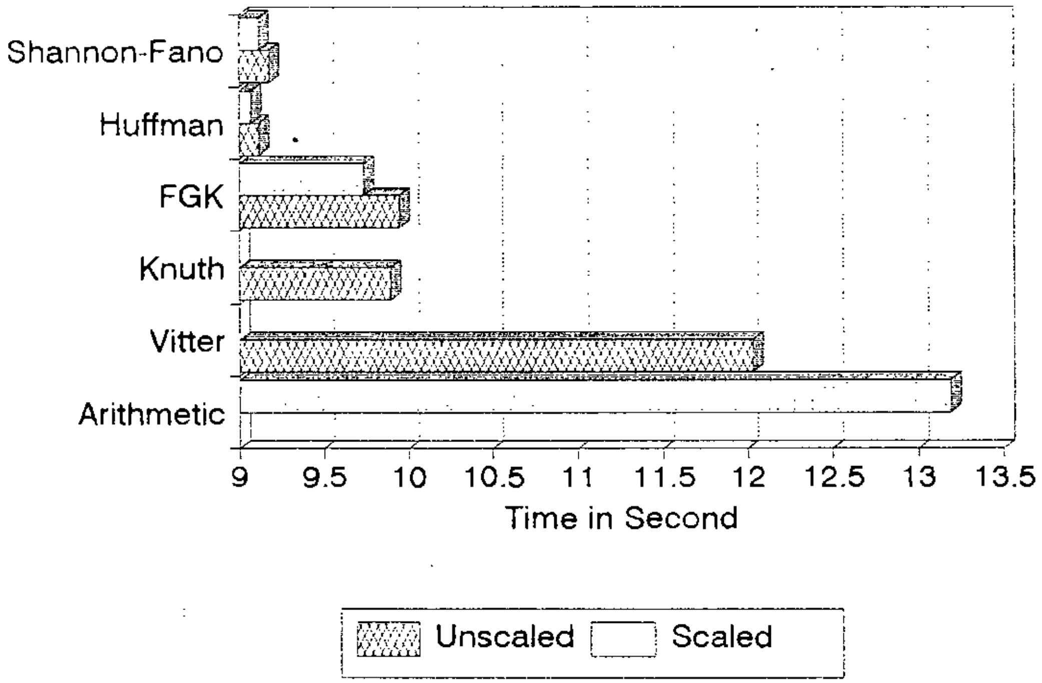
Graph 6.4:Compression Efficiency
(10000 byte BSCII format general text)



Graph 6.5:Compression Time
(10000 byte BSCII format general text)



Graph 6.6:Decompression Time
(10000 byte BSCII format general text)

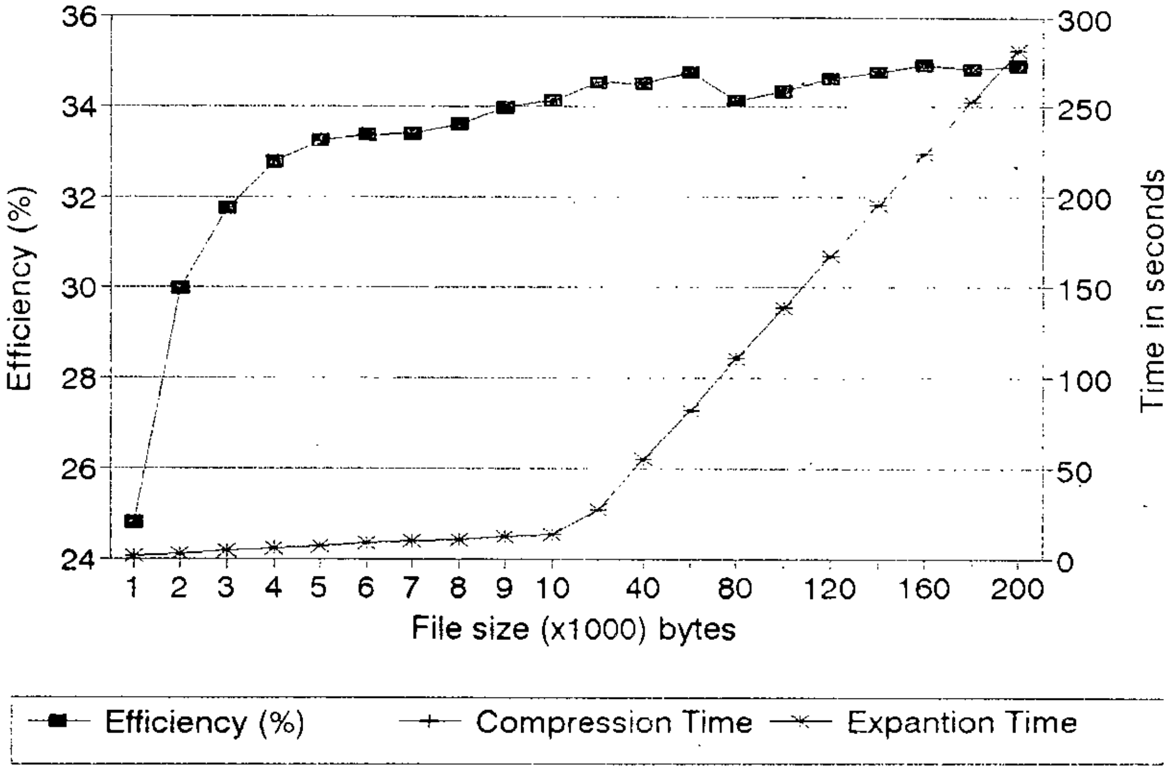


scaling the symbol counts for BSCII format general text. From Table 6.3(a) and 6.3(b) and Graph 6.5 and 6.6, coding and decoding times of all algorithms have been found faster by scaling symbol counts for all text formats both for general and specific text.

Variation of compression efficiency, coding and decoding times of a specific algorithm (Arithmetic coding) with different file size for a BSCII format general Bangla text has been shown in Graph 6.7. The variation of compression efficiency, coding and decoding times with file length for different text formats of the mentioned coding technique are given in Graph 6.8 to 6.10. And these variations for different coding techniques for general text in BSCII format are shown in Graph 6.11 to 6.13. It has been shown from these Graphs that the compression efficiency varies very slowly after 4kb file size and both coding and decoding times increase very fast after 10kb file size and their variation are linear. These results from the implementation of different techniques with varying file length has been shown in Table 6.4 to 6.57. Table 6.4 to 6.9 shows the result of Shannon-Fano algorithm with scaled symbol counts and Table 6.10 to 6.15 shows the results of the same algorithm with unscaled symbol counts. It has been found from the Table 6.4 to 6.9 that BSCII format

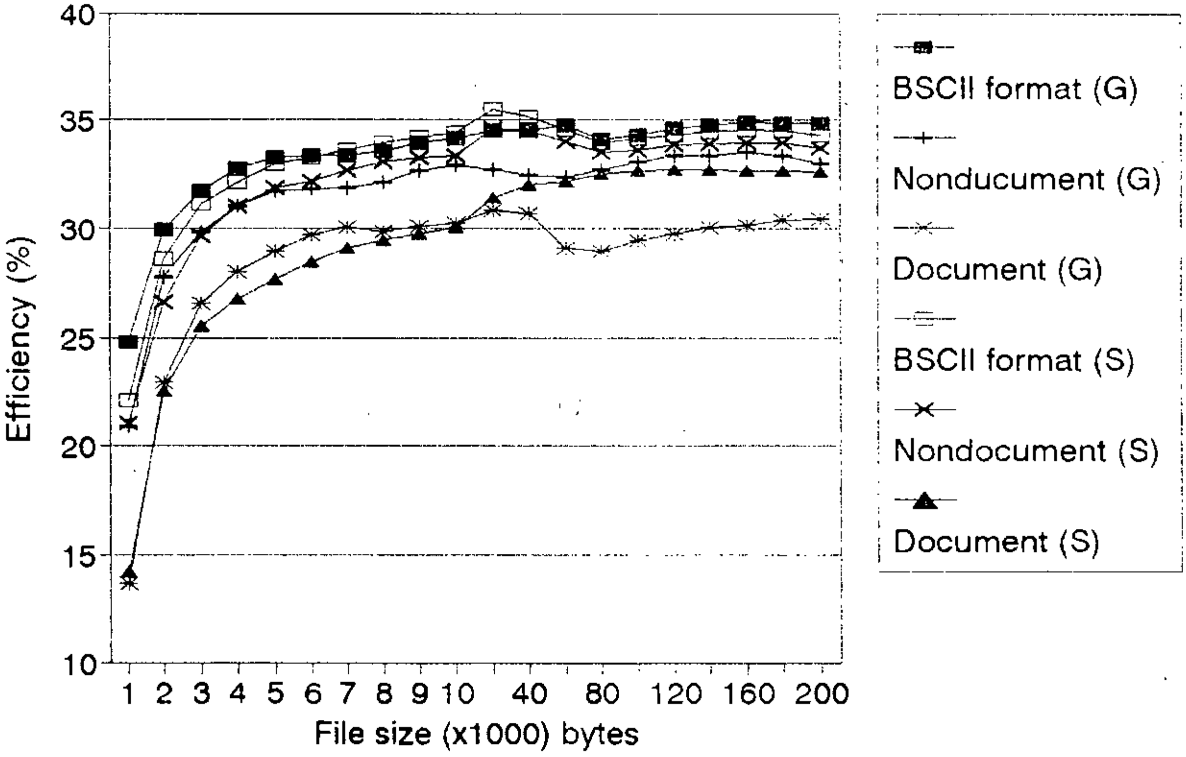
Graph 6.7: Arithmetic Coding

(Static 0-order model with scaling)



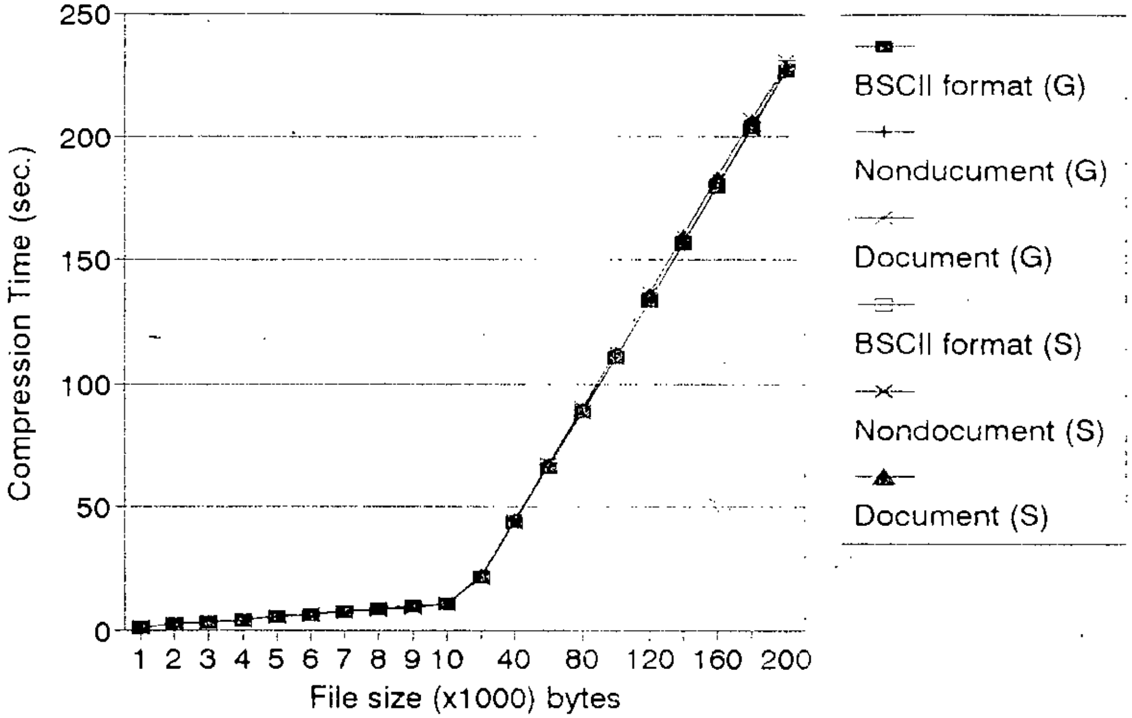
Graph 6.8: Arithmetic Coding Efficiency

(Static 0-order model with scaling)



Graph 6.9: Arithmetic Coding Time

(Static 0-order model with scaling)



Graph 6.10: Arithmetic Decoding Time

(Static 0-order model with scaling)

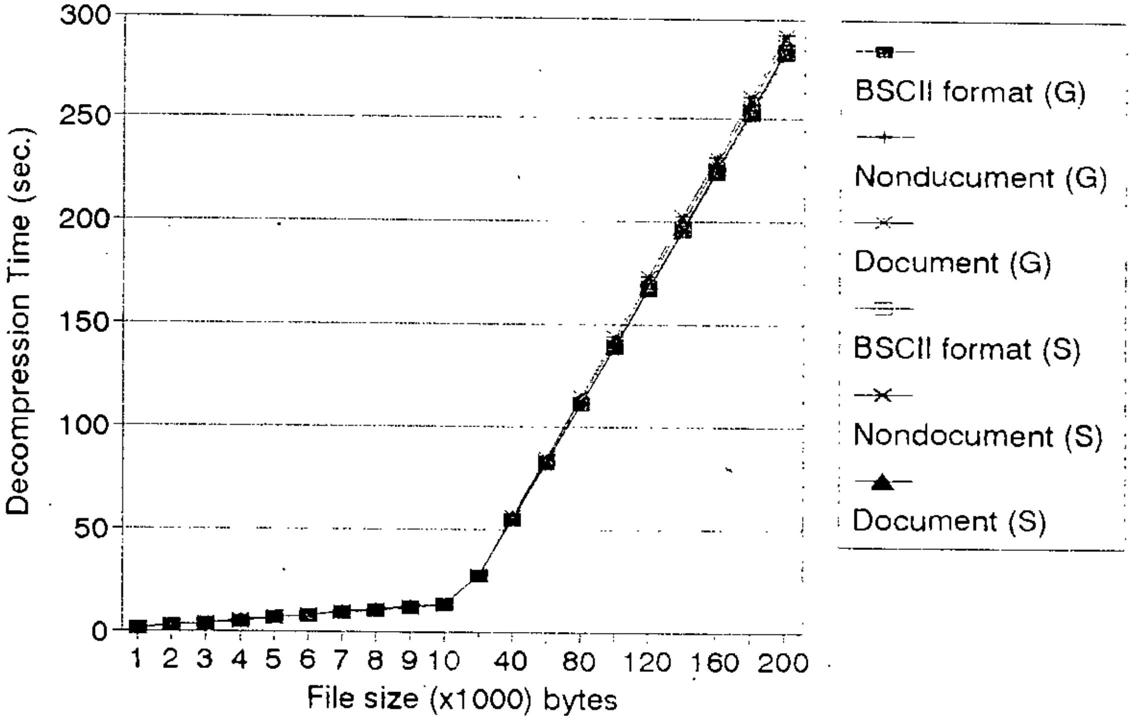


Table 6.4 :: Techniques: Shannon-Fano.
Scaled : Yes.
Model : Static 0-order model.
File type : STD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	760	114	646	24.00%	0.99	0.93
2000	1431	125	1306	28.45%	1.87	1.87
3000	2100	125	1975	30.00%	2.75	2.80
4000	2747	125	2622	31.32%	3.68	3.68
5000	3410	126	3284	31.80%	4.56	4.56
6000	4072	129	3943	32.13%	5.44	5.55
7000	4744	134	4610	32.23%	6.43	6.43
8000	5398	134	5264	32.52%	7.25	7.42
9000	6039	134	5905	32.90%	8.19	8.19
10000	6684	134	6550	33.16%	9.07	9.12
20000	13348	140	13208	33.26%	18.52	18.68
40000	26615	144	26471	33.46%	37.69	37.91
60000	39871	144	39727	33.55%	56.54	57.09
80000	53574	150	53424	33.03%	75.93	76.65
100000	67051	150	66901	32.95%	94.78	96.21
120000	80055	150	79905	33.29%	114.40	116.04
140000	93092	153	92939	33.51%	133.74	135.77
160000	106499	153	106346	33.44%	153.79	155.99
180000	119846	156	119690	33.42%	174.07	176.59
200000	132870	156	132714	33.56%	193.85	197.31

Table 6.5 :: Techniques: Shannon-Fano.
Scaled : Yes.
Model : Static 0-order model.
File type : XFR

