**Name:** Raj Jain

**CID:** 01409529

**Title of the Assignment:** Text Compression using Huffman's Coding Algorithm

**Course:** EE2 Algorithms and Data Structures

**Page Count:** 10 pages

Department of Electrical and Electronic Engineering

**Table of Contents**

Introduction to Huffman Coding Algorithm…….3

Explanation of function huffencode……..4

Explanation of function huffdecode……..8

Explanation of function valid_hufftree…….9

An experimental evaluation of the algorithms illustrated with test examples and observations…….10

Conclusion……....12

**Introduction to the Huffman Coding Algorithm**

Huffman Code is a type of an optimal prefix code that is used for lossless data compression. The process of developing such a code is referred to as Huffman Coding and this algorithm, was developed by David A. Huffman at the Massachusetts Institute of Technology in 1952. It is used with a variety of data types such as images (JPEG), audio (MP3), and videos. The essential idea is to choose codeword lengths so that more-probable sequences have shorter code words. The code can be constructed by the following steps:

1. The symbols are sorted in decreasing order of probability. The probability of each element is denoted is $p(x_i)$ where "$x_i$" denotes the i'th element in the list.
2. The elements with the smallest probabilities $p(x_i)$, are assigned each a different single binary bit (i.e. 0 or 1). Then, they are merged into a single symbol.
3. The previous step is repeated until only one symbol remains.

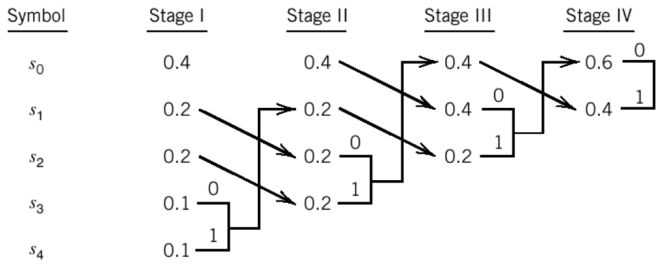


*Fig: Example showing different symbols from $s_0$ to $s_4$ arranged in descending order of probabilities indicated by stage 1*

As we can see in the above example, we proceed from stage 1 to stage 2 by summing the probabilities of elements $s_3$ and $s_4$ into 0.2. At the same time, we assign elements $s_3$ and $s_4$ to 0 and 1 binary bits. This assignment of bits can be unique wherein $s_4$ can be assigned as 0 and $s_3$ can be assigned 1. And similarly, we proceed to the remaining stages in the diagrams. The idea is to combine symbols with the least probabilities. Also, if there are symbols with the same probabilities, it doesn't matter which two are picked.

| Symbol | Probability | Code word |
|---|---|---|
| $s_0$ | 0.4 | 00 |
| $s_1$ | 0.2 | 10 |
| $s_2$ | 0.2 | 11 |
| $s_3$ | 0.1 | 010 |
| $s_4$ | 0.1 | 011 |

*Fig: Example showing the encoding of the different symbols (in binary bits) from the previous example.*

By reading the Huffman tree backwards, we can deduce a codeword for each symbol. As it can be seen, the symbols with the high probabilities are coded with the highest probabilities. For instance, $s_0$ is encoded with the code word 00, i.e. with only two binary bits compared to the encoding for the $s_3$ and $s_4$

which have three binary bits as part of their code word. The latter is due to the fact that they have lower probabilities.

In this report, the explanation for the Huffman encoding and decoding algorithms for the C++ assignment will be shown. Furthermore, the algorithm to check whether the given Huffman tree built is valid. This will be followed by an experimental evaluation of the algorithms illustrated with the test examples and observations. For instance, the analysis for the graph of the execution time versus the number of bits will be considered. Throughout this report, the input message string that will be considered is "Engineering"

**Explanation of function `huffencode`**

```
std::string huffencode(const std::string& message, std::map<char,
int>& freqtable, hufftreeptr& hufftree, std::map<char, std::string>&
hufftable);
```

1) The function is supposed to build a frequency table `freqtable` by examining the input text message and computing the frequency of each character.

```
An array initialized to 0.
int counter[128] = 0;

For each element of the message string
     Increment counter[element] by 1 (where element represents the ASCII
representation of the element)
```

The output of the above code for the input message string "engineering" is as follows:

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000302020003000100000000000000
```

Therefore, the 101st element of the output above is 3. This is because, the ASCII representation for the letter 'e' in decimal is 101. Since, the letter 'e' appears 3 times in the word "engineering", 101st element of the array is equal to 3. The same follows for all the other elements of the array which can be represented by the table given below:

| Element | ASCII Value | Frequency of the Element in the input string | The index of the counter array | Value at the index of the array |
|---|---|---|---|---|
| e | 101 | 3 | 101 | 3 |
| n | 110 | 3 | 110 | 3 |
| g | 103 | 2 | 103 | 2 |
| i | 105 | 2 | 105 | 2 |
| r | 114 | 1 | 114 | 1 |

*Table: Data showing the elements in our test case along with its ASCII value and the frequency of its occurrence in the input string 'engineering' along with the indices of the above obtained counter array along with the value at its corresponding values*

It is no coincidence that the ASCII value and the index of the counter array is the same as the ASCII value for each element and this is justified above. Consequently, we can implement the frequency table for this input string as follows:

```
for i = 0 to 127
     if the counter at i is not equal to 0
          insert the character and its frequency in a freqtable
     end if
end for
```

Therefore, the characters in the input string along with their respective frequencies are inserted into the table using the insert function from the header file, "#include <map>".

2) The Huffman tree `hufftree` is built using the given input and output parameters.

- A single-node binary tree is created storing the character c, and its frequency f(c). Then this is inserted into a collection of trees, called a Forest F.

We implement this using a method called the priority queue which can be included as part of the queue header file. The characters are to be inserted in the Forest F in the descending order of the frequencies as defined in the documentation. This could've been implemented using a vector and bubble sorting the characters. However, through further analysis, it was discovered that the bubble sort method implemented had an average complexity of $O(n^2)$ which is undesirable.

On the other hand, `std::priority_queue` under the header file of queue offers the interface such that it includes the elements in the queue and when an element has to be retrieved, the largest or smallest one is chosen, depending on the type of implementation. One way of implementing the priority queues are by using heaps. Moreover, this has a complexity of O(nlog(n)) which is better compared to the complexity offered by sorting.

```
for i = 0 to 127
     if the counter at i is not equal to 0
          hufftreeptr ptr declared as a dynamic variable
          store character at i in ptr->character
          store frequency at i in ptr->frequency
          ptr->left and ptr->right =NULL (single-node binary tree)
          push elements in the queue

     endif
endfor
```

- The next step towards building the Huffman tree is to choose two trees T1 and T2 in the forest F with the smallest frequencies $f_1$ and $f_2$. Then we merge T1 and T2 into a single tree with T1 being the left sub-tree and T2 being the right sub-tree. The frequency of the merged tree is the $f_1 + f_2$. This step is repeated until there is only one tree T left in the forest F. The single tree T in the forest F is the Huffman encoding tree called hufftree.

```
while the size of the queue is 1

     assign the top element of the queue to hufftreeptr T1
     remove this element from the queue
     assign the top element of the queue to hufftreeptr T2
     remove this element from the queue

     hufftreeptr r is declared as the dynamic variable

     frequency of r = frequency of T1 + frequency of T2

     assign left sub-tree of r as T1 (ptr->left)
     assign right sub-tree of r as T2 (ptr-right)

     insert it back into the queue

endwhile
```

Essentially, in the above pseudocode, we assign the top element of the queue to a pointer T1 of type `hufftreeptr` and this element is then removed from the list. As this is a priority queue, the next smallest element in the queue will appear at the top. The aforementioned process is repeated but with a different pointer T2. Then, the two elements are combined into "r" which has the frequency equal to the frequency of T1 + T2. After the elements with the two smallest frequencies are picked, the left sub-tree and right sub-tree of r point to T1 and T2, respectively. Finally, this is pushed back into the priority queue. The entire process is repeated until the size of the queue is equal to 1 which leads to the Huffman tree shown below:

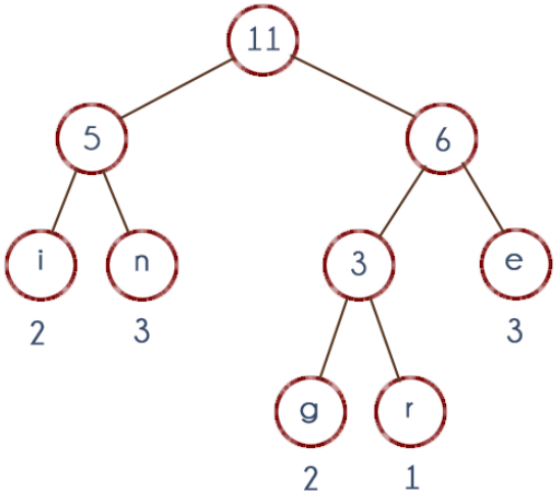


*Fig: Huffman Tree for the input string message "engineering"*

Finally, we assign the top of the priority queue to the `hufftree` which contains the final Huffman tree.

3) This will be followed by the building of the associated encoding table.

The encoding table `hufftable` was built by declaring another function called `void preorder` which is defined as follows:

```
void preorder(hufftreeptr ptr, std::string code, std::map<char, std::string>& hufftable)
```

This function is called in `huffencode` and takes in the input parameters `hufftree`, an empty string and the `hufftable`.

We can look at an example of the character "i" which is encoded in the input message string "engineering" by considering the Huffman tree built in the previous section. If we look at lines x-y of the preorder code in Appendix A, the code first executes the else statement. The leftcode is assigned the empty string along with a 0 string. Then the code enters into a recursion with the ptr->left as the first input parameter. In the first recursion step, the else statement is executed once again with the leftcode being assigned 0 one more time. Therefore, the code is concatenated to 00 for the character.

Finally, we enter the second if statement where the character and the code are inserted into the Huffman table which is "i" and "00", respectively. Similarly, the remaining elements are inserted into the Huffman table by traversing through the Huffman tree.

| Element | Frequency of the element in the string "engineering" | Codeword |
|---|---|---|
| i | 2 | 00 |
| n | 3 | 01 |
| r | 1 | 100 |
| g | 2 | 101 |
| e | 3 | 11 |

*Table: Data showing the list of the elements of the input string "engineering" along with their corresponding frequencies and codewords*

From the table above, it can also be confirmed that the most frequent elements are coded with short bit-strings and on the other hand, long bit-strings are used for less frequent elements.

4) It encodes the input message (a string of characters) into a compressed message (string of bits) and returns that compressed message.

The input message is encoded into a compressed message by using the hufftable parameter, which is considered after the Huffman table. Below is a pseudocode for the same:

```
string res;
for i = 0 to message length
     concatenate the encodings from the huffman table into res
endfor
```

Therefore, the following encoded message is obtained for the input string "engineering" is: 110110100011111000001101

**Explanation of function `huffdecode`**

```
std::string huffdecode(const std::string& encodedmessage, const
hufftreeptr& hufftree)
```

The function `huffdecode` decodes an encoded message `encodedmessage` using the Huffman tree `hufftree` and returns the output message. Specifically, the Huffman encoding table constructed from the Huffman tree T has an important *prefix property* which specifies that no code word in the encoding table is a prefix of another code word.

To implement this function, we use another function called `preorderencode` which is quite similar to `preorder` function defined for implementing the `huffencode` function. `preorderencode` is implemented as follows:

```
void preorderencode(hufftreeptr ptr, std:: string code,std::map<
std::string,char>& hufftable)
```

This function creates a `hufftable` with the code words and their corresponding characters. In `preorder`, the function created a `hufftable` wherein characters were printed with their respective code words.

After calling the function `preorderencode` in the `huffdecode` function, a for-loop is executed that runs till the length of the encoded message which extracts the characters corresponding to the codewords (bit-strings) from the encoded message. If the value of the encoded message doesn't correspond to a code word in the `hufftable`, the if statement will not be executed, and the for-loop will be incremented by 1.

```
for i = 0 to length of the encoded message
     bitstring = bitstring + encodedmessage[i]
          if (the value of the encoded message = code word in the
          hufftable)
               msg = msg + (character corresponding to the code word)
               initialize bitstring
          endif
endfor
```

The encoded message for the input string "engineering" is: 11011010001111110000011101. This is decoded as follows:

right -> right is the code word "11" which is equal to the element "e" in the Huffman table. Then the bit-string variable is initialized once again in the if-statement and we return to the for-loop where it is incremented by one. Similarly, left -> right has the code word 01 which corresponds to the character "n" and this is added to the "msg" variable in the if-statement.

**Explanation of the function `valid_hufftree`**

```
bool valid_hufftree(const hufftreeptr hufftree)
```

This utility function is required to check whether the Huffman tree built is valid and consequently, returns true if the condition is satisfied. This function was implemented by defining the following functions:

```
bool leafNode(hufftreeptr hufftree)
```

This function traverses through the Huffman tree towards the leaf nodes to check if they contain characters. It uses recursion to reach the leaves of the binary trees and uses a simple if-statement to check if a character exists in the message.

