浙江大学

Advanced Data Structures and Algorithm Analysis

Laboratory Projects

Huffman Codes

Group 15

聂俊哲 张琦 张溢弛

Date: 2020-04-20

Content

Chapter 1: Introduction

1.1 Problem Description

1.1.1 Introduce

- In this project, we will write a program to judge whether the Huffman codes designed by the students are proper answers or not because the Huffman codes are not unique at most of the time, we have find a different way to check the correctness of the students' hand-in answers instead of list all the probable results of a given situation of several characters with their frequency.

1.1.2 Input format

- The input format in this project is including the following things
    - A integer n, which means the number of characters.
    - n characters and their frequency in pairs.
    - A integer m, which means the number of hand-in answers from students.
    - m groups of hand-in answers from students, each contains n pairs of character and its Huffman code.

1.1.3 Output format

- We will check whether the hand-in answer is correct or not and output a "Yes" or "No" to show the result of our program.

1.1.4 Something to note:

- There are some traps in this topic
    - The Huffman codes are not unique
    - The hand-in results from students might be unordered.

1.2 Background data structures and algorithm

1.2.1 Huffman tree

- Huffman tree is a binary prefix tree which store the characters in its leaf nodes. The Huffman code of a character is corresponding string on the Huffman tree of the data
- The total length of compressed characters is $\sum_{i=1}^{n} L_i f_i$ where L_i is the length of the Huffman code of the i-th character and f_i is the frequency.
- Such a total length is a **smallest result** of the characters with their frequency, and any possible result of the Huffman code has the same total length. The other result of coding is larger than the smallest result.
- A Huffman tree can be implentmented like this

```
typedef struct node
{
        char name;
        int weight, parent;
        int lchild, rchild;
        node():name(), weight(0), parent(0), lchild(0),rchild(0){}
}HTNode,*HuffmanTree;
```

we have implentment the data structure in the test program, and in the main program, we use a more simple method using the min heap to simulate a Huffman tree rather than directy build a Huffuman tree.

1.2.2 Min Heap

- Min heap is a special tree-based data structure that the parent node is less than its non-empty children, but the siblings doesn't have any special relations. It can help us find the highest priority node.
- In this project Huffman code, we will use heap to get the lowest frequency characters. And we could use STL in C++ to create a min heap for us directly, which is in head file <queue>
- We will use such an algorithm in the PPT of class08 to build the min heap and calculate the total weight of the tree.
    - The time complexity is $O(NlogN)$, we will prove the theory in the Part 4.

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{   consider the C characters as C single node binary trees,
     and initialize them into a min heap;
     for ( i = 1; i < C; i++ ) {
        create a new node;
        /* be greedy here */
        delete root from min heap and attach it to left_child of node;
        delete root from min heap and attach it to right_child of node;
        weight of node = sum of weights of its children;
        /* weight of a tree = sum of the frequencies of its leaves */
        insert node into min heap;
   }
}
```

1.2.3 Greedy Algorithm

- Greedy algorithm works in optimizing problems, it choose the choice that can get most benefits. But this algorithm works if and only if the local optimum is equal to the global optimum
- In this project, we could use greedy algorithm because Huffman code problem is guaranteed to be a optimal solution.

Chapter 2: Data Structure / Algorithm Specification

2.1 Data Structure and Function

2.1.1 Two important function

```
//The function to check whether some of the code become prefix of others
bool check_prefix(vector<string> code);
//The function to calculate the smallest sum of weight in the Huffman tree
int calculate_weight(priority_queue<int,vector<int>,greater<int>>& huffman_tree);
```

2.1.2 The heap structure definition

```
priority_queue<int,vector<int>,greater<int>> huffman_tree;
```

- Using the STL priority_queue to simplify the code

2.2 Algorithm Specification

2.2.1 pseudo-code of the whole program

- The **pseudo-code of our program** is as following

```
Algorithm:Project 4: Huffman Codes
Input: The input format in the 1.1.2
Output: m results of the checking result "Yes" or "No"

read in characters and their frequency;
build_min_heap();
min_length=calculate_weight(min_heap H);
read in m;
for(i=0;i<m;i++)
{
    total_length=0;
    for(j=0;j<n;j++)
    {
        read in character and code;
        total_length+=frequency[character]*code.length();
    }
    if(total_length==min_length&&check_prefix)
    {
        printf("Yes\n");
    }
    else
    {
        printf("No\n");
    }
    return 0;
}
```

2.2.2 The important algorithm of the two functions

- Function 1: calculate_weight

