数据结构实验报告

实验成绩：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 姓名 |  | 学号 |  | 专业班级 |  |
| 指导教师 | 张先宜 | 实验时间 | 6月10日 | 实验地点 | 计算机楼 |

**实验名称： 图实验二**

6.1 实验目的

(1) 掌握图的基本概念。

(2) 掌握图的存储结构的设计与实现，基本运算的实现。

(3) 掌握图的Dijkstra和Floyd两种最短路径算法。

(4) 掌握有向无环图的的拓扑排序和关键路径求解算法。

6.2 实验要求

1. 结构定义和算法实现放入库文件，如“graph.h”中；
2. 图的测试数据用文本文件方式给出，例如测试数据名为dn.grp的有向网，可参考发来的图形状和参考存储文件；
3. 图创建方法可自行选择；
4. 可多次连续测试。

6.3 实验任务

分别设计图（网）的邻接矩阵、邻接表存储结构，编写算法实现下列问题的求解。

1．实现Dijkstra算法，求解下列给定图G指定顶点到其余顶点之间的最短路径。 实验测试数据基本要求：第一组数据： udn6.grp

第二组数据： un8.grp

第三组数据： dn8.grp

第四组数据： dn10.grp

2．实现Floyd算法，求解下列给定图G各顶点之间的最短路径。 实验测试数据基本要求：第一组数据： udn6.grp

第二组数据： un8.grp

第三组数据： dn8.grp

第四组数据： dn10.grp

3．设计算法求解下列给定图G的拓扑序列。 实验测试数据基本要求：

第一组数据： top6dg1.grp

第二组数据： top7dg1.grp

第三组数据： dn8.grp

第四组数据： dn10.grp

4．设计算法求解下列给定AOE网的关键路径。 实验测试数据基本要求：

第一组数据： dag11.grp

第二组数据： dag12.grp

**6.4数据结构设计**

typedef struct eNode //边链表结点结构

{

int adjVer; //邻接顶点地址，此处为顶点在顶点表中序号，从1开始

eInfoType eInfo; //边链表中表示边的相关信息，比如表的权值

struct eNode\* next; //指向边链表中的下一个结点

}EdgeNode; //边链表结点类型

typedef struct vNode //顶点表中元素结构

{

elementType data; //存放图中顶点的数据

EdgeNode\* firstEdge; //指向此顶点关联的第一条边的指针，即边链表的头指针

}VerNode;

typedef struct GraphAdjLinkList

{

VerNode VerList[MaxVerNum+1]; //存放顶点的顺序表，数组0单元不用

int VerNum; //顶点数

int ArcNum; //弧（边）数

GraphKind gKind; //图的类型:0-无向图；1-无向网；2-有向图；3-有向网

}Graph; //图的类型名

typedef struct GraphAdjMatrix

{

elementType Data[MaxVerNum+1]; //顶点数组，存放顶点元素的值，Data[0]单元不用

cellType AdjMatrix[MaxVerNum+1][MaxVerNum+1]; //邻接矩阵，数组下标为0单元不用，从AdjMatrix[1][1]单元开始

int VerNum; //顶点数

int ArcNum; //弧（边）数

GraphKind gKind; //图的类型:0-无向图；1-无向网；2-有向图；3-有向网

} Graph1; //图的类型名

//深度优先遍历图G，并构造出相应的生成树或生成森林

typedef struct csNode

{

elementType data;

struct csNode \*firstChild, \*nextSibling;

}csNode,\*csTree;

//Prim算法基于邻接链表的实现

int inTree[MaxVerNum+1]={0}; //标记顶点已经在Prim生成树中，或已经访问过。1或true--已访问，0或false--未访问

//inTree[0]单元未用

//或者为标记已经在集合U中的顶点

//保存候选边的信息

typedef struct minEdgeType

{

int v; //V-U中当前选中的顶点编号，从1开始。即刚从V-U中选出放到U中的顶点

cellType eWeight; //U中某个顶点到V-U中当前顶点v的最小距离

} MinEdgeType;

//Kruskal 算法--基于邻接链表

typedef struct edgetype

{

int vBegin; //边的起始顶点编号，从1开始

int vEnd; //边的另一顶点编号，从1开始

eInfoType eWeight; //边的权值

}EdgeType;

