

# 第七章 图

- ❖ 图的基本概念
- ❖ 图的存储结构
- ❖ 图的遍历
- ❖ 图的连通性问题
- ❖ 最小生成树
- ❖ 最短路径
- ❖ 活动网络

# 图的基本概念

- **图定义** 图是由顶点集合(vertex)及顶点间的关系集合组成的一种数据结构:

$$\text{Graph} = (V, E)$$

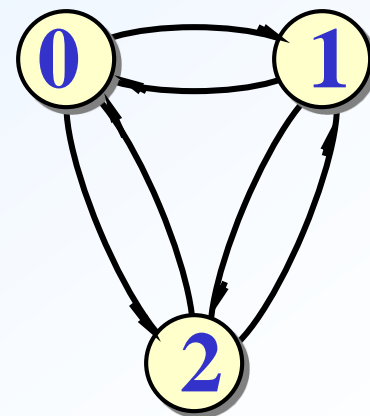
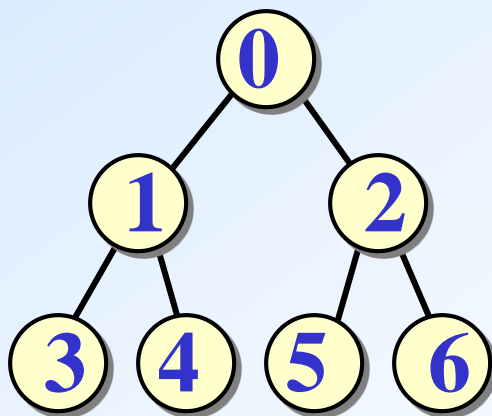
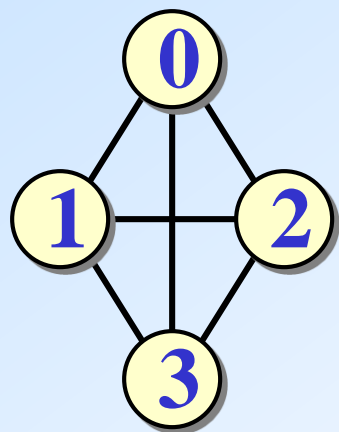
其中  $V = \{x \mid x \in \text{某个数据对象}\}$   
是顶点的有穷非空集合;

$$E1 = \{(x, y) \mid x, y \in V\}$$

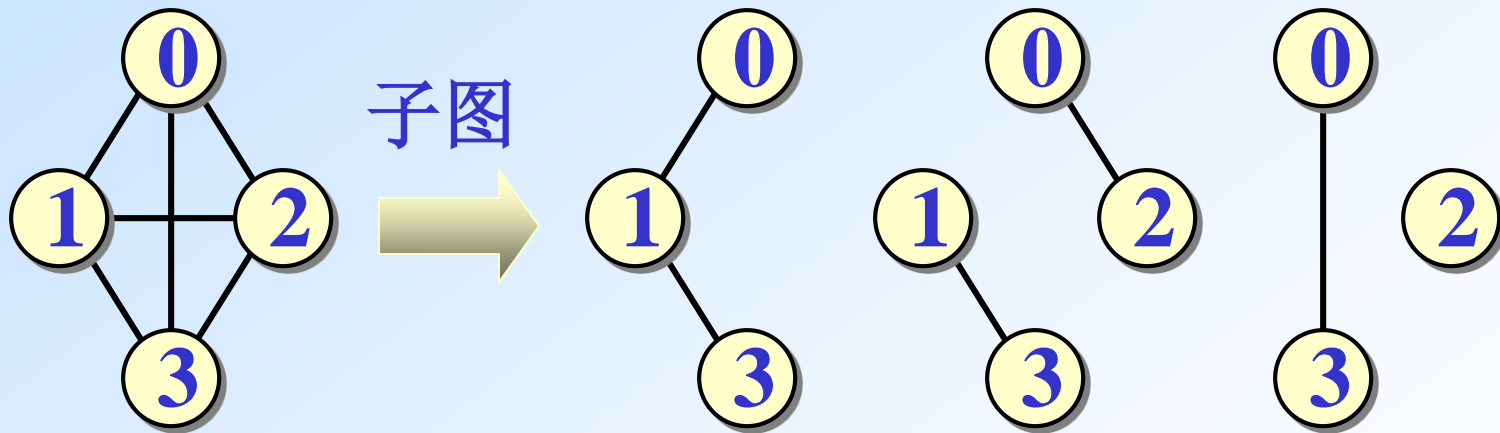
或  $E2 = \{ \langle x, y \rangle \mid x, y \in V \ \&\& \ \text{Path}(x, y) \}$

其中, **E1**是顶点之间关系的有穷集合, 也叫做边(edge)集合, 此时的图称为无向图。 **E2**表示从  $x$  到  $y$  的一条弧, 且称 $x$ 为弧尾,  $y$ 为弧头, 这样的图称为有向图。

- **有向图与无向图** 在有向图中，顶点对  $\langle x, y \rangle$  是有序的。在无向图中，顶点对  $(x, y)$  是无序的。
- **完全图** 若有  $n$  个顶点的无向图有  $n(n-1)/2$  条边，则此图为**无向完全图**。有  $n$  个顶点的有向图有  $n(n-1)$  条边，则此图为**有向完全图**。



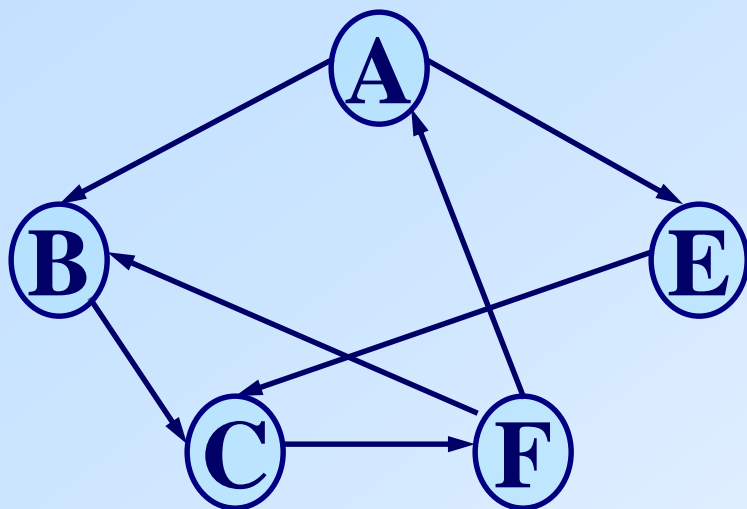
- **邻接顶点** 如果  $(u, v)$  是  $E(G)$  中的一条边，则称  $u$  与  $v$  互为邻接顶点。
- **子图** 设有两个图  $G=(V, E)$  和  $G'=(V', E')$ 。若  $V' \subseteq V$  且  $E' \subseteq E$ ，则称 图  $G'$  是 图  $G$  的子图。



- **权** 某些图的边具有与它相关的数，称之为权。这种带权图叫做网络。

- **顶点的度** 一个顶点 $v$ 的度是与它相关联的边的条数。记作 $TD(v)$ 。在有向图中,顶点的度等于该顶点的入度与出度之和。
- **顶点  $v$  的入度**是以  $v$  为终点的有向边的条数,记作  $ID(v)$ ; **顶点  $v$  的出度**是以  $v$  为始点的有向边的条数,记作  $OD(v)$ 。
- **路径** 在图  $G=(V, E)$  中,若从顶点  $v_i$  出发,沿一些边经过一些顶点  $vp_1, vp_2, \dots, vp_m$ , 到达顶点  $v_j$ 。则称顶点序列  $(v_i \ vp_1 \ vp_2 \dots vpm \ vj)$  为从顶点  $v_i$  到顶点  $v_j$  的路径。它经过的边  $(v_i, vp_1)$ 、 $(vp_1, vp_2)$ 、 $\dots$ 、 $(vp_m, v_j)$  应是属于  $E$  的边。

# 有向图



顶点的出度：以顶点v  
为弧尾的弧的数目；  
顶点的入度：以顶点v为  
弧头的弧的数目。

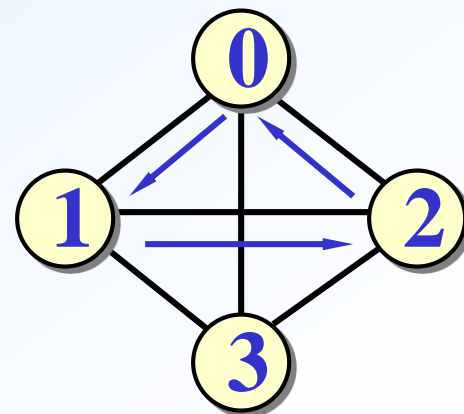
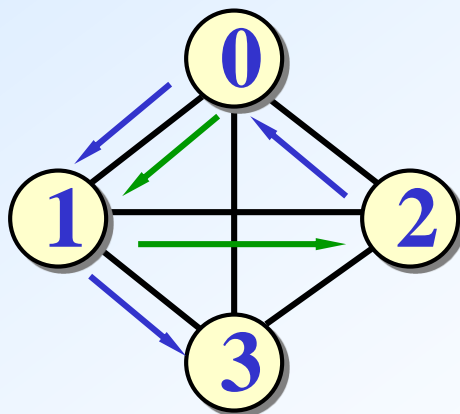
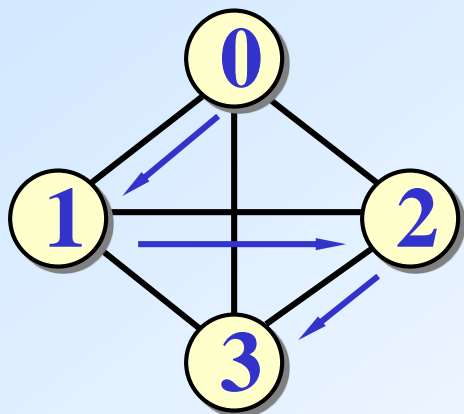


例如：

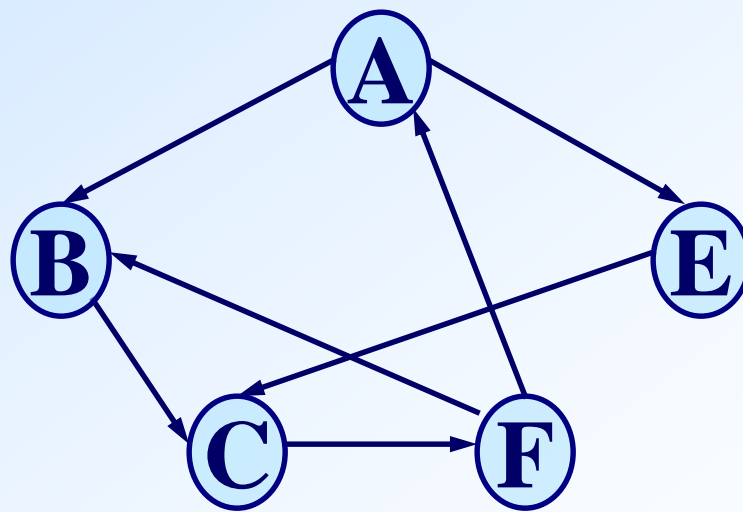
顶点的度 (TD) = 出度 (OD) + 入度 (ID)

$$TD(B) = OD(B) + ID(B) = 3$$

- **路径长度** 非带权图的路径长度是指此路径上边的条数。带权图的路径长度是指路径上各边的权之和。
- **简单路径** 若路径上各顶点  $v_1, v_2, \dots, v_m$  均不互相重复, 则称这样的路径为简单路径。
- **简单回路** 若路径上第一个顶点  $v_1$  与最后一个顶点  $v_m$  重合, 则称这样的路径为回路或环。



如：从A到F长度为 3  
的路径{A,B,C,F}

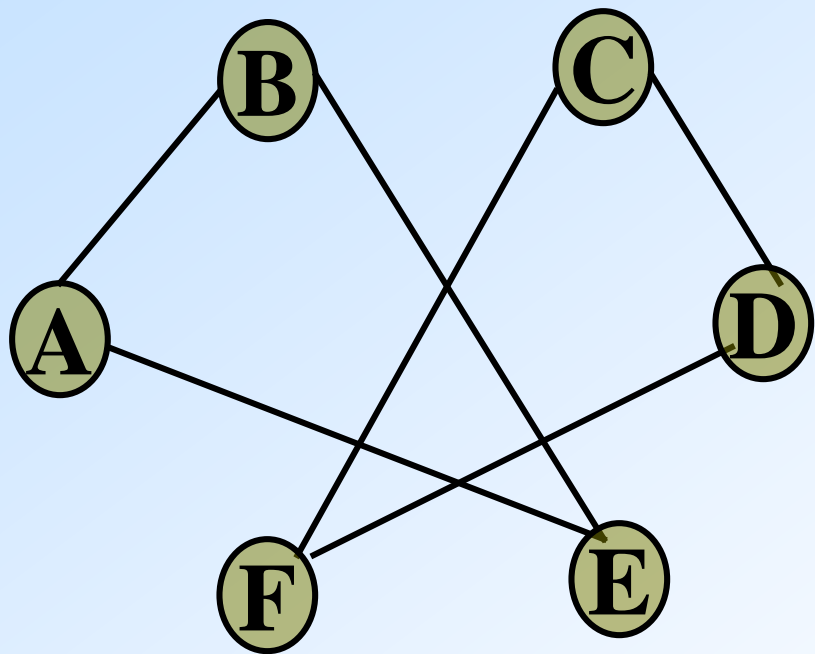
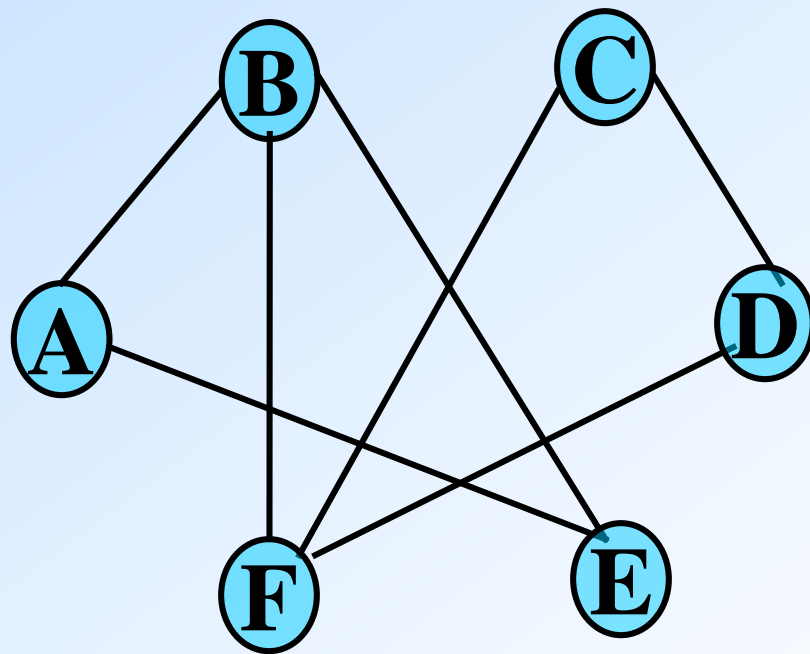




- **连通图与连通分量** 在无向图中, 若从顶点 $v_1$ 到顶点 $v_2$ 有路径, 则称顶点 $v_1$ 与 $v_2$ 是连通的。如果图中任意一对顶点都是连通的, 则称此图是连通图。非连通图的极大连通子图叫做连通分量。
- **强连通图与强连通分量** 在有向图中, 若对于每一对顶点 $v_i$ 和 $v_j$ , 都存在一条从 $v_i$ 到 $v_j$ 和从 $v_j$ 到 $v_i$ 的路径, 则称此图是强连通图。非强连通图的极大强连通子图叫做强连通分量。