```
function calculate_weight(H)
    result:=0
    while(true)
    {
        weight:=H.top()
        H.pop()
        if(H.size()==0)
            break
        weight+=H.top()
        H.pop()
        H.push(weight)
```

```
        result+=weight
    }
    return result
```

- Function 2: check_prefix

```
function check_prefix(code)
    sort(code.begin(),code.end())
    for i=0 to code.size() do
        len:=code[i].size()
        for j=i+1 to code.size() do
            sub_prefix=code[j].substr(0,len)
            if sub_prefix==code[i]
                return true
            else
                return false
```

Chapter 3: Testing Results

- We made up some test data using the test program test.cpp and other methods. You can run the test program to get some new test data. And some large-size test data

Test 1: Sample Data

- It is a bottom line of test

```
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 011
E 100
F 101
G 110
A 00000
B 00001
C 0001
D 001
E 00
F 10
G 11
Output:
```

```
Yes
Yes
No
No
```

Test 2:

- have characters of digit, '_' and letters

```
input:
7
9 1 B 1 C 1 D 3 e 3 f 6 _ 6
4
9 00000
B 00001
C 0001
D 001
e 01
f 10
_ 11
9 01010
B 01011
C 0100
D 011
e 10
f 11
_ 00
9 000
B 001
C 010
D 011
e 100
f 101
_ 110
9 00000
B 00001
C 0001
D 001
e 00
f 10
_ 11
Output:
Yes
Yes
No
No
```

Test 3:

- Have some not optimum condition

```
Input:
4
a 4 c 1 b 2 d 1
2
a 0
b 10
c 110
d 111
a 0
c 11000
d 10001
b 11111
Output:
Yes
```

```
No
```

Test 4:

- Have some code be prefix

```
Input:
7
A 1 B 1 C 1 D 3 E 3 F 6 G 6
1
A 00000
B 00001
C 0001
D 011
E 01 ---here is a prefix
F 10
G 11
Output:
No
```

Test 5:

- A smallest size of test data.

```
Input
2
a 1 b 2
2
a 0
b 1
a 1
b 10
Output:
Yes
No
```

Test 6:

- A largest size of test data

```
Input
63 characters with 50 hand-in answers
The test data is too big, you can find the data in the testdata.txt
Output
Yes
Yes
No
No
No
Yes
Yes
Yes
No
No
Yes
Yes
No
No
No
Yes
Yes
Yes
No
```

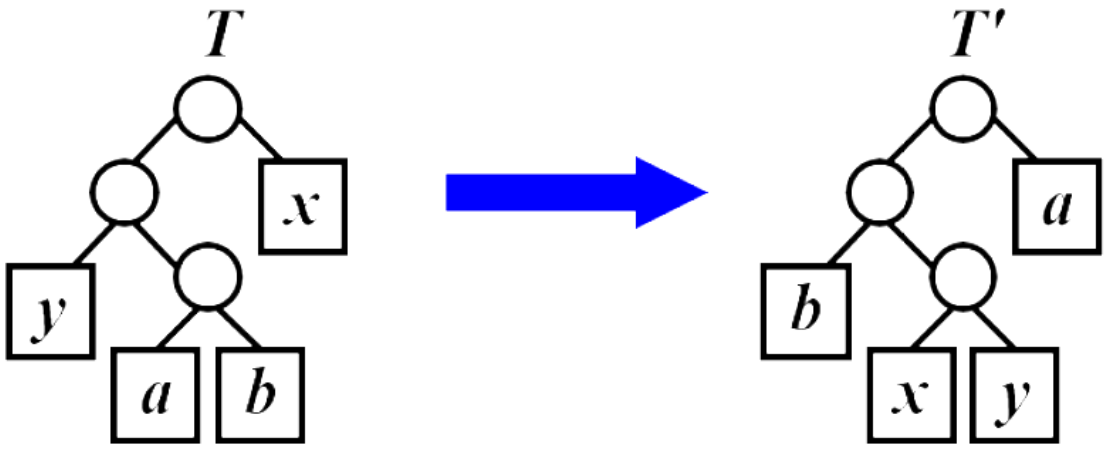
```
No
Yes
Yes
No
No
No
Yes
Yes
Yes
No
No
Yes
Yes
No
No
No
Yes
Yes
Yes
No
No
Yes
Yes
No
No
No
Yes
Yes
Yes
No
No
```

Chapter 4: Analysis and Comments

4.1 Correctness of the Greedy Algorithm

4.1.1 The Greedy choice property

- 【Lemma】Let C be an alphabet in which each character c in C has the frequency c.freq. Let x and y be two characters in C having the lowest frequencies. Then there must exist an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



4.1.2 The optimal substructure property

- Let C be a given alphabet with frequency c.freq defined for each character c in C. Let x and y be 2 characters in C with minimum frequency. . Let C' be the alphabet C with a new character z replacing x and y, and z.freq = x.freq + y.freq. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

4.2 Analysis of Min Heap

- The time complexity of the min-heap operation cost most in out program, we have serveral operations in the program
    - **Build min-heap**: It is $O(N)$ obviously to build a min heap because each insert cost $O(logN)$
    - **Calculate the total weight**: In the while loop, each time we pop 2 elements from the top of the heap(delete min) and insert 1 element into the heap. Delete min from heap and insert into heap both need $O(logN)$ each time, so each time we need $O(3logN + c)$ time. And we need do the loop N times to calculate the total weight of the Huffman tree, so the time complexity is $O(NlogN)$.
- The space complexity of the min-heap is obviously $O(N)$

4.3 Analysis of check prefix

- The time complexity is $O(N^2)$ because we need a double loop to check the prefix
- The space complexity is $O(1)$ because it needs only constant extra space.

4.4 Analysis of algorithm complexity

- Time complexity: The whole program contains several parts:
    - **Read in the data**: It needs $O(2N + MN) = O(MN)$ time complexity
    - **Heap Operations**: It needs $O(NlogN)$ according to 4.1
    - **Check each group of answer**: It has m rounds and each round need $O(N^2)$ because if the prefix checking according to 4.2, so it needs $O(MN^2)$ in total
    - All in all, the time complexity is $O(MN^2)$
- Space complexity: The whole program contains several parts:
    - Build the heap: $O(N)$
    - Store the map from character to Huffman code: $O(N)$
    - Store the result of Yes and No to output them together finally: $O(M)$
    - So the time complexity is $O(M + N)$

Appendix:

Source Code

