Implementation of Data Compression algorithms

Project report submitted in partial fulfillment of the requirement for the degree of Bachelor of Technology
in

Computer Science and Engineering/Information Technology

By

UtkarshTyagi - 131206

Under the supervision of

Prof. Dr. SatyaPrakashGhrera,
FBCS, SMIEEE
Professor, Brig (Retd.) and Head, Dept. of CSE and IT

To

Department of Computer Science & Engineering and Information Technology
Jaypee University of Information Technology Waknaghat, Solan-173234, Himachal Pradesh

Candidate's Declaration

I hereby declare that the work presented in this report entitled **"DATA COMPRESSION"**in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from August 2016 to December 2016 under the supervision of **(Prof Dr.SatyaPrakashGhrera)** (Head of CSE Department).
The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Utkarsh Tyagi
131206

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

(Supervisor Signature)
Prof. Dr. Satya Prakash Ghrera,
FBCS, SMIEEE
Professor, Brig (Retd.)
Head, Dept. of CSE and IT

Dated

Acknowledgment

In performing our project, we had to take the help and guideline of some respected persons, who deserve our greatest gratitude. We would like to show our gratitude to Prof.Dr.SatyaPrakashGhrera,FBCS,SMIEEE professor , Brig(Retd.) and Head of Dept. of CSE and ITfor giving us a good guideline for project throughout numerous consultations. We would also like to expand our deepest gratitude to all those who have directly and indirectly guided us in our project.

Date: April 30th, '2017

Utkarsh Tyagi
131206

Table of Contents

1. INTRODUCTION **1**
1.1 Introduction 1
1.2 Problem Statement 5
1.3 Objective 7
1.4 Methodology 7
1.4.1 Huffman coding 7
1.4.2 LZW 9
1.4.3 Run-Length Encoding 10
2. LITERATURE SURVEY **11**
2.1 Huffman coding 11
2.2 LZW 16
2.2 Run-Length Encoding 19
3. SYSTEM DEVELOPMENT **20**
3.1 Huffman coding 20
3.1.1 Algorithm 21
3.1.2 Model Development 22
3.1.3 Analysis of arithmetic coding 24
3.2 LZW 26
3.2.1 Algorithm 26
3.2.2 Model Development 27
3.2.3 Analysis of LZW 28
3.3 Run-Length Encoding 29
3.3.1 Algorithm 29
3.3.2 Model Development 29
3.3.3 Analysis of run-length encoding 31
4. PERFORMANCE ANALYSIS 33
4.1 Compression Ratio 33
4.1 Compression Speed 34
4.3 Result of Huffman coding 36

4.4 Result of LZW 36
4.5 results of run length encoding 37
4.6 Comparing different techniques 37

5. CONCLUSION 39
5.1 Conclusion 39
5.2 Future Scope 39
5.3 Application Contribution 39
REFERENCES 40

List of Figures

S no.	Name	Page no.
1	Principle of data compression	4
2	Preparing of Huffman code	11
3	List event in descending order of probability	12
4	Combining C and D	12
5	Combining CD and EF	13
6	Combining A and B	13
7	Combining AB and CDEF	14
8	Functioning of Huffman code	15
9	Flowchart of LZW compression	26
10	Flowchart of LZW decompression	27

List of Tables

S no.	Name	Page no.
1	Result of Huffman coding	36
2	Result of Lzw	37
3	Result of RLE	37
4	Comparing techniques	38

Chapter-1

INTRODUCTION

1.1 Introduction

Data compression is a process that reduces the data size, removing the excessive information and redundancy. Why shorter data sequence is more suitable? –the answer is simple it reduces the cost. Data compression is a common requirement for most of the computerized application . Data compression has important application in the area of file storage and distributed system. Data compression is used in multimedia field, text documents and data base table. Data compression methods can be classified in several ways. One of the most important criteria of classification is whether the compression algorithms remove some part of data, which cannot be recovered during decompression. The algorithm, which removes some part of data, is called loss data compression. And the algorithm that achieve the same what we compressed after decompression is called lossless data compression. The loss data compression algorithm is usually used when a perfect consistency with the original data is not necessary after decompression. Example of loss data compression is compression of video or picture data. Lossless data compression is used in text file, database tables and in medical image because law of regulations. Various lossless data compression algorithm have been proposed and used. Some of main techniques are Huffman Coding, Run Length Encoding, Arithmetic Encoding and Dictionary Based Encoding. In this report we examine Arithmetic Encoding and Dictionary-based Algorithm and give comparison between them according to their performances.

Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV will be compressed using MPEG-2, and several file systems automatically compress files when stored, and the rest of us do it by hand. The neat thing about compression, as with the other topics we will cover in this course, is that the algorithms used in the real world make heavy use of a wide set of algorithmic tools, including sorting, hash tables, tries, and FFTs. Furthermore, algorithms with strong theoretical foundations play a critical role in real-world applications.

The generic term message for the objects we want to compress will be used, which could be either files or messages. The task of compression consists of two components, an encoding algorithm that takes a message and generates a "compressed" representation (hopefully with fewer bits), and a decoding algorithm that reconstructs the original message or some approximation of it from the compressed representation. These two components are typically intricately tied together since they both have to understand the shared compressed representation. We distinguish between lossless algorithms, which can reconstruct the original message exactly from the compressed message, and loss algorithms, which can only reconstruct an approximation of the original message. Lossless algorithms are typically used for text, and loss for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Loss 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 loss 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 loss since the text has changed, but the "meaning" and clarity of the message might be fully maintained, or even improved. In fact Shrunk and White might argue that good writing is the art of loss text compression.

Is there a lossless algorithm that can compress all messages? There has been at least one patent application that claimed to be able to compress all files (messages)—Patent 5,533,051 titled "Methods for Data Compression". The patent application claimed that if it was applied recursively, a file could be reduced to almost nothing. With a little thought you should convince yourself that this is not possible, at least if the source messages can contain any bit-sequence. We can see this by a simple counting argument. Let's consider all 1000 bit messages, as an example. There are 21000 different messages we can send, each, which needs to be distinctly identified by the decoder. It should be clear we can't represent that many different messages by sending 999 or fewer bits for all the messages — 999 bits would only allow us to send 2999 distinct messages. The truth is that if an algorithm shortens any one message, then some other message needs to be lengthened. You can verify this in

practice by running GZIP on a GIF file. It is, in fact, possible to go further and show that for a set of input messages of fixed length, if one message is compressed, then the average length of the compressed messages over all possible inputs is always going to be longer than the original input messages. Consider, for example, the 8 possible 3 bit messages. If one is compressed to two bits, it is not hard to convince yourself that two messages will have to expand to 4 bits, giving an average of 31/8 bits. Unfortunately, the patent was granted.

Data Compression is the procedure of encoding information to fewer bits than the first representation so it consumes less storage space and less transmission time while conveying more than a system. Data compression algorithms are classified in two ways i.e. loss and lossless data compression algorithm. Compression algorithm is utilized to change over information from a simple to-utilize arrangement to one advanced for smallness. In like manner, an uncompressing system gives back the data to its unique structure.

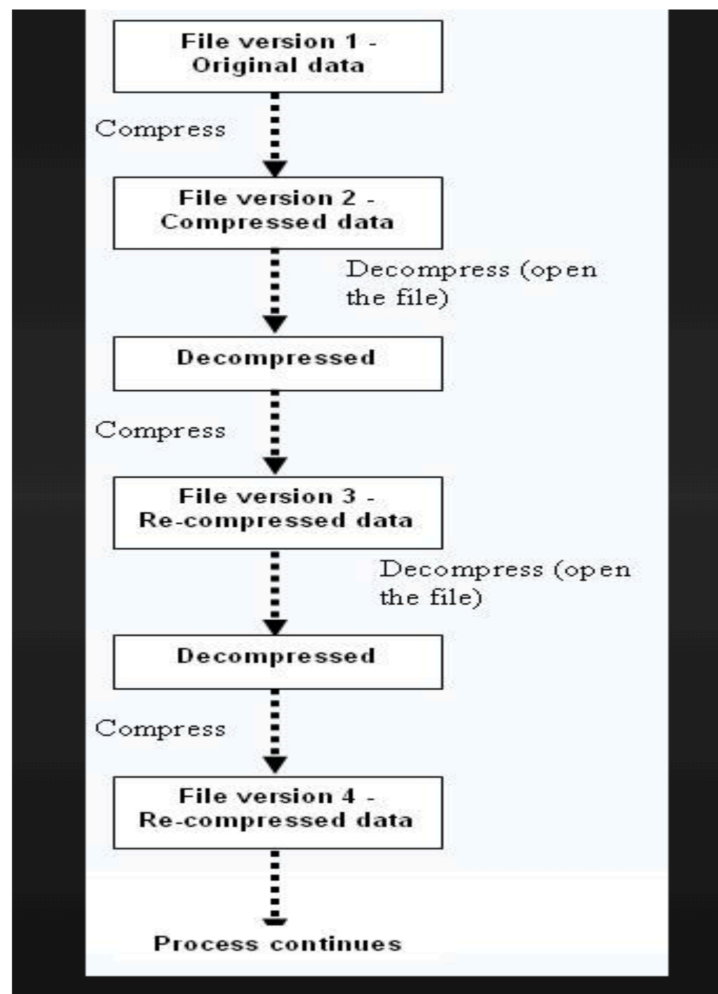

Figure 1.1: basic principle of Data Compress

1.2 Problem Statement

The fundamental problem of lossless compression is to decompose a data set (for example, a text file or an image) into a sequence of events, then to encode the events using as few bits as possible. The idea is to assign short code words to more probable events and longer code words to less probable events. Data can be compressed whenever some events are more likely than others. Statistical coding techniques use estimates of the probabilities of the events to assign the code words. Given a set of mutually distinct events e_1, e_2, e_3, _ _ _ _, e_n, and an accurate assessment of the probability distribution P of the events, Shannon proved that the smallest possible expected number of bits needed to encode an event is the entropy of P, denoted by

