数据结构课程设计项目说明文档

——关键活动

作 者 姓 名： 马威

学 号： 2151294

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

****

目 录

[1 项目分析 1](#_Toc495668153)

[1.1 项目背景 1](#_Toc495668154)

[1.2 项目要求 1](#_Toc495668154)

[1.3 项目需求分析 2](#_Toc495668155)

[2 项目设计 4](#_Toc495668156)

[2.1 数据结构设计 4](#_Toc495668157)

[2.2 类设计 4](#_Toc495668158)

2.2.1 边结点类（Edge） 4

2.2.2 顶点类（Vertex） 5

2.2.3 图的邻接表类（GraphList） 5

2.3 算法设计 8

2.3.1 算法思路 8

2.3.2 性能评估 9

2.3.3 求解关键活动算法 10

·流程图表示 10

·代码实现 10

2.3.4 拓扑排序算法 13

·流程图表示 13

·代码实现 13

2.3.5 项目主体部分 15

·流程图表示 15

·代码实现 15

[3 项目测试 1](#_Toc495668161)8

[3.1 简单测试情况 1](#_Toc495668162)8

[3.2 一般情况测试，单个起点和单个终点 1](#_Toc495668166)8

[3.3 一般情况测试，多个起点和单个终点 1](#_Toc495668170)9

[3.4 一般情况测试，单个起点和多个终点 1](#_Toc495668174)9

[3.5 一般情况测试，多个起点和多个终点 2](#_Toc495668178)0

[3.6 不可行的方案测试 2](#_Toc495668182)0

[3.7 图不连通的情况 21](#_Toc495668182)

3.8 图为平凡图的情况 21

3.9 最小边数的情况 21

3.10 顶点数或边数输入错误的情况 21

3.11 边的信息输入有误的情况 22

**1.项目分析**

1.1 项目背景

本实验项目是要求在任务调度问题中，如果还给出了完成每个子任务需要的时间，则可以算出完成整个工程项目需要的最短时间。在这些子任务中，有些任务即使推迟几天完成，也不会影响全局的工期；但是有些任务必须准时完成，否则整个项目的工期就要因此而延误，这些任务叫做“关键活动”。

请编写程序判定一个给定的工程项目的任务调度是否可行；如果该调度方案可行，则计算完成整个项目需要的最短时间，并且输出所有的关键活动。

1.2 项目要求

输入说明：输入第1行给出两个正整数N（N<=100）和M，其中N是任务交接点（即衔接两个项目依赖的两个子任务的结点，例如：若任务2要在任务1完成后才开始，则两个任务之间必有一个交接点）的数量，交接点按1～N编号，M是字任务的数量，依次编号为1～M。随后M行，每行给出3个正整数，分别是该任务开始和完成设计的交接点编号以及完成该任务所需要的时间，整数间用空格分隔。

输出说明：如果任务调度不可行，则输出0；否则第一行输出完成整个项目所需要的时间，第2行开始输出所有关键活动，每个关键活动占一行，按照格式“v->W”输出，其中V和W为该任务开始和完成涉及的交接点编号。关键活动输出的顺序规则是：任务开始的交接点编号小者优先，起点编号相同时，与输入时任务的顺序相反。如下面测试用例2中，任务<5，7>先于任务<5，8>输入，而作为关键活动输出时则次序相反。

测试用例：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 输入 | 输出 | 说明 |
| 1 | 7 8  1 2 4  1 3 3  2 4 5  3 4 3  4 5 2  4 6 6  5 7 5  6 7 2 | 17  1->2  2->4  4->6  6->7 | 简单测试情况 |
| 2 | 9 11  1 2 6  1 3 4  1 4 5  2 5 1  3 5 1  4 6 2  5 7 9  5 8 7  6 8 4  7 9 2  8 9 4 | 18  1->2  2->5  5->8  5->7  7->9  8->9 | 一般情况测试，单个起点和单个终点 |
| 3 | 4 5  1 2 4  2 3 5  3 4 6  4 2 3  4 1 2 | 0 | 不可行的方案测试 |

1.3 项目需求分析

对于求解关键活动的程序，需考虑以下需求：

**·正确性**

程序应当能够按照要求准确求解出每一个符合条件的关键活动以及关键路径的长度，并能将其没有遗漏地按照格式显示出来

**·健壮性**

程序应当对相关信息，如顶点数，权值等的错误输入进行处理，也要能对特殊的分布情况（如存在有向回路等）进行合适的处理。

**·高效性**

问题的求解有特殊要求，需要用特定的求法解决，这就要求要在尽可能短的时间内处理大量的数据。

**2.项目设计**

2.1 数据结构设计

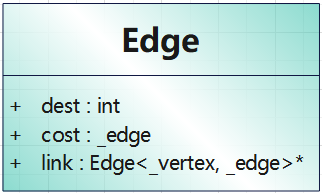
由项目分析可以得出，本项目中有两类对象：交接点与任务，每个任务连接着两个交接点，且自身带有持续时间这一属性，关系较为复杂，需要合适的数据结构对其进行描述。而数学上，图是一种带有若干个点和边的形状，用一条边连接两个点。其通常用来描述多个事物之间的关系，根据需要，还可以对边添加方向、权值等属性。根据分析，项目中任务只能从前置交接点到后续交接点，有特定的方向，也有持续时间的属性，故符合带权有向图的性质。

因此，本项目需实现一个图（Graph）类的实现，要能准确地描述题目中各事物间的关系。同时，由于关键活动的性质，问题也就转变为对有向图求关键活动和关键路径长度的问题。考虑到程序的高效性，图的存储需选择合适的结构，以降低算法的时间复杂度。

2.2 类设计

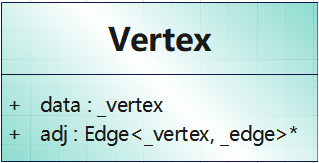
**2.2.1 边结点类（Edge）**

边结点类（Edge）类似一种链表的结点，以其为单位构成的链表构成了某一个顶点的出边（邻接）链表。其存储了某一条边另一顶点的位置和指向下一边结点的指针，其UML图如下：



