DATA COMPRESSION TECHNIQUES

A THESIS

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF

MASTER OF ENGINEERING
IN
COMPUTER SCIENCE AND ENGINEERING

TO

THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY
(DEEMED UNIVERSITY)
PATIALA - 147001

SUPERVISORS

Sh. C. Ramakrishna
Lecturer, Department of Computer Science & Engineering,
Technical Teachers' Training Institute,
Sector 26, Chandigarh.

Sh. Himanshu Aggarwal
Lecturer, Department of Computer Science & Engineering,
Punjabi University,
Patiala.

SUBMITTED BY

Adarsh Kumar Bhaskar
Regn. No. 132 / 97(R) / 1

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
TECHNICAL TEACHERS' TRAINING INSTITUTE
SECTOR 26, CHANDIGARH - 160019.
FEBRUARY 2001

CERTIFICATE

Certified that the dissertation entitled "Data Compression Techniques" is being submitted by Adarsh Kumar Bhaskar in partial fulfillment for the award of the degree of Master of Engineering in Computer Science and Engineering of Technical Teachers' Training Institute is a record of his own work carried out by him under our guidance and supervision.

Guides

(C. Ramakrishna)
Lecturer, Department of Computer Science & Engineering,
Technical Teachers' Training Institute,
Sector 26, Chandigarh.
(Internal)

(Himanshu Aggarwal)
Lecturer, Department of Computer Science & Engineering,
Punjabi University,
Patiala.
(External)

Head,
Department of Computer Science & Engineering

(Sh. V.P. Puri)
Technical Teachers' Training Institute,
Sector 26, Chandigarh.

विभागाध्यक्ष, कम्प्यूटर विज्ञान,
तकनीकी शिक्षक प्रशिक्षण संस्थान,
सेक्टर-26 चण्डीगढ़-160019
Head, Dept. of Computer Science,
Technical Teachers Training Institute,
Sector-26, Chandigarh-160019

ACKNOWLEDGEMENTS

I would like to thank my guides Sh. C. Ramakrishna, Lecturer in Computer Science & Engineering Department, Technical Teachers' Training Institute, Chandigarh and Sh. Himanshu Aggarwal, Lecturer in Computer Science & Engineering Department, Punjabi University, Patiala, under whose able supervision, guidance and motivation, I have undertaken this dissertation. They spent painstaking hours in the preparation of this dissertation and pooled invaluable suggestions to improve the quality of this dissertation. My special thanks are also due to Dr. Renu Vig, Assistant Professor in Computer Science & Engineering Department, Technical Teachers Training Institute, Chandigarh for his help and guidance.

Finally, I acknowledge my gratitude to Sh. V.P. Puri, Head, Department of Computer Science, Technical Teachers Training Institute, Chandigarh for allowing me to take this dissertation.

Adarsh Kumar Bhaskar

Adarsh Kumar Bhaskar

TABLE OF CONTENTS

ABSTRACT

With the growth of multimedia and Internet, compression techniques have become the thrust area in the fields of computers. Popularity of multimedia has led to integration of various types of computer data. Multimedia combines many data types like text, graphics, still images, animation, audio and video data. Many different data compression techniques currently exist for the compression of different types of data. Text compression requires regeneration for the original data from the compressed form in exactly the same form. Similarly, graphical data may also require the data to be same after decoding. Graphics, images, audio and video data require huge amount of storage. This not only require large storage capacity, but, the uncompressed form of these data types require higher band width for multimedia communication through computers or else the rate of data transfer is low making networks including Internet sluggish. Compression is one of the tools for better utilization of storage devices, resulting in saving of storage space. Current thesis describes various lossless and lossy data compression techniques, which includes Shannon-Fano coding, Huffman-coding algorithm, arithmetic coding, Runlength algorithm, LZ77 and LZW dictionary-based compression techniques and their implementation. Thesis also describes about the audio and image compression techniques.

Compression techniques may use more than one technique. For this, one method is applied first and encoded data so obtained undergoes another compression cycle using same or some other compression technique. With this higher compression rates can be achieved.

CHAPTER 1
INTRODUCTION

Data compression reduces the number of bits used to store or transmit information. Information is stored in forms of files. Data in these files consist of text consisting of manuscripts, program memos and other readable files or binary data including database files, executable files and spreadsheet data or graphics files stored in raw screen dump format, and many other type of information. Nearly all data contains some redundant information. The objective of the data compression algorithm is to transform redundant data into a form that is smaller but still retains the same information.

Simple compression percentage formula is given by:

(1 - (compressed_size / raw_size)) * 100

This means that a file that does not change at all when compressed will have a compression ratio of zero percent and a file compressed down to one third of its original size will have compression ratio of 67 percent. A file that goes through compression and comes out larger will show a negative compression ratio. Compression through hardware is generally expensive, but compression through software is in sense free.

Uncompressed graphics, audio and video data require considerable storage capacity and also data transfer of uncompressed video data over digital networks requires very high bandwidth.

The exploding use of World-Wide Web on the Internet has increased demand for speedy transfer of graphics and images. Compressing files before transmitting them saves telecommunications bandwidth. This requires compatible compression software on both receiving and sending end. Convenient method of conserving bandwidth is to build data compression directly into modem. International telecommunication industry has adopted compression algorithm used by modem manufacturers.

1.1 ORGANISATION OF THESIS

The organization of this thesis roughly parallels the growth of data compression, starting around 1950. This chapter gives the fundamentals of data compression by discussing birth of information theory, series of concepts, terms and theories used over rest of thesis. It also discusses difference between modeling and coding which are the two components of data compression.

Chapter 2 discusses variable length bit coding -- Shannon-Fano coding and Huffman coding. These are still in wide use today. In standard Huffman coding, the compression program has to pass a complete copy of the Huffman coding statistics to the expansion program. As the order of model increments, compression ratio improves but compression collects more statistics, it takes up more space and work against compression. This chapter also discusses adaptive Huffman coding. This is recent innovation due to CPU and memory requirements. It greatly expands the horizons of Huffman coding and hence improved compression ratios. Huffman coding use integral number of bits for each codes which is slightly less than optimal.

Chapter 3 discusses more recent innovation, arithmetic coding that uses fractional number of bits per code and it shows how to integrate an arithmetic codes with statistical model. This chapter also discusses statistical modeling. It is necessary to have a statistical model to drive the coder. It discusses powerful models using limited memory resources.

Chapter 4 provides an overview of Dictionary compression methods, which replace string of characters with single codes. These are de-facto standard for general-purpose data compression on small computers. It discusses recent adaptations to LZ77 compression used in popular archiving programs such as PKZIP. LZ77 is sliding dictionary method of data compression proposed by Ziv and Lempel in 1977. This chapter also takes detailed look at LZW compression used in UNIX compress program and MS-DOS ARC program.

Chapter 5 provides fundamental audio concepts and shows how lossy methods can be used on digitized sound data like linear predictive coding and

adaptive PCM. These methods are capable of achieving higher compression ratios.

Chapter 6 discusses industry standard JPEG for compressing graphical images. This chapter also gives detailed look at fractal compression techniques, for achieving maximum compression of graphical data, MPEG for moving pictures i.e. video compression and use of neural network for image compression.

1.2 FUNDAMENTALS OF DATA COMPRESSION

1.2.1 DATA COMPRESSION TECHNIQUE

Data compression technique consists of taking stream of symbols and transforming them into codes. It consists of two component sets -- modeling and coding i.e. **Data Compression = Modeling + Coding**

This can be pictorially represented as following:

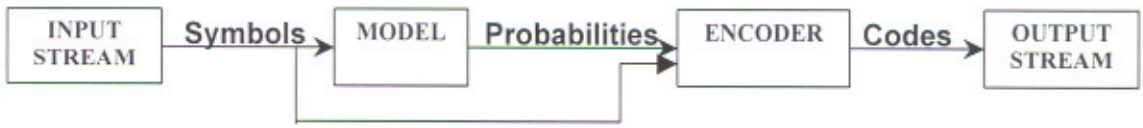


Fig. 1.1: Data compression technique.

The decision to output a certain code for a certain symbol or set of symbols is based on a model. Model is simply a collection of data and rules used to process input symbols and determine which code(s) to output. A program uses the model to accurately define the probabilities for each symbol and the coder to produce an appropriate code based on these probabilities example, Huffman coding and Run-length encoding are just coding methods used to compress data. There are countless ways to model data, all of which can use the same coding process to produce their output. A simple Huffman coding would use a model that gave the raw probability of each symbol occurring anywhere in the input stream and another program may use the model that calculate probabilities

based on the last 10 symbols in the input stream. Both these programs may use Huffman coding to produce their output.

Following parameters must be considered for the compression-decompression cycle:

- The quality of the coded and later on decoded data should be as good as possible
- The implementation should be cost-effective
- The complexity of the technique used should be minimal and
- The processing of the algorithm must not exceed certain time spans.

1.2.2 TYPES OF DATA COMPRESSION

Information in computers is stored in form of files. Data in these files is broadly divided into two categories with regard to compression. Firstly, text files consisting of word-processing files and programs, and binary files consisting of database records, spreadsheets and executable files. Secondly, graphics images and digitized voice. Depending upon the above said categories of files, data compression techniques are divided into two major families - lossless data compression and lossy data compression.

(1) **Lossless data compression: -** These techniques guarantee to generate exact duplicate of the input stream after a compress / expand cycle. These are used when storing database records, spreadsheets, word processing files and executable files where loss of even a single bit is not tolerable. Chapter 2 to chapter 8 discusses these techniques.

(2) **Lossy data compression: -** These techniques concede certain loss of accuracy after a compress / expand cycle. These are used for graphics images and digitized voice because these represent analog phenomena, which are not perfect and hence non-matching of input and output, is acceptable. The lossy compression techniques can be adjusted to different quality levels, gaining higher accuracy in exchange for less effective compression.

1.2.3 INFORMATION THEORY

Information theory is a branch of mathematics that has its genesis in late 1940s with the work of Claude Shannon at Bell Labs. Data compression enters into the field of Information theory because it concerns with redundancy. Redundant information takes extra bits to encode and can be get rid of, thus, reducing size of message. It uses the term entropy as a measure of information in a message. Entropy of symbol is defined as the negative logarithm of its probability. In terms of bits logarithm to base 2 is used for determining information content of a symbol given by,

$$\textbf{Number of bits} = -\text{Log}_2(\textbf{probability})$$

The entropy of the entire message is simply the sum of entropy of all individual symbols. Higher the entropy of message the more information it contains. Entropy fits with data compression in its determination of how many bits of information are actually present in a message. Probability changes with model. A model tracks the probability based on what symbol appeared previously in the input stream. Order of model determines how many previous symbols are taken into account, for example, order-0 model won't look at previous characters and order-1 model looks at one previous character. We select the model that predict symbols with high probabilities because a symbol that has a high probability has low information content and will need fewer bits to encode. Once model produces probabilities, next step is to encode the symbols using an appropriate number of bits.

