Huffman Codes

Author: Shen Yunfeng / Zhao Yilei / Wang Kedi

Date: 2022/4/21

Chapter 1: Introduction

Huffman Code is a particular type of prefix code which can be used to minimize the space we need for storing data. It has min WPL value, where WPL = ∑length(i) * frequency(i), so that it enables the word with higher frequency to have less length, and minimize total length of the article. What's more, in case of ambiguity, none of the codes should be the prefix of other character's code.

However, you might have noticed that the huffman codes might not be unique. Therefore, different students might generate different codes concerning the same problem, and we'll provide you with a program which can judge whether the code is a proper Huffman code.

Chapter 2: Data Structure / Algorithm Specification

Concerning this question, we can easily get two ways to solve it. One is to eatablish a standard Huffman Tree from the given frequency,

Data Structure

1. Huffman Tree

In the circumstances, we need to make the character with the most frequency to have the least length. Therefore, according to the Huffman Algorithm, we rank the characters by their order. Therefore, we'll use binary heap so that we can get them faster, which will be specified soon. Each time we get the two nodes, we make them two subtrees of a new node, where the frequency of the new tree is the sum of they two. After that, we continue to enqueue the tree and keep doing this, until there's only one tree in the queue. And that is the Huffman tree which we need. And in the Huffman tree, the depth of the node is the length of its node accordingly.

Here what should be paid special attention is that which one of the two subtrees should be the left one don't need to follow a certain rule, which means there might be different ways to coding the characters. However, they're all the best codes we want.

2. Priority queue (Binary heap)

In the previous part, we have specified that we need to get the two trees with the least frequency. So here priority queue can be useful to do that. What's more, to simplify the operations, we choose Binary heap to realize that.

Algorithm Specification:

Sketch

Our algorithm mainly contains three parts: build a standard Huffman Tree from the given information and compute the minimize WPL value, check the WPL of each given codes, and check whether one of the codes is the prefix of others. Here we'll specify them one by one.

Algorithm 1: Build standard Huffman Tree (and compute min WPL)

The process how we build standart Huffman Tree has been specified in the **"Data Structure"** part, so we won't cast too much space and time on it anymore.

Pseudo Code:

```
double BuildHuffTree(int n) //return the min WPL of the inputting characters
{
    for(int i = 0; i < n; i++)
        input the value and frequency of each character in t[i];
    Make t[i] a Binary heap;    //PercolateDown from t[n/2-1] to t[0]
    for(i = 0; i < n-1; i++)    //merge n-1 times to make n nodes into one tree
    {
        t1 = DeleteMin(t[]);    //the tree with the least frequency
        t2 = DeleteMin(t[]);    //the tree with the second least frequency
        t0->left = t1;
        t0->right = t2;     //t0 is the root of the new tree with t1 and t2
being its two subtrees
        t0->frequency = t1->frequency + t2->frequency;  //update the frequency
        t0->key = '$';      //mark that t0 is not a real node
        t[n-i-2] = t0;      //insert the new tree into the heap
        PercolateUp(t0);    //keep the heap's order
    }
    //now we have built the Huffman Tree
    double minWPL = 0;  //initialize the minWPL
    for(each level of the tree)
    {
        for(each node tn of this height)
            if(tn->key != '$')
                minWPL += tn->frequency * height;
            //here the node's length is exactly its height
    }
    return minWPL;
}
```

Algorithm 2: Checking the WPL of each case

The first condition of the proper Huffman code is that its WPL must be minimal. Therefore, according to the definition of WPL, we can easily compute them, and check.

Pseudo code

```
bool checkWPL(double minWPL, int n)
{
    double WPL = 0;
    char c; //the key
    char code[MAX_LENGTH];  //the code
    for(int i = 0; i < n; i++)
    {
        scanf("%c%s", &c, code);
        WPL += strlen(code) * freqlist[c];  //we have stored the frequency of
each char in freqlist[]
    }
    return (WPL == minWPL);
}
```

Algorithm 3: Check the prefixs

Here we choose to rebuild the Huffman tree from the codes. If one code is the prefix of the others, its end must be in one path from root to a leaf, but not NIL.

