A Dictionary based Efficient Text Compression Technique using Replacement Strategy

Debashis Chakraborty
Assistant Professor, Department of CSE, St. Thomas' College of Engineering and Technology, Kolkata, 700023, India

Debajyoti Ghosh
Department of CSE, UG Student,St. Thomas' College of Engineering and Technology, Kolkata, 700023, India

Piyali Ganguly
Department of CSE,UG Student,St. Thomas' College of Engineering and Technology, Kolkata, 700023, India

ABSTRACT
The concept of compression comes from the need to store data using as less space as possible and to ease transfer of data through a channel. The proposed algorithm deals with compression of text files using character replacement technique. For every string of length six, it is compressed by assigning a single character to it, maintaining a dictionary. The dictionary is used to decompress the encoded file. This gives a good compression ratio irrespective of the content of the text file.

Keywords
Lossless Compression, Character Replacement, Compression Ratio, Dictionary

1. INTRODUCTION
Data compression is a technique by which the same amount of data is transmitted by using a smaller number of bits[1,2,3, 4,6]. Data compression offers an attractive approach to reducing communication costs by using available bandwidth effectively.Data compression techniques can be divided into two major classes lossy and lossless. In Lossless data compression technique bits are reduced by identifying and eliminating statistical redundancy. The main feature of lossless data compression technique is, it consists of those techniques which are able to generate an exact duplicate of the input data stream after a compress or expand cycle. Lossless compression is possible because most real world data has statistical redundancy. Lossy data compression concedes a certain loss of accuracy in exchange for greatly increased compression. Lossy compression proves effective when applied to graphics image and digitized voice. In general, data compression consists of taking a stream of symbols and transforming them into codes. If the compression is effective the resulting stream of codes will be smaller than the original symbols. The decision to output a certain code for a certain symbol or set of symbols based on a model. The model is simply a collection of data and rules used to process input symbols and determine which code to output. This is a new data compression algorithm, based on dictionary based text compression technique. The efficiency of that dictionary based text compression is, it provides good compression ratio as well as fast decompression mechanism.

In this paper the main focus is on compression of Text data by lossless compression technique. The development of a new algorithm to compress to compress and decompress text data having a high compression ratio.[3] is what this paper primarily deals with the proposed algorithm aims at producing a minimum of 70 % compression irrespective of the content and the size of the text data.

2. ILLUSTRATIONS
2.1 Algorithm Strategy
The proposed algorithm deals with the replacement of string of characters [4] by a single character, thus reducing the effective length of the string. This is done by using printable and non-printable ASCII characters. As per ASCII standard, the first 256 ASCII characters are given 1 byte of memory space. The next group of ASCII characters, ranging from 256 to 4095, is each given 2 bytes of memory space. The next group of ASCII characters, ranging from 4096 to 65535, is each given 2 bytes of memory space. The algorithm is aided by the use of a dynamic dictionary [5] which the algorithm creates. Taking a string of length six from the input file, we divide it into two parts, each of length three, and put the first part in the row and the second part in the column of the dictionary respectively. At their intersection, we place one of those printable or non-printable ASCII characters. For the next set of string of length six from the input file, we divide it into two parts again and check the existence of the substrings of length three in the row and column of the dictionary to find a match [4].If both the row and column entry doesn't match, we insert them in the dictionary and add a new symbol. If only the first substring is matched with a row entry, we insert the second substring as a column entry and add a new symbol. If the first substring doesn't match any row entry but the second substring matches an entry of the column then the first substring is entered in the row and a symbol is assigned at their intersection. If both the substrings matches a row entry and a column entry respectively then no symbol is inserted, the next set of string of length six is taken from the input file. The process continues until the end of file is reached. Now, since the character replacement is done in a fashion that it is one character against six, and instead of 6 bytes it is now either 1 or 2 bytes, on an average a minimum of 65% compression is achieved. Since it is more unlikely that the frequency of occurrence of similar 6 consecutive letters will be high, it is broken into two halves, each of length three and matching is done for better results. Also the size of the dictionary remains manageable.

For example, let us consider a string:

"Thisisa$Compression$Algorithm$1234"

For our convenience, we assume that space is denoted by $. We take the input string, read the first 6 characters "This$i" and break it into two substrings of length three each as: "Thi" and "s$i". We then form the dictionary as per the algorithm and assign a unique character at their intersection.

Table 1: Dictionary Formation (1st step)

Dictionary	s$i
Thi	Φ

Proceeding in this manner we form the dictionary until end of string is reached.

