DESIGNING A SUITE TO IMPLEMENT COMPRESSION AND SECURITY ALGORITHMS FOR DATA AND IMAGE

Enrollment No. -101275

Name of Student- Shaini Gupta

Name of Supervisor- Dr. Pradeep Kumar Gupta

Submitted in partial fulfilment of the Degree of

Bachelor of Technology

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING,

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,

WAKNAGHAT

TABLE OF CONTENTS

CERTIFICATE

This is to certify that the work titled **“Designing a suite to implement compression and security algorithms for data and image”** submitted by **“Shaini Gupta”** in partial fulfilment for the award of degree of B.Tech of Jaypee University of Information Technology, Waknaghat has been carried out under my supervision .This work has not been submitted partially and wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor : ____________________

Name of Supervisor: Dr. Pradeep Kumar Gupta

Designation: Assistant Professor (Sr. Grade) Computer Science Department

Date : __________________

ACKNOWLEDGEMENT

First of all I would like to acknowledge the almighty god for bestowing his good wishes and giving me the strength at every moment of despair to complete this project.

It’s incumbent on my part to thank my project guide **Dr. Pradeep Kumar Gupta** Dept. of Computer Science and Information Technology, Jaypee University of Information Technology, Waknaghat, Solan (H.P.) who has been a great support and guided me throughout the completion of my project.

Last but not the least I would like to express my warm thanks to my respected parents and all my friends for their support and their constructive suggestions, which enabled me to bring improvement in my project.

Signature of Student: ____________________

Name of Student: Shaini Gupta

Date: __________________

SUMMARY

This project focuses on implementing and comparing the various algorithms of data compression and image compression on the basis of various parameters like efficiency, compression ratio, number of bits etc. This report covers Shannon Fano, Huffman algorithms as a part of data compression techniques and Bit Plane Slicing as a part of image compression techniques. Additionally I have added image reconstruction from its Bit planes. With this report the compression technique has been investigated to yield results on their efficiency, and a discussion is presented as to what issues have brought to bare on their efficiency results. Data compression has emerged as an important enabling technology in a wide variety of communication and storage applications, ranging from disk doubling operating systems that provide extra storage space to facsimile standards that facilitate the flow of business information and to the high definition audio and video standards that allow maximal use to be made of scarce satellite transmission bandwidth. Bit Plane Slicing is the technique in which instead of highlighting gray level ranges ,highlighting the contribution made to total image appearance by specific bits may be desired. Separating a digital image into its bit planes is useful for analyzing the relative importance played by each of the image, a process that aids in determining the adequacy of the number of bits used to quantize each pixel.

_________________________	___________________________
Signature of Student	Signature of Supervisor
Name: Shaini Gupta	Name: Dr. Pradeep kumar Gupta
Date: ____________________	Date: ______________________

LIST OF FIGURES

CHAPTER 1:

1.1 INTRODUCTION

This project composed of compression techniques and resultant information derived from investigation .This application suite of data compression technique allows for compression techniques to be applied to a data file in succession for the purpose of optimal compression . Additionally the complementary and inverse operation of decomposition can also be carried out to restore the file to its original state in a loss less manner.

Data compression is emerging as an important enabling technology in a wide variety of communication and storage applications, ranging from the facsimile standards that facilitate the flow of business information to disk doubling operating systems that provide extra storage space and to the high definition audio and video standards that allow maximal use to be made of available satellite transmission bandwidth.

1.1.1 Brief about Shannon Fano Algorithm:

The first well known method for effectively coding symbols is now known as Shannon Fano coding. Claude Shannon at bell labs and R.M. Fano at MIT developed this method nearly simultaneously. It depends on simply knowing the probability of each symbol’s appearance in a message. Given the probabilities, a table of codes could be constructed that has several important properties:

- ✓ Different codes have different number of bits.
- ✓ Codes for symbols with the low probabilities have more bits, and codes for symbols with high probabilities have fewer bits.
- ✓ Though the codes are of different bit lengths, they can be uniquely decoded.

The first two properties go hand in hand. Developing codes that vary in length according to the probability of the symbol they are encoding makes data compression possible. And arranging the codes as a binary tree solves the problem of decoding these variable-length codes.
An example of the type of decoding tree used in Shannon-Fano coding is shown below. Decoding an incoming code consists of starting at the root, then turning left or right at each node after reading an incoming bit from the data stream. Eventually a leaf of the tree is reached, and the appropriate symbol is decoded.

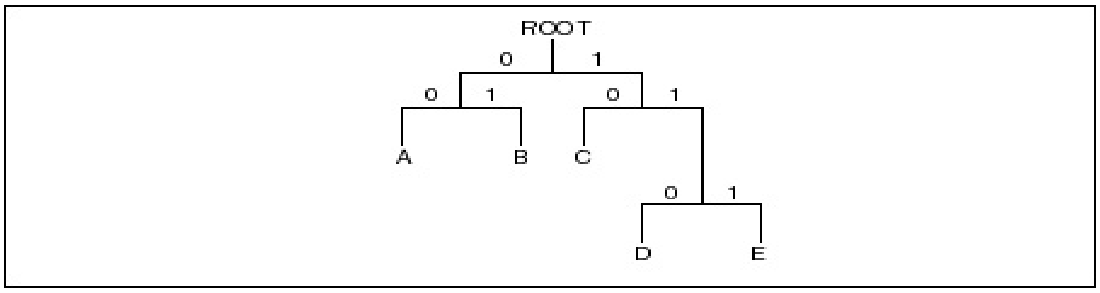


FIGURE 1.1 A simple Shannon fano tree

The tree structure shows how codes are uniquely defined though they have different numbers of bits. The tree structure seems designed for computer implementations, but it is also well suited for machines made of relays and switches, like the teletype machines of the 1950s.
While the table shows one of the three properties discussed earlier, that of having variable numbers of bits, more information is needed to talk about the other two properties.

1.1.2 Brief about Huffman algorithm:

Huffman coding shares most characteristics of Shannon-Fano coding. It creates variable-length codes that are an integral number of bits. Symbols with higher probabilities get shorter codes. Huffman codes have the unique prefix attribute, which means they can be correctly decoded despite being variable length. Decoding a stream of Huffman codes is generally done by following a binary decoder tree.
Building the Huffman decoding tree is done using a completely different algorithm from that of the Shannon-Fano method. The Shannon-Fano tree is built from the top down, starting by assigning the most significant bits to each code and working down the tree until finished. Huffman codes are built from the bottom up, starting with the leaves of the tree and working progressively closer to the root.
The procedure for building the tree is simple and elegant. The individual symbols are laid out as a string of leaf nodes that are going to be connected by a binary tree. Each node has a weight, which is simply the frequency or probability of the symbol's appearance. The tree is then built with the following steps:

- The two free nodes with the lowest weights are located.
- A parent node for these two nodes is created. It is assigned a weight equal to the sum of the
- two child nodes.

- The parent node is added to the list of free nodes, and the two child nodes are removed from
- the list.
- One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit. The other is arbitrarily set to the 1 bit.

The previous steps are repeated until only one free node is left. This free node is designated the root of the tree.

This algorithm can be applied to the symbols used in the previous example. The five symbols in our message are laid out, along with their frequencies, as shown:

TABLE: To demonstrate Huffman Algorithm

Symbol	Count
A	15
B	7
C	6
D	6
E	5

These five nodes are going to end up as the leaves of the decoding tree. When the process first starts, they make up the entire list of free nodes.

The first pass through the tree identifies the two free nodes with the lowest weights: D and E, with weights of 6 and 5. (The tie between C and D was broken arbitrarily. While the way that ties are broken affects the final value of the codes, it will not affect the compression ratio achieved.) These two nodes are joined to a parent node, which is assigned a weight of 11. Nodes D and E are then removed from the free list.

Once this step is complete, we know what the least significant bits in the codes for D and E are going to be. D is assigned to the 0 branch of the parent node, and E is assigned to the 1 branch. These two bits will be the LSBs of the resulting codes.

On the next pass through the list of free nodes, the B and C nodes are picked as the two with the lowest weight. These are then attached to a new parent node. The parent node is assigned a weight of 13, and B and C are removed from the free node list. At this point, the tree looks like that shown in Figure 1.3

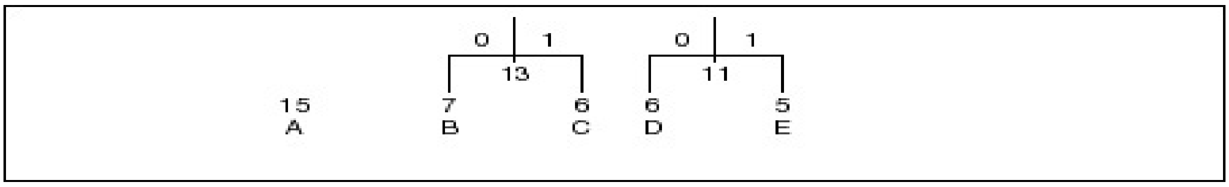


FIGURE 1.2 First phase of Huffman tree construction

On the next pass, the two nodes with the lowest weights are the parent nodes for the B/C and D/E pairs. These are tied together with a new parent node, which is assigned a weight of 24, and the children are removed from the free list. At this point, we have assigned two bits each to the Huffman codes for B, C, D, and E, and we have yet to assign a single bit to the code for A.

