# 1 需求

## 1.1 背景分析

电网结构是指电力网内各发电厂、变电所和开关站的布局，以及连接它们的各级电压电力线路的连接方式。电网造价是影响电网设计的重要因素之一。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，现希望构造这个城市n个小区之间的电网，使总工程造价最低。本项目拟开发一个自动生成最佳电网连接策略的程序，用户输入不同小区间的距离，系统输出最低工程造价对应的策略。

## 1.2 功能分析

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。当选择了其中的n-1条组成电网线路时，总的耗费最少。从程序的角度来看，得到总耗费最少电网线路问题即为构建最小生成树问题，可以用克鲁斯卡尔算法算法和普里姆算法构建，构建完成后系统将最小生成数打印出来。

因此，本系统的功能包含：创建电网顶点、添加电网的边、构造克鲁斯卡尔最小生成树、构造普里姆最小生成树、显示最小生成树以及退出系统操作。

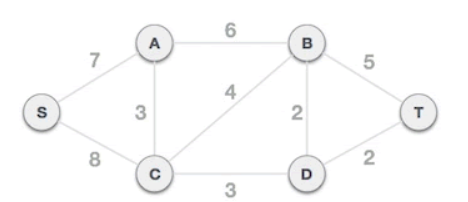
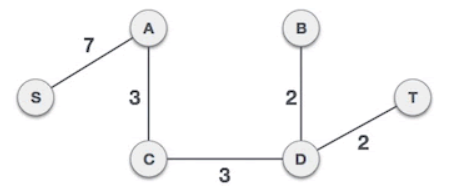


# 2 设计

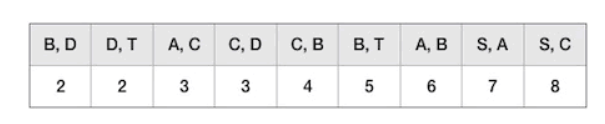
## 2.1 数据结构设计

### 2.1.1 逻辑结构设计

本项目核心功能是输入各小区间的距离，系统构造出最小生成树并打印显示。很显然一个小区可以到达多个小区，不同层次间的元素构成了“一对多”与“多对一”的映射，因此应当采用非线性逻辑结构存储。从整体来看，构造最小生成树的问题即为图的问题，所以可以采用无向图的逻辑结构存储。对于构造最小生成树问题，本项目给出了两种解决办法，一是克鲁斯卡尔算法即选边法，二是普里姆算法即选点法，程序整合了两种算法并给出了二者的结果。

对于克鲁斯卡尔算法，程序需要两个逻辑结构辅助，分别是最小堆与并查集。克鲁斯卡尔算法查找最小生成树的方法是将连通网中所有的边放入最小堆中，按照权值大小做升序排序，从权值最小的边开始选择，只要此边不和已选择的边一起构成环路，就可以选择它组成最小生成树。对于 N 个顶点的连通网，挑选出 N-1 条符合条件的边，这些边组成的生成树就是最小生成树。



而普里姆算法查找最小生成树的过程，采用了贪心算法的思想。对于包含 N 个顶点的连通网，普里姆算法每次从连通网中找出一个权值最小的边，这样的操作重复 N-1 次，由 N-1 条权值最小的边组成的生成树就是最小生成树。其需要两个辅助数组，分别是lowcost数组与nearvex数组，lowcost数组用于存放生成树外节点到达生成树所需要的最小代价，nearvex数组存放生成树外节点距离最近的树内节点，通过不断迭代更新两个数组实现普里姆算法。

### 2.1.2 存储结构设计

计算机存图有两种主要方法，分别是邻接矩阵与邻接表。邻接矩阵通过建立一个行列数为顶点个数的二维数组来存储图的相关信息，邻接表则采用链表的方

式。在一个顶点数为n,边数为e的图中，二者的时间复杂度比较如下：

|  |  |  |
| --- | --- | --- |
|  | 邻接矩阵 | 邻接表 |
| 构造功能 |  |  |
| 查找邻接对象功能 |  |  |
| 判断是否邻接功能 |  |  |
| 占用空间 |  |  |

