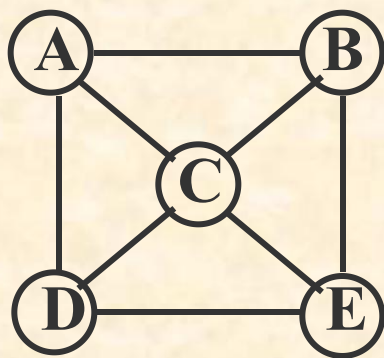


第七章 图



图是一种较线性表和树更为复杂的数据结构。

线性表： 线性结构

树： 层次结构

图： 结点之间的关系可以是任意的，即图中任意两个数据元素之间都可能相关。

图的结构定义:

图是由一个顶点集 V 和一个弧集 VR (顶点间的关系集合)构成的数据结构。

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

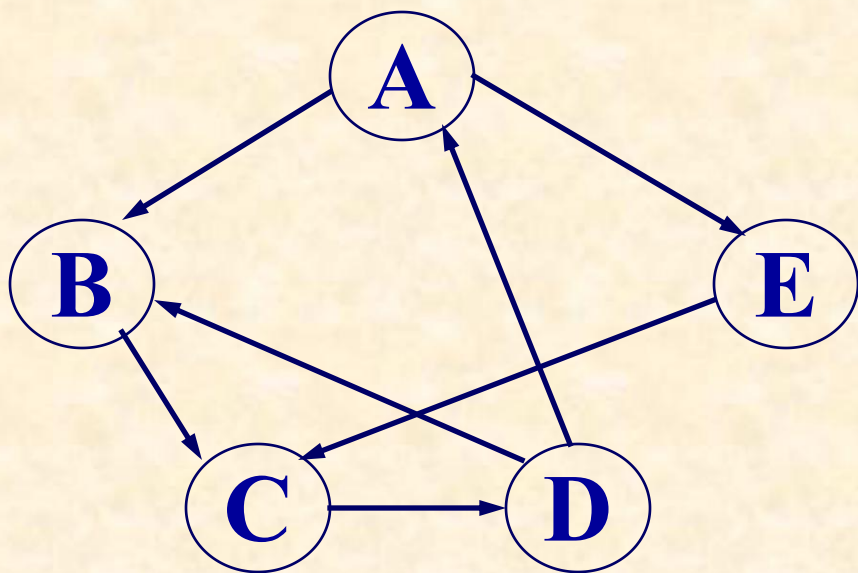
其中, $VR = \{ \langle v, w \rangle \mid v, w \in V \text{ 且 } P(v, w) \}$

$\langle v, w \rangle$ 表示从 v 到 w 的一条弧, 并称 v 为弧尾, w 为弧头。

谓词 $P(v, w)$ 定义了弧 $\langle v, w \rangle$ 的意义或信息。

由于“**弧**”是有方向的，因此称由顶点集和弧集构成的图为**有向图**。

例如： $G_1 = (V_1, VR_1)$



其中

$V_1 = \{A, B, C, D, E\}$

$VR_1 = \{ \langle A, B \rangle, \langle A, E \rangle, \langle B, C \rangle, \langle C, D \rangle, \langle D, B \rangle, \langle D, A \rangle, \langle E, C \rangle \}$

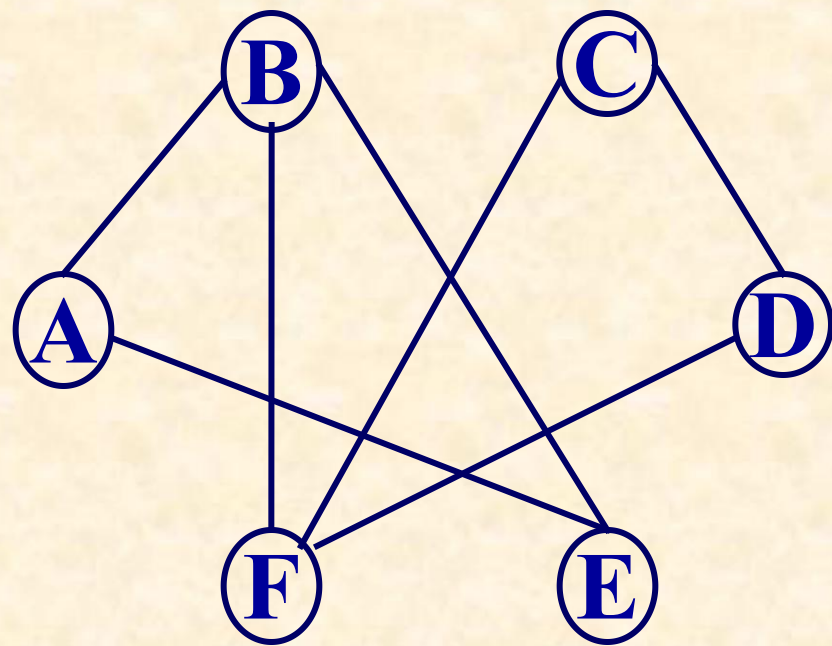
若 $\langle v, w \rangle \in VR$ 必有 $\langle w, v \rangle \in VR$,
则称 (v, w) 为顶点 v 和顶点
 w 之间存在一条**边**。

由顶点集和边
集构成的图称
作**无向图**。

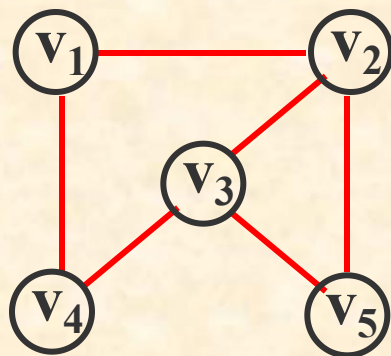
例如: $G_2 = (V_2, VR_2)$

$V_2 = \{A, B, C, D, E, F\}$

$VR_2 = \{(A, B), (A, E),$
 $(B, E), (C, D), (D, F),$
 $(B, F), (C, F) \}$



无向图：边是顶点的无序对，即边没有方向性。

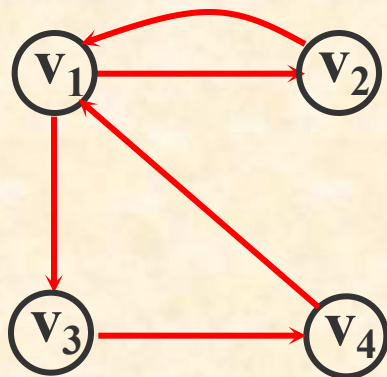


$$V = \{ v_1, v_2, v_3, v_4, v_5 \}$$

$$E = \{ (v_1, v_2), (v_1, v_4), (v_2, v_3), (v_2, v_5), (v_3, v_4), (v_3, v_5) \}$$

(v_1, v_2) 表示顶点 v_1 和 v_2 之间的边， $(v_1, v_2) = (v_2, v_1)$ 。

有向图：其边是顶点的有序对，即边有方向性。



$$V = \{ v_1, v_2, v_3, v_4 \}$$

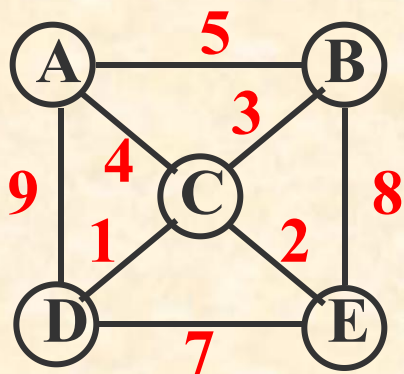
$$E = \{ \langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \langle v_3, v_4 \rangle, \langle v_4, v_1 \rangle \}$$

通常边称为**弧**， $\langle v_1, v_2 \rangle$ 表示顶点 v_1 到 v_2 的弧。

称 v_1 为弧尾，称 v_2 为弧头。

$$\langle v_1, v_2 \rangle \neq \langle v_2, v_1 \rangle$$

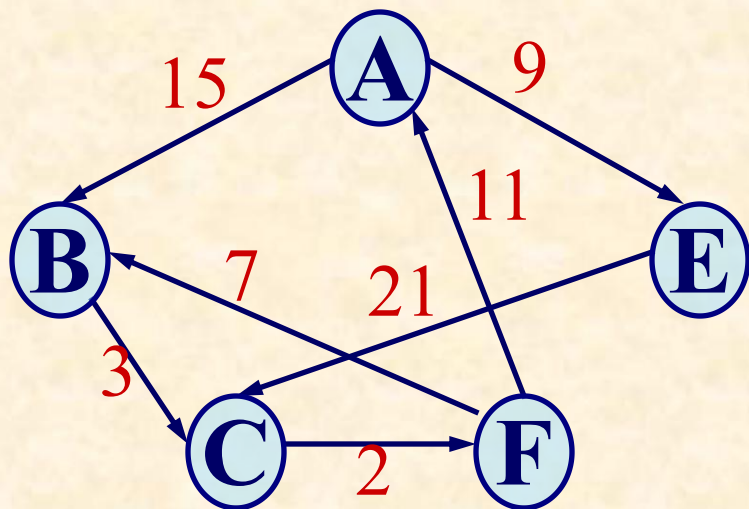
有时对图的**边或弧**赋予相关的数值，这种与图的边或弧相关的数值叫做**权**。



这些**权**可以表示从一个顶点到另一个顶点的距离。

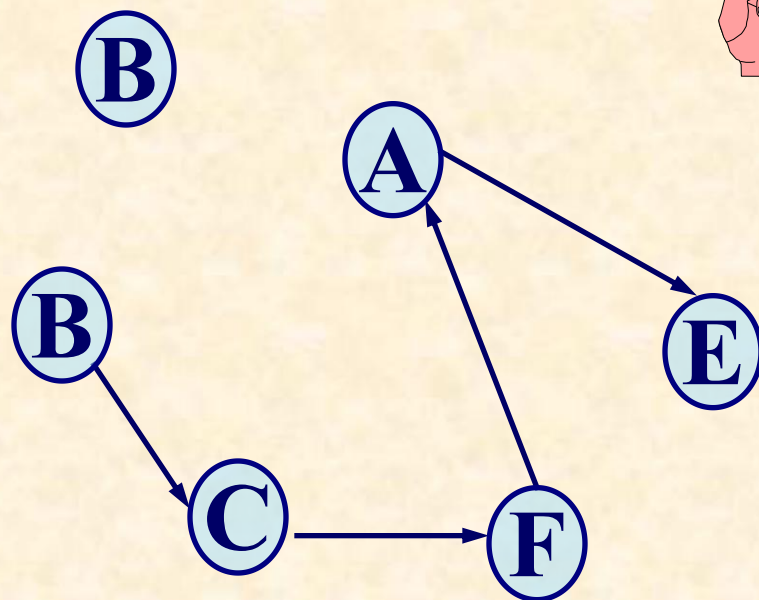
可以表示从一个顶点到另一个顶点的耗费。

这种带权的图通常称为**网**。



弧或边带权的图
分别称作**有向网**或
无向网。

设图 $G=(V, \{VR\})$ 和
图 $G'=(V', \{VR'\})$,
且 $V' \subseteq V, VR' \subseteq VR$,
则称 G' 为 G 的**子图**。



性质: 若用 **n** 表示图中顶点数目,
用 **e** 表示边或弧的数目, 若在图
中不存在顶点到自身的边或弧, 则

对于有向图, $0 \leq \mathbf{e} \leq n(n-1)$

对于无向图, $0 \leq \mathbf{e} \leq \frac{1}{2} n(n-1)$

假设图中有 n 个顶点， e 条边，则

含有 $e = n(n-1)/2$ 条边的无向图称作
完全图；

含有 $e = n(n-1)$ 条弧的有向图称作
有向完全图；

若边或弧的个数 $e < n \log n$ ，则称作
稀疏图，否则称作稠密图。



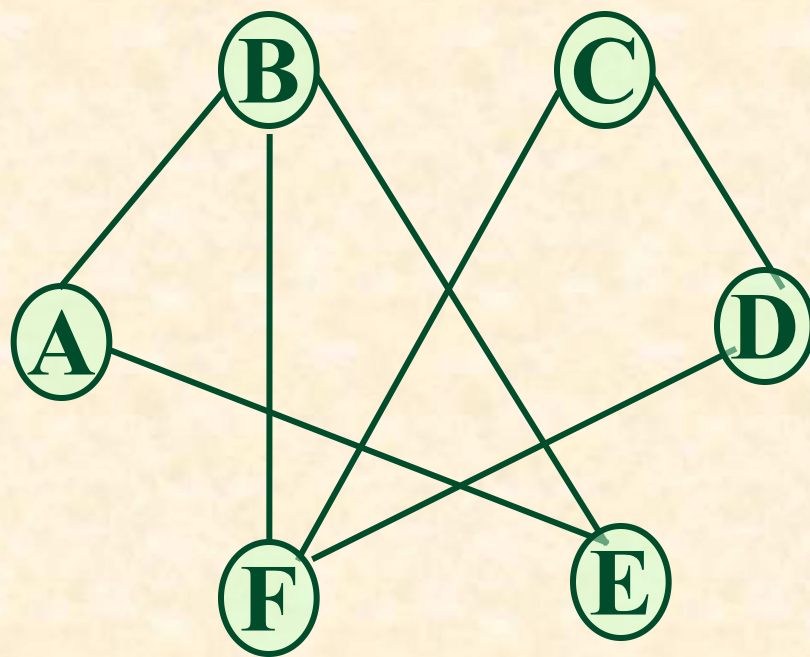
假若顶点**v**和顶点**w**之间存在一条边，
则称顶点**v**和**w**互为**邻接点**，

边(v,w)和顶点**v**和**w**相关联即**依附于**顶点**v**和**w**。和顶点**v**关联的**边的数目**定义为顶点的**度**。

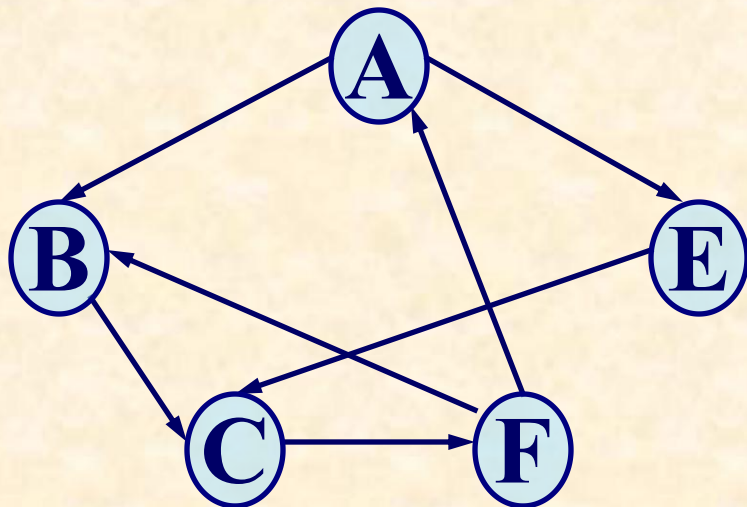
例如：右侧图中

$$TD(B) = 3$$

$$TD(A) = 2$$



对有向图来说，由于弧有方向性，则有入度和出度之分



例如：

$$OD(B) = 1$$

$$ID(B) = 2$$

$$TD(B) = 3$$

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。

顶点的度 $(TD) =$
出度 $(OD) +$ 入度 (ID)

设图 $G=(V, \{VR\})$ 中的一个顶点序列



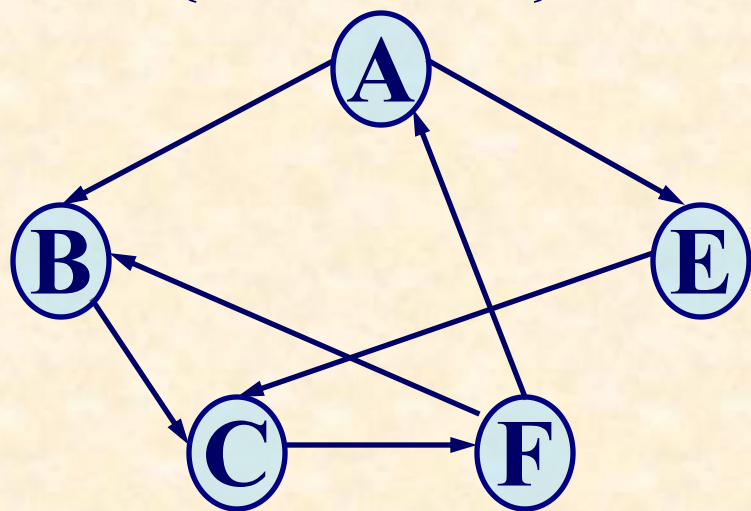
$\{u=v_{i,0}, v_{i,1}, \dots, v_{i,m}=w\}$ 中, $(v_{i,j-1}, v_{i,j}) \in VR \ 1 \leq j \leq m$,

则称从顶点 u 到顶点 w 之间存在一条**路径**。

路径上边的数目称作**路径长度**。

如:从A到F长度为 3

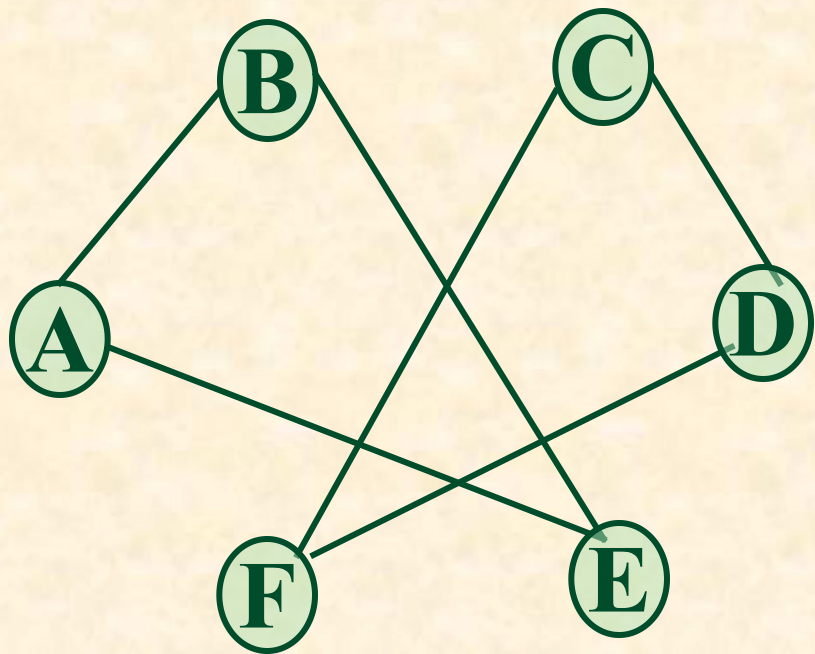
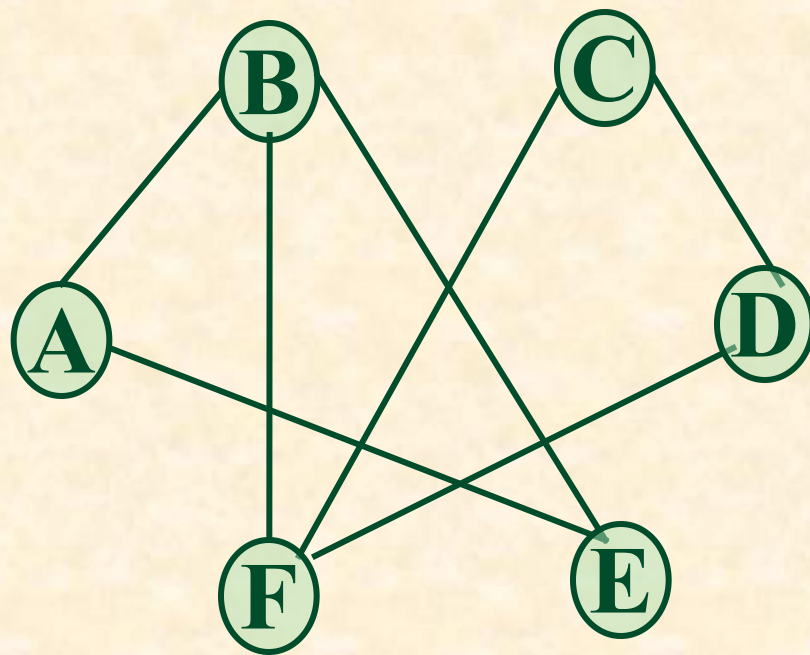
的路径 $\{A, B, C, F\}$