1.2.4 CODING

By using EBCDIC or ASCII coding we aren't very close to optimum method as every character is encoded using same number of bits. Shannon-Fano coding and Huffman coding are two different ways of generating variable length codes when a probability table for a given set of symbols is given. Huffman coding achieves minimum amount of redundancy possible in a fixed set of variable length codes. It does not mean optimal coding. Both use integral number of bits in each code. But these are easy to implement and economical. These dominated the world till early 1980s. As cost of CPU cycles goes down,

arithmetic coding a viable successor to Huffman coding emerge. Arithmetic coding is more complex in concept and implementation. It does not produce single code for each symbol, instead, it produces a code for an entire message. Each symbol added to message incrementally modifies the output code. It requires more CPU power but result in more accurate coding.

1.2.5 MODELING

Model feeds probabilities. Lossless data compression uses one of the two different types of modeling -- statistical or dictionary based. Statistical modeling reads in and encodes a single symbol at time using the probability of that character's appearance. Dictionary based modeling uses a single code to replace strings of symbols.

(a) Statistical Modeling: - Statistical models generally encode a single symbol at a time i.e. read it, calculate probability and then output a single code. In its simplest form use a static table of probabilities. Once representative blocks of data are analyzed, table of character-frequency count is prepared to build Huffman coding / decoding trees. Universal static model has limitations -- ratio degrades if input stream do not match with previously accumulated statistics. Next enhancement is to build statistics table for every unique input stream, but this costs additional overhead, as table has to be passed to the decoder. By compression research, adaptive models have resulted where data is not scanned once before coding but statistics are continually modified as new characters are read-in and coded. Data compression and decompression using adaptive methods can be pictorially represented as following :

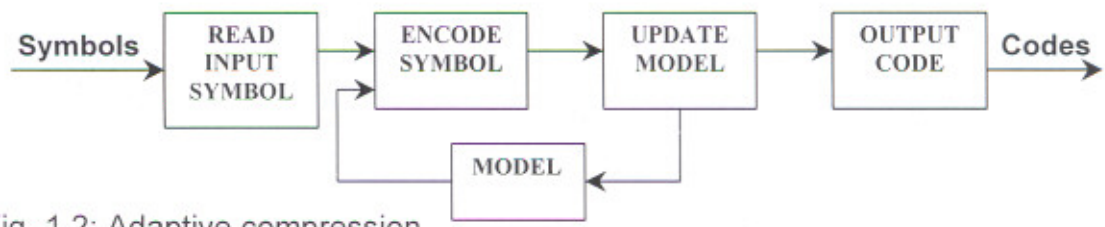


Fig. 1.2: Adaptive compression.

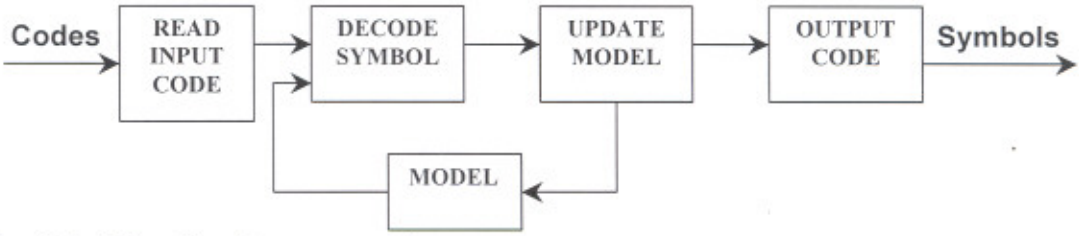


Fig. 1.3: Adaptive decompression.

Update model works exactly same for compression and decompression. After each character (or group of characters) is read-in and encoded or decoded, the model is updated to take into account the recent symbol or group of symbols. So, they start knowing essentially nothing about data and thus doesn't do very good job of compression at start. But advantage is the ability to adopt to local conditions. It may weight the most recent data higher than old data.

(b) Dictionary Schemes: - Dictionary based compression scheme reads in input data, looks for groups of symbols that appear in a dictionary, if match is found, a pointer or index into the dictionary can be output. In LZW compression, simple codes of uniform width are used for all substitutions.

1.3 SUMMARY

After learning about the fundamentals about the compression, now let us proceed to the compression techniques that use integral number of bits for coding each symbol. Next chapter discusses about Shannon-Fano and Huffman coding techniques that uses variable length integral number of bits for different codes.

CHAPTER 2
SHANNON-FANO AND HUFFMAN CODING

2.1 SHANNON-FANO CODING

Claude Shannon at Bell Labs and R.M. Fano at Massachusetts Institute of Technology (MIT) simultaneously developed the first method known as Shannon-Fano coding for effectively coding symbols. It was based on simply knowing the probability of each symbol's appearance in a message and from the probabilities a table of codes could be constructed having following important properties:

(1) Variable length codes i.e. different codes have different number of bits.

(2) Codes for symbols with low probabilities have more number of bits and codes for symbols with high probabilities have less number of bits.

(3) Codes of different bit lengths should be uniquely decoded.

Developing variable length codes according to probabilities of the symbols make data compression possible and arranging these as a binary tree solves the problem of decoding these variable length codes.

2.1.1 SHANNON-FANO ALGORITHM

Building Shannon-Fano tree is simple and consists of following five steps:

(1) For given list of symbols, calculate probabilities or frequency counts.

(2) Sort the list of symbols according to frequency in descending order.

(3) Divide the sorted list into two parts, with the total frequency counts of the upper half close to the total of bottom half.

(4) Upper half of list is assigned the binary digit 0 and the lower half is assigned the digit 1 i.e. codes for the symbols in first half start with 0 and codes in second half start with 1.

(5) Apply steps 3 and 4 recursively to each of the two halves i.e. subdividing groups and adding bits to the codes until each symbol has become a leaf on the tree.

Shannon-Fano tree is built from the topdown, starting by assigning most significant bits to each code and working down the tree until finished. It was superseded by more efficient Huffman coding.

2.2 HUFFMAN ALGORITHM

Huffman codes are built from bottom up, starting with leaves of the tree and working progressively toward the root. Building Huffman tree involves following steps:

(1) All individual symbols are laid out along with their frequencies (called weights) to make up list of free nodes (called leaf nodes). These leaf nodes are going to be connected by a binary tree.

(2) The two free nodes with the lowest weights are located.

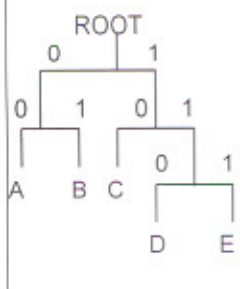
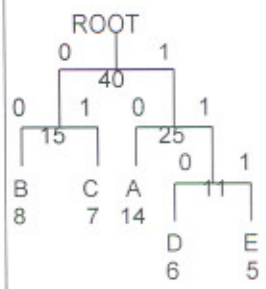
(3) Parent node for these two nodes is created and assigned weight equal to sum of weights of two child nodes

(4) Parent node is added to list of free nodes and child nodes are removed from list.

(5) One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit, other is arbitrarily set to 1 bit.

(6) Steps 2 to 5 are repeated until one free node is left. This free node is designated as the root of the tree.

The table given below gives the Huffman codes and its comparison with Shannon-Fano codes.

Sorted Frequency counts		Shannon-Fano Tree	SF Code	Huffman Tree	Huff-man Code	Info. Con-tents	Info. bits	SF bits	Huff-man bits
A	14		00		10	1.51	21.20	28	28
B	8		01		00	2.32	18.58	16	16
C	7		10		01	2.51	17.60	14	14
D	6		110		110	2.74	16.42	18	18
E	5		111		111	3.00	15.00	15	15

Tab. 2.1: Showing working and comparison of Shannon-Fano & Huffman coding.

Shannon-Fano and Huffman coding are close in performance but Huffman coding will always atleast equal in efficiency of Shannon-Fano coding.

2.2.1 DRAWBACKS OF HUFFMAN ALGORITHM:

In order 0 model, probability of given character is calculated without taking into account the character that preceded it in a message. As long as these probabilities deviate from purely uniform distribution, we are able to compress the data. A minor drawback is the requirement to transmit a copy of the probability table with compressed data. The probability is that if attempt to improve compression ability from order-0 to order-1 for better compression, we wipe out any gains by sending more modeling statistics data.

2.3 ADAPTIVE HUFFMAN ALGORITHM

It lets use higher-order modeling without paying any penalty for added statistics. It is not something than can just be used with Huffman coding, but any form of coding can be converted to use an adaptive method. In Huffman it does this by adjusting the Huffman tree on the fly based on data previously seen and having no knowledge about future statistics. This sort of coding is fairly simple.

The compressor and decompressor start off with identical models to encode and decode. After the compressor emits the first symbol, it proceeds to update model function. This is where adaptive nature of program begins. The update model takes into account the character that has just been seen and updates the frequency and encoding data used to encode the character. In Huffman tree, it means incrementing the count for the particular symbol, then updating the Huffman coding tree. Rebuilding tree would take too much work after every character. Modifying tree can be done using different approach that introduces the sibling property. Since Huffman tree is a binary tree, each node except root has a sibling i.e. other node that has same parent. A tree exhibits sibling property if the nodes can be listed in order of increasing weight and if every node appears adjacent to its sibling in the list. It shows what we need to do to a Huffman tree when we update the counts.

Updating tree consists of two basic types of operations. Firstly, increment the count. This starts at leaf node for the symbol and increment count for leaf node, then move to the parent node and increment its weight. This process continues all the way up the tree till we reach root node. Second operation required in update procedure arises when node increment causes a violation of sibling property. This occurs when the node being incremented has some weight as next highest node in the list. If increment violates the sibling property, we need to move the affected node to higher point in list i.e. the node is detached from its present position in tree and swapped with node farther up the list. To minimize the amount of work during shuffle, if newly incremented node has a weight of w+1, the next highest node will have a weight of w and there may be more nodes after it that have a value of w. The swap procedure moves up the node list till it finds the last node with weight of w. This node is swapped with the node with weight w+1. After swap, updating tree continue. The next node to be incremented will be the new parent of the incremented node. As each node is incremented, a check is performed for correct ordering. Swap is performed if necessary.

To make coding more efficient, the coder should not waste coding space for symbols not used in the message. Start coding process with empty table and add symbols only as they are seen in the incoming message. But, first time a symbol appears, it can't be encoded since it doesn't appear in table. For this escape code (special symbol) is sent out to signify that we are going to escape from the current context. Escape code signifies that next symbol to be encoded will be sent as a plain 8-bit character. Symbol is added to the table and regular encoding resumes.

2.4 SUMMARY

After learning about Shannon-Fano and Huffman coding techniques, next chapter proceeds to the compression technique that uses floating point number coding uniquely exact stream of symbols.

CHAPTER 3
ARITHMETIC CODING & STATISTICAL MODELLING

3.1 ARITHMETIC CODING

Huffman coding method discussed in the previous, Huffman codes have to be an integral number of bits long. It uses information content to efficiently encode symbols. If the probability of a character is 1/3, optimum number of bits to code is 1.6 bits. Huffman coding has to assign either one or two bits to the code. Huffman coding is not going to compress data of two-color images like those coming from fax machine.

