COMPARISON OF LOSSLESS DATA COMPRESSION ALGORITHMS FOR TEXT DATA

S.R. KODITUWAKKU[1]
Department of Statistics & Computer Science, University of Peradeniya, Sri Lanka
salukak@pdn.ac.lk

U. S.AMARASINGHE
Postgraduate Institute of Science, University of Peradeniya, Sri Lanka
udeshamar@gmail.com

Abstract

Data compression is a common requirement for most of the computerized applications. There are number of data compression algorithms, which are dedicated to compress different data formats. Even for a single data type there are number of different compression algorithms, which use different approaches. This paper examines lossless data compression algorithms and compares their performance. A set of selected algorithms are examined and implemented to evaluate the performance in compressing text data. An experimental comparison of a number of different lossless data compression algorithms is presented in this paper. The article is concluded by stating which algorithm performs well for text data.

Keywords: *Data compression, Encryption, Decryption, Lossless Compression, Lossy Compression*

1. Introduction

Compression is the art of representing the information in a compact form rather than its original or uncompressed form [1]. In other words, using the data compression, the size of a particular file can be reduced. This is very useful when processing, storing or transferring a huge file, which needs lots of resources. If the algorithms used to encrypt works properly, there should be a significant difference between the original file and the compressed file. When data compression is used in a data transmission application, speed is the primary goal. Speed of transmission depends upon the number of bits sent, the time required for the encoder to generate the coded message and the time required for the decoder to recover the original ensemble. In a data storage application, the degree of compression is the primary concern. Compression can be classified as either lossy or lossless. Lossless compression techniques reconstruct the original data from the compressed file without any loss of data. Thus the information does not change during the compression and decompression processes. These kinds of compression algorithms are called reversible compressions since the original message is reconstructed by the decompression process. Lossless compression techniques are used to compress medical images, text and images preserved for legal reasons, computer executable file and so on [2]. Lossy compression techniques reconstruct the original message with loss of some information. It is not possible to reconstruct the original message using the decoding process, and is called irreversible compression [3]. The decompression process results an approximate reconstruction. It may be desirable, when data of some ranges which could not recognized by the human brain can be neglected. Such techniques could be used for multimedia images, video and audio to achieve more compact data compression.

Various lossless data compression algorithms have been proposed and used. Some of the main techniques in use are the Huffman Coding, Run Length Encoding, Arithmetic Encoding and Dictionary Based Encoding [3]. This paper examines the performance of the Run Length Encoding Algorithm, Huffman Encoding Algorithm, Shannon Fano Algorithm, Adaptive Huffman Encoding Algorithm, Arithmetic Encoding Algorithm and Lempel Zev Welch Algorithm. In particular, performance of these algorithms in compressing text data is evaluated and compared.

2. Methods and Materials

In order to evaluate the effectiveness and efficiency of lossless data compression algorithms the following materials and methods are used.

2.1 ***Materials***

Among the available lossless compression algorithms the following are considered for this study.

Run Length Encoding Algorithm

Run Length Encoding or simply RLE is the simplest of the data compression algorithms. The consecutive sequences of symbols are identified as runs and the others are identified as non runs in this algorithm. This algorithm deals with some sort of redundancy [2]. It checks whether there are any repeating symbols or not, and is based on those redundancies and their lengths. Consecutive recurrent symbols are identified as runs and all the other sequences are considered as non-runs. For an example, the text "ABABBBBC" is considered as a source to compress, then the first 3 letters are considered as a non-run with length 3, and the next 4 letters are considered as a run with length 4 since there is a repetition of symbol B. The major task of this algorithm is to identify the runs of the source file, and to record the symbol and the length of each run. The Run Length Encoding algorithm uses those runs to compress the original source file while keeping all the non-runs with out using for the compression process.

