代码：

#include"stdio.h"

#include"stdlib.h"

#include"string.h"

#define M 30 /\*预定义图的最大顶点数\*/

#define FINITY 5000 //用 5000代替无穷

//最小生成树算法的结构体

typedef struct {

char vexs[M]; //顶点的数据域

int edges[M][M]; //定义的邻接矩阵

int n, e; //图中的顶点数和边数

}Mgraph;

typedef struct edgedata

{

int beg, en; //beg,en都是顶点序号

int length; //边长

}edge;

void create(Mgraph \*g, char \*filename, int c) //c = 0表示创建无向图

{

int weight = 0; //边的权值

int front, rear; //变得前驱和后驱

FILE \*file;

file = fopen(filename, "r"); //从文件读取图的边信息

if (file)

{

fscanf(file, "%d%d\n", &g->n, &g->e); //读取图的顶点数与边数

for (int i = 0; i < g->n; i++)

{

fscanf(file, "%c", &g->vexs[i]); //读取图中顶点的信息域

}

for (int i = 0; i < g->n; i++)

{

for (int j = 0; j < g->n; j++)

{

if (i == j)

g->edges[i][j] = 0; //对角线上的均为0

else

g->edges[i][j] = FINITY; //没有赋值的域均加成无

}

}

for (int k = 0; k < g->e; k++)

{

fscanf(file, "%d%d%d", &front, &rear, &weight);

g->edges[front][rear] = weight; //只些相应顺序的，即有向图

if (c == 0)

g->edges[rear][front] = weight; //对称图形，即为无向图

}

fclose(file);

}

else

{

g->n = 0;

printf("文件打开失败！\n");

}

}

//最小生成树的prim算法

void prim(Mgraph g, edge tree[M - 1])

{

edge x;

/\*

min 保存最小的权值

s 保存排序后当前节点，用于选择排序，就像选择排序中保存下标值一样

\*/

int d, min, s, v;

for (v = 1; v <= g.n - 1; v++) //v从1开始到顶点数 - 1

{

tree[v - 1].beg = 0; //从顶点v0开始求最小生成树

tree[v - 1].en = v; //边顶点

tree[v - 1].length = g.edges[0][v]; //距离根节点的所有边的权值

}

for (int k = 0; k <= g.n - 3; k++) //依次求当前(第k条)最小两栖边，并加入TREE,基本思想就是选择排序

{

min = tree[k].length; //min用于保存最小权值

s = k; //保存交换之前的下标

for (int j = k + 1; j <= g.n - 2; j++)

{

if (tree[j].length < min)

{

min = tree[j].length; //保存最小权值

s = j; //保存排序后当前的下标

}

}

v = tree[s].en; //入选的结点

/\*交换两个结构体，就像选择排序中交换两个值一样\*/

x = tree[s];

tree[s] = tree[k];

tree[k] = x;

for (int j = k + 1; j <= g.n - 2; j++)

{

d = g.edges[v][tree[j].en]; //从邻接矩阵当中获取接下来的节点到其他节点的权值

if (d < tree[j].length) //如果d的权值小于邻接矩阵中的实际权值

{

tree[j].length = d; //赋值权值

tree[j].beg = v; //赋值

}

}

}

printf("\nPrim算法的最小生成树为：\n");

for (int j = 0; j <= g.n - 2; j++)

{

printf("\n%c---%c %d\n", g.vexs[tree[j].beg], g.vexs[tree[j].en], tree[j].length);

}

printf("\n根节点为：%c\n", g.vexs[0]);

}

//快速排序

void QuickSort(edge edges[], int left, int right) //升序算法

{

edge x;

int i, j, flag = 1;

if (left < right)

{

i = left;

j = right;

x = edges[i];

while (i < j)

{

while (i < j && x.length < edges[j].length)

j--;

if (i < j)

edges[i++] = edges[j];

while (i < j && x.length > edges[i].length)

i++;

if (i < j)

edges[j--] = edges[i];

}

edges[i] = x;

//通过递归进行另一半的排序

QuickSort(edges, left, i - 1);

QuickSort(edges, i + 1, right);

}

}

//从图g的邻接矩阵读取图的所有边信息

void GetEdge(Mgraph g, edge edges[])

{

int i, j, k = 0;

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

{

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

{

if (g.edges[i][j] != 0 && g.edges[i][j] < FINITY)

{

edges[k].beg = i;

edges[k].en = j;

edges[k++].length = g.edges[i][j];