Arithmetic coding bypasses the idea replacing an input symbol with a specific code. It replaces a stream of input symbols with a single floating-point output number. 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 be uniquely decoded to create exact stream of symbols that went into its construction. To construct the output number, the symbols are assigned set probabilities. Example to encode the message "MR. ADARSH" can be considered by constructing table showing probability distribution and range for each character as shown:

Character	Probability	Range
SPACE	1/10	0.0 – 0.10
.	1/10	0.10 – 0.20
A	2/10	0.20 – 0.40
D	1/10	0.40 – 0.50
H	1/10	0.50 – 0.60
M	1/10	0.60 – 0.70
R	2/10	0.70 – 0.90
S	1/10	0.90 – 1.00

Tab. 3.1: Probability and Range distribution.

Algorithm to code this message can be accomplished by following process:

```
low = 0.0;
high = 1.0;
while((c = getc(input)) != EOF) {
        range = high – low;
        high = low + range * high_range(c);
        low = low + range * low_range(c);
}
output(low);
```

Following above process, the message can be coded as following:

Character	Low value	High value	Range
	0.0	1.0	1.0
M	0.6	0.7	0.1
R	0.67	0.69	0.02
.	0.672	0.674	0.002
SPACE	0.672	0.6722	0.0002
A	0.67204	0.67208	0.00004
D	0.672056	0.672060	0.000004
A	0.6720568	0.6720576	0.0000008
R	0.67205736	0.67205752	0.00000016
S	0.672057504	0.672057520	0.000000016
H	0.672057512	0.6720575136	0.0000000016

Tab 3.2: Applying Arithmetic coding algorithm.

Algorithm to decode this message can be accomplished by following process:

```
number = input_code( );
for( ; ; ) {
        symbol = find_symbol_straddling_this_range(number);
        putc(symbol);
        range = high_range(symbol) – low_range(symbol);
        number = number – low_range(symbol);
        number = number / range;
}
```

Following above process, the message can be decoded as following:

Number	Low	High	Range	Output Symbol
0.672057512	0.6	0.7	0.1	M
0.72057512	0.7	0.9	0.2	R
0.1028756	0.1	0.2	0.1	.
0.028756	0.0	0.1	0.1	SPACE
0.28756	0.2	0.4	0.2	A
0.4378	0.4	0.5	0.1	D
0.378	0.2	0.4	0.2	A
0.89	0.7	0.9	0.2	R
0.95	0.9	1.0	0.1	S
0.5	0.5	0.6	0.1	H
0.0				

Tab. 3.3: Arithmetic decoding

3.1.1 COMPRESSION PROGRAM: - It first makes pass-over the input data to count all the characters. It scales down the counts to fit in unsigned characters and builds the range table used by the coder and finally writes the counts to the output file so that the decompressor have access to them later. Then arithmetic encoder is initialized by setting up high and low integer variables. In second phase encoding loop takes the character read from in from the file and looks up the range for the given symbol. This range is stored in SYMBOL having structure:

```
typedef struct { unsigned short int low_count;
                 unsigned short int high_count;
                 unsigned short int scale;
               } SYMBOL;
```

The low and high count define where in 0 to 1 range the symbol lies and scale tells what the total span of the 0 to 1 scale is. The encoder needs only these three items of SYMBOL to process the symbol and update the output number.

3.1.2 EXPANSION PROGRAM: - Firstly, the counts are read in from the input file where compressor put them and arithmetic decoder is initialized by setting low and high registers. Then 16 bits are read into current code from the

input file, which is converted into count to determine symbol that is to be decoded. Finally, the decoded symbol is written to the output file.

3.2 PRACTICAL PROBLEM

3.2.1 ENCODING: - At first glance, encoding and decoding using arithmetic coding seems impractical because of floating point numbers. But this can be best accomplished using 16-bit or 32-bit integer math requiring incremental scheme in which fixed-size integer state variables receive new bits at the low end and shift them out at the high end, forming a single number that can be as long as conceivably millions or billions of bits. To understand and clarify this, consider five-decimal digit register. When algorithm first starts, low is set to 0 and high to 1. First simplification made to work with integer math is to change 1 to .99999 or .11111 in binary and justify these numbers so that the implied decimal point is on the left side of the word. Using the algorithm, calculate the range between low and high values. The difference between the two registers will be 100000, not 99999 because it is assumed that high register has infinite number of 9's, so it is required to increment the calculated difference. Then new value of high is calculated using the formula. Before storing the new value for high, it is decremented. Calculation for low register follows the same procedure. When most significant digits of high and low match, that digit never changes and is output and appended to encoded number. Again, 9 is shifted to the least significant digit of high. This process continues and is shown in table below:

	High	Low	Range	Cumulative Output
Initial state	99999	00000	100000	
Encode M (0.6 - 0.7)	69999	60000		
Shift out 6	99999	00000	100000	.6
Encode R (0.7 - 0.9)	89999	70000	20000	.6
Encode . (0.1 - 0.2)	73999	72000		.6
Shift out 7	39999	20000	20000	.67
Encode SPACE (0.0 - 0.1)	21999	20000		.67
Shift out 2	19999	00000	20000	.672
Encode A (0.2 - 0.4)	07999	04000		.672
Shift out 0	79999	40000	40000	.6720
Encode D (0.4 - 0.5)	59999	56000		.6720
Shift out 5	99999	60000	40000	.67205

	High	Low	Range	Cumulative Output
Encode A (0.2 - 0.4)	75999	68000	8000	.67205
Encode R (0.7 - 0.9)	75199	73600		.67205
Shift out 7	51999	36000	16000	.672057
Encode S (0.9 - 1.0)	51999	50400		.672057
Shift out 5	19999	04000	16000	.6720575
Encode H (0.5 - 0.6)	13599	12000		.6720575
Shift out 1	35999	20000	16000	.67205751
Shift out 2				.672057512

Tab. 3.4: Arithmetic encoding with 5-decimal digit registers.

After all the letters are accounted for, two extra digits need to be shifted out of either high or low value to finish the output word, so that decoder can properly track the input data as part of the information is still in the high and low registers.

3.2.2 DECODING: - While decoding we have the entire input number to work with, and the algorithm had to do things like, divide the encoded number by the symbol probability, which is not possible on number that is billions of bytes long. Decoder can operate using 16 or 32-bit integers and instead of two high and low number register, a third number register is used to store the current bits being read in from the input bit stream. The code value always lie between high and low values. As they come closer and closer to it, new shift operations will take place, and high and low will move back away from code. High and low values in the decoder will be updated after every symbol output.

3.3 STATISTICAL MODELLING: - Every algorithm whether Huffman or arithmetic coding using either fixed or adaptive approaches needs a statistical model to drive them. Model needs to do two things to achieve compression:

1) It needs to accurately predict the frequency/ probability of symbols in the input data stream, and
2) The symbol probabilities generated by the model need to deviate from a uniform distribution.

3.3.1 FINITE CONTEXT MODELLING: - In this the probabilities are calculated for each symbol based on the context in which the symbol appears.

Context consists of symbols previously encountered. The order of model refers to the number of previous symbols that make up the context. Hence, order-0 model calculates probability of each symbol independently of any previous symbols. To implement this model, a single table consisting of frequency counts for each symbol that might be encountered in the input stream is prepared. Similarly, an order-1 model keeps track of 256 different tables of frequencies, since it needs a separate set of each possible context. And for order-2 model $2^8 \times 2^8 = 65,536$ different tables of contexts.

3.3.2 ADAPTIVE MODELLING: - As order of model increases, compression ratios improve, but memory consumed by the model increases exponentially. Since statistics are also carried with the compressed data so the approach poses serious problem. Adaptive compression is the solution in which both compression and decompression start with the same model. The compressor encodes a symbol using the existing model, then it updates the model to account for the new symbol. The decompressor also decodes a symbol using the existing model and then updates the model. So the process operates perfectly without need to pass statistics table from compressor to decompressor with the help of an algorithm to update the model for compressor and decompressor. Disadvantage here is that algorithm starts with less optimal statistics. But considering the cost of transmitting the statistics with the compressed data, adaptive model usually perform better than fixed context model. Adaptive modeling also suffers from cost of updating the model.

3.4 SUMMARY

After learning about statistical model compression schemes under which symbols are encoded with bit strings depending upon the statistical probabilities of such symbols, next chapter proceed to the dictionary based compression technique that under which variable length strings of symbols are encoded was tokens or indexes.

CHAPTER 4
DICTIONARY-BASED AND SLIDING WINDOW COMPRESSION

The previous compression schemes used statistical model to encode symbols into bit strings. The model predicts the probabilities of the symbols and also has to predict the probabilities that deviate from the mean, more deviation achieves better compression. Dictionary-based compression algorithms do not encode the symbol with variable-length bit strings; it encodes variable-length strings of symbols as single token. The tokens form an index to a phrase dictionary. If tokens are smaller than the phrase they replace and compression occurs. The scheme uses indexes into databases to retrieve large amounts of storage.

4.1 STATIC DICTIONARY: - In general dictionary based algorithm replaces phrases with tokens if number of bits in the token is less than the number of bits in the phrases. This leaves enormous room for variation. In some cases it is advantageous to use pre-defined dictionary to encode text e.g. if text to be encoded is a database containing all motor-vehicle registrations for India, we could develop a dictionary with only few thousand entries that concentrated on words like "Bajaj Auto Ltd.", "LML Vespa", "Hero Auto Ltd.", "Maruti Udyog Ltd.", etc. Once this dictionary is compiled, it could be kept on-line and used by both the encoder and decoder. Dictionary like this that is built up before compression occurs is called static dictionary. It does not change while the data is being compressed. Another advantage is that static dictionary can be tuned to fit the data it is compressing. But disadvantage is that it has to pass the dictionary from the encoder to decoder.

4.2 ADAPTIVE DICTIONARY: - Adaptive scheme starts out either with no dictionary or with a default baseline dictionary. As compression proceeds, the algorithms add new phrases to be used later as encoded tokens. In this the

dictionary index is encoded as an integer index into a table. The basic components of an adaptive dictionary compression algorithm are:

1. To parse the input text stream into fragments tested against the dictionary.
2. To test the input fragments against the dictionary; it may or may not be desirable to report on partial matches.
3. To add new phrases to the dictionary.
4. To encode dictionary indices and plain text so that they are distinguishable.

The decompression has neither to parse the input text stream into fragments, nor it has to test fragments against the dictionary. The corresponding components of adaptive dictionary decompression algorithm are:

1. To decode the input stream into either dictionary indices or plain text.
2. To add new phrases to the dictionary.
3. To convert dictionary indices into phrases.
4. To output phrases as plain text.

Dictionary based compression algorithm QIC-122 (Quarter Inch Cartridge) is the industry standard used by tape-drive manufacturers worldwide and is also successful as hard disk utility called Stacker program.