结合多方面因素考虑，本项目采用邻接表的形式存储图。

## 2.2 类设计

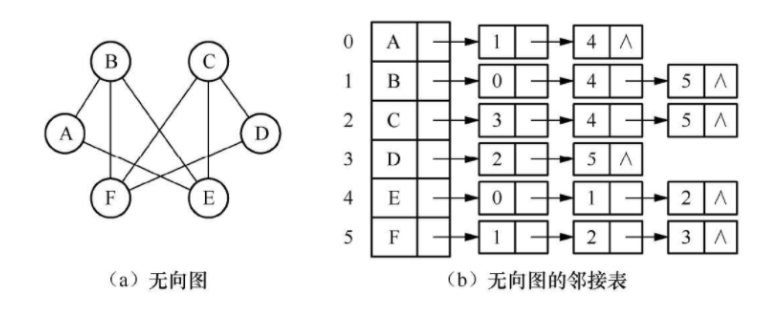
本项目定义的类较多，其主要有通过邻接表形式实现的图Graph类，以及实现Graph类所必须的顶点类Vertex与边类Edge。在克鲁斯卡尔算法中，为了每次选择权值最小的边且不构成回路，还需定义两个辅助类分别是最小堆类minHeap与并查集类UFsets。最后程序定义了一个电网造价模拟系统类，存储项目最为核心的几项功能包含创建电网顶点、添加电网的边、构造克鲁斯卡尔最小生成树、构造普里姆最小生成树、显示最小生成树、退出系统操作等。为提高代码的重用性，本项目所有的类均通过类模板的方式实现。

### 2.2.1 顶点类、边类

本项目采用邻接表形式对图进行存储。邻接表是树与图结构的一般化存储方式，可以看成“带有索引数组的多个数据链表”构成的结构集合。在这样的结构中存储的数据被分成若干类，每一类的数据构成一个链表。每一类还有一个代表元素，称为该类对应链表的“表头”。所有“表头”构成一个表头数组，作为一个可以随机访问的索引，从而可以通过表头数组定位到某一类数据对应的链表。为了实现图类，本项目预先定义了两个辅助类结构，分别是顶点类与边类。

顶点类的protected属性存放顶点表以及指向与该边相邻第一条边的指针，public属性存放构造函数。边类的protected属性存放边的终点在顶点表中对应索引下标、边的权重以及指向下一条边的指针，public属性存放构造函数与析构函数。之后在定义图类时直接调用顶点类与边类的实现即可。





//边的模板类

**template** <**class** NameType, **class** DistType>

**class** Edge {

**protected**:

**int** dest;

DistType cost;

Edge<NameType, DistType>\* link;

**public**:

Edge():dest(0), cost(0), link(NULL){};

Edge(**int** D,DistType C):dest(D),cost(C),link(NULL){};

**friend** Graph<NameType, DistType>;

};

//顶点的模板类

**template** <**class** NameType, **class** DistType>

**class** Vertex {

**protected**:

    NameType data;

    Edge <NameType,DistType> \* adj;

**public**:

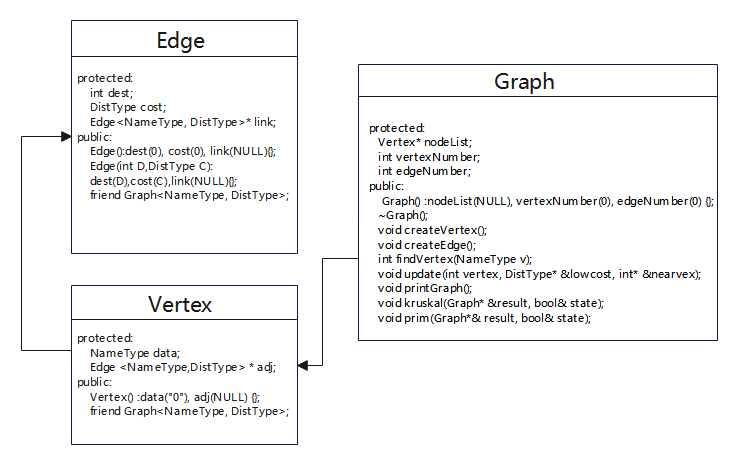
    Vertex() :data("0"), adj(NULL) {};

**friend** Graph<NameType, DistType>;

};

### 2.2.2 图类

图类的protected属性存储图的顶点表，顶点表为指向顶点的指针数组，每个顶点对象连接相应的边，通过链表的形式进行图信息的存储。图类的public属性存放图的各项操作，包括创建顶点表、创建边、查找顶点在vertexList中的位置、更新lowcost与nearvex数组、打印图、构建最小生成树Kruskal、构建最小生成树Prim等。为了提高代码的重用性，项目在定义图类的时候采用类模板的形式，使用的模板分别是顶点表的名称类型NameType和边的权重类型DistType。



//图的模板类

**template** <**class** NameType, **class** DistType>

**class** Graph {

**protected**:

    Vertex<NameType, DistType>\* nodeList;

