概念

图定义为：Graph=(V, R)
其中：

$$V = \{x | x \in \text{dataobject}\}$$
$$R = \{VR\}$$
$$VR = \{< x, y > | p(x, y) \wedge (x, y \in V)\}$$

ADT略

设图中顶点数为n，边或弧数为e，则：

- 对于无向图，e的取值范围：$0 \sim \frac{n(n-1)}{2}(C_n^2)$
- 对于有向图，e的取值范围：$0 \sim n(n-1)$（$2 \times C_n^2$）
  无向完全图：具有$\frac{n(n-1)}{2}$条边的无向图
  有向完全图：具有$n(n-1)$条边的有向图
  稀疏图：有很少边或弧的图($e < n \log n$)
  稠密图：有较多边或弧的图
  权：和图中边相关的数
  网：带权的图

邻接：有边/弧相连的两个顶点之间的关系
存在(v_i, v_j)，则称v_i和v_j互为邻接点（无向图）
存在$< v_i, v_j >$，则称v_i邻接到v_j，v_j邻接自v_i（有向图）
关联（依附）：边/弧与顶点之间的关系
存在(v_i, v_j)或$< v_i, v_j >$，则称该边/弧关联于v_i和v_j
顶点的度：与该顶点相关联的边的数目，记为$TD(v)$
顶点v的入度（有向图）是以v为头的弧的数目，记作$ID(v)$
顶点v的出度（无向图）是以v为尾的数目，记作$OD(v)$
顶点v的度$TD = ID(v) + OD(v)$

路径：路径是顶点的序列$V = \{v_{i0}, v_{i1}, \cdots, v_{in}\}$，满足$(v_{ij-1}, v_{ij}) \in E$或
$< v_{ij-1}, v_{ij} > \in E(i < j \leq n)$
路径长度：沿路径边的数目或沿路径各边权值之和
回路：第一个顶点和最后一个顶点相同的路径
简单路径：序列中顶点不重复出现的路径
简单回路：除了第一个顶点和最后一个顶点外，其余顶点不重复出现的回路

连通：从顶点V到顶点W有一条路径，则说V和W是连通的
连通图（无向图）：无向图中任意两个顶点都是连通的，有向图中要看方向

强连通图（有向图）：有向图中，如果每一对$v_i, v_j \in V$，$v_i \neq v_j$，从v_i到v_j和从v_j到v_i都存在路径，则称G是强连通图

子图：设有两个图$G = (V, \{E\})$，和$G' = (V_1, \{E_1\})$，若$V_1 \subseteq V$且$E_1 \subseteq E$，则称G'是G的子图

连通分量（无向图）：无向图中的极大连通子图。极大连通子图的意思是该图是G的连通子图，将G的任何不在该子图的顶点加入，子图不再连通
强连通分量：有向图中极大强连通子图。极大强连通子图的意思是，该子图是G的连通子图，将G的任何不在该子图中的顶点加入，子图不再是强连通的

生成树（无向图）：包含无向图G的所有顶点的极小连通子图。极小连通子图的意思是G的连通子图，在该子图中删除任何一条边，该子图不连通
生成森林：对非连通图，由各个连通分量的生成树的集合

图的存储结构

邻接矩阵表示（掌握）

设$G = (V, E)$是有n 1个顶点的图，G的邻接矩阵是具有以下性质的n阶方阵

$$ij = \begin{cases} 1 & (v_i, v_j) \in E < v_i, v_j > \in E \\ 0 & Oeie \end{cases}$$

- 无向图的邻接矩阵对称，可以压缩存储，空间为$\frac{n(n+1)}{2}$，顶点v_i的度为邻接矩阵中第i行元素之和。特别地，无向完全图的邻接矩阵中，对角元素为0，其余是1
- 有向图邻接矩阵不一定对称，有n个地点的有向图需存储空间为n^2，顶点v_i的出度是邻接矩阵中第i行元素之和，入度是邻接矩阵中第i列元素之和，度是入度+出度

网的邻接矩阵可以定义为

$$ij = \begin{cases} {}_{ij} & (v_i, v_j) \in E < v_i, v_j > \in E \\ & Oeie \end{cases}$$

```C++
#define INFINITY INT_MAX
#define MAX_VERTEX_NUM 20
typedef enm { DG, DN, UDG, UDN } GraphKind;

typedef struct ArcCell {
   VRType adj;
   InfoType* info;
} ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

typedef struct {
   VertexType vexs[MAX_VERTEX_NUM]; // 顶点表
   AdjMatrix arcs; // 邻接矩阵
   int vexnum, arcnum; // 图的当前顶点数和弧数
   GraphKind kind;
} MGraph;
```