}

}

}

}

//克鲁斯卡尔最小生成树算法

void Kruskal(Mgraph g)

{

int i, j, k = 0, ltfl;

int cnvx[M];

edge edges[M\*M]; //用于存放图的所有边

edge tree[M]; //用于存放最小生成树的边信息

GetEdge(g, edges); //读取所有的边

QuickSort(edges, 0, g.e - 1); //对边进行升序排序

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

cnvx[i] = i; //设置每一个顶点的连通分量为其顶点编号

for (i = 0; i < g.n - 1; i++) //树中共有g.n - 1条边

{

while (cnvx[edges[k].beg] == cnvx[edges[k].en]) //找到属于两个连通分量权最小的边

k++;

tree[i] = edges[k]; //将边k加入到生成树当中

ltfl = cnvx[edges[k].en]; //记录选中边的终点的连通分量编号

for (j = 0; j < g.n; j++)

{

if (cnvx[j] == ltfl)

cnvx[j] = cnvx[edges[k].beg];

}

k++;

}

printf("Kruskal算法的最小生成树是：\n");

for (j = 0; j < g.n - 1; j++)

printf("%c---%c%6d\n", g.vexs[tree[j].beg], g.vexs[tree[j].en], tree[j].length);

}

// 图的最短路径

typedef enum { FALSE, TRUE } boolean; /\*FALSE 为0，TRUE为1\*/

typedef int dist[M]; //距离向量类型

typedef int path[M]; //路径类型

/\*

Dijkstra算法求单源最短路径

g 有向图

root 根节点

p 路径 p[i] 用于保存最短路径上第i个顶点的前驱顶点的序号

d 距离向量 用于保存从源点出发的顶点到其他顶点的距离向量

\*/

void Dijkstra(Mgraph g, int root, path p, dist d)

{

int v = 0; //用于临时保存

int min = 0;

boolean final[M]; //表示当前元素是否已经求出最短路径，定义为布尔类型

for (int i = 0; i < g.n; i++) //初始化各种集合与距离向量

{

final[i] = FALSE; //全部赋值为FALSE

d[i] = g.edges[root][i];

if (d[i] < FINITY && d[i] != 0) //非自我路径和达不到的路径

p[i] = root; //前驱结点为根结点

else

p[i] = -1; //当前结点无前驱结点

}

final[root] = TRUE; //初始时s只有一个根节点root一个结点

d[root] = 0;

for (int i = 1; i < g.n; i++)

{

min = FINITY;

for (int k = 0; k < g.n; ++k) //循环找出最小边入结点

{

if (!final[k] && d[k] < min) //如果还没有求出最短路径并且存在更短的路径

{

v = k; //将初始结点替换为当前结点

min = d[k]; //min保存最短路径

}

}

printf("\n%c---%d\n", g.vexs[v], min);

if (min == FINITY) //如果得出的路径无法达到

return;

final[v] = TRUE; //该结点已经求出最短路径

//修改S到V-S中各结点的距离

for (int i = 0; i < g.n; ++i)

if (!final[i] && (min + g.edges[v][i] < d[i]))

{

d[i] = min + g.edges[v][i]; //最短路径等于之前的最短路径加上最近的最短路径

p[i] = v; //将 路径向量赋值

}

}

}

void print\_dpd(Mgraph g, path p, dist d)

{

int st[M], pre, top = -1;//定义栈st并初始化空栈

for (int i = 0; i < g.n; i++)

{

printf("\nDijkstra算法求得的最短路径：%7d,path:", d[i]); //将距离向量输出

st[++top] = i; //将i入栈

pre = st[i]; //储存当前

while (pre != -1)

{

st[++top] = pre; //让pre入栈

pre = p[pre];

}

while (top > 0)

printf("%2d", st[top--]);

}

}

int main()

{

Mgraph h;

edge tree[M - 1];

char filename[20] = "D:\\Desktop\\Test.txt";

create(&h, filename, 0);

prim(h, tree);

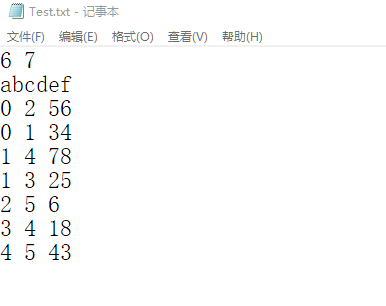
Kruskal(h);

system("pause");

return 0;

}

文件结构：



输出结果：

