**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 刘淑仪

学 号 2251730

指 导 教 师 张 颖

学 院 专 业 软件学院 软件工程



二〇二三 年 十一 月 二十八 日

目录

1 项目分析 1

1.1 项目背景分析 1

1.2 项目功能分析 1

2 项目设计 2

2.1 数据结构设计 2

2.2 类结构设计 2

2.3 成员与操作设计 3

2.4 系统设计 7

3 功能设计与项目实现 7

3.1 创建电网顶点的实现 7

3.1.1 创建电网顶点实现思路 7

3.1.2 创建电网顶点核心代码 8

3.1.3 创建电网顶点实现示例 9

3.2 添加边功能的实现 9

3.2.1 添加边功能实现思路 9

3.2.2 添加边功能核心代码 10

3.2.3 添加边功能实现示例 12

3.3 构造并显示最小生成树的实现 12

3.3.1 构造并显示最小生成树实现思路 12

3.3.2 构造并显示最小生成树核心代码 12

3.3.3 构造并显示最小生成树实现示例 14

3.4 本项目涉及Graph.h内容的主要实现 14

3.4.1 最小堆和邻接表类的定义 14

3.4.2 Graph.h核心代码 15

4 项目测试 15

4.1 功能测试 15

4.1.1 基本功能测试 15

4.1.2 边界条件测试 16

4.2 错误测试 17

4.2.1 输入验证错误 17

4.2.1.1 输入验证错误判断思路 17

4.2.1.3输入示例 17

4.2.2 异常情况处理 20

4.3 Linux环境测试 21

5 集成开发环境与编译运行环境 22

# 项目分析

## 项目背景分析

本项目旨在解决一个城市中的多个小区（n个）之间电网建设的问题，以实现它们的互联互通。关键点在于确保所有小区间的电网连接同时最小化总工程造价。如果将其抽象化为数据结构问题，则应将小区视为顶点，电网线路视为边，使用图论来模拟整个电网系统。利用如Prim或Kruskal算法，寻找覆盖所有顶点（小区）且边的总权重（成本）最小的生成树。使用合适的数据结构（例如邻接表）来有效存储和处理小区和线路之间的关系。

## 项目功能分析

该项目利用图论的概念来模拟城市电网。城市的每个小区被视为图中的一个顶点，而小区之间可能的电网连接则作为边。项目中采用邻接表来表示这个图，这样的数据结构使得与顶点相邻的边可以高效地被访问和处理。

其主要功能的实现主要依赖于

1、图的表示和操作：通过insertVertex和removeVertex方法，可以灵活地添加和删除小区（顶点）。insertEdge和removeEdge方法允许添加和删除小区之间的电网连接（边），并附带成本信息。

2、最小生成树算法（Prim算法）：该项目核心功能是通过Prim算法来构造最小生成树，从而找到最小总造价的电网布局方案。Prim算法是一种高效的图算法，用于在一个加权连通图中找出覆盖所有顶点的、权重总和最小的树结构。

3、最小堆（MinHeap）：使用了最小堆（MinHeap）来管理边。最小堆是一种特殊的完全二叉树，其中每个父节点的值都小于或等于其子节点的值。在Prim算法中，这有助于快速选出当前可用边中的最小边。用于在每一步Prim算法中快速找到权重最小的边。同时维护一个动态变化的候选边集合，以支持算法的进行。

4、边和顶点的动态处理：动态添加和删除顶点和边的功能，这对于建模一个不断变化的城市电网系统是必要的。用户可以根据实际需要增加或减少小区（顶点），或者修改小区间的电网连接（边）。

# 项目设计

## 数据结构设计

在本项目中，针对城市电网建设的特殊需求，我们采用了邻接表和最小堆作为主要的数据结构。这些结构的选择与项目的核心功能——构建最小生成树和管理电网连接紧密相关。首先，邻接表用于高效地表示和管理图中的顶点和边。在电网建设的问题中，顶点代表小区，而边代表可能的电网连接。由于并非所有小区都直接相连，因此使用邻接表可以更加高效地处理稀疏的连接网络。这种数据结构使得访问与任一顶点相连的所有边变得快捷，而且方便地添加或删除边，以动态调整电网结构。

其次，为了在Prim算法中快速选出权重最小的边，项目中使用了最小堆。最小堆作为一种优先队列，始终允许快速访问当前可用边中权重最小的那一条。这对于Prim算法来说至关重要，因为它保证了每一步都能选择当前可用的最小成本边，从而实现整个电网的成本最小化。

## 类结构设计

在本项目中，针对城市电网建设的需求，我们设计了几个关键的类来实现项目功能。首先，在Graph.h中，创建了MyGraphlnk类，用于表示和操作电网图。这个类的私有成员包括顶点表数组（NodeTable），用于存储图中的顶点和与之相连的边。公有成员则包括各种方法，如构造函数、析构函数、添加或删除顶点、添加或删除边、获取顶点或边的信息等，这些方法为电网图的建设和管理提供基础功能。