Pseudo Code

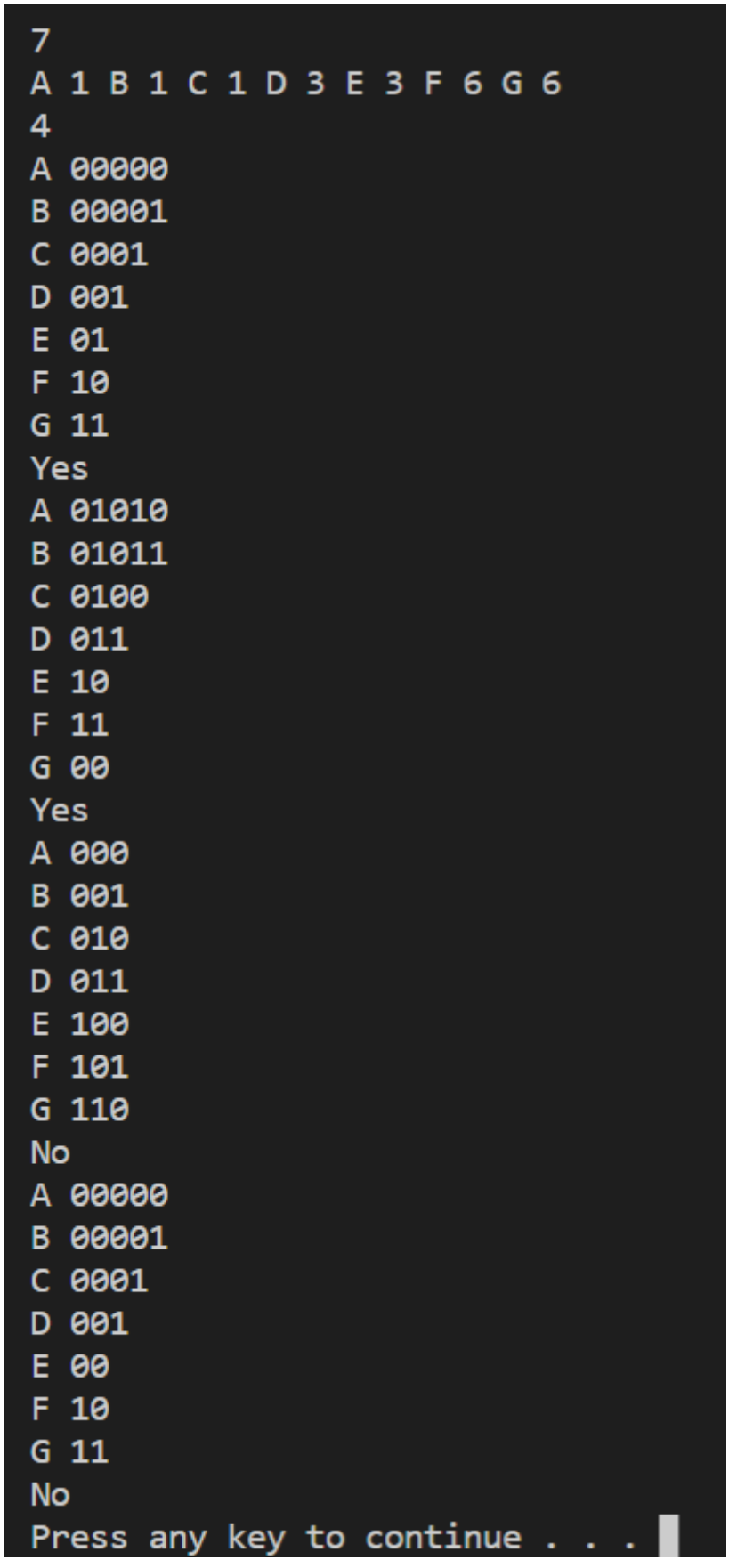
```
bool checkPrefix(int n)
{
    for(each code)  //where code is a char[]
    {
        node* ptr = root;
        for(int i = 0; i < strlen(code - 2); i++)   //travel each bit of the
node except the last one
        {
            if(code[i] == '0')
            {
                if(ptr->left == NULL)   //check whether the node has existed
                    add a new node to ptr->left;    //add a new node
                ptr = ptr->left;    //move to check the next one
            }
            else if(code[i] == '1') //the same except that now we'll check the
right
            {
                if(ptr->right == NULL)
                    add a new node to ptr->right;
                ptr = ptr->right;
            }
            if(ptr->isEnd)  return false;   //other code is prefix of this one
        }//now we have checked n-1 bits
        if(code[i] == '0')  //the last bit is 0
        {
            if(ptr->left != NULL)   //this one is a prefix of others
                return false;
            else add a new node to ptr->left;   //add a new string code[]
        }
        else if(code[i] == '1')
        {
            if(ptr->right != NULL)
                return false;
            else add a new node to ptr->right;
        }
        return true;
    }
}
```

Chapter 3: Testing Result

Sample 1: The given sample

Purpose: Test the problem easily, with the case where the WPL is not right (case 3), and some one is the prefix of others (case 4).

