EE3980 Algorithms

hw09 Encoding ASCII Texts

106061146 陳兆廷

Introduction:

In this homework, I will be analyzing, implementing, and observing 1 algorithm. The goal of the algorithm is to encode a list of characters into Huffman codes. The input of them will be an essay, and the output of them will be a list of characters used in the essay and their Huffman code. Furthermore, the ratio between the space used to store ASCII and Huffman will be calculated.

During the analysis process, I will first introduce why Heap Sort and Binary Merge Tree can be used in this task. Then, I will be using counting method to calculate the time complexities of the algorithm. Finally, I will calculate their space complexity for the total spaces used by the algorithm.

The implementation of the algorithm on C code will find the Huffman Code for the provided data, course.dat.

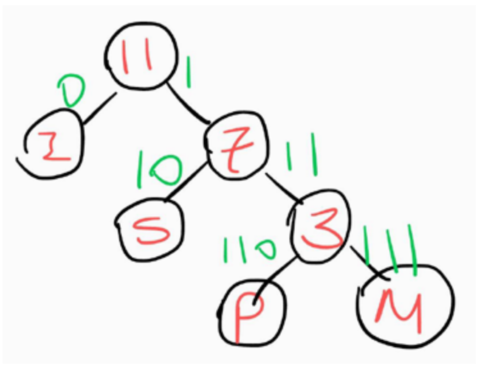
Analysis:

1. Huffman Code:

Huffman encoding is a way of representing a set of characters based on their appearing frequencies. Take the word Mississippi for an example. The data table will be each existed character and frequency table will store their appearing frequencies.

	[0]	[1]	[2]	[3]
data	M	i	s	p
freq	1	4	4	2

After Sorting characters depending on their frequencies, we implement Binary Merge Tree method to merge the least two elements until there is only one element left, and that will be the root of our Huffman Tree. Take Mississippi as an example again. After sorting it into ascending order, we get M and P and merge frequency 2 + 1 = 3 into the tree. By doing so until there is only element 11 left.



Finally, we traverse the Huffman Tree to print out our Huffman Encoding Table.

We use Heap sort as the sorting method during Huffman encoding process since there's sorting and it constantly pops the minimum value. Using heap sort ideally has O(n lg n) and O(1) for the two tasks.

2. **Data Structure:**

a. Node:

A Node has 4 properties:

properties	definition
Node->(char)data	character
Node->(int)freq	character’s frequency
Node->(Node) left	Left node
Node->(Node) right	Right node

b. Heap:

A Heap is to store the sorted character/frequency array and store the merged Binary merge tree.

properties	definition
Heap->(int)size	character
Heap->(Node)*array	Sorted array

3. Heap Sort:

a. Abstract:

Heap sort is a sorting method derives from Heap tree. With heap-representation of the array, heap sort can easily sort input array by *Heapify*, which is an algorithm that maintain heap-property on specific node.

This method of Heapify is slightly different from the original one. It constantly Heapify the child item of the inserted node (i). The process is similar to the old one that it constantly get the heaped first item and

Heapify the rest of them.

b. Algorithm:

```
// To enforce min heap property for n-element Heap with root i.
// Input: size n max heap array Heap->array, root i
// Output: updated Heap.
Algorithm Heapify(list, i, size)
{
    s := i;
    left := 2 * i + 1;
    right := 2 * i + 2;

    if (left < size && list[left]->freq < list[s]->freq) {
        s := left;
    }
    if (right < size && list[right]->freq < list[s]->freq) {
        s := right;
    }
    if (s != i) {
        t := list[s];
        list[s] := list[i];
        list[i] := t;
        Heapify(list, s, size);
    }
}
```

```
Algorithm HeapSort(list, n)
{
    for i := (n - 1) / 2 to 0 step -1 do  {
        Heapify(list, i, size);
    }
}
```

c. Time Complexity:

For the Heapify process, it traverse and heapify until the input node is a leaf node, therefore the time complexity will be O(lg n). For the HeapSort process, it does Heapify for n/2 times, therefore the total time complexity will be n/2 * O(lg n) equals **O(n lg n)**.

d. Space Complexity:

The algorithm uses several integers and Node(t) and a Heap array list[n]. **The space complexity would be O(N).**

4. Binary Merge Tree, BMT():

a. Abstract:

BMT() gets the least two element, which are characters that has least two frequencies, and merge them into one element with frequencies added.

b. Algorithm:

```
// Generate binary merge tree from list of n files.
// Input: int n, list of files
// Output: optimal merge order.
Algorithm BMT(n, list)
{
    for n > 0 do {
        pt := new node ;
        pt -> lchild := GetMin(list) ; // Find and remove min from list.
        pt -> rchild := GetMin(list) ;
        pt -> w := (pt -> lchild) -> w + (pt -> rchild) -> w ;
         while (n && pt -> freq < list[(n - 1) / 2]->freq) {
            list[n] =list[(n - 1) / 2];
            n := (n - 1) / 2;
         }
        list[n] := pt;
    }
    return GetMin(list) ;
}

Algorithm GetMin(list)
{
    temp := list[0];
    list[0] := list[size - 1];
    size--;
    Heapify(list, 0, size);
    return temp;
}
```

c. Time complexity:

For GetMin(), there is one Heapify in it, therefore it's time complexity will be O(lg n).

For BMT(), there is GetMin and a while loop with O(lg n) frequency (since it iterates with n, n/2 n/4 … etc). The time complexity for BMT() will be **O(n lg n)**.

d. Space Complexity:

The algorithm uses several integers and Nodes(temp) and a Heap array list[n]. **The space complexity would be O(N).**

5. **Huffman Tree Traversal, PrintCode():**

a. Abstract:

PrintCode() prints each input character's Huffman Code by traversing the Huffman Tree.

b. Algorithm:

```
// Generate Huffman Code from Heap Tree.
// Input: Node Node, int code[], int top
// Output: Huffman Codes
Algorithm PrintCode(Node, code, top)
{
    if Node is a leaf do {
        for i := 0 to n do {
            print(code[i]);
        }
    }
    if (node->left) {
        code[top] = 0;
        printCodes(node->left, code, top + 1);
    }
    if (node->right) {
        code[top] = 1;
        printCodes(node->right, code, top + 1);
    }
}
```

c. Time Complexity:

The printing process prints n character’s Huffman Code and traverse the Huffman tree. The time complexity would be **O(n)**.

d. Space Complexity:

The algorithm uses several integers, code array and a Heap tree. **The space complexity would be O(N).**

6. Overall Algorithm, Huff():

a. Abstract:

Gathering all elements in the above analysis, we can construct an algorithm to implement Huffman Code Encoding.

b. Algorithm:

```
// Generate Huffman Code from Heap Tree.
// Input: Node Node, int code[], int top
// Output: Huffman Codes
Algorithm Huff(data, freq, size)
{
    for i := 0 to size do {
        HeapTree->array[i] := newNwode(data[i], freq[i])
    }

    HeapTree = HeapSort(HeapTree, size);
    root = BMT(HeapTree);
    PrintCodes(root);
}
```

c. Time Complexity:

The time complexity for Huff() will be:

O(n) for initializing.

O(n lg n) for Heap Sorting.

O(n lg n) for setting Binary Merge Tree.

O(n) for printing Huffman Codes.

The total time complexity is **O(n lg n).**

d. Space Complexity:

The algorithm uses the spaces those algorithms I analyzed above takes. **The space complexity would be O(N).**

7. Time & Space:

	Huff()
Time complexity	**O(N lg N)**
Space complexity	**O(N)**

Implementation:

1. Result:

	ASCII (Bytes)	Huffman (bits)	Huffman (Bytes)	Ratio (%)
talk1.txt	11949	53830	6729	56.3143
talk2.txt	17654	79601	9951	56.3668
talk3.txt	15944	70812	8852	55.5193
talk4.txt	11014	50144	6268	56.9094
talk5.txt	11802	53813	6727	56.9988

Average Ratio: 56.42172 %

Observation:

1. Ratio between storing ASCIIs and Huffman codes:

Using Huffman encoding to store a list of characters saves **56.42172 %** of the space needed comparing with storing ASCII.

Conclusions:

1. It takes Heap Sort, Binary Merge Tree to implement a Huffman encoding process.
2. Time and space complexities of *Huff()*:

	Huff()

Time complexity	**O(N lg N)**
Space complexity	**O(N)**

3. The average ratio between storing ASCIIs and Huffman codes is 56.42172 %.