接着，在项目的主要实现文件中，创建了graphTree类，继承自MyGraphlnk类，并增加了特定于电网建设问题的功能。这个类的私有成员包括最小生成树的边数组（mstEdges）和边计数器（mstEdgeCount），用于存储构建过程中选定的边。公有成员则包括实现电网建设核心算法（如Prim算法）的方法、打印最小生成树、以及与用户交互的界面，如菜单显示和操作选择。

此外，为了支持Prim算法的高效运行，我们还设计了MinHeap类，用于管理图中的边。这个类主要用于在Prim算法过程中快速选出当前权重最小的边。它的私有成员包括堆数组、堆容量和当前堆大小，而公有成员则包括堆的操作方法，如插入、删除和获取堆顶元素等。

## 成员与操作设计

未注释的函数请见下文实现对应功能时代码的函数标签。

Graph.h内存储最小堆、邻接表及其基本函数实现。

/\* 最小堆 \*/

template<class T>

class MinHeap {

private:

T\* heap; // 指向堆数组的指针

int capacity; // 堆的最大容量

int heapSize; // 当前堆中的元素数量

// 返回父节点的索引

int parent(int i) { return (i - 1) / 2; }

// 返回左子节点的索引

int left(int i) { return (2 \* i + 1); }

// 返回右子节点的索引

int right(int i) { return (2 \* i + 2); }

// 维护堆的性质

void heapify(int i) {

int l = left(i);

int r = right(i);

int smallest = i;

if (l < heapSize && heap[l] < heap[i])

smallest = l;

if (r < heapSize && heap[r] < heap[smallest])

smallest = r;

if (smallest != i) {

std::swap(heap[i], heap[smallest]);

heapify(smallest);

}

}

public:

// 构造函数

MinHeap(int cap) {

heapSize = 0;

capacity = cap;

heap = new T[cap];

}

// 析构函数

~MinHeap() {

delete[] heap;

}

// 插入一个元素

void push(T key) {

if (heapSize == capacity) {

std::cout << "Heap overflow" << std::endl;

return;

}

heapSize++;

int i = heapSize - 1;

heap[i] = key;

// 调整堆

while (i != 0 && heap[parent(i)] > heap[i]) {

std::swap(heap[i], heap[parent(i)]);

i = parent(i);

}

}

// 检查堆是否为空

bool empty() const {

return heapSize == 0;

}

// 提取堆顶元素

T top() {

if (empty()) return T();

return heap[0];

}

// 删除堆顶元素

void pop() {

if (heapSize <= 0) return;

if (heapSize == 1) {

heapSize--;

return;

}

heap[0] = heap[heapSize - 1];

heapSize--;

heapify(0);

}

// 获取堆的大小

int size() { return heapSize;}

};

/\* 邻接表 \*/

template <class T, class E>

struct Edge {

int dest; // 边的另一顶点位置

E cost; // 边上的权值

Edge<T, E>\* link; // 下一条边链指针

Edge() {}; // 构造函数

Edge(int num, E weight) :dest(num), cost(weight), link(NULL) {}

// 判边不等否

bool operator !=(Edge<T, E>& R)const { return(dest != R.dest) ? true : false; }

// 重载大于运算符

bool operator >(const Edge<T, E>& R) const { return cost > R.cost; }

// 重载小于运算符

bool operator <(const Edge<T, E>& R) const { return cost < R.cost; }

};

template <class T, class E>

struct Vertex {

T data; // 顶点的名字

Edge<T, E>\* adj; // 边链表的头指针

};

template <class T, class E>

class MyGraphlnk {

friend std::istream& operator>>(std::istream& in, MyGraphlnk<T, E>& G); //输入

friend std::ostream& operator<<(std::ostream& out, MyGraphlnk<T, E>& G); //输出

protected:

int maxVertices; // 图中最大顶点数

int numEdges; // 当前边数

int numVertices; // 当前顶点数

int mstEdgeCount = 0; // 跟踪MST中边的数量

Edge<T, E> mstEdges[DefaultVertices]; // MST中的边

Vertex<T, E>\* NodeTable; // 顶点表（各边链表的头结点）

// 给出顶点vertex在图中的位置

int getVertexPos(const T vertex) {

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

if (NodeTable[i].data == vertex) return i;

return -1;

}

public:

const E maxWeight = 65536; // 代表无穷大的值

MyGraphlnk(int sz = DefaultVertices);

~MyGraphlnk();

bool GraphEmpty() const {

return numEdges == 0; // 如果没有边，则图为空

}

bool GraphFull()const { // 判断图满否

return numVertices == maxVertices || numEdges == maxVertices \* (maxVertices - 1) / 2;