Huffman Encoding

Huffman Encoding Algorithms use the probability distribution of the alphabet of the source to develop the code words for symbols. The frequency distribution of all the characters of the source is calculated in order to calculate the probability distribution. According to the probabilities, the code words are assigned. Shorter code words for higher probabilities and longer code words for smaller probabilities are assigned. For this task a binary tree is created using the symbols as leaves according to their probabilities and paths of those are taken as the code words. Two families of Huffman Encoding have been proposed: Static Huffman Algorithms and Adaptive Huffman Algorithms. Static Huffman Algorithms calculate the frequencies first and then generate a common tree for both the compression and decompression processes [2]. Details of this tree should be saved or transferred with the compressed file. The Adaptive Huffman algorithms develop the tree while calculating the frequencies and there will be two trees in both the processes. In this approach, a tree is generated with the flag symbol in the beginning and is updated as the next symbol is read.

The Shannon Fano Algorithm

This is another variant of Static Huffman Coding algorithm. The only difference is in the creation of the code word. All the other processes are equivalent to the above mentioned Huffman Encoding Algorithm.

Arithmetic Encoding

In this method, a code word is not used to represent a symbol of the text. Instead it uses a fraction to represent the entire source message [5]. The occurrence probabilities and the cumulative probabilities of a set of symbols in the source message are taken into account. The cumulative probability range is used in both compression and decompression processes. In the encoding process, the cumulative probabilities are calculated and the range is created in the beginning. While reading the source character by character, the corresponding range of the character within the cumulative probability range is selected. Then the selected range is divided into sub parts according to the probabilities of the alphabet. Then the next character is read and the corresponding sub range is selected. In this way, characters are read repeatedly until the end of the message is encountered. Finally a number should be taken from the final sub range as the output of the encoding process. This will be a fraction in that sub range. Therefore, the entire source message can be represented using a fraction. To decode the encoded message, the number of characters of the source message and the probability/frequency distribution are needed.

The Lempel Zev Welch Algorithm

Dictionary based compression algorithms are based on a dictionary instead of a statistical model [5]. A dictionary is a set of possible words of a language, and is stored in a table like structure and used the indexes of entries to represent larger and repeating dictionary words. The Lempel-Zev Welch algorithm or simply LZW algorithm is one of such algorithms. In this method, a dictionary is used to store and index the previously seen string patterns. In the compression process, those index values are used instead of repeating string patterns. The

dictionary is created dynamically in the compression process and no need to transfer it with the encoded message for decompressing. In the decompression process, the same dictionary is created dynamically. Therefore, this algorithm is an adaptive compression algorithm.

Measuring Compression Performances

Depending on the nature of the application there are various criteria to measure the performance of a compression algorithm. When measuring the performance the main concern would be the space efficiency. The time efficiency is another factor. Since the compression behavior depends on the redundancy of symbols in the source file, it is difficulty to measure performance of a compression algorithm in general. The performance depends on the type and the structure of the input source. Additionally the compression behavior depends on the category of the compression algorithm: lossy or lossless. If a lossy compression algorithm is used to compress a particular source file, the space efficiency and time efficiency would be higher than that of the lossless compression algorithm. Thus measuring a general performance is difficult and there should be different measurements to evaluate the performances of those compression families. Following are some measurements used to evaluate the performances of lossless algorithms.

Compression Ratio is the ratio between the size of the compressed file and the size of the source file.

$$compression\ \ Ratio = \frac{size\ \ after\ \ compression}{size\ \ before\ \ compression}$$

Compression Factor is the inverse of the compression ratio. That is the ratio between the size of the source file and the size of the compressed file.

$$compression\ \ Ratio = \frac{size\ \ before\ \ compression}{size\ \ after\ \ compression}$$

Saving Percentage calculates the shrinkage of the source file as a percentage.

$$saving\ \ percentage = \frac{size\ \ before\ \ compression - size\ \ after\ \ compression}{size\ \ before\ \ compression}\%$$

All the above methods evaluate the effectiveness of compression algorithms using file sizes. There are some other methods to evaluate the performance of compression algorithms. Compression time, computational complexity and probability distribution are also used to measure the effectiveness.

