IJCCS (Indonesian Journal of Computing and Cybernetics Systems)
Vol.12, No.1, January 2018, pp. 73~82
ISSN (print): 1978-1520, ISSN (online): 2460-7258
DOI: 10.22146/ijccs.28707

Optimization of LZW Compression Algorithm With Modification of Dictionary Formation

Restu Maulunida*[1], Achmad Solichin*[2]
[1]Magister Ilmu Komputer, Universitas Budi Luhur, Jakarta, Indonesia
[2]Teknik Informatika, Universitas Budi Luhur, Jakarta, Indonesia
e-mail: [1]vandalz12@gmail.com, *[2]**achmad.solichin@budiluhur.ac.id**

Abstrak

Pada masa sekarang ini, kebutuhan dalam mengakses data telah bertransformasi kedalam data digital dan penggunaannya sudah berkembang sangat pesat. Transformasi ini disebabkan karena penggunaan internet berkembang sangat pesat dan juga pengembangan perangkat mobile yang berkembang secara masif. Orang-orang cenderung menyimpan banyak file dalam media penyimpanan mereka dan mentransfer file dari satu media ke media yang lain. Ketika media penyimpanan mendekati batasnya, maka akan semakin sedikit file yang dapat disimpan. Untuk mengefisiensikan ukuran suatu file, dibutuhkan suatu teknik kompresi. Teknik dictionary coding merupakan salah satu teknik kompresi lossless, LZW merupakan algoritma untuk mengimplementasikan teknik kompresi dictionary coding. Pada algoritma LZW proses pembentukan dictionary menggunakan future based dictionary dan proses encoding menggunakan Fixed Length Code. Hal ini memungkinkan proses encoding menghasilkan urutan yang masih cukup panjang. Penelitian ini akan memodifikasi proses pembentukan dictionary dan menggunakan Variable Length Code, untuk mengoptimasi rasio kompresi. Penelitian ini akan menguji rasio kompresi.

Kata kunci*— Kompresi Data, Variable Length Code, Lossless, LZW*

Abstract

Digital data storage has become a fundamental requirement. The need for efficient storage media increases with increasing number of data to be stored. Data compression algorithms are designed to reduce the size of data. With compressed data, it will save data storage and also speed up the process of data exchange through the network. The dictionary coding technique is one of the lossless compression techniques. It is implemented on the LZW algorithm. In the LZW algorithm, the process of forming a dictionary uses a future based dictionary and encoding process using the Fixed Length Code. It allows the encoding process to produce a sequence that is still quite long. In this study, we modify the process of forming a dictionary on the LZW algorithm and use Variable Length Code to optimize the compression ratio of the algorithm. Based on the test using the data used in this study, the average compression ratio for LZW algorithm is 42,85%, and our proposed algorithm is 38,35%. It proves that the modification of the formation of the dictionary we proposed has not been able to improve the compression ratio of the LZW algorithm.

Keywords— *Data Compression, Variable Length Code, Lossless, LZW*

Received October 3rd,2017; Revised December 20th, 2017; Accepted January 4th, 2018

1. INTRODUCTION

Currently, digital data storage has become a fundamental requirement. The need for efficient storage media increases with increasing number of data to be stored. Data compression algorithms are designed to reduce the size of data. With compressed data, it will save data storage and also speed up the process of data exchange through the network [1][2][3]. People tend to store files in their storage when storage is close to the limit; they try to reduce the file size by using a software for data compression [4]. Compression is not only done for files of type documents, but also on digital images [5], audio, and video.

The compression algorithm basically consists of two types: lossless compression and lossy compression. In lossless compression, the compression process is performed without losing data [6]. Thus, the compressed file can be restored to the original file completely. Lossless compression is generally applied to data that does not tolerate the difference between the original data and the compressed data so that data integrity is maintained. Examples of lossless compression algorithms are Huffman, RLE, LZ77, LZ78, and LZW. Meanwhile, on the lossy compression algorithm, it allows data loss occurs. Therefore, the compressed file cannot be restored to its original file. With the possibility of data loss, the lossy compression algorithm produces a better compression ratio than the lossless compression algorithm. However, the compressed file cannot be restored to the original file [7]. Lossy compression techniques are often applied to image file types, audio, and video. Compression in digital imagery will reduce image size by reducing the intensity and detail of the color, but not reduce the visual quality of the image. Examples of lossy compression algorithms are Differential Modulation, Adaptive Coding and Discrete Cosine Transform (DCT).