**int** vertexNumber;

**int** edgeNumber;

**public**:

    Graph() :nodeList(NULL), vertexNumber(0), edgeNumber(0) {};

    ~Graph();

**void** createVertex();

**void** createEdge();

**int** findVertex(NameType v);

**void** update(**int** vertex, DistType\* &lowcost, **int**\* &nearvex);

**void** printGraph();

**void** kruskal(Graph<NameType, DistType>\* &result, **bool**& state);

**void** prim(Graph<NameType, DistType>\*& result, **bool**& state);

### 2.2.3 最小堆的模板类

最小堆，是一种经过排序的[完全二叉树](https://baike.baidu.com/item/%E5%AE%8C%E5%85%A8%E4%BA%8C%E5%8F%89%E6%A0%91/7773232?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E6%9C%80%E5%B0%8F%E5%A0%86/_blank)，其中任一非终端节点的数据值均不大于其左子节点和右子节点的值，可以看作是一种优先级队列的实现，有些应用场景需要从队列中获取最小的或者最大的元素，而且不要求数据全部有序，使用最小堆或者最大堆能很好的解决这类问题。最小堆的元素是按[完全二叉树](https://so.csdn.net/so/search?q=%E5%AE%8C%E5%85%A8%E4%BA%8C%E5%8F%89%E6%A0%91&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Huoon/article/details/_blank)的顺序存储方式存放在一维数组中，其根节点对应数组的第一个元素，后面接着依次从上到下每一层树节点从左往右排列，按照这个顺序将一段[线性](https://so.csdn.net/so/search?q=%E7%BA%BF%E6%80%A7&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Huoon/article/details/_blank)内存结构映射为一个树结构。最小堆的操作主要包括构造函数、析构函数、最小元素出堆、插入元素、自下而上调整最小堆、自上而下调整最小堆等。通过反复调整可以使一颗原本无序的二叉树变成最小堆，每次取根节点即为序列中的最小值。

//最小堆的模板类

**template** <**class** Type>

**class** minHeap {

**protected**:

**int** size;                   //当前容量

**int** maxSize = 100000;       //最大容量

    Type \*heap;                 //存放堆元素的数组

**public**:

    minHeap(**int** n, Type\* elems);        //构造函数

    ~minHeap();                         //析构函数

    Type removeMin();                   //最小元素出堆

**void** insert(**const** Type& x);         //插入元素

**void** siftDown(**int** start);           //自下而上调整最小堆

**void** siftUp(**int** start);             //自上而下调整最小堆

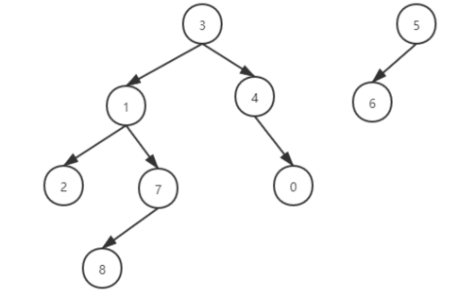
**friend** **class** haffmanTree<Type>;

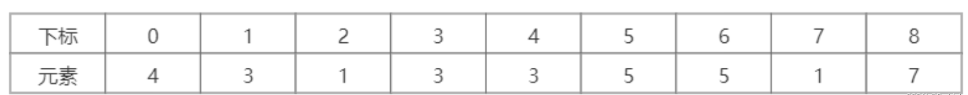
**friend** **class** haffmanTreeNode<Type>;

};

### 2.2.4 并查集类

并查集是一种树型的[数据结构](https://so.csdn.net/so/search?q=%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/the_ZED/article/details/_blank)，用于处理一些不相交集合的合并及查询问题。并查集主要由一个[整型](https://so.csdn.net/so/search?q=%E6%95%B4%E5%9E%8B&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/the_ZED/article/details/_blank)数组parents[ ]和两个函数find()、combine(x,y)构成。[数组](https://so.csdn.net/so/search?q=%E6%95%B0%E7%BB%84&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/the_ZED/article/details/_blank) parents[ ] 记录了每个点的前驱节点，函数 find() 用于查找指定节点x属于哪个集合，函数 combine(x,y)用于合并两个节点x和y。





//并查集类

**class** UFsets {

**protected**:

**int** size;

**int**\* parents;

**public**:

    UFsets(**int** n);

**int** find(**int** x);

**bool** compare(**int** x, **int** y);

**void** combine(**int** root1,**int** root2);

};