}

int getVerticesNum() { return numVertices; }

int getEdgesNum() { return numEdges; }

T getValue(int i); // 取顶点i的值，i不合理返回0

E getWeight(int v1, int v2); // 取边v1、v2上的权值

int getFirstNeighbor(int v); // 取顶点v的第一个邻接顶点

int getNextNeighbor(int v, int w); // 取邻接顶点w的下一个邻接顶点

bool insertVertex(const T& vertex); // 插入一个顶点

bool insertEdge(int v1, int v2, E cost); // 插入边(v1,v2)，权重为const

bool removeVertex(int v); // 删去顶点v和相关的所有联边

bool removeEdge(int v1, int v2); // 在图中删去边(v1,v2)

void printEdges()const; // 访问已存在的所有边

};

electric grid construction cost simulation system.cpp内存储该问题的对应解决方案类：

template <class T, class E>

struct MSTEdge {

int source; // 边的起点

int dest; // 边的终点

E cost; // 边的权值

// 默认构造函数

MSTEdge() : source(-1), dest(-1), cost(E()) {}

MSTEdge(int s, int d, E c) : source(s), dest(d), cost(c) {}

};

typedef class graphTree :public MyGraphlnk<char,int> {

private:

MSTEdge<char, int> mstEdges[DefaultVertices];

int mstEdgeCount = 0;

public:

void menu();

int operation();

void createVertex();

void addEdge();

void constructTree();

int maxEdge(const int vertex) { return vertex \* (vertex - 1) / 2; }

void prim(int startVertex);

void printMST();

void Exit();

}GT;

## 系统设计

在本项目的系统设计中，代码被细致地分为多个模块，每个模块承担着特定的职责，从而实现了城市电网建设问题的有效解决。例如，MyGraphlnk 类和其派生类 graphTree 负责图结构的实现和电网建设的核心逻辑，而 MinHeap 类则专门处理在Prim算法中的边的优先队列管理。

电网的图结构通过邻接表来实现，这种数据结构适合于电网建设问题，因为它允许高效地管理稀疏图，即使在小区之间的直接连接较少的情况下也是如此。邻接表的使用，特别是在实现Prim算法时，显著提高了对图中边和顶点的操作效率。

此外，数据和方法被封装在类中，减少了全局状态的使用。这一设计不仅提高了代码的可读性和可维护性，还使得每个类都成为一个独立的、具有特定功能的模块。例如，graphTree 类封装了电网建设的所有相关操作，如顶点和边的添加、最小生成树的构建等，而 MinHeap 类则独立管理边的优先级队列。

# 功能设计与项目实现

## 创建电网顶点的实现

### 创建电网顶点实现思路

首先，代码提示用户输入顶点的数量，即城市中小区的数量。使用一个循环来确保用户输入的是有效的数字。如果输入不是有效的正整数或超出了顶点的最大限制（DefaultVertices），则提示用户重新输入。如果输入失败（如用户输入了非数字字符），则清除输入缓冲区并提示用户重新输入。

用户被提示输入每个顶点的名称。为了简化管理，每个顶点名称被限定为一个字符，最好是字母。使用 std::cin.getline 读取整行输入到 InputName 数组中。这里假设用户将顶点名称以空格分隔。

遍历 InputName 数组，将其中的非空格字符（即顶点名称）复制到 VertexName 数组中。这个步骤是为了从用户输入的字符串中提取单个顶点名称，同时忽略空格。使用 insertVertex 方法为图中每个提取出的顶点名称创建一个顶点。遍历 VertexName 数组，对于数组中的每个字符（顶点名称），调用 insertVertex 方法将其作为一个新顶点添加到图中。

### 创建电网顶点核心代码

// 创建顶点

void graphTree::createVertex()

{

std::cout << ">>>请输入顶点个数：";

int VertexNum = 0;

char InputName[DefaultVertices];

char VertexName[DefaultVertices];

while (true) {

std::cin >> VertexNum;

if (std::cin.fail() || VertexNum <= 0 || VertexNum > DefaultVertices) {

std::cin.clear();

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

std::cout << "输入数据有误，请重新输入\n";

}

else {

if (std::cin.get() == '\n')

break;

else {

std::cin.clear();

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

std::cout << "输入数据有误，请重新输入\n";

}

}

}

std::cout << "\n\*\* 注意：为了方便管理，每个顶点名称由一个字符组成，最好为字母 \*\*\n";

std::cout << ">>>请依次输入各顶点的名称：";

std::cin.getline(InputName, MAX\_SIZE);

for (int i = 0, j = 0; i < InputName[i] != '\0'; i++, j++) {

if (InputName[i] != ' ')

VertexName[j] = InputName[i];

else

j--;