The dictionary coding technique is one of the lossless compression techniques. This technique utilizes the structure of the data to be in compression. LZW algorithm is one of the implementations of compression dictionary coding technique. The LZW algorithm forms a list of tables that will encode the sequence of symbols into the N-bit index contained in the table. Table size has 2^N dictionary list. With the encoding process using the N-bit index in the table, the LZW algorithm uses the fixed length code to encode the sequence of symbols. If the bit length used to encode the symbol sequence contained in the dictionary is 12 bits, then an index dictionary with bit length 8-12 will be encoded into 12 bits. The formation of a dictionary on the LZW algorithm uses a future based dictionary and a symbol encoding process using a fixed length code. In the experimental results compression ratio of 42.85%. This study aims to optimize the LZW compression algorithm by modifying the process of forming the dictionary.

Research related modification algorithm that has been done regarding optimization has been done, such as optimization of Artificial Neural Network algorithm [8] and chaos algorithm optimization to predict the number of unemployed people [9]. Some researchers have also modified the LZW algorithm such as Suarjaya [4] which has implemented a new algorithm to optimize data compression called by j-bit *encoding* (JBE). J-bit encoding works by manipulating bits of data to reduce the size and optimize input for other algorithms. The workings of j-bit encoding are by reading input data per byte. Then separate non zero bytes with zero bytes. Non zero bytes will be inserted into Data I and enter bit 1 into a Temporary byte. Zero bytes will only include bit 0 into a Temporary byte. Then the last process by combining the length of data input, Data I, and Data II.

Nishad and Chezian [10] have done the process of searching the sequence of symbols contained in the dictionary, to speed up the formation of the dictionary in the encoding and decoding process of LZW algorithm by applying Binary Search Tree (BST). Meanwhile, Nandi and Kumar [11] have also modified the LZW algorithm. The proposal technique begins with an empty dictionary. The encoded symbol is not contained in a dictionary encoded with 8 bits. Otherwise, it is encoded with the highest code bit length contained in the dictionary. When the dictionary is full, the elements in the dictionary will be removed using the LRU (least recently used) technique.

Jain et al. [6] have also modified the LZW algorithm by applying OLZWH with Adaptive Huffman Coding. The proposal technique begins with an empty dictionary. The encoded symbol is not contained in a dictionary encoded with 8 bits. Otherwise, it is encoded with the highest code bit length contained in the dictionary. And whenever there is an appearance of ASCII code Adaptive Huffman Coding will be applied to it. Gupta et al. [12] modify the dictionary list on the LZW algorithm by pruning the dictionary list. If the dictionary list is full, a function will be called to delete all dictionary lists that have never been used.

In this research, we modified the LZW algorithm in the process of forming the dictionary. The process of forming a dictionary adds a sequence of symbols based on the history of the previous symbol. By applying this dictionary addition process, it can reproduce a list of adjacent symbol sequences in the dictionary that is useful for encoding longer sequences of longer symbols to increase the compression ratio. This research also implements the variable length code for the encoding process, i.e., the code will be encoded with a length of 8-12 bits to maximize the use of the bits to be encoded.

This research performs a comparison of data compression performance by using dictionary coding method that is LZW algorithm and our proposed algorithm. The comparable variable is the size of the data compression ratio. To analyze the comparison between the algorithm, we took the sample of digital data as much as 200 pieces consisting of 5 file types (*.txt, *.doc, *.xls, *.ppt, *.exe). To assist in comparing the size of the compression ratio, we also create data compression applications using Java programming language.

