数据结构课程设计



班级： 1618403

学号： 161840227

姓名： 韦 鑫

指导教师：孙 涵

目录

1.采用的数据结构 ………………………………………… 3

2.算法设计思想 …………………………………………… 3

3.关键代码 ………………………………………………… 3

4.测试数据和结果 ………………………………………… 9

5.算法的时间复杂度及其改进方法 ……………………… 10

6.结束语 …………………………………………………… 10

一、采用的数据结构

采用邻接矩阵存图

//图结构

typedef struct

{

int vn,en; //顶点数,边数

char \*\*vname; //顶点名

float \*\*A; //邻接矩阵

}GNode, \*GH;

/边结构

struct e

{

int u;

int v;

float w;

}edge[maxe]

二、算法设计思想

①Prim算法：与Dijkstra算法类似，不同的在于其更新策略，稍作修改便可。

②Kruskal算法：需要定义一个结构体存储边，对边按照边权从小到大排序，若边两端点不连通，则将该边并入最小生成树且使用并查集对标记两端点连通。

三、关键代码

//运行环境：Visual Studio 2017

#include<cstdio>

#include<cstdlib>

#include<iostream>

#include<algorithm>

#pragma warning(disable:4996)

using namespace std;

const int maxv = 5000;

const int maxe = 5000;

const int nlen = 100;

const float inf = 1 << 30 - 1;

//图结构

typedef struct

{

int vn,en; //顶点数,边数

char \*\*vname; //顶点名

float \*\*A; //邻接矩阵

}GNode, \*GH;

//边结构

struct e

{

int u;

int v;

float w;

}edge[maxe];

//定义排序规则

bool cmp(e a, e b) { return a.w < b.w; }

//根据顶点名返回对应邻接矩阵下标

int index(GH G, char vn[])

{

for (int i = 0; i < G->vn; i++)

{

if ((strcmp(G->vname[i], vn)) == 0)return i;

}

return -1;

}

//建图

void Create(GH & G)

{

G = (GH)malloc(sizeof(GNode));

if (!G)printf("给图分配空间失败！\n");

FILE \*fp;

if ((fp = fopen("data.txt", "r")) == NULL)

{

printf("打开文件失败！\n");

exit(0);

}

int vnum;

fscanf(fp,"%d",&vnum);

G->vn = vnum;

G->vname = (char \*\*)malloc(G->vn \* sizeof(char \*));

for (int i = 0; i < G->vn; i++)

{

G->vname[i] = (char \*)malloc(nlen \* sizeof(char));

}

//读取顶点名

for (int i = 0; i < G->vn; i++)

{

char name[nlen];

fscanf(fp,"%s", name);

strcpy(G->vname[i], name);

}

G->A = (float \*\*)malloc(G->vn \* sizeof(float \*));

for (int i = 0; i < G->vn; i++)

{

G->A[i] = (float \*)malloc(G->vn \* sizeof(float));

}

//邻接矩阵初始化

for (int i = 0; i < G->vn; i++)

{

for (int j = 0; j < G->vn; j++)

G->A[i][j] = inf;

}

//读取边权

char un[nlen], vn[nlen];

float w;

int en = 0;

while ((fscanf(fp, "%s %s %f", un, vn, &w)) != EOF)

{

int u, v;//顶点名对应图中邻接矩阵的下标

u = index(G, un);

v = index(G, vn);

G->A[u][v] = G->A[v][u] = w;

edge[en].u = u;

edge[en].v = v;

edge[en].w = w;

en++;

}

G->en = en;

fclose(fp);

}

//Prim算法

void Prim(GH G)

{

float d[maxv];//各点到生成树集合的距离

float ans = 0;//最小生成树权值和

int pre[maxv];//记录被更新点的前驱

bool vis[maxv] = { 0 };//vis[i] = 1表示下标为i的点已经到过

for (int i = 0; i < G->vn; i++)

{

d[i] = inf;

pre[i] = -1;

}

d[0] = 0;

for (int i = 0; i < G->vn; i++)

{

int u = -1;

float Min = inf;

for (int i = 0; i < G->vn; i++)

{

if (!vis[i] && d[i] < Min)

{

u = i;

Min = d[i];

