Chapter 1

INTRODUCTION

1.1 Compression Technique

1.1.1 Lossless Algorithm

1.1.2 Lossy algorithms

1.2 Motivation

1.3 Goals of the Thesis

1.4 Organization of the Thesis

Introduction

Compression is used just about everywhere in the real world. All the data or images found on the web are compressed, typically in the JPEG or GIF formats for images and zip or gzip for files and image both, most modems use compression and several file systems automatically compress files when stored, and the rest of us do it by hand.

Data compression is often referred to as coding, where coding is a very general term encompassing any special representation of data which satisfies a given need. Information theory is defined to be the study of efficient coding and its consequences, in the form of speed of transmission and probability of error. Data compression may be viewed as a branch of information theory in which the primary objective is to minimize the amount of data to be transmitted. The purpose of this report is to present and analyze a variety of data compression algorithms and giving an idea of a new compression algorithm.

The job of compression consists of two parts

1. Encoding
2. Decoding

First an *encoding* algorithm that takes data that is to be compressed and generates a “compressed” form, and a *decoding* algorithm that reconstructs the original data or some approximation of it from the compressed representation. These two parts are typically tied together since they both have to understand the shared compressed representation.

A simple characterization of data compression is that it involves transforming a string of characters in some representation (such as ASCII) into a new string (of bits) which contains the same information but whose length is as small as possible. Data compression has important application in the areas of data transmission and data storage. Many data processing applications require storage of large volumes of data, and the number of such applications is constantly increasing as the use of computers extends to new disciplines. At the same time, the proliferation of computer communication networks is resulting in massive transfer of data over communication links. Compressing data to be stored or transmitted reduces storage and/or communication costs. When the amount of data to be transmitted is reduced, the effect is that of increasing the capacity of the communication channel. Similarly, compressing a file to half of its original size is equivalent to doubling the capacity of the storage medium. It may then become feasible to store the data at a higher, thus faster, level of the storage hierarchy and reduce the load on the input/output channels of the computer system.

**1.1 Compression Technique**

We can divide further data compression algorithms into

* Lossless algorithm
* Lossy algorithm

**1.1.1 Lossless Algorithm**

The algorithms, which can reconstruct the original message exactly from the compressed message, are called lossless algorithms. Lossless algorithms are typically used where we do not want to lose our data in any respect. If we want to compress our text data by using a lossy algorithm then it makes our data unreadable or having loss of information or it may corrupt after the reconstruction. This is the reason which makes us to use only lossless algorithm in compressing and decompressing the text data to fully recover. We can also compress images or movie or sound clips by these lossless algorithms.

**1.1.2 Lossy algorithms**

Which can only reconstruct an approximation of the original message and used for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable.

Lossy is used in an abstract sense, however, and does not mean random lost pixels, but instead means loss of a quantity such as a frequency component, or perhaps loss of noise. For example, one might think that lossy text compression would be unacceptable because they are imagining missing or switched characters.

Consider instead a system that reworded sentences into a more standard form, or replaced words with synonyms so that the file can be better compressed. Technically the compression would be lossy since the text has changed, but the “meaning” and clarity of the message might be fully maintained, or even improved. In fact Strunk and White might argue that good writing is the art of lossy text compression.

All compression algorithms must assume that there is some bias on the input messages so that some inputs are more likely than others, *i.e.* that there is some unbalanced probability distribution over the possible messages. Most compression algorithms base this “bias” on the structure of the messages – *i.e.*, an assumption that repeated characters are more likely than random characters, or that large white patches occur in “typical” images. Compression is therefore all about probability.

From the time when the compression algorithm came into existence till now, we have developed many lossless and lossy algorithms. A question about compression algorithms is how one judges the quality of one versus another.

In the case of lossless compression there are several criteria I can think of, the time to compress, the time to reconstruct, the size of the compressed messages, and the generality—*i.e.*, does it only work on Shakespeare or does it do Byron too. In the case of lossy compression the judgement is further complicated since we also have to worry about how good the lossy approximation is. There are typically tradeoffs between the amount of compression, the runtime, and the quality of the reconstruction. Depending on your application one might be more important than another and one would want to pick your algorithm appropriately.

**1.2 Motivation**

As discussed in the Introduction, data compression has wide application in terms of information storage, including representation of the abstract data type string and file compression. Huffman coding is used for compression in several file archival systems, as is Lempel-Ziv coding. An adaptive Huffman coding technique is the basis for the compact command of the UNIX operating system, and the UNIX compress utility employs the Lempel-Ziv approach.

In the area of data transmission, Huffman coding has been passed over for years in favour of block-block codes, notably ASCII. The advantage of Huffman coding is in the average number of bits per character transmitted, which may be much smaller than the log n bits per character (where n is the source alphabet size) of a block-block system. The primary difficulty associated with variable-length codeword is that the rate at which bits are presented to the transmission channel will fluctuate, depending on the relative frequencies of the source messages. This requires buffering between the source and the channel. Advances in technology have both overcome this difficulty and contributed to the appeal of variable-length codes. Current data networks allocate communication resources to sources on the basis of need and provide buffering as part of the system. These systems require significant amounts of protocol, and fixed-length codes are quite inefficient for applications such as packet headers. In addition, communication costs are beginning to dominate storage and processing costs, so that variable-length coding schemes which reduce communication costs is attractive even if they are more complex. For these reasons, one could expect to see even greater use of variable-length coding in the future.

It is interesting to note that the Huffman coding algorithm, originally developed for the efficient transmission of data, also has a wide variety of applications outside the sphere of data compression. These include construction of optimal search trees, list merging and generating optimal evaluation trees in the compilation of expressions. Additional applications involve search for jumps in a monotone function of a single variable, sources of pollution along a river, and leaks in a pipeline. The fact that this elegant combinatorial algorithm has influenced so many diverse areas underscores its importance.

**1.3 Goals of the Thesis**

As we can see that there is very large amount of data is available in the real world. That uses different storing mechanism to store this data. But this is not the end. As the time keep changing lots of data is going to created, destroyed or manipulated. Managing is one part of this data. The other one is where to and how to store this data. This require a large number of devices to store or the other way is to improve this data so that the data is free from the ambiguity and having well managed compressed form. But the algorithm developed is still not up to the mark to reduce the hardware cost. This indicate that the improvement is certainly needed somewhere in the development process or techniques and compression algorithms.

This thesis recognizes the *data compression* as the area of the development process which needs more attention. So, the first and the foremost goal of this thesis is to develop an algorithm that can generate very highly compressed form of the data and also to make a software application program to achieve this goal. This goal has been achieved by experimentation with a completely new representation technique: power of three representations.

But this is the broad way of looking at it. Data compression can be thought of as mapping. In case of lossless data compression one to one mapping is applied. The algorithm presented in this thesis is also a lossless data compression algorithm. This algorithm tries to use this mapping process in efficient way to reduce file size.

One of the important goals of this thesis work has also been to develop a tool or software application to fulfil the first goal i.e. to compress the data. Moreover, in future this work may be extended so as to improve the compression. This work may encourage future researches in this direction.

**1.4 Organization of the Thesis**

The organization of this thesis as follows

Chapter 1: As this chapter gives the brief information about the data compression and why it is use and what is the purpose of it.

Chapter 2: This chapter presents the historical work carried out in this area and the state of the art in data compression. All the work described in this chapter is related to the problem which is discussed in the present thesis.

Chapter 3: This chapter contains the information about newly designed/proposed data compression algorithm. This also shows how the data compression applies and how it can be coded into the program.

Chapter 4: This chapter presents the results comes from the software application program of this compression algorithm.

Chapter 5: This chapter presents the conclusion of the thesis.

Chapter 6: In this chapter we have discusses the possibilities and scope of the current work in the future and tells how the current work can be extended in future research.

Chapter 7: enlists the references used throughout the thesis.

