DATA COMPRESSION USING THE LEMPEL-ZIV-WELCH DYNAMIC DICTIONARY SCHEME

BY

FALOWO, OLABISI EMMANUEL {*B.Eng. (Hons.),FUTA*}
EEE/92/3590

A THESIS SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING, FEDERAL UNIVERSITY OF TECHNOLOGY, AKURE

IN PARTIAL FULFILMENT OF THE AWARD OF MASTER OF ENGINEERING (M.Eng) DEGREE IN ELECTRICAL AND ELECTRONICS ENGINEERING (COMMUNICATION ENGINEERING OPTION) OF THE FEDERAL UNIVERSITY OF TECHNOLOGY, AKURE, ONDO STATE, NIGERIA.

AUGUST 2003

DEDICATION

To God be the glory great things He has done. This thesis is dedicated to the Almighty God.

ACKNOWLEDGEMENT

I give glory, honour and adoration to the Almighty God, who is the Alpha and Omega of this work. I give Him all praises for He only is worthy.

I wish to express my profound gratitude to my supervisor, Dr V.S.A. Adeloye for his support, advice and immense contributions towards the success of this work. My sincere appreciation goes to all the staff of the department of Electrical and Electronics Engineering, FUTA.

I am highly indebted to Dr Adeniran, Engr Adewumi, Engr Alatise, Dr. Oguntuase , Engr Omosule and for their various contributions.

I grateful to Pastor Oloruntoyin and the members of the Deeper Life Bible Church Ilara-Mokin for their prayers and moral support. Special thanks to Mr. Omolofe Babatope, My Fafila and all my course mates; Mr. Popoola, Mr. Ale, Mr. Ponnle, Mr. Akingbade, Mr. Ogidiolu,, and Mrs. Abe.

To my loving and caring parents, Mr. and Mrs. J.O. Falowo, I am very grateful for their assistance. May God bless them. My special thanks goes to my sisters Mrs. Omole, Mrs. Orisagbemi and Idowu and to my brothers Dele, Taiwo and Kehinde.

Finally I thank everyone who has contributed to the success of this work.

TABLE OF CONTENTS

CERTIFICATION

This is to certify that this research was carried out by **Falowo, Olabisi Emmanuel** (EEE/92/3590) and approved as meeting the requirement for the award of Master of Engineering in Electrical and Electronics Engineering (Communication Engineering Option) of the Department Electrical and Electronics Engineering, Federal University of Technology, Akure.

Bisi Adeloye

Dr. V.S.A. Adeloye
(Supervisor)

9-9-2003.
Date

Bisi Adeloye

Dr V.S.A. Adeloye
(H.O.D, Electrical and Electronics Engineering)

9-9-2003.
Date

CHAPTER THREE

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This report focuses on the development a computer application program for data compression using the Lempel-Ziv-Welch (LZW) Dynamic Dictionary Scheme. The uncompressed media data such as text, audio, graphics, image and video often require high transmission bandwidth and considerable storage capacity. To provide a feasible and cost-effective solution for the current data storage requirement, compressed data are stored. LZW Dynamic Dictionary Scheme exploits the advantage of repetition of strings in the input stream for data compression. Different dictionary sizes of 10, 11, 12, 13, 14, 15, and 16 bits are used for data compression in this research. The American Standard Code for Information Interchange (ASCII) code is used to assign codes for single characters in the dictionaries while the "xor" type hashing function is used to form array addresses for groups of bytes. Compression ratios for different types of input data are computed and the variation in compression ratio with dictionary size is investigated to determine the optimum dictionary for data compression.

CHAPTER ONE

INTRODUCTION

Signals such as text, speech and image data are stored and processed in computer as files or collection of bits. Large files have many disadvantages in comparison with small files. They fill storage devices, they take longer time to transmit to users, and they can overwhelm algorithms designed to draw conclusions from the raw data, Madisetti. (1995). Therefore it is desirable and sometimes necessary to reduce the number of bits requires to describe a signal, but it is equally clear that this must be done with great care such that it does not result in a loss of vital information.

Data compression is the process of reducing the number of bits required to describe a signal to a prescribed accuracy. In some situations the signal is already in digital form while in many cases, the signal is analog electrical voltage or current produced by a microphone, TV camera or other transducers. However analog signal must be converted into digital signal before compression techniques can be applied.

Data compression has been a practice of mankind since the ancient time. Ancient Chinese used special concise character syntax "wen-yan" to record daily spoken language in short form, where the essence of meaning was still preserved. In this way, redundant characters not affecting meaning were discarded and therefore precious paper were saved. In the 18^{th} century, British Admiralty used a compress language format in telegraphy, and this saved time in the communication between London and important ports. Eskimos have over 200 words for different description of snow; so one word is enough for them to state the type of snow precisely. Since the advent of digital computers in the 20^{th} century, data communication and data compression are becoming more and more important. The amount of data collections has grown in the recent years mainly due to the widespread use of digital or virtual libraries, documented databases and the web. Current text

databases contain hundred of gigabytes and the web is measured in terabytes. Data compression is the key to the future expansion on the web; it is certainly the key to emerging multimedia and 3-D technology, Brown, (1998). Data compression has many practical application areas such as fax, digital cameral, compact discs (CD), space communication, remote control, teleconferencing, videophone, digital voice storage etc.

1.1 DATA COMPRESSION SYSTEM

A data compression system comprises a compression and decompression processes. A general data compression system is illustrated in fig. 1.1. As show in fig. 1.1, the information source data, S, is first transformed by the compression process to compressed data, which usually is a more compact representation of the source data. The compact form of data offers tremendous advantages in both communication and storage applications. For example, in communication applications, the compressed data is transmitted to a receiver through a communication channel with lower bandwidth as a result; more users can efficiently utilized the same bandwidth that is initially used by a single person for transmitting uncompressed data. In storage application, the compressed data takes up less space as shown in figures 1.2 and figure 1.3. The stored data can be retrieved whenever they are needed. After the transmitted (or stored data) is received (retrieved), it goes through a decompression process, which reconstruct the original data with the greatest possible fidelity. In lossy compression systems, the original data, S, cannot be perfectly retrieved from the reconstructed data, S*, which is a close approximation. In lossless compression on the other hand, the original data, s, can be perfectly recovered from the compressed data.

For a lossless compression,

$$S = S^* \quad \text{.....1.1}$$

For a lossy compression,

$$S \approx S^* \quad \text{.....1.2}$$

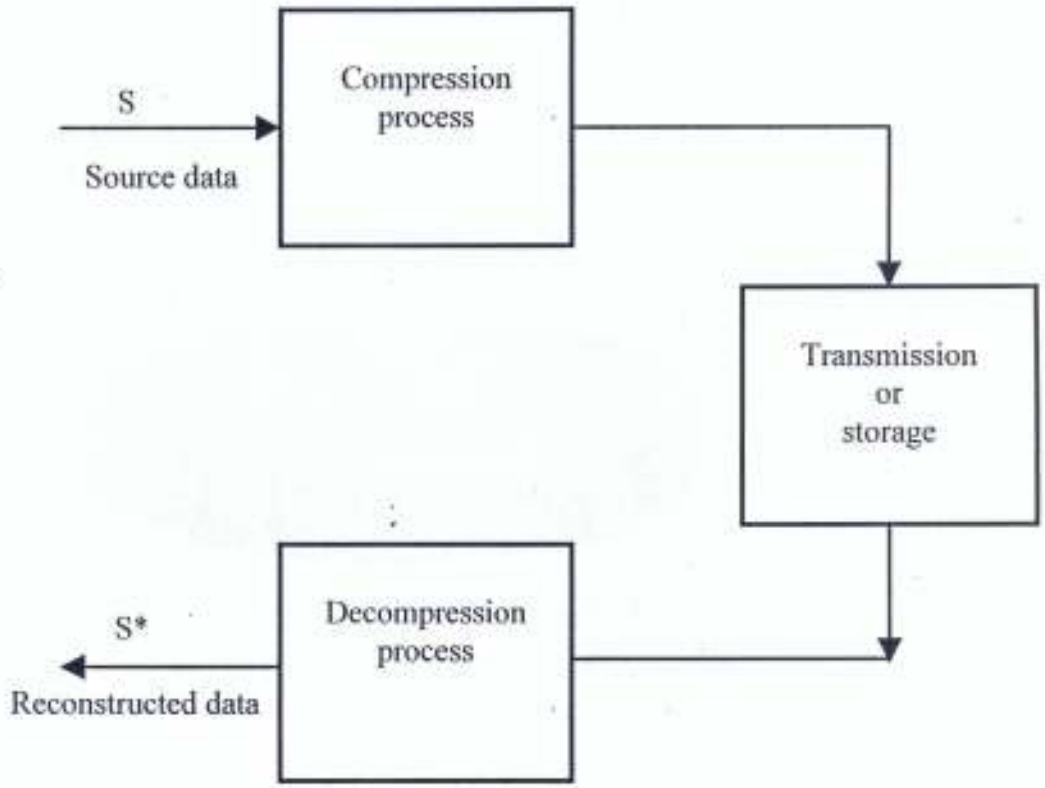

Fig. 1.1 General Data Compression Process

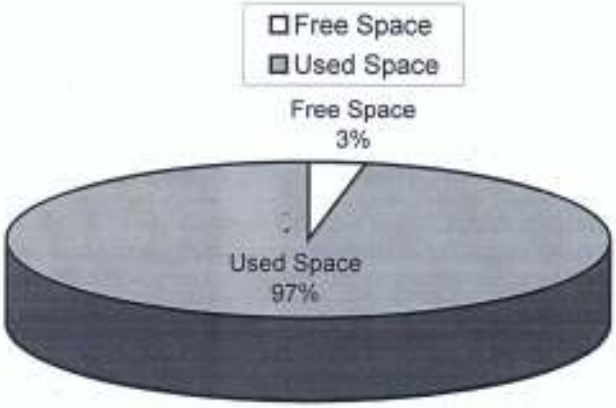

Fig.1.2 Original Data in a Storage Medium

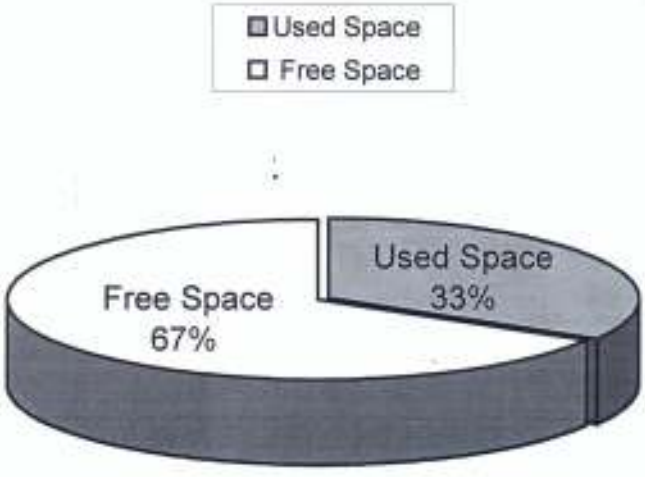

Fig. 1.3 Compressed Data in a Storage Medium

1.2 SOURCE DATA

The source data typically are presented in a format in which each datum is represented by a fixed number of bits. For textual data each character is typically represented in eight bits (one byte) using the American Standard Code for Information Interchange (ASCII). The ASCII code is a long established standard for allowing letters and numbers to be represented in digital form. Each printable character is assigned a number between 32 and 127, while the numbers between 0 and 31(non printable characters) are used for various control actions. For speech, each datum consists of a real valued sample of the waveform expressed to a set precision. Historically, digital speech signals are sampled at a rate of 8000 samples per second where each sample is represented by 8 bits. This corresponds to an uncompressed rate of 64 kbits/sec. For images, each picture element or pixel is represented by 1 bit (for black and white), eight bits (for gray), or 24 bits (for colour images, 8 bits for each of the three primary colours).

1.3 OBJECTIVES OF THE STUDY:

The general objective of this research is to develop a computer application program for data compression in order to store and transmit more data at reduced time and cost.

The specific objectives of the research are to:

1 Evaluate the compression ratio for different types of input data.

2 Determine the effect of variation in dictionary size on the compression ratio.

3 Determine the optimum dictionary size for data compression.

1.4 EXPECTED CONTRIBUTION TO KNOWLEDGE:

The study is expected to show how to highlight optimum dictionary for data compression.

1.5 METHOD OF APPROACH:

This entails

1 Extensive Literature survey.

2 Data compression system development and testing.

1.6 RESEARCH METHODOLOGY:

Lempel-Ziv-Welch (LZW) algorithm was used to develop a computer application data compression program which comprises a compression and decompression processes. Different dictionary sizes of 10,11,12,13,14,15 and 16 bits were used in the compression program. The American Standard Code for Information Interchange (ASCII) code was used to assign codes for single characters in the dictionary while the "xor" type hashing function was used to form array addresses for groups of bytes. The compression program was written in C++ programming language and tested with different types of input data using the different dictionary sizes. The compression ratios were calculated for each dictionary. The resultant data were analyzed and the optimum dictionary for data compression was determined.

CHAPTER TWO

LITERATURE REVIEW

The progress in digital technology has made the widespread use of compressed signals practical while the ever-increasing demand for data storage coupled with the need to effectively utilize transmission bandwidth have made data compression necessary. Data compression transforms information from one representation to another smaller representation from which the original, or a close approximation to it can be recovered. The compression and decompression processes are often referred to as encoding and decoding. A code is a mapping of source messages (words from source alphabet) into codewords (words of the code alphabet). The source messages are the basic units into which the string to be represented is partitioned. These basic units may be single symbols from the source or they may be string of symbols. The success of any data- compression techniques depends upon the properties of the source messages.

2.1 DATA STORAGE

There are three main types of data storage in practice: primary, secondary, and archival storage. The range of data storage systems may consist of a variety of technologies such as the magnetic tape drive, magnetic disk drives, optical tape dives, and optical disk drives such as the compact disk read-only memory (CD-ROM), CD recordable (CD-R), digital versatile disk (DVD), and write-one read-many (WORM).

2.1.1 STORAGE MODE

Data storage may be analog or digital. The audiotape is an example of analog recording, whereas a CD is an example of digital recording. However data compression is applicable to digital recording only.

2.1.2 DATA STORAGE MEDIA

Depending on the media and technology used, storage devices may be classified as semiconductor storage, magnetic storage, optical storage, and magneto-optical storage.

2.1.2.1 SEMICODUCTOR STORAGE

In a computer system, semi conductor storage is used as a fast access medium, which is called primary storage. This is primarily divided into read-write memory (RWM), and read only memory (ROM). RWM historically is called random access memory (RAM), although ROM is also is also random access. RAM access, in contrast with sequential access of a tape, means any piece of data can be accessed randomly without accessing other data items. RAM is volatile in the sense that its contents are destroyed when the power is turned off. The basic building block of a semi conductor storage system is called a flip-flop. Semi conductor memory has become the storage device with the highest density and fastest speed. In practice, data compression is not applicable to primary memory.

2.1.2.2 MAGNETIC STORAGE

Magnetic storage exploits electromagnetism. When an electric current flows through a conductor, it generates field around the conductor. This magnetic field can influence the magnetic material in the field. When the direction of current reverses, the polarity of the magnetic field is also reversed. So, particles of the magnetic substance can be polarized magnetically in one of two directions with an electromagnet. Thus, magnetic polarization can be used to distinguished "0" and "1". This two-way operation of electromagnetism makes it possible to record data on a disk or tape and read the data later. Magnetic disk and tapes are two variants of magnetic storage devices.

MAGNETIC DISC

Magnetic disks, which were preceded by magnetic drums, are of two major types: namely, hard disks and floppy disks. Hard disks cannot be bent or flexed- hence the term hard disk. Also the platters of hard disk are not removable, and for that reason it is also called a fixed disk. Discs consist of circular platters made of glass, plastic, or metal and coated with a magnetic substance. A driver motor rotates the disk platter about its central axis. The disk drive also has head motors, which cause head rotation. Head move radially in and out across the disk surface. Heads are connected to head motors through a moving arm. Usually there is one head per surface. Fig. 2.1 shows the basic components of a typical magnetic disk drive. Fig. 2.2 shows the logical geometry of the platters of a disk. Data are stored in concentric tracks on the surfaces of each platter. The number of tracts is an indicator of the storage capacity of the disk. Track density, which is the number of per unit of radial length, is expressed in tracks per millimeters or tract per inch. A typical disk drive can have more than 2000 tracts per inch (TPI) on its recording surface. A pair of tracks on each side of the disc with the same track number is called a cylinder. In the case of disk packs several tracks may constitute a cylinder. Each track is divided into sectors, which can store a block of data. The process of organizing the disk surface into tracks and sectors is called formatting. In almost all systems sectors typically contains 512 bytes of user data plus addressing information used by the drive electronics. In the earlier hard drive designs, the number of sectors per track was fixed and, because the outer tracks on a platter have larger circumference than the inner tracks, space on the outer tracks was wasted. The number of sectors that would fit on the innermost track constrained the number of sectors per track for the entire platter. However many of the advance drives use a formatting techniques called multiple zone recording that allows the number of sectors per tracks to be adjusted so more sectors on the larger, outer tracks. By dividing the outer tracks into more sectors, data can be packed uniformly throughout the

surface of a platter, disk surface is used more efficiently, and higher capacities can be achieved with fewer platters.

The mechanism for storing and retrieving data is almost the same for both the hard disk and floppy disk. The only difference is that a floppy disk has only one platter and for this, the number of surfaces is two; the top and bottom surface of the single platter. Most hard disks contain several platters, all mounted on the same axis. When data are retrieved from a hard disk drive, a command is issued to open an existing file, and the application program that is running prompts the user to enter the name of the file to open. It then passes the file name to the operating system, which determines where the file is located on the disk drive-the head number, cylinder, and sector identification. The operating system transfers this information to the disk controller, which drives an actuator connected to the actuator arms to position the heads over the right track. As the disk rotates, the appropriate head reads the address of each sector on the track. When the desire sector appears under the read/write head, the entire contents of the sector containing the necessary data are read into the computers main memory. Storing data on a hard drive is a similar process to retrieving data. The host computer operating system is responsible for remembering the addresses for each file the disk and which sectors are available for new data. The operating system instructs the controller where to begin writing information to the disk. The controller moves the read/write heads to the appropriate track and writing begins. When the first track is full, the heads write to the same track on successive platter surfaces. If still more track capacity is required to store all the data, the head moves to the next available tack with sufficient contiguous space and writes the data there. The magnetic disc controller also converts between serial and parallel data. Basically the controller is used to provide an interface between the disc drive and the CPU. Disc performance is specified by the data access time, data transfer rate, data throughput rate and the storage capacity.

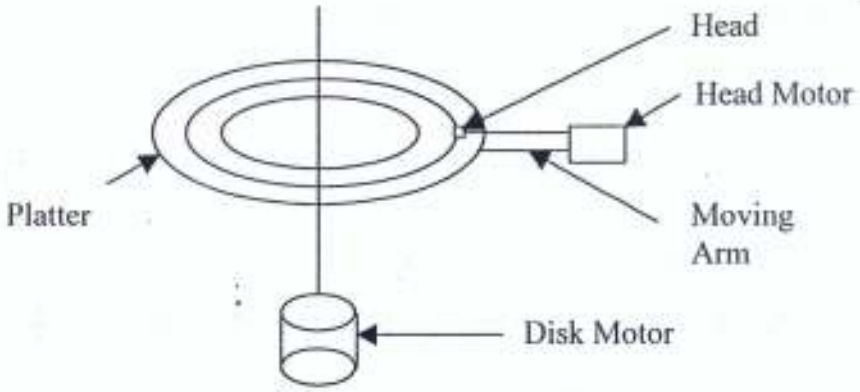

Fig.2.1 Magnetic Disk Drive Components

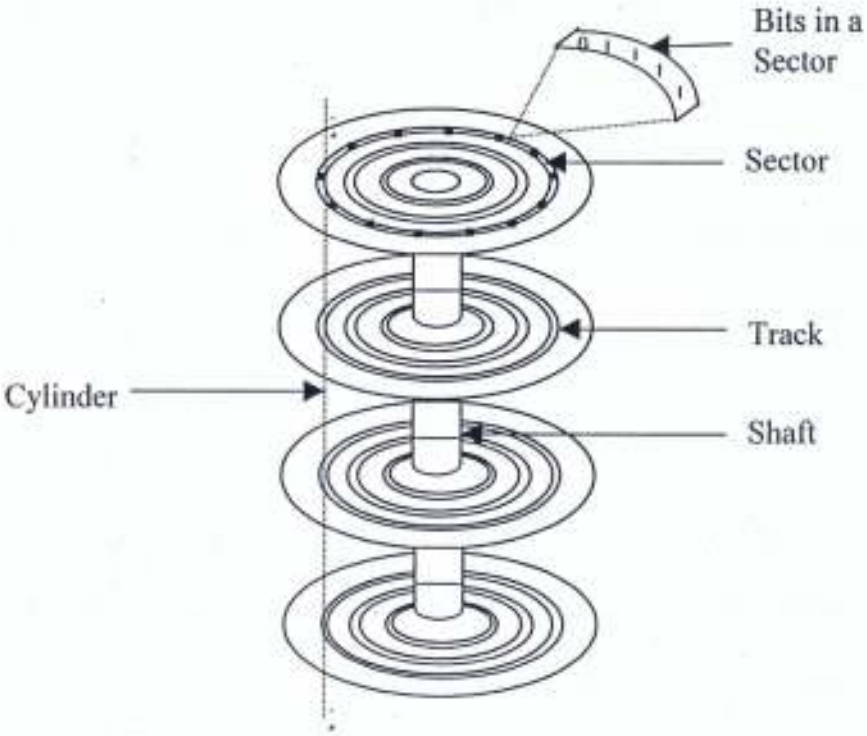

Fig. 2.2 Components of a Disk System

MAGNETIC TAPE

Magnetic tape was commonly used on early mainframe computers; one of the first computer storage technologies was the magnetic tape drive, or simply tape drive. Magnetic tape is a sequential data storage device. To read data, a tape drive winds through the spool of tape to read the exact location of the desired information. To write data, the tape drive encodes it sequentially on the tape. Because tape drives cannot randomly access or write data like disk drives, and are thus very slow, they have been replace as the primary storage device in most computer applications. The main disadvantage of using tape is the long access time

2.1.2.3 OPTICAL STORAGE

Currently, optical memory devices are widely used as secondary memory in computer systems. The most widely known optical storage device is C-D ROM, which has significantly enhance the replication, distribution, and utilization of software, game, audio, and video intended for use in computers or in entertainment instruments. For instance, a CD-ROM containing 680 Mbytes of data can be mass duplicated by injection molding in a few seconds. The main attractive features of an optical data storage system are the removability, portability, and reliability.

OPTICAL DISK

An optical disk is usually constructed from a plastic or glass substrate coated by one or more thin-film layers and contains a number of tracks along which information is stored. The manufacturer may record the information on the substrate surface or information may be recorded by the user on one or more of the thin-film layers along the tracks. The storage layer is generally sandwiched between two dielectric layers and the stack is covered with a metal layer for better reflectivity. The function of the dielectric and reflective layers include reduction of thermal cross talk during writing, optimize

absorptivity and reflectivity, and protecting the storage layer. CD disks encode 0 and 1 data as sports with different reflectivity. Binary digits (0 and 1) are stored on the storage layer as transparent and opaque marks. In optical storage systems, semiconductor laser diodes having the shortest possible wavelength that can provide enough power for read, write, and erase operations for several thousand hours are generally used.

OPTICAL TAPE

An optical tape consists of a substrate onto which an active layer and a dielectric film are sputtered flanked by a back and a top coating for better reliability. The active layer usually consists of thin metallic film or dye polymer or phase change material. Information is hole punch (called marks) into the active layer by a focused semiconductor laser beam. For data read out, the tape is scanned by a focused laser beam and the marks cause the amplitude of the reflected light to vary. These variations are detected by a detector array to generate the appropriate data readout signal. Optical tape is mainly used for high volume storage. It is so thin that very large amounts of data can be stored in a very small volume of it. For instance optical tape has about 25 times the volumetric density of CDs. Although magnetic tape is economical, it is not suitable for long-term archiving because the magnetic medium is degraded by creep, track deformation, and print-through, requiring the transfer of stored information to a new magnetic tape after a given interval. Optical tape is more durable, does not suffer from print-through, and is more reliable than the magnetic tape.

2.1.2.4 MAGNETO-OPTICAL STORAGE

Magneto-optical (MO) disk systems combine the technology of traditional magnetic media, like hard disk drives, with optical disk technology. MO technology allows user to pack hundreds of megabytes of data on a disk that looks similar to a traditional 3.5 inch

floppy disk and typically comes in a 3.5 inch or 5.25 inch form factor. An MO disk is made of materials that cause it to be highly resistant to magnetic fields, or coercive force, at room temperature. An MO drive writes to the disk using a read/write head assisted by a laser. A laser heats up the disk surface to its Curie point. Then, the read/write head passes over the disk polarizing those areas that have been heated by the laser. Because a laser can be focused on a much field than MO disk with the assistance of a laser results in a very high data density not available with traditional hard disk drive technology. During a read operation, the MO drive uses the same laser to sense the data stored on the disk. As the laser scans the disk surface, the drive detects a reflection of light by data bits oriented in one direction and no reflection from the data bits oriented in the opposite direction. Thus an MO drive can distinguish between the 0 and 1 data bits stored on the disk.