构造图G

```C++
Status CreateGraph(MGraph &G) {
   scanf(&G.kind);
   switch(G.kind) {
       case DG: return CreateDG(G); // 有向图
       case DN: return CreateDN(G); // 有向网
       case UDG: return CreateUDG(G); // 无向图
       case UDN: return CreateUDN(G); // 无向网
       default: return ERROR;
   }
}
```

使用数组构造无向网G

```C++
Status CreateUDN(MGraph &G) {
   scanf(&G.vexnum, &G.arcnum, &IncInfo);
   for(i = 0;i < G.vexnum;++i) scanf(&G.vexs[i]);
   for(i = 0; i< G.vexnum;++i)
       for(j = 0; j < G.vexnum;++j)
           G.arcs[i][j] = {INFINITY, NULL}; // 初始化为无穷，不需要额外信息
   for(k = 0; k < G.arcnum;++k) {
       // 读边的两个端点和权重
       scanf(&v1, &v2, &w);
       i = LocateVex(G, v1); j = LocateVex(G, v2);
       G.arcs[i][j].adj = w;
       if(IncInfo) Input(*G.arcs[i][j].info);
       G.arcs[j][i] = G.arcs[i][j];
   }
   return OK;
}
```

使用邻接矩阵的优点是：

1. 直观、简单、好理解
2. 方便检查任意一对顶点间是否存在边
3. 方便找任一顶点的所有“邻接点”
4. 方便计算任一顶点的度
   邻接矩阵的缺点：
5. 不便于增加和删除顶点
6. 浪费空间——存稀疏图有大量无效元素
7. 浪费时间——统计稀疏图中一共有多少条边

邻接表表示法（掌握）

顶点：按编号顺序将顶点数据存储在一维数组中
关联同一顶点的边（以顶点为尾的弧）：用线性链表存储

邻接表中是头结点，链表包含邻接点域和链域

无向图的邻接表表示

特点：

- 邻接表不唯一
- 若无向图中有n个顶点，e条边，则需n个头结点和2e个表结点
- 无向图中顶点v_i的度为第i个单链表中的结点数

有向图的邻接表和逆邻接表表示

特点：

- 邻接表也不唯一
- 若有向图中有n个顶点，e条边，则需n个头结点和e个表结点
- 顶点v_i的出度为第i个单链表中的结点个数
- 顶点v_i的入度为整个单链表中邻接点域值是$i-1$的结点个数（数组索引从0开始）

逆邻接表特点：

- 顶点v_i的入度为第i个单链表中的结点个数
- 顶点v_i的出度为整个单链表中邻接点域值是$i-1$的结点个数(数组索引从0开始)

```C++
#define MAX_VERTEX_NUM 20

typedef struct ArcNode {
   int adjvex;
   struct ArcNode* nextarc;
   InfoType* info;
} ArcNode;

typedef struct VNode {
   VertexType data;
   ArcNode* firstarc;
} VNode, AdjList[MAX_VERTEX_NUM];

typedef struct {
   AdjList Vertices;
   int vexnum, arcnum;
   int kind;
} ALGraph;
```

邻接表的特点：

- 方便找任一顶点的所有“邻接点”
- 节约稀疏图的空间
  - 无向图需要n个头指针+2e个结点
  - 有向图需要n个头指针+e个结点
- 方便计算任一顶点的度?
  - 对无向图：是的
  - 对有向图：只能计算出度；需构造逆邻接表来计算入度
- 不方便检查任意一对顶点间是否存在边

十字链表和邻接多重表（了解）

针对邻接表有向图求结点的度困难，无向图每条边都要存储两边的问题，对于有向图采用十字链表、对于无向图采用邻接多重表

十字链表

```C++
#define MAX_VERTEX_NUM 20

typedef struct ArcBox {
   int tailvex, headvex;
   struct ArcBox* hlink, tlink;
   InfoType* info;
} ArcBox;

typedef struct VexNode {
   VertexType data;
   ArcBox* firstin, firstout;
} VexNode;

typedef struct {
   VexNode xlist[MAX_VERTEX_NUM];
   int vexnum, arcnum;
} OLGraph;
```