}

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

if (getVertexPos(VertexName[i]) != -1) {

std::cout << "错误: 顶点 " << VertexName[i] << " 已存在，不允许重复添加相同的顶点。\n";

continue;

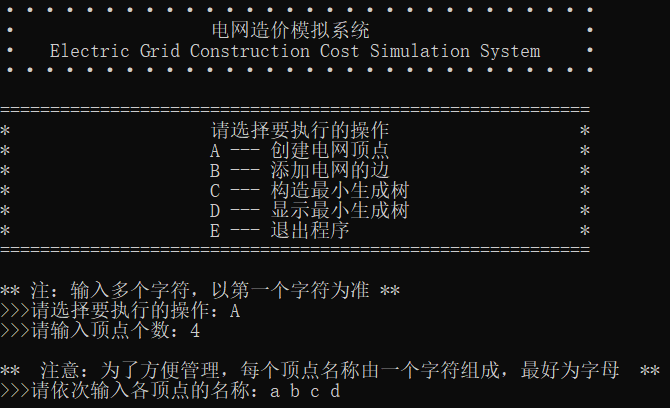
}

insertVertex(VertexName[i]);

}

}

### 创建电网顶点实现示例



## 添加边功能的实现

### 添加边功能实现思路

使用 maxEdge 函数计算基于当前顶点数量的最大可能边数，即 n(n-1)/2，其中 n 是顶点数。向用户显示最多可以添加的边数，并说明输入规则，如通过输入 'E' 来终止边的添加。提示用户输入两个顶点名称和边的权重（代表成本）。通过循环结构，允许用户根据最大边数进行多次边的添加。在每次循环中，先分别输入两个顶点名称，然后输入边的权重。

使用 getVertexPos 方法检查输入的顶点名称是否存在于图中。如果顶点不存在或用户输入 'E'，则提供相应的反馈并跳出循环或继续输入下一条边。用户输入边的权重（成本），并进行有效性检查，如确保输入为正整数。如果输入不合法，则提示用户重新输入。

使用 insertEdge 方法添加边到图中，其中包括两个顶点的索引和边的权重。如果添加边失败（例如，如果边已存在），则提示用户该边已存在，并请求重新输入。循环结束后，打印出图中所有的边，展示电网的当前状态。

### 添加边功能核心代码

// 添加边

void graphTree::addEdge()

{

int maxE = maxEdge(getVerticesNum());

char VertexName[2];

int VertexIndex[2];

int edgeWeight = 0;

bool end = 1;

std::cout << "\n\*\* 注意：1.根据已有的顶点个数，最多可以生成"<< maxE <<"条边。 \*\*\n";

std::cout << "\*\* 2.输入E可以终止输入，否则默认持续生成直至结束。 \*\*\n";

std::cout << "\*\* 3.顶点名称需存在，边由数字表示。 \*\*\n\n";

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

std::cout << ">>>请输入两个顶点及边：";

// 输入左顶点

while (true) {

std::cin >> VertexName[0];

VertexIndex[0] = getVertexPos(VertexName[0]);

if (VertexIndex[0] != -1)

break;

else if (VertexName[0] == 'E') {

end = 0;

break;

}

else

std::cout << "该顶点不存在，请重新输入！\n";

}

if (end == 0)

break;

// 输入右顶点

while (true) {

std::cin >> VertexName[1];

VertexIndex[1] = getVertexPos(VertexName[1]);

if (VertexIndex[1] != -1)

break;

else

std::cout << "该顶点不存在，请重新输入！\n";

}

// 输入边

while (true) {

std::cin >> edgeWeight;

if (std::cin.fail() || edgeWeight <= 0) {

std::cin.clear();

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

std::cout << "输入数据有误，请重新输入\n";

}

else {

if (std::cin.get() == '\n')

break;

else {

std::cin.clear();

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

std::cout << "输入数据有误，请重新输入\n";

}

}

}

if (!insertEdge(VertexIndex[0], VertexIndex[1], edgeWeight))

std::cout << "该边已存在，请重新输入！\n";