$$H(P) = \sum_{i=1}^{n} -p\{e_i\} \log_2 p\{e_i\}$$

Where p $\{e_{a.}\}$ is the probability that event $e_{a.}$ occurs. An optimal code outputs $\log_2$ p bits to encode an event whose probability of occurrence is p. Pure arithmetic codes supplied with accurate probabilities provide optimal compression. In theory, arithmetic codes assign one "code word" to each possible data set. The code words consist of half-open subintervals of the half-open unit interval [0,1), and are expressed by specifying enough bits to distinguish the subinterval corresponding to the actual data set from all other possible subintervals. Shorter codes correspond to larger subintervals and thus more probable input data sets. In practice, the subinterval is refined incrementally using the probabilities of the individual events, with bits being output as soon as they are known. Arithmetic codes almost always give better compression than prefix codes, but they lack the direct correspondence between the events in the input data set and bits or groups of bits in the coded output file.

A statistical coder must work in conjunction with a modeler that estimates the probability of each possible event at each point in the coding. The probability model need not describe the process that generates the data; it merely has to provide a probability distribution for the data items. The probabilities do not even have to be particularly accurate, but the more accurate they are, the better the compression will be. If the probabilities are wildly inaccurate, the file may even be expanded rather than compressed, but the original data can still be recovered. To obtain maximum compression of a file, we need both a good probability model and an efficient way of representing (or learning) the probability model.

Lossless data compression is a procedure that permits the utilization of data compression calculations to pack the content data further more permits the precise unique data to be remade from the compacted data. This is in as opposed to the loss data compression in which the careful unique data can't be recreated from the compacted data. The prevalent ZIP record organize that is being utilized for the compression of data documents is likewise a use of lossless data compression approach. Lossless compression is utilized when it is vital that the first data and the decompressed data be indistinguishable. Lossless content data compression calculations typically abuse factual excess in such a path in order to speak to the sender's data all the more briefly with no blunder or any kind of loss of vital data contained inside of the content information data. Since the majority of this present reality data has factual excess, thusly-lossless data compression is conceivable. Case in point, In English content, the letter "an" is a great deal more basic than the letter 'z', and the likelihood that the letter “z” will trail the letter “t” is little. So this sort of repetition can be evacuated utilizing lossless compression. Lossless compression techniques may be classified by kind of data they are intended to pack. Compression calculations are essentially utilized for the compression of content, pictures and sound. Most lossless compression projects utilize two various types of calculations: one which creates a factual model for the info data and another which maps the information data to bit strings utilizing this model as a part of such a route, to the point that soften as possible experienced data will deliver shorter yield than improbable (less continuous) data. The upside of lossless techniques over loss systems is that Lossless compression results are in a closer representation of the first info data. The execution of calculations can be thought about utilizing the parameters, for example, Compression Ratio and Saving Percentage. In lossless data compression document the first message can be precisely decoded.

Lossless data compression lives up to expectations by discovering rehashed examples in a message and encoding those examples in an effective way. Thus, lossless data compression is likewise alluded to as repetition decrease. Since repetition decrease is reliant on examples in the message, it doesn't function admirably on arbitrary messages. Lossless data compression is perfect for content.

1.3 Objective

Our objective it to implement the Huffman coding algorithms, LZW, run length-encoding algorithm and compare the results obtained to maximize the compression ratio and minimize the compression time.

1.4 Methodology

1.4.1 Huffman Coding

Huffman Data Compression algorithm works in three phases to compress the text data. In the first phase data is compressed with the help of dynamic bit reduction technique and in second phase unique words are to be found to compress the data further and in third and final phase Huffman coding is used to compress the data further to produce the final output. Following are the main steps of algorithm for compression and decompression:

Step I: Input the text data to be compressed.

Step II: Apply Dynamic bit Reduction method to compress the data.

Step III: Find the unique symbol to compress the data further.

Step IV: Create the binary tree with nodes representing the unique symbols

Step V: Apply Huffman coding to Finally compress the data.

Step VI: Display the final result obtained in previous step.

Huffman coding is an entropy-encoding algorithm used for lossless data compression in computer science and information theory. The term refers to the use of 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 way based on the estimated probability of occurrence for each possible value of the source symbol.

Huffman coding uses a specific method for choosing there presentation for each symbol, resulting in a prefix-free code (that is, the bit string representing some particular symbol is never a prefix of the bit string representing another symbol) that expresses the most common characters 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. A method was later found to do this in linear time if input probabilities (also known as weights) are sorted. For a set of symbols with a uniform probability distribution and a number of members which is a power of two, Huffman coding is equivalent to simple binary block encoding [e.g., ASCII coding.

Assume you have a source generating 4 different symbols {a1, a2, a3, and a4} with probability {0.4;0. 35;0. 2;0. 05}. Generate a binary tree from left to right taking the two less probable symbols, putting them together to form another equivalent symbol having a probability that equals the sum of the two symbols. Keep on doing it until you have just one symbol. Then read the tree backwards, from righ to left, assigning different bits to different branches. The final Huffman code is:
SYMBOL CODE
A1 0
A2 10
A3 111
A4 110
The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols (N). A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, Link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol weight, links to two child nodes and the optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1'represents following the right child. A finished tree has Leaf nodes and N−1 internal nodes. A linear-time* method to create a Huffman tree is to use two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues.

1.4.2 LZW

LZW compression replaces strings of characters with single codes. It does not do any analysis of the incoming text .Instead, it just adds every new string of characters it sees to a table of strings .Compression occurs when a single code is output instead of a string of characters .LZW also performs well when presented with extremely redundant data files, such as tabulated numbers, computer source code, and acquired signals.

When the LZW program starts to encode a file, the code table contains only the first 256 entries, with the remainder of the table being blank. This means that the first codes going into the compressed file are simply the single bytes from the input file being converted to 12 bits. As the encoding continues, the LZW algorithm identifies repeated sequences in the data, and adds them to the code table. Compression starts the second time a sequence is encountered. The key point is that a sequence from the input file is not added to the code table until it has already been placed in the compressed file as individual characters (codes 0 to 255). This is important because it allows the uncompressing program to *reconstruct* the code table directly from the compressed data, without having to transmit the code table separately.

The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the initialized dictionary. In order to rebuild the dictionary in the same way as it was built during encoding, it also obtains the next value from the input and adds to the dictionary the concatenation of the current string and the first character of the string obtained by decoding the next input value, or the first character of the string just output if the next value can not be decoded (If the next value is unknown to the decoder, then it must be the value that will be added to the dictionary this iteration, and so its first character must be the same as the first character of the current string being sent to decoded output). The decoder then proceeds to the next input value (which was already read in as the "next value" in the previous pass) and repeats the process until there is no more input, at which point the final input value is decoded without any more additions to the dictionary.

In this way the decoder builds up a dictionary, which is, identical to that used by the encoder, and uses it to decode subsequent input values. Thus the full dictionary does not need be sent with the encoded data; just the initial dictionary containing the single-character strings is sufficient (and is typically defined beforehand within the encoder and decoder rather than being explicitly sent with the encoded data.

1.4.3 Run Length Encoding

This algorithm consists of replacing large sequences of repeating data with only one item of this data followed by a counter showing how many times this item is repeated.

The algorithm works as follow:

a) Pick the first character from source string.

b) Append the picked character to the destination string.

c) Count the number of subsequent occurrences of the picked character and append the count to destination string.

d) Pick the next character and repeat steps b) c) and d) if end of string is NOT reached.

Chapter-2

LITERATURE SURVEY

2.1 Huffman Coding

The Huffman coding procedure finds the optimum (least rate) uniquely decodable, variable length entropy code associated with a set of events given their probabilities of occurrence. The procedure is simple enough that we can present it here.

