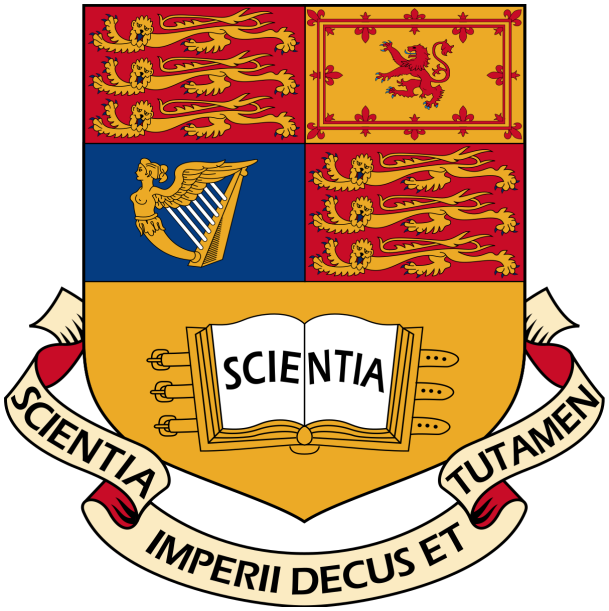
IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL & ELECTRONIC

ENGINEERING



ALGORITHMS AND DATA STRUCTURES COURSEWORK

TEXT COMPRESSION USING HUFFMAN'S CODING ALGORITHM

SNEHIL KUMAR

CID NUMBER: 01355676

COURSEWORK SUPERVISOR: MR. SAHBI BEN ISMAIL

INTRODUCTION

Over the years, hard drives are getting bigger, but the files we want to seem to keep pace with that growth which makes even today's enormous disk seem too small to hold everything. One technique to use our storage more optimally is to compress the files. By taking advantage of redundancy or patterns, we may be able to "abbreviate" the contents in such a way to take up less space yet maintain the ability to reconstruct a full version of the original when needed. Such compression could be useful when trying to cram more things on a disk or to shorten the time needed to copy/send a file over a network.

Huffman code is a particular type of optimal code that is commonly used for lossless data compression. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (a character in a string or a file). The algorithm derives this table from the estimated frequency of occurrence for each possible character of the source symbol. Unlike standard ASCII character encoding, more common symbols are generally represented using fewer bits than less common symbols.

The standard ASCII character encoding uses the same amount of space (one byte or eight bits, where each bit is either a 0 or a 1) to store each character. Common characters don't get any special treatment. They require the same 8 bits that are used for much rarer characters. For instance, a string of 500 characters encoded using the ASCII scheme will take 500 bytes (4000 bits. But in practice, not all 256 characters in the ASCII set occur with equal frequency.

The Huffman encoding scheme takes advantage of the disparity between frequencies and uses less storage for the frequently occurring characters at the expense of having to use more storage for each of rarer characters. Huffman is an example of a variable-length encoding—some characters may only require 2 or 3 bits and other characters may require more number of bits (may exceed standard 8 bits in ASCII encoding). The savings from not having to use a full 8 bits for the most common characters makes up for having to use more than 8 bits for the rare characters and the overall effect is that the file almost always requires less space.

Symbols	Frequency	ASCII Code	Codewords using Huffman code
A	50	01000001	00
B	35	01000010	101
C	42	01000011	110
D	22	01000100	1001
E	65	01000101	01
F	25	01000110	1111
G	9	01000111	1000
H	23	01001000	1110

Fig 1: Example of application of Huffman code to Compress ASCII Characters

DESCRIPTION OF ENCODING AND DECODING ALGORITHMS AND FUNCTIONS

ENCODING

The encoding algorithm works by constructing the optimal tree giving a minimal per-character encoding for a particular string. To begin generating the Huffman tree, each character gets a weight equal to the frequency of the character in the string, that is, the number of times it occurs. For example, in a string "*ABRACADABRA*", the character 'A' has weight 5, 'B' has weight 2, 'R' has weight 2, and C and D have weight 1. Our first task is to calculate these weights, and store it in a map. For each character, an unattached tree node is created which contains the character value and its corresponding weight. Each node can be described as a tree with just one entry. Priority store the nodes in the increasing order of frequency such that the node with lowest frequency is popped out first. The idea is to combine all these separate trees from the priority queue into an optimal tree by wiring them together from the bottom upwards. Conventionally, bit '0' represents left child and bit '1' represents right child. This convention is used to encode each character's binary pattern as we traverse down the Huffman tree node by node. When a leaf node is reach, the binary code described by the path followed corresponds to the character stored in that particular leaf node. The general approach is as follows:

1. Create a collection of sub-trees, one for each character, with weight equal to the character frequency.
2. From the collection, pick out the two trees with the smallest weights and remove them. Combine them into a new tree whose root has a weight equal to the sum of the weights of the two trees and with the two trees as its left and right subtrees.
3. Add the new combined tree back into the collection.
4. Repeat steps 2 and 3 until there is only one tree left.
5. The remaining node is the root of the optimal encoding tree.
6. Traverse the Huffman tree starting from the left to find the binary pattern for each character.

