ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 5, Issue 1, pp: (22-24), Month: January - March 2017, Available at: www.researchpublish.com

Huffman Algorithm Improvement

[1]Chiranjeev Singh, [2]Apratim Gupta

Division of Computer Science, Netaji Subhas Institute of Technology, New Delhi

Abstract: **The purpose of this paper is to propose an algorithm which is an improvement over the Huffman Algorithm. The proposed algorithm works in a similar manner as the Huffman algorithm, here also we use a trie to generate a "CODEWORD" for each unique symbol, depending upon the frequency of occurrence of that particular symbol. Each codeword assigned is unique, i.e. no codeword or a part of the codeword is the prefix of any other, thus preventing ambiguity. We can easily identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In the improvement part, we use a marker whose codeword is unambiguous and unique, and this marker is used to encode the uppercase English characters using the respective lowercase or vice- versa.**

Keywords: **Huffman Algorithm, codeword, marker, binary encoding, decoding, Huffman trie.**

I. INTRODUCTION

Data Compression is one of the important algorithms used in the present world. They compress any form of data into a suitable form which can be stored in comparatively less space .This not only saves space, but it also saves time and resources which are used to transfer the data from one point to another. This research paper deals with a particular type of Data Compression technique, Huffman Algorithm. This paper comprises of a technique which can be used to compress a given text of data into binary codes, which can be regenerated using the decoding algorithm. The compression factor of the improved algorithm is comparatively greater than that of the Huffman's.

This algorithm can be used to compress data even further and can be used with the normal Huffman code with some minor modifications. Time consumption of the improved version is approximately same as that of Huffman's, but at the same time compressed data will occupy less space and can be easily transferred.

II. HUFFMAN IMPROVEMENT ALGORITHM DESCRIPTION

We have improved upon the Huffman Algorithm by exploiting one simple property of Huffman encoding, which is that characters that occur less frequently are longer in length, implying that whenever they do occur, they take up a lot of space. In English language, the set of capital letters is one such set which because of its relative infrequent occurrence generally occupies the bottom of the Huffman trie.

Also in fixed length encoding scheme, 8 bits are required to represent all the given characters. If, in a situation, we are required to represent all the characters using Huffman encoding, and when each has a different frequency, then many characters would require a bit length greater than 8 since no codeword can be the prefix of another.

Now to minimize the bits used by the capital letters, we first find the codeword of the most frequent character, then concatenating 1's and 0's to the beginning of the most frequent codeword, and after each concatenation, checking that the resultant code is not a prefix of some other character, and that no other character is a prefix of this code. If both conditions are satisfied, we end the concatenation, else we continue adding 1's and 0's to the bit representation of the most frequent character. 0's and 1's are added to the front of the codeword of the most frequent character in a manner such that all possible combinations of 0 and 1 are checked and no combination is left behind. Total combinations possible for a marker generated with 'N' additional bits in addition to the codeword of the most frequent character are 2^N .

Finally when we have obtained the code from the above method, we use that as the marker. We obtain the Huffman codes for all the corresponding characters. For the improvement to be beneficial and at the same time to be predictable, we add a condition -"If the bit length of the uppercase letter is greater than the bit length of the lowercase letter by the length of the marker or if the difference between uppercase and lowercase is even greater than the length of the marker, the symbol for that particular capital letter at that particular position becomes the symbol of the small letter with the marker appended as a suffix, else it remains unchanged. In a scenario where the marker's bit length is very long, the encoding is not improved, and thus the Huffman becomes the worst case of the improved algorithm.

This algorithm works in every scenario, and under no situation can do worse than Huffman Algorithm. Since, the length of the marker can exceed 8, thus even a marker of significant length can improve over Huffman.

The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the simple path from the root to that character, where, just for instance, 0 means "go to the left child" and 1 means "go to the right child". Another way is to use hash map and starting with one bit, look for any character with that particular binary representation, if it is not there, then increment the bit length by unity and continue. This method would work since no code is a prefix of the other. If the codeword of a small character is encountered, the decoding algorithm will check whether the codeword ahead of the present codeword corresponds to the marker or not. If it is the marker, then the lowercase letter will be replaced by the uppercase one, otherwise no change will be there.

The above algorithm can also be used in case there are more of uppercase letters than the lowercase letters. Only the condition where the length of codewords for uppercase letters and lowercase letters is compared will be changed in sign.