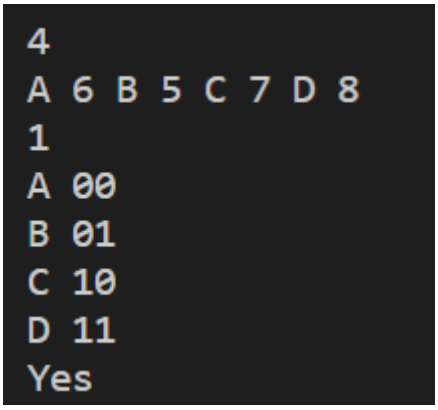
Status: Pass



Sample 2

Purpose: Check the the case when we don't always make the tree with lower frequency be left or right

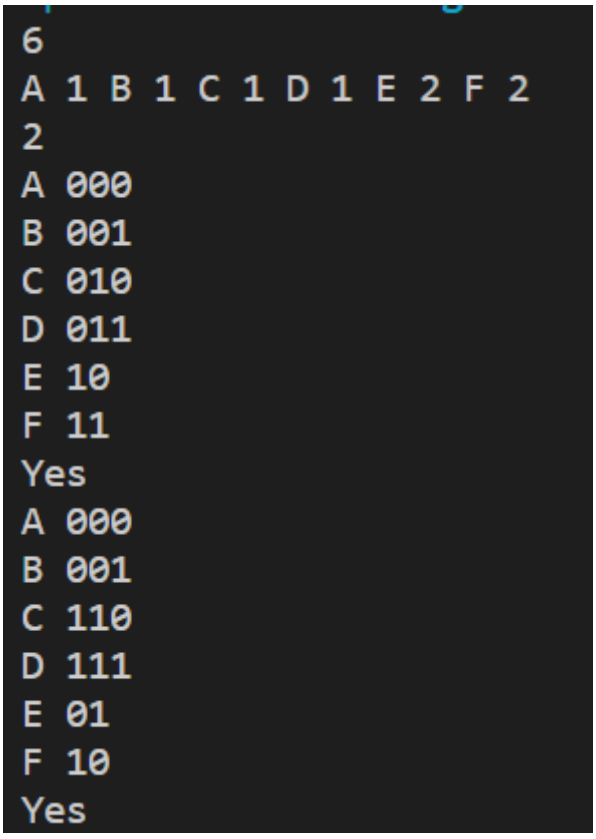
Status: Pass



Sample 3

Purpose: Check the case when the HUffman Tree is not unique

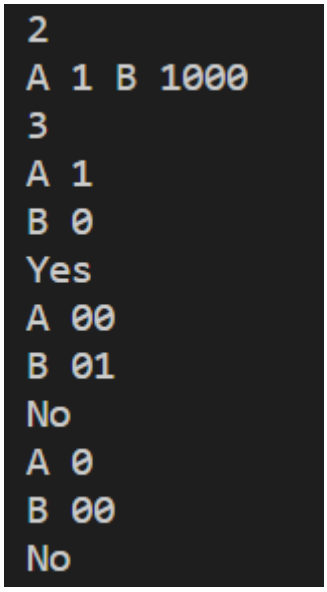
Status: Pass



Sample 4: Min size

Purpose: Check the program of min size

Status: Pass



Sample 5: Max size

Purpose: Check the program of max size

Status: Pass. (The test is too long to put here)

Chapter 4: Analysis and Comments

We suppose n is the number of characters to be coded.

Time Complexity

Algorithm 1: Build standard Huffman Tree (and compute min WPL)

In this scope, we'll create a Binary heap at first, whose time complexity is `O(n)`. And when it comes to build the Huffman Tree, each time of DeleteMin is `O(logn)`. After that, we need to insert the new tree into priority queue, where PercolateUp need `O(logn)` as well. And each time we build a tree, we reduce the number of nodes by 1, so we need to repeat it for `n-1` times, where total time complexity to build a standard Huffman Tree is `O(nlogn)`.

After that, we need to compute the minWPL from the tree. Here we need to traverse the tree in levelOrder, where the time complexity is `O(nodes)`. Meanwhile, each time we getr a subtree, we add a new empty node, so the number of nodes is `2n-1`, and therefore, time complexity is `O(n)` here.

So total time complexity is `O(n) + O(nlogn) + O(n)` = `O(nlogn)`.

Algorithm 2: Checking the WPL of each case

There's nothing to say about that. We check the length of each code and multiply it by its frequency. So time complexity is `O(n)` here.

