# 1 需求

## 1.1 背景分析

假定一个工程项目由一组子任务构成，子任务之间有的可以并行执行，有的必须在完成了其它一些子任务后才能执行。“任务调度”包括一组子任务、以及每个子任务可以执行所依赖的子任务集。比如完成一个专业的所有课程学习和毕业设计可以看成一个本科生要完成的一项工程，各门课程可以看成是子任务。有些课程可以同时开设，比如英语和C程序设计，它们没有必须先修哪门的约束，有些课程则不可以同时开设，因为它们有先后的[依赖关系](https://so.csdn.net/so/search?q=%E4%BE%9D%E8%B5%96%E5%85%B3%E7%B3%BB&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/whisperlzw/article/details/_blank)，比如C程序设计和数据结构两门课，必须先学习前者。本实验项目是要求在任务调度问题中，如果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”。

本项目拟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。若调度方案不可行，则程序打印相应信息。

## 1.2 功能分析

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。

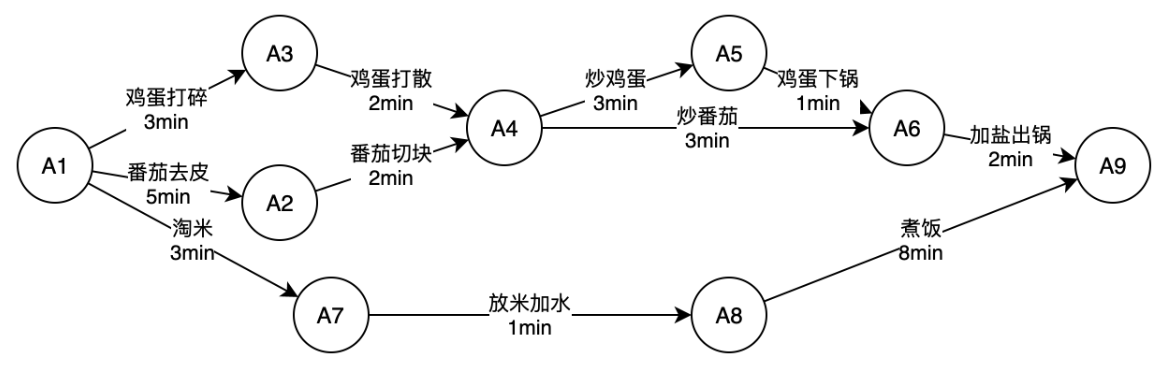
输出说明：如果任务调度不可行，则输出0；否则第一行输出完成整个项目所需要的时间，第2行开始输出所有关键活动，每个关键活动占一行，按照格式“v->W”输出，其中V和W为该任务开始和完成涉及的交接点编号。关键活动输出的顺序规则是：任务开始的交接点编号小者优先，起点编号相同时，与输入时任务的顺序相反。如下面测试用例2中，任务<5，7>先于任务<5，8>输入，而作为关键活动输出时则次序相反。

# 2 设计

## 2.1 数据结构设计

### 2.1.1 逻辑结构设计

本项目核心功能是解决任务调度问题并建立AOE网络，输出工程项目中的关键活动。可以看出项目各任务间存在依赖关系，比如活动A必须在活动B结束后才能进行，此时活动B依赖于活动A，这种依赖关系是偏序的，构成了“一对多”与“多对一”的映射，因此应当采用非线性逻辑结构存储。从整体来看，求关键路径问题即为图的问题，但各节点彼此间存在先后次序，所以可以采用有向图的逻辑结构存储。



### 2.1.2 存储结构设计

计算机存图有两种主要方法，分别是邻接矩阵与邻接表。邻接矩阵通过建立一个行列数为顶点个数的二维数组来存储图的相关信息，邻接表则采用链表的方

式。在一个顶点数为n,边数为e的图中，二者的时间复杂度比较如下：

|  |  |  |
| --- | --- | --- |
|  | 邻接矩阵 | 邻接表 |
| 构造功能 |  |  |
| 查找邻接对象功能 |  |  |
| 判断是否邻接功能 |  |  |
| 占用空间 |  |  |

