An Improved LZW Algorithm for Large Data Size and Low Bitwidth per Code

Yi-Lin Tsai and Jian-Jiun Ding
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
b06901178@ntu.edu.tw, jjding@ntu.edu.tw

***Abstract*— The Lempel-Ziv-Welch (LZW) algorithm achieves outstanding performance and is widely used in text encoding. However, when the bit-width for each code is limited and the amount of data to be encoded is huge, the dictionary will be prematurely full. As a result, the new string cannot be added to the dictionary in the later encoding process and thus the performance is compromised. In this paper, an improved LZW algorithm is proposed to address this problem. In the proposed algorithm, the string is added to the dictionary only if the frequency of the string reaches the threshold. In this way, the dictionary is left only for the common strings. The experiments on the test patterns consisting of 300,000 characters show that the proposed algorithm further enhances the compression rates and is efficient for data compression. For example, the compression rate of the proposed algorithm is 6.0% larger than that of the LZW algorithm when each codeword contains 12 bits.**

Keywords*— Data compression, LZW algorithm, dictionary coding, lossless data compression, text encoding.

I. INTRODUCTION

Data compression is a process to lower the memory requirement. It can be accomplished by removing the redundant information of the data. Data compression is either lossless or lossy. Lossless compression refers to the algorithms that can perfectly reconstruct the original data from the encoded data. Lossless compression is usually applied when any difference between the original data and the decoded data is unacceptable. Contrarily, the lossy compression may lose some information of the original data but generally achieves better compression efficiency than lossless compression does. In this paper, we focus on text compression. Since any character in the text cannot be deleted or replaced after encoding, text compression is a lossless compression problem.

Several algorithms were proposed in decades for lossless data compression. These algorithms can be divided into two categories: entropy coding and dictionary coding. Entropy coding includes Shannon-Fano coding [1-2], Huffman coding [3], static arithmetic coding (SAC) [4-7], and adaptive arithmetic coding (AAC) [8-10]. Dictionary coding can be divided into two families: the Lempel-Ziv 77 (LZ77) family and the LZ78 family. The LZ77 family includes the LZ77 algorithm [11] and the Lempel–Ziv–Storer–Szymanski (LZSS) algorithm [12]. On the other hand, the LZ78 family includes the LZ78 algorithm [13] and the Lempel-Ziv-Welch (LZW) algorithm [14].

The remaining part of this paper is organized as follows: In section II, we make a brief review on LZW, some other variations of the LZ algorithm, and entropy coding algorithms. In section III we explain the encoding and decoding part of the proposed algorithm and give an example. The experimental results and analysis are provided in section IV. Finally, we wrap up this paper in section V.

II. PRELIMINARY

A. Huffman Coding

The key concept of Huffman coding is that for the optimal code, the code length for the frequent character should be short, while the code length for the infrequent character should be long. In the Huffman coding process, initially, the free nodes to represent each character are constructed. Then, we follow the rule that the probability of the node in the L^{th} layer should be no smaller than that in the $(L+1)^{\text{th}}$ layer to construct the coding tree. We can obtain the code of a character by traversing from the root of the tree to its leaf nodes.

Huffman coding requires the probability of each character in advance, and such information needs to be transmitted to the decoder during the encoding process. In addition, we can observe that Huffman coding separately encodes each character. Hence, when characters are not independent and identically distributed, Huffman coding may be less efficient than arithmetic coding.

B. Arithmetic Coding

Arithmetic coding can be categorized into SAC and AAC. SAC outputs a number m to represent the input, where $0.0 \leq m < 1.0$. Suppose that the set of input characters is

$$A = \{a_1, a_2, .., a_n\}, \tag{1}$$

where n is the number of the kinds of input characters. Let X be a random variable representing the input character. We define that the probability density function of the input character being a_i is $P(X = a_i)$ and the cumulative density function of the input character being a_i is

$$F(X = a_i) = \sum_{k=1}^{i} P(X = a_k). \tag{2}$$

The range $[L, H]$ is initialized with $[0, 1]$, where L denotes the lower bound and H denotes the upper bound. If the current value of X is a_i, the lower bound and the upper bound is updated with the formulas

$$L_{new} = L + (H - L)F(X = a_{i-1}), \tag{3}$$

$$H_{new} = L + (H - L)F(X = a_i). \tag{4}$$

The lower bound and the upper bound are iteratively updated until the input is exhausted. Lastly, a code is generated to represent the range [L, H]. The disadvantage of SAC is that it requires prior knowledge about the probability of the input data, which is usually hard to acquire. Even if the probability can be known in advance, extra bits are required to encode the probability distribution. As a result, AAC is developed to address this problem.