Compression Time

Time taken for the compression and decompression should be considered separately. Some applications like transferring compressed video data, the decompression time is more important, while some other applications both compression and decompression time are equally important. If the compression and decompression times of an algorithm are less or in an acceptable level it implies that the algorithm is acceptable with respective to the time factor. With the development of high speed computer accessories this factor may give very small values and those may depend on the performance of computers.

Entropy

This method can be used, if the compression algorithm is based on statistical information of the source file. Self Information is the amount of one's surprise evoked by an event. In another words, there can be two events: first one is an event which frequently happens and the other one is an event which rarely happens. If a message says that the second event happens, then it will generate more surprise in receivers mind than the first message. Let set of event be S = $\{s_1,s_2,s_3,...,s_n)$ for an alphabet and each s_j is a symbol used in this alphabet. Let the occurrence probability of each event be p_j for event s_j. Then the self information I(s) is defined as follows.

$$I(s_j) = \log_b \frac{1}{p_j} \text{ or } I(s_j) = -\log_b \frac{1}{p_j}$$

The first order Entropy value H (P) of a compression algorithm can be computed as follows.

$$H(P) = \sum_{j=1}^{n} p_j I(s_j) \text{ or } H(P) = -\sum_{j=1}^{n} p_j I(s_j)$$

Code Efficiency

Average code length is the average number of bits required to represent a single code word. If the source and the lengths of the code words are known, the average code length can be calculated using the following equation.

$\bar{l} = \sum_{j=1}^{n} p_j l_j$, where p_i is the occurrence probability of j^{th} symbol of the source message, l_j is the length of the particular code word for that symbol and $L = \{l_1, l_2, \ldots. L_n\}$.

Code Efficiency is the ratio in percentage between the entropy of the source and the average code length and it is defined as follows.

$E(P,L) = \frac{H(P)}{\bar{l}(P,L)} 100\%$, where E (P, L) is the code efficiency, H (P) is the entropy and $\bar{l}$ (P, L) is the average code length.

The above equation is used to calculate the code efficiency as a percentage. It can also be computed as a ratio. The code is said to be optimum, if the code efficiency values is equal to 100% (or 1.0). If the value of the code efficiency is less than 100%, that implies the code words can be optimized than the current situation.

2.2 Methodology

In order to test the performance of lossless compression algorithms, the Run Length Encoding Algorithm, Huffman Encoding Algorithm, Shannon Fano Algorithm, Adaptive Huffman Encoding Algorithm, Arithmetic Encoding Algorithm and Lempel Zev Welch Algorithm are implemented and tested with a set of text files. Performances are evaluated by computing the above mentioned factors.

Measuring the Performance of RLE Algorithm

Since the Run Length Encoding Algorithm does not use any statistical method for the compression process, the Compression and Decompression times, File Sizes, Compression Ratio and Saving Percentage are calculated. Several files with different file sizes and text patterns are used for computation.

Measuring the Performance of Static Huffman Approaches

Static Huffman Encoding and Shannon Fano approaches are implemented and executed independently. For these two approaches, file sizes, compression and decompression times, entropy and code efficiency are calculated.

Measuring the Performance of Adaptive Huffman Encoding

Adaptive Huffman Encoding is also implemented in order to compare with other compression and decompression algorithms. A dynamic code word is used by this algorithm. File sizes, compression and decompression times, entropy and code efficiency are calculated for Adaptive Huffman Algorithm.

Measuring the Performance of LZW Algorithm

Since this algorithm is not based on a statistical model, entropy and code efficiency are not calculated. Compression and decompression process, file sizes, compression ratio and saving percentages are calculated.

Measuring the Performance of Arithmetic Encoding Algorithm

The compression and decompression times and file sizes are calculated for this algorithm. Because of the underflow problem, the original file can not be generated after the decompression process. Therefore, these values can not be considered as the actual values of the Arithmetic encoding algorithm. So the results of this algorithm are not used for the comparison process.