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	802	151	651	19.80%	1.04	0.99
2000	1479	164	1315	26.05%	1.92	1.92
3000	2155	170	1985	28.17%	2.75	2.80
4000	2822	173	2649	29.45%	3.68	3.74
5000	3505	174	3331	29.90%	4.62	4.67
6000	4184	183	4001	30.27%	5.55	5.60
7000	4871	191	4680	30.41%	6.43	6.48
8000	5533	193	5340	30.84%	7.36	7.42
9000	6213	193	6020	30.97%	8.30	8.30
10000	6860	193	6667	31.40%	9.18	9.23
20000	13655	207	13448	31.73%	18.63	18.85
40000	27210	222	26988	31.98%	37.80	38.41
60000	41223	225	40998	31.30%	57.20	58.24
80000	54623	225	54398	31.72%	76.21	77.53
100000	67859	226	67633	32.14%	95.22	96.81
120000	82120	226	81894	31.57%	115.38	117.42
140000	95556	228	95328	31.75%	134.89	137.58
160000	109050	228	108822	31.84%	155.16	158.35
180000	123033	229	122804	31.65%	175.77	179.23
200000	137396	234	137162	31.30%	196.43	200.55

Table 6.6 :: Techniques: Shannon-Fano.
Scaled : Yes.
Model : Static 0-order model.
File type : BNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	870	187	683	13.00%	1.04	1.10
2000	1556	195	1361	22.20%	1.98	1.98
3000	2265	197	2068	24.50%	2.91	2.91
4000	2911	201	2710	27.23%	3.79	3.79
5000	3657	204	3453	26.86%	4.78	4.73
6000	4364	204	4160	27.27%	5.71	5.71
7000	5062	207	4855	27.69%	6.54	6.70
8000	5793	215	5578	27.59%	7.58	7.58
9000	6504	216	6288	27.73%	8.46	8.57
10000	7089	221	6868	29.11%	9.29	9.45
20000	14072	229	13843	29.64%	18.96	19.18
40000	28150	244	27906	29.62%	38.46	38.96
60000	43653	258	43395	27.25%	59.01	60.59
80000	58748	258	58490	26.57%	79.23	81.26
100000	72611	258	72353	27.39%	98.35	108.52
120000	87050	258	86792	27.46%	118.79	121.92
140000	102750	258	102492	26.61%	139.89	144.84
160000	116157	259	115898	27.40%	160.11	165.38
180000	130160	259	129901	27.69%	181.21	186.76
200000	144331	259	144072	27.83%	201.43	208.02

Table 6.7 :: Techniques: Shannon-Fano.
Scaled : Yes.
Model : Static 0-order model.
File type : QSTD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	795	134	661	20.50%	1.04	0.99
2000	1466	137	1329	26.70%	1.92	1.92
3000	2122	137	1985	29.27%	2.80	2.80
4000	2788	137	2651	30.30%	3.68	3.74
5000	3444	137	3307	31.12%	4.62	4.62
6000	4106	140	3966	31.57%	5.49	5.55
7000	4764	141	4623	31.94%	6.43	6.43
8000	5417	143	5274	32.29%	7.42	7.36
9000	6071	143	5928	32.54%	8.24	8.24
10000	6720	143	6577	32.80%	9.18	9.18
20000	13208	143	13065	33.96%	18.63	18.57
40000	26623	145	26478	33.44%	38.08	37.91
60000	40332	145	40187	32.78%	57.47	57.53
80000	54354	146	54208	32.06%	76.81	77.42
100000	67724	146	67578	32.28%	95.27	96.81
120000	81094	146	80948	32.42%	114.89	116.76
140000	94732	146	94586	32.33%	134.84	137.14
160000	108025	146	107879	32.48%	155.00	157.64
180000	121277	146	121131	32.62%	175.44	178.08
200000	135291	146	135145	32.35%	195.77	199.34

Table 6.8 :: Techniques: Shannon-Fano.
Scaled : Yes.
Model : Static 0-order model.
File type : QXFR

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	803	146	657	19.70%	1.04	0.99
2000	1504	169	1335	24.80%	1.87	1.92
3000	2168	170	1998	27.73%	2.86	2.86
4000	2860	171	2689	28.50%	3.74	3.74
5000	3501	177	3324	29.98%	4.62	4.67
6000	4180	184	3996	30.33%	5.55	5.55
7000	4888	187	4701	30.17%	6.43	6.54
8000	5563	192	5371	30.46%	7.42	7.47
9000	6249	201	6048	30.57%	8.30	8.41
10000	6834	203	6631	31.66%	9.12	9.18
20000	13416	208	13208	32.92%	18.52	18.74
40000	26974	215	26759	32.56%	37.75	38.19
60000	40712	215	40497	32.15%	57.03	57.97
80000	54523	221	54302	31.85%	76.26	77.80
100000	68468	221	68247	31.53%	95.71	97.53
120000	81775	221	81554	31.85%	115.88	117.47
140000	95293	221	95072	31.93%	135.33	137.75
160000	108851	221	108630	31.97%	155.49	158.24
180000	122474	221	122253	31.96%	175.99	179.12
200000	136617	221	136396	31.69%	196.48	200.49

Table 6.9 :: Techniques: Shannon-Fano.
Scaled : Yes.
Model : Static 0-order model.
File type : QBNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	865	177	688	13.50%	1.10	1.04
2000	1575	199	1376	21.25%	1.98	2.03
3000	2274	207	2067	24.20%	2.97	2.97
4000	2975	212	2763	25.62%	3.85	3.90
5000	3682	220	3462	26.36%	4.84	4.84
6000	4349	222	4127	27.52%	5.77	5.77
7000	5040	223	4817	28.00%	6.70	6.70
8000	5737	223	5514	28.29%	7.69	7.58
9000	6398	223	6175	28.91%	8.57	8.52
10000	7116	225	6891	28.84%	9.45	9.45
20000	14007	229	13778	29.96%	19.23	19.12
40000	27782	238	27544	30.55%	38.90	38.74
60000	41413	240	41173	30.98%	58.24	58.35
80000	55061	240	54821	31.17%	77.86	78.24
100000	68675	244	68431	31.32%	97.64	97.80
120000	82055	246	81809	31.62%	117.80	117.75
140000	95900	247	95653	31.50%	138.24	138.08
160000	109813	247	109566	31.37%	159.01	159.23
180000	123695	248	123447	31.28%	179.51	180.11
200000	137421	248	137173	31.29%	197.75	200.99

Table 6.10:: Techniques: Shannon-Fano.
Scaled : No.
Model : Static 0-order model.
File type : STD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1051	405	646	-5.10%	1.04	0.99
2000	1740	437	1303	13.00%	1.98	1.87
3000	2406	437	1969	19.80%	2.86	2.86
4000	3056	437	2619	23.60%	3.74	3.74
5000	3724	441	3283	25.52%	4.73	4.67
6000	4420	465	3955	26.33%	5.60	5.60
7000	5091	479	4612	27.27%	6.54	6.48
8000	5745	479	5266	28.19%	7.47	7.36
9000	6381	479	5902	29.10%	8.35	8.30
10000	7039	479	6560	29.61%	9.23	9.18
20000	13692	497	13195	31.54%	18.90	18.79
40000	27055	525	26530	32.36%	38.19	38.30
60000	40306	525	39781	32.82%	57.47	57.58
80000	54183	543	53640	32.27%	77.31	77.47
100000	67385	543	66842	32.62%	96.65	96.70
120000	80353	543	79810	33.04%	116.54	116.65
140000	93488	555	92933	33.22%	136.26	136.65
160000	106322	555	105767	33.55%	155.93	156.65
180000	119846	585	119261	33.42%	176.65	177.80
200000	132692	585	132107	33.65%	197.09	197.80

Table 6.11:: Techniques: Shannon-Fano.
Scaled : No.
Model : Static 0-order model.
File type : XFR

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1150	499	651	-15.00%	1.04	1.04
2000	1865	551	1314	6.75%	1.98	1.98
3000	2558	581	1977	14.73%	2.86	2.86
4000	3227	587	2640	19.32%	3.85	3.79
5000	3912	591	3321	21.76%	4.73	4.73
6000	4616	621	3995	23.07%	5.66	5.66
7000	5329	659	4670	23.87%	6.59	6.59
8000	5993	673	5320	25.09%	7.53	7.53
9000	6669	673	5996	25.90%	8.46	8.46
10000	7324	673	6651	26.76%	9.40	9.34
20000	14149	729	13420	29.25%	19.01	18.96
40000	28273	819	27454	29.32%	38.85	38.90
60000	41843	831	41012	30.26%	58.24	58.52
80000	55266	831	54435	30.92%	77.69	78.13
100000	68884	835	68049	31.12%	97.20	97.47
120000	82180	835	81345	31.52%	117.25	117.75
140000	95676	855	94821	31.66%	137.36	138.08
160000	109105	855	108250	31.81%	158.24	158.57
180000	122866	865	122001	31.74%	178.52	179.67
200000	137535	891	136644	31.23%	199.62	201.37

Table 6.12 :: Techniques: Shannon-Fano.
Scaled : No.
Model : Static 0-order model.
File type : BNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1320	637	683	-32.00%	1.15	1.10
2000	2024	663	1361	-1.20%	2.03	2.03
3000	2743	677	2066	8.57%	3.02	2.97
4000	3410	699	2711	14.75%	3.96	3.85
5000	4157	705	3452	16.86%	4.95	4.84
6000	4864	705	4159	18.93%	5.82	5.82
7000	5568	711	4857	20.46%	6.76	6.76
8000	6329	743	5586	20.89%	7.75	7.69
9000	7036	747	6289	21.82%	8.68	8.68
10000	7654	785	6869	23.46%	9.67	9.56
20000	14677	829	13848	26.61%	19.18	19.34
40000	28813	919	27894	27.97%	38.68	39.62
60000	42175	923	41252	29.74%	59.12	58.96
80000	55718	923	54795	30.38%	78.74	78.85
100000	69562	925	68637	30.44%	98.90	98.63
120000	83154	933	82221	30.70%	119.18	119.01
140000	96805	937	95868	30.85%	139.62	139.56
160000	110578	937	109641	30.89%	160.60	160.38
180000	124316	940	123374	30.94%	180.93	173.11
200000	138069	940	137127	30.97%	198.68	194.80

Table 6.13 :: Techniques: Shannon-Fano.
Scaled : No.
Model : Static 0-order model.
File type : QSTD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1134	473	661	-13.40%	1.10	1.04
2000	1818	491	1327	9.10%	1.98	1.98
3000	2480	491	1989	17.33%	2.91	2.86
4000	3145	491	2654	21.38%	3.79	3.79
5000	3801	491	3310	23.98%	4.73	4.67
6000	4462	497	3965	25.63%	5.66	5.60
7000	5135	507	4628	26.64%	6.48	6.54
8000	5789	515	5274	27.64%	7.42	7.47
9000	6439	515	5924	28.46%	8.35	8.41
10000	7104	515	6589	28.96%	9.29	9.23
20000	13612	515	13097	31.94%	18.79	18.74
40000	26986	529	26457	32.53%	38.24	38.35
60000	40656	529	40127	32.24%	58.02	58.02
80000	54649	533	54116	31.69%	76.81	78.13
100000	68006	533	67473	31.99%	94.95	97.86
120000	81475	533	80942	32.10%	115.11	118.13
140000	94843	533	94310	32.26%	134.89	138.35
160000	107888	533	107355	32.57%	154.95	158.79
180000	121377	533	120844	32.57%	174.45	179.45
200000	135546	533	135013	32.23%	195.71	201.04

Table 6.14 :: Techniques: Shannon-Fano.
Scaled : No.
Model : Static 0-order model.
File type : QXFR

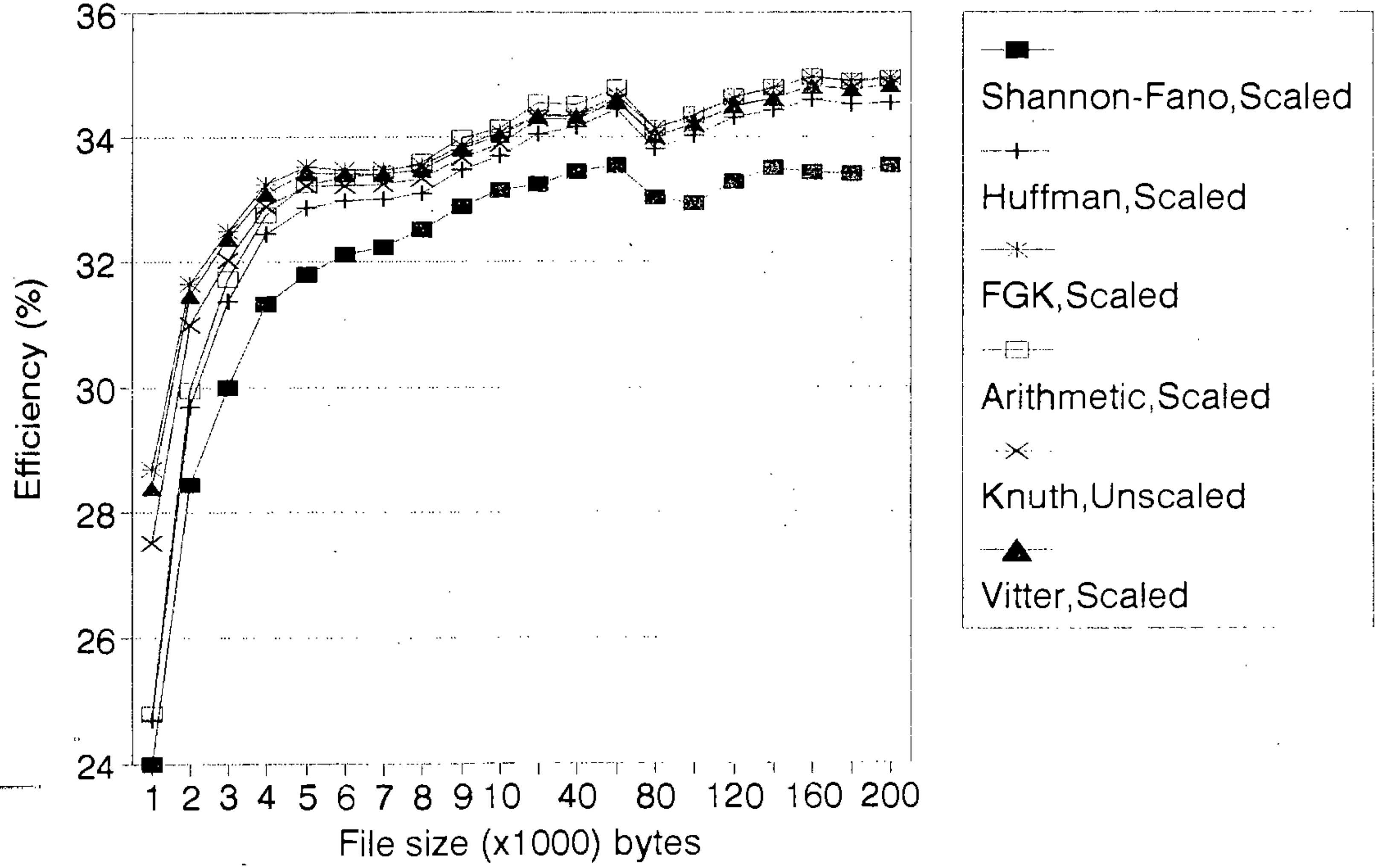
Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1136	479	657	-13.60%	1.04	1.04
2000	1893	559	1334	5.35%	1.98	1.98
3000	2559	563	1996	14.70%	2.86	2.91
4000	3257	573	2684	18.57%	3.79	3.85
5000	3916	597	3319	21.68%	4.67	4.73
6000	4614	625	3989	23.10%	5.55	5.82
7000	5351	649	4702	23.56%	6.43	6.65
8000	6027	669	5358	24.66%	7.42	7.47
9000	6650	705	5945	26.11%	8.24	8.46
10000	7339	725	6614	26.61%	9.12	9.34
20000	13931	745	13186	30.34%	18.46	19.01
40000	27463	785	26678	31.34%	37.47	38.52
60000	41217	785	40432	31.30%	56.54	58.41
80000	55158	821	54337	31.05%	76.04	78.46
100000	68588	821	67767	31.41%	94.73	98.30
120000	81986	821	81165	31.68%	115.71	118.30
140000	95515	821	94694	31.77%	135.49	138.63
160000	108859	821	108038	31.96%	155.55	160.00
180000	122545	821	121724	31.92%	175.60	179.89
200000	136488	821	135667	31.76%	196.32	201.26

