Huffman Codes

Author Names

Date: 2022-05-03

Chapter 1: Introduction

Huffman coding, also known as Huffman coding, is a variable length coding method. This method constructs the codeword with the shortest average length of different prefix completely according to the occurrence probability(or frequency) of characters. The generation of Huffman codes is based on greedy algorithm using a full tree.

In this problem, given several characters and their frequency, we need to judge if a given coding method is Huffman code, which means it will get the smallest average length.

To solve the problem, we first build one full tree greedily to get a “standard” Huffman codes and then compare the total frequency count of the standard code and the given code. If they are the same, additionally we have to check if it’s a prefix code. Only when the two conditions: frequency equivalent and prefix, can we say that the testing case is a Huffman code.

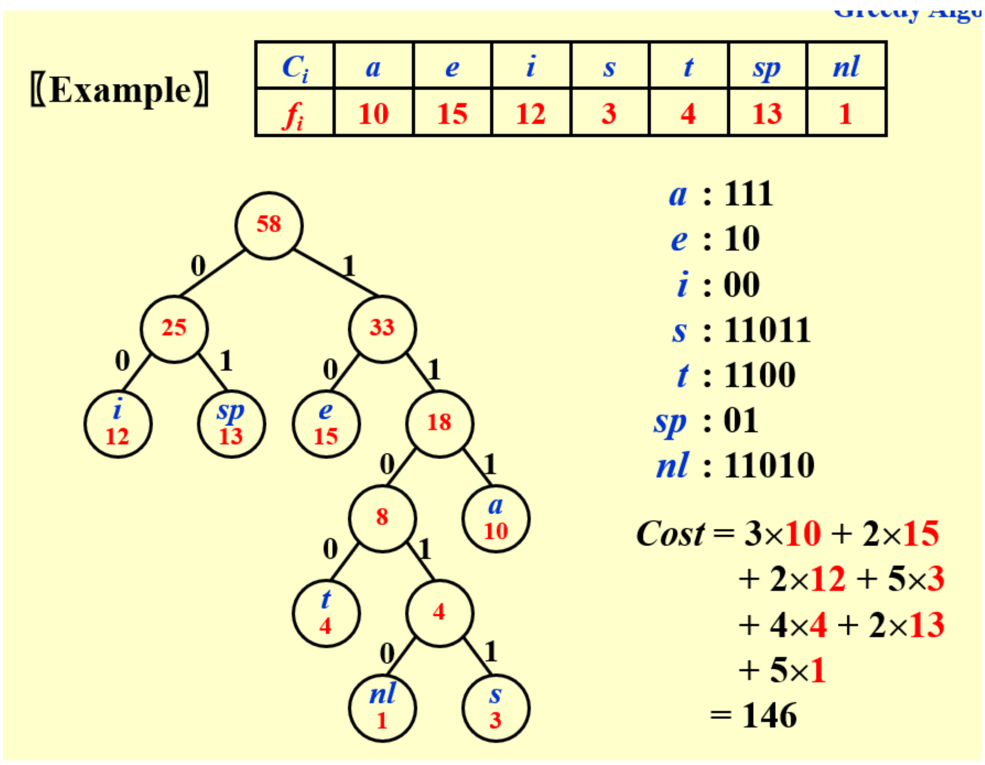
Chapter 2: Algorithm Specification

2.1 Huffman codes generation

The Huffman code is built using greedy algorithm. It has been analyzed carefully on course so we give a simple introduction.

To build Huffman code, we try to build a full tree from bottom up. We started from the characters with the lowest frequency and merge them. The parent node have the frequency property of their sum, treat as a whole new character afterwards. This is greedy because we always try to make the total cost (total length count) smallest. At the end we get a full tree, whose leaves nodes are original characters.

To get codes by the tree, we assign left edges value 0 and right ones 1, codes of a leaf node is the combination of edge value on the path. Through that we can discover the leaf nodes ensure the prefix property of Huffman codes.



【Example】

C_i	a	e	i	s	t	sp	nl
f_i	10	15	12	3	4	13	1

a	: 111
e	: 10
i	: 00
s	: 11011
t	: 1100
sp	: 01
nl	: 11010

$$Cost = 3\times 10 + 2\times 15 + 2\times 12 + 5\times 3 + 4\times 4 + 2\times 13 + 5\times 1 = 146$$

(from course PPT)

2.2 Compare the total cost (or frequency count)

In the algorithm we first build a tree and get the total frequency count. For a given coding method, a necessary and insufficient condition for it to be Huffman code is that has the equal frequency count with an arbitrary Huffman code. And what makes it to be sufficient is the prefix property (very easy to prove). So we first examine if the frequency counts are equivalent and exclude some cases.

