数据结构项目文档

> 题目：电网建设造价模拟系统
>
> 指导教师：张颖
>
> 姓名：王亮
>
> 学号：1653340

一、题目分析

1. 项目简介

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。

2. 功能需求

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n（n-1）/2条线路，选择其中的n-1条使总的耗费最少。

3. 设计思路

首先分析问题，可以确定这是一个求图的最小生成树问题。

具体设计思路：

首先确定项目采用图作为抽象数据结构，确定图的存储表示，定义类的成员变量和成员函数；然后实现计算图的最小生成树的函数；最后完成主函数以验证程序的功能并得到运行结果。

二、设计

1. 数据结构设计

这是一个求图的最小生成树的问题，因此选择图作为抽象数据结构，并通过链表实现图的邻接表表示。

2. 类的设计

2.1 Edge

图中的“边”的类定义，在邻接表表示的存储结构中，是边链表中的边结点。

数据域：目标顶点、边的权值。

指针域：指向下一个边结点的指针。

```
template <typename T, typename E>struct Edge{        //边结点的定义
    int dest;                                        //边的另一顶点位置
```

```
    E cost;                                          //边上的权值
    Edge<T,E> *link;                                 //下一条边链指针
    Edge() {}                                        //构造函数
    Edge(int num, E weight){                         //构造函数
        dest = num;
        cost = weight;
        link = NULL;
    }
    bool operator != (Edge<T,E> &R)const{            //判边不等否。#只能用于同一
行的单链表中
        return (dest != R.dest);
    }
};
```

2.2 Vertex

图中的“顶点”的类定义，在邻接表表示的存储结构中，是每条边链表的头结点。

数据域：当前顶点的名字。

指针域：指向第一个边结点的指针。

```
//顶点的定义
template <typename T, typename E>struct Vertex{
    T data;                  //顶点的名字
    Edge<T,E> *adj;          //边链表的头指针
};
```

2.3 Graph

图的抽象基类定义，提供各种存储结构的图基本操作。

具体各存储结构的图，通过继承此类的方式实现。

该类中的部分纯虚函数需要在其继承类中实现。

```
// 图的抽象基类定义#有权无向图
// T为顶点的类型；E为边权值的类型，一般应为某一整数类型。
template <typename T, typename E>class Graph{
public:
    Graph(int sz){
        maxVertices = sz;
        numVertices = 0;
        numEdges = 0;
    }
    virtual ~Graph(){};
    bool GraphEmpty() const{                 //判图空否
        return (numEdges == 0);
    }
    bool GraphFull() const{                  //判图满否
```

```
        return (numVertices == maxVertices ||
            numEdges == maxVertices*(maxVertices-1)/2);//无向图，有向图不除以2
    }
    int NumberOfVertices();                  //返回当前顶点数
    int NumberOfEdges();                     //返回当前边数

    void DFS();                              //深度优先遍历图，输出所有的连通分量
    void BFS();                              //广度优先遍历图，输出所有的连通分量
    void DFSTree(Tree<T> &tree);             //立以左子女-右兄弟链表表示的DFS生成森
林。
    void Components();                       //利用深度优先搜索求非连通图的连通分量的
算法
    void Kruskal(MinSpanTree<T,E> &MST);//最小生成树
    void Prim(MinSpanTree<T,E> &MST, int startVertex = 0);

    friend istream& operator >> (istream &in, Graph<T,E> &G);
    friend ostream& operator << (ostream &out, Graph<T,E> &G);

    // 图的子类必须实现的一些接口
    virtual T getValue(int i) = 0;                          //取位置为i的顶点
中的值
    virtual E getWeight(int v1, int v2) = 0;                //返回边(v1,v2)
上的权值
    virtual bool insertVertex(const T &vertex) = 0;         //在图中插入一个
顶点vertex
    virtual bool removeVertex(int v) = 0;                   //在图中删除一个
顶点v
    virtual bool insertEdge(int v1, int v2, E weight) = 0;  //插入权值为
weight的边(v1,v2)
    virtual bool removeEdge(int v1, int v2) = 0;            //删除边(v1,v2)
    virtual int getFirstNeighbor(int v) = 0;                //取顶点v的第一个
邻接顶点
    virtual int getNextNeighbor(int v, int w) = 0;          //取v的邻接顶点w
的下一邻接顶点
    virtual int getVertexPos(const T &vertex) = 0;          //给出顶点vertex
在图中的位置
protected:
    int maxVertices;            //图中最大顶点数
    int numEdges;               //当前边数
    int numVertices;            //当前顶点数
    void DFS(int v, bool visited[]);                        //深度优先遍历图，
子过程
    void DFSTree(int v, bool visited[], TreeNode<T> *& subTree);
};
```

2.4 GraphInk : Graph

邻接表实现的图类，继承自抽象基类Graph。

成员数据：一个顶点表（表中的元素是各边链表的头结点）。

```
template <typename T, typename E>class Graphlnk : public Graph<T,E>{
//图的类定义
public:
    Graphlnk(int sz = DefaultVertices); //构造函数
    ~Graphlnk();                        //析构函数
    T getValue(int i){  //返回该邻接顶点的编号，若不存在则返回0        //取位置为i
的顶点中的值
        return (i >= 0 && i < this->NumberOfVertices()) ? NodeTable[i].data
: 0;
    }

    E getWeight(int v1, int v2);                        //返回边(v1,v2)上的权
值
    bool insertVertex(const T& vertex);                 //在图中插入一个顶点
vertex
    bool removeVertex(int v);                           //在图中删除一个顶点v
    bool insertEdge(int v1, int v2, E weight);          //在图中插入一条边
(v1,v2)
                                                        //接受一个参数，表示插
入顶点的值，返回true表示插入成功

    bool removeEdge(int v1, int v2);                    //在图中删除一条边
(v1,v2)
    int getFirstNeighbor(int v);                        //取顶点v的第一个邻接
顶点
                                                        //返回第一个邻接定点的
编号，若不存在或参数不合理则返回-1

    int getNextNeighbor(int v, int w);                  //取v的邻接顶点w的下一
邻接顶点

    int getVertexPos(const T &vertex){                  //给出顶点vertex在图
中的位置
        for (int i = 0; i < this->numVertices; i++){
            if (NodeTable[i].data == vertex){
                return i;
            }
        }
        return -1;
    }
private:
    Vertex<T,E> *NodeTable;                             //顶点表 (各边链表的头
结点)
};
```

2.5 MSTEdgeNode

最小生成树的边结点。

记录两个顶点和边的权值。

实现了对运算符的重载。

```
//最小生成树边结点的类声明
template <typename T, typename E>struct MSTEdgeNode{//T为顶点类型，其实在生成树中未用
    int tail, head;                 //两顶点位置
    E key;                          //边上的权值,为结点关键码
    MSTEdgeNode(){                  //构造函数
        tail = -1;
        head = -1;
        key = 0;
    }
    bool operator < (MSTEdgeNode<T,E> &n2) {return this->key < n2.key;}
    bool operator > (MSTEdgeNode<T,E> &n2) {return this->key > n2.key;}
    bool operator == (MSTEdgeNode<T,E> &n2) {return this->key == n2.key;}
    bool operator <= (MSTEdgeNode<T,E> &n2) {return this->key <= n2.key;}
};
```

