



# 第7章 图



# 7.1 图的术语与定义

## ● 图的定义

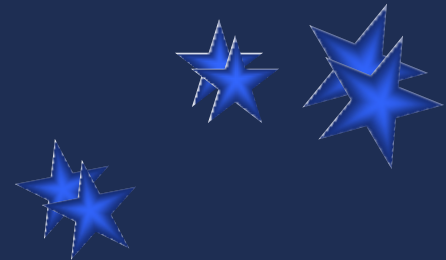
◆ 图(Graph)——图 $G$ 是由两个集合  $V(G)$  和  $E(G)$  组成的，记为  $G=(V,E)$

其中：  $V(G)$  是顶点的非空有限集

$E(G)$  是边的有限集合，边是顶点的无序对或有序对。

◆ 图的分类

- 有向图
- 无向图



# 7.1 图的术语与定义

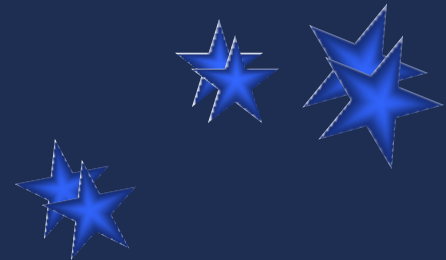
## ● 图的定义

◆ 有向图——有向图 $G$ 是由两个集合  $V(G)$  和  $E(G)$  组成的。

其中：

$V(G)$  是顶点的非空有限集。

$E(G)$  是有向边（也称弧）的有限集合，弧是顶点的有序对，记为  $\langle v, w \rangle$ ， $v$ 、 $w$  是顶点， $v$  为弧尾， $w$  为弧头。



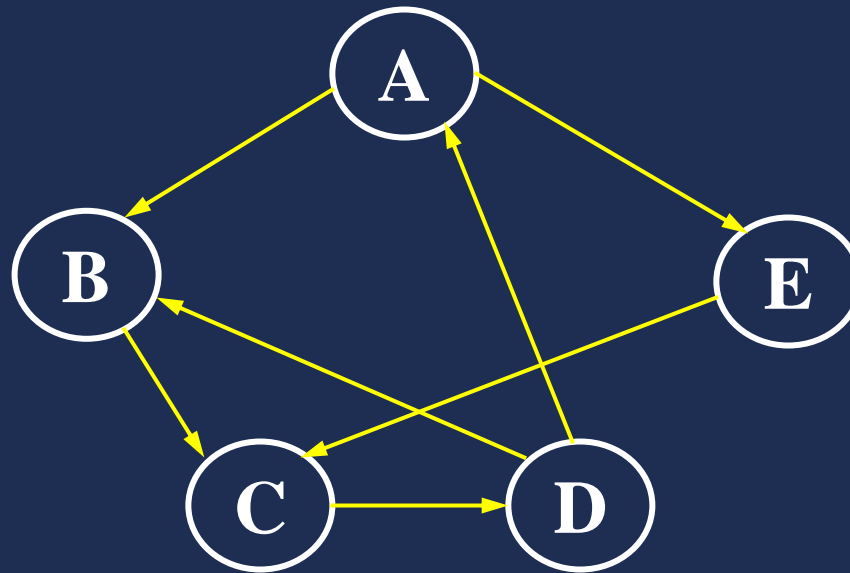
# 7.1 图的术语与定义

- 例如:

**$G1 = \langle V1, E1 \rangle$**

**$V1 = \{ A, B, C, D, E \}$**

**$E1 = \{ \langle A, B \rangle, \langle A, E \rangle, \langle B, C \rangle, \langle C, D \rangle, \langle D, B \rangle, \langle D, A \rangle, \langle E, C \rangle \}$**



# 7.1 图的术语与定义

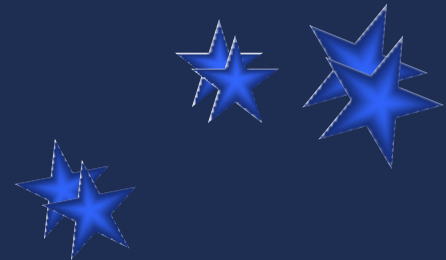
## ● 图的定义

◆ 无向图——无向图 $G$ 是由两个集合  $V(G)$  和  $E(G)$  组成的。

其中：

$V(G)$  是顶点的非空有限集。

$E(G)$  是边的有限集合，边是顶点的无序对，记为  $(v, w)$  或  $(w, v)$ ，并且  $(v, w) = (w, v)$ 。



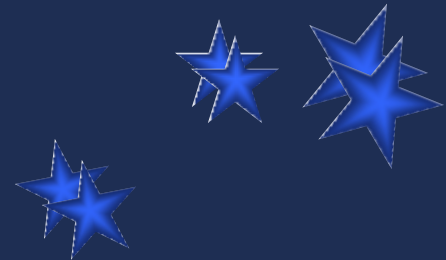
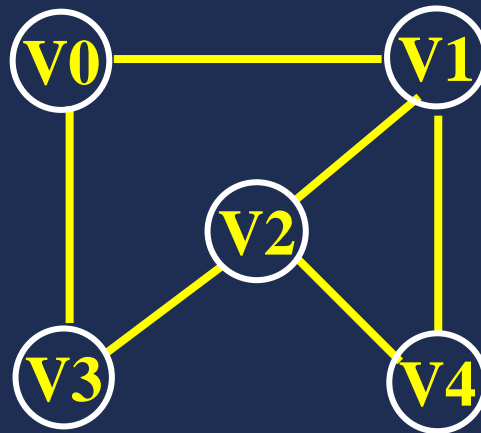
## 7.1 图的术语与定义

- 例如:

$$G2 = \langle V2, E2 \rangle$$

$$V2 = \{ v0, v1, v2, v3, v4 \}$$

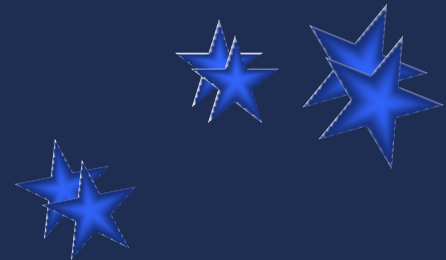
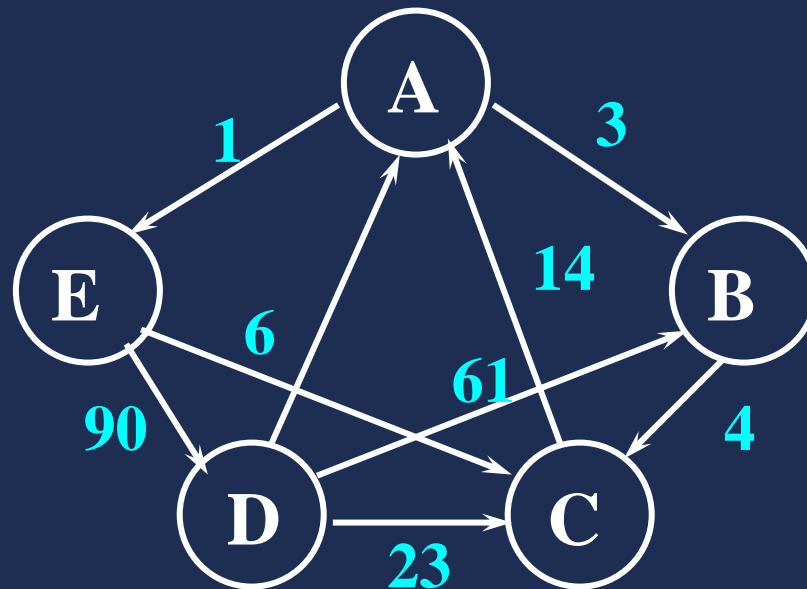
$$E2 = \{ (v0, v1), (v0, v3), (v1, v2), (v1, v4), (v2, v3), (v2, v4) \}$$



# 7.1 图的术语与定义

## 网(Network)

- ◆ 弧或边带权(Weight)的图分别称**有向网**和**无向网**



# 7.1 图的术语与定义

- 图的应用举例

例1. 交通图（公路、铁路）

顶点：地点

边：连接地点的路

例2. 电路图

顶点：元件

边：连接元件之间的线路

例3. 通讯线路图

顶点：通讯站点

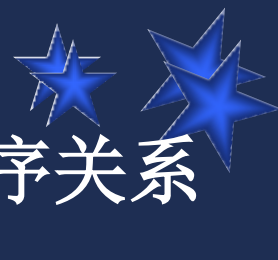
边：站点间的连线

例4. 各种流程图

如产品的生产流程图。

顶点：工序

边：各道工序之间的顺序关系





# 7.1 图的术语与定义

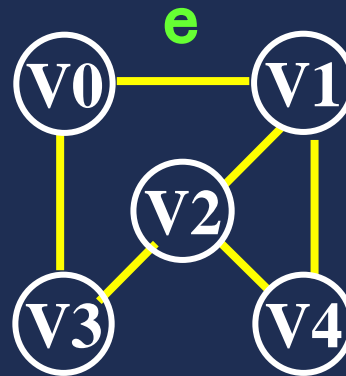
## 图的基本术语

- 邻接点及关联边

邻接点：边的两个顶点互为邻接点

关联边：若边  $e = (v, u)$ ，则称顶点  $v$ 、 $u$  关连边  $e$ 。

顶点  $V$  的度 = 与  $V$  相关联的边的数目



## 7.1 图的术语与定义

- 顶点的度、入度、出度

顶点 $v$ 的度 = 与 $v$ 相关联的边的数目  
在有向图中:

顶点 $v$ 的出度 = 以 $v$ 为起点有向边数

顶点 $v$ 的入度 = 以 $v$ 为终点有向边数

顶点 $v$ 的度 =  $v$ 的出度 +  $v$ 的入度

设图 $G$  的顶点数为  $n$ , 边数为  $e$

图的所有顶点的度数和 =  $2 \times e$

(每条边对图的所有顶点的度数和 “贡献” 2 度)



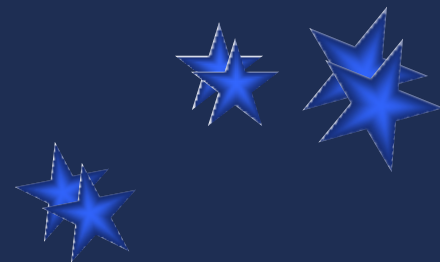
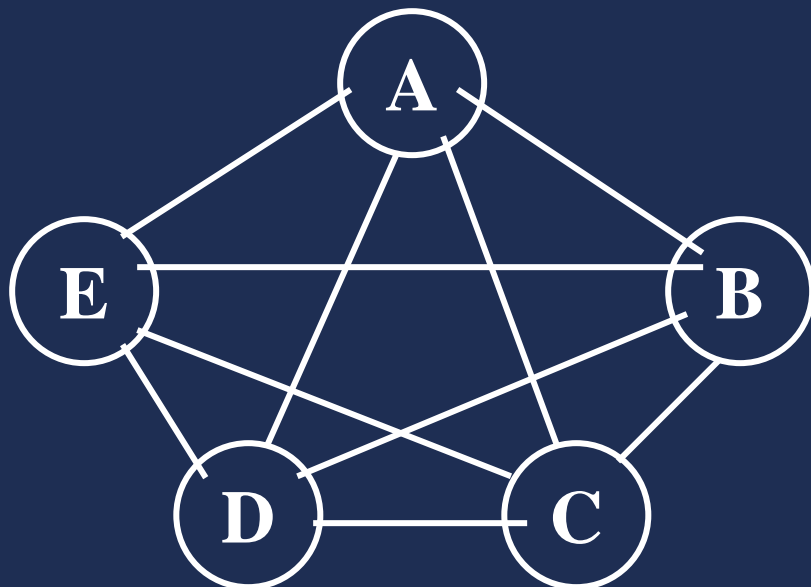
## 7.1 图的术语与定义

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**  
(Completed graph)

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**  
(Sparse graph)，否则称作**稠密图**(Dense graph)。



# 7.1 图的术语与定义

- 路径、回路

假设 $v_1, v_2, \dots, v_k$ 为图 $G=(V, E)$ 的顶点序列，若 $G$ 为无向图，则有 $(v_i, v_{i+1}) \in E$ ；或者 $G$ 为有向图，则有 $\langle v_i, v_{i+1} \rangle \in E$ ；其中 $(1 \leq i < k)$ ， $v=v_1$ ， $u=v_k$ ，则称该序列是从顶点 $v$ 到顶点 $u$ 的路径；路径上边的数目称作路径长度。

若第一个顶点和最后一个顶点相同，则称该序列为回路。

序列中顶点不重复出现的路径定义为简单路径  
构成回路的简单路径称其为简单回路

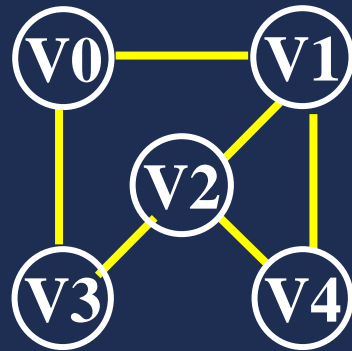


# 7.1 图的术语与定义

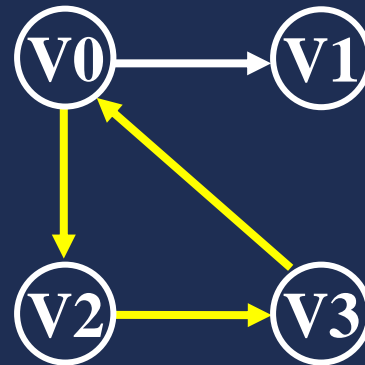
- 例如

在图G1中,  $V_0, V_1, V_2, V_3$  是  $V_0$  到  $V_3$  的路径, 而且是简单路径;  $V_0, V_1, V_2, V_3, V_0$  是简单回路。

在图G2中,  $V_0, V_2, V_3$  是  $V_0$  到  $V_3$  的简单路径;  $V_0, V_2, V_3, V_0$  是简单回路。



无向图G1



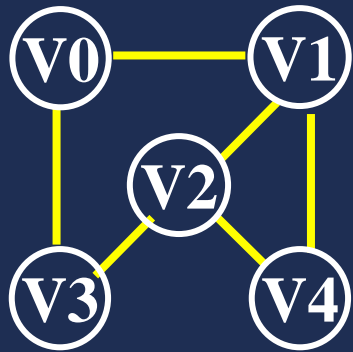
有向图G2



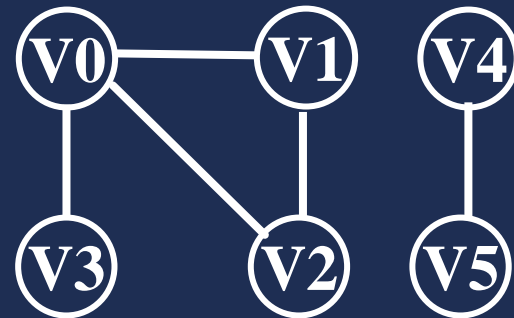
# 7.1 图的术语与定义

## ● 连通图

在无向图  $G = \langle V, E \rangle$  中，若对任何两个顶点  $v$ 、 $u$  都存在从  $v$  到  $u$  的路径，则称  $G$  是连通图。



G1: 连通图



G2: 非连通图

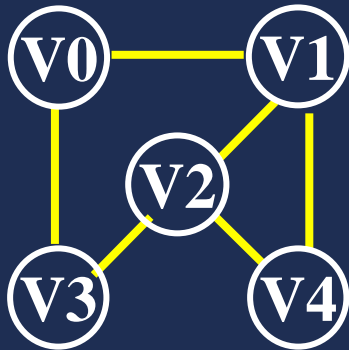


# 7.1 图的术语与定义

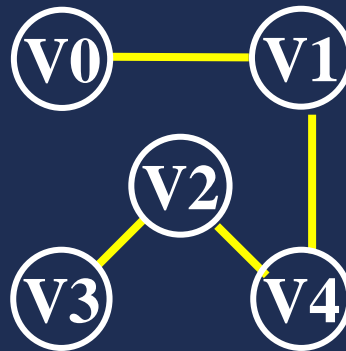
## ● 子图

设有两个图  $G=(V, E)$ ,  $G_1=(V_1, E_1)$ ,  
若  $V_1 \subseteq V$  且  $E_1 \subseteq E$ , 则称  $G_1$  是  $G$  的子图。

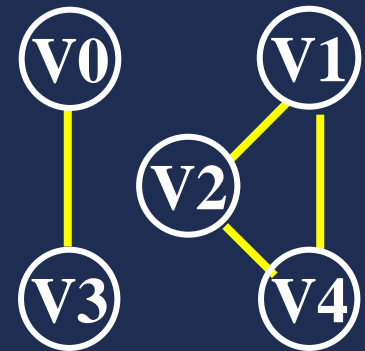
例



(a)



(b)



(c)

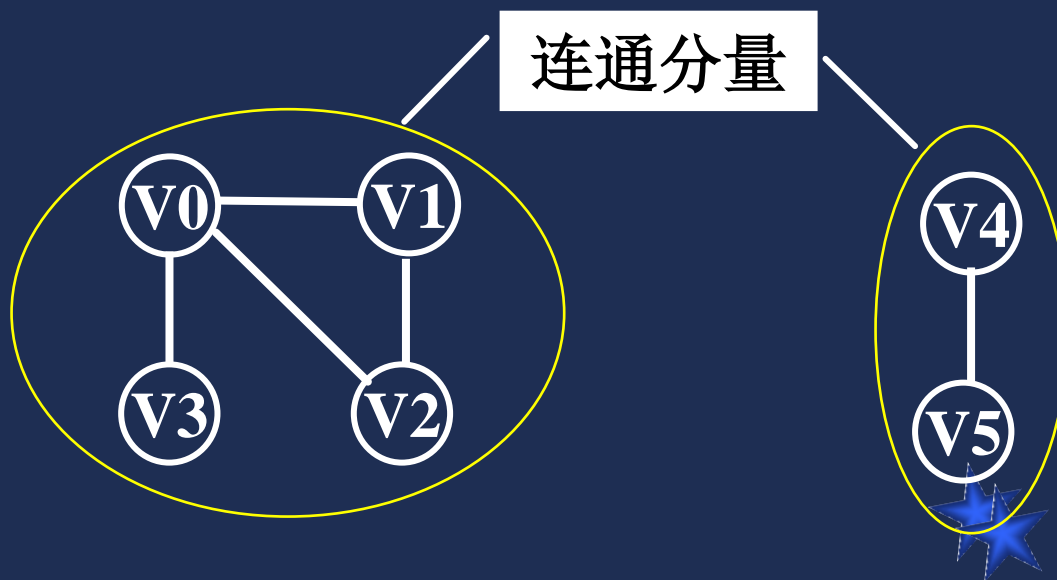
(b)、(c) 是 (a) 的子图

## 7.1 图的术语与定义

- 若无向图为非连通图，则各个极大连通子图称作此图的连通分量。

**极大连通子图**含义：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，子图不再连通。

非连通图



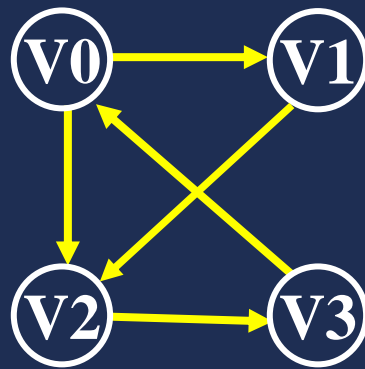


## 7.1 图的术语与定义

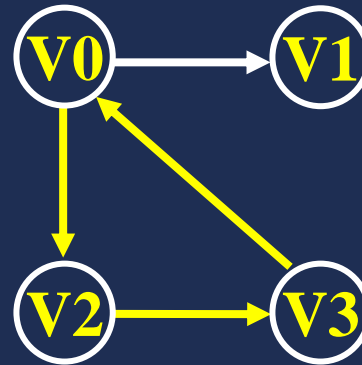
### ● 强连通图

在有向图  $G = \langle V, E \rangle$  中，若对任何两个顶点  $v$ 、 $u$  都存在从  $v$  到  $u$  的路径，则称  $G$  是强连通图。

强连通图



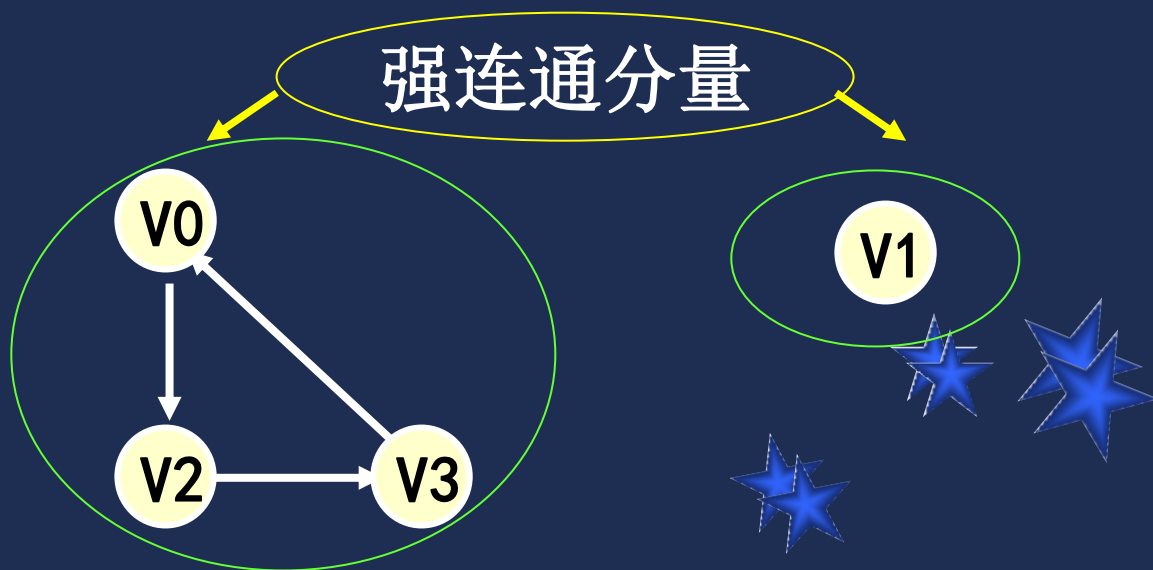
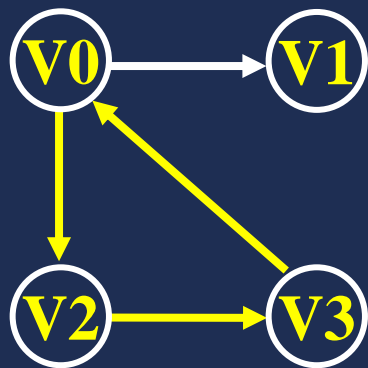
非强连通图



## 7.1 图的术语与定义

- 若有向图为非强连通图，它的各个极大强连通子图称为它的强连通分量。

**极大强连通子图**含义：该子图是  $D$  的强连通子图，将  $D$  的任何不在该子图中的顶点加入，子图不再是强连通的。

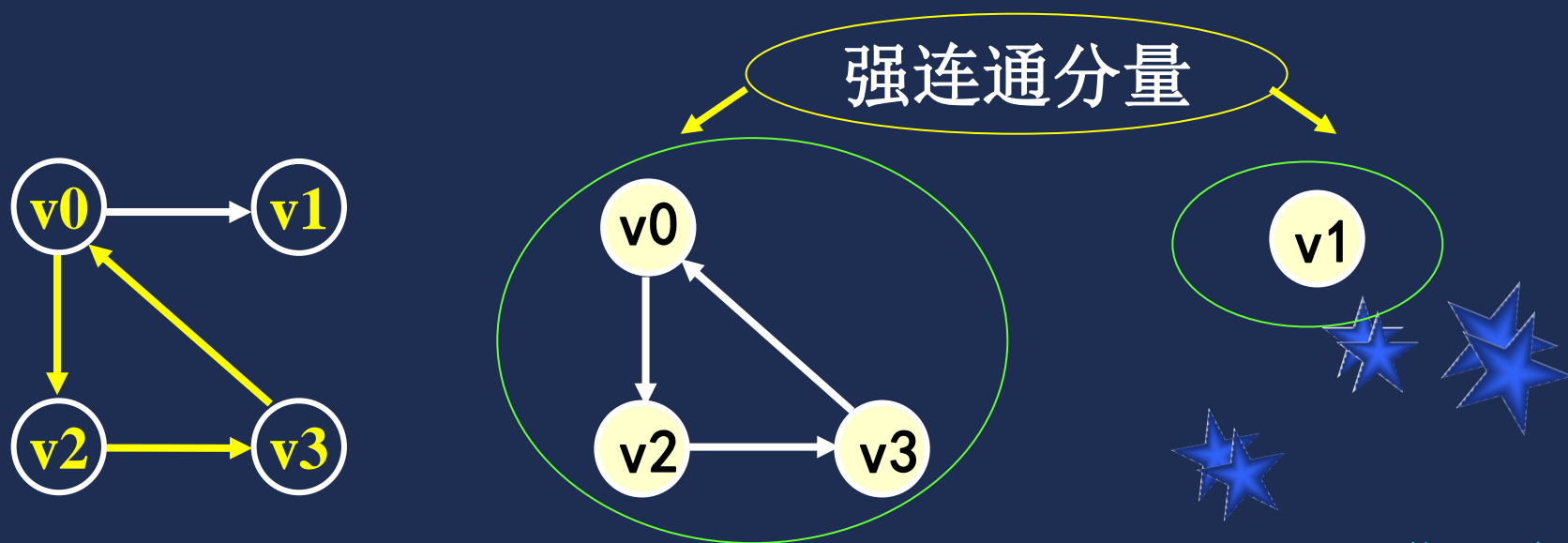


## 7.1 图的术语与定义

- 连通分图（强连通分量）

有向图 $D$ 的极大强连通子图称为 $D$ 的强连通分量。

**极大强连通子图**含义：该子图是  $D$  的强连通子图，将  $D$  的任何不在该子图中的顶点加入，子图不再是强连通的。



# 7.1 图的术语与定义

## ● 生成树

连通图  $G$  中，包含所有顶点的极小连通子图称为  $G$  的生成树。

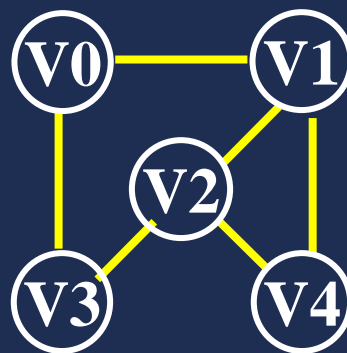
**极小连通子图** 含义：该子图是  $G$  的连通子图，在该子图中删除任何一条边，子图不再连通。

若  $T$  是  $G$  的生成树当且仅当  $T$  满足如下条件：

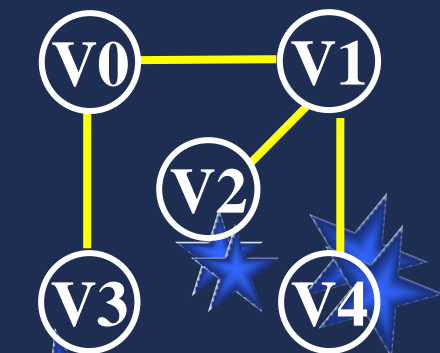
$T$  包含  $G$  的所有顶点

$T$  是  $G$  的连通子图

$T$  中无回路



连通图  $G_1$



$G_1$  的生成树

# 7.1 图的术语与定义

## ● 图的基本操作

- ◆ **CreatGraph(&G, V, VR)**

  - 按定义(**V, VR**) 构造图

- ◆ **DestroyGraph(&G)**

  - 销毁图

对顶点的访问

- ◆ **LocateVex(G, u)**

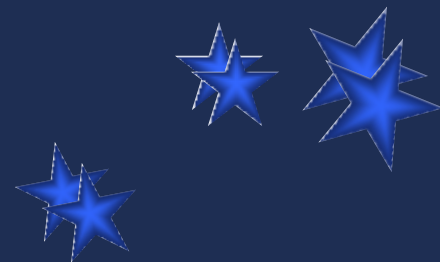
  - 若**G**中存在顶点**u**，则返回该顶点在图中“位置”；否则，返回其它信息

- ◆ **GetVex(G, v)**

  - 返回 **v** 的值

- ◆ **PutVex(&G, v, value)**

  - 对 **v** 赋值**value**



# 7.1 图的术语与定义

## ● 图的基本操作 对邻接点的操作

### ◆ FirstAdjVex( $G, v$ )

- 返回  $v$  的“第一个邻接点”。
- 若该顶点在  $G$  中没有邻接点，则返回“空”

### ◆ NextAdjVex( $G, v, w$ )

- 返回  $v$  的（相对于  $w$  的）“下一个邻接点”。
- 若  $w$  是  $v$  的最后一个邻接点，则返回“空”。

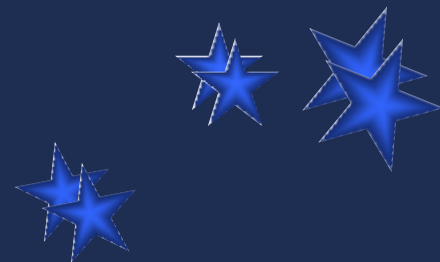
## 插入或删除顶点操作

### ◆ InsertVex(& $G, v$ )

- 在图  $G$  中增添新顶点  $v$

### ◆ DeleteVex(& $G, v$ )

- 删除  $G$  中顶点  $v$  及其相关的弧



# 7.1 图的术语与定义

## ● 图的基本操作

### 插入或删除弧操作

#### ◆ DeleteVex(&G, v)

- 删除G中顶点v及其相关的弧

#### ◆ DeleteArc(&G, v, w)

- 在G中删除弧 $\langle v, w \rangle$
- 若G是无向的, 则还删除对称弧 $\langle w, v \rangle$

### 遍历

#### ◆ DFSTraverse(G, v, Visit())

- 从顶点v起深度优先遍历图G
- 并对每个顶点调用函数Visit一次且仅一次

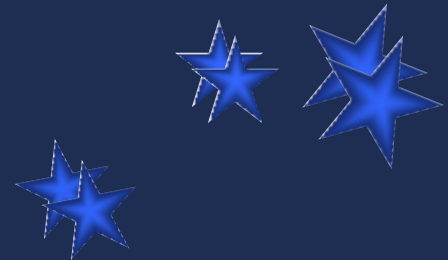
#### ◆ BFSTraverse(G, v, Visit())

- 从顶点v起广度优先遍历图G,
- 并对每个顶点调用函数Visit一次且仅一次



## 7.1 图的基本操作

- 1、结构的建立和销毁
- 2、对顶点的访问操作
- 3、对邻接点的操作
- 4、插入或删除顶点
- 5、插入和删除弧
- 6、遍历





# 7.1 图的基本操作

## 结构的建立和销毁

**CreatGraph(&G, V, VR):**

- ◆ 按定义(V, VR) 构造图

**DestroyGraph(&G):**

- ◆ 销毁图

## 对顶点的访问操作

**LocateVex(G, u);**

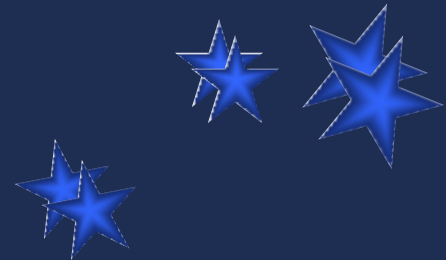
- ◆ 若G中存在顶点u，则返回该顶点在图中“位置”；
- ◆ 否则返回其它信息

**GetVex(G, v);**

- ◆ 返回 v 的值

**PutVex(&G, v, value);**

- ◆ 对 v 赋值value



# 7.1 图的基本操作

## 对邻接点的操作

**FirstAdjVex(G, v);**

- ◆ 返回 **v** 的“第一个邻接点”。
- ◆ 若该顶点在 **G** 中没有邻接点，则返回“空”

**NextAdjVex(G, v, w);**

- ◆ 返回 **v** 的（相对于 **w** 的）“下一个邻接点”。
- ◆ 若 **w** 是 **v** 的最后一个邻接点，则返回“空”。

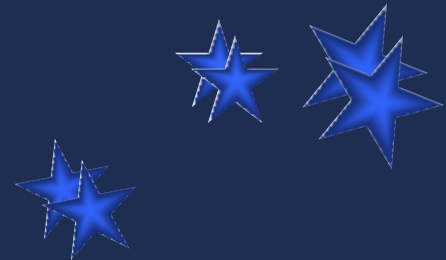
## 插入或删除顶点

**InsertVex(&G, v);**

- ◆ 在图**G**中增添新顶点**v**

**DeleteVex(&G, v);**

- ◆ 删除**G**中顶点**v**及其相关的弧



# 7.1 图的基本操作

## 插入和删除弧

DeleteVex(&G, v);

- ◆ 删除**G**中顶点**v**及其相关的弧

DeleteArc(&G, v, w);

- ◆ 在**G**中删除弧**<v,w>**
- ◆ 若**G**是无向的，则还删除对称弧**<w,v>**

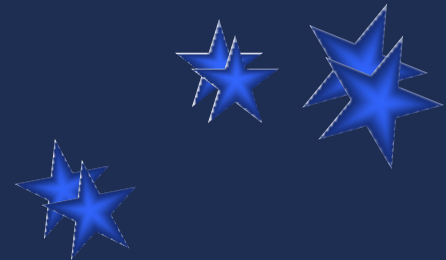
## 遍历

DFSTraverse(G, v, Visit());

- ◆ 从顶点**v**起深度优先遍历图**G**
- ◆ 并对每个顶点调用函数**Visit**一次且仅一次

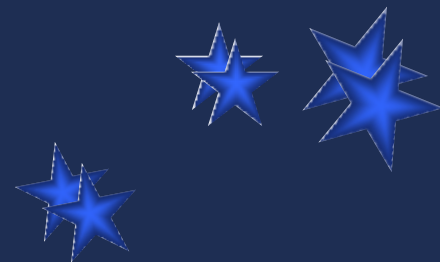
BFSTraverse(G, v, Visit());