Finally, on the last pass, only two free nodes are left. The parent with a weight of 24 is tied with the A node to create a new parent with a weight of 39. After removing the two child nodes from the free list, we are left with just one parent, meaning the tree is complete. The final result looks like that shown in Figure1.4

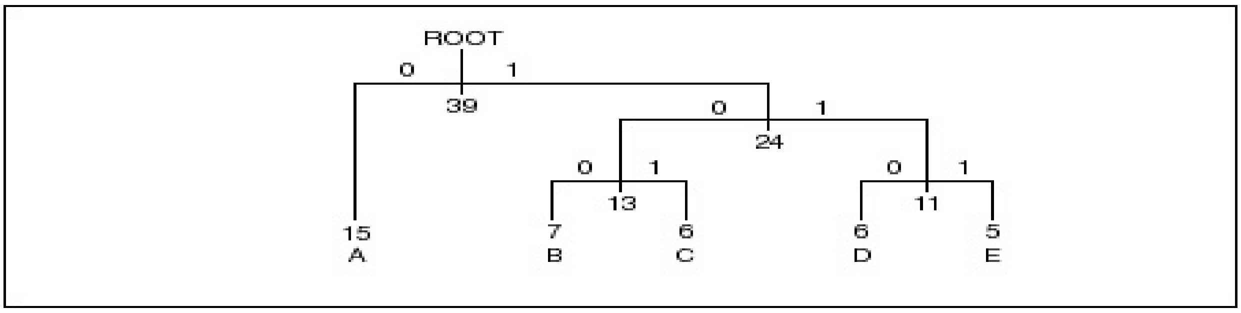


FIGURE 1.3 Final tree constructed after applying Huffman Algorithm

Bit Plane Slicing is the technique in which instead of highlighting gray-level ranges, highlighting the contribution made to total image appearance by specific bits may be desired. Separating a digital image into its bit planes is useful for analyzing the relative importance played by each bit of the image, a process that aids in determining the adequacy of the number of bits used to quantize each pixel.

Also this type of decomposition is useful for image compression. As a consequence of the language used (JAVA), the application suite can be considered portable across platforms by the inherent property of the language executable. The language is assumed to function on platform equipped with the appropriate run time environment. So far this is common place on Windows, UNIX, and Mac Operating System. Also due to the ever growing Internet, Such transparency of the use of an application is a desired quality. A final point is that the application suite is maintainable with respect to addition of new codices (compression and decompression techniques).

1.2 GOALS OF PROJECT:

This project involves dealing with data and image compression techniques. The goals were to:

1. To study the different algorithms related to this topic and implement the best ones out of them.

2. To check the implemented algorithm for various input data and their compression ratio and time efficiency.

3. To draw a comparison among the various data compression techniques.

4. To implement image compression through bit plane slicing.

5. To implement image reconstruction.

6. To create its GUI for user convenience.

1.3 SOFTWARE AND HARDWARE REQUIREMENTS:

1.3.1 SOFTWARE TOOLS

- LANGUAGE USED:

  The language used at front-end is JAVA. The reasons of selecting this language are:

  - ✓ Highly technical language.
  - ✓ User friendly environment of working.
  - ✓ Provide low cost solution to the project.
  - ✓ GUI feature.
  - ✓ Better designing aspects for developers.

- IDE USED:

  The Integrated Development Environment is NetBeans IDE, Ecilpse, and MATLAB.

1.3.2 HARDWARE REQUIREMENT SPECIFICATION:

The required hardware combination for this project will be as following:

- ✓ Intel Pentium processor.
- ✓ Processor Speed – 1.7GHz or above.
- ✓ RAM - 256 MB or above.
- ✓ HDD – 20 GB or above.
- ✓ VGA Monitor.
- ✓ Scroll mouse, Multimedia keyboard, CD-Drive.

CHAPTER 2:

LITERATURE REVIEW

2.1 COMPUTER FORENSICS

Computer forensics (sometimes known as computer forensic science) is a branch of digital forensic science pertaining to legal evidence found in computers and digital storage media. The goal of computer forensics is to examine digital media in a forensically sound manner with the aim of identifying, preserving, recovering, analyzing and presenting facts and opinions about the information. Although it is most often associated with the investigation of a wide variety of computer crime, computer forensics may also be used in civil proceedings. The discipline involves similar techniques and principles to data recovery, but with additional guidelines and practices designed to create a legal audit trail. Evidence from computer forensics investigations is usually subjected to the same guidelines and practices of other digital evidence. It has been used in a number of high profile cases and is becoming widely accepted as reliable with U.S. and European court systems.

In the early 1980s personal computers became more accessible to consumers, leading to their increased use in criminal activity (for example, to help commit fraud). At the same time, several new "computer crimes" were recognized (such as hacking). The discipline of computer forensics emerged during this time as a method to recover and investigate digital evidence for use in court. Since then computer crime and computer related crime has grown exponentially, and even has jumped 67% between 2002 and 2003. Today it is used to investigate a wide variety of crime, including child pornography, fraud, espionage, cyber stalking, murder and rape. The discipline also features in civil proceedings as a form of information gathering (for example, Electronic discovery).

Forensic techniques and expert knowledge are used to explain the current state of a digital artefact, such as a computer system, storage medium (e.g. hard disk or CD-ROM), an electronic document (e.g. an email message or JPEG image). The scope of a forensic analysis can vary from simple information retrieval to reconstructing a series of events. In a 2002 book Computer Forensics authors Kruse and Heiser define computer forensics as involving “the preservation, identification, extraction, documentation and interpretation of computer data”. They go on to describe the discipline as "more of an art than a science", indicating that forensic methodology is backed by flexibility and extensive domain knowledge. However, while several methods can be used to extract evidence from a given computer the strategies used by law enforcement are fairly rigid and lacking the flexibility found in the civilian world.

Use as evidence:

In court, computer forensic evidence is subject to the usual requirements for digital evidence. This requires that information be authentic, reliably obtained, and admissible. Different countries have specific guidelines and practices for evidence recovery. In the United Kingdom, examiners often follow Association of Chief Police Officers guidelines that help ensure the authenticity and integrity of evidence. While voluntary, the guidelines are widely accepted in courts in Wales, England, and Scotland.

Forensic process:

A portable Tableau write blocker attached to a Hard Drive Computer forensic investigations usually follow the standard digital forensic process (acquisition, analysis and reporting). Investigations are performed on static data (i.e. acquired images) rather than "live" systems. This is a change from early forensic practices where a lack of specialist tools led to investigators commonly working on live data.

Techniques:

A number of techniques are used during computer forensics investigations.

- *Cross-drive analysis:*
  A forensic technique that correlates information found on multiple hard drives. The process, still being researched, can be used to identify social networks and for perform anomaly detection.
- *Live analysis:*
  The examination of computers from within the operating system using custom forensics or existing sys admin tools to extract evidence.

The practice is useful when dealing with Encrypting File Systems, for example, where the encryption keys may be collected and, in some instances, the logical hard drive volume may be imaged (known as a live acquisition) before the computer is shut down.

- *Stochastic forensics:*

  A method which uses stochastic properties of the computer system to investigate activities lacking digital artifacts. Its chief use is to investigate data theft.

- *Deleted files:*

  A common technique used in computer forensics is the recovery of deleted files. Modern forensic software has their own tools for recovering or carving out deleted data. Most operating systems and file systems do not always erase physical file data, allowing investigators to reconstruct it from the physical disk sectors. File carving involves searching for known file headers within the disk image and reconstructing deleted materials.

- *Steganography:*

  One of the techniques used to hide data is via steganography, the process of hiding data inside of a picture or digital image. This process is often used to hide pornographic images of children as well as information that a given criminal does not want to have discovered. Computer forensics professionals can fight this by looking at the hash of the file and comparing it to the original image (if available.) While the image appears exactly the same, the hash changes as the data changes.

- *Volatile data:*

  When seizing evidence, if the machine is still active, any information stored solely in RAM that is not recovered before powering down may be lost. One application of "live analysis" is to recover RAM data (for example, using Microsoft's COFEE tool, Windows SCOPE) prior to removing an exhibit. Capture GUARD Gateway bypasses Windows login for locked computers, allowing for the analysis and acquisition of physical memory on a locked computer.

  RAM can be analyzed for prior content after power loss, because the electrical charge stored in the memory cells takes time to dissipate, an effect exploited by the cold boot attack.

The length of time that data is recoverable is increased by low temperatures and higher cell voltages. Holding unpowered RAM below −60 °C helps preserve residual data by an order of magnitude, improving the chances of successful recovery. However, it can be impractical to do this during a field examination.

Some of the tools needed to extract volatile data, however, require that a computer be in a forensic lab, both to maintain a legitimate chain of evidence, and to facilitate work on the machine. If necessary, law enforcement applies techniques to move a live, running desktop computer. These include a mouse jiggler, which moves the mouse rapidly in small movements and prevents the computer from going to sleep accidentally. Usually, an uninterruptible power supply (UPS) provides power during transit.

However, one of the easiest ways to capture data is by actually saving the RAM data to disk. Various file systems that have journaling features such as NTFS and Reiser FS keep a large portion of the RAM data on the main storage media during operation, and these page files can be reassembled to reconstruct what was in RAM at that time.

Analysis tools

A number of open source and commercial tools exist for computer forensics investigation. Typical forensic analysis includes a manual review of material on the media, reviewing the Windows registry for suspect information, discovering and cracking passwords, keyword searches for topics related to the crime, and extracting e-mail and pictures for review.

2.1.1. Incident Response

Incident response is an organized approach to addressing and managing the aftermath of a security breach or attack (also known as an incident). The goal is to handle the situation in a way that limits damage and reduces recovery time and costs. An incident response plan includes a policy that defines, in specific terms, what constitutes an incident and provides a step-by-step process that should be followed when an incident occurs.

An organization's incident response is conducted by the computer incident response team, a carefully selected group that, in addition to security and general IT staff, may include representatives from legal, human resources, and public relations departments.