2.2 COMMUNICATION CHANNEL.

Communication channel provides electrical connection between the source and the destination as shown in figure 2.3. The channel may be bounded such as the coaxial cable or unbounded such as the free space over which the information bearing signal can be radiated.

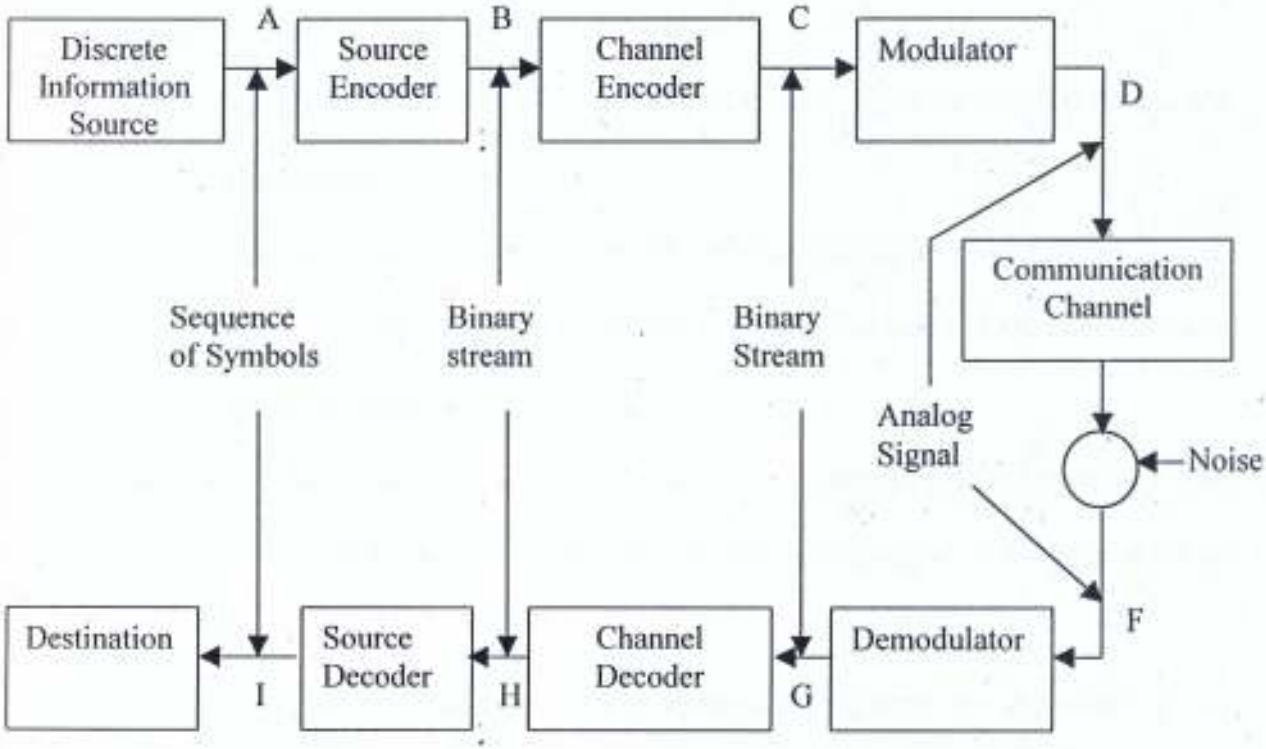

Fig. 2.3 Block Diagram of a Digital Communication System

Due to physical limitations, communication channels have only finite bandwidth (B Hz), and the information-bearing signal often suffers amplitude and phase distortion as it travels over the channel. In addition to the distortion, the signal power also decreases due to the attenuation of the channel and the signal is also corrupted by noise. The capacity of a given channel is defined as the maximum rate at which nearly errorless data transmission is theoretically possible. The following four concepts are very important in establishing the capacity of a communication channel.

(i) Data Rate: This is the rate, in bits per seconds (bps) at which data can be communicated over the channel.

(ii) Signal to Noise Ratio: This is the ratio of the signal power to the power contained in the noise that is present at a particular point in the transmission. This ratio is expressed in decibels.

(iii) Error Rate: This is the rate at which error occurs, where an error is the reception of a 1 when a 0 was transmitted or the reception of a 0 when a 1 was transmitted.

(iv) Bandwidth: All channels have a maximum frequency beyond which input components are almost entirely attenuated. This is due to distributed capacitance and inductance. As the frequencies increase, the parallel capacitance tends to "short out" the signal and the series inductance "open-circuits." Bandwidth is defined as the difference, expressed in the number of cycles per second or hertz, between the two limiting frequencies of a band. A band is defined as more than 3.0 decibels greater than the mean attenuation across the band. Some channels exhibit a low- frequency cutoff due to the effects of distributed capacitance and inductance. If there is a low-frequency

cutoff, the channel can be modeled as a band-pass filter. If there is no low-frequency cutoff, the channel model becomes a low-pass filter.

2.2.1 CHANNEL CAPACITY

Consider the case of a channel that is noise-free. In this case, the limitation on data rate is simply the bandwidth of the signal. A formulation of this limitation, due to Nyquist, states that if the rate of signal transmission is 2W, then a signal with frequencies no greater than W is sufficient to carry the data rate. The converse is also true: Given a bandwidth of W, the highest signal rate that can be carried is 2W. This limitation is due to the effect of inter symbol interference, such as is produced by delay distortions. If the symbol to be transmitted are binary, (two voltage levels), then the data rate can be supported by W Hertz is 2W bps. For example a voice channel with a bandwidth of 3100 Hertz has a capacity of 2W, which is equal to 6200 bps. However, signals with more than two levels can be used; that is, each signal element can represent more than one bit. For example, if four possible voltage levels are used as signal, then each signal elements can represent two bits. With multilevel signaling, the Nyquist formulation becomes

$$C = 2W \log_2 M \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 2.1$$

Where M is the number of discrete signal or levels. Thus, for M = 8, the capacity of the voice channel becomes 18,600 bps. Therefore, for a given bandwidth, increasing the number of different signals can increase the data rate. However this places an increased burden on the receiver: Instead of distinguishing one of two possible signals during each signal time, it must distinguish one of M possible signal. Alternatively, data rate can be increased by data compression. Noise and other impairments on the transmission line limit the practical value of M. For a noiseless channel doubling the

bandwidth doubles the data rate. However the presence of noise can corrupt one or more bits. The higher the data rate, the more damage noise can do. For a given level of noise, greater signal strength would improve the ability to correctly receive data. The key parameter involved is the signal-to-strength ratio (S/N).

$$S/N = 10 \log \text{ (signal power)/(Noise power)} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 2.2$$

This expresses the amount, in decibels, that the intended signal exceeds the noise level. A high signal-to-noise ratio will mean a high- quality signal and a low number of required intermediate repeaters. The signal-to-noise ratio is important in the transmission of digital data because it sets the upper bound on the achievable data rate. Shannon's result is that the maximum channel capacity, in bits per second, obeys equation 2.3

$$C = W \log_2 (1 + S/N) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 2.3$$

Where C is the capacity of the channel in bits per second and W is the bandwidth of the channel in Hertz. This represents the theoretical maximum data rate that can be achieved. In practice, however, only much lower rates are achieved. The capacity indicated in equation 2.3 is called the error-free capacity. Shannon proves that if the actual information rate on a channel is less than the error-free capacity, then it is theoretically possible to use a suitable signal code to achieve error free transmission through the channel. Using data compression technique, more data can be transmitted on the communication channel at a given bandwidth or more users can be allowed on the same bandwidth.

2.3 INFORMATION THEORY

Messages produced by information sources consist of sequences of symbols. While the receiver of a message may interpret the entire message as a single unit, communication system often have to deal with individual symbols. The concept of information content of a particular message is related to predictability. That is the more likely a particular message, the less information is given by transmitting that message. In order to define the information content of symbols, it is important to know the following: first, the instantaneous flow of information in a system may fluctuate widely due to the randomness involved in the symbol selection. Hence there is need to talk about average information content of symbols in a long message. Secondly, the statistical dependence of symbols in a message sequence will alter the average information content of the symbols.

The information content of a source message, X is given as

$$X = -\log_2 P(x) \quad \text{.....2.4}$$

Where P(x) is the probability of the source message.

The average information content per message is called entropy. Shannon established this as the fundamental limit to lossless data compression. This limit is denoted by H. The exact value of H depends on the statistical nature of the information source. It is possible to compress the source, in a lossless manner, with compression rate close to H but it is mathematically impossible to do better than H.

$$\text{Entropy (H)} = \sum_{i}^{n} P(x_i) \log_2(1/P(x_i)) \text{ in bits/ symbol} \quad \text{.....2.5}$$

Where n is the number of source messages.

For maximum transmission efficiency, symbols must be represented with uniquely decipherable codes of minimum length. If the various messages to be transmitted do not have equal probabilities, compression can be achieved by allowing the code words to have unequal length. This is one of the basic principles of data compression.

2.4 DATA COMPRESSION ALGORITHMS.

Algorithm is defined according to the Macmillan Dictionary of Data Communications as a prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps. Data compression algorithms can be classified into two categories: lossless compression and lossy compression. These algorithms are based on ideas that have matured over the last 10-15 years in academia and related industry. In the recent years the emphasis in compression community has shifted from that of developing new techniques to that of software and hardware implementation of these techniques, and the development and establishment of standards to achieve interoperability of equipment, Madisetti. (1995).

2.5 LOSSLESS COMPRESSION ALGORITHMS

Compression is lossless, noiseless or invertible if the original file can be perfectly recovered from the compressed file. This is absolutely necessary for many types of data such as the program files, word processing files, tabulated numbers, etc. Lossless data compression algorithms falls into one of two categories: statistical method and dictionary based schemes. The main shortcoming of lossless compression is that the amount of compression is limited. Fig.2.4 shows the block diagram of a lossless compression system.

For maximum transmission efficiency, symbols must be represented with uniquely decipherable codes of minimum length. If the various messages to be transmitted do not have equal probabilities, compression can be achieved by allowing the code words to have unequal length. This is one of the basic principles of data compression.

2.4 DATA COMPRESSION ALGORITHMS.

Algorithm is defined according to the Macmillan Dictionary of Data Communications as a prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps. Data compression algorithms can be classified into two categories: lossless compression and lossy compression. These algorithms are based on ideas that have matured over the last 10-15 years in academia and related industry. In the recent years the emphasis in compression community has shifted from that of developing new techniques to that of software and hardware implementation of these techniques, and the development and establishment of standards to achieve interoperability of equipment, Madisetti, (1995).

2.5 LOSSLESS COMPRESSION ALGORITHMS

Compression is lossless, noiseless or invertible if the original file can be perfectly recovered from the compressed file. This is absolutely necessary for many types of data such as the program files, word processing files, tabulated numbers, etc. Lossless data compression algorithms falls into one of two categories: statistical method and dictionary based schemes. The main shortcoming of lossless compression is that the amount of compression is limited. Fig.2.4 shows the block diagram of a lossless compression system.

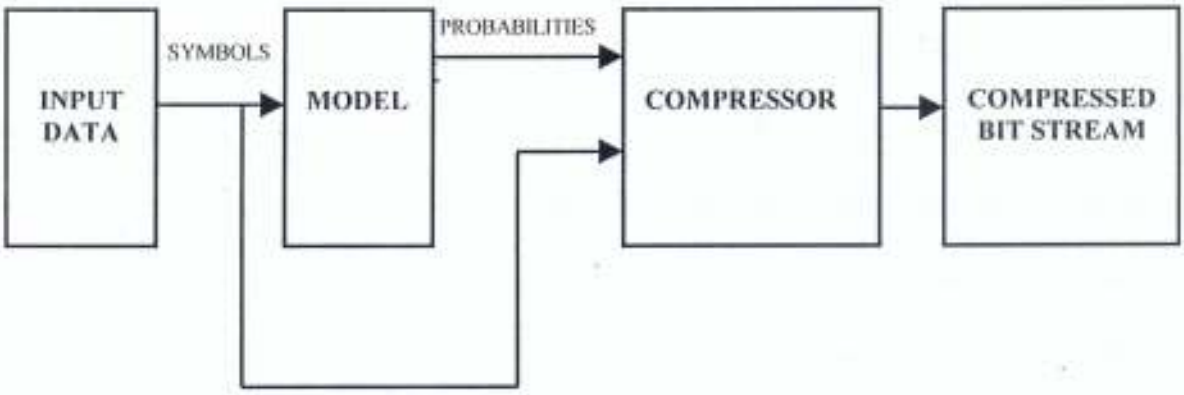

Fig. 2.4 Block Diagram of a Lossless Compression System.

2.6 STATISTICAL DATA COMPRESSION METHODS

Statistical methods of data compression operate by encoding symbols one at a time. The symbols are encoded into variable length output codes. The length of the output codes varies based on the probability or frequency of the symbol. Low probability symbols are encoded using many bits while high probability symbols are encoded using fewer bits. The Morse code of telegraphy provides a classic example by sending short patterns (dot and dashes) for frequent letters, and long pattern for rare letters, fewer bits are needed on the average than with the standard representation (ASCII), which assigns the same number of bits to all letters. Some famous examples are Shannon-Fanon code, Huffman code, arithmetic code, run length code and delta encoding.

2.6.1 STATISTICAL DATA COMPRESSION MODELS

Statistical data compression techniques are based on using appropriate model for the source (input) data in which defined elements (source values in general or within a particular context) are not all equally likely. The compressor and the decompressor must agree on identical model. The compressor determines the relevant elements under the agreed model and output sufficient information to enable the decompressor to determine these elements. A model needs to be able to do two things to effectively compress data:

(1) A model needs to accurately predict the frequency/ probability of symbols in the input data stream.

(2) The symbol probabilities generated by the model need to deviate from a uniform distribution.

The need to accurately predict the probability of symbols in the input data cannot be emphasized. The principal objective of statistical data compression is to reduce the number of bits needed to encode a character as its probabilities of appearance increases. For example, if letter 'e' represents 25% of the input data, it will only take two bits to

code. If letter 'z' represents only 0.1% of the input data, it might take 10 bits to code. If the mode is not generating probabilities accurately, it might take 10 bits to represent 'e' and 2 bits to represent 'z', causing data expansion instead of compression. The second condition is that the model needs to make predictions that deviate from a uniform distribution. The better the model is at making predictions that deviate from uniform, the better the compression ratio will be. For instance, a model can be created that assigns all 256 possible symbols a uniform probability of 1/256. This model will create an output file that is exactly the same as the input file, since every symbol will take exactly 8 bits to encode. Therefore compression can only be achieved by correctly finding probabilities that deviate from a normal distribution of the symbols.

2.6.2 CLASSIFICATION OF STATISTICAL MODELS

Statistical data compression models are classified as static or dynamic models.

2.6.2.1 STATIC MODEL

A static model is one in which the choice of elements and their assumed distribution are invariant. For example the letter 'e' might always assume to be the most likely character to occur. A static model can be predetermined with resulting unpredictable compression effect, or it can be built by the encoder by previewing the entire source and determining element frequencies This latter process suffers from the necessity of the encoder making two passes over the data, one pass for determining the frequencies and another for encoding, and also from the delay imposed on producing any output until after the entire source has been scanned. In addition, the relevance aspects of the determined frequency must be sent to the decoders, which would otherwise have no mean of determining it. The benefits of using a static model include its simplicity and the ability to decode without necessarily starting at the beginning of the compressed data.

2.6.2.2 DYNAMIC OR ADAPTIVE MODEL

In a dynamic model, both the compressor and the decompressor start with the same model. The compressor encodes a symbol using the existing model, then update the model to account for the new symbol. The decompressor likewise decodes a symbol using the existing model, then updates the model. As long as the algorithm to update the model operates identically for the compressor and the decompressor, the process can operate perfectly without the need to pass a statistics table from the compressor to the decompressor. Adaptive data compression has a slight disadvantage in that it starts compressing with less than optimal statistics. However, by subtracting the cost of transmitting the statistics with the compressed data, an adaptive model will usually performs better than a static model.

2.7 STATISTICAL DATA COMPRESSION ALGORITHMS

Examples of statistical data compression algorithms are discussed below.

2.7.1 SHANNON-FANO CODING

The Shannon-Fano algorithm is believed to have been developed independently by Shannon and Fano, in two separate publications both written in the same year, Shannon (1949); Fano, (1949). Shannon-Fano algorithm was the first to be available. Shannon-Fano coding is based on statistics obtained from the data to be coded. In other words, it takes into consideration the probability of each symbol's appearance. Using this information, a binary string is designated for each character. The code is constructed as follows: the source messages a(i) and their probabilities P(a(i)) are listed in order of decreasing probabilities. The list is then divided in such a way as to form two groups of nearly equal total probabilities. Each message in the first group receives 0 as the first digit of its coeword; the messages in the second half have codewords beginning with 1. Each of these groups is then divided according to the same criterion and additional code digits are appended. The process is continued until each subset contains only one

message. The above becomes clearer when the process is visualized as a binary tree. By doing so, a symbol's code string can be obtained by simply tracing the path to the leaf which represents it from the root of the tree. Table 2.1 shows the application of the method to a particular simple probability distribution while figure 2.5 shows the binary tree. The length of each codeword x is equal to –log P(x). This is true as long as it is possible to divide the list into subgroups of exact equal probability. When this is not possible, some codewords may be of length –log P(x) + 1 in which case the Shannon-Fano algorithm is not guaranteed to produce an optimal code.

Table 2.1 Example of Shannon- Fano Code

Symbol	Probability	Codeword
A	0.5	0
B	0.25	10
C	0.125	110
D	0.0625	1110
E	0.03125	11110
F	0.03125	11111

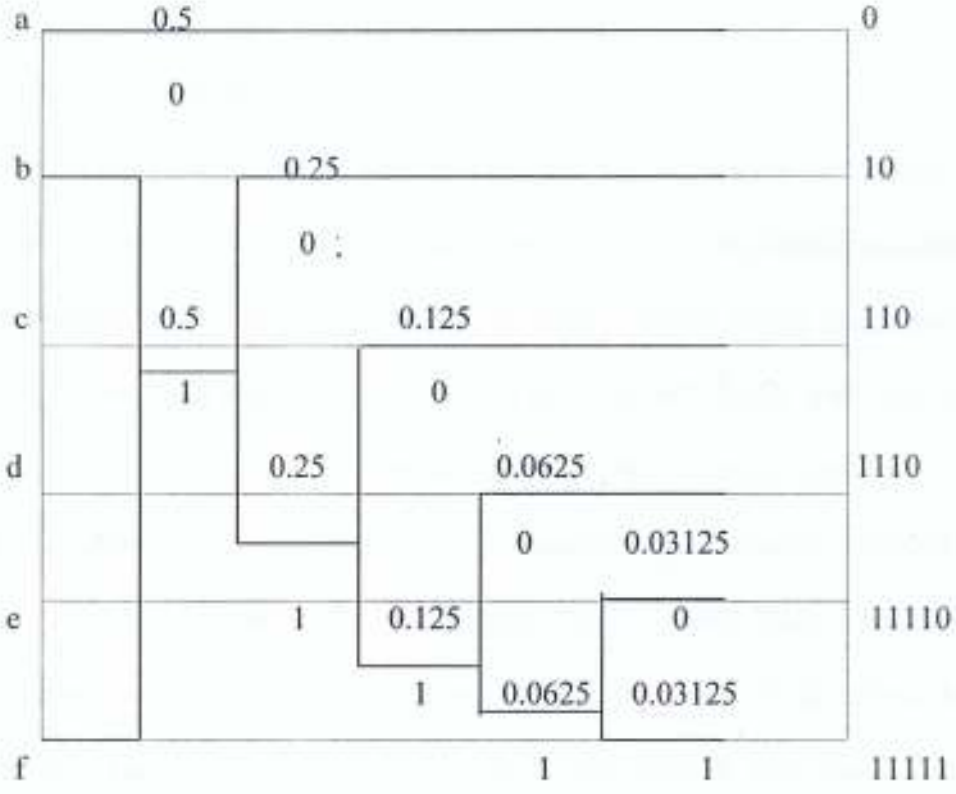

Fig. 2.5 Binary Tree of an Example of a Shannon-Fano Code

In the above figure, an upward traversal is set to be 0, and any downward traversal is set to be 1. To find the code of a symbol, one must simply trace the path to the leaf, which represents its probability. For example, the symbol "e" follows the path: *down, down, down, down, up.* Therefore, its code is 11110.

2.7.2 HUFFMAN CODING

Huffman published a paper in 1952 in which he described a method of creating a code table for a set of symbols with given probabilities. Huffman codes belong to a family of codes with variable length. That means that individual symbols, which make a message, are represented (encoded) with bit sequences that have distinct length. This characteristic of code words helps to decrease the amount of redundancy in source data. The Huffman code table was guaranteed to produce the lowest possible output bit count possible for the input stream of symbols when using fixed length source codes. Huffman called these codes "minimum redundancy codes", but the scheme is now universally called Huffman coding. Huffman coding assigns an output code to each symbol, with the output codes being as short as 1 bit, or considerably longer than the input symbols, strictly depending on their probabilities. The optimal number of bits to be used for each symbol is $\log_2(1/p)$, where p is the probability of a given character. Thus, if the probability of a character is 1/256, such as would be found in a random stream, the optimal number of bits per character is $\log_2(256)$, or 8. If the probability goes to ½, the optimum number of bits needed to code the character would go down to 1. The problem with this scheme is that Huffman codes have to be an integral number of bits long. For example, if the probability of a character is 1/3, the optimum number of bits to code that character is about 1.6. The Huffman coding scheme has to assign either 1 or 2 bits to the code, and either choice

leads to a longer compressed message that is theoretically possible. This non- optimal coding becomes a problem when the probability of a character becomes very high. If a statistical method is developed that assigns a 90% probability to a given character, the optimal code size is 0.15 bits. The Huffman coding system will probably assign a 1-bit code to the symbol, which is 6 times longer than necessary. Huffman Coding is either static or dynamic.

2.7.2.1 STATIC HUFFMAN CODING.

Static Huffman's algorithm takes as input a list of symbols, $S = [s_1, s_2, s_3, s_4, - - - -, s_n]$ with probabilities, $P = [p_1, p_2, p_3, p_4, - - - -, p_n]$ and then construct a full binary tree. The symbols are arranged in descending order of probabilities of occurrence. Initially there is a set of singleton trees, one for each probability in the set. At each level in the algorithm the trees corresponding to the smallest probabilities, pi and pj are merged into a new tree whose probability is pi + pj and whose root has two branches which are the sub trees represented by pi and pj. The pi and pj are removed from the list and pi + pj is inserted into the list. This process continues until the probabilities set contains a single value. If at any time, there are more than one way to choose a smallest pair of probabilities, any such pair may be chosen. The Huffman algorithm is demonstrated in table 2.3. The coding tree is then traced from a root (the generated symbol with probability 1) to origin symbols and 1 is assigned to each lower branch while 0 is assigned to each upper branch. Fig. 2.6 and Table 2.2 shows the Huffman tree and the code table respectively.

Table 2.2 Huffman Code Table

i	S_i	P_i	Code
1	S_1	0.25	01
2	S_2	0.20	11
3	S_3	0.15	001
4	S_4	0.12	100
5	S_5	0.10	101
6	S_6	0.10	0000
7	S_7	0.08	0001

Table. 2.3 Demonstration of Huffman Algorithm

S_1	0.25	0.25	0.25	0.33	0.42	0.58	1.0
S_2	0.20	0.20	0.22	0.25	0.33	0.42	
S_3	0.15	0.18	0.20	0.22	0.25		
S_4	0.12	0.15	0.18	0.20			
S_5	0.10	0.12	0.15				
S_6	0.10	0.10					
S_7	0.08						

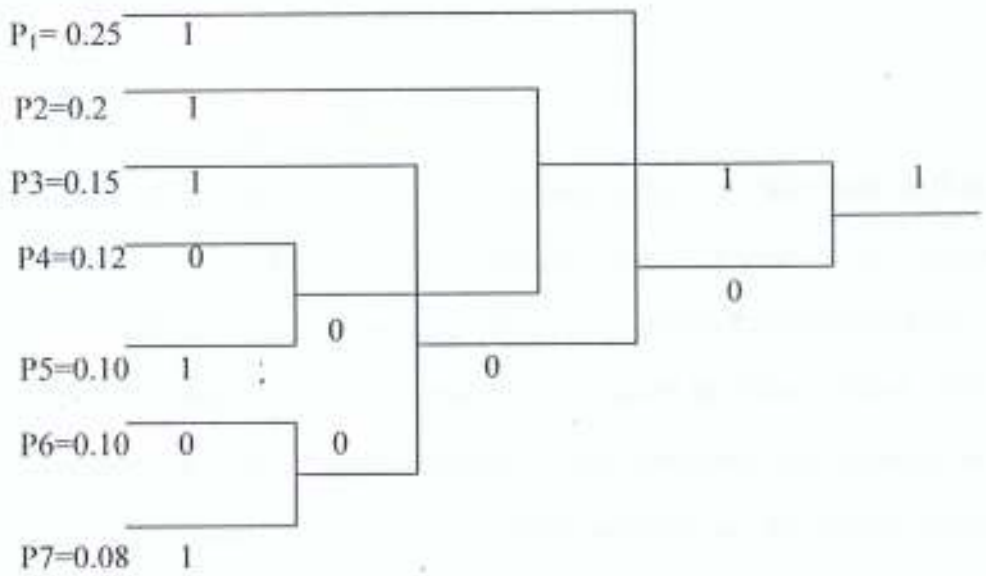

Fig. 2.6 Huffman Tree

Static Huffman algorithm has two major disadvantages, which are highlighted below.

