TEXT COMPRESSION USING HUFFMAN CODING

A Project

Submitted for the award of the degree
of

DIPLOMA
in
COMPUTER SCIENCE AND TECHNOLOGY

by

SOUVIK DEY

Under the Guidance of

Madhumita Das
Lecturers, CST Dept. GIST

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY
GLOBAL INSTITUTE OF SCIENCE AND TECHNOLOGY

HALDIA-721657

APRIL 2019

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER 3
LOSSY COMPRESSION

CHAPTER 4
HUFFMAN CODING FOR TEXT COMPRESSION AND DECOMPRESSION

CHAPTER 5
CONCLUSION & FUTURE WORK

GLOBAL INSTITUTE OF SCIENCE & TECHNOLOGY

(An Institution of ICARE)

CERTIFICATE OF APPROVAL

This project is a creditable study in the area of Computer Technology presented in a proper manner to warrant its acceptance as a requisite to the degree for which it has been submitted. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn there in but approve the project work only for the purpose for which it is being submitted.

COMMITTEE ON FINAL EXAMINATION FOR EVALUATION OF THE PROJECT

_____

_____

_____

GLOBAL INSTITUTE OF SCIENCE & TECHNOLOGY

(An Institution of ICARE)

CERTIFICATE

It is certified that the work contained in the project entitled "TEXT COMPRESSION USING HUFFMAN CODE by Souvik Dey(reg: D144055005) for the award of Diploma in computer science and technology has been carried out under my supervision.

Mr. Bikramjit Choudhury (Principal) GLOBAL INSTITUTE OF SCIENCE & TECHNOLOGY	Madhumita Das **(Project Advisor)**	Sk.Maidul Islam **(In-Charge of CST Dept.)**

GLOBAL INSTITUTE OF SCIENCE & TECHNOLOGY

(An Institution of ICARE)

DECLARATION

This is to bring your attention that this project work entitled "**TEXT COMPRESSION USING HUFFMAN CODE" which** is being submitted by us for requirement of the completion of Diploma degree in Computer Science & Technology. from Global Institute of Science & Technology, Haldia.

Souvik Dey(reg:D144055005)

ABSTRACT

In computer science and theory, Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular tray based on the estimated probability of occurrence for each possible value of the source symbol.

It was developed by David A. Huffman while He was a Ph.D. student at MIT and published in the 1952 paper" A Method for the Construction of Minimum-Redundancy Codes".

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code (sometimes called "prefix-free codes that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol) that expresses the most common source symbols using shorter strings of bits than are used for less common source symbols. Huffman was able to design the most efficient compression method of this type no other mapping of individual source symbols to unique strings of bits will produce a smaller average output size when the actual symbol frequencies agree with those used to create the code. The running time of Huffman's method is fairly efficient; it takes 0(n\log n) operations to construct it. A method was later found to design a Huffman code in linear time if input probabilities (also known as weights) are sorted.

Huffman coding today is often used as a "Backend" to some other compression methods. DEFLATE (PKZIP's algorithm) and multimedia codes such as JPEG and MP3 lute front-end model and quantization followed by Huffman coding (or variable-length prefix free codes with a similar structure, although perhaps not necessarily designed by using Huffman's algorithm [clarification needed]).

ACKNOWLEDGEMENTS

Through this acknowledgment, we express our sincere gratitude to all those people who have been associated with this assignment and have helped us with it and made it a worthwhile experience.

Firstly, we extend our thanks to our respected principal **Mr BikramjitChoudhury** and various other people who have shared their opinions and experiences through which we received the required information crucial for our project.

With immense pleasure we express our sincere gratitude, regards and thanks to our guide Madhumita Das, Lecturer GIST for his excellent guidance invaluable suggestions and continuous encouragement at all the stages of our project work. Their interest and confidence in me were the reason for all the success we have made. We have been fortunate to have him as our guide as he has been a great influence on us, both as a person and as a professional.

It was a pleasure to be associated with G.IST computer lab, and I would like to thank the entire Departmental faculties Lab member.

Above all, we are blessed with such caring parents. We extend our deepest gratitude to our parents and our younger brother and sister for their Invaluable love, affection, encouragement and support.

CHAPTER 1

Introduction:

Compressing data is an option naturally selected when faced with problems of high costs or restricted space. Written by a renowned expert in the field, this book of offers readers a succinct, reader-friendly foundation to the chief approaches, methods and techniques currently employed in the field of data compression.

1.1 Definition:

Compression or "data compression," is used to reduce the size of one or more files. When a file is compressed, it takes up less disk space than an uncompressed version and can be transferred to other system more quickly. Therefore, compression is often used to save disk space and reduce the time needed to transfer file over the Internet.

1.2 Need of compression:

1. At any given time, the ability of the Internet to transfer data is fixed.
2. Think of the capability as the Internet's collective bandwidth.
3. Thus, if data can effectively compressed wherever possible, significant improvement of data throughput can be achieved.
4. In some instants, file size can be reduced by up to 60-70 percent.
5. Many files can be combined in to one compressed document making sending easier, provided combine file size is not huge.

1.3 Types of data compression:

There are two primary types of data compression:

1. File Compression
2. Media Compression

File Compression: -

File compression can be used to compress all type of data in to a compressed archived. These archives must first be decompressed with a decompression utility in order to open the original file (s).

