****

项目说明文档

**数据结构课程设计**

**——电网建设造价模拟系统**

**培养单位：软件学院**

**本 科 生：蓝 笙 聆**

**学 号：1951096**

**指导老师：张 颖**

二○二○年十二月

目录

[第1章 分析 1](#_Toc60007913)

[1.1 背景分析 1](#_Toc60007914)

[1.2 功能分析 1](#_Toc60007915)

[第2章 设计 2](#_Toc60007916)

[2.1 数据结构设计 2](#_Toc60007917)

[2.2 类结构设计 2](#_Toc60007918)

[2.3 成员与操作设计 2](#_Toc60007919)

[2.4 系统设计 4](#_Toc60007920)

[第3章 实现 5](#_Toc60007921)

[3.1 插入点功能的实现 5](#_Toc60007922)

[3.1.1 插入点功能流程图 5](#_Toc60007923)

[3.1.2 插入点功能核心代码 5](#_Toc60007924)

[3.1.3 插入点功能截屏示例 7](#_Toc60007925)

[3.2 插入边功能的实现 8](#_Toc60007926)

[3.2.1 插入边功能流程图 8](#_Toc60007927)

[3.2.2 插入边代码实现 8](#_Toc60007928)

[3.2.3 插入边功能截屏示例 10](#_Toc60007929)

[3.3 删除点功能的实现 10](#_Toc60007930)

[3.3.1 删除点功能流程图 10](#_Toc60007931)

[3.3.2 删除点代码实现 11](#_Toc60007932)

[3.3.2 删除点功能截屏示例 12](#_Toc60007933)

[3.4 删除边功能的实现 12](#_Toc60007934)

[3.4.1 删除边功能流程图 12](#_Toc60007935)

[3.4.2 删除边代码实现 13](#_Toc60007936)

[3.4.3 删除边功能截屏示例 15](#_Toc60007937)

[3.5 Prim算法生成最小生成树功能的实现 16](#_Toc60007938)

[3.5.1 Prim算法生成最小生成树功能流程图 16](#_Toc60007939)

[3.5.2 Prim算法生成最小生成树功能代码实现 16](#_Toc60007940)

[3.5.3 Prim算法生成最小生成树功能截屏示例 18](#_Toc60007941)

[3.6 Kruskal算法生成最小生成树功能的实现 19](#_Toc60007942)

[3.6.1 Kruskal算法生成最小生成树功能流程图 19](#_Toc60007943)

[3.6.2 Kruskal算法生成最小生成树功能代码实现 19](#_Toc60007944)

[3.6.3 Kruskal算法生成最小生成树功能截屏示例 21](#_Toc60007945)

[3.7 总体系统的实现 22](#_Toc60007946)

[3.7.1 总体系统流程图 22](#_Toc60007947)

[3.7.2 总体系统代码实现 22](#_Toc60007948)

[3.7.3 总体功能截屏示例 25](#_Toc60007949)

[第4章 测试 26](#_Toc60007950)

[4.1 功能测试 26](#_Toc60007951)

[4.1.1 插入点功能测试 26](#_Toc60007952)

[4.1.2 插入边功能测试 26](#_Toc60007953)

[4.1.3 删除点功能测试 27](#_Toc60007954)

[4.1.4 删除边功能测试 27](#_Toc60007955)

[4.1.5 Prim算法生成最小生成树功能测试 28](#_Toc60007956)

[4.1.6 Kruskal算法生成最小生成树功能测试 28](#_Toc60007957)

[4.2 边界测试 29](#_Toc60007958)

[4.2.1 删除最后一个点 29](#_Toc60007959)

[4.2.2 对唯一一个点生成最小生成树 29](#_Toc60007960)

[4.3 出错测试 30](#_Toc60007961)

[4.3.1 删除点不存在 30](#_Toc60007962)

[4.3.2 插入点已存在 30](#_Toc60007963)

[4.3.3 删除边不存在 30](#_Toc60007964)

[4.3.4 插入边已存在 31](#_Toc60007965)

[4.3.5 对空图生成最小生成树 31](#_Toc60007966)

[4.3.6 对非连通图生成最小生成树 31](#_Toc60007967)

[4.3.7 操作码错误 32](#_Toc60007968)

# 第1章 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。

随着计算机科学技术的不断成熟，使用电网造价模拟系统进行模拟，具有手工计算所无法比拟的优势。这些优点能够极大地提高效率，也是社会走向信息化、科学化、国际化的重要条件。因此，开发一套电网造价模拟系统具有十分重要的意义。

## 1.2 功能分析

作为一个最简易的电网造价模拟系统，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。

# 第2章 设计

## 2.1 数据结构设计

如上功能分析所述，该系统要求作图并生成最小生成树，故考虑使用图。

## 2.2 类结构设计

经典的图一般包括三个抽象数据类型（ADT）——图类（Graph <T>）、点类（Vertex <T>）与边类（Edge <T>），而三个类之间的耦合关系可以采用嵌套、继承等多种关系。为方便处理，本系统采用模板struct结构体描述树点类（Vertex <T>）与边类（Edge <T>），这样使得图类（Graph <T>）可以访问链表结点。

## 2.3 成员与操作设计

**点结构体（Vertex）**

template <class T>

struct vertex {

    T val;

    vertex(const T& val) : val(val){};

    vertex() : val(){};

    bool operator==(const vertex& v2) const { return (this->val == v2.val); };

    bool operator==(const T& v2) const { return (this->val == v2); };

    bool operator!=(const vertex& v2) { return !(\*this == v2); };

    bool operator!=(const T& v2) { return !(\*this == v2); };

    bool operator<(const vertex& v2) const { return (this->val < v2.val); };

    bool operator>(const vertex& v2) const { return (this->val > v2.val); };

    void operator=(const vertex& v2) { this->val = v2.val; };

    void operator=(const T& v2) { this->val = v2; };

};

**边结构体（Edge）**

template <class T, class E>

struct edge {

    E val;

    vertex<T> v1;

    vertex<T> v2;

    edge(const vertex<T> v1, const vertex<T> v2, const E val)

        : v1(v1), v2(v2), val(val){};

    edge() : v1(), v2(), val(){};

    bool operator==(const edge& e2) {

        return ((this->v1 == e2.v1 && this->v2 == e2.v2) ||

                (this->v1 == e2.v2 && this->v2 == e2.v1));

    };

    bool operator!=(const edge& e2) { return !(\*this == e2); };

    bool operator<(const edge& e2) const { return (this->val < e2.val); };

    bool operator>(const edge& e2) const { return (this->val > e2.val); };