In AAC, the probability of each character is derived from the frequency table. The value of a_i in the frequency table is denoted as $T[a_i]$. First, the value of every possible character of the input in the frequency table is initialized with 1. Suppose that the encoder is encountering the input character a_i, the probability of a_i is

$$P(X = a_i) = T[a_i\,] / \sum_{k=1}^{n} T[a_k\,]. \quad (5)$$

After a_i is encoded, $T[a_i]$ is incremented by 1. We can see that AAC does not need prior knowledge about the probability of the input data. Moreover, if the length of the input is large enough, the probability obtained from the frequency table will approach the true probability distribution.

C. The LZ77 Family

LZ77 consists of two windows, the *sliding window*, and the *look-ahead window*. The sliding window contains the recently encoded characters, which is the dictionary in LZ77. The look-ahead window contains the characters to be encoded. The algorithm finds the longest sequence in the sliding window that matches the sequence in the look-ahead window starting from the first character. Then, the triple <d, l, c> is generated where d denotes the distance of the matched sequence from the look-ahead window, l denotes the length of the matched sequence, and c is the character that follows the matched sequence in the look-ahead window. The reason to output c is that there may be no match between the sliding window and the look-ahead window. In this case, c is the first character in the look-ahead window and both the distance and the matching length are zero.

LZSS further improves LZ77 by removing the third element c. Instead, LZSS includes a 1-bit flag to indicate whether the output is a triple or a single character. To be more specific, if the matching case occurs, the encoder uses a flag bit to inform the decoder and outputs the distance and the matching length. Otherwise, the encoder specifies a non-matching case and outputs the first character in the look-ahead window.

D. The LZW Algorithm

LZW is an even more efficient text encoding algorithm compared to LZ77 and LZSS. LZW only outputs the index of the dictionary. The steps of LZW are as follows. First, the dictionary is initialized to contain every possible character in the input. After that, the algorithm scans through the input strings. The encoder has some prefix string w in the dictionary, and K is the character to be scanned. If wK is in the dictionary, the prefix for scanning the next character is wK. Otherwise, if wK is not in the dictionary, wK is added to the dictionary, and the index of w in the dictionary is outputted.

E. The Challenge of LZW and the Proposed Algorithm

The dictionary of LZW has the upper bound. That is, the size of the dictionary in LZW cannot exceed two to the power of bits per code. As a result, the challenge for LZW is that if the number of bits per code is limited and the input size is large, the dictionary will be prematurely full. Therefore, no new strings can be added to the dictionary, which compromises the compression efficiency. The goal of this paper is to enhance the efficiency of dictionary storage to improve the compression rates when the number of bits per code is small. To do this, we count the frequency of every string in the dictionary. Furthermore, we add the string to the dictionary only when the number of times that the string is looked up in the dictionary reaches the threshold. In this way, the dictionary is only assigned for frequent strings and the compression rate can be further improved.

III. Proposed Algorithm

A. Overview

The proposed algorithm contains two dictionaries: the *improved dictionary* and the *original dictionary*. We add the string to the original dictionary in the same way as LZW does. On the other hand, we add the string to the improved dictionary only when the number of times that the string is looked up in the original dictionary reaches the threshold. Additionally, we only output the index of the string in the improved dictionary. With it, we can make sure that only the strings that appear frequently are used for encoding. In the following subsections, the details of the encoding and decoding processes of the proposed algorithm are described.