Media Compression: -

Media Compression is used to save compressed image, audio, and video files. Example of compressed media formats include JPEG images, MP3 audio, and MPEG video files.

Most image viewer and media playback programs can open standard compressed file type directly.

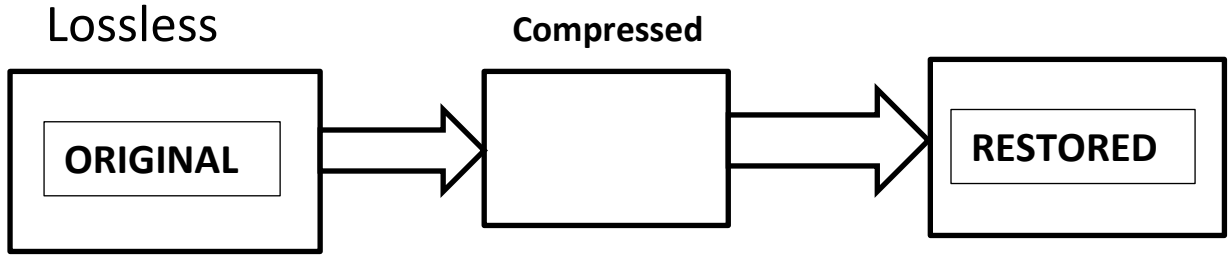
File compression is always performed using a lossless compression algorithm, meaning no information is lost during the compression process. Therefore, a compressed archive can be fully restored to the original version when it decompressed. While some media is compressed using lossless compression, most image, audio, and video files are compressed using lossy compression. This means some of the media's original quality is lost when the file is compressed. However most modern compression algorithm cam compress media with little to no loss in quality.

1.4 Compression Techniques

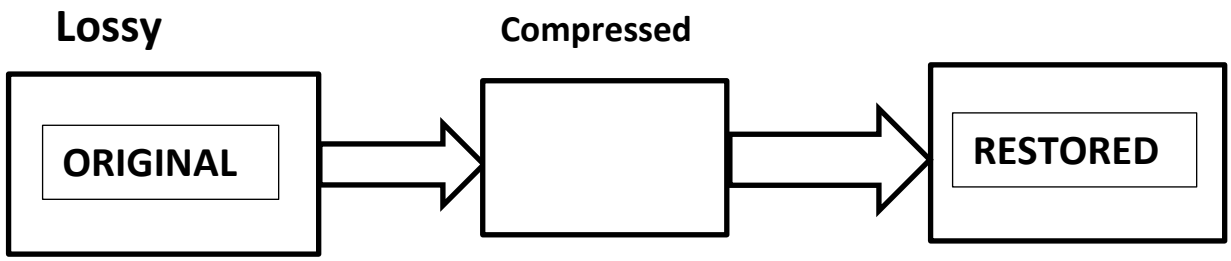
Data compression algorithm can be categorized

1. Lossless
2. Lossy

A lossless technique means that the restored data file is identical to the original. This is absolutely necessary for many types of data, for example: executable code, word processing files, tabulated numbers, etc. you cannot afford to misplaced even a single bit of this type of information.



In comparison, data files that represent images and other acquired signal do not have to not have to be keeping in perfect condition for storage or transmission. All real-world measurements inherently contain a certain amount of additional noise, no harm is done. Compression techniques that allow this type of degradation are called lossy.



CHAPTER 2

LOSSLESS COMPRESSION

2.1 Information Theory

Lossless compression involves finding a representation which will exactly represent the source. There should be no loss of information, and the decompressed, or reconstructed, sequence should be identical to the original sequence. The requirement that there be no loss of information puts a limit on how much compression we can get how much compression we can get. We can get some idea about this limit by looking at some concepts from information theory.

21.1 Entropy

Shannon borrowed the definition of entropy from statistical physics, were entropy represents the randomness or disorder of a system in particular a system is assumed to have a set of possible states it can be in and at a given time there is a probability distribution over those states. Entropy is then defined as

$$H(S) = \sum_{s \in S} p(s) \log_2 1/p(s)$$

where S is the set of possible states, and p(s) is the probability of state € S. This definition indicates that the more even the probabilities the higher the entropy (disorder) and the more biased the probabilities the lower the entropy-e.g., if we know exactly what state the system is in then H(S) - 0. One might remember that the second law of thermodynamics basically says that the entropy of a closed system can only increase.

In the context of information theory Shannon simply replaced "state with message", so S is a set of possible messages, and p () is the probability of message s S. Shannon also defined the notion of the self-information of a message as

$$i(s) = \log_2 1/p(s)$$

This self-information represents the number of bits of information contained in it and, roughly speaking, the number of bits we should use to encode the message. The definition of self-information indicates that messages with higher probability will contain less information (e. g a message saying that it will be sunny out in tomorrow is less informative than one saying that it is going to snow) The entropy is then simply a probability weighted average of the self-information of each message. It is therefore the average number of bits of information contained in a message picked at random from the probability distribution larger entropies represent larger average information and perhaps counter intuitively, the more random a set of messages (the more even the probabilities) the more information they contain on average. Here are some examples of entropies for different probability distributions over five messages