According to the SANS Institute, there are six steps to handling an incident most effectively:

- Preparation: The organization educates users and IT staff of the importance of updated security measures and trains them to respond to computer and network security incidents quickly and correctly.
- Identification: The response team is activated to decide whether a particular event is, in fact, a security incident. The team may contact the CERT Coordination Center, which tracks Internet security activity and has the most current information on viruses and worms.
- Containment: The team determines how far the problem has spread and contains the problem by disconnecting all affected systems and devices to prevent further damage.
- Eradication: The team investigates to discover the origin of the incident. The root cause of the problem and all traces of malicious code are removed.
- Recovery: Data and software are restored from clean backup files, ensuring that no vulnerabilities remain. Systems are monitored for any sign of weakness or recurrence.
- Lessons learned: The team analyzes the incident and how it was handled, making recommendations for better future response and for preventing a recurrence.

Digital forensics and incident response are two of the most critical fields in all of information security. The staggering number of reported breaches in the last year has shown that the ability to rapidly respond to attacks is a vital capability for all organizations. Unfortunately, the standard IT staff member is simply unable to effectively respond to security incidents. Successful handling of these situations requires specific training in a number of very technical areas including file system implementation, operating system design, and knowledge of possible network and host attack vectors. During this training, students will learn both the theory around digital forensics and incident response as well as gain valuable hands-on experience with the same types of evidence and situations they will see in real-world investigations. The class is structured so that a specific analysis technique is discussed and then the students immediately analyze staged evidence using their newly gained knowledge. Not only does this approach reinforce the material learned, but it also gives the investigator a number of new skills as the course proceeds. Upon completion of the training, students will be able to effectively preserve and analyze a large number of digital evidence sources, including both on-disk and in-memory data. These skills will be immediately usable in a number of investigative scenarios, and will greatly enhance even experienced investigators' skill set. Students will also leave with media that contains all the tools and resources used throughout the training.

2.1.2 DATA RECOVERY

Data recovery generally involves things that are broken - whether hardware or software. When a computer crashes and won't start back up, when an external hard disk, thumb drive, or memory card becomes unreadable, then data recovery may be required. Frequently, a digital device that needs its data recovered will have electronic damage, physical damage, or a combination of the two. If such is the case, hardware repair will be a big part of the data recovery process. This may involve repairing the drive's electronics, or even replacing the stack of read / write heads inside the sealed portion of the disk drive.

If the hardware is intact, the file or partition structure is likely to be damaged. Some data recovery tools will attempt to repair partition or file structure, while others look into the damaged file structure and attempt to pull files out. Partitions and directories may be rebuilt manually with a hex editor as well, but given the size of modern disk drives and the amount of data on them, this tends to be impractical.

By and large, data recovery is a kind of "macro" process. The end result tends to be a large population of data saved without as much attention to the individual files. Data recovery jobs are often individual disk drives or other digital media that have damaged hardware or software. There are no particular industry-wide accepted standards in data recovery.

Computer forensics has aspects data recovery. Data recovery procedures may be brought into play to recover deleted files intact. But frequently the CFE must deal with purposeful attempts to hide or destroy data that require skills outside those found in the data recovery industry.

Most often, data recovery deals with one disk drive or the data from one system. The data recovery house will have its own standards and procedures and works on reputation, not certification. Electronic discovery frequently deals with data from large numbers of systems, or from servers with that may contain many user accounts. E-discovery methods are based on proven software and hardware combinations and are best planned for far in advance (although lack of pre-planning is very common).

Computer forensics may deal with one or many systems or devices may be fairly fluid in the scope of demands and requests made, often deals with missing data, and must be defensible - and defended - in court.

2.2 DATA COMPRESSION

The primary purpose of this chapter is to explain various data-compression techniques. Data compression seeks to reduce the number of bits used to store or transmit information. It encompasses a wide variety of software and hardware compression techniques which can be so unlike one another that they have little in common except that they compress data. The LZW algorithm used in the CompuServe GIF specification, for example, has virtually nothing in common with the CCITT G.721 specification used to compress digitized voice over phone lines. The field has grown in the last 25 years to a point where this is simply not possible. What this book will cover are the various types of data compression commonly used on personal and midsized computers, including compression of binary programs, data, sound, and graphics. Furthermore, this book will either ignore or only lightly cover data-compression techniques that rely on hardware for practical use or that require hardware applications.

Many of today's voice-compression schemes were designed for the worldwide fixed bandwidth digital telecommunications networks. These compression schemes are intellectually interesting, but they require a specific type of hardware tuned to the fixed bandwidth of the communications channel. Different algorithms that don't have to meet this requirement are used to compress digitized voice on a PC, and these algorithms generally offer better performance. Some of the most interesting areas in data compression today, however, do concern compression techniques just becoming possible with new and more powerful hardware. Lossy image compression, like that used in multimedia systems, for example, can now be implemented on standard desktop platforms. This book will cover practical ways to both experiment with and implement some of the algorithms used in these techniques.

The Data-Compression Lexicon, with a History:

Like any other scientific or engineering discipline, data compression has a vocabulary that at first seem overwhelmingly strange to an outsider. Terms like Lempel-Ziv compression, arithmetic coding, and statistical modelling get tossed around with reckless abandon. While the list of buzzwords is long enough to merit a glossary, mastering them is not as daunting a project as it may first seem. With a bit of study and a few notes, any programmer should hold his or her own at a cocktail-party argument over data compression techniques.

The Two Kingdoms:

Data-compression techniques can be divided into two major families **lossy** and **lossless**. Lossy data compression concedes a certain loss of accuracy in exchange for greatly increased compression. Lossy compression proves effective when applied to graphics images and digitized voice. By their very nature, these digitized representations of analog phenomena are not perfect to begin with, so the idea of output and input not matching exactly is a little more acceptable. Most lossy compression techniques can be adjusted to different quality levels, gaining higher accuracy in exchange for less effective compression. Until recently, lossy compression has been primarily implemented using dedicated hardware. In the past few years, powerful lossy-compression programs have been moved to desktop CPUs, but even so the field is still dominated by hardware implementations.

Lossless compression consists of those techniques guaranteed to generate an exact duplicate of the input data stream after a compress/expand cycle. This is the type of compression used when storing database records, spreadsheets, or word processing files. In these applications, the loss of even a single bit could be catastrophic.

Data Compression = Modelling + Coding

In general, data compression consists of taking a stream of symbols and transforming them into codes. If the compression is effective, the resulting stream of codes will be smaller than the original symbols. The decision to output a certain code for a certain symbol or set of symbols is based on a model. The 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 those probabilities.

Modelling and coding are two distinctly different things. People frequently use the term coding to refer to the entire data-compression process instead of just a single component of that process. You will hear the phrases “Huffman coding” or “Run-Length Encoding,” for example, to describe a data-compression technique, when in fact they are just coding methods used in conjunction with a model to compress data. Using the example of Huffman coding, a breakdown of the compression process looks something like this:

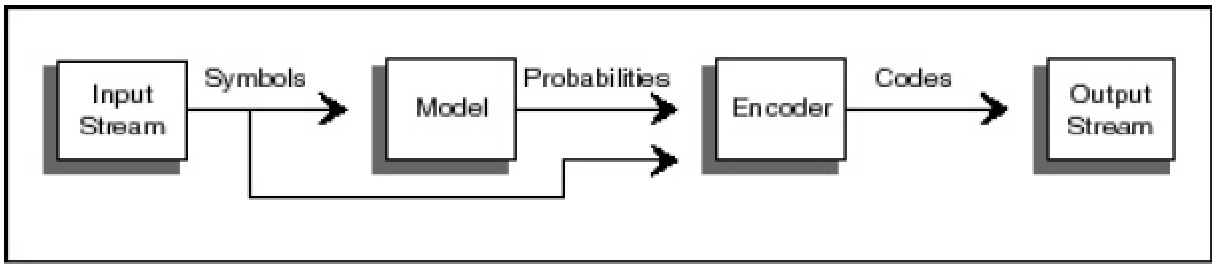


FIGURE 2.1: Statistical model of Huffman encoder

In the case of Huffman coding, the actual output of the encoder is determined by a set of probabilities. When using this type of coding, a symbol that has a very high probability of occurrence generates a code with very few bits. A symbol with a low probability generates a code with a larger number of bits.

We think of the model and the program’s coding process as different because of the countless ways to model data, all of which can use the same coding process to produce their output. A simple program using Huffman coding, for example, would use a model that gave the raw probability of each symbol occurring anywhere in the input stream.

A more sophisticated program might calculate the probability based on the last 10 symbols in the input stream. Even though both programs use Huffman coding to produce their output, their compression ratios would probably be radically different.

So when the topic of coding methods comes up at your next cocktail party, be alert for statements like “Huffman coding in general doesn’t produce very good compression ratios.” This would be your perfect opportunity to respond with “That’s like saying Converse sneakers don’t go very fast. I always thought the leg power of the runner had a lot to do with it.” If the conversation has already dropped to the point where you are discussing data compression, this might even go over as a real demonstration of wit.

The Dawn Age:

Data compression is perhaps the fundamental expression of Information Theory. Information Theory is a branch of mathematics that had its genesis in the late 1940s with the work of Claude Shannon at Bell Labs. It concerns itself with various questions about information, including different ways of storing and communicating messages.

Data compression enters into the field of Information Theory because of its concern with redundancy. Redundant information in a message takes extra bit to encode, and if we can get rid of that extra information, we will have reduced the size of the message.

Information Theory uses the term entropy as a measure of how much information is encoded in a message. The word entropy was borrowed from thermodynamics, and it has a similar meaning. The higher the entropy of a message, the more information it contains.

The entropy of a symbol is defined as the negative logarithm of its probability. To determine the information content of a message in bits, we express the entropy using the base 2 logarithm:

Number of bits = - Log base 2 (probability)

The entropy of an entire message is simply the sum of the entropy of all individual symbols.