Table 6.15 :: Techniques: Shannon-Fano.
Scaled : No.
Model : Static 0-order model.
File type : QBNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1285	597	688	-28.50%	1.15	1.15
2000	2049	673	1376	-2.45%	2.09	2.03
3000	2771	705	2066	7.63%	3.08	3.02
4000	3496	725	2771	12.60%	4.07	3.96
5000	4227	769	3458	15.46%	5.00	4.95
6000	4926	783	4143	17.90%	5.93	5.82
7000	5612	793	4819	19.83%	6.81	6.81
8000	6311	793	5518	21.11%	7.75	7.75
9000	6996	793	6203	22.27%	8.79	8.68
10000	7694	801	6893	23.06%	9.62	9.62
20000	14608	829	13779	26.96%	19.56	19.40
40000	28433	889	27544	28.92%	39.45	39.01
60000	42155	903	41252	29.74%	59.12	58.96
80000	55698	903	54795	30.38%	78.74	78.85
100000	69562	925	68637	30.44%	98.90	98.63
120000	83154	933	82221	30.70%	119.18	119.01
140000	96805	937	95868	30.85%	139.62	139.56
160000	110578	937	109641	30.89%	160.60	160.38
180000	124315	941	123374	30.94%	180.93	175.11
200000	138068	941	137127	30.97%	198.68	194.89

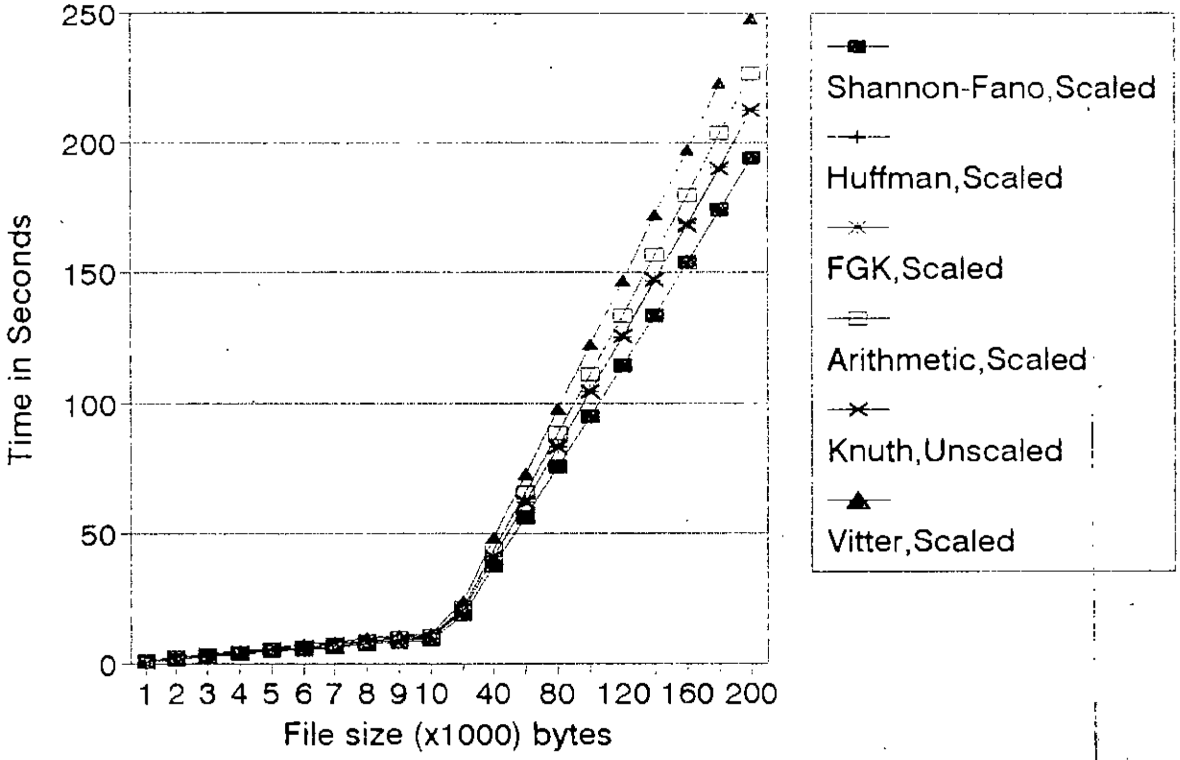
Graph 6.11:Compression Efficiency

(BSCII format general text)



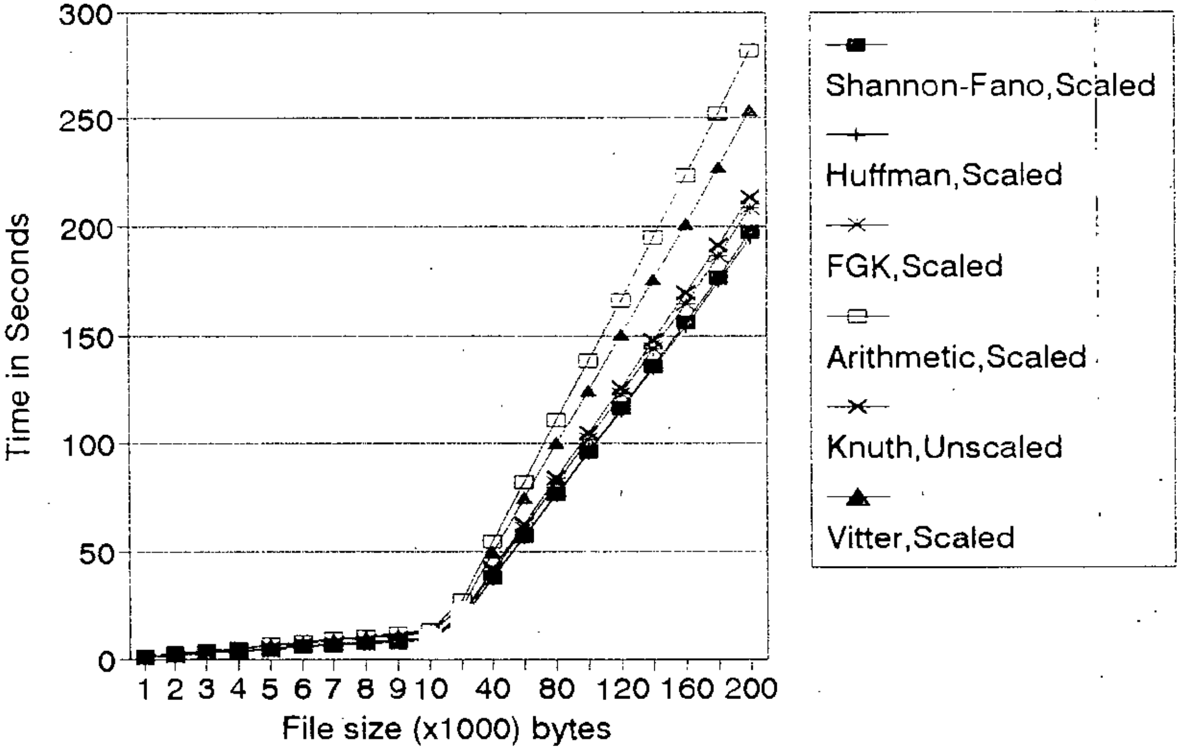
Graph 6.12:Compression Time

(BSCII format general text)



Graph 6.13:Decompression Time

(BSCII format general text)



text would be the efficient text format for this algorithm and the BNA format text is the most inefficient text format. The efficiency varies from 24.00% to 33.56% for BSCII format general texts and from 20.50% to 32.35% for specific texts whereas the variation of efficiency for BNA format general texts can be found to vary from 13.00% to 27.83% for general texts and from 13.50% to 31.29% for specific texts. Coding times vary from 0.99 sec. to 193.85 sec. for general BSCII text format and from 1.04 sec. to 201.43 sec. for general BNA format text. Decoding times also vary from 0.93 sec. to 197.31 sec for general BSCII format texts and from 1.10 sec. to 208.02 sec. for general BNA format texts. The decoding times have been found less for small files and more for large files than coding time for all texts formats both in general and specific texts. Similar relations of coding efficiency, coding and decoding times have been found for the same algorithms with unscaled symbol counts from Tables 6.10 to 6.15. Coding efficiency has been found negative for small size files with unscaled symbol counts.

The results of static Huffman algorithm have been given in Tables 6.16 to 6.21 with scaled symbol counts and in Tables 6.22 to 6.27 with unscaled symbol counts for different text formats and text types. The variations of coding efficiency, coding and decoding times have been found similar to the Shannon-Fano algorithm. BSCII text format has been found the

Table 6.16 :: Techniques: Huffman Algorithm.
Scaled : Yes.
Model : Static 0-order model.
File type : STD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	753	114	639	24.70%	0.99	0.99
2000	1406	125	1281	29.70%	1.87	1.87
3000	2059	125	1934	31.37%	2.80	2.86
4000	2702	125	2577	32.45%	3.74	3.68
5000	3356	126	3230	32.88%	4.62	4.56
6000	4020	129	3891	33.00%	5.55	5.44
7000	4689	134	4555	33.01%	6.48	6.37
8000	5350	134	5216	33.12%	7.36	7.31
9000	5985	134	5851	33.50%	8.13	8.13
10000	6630	134	6496	33.70%	9.07	9.07
20000	13190	140	13050	34.05%	18.52	18.52
40000	26330	144	26186	34.17%	37.80	37.42
60000	39323	144	39179	34.46%	56.59	56.54
80000	52937	150	52787	33.83%	75.77	76.21
100000	65982	150	65832	34.02%	94.84	95.33
120000	78826	150	78676	34.31%	114.29	114.78
140000	91782	153	91629	34.44%	133.74	134.62
160000	104637	153	104484	34.60%	153.46	154.45
180000	117862	156	117706	34.52%	173.68	174.89
200000	130883	156	130727	34.56%	193.57	195.05

Table 6.17 :: Techniques: Huffman Algorithm.
Scaled : Yes.
Model : Static 0-order model.
File type : XFR

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	793	151	642	20.70%	0.99	1.04
2000	1451	164	1287	27.45%	1.92	1.92
3000	2117	170	1947	29.43%	2.80	2.86
4000	2772	173	2599	30.70%	3.74	3.68
5000	3434	174	3260	31.32%	4.67	4.56
6000	4115	183	3932	31.42%	5.49	5.60
7000	4800	191	4609	31.43%	6.48	6.54
8000	5466	193	5273	31.68%	7.42	7.36
9000	6107	193	5914	32.14%	8.30	8.24
10000	6758	193	6565	32.42%	9.18	9.18
20000	13547	207	13340	32.27%	18.74	18.79
40000	27268	222	27046	31.83%	38.08	38.30
60000	40887	225	40662	31.86%	57.42	57.86
80000	54184	225	53959	32.27%	76.37	77.31
100000	67256	226	67030	32.74%	95.44	96.37
120000	80789	226	80563	32.68%	115.11	116.32
140000	94156	228	93928	32.75%	134.73	136.32
160000	106917	228	106689	33.18%	154.34	156.10
180000	120482	229	120253	33.07%	174.84	176.92
200000	134655	234	134421	32.67%	195.49	197.86

Table 6.18 :: Techniques: Huffman Algorithm.
Scaled : Yes.
Model : Static 0-order model.
File type : BNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	864	187	677	13.60%	1.10	1.10
2000	1543	195	1348	22.85%	2.03	1.98
3000	2209	197	2012	26.37%	2.91	2.91
4000	2888	201	2687	27.80%	3.85	3.85
5000	3561	204	3357	28.78%	4.73	4.73
6000	4231	204	4027	29.48%	5.71	5.66
7000	4914	207	4707	29.80%	6.59	6.59
8000	5627	215	5412	29.66%	7.53	7.53
9000	6314	216	6098	29.84%	8.46	8.46
10000	7007	221	6786	29.93%	9.51	9.40
20000	13890	229	13661	30.55%	19.01	19.01
40000	27823	244	27579	30.44%	38.46	38.68
60000	42716	258	42458	28.81%	58.74	59.23
80000	57075	258	56817	28.66%	78.63	79.56
100000	70912	258	70654	29.09%	97.80	99.40
120000	84659	258	84401	29.45%	117.80	119.56
140000	98426	258	98168	29.70%	137.91	139.84
160000	112202	259	111943	29.87%	158.35	160.77
180000	125844	259	125585	30.09%	178.96	181.65
200000	139658	259	139399	30.17%	199.29	202.36

Table 6.19 :: Techniques: Huffman Algorithm.
Scaled : Yes.
Model : Static 0-order model.
File type : QSTD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	780	134	646	22.00%	1.10	1.04
2000	1433	137	1296	28.35%	1.87	1.87
3000	2077	137	1940	30.77%	2.80	2.80
4000	2728	137	2591	31.80%	3.79	3.63
5000	3367	137	3230	32.66%	4.62	4.56
6000	4026	140	3886	32.90%	5.49	5.44
7000	4667	141	4526	33.33%	6.48	6.32
8000	5305	143	5162	33.69%	7.31	7.31
9000	5954	143	5811	33.84%	8.24	8.19
10000	6602	143	6459	33.98%	9.12	9.01
20000	12981	143	12838	35.09%	18.52	18.30
40000	26054	145	25909	34.87%	37.69	37.31
60000	39445	145	39300	34.26%	56.87	56.48
80000	53088	146	52942	33.64%	76.10	76.04
100000	66209	146	66063	33.79%	94.18	95.05
120000	79207	146	79061	33.99%	113.41	114.78
140000	92270	146	92124	34.09%	132.80	134.45
160000	105234	146	105088	34.23%	152.64	154.23
180000	118429	146	118283	34.21%	172.64	174.56
200000	132033	146	131887	33.98%	193.02	195.22

Table 6.20 :: Techniques: Huffman Algorithm.
Scaled : Yes.
Model : Static 0-order model.
File type : QXFR

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	790	146	644	21.00%	1.10	0.99
2000	1470	169	1301	26.50%	1.92	1.98
3000	2119	170	1949	29.37%	2.80	2.86
4000	2767	171	2596	30.82%	3.74	3.74
5000	3416	177	3239	31.68%	4.62	4.67
6000	4086	184	3902	31.90%	5.49	5.55
7000	4734	187	4547	32.37%	6.37	6.43
8000	5372	192	5180	32.85%	7.31	7.31
9000	6039	201	5838	32.90%	8.13	8.19
10000	6698	203	6495	33.02%	9.12	9.12
20000	13147	208	12939	34.27%	18.30	18.41
40000	26271	215	26056	34.32%	37.20	37.53
60000	39706	215	39491	33.82%	56.15	56.87
80000	53388	221	53167	33.27%	75.38	76.48
100000	66602	221	66381	33.40%	94.23	95.55
120000	79666	221	79445	33.61%	113.57	115.44
140000	92794	221	92573	33.72%	133.19	135.05
160000	105904	221	105683	33.81%	152.97	156.10
180000	119256	221	119035	33.75%	173.08	175.27
200000	132974	221	132753	33.51%	193.24	196.10

Table 6.21 :: Techniques: Huffman Algorithm.
Scaled : Yes.
Model : Static 0-order model.
File type : QBNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	859	177	682	14.10%	1.15	1.10
2000	1554	199	1355	22.30%	2.09	2.09
3000	2243	207	2036	25.23%	3.08	2.97
4000	2939	212	2727	26.52%	3.96	3.90
5000	3629	220	3409	27.42%	4.89	4.84
6000	4313	222	4091	28.12%	5.82	5.77
7000	4984	223	4761	28.80%	6.76	6.70
8000	5672	223	5449	29.10%	7.64	7.58
9000	6347	223	6124	29.48%	8.63	8.52
10000	7032	225	6807	29.68%	9.51	9.45
20000	13790	229	13561	31.05%	19.23	18.90
40000	27399	238	27161	31.50%	38.79	38.35
60000	40966	240	40726	31.72%	58.24	57.75
80000	54356	240	54116	32.05%	77.64	77.14
100000	67815	244	67571	32.19%	97.31	96.32
120000	81280	246	81034	32.27%	117.25	116.26
140000	94845	247	94598	32.25%	137.53	136.48
160000	108471	247	108224	32.21%	158.13	156.81
180000	122129	248	121881	32.15%	178.30	177.25
200000	135765	248	135517	32.12%	196.48	198.08

Table 6.22 :: Techniques: Huffman Algorithm.
Scaled : No.
Model : Static 0-order model.
File type : STD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1044	405	639	-4.40%	1.04	0.99
2000	1717	437	1280	14.15%	1.98	1.92
3000	2369	437	1932	21.03%	2.86	2.86
4000	3011	437	2574	24.73%	3.74	3.74
5000	3665	441	3224	26.70%	4.73	4.67
6000	4351	465	3886	27.48%	5.60	5.55
7000	5026	479	4547	28.20%	6.54	6.48
8000	5685	479	5206	28.94%	7.42	7.42
9000	6318	479	5839	29.80%	8.30	8.35
10000	6958	479	6479	30.42%	9.18	9.12
20000	13494	497	12997	32.53%	18.57	18.68
40000	26644	525	26119	33.39%	37.64	37.97
60000	39615	525	39090	33.98%	56.54	56.92
80000	53163	543	52620	33.55%	76.04	76.70
100000	66164	543	65621	33.84%	95.05	95.82
120000	78967	543	78424	34.19%	114.67	115.82
140000	91899	555	91344	34.36%	134.23	135.66
160000	104957	564	104393	34.40%	154.05	155.04
180000	118006	564	117442	34.44%	173.97	175.02
200000	131055	564	130491	34.47%	194.01	195.80

