Character Analysis Scheme for Compressing Text Files

Sunday Eric Adewumi

***Abstract*—This scheme considers a text document made up of character such as letters of the alphabet, punctuation marks and special characters/symbols. If we represent each character that makes up the document as $c_1, c_2, \dots, c_n$, compression is achieved by taking each of these characters that makes up the text one at a time and then search first, for the position of the last occurrence of a particular character being considered for compression together with the length of its digits, and then, starting from the beginning of the text file, note all the positions where this character has occurred. The positions of occurrence of this character while the search is on, is made equal to the length of the digit of the last occurrence of the character by padding it with zeroes to the left of the most significant bit, if need be. Concatenate the values representing the positions of the occurrence of a character and covert the concatenated string into a decimal value. Divide this value successively by 2 until the result lies between one and less than two. Store the quotient obtained from these divisions and the sum of the number of times the division was carried out as an index *k*. Decompression is the reverse of the steps just described, and this is achieved by taking each character; obtained their corresponding quotient (q), index *k* and length l_i. To recover the decimal positions of the concatenated values, we multiply the quotient (q) by 2^k. We then use the length of this particular character to identify positions where they occurred. This scheme, which is lossless compression, has its ratio tending to zero when the text file is very large.**

***Index Terms*—Compression, compression ratio, decompression, lossless, scheme, text file.**

I. Introduction

One primary objective of a data compression scheme is to achieve better compression ratio which is derived as $\frac{compressed\ size}{uncompressed\ size}$. When compression ratio moves towards zero the space saved will move towards 100%. It then means that the smaller the ratio, the better the compression scheme and the better the space saved. This scheme being proposed is an extension of our earlier schemes described in [1]-[3]. In [1] we use the product and sum in order to achieve compression and decompression. This scheme, represented by table I assumed that documents are made of letters of the alphabets, punctuation marks and special characters. In this scheme, characters were represented by l_1, l_2,..,l_n. Each character was then analyzed into form in table I. In this table, l_i represents the character being compressed; *k* represents the length of character under consideration which also symbolizes the number of times the character occurred in the text (document). $\prod_{i=1}^{k(l_i)} \alpha_i$ is the product of the occurrences of this character. The occurrences are taken as the factors of the product. They are represented by α_i the positional location where l_i occurred and $\alpha_{k(li)}$ called the upper bound (U), representing the last value where the position l_i occurred. Each row of the compression table I represent each character of the document being compressed. The series α_i (the occurrence of a letter or character l_i) are taken as subset of the factors of the products *P* denoted by $\prod_{i=1}^{k(l_i)} \alpha_i$, while $\sum_{i=1}^{k(l_i)} \alpha_i$ represents the sum (S) of the factors. The compression table takes each character, defines its length of occurrence, the sum of these occurrences, the product and U which is the last position where l_i occurred. The occurrences of each letter are arranged progressively in ascending order of magnitude. The occurrence when taken as the factors of a number *N*, their sum is taken as S. In this method high compression is achieved with documents with large string of characters. However, decompression algorithm could not be easily achieved because there was the need to find factors of number to be able to identify the positions where a particular character appeared in a text file. This proved difficult as factoring a large number could be prohibitive. Table I is a sample of compression from this first scheme.

TABLE I: First Compression Scheme

l_i	k	P=$\prod_{i=1}^{k(l_i)} \alpha_i$	S=$\sum_{i=1}^{k(l_i)} \alpha_i$	U=$\alpha_{k(l_i)}$

In our second scheme described in [2], a text file is compressed into a table with 3 columns (Table II). In this scheme, positions of occurrence of a character is converted into binary digits (bits), these bits position are concatenated and converted to a decimal number. The length of the last position of occurrence of a particular character is preserved for use during decompression. Specifically, compression is achieved in the following steps:

- Take each letter l_i of alphabets (characters) that makes up the text document;
- Find the positions where each of these letters occurs
- Covert each position to a binary number.
- The binary string length *k* representing the last position of occurrence is used as the standard length for each binary string. This means that if other positions are not *k*-length when converted, it has to be padded on the left to make it *k*-length compliant.
- Concatenate the binary strings for each alphabet (character); this is in turn converted to decimal number to complete the compression.
- Store the length *k* representing the binary string of the last occurrence of a particular alphabet (character) for use during the decompression.