**6.5算法设计**

1.实现Dijkstra算法，求解下列给定图G指定顶点到其余顶点之间的最短路径。

【算法思想】

（1）求解结果的表示和存储：如前所述，引入两个一维数组 path[]和 dist[]。path[]数组

存储最短路径信息，其下标对应网中的顶点编号，元素值表示下标对应顶点的直接前驱顶点，

（2）标记各顶点是否已经求解：可引入一个一维标记数组 solved[]，数组下标对应顶点

编号，数组元素值为 1，表示已求出最短路径，0 表示尚未求解，

（3）搜索下一个求解顶点 v：就是在未解顶点中搜索出 dist 值最小的一个顶点。

（4）修改所搜索出的顶点 v 的直接后继顶点的最短路径及其长度。

//Dijkstra算法--基于邻接表

void Dijkstra(Graph &G, int path[], int dist[], int vID)

{

int solved[MaxVerNum]; //标记顶点是否已经求出最短路径（已在集合S中）。1-已求出；0-未求出。

int i,j;

int v; //顶点编号

eInfoType minDist; //保存最短距离值

EdgeNode \*p; //指向边链表结点

//初始化集合S，距离数组dist[]，路径数组path[]

for(i=1;i<=G.VerNum;i++)

{

solved[i]=0; //所有顶点均为处理

dist[i]=INF; //所有顶点初始距离置为无穷大（INF）

path[i]=-1; //所有顶点的前驱置为-1，即无前驱

}

//处理顶点vID

solved[vID]=1; //标记vID已经处理

dist[vID]=0;

path[vID]=-1;

//从邻接表初始化dist[]和path[]

p=G.VerList[vID].firstEdge; //顶点vID的边链表指针

while(p)

{

v=p->adjVer; //取得vID邻接顶点编号

dist[v]=p->eInfo; //取得vID与v之间边的权值，赋给dist[v]

path[v]=vID; //顶点v的前驱为vID

p=p->next;

}

//依次找出余下n-1个顶点加入集合S中

for(i=1;i<G.VerNum;i++)

{

minDist=INF;

//寻找集合V-S中距离vID最近的顶点

for(j=1;j<=G.VerNum;j++)

{

if(solved[j]==0 && dist[j]<minDist)

{

v=j; //j为V-S中候选的距离vID最近的顶点

minDist=dist[j];

}

}

if(minDist==INF) //S与V-S没有相邻的顶点，算法退出

return;

cout<<"选择顶点："<<G.VerList[v].data<<"--距离："<<minDist<<endl; //输出本次选择的顶点距离

solved[v]=1; //标记顶点v以找到最短距离，加入集合S中

//对选中的顶点v，更新集合V-S中所有与v邻接的顶点距离vID的距离

p=G.VerList[v].firstEdge; //取得v的边链表指针

while(p)

{

j=p->adjVer; //取得v的邻接顶点编号

if(solved[j]==0 && minDist+p->eInfo<dist[j])

{

dist[j]=minDist+p->eInfo; //更新顶点j的最小距离

path[j]=v; //j的前驱改为顶点v

}

p=p->next;

}

}

}

2. 实现Floyd算法，求解下列给定图G各顶点之间的最短路径

【算法思想】

定义了两个二维矩阵：矩阵dist记录顶点间的最小路径,矩阵path记录顶点间最小路径中的中转点

它通过3重循环，m为中转点，i为起点，j为终点，循环比较dist [i][j] 和 dist [i][m] + dist [m][j] ，如果dist [i][j] 和 dist [i][m] + dist [m][j]为更小值，则把dist [i][j] 和 dist [i][m] + dist [m][j]覆盖保存在dist [i][j]中。

void Floyd(Graph1 &G, cellType dist[MaxVerNum][MaxVerNum], int path[MaxVerNum][MaxVerNum])

{

int i,j,k;

//初始化距离矩阵和路径矩阵

for(i=1;i<=G.VerNum;i++)

{

for(j=1;j<=G.VerNum;j++)

{

dist[i][j]=G.AdjMatrix[i][j]; //距离矩阵初始化为邻接矩阵

//初始化路径矩阵，路径矩阵元素path[i][j]中保存编号j顶点的前驱的顶点编号

if( i!=j && G.AdjMatrix[i][j]<INF) //如果i,j之间存在边，则j的前驱为i。否则前驱置为-1

path[i][j]=i;

else

path[i][j]=-1;

}

}

//从k=1开始，迭代到k=G.verNum。依次选择一个顶点k，作为顶点i、j之间的中转顶点，优化顶点i、j之间的距离

//下面是Floyd算法的核心--三重for循环

for(k=1; k<=G.VerNum; k++)

{

for(i=1; i<=G.VerNum; i++)