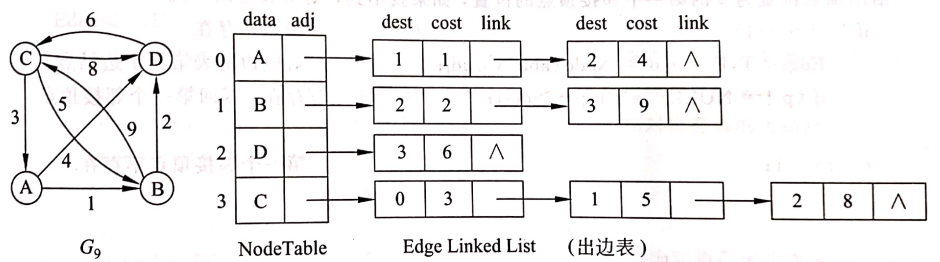
**2.2.2 顶点类（Vertex）**

顶点类（Vertex）类似一种链表的头结点，存储了一个顶点的信息和指向出边（邻接）链表的指针，其UML图如下：



**2.2.3 图的邻接表类（GraphList）**

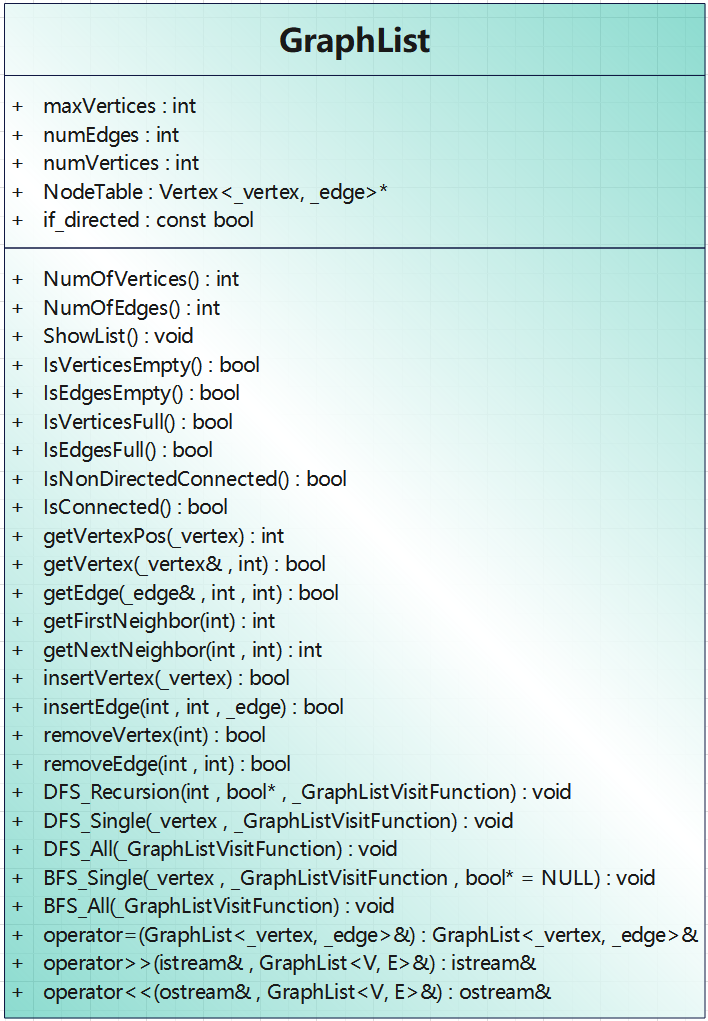
图的邻接表类（GraphList）是通过类似于一个链表的数组进行实现，从而对图进行描述（如下图）。其有一个顶点表NodeList，为一个数组，每个元素（Vertex）存储了一个顶点的信息和指向出边（邻接）链表的指针，类似于链表的头结点。而每一个链表中的其它结点（Edge）都为边结点，存储了边的权值，以及除头结点表示的顶点外这条边另一个顶点在NodeList中的位置。



空间上，这样的存储方式非常节省空间，只有两个节点之间有边相连才进行边结点的存储，在图的边比较稀疏时尤其节省空间开销；时间上，尽管对于插入边等一些基本操作复杂度会比较高，但对于求解最小生成树等问题，用邻接表来表示还是十分理想的。

由于图有有向图和无向图两种，且使用都较为广泛，为了代码的实用性，本项目封装的邻接表类有一个成员if\_directed，指示图是否为有向图，此成员有const限定，在构造函数时需手动指定，此后的插入、删除等操作均根据if\_directed的值进行调整，使同样的代码块兼容有向图和无向图的操作。

其UML图如下：



其中，主要函数如下：

//取堆顶元素，即最小元素

bool GetMin(\_class& x)const;

//返回当前顶点数

int NumOfVertices()const;

//返回当前边数

int NumOfEdges()const;

//展示邻接表

void ShowList()const;

//判断图的顶点数是否为零

bool IsVerticesEmpty()const;

//判断图的边数是否为零

bool IsEdgesEmpty()const;

//判断图的顶点数是否已满

bool IsVerticesFull()const;

//判断图的边数是否已满

bool IsEdgesFull()const;

//判断一个无向图是否连通

bool IsNonDirectedConnected()const;

//判断图是否连通

bool IsConnected()const;

//返回顶点vertex在图中的位置

int getVertexPos(\_vertex vertex)const;

//取顶点pos，pos不合理返回false

bool getVertex(\_vertex& vertex, const int pos)const;

//取边<v1,v2>或(v1,v2)上的权值

bool getEdge(\_edge& edge, const int v1, const int v2)const;

//取顶点v的第一个邻接顶点

int getFirstNeighbor(const int v)const;

//取v邻接顶点w的下一邻接顶点

int getNextNeighbor(const int v, const int w)const;

//插入顶点vertex

bool insertVertex(const \_vertex vertex);

//插入边<v1,v2>，权值为cost

bool insertEdge(const int v1, const int v2, \_edge cost);

//删去顶点v和所有与它相关联的边

bool removeVertex(const int v);

//在图中删去边(v1,v2)或<v1,v2>

    bool removeEdge(const int v1, const int v2);

2.3 算法设计

**2.3.1 算法思路**

求解关键活动可按以下步骤进行：

（1）计算V**i**的最早开始时间，即V**0**到V**i**的最长路径长度。可以使用递推：



即把V**i**所有前驱的最早开始时间分别加上对应边<V**j** ,V**i**>长度，取最大值，即是V**i**的最早开始时间了。（Ve[0]=0，顺序递推）

但由于本项目的图类仅支持求后继操作，故采用如下思路：设head表示前驱，tail表示后继。对head做循环，每取得一个tail，就看此时Ve[head]+dur(<head,tail>)是否大于Ve[tail]，若大于则更新Ve[tail]的值。由于按顺序做循环，tail的每一个前驱都会考虑到，因此Ve[tail]就可以间接求出。

（2）此时递推完成后得到Ve[n-1]是最后一个交接点的最早开始时间，因此得到整个项目的最短时间，将其输出。

（2）计算V**i**的最晚开始时间，即V**n-1**减去V**i**到V**n-1**的最长路径。也可用递推：



即把V**i**所有后继的最晚开始时间分别加上对应边<V**i** ,V**j**>长度，取最小值，即是V**i**的最晚开始时间了。（Vl[n-1]=Ve[n-1]，逆序递推）

（3）计算A**k**的最早开始时间Ae[k]，若A**k**是边<V**i** ,V**j**>，Ae[k]是V**0**到V**i**的最长路径长度，Ae[k]=Ve[i]。

