Non-Linear Data Structure for Data Coding for Size Compression

Ashwan A. Abdulmunem¹, Ahmed K. Hassan² and Dunya J. Mohammed¹

¹College of Science, University of Kerbala, Iraq, ²College of Engineering, University of Kerbala, Iraq

Abstract. Data compression is a special case of communication system which is a basic problem in engineering and computer science. In this work, the new scheme has been introduced for data coding to compress the size. Compression by using one type of Non-linear Data-Structure which is Prefix Binary Tree data compression is essential and beneficial in the diversity of application which is embraced in general into two branches: the storage and transmission. The storage of a file in a compressed form takes fewer bit than in uncompressed form, therefore make the most of storage resources (containing main memory itself), and by transmission means sending the files over a channel with reduced number of bits by compression process. This leads to more efficacious transmission process. The compression process of stored data or that to be transmitted leads to significantly reducing both of the storage and transmission. We use text type as a data in the experiments. The results show the compression rate is reduce the actual size above 50%.

1. Introduction

Data compression is an encoding information algorithm in several ways which aims to data takes up less space than the previous state. Removing redundancy, that is, removing the unnecessary bits which taking up from actual data representation, to acquire the true number of bits that are would be predicted by entropy. So that, the need for the information to measure the extent of the suggested method in the data compression. This metric called compression ratio, it is calculated by dividing the volume of compressed data by volume of original data subtracted from 1 and it is usually expressed as a percentage [1,2].

Data compression is usually indicated to as a type of coding, since coding is a common term to the representation of data in a specific way. Which serves a specific purpose. Information theory is defined as the study of the effectiveness of coding and its outcome in the form of transmission speed and error probability[3,4,5]. Data compression is sometimes seen as a field of information theory as the main goal is to reduce the data to be sent across the channel.

The process of data compression can be described in a simple way, as it works on converting a series of characters(string) that is represented in several ways (such as ASCII) into a new series (of bits, for example). Which contains fewer bits while maintaining the information without changing, new series of character with shorter length as possible. Data compression has substantial applications in the domain of the transmission of data and its storage. The majority of data processing application need storage of a large amount of data, and the number of these applications increases constantly with the expansion of the
utilization of computers to a new specialization. At the same time the spreading of communication networks leading to the outcome of transmission huge amount of data over channels.

Compressing data sent or stored leads to a reduction in the cost of storage and transmission [5,6,7,8,9]. When decreasing the amount of transmitted data, as a result increasing in the communication capacity to transfer more data. Also the compression of file to half of its size is the same way as doubling up the medium capacity of storage.

It perhaps then become convenient to store the data at a higher hierarchical level, subsequently faster, level of the storage hierarchy and decrease the load over the computer system's input/ output[6,7].

2. Dynamic Information Structures
The distinctive characteristic which distinguishes the Recursive Data Types from the fundamental structures (arrays, records, sets) is their capability to changing in its size. Thus it is impossible to set a fixed storage size to the recursively defined structure and it results in the compiler cannot link specific addresses to the components of those variables. The majority used technique for solving this problem includes an assignment of storage to these components whenever data appears during the program execution rather than assign storage during translation time. After that, the compiler assigns a bounded amount of storage to store the address of this component (dynamically allocated) rather than storing itself. For example, the pedigree illustrated in figure 1 would be represented by singular records, one for each person. These persons thereafter are then connected using their addresses associated with the particular father and son fields. Graphically, this state is clearly illustrated by the usage of pointers or arrows [4].

![Figure 1: Pointers in Data Structure](image)

3. Prefix Codes and Prefix Trees
There are codes with a property of the self-punctuating. The prefix tree that resulted in what called prefix code which can be recognized by checking its what's called the characteristic of prefix tree or prefix property for shortened.

The first few consecutive bits in a codeword that so-called prefix. It is possible that the shorter codeword conformable to the first few bits of the longer codeword, if they are two different lengths. In this status is said the shorter codeword to be the prefix of the longer codeword.
For instance, c1=010 (3 bits) and c2=01011 (5 bits) they are two binary code of different length, in this case as mentioned above c1 is the prefix for longer codeword c2 (01011), and the codeword of c2 can be obtained by adding two more bits 11 to c1. On the other hand, in code (0, 10, 010, 101), which is not uniquely decodable, codeword 0 is considered the prefix of codeword 010. Likewise, codeword 10 is the prefix of codeword 101. The prefix characteristic has turned into a favorite feature when seeking for a uniquely decodable code. A code with a prefix characteristic is called a prefix code. Simply put, Prefix code is a code in which no codeword is a prefix for another codeword, nor can a codeword be derived from another codeword by adding more bits to a shorter codeword.

