数据结构课程设计项目说明文档

——电网建设造价模拟系统

作 者 姓 名： 马威

学 号： 2151294

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

****

目 录

[1 项目分析 1](#_Toc495668153)

[1.1 项目背景 1](#_Toc495668154)

[1.2 项目要求 1](#_Toc495668154)

[1.3 项目需求分析 2](#_Toc495668155)

[2 项目设计 3](#_Toc495668156)

[2.1 数据结构设计 3](#_Toc495668157)

[2.2 类设计 3](#_Toc495668158)

2.2.1 边结点类（Edge） 3

2.2.2 顶点类（Vertex） 4

2.2.3 图的邻接表类（GraphList） 4

2.2.4 最小堆类（MinHeap） 6

2.2.5 并查集类（UFSets） 8

2.3 算法设计 9

2.3.1 算法思路 9

2.3.2 性能评估 10

2.3.3 Kruskal算法 10

·流程图表示 10

·代码实现 10

2.3.4 Prim算法 12

·流程图表示 12

·代码实现 12

2.3.5 项目主体部分 14

·流程图表示 14

·代码实现 14

[3 项目测试 1](#_Toc495668161)7

[3.1 插入顶点测试 1](#_Toc495668162)7

[3.2 插入边测试 1](#_Toc495668166)7

[3.3 Kruskal & 展示 1](#_Toc495668170)8

[3.4 清除测试 1](#_Toc495668174)9

[3.5 Prim & 展示 1](#_Toc495668178)9

[3.6 删除顶点测试 2](#_Toc495668182)0

[3.7 删除边测试 21](#_Toc495668182)

3.8 未清除就生成 21

3.9 未生成就展示 22

3.10 图不连通的情况 22

3.11 图为平凡图的情况 23

3.12 程序退出测试 24

**1.项目分析**

1.1 项目背景

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。

1.2 项目要求

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。

项目示例：



1.3 项目需求分析

对于电网建设造价模拟系统，需考虑以下需求：

**·正确性**

程序应当能够按照要求准确求解出最低造价的建设方案，并能将其没有遗漏地显示出来

**·健壮性**

程序应当对相关信息，如城市代号，权值等的错误输入进行处理，也要能对特殊的分布情况（如仅有一座城市或没有可能的输电线路等）进行合适的处理。

**·高效性**

问题的求解有特殊要求，需要用特定的求法解决，这就要求要在尽可能短的时间内处理大量的数据。

**·交互性**

程序要以系统的形式出现，涉及到菜单选择、界面切换等功能，需要给用户提供足够的提示与说明，使用方式要友好。

**2.项目设计**

2.1 数据结构设计

由项目分析可以得出，本项目中有两类对象：城市与输电线路，每根输电线路连接着两座城市，且自身带有造价这一属性，关系较为复杂，需要合适的数据结构对其进行描述。而数学上，图是一种带有若干个点和边的形状，用一条边连接两个点。其通常用来描述多个事物之间的关系，根据需要，还可以对边添加方向、权值等属性。根据分析，城市电网中输电线路无特定的方向，却有造价的属性，故符合带权无向图的性质。

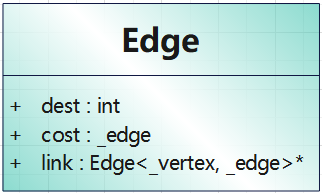
因此，本项目需实现一个图（Graph）类的实现，要能准确地描述题目中各事物间的关系。同时，由于电网需要连通，故最低造价方案也就转变为对图求最小生成树的问题。考虑到程序的高效性，图的存储需选择合适的结构，以降低算法的时间复杂度。

算法实现上，Kruskal算法还可以使用最小堆（MinHeap）和并查集（UFSets）。

2.2 类设计

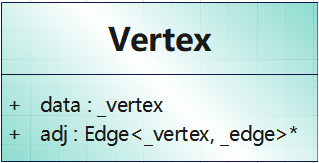
**2.2.1 边结点类（Edge）**

边结点类（Edge）类似一种链表的结点，以其为单位构成的链表构成了某一个顶点的出边（邻接）链表。其存储了某一条边另一顶点的位置和指向下一边结点的指针，其UML图如下：



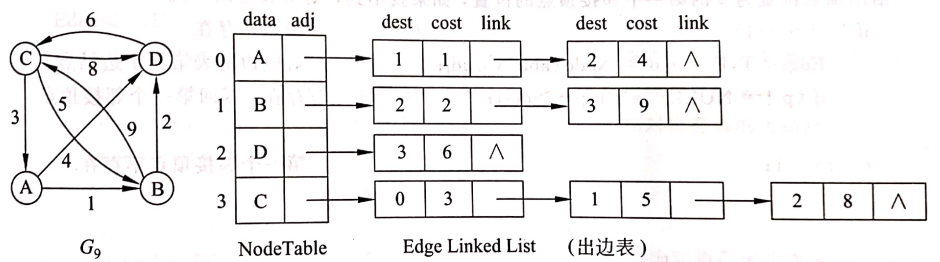
**2.2.2 顶点类（Vertex）**

顶点类（Vertex）类似一种链表的头结点，存储了一个顶点的信息和指向出边（邻接）链表的指针，其UML图如下：



**2.2.3 图的邻接表类（GraphList）**

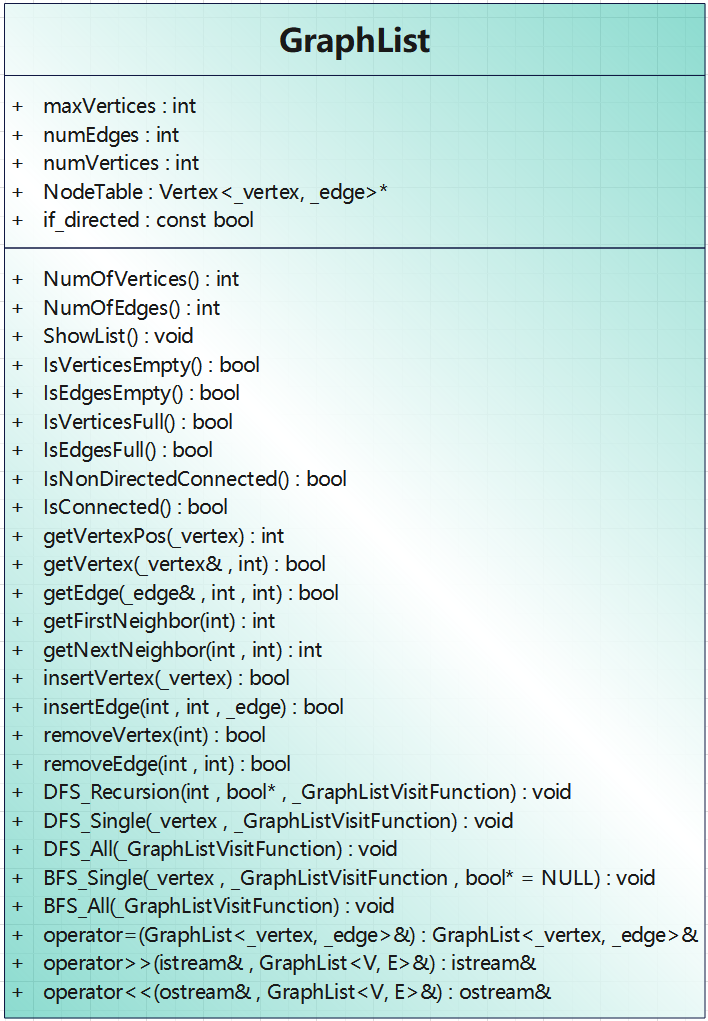
图的邻接表类（GraphList）是通过类似于一个链表的数组进行实现，从而对图进行描述（如下图）。其有一个顶点表NodeList，为一个数组，每个元素（Vertex）存储了一个顶点的信息和指向出边（邻接）链表的指针，类似于链表的头结点。而每一个链表中的其它结点（Edge）都为边结点，存储了边的权值，以及除头结点表示的顶点外这条边另一个顶点在NodeList中的位置。



空间上，这样的存储方式非常节省空间，只有两个节点之间有边相连才进行边结点的存储，在图的边比较稀疏时尤其节省空间开销；时间上，尽管对于插入边等一些基本操作复杂度会比较高，但对于求解最小生成树等问题，用邻接表来表示还是十分理想的。

由于图有有向图和无向图两种，且使用都较为广泛，为了代码的实用性，本项目封装的邻接表类有一个成员if\_directed，指示图是否为有向图，此成员有const限定，在构造函数时需手动指定，此后的插入、删除等操作均根据if\_directed的值进行调整，使同样的代码块兼容有向图和无向图的操作。

其UML图如下：



其中，主要函数如下：

//取堆顶元素，即最小元素

bool GetMin(\_class& x)const;

//返回当前顶点数

int NumOfVertices()const;

//返回当前边数

int NumOfEdges()const;

//展示邻接表

void ShowList()const;

//判断图的顶点数是否为零

bool IsVerticesEmpty()const;

//判断图的边数是否为零

bool IsEdgesEmpty()const;

//判断图的顶点数是否已满

bool IsVerticesFull()const;

//判断图的边数是否已满

bool IsEdgesFull()const;

//判断图是否连通

bool IsConnected()const;

//返回顶点vertex在图中的位置

int getVertexPos(\_vertex vertex)const;

//取顶点pos，pos不合理返回false

bool getVertex(\_vertex& vertex, const int pos)const;

//取边<v1,v2>或(v1,v2)上的权值

bool getEdge(\_edge& edge, const int v1, const int v2)const;

//取顶点v的第一个邻接顶点

int getFirstNeighbor(const int v)const;

//取v邻接顶点w的下一邻接顶点

int getNextNeighbor(const int v, const int w)const;

//插入顶点vertex

bool insertVertex(const \_vertex vertex);

//插入边<v1,v2>，权值为cost

bool insertEdge(const int v1, const int v2, \_edge cost);

//删去顶点v和所有与它相关联的边

bool removeVertex(const int v);

//在图中删去边(v1,v2)或<v1,v2>

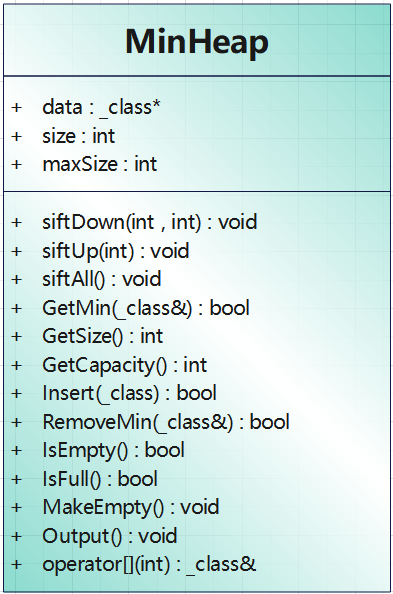
    bool removeEdge(const int v1, const int v2);

