贪心法：Kruskal

曹桢 2014211302 学号：2014211182 班内序号：28

**一、原理**

贪心策略适用的前提是：局部最优策略能导致产生全局最优解。

必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性，即某个状态以后的过程不会影响以前的状态，只与当前状态有关。

Kruskal算法的贪心准则：从剩下的边中选择一条不会产生环路的具有最小耗费的边加入已选择的边的集合中。

**二、问题描述**

输入无向图的邻接矩阵，用Kruskal算法求解最小代价生成树。

**三、算法**

1、将图中所有边按权值从小到大排序；

2、并查集：取权值最小的边，加入图中，判断是否形成了回路，若无，则保留此边，否则去掉该边，重取权值较小的边；

3、反复过程 2，直到全部顶点均连通为止。

**四、算法复杂度**

当图的边为e条时，算法复杂度为O(eloge)。Kruskal算法因为只与边相关，故适合求稀疏图的最小生成树

**五、改进:二分Kruskal算法**

假设G=（V,E）是连通图，其中V是G的顶点集合，E是G的边集合。MST为G的最小生成树。二分Kruskal算法的基本思想如下：

（1） 首先从图G的各边的权值组成序列P中，选择第一个权值K作为轴值，然后将序列分 割成两个子序列P1和P2(其中P1包含所有小于或等于轴值K的权值,P2包含所有大于轴值K的权值),根据权值对应于边的原则，分别得到对应于P1的边集E1和对应于P2的边

（2） 将顶点集分为|V|个等价类，每个等价类包括一个顶点；

（3） 对于图G1=(V,E1),以权值的从小到大的顺序处理各条边，如果某条边连接两个不同等 价类的顶点，则这条边被添加到MST，两个等价类被合并为一个；反复执行此过程，直到E1种各条边均被考虑过一次；

（4） 对于图G而言，此时等价类个数为1+|E2|，以权值的从小到大的顺序处理E2中各条边， 如果某条边连接图G中两个不同等价类的顶点，则这条边被添加到MST，两个等价类被合并为一个；反复执行此过程，直到E2中各条边均被考虑过一次。

**六、算法实现**

int find(int x)//并查集找x所属集合（树）的根节点

{

return p[x] == x ? x : p[x] = find(p[x]);

}

void Kruskal()

{

int k;

for(int i = 0, k=0; i < vexnum; i++)

{

for(int j = i; j < vexnum; j++)

{

if(arcs[i][j].adj != 0){

ed[k].u = i;

ed[k].v = j;

ed[k].w = arcs[i][j].adj;

k++;

}

}

}

for(int i = 0; i < vexnum; i++) p[i] = i;

sort(ed, ed+arcnum);//把边按权值从小到大排序

cout<<"The minimal spanning tree found by Kruskal is:"<<endl;

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

{

int x = find(ed[i].u);

int y = find(ed[i].v);

if(x != y)//x和y不在同一集合

{

cout << '(' << vexs[ed[i].u] << ',' << vexs[ed[i].v] << ',' << arcs[ed[i].u][ed[i].v].adj << ')' << endl;

p[x] = y;

}

}

cout<<endl;

}

**七、总结**

这次实验，我巩固了用Kruskal算法求解最小代价生成树的方法，并理解了其中的贪心的原理，认识到它的局部最优必然导致全局最优，更加深刻地理解了Kruskal算法。

**八、附录：源代码**

/\*包含头文件\*/

#include<iostream>

#include<algorithm>

#include<queue>

/\*命名空间\*/

using namespace std;

/\*宏定义\*/

const int MAXNUM = 20;//最大顶点数

const int MAXCOST = 50;//最大权值

/\*类型定义\*/

typedef int InfoType;

typedef int VRType;

typedef char VertexType;

typedef struct ArcCell

{

VRType adj;

InfoType \*info;

}ArcCell, Adjmatrix[MAXNUM][MAXNUM];

/\*图\*/

class Graph

{

public:

Graph()//构造函数

{

createGraph();

}

Graph(const Graph &t)//拷贝构造函数

{

\*this = t;

}

~Graph(){}//析构函数

/\*深度优先搜索，用于判断是否为连通图\*/

void DFS(int i)

{

visited[i] = 1;

for(int j = 0; j < vexnum; j++)

{

if(arcs[i][j].adj != 0)

{

if(!visited[j]) DFS(j);

}

}

}

void DFSTraverse()

{

static int connect = 0;

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

{

visited[i] = 0;

}

for(int j = 0; j < vexnum; j++)

{

if(!visited[j])

{

DFS(j);

connect++;

}

}

if(connect > 1)

{

cout<<"INPUT ERROR"<<endl;

exit(3);

}

}

void createGraph()//用邻接矩阵建立一个图

{

cout << "Please input the number of vertex (more than 1 and no more than " << MAXNUM << ") :";

cin >> vexnum;

if(vexnum > MAXNUM || vexnum <= 1)//顶点数超过

{

cout << "INPUT ERROR"<<endl;

exit(1);

}

cout << "Please input the name of each vertex:";

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

{

cin >> vexs[i];

visited[i] = 0;

}

cout << "Please input the adjacent matrix of this net (the cost is no more than " << MAXCOST << " and "<< '0' << "stands for no connection) :" <<endl;

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

{

for(int j = 0; j < vexnum; j++)

{

cin >> arcs[i][j].adj;

if(arcs[i][j].adj != 0) arcnum++;

}

}

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

{

for(int j = 0; j < vexnum; j++)

{

if(arcs[i][j].adj != arcs[j][i].adj)//不是无向图

{

cout<<"INPUT ERROR"<<endl;

exit(2);

}

}

}

DFSTraverse();

arcnum = arcnum/2;

}

/\*Kruskal算法\*/

int find(int x)//并查集找x所属集合（树）的根节点

{

return p[x] == x ? x : p[x] = find(p[x]);

}

void Kruskal()

{

int k;

for(int i = 0, k=0; i < vexnum; i++)

{

for(int j = i; j < vexnum; j++)

{

if(arcs[i][j].adj != 0){

ed[k].u = i;

ed[k].v = j;

ed[k].w = arcs[i][j].adj;

k++;

}

}

}

for(int i = 0; i < vexnum; i++) p[i] = i;

sort(ed, ed+arcnum);//把边按权值从小到大排序

cout<<"The minimal spanning tree found by Kruskal is:"<<endl;

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

{

int x = find(ed[i].u);

int y = find(ed[i].v);

if(x != y)//x和y不在同一集合

{

cout << '(' << vexs[ed[i].u] << ',' << vexs[ed[i].v] << ',' << arcs[ed[i].u][ed[i].v].adj << ')' << endl;

p[x] = y;

}

}

cout<<endl;

}

private:

VertexType vexs[MAXNUM]; //顶点

int vexnum, arcnum; //顶点数， 边数

Adjmatrix arcs, ar1, ar2; //邻接矩阵

int visited[MAXNUM]; //标记位

struct edge{ //边

int u, v, w; //u，v为边的顶点，w为边的权值

bool operator < (const edge & rhs) const{//重载运算符，是边按权值排序

return w < rhs.w;

}

}ed[MAXNUM\*MAXNUM];

int p[MAXNUM];

};

/\*主函数\*/

int main()

{

Graph G;

G.Kruskal();

return 0;

}