### 2.2.5 电网造价模拟系统类

电网造价模拟系统类是程序最外层的类结构，该类定义了一个电网造价模拟系统，其protected属性存放图信息，通过接口与顶点类、边类、图类、最小堆类、并查集类连接。电网造价模拟系统类连接前端cmd界面与内部实现，与用户进行直接交互。用户通过该类输入相应的操作码，程序执行相应的函数，当用户操作不当时，程序给出提示并报错。

//电网造价模拟系统类

**template** <**class** NameType,**class** DistType>

**class** PowerCost {

**protected**:

    Graph<NameType, DistType>\* power;    //图信息

**public**:

    PowerCost();                         //构造函数

**void** operation();                    //循环操作

};

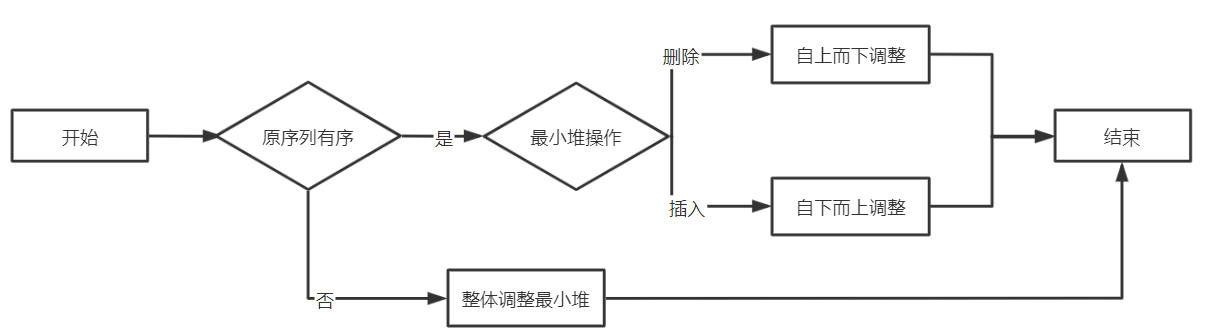
# 3 实现

## 3.1 最小堆整体调整的实现

### 3.1.1 最小堆调整实现思路

最小堆调整可分为自上而下的调整方法和自下而上的调整方法。假若原序列满足最小堆，当插入一个元素时可将该元素放于序列的最末端，然后采取自下而上的调整法；当取出堆内最小元素后可将序列最末端元素放于序列首地址，最小堆大小减一后采用自下而上的调整法。当原序列整体无序，可以从后往前访问最小堆的每个非叶子节点即分支节点，对每个非叶子节点使用自下而上调整法，反复执行后原序列满足最小堆的定义。

### 3.1.2 最小堆调整实现流程图



### 3.1.3 最小堆调整实现代码

//自上而下调整最小堆

**template**<**class** Type>

**void** minHeap<Type>::siftDown(**int** start) {

**int** i = start, j = 2 \* i + 1;

    Type temp = heap[i];

**while** (j < size) {

**if** (j < size - 1 && heap[j]->value > heap[j + 1]->value)

            j++;

**if** (heap[j]->value < temp->value) {

            heap[i] = heap[j];

            i = j;

            j = 2 \* i + 1;

        }

**else**

**break**;

    }

    heap[i] = temp;

}

//自下而上调整最小堆

**template** <**class** Type>

**void** minHeap<Type>::siftUp(**int** start) {

**int** i = start, j = (i - 1) / 2;

    Type temp = heap[i];

**while** (i > 0) {

**if** (heap[j]->value > temp->value) {

            heap[i] = heap[j];

            i = j;

            j = (j - 1) / 2;

