项目说明文档

数据结构课程设计

——关键活动

作 者 姓 名： 张喆

学 号： 1754060

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 1](#_Toc533930801)

[1.1 背景分析 1](#_Toc533930802)

[1.2 功能分析 1](#_Toc533930803)

[2 设计 2](#_Toc533930804)

[2.1 算法设计 2](#_Toc533930805)

[2.2 类结构设计 2](#_Toc533930806)

[2.3 成员与操作设计 3](#_Toc533930807)

[2.4 系统设计 4](#_Toc533930808)

[3 实现 5](#_Toc533930809)

[3.1 拓扑排序功能的实现 5](#_Toc533930810)

[3.1.1 拓扑排序功能流程图 5](#_Toc533930811)

[3.1.2 拓扑排序功能核心代码 6](#_Toc533930812)

[3.1.3 拓扑排序功能截屏示例 8](#_Toc533930813)

[3.2 寻找关键路径功能的实现 9](#_Toc533930814)

[3.2.1 寻找关键路径功能流程图 9](#_Toc533930815)

[3.2.2 寻找关键路径功能核心代码 10](#_Toc533930816)

[3.2.3 寻找关键路径功能截屏示例 11](#_Toc533930817)

[3.3 输出关键路径功能的实现 12](#_Toc533930818)

[3.3.1 输出关键路径功能核心代码 12](#_Toc533930819)

[3.3.2 输出关键路径功能截图示例 12](#_Toc533930820)

[3.4 取网络中信息功能的实现 13](#_Toc533930821)

[3.4.1 取第一个邻居功能核心代码 13](#_Toc533930822)

[3.4.2 取下一个邻居功能核心代码 13](#_Toc533930823)

[3.4.3 取边的长度功能核心代码 13](#_Toc533930824)

[3.5 读入信息功能的实现 14](#_Toc533930825)

[3.5.1 读入顶点功能核心代码 14](#_Toc533930826)

[3.5.2 读入边功能核心代码 14](#_Toc533930827)

[3.5.3 判断读入的边是否合法功能核心代码 14](#_Toc533930828)

[3.5.4 总体读入功能核心代码 14](#_Toc533930829)

[3.6 总体系统的实现 16](#_Toc533930830)

[3.6.1 总体系统核心代码 16](#_Toc533930831)

[3.6.2 总体系统截屏示例 17](#_Toc533930832)

[4 测试 18](#_Toc533930833)

[4.1 功能测试 18](#_Toc533930834)

[4.1.1 读入顶点和边功能测试 18](#_Toc533930835)

[4.1.2 取首邻居和下一个邻居功能测试 19](#_Toc533930836)

[4.1.3 拓扑排序功能测试 19](#_Toc533930837)

[4.1.4 寻找关键路径功能测试 20](#_Toc533930838)

[4.1.5 输出关键活动功能测试 21](#_Toc533930839)

[4.2 边界测试 22](#_Toc533930840)

[4.2.1 调度任务不可行 22](#_Toc533930841)

[4.2.2 多个起点 22](#_Toc533930842)

[4.2.3 多个终点 23](#_Toc533930843)

[4.2.4 起点到终点有多条最长路径 23](#_Toc533930844)

[4.3 出错测试 25](#_Toc533930845)

[4.3.1 任务交接点数或子任务数有误 25](#_Toc533930846)

[4.3.2 读入的子任务时开始和完成的交接点编号不在任务交接点标号中 25](#_Toc533930847)

# 1 分析

## 1.1 背景分析

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”。

在AOE网络中，有些活动可以并行地进行.从源点到各个顶点，以至从源点到汇点的有向路径可能不止一条。这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同，但只有各条路径上所有活动都完成了，整个工程才算完成。因此，完成整个工程所需的时间取决于从源点到汇点的最长路径长度，即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做“关键路径”。

关键路径上的所有活动都是关键活动。

## 1.2 功能分析

作为AOE网络，首先应该能读入用户指定的活动，即应能正确读入并存储活动网络的顶点（如题干描述：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）;之后应能正确读入并存储活动网络的边（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）。

其次，应能使用正确的方法判断该任务调度是否可行，如果不可行则输出0并终止程序；否则应能正确输出题目指定的信息（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。）

# 2 设计

## 2.1 算法设计

定义一些与计算关键活动有关的量：

（1）事件Vi的最早可能开始时间Ve[i]是从源点V0到顶点Vi的最长路径长度。只有当前面的活动都完成时，事件Vi才可能开始。

（2）事件VI的最迟允许开始时间Vl[i]是在保证汇点Vn-1在Ve[n-1]时刻完成的前提下，事件Vi的允许的最迟开始时间。它等于Ve[n-1]减去从Vi到Vn-1的最长路径长度。

（3）活动ak的最早可能开始时间Ae[k]：设活动ak在有向边<Vi,Vj>上，则Ae[k]是从源点V0到顶点Vi的最长路径长度。因此，Ae[k]=Ve[i]

（4）活动ak的最迟允许开始时间Al[k]：社活动ak在有向边<Vi,Vj>上，则Ae[k]是在不会引起时间延误的前提下，该活动允许的最迟开始时间。Al[k]=Vl[j] – dur(<i,j>)。其中，dur(<i,j>)是完成ak所需的时间。

（5）用Al[k]-Ae[k]表示活动ak的最早可能开始时间和最迟允许开始时间的时间余量，也叫做松弛时间。Al[k]==Ae[k]表示活动ak是没有时间余量的关键活动。

Ve[i]=max{Ve[j]+dur(<Vj, Vi>)}

Vl[i]=min{Vl[j]-dur(<Vi,Vj>)}

这两个递推公式的计算必须分别在拓扑有序及逆拓扑有序的前提下进行。计算Ve[i]时，Vi的所有前驱顶点Vj的Ve[j]都已经求出。反之，在计算Vl[i]时，也必须在Vi的所有后继顶点Vj的Vl[j]都已求出的条件下才能进行计算。