As well as the code (1, 01, 001, 0000) is considered as a prefix code since no codeword is a prefix of another codeword in that code. On the contrary, the code (0, 10, 110, 1011) is not considered as a prefix code because 10 is a prefix of 1011. It is easy to identify the binary code if it is a prefix code or not by drawing an associated binary tree. Every binary code can identical to one such binary tree, in which each codeword identicals to a path from the root to a node with the codeword name specified at the end of the path. Every value of the bit equal to 0 in a codeword matches to a left edge and each value of the bit equal to 1 matches to the right edge. Remember that, if a prefix code is represented in such an associate binary tree, at its leaves all the codeword's labels [2, 4, 6].

4. The Proposed method

There are stages of the implementation of the text compression and decompression represented by using the prefix code.

The compression algorithm take input as a string of text, and its output a binary bits string to interpret the string which inputted. While decompression algorithm's input is a bitstream. The algorithm's output is a series of characters (string). The process of compression contains three subprocesses: read the input string, and then interpret every input symbol and produce the codeword as output for every input symbol, while the process of decompression contains the next steps: read the input codeword, interpret the codeword and Output the string of characters. The algorithm of reading the text file and counting the repetition of each character as follows:

| Step1: | Set an array of records with size 93 records, each record contains two fields, the first field contain the character and the second for the repetition for each character. |
|--------|--------------------------------------------------------------------------------------------------|
| Step2: | Open the source file.                                                                               |
| Step3: | While not end of file                                                                               |
| a)     | Read character from the file.                                                                       |
| b)     | Set flag for checking if the character is already existing in the file or not.                      |
| c)     | If flag=0 then // (the character not exist in an array) add the new character with increment the count of repetition by 1 , set flag to 1  
        | Else // (the character exists in an array)  
        | The character already exists only increment count of repetition by 1                                |
| Step4: | End.                                                                                               |

After array creation, the array will contain the repetition of each character, the priority queue will be built of the index of a character in the array of printable characters and its repetition. The following algorithm represents the steps of building this priority queue:
To build the binary tree two approaches are used: 'bottom-up' approach to begin from the leaves to the root, This technique is used in prefix encoding. The second approach called 'top-down ' to construct the tree from the root to the leaves and this approach used in Shannon-Fano encoding; in our project, we used prefix encoding. After we created the priority queue from the previous algorithm, we will use it to build the prefix tree as follows:

5. Building prefix code for a character

This algorithm depends on the prefix tree that created from the above algorithm to generate the codeword of each character:

For each element in the array of repetition do the following steps:

**Step1:** If the repetition not equal to zero then

**Step2:** Insert the vector of two locations the index of repetition and the repetition , in the priority queue with keeping the ascending order of repetition.

**Step3:** End.

**Step1:** While there is more than one vector in the priority queue do the following steps:

**Step2:** Remove the two first vectors from the priority queue (lowest repetitions in priority queue). Take the repetition (second location of vector) which it’s become the data of leaf node of prefix tree.

**Step3:** Sum of lowest character repetition which resulted from above step to become data of parent’s node.

**Step4:** End.
Depending on the prefix tree created from the above algorithm, the decoded characters of the original file will be implemented by using the underneath algorithm.

**Step1:** while not end of compressed file do following steps:

**Step2:** read bit from the compressed file.

**Step3:** begin from the root of the tree, we traverse the undermost of tree one edge to a child depending on the current read of bit from codeword. If the value of the bit is equal to 0 then

- traverse to the left child,

else

- go to the right child.

**Step4:** if the current node is leaf node then

- a. The character is decoded.
- b. restart the traversal to the root.

**Step5:** end.

### 6. Experimental results

Below the outcome results after applying the above algorithms of using prefix tree in data compression and decompression with an example: The input string is "aabachcd."

#### 6.1 Compression process

1. In the first place, calculate the frequency histogram of the input file:

| Symbol | A | B | c | D |
|--------|---|---|---|---|
| Repetition | 3 | 2 | 2 | 1 |

2. For each symbol, create a vector with only one node that is the symbol. Put the vectors in preference with the recurrence of symbols being the keys.

```
  d 1
  c 2
  b 2
  a 3
```

minimum element

3. If the priority queue comprises more than one vector, then take out two minimum key vectors, merge them into a single vector (adding a new parent) with recurrence equal to the sum of the original recurrence.

