#include<stdio.h>

#include<string.h>

#include<math.h>

#define MaxLength 30

#define MaxSpace 100

#define MaxBit 30

#define MaxWeight 10000

#define MaxLeaf 30

#define MaxNode MaxLeaf\*2-1

typedef struct {

float weight;

int parent;

int lchild;

int rchild;

char value;

}HNodeType;

typedef struct {

int bit[MaxBit];

int start;

}HCodeType;

/\*生成哈夫曼树\*/

void HuffmanTree(HNodeType HuffmanTree[],int num) {

int i, j, x1, x2, n;

float m1,m2;

n=num;

/\*初始化\*/

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

HuffmanTree[i].weight = 0;

HuffmanTree[i].parent = -1;

HuffmanTree[i].lchild = -1;

HuffmanTree[i].rchild = -1;

HuffmanTree[i].value=' ';

}

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

if(i==0)

printf("输入字符和出现概率:");

scanf("%c%f", &HuffmanTree[i].value,&HuffmanTree[i].weight);

getchar();

}

/\*连接节点\*/

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

m1 = m2 = MaxWeight;

x1 = x2=0;

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

if (HuffmanTree[j].weight < m1&&HuffmanTree[j].parent == -1) {

m2 = m1;

x2 = x1;

m1 = HuffmanTree[j].weight;

x1 = j;

}

else if (HuffmanTree[j].weight < m2&&HuffmanTree[j].parent == -1) {

m2 = HuffmanTree[j].weight;

x2 = j;

}

}

HuffmanTree[x1].parent = n + i;

HuffmanTree[x2].parent = n + i;

HuffmanTree[n + i].lchild = x1;

HuffmanTree[n + i].rchild = x2;

HuffmanTree[n + i].weight = HuffmanTree[x1].weight + HuffmanTree[x2].weight;

}

}

/\*生成哈夫曼编码\*/

void HaffmanCode(HNodeType Hutree[],HCodeType HuffCode[],int n) {

HCodeType cd;

int i, j, c, p,length;

float sum,sum1,sum2=0;

sum1=0;

HuffmanTree(Hutree,n);

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

cd.start = n - 1;

c = i;

p = Hutree[c].parent;

while (p != -1) {

if (Hutree[p].lchild == c) cd.bit[cd.start] = 0;

else cd.bit[cd.start] = 1;

cd.start--;

c = p;

p = Hutree[c].parent;

}

for (j = cd.start + 1; j < n; j++) HuffCode[i].bit[j] = cd.bit[j];

HuffCode[i].start = cd.start;

}

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

printf("\n字符%c的哈弗曼编码为:",Hutree[i].value);

length=0;

for(j=HuffCode[i].start+1;j<n;j++) {

printf("%d",HuffCode[i].bit[j]);

length++;

}

sum2+=Hutree[i].weight\*length;

}

}

void Coding(HNodeType Hutree[],char text[],char result[]){

int i,j,p,head;

p=Hutree[0].parent;

j=0;

/\*查找头结点\*/

while(p!=-1){

head=p;

p=Hutree[p].parent;

}

p=head;

for(i=0;i<strlen(text);i++){

if(text[i]=='0') p=Hutree[p].lchild;

else if(text[i]=='1') p=Hutree[p].rchild;

if(Hutree[p].lchild==-1&&Hutree[p].rchild==-1){

result[j]=Hutree[p].value;

j=j+1;

p=head;

}

}

}

void transcording(char text[],HNodeType Hutree[],HCodeType HuffCode[],int n){

int m,length,i,j,x;

char c;

printf("哈夫曼码为:");

for(m=0;m<strlen(text);m++){

c=text[m];

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

if(c==Hutree[i].value) x=i;

}

for(j=HuffCode[x].start+1;j<n;j++) printf("%d",HuffCode[x].bit[j]);

}

}

int main() {

HNodeType Hutree[MaxNode];

HCodeType HuffCode[MaxNode];

char text[MaxSpace];

char result[30];

char text2[30];

int flag,n,i=0;

char c;

printf("输入叶子数:");

scanf("%d",&n);

getchar();

HaffmanCode(Hutree,HuffCode,n);

printf("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("1.haffman编码转文本(译码)\n2.文本转haffman码(编码)\n3.结束程序\n");

printf("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

do{

printf("\n请输入要进行的操作:");

scanf("%d",&flag);

getchar();

if(flag==1){

printf("输入Haffman码:");

scanf("%s",text);

getchar();

Coding(Hutree,text,result);

printf("译码文本:",strlen(result));

for (i=0;i<strlen(result);i++) {

printf("%c", result[i]);