};

**图类（Graph）**

template <class T, class E>

class Graph {

   public:

    void InsertVertex(vertex<T>& v);

    void InsertVertex(const T val);

    void DelVertex(vertex<T>& v);

    void DelVertex(const T val);

    void InsertEdge(vertex<T> v1, vertex<T> v2, E cost);

    void InsertEdge(T v1, T v2, E cost);

    void InsertEdge(edge<T, E> edge);

    void RemoveEdge(vertex<T> v1, vertex<T> v2);

    void RemoveEdge(T v1, T v2);

    void RemoveEdge(edge<T, E> edge);

    void PrintGraph(ostream& os);

    int NumVertices() const { return \_vertices.size(); };

    int NumEdges() const { return \_edges.size() / 2; }

    bool InGraph(T v);

    bool InGraph(vertex<T> v);

    bool InGraph(T v1, T v2);

    bool InGraph(vertex<T> v1, vertex<T> v2);

    bool InGraph(edge<T, E> edge);

    Graph<T, E> Prim(vertex<T> v);

   private:

    vector<vertex<T>> \_vertices;

    map<pair<vertex<T>, vertex<T>>, edge<T, E>> \_edges;

};

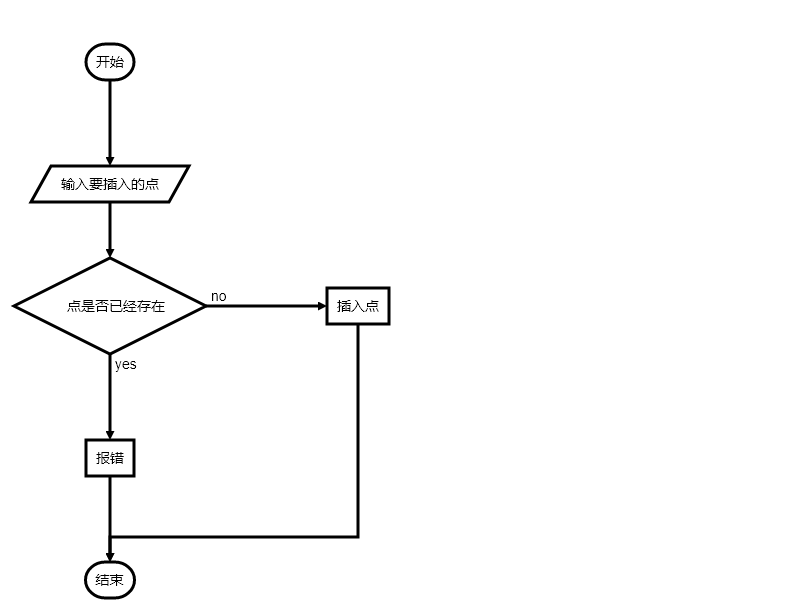
## 2.4 系统设计

系统首先调用opening ()函数实现对屏幕的初始化，创建图g，然后进入loop ()循环，根据用户所输入的操作码（operatorCode）执行图g对应的成员函数。

# 第3章 实现

## 3.1 插入点功能的实现

### 3.1.1 插入点功能流程图



### 3.1.2 插入点功能核心代码

template <class T, class E>

void Graph<T, E>::InsertVertex(vertex<T>& v) {

    if (InGraph(v)) throw "Error: Vertex already in the graph. ";

    auto it = find(\_vertices.begin(), \_vertices.end(), v);

    \_vertices.push\_back(v);

};

template <class T, class E>

void Graph<T, E>::InsertVertex(const T val) {

    if (InGraph(val)) throw "Error: Vertex already in the graph. ";

    vertex<T> vex(val);

    auto it = find(\_vertices.begin(), \_vertices.end(), vex);

    \_vertices.push\_back(vex);

};

void addVertex(Graph<string, int>& g) {

    cout << "Please input the number of vertices:";

    int num = 0;

    while (true) {

        string temp;

        cin >> temp;

        if (!isNum(temp))

            cout << "Error: Invalid input detected. Please try again." << endl;

        else {

            num = str2num(temp);

            break;

        }

    }

    cout << "Please enter the name of vertices in turn.";

    for (int i = 0; i < num; i++) {

        string v1 = "";

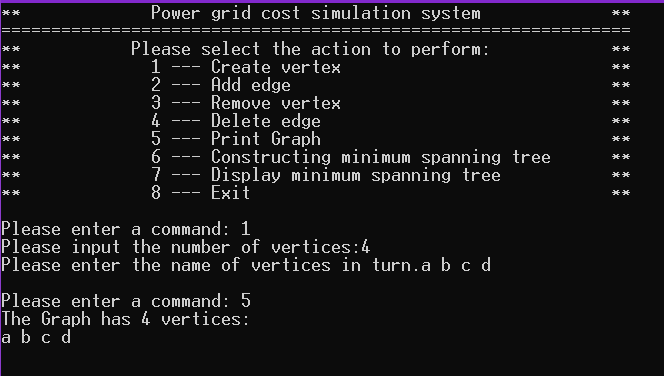
        cin >> v1;

        g.insertVertex(v1);

    }

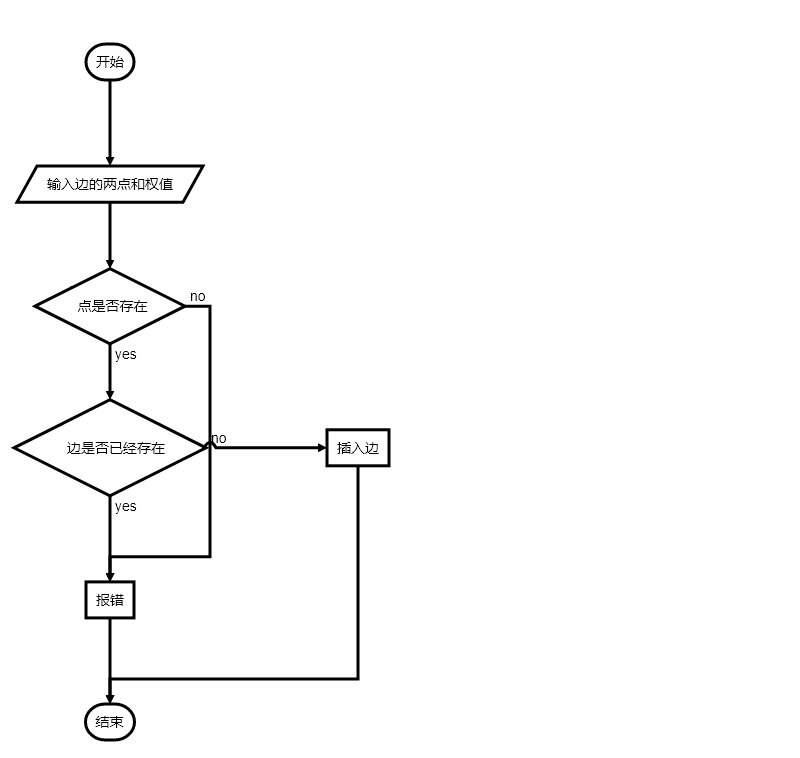
}

### 3.1.3 插入点功能截屏示例



## 3.2 插入边功能的实现

### 3.2.1 插入边功能流程图



### 3.2.2 插入边代码实现

template <class T, class E>

void Graph<T, E>::InsertEdge(vertex<T> v1, vertex<T> v2, E cost) {

    if (!InGraph(v1) || !InGraph(v2))

