High Performance Lossless Multimedia Data Compression through Improved Dictionary

K. R. Kolhe
Lecturer, Department of IT Engineering, Bharati Vidyapeeth Deemed University, College of Engineering, Pune-Satara Road, Pune-411043, Maharashtra, India

P. R. Devale
Professor, Department of IT Engineering, Bharati Vidyapeeth Deemed University, College of Engineering, Pune-Satara Road, Pune-411043, Maharashtra, India

P. Shrivastava
A-501, Shri Datta Niwas, Dattanagar, Ambegaon, Katraj, Pune-411 043, Maharashtra, India

ABSTRACT
The advent of modern electronic world has opened up various fronts in multimedia interaction. They are used in various fields for various purposes of education, entertainment, research and many more. This has led to storage and retrieval of multimedia content regularly. But due to limitations of current technology the disk space and the transmission bandwidth fall behind in the race with the requirement of multimedia content. This imposes a need to compress multimedia content so that they can be easily stored requiring lesser space and easily transferred from one point to another. Some online dictionary based compression technique can be applied to reduce the data packet size. When the repetition rate of the same symbols within the data are high the compression techniques works very well. During the process of encoding and decoding, the building of online dictionary in the primary memory ensures the single pass over the data, and the dictionary need not to be transmitted over the network. Our proposed Improved Dictionary technique scans the data byte-wise, so that the chances of repetition of individual symbols are higher for text messages. Fixed length coding transmits fixed length codes for all dictionary entries. For bigger messages better optimization in terms of size reduction can be achieved through variable length coding with L-Z technique, where transmitted code length corresponding to individual dictionary entries will vary according to the requirement dynamically.

KEY WORDS: Multimedia Compression**,** Lossless and lossy Compression, LZ-77, LZ-78, LZW

1. INTRODUCTION
Many like to accumulate data and hate to throw anything away. No matter how big a storage device one has, sooner or later it is going to overflow. Data compression is useful because it delays this inevitability. As storage devices get bigger and cheaper, it becomes possible to create, store, and transmit larger and larger data files. In the old days of computing, most files were text or executable programs and were therefore small. No one tried to create and process other types of data simply because there was no room in the computer. In the 1970s, with the advent of semiconductor memories and floppy disks, still images, which require bigger files, became popular. These were followed by audio and video files, which require even bigger files. We hate to wait for data transfers. When sitting at the computer, waiting for a Web page to come in or for a file to download, we naturally feel that anything longer than a few seconds is a long time to wait. Compressing data before it is transmitted is therefore a natural solution.

CPU speeds and storage capacities have increased dramatically in the last two decades, but the speed of mechanical components (and therefore the speed of disk in- put/output) has increased by a much smaller factor. Thus, it makes sense to store data in compressed form, even if plenty of storage space is still available on a disk drive. Compare the following scenarios: (1) A large program resides on a disk. It is read into memory and is executed. (2) The same program is stored on the disk in compressed form. It is read into memory, decompressed, and executed. It may come as a surprise to learn that the latter case is faster in spite of the extra CPU work involved in decompressing the program. This is because of the huge disparity between the speeds of the CPU and the mechanical components of the disk drive.

A similar situation exists with regard to digital communications. Speeds of communications channels, both wired and wireless are increasing steadily but not dramatically. It therefore makes sense to compress data sent on telephone lines between fax machines, data sent between cellular telephones, and data (such as web pages and television signals) sent to and from satellites. The field of data compression is often called source coding. We imagine that the input symbols (such as bits, ASCII codes, bytes, audio samples, or pixel values) are emitted by a certain information source and have to be coded before being sent to their destination. The source can be memory less, or it can have memory. In the former case, each symbol is independent of its predecessors.

Data compression is the general term for the various algorithms and programs developed to address this problem. A compression program is used to convert data from an easy-to-use format to one optimized for compactness. Likewise, an uncompression program returns the information to its original form.[1]

2. COMPRESSION TECHNIQUES
Compression Techniques use two algorithms namely Compression and Reconstructions. The compression algorithm that takes an input ‘X’ and generates a representation, ‘Xc’ to generate the reconstruction ‘Y’. It will follow convention and refer to both the compression and reconstruction algorithms together to mean the compression algorithm.