The Huffman coding method is based on the construction of what is known as a binary tree. The path from the top or root of this tree to a particular event will determine the code group we associate with that event.

Suppose, for example, that we have six events with names and probabilities given in the table below.

Event Name	Probability
A	0.30
B	0.30
C	0.13
D	0.12
E	0.10
F	0.05

Our first step is to order these from highest (on the left) to lowest (on the right) probability as shown in the following figure, writing out next to each event its probability for since this value will drive the process of constructing the code.

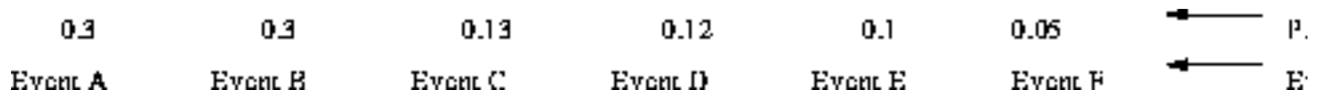

Figure 2.1 preparing for Huffman code construction

Now we perform a construction process in which we will pair events to form a new single combined event, which will replace the pair members. This step will be repeated many times, until there are no more pairs left.

First we find the two events with least combined probability. The first time we do this, the answer will always be the two right hand events. We connect them together, as shown in Figure 2.1 calling this a combined event (EF in this case) and noting its probability (which is the sum of those of E and F in this case.) We also place a 0 next to the left hand branch and a 1 next to the right hand branch of the connection of the pair as shown in the figure. The 0 and 1 make up part of the code we are constructing for these elements.

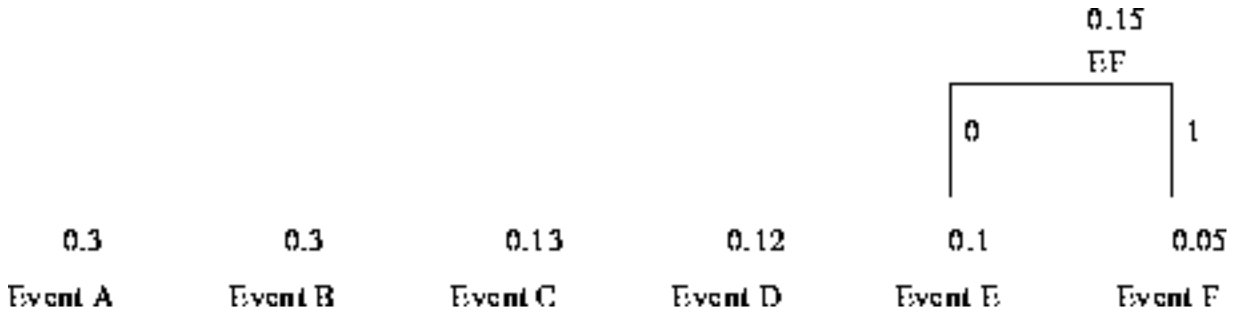

Figure2.2 list all event in descending order of probability

.

Now we repeat the last step, dealing only with the remaining events and the combined event. This time combining C and D creates a combined event with less probability than combining any others.

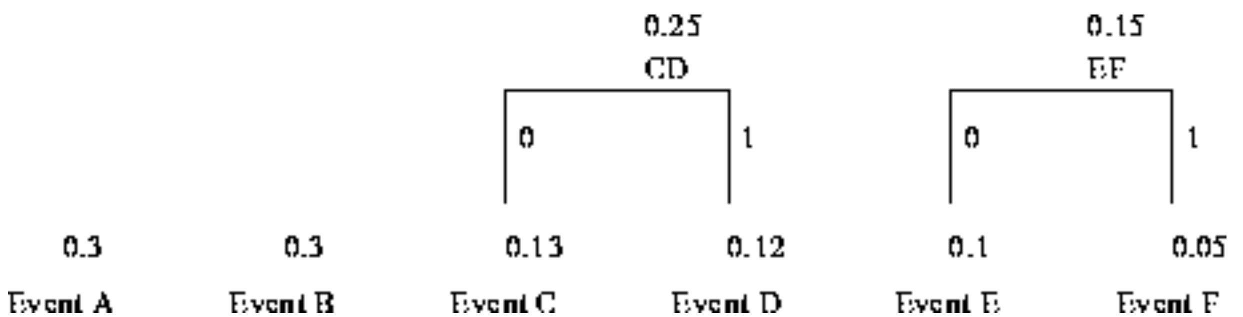

Figure 2.3 combining C and D

Again we repeat the process. This time, combining the combined events CD and EF create the new combined event with least probability.

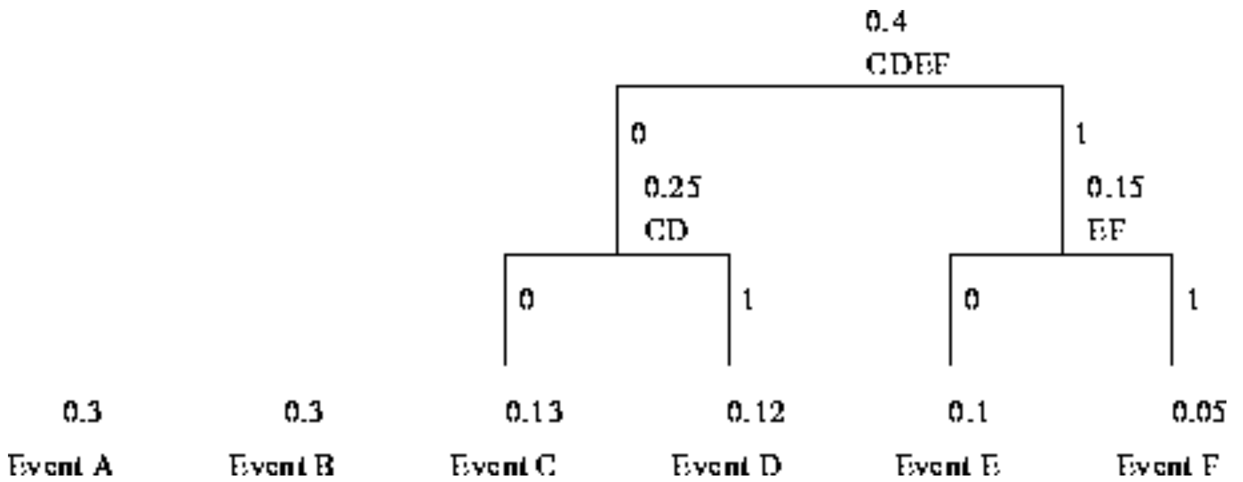

Figure2.4 combining CD and EF

The next time around the best combination is of A and B.

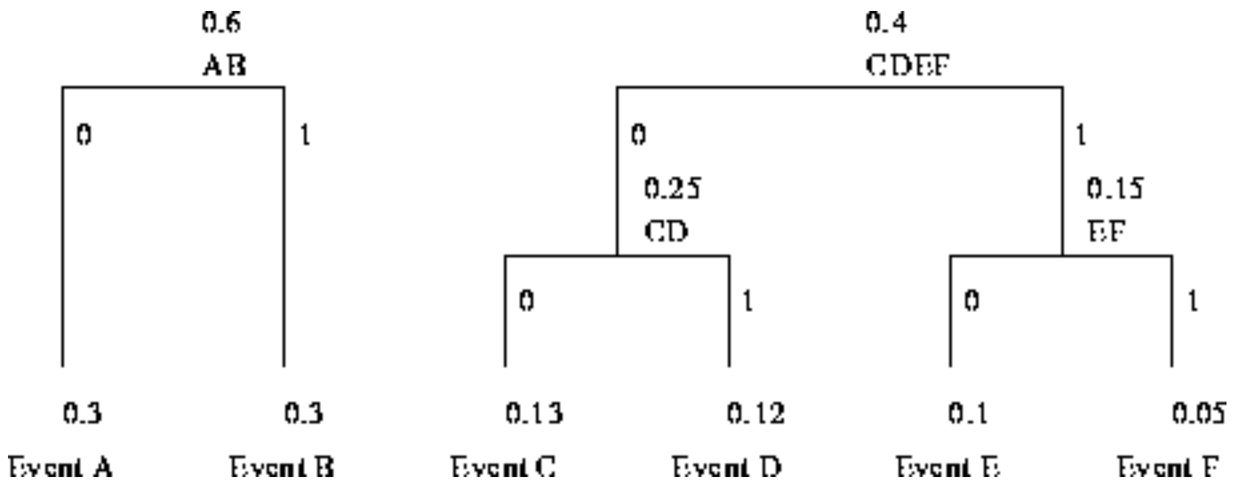

Figure 2.5 combining A and B

Finally there is only one pair left, which we simply combine.