Chapter 2

LITERATURE SURVEY

2.1 Data compression Terms

2.2 Categorization of Data compression algorithm

2.3 Entropy

2.4 Complexity

2.5 Previously Defined algorithms

## 2.5.1 Semantic dependent Method

2.5.1.1 Run length coding

2.5.1.2 Difference mapping

2.5.2 Dictionary substitution

### 2.5.2.1 Lempel-ZivWelch Codes

2.5.3 Entropy Coding

### 2.5.3.1 Shannon-Fano Coding

2.5.3.2 Static Huffman Coding

2.5.3.3 Adaptive Huffman Coding

### 2.5.3.4 Arithmetic Coding

2.5.3.5 Range encoding

2.6 Comparison between different compression algorithms

Literature Survey

The goal of data compression is to reduce redundancy, encoding only the informational content. The data to be compressed can be text-like binary data, audio, image and video files. Compression can be largely classified either lossy or lossless. Lossless data compression allows compressed data to be reconstructed back to the original one at the receiver’s end without losing any information. Data compression is also converting an inefficient form into more efficient form. But this basically used when we talk about the data base where we do not want our data to have multiple copies or other type of redundancies.

Or we can say “Data compression is the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output, the bit stream, or the compressed stream) that has less size as compared to the original” [1].



Fig 2.1: Compression and Reconstruction

In general, compression is done by encoding the symbol in the message or the entire message is replaced with code words which are usually shorter in length compared to the original one. Decompression is done by interpreting the code words to retrieve the original message. In some compression algorithms, symbols which occurred more frequently will be encoded in shorter code words and those that are less frequently will be encoded in longer code words, thus achieving better compression ratio. The field of data compression is called as source coding [1].

Data compression became so important that some researchers have proposed the “simplicity” and “power” (SP) theory [1]. This describes that all computing is compression. It describes Data compression may be interpreted as a process of removing unnecessary complexity or redundancy in information, and thereby maximizing simplicity while preserving as much as possible of its actual meaning. This theory is mainly based on the following conjectures:

All kinds of computing usefully understood as information compression by pattern matching, unification, and search. The process of finding redundancy and removing it may always be understood at a fundamental level as a process of searching for patterns that match each other, and merging or unifying repeated instances of any pattern to make one [1].

**2.1 Data compression Terms:**

Here we are going to discuss important data compression terms.

Encoder:

The *encoder* is the program which compresses the raw data of the input file and creates an output file with compressed (low-redundancy) data having less output file size.

Decoder:

The *decoder* converts in the opposite direction. The decoder is the program that decompresses the compressed file and creates an exact copy of the original input file.

Non adaptive compression:

A *non-adaptive* compression method is rigid or static that does not modify its operations, its parameters, or its tables in response to the particular data being compressed. Such a method is best used to compress data that is all of a single type. They are specifically designed for facsimile compression and would do a poor job compressing any other data.

Adaptive Compression:

An *adaptive* compression method is dynamic*.* This method examines the raw data and modifies its operations and/or its parameters accordingly. For example Huffman coding uses adaptive compression.

Semi Adaptive compression:

Some compression methods use a 2-pass or 2 phase algorithm, where the first pass reads the input file to collect statistics on the data to be compressed, and the second pass does the actual compressing using parameters set by the first pass. Such a method may be called *semi adaptive*.

Locally adaptive:

A data compression method can also be *locally adaptive*, meaning it adapts itself to local conditions in the input stream and varies this adaptation as it moves from area to area in the input.

Lossy/lossless compression:

We can find compression methods into two flavours. First one is lossy and the other one is lossless. Lossy compression has compression by losing some information. When the compressed stream is decompressed, the result is not identical to the original data stream. Such a method mainly used in compressing images, movies, or sounds. If the loss of data is small, we may not be able to tell the change.

ON the other way, in text files, especially files containing computer programs may become useless if even one bit gets modified. Such files should be compressed only by a lossless compression method.

Cascaded compression/pipeline of compression:

The difference between lossless and lossy methods can be illuminated by considering a cascade of compressions. Imagine a data file *A* that has been compressed by an encoder *X*, resulting in a compressed file *B*. It is possible, although pointless, to pass *B* through another encoder *Y*, to produce a third compressed file *C*. The point is that if methods *X* and *Y* are lossless, then decoding *C* by *Y* will produce an exact *B*, which when decoded by *X* will yield the original file *A*. However, if any of the compression algorithms is lossy, then decoding *C* by *Y* may produce a file *B\_* different from *B*. Passing *B\_* through *X* may produce something very different from *A* and may also result in an error, because *X* may not be able to read *B\_*.

Z

X

Y

W

Uncompressed compressed

File file

 Fig2.2: Compression in cascade **[3]**

X’

Z’

W’

Y’

Compressed file original file

 Uncompressed

 Fig2.3: Decompression in cascade **[3]**

Perceptive compression:

A lossy encoder deletes only, data whose absence would not be detected by our senses. Such an encoder must therefore employ algorithms based on our understanding of psychoacoustic and psychovisual perception, so it is often referred to as a perceptive encoder. Such an encoder can be made to operate at a constant compression ratio, where for each *x* bits of raw data; it outputs *y* bits of compressed data. This is convenient in cases where the compressed stream has to be transmitted at a constant rate. The trade-off is a variable subjective quality. Parts of the original data that are difficult to compress may, after decompression, look (or sound) bad. Such parts may require more than *y* bits of output for *x* bits of input.

Symmetric Compression:

*Symmetrical compression* is the case where the compressor and decompressor use basically the same algorithm but work in “opposite” directions. Such a method makes sense for general work, where the same number of files is compressed as is decompressed.

Asymmetric Compression:

In an asymmetric compression method, either the compressor or the decompressor may have to work significantly harder. Such methods have their uses and are not necessarily bad. A compression method where the compressor executes a slow, complex algorithm and the decompressor is simple is a natural choice when files are compressed into an archive, where they will be decompressed and used very often. The opposite case is useful in environments where files are updated all the time and backups are made. There is a small chance that a backup file will be used, so the decompressor isn’t used very often.

A data compression method is called *universal* if the compressor and decompressor do not know the statistics of the input stream. A universal method is *optimal* if the compressor can produce compression factors that asymptotically approach the entropy of the input stream for long inputs.

File Differencing:

The term *file differencing* refers to any method that locates and compresses the differences between two files. Imagine a file *A* with two copies that are kept by two users. When a copy is updated by one user, it should be sent to the other user, to keep the two copies identical. Instead of sending a copy of *A*, which may be big, a much smaller file containing just the differences, in compressed format, can be sent and used at the receiving end to update the copy of *A*.

Streaming mode:

Most compression methods operate in the *streaming mode*, where the compression/decompression inputs a byte or several bytes, processes them, and continues until an end-of-file is sensed.

Block mode:

Some methods, work in the *block mode*, where the input stream is read block by block and each block is encoded separately. The block size in this case should be a user-controlled parameter, since its size may greatly affect the performance of the method.

Physical and logical method:

Most compression methods are *physical*. They look only at the bits in the input stream and ignore the meaning of the data items in the input (e.g., the data items may be words, pixels, or audio samples). Such a method translates one bit stream into another, shorter, one. The only way to make sense of the output stream (to decode it) is by knowing how it was encoded. Some compression methods are *logical*. They look at individual data items in the source stream and replace common items with short codes.

The probability model:

This concept is important in statistical data compression methods. In such a method, a model for the data has to be constructed before compression can begin. A typical model may be built by reading the entire input stream counting the number of times each symbol appears (its frequency of occurrence), and computing the probability of occurrence of each symbol. The data stream is then input again, symbol by symbol, and is compressed using the information in the probability model.

