实验4 贪心算法

姓名：叶倩琳

班级：软工1706

完成时间：2019/04/18

**问题一**

1. **问题描述**

Huffman编码：

测试数据：

X={zhejiang university of technology,computer college}

Y={1310773597218806522025}

1. **代码实现**

#include <iostream>

#include<cstring>

#include<algorithm>

using namespace std;

/\*哈夫曼编码\*/

//设置一个节点结构体

struct Node{

double weight; //权值

string ch; //字符

string code; //编码

int lchild;

int rchild;

int parent; //左、右、父亲节点

};

//找权值最小的两个节点a、b，合并成节点n

void Select(Node huffTree[], int \*a, int \*b, int n){

int i;

double weight = 0;

//找叶子节点中最小的

for (i=0;i<n;i++){

if (huffTree[i].parent!=-1)

continue;

//节点还没有被选过

else{

if(weight==0){

weight=huffTree[i].weight;

\*a=i;

}

else{

if(huffTree[i].weight<weight){

weight=huffTree[i].weight;

\*a=i;

}

}

}

}

//找第二小的节点

weight=0;

for(i=0;i<n;i++){

if(huffTree[i].parent!=-1 || (i==\*a)) //排除已选过的数

continue;

else{

if(weight==0){

weight=huffTree[i].weight;

\*b=i;

}

else{

if(huffTree[i].weight<weight){

weight=huffTree[i].weight;

\*b=i;

}

}

}

}

//小的数放左边

int temp;

if(huffTree[\*a].lchild<huffTree[\*b].lchild){

temp=\*a;

\*a=\*b;

\*b=temp;

}

}

//构造Huffman树

void Huff\_Tree(Node huffTree[], int w[], char ch[], int n){

int i,k;

//初始化

for(i=0;i<2\*n-1;i++){

huffTree[i].parent=-1;

huffTree[i].lchild=-1;

huffTree[i].rchild=-1;

huffTree[i].code="";

}

//叶子节点初始化

for(i=0;i<n;i++){

huffTree[i].weight=w[i];

huffTree[i].ch=ch[i];

}

for(k=n;k<2\*n-1;k++){

int i1=0;

int i2=0;

//将i1，i2节点合成节点k

Select(huffTree, &i1, &i2, k);

huffTree[i1].parent=k;

huffTree[i2].parent=k;

huffTree[k].weight=huffTree[i1].weight+huffTree[i2].weight;

huffTree[k].lchild=i1;

huffTree[k].rchild=i2;

}

}

//编码

void Huff\_Code(Node huffTree[], int n){

int i,j;

string s = "";

//通过不断找其父亲节点，依此对n个叶子节点进行编码

for(i=0;i<n;i++){

s="";

j=i;

while(huffTree[j].parent!=-1){

int k=huffTree[j].parent;

//若为左孩子

if(j==huffTree[k].lchild)

s+="0";

else

s+="1";

j=huffTree[j].parent;

}

cout<<"字符 "<<huffTree[i].ch<<" 的编码：";

//因为是从叶子到根节点求编码，所以要将求到的编码倒转

reverse(s.begin(),s.end());

huffTree[i].code=s;

cout<<s<<endl;

}

}

//解码

string Huff\_Decode(Node huffTree[], int n,string s){

cout<< "解码后为：";

int i,j,len=s.size();

string temp="",str="";

//逐个增加要解码的字符，与每一个编码进行匹配

for(i=0;i<len;i++){

temp+=s[i];

for(j=0;j<n;j++){

//若匹配到了编码

if(temp==huffTree[j].code){

str+=huffTree[j].ch;

temp=""; //重置

break;

}

//若全部遍历后都没有

else if(i==(len-1) && j==(n-1) && temp!=""){

str="解码错误！";

}

}

}

return str;

}

int main(int argc, char\*\* argv) {

int n=0,i,j;

cout<<" 请输入要进行编码的字符串："<<endl;

char ch,c[256];

int w[256],num[256]={0};