S.R. Kodituwakku et. al. / Indian Journal of Computer Science and Engineering
Vol 1 No 4 416-426

Evaluating the performance

The performance measurements discussed in the previous section are based on file sizes, time and statistical models. Since they are based on different approaches, all of them can not be applied for all the selected algorithms. Additionally, the quality difference between the original and decompressed file is not considered as a performance factor as the selected algorithms are lossless. The performances of the algorithms depend on the size of the source file and the organization of symbols in the source file. Therefore, a set of files including different types of texts such as English phrases, source codes, user manuals, etc, and different file sizes are used as source files. A graph is drawn in order to identify the relationship between the file sizes, the compression and decompression time.

Comparing the Performance

The performances of the selected algorithms vary according to the measurements, while one algorithm gives a higher saving percentage it may need higher processing time. Therefore, all these factors are considered for comparison in order to identify the best solution. An algorithm which gives an acceptable saving percentage within a reasonable time period is considered as the best algorithm.

3. Results and Discussion

Six lossless compression algorithms are tested for ten text files with different file sizes and different contents. The sizes of the original text files are 22094 bytes, 44355 bytes, 11252 bytes, 15370 bytes, 78144 bytes, 78144 bytes, 39494 bytes, 118223 bytes, 180395 bytes, 242679 bytes and 71575 bytes. The text of the first 3 files is in normal English language. The next two files are computer programs, which have more repeating set of words than the previous case. The last five file are taken from E-books which are in normal English language. Followings are the results for 10 different text files.

3.1 Results

Due to the underflow problem, an accurate result is not given by the Arithmetic encoding algorithm. According to the results given in following tables and graphs, all the algorithms work well except Run length encoding algorithm. The LZW algorithm does not work well for large files, since it has to maintain huge dictionaries for compression and decompression processors. It requires lots of computer resources to process and the overflow problem is also occurred.

Table 1: Run Length Encoding results

Original File			Run Length			
File	File Size	No of characters	Compressed File size	Compression Ratio	Compression Time	Decompression Time
1	22,094	21,090	22,251	100.7106002	359	2672
2	44,355	43,487	43,800	98.7487318	687	2663
3	11,252	10,848	11,267	100.1333096	469	2844
4	15,370	14,468	13,620	88.6141835	94	2500
5	78,144	74,220	68,931	88.2102273	1234	17359
6	39,494	37,584	37,951	96.0930774	141	2312
7	118,223	113,863	118,692	100.3967079	344	1469
8	180,395	172,891	179,415	99.4567477	2766	2250
9	242,679	233,323	242,422	99.8940988	2953	1828
10	71,575	68,537	71,194	99.4676912	344	1532

According to the results shown in Table 1, compression and decompression times are relatively low. However, for the first third and seventh files, this algorithm generates compressed files larger than the original files. This happens due to the fewer amounts of runs in the source. All other files are compressed but the compression ratios are very high values. So this algorithm can reduce about 2% of the original file, but this can not be considered as a reasonable value.

Table 2: LZW algorithm results

Original File			LZW			
File	File Size	No of characters	Compressed File size	Compression Ratio	Compression Time	Decompression Time
1	22,094	21,090	13,646	61.7633747	51906	7000
2	44,355	43,487	24,938	56.2236501	167781	7297
3	11,252	10,848	7,798	69.303235	15688	3422
4	15,370	14,468	7,996	52.0234223	21484	3234
5	78,144	74,220	24,204	30.9735872	279641	11547
6	39,494	37,584	21,980	55.6540234	66100	5428
7	118,223	113,863	58,646	49.6062526	517739	18423
8	71,575	68,537	36,278	50.6852951	187640	5611

A dynamic dictionary is used by this algorithm and gives good compression ratios for the source text files. The disadvantage is that the size of the dictionary got increased with the size of the file since more and more entries are added by the algorithm. Table 2 shows low efficiency, because lot of resources is required to process the dictionary. This algorithm gives a good compression ratio which lies between 30% and 60%. This is a reasonable value when it compared with the other algorithms. The compression ratio decreases as the file size increases, since the number of words can be represented by shorter dictionary entries.