（4）计算A**k**的最晚开始时间Al[k]，若A**k**是边<V**i** ,V**j**>，Al[k]是在满足Vl[j]情况下的最晚开始时间，Al[k]=Vl[j]-dur(<V**i** ,V**j**>)。

（5）检查各个活动，若Ae[k]=Al[k]，则A**k**一刻也不能耽误，为关键活动，输出起始两个交接点。

上述算法能直接进行的基础是每个交接点的序号是拓扑有序的，然而一般情况下，输入的图并不一定是拓扑有序的，需要借助拓扑排序将顶点的下标排成拓扑有序的，同时判断图中是否有有向回路。该算法用了一个栈记录入度为0的顶点，还有一个数组记录顶点的拓扑序列。算法如下：

（1）在输入图的信息时记录下入度表

（2）遍历入度表，入度为0的顶点入栈

（3）弹出一个栈顶元素加入拓扑序列，与其邻接的顶点入度减1，若入度减为0入栈

（4）重复（3）直至栈空且序列中有n个顶点（若序列中不足n个顶点栈就空了，说明图中有有向回路）

若（4）中顺利得到拓扑序列，则按此顺序做递推，求解每个顶点的Ve和Vl。对于输出要求中的关键活动起点小先输出，可以通过计算Ae、Al时按交接点序号顺序做循环来满足；对于输出要求中起点相同的关键活动，先输入的后输出，可以通过建立邻接表时将边结点插在出边表表头来满足。

两种算法的基础是图为连通图，若图不为连通图，则有多个交接点的Ve和Vl均为0，失去了交接点的含义，则项目设计并不合理，故最终仅输出一个0，程序不做任何操作。

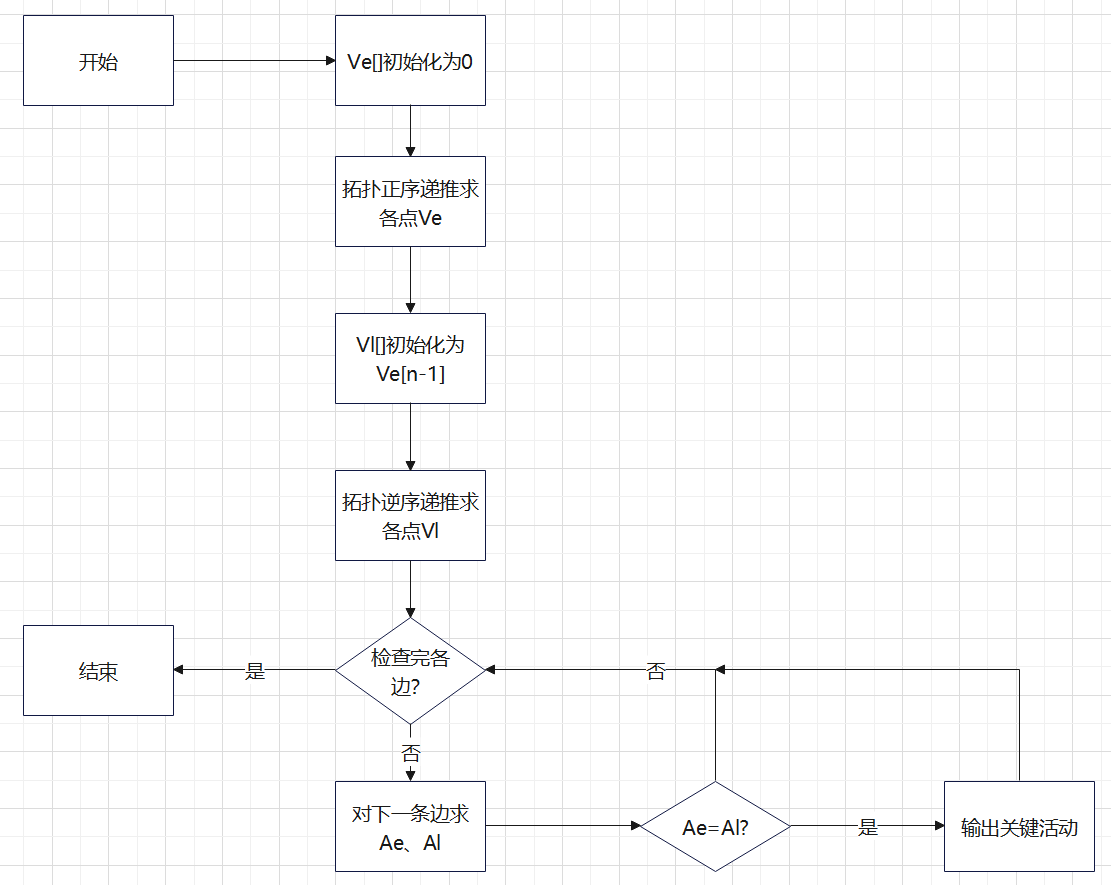
实际上，即使图不连通，就拓扑有序的定义来说，该图还是有可能为拓扑有序的，因此在拓扑排序的过程中不会判断图是否连通，而是在执行拓扑排序之前判断，这仅仅是根据项目背景做出的调整，而非擅自更改拓扑有序判断标准的错误行为。

**2.3.2 性能评估**

计算关键活动中，拓扑正序、逆序求Ve、Vl时所需时间为O(n+e)，求各个活动的Ae、Al时所需时间为O(e)，总时间为O(n+e)。拓扑排序中，n个顶点入栈出栈各一次，需要O(n)，入度减一执行了e次，需要O(e)，总时间为O(n+e)。综上，算法整体时间复杂度为O(n+e)。

**2.3.3 求解关键活动算法**

·流程图表示

****

·代码实现

template<class \_vertex,class \_edge>

void CriticalPath(const GraphList<\_vertex, \_edge>& graph, const int vertexOrder[])

{

    const int ver\_num = graph.NumOfVertices();

    \_edge Ae, Al;

    /\*1、创建Ve、Vl数组\*/

    \_edge\* Ve = new \_edge[ver\_num];

    \_edge\* Vl = new \_edge[ver\_num];

    if (Ve == NULL || Vl == NULL) {

        cerr << "存储分配错误！" << endl;

        exit(1);

    }

    for (int i = 0; i < ver\_num; i++)  /\*Ve都初始化为0\*/

        Ve[i] = 0;

    /\*2、拓扑正序递推求解Ve\*/

    for (int i = 0; i < ver\_num; i++) {

        int tail = graph.getFirstNeighbor(vertexOrder[i]);  /\*按拓扑序列求解\*/

        while (tail != -1) {

            \_edge e;

            graph.getEdge(e, vertexOrder[i], tail);

            if (Ve[vertexOrder[i]] + e > Ve[tail])  /\*Ve[tail] = max{Ve[head] + dur(<head,tail>)}\*/