**2.2.4 最小堆类（MinHeap）**

最小堆（MinHeap）是一种优先级队列，其逻辑上为二叉树结构，每个结点的值都比其子女小，根结点就是最小值。按照这样的顺序排列，使得最小堆的插入、取出最小值等操作的时间复杂度仅为O(log2n)，性能良好。

每次进行插入或删除操作时，为了保持最小堆的性质，需要频繁地对元素进行访问、比较和交换，而并不关心每个元素的具体位置。因此，虽然最小堆逻辑上是树形结构，但本质上通过数组实现，通过层次往下给出元素的下标，标号为n的左子女（若存在）标号为2n+1，右子女（若存在）标号为2n+2，由此可知，每次进行元素的比较、交换时只需通过下标即可访问元素，非常方便。

其UML图如下：



其中，主要函数如下：

//取堆顶元素，即最小元素

bool GetMin(\_class& x)const;

//取堆的当前大小

int GetSize()const;

//取堆的最大容量

int GetCapacity()const;

//将x插入最小堆中

bool Insert(const \_class x);

//删除堆顶上的最小元素

bool RemoveMin(\_class& x);

//判断堆是否为空

bool IsEmpty()const;

//判断堆是否已满

bool IsFull()const;

//置空堆

void MakeEmpty();

//按数组内存放顺序输出堆内的元素

void Output()const;

//重载函数：下标访问

\_class& operator[](const int pos)const;

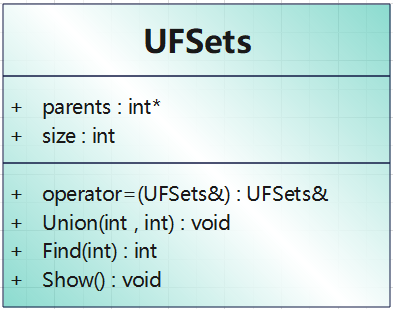
**2.2.5 并查集类（UFSets）**

对于一系列的元素，有时会根据需要分属在不同的集合中，人们有时会关心某一个特定的元素属于哪个集合，而并查集就是用来描述这样的关系的。

逻辑上，所有元素都在某一棵树上，同一个集合中的元素在同一棵树上。树的形式是一个父指针数组，设数组下标为n，元素为x。若x>0，则n号元素的父结点是x；若x<0，则n号元素是其所在集合的根结点，绝对值等于集合元素个数。

并查集支持集合合并、查找等操作。合并时只需把一个集合的根结点作为另一根节点的子女即可，而查找时只需找到元素所在的根结点即可，比较方便。

其UML图如下：



其中，主要函数如下：

//重载函数：赋值

UFSets& operator=(const UFSets& UFSet);

//两个子集合合并

void Union(const int Root1, const int Root2);

//搜寻集合x的根

int Find(const int x);

//显示并查集中的元素状况

void Show()const;

2.3 算法设计

**2.3.1 算法思路**

电网建造方案有两个特点，一是需要能连通所有城市，二是造价需要最低。这两点正好符合图的最小生成树的特点，而求解图的最小生成树算法一般有两种：Kruskal算法和Prim算法。

Kruskal算法又称选边法，每次从边集合中取权值最小的，若该边连接的两个顶点属于不同的连通分量，即加上该边的图不会产生回路，则将这条边加入最小生成树中，否则跳过该边，直到所有的边都被检查过为止，此时最小生成树就得到了。其中，每次从边集合中取权值最小的可以通过使用最小堆（MinHeap）进行优化，以降低时间复杂度；而判断某条边的两个顶点是否在同一连通分量上可以通过使用并查集（UFSets）查询两个顶点是否在一个集合中而实现，也可降低时间复杂度。

Prim算法又称选点法，需要有一个点作为起始顶点，将该点放入集合V中，表示最小生成树包括的结点，除此之外还有一个集合V’，表示除最小生成树外的剩余结点，同时将两个端点分别属于V和V’的边叫做桥。每次选出权值最小的桥，将其加入最小生成树并把属于V’的那个端点加入V，循环直至V’为空则最小生成树就得到了。实现上可以用一个bool数组标志每个顶点是否都处于V中，u为当前顶点（初始为起始顶点），然后按照以下步骤求解：

（1）检查u所有的邻接顶点v，若v∉V，则将连接u、v的边加入最小堆

（2）不断从最小堆取出权值最短的边，直到取出一条边的另一个顶点w不属于V为止，将这条边加入最小生成树中，将w加入V中。u的值变为w

重复（1）（2）直至最小生成树有n-1条边为止，最小生成树得到。

显然，两种算法的基础是图为连通图，若图不为连通图，最小生成树也就不存在，程序会给出相应提示，并且不做任何操作。

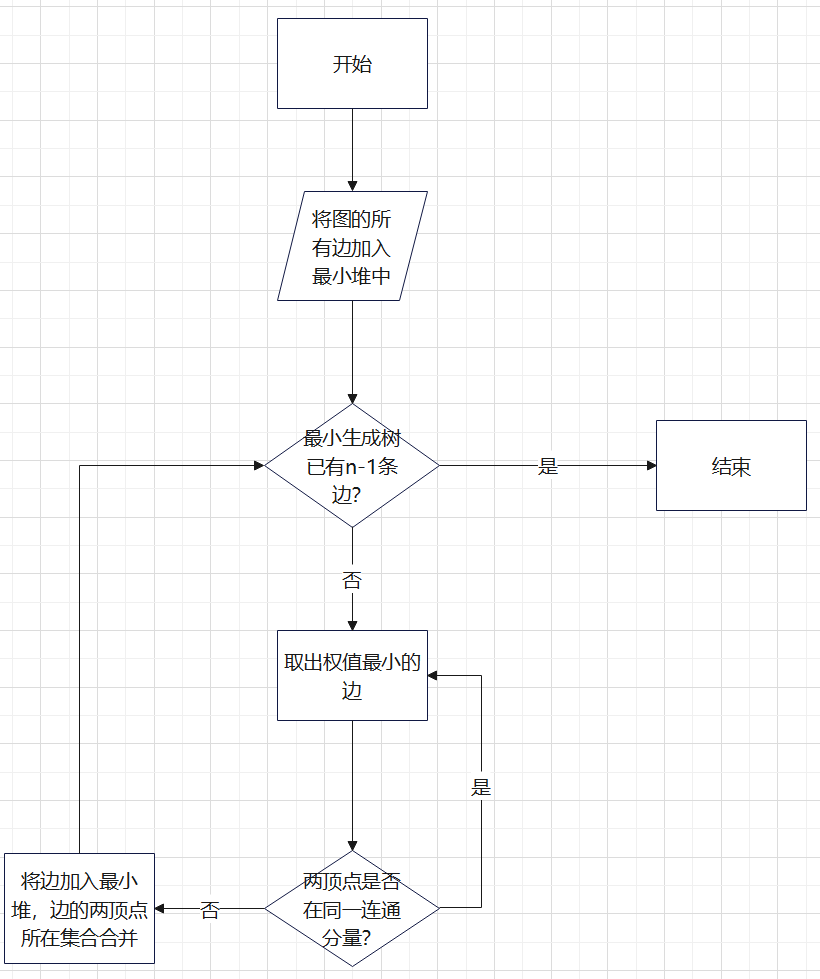
**2.3.2 性能评估**

采用Kruskal算法，由于用邻接表存储，建立最小堆遍历了邻接表，需要O(n+e)和O(elog**2** e)。构造最小生成树过程中，需要O(e)次出堆操作、2e次并查集查询操作、n-1次并查集合并操作，分别需要O(elog**2** e)、O(elog**2** n)、O(n)。又图为连通图，e>=n-1，综上，算法总体时间复杂度为O(n+elog**2** e)。

采用Prim算法，需要循环O(n)趟，每趟平均将2e/n条边插入堆，共e条边从堆中删除，而每次插入、删除操作需要O(log**2** e)。综上，算法总体时间复杂度为O(elog**2** e)。

**2.3.3 Kruskal算法**

·流程图表示

****

·代码实现

template<class \_vertex, class \_edge>

void Kruskal(const GraphMatrix<\_vertex, \_edge>& graph, MinSpanTree<\_vertex, \_edge>& tree)

{

    if (!graph.IsConnected()) {

        cout << "图不连通！" << endl;

        return;

    }

    const int ver\_num = graph.NumOfVertices();  /\*取点数\*/

    const int edge\_num = graph.NumOfEdges();    /\*取边数\*/

    MSTEdgeNode<\_vertex, \_edge> edge;

    MinHeap<MSTEdgeNode<\_vertex, \_edge> > heap(edge\_num);  /\*边的最小堆\*/

    UFSets set(ver\_num);  /\*并查集\*/

    for (int i = 0; i < ver\_num; i++) {

        for (int j = i + 1; j < ver\_num; j++) {  /\*无向图，从i+1开始\*/

            \_edge ed;

            graph.getEdge(ed, i, j);  /\*逐个扫描，取边\*/

            if (ed != maxWeight) {

                graph.getVertex(edge.tail, i);

                graph.getVertex(edge.head, j);

                graph.getEdge(edge.key, i, j);

                heap.Insert(edge);  /\*插入最小堆\*/

            }

        }

    }

    int count = 1;

    while (count < ver\_num) {

        heap.RemoveMin(edge);  /\*取出最短边\*/

        const int i = graph.getVertexPos(edge.tail);

        const int j = graph.getVertexPos(edge.head);

        const int k = set.Find(i);

        const int l = set.Find(j);

        if (k != l) {  /\*若两个顶点不在同一连通分量，将该边放入最小生成树\*/

            set.Union(k, l);

            tree.Insert(edge);

            count++;