        throw "Error: Vertex not found. ";

    if (InGraph(v1, v2)) throw "Error: Edge already in the graph. ";

    auto it1 = find(\_vertices.begin(), \_vertices.end(), v1);

    auto it2 = find(\_vertices.begin(), \_vertices.end(), v2);

    edge<T, E> e(v1, v2, cost);

    auto ite = \_edges.find({\*it1, \*it2});

    \_edges[{v1, v2}] = e;

    \_edges[{v2, v1}] = e;

};

template <class T, class E>

void Graph<T, E>::InsertEdge(T v1, T v2, E cost) {

    if (!InGraph(v1) || !InGraph(v2))

        throw "Error: Vertex not found. ";

    if (InGraph(v1, v2)) throw "Error: Edge already in the graph. ";

    auto it1 = find(\_vertices.begin(), \_vertices.end(), v1);

    auto it2 = find(\_vertices.begin(), \_vertices.end(), v2);

    edge<T, E> e(\*it1, \*it2, cost);

    auto ite = \_edges.find({\*it1, \*it2});

    \_edges[{\*it1, \*it2}] = e;

    \_edges[{\*it2, \*it1}] = e;

};

template <class T, class E>

void Graph<T, E>::InsertEdge(edge<T, E> edge) {

    if (InGraph(edge)) throw "Error: Edge already in the graph. ";

    auto it = \_edges.find({edge.v1, edge.v2});

    \_edges[{edge.v1, edge.v2}] = edge;

    \_edges[{edge.v2, edge.v1}] = edge;

};

void addEdge(Graph<string, int>& g) {

    cout << "Please input the number of edges: ";

    int num = 0;

    while (true) {

        string temp;

        cin >> temp;

        if (!isNum(temp))

            cout << "Error: Invalid input detected. Please try again." << endl;

        else {

            num = str2num(temp);

            break;

        }

    }

    for (int i = 0; i < num; i++) {

        cout << "Please enter two vertices and edge lengths.";

        string v2 = "";

        string v1 = "";

        int cost = 0;

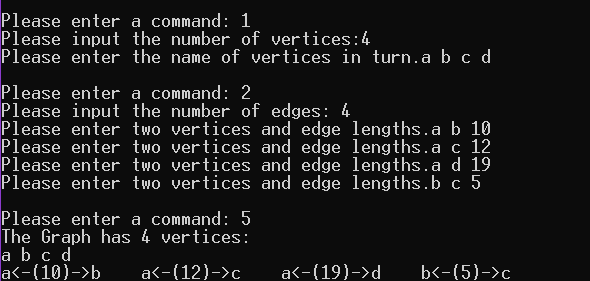
        cin >> v1 >> v2 >> cost;

        g.insertEdge(v1, v2, cost);

    }

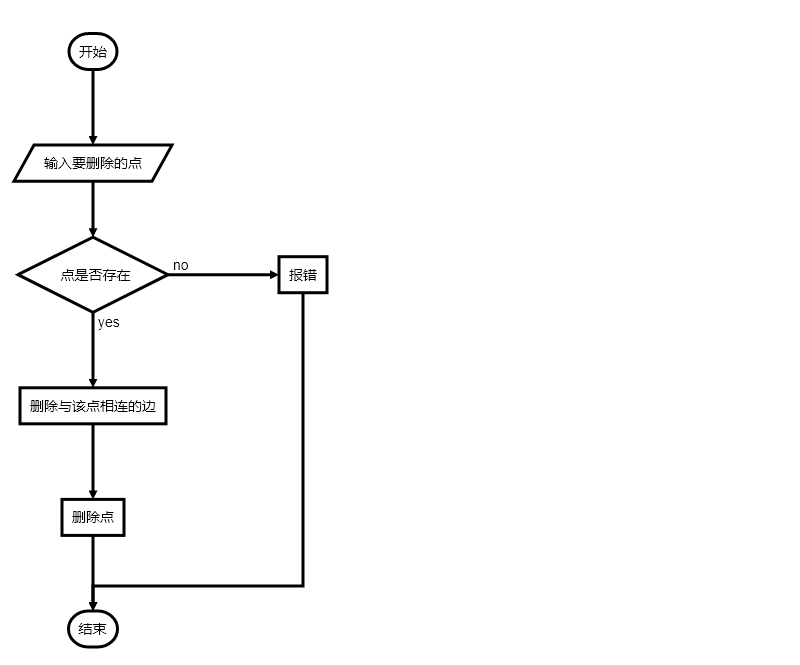
}

### 3.2.3 插入边功能截屏示例



## 3.3 删除点功能的实现

### 3.3.1 删除点功能流程图



### 3.3.2 删除点代码实现

template <class T, class E>

void Graph<T, E>::DelVertex(vertex<T>& vex) {

    if (!InGraph(vex)) throw "Error: Vertex not found. ";

    auto it = find(\_vertices.begin(), \_vertices.end(), vex);

    for (auto&& i : \_vertices) {

        if (InGraph(vex, i)) RemoveEdge(vex, i);

    }

    vector<vertex<T>> v;

    for (auto&& i : this->\_vertices)

        if (i != it) v.push\_back(i);

    this->\_vertices = v;

};

template <class T, class E>

void Graph<T, E>::DelVertex(const T val) {

    if (!InGraph(val)) throw "Error: Vertex not found. ";

    vertex<T> vex(val);

    auto it = find(\_vertices.begin(), \_vertices.end(), vex);

    for (auto&& i : \_vertices) {

        if (InGraph(vex, i)) RemoveEdge(vex, i);

    }

    vector<vertex<T>> v;

    for (auto&& i : \_vertices)

        if (i != \*it) v.push\_back(i);

    this->\_vertices = v;