2. METHODS

2.1 System Analysis

This research implements a lossless compression technique by modifying the LZW algorithm. The application designed in this research is a data compression application using Java programming language. Through this application is done data compression using LZW algorithm and our proposed algorithm. Next, compare the compression ratio of the two algorithms. The compression ratio calculation process used in this study is shown in the equation (1).

$$(((\mathit{original\_size} - \mathit{compressed\_size}) / \mathit{original\_size}) * 100) \qquad (1)$$

The application program menu schematic is shown in Fig 1.

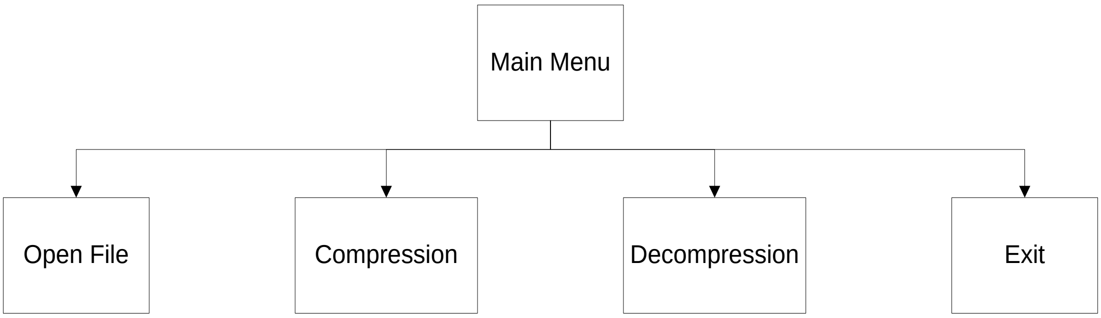
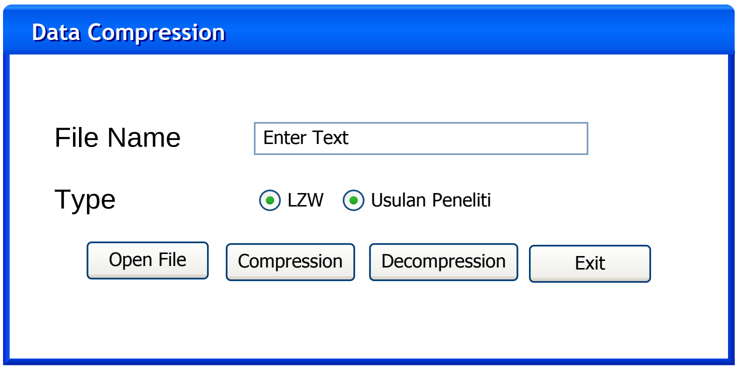


Figure 1. Application menu design


Data Compression

File Name: Enter Text

Type: ● LZW ● Usulan Peneliti

Figure 2. Application screen design

In Fig 2 is a screen design for the proposed application. For the operation of this application, the user first clicks the Open File button and choose the file to be compressed or decompressed. After the user chooses the file, the application will display the selected file name in the file name field. Next, the user must choose the type in the field type for what process will be done, in this application, there are two types of LZW and Proposed Algorithm (modified LZW). The next stage is the user to do the compression and decompression process, to perform file compression the user clicks the Compression button and to decompress the user file clicking the Decompression button. Exit button is used to exit the application.

2.2 Algorithm Design

2.2.1 The Original LZW Algorithm

The original LZW algorithm is an algorithm which is included in the dictionary coding technique. The process of forming a dictionary on the LZW algorithm uses a future based dictionary and symbol encoding process using fixed length code. For example, there is a sequence of symbols to be encoded, i.e., *wabba/bwabba/bwabba/bwabba/bwoo/bwoo/bwoo* [14]. The symbol */b* instead of a space symbol. It is assumed that the source of the alphabet is *{/b, a, b, o, w}*, dictionary for the initial conditions of the LZW algorithm shown in Table 1.