4. Once the tree is built, code the input by keeping track of the path from the root for each symbol. The generate 0 each time and follow the left section and 1 to indicate the right section.

The input is compressed to:
5. Divide the bitstream into 8-bit chunks obtaining:

\[
\begin{array}{cccc}
0 & 0 & 10 & 0 \\
0 & 0 & 11 & 1 \\
0 & 0 & 11 & 10 \\
\end{array}
\]

0 a b a c b c d

By dividing into 8-bit chunks, the bitstream will be as 00100111 10111110

The input line: 8 bytes. Compressed to 2 bytes.

6.2. Decompression process

The decompression depends on the built prefix tree from above steps:

Read the coded string (bitstream) one bit at a time. Initiate with the root, cross underneath of tree one edge to a son depending on the current value of the bit. In case the bit's value equal to 0, step toward the left child, otherwise, step toward the right child. When reaching the leaf, the algorithm should stop repeat the above process. Wherever arrive at the leaf, the decoding process will begin to decode one character and then iterate the traversal from the root. Reiterate these steps until the finish the string. The input bitstream took it as decompressed:

The results can be summarized in the figure below:

| No. | The text                          | Text length | Size of the original file | Size of compressed file | Compression ratio |
|-----|-----------------------------------|-------------|---------------------------|-------------------------|------------------|
| 1   | Bism Allah ala ala ala bism       | 24          | 216 bits                  | 89 bits                 | 59.79%           |
| 2   | aaaa bbb bbb cccccccc ddddddd     | 24          | 216 bits                  | 65 bits                 | 70%              |
| 3   | Someone In loving you Caring for you Watching you Protecting you Guess who? A*L*L*A*H | 85          | 680 bits                  | 380 bits                | 44.1%            |
| 4   | aaaaaa bb bbbbb bbb cocc ccc ccc dddddd ddd eeeeee see nnnn nnnn | 85          | 680 bits                  | 255 bits                | 62.5%            |

Figure 2: Results of proposed method on different texts

7. Conclusions

In this section, some conclusions will be presented. Binary Search Trees (prefix trees) have operations that are favorably fast: insertion, lookup, and deletion can all be done in \(O(\log n)\) time. The text after decompression is exactly similar to the original text, so the idea concluded that the compression from lossless
type. The text that has a large number of similar characters has a higher compression ratio than the compression ratio of text that has a large number of different characters. The suggested method gives a compression ratio about over than 50%. That makes the method more reliable to use for coding to reduce the actual text size.

8. Acknowledgement

We would like to thank University of Kerbala for supporting our work.

9. References

[1] M. Nelson and J. Gailly, 2006, “The Data Compression Book ” pp. 1–12.
[2] P. Yellamma and N. Challa, 2012, “Performance Analysis Of Different Data Compression Techniques On Text File,” Int. J. Eng. Res. Technol., vol. 1, no. 8, pp. 1–6.
[3] M. Nelson, 2003, “The Data Compression Book,” Books.Google.Com, pp. 1–370.
[4] K. A. Ramya and M. Pushpa, 2016, “A Survey on Lossless and Lossy Data Compression Methods,” Int. J. Comput. Sci. Eng. Commun., vol. 4, no. 1, pp. 1277–1280.
[5] S. Porwal, Y. Chaudhary, J. Joshi, and M. Jain, 2013, “Data compression methodologies for lossless data and comparison between algorithms,” Int. J. Eng. Sci. Innov. Technol., vol. 2, no. 2, pp. 142–147.
[6] C. Chung, 2014, “The Basic Principles of Data Compression”.
[7] Frost and Sullivan Inc, 1983 , “Data compression expands,” Displays, vol. 4, no. 2, p. 121.
[8] Z. Rahman , 2004 , “Data compression,” Comput. Sci. Handbook, Second Ed., no. October, pp. 76-1-76–40.
[9] Hussein Mogahed, Alexey Yakunin, Larisa Suchkova , 2016, “A Comparison of Data Compression Methods for Solving Problems of Temperature Monitoring”, MATEC Web Conf. 79 01076, DOI: 10.1051/matecconf/20167901076.