Reading the entire input stream twice is slow, which is why practical compression methods use estimates, or adapt themselves to the data as it is being input and compressed. It is easy to scan large quantities of, say, English text and calculate the frequencies and probabilities of every character. This information can later serve as an approximate model for English text and can be used by text compression methods to compress any English text. It is also possible to start by assigning equal probabilities to all the symbols in an alphabet, then reading symbols and compressing them, and, while doing that, also counting frequencies and changing the model as compression progresses. This is the principle behind *adaptive compression methods*.

Shannon established that there is a fundamental limit to lossless data compression. This limit, called the entropy rate, is denoted by *H*. The exact value of *H* depends on the statistical nature of the source data. It is possible to compress the source, in a lossless manner, with compression rate close to *H*. It is mathematically impossible to do better than *H*.

**2.2 Categorization of Data compression algorithm**

Data compression can be generalized into several categories. For simplicity, we will classify the compression algorithms into several categories based on the different types of data encoding used.

Data has to be encoded into code words which are usually shorter in length than the original ones. Several types of encoding are discussed below:

* **Entropy based encoding**: Shannon introduced the idea of entropy in information theory. Entropy encoding assigns code words to symbols so as to match code lengths with the probabilities of the symbols. The most common symbols use the shortest codes **[4].**
* **Dictionary based encoding**: Operate by searching for matches between the text to be compressed and a set of strings contained in a data structure (called the 'dictionary') that is maintained by the encoder. When the encoder finds such a match, it substitutes a reference to the string's position in the data structure **[4].**
* **Others**: Data coding which fall into this category usually have their own unique way to code the symbols or they are able to use either of the above 2 classes to encode the symbol (varies with different implementations) [**4].**

**2.3 Entropy**

Shannon borrowed the definition of *entropy* from statistical physics to capture the notion of how much information is contained in and their probabilities. For a set of possible messages S, Shannon defined entropy as [13],



Where p(s) is the probability of message s. The definition of Entropy is very similar to that in statistical physics—in physics S is the set of possible states a system can be in and p(s) is the probability the system is in state s. We might remember that the second law of thermodynamics basically says that the entropy of a system and its surroundings can only increase. Getting back to messages, if we consider the individual messages s ∈ S, Shannon defined the notion of the *self information* of a message as



This self information represents the number of bits of information contained in it and, roughly speaking, the number of bits we should use to send that message. The equation says that messages with higher probability will contain less information.

The entropy is simply a weighted average of the information of each message, and therefore the average number of bits of information in the set of messages [13]. Larger entropies represent more information, and perhaps counter-intuitively, the more random a set of messages (the more even the probabilities) the more information they contain on average.

**2.4 Complexity**

The algorithm can be defined on the basis of complexity of time and space. The algorithm is the best one if it returns highly efficient code on the expense of affordable time complexity. If the algorithm takes too much time to compress data then it is of no use. Also if in no time an algorithm compresses the data but there is no significant compression occurs then it is also of very less use. By making highly data compressing algorithm we can easily maximises the network channel capacity.

**2.5 Previously Defined algorithms**

## **2.5.1 Semantic dependent methods**

Semantic dependent data compression techniques are designed to respond to specific types of local redundancy occurring in certain applications. One area in which data compression is of great importance is image representation and processing. There are two major reasons for this. The first is that digitized images contain a large amount of local redundancy. An image is usually captured in the form of an array of pixels whereas methods which exploit the tendency for pixels of like colour or intensity to cluster together may be more efficient. The second reason for the abundance of research in this area is volume. Digital images usually require a very large number of bits, and many uses of digital images involve large collections of images.

2.5.1.1 Run length coding

If a data item *d* occurs *n* consecutive times in the input stream, replace the *n* occurrences with the single pair *nd*. “The *n* consecutive occurrences of a data item are called a *run length* of *n*, and this approach to data compression is called *run-length encoding* or RLE” [1].

In a common version of run length encoding, the sequence of image elements along a scan line (row) is mapped into a sequence of pairs (c, l) where c represents an intensity or colour and l the length of the run (sequence of pixels of equal intensity). For pictures such as weather maps, run length encoding can save a significant number of bits over the image element sequence. An image can be scanned in different ways. Some of them are shown below in the figure.

 (a) (b)



(c)

Fig2.4: Scan used in the run length coding for images

In the figure 2.4 (a) scan is done horizontally. When it reaches the end of horizontal line then it starts scan from the left most pixels. In the figure 2.4 (b), the scan is done in the zig-zag form as shown below. In the last figure 2.4 (c), it shows a number on last of the pixel line. This shows the line sequence number which is calculated manually or by the algorithm which decides to compress the image in more compact form.

2.5.1.2 Difference mapping

Another data compression technique specific to the area of image data is difference mapping, in which the image is represented as an array of differences in brightness (or colour) between adjacent pixels rather than the brightness values themselves. Difference mapping was used to encode the pictures of Uranus transmitted by Voyager 2. The 8 bits per pixel needed to represent 256 brightness levels was reduced to an average of 3 bits per pixel when difference values were transmitted. In spacecraft applications, image fidelity is a major concern due to the effect of the distance from the spacecraft to earth on transmission reliability.

Data compression is of interest in business data processing, both because of the cost savings it offers and because of the large volume of data manipulated in many business applications. The types of local redundancy present in business data files include runs of zeros in numeric fields, sequences of blanks in alphanumeric fields, and fields which are present in some records and null in others. Run length encoding can be used to compress sequences of zeros or blanks. Null suppression may be accomplished through the use of presence bits. Another class of methods exploits cases in which only a limited set of attribute values exist.

**2.5.2 Dictionary substitution**

Statistical compression methods use a statistical model of the data, which is why the quality of compression they achieve depends on how good that model is. Dictionary based compression methods do not use a statistical model, nor do they use variable-size codes. In many applications, the output of the source consists of recurring patterns. A classic example is a text source in which certain patterns or words recur constantly. Also, there are certain patterns that simply do not occur, or if they do, occur with great rarity [2].

Dictionary substitutionentails replacing alphanumeric representations of information such as bank account type, insurance policy type, sex, month, etc. by the few bits necessary to represent the limited number of possible attribute values.

A variety of approaches to data compression designed with text files in mind include use of a dictionary either representing all of the words in the file so that the file itself is coded as a list of pointers to the dictionary, or representing common words and word endings so that the file consists of pointers to the dictionary and encodings of the less common words. Hand-selection of common phrases, programmed selection of prefixes and suffixes and programmed selection of common character pairs has also been investigated.

### 2.5.2.1 Lempel-ZivWelch Codes

The technique, called Lempel-ZivWelch (LZW) coding, This Algorithms achieve their compression by replacing a repeated sequence of characters with a reference back to its previous occurrence [5]. It assigns fixed-length code words to variable length sequences of source symbols but requires no a priori knowledge of the probability of occurrence of the symbols to be encoded [6]. LZW compression has been integrated into a variety of mainstream imaging file formats, including the *graphic interchange format* (GIF), *tagged image file format* (TIFF), and the *portable document format* (PDF).

LZW is an adaptive technique. Asthe compression algorithm runs, a changing dictionary of (some *of****)*** the strings that have appeared in the text *so*far is maintained. Because the dictionary is pre-loaded with the *256*different codes that may appear in a byte, it is guaranteed that the entire input source may be converted into a series of dictionary indexes [5]**.**

Encoding involves the following steps: The first input character is used as the first POINTER. The next input character is combined with the pointer and the memory issearched to find such character-pointer combination [7].

