浙江大学

Huffman Codes

Advanced Data Structures and Algorithm Analysis

Research Project 5

Wang Zhongwei	Lin Jiafeng	Jin Yuruo
王中伟	林家丰	金雨若

May 13, 2019

Contents

Chapter 1

Introduction

1.1 Problem Description

In this project, We are supposed to write a program to judge if the answers the students hand in is right when solving problem on Huffman codes. Because the Huffman codes sometimes can be not unique, we have to think out another way to judge the correctness of the answers rather than just compare its output with the correct answer produced by our program.
The input is a sequence of character and frequency pairs, which is the problem needed to solve by building Huffman codes. And then follow M answers that the students' code produce, for each character, it gives the character and its code. The output we are supposed to give is whether the answer is right, if right, output "Yes" and output "No" otherwise.

1.2 Background of Project

1.2.1 Min Heap

Min heap is a specialized tree-based data structure that the parent node is less than its child. Although there is no particular relationship among the siblings, it is a useful data structure when we need to find and remove the highest priority node. And in this project we use min heap to get the lowest frequency characters.

1.2.2 Greedy Algorithm

Greedy algorithm is a useful algorithm which works well in optimizing problems, it choose the choice that can get most benefits for the optimizing target in each stage. Sometimes it can not get the global optimal solution, because there is a great possibility the local optimal solution.
But in this project we solve the problem on Huffman codes, and it is guaranteed to be a global optimal solution.

Chapter 2

Algorithm Specification

2.1 Definition

2.1.1 function

```
void percup(Ptr *list, int i);
void ptrcdown(Ptr *list, int i);
Ptr deletemin(Ptr *list);
void insert(Ptr *list, Ptr node);
int compute(Ptr tree);
Ptr buildprotree(int N, cf* pro);
Ptr buildanstree(int N, cc *ans);
void deletetree(Ptr tree);
```

2.1.2 Node

```
typedef struct node *Ptr;
struct node
{
    bool leaf;
    int weight;
    int depth;
    Ptr left;
    Ptr right;
};
```

There are five variables, the leaf that mark if the node is a leaf node, the weight record the weight of the sub-tree with this node to be the root, and depth is the depth of the node, and the two pointers left and right is used to record the two child node of the node.

2.1.3 CF

```
struct cf
{
```

```
    char character;
    int freq;
};
```

CF is used to record the character and frequency pairs.

2.1.4 CC

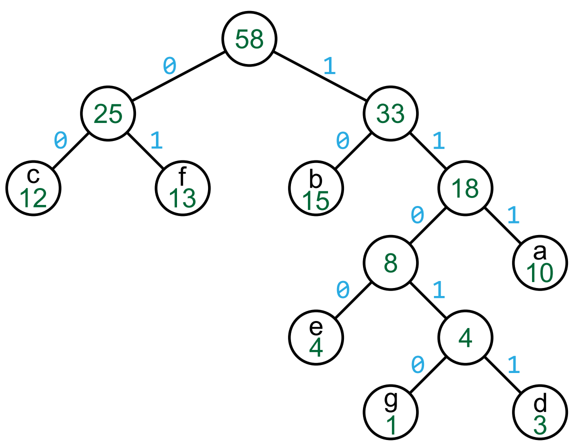
```
struct cc
{
    char anscharacter;
    int freq;
    string code;
};
```

To record the character, frequence and code submitted by the students

2.2 Huffman coding

2.2.1 Huffman tree

Huffman tree is a binary prefix tree, and all strings in Huffman tree are leaf nodes. The Huffman code of a character is corresponding string on the Huffman tree of the data.



character	a	b	c	d	e	f	g
frequency	10	15	12	3	4	13	1

Figure 2.1: Huffman tree

Frequencies of every character in the data are different. The length of compressed data is $\sum_{i=1}^{n} L_i F_i$(L_i is the length of corresponding string of i-th character,

F_i is the frequency of occurrence of i-th character in the data.) Our target is to minimize $\sum_{i=1}^{n} L_i F_i$ by arranging the structure of the tree.

2.2.2 Greedy algorithm

As we can see, the length of a character with low frequency is better to be long. By this idea, we use a greedy strategy to build the Huffman tree. Assume all characters are a node of tree. Every time we select two nodes of lowest frequency untill the size of queue is 1, then merge them (the frequency of new node is sum of two) and push into queue. (the queue here is a priority queue implemented with binary heap)

Algorithm 1 Build Huffman tree

function HUFFMAN-TREE(C)
    $Q := C$
    for $i = 1$ to $|C| - 1$ **do**
        allocate a new node z
        $x :=$ EXTRACT-MIN(Q)
        $y :=$ EXTRACT-MIN(Q)
        z.left:= x
        z.right:= y
        z.freq := x.freq $+y$.freq
        INSERT(Q, z)
    end for
    return EXTRACT-MIN(Q)
end function

2.2.3 Correctness of the algorithm

The greedy-choice property

Lemma: Let C be an alphabet in which each character $c \in C$ has frequency c.freq. Let x and y be two characters in C having the lowest frequencies. Then there exists 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.