B. Encoding

The encoder first initializes the improved dictionary and the original dictionary with every possible character in the input. The input is accumulated in the prefix string w_1 if w_1 is in the improved dictionary. Similarly, the input is accumulated in another prefix string w_2 if w_2 is in the original dictionary. Then, the encoder scans the input character K from the input. If the combination of w_1 and K (i.e., the string w_1K) is in the improved dictionary, then the prefix for scanning the next input character is w_1K. Otherwise, if w_1K is not in the improved dictionary, the index of w_1 in the improved dictionary will be outputted, and the prefix for scanning the next input character is K. After handling the improved dictionary, the encoder starts to deal with the original dictionary. If the combination of K and the prefix w_2 (i.e., the string w_2K) is in the original dictionary, the frequency of w_2K increases by 1. If the frequency of w_2K reaches the threshold, w_2K is added to the improved dictionary. The prefix for scanning the next input character is w_2K. Otherwise, if w_2K is not in the original dictionary, w_2K is added to the original dictionary, and its frequency is initialized with 1. The prefix w_2 for the scan of the next input character is K. The above step is repeated until all the input characters are processed.

Algorithm 1: Encoding Process

Initialize the improved dictionary and the original dictionary to contain every possible character.
w_1 = NIL;
w_2 = NIL;
while (there is input)
  K = input character;
  I = the index of K in the input;
  if (w_1K is in the improved dictionary)
    $w_1 = w_1K$;
  else
    if (adding index of w_1 == index of any character in w_1 in the input)
      output the index of s and the index of C in the improved dictionary;
    else
      output the index of w_1 in the improved dictionary;
    end
    $w_1 = K$;
  end
  if (w_2K is in the original dictionary)
    the frequency of w_2K += 1;
    if (the frequency of w_2K == threshold)
      add w_2 to the improved dictionary;
      the adding index of $w_2 = I$;
    end
    $w_2 = w_2K$;
  else
    add w_2K to the original dictionary;
    set the frequency of w_2K to 1;
    $w_2 = K$;
  end
end

However, the algorithm should be modified a little. Suppose that w_1 is added to the improved dictionary when the index = n (i.e., when processing the n^{th} character) and length(w_1) = L. Assume that we are processing $(m+1)^{th}$ character K, where $n < m$, and that the current prefix is w_1. Suppose that w_1K is not in the improved dictionary, so we are going to output the index of w_1 in the improved dictionary. Nevertheless, if $m-L+1 \le n$, and we output the index of w_1 in the improved dictionary, the error may occur since the string w_1 is not added to the improved dictionary yet after decoding the first $m-L$ characters. To address this problem, whenever a string is added to the improved dictionary, the adding index, which is the index of the input character, must be added to the improved dictionary with the added string. When the encoder outputs the index of some string w_1, it also has to check when w_1 was added to the improved dictionary. If w_1 was added to the improved dictionary when any character M in w_1 was being processed, the encoder needs to split w_1 into s and a single character C, where $w_1 = sC$. That is, s is the prefix of w_1 and C is the remaining character. The encoder will output the indices of s and C in the improved dictionary instead of the index of w_1. In this way, the decoder can successfully obtain the correct string w_1. The encoding algorithm with error handling is described as in Algorithm 1.

Algorithm 2: Decoding Process

Initialize the improved dictionary and the original dictionary to contain every possible character.
w_2 = NIL;
decoded string = NIL;
while (there is code)
  $CODE$ = code from input;
  find the string s whose index in the improved dictionary is $CODE$;
  append s to decoded string;
  for K = the first to the last character of s
    if w_2K is in the original dictionary
      the frequency of w_2K += 1;
      if (the frequency of w_2K == threshold)
        add w_2 to the improved dictionary;
      end
      $w_2 = w_2K$;
    else
      add w_2K to the original dictionary;
      the frequency of w_2K = 1;
      $w_2 = K$;
    end
  end
end

C. Decoding

The decoder also first initializes the improved dictionary and the original dictionary with all possible characters. Moreover, the prefix w_2 and the decoded string are initialized with NIL. After that, the decoder fetches the first code $CODE$. String s is appended to the decoded string, where the index of s in the improved dictionary is $CODE$. The decoder extracts the character K from s from the first character to the last one. If w_2K is in the original dictionary, the frequency of w_2K is increased by 1. If the frequency of w_2K reaches the threshold, w_2K is added to the improved dictionary. The prefix w_2 for the next extracted character from s is w_2K. Otherwise, if w_2K is not in the original dictionary, w_2K is added to the original dictionary, and the frequency of w_2K is initialized with 1. The prefix w_2 for the next extracted character from s is K. The above process is repeated until all codes are decoded. The decoding algorithm is summarized in Algorithm 2.