At the onset of the Coding process, a codebook or "dictionary" containing the source symbols to be coded is constructed. For 8-bit monochrome images, the first 256 words of the dictionary are assigned to the gray values 0, 1, 2, ..., 255. As the encoder sequentially examines the image's pixels, gray-level sequences that are not in the dictionary are placed in algorithmically determined (e.g., the next unused) locations. If the first two pixels of the image are white, for instance, sequence "255-255" might be assigned to location 256, the address following the locations reserved for gray levels 0 through 255.The next time that two consecutive white pixels are encountered, code word 256, the address of the location containing sequence 255-255, is used to represent them. If a 9-bit, 512-word dictionary is employed in the coding process, the original (8 + 8) bits that were used to represent the two pixels are replaced by a single 9-bit code word. Cleary, the size of the dictionary is an imp9rtant system parameter. If it is too small, the detection of matching gray-level sequences will be less likely; if it is too large, the size of the code words will adversely affect compression performance.

Consider the following 4 X 4, 8-bit image of a vertical edge [14]:

39 39 126 126

39 39 126 126

39 39 126 126

39 39 126 126

A 512-word dictionary with the following starting content is assumed.



Table 2.1 LZW Dictionary

Locations 256 through 511 are initially unused. The image is encoded by processing its pixels in a left-to-right, top-to-bottom manner. Each successive gray-level value is concatenated with a variable column1 of Table -called the "currently recognized sequence." As can be seen, this variable is initially null or empty. The dictionary is searched for each



 Table 2.2 LZW dictionary entry

concatenated sequence and if found, as was the case in the first row of the table, is replaced by the newly concatenated and recognized (i.e., located in the dictionary) sequence. This was done in column 1 of row 2. No output codes are generated, nor are the dictionary altered. 1f the concatenated sequence is not found, however, the address of the currently recognized sequence is output as the next encoded value, the concatenated but unrecognized sequence is added to the dictionary, and the currently recognized sequence is initialized to the current pixel value. This occurred in row 2 of the table. The last two columns detail the gray-level sequences that are added to the dictionary when scanning the entire 4 X 4 image. Nine additional code words are defined. At the conclusion of coding, the dictionary contains 265 code words and the LZW algorithm has successfully identified several repeating gray-level sequences-leveraging them to reduce the originall28-bit image to 90 bits (i.e., 10 9-bit codes).The encoded output is obtained by reading the third column from top to bottom. The resulting compression ratio is 1.42: 1.

A unique feature of the LZW coding just demonstrated is that the coding dictionary or code book is created while the data are being encoded. Remarkably, an LZW decoder builds an identical decompression dictionary as it decodes simultaneously the encoded data stream. Most practical applications require a strategy for handling dictionary overflow. A simple solution is to flush or reinitialize the dictionary when it becomes full and continue coding with a new initialized dictionary. A more complex option is to monitor compression performance and flush the dictionary when it becomes poor or unacceptable. Alternately, the least used dictionary entries can be tracked and replaced when necessary.

Features of LZW codes [7]

* The libraries are assembled from the input data and need not be preserved after storage or communication. The library will automatically adapt itself to the users data type and language. There is no need for exchanging libraries before communication. Data encryption is not provided.
* Compression ratios are not as high because of the long communications code which must include a pointer address and a text character. Using delayed innovation in the BTLZ code provides much higher compression. The library is very disorganized and text words are randomly chopped into fragments. This disorganized library is known as a “Greedy” algorithm.
* Data expansion will occur for short files until the library has accumulated enough nodes for comparison.
* Error propagation is a serious problem. Any storage or communications error will cause the libraries to become different to scramble the entire remaining data file.

**2.5.3 Entropy Coding**

### 2.5.3.1 Shannon-Fano Coding

Shannon-Fano coding, named after Claude Shannon and Robert Fano, was the first algorithm to construct a set of the best variable-size codes. The Shannon-Fano technique has as an advantage its simplicity. The basis for compression of classical data is Shannon’s noiseless coding theorem: if the per symbol code rate is slightly larger than the Shannon entropy, then there exists a block code (with sufficiently large block size) such that the compressed message can be recovered with probability close to unity [8].

We start with a set of *n* symbols with known probabilities (or frequencies) of occurrence. The symbols are first arranged in descending order of their probabilities. The set of symbols is then divided into two subsets that have the same (or almost the same) probabilities. All symbols in one subset get assigned codes that start with a 0, while the codes of the symbols in the other subset start with a 1. Each subset is then recursively divided into two sub subsets of roughly equal probabilities, and the second bit of all the codes is determined in a similar way. When a subset contains just two symbols, their codes are distinguished by adding one more bit to each. The process continues until no more subsets remain [1].

The code is constructed as follows: the source messages a(i) and their probabilities p( a(i) ) are listed in order of non-increasing probability. This list is then divided in such a way as to form two groups of as nearly equal total probabilities as possible. Each message in the first group receives 0 as the first digit of its codeword; the messages in the second half have codewords beginning with 1. Each of these groups is then divided according to the same criterion and additional code digits are appended. The process is continued until each subset contains only one message. Clearly the Shannon-Fano algorithm yields a minimal prefix code.

a 1/2 0

b 1/4 10

c 1/8 110

d 1/16 1110

e 1/32 11110

f 1/32 11111

Figure 2.5 -- A Shannon-Fano Code.

Figure 2.5 shows the application of the method to a particularly simple probability distribution. The length of each codeword x is equal to -log p(x). This is true as long as it is possible to divide the list into subgroups of exactly equal probability. When this is not possible, some codewords may be of length –log p(x)+1. The Shannon-Fano algorithm yields an average codeword length S which satisfies H <= S <= H + 1.

In Figure below, the Shannon-Fano code for ensemble EXAMPLE is given. As is often the case, the average codeword length is the same as that achieved by the Huffman code. That the Shannon-Fano algorithm is not guaranteed to produce an optimal code is demonstrated by the following set of probabilities: { .35, .17, .17, .16, .15 }.

g 8/40 00

f 7/40 010

e 6/40 011

d 5/40 100

space 5/40 101

c 4/40 110

b 3/40 1110

a 2/40 1111

Figure 2.6 -- A Shannon-Fano Code for EXAMPLE

 (code length=117).

The Shannon-Fano method is easy to implement but the code it produces is generally not as good as that produced by the Huffman method.

### 2.5.3.2 Static Huffman Coding

Huffman’s algorithm is a well known encoding method that generates an *optimal prefix*encoding scheme: in the sense that the average code word length is minimum. Asopposed to this, Fano’s method has not been used so much because it generates prefix encoding schemes that are not optimal [9].

Huffman's algorithm, expressed graphically, takes as input a list of nonnegative weights {w(1), ... ,w(n) } and constructs a full binary tree [a binary tree is full if every node has either zero or two children] whose leaves are labelled with the weights. When the Huffman algorithm is used to construct a code, the weights represent the probabilities associated with the source letters. Initially there is a set of singleton trees, one for each weight in the list.

At each step in the algorithm the trees corresponding to the two smallest weights, w(i) and w(j), are merged into a new tree whose weight is w(i)+w(j) and whose root has two children which are the subtrees represented by w(i) and w(j). The weights w(i) and w(j) are removed from the list and w(i)+w(j) is inserted into the list. This process continues until the weight list contains a single value. If, at any time, there is more than one way to choose a smallest pair of weights, any such pair may be chosen. In Huffman's paper, the process begins with a non-increasing list of weights. This detail is not important to the correctness of the algorithm, but it does provide a more efficient implementation. The Huffman algorithm is demonstrated in Figure.



Figure 2.7-- The Huffman process



 Fig2.8 Huffman coding procedure

The Huffman algorithm determines the lengths of the code words to be mapped to each of the source letters a(i). There are many alternatives for specifying the actual digits; it is necessary only that the code have the prefix property. The usual assignment entails labelling the edge from each parent to its left child with the digit 0 and the edge to the right child with 1. The codeword for each source letter is the sequence of labels along the path from the root to the leaf node representing that letter. The code words for the source of Figure, in order of decreasing probability, are {01, 11, 001, 100, 101, 000, 0001}. Clearly, this process yields a minimal prefix code. Further, the algorithm is guaranteed to produce an optimal (minimum redundancy) code. Gallager has proved an upper bound on the redundancy of a Huffman code of p(n) + log [(2 log e)/e] which is approximately p(n) + 0.086, where p(n) is the probability of the least likely source message.