}

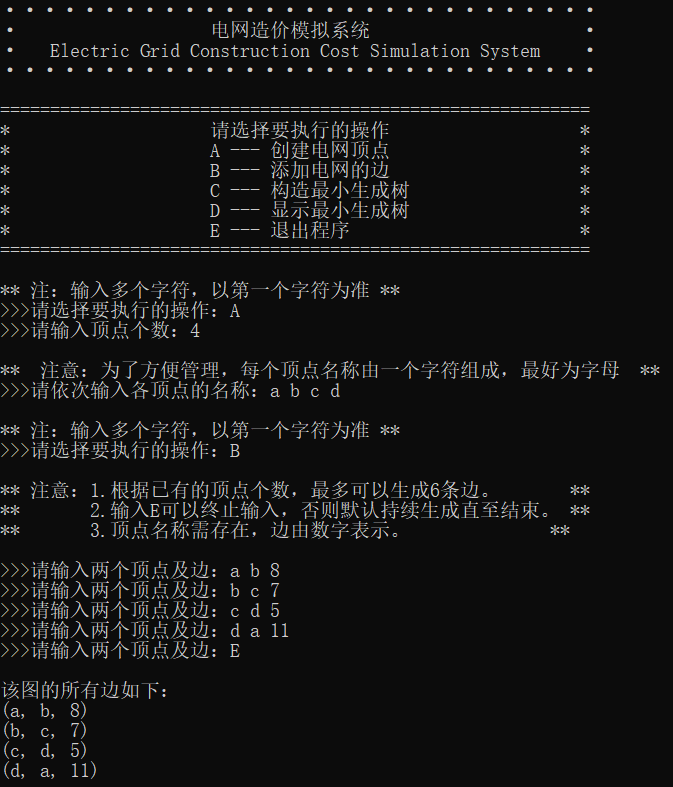
std::cout << std::endl;

std::cout << "该图的所有边如下：\n";

printEdges();

}

### 添加边功能实现示例



## 构造并显示最小生成树的实现

### 3.3.1 构造并显示最小生成树实现思路

首先，算法通过JosephusList.Locate(StartLocation)定位到起始节点p和其前驱节点q。接着，算法进入一个循环，直到被删除的人数达到(PersonNum - SurvivalPerson)。每次循环，算法检查当前的计数num是否符合死亡条件（即(num - 1) % DeathNum == 0）。如果不符合，就简单地移动p和q到下一个节点。如果符合死亡条件，则执行删除操作。首先保存当前的节点to\_delete（即p），然后移动p到下一个节点，并更新q的链接以删除to\_delete。随后调用OutputDeathPerson函数输出被删除节点的信息，并释放该节点的内存。当删除操作完成后，调用OutputSurvivalPerson方法输出最后幸存的人员信息。

### 3.3.2 构造并显示最小生成树核心代码

// 构造最小生成树

void graphTree::constructTree()

{

char inputVertex;

int startVertex;

std::cout << ">>>请输入起始顶点：";

while (true) {

std::cin >> inputVertex;

startVertex = getVertexPos(inputVertex);

if (startVertex != -1)

break;

else

std::cout << "该顶点不存在，请重新输入！\n";

}

prim(startVertex);

std::cout << "生成Prim最小生成树！\n";

}

void graphTree::prim(int startVertex)

{

int n = getVerticesNum(); // 获取顶点数

MinHeap<Edge<char,int>> heap(n \* (n - 1) / 2); // 假设最坏情况下的边数

bool visited[DefaultVertices] = { false }; // 固定大小的访问标记数组

// 确保不超出图的实际顶点数

if (n > DefaultVertices) {

std::cerr << "Error: Number of vertices exceeds MAX\_VERTICES" << std::endl;

return;

}

// 从起始顶点开始

visited[startVertex] = true;

int currentVertex = startVertex; // 初始化当前顶点为起始顶点

Edge<char, int>\* edge = NodeTable[startVertex].adj;

while (edge != NULL) {

heap.push(\*edge);

edge = edge->link;

}

// 构建最小生成树

while (!heap.empty() && mstEdgeCount < n - 1) {

Edge<char, int> minEdge = heap.top();

heap.pop();

int dest = minEdge.dest;

if (visited[dest]) continue; // 如果已经访问过，跳过

visited[dest] = true;

// 添加边到 MST，并记录起点和终点

mstEdges[mstEdgeCount++] = MSTEdge<char, int>(currentVertex, minEdge.dest, minEdge.cost);

currentVertex = dest; // 更新当前顶点为这条边的目的地

edge = NodeTable[dest].adj;

while (edge != NULL) {

if (!visited[edge->dest]) heap.push(\*edge);

edge = edge->link;

}

}

}

void graphTree::printMST()

{

std::cout << "最小生成树的顶点及边为：" << std::endl;

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

char src = getValue(mstEdges[i].source);

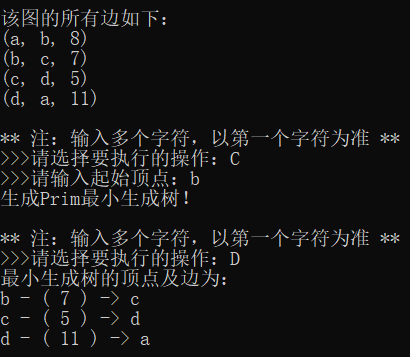
char dst = getValue(mstEdges[i].dest);

std::cout << src << " - ( " << mstEdges[i].cost << " ) -> " << dst << std::endl;