Table 1. Initial condition of dictionary of the LZW Algorithm

Index	*Entry*
1	*/b*
2	*a*
3	*b*
4	*o*
5	*w*

The encoder first looks for the *w* pattern in the dictionary. If this pattern is found in the dictionary, then the encoder adds the *w* pattern to the next pattern which is in the source, i.e., pattern *a*, forming the *wa* pattern. If this pattern is not found in the dictionary, so the encoder will encode the pattern *w* with index 5, and add the pattern *wa* into the dictionary with index 6, then the search for the new pattern will start with the symbol *a*. This process is continued until all sequences of symbols are encoded, the dictionary forming process for encoding is shown in Table 2.

Table 2 Forming of dictionary for encoding LZW

Index	*Entry*		*Index*	*Entry*
1	*/b*		16	*ba/b*
2	*a*		17	*/bwa*
3	*b*		18	*abb*
4	*o*		19	*ba/bw*
5	*w*		20	*wo*
6	*wa*		21	*oo*
7	*ab*		22	*o/b*
8	*bb*		23	*/bwo*
9	*ba*		24	*oo/b*
10	*a/b*		25	*/bwoo*
11	*/bw*			
12	*wab*			
13	*bba*			
14	*a/bw*			
15	*wabb*			

After the encoding process has been done for all symbols contained in the alphabetical order, the result of the output sequence generated by the encoder is {5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4}. The next process is to do the decoding, the result of the output sequence generated by the encoder is {5 2 3 3 2 1 6 8 12 12 11 11 16 16 16 16 4}. The decoding process reads the sequence of symbols generated during the encoding process and forms the same dictionary as the encoding process. The decoding process reads the symbol sequence of encoding results as an index in the dictionary.

2.2.2 The Proposed Algorithm

The proposed algorithm applied in this research is based on LZW algorithm by modifying the process of forming dictionary and using variable length code with a length of 8-12 bits. For example, there is a sequence of symbols to be encoded, i.e., *wabba/bwabba/bwabba/bwabba/bwoo/bwoo/bwoo* [14]. The symbol */b* instead of a space symbol. It is assumed that the source of the alphabet is *{/b, a, b, o, w}*, dictionary for the initial conditions of our proposed algorithm shown in Table 3.

Tabel 3 Initial conditions of dictionary of the proposed algorithm

Index	*Entry*
1	*/b*
2	*a*
3	*b*
4	*o*
5	*w*

The encoder first places the *prevDictionary* pointer on the empty position and looks for the *w* pattern inside the dictionary. If this pattern is found in the dictionary, then the encoder reads the next *a* pattern and combines it into a *wa* pattern. If this pattern is not found in the dictionary, the encoder returns index 5, adds the pattern *wa* into the dictionary with index 6, fills the *prevDictionary* pointer with the *wa* pattern, and adds it to the dictionary if the pattern is not already registered. Next, the search for the new pattern will start from the symbol *a* and the *prevDictionary* pointer is placed on the symbol *w*, then the encoder read the next pattern that is the *b* pattern and combine it into the *ab* pattern. If this pattern is not found in the dictionary, the encoder returns index 2, adds the *ab* pattern to the dictionary with index 7, fills the *prevDictionary* pointer with the *wab* pattern, and adds it into the dictionary with index 8. Next, the search for the new pattern will start from the symbol *b* and the pointer *prevDictionary* is placed on symbol *a*. This process continues until the entire sequence of symbols in the data source has been encoded. The process of forming a dictionary for encoding is shown in Table 4.