//计算每个字符的频率

while((ch=getchar())!='\n')

num[ch]++;

for(i=0;i<256;i++){

if(num[i]){

c[n]=i;

w[n]=num[i];

n++;

}

}

for(i=0;i<n;i++)

cout<<c[i]<<":"<<w[i]<<endl;

//编码

Node huffTree[2\*n]; //叶子节点有n个时，完全二叉树的节点共有2\*n-1个

Huff\_Tree(huffTree,w,c,n);

Huff\_Code(huffTree,n);

//解码

string s;

cout<<" 请输入编码：";

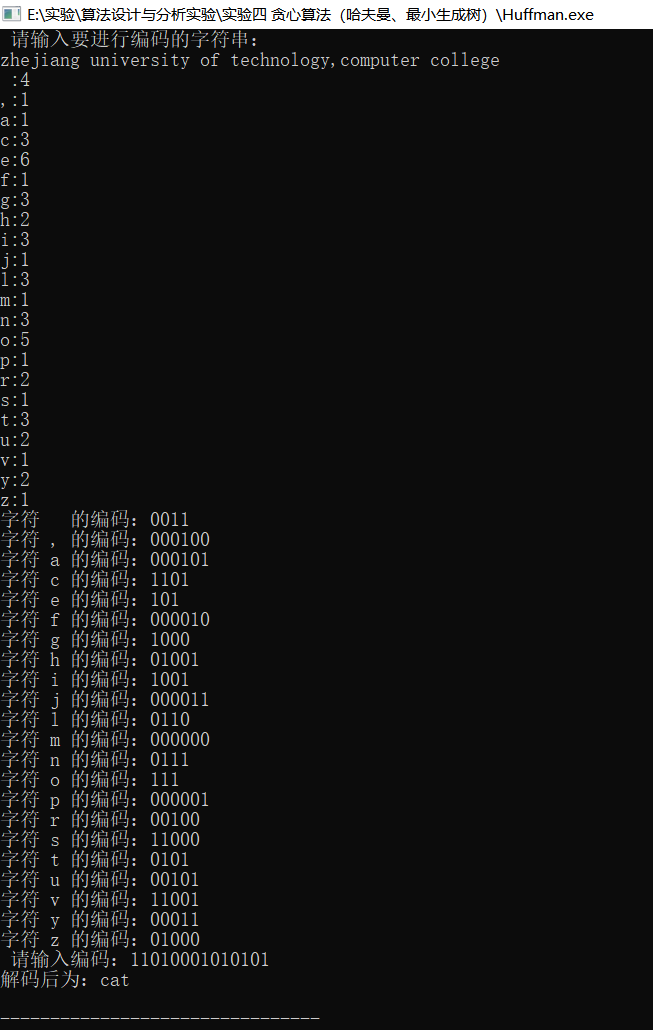
cin>>s;

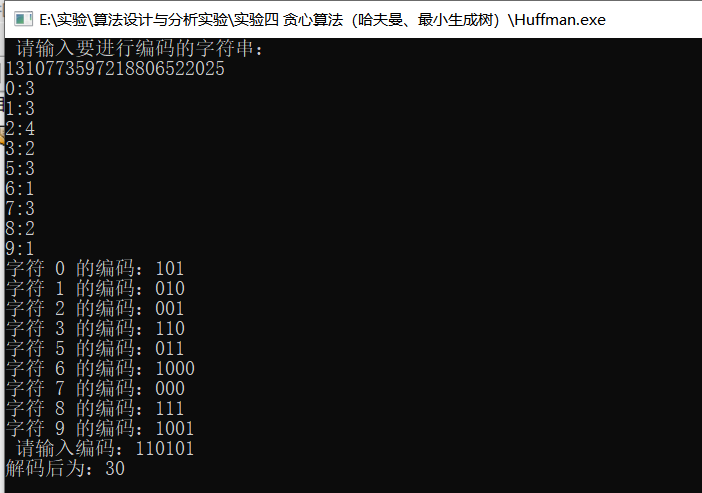
cout<<Huff\_Decode(huffTree,n,s)<<endl;

return 0;