$p(S) = \{0.25, 0.25, 0.25, 0.125, 0.125\}$

$H = 3 \times 0.25 \times \log_2 4 + 2 \times 0.125 \times \log_2 8$

$= 1.5 + 0.75$

$= 2.25$

$p(s) = \{0.5, 0.125, 0.125, 0.125, 0.125\}$

$H = 0.5 \times \log_2 2 + 4 \times 0.125 \times \log_2 8$

$= 0.5 + 1.5$

$= 2$

$p(s) = \{0.75, 0.0625, 0.0625, 0.0625, 0.0625\}$

$H = 0.75 \times \log_2 (4/3) + 4 \times 0.0625 \times \log_2 16$

$= 0.3 + 1$

$= 1.3$

2.2 Lossless Data Compression

It is convenient to minimize the space needed to store data larger files will take longer to transfer over a data link, and will more quickly fill up disk quotas Data compression techniques such as those used in common compression utilities allow reducing file sizes by exploiting redundancies in the data contained in them. Lossless coding techniques do this without compromising any information stored in the file. Lossy techniques may achieve even greater compression, but only by providing an approximate reconstruction of the original data. We discuss lossless binary coding Shannon's lower bound on the code length in terms of entropy, and the Huffman coding algorithm, a greedy approach to generating optimal prefix codes for lossless data compression.

2.2.1 Binary Codes

We encode data as strings of bits (binary digits). For example, suppose than a particular file contains text written only the three characters A, B, C We can encode each character using 2 bits as follows:

A = 00, B = 01, C = 10

Doing this already provides a savings over, say, using an ASCII or Unicode representation, which would spend 8 bits or more per character More generally, we can encode an alphabet containing n different symbols by using dlog2 ne bits per symbol.

2.2.2 Variable length codes

However, we can sometimes do even better by using different code lengths for different symbols of the alphabet. Suppose that as are significantly more common than B's and C's. We can quantify this in terms of the fraction of characters in the file that match each of the three possible candidates. For example, we might know that these probabilities are as follows:

P(A) =0.5, P(B) = 0.25, P(C) =0.25

Since A is more common, we could save space by encoding A using a single bit:

A=0, B=01, C=10

The expected number of bits per character for this new code will be the weighted sum 1* P(A)+2* (P(B) + P(C) = 0.5 + 1 = 1.5 Indeed, we save 0.5 bits per character (25%) in this way.

2.23 Prefix codes

You may have noticed that there's a difficulty with the particular encoding used in the preceding example, since some bit sequences will be ambiguous, that is, they will match more than one possible character sequence. For example, does the sequence 010 correspond to AC or BA? One way of preventing this problem is to require that the binary code of each character cannot be a prefix of any other. Any encoding that satisfies this property is known as a prefix code. We will discuss Huffman's algorithm for finding optimal prefix codes below. First, however, we will present a fundamental lower bound on the code length of any possible coding scheme.

2.2.4 Entropy; Shannon's coding theorem

The absolute lower limit on the number of bits needed to encode a probabilistic data source was discovered by Claude Shannon, a mathematician and electrical engineer working at Bell Labs in the mid-20th century. Shannon found that the concept of entropy from statistical mechanics plays a key role. Entropy measures the degree of disorder in a mechanical system, which is related to the number of possible states of the system.

2.2.5 Entropy of a data source

Consider an alphabet containing n symbols. Assume that strings over this alphabet are generated probabilistically, with the i-th symbol having probability pi. Then the entropy of the generation process (in bits) is given by the following expression:

$$H = \sum_{i=1}^{n} pi \log_2 1/pi$$

The term log, 1/pi is the number of bits needed to encode equiprobable outcomes that occur with probability pi cache. For example, the value pi 1/4 corresponds to 1/pi = 4 equally likely outcomes, which would require log 4 = 2 bits to encode. The overall expression for the entropy is a weighted sum of these bit counts, each weighted by the probability of the corresponding symbol of the alphabet.

2.2.6 Shannon's lossless coding theorem

Claude Shannon obtained a fundamental result that establishes a tight lower bound on the average code length that is needed to encode a data source without loss of information. Shannon's theorem states, in essence that any lossless compression technique must use at least as many bits per symbol, on average, as the entropy of the data source. Furthermore, the theorem asserts the

existence of codes that come arbitrarily close to achieving this limit (in other words, given any positive value e, there will exit a code that uses at most H + € bits per symbol on average to code data from a source with entropy H).

Example: Consider the example discussed above in section 2.2.2, for which the symbol probabilities are as follows:

$$P(A) = 05, P(B) = 0.25, P(C) = 0.25$$

The entropy of this data source is
$H=0.5 \log_2 2 + 0.25 \log_2 4 + 0.25 \log_2 4 = 1.5$ bits per symbol Thus, any lossless compression scheme will use at least 1.5 bits to encode each symbol
from this source, on average. The variable length code that we found above actually achieves this limit By Shannon's theorem, we know that this is the best possible result, and that no further compression is possible without some loss of information.

2.2.7 Huffman coding algorithm

Huffman's algorithm is based on the idea that a variable length code should use the shortest code words for the most likely symbols and the longest code words for the least likely symbols. In this way, the average code length will be reduced. The algorithm assigns code words to symbols by constructing a binary coding tree Each symbol of the alphabet is a leaf of the coding tree. The code of a given symbol corresponds to the unique path from the root to that leaf, with 0 or 1 added to the code for each edge along the path depending on whether the left or right child of a given node occurs next along the path.