```
bool fulltree(hufftreeptr ptr)
```

The function `fulltree` traverses through the tree to check if the non-leaf nodes have two sub-trees each.

```
int sum(hufftreeptr ptr)
```

Finally, this function checks if the occurrence frequency of the non-leaf nodes equals the sum of the frequencies of its two sub-trees.

**An experimental evaluation of the algorithms illustrated with test examples and observations**

To calculate the execution time for each of the functions, the file header #include <chrono> was used and the values were obtained as follows:

| Number of Characters | Execution time for Huffencode (Microseconds) | Execution time for Huffdecode (Microseconds | Execution time for Validhufftree (Microseconds) |
|---|---|---|---|
| 1000 | 337 | 3546 | 40 |
| 2500 | 631 | 8956 | 41 |
| 4000 | 1143 | 13688 | 41 |
| 6500 | 1250 | 21858 | 39 |
| 8000 | 1474 | 26302 | 39 |
| 9500 | 1661 | 31802 | 39 |
| 11000 | 1837 | 36309 | 37 |
| 12000 | 2180 | 40893 | 40 |

*Table: Table showing the data for the number of characters and the execution time (in microseconds) for each of the functions*

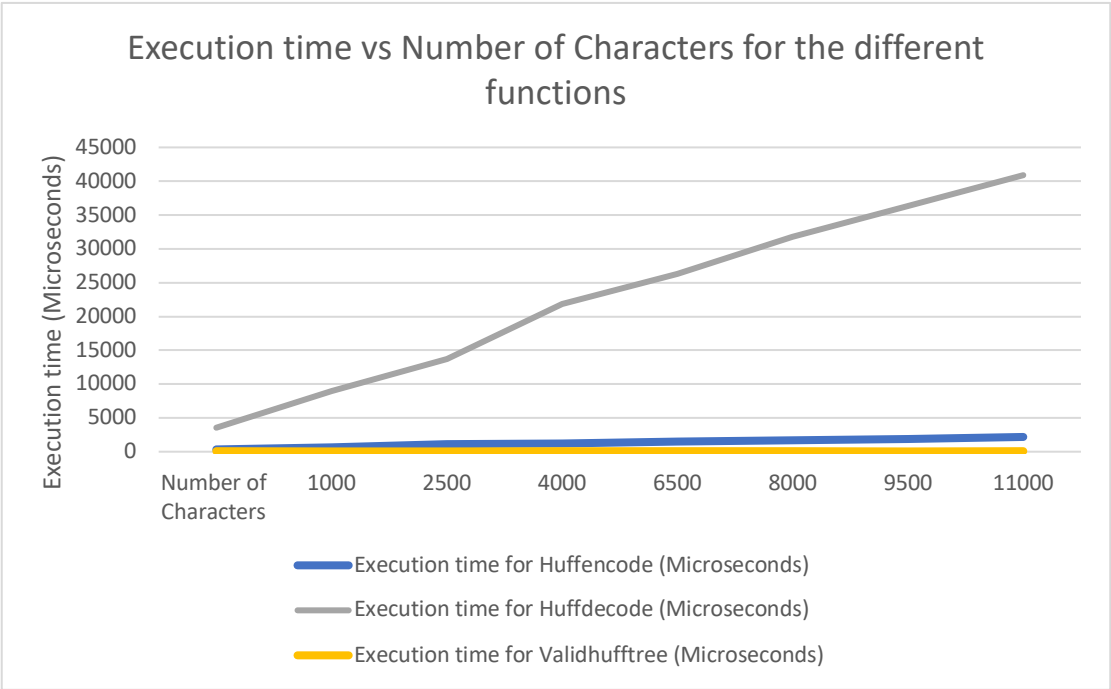


*Fig: Graph showing the execution time (in microseconds) vs the number of characters for the three functions:* `Huffencode, Huffdecode, Validhufftree`

It can be seen from the above figure that the execution time increases for `Huffencode` and `Huffdecode` as the number of characters increases. However, the execution time for the `ValidHufftree` function remains around 40 microseconds for even large values of the characters. Looking at the graphs, it can be seen that the complexities for each of the algorithms are as follows:

| Function | Complexity |
|---|---|
| `Huffencode` | O(n) |
| `Huffdecode` | O(nlogn) |
| `ValidHufftree` | O(1) |

It can be seen from the above analysis that the complexity of `Huffencode` is O(n) while using the priority queue method for the purposes of building the forest and Huffman tree. This reduces the execution time compared to other sorting methods. For instance, if vectors were used to implement sorting in the `Huffencode` algorithm, the execution time would increase to a large extent.

| Number of Characters | Execution Time (Microseconds) of the whole code |
|---|---|
| 1000 | 4266 |
| 2500 | 10621 |
| 4000 | 14320 |
| 6500 | 27777 |
| 8000 | 28603 |
| 9500 | 40279 |
| 11000 | 39835 |

*Table: Number of characters vs the execution time of the whole code (in microseconds)*

Furthermore, the data between length of the message in bits and the compression rate was obtained as follows:

| Length of the Message in Bits | Compression Rate |
|---|---|
| 88 | 0.284091 |
| 160 | 0.475 |
| 192 | 0.48375 |
| 800 | 0.56375 |