Based on the requirements of reconstruction, data compression schemes can be divided into two broad classes: Loss less compression schemes in which is ‘Y’ identical to ‘X’, and lossy compression schemes, which generally provide much higher compression than lossless compression but allow ‘Y’ to be different from ‘X’

Table 2.1 shows two different ways that data compression algorithms can be categorized [2]. Methods have been classified

as either lossless or lossy. Compression techniques that allow this type of degradation are called **lossy**. This distinction is important because lossy techniques are much more effective at compression than lossless methods. The higher the compression ratio, the more noise added to the data.

Table 2.1

Lossless	Lossy
run-length	CS&Q
Huffman	JPEG
delta	MPEG
LZW	

a. Lossless or Lossy

Method	Group size: input	output
CS&Q	fixed	fixed
Huffman	fixed	variable
Arithmetic	variable	variable
run-length, LZW	variable	fixed

b. Fixed or variable group size

Lossless compression ratios are generally in the range of 2:1 to 8:1.

Lossy compression, works on the assumption that the data doesn't have to be stored perfectly. Much information can be simply thrown away from images, video data, and audio data, and the when uncompressed; the data will still be of acceptable quality.

2.1 RUN LENGTH EN CODING

"Run length encoding (RLE)" is the simplest techniques of data compression [3]&[4], It's also known as "run length limiting (RLL)". Let a text file in which the same characters are often repeated, one after another. This redundancy provides an opportunity for compressing the file. Compression software can scan through the file, find these redundant strings of characters, and then store them using an escape character (ASCII 27), followed by the character and a binary count of the number of times it is repeated. For example, the 50 character sequence:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXabcs…
can be converted to: <ESC>X<31>abcs…

It eliminates 28 characters, compressing the text by more than a factor of two. The compression software must be smart enough not to compress strings of two or three repeated characters, since for three characters run length encoding would have no advantage, and for two it would actually increase the size of the output file.

In this scheme have two potential problems. First, an escape character may actually occur in the file. Then it uses two escape characters to represent it, which can, once again, actually make the output file bigger if the uncompressed input file includes lots of escape characters.

The second problem is that a single byte cannot specify run lengths greater than 256. This difficulty can be dealt with by using multiple escape sequences to compress one very long string.

Run length encoding is really not very useful for compressing text files, a typical text file doesn't have a lot of long, repetitive character strings. It is very useful for compressing bytes of a monochrome image file, which normally consists of solid black picture bits, or "pixels", in a sea of white pixels, or the reverse. It can also be used effectively with colour graphics files that consist of large simple blocks of a single colour.

2.2 HUFFMAN CODING

"Huffman coding" is a more sophisticated and efficient lossless compression technique [3],[4]&[1], in which the characters in a data file are converted to a binary code, where the most common characters in the file have the shortest binary codes, and the least common have the longest.

The Huffman [6] code is a Variable Length Code (VLC), which means that symbols may be mapped into code words with different number of bits. The idea with Huffman encoding is that symbols that occur more frequently are coded with shorter code words. This means that the probability of the occurrence of each symbol must be known. The Huffman code is constructed by building a tree called Huffman tree, where each symbol corresponds to a leaf in the Huffman tree. The two symbols with lowest probability are combined into a new node in the tree. The probability of this node is the sum of the probability of two merged symbols. The two branches from the new node are assigned with 0 and 1 respectively. The procedure of combining two leaves and/or nodes with lowest probability is then repeated until the root node is reached. The probability of the root node is 1 because it is the sum of the probabilities for all symbols. The code word for each symbol is obtained from the Huffman tree.