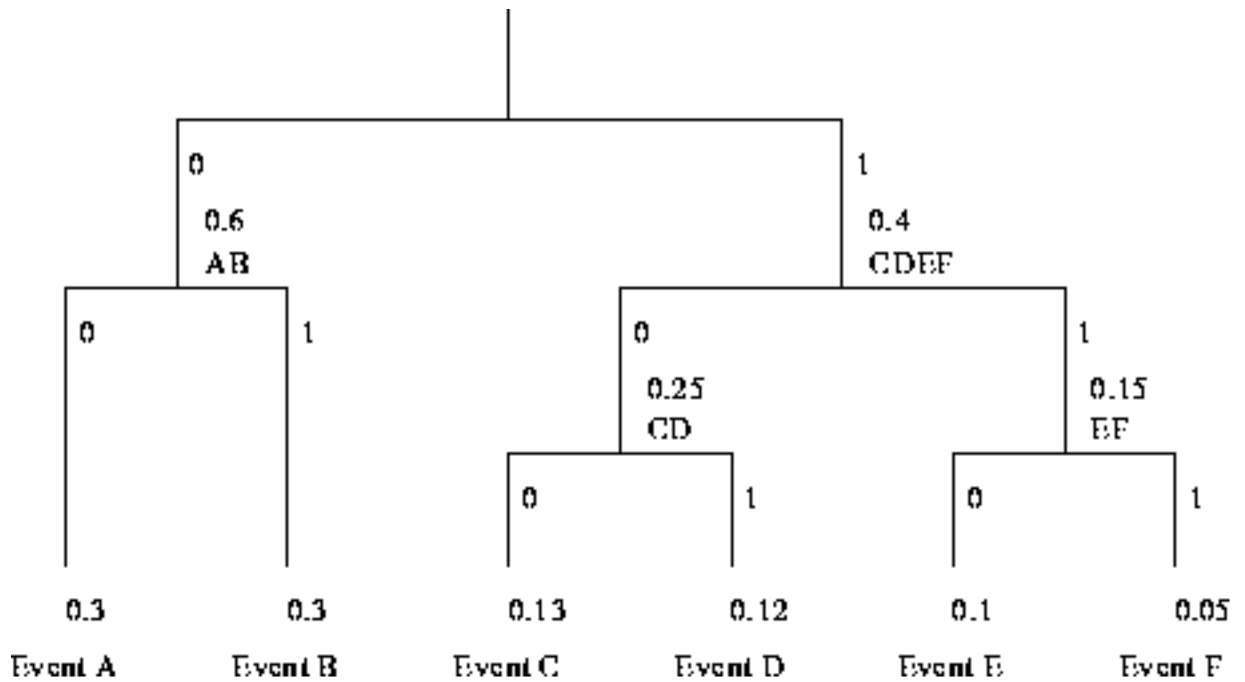

Figure 2.6 combining AB and CDEF

Having finished our connection tree, we are ready to read off of the diagram the codes that we will associate with each of the original events. To obtain the code, we start at the top level of the tree and make our way to the event we wish to code. The series of 0's and 1's we encounter along the way on the branches of the tree comprise our code. Doing so for each event in this case yields the following result.

Event Name	Probability	Code	Length
A	0.3	00	2
B	0.3	01	2
C	0.13	100	3
D	0.12	101	3
E	0.1	110	3
F	0.05	111	3

If we sum the products of the event probabilities and the code lengths for this case we obtain an average bit rate of 2.4 bits per event. If we compute the true minimum bit

rate, that is the information rate, of these events as we did with the previous example, we obtain 2.34 bits.

Suppose that we had been originally planning to code our events originally as all 3-bit codes in a fixed length code scheme. Then, if we code a document, which is long enough so that we obtain the average promised by this new scheme instead, we will find that we will obtain a compression ratio over the original scheme of 2.4/3 = 80% whereas the ultimate possible compression ratio is 2.34/3 = 78%.

It can be shown that the Huffman code provides the best compression for any communication problem for a given grouping of the events. In this problem we chose not to group events, but to code them individually. If we were to create the 36 events we would get by forming pairs of the above events, we would get substantially closer to the optimum rate suggested by the information rate calculation.

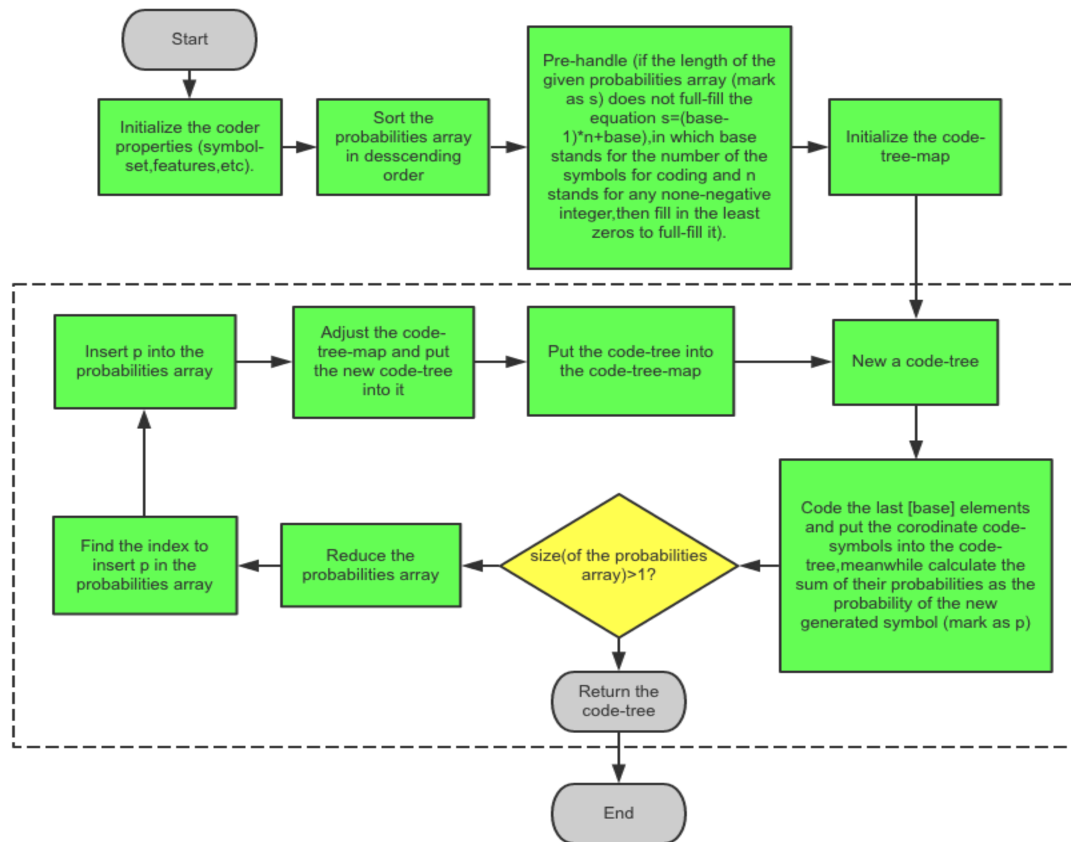

Figure 2.7 functioning of Huffman algorithm

2.2 LZW

The LZW algorithm is a greedy algorithm in that it tries to recognize increasingly longer and longer phrases that are repetitive, and encode them. Each phrase is defined to have a prefix that is equal to a previously encoded phrase plus one additional character in the alphabet. Note “alphabet” means the set of legal characters in the file. For a normal text file, this is the acai character set. For a gray level image with 256 gray levels, it is an 8-bit number that represents the pixel’s gray level.

For an instance the compression for the phrase “the/rain/in/spain/falls/mainly/on/the/plain will be as follows :

s.no	Char	String+char	In table	Output	Add to table	New string	comment
1	t	t	no			t	first charno action
2	h	tg	no	t	256	h	
3	e	he	no	h	257	e	
4	/	e/	no	e	258	/	
5	r	/r	no	/	259	r	
6	a	ra	no	r	260	a	
7	i	ai	no	a	261	i	
8	n	in	no	i	262	n	
9	/	n/	no	n	263	/	
10	i	/i	no	/	264	i	
11	n	in	yes(262)			in	first match found
12	/	in/	no	262	265	/	
13	s	/s	no	/	266	s	
14	p	sp	no	s	267	p	
15	a	pa	no	p	268	a	

16	i	ai	yes(261)			ai	
17	n	ain	no	261	269	n	
18	/	n/	yes(263)			n/	
19	f	n/f	no	263	270	f	
20	a	fa	no	f	271	a	
21	l	al	no	a	272	l	
22	l	ll	no	l	273	l	
23	s	ls	no	l	274	s	
24	/	s/	no	s	275	/	
25	m	/m	no	/	276	m	
26	a	ma	no	m	277	a	
27	i	ai	yes(261)			ai	match ai
28	n	ain	yes(269)			ain	match longer string,ain
29	l	ainl	no	269	278	l	
30	y	ly	no	l	279	y	
31	/	y/	no	y	280	/	
32	o	/o	no	/	281	o	
33	n	on	no	o	282	n	
34	/	n/	yes(263)			n/	
35	t	n/t	no	263	283	t	
36	h	th	yes(256)			th	matches th,the is not in table yet
37	e	the	no	256	284	e	the added to table
38	/	e/	yes()			e/	
39	p	e/p	no	258	285	p	

40	l	pl	no	p	286	l	
41	a	la	no	l	287	a	
42	i	ai	yes(261)			ai	matches ai
43	n	ain	yes(261)			ain	matches longer string ain
44	/	ain/	no	269		/	
45	EOF	/		/	288		end of file,output STRING

2.3 run length encoding

Run-length encoding is an information pressure calculation that is upheld by most bitmap document arrangements, for example, TIFF, BMP, and PCX. RLE is suited for compacting any sort of information paying little respect to its data content, yet the substance of the information will influence the pressure proportion accomplished by RLE. Albeit most RLE calculations can't accomplish the high pressure proportions of the more propelled pressure techniques, RLE is both simple to actualize and brisk to execute, making it a decent other option to either utilizing a perplexing pressure calculation or leaving your picture information uncompressed.