Table 2: Dictionary Formation (2nd step)

Dictionary	$Co
s$a	Ξ

Table 3: Dictionary Formation (3rd step)

Dictionary	Ess
Mpr	©

Table 4: Dictionary Formation (4th step)

Dictionary	$Al
Ion	☹

Table 5: Dictionary Formation (5th step)

Dictionary	Ith
Gor	☺

Table 6: Dictionary Formation (6th step)

Dictionary	234
m$1	□≗

After forming the dictionary, the file is compressed by character replacement technique with the help of the dictionary. After compression the compressed file will be:

Φξ©☹☺≗

Using the dictionary we can again generate the original file without any loss of information. Thus the algorithm proposed is lossless.

2.2 Algorithm

A text file is taken as input and the following compression algorithm is used on it to compress the original file.

Steps for Compression:

1. Open the text file.
2. Read the first six characters of the file.
3. Generate a dictionary with two fields, row and column and a third field, symbol, at their intersection.
4. Divide the string into two parts and place them in row and column respectively.
5. Allocate a symbol/character value to the field 'symbol'.
6. Read the next sequence of six characters from the file.
7. Divide the string into two parts and put the parts in two separate string variables, first part and second part respectively.
8. Compare first part with row and second part with column in the dictionary.
9. If an entry is found in the dictionary,
   9.1. If only row value match, make a new column entry for the dictionary and put the string in it. Assign a character to the intersection of the existing row value and new column value.
   9.2. If only column value match, make a new row entry for the dictionary and put the string in it. Assign a character to the intersection of the new row value and existing column value.
10. Repeat steps 6 to 9 until end of file is reached.
11. Store the dictionary in the compressed file.
12. Rewind to the start of the source file.
13. Read six characters from the file and place the characters in two string files first part and second part sequentially 3 characters each.
14. Compare first part with row and second part with column in the dictionary.
15. When a match is found, place symbol value in compressed file.
16. Repeat steps 13 to 15 till end of file is reached.

After the compression of the original file is done, the compressed file is decompressed using the decompression algorithm. The technique is lossless because after decompression the decompressed file will be identical to that of the original file.

Steps for Decompression:

1. Open the compressed file.
2. Reconstruct the dictionary from the stored values in the compressed file.
3. Read one character at a time from the compressed file.
4. Search the symbol contents of the dictionary for a match.
5. Once a match is obtained,
   5.1 Read the row and column value at the intersection where a symbol match has been obtained.
   5.2 Place the row and the column strings respectively in the decompressed file.
   5.3 Repeat steps 3 to 5 till end of file is reached.

2.3 Measuring Compression Performances

Performance measure [1, 4,9] determines whether a compression technique is efficient or not depending upon certain criteria. Depending on the nature of application there are various criteria to measure the performance of compression algorithms. The two most important factors are time complexity and space complexity. There is always a trade-off between the two. The compression behavior depends on the category of compression algorithm: lossy or lossless. Following are some measurements to calculate the performances of lossless algorithms.

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

$$\text{Compression Ratio} = \frac{\text{size after compression}}{\text{size before compression}} \quad (1)$$

Compression Factor: The inverse of Compression Ratio is the Compression factor. It is the ratio between the size of source file and the size of compressed file.

$$\text{Compression Factor} = \frac{\text{size before compression}}{\text{size after compression}} \quad (2)$$

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

$$\text{Saving Percentage} = \frac{(\text{size before compression} - \text{size after compression})}{\text{size before compression}} \% \quad (3)$$

Based on the above three metrics we calculate the compression performance and efficiency. The smaller the compression ratio the better would be the performance of the compression algorithm. Again for better compression algorithms, the compression factor must be high. The saving percentage [3, 5, 7, 8] gives the idea about the percentage of compression actually done. Using these parameters we measure the performance and efficiency of our proposed algorithm and compare it with other standard algorithms. We can show that for the proposed algorithm in case of worst case analysis also we get a minimum compression with saving percentage 70%. For other cases, the saving percentage may reach as high as 80% as well. That is where the proposed algorithm scores more even though there is a trade-off with the space complexity that the algorithm faces.

2.4 Calculation

For the proposed algorithms, the performance of the same for the worst case compression is analyzed here. We take the worst case analysis where no matching is found and every time a new entry is inserted in the dictionary. The analysis done is independent of the content of the file.

1st part: For first 256 symbols

Size taken up by 1st 256 bytes in the compressed file:
256*1=256 bytes
Original size of the file for those 256 symbols representation:
256 * 6=1536 bytes.
Hence, Compression ratio:
256/1536 = 0.167

Saving Percentage:
(1536-256)/1536 = 83.33%.