2.6 MinSpanTree

最小生成树的类定义。

保存边的数量和树中的边。

```
//最小生成树的类定义
template <typename T, typename E>class MinSpanTree{
protected:
    MSTEdgeNode<T,E> *edgevalue;            //用边值数组表示树
    int maxSize, n;                         //数组的最大元素个数和当前个数
public:
    MinSpanTree(int sz = DefaultSize2 /*- 1*/){
        maxSize = sz;
        n = 0;
        edgevalue = new MSTEdgeNode<T,E>[sz];
        assert(edgevalue);
    }
    bool Insert(MSTEdgeNode<T,E> &item);    //将边item插入到树中，若树中节点已满，则返回false;
    void output();                          //自定义函数，顺序输出所有边
    int getNum() const {return n;}
    MSTEdgeNode<T, E>* getEdgeValue() const {return edgevalue;}
};
```

2.7 MinHeap

最小堆的类定义。

由于使用Kruskal算法和Prim算法求最小生成树的过程中，需要寻找与已找出顶点连接的最短边，需要用到最小堆。

最小堆的定义已在前面的题目中阐述过，这里不再赘述。

2.8 UFSets、Tree、SeqQueue

使用Kruskal算法求最小生成树的过程中需要使用并查集和最小堆配合实现。

通过DFS建立DFS生成森林的过程中需要用到Tree。

使用BFS求所有连通分量的过程中需要用到SeqQueue。

由于本项目仅需要使用Prim算法求最小生成树，所以这三个类在本项目中并未用到，但为了保证代码的完整性，并为了让阅读者能够理解项目结构，所以在这里列出，但不作过多赘述。

