项目说明文档

数据结构课程设计

——关键活动

作 者 姓 名： 王加炜

学 号： 2150265

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

# 分析

## 1.1背景分析

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”。

## 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。

# 二．设计

## 2.1数据结构设计

如上功能分析所示，需要一个存储结构来存储各边信息，因此采用图的数据结构来对所有边的排列进行存储。

## 2.2类结构设计

因为邻接矩阵相比邻接表更能反映出顶点之间的相邻关系，所以我决定用邻接矩阵的形式来存储图的信息。在邻接矩阵中，用顶点数组存储各个顶点，用edge\_二维数组存储顶点间的相邻关系，用edge\_\_二维数组存储各个边的输入优先级。而边的信息存储在一个结构体中，该结构体包含首尾顶点，优先级以及边的权值四个数据成员。

## 2.3成员与操作设计

Graphmtx类：

class graphmtx//邻接矩阵(有向边)

{

protected:

char\* vertex;//顶点数组

int vertex\_num;//顶点个数

int edge\_num;//边个数

int \*\*edge\_;//邻接矩阵存权值

int \*\*edge\_\_;//邻接矩阵存pri

public:

graphmtx()

{

vertex\_num = 0;

edge\_num = 0;

vertex = new char[defaultsize];

for (int i = 0; i < defaultsize; i++)//动态生成edge\_数组

{

edge\_[i] = new int[defaultsize];

}

for (int i = 0; i < defaultsize; i++)//动态生成edge\_\_数组

{

edge\_\_[i] = new int[defaultsize];

}

for (int i = 0; i < defaultsize; i++)//对edge\_数组初始化

{

for (int j = 0; j < defaultsize; j++)

{

edge\_[i][j] = (i == j) ? 0 : maxweight;

}

}

for (int i = 0; i < defaultsize; i++)//对edge\_\_数组初始化

{

for (int j = 0; j < defaultsize; j++)

{

edge\_\_[i][j] =-1;

}

}

}

~graphmtx()

{

delete vertex;

}

char get\_vertex(int i)

{

return vertex[i];

}

int get\_edge\_num()

{

return edge\_num;

}

int get\_vertex\_num()

{

return vertex\_num;

}

void insert\_vertex(char a);

void insert\_edge(edge a);

int getvertexpos(char a);

int getfirstneighbor(int a);

int getnextneighbor(int w, int v);

int getweight(int a, int b)

{

return edge\_[a][b];

}

int getpri(int a, int b)

{

return edge\_\_[a][b];

}

};

Edge结构体：

struct edge {//边信息

int pri;

char head;

char tail;

int weight;//权值

edge& operator=(edge& x)//=运算符重载

{

pri = x.pri;

head = x.head;

tail = x.tail;

weight = x.weight;

return \*this;

}

bool operator<=(edge& x)//<=运算符重载

{

return (weight <= x.weight);

}

bool operator>(edge& x)//>运算符重载

{

return(weight > x.weight);

}

bool operator==(edge& x)

{

if (weight == x.weight && head == x.head && tail == x.tail&& pri==x.pri)

{

return 1;

}

else

{

return 0;

}

}

};

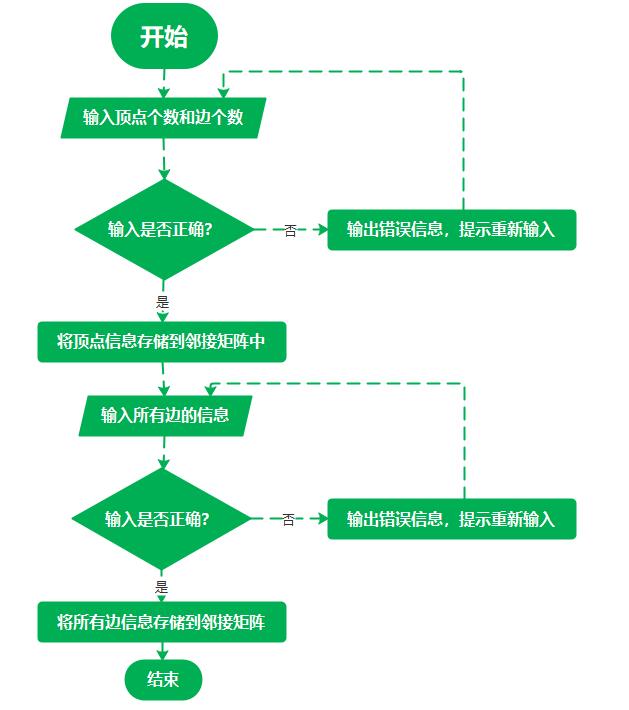
## 2.4系统设计

用户首先在input函数中输入图的信息，将该信息存储到邻接表的实例化对象中；其次调用关键路径函数对邻接表中存储的边信息进行处理，得到Ae和Ai相等的活动，将其列入关键活动中；最后通过output函数对整个活动所需事件进行计算并输出，同时输出由关键活动构成的完整关键路径。