2.3 Some of the basic Text Compression techniques

2.3.1 LZ77

LZ77 exploits the fact that words and phrases within a text file are likely to be repeated. When there is repetition, they can be encoded as a pointer to an earlier occurrence, with the pointer accompanied by the number of characters to be matched. It is a very simple adaptive scheme that requires no prior knowledge of the source and seems to require no assumptions about the characteristics of the source. In the LZ77 approach, the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window which consists of two parts a search buffer that contains a portion of the recently encoded sequence and a look ahead buffer that contains the next portion of the sequence to be encoded. The algorithm searches the sliding window for the longest match with the beginning of the look-ahead buffer and outputs a reference (a pointer) to that match. It is possible that there is no match at all, so the output cannot contain just pointers. In LZ77 the reference is always output as a triple <o,l,c>,where 'o' is an offset to the match, T is length of the match, and ‘c’

is the next symbol after the match. If there is no match the algorithm outputs a null-pointer (both the offset and the match length equal to 0) and the first symbol in the look-ahead buffer. The values of an offset to a match and length must be limited to some maximum instants. Moreover, the compression performance of LZ77 mainly depends on these values. Usually, the offset is encoded on 12-16 bits, so it is limited from 0 to 65535 symbols. So, there is no need to remember more than 65535 last seen symbols in the sliding window. The match length is usually encoded on 8 bits, which gives maximum match length equal to 255

2.3.2 LZ78

LZ78 algorithms achieve compression by replacing repeated occurrences of data with references to a dictionary that is built based on the input data stream. Each dictionary entry is of the form dictionary - (index, character). Where index is the index to a previous dictionary entry, and character is appended to the string represented by dictionary [index]. For example, "abc" would be stored in reverse order) as follows dictionary[k] = {j,'c'}, dictionary –[j]= {i, 'b'}, dictionary [i]={0,'a'}, where an index of 0 specifies the first character of a string. The algorithm initializes last matching index = 0 and next available index = 1. For each character of the input stream, the dictionary is searched for a match: {last matching index, character}. If a match is found, then last matching index is set to the index of the matching entry, and nothing is output. If a match is not found, then a new dictionary entry is created: dictionary [next available index] = (last matching index, character), and the algorithm outputs last matching index, followed by character, then resets last matching index= 0 and increments next available index. Once the dictionary is full, no more entries are added. When the end of the input Stream is reached, the algorithm outputs last matching index Note that strings are stored in the dictionary in reverse order, which an LZ78 decoder will have to deal with.

2.3.3 LZW Data Compression

If you were to take a look at almost any data file on a computer, character by character, you would notice that there are many recurring patterns. LZW is a data compression method that takes advantage of this repetition. The original version of the method was created by Lempel and Ziv in 1978 (LZ78) and was further refined by Welch in 1984, hence the LZW acronym. Like any adaptive/dynamic compression method, the idea is to

(1) start with an initial model, (2) read data piece by piece, (3) and update the model and encode the data as you go along LZW is a "dictionary"-based compression algorithm. This means that instead of tabulating character counts and building trees (as for Huffman encoding), LZW encodes data by referencing a dictionary. Thus, to encode a substring, only a single code number, corresponding to that substring's index in the dictionary, needs to be written to the output file. Although LZW is often explained in the context of compressing text files, it can be

used on any type of file However, it generally performs best on files with repeated substrings, such as text files.

LZW starts out with a dictionary of 256 characters in the case of 8 bits) and uses those as the "standard character set. It then reads data 8 bits at a time (e.g., 't','r', etc.) and encodes the data as the number that represents its index in the dictionary. Every time it comes across a new substring (say, "tr), it adds it to the dictionary; every time it comes across a substring it has already seen, it just reads in a new character and concatenates it with the current string to get a new substring. The next time LZW revisits a substring, it will be encoded using a single number. Usually a maximum number of entries (say, 4096) is defined for the dictionary, so that the process doesn't run away with memory. Thus, the codes which are taking place of the substrings in this example are 12 bits long (2^12=4096) It is necessary for the codes to be longer in bits than the characters (12 v s , 8bits), but since many frequently occurring substrings will be replaced by a single code, in the long haul, compression is achieved.

2.3.4 Huffman Coding

It is an algorithm for the Lossless compression of files based on the frequency of occurrence of a symbol in the file that is being compressed. The Huffman algorithm is based on statistical coding which means that the probability of a symbol has a direct bearing on the length of its representation. The more probable the occurrence of a symbol is, the shorter will be its bit-size representation. In any file, certain characters are used more than others. Using binary representation, the number of bits required to represent each character depends upon the number of characters that have to be represented. Using one bit we can represent two characters, Le. 0 represents the first character and 1 represents the second character. Using two bits we can represent four characters, and so on.

CHAPTER 3

LOSSY COMPRESSION

Lossy file compression results in lost data and quality from the original version lossy schemes provide much higher compression ratios than lossless, schemes.