3. 主函数设计

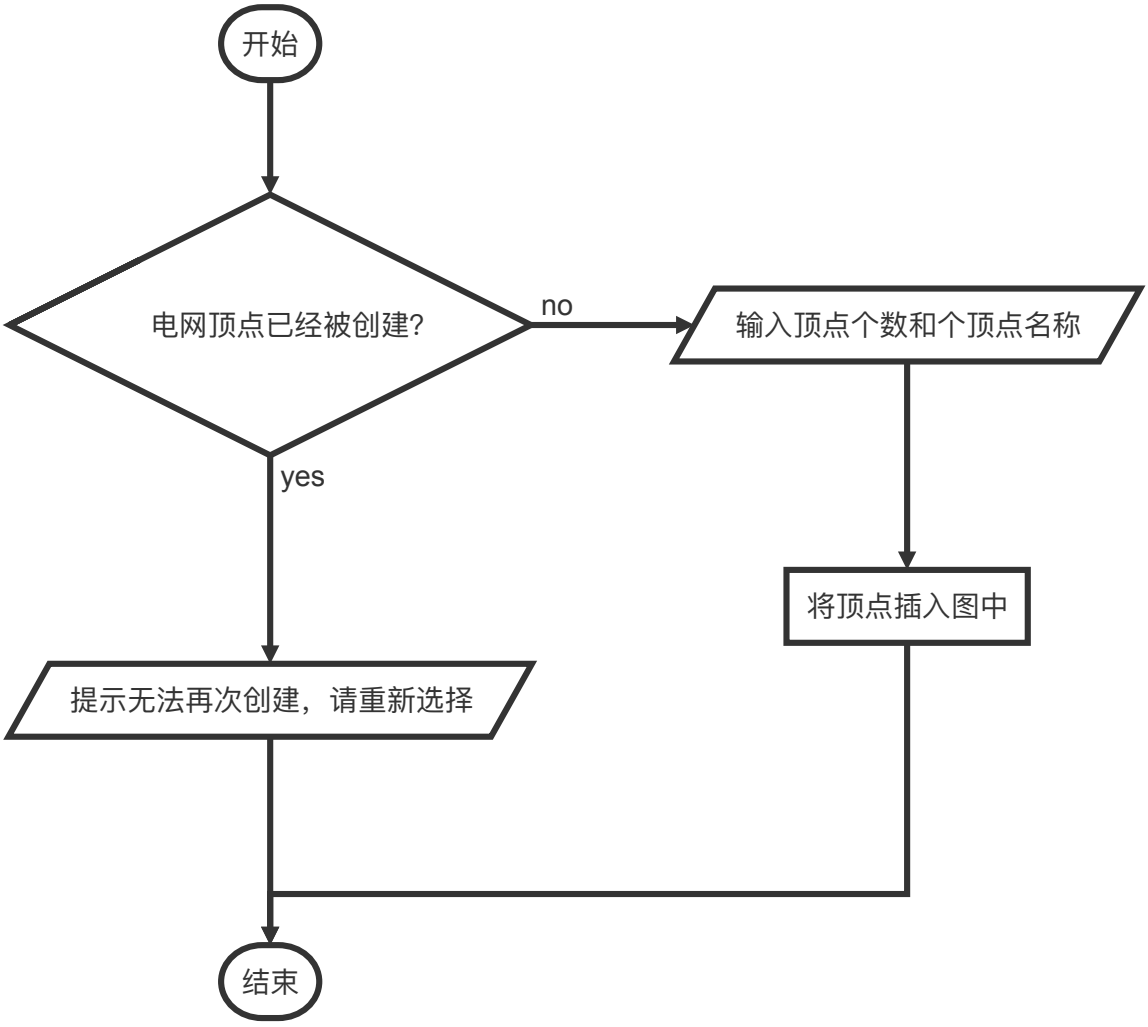
系统共提供五项功能。

通过一个循环实现不断接受用户的指令，并执行相关操作。

三、实现

1. 创建电网顶点

1.1 流程图



1.2 相关代码

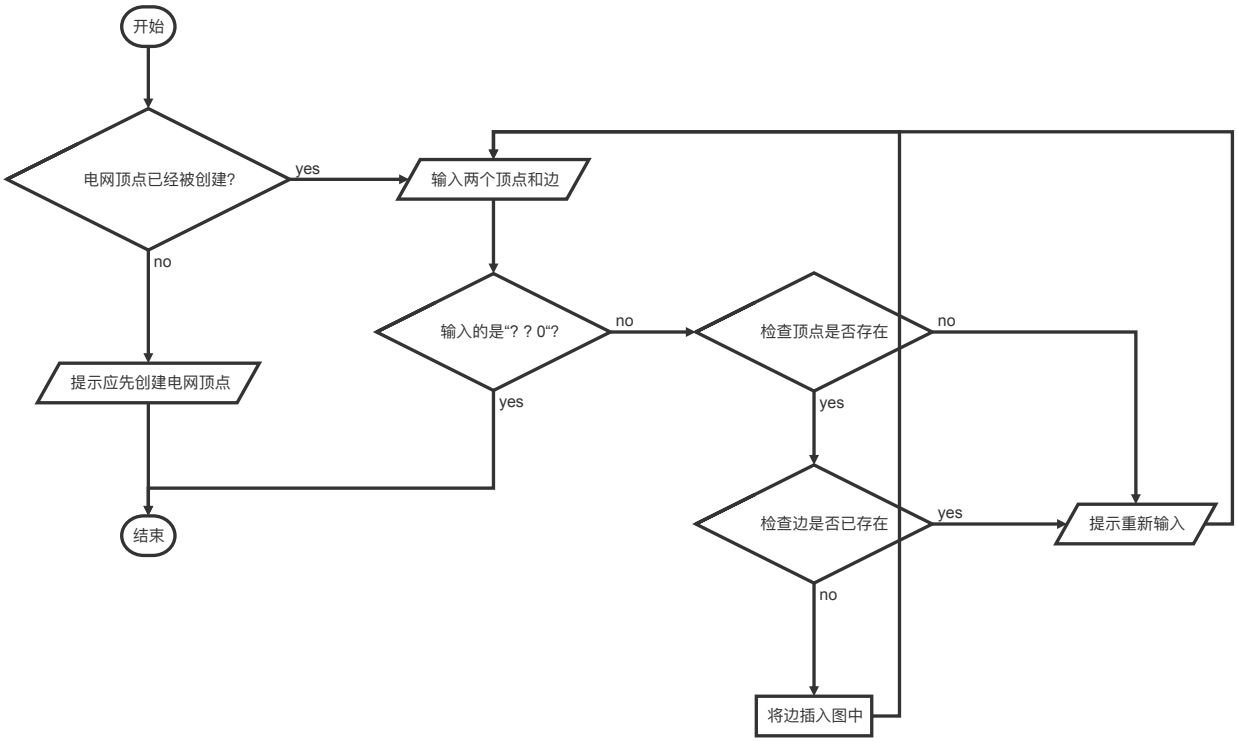
```
if (isSystemBuilt)
{
    cout << "电网顶点已经被创建，请重新选择！" <<endl;
    break;
}
else
{
    int cnt;
    int numVertices;
    string tempName = "";

    cout << "请输入顶点的个数：";
    cin >> numVertices;
    cout << "请依次输入各顶点的名称：" <<endl;
    for (cnt = 0; cnt < numVertices; cnt++)
    {
        cin >> tempName;
```

```
                    //cout << tempName <<endl;
                    grid->insertVertex(tempName);
                }
                isSystemBuilt = true;
            }
            cout << endl;
```

1.3 运行截图

2. 添加电网的边

2.1 流程图

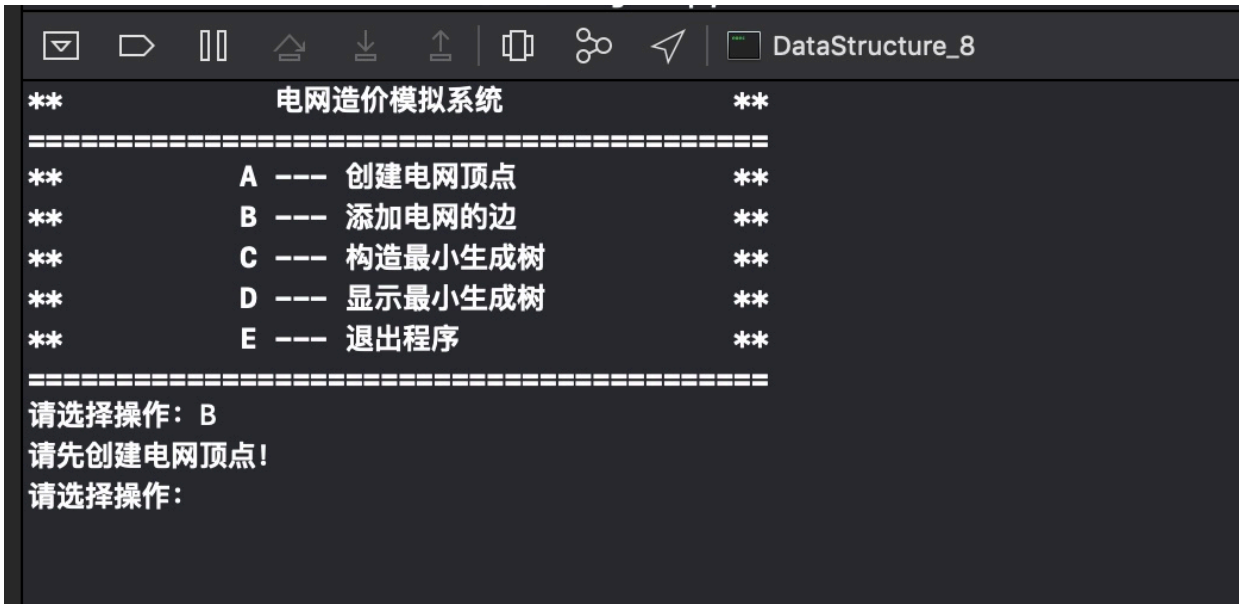


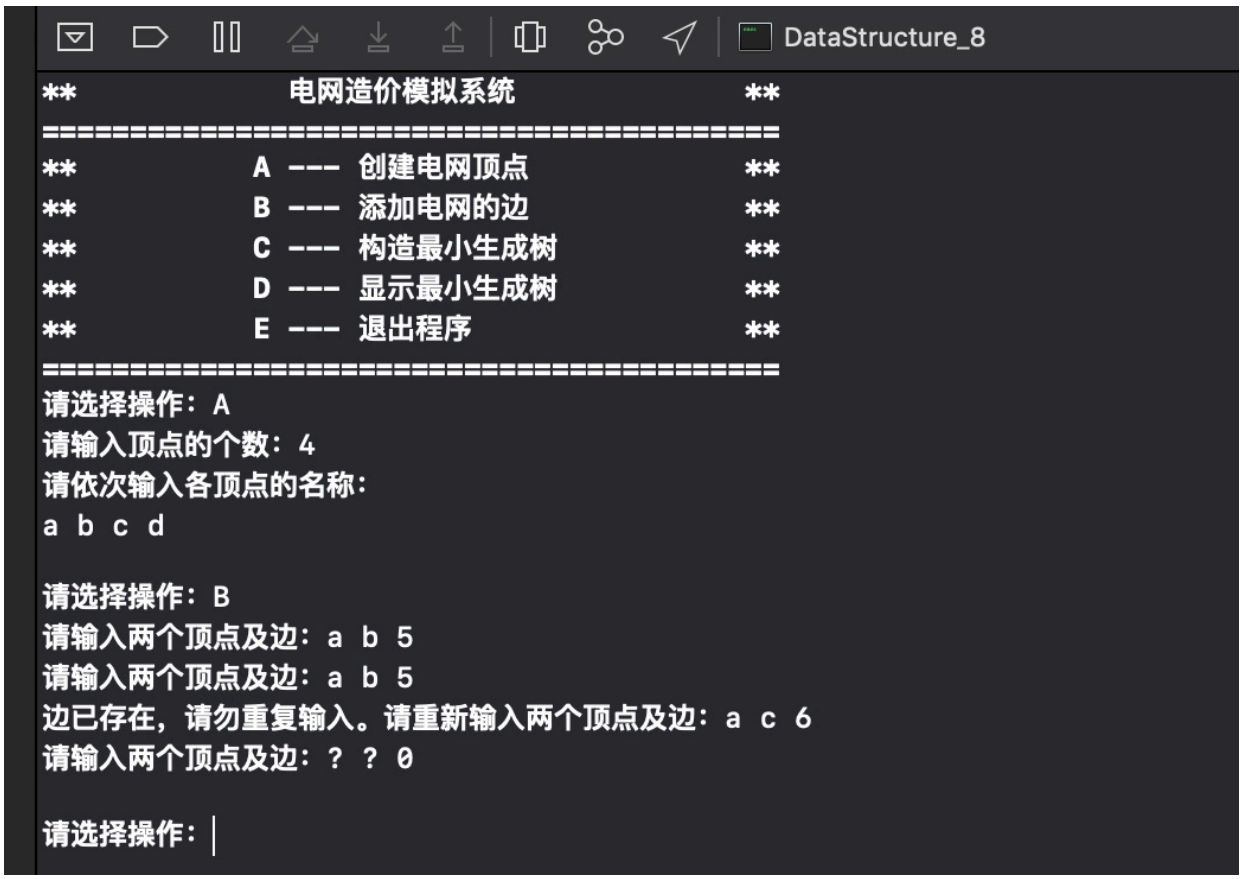
2.2 相关代码

```
                if (!isSystemBuilt)
                {
                    cout << "请先创建电网顶点！" <<endl;
                    break;
                }
                else
```

```
                {
                    string v1 = "init", v2 = "init";
                    int edge = -1;
                    bool isSuccessful = true;                    //是否成功
插入边，添加电网的边时使用，当顶点不存在或边已经被插入过时为false

                    while (true)
                    {
                        if (isSuccessful == true)
                        {
                            cout << "请输入两个顶点及边：";
                        }
                        isSuccessful = true;                     //重新置为
true
                        cin >> v1 >> v2 >> edge;
                        if (v1=="?" && v2=="?" && edge==0)
                            break;

                        //输入合法检测
                        int i_v1 = grid->getVertexPos(v1);
                        int i_v2 = grid->getVertexPos(v2);
                        if (i_v1 == -1 || i_v2 == -1)
                        {
                            cout << "顶点不存在，请重新输入两个顶点及边：";
                            isSuccessful = false;
                        }
                        else
                        {
                            isSuccessful = grid->insertEdge(i_v1, i_v2,
edge);
                            if (!isSuccessful)
                            {
                                cout << "边已存在，请勿重复输入。请重新输入两个顶
点及边：";
                            }
                        }

                    }

                }
                cout << endl;
```