该算法要以拓扑排序为基础，在吧各个顶点排除拓扑有序的同时，计算Ve[i]；再以逆拓扑有序的顺序计算Vl[i]。

## 2.2 类结构设计

该系统采用两种方式完成关键活动的求解。

第一种使用变形的邻接表存储，定义边类Edge和活动类Activity，其中活动类中分别储存顶点集合和边集合。并将入度出度及活动开始的最早时间和最迟时间都封装在活动类中，并设置方法可以判断该调度任务是否可行，如果可行的话可以正确找出关键活动并输出。

第二种使用邻接矩阵储存网络中的顶点及边，同样将入度出度以及活动开始的最早时间和最迟时间封装在活动类中，其他内容同第一种类的定义一致。

## 2.3 成员与操作设计

**边类（Edge）**

**公有成员：**

Vertex v1, v2;

int cost = 0;

**边类的友元：**

friend istream& operator>>(istream &is, const Edge &buf);

istream& operator>>(istream &is, Edge &buf)

{

is >> buf.v1 >> buf.v2 >> buf.cost;

return is;

}

**邻接表表示的活动类（Activity）**

**私有成员：**

vector<Vertex> vertex; //顶点集合

vector<Edge> edge; //网络中的边集合

vector<int> Ve, Vl; //事件最早和最迟开始时间

int Ae, Al; //活动最早和最迟开始时间

vector<int> indegree, outdegree; //结点的入度和出度

vector<vector<int > > result; //储存结果集合

**公有操作：**

Activity() = default;

Activity(int VertexNum, int EdgeNum);

bool judgeFeasible(); //判断任务调度是否可行

void findCriticialPath(); //求关键路径的算法

void showCriticialPath(); //输出关键路径

**私有操作：**

bool islegal(const Edge &buf); //判断读入的边是否合法

void InitVertexs(); //读入网络中的顶点

void InitEdges(); //读入网络中的边

Vertex getFirstNeighbour(const Vertex &source); //取source的第一个邻居

Vertex getNextNeighbour(const Vertex &source, const Vertex &dest); //取source的下一个邻居(不为dest)

Vertex getLastNeighbour(const Vertex &source); //取source的最后一个邻居

Vertex getBackNeighbour(const Vertex &source, const Vertex &dest); //取source的前一个邻居(不为dest)

int getWeight(const Vertex &v1, const Vertex &v2); //获取某条边的长度

**邻接矩阵表示的活动类（Activity\_m）**

**私有成员：**

vector<vector<Vertex> > NetWork; //用邻接矩阵存储网络的顶点和边

vector<vector<int> > result;

vector<int> Indegree, Outdegree;

vector<int> EarliestTime, LatestTime; //最早开始时间和最迟开始时间

int ECT; //最早完成时间

int idx;

**公有操作：**

Activity\_m() = default;

Activity\_m(int VertexNum, int EdgeNum);

void Init(int EdgeNum);

bool TopoSort(); //正向拓扑排序用于检测调度是否可行 & 计算最早开始时间

void findCriticialPath(); //逆向拓扑排序计算最迟开始时间 & 时间余量

void showCriticialPath();

## 2.4 系统设计

系统首先读入用户指定的任务交接点数和子任务数量，之后创建以此为基础的活动类。

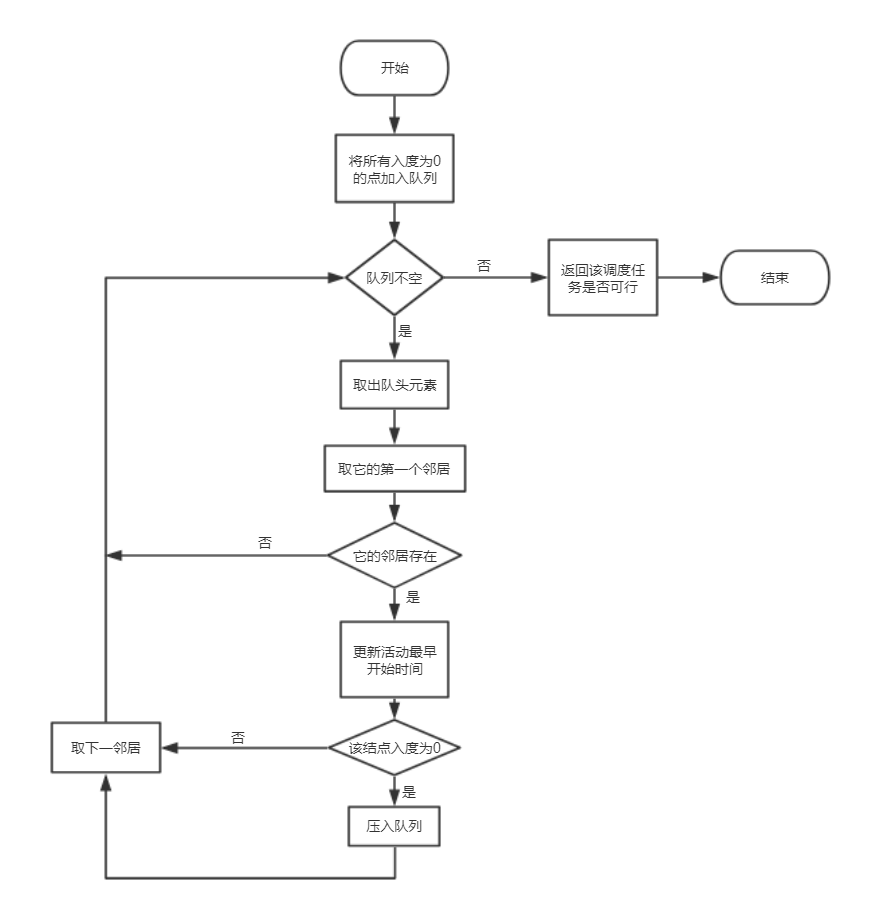
调用相应的方法判断该调度任务是否可行。如果不可行则输出0并终止程序；如果该调度可行调用寻找关键路径的方法计算关键活动和关键路径。

最后按照题干的要求输出关键路径。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。

# 3 实现

## 3.1 拓扑排序功能的实现

### 3.1.1 拓扑排序功能流程图



### 3.1.2 拓扑排序功能核心代码

bool Activity::judgeFeasible()

{

int i, j, w;

/\*计算入度和出度\*/

for (i = 0; i < this->edge.size(); ++i)

