**数据结构课程设计**

**-电网造价模拟系统**

# 题目描述

假设一个城市有n个小区，要实现n个小区之间的电网都能够相互接通，构造这个城市n个小区之间的电网，使总工程造价最低。请设计一个能够满足要求的造价方案。

在每个小区之间都可以设置一条电网线路，都要付出相应的经济代价。n个小区之间最多可以有条线路，选择其中的n-1条使总的耗费最少。

# 总体思路

此题就是一个最小生成树的模板题，而构建最小生成树普遍有两种算法，即Kruskal算法和Prim算法。考虑现实情况，两个小区之间铺设电网的代价和距离和之间的地形有关，如果要构建合理的电网，每个小区地理位置和之间地形情况必须全部已知，才能构建出符合现实意义的最小代价的电网，因此可认为此题的图是稠密图，而Prim算法更适合稠密图。若不考虑现实意义，则Kruskal算法也可以。

# 代码结构设计

1. 主函数所在文件包含了系统的界面的搭建和步骤的选择
2. 头文件MSTE.h中包含了两个算法类：Prim类、Kruskal类，小根堆，边Edge结构体的定义和实现，后两者都是Kruskal算法中需要用到的。

以下是类Prim的定义，其他不做展示。

1. **template**<**class** T, **class** N>
2. **class** Prim {
3. **private**:
4. **int** num;            //节点数量
5. T\*\* net;            //邻接矩阵
6. T\* lowcost;         //其他节点到生成树的最短距离
7. **bool**\* visit;        //标记节点是否加入到树中
8. **int**\* parent;        //存储最小生成树路径
9. N\* index;           //存储节点类型
10. **int** findIndex(N x); //找到节点对应的数字编号
11. **public**:
12. Prim(**int** sz=4);             //构造函数
13. ~Prim() { makeEmpty(); }    //析构函数
14. **void** init(**const** **int**& num);  //初始化私有数组
15. **bool** nodeInput();           //输入节点
16. **bool** input();               //输入边信息
17. **int** minCost();              //求解当前距离生成树最近的节点
18. **bool** createTree(N start);   //构建最小生成树
19. **void** printTree();           //显示最小生成树
20. **void** makeEmpty();           //清空所有数组内容
21. };

# 具体实现

## 输入处理

1. 错误输入

对于操作符A~E的输入大小写不敏感，但对电网顶点严格区分大小写。规定电网节点不得低于2个，因为要生成现实意义上的一棵树，至少需要一条边，对应的需要两个顶点。对于Prim类中的节点输入函数和边输入函数，函数类型均选择bool型，以便响应主函数的错误输入。若果有错误输入，如电网边的录入出现报错，则需要全部重新录入。

由于搭建系统过程中，有很多包括功能选择和电网数据的输入，那么错误处理可以包装成一个函数，但由于string类禁用，inputClear函数减少不了多少代码编写工作量，但单单为这一错误处理编写自定义string类则代价过大，因此不封装成函数。

1. 节点与数字编号的转换

由于节点信息是用户自定义，因此有可能不是连续且从a开始编号，需要字母标签和数组存储编号

## 顺序系统界面的搭建

最外层使用while循环，变量choose用于选择算法，不同算法对操作数的输入和基本提示共用一个外while循环、内多条并行的if-else分支，但对其中具体操作根据choose进行不同的操作。

虽然要求达到的效果和并行选择的系统相似，但实际上所需的选项从小到大暗含一个先后关系，即必须先创建电网节点，再添加电网的边，再构造最小生成树，然后才能显示最小生成树。于是需要一个判断执行进度的逻辑，保证不能跳步骤。可以开一个大小为3的bool型数组，用于记录前三个步骤有无执行，针对目前进度，如果在此进度之前有进度标记为false则报错需要重新执行，若此进度正常执行，则标记为true。若执行步骤为第一步则所有进度全部清零，重新开始。