3. DICTIONARY BASED CODING

3.1 L-Z-W CODING

LZ-77 is an example of "substitutional coding". Lempel and Ziv came up with an improved scheme in 1978, appropriately named "LZ-78", and it was refined by a Mr. Terry Welch in 1984, making it "LZW". LZ-77 uses pointers to previous words or parts of words in a file to obtain compression. LZW takes that scheme one step further, actually constructing a "dictionary" of words or parts of words in a message, and then using pointers to the words in the dictionary [3].

Let's the example messages are:-

how_many_cans_can_a_canner_can

The LZW algorithm stores strings in a "dictionary" with entries for 4,096 variable-length strings. The first 255 entries are used to contain the values for individual bytes, so the actual first string index is 256. As the string is compressed, the dictionary is built up to contain every possible string combination that can be obtained from the message, starting with two characters, then three characters, and so on.

Dictionary contains the following strings:

256 -> ho 263 -> y_ 270 -> _a
257 -> ow 264 -> _c 271 -> a_
258 -> w_ 265 -> ca 272 -> _can
259 -> _m 266 -> ans 273 -> nn
260 -> ma 267 -> s_ 274 -> ne
261 -> an 268 -> _ca 275 -> er
262 -> ny 269 -> an_ 276 -> r_

The compressed output for example looks like this

how_many_c<261>s<264><261>_a<268>nner<272>

3.2 L-Z-W DATA COMPRESSION

LZW Fundamentals

The original Lempel Ziv approach to data compression was first published in 1977. Terry Welch's refinements to the algorithm were published in 1984. The algorithm is surprisingly simple. 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. The code that the LZW algorithm outputs can be of any arbitrary length, but it must have more bits in it than a single character. The first 256 codes (when using eight bit characters) are by default assigned to the standard character set. The remaining codes are assigned to strings as the algorithm proceeds. The sample program runs as shown with 12 bit codes. This means codes 0-255 refer to individual bytes, while codes 256-4095 refer to sub strings.

LEMPEL-ZIV [1977], OR LZ-77

Lempel-Ziv [1977], or LZ-77 is an adaptive dictionary-based compression algorithm [4]. Its main data structure is a text window, divided into two parts. The first part is a large block of decoded text held in a fixed-size window and the second part is a look-ahead buffer which has characters read in from the input but not yet encoded.

Symbols within the buffer are then compared with data in the fixed-size window. The algorithm tries to match the **contents** of the look-ahead buffer to a string in the fixed-size window.

In general, dictionary-based compression replaces phrases with tokens. If the number of bits in the token is less than the number of bits in the phrase, compression will occur.

Example:-

lease.' 'Er...Kishorkolhe, ' said	Kishorkolhe. ' I s
Fixed-size window of previously read data (32 bytes)	Look ahead buffer (16 bytes)

In the example, if any matches are found then a token is passed to the output stream describing the match. After matching a symbol, the data is shifted by an amount equal to the length of the symbol represented by the previous token. Data is pushed out of the token and new data is read into the buffer as shown below:

The three items in the token are:

(1) Length of an offset to a phrase in the fixed-size window

(2) The length of the phrase

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

In the example above, the output token would be (17, 11, '.') (phrase includes spaces). The LZ77 algorithm first emits the token, then shifts text window over 10 characters, which is the width of the phrase just encoded. 11 new symbols are then read into the look-ahead buffer, and the process repeats.

.. Kishorkolhe, ' said Kishorkolhe.	' I said what's

The next token issued by the compression algorithm would encode the symbol " ' " as (16, 1, ' '). The look-ahead buffer shown above has no match for " I ", thus it could be encoded a single character at a time using a phrase length of zero (0, 0, ' ').

The syntax of this token allows for phrases that have no match of any length in the window. Therefore, this method is not efficient, but it ensures that the algorithm can encode any message.