QIC-122 is based on LZ77 sliding-window, dictionary-based compression. Symbols are read in by encoder and added to the end of a 2K window that forms the phrase dictionary. To encode a symbol, the encoder checks to see if it is part of a phrase already in the dictionary, If it is, it creates a token that defines the location of the phrase and it's length and if not, the symbol is passed through unencoded. Thus output of QIC-122 encoder consists of tokens and symbols freely intermixed. Each token or symbol is prefixed by a single bit flag that indicates whether the following data is a dictionary reference or a plain symbol like for plaintext - <1><eight-bit-symbol> and for dictionary reference - <0><window-offset><phrase-length>. Windows offset of less than 128 bytes are encoded in seven bits and between 128 bytes to 2K bytes are encoded in eleven

bits. The phrase length uses a variable bit-coding scheme that favors short phrases over long. This would be clear from the example given below:

output_bit(1);\r output_byte(buffer[0]);

In this example, previous encoded text ends with the phrase "output_bit(1);\r at the end of the window and is to encode the phrase "output_byte". Value of phrase-length is 8 since first eight characters match. So, the encoder will output a 0 bit to indicate that a dictionary reference is following. Followed by 15 as window-offset to indicate that the start of the phrase is fifteen characters back into the window. This is followed by 8 as phrase-length. Hence it takes only 2 bytes to encode 8 bytes of data. After dictionary reference is output, the input stream over eight characters, with the last symbol encoded becoming the last symbol in the window. The next three symbols will not match anything in the window, so they will have to be individually encoded.

4.3 LZ77 COMPRESSION ALGORITHM: - LZ77 is sliding window based compression scheme in which the compression process replaces variable-length phrases in the input text with fixed-size pointers into the dictionary consisting of previously seen text. The amount of compression depends upon:

- how long the dictionary phrases are,
- how large the window into previously seen text is, and
- the entropy of the source text with respect to the model.

The main data-structure of LZ77 is text window divide into two parts. The first part consists of large block of recently decoded text and the second part which is normally much smaller in size is a look-ahead buffer consisting of characters read in from the input stream but that have not yet been encoded. The normal size of text window is several thousand characters of which look-ahead buffer is generally much smaller say ten to hundred characters. The algorithm tries to match the contents of the look-ahead buffer to a string that the longest match in the entire dictionary. The algorithm issues sequences of tokens consisting of three data items that define a phrase of variable length in the current look-ahead buffer. These three items in the token are:

(1) an offset to a phrase in the text window;

(2) length of the phrase; and

(3) the first symbol in the look-ahead buffer that follows the phrase.

This can be explained with the following example:

for (i=0 ; i<MAX-1 ; i++)\r for (j=i+1 ; j	<MAX ; j++)\r

Here text window consists of total 57 characters out of which 13 characters are used for look-ahead buffer which contains the phrase"<MAX;j++)\r". By searching through buffer, we find that "<MAX" is located at position 14 in the text window and matches the look-ahead buffer for the first four symbols. The first symbol not present in the look-ahead buffer is the " " character. The compression program for LZ77 algorithm first emits the token 14,4,' ', then shifts the text window by five characters and then read five new symbols into the look-head buffer such that the new text window becomes:

i=0 ; i<MAX-1 ; i++)\r for (j=i+1 ; j<MAX	; j++)\r a[i]=

The next token issued by the compression algorithm would encode the phrase that "; j" as 37,3,'+'. If the look-ahead buffer had no match, for example, it could be encoded a single character at a time using a phrase length of zero as 0,0,';'. But this method is not efficient though it ensures that the algorithm can encode any message.

The decompression algorithm for LZ77 doesn't have to do comparisons. It reads in token, outputs the indicated phrase, outputs the following character, shifts, and repeat. Though it maintains the window, but it does not work with string comparisons.

4.3.1 PROBLEMS WITH LZ77 COMPRESSION ALGORITHM: - For encoding, the algorithm has to perform string comparisons against the look-ahead buffer for every position in the text window. To improve compression, window size has to be increased. This makes the string comparison worse. However, the decompression portion of this algorithm does not have to suffer through this

bottleneck, as either increase in size of window or the look-head buffer will not affect it.

Second problem occurs with the way the sliding window is managed. The window is sliding across the text, progressing from the end of the buffer to the front as encoding process is executed. This is obviously not the best way to code. It is much better to have a sliding index or pointer into a fixed buffer. But by using sliding pointers, string-compare function strncmp() can't be used directly to look for longest phrase. For this a separate module has to be developed.

Thirdly, LZ77 algorithm has major efficiency problem. If phrases are not found in the dictionary, even then the compression program has to use the same three component tokens to encode single character. For example to encode 8-bit single character using 4KB window with 16-byte look-ahead buffer as (0,0,c) token would take twenty-four bits (twelve bits to encode window position, four bits to encode phrase length and other eight bits for the character). This effects the compression efficiency to great extent.

4.3.2 LZSS COMPRESSION: - This algorithm makes two changes to the LZ77 algorithm. Firstly, the way in which text window is maintained. Like in LZ77, the phrases in the text window are stored as a single contiguous block of text. But it creates additional data structure that improves the organization of the phrases. As each phrase passes out of the look-ahead buffer and into the encoded portion of the text window, the algorithm adds the phrase to binary search tree structure. This reduces the time required to find the longest matching phrase in the tree.

Secondly, as regards the tokens, the algorithm allows pointers and characters to be freely intermixed. The algorithm uses a single bit as a prefix to every output token to indicate whether it is an offset/ length pair or a symbol for output. So for several consecutive single characters, this method reduces the overhead from possibly several bytes per character down to a single bit per character.

4.4 LZ78 COMPRESSION: - First deficiency in LZ77 algorithm is that it uses only small window into previous seen text, hence, it continuously throws away valuable phrases because they slide out of the dictionary. Thus, sliding window

makes LZ77 biased toward exploiting recency in the text. Second deficiency in LZ77 compression is the limited size of a phrase that can be matched. The longest match is only equal to the size of the look-ahead buffer. These problems can be tackled by increasing the size of text window as well as look-ahead buffer. But, by doing so, number of bits used to encode index location offset and phrase length increases. This has severely negative effect on compression algorithm.

LZ78 abandons the concept of fixed text window, instead, the dictionary is a potentially unlimited list of previously seen phrases. Each LZ78 token consists of a code that selects a given phrase and a single character that follows the phrase. Phrase length is not passed because the decoder knows it.

4.4.1 LZ78 COMPRESSION ALGORITHM: - Both encoder and decoder start off with nearly empty dictionary. Dictionary has single encoded string – the null string. As each character is read in, it is added to the current string until some phrase in dictionary matches with current string. Eventually when the string no longer has a corresponding phrase in the dictionary, LZ78 sends a token and a character. At this point, LZ78 adds to the dictionary a new phrase consisting of the dictionary match and the new character last read in. Next time that phrase appears, it can be used to build an even longer phrase.

This can be explained through the following example shown below:

Input text: "ABRA CADABRA GILI GILI GILI G..."

Phrase #	Output Phrase	Output Character	Encoded String
1	0	'A'	"A"
2	0	'B'	"B"
3	0	'R'	"R"
4	1	' '	"A "
5	0	'C'	"C"
6	1	'D'	"AD"
7	1	'B'	"AB"
8	3	'A'	"RA"
9	0	' '	" "
10	0	'G'	"G"
11	0	'I'	"I"
12	0	'L'	"L"
13	11	' '	"I "

14	10	'I'	"GI"
15	12	'I'	"LI"
16	9	'G'	" G"
17	11	'L'	"IL"
18	13	'G'	"I G"

Tab. 4.1: Working of LZ78 algorithm

Here the first three characters 'A', 'B' and 'R' that have not been seen before come through the encoder as a phrase 0+character pair and forms phrase 1, 2 and 3 respectively. When the 4th character 'A' is read in, it matches the phrase 1 in dictionary, then, the next character ' ' creates a new phrase 4 - "A ". While implementing the algorithm the phrases are stored in multi-way tree as shown below for above example.

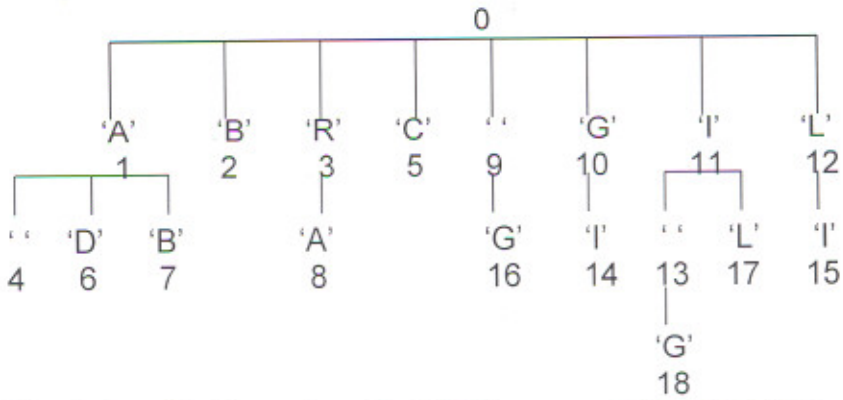


Fig. 4.1: Multi-way tree for LZ78 compression algorithm

The tree starts at a root node, 0, the null string. Each possible character that can be added to the null string is a new branch of the tree, with each phrase created that way getting a new node number. With a tree like this, comparing an existing string to the dictionary is simple. It is just a matter of walking through the tree, traversing a single node of the tree for every character in the phrase. If the phrase terminates at a particular node, then match is there. If there are more phrases but we have reached a leaf node, there is not a match. After encoding symbol, it is added to tree by inserting a new descendant node at the node last matched.

4.4.2 PROBLEMS WITH LZ78 COMPRESSION ALGORITHM: - Firstly, the decoder has to maintain the tree in same fashion as the encoder. Secondly, dictionary space is going to fill up sooner or later. For this safest choice is to stop adding phrases to the dictionary after it is full. This results in out-of-date dictionary that may not compress well. One solution to this is to monitor compression ratio of the file. If compression ratio ever starts deteriorating, the dictionary is deleted and program starts over from scratch, otherwise, existing dictionary is continued with though no new phrase are added.

4.5 LZW COMPRESSION: - This algorithm makes improvement over LZ78 compression. In this compressor never outputs single characters but phrases only. In this the dictionary is preloaded with single-symbol alphabets, so that there is no symbol that cannot be encoded even if it has not already appeared in the input stream. Sample example to explain the algorithm is given as following: -

Input text: "ABRA CADABRA GILI GILI GILI G"

Character Input	Code Output	New code value & associated string
"AB"	'A'	256 = "AB"
"R"	'B'	257 = "BR"
"A"	'R'	258 = "RA"
" "	'A'	259 = "A "
"C"	' '	260 = " C"
"A"	'C'	261 = "CA"
"D"	'A'	262 = "AD"
"A"	'D'	263 = "DA"
"BR"	256	264 = "ABR"
"A "	258	265 = "RA "
"G"	' '	266 = " G"
"I"	'G'	267 = "GI"
"L"	'I'	268 = "IL"
"I"	'L'	269 = "LI"
" "	'I'	270 = "I "
"GI"	266	271 = " GI"
"LI "	268	272 = "ILI"
" G"	270	273 = "I G"
"IL"	267	274 = "GIL"
"I "	269	275 = "LI "
"G<EOF>"	266	

Tab. 4.2: LZW compression algorithm

O/p code: "ABRA CAD<256><258> GILI<266><268><270><267><269><266>"

At first, check is performed to see if the string "AB" is in table. Since it isn't the code for 'A' is output, and the string "AB" is added to the table. Since the dictionary has codes 0-255 already defined as the 256 possible character values, the first string definition is assigned to code, "AB". After third letter, 'R', has been read in, the second string code, "BR", is added to the table, and the code for letter 'B' is output and so on. When characters 'A' and 'B' are read in, match at 256 is found, so, code 256 is output, and three-character string "ABR" is added to the string table. This process continues until the string is exhausted and all codes have been output. The string table fills up rapidly because a new string is added each time a code is output. For above example, eight code substitutions were output along with thirteen characters. If we are using nine-bit codes for output, the twenty-nine character input string would be reduced to a 23.625 byte output string.