{

for(j=1; j<=G.VerNum;j++)

{

if(i!=j && dist[i][k]+dist[k][j]<dist[i][j]) //k作为中转跳点，i、j之间距离变小，接收k作为中转点，更新i、j之间的距离

{

dist[i][j]=dist[i][k]+dist[k][j]; //更新距离

path[i][j]=path[k][j]; //修改前驱顶点

}

}

}

}

}

3.设计算法求解下列给定图G的拓扑序列。

【算法思想】

（1）采用一个入度数组 ind保存各顶点的入度。假设 ind 中的各元素的值已经设置好了。

（2）将入度为 0 并且未输出的顶点放在一个结构中，需要时就直接从中取出，而不

必搜索，从而节省搜索时间。

（3） 步骤 ② 中“删除”顶点的实现：

① 初始化空栈 S。

② 将 AOV 网中所有入度为 0 的顶点压入栈 S 中。

③ 若 S 不空，则 V=POP(S)，并输出 V。

④ 将 V 的每个后继的入度减 1，若其中某个后继的入度变成了 0，则将其压入栈 S 中。

转③。

//拓扑排序算法--基于邻接表

//初始化获取每个顶点的入度，存入入度数组inds[]中

void GetInDegrees(Graph &G, int inds[])

{

EdgeNode \*p; //边链表指针

int i;

int k;

for(i=1;i<=G.VerNum;i++)

{

p=G.VerList[i].firstEdge;

while(p)

{

k=p->adjVer;

inds[k]++; //编号为k的顶点入度加1

p=p->next;

}

}

}

//拓扑排序算法--使用栈和队列，使用一个标记数组solved[]

int TopologicalSort(Graph &G, int topoList[])

{

int inds[MaxVerNum]; //存放顶点入度

int solvedlen = 0;

int solved[MaxVerNum]; //标记入度为0的顶点是否已经处理。0--未处理；1--已处理。

int i;

int v=-1; //顶点编号

int vCount=0; //记录入度为0的定点数

EdgeNode \*p; //指向边链表结点的指针

//初始化

for(i=1;i<=G.VerNum;i++)

{

inds[i]=0; //所有顶点初始入度置为0

topoList[i]=-1; //拓扑序列初始化为-1

}

//从邻接表获取各顶点初始入度

GetInDegrees(G,inds);

//取得第一个入度为0的顶点（如果存在），保存到v

for(i=1;i<=G.VerNum;i++)

{

if(inds[i]==0)

{

solved[solvedlen++] = i;

}

}

while(solvedlen > 0)

{

v = solved[--solvedlen];

topoList[vCount+1]=v;

vCount++;

//以顶点v相邻的顶点入度减1

p=G.VerList[v].firstEdge;

while(p)

{

v=p->adjVer;

inds[v]--; //邻接点入度减1

if(inds[v] == 0){

solved[solvedlen++] = v;

}

p=p->next;

}

}

if(vCount==G.VerNum)

return 1; //拓扑排序成功

else

return 0; //存在回路，拓扑排序失败

}

void PrintKeyPath(Graph1& G, int topoList[], int vet[MaxVerNum], int vlt[MaxVerNum]) {

int v, w;

cout << "其中一条关键路径为：\t";

v = topoList[1];

cout << G.Data[v] << "\t";

while (v != -1) {

w = firstAdj(G, v);

while (w != -1) {

if (vet[w] == vlt[w]) {

cout << G.Data[w] << "\t";

break;

}

else {

w = nextAdj(G, v, w);

}

}

v = w;

}

}

4.设计算法求解下列给定AOE网的关键路径。

【算法思想】

（1）设每个结点的最早发生时间为0，将入度为0的结点进栈

（2）将栈中入度为0的结点V取出，并压入另一个栈，用于形成逆向拓扑排序的序列

（3）根据邻接表找到结点V的所有的邻接结点，将结点V的最早发生时间+活动的权值 得到的和同邻接结点的原最早发生时间进行比较；如果该值大，则用该值取代原最早发生时间。另外，将这些邻接结点的入度减一。如果某一结点的入度变为0，则进栈。

（4）反复执行2、3；直至栈空为止。

void KeyPath(Graph1& G, int topoList[]) {

int i, j;

int vPre;//保存顶点的前驱顶点编号

int vSuc;//保存顶点的后继顶点编号

int vet[MaxVerNum + 1];

int vlt[MaxVerNum + 1];

for (i = 1; i <= G.VerNum; i++) {//初始化最早发生时间为0

vet[i] = 0;