(1) Static Huffman Coding requires two passes over the data.

(2) There is need to include the Huffman code table in the compressed file as side information for the decompression process.

To avoid this, a standard code table that is derived for the relevant class of data, could be used as a standard table, this is an option in the JPEG compression scheme. Another alternative is the adaptive Huffman coding.

2.7.2.2 ADAPTIVE HUFFMAN CODING.

Adaptive Huffman coding was first conceived independently by Faller and Gallager, Faller, (1973); Gallager, (1978). Knuth contributed improvements to the original algorithm Knuth, (1985) and the resulting algorithm is referred to as algorithm FGK. A more recent version of adaptive Huffman coding is described by Vitter, Vitter, (1987). All of these methods are defined-word schemes, which determine the mapping from source messages to codewords based upon a running estimate of the source message probabilities. The code is adaptive, changing so as to remain optimal for the current estimates. In this way, the adaptive Huffman codes respond to locality. In essence, the encoder is "Learning" the characteristics of the source. The decoder must learn along with the encoder by continually updating the Huffman tree so as to stay in synchronization with the decoder. Therefore there is no need to include the Huffman probability table in the compressed file.

Another advantage of these systems is that they require only one pass over the data. It is important to note that one-pass methods are not desirable if the number of bits they transmit is significantly greater than that of the two-pass scheme. Interestingly, the performance of adaptive Huffman method, in term of bit transmitted, can be better than that of static Huffman coding.

2.7.3 ARITHMETIC CODING

The method of arithmetic coding was suggested by, Elias and presented by Abramson in his text on Information Theory Abrahmson, (1963). Implementations of Elias, technique were developed by Rissane, Pasco, Rubin, and most recently Witten. The concept of arithmetic coding is presented below. In arithmetic coding a source of ensemble (stream of input symbols) is represented by an interval between o and 1 on the real number line. It completely bypasses the idea of replacing an input symbol with a specific code. As the interval becomes smaller the no of bits needed to specify it grows. Arithmetic coding assumes an explicit probabilistic model of the source. It is a defined word scheme, which uses the probabilities of the source to successfully narrow the interval used to represent the ensemble. A high probability symbol narrows the interval less than a low probability symbol, so that high probability symbols contribute fewer bits to the coded ensemble. The method begins with an unordered list of source symbol and their probabilities. The number line is partitioned into subintervals based on cumulative probabilities. The output from an arithmetic coding process is a single number less than 1 and greater than or equal to 0. This single number can uniquely decoded to create the exact stream of the symbols that went into its construction. The following example is used to illustrate the idea of arithmetic coding. Given an input string " FALOWO ADE" with the probabilities shown in Table 2.4.

Table 2.4 Symbols With Given Probabilities

Symbol	Probability	Cumulative Probability	Range
SPACE	1/10	0.1	0-0.1
A	2/10	0.3	0.1-0.3
D	1/10	0.4	0.3-0.4
E	1/10	0.5	0.4-0.5
F	1/10	0.6	0.5-0.6
L	1/10	0.7	0.6-0.7
O	2/10	0.9	0.7-0.9
W	1/10	1.0	0.9-1.0

Table 2.5 demonstrates the initial partitioning of the number line. For the ensemble FALOWO ADE, the first symbol, 'F' reduces the interval to (0.5 – 0.6) and the second symbol, 'A' to (0.51 – 0.53), (the first 2/10 of the previous interval). The 'L' further narrows the interval to (0.522 – 0.524) (1/10 of the previous interval), the 'O' to (0.5234 – 0.5238) (2/10 of the previous interval), the 'W' to (0.52376 – 0.52380) (1/10 of the previous interval). The 'O' to (0.3788 – 0.523795) (2/10 of the previous interval), the 'space' to (0.52378808 – 0.52378824) (2/10 of the previous interval), the 'D' to (0.523788128 – 0.523788144) (1/10 of the previous interval) and the 'E' yield a final interval (0.5237881344 – 0.5237881360) (1/10 of the previous interval). The interval, of alternatively any number, n within the interval, may now be used to represent the source ensemble.

Table 2.5 Arithmetic Coding Process

Message	Left Endpoint of the Interval	Right Endpoint of the Interval
F	0.5	0.6
A	0.51	0.53
L	0.522	0.524
O	0.5234	0.5238
W	0.52376	0.52380
O	0.523788	0.523796
SPACE	0.5237880	0.5237888
A	0.52378808	0.52378824
D	0.523788128	0.523788144
E	0.5237881344	0.5237881360

Three equations may be used to define the narrowing process described in Table 2.5:

Newleft = Prevleft + PRCLM × prevsize 2.6

Newsize = Prevsize × PRCM 2.7

Newright = Newleft + Newsize 2.8

Where

Newleft = The left endpoint of the new interval

Prevleft = The left endpoint of the previous interval

Presize = The previous interval

PRCML = The left endpoint of the cumulative Probability range of the current symbol.

Newright = The right endpoint of the new interval

Newsize = The new interval

PRCM = The probability of current message

The size of the final subinterval determines the number of bits needed to specify a number in that range. The number of bits needed to specify a subinterval of (0,1) of size, is $\log_2 S$. Since the size of the final subinterval is the product of the probabilities of the source symbols in the ensemble, S is given by:

$$S = \prod_{i=1}^{N} P \text{ (source symbol i)} \quad \text{.....2.4}$$

where N = is the length of the ensemble

$$Log\ S = \sum_{i=1}^{N} P\ (a\ (i))\ \log\ (a\ (i)) \quad \text{.....2.5}$$

where N is the number of unique source messages a (1), a(2), ...a(n)

Thus, the number of bits generated by the arithmetic coding technique is exactly equal to entropy, H. This demonstrates the fact that arithmetic coding achieves compression, which is almost exactly that, predicted by the entropy of the source.

2.7.3.1 ARITHEMETIC DECODING

In order to recover the original ensemble, the decoder must know the model of the source used by the encoder (e.g. source symbols and associated ranges) and single number, n within the interval determined by the encoder. Decoding consists of a series of comparisons of the number, n to the ranges representing the source symbols. This is shown in table 2.6. The algorithm below is used to define the decoding process.

Get the encoded number

Do

Find the symbol whose range straddles the encoded number

Output the symbol

Range = symbol low value - symbol high value.

Subtract symbol low value from encoded number

Divide encodednumber by range

Until no more symbol.

Table 2.6 Arithmetic Decoding Process for the "FALOWO ADE" Message.

Encoded Number	Output Symbol	Low	High	Range
0.5237881344	F	0.5	0.6	0.1
0.237881344	A	0.1	0.3	0.2
0.68940672	L	0.6	0.7	0.1
0.8940672	O	0.7	0.9	0.2
0.970336	W	0.9	1.0	0.1
0.70336	O	0.7	0.9	0.2
0.0168	SPACE	0.0	0.1	0.1
1.68	A	0.1	0.3	0.2
0.34	D	0.3	0.4	0.1
0.4	E	0.4	0.5	0.1

2.7.3.2 ARITHMETIC CODING IMPLEMENTATION PROBLEMS

Several difficulties become evident when implementation of arithmetic coding is attempted. The first is that the decoder needs some way of knowing when to stop. As evidence of this, the number 'O' could represent any of the source ensembles space, space space, space space space etc. Two solutions to this problem have been suggested. One is that the encoder transmit the size of the ensemble as part of the description of the model. Another is that a special symbol be included in the model for the purpose of signaling and of message. The second alternative is preferable for several reasons. First, sending the size of the ensemble requires a two-pass process and precludes arithmetic coding as part of a hybrid codebook scheme (i.e. a scheme where the sender and receiver maintain identical codebooks containing K static codes. For each transmission, the

sender must choose one of the K previously agree-upon codes and inform the receiver of his choice). Secondly, adaptive methods of arithmetic coding can easily been developed and a first pass to determine the ensemble size is inappropriate in an on-line adaptive scheme.

A second issue left unresolved by the fundamental concept of arithmetic coding is that of incremental transmission and reception. It appears from the above discussion that the encoding algorithm transmits nothing until the final interval is determined. However, this delay is not necessary. As the interval grows, the leading bits of the left and right endpoints become the same. Any leading bits that are the same may be transmitted immediately, as this will not affect further narrowing.

A third issue is that of precision. From the description of arithmetic coding it appears that the precision required grows without bound as the length of the ensemble grows. Witten et al and Rubin address this issue Witten et al, (1987) ; Rubin, (1979). Fixed precision registers may be used as long as underflow and overflow are detected and managed. The degree of compression achieved by an implementation of arithmetic coding is not exactly equal to entropy, as implied by he concept of arithmetic coding. Both the use of a message terminator and the use of fixed-length arithmetic reduce coding effectiveness. However, it is clear that an end-of-message symbol will not have a significant effect on a large source ensemble. Witten approximated the overhead due to the use of fixed precision at 10^{-4} bits per source message, which is also negligible. Lastly, arithmetic coding consume large amount of computing resources, both in terms and CPU power and memory.

2.7.4 RUN-LENGTH CODING

In any ordered sequence of elements of different kinds, each maximum subsequence of elements of like kind is called a run. For example, the sequence α α α β β A A ..... opens with an alpha (α) run of length 3 followed by a betta (β) run of length 2. Run length coding was popularized by Golomb in early 1960s. Data files frequently contain the same character repeated many times in a row. For example, text files use multiple spaces to separate sentences, indent paragraphs, format tables, and charts etc. Likewise, digitized music might have long runs of zeros between songs format. Image and video signals also contain runs of the same value. For example, a standard TV picture contains a sequence of adjacent pixel with the same luminance. That is, as we trace across a horizontal line, we may encounter as many as several hundred pixels with the same brightness. In such cases we can use run-length coding to reduce the number of bits required to send the signal. Instead of sending the luminance of every single pixel, we send the starting position and luminance of the first of a number of pixels with the same brightness.

Fig. 2.4 illustrates run-length encoding for a data sequence having frequent runs of zero. Each time a zero is encountered in the input data, two values are written to the output file. The first of these values is a zero, a flag to indicate that run-length compression is beginning. The second value is the number of zeros in the run. If the average run-length is longer than two, compression will take place. On the other hand, many single zeros in the data can make the encoder file longer than the original file (expansion). Fig. 2.7 illustrates run-length coding. Each run of zero is replace by two characters in the

compressed file. A zero to indicate that compression is occurring followed by the number of zeros in the run.

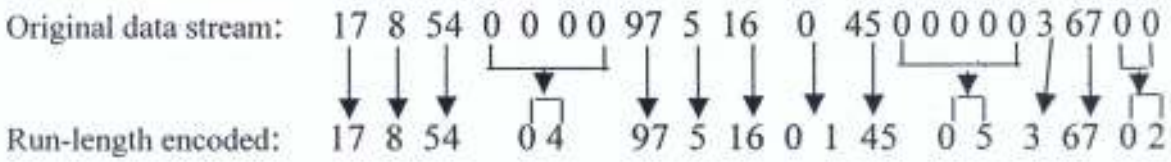

Fig. 2.7 Example of Run Length Coding

Many different run-length schemes have been developed. For example, the input data can be treated as individual bytes, or group of bytes that represent something more elaborate, such as floating point numbers. Run-length encoding can be used on only one of the characters (as with the zero in fig. 2.7), several of the characters, or all of the characters. A good example of a generalized run-length scheme is pack bits algorithm in which each byte (eight bits) from the input file is replaced by nine bits in the compressed file. The added ninth bit is interpreted as the sign of the number. That is, each character read from the input file is between 0 and 255, while each character written to the encoded file is between –255 and 255. To understand how this is used, consider the input file: 1, 2, 3, 4, 2, 2, 2, 2, 4, and the compressed file generated by pack bits algorithm: 1, 2, 3, 4, 2, -3, 4. The compression program simply transfers each number from the input file to the compressed file, with the exception of the run: 2, 2, 2, 2. This is represented in the compressed file by the two numbers: 2, -3. The first number '2' indicates what character the run consists of. The second number '-3' indicates the number of character in the run, found by taking the absolute value and adding one. For example 4, -2 means 4, 4, 4 while 21, -4 means 21, 21, 21, 21, 21 etc.

One problem with pack bits is that the nine bits must be reformatted into the standard eight bit bytes used in computer storage old transmission. A useful modification to this scheme can be made when the input is restricted to be ASCII text. Each ASCII character is usually stored as a full byte (eight bits), but really only uses seven of the bits of identify the character. In other words, the value 127 through 255 are not defined with any standardized meaning, and do not need to be stored or transmitted. This allows the eight bit to indicate if run-length encoding is in program.

2.7.5 DELTA ENCODING

In science, engineering and mathematics, the Greek letter delta (δ) is used to denote the change in a variable. The term delta encoding, refers to the technique that store data as the difference between successive samples (or character), rather than directly storing the samples themselves. Figure 2.7 shows an example of delta encoding. The value in the delta-encoded file is the same as the first value in the original data. All the following Values are equal to the difference (delta) between the corresponding value in the input file, and the previous value in the input file. Delta encoding can be used for data compression when the values in the original data are smooth, that is, there is typically only a small change in between adjacent values. This is not the case for ASCII text and executable code; however, it is very common when the file represents a digitized signal. Fig. 2.8(a) shows a segment of an audio signal, digitized to 8 bits, with each samples between −127 and 127. Fig. 2.8(b) shows the delta-encoded version of the digitized signal. The key feature is that the delta-encoded signal has lower amplitude than the original signal

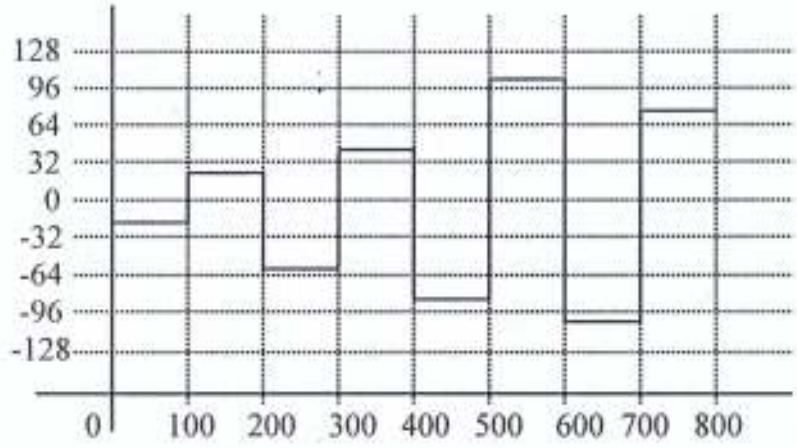

Fig. 2.8 (a) Audio Signal Digitized To Eight Bits

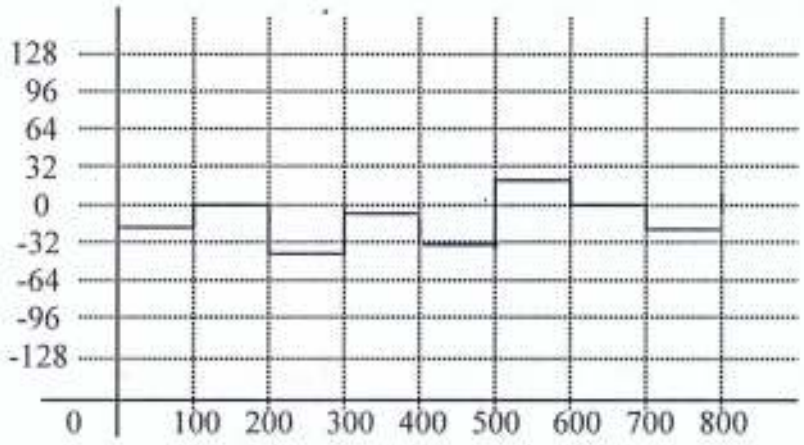

Fig. 2.8 (b) Delta Encoded Signal

2.8 DICTIONARY BASED SCHEME

The basic idea behind a dictionary-based scheme is to replace an occurrence of a particular phrase or group of bytes in the input text with a reference to a previous occurrence of that phrase. Examples of dictionary-based schemes are LZ77, LZ78 and LZW, Welch, methods. LZW dynamic dictionary scheme is used in this research work because it is very efficient for text compression. The dictionary is dynamic in the sense that every new string of characters it comes across is added to it until it gets to the maximum size during the compression and decompression processes.

2.8.1 THE LZ 77 COMPRESSION ALGORITHM

LZ 77 compression algorithm uses a previously seen text as a dictionary. The main structure in the original LZ 77 is a two-part sliding window. The larger part of the window is text that is already coded, while the smaller part, called the look-ahead buffer, contains text that is to be coded. Incoming text is coded by tuples of the form (index, length, successor symbol). Index points to a location within the window on which a match with some of the to-be-encoded text is found. The number of matching symbols is given by length, and successor symbol is the first symbol in the look-ahead buffer that doesn't match.

The algorithm is described below.

1. Fill the look-ahead buffer with symbols from the input stream.
2. Find the longest match between the (start of the) look-ahead buffer and the already seen text.
3. Output a tuple as described above to the output stream.
4. Shift the contents of the window length + 1 symbols to the left, and fill empty bytes in the look-ahead buffer from the input stream.
5. Go to step 2 if the look-ahead buffer is not empty.

Figure 2.9 illustrates the LZ 77 compressor. The figure shows the contents of the window during the last phase of the coding, which is "abracadabra".

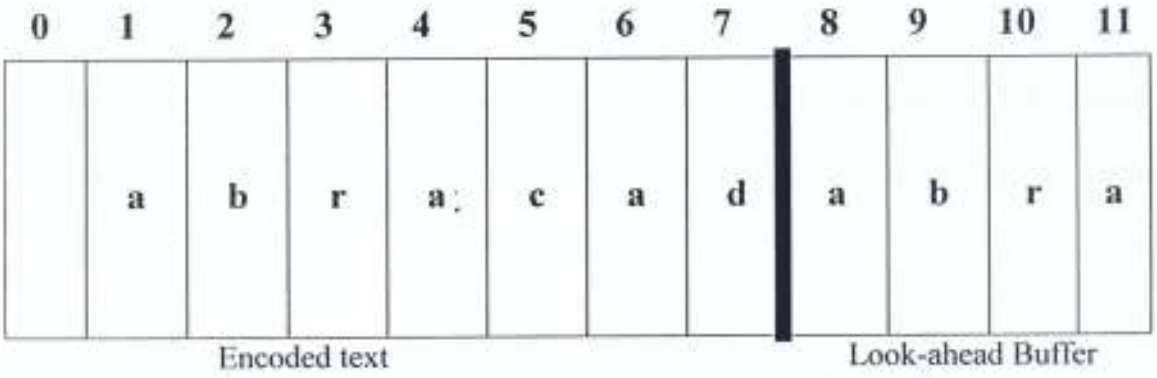

Fig. 2.9 LZ Coding Proccess

The first seven symbols are already coded and sent to the output stream, while the last four characters remain to be coded. By inspection, we see that the entire look-ahead buffer match text that is already seen. The final step of the coding thus yields the tuple (1, 4, EOF), where EOF is a special symbol indicating the end of the stream. Since the window is 12 bytes long, we need four bits to encode an index (leaving four possible indexes unused). The look-ahead buffer is four bytes long, so we need two bits to represent the length if we allow zero length to be encoded using one of the unused index codes. The input alphabet contains five symbols and the special EOF symbols, so three bits will suffice for this. Adding it all up, each tuple occupies nine bits. When coding, the first three symbols each yield one tuple, while the next two pairs each give one tuple since a is found in the window. The four last symbols yield one tuple, giving a total of six tuples, and 54 bits. Decompression is simple and fast: whenever on (index, length, successor symbol) is encountered, go the that (index) in the window and copy (length) bytes to the output then add the (successor symbol).

2.8.2 LZ 78 COMPRESSION ALGORITHM

LZ 78 – based schemes work by entering phrases into a dictionary and then, when a repeated occurrence of that particular phrase is found, outputting the dictionary index instead of the phrase. There exist several compression algorithm based on this principle, differing mainly in the manner in which they manage the dictionary. The most well-know scheme (in fact the most well-known of all the Lempel-Ziv compressors), is Lempel-Ziv-Welch scheme. A sample run of LZ 78 compressor is shown table 2.5.

Table 2.7 LZ 78 Coding Process

Input String: /WED/WE/WEE/WEB		
Character Input	Code Output	New code value
/W	/	256 = /W
E	W	257 = WE
D	E	258 = ED
/	D	259 = D/
WE	256	260 = /WE
/	E	261 – E/
WEE	260	262 = /WEE
/W	261	263 = E/W
EB	257	264 = WEB
END	B	

LZ78 starts with a dictionary, of which entries 0 – 255 refer to individual bytes, the remaining entries refer to substrings. Each time a new code is generated it means a new string has been pursed. New strings are generated by appending the current character K to the end of an existing string W.

2.9 LOSSY COMPRESSION ALGORITHMS

Compression is lossy or non invertible if the original file cannot be perfectly recovered from the compressed file. The goal is to get the best possible quality from the given compression. Examples of lossy compression algorithms are predictive coding, transform coding and vector quantization. Lossy compression algorithms are important for transmission of voice and image. Fig 2.10 shows the block diagram of a lossy compression system.

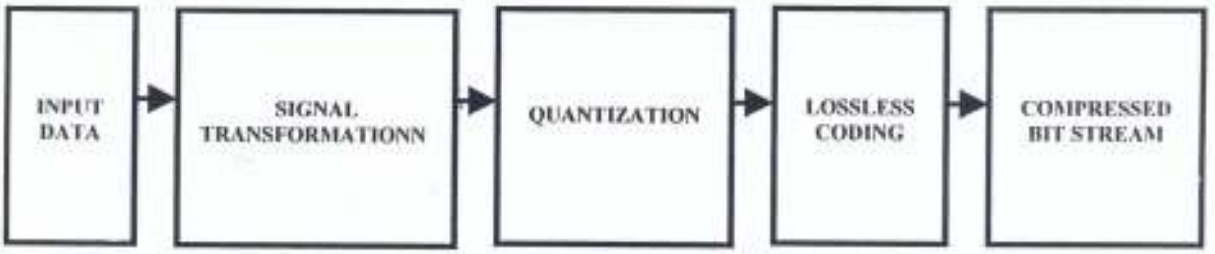

Fig. 2.10 Block Diagram of a General Lossy Compression System.

2.9.1 PREDICTIVE CODING

The basis of predictive coding is to reduce the number of bits used to represent information by taking advantage of correlation in the input data, which can be scalar, vector, or even block samples. If the source data have a high correlation between adjacent samples, the variance of the difference between adjacent samples is smaller than the variance of the original data. If this difference is coded rather than the original data, fewer bits are needed for the same desired accuracy. There are many types of predictive coding systems. The most popular one is the linear predictive coding system based on the following linear relationship:

$$X'[k] = \sum_{i=0}^{k-1} \alpha X[i] \quad \text{.....2.3}$$

Where the X[i] are the input data, the α_i are the prediction coefficients, and X'[k] is the predicted value of X[k]. The difference between the predicted value and actual value, e[k], is called the prediction error.

$$e[k] = X[k] - X'[k] \quad \text{.....2.4}$$

It is found that the prediction error usually has much lower variance than the original signal, and is significantly less correlated. It has a stable histogram that can be approximated by a Laplacian distribution, Habibi, (1971). With linear predictive coding, one can achieve a much higher SNR at a given bit rate. There are three basic components in the predictive coding encoder. They are predictor, quantizer, and coder, as illustrated in figure 2.11. As shown in the figure, the predicted signal, X[k], is subtracted from the input data, X[k]. The result of the is the prediction error, e[k], according to equation (2.4). The prediction error is quantized coded, and sent through communication channel to the decoder. In the mean time the predicted data is added back to quantized prediction error, e_q [k], to create reconstructed data, x. It should be noticed that the predictor makes the prediction according to equation (2.3), with previously reconstructed data, x's.