3.1 Lossless compression technique

3.1.1. Transformation Coding

In this coding scheme, transforms such as DFT (Discrete Fourier Transform) and DCT (Discrete Cosine Transform) are used to change the pixels in the original image into frequency domain coefficients (called transform coefficients) These coefficients have several desirable properties. One is the energy compaction property that results in most of the energy of the original data being concentrated in only a few of the significant transform coefficients. This is the basis of achieving the compression. Only those few significant coefficients are selected and the remaining are discarded. The selected coefficients are considered for further quantization and entropy encoding. DCT coding has been the most common approach to transform coding, It is also adopted in the JPEG image compression standard.

3.1.2 Vector Quantization

The basic idea in this technique is to develop a dictionary of fixed-size vectors, called code vectors. A vector is usually a block of pixel values. A given image is then partitioned into non-overlapping blocks (vectors) called image Vectors. Then for each in the dictionary is determined and its index in the dictionary is used as the encoding of the original image vector. Thus, cache image is represented by a sequence of indices that can be further entropy coded.

3.1.3 Fractal Coding

The essential idea here is to decompose the image into segments by using standard image processing techniques such as colour separation, edge detection and spectrum and texture analysis. Then each segment is looked up in a library of fractals. The library actually, contains codes called iterated function system (IFS) codes, which are compact sets of numbers. Using a systematic procedure, a set of codes for a given image are determined, such that when the IFS codes are applied to a suitable set of image blocks

yield an image that is a very close approximation of the original. This scheme is highly effective for compressing images that have good regularity and self-similarity.

3.1.4 Block truncation coding

In this scheme, the image is divided into non overlapping blocks of pixels. For each block, threshold and reconstruction values are determined. The threshold is usually the mean of the pixel values in the block Then a bitmap of the block is derived by replacing all pixels whose values are greater than or equal (less than) to the threshold by a 1 (0). Then
for each segment (group of Is and 0s) in the bitmap, the reconstruction value is determined. This is the average of the values of the corresponding pixels in the original block.

3.15 Sub band coding

In this scheme, the image is analysed to produce the components containing frequencies in well-defined bands, the sub bands. Subsequently, quantization and coding is applied to each of the bands. The advantage of this scheme is that the quantization and coding well suited for each of the sub bands, can be designed separately.

3.2 Image Compression

Image compression is minimizing the size in bytes of a graphics file without degrading the quality of the image to an unacceptable level The reduction in file size allows more images to be stored in a given amount of disk or memory space. It also reduces the time required for images to be sent over the Internet or downloaded from Web pages. Some image compression formats are PNG (Portable Network Graphics) JPEG (Joint Photographic Experts Group) etc.

3.3 Audio Compression

Lossy compression is a data reduction method that reduces the amount of data during the coding process, but retains enough information to be useful. For example, an MP3 file is a lossy music file that discards some of the original data but is still acceptable

for music listening. The advantage of lossy compression is that the music file takes much less storage space. Some Audio Compression techniques are MP3 (MPEG Audio Layer 3), AAC etc.

3.4 Video Compression

Video compression uses modern coding techniques to reduce redundancy in video data Most Video compression algorithms and codecs combine spatial image compression and temporal motion compensation. Video compression is a practical implementation of source coding in information theory. In practice, most video codecs also use audio compression techniques in parallel to compress the separate but combined data streams as one package. Some Video Compression Techniques are MPEG (Moving Picture Experts Group). MPEG-1, MPEG-2, MPEG-4, MPEG-4 Part 10, or H.264 etc.

Chapter 4

HUFFMAN CODING FOR TEXT COMPRESSION AND DECOMPRESSION

The idea behind Huffman coding is to find a way to compress the storage of data variable length codes. Our standard model of storing data uses fixed length codes. For example, each character in a text file is stored using 8 bits. There are certain advantages to this system. When reading a file, we know to ALWAYS read 8bits at a time to read a single character. But as you might imagine, this coding scheme is inefficient. The reason for this is that some characters are more frequently used than other character. Let's say that some character 'e' is used 10 times more frequently than the character 'q'. It would then be advantageous for us to use a 7-bit code for e and a 9-bit code for q instead because that could shorten our overall message length.

Huffman coding finds the optimal way to take advantage of varying character frequencies in a particular fill. On average, using Huffman coding on standard files can shrink them anywhere from 10%to 30%depending to the character distribution. (The more skewed the distribution, the better Huffman coding will do).

The idea behind the coding is to give less frequent character and groups of character longer codes. Also, the coding is constructed in such a way that no two constructer codes are prefixes of each other. This property about the code is crucial with respect to easily deciphering the code.

4.1 Building a Huffman Tree

Steps to build Huffman Tree

Input is array of unique character along with their frequency of occurrence and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequency character is at root).
2. Extract two nodes with the minimum frequency the min heap.
3. Create a new internal node with frequency equal to the sum of the two nodes frequency. Make the first extracted node as its left child and the other extracted node is the root node and the tree is complete.

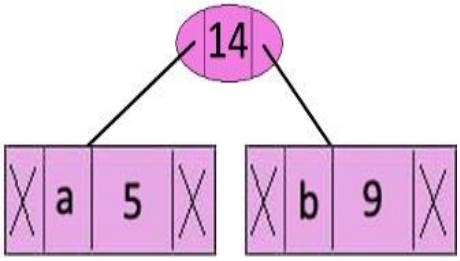
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

Character	**Frequency**
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represented root of a tree with single node.

Step 2. Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5+9=14.