RLE works by decreasing the physical size of a rehashing series of characters. This rehashing string, called a run, is ordinarily encoded into two bytes. The primary byte speaks to the quantity of characters in the run and is known as the run number. By and by, an encoded run may contain 1 to 128 or 256 characters; the run consider more often than not contains the quantity of characters short one (an incentive in the scope of 0 to 127 or 255). The second byte is the estimation of the character in the run, which is in the scope of 0 to 255, and is known as the run esteem.

Run length encoding (RLE) is a very simple form of lossless data compression which runs on sequences having same value occurring many consecutive times and it encode the sequence to store only a single value and its count.

For example,

Consider a screen containing plain black text on a solid white background. There will be many long runs of white pixels in the blank space, and many short runs of black pixels within the text.

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWW
WWWWWWWWWWWWBWWWWWWWWWWWWWW

With a run length encoding (RLE) data compression algorithm applied to the above hypothetical scan line, it can be rendered as follows:

12W1B12W3B24W1B14W

This can be interpreted as a sequence of twelve Ws, one B, twelve Ws, three Bs, etc.

Chapter-3
SYSTEM DEVELOPMENT

3.1 Huffman Coding

3.1.1 Compression Algorithm

```
shortcreate_tree()
{
voidfind_lowest_freqs(void);
shortonly_one_up_ptr_left(void);
doublemaxfreq = 0 ;
structchardata *new_node = NULL;
fprintf(fpp,"Creating tree from frequencies...");
while (maxfreq< 0.99999 )
 {
find_lowest_freqs();
if ((new_node = (structchardata *)malloc(sizeof
(structchardata)) )
 == NULL
 )
{
printf(fpp,"Insufficient memory, malloc()
failed in create_tree().")
 ;
return FALSE;
 }
new_node->up = NULL;
new_node->left = ptr2;
new_node->right = ptr1;
new_node->charnum = -1;
ptr1->up = new_node;
```

```
ptr2->up = new_node;
new_node->frequency = ptr1->frequency + ptr2-
>frequency;
maxfreq = new_node->frequency;
#ifdef VERBOSE
fprintf(fpp,"Newly created freq == %f\n",
maxfreq);
#endif
 }
root = new_node;
if (only_one_up_ptr_left())
 {
fprintf(fpp,"Done creating tree.");
#ifdef verbose
fprintf(fpp,"Win: apparently only one remaining
up-pointer.");
#endif
}
else
{
fprintf(fpp,"Lose: apparently more than one remaining up-pointer.");
return FALSE;
 }
return TRUE;
}
```

3.1.2 Model Developement

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, n. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol weight, links to two child nodes and the optional link to a parentnode. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and $n-1$ internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process essentially begins with the leaf nodes containing the probabilities of the symbol they represent, then a new node whose children are the 2 nodes with smallest probability is created, such that the new node's probability is equal to the sum of the children's probability. With the previous 2 nodes merged into one node (thus not considering them anymore), and with the new node being now considered, the procedure is repeated until only one node remains, the Huffman tree.

The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
- Remove the two nodes of highest priority (lowest probability) from the queue.
- Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
- Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require $O(\log n)$ time per insertion, and a tree with n leaves has $2n-1$ nodes, this algorithm operates in $O(n \log n)$ time, where n is the number of symbols.

If the symbols are sorted by probability, there is a linear-time (O(n)) method to create a Huffman tree using two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:

- Start with as many leaves as there are symbols.
- Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).

While there is more than one node in the queues:

- Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
- Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
- Enqueue the new node into the rear of the second queue.
- The remaining node is the root node; the tree has now been generated.

Although linear-time given sorted input, in the general case of arbitrary input, using this algorithm requires pre-sorting. Thus, since sorting takes O(n log n) time in the general case, both methods have the same overall complexity.

In many cases, time complexity is not very important in the choice of algorithm here, since n here is the number of symbols in the alphabet, which is typically a very small number (compared to the length of the message to be encoded); whereas complexity analysis concerns the behavior when n grows to be very large.

It is generally beneficial to minimize the variance of codeword length. For example, a communication buffer receiving Huffman-encoded data may need to be larger to deal with especially long symbols if the tree is especially unbalanced. To minimize variance, simply break ties between queues by choosing the item in the first queue. This modification will retain the mathematical optimality of the Huffman coding while both minimizing variance and minimizing the length of the longest character code.

3.1.3 Analysis of Huffman coding

Although Huffman's original algorithm is optimal for a symbol-by-symbol coding (i.e., a stream of unrelated symbols) with a known input probability distribution, it is not optimal when the symbol-by-symbol restriction is dropped, or when the probability mass functions are unknown. Also, if symbols are not independent and identically distributed, a single code may be insufficient for optimality. Other methods such as arithmetic codingand LZW coding often have better compression capability: Both of these methods can combine an arbitrary number of symbols for more efficient coding, and generally adapt to the actual input statistics, useful when input probabilities are not precisely known or vary significantly within the stream. However, these methods have higher computational complexity. Also, both arithmetic coding and LZW were historically a subject of some concern over patent issues. However, as of mid-2010, the most commonly used techniques for these alternatives to Huffman coding have passed into the public domain as the early patents have expired.

However, the limitations of Huffman coding should not be overstated; it can be used adaptively, accommodating unknown, changing, or context-dependent probabilities. In the case of known independent and identically distributed random variables, combining symbols ("blocking") reduces inefficiency in a way that approaches optimality as the number of symbols combined increases. Huffman coding is optimal when each input symbol is a known independent and identically distributed random variable having a probability that is an the inverse of a power of two.

Prefix codes tend to have inefficiency on small alphabets, where probabilities often fall between these optimal points. The worst case for Huffman coding can happen when the probability of a symbol exceeds $2^{-1} = 0.5$, making the upper limit of inefficiency unbounded. These situations often respond well to a form of blocking called run-length encoding ; for the simple case of Bernoulli processes, Golomb coding is a provably optimal run-length code.

For a set of symbols with a uniform probability distribution and a number of members which is a power of two Huffman coding is equivalent to simple binary block encoding , e.g., ASCII coding. This reflects the fact that compression is not possible with such an input.

Most often, the weights used in implementations of Huffman coding represent numeric probabilities, but the algorithm given above does not require this; it requires only that the weights form a totally ordered commutative monoid, meaning a way to order weights and to add them. The Huffman template algorithm enables one to use any kind of weights (costs, frequencies, pairs of weights, non-numerical weights) and one of many combining methods (not just addition). Such algorithms can solve other minimization problems, such as minimizing$\max_i\left[w_{i}+\mathrm{length}\left(c_{i}\right)\right]$,a problem first applied to circuit design.

In the standard Huffman coding problem, it is assumed that each symbol in the set that the code words are constructed from has an equal cost to transmit: a code word whose length is N digits will always have a cost of N, no matter how many of those digits are 0s, how many are 1s, etc. When working under this assumption, minimizing the total cost of the message and minimizing the total number of digits are the same thing.

Huffman coding with unequal letter costs is the generalization without this assumption: the letters of the encoding alphabet may have non-uniform lengths, due to characteristics of the transmission medium. An example is the encoding alphabet of Morse code, where a 'dash' takes longer to send than a 'dot', and therefore the cost of a dash in transmission time is higher. The goal is still to minimize the weighted average codeword length, but it is no longer sufficient just to minimize the number of symbols used by the message. No algorithm is known to solve this in the same manner or with the same efficiency as conventional Huffman coding.