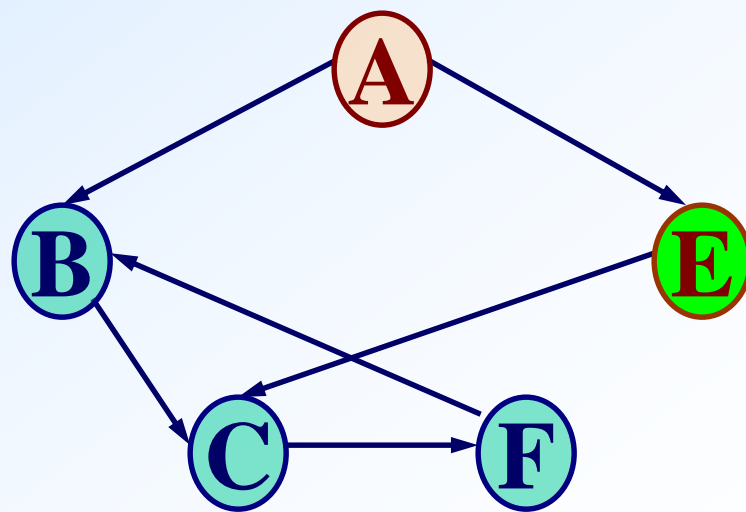
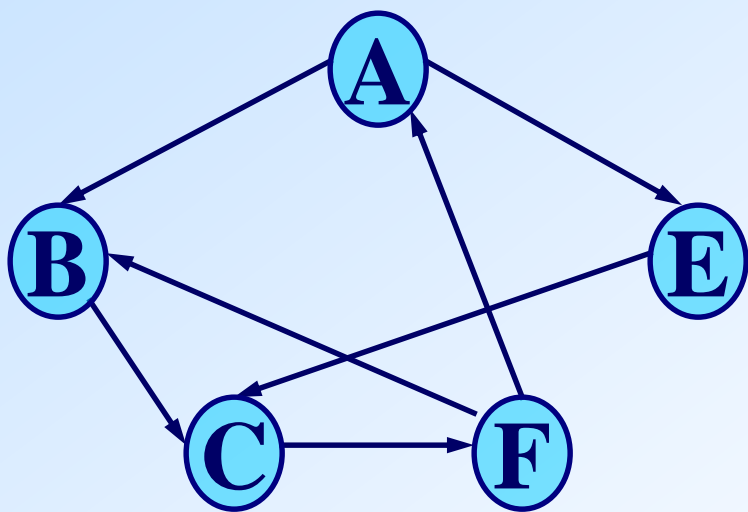


无向图,若图中任意两个顶点之间都有路径相通,则称此图为**连通图**;

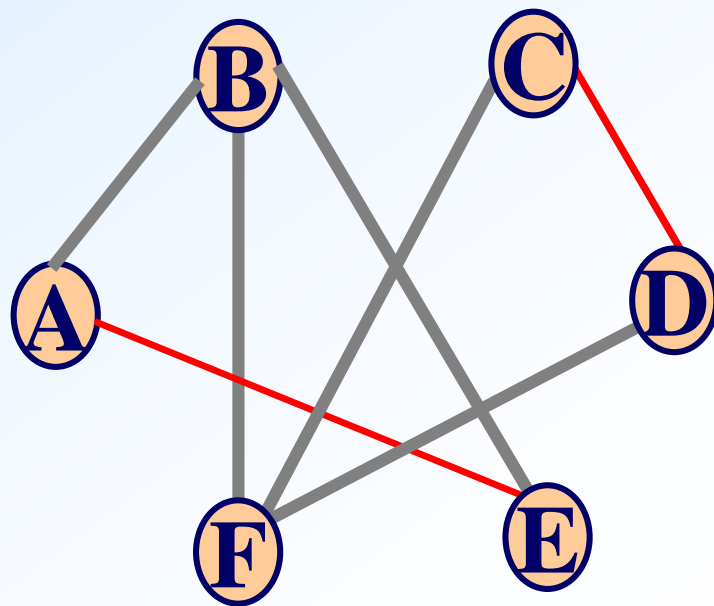
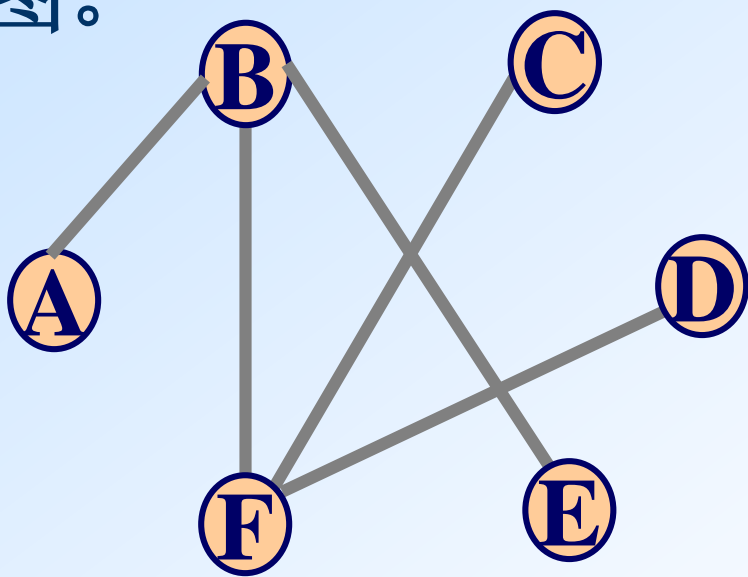


若无向图为非连通图,则图中各个极大连通子图称作此图的**连通分量**。

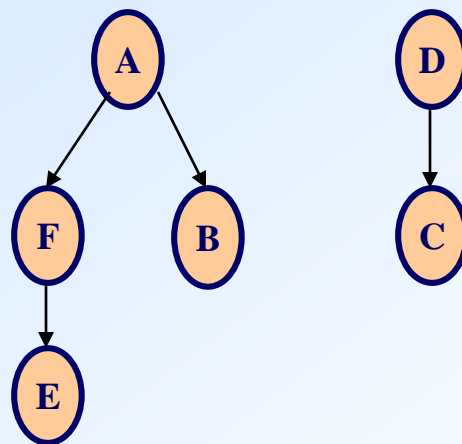
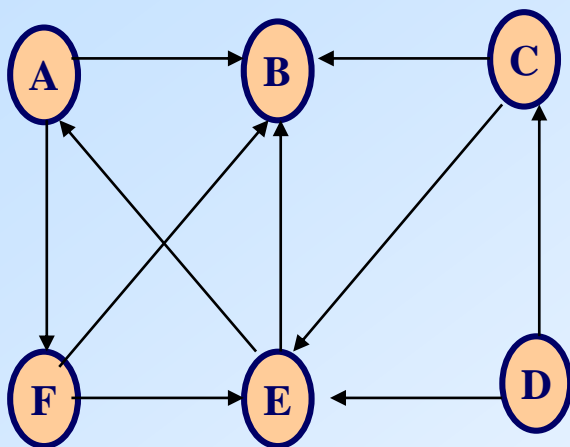
**有向图**,若任意两个顶点之间都存在一条有向路径,则称此有向图为**强连通图**。  
否则,其各个强连通子图称作它的**强连通分量**。



**生成树**: 假设一个连通图有 $n$ 个顶点和 $e$ 条边, 其中 $n-1$ 条边和 $n$ 个顶点构成一个极小连通子图, 称该极小连通子图为此连通图的**生成树**。在极小连通子图中增加一条边, 则一定有环。在极小连通子图中去掉一条边, 则成为非连通图。



有 $n$ 个顶点， $n-1$ 条边的图必定是生成树吗？



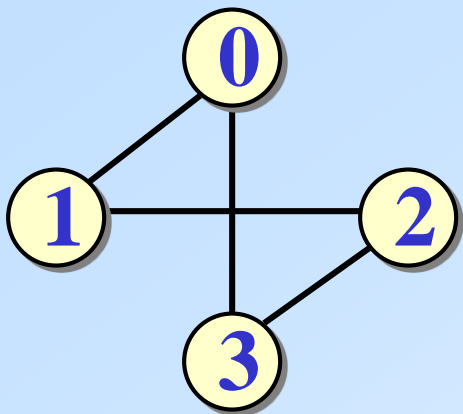
对非连通图，则称由各个连通分量的生成树的集合为此非连通图的**生成森林**。

# 图的存储结构

## 邻接矩阵 (Adjacency Matrix)

- 在图的邻接矩阵表示中，有一个记录各个顶点信息的顶点表，还有一个表示各个顶点之间关系的邻接矩阵。
- 设图  $A = (V, E)$  是一个有  $n$  个顶点的图，图的邻接矩阵是一个二维数组  $A.\text{edge}[n][n]$ ，定义：

$$A.\text{Edge}[i][j] = \begin{cases} 1, & \text{若 } \langle i, j \rangle \in E \text{ 或 } (i, j) \in E \\ 0, & \text{否则} \end{cases}$$



$$\mathbf{A}.\text{edge} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$



$$\mathbf{A}.\text{edge} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

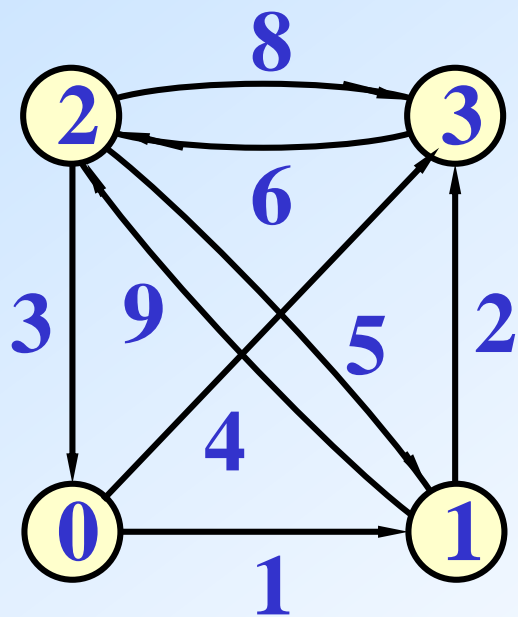
- 无向图的邻接矩阵是对称的；
- 有向图的邻接矩阵可能是不对称的。

- 在有向图中, 统计第  $i$  行 1 的个数可得顶点  $i$  的出度, 统计第  $j$  列 1 的个数可得顶点  $j$  的入度。
- 在无向图中, 统计第  $i$  行 (列) 1 的个数可得顶点  $i$  的度。



## 网络的邻接矩阵

$$\mathbf{A}.\text{edge}[i][j] = \begin{cases} \mathbf{W}(i,j), & \text{若 } i \neq j \text{ 且 } \langle i,j \rangle \in \mathbf{E} \text{ 或 } (i,j) \in \mathbf{E} \\ \infty, & \text{若 } i \neq j \text{ 且 } \langle i,j \rangle \notin \mathbf{E} \text{ 或 } (i,j) \notin \mathbf{E} \\ 0, & \text{若 } i == j \end{cases}$$



$$\mathbf{A}.\text{edge} = \begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix}$$

# 用邻接矩阵表示的结构定义

```
#define MaxValue Int_Max
const int NumEdges = 50;      //边条数
const int NumVertices = 10;  //顶点个数
typedef char VertexData;      //顶点数据类型
typedef int EdgeData;         //边上权值类型
typedef struct {
    VertexData vexList[NumVertices]; //顶点表
    EdgeData Edge[NumVertices][NumVertices];
    //邻接矩阵, 可视为边之间的关系
    int n, e; //图中当前的顶点个数与边数
} MTGraph;
```

# 邻接表 (Adjacency List)

- **邻接表:**是图的一种链式存储结构。

- **弧的结点结构**

adjvex	nextarc	info
--------	---------	------

**adjvex;** // 该弧所指向的顶点的位置

**nextarc;** // 指向下一条弧指针

**info;** // 该弧相关信息的指针

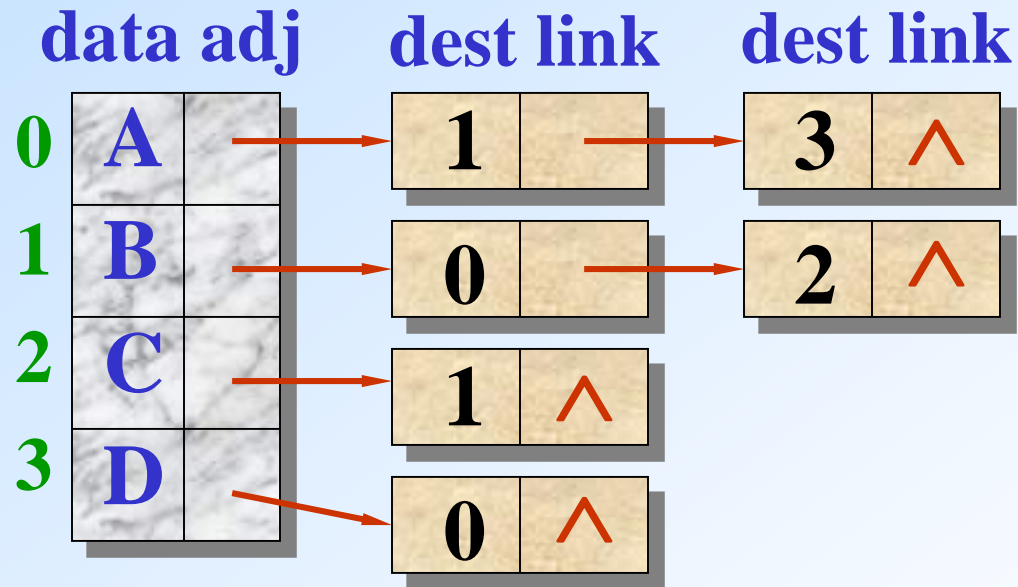
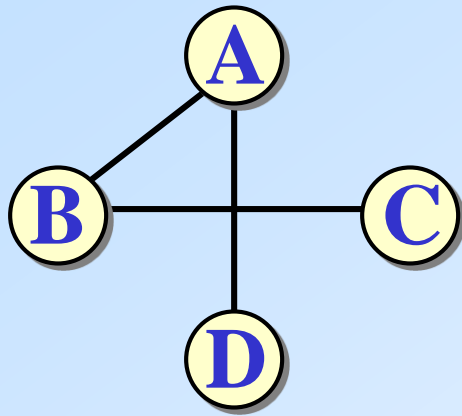
- **顶点的结点结构**

data	firstarc
------	----------

**data;** // 顶点信息

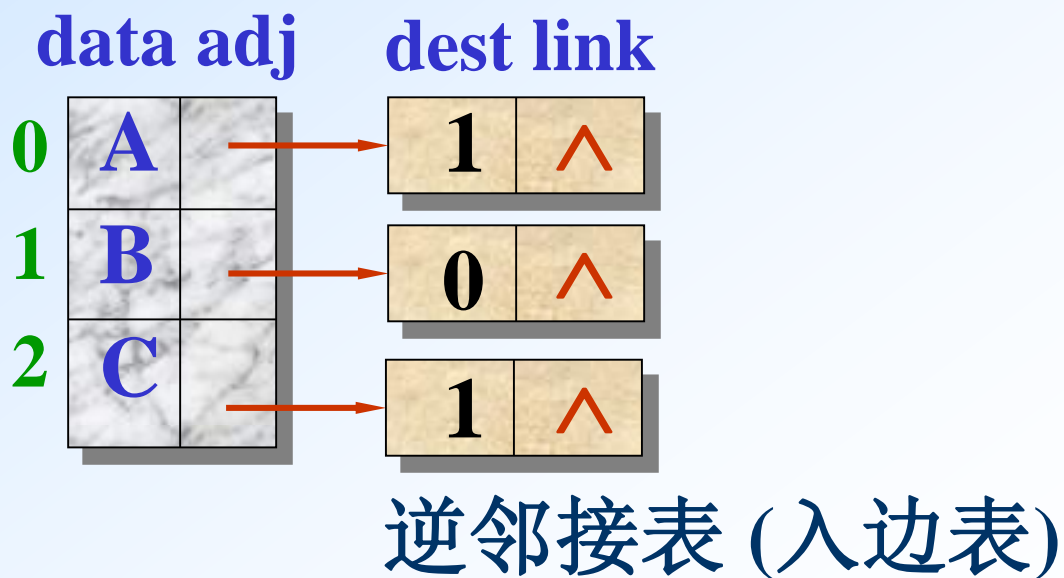
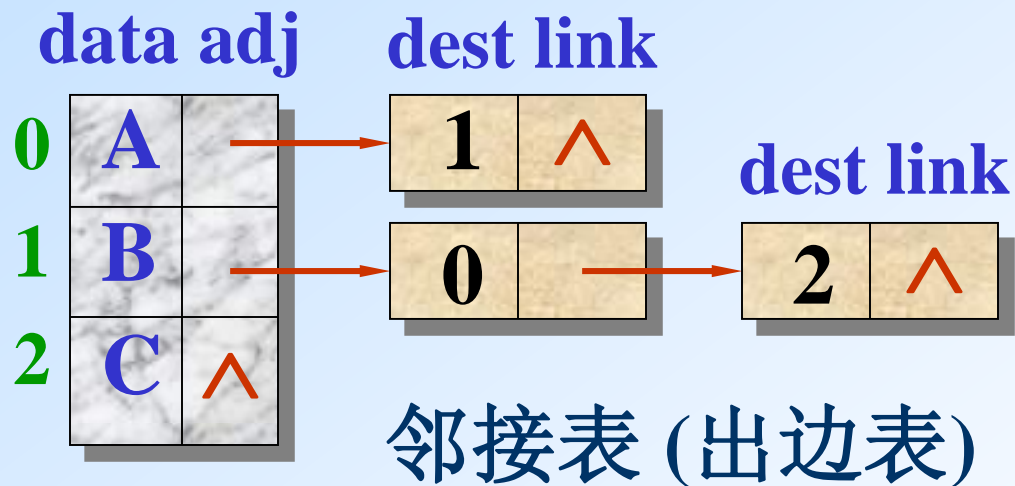
**firstarc;** //指向第一条依附该顶点的弧