                Ve[tail] = Ve[vertexOrder[i]] + e;

            tail = graph.getNextNeighbor(vertexOrder[i], tail);

        }

    }

    /\*3、将Vl数组都初始化为Ve[n-1]，原因如下：

        (1)Vl[n-1]=Ve[n-1]

        (2)网络可能有多个终点，它们的Vl都为Ve[n-1]但下标不一定为n-1，这样做将这类特殊情况考虑到了

        (3)后续取最小值时有比较基准了（否则未初始化时，Vl都是一个绝对值很大的负数，比较会出现错误）

    \*/

    for (int i = 0; i < ver\_num; i++)

        Vl[i] = Ve[vertexOrder[ver\_num - 1]];

    cout << Ve[vertexOrder[ver\_num - 1]] << endl;    /\*输出整个项目完成所需时间\*/

    /\*4、拓扑逆序递推求解Vl\*/

    for (int i = ver\_num - 2; i >= 0; i--) {

        int tail = graph.getFirstNeighbor(vertexOrder[i]);  /\*按拓扑序列求解\*/

        while (tail != -1) {

            \_edge e;

            graph.getEdge(e, vertexOrder[i], tail);

            if (Vl[tail] - e < Vl[vertexOrder[i]])  /\*Vl[head] = min{Vl[tail] - dur(<head,tail>)}\*/

                Vl[vertexOrder[i]] = Vl[tail] - e;

            tail = graph.getNextNeighbor(vertexOrder[i], tail);

        }

    }

    /\*4、求解关键活动\*/

    for (int i = 0; i < ver\_num; i++) {  /\*为符合要求输出，直接从序号检查\*/

        int j = graph.getFirstNeighbor(i);

        while (j != -1) {

            Ae = Ve[i];  /\*计算Ae\*/

            \_edge e;

            graph.getEdge(e, i, j);

            Al = Vl[j] - e;  /\*计算Al\*/

            if (Al == Ae) {  /\*若Al=Ae，则为关键活动，输出\*/

                \_vertex v1, v2;

                graph.getVertex(v1, i);

                graph.getVertex(v2, j);

                cout << v1 << "->" << v2 << endl;

            }

            j = graph.getNextNeighbor(i, j);

        }

    }

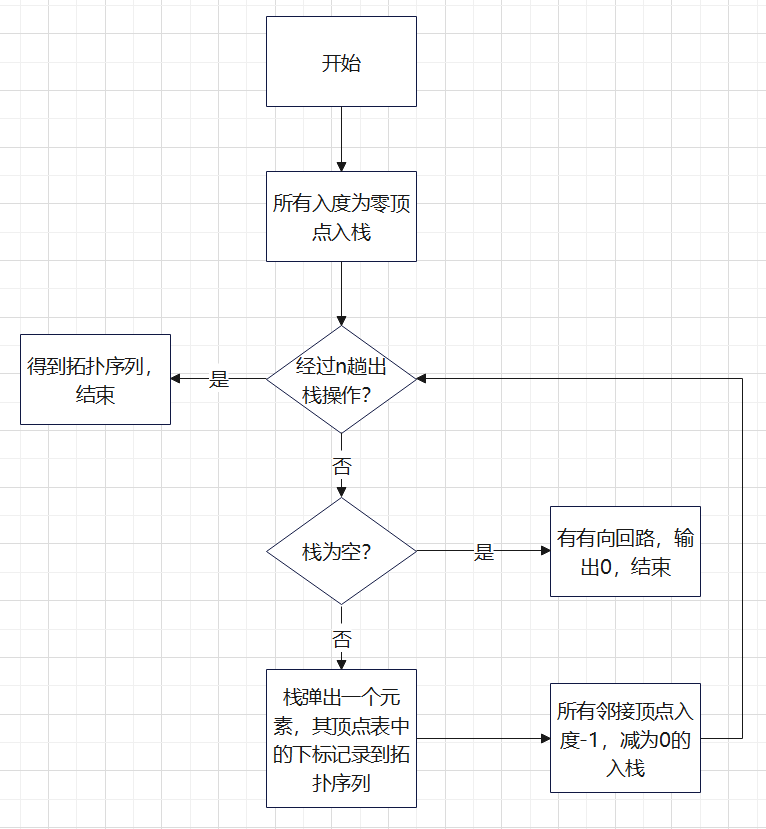
    delete[] Ve;

    delete[] Vl;

}

**2.3.4 拓扑排序算法**

·流程图表示



·代码实现

template<class \_vertex, class \_edge>

bool TopologicalSort(const GraphList<\_vertex, \_edge>& graph, int vertexOrder[], int count[])

{

    const int ver\_num = graph.NumOfVertices();

    int top = -1;

    for (int i = 0; i < ver\_num; i++) {

        if (!count[i]) {  /\*将入度为零的顶点压入栈中\*/

            count[i] = top;

            top = i;

        }

    }

    for (int i = 0; i < ver\_num; i++) {

        if (top == -1) {  /\*若还没经过n趟出栈后栈就为空，则有有向回路，图不为拓扑有序，返回假\*/

            cout << '0' << endl;

            return false;

        }

        else {

            int v = -1;

            v = top;  /\*从栈弹出一个入度为0顶点\*/

            top = count[top];

            vertexOrder[i] = v;     /\*将该顶点插入拓扑序列中\*/

            int w = graph.getFirstNeighbor(v);

            while (w != -1) {

                if (--count[w] == 0) {  /\*让v的所有邻接顶点入度-1，若入度变为0，压入栈中\*/

                    count[w] = top;

                    top = w;

                }

                w = graph.getNextNeighbor(v, w);

