Journal of Theoretical and Applied Information Technology
10th March 2014. Vol. 61 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A MODIFIED LEMPEL–ZIV WELCH SOURCE CODING ALGORITHM FOR EFFICIENT DATA COMPRESSION

[1] MAHMOUD A. SMADI, [2] QASEM ABU AL-HAIJA
[1, 2] Electrical Engineering Department, King Faisal University,
Saudi Arabia, Alhasa, 31982, P. O. Box 380
E-mail: [1] msmadi@kfu.edu.sa, [2] qalhaija@kfu.edu.sa

ABSTRACT

Lempel–Ziv Welch (LZW) algorithm is a well-known powerful data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. The algorithm is designed to be fast to implement but is not usually optimal because it performs only limited analysis of the data. A modified LZW algorithm on source coding will be proposed in this paper to improve the compression efficiency of the existing algorithms. Such method is to be implemented with appropriate modifications that gives the best performance and satisfies the requirements of the applications.

Keywords: *LZW algorithm, substitution compression, encoding/decoding process.*

1. INTRODUCTION

Data compression seeks to reduce number of bits used to store or transmit information. It encompasses a wide variety of software and hardware compression techniques [1], which can be so unlike one another except that they compress data. These techniques for data compression can ease a variety of problems in storing and transmitting large amount of data [2].

Because of the tremendous amount of digital data used today in several applications in digital data systems, i.e. video-on-demand, word processing programs, digital signal processing etc., the channel bandwidth and the disk drive that looked gigantic become inadequate for most applications. To provide transmission or storage facilities for the data, we need an additional communication lines for transmission or disk drives for storage. In addition to these solutions, the auxiliary devices such as modems, multiplexers … etc. have been continuously upgraded to permit higher data transfer capability [3].

The above ordinary solutions used in transmitting or storing large amount of data require an additional increase in organization equipment and operating costs. One method that can be employed to improve a portion of data storage and information transfer problems is to seek about sophisticated algorithms to search data for redundancy [2]. This redundancy can be removed from the original data by compression algorithms; consequently, the resulting data is smaller than the original data (reduction the quantity of data or information). After that, the compressed data enters to the transmission medium and then it is expanded to its original format at a sink location or we store it on disk drives and then it is expanded when it is needed.

Different compression performance will be obtained by applying the context in different way. For example, a compression method using fixed dictionary has a high speed, but the compression ratio is worse than that of methods using dynamic dictionary. Hence, we search in this paper to propose a variant matching procedure to improve compression performance based on LZW algorithm.

2. DATA COMPRESSION TECHNIQUES

There are many classifications for data compression schemes. These classifications depend on the way in which the techniques treat the text to be compressed. A text is constructed as a group of characters, which are arranged in a random sequence. The probability of occurrence of a character in a text is not the same for all characters. For example, in a typical English language the probability of occurrence of the space and vowel characters (e, o, a, i, u) is much higher than that of other characters such as z or the question mark character. By utilizing this fact, we can assign short codes for most frequently occurring set of characters, while long codes are assigned for seldom occurring set of characters. This method of data compression depends on the frequency of

Journal of Theoretical and Applied Information Technology
10th March 2014. Vol. 61 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

occurrence of the individual character in a text. Therefore, it is classified as statistical data compression. Several techniques are considered as statistical data compression techniques such as Shanno-Fano code, Huffman coding… etc.

Other methods achieve compression by replacing groups of consecutive characters, or "phrase" with shorter representations. These representations may be indices to an actual dictionary (Lempel-Ziv 78) [4] or pointers to previous occurrence groups (Lempel-Ziv 77) [5]. Therefore, it is classified as a dictionary compression technique. Several techniques are considered as dictionary compression method such as LZ77, LZ78, LZW, LZSS, etc. [3]. The idea of developing efficient compression techniques was starting to be fleshed out in the late 1940s (i.e., the early years of information theory). Researchers were exploring the idea of information theory, information contents, and redundancy. Redundant information in a message takes extra bits to encode, and if we can get rid of that extra information, we will reduce the size of the message. After that, the idea of developing algorithms for data compression was really a great leap forward.