邻接多重表

```C++
#define MAX_VERTEX_NUM 20

typedef enum { unvisited, visited } VisitIf;
typedef struct EBox {
   VisitIf mark;
   int ivex, jvex;
   struct EBox* ilink, jlink;
   InfoType* info;
} EBox;

typedef struct VexBox {
   VertexType data;
   EBox* firstedge;
} VexBox;

typedef struct {
   VexBox adjmulist[MAX_VERTEX_NUM];
   int vexnum, edgenum;
} AMLGraph;
```

图的遍历

图的遍历：从图中某一顶点出发，对图中所有顶点访问一次且仅访问一次
访问：操作的抽象，可以是对结点进行的各种处理，可以以输出结点的数据为例来理解
遍历的实质：找每个顶点的邻接点的过程

DFS

1. 从图中某以顶点v出发，访问此顶点
2. 然后依次从v的未被访问的邻接点出发，深度优先搜索图
3. 重复上述过程，直到所有与v有路径相通的顶点都被访问过
4. 若此时图中尚有顶点未被访问，则另选图中一个未被访问的顶点作起点，重复上述过程，直到所有顶点都被访问过

算法：

```C++
bool visited[MAX];
Status (*VisitFunc)(int v);

void DFSTraverse(Graph G, Status (*Visit)(int v)) {
   VisitFunc = Visit;
   for(v = 0; v < G.vexnum; ++v) visited[v] = false;
   for(v = 0; v < G.vexnum; ++v)
       if(!visited[v]) DFS(G,v);
}

 void DFS(Graph G, int v) {
    visited[v] = true;
    VisitFunc(v);
    for(w = FirstAdjVex(G, v);w;w = NextAdjVex(G, v, w))
    if(!visited[w]) DFS(G, w);
 }
```

邻接表：复杂度$O(n+e)$

```C++
void DFS(ALGraph G, int v) {
   visited[v] = TRUE;
   VisitFunc(v);
   p = G.vertices[v].firstarc;
   while(p) {
       if(!visited[p->adjvex]) DFS(G, p->adjvex);
       p = p->nextarc;
   }
}
```

邻接矩阵：复杂度$O(n^2)$

```C++
void DFS(MGraph G, int v) {
    visited[v] = TRUE;
    VisitFunc(v);
    for(j = 0; j < G.vexnum;++j) {
        if(G.arcs[v][j].adj == 1 && !visited[j])
            DFS(G, j);
    }
}
```

BFS

```C++
void BFSTraverse(Graph G, Status(*visit)(int v)) {
    for(v = 0; v < G.vexnum; ++v)
        visited[v] = FALSE;
    InitQueue(Q);
    for(v = 0; v < G.vexnum; ++v)
        if(!visited[v]) {
            visited[v] = TRUE;
            visit(v);
            EnQueue(Q, v);
            while(!QueueEmpty(Q)) {
                DeQueue(Q, u);
                for(w = FirstAdjVex(G, u);w;w = NextAdjVex(G, u, w)) {
                    if(!visited[w]) {
                        visited[w] = TRUE:
                        visit(w);
                        EnQueue(Q, w);
                    }
                }
            }
        }
}
```

图的连通性问题

DFS建立非连通图G的生成森林

```C++
void DFSForest(Graph, CSTree &T) {
   T = NULL;
   for(v = 0; v < G.vexnum; ++v) visited[v] = FALSE;
   for(v = 0; v < G.vexnum; ++v)
       if(!visited[v]) {
           p = (CSTree)malloc(sizeof(CSNode))'
           *p = { GetVex(G, v), NULL, NULL };
           if(!T) T=p;
           else q->nextsibling = p;// q是上一个孩子结点
           q = p;
           DFSTre(G, v, p);
       }
}

void DFSTree(Graph G, int v, CSTree &T) {
   visited[v] = TRUE; first = TRUE;
   for(w = FirstAdjVex(g, v);w;w = NextAdjVex(G, u , w))
   if(!visited[w]) {
       p = (CSTree)malloc(sizeof(CSNode));
       *p = { GetVex(G, w), NULL, NULL };
       if(first) { T->lchild = p; first = False; }
       else { q->nextsibling = p; }
       q = p;
       DFSTree(G, w, q);
   }
}
```