3.2. LZW

3.2.1 compression algorithm

```
  w = NIL;
while ( read a character k )
   {
    if wk exists in the dictionary
     w = wk;
    else
      add wk to the dictionary;
      output the code for w;
      w = k;
```

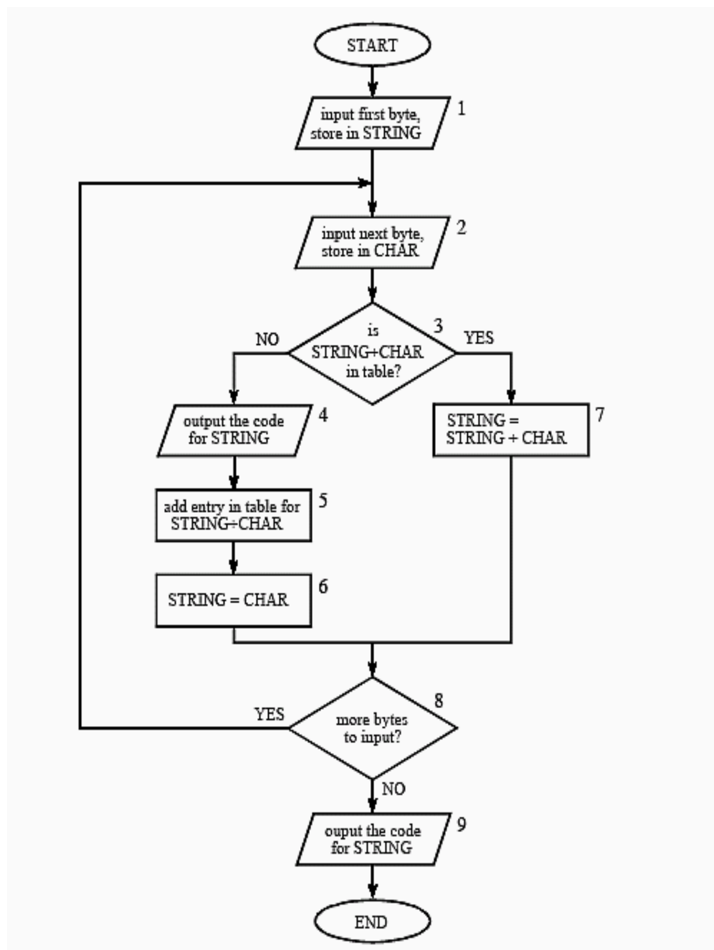

Figure 3.1 flowchart of LZW compression

3.2.2 decompression algorithm

```
read a character k;
  output k;
  w = k;
  while ( read a character k )
 /* k could be a character or a code. */
     {
      entry = dictionary entry for k;
      output entry;
      add w + entry[0] to dictionary;
      w = entry;
     }
```

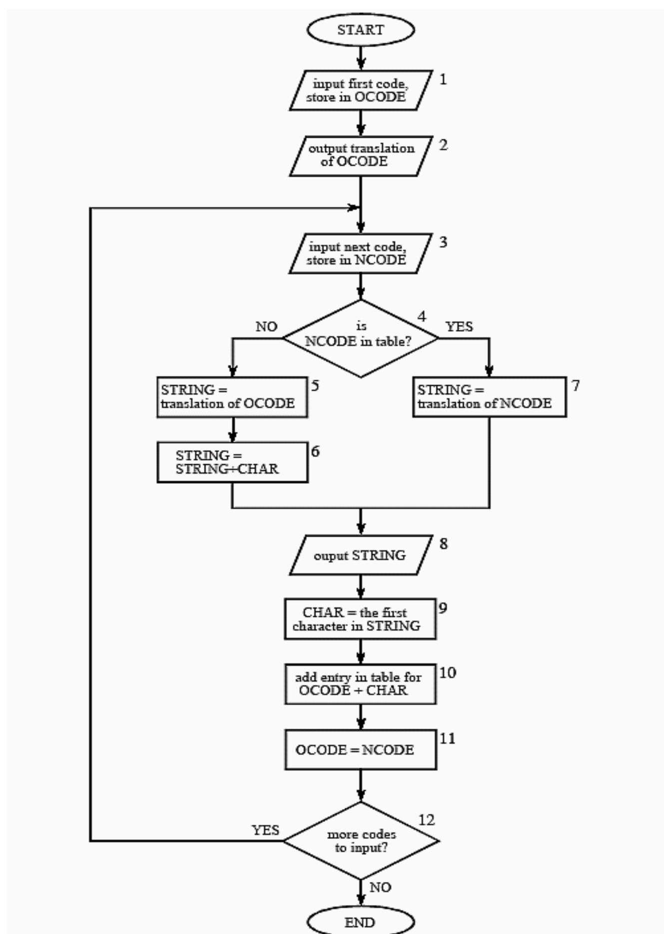

Figure 3.2 flowchart of LZW decompression

3.2.3 model development

When the LZW program starts to encode a file, the code table contains only the first 256 entries, with the remainder of the table being blank. This means that the first codes going into the compressed file are simply the single bytes from the input file being converted to 12 bits. As the encoding continues, the LZW algorithm identifies repeated sequences in the data, and adds them to the code table. Compression starts the second time a sequence is encountered. The key point is that a sequence from the input file is not added to the code table until it has already been placed in the compressed file as individual characters (codes 0 to 255). This is important because it allows the uncompression program to *reconstruct* the code table directly from the compressed data, without having to transmit the code table separately.

The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the initialized dictionary. In order to rebuild the dictionary in the same way as it was built during encoding, it also obtains the next value from the input and adds to the dictionary the concatination of the current string and the first character of the string obtained by decoding the next input value, or the first character of the string just output if the next value can not be decoded (If the next value is unknown to the decoder, then it must be the value that will be added to the dictionary this iteration, and so its first character must be the same as the first character of the current string being sent to decoded output). The decoder then proceeds to the next input value (which was already read in as the "next value" in the previous pass) and repeats the process until there is no more input, at which point the final input value is decoded without any more additions to the dictionary.

In this way the decoder builds up a dictionary which is identical to that used by the encoder, and uses it to decode subsequent input values. Thus the full dictionary does not need be sent with the encoded data; just the initial dictionary containing the single-character strings is sufficient (and is typically defined beforehand within the encoder and decoder rather than being explicitly sent with the encoded data.

3.2.4 Analysis

LZW algorithm is larger than the Huffman algorithm because the scanning window or the LZW algorithm takes more time in order to fill up the dictionary inside the LZW. Although the compression time is longer, it takes a shorter time to decompress using the LZW

algorithm than the Huffman algorithm. This is because the decoding process only needs to decode the data by matching the LZW code with the code inside the library.

3.3 Run length encoding

3.3.1 compression algorithm

```
// for input b with length l
index = 0
while( index < l )
{
  run = b[index]
  length = 0
  if run = b[++index]
    while run = b[index+length]
      index++, length++;
  output (run, length);
}
```

3.3.2 decompression algorithm

```
// for input b with length l
index = 0
while( index < l )
{
  run = b[index++]
  length = b[index++]
  output (run, length+1)
}
```

3.3.3 model development

Run-length encoding is a data compression algorithm that is supported by most bitmap file formats, such as TIFF, BMP, and PCX. RLE is suited for compressing any type of data regardless of its information content, but the content of the data will affect the compression ratio achieved by RLE. Although most RLE algorithms cannot achieve the high compression ratios of the more advanced compression methods, RLE is both easy to implement

and quick to execute, making it a good alternative to either using a complex compression algorithm or leaving your image data uncompressed.

RLE works by reducing the physical size of a repeating string of characters. This repeating string, called a *run*, is typically encoded into two bytes. The first byte represents the number of characters in the run and is called the *run count*. In practice, an encoded run may contain 1 to 128 or 256 characters; the run count usually contains as the number of characters minus one (a value in the range of 0 to 127 or 255). The second byte is the value of the character in the run, which is in the range of 0 to 255, and is called the *run value*.