Table 6.23 :: Techniques: Huffman Algorithm.
Scaled : No.
Model : Static 0-order model.
File type : XFR

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1141	499	642	-14.10%	1.10	1.04
2000	1836	551	1285	8.20%	1.98	1.98
3000	2525	581	1944	15.83%	2.91	2.86
4000	3181	587	2594	20.48%	3.79	3.85
5000	3841	591	3250	23.18%	4.67	4.73
6000	4542	621	3921	24.30%	5.60	5.66
7000	5252	659	4593	24.97%	6.54	6.70
8000	5925	673	5252	25.94%	7.47	7.53
9000	6563	673	5890	27.08%	8.35	8.41
10000	7212	673	6539	27.88%	9.29	9.40
20000	13994	729	13265	30.03%	18.79	18.85
40000	27602	819	26783	31.00%	38.19	38.68
60000	41141	831	40310	31.43%	57.36	58.24
80000	54331	831	53500	32.09%	76.54	77.75
100000	67361	835	66526	32.64%	95.55	97.03
120000	80437	835	79602	32.97%	115.38	117.03
140000	93676	855	92821	33.09%	134.95	137.09
160000	106815	855	105960	33.24%	154.62	157.42
180000	120248	865	119383	33.20%	175.00	178.46
200000	133520	873	132647	33.24%	195.74	198.28

Table 6.24 :: Techniques: Huffman Algorithm.
Scaled : No.
Model : Static 0-order model.
File type : BNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1314	637	677	-31.40%	1.21	1.15
2000	2011	663	1348	-0.55%	2.14	2.03
3000	2686	677	2009	10.47%	3.08	3.02
4000	3383	699	2684	15.43%	3.96	3.96
5000	4056	705	3351	18.88%	4.89	4.84
6000	4727	705	4022	21.22%	5.77	5.82
7000	5409	711	4698	22.73%	6.70	6.76
8000	6148	743	5405	23.15%	7.64	7.69
9000	6834	747	6087	24.07%	8.68	8.63
10000	7558	785	6773	24.42%	9.56	9.62
20000	14447	829	13618	27.77%	19.18	19.45
40000	28372	919	27453	29.07%	38.85	39.12
60000	43017	1017	42000	28.30%	59.07	59.73
80000	57203	1017	56186	28.50%	78.52	79.89
100000	70827	1017	69810	29.17%	98.08	99.78
120000	84448	1017	83431	29.63%	117.86	119.89
140000	98357	1021	97336	29.74%	138.06	139.87
160000	112262	1021	111241	29.83%	159.14	161.35
180000	126167	1021	125146	29.90%	179.39	182.13
200000	140072	1021	139051	29.96%	201.43	203.11

Table 6.25 :: Techniques: Huffman Algorithm.
Scaled : No.
Model : Static 0-order model.
File type : QSTD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1119	473	646	-11.90%	1.10	1.04
2000	1786	491	1295	10.70%	1.98	1.98
3000	2429	491	1938	19.03%	2.91	2.86
4000	3077	491	2586	23.07%	3.79	3.79
5000	3719	491	3228	25.62%	4.78	4.67
6000	4375	497	3878	27.08%	5.66	5.60
7000	5024	507	4517	28.23%	6.54	6.48
8000	5666	515	5151	29.18%	7.47	7.36
9000	6308	515	5793	29.91%	8.30	8.24
10000	6953	515	6438	30.47%	9.23	9.18
20000	13315	515	12800	33.42%	18.74	18.57
40000	26380	529	25851	34.05%	38.02	37.64
60000	39739	529	39210	33.77%	57.36	57.14
80000	53332	533	52799	33.34%	76.59	76.87
100000	66377	533	65844	33.62%	94.84	96.26
120000	79360	533	78827	33.87%	114.73	115.93
140000	92342	533	91809	34.04%	134.56	135.93
160000	105459	535	104924	34.08%	153.78	155.34
180000	118574	535	118039	34.12%	173.00	174.76
200000	131689	535	131154	34.15%	194.22	195.58

Table 6.26 :: Techniques: Huffman Algorithm.
Scaled : No.
Model : Static 0-order model.
File type : QXFR

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1123	479	644	-12.30%	1.10	1.04
2000	1859	559	1300	7.05%	1.98	2.03
3000	2509	563	1946	16.37%	2.91	2.91
4000	3163	573	2590	20.93%	3.79	3.85
5000	3833	597	3236	23.34%	4.67	4.73
6000	4515	625	3890	24.75%	5.60	5.66
7000	5181	649	4532	25.99%	6.48	6.48
8000	5837	669	5168	27.04%	7.31	7.42
9000	6513	705	5808	27.63%	8.30	8.35
10000	7185	725	6460	28.15%	9.18	9.29
20000	13609	745	12864	31.95%	18.35	18.85
40000	26721	785	25936	33.20%	37.14	37.97
60000	40079	785	39294	33.20%	56.26	57.75
80000	53177	785	52392	33.52%	75.51	77.00
100000	66304	814	65490	33.69%	94.76	96.25
120000	79402	814	78588	33.83%	114.52	115.70
140000	92589	903	91686	33.86%	134.27	135.75
160000	105687	903	104784	33.94%	153.12	157.00
180000	118785	903	117882	34.00%	174.78	176.25
200000	131883	903	130980	34.05%	194.53	196.50

Table 6.27 :: Techniques: Huffman Algorithm.
Scaled : No.
Model : Static 0-order model.
File type : QBNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	1279	597	682	-27.90%	1.15	1.21
2000	2028	673	1355	-1.40%	2.14	2.09
3000	2737	705	2032	8.77%	3.13	3.02
4000	3450	725	2725	13.75%	4.12	4.01
5000	4173	769	3404	16.54%	5.00	5.00
6000	4867	783	4084	18.88%	5.99	5.88
7000	5541	793	4748	20.84%	6.87	6.76
8000	6230	793	5437	22.12%	7.80	7.69
9000	6902	793	6109	23.31%	8.74	8.74
10000	7588	801	6787	24.12%	9.62	9.62
20000	14477	903	13574	27.61%	19.54	19.24
40000	28051	903	27148	29.87%	39.48	38.48
60000	41625	903	40722	30.62%	58.72	57.72
80000	55217	921	54296	30.97%	77.96	76.96
100000	68791	921	67870	31.20%	98.20	96.80
120000	82379	935	81444	31.35%	117.94	116.64
140000	95953	935	95018	31.46%	138.28	137.48
160000	109531	939	108592	31.54%	158.92	157.32
180000	123105	939	122166	31.60%	179.16	177.89
200000	136679	939	135740	31.66%	197.40	198.90

most efficient format and BNA format the most inefficient format also for this algorithm both for scaled and unscaled symbol counts. Efficiency for this algorithm has been found (about 2.5%) better than the Shannon-Fano algorithm for all text format and text types. Coding and decoding times have been found very little smaller in this algorithm relative to the Shannon-Fano algorithm. The coding efficiency has been found 24.70% to 34.56% for general STD format text and 13.60% to 30.17% for BNA format general text with scaled symbol counts, whereas -4.40% to 34.47% for STD format general text and -31.40% to 29.96% for BNA format general text with unscaled symbol counts. The variation of coding and decoding times have been found from 0.99 sec. to 193.57 sec. and 0.99 sec. to 195.05 sec. respectively for general STD format text whereas the coding and decoding times for BNA format general text have been found from 1.10 sec to 199.29 sec. and 1.10 sec. to 202.36 sec. respectively.

Similar variation of coding and decoding times has been found with unscaled symbol counts.

The results for FGK algorithm have been given in Tables 6.28 to 6.33 for scaled symbol counts and Tables 6.34 to 6.39 for unscaled symbol counts. In these algorithm with scaled symbol counts for general text the coding efficiencies haven found from 28.70% to 34.96% for STD format text and from 22.00% to 31.23% for BNA format texts. With unscaled symbol counts,

Table 6.28 :: Techniques: FGK algorithm.
Scaled : Yes.
Model : Dynamic 0-order model.
File type : STD

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	713	28.70%	1.04	1.04
2000	1367	31.65%	2.03	1.98
3000	2025	32.50%	3.02	2.97
4000	2670	33.25%	4.07	3.96
5000	3323	33.54%	5.11	4.95
6000	3990	33.50%	6.10	5.88
7000	4656	33.49%	7.14	6.81
8000	5316	33.55%	8.08	7.91
9000	5952	33.87%	9.12	8.74
10000	6593	34.07%	10.00	9.73
20000	13127	34.37%	20.38	19.84
40000	26253	34.37%	41.32	40.27
60000	39205	34.66%	62.09	60.60
80000	52691	34.14%	83.57	81.54
100000	65675	34.33%	104.40	101.98
120000	78439	34.63%	125.27	122.86
140000	91301	34.78%	146.59	143.85
160000	104062	34.96%	168.35	164.73
180000	117186	34.90%	190.44	186.76
200000	130077	34.96%	212.53	208.24

Table 6.29 :: Techniques: FGK algorithm.
Scaled : Yes.
Model : Dynamic 0-order model.
File type : XFR

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	722	27.80%	1.04	1.04
2000	1379	31.05%	2.03	2.03
3000	2046	31.80%	3.08	2.97
4000	2699	32.52%	4.07	3.96
5000	3360	32.80%	5.05	4.95
6000	4036	32.73%	6.15	5.93
7000	4724	32.51%	7.14	6.98
8000	5386	32.67%	8.19	7.91
8704	6352	27.02%	9.22	9.07
10000	6678	33.22%	10.16	9.95
20000	13325	33.38%	20.55	20.00
40000	26705	33.24%	41.87	40.55
60000	40006	33.32%	62.97	61.43
80000	53709	32.86%	84.40	82.47
100000	66905	33.09%	105.60	102.97
120000	79884	33.43%	127.09	124.29
140000	92930	33.62%	148.74	145.38
160000	106136	33.66%	170.38	166.81
180000	119224	33.76%	192.53	188.85
200000	132289	33.86%	215.27	210.60

Table 6.30 :: Techniques: FGK algorithm.
Scaled : Yes.
Model : Dynamic 0-order model.
File type : BNA

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	780	22.00%	1.10	1.10
2000	1464	26.80%	2.14	2.09
3000	2133	28.90%	3.13	3.13
4000	2816	29.60%	4.18	4.12
5000	3486	30.28%	5.22	5.11
6000	4161	30.65%	6.21	6.04
7000	4840	30.86%	7.25	7.09
8000	5557	30.54%	8.35	8.13
9000	6241	30.66%	9.40	9.07
10000	6933	30.67%	10.38	10.05
20000	13799	31.00%	20.99	20.38
40000	27649	30.88%	42.69	41.54
60000	42035	29.94%	64.56	63.19
80000	56183	29.77%	86.65	84.67
100000	66905	33.09%	107.64	103.08
120000	83401	30.50%	130.22	127.64
140000	96917	30.77%	152.14	149.07
160000	110488	30.95%	174.51	171.10
180000	123909	31.16%	197.14	193.52
200000	137538	31.23%	219.89	215.77

Table 6.31 :: Techniques: FGK algorithm.
Scaled : Yes.
Model : Dynamic 0-order model.
File type : QSTD

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	728	27.20%	1.04	1.04
2000	1389	30.55%	2.09	1.98
3000	2038	32.07%	3.08	2.91
4000	2690	32.75%	4.07	3.96
5000	3335	33.30%	5.16	4.95
6000	3988	33.53%	6.15	5.93
7000	4629	33.87%	7.14	6.87
8000	5268	34.15%	8.13	7.80
9000	5912	34.31%	9.23	8.74
10000	6559	34.41%	10.16	9.73
20000	12928	35.36%	20.60	19.78
40000	25987	35.03%	41.92	40.00
60000	39313	34.48%	63.30	60.71
80000	52834	33.96%	84.34	81.59
100000	65898	34.10%	104.95	102.03
120000	78847	34.29%	126.54	123.08
140000	91806	34.42%	148.30	144.18
160000	104720	34.55%	169.89	165.22
180000	117871	34.52%	192.03	186.92
200000	131401	34.30%	214.95	208.85

Table 6.32 :: Techniques: FGK algorithm.
Scaled : Yes.
Model : Dynamic 0-order model.
File type : QXFR

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	727	27.30%	1.04	0.99
2000	1405	29.75%	2.03	2.03
3000	2055	31.50%	3.13	3.02
4000	2705	32.38%	4.12	3.96
5000	3356	32.88%	5.16	4.95
6000	4014	33.10%	6.15	5.93
7000	4659	33.44%	7.09	6.92
8000	5300	33.75%	8.13	7.86
9000	5946	33.93%	9.18	8.79
10000	6601	33.99%	10.11	9.78
20000	13018	34.91%	20.44	19.78
40000	26110	34.73%	41.48	40.16
60000	39447	34.26%	62.75	60.93
80000	52985	33.77%	84.18	81.59
100000	66077	33.92%	105.49	102.25
120000	79041	34.13%	127.14	123.57
140000	92005	34.28%	148.46	144.40
160000	104985	34.38%	170.38	165.82
180000	118193	34.34%	192.75	187.75
200000	131823	34.09%	215.55	209.67

Table 6.33 :: Techniques: FGK algorithm.
Scaled : Yes.
Model : Dynamic 0-order model.
File type : QBNA

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	789	21.10%	1.10	1.10
2000	1478	26.10%	2.20	2.09
3000	2166	27.80%	3.24	3.13
4000	2867	28.32%	4.29	4.18
5000	3553	28.94%	5.38	5.11
6000	4237	29.38%	6.43	6.15
7000	4904	29.94%	7.47	7.09
8000	5595	30.06%	8.46	8.08
9000	6268	30.36%	9.51	9.18
10000	6951	30.49%	10.60	10.05
20000	13696	31.52%	21.32	20.33
40000	27225	31.94%	43.08	41.21
60000	40688	32.19%	64.67	61.98
80000	53980	32.52%	86.43	82.64
100000	67294	32.71%	107.86	103.68
120000	80622	32.81%	130.05	124.95
140000	93950	32.89%	152.42	146.21
160000	107307	32.93%	175.05	167.97
180000	120723	32.93%	197.64	190.00
200000	134196	32.90%	218.13	212.31

Table 6.34 :: Techniques: FGK algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : STD

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	713	28.70%	1.10	1.04
2000	1367	31.65%	2.09	2.03
3000	2025	32.50%	3.08	3.08
4000	2670	33.25%	4.18	4.07
5000	3323	33.54%	5.11	5.05
6000	3990	33.50%	6.15	6.04
7000	4656	33.49%	7.25	7.03
8000	5316	33.55%	8.24	8.02
9000	5952	33.87%	9.29	8.96
10000	6593	34.07%	10.22	9.95
20000	13127	34.37%	20.77	20.27
40000	26261	34.35%	42.14	41.15
60000	39239	34.60%	63.02	61.92
80000	52785	34.02%	84.89	83.13
100000	65791	34.21%	105.82	103.85
120000	78596	34.50%	127.47	124.95
140000	91519	34.63%	149.01	146.32
160000	104290	34.82%	170.44	167.64
180000	117413	34.77%	192.53	189.67
200000	130326	34.84%	214.95	211.59

Table 6.35 :: Techniques: FGK algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : XFR

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	722	27.80%	1.10	0.99
2000	1379	31.05%	2.09	2.03
3000	2046	31.80%	3.08	3.08
4000	2699	32.52%	4.23	4.07
5000	3360	32.80%	5.22	5.11
6000	4036	32.73%	6.21	6.10
7000	4724	32.51%	7.31	7.14
8000	5386	32.67%	8.41	8.13
8704	6352	27.02%	9.43	9.34
10000	6678	33.22%	10.33	10.11
20000	13325	33.38%	20.93	20.49
40000	26712	33.22%	42.47	41.54
60000	40045	33.26%	63.90	62.64
80000	53856	32.68%	85.60	84.18
100000	67049	32.95%	106.87	105.11
120000	80095	33.25%	128.68	126.59
140000	93184	33.44%	150.55	147.80
160000	106406	33.50%	172.42	169.73
180000	119517	33.60%	194.89	191.70
200000	132641	33.68%	216.92	213.85

Table 6.36 :: Techniques: FGK algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : BNA