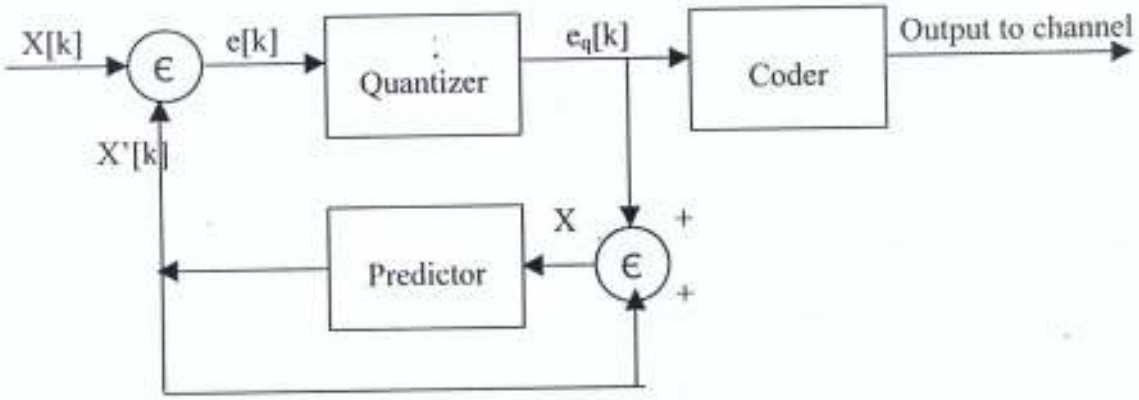

Fig. 2.11 Block Diagram of a Predictive Coder

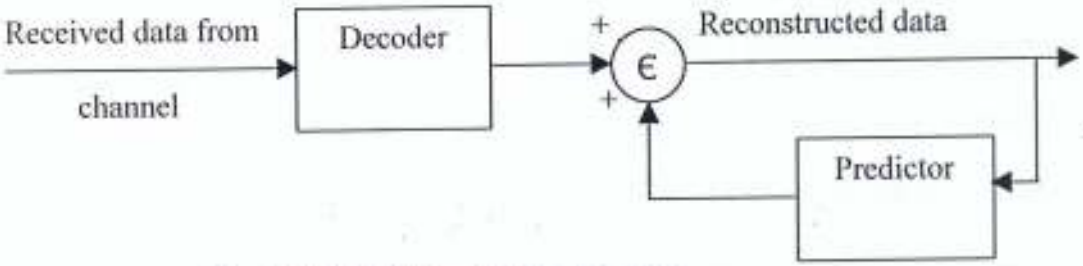

Fig. 2.12 Predictive Coding Decoder

Just like the encoder, the predictive coding decoder has a predictor, as shown in figure 2.12, which also operates in the same way as the one in the encoder. After receiving the prediction error from the encoder, the decoder first decodes the received data. Then the predicted signal is added back to recreate the reconstructed data. Even though linear prediction coding is the most popular predictive coding system, there are many variations. If the prediction coefficients remain fixed, then it is called global prediction. If the prediction coefficients change on each frame basis, then it is called local prediction. If they change adaptively, then it is called adaptive prediction. The main criterion of a good linear prediction coding is to have a set of prediction coefficients that minimize the mean square prediction error. Linear predictive coding (LPC) is widely used in both audio and video compression applications. The most popular linear predictive codings are differential pulse code modulation (DPCM) and the adaptive differential pulse code modulation (ADPCM). In digital video where source data exhibit high correlation both between pixels within a frame (spatial correlation) and between pixels in differing frames (temporal correlation). Video compression falls into two main types: (1) interframe predictions, which uses a combination of key motion – predicted and interpolated frames to achieve high – compression ratio; (2) intraframe coding, which compresses every frame of video individually. Interframe prediction technique takes advantage of the temporal correlation, while the spatial correlation are exploited by intraframe coding method.

Motion compensation (MC), one of the most complex prediction methods, reduces the prediction error, by predicting the motion of the image objects. The basic idea of MC prediction arises from common sense observation: in a video sequence, successive frames (or field) are likely to represent the same details, with little difference between one frame

and the next. A sequence showing moving objects over a still background is a good example. Data compression can be effected if each component of a frame is represented by its difference with the most similar component – the predictor in the previous frame, and by a vector - the motion vector – expressing the relative position of the two components. If an actual motion exists between the two frames, the difference may be null or very small. The original component can be reconstructed from the difference the motion vector, and the previous frame. It has long been recognized that transmission errors can seriously degrade any data that have be compressed whether using lossless or lossy methods, Solomon, (1998). In prediction - based coding, the influence of any errors during data transmission affects all subsequence data. In particular, when interframe prediction is used, the influence of transmission error is quite noticeable. Since predictive coding schemes are often used in combination with other schemes, such as transform – based scheme, the influence of transmission error must be given due consideration.

2.9.2 TRANSFORM CODING

When the frequency distribution of signals containing strong correlation is considered, it appears that the signal power is concentrated in the low – frequency region. Thus the low frequency components of a signal are more important than the high frequency components. It is possible to exploit for compression any system bias in components of the signal. The key idea behind transform coding is to transform the original signal in such a way as to emphasize the bias, making it more amenable to techniques that remove redundancy.

Consider a reversible orthogonal transform, T, that transforms a block of data to a transform domain with the transform pair as :

$$y = T(x) \quad \text{.....(2.5)}$$

$$x = T^{-1}(y) \quad \text{.....(2.6)}$$

Where x is the original data block and T^{-1} is the inverse transform of T. In the transform domain the component, y is referred to as the transform coefficients. If transform t has the characteristics that most of the transform coefficients are very small. Then the insignificant transform coefficients need not to be transmitted to decoder and can be eliminated during the quatization stage. As a result very good compression can be achieved with the transform coding approach. Figure 2.13 shows a typical lossy transform coding data compression system. In the figure the input data block, x, passes through the forward transform, T, with transform coefficients, y, as its output. T has the characteristics that most of its output, y, are small and insignificant and that there is little statistical correlation amoung the transform coefficients, which usually results in efficient compression by simple algorithms. The transform coefficients, y are quantized by quantizer, Q. Small and insignificant coefficients have a zero quantized value; therefore only few nonzero coefficients need to be coded and transmitted to the decoder. After receiving the signal from the network, the decoder decodes and inverse quantizes the received signal and reconstructs the transform coefficients, y'. The reconstructed transform coefficients passes through the inverse transform, T^{-1}, which generates the constructed signal, x'.

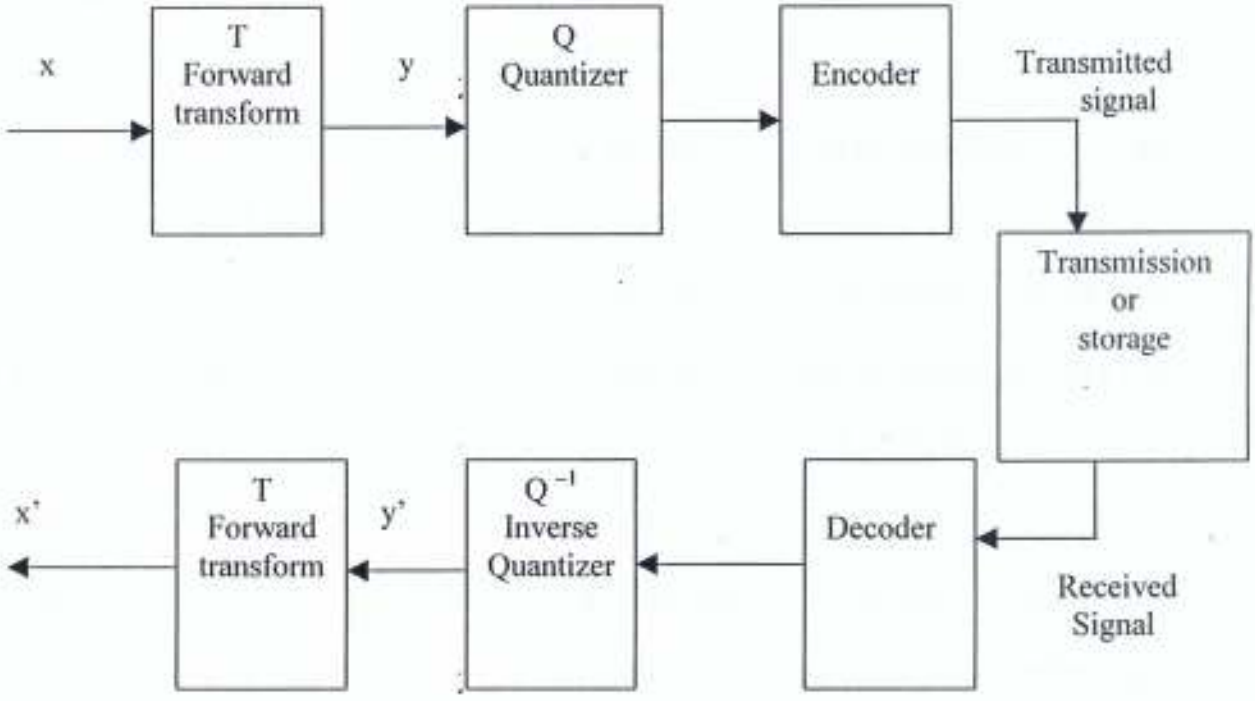

Fig. 2.13 Transform Coding Data Compression System

In general, transform coding takes advantage of the linear dependency of adjacent input samples. The linear transform actually converts the input samples to the transform domain for efficient quantization. In quantization stage the transform coefficients can be quantized with a scalar quantizer or a vector quantizer. However bit allocation amoung transform coefficients is crucial to the performance of the transform coding. A proper bit allocation at the quantization stage can achieve the output with a good fidelity as well as a good compression ratio. There are quite a few transform-coding techniques. Each has its characteristics and applications. The discrete Fourier transform (DFT) is popular and is commonly used for spectral analysis ans filtering. Fast implementation of the DFT, also known as fast Fourier transform (FFT), reduces the transform operation to n(n log2 n) for an n-point transform. The Karhunen-Loeve transform (KLT) is an optimal transform in the sense that its coefficients contain a larger fraction of the total energy compared to any other transform. The KL transform can completely remove the statistical correlation of image data and provide a minimum mean-square-error, Jain, (1989). There is no fast implementation of the KLT therefore it is not widely used. Another orthogonal transform is the discrete cosine transform (DCT). DCT is widely used for video compression, because the power of the transform signal is highly concentrated in the low frequencies, and can be computed rapidly.

2.8.3 VECTOR QUATIZATION

As opposed to scalar quantization, in which samples are independently quantized one at a time, vector quantization (VQ) attempts to remove redundancy between sample value by collecting several sample values and quantizing them as a single vector. Since the input to a scalar quantizer consists of individual sample values, the signal space is a finite interval of the real number line. This interval is divided into several regions, and each region is represented in the quantized outputs by a single value. The input to a vector

quantizer is typically an n-dimensional vector, and the signal space is likewise an n-dimetional space. For simplicity, consider the case where n = 2. In this case, the input to the quantizer is the vector X_j, which corresponds to the pair of sample (S^1_j, S^2_j). To perform vector quantization, the signal space is divided into a finite number of overlapping regions, and a single vector to represent each region is determined. When the vector X_j is input, the region containing X_j is determined, and the representative vector for that region, Y_i is output. This concept is shown in figure 2.14. The encoder determines the region to which the input X_j belongs and outputs j, the index value, which represents the region. The decoder receives this value, j extracts the corresponding vector Y_j from the representative vector set, and outputs it. The set of representative vector is called the codebook. To design a high performance vector quantizer, the representative vectors and the regions they cover must be chosen to minimize total distortion. If the input vector probability density function is known in advance, and the vector dimensionality is low, it is possible to perform an exact optimization. However, in an actual application it is rare for the input vector probability density to be known in advance. An algorithm called the LBG algorithm is widely used for adaptively designing vector quantizers in this situation, Gersho, (1992). LBG is a practical algorithm that starts out with some reasonable codebook and, by adaptively interacting the determination of regions and representative vectors, converges on a better codebook.

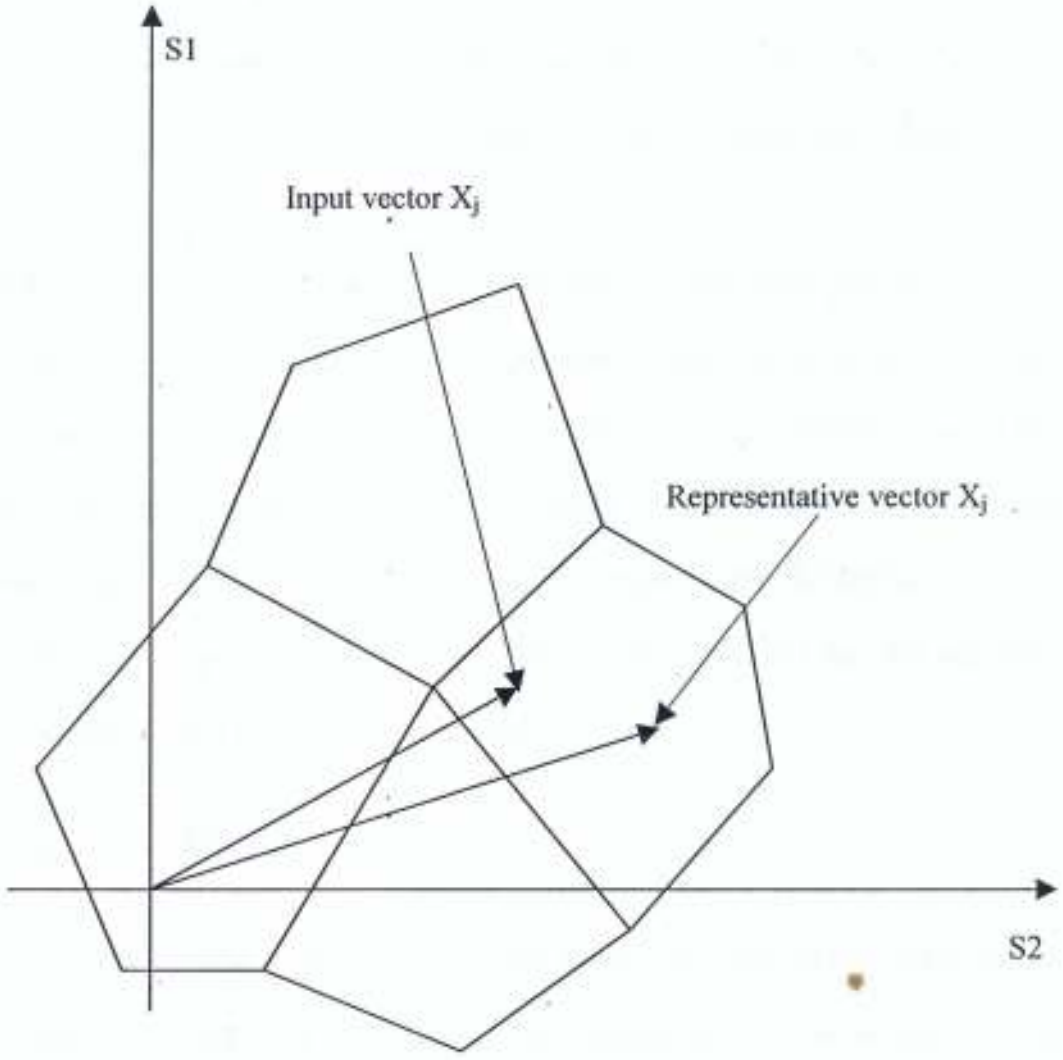

Fig. 2.14 Example of VQ With Two- Dimensional Vector

2.10 DATA COMPRESSION STANDARDS

Standardization is very important in the development of common compression methods to be used in the new services and products that are now possible. This allows the new services to interoperate with each other and encourages the investment needed in integrated circuits to make technology cheap. In the framework of International Standard

Organization and International Electro-Technical Commission (ISO/IEC), subgroups were established in May 1988: the Joint Photographic Experts Group (JPEG) is working on coding algorithms for still images; the Joint Bilevel Image Expects Group (JBIG) is working on the progressive processing of bilevel coding algorithms, and the Moving Picture Experts Group (MPEG) is working on representation of motion video. In the International Telecommunication Union (ITU), H.261 and H.263 are also developed for video conference and telephone applications.

2.10.1 JPEG (or JFIF)

JPEG is a compression standard aimed at still images, in colour or grey scale, and having some degree of complexity (for example, photographs of the 'real' world). For moving images, MPEG is more appropriate, and for simple images (highly geometric or stylised) GIF is more appropriate. Public domain viewers are available for all major platforms. JFIF (JPEG File Interchange Format) is the current de facto interchange format for JPEG compressed images. JPEG is a lossy compression scheme. The greater the compression, the greater the degree of information loss. This can be user determined, to optimise the trade-off between resultant image size and image quality. The algorithm exploits some of the ways in which the human eye perceives and analyses images, so that compressed images still appear to be of high quality when looked at by human eyes. Times for

compression/decompression are symmetric - they take roughly the same amount of time. It provides fast compression speed compared to fractal compression.

The disadvantages of JPEG are:

JPEG compression is a trade-off between degree of compression, resultant image quality and time required for compression/decompression. Blockiness results at high image compression ratios. It produces poor image quality when compressing text or images containing sharp edges or lines. Gibb's effect is the name given to this phenomenon where disturbances/ripples may be seen at the margins of objects with sharp borders. It is not suitable for 2 bit black and white images. The degree of compression is greater for full colour images than it is for grey scale images. It is not suitable as a strategy for images that are still being edited, because every compression/decompression cycle continues to lose information. It is not intended for moving images/video. It is not resolution independent. Does not provide for scalability, where the image is displayed optimally depending on the resolution of the viewing device.

2.10.2 GIF

This is not normally looked upon as a lossy form of compression, but GIF is nominally a lossless compression scheme; for grayscale images, it truly is lossless. Colour, however, is a different story. GIF works only on indexed colour images, and a huge amount of information is lost when you convert a 24-bit colour image to 8-bit indexed colour - you go from a possible 16.7 million colours to a mere 256. Images destined for the Web never contain anything close to 16.7 million colours, for the simple reason that they never contain anywhere near that number of pixels. But even a small, 320-by-240-pixel image

can contain 300 times more colours than indexed colour can represent, which can result in an 8-bit or 5-bit GIF that looks grainy.

2.10.3 MPEG

The MPEG Standards for Audio and Video Compression Motion Picture Experts Group is a joint committee of the International Standards Organization (ISO) who generate standards for digital video and audio compression. MPEG defines a compressed bit stream, which implicitly defines a decompressor. The compression algorithms are up to the encoder (usually a manufacturer) allowing further enhancement. MPEG coding uses block-based motion compensation in interframe coding and block-based Discrete. MPEG1 is the name given to the original standard generated by MPEG. It was mainly addressing the issue of standardizing compressed audio for CD storage systems. It defines a bit stream for compressed video (SIF - 352x240) and audio optimized to fit into a bandwidth (data rate) of 1.5Mb/s. The complexity of the encoder is much higher and its speed is not critical since CD's are ROMs, i.e., write once read many times. This allows expensive encoder systems and encoding time is only critical as far as manufacturing efficiency is concerned and the end user suffers no loss in reproduced sound quality. Hardware is now available, be it expensive, to compress video sequences in real-time into MPEG1 compress image sequences.

2.11 MEASURE OF COMPRESSION

In order to discuss the relative merits of data compression techniques, a framework for comparison must be established. There are two dimensions along which each of the schemes discussed here may be measured, algorithm complexity and amount of compression. When data compression is used in a data transmission application, the goal

is speed. Speed of transmission depends upon the number of bits sent, the time required for the encoder to generate the coded message, and the time required for the decoder to recover the original ensemble.

In data storage application, although the degree of compression is the primary concern, it is nonetheless necessary that the algorithm be efficient in order for the scheme to be practical. For static scheme, there are three algorithms to analyze: the model construction algorithm, the encoding algorithm, and the decoding algorithm. For a dynamic scheme, there are just two algorithms: the encoding algorithm, and the decoding algorithm. Several common measures of compression have been suggested: redundancy, Shannon and weaver, (1949), average message (codeword) length, Huffman, (1952), and compression ratio, Rubin, (1976). These measures are discussed bellow.

2.11.1 REDUNDANCY

Redundancy, R can be defined as;

$$R = \sum_{i=1}^{n} P[X(i)]l(i) - \sum_{i=1}^{n} - P[X(i)]\log_2 P[X(i)] \quad \text{.....2.7}$$

Where l(i) is the length of the codeword representing message x(i) of probability P[x(i)]. The expression $\sum_{i=1}^{n} P[X(i)]l(i)$ represents the length of the codewords weighted by their probabilities of occurrence, that is, the average codeword length. The expression $\sum_{i=1}^{n} - P[X(i)]\log_2 P[X(i)]$ is entropy, H. Thus redundancy is a measure of the difference between the average codeword length and average information content. If a code has minimum average codeword length for a given discrete probability distribution, it is said to be a minimum redundancy code. Redundancy is measured in bits/ symbols.

2.11.2 AVERAGE CODEWORD LENGTH

Huffman uses the average length of a coded message, $\sum_{i=1}^{n} P[X(i)]l(i)$, as a measured of a code. Since redundancy is defined to be average codeword length minus entropy and entropy is constant for a given probability distribution, minimum average codeword length minimizes redundancy. The average codeword length is measured in bits/ symbols.

2.11.3 COMPRESSION RATIO

The amount of compression yielded by a coding scheme can be measured by the compression ratio. The term compression ratio has been defined in several ways.

The definition, compression ratio = $\left(\frac{compressedlength}{originallength}\right) \times 8$ in bits per bytes (bpb) is one of the common ways of expressing compression ratio. If a 400 bytes file is compressed down to 100 bytes, the compression ratio is 2bpb. Compression ratio can also be expressed in percentage. 25% compression ratio implies that the compressed file is 25% of the original file. Similarly, compression ratio 2:1 means that the file has been compressed to the half.

CHAPTER THREE

RESEARCH METHODOLOGY

Lempel-Ziv-Welch (LZW) algorithm is used in this research work to develop a computer application program for data compression. LZW compression algorithm was named after its developers: Lempel, Ziv, and Welch. Original Lempel approach to data compression was first published in 1977 with later modification by Welch (1984). LZW is a way of compressing data that takes advantage of repetition of strings in the input data. It replaces strings of characters with single codes. It does not do any analysis on the incoming text. Instead it just adds every new string of characters it sees to a table of strings (dictionary). Compression occurs when a string of character is replaced by a single code.

3.1 LZW COMPRESSION ALGORITHM

The LZW compression algorithm always output codes for strings that are already known and each time a new code is output, a new string is added to the dictionary. The compression algorithm in its simplest form is shown below.

1. Initialize string table (Dictionary)
2. STRING = get input character
3. While there are still input characters Do
4. CHAR = get input character

   If STRING + CHAR is in the string table then

   STRING = STRING + CHAR

   ELSE

   Output the code for string.

```
        Add STRING + CHAR to the string table.
        STRING = CHAR
    END of IF
    END OF WHILE
Output the code for STRING.
```

The compression algorithm uses two variables: CHAR and STRING. The variable, CHAR, holds a single character, that is, a single byte value between 0 and 255. The variable, STRING, is a variable length string, that is, a group of one or more characters, with each character being a single byte. Fig. 3.1 is the compression flow chart. In box 1 of fig. 3.1, the program starts by taking the first byte from the input file, and placing it in the variable, STRING. Table 3.1 shows this action in line one. The algorithm loops for each additional byte the input file; this is controlled in the flow diagram by box 8. Each time a byte is read from the input file (box 2), it is stored in the variable, CHAR. The data table is then searched to determine if the concatenation of the two variables, STRING + CHAR, has already been assigned a code (box 3).

If a match in the code table is not found, three actions are taking, as show in boxes 4, 5 and 6. In box 4, an 'N' bit code corresponding to the content of the variable, STRING, is written to the compressed file. where 'N' is the dictionary size in bits. In box 5, a new code is created in the table for the concatenation of STRING + CHAR. In box 6, the variable, STRING, takes the value of the variable, CHAR. These actions are shown in line 2 through 8 in Table 3.1.

When a match in the code table is found (Box 3), the concatenation of STRING + CHAR is stored in the variable, STRING, without any other action taking place (box 7). That is, if a matching sequence is found in the table, no action should be taken before determining if there is a longer matching sequence also in the table. This is shown in line 9, where the sequence: STRING + CHAR = LO, is identified as already having a code in the table. In line 10, the next character from the input file, is added to the sequence and code table is searched for: LOV. Since this longer sequence is not in the table the program adds it to the table, outputs the code for the shorter sequence that is in the table (code 258), and starts over searching for sequences beginning with the character, V. This flow of event is continued until there are no more characters in the input file. The program is wrapped up with the code corresponding to the table value of STRING being written to the compressed file.