## ■ 无向图的邻接表

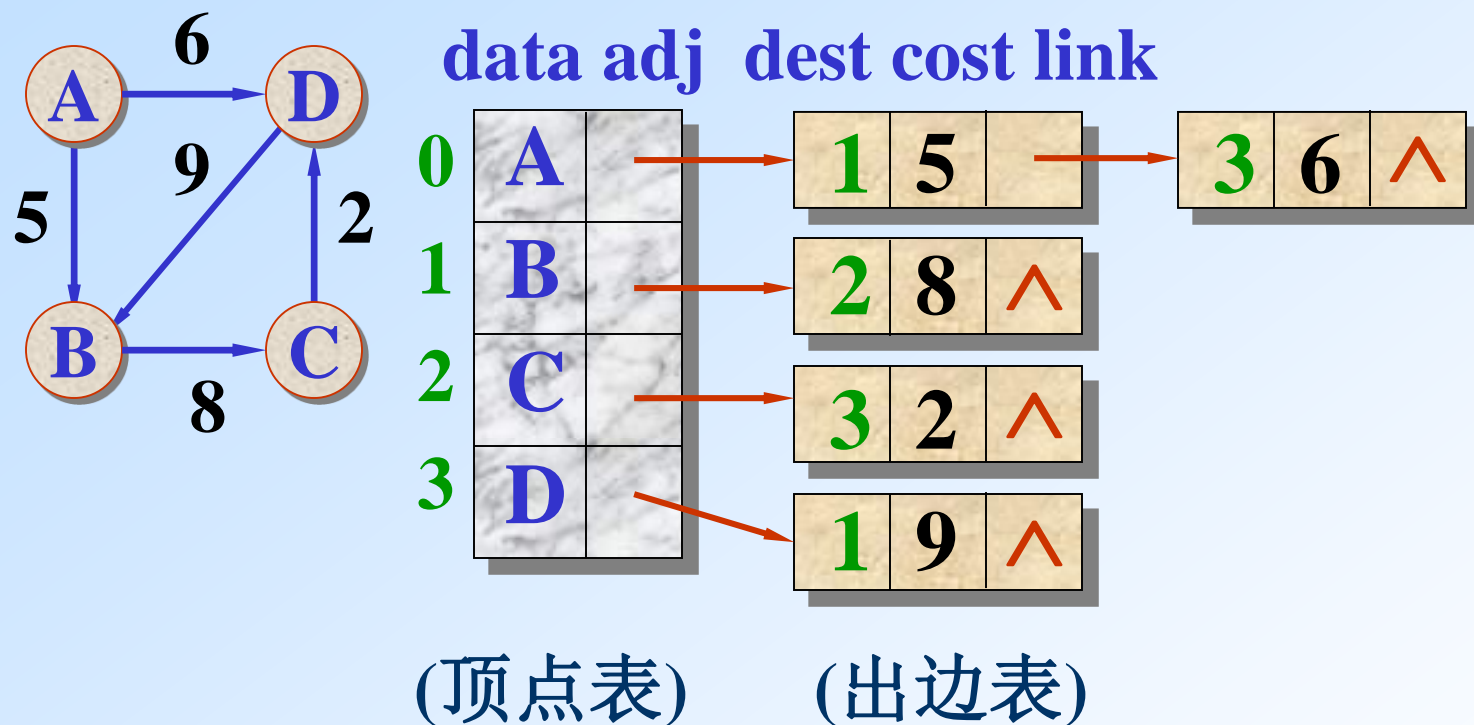


同一个顶点发出的边链接在同一个边链表中，每一个链结点代表一条边(边结点)，结点中有另一顶点的下标 **dest** 和指针 **link**。

# 有向图的邻接表和逆邻接表



## ■ 网络 (带权图) 的邻接表



- 带权图的边结点中保存该边上的权值 **cost**。
- 顶点 **i** 的边链表的表头指针 **adj** 在顶点表的下标为 **i** 的顶点记录中，该记录还保存了该顶点的其它信息。
- 在邻接表的边链表中，各个边结点的链入顺序任意，视边结点输入次序而定。
- 设图中有 **n** 个顶点，**e** 条边，则用邻接表表示无向图时，需要 **n** 个顶点结点，**2e** 个边结点；用邻接表表示有向图时，若不考虑逆邻接表，只需 **n** 个顶点结点，**e** 个边结点。

# 邻接表表示的图的定义（为算法）

```
typedef char VertexData;           //顶点数据类型
typedef int EdgeData;             //边上权值类型

typedef struct node {               //边结点
    int dest;                       //目标顶点下标
    EdgeData cost;                 //边上的权值
    Struct node * link;           //下一边链接指针
} EdgeNode;
```



```
typedef struct {                                //顶点结点
    VertexData data;                            //顶点数据域
    EdgeNode * firstAdj;                       //边链表头指针
} VertexNode;
```

```
typedef struct {                                //图的邻接表
    VertexNode VexList [NumVertices]; //邻接表
    int n, e;                                //图中当前的顶点个数与边数
} AdjGraph;
```

## 邻接表的构造算法(无向图)

```
void CreateGraph (AdjGraph G) {  
    cin >> G.n >> G.e;    //输入顶点个数和边数  
    for ( int i = 0; i < G.n; i++) {  
        cin >> G.VexList[i].data;    //输入顶点信息  
        G.VexList[i].firstAdj = NULL;  
    }  
    for ( i = 0; i < e; i++) {    //逐条边输入  
        cin >> tail >> head >> weight;  
        EdgeNode * p = new EdgeNode;  
        p->dest = head;  p->cost = weight;  
    }  
}
```

**//链入第 tail 号链表的前端**

```
p->link = G.VexList[tail].firstAdj;  
G.VexList[tail].firstAdj = p;
```

```
p = new EdgeNode;
```

```
p->dest = tail; p->cost = weight;
```

**//链入第 head 号链表的前端**

```
p->link = G.VexList[head].firstAdj;  
G.VexList[head].firstAdj = p;
```

```
}
```

```
}
```



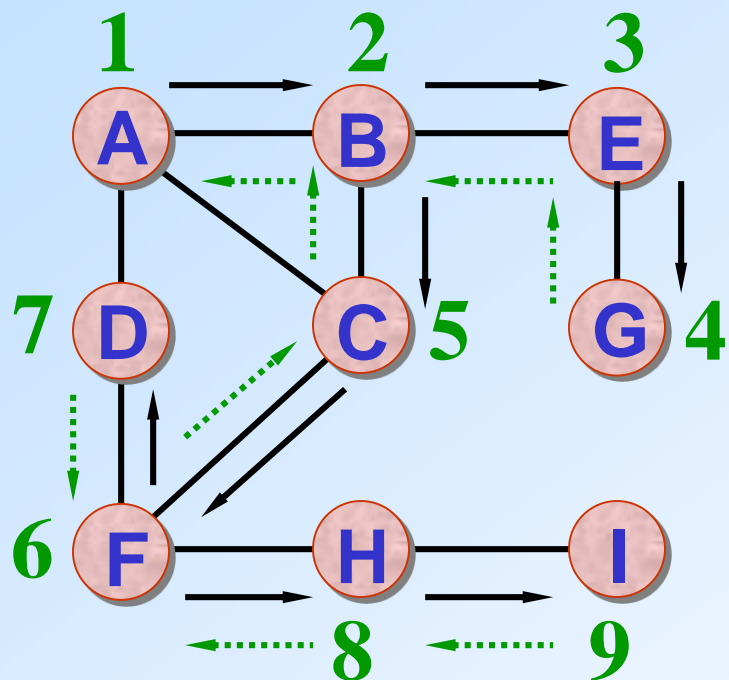
# 图的遍历

- 从图中某一顶点出发访遍图中所有的顶点，且使每个顶点仅被访问一次，这一过程就叫做图的遍历 (Traversing Graph)。
- 图中可能存在回路，且图的任一顶点都可能与其它顶点相通，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。
- 为了避免重复访问，可设置一个标志顶点是否被访问过的辅助数组 `visited [ ]`。

- 辅助数组 **visited [ ]** 的初始状态为 **0**, 在图的遍历过程中, 一旦某一个顶点 **i** 被访问, 就立即让 **visited [i]** 为 **1**, 防止它被多次访问。
- 两种图的遍历方法:
  - ◆ 深度优先搜索  
**DFS (Depth First Search)**
  - ◆ 广度优先搜索  
**BFS (Breadth First Search)**

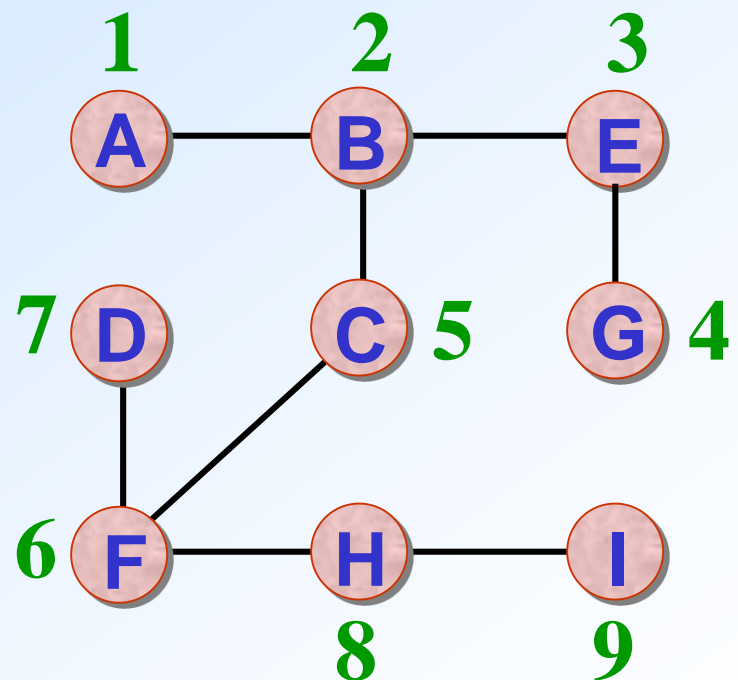
# 深度优先搜索DFS ( Depth First Search )

## ■ 深度优先搜索过程



前进 —————→

回退 ······→



深度优先生成树

- **DFS** 在访问图中某一起始顶点  $v$  后, 由  $v$  出发, 访问它的任一邻接顶点  $w_1$ ; 再从  $w_1$  出发, 访问与  $w_1$  邻接但还没有访问过的顶点  $w_2$ ; 然后再从  $w_2$  出发, 进行类似的访问, ... 如此进行下去, 直至到达所有的邻接顶点都被访问过的顶点  $u$  为止。接着, 退回一步, 退到前一次刚访问过的顶点, 看是否还有其它没有被访问的邻接顶点。如果有, 则访问此顶点, 之后再从此顶点出发, 进行与前述类似的访问; 如果没有, 就再退回一步进行搜索。重复上述过程, 直到连通图中所有顶点都被访问过为止。

## 图的深度优先搜索算法

```
void Graph_Traverse (AdjGraph G) {  
    int * visited = new int [NumVertices];  
    for ( int i = 0; i < G.n; i++ )  
        visited [i] = 0; //访问数组 visited 初始化  
    for ( int i = 0; i < G.n; i++ )  
        if ( ! visited[i] ) DFS (G, i, visited );  
                                //从顶点 i 出发开始搜索  
    delete [ ] visited; //释放 visited  
}
```



```

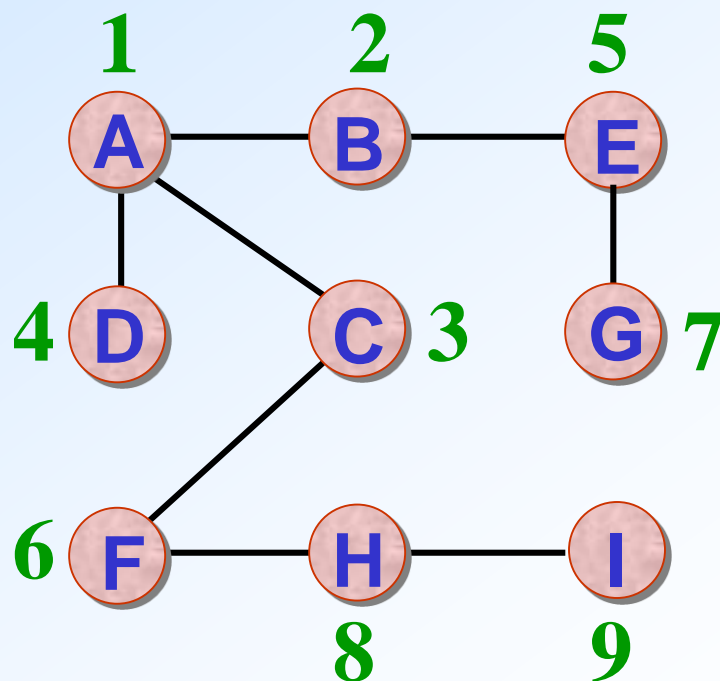
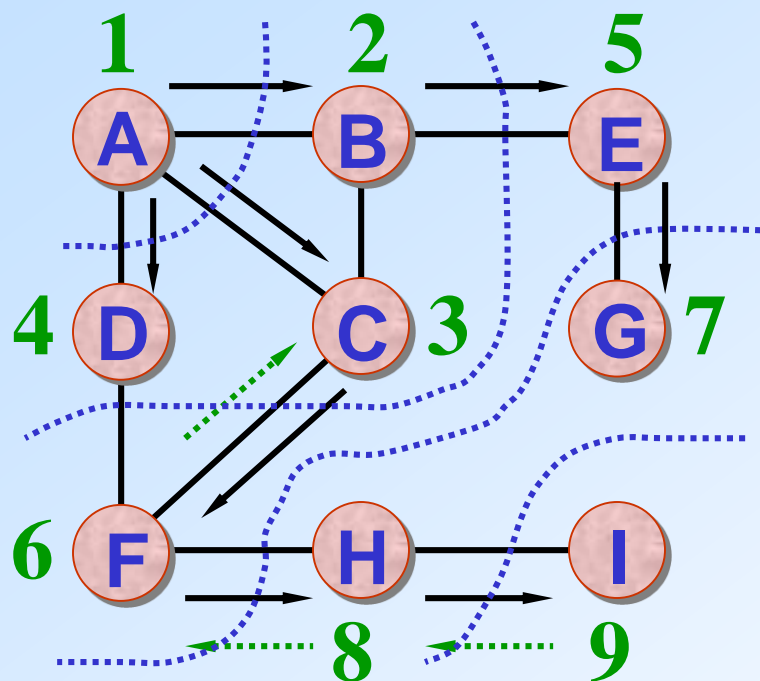
void DFS (AdjGraph G, int v, int visited [ ] ) {
    cout << GetValue (G, v) << ' '; //访问顶点 v
    visited[v] = 1;                    //顶点 v 作访问标记
    int w = GetFirstNeighbor (G, v);
    //取 v 的第一个邻接顶点 w
    while ( w != -1 ) {                //若邻接顶点 w 存在
        if ( !visited[w] ) DFS (G, w, visited );
        //若顶点 w 未访问过, 递归访问顶点 w
        w = GetNextNeighbor (G, v, w );
        //取顶点 v 排在 w 后的下一个邻接顶点
    }
}

```



# 广度优先搜索BFS ( Breadth First Search )

## ■ 广度优先搜索过程



广度优先生成树

- **BFS**在访问了起始顶点  $v$  之后,由  $v$  出发,依次访问  $v$  的各个未被访问过的邻接顶点  $w_1, w_2, \dots, w_t$ ,然后再顺序访问  $w_1, w_2, \dots, w_t$  的所有还未被访问过的邻接顶点。再从这些访问过的顶点出发,再访问它们的所有还未被访问过的邻接顶点, ... 如此做下去,直到图中所有顶点都被访问到为止。
- 广度优先搜索是一种分层的搜索过程,每向前走一步可能访问一批顶点,不像深度优先搜索那样有往回退的情况。因此,广度优先搜索不是一个递归的过程。

- 为了实现逐层访问, 算法中使用了一个队列, 以记忆正在访问的这一层和下一层的顶点, 以便于向下一层访问。
- 为避免重复访问, 需要一个辅助数组 **visited** [], 给被访问过的顶点加标记。