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	780	22.00%	1.15	1.15
2000	1464	26.80%	2.25	2.14
3000	2133	28.90%	3.24	3.24
4000	2816	29.60%	4.34	4.29
5000	3486	30.28%	5.38	5.22
6000	4161	30.65%	6.43	6.21
7000	4840	30.86%	7.42	7.25
8000	5557	30.54%	8.57	8.35
9000	6241	30.66%	9.51	9.29
10000	6933	30.67%	10.60	10.33
20000	13799	31.00%	21.37	21.04
40000	27664	30.84%	43.30	42.42
60000	42287	29.52%	65.71	64.84
80000	56484	29.39%	88.08	86.48
100000	67049	32.95%	108.74	105.00
120000	83739	30.22%	131.70	129.89
140000	97309	30.49%	154.01	151.48
160000	110965	30.65%	176.59	173.52
180000	124426	30.87%	198.96	196.10
200000	138056	30.97%	221.81	218.30

Table 6.37 :: Techniques: FGK algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : QSTD

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	728	27.20%	1.10	1.04
2000	1389	30.55%	2.09	2.03
3000	2038	32.07%	3.13	3.08
4000	2690	32.75%	4.18	4.07
5000	3335	33.30%	5.27	5.05
6000	3988	33.53%	6.26	5.99
7000	4629	33.87%	7.25	7.03
8000	5268	34.15%	8.30	8.02
9000	5912	34.31%	9.34	9.07
10000	6559	34.41%	10.33	10.05
20000	12928	35.36%	20.93	20.11
40000	25991	35.02%	42.31	40.99
60000	39356	34.41%	64.01	61.87
80000	52954	33.81%	85.60	83.19
100000	66002	34.00%	106.26	104.18
120000	78987	34.18%	127.75	125.00
140000	91973	34.30%	149.18	145.99
160000	104912	34.43%	171.37	167.58
180000	118084	34.40%	193.85	189.73
200000	131668	34.17%	216.32	211.76

Table 6.38 :: Techniques: FGK algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : QXFR

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	727	27.30%	1.10	1.10
2000	1405	29.75%	2.09	2.03
3000	2055	31.50%	3.19	3.13
4000	2705	32.38%	4.23	4.07
5000	3356	32.88%	5.22	5.11
6000	4014	33.10%	6.26	6.10
7000	4659	33.44%	7.25	7.03
8000	5300	33.75%	8.24	8.08
9000	5946	33.93%	9.34	9.07
10000	6601	33.99%	10.33	10.00
20000	13018	34.91%	20.71	20.33
40000	26113	34.72%	41.92	41.10
60000	39477	34.20%	63.46	62.09
80000	53069	33.66%	85.05	83.19
100000	66159	33.84%	106.54	103.90
120000	79143	34.05%	128.19	125.22
140000	92137	34.19%	149.89	146.54
160000	105149	34.28%	171.87	168.02
180000	118378	34.23%	194.07	190.11
200000	132037	33.98%	217.03	212.47

Table 6.39 :: Techniques: FGK algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : QBNA

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	789	21.10%	1.10	1.15
2000	1478	26.10%	2.20	2.14
3000	2166	27.80%	3.30	3.19
4000	2867	28.32%	4.40	4.18
5000	3553	28.94%	5.44	5.27
6000	4237	29.38%	6.59	6.32
7000	4904	29.94%	7.58	7.31
8000	5595	30.06%	8.63	8.35
9000	6268	30.36%	9.67	9.34
10000	6951	30.49%	10.77	10.33
20000	13696	31.52%	21.59	20.82
40000	27234	31.91%	43.46	42.03
60000	40711	32.15%	65.38	63.02
80000	54024	32.47%	87.20	83.85
100000	67371	32.63%	109.01	104.95
120000	80711	32.74%	131.32	126.59
140000	94174	32.73%	153.46	148.24
160000	107675	32.70%	176.43	170.44
180000	121201	32.67%	198.68	192.64
200000	134745	32.63%	219.67	214.78

this variation has been found from 28.70% to 34.84% for STD format text and from 22.00% to 30.97% for BNA format texts. No variation of coding efficiency for scaled and unscaled text have been found up to 20000 bytes file length for any format text and both for general and specific text. With scaled symbol counts, coding times have been found for same text types from 1.04 sec. to 212.53 sec. for STD format and from 1.10 sec. to 219.89 sec. for BNA format and decoding times have been found from 1.04 sec. to 208.24 sec. and from 1.10 sec. to 215.77 sec. respectively for the mentioned text formats and types. That is, the decoding operation has been found faster than coding operation. For unscaled symbol counts the coding and decoding times for all text formats and text types have been found higher than those of scaled counts.

The results for Knuth algorith have been given in Tables 6.40 to 6.45. The variation of coding efficiency for general texts of this algorithm have been found from 27.50% to 34.83% for STD text formats and from 20.60% to 30.97% for BNA format texts. The coding times vary from 0.99 sec. to 21.53 sec. for STD format and from 0.99 sec. to 217.75 sec. for BNA format. Similarly the decoding times have been found from 0.99 sec. to 213.46 sec. and from 1.04 sec. to 220.33 sec. for STD and BNA format texts respectively.

Table 6.40 :: Techniques: Knuth Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : STD

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	725	27.50%	0.99	0.99
2000	1380	31.00%	2.03	1.92
3000	2039	32.03%	3.02	3.02
4000	2684	32.90%	4.01	3.96
5000	3338	33.24%	5.00	5.00
6000	4006	33.23%	5.99	5.99
7000	4672	33.26%	6.98	6.98
8000	5333	33.34%	7.97	7.97
9000	5969	33.68%	9.01	8.96
10000	6610	33.90%	9.95	9.89
20000	13144	34.28%	20.27	20.33
40000	26279	34.30%	41.32	41.26
60000	39256	34.57%	62.14	62.20
80000	52803	34.00%	83.41	83.68
100000	65808	34.19%	104.23	104.73
120000	78614	34.49%	125.71	126.04
140000	91537	34.62%	146.92	147.64
160000	104308	34.81%	168.19	169.23
180000	117431	34.76%	190.16	191.65
200000	130345	34.83%	212.53	213.46

Table 6.41 :: Techniques: Knuth Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : XFR

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	734	26.60%	1.04	0.99
2000	1393	30.35%	1.98	1.98
3000	2059	31.37%	2.97	3.02
4000	2712	32.20%	3.96	4.01
5000	3373	32.54%	5.00	4.95
6000	4049	32.52%	5.99	5.99
7000	4729	32.44%	7.03	7.03
8000	5392	32.60%	8.02	8.02
9000	6033	32.97%	9.01	9.01
10000	6684	33.16%	10.00	10.00
20000	13339	33.30%	20.16	20.44
40000	26726	33.19%	41.04	41.65
60000	40060	33.23%	61.92	62.86
80000	53871	32.66%	83.24	84.51
100000	67063	32.94%	103.63	105.44
120000	80110	33.24%	124.73	127.03
140000	93199	33.43%	145.88	148.85
160000	106421	33.49%	167.64	171.43
180000	119532	33.59%	189.78	192.86
200000	132655	33.67%	211.92	214.84

Table 6.42 :: Techniques: Knuth Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : BNA

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	794	20.60%	0.99	1.04
2000	1479	26.05%	2.09	2.03
3000	2148	28.40%	3.08	3.08
4000	2831	29.23%	4.12	4.07
5000	3502	29.96%	5.11	5.05
6000	4177	30.38%	6.10	6.10
7000	4855	30.64%	7.09	7.14
8000	5573	30.34%	8.13	8.13
9000	6257	30.48%	9.29	9.18
10000	6949	30.51%	10.22	10.22
20000	13817	30.91%	21.15	20.82
40000	27682	30.80%	42.97	42.42
60000	42291	29.52%	65.33	65.16
80000	56487	29.39%	87.58	88.57
100000	70117	29.88%	107.64	108.19
120000	83742	30.21%	129.56	129.67
140000	97312	30.49%	151.37	151.98
160000	110967	30.65%	174.12	176.04
180000	124428	30.87%	195.44	196.81
200000	138058	30.97%	217.75	220.33

Table 6.43 :: Techniques: Knuth Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : QSTD

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	739	26.10%	1.04	0.99
2000	1401	29.95%	2.03	1.98
3000	2051	31.63%	3.02	2.97
4000	2703	32.42%	4.01	3.90
5000	3348	33.04%	5.05	4.95
6000	4001	33.32%	6.04	5.88
7000	4643	33.67%	7.03	6.92
8000	5281	33.99%	8.08	7.91
9000	5926	34.16%	9.07	8.85
10000	6573	34.27%	10.00	9.78
20000	12942	35.29%	20.38	20.00
40000	26005	34.99%	41.48	40.60
60000	39371	34.38%	62.91	61.87
80000	52969	33.79%	84.07	83.13
100000	66017	33.98%	103.85	103.52
120000	79002	34.16%	125.33	125.16
140000	91987	34.30%	146.76	146.32
160000	104926	34.42%	168.30	168.08
180000	118099	34.39%	190.33	191.10
200000	131683	34.16%	213.24	212.31

Table 6.44 :: Techniques: Knuth Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : QXFR

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	737	26.30%	0.99	0.99
2000	1415	29.25%	1.98	1.98
3000	2065	31.17%	3.02	2.97
4000	2714	32.15%	4.01	3.96
5000	3366	32.68%	5.00	4.95
6000	4025	32.92%	5.99	5.93
7000	4670	33.29%	6.98	6.92
8000	5311	33.61%	7.86	7.91
9000	5957	33.81%	8.90	8.90
10000	6612	33.88%	9.95	9.84
20000	13029	34.85%	20.11	20.11
40000	26124	34.69%	41.10	40.88
60000	39488	34.19%	62.09	62.03
80000	53080	33.65%	83.41	83.19
100000	66169	33.83%	104.01	103.85
120000	79153	34.04%	125.66	125.22
140000	92148	34.18%	146.98	146.48
160000	105159	34.28%	168.52	168.02
180000	118388	34.23%	190.77	190.33
200000	132046	33.98%	213.08	212.64

Table 6.45 :: Techniques: Knuth Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : QBNA

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	803	19.70%	1.10	1.04
2000	1492	25.40%	2.14	2.03
3000	2180	27.33%	3.13	3.08
4000	2881	27.98%	4.18	4.07
5000	3567	28.66%	5.22	5.11
6000	4251	29.15%	6.32	6.10
7000	4918	29.74%	7.25	7.14
8000	5609	29.89%	8.30	8.08
9000	6282	30.20%	9.34	9.07
10000	6965	30.35%	10.38	10.11
20000	13709	31.45%	21.15	20.60
40000	27246	31.89%	42.80	41.59
60000	40723	32.13%	64.34	62.75
80000	54035	32.46%	85.71	83.74
100000	67383	32.62%	107.20	104.78
120000	80722	32.73%	129.07	126.48
140000	94185	32.73%	151.26	148.13
160000	107686	32.70%	173.90	170.27
180000	121211	32.66%	195.88	192.64
200000	134755	32.62%	216.26	214.95

The results of Vitter algorithm have been given in Tables 6.46 to 6.51. The variation of coding efficiency has been found in these tables from 28.40 % to 34.84% for STD format and from 21.70% to 30.99% for BNA format for general texts. The coding and decoding times for general STD format texts have been found from 1.21 sec. to 247.75 sec. and 1.21 sec. to 253.46 sec. respectively in Table 6.46. Similar variations for BNA format text have been found in Table 6.48 from 1.32 sec. to 254.56 sec. and 1.26 sec. to 260.93 sec. respectively.

The results of Arithmetic coding algorithm have been given in Tables 6.52 to 6.57. Coding efficiency for general texts have been varied from 24.80% to 34.92% for STD format and from 13.70% to 30.47% for BNA format text. Coding times for these text type and formats have been found from 1.10 sec. to 226.32 sec. and 1.10 sec. to 230.77 sec. respectively. Whereas decoding times have been found from 1.26 sec. to 281.59 sec. for STD format and from 1.37 sec. to 291.81 sec. for BNA format.

Table 6.46 :: Techniques: Vitter Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : STD

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	716	28.40%	1.21	1.21
2000	1371	31.45%	2.36	2.42
3000	2029	32.37%	3.57	3.63
4000	2675	33.12%	4.73	4.89
5000	3328	33.44%	5.93	5.99
6000	3995	33.42%	7.09	7.20
7000	4660	33.43%	8.41	8.46
8000	5321	33.49%	9.51	9.67
9000	5956	33.82%	10.71	10.82
10000	6597	34.03%	11.81	12.03
20000	13131	34.34%	23.96	24.40
40000	26264	34.34%	48.68	49.40
60000	39241	34.60%	72.97	74.23
80000	52786	34.02%	97.86	99.89
100000	65790	34.21%	122.36	124.73
120000	78596	34.50%	146.87	150.05
140000	91518	34.63%	171.98	175.49
160000	104290	34.82%	197.03	201.10
180000	117412	34.77%	222.69	227.36
200000	130325	34.84%	247.75	253.46

Table 6.47 :: Techniques: Vitter Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : XFR

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	726	27.40%	1.26	1.21
2000	1383	30.85%	2.42	2.42
3000	2050	31.67%	3.57	3.68
4000	2702	32.45%	4.73	4.89
5000	3363	32.74%	5.99	6.04
6000	4038	32.70%	7.20	7.31
7000	4716	32.63%	8.35	8.52
8000	5379	32.76%	9.51	9.73
9000	6020	33.11%	10.66	10.93
10000	6670	33.30%	11.98	12.09
20000	13323	33.38%	24.07	24.34
40000	26709	33.23%	48.57	49.62
60000	40041	33.27%	73.02	74.95
80000	53848	32.69%	98.19	100.44
100000	67039	32.96%	122.36	125.27
120000	80084	33.26%	147.31	150.93
140000	93173	33.45%	172.25	176.48
160000	106394	33.50%	197.64	202.36
180000	119504	33.61%	223.57	229.34
200000	132627	33.69%	248.85	254.45

Table 6.48 :: Techniques: Vitter Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : BNA

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	783	21.70%	1.32	1.26
2000	1467	26.65%	2.53	2.53
3000	2135	28.83%	3.74	3.74
4000	2818	29.55%	4.95	4.95
5000	3488	30.24%	6.10	6.21
6000	4163	30.62%	7.36	7.42
7000	4841	30.84%	8.46	8.57
8000	5557	30.54%	9.78	9.89
9000	6241	30.66%	10.99	11.15
10000	6932	30.68%	12.25	12.36
20000	13797	31.02%	24.62	25.00
40000	27660	30.85%	49.67	50.66
60000	42256	29.57%	75.60	77.20
80000	56450	29.44%	101.04	103.24
100000	70079	29.92%	125.49	128.52
120000	83704	30.25%	150.99	154.73
140000	97274	30.52%	176.76	180.99
160000	110928	30.67%	203.46	207.14
180000	124389	30.89%	228.74	233.79
200000	138019	30.99%	254.56	260.93

Table 6.49 :: Techniques: Vitter Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : QSTD

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	732	26.80%	1.21	1.26
2000	1394	30.30%	2.42	2.42
3000	2043	31.90%	3.63	3.57
4000	2695	32.62%	4.84	4.78
5000	3340	33.20%	6.04	5.99
6000	3993	33.45%	7.14	7.20
7000	4634	33.80%	8.35	8.46
8000	5272	34.10%	9.56	9.56
9000	5917	34.26%	10.77	10.82
10000	6564	34.36%	11.87	11.92
20000	12933	35.34%	24.07	24.12
40000	25996	35.01%	48.63	49.01
60000	39361	34.40%	73.57	74.23
80000	52957	33.80%	98.19	99.67
100000	66004	34.00%	121.65	124.45
120000	78990	34.17%	146.92	149.95
140000	91975	34.30%	171.54	175.38
160000	104914	34.43%	196.54	200.93
180000	118086	34.40%	222.03	227.09
200000	131669	34.17%	248.41	253.96

Table 6.50 :: Techniques: Vitter Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : QXFR

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	730	27.00%	1.26	1.21
2000	1407	29.65%	2.42	2.42
3000	2058	31.40%	3.63	3.68
4000	2707	32.33%	4.78	4.84
5000	3358	32.84%	5.99	6.04
6000	4016	33.07%	7.14	7.25
7000	4661	33.41%	8.24	8.46
8000	5301	33.74%	9.40	9.62
9000	5947	33.92%	10.60	10.77
10000	6601	33.99%	11.76	12.03
20000	13017	34.91%	23.68	24.29
40000	26111	34.72%	48.19	49.18
60000	39475	34.21%	72.75	74.56
80000	53066	33.67%	97.58	99.95
100000	66155	33.84%	122.03	124.84
120000	79139	34.05%	146.98	150.11
140000	92132	34.19%	171.81	175.71
160000	105144	34.28%	197.03	202.53
180000	118373	34.24%	222.69	227.58
200000	132031	33.98%	248.79	254.23

