**项目说明文档**

**数据结构课程设计**

**——关键活动**

作 者 姓 名： 张梓瀚

学 号： 2051943

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 项目分析 - 2 -](#_Toc121489428)

[1.1项目背景 - 2 -](#_Toc121489429)

[1.2 项目要求 - 3 -](#_Toc121489430)

[1.2.1 功能要求 - 3 -](#_Toc121489431)

[1.2.2 输入格式 - 3 -](#_Toc121489432)

[1.2.3 输出格式 - 3 -](#_Toc121489433)

[1.2.4 项目示例 - 4 -](#_Toc121489434)

[2 项目设计 - 4 -](#_Toc121489435)

[2.1 数据结构设计 - 4 -](#_Toc121489436)

[2.2 类设计 - 4 -](#_Toc121489437)

[2.2.1 链表及链表结点类（ListNode、List） - 4 -](#_Toc121489438)

[2.2.2 栈类（myStack） - 6 -](#_Toc121489439)

[2.2.3 队列类（Queue） - 6 -](#_Toc121489440)

[2.3 关键活动算法 - 7 -](#_Toc121489441)

[2.3.1 主要思想 - 7 -](#_Toc121489442)

[2.3.2代码 - 7 -](#_Toc121489443)

[3 项目测试 - 9 -](#_Toc121489444)

[3.1 简单情况测试 - 9 -](#_Toc121489445)

[3.2 一般情况测试，单个起点和单个终点 - 9 -](#_Toc121489446)

[3.3 不可行的情况测试 - 10 -](#_Toc121489447)

# 1 项目分析

## 1.1项目背景

通常把计划、施工过程、程序流程等都当做一个工程。除了很小的工程外，一般都把工程分为若干个叫做“活动”的子工程。完成了这些活动，这个工程就可以完成了。例如，计算机专业学生的学习就是一个工程，每一门课程的学习就是整个工程的一些活动。我们可以利用有向图表示一个工程，在这种有向图中，用顶点表示活动，用有向边表示活动进行的先后关系，这种有向图成为定点表示活动的网络（activity on vertices），记作AOV网络。

与AOV网络密切相关的另一种网络就是AOE网络。如果在无有向环的有向带权图中用有向边表示一个工程中的各项活动，用有向边上的权值表示活动的持续时间，用顶点表示事件，则这样的有向图叫做用边表示活动的网络（activity on edges），记作AOE网络。

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尽相同。完成不同路径的活动所需的时间虽然不同，但只有各条路径上的所有活动都完成了，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。

要找出关键路径，必须找出关键活动，即不按期完成就会影响整个工程完成的活动。关键路径上的所有活动都是关键活动。因此，只要找到了关键活动，就可以找到关键路径。

## 1.2 项目要求

### 1.2.1 功能要求

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。

### 1.2.2 输入格式

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。

### 1.2.3 输出格式

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。

### 1.2.4 项目示例

|  |  |  |  |
| --- | --- | --- | --- |
| 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 | 简单情况测试 |
| 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 |

# 2 项目设计

## 2.1 数据结构设计

本题需要对顶点进行拓扑排序和逆拓扑排序，因此考虑使用一个栈存储拓扑排序的结果，在需要逆拓扑序时顶点依次出栈的顺序就是逆拓扑序。同时，记录答案过程中使用队列存储关键活动。

## 2.2 类设计

本题需要实现的类主要包括栈和队列类，为了提高代码的泛用性，我使用模板类的方法编写这两个类的代码； 同时为了实现代码复用，减少开发时间，我使用了之前项目编写的链表类和链表结点类，将栈和队列都用链式方法实现。

### 2.2.1 链表及链表结点类（ListNode、List）

template <typename T>

class ListNode

{

T\* data;

ListNode<T>\* next;

public:

//默认构造函数

ListNode<T>();

//赋值构造函数

ListNode<T>(const T& src);

//获取数据

const T& get\_data() const;

//获取后继结点

const ListNode<T>\* get\_next() const;

//修改数据

void set\_data(const T& src);

friend class List<T>;

};

template <typename T>

class List

{

private:

ListNode<T>\* start;

int size = 0;

inline ListNode<T>\* findLastByData(const T& tar);

inline ListNode<T>\* findLastByPos(int pos);

public:

//默认构造函数

List();

//析构函数

~List();

//获取头结点

const ListNode<T>\* get\_head();

//查找指定位置的结点

ListNode<T>\* Locate(int pos);

//向链表尾部插入结点

inline void insert(const T& s);

//向特定位置插入结点

inline bool insert(const T& s, int pos);

//根据数据内容查找结点

inline ListNode<T>\* findByData(const T& s);

//根据在链表中的位置查找结点

inline ListNode<T>\* findByPos(int tarID);

//修改结点数据

inline void modify(T tar, const T& new\_s);

//删除指定位置的结点

inline void remove(int tarID);

//按序打印链表所有结点

inline void printAll(ostream& os = cout);

//获取链表长度

int get\_list\_size();

//判断链表是否为空

bool is\_empty();

};

### 2.2.2 栈类（myStack）

//基于List类实现的链式栈

template<typename T>

class myStack : private List<T> {

public:

myStack();

//向链表头部（栈顶）插入结点

void push(const T& data);

//返回第一个结点的数据

const T top();

//删除第一个结点

void pop();

//判断栈是否为空

bool empty();

};

### 2.2.3 队列类（Queue）

template<typename T>

class Queue : List<T>

{

T head;

public:

//元素入队（插入链表末尾）

void enqueue(const T& data);

//元素出队（删去链表第一个结点）

const T& dequeue();

//判断队列是否为空

bool isEmpty();

};

## 2.3 关键活动算法

### 2.3.1 主要思想

首先定义几个与计算关键活动有关的量：

时间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。

时间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。

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]。

活动ak的**最迟允许开始时间Al[k]**：设活动ak在有向边<Vi,Vj>上，则Al[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。

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**，也叫作松弛时间(slack time)。Al[k] == Ae[k]表示活动ak是没有时间余量的关键活动。

为了找出关键活动，就要求的各个活动的Ae[k]与Al[k]，以判别是否Al[k] == Ae[k]；而为了求得Ae[k]与Al[k]，就要先求得从源点V0到各个顶点Vi的最早可能开始时间Ve[i]与最迟允许开始时间Vl[i]。

首先对各顶点做拓扑排序，在拓扑序下依次求出各顶点的Ve：

ve[i]=max(ve[i],ve[k]+path[k][i])

再按逆拓扑序求出Vl：

vl[i]=min(vl[i],vl[k]−path[i][k])

对于每条有向边<Vi,Vj>表示的活动ak，有

Ae[k]=Ve[i]

Al[k] = Vl[j] – dur(<Vi,Vj>)

最后依次扫描有向边，满足Ae[k]==Al[k]的活动ak即为关键活动。

### 2.3.2代码

int main()

{

memset(head, -1, sizeof(head));

cin >> n >> m;

//建立邻接表

for (int i = 1; i <= m; ++i)

{

cin >> edge[i].x >> edge[i].y >> edge[i].w;

indegree[edge[i].y]++; //终点的入度加一

edge[i].next = head[edge[i].x];

head[edge[i].x] = i;

}

if (TopologicalSort() == 0)

{

cout << 0 << endl << "不存在关键路径,存在环" << endl;

return 0;

}

memset(vl, inf, sizeof(vl));

vl[n] = ve[n]; //最后一个事件的最迟发生事件等于最早发生时间

while (!s.empty()) //逆拓扑排序求vl[i]

{

//拓扑序入栈，出栈顺序即为逆拓扑序

auto tmp = s.top();

s.pop();

int k = head[tmp];

while (k != -1)

{

if (vl[tmp] > vl[edge[k].y] - edge[k].w)

vl[tmp] = vl[edge[k].y] - edge[k].w;

k = edge[k].next;

}

}

Queue<Edge>critical\_activities;

for (int i = 1; i <= n; i++)

{

int k = head[i];

while (k != -1)

{

e = ve[i]; //该条边的起点代表事情，该条边表示的活动的最早发生时间就等于起点代表的事情的最早发生时间

l = vl[edge[k].y] - edge[k].w; //活动的最迟发生时间

if (l == e)

critical\_activities.enqueue({ i,edge[k].y,edge[k].w,-1 });

k = edge[k].next;

}

}

cout << ve[n] << endl;

Edge\* edge1;

while (!critical\_activities.isEmpty())

{

edge1 = new Edge(critical\_activities.dequeue());

cout << edge1->x << "->" << edge1->y << endl;

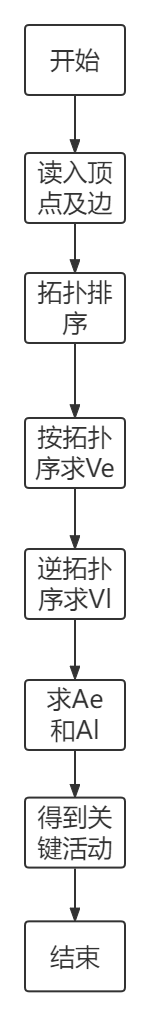
}

return 0;

}

}

### 2.3.3 流程图



# 3 项目测试

## 3.1 简单情况测试

测试用例：

预期结果：

实验结果：



## 3.2 一般情况测试，单个起点和单个终点

测试用例：

预期结果：

实验结果：



## 3.3 不可行的情况测试

测试用例：

预期结果：

实验结果：