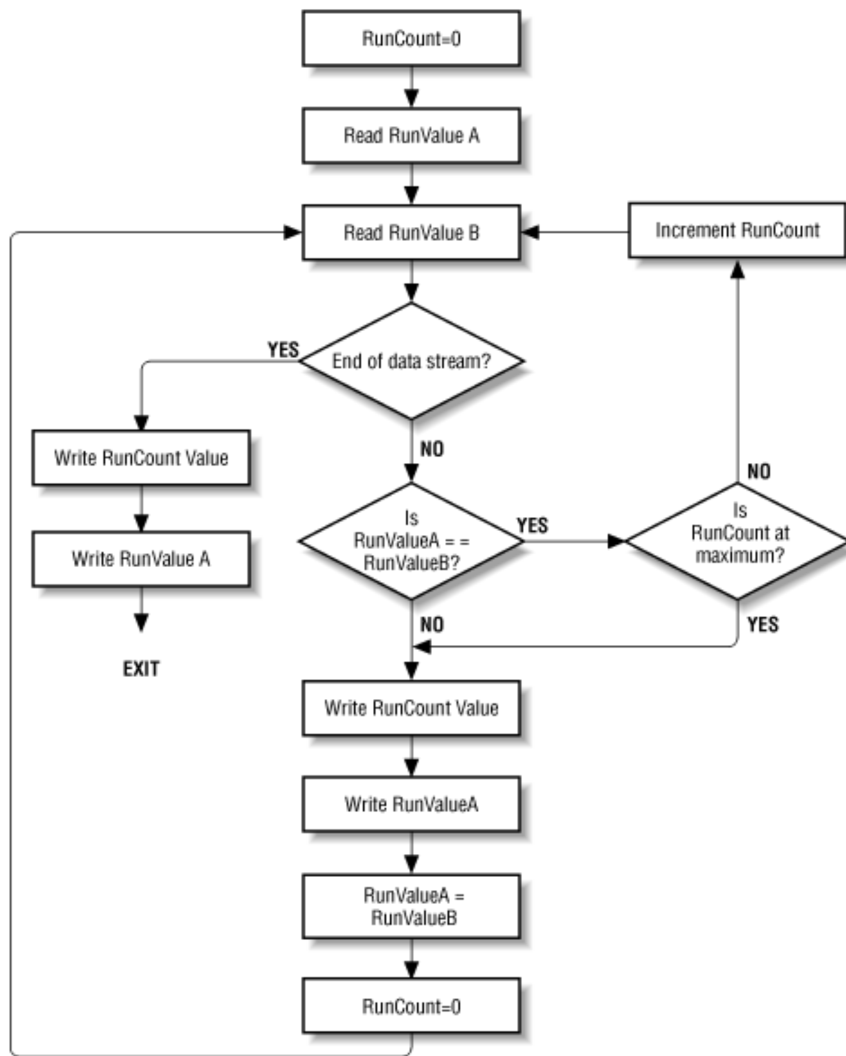

Figure 3.3 flowchart of run length encoding

3.3.4 analysis

The parts of run-length encoding algorithms that differ are the decisions that are made based on the type of data being decoded such as the length of data runs. RLE schemes used to encode bitmap graphics are usually divided into classes by the type of atomic (that is, most fundamental) elements that they encode. The three classes used by most graphics file formats are bit-, byte-, and pixel-level RLE.

Bit-level RLE schemes encode runs of multiple bits in a scan line and ignore byte and word boundaries. Only monochrome (black and white), 1-bit images contain a sufficient number of bit runs to make this class of RLE encoding efficient. A typical bit-level RLE scheme encodes runs of one to 128 bits in length in a single-byte packet. The seven least significant bits contain the run count minus one, and the most significant bit contains the value of the bit run, either 0 or 1. A run longer than 128 pixels is split across several RLE-encoded packets.

Byte-level RLE schemes encode runs of identical byte values, ignoring individual bits and word boundaries within a scan line. The most common byte-level RLE scheme encodes runs of bytes into 2-byte packets. The first byte contains the run count of 0 to 255, and the second byte contains the value of the byte run. It is also common to supplement the 2-byte encoding scheme with the ability to store literal, unencoded runs of bytes within the encoded data stream as well.

In such a scheme, the seven least significant bits of the first byte hold the run count minus one, and the most significant bit of the first byte is the indicator of the type of run that follows the run count byte. If the most significant bit is set to 1, it denotes an encoded run. Encoded runs are decoded by reading the run value and repeating it the number of times indicated by the run count. If the most significant bit is set to 0, a *literal run* is indicated, meaning that the next run count bytes are read literally from the encoded image data . The run count byte then holds a value in the range of 0 to 127 (the run count minus one). Byte-level RLE schemes are good for image data that is stored as one byte per pixel.

Pixel-level RLE schemes are used when two or more consecutive bytes of image data are used to store single pixel values. At the pixel level, bits are ignored, and bytes are counted only to identify each pixel value. Encoded packet sizes vary depending upon the size of the pixel values being encoded. The number of bits or bytes per pixel is stored in the image file header. A run of image data stored as 3-byte pixel values encodes to a 4-byte packet.

Chapter-4

PERFORMANCE ANALYSIS

Performance analysis of compression algorithms can be done by various factors. However, the main concern has always been the space efficiency and time efficiency. We are using different factors to analyze the algorithm.

4.1 Compression Ratio

Compression ratio, also known as compression power, is used to quantify the reduction in data-representation size produced by a data compression algorithm. The data compression ratio is analogous to the physical compression ratio used to measure physical compression of substances.

Data compression ratio is defined as the ratio between the uncompressed size and compressed size.

$$\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$$

Thus a representation that compresses a 10 MB file to 2 MB has a compression ratio of 10/2 = 5, often notated as an explicit ratio, 5:1 (read "five" to "one"), or as an implicit ratio, 5/1. Note that this formulation applies equally for compression, where the uncompressed size is that of the original; and for decompression, where the uncompressed size is that of the reproduction.

Sometimes the space savings is given instead, which is defined as the reduction in size relative to the uncompressed size:

$$\text{Space Savings} = 1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}}$$

Thus a representation that compresses a 10MB file to 2MB would yield a space savings of 1 - 2/10 = 0.8, often notated as a percentage, 80%.

For signals of indefinite size, such as streaming audio and video, the compression ratio is defined in terms of uncompressed and compressed data rates instead of data sizes:

$$\text{Compression Ratio} = \frac{\text{Uncompressed Data Rate}}{\text{Compressed Data Rate}}$$

Instead of space savings, one speaks of data-rate savings, which is defined as the data-rate reduction relative to the uncompressed data rate:

$$\text{Data Rate Savings} = 1 - \frac{\text{Compressed Data Rate}}{\text{Uncompressed Data Rate}}$$

For example, uncompressed songs in CD format have a data rate of 16 bits/channel x 2 channels x 44.1 kHz ≅ 1.4 Mbit/s, whereas AAC files on an iPod are typically compressed to 128 Kbit/s, yielding a compression ratio of 10.9, for a data-rate savings of 0.91, or 91%. When the uncompressed data rate is known, the compression ratio can be inferred from the compressed data rate.

4.2 Compression Speed

Compression speed is related to the data format and the machine type. The relationship between application performance and host machine parameters is a research topic that is outside of the scope of this paper. During the experiments, we keep using the same machine for all the compressions, and make sure that our application is the only workload. This way, we can think of compression speed as a function of compression algorithm. The compression speed is also affected by compression buffer size, but we omit this factor by using the "Compression is an important technique in the multimedia computing field. This is because we can reduce the size of data and transmitting and storing the reduced data on the Internet and storage devices are faster and cheaper than uncompressed data. Many image and video compression standards such as JPEG, JPEG2000, and MPEG-2, and MPEG-4 have been proposed and implemented. In all of them entropy coding, arithmetic and Huffman

algorithms are almost used. In other words, these algorithms are important parts of the multimedia data compression standards. In this paper we have focused on these algorithms in order to clarify their differences from different points of view such as implementation, compression ratio, and performance." We have explained these algorithms in detail, implemented, and tested using different image sizes and contents." From implementation point of view, Huffman coding is easier than arithmetic coding. Arithmetic algorithm yields much more compression ratio than Huffman algorithm while Huffman coding needs less execution time than the arithmetic coding. This means that in some applications that time is not so important we can use arithmetic algorithm to achieve high compression ratio, while for some applications that time is important such as real-time applications, Huffman algorithm can be used.""

same size of buffer, which is 16KB.

When evaluating data compression algorithms, speed is always in terms of uncompressed data handled per second.

Some applications use data compression techniques even when they have so much RAM and disk space that there's no real need to make files smaller. File compression and delta compression are often used to speed up copying files from one end of a slow connection to another. Even on a single computer, some kinds of operations are significantly faster when performed on compressed versions of data rather than directly on the uncompressed data. In particular, some compressed file formats are designed so that compressed pattern matching -- searching for a phrase in a compressed version of a text file -- is significantly faster than searching for that same phrase in the original uncompressed text file.

$$\text{speed} = \frac{\text{Uncompressed bits}}{\text{seconds to compress}}$$

In a few applications, the compression speed is critical. If a particular implementation of an audio compressor running on a prototype voice recorder cannot sustain 7 bits/sample/channel x 1 channel x 8 kSamples/s = 56 kbit/s from the microphones to storage, then it is unusable. No one wants their recorded voice to have silent gaps where the hardware could not keep up.

No one will buy it unless you switch to a different implementation or faster hardware (or both) that can keep up with standard telephone-quality voice speeds.

The speed varies widely from one machine to another, from one implementation to another. Even on the same machine and same benchmark file and same implementation source code, using a different compiler may make a decompressor run faster.The speed of a compressor is almost always slower than the speed of its corresponding decompressor.

Even with a fast modern CPU, compressed file system performance is often limited by the speed of the compression algorithm. Many modern embedded systems -- as well as many of the early computers that data compression algorithms were first developed on -- are heavily constrained by speed.

