数据结构课程设计

项目说明文档

电网建造造价模拟系统

|  |  |
| --- | --- |
| 作者姓名： | 高逸轩 |
| 学 号： | 2053385 |
| 指导教师： | 张 颖 |
| 学院专业： | 软件学院 软件工程 |



同济大学

Tongji University

# 1项目分析

## 1.1 项目需求分析

针对于电网建造造价模拟系统，本项目在实现的过程中，考虑并且满足了以下的需求：

* 功能完善

系统所构造的用来解决电网构建问题的最小生成树应当正确无误。

* 执行效率高

针对数据量比较大的情况，本系统也应该具有在较短时间内求解出正确答案的能力。

* 代码可读性强

本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，同时进行了合理封装。

* 健壮性

当用户输入的数据不合理时，系统应当给予相应的提示而非直接报错。

## 1.2 项目要求

### 1.2.1 功能要求

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。

### 1.2.2 输入格式

不同操作、图的信息

### 1.2.3 输出格式

执行相应的操作，执行最小生成树。

### 1.2.4 项目示例

# 2 项目设计

## 2.1 数据结构设计

图是由若干给定的点及连接两点的线所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系，用点代表事物，用连接两点的线表示相应两个事物间具有这种关系。

本项目中设计了一个**图**（Graph）类。由于本项目中所述电网之间任意两个顶点不存在方向关系，因此属于无向图，即任意两个顶点之间的边都是无向边。

此外，图的存储结构有两种：一种是邻接矩阵法，另一种则是邻接表法。为了降低时间复杂度，本项目中采用了**邻接表**法存储图。

另外，为了存储图的信息，另外创建了边（Edge）和顶点（Vertex）。

## 2.2 算法设计

在本题中，我采取了Kruskal贪心算法来实现最小生成树的求解。求解最小生成树有Prim和Kruskal两种算法，以下对其进行分析：

### 2.2.1 Kruskal

Kruskal 是一种典型的**贪心**策略算法，其基本思想是：按照权值从小到大的 顺序选择 n-1 条边（利用排序），并保证这 n-1 条边不构成回路。

具体做法为：首先构造一个只含 n 个顶点的森林，然后依权值从小到大从连通网中选择边加入到森林中，并使森林中不产生回路，直至森林变成一棵树为止。 其中，在判断两点是否联通、添加边的时候，都需要借助**并查集**来实现。

### 2.2.2 Prim

Prim 算法是从图中某一个顶点出发，寻找它相连的所有结点，比较这些结点 的权值大小，然后连接权值最小的那个结点。然后将寻找这两个结点相连的所有 结点，找到权值最小的连接。重复上一步，直到所有结点都连接上。

### 2.2.3 两种算法时间复杂度的对比

设图有 n 个点，m 条边。

易分析得到，Prim 算法对于每一个点，都需要将全部点遍历一遍，以找到最近 的点加入集合。故时间复杂度为

对 Kruskal 算法分析，对 m 条边的排序所花时间为。在建立最小生成树时，可以通过路径压缩的方法，将每次并查集操作看为 ，故建立最小生成树的时间复杂度为。总时间复杂度为.

综上，可以分析得到：当图为稠密图，边数m接近时，Prim 的复杂度为 要优秀于 Kruskal 的，所以 Prim 比 Kruskal 更适合于稠密图的求解；当图为稀疏图时，边数远远少于时，则 Kruskal 的复杂度优于 Prim.

## 2.3 类设计

### 2.3.1 顶点(Vertex)

// 点

struct Vertex

{

    Vertex()

    {

        name = "";

        ancestor = -1;

    }

    string name;

    int ancestor;          // 记录在并查集中的祖先

};

每个点中记录了自己的名字，以及在并查集中的祖先。

### 2.3.2 边(Edge)

// 边

struct Edge

{

    Edge()

    {

        src = dst = -1;

        value = 0;

    }

    int src;          // 起始点编号

    int dst;          // 目标点序号

    double value;     // 边长度权重

    bool operator<(const Edge& other) { return value < other.value; };// 重载小于比较运算符

    bool operator>(const Edge& other) { return value > other.value; };// 重载大于比较运算符

};

