Project5
Huffman Tree

Date: 2022.5.4

1.Introduction

Using Huffman encoding can greatly save the space required for string storage, but Huffman encoding may not be unique for the same string. In this project, we will programmatically determine whether the encoding submitted by the students for a given string is the correct Hoffman encoding.

The students are given a question consisting of an integer `N`, followed by `N` different characters and their frequencies, and `M` students will submit their own encoding for each of the `N` characters. The program will determine the encoding submitted by the students, then output *'Yes'* if *WPL(Weighted Path Length of Tree)* is the smallest and Each character encoding meets the requirements of prefix encoding, otherwise it will output *'No'*.

1.Huffman Tree

Given n weights as n leaf nodes, construct a binary tree. If the length of the weighted path reaches the minimum, such a binary tree is called an optimal binary tree, also known as a Huffman tree.

2.WPL

As known as Weighted Path Length of Tree, WPL is the sum of the weighted path lengths of all leaf nodes in the tree.

3.Prefix encoding

The encoding of any character is not a prefix of any other character encoding.

2.Algorithm Specification

1.General idea

- Establish the min-heap of weight value according to the input
- Establish a Huffman tree according to the min-heap
- Calculate the best WPL, as known as **Weighted Path Length of Tree**.
- Check whether the codes input by students is correct and best for the problem, in detail:
    - Whether the length of codes is greater than or equal to **N**. Noted that the greatest length of codes is **N-1**.
    - Whether WPL value is equal to the standard value.
    - Whether the codes have prefix codes.

  To determine the prefix code， here we give two possible way to solve:

    - Imitate the Huffman Tree, build the prefix code tree for each list. When going to the next node, if it doesn't reach the end and has a weight value, we can prove that it is a

prefix code. When it reaches the end, if this final node has leaf nodes, then we can conclude that it is a prefix code.
    - Or, start with the definition of prefix code, that one of the codes with n bits should not exist in any other codes' top n bits. That is, we check each two codes in list whether they obey the upper rule. In this program, we finally choose this way to determine the prefix code.
- Finally output the result of running program.

2.Code Structure

(1) Data Structure

- The structure to store the Huffman Tree

```
typedef struct Treenode* HuffmanTree;
struct Treenode
{
    int val;
    HuffmanTree left;
    HuffmanTree right;
};
HuffmanTree create_tree()
{
    HuffmanTree T = (HuffmanTree)malloc(sizeof(Treenode));
    T->left=NULL;
    T->right=NULL;
    T->val=0;
    return T;
}
```

- We use priority queue to replace the self-construct heap, which can lead to a considerable reduction in code size

```
#include<queue>
priority_queue<HuffmanTree,vector<HuffmanTree>,cmp> heap;
struct cmp
{
    bool operator()(HuffmanTree A, HuffmanTree B)
    {
        return A->val>B->val;
    }
};
```

(2) Functions

- The `main` function.

```
int main()
{
    ...
    /* omitted codes are specified above in the General Idea part */
    return 0;
}
```

- As mentioned above, we will establish a **priority queue** instead of a self-construct Minheap based on **Huffman Tree**. We select 2 **Huffman Tree** with the least frequency, and build a binary tree, then insert the new tree into the **priority queue**. Repeat this operation for N-1 times(N elements in total).

```
HuffmanTree build()
{
    HuffmanTree T;
    for(int i=1;i<n;i++)
    {
        T=create_tree();
        T->left=heap.top();
        heap.pop();
        T->right=heap.top();
        heap.pop();
        T->val=T->left->val+T->right->val;
        heap.push(T);
    }
    T=heap.top();
    heap.pop();
    return T;
}
```

- Calculate the WPL value from Huffman Tree established by ourselves and compare with those submitted by students. Realized by Recursion.

```
int WPL(HuffmanTree root, int depth)
{
    if(root->left||root->right)
    return WPL(root->left,depth+1)+WPL(root->right,depth+1);
    else return root->val*depth;
}
```