};

void delVex(Graph<string, int>& g) {

    string temps;

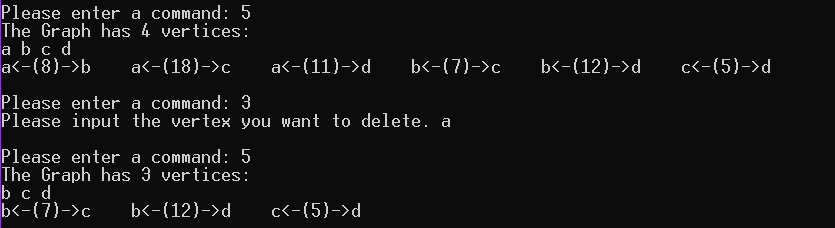
    cout << "Please input the vertex you want to delete. ";

    cin >> temps;

    g.delVertex(temps);

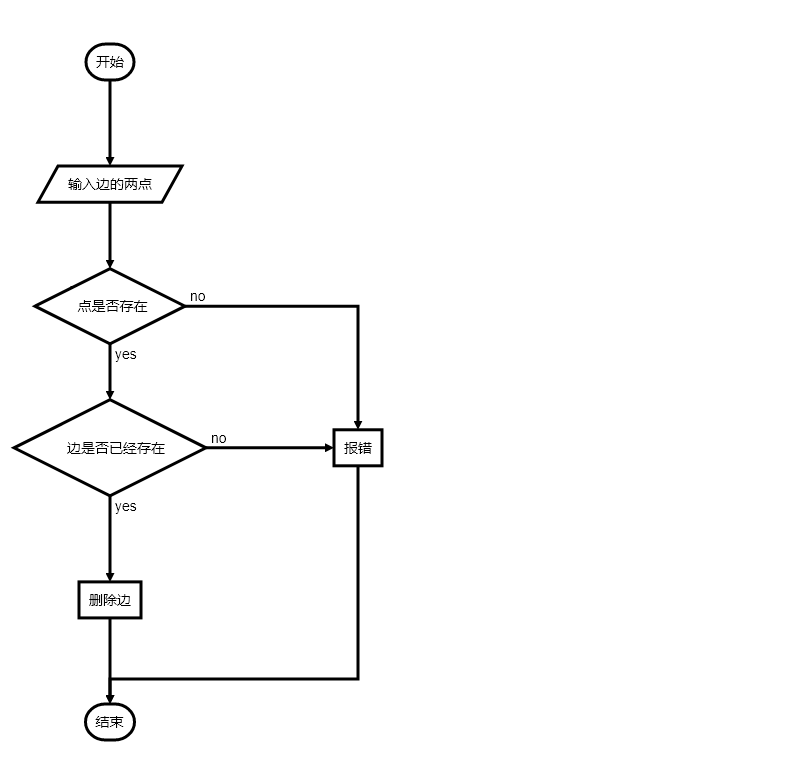
}

### 3.3.2 删除点功能截屏示例



## 3.4 删除边功能的实现

### 3.4.1 删除边功能流程图



### 3.4.2 删除边代码实现

template <class T, class E>

void Graph<T, E>::RemoveEdge(vertex<T> v1, vertex<T> v2) {

    if (!InGraph(v1, v2)) throw "Error: Edge not found. ";

    auto it1 = find(\_vertices.begin(), \_vertices.end(), v1);

    auto it2 = find(\_vertices.begin(), \_vertices.end(), v2);

    auto ite1 = \_edges.find({\*it1, \*it2});

    \_edges.erase(ite1);

    auto ite2 = \_edges.find({\*it2, \*it1});

    \_edges.erase(ite2);

};

template <class T, class E>

void Graph<T, E>::RemoveEdge(T v1, T v2) {

    if (!InGraph(v1, v2)) throw "Error: Edge not found. ";

    auto it1 = find(\_vertices.begin(), \_vertices.end(), v1);

    auto it2 = find(\_vertices.begin(), \_vertices.end(), v2);

    auto ite1 = \_edges.find({\*it1, \*it2});

    \_edges.erase(ite1);

    auto ite2 = \_edges.find({\*it2, \*it1});

    \_edges.erase(ite2);

};

template <class T, class E>

void Graph<T, E>::RemoveEdge(edge<T, E> edge) {

    if (!InGraph(edge)) throw "Error: Edge not found. ";

    auto it1 = \_edges.find({edge.v1, edge.v2});

    \_edges.erase(it1);

    auto it2 = \_edges.find({edge.v2, edge.v1});

    \_edges.erase(it2);

};

template <class T, class E>

void Graph<T, E>::removeEdge(edge<T, E> edge) {

    if (!inGraph(edge)) throw "Error: Edge not found. ";

    auto it1 = edges.find({edge.v1, edge.v2});

    edges.erase(it1);

    auto it2 = edges.find({edge.v2, edge.v1});

    edges.erase(it2);

};

void delEdge(Graph<string, int>& g) {

    cout << "Please input the edge you want to delete " << endl;

    string temp1;

    string temp2;

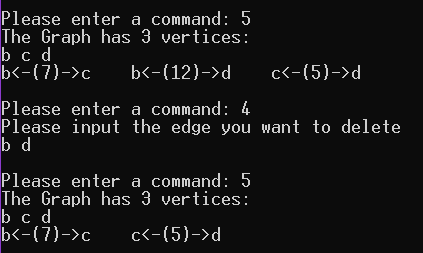
    cin >> temp1;

    cin >> temp2;

    g.removeEdge(temp1, temp2);