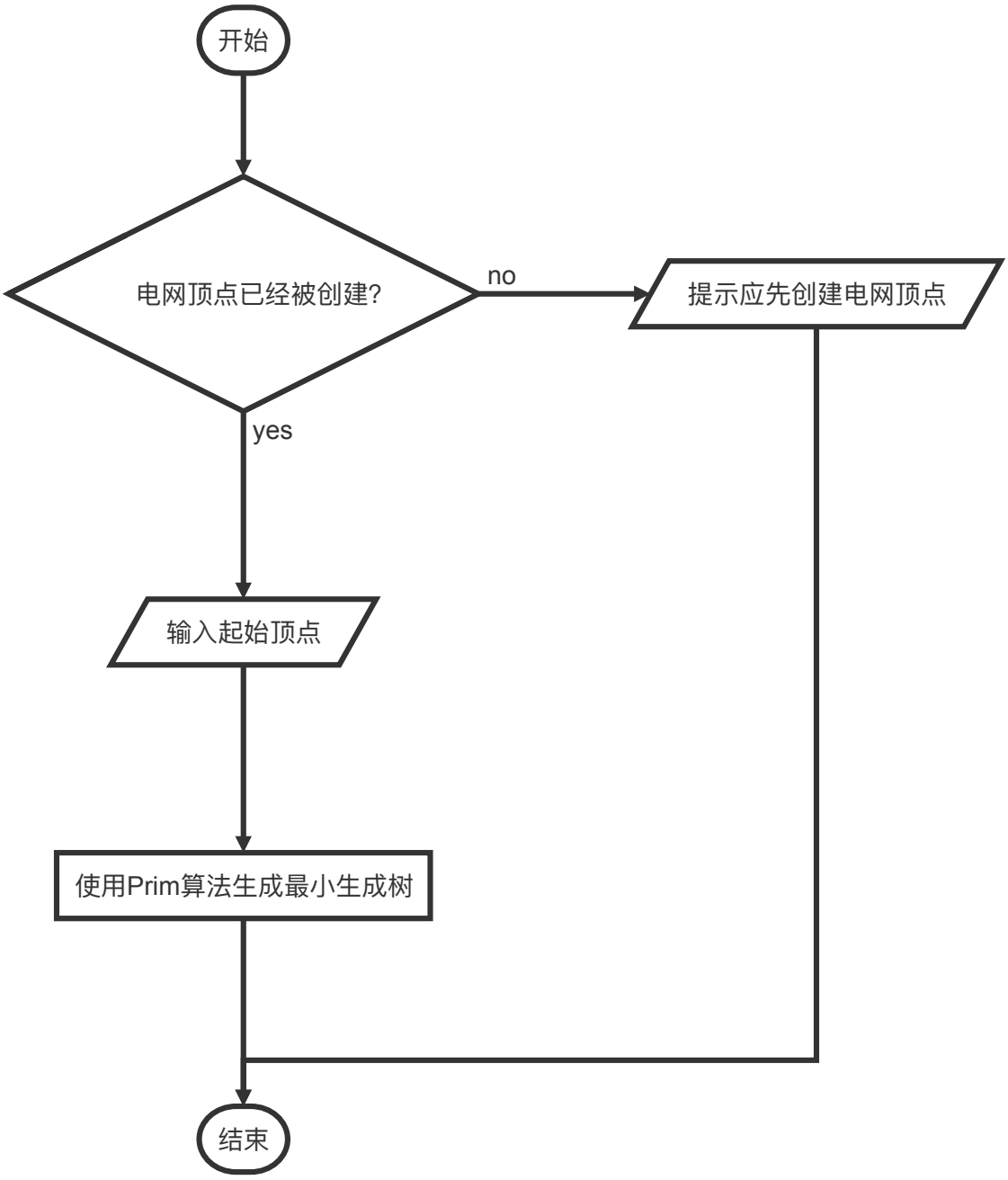
2.3 运行截图





3. 构造最小生成树

3.1 流程图

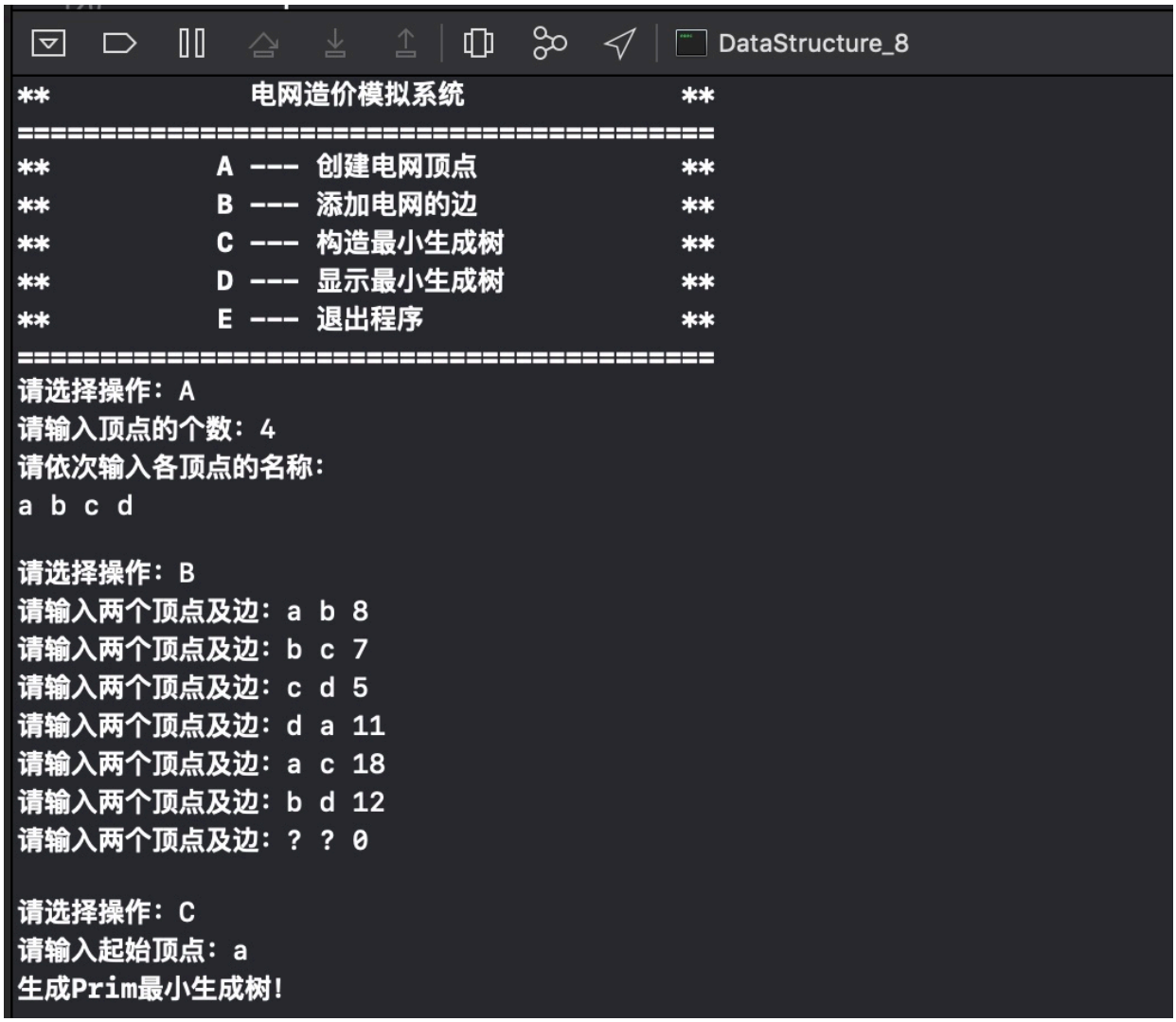


3.2 相关代码

```
                if(!isSystemBuilt)
                {
                    cout << "请先创建电网顶点! " <<endl;
                    break;
                }
                else
                {
                    string tempName;
```

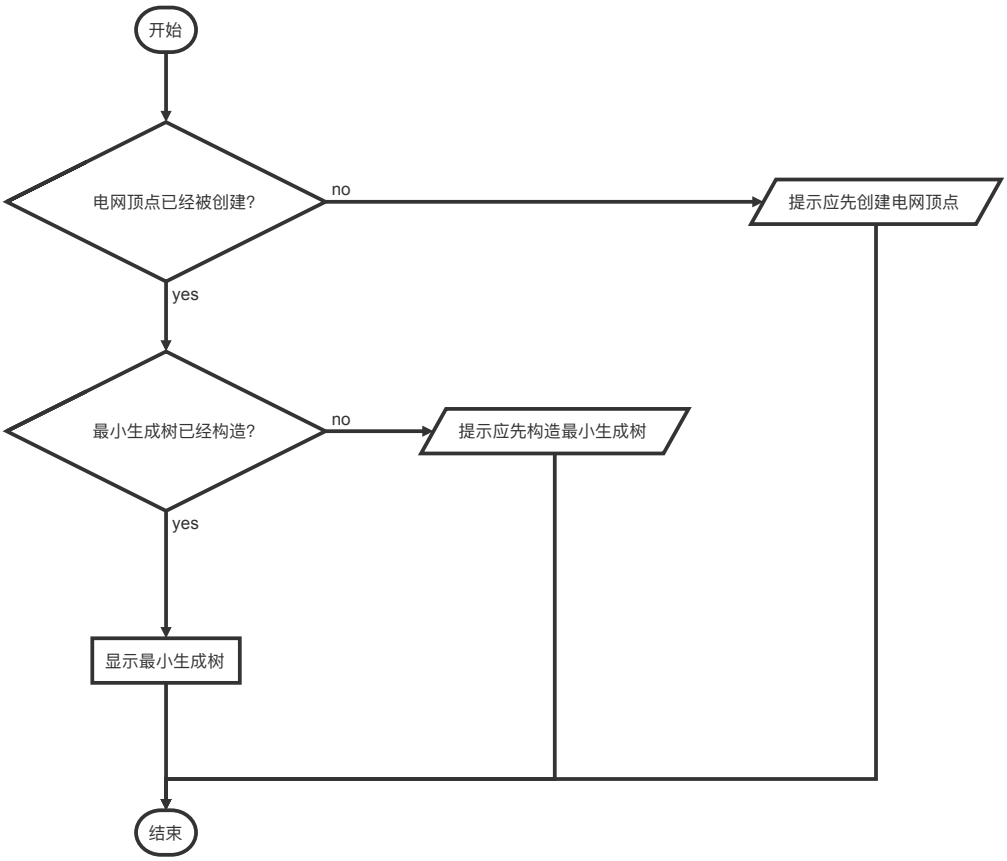
```
                cout << "请输入起始顶点: ";
                cin >> tempName;
                grid->Prim(*minTree, grid->getVertexPos(tempName));
                cout << "生成Prim最小生成树! " << endl;
            }
            cout << endl;
```