- ◆ 从顶点**v**起广度优先遍历图**G**,
- ◆ 并对每个顶点调用函数**Visit**一次且仅一次



## 7.2 图的存储表示

- 1、图的数组(邻接矩阵)存储表示
- 2、图的邻接表存储表示
- 3、有向图的十字链表存储表示
- 4、无向图的邻接多重表存储表示



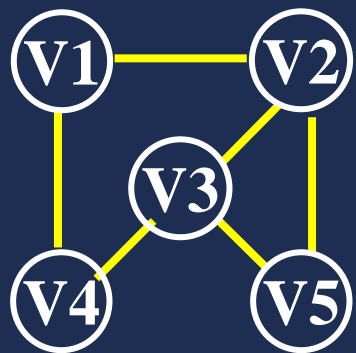
## 7.2 图的存储结构

### 一、数组表示法（邻接矩阵表示）

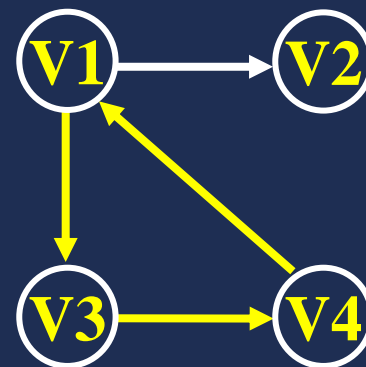
**邻接矩阵：**  $G$  的邻接矩阵是满足如下条件的  $n$  阶矩阵：

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

在数组表示法中，用邻接矩阵表示顶点间的关系



$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$v_i$  的度？

第  $i$  行 (列) 1 的个数。

•  $v_i$  的出度？

•  $v_i$  的入度？

第  $i$  行 1 的个数

第  $i$  列 1 的个数。

## 7.2 图的存储结构

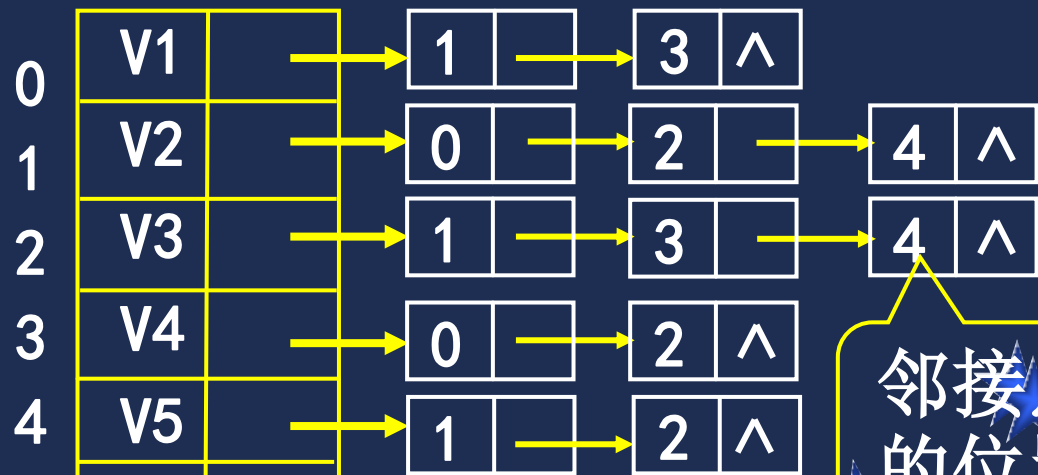
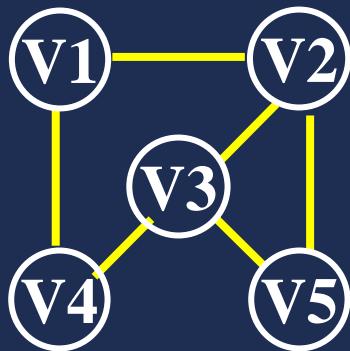
### 二、邻接表

邻接表是图的链式存储结构

#### 1、无向图的邻接表

顶点：通常按编号顺序将顶点数据存储在在一维数组中；

边节点：对于每个顶点，用线性边节点链表存储关联邻接点编号。



邻接点的位置

共有多少边节点？

$2 * e$

## 7.2 图的存储结构

- typedef struct ArcNode // 边结点定义 

adjvex	next
--------	------

  
{ int adjvex; // 邻接点域,  
    // 存放与Vi邻接的点在表头数组中的位置  
    struct ArcNode \*next; // 链域, 下一条边或弧  
} ArcNode;
- typedef struct tnode // 顶点结点定义 

vexdata	firstarc
---------	----------

  
{ int vexdata; // 存放顶点信息  
    ArcNode \*firstarc; // 指向第一个边或弧  
} VNode, AdjList [ MAX\_VERTEX\_NUM ] ;
- typedef struct // 图的定义  
{ AdjList vertices;  
    int vexnum, arcnum; // 顶点数和弧数  
    int kind; // 图的种类  
}

## 7.2 图的存储结构

- 无向图的邻接表的特点

- 1) 顶点 $v$ 的度：等于 $v$ 对应线性链表的长度；
- 2) 判定两顶点 $v$ ， $u$ 是否邻接：要看 $v$ 对应线性链表中是否存在 $u$ 。
- 3) 在 $G$ 中增减边：要在两个单链表插入、删除结点；
- 4) 设存储顶点的一维数组大小为  $m$  ( $m \geq$  图的顶点数 $n$ )，图的边数为  $e$ ， $G$  占用存储空间为： $m$ 个点+ $2 * e$ 个表节点。 $G$  占用存储空间与 $G$ 的顶点数、边数均有关；适用于边稀疏的图。





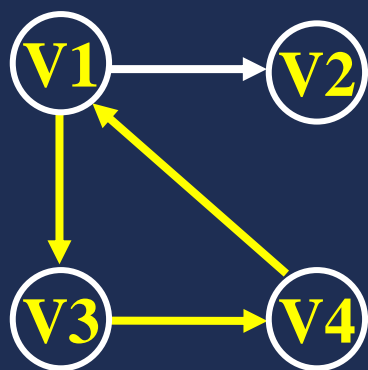
## 7.2 图的存储结构

### 二、邻接表

#### 2、有向图的邻接表

顶点：用一维数组存储（按编号顺序）

以同一顶点为起点的弧：用线性出边节点链表存储弧头位置



	vexdata	firstarc	adjvex	next
1	v1	→	3	→ 2   ^
2	v2	^		
3	v3	→	4	→ ^
4	v4	→	1	→ ^

弧头的  
位置

顶点  $i$  的出度？

顶点  $i$  的出边表长度  
出边表

共有多少边节点？

e

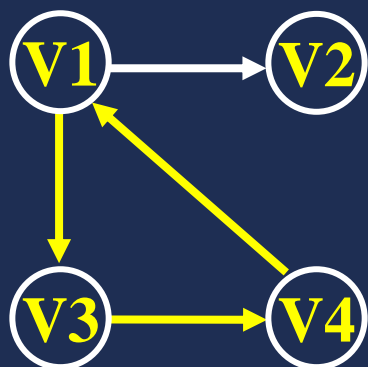
## 7.2 图的存储结构

### 二、邻接表

#### 3、有向图的逆邻接表

顶点：用一维数组存储（按编号顺序）

以同一顶点为终点的弧：用记录弧尾位置的线性入边节点链表存储。



vexdata      firstarc

1	v1	→	4   ^
2	v2	→	1   ^
3	v3	→	1   ^

顶点  $i$  的入度?

顶点  $i$  的入边表长度

共有多少边节点?

e

入边表

弧尾的  
位置

## 7.2 图的存储结构

### 三、有向图的十字链表表示法

弧结点:

tailvex	headvex	hlink	tlink
---------	---------	-------	-------

```
typedef struct ArcBox
```

```
{ int tailvex, headvex; // 弧尾、弧头在表头数组中位置
```

```
    struct arcnode *hlink; // 指向弧头相同的下一条弧
```

```
    struct arcnode *tlink; // 指向弧尾相同的下一条弧
```

```
} ArcBox;
```

顶点结点:

data	firstin	firstout
------	---------	----------

```
typedef struct VexNode
```

```
{ VertexType data; // 存与顶点有关信息
```

```
    ArcBox *firstin; // 指向以该顶点为弧头的第1个弧结点
```

```
    ArcBox *firstout; // 指向以该顶点为弧尾的第1个弧结点
```

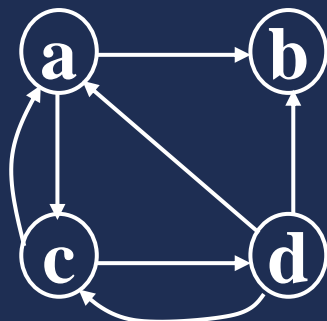
```
} VexNode;
```

```
VexNode OLGraph[M];
```

## 7.2 图的存储结构

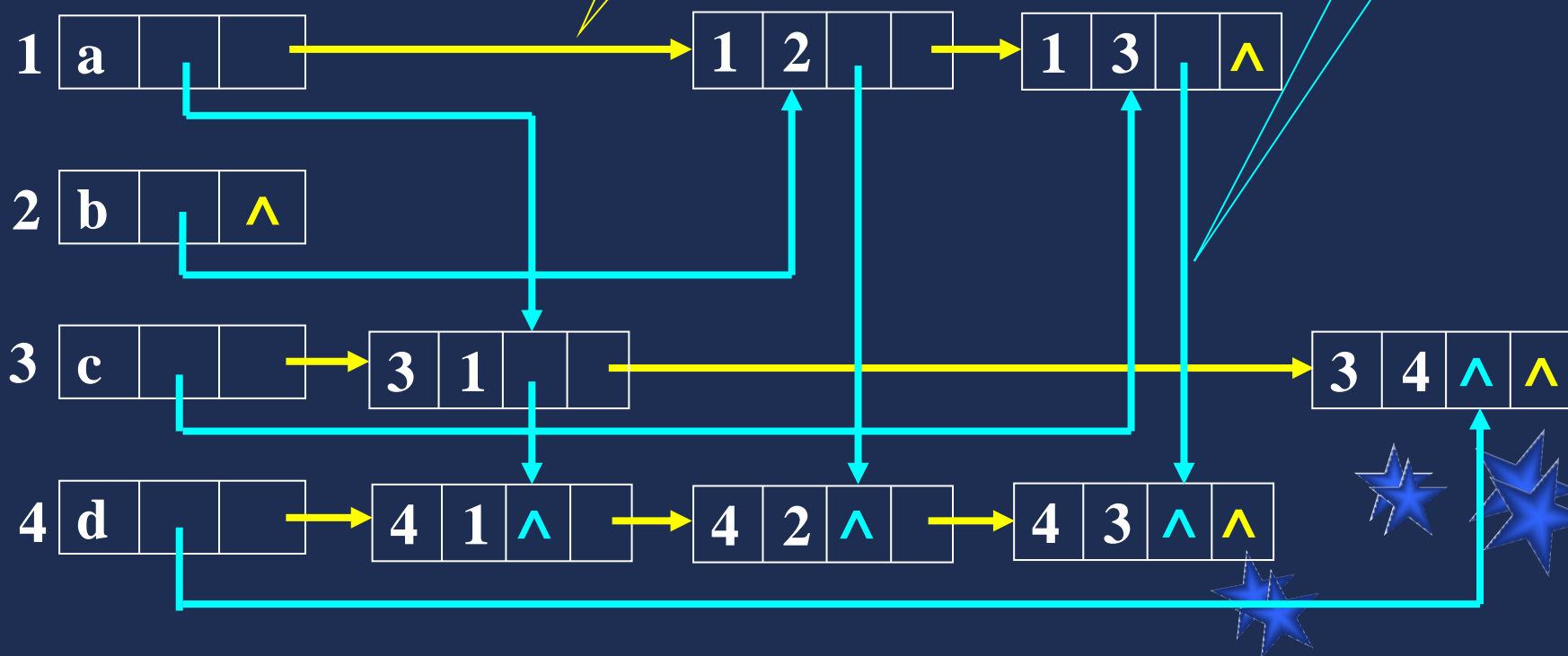
### 三、有向图的十字链表表示法

例



相同  
弧尾

相同  
弧头



## 7.2 图的存储结构

### 四、无向图的邻接多重表表示法

边结点:

```
typedef struct node
```

```
{ VisitIf mark; // 标志域, 记录是否已经搜索过
```

```
  int ivex, jvex; // 该边依附的两个顶点在表头数组中位置
```

```
  struct EBox * ilink, * jlink;
```

//分别指向依附于ivex和jvex的下一条边

```
} EBox;
```

mark	ivex	ilink	jvex	jlink
------	------	-------	------	-------

顶点结点:

```
typedef struct VexBox
```

```
{ VertexType data;
```

```
  EBox * firstedge;
```

```
} VexBox;
```

```
VexBox AMLGraph[M];
```

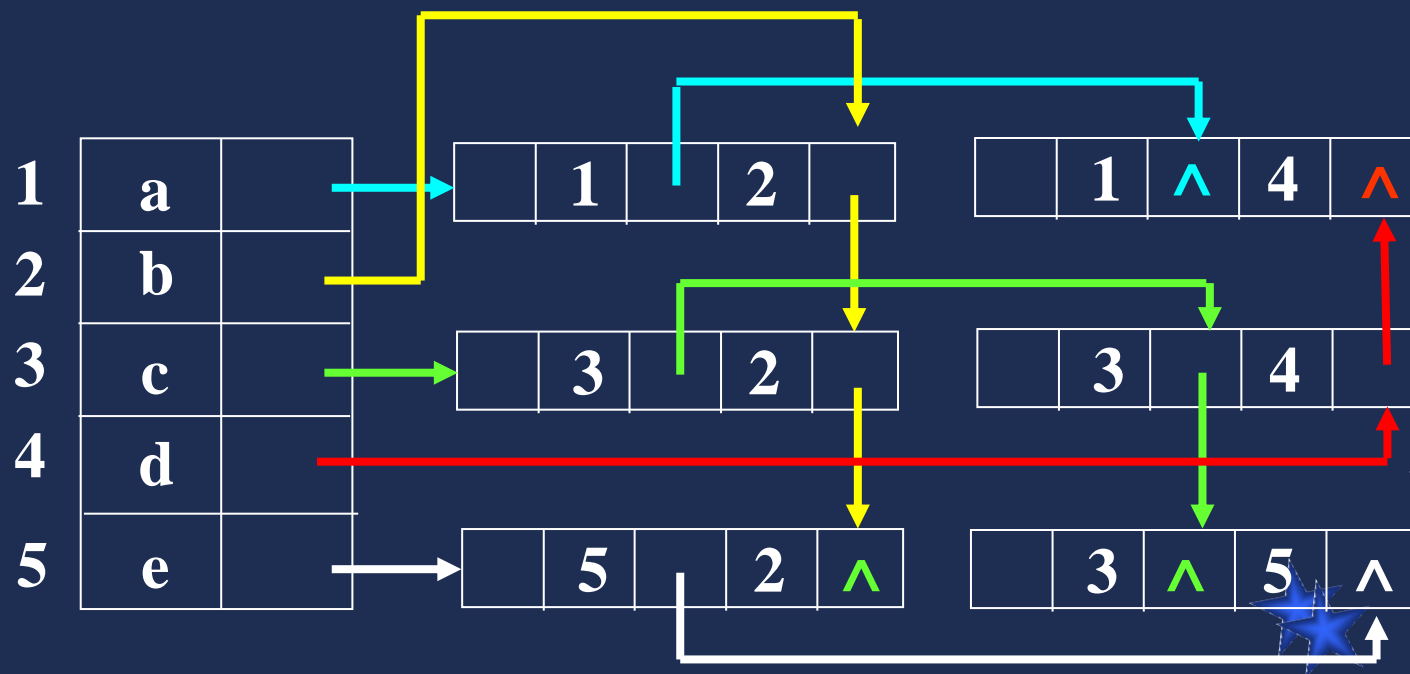
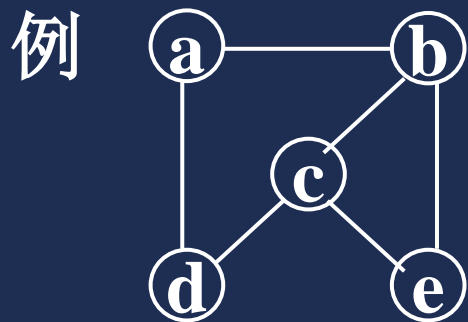
// 存与顶点有关的信息

// 指向第一条依附于该顶点的边

data	firstedge
------	-----------

## 7.2 图的存储结构

### 四、无向图的邻接多重表表示法



## 7.3 图的遍历

- 图的遍历

访遍图中所有的顶点，并且使图中的每个顶点仅被访问一次。

- 遍历实质

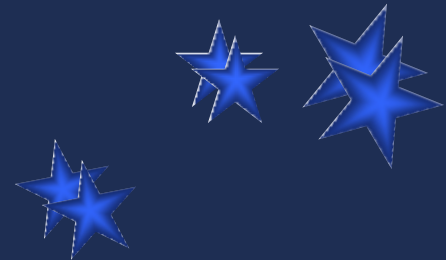
遍历所有连通分量，

对于连通子图：根据邻接关系遍历所有顶点。

设置数组**visited**[0,...n]区分未访问的子图信息

- 搜索路径

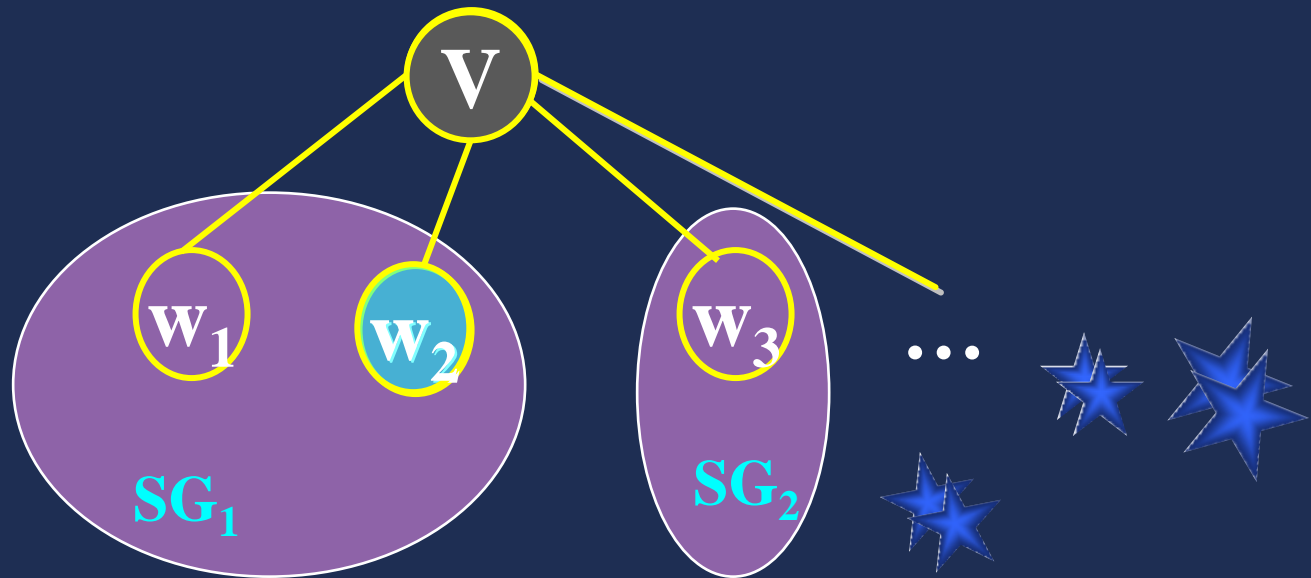
- ◆ 深度优先遍历（DFS）
- ◆ 广度优先遍历（BFS）



## 7.3 图的遍历

- 图的深度遍历 (DFS)

深度优先遍历连通图的过程类似于树的先根遍历，从图中某个顶点 $V$ 出发，访问此顶点，然后依次从 $V$ 的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和 $V$ 有路径相通的顶点都被访问到

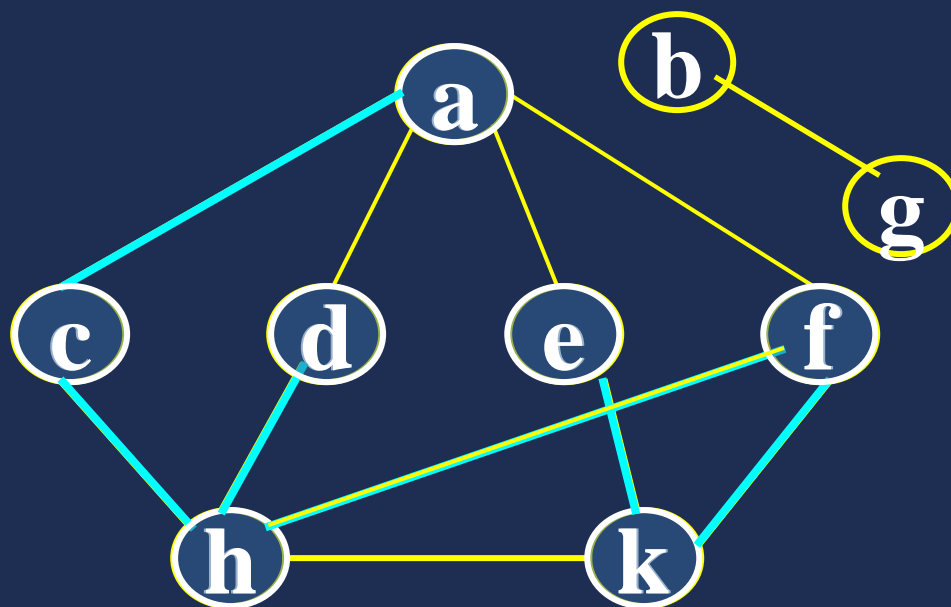




## 7.3 图的遍历

- 图的深度遍历 (DFS)

例:



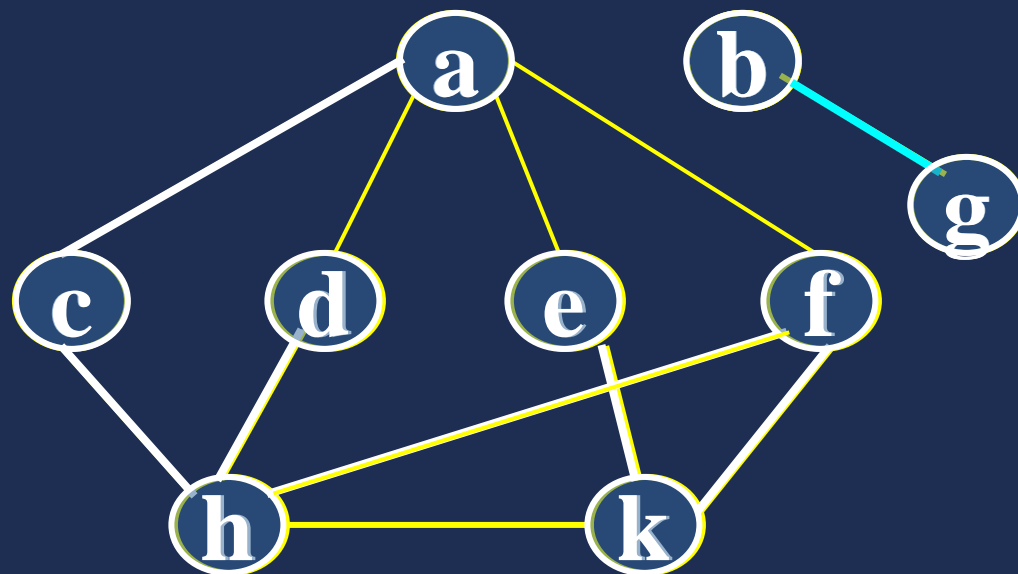
访问次序      a   c   h   d   f   k   e



## 7.3 图的遍历

- 图的深度遍历 (DFS)

例:



访问次序

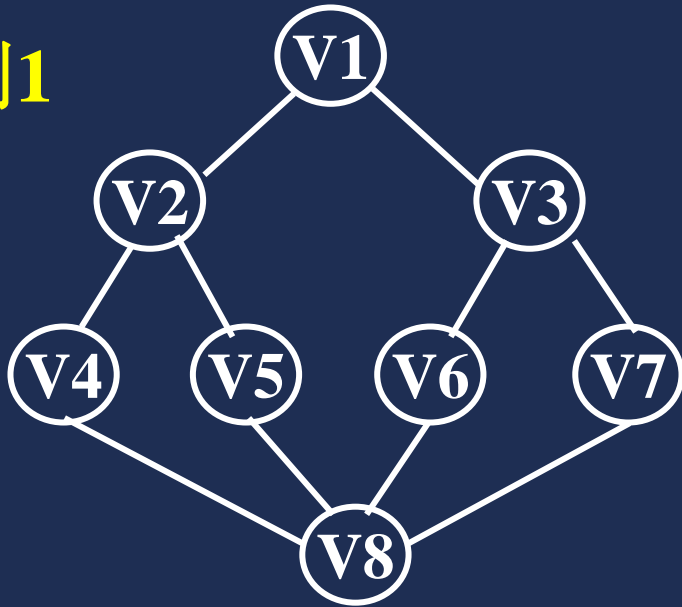
a c h d f k e  
b g



## 7.3 图的遍历

- 图的深度遍历 (DFS)

例1



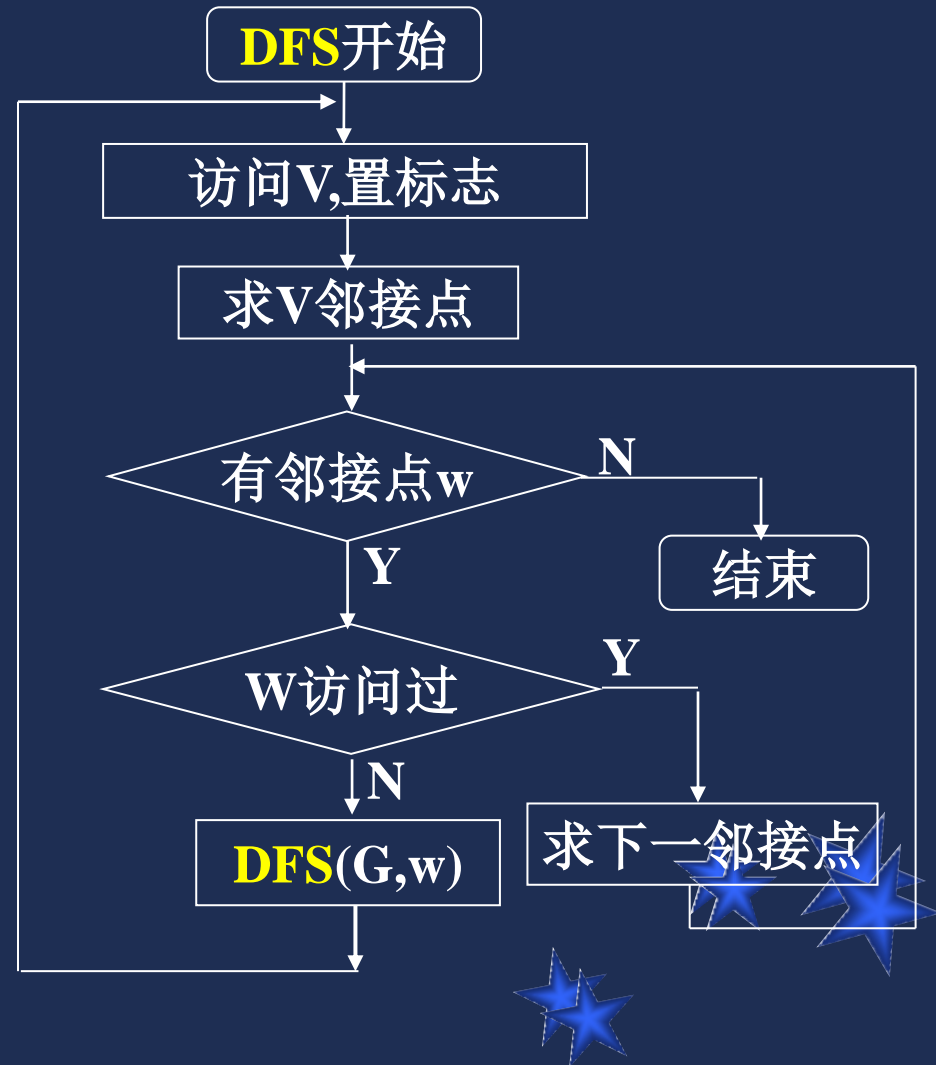
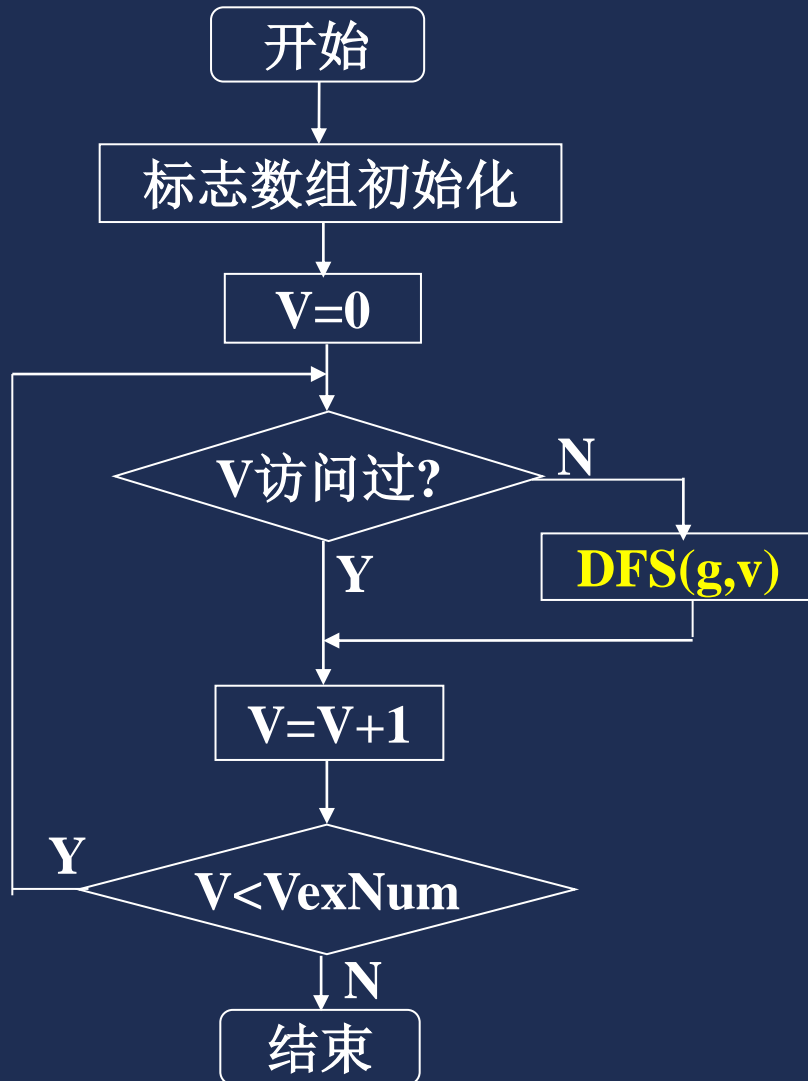
深度遍历1:  $V1 \Rightarrow V2 \Rightarrow V4 \Rightarrow V8 \Rightarrow V5 \Rightarrow V6 \Rightarrow V3 \Rightarrow V7$

深度遍历2:  $V1 \Rightarrow V3 \Rightarrow V7 \Rightarrow V8 \Rightarrow V6 \Rightarrow V5 \Rightarrow V2 \Rightarrow V4$

由于没有规定访问邻接点的顺序，所以深度优先序列不惟一。

## 7.3 图的遍历

- 图的深度遍历 (DFS)



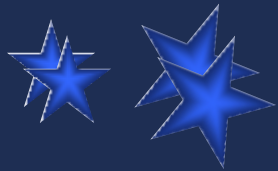
## 7.3 图的遍历

- 图的深度遍历（DFS）——递归算法

```
void DFSTrav ( Graph G,  
              Void ( * Visit ) ( VertexType e ) )  
{  
    for ( v=0; v< G.vexnum; ++v )  
        visited[v] = FALSE;  
    for ( v=0; v<G.vexnum; ++v )  
        if ( ! visited[ v ] )  
            DFS( G, v, Visit );  
} //DFSTrav
```

访问标志数组: int visited[ ]

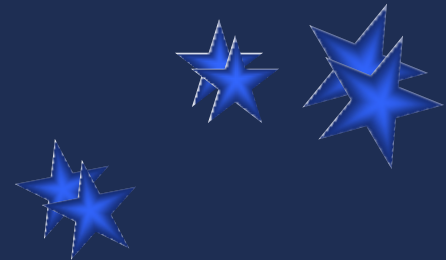
全局变量，初始时所有分量全为FALSE



## 7.3 图的遍历

- 图的深度遍历（**DFS**）——递归算法

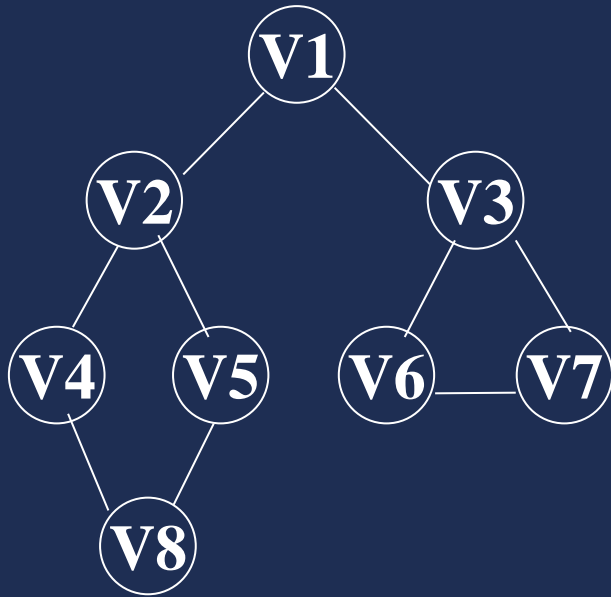
```
void DFS( Graph G, int v,  
          void ( * Visit ) ( VertexType e ) )  
{ /* 从v出发（v是顶点位置），深度优先遍历v所在  
   的连通分量 */  
  visited[v] = TRUE;  
  Visit( v );    //先根遍历  
  for ( w = FirstAdjVex( G, v ); w;  
        w = NextAdjVex( G, v, w ) )  
    if ( ! visited[ w ] )  
      DFS( G, w, Visit( w ) );  
} //DFS
```



## 7.3 图的遍历

### ● 图的深度遍历（DFS）——递归算法

例



	vexdata	firstarc		adjvex	next
1	1	→	3	→	2 ^
2	2	→	5	→	4 → 1 ^
3	3	→	7	→	6 → 1 ^
4	4	→	8	→	2 ^
5	5	→	8	→	2 ^
6	6	→	7	→	3 ^
7	7	→	6	→	3 ^
8	8	→	5	→	4 ^

深度遍历:  $V1 \Rightarrow V3 \Rightarrow V7 \Rightarrow V6 \Rightarrow V2 \Rightarrow V5 \Rightarrow V8 \Rightarrow V4$