每条边中存储了起始点和目标点的编号，以及长度（权重）。由于在Kruskal中要对边进行排序，所以重载了比较运算符，便于过后对Edge进行排序使用。

### 2.3.3 图(Graph)

class Graph

{

    template<typename T>

    friend void QuickSort(T a[], int l, int r);

public:

    // 构造函数

    Graph(int n = 0, int m = 0)

    // 析构函数

    ~Graph()

    // Kruskal算法求得最小生成树，将信息存储进MST

    bool Kruskal()

    // 初始化顶点信息

    void InitVertex()

    // 初始化边信息

    void InitEdge()

    // 打印最小生成树

    void printMST()

private:

    // 将图重置

    void MakeEmpty()

    // 寻找在图中是否有名字为s的顶点

    int SearchVertex(const string& s)

    // 并查集寻父节点，采用记忆化搜索，进行路径压缩优化

    int FindAncestor(int i)

    Graph\* MST;                   // 最小生成树

    int vertexNumber, edgeNumber; // 记录顶点数、边数

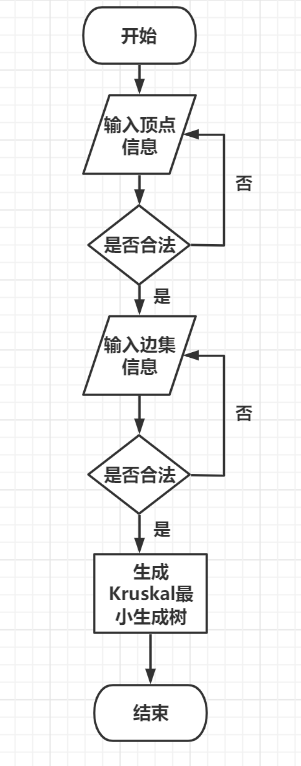
    Edge\* edges;                  // 顶点数组

    Vertex\* vertexs;              // 边数组

};

图中记录了边、顶点的信息，同时存储了图中最小生成树的信息。

## 2.4 项目流程图



# 3 核心代码介绍

## 3.1 并查集

并查集是一种树型的数据结构，用于处理一些不相交集合的合并及查询问题（即所谓的并、查）。主要作用是求连通分支数（如果一个图中所有点都存在可达关系（直接或间接相连），则此图的连通分支数为1；如果此图有两大子图各自全部可达，则此图的连通分支数为2……）；或者求图中两点的联通关系等。

在本题中，每一次Kruskal选边时，因为要判断端点的两个点是否联通来决定是否选边，所以需要频繁两个点的联通关系，在此题中采用并查集实现此功能。

### 3.1.1 祖先记录

// 点

struct Vertex

{

    Vertex()

    {

        name = "";

        ancestor = -1;

    }

    string name;

    int ancestor;          // 记录在并查集中的祖先

};

在点集中，ancestor代表某个点的祖先位置。在初始状态下，所有点各自不连通，各自的祖先即为自己；随着边的加入，一些点的祖先会合并，直至最终所有点的祖先都统一，即整个图联通，生成了最小生成树。

### 3.1.2 寻找祖先

    // 并查集寻父节点，采用记忆化搜索，进行路径压缩优化

    int FindAncestor(int i)

    {

        if (i == vertexs[i].ancestor) return i;

        return vertexs[i].ancestor = FindAncestor(vertexs[i].ancestor);

    }

在本题中，为了实现每次并查集操作近似可看做，利用了**路径压缩**来优化，避免单链表的特殊情况使邻接链表退化。（加权标记的优化效果在数据量大的时候不如路径压缩优秀，故在此未实现）

### 3.1.3 合并祖先

vertexs[f1].ancestor = f2; // f1的祖先设置为f2，两点属于同一个联通分支

将第f1个点的祖先设置为f2点，即将两者连接。

## 3.2 初始化信息

在开始时，需要对各个点、各条边的信息进行初始化。其中涉及到了健壮性问题，如节点名称不可重复、边不能超过点的n\*(n-1)/2等。利用了自行编写的getint()函数等进行了实现，后续会介绍。另外，**本项目允许多次生成图及其信息**，即多次设置点和边的信息。