# 三．实现

## 3.1输入图的信息

### 3.1.1流程图



### 3.1.2代码实现

void input(graphmtx& G)

{

int vertex\_num;// 插入顶点

cin >> vertex\_num;

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

{

G.insert\_vertex(i + 49);

}

int edge\_num;// 插入边

cin >> edge\_num;

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

{

edge x;

cin >> x.head;

cin >> x.tail;

cin >> x.weight;

x.pri = i;

G.insert\_edge(x);

}

}

## 3.2求关键路径

### 3.2.1背景知识

求关键路径，只需理解顶点（事件）和边（活动）各自的两个特征属性以及求法即可：

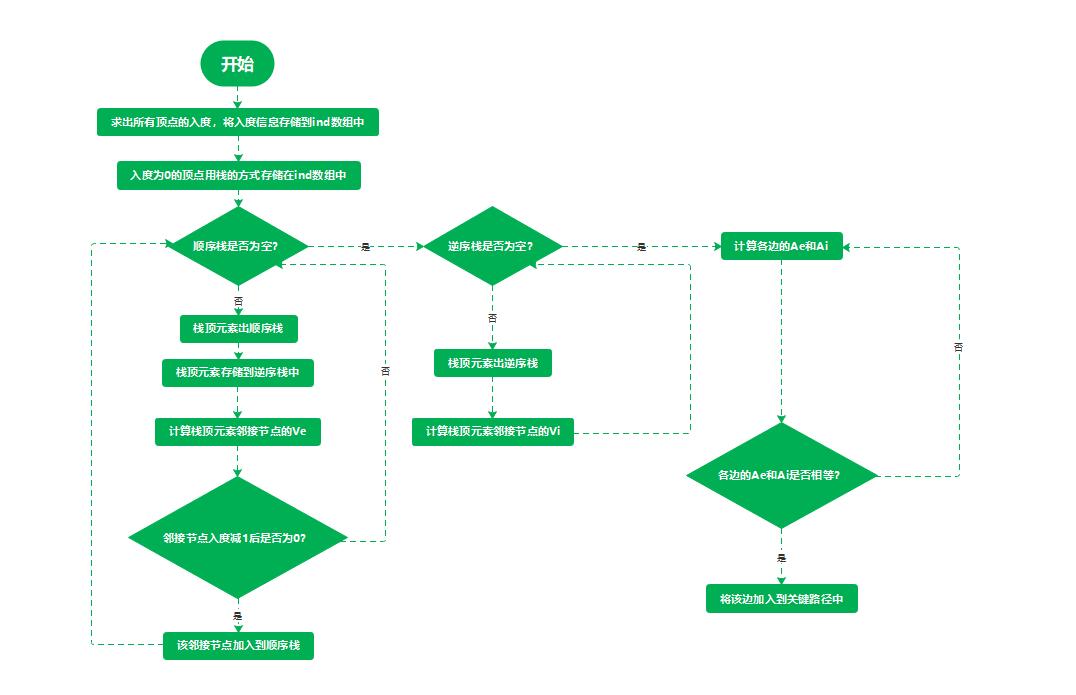
　1.先根据首结点的Ve(j)=0由前向后计算各顶点的最早发生时间

　2.再根据终结点的Vl(j)等于它的Ve(j)由后向前依次求解各顶点的最晚发生时间

　3.根据边的e(i)等于它的发出顶点的Ve(j)计算各边的最早开始时间（最早开始，对应最早发生）

　4.根据边的l(i)等于它的到达顶点的Vl(j)减去边的权值计算各边的最晚开始时间（最晚开始，对应最晚发生）

### 3.2.2流程图



### 3.2.3代码实现

void CriticalPath(graphmtx& G, edge\* cp,int& num)

{

edge ed;

int top = -1;

int lnk = -1;

int j=0;

int u=0;

int v=0;

int vertex\_num = G.get\_vertex\_num();

int edge\_num = G.get\_edge\_num();

int\* ind=new int [defaultsize];//建立入度数组

for (int i = 0; i < vertex\_num; i++)//所有顶点的入度初始化为0

{

ind[i] = 0;

}

for (int i = 0; i < vertex\_num; i++)//统计各顶点的入度

{

j = G.getfirstneighbor(i);

while (j != -1)

{

ind[j]++;

j = G.getnextneighbor(i, j);

}

}

for (int i = 0; i < vertex\_num; i++)//初始的入度为0的元素通过栈来存储

{

if (ind[i] == 0)

{

ind[i] = top;

top = i;