```cpp
#include<iostream>
#include<vector>
#include<map>
#include<string>
#include<queue>
#include<algorithm>

using namespace std;

//The function to check whether some of the code become prefix of others
bool check_prefix(vector<string> code);
//The function to calculate the smallest sum of weight in the Huffman tree
int calculate_weight(priority_queue<int,vector<int>,greater<int>>&
huffman_tree);

int main()
{
    //some basic variables in the program
    int i,j,n,m,fluquency;
    char character;
    //using STL in C++ to build a min-heap for the huffman tree
    priority_queue<int,vector<int>,greater<int>> huffman_tree;
    //a map to store the weight of each character
    map<char,int> huffman_code;
    //store the result of each group of the input huffman code
    vector<int> result;
    //read in the fluquency of each character
    cin>>n;
    for(i=0;i<n;i++)
    {
        cin>>character>>fluquency;
        huffman_code[character]=fluquency;
        //build a min-heap as huffman tree
        huffman_tree.push(fluquency);
    }
    //calculate the total weight of the huffman tree
    int total_weight=calculate_weight(huffman_tree);
    // cout<<total_weight<<endl;
    //check each group of the huffman code
    cin>>m;
    for(i=0;i<m;i++)
    {
        map<char,string> temp;
        vector<string> code;
        char x;
        string y;
        int temp_weight=0;
        //read in the characters and calculate the total weight
        for(j=0;j<n;j++)
        {
            cin>>x>>y;
            code.push_back(y);
            temp_weight+=huffman_code[x]*y.length();
        }

        //a correct group of huffman code should satisfy the 2 conditions
        //1. the total weight is equal to the smallest weight
        //2. none of the huffuman code is others' prefix
        if(temp_weight==total_weight&&check_prefix(code))
        {
            result.push_back(1);
        }
        else{
            result.push_back(0);
        }
    }
```

```
    //output the result: Yes or No
    for(i=0;i<result.size();i++)
    {
        if(result[i])
            cout<<"Yes"<<endl;
        else
            cout<<"No"<<endl;
    }
    return 0;
}

//an algorithm to check if exist prefix
bool check_prefix(vector<string> code)
{
    //sort the code in a increasing order
    sort(code.begin(), code.end());
    int i,j;
    //check each huffman code whether is is others' prefix or not
    //It need O(N^2) time complexity
    for(i=0;i<code.size();i++)
    {
        int len=code[i].size();
        for(j=i+1;j<code.size();j++)
        {
            //get the substring of the first len characters of the huffman code
            string sub_prefix=code[j].substr(0,len);
            //check whether they are the same
            if(sub_prefix==code[i])
            {
                return false;
            }
        }
    }
    return true;
}