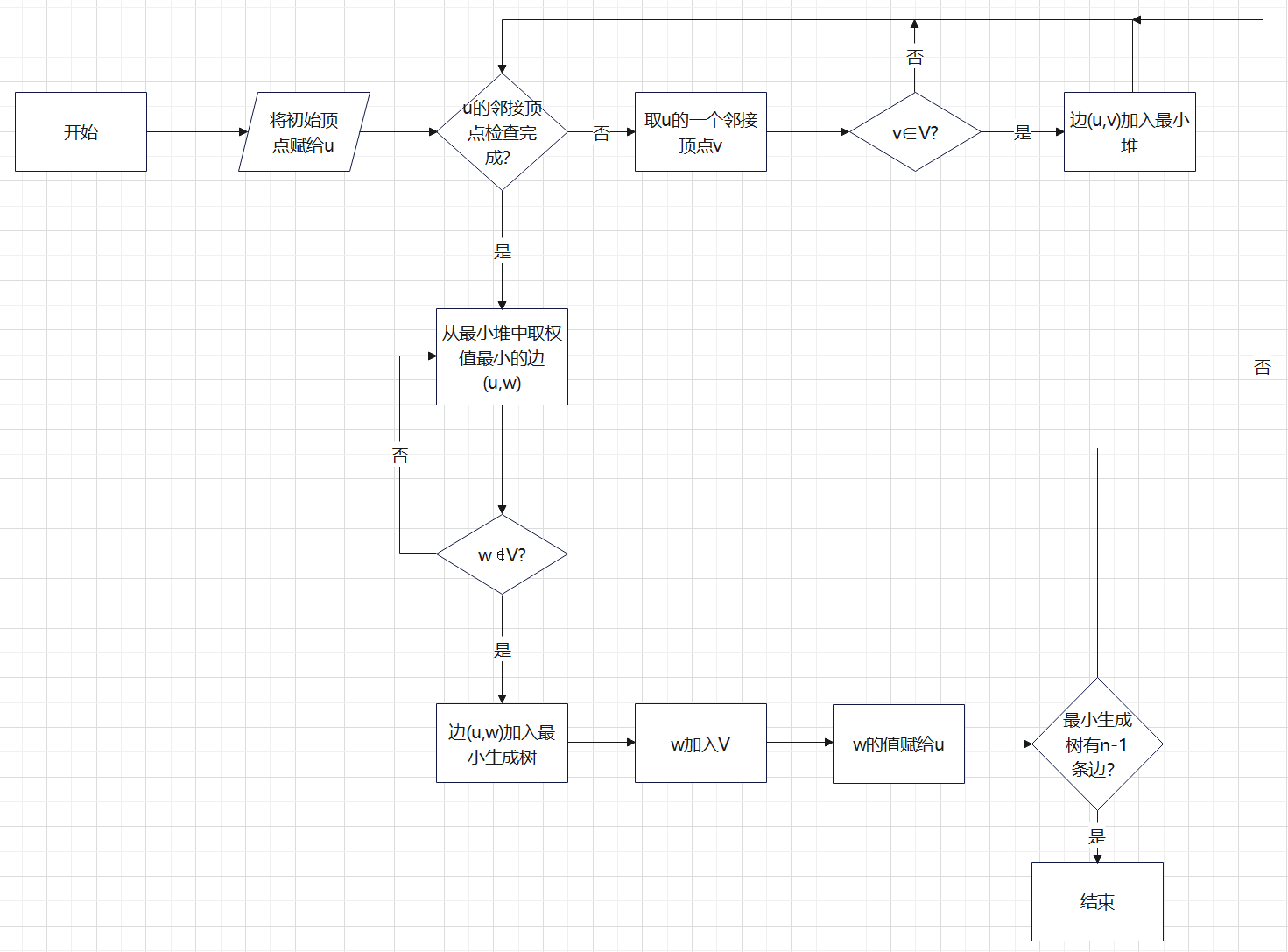
        }

    }

}

**2.3.4 Prim算法**

·流程图表示



·代码实现

template<class \_vertex, class \_edge>

void Prim(const GraphMatrix<\_vertex, \_edge>& graph, const \_vertex u0, MinSpanTree<\_vertex, \_edge>& tree)

{

    if (!graph.IsConnected()) {

        cout << "图不连通！" << endl;

        return;

    }

    MSTEdgeNode<\_vertex, \_edge> edge;

    const int ver\_num = graph.NumOfVertices();  /\*取点数\*/

    const int edge\_num = graph.NumOfEdges();    /\*取边数\*/

    int u = graph.getVertexPos(u0);  /\*取起始顶点编号\*/

    MinHeap<MSTEdgeNode<\_vertex, \_edge> > heap(edge\_num);  /\*边的最小堆\*/

    /\*标志各点是否已被连接起来的数组，也可看做是集合，false为不在集合中，true为在集合中\*/

    bool\* Vmset = new bool[ver\_num];

    for (int i = 0; i < ver\_num; i++)

        Vmset[i] = false;

    Vmset[u] = true;  /\*将起始顶点放入集合中\*/

    int count = 1;

    while (count < ver\_num) {

        int v = graph.getFirstNeighbor(u);  /\*取第一个邻接顶点\*/

        while (v != -1) {

            if (!Vmset[v]) {  /\*若该邻接顶点不在集合中，将连接当前顶点和该邻接顶点的边放入最小堆中\*/

                graph.getVertex(edge.tail, u);

                graph.getVertex(edge.head, v);

                graph.getEdge(edge.key, u, v);

                heap.Insert(edge);

            }

            v = graph.getNextNeighbor(u, v);  /\*取下一邻接顶点，重复操作\*/

        }

        while (!heap.IsEmpty() && count < ver\_num) {

            heap.RemoveMin(edge);  /\*取出最短边\*/

            if (!Vmset[graph.getVertexPos(edge.head)]) {  /\*若该边的另一头不在集合中，则将该边放入最小生成树，同时将另一头的顶点放入集合中\*/

                tree.Insert(edge);

                u = graph.getVertexPos(edge.head);

                Vmset[u] = true;

                count++;

                break;

            }

        }

    }

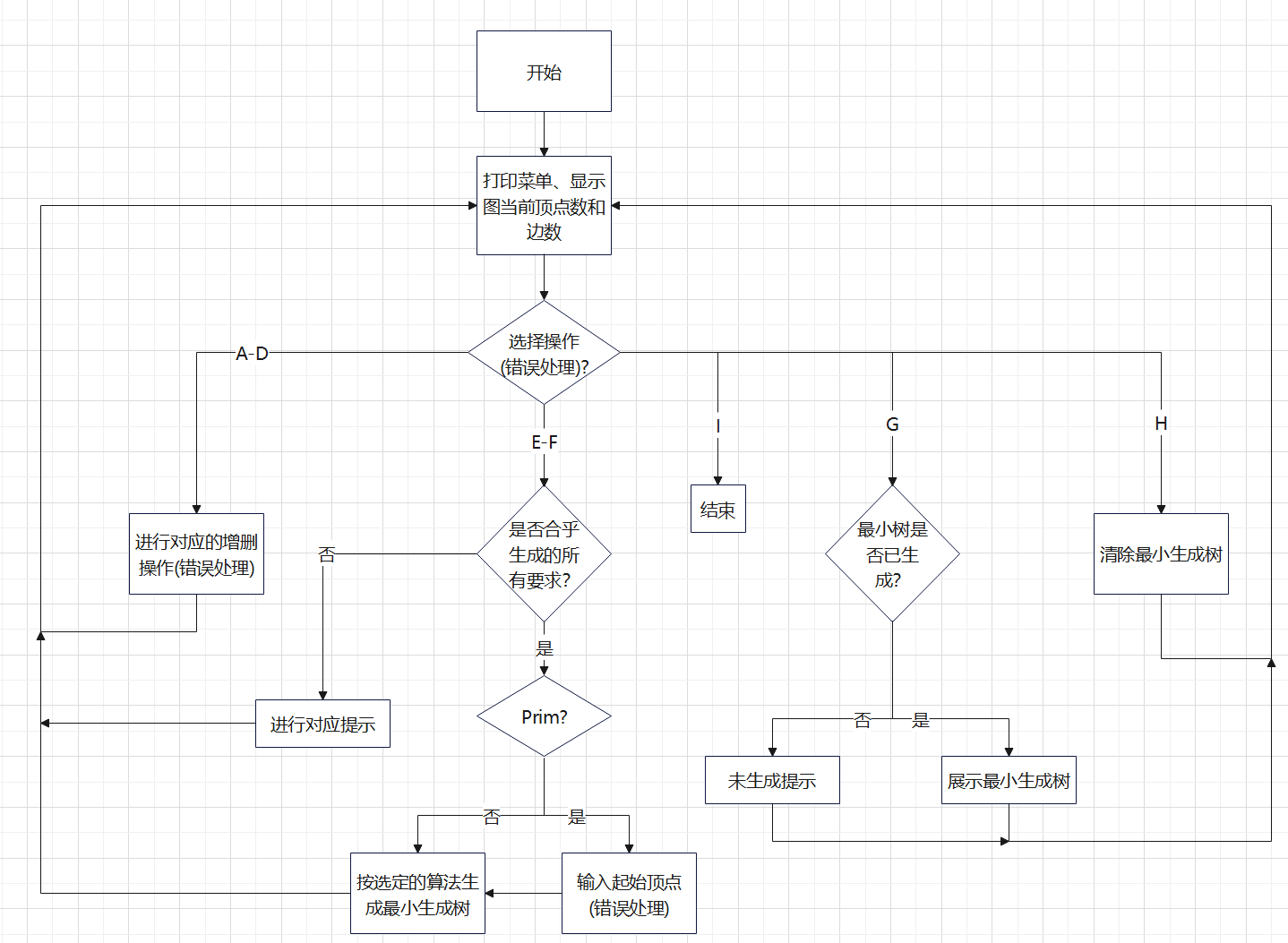
    delete[] Vmset;

}

**2.3.5 项目主体部分**

由于实现了两种算法，为了方便用户对两种算法进行对比，程序对最小生成树进行了暂存，由用户手动选择生成、删除和展示。若未生成，则无法展示；若已生成，则无法再次生成，需要先清除。

·流程图表示：



·代码实现：

int main()

{

    GraphMatrix<String, int> graph;  /\*图\*/

    MinSpanTree<String, int> tree;   /\*最小生成树\*/

    bool if\_on = true;

    while (if\_on) {

        cls();

        print\_menu();  /\*打印菜单\*/

        cout << "\n当前状态：" << graph.NumOfVertices() << "个顶点，"  /\*打印当前状态\*/

            << graph.NumOfEdges() << "条边，"

            << "最小树：" << (!tree.IsEmpty() ? "已生成" : "未生成") << endl;

        cout << "\n请选择操作：";

        char selection = '\0';

        while (1) {

            cin >> selection;

            cin.ignore(INT\_MAX, '\n');

            if ((selection >= 'A' && selection <= 'I') || (selection >= 'a' && selection <= 'i'))

                break;

            cout << "请重新选择：";

        }

        switch (selection) {

        case 'A':

        case 'a':

            AddVertices(graph);

            break;

        case 'B':

        case 'b':

            AddEdges(graph);

            break;

        case 'C':

        case 'c':

            RemoveVertices(graph);

            break;

        case 'D':

        case 'd':

            RemoveEdges(graph);

            break;

        case 'E':

        case 'e':

            SpanKruskal(graph, tree);

            break;

        case 'F':

        case 'f':

            SpanPrim(graph, tree);

            break;

        case 'G':

        case 'g':

            if (!tree.IsEmpty())

                tree.Show();

            else

                cout << "最小树未生成过，请先生成！" << endl;

            pause();

            cls();

            break;

        case 'H':

        case 'h':

            tree.MakeEmpty();

            cout << "最小树清除成功！" << endl;

            pause();

            cls();

            break;

        case 'I':

        case 'i':

            if\_on = false;

            cls();

            print\_menu();

            cout << "\n系统已经关闭！" << endl;

            break;