简单路径:指序列中顶点不重复出现的路径。

简单回路:指序列中第一个顶点和最后一个顶点相同的路径。

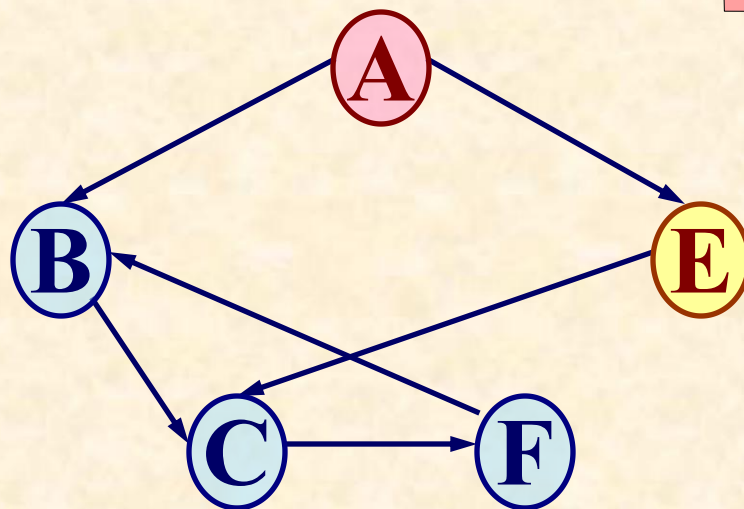
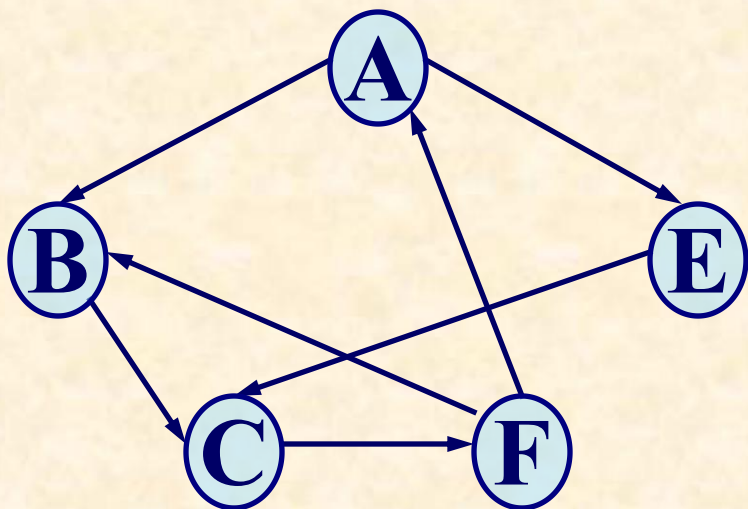
若图G中任意两个顶点之间都有路径相通，
则称此图为**连通图**；



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**

【P159图7.3】。

对有向图，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**【P159 图7.4】。

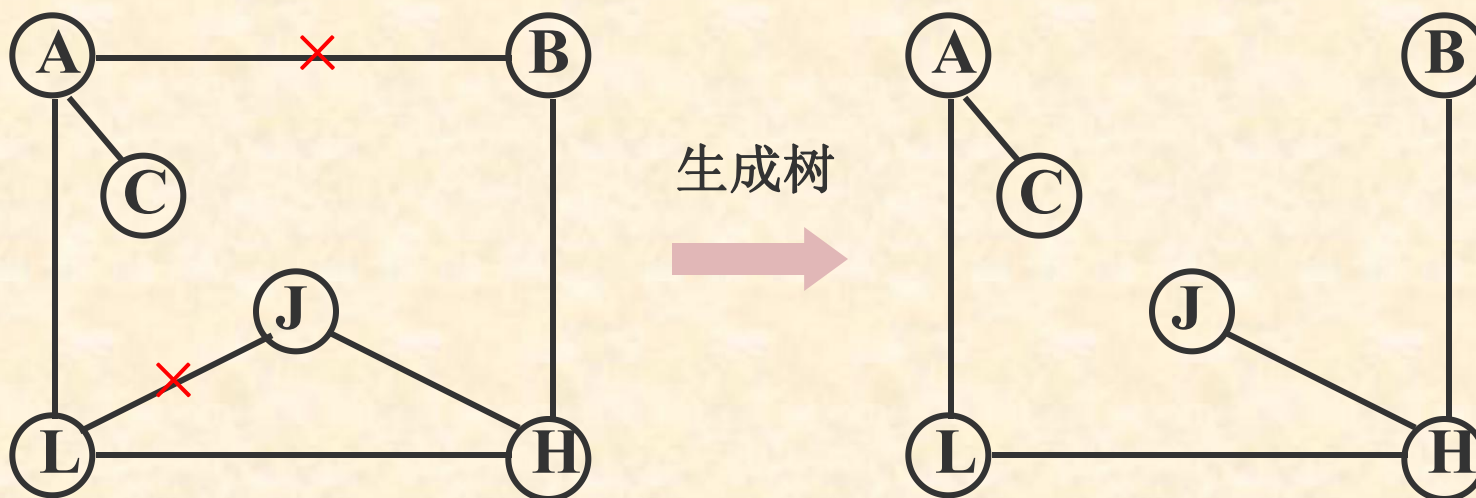


生成树、生成森林

一个连通图 G 的一个包含所有顶点的极小连通子图 T 是

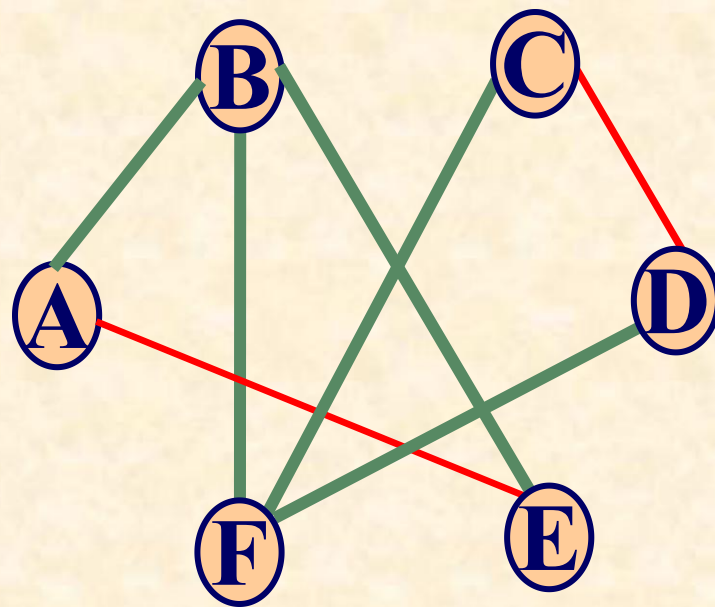
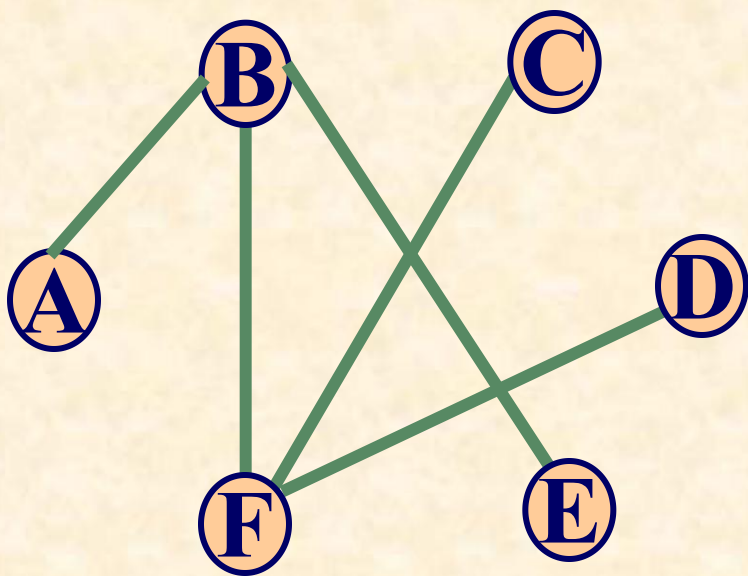
- (1) T 包含 G 的所有顶点 n 个
- (2) T 为连通子图
- (3) T 包含的边数最少

是一棵有 n 个顶点， $n-1$ 条边的生成树。

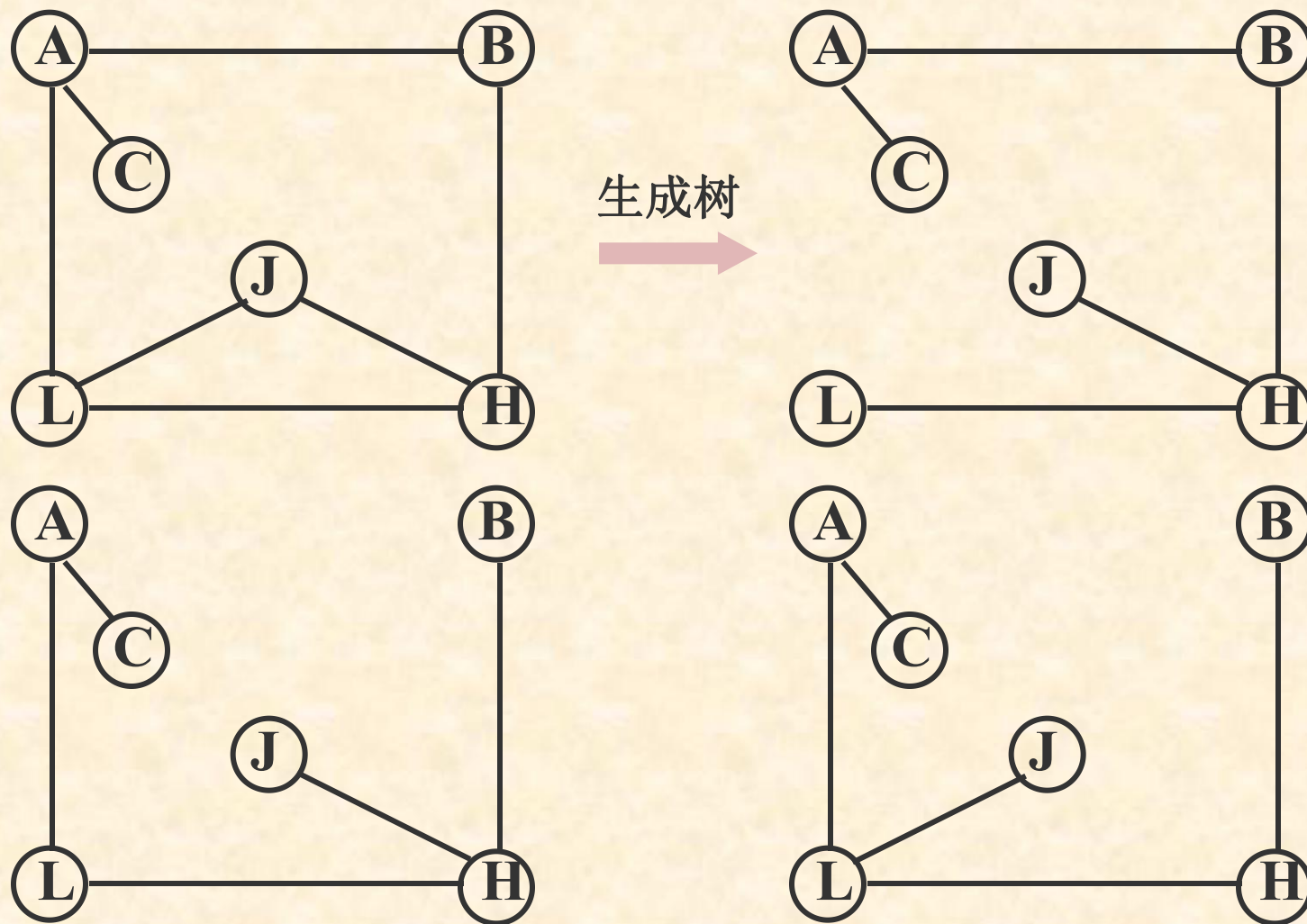


性质1: 一个有 n 个顶点的连通图的**生成树**有且仅有 $n-1$ 条边。

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。

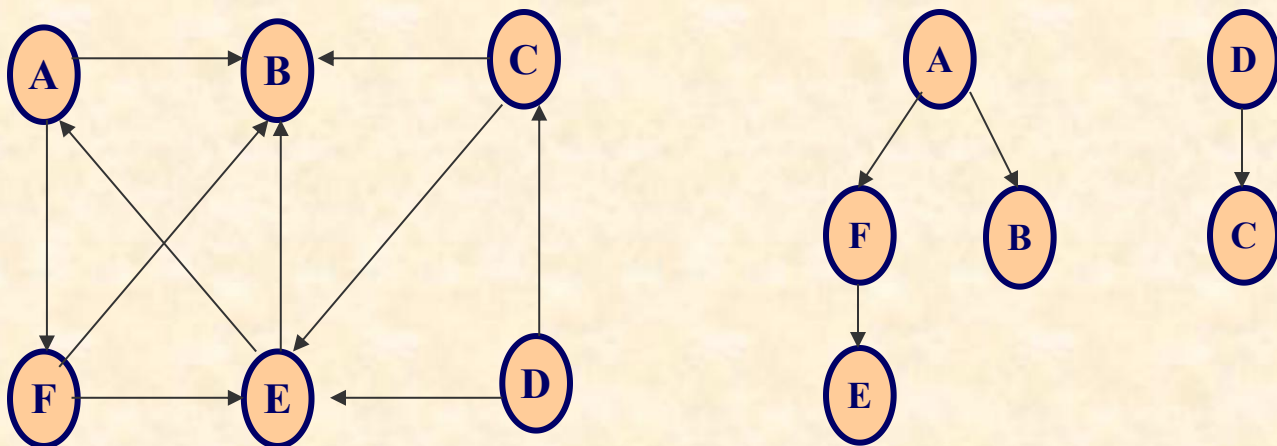


性质2: 一个连通图的生成树并不唯一



删除环中的任一条边

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**。

7.2 图的存储表示

一、图的数组(邻接矩阵)存储表示

二、图的邻接表存储表示

三、有向图的十字链表存储表示

四、无向图的邻接多重表存储表示

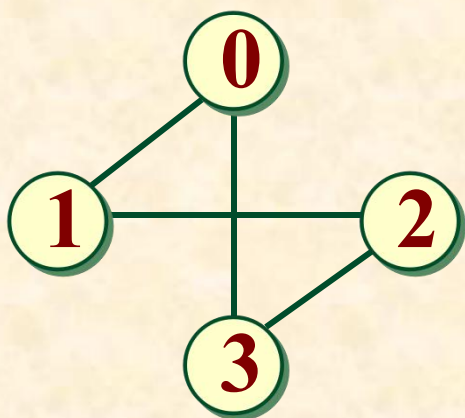


邻接矩阵 (Adjacency Matrix)-----数组表示法

- 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:

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



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



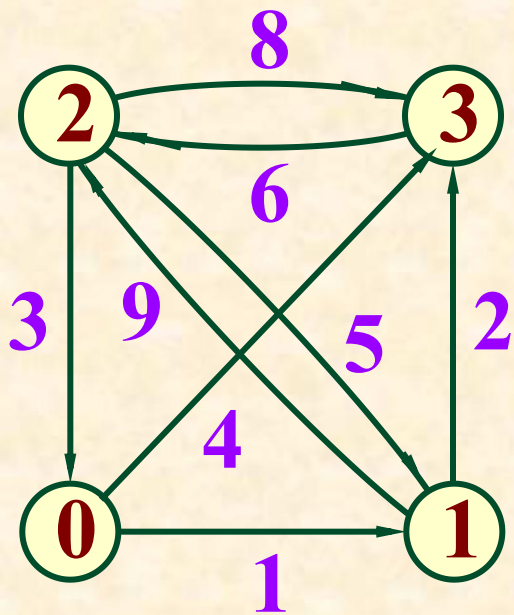
$$\mathbf{A}.\text{edge} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

- 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

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



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

```
typedef struct ArcCell { // 弧或边的定义
```

```
    VRType adj; // VRType是顶点关系类型。
```

```
    // 对无权图，用1或0表示相邻否；
```

```
    // 对带权图，则为权值类型。
```

```
    InfoType *info; // 该弧相关信息的指针
```

```
} ArcCell, AdjMatrix[MAX_VERTEX_NUM]  
[MAX_VERTEX_NUM];
```

```
typedef struct {                                // 图的定义
    VertexType                                // 顶点信息
        vexs[MAX_VERTEX_NUM];
    AdjMatrix  arcs;                            // 弧的信息
    int  vexnum, arcnum;                        // 顶点数，弧数
    GraphKind  kind;                            // 图的种类标志
} MGraph;
```

【p162算法7.2】



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**

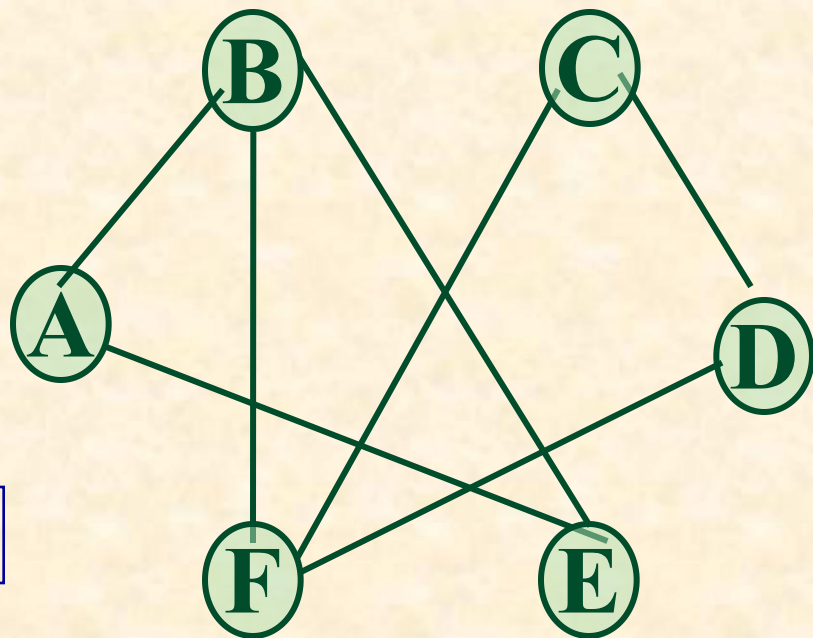
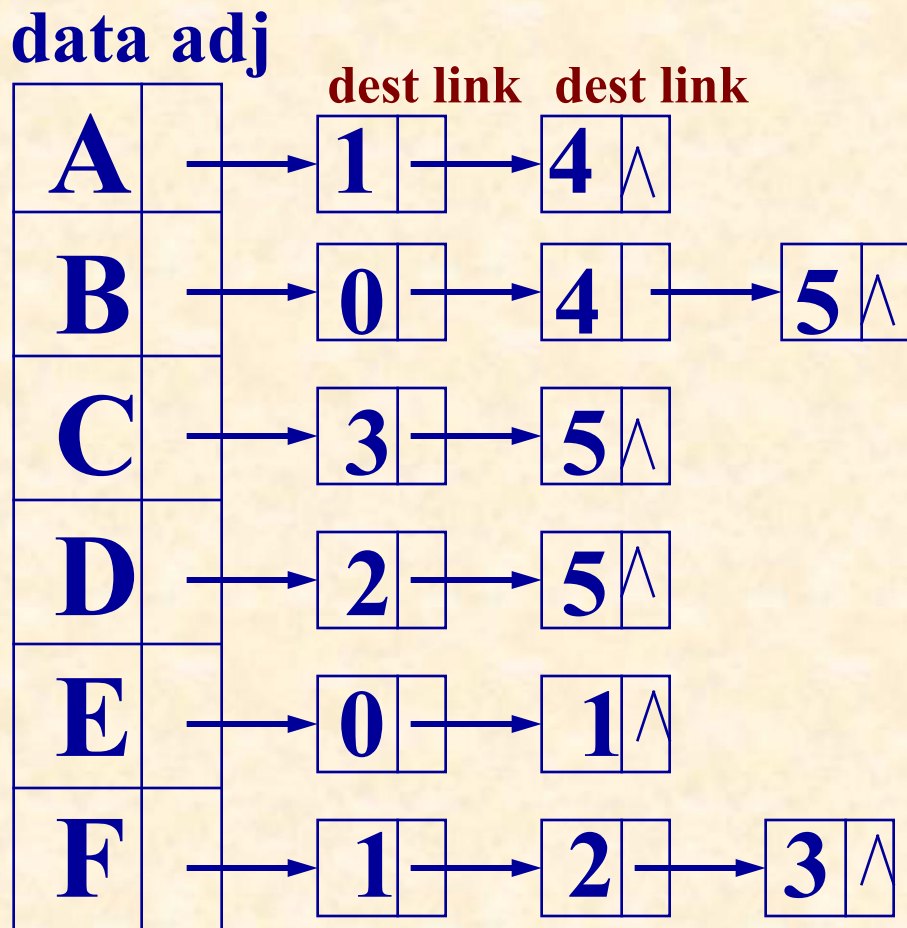
data	firstarc
------	----------