The first will-known method for effectively coding symbols is known as Shannon-Fano coding by Claude Shannon at Bell Labs and R. M. Fano at MIT [1]. It depends on knowing the probability of each symbol in a message. While Shannon-Fano coding was a great leap forward, it had the unfortunate luck to be quickly superseded by a more efficient coding system: Huffman coding, which is published in 1952. Huffman original work spawned numerous minor variations [1], and it dominated the coding world until the early 1980s.

Until 1980, most general-compression schemes used statistical modeling. Nevertheless, in 1967 a paper was published describing a semi-adaptive dictionary coding technique and it is closed with the remark that better compression could be obtained by "replacing a repeated string by a reference to an earlier occurrence." [6-8]. This idea was not pursued until 1977, when Jacob-Ziv and Abraham-Lempel described an adaptive dictionary encoder in which they employ the concept of encoding future segments of the input via maximum-length copying from a buffer containing the recent past output in the publication "A universal algorithm for sequential data compression" in IEEE transaction on Information Theory [5]. This paper, with its 1978 sequel in [4] triggered a flood of dictionary-based compression researches, algorithms, and programs.

Dig beneath the surface of any dictionary based compression program, you will find the work of Jacob Ziv and Abraham Lempel. For all practical purposes, these two researchers gave birth to this branch of information theory in the late 1970s. Despite that, the publication of those two papers entered the world of information theory in 1977 and 1978 respectively; it was some time before programmers noticed their effects. The June 1984 issue of IEEE computer had an article entitled "A Technique for High Performance Data Compression" by Terry Welch [9]. His paper was a practical description of LZ78 algorithm implementation, which he called LZW. He discussed the LZW compression algorithm and explained that it is possible to use it in disk and tape-drive controllers (i.e., hardware implementation). All researchers seek to improve the compression performance (i.e., memory requirement, encoding-decoding speed, compression ratio, etc.). Therefore, earlier schemes in the literature tended to use small amount of memory and CPU time, but recently both of these become cheaper, and later schemes have concentrated on achieving the best possible compression.

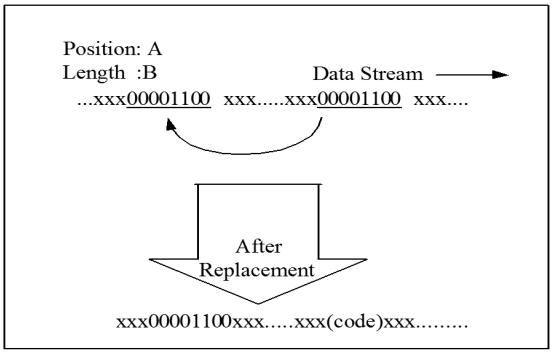

Fig. 1: The Idea Of The Substitution Compression By Using A Code To Old Information In Position A With Length B.

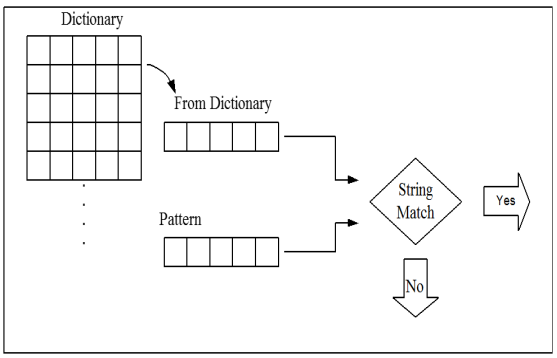

Fig. 2: Core Of Substitution Compression.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The basic idea behind a dictionary compression technique is to replace an occurrence of string in the data stream by a pointer to a previous occurrence of the phrase. Since the pointer may have a shortest length than the original string, this replacement leads to more compact representation of the data. The proposed compression algorithm will be achieved as shown in Fig. 1. The first task in coding a stream by substitution compression algorithm is to find the string that is repeating. A matching procedure is used to locate repeating strings in the stream. The procedure compares the new incoming strings with previously occurring strings store in a buffer called dictionary, see Fig. 2. Compression performance is determined by the matching procedure. Therefore, a good choice of matching method can affect both compression speed and compression ratio. A main part of matching procedure is the dictionary, which stores the previous incoming stream that can record the source characteristics or the context of the source.

