项目说明文档

数据结构课程设计

——**电网建设造价模拟系统**

作 者 姓 名： 陈翔飞

学 号： 1851756

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 分析 4](#_Toc27681873)

[1.1 背景分析 4](#_Toc27681874)

[1.2 功能分析 4](#_Toc27681875)

[2 设计 4](#_Toc27681876)

[2.1 算法设计 4](#_Toc27681877)

[2.1 数据结构设计 5](#_Toc27681878)

[2.2 类结构设计 5](#_Toc27681879)

[2.3 成员与操作设计 5](#_Toc27681880)

[2.4 系统设计 13](#_Toc27681881)

[3 实现 14](#_Toc27681882)

[3.1 添加顶点功能的实现 14](#_Toc27681883)

[3.1.1 添加顶点功能流程图 14](#_Toc27681884)

[3.1.2 添加顶点核心代码 15](#_Toc27681885)

[3.1.3 添加顶点功能截屏示例 16](#_Toc27681886)

[3.2 添加边功能的实现 17](#_Toc27681887)

[3.2.1 添加边功能流程图 17](#_Toc27681888)

[3.2.2 添加边功能核心代码 18](#_Toc27681889)

[3.2.3 添加边功能截屏示例 19](#_Toc27681890)

[3.3 构造最小生成树功能的实现 20](#_Toc27681891)

[3.3.1 构造最小生成树功能流程图 20](#_Toc27681892)

[3.3.2 构造最小生成树功能核心代码 20](#_Toc27681893)

[3.2.3 构造最小生成树功能截屏示例 23](#_Toc27681894)

[3.4 总体系统的实现 24](#_Toc27681895)

[3.4.1 总体系统流程图 24](#_Toc27681896)

[3.4.2 总体系统核心代码 24](#_Toc27681897)

[3.6.3 总体系统截屏示例 26](#_Toc27681898)

[4 测试 26](#_Toc27681899)

[4.1 基本测试 26](#_Toc27681900)

[4.2 边界测试 30](#_Toc27681901)

[4.2.1 新建顶点数为0 30](#_Toc27681902)

[4.2.2 添加边数为0 31](#_Toc27681903)

[4.2.3 构造最小生成树时顶点数为0 31](#_Toc27681904)

[4.3 出错测试 32](#_Toc27681905)

[4.3.1 新建顶点数量出错 32](#_Toc27681906)

[4.3.2 添加边数错误 33](#_Toc27681907)

[4.3.3 最小生成树未构造时选择显示最小生成树 33](#_Toc27681908)

[4.3.4 新建顶点名称与原有顶点重复 34](#_Toc27681909)

[4.3.5 新建顶点名称序列中有重复的项 35](#_Toc27681910)

[4.3.6 构造最小生成树时输入不存在的起始顶点 36](#_Toc27681911)

[4.3.7 构造最小生成树时原图不联通 37](#_Toc27681912)

# 1 分析

## 背景分析

在电力工程管理中，造价管理作为一个重要管理环节，主要是依靠基于各种经济技术和法律于段 而制定的科学、适宜的策略，加强对电力造价的管理，以此达到有效控制电力成本的目的，从而 确保电力企业能够获得更人的经济利益。现阶段，电力造价管理因受成本上涨、融资渠道减少等 多方而因素的影响，导致其的运行和管理出现了问题。针对这些问题，应利用更加科学、有效地 控制策略，加强对电力造价成本的管理，以便确保电力企业能够从电力工程中获得更人的利益。

## 1.2 功能分析

本实验项目场景是在一个城市中有n个小区，要实现n个小区之间的电网的相互接通，构造这个城市n个小区之间的电网，使得总工程造价最低，设计一个能够满足要求的造价方案。据此可知所需的功能有：

1. 创建电网节点
2. 向电网中添加边
3. 构造最小生成树
4. 显示最小生成树
5. 清空整个电网网络

# 设计

## 算法设计

一个连通图的生成树是一个极小的连通子图，它包含图中全部的顶点（n个顶点），但只有n-1条边。

最小生成树：构造连通网的最小代价（最小权值）生成树。

可以看出，我们要求的最小电网构造就是求电路网络的最小生成树。

在这里，我们选用Prim算法求电路网络的最小生成树。

Prim算法过程为：

假设N=（V,{E}）是连通图，TE是N上最小生成树中边的集合。算法从U={u0}（u0∈V），TE={}开始，

重复执行下述操作：

在所有u∈U，v∈V-U的边（u，v）∈E中找一条代价最小的边（u0，v0）并入集合TE，同时v0 并入U，直至U=V为止。

此时TE中必有n-1条边，则T=（V，{TE}）为N的最小生成树。

## 2.1 数据结构设计

如上功能分析所述，我们需要存储小区以及小区和小区之间电网的造价。所以考虑采用图的数据结构，用顶点表示小区，用边上的权值表示小区之间电网的造价。同时考虑到处理的情况大多为稀疏图，这里还选用了堆来优化Prim算法。

## 2.2 类结构设计

为了保证设计的数据结构的泛用性，本项目选择将Heap类和Graph类都设计为模板类。然后设计了STP类来存储最小生成树，然后设计了电网类来打包生成树和总体的电网分布图，并提供给用户所需要的接口。同时，为了能通过顶点名称快速查找顶点，本项目还设计了AVL平衡树，并设计了Vector类来灵活存储数据。

## 2.3 成员与操作设计

**向量类（Vector）**

**类定义：**

1. **template**<**typename** ElementType> **class** Vector
2. {
3. **public**:
4. ~Vector<ElementType>();
5. Vector<ElementType>() = **default**;
6. Vector<ElementType>(**const** Vector<ElementType> & v);
7. Vector<ElementType>& operator = (**const** Vector<ElementType>&v);
8. **void** PushBack(**const** ElementType& t);
9. **void** PopBack();
10. **void** Clear();
11. **int** GetSize() **const**;
12. **void** ReSize(**int** NewSize);
13. ElementType& operator[](**int** Index) **const**;
14. **bool** Empty()**const**;
15. **const** ElementType& Back() **const**;
16. **private**:
17. **void** Extend();
18. **int** Size = 0;
19. **int** Capacity = 0;
20. ElementType\* Array = nullptr;
21. };

**私有成员：**

int Size;//Vector中实际储存的元素数量

int Capacity;//Vector已经申请的空间

ElementType\* Array;//储存的数据的起始地址

**私有操作：**

void Extend();

//扩容函数，当容量不足时调用

**公有操作：**

~Vector<ElementType>();

//析构函数，通过调用Clear()来释放内存

Vector<ElementType>() = default;

//默认构造函数

Vector<ElementType>(const Vector<ElementType> & v);

//拷贝构造函数

Vector<ElementType>& operator = (const Vector<ElementType>&v);

//重载=运算符，使该类支持赋值运算

void PushBack(const ElementType& t);

//向Vector末尾添加一个元素

void PopBack();

//删除末尾的元素

void Clear();

//清空Vector，释放内存

int GetSize() const;

//返回储存元素的数量

void ReSize(int NewSize);

//重设Vector的大小

ElementType& operator[](int Index) const;

//重载[]运算符，使Vector可以像数组一样使用

bool Empty()const;

//判断Vector是否为空

const ElementType& Back() const;

//返回末尾的元素

**堆类（Heap）**

**类定义：**

1. **template**<**typename** ElementType, **typename** Comparator=Greater<ElementType>> **class** Heap
2. {
3. **public**:
4. ~Heap<ElementType, Comparator>() = **default**;
5. Heap<ElementType, Comparator>() ;
6. Heap<ElementType, Comparator>(**int** Size);
7. **int** GetSize() { **return** Elements.GetSize() - 1; };
8. **bool** Empty() { **return** Elements.GetSize() <= 1; };
9. **void** Push(**const** ElementType& E);
10. **void** Build(**int** Size);
11. **void** Pop();
12. **void** Clear();
13. ElementType Top();
14. **private**:
15. Vector<ElementType> Elements;
16. };

**私有成员：**

Vector<ElementType> Elements;//向量类存储堆中实际数据

**公有操作：**

~Heap<ElementType, Comparator>() = default;

//析构函数

Heap<ElementType, Comparator>() ;

//默认构造函数

Heap<ElementType, Comparator>(int Size);

//带大小参数的初始化函数

int GetSize() { return Elements.GetSize() - 1; };

//返回堆大小

bool Empty() { return Elements.GetSize() <= 1; };

//判断堆是否为空

void Push(const ElementType& E);

//向堆中推入数据

void Build(int Size);

//重设堆的大小

void Pop();

//删除堆顶元素

void Clear();

//清空堆

ElementType Top();

//返回堆顶元素

**树结点类（TreeNode）**

**类定义：**

1. **template** <**typename** Key, **typename** Value> **class** TreeNode
2. {
3. **public**:
4. **void** MidVisit(ostream& os);
5. TreeNode<Key, Value>() = **default**;
6. TreeNode<Key, Value>(Key K, Value V);
7. ~TreeNode<Key, Value>();
8. Key ElementKey;
9. Value ElementValue;
10. TreeNode<Key, Value>\* Left = nullptr, \*Right = nullptr;
11. **int** Height = 0;
12. };

**公有成员：**

Key ElementKey；//树节点的键，作为平衡树的关键词

Value ElementValue；//树节点的值，储存需要保存的信息

TreeNode<Key,Value>\* Left；//树节点的左儿子

TreeNode<Key,Value>\* Right；//树节点的右儿子

Int Height；//树节点的高度，用于维护平衡

**公有操作：**

TreeNode<Key, Value>() = default; //默认构造函数

TreeNode<Key, Value>(Key K, Value V); //含键值对参数的构造函数

void MidVisit(ostream& os);//中序遍历该节点并输出值

**AVL树类（AVLTree）**

**类定义：**

1. **template** <**typename** Key = **int**, **typename** Value = **int**> **class** AVLTree
2. {
4. **public**:
5. AVLTree<Key, Value>();
6. AVLTree<Key, Value>(Key K, Value V);
7. ~AVLTree<Key, Value>();
8. TreeNode<Key, Value>\* Change(Key K, Value V);
9. TreeNode<Key, Value>\* Insert(Key K, Value V = Value());
10. TreeNode<Key, Value>\* Find(Key K);
11. TreeNode<Key, Value>\* Delete(Key K);
12. **void** MakeEmpty();
13. **void** DisPlay(ostream& os);
14. **int** GetHeight();
15. TreeNode<Key, Value>\* UncheckChange(TreeNode<Key, Value>\* P, Value V);
16. TreeNode<Key, Value>\* UncheckInsert(Key K, Value V);
17. **int** GetSize();
18. **private**:
19. **int**  Max(**int** x, **int** y);
20. **int** GetHeight(TreeNode<Key, Value>\* P);
21. TreeNode<Key, Value>\* SingleRotateWithLeft(TreeNode<Key, Value>\* K2);
22. TreeNode<Key, Value>\* SingleRotateWithRight(TreeNode<Key, Value>\* K2);
23. TreeNode<Key, Value>\* DoubleRotateWithLeft(TreeNode<Key, Value>\* K3);
24. TreeNode<Key, Value>\* DoubleRotateWithRight(TreeNode<Key, Value>\* K3);
25. TreeNode<Key, Value>\* Find(Key K, TreeNode<Key, Value>\* P);
26. TreeNode<Key, Value>\* FindMin(TreeNode<Key, Value>\* P);
27. TreeNode<Key, Value>\* FindMax(TreeNode<Key, Value>\* P);
28. TreeNode<Key, Value>\* Insert(Key K, Value V, TreeNode<Key, Value>\* P);
29. TreeNode<Key, Value>\* Delete(Key K, TreeNode<Key, Value>\* P);
30. TreeNode<Key, Value>\* MakeEmpty(TreeNode<Key, Value>\* P);
31. **int** Size = 0;
32. TreeNode<Key, Value>\* Root = nullptr;
33. };

**私有成员：**

int Size = 0;//AVL树的大小，即储存键值对数

TreeNode<Key, Value>\* Root ;//AVL树的树根

**私有操作：**

int Max(int x, int y); //返回x，y中的最大值

int GetHeight(TreeNode<Key, Value>\* P); //获取节点P的高度

TreeNode<Key, Value>\* SingleRotateWithLeft(TreeNode<Key, Value>\* K2);

//左单旋，用于维护平衡

TreeNode<Key, Value>\* SingleRotateWithRight(TreeNode<Key, Value>\* K2);

//右单旋，用于维护平衡

TreeNode<Key, Value>\* DoubleRotateWithLeft(TreeNode<Key, Value>\* K3);

//左双旋，用于维护平衡

TreeNode<Key, Value>\* DoubleRotateWithRight(TreeNode<Key, Value>\* K3);

//右双旋，用于维护平衡

TreeNode<Key, Value>\* Find(Key K, TreeNode<Key, Value>\* P);

//以节点P为根，查找有关键词K的节点

TreeNode<Key, Value>\* FindMin(TreeNode<Key, Value>\* P);

//以节点P为根，查找最小节点

TreeNode<Key, Value>\* FindMax(TreeNode<Key, Value>\* P);

//以节点P为根，查找最大节点

TreeNode<Key, Value>\* Insert(Key K, Value V, TreeNode<Key, Value>\* P);

//以节点P为根，插入键值对<K,V>

TreeNode<Key, Value>\* Delete(Key K, TreeNode<Key, Value>\* P);

//以节点P为根，删除有关键词K的节点

TreeNode<Key, Value>\* MakeEmpty(TreeNode<Key, Value>\* P);

//清空节点P以及P的所有子节点并释放内存

**公有操作：**

AVLTree<Key, Value>();

//AVLTree的默认构造函数

AVLTree<Key, Value>(Key K, Value V);

//AVLTree的带键值对参数的构造函数，以参数中的键值对开辟一个树节点并使Root指向该节点

~AVLTree<Key, Value>();

//AVLTree的析构函数，通过调用MakeEmpty()实现对内存的回收

TreeNode<Key, Value>\* Change(Key K, Value V);

//将树中以K为键的节点的值改为V，如果不存在就插入键值对<K,V>

TreeNode<Key, Value>\* Insert(Key K, Value V = Value());

//将键值对<K,V>插入树中，如果键K已经存在则将对应节点中的值改为V

TreeNode<Key, Value>\* Find(Key K);

//返回键为K的树节点指针

TreeNode<Key, Value>\* Delete(Key K);

//删除键为K的树节点

void MakeEmpty();

//清空整个树并释放内存

void DisPlay(ostream& os);

//中序展示整棵树的所有树节点

int GetHeight();

//返回树的高度

TreeNode<Key, Value>\* UncheckChange(TreeNode<Key, Value>\* P, Value V);

//直接将节点P的值改为V

TreeNode<Key, Value>\* UncheckInsert(Key K, Value V);

//不检查K是否存在于树中，直接插入键值对<K,V>，如果插入失败则返回空

int GetSize();

//返回已经储存的键值对数量

**顶点类（Vertex）**

**类定义：**

1. **template**<**typename** Tag>
2. **class** Vertex
3. {
4. **public**:
5. Vertex<Tag>() = **default**;
6. Vertex<Tag>(**const** Tag& T) : VertexTag(T) {};
7. **int** Index;
8. Tag VertexTag;
9. Vector<Vertex\*> Adjacency;
10. Vector<**int**> Weight;
11. };

**公有成员：**

int Index;

//顶点的数字编号

Tag VertexTag;

//顶点的特定标记，一般为名字

Vector<Vertex\*> Adjacency;

//顶点的邻接顶点

Vector<int> Weight;

//边的权值

**公有操作：**

Vertex<Tag>() = default;

//默认构造函数

Vertex<Tag>(const Tag& T) : VertexTag(T) {};

//带标记参数的构造函数

**图类（Graph）**

**类定义：**

1. **template**<**typename** Tag> **class** Graph
2. {
3. **public**:
4. Graph<Tag>() = **default**;
5. ~Graph<Tag>() ;
6. **void** Clear();
7. Vertex<Tag>\* AddVertex(**const** Tag& T);
8. **void** Link(**const** Tag&T1, **const** Tag&T2, **int** Weight);
9. **void** Link(Vertex<Tag>\*V1, Vertex<Tag>\*V2, **int** Weight);
10. **void** SingleLink(Vertex<Tag>\*V1, Vertex<Tag>\*V2, **int** Weight);
11. Vector<Vertex<Tag>\*> Vertexes;
12. AVLTree<Tag, **int**> VertexTree;
13. };

**公有成员：**

Vector<Vertex<Tag>\*> Vertexes;//图中所有顶点

AVLTree<Tag, int> VertexTree;//通过标记查找顶点的数字编号

**公有操作：**

Graph<Tag>() = default;

//默认构造函数

~Graph<Tag>();

//析构函数，通过调用Clear（）来回收内存

void Clear();

//清空图

Vertex<Tag>\* AddVertex(const Tag& T);

//向图内添加顶点

void Link(const Tag&T1, const Tag&T2, int Weight);

//通过标记双向连接两个顶点

void Link(Vertex<Tag>\*V1, Vertex<Tag>\*V2, int Weight);

//通过顶点指针双向连接两个顶点

void SingleLink(Vertex<Tag>\*V1, Vertex<Tag>\*V2, int Weight);

//通过顶点指针单向连接两个顶点

**生成树类（STP）**

**类定义：**

1. **class** STP
2. {
3. **public**:
4. **void** Display();
5. **void** DFS(Vertex<string> \*Cur, **int**&Cnt);
6. Vertex<string>\* Root = nullptr;
7. Graph<string> G;
8. };

**公有成员：**

Vertex<string>\* Root = nullptr;

//生成树的根

Graph<string> G;

//生成树网络

**公有操作：**

void Display();

//从根开始展示整颗生成树

void DFS(Vertex<string> \*Cur, int&Cnt);

//深度优先搜索生成树，并打印边，通过Cnt来格式化输出

**电网类（PowerGrid）**

**类定义：**

1. **class** PowerGrid
2. {
3. **public**:
4. **void** Menu();
5. **void** CreatNetVertex();
6. **void** AddEdge();
7. **void** BuildSTP();
8. **void** DisplaySTP();
9. **void** Destory();
10. **private**:
11. **bool** STPIsReady = **false**;
12. **bool** CheckName(**const** Vector<string>& AllName);
13. Graph<string> Net;
14. STP NetSTP;
16. };

**私有成员：**

bool STPIsReady = false;//标记电网生成树是否构造完成

Graph<string> Net;//用图储存的电路网络

STP NetSTP;//电路网络的最小生成树

**私有操作：**

bool CheckName(const Vector<string>& AllName);

//用于检查新增顶点名字是否合法

**公有操作：**

void Menu();

//展示菜单

void CreatNetVertex();

//新建顶点

void AddEdge();

//向网络中添加边

void BuildSTP();

//构造最小生成树

void DisplaySTP();

//展示最小生成树

void Destory();

//摧毁整个网络

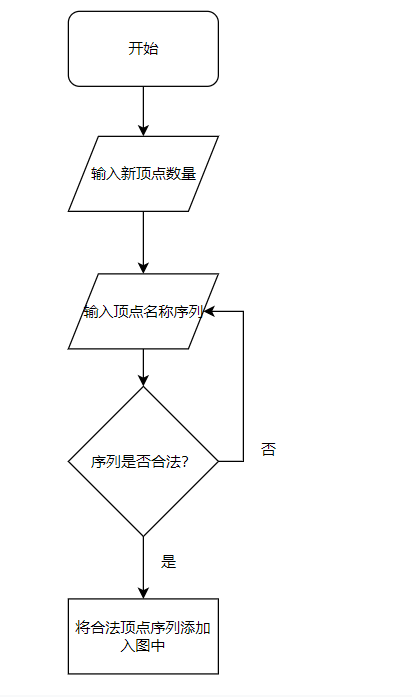
## 2.4 系统设计

系统首先调用Menu()来展示菜单， 然后根据用户所输入的操作码（Cmd）执行电网对应的成员函数。

# 3 实现

## 3.1 添加顶点功能的实现

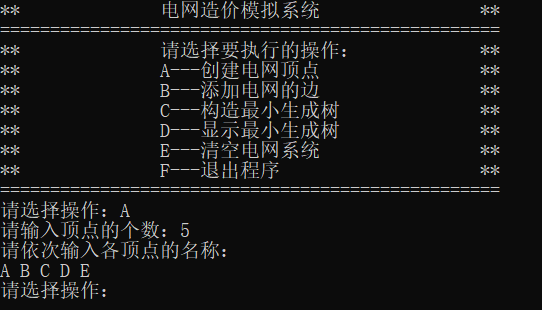
### 3.1.1 添加顶点功能流程图



### 3.1.2 添加顶点核心代码

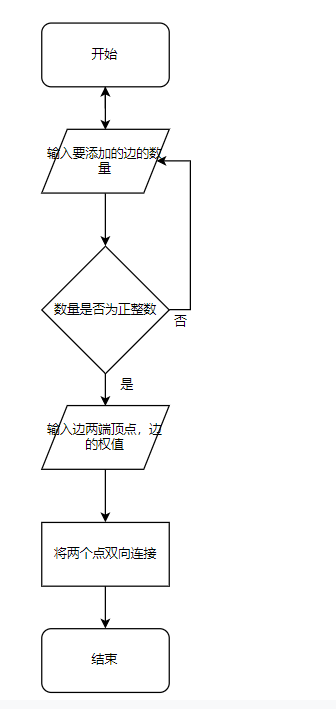
1. **void** PowerGrid::CreatNetVertex()
2. {
3. STPIsReady = **false**;
4. **int** n;
5. cout << "请输入顶点的个数：";
6. cin >> n;
7. **while** (n <= 0)
8. {
9. cout << "顶点个数为负数或0！请重新输入一个正整数：";
10. cin >> n;
11. }
12. Vector<string> AllName;
13. AllName.ReSize(n);
14. cout << "请依次输入各顶点的名称：" << endl;
15. **int** i;
16. **for** (i = 0; i < n; ++i)
17. {
18. cin >> AllName[i];
19. }
20. **while** (CheckName(AllName) == **false**)
21. {
22. cout << "请重新输入顶点名称序列：" << endl;
23. **for** (i = 0; i < n; ++i)
24. {
25. cin >> AllName[i];
26. }
27. }
28. **for** (i = 0; i < n; ++i)
29. {
30. Net.AddVertex(AllName[i]);
31. }
32. }

### 3.1.3 添加顶点功能截屏示例



## 3.2 添加边功能的实现

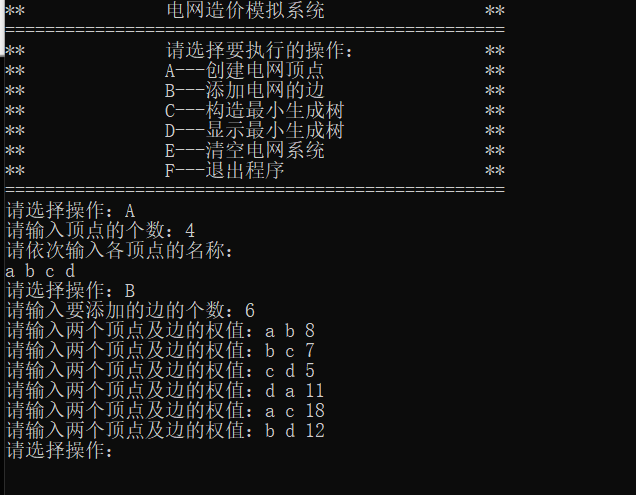
### 3.2.1 添加边功能流程图



### 3.2.2 添加边功能核心代码

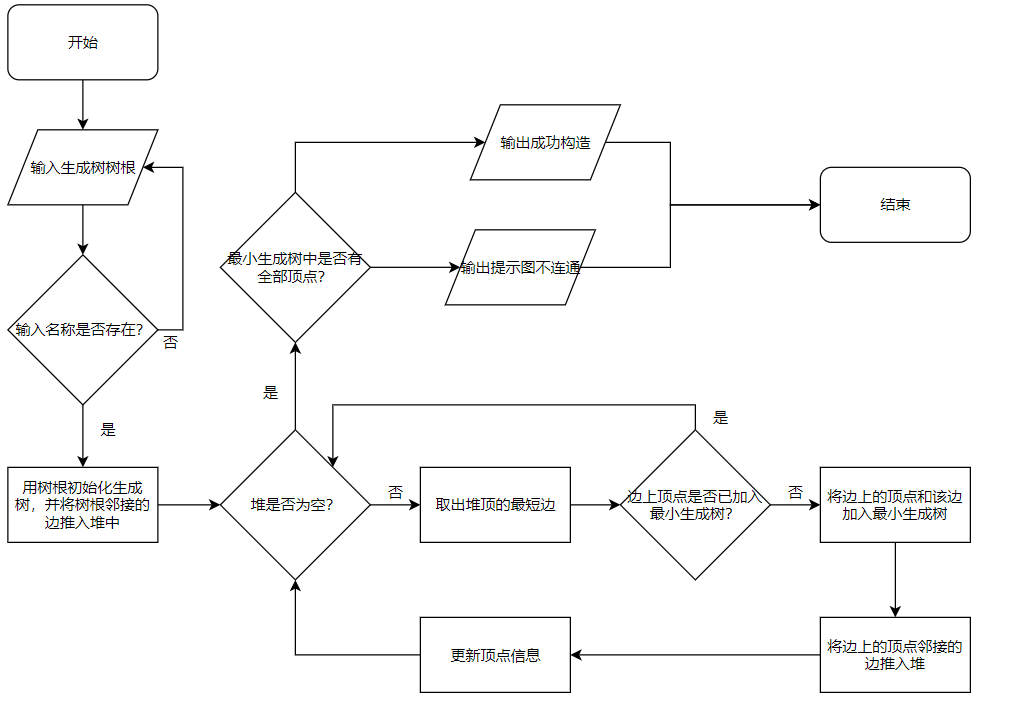
1. **void** PowerGrid::AddEdge()
2. {
3. STPIsReady = **false**;
4. **int** n, i;
5. cout << "请输入要添加的边的个数：";
6. cin >> n;
7. **while** (n <= 0)
8. {
9. cout << "顶点个数为负数或0！请重新输入一个正整数：" ;
10. cin >> n;
11. }
12. string S1, S2;
13. **int** Weight;
14. **for** (i = 0; i < n; ++i)
15. {
16. cout << "请输入两个顶点及边的权值：" ;
17. cin >> S1 >> S2 >> Weight;
18. Net.Link(S1, S2, Weight);
19. }
20. }

### 3.2.3 添加边功能截屏示例



## 3.3 构造最小生成树功能的实现

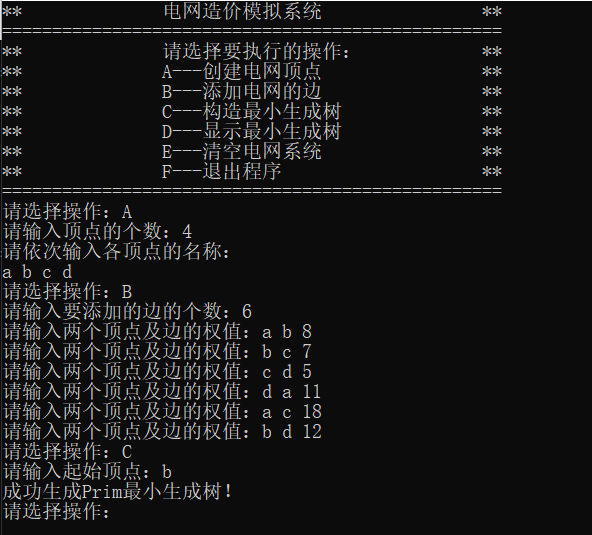
### 3.3.1 构造最小生成树功能流程图



### 3.3.2 构造最小生成树功能核心代码

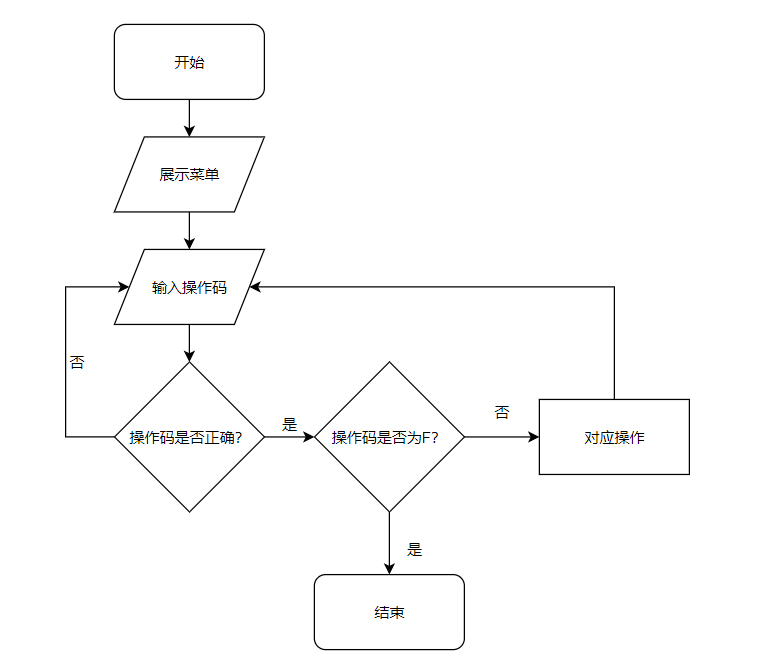
1. **void** PowerGrid::BuildSTP()
2. {
4. **struct** VertexInfo
5. {
6. **bool** IsJoined = **false**;
7. **int** Weight= 0x7fffffff;
8. **int** PreV;
9. VertexInfo(**int** W, **int** s,**bool** b=**false**) :Weight(W), PreV(s),IsJoined(b) {};
10. **const** VertexInfo& operator =(**const** VertexInfo& VI)
11. {
12. Weight = VI.Weight;
13. IsJoined = VI.IsJoined;
14. PreV = VI.PreV;
15. **return** \***this**;
16. }
17. VertexInfo() = **default**;
18. };
19. **struct** Edge
20. {
21. Edge(**int** T1, **int** T2, **int** W) : VT1(T1), VT2(T2), Weight(W) {};
22. Edge() = **default**;
23. **bool** operator <(**const** Edge& E) **const** { **return** Weight < E.Weight; };
24. **int** VT1, VT2;
25. **int** Weight;
26. };
27. **if** (Net.Vertexes.Empty())
28. {
29. cout << "网络中不存在顶点，请至少创建一个顶点！" << endl;
30. **return**;
31. }
32. string Name;
33. **int** VertexNum = Net.Vertexes.GetSize();
34. cout << "请输入起始顶点：";
35. cin >> Name;
36. TreeNode<string, **int**>\* Start;
37. **while** ((Start=Net.VertexTree.Find(Name)) == nullptr)
38. {
39. cout << "该顶点不存在！请重新输入：";
40. cin >> Name;
41. }
42. **if** (STPIsReady == **true**&&NetSTP.Root->VertexTag==Name) **return**;
43. NetSTP.G.Clear();
44. NetSTP.Root = nullptr;
45. **int** i;
46. **int** StartIndex = Start->ElementValue;
47. NetSTP.Root=NetSTP.G.AddVertex(Name);
48. Vector<VertexInfo> Info;
49. Info.ReSize(VertexNum);
50. Info[StartIndex].IsJoined = **true**;
51. Info[StartIndex].Weight = 0;
52. Heap<Edge> MinEdgeH;
53. **for** (i = 0; i < Net.Vertexes[StartIndex]->Adjacency.GetSize(); i++)
54. {
55. Vertex<string> \*t= Net.Vertexes[StartIndex]->Adjacency[i];
56. **int** NextVertex = Net.Vertexes[StartIndex]->Adjacency[i]->Index;
57. **int** NextWeight= Net.Vertexes[StartIndex]->Weight[i];
58. MinEdgeH.Push(Edge(StartIndex,NextVertex,NextWeight));
59. Info[NextVertex].PreV = 0;
60. Info[NextVertex].Weight = NextWeight;
61. }
62. **while** (!MinEdgeH.Empty())
63. {
64. **int** Cur = MinEdgeH.Top().VT2; MinEdgeH.Pop();
65. **if** (Info[Cur].IsJoined == **true**) **continue**;
66. string CurName = Net.Vertexes[Cur]->VertexTag;
67. auto NewNode=NetSTP.G.AddVertex(CurName);
68. NetSTP.G.SingleLink(NetSTP.G.Vertexes[Info[Cur].PreV], NewNode, Info[Cur].Weight);
69. Info[Cur].IsJoined = **true**;
70. **for** (i = 0; i < Net.Vertexes[Cur]->Adjacency.GetSize(); i++)
71. {
72. **int** NextVertex = Net.Vertexes[Cur]->Adjacency[i]->Index;
73. **int** NextWeight = Net.Vertexes[Cur]->Weight[i];
74. **if** (Info[NextVertex].IsJoined == **true**) **continue**;
75. **if** (Info[NextVertex].Weight > NextWeight)
76. {
77. Info[NextVertex].PreV = NetSTP.G.Vertexes.GetSize() - 1;
78. Info[NextVertex].Weight = NextWeight;
79. MinEdgeH.Push(Edge(Cur, NextVertex, NextWeight));
80. }
81. }
82. }
83. **if** (NetSTP.G.Vertexes.GetSize() < Net.Vertexes.GetSize())
84. {
85. cout << "原网络不连通！请重建网络或者添加上其他边!" << endl;
86. **return**;
87. }
88. cout << "成功生成Prim最小生成树！" << endl;
89. STPIsReady = **true**;
90. }

### 3.2.3 构造最小生成树功能截屏示例



## 3.4 总体系统的实现

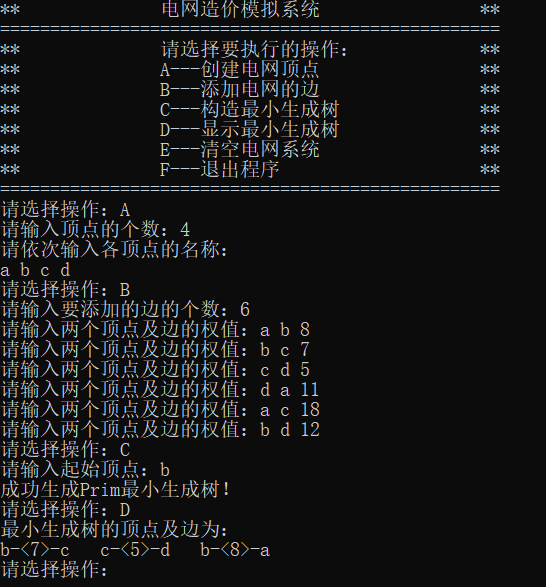
### 3.4.1 总体系统流程图



### 3.4.2 总体系统核心代码

1. **int** main(**void**)
2. {
3. PowerGrid MyPowerGrid;
4. MyPowerGrid.Menu();
5. string Cmd;
6. **bool** IsLooping = **true**;
7. **while** (IsLooping)
8. {
9. cout << "请选择操作：";
10. cin >> Cmd;
11. **switch** (Cmd[0])
12. {
13. **case** 'A':
14. MyPowerGrid.CreatNetVertex();
15. **break**;
16. **case** 'B':
17. MyPowerGrid.AddEdge();
18. **break**;
19. **case** 'C':
20. MyPowerGrid.BuildSTP();
21. **break**;
22. **case** 'D':
23. MyPowerGrid.DisplaySTP();
24. **break**;
25. **case** 'E':
26. MyPowerGrid.Destory();
27. **break**;
28. **case** 'F':
29. IsLooping = **false**;
30. **break**;
31. **default**:
32. cout << "未知命令！" << endl;
33. **break**;
34. }
35. }
36. **return** 0;
37. }

### 3.6.3 总体系统截屏示例



# 4 测试

## 4.1 基本测试

**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

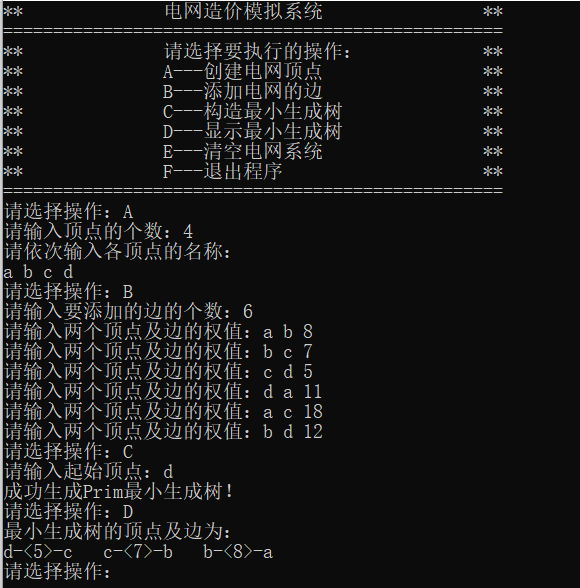
d

D

**预期结果**：

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

**实验结果：**



**测试用例**：

A

15

a b c d e f g h i j k l m n o

B

30

a b 2

a c 7

a d 5

a f 13

a i 12

b c 3

b e 9

b k 11

b l 5

b o 4

c h 7

c g 3

c j 14

f m 11

g n 4

g h 7

g o 5

h k 10

h n 2

h j 8

h m 15

h o 3

i j 5

i k 6

i o 20

j m 7

j l 13

m n 3

m o 1

n o 19

C

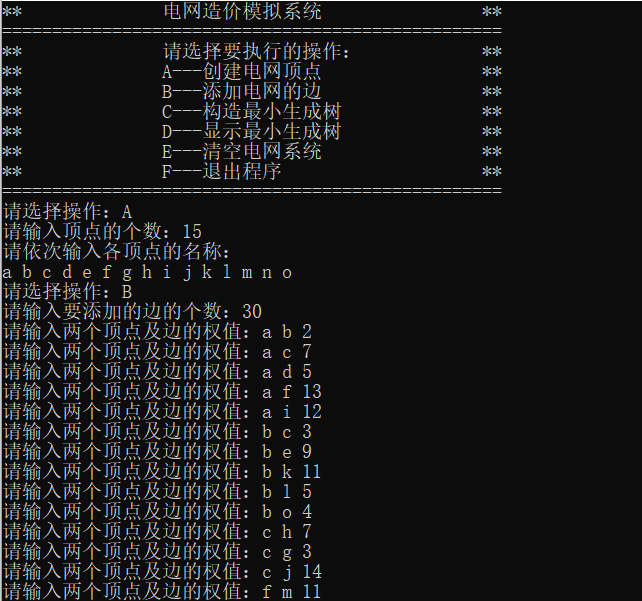
i

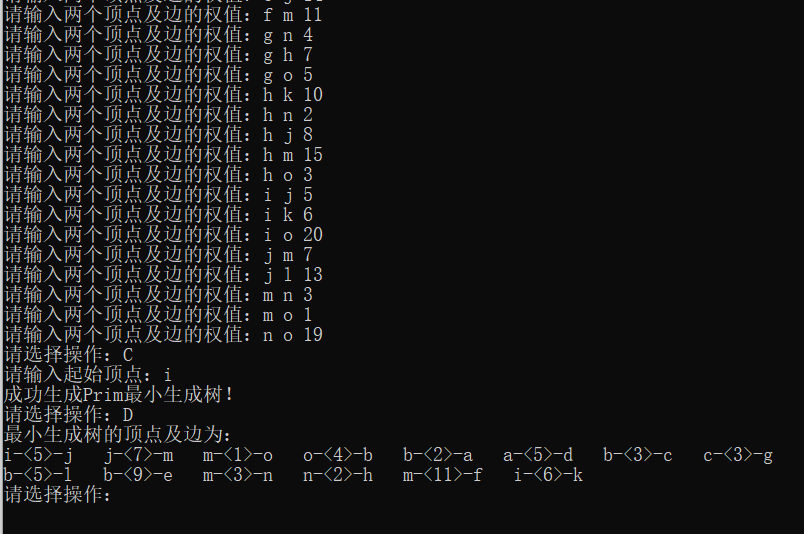
D

**预期结果**：

i-<5>-j j-<7>-m m-<1>-o o-<4>-b b-<2>-a a-<5>-d b-<3>-c c-<3>-g b-<5>-l b-<9>-e m-<3>-n n-<2>-h m-<11>-f i-<6>-k

**实验结果：**





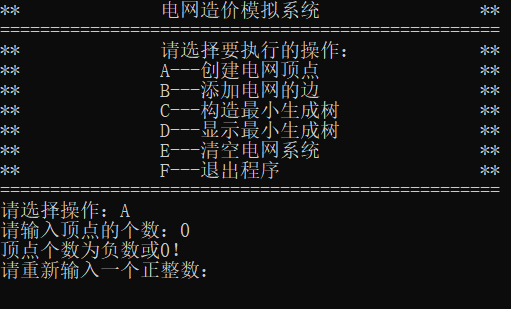
## 4.2 边界测试

### 4.2.1 新建顶点数为0

**测试用例：**新建顶点数为0

**预期结果：**给出错误提示，程序运行正常不崩溃。

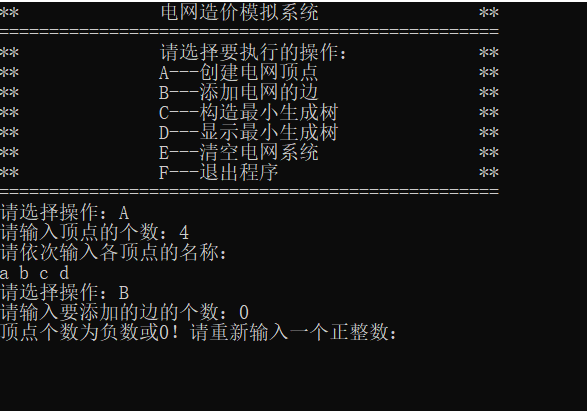
**实验结果：**



### 4.2.2 添加边数为0

**测试用例：**添加边数为0

**预期结果：**给出错误提示，程序运行正常不崩溃。

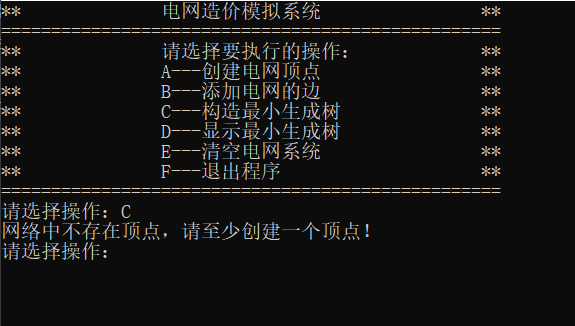
**实验结果：**

### 4.2.3 构造最小生成树时顶点数为0

**测试用例：**构造最小生成树时顶点数为0

**预期结果：**给出错误提示，程序运行正常不崩溃。

**实验结果：**



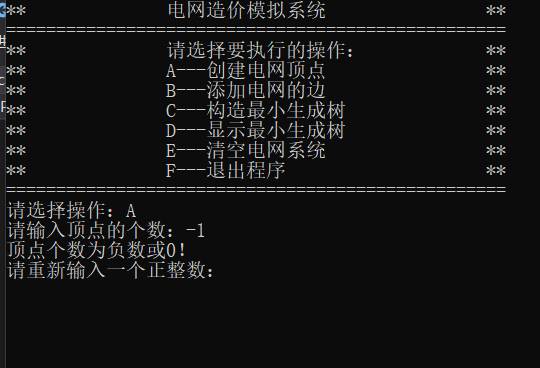
## 4.3 出错测试

### 4.3.1 新建顶点数量出错

**测试用例：**新建-1个顶点

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

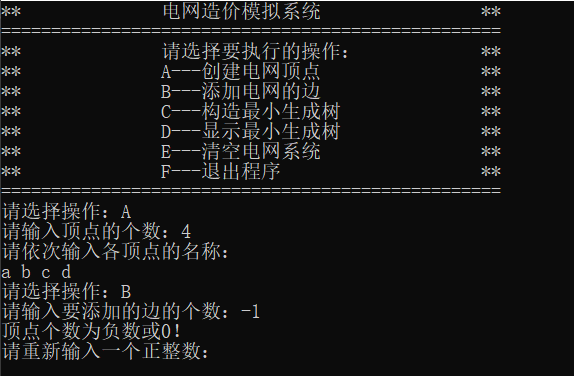


### 4.3.2 添加边数错误

**测试用例：**添加-1条边

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

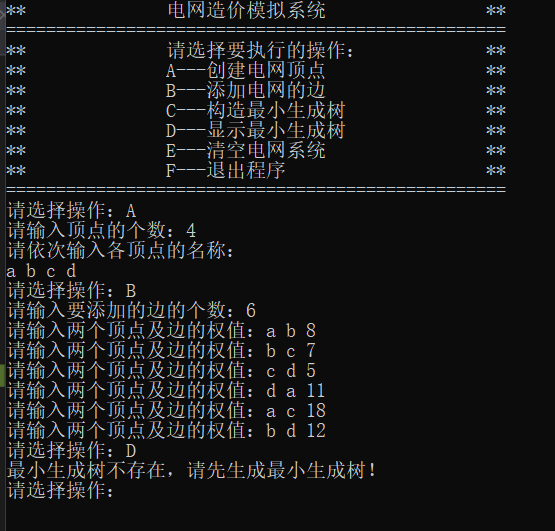


### 4.3.3 最小生成树未构造时选择显示最小生成树

**测试用例：**最小生成树未构造时选择显示最小生成树

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

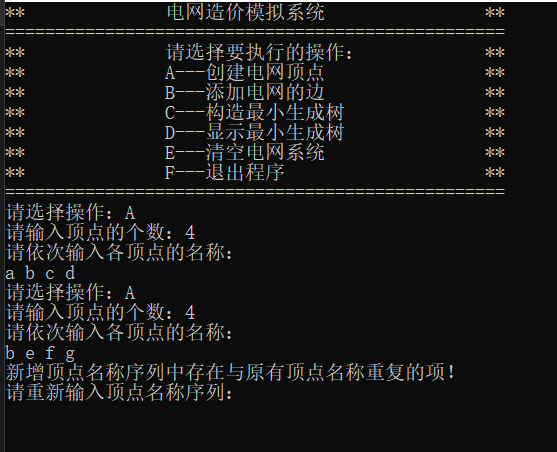


### 4.3.4 新建顶点名称与原有顶点重复

**测试用例：**新建顶点名称与原有顶点重复

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

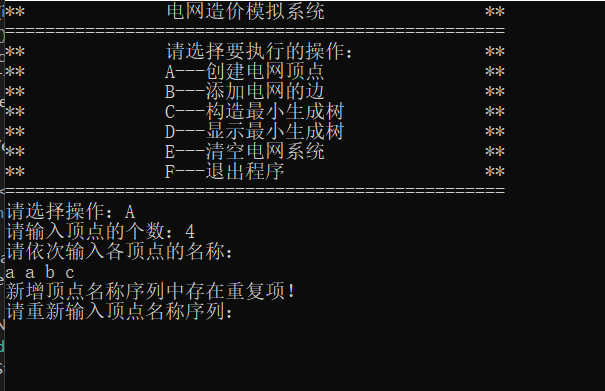


### 4.3.5 新建顶点名称序列中有重复的项

**测试用例：**新建顶点名称序列中有重复的项

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

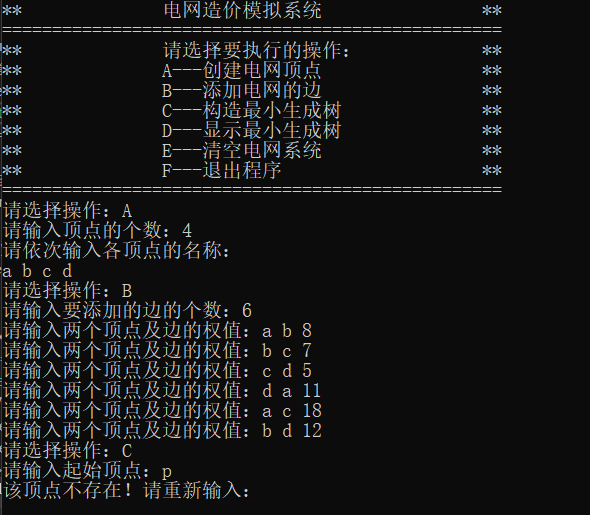


### 4.3.6 构造最小生成树时输入不存在的起始顶点

**测试用例：**构造最小生成树时输入不存在的起始顶点

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**



### 4.3.7 构造最小生成树时原图不联通

**测试用例：**

A

4

a b c d

B

2

a b 8

c d 5

C

d

D

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