data; // 顶点信息

firstarc; //指向第一条依附该顶点的弧

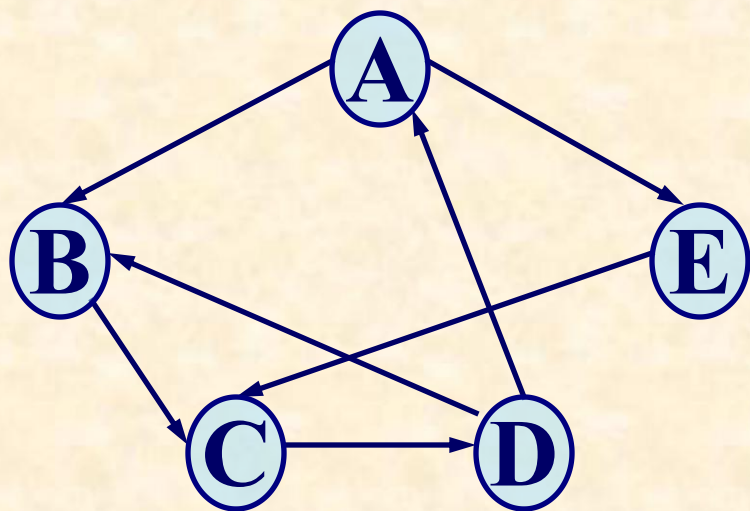
无向图的邻接表存储表示

0 1 2 3 4 5

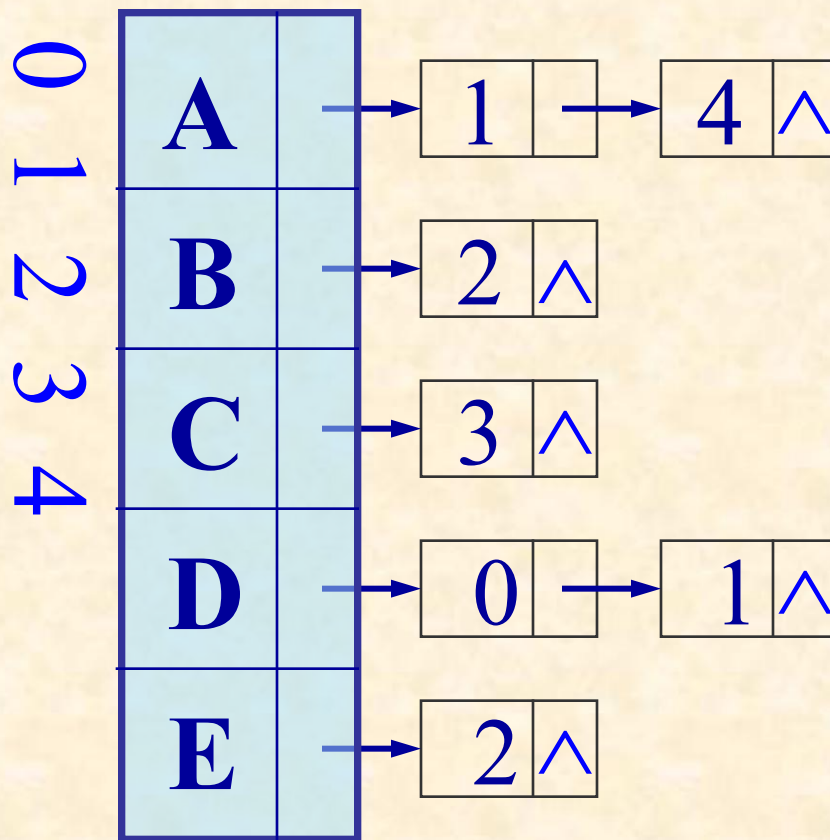


同一个顶点发出的边链接在同一个边链表中，每一个链结点代表一条边(边结点)，结点中有另一顶点的下标 **dest** 和指针 **link**。

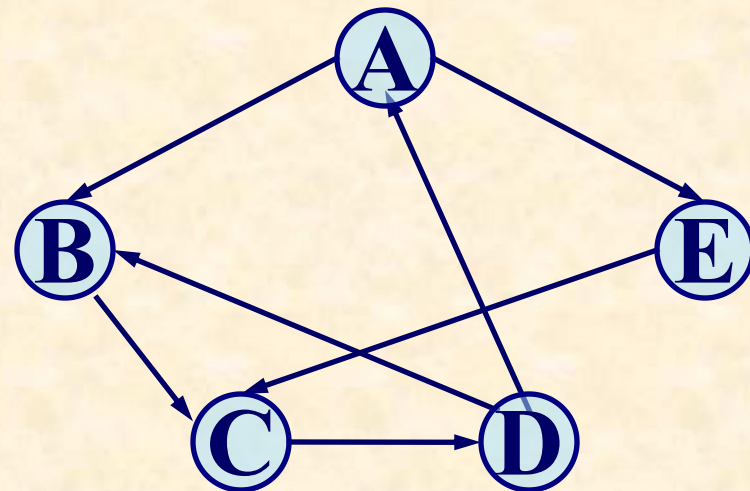
有向图的邻接表



可见，在有向图的邻接表中不易找到指向该顶点的弧



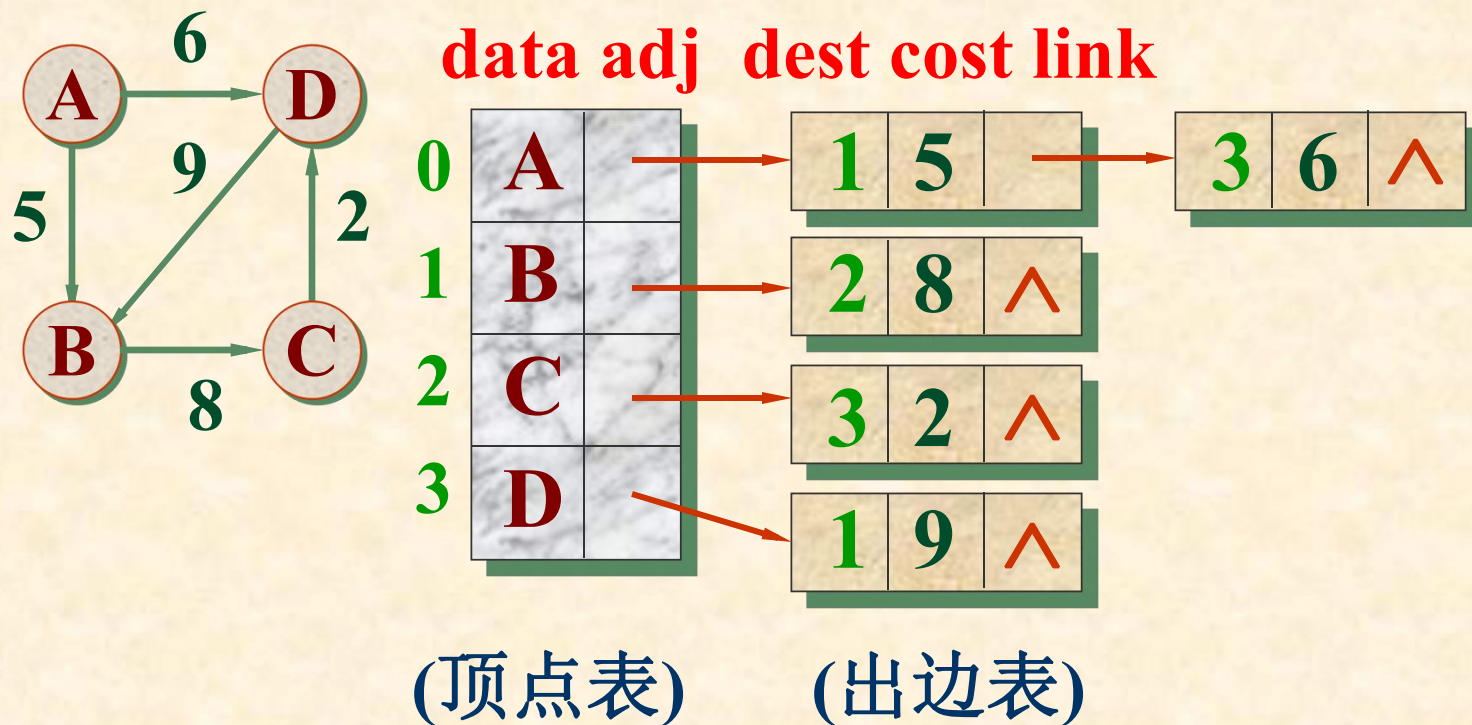
有向图的逆邻接表



在有向图的逆邻接表中，对每个顶点，链接的是指向该顶点的弧

0	A	→	3	^
1	B	→	3	→ 0
2	C	→	4	^
3	D	→	2	^
4	E	→	0	^

■ 网络 (带权图) 的邻接表



弧的结点结构

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

```
typedef struct ArcNode {  
    int      adjvex; // 该弧所指向的顶点的位置  
    struct ArcNode *nextarc;  
                                   // 指向下一条弧的指针  
    InfoType *info; // 该弧相关信息的指针  
} ArcNode;
```

顶点的结点结构

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

```
typedef struct VNode {  
    VertexType data; // 顶点信息  
    ArcNode *firstarc;  
    // 指向第一条依附该顶点的弧  
} VNode, AdjList[MAX_VERTEX_NUM];
```

图的结构定义(邻接表)

```
typedef struct {  
    AdjList vertices;  
  
    int    vexnum, arcnum;  
  
    int    kind;        // 图的种类标志  
  
} ALGraph;
```



邻接表的构造算法

```
void CreateGraph (AdjGraph G) {  
    scanf(&G. vexnum );           //输入顶点个数  
    scanf(&G. arcnum );           //输入边数  
    for ( int i = 0; i < G.n; i++) {  
        scanf( &G.vertices[i].data ); //输入顶点信息  
        G. vertices[i].firstAdj = NULL;  
    }  
    for ( i = 0; i < e; i++) {      //逐条边输入  
        scanf( &tail);scanf(&head);scanf(&weight);  
        EdgeNode * p = new EdgeNode;  
        p->adjvex = head;  p->cost = weight;  
    }  
}
```

//链入第 tail 号链表的前端

p->nextarc = G.vertices[tail].firstAdj;

G.vertices[tail].firstAdj = p;

p = new EdgeNode;

p->adjvex = tail; p->cost = weight;

//链入第 head 号链表的前端

p->nextarc = G.vertices[head].firstAdj;

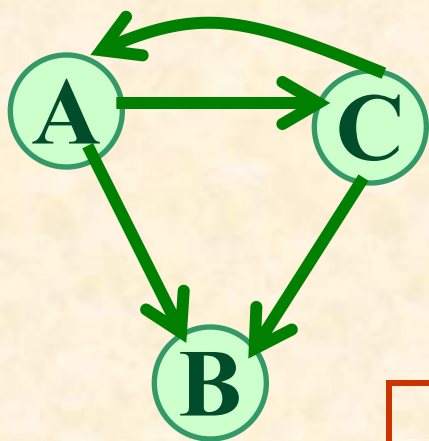
G.vertices[head].firstAdj = p;

}

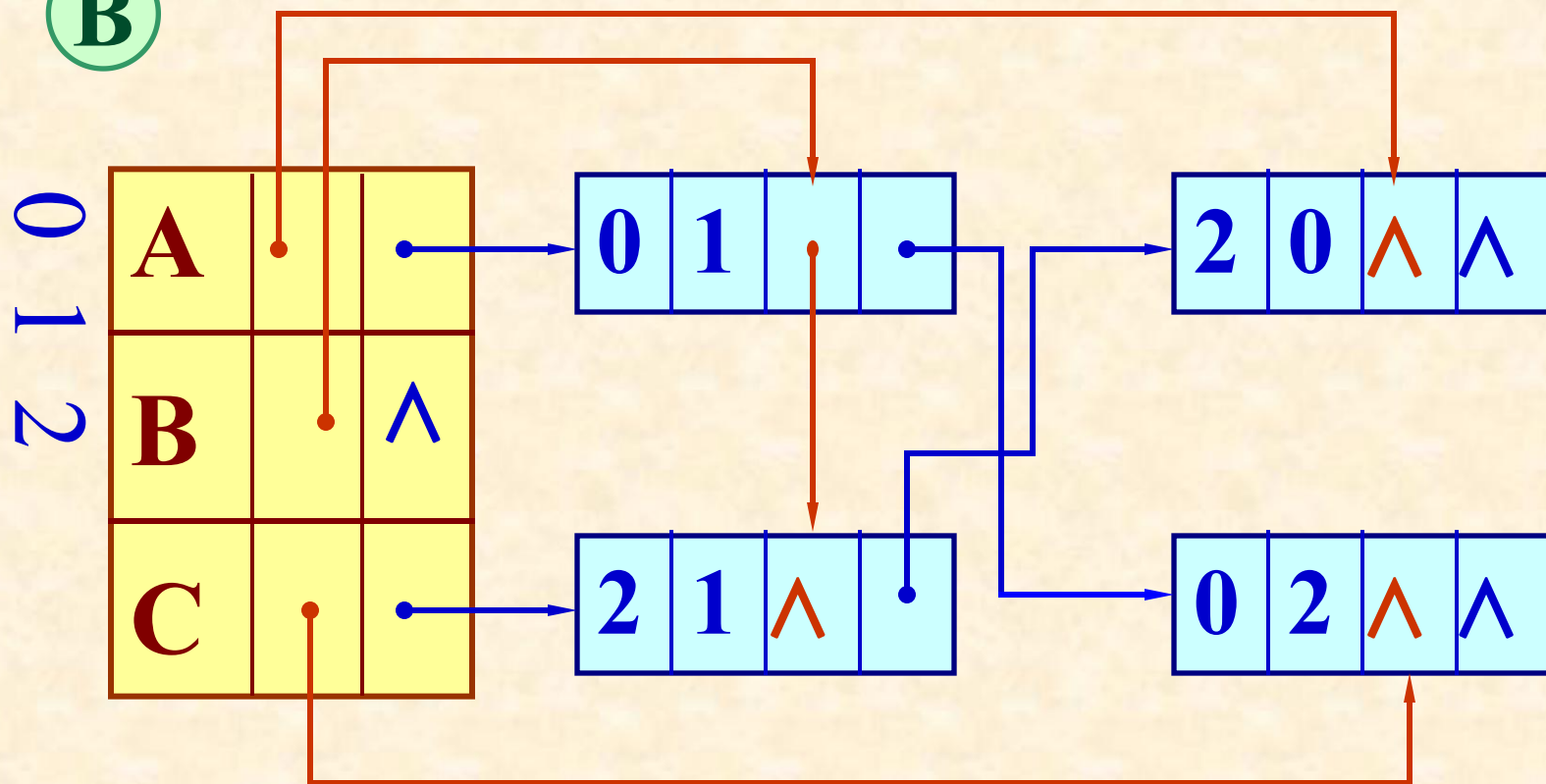
}



三、有向图的十字链表存储表示



是将有向图的邻接表和逆邻接表结合起来得到的一种链表



弧的结点结构

tailvex

headvex

hlink

tlink

info

弧尾顶点位置

弧头顶点位置

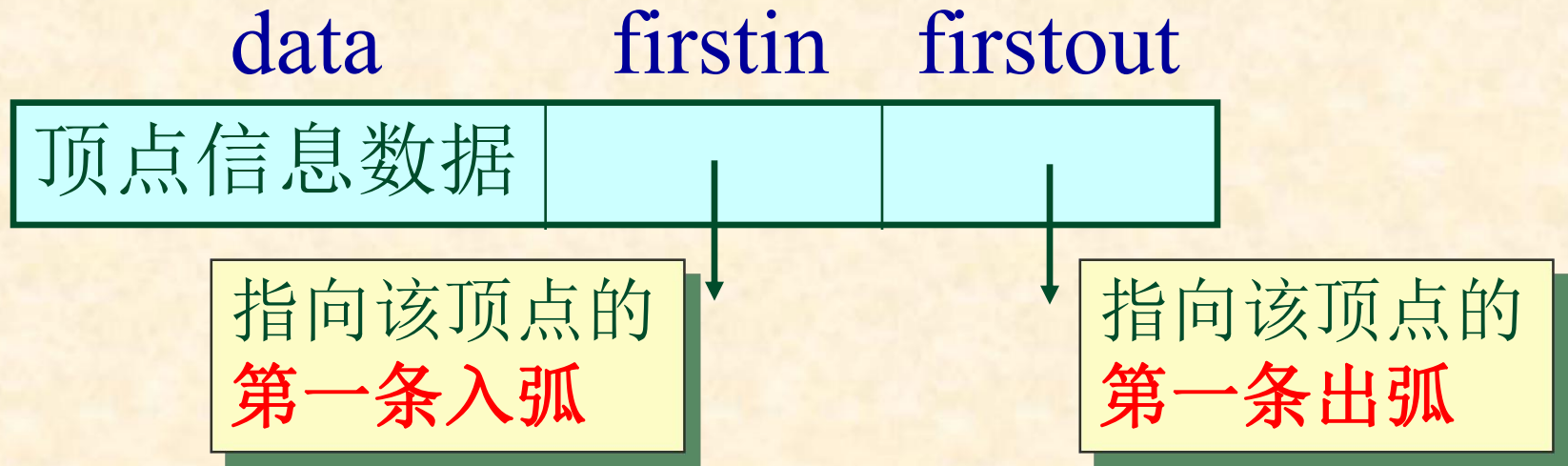
弧的相关信息

指向下一个
有**相同弧头**
的结点

指向下一个
有**相同弧尾**
的结点

```
typedef struct ArcBox {           // 弧的结构表示
    int tailvex, headvex; InfoType *info;
    struct ArcBox *hlink, *tlink;
} ArcBox ;
```

顶点的结点结构



```
typedef struct VexNode { // 顶点的结构表示  
    VertexType data;  
    ArcBox *firstin, *firstout;  
} VexNode;
```

有向图的结构表示(十字链表)

```
typedef struct {
```

```
    VexNode xlist[MAX_VERTEX_NUM];
```

```
    // 顶点结点(表头向量)
```

```
    int vexnum, arcnum;
```