Table 4. Forming in dictionary for encoding the proposed algorithm

Index	*Entry*		*Index*	*Entry*
1	*/b*		21	*wabba*
2	*a*		22	*/bwabba*
3	*b*		23	*a/bwo*
4	*o*		24	*ba/bwo*
5	*w*		25	*oo*
6	*wa*		26	*woo*
7	*ab*		27	*o/b*
8	*wab*		28	*oo/b*
9	*bb*		29	*/bwo*
10	*abb*		30	*o/bwo*
11	*ba*		31	*oo/bw*
12	*bba*		32	*woo/bw*
13	*a/b*		33	*woo*
14	*ba/b*		34	*/bwo*
15	*/bw*			
16	*a/bw*			
17	*wabb*			
18	*/bwabb*			
19	*ba/bw*			
20	*bba/bw*			

After the encoding process has been performed for all symbols contained in the alphabetical order, the result of the output sequence generated by the encoder is {5 2 3 3 2 1 8 14 17 16 4 4 15 28 5 25}. The next process is to do the decoding, the result of the output sequence generated by the encoder is {5 2 3 3 2 1 8 14 17 16 4 4 15 28 5 25}. The decoding process reads the sequence of symbols generated during the encoding process and forms the same dictionary as the encoding process. The decoding process reads the symbol sequence of encoding results as an index in the dictionary. Figures 3 and 4 show the compression and decompression process of the proposed algorithm.

