Huffman Code

何钦铭

April 22,2022

1 Chapter 1: Introduction

1.1 Background

Huffman coding is an entropy coding (Weight Coding) algorithm for lossless data compression. It was invented by david hoffman in 1952.
In computer data processing, Huffman coding uses variable length coding table to encode source symbols (such as a letter in a file). The variable length coding table is obtained by a method of evaluating the occurrence probability of source symbols. Letters with high occurrence probability use shorter coding, while those with low occurrence probability use longer coding, which reduces the average length and expected value of encoded strings, So as to achieve the purpose of lossless data compression.

1.2 Tasks specification

In this question, it is required to read in several letters and the corresponding occurrence frequency, and judge whether the coding of these letters by several students is the optimal coding (the coding length is the smallest). Even if it is not Huffman coding, it may be the shortest coding. The problems to be solved are as follows:

1.3 Shortest length

Calculate the shortest length of Huffman code and compare it with the length of other codes. If the length of other codes is longer than that of Huffman code, it must not be the shortest code and is rounded off.

1.4 Prefix code judgment

Even if it is verified to be the shortest coding, it is also necessary to consider the rationality of one step, that is, there will be no confusion in the process of practical use. This requires that codes with shorter length cannot be prefix codes with codes with longer length.

2 Chapter 2:Algorithm Specification

2.1 Algorithm 1

- Build a minimum heap and insert the frequency of each letter into the heap. Take out the two elements at the top of the heap each time (minimum frequency), add them and then enter the heap. If there are n elements in the minimum heap. Then the extraction operation needs n-1 times, and the last extraction of the top element is the shortest coding length

- Sort the codes proposed by each student according to their length, and then search directly to see whether the shorter code is a prefix code of a longer code.

2.2 algorithm 2

- Because the time complexity of violent search is too high, a new scheme of judging prefix code is needed.
  We use the Huffman code entered by the students Build Huffman tree. If the node is stored in a non leaf node or other node information has been stored, it is an illegal code.

Algorithm 1: algorithm caption

Input: input N parameters frequencies

Output: output theShortestLength

some description;
p=new Minheap();
foreach *frequencies* **do**
    push $frequencies$ into p;
end
for *N-1 times* **do**
    a=pop p;
    b=pop p;
    push $a + b$ into p;
end
theShortestLength=pop p;

```
sort(codes,codes+N-1,cmp);
bool flag = 1;
for(int x=0;x<N-1;x++)
{
    for(int y=x+1;y<N;y++)
    {
        if(codes[x].find(codes[y]) == 0)
        {
            flag=0;
        }
    }
}
if(flag)
cout << "Yes" << endl;
else
cout << "No" << endl;
```

图 1: Prefix code violent search

Algorithm 2: algorithm caption

Input: input N parameters codes

Output: output YES or NO

```
1  some description;
2  Tree root=new Tree();
3  foreach codes do
4      foreach char i in code do
5          Node node=new Node(i);
6          if i =0 then
7              turn to left child;
8          end
9          else
10             turn to right child;
11         end
12     end
13     if already exist a node or not a leaf then
14         False Code;
15         break;
16     end
17     else
18         Set node;
19     end
20 end
```

3 Chapter 3: Testing results

In order to explore the time complexity, we need to encode in millions or tens of millions of orders, but there may be character coincidence. However, according to the previous algorithm introduction, Huffman coding is actually irrelevant to the problem of "what is a character". What we encode is actually the frequency of a character

Through the statistics and measurement of tens of millions of data randomly generated by the test program, both algorithms can get correct results, and the improved second scheme has better time complexity

The random data generation and test procedures are shown in the figure. The data test set generates 99 % of the students' currentwpl, which is consistent with the correctwpl, and 1 % random (to reflect the optimization effect)