}

1. **输出结果**





1. **性能分析**

Huffman算法的基本思想如下：

1. 统计符号的发生概率，复杂度为O(n)；
2. 把频率按从小到大的顺序排列
3. 每一次选出最小的两个值，作为二叉树的两个叶子节点，将和作为它们的根节点
4. 重复 3，直到取完所有的叶子节点；

使用优先队列算法的时间复杂度为O（nlogn），因为有n个叶子节点，所以树总共有2n-1个节点，使用优先队列每个循环须O（logn）。

1. **实验体会**

Huffman编码的实现也是贪心算法的一个案例，即不断选取概率最小的节点进行合并，我认为它实现的难点在于不仅要删除取出的两个概率最小的节点，还要把合并的节点也加入优先队列中。

**问题二**

**1、问题描述**

最小生成树：

Prim算法。

Kruskal算法。

测试数据：

课堂上的无向图；

1. 0 2 8 1 0 0 0 0
2. 2 0 6 0 1 0 0 0
3. 8 6 0 7 5 1 2 0
4. 1 0 7 0 0 0 9 0
5. 0 1 5 0 0 3 0 8
6. 0 0 1 0 3 0 4 6
7. 0 0 2 9 0 4 0 3
8. 0 0 0 0 8 6 3 0

其中矩阵中的数据为节点之间的距离，比较并分析他们的算法复杂度（注意优先队列与邻接矩阵等不同实现方式）。

**2、代码实现**

**（1）Prim 算法**

#include <iostream>

using namespace std;

/\* Prim最小生成树问题变量声明：

closest[j]=i---j在S中的邻接顶点i（j属于V-S,i属于S）

c[i][j]--------边（i，j）的权

lowcost[j]=c[j][closest[j]],其中c[j][closest[j]]<=c[j][k]

\*/

#define maxint 1000

#define inf 100000

//NOTE:顶点是从1开始的,数组中是从下标0开始的

void Prim(int n,int c[][8]){

int lowcost[maxint];

int closest[maxint];

bool s[maxint];

int min=inf;

int k,i,j;

//若邻阶矩阵中两个顶点为0，则两者距离为inf无穷远

for(i=0;i<n;i++){

for(j=0;j<n;j++){

if(c[i][j]==0)

c[i][j]=inf;

}

}

//初始化

s[0] = true; //刚开始时S={1}

for(i=1;i<n;i++){

lowcost[i]=c[0][i];

closest[i]=0;

s[i]=false;

}

//n个顶点的最小生成树有(n-1)条边

for(i=0;i<(n-1);i++){

j=0;

min=inf;

//找顶点

for(k=1;k<n;k++){

if((lowcost[k]<min) && (!s[k])){

min=lowcost[k];

j=k;

}

}

s[j]=true; //把j顶点加入到集合中

cout<<j+1<<" "<<closest[j]+1<<endl;

/\*更新 lowcost 和 closest,

只要考虑相对于新加入顶点j有变化的V-S中的顶点（即还没有被取到的顶点）\*/

for(k=1;k<n;k++)

if((c[j][k]<lowcost[k])&&(!s[k])){

lowcost[k]=c[j][k];

closest[k]=j;

}

}

}

int main(){

int c[][8]={{0,2,8,1,0,0,0,0},{2,0,6,0,1,0,0,0},

{8,6,0,7,5,1,2,0},{1,0,7,0,0,0,9,0},

{0,1,5,0,0,3,0,8},{0,0,1,0,3,0,4,6},

{0,0,2,9,0,4,0,3},{0,0,0,0,8,6,3,0}};

cout<<" 邻阶矩阵为："<<endl;

for(int i=0;i<8;i++){

for(int j=0;j<8;j++){

cout<<c[i][j]<<" ";

}

cout<<endl;

}

cout<<" 用Prim算法生成最小生成树的操作过程如下："<<endl;

Prim(8,c);

return 0;

}

**（2）Kruskal 算法**

#include <iostream>

#include<algorithm>

using namespace std;

#define maxint 1000

#define inf 100000

/\* Kruskal最小生成树问题:

要设置一个数组connected给不同的连通分量标号，以此判断两个顶点是否连通

\*/

//定义一个边的结构体

typedef struct{

int u; //边的起始顶点

int v; //边的n终点

int w; //边的权值

}Edge;

bool comp(Edge A,Edge B){

return A.w<B.w;

}

void Kruskal(int n,int c[][8]){

int i,j,uu,vv,connected\_num1,connected\_num2,k=0;

int connected[maxint]; //数组connected用于判断两顶点之间是否连通

Edge E[maxint]; //E为一个边的数组

//初始化

for(i=0;i<n;i++){

for(j=0;j<n;j++){

if(i<j && c[i][j]!=0){

E[k].u=i;

E[k].v=j;

E[k].w=c[i][j];

k++;

}

}

}

//对每条边按权值进行sort排序

sort(E,E+k,comp);

//给每个顶点置不同连通分量编号，即初始时有n个连通分量

for(i=0;i<n;i++)

connected[i]=i;

k=1; //k表示当前构造生成树的第n条边，初始值为1

j=0; //j为数组E中元素的下标，初值为0

//循环产生最小生成树的n-1条边

while(k<n){

uu=E[j].u;

vv=E[j].v; //边的头尾顶点

connected\_num1=connected[uu];

connected\_num2=connected[vv]; //顶点所属的连通分量编号

//若两个顶点不在同一个连通分量上，把该边加入集合中

if(connected\_num1!=connected\_num2){

//NOTE:顶点是从1开始的,数组中是从下标0开始的

cout<<" Edge: ("<<uu+1<<","<<vv+1<<"), Wight: "<<E[j].w<<endl;

k++; //生成的边数增1

/\*更新这两个顶点所属的连通分量的编号，

即所有连通分量的编号为sn2的边改其连通分量编号为sn1\*/

for(i=0;i<n;i++){

if(connected[i]==connected\_num2)

connected[i]=connected\_num1;

}

}

j++; //扫描下一条权值最小的边

}

}

int main(){

int c[][8]={{0,2,8,1,0,0,0,0},{2,0,6,0,1,0,0,0},

{8,6,0,7,5,1,2,0},{1,0,7,0,0,0,9,0},

{0,1,5,0,0,3,0,8},{0,0,1,0,3,0,4,6},

{0,0,2,9,0,4,0,3},{0,0,0,0,8,6,3,0}};

cout<<" 邻阶矩阵为："<<endl;

for(int i=0;i<8;i++){

for(int j=0;j<8;j++){

cout<<c[i][j]<<" ";

}

cout<<endl;

}

cout<<" 用Kruskal算法生成最小生成树的操作过程如下："<<endl;

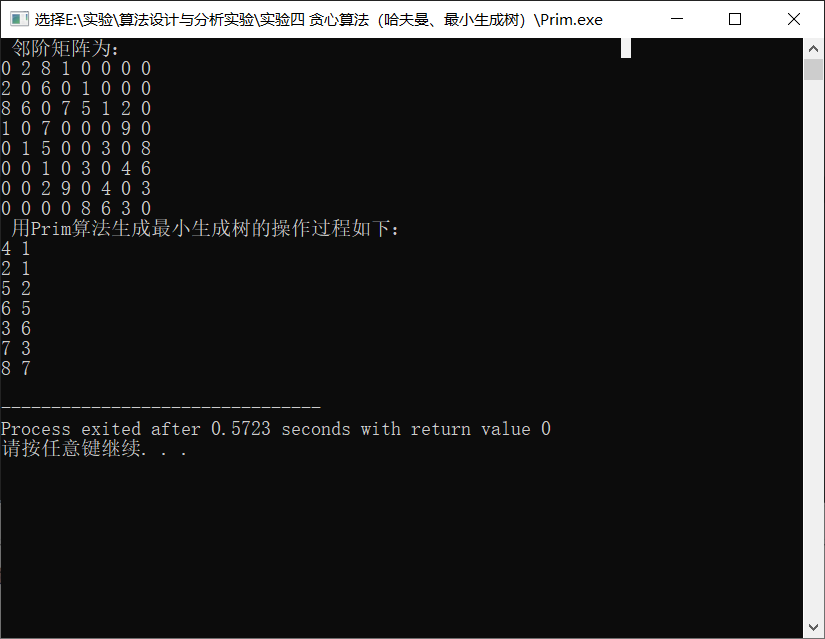
Kruskal(8,c);

return 0;