Table 6.51 :: Techniques: Vitter Algorithm.
Scaled : No.
Model : Dynamic 0-order model.
File type : QBNA

Text file Size	Code file Size	Compression Efficiency	Time in seconds	
			Compression	Expansion
1000	789	21.10%	1.32	1.26
2000	1478	26.10%	2.47	2.53
3000	2164	27.87%	3.74	3.79
4000	2866	28.35%	5.05	5.05
5000	3551	28.98%	6.21	6.21
6000	4235	29.42%	7.47	7.47
7000	4901	29.99%	8.68	8.68
8000	5592	30.10%	9.84	9.89
9000	6264	30.40%	11.15	11.15
10000	6948	30.52%	12.31	12.36
20000	13690	31.55%	24.23	24.89
40000	27226	31.93%	48.79	50.16
60000	40702	32.16%	73.52	75.99
80000	54015	32.48%	98.19	102.64
100000	67360	32.64%	125.00	126.37
120000	80699	32.75%	150.77	151.43
140000	94162	32.74%	176.70	177.42
160000	107662	32.71%	203.30	204.89
180000	121187	32.67%	229.18	230.71
200000	134732	32.63%	252.58	257.36

Table 6.52 :: Techniques: Arithmetic coding.
Scaled : Yes.
Model : Static 0-order model.
File type : STD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	752	114	638	24.80%	1.10	1.26
2000	1401	125	1276	29.95%	2.09	2.64
3000	2048	125	1923	31.73%	3.19	4.01
4000	2689	125	2564	32.77%	4.23	5.33
5000	3337	126	3211	33.26%	5.38	6.65
6000	3998	129	3869	33.37%	6.37	7.97
7000	4661	134	4527	33.41%	7.47	9.34
8000	5312	134	5178	33.60%	8.52	10.60
9000	5941	134	5807	33.99%	9.51	11.92
10000	6584	134	6450	34.16%	10.55	13.19
20000	13092	140	12952	34.54%	21.65	26.98
40000	26194	144	26050	34.52%	44.01	54.56
60000	39134	144	38990	34.78%	66.10	82.36
80000	52689	150	52539	34.14%	88.90	110.77
100000	65643	150	65493	34.36%	111.32	138.41
120000	78416	150	78266	34.65%	133.79	166.70
140000	91298	153	91145	34.79%	156.76	195.11
160000	104071	153	103918	34.96%	179.40	223.46
180000	117209	156	117053	34.88%	203.24	252.75
200000	130155	156	129999	34.92%	226.32	281.59

Table 6.53 :: Techniques: Arithmetic coding.
Scaled : Yes.
Model : Static 0-order model.
File type : XFR

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	791	151	640	20.90%	1.10	1.37
2000	1444	164	1280	27.80%	2.14	2.64
3000	2104	170	1934	29.87%	3.19	4.01
4000	2757	173	2584	31.07%	4.23	5.33
5000	3413	174	3239	31.74%	5.27	6.65
6000	4089	183	3906	31.85%	6.37	8.02
7000	4767	191	4576	31.90%	7.47	9.34
8000	5428	193	5235	32.15%	8.52	10.77
9000	6061	193	5868	32.66%	9.56	12.03
10000	6710	193	6517	32.90%	10.71	13.30
20000	13455	207	13248	32.73%	21.76	27.20
40000	27003	222	26781	32.49%	44.12	55.27
60000	40558	225	40333	32.40%	66.54	83.46
80000	53793	225	53568	32.76%	89.12	111.32
100000	66900	226	66674	33.10%	111.32	139.34
120000	79955	226	79729	33.37%	133.85	167.86
140000	93218	228	92990	33.42%	157.09	196.59
160000	106384	228	106156	33.51%	179.95	225.44
180000	119882	229	119653	33.40%	204.07	255.00
200000	133948	234	133714	33.03%	227.75	284.84

Table 6.54 :: Techniques: Arithmetic coding.
Scaled : Yes.
Model : Static 0-order model.
File type : BNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	863	187	676	13.70%	1.10	1.37
2000	1541	195	1346	22.95%	2.14	2.75
3000	2203	197	2006	26.57%	3.24	4.12
4000	2880	201	2679	28.00%	4.23	5.44
5000	3551	204	3347	28.98%	5.38	6.87
6000	4219	204	4015	29.68%	6.43	8.24
7000	4898	207	4691	30.03%	7.53	9.56
8000	5609	215	5394	29.89%	8.63	10.93
9000	6292	216	6076	30.09%	9.62	12.36
10000	6980	221	6759	30.20%	10.77	13.68
20000	13831	229	13602	30.84%	21.87	27.75
40000	27725	244	27481	30.69%	44.56	56.32
60000	42530	258	42272	29.12%	67.80	86.04
80000	56822	258	56564	28.97%	90.82	115.16
100000	70568	258	70310	29.43%	113.24	143.63
120000	84293	258	84035	29.76%	136.48	173.02
140000	97975	258	97717	30.02%	159.73	202.25
160000	111728	259	111469	30.17%	183.24	231.87
180000	125308	259	125049	30.38%	207.03	261.65
200000	139054	259	138795	30.47%	230.77	291.81

Table 6.55 :: Techniques: Arithmetic coding.
Scaled : Yes.
Model : Static 0-order model.
File type : QSTD

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	779	134	645	22.10%	1.15	1.32
2000	1428	137	1291	28.60%	2.14	2.64
3000	2067	137	1930	31.10%	3.13	3.96
4000	2715	137	2578	32.12%	4.29	5.27
5000	3351	137	3214	32.98%	5.33	6.65
6000	4004	140	3864	33.27%	6.37	7.91
7000	4644	141	4503	33.66%	7.47	9.23
8000	5278	143	5135	34.02%	8.57	10.60
9000	5918	143	5775	34.24%	9.62	11.92
10000	6561	143	6418	34.39%	10.71	13.30
20000	12897	143	12754	35.52%	21.70	26.92
40000	25915	145	25770	35.21%	44.07	54.62
60000	39233	145	39088	34.61%	66.81	82.75
80000	52800	146	52654	34.00%	89.34	111.32
100000	65833	146	65687	34.17%	110.77	138.96
120000	78778	146	78632	34.35%	133.52	167.20
140000	91766	146	91620	34.45%	156.37	195.99
160000	104680	146	104534	34.58%	179.18	224.73
180000	117812	146	117666	34.55%	202.53	253.68
200000	131341	146	131195	34.33%	226.54	283.35

Table 6.56 :: Techniques: Arithmetic coding.
Scaled : Yes.
Model : Static 0-order model.
File type : QXFR

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	790	146	644	21.00%	1.10	1.32
2000	1467	169	1298	26.65%	2.09	2.69
3000	2111	170	1941	29.63%	3.13	4.01
4000	2757	171	2586	31.07%	4.23	5.33
5000	3405	177	3228	31.90%	5.27	6.65
6000	4066	184	3882	32.23%	6.37	8.02
7000	4712	187	4525	32.69%	7.36	9.34
8000	5352	192	5160	33.10%	8.41	10.66
9000	6005	201	5804	33.28%	9.45	11.92
10000	6661	203	6458	33.39%	10.60	13.35
20000	13064	208	12856	34.68%	21.37	26.98
40000	26187	215	25972	34.53%	43.68	54.89
60000	39565	215	39350	34.06%	65.99	83.13
80000	53198	221	52977	33.50%	88.46	111.43
100000	66346	221	66125	33.65%	110.99	139.51
120000	79383	221	79162	33.85%	133.74	167.91
140000	92484	221	92263	33.94%	156.59	196.43
160000	105576	221	105355	34.02%	179.56	225.44
180000	118838	221	118617	33.98%	203.02	254.62
200000	132535	221	132314	33.73%	226.76	284.23

Table 6.57 :: Techniques: Arithmetic coding.
Scaled : Yes.
Model : Static 0-order model.
File type : QBNA

Text file Size	Code file size			Compression Efficiency	Time in seconds	
	Total	Statistics	Codes		Compression	Expansion
1000	858	177	681	14.20%	1.10	1.37
2000	1550	199	1351	22.50%	2.14	2.75
3000	2234	207	2027	25.53%	3.24	4.12
4000	2929	212	2717	26.77%	4.40	5.49
5000	3617	220	3397	27.66%	5.49	6.87
6000	4295	222	4073	28.42%	6.54	8.30
7000	4963	223	4740	29.10%	7.64	9.62
8000	5646	223	5423	29.43%	8.68	11.04
9000	6321	223	6098	29.77%	9.89	12.36
10000	6998	225	6773	30.02%	10.88	13.74
20000	13710	229	13481	31.45%	22.14	27.69
40000	27196	238	26958	32.01%	44.95	56.21
60000	40662	240	40422	32.23%	67.69	84.73
80000	53942	240	53702	32.57%	90.44	113.63
100000	67322	244	67078	32.68%	113.13	141.54
120000	80703	246	80457	32.75%	136.59	170.44
140000	94152	247	93905	32.75%	159.89	199.34
160000	107681	247	107434	32.70%	183.41	228.85
180000	121228	248	120980	32.65%	206.98	258.46
200000	134786	248	134538	32.61%	228.35	288.68

Chapter Seven

DISCUSSIONS AND RECOMMENDATIONS

7.1 Discussions

I have studied and implemented different coding techniques and tested in Bangla text. Effect of coding efficiency, compression and decompression time of these techniques on file length is point out. Similar effects on file format for general and specific Bangla text are also studied.

Static Algorithms: We studied both static and dynamic versions of the coding techniques. Among static versions of the coding techniques, Shannon-Fano, Huffman and Arithmetic Coding techniques are tested and Shannon-Fano and Huffman algorithm are implemented both scaled and unscaled counted, arithmetic coding technique is implemented only for scaled counted. Among these static techniques with scaled count, arithmetic coding shows better efficiency and Huffman coding techniques shows faster compression and decompression times.

Dynamic Algorithms: Among the dynamic coding techniques, Huffman algorithm modified Faller, Gallager and Knuth (FGK algorithm) and Optimal algorithm by Vitter are studied and tested. FGK algorithm is implemented both in Linked-list and array data structures. Knuth suggested implementing dynamic Huffman algorithm in array structures for efficient modification of the tree, so we call this version as Knuth algorithm whereas algorithm implementation in linked-list structures is called as FGK algorithm. Among these dynamic algorithms without scaling the symbol count, Vitter algorithm is found most efficient complying with its theoretically established results. Compression and decompression times of this algorithm are higher than any other algorithms implemented in this work and decompression time is higher than the compression time for all formats and types of texts. FGK algorithm with scaled count has shown better efficiency among all dynamic algorithms but the same algorithm with unscaled count has wooer efficiency than Vitter algorithm with unscaled count. Dynamic algorithm by Knuth is found to have faster compression time than any other dynamic algorithm whereas FGK algorithm with scaled count shows faster decompression time.

From the Table 6.28 to 6.39, the same efficiencies have been found same in dynamic Huffman (FGK) algorithm up to 20,000 bytes file length with scaled and unscaled symbol counts independent of text type and text format.

Static and dynamic algorithm: Dynamic versions of Huffman algorithm are FGK algorithm, Knuth algorithm and Vitter algorithm. Shannon-Fano algorithm and Arithmetic coding algorithm are implemented only for static coding. Huffman algorithm with and without scaled count is found to have better efficiency in dynamic versions. Better coding and decoding times have been found in dynamic version for small files and static version for large files.

Effect of Scaling: Static Shannon-Fano algorithm, Huffman algorithm and dynamic FGK algorithm are implemented both for scaled and unscaled count. Static Huffman and Shannon-Fano algorithms without scaling the symbol counts show the negative compression efficiency for small files, i.e., up to 2000 bytes file compression is not possible for these coding techniques with unscaled counts. No effect of scaling symbol counts on coding efficiency is found in dynamic version of Huffman (FGK) algorithm for small files (up to 20000 byte). Compression and decompression times is found smaller in scaled version of these algorithms for all file formats except BNA (i.e. document) file format for both general and specific text.

Effect of file format: Each algorithm shows better performance for BSCII format and worst performance for BNA file format for both general and specific texts.

Effect of file length: We tested the performance of the algorithms by varying file lengths from 1,000 byte to 200,000 bytes. With this range of file sizes, coding efficiency vary from -32.00% (Shannon-Fano algorithm with unscaled count for BNA format in general text) to 34.92% (Arithmetic coding with scaled count for STD format in general text). Efficiency increases rapidly up to 4000 byte file and very slowly with increasing file length. Small irregularity of efficiency change is found around 80000 byte length. Similar effect on coding and decoding time on file length is found. Compression time varies from 0.99 sec. (Shannon-Fano, Huffman and FGK algorithms with scaled count for STD format) to 254.56 sec.(Vitter algorithm with unscaled count for BNA format) and decoding time varies from 0.93 sec.(Shannon-Fano algorithms with scaled count for STD format to 291.81 sec. (Arithmetic coding with scaled count for BNA format). Both compression and decompression time increase slowly for smaller files and increase very rapidly after around 15000 byte of file length. Increment of compression and decompression time are almost linear.

Static variable length Codes: We have given two tables of static variable length codes for Bangla BSCII format text. In Table 6.1 static Huffman codes is given and static Shannon-Fano code is given in Table 6.2. Average code length are found 5.009 and 5.129 for Huffman and Shannon-Fano algorithm respectively. Corresponding Huffman and Shannon-Fano trees are given in Fig.6.1 and Fig.6.2 respectively. It is clear from the code tables and trees, Huffman codes is better than Shannon-Fano algorithm and Huffman algorithm is easier to implement.

Text analysis: We have analyzed Bangla texts in BSCII format. The text we considered for analysis has a file length of 25024 bytes. The frequency of Bangla character in BSCII format and frequency of Bangla Akkharas are found. The character া (194) is found the highest frequent character after space and র (102) is found most frequent akkhara. We have also given the 25 most frequent Bangla words. The word এবং is found the most frequent among Bangla words.

The n-Grams for Bangla text both for general and specific texts up to 8-gram is given and entropy and redundancy for them is calculated. We found that with the increase of order of n-gram the entropy decreases whereas redundancy increases. Both entropy and redundancy become constant for higher order n-gram. So we can expect more efficiency for higher order n-gram.

7.2 Recommendations

In this work static lossless algorithms have been implemented and tested. Effect of scaled count is also studied on some of these algorithms. An n-gram statistic of Bangla text is given for response of redundancy. Future research work on compression algorithm for Bangla text can be carried on:

(a) Effects of coding efficiency, coding and decoding time can be studied by dictionary based coding algorithms like LZW algorithm.

(b) Specific compression algorithms can be developed for Bangla text exploiting n-gram statistics and redundancy of the text.

(c) Global standard static variable length codes for Bangla texts can be developed for efficient and fast compression-decompression.

(d) A set of standard static variable length codes for different types of text can be established.

(e) Adaptive version of arithmetic coding can be studied for Bangla texts.

(f) Higher order model of these algorithms can be studied.

REFERENCES

[1] Ahmed, M., "Design of Bengali Alphanumeric Segment and Dot Matrix Display", M. Sc. Thesis, CSE Deptt., BUET, 1986.

[2] Anson, L.F., "Fractal Image Compression", BYTE, (October 1993), 195-202.

[3] Apiki, S., "Lossless Data Compression", BYTE (March 1991), 309-314, 386-387.

[4] Aronson, J., "Data Compression - A Comparison of Methods", National Bureau of Standards, PB-269-296, (June 1977).

[5] Bassiouni, M.A. and Ok, B., "Double Encoding - A Technique for Reducing Storage Requirement of Text", Information Systems, 11, 2(1986), 177-184.

[6] Bookstein, A. and Storer, J.A., "Data Compression", Information Processing and Management, 28, 6(1992), 675-680.

[7] Byrd, M., "Data Compression: Is It All It Claims", PC Magazine, (Dec.11, 1990), 316-317.

[8] Capocelli, R.M. and Santis, A.D., "A Note on D-ary Huffman Codes", IEEE Trans. Inform. Theory, IT-37, 1(Jan 1991), 174-179.

[9] Chowdhury, M.H., "Image Encoding and Representation Techniques with Application to An On-Line Banking System", M. Sc. Thesis, CSE Deptt., BUET, 1990.

[10] Connel, J.B., "A Huffman-Shannon-Fano Code," Proc. IEEE 61 (Jul. 1973), 1046-1047.

[11] Cover, T.M., "On the Competitive Optimality of Huffman Codes", IEEE Trans. Inform. Theory, IT-37, 1(Jan.1991), 172-174.