## 图的广度优先搜索算法

```
void Graph_Traverse (AdjGraph G) {  
    .....  
    for ( int i = 0; i < G.n; i++ )  
        if ( ! visited[i] ) BFS (G, i, visited );  
    .....  
}
```

```
void BFS (AdjGraph G, int v, int visited[ ] ) {  
    cout << GetValue (v) << ' ';  visited[v] = 1;  
    Queue<int> q;  InitQueue(&q);  
    EnQueue (&q, v);  //进队列  
    while ( ! QueueEmpty (&q) ) {  //队空搜索结束  
        DeQueue (&q, v);  
        int w = GetFirstNeighbor (G, v);  
        while ( w != -1 ) {  //若邻接顶点 w 存在  
            if ( !visited[w] ) {  //未访问过  
                cout << GetValue (w) << ' ';
```

```
        visited[w] = 1; EnQueue (&q, w);
    }
    w = GetNextNeighbor (G, v, w);
}    //重复检测 v 的所有邻接顶点
}    //外层循环，判队列空否
delete [ ] visited;
}
```



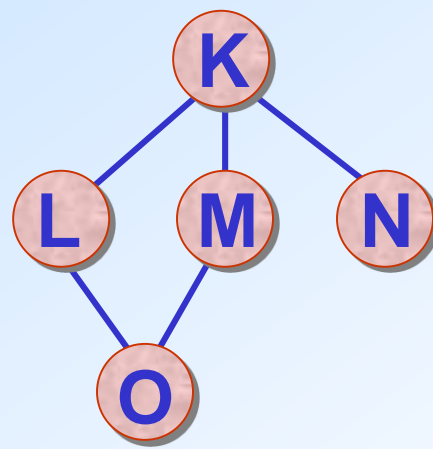
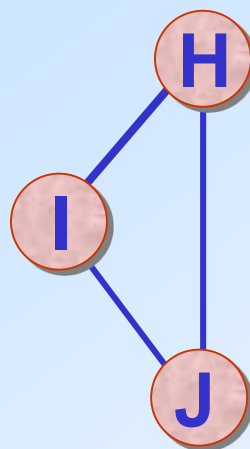
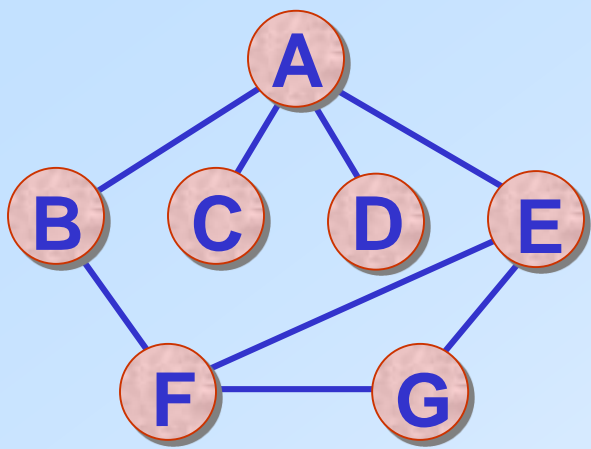
# 图的连通性问题

## 连通分量 (Connected component)

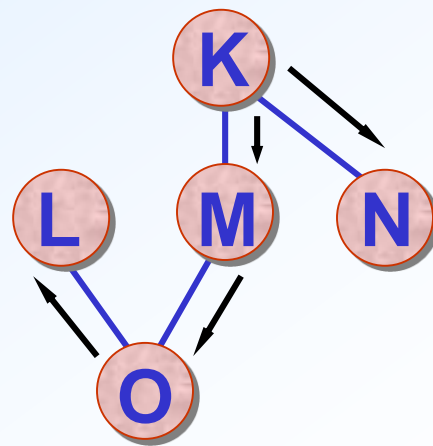
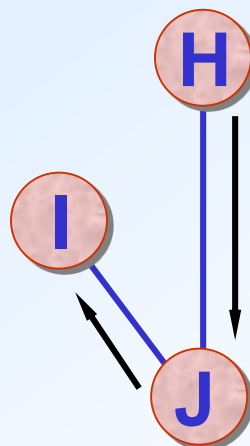
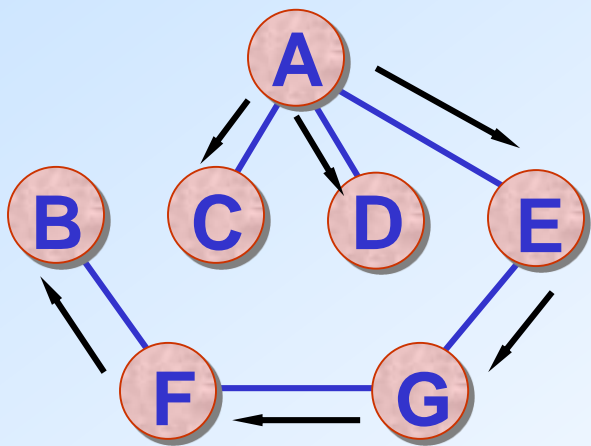
- 当无向图为非连通图时, 从图中某一顶点出发, 利用深度优先搜索算法或广度优先搜索算法不可能遍历到图中的所有顶点, 只能访问到该顶点所在的**最大连通子图**(连通分量)的所有顶点。
- 若从无向图的每一个连通分量中的一个顶点出发进行遍历, 可求得无向图的所有连通分量。

- 求连通分量的算法需要对图的每一个顶点进行检测：若已被访问过，则该顶点一定是落在图中已求得的连通分量上；若还未被访问，则从该顶点出发遍历图，可求得图的另一个连通分量。
- 对于非连通的无向图，所有连通分量的生成树组成了非连通图的生成森林。





非连通无向图的三个连通分量

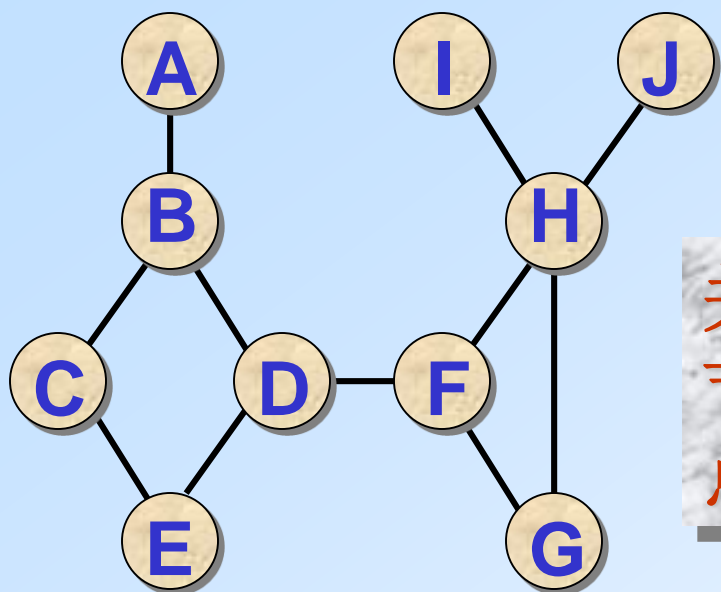


非连通图的连通分量的极小连通子图

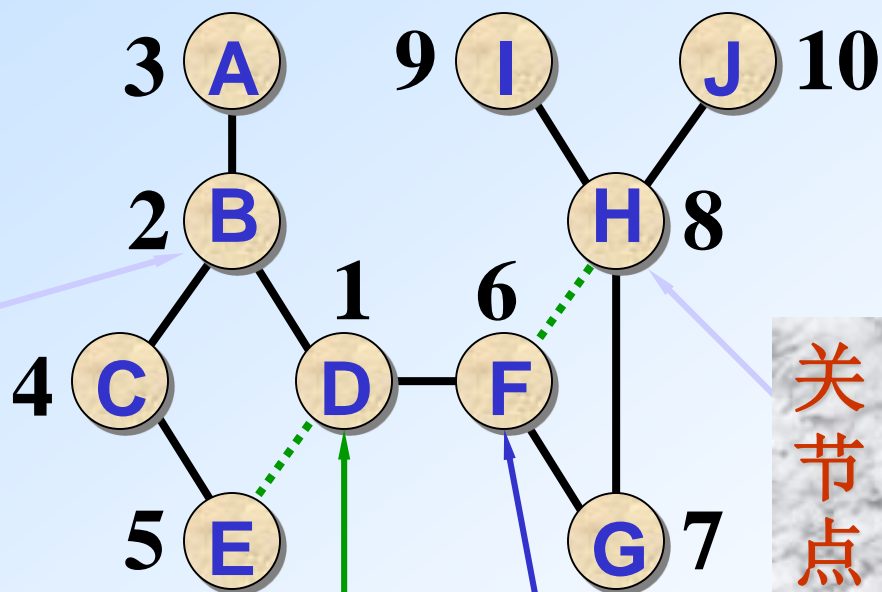
# 重连通分量 (Biconnected Component)

- 在无向连通图 $G$ 中, 当且仅当删去 $G$ 中的顶点 $v$ 及所有依附于 $v$ 的所有边后, 可将图分割成两个或两个以上的连通分量, 则称顶点 $v$ 为关节点。
- 没有关节点的连通图叫做重连通图。
- 在重连通图上, 任何一对顶点之间至少存在有两条路径, 在删去某个顶点及与该顶点相关联的边时, 也不破坏图的连通性。
- 一个连通图 $G$ 如果不是重连通图, 那么它可以包括几个重连通分量。

- 在一个无向连通图 $G$ 中, 重连通分量可以利用深度优先生成树找到。
- 在图中各顶点旁标明的深度优先数, 给出进行深度优先搜索时各顶点访问的次序。
- 深度优先生成树的根是关节点的充要条件是它至少有两个子女。
- 其它顶点  $u$  是关节点的充要条件是它至少有一个子女  $w$ , 从  $w$  出发, 不能通过  $w$ 、 $w$  的子孙及一条回边所组成的路径到达  $u$  的祖先。



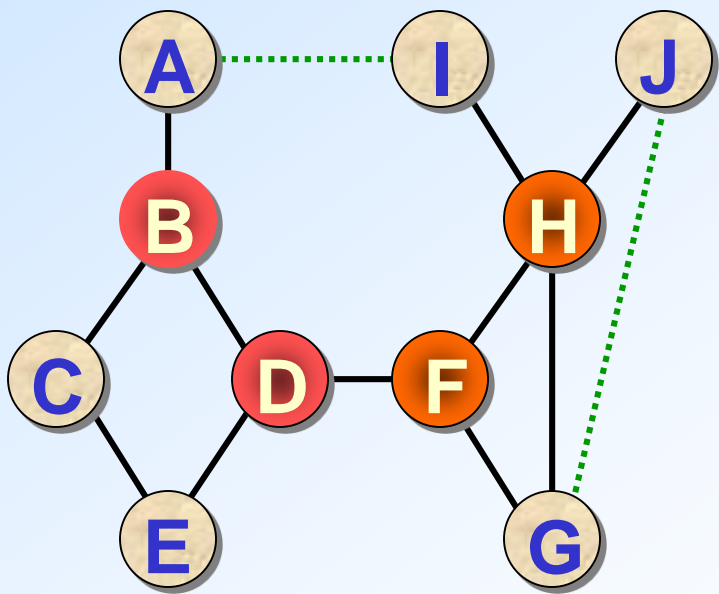
关节点

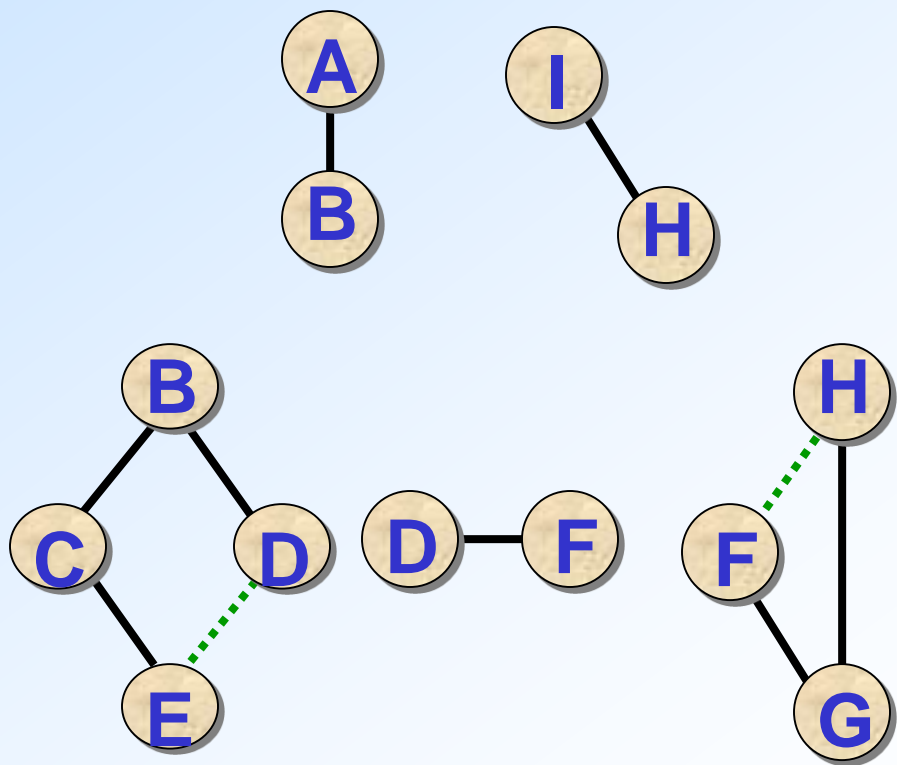
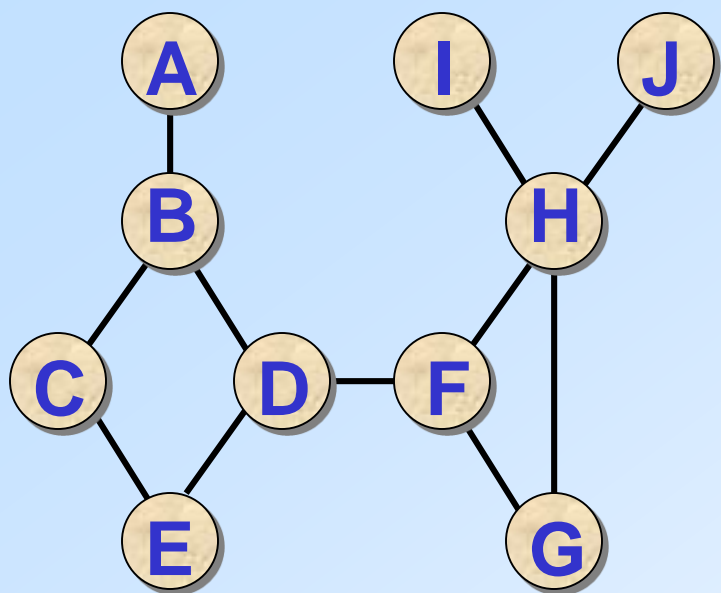


关节点

关节点

根有两个子女





连通图和它的连通分量



# 最小生成树 ( minimum cost spanning tree )

- 使用不同的遍历图的方法，可以得到不同的生成树；从不同的顶点出发，也可能得到不同的生成树。
- 按照生成树的定义， $n$  个顶点的连通网络的生成树有  $n$  个顶点、 $n-1$  条边。
- 构造最小生成树的准则
  - 必须使用且仅使用该网络中的 $n-1$  条边来联结网络中的  $n$  个顶点；
  - 不能使用产生回路的边；
  - 各边上的权值的总和达到最小。

# 普里姆(Prim)算法

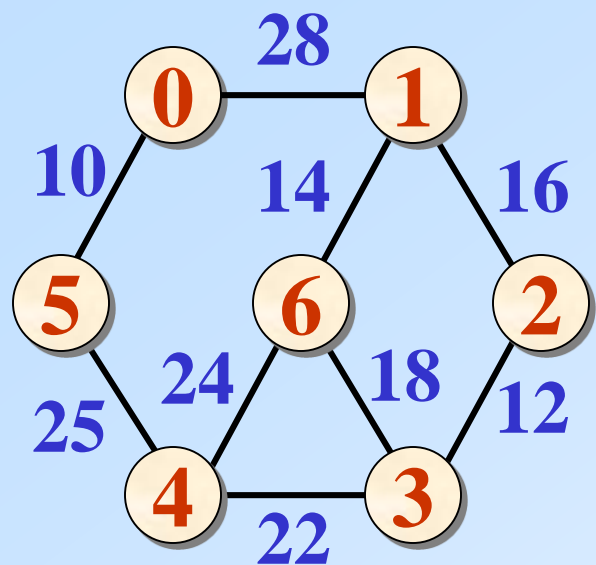
- 普里姆算法的**基本思想**:

从连通网络  $N = \{ V, E \}$  中的某一顶点  $u_0$  出发, 选择与它关联的具有最小权值的边  $(u_0, v)$ , 将其顶点加入到生成树顶点集合  $U$  中。

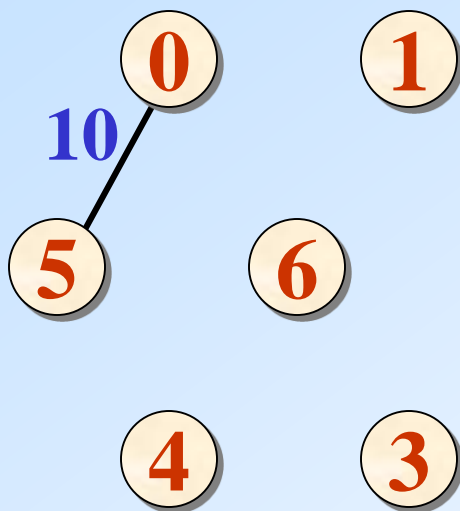
以后每一步从一个顶点在  $U$  中, 而另一个顶点不在  $U$  中的各条边中选择权值最小的边  $(u, v)$ , 把它的顶点加入到集合  $U$  中。如此继续下去, 直到网络中的所有顶点都加入到生成树顶点集合  $U$  中为止。

- 采用邻接矩阵作为图的存储表示。

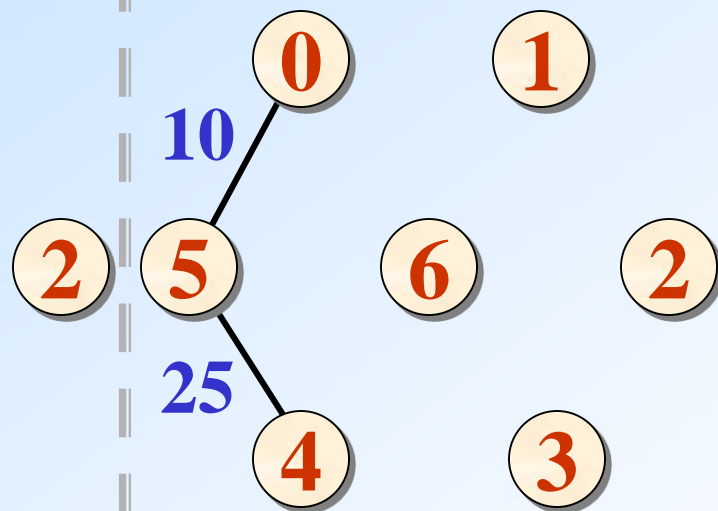




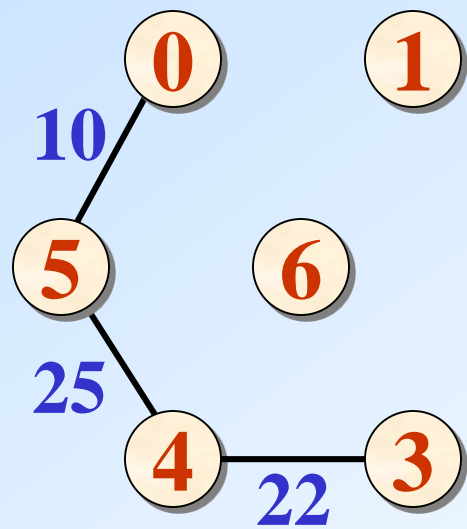
原图



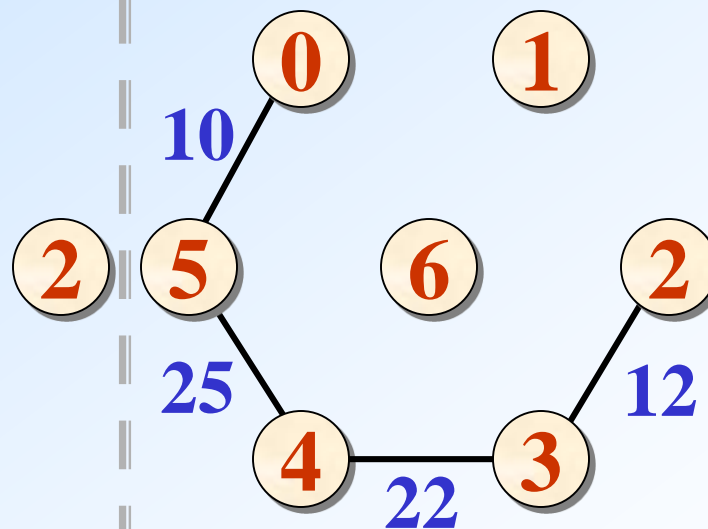
(a)



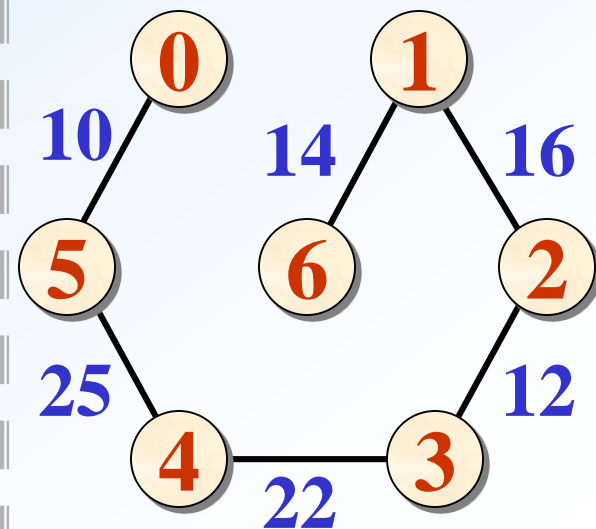
(b)



(c)



(d)

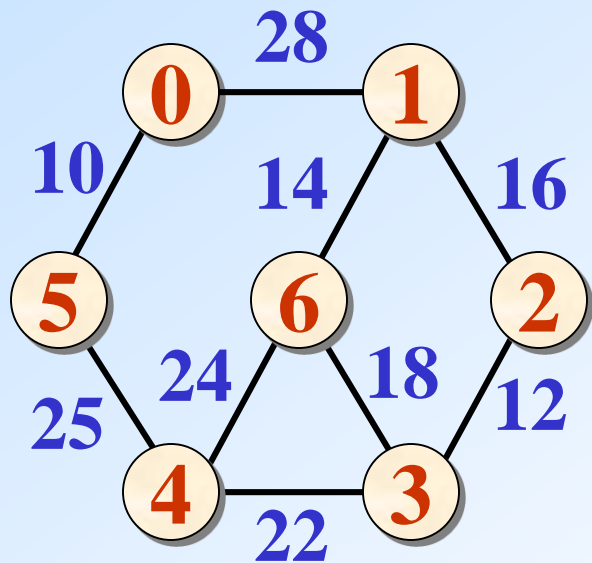


(e) (f)



- 在构造过程中，还设置了两个辅助数组：
  - ◆ **lowcost[ ]** 存放生成树顶点集合内顶点到生成树外各顶点的各边上的当前最小权值；
  - ◆ **nearvex[ ]** 记录生成树顶点集合外各顶点距离集合内哪个顶点最近(即权值最小)。

■ 例子



0	28	$\infty$	$\infty$	$\infty$	10	$\infty$
28	0	16	$\infty$	$\infty$	$\infty$	14
$\infty$	16	0	12	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	12	0	22	$\infty$	18
$\infty$	$\infty$	$\infty$	22	0	25	24
10	$\infty$	$\infty$	$\infty$	25	0	$\infty$
$\infty$	14	$\infty$	18	24	$\infty$	0

- 若选择从顶点0出发，即 $u_0 = 0$ ，则两个辅助数组的初始状态为：


	0	1	2	3	4	5	6
lowcost	0	28	$\infty$	$\infty$	$\infty$	10	$\infty$
nearvex	-1	0	0	0	0	0	0

- 然后反复做以下工作：
  - ◆ 在 lowcost [ ]中选择 nearvex[i]  $\neq$  -1 && lowcost[i]最小的边, 用 v 标记它。则选中的权值最小的边为(nearvex[v], v), 相应的权值为 lowcost[v]。

- ◆ 将  $\text{nearvex}[v]$  改为-1, 表示它已加入生成树顶点集合。
- ◆ 将边  $(\text{nearvex}[v], v, \text{lowcost}[v])$  加入生成树的边集合。
- ◆ 取  $\text{lowcost}[i] = \min\{ \text{lowcost}[i], \text{Edge}[v][i] \}$ , 即用生成树顶点集合外各顶点  $i$  到刚加入该集合的新顶点  $v$  的距离  $\text{Edge}[v][i]$  与原来它们到生成树顶点集合中顶点的最短距离  $\text{lowcost}[i]$  做比较, 取距离近的作为这些集合外顶点到生成树顶点集合内顶点的最短距离。

- ◆ 如果生成树顶点集合外顶点  $i$  到刚加入该集合的新顶点  $v$  的距离比原来它到生成树顶点集合中顶点的最短距离还要近, 则修改  $\text{nearvex}[i] : \text{nearvex}[i] = v$ 。表示生成树外顶点  $i$  到生成树内顶点  $v$  当前距离最近。

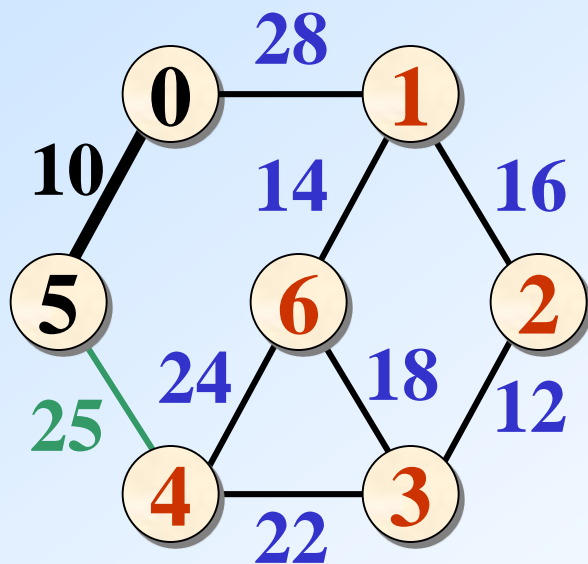
	0	1	2	3	4	5	6
lowcost	0	28	$\infty$	$\infty$	$\infty$	10	$\infty$
nearvex	-1	0	0	0	0	0	0

选  $v=5$   选边 (0,5)

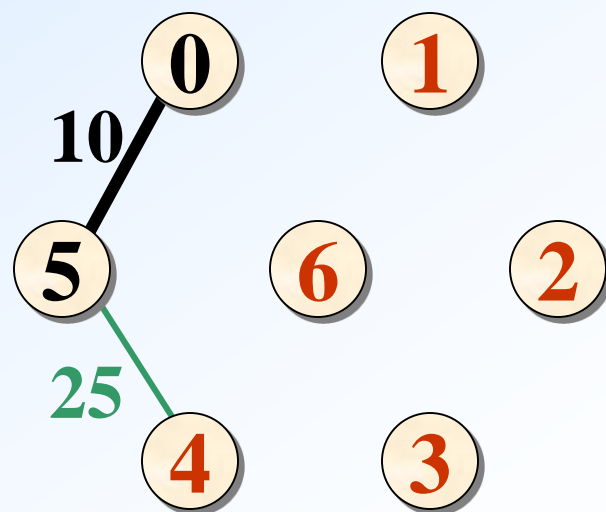
顶点  $v=5$  加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	28	$\infty$	$\infty$	25	10	$\infty$
nearvex	-1	0	0	0	5	-1	0

选  $v=4$   $\uparrow$  选边 (5,4)



原图




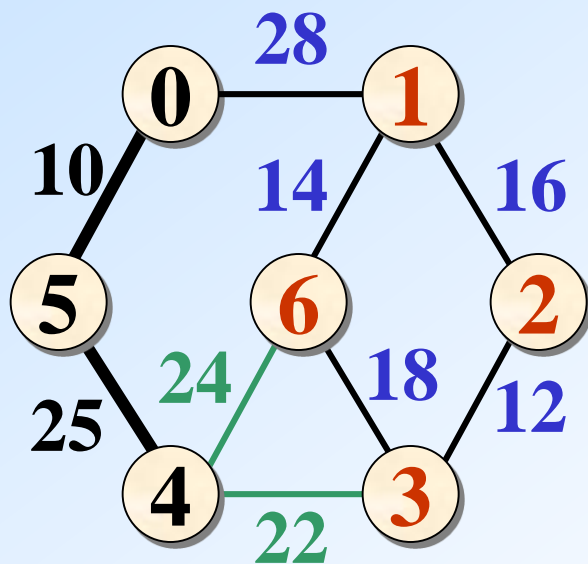
边 (0,5,10) 加入生成树<sub>55</sub>

顶点  $v=4$  加入生成树顶点集合:

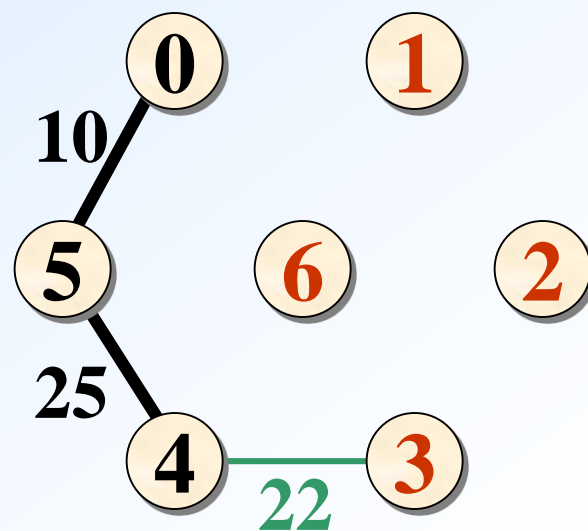
	0	1	2	3	4	5	6
lowcost	0	28	$\infty$	22	25	10	24

nearvex	-1	0	0	4	-1	-1	4
---------	----	---	---	---	----	----	---

选  $v=3$   选边 (4,3)



原图

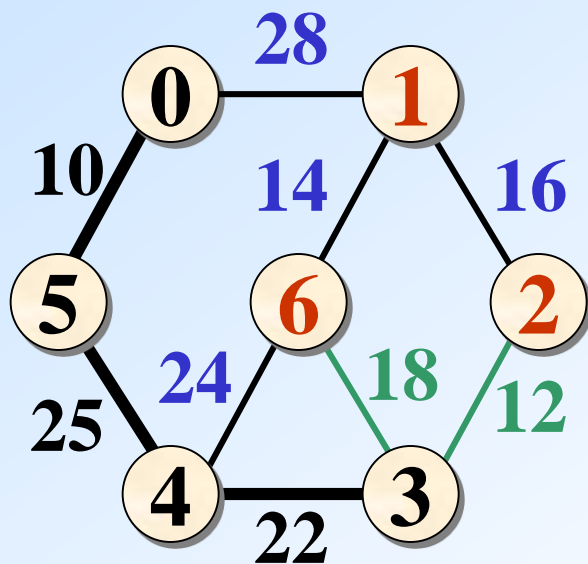


边 (5,4,25) 加入生成树<sub>56</sub>

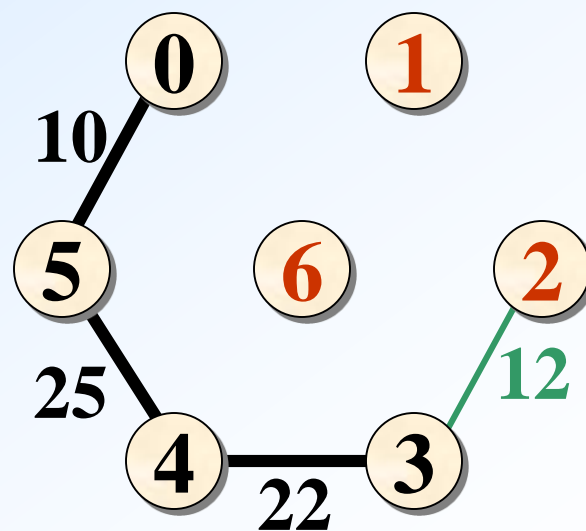
顶点 $v=3$ 加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	28	12	22	25	10	18
nearvex	-1	0	3	-1	-1	-1	3

选  $v=2$   选边 (3,2)