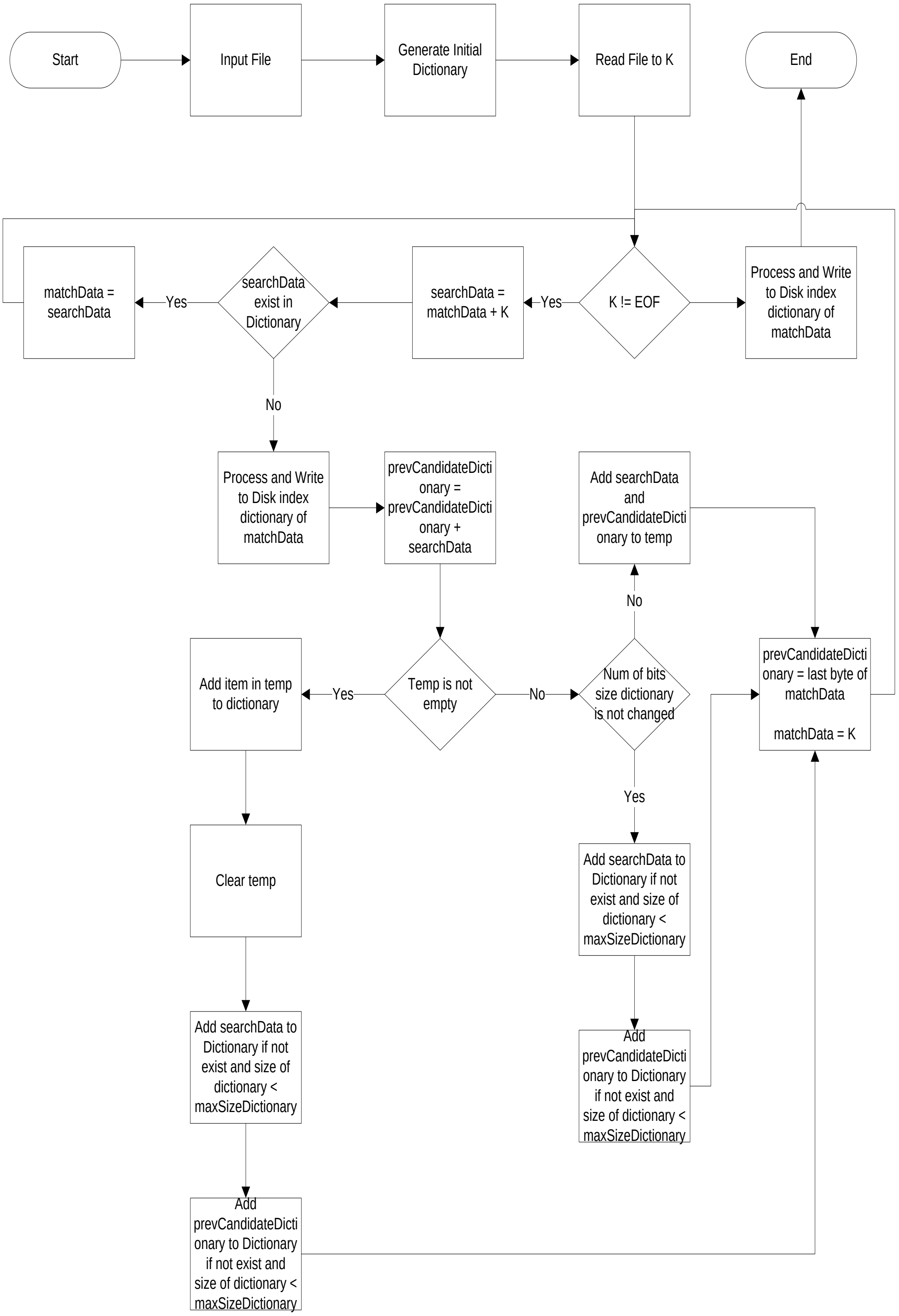
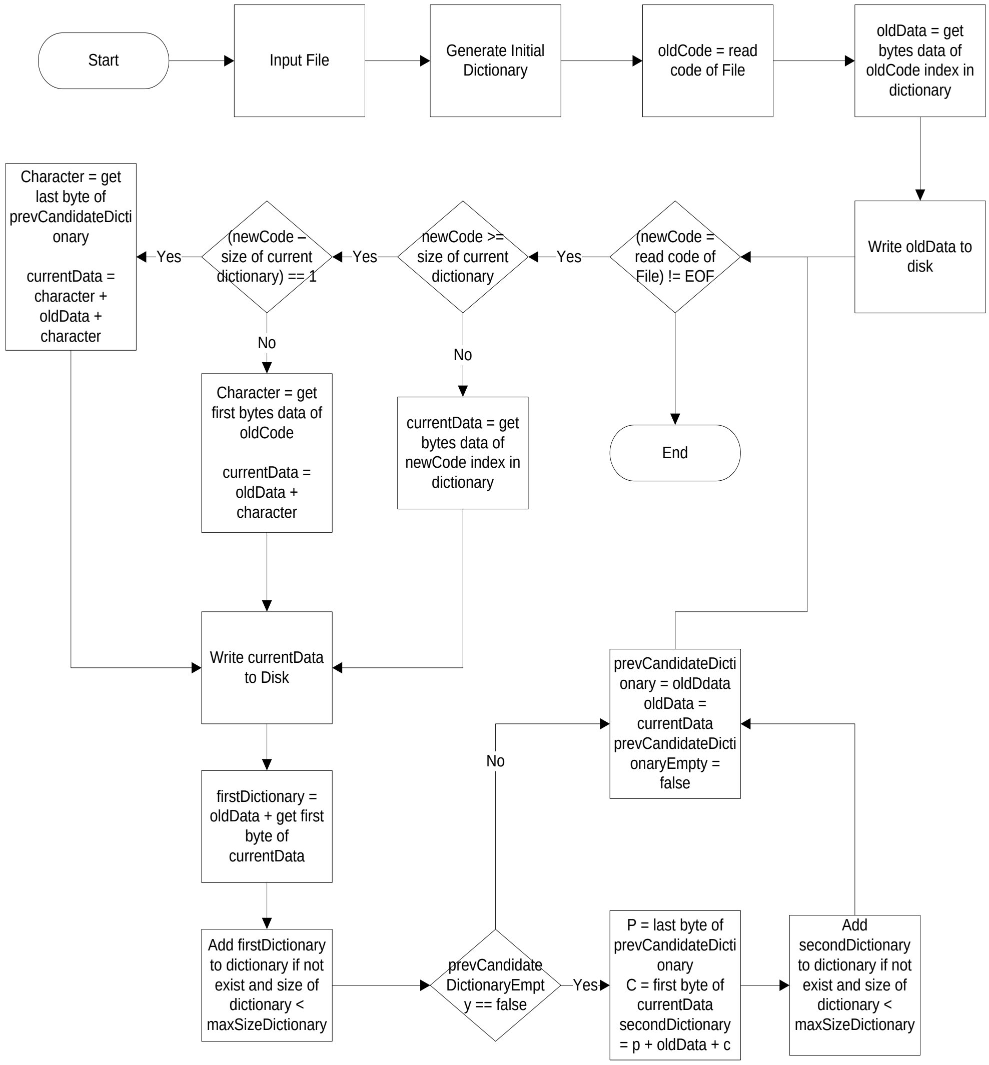