The optimal substructure property

Lemma: Let C be a given alphabet with frequency c.freq defined for each character $c \in C$. Let x and y be two 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.

These two properties are proved in the book ”Introduction to Algorithms”[1]. The first property implies that choosing first and second minimum nodes is optimum. The second property implies that the weight of a node form from two nodes is sum of two nodes. Thus, the properties provide the correctness of the algorithm.

[1]Introduction to Algorithms, 3rd Edition: Ch.16.3; Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. The MIT Press. 2009

Chapter 3

Testing Results

3.1 Correctness Test

3.1.1 Test data 1

Purpose

Different characters with same frequencies.

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
```

Output

```
Yes
Yes
```

3.1.2 Test data 2

Purpose

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
```

Output

```
Yes
Yes
No
No
```

3.1.3 Test data 3

Purpose

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
```

Output

```
Yes
No
```

3.1.4 Test data 4

Purpose

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
```

Output

```
No
```

3.1.5 Test data 5

Purpose

The input is exactly the best peak shape.

Input

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

Output

```
Yes
```

Chapter 4

Analysis and Comments

4.1 Min-Heap Analysis

The time complexity of heap sorting is mainly in the process of initializing the heap process and rebuilding the heap after each maximum number is selected; Initialize the heap : time is O(n) Process:
Assume that the height is k, then from the node on the right of the penultimate layer, the nodes of this layer must perform the comparison of the child nodes and then exchange (if the order is correct, no exchange is needed); if the countdown layer is the third layer, it will select Sub-nodes are compared and exchanged, and if they are not exchanged, they can no longer be implemented. If it is exchanged, then a sub-tree should be selected for comparison and exchange. Then the total time is calculated as:$s = 2^{(i-1)*(k-i)}$; where i represents the layer, and 2^{i-1} represents how many elements are on the layer, (k - i) Indicates the number of times to be compared in the subtree. If it is in the worst condition, it will be exchanged after the number of comparisons; because this is a constant, it can be ignored after it is proposed;

$$S = 2^{k-2} * 1 + 2^{k-3} * 2..... + 2 * (k-2) + 2^0 * (k-1)$$

Because the leaf layer is not exchanged, i starts from k-1 to 1;
Solve this equation: The equation is multiplied by 2 and then subtracted from the original equation, it becomes:

$$S = 2^{k-1} + 2^{k-2} + 2^{k-3}..... + 2 - (k-1)$$

Except for the last item, it is a series of equal numbers, and the sum formula is used directly:$S = \frac{a1[1-(q^n)]}{1-q}$; $S = 2^k - k - 1$; and because k is the depth of the complete binary tree, $(2^k) <= n < (2^k - 1)$, in short, it can be assumed that: $k = logn$(The actual calculation should be $log(n+1) < k \leq log(n)$); In summary, we get: S = n - longn -1.
so the time complexity is: O(n) Rebuild heap time after changing heap elements: O(nlogn) Process: Loop n -1 times, each time from the root node to find the next cycle, so each time is logn, the total time: logn (n-1) = nlogn - logn; To sum up: the time complexity of heap sorting is: O(nlogn) Spatial complexity: The space complexity is O(N) because it need to store the N nodes.

4.2 Build Tree

The worst situation to build a Huffman Tree is O(n),the best situation to build tree is O(logn)

4.3 Huffman Tree Analysis

There are many ways to implement the Huffman tree. We Min-Heap to simply achieve this process. The algorithm is as follows:
1. Add n terminal nodes to the priority queue, then n nodes have a priority Pi, 1 $\leq i \leq n$
2. If the number of nodes in the queue is greater than 1, then:
(1) Remove the two smallest Pi nodes from the queue, ie make two consecutive removals (min(Pi), Min-Heap)
(2) Generate a new node, this node is the parent node of the removed node of (1), and the weight value of this node is (1) the weight of the two nodes
(3) Adding the node generated by (2) to the priority queue
3. The last point in the Min-Heap is the root node of the tree (root) The time complexity of this algorithm is $O(nlogn)$; because there are n terminal nodes, the tree has $2n-1$ nodes in total, and the priority queue must use $O(logn)$ for each cycle. The space complexity is $O(N)$ because it need to store the N nodes.