}

for (i = 1; i <= G.VerNum; i++) {

vPre = topoList[i];

for (j = 1; j <= G.VerNum; j++) {

if (G.AdjMatrix[vPre][j] >= 1 && G.AdjMatrix[vPre][j] < INF) {

if (vet[j] < vet[vPre] + G.AdjMatrix[vPre][j]) {

vet[j] = vet[vPre] + G.AdjMatrix[vPre][j];

}

}

}

}

for (i = 1; i <= G.VerNum; i++) {//初始化vlt值为vet[G.VerNum]

vlt[i] = vet[G.VerNum];

}

for (i = G.VerNum; i >= 1; i--) {//按照逆拓扑次序求解最迟发生时间

vSuc = topoList[i];

for (j = G.VerNum; j >= 1; j--) {

if(G.AdjMatrix[j][vSuc]>=1&&G.AdjMatrix[j][vSuc]<INF){

if (vlt[j] > vlt[vSuc] - G.AdjMatrix[j][vSuc]) {

vlt[j] = vlt[vSuc] - G.AdjMatrix[j][vSuc];

}

}

}

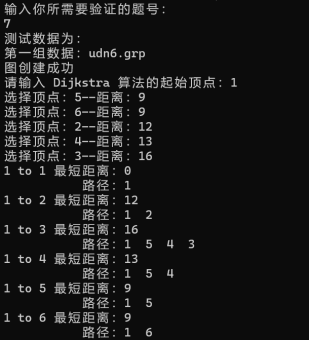
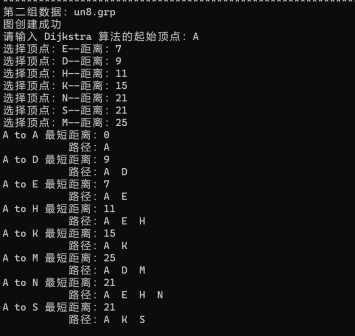
}

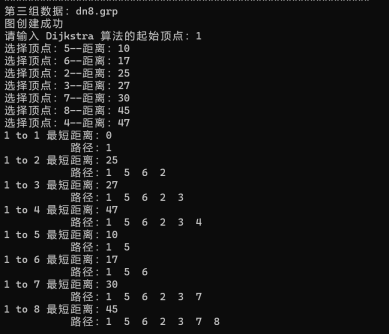
PrintKeyPath(G, topoList, vet, vlt);