        }

**else**

**break**;

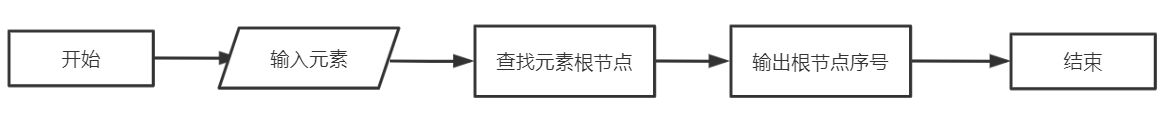
## 3.2 并查集功能的实现

### 3.2.1 并查集实现思路

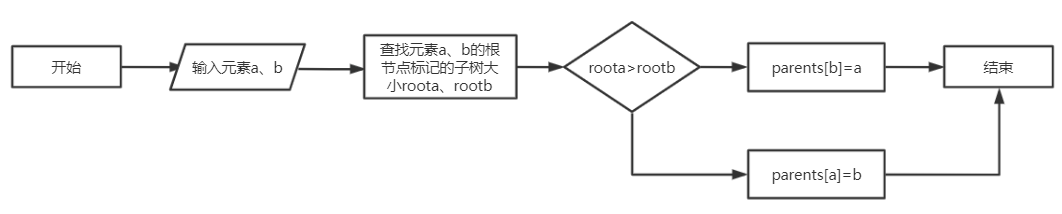
并查集是一种用来管理元素分组情况的数据结构。并查集可以高效地查询两个元素是否在同一个集合、合并两个不同的集合。初始化并查集时，每个元素都是单独的一个集合，之后通过不断的合并形成森林。由于并查集结构的特殊性，查找两个元素是否属于同一个集合就是查找两个元素是否在同一个树上，而合并存在于两个树的元素时，只需将另一颗树的根连到另一个树上，如若想进一步降低时间复杂度的话可以采用路径压缩的思想。

### 3.2.2 并查集实现流程图

查找操作：



合并操作：



### 3.2.3 并查集实现代码

/\*并查集模板类的类外实现\*/

//构造函数

UFsets::UFsets(**int** n) {

    size = n;

    parents = **new** **int**[n];

**for** (**int** i = 0; i < n; i++) {

        parents[i] = -1;

    }

}

//析构函数

UFsets::~UFsets() {

**delete** parents;

}

//比较操作

**bool** UFsets::compare(**int** x, **int** y) {

**if** (parents[x] == -1 && parents[y] == -1)

**return** **false**;

**return** (parents[x] == parents[y]);

}

//搜索操作

**int** UFsets::find(**int** x) {

**while** (parents[x] != -1) {

        x = parents[x];

    }

**return** x;

}

//合并操作

**void** UFsets::combine(**int** root1, **int** root2) {

    parents[root2] = root1;

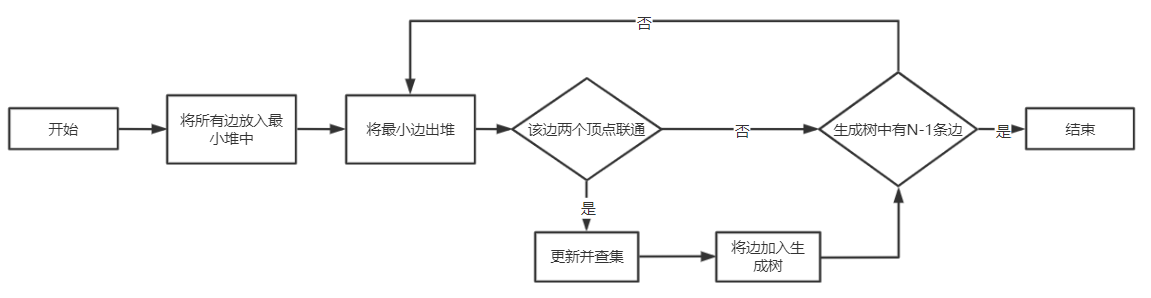
}

## 3.3 生成最小生成树功能的实现—克鲁斯卡尔算法

### 3.3.1 克鲁斯卡尔算法实现思路

克鲁斯卡尔算法从另一途径求网的最小生成树。其基本思想是假设连通网G=（V，E），令最小生成树的初始状态为只有n个顶点而无边的非连通图T=（V，{}），概述图中每个顶点自成一个连通分量。在E中选择代价最小的边，若该边依附的顶点分别在T中不同的连通分量上，则将此边加入到T中；否则，舍去此边而选择下一条代价最小的边。依此类推，直至T中所有顶点构成一个连通分量为止。对于克鲁斯卡尔算法，程序需要两个逻辑结构辅助，分别是最小堆与并查集。初始状态程序将连通图中所有的边放入最小堆中，按照权值大小做升序排序，之后每次从权值最小的边开始选择，只要此边和已选择的边并查集搜索结果不同，说明二者不连通，可以选择它组成最小生成树。

### 3.3.2 克鲁斯卡尔算法实现流程图



### 3.3.3 克鲁斯卡尔算法实现代码

//构建最小生成树Kruskal

**template** <**class** NameType,**class** DistType>

**void** Graph<NameType, DistType>::kruskal(Graph<NameType, DistType>\*& result, **bool**& state) {

    //最小生成树边结构体

**struct** MSTEdge {

**int** value = 0;

        Vertex<NameType, DistType> head;

        Edge<NameType, DistType>\* tail=**new** Edge<NameType, DistType>;

    };

    //最小堆与并查集初始化

    minHeap<MSTEdge\*> heap;