}

}

### 3.3.3 构造并显示最小生成树实现示例



## 本项目涉及Graph.h内容的主要实现

### 3.4.1 最小堆和邻接表类的定义

实现了一个最小堆数据结构，用于存储元素并能够快速访问最小元素。堆中的元素通过动态数组（heap）进行管理，数组大小由容量（capacity）定义。提供了基本的堆操作，包括插入新元素（push）、获取堆顶元素（top）、删除堆顶元素（pop）和检查堆是否为空（empty）。

表示一个图，可以是无向图或有向图。包含顶点和边的数量（numVertices 和 numEdges）、顶点表（NodeTable，一个数组，每个元素是一个 Vertex 类型的对象），以及最小生成树（MST）相关的成员。提供了图的基本操作，如插入和删除顶点（insertVertex、removeVertex）、插入和删除边（insertEdge、removeEdge）、获取顶点和边的信息（getValue、getWeight、getFirstNeighbor、getNextNeighbor）等。

### 3.4.2 Graph.h核心代码

详见类设计。

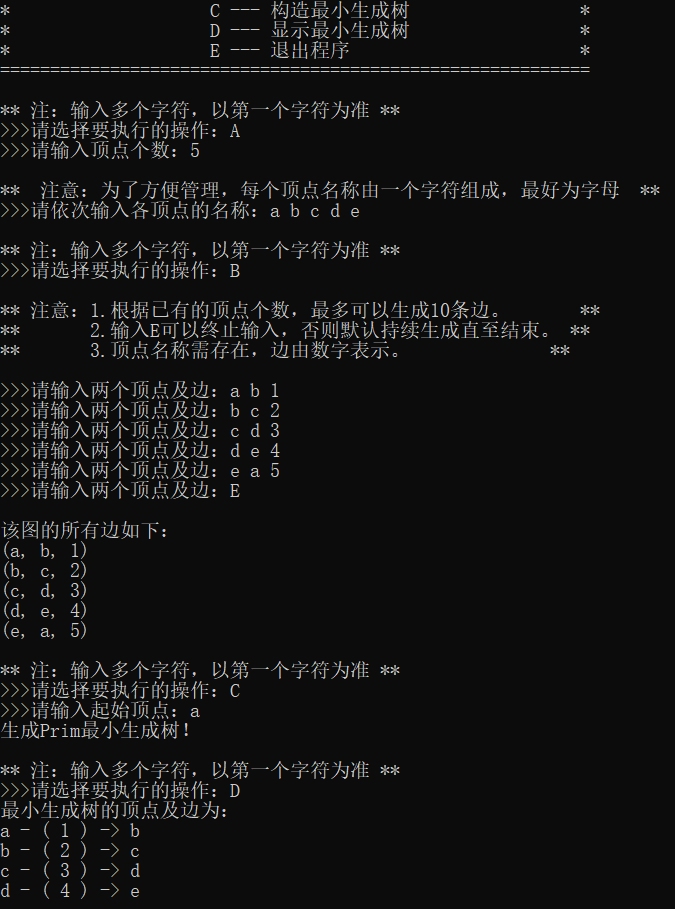