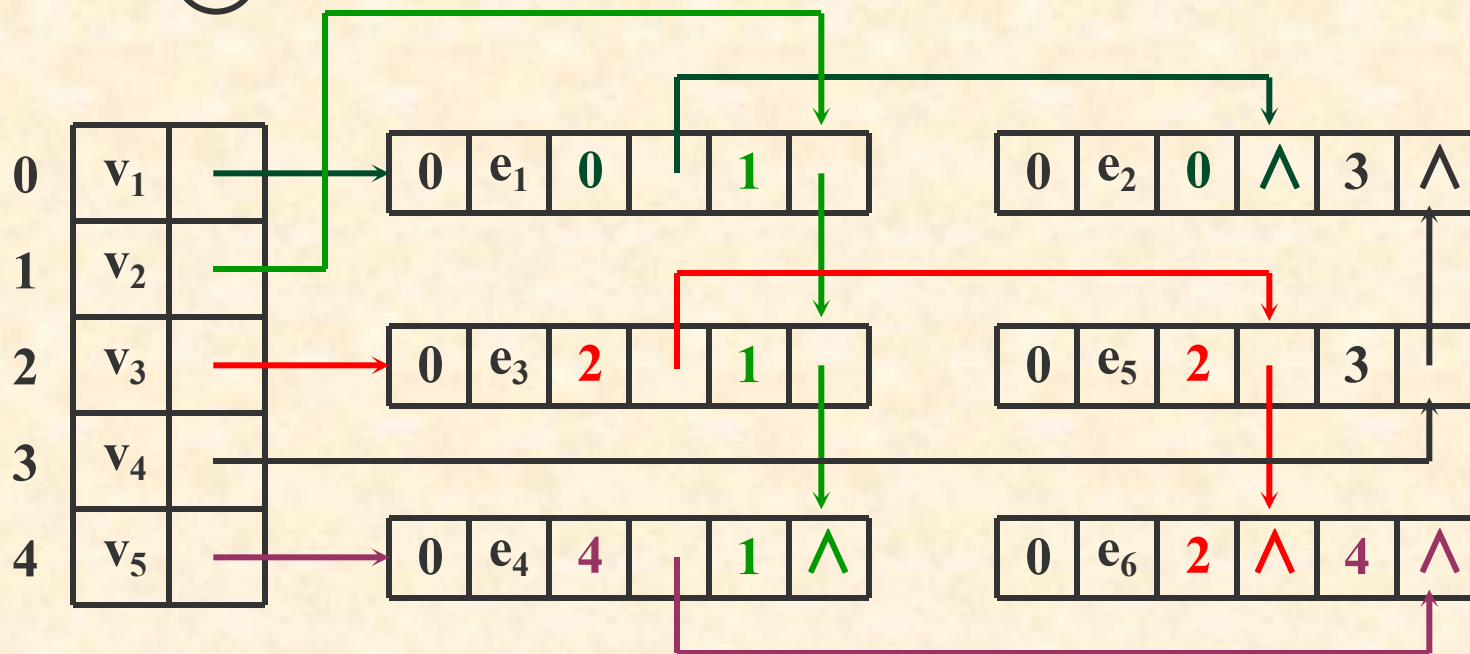
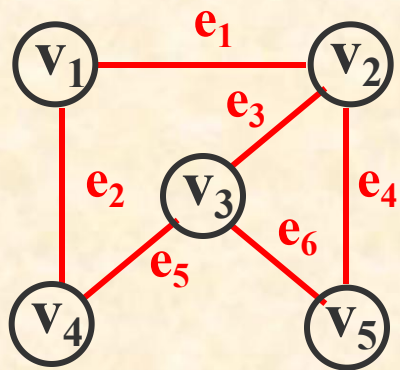
```
    //有向图的当前顶点数和弧数
```

```
} OLGraph;
```

【p165建立有向图算法】



四、无向图的邻接多重表存储表示



边的结构表示

```
typedef struct Ebox {  
    VisitIf    mark;           // 访问标记  
  
    int        ivex, jvex;     //该边依附的两个顶点的位置  
  
    struct EBox *ilink, *jlink;  
  
    InfoType    *info;         // 该边信息指针  
} EBox;
```

顶点的结构表示

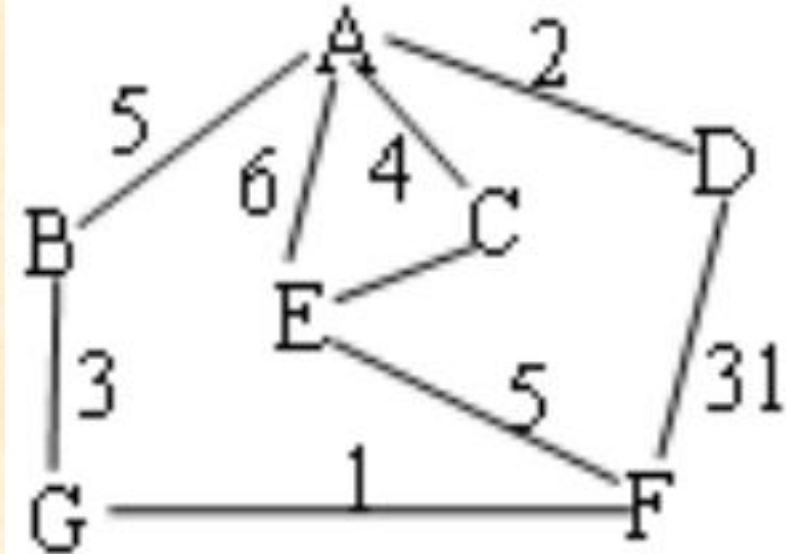
```
typedef struct VexBox {  
    VertexType data;  
    EBox *firstedge; // 指向第一条依附该顶点的边  
} VexBox;
```

无向图的结构表示

```
typedef struct { // 邻接多重表  
    VexBox adjmulist[MAX_VERTEX_NUM];  
    int vexnum, edgenum;  
} AMLGraph;
```



作业1:画出G的邻接表存储结构



作业2: 已知无向图G, $V(G) = \{1, 2, 3, 4\}$, $E(G) = \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)\}$
试画出G的邻接多重表, 并说明, 若已知点i, 如何根据邻接多重表找到与i相邻的点j?

7.3 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[0 .. n-1]** 。

- 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)

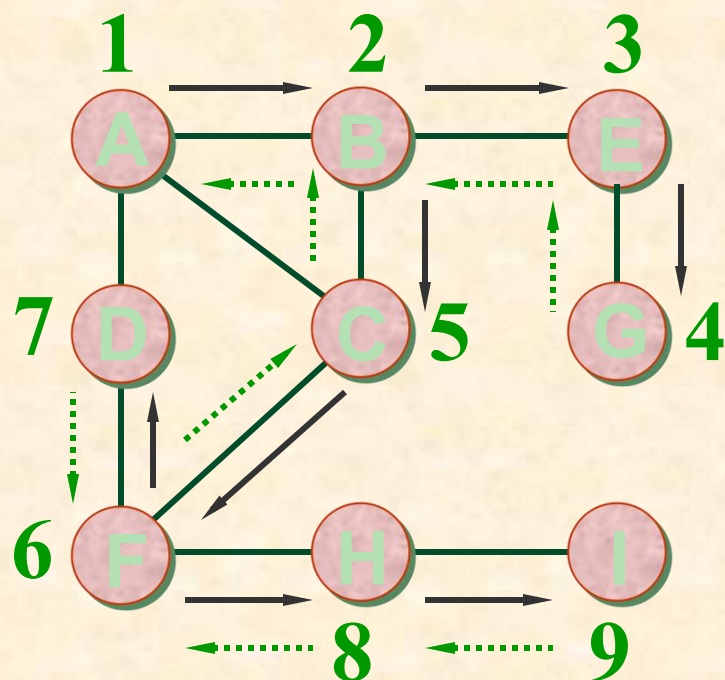
- ◆ 广度优先搜索

BFS (Breadth First Search)

- **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 **w1**; 再从 **w1** 出发, 访问与 **w1** 邻接但还没有访问过的顶点 **w2**; 然后再从 **w2** 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。
- 若此时图中尚有顶点未被访问, 则另选图中一个未曾被访问的顶点做起始点, 重复上述过程, 直至图中所有顶点都被访问到。

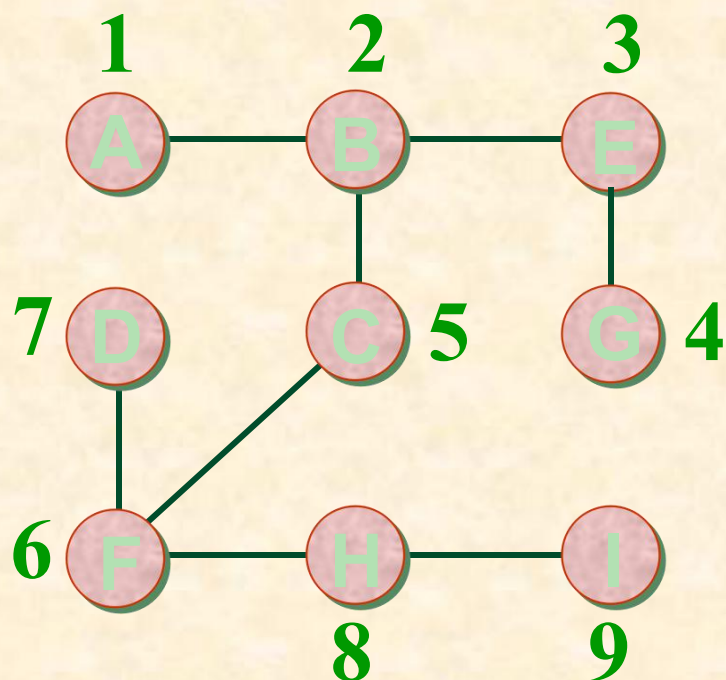
深度优先搜索DFS (Depth First Search)

■ 深度优先搜索过程



前进 —————>

回退>

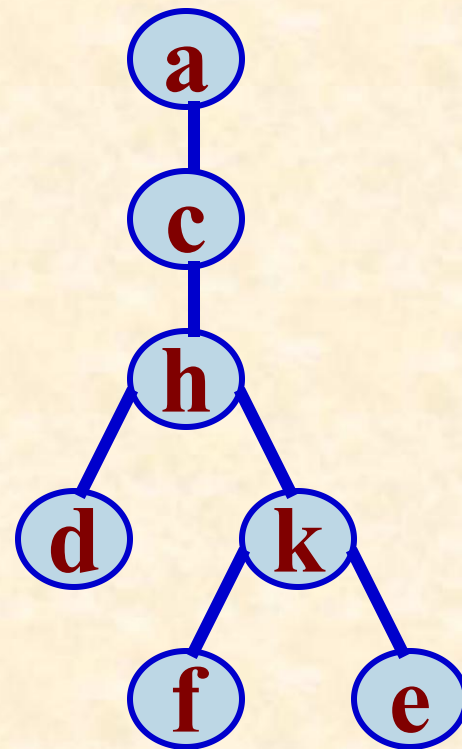
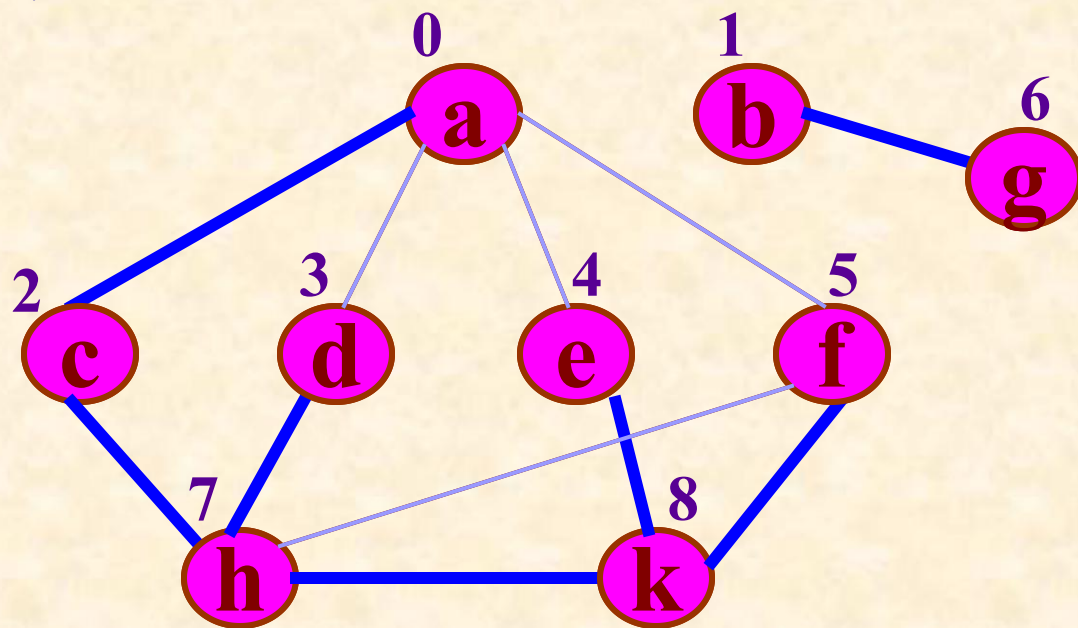


深度优先生成树

```
void DFS(Graph G, int v) {  
    // 从顶点v出发，深度优先搜索遍历连通图 G  
    visited[v] = TRUE;  printf(v);  
    for(w=FirstAdjVex(G, v);  
        w!=0; w=NextAdjVex(G,v,w))  
        if (!visited[w]) DFS(G, w);  
        // 对v的尚未访问的邻接顶点w  
        // 递归调用DFS  
} // DFS
```

```
void DFSTraverse(Graph G,  
                  Status (*Visit)(int v)) {  
    // 对图 G 作深度优先遍历  
    VisitFunc = Visit;  
    for (v=0; v<G.vexnum; ++v)  
        visited[v] = FALSE; // 访问标志数组初始化  
    for (v=0; v<G.vexnum; ++v)  
        if (!visited[v]) DFS(G, v);  
        // 对尚未访问的顶点调用DFS  
}
```

例如:



访问标志:

0	1	2	3	4	5	6	7	8
T	T	T	T	T	T	T	T	T

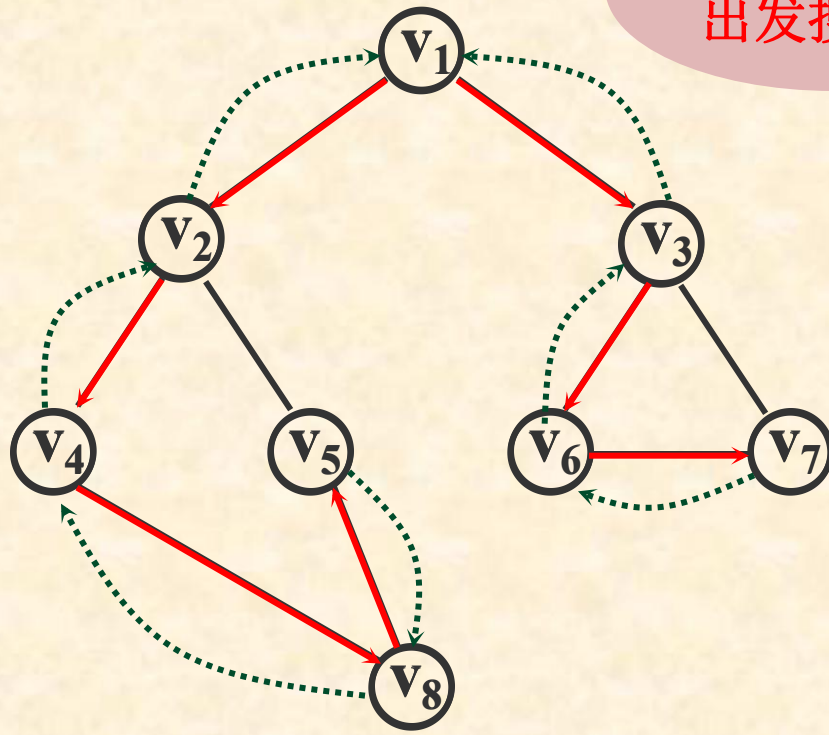
访问次序:

a	c	h	d	k	f	e	b	g
---	---	---	---	---	---	---	---	---



栈实现深度优先搜索

总是从栈顶
出发搜索！



深度优先搜索顺序: V_1 V_2 V_4 V_8 V_5 V_3 V_6 V_7

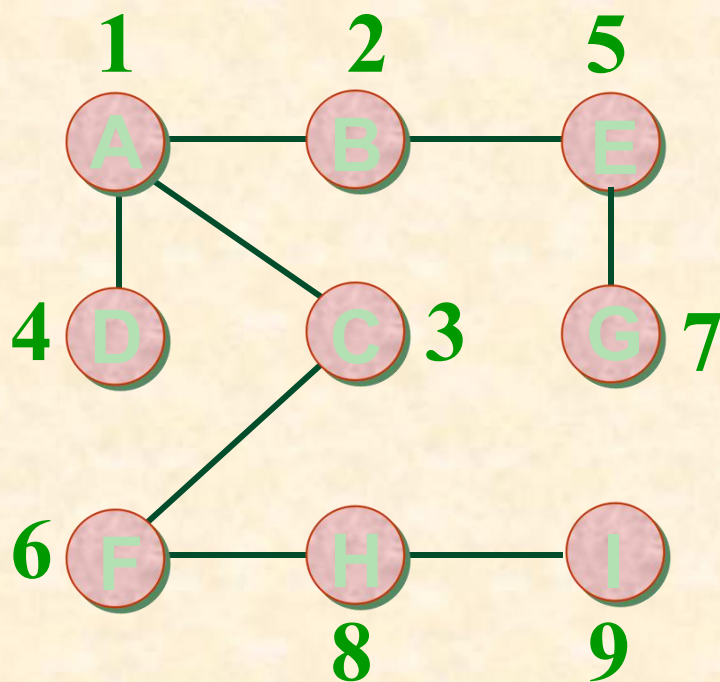
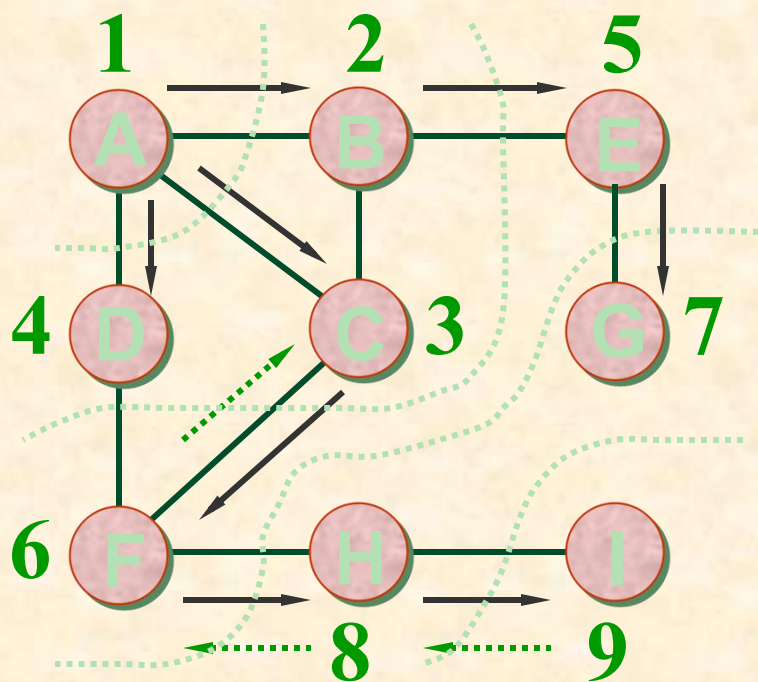
```
void DFS ( Graph G , int v ) {  
    initstack(S) ; visited[v] = TRUE ; Push(S , v) ; printf(v) ;  
    while ( ! StackEmpty(S) ) {  
        Gettop(S , v) ;  
        for ( w=FirstAdjVex(G,v) ; w>=0 ; w=NextAdjVex(G, v,w) )  
            if ( !visited[w] ) {  
                visited[w] = TRUE ; Push(S , w) ; printf(w) ;  
                Gettop(S , v) ; break;}  
        Pop(S) ;  
    }  
}
```

