King Saud University

College of computer and Information Sciences

Computer Science Department

Algorithms Design and Analysis (CSC 311)

Spring 2020

***Huffman Coding and Decoding***

Group

|  |  |
| --- | --- |
| Name | ID |
| Mohammed Thabit | 437103405 |
| Abdulaziz Alsannat | 437106037 |

April 2020

**Table of Contents**

[I. Introduction 3](#_Toc38406106)

[II. Huffman Coding and Decoding 3](#_Toc38406107)

[III. Implementation 4](#_Toc38406108)

[Algorithms 5](#_Toc38406109)

[Experimental Results 7](#_Toc38406110)

[References 9](#_Toc38406111)

# **I.** **Introduction**

Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding and/or using such a code proceeds by means of Huffman coding. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol. More common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted.

# **II. Huffman Coding and Decoding**

A. Coding and Encoding

The technique works by creating a binary tree of nodes. These are stored in a Priority Queue, the size of which depends on the number of symbols. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a weight, links to two child nodes and an optional link to a parent node. bit '1' represents following the left child and bit '0' represents following the right child. Generally speaking, the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value).

B. Decoding

There is a general decoding technique used on an actual Huffman tree, but since our program will accept a text file containing both an encoded message and a Huffman tree, we created our own algorithm to decode the text file. Firstly, the Huffman tree is represented by 0s, 1s and characters; Starting from the root, we start moving using pre-Order traversal, all 0s are internal nodes and all 1s are leaf nodes followed by a character. An example of a Huffman tree code would be:

Encoded Message: 000001100101110111

Huffman Tree: 001A1B001C1D01E1F

Our decoding technique is that we take one bit at a time from the encoded message and if it is a ‘0’, we move to the next node (in this case, from ‘0’ to ‘0’). if it is a ‘1’ we traverse the left subtree of our current node, we do this by moving through the tree and counting 0s and 1s, we stop counting whenever they have the same number of occurrences, this means we traversed the whole left subtree. When we count, we include the current node in the 0s, and when counting 1s we skip the followed character. If we took ne bit from the encoded message and we moved to a leaf node in the tree, we write the character in the decoded text file. And so on.

# **III. Implementation**

When we were implementing, our approach was using a priority queue of nodes, the node contains the character and the frequency and the other left child and the right child. We also used a TreeMap which contains all the characters and their code word.

For the GUI we used the javax swing library to implement our GUI, we created three buttons: About us, Start and How to use. When clicking on About us button, a brief description for the developers and the project will appear. When clicking on the Start button, it will take the user to the main window containing five buttons: Browse, Decode, Encode, Build and a Back button. When clicking on the How to use button it will display to the user a brief description on how to use the program.

The text file and the output file will be measured in bytes, the execution speed will be measured in milliseconds.

# **Algorithms**

1. **Building the Huffman tree**

1: n := |C|;

2: Q := C;

3: for i := 1 to n − 1 do

4: allocate a new node z

5: z.left := x := Extract-Min(Q);

6: z.right := y := Extract-Min(Q);

7: z.freq := x.freq + y.freq;

8: Insert(Q, z);

9: end for

10: return ROOT {return the root of the tree}

O(nlogn)

1. **Assigning the codeword**

We start for the root node

1: if the root is not Null do

2: if the right child is not null do

3: we call the same method with the right child and '0'

4: if the right child is not null do

5: we call the same method with the left child and '1'

6: if the node is leaf node do

7: assign the character with the called '1' or '0'

T(n)=2T(n/2)+O(1)

Using the master theorem, we know that the algorithm Is O(n).

1. **Encoding the text**

1: for i=0 to input length do

2: save to file 🡨 the code word of each character of the text

3: show output

O(n) where n is the text length

1. **Decoding**

k=0 for(j=0 to encoded.length)

if encoded.character(j) = ‘0’){

if(treeCode.character(k) = ‘1’)

decoded += treeCode.character(j)

k=0 continue

} else

nb0s, nb1s = 0

for(k to treeCode.length){

if(treeCode.character = 0)

nb0s++

if(treeCode.character = 1){

nb1s++ k++ }

if (nb0s = nb1s)

break;

if(treeCode.character(k) = ‘1’)

decoded += treeCode.character(j)

k=0 continue

O(n)

# **Experimental Results**

We will browse the text file from the GUI and then encode it, the encoding will write the encoded text into a file and measure its size.

We took each character and filled it’s codeword, codeword bits, and the character frequency.

We take the codeword bits and we multiply it with the frequency so we can get the number of bits that will be stored in the encoded text file.

In *Table 1* below, we will take the input text: “BCCABBDDAECCBBAEDDCC”

Table 1: Example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Character | A | B | C | D | E | Sum |
| Codeword | 010 | 10 | 00 | 11 | 011 |  |
| Codeword bits | 3 | 2 | 2 | 2 | 3 |  |
| Character Frequency | 3 | 5 | 6 | 4 | 2 |  |
| Codework bits\* frequency | 9 | 10 | 12 | 8 | 6 | 45 bits |

In *Figure 1* below, the chart shows that we used four randomly generated texts to measure the execution time as we increase in input size.

The measuring of performance and execution time of the algorithm, we used the System.currentMillies() to get the execution time of the algorithm.

Figure 1: Encoding - Input size vs Execution time

**References**

[1] Huffman coding Wiki

[https://en.wikipedia.org/wiki/Huffman\_coding#Problem\_definitiont](https://en.wikipedia.org/wiki/Huffman_coding%23Problem_definitiont)

[2] Some of the code prior to change <https://gist.github.com/ahmedengu/aa8d85b12fccf0d08e895807edee7603>

[3] Some of the GUI code prior to change <https://github.com/sukalyansen123/Huffman-Tree-Animation-GUI>

[4] Algorithm: <http://math.ubbcluj.ro/~tradu/TI/huffmancode.pdf>

[5] how to use java swing to implement buttons and draw lines and squares

[https://www.youtube.com/watch?v=zG8CrISqPpU](https://www.youtube.com/watch?v=zG8CrISqPpU%20)