{

j = getFirstNeighbour(i);

while(j != -1)

{

indegree[j]++;

outdegree[i]++;

j = getNextNeighbour(i, j);

}

}

/\*拓扑排序\*/

int cnt = 0;

queue<int> Q;

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

//先将所有入度为0的顶点压入队列

{

if (indegree[i] == 0)

{

Q.push(i);

}

}

while (!Q.empty())

{

i = Q.front(); Q.pop();

++cnt;

j = getFirstNeighbour(i);

while (j != -1)

//如果该顶点往后还有边的话

{

w = getWeight(i, j);

if (Ve[i] + w > Ve[j]) { Ve[j] = Ve[i] + w; } //正拓扑序列计算最早开始时间

if (--indegree[j] == 0) //将度数为0的顶点压入队列

{

Q.push(j);

}

j = getNextNeighbour(i, j);

}

}

return (cnt == this->vertex.size());

//如果所有顶点都被拓扑排序遍历过则该调度可行

}

bool Activity\_m::TopoSort()

{

int i, j, V, cnt = 0;

queue<int> Q;

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

//将所有入度为0的点压入队列

{

if (Indegree[i] == 0)

{

Q.push(i);

}

}

while (!Q.empty())

{

V = Q.front(); Q.pop();

++cnt;

for (int j = 0; j < NetWork.size(); ++j)

{

if (NetWork[V][j] != INFINITE)

{

if (EarliestTime[V] + NetWork[V][j] > EarliestTime[j])

//选取最长路径

{

EarliestTime[j] = EarliestTime[V] + NetWork[V][j];

}

if (--Indegree[j] == 0)

//将入度为0的点压入队列

{

Q.push(j);

}

}

}

}

this->ECT = \*max\_element(EarliestTime.begin(), EarliestTime.end());

//最早完成时间应是所有元素中最大的

idx = max\_element(EarliestTime.begin(), EarliestTime.end()) - EarliestTime.begin();

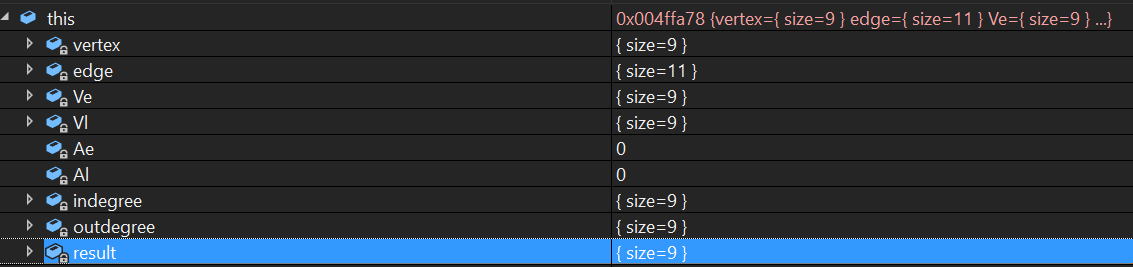
//最早完成时间的位置

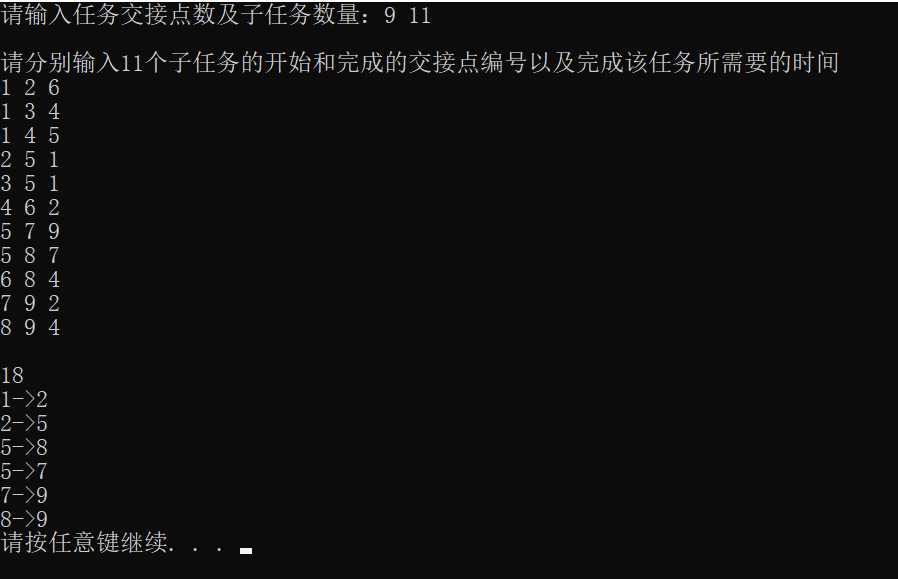
return (cnt == NetWork.size());