图的连通性问题

要在n个城市间建立通信联络网，则n个城市应铺$n-1$条线路，但因为每条线路都会有对应的经济成本，n个城市应铺$n-1$条线路，但因为每条线路都会有对应的经济成本，n个城市最多有$\frac{n(n-1)}{2}$条线路，那么，如果选择$n-1$条线路，使总费用最少？

- 顶点——城市，有n个
- 边——线路
- 边的权值——线路的经济代价
- 连通网——n个城市间通信网

问题转化为：构造联通给你忘的最小代价生成树——最小生成树

MST性质：假设$=\{V,\{E\}\}$是一个连通网，是顶点集V的一个非空子集。若$(,v)$是一条具有最小权值（代价）的边，其中$\in,v\in V-$，则必然存在一棵包含边$(,v)$的最小生成树

普里姆（Prim）算法

思想：设$=(V,\{E\})$是连通网，TE是N上最小生成树中边的集合

- 初始令$=\{_0\}(_0\in V)$，$TE=$
- 在所有$\in,v\in V-$的边$(,v)\in E$中，找一条代价最小的边$(_0,v_0)$
- 将$(_0,v_0)$并入集合TE，同时v_0并入U
- 重复上述操作直至$=V$为止，则$T=(V,\{TE\})$为的最小生成树

```C++
Struct {
   VertexType adjvex;
   VRType lowcost;
} closedge[MAX_VERTEX_NUM];
```

```C++
void MiniSpanTree_PRIM(MGraph G, VertexType u) {
   k = LocateVex(G, u);
   for(j = 0; j < G.vexnum; ++j)
       if(j != k) closedge[j] = { u, G.arcs[k][j].adj };
   for(i = 1; i < G.vexnum; ++i) {
       k = minimum(closedge);
       printf(closedge[k].adjvex, G.vexs[k]);
       closedge[k].lowcost = 0;
       for(j = 0; j < G.vexnum; ++j)
           if(G.arcs[k][j].adj < closedge[j].lowcost)
               closedge[j] = { G.vexs[k], G.arcs[k][j].adj };
   }
}
```

复杂度$O(n^2)$，和边无关，适用与边稠密的图

克鲁斯卡尔（Kruskal）算法

算法思想：

- 设连通网$=(V,\{E\})$，令最小生成树初始状态为只有n个顶点而无边的非连通图$T=(V,\{\})$，每个顶点自成一个连通分量
- 在E中选取代价最小的边，若该边依附的顶点落在T中不同的连通分量上，则将此边加入到T中；否则，舍去此边，选取下一条代价最小的边
- 依此类推，直至T中所有顶点都在同一连通分量上为止

复杂度$O(e\log e)$，与边有关，适用于边稀疏的图

有向无环图及其应用

一个无环的有向图，简称DAG图

拓排

拓扑排序：由某个几何上的一个偏序得到该集合上的一个全序，这个操作称为拓扑排序

AOV网：顶点表示活动，弧表示优先关系的有向图称为顶点表示活动的网（Activity On Vertex Network）

学生选修课程问题

- 顶点：表示课程
- 有向弧：表示先决条件，若课程i是课程j的先决条件，则图中有弧$< i, j >$
- 学生应按怎样的顺序学习这些课程，才能无矛盾、顺利地完成学业?

拓排的方法：

1. 在有向图中选一个没有前驱的顶点且输出
2. 从图中删除该顶点和所有以它为尾的弧
3. 重复上述两步直至全部顶点均已输出；或者当图中不存在无前驱的结点为止

拓排算法：

1. 扫描顶点表，求各顶点入度$ineee0\,ven-1$
2. 初始化栈，入度一个0的顶点入栈，计数count=0
3. 当栈不空时重复
    1. 弹出栈顶顶点，并输出，计数count++
    2. 取该顶点的所有邻接点将其入度减1，若减1后入度为0，则该顶点入栈
4. 当栈空时，若输出的顶点数count小于图的顶点数，则图有回路，否则正常

```C++
Status TopologicalSort(ALGraph G) {
   FindInDegree(G, indegree);
   InitStack(S);
   for(i = 0; i< G.vexnum; ++i)
       if(!indegree[i]) Push(S, i);
   count = 0;
   while(!StackEmpty(S)) {
       Pop(S, i);
       printf(i, G.vertices[i].data); ++count;
       for(p = G.vertices[i].firstarc; p; p=p->nextarc) {
           k = p->adjvex;
           if(!(--indegree[k])) Push(S, k);
       }
   }
   if(count < G.vexnum) return ERROR;
   else return OK;
}
```