Start
Input File
Generate Initial Dictionary
Read File to K
End
matchData = searchData
Yes
searchData exist in Dictionary
searchData = matchData + K
Yes
K != EOF
Process and Write to Disk index dictionary of matchData
No
Process and Write to Disk index dictionary of matchData
prevCandidateDictionary = prevCandidateDictionary + searchData
Add searchData and prevCandidateDictionary to temp
No
Add item in temp to dictionary
Yes
Temp is not empty
No
Num of bits size dictionary is not changed
prevCandidateDictionary = last byte of matchData
matchData = K
Yes
Clear temp
Add searchData to Dictionary if not exist and size of dictionary < maxSizeDictionary
Add searchData to Dictionary if not exist and size of dictionary < maxSizeDictionary
Add prevCandidateDictionary to Dictionary if not exist and size of dictionary < maxSizeDictionary
Add prevCandidateDictionary to Dictionary if not exist and size of dictionary < maxSizeDictionary

Figure 3. The Flowchart of the proposed compression algorithm


Start
Input File
Generate Initial Dictionary
oldCode = read code of File
oldData = get bytes data of oldCode index in dictionary
Write oldData to disk
(newCode = read code of File) != EOF
Yes
newCode >= size of current dictionary
Yes
(newCode – size of current dictionary) == 1
Yes
Character = get last byte of prevCandidateDictionary

currentData = character + oldData + character
No
Character = get first bytes data of oldCode

currentData = oldData + character
No
currentData = get bytes data of newCode index in dictionary
End
Write currentData to Disk
firstDictionary = oldData + get first byte of currentData
Add firstDictionary to dictionary if not exist and size of dictionary < maxSizeDictionary
prevCandidateDictionaryEmpty == false
No
Yes
prevCandidateDictionary = oldDdata
oldData = currentData
prevCandidateDictionaryEmpty = false
P = last byte of prevCandidateDictionary
C = first byte of currentData
secondDictionary = p + oldData + c
Add secondDictionary to dictionary if not exist and size of dictionary < maxSizeDictionary

Figure 4. The Flowchart of the proposed decompression algorithm

3. RESULTS AND DISCUSSION

In this section, we will discuss the results of the testing of the compression algorithm. The experimental data was performed on 200 files consisting of 5 file types: extension *.doc, *.exe, *.ppt, *.txt and *.xls. Table 5 shows the average of the compression ratio of the experimental result.

Table 5. Compression ratio of the proposed algorithm and the original LZW algorithm

File type	Compression ratio (%)	
	Original LZW	Proposed algorithm
doc	40,13	36,83
exe	12,51	9,08
ppt	34,57	31,22
txt	48,50	45,15
xls	46,87	39,85
Average	42,85	38,35

Table 5 shows the average of the experimental results. The size of the compression ratio result is written in percentage units. Based on the average result data in Table 5 it is known that the average compression ratio for LZW algorithm is 42.85% and the average compression ratio for the proposed algorithm is 38.35%. The compression ratio used in this research is the storage media space that can be spared.

This research proposes a new method to perform data compression by dictionary coding technique by modifying the formation of the dictionary on LZW algorithm and applying variable length code. Nevertheless, the result is still not very good. The compression ratio of the proposed algorithm is smaller than the original LZW algorithm. On the other hand, the resulting application can perform file compression of various types of file types and can perform the decompression process into its original form. However, there are cases where the compression results are larger than the original file size. This happens because the dictionary coding technique is influenced by the structure of the data and the dictionary.