        }

    }

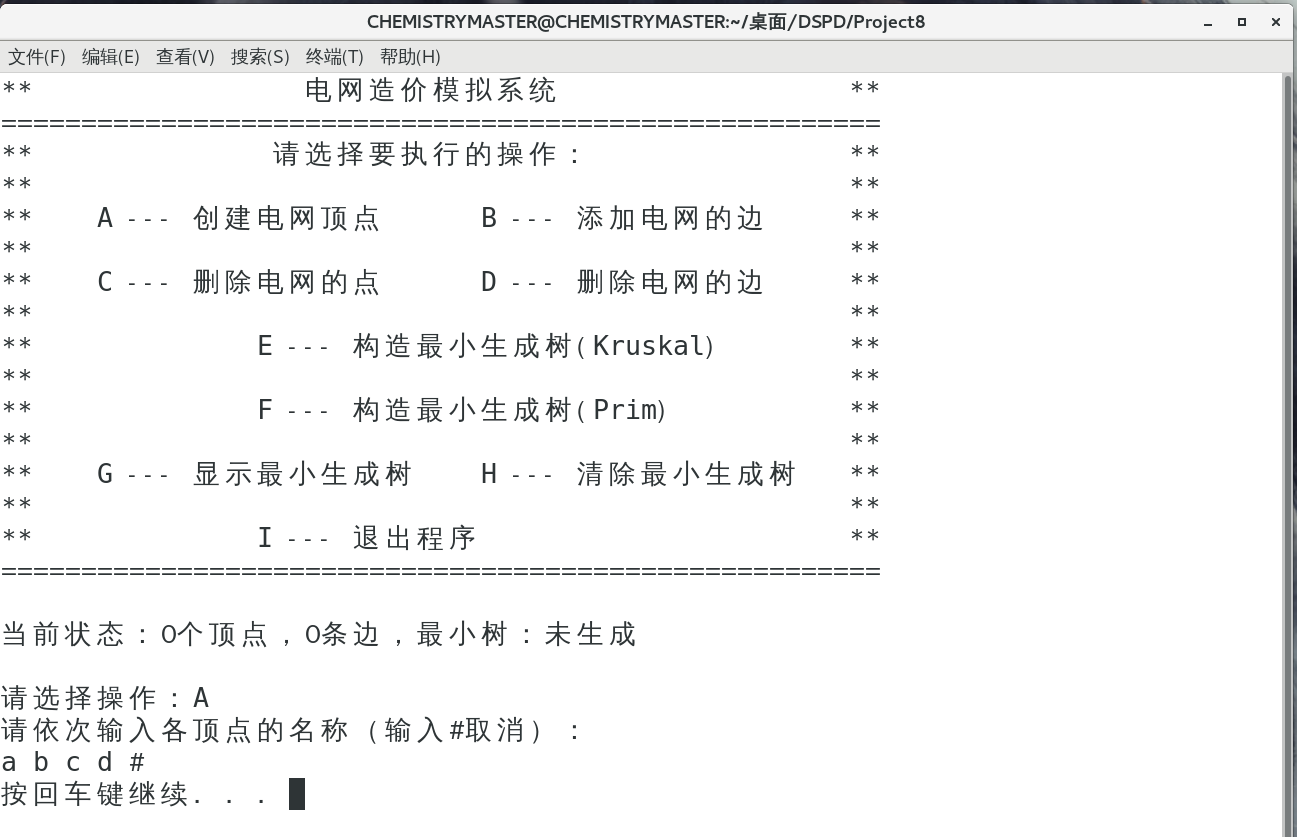
    return 0;