D. An Encoding Example of the Proposed Algorithm

In Fig. 1, Table I, and Table II, an encoding example of the proposed algorithm is shown. In this example, the threshold is set to two. Since the encoder adds the new strings to the original dictionary in the same way as LZW does, we will focus on the improved dictionary.

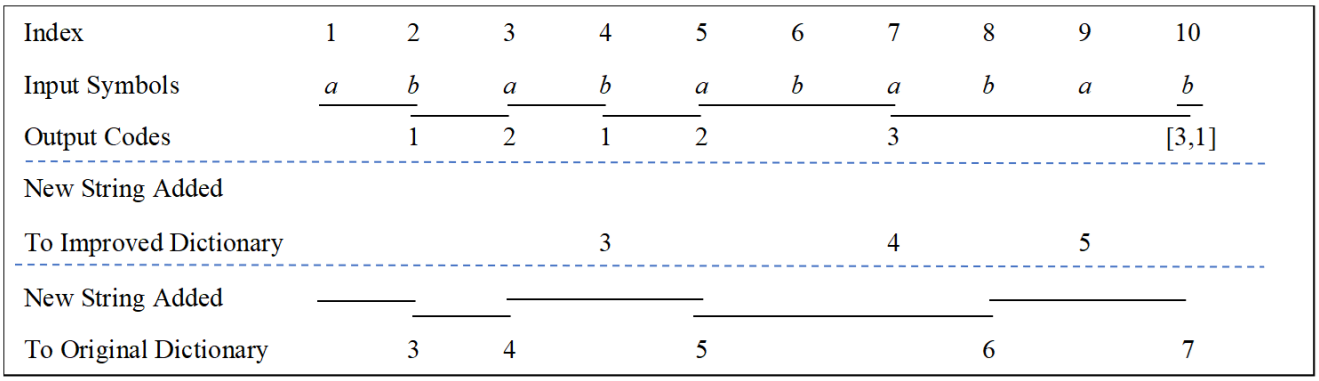

Figure 1. An encoding example where the threshold is set to 2.

TABLE I. THE IMPROVED DICTIONARY IN THE EXAMPLE

String Number	String	When the String is Added
1	*a*	Initialization
2	*b*	Initialization
3	*ab*	Index = 4
4	*aba*	Index = 7
5	*ba*	Index = 9

At index 1, the encoder finds *a* in the improved dictionary. Thus, w_1 is *a*. At index 2 *ab* is not in the improved dictionary. Therefore, the encoder outputs the index of *a* in the improved dictionary, and w_1 is changed to *b*. Note that at index 4, first, since the encoder cannot find *ab* in the improved dictionary, the index of *a* in the improved dictionary is outputted, and w_1 is set to *b*. Then, the encoder looks up *ab* in the original dictionary. Since at this moment the frequency of *ab* is 2 and reaches the threshold, *ab* is added to the improved dictionary. Moreover, the prefix w_2 is changed from *b* to *ab*. However, to make the sequence able to be decoded from the causal part, the output at index 4 remains to be the index of *a* in the improved dictionary. The string *aba* is added to the original dictionary at index 5. It reaches the frequency of 2 and is added to the improved dictionary at index 7. The string *ba* reaches the frequency of 2 and is added to the improved dictionary at index 9.

Note that at index 10 the encoder deals with error handling. The encoder would have outputted the index of *aba* in the improved dictionary if it were a normal case. However, *aba* was added to the improved dictionary at index 7, and the first character of *aba* is also at index 7. It means that for the decoder, after decoding the first 6 characters, the string *aba* is not in the improved dictionary yet. Therefore, instead of the index of *aba*, the encoder outputs the indexes of *ab* and *a* in the improved dictionary. Table I shows the improved dictionary in this example, while Table II shows the original dictionary.