Table 3: Adaptive Huffman Algorithm results

Original File			Adaptive Huffman			
File	File Size	No of characters	Compressed File size	Compression Ratio	Compression Time	Decompression Time
1	22,094	21,090	13,432	60.7947859	80141	734469
2	44,355	43,487	26,913	60.6763612	223875	1473297
3	11,252	10,848	7,215	64.1219339	30922	297625
4	15,370	14,468	8,584	55.8490566	41141	406266
5	78,144	74,220	44,908	57.4682637	406938	2611891
6	39,494	37,584	22,863	57.889806	81856	1554182
7	118,223	113,863	73,512	62.1807939	526070	1271041
8	180,395	172,891	103,716	57.493833	611908	1554182
9	242,679	233,323	147,114	60.6208201	1222523	2761631
10	71,575	68,537	44,104	61.6192805	231406	633117

A dynamic tree used in this algorithm has to be modified for each and every character of the message, and this has to be done in both the processes. So the compression and decompression times are relatively high for this algorithm. Results given in Table 3 indicated that it needs higher compression and decompression times. The compression ratios for the selected files are in the range from 55% to 65%. The compression ratio does not depend on the file size but it depends on the structure of the file. File number 4, the program source code, has higher amount of repeating words and this causes the compression ratio of 55.849%.

Compression and decompression times are relatively low for the Huffman Encoding algorithm compared to the adaptive approach because of the usage of static code word. Compression ratios lay between 58% and 67%. Since this is an algorithm which is a static approach based on statistical data, Entropy and Code Efficiency can be calculated. The results in Table 4 show that entropy values lay between 4.3 and 5.1. To compress a single

S.R. Kodituwakku et. al. / Indian Journal of Computer Science and Engineering
Vol 1 No 4 416-426

character of 1 byte, this algorithm needs only 4-5 bits. Code efficiencies are greater than 98% for all the cases. Thus this algorithm can be considered as an efficient algorithm. According to the definition of the Code Efficiency, used code words can be further improved.

Table 4: Huffman Encoding results

Original File		Huffman Encoding					
File	File Size	Comp File size	Compression Ratio	Comp Time	Decomp Time	Entropy	Code Efficiency
1	22,094	13,826	62.5780755	16141	16574	4.788069776	99.2827818
2	44,355	27,357	61.6773757	54719	20606	4.81093394	99.3618801
3	11,252	7,584	67.4013509	3766	6750	5.026545403	99.4386035
4	15,370	8,961	58.3018868	5906	9703	4.354940529	98.6695303
5	78,144	45,367	58.0556409	156844	224125	4.540845425	99.065523
6	39,494	23,275	58.9330025	13044	12638	4.568886939	98.9116913
7	118,223	74,027	62.6164114	134281	99086	4.943608986	99.2741005
8	180,395	104,193	57.7582527	368720	288232	4.597479744	99.4383889
9	242,679	147,659	60.8453966	655514	470521	4.822433287	99.2875144
10	71,575	44,586	62.2927	42046	34293	4.878965395	99.228234

Table 5: Shannon Fano Algorithm results

Original File		Shanon Fano Algorithm					
File	File Size	Comp File size	Comp Ratio	Comp Time	Decomp Time	Entropy	Code Efficiency
1	22,094	14,127	63.9404363	14219	19623	4.788069776	97.0894373
2	44,355	27,585	62.1914102	55078	69016	4.81093394	98.5248397
3	11,252	7,652	68.0056879	3766	8031	5.026545403	98.5034116
4	15,370	9,082	59.0891347	6078	9547	4.354940529	97.2870497
5	78,144	46,242	59.1753686	162609	229625	4.540845425	97.1668752
6	39,494	23,412	59.2798906	12638	12022	4.568886939	98.3156717
7	118,223	75,380	63.7608587	153869	114187	4.943608986	97.4763757
8	180,395	107,324	59.4938884	310686	255933	4.597479744	96.5212428
9	242,679	150,826	62.1504127	549523	441153	4.822433287	97.1932952
10	71,575	44,806	62.6000699	42997	32869	4.878965395	98.7353307

Shannon Fano is another variant of Static Huffman algorithm. Results obtained for this algorithm is given in Table 5. The compression ratios for Shannon Fano approach are in the range of 59% to 64% which is slightly equivalent to the previous algorithm. Entropy values are in the range of 4.3 to 5.1. Code efficiencies are greater than 97%. Even this is relatively less than that of the Huffman algorithm, it is also a significant value.

