项目说明文档

数据结构课程设计

——关键活动

作 者 姓 名： 祝新元

学 号： 1751629

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](#_Toc533777322)

[1.1 项目内容 1](#_Toc533777323)

[1.2项目功能要求 1](#_Toc533777324)

[2 设计 2](#_Toc533777325)

[2.1 数据结构设计 2](#_Toc533777326)

[2.2 类结构设计 2](#_Toc533777327)

[2.3 成员与操作设计 2](#_Toc533777328)

[2.4 系统设计 4](#_Toc533777329)

[3 实现 5](#_Toc533777330)

[3.1 插入结点v 5](#_Toc533777331)

[3.1.1核心代码 5](#_Toc533777332)

[3.2 插入边（v1,v2） 5](#_Toc533777333)

[3.2.1核心代码 5](#_Toc533777334)

[3.3 计算关键路径 6](#_Toc533777335)

[3.3.1核心代码 6](#_Toc533777336)

[3.3 拓扑排序判断有没有回路 7](#_Toc533777337)

[3.3.1核心代码 7](#_Toc533777338)

[4 测试 9](#_Toc533777339)

[4.1 测试用例 9](#_Toc533777340)

[4.1.1简单情况测试 10](#_Toc533777341)

[4.1.2 一般情况测试，单个起点和单个终点 11](#_Toc533777342)

[4.1.3 不可行的方案测试 12](#_Toc533777343)

# 1 分析

## 1.1 项目内容

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字任务需要的时间，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项目功能要求

1. 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。
2. 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

考虑到要实现任务调度，每个任务都有完成时间，所以考虑采用用边表示活动的网络，也就是AOE网络。用有向图来描述和分析一项工程的计划和实施过程，一个工程常被分为多个小的子工程，这些子工程被称为活动（Activity)，在带权有向图中若以顶点表示事件，有向边表示活动，边上的权值表示该活动持续的时间，这样的图简称为AOE网。

## 2.2 类结构设计

有向图中，用顶点表示活动，用有向边表示活动之间开始的先后顺序，则称这种有向图为AOV（Activity On Vertex）网络；AOV网络可以反应任务完成的先后顺序（拓扑排序）。在AOV网的边上加上权值表示完成该活动所需的时间，则称这样的AOV网为AOE（Activity On Edge）网

本次采用三个抽象数据类型（ADT）——边结点类（Edge），点结点类（Vertex）和有向图类（Graphlnk），而各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struct描述边结点类（LinkNode）和点结点类（Vertex），这样使得边结点类（LinkNode）和点结点类（Vertex）可以访问有向图类（Graphlnk）。为了书写方便，采用邻接表的形式构造图。同时为体现面向对象编程的思想，在设计时所采用的数据结构均为模板类，template<class T,class E>，之后不在重复。

## 2.3 成员与操作设计

**边结点类（**Edge**）**

**公有成员：**

int dest; //the other vertex

E cost; //value

Edge<T,E>\* link; //next point

**公有操作：**

Edge():dest(0),cost(0),link(NULL){}//构造函数

Edge(int num,E weight):dest(num),cost(weight),link(NULL){}//具有默认参数的构造函数

**点结点类（**Vertex**）**

**公有成员：**

T data; //the name of the Vertex

Edge<T,E> \*adj; //\*first to the list

**有向图类（**Graphlnk**）**

**私有成员：**

Vertex<T,E> \*NodeTable; //顶点表（各边链表的头结点）

int maxVertices; //max of Vertices

int numEdges; //the number of edges

int numVertices; //the number of vertices

**公有操作：**

Graphlnk<T,E>::Graphlnk(int sz)

// Graphlnk的构造函数，构造一个邻接表

Graphlnk<T,E>::~Graphlnk()

// Graphlnk的析构函数，实现对内存的回收

int Graphlnk<T,E>::getFirstNeighbor(int v)

//得到v的第一个邻居

int Graphlnk<T,E>::getNextNeighbor(int v,int w)

//得到v的邻居中w后面一个

E Graphlnk<T,E>::getWeight(int v1,int v2)

//得到边（v1，v2）的权值

bool Graphlnk<T,E>::insertVertex(const T& v)

//在有向图中插入结点v

bool Graphlnk<T,E>::removeVertex(int v)

//从有向图中删除结点v

bool Graphlnk<T, E>::insertEdge(int v1, int v2, E weight)

//插入边(v1,v2)

bool Graphlnk<T, E>::removeEdge(int v1, int v2)

//删除边(v1,v2)

## 2.4 系统设计

先读入任务交接点的数量N和子任务的数量M。建立点大小为N的有向图，然后读入M条边，判断他是否有回路，若是没有则开始进行关键路径的计算。

# 3 实现

## 3.1 插入结点v

### 3.1.1核心代码

template<class T,class E>

bool Graphlnk<T,E>::insertVertex(const T& v)

{

if(numVertices==maxVertices) return false; //如果有向图已经满了则返回错误

NodeTable[numVertices].data=v;

numVertices++;//加入到邻接表中

return true;

}

## 3.2 插入边（v1,v2）

### 3.2.1核心代码

template<class T,class E>

bool Graphlnk<T, E>::insertEdge(int v1, int v2, E weight)

{

if (v1 >= 0 && v1 < numVertices && v2 >= 0 && v2 < numVertices)

{

Edge<T, E> \*p = NodeTable[v1].adj;

while (p != NULL && p->dest !=v2)

{

p = p->link;

