ASSIGNMENT – 3 REPORT

On

A Fast-Adaptive Huffman Coding Algorithm

By

SHRIYA CHOUDHARY - 2017AAPS0409H

DIGVIJAY SINGH- 2017AAPS0317H

SURABHI TOSHNIWAL - 2017AAPS0438H

Under the supervision of

PROF. RUNA KUMARI

Submitted in partial fulfilment of the requirements of

ECE F344: INFORMATION THEORY AND CODING

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI (RAJASTHAN)

HYDERABAD CAMPUS

(2nd Semester 2019-20)

INTRODUCTION

Objective: The key aim of this assignment is to implement the Fast Adaptive Huffman Algorithm using a suitable programming language, such that the program takes an input word, and gives the Fast Adaptive Huffman Code as output, as illustrated in the research paper.

Huffman coding was developed in 1952 by David Huffman. He developed the method during his Sc. D. study if MIT and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes ". Huffman coding is the most optimal among methods for encoding symbols separately, but it is not always optimal compared to some other compression methods, such as Arithmetic and Lempel-Ziv-Welch coding. However, the last two methods, as it has been said, are patent-covered, so developers often tend to use Huffman Coding. Comparative simplicity and high speed due to lack of arithmetic calculations are the advantages of this method as well. Due to these reasons Huffman Coding is often used for developing encoding engines for many applications in various areas. Huffman coding is an algorithm for data compression. It is a scheme to encode a data source driven by a known probability distribution. But practically it has two problems. First, prior knowledge of the probability distribution of the data is necessary which is not possible in most cases. Also, the statistics may change with time for some of the cases. This problem can be resolved by a modified version of Huffman coding which is adaptive coding. It does not use a fixed model. Instead, both the sender and receiver maintain their own symbol count. Each time a symbol occurs, its count is increased by one. Since both sender and receiver update their individual symbol counts based on the same previous data sequence, the codes used by both sides always agree. Other problem is that, the encoded data propagate errors. This will cause the receiver to mismatch a code-word and cause the de-synchronization of the bit streams on both sides of the channel. One Solution to this problem is segmentation of data into small segments whereby the error propagation is limited within the range of a segment. However, low compression efficiency arises when the adaptive coding is applied to segmented data which is not desirable. A better approach in solving this problem is presented in this paper. It introduces a new algorithm which traps the local statistics more quickly than the above adaptive coding. The fast-adaptive Huffman code requires a smaller time constant to yield the best performance, compared to that required by adaptive coding. Also, dispersion in data to be encoded affects the compression efficiency of this algorithm.

LITERATURE SURVEY

In September 1952, A Method for the Construction of Minimum-Redundancy Codes by David A. Huffman was published. A minimum redundancy code was constructed in such a way that the average number of coding digits per message is minimized. This optimum method of coding an ensemble of messages consisting of a finite number of members is known as the Huffman Code.

If it is assumed that each symbol requires the same time for transmission, then the time for transmission (length) of a message is directly proportional to the number of symbols associated with it. Thus, a code which yields the lowest possible average message length will have the least transmission time.

To achieve this, the following basic restrictions were imposed on the code: (a) No two messages will consist of identical arrangements of coding digits. (b) The message codes will be constructed in such a way that no additional indication is necessary to specify where a message code begins and ends once the starting point of a sequence of messages is known. (c) $L(1) < L(2) < ... < L(N-1) = L(N)$. Given: $P(1) > P(2) > ... P(N-1) > P(N)$ (d) At least two and not more than two of the messages with code length L(N) have codes which are alike except for their final digits. (e) Each possible sequence of L(N) -1 digits must be used either as a message code or must have one of its prefixes used as a message code. The result was the Huffman code. It is the most optimal among methods encoding symbols separately. However, it does not give a unique code.

In November 1978 in honour of the twenty-fifth anniversary of Huffman coding, four new results about Huffman codes were presented in the paper "Variations on a Theme by Huffman". The results included:

1. Binary prefix condition code is a Huffman code i.f.f. the intermediate and terminal nodes in the code tree can be listed by non-increasing probability so that each node in the first is adjacent to its sibling. 2. Upper bounds the redundancy (expected length minus entropy) of a binary Huffman code by $P + \log2[2(\log2e)/e] = P + 0.086$, where P, is the probability of the most likely source letter. 3. Leaving code word of length two unused and still have a redundancy of at most one. 4. Simple algorithm for adapting a Huffman code to slowly vary estimates of the source probabilities.
The paper presents difficulties in variable-length codes like fluctuation in the rate at which binary digits are delivered to the channel and dependency of expected rate on relative frequencies of the source letters. On the other hand, it also highlights several uses of these codes in future like growth of statistical multiplexers, concentrators, and data networks. Also, processing costs and storage costs are dropping very much faster than communication costs has fundamentally changed the trade-off between communication efficiency and processing complexity.