Saving percentages of all selected algorithms are shown in Table 6. Lowest saving percentage is given by Run Length Encoding algorithm and relatively best values are given by LZW algorithm. Average values are given by all the three Huffman algorithms, but the values of Adaptive method are higher than the two static methods. The differences do not exceed more than 2%.

Table 6: Saving percentages of all selected algorithms

	RLE	LZW	Adaptive	Huffman	Shannon
1	-0.71	38.24	39.21	37.42	36.06
2	1.25	43.78	39.32	38.32	37.81
3	-0.13	30.70	35.88	32.60	31.99
4	11.39	47.98	44.15	41.70	40.91
5	11.79	69.03	42.53	41.94	40.82
6	3.91	44.35	42.11	41.07	40.72
7	-0.40	50.39	37.82	37.38	36.24
8	0.54	55.85	42.51	42.24	40.51
9	0.11		39.38	39.15	37.85
10	0.53	49.31	38.38	37.71	37.40

3.2 Comparison of Results

In order to compare the performances of the selected algorithms the compression and decompression times, and compressed file sizes are compared. Figure 1 shows the compression times of selected 10 files for all the algorithms.

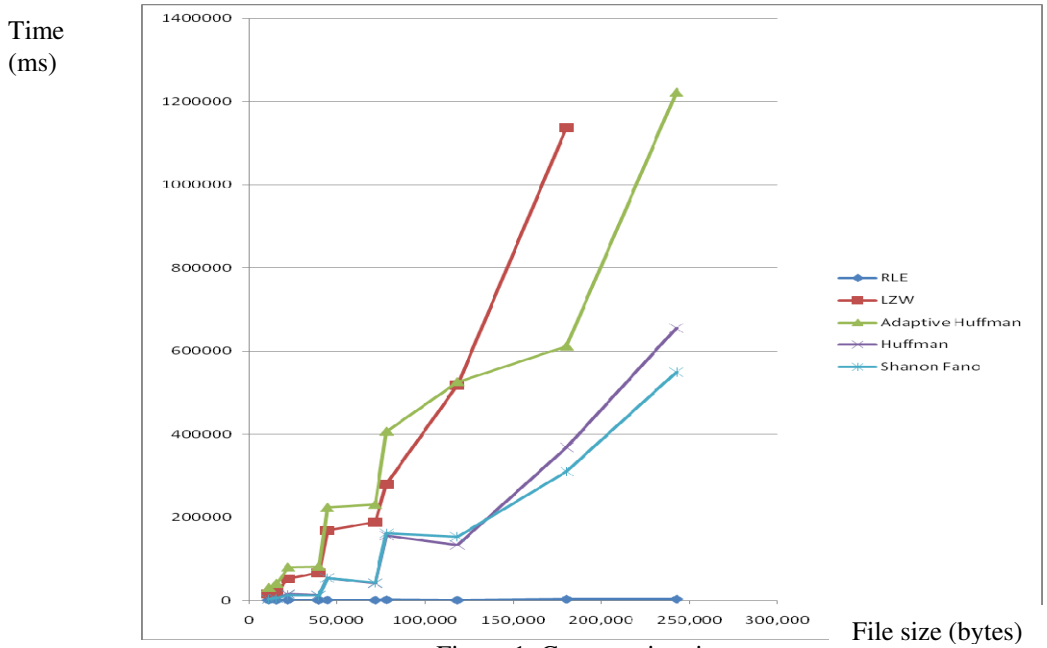

Figure 1: Compression times

The compression time is increased as file size increases. For Run Length Encoding it is a constant value and not effected by the file size. Compression times are average values for two Static Huffman approaches, and times of Shannon Fano approach are smaller than the other algorithm. The LZW algorithm works well for small files but not for the large files due to the size of the dictionary. Compression times of Adaptive Huffman algorithm are the highest.