- Judge whether the codes of students satisfied the conditions of prefix codes, that is: there is no ambiguity problem.

```
for(int i=0;i<m;i++)
    {
        char ch;
        int flag=0;
        char str[70][70];
        int wgh=0;
        for(int j=0;j<n;j++)
        {
            cin>>ch>>str[j];
            wgh+=strlen(str[j])*f[j];
        }
        if(wgh!=length)
        {
            cout<<"No"<<endl;
            continue;
        }
        for(int j=0;j<n;j++)
```

```
        for(int k=j+1;k<n;k++)
        {
            int flag2=0;
            int len1=strlen(str[j]);
            int len2=strlen(str[k]);
            for(int l=0;l<len1&&l<len2;l++)
            if(str[j][l]!=str[k][l]){flag2=1;break;}
            if(!flag2)flag=1;
        }
        if(flag)cout<<"No"<<endl;
        else cout<<"Yes"<<endl;
    }
```

3.Test Result

The main test data is supported by PTA online judge.

[题目详情 - 7-9 Huffman Codes (pintia.cn)]

Here just show the sample test data.

Input

```
7
A 1 B 1 C 1 D 3 E 3 F 6 G 6
4
A 00000
B 00001
C 0001
D 001
E 01
F 10
G 11
A 01010
B 01011
C 0100
D 011
E 10
F 11
G 00
A 000
B 001
C 010
D 01
E 100
F 101
G 110
A 00000
B 00001
C 0001
D 001
E 00
F 10
G 11
```

Output/Analysis

Output	Analysis
Yes	WPL=Standard Length, No prefix code, No longer than N
Yes	WPL=Standard Length, No prefix code, No longer than N
No	WPL != Standard Length
No	Exist prefix code

Further test

2022/05/03 22:35:27 答案正确 30 编程题 C++ (g++) 41 ms

测试点	提示	结果	分数	耗时	内存
0	sample 有并列、多分支，有长度错、长度对但是前缀错；仅英文大写字符	答案正确	16	5 ms	324 KB
1	小写字母，01反、且2点对换；有2点重合	答案正确	7	5 ms	456 KB
2	几组编码不等长，都对；等长但前缀错误；code长度超过N	答案正确	3	5 ms	324 KB
3	最大N&M，code长度等于63	答案正确	1	41 ms	328 KB
4	最小N&M	答案正确	1	5 ms	452 KB
5	编码的字符是双数个，而提交采用的是等长编码。卡仅判断叶结点和度的错误算法	答案正确	1	5 ms	452 KB
6	非Huffman编码，但是正确；没有停在叶子上	答案正确	1	5 ms	316 KB

4.Analysis and comment

1.Time Complexity

(1)Huffman Tree Build:

Given N tree node， and the `build()` function is suppose to do the loop N times. In each loop, we pop two element from priority queue which cost O(1) and push the new build tree node into the priority queue which cost O(log N). The two are product relations: therefore, the time complexity of building a Huffman tree is O (N*log N).

(2)Check Code:

Getting WPL cost O(N), so just ignore it.

In each list of code (in this problem, it's suppose to give M lists), we iterate all pairs of codes in lists, which cost O(N*N). And in each check, we will search all bits of the shorter code in the worst case, which cost averagely O(log N). So the total time complexity cost is O(N^2 *log N) which is not so good.

If we follow the first way to check the prefix codes, we just build another pseudo-Huffman Tree,and we build it in O(N*log N) similar to the Huffman Tree Build. It's suppose to be a better way, but our code about it possibly have some bug, which can't pass all the test point in OJ platform. We attach the code in note.

2.Space Complexity

To store the Huffman Tree, we can find that originally there are N tree nodes. Each time we pop two elements and combine them, which add one node in Huffman Tree. And it executes for N times. So it's suppose to have 2*N nodes in Huffman Tree, with space complexity O(N).

We can reuse the room in each list, so the space complexity in this stage is O(N*N). (N codes with N bits)

3.Further Comment

Regret to not realize the first way to check the prefix code, and not too much to say as it's not a very hard problem.