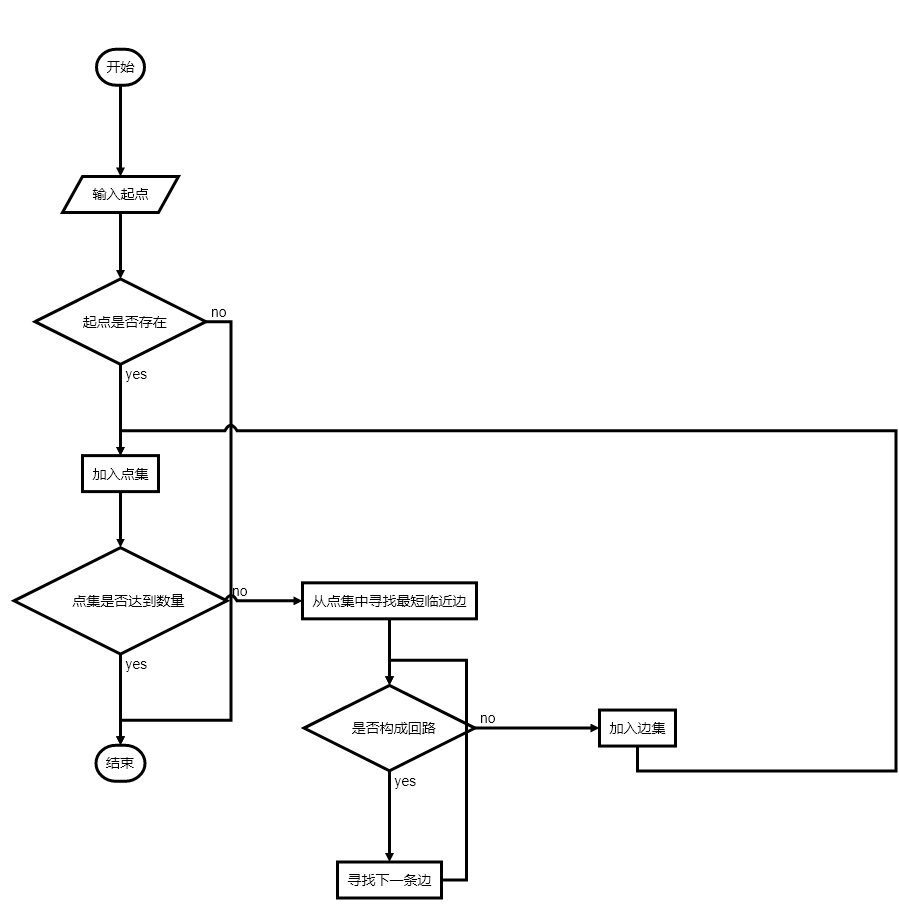
}

### 3.4.3 删除边功能截屏示例



## 3.5 Prim算法生成最小生成树功能的实现

### 3.5.1 Prim算法生成最小生成树功能流程图



### 3.5.2 Prim算法生成最小生成树功能代码实现

template <class T, class E>

Graph<T, E> Graph<T, E>::Prim(vertex<T> v) {

    auto it = find(this->\_vertices.begin(), this->\_vertices.end(), v);

    if (it == this->\_vertices.end()) throw "Error: Vertex not found.";

    Graph<T, E> mst;

    mst.InsertVertex(v);

    while (mst.NumVertices() != this->NumVertices()) {

        E min = 999999;

        edge<T, E> emin;

        for (auto&& j : mst.\_vertices) {

            for (auto&& i : this->\_vertices) {

                if (i == j) continue;

                auto ite = this->\_edges.find({j, i});

                auto itg = mst.\_edges.find({j, i});

                if (ite == this->\_edges.end()) continue;

                if (itg != mst.\_edges.end()) continue;

                bool bflag = false;

                int temp1 = 1;

                int temp2 = 1;

                for (auto&& k : mst.\_vertices) {

                    if (k == this->\_edges[{j, i}].v1) temp1++;

                    if (k == this->\_edges[{j, i}].v2) temp2++;

                }

                if (temp1 >= 2 && temp2 >= 2) continue;

                auto temp = min;

                if (this->\_edges[{j, i}].val < min)

                    min = this->\_edges[{j, i}].val;

                if (temp != min) emin = this->\_edges[{j, i}];

            }

        }

        if (!mst.InGraph(emin.v1)) mst.InsertVertex(emin.v1);

        if (!mst.InGraph(emin.v2)) mst.InsertVertex(emin.v2);

        mst.InsertEdge(emin);

    }

    return mst;

}

void constructPrimMst(Graph<string, int>& g, Graph<string, int>& mst) {

    vertex<string> v;

    string p = "";

    cout << "Please enter the starting vertex: ";

    cin >> p;

    v = p;

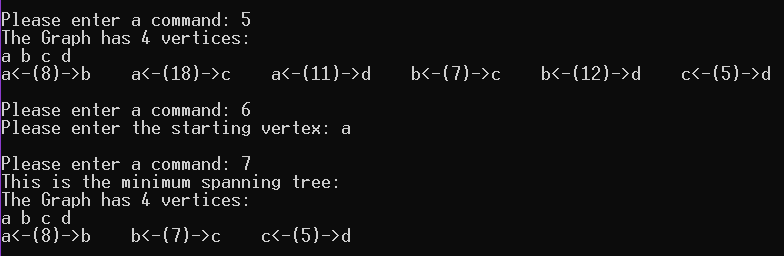
    mst = g.Prim(v);

    cout << "This is the Prim minimum spanning tree: " << endl;

    mst.PrintGraph(cout);

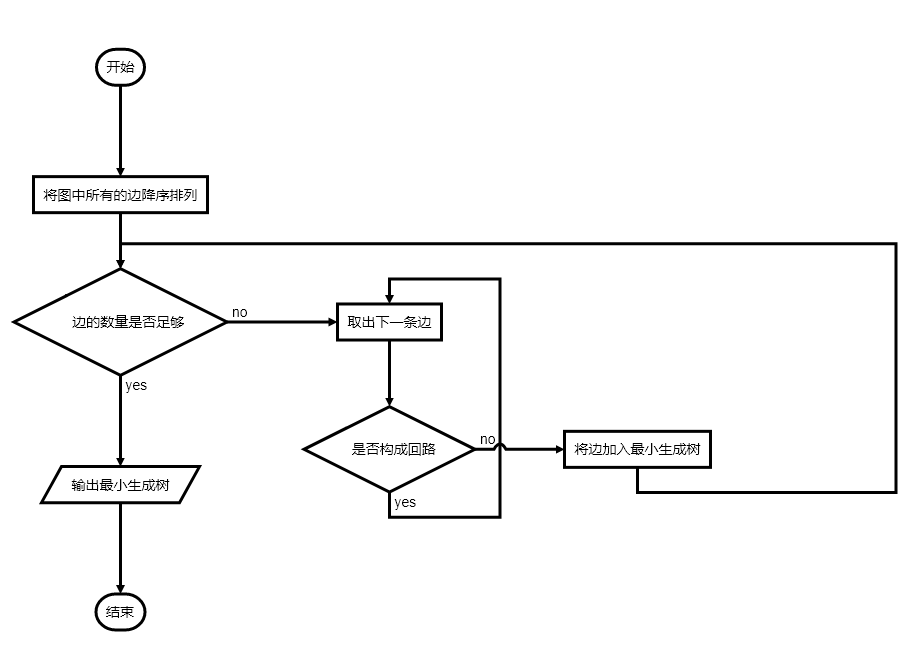
}

### 3.5.3 Prim算法生成最小生成树功能截屏示例



## 3.6 Kruskal算法生成最小生成树功能的实现

### 3.6.1 Kruskal算法生成最小生成树功能流程图



### 3.6.2 Kruskal算法生成最小生成树功能代码实现

template <class T>

T unionFind(map<T, T> m, T x) {

    if (m[x] == x)

        return x;

    else

        m[x] = unionFind(m, m[x]);

    return m[x];