检查执行进度的函数代码如下：

1. **bool** checkProgress(**bool** is[3],**const** **char** &oper) {
2. **int** nowStep = 0, step = -1;
3. **if** (oper >= 'A' && oper <= 'D')nowStep = oper - 'A';
4. **else** **if** (oper >= 'a' && oper <= 'd')nowStep = oper - 'a';
5. **for** (**int** i = 0; i < nowStep; i++) {
6. **if** (is[i] == **false**)step = i;
7. }
8. **switch** (step) {
9. **case** 0:
10. inputClear("顶点个数未知，无法进行此操作！");
11. **break**;
12. **case** 1:
13. inputClear("电网边信息未知，无法进行此操作！");
14. **break**;
15. **case** 2:
16. inputClear("未构造最小生成树，无法进行此操作！");
17. **break**;
18. **default**:
19. **return** **true**;
20. }
21. **return** **false**;
22. }

## 最小生成树的构建和打印

### Prim算法

采用邻接矩阵作为图的存储表示。算法基本思想为：从连通网络中某一顶点出发，选择与它关联的具有最小权值的边，将其顶点加入到生成树的顶点集合中。以后每一步都从一个在顶点集合，一个不在顶点集合的各条边中选择权值最小的，将其顶点加入到顶点集合中，循环下去直到所有顶点均加入顶点集合中。而代码实现方面，则需要两个辅助数组：

Lowcost[]：存放当前生成树顶点集合内顶点到生成树外顶点的最小距离

Visit[]：记录顶点是否加入到生成树中

初始化lowcost数组为最大值，visit数组全为false。然后令起始顶点的lowcost为0，接下来循环N-1次如下操作：找出lowcost数组中最小的数据，返回的下标即此次要加入生成树的顶点编号，令对应visit数组为true表示该顶点已加入顶点集合；然后遍历邻接矩阵中以此顶点为顶点的边，并将边的另一顶点在lowcost数组中对应的位置赋值为此边的权重。

Parent数组用于实现打印最小生成树。构建最小生成树过程中，会得到加入生成树的边的两个顶点u，v，令parent[v]=u；最后遍历此数组，将有存储的下标和内容在邻接矩阵中，以两者为横纵坐标找到对应权值后输出。

Prim算法构建最小生成树的代码如下：

1. **template**<**class** T, **class** N>
2. **bool** Prim<T, N>::createTree(N start)
3. {
4. //初始化
5. **for** (**int** i = 0; i < num; i++) {
6. lowcost[i] = INT\_MAX;
7. visit[i] = **false**;
8. parent[i] = -1;
9. }
10. **if** (findIndex(start) == -1)**return** **false**;    //防止起始点不存在
11. lowcost[findIndex(start)] = 0;              //初始赋值
12. parent[findIndex(start)] = -1;
13. **for** (**int** i = 0; i < num - 1; i++) {
14. //当前和生成树相连的最小权值的边，此边不在顶点集合的顶点
15. **int** u = minCost();
16. visit[u] = **true**;    //标记顶点已加入生成树
17. //lowcost更新新增顶点相连的所有边
18. **for** (**int** v = 0; v < num; v++) {
19. **if** (net[u][v] != 0 && visit[v] == **false**
20. && net[u][v] < lowcost[v]) {
21. parent[v] = u;
22. lowcost[v] = net[u][v];
23. }
24. }
25. }
26. **return** **true**;
27. }

### Kruskal算法

准备工作：结构体Edge，类Heap，并查集。

1. Edge结构体格式为nodeA，nodeB，cost，并重载有<、>、<=、>=、=，输入输出运算符。
2. Heap类，作为构建小根堆，根节点是权重最小的边，每次添加元素都添加至末尾并向上调整，删除顶点元素都需要向下调整。并有init函数作为堆大小的初始化和重置。
3. 并查集可直接嵌套在Kruskal类中，需要一个大小为节点数量的数组存储某顶点的父节点，即相连边的另一个顶点。只需要寻找祖先的函数，可以不需要合并函数。