            }

        }

    }

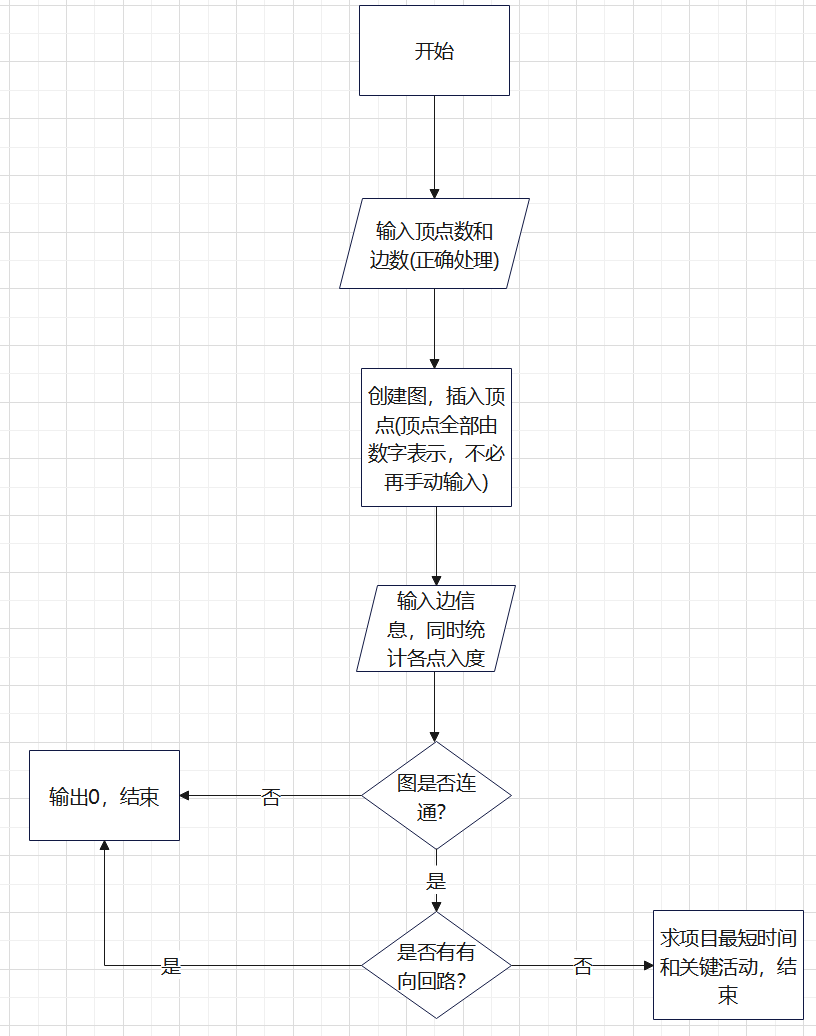
    return true;

}

**2.3.5 项目主体部分**

由于需要记录各顶点的入度，故图的信息输入未使用重载函数，而是直接实现。同时，根据项目背景，在对图进行拓扑排序之前，先判断图是否连通。若不连通则金输出0。

·流程图表示：



·代码实现：

int main()

{

    int ver\_num = 0, edge\_num = 0;

    /\*1、输入顶点数和边数\*/

    while (1) {

        cin >> ver\_num >> edge\_num;

        if (cin.good() && ver\_num > 0 && ver\_num <= 100 && edge\_num > 0 && edge\_num <= ver\_num \* (ver\_num - 1))

            break;

        cin.clear();

        cin.ignore(INT\_MAX, '\n');

    }

    /\*2、构建图，插入顶点\*/

    GraphList<int, int> graph(true, ver\_num);

    for (int i = 0; i < ver\_num; i++)

        graph.insertVertex(i + 1);

    /\*3、构建顶点入度记录数组\*/

    int\* count = new int[ver\_num];

    if (count == NULL) {

        cerr << "存储分配错误！" << endl;

        exit(1);

    }

    for (int i = 0; i < ver\_num; i++)

        count[i] = 0;

    /\*4、输入边信息，同时统计各点入度\*/

    for (int times = 0; times < edge\_num; times++) {

        int i, j, num = 0;

        int cost;

        while (1) {

            cin >> i >> j >> cost;

            //适应题目输入要求，已对范围进行调整

            if (cin.good() && i > 0 && i <= ver\_num && j > 0 && j <= ver\_num && cost >= 0 && cost <= INT\_MAX) {

                graph.insertEdge(i - 1, j - 1, cost);

                num++;

                count[j - 1]++;

                break;

            }

            cin.clear();

            cin.ignore(INT\_MAX, '\n');

            cout << "第" << num + 1 << "条边信息有误，请重新输入！" << endl;

        }

    }

    /\*5、为拓扑序列开辟空间\*/

    int\* vertexOrder = new int[ver\_num];

    if (vertexOrder == NULL) {

        cerr << "存储分配错误！" << endl;

        exit(1);

    }

    /\*6、若图不连通，则方案不可行，输出0\*/

    if (!graph.IsConnected())

        cout << '0' << endl;

    /\*7、进行拓扑排序，若拓扑有序，则计算关键路径\*/

    else if (TopologicalSort(graph, vertexOrder, count))

        CriticalPath(graph, vertexOrder);

    delete[] count;

    delete[] vertexOrder;

    return 0;