}

template <class T, class E>

Graph<T, E> Graph<T, E>::Kruskal() {

    Graph<T, E> mst;

    if (NumVertices() == 1) {

        mst.InsertVertex(\_vertices[0]);

        return mst;

    }

    vector<edge<T, E>> e;

    for (auto&& i : \_vertices) {

        for (auto&& j : \_vertices) {

            if (i == j) continue;

            auto it = \_edges.find({i, j});

            if (it == \_edges.end()) continue;

            bool cflag = false;

            for (int k = 0; k < e.size(); k++)

                if (e[k] == \_edges[{i, j}]) {

                    cflag = true;

                    break;

                }

            if (cflag) continue;

            e.push\_back(\_edges[{i, j}]);

        }

    }

    sort(e.begin(), e.end());

    map<vertex<T>, vertex<T>> ufsets;

    for (auto&& i : this->\_vertices) ufsets[i] = i;

    for (auto&& i : e) {

        if (unionFind(ufsets, i.v1) == unionFind(ufsets, i.v2)) continue;

        ufsets[unionFind(ufsets, i.v1)] = unionFind(ufsets, i.v2);

        if (!mst.InGraph(i.v1)) mst.InsertVertex(i.v1);

        if (!mst.InGraph(i.v2)) mst.InsertVertex(i.v2);

        mst.InsertEdge(i);

    }

    if (mst.NumEdges() != this->NumVertices() - 1)

        throw "Error: The graph does not have a minimum spanning tree. ";

    return mst;

}

void constructKruskalMst(Graph<string, int>& g, Graph<string, int>& mst) {

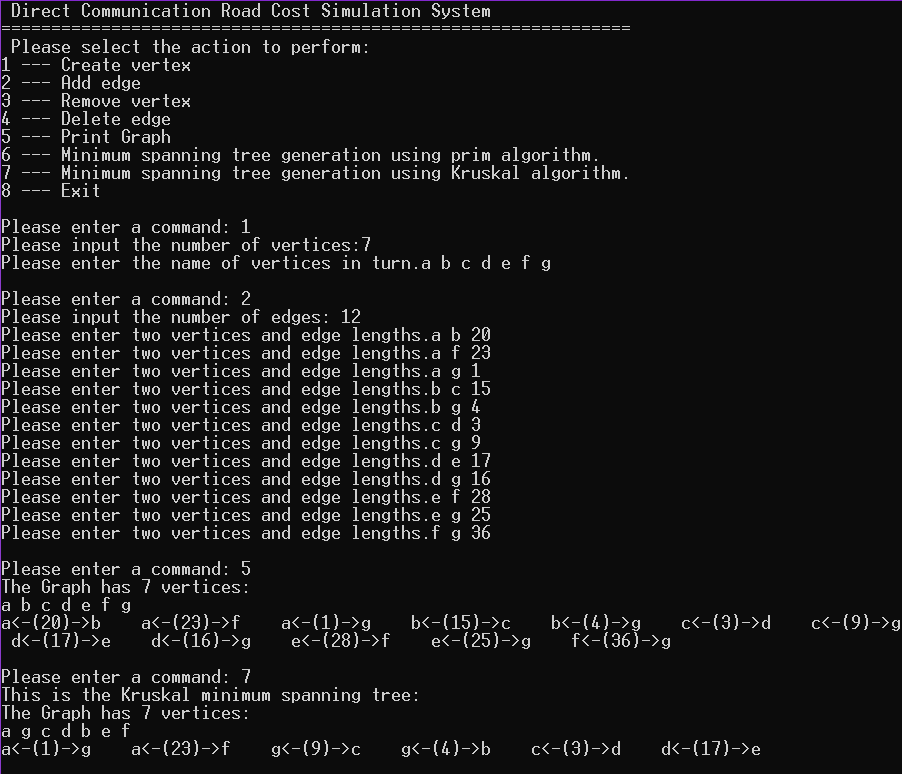
    mst = g.Kruskal();

    cout << "This is the Kruskal minimum spanning tree: " << endl;

    mst.PrintGraph(cout);

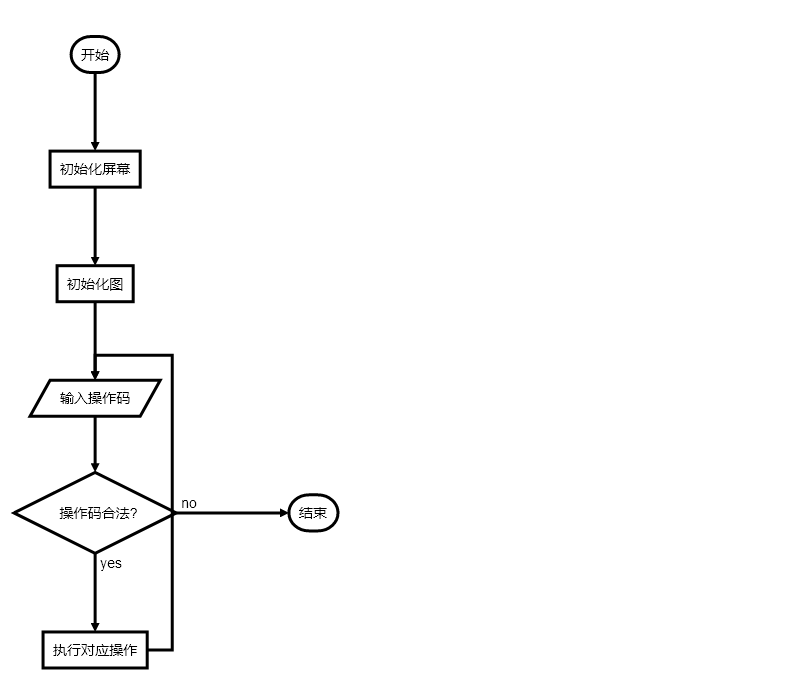
}

### 3.6.3 Kruskal算法生成最小生成树功能截屏示例



## 3.7 总体系统的实现

### 3.7.1 总体系统流程图



### 3.7.2 总体系统代码实现

void opening() {

    cout << " Power grid cost simulation system " << endl

         << "==============================================================="

         << endl

         << " Please select the action to perform:" << endl

         << "1 --- Create vertex " << endl

         << "2 --- Add edge" << endl

         << "3 --- Remove vertex " << endl

         << "4 --- Delete edge " << endl

         << "5 --- Print Graph " << endl

         << "6 --- Minimum spanning tree generation using prim algorithm."