4.5.1 LZW DECOMPRESSION: - Compression algorithm takes stream of codes output from the compression algorithm and uses them to recreate the exact input stream. LZW algorithm does not need to pass the dictionary to the decompressor. The table can be built as it was during compression. Like the compression algorithm, decompression adds a new string table each time it reads in a new code. Decompression process also translates each incoming code into a string and sends it to the output. Like compression, first 256 codes are defined to translate single-characters strings. This can be explained as follows by considering the above compressed code.

I/p code: "ABRA CAD<256><258> GILI<266><268><270><267><269><266>"

Input/ New_code	Old_Code	O/p String	Character	Table entry
'A'	''	"A"		
'B'	'A'	"B"	'B'	256 = "AB"
'R'	'B'	"R"	'R'	257 = "BR"
'A'	'R'	"A"	'A'	258 = "RA"

Input/ New_code	Old_Code	O/p String	Character	Table entry
' '	'A'	" "	' '	259 = "A "
'C'	' '	"C"	'C'	260 = " C"
'A'	'C'	"A"	'A'	261 = "CA"
'D'	'A'	"D"	'D'	262 = "AD"
256	'D'	"AB"	'A'	263 = "DA"
258	256	"RA"	'R'	264 = "ABR"
' '	258	" "	' '	265 = "RA "
'G'	' '	"G"	'G'	266 = " G"
'I'	'G'	"I"	'I'	267 = "GI"
'L'	'I'	"L"	'L'	268 = "IL"
'I'	'L'	"I"	'I'	269 = "LI"
266	'I'	" G"	' '	270 = "I "
268	266	"IL"	'I'	271 = " GI"
270	268	"I "	'I'	272 = "ILI"
267	270	"GI"	'G'	273 = "I G"
269	267	"LI"	'L'	274 = "GIL"
266	269	" G"	' '	275 = "LI "

Tab. 4.3: LZW decompression algorithm.

Like LZ78 algorithm, LZW dictionary is maintained as a multi-way dictionary. Data structure defining the multi-way dictionary in LZW algorithm is:

```
struct dictionary {
        int code_value;
        int parent_code;
        char character;
        } dict [TABLE_SIZE];
```

Each element in the data structure representing a single node has three items: (1) code_value - this number is the actual code for the string that terminates at this node and is what the compression program emits when it wants to encode the string; (2) parent_code - this integer is code for the parent string as every string in dictionary has a parent string that is one character shorter than it; (3) character - character for this particular node. Tree maintains the dictionary pointers through a hashed array of nodes. The hashing function combines the numeric values of the parent_code and the child_character to form offset into the list of nodes. If the target node is used by some other element in the tree, then, the function either finds a node already defined as belonging to the

parent and child or it finds an empty node that can be used that way. With this scheme navigation down the tree is effective but we can't move up the tree. But, during compression we don't need to move up the tree. Whereas, during decompression, hashing function is not used because we don't need to move down the tree. The structure of dictionary is such that each node in the tree has its parent code and character value stored at the array offset defined by its own code, that allows for quick lookup of dictionary values and move up the tree during decompression. During decompression the decoded characters are gathered in reverse order, so they have to be pushed into a stack, then popped off in reverse order, and written to the output file.

4.6 SUMMARY

Compression techniques discussed in this chapter and those before are applicable on the textual data where the decompressed data has to be reproduced exactly same as the original data. As regards audio and image compression better compression ratio can be obtained using the techniques to be discussed in chapters to follow. Next chapter discusses about the fundamentals of sound and compression techniques used for audio data.

CHAPTER 5
AUDIO COMPRESSION

With multimedia revolution, digital computers are able to play and manipulate digital sound. Both audio and graphical can quickly swallow up all free space on hard disk. For modern computers to manipulate sound, they first have to convert it to a digital format. The sound samples are then processed, transmitted and converted back to analog format.

5.1 FUNDAMENTAL AUDIO CONCEPTS: - Sound is produced by the vibration of matter. During the vibration, pressure variations are created in the air surrounding it. Human ears can hear sounds at frequencies ranging from 20 Hz to 20,000 Hz. This dynamic range is so wide that we have to employ logarithmic scale of measurement. Human voice frequency range from 85Hz to 1400Hz and musical instrument frequencies range from 16Hz to 4.3KHz. Sound is a waveform that is a combination of various frequencies at different amplitudes and phases. It is composed of fundamental frequency & harmonics or overtones. Sampling is the first step in working with digital audio. It consists of taking measurements of the input signal at regular times, converting them to appropriate scale, and storing them.

5.1.1 SAMPLING: - Analog signal varies continuously with time. Signal must be sampled meaning that at certain points in time a sample of input value must be taken. The rate at which a continuous waveform is sampled is called the sampling rate. In most computers, sampling is done with an analog-to-digital converter (ADC). Since audio signals are AC in nature, an eight-bit ADC may adjust the zero voltage in the middle of the range, and other outputs from the ADC would range from –128 to +127. These stored sample points represent a series of voltages that measure the input of the ADC. This forms the digitization of sound that can be stored on media. The actual speed of the storage medium is relatively unimportant with digital sound because bandwidth needed to accurately store the sound is relatively slow compared to most digital media.

In order to play back sound, digital-to-analog converter (DAC) take digital value and convert it to corresponding analog signal. This conversion must produce the exact mirror image of that performed when converting analog signal to digital. Voltages produced by DAC need not be identical to that seen at the input, but they should be proportional to one another so that one wave corresponds to the other. Also the rate at which samples are output must be same as that at which these are read. Output from DAC produces wave that give sharp jumps from one sample to other, representing high frequency components that can be eliminated using low-pass filter.

5.1.2 QUANTIZATION: - After sampling, signal is still in analog domain. Quantization is the process by which each sample is converted into binary value. During this some information is thrown away, keeping only as much as necessary to retain required accuracy or fidelity. This introduces error called quantization error.

5.1.3 SAMPLING VARIABLES: - While sampling, sample resolution (quantization) and sample rate affect quality of reproduced audio waveform. The sample resolution is a measure of how accurately the digital sample can measure the voltage it is recording. Just as a waveform is sampled at discrete times, the value of the sample is also discrete. The resolution or quantization of a sample value depends on the number of bits used in measuring the height of the waveform. For example, when input range is -500mv to +500mv, an eight-bit ADC can resolve about 4mv {2^8 = 256; 500-(-500) = 1000; 1000/256 = 4}. Thus input signal of 2mv will either get rounded to 4mv or 0mv causing quantization error. So when digital signal is played back by DAC, output waveform gets distorted. Because of large dynamic range of audio that human ears can detect, audio data may not get accurately recorded.

5.1.4 SAMPLING RATE: - This plays an important role in determining the quality of reproduction of sound. As per the Nyquist criterion, to accurately reproduce a signal of frequency 'f', the sampling rate has to be greater than 2*f. Human ears hear sound upto 20KHz, which implies that we need to sample

audio to 40KHz or better to achieve good reproduction. Quality of sound reproduced using sampling rate of music @44KHz, using sixteen-bit samples is considered as superior. Virtually every digital phone system in the world using an 8KHz sampling rate to record human speech is unable to pass signals with frequencies higher than 4KHz. Our ears detect this loss as a lower-fidelity signal, but they still understand it quit well.

5.2 LOSSY COMPRESSION: - Lossy compression implies giving up certain amount of precision. Such compression is not acceptable when compressing the data or text files. Digitizing sound samples effects quit a bit of precision, but it can be adjust to get better or worse fidelity and likewise better or worse compression.

5.2.1 SILENCE COMPRESSION: - In this lossy technique, sequences of relative silence is replaced by absolute silence. For this it needs to have:

- Threshold value below which sample can be considered as silence.
- Way to encode a run of silence.
- Parameter that gives a threshold for recognizing start of run of silence.
- Parameter that indicates that how many consecutive non-silence codes need to be seen before it is declared silence run to be over. Setting this parameter greater than one filters out anomalous spikes in the input data and thus cuts noise.

5.2.2 COMPANDING: - Silence compression can be good way of removing redundant information from sound files, but in some cases it may be ineffective. Resolution of thirteen bits when sampled at 8KHz produce acceptable sound quality, but much of this resolution goes waste. Normally in a linear conversion scheme each increase in a code value corresponds to a uniform increase in input/ output voltage. If a signal falls exactly between two binary values, the quantization error would be significantly greater. As signal's level diminishes the relative S/N ratio decreases. So a loud signal may be accurately produced, while whisper might be distorted. Today companding codec (compression/ expansion coder/ decoder) uses an exponential function that changes the size of the voltage

step between codes as the codes grows larger. This method reproduces low level signals as fine as high level ones. The resolution for smaller code values is much finer than at the extremes of the range. For larger signals, the quantization error increases but the effect is not noticeable. If N codes are there to express the range of zero to 127, following equation can be used as transfer function for each code:

Output = 127 * (pow(2.0, code / N) – 1.0)

i.e. raise 2 to the power code/N. The value of code/N range from 0 for code 0 to one for code N. This results in an output range running from 0 to 127, with non-linear look. This scheme results in more resolution in the smaller input range differentiating low sound and less resolution for loud sounds. This results in compressing sound without damaging quality. Both compression and decompression can take place via look up table, making processing fast and the amount of compression is known in advance. So, the compression can be tuned to any compression ratio by varying the number of codes.

Lossy compression is frequently used as a front end to a lossless compressor. After the files have been compressed, far fewer codes are present in the output file, which makes string matching more likely, so that high compression of sound files is achievable.