}

**3.项目测试**

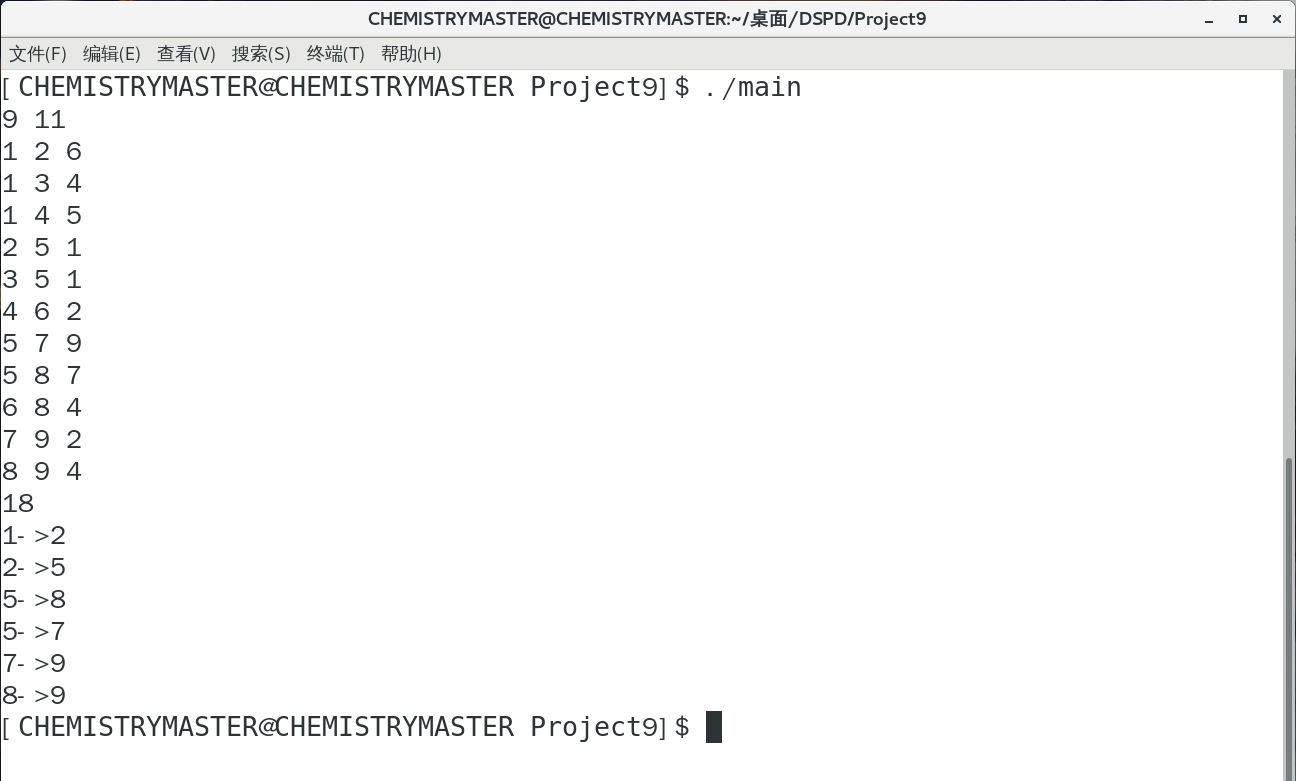
3.1 简单测试情况

测试结果：



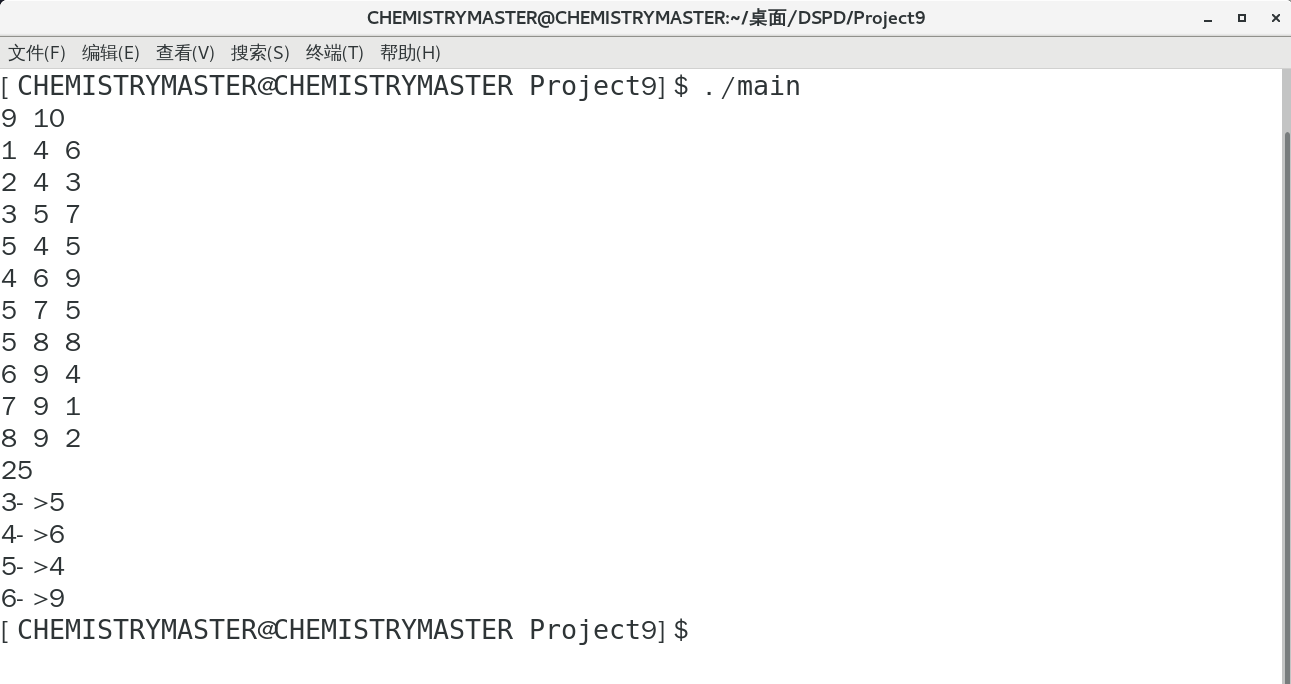
3.2 一般情况测试，单个起点和单个终点

测试结果：



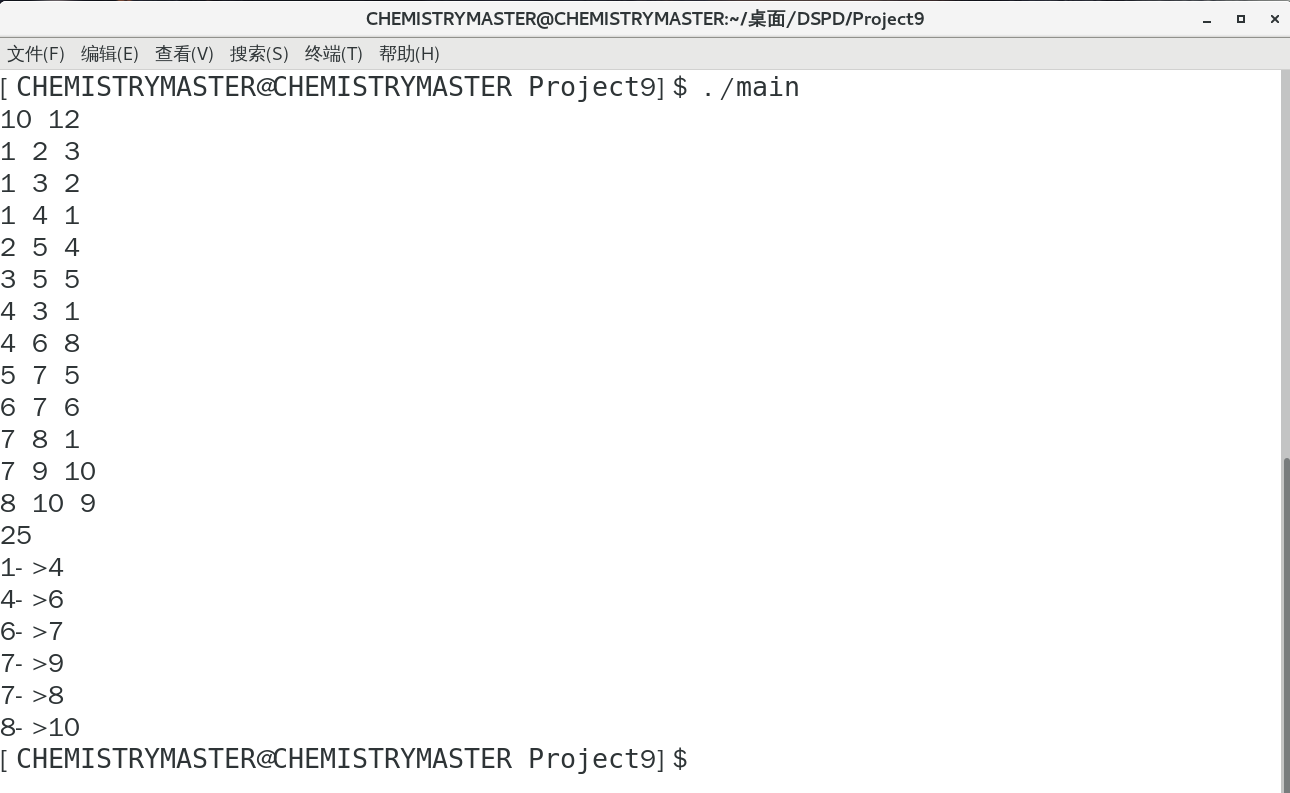
3.3 一般情况测试，多个起点和单个终点

测试结果：