3. PROPOSED COMPRESSION ALGORITHM

The scenario described in Welch's 1984 paper [9] encodes sequences of 8-bit data as fixed-length 12-bit codes. The codes from 0 to 255 represent 1-character sequences consisting of the corresponding 8-bit character, and the codes 256 through 4095 are created in a dictionary for sequences encountered in the data as it is encoded. At each stage in compression, input bytes are gathered into a sequence until the next character would make a sequence for which there is no code yet in the dictionary. The code for the sequence (without that character) is emitted, and a new code (for the sequence with that character) is added to the dictionary [10].

After importing the data as a text file the next step is how to convert it to binary. To convert to binary first characters musts convert to its ASCII code values and then to binary. The ASCII values range are 0→127 so it must be converted to 7-bit (b_6-b_0) binary. A simple way to convert by subtracting 64 from the ASCII value and if the result is positive that means the MSB (Most Significant Bit: b_6) = 1, else b_6 = 0 and keep the value without subtraction and then subtract 32 and retest the value if positive. The above procedure will be applied on all bits until subtracting 1 from ASCII value and find the last bit.

3.1 Encoding Process

At the beginning a specific length of the binary code must be determined. We could just use a fixed dictionary like 4096 location (12-bit code and 1 for prefix) but in this case some problems will occur:

a) When a short length characters are input, the codes will be so much great comparing to their number so the size of text will dramatically boost up.

b) Sometimes when the input is very long the dictionary may not be enough.

To avoid these problems we propose an idea which is to assign a variable dictionary size depending on the input length. To determine the appropriate dictionary size we consider the worst case that could happen if the all possible combination are exist in the code like: 0**1;** 00 **01** 10 **11;** 000 **001** 010 **011** 100 **101** 110 **111**

The algorithm will go as follow:

- If input is 2 bits length it will give us a maximum number of 2 symbols
- If input is 2 + 8 = 10 bits length it will give a maximum number of 2 + 4 = 6 symbols
- If input is 2 + 8 + 24 = 36 bits length it will give a maximum number of 2+ 4 + 8 = 14 symbols
- In general if input is 1.2 + 2.4 + 3.8 + … bits length it will give a maximum number of 2 + 4 + 8 + … symbols

Mathematically, if the input is $\sum_{k=1}^{n} k\,2^k$ bits length (where n is maximum symbol length) we will assign a maximum number of $\sum_{k=1}^{n} 2^k$ symbols. But $\sum_{k=1}^{n} 2^k = 2^{n+1} - 2$, and 2^{n+1} symbols can be represented using $n+1$ bits, so it will be true to assume that for an input length of $\sum_{k=1}^{n} k\,2^k$ bits, $n+1$ bits or less are needed to represent the possible contained symbols. Table 1 below provides an example on that.

Following that, the compressed binary data has to be converted into symbols. This operation is needed because the PC will deal with 0 and 1 as a bytes not a bits so the size of the file is zipped from 7 times the original size but it is still very much big. So in converting to symbols the size is going to be divided by 7.

In the process of converting to symbols each 7 bits are returned to an integer value (0→127) and then take the character that meet that value in ASCII. But what will happen if the integer is zero? The algorithm will return a NULL and it does not exist so in decoding it won't be seen and a loss in data will happen. The solution is to convert each 6 bits to an integer and add an offset to get rid from

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

the zero value and then return a symbol as follow: 6 bits (0→63) + 32 = (32→95) so that all characters are included. But in this case the bits of binary must be in multiple of 6 so the algorithm adds bits in case that the zipped binary is not a multiple of 6. Then the new text will be converted and indication bytes will be added for the decoding. We can simply use one byte to indicate the number of zero's that been added and two bits to indicate the length of each code.

Table 1 : Variable Dictionary Size Example

Symbol length	Input binary code length bits	Maximum number of symbols	Number of bits for code	Number of bits for code & prefix
1	2	2	1	2
2	10	6	3	4
3	34	14	4	5
4	98	30	5	6
5	258	62	6	7
6	642	126	7	8
7	1538	254	8	9
8	3586	510	9	10
9	8194	1022	10	11
10	18434	2046	11	12
11	40962	4094	12	13
12	90114	8190	13	14
13	196610	16382	14	15
14	425986	32766	15	16

Therefore, successively longer strings are registered in the dictionary and made available for subsequent encoding as single output values. One can also note that the algorithm will perform better on data with repeated patterns, so the initial parts of a message will see little compression. As the message grows, however, the compression ratio tends asymptotically to the maximum as expected.