Entropy fits with data compression in its determination of how many bits of information are actually present in a message. If the probability of the character ‘e’ appearing in this manuscript is 1/16, for example, the information content of the character is 4 bits. So the character string “eeeee” has a total content of 20 bits. If we are using standard 8-bit ASCII characters to encode this message, we are actually using 40 bits. The difference between the 20 bits of entropy and the 40 bits used to encode the message is where the potential for data compression arises.

One important fact to note about entropy is that, unlike the thermodynamic measure of entropy, we can use no absolute number for the information content of a given message. The problem is that when we calculate entropy, we use a number that gives us the probability of a given symbol. The probability figure we use is actually the probability for a given model, not an absolute number. If we change the model, the probability will change with it. How probabilities change can be seen clearly when using different orders with a statistical model. A statistical model tracks the probability of a symbol based on what symbols appeared previously in the input stream. The order of the model determines how many previous symbols are taken into account. An order-0 model, for example, won’t look at previous characters. An order-1 model looks at the one previous character, and soon. The different order models can yield drastically different probabilities for a character. The letter ‘u’ under an order-0 model, for example, may have only a 1 percent probability of occurrence. But under an order-1 model, if the previous character was ‘q,’ the ‘u’ may have a 95 percent probability.

This seemingly unstable notion of a character’s probability proves troublesome for many people. They prefer that a character have a fixed “true” probability that told what the chances of its “really” occurring are. Claude Shannon attempted to determine the true information content of the English language with a “party game” experiment. He would uncover a message concealed from his audience a single character at a time. The audience guessed what the next character would be, one guess at a time, until they got it right. Shannon could then determine the entropy of the message as a whole by taking the logarithm of the guess count. Other researchers have done more experiments using similar techniques.

While these experiments are useful, they don’t circumvent the notion that a symbol’s probability depends on the model. The difference with these experiments is that the model is the one kept inside the human brain. This may be one of the best models available, but it is still a model, not an absolute truth. In order to compress data well, we need to select **models** that predict symbols with high probabilities. A symbol that has a high probability has low information content and will need fewer bits to encode. Once the model is producing high probabilities, the next step is to encode the symbols using an appropriate number of bits.

2.3 EXISTING ALGORITHM

2.3.1 SHANNON FANO:

This is a basic information theoretic algorithm. A simple example will be used to illustrate the algorithm:

Symbol	A	B	C	D	E
Count	15	7	6	6	5

Encoding for the Shannon-Fano Algorithm:

- A top-down approach

- Sort symbols according to their frequencies/probabilities, e.g., ABCDE.
- Recursively divide into two parts, each with approx. same number of counts.

Symbol	Count	log(1/p)	Code	Subtotal (# of bits)
A	15	1.38	00	30
B	7	2.48	01	14
C	6	2.70	10	12
D	6	2.70	110	18
E	5	2.96	111	15
				TOTAL (# of bits): 89

DETAILED DESCRIPTION:

A Shannon-Fano tree is built according to a specification designed to define an effective code table. The actual algorithm is simple:

- For a given list of symbols, develop a corresponding list of probabilities or frequency counts so that each symbol's relative frequency of occurrence is known.
- Sort the lists of symbols according to frequency, with the most frequently occurring symbols at the top and the least common at the bottom.
- Divide the list into two parts, with the total frequency counts of the upper half being as close to the total of the bottom half as possible.
- The upper half of the list is assigned the binary digit 0, and the lower half is assigned the digit 1. This means that the codes for the symbols in the first half will all start with 0, and the codes in the second half will all start with 1.
- Recursively apply the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree.

2.3.2 HUFFMAN ALGORITHM:

1. Scan text to be compressed and tally occurrence of all characters.
2. Sort or prioritize characters based on number of occurrences in text.

3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.

2.4 IMAGE COMPRESSION

2.4.1 BIT PLANE SLICING

Instead of highlighting gray-level ranges, highlighting the contribution made to total image appearance by specific bits might be desired.

Suppose that each pixel in an image is represented by 8 bits. Imagine that the image is composed of eight 1-bit planes, ranging from bit-plane 0 for the least significant bit to bit plane7 for the most significant bit. In terms of 8-bit bytes, plane 0 contains all the lowest order bits in the bytes comprising the pixels in the image and plane 7 contains all the high-order bits.

Note that the higher-order bits (especially the top four) contain the majority of the visually significant data. The other bit planes contribute to more subtle details in the image.

Separating a digital image into its bit planes is useful for analyzing the relative importance played by each bit of the image, a process that aids in determining the adequacy of the number of bits used to quantize each pixel. Also, this type of decomposition is useful for image compression. Bit plane slicing is a part of spatial domain. With the help of Bit plane slicing we are able to get image compression. With the help of bit plane slicing we are able to reconstruct any image.

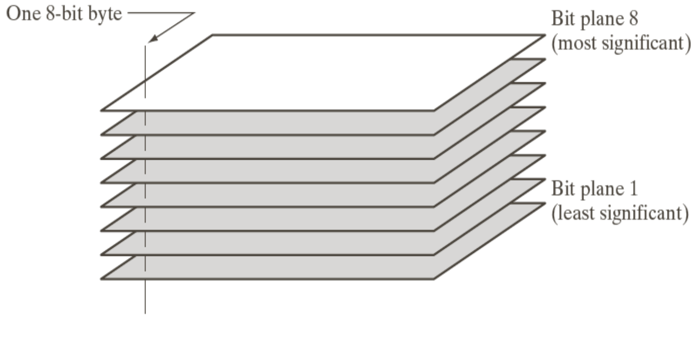


FIGURE 2.2 Representation of various bit planes

2.4.2 IMAGE RECONSTRUCTION

It is basically a process in which image is reconstructed with the help of various bit planes which are extracted from the original image. With this technique, number of bits to represent a pixel, is reduced significantly .Thus leads to its compression.

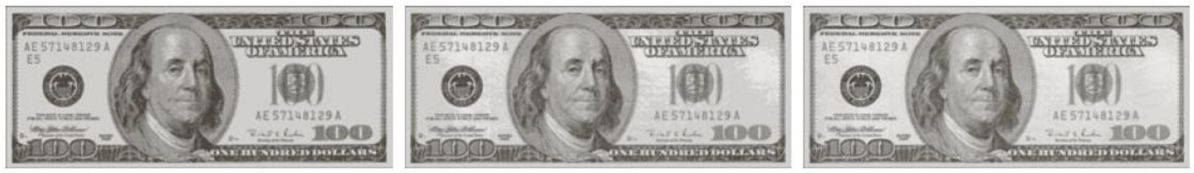


FIGURE 2.3 Image reconstruction using 2, 3 and 4 bit planes respectively instead of 8.

CHAPTER 3:

PROPOSED SOLUTION

3.1 SHANNON FANO IMPLEMENTATION FLOW:

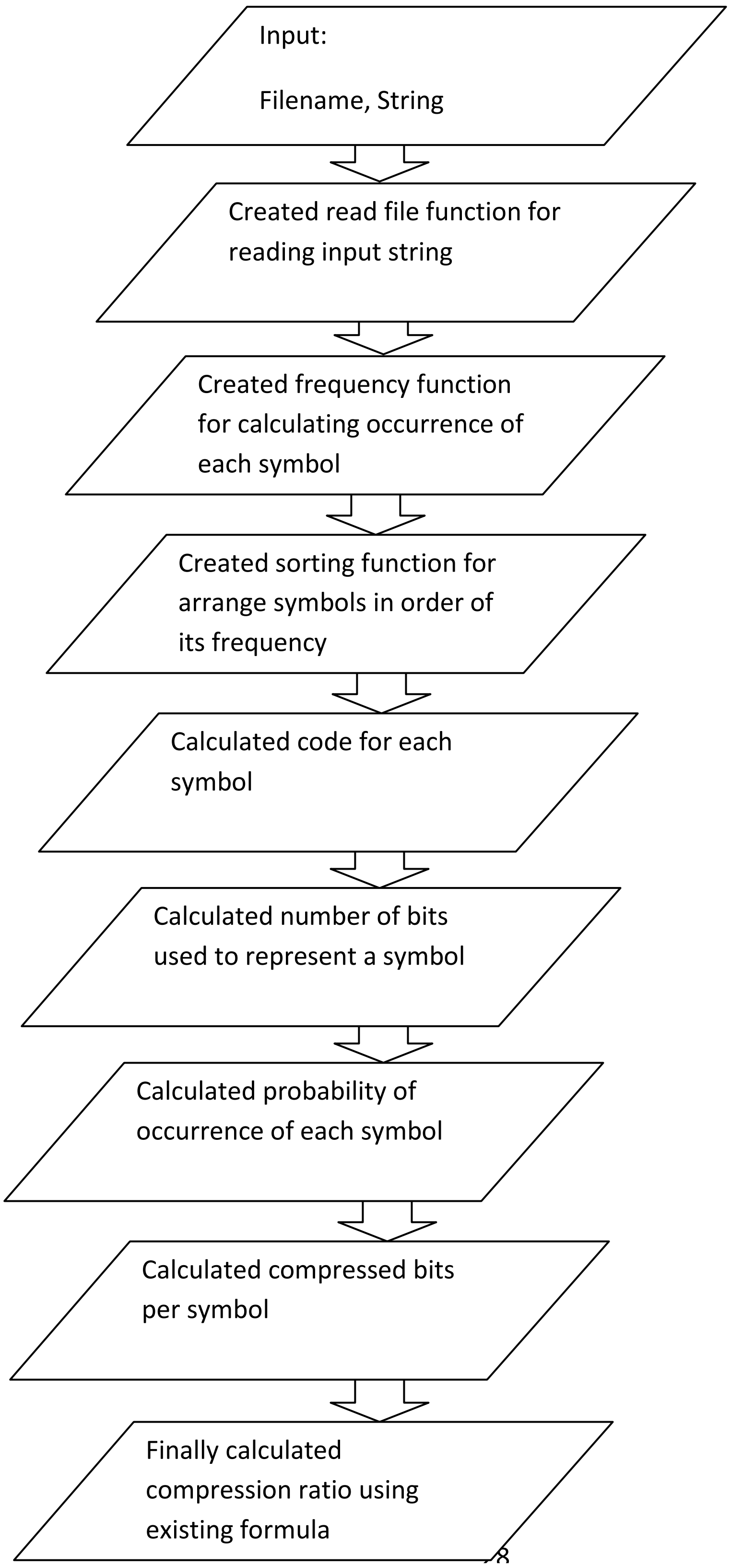

Input:
Filename, String
Created read file function for reading input string
Created frequency function for calculating occurrence of each symbol
Created sorting function for arrange symbols in order of its frequency
Calculated code for each symbol
Calculated number of bits used to represent a symbol
Calculated probability of occurrence of each symbol
Calculated compressed bits per symbol
Finally calculated compression ratio using existing formula