- 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。

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

队列实现广度优先搜索算法

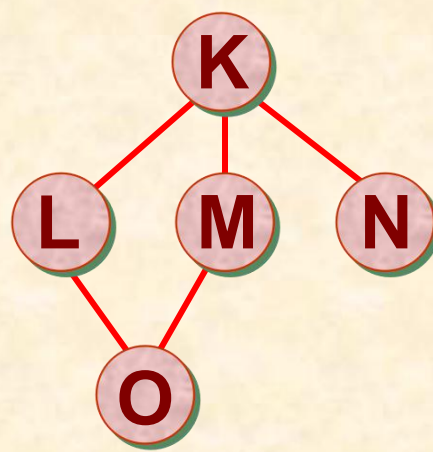
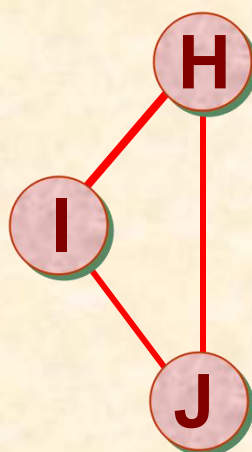
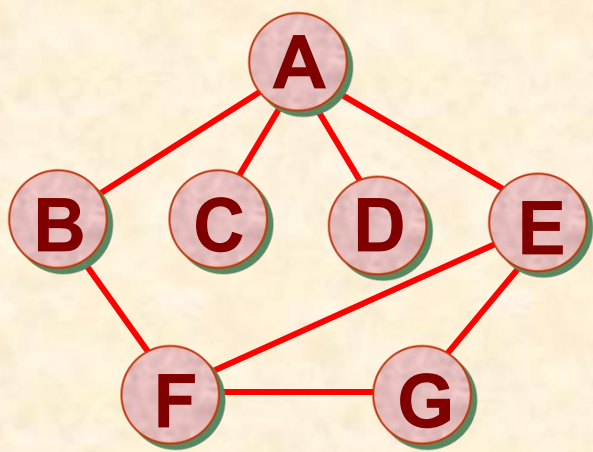
```
void BFSTraverse ( Graph G , int v ) {  
    // visited[0..n-1] 初始均为 0 ; v 指示顶点在数组中的位置 ;  
    InitQueue(Q) ;  
    visited[v]=TRUE ; EnQueue(Q , v) ; printf(v) ;  
    while ( ! QueueEmpty(Q) ) {  
        DeQueue(Q , u) ;  
        for ( w=FirstAdjVex (G,u) ; w>=0 ; w=NextAdjVex(G, u, w) )  
            if ( ! visited[w] ) {  
                visited[w] = TRUE ; EnQueue(Q , w) ; printf(w) ; }  
                //其邻接顶点均送入队列  
            }  
    }  
}
```

图的连通性问题

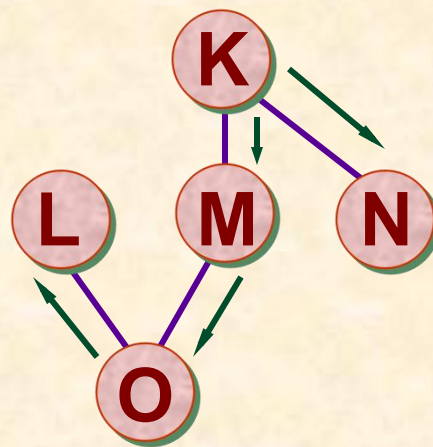
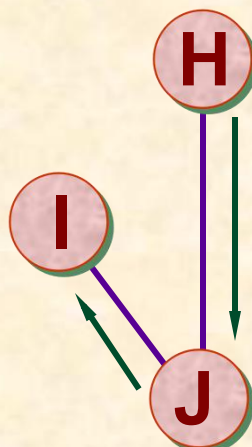
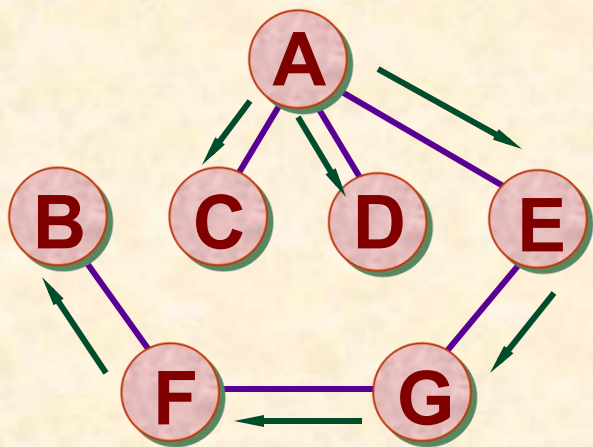
连通分量 (Connected component)

- 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

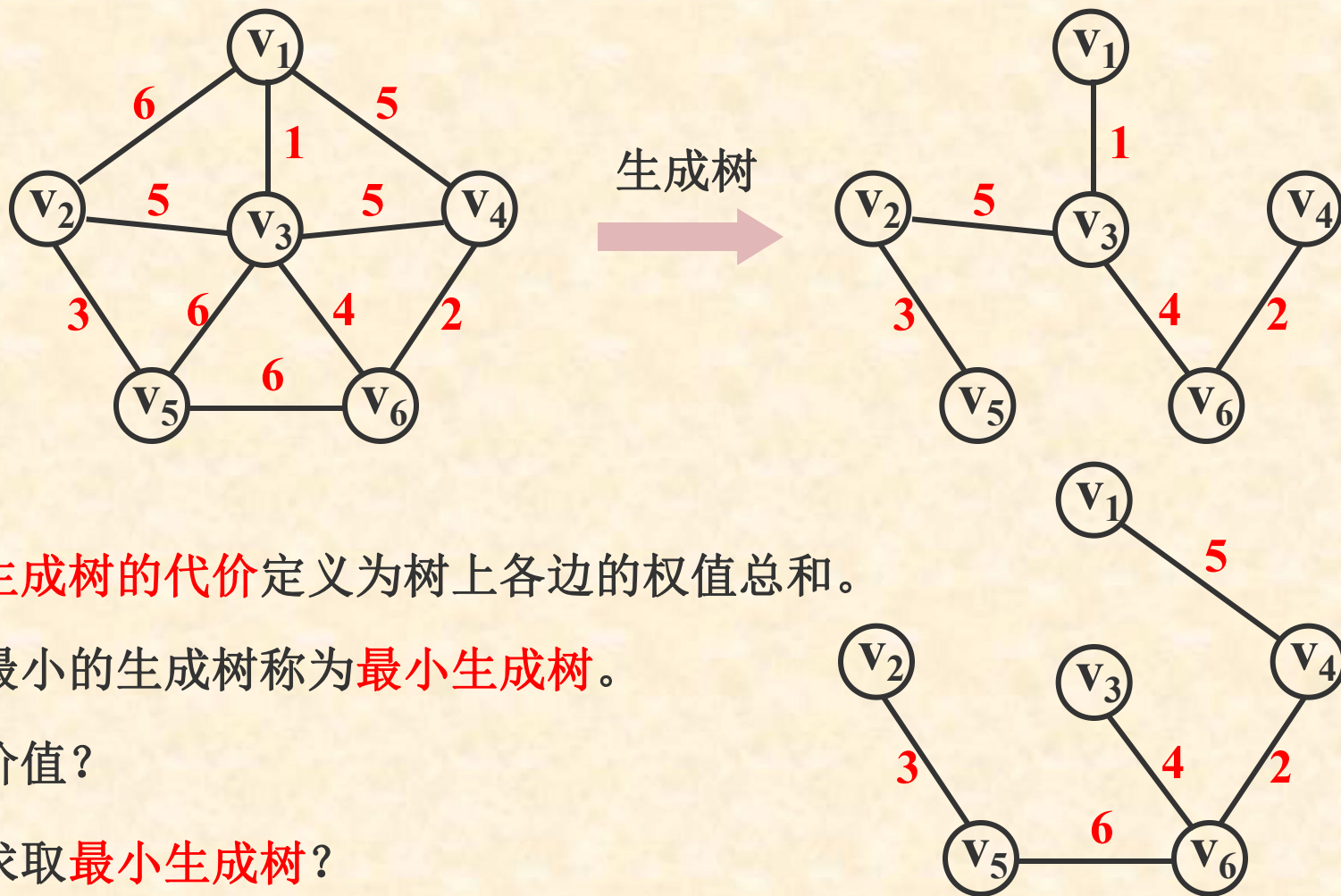


非连通图的连通分量

7.4.3 最小生成树

一个无向图可以对应多个生成树。

一个带权无向图(无向网)同样可以对应多个生成树。



一棵生成树的代价定义为树上各边的权值总和。

代价最小的生成树称为最小生成树。

实际价值？

如何求取最小生成树？

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。

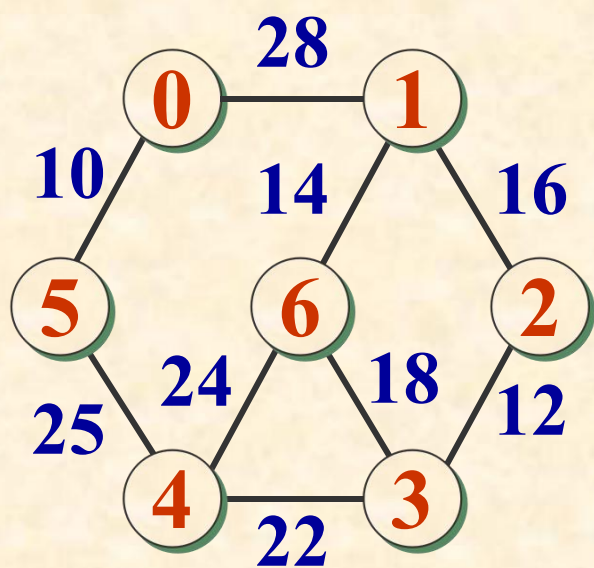
普里姆(Prim)算法

- 普里姆算法的基本思想:

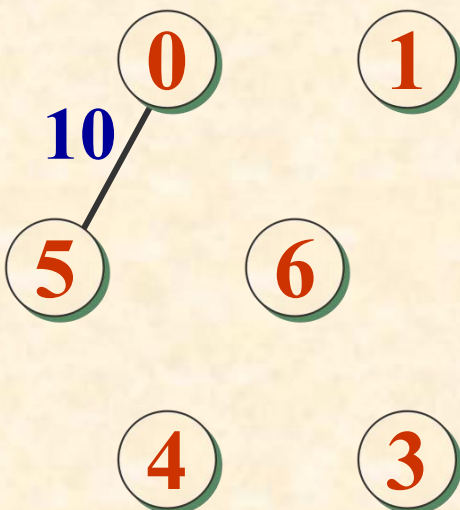
从连通网络 $N = \{V, E\}$ 中的某一顶点 u_0 出发, 选择与它关联的具有最小权值的边 (u_0, v) , 将其顶点加入到生成树顶点集合 U 中。

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。

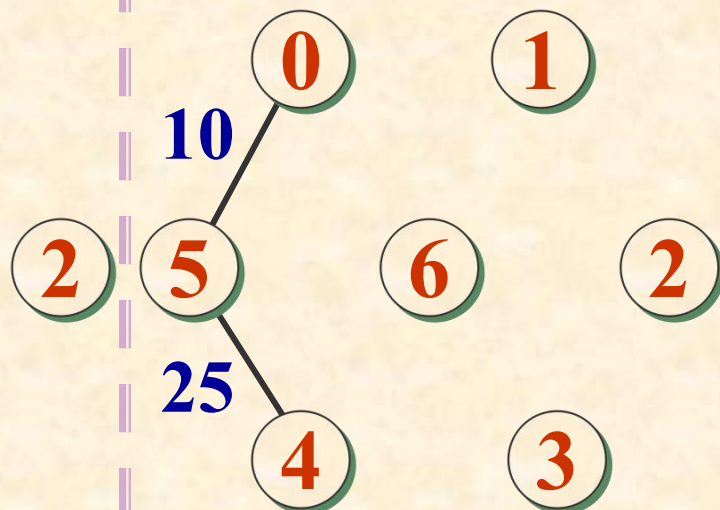
重点：边一定存在于 U 与 $V-U$ 之间



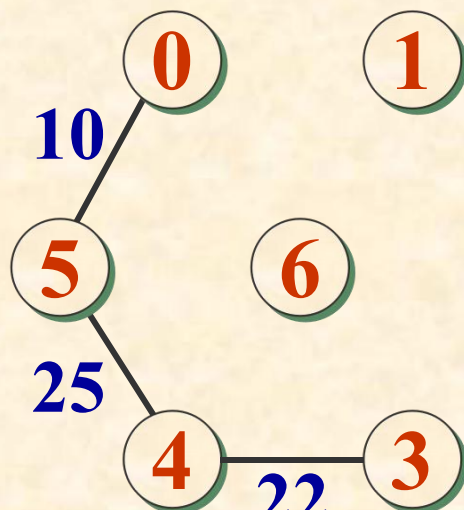
原图



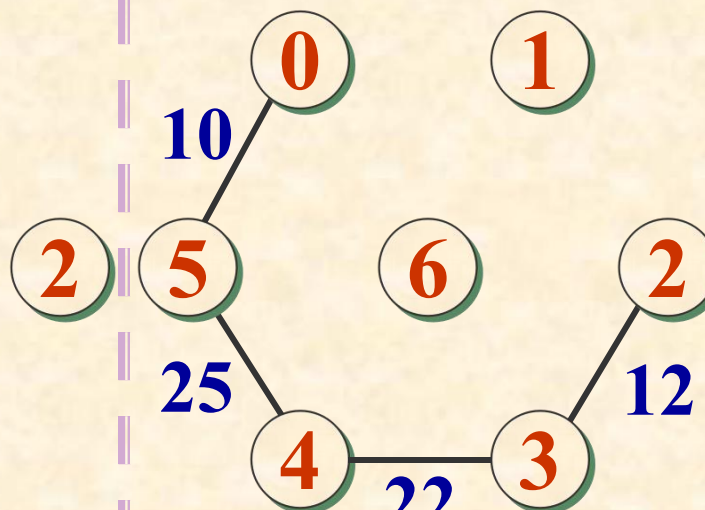
(a)



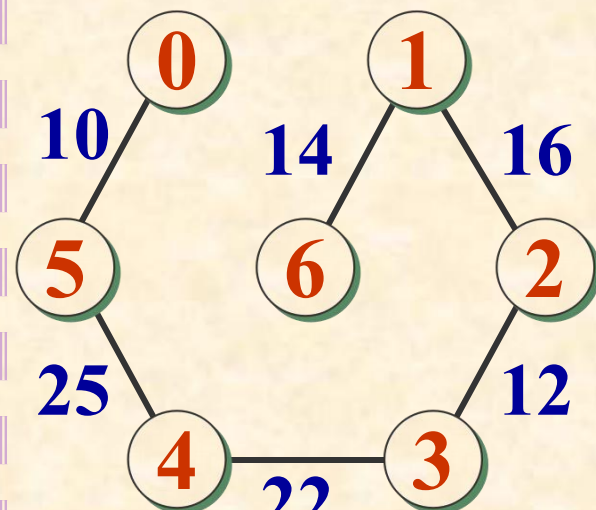
(b)



(c)



(d)

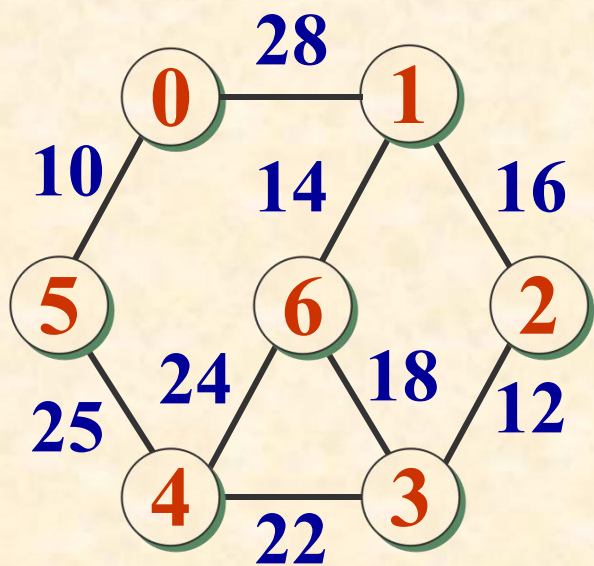


(e) (f)

采用邻接矩阵作为图的存储表示。

- 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	∞	∞	∞	10	∞
28	0	16	∞	∞	∞	14
∞	16	0	12	∞	∞	∞
∞	∞	12	0	22	∞	18
∞	∞	∞	22	0	25	24
10	∞	∞	∞	25	0	∞
∞	14	∞	18	24	∞	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	∞	∞	∞	10	∞
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]。

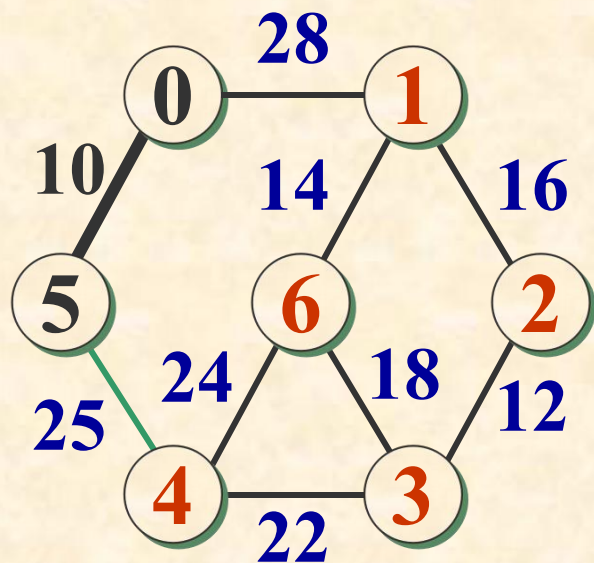
- ◆ 将 $\text{nearvex}[v]$ 改为-1, 表示它已加入生成树顶点集合。
- ◆ 将边 $(\text{nearvex}[v], v, \text{lowcost}[v])$ 加入生成树的边集合。
- ◆ 取 $\text{lowcost}[i] = \min\{\text{lowcost}[i], \text{Edge}[v][i]\}$, 即用生成树顶点集合外各顶点 i 到刚加入该集合的新顶点 v 的距离 $\text{Edge}[v][i]$ 与原来它们到生成树顶点集合中顶点的最短距离 $\text{lowcost}[i]$ 做比较, 取距离近的作为这些集合外顶点到生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

顶点 $v=5$ 加入生成树顶点集合:

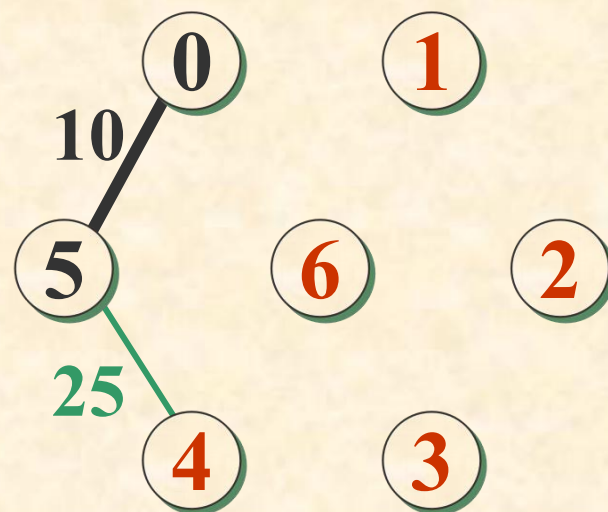
	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	25	10	∞

nearvex	-1	0	0	0	5	-1	0
---------	----	---	---	---	---	----	---

选 $v=4$ \uparrow 选边 (5,4)



原图

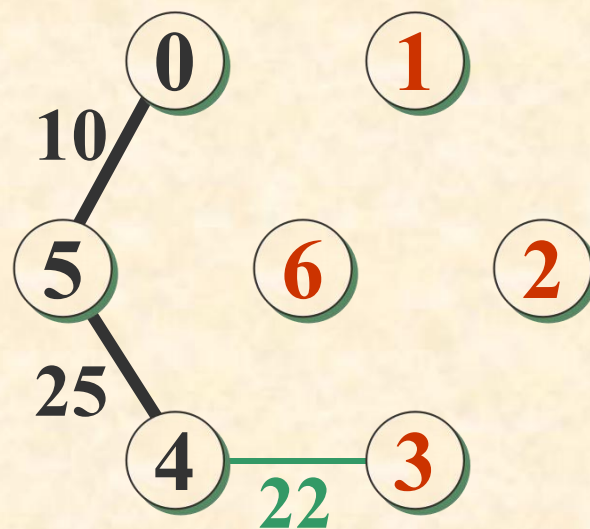
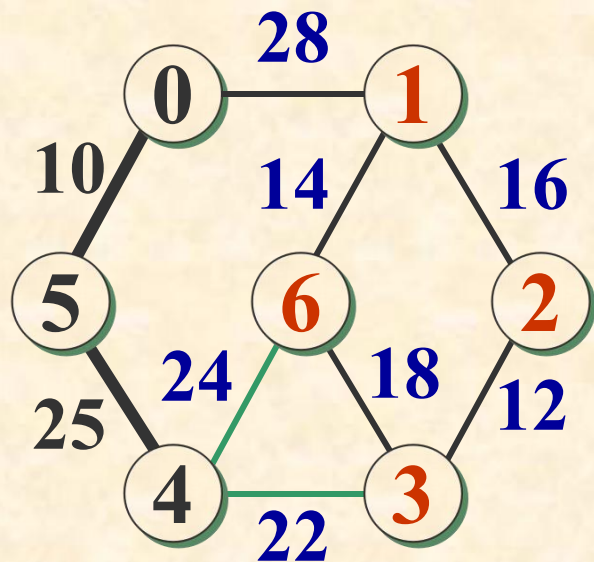