 In addition to the fact that there are many ways of forming codewords of appropriate lengths, there are cases in which the Huffman algorithm does not uniquely determine these lengths due to the arbitrary choice among equal minimum weights. As an example, codes with codeword lengths of {1,2,3,4,4} and of {2,2,2,3,3} both yield the same average codeword length for a source with probabilities {.4,.2,.2,.1,.1}.

2.5.3.3 Adaptive Huffman Coding

In the adaptive Huffman coding procedure, neither transmitter nor receiver knows anything about the statistics of the source sequence at the start of transmission. The tree at both the transmitter and the receiver consists of a single node that corresponds to all symbols not yet transmitted (NYT) and has a weight of zero. As transmission progresses, nodes corresponding to symbols transmitted will be added to the tree, and the tree is reconfigured using an update procedure. Before the beginning of transmission, a fixed code for each symbol is agreed upon between transmitter and receiver [2].

The Huffman coding creates variable-length codes that are an integral number of bits. Symbols with higher probabilities get shorter codes. In the DCS, Huffman codes are built by using bottom up approach, starting with the leaves of the tree and working progressively closer to the root [10]. The Huffman method assumes that the frequencies of occurrence of all the symbols of the alphabet are known to the compressor. In practice, the frequencies are seldom, if ever, known in advance. One approach to this problem is for the compressor to read the original data twice.

The first time, it just calculates the frequencies. The second time, it compresses the data. Between the two passes, the compressor constructs the Huffman tree. Such a method is called semi-adaptive and is normally too slow to be practical. The method that is used in practice is called adaptive (or dynamic) Huffman coding [1]. This method is the basis of the UNIX compact program. The method was originally developed by Faller and Gallager with substantial improvements by Knuth.

The main idea is for the compressor and the decompressor to start with an empty Huffman tree and to modify it as symbols are being read and processed (in the case of the compressor, the word “processed” means compressed; in the case of the decompressor, it means decompressed). The compressor and decompressor should modify the tree in the same way, so at any point in the process they should use the same codes, although those codes may change from step to step. We say that the compressor and decompressor are synchronized or that they work in *lockstep* (although they don’t necessarily work together; compression and decompression normally take place at different times). The term *mirroring* is perhaps a better choice. The decoder mirrors the operations of the encoder.

Initially, the compressor starts with an empty Huffman tree. No symbols have been assigned codes yet. The first symbol being input is simply written on the output stream in its uncompressed form. The symbol is then added to the tree and a code assigned to it. The next time this symbol is encountered, its current code is written on the stream and its frequency incremented by one. Since this modifies the tree, it (the tree) is examined to see whether it is still a Huffman tree (best codes). If not, it is rearranged, which results in changing the codes. The decompressor mirrors the same steps. When it reads the uncompressed form of a symbol, it adds it to the tree and assigns it a code. When it reads a compressed (variable-size) code, it scans the current tree to determine what symbol the code belongs to, and it increments the symbol’s frequency and rearranges the tree in the same way as the compressor.

The only subtle point is that the decompressor needs to know whether the item it has just input is an uncompressed symbol or a variable-size code. To remove any ambiguity, each uncompressed symbol is preceded by a special, variable-size *escape code*. When the decompressor reads this code, it knows that the next 8 bits are the ASCII code of a symbol that appears in the compressed stream for the first time. The trouble is that the escape code should not be any of the variable-size codes used for the symbols. These codes, however, are being modified every time the tree is rearranged, which is why the escape code should also be modified. A natural way to do this is to add an empty leaf to the tree, a leaf with a zero frequency of occurrence, that’s always assigned to the 0-branch of the tree. Since the leaf is in the tree, it gets a variable-size code assigned. This code is the escape code preceding every uncompressed symbol. As the tree is being rearranged, the position of the empty leaf—and thus its code—change, but this escape code is always used to identify uncompressed symbols in the compressed stream. Figure shows how the escape code moves and changes as the tree grows.



Fig 2.9: The Escape code

### 2.5.3.4 Arithmetic Coding

The method of arithmetic coding was suggested by Elias, and presented by Abramson in his text on Information Theory [Abramson 1963]. Implementations of Elias technique were developed by Rissanen, Pasco, Rubin, and, most recently, Witten et al. [Rissanen 1976; Pasco 1976; Rubin 1979; Witten et al. 1987]. Arithmetic coding is an efficient encoding of interval. In this, we divide the interval into subintervals according to the probabilities. Symbols with higher probability get bigger interval proportionally with respect to the other symbol [11].

In arithmetic coding a source ensemble is represented by an interval between 0 and 1 on the real number line. Each symbol of the ensemble narrows this interval. As the interval becomes smaller, the number of bits needed to specify it grows. Arithmetic coding assumes an explicit probabilistic model of the source. It is a defined-word scheme which uses the probabilities of the source messages to successively narrow the interval used to represent the ensemble. A high probability message narrows the interval less than a low probability message, so that high probability messages contribute fewer bits to the coded ensemble. The method begins with an unordered list of source messages and their probabilities. The number line is partitioned into subintervals based on cumulative probabilities. Two equations may be used to define the narrowing process described above:

newleft = prevleft + msgleft\*prevsize (1)

newsize = prevsize \* msgsize (2)

The first equation states that the left endpoint of the new interval is calculated from the previous interval and the current source message. The left endpoint of the range associated with the current message specifies what percent of the previous interval to remove from the left in order to form the new interval.

The second equation computes the size of the new interval from the previous interval size and the probability of the current message (which is equivalent to the size of its associated range). The size of the final subinterval determines the number of bits needed to specify a number in that range. The number of bits needed to specify a subinterval of [0,1) of size s is -log s. In order to recover the original ensemble, the decoder must know the model of the source used by the encoder (e.g., the source messages and associated ranges) and a single number within the interval determined by the encoder. Decoding consists of a series of comparisons of the number i to the ranges representing the source messages

Several difficulties become evident when implementation of arithmetic coding is attempted. The first is that the decoder needs some way of knowing when to stop. As evidence of this, the number 0 could represent any of the source ensembles A, AA, AAA, etc. Two solutions to this problem have been suggested.



Fig2.10: Arithmetic Coding

One is that the encoder transmits the size of the ensemble as part of the description of the model. Another is that a special symbol be included in the model for the purpose of signalling end of message. The second alternative is preferable for several reasons.

* First, sending the size of the ensemble requires a two-pass process and precludes the use of arithmetic coding as part of a hybrid codebook scheme.
* Secondly, adaptive methods of arithmetic coding are easily developed and a first pass to determine ensemble size is inappropriate in an on-line adaptive scheme.

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

A third issue is that of precision. From the description of arithmetic coding it appears that the precision required grows without bound as the length of the ensemble grows. Fixed precision registers may be used as long as underflow and overflow are detected and managed.

2.5.3.5 Range encoding

Range encoding is a [data compression](file:///%5C%5Cwiki%5CData_compression) method defined by G. Nigel N. Martin. Range encoding (or range coding) is an improvement to arithmetic coding that reduces the number of renormalizations and thereby speeds up integer-based arithmetic coding by factors of up to 2 [1]. Range encoding is a form of [arithmetic coding](file:///%5C%5Cwiki%5CArithmetic_coding). Range encoding conceptually encodes all the symbols of the message into one number, unlike [Huffman coding](file:///%5C%5Cwiki%5CHuffman_coding) which assigns each symbol a bit-pattern and concatenates all the bit-patterns together. Thus range encoding can achieve greater compression ratios than the one-bit-per-symbol upper bound on [Huffman encoding](file:///%5C%5Cwiki%5CHuffman_encoding) and it does not suffer the inefficiencies that Huffman does when dealing with probabilities that are not exact [powers of two](file:///%5C%5Cwiki%5CPower_of_two).