原图



边 (4,3,22) 加入生成树 57

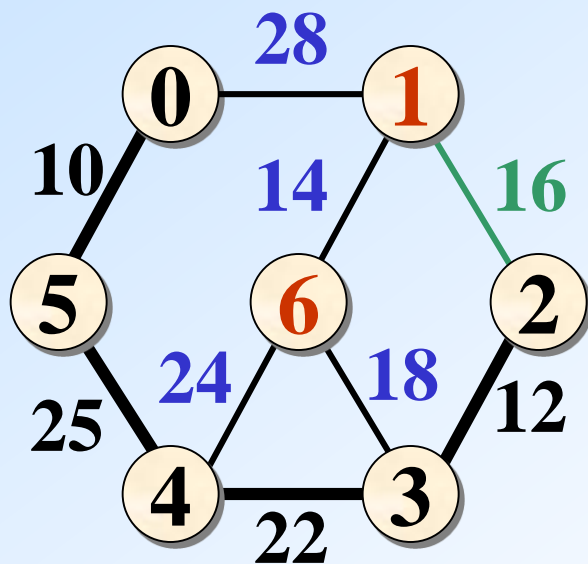


顶点 $v=2$ 加入生成树顶点集合:

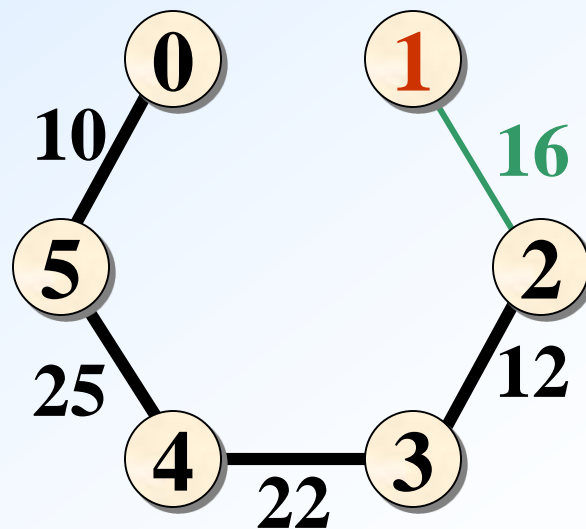
	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	18
nearvex	-1	2	-1	-1	-1	-1	3

选  $v=1$

选边 (2,1)



原图



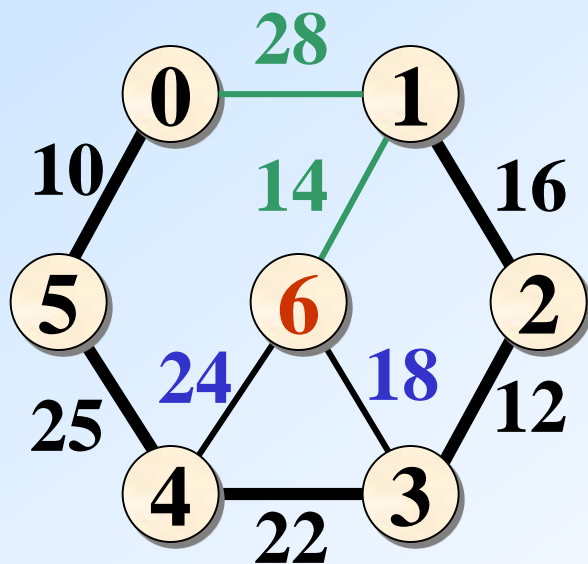
边 (3,2,12) 加入生成树



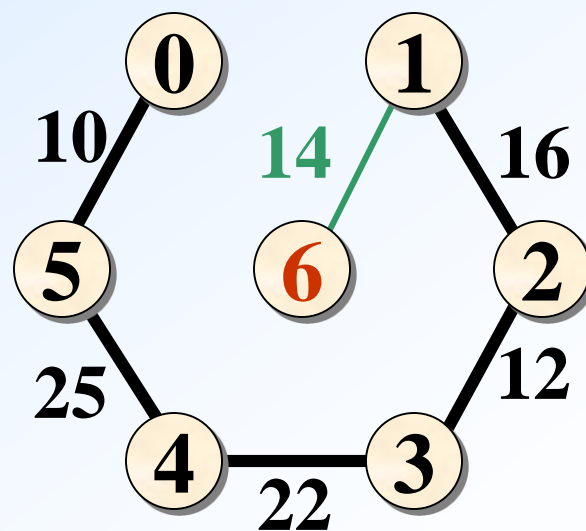
顶点  $v=1$  加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
nearvex	-1	-1	-1	-1	-1	-1	1

选  $v=6$  ↑ 选边 (1,6)



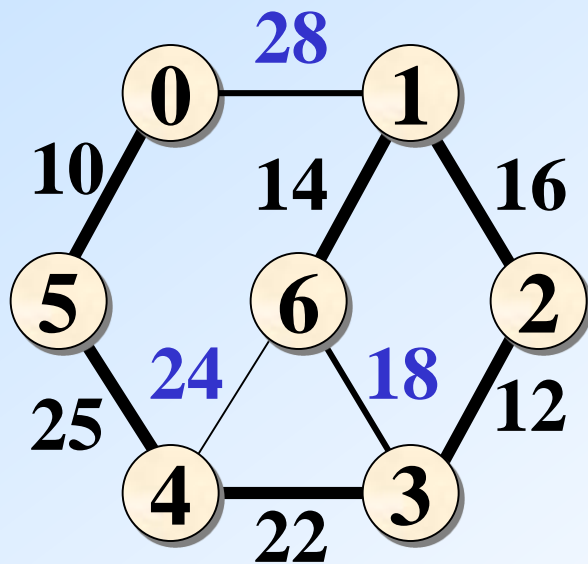
原图



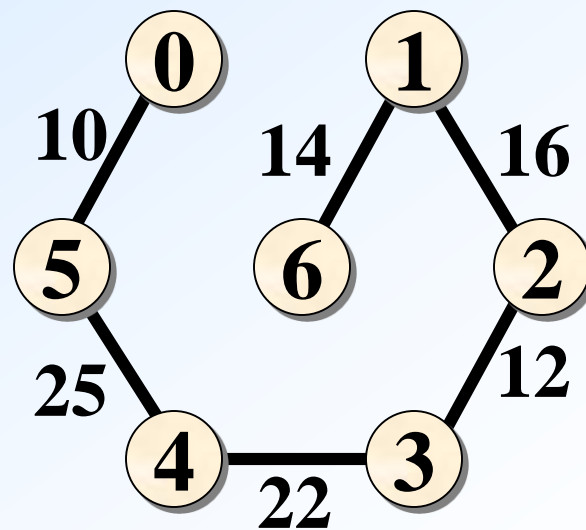
边 (2,1,16) 加入生成树<sub>9</sub>

顶点 $v=6$ 加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
nearvex	-1	-1	-1	-1	-1	-1	-1



原图



边 (1,6,14) 加入生成树

最后生成树中边集合里存入得各条边为:

(0, 5, 10), (5, 4, 25), (4, 3, 22),  
(3, 2, 12), (2, 1, 16), (1, 6, 14)

利用普里姆算法建立最小生成树

```
void Prim ( Graph G, MST& T, int u ) {  
    float * lowcost = new float[NumVertices];  
    int * nearvex = new int[NumVertices];  
    for ( int i = 0; i < NumVertices; i++ ) {  
        lowcost[i] = G.Edge[u][i]; //Vu到各点代价  
        nearvex[i] = u;           //及最短带权路径  
    }  
}
```

```

nearvex[u] = -1;    //加到生成树顶点集合
int k = 0;         //存放最小生成树结点的变量
for ( i = 0; i < G.n; i++ )
    if ( i != u ) { //循环n-1次, 加入n-1条边
        EdgeData min = MaxValue; int v = 0;
        for ( int j = 0; j < NumVertices; j++ )
            if ( nearvex[j] != -1 && //=-1不参选
                lowcost[j] < min )
                { v = j; min = lowcost[j]; }
        //求生成树外顶点到生成树内顶点具有最
        //小权值的边, v是当前具最小权值的边
    }

```

```

if ( v ) {    //v=0表示再也找不到要求顶点
    T[k].tail = nearvex[v]; //选边加入生成树
    T[k].head = v;
    T[k++].cost = lowcost[v];
    nearvex[v] = -1; //该边加入生成树标记
    for ( j = 0; j < G.n; j++ )
        if ( nearvex[j] != -1 &&
            G.Edge[v][j] < lowcost[j] ) {
            lowcost[j] = G.Edge[v][j];    //修改
            nearvex[j] = v;
        }
    }
}

```

```
} //循环n-1次, 加入n-1条边  
}
```

- 分析以上算法, 设连通网络有  $n$  个顶点, 则该算法的时间复杂度为  $O(n^2)$ , 它适用于边稠密的网络。
- 注意: 当各边有相同权值时, 由于选择的随意性, 产生的生成树可能不唯一。当各边的权值不相同, 产生的生成树是唯一的。

# 活动网络 (Activity Network)

## 用顶点表示活动的网络 (AOV网络)

- 计划、施工过程、生产流程、程序流程等都是“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。其中有些课程要求先修课程，有些则不要求。这样在有的课程之间有领先关系，有的课程可以并行地学习。

## 课程代号

## 课程名称

## 先修课程

C<sub>1</sub>

高等数学

C<sub>2</sub>

程序设计基础

C<sub>3</sub>

离散数学

C<sub>1</sub>, C<sub>2</sub>

C<sub>4</sub>

数据结构

C<sub>3</sub>, C<sub>2</sub>

C<sub>5</sub>

高级语言程序设计

C<sub>2</sub>

C<sub>6</sub>

编译方法

C<sub>5</sub>, C<sub>4</sub>

C<sub>7</sub>

操作系统

C<sub>4</sub>, C<sub>9</sub>

C<sub>8</sub>

普通物理

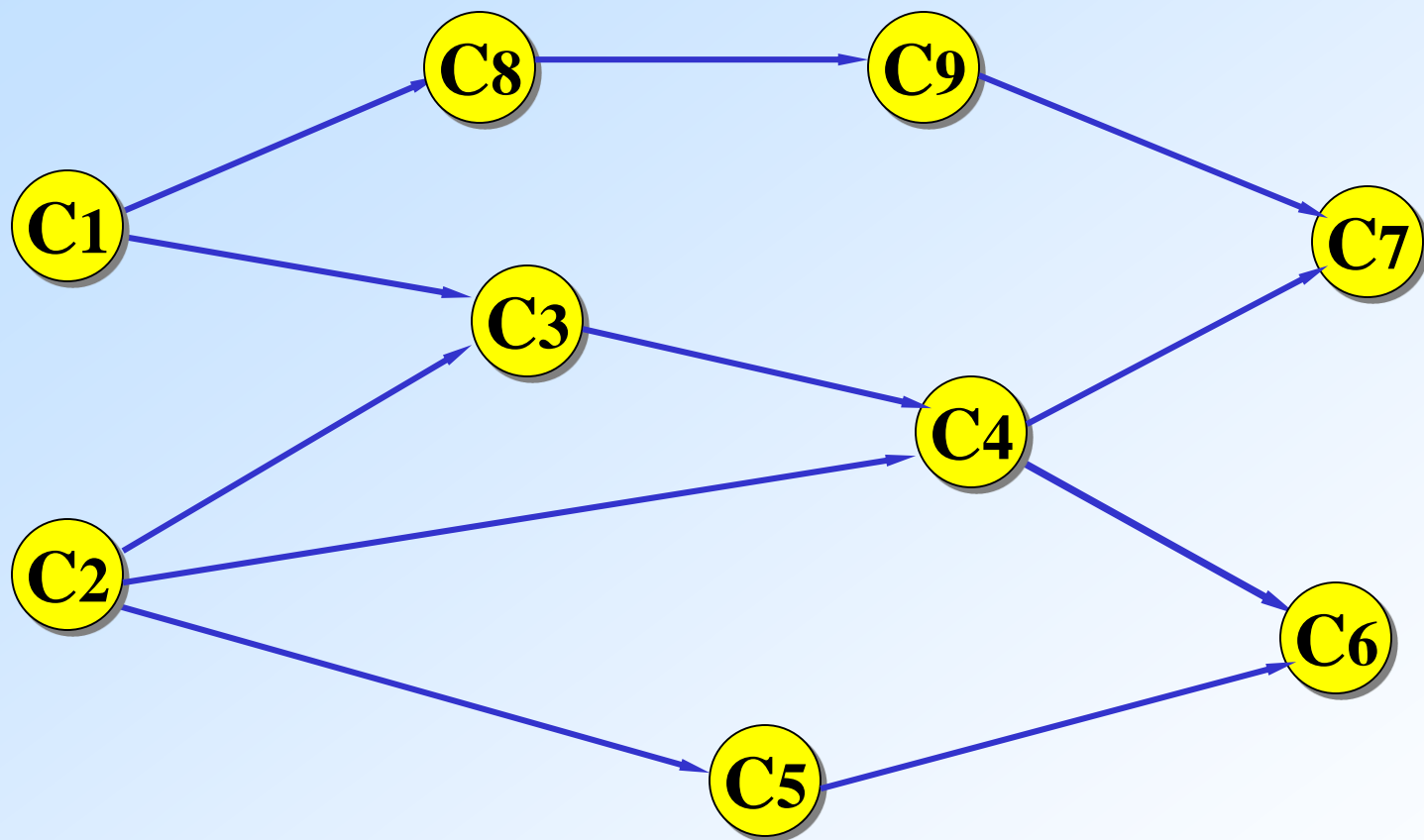
C<sub>1</sub>

C<sub>9</sub>

计算机原理

C<sub>8</sub>





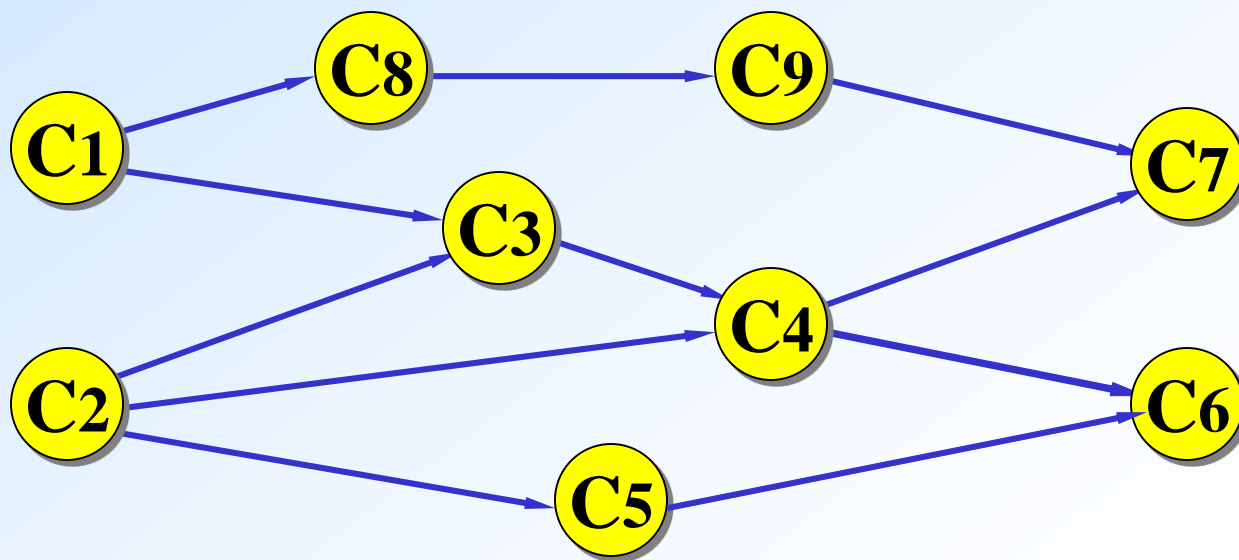
**学生课程学习工程图**

- 可以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 $\langle V_i, V_j \rangle$ 表示活动 $V_i$ 必须先于活动 $V_j$ 进行。这种有向图叫做**顶点表示活动的AOV网络** (Activity On Vertices)。
- 在AOV网络中不能出现有向回路，即有向环。如果出现了有向环，则意味着某项活动应以自己作为先决条件。
- 因此，**对给定的AOV网络，必须先判断它是否存在有向环。**

- 检测有向环的一种方法是对AOV网络构造它的拓扑有序序列。即将各个顶点 (代表各个活动)排列成一个线性有序的序列, 使得AOV网络中所有应存在的前驱和后继关系都能得到满足。
- 这种构造AOV网络全部顶点的拓扑有序序列的运算就叫做拓扑排序。
- 如果通过拓扑排序能将AOV网络的所有顶点都排入一个拓扑有序的序列中, 则该网络中必定不会出现有向环。