边 (0,5,10) 加入生成树

顶点 $v=4$ 加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	28	∞	22	25	10	24
nearvex	-1	0	0	4	-1	-1	4

选 $v=3$ \uparrow 选边 (4,3)

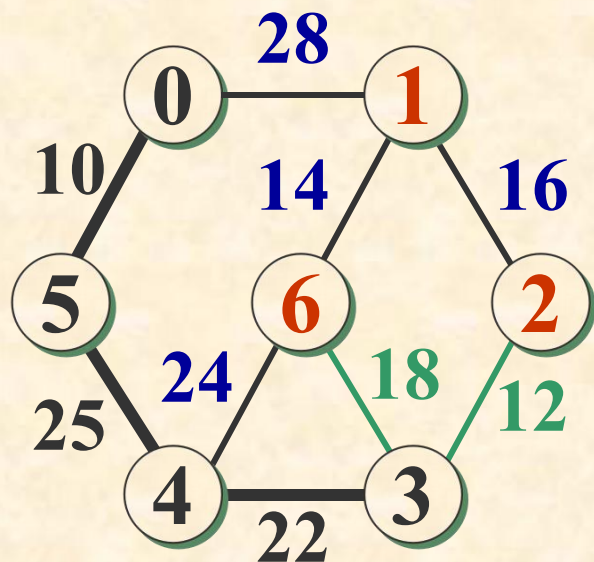


顶点 $v=3$ 加入生成树顶点集合:

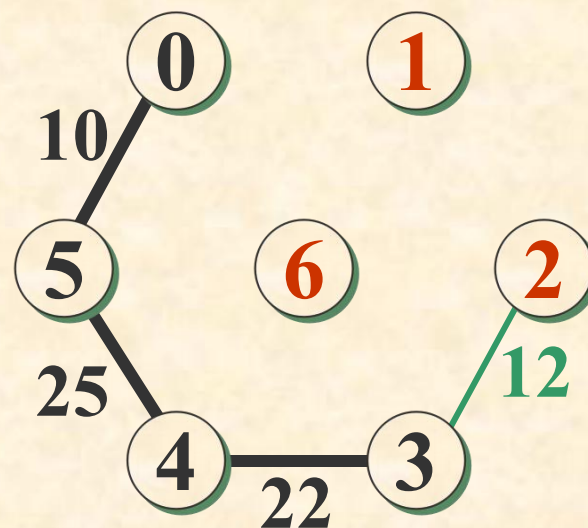
	0	1	2	3	4	5	6
lowcost	0	28	12	22	25	10	18

nearvex	-1	0	3	-1	-1	-1	3
---------	----	---	---	----	----	----	---

选 $v=2$  选边 (3,2)



原图

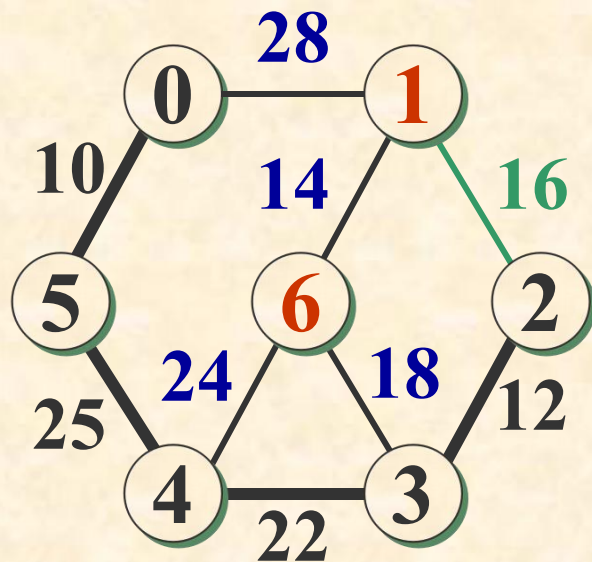


边 (4,3,22) 加入生成树

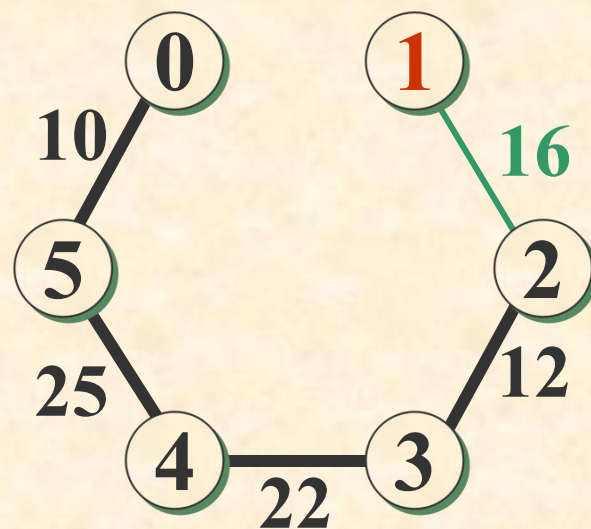
顶点 $v=2$ 加入生成树顶点集合:

	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

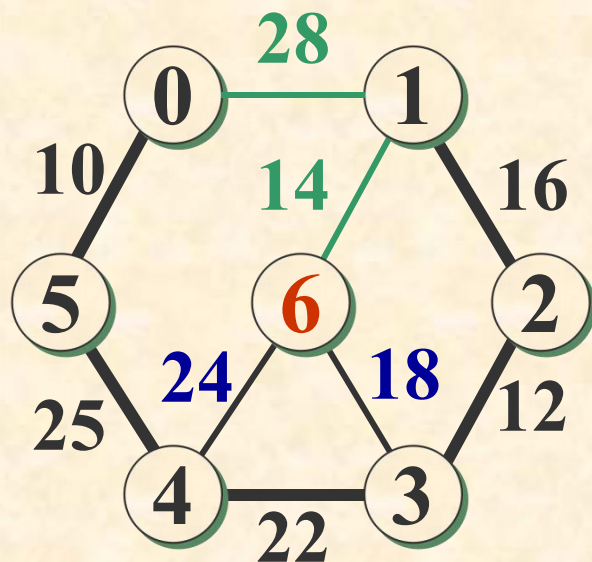


边 (3,2,12) 加入生成树

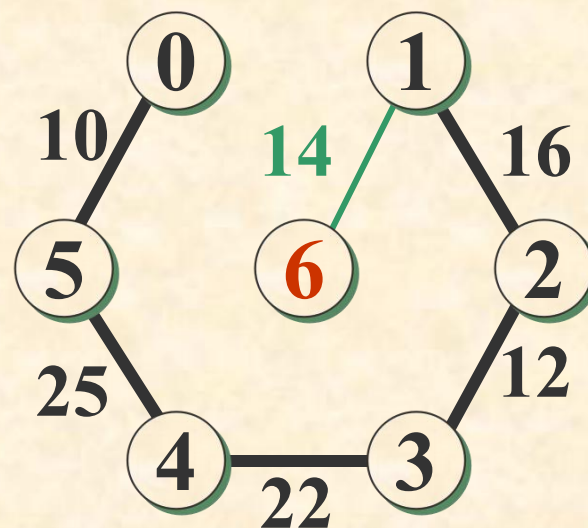
顶点 $v=1$ 加入生成树顶点集合:

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

选 $v=6$ \uparrow 选边 (1,6)



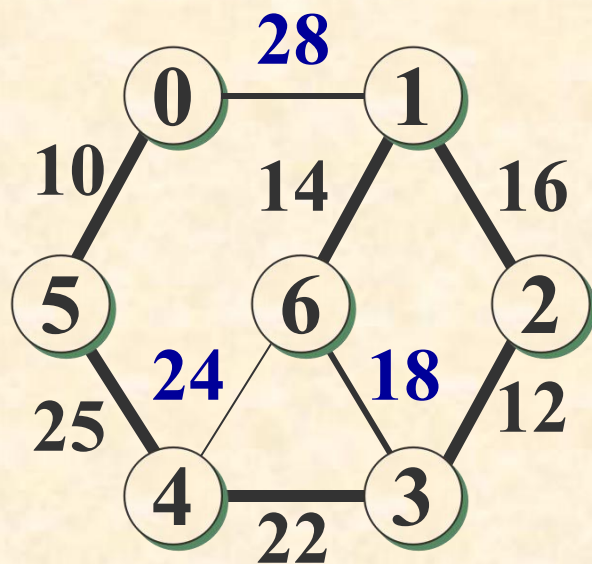
原图



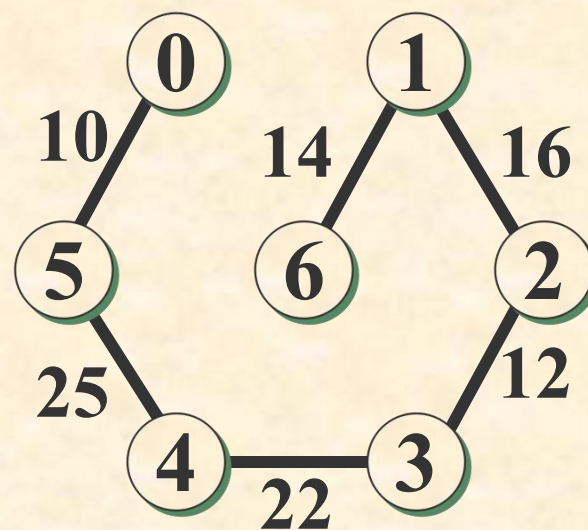
边 (2,1,16) 加入生成树

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指针
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 = -1;
        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 != -1 ) {           //v=-1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。

Kruskal 算法

思想：

$N = (V, E)$ 是 n 个顶点的连通网，设 E 是连通网中边的集合；

构造最小生成树 $N' = (V, TE)$ ， TE 是最小生成树中边的集合，
初始 $TE = \{\}$ ；

重复执行：

选取 E 中权值最小的边 (u, v) ，

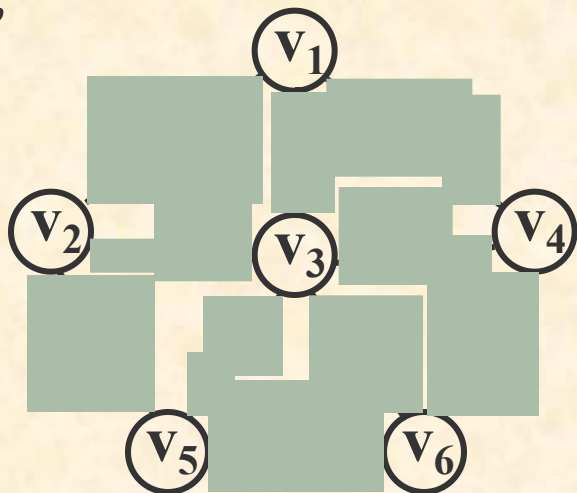
判断边 (u, v) 与 TE 中的边是否构成回路？

否，将边 (u, v) 纳入 TE 中，并从 E 中删除边 (u, v) ；

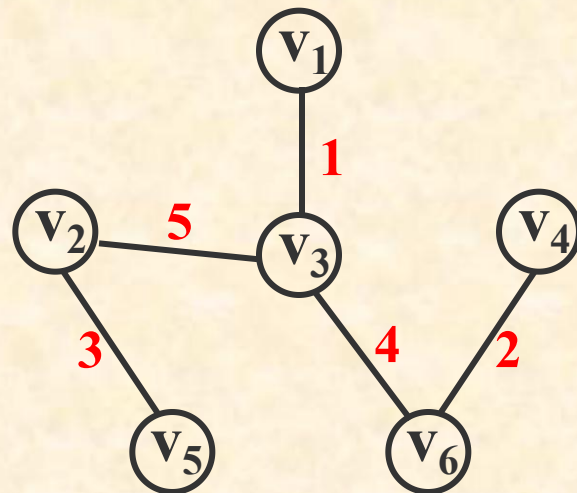
直至 E 为空；

u 和 v 一定
不在同一个
连通分量中

例,



当前权值最小边 (v_5, v_6)



初始 $TE = \{ \}$

$\langle v_1, v_3 \rangle$

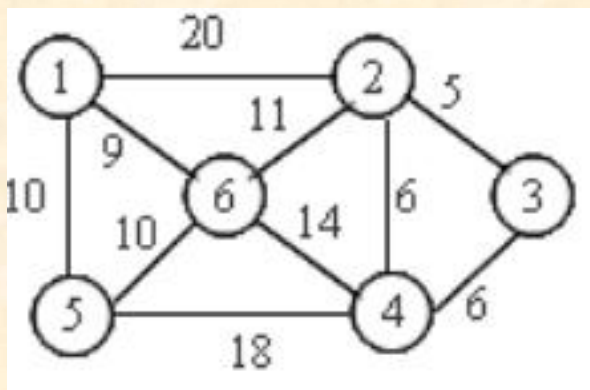
$\langle v_4, v_6 \rangle$

$\langle v_2, v_5 \rangle$

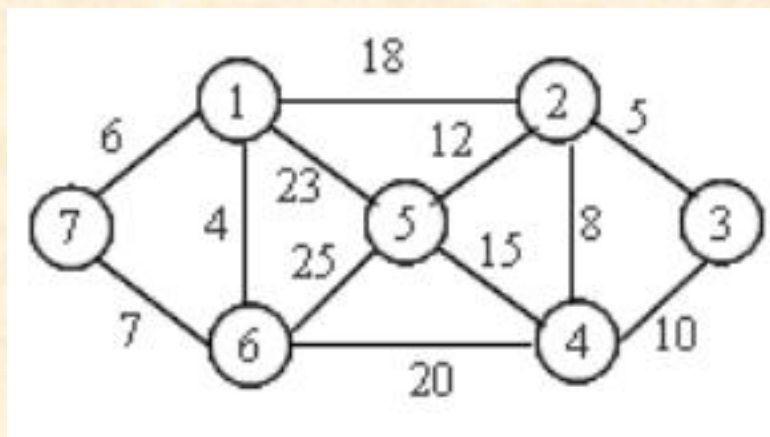
$\langle v_3, v_6 \rangle$

$\langle v_2, v_3 \rangle$

1. 已知一个无向图如下图所示，要求分别用Prim和Kruskal算法生成最小树（假设以①为起点，试画出构造过程）。



2. 试写出用克鲁斯卡尔（Kruskal）算法构造下图的一棵最小支撑（或生成）树的过程。



普里姆算法算法时间复杂度:

与边的个数无关; $O(n^2)$

适合于求边稠密的网的最小生成树。

Kruskal 算法

算法时间复杂度: $O(e \log e)$ e 为网的边的数目

适合于求边稀疏的网的最小生成树。

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)

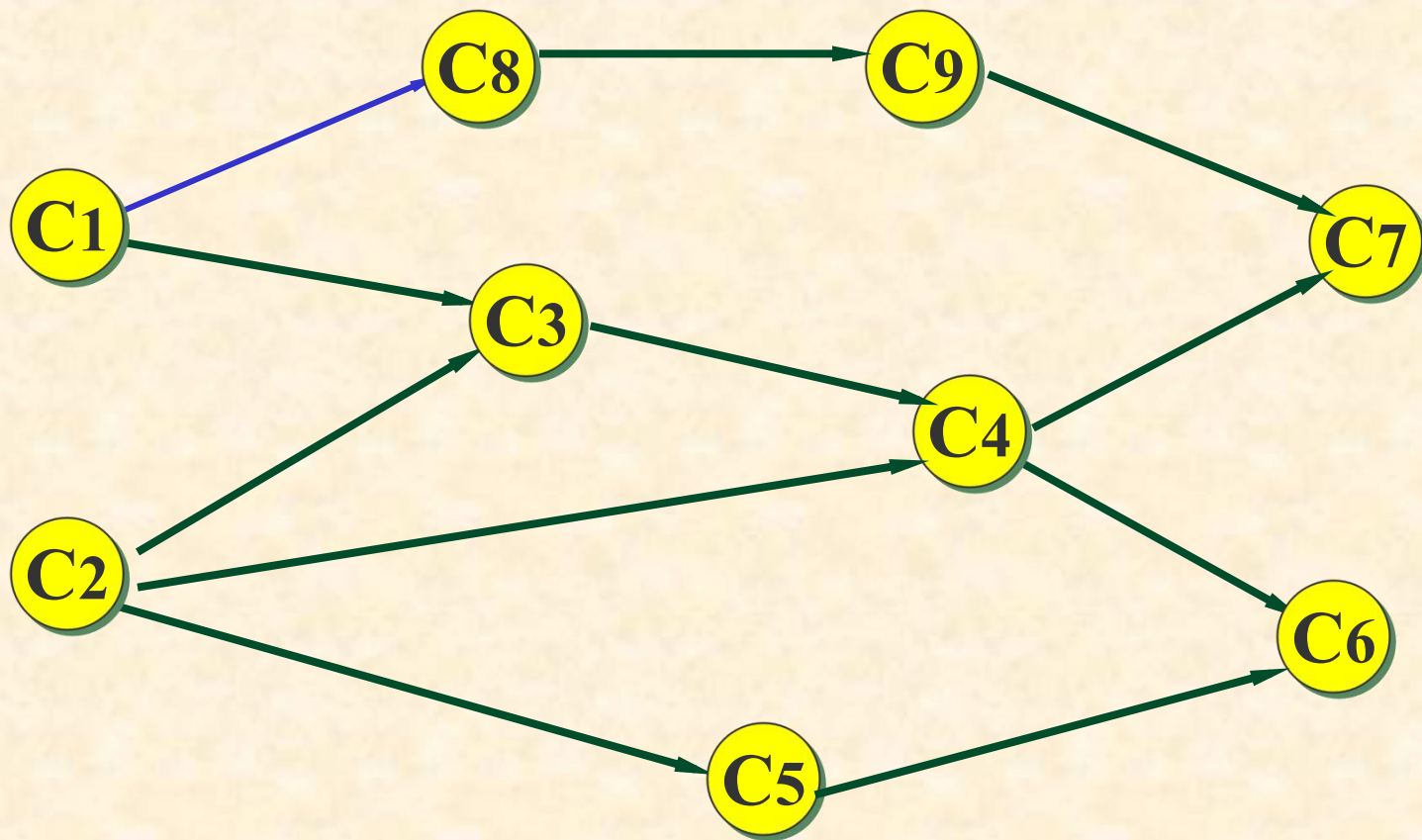
- 计划、施工过程、生产流程、程序流程等都是“工程”。除了很小的工程外，一般都把工程分为若干个叫做“活动”的子工程。完成了这些活动，这个工程就可以完成了。
- 例如，计算机专业学生的学习就是一个工程，每一门课程的学习就是整个工程的一些活动。其中有些课程要求先修课程，有些则不要求。这样在有的课程之间有领先关系，有的课程可以并行地学习。

课程代号

课程名称

先修课程

C ₁	高等数学	
C ₂	程序设计基础	
C ₃	离散数学	C ₁ , C ₂
C ₄	数据结构	C ₃ , C ₂
C ₅	高级语言程序设计	C ₂
C ₆	编译方法	C ₅ , C ₄
C ₇	操作系统	C ₄ , C ₉
C ₈	普通物理	C ₁
C ₉	计算机原理	C ₈