Appendices

Appendix A

Source Code (in C/C++)

```
#include <iostream>
#include <string>
#include <time.h>

#define KTIMES 1000
//Number of cycles, time of recording small N
#define CLK_TCK CLOCKS_PER_SEC

using namespace std;

typedef struct node *Ptr;
struct node//define huffman tree
{
    bool leaf;// whether leaf or not;
    int weight;// used to sort;
    int depth;//depth;
    Ptr left;
    Ptr right;
};

struct cf//use to establish problem huffman tree
{
    char character;
    int freq;
};

struct cc//use to eatablish answer huffman tree
{
    char anscharacter;
    int freq;
    string code;
};
//heap operation:
void percup(Ptr *list, int i);//used in insert heap
void ptrcdown(Ptr *list, int i);//used in deletemin
```

```
Ptr deletemin(Ptr *list);
void insert(Ptr *list, Ptr node);

int compute(Ptr tree);//compute PWL
Ptr buildprotree(int N, cf* pro);
//build problem huffman tree
Ptr buildanstree(int N, cc *ans);
//build answer huffman tree
void deletetree(Ptr tree);
// delete huffman tree

clock_t start, stop;//used to record operation time
double duration[KTIMES];

int main()
{
    srand((unsigned)time(NULL));//reset clock time
    long long ticks = 0;
    double totaltime = 0;

    int N;
    int M, count = 1;
    int *result;
    cin >> N;//get N
    cf *pro;
    Ptr protree;
    Ptr anstree;
    int proPWL;
    cc *ans;
    int ansPWL;
    //used to One-time processing of multi-clock data
    while(N != 0)
    {
        N = N <= 63 ? N : 63;
        pro = new cf[N];
        for (int i = 0; i < N; i++)//Initialization
        {
            cin >> pro[i].character;
            cin >> pro[i].freq;
        }
        //Cyclic time measurement
        for (int e = 0; e < KTIMES; e++)
        {
            start = clock();
            protree = buildprotree(N, pro);
            //build problem tree
            proPWL = compute(protree);
            //compute problem PWL
```

```
            stop = clock();
            //compute build tree and compute PWL time
            ticks = ticks + stop - start;
            duration[e] = ((double)(stop-start))/CLK_TCK;
            totaltime = totaltime + duration[e];
        }
        cout << "Build huffman codes and compute PWL: "<<\
        totaltime / KTIMES << endl;
        cout << "Ticks: " << ticks << endl;
        ans = new cc[N];
        cin >> M;
        result = new int[M];
        for (int i = 0; i < M; i++)
        {
            for (int k = 0; k < N; k++)//Initialization
            {
                cin >> ans[k].anscharacter;
                for (int j = 0; j < N; j++)
                {
                    // get the frequence from
                    // problem Initialization
                    if (pro[j].character == \
                        ans[k].anscharacter)
                    {
                        ans[k].freq = pro[j].freq;
                        break;
                    }
                }
                cin >> ans[k].code;
            }
            anstree = buildanstree(N, ans);//build tree
            ansPWL = compute(anstree);//conpute PWL
            if (ansPWL == proPWL) result[i] = 1;
            // if problem PWL==answer PWL,
            // the answer is right
            else result[i] = 0;
        }
        cout << "Case " << count << ":" << endl;
        for (int i = 0; i < M; i++)
        {
            if (result[i] == 1) cout << "Yes" << endl;
            else cout << "No" << endl;
        }
        count++;
        cin >> N;
        }
    system("PAUSE");
}
```

```
void percup(Ptr *list, int i)//heap up
{
    Ptr nownode = list[i];
    int p;
    //from up to down, if bigger
    for (p=i; list[p/2]->weight>nownode->weight; p=p/2)
    {
        list[p] = list[p / 2];//up the smaller
    }
    list[p] = nownode;//set the node
}

void ptrcdown(Ptr *list, int i)//heap down
{
    Ptr nownode = list[i];
    int p = i;
    int child = p * 2;
    //from down to up, if not out of range
    for (p = i; p * 2 <= list[0]->depth; p = child)
    {
        child = p * 2;
        if (child + 1 <= list[0]->\
        depth&&list[child + 1]->weight < \
        list[child]->weight)//get the smaller child
        {
            child++;
        }
        //if bigger bobble down
        if (list[child]->weight < nownode->weight)
        {
            list[p] = list[child];
        }
        else break;
    }
    list[p] = nownode;
}

Ptr deletemin(Ptr *list)//delete min
{
    Ptr min = list[1];
    int N = list[0]->depth;
    list[0]->depth--;
    list[1] = list[N];
    //get the last to the first , bobble down
    list[N] = NULL;
    ptrcdown(list, 1);
    return min;
```

```
}

void insert(Ptr *list, Ptr node)//insrtion
{
    list[0]->depth++;
    list[list[0]->depth] = node;
    percup(list, list[0]->depth);//bobble up
}

int compute(Ptr tree)//conpute PWL
{
    if (tree == NULL || (tree->left == NULL &&\
     tree->right == NULL))
    {
        return 0;
    }
    else
    {
        Ptr queue[63], p;
        //use tail queue front to get the depth*weight
        int front = 0;
        int tail = 0;
        int PWL = 0;
        p = tree;
        p->depth = 0;
        queue[tail++] = p;
        while (tail != front)
        {
