ADS Project-5
Huffman Codes

Author Names

Programmer: 王钟毓

Tester: 彭子帆

Reporter: 陈宇威

Date: 2019-05-06

I. Introduction

1.1 Problem Description

Huffman code is a very classic method to cut down the memory spends which can be easily realized by Greedy method.

We always pop out two nodes with the largest weight and establish a new tree. And we point its left pointer to the larger one while right pointer to the smaller one. This process can lend the help of heap, as a data structure.

In this problem, we are required to examine whether the input of student is correct. Two factors contribute to the correctness of different expression of Huffman Tree. The first is the uniqueness of prefix Huffman code. And the second is the overall weight of the whole tree should be same. Thus, we can simply go through the student input and calculate whether it's true and the reestablish Huffman Tree is unnecessary at all.

II. Algorithm Specification

The pseudo-code of solving this problem contains two parts. Above all, we need to establish a standard tree. Then we need to calculate its weight. Finally, we need to examine all the input and output the correctress of students' answers.

The pseudo-code of each functions is listed as follows.

2.1 Establish a Huffman Tree

```
1.   for(i = 0;i < Numberofnodes;i ++)
2.      initialize(Newnode);
3.      push Newnode into heap
4.  for(i = 0;i < Numberofnodes - 1;i ++)
5.      leftnode = heap.pop();
6.      rightnode = heap.pop();
7.      initialize(Newnode);
8.      point the left pointer to leftnode;
9.      point the right pointer to rightnode;
10.     push Newnode innto the heap
```

2.2 Calculate the Weight

```
1.   int CalculateWeight(Node* p,depth)
2.  {
3.
4.      if(p is a leaf)
5.          return depth*(p->weight);
6.      else
7.          return (CalculateWeight(p->left, depth+1) + CalculateWeight(p->right
    , depth+1));
8.  }
```

2.3 Examine the input

```
while(currentnode != input.end())
{
    weight = 0;
    weight += length(huffcode) * the weight of huff node;
    if(currentnode is the prefix of other nodes)
        raise False;
}
if(weight == cost_of_standard_tree)
    raise True;
else
    raise False;
```

2.4 Postscripts

There may be some differences between the pseudo-code given above and the actual program. But the whole algorithm is more or less the same.

III. Testing Results

3.1 The largest cases

Due to the size of input is too large that will affect the outlook of the whole report. The input file has been saved in the directory which named *Max_input_Unicode16.*

3.2 The smallest cases

The input file has been saved in the Test directory which named *Min_input_Unicode16.*

3.3 The medium cases

The input file which contains three parts has been saved in the Test directory which named *Meidum_input1_Unicode16*、*Meidum_input1_Unicode16*、*Meidum_input1_Unicode16.*

IV. Analysis and Comments

When it comes to the analysis and comments of the program, the time complexity is mainly the cost of examine procedure which takes up for O(N^2).Although the time complexity for heap building and tree traverse is O(log N), the time complexity is no doubt O(N^2). Because each time we need examine whether the node is the prefix of other nodes which need to traverse through all nodes already saved in the vector. And it needs load in the input which is another loop. Thus, two nested loops contribute to the outcome with O(N^2).

The space complexity mainly comes from the cost of vector and nodes saved in the computer. Both of them take up O(N) space in the computer. The nodes is each struct parameters that saves weight, pointer and character while vector saves each input to make it eaiser to examine.

V. Appendix

```
#include <iostream>
#include <queue>
#include <vector>
#include <map>
using namespace std;
struct node // intialize the node
{
    int weight; // the weight of each character
    char character; // character of the node
    node* left; // point to the left
    node* right; // point to the right
    node(int a = 0, char b = '*', node* c = nullptr, node* d = nullptr):
    weight(a),character(b),left(c),right(d){}
};
// overload cmp function
bool operator <(const node &a,const node &b)
{
    return a.weight > b.weight;
}

int CalculateWeight(const node*p,int depth)
{

    if(p->left == nullptr && p->right == nullptr)
    {
        return depth*(p->weight);
    }
    else
        // calculate the weight of left node & right node
        return (CalculateWeight(p->left, depth+1) + CalculateWeight(p->right
, depth+1));
}


int main()
{
    // use priority_heap to simulate the heap
    // number refers the number of huffman nodes
    // use map to save the relationship between char & int
    priority_queue<node>heap;
    int number;
    cin >> number;
```

```
    char character;
    int weight;
    map<char,int>table;
    // input and create the huffman tree
    for(int i = 0;i < number; i++)
    {
        cin >> character >> weight;
        table[character] = weight;
        heap.push(node(weight,character));
    }
    for(int i = 0;i < number - 1;i ++)
    {
        // the first pop returns the node with current largest weight
        // use copy constructor for the top element
        // because it will be deleted soon
        auto left = new node(heap.top());
        heap.pop(); // delete the node at the top of tree
        auto right =new node(heap.top());
        heap.pop();
        heap.push(node(left->weight+right->weight,'*',left,right));
    }
    // cost refers to the total cost the heap
    int cost = CalculateWeight(&heap.top(), 0);

    // the numberinput refers to the number of loops to be examined
    int numberinput;
    cin >> numberinput;
    vector<string>huff;
    // flag saves whether exists one node that is prefix of other node
    int flag = 1;
    for(int i = 0;i < numberinput; i++)
    {
        char tempchacter;
        int tempcost = 0;
        // loop to examine all the input
        for(int j = 0;j < number; j++)
        {
            string huffcode;
            cin >> tempchacter >> huffcode;
            // the first element should be saved in the huffcode
            if (huff.size() > 1)
            {
                // loop the vector & test all the nodes in the hufftree
                for(int k = 0;k < huff.size();k ++)
```

```
86.                 {
87.                     int z;
88.                     // always compare node with the smaller length to larger
    one
89.                     if(huff[k].length() > huffcode.length())
90.                     {
91.                         for(z = 0;z < huffcode.length();z ++)
92.                             if(huff[k][z] != huffcode[z])
93.                                 break;
94.                         if(z == huffcode.length())
95.                             flag = 0;
96.                     }
97.                     else
98.                     {
99.                         for(z = 0;z < huff[k].length();z ++)
100.                            if(huff[k][z] != huffcode[z])
101.                                break;
102.                        if(z == huff[k].length())
103.                            flag = 0;
104.                    }
105.                }
106.            }
107.            // push it to the vector
108.            huff.push_back(huffcode);
109.            // save it in the tempcost
110.            tempcost += huffcode.length() * table[tempchacter];
111.        }
112.        // the cost of two tree is the same & no prefix is the same
113.        if(tempcost == cost && flag)
114.            cout << "Yes" << endl;
115.        else
116.            cout << "No" << endl;
117.        huff.clear();
118.        flag = 1;
119.    }
120. }
```