FIGURE 3.1 Steps of Shannon fano algorithm

3.2 HUFFMAN ALGORITHM IMPLEMENTATION FLOW:

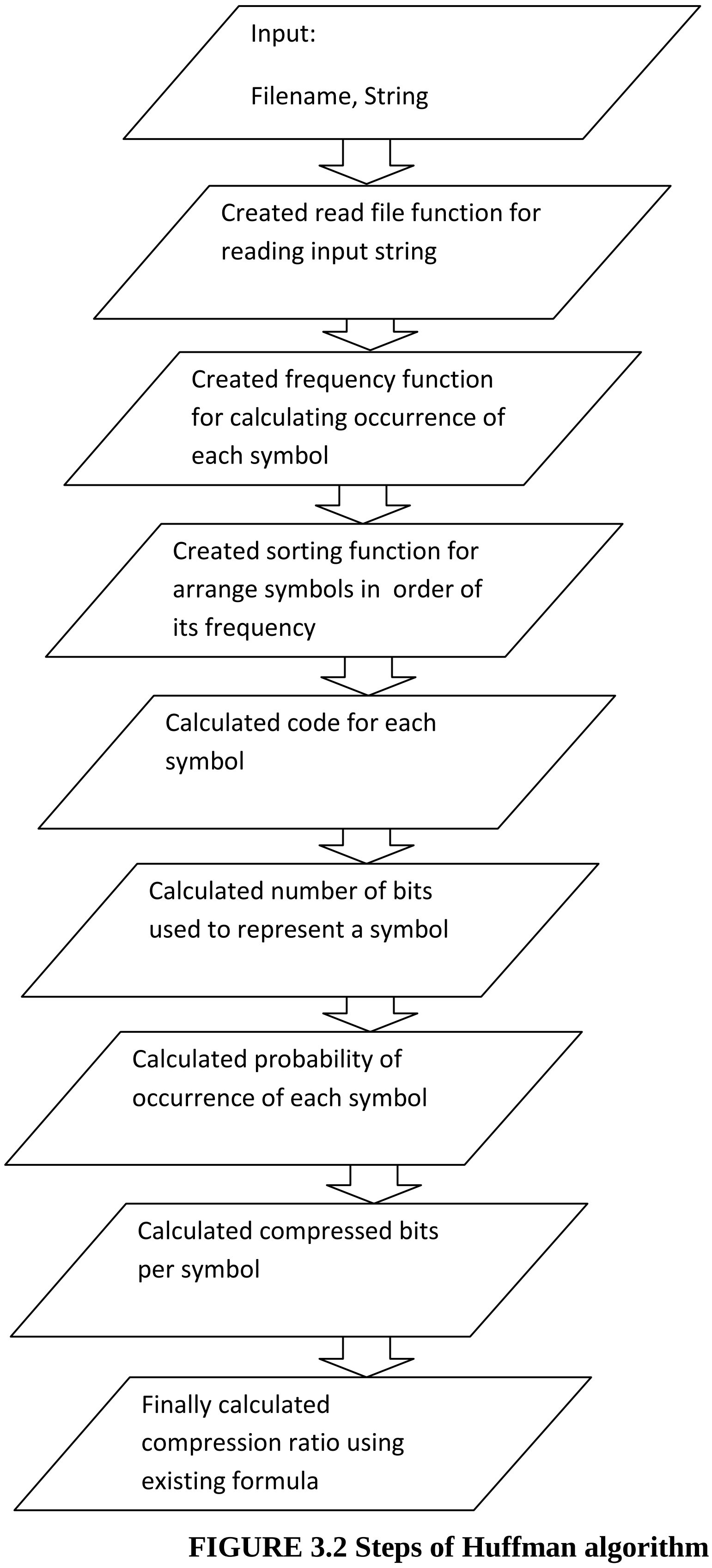

Input:
Filename, String
Created read file function for reading input string
Created frequency function for calculating occurrence of each symbol
Created sorting function for arrange symbols in order of its frequency
Calculated code for each symbol
Calculated number of bits used to represent a symbol
Calculated probability of occurrence of each symbol
Calculated compressed bits per symbol
Finally calculated compression ratio using existing formula

FIGURE 3.2 Steps of Huffman algorithm

3.3 BIT PLANE SLICING IMPLEMENTATION FLOW:

Steps to implement bit plane slicing for grey/colour /RGB images are:

- Input image with the help of MATLAB function imread() whose parameter is input image itself.
- A function rgbtogray() is used then as bit plane slicing is only possible for gray scale images.
- Representation of image with the help of figure, imshow function. This **figure function** creates a new **figure** object using default property values and imshow displays the image in a **MATLAB figure** window
- Division of various planes with the help of bitget function which has 2 parameters i.e. input image and one bit parameter and it returns the value of the bit at position bit in input image.
- Finally, Representation of bitplanes with the help of figure, imshow, logical function. Logical it will return value in the form of true or false that is value at that bit exists or not.

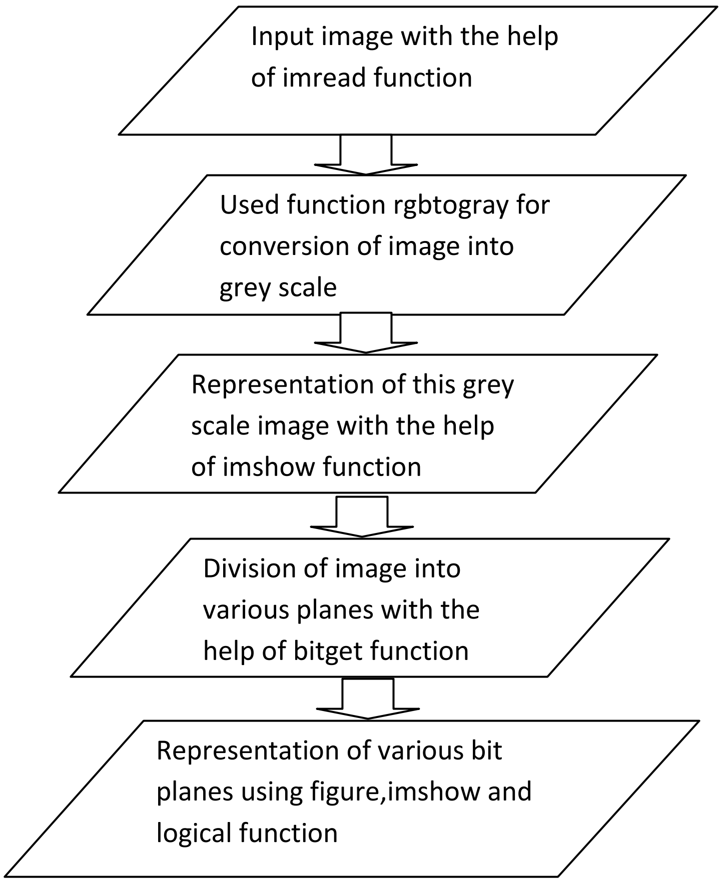

Input image with the help of imread function
Used function rgbtogray for conversion of image into grey scale
Representation of this grey scale image with the help of imshow function
Division of image into various planes with the help of bitget function
Representation of various bit planes using figure,imshow and logical function

FIGURE 3.3 Steps of bit plane slicing

3.4 IMAGE RECONSTRUCTION IMPLEMENTATION FLOW:

- Firstly an input image is shown with the help of show function.
- Then an array of zeros is created with the help of zeros function whose size is equal to that of an input image.
- Then bitset function is applied on this array and this function returns the value of input image with position **bit set** to 1 and so on and gets its value from bitget funtion.
- **uint8** function converts the elements of an array into unsigned 8-bit (1-byte) integers of class **uint8**.
- Finally the resulting image is displayed by any combination of bit planes.