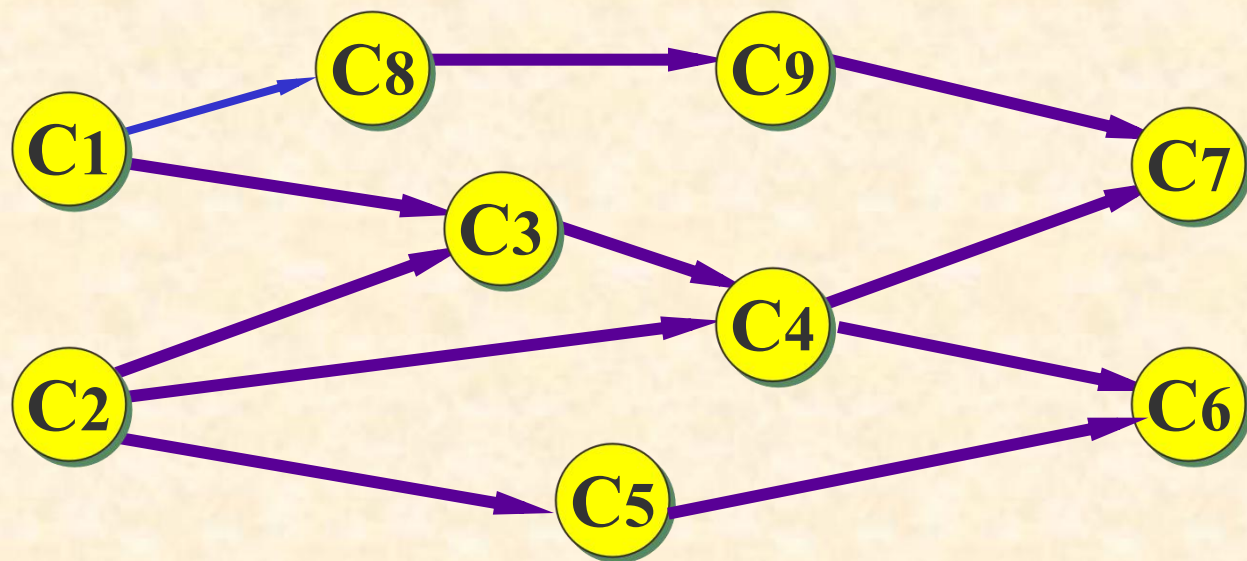
学生课程学习工程图

- 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

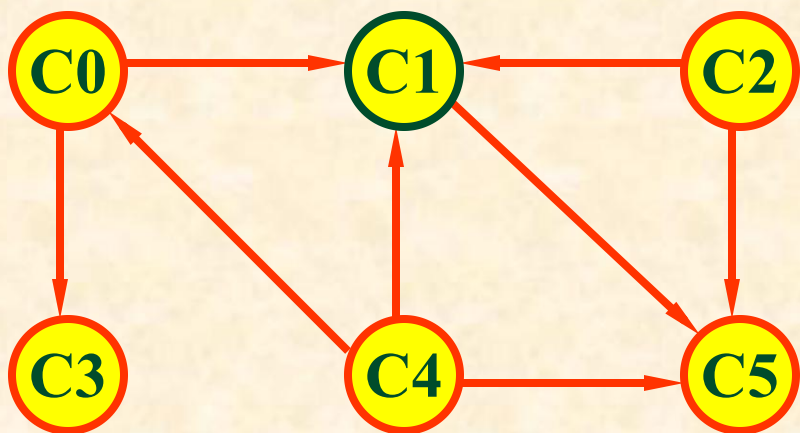
$C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$
或 $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$



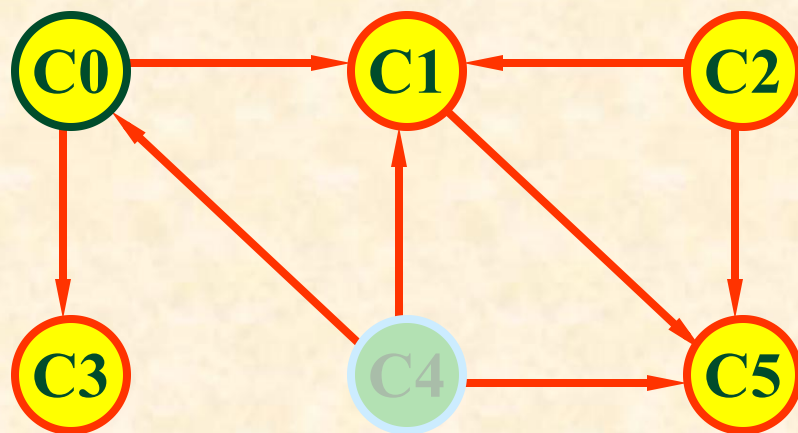
拓扑排序的方法

- 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，同时删去所有由它发出的有向边；
- 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
 - 图中还剩下一些顶点，但不存在无前驱的顶点。这时网络中必存在有向环。