The decompression times of the algorithms are also compared. Figure 2 depicts the decompression times of all the algorithms. Decompression times of all the algorithms are less than 500000 milliseconds except the Adaptive Huffman Algorithm and LZW. Again the decompression times of the Huffman and Shannon Fano algorithms are almost similar. The decompression times of the Run Length Encoding algorithms are relatively less for all the file sizes.

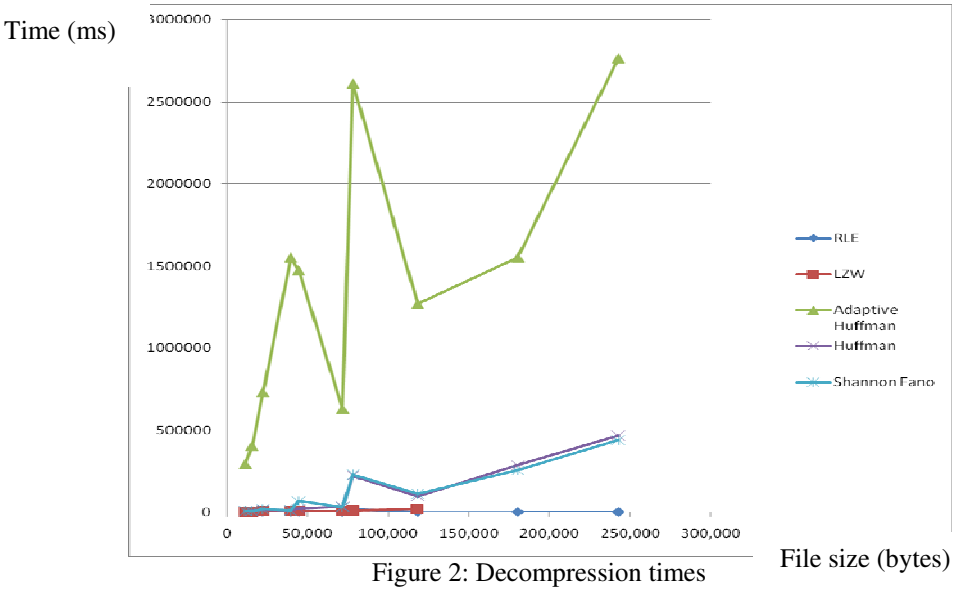

Figure 2: Decompression times

The sizes of the compressed files are compared with the original file sizes. These results are depicted in Figure 3. According to the Figure 3, the saving percentage of the Run Length Encoding is a very less value. The compressed file sizes of all the Huffman approaches are relatively similar. The compressed file size is increased according to the original file size except for the LZW algorithm. This implies that the saving percentage of this algorithm highly depends on the redundancy of the file. The LZW algorithm shows the highest saving percentage from the selected algorithms. However, it fails when the file size is large. So it can not be used in such situations.

3.3 Discussion

Adaptive Huffman Algorithm needs a relatively larger time period for processing, because the tree should be updated or recreated for both processes. The processing time is relatively small since a common tree for both the processes is used and is created only once. LZW approach works better as the size of the file grows up to a certain amount, because there are more chances to replace identified words by using a small index number. However, it can not be considered as the most efficient algorithm, because it can not be applied for all the cases.

The speed of the Run Length Encoding algorithm is high, but the saving percentage is low for all selected text files. Run Length Encoding algorithm is designed to identify repeating symbols and to replace by a set of characters which indicate the symbol and number of characters in the run. The saving percentage is low for selected text files as there is less number of repeating runs.