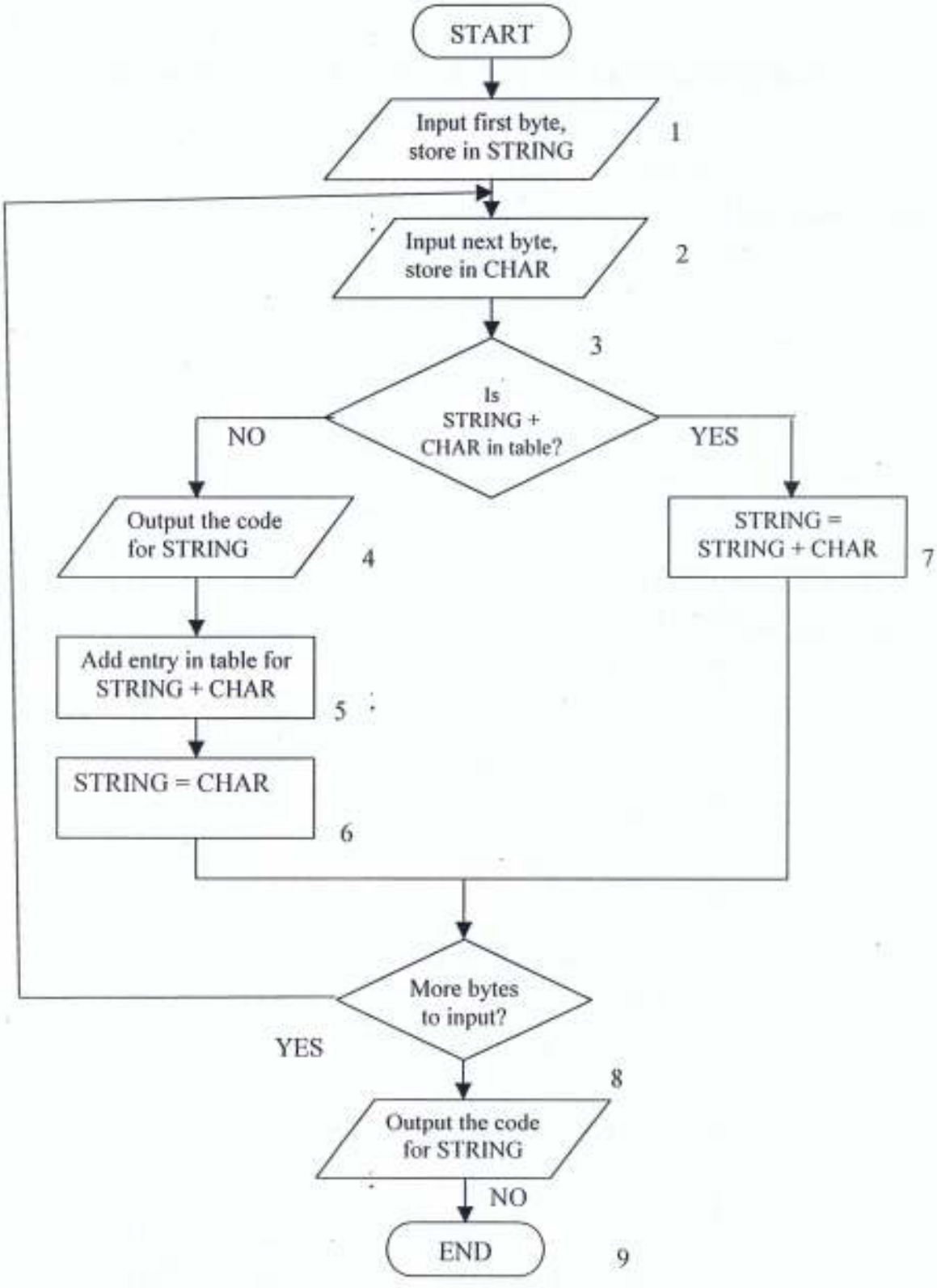

Fig. 3.1 LZW Compression Flow Chart

Table 3.1 LZW Compression Process

Input String = FALOWO/LOVES/OLOWO/OF/OWO//							
S/N	CHAR	STRING + CHAR	In Table?	Output	Add to Table	New STRING	Comment
1	F	F				F	First character (no action).
2	A	FA	No	F	256 = FA	A	
3	L	AL	No	A	257 = AL	L	
4	O	LO	No	L	258 = LO	O	
5	W	OW	No	O	259 = OW	W	
6	O	WO	No	W	260 = WO	O	
7	/	O/	No	O	261 = O/	/	
8	L	/L	No	/	262 = /L	L	
9	O	LO	Yes			LO	First match found.
10	V	LOV	No	256	263 = LOV	V	LOV added to the table
11	E	VE	No	V	264 = VE	E	
12	S	ES	No	E	265 = ES	S	
13	/	S/	No	S	266 = S/	/	
14	O	/O	No	/	267 = /O	O	
15	L	OL	No	O	268 = OL	L	
16	O	LO	Yes			LO	
17	W	LOW	No	257	269 = LOW	W	
18	O	WO	Yes			WO	
19	/	WO/	No	258	270 = WO	/	
20	O	/O	Yes			/O	
21	F	/OF	No	259	271 = /OF	F	
22	/	F/	No	F	272 = F/	/	
23	O	/O	Yes			/O	
24	W	/OW	No	260	273 = /OW	W	
25	O	WO	Yes			WO	Matches WO
26	/	WO/	Yes			WO/	Matches longer string WO/
27	/	WO//	No	261	274 = WO//	/	
28	EOF	/		/			End of file, output STRING

3.2 LZW DECOMPRESSION ALGORITHM

The LZW decompression algorithm is shown below.

1. Initialize string table (Dictionary)
2. Get the first code: OLD_CODE
3. Output the string for OLD_CODE
4. CHARACTER = OLD_CODE
5. WHILE there are still input characters DO

    Get NEW_CODE

    IF NEW_CODE is not in the translation table THEN

        STRING = get translation of OLD_CODE

        STRING = STRING + CHARACTER

    ELSE

        STRING = get translation of NEW_CODE

    END of IF

    Output STRING

    CHARACTER = first character in STRING

    Add OLD CODE + CHARACTER to the translation table

    OLD_CODE = NEW_COD

    END of WHILE

A flow chart of the LZW decompression algorithm is shown in Fig. 3.2. Each code is read from the compressed file and compared to the code table to provide the translation. As each code is processed in this manner, the code table is updated so that it continually matches the one used during decompression. However, there is a small complication in the decompression routine there are certain combinations of data that result in the decompression algorithm receiving a code that does not yet exist in the code table. This contingency is handled in boxes 4, 5 and 6. The decompression process is shown in Table 3.2.

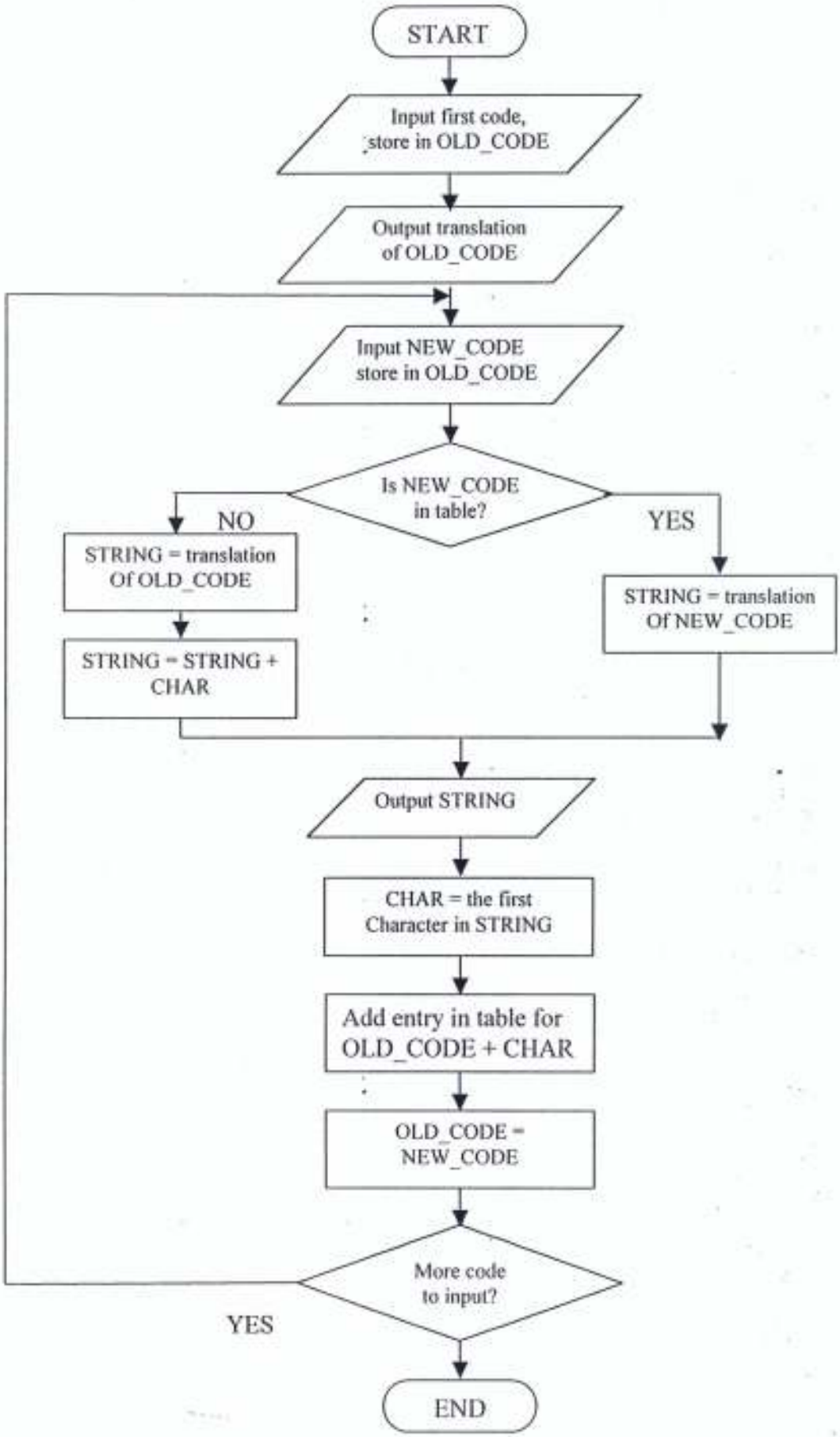

Fig. 3.2 LZW Decompression Flow Chart

Table 3.2 LZW Decompression Process

Input Code: F A L O W O / 258 V E S / O 258 260 267 F 267 270 /					
S/N	nput/NEW_CODE	OLD_CODE	STRING / output	CHARACTER	Add to table
1	F	F	F		
2	A	F	A	A	FA = 256
3	L	A	L	L	AL = 257
4	O	L	O	O	LO = 258
5	W	O	W	W	OW = 259
6	O	W	O	O	WO = 260
7	/	O	/	/	O/ = 261
8	258	/	LO	L	/L = 262
9	V	258	V	V	LOV = 263
10	E	V	E	E	VE = 264
11	S	E	S	S	ES = 265
12	/	S	/	/	S/ = 266
13	O	/	O	O	/O = 267
14	258	O	LO	L	OL = 268
15	260	258	WO	W	LOW = 269
16	267	260	/O	/	WOL = 270
17	F	267	F	F	/OF = 271
18	267	F	/O	/	F/ = 272
19	270	267	WO/	W	/OW = 273
20	/	270	/	/	WO// = 274

3.3 IMPLIMENTATION

3.3.1 CODE TABLE (DICTIONARY)

Different Dictionary sizes of 10, 11,12,13,14,15 and 16 bits are used in this project. This corresponds to code tables of 1024, 2048, 4096, 8192, 16384, 32768 and 655336 entries respectively. Codes 0-255 in the code table are always assigned to represent single bytes from the standard ASCII code.

When a new character is read in, the string character has to be searched for a match. If a match is not found, the new string has to be added to the table. The most remarkable feature of this type of compression is that the entire dictionary is transmitted to the decompressor without actually explicitly transmitting it. At the end of the decompression the decompressor will have a dictionary identical to that of the compressor.

3.3.2 HASHING FUNCTION

One of the major implementation considerations involves the way in which the string table is stored and accessed. The search through the table can be computational intensive therefore a hashing function becomes necessary in order to reduce the amount of time require for a string comparison. The hashing function used in this research work is the "xor" type hashing function. The prefix code and appended character are combined to form an array address. If the contents of the prefix code and character are matched, the correct address is returned. If that element in the array is in used, a fixed offset probe is used to search new locations. This continues until either an empty slot is found, or a match is found. The tables used in this research work are about 10% larger than necessary in order to reduce the average number of searches in the table.

3.3.3 STACK BUFFER

The method used for storing string values causes the string to be decoded in reversed order. Therefore it is necessary to decode all the characters for a given string into a stack buffer and then output them in the reverse order. In this research work this is done using the decode_string function.

3.4 THE DATA COMPRESSION APPLICATION PROGRAM

The input to the data compression application program is a file name. It takes the file, opens it, compresses the content and put it in another file called the output file. The name of the output file is formed from the input file name by appending '**c**' to the input file name. For example, if the input file name is '**bisi**', the output file is named '**bisic**'. The appended '**c**' is to indicate that **bisic** is a compressed form of the original file (input file), **bisi**. To decompress the file, the decompressor takes the name of the compressed file as its input. It opens the file, decompresses (expands) it and put the content in an output file. The output file name created by the decompressor is the same as the name of the original file that was compressed. For example, if the input to the decompressor is a file named **bisic,** the output file is named **bisi.**

3.5. WHY C ++

Every programming language has its inherent strengths and weaknesses. The code for this research work was written in C ++ for the following reasons.

3.5.1 BIT MANIPULATION

In data compression programming it is necessary to manipulate objects at the bit level. C ++ provides a rich set of bit-manipulation operators

3.5.2 PORTABILITY

C ++ is highly portable. This means that a C ++ program can run with little or no modification on different kinds of computers (computer with different processor or different operating systems).

3.5.3 SMALL SIZE

There fewer syntax rule in C ++ than in many other programming languages, and there are actually more operators and combinations of operators in C ++ than there are key words.

3.5.4 SPEED

Speed is an important factor in data compression programs. The C ++ code produced by most compilers tends to be very efficient. The combination of small language, a small run-time, and the fact that the language is close to the hardware makes many C ++ programs run at speed close to their assembly language equivalent.

3.5.5 MEMORRY EFFICIENCY

C ++ programs tend to be memory efficient for the same reason that they have high speed. The lack of in-built function saves program from having to carry around support for functions that are not needed by that application.

3.5.6 FLEXIBILITY

C ++ allows programs to be written is a structured way yet it permits freedom of expression. It contains all the control structures you would expect of a modern-day language.

CHAPTER FOUR

RESULTS AND DISCUSSION

In order to determine the efficiency of the developed data compression system, it is necessary to carry out a series of tests. The compression program was tested with source data 1, 2, 3, 4 and 5 as inputs. The compression ratio for each dictionary was determined for different sizes of the various input data. In this project the amount of compression is measured by the compression ratio. For example, 35% compression ratio implies that the compressed file is 35% of the original file.

$$Compression\ Ratio\ (CR) = \frac{Size\ of\ the\ Compressed\ File}{Size\ of\ the\ Original\ File} \times 100\% \quad \text{.....4.1}$$

$$CR = \frac{No\ of\ Characters\ in\ the\ Compressed\ File \times the\ Dictionary\ Size}{No\ of\ Characters\ in\ the\ Original\ File \times 8} \times 100\% \text{.....4.2}$$

4.1 VARIATION IN COMPRESSION RATIO WITH DICTIONARY SIZE

Consider a highly redundant input data stream: **W A W A W A W A W A W A.** Figure 4.1 shows the code table entries for the dictionary sizes of 8, 10, 11, 12 and 13 bits while figure 4.2 shows the code table entries for the dictionary sizes of 14, 15 and 16 bits. The LZW compression process for the input stream is show in figure 4.3. Tables 4.4, 4.5, 4.6, 4.7, 4.8, 4.9 and 4.10 shows the compressing process using dictionary sizes of 10, 11, 12, 13, 14, 15 and 16 bits respectively. The compression ratio for each dictionary size is calculated. It can be seen that the compression ratios vary with the dictionary sizes.

Table 4.1 Code Table Entries For Dictionary Sizes of 8, 10, 11, 12 and 13 Bits

Character	Decimal Value	8 Bit Dictionary	10 Bit Dictionary	11 Bit Dictionary	12 Bit Dictionary	13 Bit Dictionary
Null	0	00000000	0000000000	00000000000	000000000000	0000000000000
Start of Heading	1	00000001	0000000001	00000000001	000000000001	0000000000001
Start of Text	2	00000010	0000000010	00000000010	000000000010	0000000000010
.	.	.	.	.	.	.
.	.	.	.	.	.	.
@	64	01000000	0001000000	00001000000	000001000000	0000001000000
A	65	01000001	0001000001	00001000001	000001000001	0000001000001
B	66	01000010	0001000010	00001000010	000001000010	0000001000010
.	.	.	.	.	.	.
.	.	.	.	.	.	.
V	86	01010110	0001010110	00001010110	000001010110	0000001010110
W	87	01010111	0001010111	00001010111	000001010111	0000001010111
X	88	01011000	0001011000	00001011000	000001011000	0000001011000
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	255	11111111	0011111111	00011111111	000011111111	0000011111111
.	.		.	.	.	.
.	.		.	.	.	.
.	1023		1111111111	01111111111	00111111111	00011111111111
.	.			.	.	.
.	2047			11111111111	01111111111	00111111111111
.	.					.
.	4095				11111111111	01111111111111
	.					.
	8191					11111111111111

Table 4.2 Code Table Entries For Dictionary Sizes of 14, 15 and 16 Bits

Character	Decimal Value	14 Bit Dictionary	15 Bit Dictionary	16 Bit Dictionary
Null	0	00000000000000	000000000000000	0000000000000000
Start of Heading	1	00000000000001	000000000000001	0000000000000001
Start of Text	2	00000000000010	000000000000010	0000000000000010
.	.	.	.	.
.	.	.	.	.
@	64	00000001000000	0000000001000000	0000000001000000
A	65	00000001000001	0000000001000001	0000000001000001
B	66	00000001000010	0000000001000010	0000000001000010
.	.	.	.	.
.	.	.	.	.
V	86	00000001010110	0000000001010110	0000000001010110
W	87	00000001010111	0000000001010111	0000000001010111
X	88	00000001011000	0000000001011000	0000000001011000
.	.	.	.	.
.	.	.	.	.
	255	00000011111111	0000000111111111	0000000011111111
.	.	.	.	.
.	.	.	.	.
.	16383	11111111111111	011111111111111	0011111111111111
.	.	.	.	.
.	.	.	.	.
.	32767		111111111111111	0111111111111111
.	.	.		.
.	.			.
.	65535			1111111111111111

Table 4.3 LZW Compression Process

Input String = WAWAWAWAWAWA						
S/N	CHAR	STRING + CHAR	In Table?	Output	Add to Table	New STRING
1	W	W				W
2	A	WA	No	W	256 = WA	A
3	W	AW	No	A	257 = AW	L
4	A	WA	Yes			WA
5	W	WAW	No	WA	258 = WAW	W
6	A	WA	Yes			WA
7	W	WAW	Yes			WAW
8	A	WAWA	No	WAW	259= WAWA	A
9	W	AW	Yes			AW
10	A	AWA	No	AW	260=AWA	A
11	W	AW	Yes			AW
12	A	AWA	Yes			AWA
13	EOF	AWA		AWA		

Table 4.4 LZW Compression Process Using 8-Bit Dictionary

Input Data: W A W A W A W A W A W A		
S/N	Input Character	8 Bit Word
1	W	01010111
2	A	01000001
3	W	01010111
4	A	01000001
5	W	01010111
6	A	01000001
7	W	01010111
8	A	01000001
9	W	01010111
10	A	01000001
11	W	01010111
12	A	01000001

Table 4.5 LZW Compression Process Using 10-Bit Dictionary

Input Data: W A WA WAW AW AWA				
S/N	Input Character	Decimal Value	10 Bit Word	8 Bit Word (Storage Format)
1	W	87	0001010111	00010101
2	A	65	0001000001	11000100
3	WA	256	0100000000	00010100
4	WAW	258	0100000010	00000010
5	AW	257	0100000001	00000010
6	AWA	260	0100000100	01000000
7				01010000
8				01000000

Using equation 4.2, Compression Ratio (CR) for the dictionary size of 10 bits is given by:

$$CR = \frac{6 \times 10}{12 \times 8} \times 100\%$$

$$CR = 62.5\ \%$$

Table 4.6 LZW Compression Process Using 11-Bit Dictionary

Input Data: W A WA WAW AW AWA				
S/N	Input Character	Decimal Value	11 Bit Word	8 Bit Word (Storage Format)
1	W	87	00001010111	00001010
2	A	65	00001000001	11100001
3	WA	256	00100000000	00000100
4	WAW	258	00100000010	10000000
5	AW	257	00100000001	00010000
6	AWA	260	00100000100	00100010
7				00000010
8				01000001
9				00000000

Using equation 4.2, Compression Ratio (CR) for the dictionary size of 11 bits is given by:

$$CR = \frac{6 \times 11}{12 \times 8} \times 100\%$$

$$CR = 68.75\ \%$$

Table 4.7 LZW Compression Process Using 12-Bit Dictionary

Input Data: W A WA WAW AW AWA				
S/N	Input Character	Decimal Value	12 Bit Word	8 Bit Word (Storage Format)
1	W	87	000001010111	00000101
2	A	65	000001000001	01110000
3	WA	256	000100000000	01000001
4	WAW	258	000100000010	00010000
5	AW	257	000100000001	00000001
6	AWA	260	000100000100	00000010
7				00010000
8				00010001
9				00000100

Using equation 4.2, Compression Ratio (CR) for the dictionary size of 12 bits is given by:

$$CR = \frac{6 \times 12}{12 \times 8} \times 100\%$$

$$CR = 75\ \%$$

Table 4.8 LZW Compression Process Using 13-Bit Dictionary

Input Data: W A WA WAW AW AWA				
S/N	Input Character	Decimal Value	13 Bit Word	8 Bit Word (Storage Format)
1	W	87	0000001010111	00000010
2	A	65	0000001000001	10111000
3	WA	256	0000100000000	00010000
4	WAW	258	0000100000010	01000010
6	AWA	260	0000100000100	00010000
7				00100000
8				10000000
9				10000100
10				00010000

Using equation 4.2, Compression Ratio (CR) for the dictionary size of 13 bits is given by:

$$CR = \frac{6 \times 13}{12 \times 8} \times 100\%$$

$$CR = 81.25\ \%$$

Table 4.9 Compression Process Using 14-Bit Dictionary

Input Data: W A WA WAW AW AWA				
S/N	Input Character	Decimal Value	14 Bit Word	8 Bit Word (Storage Format)
1	W	87	00000001010111	00000001
2	A	65	00000001000001	01011100
3	WA	256	00000100000000	00000100
4	WAW	258	00000100000010	00010000
5	AW	257	00000100000001	01000000
6	AWA	260	00000100000100	00000001
7				00000010
8				00000100
9				00000100
10				00010000
11				01000000

Using equation 4.2, Compression Ratio (CR) for the dictionary size of 14 bits is given by:

$$CR = \frac{6 \times 14}{12 \times 8} \times 100\%$$

$$CR \approx 87.5\ \%$$

Table 4.10 Compression Process Using 15-Bit Dictionary

Input Data: W A WA WAW AW AWA				
S/N	Input Character	Decimal Value	15 Bit Word	8 Bit Word (Storage Format)
1	W	87	000000001010111	00000000
2	A	65	000000001000001	10101110
3	WA	256	000000100000000	00000001
4	WAW	258	000000100000010	00000100
5	AW	257	000000100000001	00001000
6	AWA	260	000000100000100	00000000
7				00010000
8				00100000
9				00100000
10				00100000
11				01000001
12				00**000000**

Using equation 4.2, Compression Ratio (CR) for the dictionary size of 10 bits is given by:

$$CR = \frac{6 \times 15}{12 \times 8} \times 100\%$$

$$CR = 93.75\ \%$$

Table 4.11 Compression Process Using 16-Bit Dictionary

Input Data: W A WA WAW AW AWA				
S/N	Input Character	Decimal Value	16 Bit Word	8 Bit Word (Storage Format)
1	W	87	0000000001010111	00000000
2	A	65	0000000001000001	01010111
3	WA	256	0000000100000000	00000000
4	WAW	258	0000000100000010	01000001
5	AW	257	0000000100000001	00000001
6	AWA	260	0000000100000100	00000000
7				00000001
8				00000010
9				00000001
10				00000001
11				00000001
12				00000100

Using equation 4.2, Compression Ratio (CR) for the dictionary size of 10 bits is given by:

$$CR = \frac{6 \times 16}{12 \times 8} \times 100\%$$

$$CR = 100\,\%$$

Tables 4.12, 4.13, 4.14, 4.15, and 4.16 shows the compression ratios for the different dictionary sizes using the input source data 1, 2, 3, 4 and 5 respectively, table 4.17(a) shows the compression ratio average for the different dictionary sizes while table 4.17 (b) shows the compression ratio average for the different dictionary using the mean value of the four data sizes. The compression ratio for any input source data depends on the following:

(i) Dictionary Size of the compressor

(ii) Size of the input data stream

(iii) Redundancy in the input data.

Figures 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 are bar charts showing the compression ratios for the dictionary sizes of 10, 11, 12, 13, 14, 15, and 16 bits respectively using source data 1 as the input. From the bar charts it can be seen that the smaller dictionaries compress better when the input data is a few kilobytes. As the size of the input data increases, compression ratio decreases. On the other hand, the compression ratio for the bigger dictionary sizes is small at the beginning but it later increases as the size of the input data increases until it gets to a limit. Figures 4.7, 4.8, 4.9, 4.10 and 4.11 compare the compression ratio of the seven dictionaries for input source data 1, 2, 3, 4 and 5 respectively. The compression ratio average for the different dictionary sizes is shown in figures12 (a) and 12 (b). The optimum dictionary size is the one that gives the best compression in the average for different sizes of input data. From figure 12(b), it is seen that the dictionary size of 14 bits is the optimum dictionary.