The central concept behind range encoding is this: given a large-enough range of [integers](file:///%5C%5Cwiki%5CInteger), and probability estimation for the symbols, the initial range can easily be divided into sub-ranges whose sizes are proportional to the probability of the symbol they represent. Each symbol of the message can then be encoded in turn, by reducing the current range down to just that sub-range which corresponds to the next symbol to be encoded. The decoder must have the same probability estimation the encoder used, which can either be sent in advance, derived from already transferred data or be part of the compressor and decompressor.

When all symbols have been encoded, merely identifying the sub-range is enough to communicate the entire message (presuming of course that the decoder is somehow notified when it has extracted the entire message). A single integer is actually sufficient to identify the sub-range, and it may not even be necessary to transmit the entire integer; if there is a sequence of digits such that every integer beginning with that prefix falls within the sub-range, then the prefix alone is all that's needed to identify the sub-range and thus transmit the message.

**2.6 Comparison between different compression algorithms**

Here is a Table 2.3 [4] shows comparison between different data compression algorithm.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithms** | **Data****encoding** | **Compression****scheme** | **Compression ratio** | **Time complexity** | **Remarks** |
| Static Huffman | Entropy | Static | Achieve good Compression ratio in most cases. | N [log (n) + a] + Sn where N is the total number of input symbols, n is the current number of unique symbols, a is the arithmetic to be performed, and S is the time required, if necessary, to maintain internal data structures | * Symbol frequencies must be known in advance.
* Symbol frequency table must be stored along with the compressed data.
* Require two passes: one pass to compute the Symbol
* Frequency table and the other for compression.
* For large files containing a large number of characters, building
 |
| Arithmeticcoding | Entropy | Static | Achieve better compression ratio than Huffman. | N [log (n) + a] + Sn where N is the total number of input symbols, n is the current number of unique symbols, a is the arithmetic to be performed, and S is the time required, if necessary, to maintain internal data structures | * Similar to Huffman, symbol frequencies must be known in advance.
* Similar to Huffman, symbol frequency table must bestored/transmitted along with the compressed data which produces some overhead.
* ·Complex to implement.
 |
| AdaptiveHuffman | Dictionary | Adaptive | Achieve relativeSame compression ratio with Huffman | N [n+log (2n − 1) + Sn where N is the total number of input symbols, n is the current number of unique symbols, and S is the time required, if necessary, to rebalance the tree | * Require only one pass of the data.
* Lose out in terms of speed.
* No overhead of Symbol frequency table
* Complex in implementing as the tree has to evolve itself
 |
| LZW | Dictionary | Adaptive | Relative good compression ratio for Text data. Other type of data varies. | O(n) where n is the number of symbols | * Table can get very large easily.
* Require only one pass of the data.
* Slow in speed as each time a new character is read in, the algorithm has to search for the new string formed by STRING+CHARACTER.
* No overhead of Symbol frequency table
 |
| LZ77 | Dictionary | Adaptive | Compression ratio is not as good as others | O(n) where n is the number of symbols | * Decompression is one of the fastest among all the other algorithms.
* Easy to implement
 |

Table 2.3 Comparison between different compression algorithms

This table 2.4 [12] shows different lossless data compression algorithm

****

Table 2.4: Different Data Compression Algorithm

The table 2.5 [12] shown below shows the advantage and disadvantages of the lossless data compression algorithms.



Table 2.5 Evaluations of four lossless compressions methods

Chapter 3

Data Compression Algorithm: Proposed

3.1 Previous Techniques

3.2 Proposed Technique

3.2.1 Example related to proposed algorithm

3.2.2 About the algorithm

3.3 Proposed Data Compression algorithm

3.3.1 Algorithm Compression

3.3.2 Algorithm Decompression

Data Compression Algorithm: Proposed

**3.1 Previous Techniques**

The algorithm previously developed based on the assumption that, any type of data is going to be stored in the form of bits on the memory device. The basic manipulation we are doing on the basis of power of 2. This is the base of the compression technique. The mapping of uncompressed to compressed and compressed to decompress is on the power of 2.

Since our computer is also based on this architecture, this is the only way to make a new algorithm or to improve the previously designed algorithm.

The technique developed using this system requires at least information about the probability of the words or letters, or some type of dictionary or some other information that can be used to compress or to decompress data. This is the necessary information to be attached with the compressed data used for lossless decompression. This is the reason the algorithm not work to its full extent.

**3.2 Proposed Technique**

**3.2.1 Example related to proposed algorithm**

On the above discussion I am going to propose a new data compression algorithm to break the limits of the compressed data. In this algorithm, I am going to introduce a new data representation technique that can be used for the data compression. The main focus of this algorithm is to use recursion in the data compression. All previously defined algorithms used mapping, occurrence, dictionary for compression, that’s why they are not suitable for recursion, or if we are going to compress a compressed data that makes no sense because it may or may not compress. If the data compressed like that but the compression ratio will be very poor. So using recursion on these algorithms is not efficient.

So we require such type of algorithm that is suitable for the recursion. To show how the recursion takes place in the compression technique, let’s take a real life example.

A person Bob wants to share the file (in MB) on the internet with Alice. But Bob is having very low bandwidth. So he decided to compress it. The information on the file is something like this:-

 314285714285714285123425.................

If we can see closely first 18 number belongs to the fraction 22/7 i.e.; value of$ π$ .

 $π$=22/7= 3.1428571428571428571428571428571

 $\uparrow $

So if we can compress like this 22+7+18+123425........

So the partially compressed data will be 22718123425........

This data is not fully compressed. Now again we compress the data but not started from the 123425..... But we are going to compress from the starting of the data stream that is from 22718123425........

Now we again calculate the numerator and denominator for the new fraction part. This procedure is recursively applied on the partially compressed data till the complete data not compressed. This is one of the algorithms that can be used for the compression of the data. This is the way how my algorithm is going to work.

**3.2.2 About the algorithm:**

The binary system we are using yet based on the power of 2. For representing a number we can define it by using this system.

For example: To represent 27

 27= 1\*$2^{4}$ + 1\*$2^{3} $+ 0\*$2^{2}$ + 1\*$2^{1}$ + 1\*$2^{0}$.

This actually stores in the memory device in the bit pattern:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 1 | 1 |

This number system can define a number uniquely. But this is not only the series that can be used to define a number uniquely. We are having one more like that. The series that can be used is the power of 3. This is the series that can be used for the compression purpose.

For example: To represent 29

 29= 1\*$3^{3}$ + 0\*$3^{2} $+ 1\*$3^{1}$ - 1\*$3^{0}$.

 = 27 + 0 + 3 - 1

 = 29.

By using in this manner can represent any number. This is using only total of 4 summation subtraction. But on comparing these series we find that there is a difference in between.

* In the series of 2, a number is having only with addition sign on each power of 2. It indicates that whether the power of 2 is applicable for representing a number or not.
* In the series of 3, a number consists of some power of 3 with negative sign, some with positive sign and some of power of 3 is not having any use.

So for representing a number in power of 3 we should know the

1. Positive power
2. Negative power
3. Which has no role

We can also think of this as whether the power is having a positive sign or 0 or whether it is having a negative sign or 0 values.

For representing it and to store into the memory we require both positive and negative representation. So for this we require a separate positive value stream bits and negative value stream bits.