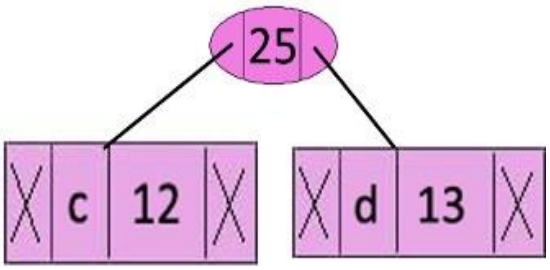


Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	**Frequency**
c	12
d	13

Internal Node	14
e	16
f	45

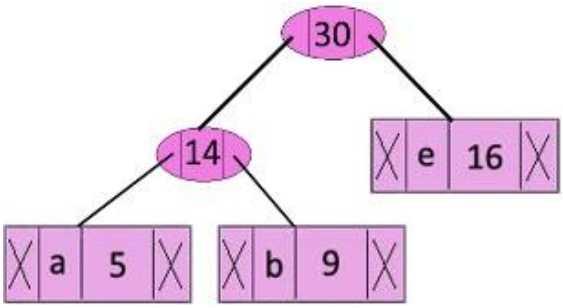
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one node.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

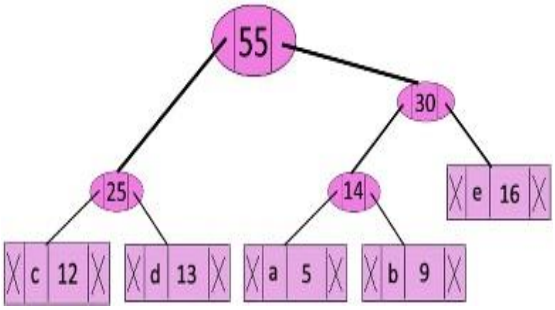
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

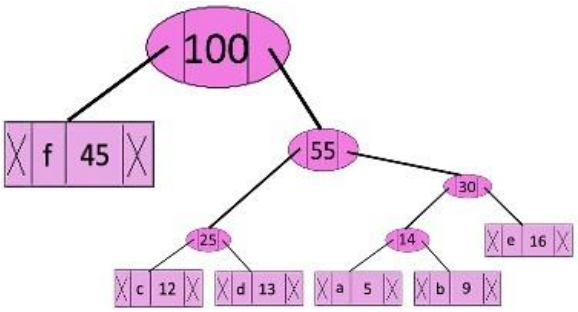
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now min heap contains 2 nodes.

character	**Frequency**
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



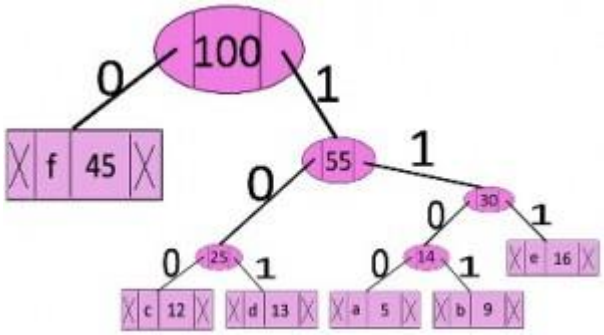
Now min heap contains only one node.

character	**Frequency**
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

4.2 Steps to print codes from Huffman Tree:

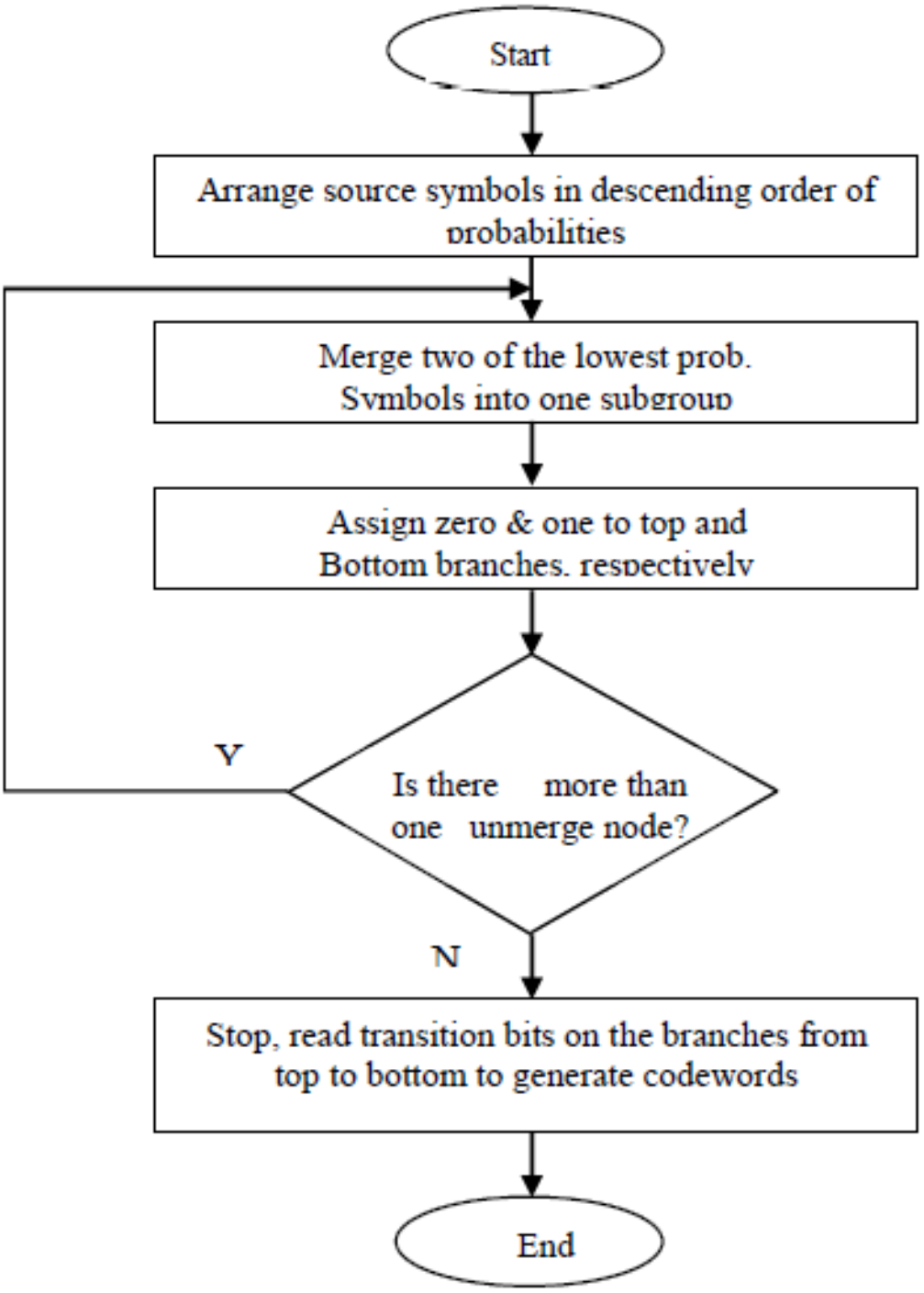
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