## 7.3 图的遍历

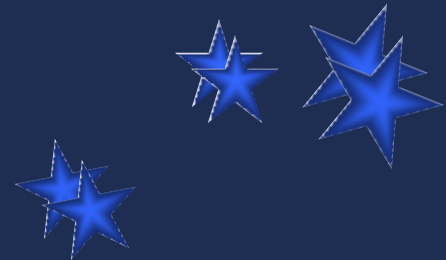
- 深度优先遍历的时间复杂度

访问状态数组初始化，时间复杂度： $O(n)$ 。

- ◆ 邻接表： $O(n + e)$

- ◆ 邻接矩阵： $O(n^2)$

- 查询单个顶点的所有邻接点信息，需要 $O(n)$ 的时间，所以总代价为  $O(n^2)$





## 7.3 图的遍历

- 图的广度遍历 (BFS)

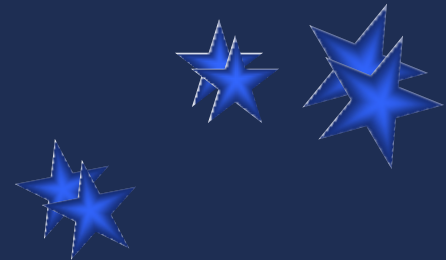
从图中某顶点 $v$ 出发:

1)访问顶点 $v$ ;

2)访问 $v$ 所有未被访问的邻接点 $w_1, w_2, \dots, w_k$

;

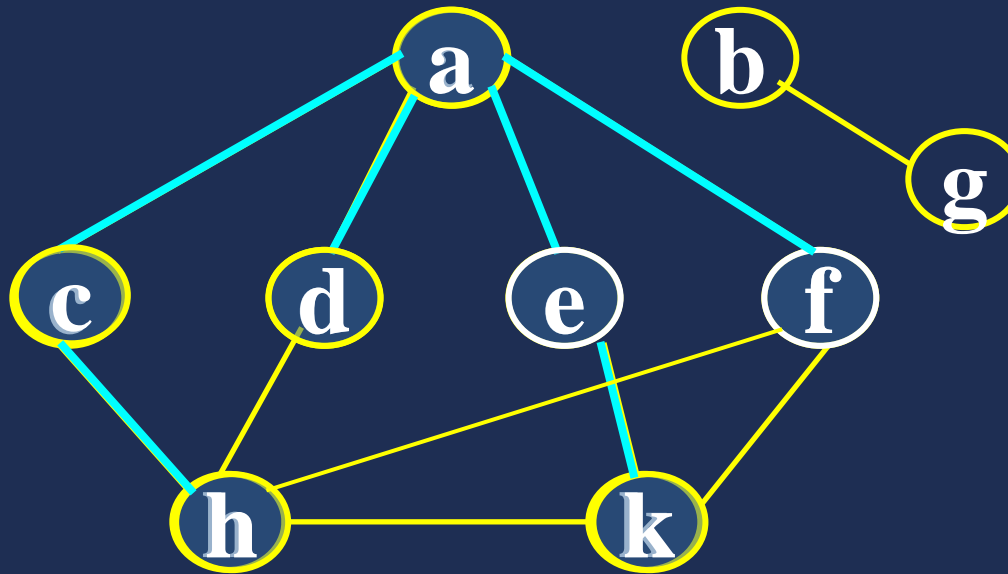
3)依次从这些邻接点出发, 访问其所有未被访问的邻接点。依此类推, 直至图中所有和 $V_0$ 有路径相通的顶点都被访问到。



## 7.3 图的遍历

- 图的广度遍历 (BFS)

例:

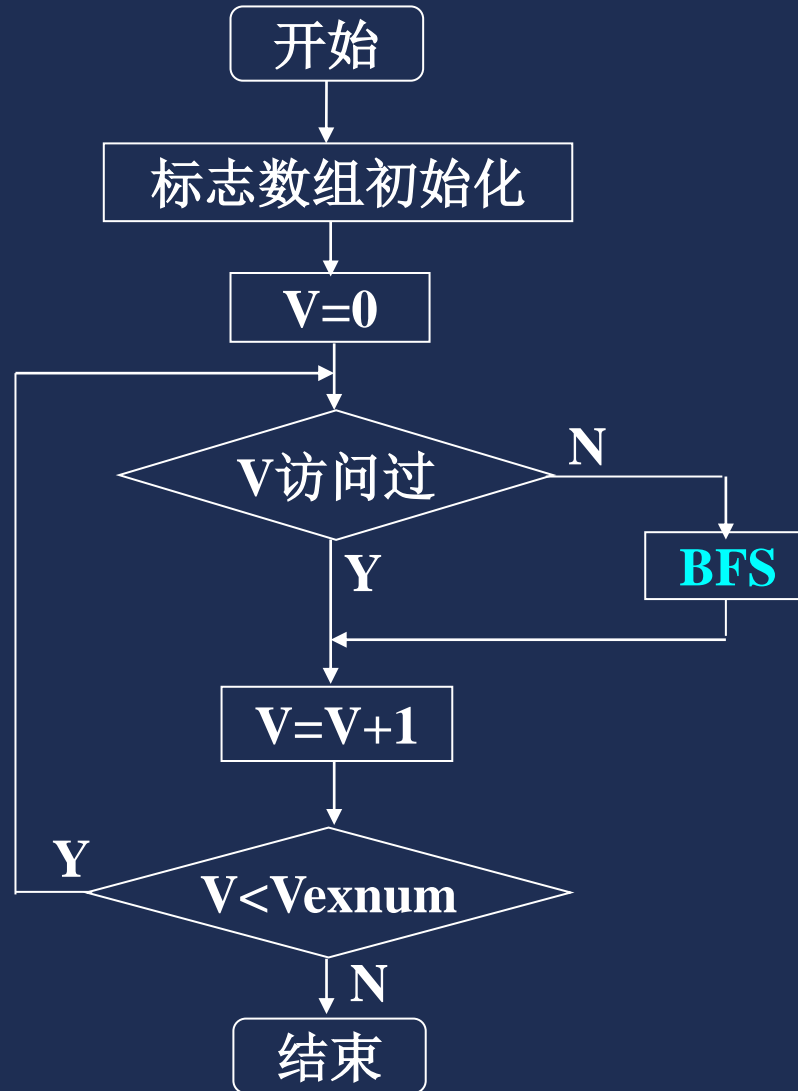


访问次序    a   c   d   e   f   h   k



## 7.3 图的遍历

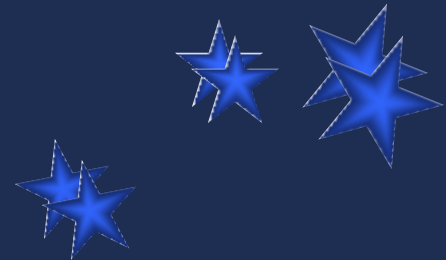
- 图的广度遍历（**BFS**）——递归算法



## 7.3 图的遍历

- 图的广度遍历 (**BFS**) ——递归算法

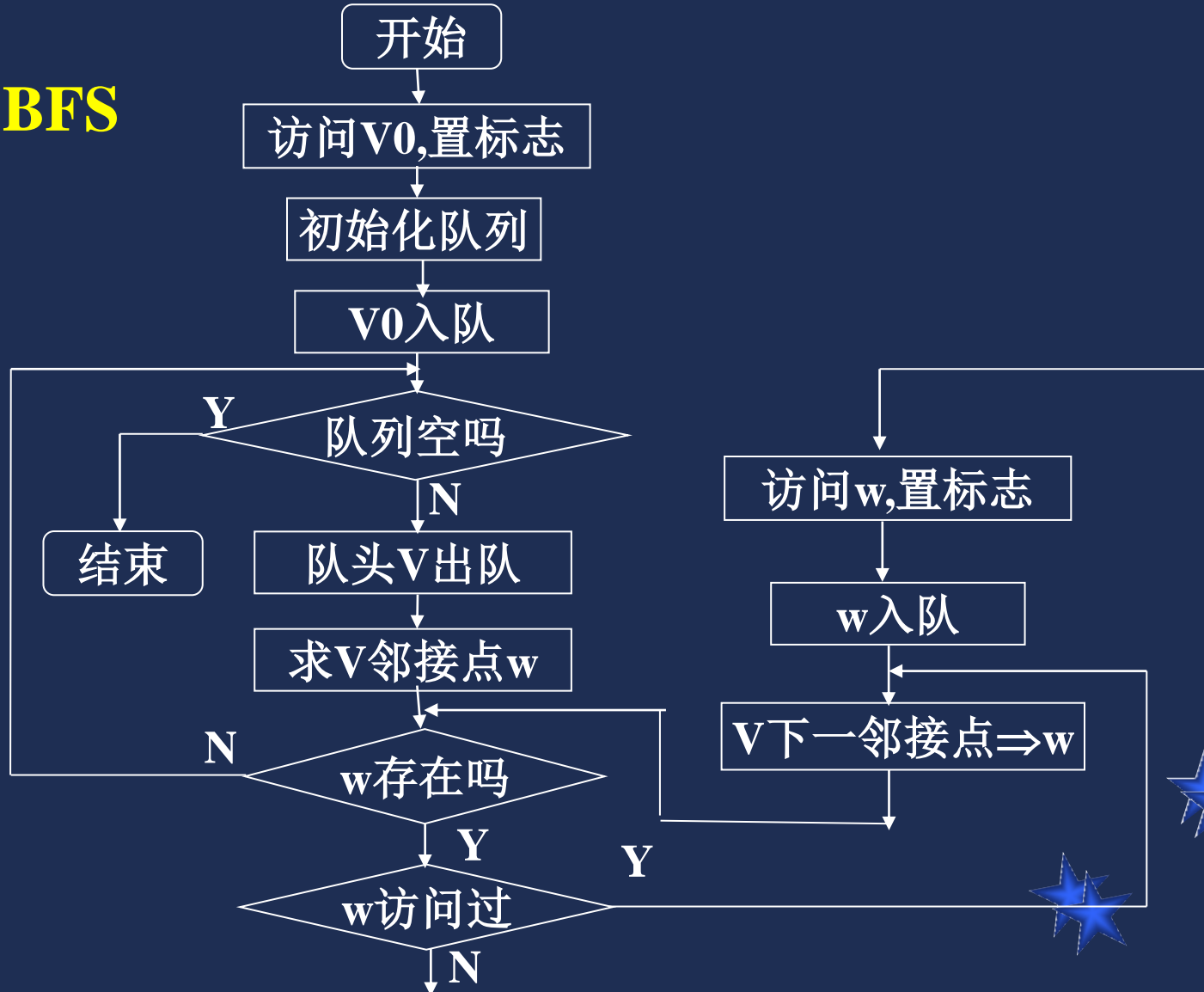
```
void BFSTraverse ( Graph G,  
                  void (* Visit) ( VertexType ) )  
{  
    //本算法对图G进行广度优先遍历  
    for ( v=0; v<G.vexnum; ++v )  
        visited[v] = FALSE; // 访问标志数组初始化  
    for ( v=0; v<G.vexnum; ++v )  
        if ( ! visited[v] )  
            BFS( G, v, Visit );  
} //BFSTraverse
```



## 7.3 图的遍历

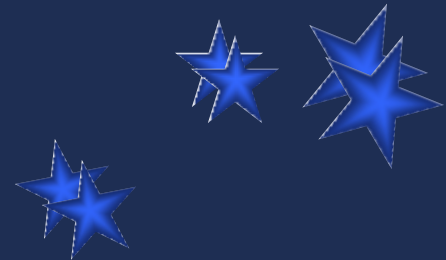
### ● 图的广度遍历（BFS）——算法7.6

**BFS**



## 7.3 图的遍历

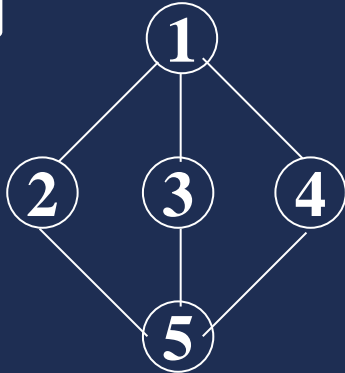
```
void BFS( Graph G, int v, void(* Visit) (VertexType e) )
{ // 从第v个顶点出发
  InitQueue(Q); // 建立辅助空队列Q
  Visit(v); visited[v]=TRUE; // 访问u, 访问标志数组
  EnQueue(Q,v); // v入队
  while ( ! QueueEmpty( Q ) )
  { DeQueue(Q,u); // 队头元素出队, 并赋值给u
    for ( w=FirstAdjVex(G,u); w; w=NextAdjVex(G,u,w) )
      if ( ! visited[w] )
      { Visit(w);
        visited[w]=TRUE; // 访问u
        EnQueue(Q,w);
      }
  } //while
} //BFS
```



## 7.3 图的遍历

### ● 图的广度遍历 (BFS)

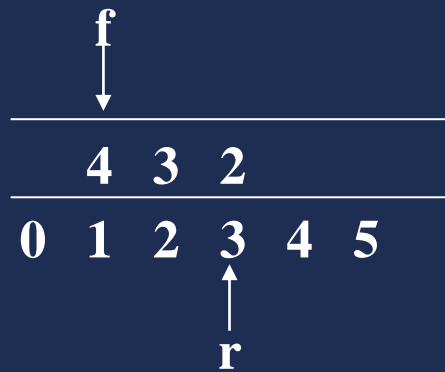
例



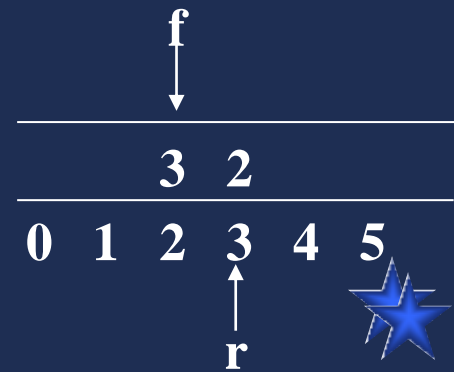
	vexdata	firstarc	adjvex	next
1	1	→	4	→ 3 → 2 ^
2	2	→	5	→ 1 ^
3	3	→	5	→ 1 ^
4	4	→	5	→ 1 ^
5	5	→	4	→ 3 → 2 ^



遍历序列: 1



遍历序列: 1 4 3 2

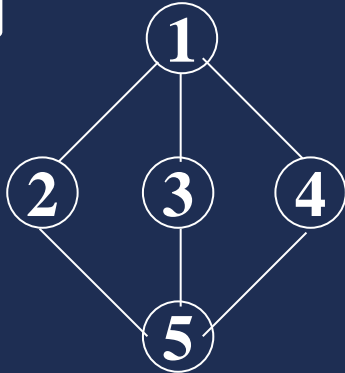


遍历序列: 1 4 3 2

# 7.3 图的遍历

## ● 图的广度遍历 (BFS)

例



	vexdata	firstarc	adjvex	next
1	1	→	4	→ 3 → 2 ^
2	2	→	5	→ 1 ^
3	3	→	5	→ 1 ^
4	4	→	5	→ 1 ^
5	5	→	4	→ 3 → 2 ^

f	f	f	f
↓	↓	↓	↓
3 2 5	2 5	5	
0 1 2 3 4 5	0 1 2 3 4 5	0 1 2 3 4 5	0 1 2 3 4 5
↑ r	↑ r	↑ r	↑ r

遍历序列: 1 4 3 2 5    遍历序列: 1 4 3 2 5    遍历序列: 1 4 3 2 5    遍历序列: 1 4 3 2 5

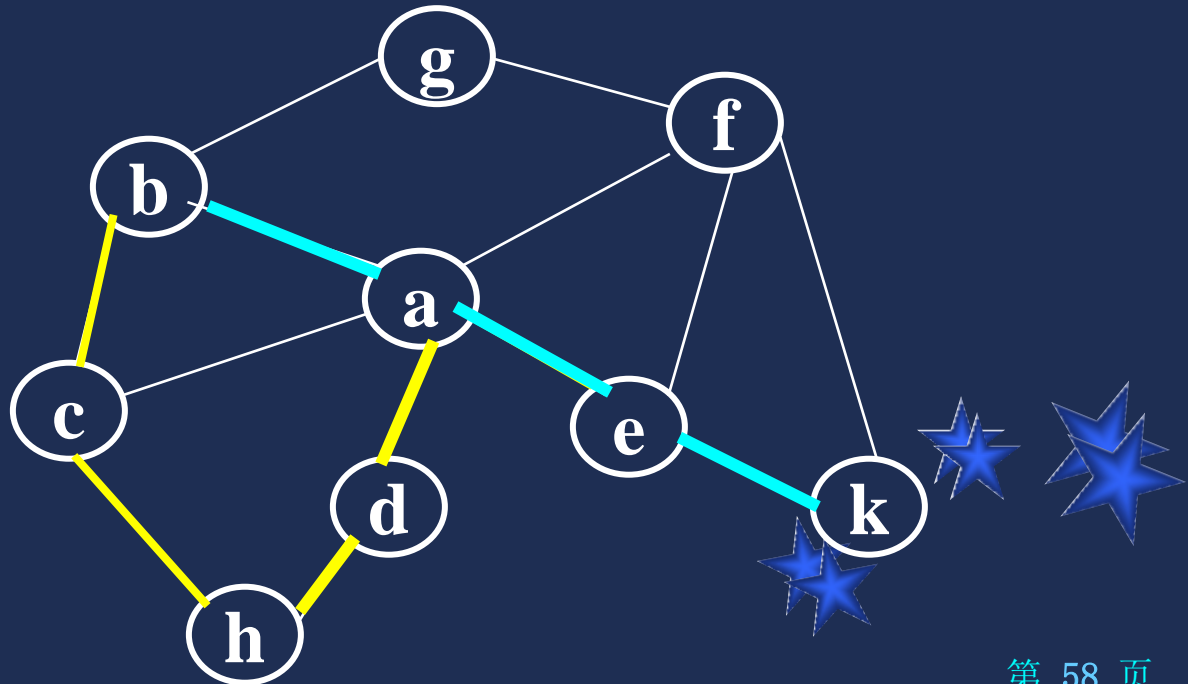


## 7.3 图的遍历

- 遍历的应用

求两个顶点之间的最短路径长度

广度优先搜索访问顶点的次序是按“路径长度”渐增的次序。求路径长度最短的路径可以基于广度优先搜索遍历进行。



## 7.4 图的最小生成树

### ● 生成树

包含无向图  $G$  所有顶点的极小连通子图称为 $G$ 生成树，它只有 $n-1$ 条边，可以构成一棵树。

**极小连通子图**含义：该子图是 $G$ 的连通子图，在该子图中删除任何一条边，子图不再连通。

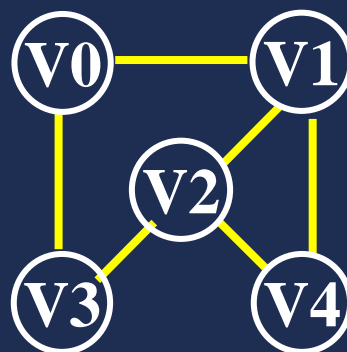
生成树 $T$ 的特点：

$T$ 是 $G$ 的连通子图

$T$ 包含 $G$ 的所有顶点

$T$ 中有 $n-1$ 条边

$T$ 中无回路



连通图 $G_1$



$G_1$ 的生成树

## 7.4 图的最小生成树

- 问题提出

假设要在  $n$  个城市之间建立通讯联络网，则连通  $n$  个城市只需要修建  $n-1$  条线路，如何在最节省经费的前提下建立这个通讯网？

- 问题分析和数学建模：

顶点——表示城市

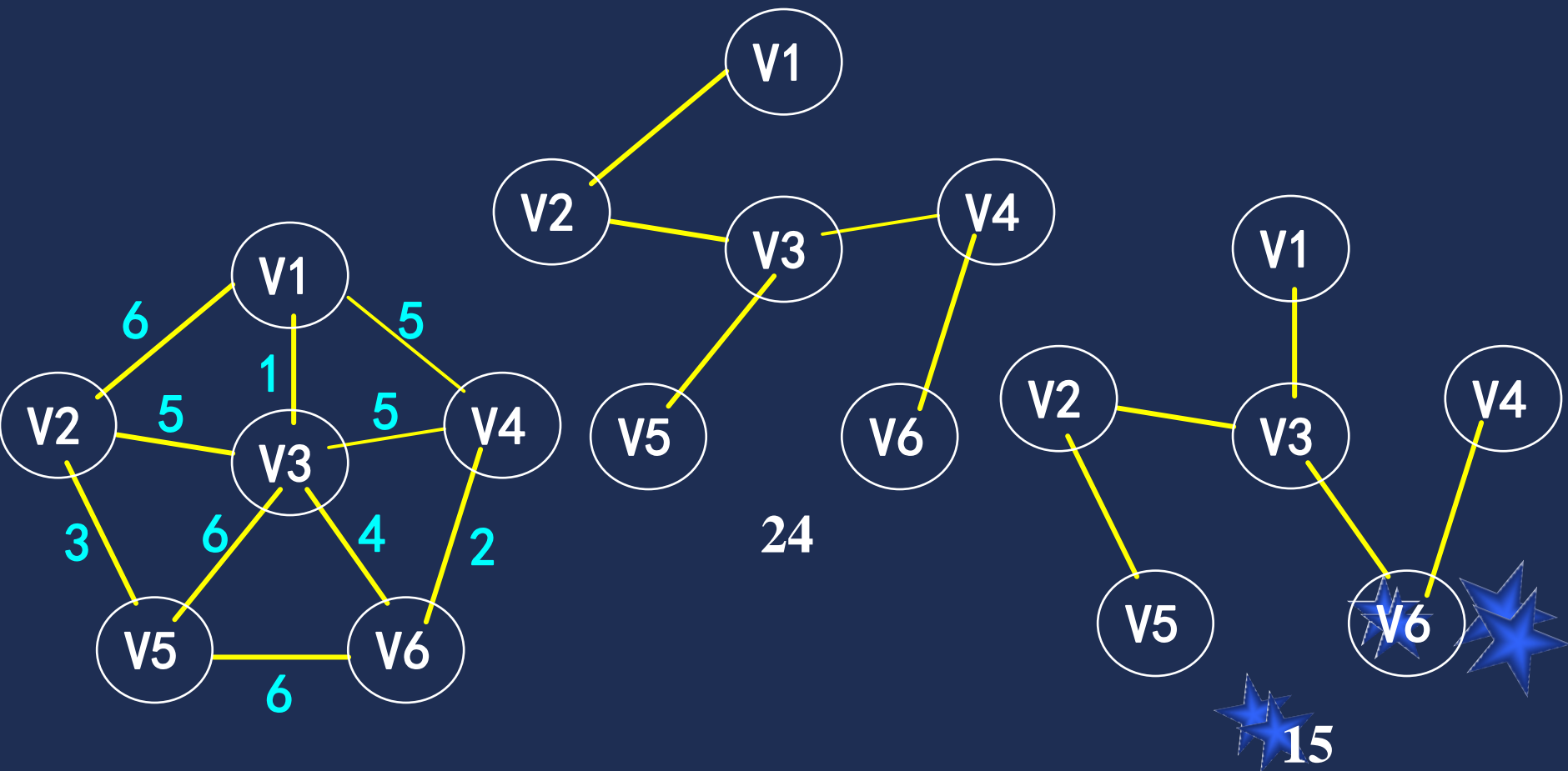
权——城市间建立通信线路所需花费代价

希望找到一棵生成树，它的每条边上的权值之和（即建立该通信网所需花费的总代价）

最小——最小代价生成树 **MST(Minimum cost Spanning Tree)**

## 7.4 图的最小生成树

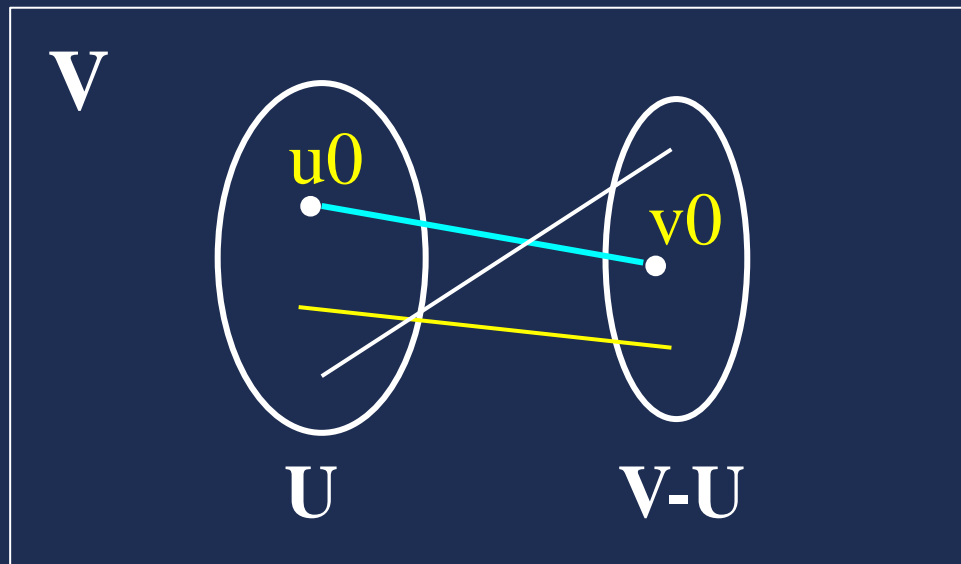
- **最小生成树(Least weighted spanning tree):**  
权（之和）最小的生成树。



## 7.4 图的最小生成树

- 利用 MST 性质构造最小生成树

若  $U$  集是  $V$  的一个非空子集，若  $(u_0, v_0)$  是一条最小权值的边，其中  $u_0 \in U$ ， $v_0 \in V-U$ ；  
则：  $(u_0, v_0)$  必在某一棵最小生成树上。



## 7.4 图的最小生成树

- MST性质证明：用反证法

假设连通网  $N$  的任何一棵最小生成树都不包含边  $(u_0, v_0)$ 。设  $T$  是连通网上的一棵最小生成树，当把边  $(u_0, v_0)$  加入到  $T$  中时，由生成树的定义可知， $T$  中必存在一条包含  $(u_0, v_0)$  的回路。

另一方面，由于  $T$  是生成树，则在  $T$  上必存在另一条边  $(u', v')$ ，其中  $u' \in U$ ， $v' \in V-U$ ，且  $u_0$  和  $u'$  之间， $v_0$  和  $v'$  之间均有路径相通。删去边  $(u', v')$ ，便可消除上述回路，同时得到另一棵包含边  $(u_0, v_0)$  生成树  $T'$ 。

因为  $(u_0, v_0)$  的代价不大于  $(u', v')$  的代价，所以  $T'$  的代价也不大于  $T$  的代价。与假设矛盾，因此命题成立。

## 7.4 图的最小生成树

- 最小生成树的**MST** 性质

若 $U$ 集是 $V$ 的一个非空子集，若在所有**联接 $U$ 和 $V-U$** 的边中， **$(u, v)$ 权值最小**，其中 $u \in U$ ， $v \in V-U$ ；则：必有一棵**最小生成树包含 $(u, v)$** 。

- 典型算法

- ◆ 普里姆(Prim)算法

**将顶点归并**，与边数无关，适于**稠密网**。

- ◆ 克鲁斯卡尔(Kruskal)算法

**将边归并**，适于求**稀疏网**的最小生成树。

## 7.4 图的最小生成树

- 普里姆算法 (Prim)

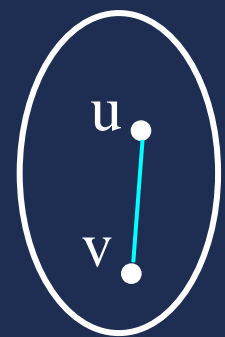
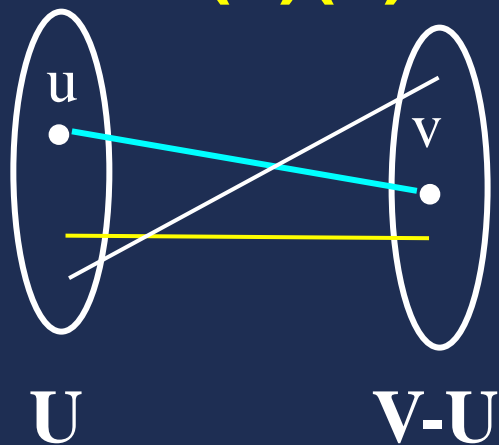
设  $G = (V, GE)$  为一个具有  $n$  个顶点的连通网络,  $T = (U, TE)$  为构造的生成树。

(1) 初始时,  $U = \{u_0\}$ ,  $TE = \phi$ ;

(2) 在所有  $u \in U$  且  $v \in V - U$  的边  $(u, v)$  中选择一条权值最小的边, 不妨设为  $(u, v)$ ;

(3)  $(u, v)$  加入  $TE$ , 同时将  $v$  加入  $U$ ;

(4) 重复(2)(3), 直到  $U = V$  为止;

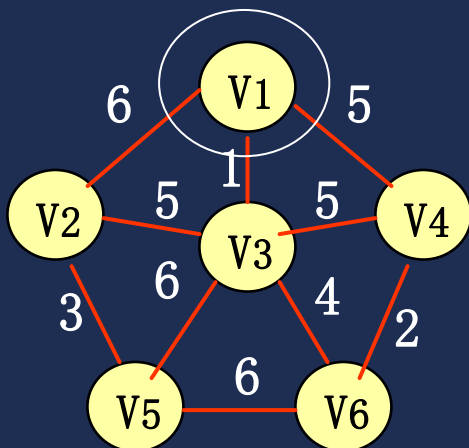




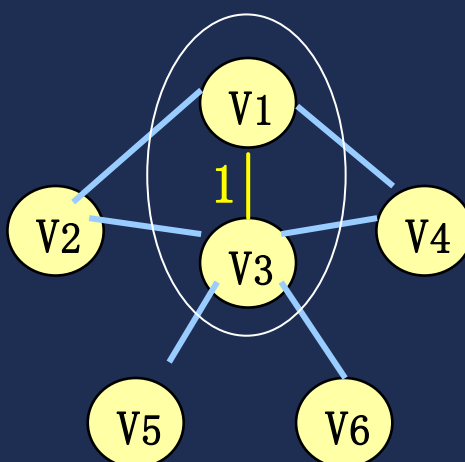
# 7.4 图的最小生成树

## • 普鲁姆算法 (Prim)

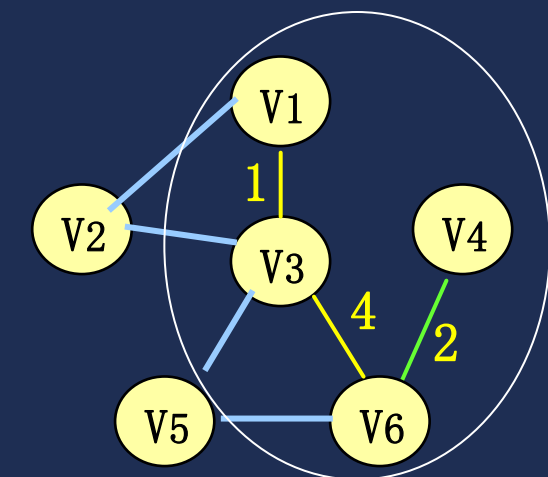
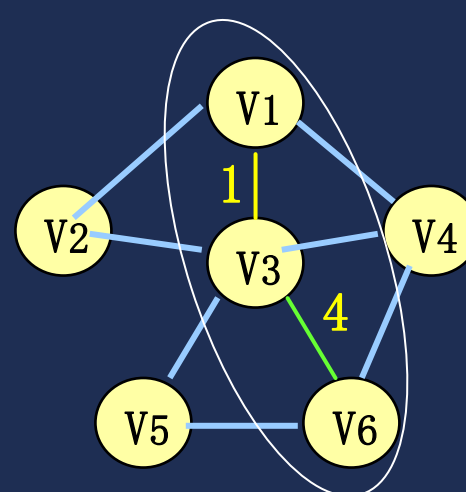
$U = \{ V1 \}$



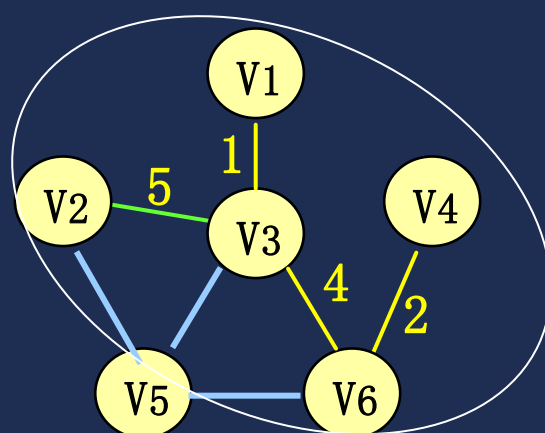
$U = \{ V1, V3 \}$



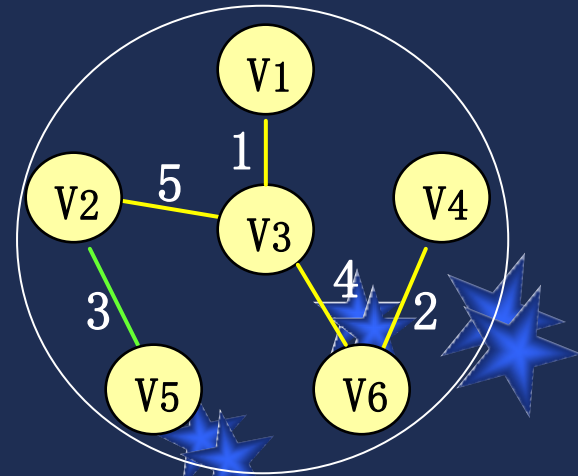
$U = \{ V1, V3, V6 \}$



$U = \{ V1, V3, V6, V4 \}$



$U = \{ V1, V3, V6, V4, V2 \}$



$U = \{ V1, V3, V6, V4, V2, V5 \}$

## 7.4 图的最小生成树

- 辅助数组 `closedge[ ]` 对不在生成树中的每个顶点，记录其和生成树顶点相关联且代价最小的边：

```
struct { VertexType Adjvex; // 相关顶点
        VRType      lowcost; // 最小边的权值
} closedge[ MAX_VERTEX_NUM ];
```