- 如果AOV网络中存在有向环，此AOV网络所代表的工程是不可行的。
- 例如，对学生选课工程图进行拓扑排序，得到的拓扑有序序列为

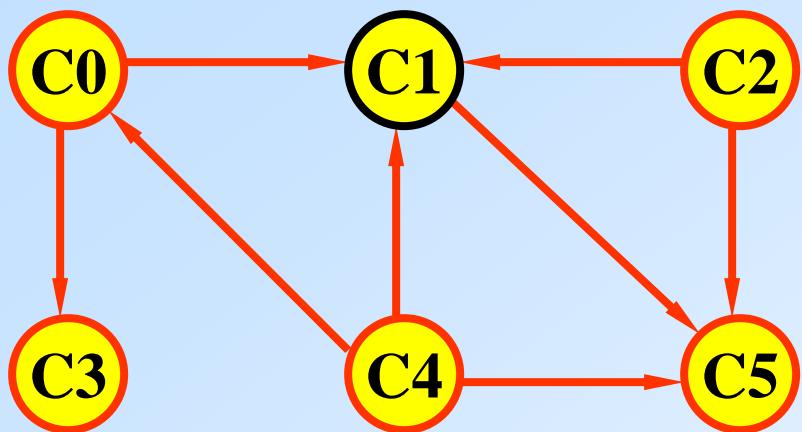
$C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$   
或  $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$



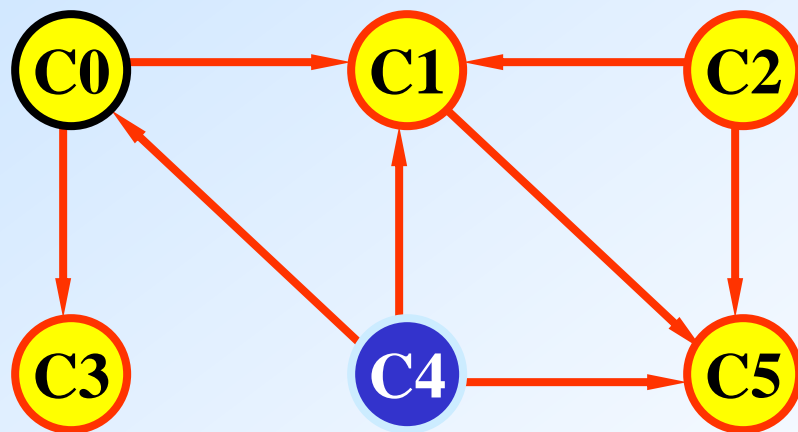
# 拓扑排序的方法

- ① 输入AOV网络。令  $n$  为顶点个数。
- ② 在AOV网络中选一个没有直接前驱的顶点，并输出之；
- ③ 从图中删去该顶点，同时删去所有它发出的有向边；
- ④ 重复以上 ②、③步，直到
  - 全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；或
  - 图中还有未输出的顶点，但已跳出处理循环。说明图中还剩下一些顶点，它们都有直接前驱。这时网络中必存在有向环。

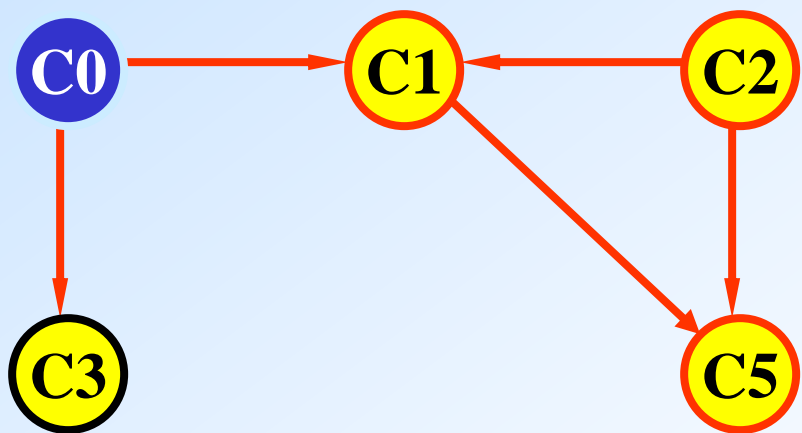
# 拓扑排序的过程



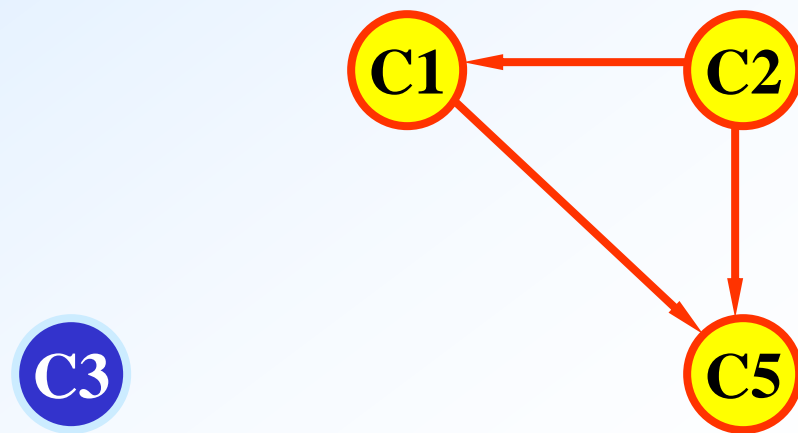
(a) 有向无环图



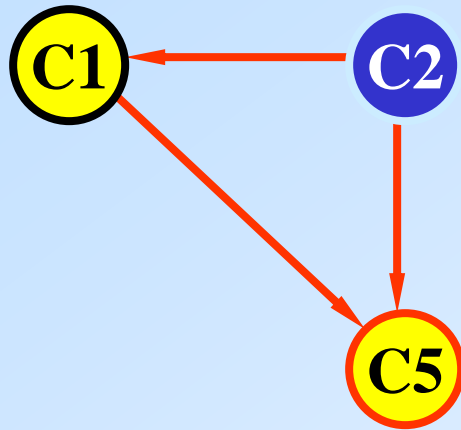
(b) 输出顶点C4



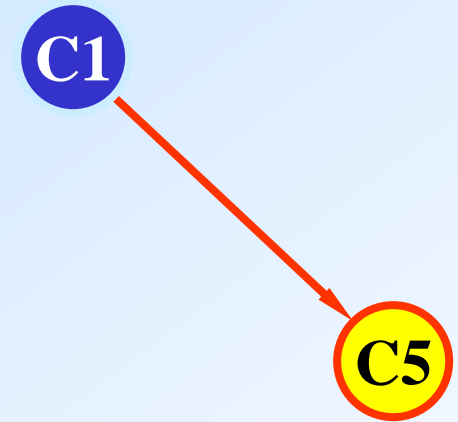
(c) 输出顶点C0



(d) 输出顶点C3



(e) 输出顶点C2



(f) 输出顶点C1

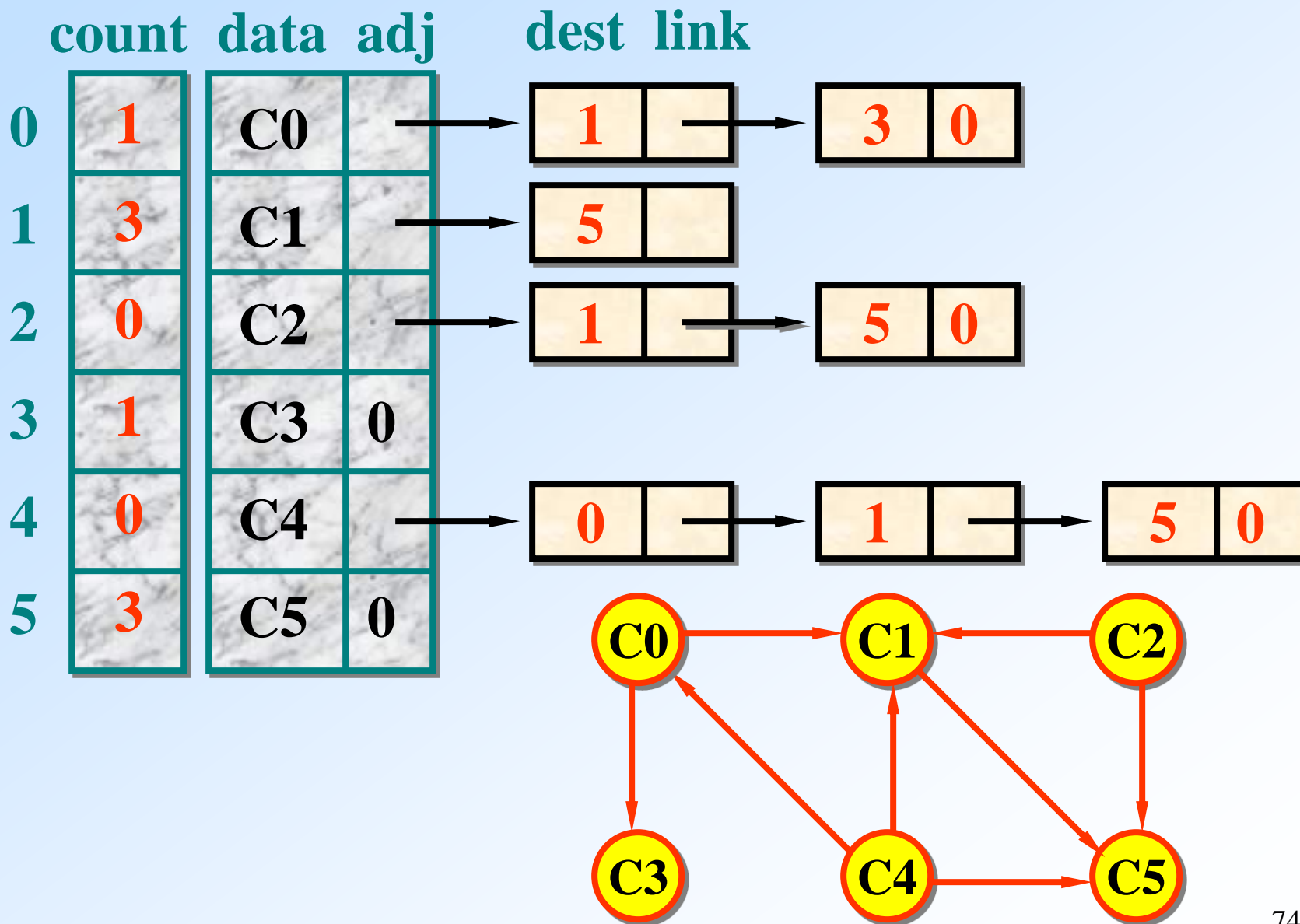


(g) 输出顶点C5

(h) 拓扑排序完成

最后得到的拓扑有序序列为  $C_4, C_0, C_3, C_2, C_1, C_5$ 。它满足图中给出的所有前驱和后继关系，对于本来没有这种关系的顶点，如  $C_4$  和  $C_2$ ，也排出了先后次序关系。

# AOV网络及其邻接表表示





- 在邻接表中增设一个数组count[ ], 记录各顶点入度。入度为零的顶点即无前驱顶点。
- 在输入数据前, 顶点表VexList[ ]和入度数组count[ ]全部初始化。在输入数据时, 每输入一条边<j, k>, 就需要建立一个边结点, 并将它链入相应边链表中, 统计入度信息:

```
EdgeNode * p = new EdgeNode;
```

```
p->dest = k;           //建立边结点
```

```
p->link = G.VexList[j].firstAdj;
```

```
VexList[j].firstAdj = p;
```

```
    //链入顶点 j 的边链表的前端
```

```
count[k]++;           //顶点 k 入度加一
```

- 在算法中, 使用一个存放入度为零的顶点的链式栈, 供选择和输出无前驱的顶点。
- 拓扑排序算法可描述如下:
  - ◆ 建立入度为零的顶点栈;
  - ◆ 当入度为零的顶点栈不空时, 重复执行
    - ✿ 从顶点栈中退出一个顶点, 并输出之;
    - ✿ 从AOV网络中删去这个顶点和它发出的边, 边的终顶点入度减一;
    - ✿ 如果边的终顶点入度减至0, 则该顶点进入度为零的顶点栈;
  - ◆ 如果输出顶点个数少于AOV网络的顶点个数, 则报告网络中存在有向环。

# 拓扑排序的算法

```
void TopologicalSort (AdjGraph G) {  
    Stack S; StackEmpty(S); int j;  
        //入度为零的顶点栈初始化  
    for ( int i = 0; i < n; i++ )    //入度为零顶点  
        if ( count[i] == 0 ) Push(S, i);    //进栈  
    for ( i = 0; i < n; i++ )    //期望输出 n 个顶点  
        if ( StackEmpty(S) ) {    //中途栈空,转出  
            cout << “网络中有回路! ” << endl;  
            return;  
        }  
}
```

```

else {                                     //继续拓扑排序
    Pop( S, j );                          //退栈
    cout << j << endl;                    //输出
    EdgeNode * p = VexList[j].firstAdj;
    while ( p != NULL ) {                //扫描出边表
        int k = p->dest;                  //另一顶点
        if ( --count[k] == 0 )           //顶点入度减一
            Push( S, k );
        //顶点的入度减至零, 进栈
        p = p->link;
    }
}
}
}

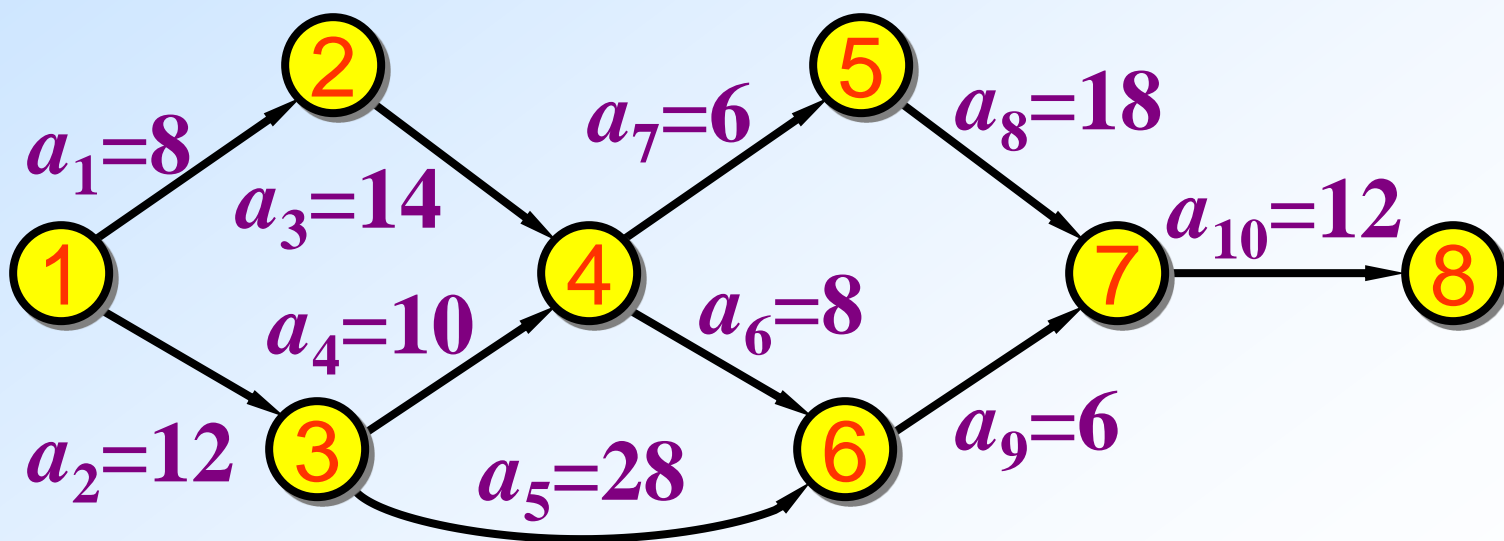
```

# 用边表示活动的网络(AOE网络)

- 如果在无有向环的带权有向图中,用有向边表示一个工程中的活动 (Activity),用边上权值表示活动持续时间 (Duration),用顶点表示事件 (Event),则这样的有向图叫做用边表示活动的网络,简称 AOE ( Activity On Edges ) 网络。
- AOE网络在某些工程估算方面非常有用。例如,可以使人们了解:
  - ◆ 完成整个工程至少需要多少时间(假设网络中没有环)?