### 3.2.1 初始化点

    // 初始化顶点信息

    void InitVertex()

    {

        // 将图重置

        MakeEmpty();

        // 初始化顶点数量

        cout << "请输入顶点的个数：";

        vertexNumber = getint(1, 1024, "顶点必须是1-1024的数字，请重新输入：");

        vertexs = new Vertex[vertexNumber];

        // 设置顶点名称

        cout << "请依次输入各顶点的名称：" << endl;

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

        {

            string vertexName;

            cin >> vertexName;

            // 判断姓名是否已存在

            if (SearchVertex(vertexName) != -1)

            {

                // 若已经存在，请重新输入

                cout << "顶点" << vertexName << "已存在！请重新输入：";

                i--;

            }

            // 初始化并查集祖先为自己

            else

            {

                vertexs[i].name = vertexName;

                vertexs[i].ancestor = i;

            }

        }

    }

### 3.2.2 初始化边

// 初始化边信息

    void InitEdge()

    {

        // 删除原来的边集信息

        delete[]edges;

        cin.clear();

        cin.ignore(1024, '\n');                           // 清除缓存区

        // 根据题目信息，边最多有 n\*(n-1)/2 条

        cout << "请输入边的数目：";

        edgeNumber = getint(1, vertexNumber \* (vertexNumber - 1) / 2, "根据题目信息，边最多有 n\*(n-1)/2 条，请重新输入：");

        edges = new Edge[edgeNumber];

        // 输入边集信息，通过getint函数处理了错误输入，保证了健壮性

  cout << "请依次输入" << edgeNumber << "条边的起点、终点、权重：" << endl;

        for(int i=0;i<edgeNumber;i++)

        {

            // 设置起点终点

            cout << "请输入第" << i + 1 << "条边的信息：";

            string src, dst;

            double value;

            cin >> src >> dst;

            edges[i].src = SearchVertex(src);

            edges[i].dst = SearchVertex(dst);

            // 信息异常处理

            if (edges[i].src == -1 || edges[i].dst == -1)

            {

                cout << "点信息输入错误，请重新输入：" << endl;

                cin.clear();

                cin.ignore(1024, '\n');                     // 清除缓存区

                i--;

                continue;

            }

            // 设置边权

            cin >> value;

            // 信息异常处理

            if (cin.fail() || value <= 0)

            {

                cout << "点信息输入错误，请重新输入：" << endl;

                cin.clear();

                cin.ignore(1024, '\n');                     // 清除缓存区

                i--;

                continue;

            }

            edges[i].value = value;

        }

    }

## 3.3 Kruskal求解最小生成树

// Kruskal算法求得最小生成树，将信息存储进MST

    bool Kruskal()

    {

        // 快速排序，区间[0,edgeNumber - 1]

        QuickSort(edges, 0, edgeNumber - 1);

        // 删除原来的MST信息

        delete MST;

        MST = NULL;

        // 为MST申请空间

        MST = new Graph(vertexNumber, vertexNumber - 1);

        // cnt记录已选边数量

        int cnt = 0;

        for(int i=0;i<edgeNumber;i++)

        {

            int f1 = FindAncestor(edges[i].src); // f1为边起点的祖先

            int f2 = FindAncestor(edges[i].dst); // f2为边终点的祖先

       // 若两点祖先不同，则分属不同的连通分支，需要合并，将此边加入最小生成树

            if (f1 != f2)

            {

                cnt++;                     // 已选边数+1

                vertexs[f1].ancestor = f2; // f1的祖先设置为f2，两点属于同一个联通分支

                MST->edges[cnt - 1] = edges[i];

            }

            if (cnt == vertexNumber - 1)   // 当最小生成树终已选边数=顶点数-1时，最小生成树构造完成

            {

                cout << endl;

                return true;

            }

        }

        // 若未能选满vertexNumber-1条边，则不能构成最小生成树

        delete MST;

        MST = NULL;

        cout << endl;

        return false;