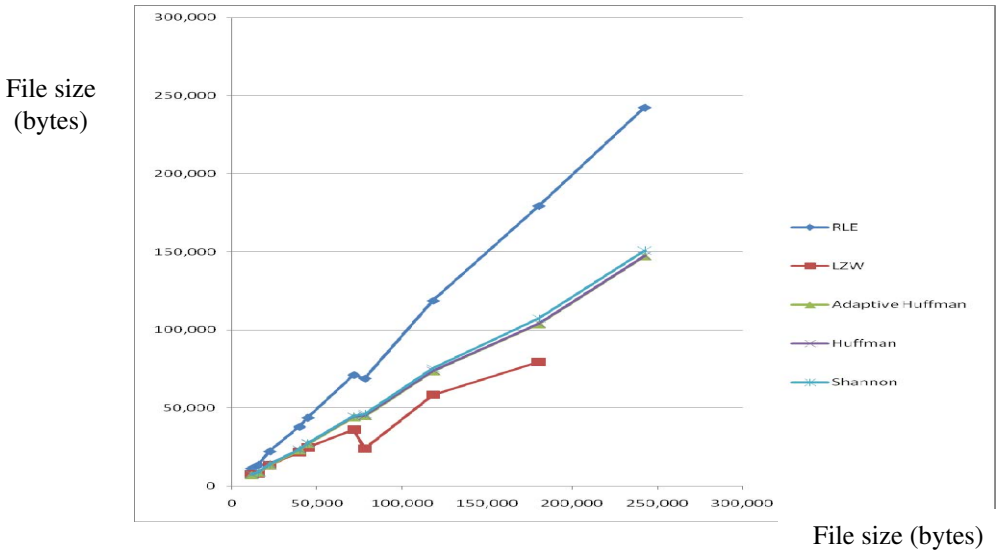

Figure 3: Sizes of compressed files

Arithmetic encoding has the major disadvantage, Underflow Problem, which gives an erroneous result after few number of iteration. Therefore, it is not used for this comparison. All the results of the system for Arithmetic Encoding are neglected.

Huffman Encoding and Shannon Fano algorithm show similar performances except in the compression times. Huffman Encoding algorithm needs more compression time than Shannon Fano algorithm, but the differences of the decompression times and saving percentages are extremely low. The code efficiency of Shannon Fano algorithm is a quite a low value compared to the Huffman encoding algorithm. So the generated code words using Shannon Fano algorithm have to be improved more than the code words of the Huffman Encoding. According to the differences of the compression time Shannon Fano algorithm is faster than the Huffman Encoding algorithm. So this factor can be used to determine the more efficient algorithm from these two.

4. Conclusions

An experimental comparison of a number of different lossless compression algorithms for text data is carried out. Several existing lossless compression methods are compared for their effectiveness. Although they are tested on different type of files, the main interest is on different test patterns. By considering the compression times, decompression times and saving percentages of all the algorithms, the Shannon Fano algorithm can be considered as the most efficient algorithm among the selected ones. Those values of this algorithm are in an acceptable range and it shows better results for the large files.

Reference

[1] Pu, I.M., 2006, *Fundamental Data Compression*, Elsevier, Britain.
[2] Blelloch, E., 2002. *Introduction to Data Compression*, Computer Science Department, Carnegie Mellon University.
[3] Kesheng, W., J. Otoo and S. Arie, 2006. *Optimizing bitmap indices with efficient compression*, ACM Trans. Database Systems, 31: 1-38.
[4] Kaufman, K. and T. Shmuel, 2005. *Semi-lossless text compression*, Intl. J. Foundations of Computer Sci., 16: 1167-1178.
[5] Campos, A.S.E, *Basic arithmetic coding by Arturo Campos Website*, Available from: http://www.arturocampos.com/ac_arithmetic.html. (Accessed 02 February 2009)
[6] Vo Ngoc and M. Alistair, 2006. *Improved wordaligned binary compression for text indexing*, IEEE Trans. Knowledge & Data Engineering, 18: 857-861.
[7] Cormak, V. and S. Horspool, 1987. *Data compression using dynamic Markov modeling*, Comput. J., 30: 541–550.
[8] Capocelli, M., R. Giancarlo and J. Taneja, 1986. *Bounds on the redundancy of Huffman codes*, IEEE Trans. Inf. Theory, 32: 854–857.
[9] Gawthrop, J. and W. Liuping, 2005. *Data compression for estimation of the physical parameters of stable and unstable linear systems,* Automatica, 41: 1313-1321.