3.3 THE CATCH

There is a single exception case in the LZ-78 compression algorithm that causes some trouble to the decompression side [5]. If there is a string consisting of a (STRING, CHARACTER) pair already defined in the table, and the input stream then sees a sequence of STRING, CHARACTER, STRING, CHARACTER, STRING, the compression algorithm will output a code before the decompressor gets a chance to define it. A simple example will illustrate the point. Imagine the string JOEYN is defined in the table as code 300. Later on, the sequence JOEYNJOEYNJOEY occurs in the table. The compression output looks like

Input String: ...JOEYNJOEYNJOEY		
Character Input	New Code/String	Code Output
JOEYN	300 = JOEYN	288 (JOEY)
A	301 = NA	N
.	.	.
.	.	.
.	.	.
JOEYNJ	400 = JOEYNJ	300 (JOEYN)
JOEYNJO	401 = JOEYNJO	400 (???)

When the decompression algorithm sees this input stream, it first decodes the code 300, and outputs the JOEYN string. After doing the output, it will add the definition for code 399 to the table, whatever that may be. It then reads the next input code, 400, and finds that it is not in the table. In this case where the decompression algorithm will encounter an undefined code.

3.4 LZW COMPRESSION ALGORITHM

Terry Welch's refinements to the algorithm were published in 1984 [5]. The modified algorithm looks for the special case of an undefined code, and handles it. In the example, the decompression routine sees a code of 400, which is undefined. Since it is undefined, it translates the value of OLD_CODE, which is code 300. It then adds the CHARACTER value, which is 'J', to the string. This results in the correct translation of code 400 to string "JOEYNJ".

MODIFICATION

ROUTINE LZW_DECOMPRESS

Read OLD_CODE

output OLD_CODE

CHARACTER = OLD_CODE

WHILE there are still input characters DO
  Read NEW_CODE

```
IF NEW_CODE is not in the translation table THEN
   STRING = get translation of OLD_CODE
   STRING = STRING+CHARACTER
ELSE
   STRING = get translation of NEW_CODE
END of IF
output STRING
CHARACTER = first character in STRING
add OLD_CODE + CHARACTER to the translation table
OLD_CODE = NEW_CODE
END of WHILE
```

FACTORS AFFECTING THE PERFORMANCE OF LZW

The factors that are crucial for LZW and can be used to alter the compression ratio have been identified as the following:

1) Dynamically restructuring of the number of bits to represent a code word depending upon its magnitude

2) Number of bits to represent a dictionary code word and in turn the maximum size of the dictionary.

3) Static or dynamic nature of the dictionary.

SIZE OF THE DICTIONARY - MAXIMUM

The maximum size of the dictionary plays an important role in the sense that the larger the size the greater number of bits will be required to represent a single character or a code word. That is, if the size of the file, which needs to be compressed were small. Thus the large size of the dictionary will not be optimally used and will end in the increased size of the compressed file. Thus the dictionary size should be based on the size of the file that needs to be compressed.

RESTRUCTURING OF BITS DYNAMICALLY

Dynamic restructuring of the number of bits to represent the code word would enable us to save those extra bits that need not be used by the code word which lies in the range that can be represented using the lesser number of bits.

The standard LZW uses a fixed number of bits to represent each code thus if the size of the dictionary is 4096, then it will use 12-bits ($\log_2 4096$) for each code even if the code is 268(say) which can be represented by 9-bits. Thus dynamic restructuring of the number of bits helps us to save this loss.

DYNAMIC NATURE OF DICTIONARY

Dynamic nature of dictionary can be employed, which will be based on the feedback from the compression algorithm. The feedback parameter will be the current compression ratio relative to the threshold. In the dynamic dictionary the compression ratio will be the continuously matched against a threshold and if the ratio goes beyond what is specified in the threshold, the dictionary is flushed at that moment itself and a new one is created from the remaining inputs.

Thus if dynamic nature is employed the additional factors that need to be analyzed for their effect on the LZW are:

- The threshold value
- The compression check period.

It is not necessary that a higher threshold will result in higher compression always. It all depends on the contents of the file that is to be compressed. At times it might happen that if a very large threshold value is specified, the dictionary would be frequently flushed (since the compression ratio doesn't match) and this frequent flushing may result in a compression ratio which would be smaller than the optimum which could have been achieved using a lower threshold value. Like the threshold value the compression check period also plays an important role. The check period implies the duration of characters after which the current compression ratio is matched against the threshold value.