Algorithm 3: Check the prefixs

Each time we rebuild a path in the Huffman Tree, where the length is `O(n)`. And we continue to execute the progress for n times, where time complexity is `O(n^2)`.

Totally, the time complexity is `O(nlogn) + O(n) + O(n^2)` = `O(n^2)`.

Space Complexity

Algorithm 1: Build standard Huffman Tree (and compute min WPL)

As specified before, each time we add an empty node to be the root, and there's `2n-1` nodes totally to build the tree. So total space complexity is `O(n)` here.

Algorithm 2: Checking the WPL of each case

It depends on the length of codes given. If you'd like it, you can input codes which are infinitely long, despite it is not a proper Huffman code. So the space complexity is `O(total bits)`. Meanwhile, for a proper Huffman code, one code will take at most `n` bits, so space complexity is `O(n^2)` for a **proper** Huffman code.

Algorithm 3: Check the prefixs

Because we have checked the WPL before, the code can't be arbitrarily long now, which is `O(n)` here. Because it should always be a proper Huffman Tree before we find error. So the space complexity is `O(n)`.

Totally, for a **proper** Huffman code, total space time complexity is `O(n) + O(n^2) + O(n)` = `O(n^2)`.

Comments

Out of time, there's still many aspects for us to improve. For example, we can see it clearly that the time complexity of checking prefix is O(n^2). Of course an easy way to check prefix is to travel each code and test whether it is a prefix of others, which is O(n^2) as well. We tried to improve it, but still failed, and the time complexity is still O(n^2). So maybe we could improve it in the future.

Appendix: Source Code

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

typedef struct TreeNode  //node of huffman tree
{
    int level;
    char key;
    int freq;           //frequncy of key
    TreeNode *lchild;
    TreeNode *rchild;
}tnode;
tnode* t[500];          //min_heap


typedef struct CheckNode //node of check tree
{
    CheckNode* lchild;
    CheckNode* rchild;
}cnode;
cnode* cT = NULL;       //root of check tree
```

```
tnode * buildtnode(int level,char key, int f,TreeNode *lchild,TreeNode *rchild);
//initialize huffman tree node
void insert(tnode* address);
//inset node into min_heap
void PercolateUp(int child);
//percolate up
void PercolateDown(int parent);
//percolate down
void swap(int a, int b);
//swap heap node
tnode* DeleteMin();
//pop min node & delete min
int LevelOrder(tnode* root);
//travel huffman tree in level-order to calculate WPL
cnode* buildcnode();
//initialize check tree node
int buildtree(string code);
//insert node into check tree, return: 1-OK; 0-it's prefix
void FreeTree(cnode* t);
//free check tree
bool cmp(string c1, string c2);
//int sort: sort code by length
int t_size = 0;
//size of heap

int main()
{
    int freqlist[160] = {0};//freqlist[i] = the frequency of character whose
ASCII = i
    int n,f;                //n_character, frequency
    char c;                 //character
    cin >> n;
    for(int i = 0; i < n; i++)
    {
        cin >> c >> f;
        freqlist[c] = f;
        tnode* cur_tnode =buildtnode(-1,c,f,NULL,NULL);
        t[t_size++] = cur_tnode; //use linear algorithm to build heap
    }

    for(int i = (n-1)/2 ; i >= 0 ; i--) PercolateDown(i); //adjust min heap

    for(int i = 1; i < n; i++) //build huffman tree by n-1 * merge
    {
        tnode* n1 = DeleteMin();
        tnode* n2 = DeleteMin();
        tnode* cur_tnode = buildtnode(-1,'$',n1->freq + n2->freq,n1,n2);
//insert merged node into min_heap
        insert(cur_tnode);
    }
    tnode* fn = DeleteMin();  //the last node in min_heap = root of huffman tree
    int WPL = LevelOrder(fn); //travel huffman tree in level-order, calculate
standard WPL

    int n_case;
    cin >> n_case;
    for(int i = 0; i < n_case; i++)  //check case by case
    {
```