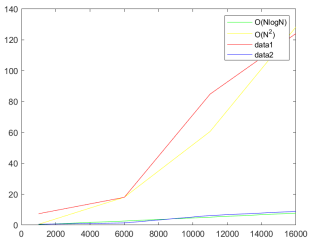

图 2: Comparison of running time, abscissa n

图 3: PTA operation results are shown in the figure

表 1: Test Cases

input	goal	output	state
4 A 2 B 2 C 2 D 4 2 A 00 B 01 C 10 D 11 A 000 B 001 C 01 D 1	Four data, coding and coding length will change	Yes Yes	pass
6 A 2 B 2 C 2 D 2 E 2 F 2 2 A 000 B 001 C 010 D 011 E 10 F 11 A 111 B 110 C 101 D 100 E 01 F 00	All frequencies are the same	Yes Yes	pass

表 2: Test Cases

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	Same as example, general situation	Yes Yes No No	pass
2 A 1 B 2 1 A 1 B 0	Minimum data	Yes Yes	pass

4 Chapter 4: Complexity Analysis

4.1 Space complexity:

4.1.1 Algorithm 1

It is necessary to establish a minimum stack of n elements, and the string array also has n elements. The overall space complexity is O (n).

4.1.2 Algorithm 2

1. It is necessary to establish a minimum stack of n elements with a complexity of O (n)

2. Some arrays for storing data, with a complexity of O (n)

3. On constructing prefix code discriminant tree

(1) If the weighted length of the input data does not meet the minimum length, the tree will not be built, and the total space is O (n)

(2) If the weighted sum of the input data is equal to the optimal solution, consider the worst case of the test data: the character with the lowest frequency occupies almost all the weighted sum. At this time, the tree needs to generate the most nodes, so we should consider the weighted sum corresponding to n characters. Considering the complexity of weighted sum, we consider two relatively extreme cases

1、The frequency of each character is equal. At this time, the tree constructed when solving the optimal coding length is a balanced binary tree. At this time, the tree height is $o(logn)$, that is, the coding length of each character is. Because there are n characters, it needs $O(nlogn)$space
2、The frequency of each character varies greatly. At this time, the tree constructed is a tree inclined to one extreme, and the corresponding required space size is $O(1+2+3+...+n) = O(n^2)$

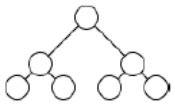

图 4: Balanced tree

So we estimate the weighted sum as $O(nlogn) - O(n^2)$

Considering that the code of each group of students will return to memory after running, the final conclusion is between $O(nlogn)$and $O(n^2)$

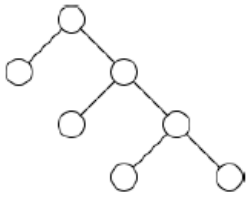

图 5: Extreme unbalanced tree

4.2 Time complexity:

4.2.1 Algorithm 1

1. Huffman coding with minimum heap: $o(nlogn)$
2. Calculate frequency weighted length $o(n)$
3. Violent search prefix code $o(n^2)$
4. There are m groups of data
5. Total time complexity $o(Mn^2)$

4.2.2 Algorithm 2

1. Huffman coding with minimum heap: $O(nlogn)$

2. Calculate frequency weighted length $O(n)$

3. For establishing prefix code discrimination tree, the time complexity should be $o(NH)$, and H is the tree height

We still consider the above two extreme cases

1、The frequency of each character is equal. At this time, the tree constructed when solving the optimal coding length is a balanced binary tree. At this time, the tree height is $O(logn)$, so the time complexity is $O(nlogn)$

2、The frequency of each character varies greatly. At this time, the tree constructed is a tree inclined to one extreme. At this time, the tree height is $O(n)$, so the time complexity is $O(n^2)$