}

}

if (u == -1)return;//说明剩下的点不连通

vis[u] = 1;

ans += d[u];

for (int v = 0; v < G->vn; v++)

{

if (!vis[v] && G->A[u][v] < d[v] && G->A[u][v] != inf)

{

d[v] = G->A[u][v];

pre[v] = u;

}

}

}

printf("最小生成树的权值之和：%.2f\n", ans);

printf("用到的边：\n");

for (int i = 0; i < G->vn; i++)

{

if (pre[i] != -1)

{

printf("%s ---- %s\n", G->vname[i], G->vname[pre[i]]);

}

}

}

//Kruskal算法

//并查集

int fa[maxv];

int findeFa(int x)

{

if (x == fa[x])return x;

else return (fa[x] = findeFa(fa[x]));//路径压缩

}

void Kruskal(GH G)

{

//并查集初始化

for (int i = 0; i < G->vn; i++)fa[i] = i;

float ans = 0; //最小生成树边权值之和

int num = 0,temp[maxe]; //记录生成树边的数量,temp[]记录用到的边的下标

bool flag = 0; //1表示不存在最小生成树

//边排序

sort(edge, edge + G->en, cmp);

//由并查集初始化得一开始所有顶点都不连通(即父节点均不等)

//由边权值递增的顺序，若边两端点不连通，则连接

for (int i = 0; i < G->en; i++)

{

int fau = findeFa(edge[i].u);

int fav = findeFa(edge[i].v);

//无共同父节点说明两端点不连通

if (fau != fav)

{

fa[fau] = fav;

ans += edge[i].w;

temp[num] = i;

num++;

if (num == G->vn - 1)break;

}

}

if (num != G->vn - 1)flag = 1;

if (flag == 0)

{

printf("最小生成树的权值之和：%.2f\n", ans);

printf("用到的边：\n");

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

{

printf("%s ---- %s\n", G->vname[edge[temp[i]].u], G->vname[edge[temp[i]].v]);

}

}

else printf("无法形成最小生成树！\n");

}

//以邻接矩阵的形式输出图

void Gprint(GH G)

{

for (int i = 0; i < G->vn; i++)

{

for (int j = 0; j < G->vn; j++)

{

if (G->A[i][j] != inf)printf("%-8.2f", G->A[i][j]);

else printf("X ");

}

printf("\n");

}

}

int main()

{

//存图测试

GH G;

Create(G);

printf("Prim算法：\n");

Prim(G);

printf("\n");

printf("Kruskal算法：\n");

Kruskal(G);

printf("\n");

system("pause");

return 0;

}

四、测试数据和结果

data.txt:

6

v1

v2

v3

v4

v5

v6

v1 v2 6

v1 v3 1

v1 v4 5

v2 v3 5

v2 v5 3

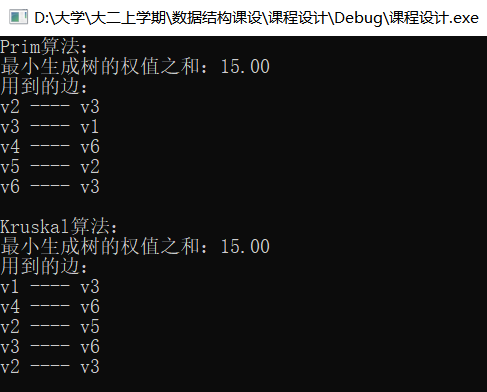
v3 v5 6

v3 v4 5

v3 v6 4

v4 v6 2

v5 v6 6



五、算法的时间复杂度即改进方法

Prim算法:O(n\*n) n为顶点数

Kruskal算法:O(e\*loge) e为边数

通过使用优先级队列可以对Prim算法进行堆优化，使其复杂度下降到O(n\*logn)，实际上Kruskal算法在此已经过优化，即使用sort()对其边进行了快排，若按照常规的选择排序则复杂度为O(e\*e)。

六、结束语

代码共238行

通过本次实验，可以思考得到，当一个图顶点数n较小，而边数e较大时，即该图为稠密图时，可以考虑使用Prim算法求解其最小生成树；而顶点数n较大，而边数e较小时，即该图为稀疏图时，考虑使用Kruskal算法求解最小生成树。这说明了不同的问题，需要具体思考哪一种算法更适合对其进行求解。