`closedge.Adjvex[ v ]:`

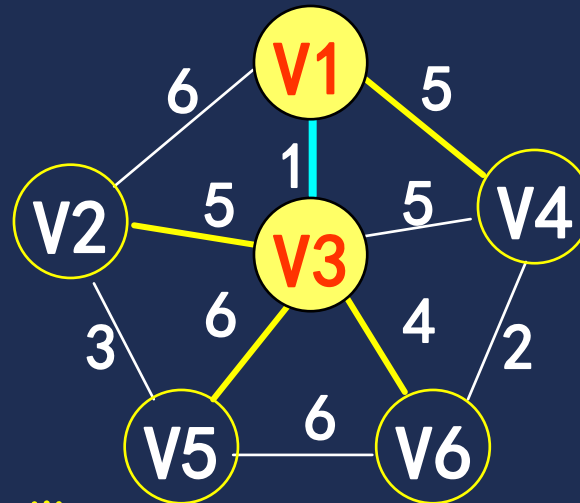
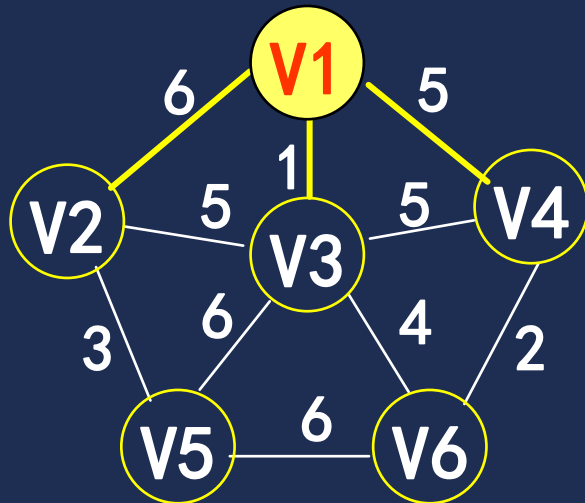
顶点 `v` 到子集 `U` 中权最小边  $(v, u)$  相关联的顶点 `u`

`closedge.lowcost[v]:`

顶点 `v` 到子集 `U` 权最小边  $(v, u)$  的权值(距离)



# 7.4 图的最小生成树



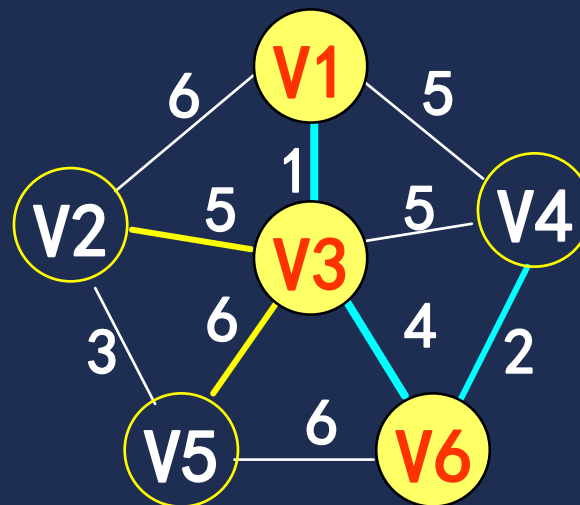
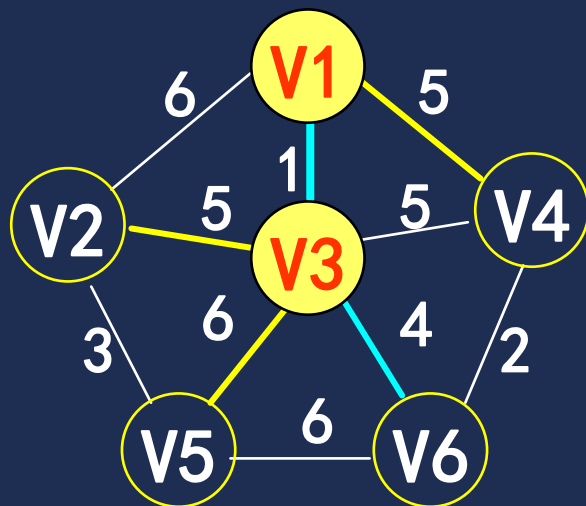
0(V1)   1(V2)   2(V3)   3(V4)   4(V5)   5(V6)

closedge.Adjvex	V1	V1	V1	V1		
closedge.Lowcost	0	6	1	5	max	max

0(V1)   1(V2)   2(V3)   3(V4)   4(V5)   5(V6)

closedge.Adjvex	V1	V3	V1	V1	V3	V3
closedge.Lowcost	0	5	0	5	6	4

# 7.4 图的最小生成树



0(V1) 1(V2) 2(V3) 3(V4) 4(V5) 5(V6)

closedge.Adjvex

closedge.Lowcost

	V3	V1	V1	V3	V3
0	5	0	5	6	4

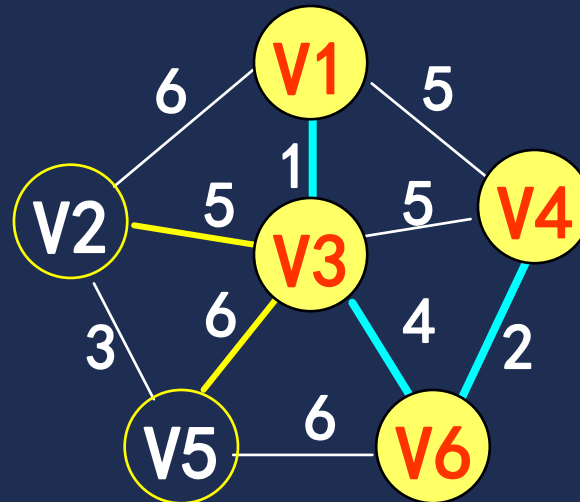
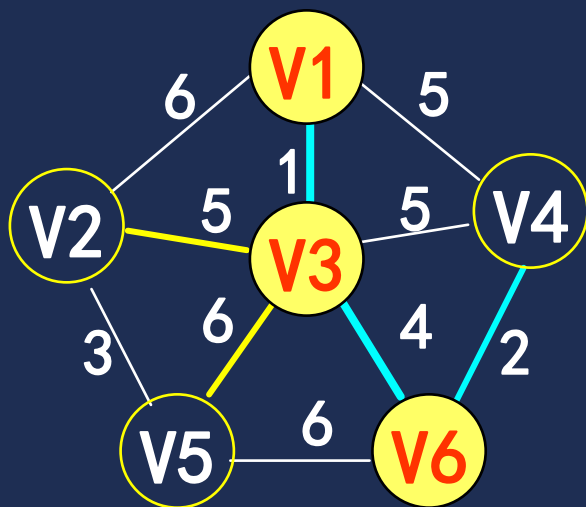
0(V1) 1(V2) 2(V3) 3(V4) 4(V5) 5(V6)

closedge.Adjvex

closedge.Lowcost

	V3	V1	V6	V3	V3
0	5	0	2	6	0

# 7.4 图的最小生成树



0(V1) 1(V2) 2(V3) 3(V4) 4(V5) 5(V6)

closedge.Adjvex  
closedge.Lowcost

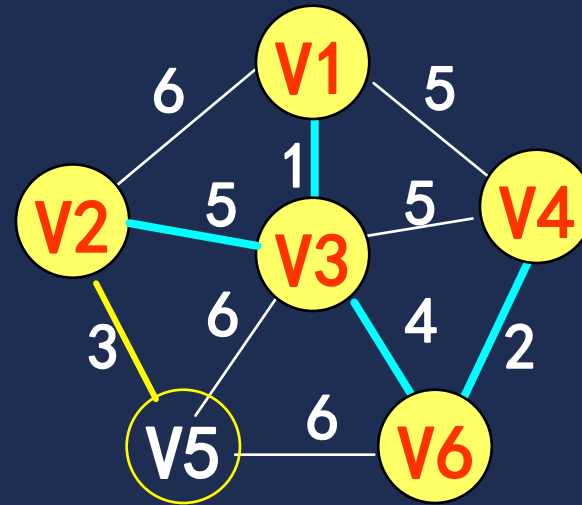
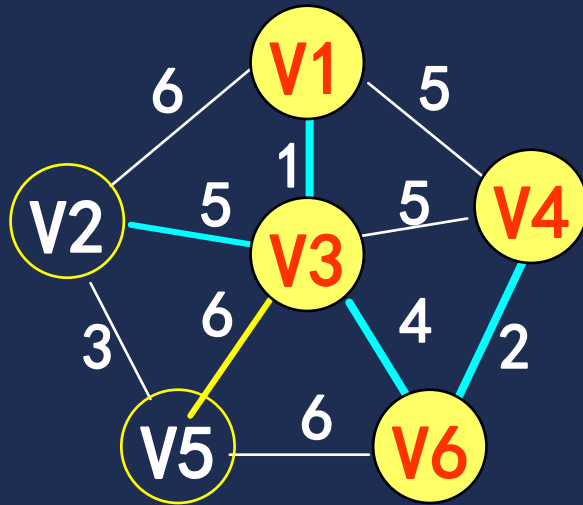
	V3	V1	V6	V3	V3
0	5	0	0	6	0

0(V1) 1(V2) 2(V3) 3(V4) 4(V5) 5(V6)

closedge.Adjvex  
closedge.Lowcost

	V3	V1	V6	V3	V3
0	5	0	0	6	0

# 7.4 图的最小生成树



0(V1) 1(V2) 2(V3) 3(V4) 4(V5) 5(V6)

closedge.Adjvex  
closedge.Lowcost

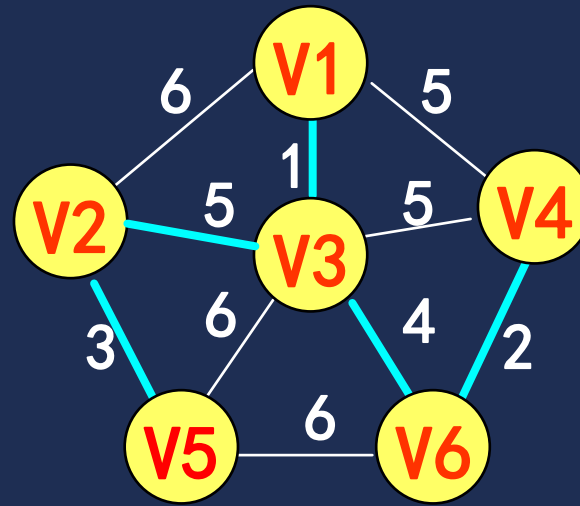
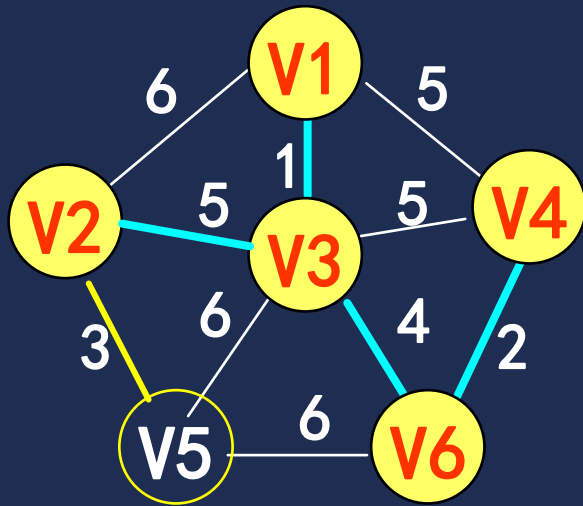
	V3	V1	V6	V3	V3
0	5	0	0	6	0

0(V1) 1(V2) 2(V3) 3(V4) 4(V5) 5(V6)

closedge.Adjvex  
closedge.Lowcost

	V3	V1	V6	V3	V3
0	0	0	0	3	0

# 7.4 图的最小生成树



0(V1)   1(V2)   2(V3)   3(V4)   4(V5)   5(V6)

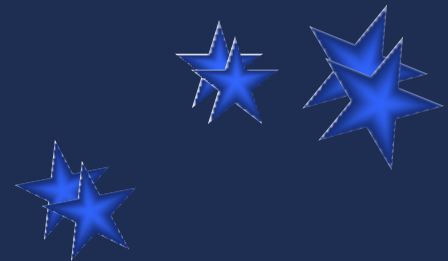
	V3	V1	V6	V3	V3
0	0	0	0	0	0

closedge.Adjvex  
closedge.Lowcost



## 7.4 图的最小生成树

```
void MiniSpanTree_P( MGraph G, VertexType u )
{
    //用普里姆算法从顶点u出发构造网G的最小生成树
    k = LocateVex ( G, u );
    for ( j=0; j<G.vexnum; ++j ) // 辅助数组初始化
        if (j!=k)
            closedge[j] = { u, G.arcs[k][j] };
    closedge[k].Lowcost = 0;    // 初始, U={u}
    for ( i=1; i<G.vexnum; ++i )
    {
        继续向生成树上添加顶点;
    }
```





## 7.4 图的最小生成树

// 依次向生成树上添加顶点

k = minimum( closedge );

// 求出加入生成树的下一个顶点(k)

printf( closedge[k].Adjvex, G.vexs[k] );

// 输出生成树上一条边

closedge[k].Lowcost = 0; // 第k顶点并入U集

for ( j=0; j<G.vexnum; ++j )

// 修改其它顶点的最小边

if ( G.arcs[k][j] < closedge[j].Lowcost )

closedge[j] = { G.vexs[k], G.arcs[k][j] };

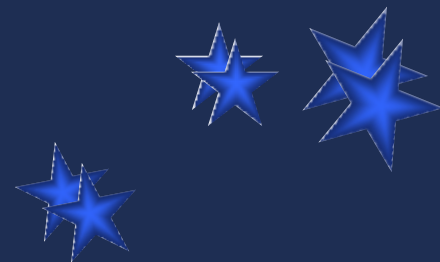
复杂度:  $O(n^2)$

与边数e无关

适用于稠密图

# 普里姆算法不同实现下的复杂度

数据结构、找最小	时间复杂度（总计）
邻接矩阵、扫描	$O(V^2)$
邻接表、二叉堆	$O((V + E) \log(V)) = O(E \log(V))$
邻接表、斐波那契堆	$O(E + V \log(V))$



## 7.4 图的最小生成树

- 克鲁斯卡尔(Kruskal)算法

设连通网  $N = (V, \{E\})$ 。

① 初始时最小生成树只包含图的  $n$  个顶点，每个顶点为一棵子树（构成一个连通分量）；

② 选取权值较小且所关联的两个顶点不在同一连通分量的边，将此边加入最小生成树中；

③ 重复②  $n-1$  次，即得到包含  $n$  个顶点和  $n-1$  条边的最小生成树。

## 7.4 图的最小生成树

- 克鲁斯卡尔(Kruskal)算法

构造非连通图  $ST = (V, \{\})$ ; //  $n$ 个独立分量

$k = i = 0$ ; //  $k$ : 记录选中的边数

while (  $k < n-1$  ) {

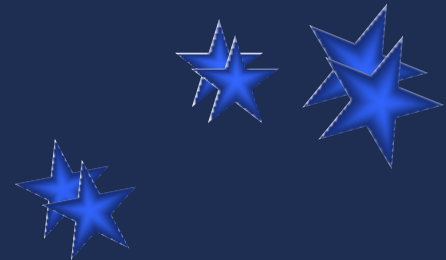
++  $i$ ;

检查边集  $E$  中第  $i$  条权值最小的边  $(u, v)$ ;

若  $(u, v)$  加入  $ST$  后不使  $ST$  中产生回路,

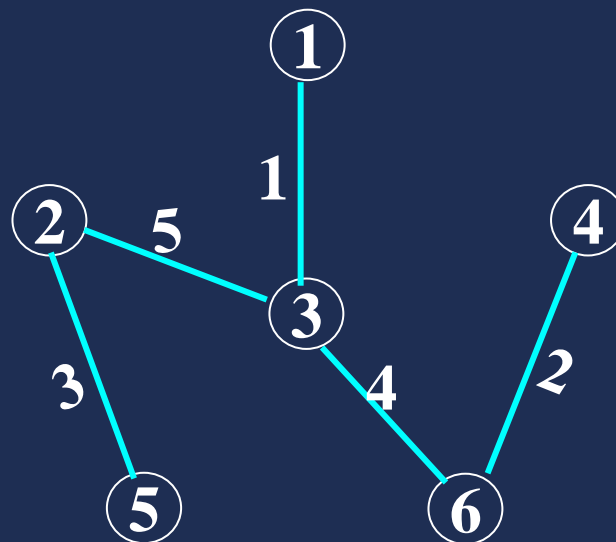
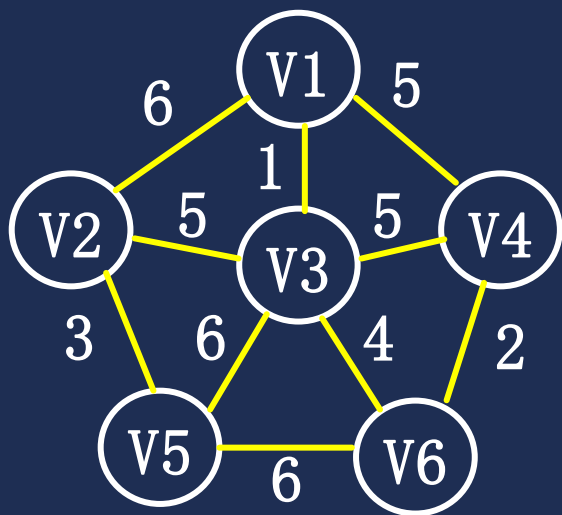
则 输出边  $(u, v)$ , 且  $k++$ ;

}



## 7.4 图的最小生成树

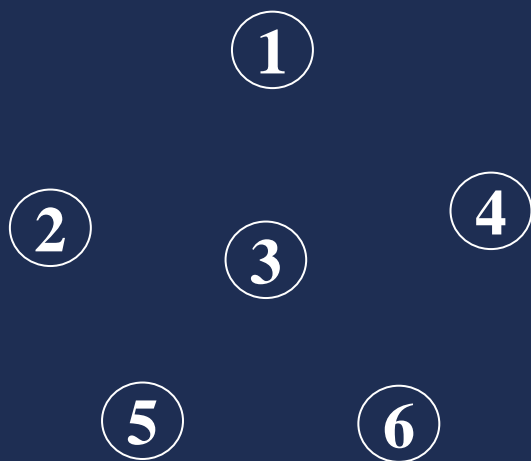
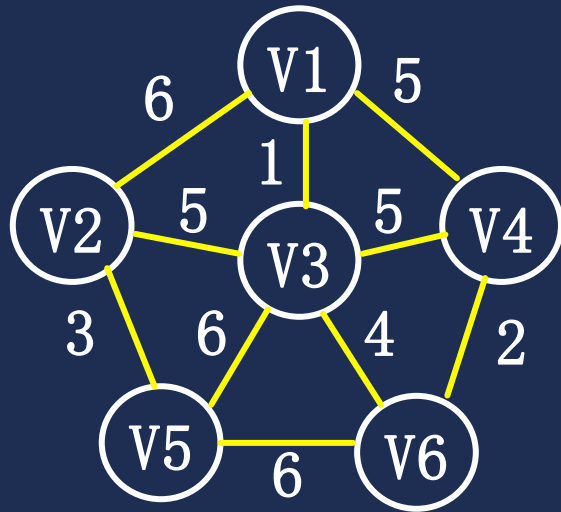
- 克鲁斯卡尔(Kruskal)算法



## 7.4 图的最小生成树

### • 克鲁斯卡尔(Kruskal)算法

采用边集数组存储图:



data set

1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6

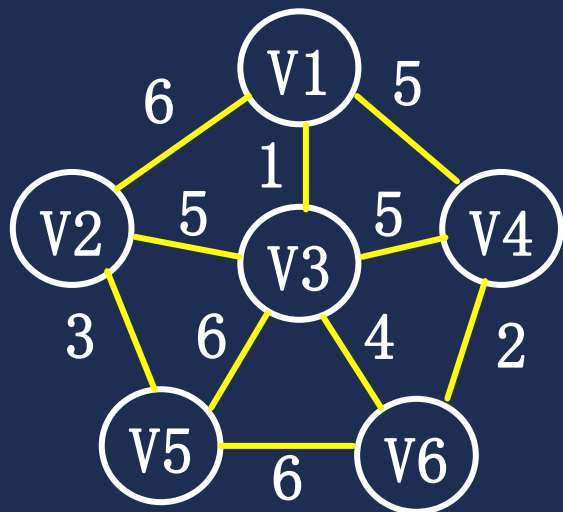
vexh vext weight flag

0	1	2	6	0
1	1	3	1	0
2	1	4	5	0
3	2	3	5	0
4	2	5	3	0
5	3	4	5	0
6	3	5	6	0
7	3	6	4	0
8	4	6	2	0
9	5	6	6	0

# 7.4 图的最小生成树

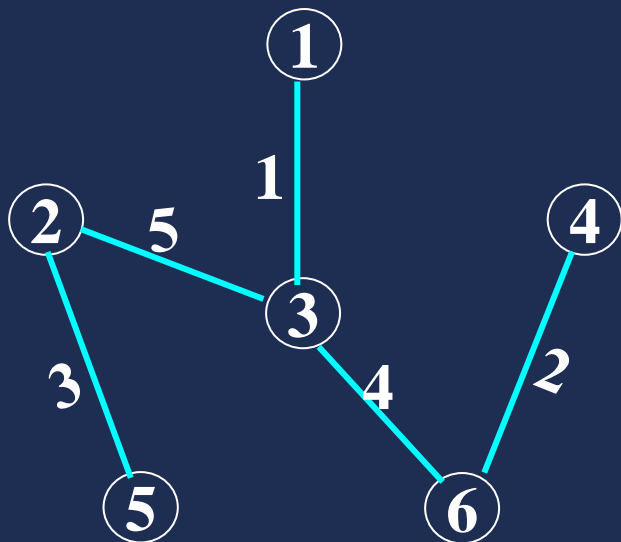
## ● 克鲁斯卡尔(Kruskal)算法

采用边集数组的形式保存图：



	data	set
1	1	2
2	2	2
3	3	2
4	4	2
5	5	2
6	6	2

	vexh	vext	weight	flag
0	1	2	6	0
1	1	3	1	1
2	1	4	5	0
3	2	3	5	1
4	2	5	3	1
5	3	4	5	0
6	3	5	6	0
7	3	6	4	1
8	4	6	2	1
9	5	6	6	0

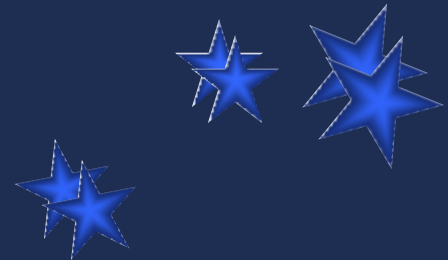


## 7.4 图的最小生成树

- 克鲁斯卡尔的性能

设图的边数是 $e$ ，克鲁斯卡尔算法的时间复杂度为 $O(e \log e)$ 。

适用于求边稀疏的网的最小生成树。

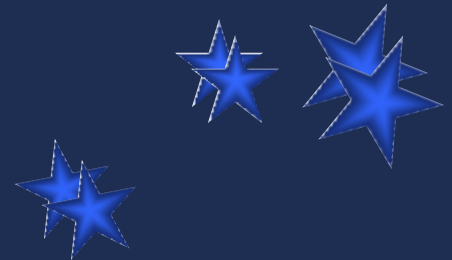




## 7.4 图的最小生成树

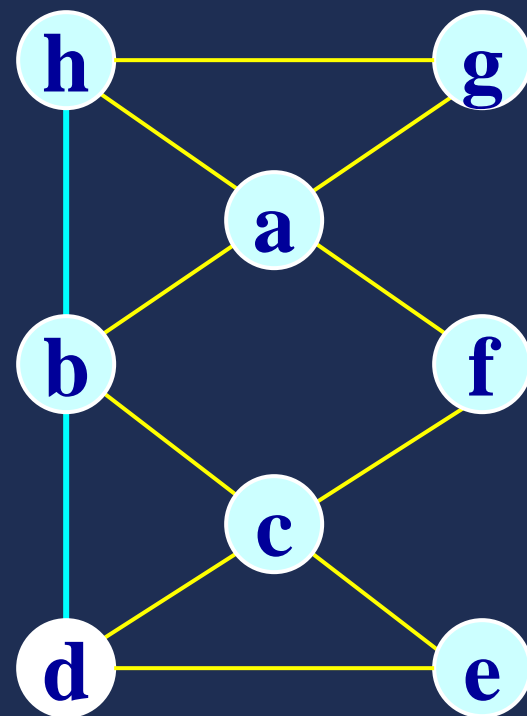
- 两种算法比较

算法名	普里姆算法	克鲁斯卡尔算法
时间复杂度	$O(n^2)$	$O(e \log e)$
适应范围	稠密图	稀疏图



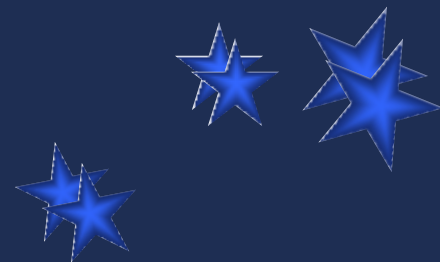
## 7.5 重连通图和关节点

- 定义：若从一个连通图中删去一个顶点及其相关联的边，连通图成为两个或多个连通分量，则该点称为**关节点**。
- 定义：若从一个连通图中删去任意一个顶点及其相关联的边，它仍为一个连通图的话，则该连通图被称为**重（双）连通图**。



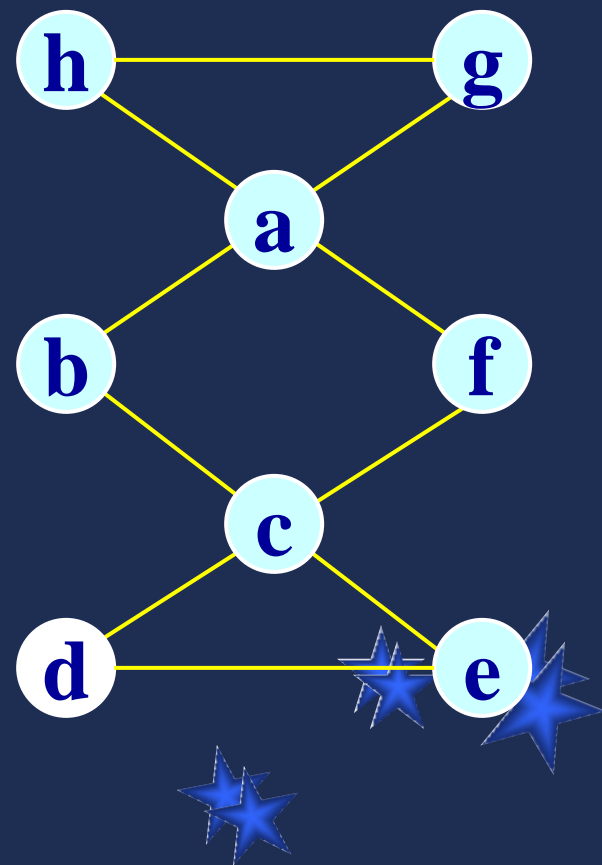
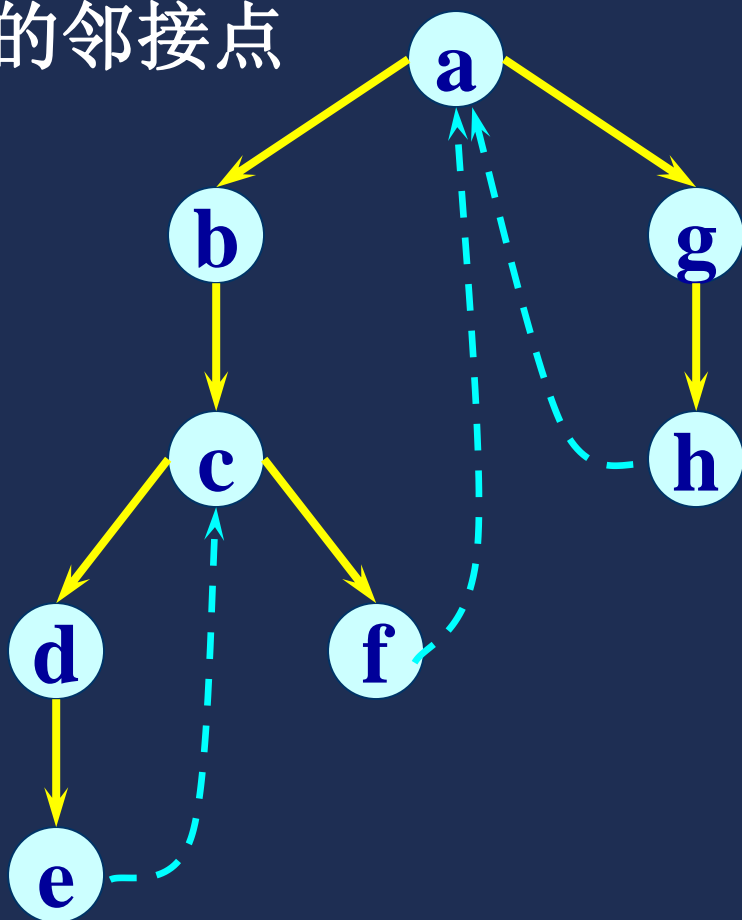
双连通图中没有关节点

没有关节点的连通图为双连通图

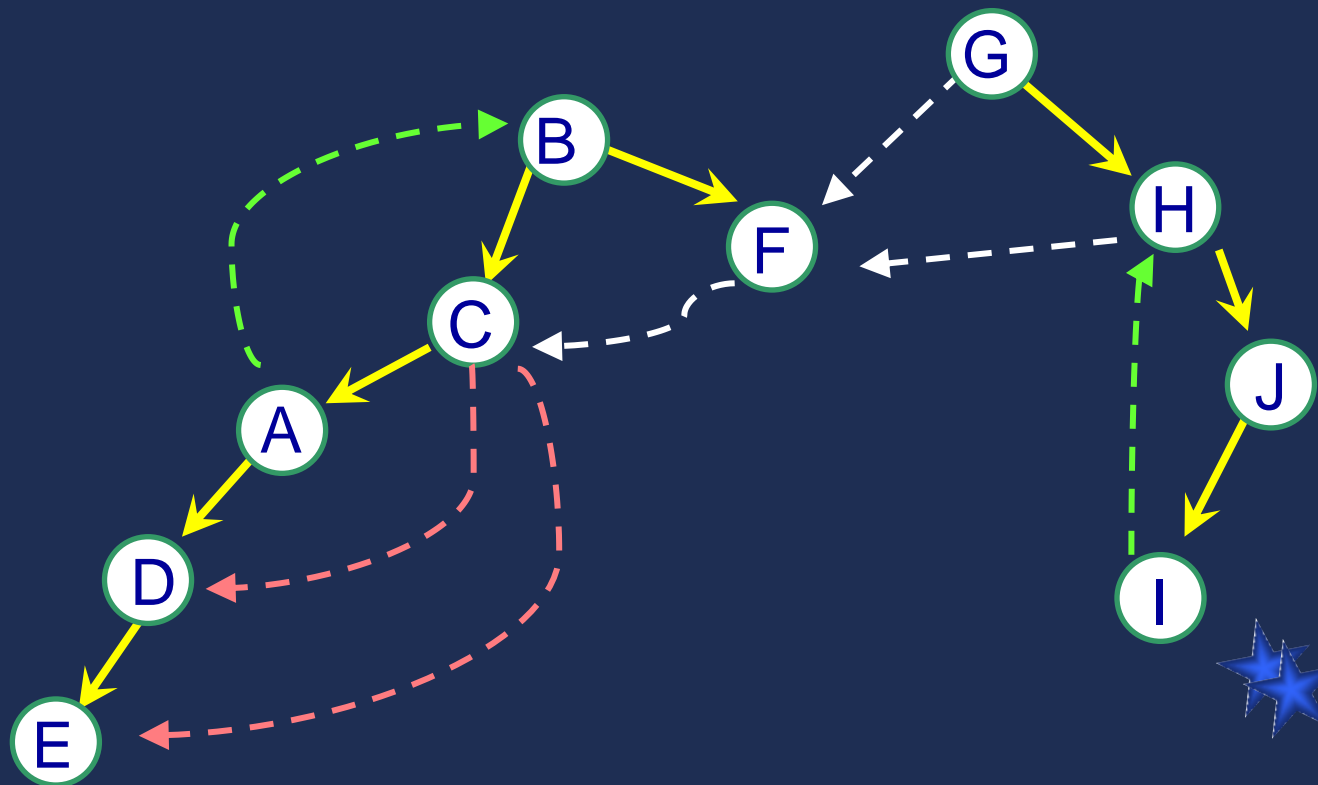
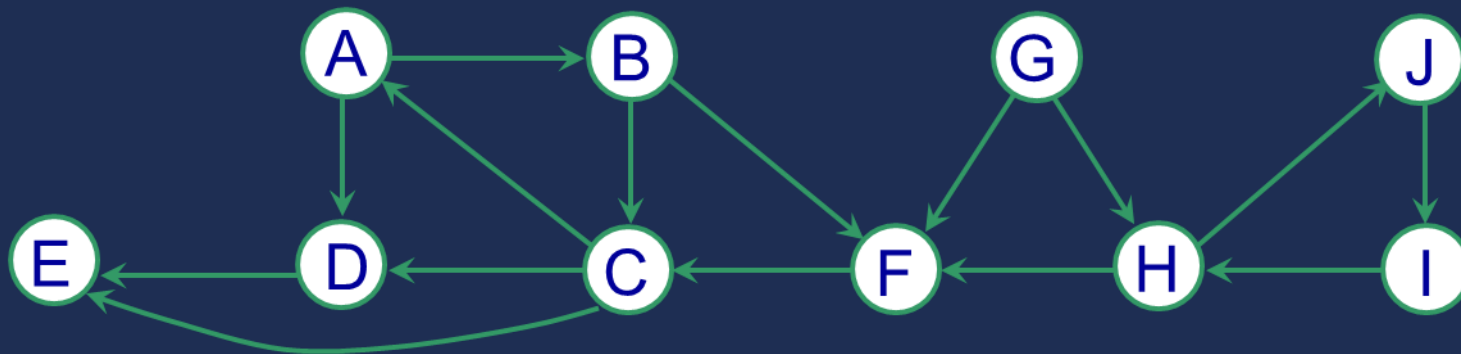