结合多方面因素考虑，本项目采用邻接表的形式存储图。

## 2.2 类设计

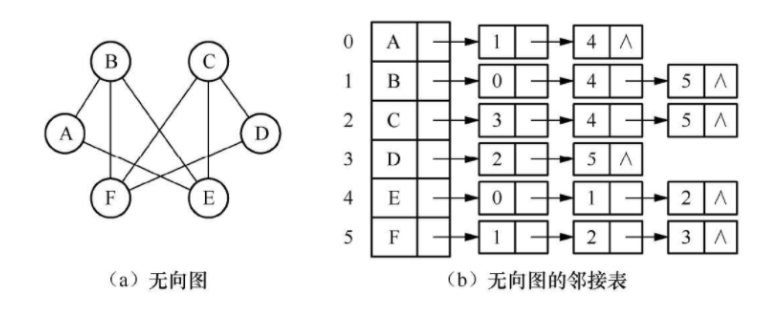
由于求关键路径的问题是图论中的经典问题，因此本项目主要设计的类为通过邻接表形式实现的图Graph类，以及实现Graph类所必须的顶点类Vertex与边类Edge。其中项目主体操作存储于Graph类的成员函数中，包括构造函数、析构函数、查找顶点在vertexList中的位置、对顶点拓扑排序、查找关键活动、打印关键活动、打印关键路径长度、打印图等。

### 2.2.1 顶点类、边类

本项目采用邻接表形式对图进行存储。邻接表是树与图结构的一般化存储方式，可以看成“带有索引数组的多个数据链表”构成的结构集合。在这样的结构中存储的数据被分成若干类，每一类的数据构成一个链表。每一类还有一个代表元素，称为该类对应链表的“表头”。所有“表头”构成一个表头数组，作为一个可以随机访问的索引，从而可以通过表头数组定位到某一类数据对应的链表。为了实现图类，本项目预先定义了两个辅助类结构，分别是顶点类与边类。

顶点类的protected属性存放顶点表以及指向与该边相邻第一条边的指针，public属性存放构造函数。边类的protected属性存放边的终点在顶点表中对应索引下标、边的权重以及指向下一条边的指针，public属性存放构造函数与析构函数。之后在定义图类时直接调用顶点类与边类的实现即可。





//边的模板类

**template** <**class** NameType, **class** DistType>

**class** Edge {

**protected**:

**int** dest;

DistType cost;

Edge<NameType, DistType>\* link;

**public**:

Edge():dest(0), cost(0), link(NULL){};

Edge(**int** D,DistType C):dest(D),cost(C),link(NULL){};

**friend** Graph<NameType, DistType>;

};

//顶点的模板类

**template** <**class** NameType, **class** DistType>

**class** Vertex {

**protected**:

    NameType data;

    Edge <NameType,DistType> \* adj;

**public**:

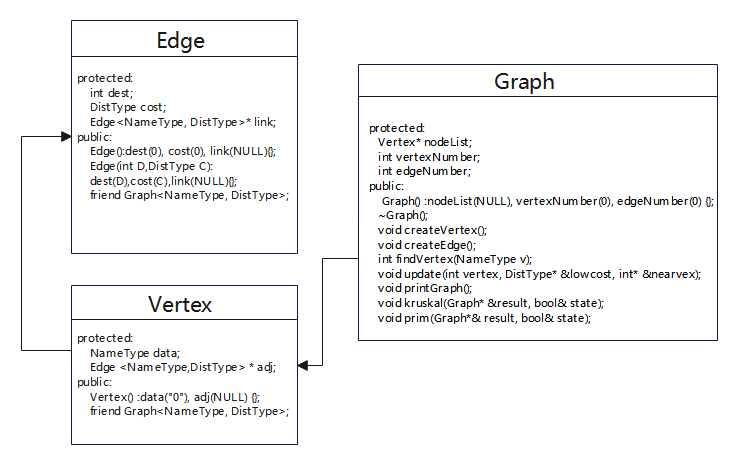
    Vertex() :data("0"), adj(NULL) {};

**friend** Graph<NameType, DistType>;

};

### 2.2.2 图类

图类的protected属性存储图的顶点表，顶点表为指向顶点的指针数组，每个顶点对象连接相应的边，通过链表的形式进行图信息的存储。图类的public属性存放图的各项操作，包括创建顶点表、创建边、查找顶点在vertexList中的位置、更新lowcost与nearvex数组、打印图、构建最小生成树Kruskal、构建最小生成树Prim等。为了提高代码的重用性，项目在定义图类的时候采用类模板的形式，使用的模板分别是顶点表的名称类型NameType和边的权重类型DistType。



//图的模板类

**template** <**class** NameType, **class** DistType>

**class** Graph {

**protected**:

    Vertex<NameType, DistType>\* nodeList;

**int** vertexNumber;

**int** edgeNumber;

**public**:

    Graph() :nodeList(NULL), vertexNumber(0), edgeNumber(0) {};

    ~Graph();

**void** createVertex();

**void** createEdge();

**int** findVertex(NameType v);

**void** update(**int** vertex, DistType\* &lowcost, **int**\* &nearvex);

**void** printGraph();

**void** kruskal(Graph<NameType, DistType>\* &result, **bool**& state);

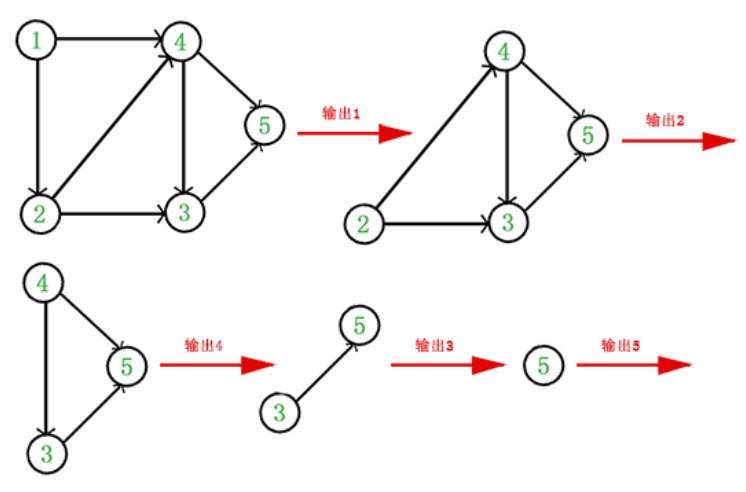
**void** prim(Graph<NameType, DistType>\*& result, **bool**& state);       

# 3 实现

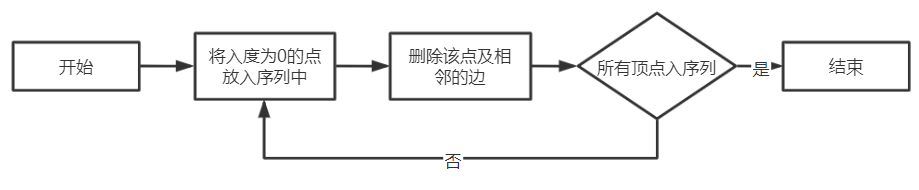
## 3.1 对顶点拓扑排序的实现

### 3.1.1 对顶点拓扑排序实现思路

拓扑排序就是将AOV-网中所有顶点排成一个线性序列，若在AOV网中由顶点vi到顶点 vj有一条路径，则在该线性序列中的顶点 Vi必定在顶点Vj之前。拓扑排序的实现需要定义两个辅助数组结构分别用来存放各顶点入度和记录拓扑排序的顶点序号。从第一个无入度的顶点开始，将所有无入度的顶点依次输出并从已有图中摘除，同时将此结点与其他结点所依附的边摘除，最终输出的顶点序列就是拓扑排序序列，此处需要注意的是，有些有向无环图的拓扑排序序列的结果并不唯一，但并不影响最后求解关键活动。



### 3.1.2 对顶点拓扑排序实现流程图



### 3.1.3 对顶点拓扑排序实现代码

//对顶点拓扑排序

**template** <**class** NameType,**class** DistType>

**bool** Graph<NameType, DistType>::TopologicalSort() {

**int** top = -1;

**for** (**int** i = 0; i < vertexNumber; i++) {

**if** (count[i] == 0) {

            count[i] = top;

            top = i;

        }

    }

**for** (**int** i = 0; i < vertexNumber; i++) {

**if** (top == -1)

**return** **false**;

**else** {

            topo[i] = top;

            top = count[top];

            Edge<NameType, DistType>\* p = nodeList[topo[i]].adj;

**while** (p != NULL) {

                count[p->dest] -= 1;

**if** (count[p->dest] == 0) {

                    count[p->dest] = top;

                    top = p->dest;

                }

                p = p->link;

            }

        }

    }

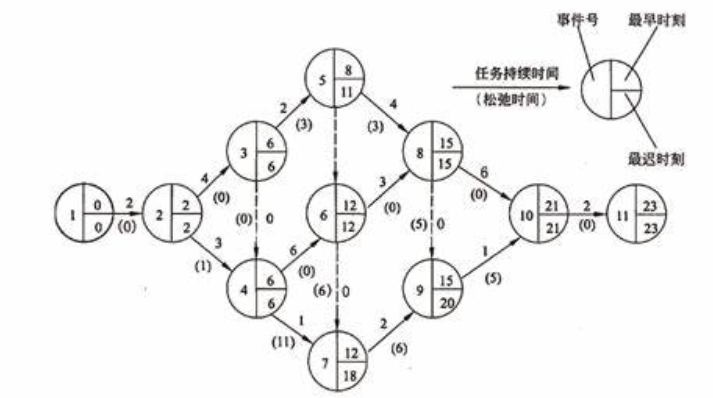
**return** **true**;

}

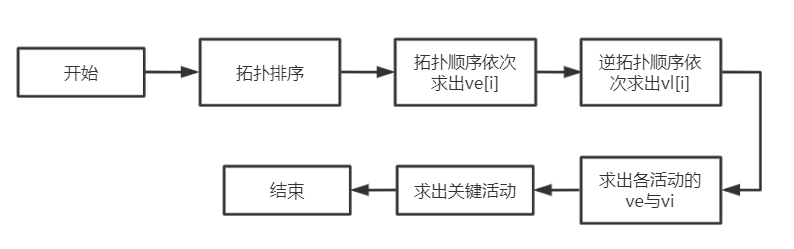
## 3.2 查找关键活动功能的实现

### 3.2.1 查找关键活动功能实现思路

求关键活动时需要求出事件i的最早发生时间Ve(i)、事件i的最晚发生时间 Vl(i)、活动ai的最早开始时间e(i)、活动ai的最晚开始时间l(i)，遍历所有活动，其中e(i)=l(i)的即为关键活动。当拓扑排序完成之后，程序从源点V1出发，令Ve[1]=0，按拓扑序列求各顶点的Ve[i]，之后从汇点Vn出发，令Vl[n]=Ve[n]，按逆拓扑序列求其余各顶点的Vl[i]。根据已求得的ve,vl数组可求出各活动的最早开始时间e(i)和最晚开始时间l(i)，进而求出所有的关键活动。



### 3.2.2 查找关键活动功能实现流程图



### 3.2.3 查找关键活动功能实现代码

//查找关键活动

**template** <**class** NameType,**class** DistType>

**void** Graph<NameType, DistType>::criticalPath() {

**if** (!TopologicalSort()) {

        cout << 0 << endl;

**return**;

    }

    DistType\* ve = **new** DistType[vertexNumber];

    DistType\* vl = **new** DistType[vertexNumber];

**for** (**int** i = 0; i < vertexNumber; i++) {

        ve[i] = 0;

        vl[i] = 0;

    }

**for** (**int** i = 0; i < vertexNumber; i++) {

        Edge<NameType, DistType>\* p = nodeList[topo[i]].adj;

**while** (p != NULL) {

**if** (ve[topo[i]] + p->cost > ve[p->dest])

                ve[p->dest] = ve[topo[i]] + p->cost;

            p = p->link;

        }

    }

**for** (**int** i = 0; i < vertexNumber; i++)

        vl[i] = ve[vertexNumber - 1];

**for** (**int** i = vertexNumber - 1; i >= 0; i--) {

        Edge<NameType, DistType>\* p = nodeList[topo[i]].adj;

**while** (p != NULL) {

**if** (vl[p->dest] - p->cost < vl[topo[i]])

                vl[topo[i]] = vl[p->dest] - p->cost;

            p = p->link;

        }

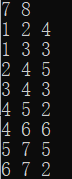
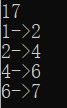
    }

    printLength(ve, vl);

    printPath(ve, vl);

}

### 3.2.3 查找关键活动功能实现样例

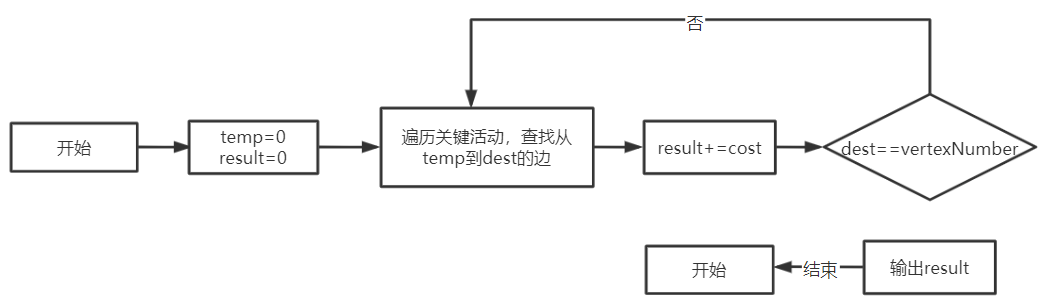
## 3.3 计算关键路径长度功能的实现

### 3.3.1 计算关键路径长度功能实现思路

求关键路径时，可以用深度优先搜索搜索出现的路径。首先需要设置一个一维数组用来标记关键事件，当元素为0时，该事件不是关键事件；当元素为1时，该事件是关键事件；当元素为2时，该事件是关键事件且已被访问过。再设置一个二维数组用来标记关键活动，当两事件都为关键事件且两事件间的活动为关键活动时，才会继续往下搜索。每搜索到一个关键事件时，将其压入栈，将该事件标记为 2，表示已经访问过。直到最后一个事件进栈，将栈中所有事件输出，并令栈顶元素出栈，将其事件标记为 1，表示没有访问，再次搜索其它路径。

但由于之前我们已经求得了拓扑排序，而关键路径上的节点顺序是满足拓扑排序的，因此我们可以遍历关键活动，从拓扑排序第一个节点寻找到达拓扑排序最后一个节点的通路即可，这样可以简化寻找关键路径的步骤。

### 3.3.2 计算关键路径长度功能实现流程图



### 3.3.3 计算关键路径长度功能实现代码

//打印关键路径长度

**template** <**class** NameType, **class** DistType>

**void** Graph<NameType, DistType>::printLength(DistType\* ve, DistType\* vl) {

**int** result = 0;

**int** temp = topo[0], end = topo[vertexNumber - 1];

**while** (temp != end) {

        Edge<NameType, DistType>\* p = nodeList[temp].adj;

**while** (p != NULL) {

**if** (ve[temp] == vl[p->dest] - p->cost) {

                result += p->cost;

                temp = p->dest;

**continue**;

            }

            p = p->link;

        }

    }

    cout << result << endl;

}

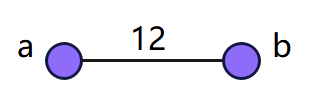
# 4 测试

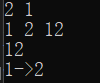
## 4.1 常规测试

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 输入 | 预期输出 | 实际输出 |
| 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 |  |
| 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 |  |
| 3 | 11 14  1 2 4  1 3 3  2 4 5  3 4 3  4 5 1  4 6 6  5 7 5  6 7 2  8 3 7  9 3 7  9 10 6  4 10 2  10 6 5  6 11 4 | 21  3->4  4->10  6->11  8->3  9->3  10->6 |  |
| 4 | 4 5  1 2 4  2 3 5  3 4 6  4 2 3  4 1 2 | 0 |  |

## 4.2 边界测试

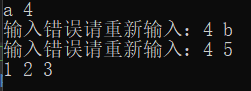
只有两个顶点一条边的情况。



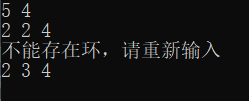


## 4.3 错误测试

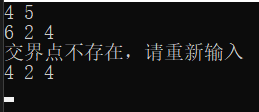
### 4.3.1 输入顶点数和边数错误



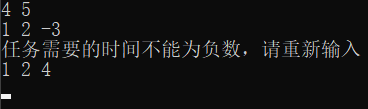
### 4.3.2 存在环



### 4.3.3 交界点不存在



### 4.3.4 输入任务时间为负数



### 4.3.5 任务交界点数量为负数



### 4.3.6 任务数量为负数