3.5 LEMPEL-ZIV-WELCH'S ENCODING FOR STRING 'ABBABABC'

Table 3.1

SN	CHAR	STRING+CHAR	IN TABLE	OUTPUT	ADD TO TABLE	NEW STRING	COMMENTS
1 2 3 4 5	A B C R X				1=A 2=B 3=C 4=R 5=X		The first five entries are A,B,C,R,X
6	A	A				A	First character no actions
7	B	AB	NO	A (1)	6=AB	B	
8	B	BB	NO	A (2)	7=BB	B	
9	A	BA	NO	B (2)	8=BA	A	
10	B	AB	YES (6)			AB	First match found
11	A	ABA	NO	AB (6)	9=ABA	A	
12	B	AB	YES (6)			AB	Another match found
13	C	ABC	NO	AB (6)	10=ABC	C	
14	EOF	C		C (3)			End of file

Table 3.1 shows representation of the working of LZW algorithm. According to the conventional approach the number of bits used to represent the code depends on the magnitude of the index of the dictionary i.e. if its follow the conventional approach then it shall be using 4-bit code word to represent the output of the 11th entry (since it depends on the magnitude of the index) but here the output code word has the magnitude 6, which requires only 3-bits to be represented. Thus applying dynamic restructuring based on the magnitude of output the results are more optimized.

In the example by the conventional approach the total number of bits used for output of the entries starting from the 8th entry is 4*5=20. But contrary to this approach uses 3*5=15, thus saving 5 bits.

DYNAMIC FEEDBACK

The standard LZW uses a single dictionary, i.e. the dictionary is only created once, if it gets filled then no more patterns can be formed and hence the only replaceable patterns are the ones which exist in the dictionary. Its means that once the dictionary is full the input read is compared to the patterns within the dictionary if matched they are replaced by the appropriate code word otherwise the characters are outputted as such. This approach followed by the standard LZW at times leads to reduction in the compression ratio.

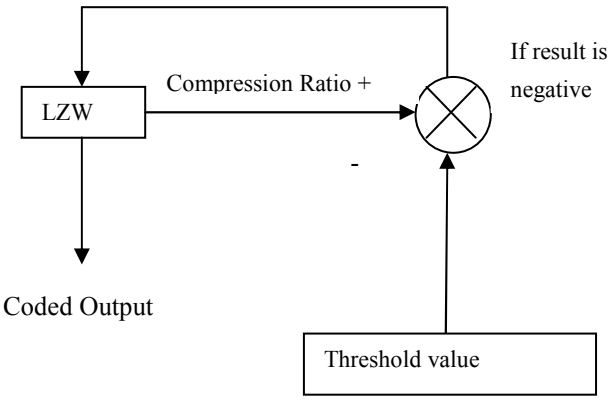

Fig 3.1 Dynamic Feedback

To overcome the above stated drawback we suggest dynamic feedback to the algorithm. The feedback is in the sense that we keep monitoring the compression ratio at suitable intervals, after each such interval the compression ratio is compared against a threshold value, if the compression ratio is below the threshold the dictionary is flushed out and a new one is created which helps us to improve the compression ratio. This process of feedback is schematically depicted in figure 3.1. It is only a schematic representation and cannot be directly viewed as an algorithm to be implemented.

4. FLOWCHARTS

4.1 COMPRESSION

Figure 4.1 shows a flowchart for LZW compression [2]. Table 4.1 provides the step-by-step details for an example input file consisting of 45 bytes, the ASCII text string: the/rain/in/Spain/falls/mainly/on/the/plain. When the LZW algorithm reads the character "a" from the input file, it reads the value: 01100001 (97 expressed in 8 bits), where 97 is "a" in ASCII. When it writes the character "a" to the encoded file, we mean it writes: 000001100001 (97 expressed in 12 bits).

The variable, CHAR, is a single byte. The variable, STRING, is a variable length sequence of bytes. Data are read from the input file (box 1 & 2) as single bytes, and written to the compressed file (box4) as 12 bit codes.