5.2.3 MUE-LAW: - This is companding standard used in North America. Actual voltage value (Y) for each segment is determined by following formula:

$$Y = \frac{V \ln (1 + u\, v / V)}{\ln (1 + u)}$$

where Y is compressed value out of input voltage v for a signal whose maximum peak value is V. 'u' is constant. This technique is also called 255 law because value of u = 255. Graph comparing compressed voltage (Y) with input voltage (v) for peak values of +/- 1 V is as shown:

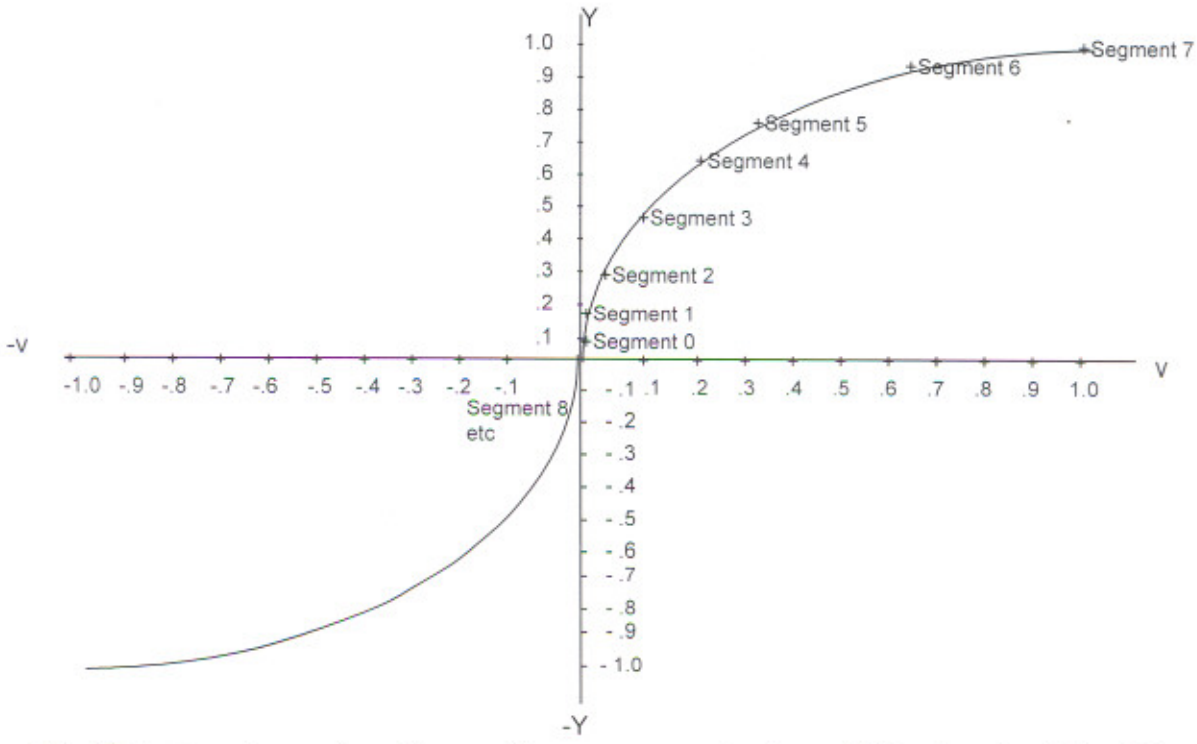


Fig. 5.1: Input sample voltage v Vs compressed voltage Y (Peak value V = 1.0)

Whole graph is divided into 15 segments. Segments are coded as shown below:

Sign	Segment Number			Level Value			
S				A	B	C	D
b7	b6	b5	b4	b3	b2	b1	b0

The first bit is sign bit, logic 1 representing positive value and logic 0, negative values. The next three bits designate the segment numbers 000 to 111 for 0 to 7. The fifteenth segment is the 0 segment, which is half-positive and half-negative. The last four bits referred as bits ABCD determine the actual value level within the range of a particular segment. Table Tab. 5.1 shows the voltage levels represented by a compressed signal using mue-law. In this step size doubles with each next segment and levels are divided into sixteen steps within each segment.

segment	step (mV)	Maximum Voltage (mV)
0	0.5	16 * 0.5 = 8
1	1.0	8 + 16 * 1.0 = 24
2	2.0	24 + 16 * 2.0 = 56
3	4.0	56 + 16 * 4.0 = 120
4	8.0	120 + 16 * 8.0 = 248
5	16.0	248 + 16 * 16 = 504
6	32.0	504 + 16 * 32 = 1016
7	64.0	1016 + 16 * 64 = 2040

Tab. 5.1: Example of mue-law

For digital compression, analog signal is first digitized and coded using 12 bits. Format of this 12 bit code is as shown in table 5.2. First bit is the sign bit with remaining 11 bits reflecting the magnitude of the sample. Table shows eight codes for the 12 bits. Bits ABCD contain any one of sixteen values from 0000 to 1111. The x values have actual digital values that get discarded during compression process.

b11 ◄——————————————————————► b0

Segment	S	Data code bits										
0	S	0	0	0	0	0	0	0	A	B	C	D
1	S	0	0	0	0	0	0	1	A	B	C	D
2	S	0	0	0	0	0	1	A	B	C	D	x
3	S	0	0	0	0	1	A	B	C	D	x	x
4	S	0	0	0	1	A	B	C	D	x	x	x
5	S	0	0	1	A	B	C	D	x	x	x	x
6	S	0	1	A	B	C	D	x	x	x	x	x
7	S	1	A	B	C	D	x	x	x	x	x	x

Tab. 5.2: 12 bit quantize sample codes.

Digitally, the 12-bit values are encoded into 8 bit compressed code as per following algorithm:

1. Retain the sign bit as the first bit of the 8-bit code.

2. Count the number of 0s until the occurrence of the first 1 bit. Subtract the zero count from 7. This is the segment number.
3. The first occurrence of a 1 is assumed during the expanding process, so it is set-aside during compression.
4. Copy the next 4 bits ABCD in to the 8 bit compressed code.

Hence the compressed code consists of a sign bit, segment number, and the value of the ABCD bits.

Expanding back digitally reverses the process:

1. Retain the sign bit.
2. Take the segment number, subtract from 7, and add that many 0s.
3. Make the next bit as 1, append ABCD bits.
4. Add a 1 and sufficient 0s to complete the 12-bit value.

The addition of the 1 and 0 its following the ABCD bits sets a value in the middle between the actual values the remaining bits could have had. This introduces some error that is minimal since it occurs in the least significant bits of the 12-bit code.

For example, 12-bit 100001011010 is coded as 10110110 as per 8-bit compressed law code (Regain sign bit 1, zero count is 4, so subtract 7-4 = 3 = 011, discard first 1 and get ABCD = 0110). Now to expand back, retain sign bit 1, subtract the segment number from 7 i.e. 7-3 = 4 and insert 4 0s, replace the discarded 1, append the ABCD bits followed by 10 to get 10001011010.

5.2.4 ADAPTIVE DIFFERENTIAL PULSE CODE MODULATION: - ADPCM algorithm combines two techniques. Firstly, the delta pulse code modulation encodes sound signals by measuring the difference between two consecutive samples. This depends on the notion that analog signal tends to vary in smooth patterns, which is true as long as the sampling rate of the signal is higher than the maximum frequency. Secondly, the quantization level adapts itself to the changing input signals, so that the size of the encoded voltage changes as the input signal changes. High step value is required as signal moves from high voltage to low voltage and low step value is required for a quiet signal. So, most

of the processor time in ADPCM algorithm is utilized in predicting in advance where the input signal is heading.

Linear Predictive Coding (LPC) can be used to compress and replay human speech using lesser bit rates. LPC attempts to compress human.speech by modeling the vocal tract that produces the speech. It attempts to determine just few parameters that model the process used to create the sound.

5.3 SUMMARY

Compression techniques discussed how audio signal could be compressed, these techniques were lossy techniques, but the errors introduced are such that these do not effect the fidelity to appreciable extent, so as to distort the signal that is totally different from the actual. The next chapter discusses about the compression techniques used for the compression of the image data.

CHAPTER 6
IMAGE COMPRESSION

Image is a spatial representation of an object, a two-dimensional or three-dimensional scene or another image. A recorded image may be in a photographic, analog video signal or digital format. In computer graphics, image is always a digital image. Millions of man-hours and billions of money are spent just making improvements in the way programs display data. Programmers spend enormous amount of time and effort trying to accommodate the proliferation of the Graphical User Interface (GUI). Use of graphics is involved in virtually every area of applications like games, education, desktop publishing, and graphical design. The images used in these consume enormous amount of disk storage. Statistical and dictionary based compression don't do very well on graphical and image data because pixels in photographic images tend to spread out over entire range. Like audio data, graphical images can be slightly modified during compression/ expansion cycle without affecting the perceived quality. Minor changes may go unnoticed if done carefully. Graphical images are generally scanned from real world sources that represent an already imperfect representation of a photograph or some other printed media. So it is feasible to represent the image using lossy compression/ expansion cycle.

Image compression exploits redundancies in the image that are: -

- Spatial redundancy: - due to correlation between neighboring pixels.
- Spectral redundancy: - due to correlation between different color planes.
- Temporal redundancy: - due to correlation of different frames in a sequence of images.

Major steps in image compression are: -

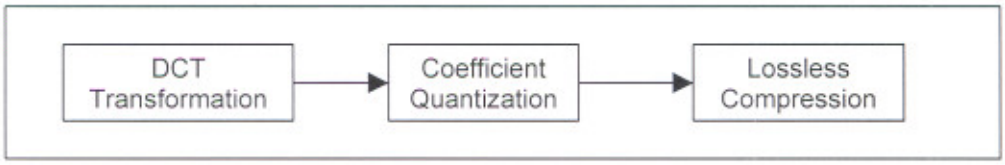
- Picture preparation
- Picture processing
- Quantization

- Entropy encoding

6.1 RUNLENGTH ENCODING: - This technique exploits spatial redundancies. A run-length is the number of successive pixels having the same value. Image can be coded as pair: (run-length, gray-code). The algorithm can be:

- A run-length is identified by having all its 3 most significant bits set to 1. Its lower 5 bits provide a counting mechanism of following byte.
- Example, 111XXXXXbbbbbbbb

6.2 JOINT PHOTOGRAPHIC EXPERTS GROUP (JPEG) COMPRESSION: - JPEG is the leading format for lossy graphics compression. It comprises of specifications for both lossless and lossy encoding. In this three steps, shown by the block representation, combine to form a powerful compressor capable of compressing continuous tone images to less than 10 percent of their original size while losing little of original fidelity.



6.2.2 DISCRETE COSINE TRANSFORM (DCT): - DCT performs mathematical operation including Fast Fourier Transform (FFT) that takes signal and transform it from one type of representation to another. FFT transforms the set of sample points in time domain into a set of frequency values describing the same signal. Such transformation is reversible i.e. using inverse FFT function; the same set of time domain signal can be obtained. These two transformation cycles are lossless except for loss of precision resulting from rounding and truncation. Like

FFT, DCT takes a set of points from the spatial domain and transforms them into frequency domain. DCT operate on three-dimensional signal – two dimensions X and Y of the screen and Z-axis denoting the colour on the screen at that point. This spatial information can be converted into frequency or spectral information with X and Y axes representing frequencies of the signal in two different dimensions. Inverse DCT (IDCT) function can convert the spectral representation of the signal back to a spatial one.