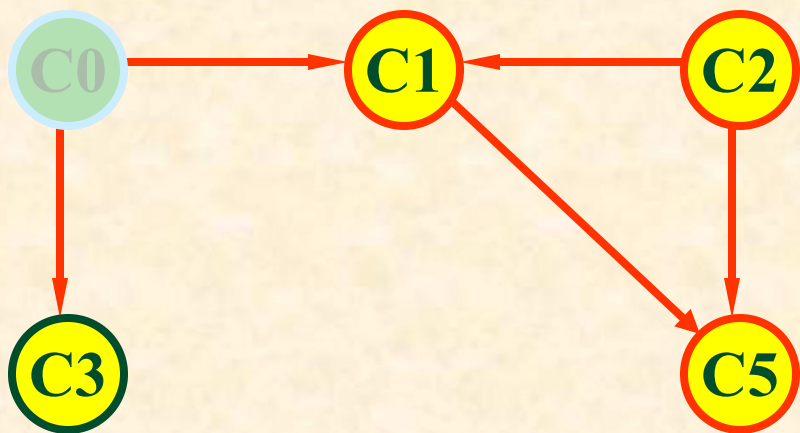
拓扑排序的过程



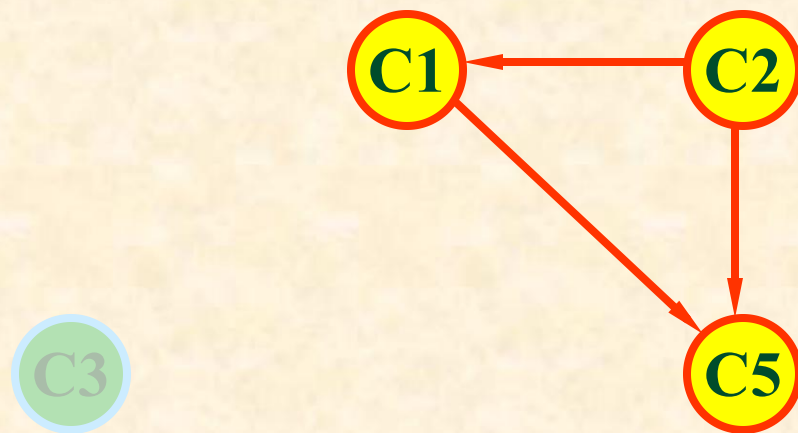
(a) 有向无环图



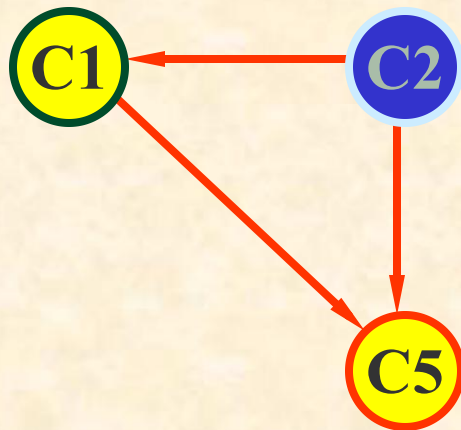
(b) 输出顶点C4



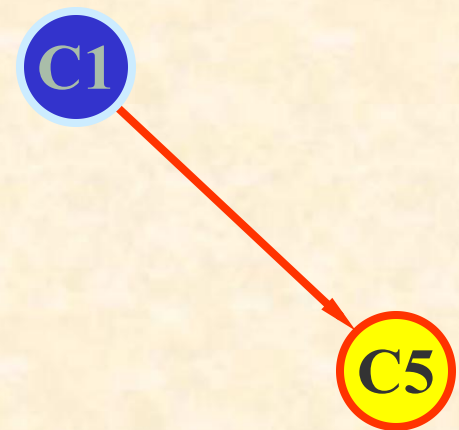
(c) 输出顶点C0



(d) 输出顶点C3



(e) 输出顶点C2



(f) 输出顶点C1

(g) 输出顶点C5



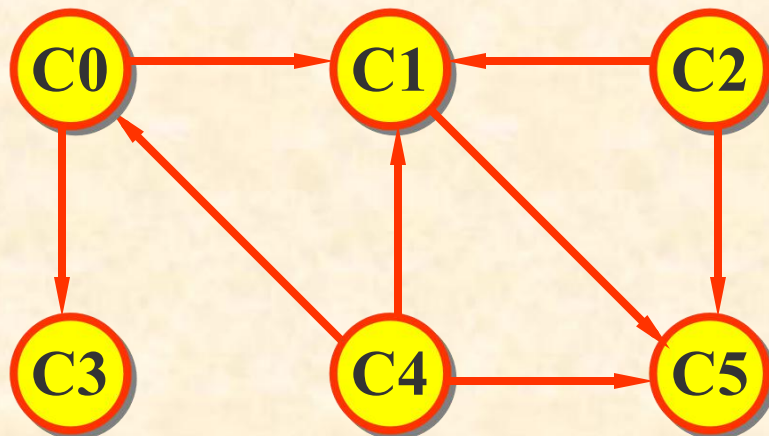
(h) 拓扑排序完成

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

inDeg data adj dest link

0	1	C0		1		3	NULL
1	3	C1		5			
2	0	C2		1		5	NULL
3	1	C3	0				
4	0	C4		0		1	
5	3	C5	0			5	NULL



- 在邻接表中增设一个数组**inDeg[]**，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**inDeg[]**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 = (EdgeNode*)malloc(sizeof(EdgeNode));  
p->dest = k;           //建立边结点  
p->link = G.VexList[j].adj;  
G.VexList[j].adj = p;  
                        //链入顶点 j 的边链表的前端  
inDeg[k]++;         //顶点 k 入度加1
```

- 在算法中, 使用一个存放入度为零的顶点的栈, 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1;
 - ✧ 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;  InitStack(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 (inDeg[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  
            printf( “网络中有回路);  
            return;  
        }  
}
```

```

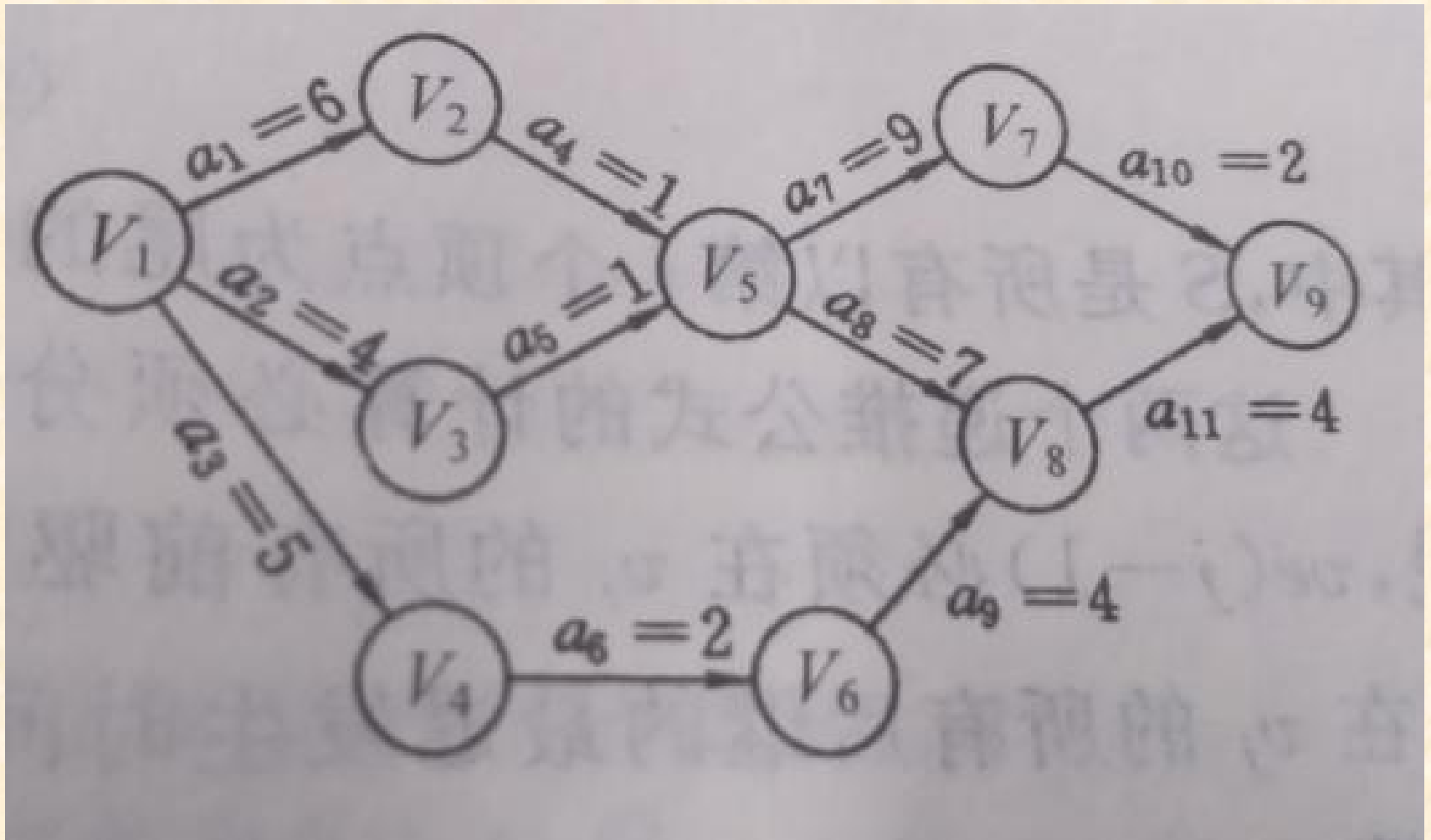
else {                                     //继续拓扑排序
    Pop( S, j );                          //退栈
    printf( j );                          //输出
    EdgeNode * p = VexList[j].firstAdj;
    while ( p != NULL ) {                 //扫描出边表
        int k = p->dest;                   //另一顶点
        if ( --inDeg [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)

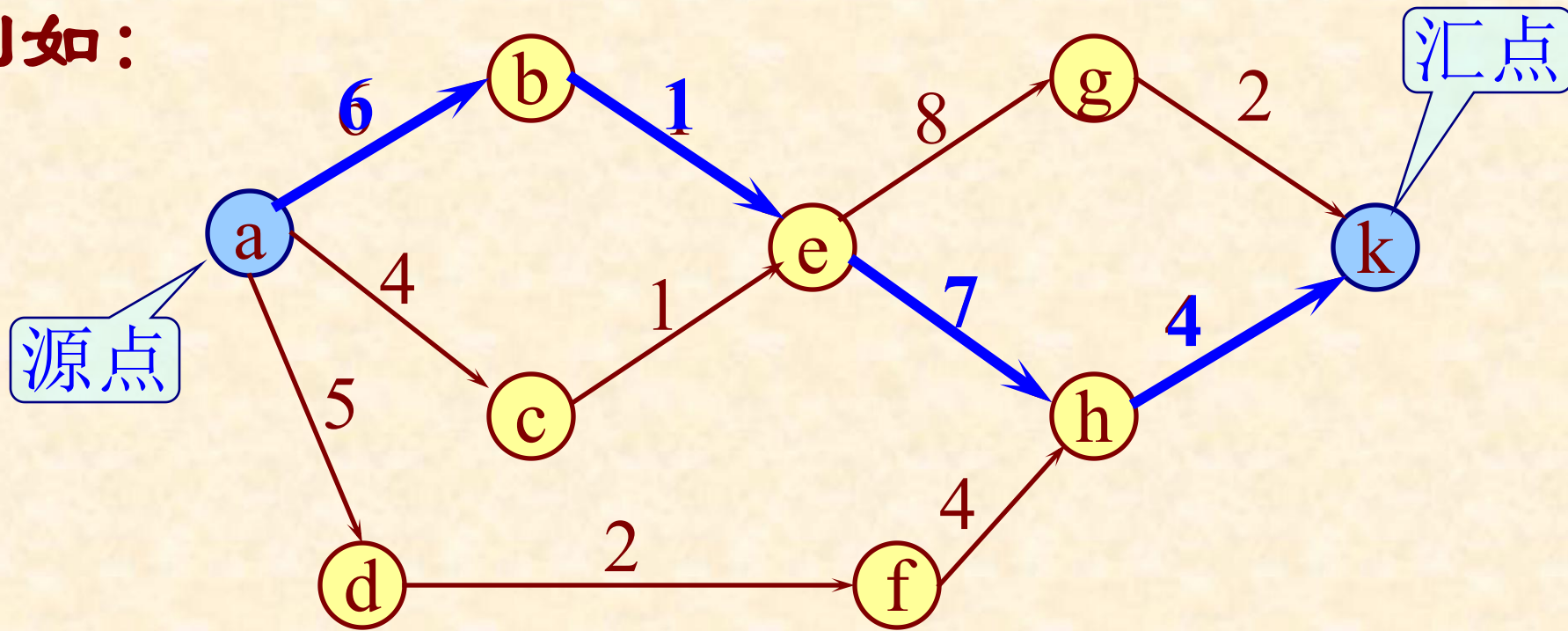
- 如果在没有环的带权有向图中, 用有向边表示一个工程中的活动 (Activity), 用边上权值表示活动持续时间 (Duration), 用顶点表示事件 (Event), 则这样的有向图叫做用边表示活动的网络, 简称 AOE (Activity On Edges) 网络。
- AOE网络在某些工程估算方面非常有用。例如, 可以使人们了解:
 - ◆ 完成整个工程至少需要多少时间(假设网络中没有环)?

- ◆ 为缩短完成工程所需的时间, 应当加快哪些活动?
- ◆ **源点**: 入度为0的点; **汇点**: 出度为0的点。



整个工程完成的时间为： 从有向图的源点到汇点的最长路径。

例如：



“关键活动” 指的是：该弧上的权值增加 将使有向图上的最长路径的长度增加。



如何求关键活动？

$ee(i)$:活动*i*最早开始时间。

$el(i)$:活动*i*最迟开始时间。

什么是“关键活动”？

该活动的最早开始时间

== 该活动的最迟开始时间



“事件(顶点)”的 最早发生时间 $ve(j)$

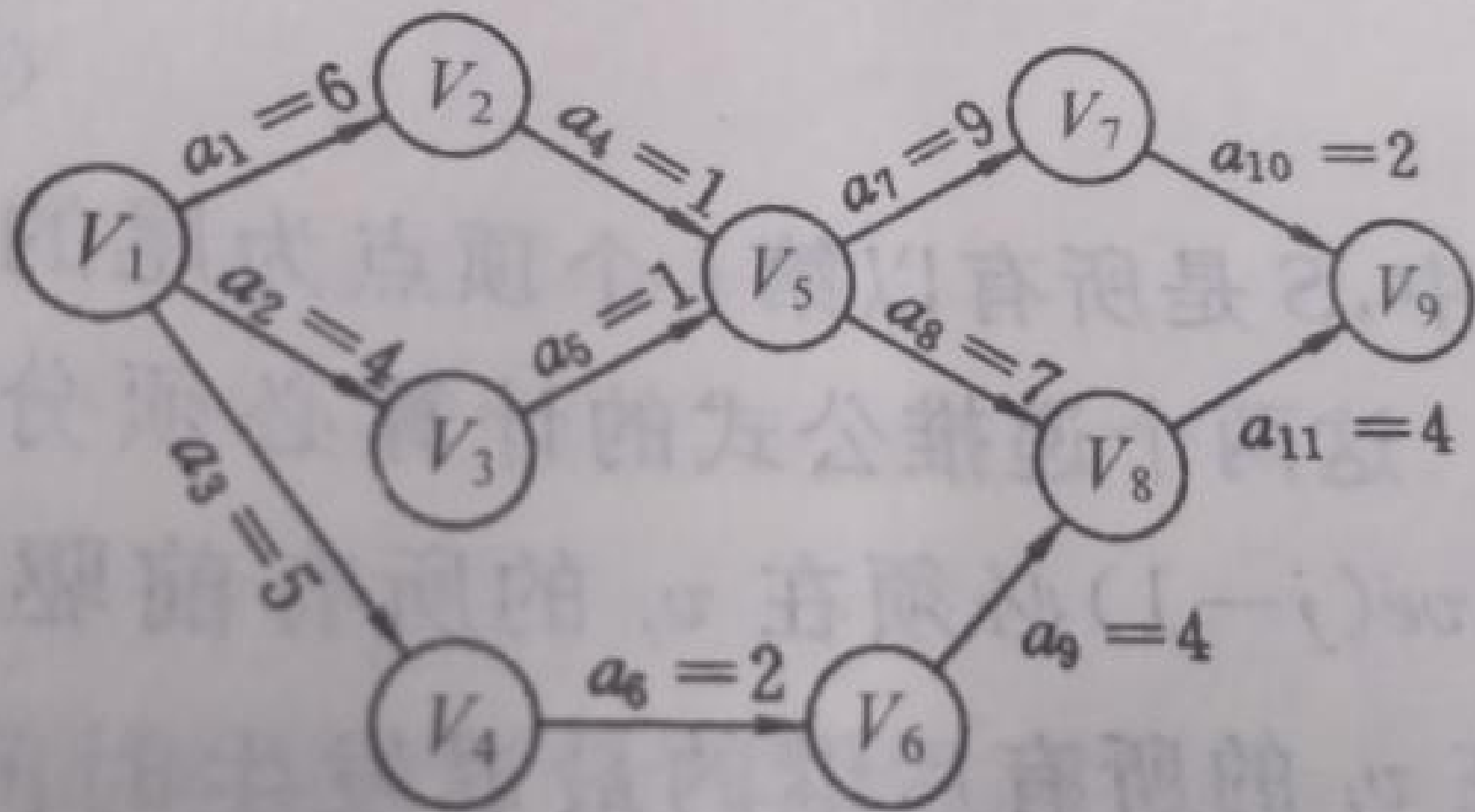
$ve(j)$ = 从源点到顶点j的最长路径长度;

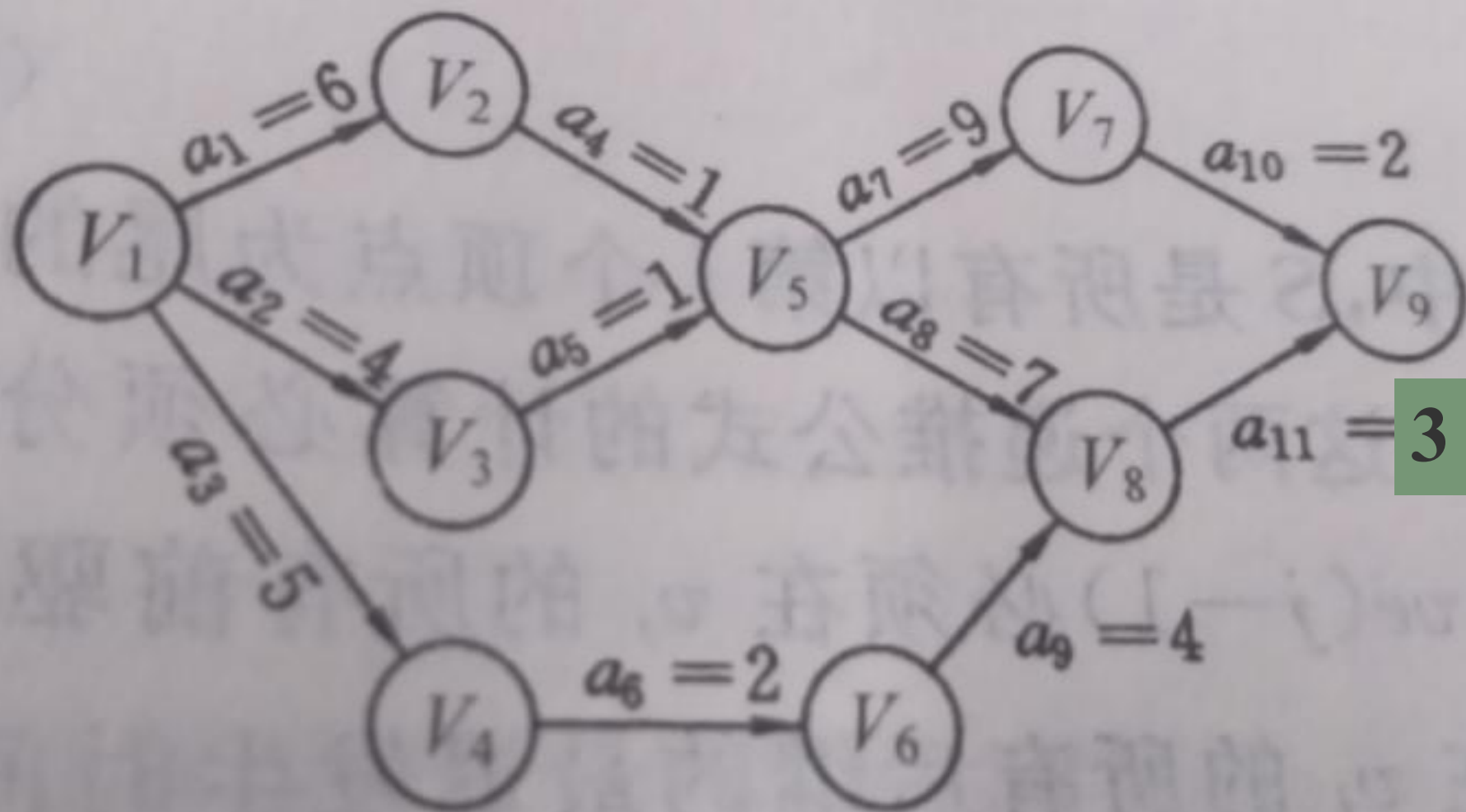


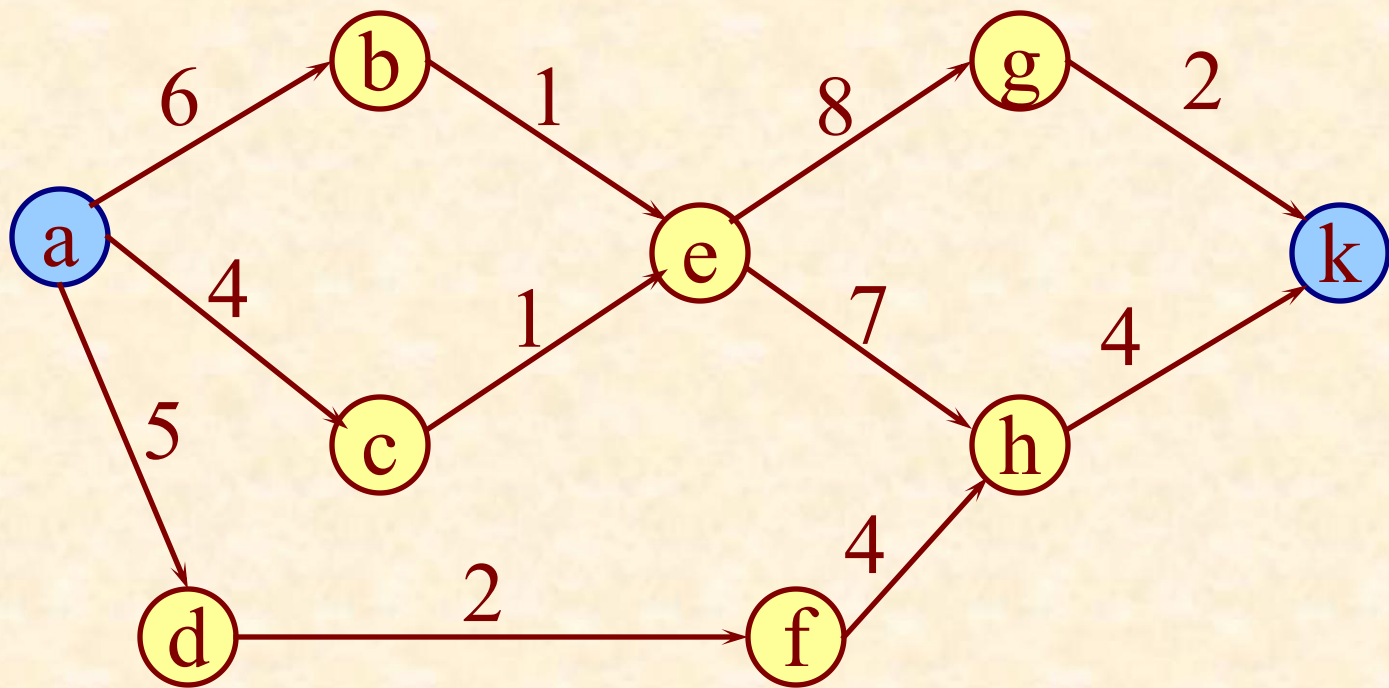
“事件(顶点)”的 最迟发生时间 $vl(k)$

$vl(k)$ = 最长路径长度 -

从顶点k到汇点的最长路径长度;







	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

拓扑有序序列: a - d - f - c - b - e - h - g - k

	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

	ab	ac	ad	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4
e	0	0	0	6	4	5	7	7	7	15	14
l	0	2	3	6	6	8	8	7	10	16	14
	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>				<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>

算法的实现要点:

显然, 求 ve 的顺序应该是按拓扑有序的次序;

而 求 vl 的顺序应该是按拓扑逆序的次序;

因为 拓扑逆序序列即为拓扑有序序列的逆序列,

因此 应该在拓扑排序的过程中,

另设一个“栈”记下拓扑有序序列。

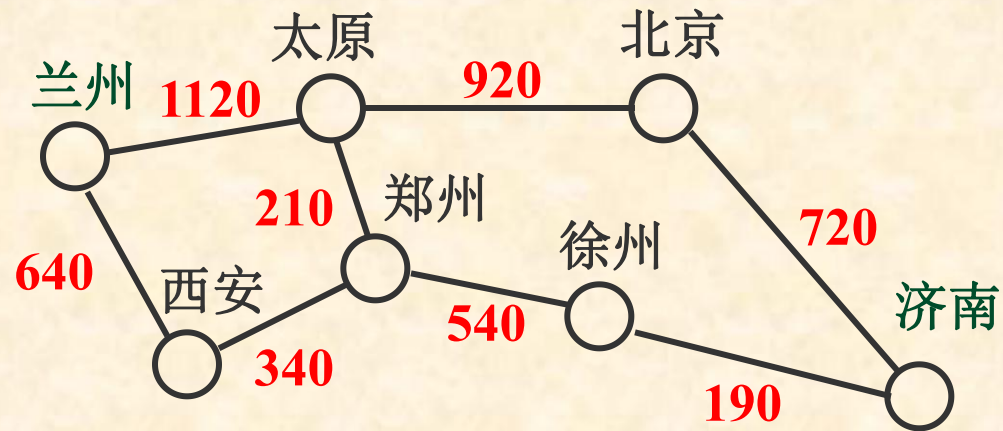


注 意

- 仅计算 $Ve[i]$ 和 $VI[i]$ 是不够的，还须计算 $ee[k]$ 和 $el[k]$ 。
- 不是任一关键活动加速一定能使整个工程提前。[P186]
- 想使整个工程提前，要考虑各个关键路径上所有关键活动。



7.6 最短路径



旅客希望停靠站越少越好，则应选择

济南——北京——太原——兰州

旅客考虑的是旅程越短越好，

济南——徐州——郑州——西安——兰州

带权图的最短路径计算问题

通常在实际中，航运、铁路、船行都具有有向性，故我们以带权有向图为例介绍最短路径算法。

带权无向图的最短路径算法也通用。

- 从单个源点到其余各顶点的最短路径算法。
- 每一对顶点之间的最短路径算法。

- 引入辅助数组 **dist[]**。它的每一个分量 **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 **dist[i]** 为该边上的权值;
 - 若从源点 v_0 到顶点 v_i 无边, 则 **dist[i]** 为 ∞ 。
- 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 **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:

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

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

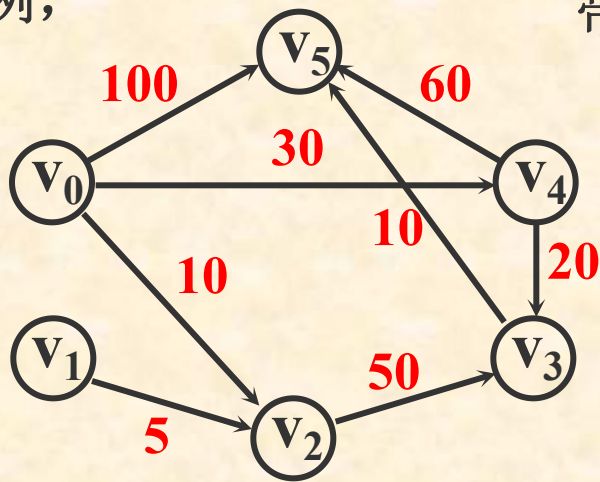
③ 修改:

$\text{dist}[i] \leftarrow \min \{ \text{dist}[i], \text{dist}[k] + \text{Edge}[k][i] \},$

对于每一个 $i \in V - S;$

④ 判断: 若 $S = V$, 则算法结束, 否则转 ②。

例,



带权邻接矩阵

	0	1	2	3	4	5
0	0	∞	∞	10	∞	30
1	∞	0	∞	∞	∞	∞
2	∞	∞	0	∞	50	∞
3	∞	∞	∞	0	∞	10
4	∞	∞	∞	20	0	60
5	∞	∞	∞	∞	∞	0

顶点 \ S	S	{v ₂ }	{v ₂ , v ₄ }	{v ₂ , v ₄ , v ₃ }	
v ₁	∞	∞	∞	∞	
v ₂	10				
v ₃	∞	60	50		
v ₄	30	30			
v ₅	100	100	90	60	
最短路径	v ₀ v ₂	v ₀ v ₄	v ₀ v ₄ v ₃	v ₀ v ₄ v ₃ v ₅	
新顶点	v ₂	v ₄	v ₃	v ₅	v ₁
路径长度	10	30	50	60	∞

每次修改都用的是最新加入集合 S 的顶点

求每一对顶点之间的最短路径

弗洛伊德算法的基本思想是：

从 V_i 到 V_j 的所有可能存在的路径中，选出一条长度最短的路径

弗洛伊德算法思想：逐步试着在原直接路径中考虑其它顶点作为中间点，如增加中间点后得到的路径较原路径长度减小，则以此新路径长度代替原值而修改矩阵元素；若增加中间点后的路径比原路径更长，就维持原相应的矩阵元素值不变。

(1) 初始状态

							A	B	C	D	E	F	
$A^{(0)} =$	0	6	∞	8	∞	∞							A
	18	0	7	∞	∞	10							B
	9	∞	0	15	∞	∞							C
	∞	∞	12	0	∞	∞							D
	∞	∞	4	∞	0	∞							E
	24	5	∞	25	∞	0							F

							A	B	C	D	E	F	
								AB		AD			A
							BA		BC			BF	B
Path ⁽⁰⁾							CA			CD			C
									DC				D
									EC				E
							FA	FB		FD			F



(2) 取顶点A为中间点, 尝试修改邻接矩阵

$$A^{(1)} =$$

A	B	C	D	E	F	
0	6	∞	8	∞	∞	A
18	0	7	26	∞	10	B
9	15	0	15	∞	∞	C
∞	∞	12	0	∞	∞	D
∞	∞	4	∞	0	∞	E
24	5	∞	25	∞	0	F

A	B	C	D	E	F	
	AB		AD			A
BA		BC	BAD		BF	B
Path ⁽¹⁾ CA	CAB		CD			C
		DC				D
		EC				E
FA	FB		FD			F

(3) 取顶点B为中间点, 尝试修改邻接矩阵

$$A^{(2)} =$$

A	B	C	D	E	F	
0	6	13	8	∞	16	A
18	0	7	26	∞	10	B
9	15	0	15	∞	25	C
∞	∞	12	0	∞	∞	D
∞	∞	4	∞	0	∞	E
23	5	12	25	∞	0	F

A	B	C	D	E	F	
	AB	ABC	AD		ABF	A
BA		BC	BAD		BF	B
Path ⁽²⁾ CA	CAB		CD		CABF	C
		DC				D
		EC				E
FBA	FB	FBC	FD			F

(4) 取顶点C为中间点, 尝试修改邻接矩阵

$A^{(3)} =$	A	B	C	D	E	F	
	0	6	13	8	∞	16	A
	16	0	7	22	∞	10	B
	9	15	0	15	∞	25	C
	21	27	12	0	∞	37	D
	13	19	4	19	0	29	E
	21	5	12	25	∞	0	F

	A	B	C	D	E	F	
Path ⁽³⁾		AB	ABC	AD		ABF	A
	BCA		BC	BCD		BF	B
	CA	CAB		CD		CABF	C
	DCA	DCB	DC			DCABFD	D
	ECA	ECAB	EC	ECD		ECABFE	E
	FBCA	FB	FBC	FD			F

(5) 取顶点D为中间点, 尝试修改邻接矩阵

$A^{(4)} =$	A	B	C	D	E	F	
	0	6	13	8	∞	16	A
	16	0	7	22	∞	10	B
	9	15	0	15	∞	25	C
	21	27	12	0	∞	37	D
	13	19	4	19	0	29	E
	21	5	12	25	∞	0	F

Path⁽⁴⁾ Path⁽⁵⁾ Path⁽⁶⁾ 同 Path⁽³⁾

1. 熟悉图的各种存储结构及其构造算法，了解实际问题的求解效率与采用何种存储结构和算法有密切联系。

2. 熟练掌握图的两搜索路径的遍历：遍历的逻辑定义、深度优先搜索和广度优先搜索的算法。

在学习中应注意图的遍历算法与树的遍历算法之间的类似和差异。

3. 应用图的遍历算法求解各种简单路径问题。

4. 理解教科书中讨论的各种图的算法。



- 图的广度优先搜索类似于树的()次序遍历。

A. 先根 B. 中根

C. 后根 D. 层次

- n 个顶点的强连通图中至少含有()。

A. $n-1$ 条有向边 B. n 条有向边

C. $n(n-1)/2$ 条有向边

D. $n(n-1)$ 条有向边

■ 任何一个无向连通图的最小生成树（
）。

- A. 只有一棵 B. 有一棵或多棵
C. 一定有多棵 D. 可能不存在

- 具有 n 个顶点的有向无环图最多可包含（ ）条有向边。

A. $n-1$

B. n

C. $n(n-1) / 2$

D. $n(n-1)$

- 图的广度优先搜索类似于树的()次序遍历。

A. 先根

B. 中根

C. 后根

D. 层次

- 设图 $G=(V, E)$, $V=\{1, 2, 3, 4\}$, $E=\{<1, 2>, <1, 3>, <2, 4>, <3, 4>\}$, 从顶点1出发, 对图G进行广度优先搜索的序列有_____种。
- 有向图G用邻接矩阵 $A[1..n, 1..n]$ 存储, 矩阵中元素值1代表有弧, 0代表无弧, 其第i行的所有元素之和等于顶点i的_____度。

- 一个连通图的生成树是该图的_____连通子图。若这个连通图有 n 个顶点，则它的生成树有_____条边。
- 在用于表示有向图的邻接矩阵中（矩阵中元素值 1 代表有弧， 0 代表无弧），对第 i 行的元素进行累加，可得到第 i 个顶点的_____度。

- 设图 $G = (V, E)$ ， $V = \{1, 2, 3, 4, 5, 6\}$ ， $E = \{<1, 2>, <1, 3>, <2, 5>, <3, 6>, <6, 5>, <5, 4>, <6, 4>\}$ 。请写出图 G 中顶点的所有拓扑序列。

- 已知一个图的顶点集V和边集G分别为:

$$V=\{0, 1, 2, 3, 4, 5, 6, 7\};$$

$$G=\{(0,1)3,(0,3)5,(0,5)18,(1,3)7,(1,4)6,(2,4)10,\\(2,7)20,(3,5)15,(3,6)12,(4,6)8,(4,7)12\};$$

按照普里姆算法从顶点**2**出发得到最小生成树，试写出在最小生成树中依次得到的各条边。

- 已知一个带权图的顶点集**V**和边集**G**分别为:

$V = \{0, 1, 2, 3, 4, 5, 6\};$

$G = \{(0, 1)19, (0, 2)10, (0, 3)14, (1, 2)6,$

$(1, 5)5, (2, 3)26, (2, 4)15, (3, 4)18,$

$(4, 5)6, (4, 6)6, (5, 6)12\};$

试根据迪克斯特拉(**Dijkstra**)算法求出从顶点**0**到其余各项点的最短路径, 在下面填写对应的路径长度。

顶点: 0 1 2 3 4 5 6

路径长度: