同济大学
TONGJI UNIVERSITY

项目说明文档

电网建设造价模拟系统

指导老师：张颖

1851009 沈益立

目录

1.分析

1.1 背景分析

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 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，选择其中的n-1条使总的耗费最少。

2.设计

2.1 数据结构设计

n个小区之间通过电网的形式连通，容易让人联想到图的结构，由于电网的特性（顶点和边）以及题目中的造价（边权值）能很好地匹配题目的需求，因此将电网定义为带权的无向图。

对于图的边，设计了一个结构体`EdgeNode`来存储，其包含边的头、尾和权值。

最小生成树所包含的每个边应该用一个数组储存下来。

同时，由于Kruskal算法可以利用最小堆来进行优化，因此设计了一个最小堆的结构。

由于Kruskal算法需要判断堆顶的边是否已经加入了最小生成树，因此设计了一个并查集`UFS`判断两个顶点是否在同一连通图内。

2.2 类结构设计

由于作业的规定，n个小区之间通过电网来连通，对小区的定义为带权值的无向图，设计了一个类Graph来存储这个无向图，并且使用邻接矩阵来作为图的存储形式。由于可能输入的顶点是char类型，因此设计了两个数组：`m_hash[256]`将字符(char)转化为顶点在邻接矩阵中的真实位置(int)、`m_charHash[101]`将顶点在邻接矩阵中的真实位置(int)转化为它的字符形式(char)。

定义了一个结构体：节点的边`EdgeNode`，其包含边的头、尾和权值。

定义了一个类：最小生成树`MST`，用于插入和存储最小生成树的结果。

定义了一个类：并查集`UFS`，可以判断两个顶点是否在同一连通图内。

定义了一个类：最小堆`MinHeap`，用于优化Kruskal算法的查找速度。

定义了一个类：`Solution`，用于处理IO和存放各个类指针，避免使用全局变量。

经典的树结构一般包括两个抽象数据类型(ADT)——树节点类`FamilyMember`和树类`FamilyTree`， 而两个类之间的耦合关系可以采用嵌套、继承等多种关系。本程序中将树节点类作为树类的成员变量，使得家谱树可以访问成员。同时，我设计了一个解决方案类`Solution`来调用树类，实现功能控制。

2.3 成员和操作设计

无向图类`Graph`

公有操作：

```
Graph();
void insertVertex(char vertexName);          //插入一个新的顶点，并且给它映射出一个唯一的编号
bool insertEdge(EdgeNode& e);                //插入新边，存入邻接矩阵中
int getVertexNum() { return m_vertexNum; }   //返回顶点总量
int getEdgeNum() { return m_edgeNum; }       //返回无向边的总量
int getEdgeWeight(char u, char v);           //返回u，v节点之间的权值
char getVertexByIndex(int index);            //通过顶点的编号获得顶点的名称
int getVertexByName(char name);              //通过顶点的名称获得顶点的编号
void showMatrix();                           //展示邻接矩阵
```

公有成员：

```
int m_edgeNum = 0;                                  //存储总边数
int m_vertexNum = 0;                                //存储总节点数

char m_charHash[101];                               //将顶点编号映射
为字符
int m_hash[256];                                    //将字符名称映射
成顶点编号
int m_graphMatrix[101][101];                        //邻接矩阵容器
```

无向图边类`EdgeNode`:

公有成员:

```
struct EdgeNode {
    char head;                                      //头结点名
    char tail;                                      //尾结点名
    int cost;                                       //权值
};
```

最小生成树类`MST`

公有操作:

```
MST() { m_edgeArr = new EdgeNode[VERTEXMAX]; }  //无参构造
函数
MST(int maxSize);                                   //含参构造
函数
void insert(EdgeNode item);                         //插入新的
最小生成树的边

void showMST();                                     //展示MST的
结论
```

私有成员

```
EdgeNode* m_edgeArr;                    //存放最小生成树边的数组
int m_num = 0;                          //最小生成树总数
int m_maxSize = VERTEXMAX;              //数组的最大值
```

并查集类`UFS`

公有操作:

```
UFS(int size);                          //含初始大小的构造函数
int find(int index);                    //寻找某节点的并查集的根
void unite(int i, int j);               //将i, j节点合并至同一集合中
```

私有成员

```
int m_size;                             //并查集的大小
int* m_parent;                          //存储并查集的数组
```

最小堆类`MinHeap`

公有操作:

```
MinHeap();                  //无参构造函数
MinHeap(int size)           //含大小的构造函数
    :m_maxSize(size) {
        m_heap = new EdgeNode[size];
        auto p = m_heap;
        for (int i = 0; i < m_maxSize; i++) {
            p = new EdgeNode();
            p++;
```

```
        }

    };
void insert(EdgeNode val);       //向小根堆插入新元素
EdgeNode pop();                 //小根堆弹出元素
EdgeNode getTop();                  //获取堆顶值
bool isEmpty();             //判断堆空
bool isFull();              //判断堆满
~MinHeap() {};              //默认析构函数
```

私有操作

```
void filterUp(int start);//自底向上调整堆
void filterDown(int start, int end);//自顶向下调整堆
```

私有成员

```
EdgeNode* m_heap;                   //用数组模拟堆
int m_maxSize;              //堆的最大大小
int m_defaultSize = 10; //默认大小
int m_curSize = 0;          //当前堆的大小
```

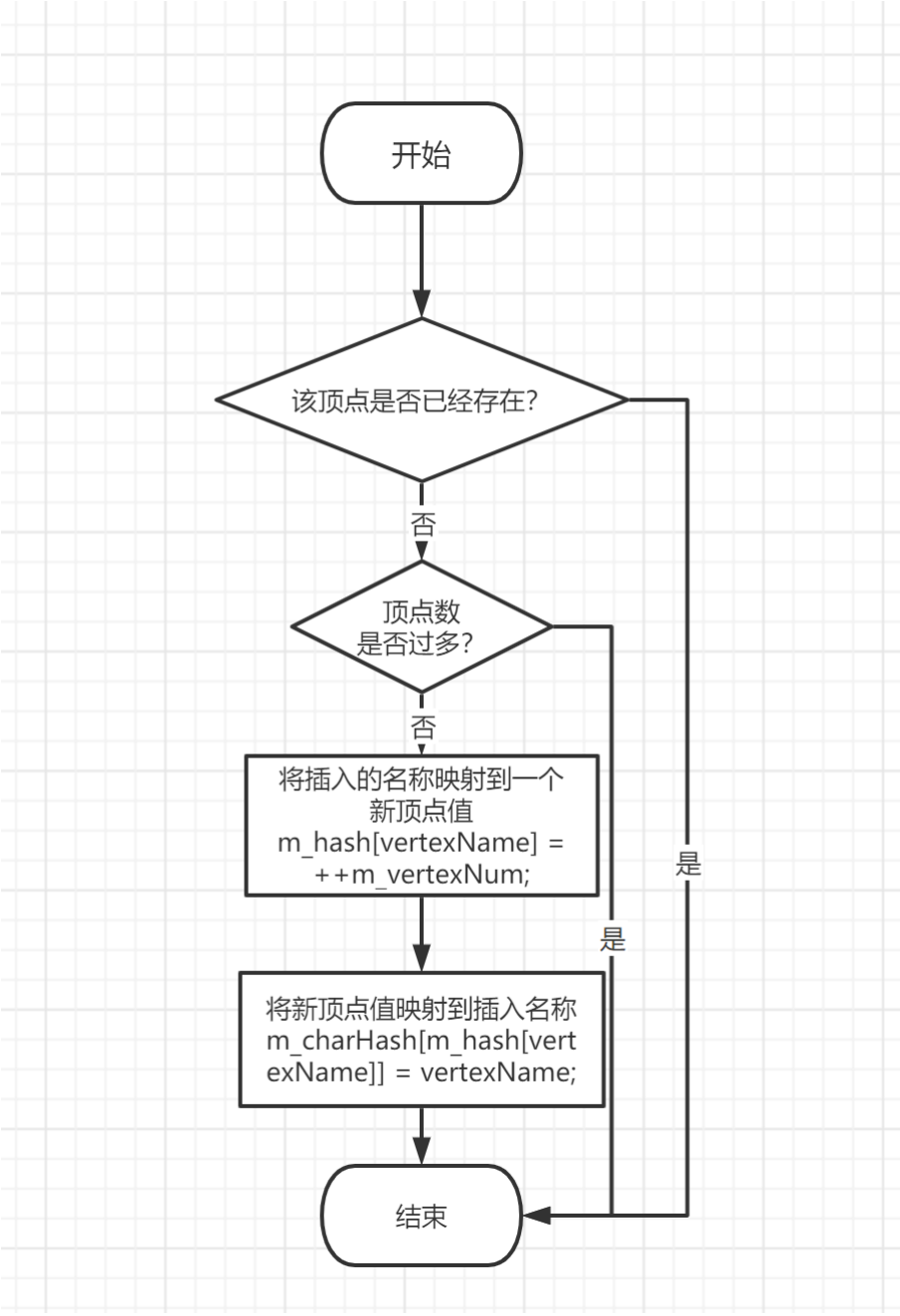
2.4 系统设计

程序运行后，系统会自动创建一个解决方案的实例化对象`sol`，随后调用基本的IO来实现屏幕的初始化和功能输入，建立整个表示电网的带权值的无向图；之后，程序会根据用户的输入进行选择，并根据用户的输入执行不同的操作，当输入的操作码不为'E'时，给出相应的操作，否则会退出程序。

3.实现

3.1 无向图插入新顶点功能的实现

3.1.1 无向图插入新顶点功能流程图

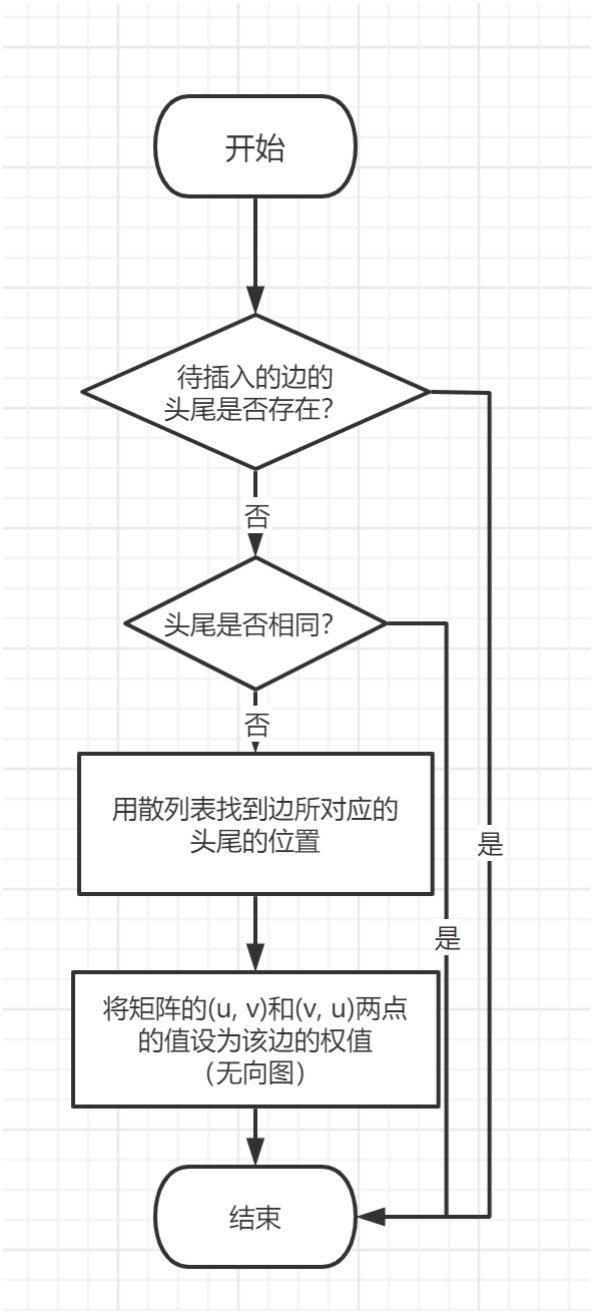


3.1.2 无向图插入新顶点核心代码实现

```
void Graph::insertVertex(char vertexName) {
    if (m_hash[vertexName] != -1) {
        cerr << "this vertex is existed, insertion
failed." << endl;
        return;
    }

    if (m_vertexNum >= VERTEXMAX) {
        cerr << "too much vertexes, insertion failed." <<
endl;
        return;
    }
    m_hash[vertexName] = ++m_vertexNum;
    m_charHash[m_hash[vertexName]] = vertexName;
}
```

3.2 无向图插入新边功能的实现

3.2.1 无向图插入新边功能流程图



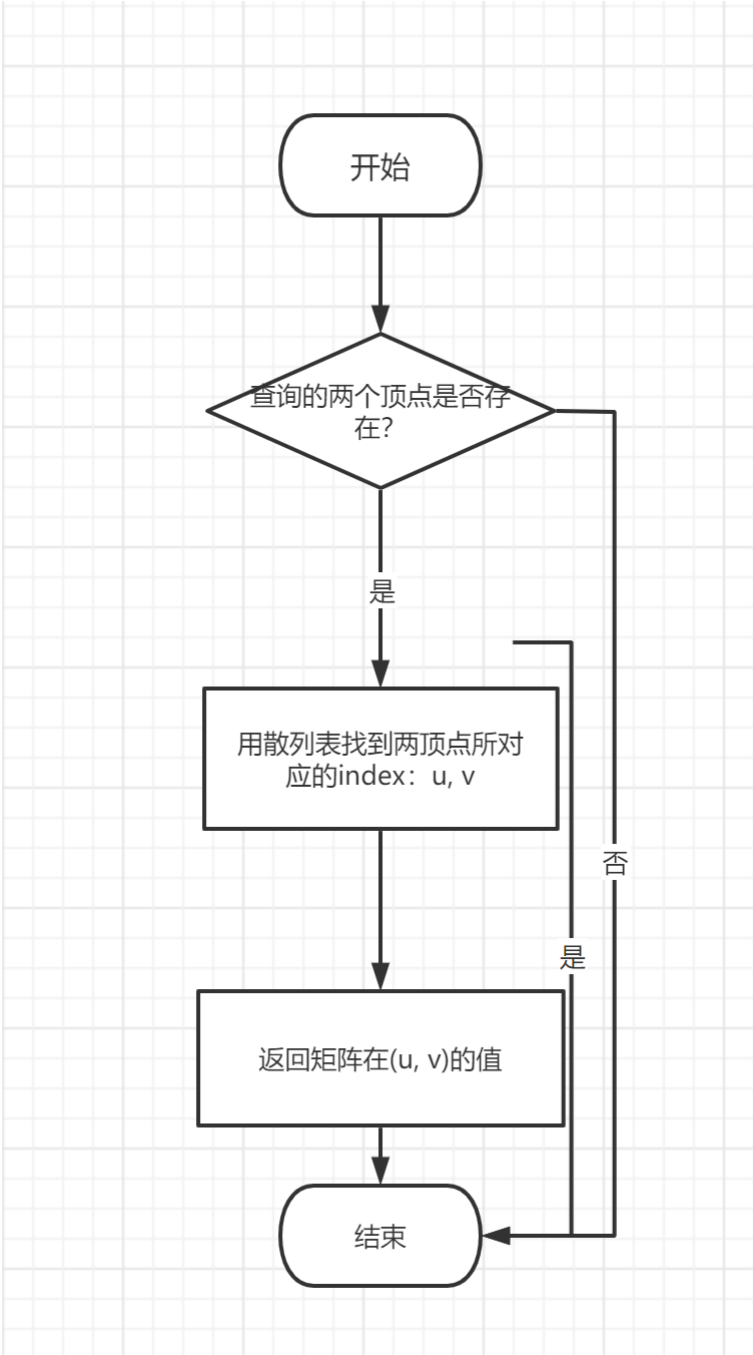
3.2.2 完善功能核心代码实现

```
bool Graph::insertEdge(EdgeNode& e) {
    if (m_hash[e.head] == -1 || m_hash[e.tail] == -1) {
        cerr << "Insert edge failed. The vertexes of this
edge does not exist, please check." << endl;
        return false;
    }

    if (e.head == e.tail) {
```

```
        cerr << "Insert edge failed. The cost to vertex
itself should be 0. (There should be no ring in the
graph.)" << endl;
        return false;
    }
    if (m_graphMatrix[e.head][e.tail] == MAXVALUE) {
        m_edgeNum++;
    }
    int indexU = m_hash[e.head] , indexV = m_hash[e.tail]
;
    m_graphMatrix[indexU][indexV] = e.cost;
    m_graphMatrix[indexV][indexU] = e.cost;//无向图，对称

    return true;
}
```

3.3 取出两点之间权值功能的实现

3.3.1 取出两点之间权值功能流程图

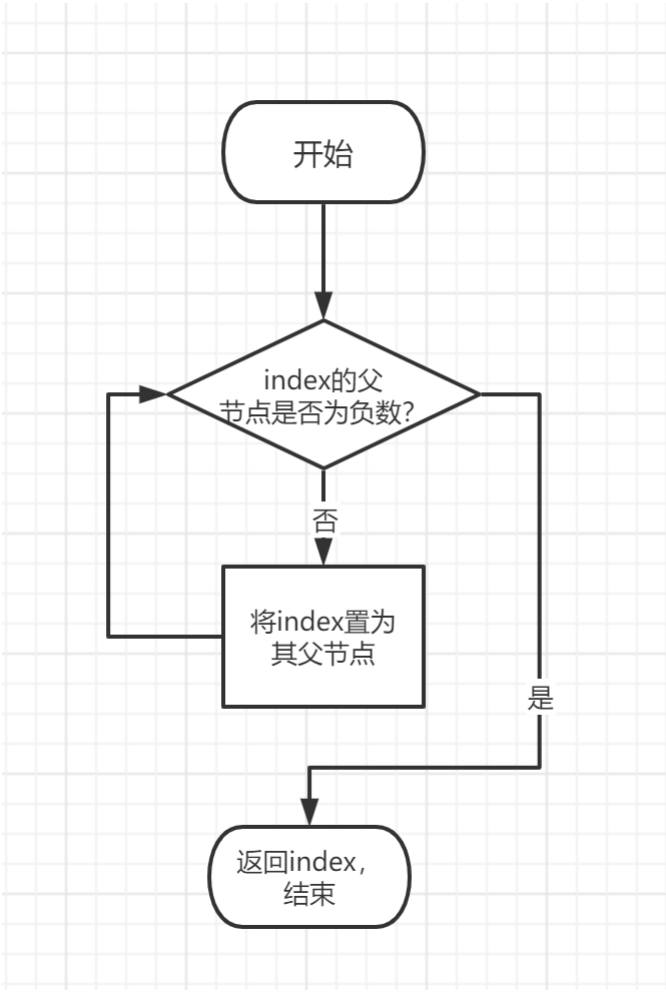


3.3.2 取出两点之间权值核心代码实现

```cpp
int Graph::getEdgeWeight(char u, char v) {
    if (m_hash[u] == -1 || m_hash[v] == -1) {
        cerr << "no such vertex." << endl;
        return 0;
    }

    int indexU = m_hash[u], indexV = m_hash[v];

    return m_graphMatrix[indexU][indexV];
}
```

3.4 并查集查找根节点功能的实现

3.4.1 并查集查找根节点流程图

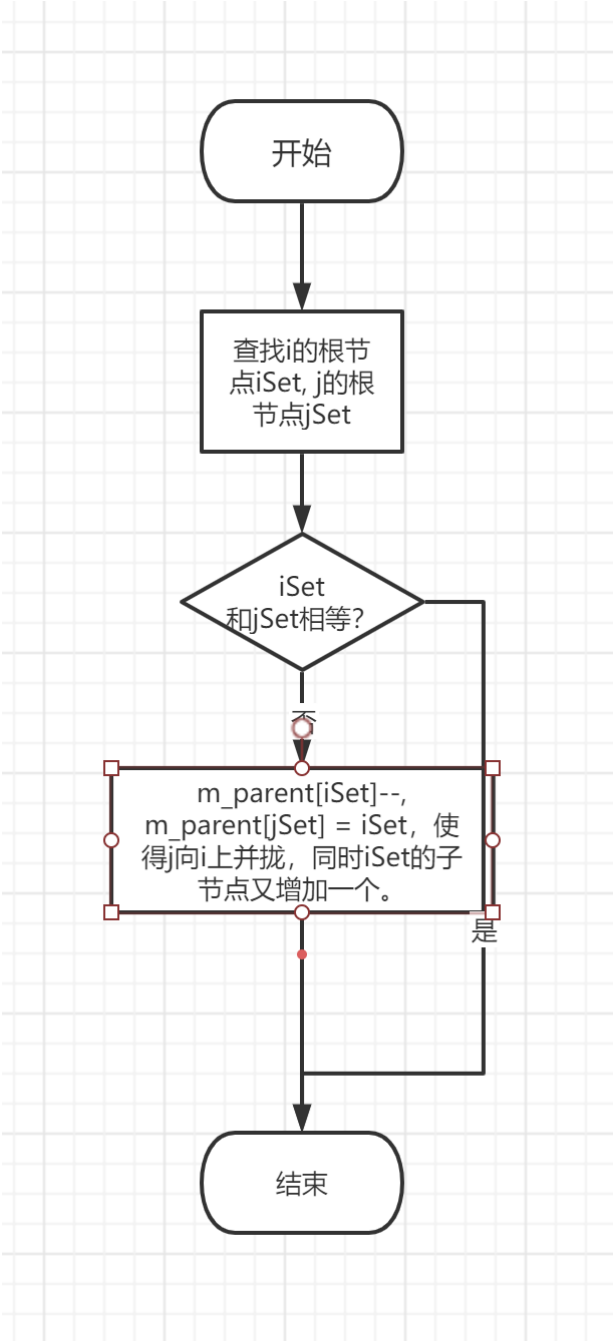


3.4.2 并查集查找根节点核心代码实现

```
int UFS::find(int index) {
    while (m_parent[index] >= 0) {
        index = m_parent[index];
    }

    return index;
}
```

3.5 并查集合并功能的实现

3.5.1 并查集合并功能流程图

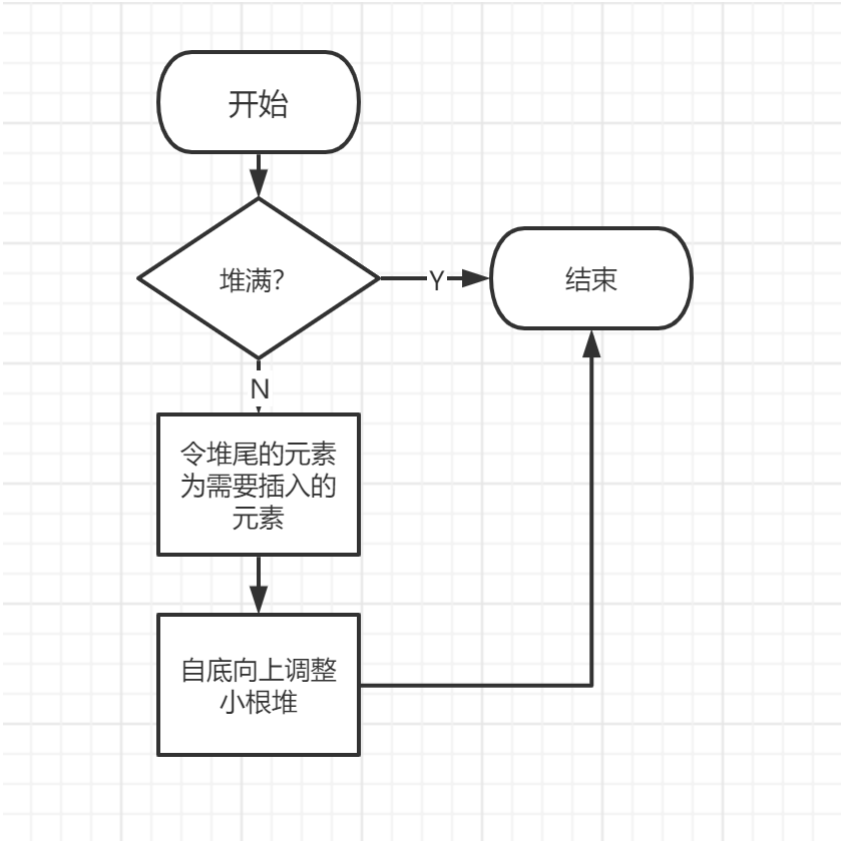


3.5.2 并查集合并功能核心代码实现

```
void UFS::unite(int i, int j) {
    auto iSet = find(i), jSet = find(j);
    if (iSet == jSet) {
        cerr << "They belong to the same set." << endl;
        return;
    }

    m_parent[iSet]--;
    m_parent[jSet] = iSet;
}
```

3.6 小根堆压入功能的实现

3.6.1 小根堆压入功能流程图

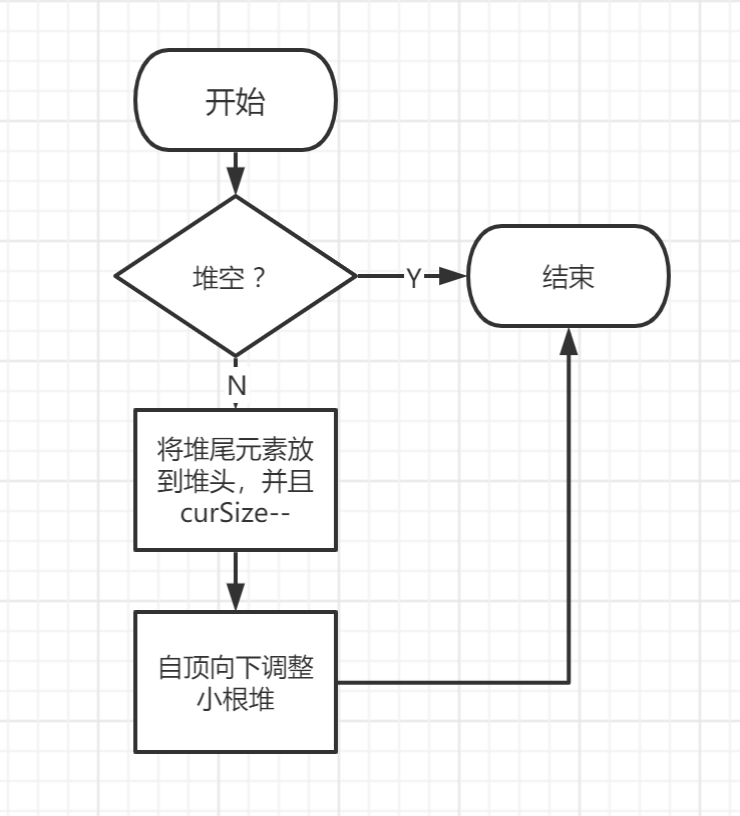


3.6.2 小根堆压入功能核心代码实现

```
void MinHeap::insert(EdgeNode val) {
    if (isFull()) {
        cerr << "堆满，请建更大的堆" << endl;
        return;
    }
    //插入逻辑，尾插后上浮调整
    m_heap[m_curSize] = val;
    filterUp(m_curSize++);
}
```

3.7 小根堆弹出功能的实现

3.7.1 小根堆弹出功能流程图

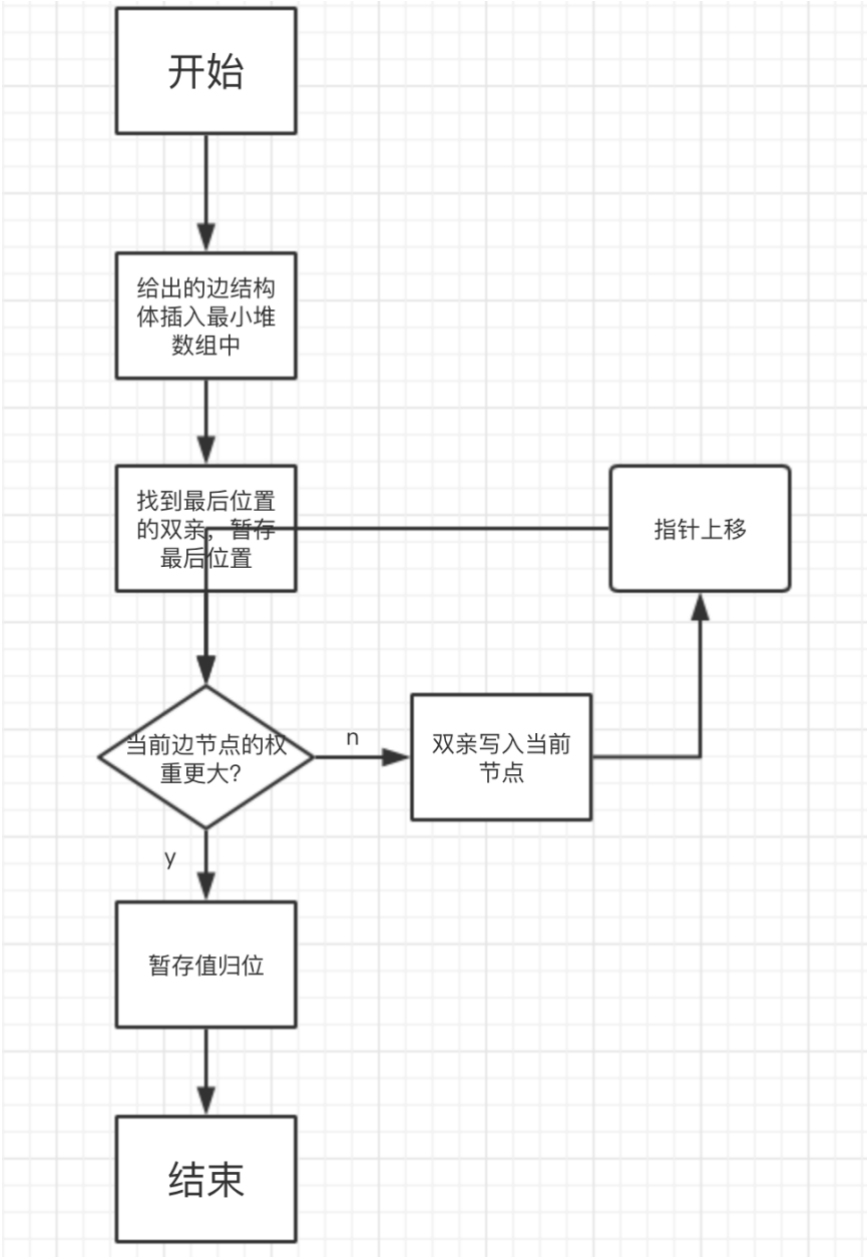


3.7.2 小根堆弹出功能核心代码实现

```
EdgeNode MinHeap::pop() {
    if (isEmpty()) {
        cerr << "空集，无法弹出堆顶元素" << endl;
        return EdgeNode();
    }
    EdgeNode retVal = m_heap[0];
    m_heap[0] = m_heap[m_curSize - 1];  //取堆尾元素
    m_curSize--;
    filterDown(0, m_curSize - 1);   //自上而下调整堆
    return retVal;
}
```

3.8 小根堆向上调整功能的实现

3.8.1 小根堆向上调整功能流程图



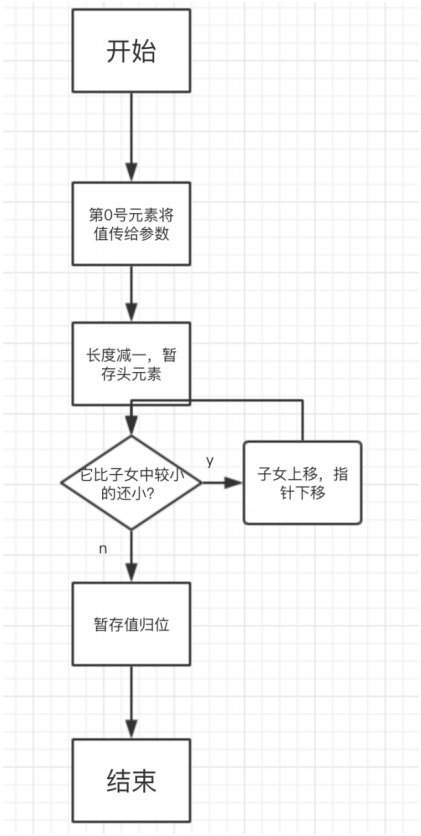
3.8.2 小根堆向上调整功能核心代码实现

```
void MinHeap::filterUp(int start) {
    int j = start, i = (j - 1) / 2; //i表示j的父亲节点
    EdgeNode tmp = m_heap[start];            //暂存，从下向上调整，取要调整的节点
    while (j) {
        if (m_heap[i].cost <= tmp.cost) {        //父节点较小的情况
            break;
        }
        else {                          //父节点较大，与父节点互换
            m_heap[j] = m_heap[i];
```

```
            j = i;
            i = (i - 1) / 2;
        }
    }
    m_heap[j] = tmp;                        //回送
}
```

3.9 小根堆向下调整功能的实现

3.9.1 小根堆向下调整功能流程图



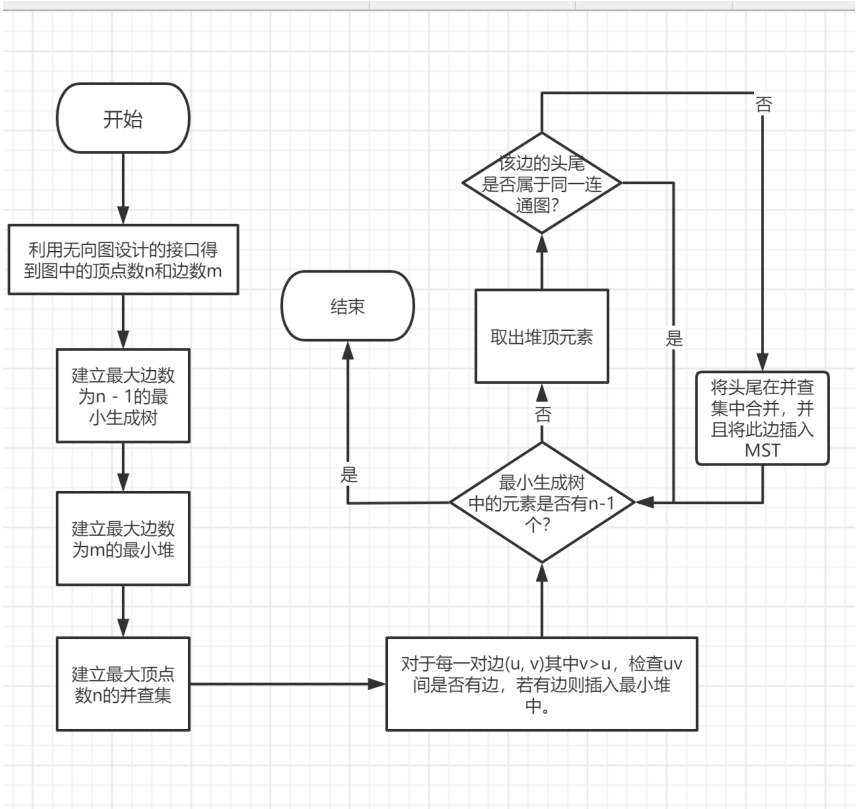
3.9.2 小根堆向下调整功能核心代码实现

```
void MinHeap::filterDown(int start, int end) {
    int i = start, j = 2 * i + 1;    //i表示父节点，j表示左子树
    EdgeNode tmp = m_heap[start];            //需要下浮，因此暂存起始节点
    while (j <= end) {                //j到达终点前
```

```
        if (j < end && m_heap[j].cost > m_heap[j +
1].cost) {
            j++;                        //取两子女中小者
        }
        if (tmp.cost <= m_heap[j].cost) {        //如果小，
就调整结束
            break;
        }
        else {                          //否则继续调整
            m_heap[i] = m_heap[j];
            i = j;
            j = 2 * j + 1;
        }
    }
    m_heap[i] = tmp;                        //回放tmp中的暂存元
素
}
```

3.10 Kruskal算法构造最小生成树的实现

3.10.1 Kruskal算法构造最小生成树流程图



3.10.2 Kruskal算法构造最小生成树核心代码实现

```
void Solution::kruskal() {
    auto n = m_graph->getVertexNum();
    auto m = m_graph->getEdgeNum();
    m_mst = new MST(n - 1);
    m_minPQ = new MinHeap(m);
    UFS F(n);
    for (auto u = 0; u < n; u++) {
        for (auto v = u + 1; v < n; v++) {
            auto charU = m_graph->getVertexByIndex(u),
                 charV = m_graph->getVertexByIndex(v);

            auto uvCost = m_graph->getEdgeWeight(charU,
charV);


            if (uvCost != MAXVALUE) {//uv间有边
                EdgeNode tmpEdge({ charU, charV, uvCost
});
                m_minPQ->insert(tmpEdge);
```

```
            }
        }
    }

    auto cnt = 1;
    while (cnt < n) {
        auto edge = m_minPQ->pop();
        auto indexU = m_graph->getVertexByName(edge.head)
           , indexV = m_graph->getVertexByName(edge.tail);
        auto u = F.find(indexU), v = F.find(indexV);

        if (u != v) {
            F.unite(u, v);
            m_mst->insert(edge);
            cnt++;
        }

    }
}
```

4.测试

4.1 常规测试

4.1.1 题目要求测试

测试用例：

```
A
4
a b c d
B
6
a b 8
b c 7
c d 5
d a 11
a c 18
b d 12
C
D
```

预期结果：

```
a<-8->b, b<-7->c, c<-5->d
```

实验结果：

```
==================================
注意：执行完一轮之后输入新图需初始化

请选择操作:
A
请输入顶点个数:4
请依次输入顶点的名称（仅支持一个char）：
a b c d
请选择操作:
B
请输入边的个数:（请注意，重复输入的边会覆盖此前输入的边）6
请依次输入两个顶点和边花费:
a b 8
b c 7
c d 5
d a 11
a c 18
b d 12
请选择操作:
C
成功生成MST(使用kruskal算法)
请选择操作:
D
c<-5->d b<-7->c a<-8->b
```

4.1.2 课本要求测试

测试用例：

```
A
7
0 1 2 3 4 5 6
B
9
0 1 28
0 5 10
1 2 16
2 3 12
3 4 22
3 6 18
1 6 14
4 5 25
4 6 24
C
D
```

预期结果：

```
5<-10->0, 4<-25->5, 3<-22->4, 2<-12->3, 1<-16->2, 6<-14->1
```

实验结果：

```
A
请输入顶点个数:7
请依次输入顶点的名称（仅支持一个char）：
0 1 2 3 4 5 6
请选择操作:
B
请输入边的个数:（请注意，重复输入的边会覆盖此前输入的边）9
请依次输入两个顶点和边花费:
0 1 28
0 5 10
1 2 16
2 3 12
3 4 22
3 6 18
1 6 14
4 5 25
4 6 24
请选择操作:
C
成功生成MST(使用kruskal算法)
请选择操作:
D
0<-10->5        2<-12->3        1<-14->6        1<-16->2        3<-22->4        4
<-25->5
```

4.1.3 离散数学样例测试

（感谢唐老师提供的测试点！）

测试用例：

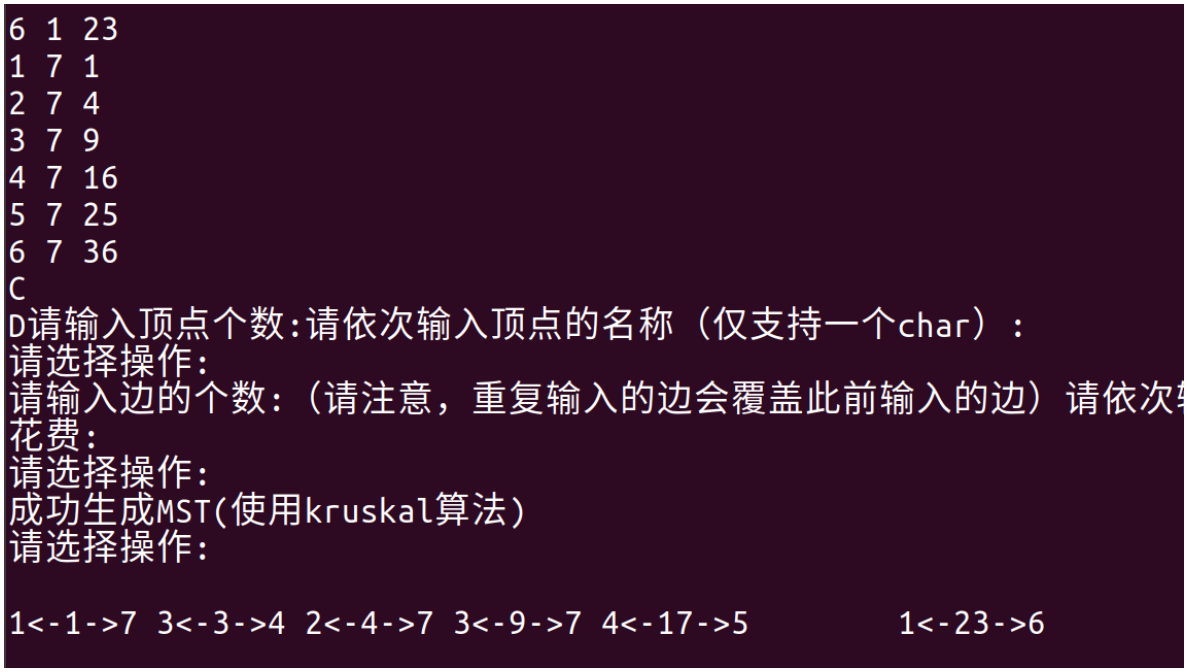
```
A
7
1 2 3 4 5 6 7
B
12
1 2 20
2 3 15
3 4 3
4 5 17
5 6 28
6 1 23
1 7 1
2 7 4
3 7 9
4 7 16
```

```
5 7 25
6 7 36
C
D
```

预期结果：

```
1<-1->7 3<-3->4 2<-4->7 3<-9->7 4<-17->5 1<-23->6
```

实验结果：



4.2 错误测试

4.2.1 非法节点个数

测试用例：

```
A
0
```

预期结果：

```
提示顶点非法
```

实验结果：

```
shenyili@shenyili:~/桌面/CostSimulation$ ./CostSimulation
**          电网造价模拟系统          **
====================================
**          A --- 创建电网顶点        **
**          B --- 添加电网的边        **
**          C --- 构造最小生成树      **
**          D --- 显示最小生成树      **
**          E --- 退出程序            **
**          F --- 显示邻接矩阵        **
**          G --- 重新初始化          **
====================================
注意：执行完一轮之后输入新图需初始化

请选择操作:
A
请输入顶点个数:0
顶点太多或非正shenyili@shenyili:~/桌面/CostSimulation$
```

4.2.2 不能添加的边

测试用例：

```
A
1
B
1
a b 1
```

预期结果：

> 提示不存在如下节点

实验结果：

```
请选择操作:
A
请输入顶点个数:1
请依次输入顶点的名称（仅支持一个char）：
a
请选择操作:
B
请输入边的个数:（请注意，重复输入的边会覆盖此前输入的边）1
请依次输入两个顶点和边花费:
a b 1
Insert edge failed. The vertexes of this edge does not exist, please check.
插入失败，请检查是否存在这些顶点
```

4.2.3 在初始化之前进行操作

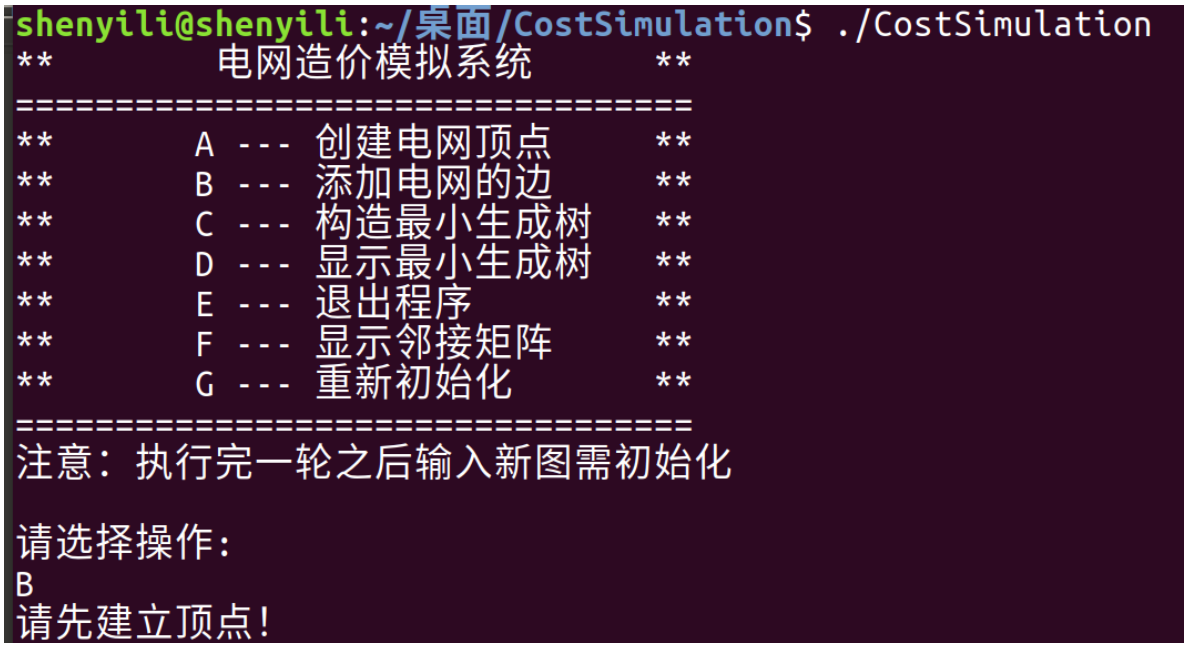
测试用例：

```
B
```

预期结果：

```
提示先建立顶点
```

实验结果：



4.2.3 MST为空

测试用例：

```
D
```

预期结果：

```
null
```

实验结果：

```
shenyili@shenyili:~/桌面/CostSimulation$ ./CostSimulatio
**          电网造价模拟系统          **
====================================
**          A --- 创建电网顶点        **
**          B --- 添加电网的边        **
**          C --- 构造最小生成树      **
**          D --- 显示最小生成树      **
**          E --- 退出程序            **
**          F --- 显示邻接矩阵        **
**          G --- 重新初始化          **
====================================
注意：执行完一轮之后输入新图需初始化

请选择操作:
D
```

4.2.4 输入自反的环

测试用例：

```
A
1
1
B
1
1 1 1
```

预期结果：

提示输入中有环存在

实验结果：

```
请选择操作:
A
请输入顶点个数:1
请依次输入顶点的名称（仅支持一个char）：
1
请选择操作:
B
请输入边的个数:（请注意，重复输入的边会覆盖此前输入的边）1
请依次输入两个顶点和边花费:
1 1 1
Insert edge failed. The cost to vertex itself should be 0. (There should be no r
ing in the graph.)
```

4.2.5 边数非法

测试用例：

```
A
1
1
B
-1
```

预期结果：

```
提示边数非法
```

实验结果：