[12] Das, G., Bhattacharya, S., and Mitra, S., "Representing Ahamia, Bengali and Monipuri Text in Line Printer and Display-wheel Printer", Journal of the Institute of Electronics and Telecommunication Engineers, India, v-30, n-6, (1984), 251-256.

[13] Das, P.K., "On Information Content of Bengali Language and Noise in Microwave Communication in Bangladesh", M. Sc. Thesis, EEE Deptt., BUET, 1976.

[14] Even, S.,and Lempel, A. "Generation and Enumeration of all Solutions of the Characteristic Sum Condition," Inform. 21 (1972), 476-482.

[15] Faller, N., "An Adaptive System or Data Compression", In Record of the 7th Asilomar Conference on Circuits, Systems, and Computers. (1973), 593-597.

[16] Fano, R.M., "Transmission of Information", Cambridge, MA; MIT Press and New York; Yiley, 1961.

[17] Felician, L. and Gentili, A., "A Nearly Optimal Huffman Technique in the Microcomputer Environment", Information Systems, 12, 4(1987), 371-373.

[18] Ferguson, T.J. and Rabinowitz, J.H., "Self-Synchronizing Huffman Codes", IEEE Trans. on Inform. Theory, IT-30(4), (July 1984), 687-693.

[19] Frazer, W.D. and Bennett, B.T., "Bounds on Optimal Merge Performance, and a Strategy for Optimality", JACM, 19, (1972), 641-648.

[20] Gallager, R.G., "Variations on the Theme by Huffman", IEEE Trans. Inform. Theory, IT-24 (1978), 668-674.

[21] Geckinli, N.C., "Two Corollaries to the Huffman Coding Procedure", IEEE Trans. Inform. Theory, (May 1975), 342-345.

[22] Gilbert, E.N. and Moore, E.F., "Variable Length Binary Encodings", Bell Syst. Tech. J. (July 1959), 933-967.

[23] Glassey, C.R., and Karp, M.R., "Optimum Binary Search Trees", SIAM J. Appl. Math. 31, (1976), 368-372.

[24] Golumbic, M.C., "Combinatorial Merging ", IEEE Trans. Computers, TC-25, (1976), 1164-1167.

[25] Guazzo, M., "A General Minimum-Redundancy Source-Coding Algorithm", IEEE Trans. Inform. Theory, IT-26(1), (Jan. 1980), 15-25.

[26] Hamming, R.W., "Coding and Information Theory", Prentice-Hall, Englewood Cliffs NJ 07632, 1986.

[27] Haque, S.Z., "Statistical Analysis of Messages and Their Coding of Bangli Language", M. Sc. Thesis, EEE Deptt., BUET, 1985.

[28] Harmon, G., "The Measurement of Information", Information Processing and Management, 20, 1-2(1984), 193-198.

[29] Held, G., "Data Compression", John Wiley and Sons, New York, 1984.

[30] Howard, P.G. and Vitter, J.S., "Analysis of Arithmetic Coding for Data Compression", in Proc. Data Compression Conference, J. A. Storer and J. H. Reif, eds., Snowbird, Utah, Apr. 8-11, 1991, 3-12, invited paper, also to appear as an in the special issue of Information Processing and Management, also appears as Brown University Technical Report No. CS-91-03.

[31] Howard, P.G. and Vitter, J.S., "New Methods for Lossless Image Compression Using Arithmetic Coding", in Proc. Data Compression Conference, J. A. Storer and J. H. Reif, eds., Snowbird, Utah, Apr. 8-11, 1991, 257-266, also to appear as an in the special issue of Information Processing and Management, also appears as Brown University Technical Report No. CS-91-47.

[32] Howard, P.G. and Vitter, J.S., "Practical Implementations of Arithmetic Coding", Brown University Technical Report No. CS-92-18.

[33] Hu, T.C., Kleitman, D.J. and Tamaki, J.K., "Binary Trees Optimum Under Various Criteria", SIAM J. Appl. Math, 37, 2(Oct.1979), 246-256.

[34] Hu, T.C. and Tucker, A.C., "Optimal Computer Search Trees and Variable-length Alphabetical Codes", SIAM J. Appl. Math. 21, (1971), 514-532.

[35] Huffman, D.A., "A Method for the Construction of Minimum Redundancy Codes," Proc. IRE 40 (Sep. 1952), 1098-1101.

[36] Humayun, S.M., Rahman, S.H. and Kaykobad, M., "Static Huffman Code for Bangla Text", 15th Annual Conference of BAAS, Section III, AERE, Savar, March 5-8, 1990.

[37] Itai, A., "Optimal Alphabetic Trees", SIAM J. Computer, 9(3), (Aug. 1980), 9-18.

[38] Jayant, N.S., and Noll, P., "Digital Coding of Waveforms", Englewood Cliffs, NJ, Prentice-Hall, 1984.

[39] Jones, C.B., "An Efficient Coding System for Long Source Sequences", IEEE Trans. Inform. Theory, IT-27, 3(May 1981), 280-291.

[40] Ju, R.H., Jou, I.C. and Tsay,M.K., "Global Study on Data Compression Techniques for Digital Chiniese Character Patterns", Proc. IEE, 139, 2(Jan 1992), 1-8.

[41] Kerp, R.M., "Minimum-redundancy Coding for Discrete Noiseless Channel", IRE Trans. Inform. Theory IT-17, (Jan. 1961), 27-38.

[42] Kerpez, K.J., "Runlength Codes From Source Codes", IEEE Trans. Inform. Theory, IT-37, 3(May 1991), 682-687.

[43] Khan, M.M.H.A., "Optimal Realization of Bengali Keyboard and Character Encoding for Computer Applications", M. Sc. Thesis, CSE Deptt., BUET, 1986.

[44] Knuth, D.E., "Dynamic Huffman Coding", J. Algorithms, 6(1985), 163-180.

[45] Knuth, D.E., "The Art of Computer Programming", Volume I and III, Addison-Wesley Publishing Company, Inc., 1973.

[46] Kuraisi, F.A., "Bangla Barnomalar Anatomy", Shamikha Prokashoni, Dhaka, Feb. 1990.

[47] Langdon, G.G., "An Introduction to Arithmetic Coding" IBM J. Res. Develop. 28, 2(Mar. 1984), 135-149.

[48] LeGall, D., "MPEG: A Video Compression Standard for Multimedia Applications", Communication of ACM, 34(4), (1991), 46-58.

[49] Liu, J.W., "Algorithms for Parsing Search Queries in Inverted File Document Retrieval Systems", ACM Trans. Database Systems, 1, (1976), 299-316.

[50] Longo, G. and Galasso, G., "An Application of Informational Divergence to Huffman Codes", IEEE Trans. Inform. Theory, IT-28(1) (Jan. 1982), 36-43.

[51] Moffat, A., "Word-Based Text Compression", Software-Practice and Experience, 19 (Feb. 1989), 185-198.

[52] Nelson, M.R., "Arithmetic Coding and Statistical Modeling" Dr. Dobb's Journal, (February 1991), 16-29.

[53] Nelson, M., "The Data Compression Book", M&T Publishing Inc. USA, 1991.

[54] Nichols, S.J.V., "Getting Your Byte's Worth", Byte, (Nov. 1990), 331-336.

[55] Nichols, S.J.V., "Saving Space", Byte, (Mar. 1990), 237-243.

[56] Norwood, E., "The Number of Different Possible Compact Codes," IEEE Trans. Inform. Theory IT-13 (Oct. 1967).

[57] Parker, D.S., "Conditions for Optimality of the Huffman Algorithm", SIAMJ. Computer 9(3), (Aug. 1980), 470-489.

[58] Rahman, S.M., Ahmed, M., "Bangali Alphanumeric Segment And Dot Matrix Display", M. Sc. Thesis, CSE Deptt., BUET, 1986.

[59] Reif. J., and Storer,J.A., "A Parallel Architecture for High Speed Data Compression", Journal of Parallel & Distributed Computing, 13, (1992), 222-227.

[60] Rissanen, J. and Langdon, G.G., "Universal Modeling and Coding", IEEE Trans. Inform. Theory, IT-27, 1(Jan 1981), 12-23.

[61] Rubin, F., "Arithmetic Stream Coding Using Fixed Precision Registers", IEEE Trans. Inform. Theory IT-25 (Nov. 1979), 672-675.

[62] Rubin, R., "Experiments in Text-file Compression", Comm. ACM 19, (1976), 617-623.

[63] Ruth, S., and Kreutzer, P., "Data Compression for Large Business Files", Datamation, 18(9), (1972), 62-66.

[64] Severance, D.G., "Practical Guide to Data Base Compression", Information Systems, 8, 1(1983), 51-62.

[65] Shannon, C.E., "A Mathematical Theory of Communication, "Bell Syst. Tech. J. 27 (July 1948), 398-403.

[66] Shannon, C.E., "Prediction of Entropy of Printed English", Bell Sysb. Tech. J. 30 (1951), 50-64.

[67] Sijstermans, F., and van der Meer, J., "CD-1 Full-motion Video Encoding on a parallel computer", Comm. of ACM 34(4) (1991), 81-91.

[68] Simon, B., "Squeeze Play", PC Magazine, (October 15, 1991), 291-312.

[69] Stevens, A., "CUA and Datacompression", Dr. Dobb's Journal, (February 1991), 135-136,138,140,142.

[70] Suen, C.Y., "n-Gram Statitics for Natural Language Understanding and Text Processing", IEEE Trans. Pattern Anal. and Mach. Intelligence, PAMII-1(2), (1979), 164-172.

[71] Swan, T., "Alien Text-file Compression", Dr. Dobb's Journal, (July 1993), 121-123.

[72] Tanaka, H., "Data Structure of Huffman Codes and Its Application to Efficient Encoding and Decoding", IEEE Trans. Inform. Theory IT-33(Jan. 1987), 154-156.

[73] Thomas, K., "Entropy", Dr. Dobb's Journal, (Feb. 1991), 32-34.

[74] Trivedi, K.S., "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1982.

[75] Usher, M.J., "Information Theory for Information Technologists", MacMillan Publishers Ltd., London, 1984.

[76] Ushijima, K. "Steps to an Efficient Program for Floating-point Summation", Software-Practice and Experience, 7, (1977), 759-769.

[77] Vaughan, S.J. and Nichols, "Saving Space", BYTE (March 1990), 237-243.

[78] Vitter, J.S., "Design and Analysis of Dynamic Huffman Codes", J. ACM 34, 4(Oct. 1987), 825-845.

[79] Vitter, J.S., "Dynamic Huffman Coding", ACM Trans. Math. Software 15, 2(June 1989), 158-167, also appears as Algorithm 673, Collected Algorithms of ACM, 1989.

[80] Weiss, J. and Schremp, D., "Putting Data on a Diet", IEEE Spectrum, (August 1993), 36-39.

[81] Welch, T., "A Technique for High Performance Data Compression", IEEE Computer, 17(6), (1984), 8-19.

[82] Wells, M., "File Compression Using Variable Length Encoding", Computer Journal, 15, 4(1973), 308-813.

[83] Yannakoudakis, E.J., Goyal, P. and Huggill, J.A., "The Generation and Use of Text Fragments for Data Compression", Information Processing and Management, 18, 1(1982), 15-21.

[84] Yeung, R.W., "Local Redundancy and Progressive Bounds on the Redundancy of a Huffman Code", IEEE Trans. Inform. Theory, IT-37, 3(May 1991), 687-690.

[85] Zimmerman, S., "An Optimal Search Procedure", Amer. Math. Monthly, 66, (1959), 690-693.

[86] Ziv, J. and Lempel, A., "Compression of Individual Sequences via Variable-Rate Coding", IEEE Trans. Inform. Theory, IT-24, 5(Sep 1978), 530-536.

[87] Ziv, J. and Lempel, A., "A Universal Algorithm for Sequential Data Compression", IEEE Trans. Inform. Theory, IT-23, 3(May 1977), 337-343.

APPENDIX - A

BSCII (Bangla Standard Code for Information Interchange) codes

HEX		00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
	DEC	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0	0				০	[illegible]	ঙ	প	ং						[illegible]	[illegible]	
1	1			!	১	অ	চ	ফ	ঃ						[illegible]	[illegible]	
2	2			"	২	আ	ছ	ব	ঁ					া	[illegible]	[illegible]	
3	3			#	৩	ই	জ	ভ						ি	[illegible]	[illegible]	
4	4			৳	৪	ঈ	ঝ	ম						ী	[illegible]	[illegible]	
5	5			%	৫	উ	ঞ	য						ু	[illegible]	[illegible]	
6	6			÷	৬	ঊ	ট	র						ূ	[illegible]	[illegible]	
7	7			'	৭	ঋ	ঠ	ল						ৃ	[illegible]	[illegible]	
8	8			(	৮	এ	ড	শ	[					ে	[illegible]	[illegible]	
9	9			)	৯	ঐ	ঢ	ষ	|					ৈ	[illegible]	[illegible]	
A	10			*	:	ও	ণ	স	]					ে া	[illegible]	[illegible]	
B	11			+	;	ঔ	ত	হ	{					ে ৗ	[illegible]	[illegible]	
C	12			,	<	ক	থ	ড়	X					[illegible]	[illegible]	[illegible]	
D	13			−	=	খ	দ	ঢ়	}					[illegible]	[illegible]	[illegible]	
E	14			.	>	গ	ধ	য়	[illegible]					[illegible]	[illegible]	[illegible]	
F	15			/	?	ঘ	ন	ৎ	[illegible]					[illegible]	[illegible]		