# 项目测试

## 功能测试

### 基本功能测试

**测试用例：**顶点个数为5，顶点名称为a,b,c,d,e；添加边 A-B, B-C, C-D, D-E, E-A，边的权重分别为 1, 2, 3, 4, 5。从顶点 A 开始构建最小生成树。

**实验结果：**

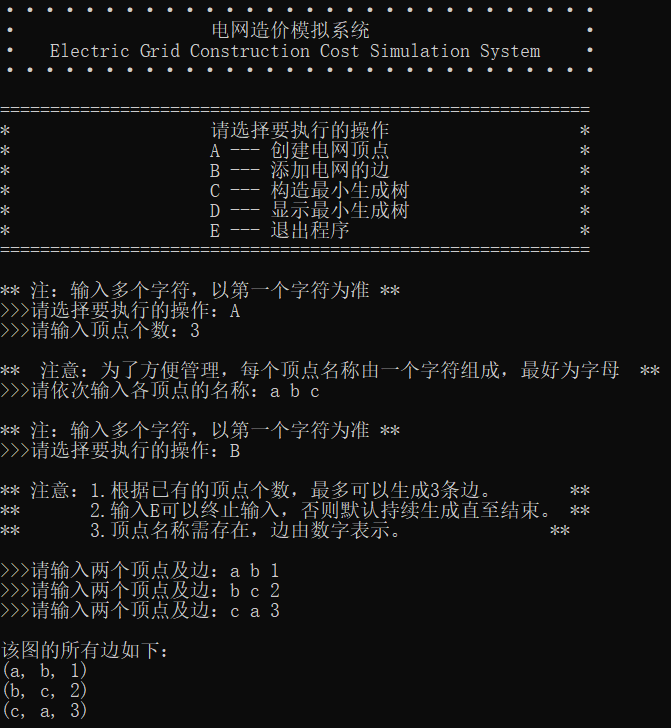


### 4.1.2 边界条件测试

**测试用例：**测试输入边数大于等于最大边

**预期结果：**在满最大边后，系统自动完成输入，不会有超过最大边的情况发生。一般情况下需要输入E才能结束输入。

**实验结果：**



## 错误测试

### 输入验证错误

#### 4.2.1.1 输入验证错误判断思路

测试添加已存在的顶点、测试添加已存在的边、测试添加不存在顶点的边。

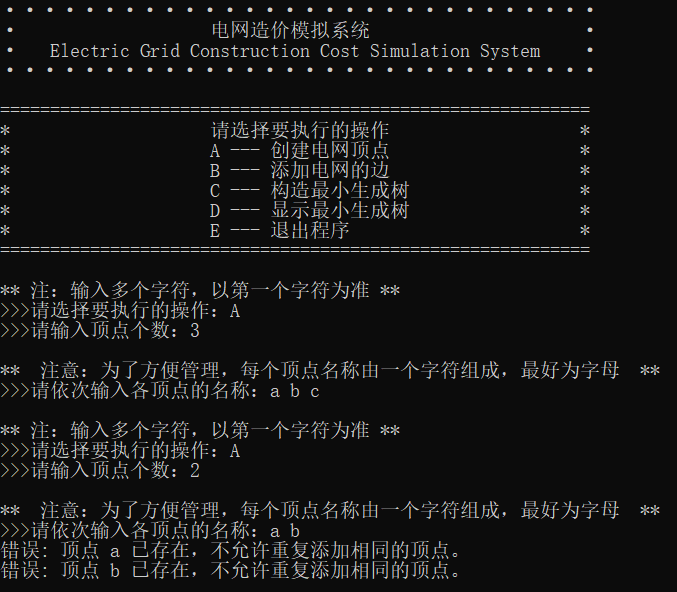
#### 4.2.1.3输入示例

**测试用例：**尝试再次添加顶点 A

**用例说明：**测试添加已存在的顶点；

**预期结果：**程序应提供错误信息，指出顶点已存在。

**实验结果：**

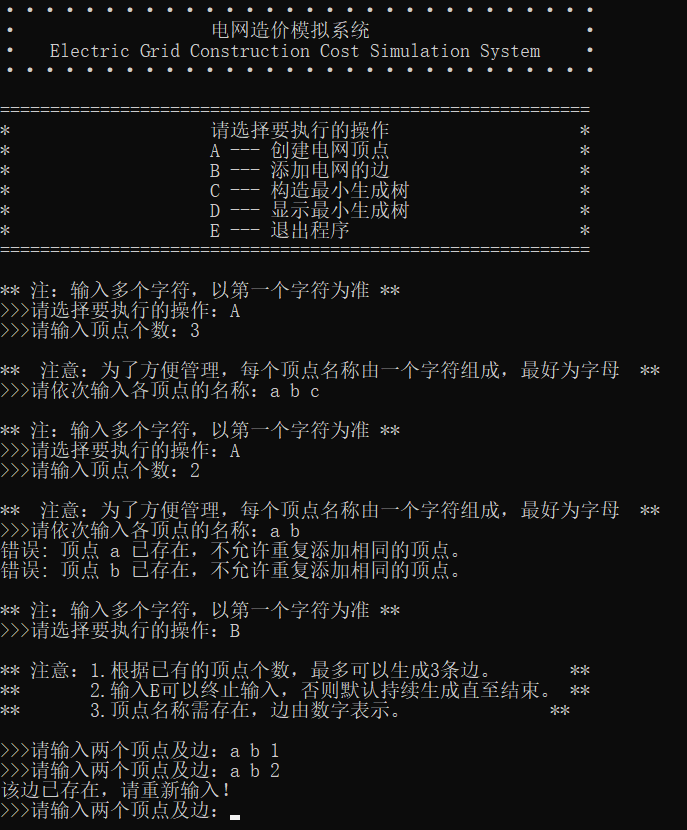


**测试用例：**尝试再次添加边 A-B

**用例说明：**测试添加已存在的边

**预期结果：**程序应提供错误信息，指出边已存在。

**实验结果：**

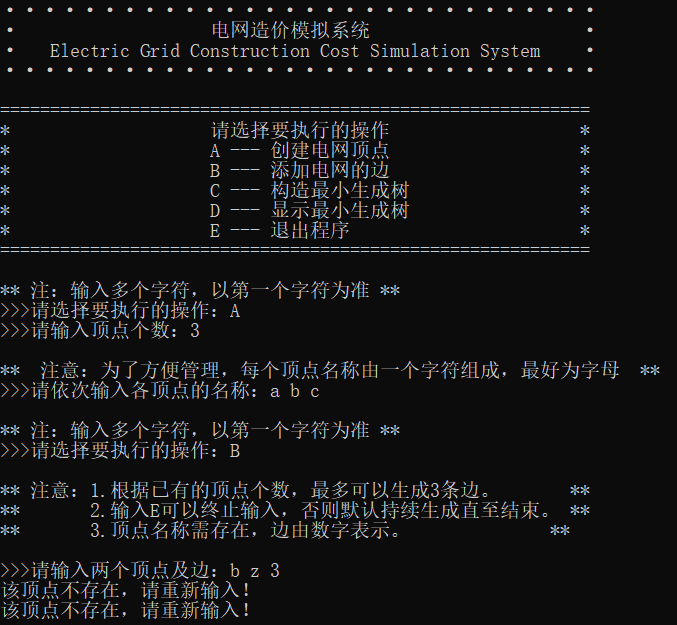