The compression algorithm uses two variables: CHAR and STRING. The variable, CHAR, holds a single character, i.e., a single byte value between 0 and 255. The variable, STRING, is a variable length string, i.e., a group of one or more characters, with each character being a single byte. In box1 of Fig. 4.1, the program starts by taking the first byte from the input file, and placing it in the variable, STRING. This is followed by the algorithm looping for each additional byte in the input file, controlled in the flow diagram by box 8. Each time a byte is read from the input file (box2), it is stored in the variable, CHAR. The data table is then searched to determine if the concatenation of the two variables, STRING+CHAR, has already been assigned a code (box3).

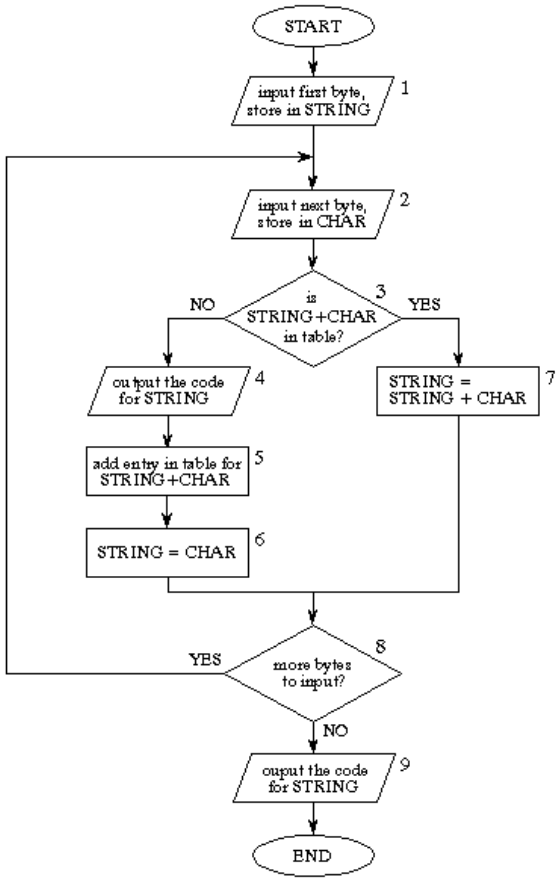

Fig. 4.1 LZW compression flowcharts

If a match in the code table is not found, three actions are taken, as shown in boxes 4, 5 & 6. In box4, the 12-bit code corresponding to the contents of the variable, STRING, is written to the compressed file. In box 5, a new code is created in the table for the concatenation of STRING+CHAR. In box 6, the variable, STRING, takes the value of the variable, CHAR.

When a match in the code table is found (box3), the concatenation of STRING+CHAR is stored in the variable, STRING, without any other action-taking place (box 7). That is, if a matching sequence is found, no action should be taken before determining if there is a longer matching sequence. The sequence: STRING+CHAR= in, is identified as already having a code in the table. The next character from the input file, /, is added to the sequence, and the code table is searched for: in/. Since this longer sequence is not in the table, the program adds it to the table, outputs the code for the shorter sequence that is in the table and starts over searching for sequences beginning with the character, /. This flow of events is continued until there are no more characters in the input file. The program is wrapped up with the code

corresponding to the current value of STRING being written to the compressed file as illustrated in box9 of Fig. 4.1.

4.2 DECOMPRESSION

A flowchart of the LZW decompression algorithm is shown in Fig. 4.2. Each code is read from the compressed file and compared to the code table to provide the translation [2]. The code table is updated so that it continually matches the one used during the compression. However, there is a small complication in the decompression routine. There are certain combinations of data that result in the decompression algorithm receiving a code that does not yet exist in its code table. This contingency is handled in boxes 4, 5 & 6.