For example to represent 29:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 1 | 0 |

Positive power value of 3

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 1 |

Negative power value of 3

But on seeing this, it requires 8 bits in comparison to the power of 2. Then how the compression will work? Instead of making a compression the data is going to expand. By closely examining, for representing a number it is found that

1. If a number is having positive sign of particular power of 3 then it will not have a negative sign for this power of 3.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 1 | 0 |

+ Power

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 1 |

* power

 $\uparrow $

1. If a number is having negative sign of particular power of 3 then it will not have a positive power of 3 for it.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 1 | 0 |

 + Power

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 1 |

+ Power

 $\uparrow $

1. A number can have both 0 in the power of 3.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 1 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 1 |

 $\uparrow $

 4. So maximum of four 1’s in the above example. This example contains only three.

So for making compression to work on this power series, I make out four conditions to take this series useful. Here is the base of the compression algorithm that actually does the compression. Let, we have a message of length m, now it is converted to power of 3. It requires n bits to store positive coefficient and n bits to store negative bits. The total of 2n bits are required till now. The four conditions are:-

1. If number of coefficient with positive power of 3 is greater than or equal to half of the number of bits used that is ‘n/2’.
2. If number of coefficient with negative power of 3 is greater than or equal to half of the number of bits used that is ‘n/2’.
3. If total number of coefficient with positive and negative power is less than or equal to half of the number of bits that is ‘n/2’.
4. If total number of coefficient with positive and negative power is greater than half of the number bits that is ‘n/2’.

These are the possible combination that a number representation may have. In the first two cases if we have number of coefficient (1 with positive or negative sign) greater than n/2 then we require less than or equal to ‘n/2’ more bits to represent the other bits that whether it is having positive appearance or negative in the number. So a total of (n + n/2) required instead of 2n bits.

For example: 29

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 1 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 1 |

Number of positive bits =4

Number of negative bits=4.

No of bits having positive coefficient =2

 that is equal to n/2.

No of bits having negative coefficient =1.

Since in the positive bits only two of them are 0, So we require only two bits to decide whether the corresponding negative bits have any appearance or not. So first four bits of positive coefficient are used unchanged.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | ? | ? |

We can see from here that first bit in the negative bits is having 1, so we can put it into the bit stream.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | ? | 1 |

 $\uparrow $

Third bit in negative bits is 0. So we can put 0 in the next bit position.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | 0 | 1 |

 $\uparrow $

So now we require 6 bits instead of 8 bits. Its reverse procedure is also possible.

For the third case if total number of positive and negative coefficients are less than or equal to n/2. Then we require only n/2 more bits to decide whether the bit is having a positive or negative coefficient. Thus we require only 3n/2 bits.

In the fourth condition, I still not found any solution. So I am still thinking a solution, that a forward and reverse procedure is possible in this case for compression and decompression. In my code if this condition met then I have made a termination condition so that it will stop compression and store it into the compressed file and starts compression on the remaining data.

This is my future work to find the solution of this problem because if this problem solved then this algorithm will achieve highest compression ration ever in the past or future by any algorithm.

The examples I have showed here still not convey the compression but if we increase the number of bits according to the compressed bits we will achieve a limit where we are actually going to compress the data. By doing this we will again reach at a position where if we have m partial bits of data, then we will get a partially compressed data which takes less than m bits. And if we can find solution for the fourth case then we can compress data of any length with a constant number of bits.

This is the internal part of the newly designed algorithm. This is used while doing compression.

**3.3 Proposed Data Compression algorithm**

**3.3.1 Algorithm Compression**

Let we have Message data of m bits. Partial data array PDA[n] is of size n bits. Partially compressed data array PCA[k] is of size k bits (here k<n<m) and two array to store the positive pos[c] and negative neg[c] coefficient bits c bits each (here k=3/2c). And a counter that counts the number of bits compressed.

1. Take next n bits of data from the Message data into PDA[n].
2. Convert these as a power series of 2 into power series of 3 that is having c positive pos[c] and c negative neg[c] bits.
3. Now compress pos[c] and neg[c] partial data using above four conditions into 3/2c bits partially compressed data PCA[k].
4. Now copy these m bits into the first m bits of PDA[n].
5. Now take out next (n-k) bits from the Message data and put into the array PDA[n] from (n-k-1) to 0th bits.
6. Repeat step II to IV till all or fourth condition not met
7. If fourth condition met store PDA[n] to the compressed file with number of bits it has compressed and then go to step I.
8. Else if all data compressed then save PDA[n] onto the compressed file and terminate the program.

Figure 3.1 shows the flow chart on the basis of the data compression algorithm presented here.

 START

Set counter to zero

Read next k bits from the file

Read next n-k bits from the file

Convert the power of 2 representations into power of 3

Check if one of the three conditions met

 Yes No

Store the previous compressed form with bytes that represents the counter

Calculate its compressed form and increment the counter

Is n bits are available

Is n-k bits are available

 Yes Yes

 No No

Store previous compressed form with remaining bits and counter bytes

Store these remaining bits without change

END Fig 3.1 Flow chart Data compression

**3.3.2 Algorithm Decompression:**

1. Fetch next n number of bits from the compressed data into PDA[n].
2. Fetch next byte/bytes that store the count.
3. Take most significant k bits from PDA[n] and expand them using reverse procedure that is by convert back into positive and negatives of power series of 3 as describe above
4. And now convert it to the power series of 2, and store this as partially decompressed data of n bits into PDA[n] itself. Decrement the counter by 1.
5. Append at start the least significant (n-k) bits in the partially decompressed file.
6. Repeat steps III to IV till the count not zero.
7. Add this file to compressed file.
8. Repeat step I to VI till complete file will not decompressed.

Figure 3.2 shown on the next page is the flow chart of the data decompression on the basis of the algorithm presented here.

 START

Append available bits at the star of compressed file

Check if counter =0

Append the least n-k onto the decompressed file at the start and decrement the counter

Convert k bits from power of 3 representations into 2

Load n bits and counter bytes from the file

 No

 Yes

Check if n bits are available

 Yes

 No

 END

 Fig 3.2 Flow chart Data decompression

Chapter 4

Results and Performance

4.1 Measure of Performance

4.2 Results

Results and Performance

**4.1 Measure of Performance**

The performance of a compression algorithm can be measured in various ways. A common method used to evaluate a compression algorithm is to measure the amount of reduction in size in the compressed version of the files, which is often referred to as compression ratio. The compression ratio is simply the ratio of the compressed file size to the original size. Thus, a bigger compression ratio indicates higher reduction in the compressed file size. This can be calculated as follows:

$$Compression ratio =100\*\left(1-\frac{compressed file}{uncompressed file size}\right)^{}$$

Swapping the numerator with denominator of the previous equation is called compression factor [3]. The compression factor is used to numerically quantify the amount of data reduction in numbers.

$$Compression factor =\frac{uncompressed file size}{compressed file}$$

An alternative method for measuring the algorithm’s performance is to report the average number of bits required to encode a single symbol. This is often called the compression rate [1].

Other criteria are used as well. For example, the computational complexity of the algorithm used to perform compression and/or decompression is used to assess compression systems [3]. Also, the amount of memory required to implement compression is considered important [1]. The overhead is used to measure the amount of extra information stored to the compressed file to be used in decompression [3]. Furthermore, the entropy is used to quantify the theoretical bound of the source [3]. The difference between the average code length and the entropy of the source is referred to as redundancy [3].

**4.2 Results**

This chapter shows the outcome/results obtained from the partially developed compression algorithm. The algorithm developed on first three conditions as described in the algorithm. The fourth condition reverse procedure is still to be solved. That makes it not to be applied on the actual data. So the requirement is to make it to apply on a special type of data where we can see the compression occur. So I have made some test data files. And then I have applied the compression algorithm on these data. I have taken these test data in a special form such that it will not reach the fourth condition. This makes these test data files to be very small to make it to compress the test data files. The minimum size of any compressed file is 114 bits. Some of the test results are shown below. Shows how much compression can be done using this algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S. No. | Compressed file | File size | Compression ratio | Compression factor |
| 1 | File 1 | 120 | 5 | 1.05 |
| 2 | File 2 | 135 | 15 | 1.18 |
| 3 | File 3 | 148 | 23 | 1.29 |
| 4 | File 4 | 158 | 27 | 1.38 |
| 5 | File 5 | 168 | 32 | 1.47 |

Table 4.1 Results

The above table 1 shows the results taken by compressing the files 1,2,3,4 and 5. This table 1 shows the corresponding file size with their compression ratio and compression factor. Here I am giving only the results of this new algorithm. This is not a comparison to any algorithm.