    }

Kruskal是一种贪心算法。为了使图中最小生成树边权和最小，我们需要先将其按边权排序**（对边进行了快速排序，后续排序作业中会对其进行介绍，在此不做介绍）**。再依次从小到大遍历这些边，如果某条边的加入可以使非联通块的数量减少，则将其加入最小生成树MST中。最终，因为有n个点，若选择的边不够n-1条，则说明未能成功生成最小生成树。

## 3.4 输出最小生成树信息

// 打印最小生成树

    void printMST()

    {

        // 未生成成功

        if (MST == NULL)

        {

            cout << "未生成最小生成树，请先生成！" << endl << endl;

            return;

        }

        // 生成成功

        cout << "最小生成树信息如下：" << endl;

        // 输出MST中的信息

        for(int i=0;i<MST->edgeNumber;i++)

            cout << "选边信息：起点<" << vertexs[MST->edges[i].src].name << ">  终点<" << vertexs[MST->edges[i].dst].name << ">  权值：" << MST->edges[i].value << endl;

        cout << endl;

    }

## 3.5 重置图

    // 将图重置

    void MakeEmpty(){

        vertexNumber = 0, edgeNumber = 0;

        if (edges != NULL)  delete[]edges;

        if (vertexs != NULL)delete[]vertexs;

        edges = NULL, vertexs = NULL;

    }

由于当重新设置顶点信息时，需要重新设置图的信息，故编写此函数实现功能。

## 3.6 寻点

// 寻找在图中是否有名字为s的顶点

    int SearchVertex(const string& s)

    {

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

        {

            // 如果有，返回下标

            if (vertexs[i].name == s)

                return i;

        }

        // 如果没有，返回-1

        return -1;

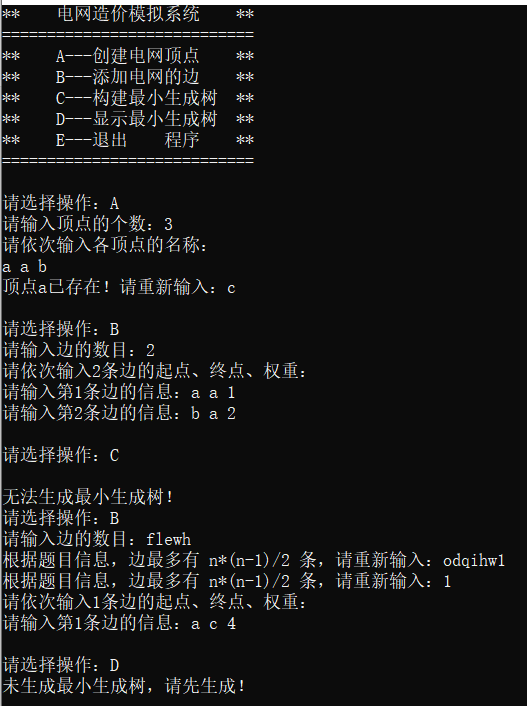
    }

在初始化点集信息时，由于需要实现健壮性，保证没有名字重复的点出现，需要对已经存储的点进行查找，查看当前是否已经存在某个名字的点；当建立边时，用户输入的为点的名字，我们需要将名字对应至点的编号。为实现以上两个功能，编写了此函数来根据名字对点进行定位。