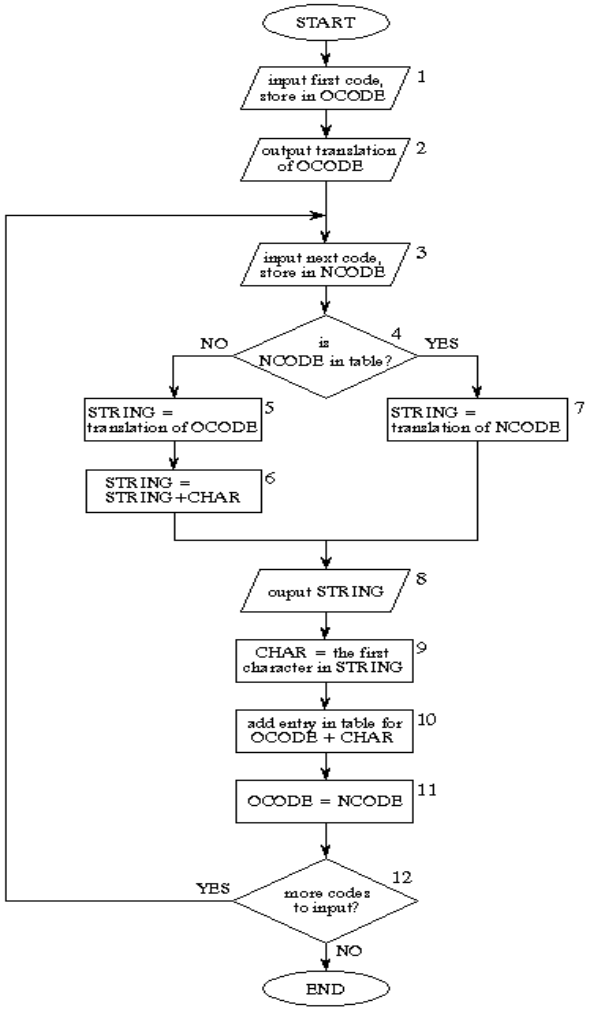

Fig 4.2 LZW Decompression flowchart

The variables, OCODE and NCODE (old code and new code), hold the 12 bit codes from the compressed file, CHAR holds a single byte, and STRING holds a string of bytes.

5. EXPERIMENTAL RESULTS

It is somewhat difficult to characterize the results of any data compression technique. The level of compression achieved varies quite a bit depending on several factors. LZW compression excels when confronted with data streams that have any type of repeated strings. Because of this, it does extremely well when compressing English text. Compression levels of 50% or better should be expected. Likewise, compressing saved screens and displays will generally show very good results.

Trying to compress binary data files is a little more risky. Depending on the data, compression may or may not yield good results. In some cases, data files will compress even more than text. A little bit of experimentation will usually give you a feel for whether your data will compress well or not.

COMPRESSION RESULT - Table 5.1

SN	Files	Starting Size (kb)	Compressed Size (kb)	Compression Ratio (%)
1	AFMD01.txt	118	64	45.7
2	BGNE02.txt	138	77	44.2
3	RMDAQ44.txt	76	44	42.1
4	TREWA22.txt	34	18	47.0
5	Cricket.bmp	729	7	99.0
6	Nero.bmp	668	336	49.7
7	Que.bmp	351	177	49.5
8	Game02.bmp	331	119	64.0
9	Exe1.exe	544	471	13.4
10	Exe2.exe	536	485	09.5
11	Exe3.exe	535	377	29.5
12	Exe4.exe	529	453	14.3
13	Gimme.dll	4493	3739	16.7
14	Mps.dll	3404	3289	03.3
15	Scanres.dll	6264	6256	00.1
16	Comsvcs.dll	1223	1201	01.7

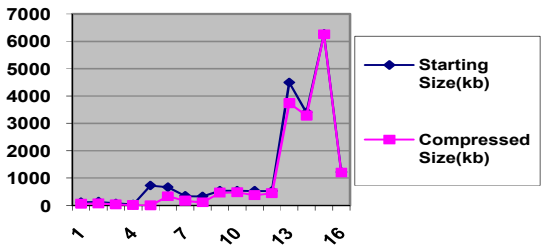

Fig. 5.1 Graph – Compression (Ref Table 5.1)

6. FUTURE SCOPE

We have developed this as a utility, which performs some of the most elementary task like compressing the given file whether it is text file, image file, sound file or executable file and this file can

again decompressed in an original form without any loss. Thus we have developed a comprehensive toolbox, which can be used to perform suitable compression and decompression. Data compression is one such field where this toolbox can prove to be of immense help since it can execute in a few steps. LZW algorithm for data compression being a wide field which is rapidly finding use in many applied fields and technologies, providing tools for some of the most elementary and cumbersome operation which is a major step towards the future technology.