}

printf("\n");

}

if(flag==2){

printf("输入要转码的文本:");

scanf("%s",text2);

getchar();

transcording(text2,Hutree,HuffCode,n);

printf("\n");

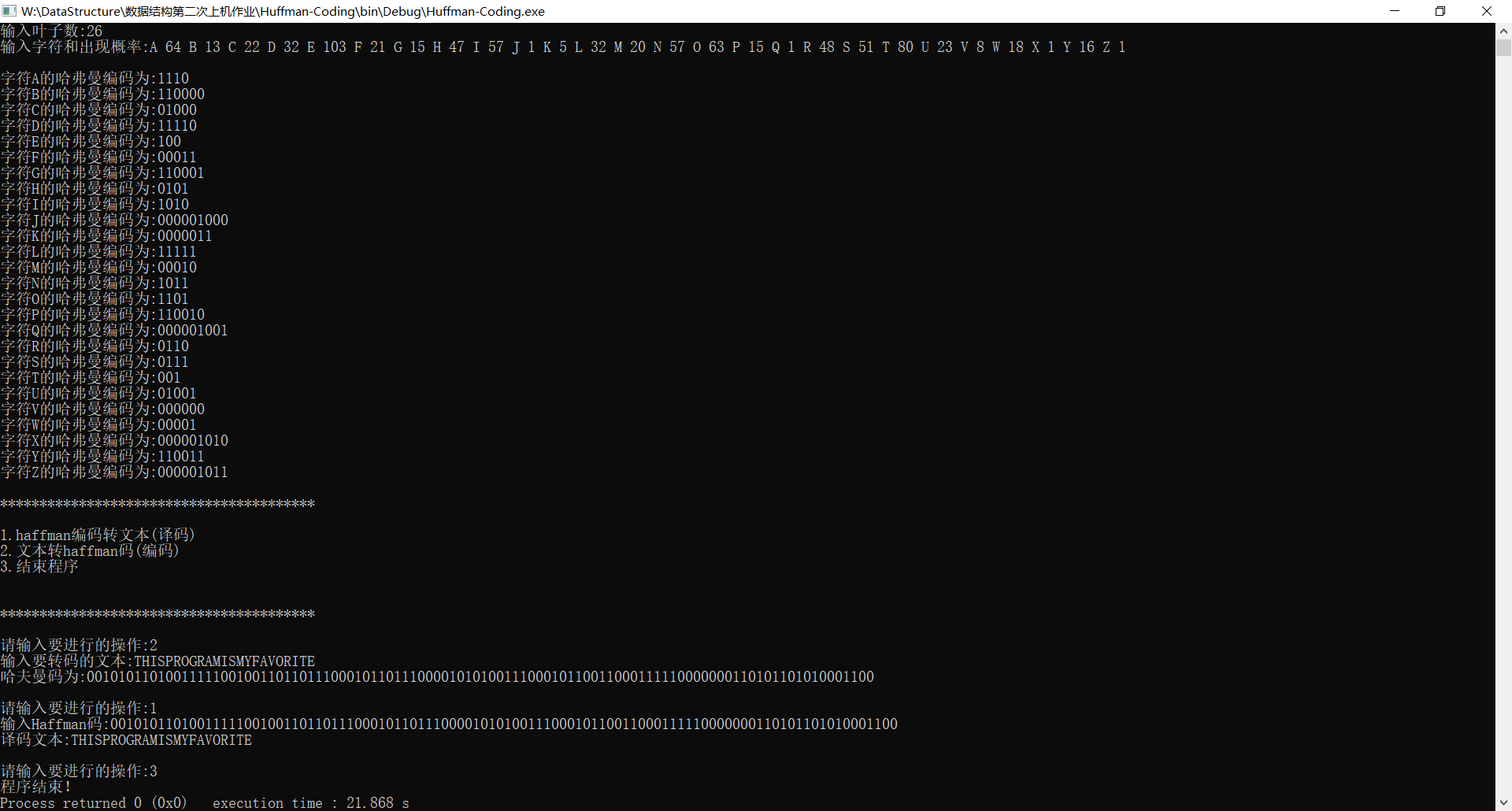
}

}while(flag!=3);

printf("程序结束！");

return 0;

}



#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MaxSize 20

typedef char VertexType;

typedef struct Graph { //定义图的邻接矩阵表示法结构

VertexType ver[MaxSize+1];

int edg[MaxSize][MaxSize];

}Graph;

typedef struct gEdge { //定义一个边集结构，用来存储图的所有边信息

int begin;

int end;

int weight;