2nd part: For next 3840 symbols

Size taken up by symbols from256 to 4095 in the compressed file:
3840*1=3840 bytes.
Size of the symbols from256 to 4095 in the original file:
3840 *6=23040 bytes.
Thus, Compression ratio:
3840/23040 = 0.167

Saving Percentage:
(23040-3840)/23040 = 83.33%

3rd part: For next 61440 symbols

Size taken up by symbols from4096 to 65535 in the compressed file: 61440*2=122880 bytes.
Size of the symbols from 4096 to 65535 in the original file: 61440 *6=368640 bytes.
Thus, Compression ratio:
122880/368640 = 0.333

Saving Percentage:
(368640-122880)/368640 = 66.67%

For calculating the worst case scenario, we have to consider that all the entries of the dictionary have been filled, i.e. the dictionary contains all possible entries and every string combination is present in the file once. In doing so we consider the total size of the compressed file with respect to that of the original file. In such situation, the compression measure will be as follows:

Original file size:

(256*6) + (3840*6) + (61440*6) =393216 bytes.
Compressed file size:
(256*1) + (3840*1) + (61440*2) =126976 bytes.
Final Compression ratio:
126976/393216 = 0.323

Saving Percentage:
(393216-126976)/393216 = 67.7%

Thus, in the worst case analysis we can say that approximately the compression will be more than that of 65% irrespective of the content of the file. The storage of the dictionary in the compressed file will increase the size of the compressed file affecting the compression ratio for smaller files. However, in larger files, the dictionary size should become relatively insignificant.

2.5 Experimental Results

The proposed algorithm when implemented on any arbitrary file size shows compression ratio and saving factors matching the one shown in calculation theoretically. The algorithm works on different file sizes irrespective of the content of the file and yields the same results. Since the decompressed files are rebuild using the dictionary so there is no loss of information and hence the algorithm proves itself to be lossless. The figure below shows the compression ratio graphically where the plotting is done as original file size vs. the compressed file size. The ratio between the compressed file size and the original file size gives the compression ratio which can be easily depicted from the chart. The experimental

result shows the saving percentage to be above 70% always. The chart given below displays the stability of the compression ratio [3, 5] achieved irrespective of the file size. We observe that as the file size increases the saving percentage decreases. But theoretically it is shown that it is always above 70% even if worst case compression is done, which is better than or comparable with existing standard algorithms. The comparison with two of the existing standard algorithm for text compression is also done in order to compare the compression ratio and saving percentage.

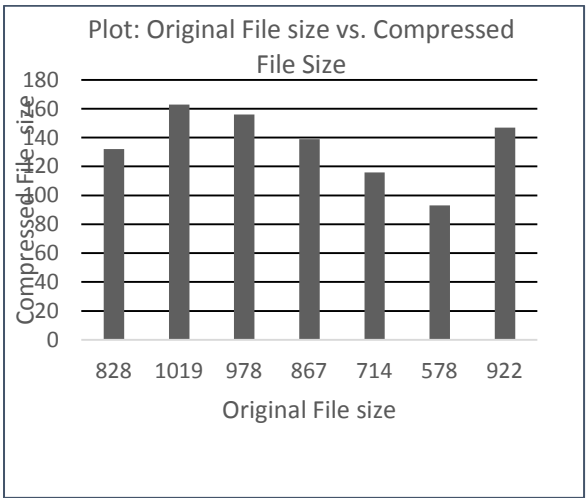

Fig.1. Graph of Original File Size vs. Compressed File Size

A brief analysis of the result of proposed algorithm shows that a stable compression ratio is achieved irrespective of the file size and the content of the file. Table 7 shows the original size and size after compression of the text files taken under consideration. It also shows a comparative study of the compression ratio of the existing standard algorithms and the proposed algorithm. The algorithms taken under consideration are Huffman Encoding and LZW encoding algorithms. [11, 12, 13, 14]

Table 7: Comparative study of algorithms

File Name	Origi-nal File Size (in KB)	Compr-essed File Size for Proposed Algo-rithm (in KB)	Compr-essed File Size for Winrar Compressi on (in KB)	Compr-essed File Size for Arithmetic Compressi on (in KB)
sample1.java	828	132	223	315
sample2.java	1019	163	265	387
sample4.doc	978	156	264	371
sample5.doc	867	139	234	321
sample6.doc	714	116	179	264
sample8.htm	578	93	156	219
sample9.htm	922	147	224	350

Table 8: Comparisons of Compression Performance

File Name	Saving Percentage of Proposed Algorithm (%)	Saving Percentage of Winrar Compression (%)	Saving Percentage of Arithmetic Compression (%)
sample1.txt	84.05	73.06	61.95
sample2.txt	84.00	73.99	62.02
sample4.doc	84.04	73.00	62.06
sample5.doc	83.96	73.01	62.97
sample6.doc	83.75	74.93	63.02
sample8.htm	83.91	73.01	62.11
sample9.htm	84.05	75.70	62.03

The Compression performance is shown in Table 8 with the help of saving percentage and compression ratio. The comparative study shows that for the input text files, the saving factor for the proposed algorithm is in the range of 84% approximately which is higher than that of two existing standard

algorithms: Huffman encoding and LZW encoding. The compression ratios can also be clearly depicted from the table above. This algorithm gives better result than winrar and arithmetic compression also. The stable compression ratio is the area where the proposed algorithm scores over the existing standard algorithms in comparison.