         << endl

         << "7 --- Minimum spanning tree generation using Kruskal algorithm."

         << endl

         << "8 --- Exit" << endl;

    return;

}

bool loop(Graph<string, int>& g, Graph<string, int>& mst) {

    cout << "\nPlease enter a command: ";

    char temp = '\0';

    cin >> temp;

    try {

        switch (temp) {

            case '1':

                // add vertices

                addVertex(g);

                return true;

            case '2':

                // add edges;

                addEdge(g);

                return true;

            case '3':

                // del vertex;

                delVex(g);

                return true;

            case '4':

                // del edge;

                delEdge(g);

                return true;

            case '5':

                // print graph

                g.PrintGraph(cout);

                return true;

            case '6':

                // Constructing minimum spanning tree;

                constructPrimMst(g, mst);

                return true;

            case '7':

                // Constructing minimum spanning tree;

                constructKruskalMst(g, mst);

                return true;

            default:

                return false;

        }

    } catch (const char\* e) {

        std::cerr << e << '\n';

        return true;

    }

}

int main() {

    opening();

    Graph<string, int> g;

    Graph<string, int> mst;

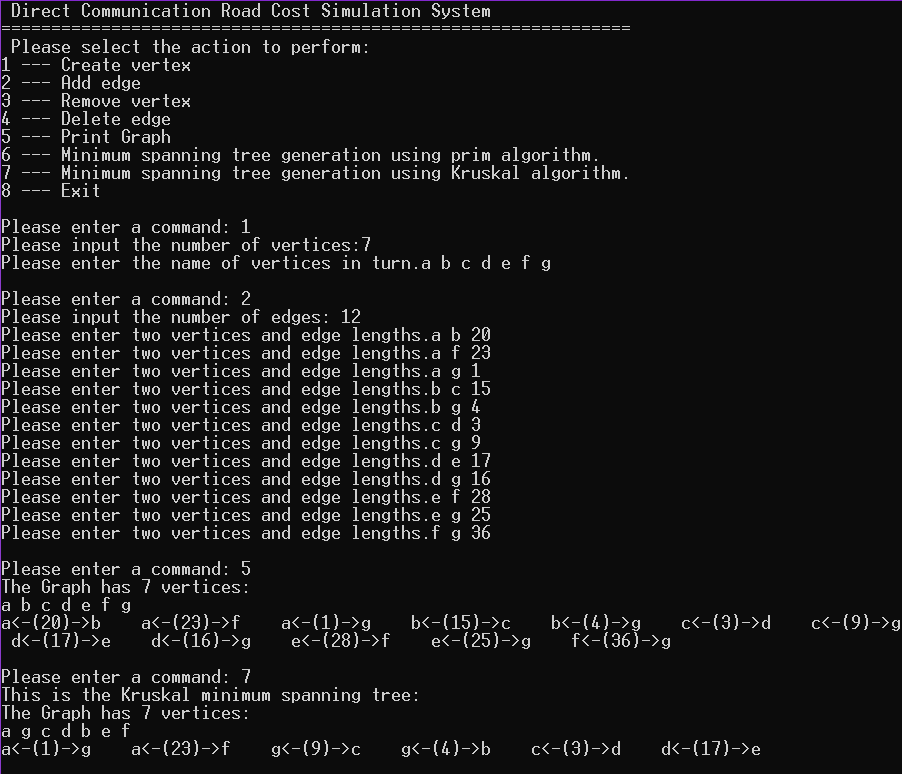
    while (loop(g, mst))

        ;

    return 0;