}

**3.项目测试**

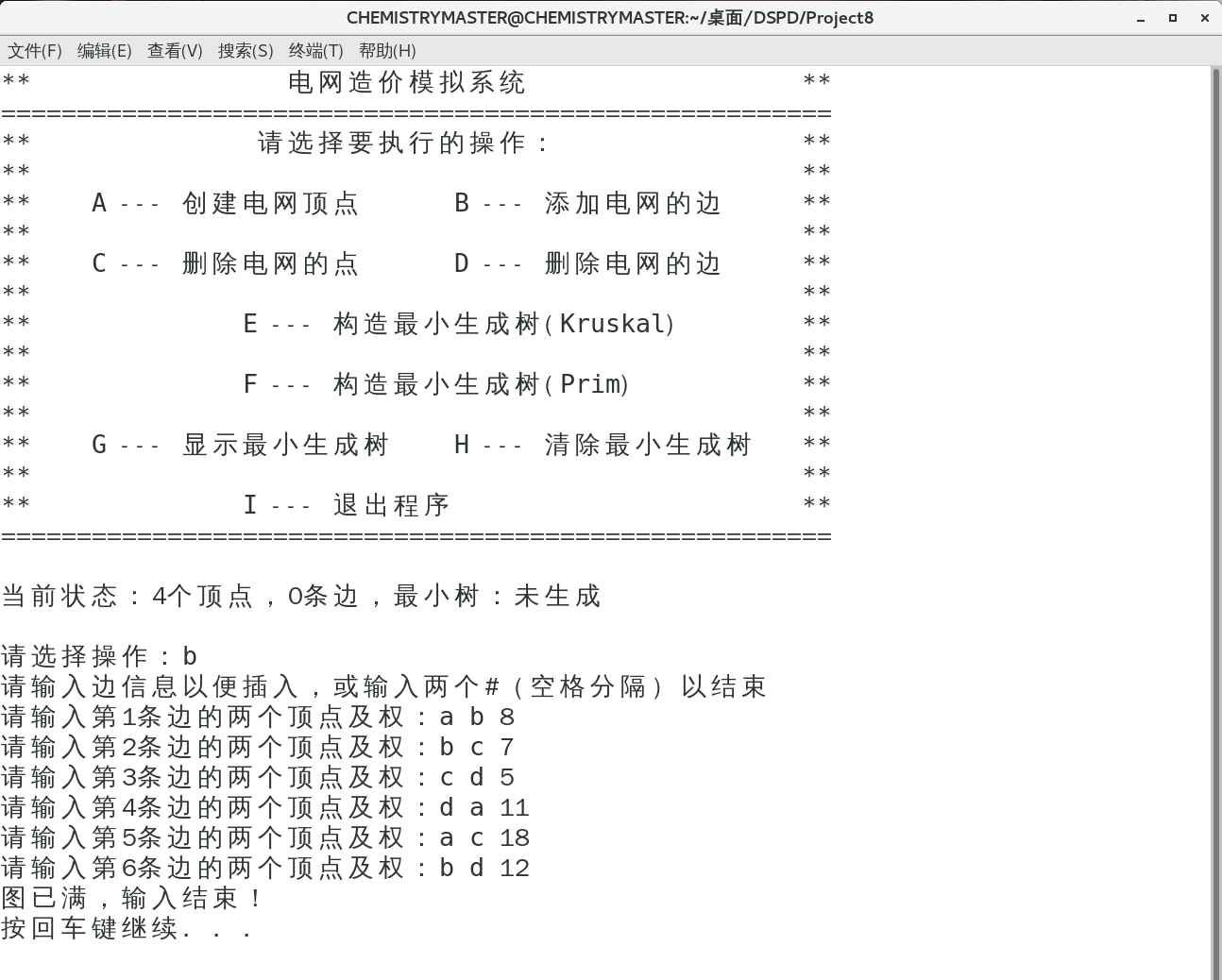
3.1 插入顶点测试

测试结果：



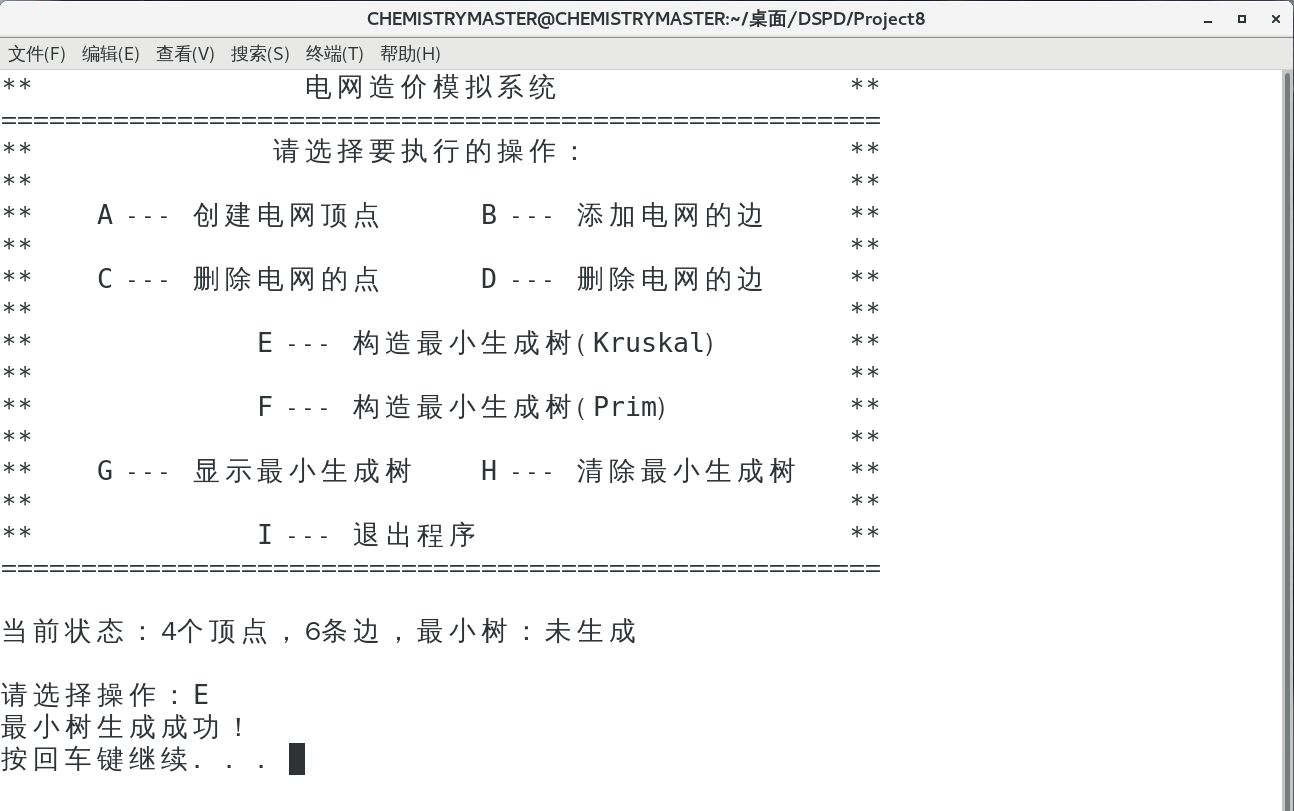
3.2 插入边测试

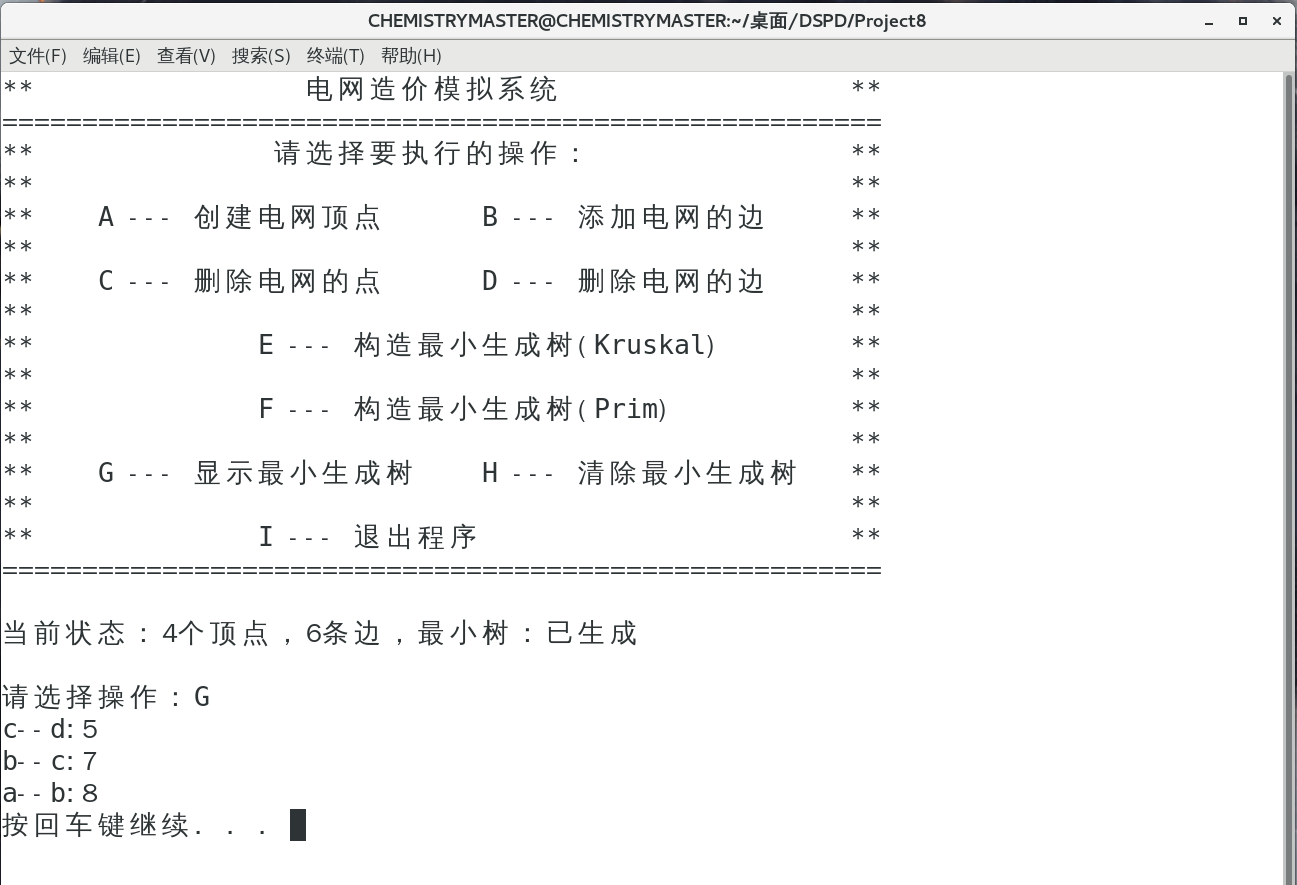
测试结果：



3.3 Kruskal & 展示