APPENDIX - B

Sample Listing of Source Code. Vitter algorithm and related routines.

```
// COMDECOM.H: Header file for compression and decompression routine.
#ifndef _COMDECOM_H
#define _COMDECOM_H

#include "tech.h"

enum action_t {cmp, dcmp};

class ComDecom : public Technique {
protected:
        char *tfspec, *cfspec;
        FILE *tfp, *cfp;
        long tcount, scount, ccount;
        clock_t stime, etime;
public:
        ComDecom (enum action_t act, int margc, char* margv[]);
        ~ComDecom (void){};
        void open_comp (void);
        void open_decomp (void);
        void compress (void);
        void decompress (void);
        void report  (void);
};

#endif

// COMDECOM.CPP: Implementation of ComDecom Class member functions
#include <conio.h>
#include <stdio.h>
#include <alloc.h>
#include "comdecom.h"
#include "tech.h"
#include "util.h"

// Constructor of ComDecom class
ComDecom :: ComDecom (enum action_t act, int margc, char* margv[]):Technique()
{
        if (margc < 3) usage (margv[0]);
        if (act == cmp) {
                strcpy (tfspec, margv[1]);
                strcpy (cfspec, margv[2]);
        }
        else {
                strcpy (cfspec, margv[1]);
                strcpy (tfspec, margv[2]);
        }
}

// Open files for compression
void ComDecom :: open_comp (void)
{
        disp_scr (" Compressing ", tfspec, cfspec, tech, scaled, model);

        tfp = fopen(tfspec, "rb");
        if (tfp == NULL) error(1, "Source text file opening error ...");

        cfp = bfopen(cfspec, "wb");
        if (cfp == NULL) error(1,"Terget code file opening error...");
}
```

```
// Open files for decompression
void ComDecom :: open_decomp (void)
{
        disp_scr (" Decompressing ", tfspec, cfspec, tech, scaled, model);

        tfp = fopen(tfspec, "wb");
        if (tfp == NULL) error(1, "Terget text file opening error ...");

        cfp = bfopen(cfspec, "rb");
        if (cfp == NULL) error(1,"Source code file opening error...");
}

// Compress source string to code string.
void ComDecom :: compress (void)
{
        stime = clock();
        Technique::compress (tfp, cfp, &tcount, &scount, &ccount);
        etime = clock();
}

// Decompress code string to original string
void ComDecom :: decompress (void)
{
        stime = clock();
        Technique::decompress (cfp, tfp, &tcount, &scount, &ccount);
        etime = clock();
}

// Reports the compression efficiency and times
void ComDecom :: report (void)
{
        Utility::report(stime, etime, tcount, scount, ccount);
}

//  COMPRESS.CPP: Main files for Compression
#include "comdecom.h"
#include "tech.h"

void main(int argc, char *argv[])
{
        ComDecom comp(cmp, argc, argv);

        comp.open_comp ();
        comp.compress ();
        comp.report ();
}

// EXPAND.CPP : Main file for decompression
#include "comdecom.h"
#include "tech.h"

void main(int argc, char *argv[])
{
        ComDecom decomp(dcmp, argc, argv);

        decomp.open_decomp ();
        decomp.decompress ();
        decomp.report ();
}
```

```
// TECH.H : for Vitter algorithm

#ifndef _TECH_H
#define _TECH_H

#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include "util.h"

#define  EOS  257
#define  N    258
#define  N1   (N+1)
#define  N2   (N*2)

class Technique : public Utility {
private:
        int m, r, e, z;
        int alpha[N1], rep [N1];
        int block[N2];

        long int weight[N2];
        int parent[N2], parity[N2], rtChild[N2], first[N2], last[N2];
        int prevBlock[N2], nextBlock[N2];

        int availBlock;
        int stack [N1];

        int q, leafToIncreament, bq, b, oldParent, oldParity;
        int slide, nbq, par, hpar;

protected:
        char tech[50], scaled[5], model[50];
public:
        Technique(void);
        void initialize(void);
        void update(int k);
        void encode(int k);
        unsigned decode(void);

        int  FindChild (int j, int parity);
        void InterchangeLeaves (int e1, int e2);
        void FindNode (int k);
        void SlideAndIncreament (void);

        void compress (FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
        void decompress (FILE *fi, FILE *fo, long *tc, long *sc, long *cc);
};
#endif
```

```
/*   VITT(U).CPP:Vitter algorithm with unscaled symbol counts */

#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include "comdecom.h"
#include "tech.h"

Technique :: Technique(void) : Utility()
{
     strcpy(tech, "Vitter Algorithm.");
     strcpy(scaled, "No.");
     strcpy(model, "Dynamic 0-order model.");
}
```

```
// Initialize the Vitter model
void Technique::initialize(void)
{
      int i;

      m = 0;
      e = 0;
      r = - 1;
      z = 2 * N - 1;

      for(i = 1; i <= N; i++) {
           m += 1;
           r += 1;
           if(m == 2*r) {
               e += 1;
               r = 0;
           }
           alpha[i] = i;
           rep[i] = i;
      }

      block[N] = prevBlock[1] = nextBlock[1] = 1;
      weight[1] = 0L;
      first[1] = N;
      last[1] = N;
      parity[1] = 0;
      parent [1] = 0;
      rtChild[1] = 0;
      availBlock = 2;
      for (i = availBlock; i < z; i++)
          nextBlock[i] = i + 1;
      nextBlock [z] = 0;
}

// Find the Right child of the node j
int Technique::FindChild (int j, int parity)
{
      int delta, right, gap;

      delta = 2 * (first [block [j]] - j) + 1 - parity;
      right = rtChild [block [j]];
      gap = right - last [block [right]];
      if (delta <= gap) {
           return (right - delta);
      }

      else {
           delta = delta - gap - 1;
           right = first [prevBlock [block [right]]];
           gap = right - last [block [right]];
           if (delta <= gap)
                return (right - delta);
           else
                return (first [prevBlock [block [right]]] - delta + gap + 1);
      }
}

//  Interchange the nodes e1 and e2
void Technique::InterchangeLeaves (int e1, int e2)
{
      int temp;

      rep [alpha [e1]] = e2;
      rep [alpha [e2]] = e1;
      temp = alpha [e1];
      alpha [e1] = alpha [e2];
      alpha [e2] = temp;
}
```

```
// Update model for the symbol k
void Technique::update(int k)
{
     FindNode(k);

     while (q > 0)
          SlideAndIncreament();
     if (leafToIncreament != 0) {
          q = leafToIncreament;
          SlideAndIncreament();
     }
}

//  Find the node corresponding to the symbol k
void Technique::FindNode (int k)
{
     q = rep[k];
     leafToIncreament = 0;
     if (q <= m) {
          InterchangeLeaves (q, m);
          if (r == 0) {
               r = m/2;
               if (r > 0) e -= 1;
          }
          m--;
          r--;
          q = m + 1;
          bq = block [q];
          if (m > 0) {
               block [m] = bq;
               last [bq] = m;
               oldParent = parent [bq];
               parent [bq] = m + N;
               parity [bq] = 1;
               b = availBlock;
               availBlock = nextBlock [availBlock];
               prevBlock [b] = bq;
               nextBlock [b] = nextBlock [bq];
               prevBlock [nextBlock[bq]] = b;
               nextBlock [bq] = b;
               parent[b] = oldParent;
               parity [b] = 0;
               rtChild [b] = q;
               block [m + N] = b;
               weight [b] = 0L;
               first [b] = m + N;
               last [b] = m + N;
               leafToIncreament = q;
               q = m + N;
          }
     }
     else {
          InterchangeLeaves (q, first[block[q]]);
          q = first[block[q]];
          if ((q == (m + 1)) && (m > 0)) {
               leafToIncreament = q;
               q = parent [block [q]];
          }
     }
}


// Slide the current node to the next block
void Technique::SlideAndIncreament (void)
{
     bq = block[q];
     nbq = nextBlock[bq];
     par = parent[bq];
     oldParent = par;
```

```
oldParity = parity[bq];

if (((q <= N) && (first[nbq] > N) && (weight[nbq] == weight[bq])) ||
     ((q > N) && (first[nbq] <= N) && (weight[nbq] == weight[bq] + 1L)))
{
     slide = 1;
     oldParent = parent[nbq];
     oldParity = parity[nbq];
     if (par > 0) {
          bpar = block[par];
          if (rtChild[bpar] == q)
               rtChild[bpar] = last[nbq];
          else
               if (rtChild [bpar] == first [nbq])
                    rtChild [bpar] = q;
               else
                    rtChild [bpar] += 1;
          if (par != z) {
               if (block [par + 1] != bpar)
                    if (rtChild [block[par +1]] == first [nbq])
                         rtChild [block [par + 1]] = q;
                    else if (block [rtChild [block [par + 1]]] == nbq)
                         rtChild [block [par + 1]] += 1;
          }
     }
     parent [nbq] += - 1 + parity [nbq];
     parity [nbq] = 1 - parity [nbq];
     nbq = nextBlock [nbq];
}
else
     slide = 0;

if ((((q <= N) && (first [nbq] <= N)) ||
       ((q > N) && (first [nbq] >  N))) &&
      (weight [nbq] == weight [bq] +  1L))
{
     block [q] = nbq;
     last [nbq] = q;
     if (last [bq] == q) {
          nextBlock [prevBlock [bq]] = nextBlock [bq];
          prevBlock [nextBlock [bq]] = prevBlock [bq];
          nextBlock [bq] = availBlock;
          availBlock = bq;
     }
     else {
          if (q > N) rtChild [bq] = FindChild (q - 1, 1);
          if (parity [bq] == 0) parent [bq] -= 1;
          parity [bq] = 1 - parity[bq];
          first [bq] = q - 1;
     }
}
else if (last [bq] == q) {
     if (slide) {
          prevBlock [nextBlock [bq]] = prevBlock [bq];
          nextBlock [prevBlock [bq]] = nextBlock [bq];
          prevBlock [bq] = prevBlock [nbq];
          nextBlock [bq] = nbq;
          prevBlock [nbq] = bq;
          nextBlock [prevBlock [bq]] = bq;
          parent [bq] = oldParent;
          parity [bq] = oldParity;
     }
     weight [bq] += 1;
}
else {
     b = availBlock;
     availBlock = nextBlock [availBlock];
     block [q] = b;
     first [b] = q;
```

```
        last  [b] = q;
        if (q > N) {
            rtChild [b] = rtChild [bq];
            rtChild [bq] = FindChild (q - 1, 1);
            if (rtChild [b] == (q - 1))
                parent [bq] = q;
            else
                if (parity [bq] == 0)
                    parent [bq] -= 1;
        }
        else
            if (parity [bq] == 0)
                parent [bq] -= 1;
        first [bq] = q - 1;
        parity [bq] = 1 - parity [bq];
        prevBlock [b] = prevBlock [nbq];
        nextBlock [b] = nbq;
        prevBlock [nbq] = b;
        nextBlock [prevBlock [b]] = b;
        weight [b] = weight [bq] + 1;
        parent [b] = oldParent;
        parity [b] = oldParity;
    }
    if (q <= N)
        q = oldParent;
    else
        q = par;
}

// Encode the symbol j
void Technique::encode (int j)
{
    int i, ii, q, t, root;

    q = rep [j];
    i = 0;
    if (q <= m) {
        q = q - 1;
        if ( q < 2 * r)
            t = e + 1;
        else {
            q -= r;
            t = e;
        }
        for (ii = 1; ii <= t; ii++) {
            i++;
            stack [i] = q % 2;
            q = q /2;
        }
        q = m;
    }
    if (m == N)
        root = N;
    else
        root = z;

    while (q != root) {
        i++;
        stack[i] = (first [block [q]] - q + parity [block [q]]) % 2;
        q = parent [block [q]]
             - (first [block [q]] - q + 1 - parity [block [q]]) / 2;
    }
    for (ii = i; ii >= 1; ii--) fputb(stack [ii]);
}
```

```
// Decode the next symbol from the code string
unsigned int Technique::decode (void)
{
     int i, q;

     if (m == N)
          q = N;
     else
          q = z;

     while (q > N) {
          q = FindChild (q, fgetb());
     }

     if (q == m) {
          q = 0;
          for (i = 1; i <= e; i++) q = 2 * q + fgetb ();
          if (q < r)
               q = 2 * q + fgetb ();
          else
               q += r;
          q++;
     }
     return (alpha [q]);
}

// UTIL.H: Header file for utility routines.
#ifndef _UTIL_H
#define _UTIL_H
#include <stdio.h>
#include <time.h>

class Utility {
private:
        int dx1, dx2, dy1, dy2, dy3;
//protected:
        FILE *file;
        unsigned char mask;
        int rack;
        long int ccount;
public:
        Utility(void) {
                dx1 = 14;
                dx2 = 28;
                dy1 = 2;
                dy2 = 3;
                dy3 = 9;
        }
        void disp_scr (const char *act,  const char *fnt, const char *fnc,
                   const char *tech, const char *scaled, const char *model);
        char* getfname (const char *path);
        void usage(char *comm);
        void report (clock_t stime, clock_t etime,
                long tcount, long scount, long ccount);
        void error (const int flag, const char *message);
        void outcode (const long scount, const long ccount);
        void outtcount (const long tcount);

        FILE *bfopen(const char *name, const char *mode);
        void fputb (int bit);
        void fputbs   (unsigned long code, int count);
        int  fgetb    (void);
        unsigned long  fgetbs (int bit_count);
        void bfclose  (void);
        void hfflush  (void);
        void fprintbs (FILE *file, unsigned int code, int bits);
        long int getccount(void);
};
#endif
```

```
// UTIL.CPP: Member functions Implementation for Utility Class.
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <dir.h>
#include "util.h"

// Open file for bit oriented I/O operation.
FILE *Utility::bfopen (const char *name, const char *mode)
{
        file = fopen(name, mode);
        rack = 0;
        mask = 0x80;
        ccount = 0;
        return file;
}

// Close binary file
void Utility::bfclose(void)
{
        fclose(file);
}

// Flushes the remaining bits of the code file.
void Utility::bfflush(void)
{
        if(mask != 0x80)
                if(putc(rack, file) != rack) {
                        error(1, "Error in writing code . . .");
                }
                else {
                        outcode(0, ++ccount);
                }
        fflush(file);
}

// Output a single bit.
void Utility::fputb  (int bit)
{
        if(bit)
                rack ¦= mask;
        mask >>= 1;
        if(mask == 0) {
                if(putc(rack, file) != rack)
                        error(1, "Fatal error in writing bit ...");
                else {
                        outcode(0, ++ccount);
                }
                rack = 0;
                mask = 0x80;
        }
}

// Output a group of bits.
void Utility::fputbs (unsigned long code, int count)
{
        unsigned long testbit;

        testbit = 1L << (count - 1);
        while (testbit != 0) {
                if(testbit & code)
                        rack ¦= mask;
                mask >>= 1;
                if(mask == 0) {
                        if(putc(rack, file) != rack)
                                error(1, "Fatal error in writing bit ...");
                        else {
                                outcode(0, ++ccount);
```

```
                        }
                        rack = 0;
                        mask = 0x80;
                }
                testbit >>= 1;
        }
}

// Input a single bit.
int Utility::fgetb(void)
{
        int value;

        if(mask == 0x80) {
                rack = getc(file);
                if (rack == EOF)
                        error (1, "Fatal error in reading bit ...");
                else {
                        outcode(0, ++ccount);
                }
        }
        value = rack & mask;
        mask >>= 1;
        if (mask == 0)
                mask = 0x80;
        return (value ? 1 : 0);
}

// Input a number of bits.
unsigned long Utility::fgetbs (int bit_count)
{
        unsigned long testmask;
        unsigned long return_value;

        testmask = 1L << (bit_count - 1);
        return_value = 0;
        while (testmask != 0) {
                if(mask == 0x80) {
                        rack = getc(file);
                        if (rack == EOF)
                                error(1, "Fatal error in reading bits...");
                        outcode(0, ++ccount);
                }
                if(rack & mask)
                        return_value |= mask;

                testmask >>= 1;
                mask >>= 1;
                if(mask == 0)
                        mask = 0x80;
        }
        return(return_value);
}

// print a integer data as a group of bit.
void Utility :: fprintbs(FILE *file, unsigned int code, int bits)
{
        unsigned int mask;

        mask = 1 << (bits - 1);
        while (mask != 0){
                if(code & mask)
                        fputc('1', file);
                else
                        fputc('0', file);
                mask >>= 1;
        }
}
```

```
long int Utility::getccount(void)
{
        return ccount;
}

// Display the screen for compression and decompression operation.
void Utility::disp_scr (const char *act,  const char *fnt, const char *fnc,
                        const char *tech, const char *scaled, const char *model)
{
        const int x1 = 14, y1 = 7, x2 = 67, y2 = 19;
        const int lt = 201, lb = 200, rt = 187, rb = 188, hor = 205, ver = 186;

        int x, y;

        clrscr();
        /* draw the box */
        gotoxy(x1,y1);putch(lt);
        for (x = x1 + 1; x < x1 + 5; x++) putch(hor);
        printf(act);
        for(x = wherex(); x < x2; x++) putch(hor); putch(rt);
        for(y = y1+1; y <y2; y++ ){
                gotoxy(x1,y); putch(ver);
                gotoxy(x2,y); putch(ver);
        }
        gotoxy(x1,y2); putch(lb);
        for (x = x1 + 1; x < x2; x++) putch(hor);
        putch(rb);

        /* writing message */
        window(x1+1, y1+1, x2-1, y2-1);

        gotoxy(13,1);
        cprintf(" Name          Size\n\r");
        cprintf(" Text File : %s\n\r",getfname(fnt));
        cprintf(" Code File : %s\n\r\n\r", getfname(fnc));
        cprintf(" Technique : %s\n\r", tech);
        cprintf(" Scaled    : %s\n\r", scaled);
        cprintf(" Model     : %s\n\r\n\r", model);
        cprintf(" Efficiency:");

        gotoxy(wherex()+10,wherey());
        cprintf("Time:\n\r\n\r Message   :");
}

// Seperate file name from the full path name.
char* Utility::getfname (const char *path)
{
        char p[MAXPATH];
        char drive[MAXDRIVE];
        char dir[MAXDIR];
        char file[MAXFILE];
        char ext[MAXEXT];
        strcpy(p,path);
        fnsplit(p, drive,dir, file, ext);
        fnmerge(p, "",   "",  file, ext);
        return p;
}

void Utility::usage(char *comm)
{
        printf("\nSyntax: %s <SourceFile> <TergetFile>", getfname(comm));
        exit(-1);
}
```

```
// report times and compression efficiency.
void Utility::report (clock_t stime, clock_t etime,
               long tcount, long scount, long ccount)
{
       double dtime;

       dtime = (double)(etime - stime)/(double)CLK_TCK;
       outcode(scount, ccount);
       gotoxy(dx1, dy3);
       cprintf ("%5.2f%%", (1.0-(double)(scount+ccount)/(double) tcount)*100.0);
       gotoxy  (dx2, dy3);
       cprintf(" %0.3f seconds.", dtime);
//     error(0, "End of compression ...");
       /*
       getch();
       window(1,1,80,25);
       clrscr();
       */
}

// Give error message.
void Utility::error (const int flag, const char *message)
{
       gotoxy(14,11);
       clreol();
       textattr(0x87);
       cprintf(message);
       textattr(0x07);
       if(flag) {
//             getch();
//             window(1,1,80,25);
//             clrscr();
               exit(-1);
       }
}

// Display static counts on the screen.
void Utility::outcode (const long scount, const long ccount)
{

       gotoxy(dx2, dy2);
       if(scount && ccount){
               cprintf("%ld",  scount+ccount);
               gotoxy(dx2, dy2+1);
               cprintf("[%ld + %ld]", scount,ccount);
       }
       else if(scount)
               cprintf("%ld", scount);
       else
               cprintf("%ld", ccount);
}

// Display compression counts.
void Utility::outtcount (const long tcount)
{
       gotoxy(dx2, dy1);
       cprintf("%ld", tcount);
}
```