A method of recursive splitting was proposed in Ref[5] as an improvement in huffman coding,where the sequence was divided into two sequences in such a way that both sequences have different symbol

statistics. Individual Huffman coding for each of these sequences will reduce the average bit rate.The sequence in a split recursively until the gain dominates the cost associated in splitting.

The problem with Huffman coding is that we require estimation of probability or frequency of occurrence for each possible value of the source symbol. One such method to resolve this problem is Adaptive Huffman coding. As a solution to the problem, the method of Adaptive Huffman Coding (AHC) was proposed by Jeffrey Vitter in his paper published in 1987.In order to transmit symbols without any idea of initial knowledge of source distribution this method was proposed. These Huffman encoders maintain a running estimate of the source letter probabilities; as these estimates change, the code will change, remaining optimal for the current estimates .A modification of Adaptive Huffman Coding for use in encoding large alphabets was also suggested in Ref[4]. It was related to the process of adding a new character to the coding tree, namely, the author proposes to introduce two special nodes instead of a single NYT (not yet transmitted) node as in the classic method. The modified method was compared with existing methods of coding in terms of overall data compression ratio and performance. The proposed method may be used for large alphabets i.e. for encoding the whole words instead of separate characters, when new elements are added to the tree comparatively frequently.

Another problem that was observed with the Huffman code was that the encoded data may propagate errors. To deal with this issue, in April, 1989 an error detection method was suggested. The decoding process is desynchronized whenever the received code word has a different length than the transmitted code word. For resynchronization, the data stream is segmented. This ensures that an error bit cannot affect more than one segment.

However, the adaptive Huffman code performs badly when segmenting data into relatively small segments because of its relatively slow adaptability. This poses a problem as both the issues must be resolved together. So, this paper offers a fast-adaptive coding algorithm which tracks the local data statistics more quickly, thus yielding better compression efficiency.

In Ref[7] Huffman coding is compared to Shannon Fano Elias Coding in terms of use of both algorithms in encryption. Though, it is known that Huffman coding is one of the best-known compression techniques that produce optimal compression for any given probability distribution. But, during encryption it is less effective. If the code construction rule and probability mass function(PMF) of the source is known Huffman code provides no ambiguity. For this reason, Shannon Fano Elias coding is a better candidate for encrypting data. The order of symbols can be arbitrary in the encoding process which produces exponential complexity in deciphering.

WORK DONE

Approach

The following steps were followed in approaching the problem statement and finding a suitable solution:

1) Firstly, the key objectives of the problem statement were enlisted. As mentioned before, the primary goal was to implement the Fast Adaptive Huffman Code, as described in Section 'II. Description of Algorithm' of our chosen research paper. So a program was to be written that would take an input word, and give the Fast Adaptive Huffman Code as output, as illustrated in the research paper.
2) Next, a suitable Programming Language needed to be chosen. Based on our skill set, 4 languages- C, C++, Python and MATLAB were available. As the code was going to be considerably complex, in order to reduce syntactical complexity, C and C++ were eliminated. Also, as a variety of Data Structures were to be used, MATLAB was discarded. Python was chosen as it is well-documented, has a concise syntax with no unnecessary code lines, and has a wide variety of built-in Data Structures and Libraries that can be used
3) The next step was to divide the whole algorithm into stages, and then implement all the stages one-by-one. The Front Tree and the Back Tree were implemented separately at first, and then they were used together to implement the algorithm.

Implementation (Code)

Back Tree: It is implemented as a *List*. A list, in Python is analogous to Array in C and Vector in C++. Initially, it contains all 26 letters, ordered from A-Z. According to the 26 possible lengths of the list, the program stores 26 possible combinations of the Binary Code Values. When the Input character is in the Back tree, its index in the list is found, and the corresponding Binary Code is found. This is then appended to the Output, which is also in the form of a List.