//如果所有顶点都被拓扑排序遍历过则该调度可行

}

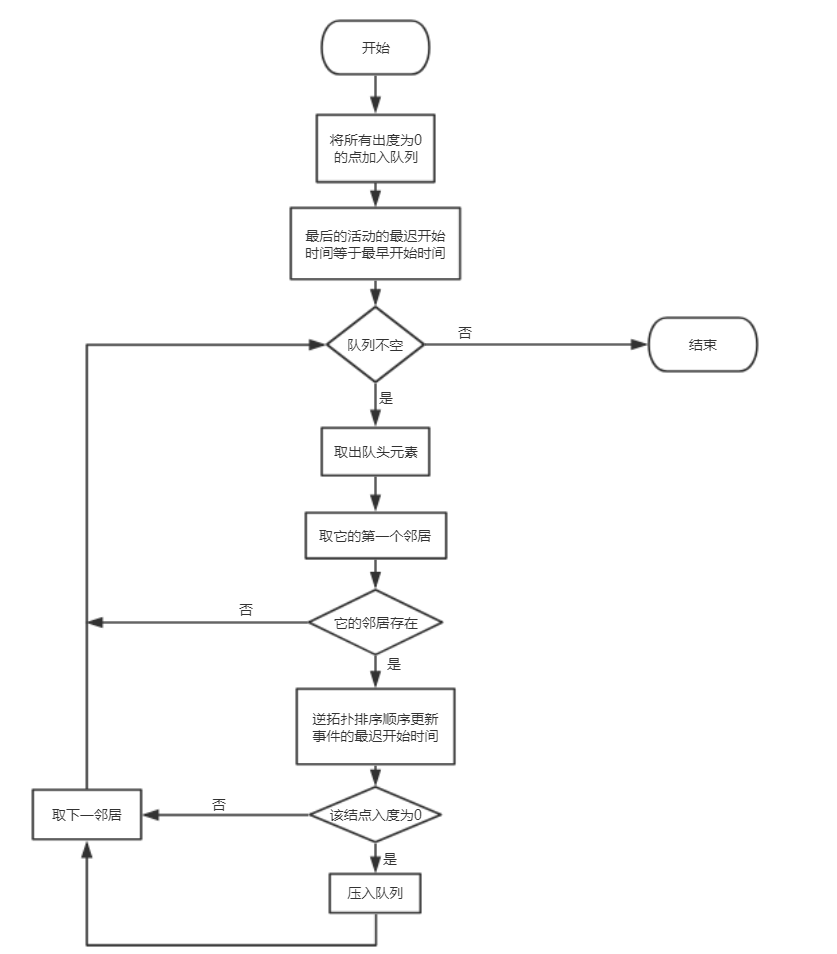
### 3.1.3 拓扑排序功能截屏示例





## 3.2 寻找关键路径功能的实现

### 3.2.1 寻找关键路径功能流程图



### 3.2.2 寻找关键路径功能核心代码

void Activity::findCriticialPath()

{

int i, j, w;

queue<int> Q;

for (i = 0; i < this->vertex.size(); ++i)

//将所有出度为0的顶点压入队列

{

if (outdegree[i] == 0)

{

Q.push(i);

}

}

Vl[max\_element(Ve.begin(), Ve.end()) - Ve.begin()] = \*max\_element(Ve.begin(), Ve.end());

//最后的活动的最迟开始时间等于最早开始时间

while (!Q.empty())

{

j = Q.front(); Q.pop();

i = getLastNeighbour(j);

while (i != -1)

{

w = getWeight(i, j);

if (Vl[j] - w < Vl[i]) { Vl[i] = Vl[j] - w; } //逆拓扑排序顺序计算事件的最迟开始时间

result[i][j] = Vl[j] - Ve[i] - getWeight(i, j); //时间余量等于三者之差

if (--outdegree[i] == 0)

//将出度为0的顶点压入队列中

{

Q.push(i);

}

i = getBackNeighbour(j, i);

}

}

}

void Activity\_m::findCriticialPath()

{

int i, j, V;

LatestTime[idx] = ECT; //假设最后活动的最迟开始时间等于最早开始时间

queue<int> Q;

for (i = 0; i < NetWork.size(); ++i)

//将所有出度为0的顶点压入队列

{

if (Outdegree[i] == 0)

{

Q.push(i);

}

}

while (!Q.empty())

{

V = Q.front(); Q.pop();

for (j = 0; j < NetWork.size(); ++j)

{

if (NetWork[j][V] != INFINITE)

{

if (LatestTime[V] - NetWork[j][V] <= LatestTime[j])

//逆推时选取最小的情况

{

LatestTime[j] = LatestTime[V] - NetWork[j][V];

result[j][V] = LatestTime[V] - EarliestTime[j] - NetWork[j][V];

//计算时间余量

}

if (--Outdegree[j] == 0)

//将出度为0的顶点压入队列

{

Q.push(j);

}

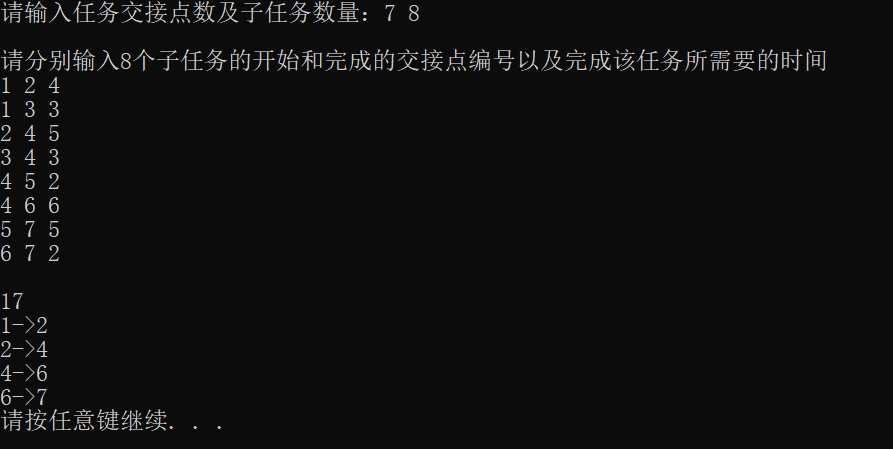
}

}

}

}

### 3.2.3 寻找关键路径功能截屏示例



## 3.3 输出关键路径功能的实现

### 3.3.1 输出关键路径功能核心代码

void Activity::showCriticialPath()

{

/\*输出结果\*/

cout << endl << \*max\_element(Ve.begin(), Ve.end()) << endl; //输出整个项目所需要的时间

for (int i = 0; i < this->vertex.size(); ++i)

{

for (int j = this->vertex.size() - 1; j >= 0; --j)

//按要求第一个顶点相同时，逆读入顺序输出

{

if (result[i][j] == 0)

{

cout << i + 1 << "->" << j + 1 << endl;