    UFsets sets(vertexNumber);

    result->vertexNumber = vertexNumber;

    result->edgeNumber = edgeNumber;

    result->nodeList = **new** Vertex<NameType,DistType>[vertexNumber];

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

        result->nodeList[i] = nodeList[i];

        result->nodeList[i].adj = NULL;

        Edge<NameType, DistType>\* p = nodeList[i].adj;

**while** (p != NULL) {

            MSTEdge\* m = **new** MSTEdge;

            m->value = p->cost;

            m->head = nodeList[i];

            m->tail = p;

            heap.insert(m);

            p = p->link;

        }

    }

    //Kruskal算法实现

**int** sum1 = 0;

**int** sum2 = 0;

**while** (sum1 < vertexNumber - 1) {

        sum2 += 1;

        MSTEdge\* edge = heap.removeMin();

**int** flag1 = findVertex(edge->head.data);

**int** flag2 = edge->tail->dest;

**if** (sets.compare(flag1,flag2))

**continue**;

**else** {

            Edge<NameType, DistType>\* p = edge->tail;

            p->link = result->nodeList[flag1].adj;

            result->nodeList[flag1].adj = p;

            sets.combine(flag1, flag2);

            sum1 += 1;

        }

**if** (sum1 < vertexNumber - 1 && sum2 == edgeNumber) {

            cout << "非连通图不存在最小生成树！" << endl;

            state = **false**;

**return**;

        }

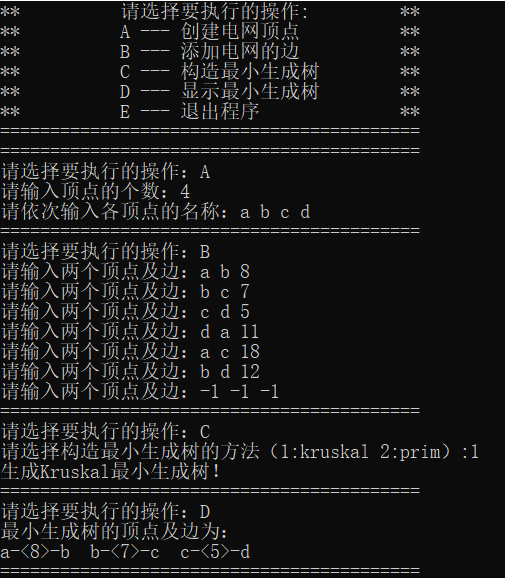
    }

    cout << "生成Kruskal最小生成树！" << endl;

    state = **true**;

}

### 3.3.4 克鲁斯卡尔算法实现样例

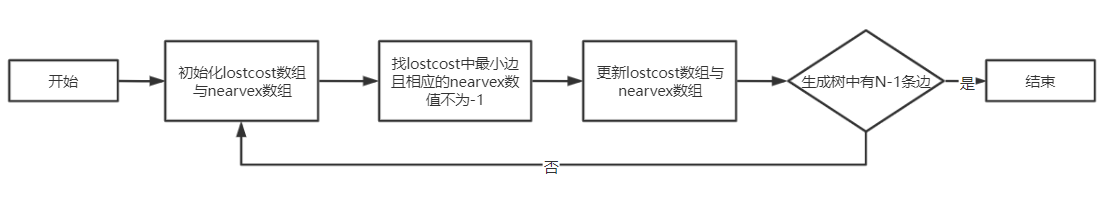


## 3.4 生成最小生成树功能的实现—普里姆算法

### 3.4.1 普里姆算法实现思路

普里姆算法在找最小生成树时，将顶点分为两类，一类是在查找的过程中已经包含在树中的（假设为 A 类），剩下的是另一类（假设为 B 类）。对于给定的连通图，起始状态全部顶点都归为 B 类。在找最小生成树时，选定任意一个顶点作为起始点，并将之从 B 类移至 A 类；然后找出 B 类中到 A 类中的顶点之间权值最小的顶点，将之从 B 类移至 A 类，如此重复，直到 B 类中没有顶点为止，所走过的顶点和边就是该连通图的最小生成树。普里姆实现需要两个辅助数组，分别是lowcost数组与nearvex数组，lowcost数组用于存放生成树外节点到达生成树所需要的最小代价，nearvex数组存放生成树外节点距离最近的树内节点，通过不断迭代更新两个数组实现普里姆算法。

### 3.4.2 普里姆算法实现流程图



### 3.4.3 普里姆算法实现代码

//构建最小生成树Prim

**template** <**class** NameType, **class** DistType>

**void** Graph<NameType, DistType>::prim(Graph<NameType, DistType>\*& result, **bool**& state) {

    NameType start;