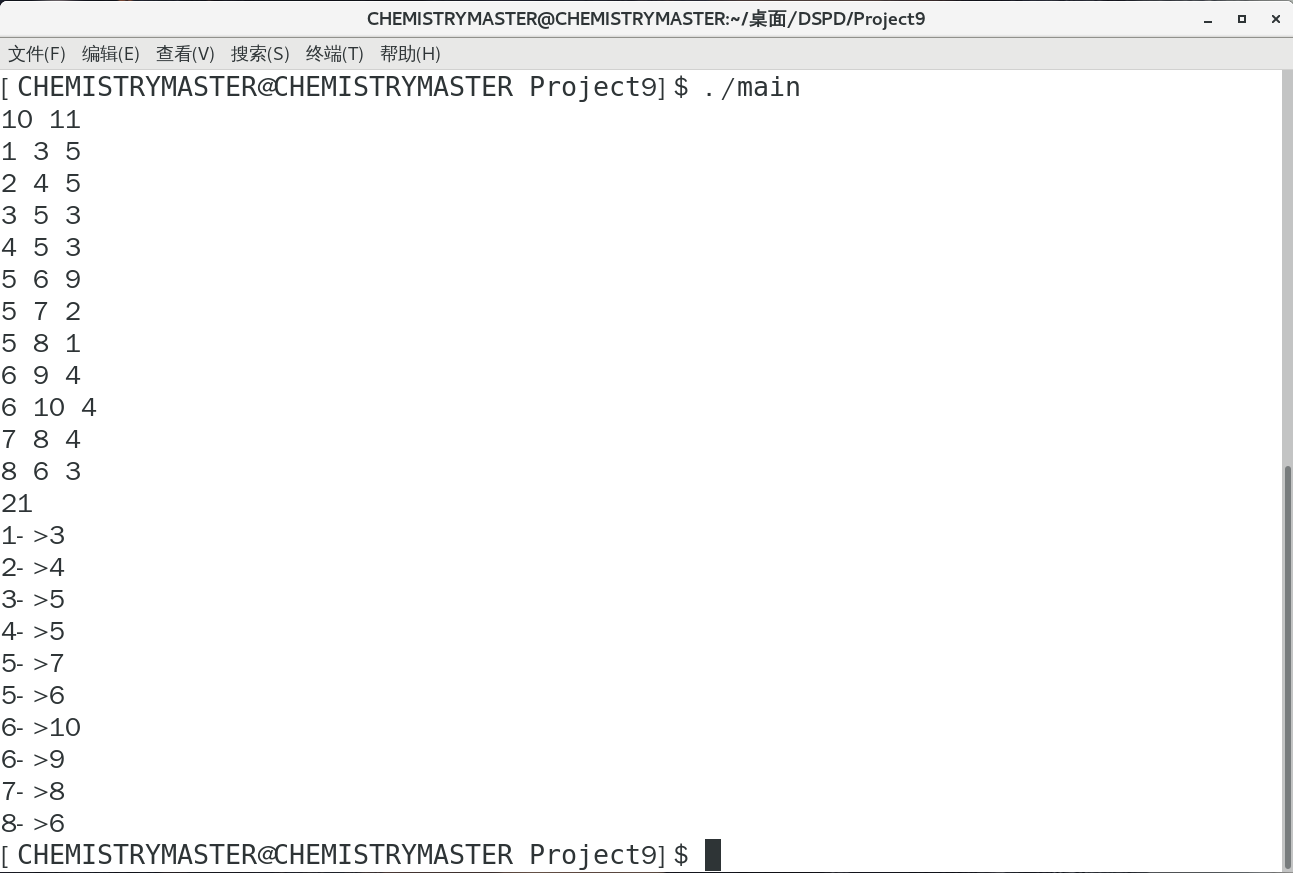
3.4 一般情况测试，单个起点和多个终点

测试结果：



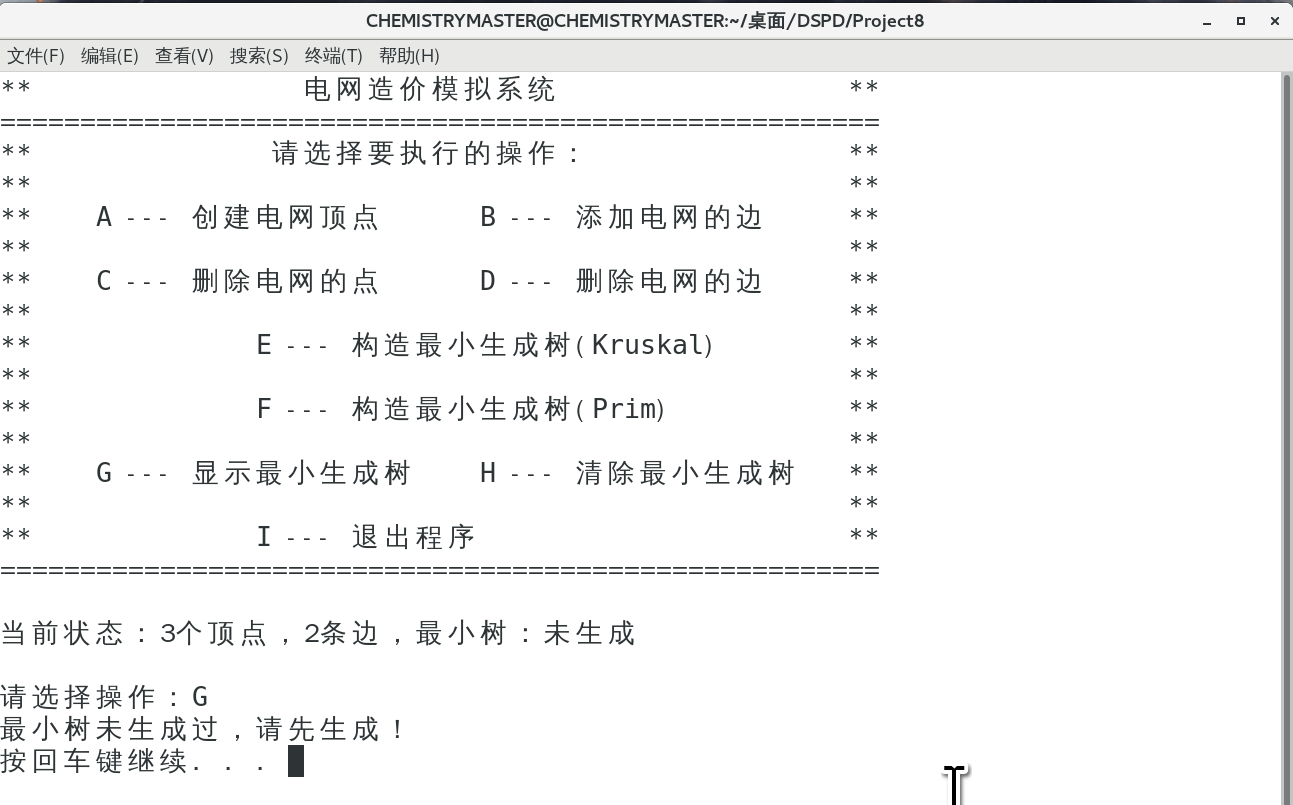
3.5 一般情况测试，多个起点和多个终点

测试结果：



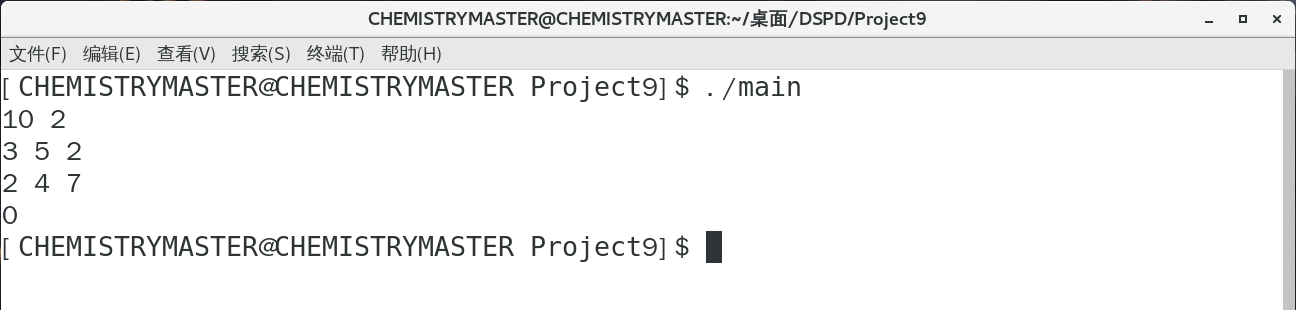
3.6 不可行的方案测试

测试结果：