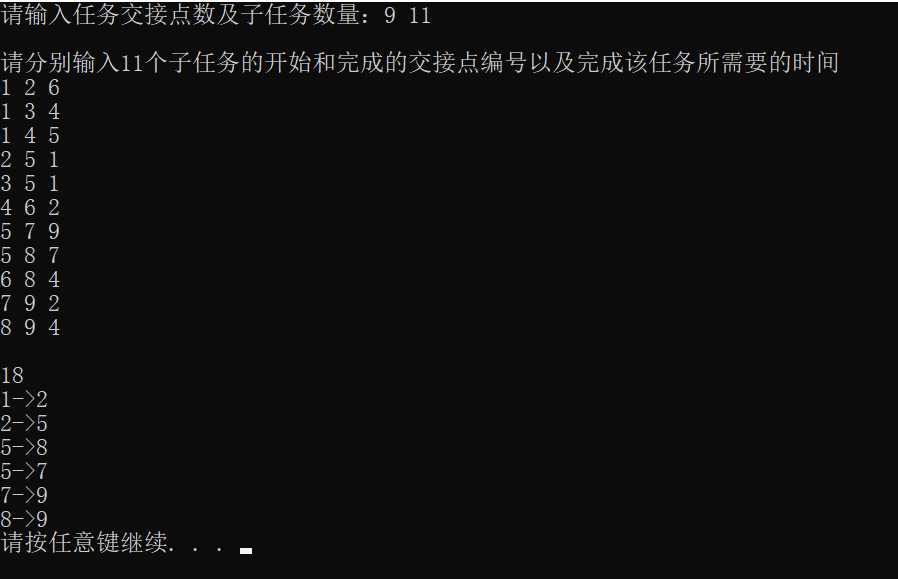
}

}

}

}

### 3.3.2 输出关键路径功能截图示例



## 3.4 取网络中信息功能的实现

### 3.4.1 取第一个邻居功能核心代码

Vertex Activity::getFirstNeighbour(const Vertex & source)

{

for (Edge \*p = &this->edge[0]; p < &this->edge[0] + this->edge.size(); ++p)

{

if (p->v1 == source)

{

return p->v2;

}

}

return -1;

}

### 3.4.2 取下一个邻居功能核心代码

Vertex Activity::getNextNeighbour(const Vertex & source, const Vertex & dest)

{

Edge \*p = NULL;

for (p = &this->edge[0]; p < &this->edge[0] + this->edge.size(); ++p)

//越过当前已经找过的邻居

{

if (p->v1 == source && p->v2 == dest)

{

break;

}

}

for (++p; p < &this->edge[0] + this->edge.size(); ++p)

{

if (p->v1 == source)

{

return p->v2;

}

}

return -1;

}

### 3.4.3 取边的长度功能核心代码

int Activity::getWeight(const Vertex & v1, const Vertex & v2)

{

for (Edge \*p = &this->edge[0]; p < &this->edge[0] + this->edge.size(); ++p)

{

if (p->v1 == v1 && p->v2 == v2)

{

return p->cost;

}

}

}

## 3.5 读入信息功能的实现

### 3.5.1 读入顶点功能核心代码

void Activity::InitVertexs()

{

for (int i = 0; i < this->vertex.size(); ++i)

{

this->vertex[i] = i; //读入的时候默认是0~n-1

}

}

### 3.5.2 读入边功能核心代码

void Activity::InitEdges()

{

cout << endl << "请分别输入" << this->edge.size() << "个子任务的开始和完成的交接点编号以及完成该任务所需要的时间" << endl;

Edge buf;

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

{

cin >> buf;

if (!islegal(buf))

//读入边时的检测

{

cerr << "子任务的开始和完成的交接点编号必须是任务交接点标号，请重新输入" << endl;

--i;

continue;

}

this->edge[i].v1 = buf.v1 - 1;

this->edge[i].v2 = buf.v2 - 1;

this->edge[i].cost = buf.cost;

}

}

### 3.5.3 判断读入的边是否合法功能核心代码

bool Activity::islegal(const Edge & buf)

{

return (find(this->vertex.begin(), vertex.end(), buf.v1 - 1) != this->vertex.end()

&& find(this->vertex.begin(), this->vertex.end(), buf.v2 - 1) != this->vertex.end());

}

### 3.5.4 总体读入功能核心代码

Activity::Activity(int VertexNum, int EdgeNum)

{

/\*读入顶点集合\*/

vertex.resize(VertexNum);

InitVertexs();

/\*读入边集合\*/

edge.resize(EdgeNum);

InitEdges();

/\*初始化入度和出度\*/

indegree.resize(VertexNum,0);

outdegree.resize(VertexNum, 0);

/\*初始化最早和最迟开始时间\*/

Ve.resize(VertexNum, 0);

Vl.resize(VertexNum, INFINITE); //因为最迟时间是取最小的, 所以要初始化为无穷大

/\*初始化结果集合\*/

result.resize(vertex.size(), vector<int>(vertex.size(), INFINITE));

}

Activity\_m::Activity\_m(int VertexNum, int EdgeNum)

{

/\*初始化邻接矩阵\*/

NetWork.resize(VertexNum, vector<Vertex>(VertexNum, INFINITE));

/\*初始化结果集合\*/

result.resize(NetWork.size(), vector<int>(NetWork.size(), INFINITE));

/\*初始化入度和出度\*/

Indegree.resize(NetWork.size(), 0);

Outdegree.resize(NetWork.size(), 0);

/\*初始化最早开始时间和最迟开始时间\*/

EarliestTime.resize(NetWork.size(), 0);

LatestTime.resize(NetWork.size(), INFINITE);

/\*读入顶点和边 并 计算入度和出度\*/

Init(EdgeNum);

}

void Activity\_m::Init(int EdgeNum)

{

Edge buf;

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

{

cin >> buf;

NetWork[buf.v1-1][buf.v2-1] = buf.cost; //题目从1开始输入的

}

/\*计算各顶点的入度和出度\*/

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

{

for (int j = 0; j < NetWork.size(); ++j)

{

if (NetWork[i][j] != INFINITE)

{

++Indegree[j];

++Outdegree[i];

}

}

}

}

## 3.6 总体系统的实现

### 3.6.1 总体系统核心代码

int VertexNum, EdgeNum;

while (true)

{

cout << "请输入任务交接点数及子任务数量：";

cin >> VertexNum >> EdgeNum;

if (VertexNum > 0 && EdgeNum > 0)

{

break;

}

else

{

cerr << "任务交接点数和子任务数量必须非负，请重新输入" << endl;

}

}

Activity act(VertexNum, EdgeNum);

if (!act.judgeFeasible())

//判断调度任务是否可行

{

cout << endl << 0 << endl;