DCT formula that is performed on an N x N square matrix of pixel values yielding an N x N matrix of frequency coefficients is given below:

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i)\, C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos \frac{(2x+1)i(pie)}{2N} \cos \frac{(2y+1)j(pie)}{2N}$$

$C(x) = 1/\sqrt{2}$ if x is 0, else 1 if x > 0

Inverse DCT that is performed on an N x N square matrix of frequency coefficients yielding back an N x N square matrix of pixel values is given by:

$$pixel(x,y) = \frac{1}{\sqrt{2N}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C(i)\, C(j)\, DCT(i,j) \cos \frac{(2x+1)i(pie)}{2N} \cos \frac{(2y+1)j(pie)}{2N}$$

$C(x) = 1/\sqrt{2}$ if x is 0, else 1 if x > 0

The above functions can be implement using straightforward code. Lookup tables can replace many terms of the equation. The two cosine terms that have to be multiplied together need to calculated once at the beginning for the program. Similarly the C(x) terms that fall outside the summation loops can also be replaced with lookup table. After this the code look somewhat like this:

```
for ( i = 0; i < N; i++ )
  for ( j = 0; j < N; j++ ) {
      temp = 0.0;
      for ( x = 0; x < N; x++ )
        for ( y = 0; y < N; y++ ) {
            temp += cosines[x][i] * cosines[y][j] * pixel[x][y];
      }
      temp *= sqrt( 2 * N ) * coefficients[i][j];
      DCT[i][j] = INT_ROUND(temp);
}
```

Using DCT, pixels are transformed into frequency coefficients, but number of points remains the same i.e. N x N matrix. All elements in row 0 have a frequency component of zero in one direction of the signal and all elements in column 0 have a frequency component of zero in the other direction. Origin represents the DC component. As the rows and columns move away from origin, the coefficients in the transformed DCT matrix begin to represent higher frequencies, with N-1 position indicating highest frequency. Most graphical images are composed of low-frequency information so the components found in zero row and column, i.e. DC components, carry more useful information about the image than the higher-frequency components. Also the coefficients far away from the DC components not only tend to have lower values but they become far less important for describing the image. Thus DCT transformation identifies pieces of information that can be thrown without compromising with the quality of image.

6.2.2.1 **SIZE OF MATRIX**: - Time required for each element of DCT is heavily dependent on the size of the matrix N. Amount of calculation needed to perform DCT transformation on even 256x256 grey-scale block is large. So, DCT implementation breaks the image into smaller manageable blocks. JPEG selected an 8x8 block though increasing the size would probably give better compression (diminishing returns). Connections between pixels that are even fifteen or twenty positions away are of very little use as predictors.

Since DCT is related to Discrete Fourier Transform, many techniques used to speed up the family of Fourier Transforms can be applied to DCT.

Input consisting of an 8x8 matrix of pixel values is fed to DCT algorithm. Output comprises of 8x8 matrix having spectral compression characteristics. The value at (0,0) i.e. upper-left corner position is the DC coefficient representing the average of the overall input matrix. As the elements move farther from DC coefficient, they tend to become lower and lower in magnitude meaning thereby that performing DCT results in concentrating the image in the upper left coefficients of the output matrix with lower right coefficients of the DCT matrix containing less useful information.

6.2.3 QUANTIZATION: - Quantization is the second process after DCT transformation in JPEG compression process. DCT transformation itself is lossless transformation that prepares for the lossy quantization process. Quantization is a process of reducing the number of bits required to store integer value by reducing the precision of the integer. This is implemented using quantization matrix. Every element position of quantization matrix gives quantum value that indicate what step size is going to be for that element in the compressed rendition of picture with values ranging from 1to 255. The elements that matter most to the picture are encoded with small step size. Formula used for quantization for encoding and dequantization for decoding are:

$$\text{Quantized value}(i,j) = \frac{\text{DCT}(i,j)}{\text{Quantum}(i,j)} \quad \text{Rounded to nearest integer}$$

$$\text{DCT}(i,j) = \text{Quantized value}(i,j) * \text{Quantum}(i,j)$$

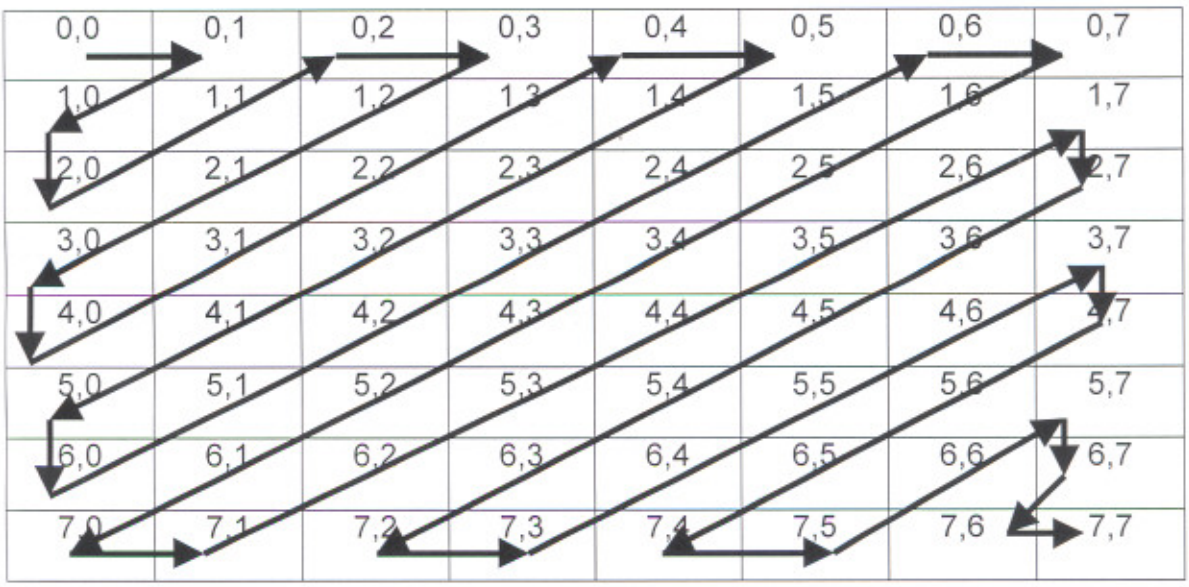
To determine the value of the quantum step sizes, the user can select a single quality factor ranging from about twenty-five. Following code can be used to constructing the quantization matrix:

```
for ( i = 0; i < N; j++ )
for ( j = 0; j < N; j++ )
Quantum[i][j] = 1 + ((1+i+j) * quality);
```

Value larger than 25 can work for quality factor, but this would degrade the picture quality. The elements of quantization matrix set the threshold value for the corresponding elements of DCT coefficients to be discarded. Applying quantization/ dequantization cycle, the frequency portions of the DCT matrix get truncated down to zero. Even quality factor of two that makes only minor changes barely noticeable can allow image compression by 60 percent.

6.2.4 CODING: - In JPEG compression, coding combines three steps. Firstly, it changes the DC coefficient from an absolute value to relative value because adjacent blocks in an image exhibit a high degree of correlation. Coding DC element as difference from the previous DC element produces very small number. Secondly, the resultant coefficients of the image are arranged in zig-zag sequence. And finally, these are encoded either using run-length or entropy encoding.

6.2.4.1 ZIG-ZAG ARRANGEMENT: - After applying quantization, many values are set to zero. These are coded using run-length encoding. In order to increase the length of runs, coefficients are reordered in zig-zag sequence. For this JPEG algorithm traces the coefficients in zig-zag pattern using simple lookup table as shown as following:



6.2.4.2 ENTROPY ENCODING:- After reordering the DCT block in zig-zag sequence, JPEG algorithm outputs the elements using an entropy encoding mechanism consisting of three tokens:

(i) Run-length: number of consecutive zeros that preceded the current element

(ii) Bit-count: number of bits to follow in the amplitude number and the

(iii) Amplitude: of DCT coefficient.

6.3 COLOURED IMAGE: - The above algorithm gives the procedure for compressing one colour component i.e. grey scale image. On the other coloured images are composed of three components i.e. red, green and blue (RGB) or the luminance and chrominance (YUV). For the coloured image, JPEG treats it as if it were actually three separate images i.e. RGB image would first have its red component compressed, then its green, and then its blue.

6.4 FILE EXPANSION: - After the compression, file expansion is relatively easy. The expansion routine first reads in the quality factor from the file, then it sits in a loop reading 8x8 DCT blocks. This routine also takes DCT data out of zigzag sequence and storing it in row normal pattern. Then it dequantizes this data, after which inverse DCT procedure is executed which returns it to pixel data. After reading entire strip of pixel data it is written to the uncompressed output file.

6.5 FRACTAL IMAGE COMPRESSION: - The term fractal is used to designate objects that are self-similar at different scales. This can be viewed as a form of compression in which equation, described with a few bits of information can be implemented in a very short program, whereas the resulting image needs infinite amount of bits in form of set of pixels. Thus, instead of representing the image as long sequence of pixel values, the image can be constructed from a formula, which can be encoded in a much smaller number of bytes. For example, to represent black circular disk on a white background, the equation specifying it is given by: $(x-a)^2 + (y-b)^2 < r^2$, where r is radius of disk with center (a,b).

Unfortunately, real-world images never let themselves be represented as such simple equations. It takes advantage of the self-similarity present in an image to find an approximate representation of the image as a fractal. Most of the time, it is impossible to find an iterated function system to represent complete image. Instead image is partitioned into non-overlapping ranges and then finding a local iterated function system for each range.

6.5.1 PARTITIONED ITERATED FUNCTION SYSTEM: - Fractal compression with PIFS work by finding redundancy within the input image in the form of similar image portions. Such image compression is lossy as the method look for approximate matching. In another method called vector quantization, the input image is partitioned into small pixel blocks and each block is encoded as a reference to the dictionary pattern that most resembles the block. Block has same size as corresponding dictionary pattern, but all blocks need not be of same size. The decoder must have a copy of dictionary from which the approximate original image can be constructed. In fractal compression, there is no external dictionary. Input image acts as its own dictionary. The decoder doesn't have this image initially, but it can reconstruct it gradually by iterating a PIFS.

The compressor partitions the input image into a set of non-overlapping square, rectangular or triangular ranges. For each range, the compressor looks for a part of the input image called a domain, which is similar to the range. Domains may overlap. In general, compressor looks for domains that are twice as large as the range. To assess the similarity between a domain D and a range R, the compressor finds the best possible mapping w from the domain to the range, so that the image w(D) is as close as possible to the image R. Two dimensional transformations are used for black and white images whereas three dimensions are used for grey scale images – two for the spatial components and one for the luminance component. A point (x,y) with luminance z belonging to domain D_I is mapped as:

$$W_i \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_{i,1} & a_{i,2} & 0 \\ a_{i,3} & a_{i,4} & 0 \\ 0 & 0 & c_i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} d_{i,1} \\ d_{i,2} \\ b_i \end{bmatrix}$$

Where constants $a_{i,j}$ and $d_{i,j}$ specify the geometric part and the constants c_i and b_i specify the luminance part. In practice, the constants $a_{i,j}$ and $d_{i,j}$ are implicitly defined by the relative size, orientation and position of the domain with respect to the range. To reduce the complexity to manageable proportions, compressor only looks for domains that are exactly twice as large as the range so that scaling factor is equal to 0.5. Similarly, domains and ranges are restricted to squares so that there are only 8 possible orientations of the domain relative to the square – 4 rotations and four symmetries. Thus requiring only 3 bits to encode these orientations. Translation constants $d_{i,j}$ are determined by the position of the top left corner of the domain. For a given range R, compressor examines a number of possible domains. For each such domain D, compressor must find the optimal affine map w from D to R which minimizes the distance between the image R and the image w(D). The RMS distance depends only on the contrast factor c_i and the brightness offset b_i. The distance is minimum when the partial derivatives with respect to these two variables are both zero. c_i and b_i are obtained by solving two simple linear equations. The compressor chooses the domain with the smallest distance, encodes the corresponding affine map and goes on working to the next range.

