**Advanced Data Structures and**

**Algorithm Analysis**

**Laboratory Projects**

**Huffman Codes**

**Author Names**

**Date: yyyy-mm-dd**

**Chapter 1: Introduction**

Huffman coding is a sort of encoding algorithm which apply variable word length when encoding. Usually it is implemented with binary tree, which brings some kind of uncertainty to the final morphology since sometimes whether a tree is a left subtree or right subtree does not matter, thus causes some difficulty for professors’ judgement. The task is to design a program to determine whether a binary tree is the right implementation of Huffman encoding.

**Chapter 2: Data Structure / Algorithm Specification**

The general idea to solve this problem is :

First we calculate the cost of the correct Huffman encoding tree, and then compare it with the costs of the samples, during which those with a larger cost will be judged to be wrong implementation of Huffman codes.

Second, we have a traverse in all the binary codes and compare them in pairs. If one of them happens to be the prefix of another one, this implementation will be judged to be wrong.

In this project we use several data structures to judge whether the given Huffman encoding is correct.

First we use a minimum heap to calculate the total cost of correct Huffman encoding. And the corresponding data structure in c++ is priority\_queue. This process is expressed in pseudo code as below:

1. **procedure** calculate\_cost(priority\_queue):
2. cost := 0
3. **while** priority\_queue **is** **not** empty:
4. cost\_left := priority\_queue.pop()
5. **if** priority\_queue **is** empty:
6. break
7. cost\_right := priority\_queue.pop()
8. cost:= cost + cost\_left + cost\_right
9. priority\_queue.push(cost\_left + cost\_right)
10. return cost(cost **is** the total cost **of** the correct Huffman codes)

Other than that, we also use the “map” structure, which is also known as “dictionary” , to construct connect between the character and its frequency.

In the original version, we shall use two nested for loops to check whether a code happens to be the prefix of another one by comparing from byte to byte. In the alternative version, we shall use another important data structure, the prefix tree, to handle this problem. The data structure is shown as below:

1. **struct** trie
2. {
3. **bool** is\_wordend = **false**;
4. **int** num = 0;
5. **struct** trie \*next[2];
6. };

The bool variable “is\_wordend” is used to denote that this node is the end byte of some binary code, and the array “next” will store the address of the 2 subtrees. The variable “num” is used to denote how many binary codes have the string corresponding to the path from the root to the current node to be the prefix. For example, for the current node, if the path from root here is

Then the variable “num” shows the number of binary codes with “011” to be the prefix. And we can use the process below to know if the Huffman codes is valid:

1. **procedure** insert(trie \* root, **string** s):
2. has\_invalid\_node := **false**
3. **for** i **in** 0 **to** s.length - 1:
4. index := s[i] - '0'
5. **if** node->next[index] **is** NULL:
6. node->next[index] = new\_node
7. node := node->next[index]
8. node->num++
9. **if** i == s.length - 1:
10. node->is\_wordend := **true**
11. **if** node->num > 1 **and** node->is\_wordend:
12. has\_invalid\_node = **true**
13. return has\_invalid\_node
15. **procedure** check\_huffman\_codes(**string** codes[num\_of\_characters]):
16. root := new\_node
17. is\_valid\_codes := **true**
18. **for** i **in** 0 **to** num\_of\_characters - 1:
19. **if** insert(root, codes[i]) == **true**:
20. is\_valid\_codes = **false**
21. return is\_valid\_codes

With this process, we can easily check whether the given Huffman codes is valid.

**Chapter 3: Testing Results**

case 1.

purpose: correctness test for multiple solutions

input:

3

A 1 B 1 C 1

2

A 00

B 1

C 01

A 0

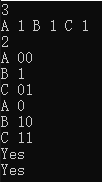
B 10

C 11

output:

Yes

Yes



state: pass

case 2.

purpose: correctness test for incorrect WPL

input:

3

A 1 B 2 C 3

2

A 00

B 1

C 01

A 0

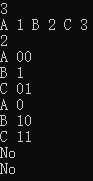
B 10

C 11

output:

No

No



state: pass

case 3:

purpose: correctness test for ambiguous coding input

input:

3

A 1 B 2 C 3

2

A 00

B 01

C 0

A 00

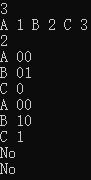
B 10

C 1

output:

No

No



state: pass

case 4:

purpose: correctness test for large input(n = 63, m = 1000)

input:

*Attached in "..\test\test.in" file.*

*The input generator source code is attached in "..\test\test.cpp" file.*

output:

*(1000 Yeses)*

state: pass

We have also test this program on PTA, and the result shows that it has a good performance.



**Chapter 4: Analysis and Comments**

Space Complexity:

We use a priority queue of size *n* to store the raw frequency input. The Space Complexity is *O*(*n*).

Time Complexity:

We use the trie to check if one code is the prefix of another. The insert operation costs time *O*(*w*) where *w* is the maximum length of codes. Because the length of codes is no more than 63, the insert operation costs time *O*(1).

We use priority queue to calculate the WPL. The pop and push operations both cost time *O*(log *n*). Hence, the calculation of WPL cost time *O*(*n*log *n*).

For each case, we create a trie to check prefixes. The process costs time *O*(*n*log *n*) due to *n* insertions. For *m* cases, the time complexity of the whole process is *O*(*mn*log *n*).

In conclusion, the time complexity is *O*(*mn*log *n*).

**Appendix: Source Code (if required)**

Please open the encodes in “..\code\Huffman\_code.cpp” and “..\code\trie.cpp” to view the codes.

**Declaration**

***We hereby declare that all the work done in this project titled "Huffman Codes" is of our independent effort as a group.***