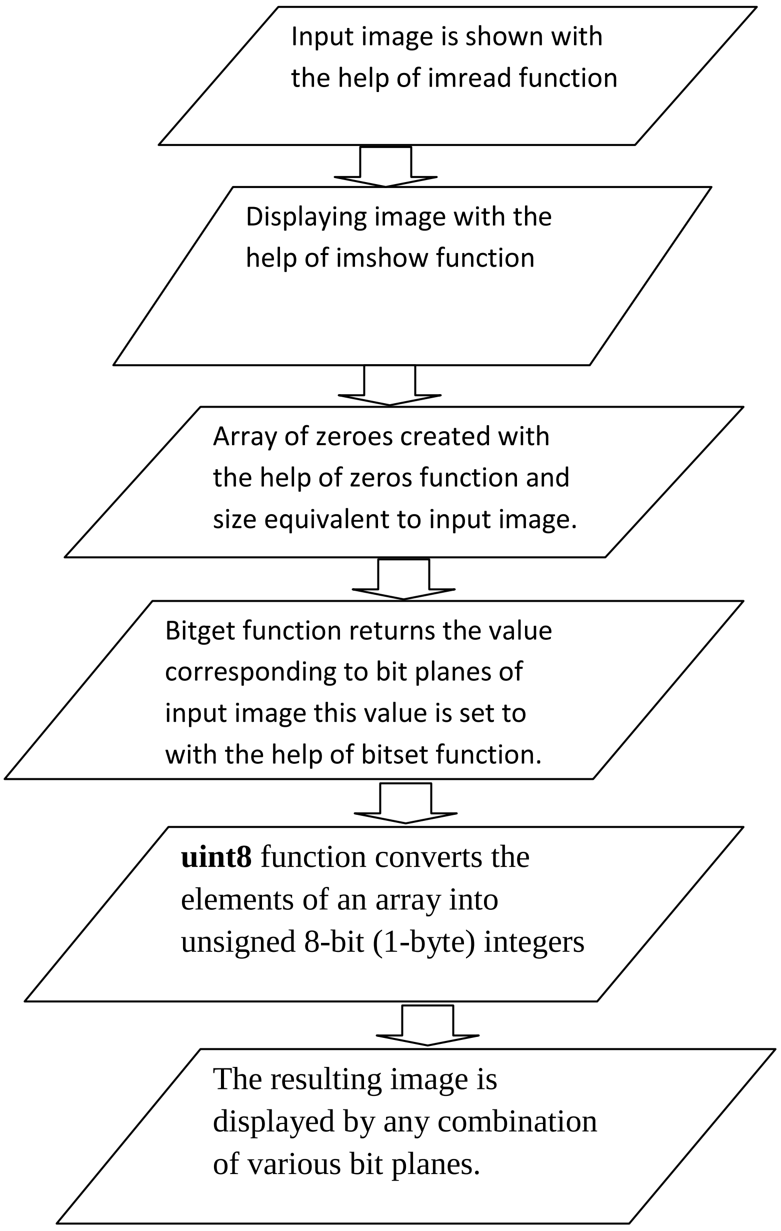

Input image is shown with the help of imread function
Displaying image with the help of imshow function
Array of zeroes created with the help of zeros function and size equivalent to input image.
Bitget function returns the value corresponding to bit planes of input image this value is set to with the help of bitset function.
uint8 function converts the elements of an array into unsigned 8-bit (1-byte) integers
The resulting image is displayed by any combination of various bit planes.

FIGURE 3.4 Steps of image reconstruction algorithm

CHAPTER 4:

CODES AND OUTPUT ILLUSTRATION:

4.1 SHANNON FANO CODE:

```
import java.io.BufferedReader;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.Writer;
import java.util.Scanner;
public class ShaFano
 {
        public static void main(String[] args) throws Exception
        {
                Scanner keyboard = new Scanner(System.in);
                String fileName;
                String dir = "C:\\shiny\\";
                System.out.print("Enter File Name: ");
                fileName = keyboard.nextLine();
                dir = dir + fileName + ".txt";
                File freq = new File(dir);
                Writer writer = new FileWriter(freq);
                Writer output = new BufferedWriter(writer);
                System.out.print("Write into file: ");
                String input = keyboard.nextLine();
                output.write(input);
                output.close();
                readFile(freq);
        }
        private static void readFile(File freq) throws Exception
        {
                FileReader read = new FileReader(freq);
                BufferedReader br = new BufferedReader(read);
                String str = br.readLine();
                frequency(str);
        }
       private static void frequency(String str)
      {
                char sentence[] = str.toCharArray();
                int len = sentence.length;
                char done[] = new char[len];
                int count[] = new int[len];
                long code[] = new long[len];
                int nob[] = new int[len];
```

```
double prob[]= new double[len];
double log[]=new double[len];
double cbits=0;
int fre = 0;
double cratio;
for(int i=0;i<len;i++)
{
        int k = 0;
        int flag=0;
        while(k!=fre)
        {
                if(sentence[i]==done[k])
                {
                        flag++;
                        break;
                }
                else
                        k++;
        }
        if(flag!=0)
        {
                count[k]++;

                continue;
        }
        else
        {
                done[fre] = sentence[i];
                count[fre]++;
                fre++;
        }
}
//----------------------------------SORTING----------------------------------
for(int inc = 0; inc<fre; inc++)
{
        int max=count[inc];
        char maxc = done[inc];
        int maxi = inc;
        int swap;
        char swapc;
        for(int inc2 = inc; inc2<fre; inc2++)
        {
                if(max<count[inc2])
                {
                        max = count[inc2];

                        maxc = done[inc2];
                        maxi = inc2;
                }
        }
```

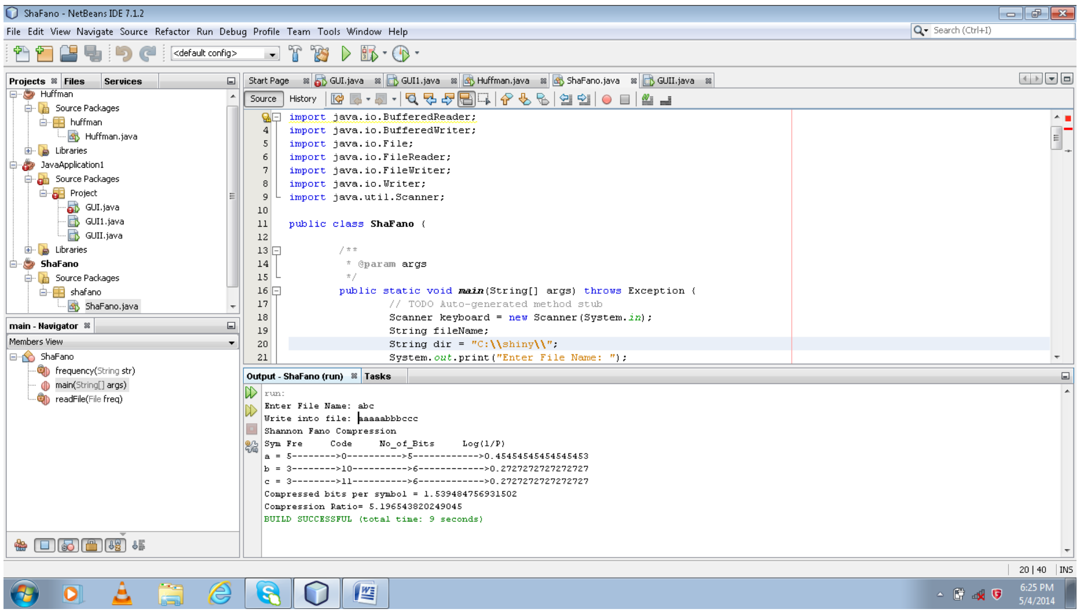
```
			swap = count[inc];
			count[inc] = max;
			count[maxi] = swap;
			swapc = done[inc];
			done[inc] = maxc;
			done[maxi] = swapc;
		}
		// --------------------------------No. of Bits Used-----------------------------------
		for(int l=0; l<fre;l++)
		{
			if(l==fre -1)
			{
				nob[l] = (l)*count[l];
			}
			else
			{
				nob[l] = (l+1) * count[l];
			}
			}
		//---------------------------------PROBABILITY-------------------------------------
		for(int l=0;l<fre;l++)
		{
			prob[l]=(double)count[l]/(double)len;
			log[l]=Math.log(1/prob[l])/Math.log(2);
		}
	//-------------------------------compressed bits per symbol--------------------------
		for(int l=0;l<fre;l++)
		{
			cbits+=(prob[l]*log[l]);
		}
		System.out.println("Shannon Fano Compression");
		System.out.println("Sym Fre	Code	No_of_Bits	Log(1/P)");

		//------------------------------------ CODE ------------------------------------------
				int a=1;
				for(int l=0;l<fre;l++)
				{
					if(l==0)
					code[l]=0;
					else
					{
						if(l==fre-1)
						{
							code[l] = code[l-1] + 1;
						}
						else
						{
							a*=10;
							code[l]=a+code[l-1];
						}
```

```
                                                            }
                                                            System.out.println(done[l]+" = " + count[l] + "-------->"
+ code[l] + "---------->" + nob[l]+ "------------>"+prob[l]);
                                                }
          System.out.println("Compressed bits per symbol = " + cbits);
            cratio=8/cbits;
            System.out.println("Compression Ratio= " + cratio);
            }
}
```

RESULT OR INFERENCE:

- Easy to implement.
- Compression ratio is more .
- Less efficient.
- Code of each symbol is easily calculated in this case.



```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.Writer;
import java.util.Scanner;

public class ShaFano {

        /**
         * @param args
         */
        public static void main(String[] args) throws Exception {
                // TODO Auto-generated method stub
                Scanner keyboard = new Scanner(System.in);
                String fileName;
                String dir = "C:\\shiny\\";
                System.out.print("Enter File Name: ");
```

```
run:
Enter File Name: abc
Write into file: aaaaabbbccc
Shannon Fano Compression
Sym Fre     Code     No_of_Bits     Log(1/P)
a = 5-------->0---------->5------------>0.4545454545454545453
b = 3-------->10---------->6------------>0.2727272727272727
c = 3-------->11---------->6------------>0.2727272727272727
Compressed bits per symbol = 1.539484756931502
Compression Ratio= 5.196543820249045
BUILD SUCCESSFUL (total time: 9 seconds)
```

