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： 杨滕超

学 号： 2151298

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 项目概述](#_Toc495668153) 3

[1.1 项目背景](#_Toc495668154) 3

[1.2 项目功能](#_Toc495668155) 3

[1.3 项目分析](#_Toc495668154) 4

[2 项目设计](#_Toc495668156) 4

[2.1 类设计](#_Toc495668158) 4

[2.1.1 String类](#_Toc495668163) 4

[2.1.2 Vector类](#_Toc495668163) 5

[2.1.3 Vector游标类](#_Toc495668163) 6

[2.1.4 Heap类](#_Toc495668163) 7

[2.1.5 Priority\_Queue类](#_Toc495668163) 7

[2.1.6 并查集类](#_Toc495668163) 8

[2.1.7 边结点类](#_Toc495668163) 9

[2.2 总体设计](#_Toc495668158) 10

[3 项目实现](#_Toc495668161) 11

[3.1 总体实现](#_Toc495668162) 11

[3.1.1 总体实现流程图](#_Toc495668167) 11

[3.1.2 总体实现思路](#_Toc495668167) 11

[3.1.3 总体实现主要代码](#_Toc495668167) 12

[3.2 输入顶点名称实现 1](#_Toc495668166)3

[3.2.1 输入顶点名称实现流程图 1](#_Toc495668167)3

[3.2.2 输入顶点名称实现思路](#_Toc495668167) 13

[3.2.3 输入顶点名称实现主要代码](#_Toc495668168) 14

[3.3 输入添加电网的边实现](#_Toc495668162) 15

[3.3.1 输入添加电网的边实现流程图](#_Toc495668167) 15

[3.3.2 输入添加电网的边实现思路](#_Toc495668167) 16

[3.3.3 输入添加电网的边实现主要代码](#_Toc495668167) 16

[3.4 生成最小生成树实现](#_Toc495668162) 17

[3.4.1 生成最小生成树实现思路](#_Toc495668167) 17

[3.4.2 生成最小生成树实现主要代码](#_Toc495668167) 17

[3.4.3 Prim算法生成最小树流程图](#_Toc495668167) 18

[3.4.4 Prim算法生成最小树思路](#_Toc495668167) 18

[3.4.5 Prim算法生成最小树主要代码](#_Toc495668167) 19

[3.4.6 Kruskal算法生成最小树流程图](#_Toc495668167) 20

[3.4.7 Kruskal算法生成最小树思路](#_Toc495668167) 20

[3.4.8 Kruskal算法生成最小树主要代码](#_Toc495668167) 21

[3.5 展示生成最小树的边实现](#_Toc495668162) 21

[3.5.1 展示生成最小树的边实现思路](#_Toc495668167) 21

[3.5.2 展示生成最小树的边实现主要代码](#_Toc495668167) 21

[4 项目测试](#_Toc495668161) 22

[4.1 输入顶点名称测试](#_Toc495668174) 22

[4.1.1 正确输入](#_Toc495668167) 22

[4.2 添加电网边测试](#_Toc495668174) 22

[4.2.1 正确输入](#_Toc495668167) 22

[4.2.2 输入不存在的顶点名称](#_Toc495668167) 23

[4.2.3 输入相同顶点名称](#_Toc495668167) 23

[4.2.4 未建立顶点名称](#_Toc495668167) 23

[4.3 最小生成树测试](#_Toc495668174) 24

[4.3.1 输入不存在顶点并正确输入](#_Toc495668167) 24

[4.3.2 未建立顶点名称](#_Toc495668167) 24

[4.3.3 未添加边](#_Toc495668167) 24

[4.3.4 已添加边但不连通](#_Toc495668167) 25

[4.4 展示最小生成树顶点和边测试](#_Toc495668174) 25

[4.4.1 正确输入](#_Toc495668167) 25

[4.4.2 为建立顶点名称](#_Toc495668167) 26

[4.4.3 未添加边](#_Toc495668167) 26

[4.4.4 未生成最小生成树](#_Toc495668167) 26

[4.5 第二轮电网建设测试](#_Toc495668174) 27

[4.6 有多个平行边的图](#_Toc495668174) 28

1 项目概述

* 1. 项目背景

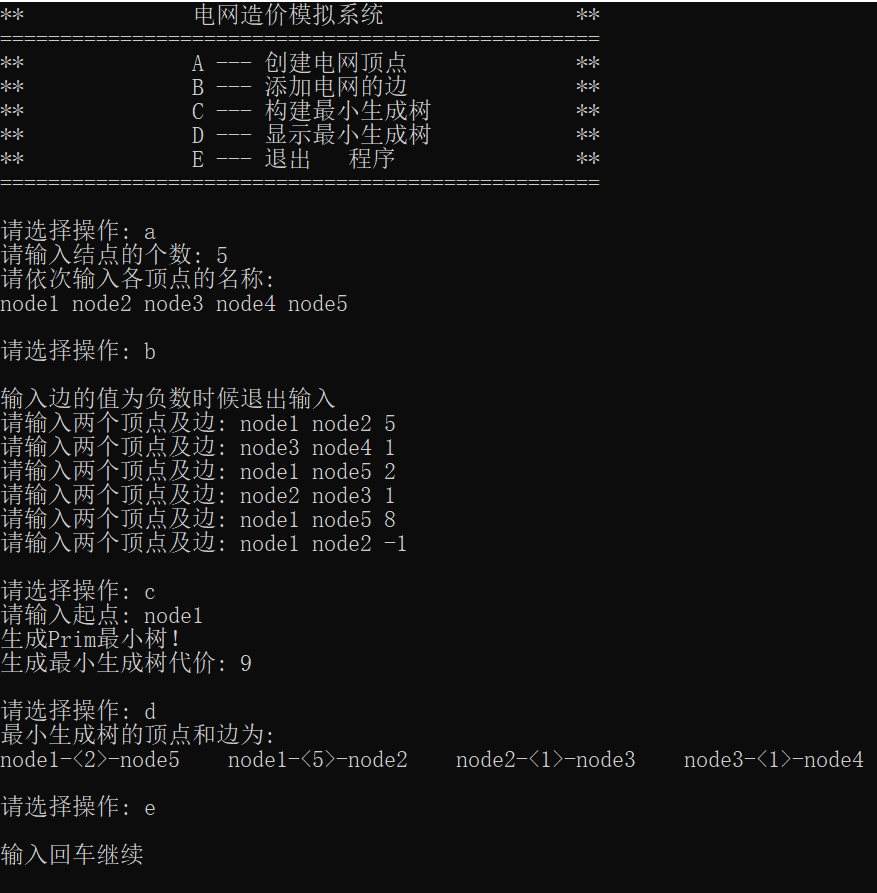
对于一个城市的n个小区，要实现最小成本地将n个小区相连，这是一个在城市建设中十分常见的问题。将其抽象为顶点与边，其实质就是需要我们解决图论中的最小生成树问题——即如何将所有顶点连接的同时使得边的权值最小，并且在连接生成的途中不存在回路——满足树的要求。将此问题发散，最小生成树问题又可以对应许许多多实际的问题，例如：修建铁路、构建航线等等。

完成本项目，对于解决其他许多实际问题具有重要意义，帮助我们在图的学习中得到更深的体会。在本项目中，我们采用Prim算法与Kruskal算法实现，并体会两种算法的异同由此进行比较与分析。