**int** temp = 0;

**int** dest = 0;

    //输入起始顶点

    cout << "请输入起始顶点：";

**while** (1) {

        cin >> start;

**if** (cin.fail()) {

            cout << "输入错误请重新输入：";

            cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

**continue**;

        }

**else** **if** ((temp = findVertex(start)) == -1) {

            cout << "该节点不存在，请重新输入：";

**continue**;

        }

**else**

**break**;

    }

    //初始化lowcost数组与nearvex数组

    DistType\* lowcost = **new** DistType[vertexNumber];

**int**\* nearvex = **new** **int**[vertexNumber];

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

        lowcost[i] = 214783647;

        nearvex[i] = 0;

    }

    //prim算法实现

    result->vertexNumber = vertexNumber;

    result->edgeNumber = edgeNumber;

    result->nodeList = **new** Vertex<NameType, DistType>[vertexNumber];

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

        result->nodeList[i] = nodeList[i];

        result->nodeList[i].adj = NULL;

    }

    lowcost[temp] = 0;

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

        update(temp, lowcost, nearvex);

**int** min = 2147483647;

**for** (**int** j = 0; j < vertexNumber; j++) {

**if** (lowcost[j] < min && nearvex[j] != -1) {

                min = lowcost[j];

                dest = j;

            }

        }

        Edge<NameType, DistType>\* minEdge = **new** Edge<NameType, DistType>(dest, lowcost[dest]);

**if** (i != vertexNumber - 1) {

            minEdge->link = result->nodeList[nearvex[dest]].adj;

            result->nodeList[nearvex[dest]].adj = minEdge;

        }

        temp = dest;

    }

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

**if** (nearvex[i] != -1) {

            cout << "非连通图不存在最小生成树！" << endl;

            state = **false**;

**return**;

        }

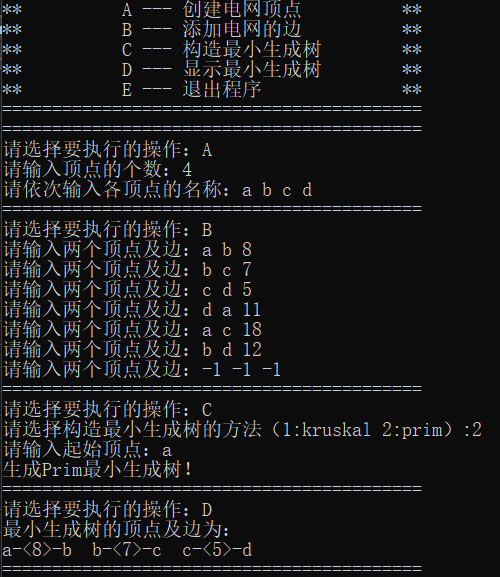
    }

    cout << "生成Prim最小生成树！" << endl;

    state = **true**;