2.3 Judge the prefix property

To ensure the coding method is legal, we need to ensure it's a prefix code. For given N characters and their frequency, we check whether one of them is the prefix of the other. If it is, we exclude the case.

This is a little time-consuming, but since most time the number of encoded characters is not very big (), for example 26 for English characters, the method is acceptable in fact.

Chapter 3: Testing Results

Testing data 1

Description:

Different characters at the same frequency.

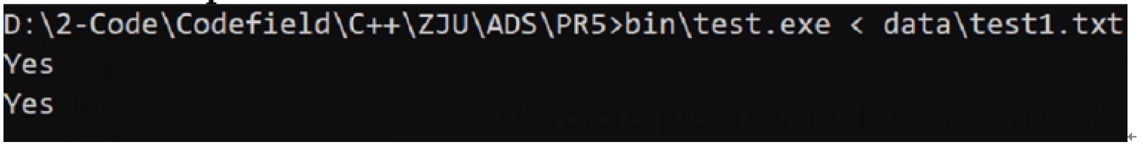
Input

```
4
A 4 B 2 C 1 D 1
2
A 0
B 10
C 110
D 111
A 0
B 10
C 111
D 110
```

Expected Result:

```
Yes
Yes
```

Actual Output:



```
D:\2-Code\Codefield\C++\ZJU\ADS\PR5>bin\test.exe < data\test1.txt
Yes
Yes
```

Testing data 2

Description:

Different characters at the same frequency.

Three kinds of characters

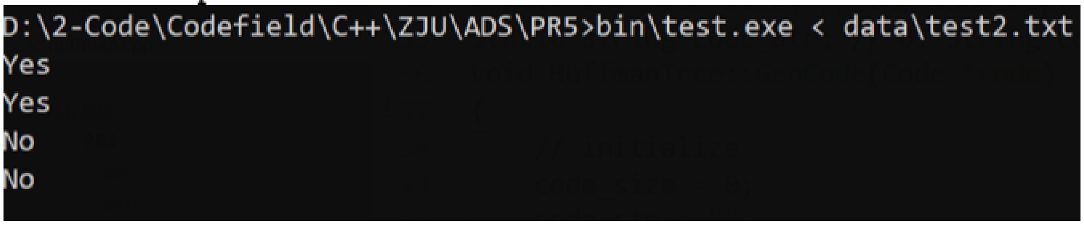
Input

```
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
```

Expected Result:

```
Yes
Yes
No
No
```

Actual Output:



```
D:\2-Code\Codefield\C++\ZJU\ADS\PR5>bin\test.exe < data\test2.txt
Yes
Yes
No
No
```

```
Testing data 3
```

Description:

Not optimum

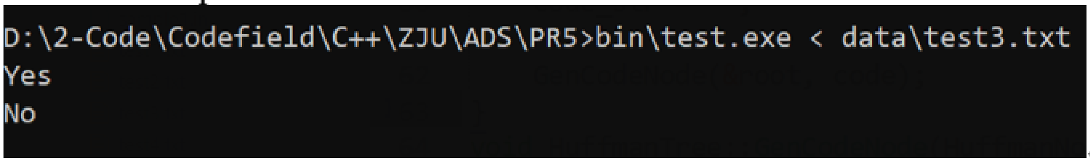
Input

```
4
a 4 b 2 c 1 d 1
2
a 0
b 10
c 110
d 111
a 0
b 10
c 1100
d 1101
```

Expected Result:

```
Yes
No
```

Actual Output:



```
D:\2-Code\Codefield\C++\ZJU\ADS\PR5>bin\test.exe < data\test3.txt
Yes
No
```

```
Testing data 4
```

Description:

One code is a prefix of another code

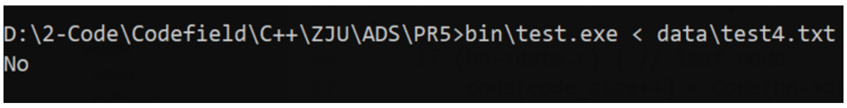
Input

```
4
a 4 b 2 c 1 d 1
1
a 0
b 10
c 1110
d 1101
```

Expected Result:

```
No
```

Actual Output:



```
D:\2-Code\Codefield\C++\ZJU\ADS\PR5>bin\test.exe < data\test4.txt
No
```

```
Testing data 5
```

Description:

The best peak shape.