* 1. 项目功能

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。输出提示，请用户输入字母选择对应操作，依次构建电网的顶点与边，然后生成最小生成树。在上述过程中，需要注意的是若上一个步骤没有执行，当前步骤即使选择了也不会运行，这要求我们输出相应提示，帮助用户更好地使用本程序。

示例如下：



* 1. 项目分析

1. 对于功能完整性，在完成题目要求的基本功能之外，本项目不仅提供Prim算法的解答，并且提供Kruskal算法的实现，由此比较两种方法的异同，并在后文进行详细分析。
2. 对于用户友好性与健壮性，本项目提供丰富的提示，首先是提示用户关于如何操作本程序，其次是在用户输入有误的情况下，输出输入错误的提示，由此保证程序不至于崩溃，并请用户重新输入，保证程序的顺利运行。
3. 适宜的数据结构的考虑，对于存图的方式主要分为邻接矩阵与邻接表两种方式，杜宇不同的顶点个数与边的个数的情况，两者具有不同的优势。当图为稠密图时，利用邻接矩阵存储并对其进行访问有着比邻接表更好的性能；但对于稀疏图若仍然使用邻接矩阵未免有些过于浪费空间了，当顶点数量n很大，并且是稀疏图时，n2是远远大于边的个数，由此邻接矩阵中存在相当大的空间未被利用，因此，邻接表在这种情况下发挥着更好的性能。
4. 输入重复边的考虑。对于本项目要求，输入的边应该是一条无向边，意味着在途中需要存储两。若是在邻接矩阵中，可以通过下标快速寻找到是否已经存在，或者更新为最小值。但在邻接表中，寻找到某一条边在极端边界情况下，即该边在对应链表的最后一个结点，并且对应链表已经具有n-1条边，需要O(n)的时间复杂度。这也是一个值得讨论的问题。考虑到日常生活中虽然出现该情况的几率不大，但对程序的优化仍然是我们需要考虑的一个重要问题。

2 项目设计

2.1 类设计

2.1.1 String类

字符串作为使用频率较高的一种数据结构，在程序发挥着巨大的作用。C++string类中配套的多种函数方便我们对字符串的处理，例如：比较、连接、替换、搜索等等。将其完全实现实现任务艰巨。因此我们实现其中的主要功能，以完成对主要的程序的要求。

该类主要成员函数如下：

·//空构造函数String();

·//赋值构造函数String(const String& str);

·//字符串构造函数String(const char\* str);

·//大小为size的字符串构造函数String(const char\* str, int size);

·//字符串长度int length()const;

·//重载=，StirngString& operator=(const String& str);

·//重载=，字符串String& operator=(const char\* str);

·//重载=，一个字符 String& operator=(const char& ch);

·//重载比较函数friend int strCmp(const String& str1, const String& str2);

·//重载< > <= >= friend bool operator<(const String& str1, const String& str2);

friend bool operator>(const String& str1, const String& str2);

friend bool operator<=(const String& str1, const String& str2);

friend bool operator>=(const String& str1, const String& str2);

·//重载[]inline char& operator[](const int index);

·//重载== bool operator==(const String& str);

bool operator==(const char\* str);

·//重载+ String operator+(const String& str);

String operator+(const char\* str);

·//重载+= String& operator+=(const String& str);

String& operator+=(const char\* str);

String& operator+=(const char ch);

·//返回字符串类型char\* c\_str();

·//模式匹配 int find(const char\* str);

int find(String& str);

·//删除所有的某个字符void erase(char ch);

·//翻转void reverse();

·//清空void clear();

·//删掉最后一个void pop\_back();

·//获得最后一个字符char back() const;

2.1.2 Vector类

向量作为存储同一种类型数据的一维数组，相当于是数组的扩展，在其基础上增加方便程序员工作的操作。其中的优点在于：对于查找某个位置的元素的时间复杂度为O(1)，但是它的缺点也十分明显：向量的删除与插入的时间代价是巨大的，时间复杂度基本是O(n)级别；但特殊的是，在最后插入元素时，时间复杂度为O(1)。

我们设计Vector类的同时，同样设计了它的迭代器。

主要成员函数如下：

·//构造函数

Vector();

Vector(int size);

Vector(const Vector<T>& V);

Vector(int size, const T& val);

·//清空Vector

void clear();

·//最后添加元素

void emplace\_back(const T& x);

·//最后删除元素

void pop\_back();

·//返回最后元素

const T back() const ;

T& back() ;

·//返回开头元素

const T front() const ;

T& front() ;

·//插入

void insert(const Vector<T>::vector\_iterator& it, const T& x);

·//删除

void remove(const Vector<T>::vector\_iterator& it);

·//迭代器begin

inline Vector<T>::vector\_iterator begin() ;

·//迭代器end

inline Vector<T>::vector\_iterator end() ;

·//返回迭代器的查找

typename Vector<T>::vector\_iterator find(const T& x);

·//重载[]

T& operator[](const int index) ;

·//重载=

Vector<T>& operator=(const Vector<T>& V);

·//重新设置大小

void resize(int size);

void resize(int size, const T& val);

2.1.3 Vector游标类

其中数据成员包含了指向Vector中数组元素的指针，并实现的重载\*、->、++、--、<、>等操作符。

主要函数如下：

·//重载\*

T& operator\*();

·//重载->

T\* operator->();

·//重载==

inline bool operator==(const vector\_iterator& it) const ;

·//重载!=

inline bool operator!=(const vector\_iterator& it) const;

·//重载不等号

inline bool operator<(const vector\_iterator& it) const;

inline bool operator>(const vector\_iterator& it) const ;

inline bool operator<=(const vector\_iterator& it) const;

inline bool operator>=(const vector\_iterator& it) const ;

·//重载自加

inline Vector<T>::vector\_iterator& operator++();

·//重载自减

inline Vector<T>::vector\_iterator& operator--();

·//后置自加

Vector<T>::vector\_iterator operator++(int);

·//后置自减

Vector<T>::vector\_iterator operator--(int);

·//与数字相加

friend Vector<T>::vector\_iterator operator+(const vector\_iterator& it,int num);

·//与数字相减

friend Vector<T>::vector\_iterator operator-(const vector\_iterator& it,int num);

·//自加

Vector<T>::vector\_iterator operator+=(int num);

·//自减

Vector<T>::vector\_iterator operator-=(int num);

·//两个相减

friend int operator-(const vector\_iterator& it1, const vector\_iterator& it2);

2.1.4 Heap类

堆是一种树形结构，是完全树，因此常常利用数组储存。分为大根堆和小根堆，以小根堆为例，任意一个根的孩子结点的值总是不小于其父节点，同时两个孩子也是一个小根堆。

我们利用向量作为堆的基础，通过迭代方式，实现堆的操作，主要成员函数如下：

·上浮调整void siftFix(Iter beg, Iter end, Compare cmp);

·下沉调整void sinkFix(Iter beg, Iter end, Compare cmp);

2.1.5 Priority\_queue类

基于Vector类作为存储数据的数据结构，Heap类中的方法，形成优先队列，其中参数模板包括数据类型、数据底层结构（默认为Vector）以及元素之间的比较方法。比较方法作为仿函数传入类中作为类的数据成员。其中入队出队的时间复杂度均为O(logn)

主要代码如下：

template<class T, class Seq = Vector<T>, class Compare = Less<T>>

class Priority\_queue {

private:

Seq data;

Compare cmp;

public:

//构造函数

Priority\_queue() :data(), cmp() {}

Priority\_queue(const Compare& cmp) :data(), cmp(cmp) {}

//析构函数

~Priority\_queue() {}

//判断是否为空

bool empty() const;

//返回大小

int size() const;

//顶

const T top() const;

//加入元素

void emplace(const T& x){

data.emplace\_back(x);

pushHeap(data.begin(), data.end(), cmp);

}

//弹出元素

void pop(){

popHeap(data.begin(), data.end(), cmp);

data.pop\_back();

}

};

2.1.6 并查集类

并查集作为一种基于数组存储的树型结构，用与处理一些不相交集合的合并以及查询两个元素是否在同一个集合中的问题具有及其重要的作用。对应实际问题例如等价类的划分。

并查集数组作为双亲数组，初始作为一个森林，双亲均指向自己。随着集合的合并——两棵树合并为一棵树，选择将双亲指向另一棵树，实现集合的合并。

对应并查集的各种操作，通常来说最坏的时间复杂度为O(n)，但通过一些方法的优化，最实际问题中可以将其降低。例如，对于合并两个集合的时候，可以将层数小的树合并到层数大的树的根，由此来减少层数的增加带来搜索遍历的结点个数的增加、时间的增多。若是现实问题对于连接结构不足要求，我们可以采用压缩路径的方法，在搜索双亲结点的过程中不断将双亲指向双亲的双亲，由此实现树形结构的压缩，使得对于每个结点的搜索双亲的路径尽可能的减少。

主要代码如下：

//并查集

class unionFind {

public:

//构造函数

unionFind() {

numOfSet = 0;

}

unionFind(const int n) {

fa.resize(n);

for (int i = 0; i < n; ++i)

fa[i] = i;

numOfSet = n;

}

void resize(const int n) {

fa.resize(n);

for (int i = 0; i < n; ++i)

fa[i] = i;

numOfSet = n;

}

int find(int x) {

if (numOfSet == 0)

return 0;

while (fa[x] != x) {

fa[x] = fa[fa[x]];

x = fa[x];

}

return x;

}

bool union\_xy(int x, int y) {

int root\_x = find(x);

int root\_y = find(y);

if (root\_x == root\_y)

return false;

fa[root\_x] = root\_y;

//连接上了

--numOfSet;

return true;

}

bool is\_connected(int x, int y) {

return find(x) == find(y);

}

inline int getNumOfSet()const {

return numOfSet;

}

void clear() {

fa.clear();

numOfSet = 0;

}

private:

Vector<int> fa;

int numOfSet;

};

2.1.7 边结点类

该类定义结构体存储边的信息，其中包括：两端结点序号以及边的权值大小。于此同时，重载 < 和 > 操作符，为在优先队列中的排序做准备。

主要代码如下：

struct edge {

int first, second, val;

//进入优先队列大于小于相反重载

friend bool operator>(const edge& a, const edge& b) {

return a.val < b.val;

}

friend bool operator<(const edge& a, const edge& b) {

return a.val > b.val;

}

};

2.2 总体设计

设计电网系统类，其中主要私有成员包括：存储顶点名称的向量Vector<edge> edges;根据顶点名称寻找对应序号的映射Map<String, int> hash;存储最小生成树边的集合Vector<edge> ansEdge记录所有顶点是否都已连通的并查集;unionFind uf;邻接矩阵Vector<Vector<int>> adj;顶点个数int tot;最小生成树代价int cost;最小生成树起始点序号int start;

成员函数包括：

//构造函数elecSys();

//析构函数~elecSys(){}

//创建各个结点名称void buildNode();

//添加电网的边void addEdges();

//构建最小生成树提示void buildTree();

//Prim算法构造最小生成树void Prim();

//Prim算法优先队列版本void PrimPriorityQueue();

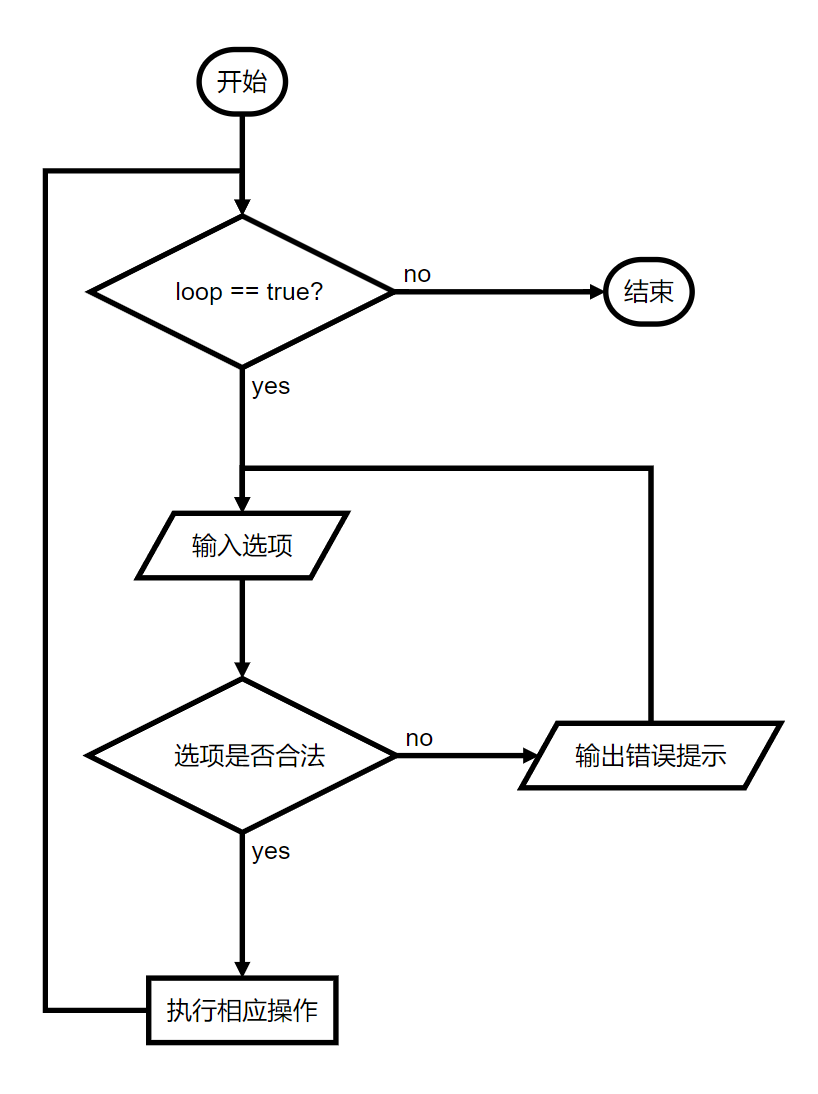
//Kruskal算法int Kruskal();

//显示最小生成树void show();

3 项目实现

3.1 总体实现

3.1.1 总体实现流程图



3.1.2 总体实现思路

通过输出操作提示，请用户输入对应的字母进行对应的操作，进行对应函数的调用。这时候我们需要注意到若输入了错误的字母，需要进行输入错误判断，并输出错误提示，请求重新输入。

3.1.3 总体实现主要代码

cout << "\*\*\t\t电网造价模拟系统\t\t\*\*" << endl;

cout << "==================================================" << endl;

cout << "\*\*\t\tA --- 创建电网顶点\t\t\*\*" << endl;

cout << "\*\*\t\tB --- 添加电网的边\t\t\*\*" << endl;

cout << "\*\*\t\tC --- 构建最小生成树\t\t\*\*" << endl;

cout << "\*\*\t\tD --- 显示最小生成树\t\t\*\*" << endl;

cout << "\*\*\t\tE --- 退出 程序\t\t\*\*" << endl;

cout << "==================================================" << endl;

elecSys es;

bool loop = true;

char op;

while (loop)

{

cout << "\n请选择操作: ";

cin >> op;

switch (op)

{

case 'a':

case 'A':

es.buildNode();

break;

case 'b':

case 'B':

es.addEdges();

break;

case 'c':

case 'C':

es.buildTree();

break;

case 'd':

case 'D':

es.show();

break;

case 'e':

case 'E':

loop = false;

break;

default:

cout << "输入有误，请重新输入！" << endl;

cin.clear();

cin.ignore(65536, '\n');

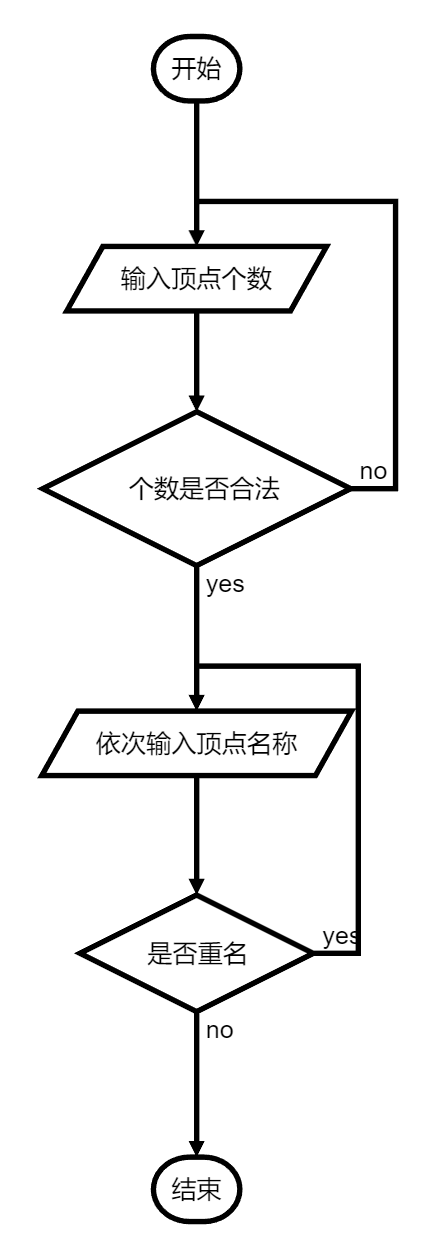
break;

}

}

3.2 输入顶点名称实现

3.2.1 输入顶点名称实现流程图



3.2.2 输入顶点名称实现思路

这是构造最小生成树的第一个操作，因此，在进行此操作之前，我们需要对整个类初始化，清空之前信息的记录，以防止之前的数据对本次最小生成树造成的影响。其次，请用户输入结点个数，要求是正整数，否则输出输入错误的提示，并请求重新输入，直至输入正确为止。再者，根据所输入的结点个数，请用户输入各个节点的名称，这里使用Vector<String>存储，更加灵活。这时候需要注意所输入的顶点名称不应该与之前的输入重复，因此，我们利用由红黑树作为底层的Map<String, int>进行再一次存储，保存顶点名称与对应需要的键值对，由此牺牲一定的空间来加快查找的速度。在数据量很大的时候，相比于顺序查找的时间复杂度O(n)，我们使用的红黑树的查找O(logn)相信时间上的效率会由更好的表现。只有当找不到相同的关键码时，才进行新输入的顶点名称的存储，否则提示用户出现重名，请求重新输入。最后，得到了顶点的个数，可以由此对并查集以及邻接矩阵进行初始化。

3.2.3 输入顶点名称实现主要代码

//清除上次记录

clear();

cout << "请输入结点的个数: ";

while (1){

cin >> tot;

if (cin.fail() || tot <= 0){

cout << "请输入正整数" << endl;

cin.clear();

cin.ignore(65536, '\n');

}

else

break;

}

cout << "请依次输入各顶点的名称:" << endl;

String temp;

for (int i = 0; i < tot; ++i){

cin >> temp;

//保证没有相同顶点

auto Nullptr = tree\_iterator<String, int>(NULL);

auto it = hash.find(temp);

if (it == Nullptr){

hash[temp] = i;//顶点名称与下标对应

node.emplace\_back(temp);

}

else{

cout << "不能出现相同顶点，请重新输入该顶点名称" << endl;

--i;//出现重名，重新输入本次顶点的名称

}

}

//邻接矩阵初始化

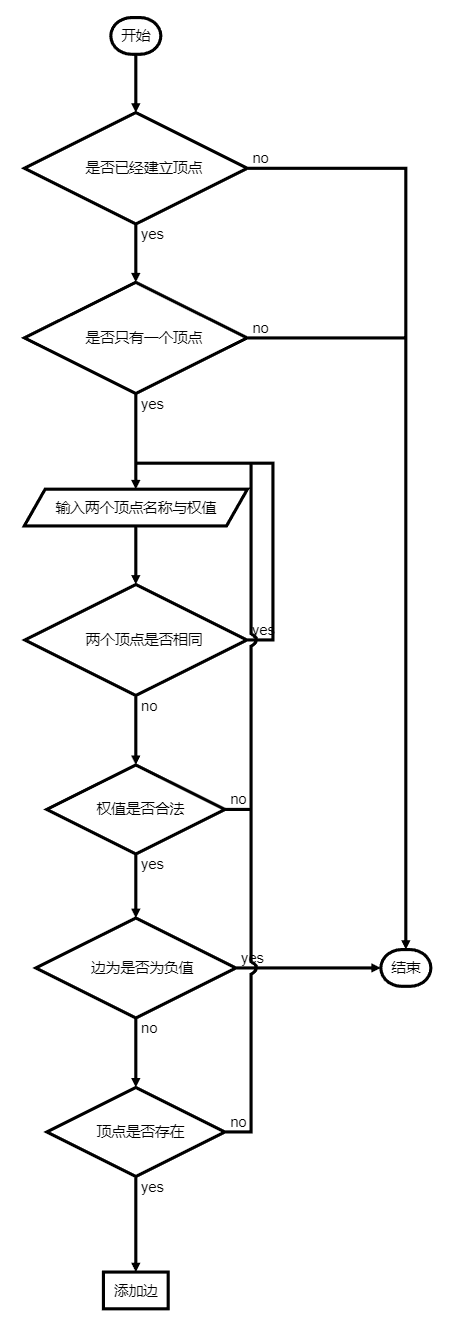
adj.resize(tot, Vector<int>(tot, INT\_MAX));

//并查集初始化

uf.resize(tot);

3.3 输入添加电网的边实现

3.3.1 输入添加电网的边实现流程图



3.3.2 输入添加电网的边实现思路

首先需要判断是否已经构建电网顶点，若否，则输出提示用户需要进行上一个操作——构建电网顶点。其次，需要判断顶点个数，若只有一个顶点，则输出提示无需添加电网的边。接着才进行电网边的输入，并更新图的存储。在此过程中，我们需要判断所输入的顶点名称是否在已输入的顶点集合当中，这里利用Map<String, int>来记录与查找。同时我们在此将边加入并查集，由此在下一个操作之前判断输入所构成的图是否连通。

3.3.3 输入添加电网的边实现主要代码

主要代码如下：

if (tot == 0){

cout << "请先构建电网顶点" << endl;

return;

}

if (tot == 1){

cout << "只有一个顶点，无需构造最小生成树" << endl;

return;

}

String nodeA, nodeB;

int val;

cout << endl << "输入边的值为负数时候退出输入" << endl;

while (1){

cout << "请输入两个顶点及边: ";

cin >> nodeA >> nodeB >> val;

if (nodeA == nodeB)//判断输入顶点是否相同

{

cout << "输入电网顶点不应相同，请重新输入" << endl;

cin.ignore(65536, '\n');

continue;

}

if (cin.fail()){

cout << "输入电网边值有误，请重新输入" << endl;

cin.clear();

cin.ignore(65536, '\n');

continue;

}

if (val <= 0)//不出现负边，退出

break;

//判断输入顶点是否有效

auto itA = hash.find(nodeA);

auto itB = hash.find(nodeB);

if (itA == hash.end()){

cout << "顶点" << nodeA << "不存在" << endl;

continue;

}

if (itB == hash.end()){

cout << "顶点" << nodeB << "不存在" << endl;

continue;

}

//更新邻接表

adj[itA->second][itB->second] = min(adj[itA->second][itB->second], val);

adj[itB->second][itA->second] = adj[itA->second][itB->second];

adj[itA->second][itA->second] = 0;

adj[itB->second][itB->second] = 0;

//标记连接上结点

uf.union\_xy(itA->second, itB->second);

}

//都连上了numOfSet == 1

if (uf.getNumOfSet() == 1)

edgeFinish = true;

3.4 生成最小生成树实现

3.4.1 生成最小生成树实现思路

一个n阶连通图中可能存在多条相连的边，我们需要从其中挑选出n - 1条边满足不构成环，将所有顶点相连，并且边权值之和最小，由此生成的子图为最小生成树。构造最小生成树的思想在于如何优先挑选权值最小的边，并且在挑选的过程中不构成回路。这里可以运用贪心算法的思想，在求解问题的时候，总是做出在当前看来是最好的选择，这里的局部最优解可以作为整体最优解。

对于细节，这里我们同样需要判断电网的边是否添加完成，只有当输入的图为连通图时，我们才可以调用最小生成树生成函数，否则输入提示，请求用户继续完善边的添加。

3.4.2 生成最小生成树实现主要代码

if (tot == 0){

cout << "请先构建电网顶点" << endl;

return;

}

if (tot == 1){

cout << "只有一个顶点，无需构造最小生成树" << endl;

return;

}

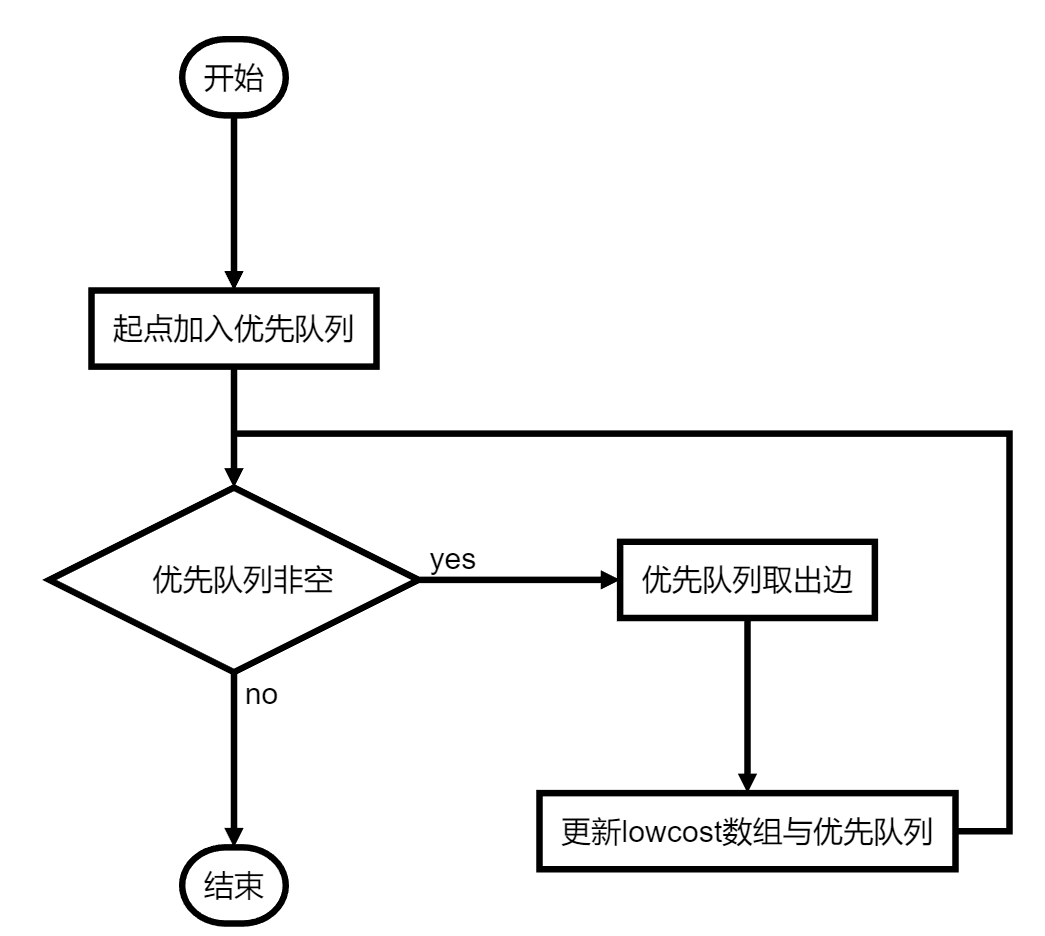
if (edgeFinish == false){

cout << "仍有电网顶点尚未连接，无法构造最小生成树，请继续添加电网边" << endl;

return;

}

3.4.3 Prim算法生成最小树流程图



3.4.4 Prim算法生成最小树思路

Prim算法的具体过程为：任意选择图中的一个顶点作为生成树的根结点，并声明一个标记数组，记录已经加入生成树的集合中的结点；声明一个距离数组dis[]，其中dis[i]表示生成树集合中的顶点到顶点i的最短距离。这里我们利用优先队列优化，使得每次取到的都是队列中权值最小的边，每次循环中，我们需要判断取到的边的对应顶点是否已经在生成树的顶点集合中，若在则跳过本轮循环，若不在则将该点加入最小生成树的顶点集合——进行标记，并且加入答案边集。然后在该点周围寻找没有遍历过的且能更新dis[]的顶点，存在则加入优先队列，进入下一次循环。

可以发现Prim算法与Dijkstra算法类似，只是在更新dis数组的时候略有不同。前者是为了每条边的权值都尽可能短，后者是为了使得从开始的结点到其余每个结点的路径尽可能短。由此说明使用Dijkstra算法虽然也可以生成生成树，但并不一定是最小生成树。

上述过程由于利用堆优化，因此我们将时间复杂度从O(n2)降低到了O(logn)。

3.4.5 Prim算法生成最小树主要代码

Vector<bool> vis(tot, false);

Vector<int> dis(tot, INT\_MAX);

Priority\_queue<edge> q;

q.emplace(edge{start, start, 0});

edge temp;

while (!q.empty()){

//取出最小距离

temp = q.top();

q.pop();

//遍历过了跳过

if (vis[temp.second])

continue;

//没遍历过加入答案边集

if(temp.first != temp.second)

ansEdge.emplace\_back(temp);

//标记已经遍历过

vis[temp.second] = true;

//更新代价

cost += temp.val;

//从temp.second周围找更短的边

for (int j = 0; j < tot; ++j)

{

//没有遍历过，并且从start到j比temp.second到j长

if (vis[j] == false && adj[temp.second ][j] < dis[j])

{

dis[j] = adj[temp.second][j];

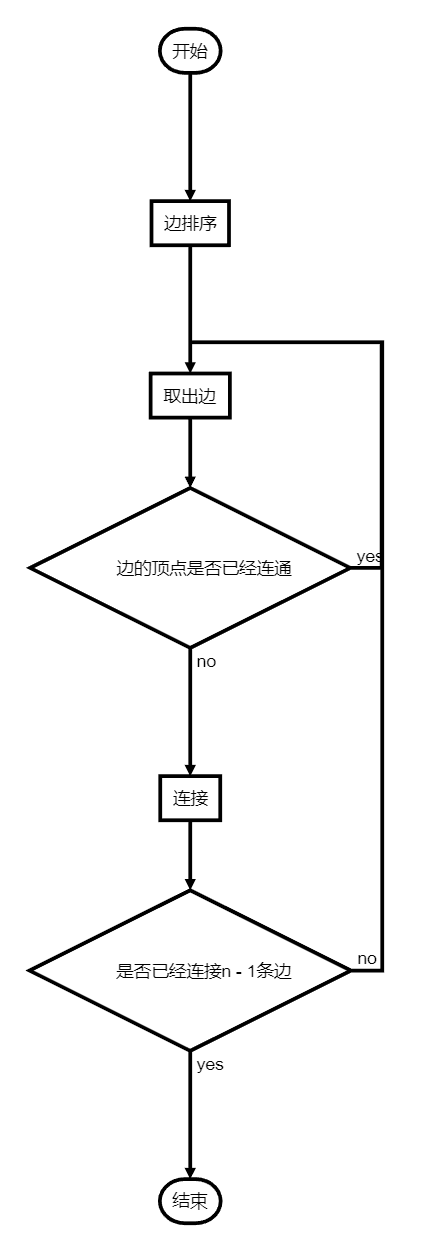
q.emplace(edge{temp.second, j, dis[j]});

}

}

}

3.4.6 Kruskal算法生成最小树流程图



3.4.7 Kruskal算法生成最小树思路

与Prim算法不同，本算法不需要选定一个开始的顶点，而是直接从边的角度除法，首先根据边的权值大小排序，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回路，就可以将它添加入答案边集。对于顶点个数n，只要选取了n - 1条边，即构成了最小生成树。

这里我们选择使用堆排序，对于可以在较少的时间复杂度中完成。

这里需要主义的问题在于，如何判断当前选择的边是否和已选择边构成回路。我们采用并差集的方法。将已选择的边的两个端点合并为同一个结合，由此在加入新边之前，首先判断改变的两个端点是否已经存在于同一个集合当中，若存在，则说明不能再添加边，否则可以。

3.4.8 Kruskal算法生成最小树主要代码

//从小到达代价排序

Sort(&edges[0], edges.size());

//顶点个数初始化并查集

unionFind uf(tot);

for (int i = 0; i < edges.size(); ++i)

{

//判断是否成环

if (uf.is\_connected(edges[i].first, edges[i].second))

continue;

//总代价

cost += edges[i].val;

//连起来的边数

++edgeNum;

//将两个顶点连起来

uf.union\_xy(edges[i].first, edges[i].second);

//连起来的边数为tot-1，则说明完成

if (edgeNum == tot - 1)

break;

}

3.5 展示生成最小树的边实现

3.5.1 展示生成最小树的边实现思路

在之前最小生成树的算法的过程中，我们对边集数组的更新，在此可以发挥作用。通过顶点名称数组查询顶点编号对应的名称，我们可以依次输出最小生成树边的信息。

在此之前我们需要判断是否已经完成最小生成树的生成。

3.5.2 展示生成最小树的边实现主要代码

if (ansEdge.empty()){

cout << "请先生成最小生成树！" << endl;

return;

}

cout << "最小生成树的顶点和边为: " << endl;

for (int i = 0; i < ansEdge.size(); ++i){

cout << node[ansEdge[i].first] << "-<"

<< ansEdge[i].val << ">-"

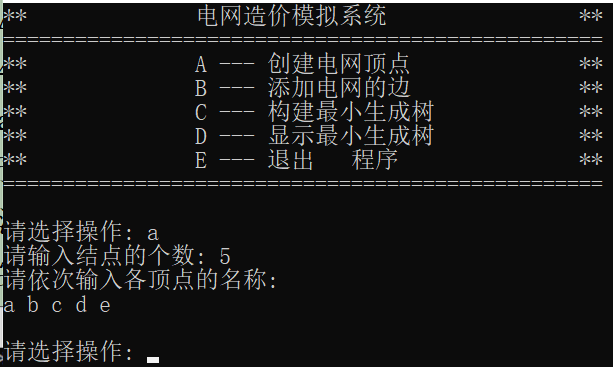
<< node[ansEdge[i].second] << " ";

}

4 项目测试

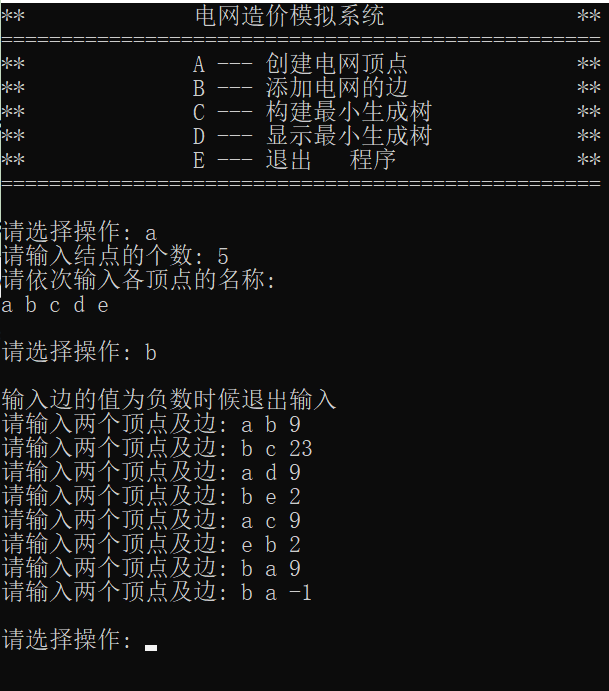
4.1 输入顶点名称测试

4.1.1 正确输入

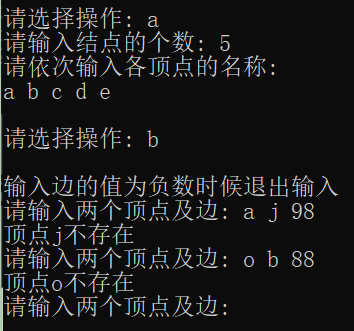


4.2 添加电网边顶点测试

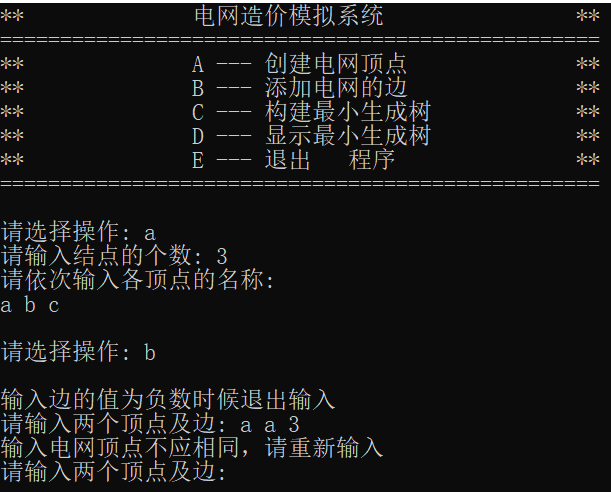
4.2.1 正确输入



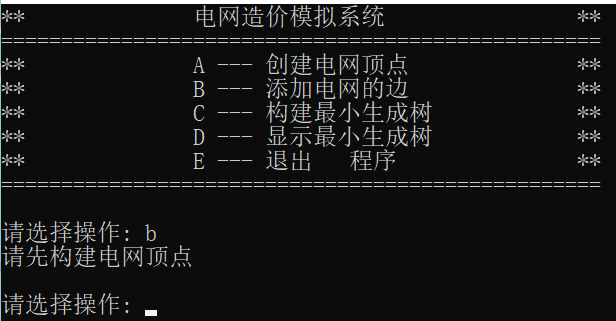
4.2.2 输入不存在的顶点名称



4.2.3 输入相同顶点名称

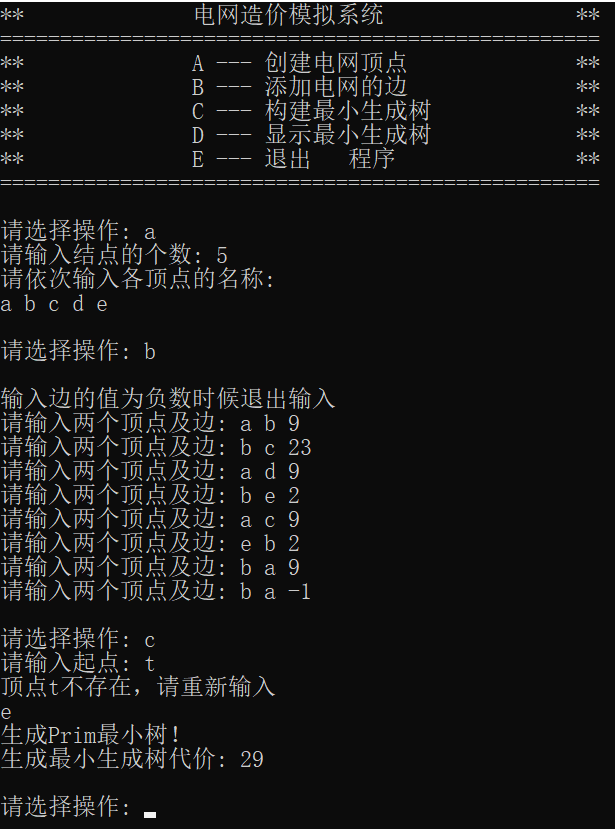


4.2.4 未建立顶点名称

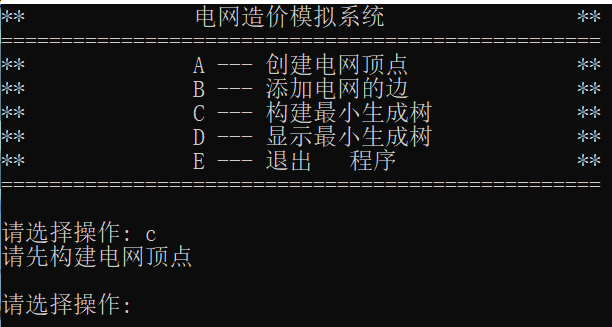


4.3 最小生成树测试

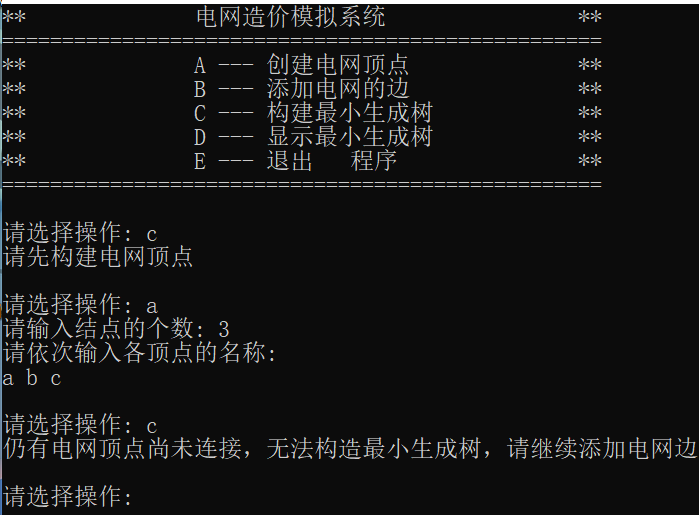
4.3.1 输入不存在的顶点名称并正确输入



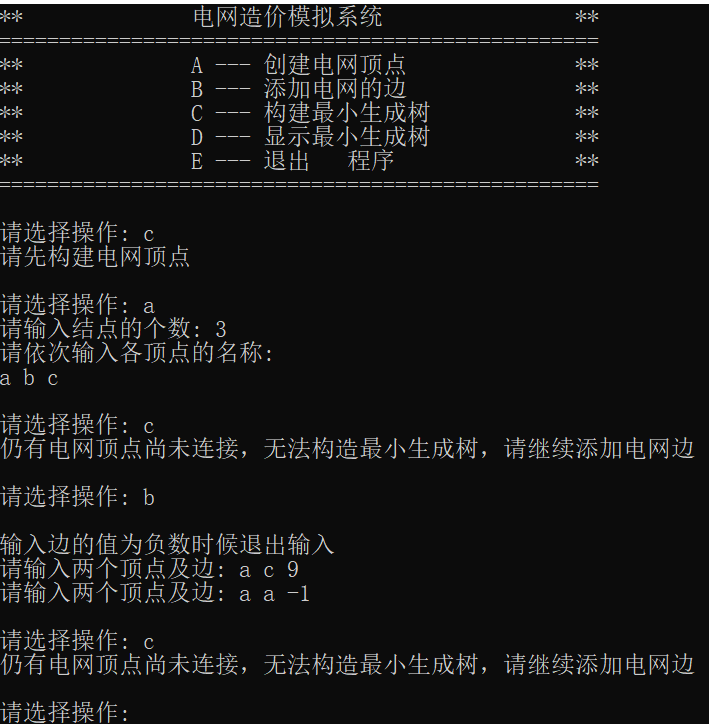
4.3.2 未建立顶点名称



4.3.3 未添加边

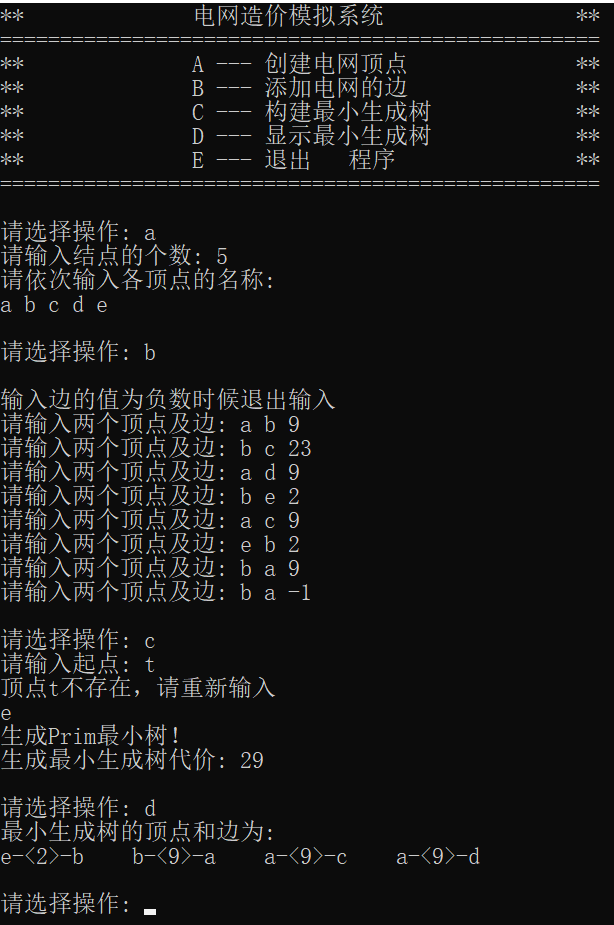


4.3.4 已添加边但不连通图

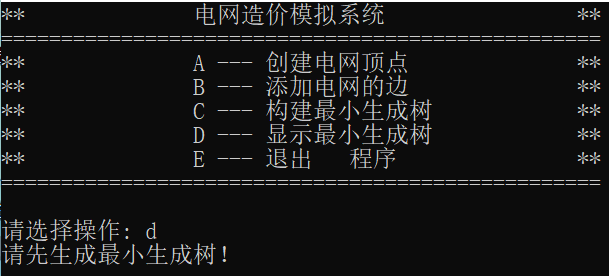


4.4 展示最小生成树顶点和边测试

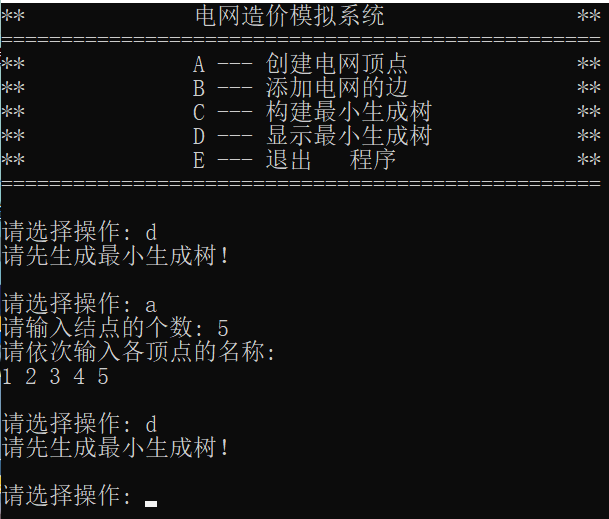
4.4.1 正确输入



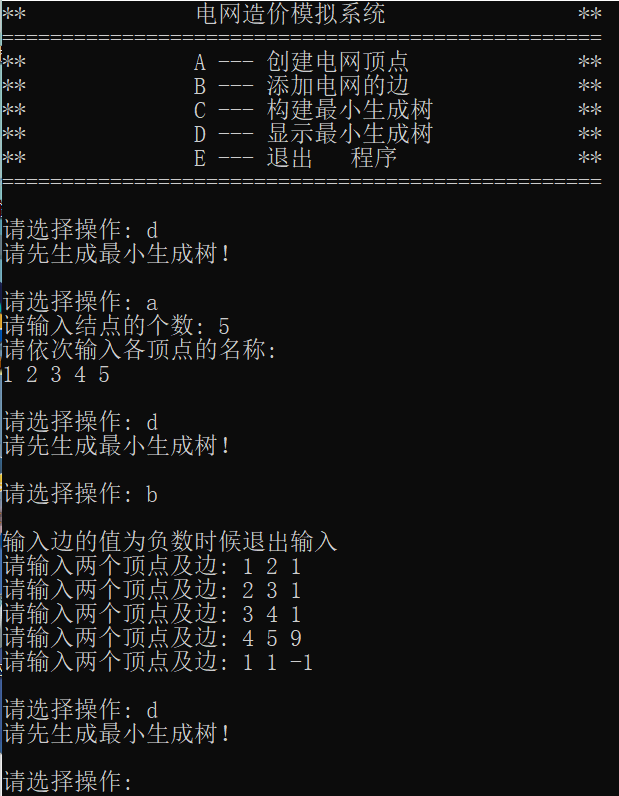
4.4.2 未建立顶点名称



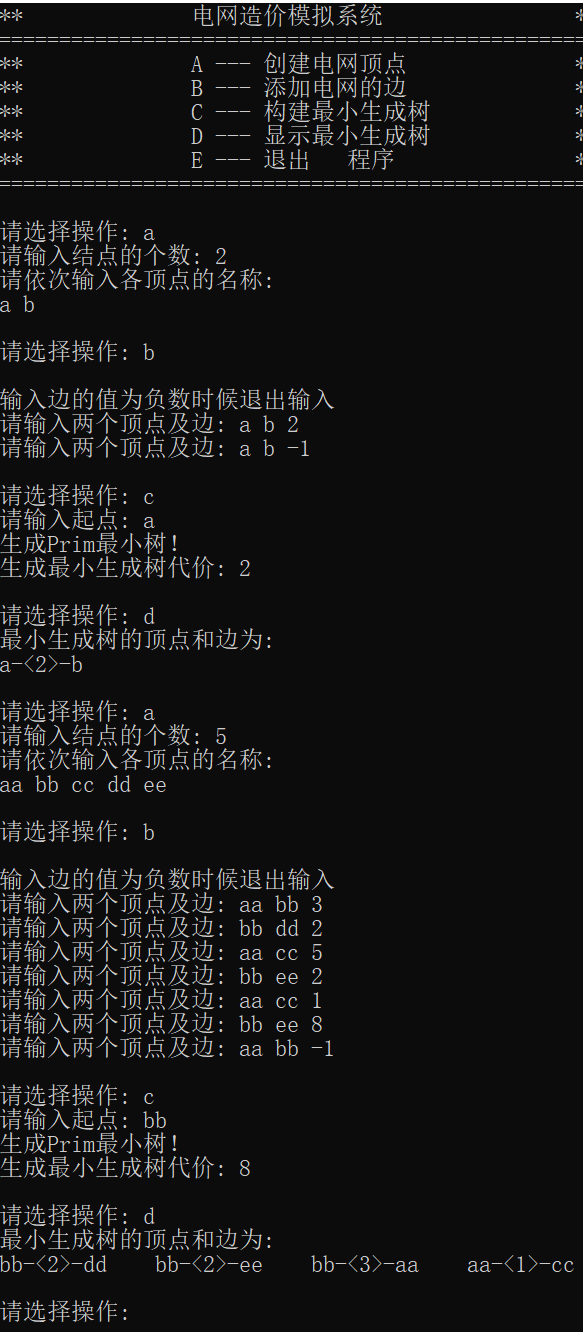
4.4.3 未添加边



4.4.4 未生成最小生成树



4.5 第二轮电网建设测试



4.6 有多个平行边的图