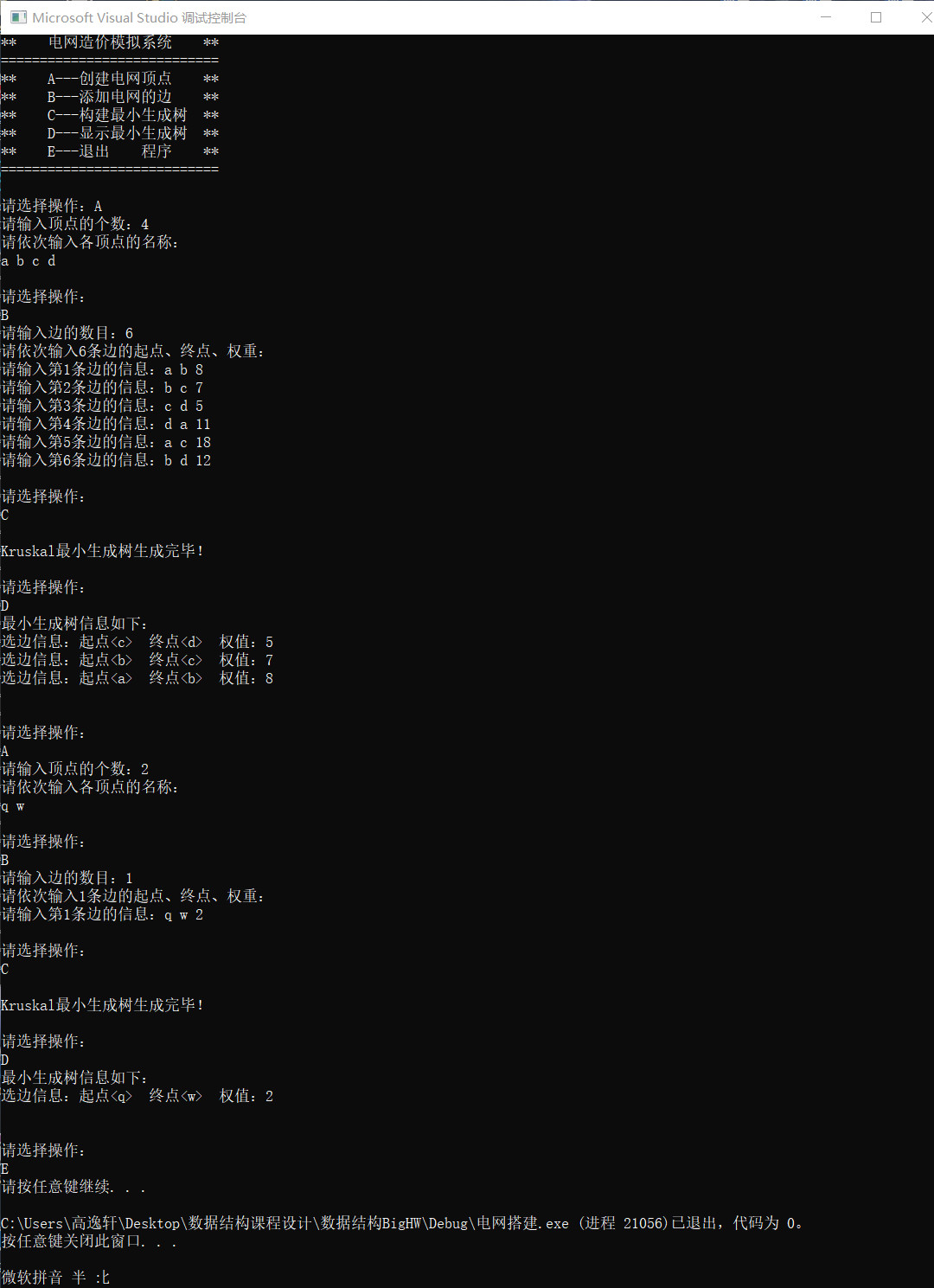
# 4 项目测试

## 4.1 健壮性测试

对建立已经存在的名字的节点、输入数据类型与所需不符合等问题实现了健壮性处理：



## 4.2 功能测试

为方便老师测试，提供了文件8\_test.txt，内含一组数据，测试了本程序的全部功能。以下为一组测试数据：

# 5 心得与体会

在本次最小生成树的作业中，我学习到了并查集及其优化的知识、快速排序 的原理及其优化的知识、Kruskal 算法和 Prim 算法的对比。在这次作业中，我得到了很好的机会，来复习高中时信息学奥赛的知识，并且将“仅会应用”提升到了 “深刻理解”的层次。并且，在不断搜索资料进行算法优化的过程中，我对自学能力的重要性有了更好的认识。另外，我对算法时间复杂度的分析能力也更上一层楼，可以判断在不同情况、极端情况时，某个算法的表现不同的原因，并对应给出了解决方法。

由于篇幅原因，报告内还有很多内容与解释没有展示，请老师和助教老师再移步源程序，在其中的注释写了每一步过程的详解。