The pseudocode is provided below:

```
Procedure Huffman Coding(str):     // str is
the string with a set of n characters
n = str.size
PQ = priority_queue()
for i = 1 to n
    n = node(str[i])
    PQ.push(n)
end for
while PQ.size() is not equal to 1
    tree_node = new node()
    tree_node.left = x = PQ.pop
    tree_node.right = y = Q.pop
    tree_node.frequency = x.frequency +
y.frequency
    PQ.push(tree_node)
end while
```

The **huffencode** function encodes our string into bit form. Initially, it stores the frequency of each character in the string in the map **freqtable**. Then we use a priority queue, **pq** to store the node for each character. The **getTreeNode** function makes a single tree for the character. Each node is stored in ascending order of frequency by **std::greater<hufftreenode*>** function object. It checks whether the first argument is *greater* than the second to sort the nodes in ascending order of frequency in the priority queue. As a result, when we pop an item from the priority queue, the node with the least weights pops first. Within the while loop, we pop two trees from the priority queue which have the least weights, and combine them into a tree whose root has the sum of their frequencies, with the two nodes at its left and right leaves. It loops until we are left with one tree in the priority queue. It then calls the **encode** function, where the Huffman tree is traversed node by node starting from the left. It calculates the binary pattern by traversingthe tree using the convention, bit '0' for left child and bit '1' for right child, until we reach a leaf node. The binary pattern obtained corresponds to the character in that particular leaf node. Since the function is called recursively, it continues through the remaining nodes and encodes a bit pattern for all characters in leaf nodes. Binary pattern for each character is stored in **hufftable** map.

For instance, we input a string "*ABRACADABRA*". There is a total of 8 characters in the string. Our frequencies are A=5, B=2, R=2, C=1 and D=1. The two smallest frequencies are for C and D, both equal to 1, so we'll create a tree with them. The root node will contain the sum of the frequencies, in this case 1+1=2. The left node will be the first character encountered, C, and the right will contain D. Next, we have 3 items with a frequency of 2: the tree we just created, the character B and the character R. The tree came first, so it will go on the left of our new root node. B will go on the right. Repeat until the tree is complete, then fill in the 0's and 1's for the edges. The finished graph looks like:

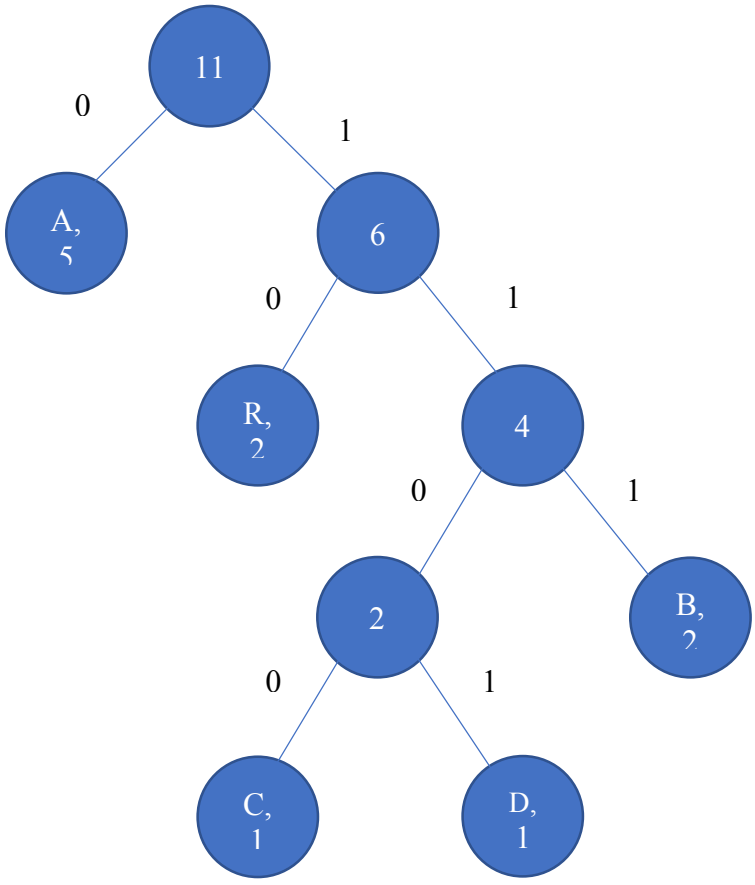


Fig 2: Huffman Tree for the string "*ABRACADABRA*"