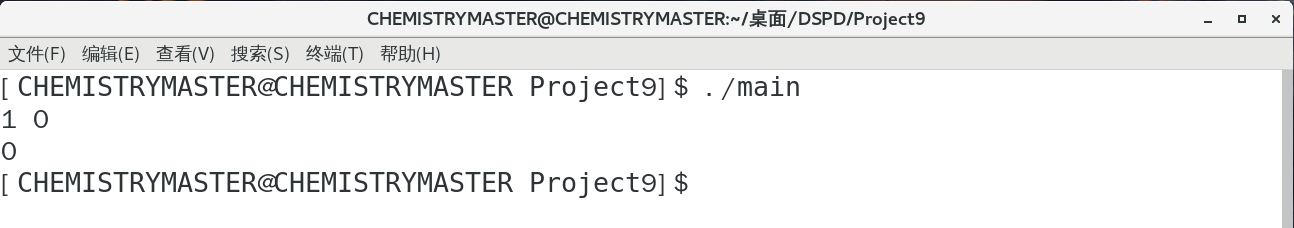
3.7 图不连通的情况

测试结果：



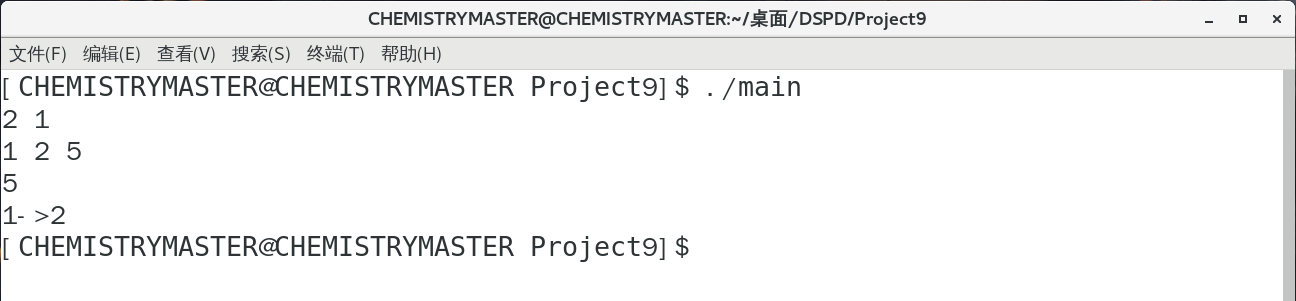
3.8 图为平凡图的情况

测试结果：



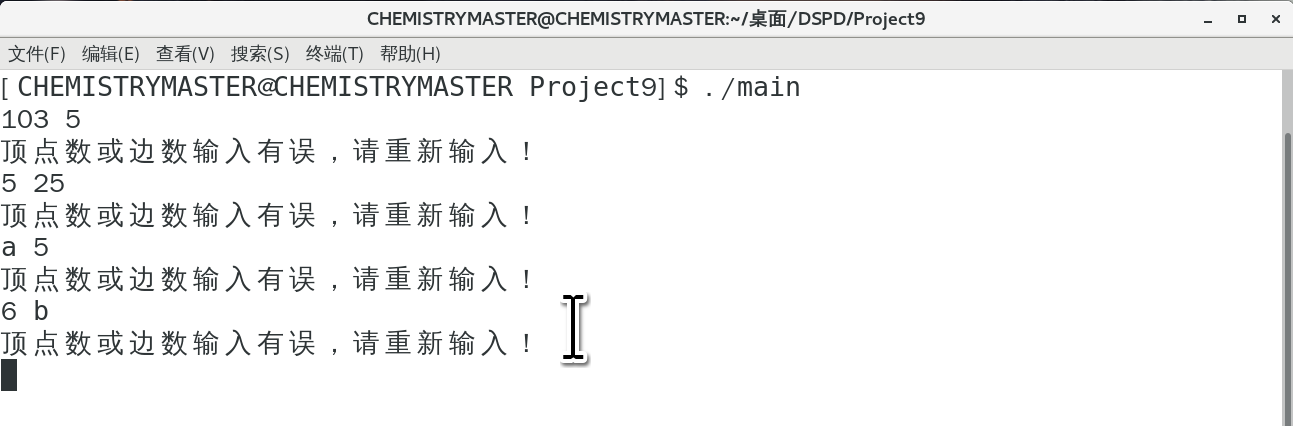
3.9 最小边数的情况

测试结果：



3.10 顶点数或边数输入错误的情况

测试结果：



3.11 边的信息输入有误的情况

测试结果：