// calculate the total weight of the huffuman tree
// the result is also a smallest weight
int calculate_weight(priority_queue<int,vector<int>,greater<int>>&
huffman_tree)
{
    int result=0;
    while(true)
    {
        //using a greedy method to calculate the total weight of Huffman tree
        int weight=huffman_tree.top();
        huffman_tree.pop();
        if(huffman_tree.size()==0){
            break;
        }
        //simulation the merge of the 2 smallest nodes in the huffuman tree
        weight+=huffman_tree.top();
        huffman_tree.pop();
        huffman_tree.push(weight);
        result+=weight;
    }
    return result;
}
```

Test Program

```
#include<iostream>
#include <cstring>
#include<algorithm>

using namespace std;

```

```
typedef struct node
{
	char name;
	int weight, parent;
	int lchild, rchild;
	node():name(), weight(0), parent(0), lchild(0),rchild(0){}
}HTNode,*HuffmanTree;

typedef char **HuffmanCode;
typedef struct
{
	int weight, locate;
}TNode,*Temp;

void CreatHuffmanTree(HuffmanTree &HT, int n);
void CreatHuffmanCode(HuffmanTree HT, HuffmanCode &HC, int n, int m);
int main()
{
	HuffmanTree HT;
	HuffmanCode HC;
	char p = 'y';
	int t,n;
	while (p=='y')
	{
		cout << "Please enter the number of characters :";
		cin >> n;
		CreatHuffmanTree(HT, n);
		cout << "Please enter the number of test cases :";
		cin >> t;
		for(int m = 0; m < t; m++)
			CreatHuffmanCode(HT, HC, n , m);
		cout << "input again: y, quit: n" << endl;
		cin >> p;
	}
	system("pause");
	return 0;
}

bool compare(const TNode &x, const TNode &y) {

    return x.weight < y.weight;
}
//Generate Huffman Tree
void CreatHuffmanTree(HuffmanTree &HT, int n)
{
	int i, j, k, s1, s2, num;
	HT = new HTNode[2 * n];
	cout << "Enter the frequency of characters in format(character
frequency):"<<endl;
	for (i = 1; i <= n; i++)
	{
		cin >> HT[i].name >> HT[i].weight;
	}

	for (i = n + 1; i < 2 * n; i++)
	{
		/*Select the subscripts of the two nodes whose parent
		is 0 and the weight is the smallest from 1 ~ i-1, and
		count the number of nodes whose parents are 0 in 1 ~ i-1*/
		num = 0;
		for (k = 1; k < i; k++)
		{
			if (HT[k].parent == 0)
			{
				num++;
			}
		}
		Temp T;
		T = new TNode[num];
```

```
                for (j = 0, k = 1; k < i; k++)
                {
                        if (HT[k].parent == 0)
                        {
                                T[j].weight = HT[k].weight;
                                T[j].locate = k;
                                j++;
                        }
                }
                //sort
                sort(T,T+num,compare);
                s1 = T[0].locate;
                s2 = T[1].locate;
                HT[s1].parent = i;
                HT[s2].parent = i;
                HT[i].lchild = s1;
                HT[i].rchild = s2;
                HT[i].weight = HT[s1].weight + HT[s2].weight;
                delete T;
        }
}
//Create Huffman Code
void CreatHuffmanCode(HuffmanTree HT, HuffmanCode &HC, int n ,int m)
{
        int i, c, f,start;
        char *cd;
        HC = new char*[n + 1];
        cd = new char[n];
        cd[n - 1] = '\0';
        for (i = 1; i <= n; i++)
        {
                start = n - 1;
                c = i;
                f = HT[i].parent;
                while (f != 0)
                {
                        start--;
                        if (HT[f].lchild == c)
                        {
                                cd[start] = (m % 2)?'0':'1';
                        }
                        else
                        {
                                cd[start] = (m % 3)?'1':'0';
                        }
                        c = f;
                        f = HT[f].parent;
                }
                HC[i] = new char[n - start];
                strcpy(HC[i], &cd[start]);
        }
        delete cd;
        cout << "Haffman code:";
        for (i = 1; i <= n; i++)
        {
                cout << HT[i].name<<" "<< HC[i] << endl;
        }
        cout << endl;
}
```

References

List all the references here in the following format:

[1] 《算法导论》第三版

[2] PTA website https://pintia.cn/

[3] Course Slides of ADS

Author List

- Zhang Yichi 3180103772
- Zhang Qi 3180103162
- Nie Junzhe 3180103501

Declaration

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

Signatures