3.2 Decoding Process

The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the initialized dictionary. At the same time, it obtains the next value from the input, and adds to the dictionary the concatenation of the string just output and the first character of the string obtained by decoding the next input value. 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.

Note that a reverse operations of the encoding process have to be performed. So, after importing the encoded data it will be converted to binary and unzipped. This unzipping operation is much easier than zipping and it doesn't return values as string but it just find the symbols and save them in array and they're printed after that in the button command. Finally, we convert the unzipped binary into symbols and output them as text file.

In this way, the decoder builds up an identical dictionary to that one used by the encoder, and uses it to decode 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 earlier within the encoder and decoder rather than being explicitly sent with the encoded data.)

4. ILLUSTRATION EXAMPLE

The following example illustrates the proposed modified LZW algorithm in action, showing the status of the output and the dictionary at every stage, both in encoding and decoding the data. This example has been constructed to give reasonable compression on a very short message. In real text data, repetition is generally less pronounced, so longer input streams are typically necessary before the compression builds up efficiency. The massage to be encoded (from an alphabet using only the capital letters) is:

"ABBCBCABABCAABCAAB"

An 8-bit codes are needed to give sufficient combinations to encompass this set of 18 values. The dictionary is initialized with these 18 values. As the dictionary grows, the codes will need to grow in width to accommodate the additional entries. A 8-bit code gives 64 possible combinations of bits, so when the 65th dictionary word is created, the algorithm will have to switch at that point from 8-bit strings to 9-bit strings (for all code values, including those which were previously output with only eight bits). Previously generated output is not affected by the code-width change, but once a 9-bit value is generated in the dictionary, it could conceivably be the next code emitted, so the width for subsequent output shifts to 9 bits to accommodate that.

4.1 Encoding

Buffer input characters in a sequence ω until ω + next character is not in the dictionary. Emit the code for ω, and add ω + next character to the dictionary. Start buffering again with the next character. The

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

encoding process based on the proposed algorithm is illustrated in the table 2 below.

Table 2 : Encoding Process Example

	Dictionary	
output	index	string
(0, A)	1	A
(0, B)	2	B
(2, C)	3	BC
(3, A)	4	BCA
(2, A)	5	BA
(4, A)	6	BCAA
(6, B)	7	BCAAB

As we can see, each codeword consists of an integer and a character, i.e., (0, A): 1, (0, B): 2, (2, C): 3, (3, A): 4, (2, A): 5, (4, A): 6, and (6, B): 7. Now each character will be represented by 8 bits and the number of bits *n* required to represent the integer part of the codeword with index *i* will be

$$n = \begin{cases} 1, & i = 1 \\ \log_2 i, & i > 1 \end{cases}$$

Therefore, the number of significant bits to represent each integer is given in the table 3.

Table 3 : Number Of Significant Bits

index	Index-1	bits	number of significant bits
1	0	0	1
2	1	1	
3	2	10	2
4	3	11	
5	4	100	3
6	5	101	
7	6	110	
8	7	111	
9	8	1000	4
10	9	1001	
11	10	1010	
12	11	1011	
13	12	1100	
14	13	1101	
15	14	1110	
16	15	1111	

Note that the actual encoded message is: "0A0B10C11A010A100A110B" where each character is replaced by its binary 8-bit ASCII code. To calculate the compression efficiency of the proposed algorithm we note that:

- Unencoded length = 18 symbols *8 bits/symbol = 144 bits
- Encoded length = (1+8)+(1+8)+ (2+8)+(2+8)+ (3+8)+(3+8) +(3+8) = 71 bits.

Hence, the proposed modified LZW algorithm has saved 73 bits out of 144, reducing the message by almost 50%. If the message were longer, then the dictionary words would begin to represent longer and longer sections of text, allowing repeated words to be sent very compactly.

4.2 Decoding

To decode a compressed message, one needs to know in advance the initial dictionary used, but additional entries can be reconstructed as they are always simply concatenations of previous entries. At each stage, the decoder receives a code χ; it looks χ up in the table and outputs the sequence χ it codes, and it conjectures χ +? as the entry the encoder just added. Because the encoder emitted χ for χ precisely since χ +? was not in the table, and the encoder goes ahead and adds it. However, what is the missing letter? It is the first letter in the sequence coded by the next code Z that the decoder receives. Therefore, the decoder looks up Z, decodes it into the sequence ω, takes the first letter z, and tacks it onto the end of χ as the next dictionary entry.