}

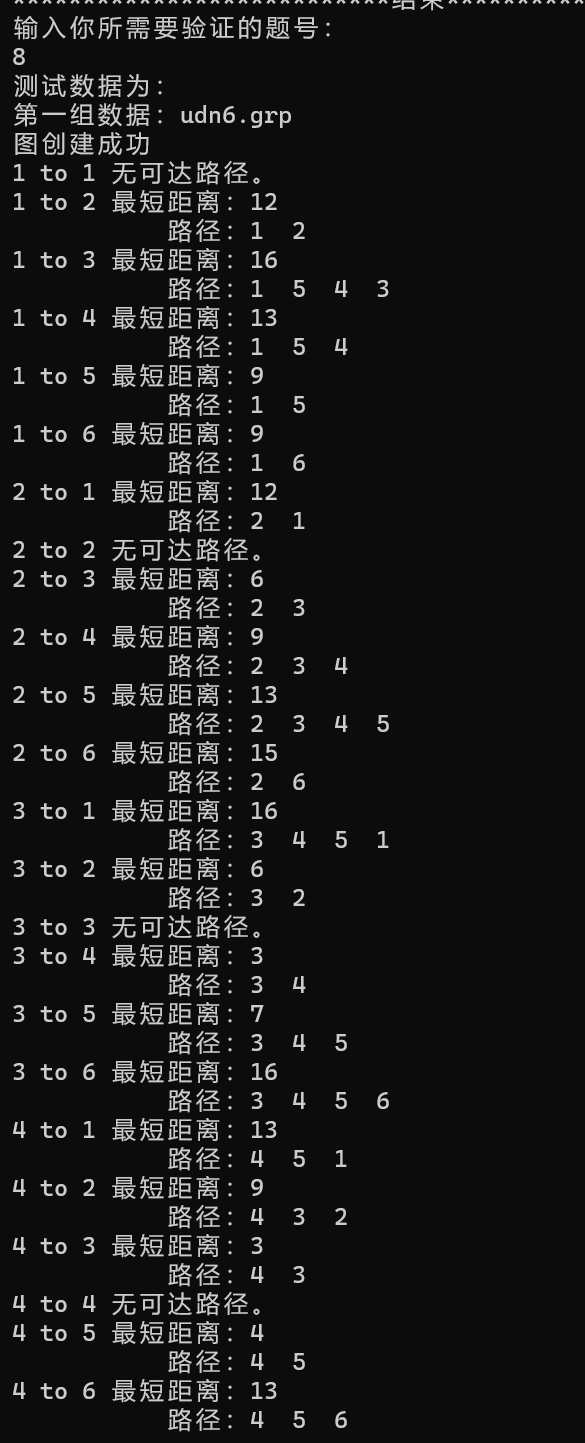
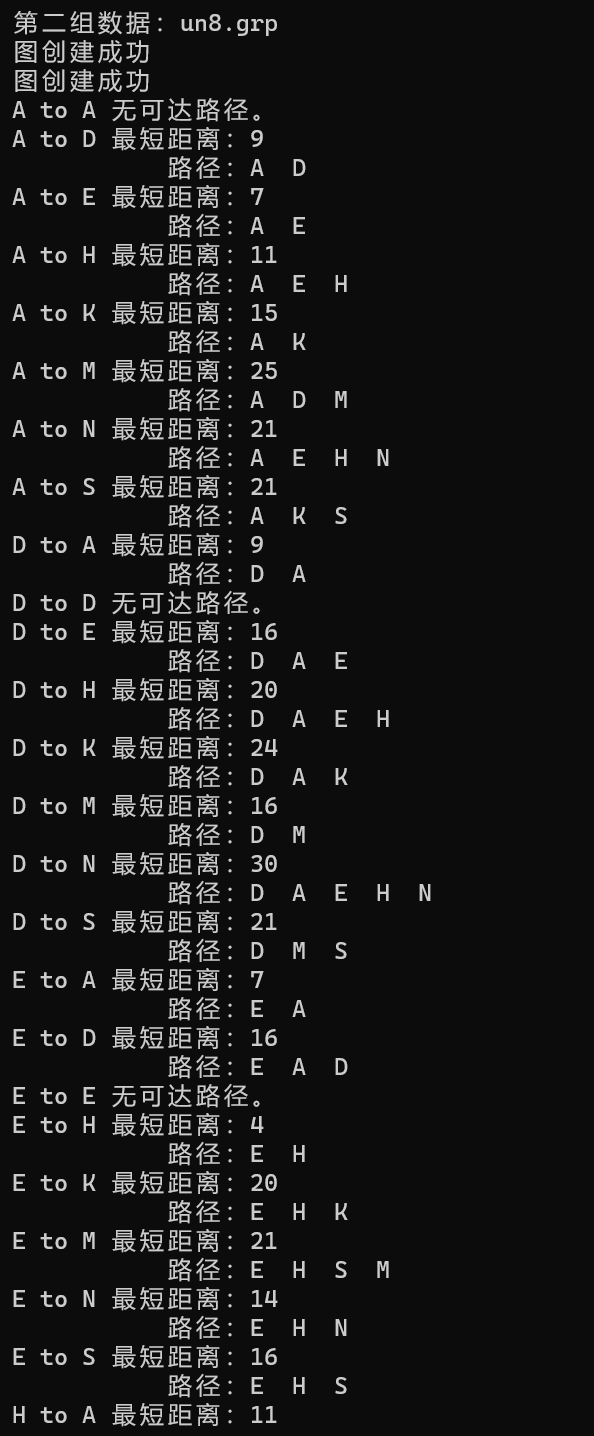
**6.6运行和测试**

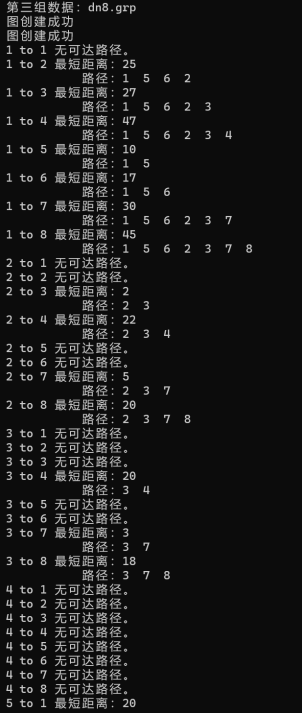
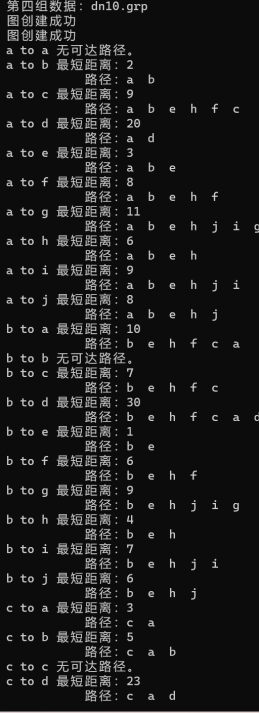
1）实现Dijkstra算法，求解下列给定图G指定顶点到其余顶点之间的最短路径