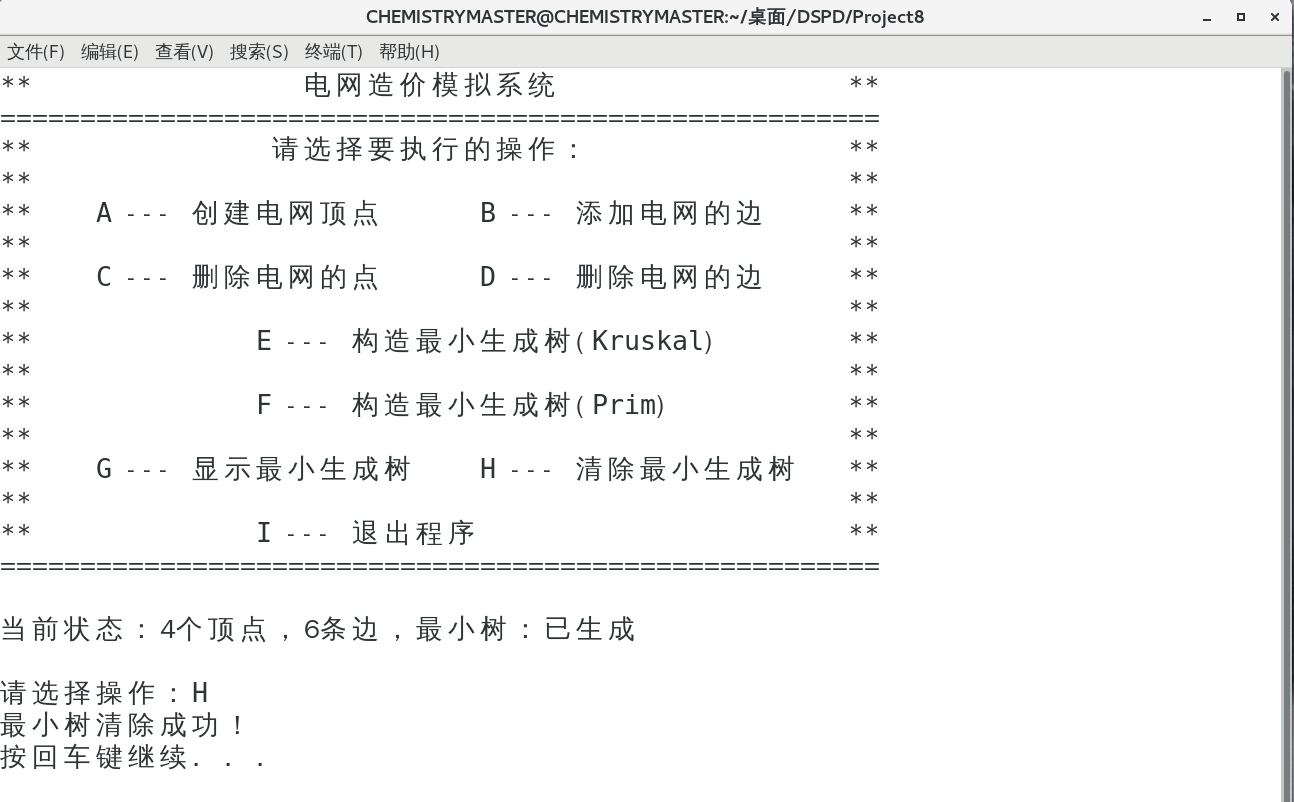
测试结果：





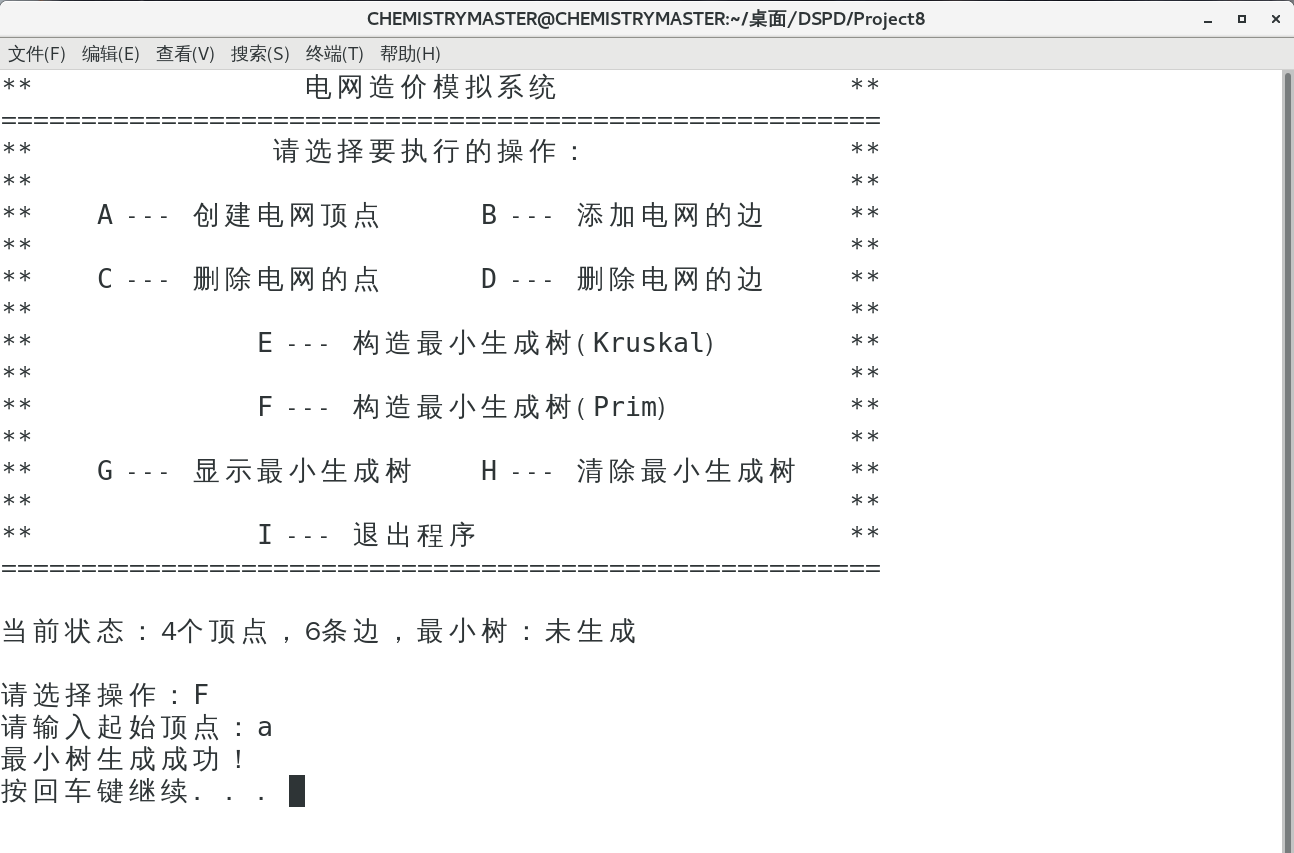
3.4 清除测试

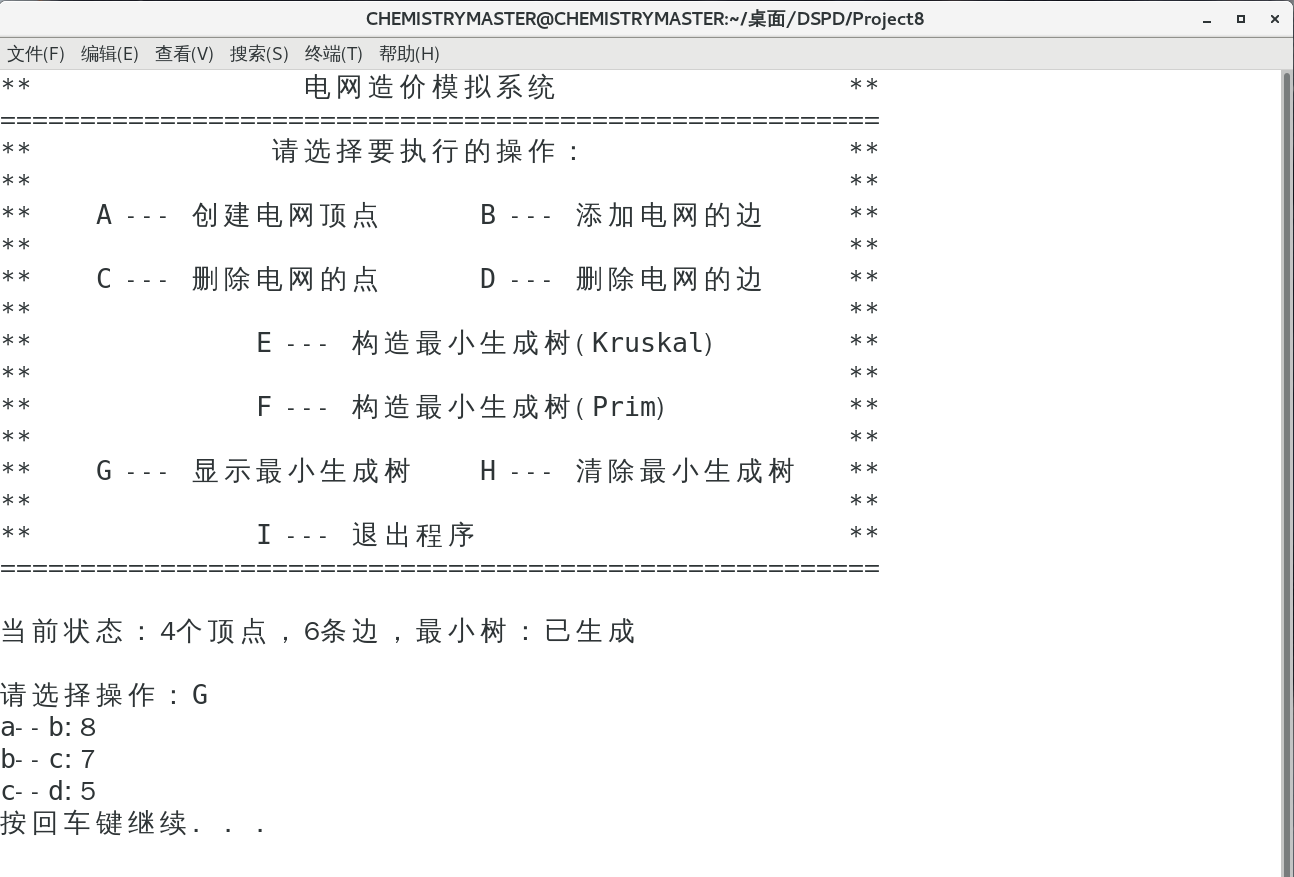
测试结果：



3.5 Prim & 展示

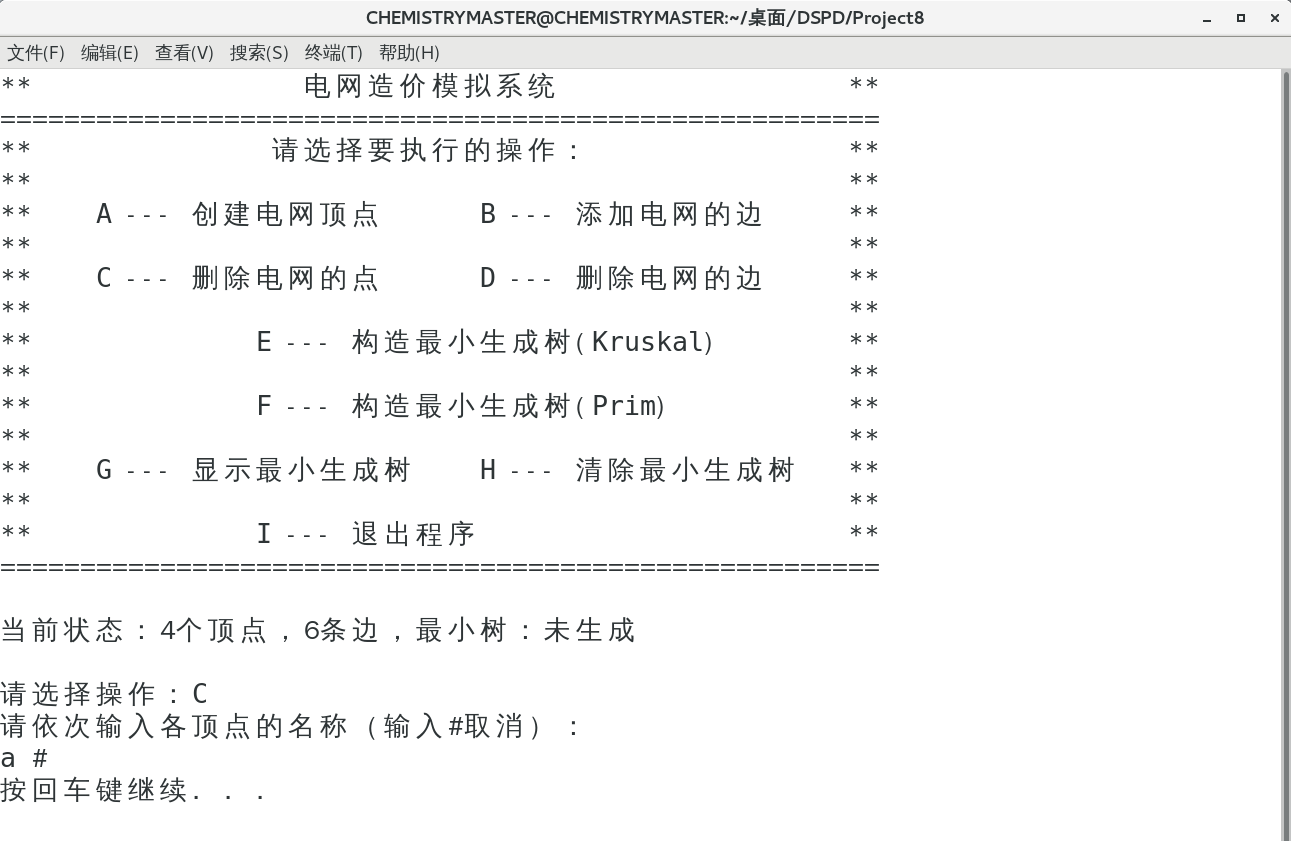
测试结果：