## 7.5 重连通图和关节点

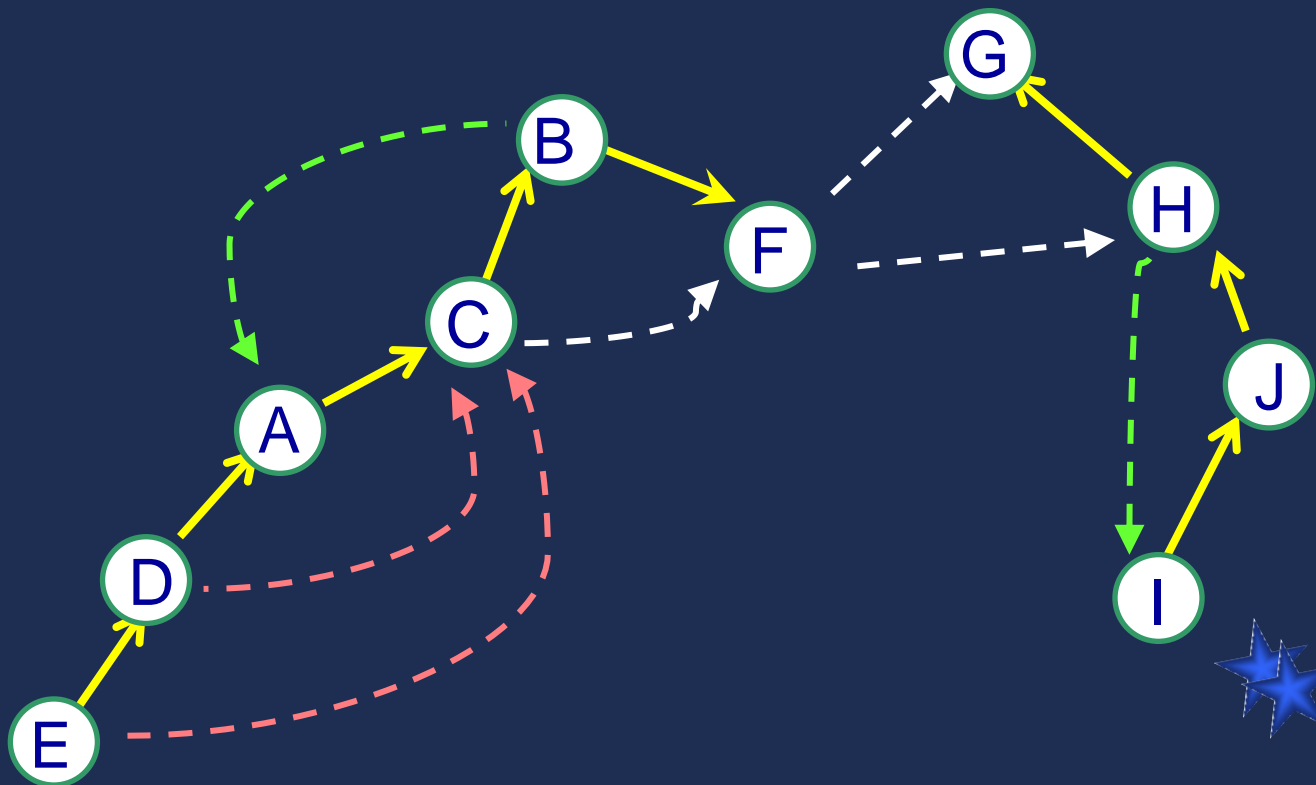
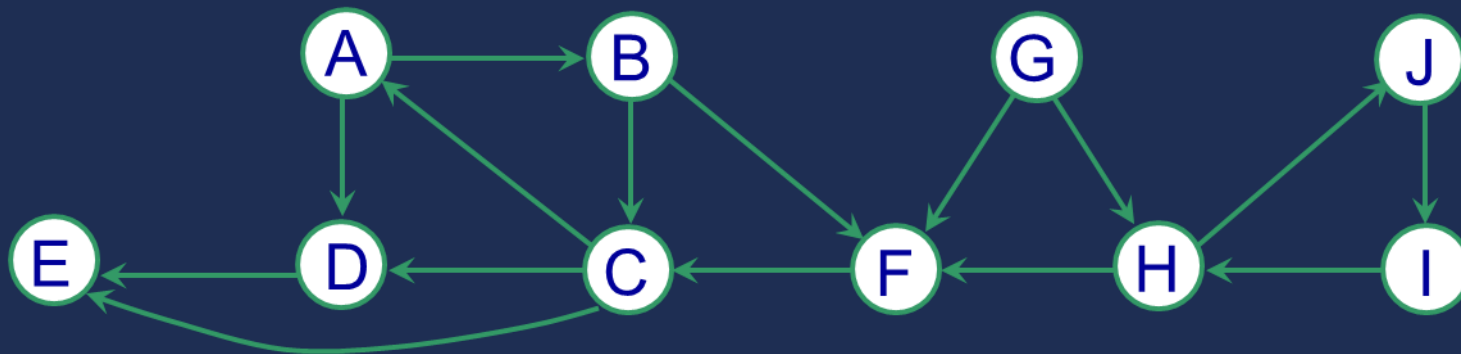
- 对G进行深度优先遍历，得到深度优先生成树T
- 虚线表示回边：即在G中但不在T中的边，是遍历时选择已访问的邻接点



## 7.5 强连通图

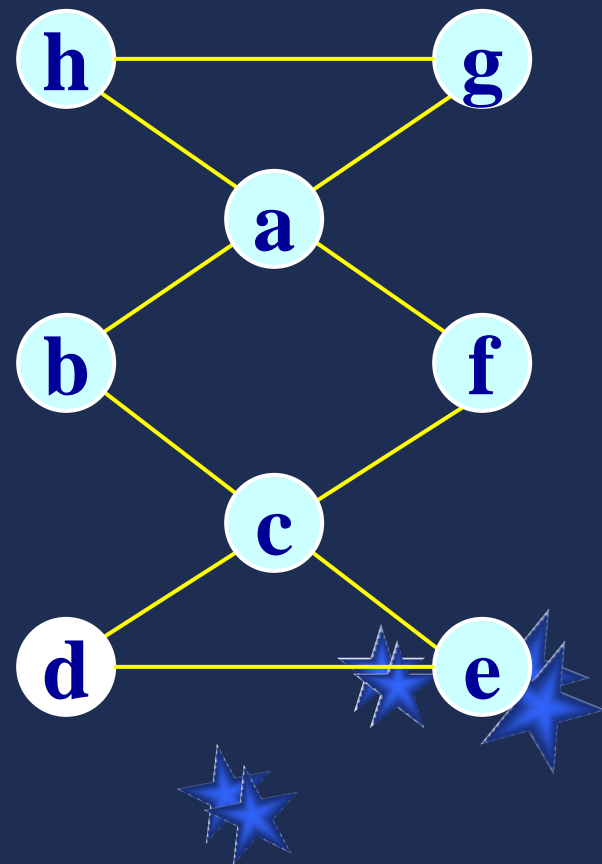
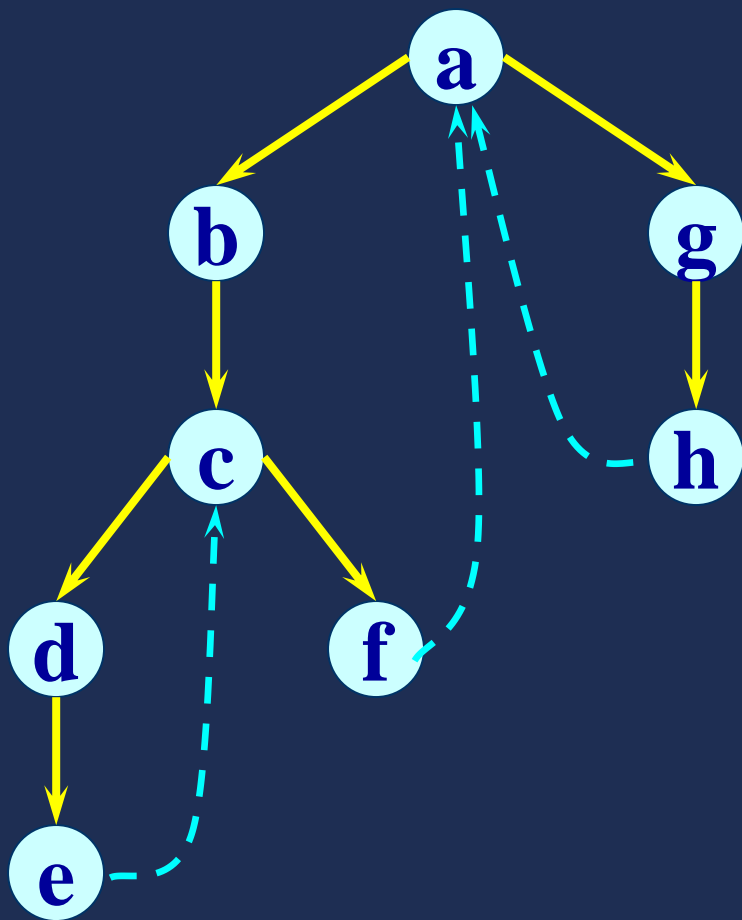


## 7.5 强连通图



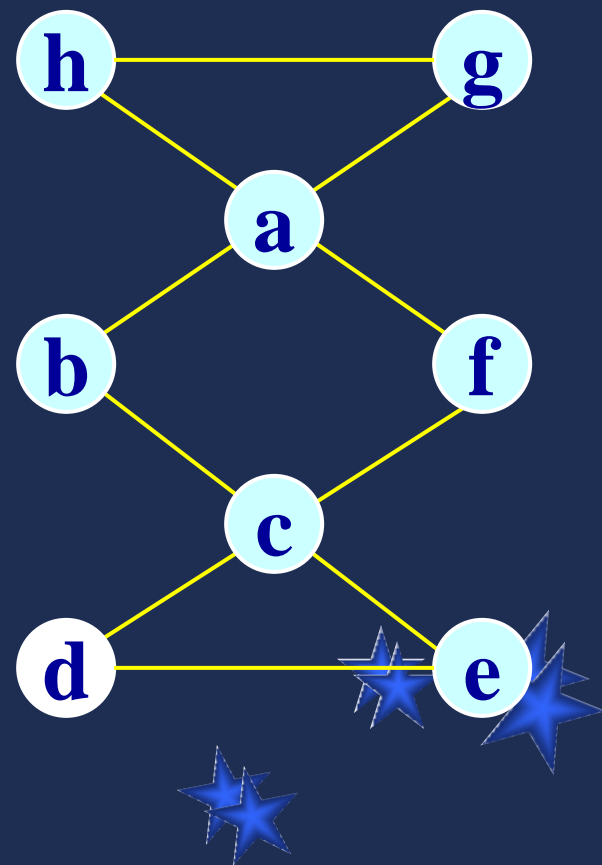
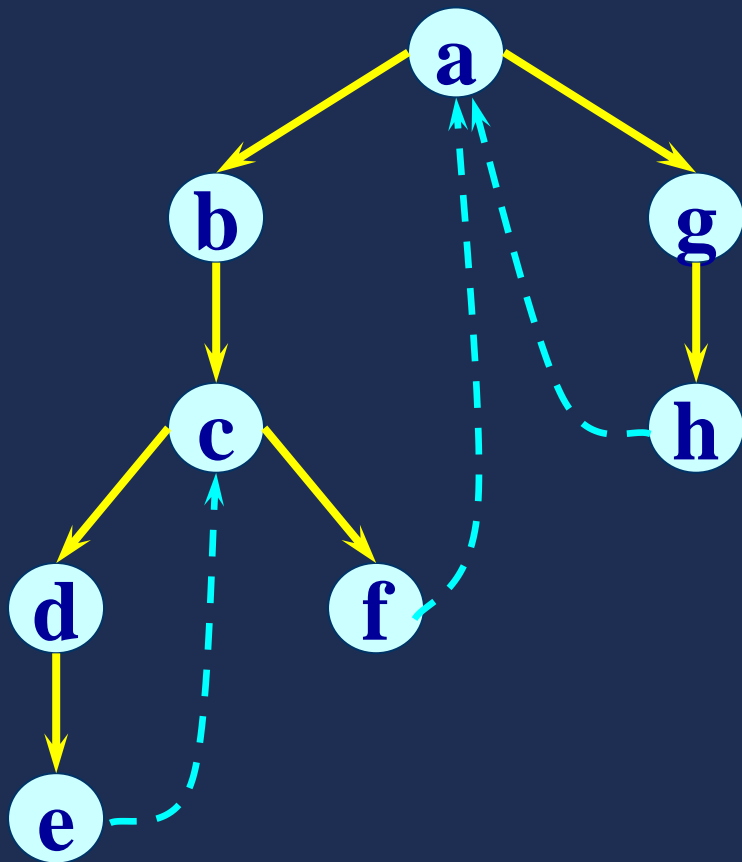
# 关节点的特征

- 特征1: 若生成树的根结点, 有**两个或两个以上**的分支, 则此顶点(生成树的根)必为**关节点**;

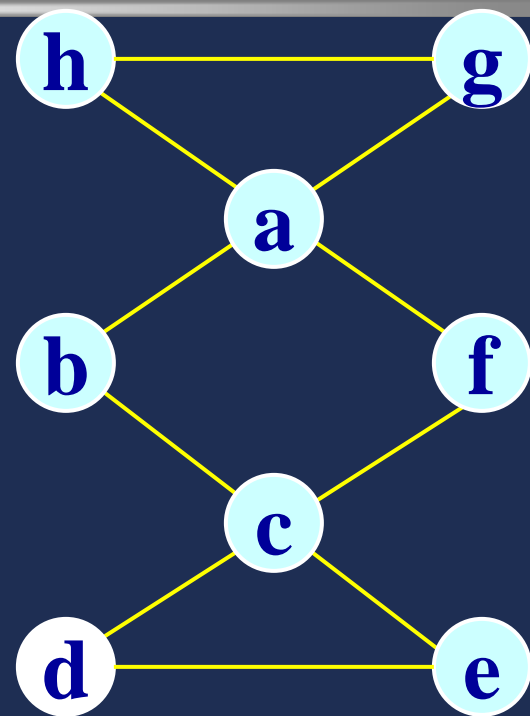
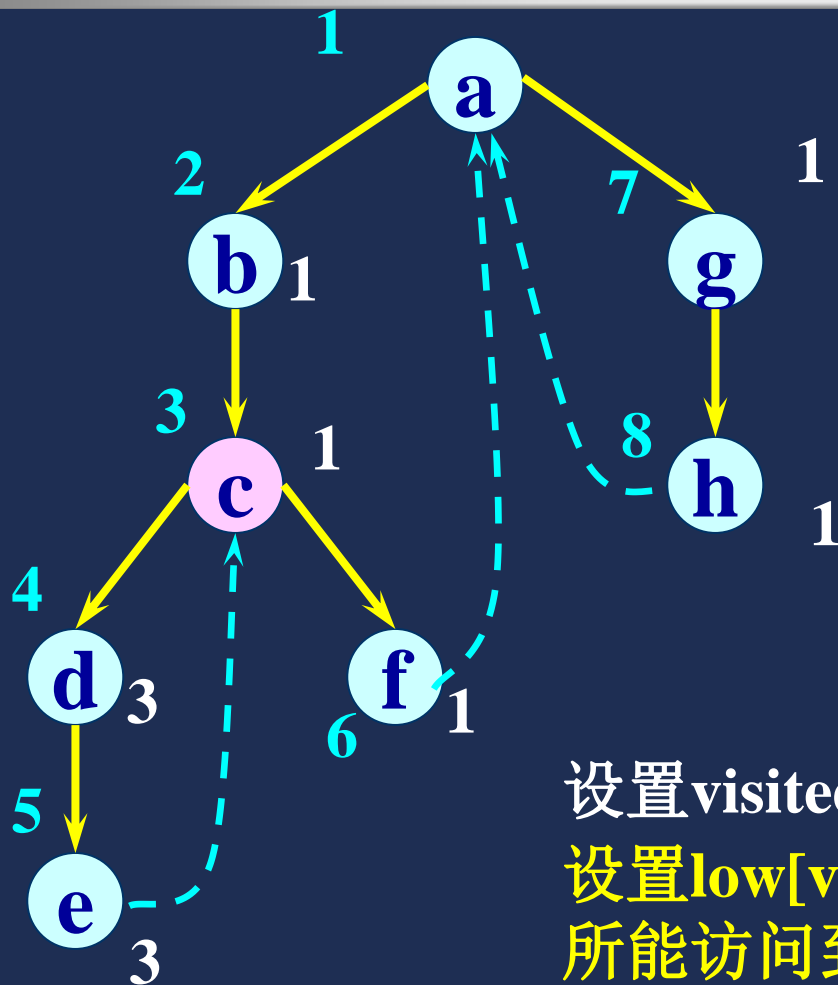


# 关节点的特征

- 特征2：对生成树上的任意一个“顶点”，若某棵子树的根或子树中的其它“顶点”没有和其祖先相通的回边，则该“顶点”必为关节点。
- 如何判断节点满足特征2？



## • 如何判断节点满足特征2？



设置visited[v]: 顶点在DFS中的序号;

设置low[v]: 以顶点v为根的子树在DFS中所能访问到的节点最小序号(包括回边)

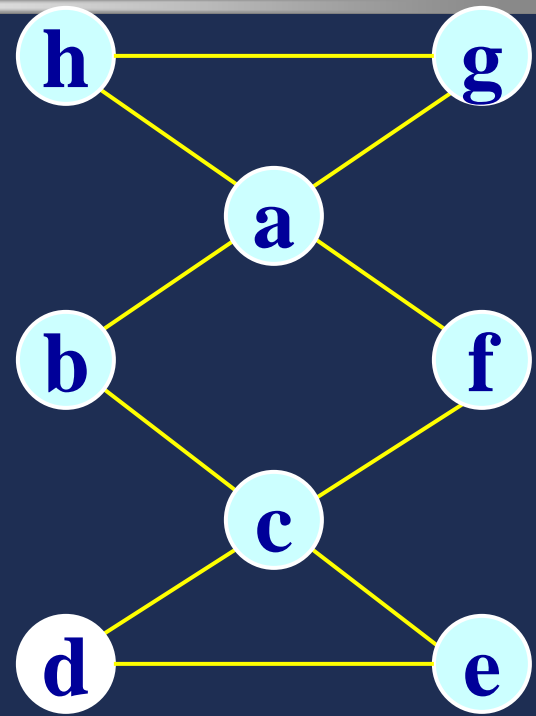
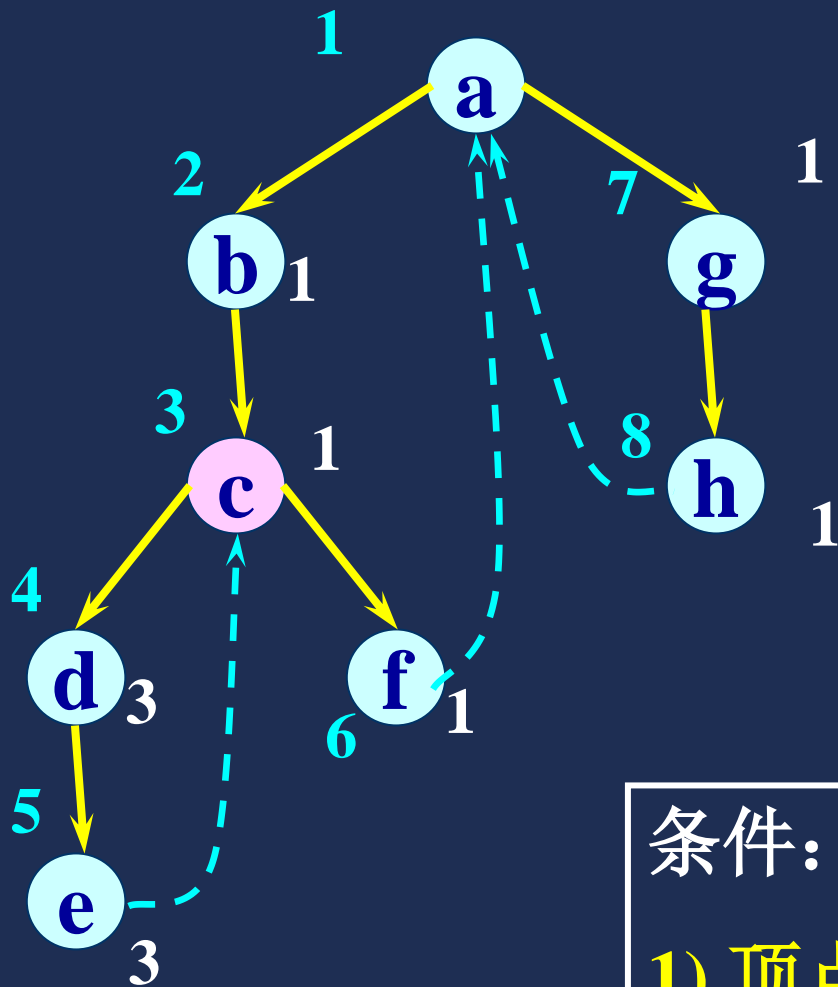
$$\text{low}[v] = \text{Min}\{ \text{visited}[v], \text{low}[w], \text{visited}[k] \}$$

w是顶点v在DFS树上的子节点;

k是顶点v在DFS树上回联的祖先节点;



## • 如何判断节点满足特征2?



条件:

- 1) 顶点 $v$ 存在孩子节点 $w$ ;
- 2)  $\text{visited}[v] \leq \text{low}[w]$ ;



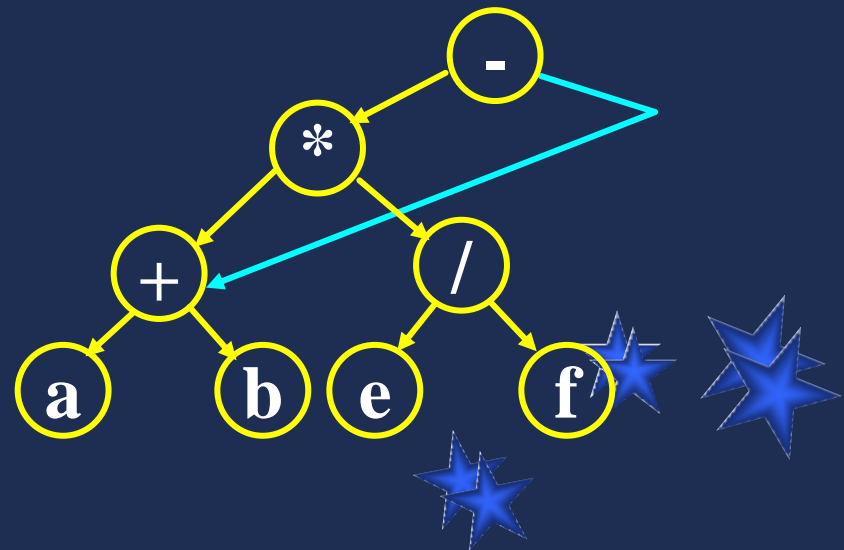
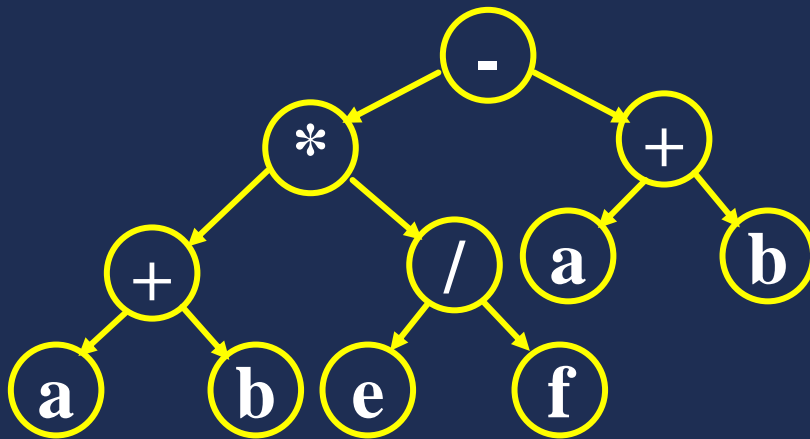
## 7.5 有向无环图

- 有向无环图(**DAG**, Directed Acyclic Graph)

没有回路的有向图。

➤ 含有公共子式的表达式

$$(a + b) * (e / f) - (a + b)$$



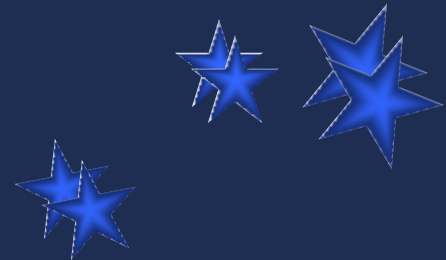
## 7.5 有向无环图

- 问题提出：学生选修课程问题

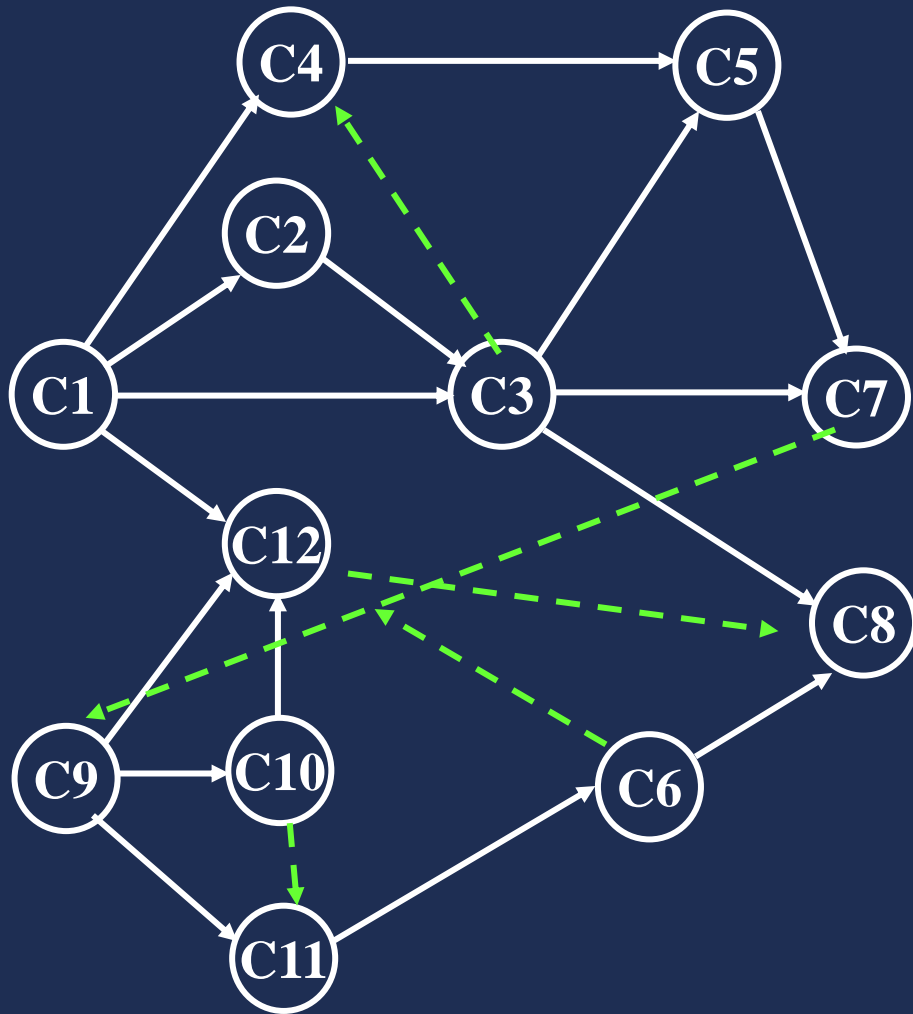
顶点——表示课程

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、顺利地完学业。



# 7.5 有向无环图



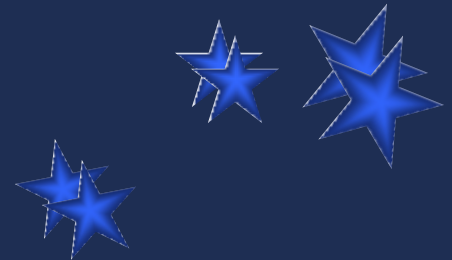
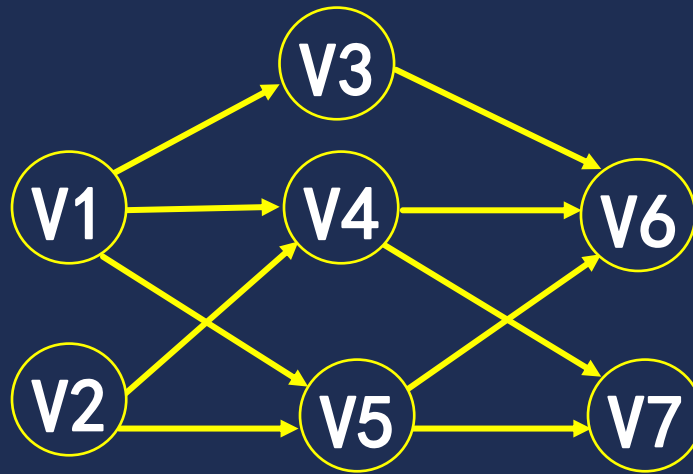
C1--C2--C3--C4--C5--C7--C9--  
C10--C11--C6--C12--C8

课程 代号	课程名称	先修课
<b>C1</b>	程序设计基础	无
<b>C2</b>	离散数学	C1
<b>C3</b>	数据结构	C1,C2
<b>C4</b>	汇编语言	C1
<b>C5</b>	语言的设计和分析	C3,C4
<b>C6</b>	计算机原理	C11
<b>C7</b>	编译原理	C3,C5
<b>C8</b>	操作系统	C3,C6
<b>C9</b>	高等数学	无
<b>C10</b>	线性代数	C9
<b>C11</b>	普通物理	C9
<b>C12</b>	数值分析	C1,C9,C10

## 7.5 有向无环图

- 有向无环图(DAG)

某工程可分为7个子工程，工程流程图。



## 7.5 有向无环图

- 定义

**AOV网**——用顶点表示活动，用弧表示活动间优先关系的有向图称为顶点表示活动的网（**Activity On Vertex network**），简称**AOV网**。

若  $\langle v_i, v_j \rangle$  是图中有向边，则  $v_i$  是  $v_j$  的直接前驱； $v_j$  是  $v_i$  的直接后继。