Table 4.12 Compression Ratios for Different Dictionary Sizes Using Source Data 1 as the Input

Dictionary Size (No of Bits) / Size of Input Data. (kB)	8	10	11	12	13	14	15	16
10	100	30.0	29.6	32.2	34.7	37.4	40.1	42.8
100	100	38.3	34.6	34.0	33.6	32.8	34.9	37.3
1000	100	47.6	44.3	42.9	41.6	39.7	39.5	37.0
10000	100	51.8	47.4	48.1	47.1	42.9	42.8	43.5

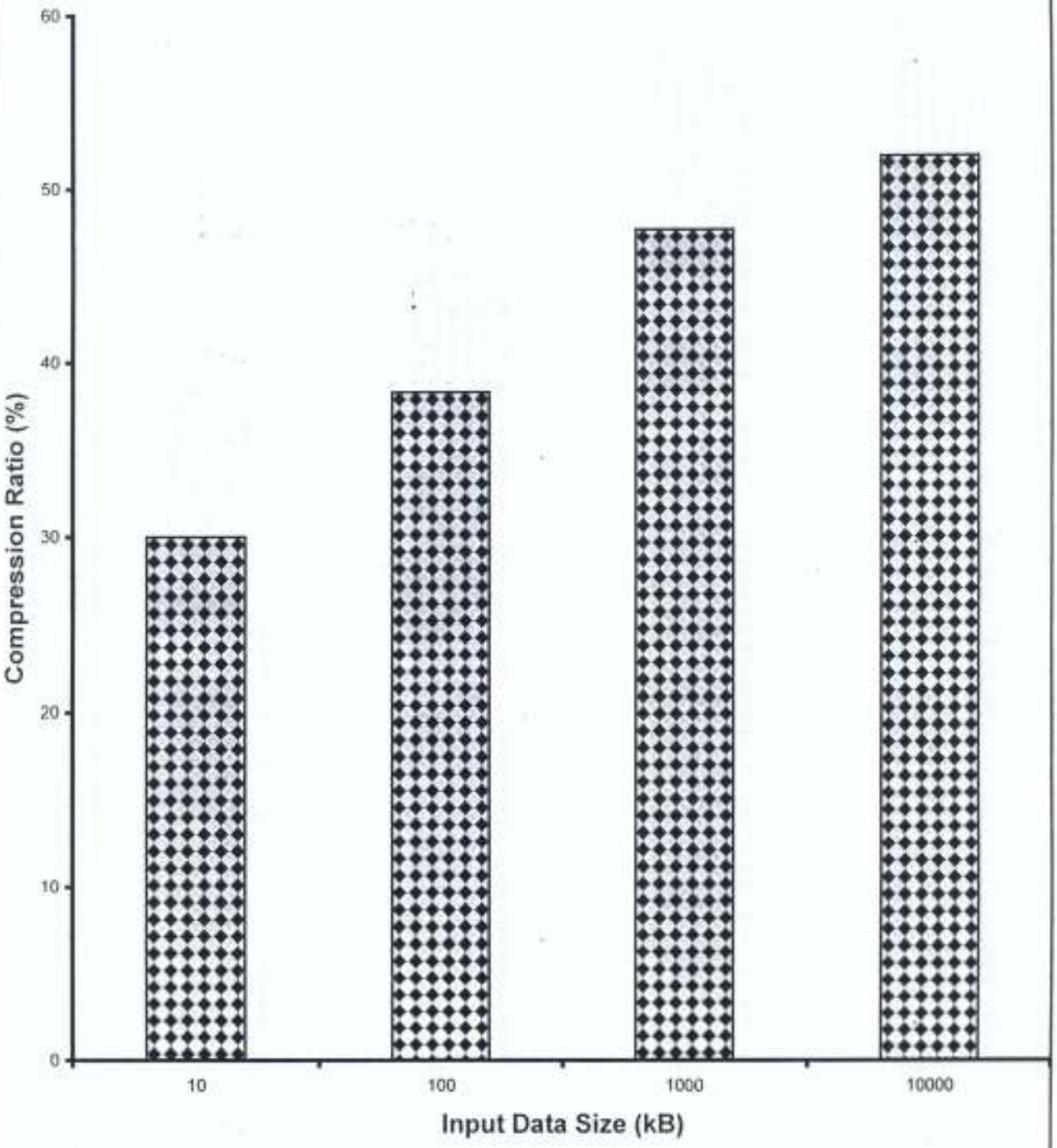

Fig. 4.0 Compression Ratio for a Dictinary Size Of 10 Bits Using The Input Source Data 1

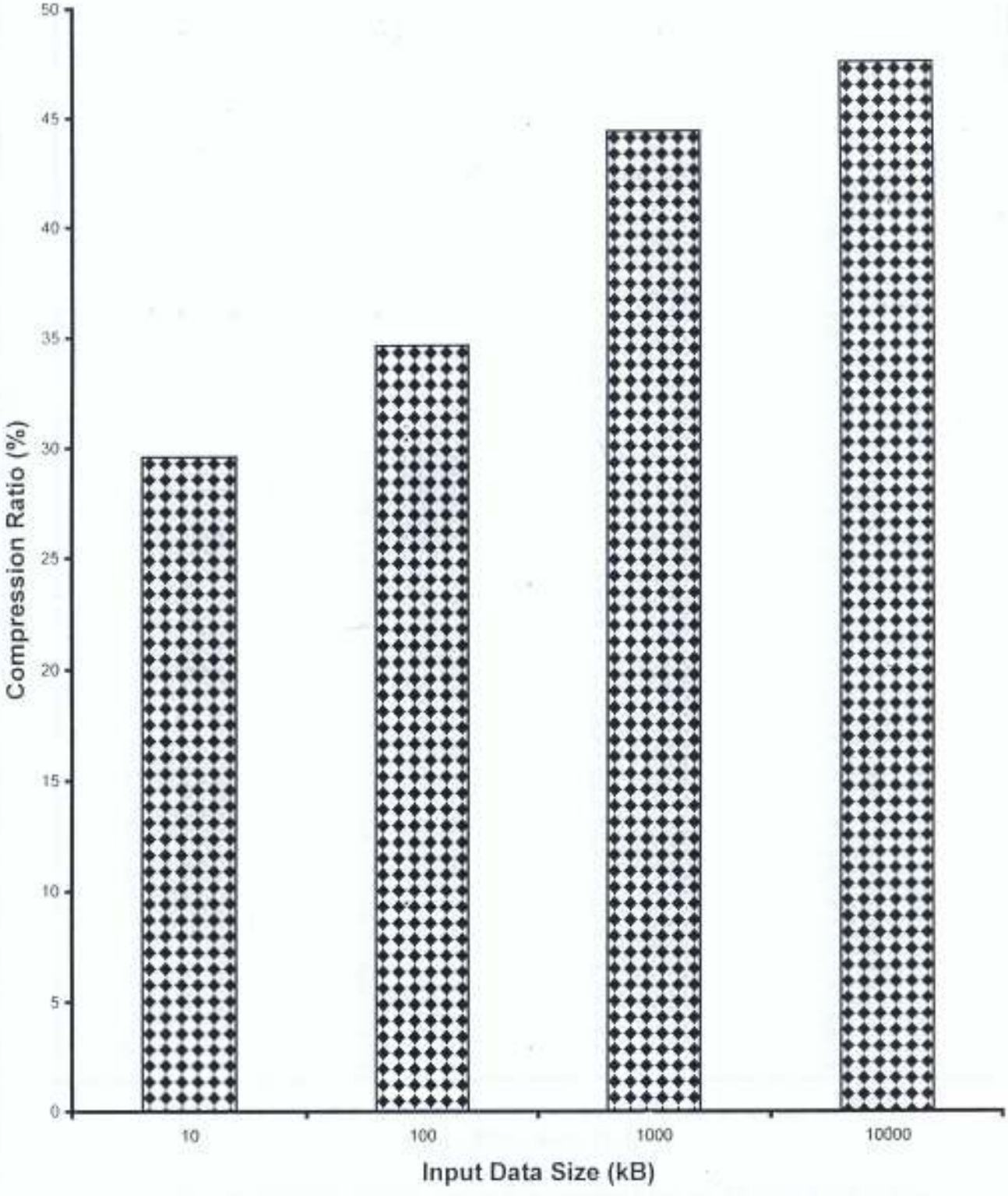

Fig.4.1 Compression Ratio For a Dictionary Size Of 11 Bits Using The Input Source Data 1.

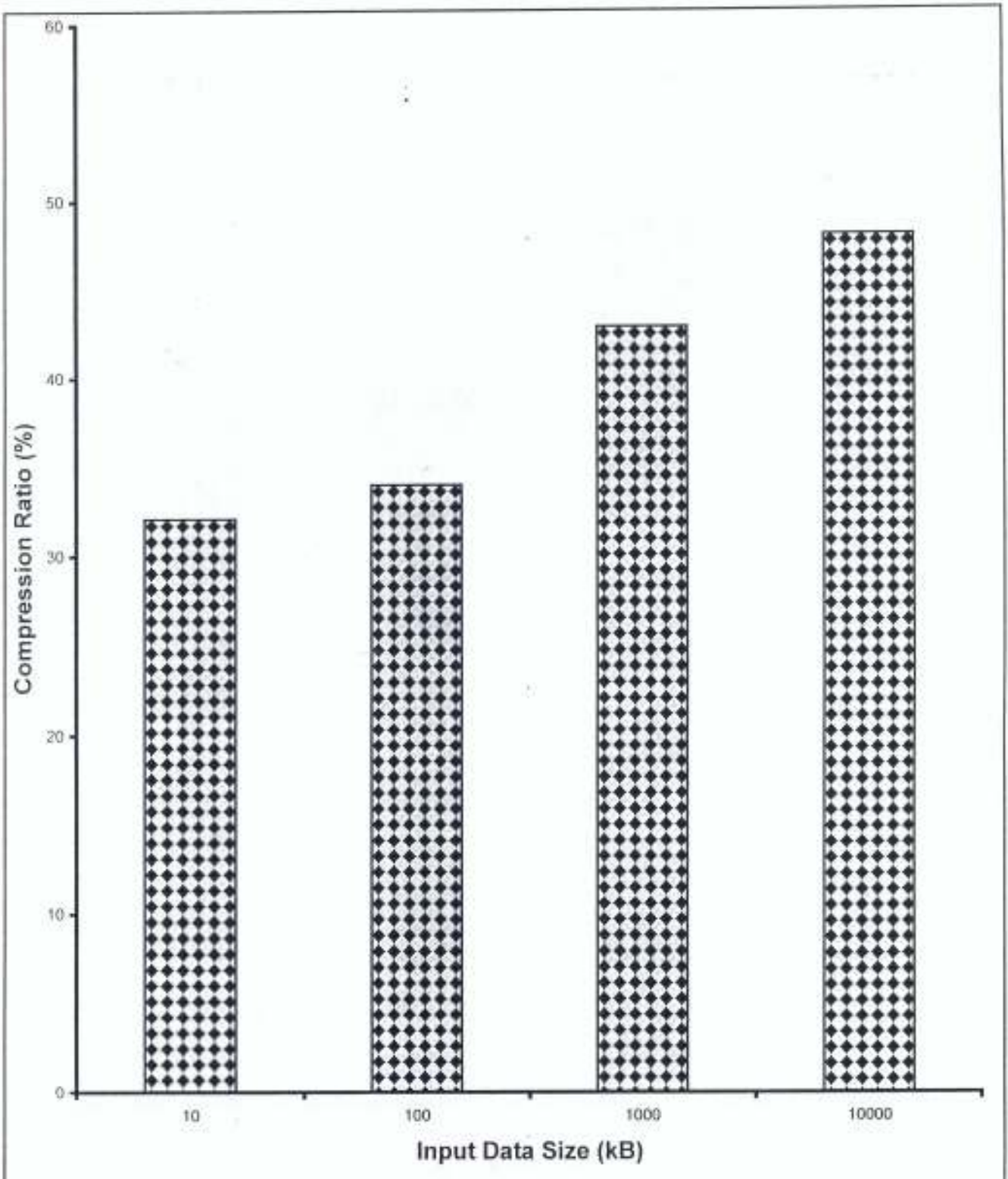

Fig.4.2 Compression Ratio For a Dictionary Size of 12 Bits Using The Input Source Data 1.

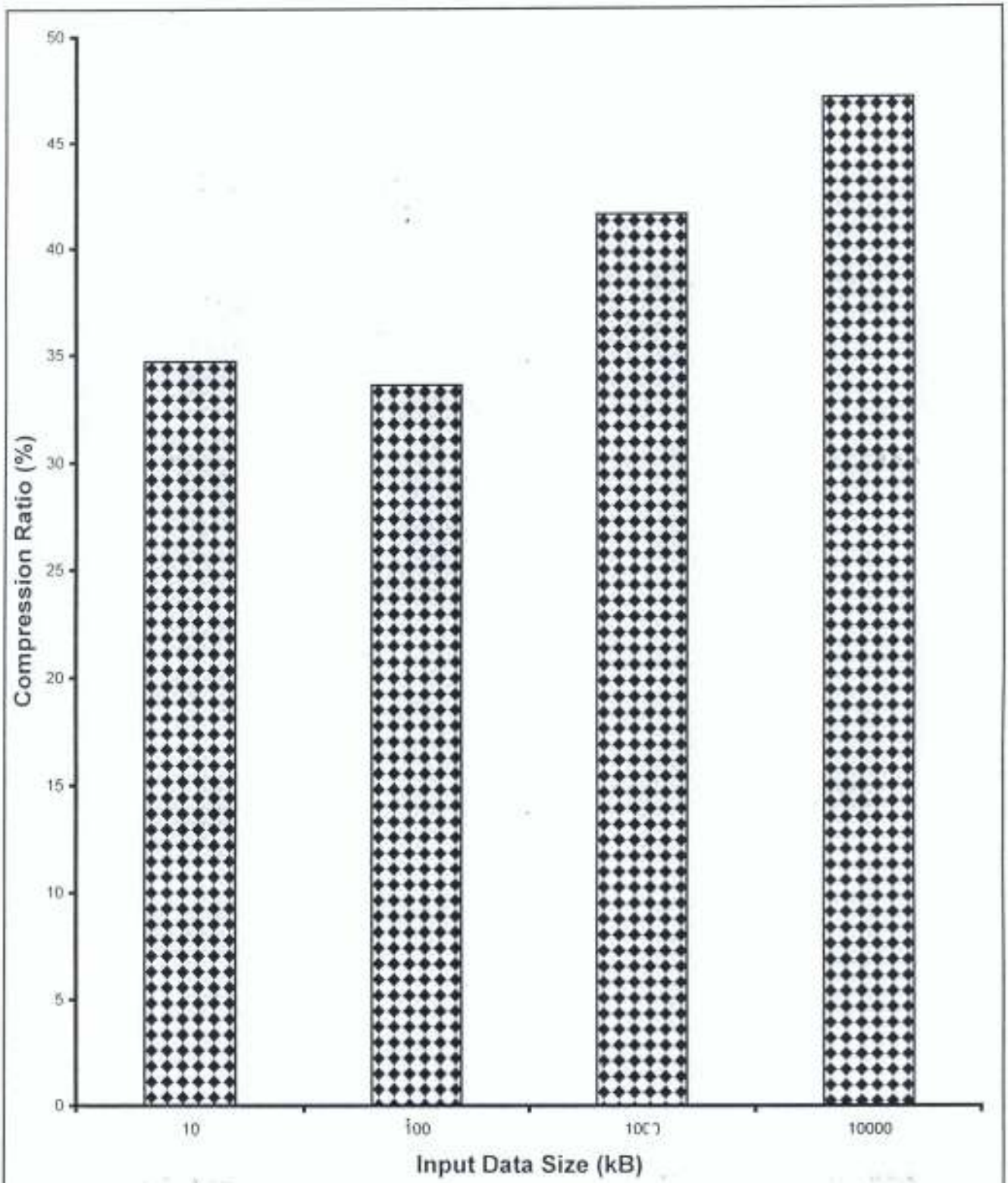

Fig.4.3 Compression Ratio For a Dictionary Size of 13 Bits Using The Input Source Data 1.

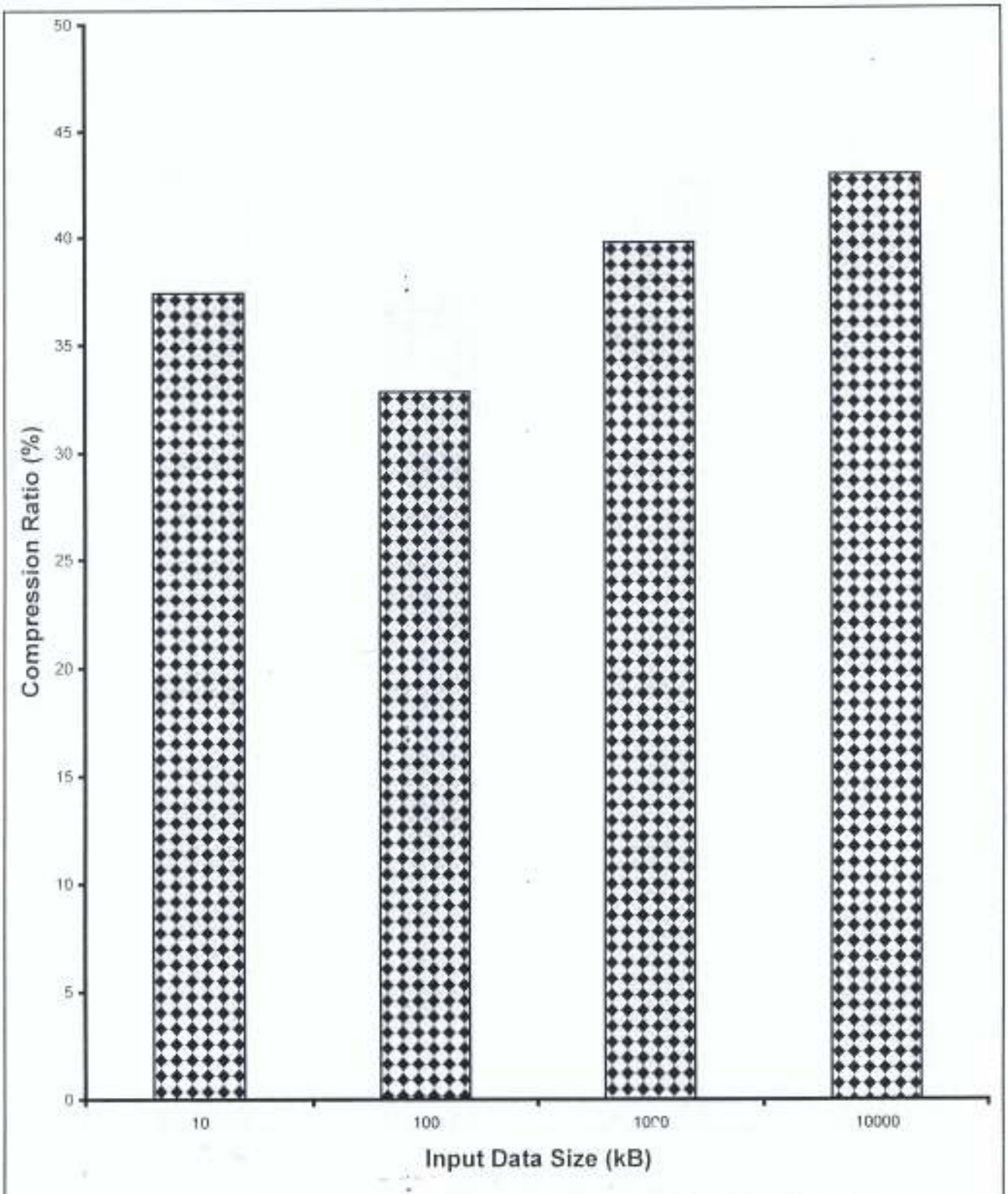

Fig.4.4 Compression Ratio For a Dictionary Size Of 14 Using The Input Source Data 1.

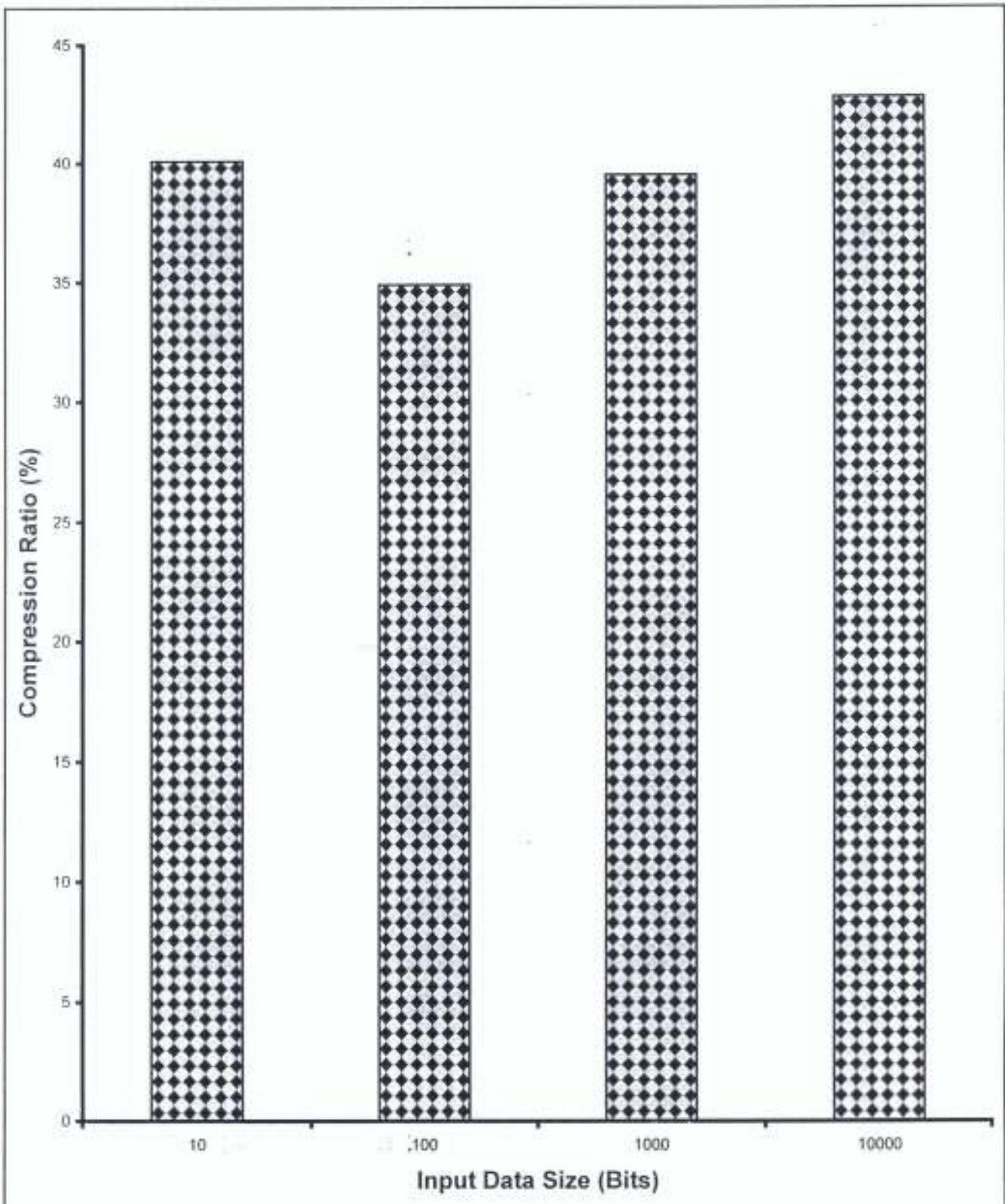

Fig.4.5 Compression Ratio For a Dictionary Size of 15 Bits Using The Input Source Data 1.