4.3 Results of Huffman coding

File	Filesize	Comp.size	Space saving	Time	Comp.ratio
File 1	1384 bytes	771 bytes	4901 bits	22 ms	1.795
File 2	2768 bytes	1542 bytes	9802 bits	91.6 ms	1.645
File 3	5992 bytes	3313 bytes	21425 bits	442.1 ms	1.808

Table 1: result of Huffman coding

4.4 Results of LZW

File	File size	Comp. size	Space saving	Time	Comp.ratio
File 1	1360 bytes	1067 bytes	293 bytes	3.00 ms	1.274
File 2	2719 bytes	1730 bytes	989 bytes	6.001 ms	1.571
File 3	4079 bytes	2396 bytes	1683 bytes	8.04 ms	1.702

Table 2: result of LZW

4.5 Results of run length encoding

File	File size	Comp. size	Space saving	Time	Comp.ratio
File 1	1856 bytes	633 bytes	1223 bytes	0.015ms	2.932
File 2	1398 bytes	407 bytes	991 bytes	0.016ms	3.434
File 3	8760 bytes	2737 bytes	6023 bytes	0.019	3.200

Table 3:result of RLE

4.6 comparison of different compression techniques

Technique	File size	Comp. size	Space saving	Time	Comp.ratio
Huffman	794 bytes	428 bytes	366 bytes	6.2ms	1.855
LZW	795 bytes	701 bytes	94 bytes	3ms	1.134
RLE	795 bytes	511 bytes	284 bytes	1 ms	1.555

Table 4:comparison of different techniques

Chapter-5
CONCLUSION

5.1 Conclusion

Compression is an important technique in the multimedia computing field. This is because we can reduce the size of data and transmitting and storing the reduced data on the Internet and storage devices are faster and cheaper than uncompressed data. Many image and video compression standards such as JPEG, JPEG2000, and MPEG-2, and MPEG-4 have been proposed and implemented. In all of them entropy coding, arithmetic and Huffman algorithms are almost used. In other words, these algorithms are important parts of the multimedia data compression standards. In this paper we have focused on these algorithms in order to clarify their differences from different points of view such as implementation, compression ratio, and performance. We have explained these algorithms in detail, implemented, and tested using different image sizes and contents. From implementation point of view, Huffman coding is easier than arithmetic coding. Arithmetic algorithm yields much more compression ratio than Huffman algorithm while Huffman coding needs less execution time than the arithmetic coding. This means that in some applications that time is not so important we can use arithmetic algorithm to achieve high compression ratio, while for some applications that time is important such as real-time applications, Huffman algorithm can be used.

LZW algorithm is larger than the Huffman algorithm because the scanning window or the LZW algorithm takes more time in order to fill up the dictionary inside the LZW. Although the compression time is longer, it takes a shorter time to decompress using the LZW algorithm than the Huffman algorithm. This is because the decoding process only needs to decode the data by matching the LZW code with the code inside the library.

5.2 Future scope

LZW is Easy to implement , Fast compression, Dictionary based technique.

Produce a lossless compression of images

With the advancements in compression technology, it is now very easy and efficient to compress video ,text ,images or standard data files

LZW compression became the first widely used universal data compression method on computers

text file can typically be compressed via LZW to about half its original size.

LZW became very widely used when it became part of the GIF , TIFF and pdf file.

Huffman is widely used in all the mainstream compression formats that you might encounter - from GZIP, PKZIP (winzip etc) and BZIP2, to image formats such as JPEG and PNG.

All compression schemes have pathological data-sets that cannot be meaningfully compressed; the archive formats I listed above simply 'store' such files uncompressed when they are encountered.

Newer arithmetic and range coding schemes are often avoided because of patent issues meaning Huffman remains the work-horse of the compression industry.

5.3 application contribution

Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV will be compressed using MPEG-2, and several file systems automatically compress files when stored, and the rest of us do it by hand. The neat thing about compression, as with the other topics we will cover in this course, is that the algorithms used in the real world make heavy use of a wide set of algorithmic tools, including sorting, hash tables, tries, and FFTs. Furthermore, algorithms with strong theoretical foundations play a critical role in real-world applications.

The generic term message for the objects we want to compress will be used, which could be either files or messages. The task of compression consists of two components, an encoding algorithm that takes a message and generates a “compressed” representation (hopefully with fewer bits), and a decoding algorithm that reconstructs the original message or some approximation of it from the compressed representation. These two components are typically intricately tied together since they both have to understand the shared compressed representation. We distinguish between lossless algorithms, which can reconstruct the original message exactly from the compressed message, and loss algorithms, which can only reconstruct an approximation of the original message. Lossless algorithms

are typically used for text, and loss for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Loss 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..

REFERENCES

[1] Khalid Sayood, "Introduction to Data Compression", Ed Fox (Editor), March 2000.

[2] Burrows M., and Wheeler, D. J. 1994," A Block-Sorting Lossless Data Compression Algorithm" SRC Research Report 124, Digital Systems Research Center.

[3] C.E. Shannon, "A mathematical theory of communication," Bell Syst. Tech. J., vol. 27, pp. 398-403.

[4] Glen G. Langdon, Jr, "An Introduction to Arithmetic Coding." IBM Research Division, California.

[5]"Data Compression Methodologies for LossLess Data and Comparison between Algorithms",IJESIT Volume 2, Issue 2, March 2013.

[6] Amir Said, "Introduction to Arithmetic Coding - Theory and Practice",Imaging Systems Laboratory, 2004.

[7] Somefun, M. Adebayo & Adewale, "Evaluation of dominant text data compression techniques," IJAIEM, 2014.

[8] I.H. Witten, R.M. Neal, and J.G. Cleary, "Arithmetic Coding for the data compression," Commun. ACM, vol. 30, no. 6, pp. 520-540, June 1987.

[9] R. Pasco," Source coding algorithms for fast data compression," Stanford Univ., Ph.D. dissertation, 1976.

[10] J.J. Rissanen, "Generalized Kraft inequality and arithmetic coding," IBM J. Res. Devel. , vol. 20, no. 3, pp. 198-203, May 1976.

[11] F. Rubin, " Arithmetic stream coding using fixed precision registers," IEEE Trans. Information Theory, vol. IT-25, no. 6, pp. 520-540, June 1987.

[12] J.J. Rissanen and G.G. Langdon ," Arithmetic coding," IBM J. Res. Devel, vol. 23 no. 2, pp. 146-162, Mar. 1979.

[13] M. Guazoo, "A general minimum-redundancy source-coding algorithm," IEEE Trans. Information Theory, vol. IT-26, no. 1, pp. 15-25, Jan 1980.

[14] ManjeetKaur, Er. UpasnaGarg," Lossless Text Data Compression Algorithm Using Modified Huffman Algorithm," IJARCSSE, vol. 5, Issue. 7, 2015.

[15] A. Said, "Comparative Analysis of Arithmetic Coding Computational Complexity," Hewlett Packard Laboratories Report, HPL–2004–75, Palo Alto, CA, April 2004.

[16] M. Schindler, “A fast renormalization for arithmetic coding,” Proc. IEEE Data Compression Conf., 1998.

[17] Texas Instruments Incorporated, “TMS320C6000 CPU and Instruction Set Reference Guide,” Literature Number: SPRU189F, Dallas, TX, 2000.

[18] International Business Machines Corporation, “PowerPC 750CX/CXe RISC Microprocessor User’s Manual,” (preliminary edition), Hopewell Junction, NY, 2001.

[19] Intel Corporation, “Intel Pentium 4 Processor Optimization,” Reference Manual 248966, Santa Clara, CA, 2001.

[20] Sun Microsystems Inc., “UltraSPARC III Technical Highlights,” Palo Alto, CA, 2001.

[21] Welch, Terry (1984), “A Technique for High-Performance Data Compression”. (6): 8–19. doi:10.1109/MC.1984.1659158.

[22] Jump up ^ Ziv, J.; Lempel, A. (1978). “Compression of individual sequences via variable-rate coding”, IEEE Transactions on Information Theory. 24 (5): 530. doi:10.1109/TIT.1978.1055934

[23] D.S. Taubman and M.W. Marcellin, “JPEG 2000: Image Compression Fundamentals,” Standards and Practice, Kluwer Academic Publishers, Boston, MA, 2002.

Web References:

http://www.stringology.org/DataCompression/ak-int/index_en.html

http://akbar.marlboro.edu/~mahoney/courses/Fall01/computation/compression/ac/ac_arithmetic.html

http://cotty.16x16.com/compress/nelson1.htm

http://www.drdobbs.com/parallel/arithmetic-coding-and-statistical-modeli/184408491

http://www.dspguide.com/ch27/5.htm

https://www.cs.duke.edu/csed/curious/compression/lzw.html