AOV网中不允许有回路，因为回路意味着某项活动以自己（或者后继）为先决条件。

## 7.5 有向无环图——拓扑排序

- 拓扑排序

把**AOV**网络中各顶点按照它们相互之间的优先关系排列成一个线性序列的过程。

检测**AOV**网中是否存在环方法：对有向图构造其顶点的拓扑有序序列，若网中所有顶点都在它的拓扑有序序列中，则该**AOV**网必定不存在环。



## 7.5 有向无环图——拓扑排序

- 拓扑排序的方法

在有向图中选一个没有前驱的顶点且输出之。

从图中删除该顶点和所有以它为尾的弧。

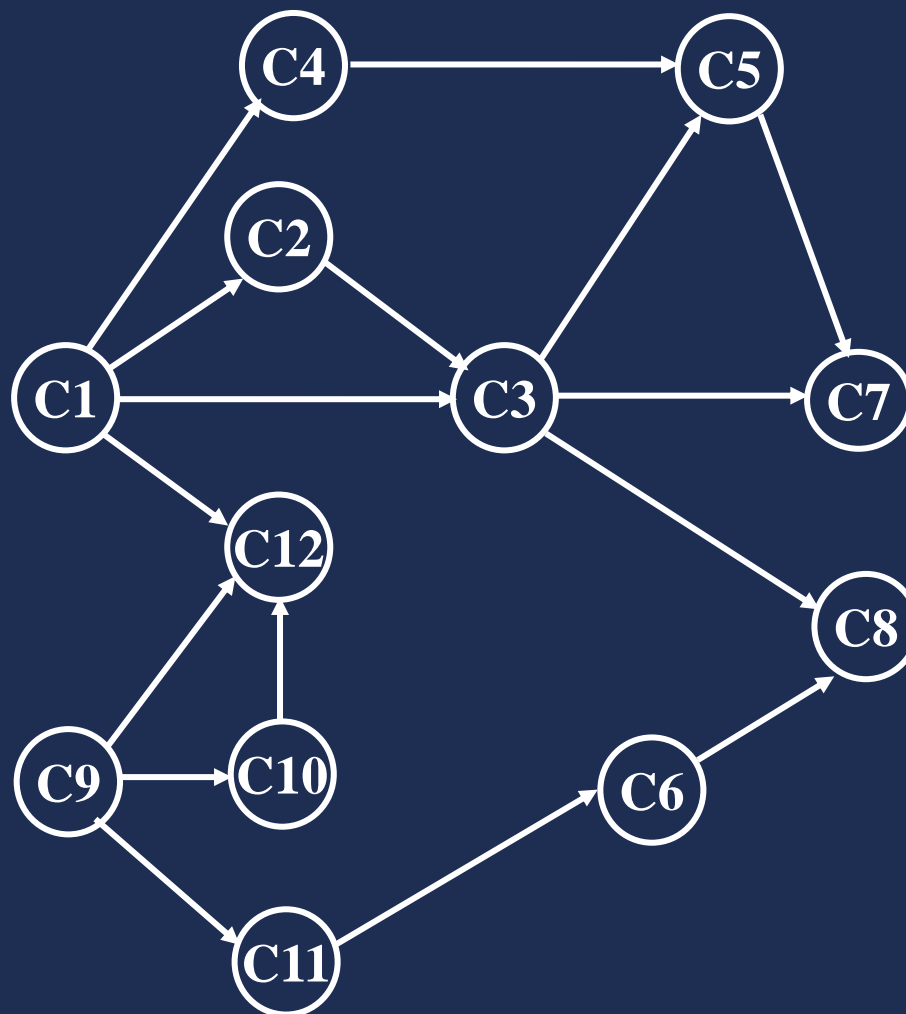
重复上述两步，直至全部顶点均已输出；或者当图中不存在无前驱的顶点为止。





## 7.5 有向无环图——拓扑排序

- 拓扑排序

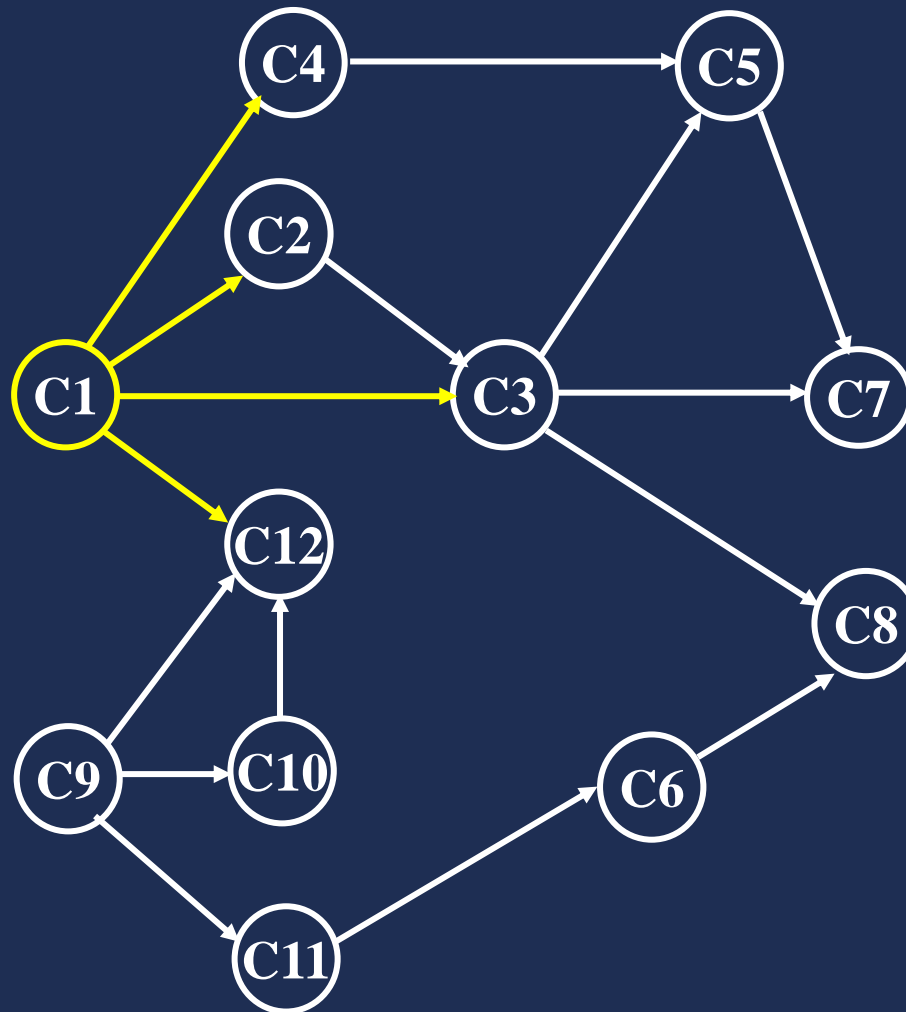


拓扑序列: C1



## 7.5 有向无环图——拓扑排序

- 拓扑排序

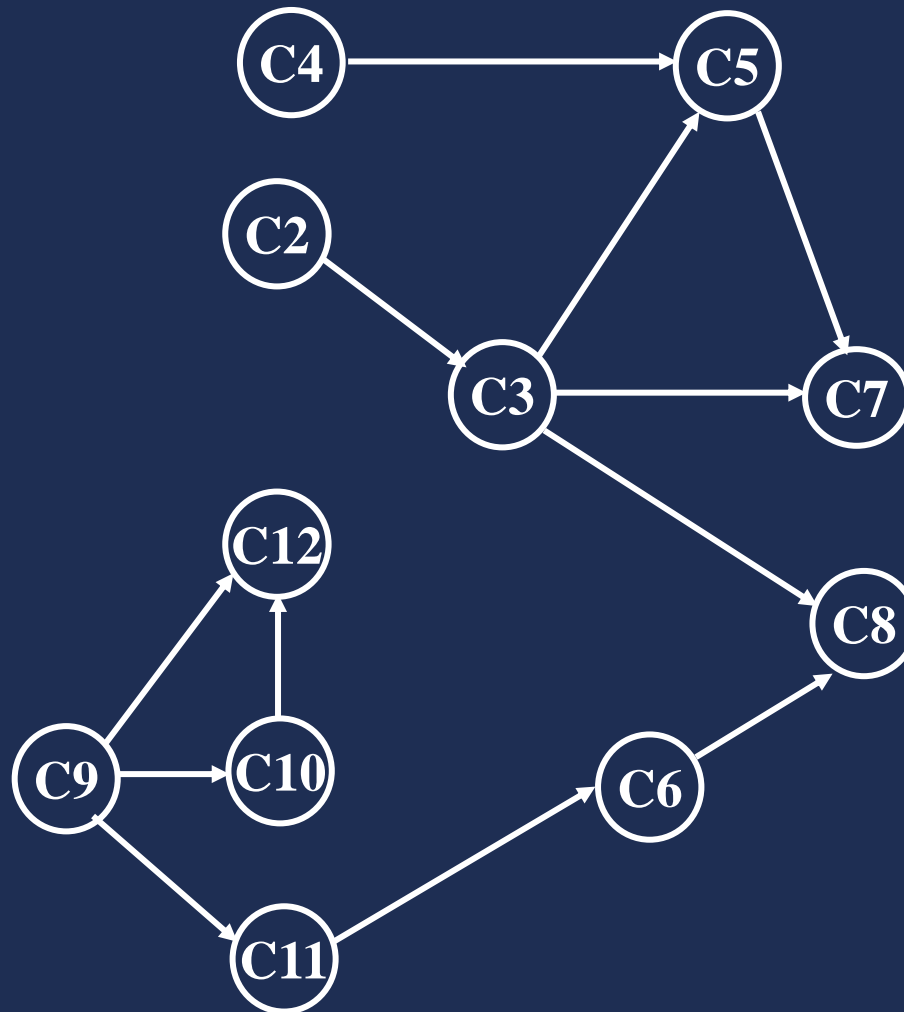


拓扑序列: C1



## 7.5 有向无环图——拓扑排序

- 拓扑排序

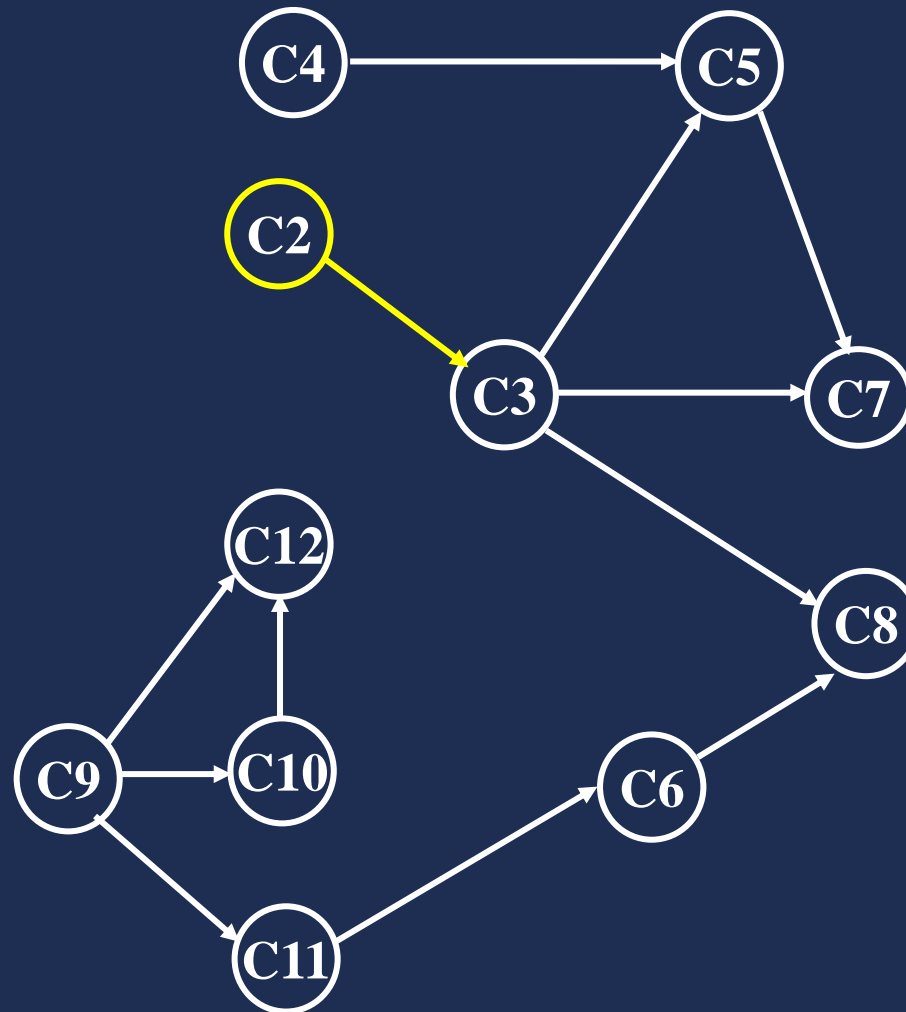


拓扑序列: C1 --C2



## 7.5 有向无环图——拓扑排序

- 拓扑排序

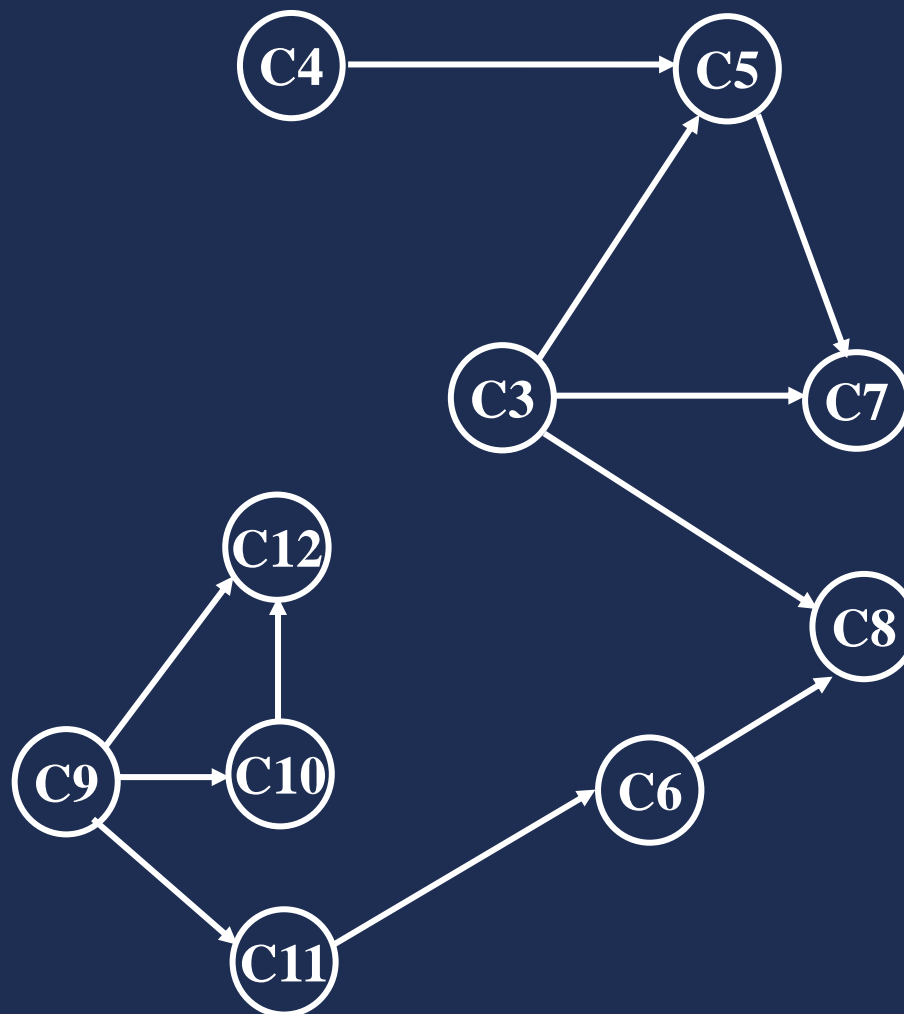


拓扑序列: C1 --C2



## 7.5 有向无环图——拓扑排序

- 拓扑排序

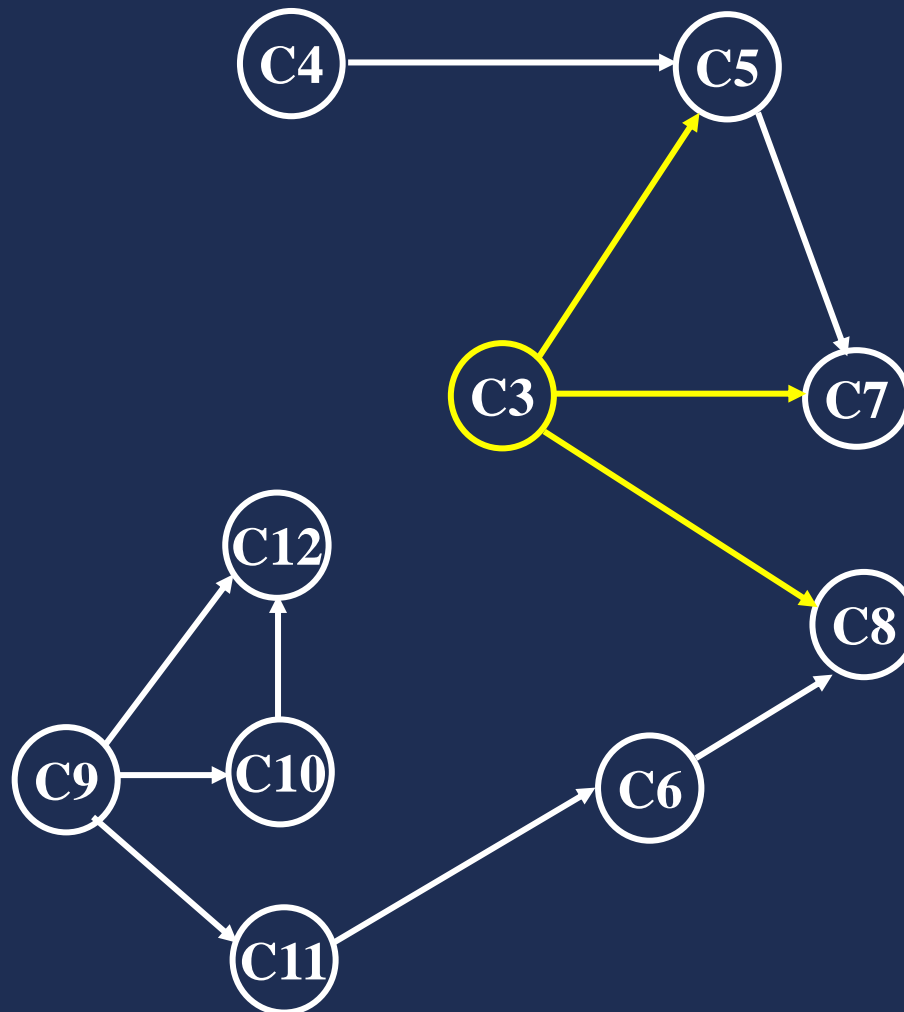


拓扑序列: C1 --C2 --C3



## 7.5 有向无环图——拓扑排序

- 拓扑排序

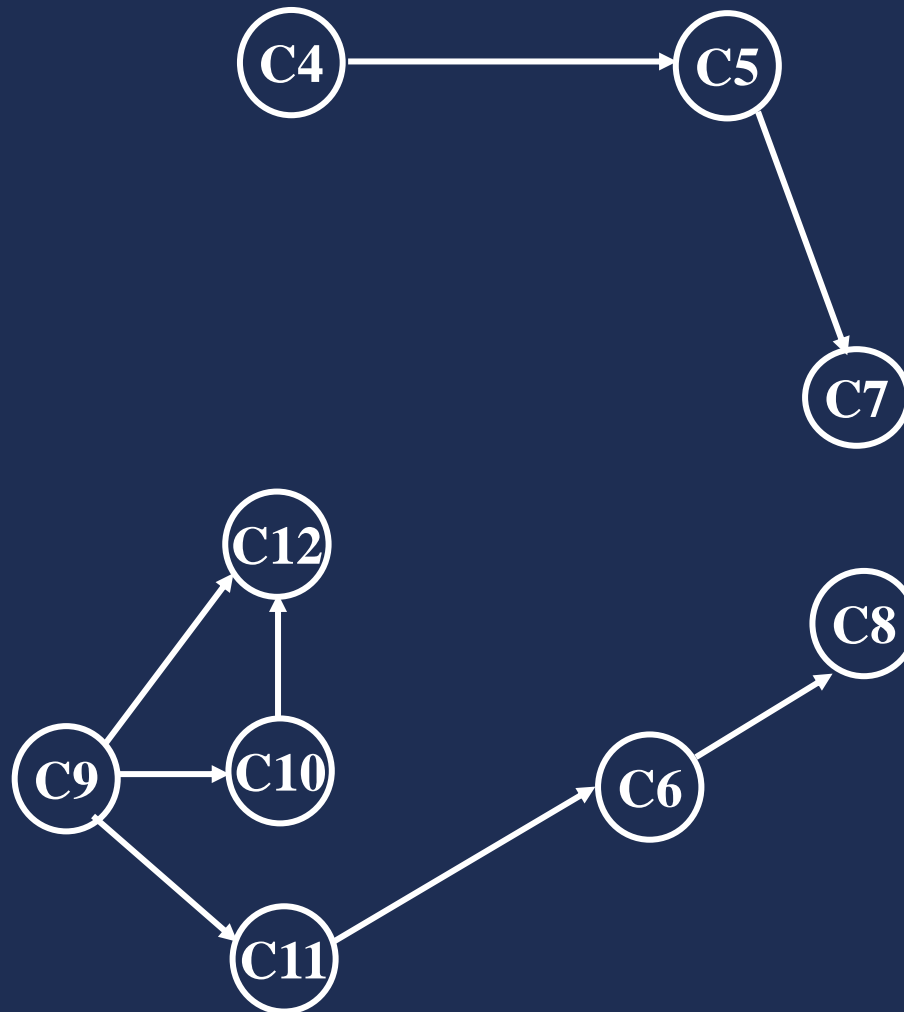


拓扑序列: C1 --C2 --C3



## 7.5 有向无环图——拓扑排序

- 拓扑排序

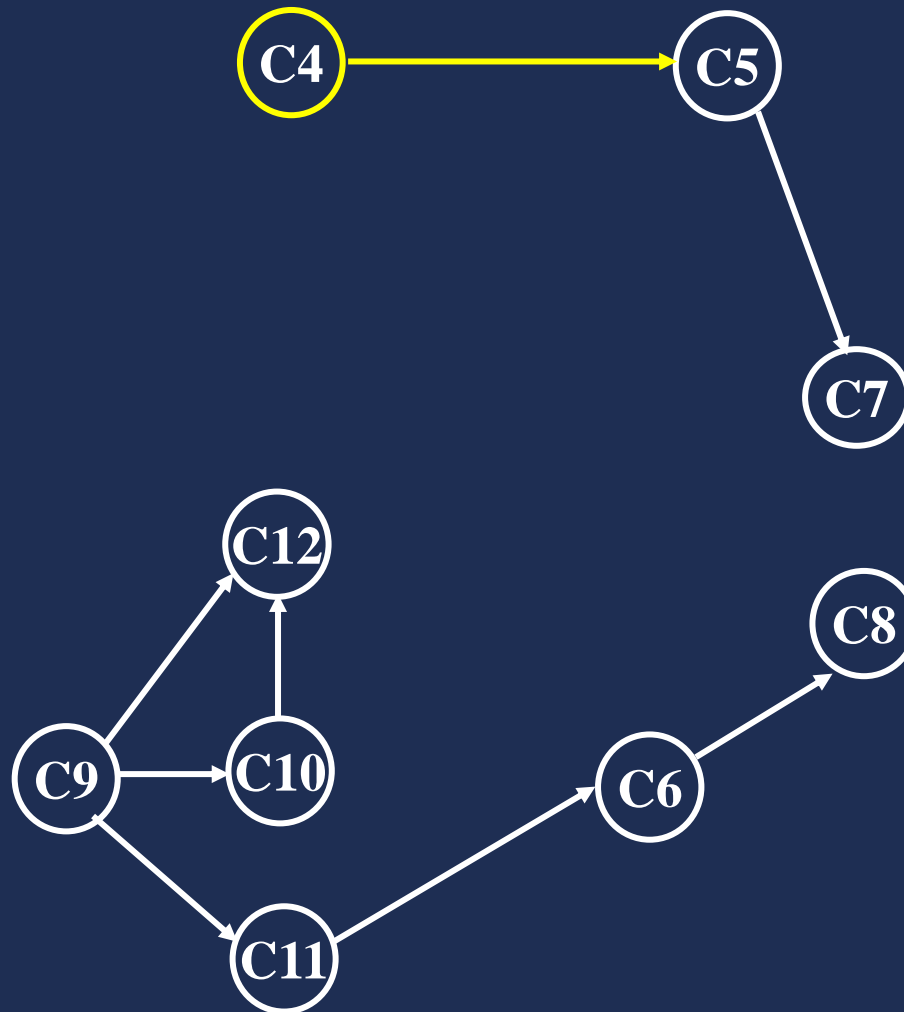


拓扑序列: C1 --C2 --C3 --C4



## 7.5 有向无环图——拓扑排序

- 拓扑排序



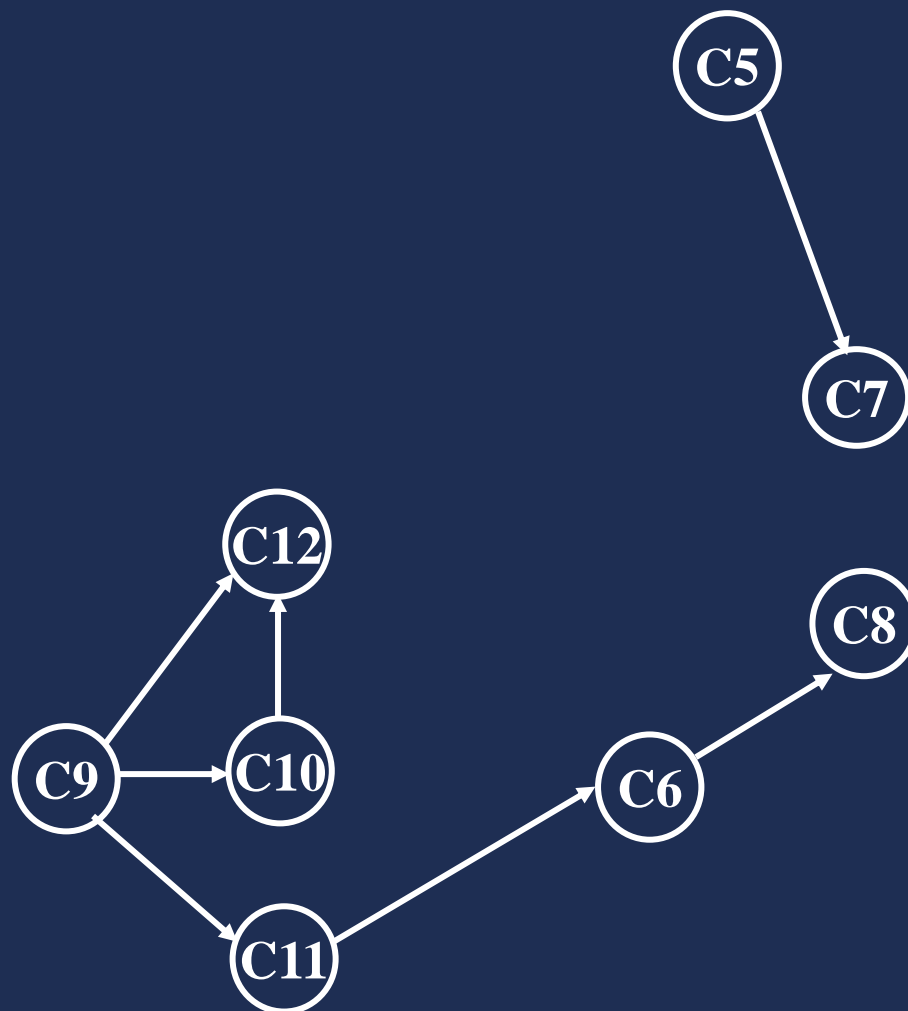
拓扑序列: C1 --C2 --C3 --C4





## 7.5 有向无环图——拓扑排序

- 拓扑排序

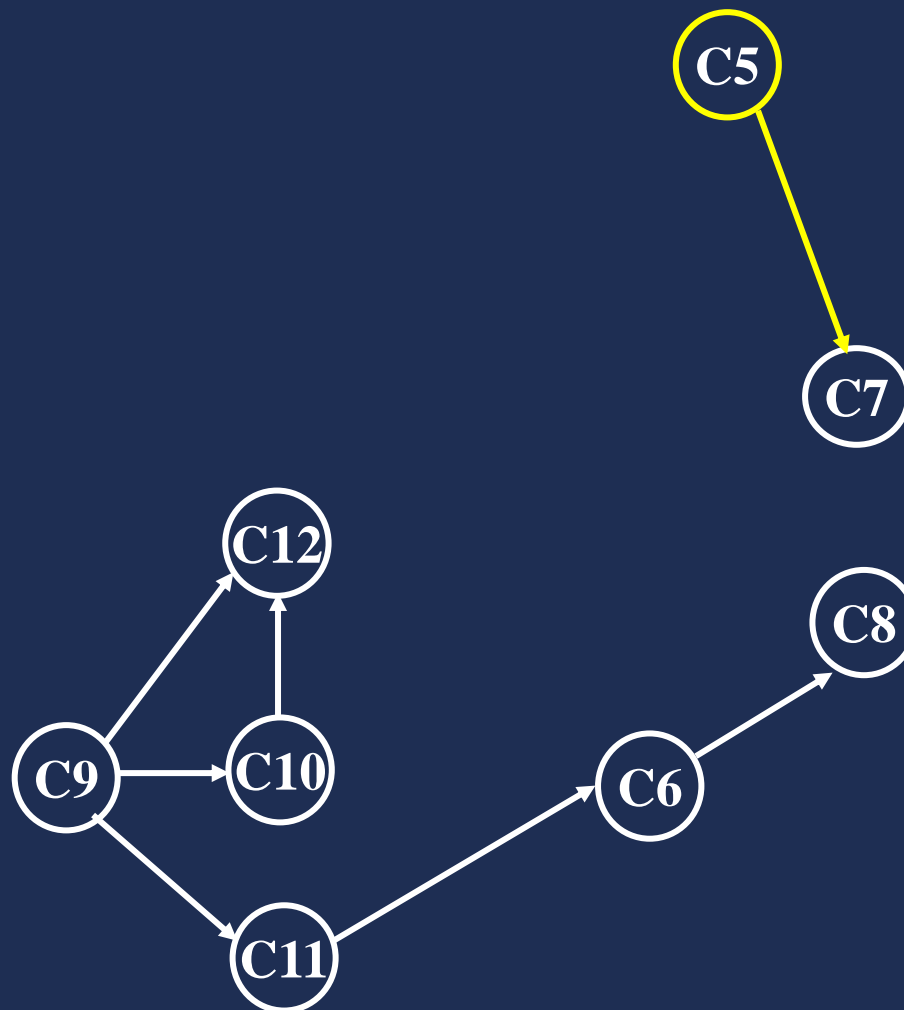


拓扑序列: C1 --C2 --C3 --C4 --C5



## 7.5 有向无环图——拓扑排序

- 拓扑排序

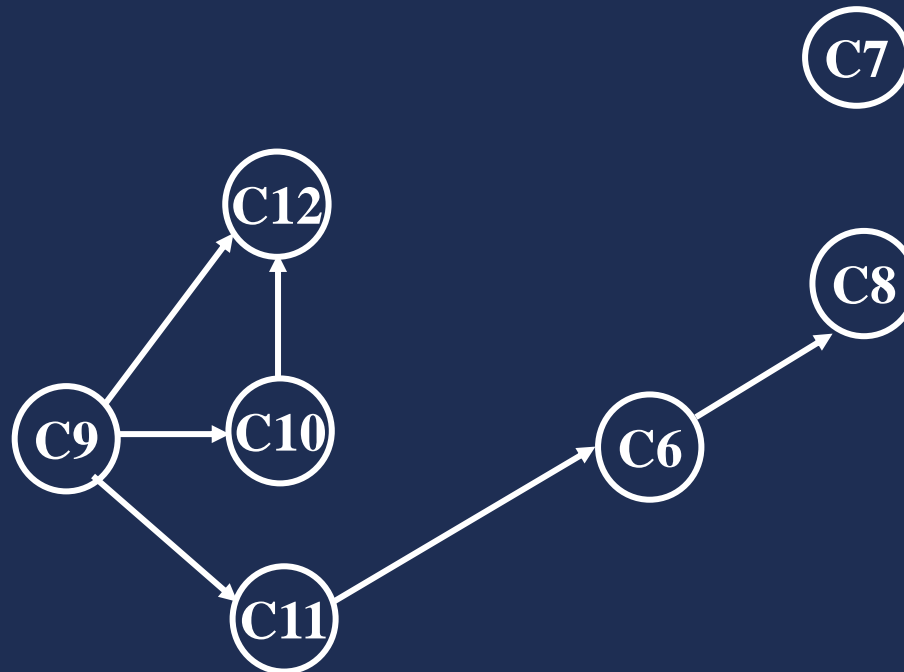


拓扑序列: C1 --C2 --C3 --C4 --C5



## 7.5 有向无环图——拓扑排序

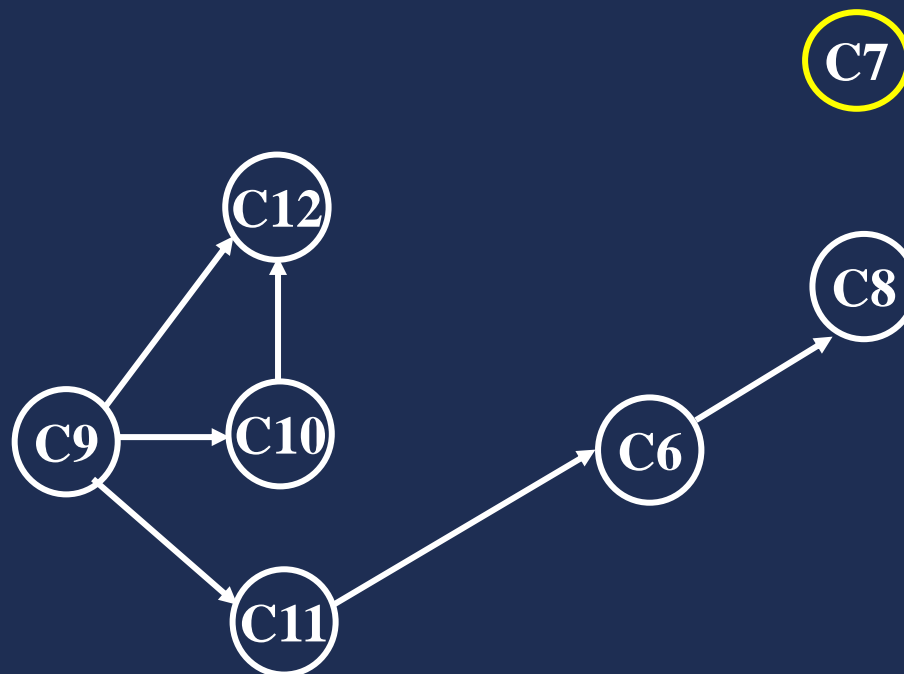
- 拓扑排序



拓扑序列: C1 --C2 --C3 --C4 --C5 --C7

## 7.5 有向无环图——拓扑排序

- 拓扑排序

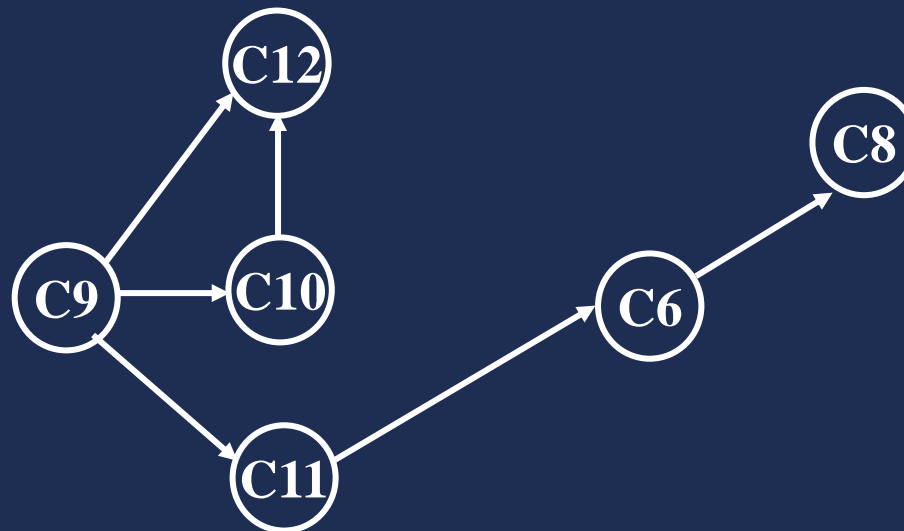


拓扑序列: C1 --C2 --C3 --C4 --C5 --C7



## 7.5 有向无环图——拓扑排序

- 拓扑排序

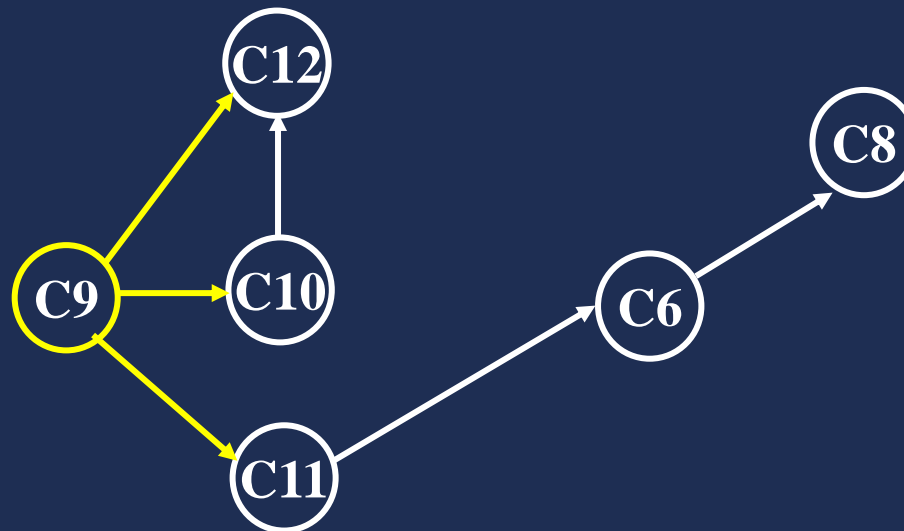


拓扑序列: C1 --C2 --C3 --C4 --C5 --C7 --C9

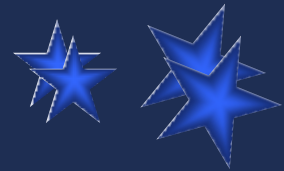


## 7.5 有向无环图——拓扑排序

- 拓扑排序

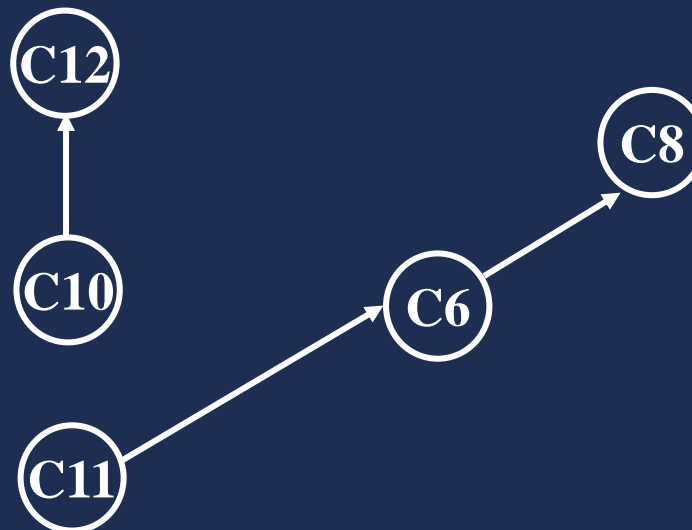


拓扑序列: C1 --C2 --C3 --C4 --C5 --C7 --C9



## 7.5 有向无环图——拓扑排序

- 拓扑排序



拓扑序列: C1 --C2 --C3 --C4 --C5 --C7 --C9 --C10



## 7.5 有向无环图——拓扑排序

- 拓扑排序

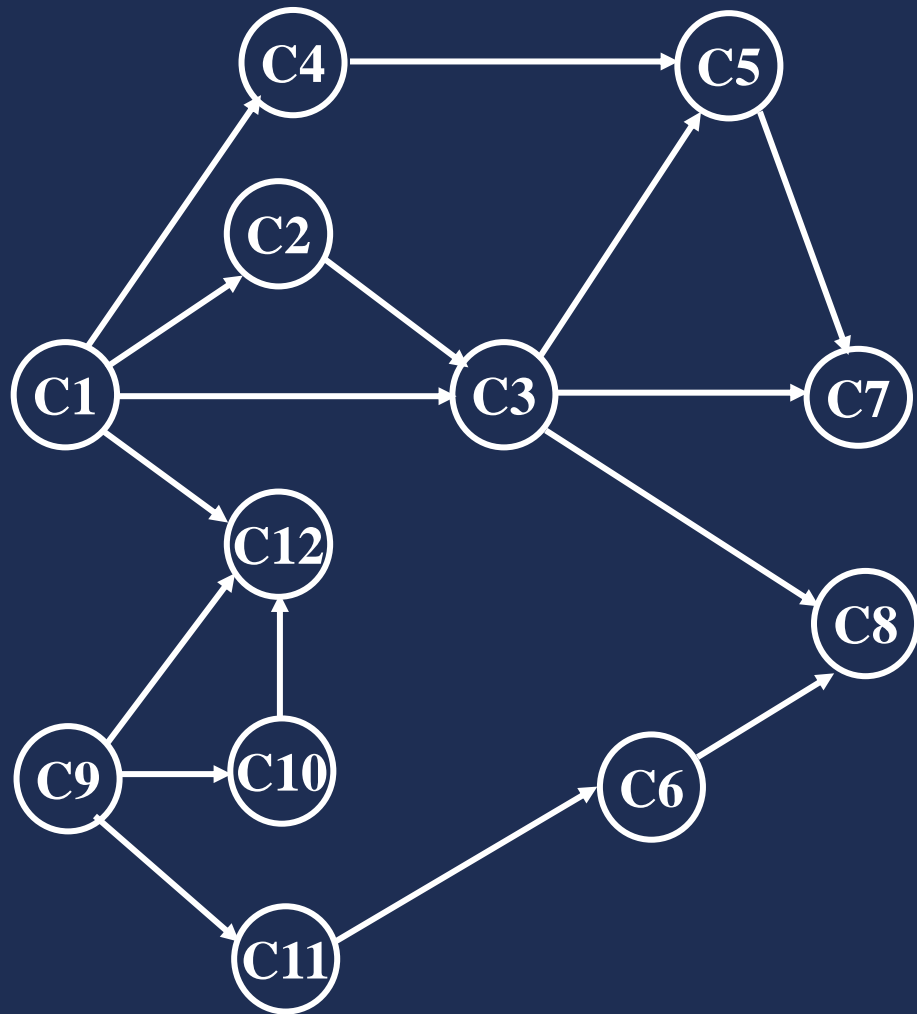
拓扑序列: C1 --C2 --C3 --C4 --C5 --C7 --C9 --C10  
--C11 --C6 --C12--C8