IV. PERFORMANCE EVALUATION

TABLE II. THE ORIGINAL DICTIONARY OF THE EXAMPLE

String Number	String	Frequency
1	*a*	NIL
2	*b*	NIL
3	*ab*	3
4	*ba*	2
5	*aba*	2
6	*abab*	1
7	*bab*	1

In Table III, we perform the proposed algorithm and the LZW algorithm for text compression with different values of the bits per code that range from 8 to 14 bits and with six files. Every file contains 300,000 characters. Note that the value of the bits per code is different from that of bits per character. If bits per code = k, then the dictionary size = 2^k, and the dictionary size in the proposed algorithm refers to the dictionary size of the improved dictionary. The ASCII values of the characters range from 0 to 127. The threshold for the proposed algorithm is two. We provide a comparative analysis of how the compression rates are improved with various values of the bits per code. The improvement of the compression rates of the proposed algorithm ranges from 0.85% to 19.63%. Furthermore, it is noteworthy that the compression rate significantly increases when the value of bits per code is small. To be more specific, the average improvement is 16.24% when the value of bits per code is 8. Accordingly, we can conclude that the proposed algorithm improves the compression efficiency when the value of bits per code is limited.

In Table IV, we compare the performance of the proposed algorithm and other compression algorithms. The algorithms compared with the proposed algorithm include entropy-based algorithms and dictionary-based algorithms. The entropy-based algorithms include Huffman coding, SAC, and AAC. On the other hand, the dictionary-based algorithms are divided into the LZ77 family and the LZ78 family.

TABLE III. PERFORMANCE COMPARISON OF THE PROPOSED ALGORITHM AND LZW UNDER DIFFERENT BITS PER CODE. (NOTE THAT BITS PER CODE ARE DIFFERENT FROM BITS PER CHARACTER. IF BITS PER CODE = K, THEN THE DICTIONARY SIZE = 2^K).

File Name	Bits per code = 8 (means that the dictionary size = 2^8)		Bits per code = 10		Bits per code = 12		Bits per code = 14	
	Proposed Algorithm	LZW	**Proposed Algorithm**	LZW	**Proposed Algorithm**	LZW	**Proposed Algorithm**	LZW
	Bits per Character	Bits per Character	Bits per Character	Bits per Character	Bits per Character	Bits per Character	Bits per Character	Bits per Character
Freakonomics	5.19	5.96	4.4	4.65	3.90	4.15	3.69	3.75
Harry Potter	5.04	6.10	4.16	4.42	3.71	3.92	3.50	3.56
Rich Dad Poor Dad	4.94	6.13	4.10	4.41	3.64	3.85	3.42	3.51
To Kill A Mockingbird	5.06	6.21	4.22	4.39	3.65	3.86	3.49	3.52
Good to Great	5.28	6.57	4.49	4.77	4.04	4.38	3.74	3.81
Sophie's World	4.95	5.47	4.15	4.45	3.66	3.89	3.45	3.49
Average Bits per Character	5.08	6.07	4.25	4.52	3.77	4.01	3.55	3.61

TABLE IV. PERFORMANCE COMPARISON OF THE PROPOSED AND OTHER ALGORITHMS FOR TEXT COMPRESSION IN TERMS OF BITS PER CHARACTER.

File Names	**Proposed Algorithm**	LZW	LZ77	LZSS	Huffman coding	SAC	AAC
	Bits per Character	Bits per Character	Bits per Character	Bits per Character	Bits per Character	Bits per Character	Bits per Character
Freakonomics	3.90	4.15	4.58	4.42	4.64	4.62	4.59
Harry Potter	3.71	3.92	4.56	4.35	4.61	4.57	4.55
Rich Dad Poor Dad	3.64	3.85	4.24	4.04	4.55	4.51	4.49
To Kill A Mockingbird	3.65	3.86	4.63	4.42	4.59	4.56	4.53
Good to Great	4.04	4.38	4.73	4.61	4.74	4.71	4.69
Sophie's World	3.66	3.89	4.40	4.16	4.53	4.50	4.47
Average Bits per Character	3.77	4.01	4.52	4.33	4.61	4.58	4.55

The LZ77 family includes the LZ77 algorithm and the LZSS algorithm. For the LZ77 algorithm and the LZSS algorithm, the distance is encoded with 12 bits and the matching length is encoded with 4 bits. Finally, the LZ78 family includes the LZW algorithm. The value of bits per code for the proposed algorithm and the LZW algorithm is 12.