| 8000 | 0.58875 |
| 12000 | 0.590583 |
| 16000 | 0.591875 |
| 80000 | 0.5942 |

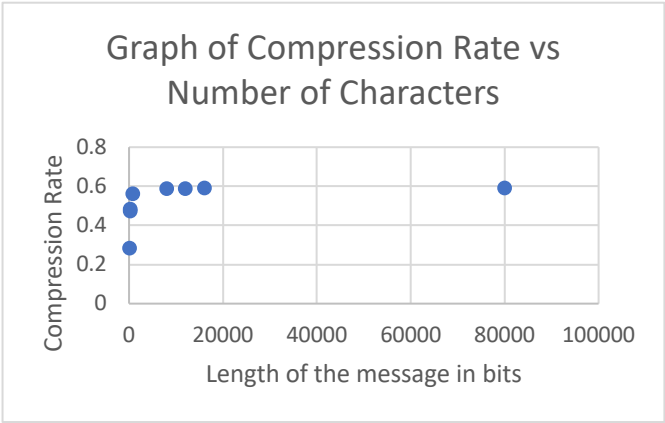


*Fig: Table showing the length of the message in bits along with the corresponding compression rate along with its plot on the right*

It can be seen from both the table and the graph that the compression rate increases as the we increase the number of message bits. However, for large message bits, the compression rate remains constant. Through this relationship, it can be seen that the compression ratio remains around 50% for English messages which is also specified in the documentation.

**Conclusion:**

In conclusion, this report began by looking at the history of the Huffman's coding algorithm and its applications along with an example. The general idea is to use smaller binary code words for messages that occur more frequently (have higher probabilities). In this assignment, lossless text compression was analyzed from the perspective of this assignment along with explanation of the implementation of the functions.

Specifically, there were three functions to implement: namely `huffencode, huffdecode` and `valid_hufftree`. The logic behind these algorithms was explored in line with the requirements. A more detailed explanation for the output of these functions was shown with the help of an input string message which was "engineering". Again, the idea of using smaller code words for repeated characters was also explored. Furthermore, to implement the two functions `huffencode` and `huffdecode`, preorder and `preorderencode` were coded, respectively. The preorder function had the objective of encoding the input binary tree along with building the `hufftable` with the characters and their corresponding codewords. On the other hand, `preorderencode` was quite similar to preorder except for the fact that the former builds an `hufftable` with the binary code along with its corresponding characters.

Furthermore, the implementation of the `valid_hufftree` function was done by implementing three separate functions, i.e. `leafnode`, `sum` and `fulltree.` Each of these functions check the required characteristics of a valid Huffman tree. Finally, a basic complexity analysis was done by looking at the execution time for each of the functions against the number of characters. Furthermore, the relationship between the compression rate and the length of the message in bits was explored. It was determined that as the number of bits increase, the compression rate will also increase. However, for larger values, the compression rate remains around 58%. However, this graph clearly supports the fact that English messages usually have a compression rate around 50%.