Huffman Code

3150102418 张倬豪

2017年5月15日

1 Introduction

1.1 Concepts Description

1.1.1 Huffman Code

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding and/or using such a code proceeds by means of Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted. However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

1.2 Project Description

Given the string and it's frequencies of chars. Then input a series of Huffman Codes based on the string. Our job is to determine whether the codes are correct.

Inputs and outputs are simple enough, no extra explanation is needed.

2 Algorithm Specification

2.1 Data Structure

```
struct Node{
    char c;
    int f;
} huffmanTreeNode[105]; // input chars and frequencies
struct Code{
    char ch;
    string hCode;
```

```
} huffmanTreeCode[105]; // input chars and huffman codes

priority_queue<int, vector<int>, greater<int>> queue;
// priority queue in STl, increasing order, to calculate WPL
```

2.2 Algorithm and Pseudo Code

The algorithm is simple and clear. 1. Calculate the WPL of the string.

We only need to use priority queue in STL and push all the frequencies into it, pop two elements and push back their sum. At the same time, ans increases itself with the former 2 elements' sum.

```
function getWPL(int n) begin
    int i, x ← 0, y ← 0, ans ← 0;
    priority_queue queue;
    for i ← 1 to n do begin
        queue.push(huffmanTreeNode[i].f);
    end
    while (queue is not empty) begin
        x ← queue.top();
        queue.pop();
        if(!queue.empty()) then begin
            y ← queue.top();
            queue.pop();
            queue.push(x + y);
        end
        ans += x + y;
    end
    return ans - x - y;
end
```

2. Read all the Huffman Codes and calculate their WPL

For every input case, we add up each node's WPL by finding this char in the input frequencies. Then we multiply this with the input Huffman code's size. The sum of these node's WPL is the input code's WPL.

```
function wplJudge(int n, int ans) begin
    int i, resWPL ← 0;
```

```
    for i ← 1 to n do begin
        resWPL += huffmanTreeNode[findPos(huffmanTreeCode[i].ch, n)].f * huffmanTreeCode[i].hCode.size();
    end
    return resWPL == ans;
end
```

3. Check prefix

This is a simple $O(n^2)$ algorithm. First we sort the input codes in increasing size order. We check if every other code is the sub-string of the current checking code. If there exists any match, return with 0.

```
function prefixJudge(int n) begin
    int i, j;
    sort(huffmanTreeCode + 1, huffmanTreeCode + 1 + n, cmp);
    for i ← 1 to n do begin
        string t ← huffmanTreeCode[i].hCode;
        for j ← i + 1 to n do  if(huffmanTreeCode[j].hCode.substr(0,t.size()) == t) return 0;
    end
    return 1;
end
```

4. If WPL does not match or there are no two codes which have the same prefix, return with “No”, otherwise return with “Yes”

This is simple. No more extra code.

3 Testing Results

3.1 Test Cases

1. Case 1

7
A 1 B 1 C 1 D 3 E 3 F 6 G 6

7
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

2. Case 2

7
A 1 B 1 C 1 D 3 E 3 F 6 G 6
1

7
A 00000
B 00000
C 0001
D 001
E 01
F 10
G 11

3. Case 3

7
A 1 B 1 C 1 D 3 E 3 F 6 G 6
1
A 01010
B 01010
C 0100
D 011
E 10
F 11
G 00

4. Case 4

7
A 1 B 1 C 1 D 3 E 3 F 6 G 6
1
A 00010
B 01001
C 0001
D 001
E 01
F 10
G 11

3.2 Results

1. Case 1

   Yes Yes No No

2. Case 2

   No

3. Case 3

   No

4. Case 4

   Yes

运行详情 ×

评测结果

时间	结果	得分	题目	编译器	用时（ms）	内存（MB）	用户
2017-05-15 08:24	答案正确	50	P5	g++	311	1	zju3150102418

测试点结果

测试点	结果	得分/满分	用时（ms）	内存（MB）
测试点1	答案正确	26/26	4	1
测试点2	答案正确	10/10	18	1
测试点3	答案正确	6/6	4	1
测试点4	答案正确	2/2	311	1
测试点5	答案正确	2/2	3	1
测试点6	答案正确	2/2	2	1
测试点7	答案正确	2/2	3	1

查看代码

PS: As shown in the picture below, my program passed all the tests in PTA.

4 Analysis and Comments

4.1 Time Complexity

For WPL calculating, the time complexity is $O(NlogN)$ because it's priority queue. We talked about its time complexity in the Data Structure course. For WPL and prefix check process, the time complexity are both $O(n^2)$ because clearly for each i, we may need to go through all the j in the array to find the position or check prefix. Because the case number is $O(m)$, the whole time complexity of this project is $O(m * N^2)$

4.2 Space Complexity

The space complexity is very simple, $O(n)$.

4.3 Comment

This project is not difficult. We only need to check two things of Huffman Codes. Each of them is easy if you had already handled the algorithm of Huffman Code.

5 Appendix

5.1 Source Code

```
//
//  main.cpp
//  adspj5
//
//  Created by 张倬豪 on 2017/5/13.
//  Copyright © 2017年 Icarus. All rights reserved.
//

#include<iostream>
#include<string>
#include<algorithm>
#include<queue>

using namespace std;

```

```
struct Node{
    char c;
    int f;
} huffmanTreeNode[100];

struct Code{
    char ch;
    string hCode;
} huffmanTreeCode[100];

int cmp(Code a, Code b) {
    return a.hCode.size() < b.hCode.size();
}

int getWPL(int n) {
    int i, x = 0, y = 0, ans = 0;
    priority_queue<int, vector<int>, greater<int>> queue;
    for(i = 1; i <= n; i++){
        queue.push(huffmanTreeNode[i].f);
    }
    while(!queue.empty()) {
        x = queue.top();
        queue.pop();
        if(!queue.empty()) {
            y = queue.top();
            queue.pop();
            queue.push(x + y);
        }
        ans += x + y;
    }
    return ans - x - y;
}

int findPos(char a, int n) {
    int i;
    for(i = 1; i <= n; i++) if(huffmanTreeNode[i].c == a) return i;
    return 0;
}

int wplJudge(int n, int ans) {
    int i, resWPL = 0;
```

```
    for(i = 1; i <= n; i++){
        resWPL += huffmanTreeNode[findPos(huffmanTreeCode[i].ch, n)].f * huffmanTreeCode[i].hCode.size();
    }
    return resWPL == ans;
}

int prefixJudge(int n) {
    int i, j;
    sort(huffmanTreeCode + 1, huffmanTreeCode + 1 + n, cmp);
    for(i = 1; i <= n; i++){
        string t = huffmanTreeCode[i].hCode;
        for(j = i + 1; j <= n; j++) if(huffmanTreeCode[j].hCode.substr(0,t.size()) == t) return 0;
    }
    return 1;
}

int main() {
    int n, m, rightWPL, i;
    cin >> n;
    for(i = 1; i <= n; i++){
        cin >> huffmanTreeNode[i].c >> huffmanTreeNode[i].f;
    }
    cin >> m;

    rightWPL = getWPL(n);

    while(m--) {
        for(i = 1; i <= n; i++){
            cin >> huffmanTreeCode[i].ch >> huffmanTreeCode[i].hCode;
        }
        if(!wplJudge(n, rightWPL) || !prefixJudge(n)) cout << "No" << endl;
        else cout << "Yes" << endl;
    }
    return 0;
}
```

5.2 Reference

1. Wikipedia - Huffman Code

5.3 Declaration

We hereby declare that all the work done in this project titled "Binary Search Trees" is of my independent effort.