For our example, the decoded (or decompressed) message for the sequence (0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B) will be readily obtained as given in the below table

Table 4 : Decoding Process Example

	Dictionary	
output	index	string
A	1	A
B	2	B
BC	3	BC
BCA	4	BCA
BA	5	BA
BCAA	6	BCAA
BCAAB	7	BCAAB

Hence, the decompressed message is: "**ABBCBCABABCAABCAAB**" which is the original text message. This works as long as the codes received are in the decoder's dictionary, so that they can be decoded into sequences. What happens if the decoder receives a code Z that is not yet in its dictionary? Since the decoder is always just one code behind the encoder, Z can be in the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

encoder's dictionary only if the encoder just generated it, when emitting the previous code χ for χ. Thus Z codes some ω that is χ +? And the decoder can determine the unknown character as follows [10]:

- The decoder sees χ and then Z.
- It knows χ codes the sequence χ and Z codes some unknown sequence ω.
- It knows the encoder just added Z to code χ + some unknown character,
- And it knows that the unknown character is the first letter z of ω.
- But the first letter of ω (= χ +?) must then also be the first letter of χ.
- Therefore, ω must be χ + x, where x is the first letter of χ.
- So the decoder figures out what Z codes even though it's not in the table,
- In addition, upon receiving Z, the decoder decodes it as χ + x, and adds χ + x to the table as the value of Z.

This situation occurs whenever the encoder encounters input of the form cScSc, where c is a single character, S is a string and cS is already in the dictionary, but cSc is not. The encoder emits the code for cS, putting a new code for cSc into the dictionary. Next, it sees cSc in the input (starting at the second c of cScSc) and emits the new code it just inserted. The argument above shows that whenever the decoder receives a code not in its dictionary, the situation must look like this.

Although input of form cScSc might seem unlikely, this pattern is common when the input stream is characterized by significant repetition. In particular, long strings of a single character (which are common in the kinds of images LZW is often used to encode) repeatedly generate patterns of this sort.

5. CONCLUSION

In this paper we proposed an efficient matching algorithm based on Lempel-Ziv technique that improved the compression process. The performance of the proposed algorithm is studied in order to draw a comparison between its performance and the performance of the exciting compression techniques. We showed that the proposed algorithm has a compression efficiency of 50% achieved even we applied it on a short message text. Finally, the algorithm can be implemented with appropriate modifications that satisfies the requirements of many data compression applications.

ACKNOWLEDGMENT

Authors would like to thank the Deanship of Scientific Research at King Faisal University (KFU), Alhas, Saudi Arabia for supporting this research.

REFERENCES:

[1] M. Nelson, The data compression books. Prentice Hall, 1996.

[2] J. Weiss and D. Shremp, “Putting data on diet,” IEEE spectrum, vol. 30, pp.36-39, Aug. 1993.

[3] G. Held and T. R. Marshell, Data Compression, John Wiley, New York, 1991.

[4] J. Ziv and A. Lempel, “Compression of individual sequence via variable-rate coding,” IEEE Trans. on Inform. Theory, vol. IT-22, pp. 75-81, Jan. 1978.

[5] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” IEEE Trans. on Inform. Theory, vol. IT-23, pp. 337-343, May 1977.

[6] Uyematsu, T.; Kuzuoka, S.,"Conditional Lempel-Ziv complexity and its application to source coding theorem with side information," IEEE International Symposium on Information Theory, vol. 29, no. 4, pp. 142, 2003.

[7] Savari, S.A. ,"Redundancy of the Lempel-Ziv string matching code," IEEE Transactions on Information Theory, vol.44, no.2, pp.787-791, Mar 1998.

[8] Yuriy A. Reznik, Wojciech Szpankowski, "On the Average Redundancy Rate of the Lempel-Ziv Code with K-Error Protocol," dcc, pp.373, Data Compression Conference (DCC '00), 2000.

[9] Welch, T.A., "A Technique for High-Performance Data Compression," IEEE Trans. On Computer, vol. 17, no. 6, pp. 8–19, June 1984.

[10] www. wikipedia.org.