5.Appendix

```
#include<iostream>
#include<algorithm>
#include<cstdio>
#include<queue>
#include<cstring>
using namespace std;
const int maxn=400;
const int INF=2147483647;
int n,m,length;
char c[maxn];//store the character
int f[maxn];//store the frequency (also the weight)
typedef struct Treenode* HuffmanTree; //structure
struct Treenode
{
    int val;
    HuffmanTree left;
    HuffmanTree right;
};
struct cmp//used for priority queue
{
    bool operator()(HuffmanTree A, HuffmanTree B)
    {
        return A->val>B->val;//put the node with smallest key value to top
    }
};
```

```
HuffmanTree create_tree()//tree node initialize
{
    HuffmanTree T = (HuffmanTree)malloc(sizeof(Treenode));
    T->left=NULL;
    T->right=NULL;
    T->val=0;
    return T;
}
priority_queue<HuffmanTree,vector<HuffmanTree>,cmp> heap;//declare of the
priority queue, replace the heap

////////////////////////////////////////////////
// build():
// We select 2 Huffman Tree with the least frequency,
// and build a binary tree, then insert the new tree into the priority queue.
// Repeat this operation for N-1 times (N elements in total).
////////////////////////////////////////////////
HuffmanTree build()
{
    HuffmanTree T;
    for(int i=1;i<n;i++)
    {
        T=create_tree();
        T->left=heap.top();
        heap.pop();
        T->right=heap.top();
        heap.pop();
        T->val=T->left->val+T->right->val;
        // the key value is the left subtree value plus right subtree value
        heap.push(T);
    }
    T=heap.top();
    heap.pop();
    return T;
}
int WPL(HuffmanTree root, int depth)
{
    if(root->left||root->right)//Not leaf
    return WPL(root->left,depth+1)+WPL(root->right,depth+1);
    else return root->val*depth;//reach leaf, return the weight multiply depth
}
int main()
{
    cin>>n;
    for(int i=0;i<n;i++)
    {
        cin>>c[i]>>f[i];
        HuffmanTree T=create_tree();
        T->val=f[i];
        heap.push(T);
        //Huffman Tree list initialize
    }
    HuffmanTree ans=build();
    int length=WPL(ans,0);
    cin>>m;
```

```
    for(int i=0;i<m;i++)
    {
        char ch;
        int flag=0;
        char str[70][70];
        int wgh=0;
        for(int j=0;j<n;j++)
        {
            cin>>ch>>str[j];
            wgh+=strlen(str[j])*f[j];
            //input the code list and calculate the WPL at the same time
        }
        if(wgh!=length)
        {
            cout<<"No"<<endl;
            continue;
        }
        for(int j=0;j<n;j++)
        for(int k=j+1;k<n;k++)
        {
            int flag2=0;
            int len1=strlen(str[j]);
            int len2=strlen(str[k]);
            //str[j],str[k]:the chosen two code to compare
            for(int l=0;l<len1&&l<len2;l++)
            if(str[j][l]!=str[k][l]){flag2=1;break;}//if there exist one bit
distinct, it's legal; else it's prefix code
            if(!flag2)flag=1;
        }
        if(flag)cout<<"No"<<endl;//if exist prefix code, it's not legal
        else cout<<"Yes"<<endl;
    }
    /*
    //the following code is the prefix code check completed by the firsy way,
mentioned in report
    for(int i=0;i<m;i++)
    {
        char ch;
        char str[70];
        bool flag=0;
        int test_WPL=0;
        HuffmanTree test=create_tree();
        for(int l=0;l<n;l++)
        {
            cin>>ch>>str;
            HuffmanTree branch=test;
            int weight;
            int len=strlen(str);
            if(len>n){flag=1;break;}
            for(int k=0;k<len;k++)
            {
                if(str[k]=='0')
                {
                    if(!branch->left)branch->left=create_tree();
                    branch=branch->left;
```

```
                }
                else if(str[k]=='1')
                {
                    if(!branch->right)branch->right=create_tree();
                    branch=branch->right;
                }
                if(branch->val)
                {
                    flag=1;
                    break;
                }
            }
            if(branch->left||branch->right)flag=1;
            if(flag)break;
            branch->val=f[l];
            test_WPL+=f[l]*len;
        }
        if(!flag&&test_WPL==length)cout<<"Yes"<<endl;
        else cout<<"No"<<endl;
    }*/
    return 0;
}
```

6.Declaration

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