**测试用例：**尝试添加边 A-Z（假设 Z 顶点不存在）

**用例说明：**测试添加不存在顶点的边

**预期结果：**程序应提供错误信息，指出顶点不存在。

**实验结果：**



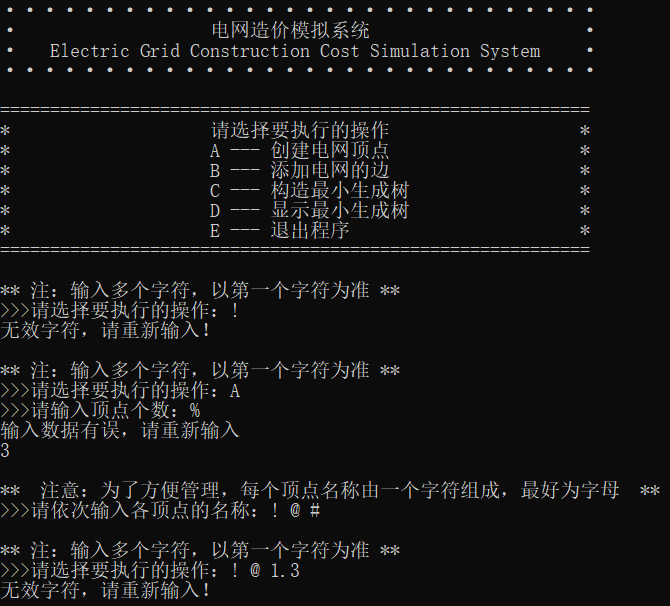
### 异常情况处理

**测试用例：**abc/ @#￥ / 五

**用例说明：**测试非法输入（如字母）

**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：**

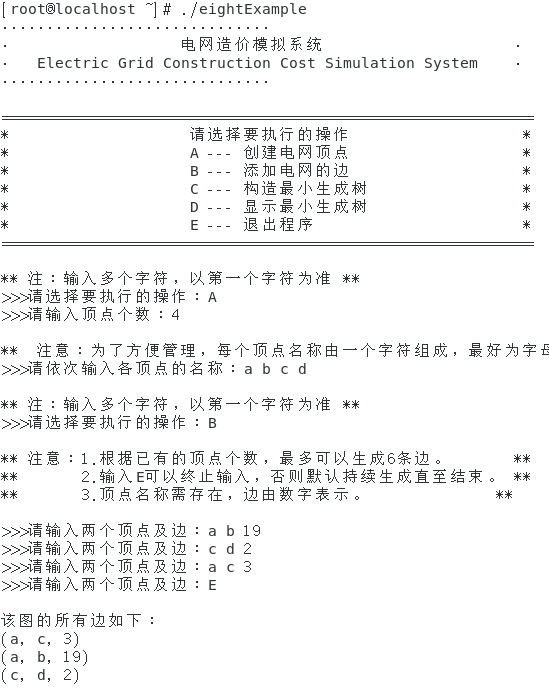


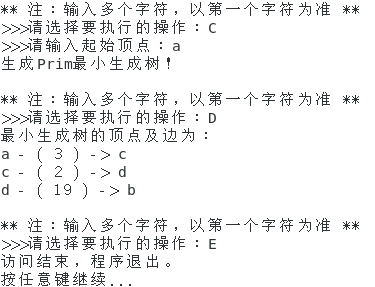
## Linux环境测试

**编译命令：** g++ -std=c++11 '/mnt/hgfs/ShareFolder/ 08\_electric\_grid\_construction\_cost\_simulation\_system' -o eightExample

**运行命令：** ./eightExample

**实验结果：**





# 集成开发环境与编译运行环境

Windows系统：Windows 11 x64

Windows集成开发环境：Microsoft Visual Studio 2022 (Release模式)

Windows编译运行环境：本项目适用于x86架构和x64架构

Linux系统：CentOS 7 x64