3.3 运行截图



4. 显示最小生成树

4.1 流程图

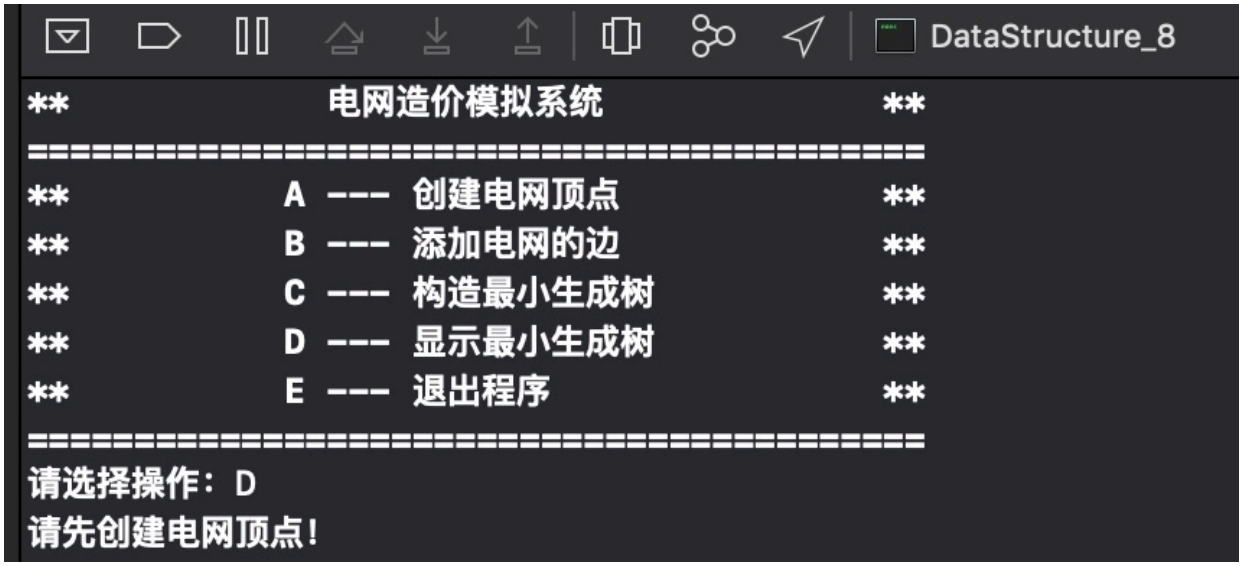


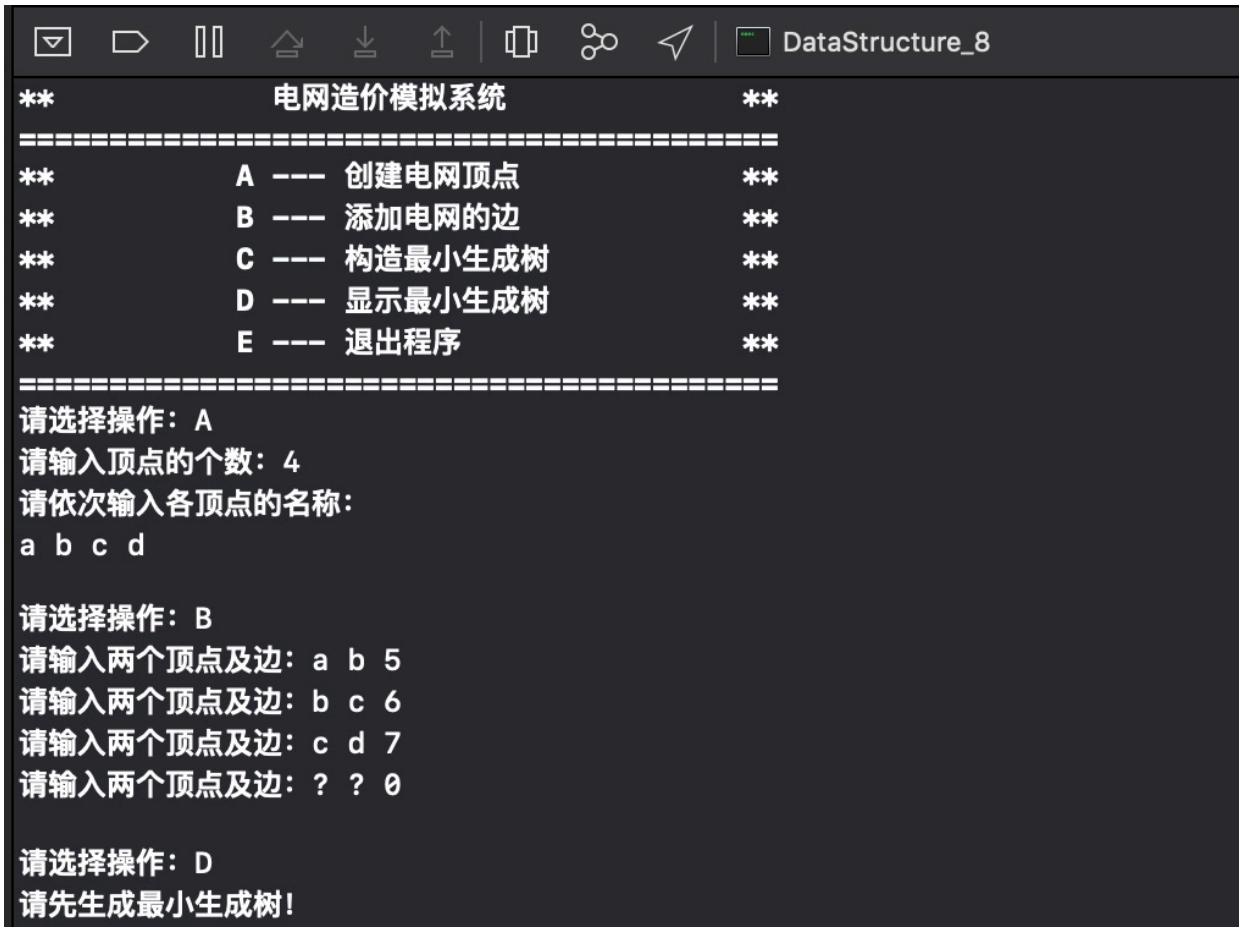
4.2 相关代码

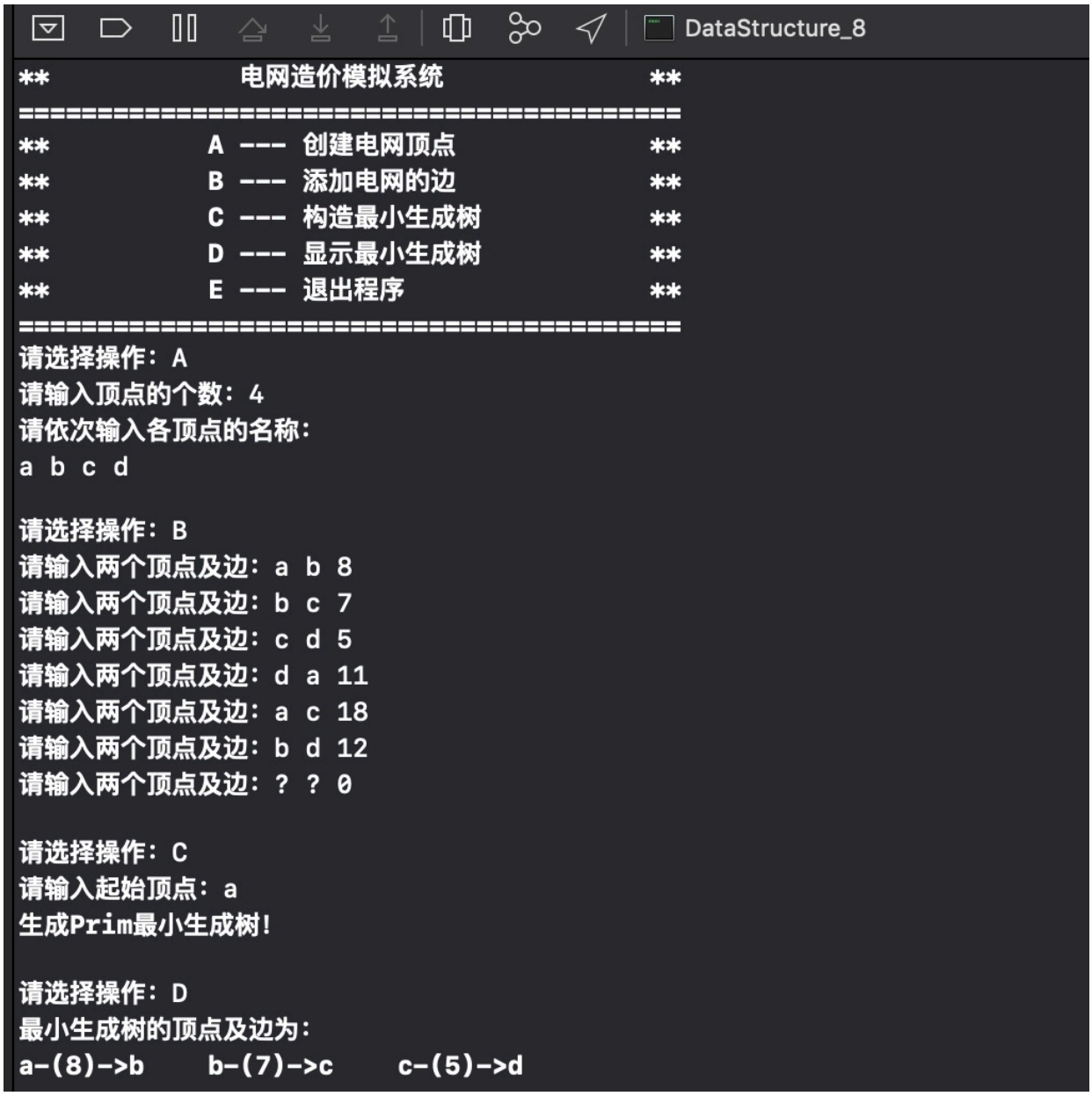
```
                if(!isSystemBuilt)
                {
                    cout << "请先创建电网顶点！" <<endl;
                    break;
                }
                if(!isGene)
                {
```

```
                cout << "请先构造最小生成树！" << endl;
                break;
            }
            else
            {
                cout << "最小生成树的顶点及边为：" << endl;
                output(grid, minTree);
            }
            cout << endl;
```

4.3 运行截图

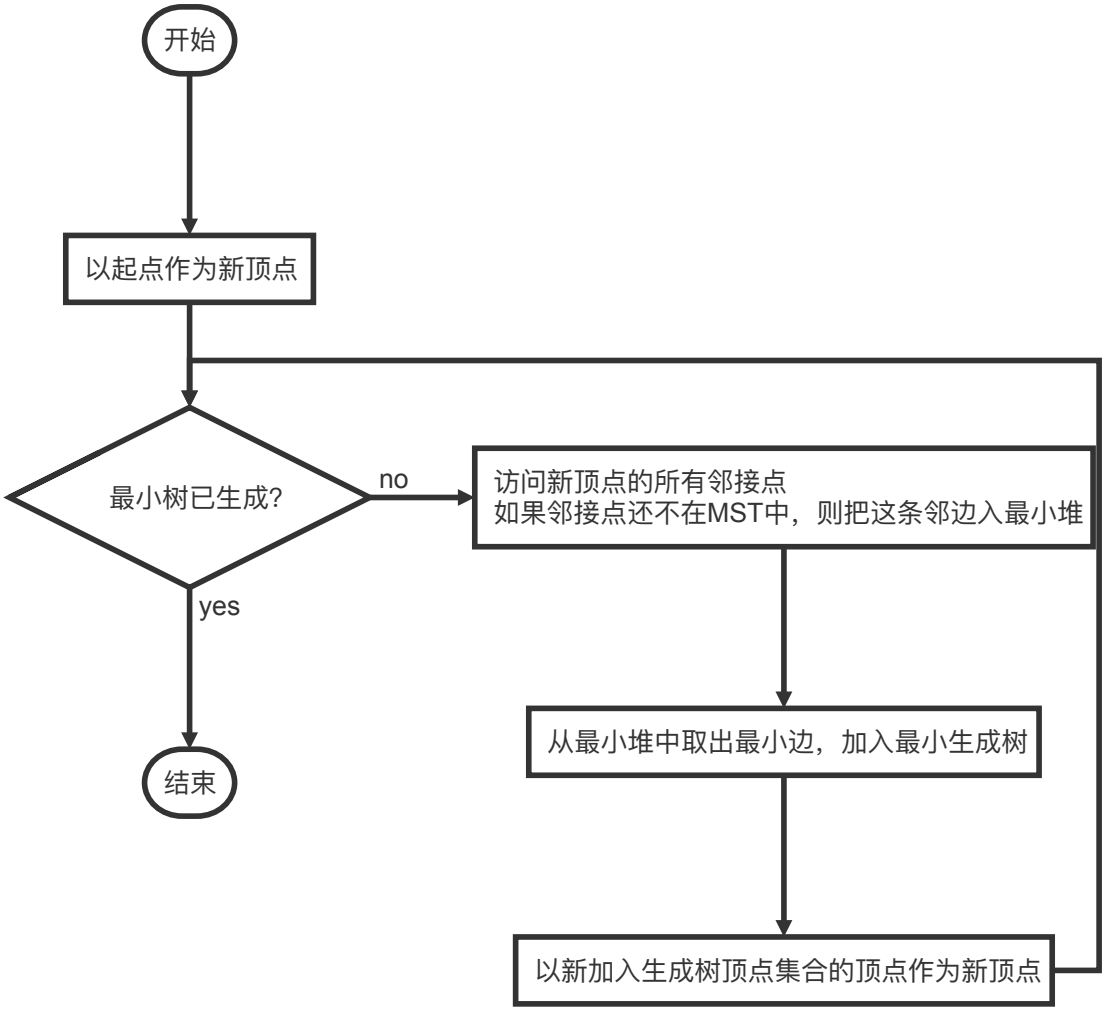






5. Prim算法求最小生成树

5.1 流程图



5.2 相关代码

```
template <typename T, typename E>void Graph<T,E>::Prim(MinSpanTree<T, E>
&MST, int startVertex){
    MSTEdgeNode<T, E> ed; //边结点辅助单元
    int i, v, count;
    int n = NumberOfVertices();   //顶点数
    int m = NumberOfEdges();      //边数
    int u = startVertex;          //起始顶点号
    //int u = 0;
    MinHeap <E, MSTEdgeNode<T, E> > H(m);  //最小堆
    bool *Vmst = new bool[n];   //最小生成树顶点集合
    for (i = 0; i < n; i++) Vmst[i] = false;
    Vmst[u] = true;             //u 加入生成树
    count = 1;
    do{                  //迭代
```

```
            v = getFirstNeighbor(u);
            while (v != -1){      //检测u所有邻接顶点
                if (!Vmst[v]){           //v不在mst中
                    ed.tail = u;  ed.head = v;
                    ed.key = getWeight(u, v);
                    H.Insert(ed);            //(u,v)加入堆
                }      //堆中存所有u在mst中, v不在mst中的边
                v = getNextNeighbor(u, v);
            }
            while (!H.IsEmpty() && count < n)        {
                H.RemoveMin(ed);              //选堆中具最小权的边
                if (!Vmst[ed.head])          {
                    MST.Insert(ed);        //加入最小生成树
                    u = ed.head;
                    Vmst[u] = true;//u加入生成树顶点集合
                    count++;
                    break;
                }
            }
        } while (count < n);
    }
```