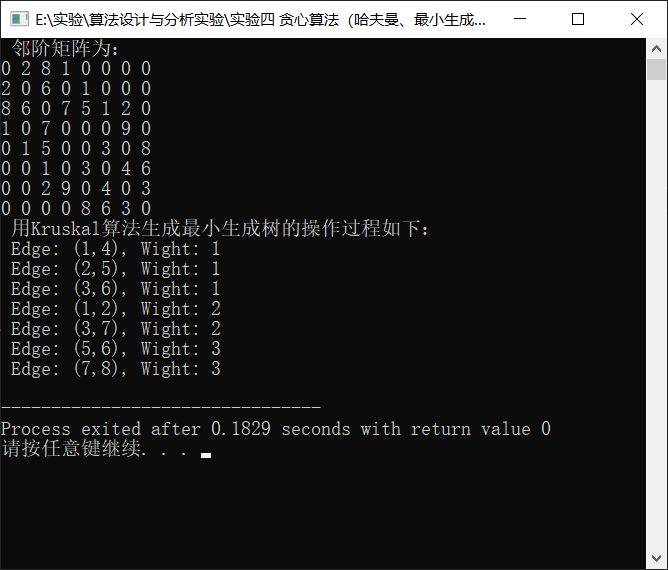
}

**3、输出结果**

Prim:



Kruskal:



**4、性能分析**

（1）Prim基本思想：

1）置S={1}；

2）只要S是V的真子集就做贪心选择：选取i（i属于S），j输入V-S， 且c[i][j]最小的边，并将定点j加入S中，直到S==V；

3） 这个过程所选的边，恰好就是最小生成树。

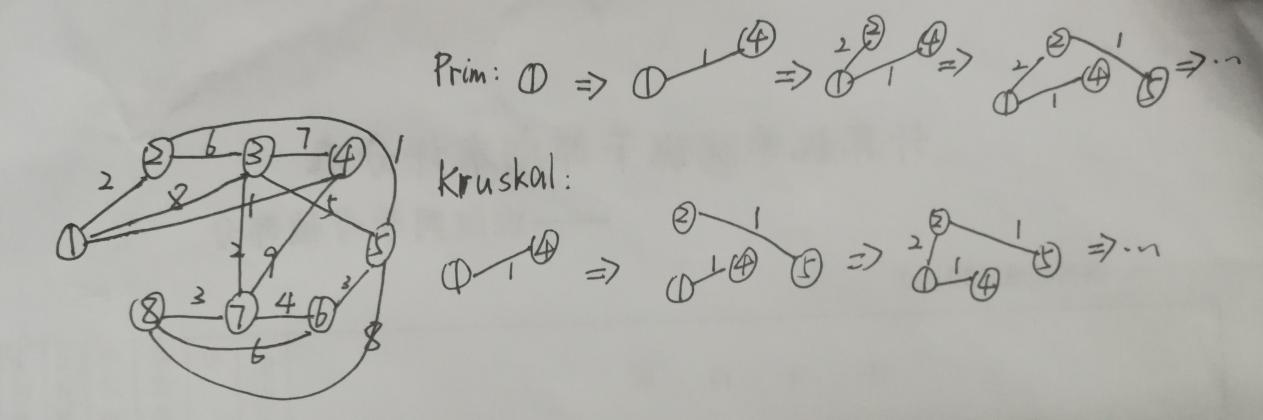
根据贪心依次选取S与V-S的权最短的边，算法步骤执行O(n)次，每次都要找连接S与V-S的最短边，查找时间为O(n)，因此，Prim的时间复杂度为O(n^2)。