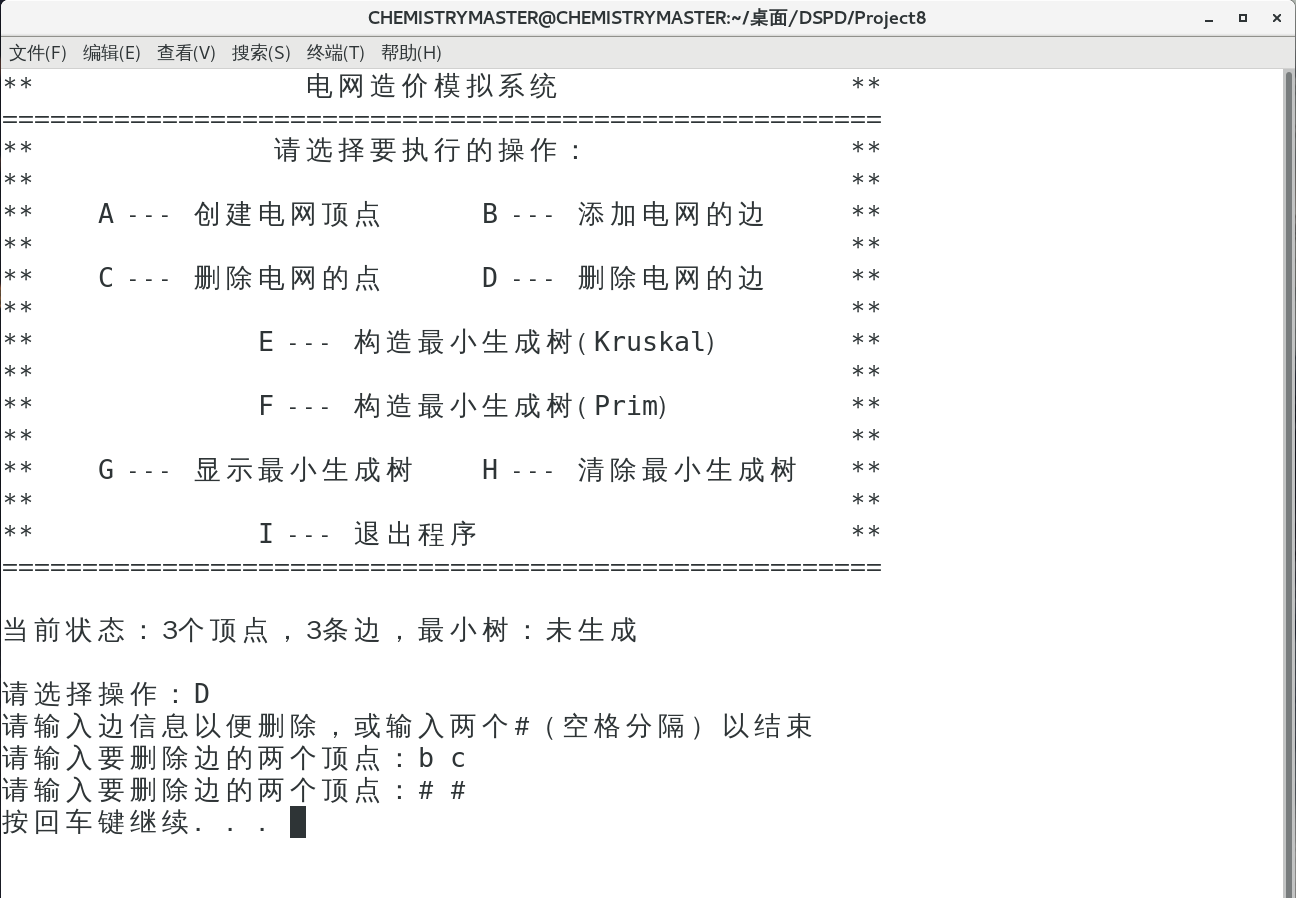
3.6 删除顶点测试

测试结果：



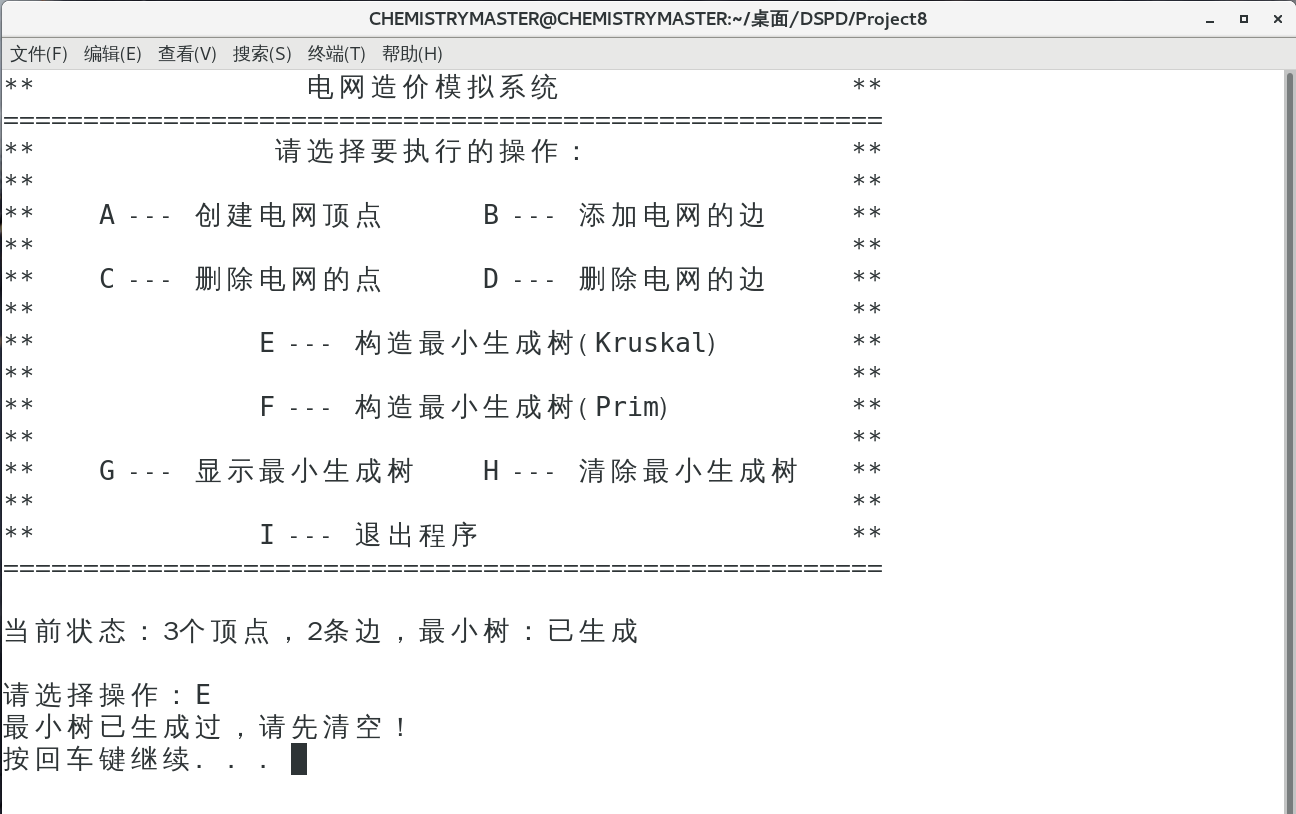
3.7 删除边测试

测试结果：



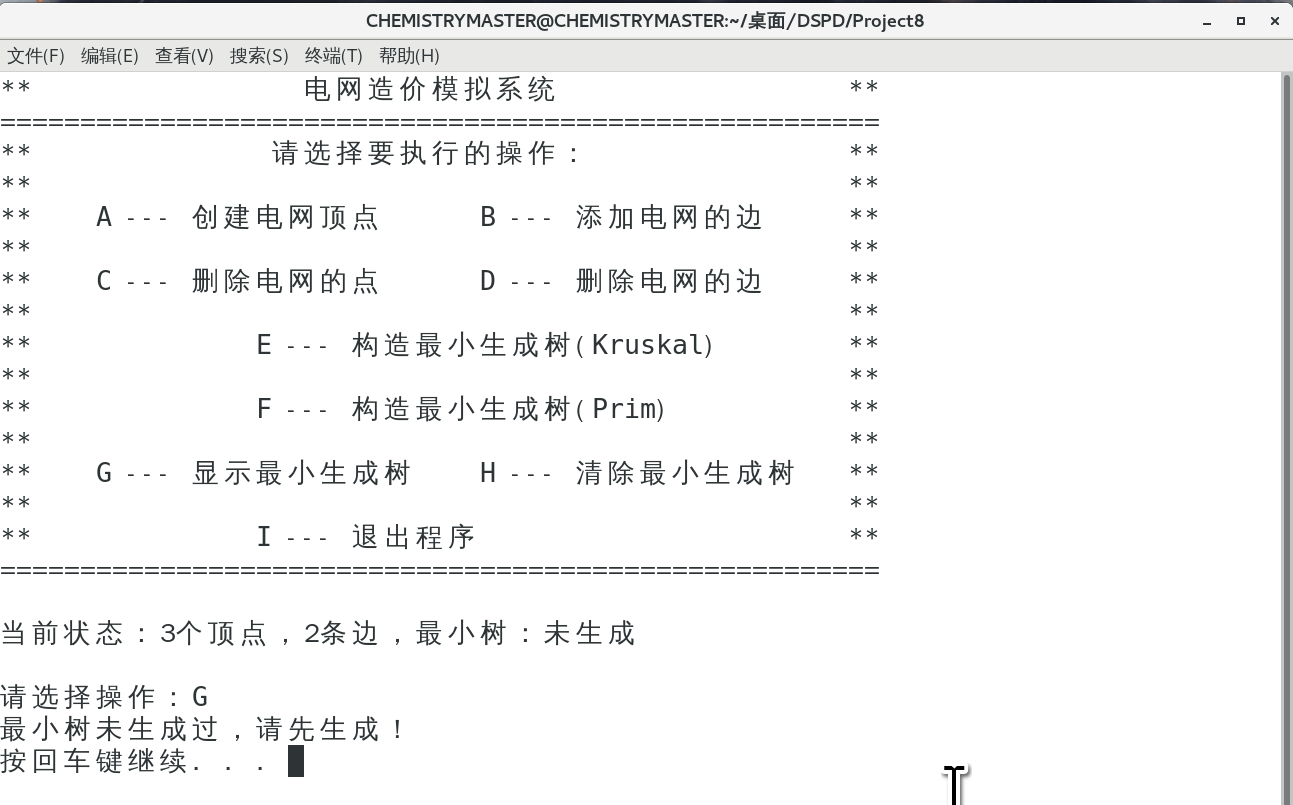
3.8 未清除就生成

测试结果：



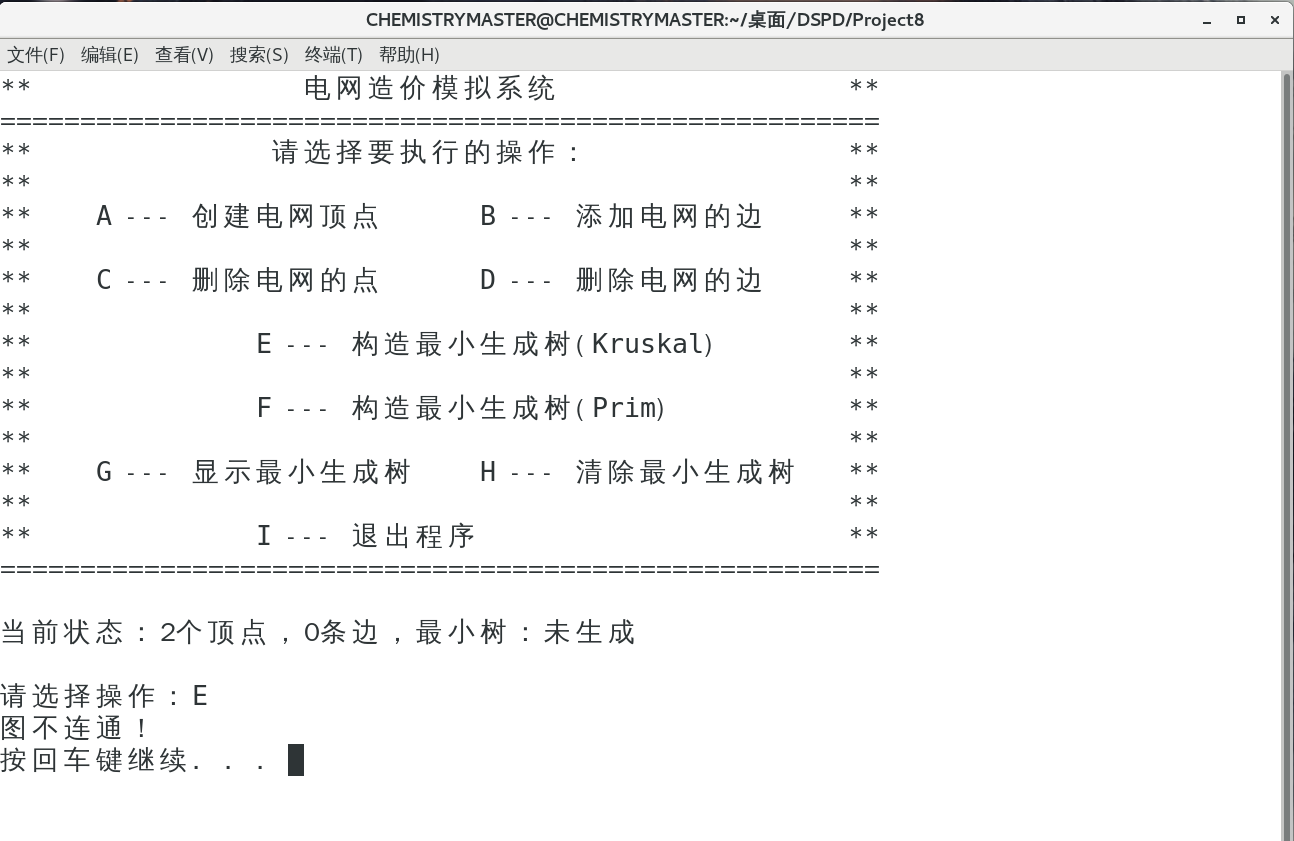