7. CONCLUSION

This paper includes a complete and comprehensive study of all aspects of LZW algorithm. Our study goes from the most basic concepts like what were the limitations of LZ77, LZ78 and LZW algorithm and how modified LZW algorithm overcame those limitations. We have improved the basic LZW algorithm by increasing the size of dictionary. As it possible to store more and longer phrases to compress better. Moving to larger code size actually retards the compression when the file size to be compressed is small. Since phrases are initially found and added to the dictionary at the same pace, whether the code is nine bits or fifteen bits long, the nine bit actually produce a smaller file. Another enhancement is the flush code. This tells the decompressor to throw away all phrases currently in the dictionary and to start over a blank state in order to increase the degrading compression ratio. Typically, you can expect LZW to compress text, executable code, and similar data files to about one-half their original size. LZW also performs well when presented with extremely redundant data files, such as tabulated numbers, computer source code, and acquired signals.

8 REFERENCES

[1] Data Compression by Debra A. Lelewer and Daniel S. Hirschberg, http://www.ics.uci.edu

[2] Steven W. Smith, The Scientist and Engineer's Guide to Digital Signal Processing, http://www.dspguide.com

[3] [1.0] Introduction / Lossless Data Compression. (v1.1.1 / chapter 1 of 3 / 01 apr 05 / greg goebel / public domain), http://www.vectorsite.net

[4] Mark Nelson, Interactive Data Compression Tutor & The data compression book - -2nd Ed. by M&T books, http:// www.eee.bham. ac.uk

[5] LZW Data Compression by Mark Nelson, Dr. Dobb's Journal October, 1989.

[6] D. A. Huffman, ``A Method for the Construction of Minimum Redundancy Codes," Proceedings of the IRE, Vol. 40, pp. 1098--1101, 1952

[7] Jeffrey N. Ladino,"Data Compression Algorithms", http://www.faqs.org/faqs/compression-faq/part2/section-1.html

[8] Article-Compressing and Decompressing Data using Java by Qusay H. Mahmoud with contributions from Konstantin Kladko February 2002

[9] The Data Compression Book, 2nd edition by Mark Nelson and Jean-loup Gailly, M&T Books, New York, NY 1995 ,ISBN 1-55851-434-1.

[10] T. A. Welch, ``A Technique for High-Performance Data Compression," Computer, pp. 8--18, 1984.

[11] J. Ziv and A. Lempel, ``Compression of Individual Sequences Via Variable-Rate Coding," IEEE Transactions on Information Theory, Vol. 24, pp. 530--536, 1978.

[12] J. Ziv and A. Lempel, ``A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, Vol. 23, pp. 337--342, 1977.

[13] K. Sayood, "Introduction to Data Compression".

[14] G. Held and T. R. Marshall, "Data and Image Compression: Tools and Techniques".

[15] D. Hankerson, P. D. Johnson, and G. A. Harris, "Introduction to Information Theory and Data Compression".

[16] Storer, James A., Data Compression: Methods and Theory, Computer Science Press, Rockville, MD, 1988

[17] Soumit Chowdhury, Amit Chowdhury, S. R. Bhadra Chaudhuri, C.T. Bhunia "Data Transmission using Online Dynami Dictionary Based Compression Technique of Fixed and Variable Length Coding" published at International Conference on Computer Science and Information Technology, 2008.

[18] P.G.Howard and J.C.Vitter, "Arithmetic Coding for Data Compression," Proceedings of the IEEE, vol. 82, no.6, 1994, pp.857-865