            p = queue[front++];
            if (front == 63) front = 0;
            if (p->leaf)//if leaf, conpute
            {
                PWL = PWL + p->weight*p->depth;
            }
            else
            {
                // It ends before the end, indicating
                // that the establishment is incorrect.
                if (p->right == NULL || p->left == NULL)
                {
                    return -1;
                }
                else
                {
                    p->right->depth = p->depth + 1;
                    //set the right/left child depth
                    p->left->depth = p->right->depth;
                    queue[tail++] = p->left;
```

```
                    //set into the queue
                    if (tail == 63) tail = 0;//loop to use
                    else tail = tail;
                    queue[tail++] = p->right;
                    //set into the queue
                    if (tail == 63) tail = 0;
                    else tail = tail;
                }
            }
        }
    return PWL;
    }
}

Ptr buildprotree(int N, cf* pro)//build heap
{
    Ptr *heap, ptr, ptr1, ptr2;
    heap = new Ptr[N + 1];
    for (int i = 0; i < N + 1; i++)//Initialization
    {
        ptr = new node;
        if (i == 0)
        ptr->depth = N;//heap[0] used to insert
        else ptr->depth = 0;//Initialization depth
        ptr->leaf = true;
        ptr->left = NULL;
        ptr->right = NULL;
        if (i == 0) ptr->weight = -1e8;
        //Setting an Impossible Value
        else ptr->weight = pro[i - 1].freq;
        heap[i] = ptr;
    }

    for (int i = N / 2; i > 0; i--) ptrcdown(heap, i);
    //Sort heaps from small to large

    while (heap[0]->depth > 1)//build heap trees by heaps
    {
        ptr1 = deletemin(heap);//get the first smallest
        ptr2 = deletemin(heap);//get the second smallest
        ptr = new node;
        ptr->weight = ptr1->weight + ptr2->weight;
        //get the new node weight
        ptr->leaf = false;
        ptr->left = ptr1;//link the old one to the new one
        ptr->right = ptr2;
        ptr->depth = 0;
        insert(heap, ptr);//insert heap into heaps
```

```
    }
    ptr = heap[1];
    delete[]heap;
    return ptr;
}

Ptr buildanstree(int N, cc *ans)
{
    Ptr tree = new node;
    Ptr ptr;
    string nowstring;
    tree->leaf = false;
    tree->left = NULL;
    tree->right = NULL;
    for (int i = 0; i < N; i++)
    {
        ptr = tree;
        nowstring = ans[i].code;
        for (int j = 0; j < nowstring.length(); j++)
        {
            if (nowstring.at(j) == '0')//0-> is left
            {
                if (ptr->left == NULL)
                {
                    ptr->left = new node;//initialization
                    ptr = ptr->left;
                    ptr->left = NULL;
                    ptr -> right = NULL;
                    //if loop into the last one character
                    if (j + 1 == nowstring.length())
                    {
                        ptr->leaf = true;// set is leaf
                        ptr->weight = ans[i].freq;
                        //set weight
                        break;
                    }
                    else
                    {
                        ptr->leaf = false;
                    }
                }
                // not null and not the string ends
                // but is leaf,
                // not the huffman code answer
                else if (ptr->left->leaf == true)
                {
                   deletetree(tree);
                    return NULL;
```

```
                }
                else//loop in
                {
                   ptr = ptr->left;
                }
           }
            else//1-> is right
            {
                if (ptr->right == NULL)
                {
                    ptr->right = new node;//initialization
                    ptr = ptr->right;
                    ptr->left = NULL;
                    ptr->right = NULL;
                    //if loop into the last one character
                    if (j + 1 == nowstring.length())
                    {
                        ptr->leaf = true;// set is leaf
                        ptr->weight = ans[i].freq;
                        //set weight
                        break;
                    }
                    else
                    {
                        ptr->leaf = false;
                    }
                }
                // not null and not the string ends
                // but is leaf,
                // not the huffman code answer
                else if (ptr->right->leaf == true)
                {
                    deletetree(tree);
                }
                else
                {
                    ptr = ptr->right;
                }
            }
        }
    }
    return tree;
}

void deletetree(Ptr tree)//delete tree
{
    if (tree->left != NULL) deletetree(tree->left);
    if (tree->right != NULL) deletetree(tree->right);
```

```
    delete(tree);
}
```

Appendix B

Declaration and Signatures

Declaration

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

Signatures

毛中伟

林家丰

金雨若