5 Code appendix

```
#include <iostream>
#include <queue>
#include <map>
#include <algorithm>
using namespace std;
bool cmp(string a, string b)//用来排序的比较函数
{
    return a.length() > b.length();
}
string codes[1000];//存放编码
void print(int N)//打印结果
{
```

```
    sort(codes, codes + N - 1, cmp);//对编码按长度排序
    bool flag = 1;
    for (int x = 0; x < N - 1; x++)//N^2遍历编码
    {
        for (int y = x + 1; y < N; y++)
        {
            if (codes[x].find(codes[y]) == 0)//如果是
               前缀码
            {
                flag = 0;//则输出NO
            }
        }
    }
    if (flag)
        cout << "Yes" << endl;//不是前缀码
    else
        cout << "No" << endl;//是
}
priority_queue<int, vector<int>, greater<int>> p;//最
   小堆
int FreSum(int N)
{
    int s = 0;
    for (int i = 0; i < N - 1; i++)//霍夫曼编码计算长
       度
    {
        int a = p.top();//取堆顶元素
        p.pop();
        int b = p.top();//取堆顶元素
        p.pop();
        s += a + b;//加和后放回
        p.push(a + b);//加和后放回
    }
```

```
    return s;//返回最小加权长度
}
int main()
{

    int N, M;
    cin >> N;//读入数据
    char ch;
    int f;
    int table[1000];//记录频率
    for (int i = 0; i < N; i++)
    {

        cin >> ch;
        cin >> f;
        table[i] = f;//频率入数组
        p.push(f);//频率入堆
    }
    cin >> M;
    int sum = FreSum(N);//计算加权和
    // cout<<sum;
    int k = 0;
    for (int i = 0; i < M; i++)//判断学生编码正确性
    {
        char ch;
        string code;
        int sums = 0;
        for (int j = 0; j < N; j++)//遍历每一种方案
        {
            cin >> ch;
            cin >> code;
            codes[j] = code;
```

```
            sums += table[j] * code.length();//计算加
                权和
        }
        if (sums != sum)//加权和不是最小值
            cout << "No"<< endl;//一定不是最佳编码
        else
            print(N);//进行下一步判断
    }
}
```

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <map>
using namespace std;

map<char, int> c_f;//用于存放输入字符与频率
map<char, string> stu_c_f;//用于存放学生的字符与代码
map<char, string> Huffman;//用于存放每个字符的霍夫曼编
    码
int N;//字符的个数
typedef struct CTree TreeNode;
typedef struct CTree* Tree;

struct CTree//前缀码树
{
    char Element;
    Tree Left;
    Tree Right;
};

typedef struct HuffTree HuffTreeNode;
typedef struct HuffTree *HTree;
```

```
struct HuffTree//霍夫曼树
{
    int freq; // 出现频率, 即权重
    char key;
    HTree left;
    HTree right;
}  ;


struct n_Heap//堆
{
    HTree heap[300];
    int size;
};
struct n_Heap heap;
//以下函数是堆操作函数, 不详细介绍
void swap(int p1, int p2)
{
    HTree t = heap.heap[p1];
    heap.heap[p1] = heap.heap[p2];
    heap.heap[p2] = t;
}
int Minf(int fa, int ls, int rs)
{
    int min = fa;
    int minf = heap.heap[fa]->freq;
    if (ls <= heap.size && minf > heap.heap[ls]->freq)
    {
        minf = heap.heap[ls]->freq;
        min = ls;
    }
    if (rs <= heap.size && minf > heap.heap[rs]->freq)
    {
```

```
            minf = heap.heap[rs]->freq;
            min = rs;
        }
        return min;
}
void p_down(int v)
{
        int min;
        while (1)
        {
            min = Minf(v, 2 * v, 2 * v + 1);
            if (min != v)
            {
                swap(v, min);
                v = min;
            }
            else
                break;
        }
}
void p_up(int v)
{
        int min;
        while (1)
        {
            int fa = v / 2;
            if (fa == 0)
                break;
            min = Minf(fa, 2 * fa, 2 * fa + 1);
            if (min != fa)
            {
                swap(fa, min);
                v = fa;
```

```
        }
        else
            break;
    }
}
//计算霍夫曼编码
void count_code(HTree proot,string code)
{
    if(proot == nullptr)
        return;
    if(proot->left)
        code +='0';
    count_code(proot->left,code);
    if(!proot->left && !proot->right)
    {
        Huffman[proot->key]=code;//储存霍夫曼编码
    }
    code.pop_back();
    if(proot->right)
    {
        code +='1';
    }
    count_code(proot->right,code);
}
int min_num = 0;
void is_true()//判断同学编码是否正确
{
    int num=0;
    map<char, int>::iterator it;
    for (it = c_f.begin(); it != c_f.end(); it++)//计
        算同学编码的加权长
    {
        num += stu_c_f[it->first].size() * c_f[it->
```