关键路径

问题提出：把工程计算表示为有向图，用顶点表示事件，弧表示活动

AOE网：带权的有向无环图，其中顶点表示事件，弧表示活动，权表示活动持续时间，这称为AOE网（Activity On Edg）

- 路径长度——路径上各活动持续时间之和
- 关键路径——路径长度最长的路径
- $ve(j)$——表示事件V_j的最早发生时间
- $v(j)$——表示事件V_j的最迟发生时间
- $e(i)$——表示活动$_i$的最早开始时间
- (i)——表示活动$_i$的最迟开始时间
- $(i)-e(i)$——表示完成活动$_i$的时间余量
- $(i)-e(i)=0$的活动——关键活动

我们关心的是哪些活动是影响工程进度的关键（时间最长的）

要求活动的最早最迟开始时间$e(i),(i)$，首先要求得事件的最早最迟开始时间
求$ve(j)$和$v(j)$要分两步

1. 从$ve(0)=0$开始向前递推：

$$ve(j)=\max_i\{ve(i)+(<i,j>)\}$$
$$<i,j>\in T,j=1,2,\cdots,n-1$$

取max的原因是需要等待其他进入结点j的活动都完成后才能开始

2. 从$v(n-1)=ve(n-1)$起向后递推：

$$v(i)=\min_{<i,j>\in}\{v(i)-(<i,j>)\}, j=1,2,\cdots,n-1$$

取min的原因是为了表示最早可以开始的时间，即最早开始的那个事件可以开始的时间

3. $ve(0)$约定为0，$v(n-1)$约定为用户指定的值

之后，活动的最早开始时间就是弧的起点的最早开始时间，活动的最迟开始时间就是弧终点的最迟开始时间减去活动的时间

算法：

1. 从原点v_1出发，令$ve0=0$，按拓扑序列求各顶点的最早开始时间vei
2. 从汇点v_n出发，令$vn-1=ven-1$，按逆拓扑序列求其余各顶点的最迟发生时间vi
3. 根据各顶点的ve和v值，计算每条弧的最早开始使劲爱你ei和最迟开始时间i，找出$ei=i$的关键活动

```C++
Status TopologicalOrder(ALGraph G, Stack &T) {
   InitStack(S);
   FindInDegree(G, indegree); // 在这里把所有入度为0的结点压入栈S
   InitStack(T); count = 0; ve[0..G.vexnum-1] = 0;
   while(!StackEmpty(S)) {
       Pop(S, j); Push(T, j); ++ count;
       for(p = G.vertices[j].firstarc; p; p=p->nextarc) {
           k = p->adjvex;
           if(--indegree[k] == 0) Push(S, k);
           if(ve[j] + *(p->info) > ve[k]) ve[k] = ve[j] + *(p->info);
       }
   }
   if(count < G.vexnum) return ERROR;
   else return OK;
}

Status CriticalPath(ALGraph G) {
   if(!TopologicalOrder(G, T)) return ERROR;
   vl[0..G.vexnm - 1] ve[G.vexnum - 1];
   while(!StackEmpty(T))
       for(Pop(T, j), p - G.vertices[j].firstarc; p; p->nextarc) {
           k = p->adjvex; dut = *(p->info);
           if(vl[k] - dut < vl[j]) vl[j] = vl[k] - dut;
       }
   for(j = 0; j<G.vexnum; ++j)
       for(p = G.vertices[j].firstarc; p; p->nextarc) {
           k = p->adjvex; dut = *(p->info);
           ee = ve[j]; el = vl[k] - dut;
           tag = (ee==el) ? '*' : '';
           printf(j, k, dut, ee, el, tag);
       }
}
```

复杂度$O(n+e)$

最短路径

问题提出：用带权的有向图表示一个交通运输网

- 顶点：表示城市
- 边：表示城市间的交通联系
- 权：表示此线路的长度或沿此线路运输所花的时间或费用等

问题：从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径——最短路径

从某个源点到其余各顶点的最短路径

Dijsktra算法：把图中所有顶点分为两组，第一组包含已确定最短路径的顶点，第二组包括尚未确定最短路径的顶点，按最短路径递增的顺序，逐个把第二组的顶点加到第一组去，直至v_0出发可以达到的所有顶点都包含到第一组中