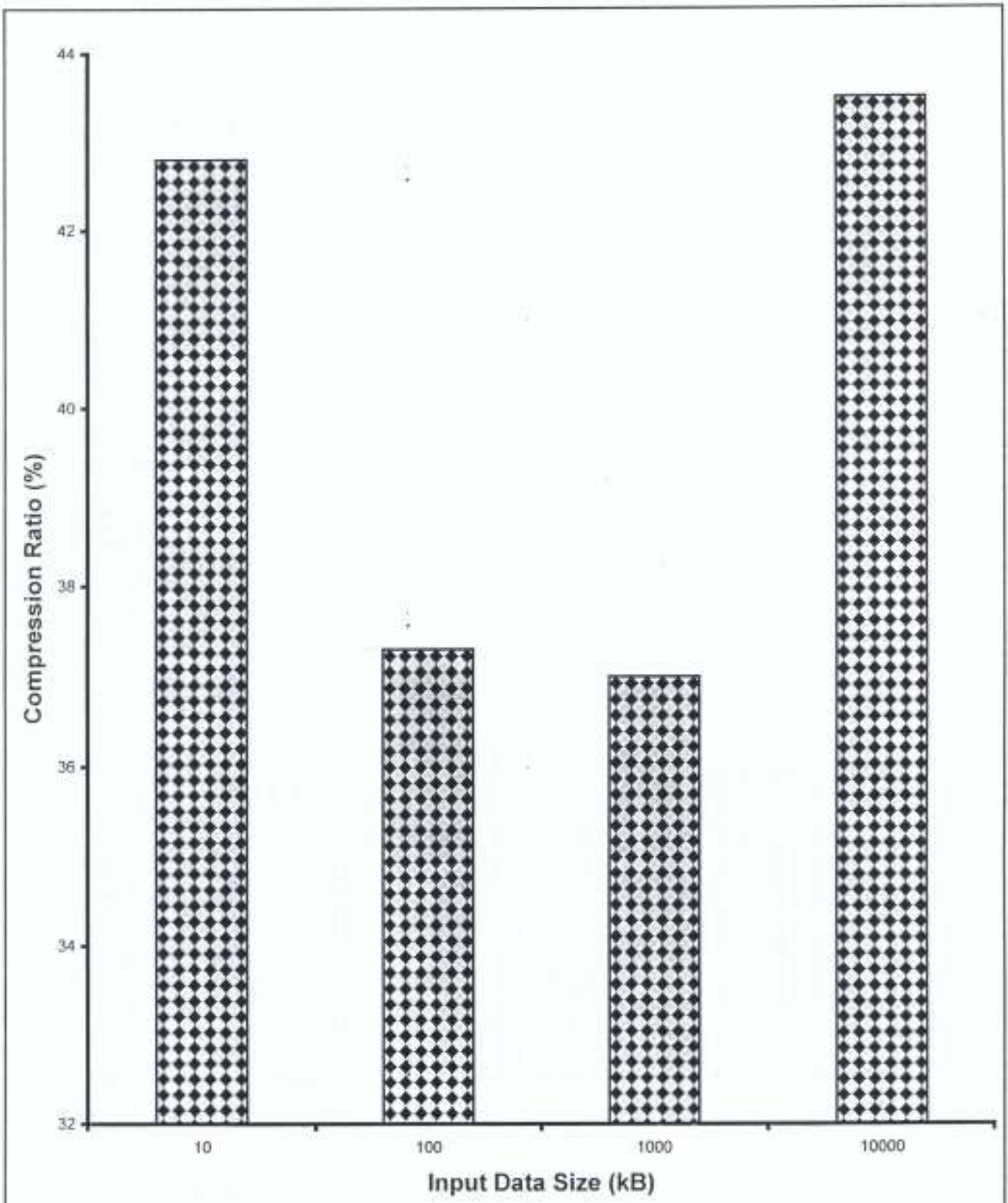

Fig.4.6 Compression Ratio For a Dictionary Size Of 16 Bits Using The Input Source Data 1

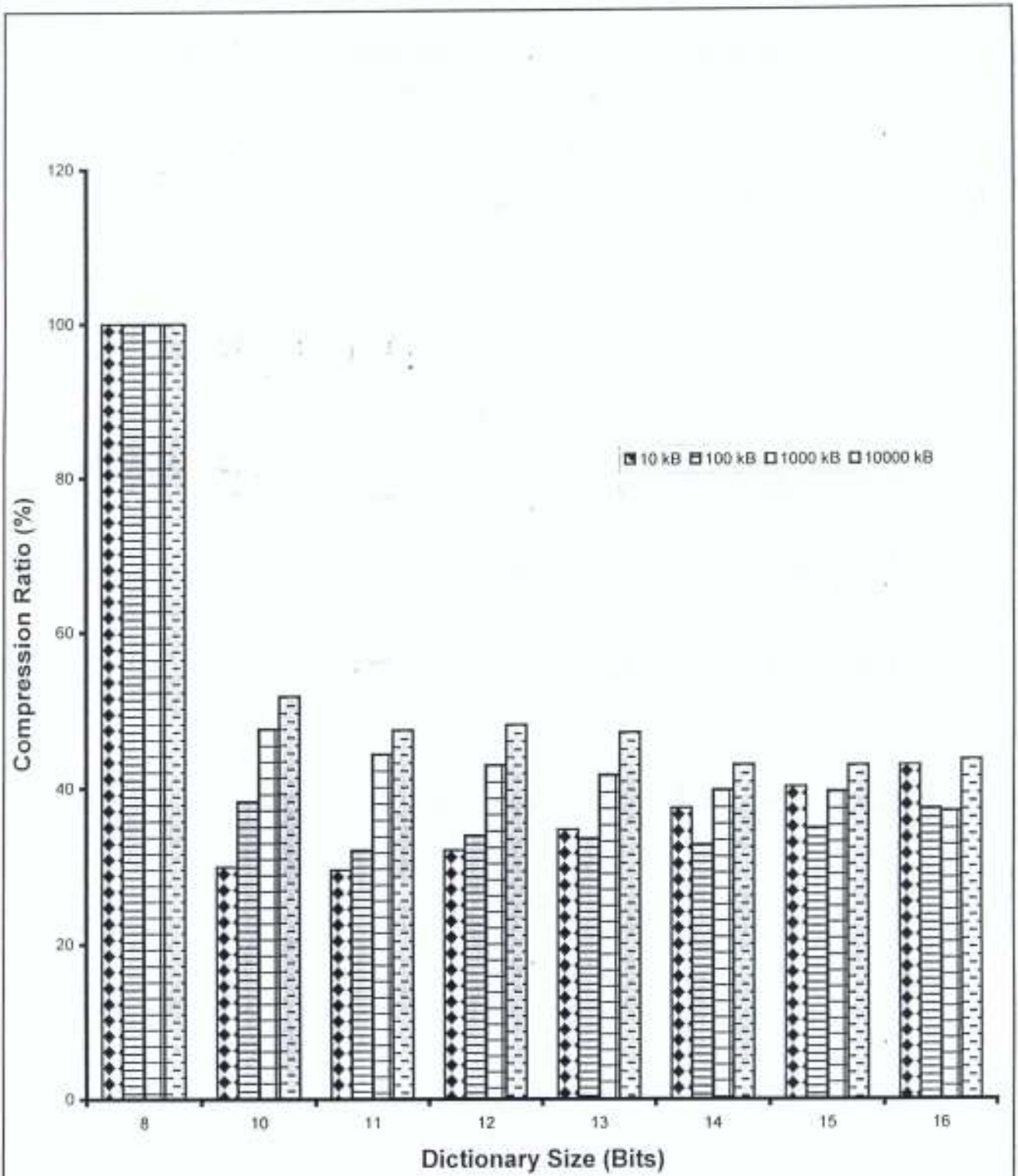

Fig. 4.7 Compression Ratio For Different Dictionary Sizes Using The Input Source Data 1

Table 4.13 Compression Ratios For Different Dictionary Sizes Using Source Data 2 as the Input

Dictionary Size (No of Bits) / Size of Input Data. (kB)	8	10	11	12	13	14	15	16
1	100	35.1	31.2	32.7	35.4	38.1	40.9	43.6
2	100	42.1	38.2	36.5	36.7	34.3	35.0	37.4
3	100	49.5	45.8	45.6	44.8	41.4	32.0	42.1
4	100	50.3	46.3	46.4	45.6	41.7	41.2	42.5

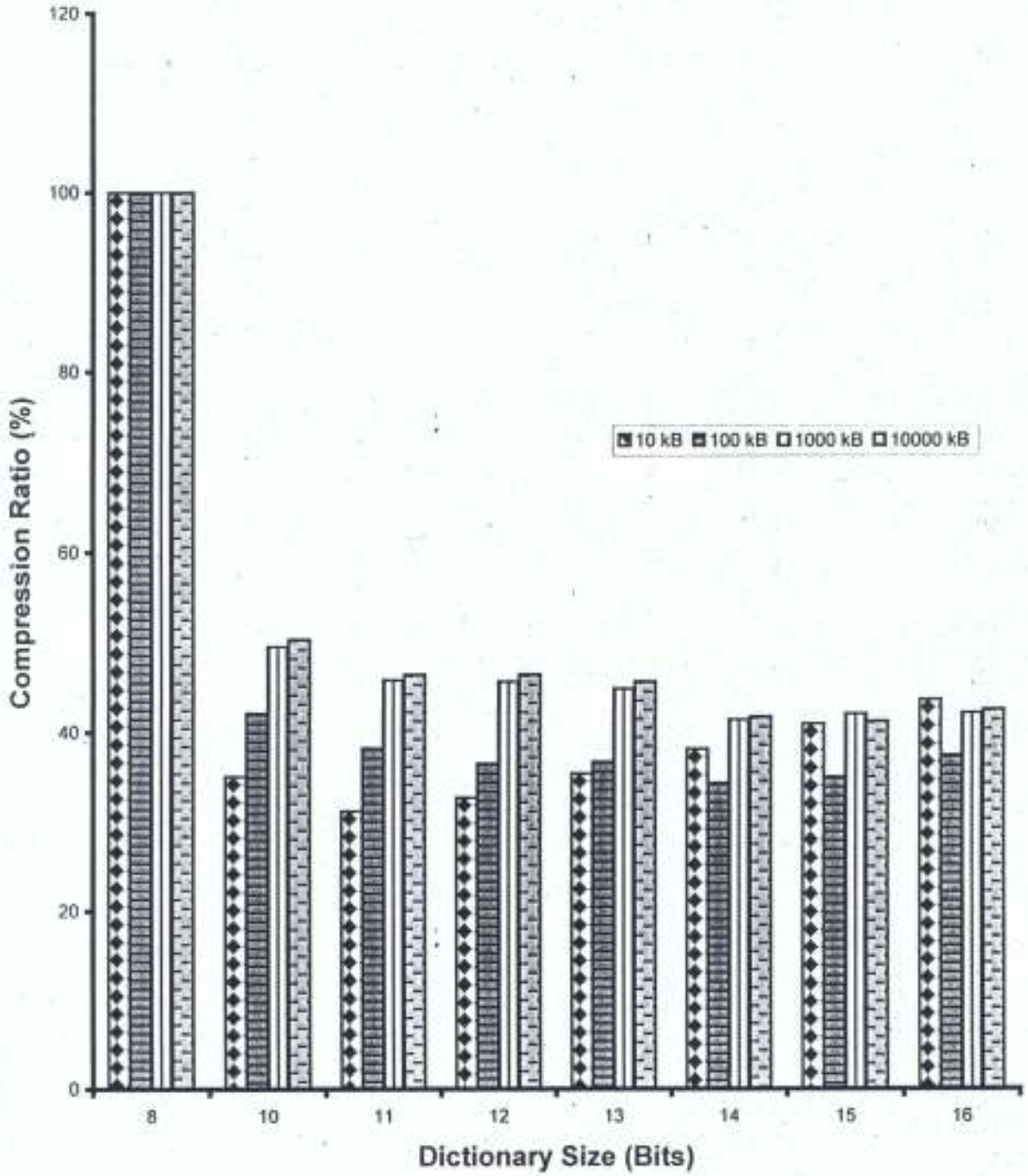

Fig. 4.8 Compression Ratios for Different Dictionary Sizes Using The Input Source Data 2.

Table 4.14 Compression Ratios For Different Dictionary Sizes Using Source Data 3 as the Input

Dictionary Size (No of Bits) / Size of Input Data. (kB)	8	10	11	12	13	14	15	16
10	100	34.9	31.3	33.5	36.3	39.1	41.9	44.7
100	100	41.3	38.1	38.0	36.2	34.1	35.4	37.7
1000	100	42.3	40.1	36.4	35.14	32.8	29.9	29.0
10000	100	55.1	50.8	51.3	50.4	48.3	47.8	49.0

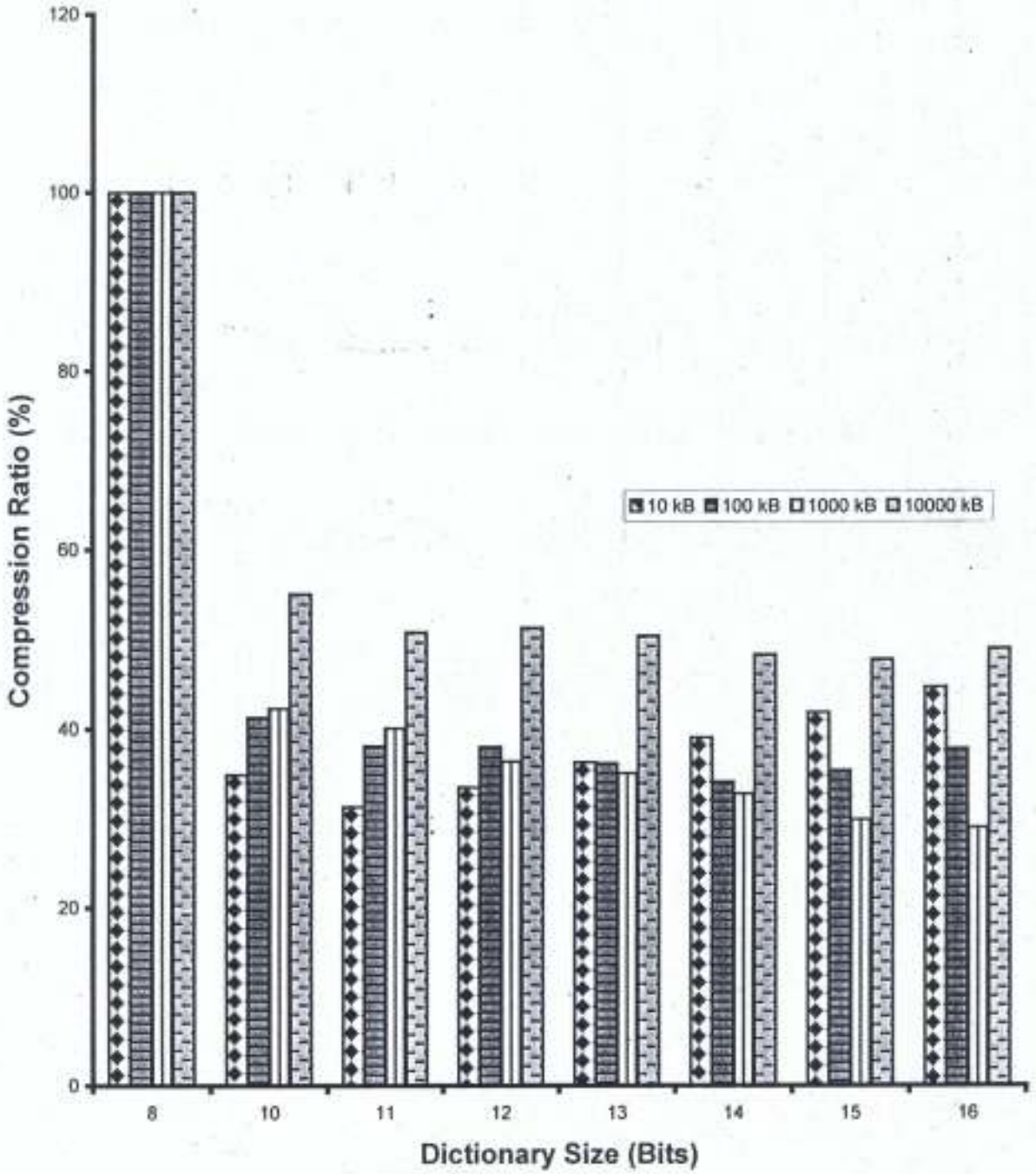

Fig. 4.9 Compresion Ratios for Different Dictionary Sizes Using The Input Source Data 3.

Table 4.15 Compression Ratios For Different Dictionary Sizes Using Source Data 4 as the Input

Dictionary Size (No of Bits). / Size of Input Data. (kB)	8	10	11	12	13	14	15	16
10	100	65.9	59.4	57.4	61.3	66.0	70.75	75.5
100	100	66.3	56.8	49.8	44.7	41.6	43.2	46.1
1000	100	64.0	54.9	47.7	41.3	35.8	30.7	26.96
10000	100	66.2	56.3	48.8	42.1	35.5	29.8	25.5

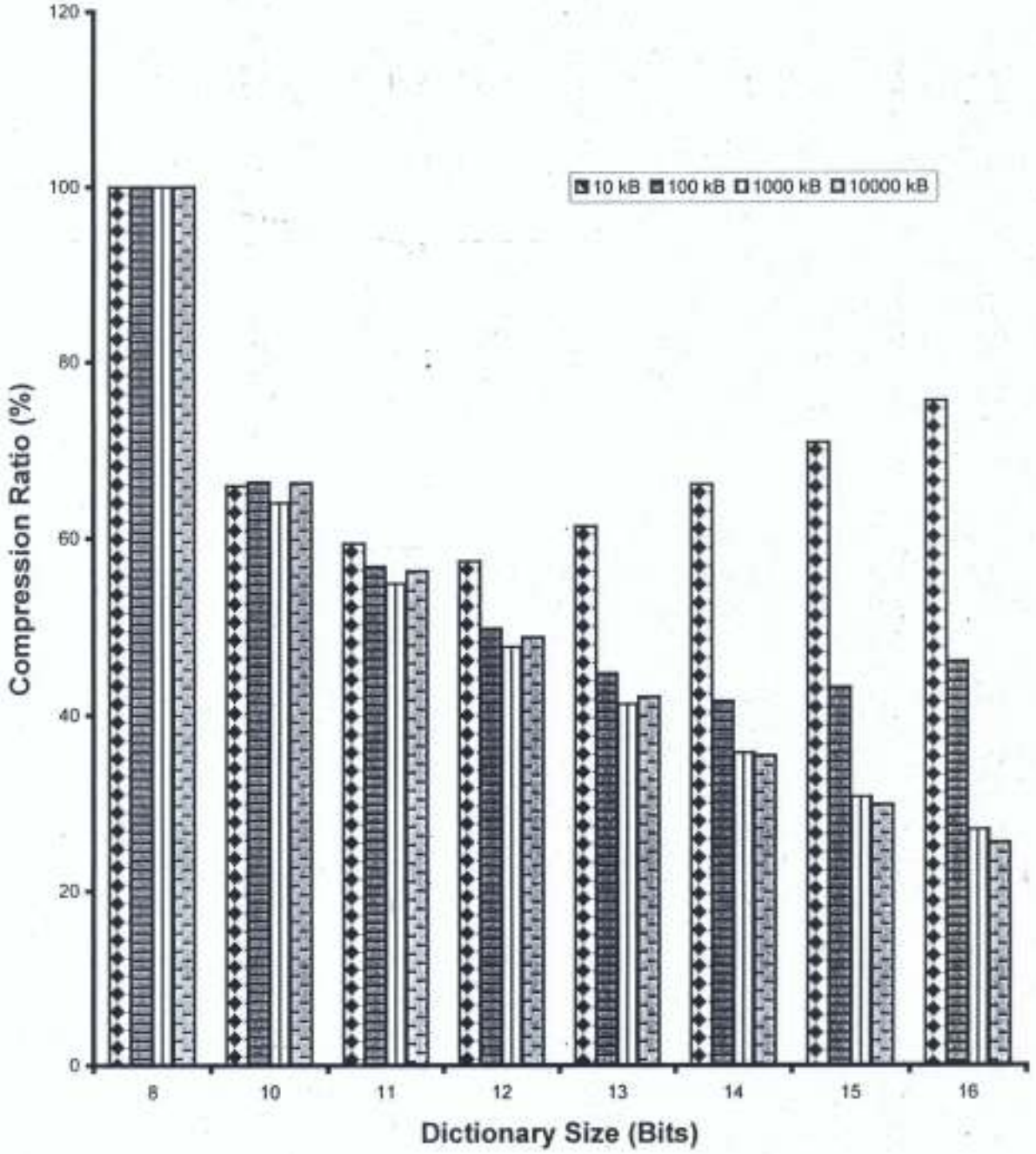

Fig. 4.10 Compression Ratios for Different Dictionary Sizes Using The Input Source Data 4.

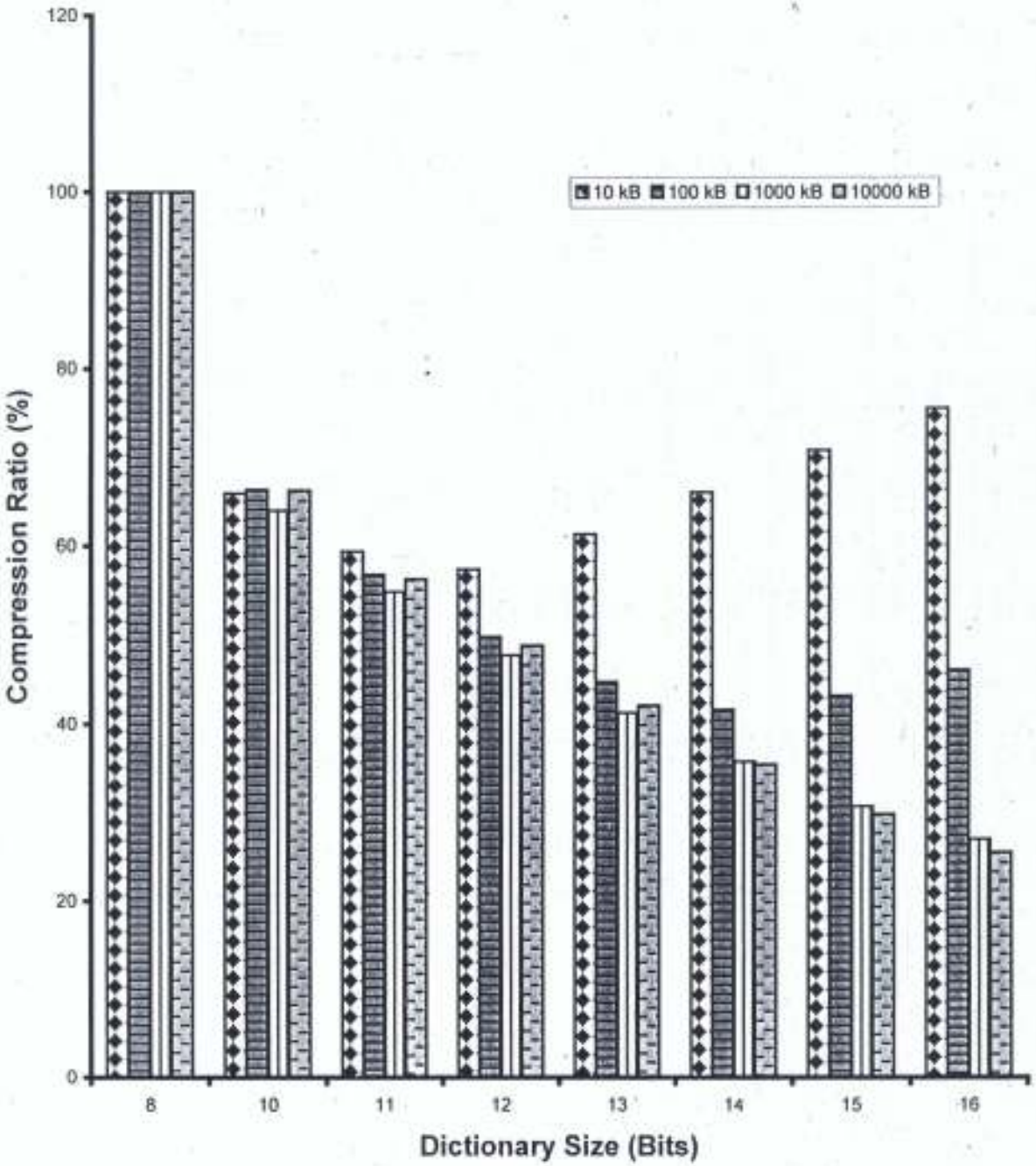

Fig. 4.10 Compression Ratios for Different Dictionary Sizes Using The Input Source Data 4.

Table 4.16 Compression Ratios For Different Dictionary Sizes
Using Source Data 5 as the Input

Dictionary Size (No of Bits). / Size of Input Data. (kB)	8	10	11	12	13	14	15	16
10	100	63.4	52.5	51.8	56.1	60.4	64.8	69.0
100	100	60.5	53.7	50.1	45.7	42.5	44.5	47.5
1000	100	60.3	53.4	49.0	43.4	37.5	34.2	31.4
10000	100	60.3	53.4	49.0	43.2	37.0	33.2	29.8

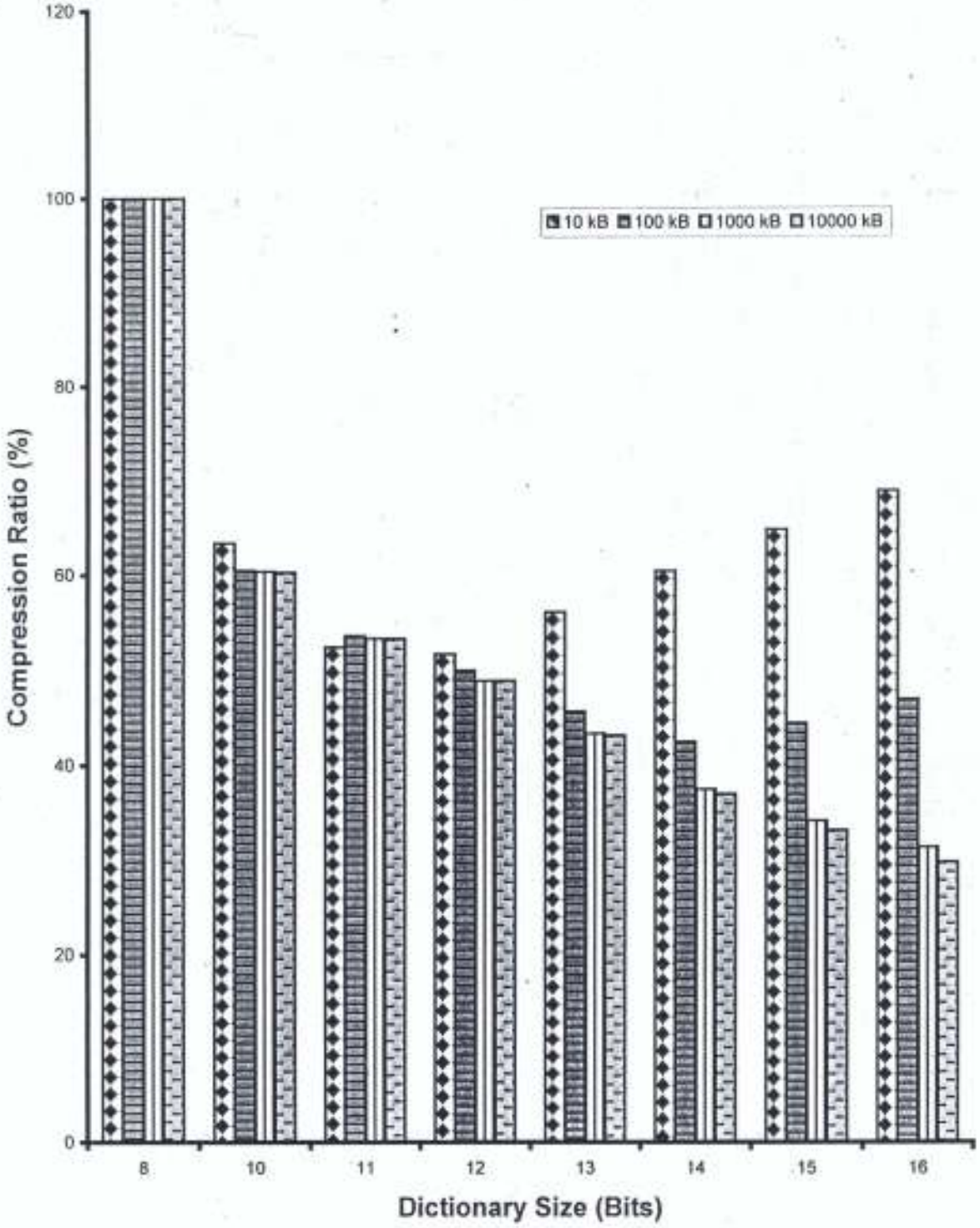

Fig. 4.11 Compression Ratios for Different Dictionary Sizes Using The Input Source Data 5.