Looking at the binary tree, the value of the characters are:

Characters	A	B	C	D	R
Huffman Code	0	111	1100	1101	10

Therefore, our encoded string is:

A	B	R	A	C	A	D	A	B	R	A
0	111	10	0	1100	0	1101	0	111	10	0

or
01111001100011010111100

If we had used standard ASCII method for encoding the string, it would have taken, 11x8=88 bits for encoding, as each character will have to be represented by an 8-bit binary pattern.

But in Huffman Encoding method, each character is assigned lesser number of bits, varying on their frequency in the string. Here, "*ABRACADABRA*" is encoded in just 23 bits.

Compression efficiency = (23/88)x100= 26%, that is, it takes only 26% of the number of bits required to encode the same string using ASCII method. The encoded string is compressed by 74%

Compression ratio= Uncompressed size/compressed size = 88/23=3.82

DECODING

The process of decoding is simply translating the stream of bit codes for each character stored in **hufftable** map. We traverse the Huffman tree node by node as each bit is read from the encoded string. Reaching a leaf node indicates that we have just completed reading the bit pattern for a single character. The single character decoded is the character stored in the leaf node obtained by following the bit pattern. Now we pick up where we left off in the bits and start tracing again from the root node. Similarly, we continue through the remaining bits and decode the input string.
The pseudo-code:

```
Procedure Huffman Decoding(root, S):   //
root represents the root of Huffman Tree
// S refers to bit-stream to be encoded bit
pattern
n := S.length
for i := 1 to n
    current = root
    while current.left != NULL and
current.right != NULL
        if S[i] is equal to '0'
            current := current.left
        else
            current := current.right
        endif
        i := i+1
    endwhile
```

The **huffdecode** function decodes the binary pattern back to the input string. It runs a while loop until we reach the end of encoded binary pattern. Within the loop, it calls the **decode** function. The decode function traverses the Huffman tree as it reads the bits from the encoded pattern. It adds the corresponding character to a string once a leaf node is reached. Since the function is called recursively, it continues through the remaining bits, starting from the root node each time, and adds characters one by one to the string every time a leaf node is reached. Finally, the decoded string is obtained which is equivalent to the input string. Therefore, Huffman Coding is a lossless algorithm.

Now we will decode the encoded string from the above encoding example.

The string "*ABRACADABRA*" encodes to 011110011000110101111100

S="011110011000110101111100"
Processing the string from left to right.
S[0]='0' : we move to the left child of the root. We encounter a leaf node with value 'A'. We add 'A' to the decoded string.

We move back to the root.

S[1]='1' : we move to the right child.
S[2]='1' : we move to the right child.
S[3]= '1': we move to the right child. We encounter a leaf node with value 'B'. We add 'B' to the decoded string.

We move back to the root.

S[4]=’1’ : we move to the left child.
S[5]=’0’ : we move to the right child. We encounter a leaf node with value ‘R. We add ‘R’ to the decoded string.

We move back to the root.

S[6] = ‘0’ : we move to the right child of the root. We encounter a leaf node with value ‘A’. We add ‘A’ to the decoded string.

We move back to the root.

Similarly, we continue the process for the remaining bits. The final decoded string:

Decoded String = “ABRACADABRA”

The **valid_hufftree** is called in the **int main()** function to check whether the obtained Huffman tree is valid or not. It returns true if the input hufftree is a valid Huffman tree. Requirements for a valid Huffman tree:

1. All the terminal nodes (leaves) have characters,
2. All the non-leaf nodes have two sub-trees each,
3. And the occurrence frequency of a non-leaf node equals
4. The sum of the occurrence frequencies of its two sub-trees.

In the Huffman tree in fig 2, we observe that all the leaf nodes contain characters and each non-leaf node has two subtrees and weight equivalent to the sum of the weights of its two sub-trees.

COMPLEXITY ANALYSIS OF THE ALGORITHM

For encoding, firstly n sub-trees are constructed for n characters are constructed. The frequency of the nodes is compared and the node with the minimum frequency is found. Then the node is pushed into the priority queue, and the process continues until the queue contains sub-trees in the increasing order of weights/frequencies. This process takes O(n) time.