FIGURE 4.1 Output snapshot of Shannon fano algorithm

4.2 HUFFMAN CODE:

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.Writer;

import java.util.Scanner;

```
public class Huffman
{
        public static void main(String[] args) throws Exception
        {
                Scanner keyboard = new Scanner(System.in);
                String fileName;
                String dir = "C:\\shiny\\";
                System.out.print("Enter File Name: ");
                fileName = keyboard.nextLine();
                dir = dir + fileName + ".txt";
                File freq = new File(dir);
                Writer writer = new FileWriter(freq);
                Writer output = new BufferedWriter(writer);
                System.out.print("Write into file: ");
                String input = keyboard.nextLine();
                output.write(input);
                output.close();
                readFile(freq);
      }
        private static void readFile(File freq) throws Exception
      {
                FileReader read = new FileReader(freq);
                BufferedReader br = new BufferedReader(read);
                String str = br.readLine();
                frequency(str);
        }
        private static void frequency(String str)
       {
               char sentence[] = str.toCharArray();
```

```
int len = sentence.length;
char done[] = new char[len];
int count[] = new int[len];
long code[] = new long[len];
int nob[] = new int[len];
double prob[]= new double[len];
double log[]=new double[len];
double cbits=0;
int fre = 0;
double cratio;
for(int i=0;i<len;i++)
{
        int k = 0;
        int flag=0;
        while(k!=fre)
                {
                if(sentence[i]==done[k])
                {
                        flag++;
                        break;
                }
                else
                        k++;
        }
        if(flag!=0)
        {
                count[k]++;
                continue;
        }
        else
        {
                done[fre] = sentence[i];
                count[fre]++;
                fre++;
        }
}
//----------------------------------SORTING----------------------------------
for(int inc = 0; inc<fre; inc++)
{
        int max=count[inc];
        char maxc = done[inc];
        int maxi = inc;
        int swap;
        char swapc;
        for(int inc2 = inc; inc2<fre; inc2++)
        {
        if(max<count[inc2])
                {
                        max = count[inc2];
                        maxc = done[inc2];
                        maxi = inc2;
                }
        }
        swap = count[inc];
        count[inc] = max;
        count[maxi] = swap;
        swapc = done[inc];
        done[inc] = maxc;
        done[maxi] = swapc;
}
// --------------------------------No. of Bits Used-----------------------------------
for(int l=0; l<fre;l++)
{
        if(l==fre -1)
        {
        nob[l] = (l)*count[l];
        }
        else
        {
                nob[l] = (l+1) * count[l];
        }
}
//----------------------------------PROBABILITY--------------------------------------
for(int l=0;l<fre;l++)
{
        prob[l]=(double)count[l]/(double)len;
        log[l]=Math.log(1/prob[l])/Math.log(2);
}

//-------------------------------compressed bits per symbol---------------------------
for(int l=0;l<fre;l++)
{
        cbits+=(prob[l]*nob[l]);
}



System.out.println("Huffman Compression");
```

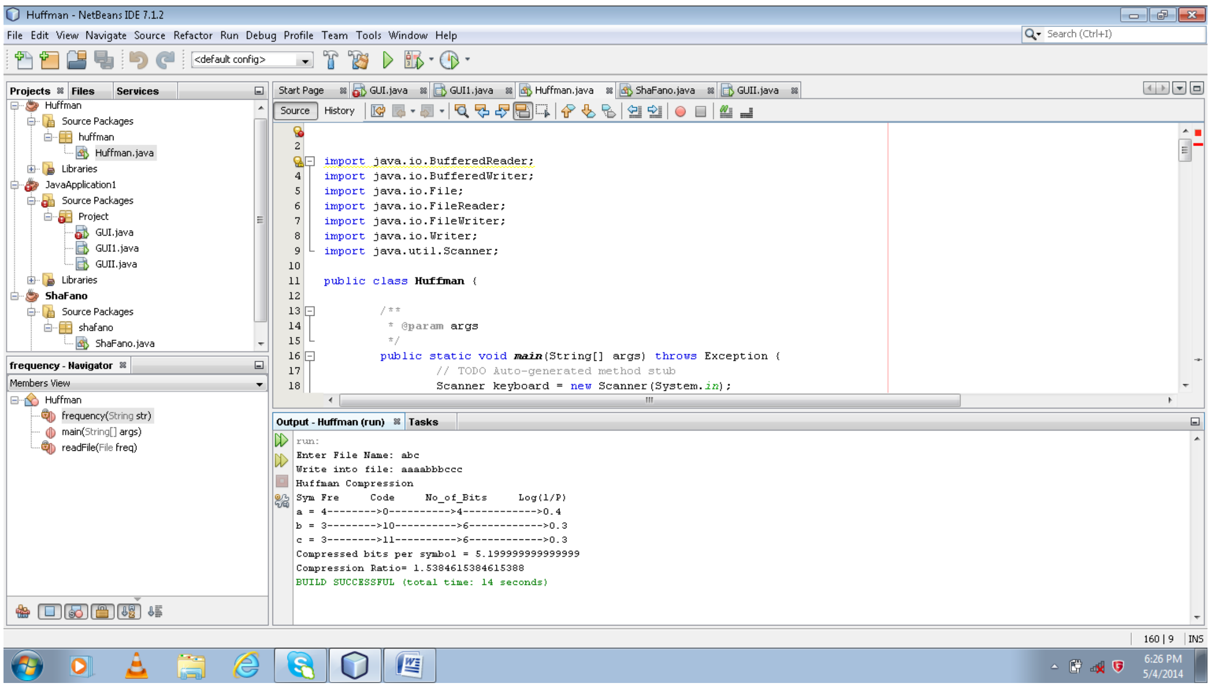
```
                System.out.println("Sym Fre     Code     No_of_Bits     Log(1/P)");
                //------------------------------------ CODE -----------------------------------------
                        int a=1;
                        for(int l=0;l<fre;l++)
                        {
                            if(l==0)
                                code[l]=0;
                            else
                            {
                                if(l==fre-1)
                                {
                                    code[l] = code[l-1] + 1;


                            }


                            else
                                {
                                    a*=10;
                                    code[l]=a+code[l-1];
                               }
                        }
                            System.out.println(done[l]+" = " + count[l] + "-------->"
+ code[l] + "---------->" + nob[l]+ "------------>"+prob[l]);
                        }
            System.out.println("Compressed bits per symbol = " + cbits);
            cratio=8/cbits;
            System.out.println("Compression Ratio= " + cratio);
            }}
```

RESULT OR INFERENCE:

- Easy to implement.
- Compression ratio is more.
- Less efficient.
- Code of each symbol is easily calculated in this case.



```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.Writer;
import java.util.Scanner;

public class Huffman {

        /**
         * @param args
         */
        public static void main(String[] args) throws Exception {
                // TODO Auto-generated method stub
                Scanner keyboard = new Scanner(System.in);
```

```
run:
Enter File Name: abc
Write into file: aaaabbbccc
Huffman Compression
Sym Fre     Code     No_of_Bits     Log(1/P)
a = 4-------->0---------->4------------>0.4
b = 3-------->10---------->6------------>0.3
c = 3-------->11---------->6------------>0.3
Compressed bits per symbol = 5.199999999999999
Compression Ratio= 1.5384615384615388
BUILD SUCCESSFUL (total time: 14 seconds)
```

FIGURE 4.2 Output snapshot of Huffman algorithm

4.3 BIT PLANE SLICING CODE IN MATLAB:

```
clc
c=imread('C:\Users\Dell\Desktop\download.jpg');
%figure,imshow(c);
d=rgb2gray(c);
%figure,imshow(d);
B=bitget(d,1);
figure,imshow(logical(B));
title(' plane 1');
B=bitget(d,2);
figure,imshow(logical(B));
title(' plane 2');
B=bitget(d,3);
figure,imshow(logical(B));
title(' plane 3');
B=bitget(d,4);
figure,imshow(logical(B));
title(' plane 4');
B=bitget(d,5);
```

```
figure,imshow(logical(B));
title(' plane 5');
B=bitget(d,6);
figure,imshow(logical(B));
title(' plane 6');
B=bitget(d,7);
figure,imshow(logical(B));
title(' plane 7');
B=bitget(d,8);
figure,imshow(logical(B));
title(' plane 8');
```

<u>RESULT OR INFERENCE</u>:

- Good and ample amount of knowledge of MATLAB is required
- Little bit of difficulty in handling the bit planes.
- Difficulty in understanding some of MATLAB functions.
- Overall this algorithm used for implementation is easy.