Table 4.17 Compression Ratios Average for Different Dictionary Sizes

Dictionary Size (No of Bits) / Size of Input Data. (kB)	8	10	11	12	13	14	15	16
10	45.86	40.8	41.52	44.76	48.2	51.7	55.12	45.86
100	49.7	44.28	41.68	39.38	37.06	38.6	41.12	49.7
1000	52.76	47.7	44.32	41.24	37.44	35.26	33.3	52.76
10000	56.74	50.86	48.72	45.68	41.08	38.96	38.06	56.74

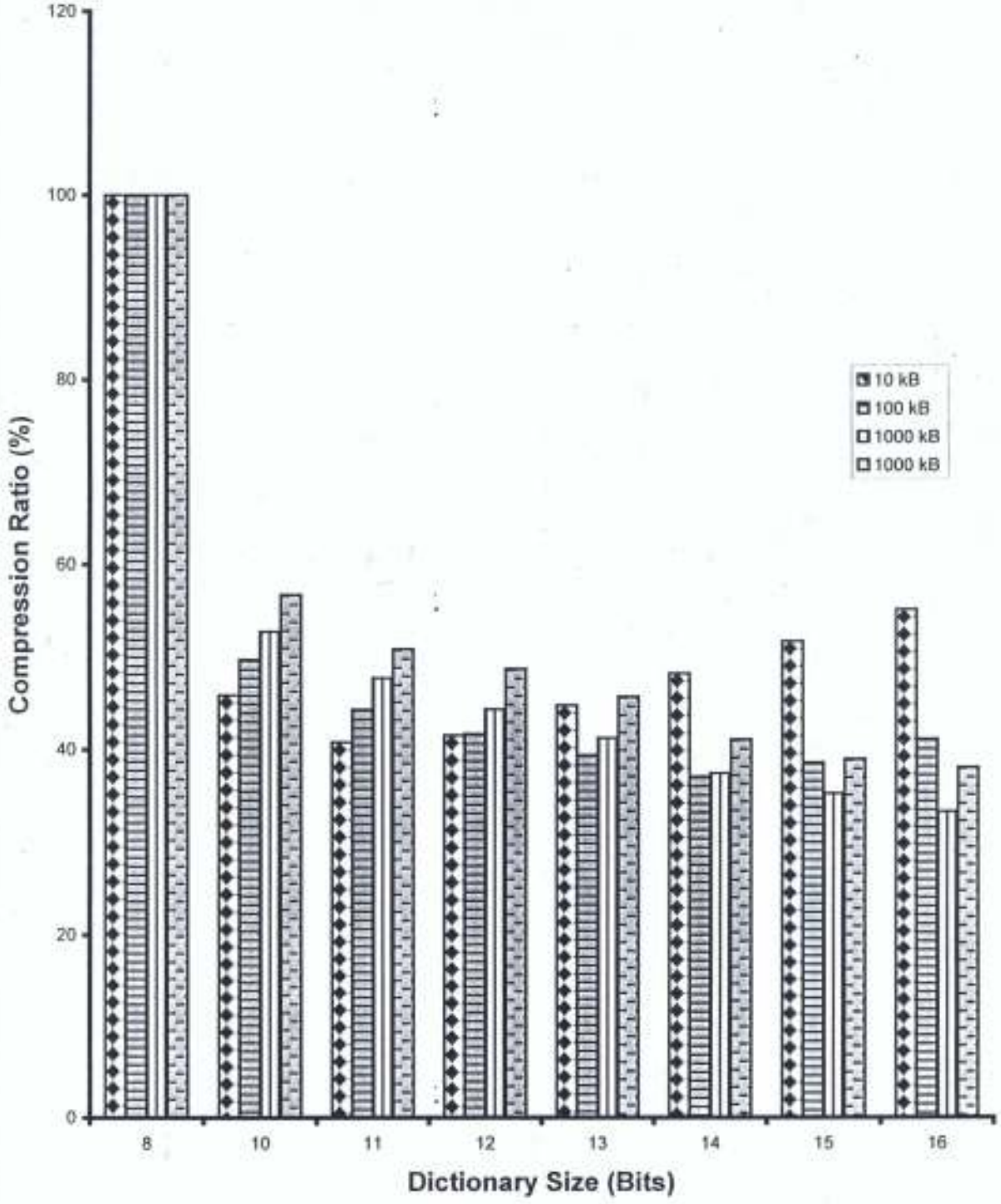

Fig. 4.12 (a) Compression Ratio Average For Different Dictionary Sizes

Table 4.18 Compression Ratio Average for Different Dictionary Sizes Using the Mean Value of the Four Data Sizes

Dictionary Size (No of Bits)	8	10	11	12	13	14	15	16
Compression Ratio (%)	100	51.27	45.91	44.06	42.77	40.95	41.13	41.9

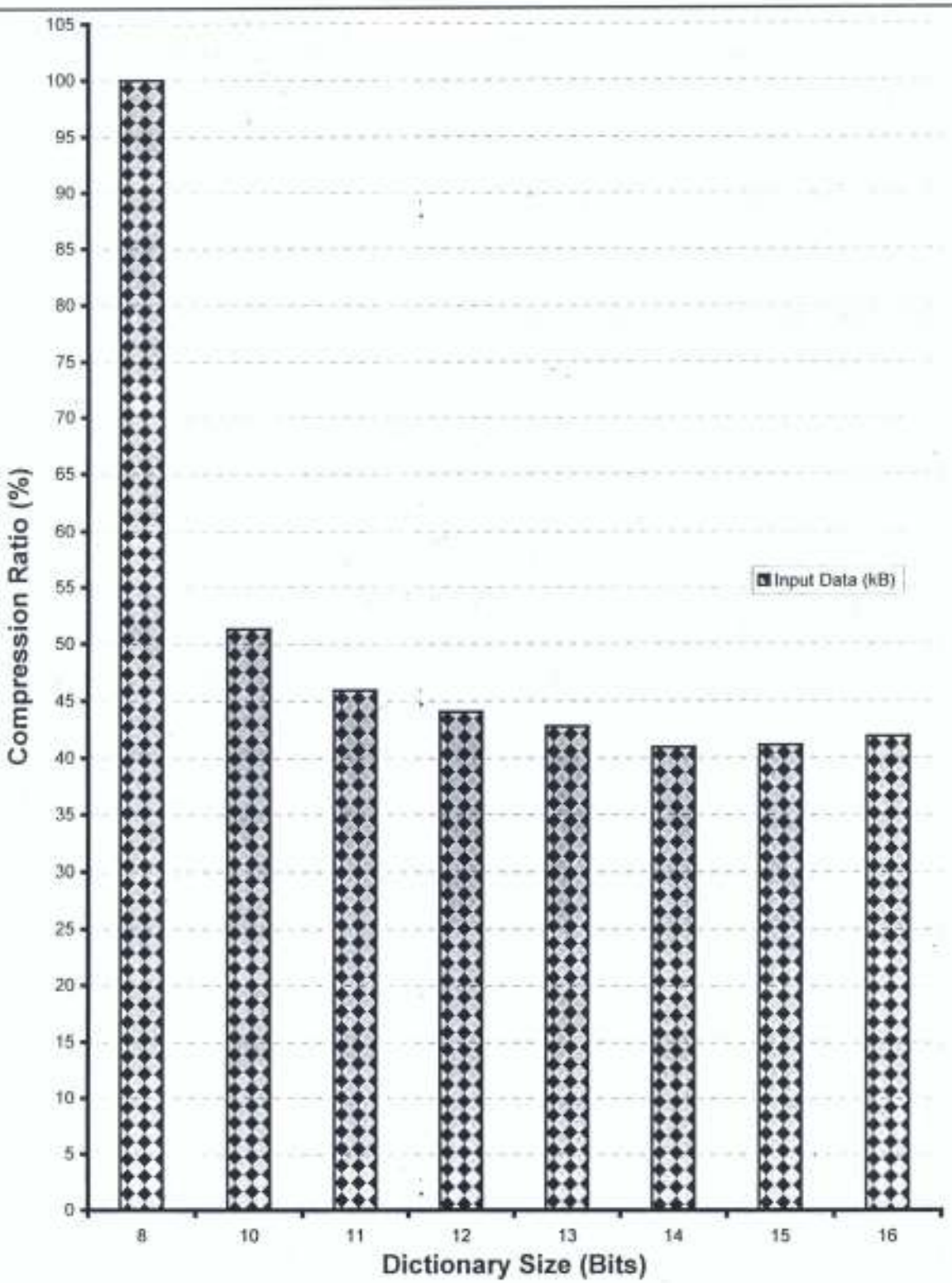

Fig. 4.12 (b) Compression Ratio Average for Different Dictionary Sizes Using the Mean Value of the Four Data Sizes.

Table 4.19 compares the compression ratios obtained from input data a, b, c, d, and e using the developed LZW compression program and the WinZip compression program.

Table 4.19 Compression Ratios Obtained From Various Input Data Using The Developed LZW Program And The WinZip Compression Program

Input Data	LZW Compression Program (%)	WinZip Compression Program (%)
Data a	32.5	21.3
Data b	32.7	31.07
Data c	31.6	14.3
Data d	39	13.7
Data e	29.8	12.7

CHAPTER FIVE

CONCLUSION AND SUGESTIONS FOR FURTHER RESEARCH WORK

5.1 CONCLUSION

The developed data compression application program worked well. For any dictionary size, compression ratio varies with the size of input data and the redundancy in the data. The highest compression ratio achieved from the various input data was 26% while the least compression ratio achieved was 71%. A compression ratio of about 40% was achieved for most input data. In the average, a dictionary size of 14 bits compressed best for different sizes of input data and it is therefore recommended. The results show that more data can be stored on a given storage medium using the developed data compression program. The program was written in C^{++} that enhanced a very small run-time.

5.2 SUGESTIONS FOR FURTHER RESEARCH WORK

Having realized the stated objectives of this research, the following suggestions were made:

(1) Further research can be carried out on the effect of variation in the lengths of the dictionary entries on the compression ratio.

(2) Further research can be carried out on data compression for transmission.

(3) Group research work should be encouraged especially in areas such as speech and image compression, which require intensive programming.

REFERENCES

Abramson, N. (1963). Information Theory and Coding. McGraw-Hill, New York.

Brigham, E. O. (1974). The Fast Fourier Transform. Englewood Cliffs, New Jersy.

Campos, A .S. (2000). Compression Programming. http://www.arturocampos.com

Charles J. S. (1985). Macmillan Dictionary of Data Communications. Second Edition, Macmillan Press, London.

Faller, N. (1973). An Adaptive System for Data Compression: Record of the 7th Asilomar Conf. on Circuits, Systems and Computers. Pacific Grove, Ca., Pp. 593-597.

Feller, W. (1968). An Introduction to Probabilistic Theory and Its Application. Third Edition, John Wiley and Sons Inc. Pp. 42

Gallager, R. G. (1978). Variations on a Theme by Huffman. IEEE Trans. Inform. Theory 24, 6 (Nov.), Pp. 668-674.

Gersho, A. and Gray, R. M. (1992). Vector Ouantization and Signal compression. Kluwer, Boston.

Habibi, A. (1971). Comparison of Nth-Order DPCM Encoder With Linear Transformation and Block Quantization Techniques. IEEE Transactions on Communications. Vol. 19, Pp. 948-956.

Herbert, S. (1992). Teach Yourself C^{++}. McGraw-Hill

Huffman D. A. (1952). A Technique for the Construction of Minimum Redundancy. Proceeding of IRE. Vol. 40, Pp. 1098-1101.

Jain, A. K. (1989). Fundamentals of Digital Image Processing. Englewood Cliffs, New Jersey.

Jer, M. J. ; Pei-Yin, C. (1999). A Fast and Efficient Lossless Data-Compression Method. IEEE Transactions on Communications. Vol. 47, Pp. 1278 – 1283.

Knuth, D. E. (1985). Dynamic Huffman Coding. J. Algorithms 6, 2 (June), Pp.163-180.

Madisetti, V. K. (1995). Signal Compression.

http://www.users.ece.gatech.edu/~vkm/nii/node36.html.

Mark, N. (1991). Arithmetic Coding and Statistical Modeling. Doctor Dobb's Journal, February, Pp. 16-29.

Mark, N. (1989). LZW Data Compression. Doctor Dobb's Journal, October, Pp. 29-37.

Pasco, R. (1976). Source Coding Algorithms for Fast Data Compression. Ph. D. dissertation, Dept. of Electrical Engineering, Stanford Univ., Stanford, Calif.

Phamdo, N. (2000). Lossless Data Compression, http://www.Data-Compression.com

Phelps, M. F. (1995). Dictionary of Computer Words. Revised Edition, Houghton Mifflin Company (New York).

Rissanen, J. J. 1976. Generalized Kraft Inequality and Arithmetic Coding. IBM J. Res. Dev. 20 (May), Pp. 198-203.

Roden, M. S. (1988). "Digital Communication Systems Design" Prentice Hall International, Inc (USA), Pp. 154-174.

Rubin, F. (1976). Experiments in Text File Compression. Commun. ACM 19, 11 (Nov.), Pp. 617-623.

Ruth, S. S. ; Kreutzer, P. J. (1972). Data Compression for Large Business Files. Datamation 18, 9 (Sept.), Pp. 62-66.

Solomon, D. (1998). Data Compression, The Complete Reference. Springer-Verlag, New York.

Steven, W. S. (1997). The Scientist and Engineer's Guide to Digital Signal Processing. Second Edition, California Technical Publishing, Pp. 481-494

Terry, A. W (1984). A Technique for High Performance Data Compression. IEEE Computer, Vol. 17, No. 6, Pp. 8-19.

Vitter, J. S. (1987). Design and Analysis of Dynamic Huffman Codes. J. ACM 34, 4 (Oct.), Pp. 825-845.

Ziv, J. and Lempel, A. (1977). A Universal Algorithm for Sequential Data Compression. . IEEE Transactions on Information Theory, Vol. 23, Pp. 337-342.

```
//Data Compression Program Written in C++
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define BITS 14
#define HASHING_SHIFT BITS-8
#define MAX_VALUE (1 << BITS)-1
#define MAX_CODE MAX_VALUE - 1
#if BITS == 8
#define TABLE_SIZE 299
#elif BITS == 9
#define TABLE_SIZE 599
#elif BITS == 10
#define TABLE_SIZE 1129
#elif BITS == 11
#define TABLE_SIZE 2251
#elif BITS == 12
#define TABLE_SIZE 5021
#elif BITS == 13
#define TABLE_SIZE 9029
#elif BITS == 14
#define TABLE_SIZE 18041
#elif BITS == 15
#define TABLE_SIZE 37061
#else BITS == 16
#define TABLE_SIZE 77971
#endif
```

```
long double  zz,xx,xz;
int *code_value,Process;          // This is the code value array
unsigned int *prefix_code;        // This array holds the prefix codes
unsigned char *append_character;  // This array holds the appended chars
unsigned char decode_stack[4000]; // This array holds the decoded string


// This program gets a file name.It compresses the file, placing
// its output in aother file.
 void output_code(ofstream output_file,unsigned int code);
  input_code(ifstream input_file);
 char *decode_string( unsigned char *buffer,unsigned int code);
  unsigned char ch;
void main()
{
ifstream input_file;
ofstream output_file;
ofstream lzw_fileo;
ifstream lzw_filei;
 int m,ff;
char input_file_name[40],compress_file_name[40];
char decompress_file_name[40],input_file_namee[38];
char decompress_file_nam[39];
void compress(ifstream input_file,ofstream lzw_fileo);
 void  decompress(ifstream lzw_filei,ofstream output_file);
 cout<<" Which Process? Compression or Decompression?"<<endl;
 cout<<" "<<endl;
         cout<<" If  Compression, type: 1"<<endl;
         cout<<" "<<endl;
         cout<<" If Decompression,type: 2"<<endl;
         cout<<" "<<endl;
         cout<<"                  : ";
         cin>> Process;
         switch(Process){
         case 1:
//  These three buffers are needed for the compression phase.

 code_value= new int[TABLE_SIZE];
```

```
prefix_code= new unsigned int[TABLE_SIZE];
append_character= new unsigned char[TABLE_SIZE];
if (! code_value || ! prefix_code || ! append_character)
{
 cout<<"Fatal error allocating table space(for malloc) !"<<endl;
 exit(1);
}

 cout<<" Input file name?    : ";
 cin>>input_file_name;
      cout<<" "<<endl;

      strcpy(compress_file_name,input_file_name);
      strcat(compress_file_name,"c");
      cout<<" The compressed file is named:     "<<compress_file_name<<endl;

      input_file.open(input_file_name, ios::binary);
      lzw_fileo.open(compress_file_name, ios::binary);

       if (!(input_file.good()))

      {
       cout<<"Fatal error opening files(1)"<<endl;
       exit(1);
      }
                          else
       if(!(lzw_fileo.good()))
       {cout<<"Fatal error opening files(2)"<<endl;
       exit(1);
       }

// Compress the file.
 compress(input_file,lzw_fileo);
 input_file.close();
 lzw_fileo.close();
 delete code_value;
 delete prefix_code;
 delete append_character;
```

```
// The following equation calculates the compression ratio
   xz=((zz*BITS*100)/(xx*8));
   cout<<" The Compression Ratio is    : " <<xz<<" %"<<endl;
   cout<<"  "<<endl;
   break;


// Open the file for the decompression.
        case 2:
        cout<<" Input file name?    : ";
        cin>>input_file_namee;
        cout<<" "<<endl;
        ff=strlen(input_file_namee);
        m=(ff-1);
        strncpy(decompress_file_nam,input_file_namee,m);
        strncpy(decompress_file_name,decompress_file_nam,m);
        cout<<" The decompressed file is named : "<<decompress_file_name<<endl;

// These two buffers are needed for the compression phase.
 lzw_filei.open(input_file_namee, ios::binary);
 output_file.open(decompress_file_name, ios::binary);
 if (!(lzw_filei.good()))
 { cout<<"Fatal error opening files (2a)."<<endl;}
 else
  if (!(output_file.good()))
  { cout<<"Fatal error opening files (2b)."<<endl;
                                }

// Decompressing the file.
 prefix_code= new unsigned int[TABLE_SIZE];
 append_character = new unsigned char[TABLE_SIZE];
 decompress(lzw_filei,output_file);
 lzw_filei.close();
 output_file.close();
 delete prefix_code;
 delete append_character;
 break;
```

```
                    }
}

// This is the compression routine.
void compress(ifstream input,ofstream output)
{
unsigned int next_code;
unsigned int character;
unsigned int string_code;
unsigned int index;
int i;
unsigned int find_match(unsigned int hash_prefix,unsigned int hash_character);
 next_code=256;
 for (i=0;i<TABLE_SIZE;i++)
  code_value[i]=-1;
 i=0;
 zz=2;
 xx=0;
  input.get(ch);
 string_code=ch;
  while (! input.eof())
 {
  input.get(ch);
 character=ch;
  xx++;
  if (++i==1000)
  {
   i=0;
      }
  index=find_match(string_code,character);
  if (code_value[index] != -1)
   string_code=code_value[index];
  else
  {
   if (next_code <= MAX_CODE)
   {
    code_value[index]=next_code++;
    prefix_code[index]=string_code;
```

```
      append_character[index]=character;
     }
     output_code(output,string_code);
                                zz++;
     string_code=character;
    }
   }

// End of the main loop.
  output_code(output,MAX_VALUE);
  output_code(output,0);
  cout<<endl;
}
// This is the hashing routine.
unsigned int find_match(unsigned int hash_prefix,unsigned int hash_character)
{
int index;
int offset;
  index = (hash_character << HASHING_SHIFT) ^ hash_prefix;
  if (index == 0)
   offset = 1;
  else
   offset = TABLE_SIZE - index;
  while (1)
  {
   if (code_value[index] == -1)
    return(index);
   if (prefix_code[index] == hash_prefix &&
      append_character[index] == hash_character)
    return(index);
   index -= offset;
   if (index < 0)
    index += TABLE_SIZE;
   continue ;
    }
    }
// This is the decompression routine.  It takes a compressed file
// and decompresses placing the output in another file.
```

```
void decompress(ifstream input,ofstream output)
{
unsigned int next_code;
unsigned int new_code;
unsigned int old_code;
unsigned int character;
int counter;
 unsigned char *string;
  char *decode_string( unsigned char *buffer,unsigned int code);

  next_code=256;          // This is the next available code to define
  counter=0;              // Counter is used as a pacifier.

  old_code=input_code(input);  // Read in the first code, initialize the
  character=old_code;          // character variable, and send the first

  output.put((unsigned char)old_code);

   while ((new_code=input_code(input)) != (MAX_VALUE))

  {
   // This code checks for the special
  //STRING+CHARACTER+STRING+CHARACTER+STRING
  // case which generates an undefined code.  It handles it by decoding
  // the last code, and adding a single character to the end of the decode string.

    if (new_code>=next_code)
    {
      *decode_stack=character;
   string=(unsigned char *)decode_string(decode_stack+1,old_code);
    }
// Otherwise we do a straight decode of the new code.
//
    else
      string=(unsigned char *)decode_string(decode_stack,new_code);
// Now we output the decoded string in reverse order.
    character=*string;
    while (string >= decode_stack)
```

```
    output.put(*string-- );
 // Finally, if possible, add a new code to the string table.

   if (next_code <= MAX_CODE)
   {
    prefix_code[next_code]=old_code;
    append_character[next_code]=character;
    next_code++;
   }
   old_code=new_code;
  }
 cout<<endl;
}
// This routine simply decodes a string from the string table, storing
// it in a buffer.  The buffer can then be output in reverse order by
// the decompression  program.
 char *decode_string(  unsigned char *buffer,unsigned int code)
{
int i;

 i=0;
 while (code > 255)
 {
  *buffer++ = append_character[code];
  code=prefix_code[code];
  if (i++>=4094)
  {
   cout<<"Fatal error during code expansion."<<endl;

  }

 }
 *buffer=code;
 return((char *)buffer);
}
// The following two routines serve as buffers. They are used to
// output variable length codes.
input_code(ifstream input)
```

```
{
unsigned int return_value;
static int input_bit_count=0;
static unsigned long input_bit_buffer=0L;
 while (input_bit_count <= 24)
                                {

  input.get(ch);

   input_bit_buffer |=
      ((unsigned long)(ch) << (24-input_bit_count));
   input_bit_count += 8;
 }
 return_value=input_bit_buffer >> (32-BITS);
 input_bit_buffer <<= BITS;
 input_bit_count -= BITS;
 return(return_value);
 }
void output_code(ofstream output,unsigned int code)
{
static int output_bit_count=0;
static unsigned long output_bit_buffer=0L;
 output_bit_buffer |= (unsigned long) code << (32-BITS-output_bit_count);
 output_bit_count += BITS;
 while (output_bit_count >= 8)
 {
 ch=(output_bit_buffer>>24);
  output.put(ch);
     output_bit_buffer <<= 8;
  output_bit_count -= 8;

 }

}
```

Appendix B

Standard ASCII Character Set - Control code and Space Characters

Decimal Value	Hexadecimal value	Character	Decimal Value	Hexodecimal value	Character
0	00	NUL Null	18	12	DC2 Device control 2
1	01	SOH Start of heading	19	13	DC3 Device control 3
2	02	STX Start of text	20	14	DC4 Device control 4
3	03	ETX End of text	21	15	NAK Negative acknowledgement
4	04	EOT End of transmission	22	16	SYN Synchronous
5	05	ENQ enquiry			idle
6	06	ACK Acknowledge	23	17	ETB End transmission
7	07	BEL Audible			block
8	08	BS Backspace	24	18	CAN Cancel
9	09	HT Horizontal tab	25	19	EM End of medium
10	0A	LF line feed	26	1A	SUB Substitute
11	0B	VT Vertical	27	1B	ESC Escape
12	OC	FF Form feed	28	1C	FS File separator
13	OD	CR Carriage turn	29	ID	GS Group separator
14	OE	SO Shift out	30	1E	RS Record separator
15	OF	SI Shift in	31	IF	US Unit separator
16	10	DLE Data lin escape	32	20	SP Blank space
17	11	DC1 Device control I			character

Standard ASCI Character Set-Alphanumeric Characters

Decimal Value	Hexadecimal Value	Character
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	/
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	65	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	]
125	7D	}
126	7E	~