```
        int curWPL = 0;
        string code[100];
        //current WPL = SUM(frequency(key)*code_length(key))
        for(int j = 0 ; j < n; j++)
        {
            cin >> c >> code[j]; getchar();
            curWPL += code[j].length()*freqlist[c];
        }
        if(curWPL != WPL) //current WPL != standard WPL, incorrect
        {
            cout << "No" << endl;
            continue;
        }
        sort(code, code+n, cmp); //sort by length in decending order

        cT = buildcnode(); //initialize the root of check tree
        for(int i = 0; i < n ; i++) //start checking from the longest code
        {
             if(buildtree(code[i]) == 0) //code[i] is a prefix, incorrect
            {
                cout << "No" << endl;
                break;
            }
            if(i==n-1) cout << "Yes" << endl;
        }
        FreeTree(cT);
    }
    return 0;
}


tnode * buildtnode(int level,char key, int f,TreeNode *lchild,TreeNode *rchild)
{   //have a new huffman tree node
    tnode* curnode = (tnode*)malloc(sizeof(tnode));
    curnode->level = level;
    curnode->key = key;
    curnode->freq = f;
    curnode->lchild = lchild;
    curnode->rchild = rchild;
    return curnode;
}

cnode* buildcnode()
{   //have a new check tree node
    cnode* cur = (cnode*)malloc(sizeof(cnode));
    cur->lchild = cur->rchild = NULL;
    return cur;
}


void insert(tnode* address)//insert at the bottom of min_heap, then percolate up
{   //insert at bottom, then percolate up
    t[t_size] = address;
    PercolateUp(t_size);
    t_size++;
}

void PercolateUp(int child)
```

```
{
    int c = child;
    int p = (c-1)/2;
    while(p>=0)
    {
        if(t[c]->freq < t[p]->freq)
        {
            //swap
            tnode* tmp = t[c];
            t[c] = t[p];
            t[p] = tmp;
            //go up
            c = p;
            p = (c-1)/2;
        }
        else break;
    }
}

void PercolateDown(int parent)
{
    int p = parent;
    int c = 2*p + 1;
    while(c < t_size)
    {
        if(c+1 < t_size && t[c+1]->freq < t[c]->freq) ++c;//find min_child
        if(t[c]->freq < t[p]->freq)
        {
            swap(c,p);
            //go down
            p = c;
            c = 2*p + 1;
        }
        else break;
    }
}

tnode* DeleteMin()//pop & delete min
{   //swap top & bottom, percolate down, return bottom
    swap(0,t_size-1);
    t_size--;
    PercolateDown(0);

    return t[t_size];
}

int LevelOrder(tnode* root)
{   //travel huffman tree in level-order, return WPL
    int WPL = 0;
    queue <tnode*> q;
    //root enqueue
    root->level = 0;
    q.push(root);

    while(!q.empty())
    {
        //dequeue
        tnode* pop = q.front();
```

```
        q.pop();

        if(pop->lchild && pop->rchild)//internal node -> lchild & rchild enqueue
        {
            pop->lchild->level = pop->rchild->level = pop->level + 1;
            q.push(pop->lchild);
            q.push(pop->rchild);
        }
        else
            WPL += pop->level*pop->freq;
    }
    return WPL;
}

void FreeTree(cnode* t)
{   //clear check tree
    if(t)
    {
        if(t->lchild)
        {
            FreeTree(t->lchild);
            t->lchild = NULL;
        }
        if(t->rchild)
        {
            FreeTree(t->rchild);
            t->rchild = NULL;
        }
    }
    t = NULL;
    return;
}

int buildtree(string code)
{   //insert node by code into check tree, return: 1-it's ok; 0-it's a prefix
    cnode* cur = cT;
    int len = code.size();
    for(int i = 0; i < len-1; i++)
    {
        if(code[i] == '0')//code[i]: '0': turn left; '1': turn right
        {
            if(cur->lchild == NULL)
                cur->lchild = buildcnode();
            cur = cur->lchild;
        }
        else
        {
            if(cur->rchild == NULL)
                cur->rchild = buildcnode();
            cur = cur->rchild;
        }
    }
    if(code[len-1] == '0')
    {   //final position have a node already, it's a prefix
        if(cur->lchild) return 0;
        else
        {
            cur->lchild = buildcnode();
```

```
            cur = cur->lchild;
        }
    }
    else
    {   //final position have a node already, it's a prefix
        if(cur->rchild) return 0;
        else
        {
            cur->rchild = buildcnode();
            cur = cur->rchild;
        }
    }
    if(cur->lchild || cur->rchild)
        return 0;
    return 1;
}

bool cmp(string c1, string c2)
{
    return c1.length() > c2.length();
}

void swap(int a, int b)
{   //swap tnode a & b
    tnode* tmp = t[a];
    t[a] = t[b];
    t[b] = tmp;
}
```

Author List

Name	Work
Shen Yunfeng	Code
Zhao Yilei	Test & Debug
Wang Kedi	Report

References:

None.

Declaration:

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