4.3 Flow Chart of Huffman Algorithm



4.4 Huffman Decoding

The Huffman encoding process is relatively straightforward. The symbol to-codeword mapping table provided by the modeler is used to generate the codewords for each input symbol. On the other hand, the Huffman decoding process is somewhat more complex.

4.4.1 Bit-Serial Decoding

Let us assume that the binary coding tree is also available to the decoder. In practice, this tree can be reconstructed from the symbol-to-codeword mapping table that is known to both the encoder and the decoder. The decoding process consists of the following steps:

1. Read the input compressed stream bit by bit and traverse the tree until a leaf node is reached.

2. As each bit in the input stream is used, it is discarded. When the leaf node is reached, the Huffman decoder outputs the symbol at the leaf node. This completes the decoding for this symbol.

We repeat these steps until all of the input is consumed for the example discussed in the previous section, since the longest codeword is five bits and the shortest codeword is two bits, the decoding bit rate is not the same for all symbols. Hence, this scheme has a fixed input bit rate but a variable output symbol rate.

4.4.2 Lookup-Table-Based Decoding

Lookup-table-based methods yield a constant decoding symbol rate. The lookup table is constructed at the decoder from the symbol-to-codeword mapping table Tf the longest codeword in this table is L bits, then a 2^L entry lookup table is needed Recall the first example that we presented in that section where L=5. Specifically, the lookup table construction for each symbol, is as follows:

- Let c, be the codeword that corresponds to symbols S_i. Assume that C_i has, L_1 Bits. We form an L-bit address in which the first L_i bits are C_i, and the remaining $L-l_i$ bits take on all possible combinations of zero and one. Thus, for the symbol s, there will be 2^{L-l_i} addresses.
- At each entry we form the two-tuple *(S_i,l_i).*
-

Decoding using the lookup table approach is relatively easy:

1. From the compressed input bit stream, we read in L bits into a buffer.
2. We use the L-bit word in the buffer as an address into the lookup table and obtain the corresponding symbol, say S_k. Let the codeword length be l_k. We have now decoded one symbol.

3. We discard the first l_k bits from the buffer and we append to the buffer, the next l_k bits from the input, so that the buffer has again L bits.
4. We repeat Steps 2 and 3 until all of the symbols have been decoded.

The primary advantages of lookup-table-based decoding are that it is fast and that the decoding rate is constant for all symbols, regardless of the corresponding codeword length However, the input bit rate is now variable. For image or video data, the longest codeword could be around 16 to 20 bits. Thus, in some applications, the lookup table approach may be impractical due to space constraints.
Variants on the basic theme of lookup-table-based decoding include using hierarchical lookup tables and combinations of lookup table and bit-by-bit decoding.
There are codeword construction methods that facilitate lookup-table-based decoding by constraining the maximum codeword length to a fixed-size L, but these are out of the scope of this course.

4.5 Calculating Bits Saved

All we need to do for this calculation is figure out how many bits are originally used to store the data and subtract from that how many bits are used to store the data using the Huffman code.

In the first example given, since we have six characters, let's assume each is stored with a three-bit code Since there are 133 such characters, the total number of bits used is 3133399

Now, using the Huffman coding frequencies we can calculate the new total number of bits used:

Letter	**Code**	**Frequency**	**Total Bits**
‘a’	001	12	36
‘b’	0000	2	8
‘c’	0001	7	28
‘d’	010	13	39
‘e’	011	14	42
‘f’	1	85	85
Total:			**238**

Thus, we saved 399-238 161 bits, or nearly 40% storage space.