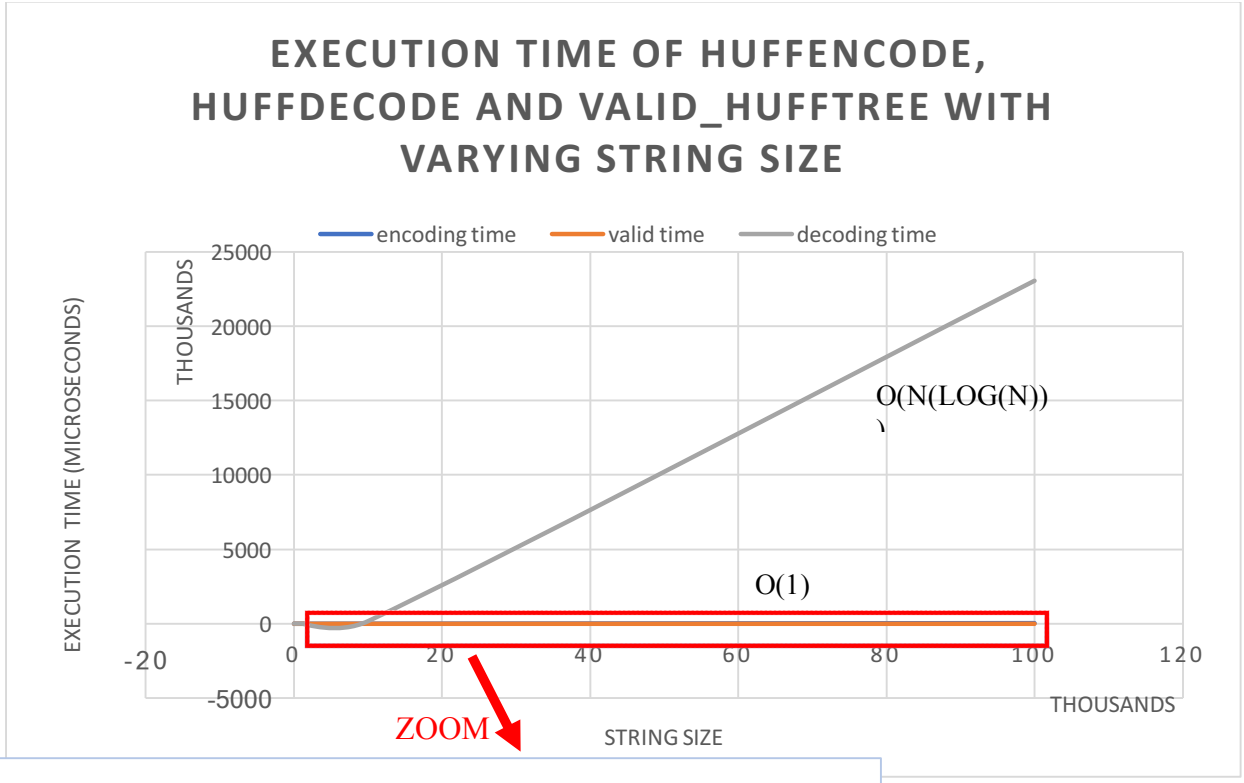
Now, we remove the two least weighted nodes from the priority queue and combine them into a tree. The deletion an element from a priority queue takes O(log(n)) time, therefore the complexity of this process of removing two least weighted nodes in O(2log(n)). The weight of the root node of the combined tree should be equivalent to the sum of the two leaf node weights. The combined tree is inserted back into the queue. Insertion into the priority queue again takes O(log(n)) time. Now the total complexity becomes O(3log(n)). Initially, there are n nodes in the priority queue. The above operation is performed until we are left with one tree, which means the

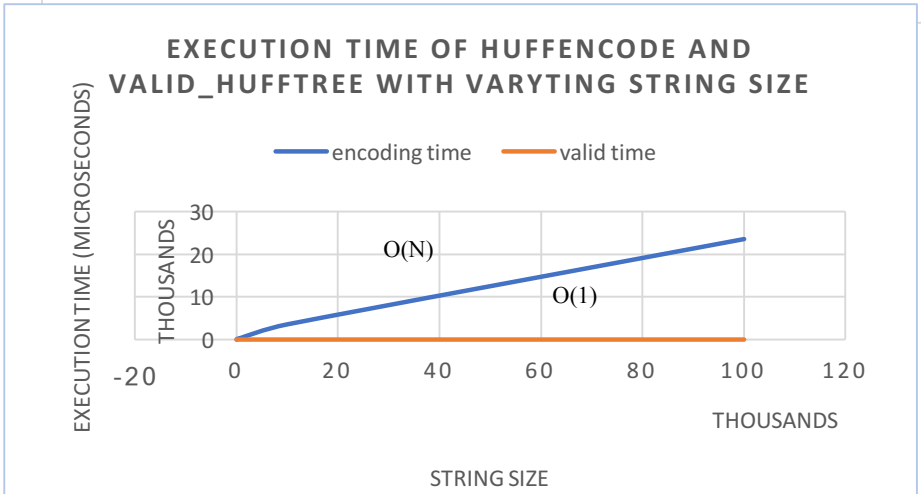
operation iterates O(n-1) times. This gives a complexity O((n-1)*3log(n)). On approximation, n-1→n and 3log(n)→log(n), so complexity is O(nlog(n)).