3.9 未生成就展示

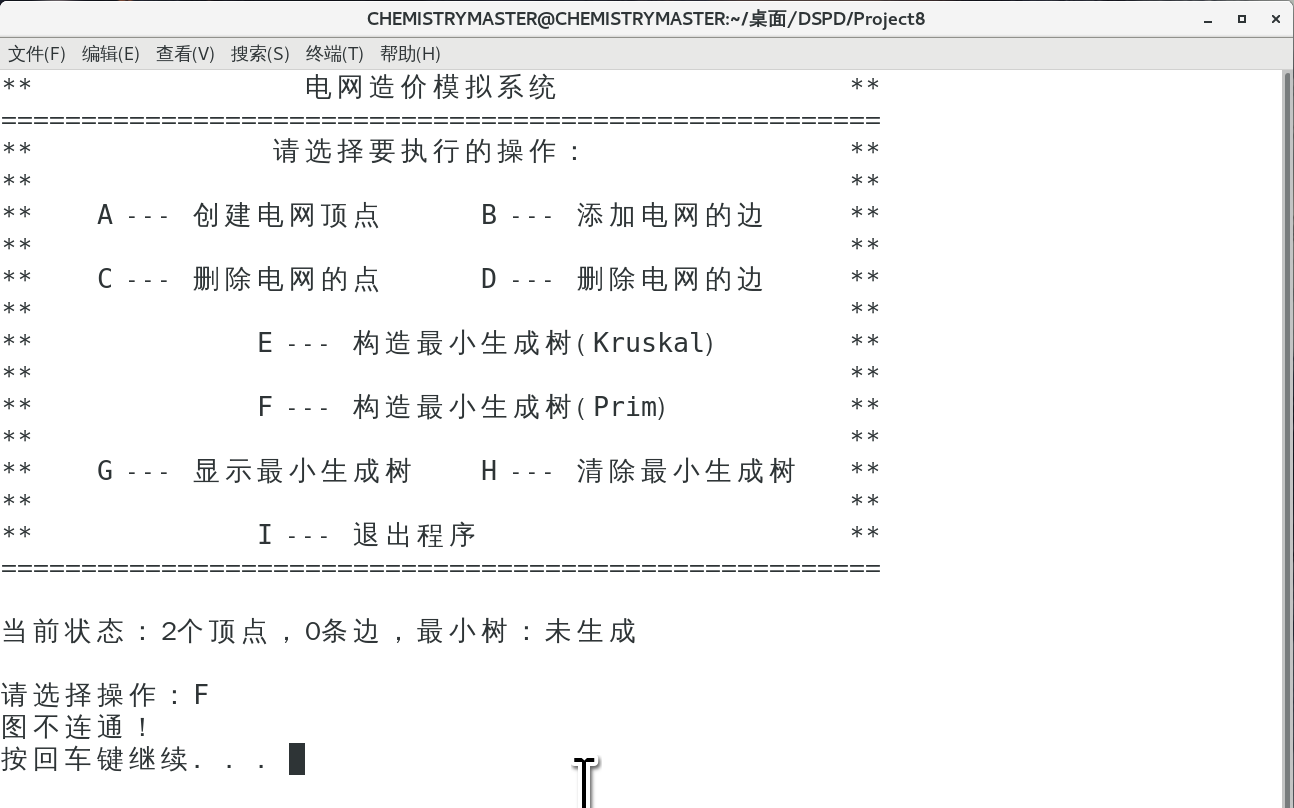
测试结果：



3.10 图不连通的情况

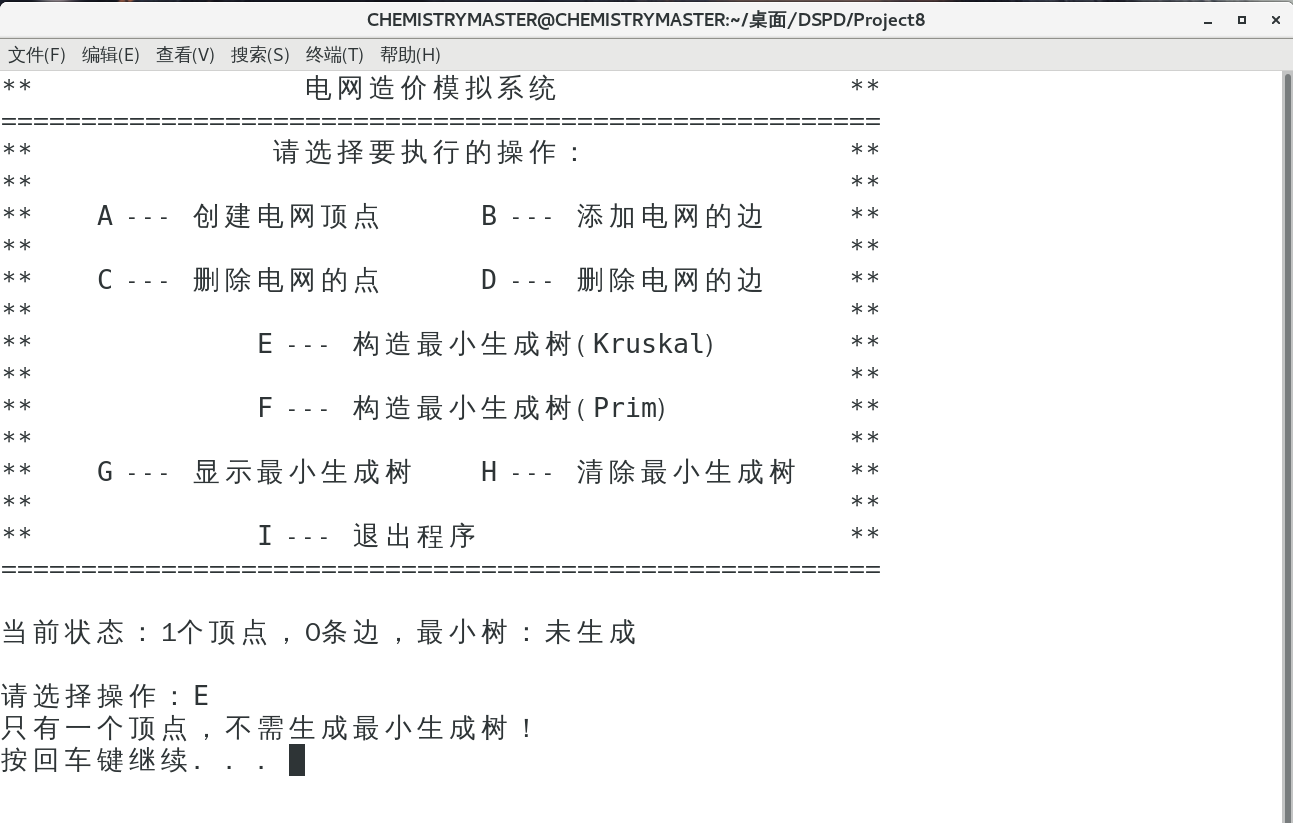
测试结果：

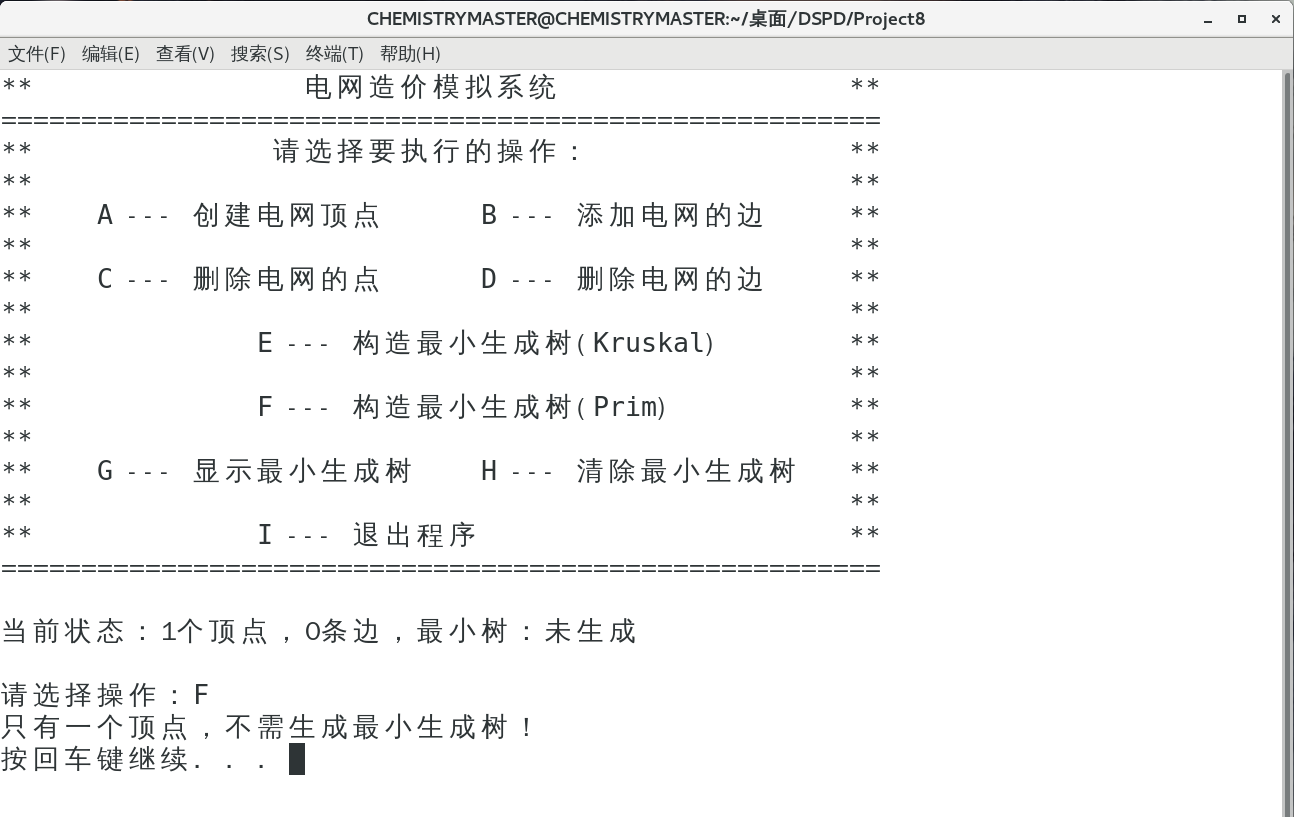




3.11 图为平凡图的情况

测试结果：





3.12 程序退出测试

测试结果：