（2）Kruskal算法基本思想：

1）对所有边按边权值从小到大排序；

2）依次选取最小的且不属于同一连通分支的边加入到集合中，直到加入的集合的边的数量k==(n-1)为止；

与Prim不同的是，Kruskal要先进行排序，排序算法的复杂度为 O(nlongn)，排序后再for循环遍历E数组，选取符合条件的最小权边，复杂度为 O(n)。因此，Kruskal的时间复杂度为 O(nlogn)。



**5、实验体会**

（1）求最小生成树的基本思想和求解过程我在上学期的数据结构中已经学到，但通过这次的实验，我学会了用代码实现之前的操作，我发现明白原理和具体实现还是有一定差别的，Prim 算法和 Kruskal 算法的具体实现还是花费了我一点时间，以下是我觉得有必要阐述的问题。

（2）我认为 Prim 算法的难点在于:因为要求出V-S中的顶点和已选集合S中的顶点的权最小的边，所以要引入两个数组lowcost和closest，具体操作详见代码注释。

（3）我认为 Kruskal 算法的难点在于:需要用一个数组来存放图G中所有的边，并按边的权值由小到大排序（sort）。进行取边操作时，既要保证边权最小又要满足与已选集合不在同一分量上的条件，所以要引入一个数组 connected 给不同的连通分量标号，以此判断两个顶点是否连通。

**问题三**

**1、问题描述**

课本p109,4-4磁盘文件最优存储问题。

**2、代码实现**

#include<iostream>

#include<algorithm>

using namespace std;

/\*贪心算法-磁盘文件最优存储问题\*/

bool cmp(int a,int b){

return a>b;

}

void Sort(int n,int a[]){

int i,k,mid,b[n];

sort(a,a+n,cmp); //递减

mid=n/2;

b[mid]=a[0]; //中间的最大

for(i=1,k=1;i<n;i++,k++){ //a的值从中间往两边依次减少

b[mid-k]=a[i];

i++;

if(i!=n)

b[mid+k] = a[i];

}

for(i=0;i<n;i++) //b->a

a[i]=b[i];

}

//第k个文件的检索概率为 pi = ak / ∑ai

void minStore(int n,int a[]){

int i,j,sum = 0;

for(i=0;i<n;i++)

sum+=a[i];

double ans=0;

for(i=0;i<(n-1);i++){ //磁道间的检索时间 ∑(pi\*pj\*d(j-i)),0<=i<j<n

for(j=i+1;j<n;j++){

ans+=(a[i]\*1.0/sum)\*(a[j]\*1.0/sum)\*(j-i);

}

}

cout<<"最小期望检索时间:"<<ans<<endl;

}

int main(){

int n,k,mid,i;

cout<<"输入文件个数"<<endl;

cin>>n;

int a[n];

cout<<"输入文件的检索概率:"<<endl;

for(i = 0;i<n;i++){

cin>>a[i];

}

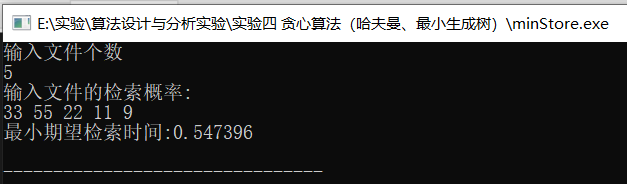
Sort(n,a);

minStore(n,a);

return 0;

}

**3、输出结果**



**4、性能分析**

要先将文件的检索概率排序，复杂度为 O(nlogn)；

利用贪心求检索时间需要用到两个for循环，复杂度为 O(n^2)；

综上，磁盘文件最优存储路径问题的时间复杂度为 O(n^2)。

**5、实验体会**

磁盘文件最优存储路径问题很显然适用于贪心，我认为本实验的难点在于排列顺序与以往不同，这个问题我一开始没有察觉，多次调试后发现，由

磁道间的检索时间 = ∑(pi\*pj\*d(j-i)),0<=i<j<n

可知它的排序不是严格单调排列，而是将相应值的分布从中间往两边依次减少排列。解决了这个问题，之后的实现就很简单了。