}

if (p != NULL){return false;} // if find this edge, return false

//if do not find ,then create new node

p = new Edge<T, E>;

p->dest = v2;

p->cost = weight;

p->link = NodeTable[v1].adj; //link to v1's list

NodeTable[v1].adj = p;

numEdges++;

return true;

}

return false;

}

## 3.3 计算关键路径

### 3.3.1核心代码

//计算关键路径

template<class T,class E>

void CriticalPath(Graphlnk<T, E>& G)

{

int i, j, k;

E Ae, Al, w;

int n = G.getNumVertices();//n为结点数

E \*Ve = new E[n];

E \*Vl = new E[n];

int sum=0;

for (i = 0; i < n; i++)//初始化Ve[]

{

Ve[i] = 0;

}

for (i = 0; i < n; i++) //正向计算Ve[]

{

j = G.getFirstNeighbor(i);

while (j != -1)//遍历所有邻居

{

w = G.getWeight(i, j);

if (Ve[i] + w > Ve[j]) Ve[j] = Ve[i] + w;//最早可能开始时间=max（前+边）

j = G.getNextNeighbor(i, j);

}

}

Vl[n-1] = Ve[n-1];

for (j = n - 2; j >= 0; j--) //逆向计算Vl[]

{

k = G.getFirstNeighbor(j);

Vl[j]=Vl[k]-G.getWeight(j, k);

while (k != -1)

{

w = G.getWeight(j, k);

if (Vl[k] - w < Vl[j]) Vl[j] = Vl[k] - w;//Vl=min（后-边）

k = G.getNextNeighbor(j, k);

}

}

int start=0;

int end=0;

for (i = 0; i < n; i++) //求各活动的e和l

{

j = G.getFirstNeighbor(i);

if(end==i) start=i;

while (j != -1)

{

Ae = Ve[i];

Al = Vl[j] - G.getWeight(i, j);

if (Al == Ae)

{

end=j;

}

j = G.getNextNeighbor(i,j);

}

if(start==i) sum += G.getWeight(start, end); //如果确定了这条边为关键边则加到sum

}

cout<<sum<<endl; //输出最短时间

for (i = 0; i < n; i++)

{

j = G.getFirstNeighbor(i);

while (j != -1)

{

Ae = Ve[i];

Al = Vl[j] - G.getWeight(i, j);

if (Al == Ae)

{

cout << G.getValue(i)+1 << "->" << G.getValue(j)+1 << endl;

}

j = G.getNextNeighbor(i,j);

}

}

delete[]Ve;

delete[]Vl;

}

## 3.3 拓扑排序判断有没有回路

### 3.3.1核心代码

//拓扑排序

template<class T,class E>

bool TopologicalSort(Graphlnk<T, E>& G,int M)

{

int i, w, v;

int top = -1; //入度为零顶点的栈初始化

int n = G.getNumVertices(); //网络中顶点个数

int \*count = new int[n]; //入度数组兼入度为零顶点栈

int \*topsort = new int[n];

int sum=0; //记录拓扑排序后得到的数组

for (i = 0; i < n; i++)

{

count[i] = 0;

}

int v1, v2, cost;

for (i = 0; i<M; i++)

{

cin >> v1 >> v2 >> cost;

v1--;

v2--;

count[v2]++;

G.insertEdge(v1, v2, cost);

}

for (i = 0; i < n; i++) //检查网络中所有的顶点

{

if (count[i] == 0) //入度为零的顶点进栈

{

count[i] = top;

top = i;

}

}

for (i = 0; i < n; i++) //期望输出n个顶点

{

if (top == -1) //中途栈空，转出,表示有回路

{

cout<<0;

return false;

}

else //继续拓扑排序

{

v = top;

top = count[top]; //v退栈

//将得到的拓扑排序结果存入topsort数组

topsort[sum]=G.getValue(v);

sum++;

w = G.getFirstNeighbor(v);

while (w != -1) //扫描出边表

{

if (--count[w] == 0) //邻接顶点入度减少1

{

count[w] = top;

top = w; //顶点入度减少到0，进栈

}

w = G.getNextNeighbor(v, w);

}

}

}

cout << endl;

if(sum!=n) return false;

return true;

}

# 4 测试

## 4.1 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 |

### 4.1.1简单情况测试

**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**：

17

1 –>2

2 –>4

4 –>6

6 –>7

**实验结果**



### 4.1.2 一般情况测试，单个起点和单个终点

**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：**

18

1 –>2

2 –>5

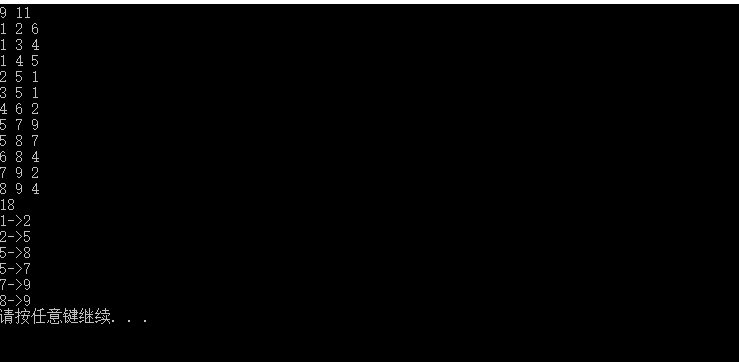
5 –>8

5 –>7

7 –>9

8 –>9

**实验结果：**



### 4.1.3 不可行的方案测试

**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：**

0

**实验结果：**