}

}

int\* Ve=new int[defaultsize];//各事件最早和最晚开始时间

int\* Vi=new int[defaultsize];

int\* Ae=new int[defaultsize];//各活动最早和最晚开始时间

int\* Ai=new int[defaultsize];

for (int i = 0; i < vertex\_num; i++)//事件最早开始时间初始化

{

Ve[i] = 0;

}

for (int i = 0; i < vertex\_num; i++)//事件最晚开始时间初始化

{

Vi[i] = 0;

}

while (top != -1)

{

u = top; top = ind[top];//退栈顶点存u

ind[u] = lnk;//反向拉链

lnk = u;

j = G.getfirstneighbor(u);

while (j != -1)

{

if ((G.getweight(u, j) + Ve[u]) > Ve[j])//Ve[j]取最大的路径

{

Ve[j] = G.getweight(u, j) + Ve[u];

}

if (--ind[j] == 0)//当前结点的入度减1，若入度为0，加入栈中

{

ind[j] = top;

top = j;

}

j = G.getnextneighbor(u, j);

}

}

for (int i = 0; i < vertex\_num; i++)//事件最晚开始时间初始化

{

if(lnk>0)

Vi[i] = Ve[lnk];

}

while (lnk != -1 && ind[lnk]>=0)

{

v= ind[lnk];//lnk的前一个节点

lnk = v;

if (lnk == -1)

{

break;

}

j = G.getfirstneighbor(v);

while (j != -1)

{

if (Vi[v] > Vi[j] - G.getweight(v, j))

{

Vi[v] = Vi[j] - G.getweight(v, j);

}

j = G.getnextneighbor(v, j);

}

}

int k = 0;

for (int i = 0; i < vertex\_num; i++)//遍历所有顶点

{

j = G.getfirstneighbor(i);// 遍历每个顶点为起点的边

while (j != -1)

{

int weight = G.getweight(i, j);

Ae[k] = Ve[i];//求出Ae[k]和Ai[k]

Ai[k] = Vi[j] - weight;

if (Ae[k] == Ai[k])

{

ed.head = i + 49;

ed.tail = j + 49;

ed.weight = weight;

ed.pri = G.getpri(i, j);

cp[num++] = ed;

}

k++;

j = G.getnextneighbor(i, j);

}

}

delete []ind;

delete[]Ve;

delete[]Vi;

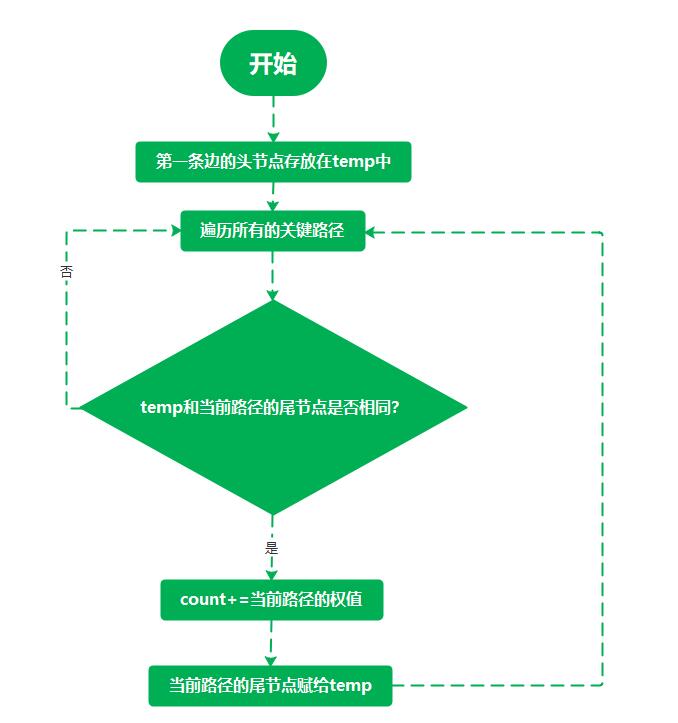
delete[]Ae;

delete[]Ai;

}

## 3.3输出完成活动所需的总时长

### 3.3.1流程图



### 3.3.2代码实现

char temp = cp[0].head;

for (int i = 0; i < num; i++)//计算整个项目所需的时间

{

if (cp[i].head == temp)

{

count += cp[i].weight;

temp = cp[i].tail;