Therefore, the total complexity of the Huffman algorithm is:
O(nlog(n)) + O(n)= **O(nlog(n))**

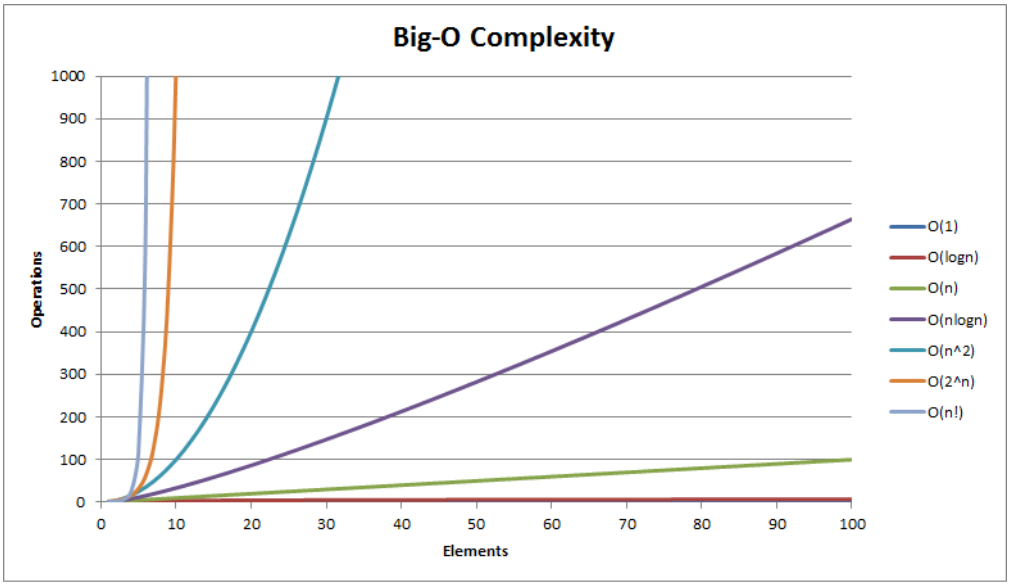
TESTING

The Huffman code was tested for strings with characters ranging from 10 to 100000. The graph obtained below plots the execution time for **huffdecode**, **huffencode**, and **valid_hufftree** functions versus the string size.





Comparing the above graph with the big-O complexity graph, execution time for each function corresponds to the complexity values analyzed in the previous section.



CONCLUSION

Huffman's algorithm is an example of a greedy algorithm. It's called greedy because the two smallest nodes are chosen at each step, and this local decision results in a globally optimal encoding tree. In general, greedy algorithms use small-grained, or local minimal/maximal choices to result in a global minimum/maximum.

This algorithm observes the occurrence of each character and stores it as an optimized binary string. It assigns variable-length codes to input characters, where length of the assigned codes is based on the frequencies of corresponding characters. A binary tree is created and operated in bottom-up manner so that the least two frequent characters are as far as possible from the root. In this way, the most frequent character gets the smallest code and the least frequent character gets the largest code.

It can also be inferred from the above analysis that a Huffman coding would compress the information by an even larger percentage than standard ASCII (fixed length) coding.

Regarding its real-world application, Huffman coding is used with data that repeats order a lot and contains a sub-set of the character space, for example, English language text files. Almost all communications with and from the internet are at some point Huffman encoded. Most image files (jpegs) and music files (mp3s) are Huffman encoded. The ZIP file format permits a number of compression algorithms though Deflate, which is a lossless data compression algorithm and associated file format that uses a combination of the LZ77 algorithm and Huffman coding.

REFERENCES

- https://en.wikipedia.org/wiki/Huffman_coding
- http://stevenpigeon.com/Publications/publications/HuffmanChapter.pdf