步骤：

- 初始时令 $=\{v_0\}, T=\{$其余顶点$\}$，T中顶点对应的距离值
  - 若存在$< v_0, v_i >$，为$< v_0, v_i >$弧上的权值
  - 若不存在则为
- 从T中国选取一个其距离值为最小的顶点W，加入S
- 对T中顶点的距离值进行修改：若加进W作中间顶点，从v_0到v_i的距离值比不加的路径要短，则修改此距离值
- 重复上述步骤直到S中包含所有顶点，即S=V为止

```C++
void Dijkstra(MGraph G, int v0, PathMatrix &P, ShortPathTable &D) {
   for(v = 0; v < G.vexnum; ++v) {
       final[v] = false; D[v] = G.arcs[v0][v];
       for(w = 0; w < G.vexnum; ++w) P[v][w] = false;
       if(D[v] < INFINITY) { P[v][v0] = TRUE; P[v][v] = TRUE; }
   }
   D[v0] = 0; final[v0] = true;
   for(i = 1; i< G.vexnum; ++i) {
       min = INFTY;
       for(w = 0; w < G.vexnum; ++w)
           if(!final[w])
               if(D[w] < min) { v = w; min = D[w]; }
       final[v] = true;
       for(w = 0; w < G.vexnum; ++w)
           if(!final[w] && (min + G.arcs[v][w] < D[w])) {
           D[w] = min + G.arcs[v][w];
           P[w] = P[v]; P[w][w] = TRUE;
       }
   }
}
```

复杂度$O(n^2)$

每一对顶点之间的最短路径

方法一：每次以一个顶点为源点，重复执行Dijkstra算法n次，复杂度$O(n)$

方法二：Floyd-Warshall算法
基本思想：
递推地产生一个矩阵序列：$D^{(-1)}, D^{(0)}, \cdots, D^{(n-1)}$
其中：$D^{(-1)} =$
$D^{(-1)}ij$等于从顶点v_i到v_j中间顶点序号不大于-1（即不经过任何中间顶点）的最短路径长度
$D^{(0)}ij$等于从顶点v_i到v_j中间顶点序号不大于0的最短路径长度。即在v_i, v_j间考虑v_0顶点，也就是判别(v_i, v_j)和(v_i, v_0, v_j)（若存在）的最短者
$D^{(1)}ij$等于从顶点v_i到v_j中间顶点需要不大于1的最短路径长度
...
$D^{(n-1)}ij$等于从顶点v_i到v_j中间顶点需要不大于$n-1$的最短路径长度，即为所求v_i到v_j的最短路径长度
假设已经求得矩阵$D^{(-1)}$，求$D^{()}$
从顶点v_i到v_j中间顶点序号不大于的最短路径有两种情况：
1. 不经v，则$D^{()}ij = D^{(-1)}$
没有通路或有通路但经v长度值较大
2. 经v，则$D^{()}ij < D^{(-1)}ij$
若不小于，则属1的情况
而路径$D^{()}ij$为v_i经v到v_j的中间顶点序号不大于的最短路径由两段组成：一段是从v_i到v中间顶点序号不大于-1的最短路径，一段是从v到v_j中间顶点序号不大于-1的最短路径
即：$D^{()}ij = D^{(-1)}i + D^{(-1)}j$
条件：$D^{()}ij < D^{(-1)}ij$
合并两个条件得到：$D^{()}ij = (D^{(-1)}ij, D^{(-1)}i + D^{(-1)}j)$
$0 \leq \ \leq n-1$

C++

```
void Floyd(MGraph G, Path Matrix P[], DistanceMatrix &D) {
   for(v = 0; v < G.vexnum; ++v)
       for(w = 0; w < G.vexnum; ++w) {
           D[v][w] = G.arcs[v][w];
           for(u = 0; u < G.vexnum; ++u) P[v][w][u] = FALSE;
           if(D[v][w]  < INFINITY) {
               P[v][w][v] = true; P[v][w][w] = true;
           }
       }
   for(u = 0; u < G.vexnum; ++u)
       for(v = 0; v < G.vexnum; ++v)
           for(w = 0; w < G.vexnum; ++w)
               if(D[v][u] + D[u][w] < D[v][w]) {
                   D[v][w] = D[v][u] + D[u][w];
                   for(i = 0; i < G.vexnum; ++i)
                       P[v][w][i] = P[v][u][i] || P[u][w][i];
               }
}
```