- ◆ 为缩短完成工程所需的时间,应当加快哪些活动?
- 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(Critical Path)。

- 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。例如，下图就是一个AOE网。





## 定义几个与计算关键活动有关的量:

- ① 事件 $V_i$ 的最早可能开始时间 $Ve(i)$   
是从源点 $V_0$ 到顶点 $V_i$ 的最长路径长度。
- ② 事件 $V_i$ 的最迟允许开始时间 $VI[i]$   
是在保证汇点 $V_{n-1}$ 在 $Ve[n-1]$ 时刻完成的前提下, 事件 $V_i$ 的允许的最迟开始时间。
- ③ 活动 $a_k$ 的最早可能开始时间  $e[k]$   
设活动 $a_k$ 在边 $\langle V_i, V_j \rangle$ 上, 则 $e[k]$ 是从源点 $V_0$ 到顶点 $V_i$ 的最长路径长度。因此,  
$$e[k] = Ve[i].$$
- ④ 活动 $a_k$ 的最迟允许开始时间  $l[k]$



$l[k]$ 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。

$$l[k] = vl[j] - \text{dur}(\langle i, j \rangle)。$$

其中,  $\text{dur}(\langle i, j \rangle)$ 是完成  $a_k$  所需的时间。

### ⑤ 时间余量 $l[k] - e[k]$

表示活动  $a_k$  的最早可能开始时间和最迟允许开始时间的的时间余量。 $l[k] == e[k]$  表示活动  $a_k$  是没有时间余量的关键活动。

- 为找出关键活动, 要求各活动的  $e[k]$  与  $l[k]$ , 以判别是否  $l[k] == e[k]$ 。

- 为求得  $e[k]$  与  $l[k]$ , 需要先求得从源点  $V_0$  到各个顶点  $V_i$  的  $Ve[i]$  和  $VI[i]$ 。

- 求  $Ve[i]$  的递推公式

- ◆ 从  $Ve[0] = 0$  开始, 向前递推

$$Ve[i] = \max_j \{ Ve[j] + dur(<V_j, V_i>) \},$$
$$<V_j, V_i> \in S2, i = 1, 2, \dots, n-1$$

$S2$  是所有指向  $V_i$  的有向边  $<V_j, V_i>$  的集合。

- ◆ 从  $VI[n-1] = Ve[n-1]$  开始, 反向递推

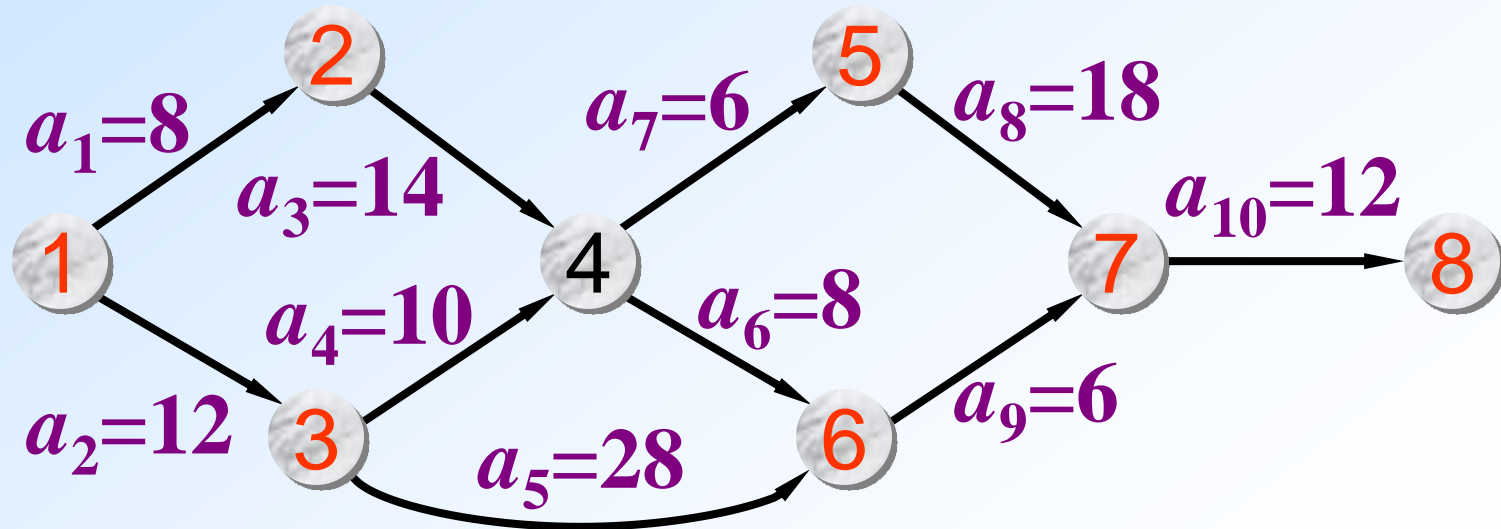
$$VI[i] = \min_j \{ VI[j] - dur(<V_i, V_j>) \},$$
$$<V_i, V_j> \in S1, i = n-2, n-3, \dots, 0$$

$S1$  是所有源自  $V_i$  的有向边  $<V_i, V_j>$  的集合。

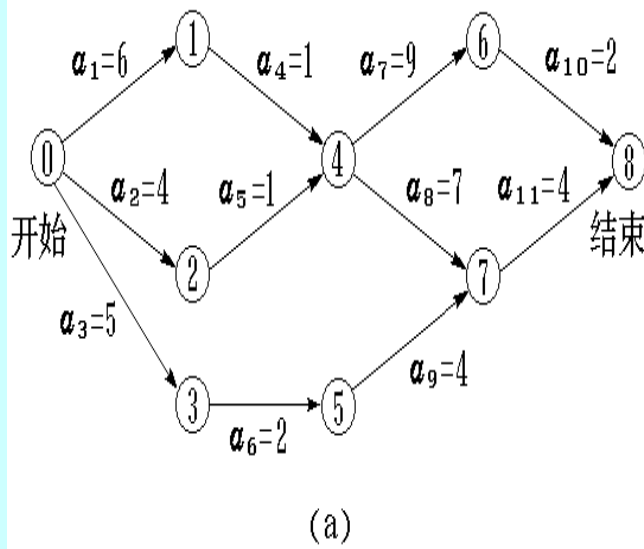
- 这两个递推公式的计算必须分别在拓扑有序及逆拓扑有序的前提下进行。
- 设活动 $a_k$  ( $k=1,2,\dots,e$ )在带权有向边 $\langle V_i, V_j \rangle$ 上, 其持续时间用 $\text{dur}(\langle V_i, V_j \rangle)$ 表示, 则有
$$e[k] = Ve[i];$$
$$l[k] = Vl[j] - \text{dur}(\langle V_i, V_j \rangle); \quad k = 1, 2, \dots, e。$$
这样就得到计算关键路径的算法。
- ⑩ 为了简化算法, 假定在求关键路径之前已经对各顶点实现了拓扑排序, 并按拓扑有序的顺序对各顶点重新进行了编号。

	1	2	3	4	5	6	7	8
$Ve$	0	8	12	22	28	40	46	58
$VI$	0	8	12	22	28	40	46	58

	1	2	3	4	5	6	7	8	9	10
$e$	0	0	8	12	12	22	22	28	40	46
$l$	0	0	8	12	12	32	22	28	40	46



事件	Ve[i]	VL[i]
V0	0	0
V1	6	6
V2	4	6
V3	5	8
V4	7	7
V5	7	10
V6	16	16
V7	14	14
V8	18	18



NodeTable

	count	adj	dest	dur	link
0	0		1	6	→ 2 4 → 3 5 ^
1	1		4	1	^
2	1		4	1	^
3	1		5	2	^
4	2		6	9	→ 7 7 ^
5	1		7	4	^
6	1		8	2	^
7	2		8	4	^
8	2				^

(b)

边	<0,1><0,2><0,3><1,4><2,4><3,5><4,6><4,7><5,7><6,8><7,8>										
活动	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$	$a_7$	$a_8$	$a_9$	$a_{10}$	$a_{11}$
$e$	0	0	0	6	4	5	7	7	7	16	14
$l$	0	2	3	6	6	8	7	7	10	16	14
$l-e$	0	2	3	0	2	3	0	0	3	0	0
关键	是			是			是	是		是	是

# 利用关键路径法求AOE网的各关键活动

```
void graph::CriticalPath ( ) {
```

```
//在此算法中需要对邻接表中单链表的结点加以  
//修改,在各结点中增加一个int域 cost,记录该结  
//点所表示的边上的权值。
```

```
    int i, j; int p, k; int e, l;
```

```
    Ve = new int[n]; Vl = new int[n];
```

```
    for ( i = 0; i < n; i++ ) Ve[i] = 0; //初始化
```

```
    for ( i = 0; i < n; i++ ) { //顺向计算事件最早允许开始时间
```

```
        Edge<int> *p = NodeTable[i].adj; //该顶点边链表链头指针p
```

```
        while ( p != NULL ) { //找所有后继邻接顶点
```

```
            k = p->dest; //i的后继邻接顶点k
```

```
            if ( Ve[i] + p->cost > Ve[k] )
```

```
                Ve[k] = Ve[i] + p->cost;
```

```
                p = p->link; //找下一个后继
```

```
    }
```

```

}
for (  $i = 0; i < n; i++$  )  $Vl[i] = Ve[n-1]$ ; //逆向计算事件
                                         的最迟开始时间
for (  $i = n-2; i; i--$  ) { //按逆拓扑有序顺序处理
     $p = NodeTable[i].adj$ ; //该顶点边链表链头指针p
    while (  $p \neq NULL$  ) {
         $k = p \rightarrow dest$ ; //i的后继邻接顶点k
        if (  $Vl[k] - p \rightarrow cost < Vl[i]$  )
             $Vl[i] = Vl[k] - p \rightarrow cost$ ;
         $p = p \rightarrow link$ ; //找下一个后继
    }
}

```

```

for ( i = 0; i < n; i++ ) { //逐个顶点求各活动的e[k]和l[k]
    p = NodeTable[i].adj; //该顶点边链表链头指针p
    while ( p != NULL ) {
        k = p→dest; //k是i的后继邻接顶点
        e = Ve[i]; l = Vl[k] - p→cost;
        if ( l == e ) //关键活动
            cout << "<" << i << ", " << k
                << ">" << "是关键活动" << endl;
        p = p→link; //找下一个后继
    }
}
}

```



# 注意

在拓扑排序求 $Ve[i]$ 和逆拓扑有序求 $VI[i]$ 时,所需为 $O(n+e)$ , 求各个活动的 $e[k]$ 和 $l[k]$ 时所需时间为 $O(e)$ , 总共花费时间仍然是 $O(n+e)$ 。

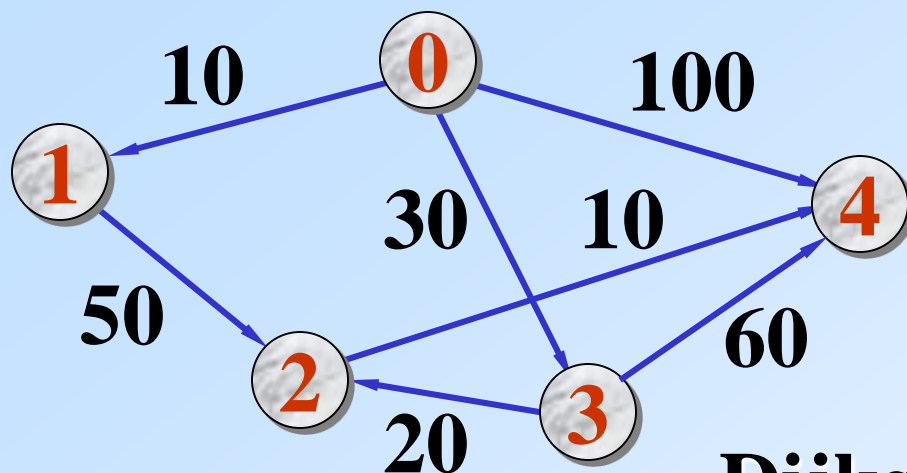
- 所有顶点按拓扑有序的次序编号
- 仅计算  $Ve[i]$  和  $VI[i]$  是不够的, 还须计算  $e[k]$  和  $l[k]$ 。
- 不是任一关键活动加速一定能使整个工程提前。
- 想使整个工程提前, 要考虑各个关键路径上所有关键活动。

# 最短路径 (Shortest Path)

- **最短路径问题：** 如果从图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小。
- **问题解法**
  - ◆ 边上权值非负情形的单源最短路径问题  
— Dijkstra算法
  - ◆ 边上权值为任意值的单源最短路径问题  
— Bellman和Ford算法
  - ◆ 所有顶点之间的最短路径  
— Floyd算法

# 边上权值非负情形的单源最短路径问题

- **问题的提法：** 给定一个带权有向图 $D$ 与源点  $v$ ，求从  $v$  到 $D$ 中其它顶点的最短路径。限定各边上的权值大于或等于0。
- 为求得这些最短路径, **Dijkstra**提出按路径长度的递增次序，逐步产生最短路径的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从顶点 $v$ 到其它各顶点的最短路径全部求出为止。
- **举例说明**



## Dijkstra逐步求解的过程

源点	终点	最短路径	路径长度
$v_0$	$v_1$	$(v_0, v_1)$	10
	$v_2$	— $(v_0, v_1, v_2)$ $(v_0, v_3, v_2)$	$\infty, 60, 50$
	$v_3$	$(v_0, v_3)$	30
	$v_4$	$(v_0, v_4)$ $(v_0, v_3, v_4)$ $(v_0, v_3, v_2, v_4)$	100, 90, 60

- 引入辅助数组 $\text{dist}$ 。它的每一个分量 $\text{dist}[i]$ 表示当前找到的从源点  $v_0$  到终点  $v_i$  的最短路径的长度。初始状态：
  - 若从源点  $v_0$  到顶点  $v_i$  有边, 则 $\text{dist}[i]$ 为该边上的权值;
  - 若从源点  $v_0$  到顶点  $v_i$  无边, 则 $\text{dist}[i]$ 为 $\infty$ 。
- 假设  $S$  是已求得的最短路径的终点的集合, 则可证明: 下一条最短路径必然是从 $v_0$ 出发, 中间只经过  $S$  中的顶点便可到达的那些顶点  $v_x$  ( $v_x \in V-S$ ) 的路径中的一条。
- 每次求得一条最短路径后, 其终点  $v_k$  加入集合  $S$ , 然后对所有的  $v_i \in V-S$ , 修改其  $\text{dist}[i]$  值。

# Dijkstra算法

① 初始化:  $S \leftarrow \{ v_0 \};$

$\text{dist}[j] \leftarrow \text{Edge}[0][j], \quad j = 1, 2, \dots, n-1;$   
//  $n$ 为图中顶点个数

② 求出最短路径的长度:

$\text{dist}[k] \leftarrow \min \{ \text{dist}[i] \}, \quad i \in V - S;$

$S \leftarrow S \cup \{ k \};$

③ 修改:

$\text{dist}[i] \leftarrow \min \{ \text{dist}[i], \text{dist}[k] + \text{Edge}[k][i] \},$   
对于每一个  $i \in V - S;$

④ 判断: 若  $S = V$ , 则算法结束, 否则转 ②。

# 计算从单个顶点到其它各顶点最短路径

```
void ShortestPath (MTGraph G, int v ){  
    //MTGraph是一个有 n 个顶点的带权有向图,各边上的  
    权值由Edge[i][j]给出。  
    //dist[j],  $0 \leq j < n$ , 是当前求到的从顶点v 到顶点 j 的最短  
    路径长度, 用数组path[j],  $0 \leq j < n$ , 存放求到的最短路径。  
    EdgeData dist[G.n];    //最短路径长度数组  
    int path[G.n];          //最短路径数组  
    int S[G.n];             //最短路径顶点集
```



```

for ( int i = 0; i < n; i++) {
    dist[i] = G.Edge[v][i];    //dist数组初始化
    S[i] = 0;                  //集合S初始化
    if ( i != v && dist[i] < MaxValue )
        path[i] = v;
    else path[i] = -1;          //path数组初始化
}
S[v] = 1;  dist[v] = 0;    //顶点v加入顶点集合
for ( i = 0; i < n-1; i++ ) {#//从顶点v确定n-1条路径
    float min = MaxValue;  int u = v;

```



```

for ( int j = 0; j < n; j++ ) //选当前不在集合S中具有最
    if ( !S[j] && dist[j] < min ) //最短路径的顶点u
        { u = j; min = dist[j]; }
S[u] = 1; //将顶点u加入集合S, 表示它已在最短路径上
for ( int w = 0; w < n; w++ ) //修改
    if (!S[w] && G.Edge[u][w] < MaxValue
        && dist[u] + G.Edge[u][w] < dist[w] ) {
        //顶点w未加入S, 且绕过u可以缩短路径
        dist[w] = dist[u] + G.Edge[u][w];
        path[w] = u; //修改到w的最短路径
    }
}# //选定各顶点到顶点 v 的最短路径
}

```

```

//打印各顶点的最短路径: 路径是逆向输出的
for ( i = 0; i < G.n; i++ ) {
    cout << endl;
    cout << "Distance: " << dist[i]
        << " Path: " << i;
    //输出终点的最短路径长度和终点
    int pre = path[i];           //取终点的直接前驱
    while ( pre != v ) {        //沿路径上溯输出
        cout << ", " << pre;
        pre = path[pre];
    }
}

```