Manuscript received April 18, 2014; revised August 5, 2014.

Sunday Eric Adewumi is with the Federal University Lokoja, Nigeria (e-mail: sunday.adewumi@fulokoja.edu.ng).

However, it was observed that even though compression was achieved, the amount of bit strings involved gave us reasons to continue to search for better schemes.

TABLE II: SECOND COMPRESSION ALGORITHM

l_i	d = decimal number equivalent of kj (the length of binary string)	k (the length representing the binary string of the last occurrence of a particular alphabet)

In our third scheme as shown in [3], a table of six columns is used to achieve compression. Table III is a representation of this scheme. The scheme starts by obtaining the ASCII code of each character being compressed. The first column contains the converted ASCII value to decimal; the next column represents the length of binary string for each character, their decimal equivalent, length of binary string for each repetition of each character, decimal values for the positions and length of binary string for each position. In summary, compression is made up the following:

- One decimal value representing the ASCII codes that makes up the document
- The length in binary of each of the characters that makes up the document
- The decimal number representing the positions of all the characters that make up the document
- Decimal value representing the number of time each character occurred in the text
- The binary length for the positions of each character
- The binary length for each character that makes up the document.

TABLE III: THIRD COMPRESSION

Decimal values for the character texts that make up the document	Length of binary string for each character	Decimal value for the number of times a character repeats	Length of binary string for each repetition	Decimal values for all the positions	Length of binary string for each position

We noted again that lots of strings are involved and these might not make for the compression we really desire; hence the new algorithm, which we hope will provide better compression ratio. This is described in the methodology section of this paper.

Various schemes have been devised to reduce the compression ratio of text files. A study of different methods of data compression algorithms on English text files were carried out by [4]; namely: LZW, Huffman, Fixed-length code (FLC), and Huffman after using Fixed-length code (HFLC). They evaluated and tested these algorithms on different text files of different sizes and made a comparison in terms of compression: Size, Ratio, Time (Speed), and Entropy. At the end they found that LZW is the best algorithm in all of the compression scales that were tested.

In their paper, [5] presented a new lossless text compression technique which utilizes syllable-based morphology of multi-syllabic languages. The proposed algorithm is designed to partition words into its syllables and then to produce their shorter bit representations for compression. The method which has six main components namely source file, filtering unit, syllable unit, compression unit, dictionary file and target file. The number of bits in coding syllables depends on the number of entries in the dictionary file. They concluded by noting that the compression rates were observed to change from 13.0% to 43.2%. Their experiences indicated that higher compression rates were achieved with increasing text sizes. A new optimization technique has been proposed in [6]. In this paper, they proposed a new algorithm for data compression, called j-bit encoding (JBE). This algorithm is designed to manipulate each bit of data inside a file to minimize the size without losing any data after decoding, classified as lossless compression. The algorithm was intended to be combined with other data compression algorithms to optimize the compression ratio.

A compression techniques based on two level approaches was proposed by [7]. In this paper they proposed two levels: first reduction and second compression. Reduction is accomplished using a word lookup table and not the traditional indexing system, and then compression was done using currently available compression methods. The lookup table would be a part of the operating system and the reduction would be done also by the operating system. They concluded by saying the method that uses word lookup table will make any text segment able to use lesser memory space.

Ref. [8] uses a two stage method to achieve this. In their proposal, they combined both LZW and Huffman algorithm and discovered compression reduced by 5%.

In Ref. [9], a survey of dictionary based preprocessing methods in text compression was carried out and the authors concluded that it provided 2% more compression ratio.

II. METHODOLOGY

In this new technique being proposed, compression of a text file when completed produces a table with the following entries: the list of characters that makes up the text file, the quotient (index k) that emerges after the successive division by 2, the decimal number representing the concatenated values of the positions of an alphabet and the length of the last occurrence of a particular character within the text will emerge. During decompression, each alphabet that makes up the text is taken one at a time and their positions recovered by multiplying the quotient obtained for that character by 2^k. The resulting value represents the positions where a particular character occurred in the text. We then use the length to identify these positions of occurrence and subsequently write the alphabet into their positions one after the other until all the characters have been written. When a text is compressed by this scheme, the compression table is likely to come to a page or two, no matter how large the text file may be. This would happen because compression is achieved through the analysis of the alphabets that makes up the text in addition to all special characters that also appeared in the text. The compression ratio, which is derived by $\frac{\text{compressed size}}{u\text{ ncompressed size}}$ tends to zero when the text file is large. The scheme leads to a lossless compression.

Compression is achieved by doing the following analysis:

- Take each characters that makes up the text file
- Find all the positions the character occurred in the text and concatenate these positions into one decimal value
- Find the Quotient (*q*) of successive division by 2 until the result is less than 2 but greater than one
- Store the index (*k*) which is the sum of the number of times division by 2 was done on the decimal value (d_i)
- Store the number of times each character occurred.

TABLE IV: COMPRESSION ANALYSIS

Character c_i	Decimal value of the concatenated position of a character d_i	Quotient (*q*) of successive division by 2 until the result is less than 2 but greater than one *q*	index (*k*) which is the sum of the number of times the value was used to divide decimal (d_i) *k*	Length of the occurrence of each character l_i

After the character analysis in Table IV, the actual compression table would Table V.

TABLE V: THE ACTUAL COMPRESSION

Character` c_i	Quotient (*q*) of successive division by 2 until the result is less than 2 but greater than one *q*	index (*k*) which is the sum of the number of times the value was used to divide decimal (d_i) *k*	Length of the occurrence of each character l_i

Decompression is represented in Table VI and it is achieved by doing the following:

- Take each compressed character (c_i)
- Find the decimal value of the positions of c_i by multiplying the quotient (*q*) with 2^k where *k* represents the sum of the number of division
- Use the length (l_i) to the positions where the characters have occurred in the text.

TABLE VI: DECOMPRESSION

Character c_i	Find the decimal value of the concatenated position of a character d_i	Identify the length of the occurrence of each character l_i

III. RESULT/EXAMPLES

We use an example to demonstrate how a compression and decompression is achieved using this scheme. Assuming we wish to compress a text such as Lokoja and Abuja. Table VII is text and the positions of the characters that make up the text. To compress this text, we carry out an analysis of the characters that make up the text as follows:

TABLE VII: POSITIONS OF A TEXT TO BE COMPRESSED

L	o	K	O	j	A		a	n	D		A	b	U	J	a
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

We now put this text in form of Table IV to produce the compression table (Table VIII) as follows:

TABLE VIII: COMPRESSION ANALYSIS

Character c_i	Decimal value of the concatenated position of a character d_i	Quotient (*q*) of successive division by 2 until the result is less than 2 but greater than one *Q*	index (*k*) which is the sum of the number of times the value was used to divide decimal (d_i) *k*	Length of the occurrence of each character l_i
A	06081216	1.4498748779296875	22	2
B	13	1.625	3	2
D	10	1.25	3	2
J	0515	1.005859375	9	2
K	3	1.5	1	1
L	1	0.5	1	1
N	9	1.125	3	1
O	24	1.5	4	1
U	14	1.75	3	2
Space	0711	1.388671875	9	2

The actual compression will be represented by the Table IX.

TABLE IX: ACTUAL COMPRESSION

Character c_i	Quotient (*q*) of successive division by 2 until the result is less than 2 but greater than one *Q*	index (*k*) which is the sum of the number of times the value was used to divide decimal (d_i) *k*	Length of the occurrence of each character l_i
A	1.4498748779296875	22	2
B	1.625	3	2
D	1.25	3	2
J	1.005859375	9	2
K	1.5	1	1
L	0.5	1	1
N	1.125	3	1
O	1.5	4	1
U	1.75	3	2
Space	1.388671875	9	2

IV. DECOMPRESSION

Decompression is the process of reversing what we have in Table IX in order to recover the positions of the characters indicated in the first column. This is achieved in Table X.

In Table X, the characters that make up the text together with their positions have been recovered. The length of occurrence for each character is then used to identify positions where they have occurred and can then be written.

TABLE X: DECOMPRESSION

Character c_i	Decimal value of the concatenated position of a character d_i	Length of the occurrence of each character l_i
A	$1.4498748779296875 \times 2^{22} = 06081216$	2
B	13 = 06081216	2
D	$1.25 \times 2^3 = 10$	2
J	$1.005859375 \times 2^9 = 0515$	2
K	$1.5 \times 2^1 = 3$	1
L	$0.5 \times 2^1 = 1$	1
N	$1.125 \times 2^3 = 9$	1
O	$1.5 \times 2^4 = 24$	1
U	$1.75 \times 2^3 = 14$	2
Space	$1.388671875 \times 2^9 = 0711$	2

V. CONCLUSION

In this paper, we have shown that text data can be compressed by taking each character that form a text file, find positions they occurred and concatenate them into one decimal value, then divide the value by 2 until the quotient of that division becomes less than 2 but greater than one. While doing this, we store the length of the last occurrence of a character together with the number of times the division by 2 was done as index. We have also used a simple example to demonstrate how compression and decompression could be achieved by the scheme. We have shown that compression could be achieved by analyzing the alphabets that makes up the text to be compressed. An English text for example, will essentially be made up of letters of the aA – zZ (26 letters) and some other special characters. However large the text may be, it might not be more than 2 to 3 pages when compressed by analyzing the characters. A dictionary, for example, might not be more than 3 pages when compressed with this method.

REFERENCES

[1] S. E. Adewumi and E. J. D Garba, “A new text compression algorithm,” *European Journal of Scientific Research,* AMS Publishing, Inc., vol. 8, no. 3, pp. 6-14, 2005.

[2] S. E. Adewumi and E. J. D Garba, “Text compression algorithm: A new approach,” *Journal of Institute of Mathematics and Computer Science,* vol. 19, no. 1, pp. 63-68, 2008.

[3] S. E. Adewumi, “An optimal scheme for compressing text documents,” *Nigerian Journal of Pure and Applied Sciences,* vol. 5, no. 1, 2012.

[4] H. Altarawneh and M. Altarawneh, “Data compression techniques on text files: A comparison study,” *International Journal of Computer Applications,* vol. 26, no. 5, pp. 42-54, July 2011.

[5] I. Akman, H. Bayindir, S. Ozleme, Z. Akin, and S. Misra, “Lossless text compression technique using syllable based morphology,” *The International Arab Journal of Information Technology*, vol. 8, no. 1, pp. 66-74, January 2011.

[6] M. A. D. Suarjaya, “A new algorithm for data compression optimization,” *International Journal of Advanced Computer Science and Applications,* vol. 3, no. 8, pp. 14-17, 2012.

[7] M. A. K. Azad, R. Sharmeen, S. Ahmad, and S. M. Kamruzzaman, “An efficient technique for text compression,” in *Proc. The 1st International Conference on Information Management and Business*, 2005, pp. 467-473.

[8] P. Raja and D. Saraswathi, “An effective two stage text compression anddecompression technique for data communication,” *International Journal of Electronics and Communication Engineering,* vol. 4, no. 2, pp. 233-241, 2011.

[9] S. J. Rexline and L. Robert, “Dictionary based preprocessing methods in text compression — A survey,” *International Journal of Wisdom Based Computing,* vol. 1, no. 2, pp. 13-18, August 2011.

Sunday Eric Adewumi earned a B.Sc. degree in computer science from the University of Ibadan, Nigeria in 1983; M.Sc. degree in mathematics from the University of Jos, Nigeria in 2000 and Ph.D. degree in mathematics from Abubakar Tafawa Balewa University, Bauchi, Nigeria in 2005. He is currently a professor and the head of the Department of Computer Science, Federal University Lokoja.

His areas of interests are in data compression and data security (using systems of equations to encrypt messages). His experience in software development spans over three decades.

Professor Adewumi is a fellow of Nigeria Computer Society (FNCS) and a member of Computer Professional Registration Council of Nigeria (CPN).