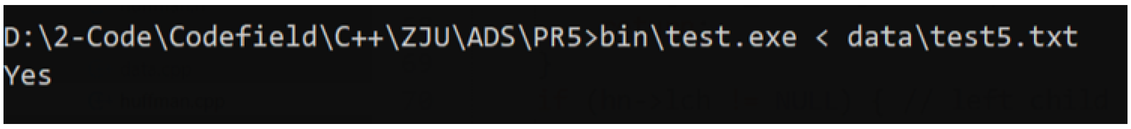
Input

```
2
a 2 b 1
1
a 0
b 1
```

Expected Result:

```
Yes
```

Actual Output:



```
D:\2-Code\Codefield\C++\ZJU\ADS\PR5>bin\test.exe < data\test5.txt
Yes
```

After testing, we can confirm the correctness of the algorithm and its ability to handle special cases.

Chapter 4: Analysis and Comments

4.1 Time complexity

Let N be the number of input characters with frequency. Since we need to sort the characters and build a tree, the time complexity for this part is O(NlogN) using advanced sorting algorithm. The procedure to judge the prefix property compare every two characters, which takes O(N^2). So the final time complexity is O(N^2).

4.2 Space complexity

For N characters the total coding length is O(NlogN), which is the major part. So it’s O(NlogN).

4.3 Potential optimization

In the procedure to judge prefix property, we may develop some better algorithm such as dynamic programming to solve it faster.

Appendix: Source Code(only main):

```
/***********************************************************************
*******
 * File name: main.cpp
 * Author:
 * Version:
 * Date: 2022-5-1
 * Description: main task
 ***********************************************************************
******/
#include <iostream>
#include <algorithm>

#include "data.h"
#include "huffman.h"

using namespace std;

const int kMaxChNum = 63; // the max number of different characters

/* for some characters and their code, judge if exists some code being
other code's prefix
 * if exists, return true, else return false
 * size is the number of characters with codes */
bool JudgePrefix(Code *code, int size);

int main(int argc, char const *argv[])
{
    int n, m;
    Character ch[kMaxChNum+5];

    // input the n characters with frequency, and sort them in ASCII
ascending order
    cin>>n;
    for (int i = 0; i < n; ++i) ch[i].Read();
    sort(ch, ch+n);

    HuffmanTree ht(ch, n); // huffman tree build from ch[]

    Code huffman_code[kMaxChNum+5];
    ht.GenCode(huffman_code); // get the code of huffman tree, store into
huffman_code[]
    sort(huffman_code, huffman_code+n); // sort in ASCII ascending order

    int huffman_freq_count;
    huffman_freq_count = CodeFreqCount(ch, huffman_code, n); // get the
total frequency of huffman code
```

```
    #ifdef DEBUG
    for (int i = 0; i < n; ++i)
cout<<"("<<huffman_code[i].c<<","<<huffman_code[i].code<<")"<<endl;
    #endif

    // deal with each student's submission
    cin>>m;
    while (m--) {
        // read and sort the code
        Code code[kMaxChNum+5];
        for (int i = 0; i < n; ++i) code[i].Read();
        sort(code, code+n);

        // two judgments:
        // 1. multiply each character's frequency and code's length, sum
them up
        //    if this total frequency is greater than huffman code's
total frequency, the submission is incorrect
        // 2. judge if some code can be another code's prefix
        //    if this situation exits, the submission is incorrect
        if (CodeFreqCount(ch, code, n) > huffman_freq_count ||
JudgePrefix(code, n)) {
            cout<<"No"<<endl;
        } else {
            cout<<"Yes"<<endl;
        }
    }

    return 0;
}

bool JudgePrefix(Code *code, int size)
{
    for (int i = 0; i < size; ++i) {
        for (int j = i+1; j < size; ++j) {
            // strl: the longer code
            // strs: the shorter code
            string strl(code[i].code), strs(code[j].code);
            if (strl.length() < strs.length()) swap(strl, strs);

            // if strs is the prefix of strl, return true
            if (strl.substr(0, strs.length()) == strs) {
                return true;
            }
        }
    }
    return false;
}
```

References

PPT on ADS course.

Author List

Declaration

We hereby declare that all the work done in this project titled "Huffman Codes

" is of our independent effort as a group.

Signatures