III. ALGORITHM AND ANALYSIS

STEP 1: The input text file is read and the frequency of each character is stored in an array, ARRAY_FREQ.

STEP 2: Let Y be the codeword of the most frequent character.

STEP 3: WHILE (TRUE)

```
Case 1: Y ← Z
        "1" + Y ← Y
          IF PREFIX (Y) IS FALSE
            Y is our MARKER
          ELSE   Y ← W
            Goto Case 2 with Z
            IF PREFIX (Z) IS TRUE
                 CONTINUE
            ELSE   Z is our MARKER
Case 2: Y ← Z
        "0" + Y ← Y
          IF PREFIX (Y) IS FALSE
            Y is our MARKER
          ELSE   Y ← W
            Goto Case 1 with Z
            IF PREFIX (Z) IS TRUE
                 CONTINUE
            ELSE   Z is our MARKER
```

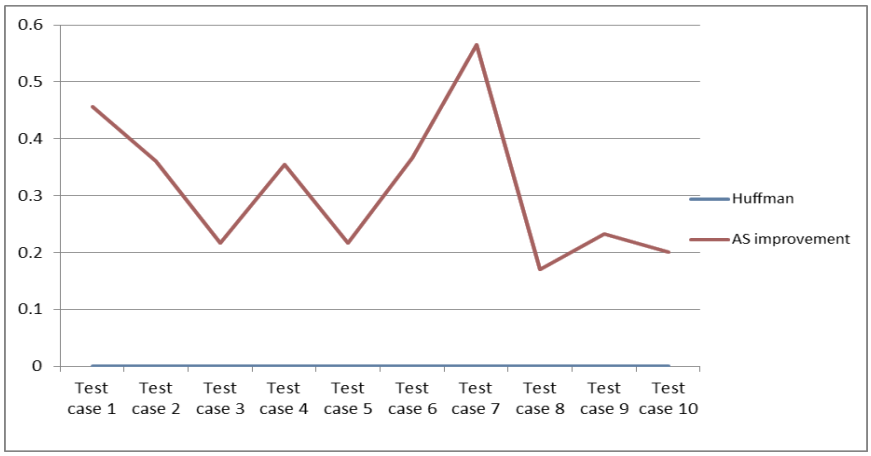
// It is not necessary that case one will be executed first. The program will check for the smallest possible string to be added to the front of the codeword of the most frequent character.

PREFIX ( ) FUNCTION

It is a function which checks whether the modified marker that we obtain by adding '0' and '1' to the beginning of the initial marker is prefix of the codeword of any other character and it also checks whether the codeword of any character is the prefix of the marker.

This function is used as it leads to the production of "PREFIX FREE CODES" . Prefix free codes are desirable because the simplify decoding. Since no codeword is a prefix of any other, the marker generated is unambiguous. We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process for the rest of the binary file.

IMPROVEMENT ALGORITHM ANALYSIS GRAPH



The following graph shows the comparison of Huffman algorithm and the Improvement algorithm.

The Y-axis represent the percentage efficiency of the improvement algorithm relative to Huffman's. On the

X-axis are the test cases whose references are provided below.

Test Cases References:

1) http://gutenberg.net.au/ebooks/m00017.txt

2) http://gutenberg.net.au/ebooks04/0400351.txt

3) http://gutenberg.net.au/ebooks06/0600181.txt

4) http://gutenberg.net.au/ebooks04/0400291.txt

5) http://gutenberg.net.au/ebooks14/1401701.txt

6) https://www.gnu.org/software/octave/octave.pdf

7) https://www.math.ucdavis.edu/~linear/linear-guest.pdf

8) http://eng.uok.ac.ir/daneshfar/IntroductionToFormalLanguages/Peter%20LinzBook%205ed/An %20Introduction%20to%20Formal%20Languages%20and%20Automata%20-%205th%20Edition%20-%202011.pdf

9) https://ocw.mit.edu/ans7870/9/9.00SC/MIT9_00SCF11_text.pdf

10) http://www.gasl.org/refbib/Bible_King_James_Version.pdf

IV. CONCLUSION

Data Compression is considered as one of the most important techniques in the present world. There are many ways to compress data and make it a bit more portable, and one of the ways is discussed in this paper. The above algorithm not only posed as an improvement over Huffman Algorithm, but also paved a way for various other improvements in the sector of data compression. As the technology is advancing, so are the ways to handle it.

REFERENCES

[1] Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

[2] Handbook of Data Compression by David Salomon and Giovanni Motta.