6.6 MOVING PICTURES EXPERTS GROUP: - Moving pictures experts group or MPEG is the coded representation of motion video. Moving images often contain non-translational moving patterns. MPEG distinguishes four types of image coding for processing. These still images are called frames.

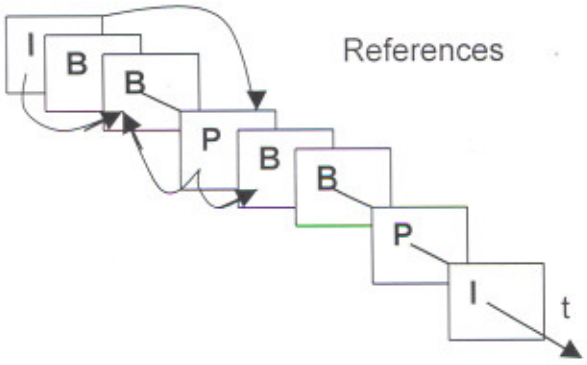
6.6.1 I-Frames: - These are self-contained Intra-coded images i.e. these are coded without any reference to other images. An I-frame is still image for which MPEG uses JPEG. Compression rate for I-frame is lowest within the MPEG. These are the points for random access in MPEG streams. I-frame use 8x8 blocks defined within a macro block on which DCT is performed. MPEG distinguishes two types of macro blocks – the first type includes only the encoded data and the second covers a parameter used for scaling by adjustment of the quantization characteristics.

6.6.2 P-Frames: - P-frames called the predictive coded frames require information of the previous I-frame and/ or all previous P-frames for encoding and decoding. Coding for these are based on the fact that, by successive images, their areas often do not change at all but instead, the whole area is shifted. In this case of temporal redundancy, the block of the last P- or I- frame that is most similar to the block under consideration is determined and several methods for motion estimation are available to the encoder.

6.6.3 B-Frames: - These are the Bi-directional predictive-coded frames that require the information of the previous and following I- and/ or P- frames for encoding and decoding. Highest compression ratio is attainable by using these frames. A B-frame is defined as the difference of a prediction of the past image and the following P- or I-frame. B-frames can never be randomly accessed. B-frames must not be stored in the decoder as a reference for subsequent decoding images.

6.6.4 D-Frames: - DC-coded frames are intraframe-encoded. The DC-parameters are DCT-coded and the AC-coefficients are neglected. D-frames consist only of the lowest frequencies of an image. They only use one type of macro block and only the DC coefficients are encoded. These are used for fast forward or fast rewind modes. This could also be realized by a suitable order of I-frames, whereby, I-frames occur periodically in data stream.

A sequence of I-, P- and B-frames is shown below.



For practical applications "IBBPBBPBB IBBPBBPBB....." sequence has proved to be useful. Decoding and display order are shown below:

Decoding order:

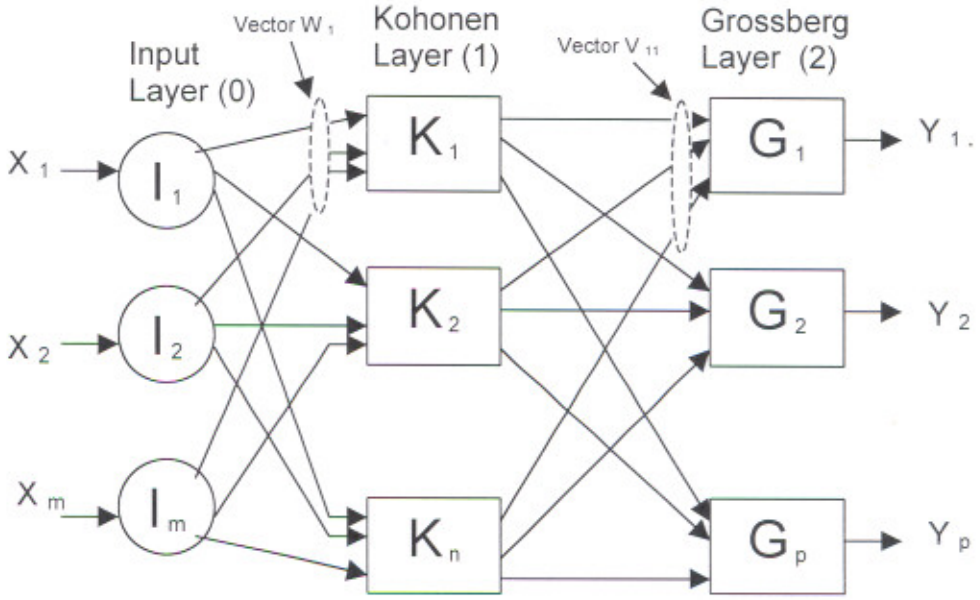
Type of Frame	: I	B	B	P	B	B	P	B	B	P	B	B
Number of Frame	: 2	0	1	5	3	4	8	6	7	11	9	10

Display order:

Type of Frame	: B	B	I	B	B	P	B	B	P	B	B	P
Number of Frame	: 0	1	2	3	4	5	6	7	8	9	10	11

6.7 COUNTERPROPAGATION NETWORK: - In neural networks, a counterpropagation network can be used to compress both speech and image data. An image is divided into subimages. Each subimage is further divided into pixels. Each subimage is then a vector, the elements of which are the pixels of which the subimage is composed. For simplicity, suppose that each pixel is either one (light) or zero (dark). If there are n pixels in a subimage then n bits will be required to transmit it. If some distortion can be tolerated, substantially fewer bits are actually required to transmit typical images, thereby allowing an image to be compressed and be transmitted more rapidly. This is possible because of the statistical distribution of subimage vectors; some occur frequently, while others occur so seldom that they can be approximated roughly. The method of vector quantization finds these shorter bit strings that best represent the subimages. Vector quantization can be performed by counterpropagation network. The set of subimage vectors is used as input to train the Kohonen layer in the accertive mode in which only a single neuron is allowed to be 1. The Grossberg layer weights are trained to produce the binary code of the index of the Kohonen neuron that is 1. Example, if Kohonen neuron 7 is 1 (and the others are all 0), the Grossberg layer will be trained to output 00...000111 (binary code for 7). It is this shorter bit string that is transmitted. At receiving end, an identically trained counterpropagation network accepts the binary code and produces the inverse function, an approximate of the original subimage.

6.7.1 COUNTERPROPAGATION NETWORK STRUCTURE:-



Each neuron in layer 0 shown as circle serve only as fan-out points that connects to every neuron in layer 1 called Kohonen layer through separate weight w_{mn} collectively referred to as weight matrix W. Similarly, each neuron in Kohonen layer connects to every neuron in layer 2 called Grossberg layer through weight v_{np} referred to as weight matrix V. In normal mode, counterpropagation network accept input vector X and produces output vector Y, whereas, in training mode, an input vector is applied and weights are adjusted to yield the desired output vector.

6.8 SUMMARY

Although compression techniques that are used to compress textual data can also be used for compressing the audio and image data, but, more efficient compression techniques discussed in this chapter are used to get much higher compression rates. The techniques described in this chapter exploit the perception of human along with the redundancy and probability of data. Such audio or image data can be compressed to large extent depending upon the quality of the encoded data output. So, one has to strive a balance between the compression and quality.

6.9 PROGRAMS TO KNOW

COMPACT in Unix world is general-purpose order-0 compression program that uses adaptive Huffman coding. It is slow and proprietary, so not available to all Unix users. compress on Unix is implementation of LZW dictionary based compression. It is better than COMPACT as it is faster. Source code is public-domain program, but continued use is questionable because LZW is patent held by Unisys. Today, compression standard in DOS world is the shareware program PKZIP by Phil Katz called WinZIP that integrate with Windows File Manager provide a modern interface to a venerable file format.

Dictionary based compression algorithm QIC-122 (Quarter Inch Cartridge) is the industry standard used by tape drive manufacturers worldwide and is also successful as hard disk utility called stacker.

CONCLUSION

Some of the compression techniques were practically implemented and executed on data files of different sizes and having different probabilities of data. Txt1.txt file is lengthy file whereas txt5.txt is a file that repeatedly contains the alphabets 'A', 'B' and 'C'. Other txt2.txt, txt3.txt and txt4.txt are normal files of different sizes. Results obtained are mentioned below.

Data file name	Compression Technique applied	Original size of file in bytes	Size of compressed file in bytes	Compression ratio
ssf.doc	Huffman	256512	168760	35%
Txt1.txt	Huffman	29962	17519	42%
Txt1.txt	Arithmetic	29962	17420	42%
Txt1.txt	LZW	29962	13650	55%
Txt2.txt	Huffman	1874	1324	30 %
Txt2.txt	Arithmetic	1874	1320	30 %
Txt2.txt	LZW	1874	1282	32 %
Txt3.txt	Huffman	5140	3237	38 %
Txt3.txt	Arithmetic	5140	3211	38 %
Txt3.txt	LZW	5140	2824	46 %
Txt4.txt	Huffman	2426	1496	39 %
Txt4.txt	Arithmetic	2426	1487	39 %
Txt4.txt	LZW	2426	1440	42 %
Txt5.txt	Huffman	832	225	73 %
Txt5.txt	Arithmetic	832	202	76 %
Txt5.txt	LZW	832	107	88 %
Logo.bmp	jpeg	27778	8896	68%

Concludingly, data compression depends on the nature of data i.e. text, audio or image. Loss of data cannot be compromised with text including database records, spreadsheets and executable files. Whereas, compromising with accuracy of data is tolerable in case of audio and image files and higher compression rates can be attained. For textual data, more compression is attainable where data is not uniformly distributed and contains more redundancies. Compression also depends upon type of compression technique applied. There is neither any single optimum technique by which maximum compression can be attained nor any particular technique is most appropriate for all kinds of data. It is quit possible that a technique when it is applied to one file compresses better than other techniques, whereas on the other hand, some other compression technique is more suitable to some other file. But, definitely there is always a scope that a more efficient and better compression can be done. It must also be taken into consideration that the implementation of data compression should be cost-effective with minimal complexity in the technique and that the processing of the algorithm must not exceed certain time spans.

BIBLIOGRAPHY

1. "The Data Compression Book", William Gibson and Bruce Sterling.
2. "Neural Computing: Theory & Practice", Philip D. Wasserman.
3. "Digital Image Processing", Gonzalez, Rafael C. & Richard E. Woods.
4. "The JPEG Still Picture Compression Standard" by Gregory K. Wallace, Multimedia Engineering, DEC, Maynard, Massachusetts, December 1991 for IEEE Transaction on Consumer Electronics.
5. "Multimedia Computing and Communications", Nicolas D. Georganas and Donaenico Ferrari.