CHAPTER 5

CONCLUSION & FUTURE WORK

The following conclusion can be drawn from the above Huffman's Coding for compression using C++:

- The above program is a simple form of Huffman's coding with the ability to compress a string.
- It is of minimum quality in terms of graphics and versatility.
- Files as well as different image/audio/video formats are not supported by it.
- This can run on any OS having a C++ compiler installed in it.
- This process of encoding or decoding is not likely used in Real World much frequently.
- Now many new coding techniques like Shannon-Fano coding Hu-Tucker coding has been developed
- Special symbols like '.' (full stop), ','(comma) etc. cannot be encoded.

Future Work:

It can be implemented using Java for a better graphic quality and versatility.

- Greedy-Algorithm can be used for a more fluent and easy programming.
- Other encoding techniques that can be implemented using Files/audio/image/ video formats can be learnt.
- Decoding Le not implemented...can also be implemented.

APPENDIX

Source Code for text Compression:

```
/*HUFFMAN ENCODING Implementation in C*/
/*SOURCE CODE*/
#include <stdio.h>
#include <string.h>
 typedef struct node_t {
        struct node_t *left, *right;
        int freq;
        char c;
} *node;
 struct node_t pool[256] = {{0}};
node qqq[255], *q = qqq - 1;
int n_nodes = 0, qend = 1;
char *code[128] = {0}, buf[1024];
 node new_node(int freq, char c, node a, node b)
{
        node n = pool + n_nodes++;
        if (freq) n->c = c, n->freq = freq;
        else {
        n->left = a, n->right = b;
        n->freq = a->freq + b->freq;
        }
        return n;
}
```

```
/* priority queue */
void qinsert(node n)
{
        int j, i = qend++;
        while ((j = i / 2)) {
        if (q[j]->freq <= n->freq) break;
        q[i] = q[j], i = j;
        }
        q[i] = n;
}
 node qremove()
{
        int i, l;
        node n = q[i = 1];

        if (qend < 2) return 0;
        qend--;
        while ((l = i * 2) < qend) {
        if (l + 1 < qend && q[l + 1]->freq < q[l]->freq) l++;
        q[i] = q[l], i = l;
        }
        q[i] = q[qend];
        return n;
}
 /* walk the tree and put 0s and 1s */
void build_code(node n, char *s, int len)
{
```

```
        static char *out = buf;
        if (n->c) {
        s[len] = 0;
        strcpy(out, s);
        code[n->c] = out;
        out += len + 1;
        return;
        }
        s[len] = '0'; build_code(n->left,  s, len + 1);
        s[len] = '1'; build_code(n->right, s, len + 1);
}
 void init(const char *s)
{
        int i, freq[128] = {0};
        char c[16];
        while (*s) freq[(int)*s++]++;
        for (i = 0; i < 128; i++)
        if (freq[i]) qinsert(new_node(freq[i], i, 0, 0));
        while (qend > 2)
        qinsert(new_node(0, 0, qremove(), qremove()));
        build_code(q[1], c, 0);
}
        void encode(const char *s, char *out)
{
        while (*s) {
        strcpy(out, code[*s]);
        out += strlen(code[*s++]);
```

```
        }
}
 void decode(const char *s, node t)
{
        node n = t;
        while (*s) {
        if (*s++ == '0') n = n->left;
        else n = n->right;
        if (n->c) putchar(n->c), n = t;
        }
        putchar('\n');
        if (t != n) printf("garbage input\n");
}
 int main(void)
{
        int i;
        const char *str = "Global";
        char buf[1024];
        init(str);
        for (i = 0; i < 128; i++)
        if (code[i]) printf("'%c': %s\n", i, code[i]);
        encode(str, buf);
        printf("encoded: %s\n", buf);
        printf("decoded: ");
        decode(buf, q[1]);
        return 0;
}
```

Sample Output:

'G': 001

'a': 000

'b': 11

'l': 01

'o': 10

encoded: 00101101100001

decoded: Global

Process exited after 0.1498 seconds with return value 0

Press any key to continue . . .

REFERENCE

[1] https//www.cs.duke.edu/csed/curious/compression/Izw.html
[2]http://learnedstuffs.wordpress.com/2013/10/18/an-implementation-of-the huffman-code-for-compressing-and-decompressing-strings/
[3]http://www.csceumbe.edu/courses/undergraduate/341/fall11/projects/project3/ HuffmanExplanation.shtml
[4] Sayood, K. (2000). Introduction to Data Compression, Second Edition
[5] San Mateo, CA: Morgan Kauffman/Academic Press. Nelson M, and Gailly, J. L (1996), The Data Compression Book. California: M&T Books
[6] www.cs.cmu.edu/-guyb/realworld/compression.pdf
[7] www.csd.uoc.gr/-hy438/lectures/Sayood-DataCompression.pdf
[8] en.wikipedia.org/wiki/Huffman_coding
[9] www.data-compression.com/lossless.html
[10] www.cs.utexas.edu/users/lavender/courses/EE360C/-/lecture-26.pdf
[11] Applied Coding and Information Theory for Engineers text book by Richard B. Wells.
[12] http://www.answers.com/topic/data-compression