## 7.5 有向无环图——拓扑排序

- 拓扑排序



拓扑序列: C1--C2--  
C3--C4--C5--C7--  
C9--C10--C11--C6--  
C12--C8

或 : C9--C10--  
-C11--C6--C1--C12--  
C4--C2--C3--C5--  
C7--C8

一个AOV网的拓  
扑序列不是唯一的

## 7.5 有向无环图——拓扑排序

- 拓扑排序算法实现

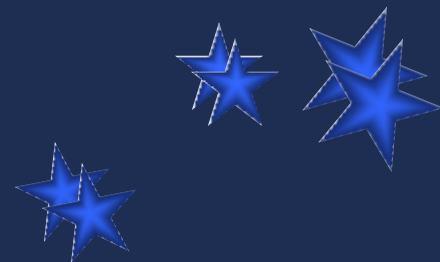
以邻接表作存储结构。

把邻接表中所有入度为0的顶点进栈。

栈非空时，输出栈顶元素  $V_j$  并退栈；在邻接表中查找  $V_j$  的直接后继  $V_k$ ，把  $V_k$  的入度减1；若  $V_k$  的入度为 0 则进栈。

重复上述操作直至栈空为止。

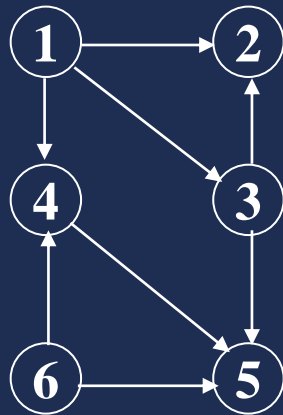
若栈空时输出的顶点个数不是  $n$ ，则有向图有环；否则，拓扑排序完毕。



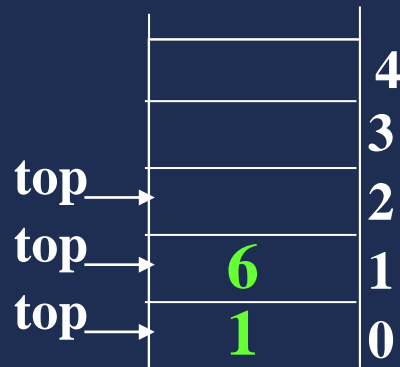
# 7.5 有向无环图——拓扑排序

- 拓扑排序

例

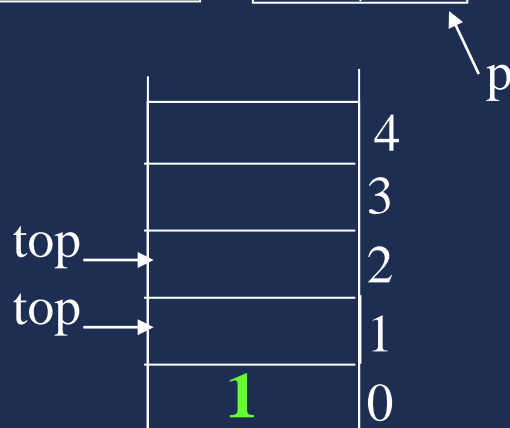
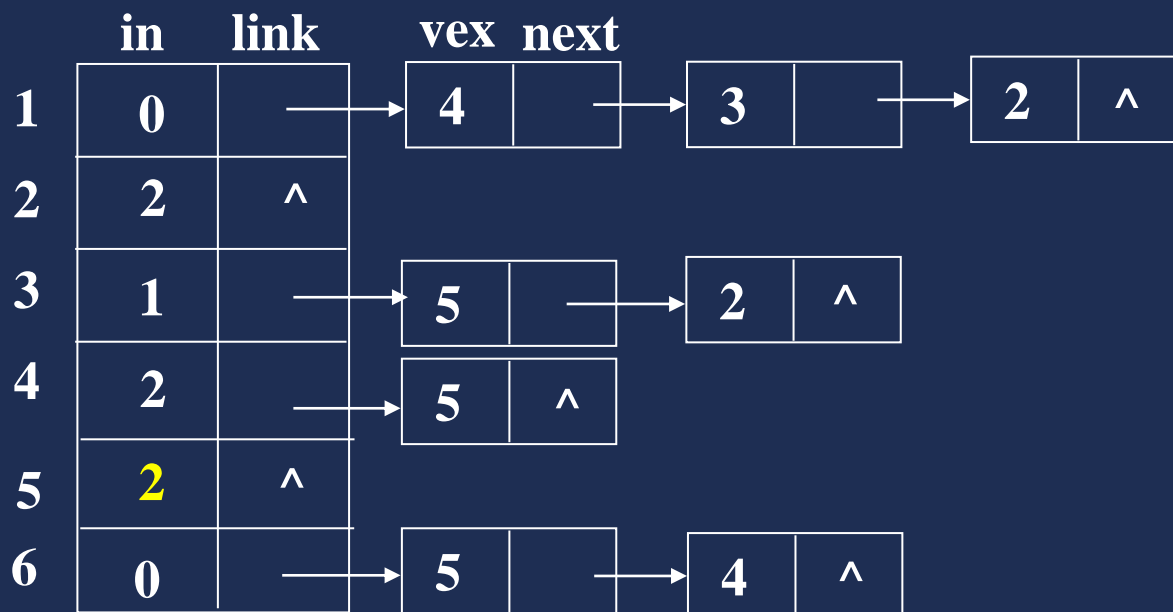


	in	link	vex	next
1	0	—	4	3 → 2 ^
2	2	^		
3	1	—	5	2 ^
4	2	—	5	^
5	3	^		
6	0	—	5	4 ^



# 7.5 有向无环图——拓扑排序

- 拓扑排序

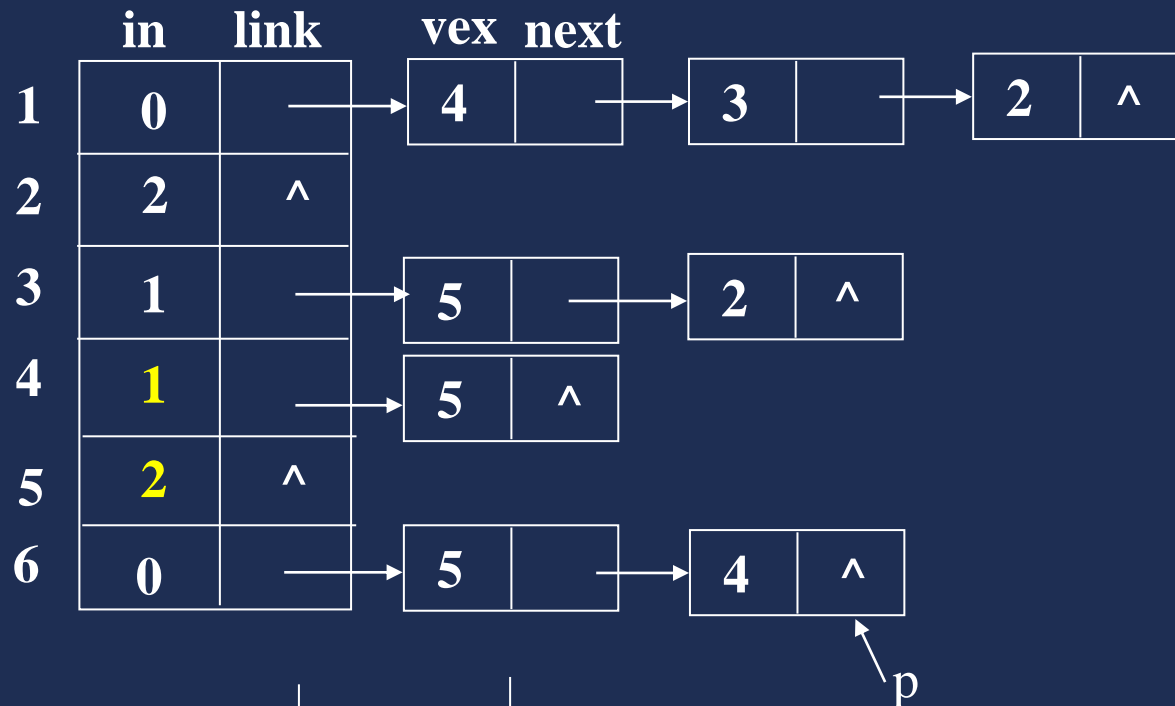


输出序列: 6



# 7.5 有向无环图——拓扑排序

- 拓扑排序

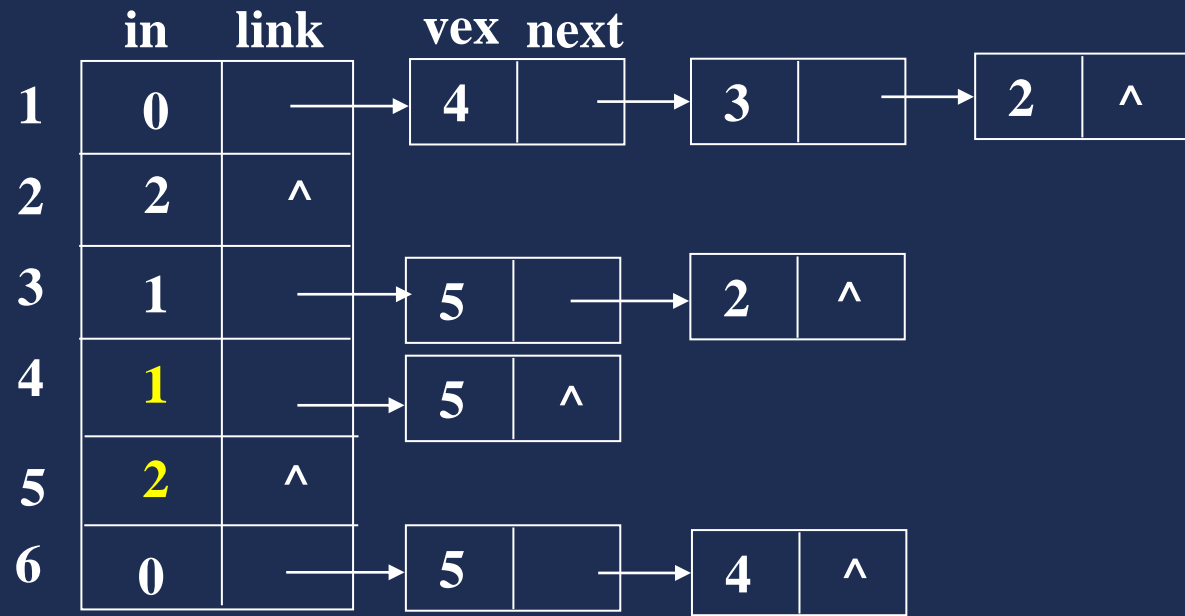


输出序列: 6

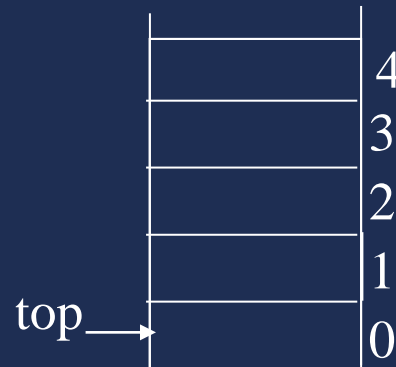


# 7.5 有向无环图——拓扑排序

- 拓扑排序



P=NULL

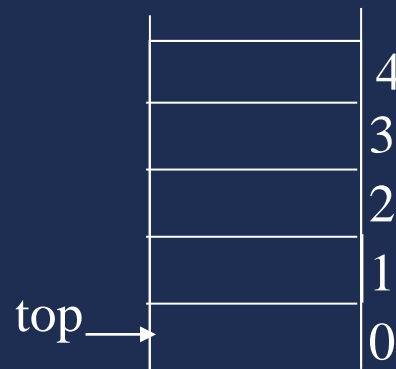
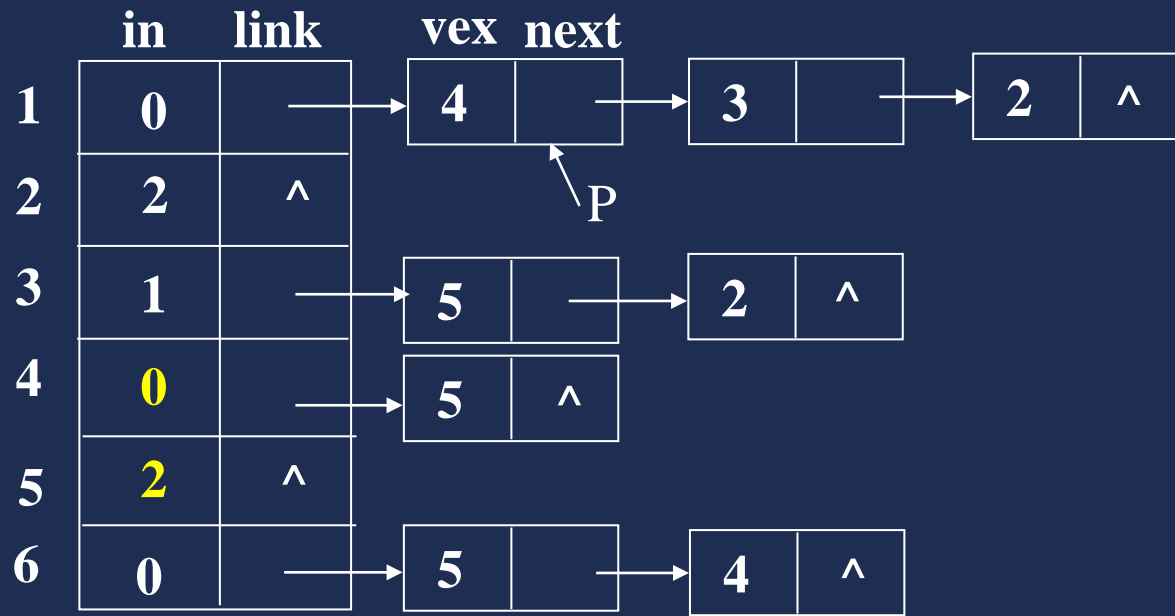


输出序列: 6 1



# 7.5 有向无环图——拓扑排序

- 拓扑排序

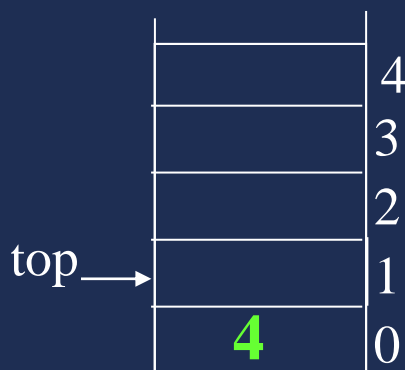
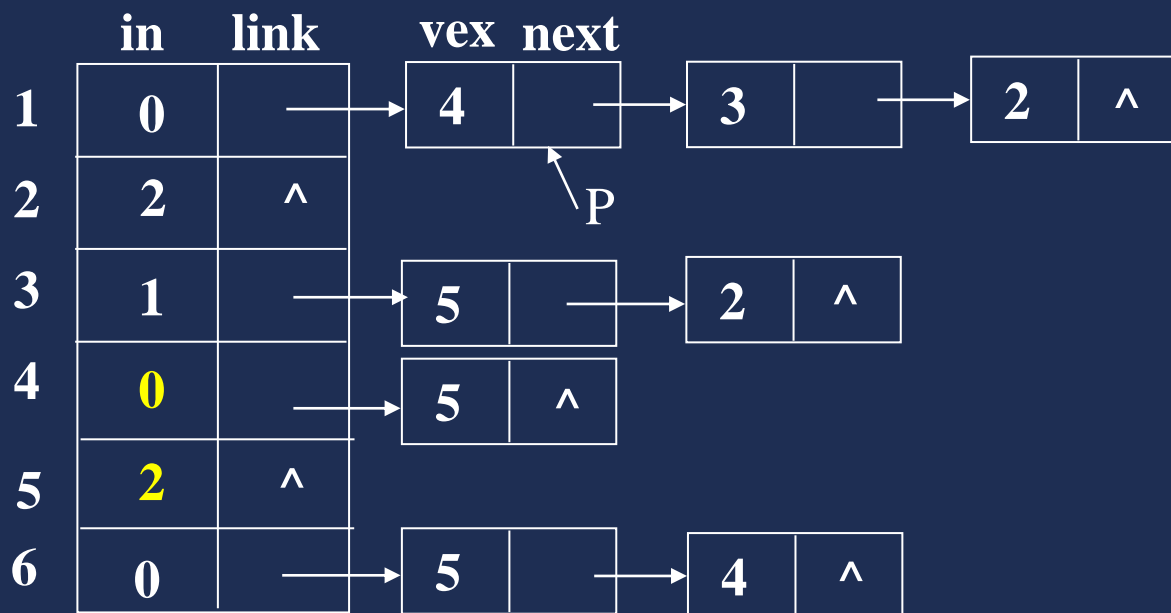


输出序列: 6 1



# 7.5 有向无环图——拓扑排序

- 拓扑排序



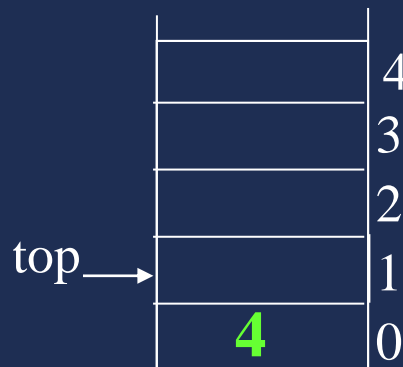
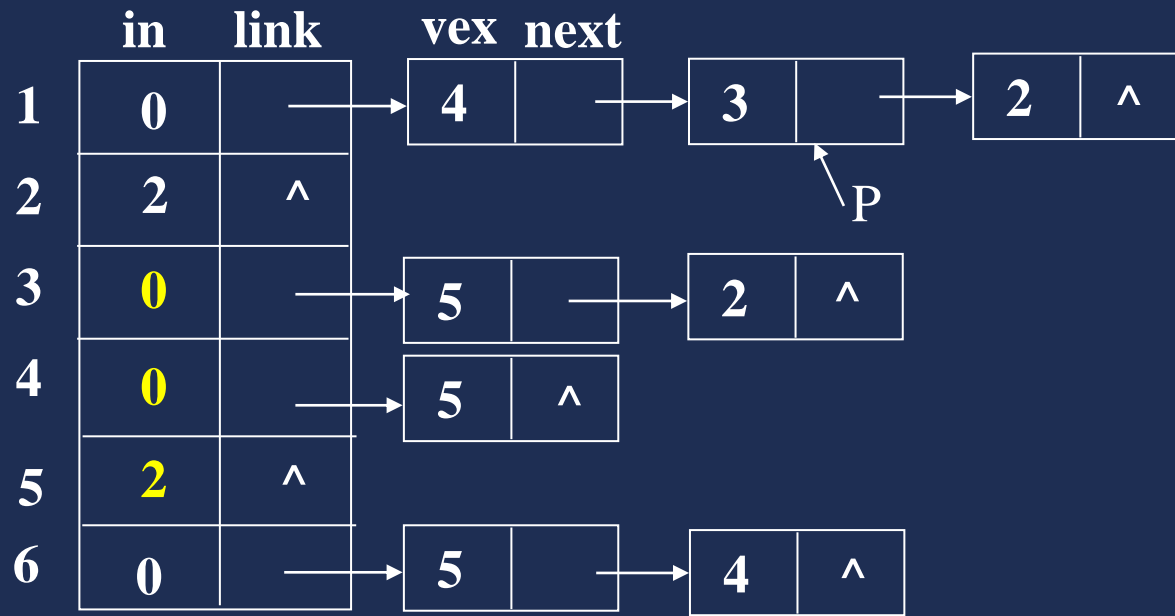
输出序列: 6 1





# 7.5 有向无环图——拓扑排序

- 拓扑排序

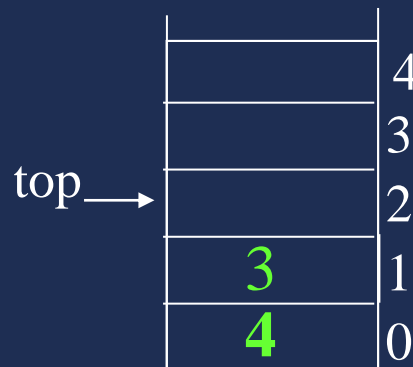
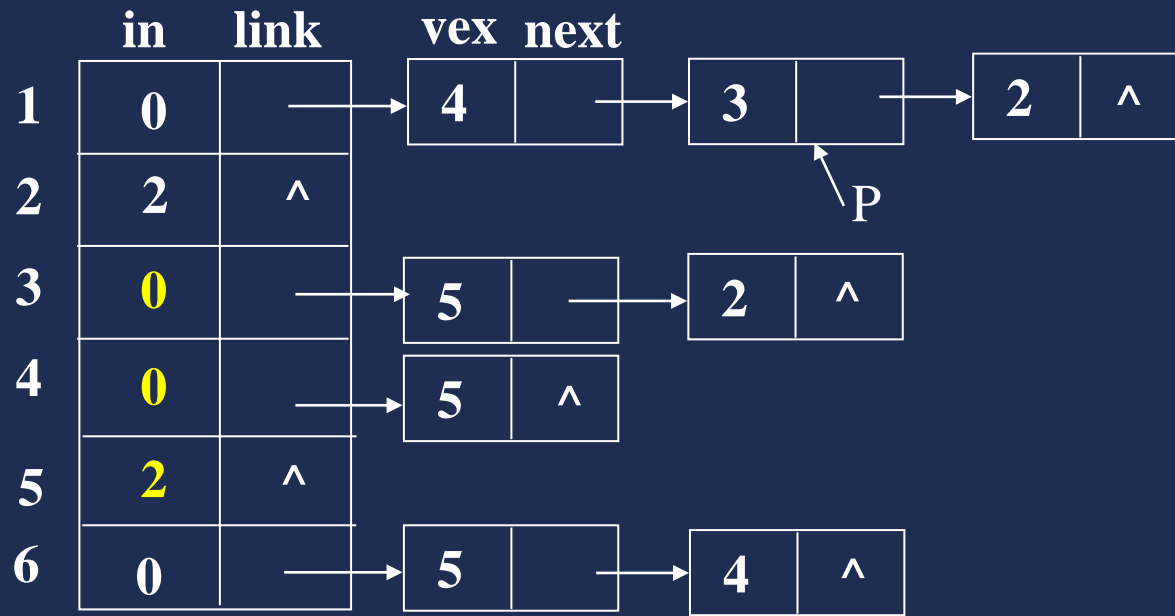


输出序列: 6 1



# 7.5 有向无环图——拓扑排序

- 拓扑排序

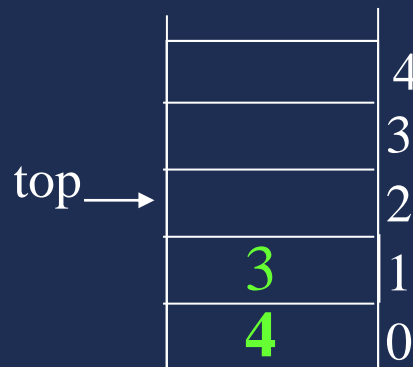
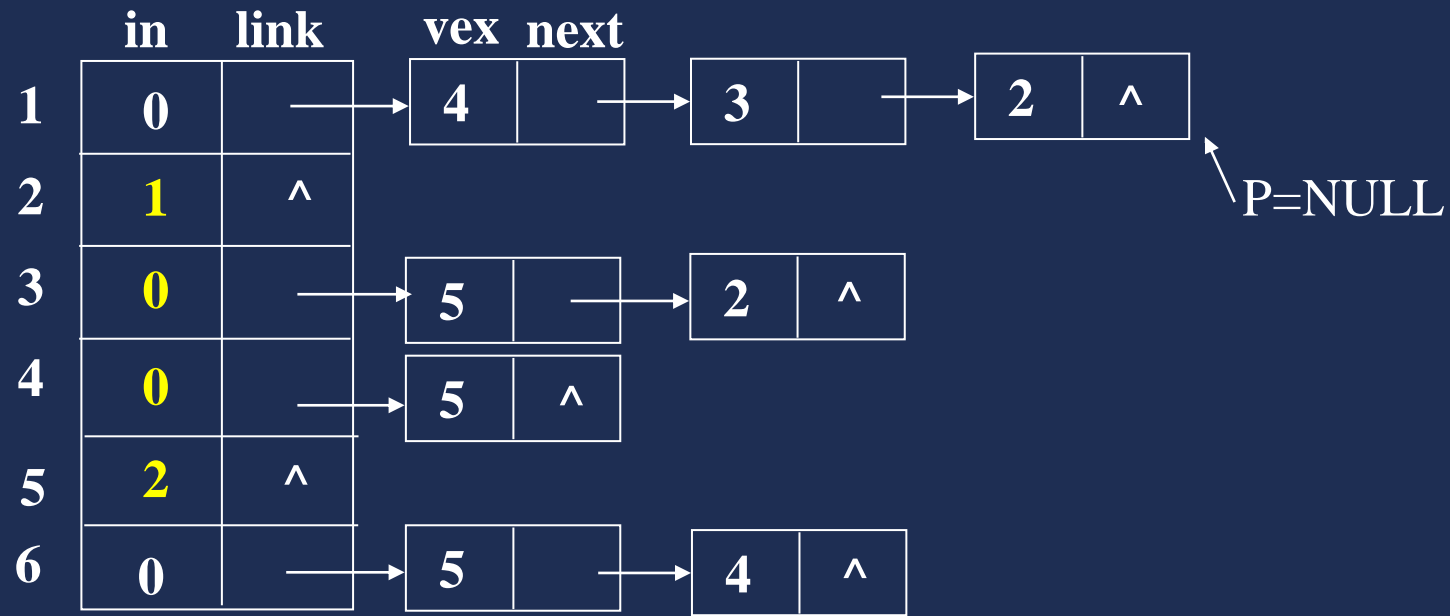


输出序列: 6 1



# 7.5 有向无环图——拓扑排序

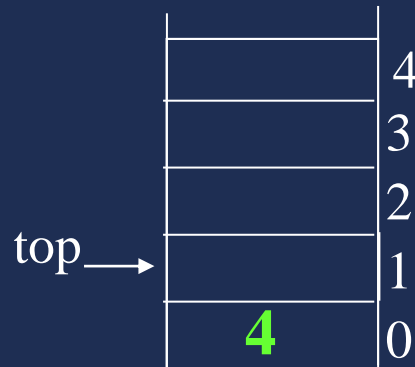
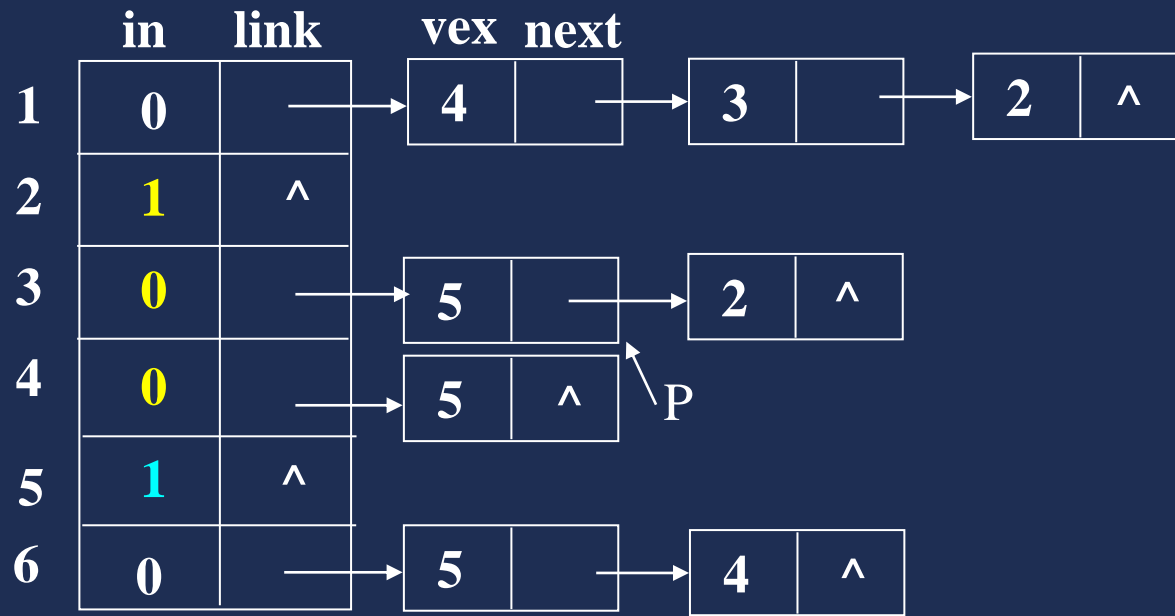
- 拓扑排序



输出序列: 6 1

# 7.5 有向无环图——拓扑排序

- 拓扑排序

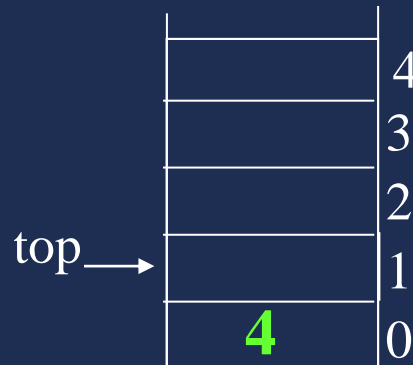
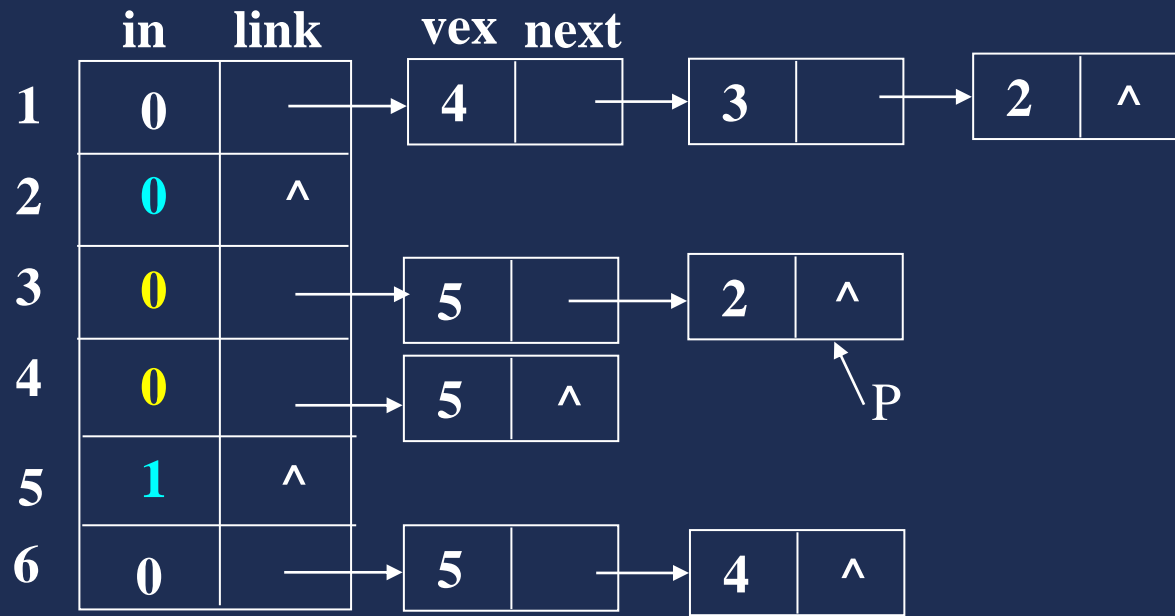