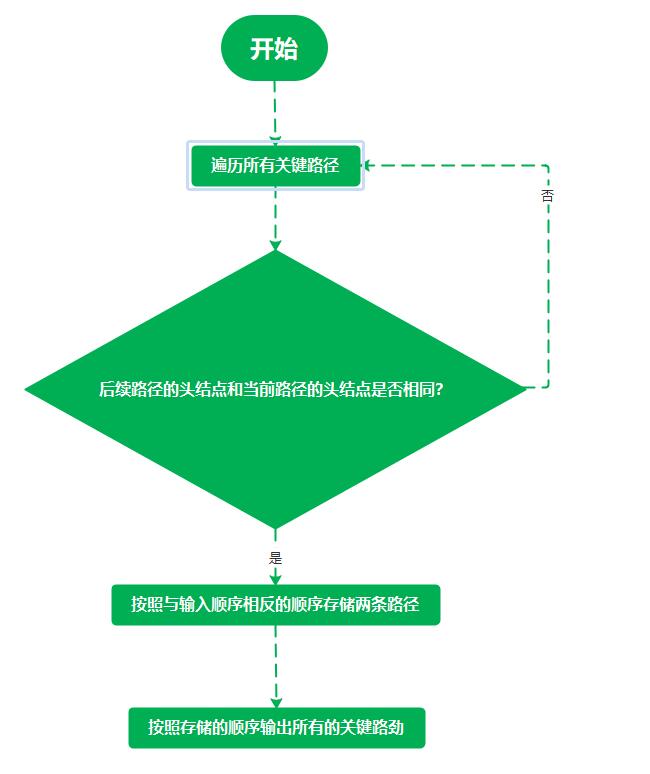
}

}

cout << count << endl;

## 3.4输出关键路径

### 3.4.1流程图



### 3.4.2代码实现

edge x;

for (int i = 0; i < num; i++)//输出关键路径

{

for (int j = i + 1; j < num - 1; j++)

{

if (cp[i].head == cp[j].head)//将输入的边的顺序颠倒

{

if (cp[i].pri<cp[j].pri)

{

x = cp[j];

cp[j] = cp[i];

cp[i] = x;

}

}

else

{

break;

}

}

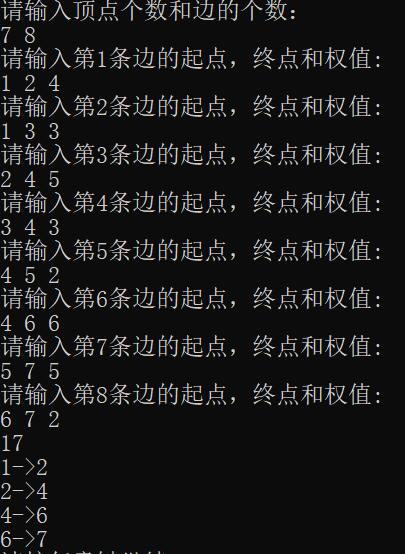
cout << cp[i].head << "->" << cp[i].tail << endl;

}

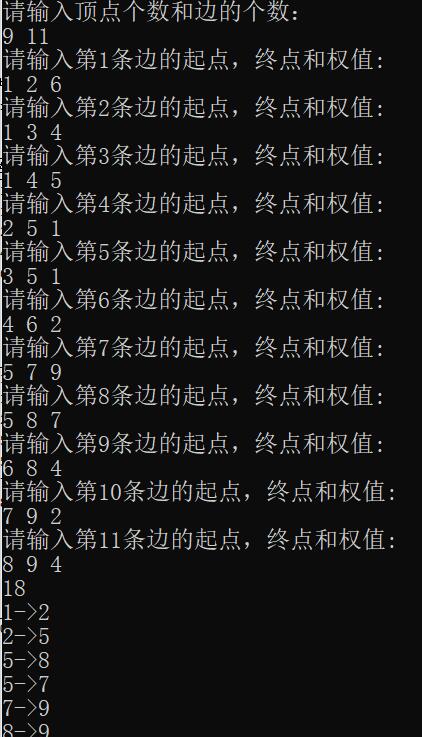
# 测试

## 4.1功能测试

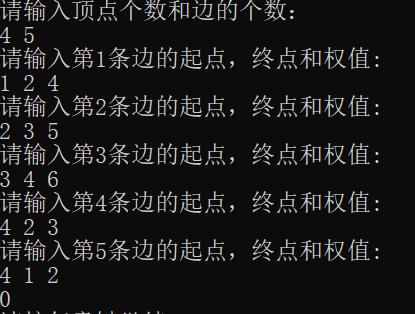
### 4.1.1简单情况测试



### **4.1.2一般情况测试（单个起点和单个终点）**

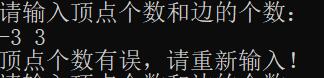


### **4.1.3不可行的方案测试**

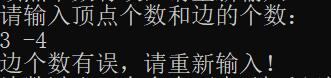


## 4.2错误测试

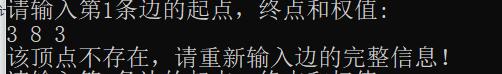
### 4.2.1顶点个数有误：



### **4.2.2边个数有误：**



### **4.2.3顶点不存在：**



### **4.2.4权值有误**：