}

else

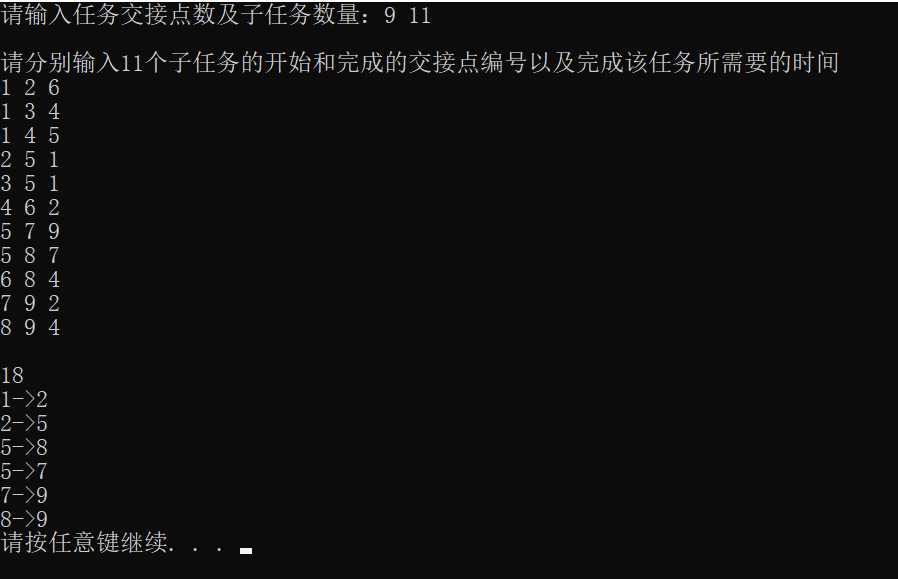
{

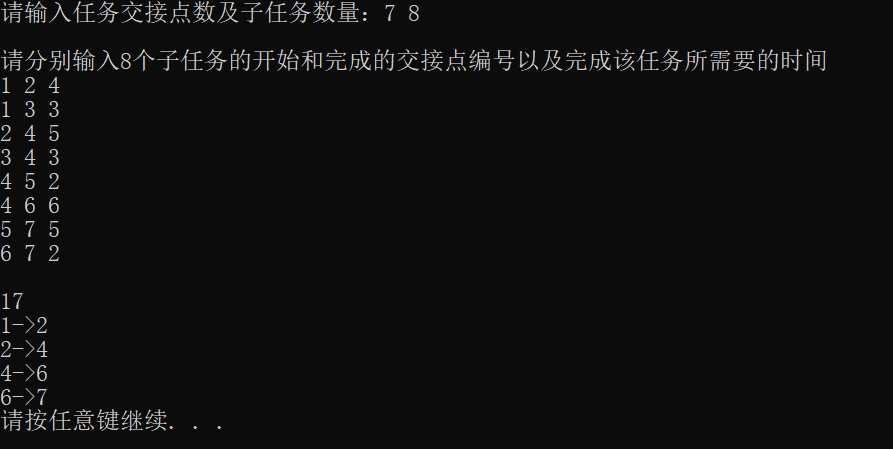
act.findCriticialPath(); //寻找关键路径

act.showCriticialPath(); //输出关键路径

}

### 3.6.2 总体系统截屏示例





# 4 测试

## 4.1 功能测试

### 4.1.1 读入顶点和边功能测试

**测试用例**：

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

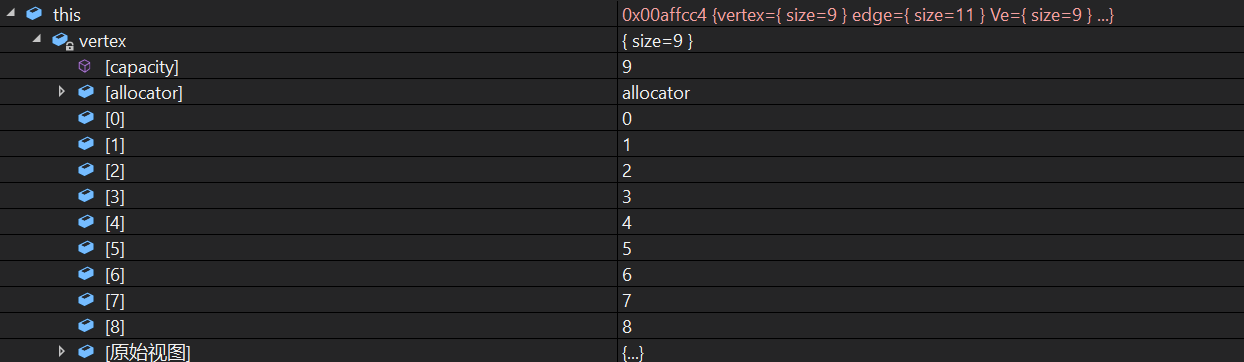
6 8 4

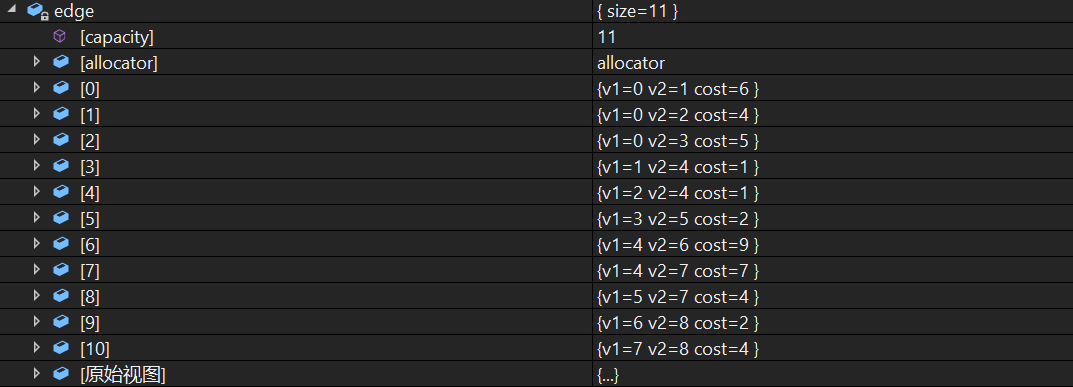
7 9 2

8 9 4

**预期结果**：正确存储网络中的顶点和边

**实验结果：**





### 4.1.2 取首邻居和下一个邻居功能测试

**测试用例：**正向拓扑排序时观察

j = getFirstNeighbour(i); 和 j = getNextNeighbour(i, j);

**预期结果：**

当该顶点还存在邻居时则取出该邻居，如果不存在更多的邻居则返回-1

**实验结果：**









### 4.1.3 拓扑排序功能测试

**测试用例：**

7 8

1 2 4

1 3 3

2 4 5

3 4 3

4 5 2

4 6 6

5 7 5

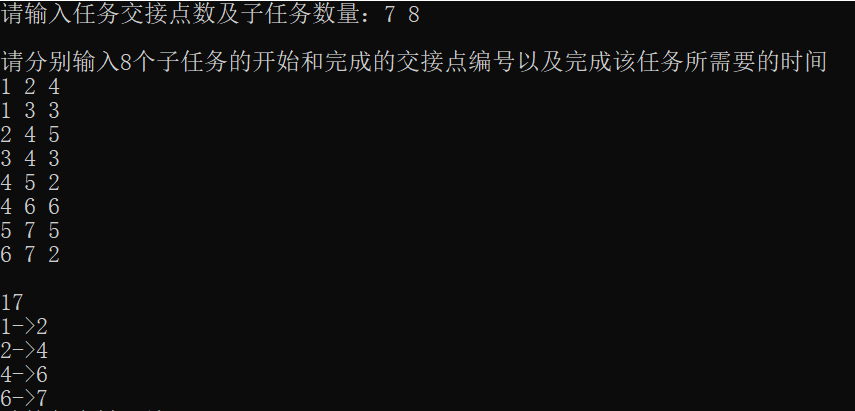
6 7 2

**预期结果：**正确进行拓扑排序，并返回该调度是否可行

**实验结果：**







### 4.1.4 寻找关键路径功能测试

**测试用例：**

7 8

1 2 4

1 3 3

2 4 5

3 4 3

4 5 2

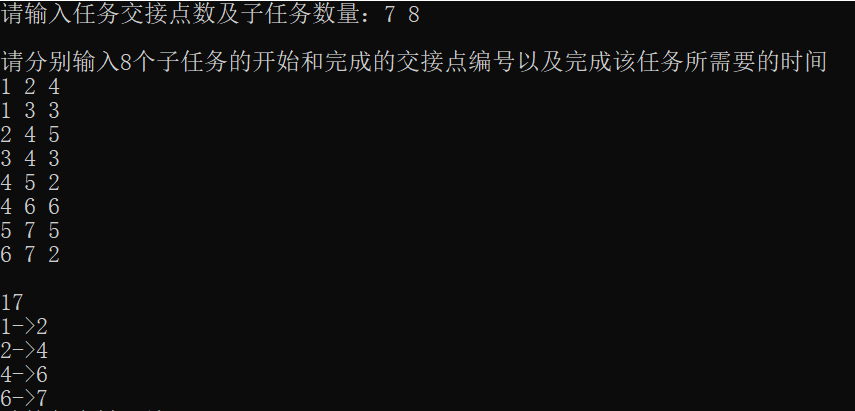
4 6 6

5 7 5

6 7 2

**预期结果：**在拓扑排序返回true的前提下（该调度可行），正确计算关键路径

**实验结果：**



### 4.1.5 输出关键活动功能测试

**测试用例：**

7 8

1 2 4

1 3 3

2 4 5

3 4 3

4 5 2

4 6 6

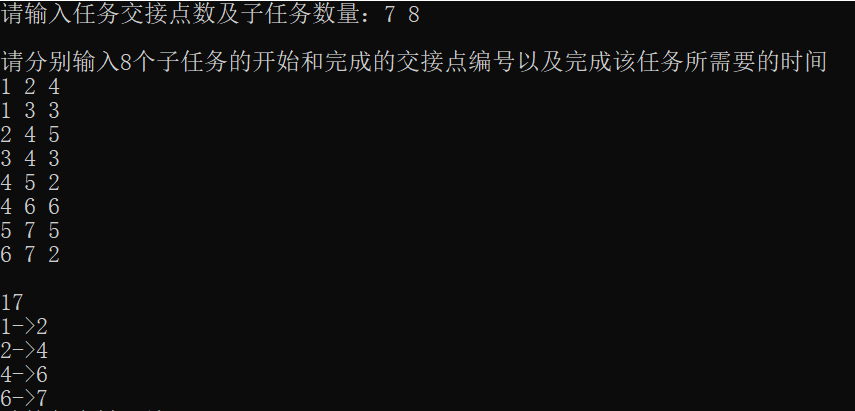
5 7 5

6 7 2

**预期结果：**在拓扑排序返回true的前提下，按照题目要求正确输出关键路径信息

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。

**实验结果：**



## 4.2 边界测试

### 4.2.1 调度任务不可行

**测试用例：**

4 5

1 2 4

2 3 5

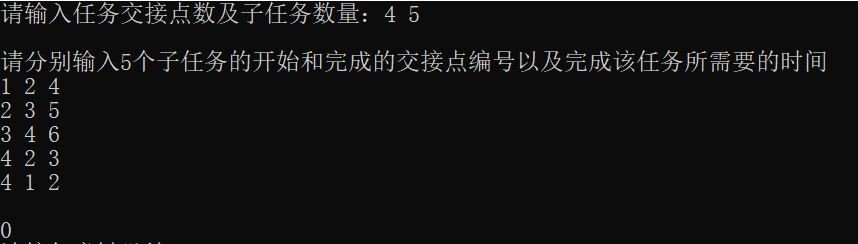
3 4 6

4 2 3

4 1 2

**预期结果：**调度任务不可行，输出0

**实验结果：**



### 4.2.2 多个起点

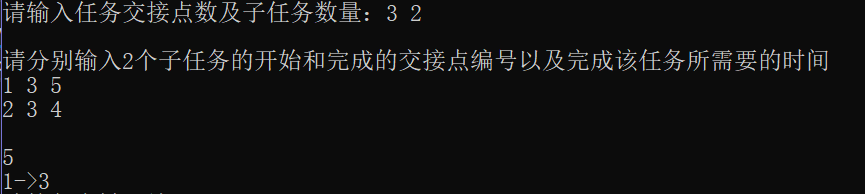
**测试用例：**

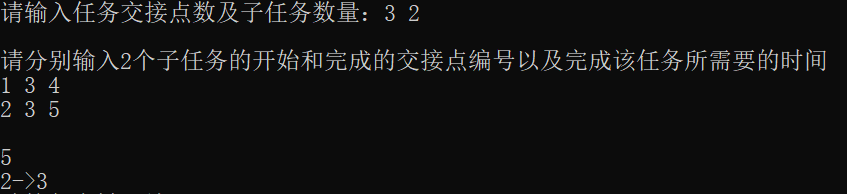
任务交接点数及子任务数均为3和2

两个子任务分别作为最长路径，且有多个起点

**预期结果：**程序仍能正确输出最长路径上的关键活动

**实验结果：**





### 4.2.3 多个终点

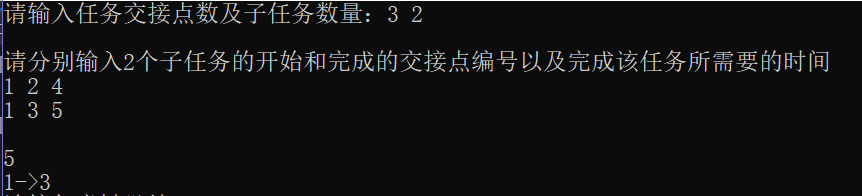
**测试用例：**

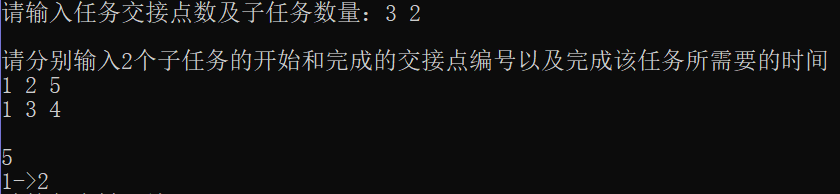
任务交接点数及子任务数均为3和2

两个子任务分别作为最长路径，且有多个终点

**预期结果：**程序仍能正确输出最长路径上的关键活动

**实验结果：**



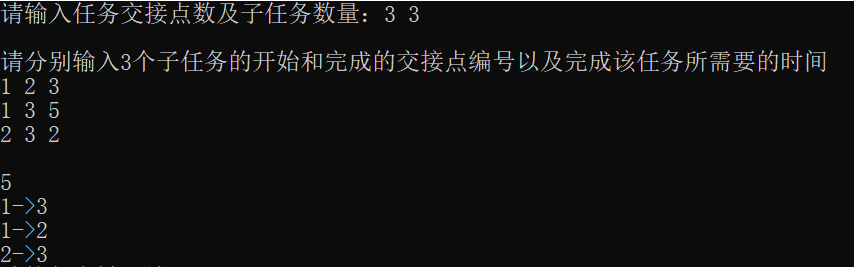


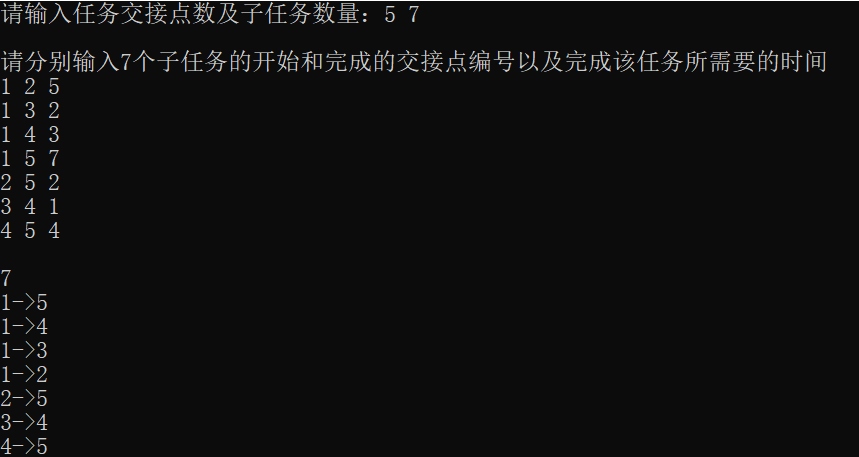
### 4.2.4 起点到终点有多条最长路径

**测试用例：**有多条同样长度的最长路径

**预期结果：**程序仍能正确求出关键路径和关键活动，并在第一个顶点相同时按照输入顺序的逆序输出结果

**实验结果：**





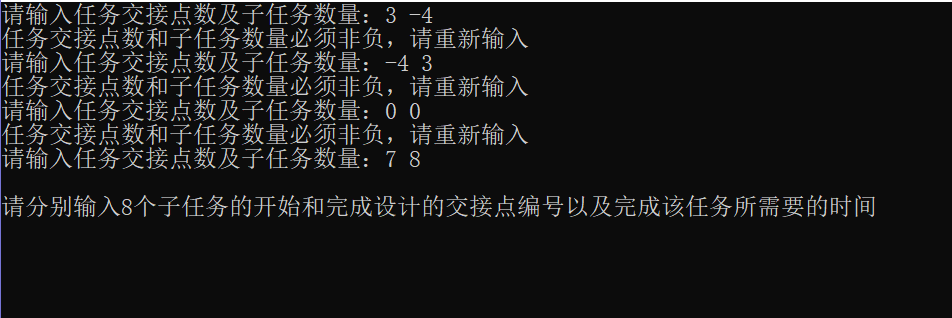
## 4.3 出错测试

### 4.3.1 任务交接点数或子任务数有误

**测试用例：**任务交接点数或子任务数为负数

**预期结果：**程序给出提示信息，程序正常运行不崩溃，并让用户继续输入。

**实验结果：**



### 4.3.2 读入的子任务时开始和完成的交接点编号不在任务交接点标号中

**测试用例：**输入的子任务的开始和完成的交接点标号不是任务交接点标号

**预期结果：**程序给出提示信息，程序正常运行不崩溃，并允许用户重新输入该子任务的开始和完成的交接点及完成该任务所需的时间

**实验结果：**