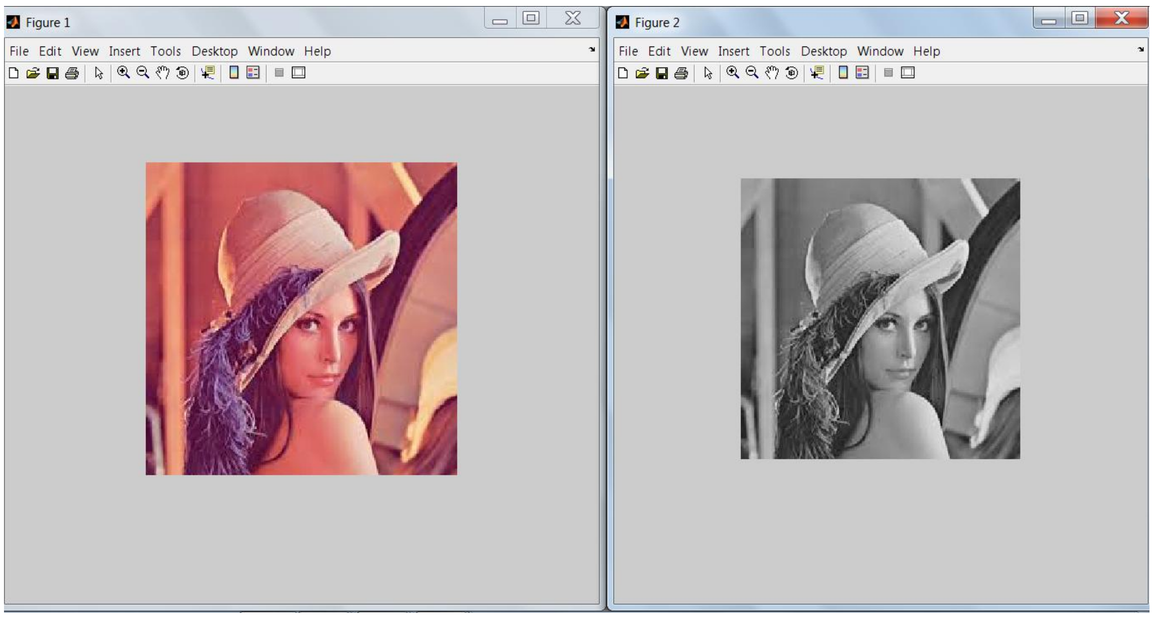


Figure 1
File Edit View Insert Tools Desktop Window Help
Figure 2
File Edit View Insert Tools Desktop Window Help

FIGURE 4.3 Output Snapshot of conversion from rgb to grey scale

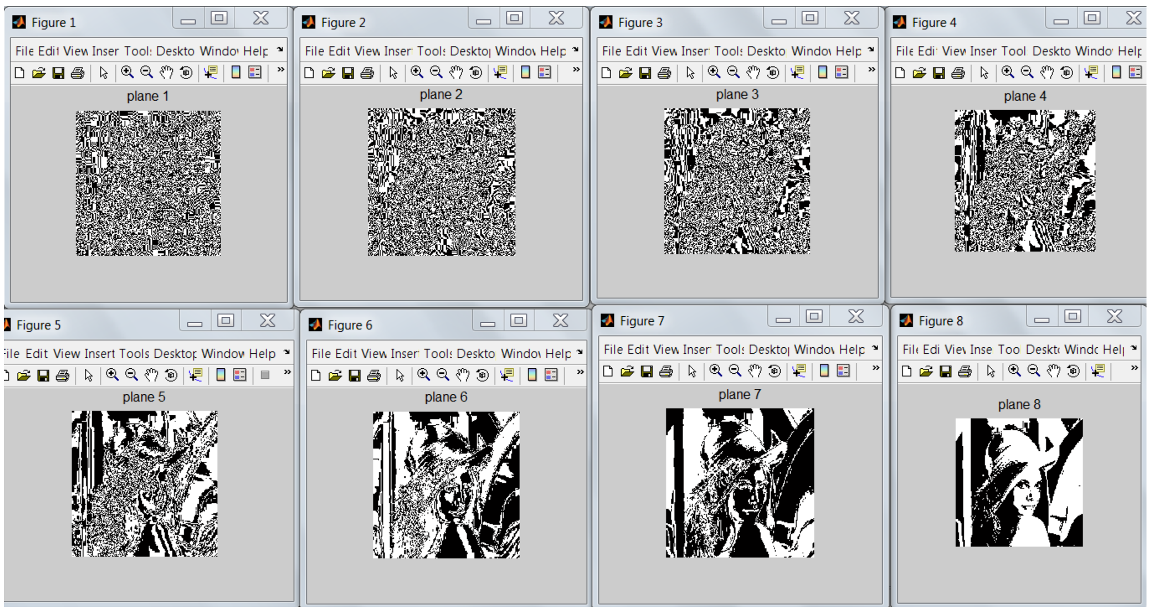


FIGURE 4.4 Output Snapshot of Representation of various Bit Planes

4.4 IMAGE RECONSTRUCTION CODE IN MATLAB:

```
clc
c=imread('C:\Users\Dell\Desktop\download.jpg');
figure,imshow(c);
title('Original Image');
B=zeros(size(c));
B=bitset(B,8,bitget(c,8));
B=bitset(B,7,bitget(c,7));
B=bitset(B,6,bitget(c,6));
B=bitset(B,5,bitget(c,5));
%B=bitset(B,4,bitget(c,4));
%B=bitset(B,3,bitget(c,3));
%B=bitset(B,2,bitget(c,2));
%B=bitset(B,1,bitget(c,1));
B=uint8(B);
figure,imshow(B);
title('Compressed Image');
```

<u>RESULT OR INFERENCE</u>:

- Good and ample amount of knowledge of MATLAB is required
- Little bit of difficulty in handling the bit planes.
- Difficulty in understanding some of MATLAB functions.
- Overall this algorithm used for implementation is easy.

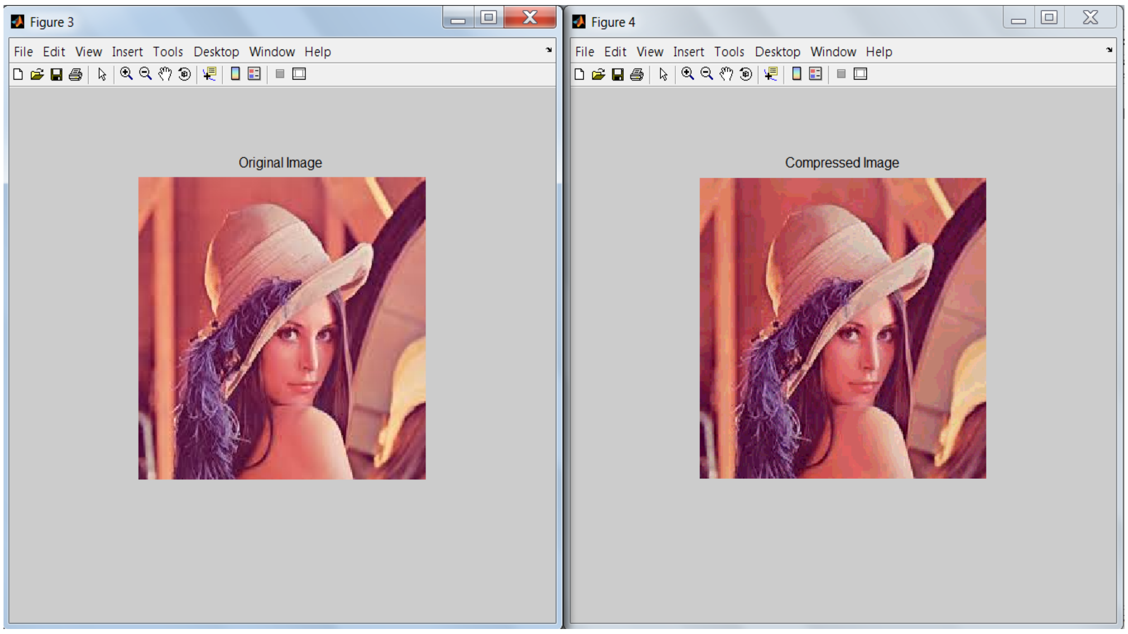


Figure 3
File Edit View Insert Tools Desktop Window Help
Original Image
Figure 4
File Edit View Insert Tools Desktop Window Help
Compressed Image

FIGURE 4.5 Output Snapshot of image reconstruction using four bit planes

CHAPTER 5:

5.1 CONCLUSION AND FUTURE SCOPE

By encoding characters using **EBCDIC** or **ASCII**, I clearly am not going to be very close to an optimum method. Since every character is encoded using the same number of bits, I introduce lots of error in both directions, with most of the codes in a message being too long and some being too short.

Solving this coding problem in a reasonable manner was one of the first problems tackled by practitioners of Information Theory. Two approaches that worked well were **Shannon- Fano coding** and **Huffman coding**—two different ways of generating **variable-length codes** when given a probability table for a given set of symbols.

Huffman coding, named for its inventor D.A. Huffman, achieves the minimum amount of redundancy possible in a fixed set of variable-length codes. This doesn't mean that Huffman coding is an optimal coding method. It means that it provides the best approximation for coding symbols when using fixed-width codes. The problem with **Huffman** or **Shannon-Fano** coding is that they use an **integral number of bits** in each code. If the **entropy** of a given character is **2.5 bits**, the Huffman code for that character must be either **2 or 3 bits**, **not 2.5**. Because of this, Huffman coding can't be considered an optimal coding method, but it is the best approximation that uses fixed codes with an integral number of bits. Here are our future objectives:

DATA ENCRYPTION:

Data Encryption is the process of encoding messages or information in such a way that only authorized parties could read it. Encryption doesn't prevent hacking but it reduces the likelihood that the hacker will be able to read the data that is encrypted.

DATA DECRYPTION:

Data Decryption is the reverse operation of encryption. For secret-key encryption, you might know both the key and IV that were used to encrypt the data. For public-key encryption, you must know either the public key (if the data was encrypted using the private key) or the private key (if the data was encrypted using the public key).

IMAGE ENCRYPTION:

Image Encryption is the conversion of image into a form, called a cipher text , that cannot be easily understood by unauthorized person.

IMAGE DECRYPTION:

Image Decryption is the process of converting encrypted image back into its original form, so it could be understood.

REFERENCES

1. Mark Nelson and Jean-loup Gailly, The Data Compression Book, Second edition, M&T books,

2. A Compression & Encryption Algorithm on DNA Sequences Using Dynamic Look up Table and Modified Huffman Techniques Published Online September 2013 in MECS (http://www.mecs-press.org/)

3. D.E.Knuth Dynamic Huffman coding J.Algorithms, 1985, pp.163-180.

4. R.C. Gonzalez and R.E. Woods, “Digital Image Processing” , Second edition Pearson Prentice Hall, 2008.

5. Anil K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, 1989.

6. William K. Pratt & John Wile, *Digital Image Processing*, 3rd Edition, 2001.