此算法在录入边信息时即可加入到堆中，输入完成小根堆建立完成。随后循环N-1次，每次取权重最小的边，加入生成树前用并查集判断此边两侧节点的祖先是否为同一个，若为同一个则说明加入此边后会形成环，不能加入生成树。否则就加入，并令该边起始点的父亲为该边终止点。

Kruskal类中还有一个Edge类型的数组，大小为N-1，每次有需要加入最小生成树的边，就赋值该边信息并下标自增。最后只需要遍历此数组输出即可。

Kruskal算法构建最小生成树的代码如下：

1. **template**<**class** N>
2. **bool** Kruskal<N>::createTree()
3. {
4. **int** i = 0;
5. **while** (!net.isEmpty()) {
6. Edge<N> t = net.top();                    //获取当前权值最小的边
7. net.pop();
8. **int** indexA = findIndex(t.nodeA), indexB = findIndex(t.nodeB);
9. **if** (findFa(indexA) == findFa(indexB))   //判断是否成环
10. **continue**;
11. father[indexA] = findFa(indexB);        //加入最小生成树
12. res[i++] = t;                           //边信息存入打印数组
13. }
14. **return** **true**;
15. }

# 项目小结

1. 性能分析

由于Prim算法是由邻接矩阵作为存储格式的，所以空间复杂度为，*n*为顶点个数。此算法迭代次数为N-1，每次迭代选择权值最小的边和更新新节点相邻边都需要的时间复杂度，因此此算法时间复杂度为。

而Kruskal算法由结构体为单元的数组存储，空间复杂度为，*m*为边的个数。此算法建立小根堆的时间复杂度为，在N-1次迭代中，每次迭代所进行的出堆调整操作和并查集寻找祖先的操作时间复杂度分别为和，因此总时间复杂度为，具体可以表示根据边的数量而定。

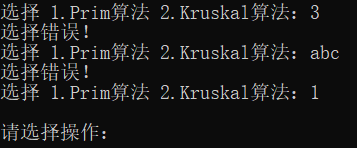
由上述分析我们可以得出当连通图为稀疏图时，即*m*数量和*n*相近时，使用数量较少的边来构建最小生成树数会得到很好的效率，Kruskal算法正是这种思想，此时时间复杂度为。而当连通图为稠密图时，即时，Kruskal算法的时间复杂度将来到，而用数量较少的点来构建最小生成树的Prim算法此时时间开销上更小。

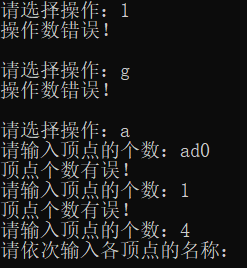
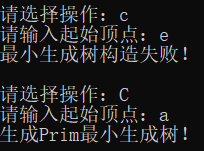
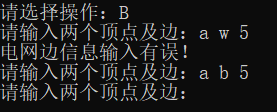
1. 算法优化

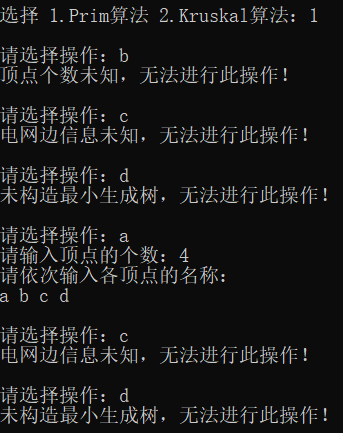
虽然Prim算法此时优势并不明显，但分析可知此算法在查找最小边和新增边时的时间消耗太大，可以对其进行堆优化。简单地说就是将添加节点相连的边给添加到一个小根堆中，这样取最小和添加边都是，而且每次迭代添加的边平均为，因此时间复杂度会降为。

而Kruskal算法中还可以对并查集进行路径压缩以进一步减小时间开销。

# 项目功能调试

算法选择错误输入

操作数错误输入

顺序执行测试

正常演示