3. CONCLUSION

A new algorithm for text compression has been recommended in this paper, where the key element is the usage of printable and non-printable ASCII characters to replace strings of characters. Character Replacement is the main feature that is highlighted. The creation of a dynamic dictionary which comes in handy while decompressing, is another striking feature which in turns makes the algorithm a lossless one. The stable compression ratio and saving factor being more than 70% even in the worst case analysis makes this algorithm efficient than most of the existing standard algorithms. The comparative study in tabular form supports the argument. The high saving factor achieved regardless of the file size and the content of the file makes this algorithm useful. The basic purposes of data compression are to reduce the file size for storage and for transmission of the same over a channel. With the compression ratio achieved using the proposed algorithm, both the purposes can be served. With proper implementation the space-time tradeoff can also be handled effectively. The proposed algorithm has an overhead, the dictionary, but in case of lager files the overhead becomes negligible. The space complexity may be reduced by selection of proper data structure. With better implementation techniques these issues can be handled easily. However, the algorithm is recommended for text compression because of its high saving percentage, stable compression ratio and lossless nature.

4. ACKNOWLEDGEMENT

First, we would like to thank Professor Subarna Bhattacharjee, [15] for her valuable advice, provision and constant encouragement. Her constant assessments and

reviews gave us the much needed theoretical clarity. We owe a substantial lot to all the faculty members of the Department of Computer Science and Engineering and the Department of Information Technology.

We would also like to thank our friends for tolerantly listening to our explanations. Their reviews and comments were exceptionally helpful. And of course, we owe our capability to complete this project to our families whose love and encouragement consumes remained our cornerstone.

5. REFERENCES

[1] Debra A. Lelewer and Daniel S. Hirschberg, "Data Compression", Journal ACM Computing Surveys (CSUR), Vol. 19 Issue 3, Sept. 1987, pp. 261-296

[2] Khalid Sayood, "An Introduction to Data Compression", Academic Press, 1996.

[3] David Solomon, "Data Compression: The Complete Reference", Springer Publication, 2000.

[4] Mark Nelson,"The Data Compression Book".

[5] Debashis Chakraborty, Sandipan Bera, Anil Kumar Gupta and Soujit Mondal,, "Efficient Data Compression using Character Replacement through Generated Code", IEEE NCETACS 2011, Shillong, India, March 4-5,2011,pp 334.

[6] Mark Nelson and Jean-Loup Gaily, "The Data Compression Book", Second Edition, M&T Books.

[7] Gonzalo Navarro and Mathieu A Raffinot, "General Practical Approach to Pattern Matching over Ziv-Lempel Compressed Text", Proc. CPM'99, LNCS 1645, Pages14-36.

[8] M. Atallah and Y. Genin, "Pattern matching text compression: Algorithmic and empirical results", International Conference on Data Compression, vol II: pp. 349-352, Lausanne, 1996.

[9] Shrusti Porwal, Yashi Chaudhary, Jitendrajoshi, Manish Jain, "Data Compression Methodologies for Lossless Data and Comparision between Algorithms", International Journal of Engineering Science and Innovative Technology (IJESIT), Vol. 2 Issue 2, March 2013.

[10] Timothy C. Bell, "Text Compression", Prentice Hall Publishers, 1990.

[11] J.G. Cleary and I.H.Witten, "Data Compression using adaptive coding and partial string matching", IEEE Trans. Commun., Vol. COM-32, no. 4, pp. 396-402, Apr.1984.

[12] Pankaj Kumar Ankur Kumar Varshney, "Double Huffman Coding",International Journal

[13] of Advanced Research in Computer Science and Software Engineering, Volume 2, Issue 8, August 2012.

[14] Nishad PM, R.ManickaChezian, "Enhanced LZW (Lempel-Ziv-Welch) Algorithm by Binary Search with Multiple Dictionary to Reduce Time Complexity for Dictionary Creation in Encoding and Decoding", International Journal of Advanced Research in Computer Science and Engineeering.

[15] Dheemanth H N, "LZW Data Compression", American Journal of Engineering Research (AJER), Volume-03, Issue-02, pp-22-26.

[16] S. Bhattacharjee, J. Bhattacharya, U. Raghavendra, D.Saha, P. Pal Chaudhuri, "A VLSI architecture for cellular automata based parallel data compression", IEEE-2006,Bangalore, India, Jan 03-06.