}gEdge;

void CreateGraph( Graph \*g ) //邻接矩阵法图的生成函数

{

int i = 0;

int j = 0;

int VertexNum;

VertexType Ver;

printf("请输入图的顶点:\n");

while( '\n' != (Ver=getchar()) ) {

g->ver[i++] = Ver;

}

g->ver[i] = '\0';

VertexNum = strlen(g->ver);

printf("请输入相应的的邻接矩阵:\n");

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

for( j=0; j<VertexNum; j++ )

scanf("%d", &g->edg[i][j]);

}

int CalVerNum( Graph g ) //求图的顶点数

{

return strlen(g.ver);

}

int CalEdgNum( Graph g ) //获取图的边数

{

Graph p = g;

int count = 0;

int VertexNum = CalVerNum( p );

for( int i=0; i<VertexNum; i++ )

for( int j=i; j<VertexNum; j++ ) //邻接矩阵对称，计算上三角元素和即可

if( 0 != p.edg[i][j] )

count++;

return count;

}

gEdge \*CreateEdges( Graph g ) //生成图的排序过的边集数组

{

int i, j;

int k = 0;

int EdgNum = CalEdgNum( g );

int VertexNum = CalVerNum( g );

gEdge temp;

gEdge \*p = (gEdge \*)malloc(sizeof(gEdge)\*EdgNum);

for( i=0; i<VertexNum; i++ ) //边集数组初始化,同样只考虑上三角矩阵

for( j=i; j<VertexNum; j++ )

if( 0 != g.edg[i][j] ) {

p[k].begin = i;

p[k].end = j;

p[k].weight = g.edg[i][j];

k++;

}

for( i=0; i<EdgNum-1; i++ ) //对边集数组进行选择排序

for( j=i+1; j<EdgNum; j++ )

if( p[i].weight > p[j].weight ) {

temp = p[i];

p[i] = p[j];

p[j] = temp;

}

return p;

}

//Kruskal算法生成MST

void Kruskal( Graph g )

{

int VertexNum = CalVerNum( g );

int EdgNum = CalEdgNum( g );

gEdge \*p = CreateEdges( g );

int \*index = (int \*)malloc(sizeof(int)\*VertexNum); //index数组，其元素为连通分量的编号，index[i] = index[j] 表示编号为i 和 j的顶点在同一个连通分量中，反之则不在同一个连通分量

int \*MSTEdge = (int \*)malloc(sizeof(int)\*VertexNum-1); //MSTEdge数组，用来存储已确定的MST的边的编号，共VertexNum-1条边

int k= 0;

int WeightSum = 0;

int IndexBegin, IndexEnd;

int i=0;

for(i=0; i<VertexNum; i++ ) //初始化所有index = -1

index[i] = -1;

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

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

if( !(index[p[j].begin]>=0 && index[p[j].end]>=0 && index[p[j].begin]==index[p[j].end]) ) { //找到当前还没加入到同一个连通分量且权值最下的边

MSTEdge[i] = j; //将其加入到MST中去

if( (-1 == index[p[j].begin]) && (-1 == index[p[j].end]) ) //该边的两个顶点都没加入过任何一个连通分量

index[p[j].begin] = index[p[j].end] = i;

else if( (-1 == index[p[j].begin]) && (index[p[j].end] >= 0) ) { //该边的尾end已在一个连通分量，头begin未加入过任何连通分量

index[p[j].begin] = i;

IndexEnd = index[p[j].end];

for(int n=0; n<VertexNum; n++ )

if( index[n] == IndexEnd )

index[n] = i;

}

else if( (-1 == index[p[j].end]) && (index[p[j].begin] >= 0) ) { //该边的头begin已在一个连通分量，尾end未加入过任何连通分量

index[p[j].end] = i;

IndexBegin = index[p[j].begin];

for(int n=0; n<VertexNum; n++ )

if( index[n] == IndexBegin )

index[n] = i;

}

else {

IndexEnd = index[p[j].end];

IndexBegin = index[p[j].begin];

for(int n=0; n<VertexNum; n++ ) //该边的两个顶点都已经存在于两个不同的连通分量中

if( index[n] == IndexBegin || index[n] == IndexEnd )

index[n] = i; //将该连通分量编号全部修改为相同的值

}

break;

}

}

}

printf("最小生成树的边为:\n"); //输出最小生成树的边

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

printf("%c--%c %d\n", g.ver [p[MSTEdge[i]].begin] , g.ver [p[MSTEdge[i]].end] ,p[MSTEdge[i]].weight);

}

}

int main()

{

Graph g;

CreateGraph( &g );

Kruskal( g );

return 0;

}