From Table IV, we can observe that the proposed algorithm not only outperforms the LZW algorithm but also outperforms entropy-based algorithms and the algorithms in the LZ77 family.

V. CONCLUSION

In this paper, an algorithm to improve the compression rates when the value of bits per code is small is proposed. In the proposed algorithm, a string is added to the improved dictionary only when the number of times that the string is looked up in the original dictionary reaches the threshold. Hence, the improved dictionary only contains the frequent strings for further optimization and the efficiency of compression is thus improved. Experimental results in section IV indicate that the compression rates are considerably improved as the value of bits per code decreases. Specifically, the improvement of compression rates of the average bits per character increases from 1.7% to 16.3% as the value of bits per code decreases from 14 to 8. It confirms that the proposed algorithm is very suitable for the case where the bit-width of a codeword is limited.

ACKNOWLEDGMENT

This work was supported by the Ministry of Science and Technology, Taiwan, under the contract of 108-2221-E-002 -041 -MY3.

REFERENCES

[1] T. Tjalkens, “Implementation cost of the Huffman-Shannon-Fano code,” *IEEE Data Compression Conference*, pp. 123-132, 2005.

[2] M. Vaidya, E. S. Walia, and A. Gupta. "Data compression using Shannon-fano algorithm implemented by VHDL," *IEEE Int. Conf. Advances in Engineering & Technology Research*, pp. 1-5, 2014.

[3] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, issue 9, pp. 1098-1101, 1952.

[4] J. Rissanen, Jorma, and Glen G. Langdon. "Arithmetic coding," *IBM J. Research and Development*, vol. 23, issue 2, pp. 149-162, 1979.

[5] A. Said, “Comparative analysis of arithmetic coding computational complexity,” *Data Compression Conference*, pp. 1-21, 2005.

[6] H. Li and J. Zhang. "A secure and efficient entropy coding based on arithmetic coding," *Communications in Nonlinear Science and Numerical Simulation*, vol. 14, issue 12, pp. 4304-4318, 2009.

[7] V. Itier and W. Puech, "High capacity data hiding for 3D point clouds based on static arithmetic coding," *Multimedia Tools and Applications*, vol. 76, issue 24, pp. 26421-26445, 2017.

[8] K. Sayood, *Introduction to Data Compression*, Morgan Kaufmann, 2017.

[9] P. G. Howard and J. S. Vitter. "Analysis of arithmetic coding for data compression," *Information Processing & Management*, vol. 28, issue 6, pp. 749-763, 1992.

[10] J. J. Ding, I. H. Wang, and H. Y. Chen, “Improved efficiency on adaptive arithmetic coding for data compression using range-adjusting scheme, increasingly adjusting step, and mutual-learning scheme,” *IEEE Trans. Circuits Syst. Video Technol.,* vol. 28, issue 12, pp. 3412-3423, Dec. 2018.

[11] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Info. Theory*, vol. 23, issue 3, pp. 337-343, 1977.

[12] A. Ozsoy and M. Swany, “CULZSS: LZSS lossless data compression on CUDA,” *IEEE Int. Conf. Cluster Computing*, pp. 403-411, 2011.

[13] H. Bannai, S. Inenaga, and M. Takeda, “Efficient LZ78 factorization of grammar compressed text,” Int.l Symp. String Processing and Information Retrieval, pp. 86-98, 2012.

[14] Y. Wiseman, “The relative efficiency of data compression by LZW and LZSS,” Data Science Journal, vol. 6, pp. 1-6, 2007.