X axis- file size Y axis- compression ratio

Figure 4.1: Graph between uncompressed file size Vs compression ratio

The above figure shows a graph between uncompressed file size and compression ratio. On X-axis I have taken the size of the uncompressed file and on Y-axis I have taken compression ratio. This graph basically shows how much compression we can achieve as our file size grows. The curve shows on increasing uncompressed file size the size of compressed will become very low.

Fig 4.2: Graph Uncompressed file Vs Compression factor

The figure shown above, shows graph between uncompressed file on X-axis and compression factor on Y-axis.

Chapter 5

LIMITATION

Limitation

The algorithm presented in this thesis has very bright future. On completion it can give very high compression ratio. But presently the desired performance has not been reached due to some limitation presents in the architecture.

This data compression algorithm uses recursion which is the main pillar in the technique. Converting one representation form to another in this algorithm requires large calculation. And each calculation require so much time. The compression technique uses the whole procedure to compress very small number of bits. This procedure applied to complete data using recursion. That takes a lot of time to compress the small data. This makes higher time complexity. Therefore in present condition it is not feasible to compress very large size data file.

The representation system require in the algorithm uses large number of predefined arrays that are stored permanently in the during whole compression process. That makes it uses large space in the main memory. That means some higher space complexity during compression on run time.

In the present conditions this cannot be suitable for the large file so we have created some test file to compress. So the implemented version of this compression technique is currently used on the testing data. And this is used only for the testing purpose.

Chapter 6

CONCLUSION

Conclusion

Data compression is one of many technologies that have enabled the information revolution. Evidently there is an explosive growth in the information that needs to be transmitted or stored. There is a stable improvement of storage and transmission technologies in response to this (e.g., Blue Ray CDs and optical fibre networks). However, the development of these technologies would need to be twice fast as to match the information growth.

A novel lossless data compression algorithm using a new representation coding was proposed. The main objective of this thesis was to develop a lossless compression algorithms based on the given encoding situation. Previously designed algorithms are designed on the basis of power of two series representation. This is a completely new representation technique. But this requires more research in this field. This algorithm is only in the starting phase so we have made special test data to show its performance. The implemented software of this data compression technique is working well for small data input file. Recursion makes this algorithm to give a highly compressed data. We have not yet able to develop the last fourth condition but still we can compress a self made specialized data. The compression ratio of this algorithm is low but we are trying to achieve higher. The above developed algorithm software application takes input as a bit stream. This is not based on the block compression method. This algorithm does not recognize other representation or meaning therefore this cannot be applied as a more compressed image or video. But it can take these as a simple file and compress it.

Chapter 7

FUTURE SCOPE

FUTURE SCOPE

As this algorithm is in the starting phase, so we have developed it in a very casual manner. We are using strings to convert to different representation this makes to use large number of instruction to perform a single task. This requires higher time requirement. Also the designed fourth condition is not fully developed, that makes to it not to apply on any general purpose data. So this requires improvement in the future.

Our first and the most important future aspect are to develop the fourth condition solution. That has its forward and reverse solution that means we can decompress our data if we can compress it by fourth condition. Already we worked on the alternatives but they are not giving the complete solutions. If we do not get its solution we can have the option by splitting this problem into sub problems and find their reversible solution for our application.

Our second most important future aspect is to minimize the instruction per single task to improve its time requirement by using some efficient mapping technique that can be used directly to convert one representation to the other representation.

This algorithm uses recursion which is the root of this algorithm that makes our algorithm to compress so much. But this self digestion property of recursion in this algorithm makes it to more vulnerable to change. By any means if single bit has been changed that makes a very big change in decompression. So our aim is to eliminate the recursion if possible or to improve. Also recursion makes high time complexity and space complexity (by stacking the blocks in the memory).

A data compression algorithm itself is an encryption technique which enables some security to hide data from unauthorized user. But if someone knows the data compression algorithm, he can crack it and get this secured data. So our aim is to use some encryption technique to encrypt this data by a user defined key so that only authorized user can check this data. These are some problems that require, doing some improvement in the algorithm.

Chapter 8

REFRENCES

References

1. David Salomon, Computer Science Department California State University Northridge, “*Data Compression The Complete Reference*” Fourth Edition, , CA 91330-8281 USA, Springer-Verlag London Limited 2007, ISBN-13: 978-1-84628-602-5.
2. Khalid Sayood, University of Nebraska, “*Introduction to Data Compression*” Third edition, by Elsevier Inc, 2006, ISBN 13: 978-0-12-620862-7.
3. Jiaiimin JiaiigDepartment of Computer Studies, Loughborough University, United Kingdom, “*Pipeline Algorithms of RSA Data Encryption and Data Compression*”, IEEE.
4. *Su-Lim TAN, Chee-Hui LEE, Quoc-Huy PHAN*, Singapore, “*A COMPARITIVE STUDY OF COMPRESSION TECHNIQUE FOR GPS DATA FOR TECTRONIC AND VOLCANIC OBSERVATION*”, IEEE 9th Malaysia International Conference on Communications15 -17 December 2009 Kuala Lumpur Malaysia, 2009.
5. R. Nigel Horspool Dept. of Computer Science, University of Victoria,” Improving LZW”, IEEE,1991.
6. H.C. KOTZI and G.J. KUHN, Department of Electronic and Computer Engineering University of Pretoria, “*AN EVALUATION OF THE LEMPEL - ZIV - WELCH DATA COMPRESSION ALGORITHM*”, IEEE 1989,
7. Klaus Holtz, San Francisco, “*The Evolution of Lossless Data Compression Techniques*” IEEE.
8. Isaac L. Chuang Dharmendra S. Modha IBM Almaden Research Center, “*Quantum Arithmetic Coding*”, Sorrento, Italy, IEEE. June *25-30, 2000.*
9. LUIS *G.* RUEDA and B. JOHN OOMMEN ,School of Computer Science, Carleton University 1125 Colonel By Dr., Ottawa, ON, K1S 336, Canada, “*ENHANCED STATIC FAN0 CODING*”,IEEE,2001.
10. Dr. Muhammad Younus Javed and Mr. Abid Nadeem, Rawalpindi, “*DATA COMPRESSION THROUGH ADAPTIVE HUFFMAN CODING SCHEME*”,IEEE,2000.
11. Ulrich Graf Darmstadt Germany, “*Dense Coding - A Fast Alternative to Arithmetic Coding*”, IEEE,1998.
12. YAN Jing,LUO Rong, GAO Rui, XU Ning-Yi, “*An Efficient Lossless Compression Method for Internet Search Data in Hardware Accelerators*” World Congress on Computer Science and Information Engineering, IEEE, 2009.
13. Guy E. Blelloch Computer Science Department Carnegie Mellon University, “*Introduction to Data Compression*” Septamber 25, 2010.
14. Rafael C. Gonzalez University of Tennessee, Richard E. Woods MedData Interactive, ”*Digital Image Processing Second Edition*” Prentice Hall Upper Saddle River, New Jersey 07458, ISBN: 0-201-18075-8.
15. Usha S.Mehta, Kankar S. Dasgupta, and NiranjanM. Devashrayee, “*Run-Length-Based Test Data Compression Techniques: How Far from Entropy and Power Bounds*”, 11 January 2010
16. <http://www.wikipedia.com/>
17. <http://www.ics.uci.edu/>