}

### 3.4.4 普里姆算法实现样例



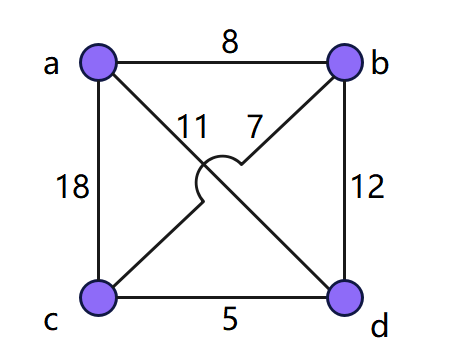
# 4 测试

## 4.1 常规测试

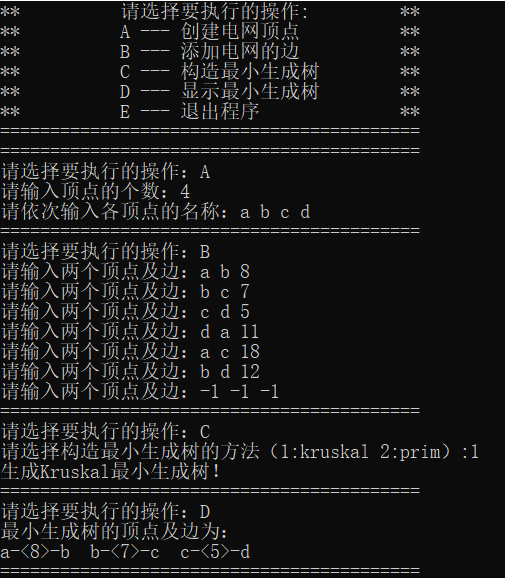
测试用例：

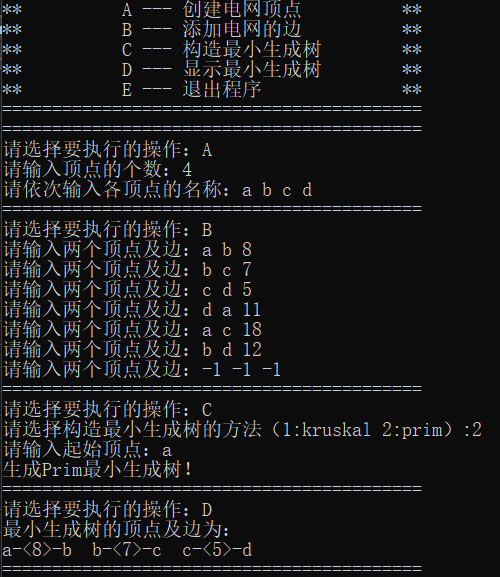
顶点集合：a b c d

边集合：



实际输出：



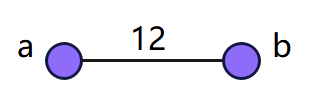


## 4.2 边界测试

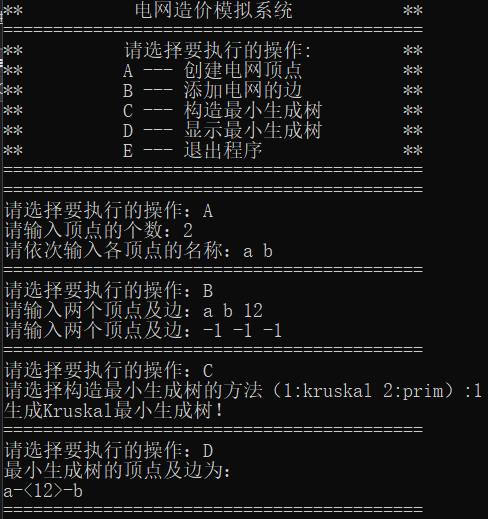
测试用例：

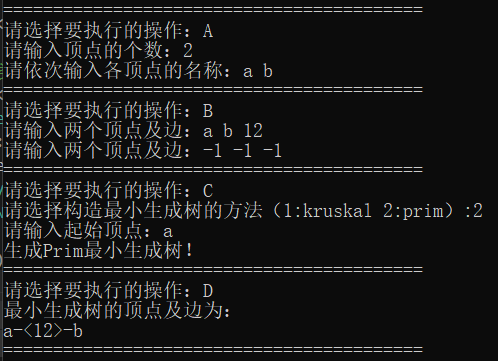
顶点集合：a b

边集合：



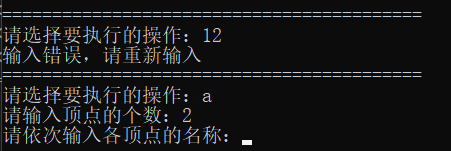
实际输出：



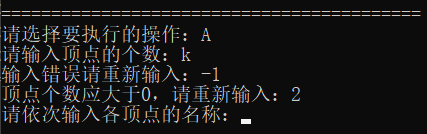


## 4.3 错误测试

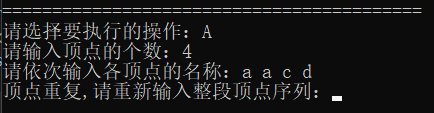
### 4.3.1 输入操作码错误



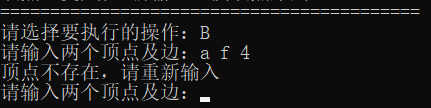
### 4.3.2 输入顶点数错误



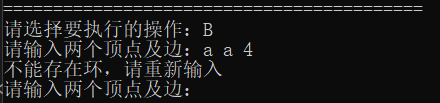
### 4.3.3 输入顶点重复



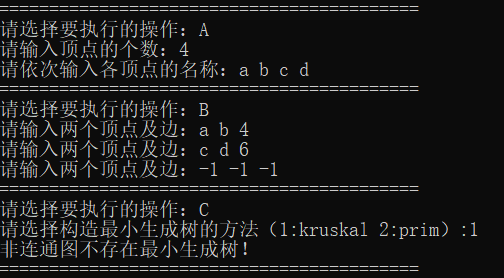
### 4.3.4 输入边不存在



### 4.3.5 输入边有环



### 4.3.6 不存在生成树



### 4.3.7 普里姆算法起始顶点不存在