4. CONCLUSION

Based on the test using the data used in this study, seen the difference in the compression ratio between the original LZW algorithm and our proposed algorithm. The average compression ratio of experimental results for LZW algorithm is 42,85%, and our proposed algorithm is 38,35%. It proves that the compression ratio of our algorithm is smaller than the original LZW compression ratio which states that the LZW algorithm is better than the research proposed algorithm.

Based on the discussion that has been described, the authors provide suggestions based on what has been known to the author of this data compression research. For further development, it is expected that the writer's suggestion algorithm can be improved so that it can improve the performance of the algorithm that is the compression ratio and expected to perform the compression and decompression process for many files or folders or files in the folder.

REFERENCES

[1] W. Al Hayek, “An Effective Method For Data Compression Based On Adaptive Character Wordlength.pdf,” *Int. Arab J. e-Technology*, vol. 2, no. 4, pp. 197–201, 2012.

[2] A. Kaur and N. S. Sethi, “Approach for Lossless Text data Compression using Advanced Bit Reduction Algorithm,” *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 5, no. 7, pp. 1172–1176, 2015.

[3] S. M. Choudhary, A. S. Patel, and S. J. Parmar, “Study of LZ77 and LZ78 Data Compression Techniques,” *Certif. Int. J. Eng. Sci. Innov. Technol.*, vol. 4, no. 3, pp. 45–49, 2015.

[4] I. M. A. D. Suarjaya, “A New Algorithm for Data Compression Optimization,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 3, no. 8, pp. 14–17, 2012.

[5] A. P. Utomo, A. E. Putra, and C. Atmaji, “Analisis Hasil Proses Pemampatan JPEG dengan Metode Discrete Cosine Transform,” *IJEIS*, vol. 2, no. 1, pp. 1–10, 2013.

[6] P. Jain, A. Jain, and C. Agrawal, “IMPROVING DATA COMPRESSION RATIO BY THE USE OF OPTIMALITY OF LZW &amp; ADAPTIVE HUFFMAN ALGORITHM (OLZWH),” *Int. J. Inf. Theory*, vol. 4, no. 1, pp. 11–19, 2015.

[7] M. Kaur and G. Kaur, “A Survey of Lossless and Lossy Image Compression Techniques,” *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 3, no. 2, pp. 323–326, 2013.

[8] H. G. Nugraha and A. S. N., “Optimasi Bobot Jaringan Syaraf Tiruan Mengunakan Particle Swarm Optimization,” *Indones. J. Comput. Cybern. Syst.*, vol. 8, no. 1, pp. 25–36, 2014.

[9] R. Pramitasari and R. Wardoyo, “Penerapan Algoritma Optimasi Chaos pada Jaringan

Ridge Polynomial untuk Prediksi Jumlah Pengangguran," *IJCCS-Indonesian J. Comput. Cybern. Syst.*, vol. 6, no. 2, pp. 47–56, 2012.

[10] P. M. Nishad and R. M. Chezian, "OPTIMIZATION OF LZW (LEMPEL-ZIV-WELCH) ALGORITHM TO REDUCE TIME COMPLEXITY FOR DICTIONARY CREATION IN ENCODING AND DECODING," *Asian J. Comput. Sci. Inf. Technol.*, vol. 5, pp. 114–118, 2012.

[11] U. Nandi and J. K. Mandal, "Modified Compression Techniques Based on Optimality of LZW Code (MOLZW)," *Int. Conf. Comput. Intell. Model. Tech. Appl.*, vol. 10, pp. 949–956, 2013.

[12] N. Gupta, R. Kumar, and A. Gupta, "Removing Redundancy in Dictionary based Compression Techniques," *Int. J. Comput. Sci. Emerg. Technol.*, vol. 1, no. 4, pp. 237–240, 2010.

[13] M. Singh, S. Kumar, S. Singh, and M. Shrivastava, "Various Image Compression Techniques : Lossy and Lossless," *Int. J. Comput. Appl.*, vol. 142, no. 6, pp. 23–26, 2016.

[14] K. Sayood, *Introduction to Data Compression*, 3rd ed. San Francisco: Morgan Kaufmann, 2012.