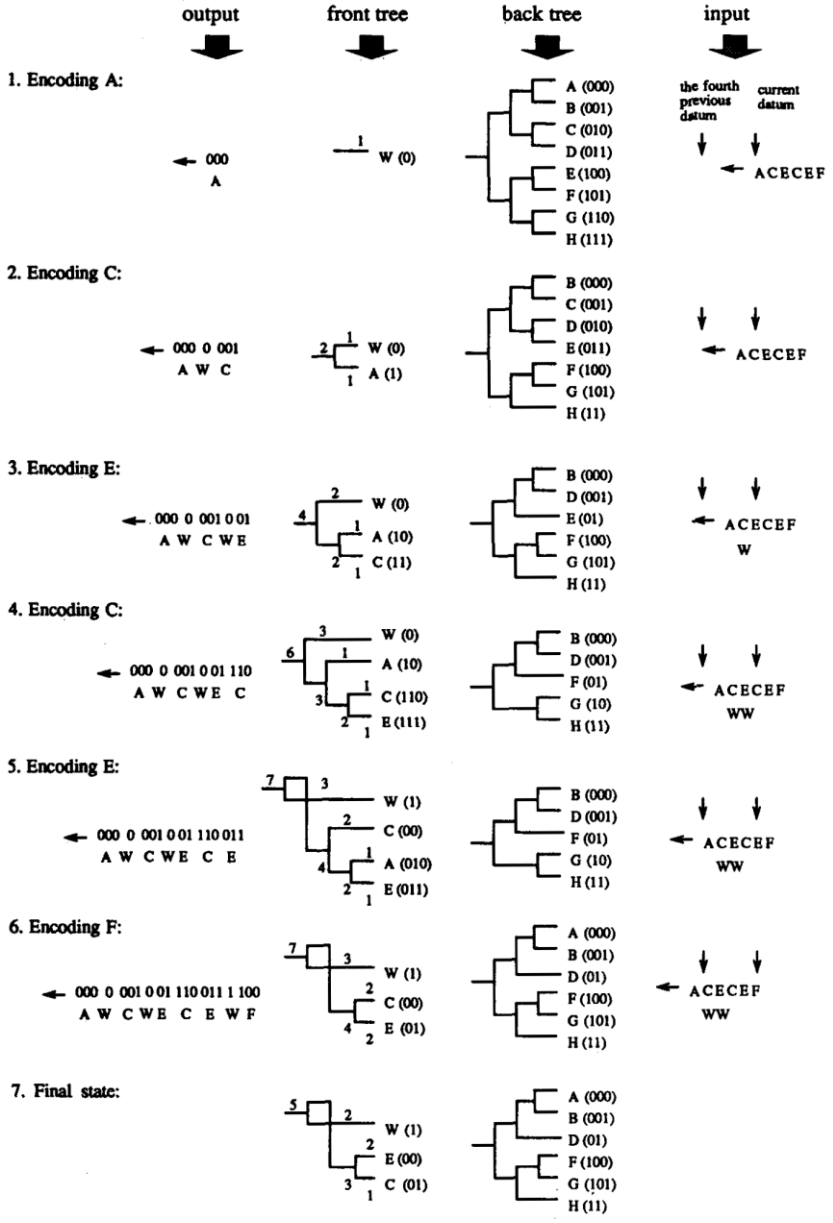
Front Tree: It uses Adaptive Huffman Coding to code its symbols and is implemented as a *Dictionary* . A dictionary in Python is a collection of Keys and their corresponding values. In the code, one dictionary is used to store the characters (symbols) in the Front Tree along with their respective counts. The second dictionary maps the symbols to their respective Adaptive-Huffman codes. Both the dictionaries are updated when a symbol is added or when a symbol is removed.

Demo Run: To check the accuracy of the code, it was tested against the input given in the paper, i.e. 'ACECEF'. In our code, the Special Code-Word was chosen as '$' instead of 'W' as done in the paper. The Time Constant was taken as 4, but can be easily changed.
The output of the Python command line after executing the code with given input is shown below:

```
In [56]: OS
Out[56]: ['A', '$', 'C', '$', 'E', 'C', 'E', '$', 'F']

In [57]: OC
Out[57]: ['000', '0', '001', '0', '01', '110', '011', '1', '100']
```

Note: The working of code for other inputs can also be demonstrated, as and when instructed by the IC.