}

### 3.7.3 总体功能截屏示例



# 第4章 测试

## 4.1 功能测试

### 4.1.1 插入点功能测试

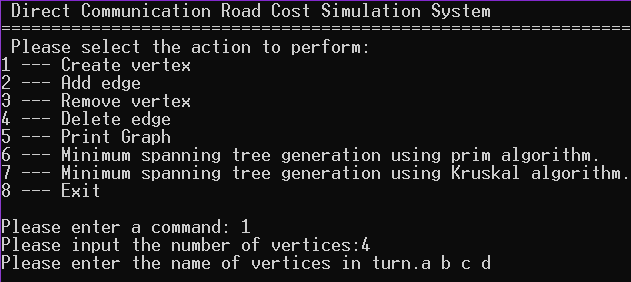
**测试用例**：

插入4个点：a，b，c，d

**预期结果**：

图里有4个点a，b，c，d

**实验结果**



### 4.1.2 插入边功能测试

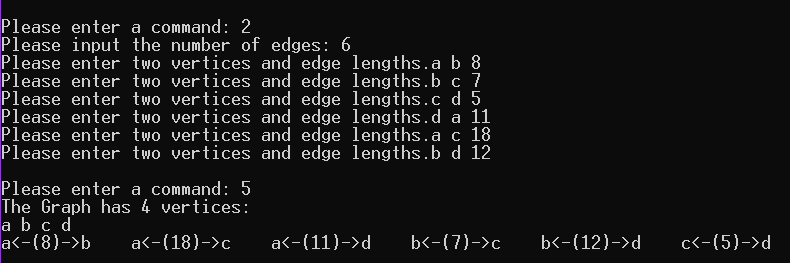
**测试用例**：

插入6条边

**预期结果**：

图里有6条边

**实验结果**



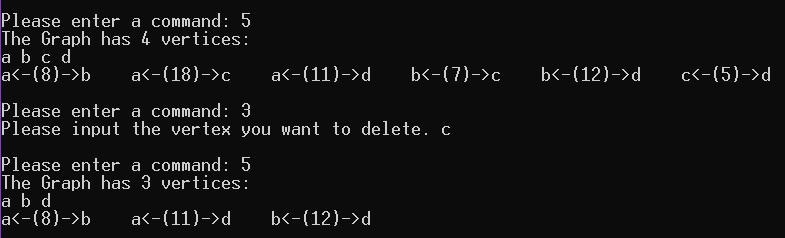
### 4.1.3 删除点功能测试

**测试用例：**删除c

**预期结果：**

删除c和与c相连的ac，bc，dc

**实验结果：**

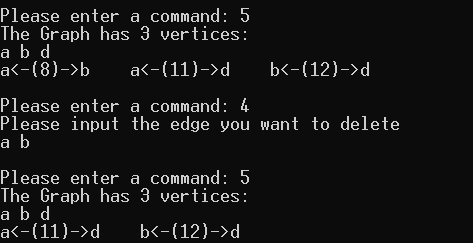


### 4.1.4 删除边功能测试

**测试用例：**删除边ab。

**预期结果：**删除边ab。

**实验结果：**



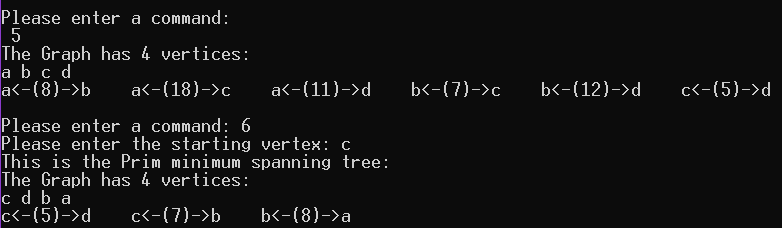
### 4.1.5 Prim算法生成最小生成树功能测试

**测试用例：**以c为起点生成最小生成树

**预期结果：**

c-5-d, c-7-b, b-8-a

**实验结果：**



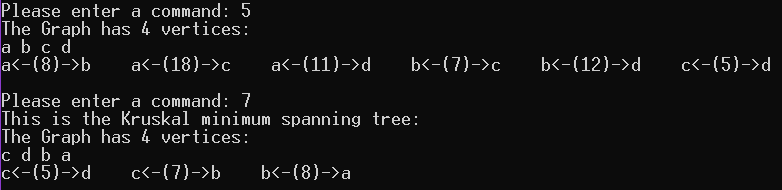
### 4.1.6 Kruskal算法生成最小生成树功能测试

**测试用例：**同上

**预期结果：**

c-5-d, c-7-b, b-8-a

**实验结果：**



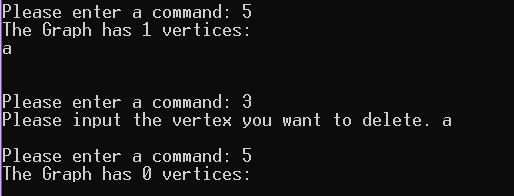
## 4.2 边界测试

### 4.2.1 删除最后一个点

**测试用例：**删除图中唯一个的点a。

**预期结果：**程序正常运行，不崩溃。

**实验结果：**

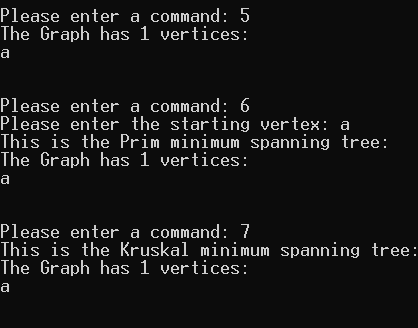


### 4.2.2 对唯一一个点生成最小生成树

**测试用例：**对图中唯一个的点a生成最小生成树。

**预期结果：**程序正常运行，不崩溃。

**实验结果：**



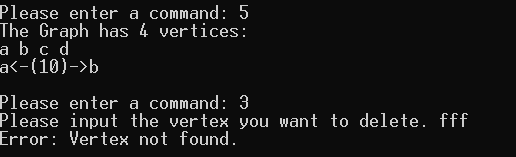
## 4.3 出错测试

### 4.3.1 删除点不存在

**测试用例：**删除不存在的点p0

**预期结果：**抛出异常，程序正常运行不崩溃。

**实验结果：**

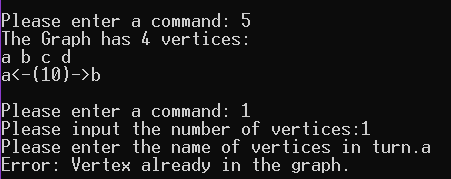


### 4.3.2 插入点已存在

**测试用例：**向已有点a的图中尝试插入a

**预期结果：**程序抛出异常，程序正常运行不崩溃。

**实验结果：**

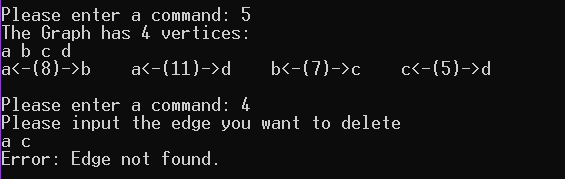


### 4.3.3 删除边不存在

**测试用例：**删除图中不存在的边ac

**预期结果：**程序抛出异常，程序正常运行不崩溃。

**实验结果：**

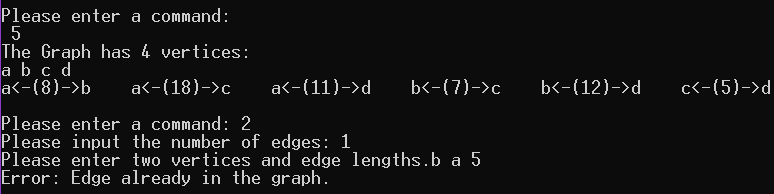


### 4.3.4 插入边已存在

**测试用例：**插入图中已存在的边ab

**预期结果：**程序抛出异常，程序正常运行不崩溃。

**实验结果：**

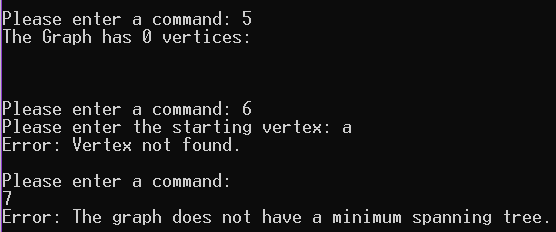


### 4.3.5 对空图生成最小生成树

**测试用例：**对空图生成最小生成树

**预期结果：**程序抛出异常，程序正常运行不崩溃。

**实验结果：**

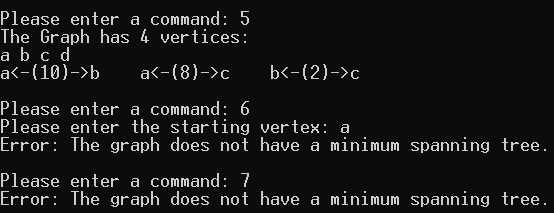


### 4.3.6 对非连通图生成最小生成树

**测试用例：**对非连通图生成最小生成树

**预期结果：**程序抛出异常，程序正常运行不崩溃。

**实验结果：**



### 4.3.7 操作码错误

**测试用例：**输入操作码错误

**预期结果：**程序认为输入退出指令，自动退出。

**实验结果：**