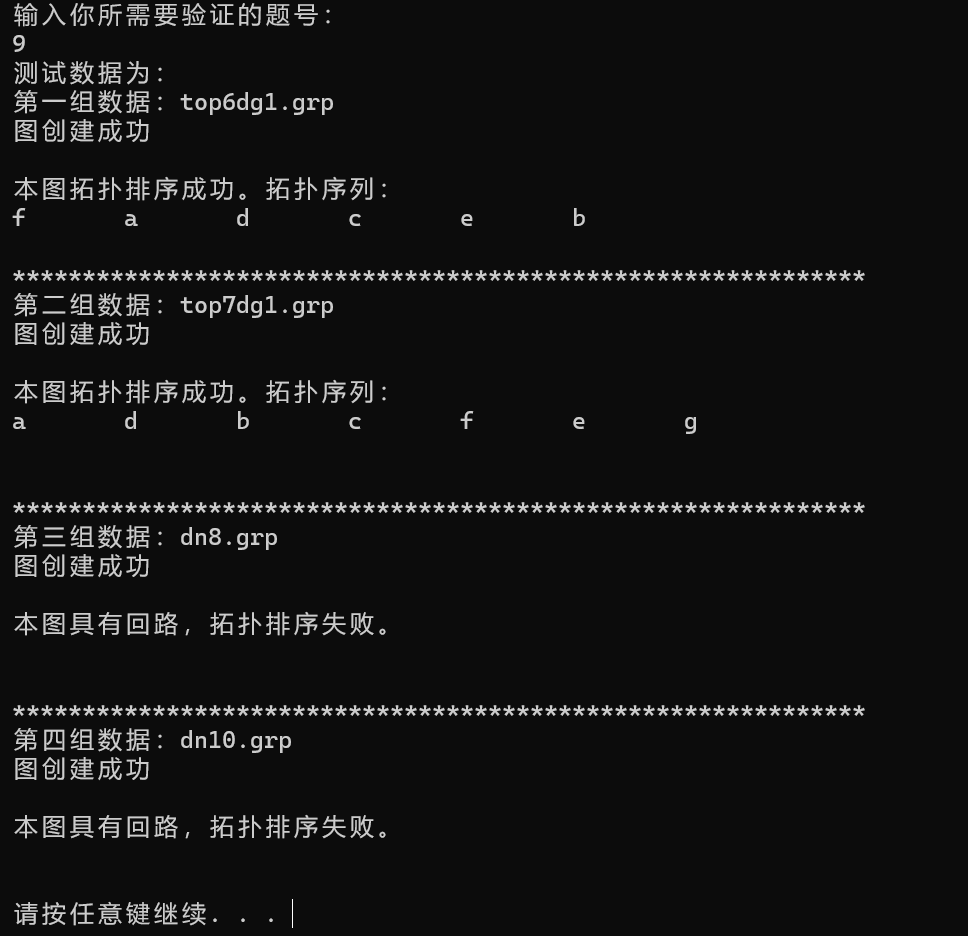
** **

2）实现Floyd算法，求解下列给定图G各顶点之间的最短路径。

3）设计算法求解下列给定图G的拓扑序列。



4）设计算法求解下列给定AOE网的关键路径。



**6.7总结、心得和建议**

图结构是目前数据结构实验中最复杂的实验，所以，图的实验较其它实验来说也比较复杂，但此次图实验基本为书上已有的算法。此次实验主要是对图的基本性质及结构加深理解，对书上的相关图的算法理解并运用。

经过此次实验，我学到了了两种的图常用的存储结构，此次实验主要运用邻接矩阵存储结构。更加体会了图的基本遍历的重要性以及图的广泛运用。关于不同的图问题，有不同不求解算法，书中只是给出了最常用的几种算法，既是基础的算法，在这次试验中，通过对书本的理解以及对老师给出的程序的学习，加深了我对图的理解。