Modifications/Contribution

1) The Code for the Described Algorithm was modified to include all 26 letters of the English Alphabet (A-Z), as well as 6 other symbols- "."(Full stop), ","(Comma), "("(Round Open Bracket), ")" (Round Close Bracket), "-"(Hyphen), " "(Space). This allows normal text to be encoded using this algorithm (Code can be found in 'FastAdaptiveHuffmanCoding-Modified.py')

2) According to Ref[7], Shannon Fano Elias Coding has some advantages over Adaptive Huffman Coding in terms of less ambiguity. Therefore, a 'Fast Shannon Fano Elias' coding scheme was devised, wherein the Front Tree uses Shannon Fano Elias Coding instead of Adaptive Huffman Coding. (Code: Fast_SFE.py and SFE-Modified.py)

3) An executable File (Encoder.exe) was developed. It reads data from 'Information.txt', compresses it and stores it in 'Code.txt', just at a click of a button. This application can be further developed and if possible, patented.

COMPARATIVE ANALYSIS

Compression Ratio = Output Code Length/ (8 * Input length)
The compression Ratio and timing for various codes are-

1) Fast Adaptive:

```
['A', '$', 'C', '$', 'E', 'C', 'E', '$', 'F']
['000', '0', '001', '0', '01', '110', '011', '1', '100']
Compression ratio is =  41.66666666666667
--- 0.0019991397857666016 seconds ---
```

2) Fast Adaptive Modified

```
['A', '$', 'C', '$', 'E', 'C', 'E', '$', 'F']
['00000', '0', '00001', '0', '00010', '110', '011', '1', '01001']
Compression ratio is =  60.416666666666664
--- 0.003004312515258789 seconds ---
```

3) SFE

```
['A', '$', 'C', '$', 'E', 'C', 'E', '$', 'F']
['000', '11', '001', '11', '01', '0100', '1000', '110', '100']
Compression ratio is =  54.166666666666664
--- 0.0020024776458740234 seconds ---
```

4) SFE Modified

```
['A', '$', 'C', '$', 'E', 'C', 'E', '$', 'F']
['00000', '11', '00001', '11', '00010', '0100', '1000', '110', '01001']
Compression ratio is =  72.91666666666666
--- 0.3319685459136963 seconds ---
```

APPLICATIONS

Huffman coding is used as lossless data compression algorithm. It has wide use because of their comparative simplicity, high speed due to lack of arithmetic calculations, and lack of patent coverage. Huffman code is used to convert fixed length codes into variable length codes, which results in lossless compression. Some of its applications include:

1. Huffman coding is a technique used to compress files for transmission. Works well for text and fax transmission.
2. They are often used as a "back-end" to other compression methods.
3. Huffman is widely used in all the mainstream compression formats that you might encounter - from GZIP, PKZIP (winzip etc) and BZIP2, to image formats such as JPEG and PNG.
4. DEFLATE (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by the use of prefix codes.
5. A slightly different use of Huffman encoding is in conjunction with cryptography.

Due to large number of sources, mistaken assumptions about some of the source statistics lead to inefficiency when using variable-length codes. Thus, modifications to the code are being tried to improve efficiency.

REFERENCES

[l] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” Proc. IRE, vol. 40, pp. 1098-1101, Sept. 1952.

[2] R. G. Gallager, “Variations on a theme by Huffman,” IEEE Trans. Inform.Theory, vol. IT-24, pp. 668-674, Nov. 1978.

[3] K. M. Rose, and A. Heiman, “Enhancement of one-dimensional variable- length DPCM images corrupted by transmission errors,” IEEE Trans.Commun., vol. 37, pp., 373- 379, Apr. 1989.

[4]Mikhail Tokovarov1-”Modification of Adaptive Huffman Coding for use in encoding large alphabets,”1Lublin University of Technology, Electrical Engineering and Computer Science Faculty, Institute of Computer Science, Nadbystrzycka 36B,20-618 Lublin, Poland.

[5]Karl Skretting, John H˚akon Husøy and Sven Ole Aase,” Improved Huffman Coding using Recursive Splitting”

[6] Satpreet Singh and Harmandeep Singh”Improved Adaptive Huffman Compression Algorithm”International Journal of Computers & Technology,Volume 1 No.1, Dec. 2011

[7] X. Ruan and R. Katti, "Using Improved Shannon-Fano-Elias Codes for Data Encryption," 2006 IEEE International Symposium on Information Theory, Seattle, WA, 2006, pp. 1249-1252.

APPENDIX

All codes, and instruction to run codes can be found on – https://github.com/JayDigvijay/Fast-Adaptive-Huffman-Coding
Please refer to README.md for instructions to use the code, or e-mail: f20170317@hyderabad.bits-pilani.ac.in