四、测试

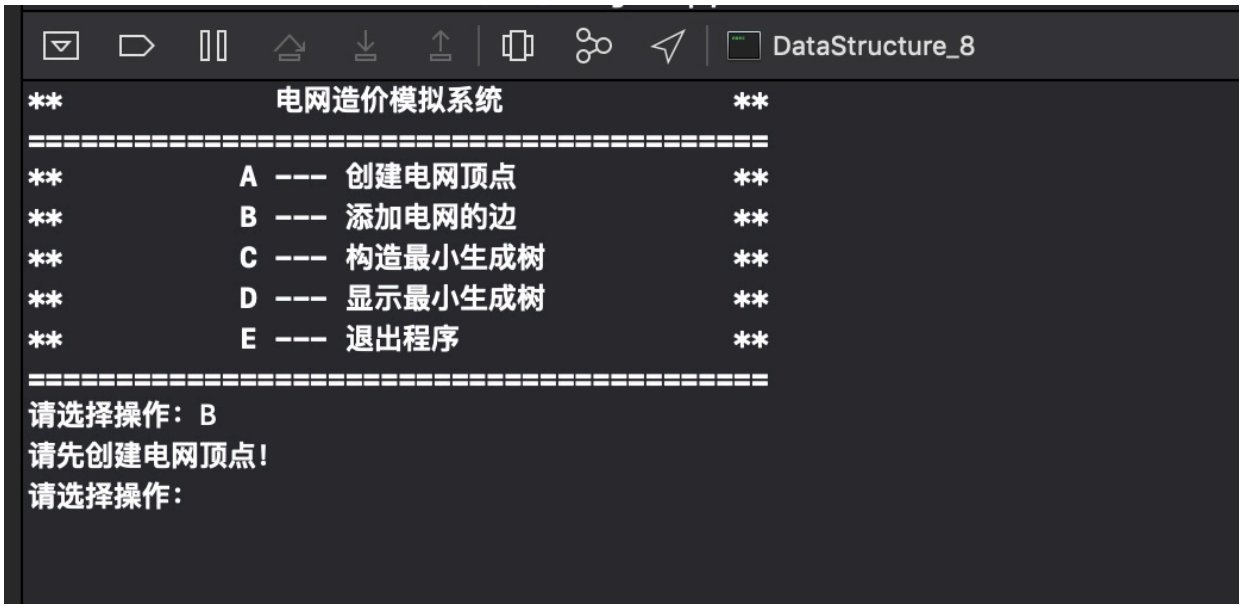
1. 功能测试

一般情况下的功能已经在 “**三、实现**” 中以运行截图的形式给出，经过测试，结果均正常，这里不再重复展示。

2. 出错测试

2.1 未创建电网顶点的情况下选择其他功能

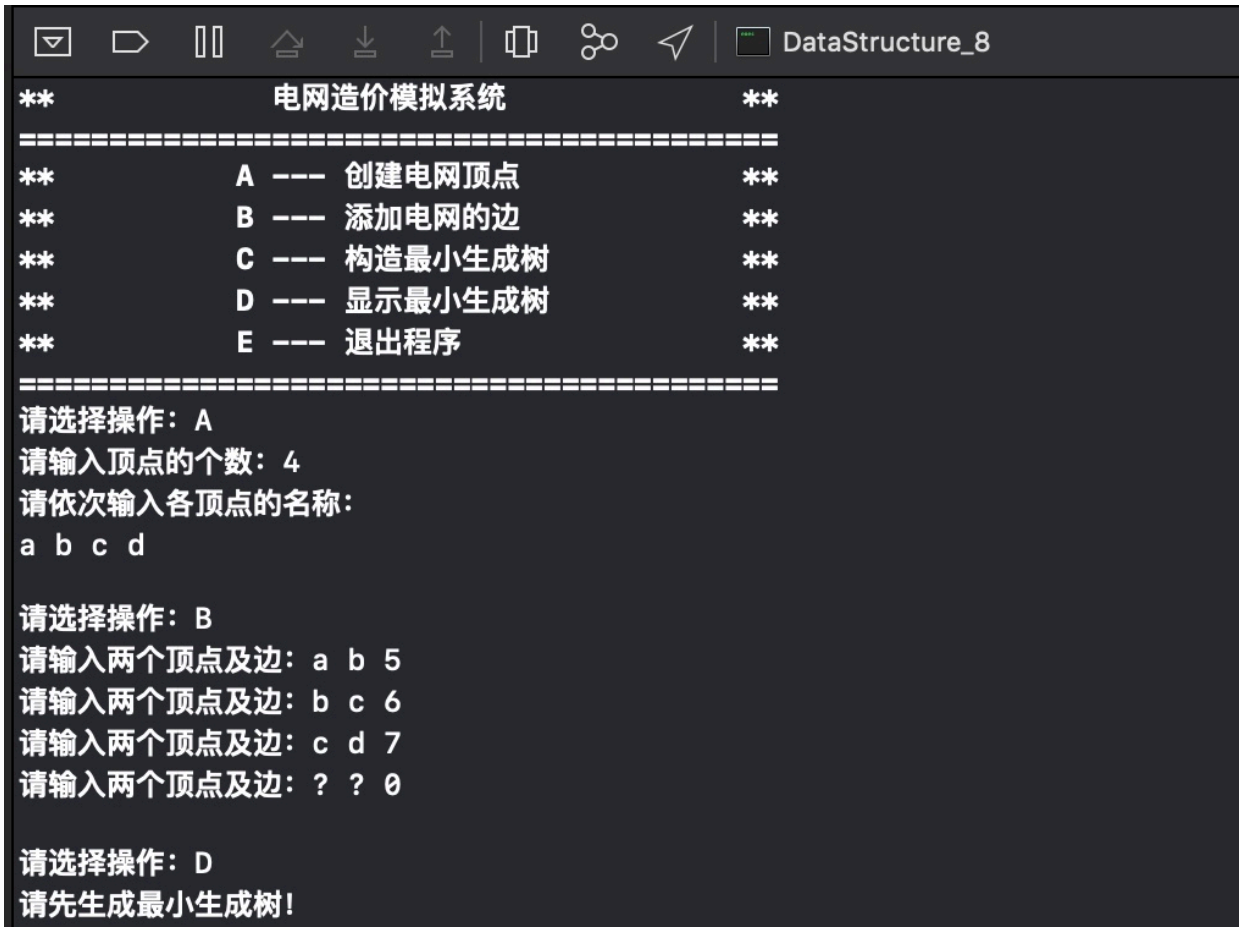
测试结果：



均提示应先创建电网顶点，测试结果正确。

2.2 未构造最小生成树的情况下显示最小生成树

测试结果：



提示应先构造最小生成树，测试结果正确。