```
        first];
}
if(num!=min_num)//如果该编码的加权长不符合最佳加权
    长, 就return
{
    cout<<"No"<<endl;
    return;
}
Tree proot = (Tree)malloc(sizeof(TreeNode));
proot->Left=nullptr;
proot->Right=nullptr;
proot->Element='\0';
for (it = c_f.begin(); it != c_f.end(); it++)
{
    Tree p=proot;

    for(int i=0;i<stu_c_f[it->first].size();i++)
    {
        if(stu_c_f[it->first][i]=='0'){
            if(p->Left==nullptr)
            {
                if(p->Element!='\0')//如果该字符要
                    放在其他字符的后续节点, 则
                    return
                {
                    cout << "No" << endl;
                    return;
                }
                p->Left=(Tree)malloc(sizeof(
                    TreeNode));
                p=p->Left;
                p->Element='\0';
                p->Left=nullptr;
```

```
                p->Right=nullptr;
            }
            else p=p->Left;
        }else{
            if (p->Right == nullptr)
            {
                if (p->Element != '\0') //如果该字
                    符要放在其他字符的后续节点, 则
                    return
                {
                    cout << "No" << endl;
                    return;
                }
                p->Right = (Tree)malloc(sizeof(
                    TreeNode));
                p = p->Right;
                p->Element = '\0';
                p->Left = nullptr;
                p->Right = nullptr;
            }
            else
                p = p->Right;
        }
    }
    if((p->Left!=nullptr || p->Right !=nullptr)||p
        ->Element!='\0')//如果该字符放在非叶节点处
        或其他字符的位置, 则return
    {
        cout<<"No"<<endl;
        return;
    }
    p->Element = it->first;
}
```

```
    cout<<"Yes"<<endl;//前面的条件都符合, 则该同学编码
        正确
}
int main()
{
    cin>>N;
    for(int i=0;i<N;i++)//读入字符与频率
    {
        char c;
        cin>>c;
        int f;
        cin>>f;
        c_f[c]=f;
        HTree pt = (HTree)malloc(sizeof(HuffTreeNode))
            ;
        pt->freq=f;
        pt->key=c;
        pt->left=nullptr;
        pt->right=nullptr;
        heap.heap[i+1]=pt;
    }
    heap.size=N;
    for (int i = heap.size / 2; i > 0; i--)//构建堆
    {
        p_down(i);
    }
    while (heap.size>1)
    {
        HTree proot = (HTree)malloc(sizeof(
            HuffTreeNode));
        HTree pl,pr;
        //每次取两个频率最小的树
        pl=heap.heap[1];
```

```
        swap(1,heap.size--);
        p_down(1);
        pr = heap.heap[1];
        swap(1, heap.size--);

        //合并树, 并将频率相加, 压入heap中
        p_down(1);
        proot->freq=pl->freq+pr->freq;
        proot->left=pl;
        proot->right=pr;
        heap.heap[++heap.size]=proot;
        p_up(heap.size);
    }

    string code;
    count_code(heap.heap[1],code);//霍夫曼编码

    map<char,int>::iterator it;
    min_num=0;
    for(it=c_f.begin();it!=c_f.end();it++)//计算最优加
        权长
    {
        min_num+=Huffman[it->first].size()*c_f[it->
            first];
    }
    // cout<<min_num<<endl;
    int num_s;
    cin>>num_s;
    Tree T=(Tree)malloc(sizeof(TreeNode));
    for(int i=0;i<num_s;i++)//读入同学提交的编码
    {
        for(int j=0;j<N;j++)
        {
```

```
            char c;
            cin>>c;
            string code;
            cin>>code;
            stu_c_f[c]=code;
        }
        is_true();//进行判断
    }
    // cout<<"1"<<endl;
}
```