输出序列: 6 1 3



# 7.5 有向无环图——拓扑排序

- 拓扑排序

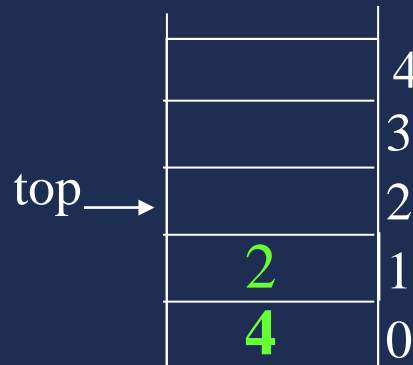
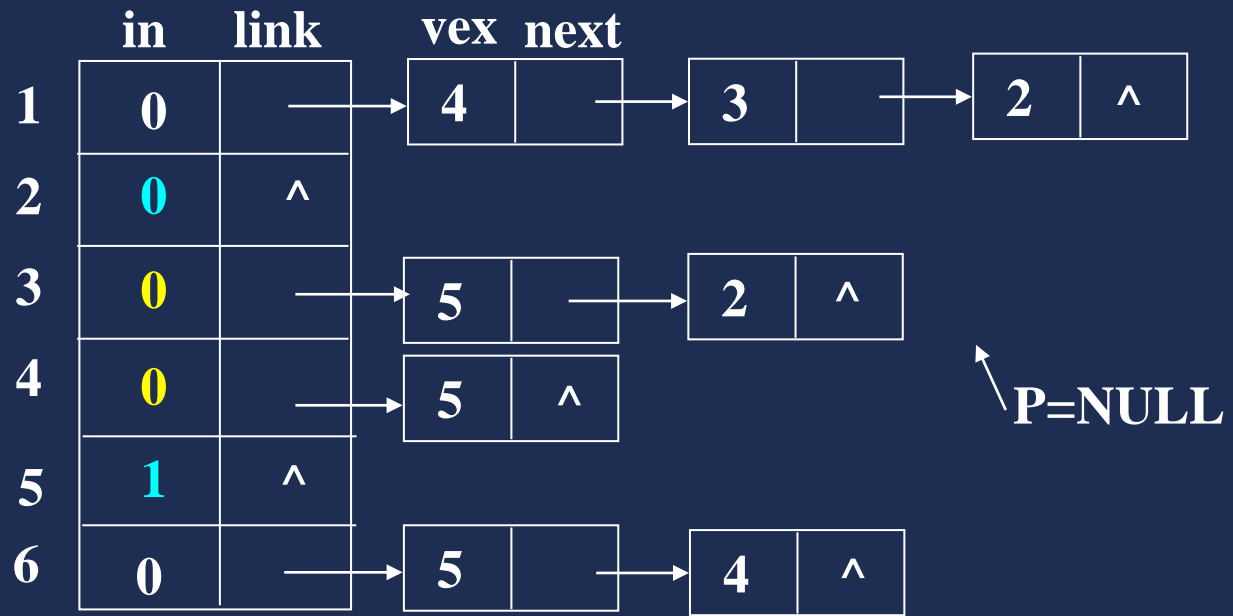


输出序列: 6 1 3



# 7.5 有向无环图——拓扑排序

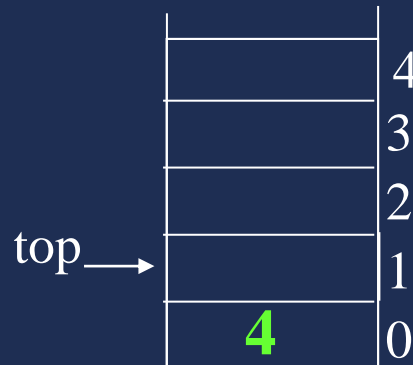
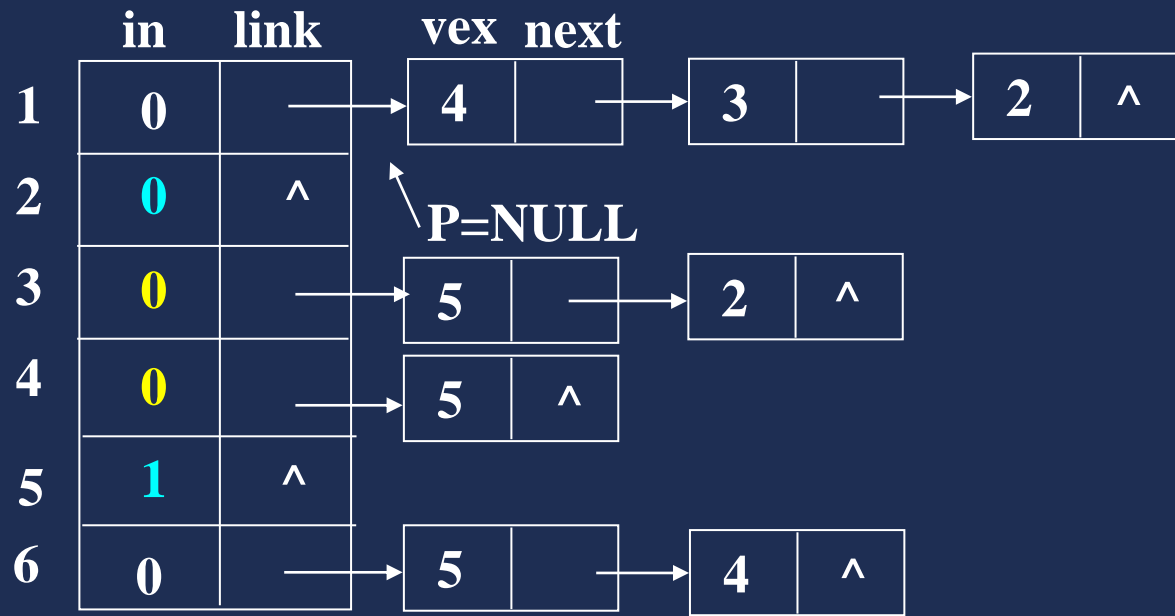
- 拓扑排序



输出序列: 6 1 3

# 7.5 有向无环图——拓扑排序

- 拓扑排序

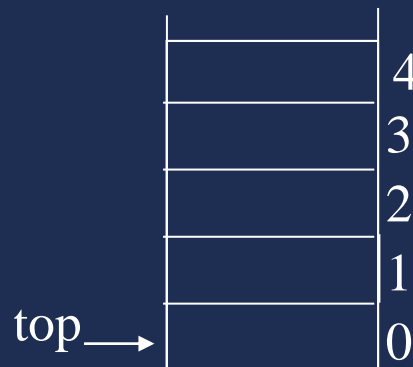
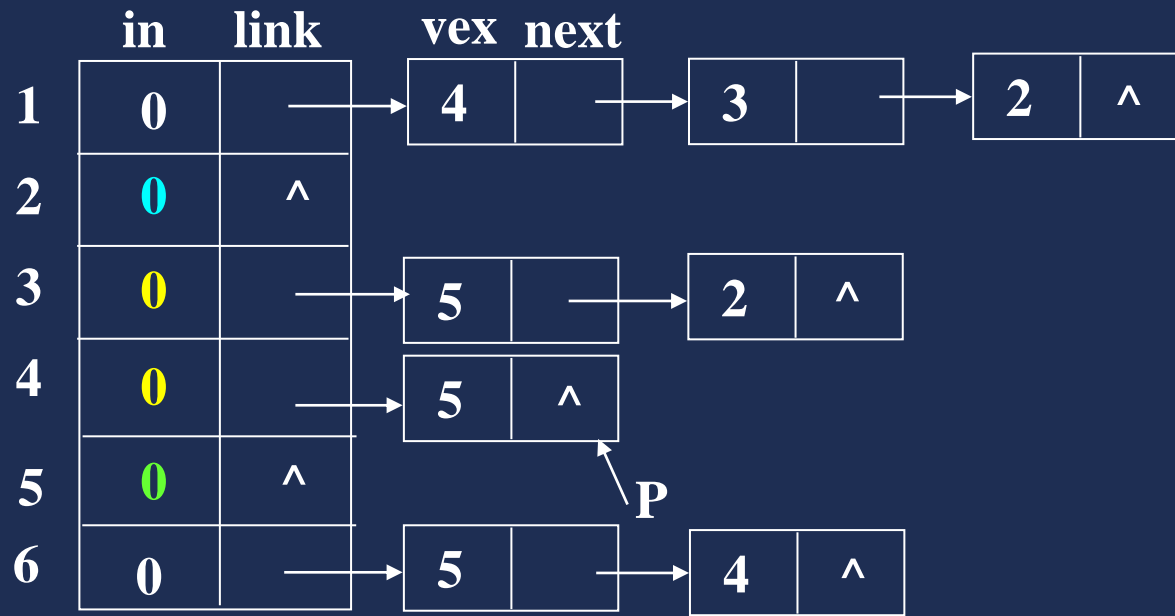


输出序列: 6 1 3 2



# 7.5 有向无环图——拓扑排序

- 拓扑排序



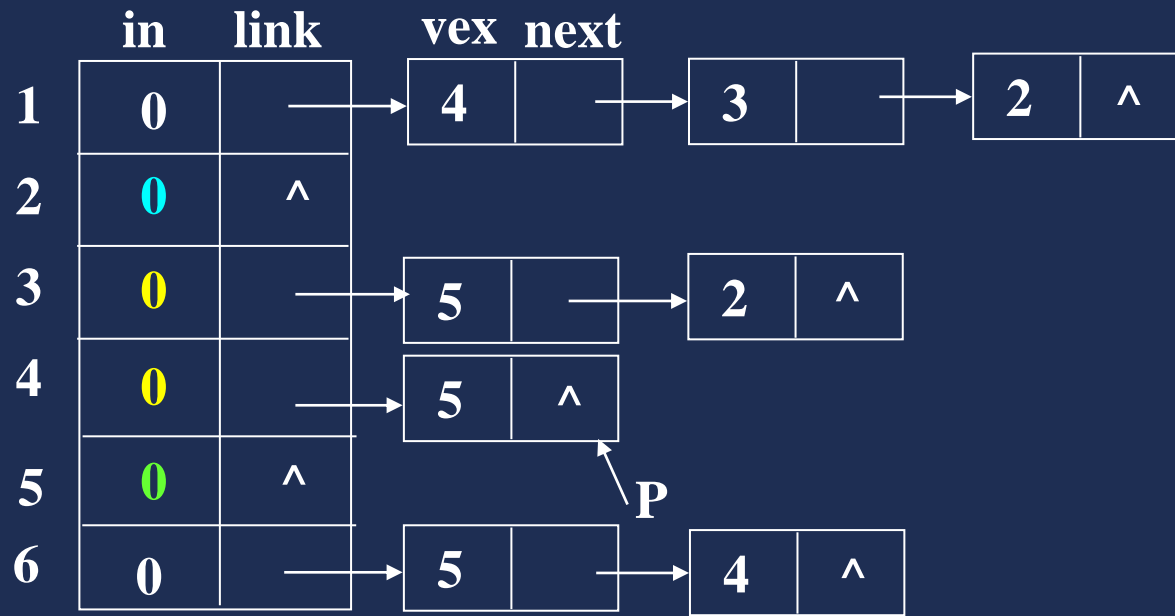
输出序列: 6 1 3 2 4





# 7.5 有向无环图——拓扑排序

- 拓扑排序

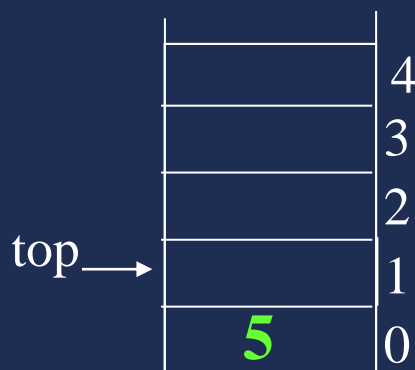
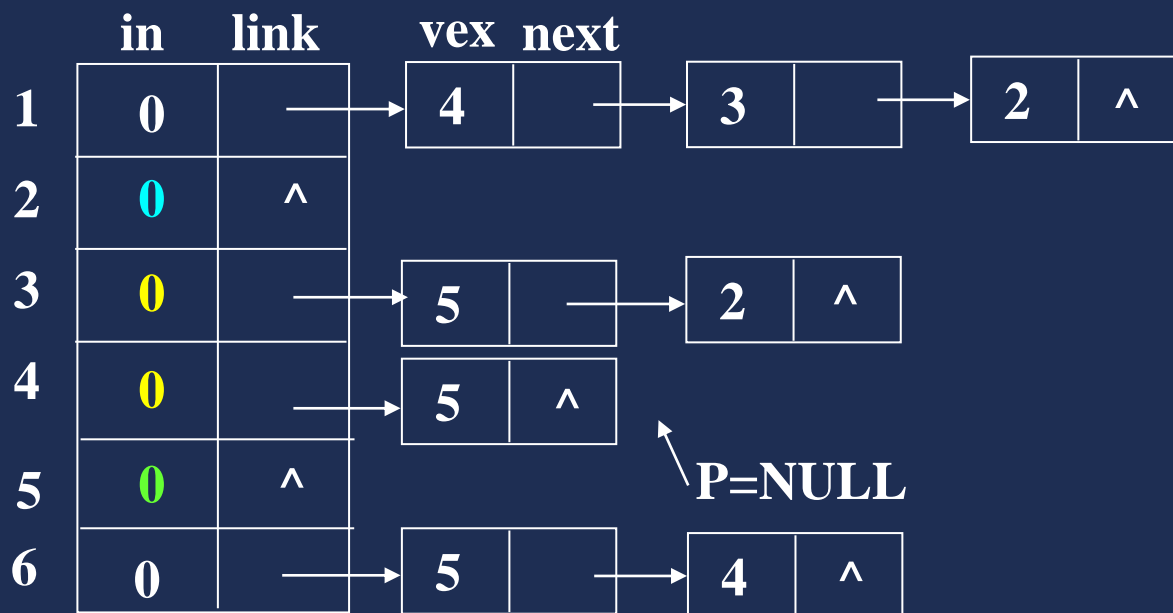


输出序列: 6 1 3 2 4



# 7.5 有向无环图——拓扑排序

- 拓扑排序

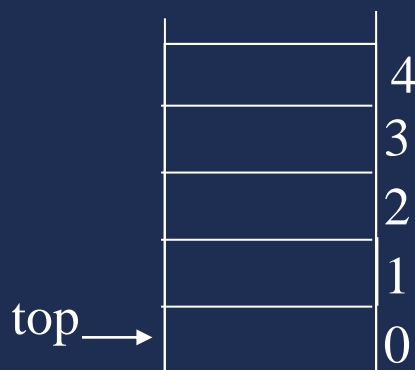
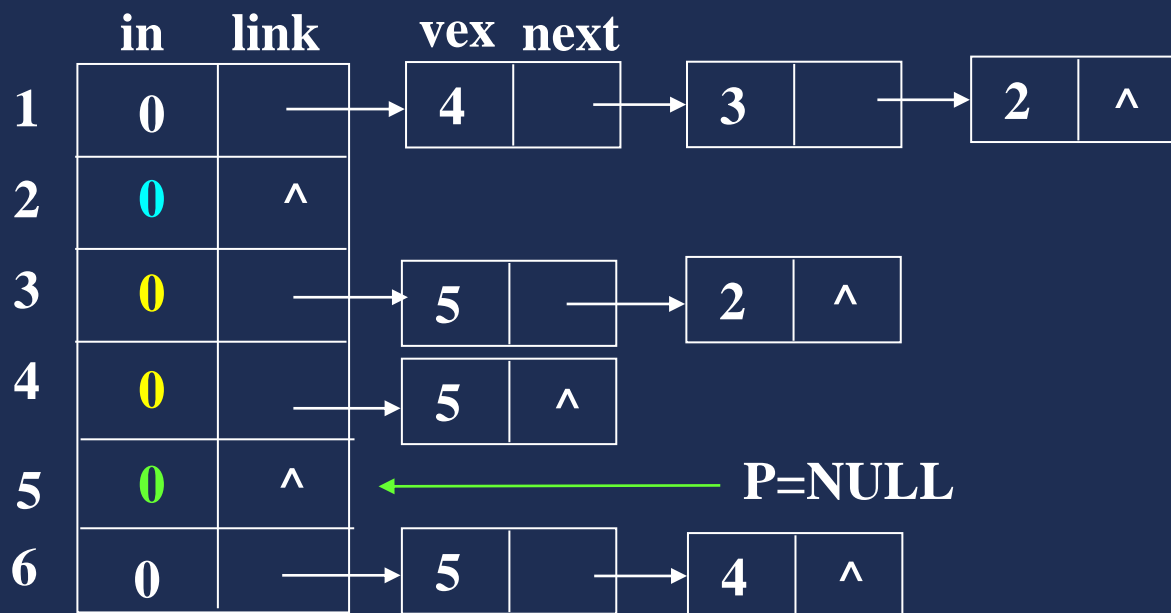


输出序列: 6 1 3 2 4



# 7.5 有向无环图——拓扑排序

- 拓扑排序



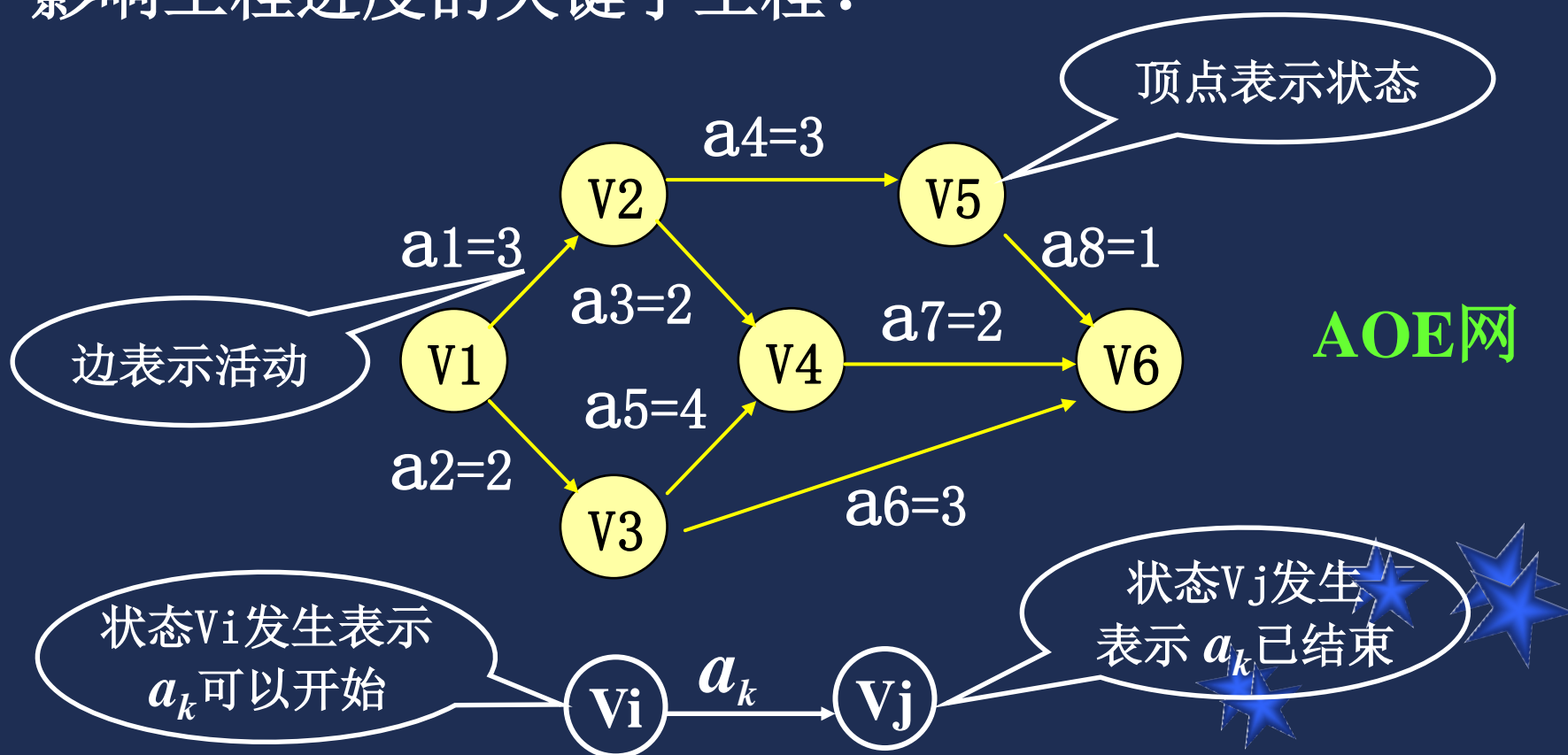
输出序列: 6 1 3 2 4 5



## 7.5 有向无环图——关键路径

- 问题提出:

- 1) 工程能否顺序进行, 即工程流程是否“合理”
- 2) 完成整项工程至少需要多少时间, 哪些子工程是影响工程进度的关键子工程?



## 7.5 有向无环图——关键路径

- AOE网

**AOE**——用边表示活动的网。它是有一个带权的有向无环图。

顶点——表示事件/状态，弧——表示活动

,

权值——活动持续的时间。

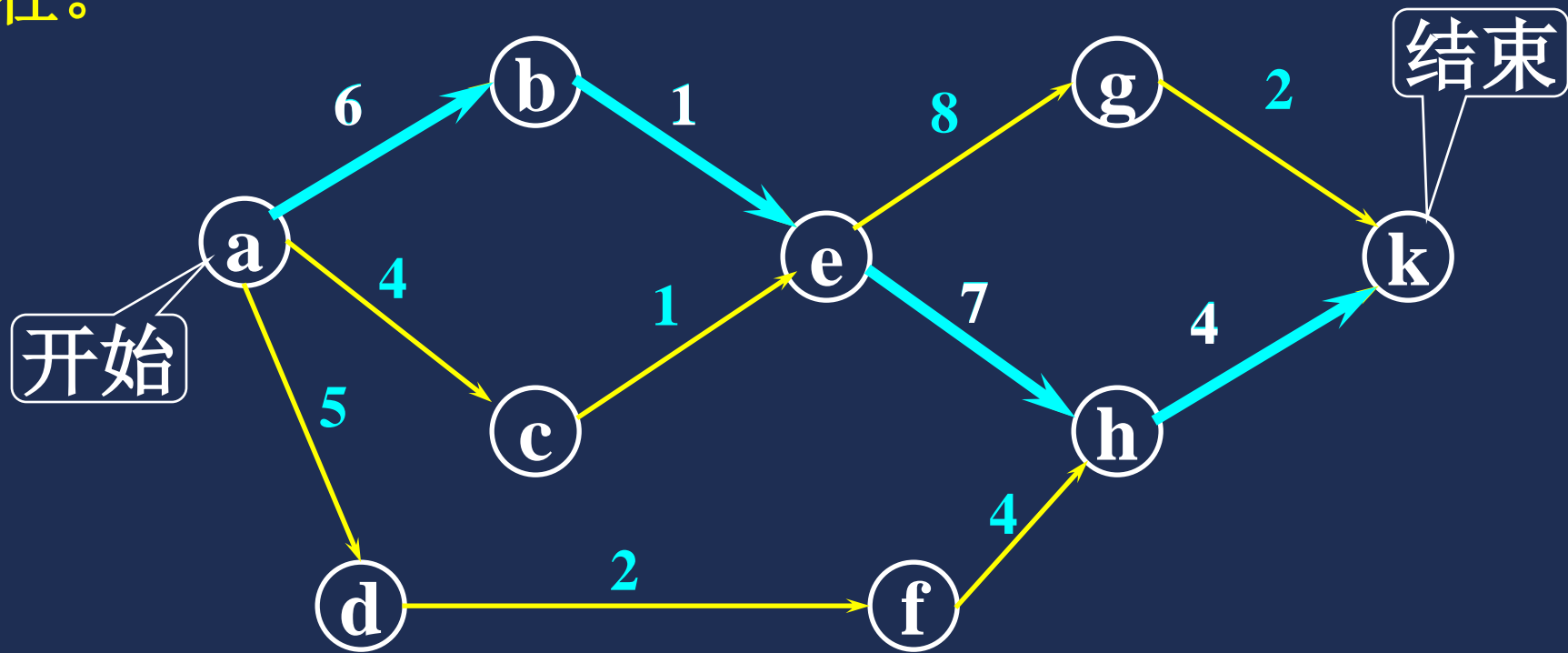
路径长度——路径上各活动持续时间之和

关键路径——路径长度最长的路径叫关键  
路径



## 7.5 有向无环图——关键路径

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。



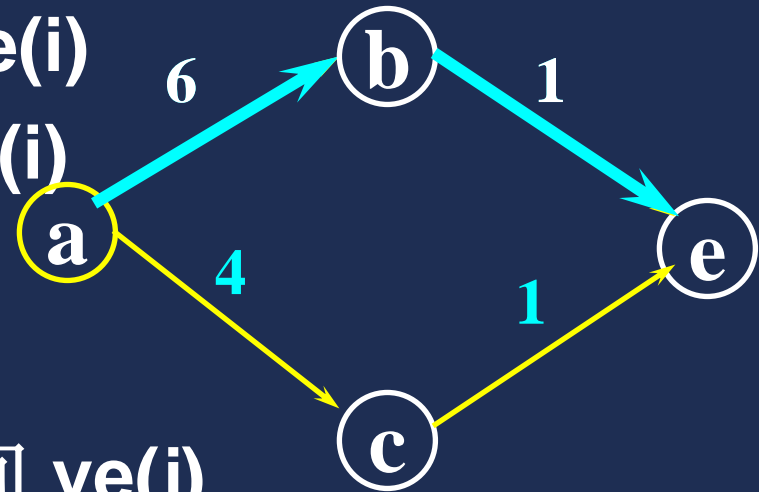
“关键活动”指的是：该弧上的权值增加将使有向图上的最长路径的长度增加。

# 如何求关键活动?

“活动(弧)”的 最早开始时间  $e(i)$

“活动(弧)”的 最迟开始时间  $l(i)$

关键活动:  $e(i) = l(i)$



“状态(顶点)” 的最早发生时间  $ve(j)$

“状态(顶点)” 的最迟发生时间  $vl(k)$

活动(弧)发生时间的计算公式

假设第  $i$  条弧为  $\langle j, k \rangle$  , 则 对第  $i$  项活动言

$e(i) = ve(j)$ ;

$l(i) = vl(k) - dut(\langle j, k \rangle)$ ;

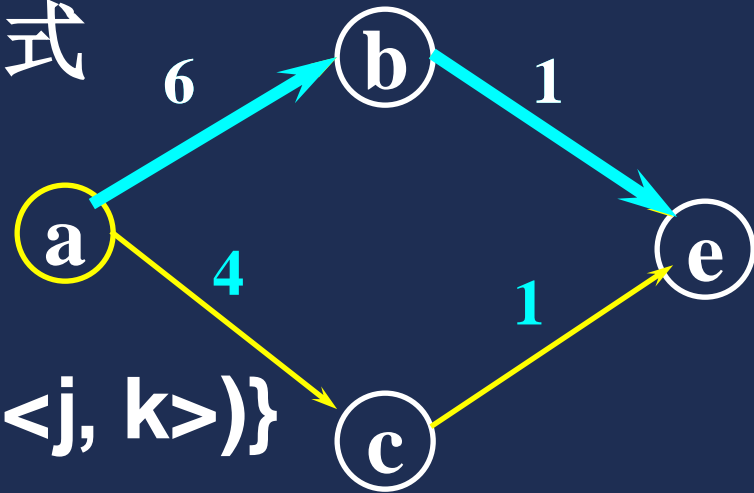


# 如何求关键活动?

状态(顶点)发生时间的计算公式  
最早开始时间:

◆  $ve(\text{源点}) = 0;$

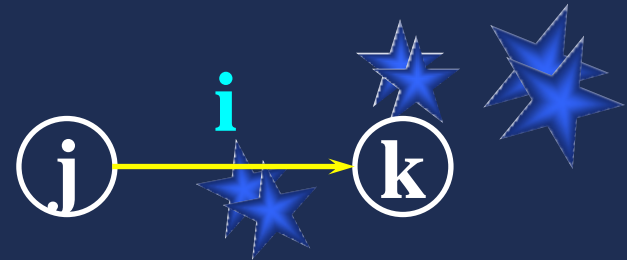
◆  $ve(k) = \text{Max}\{ve(j) + \text{dut}(<j, k>)\}$



最迟开始时间:

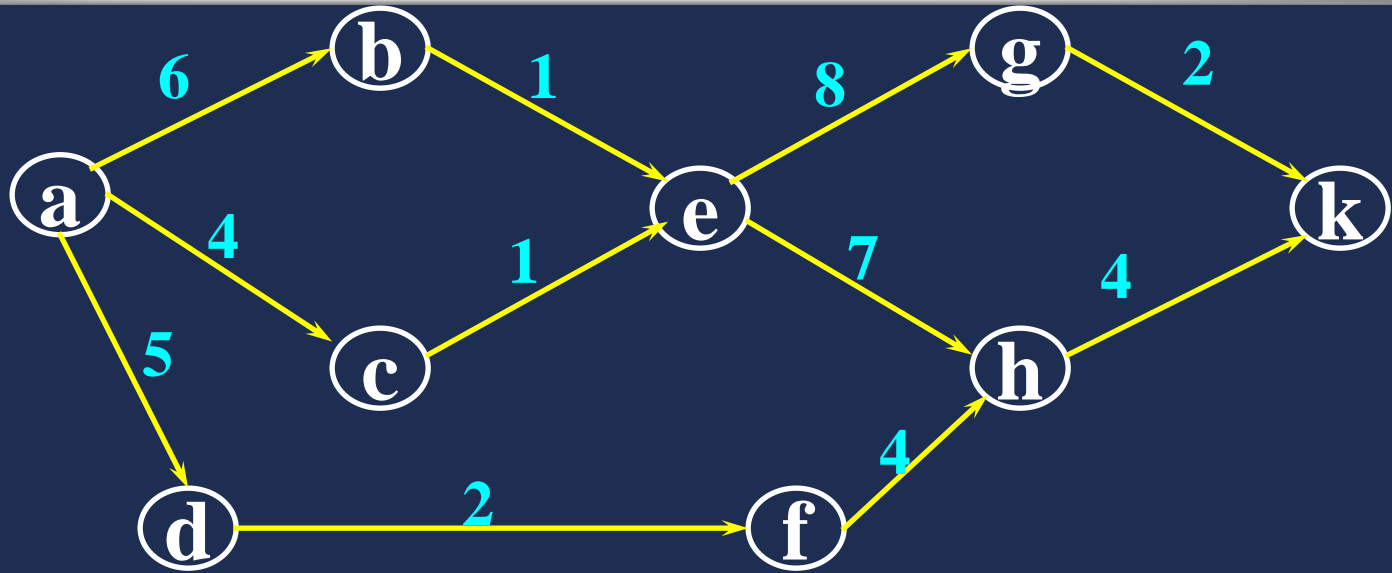
◆  $vl(\text{汇点}) = ve(\text{汇点});$

◆  $vl(j) = \text{Min}\{vl(k) - \text{dut}(<j, k>)\}$





# 如何求关键活动?



	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"/>

$$e(i) = ve(j); \quad l(i) = vl(k) - dut(<j,k>);$$

# 算法的实现要点

按**AOE网拓扑序列**的顺序，求顶点的**ve**；  
按**逆拓扑序列**的顺序，求顶点的**vl**；  
由**ve**、**vl**，计算每个活动的**ee[k]**和**el[k]**；  
找出**ee[k]==el[k]**的**关键活动**

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。

## 7.6 最短路径

- 问题提出:

用带权的有向图表示一个交通运输网，图中：

**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。

## 7.6 最短路径

- 求从某个源点到其余各顶点的最短路径——迪杰斯特拉(Dijkstra)算法

指的是对已知图  $G = (V, E)$ ，给定源顶点  $s \in V$ ，找出  $s$  到图中其它各顶点的最短路径。

- 求每一对顶点之间的最短路径——弗洛伊德(Floyd)算法

指的是对已知图  $G = (V, E)$ ，任意的顶点  $v_i, v_j \in V$ ，找出从  $v_i$  到  $v_j$  的最短路径。

## 7.6 迪杰斯特拉(Dijkstra)算法

问题：求从某个源点到其余各点的最短路径

基本思想：

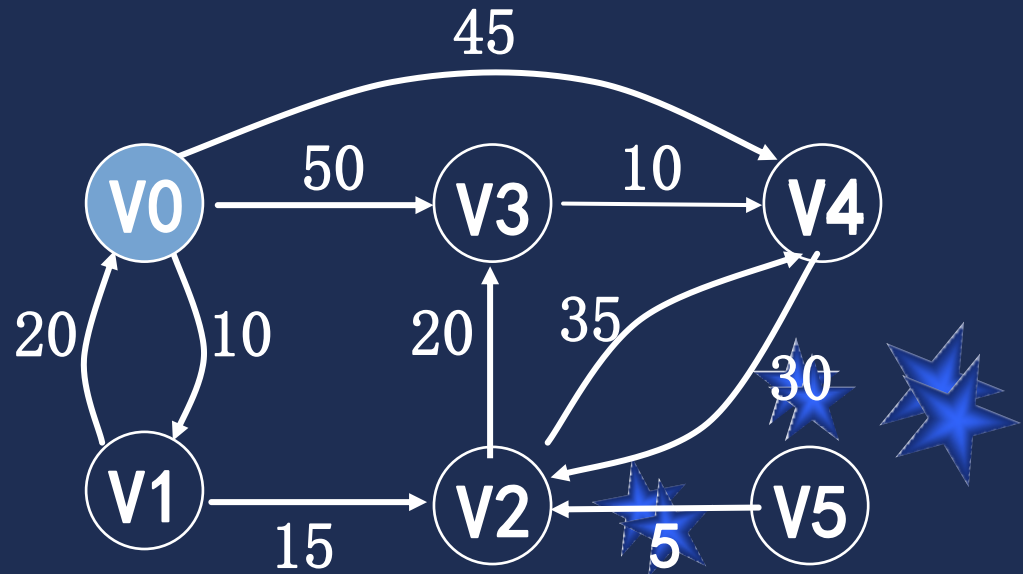
- ◆ 依最短路径的长度递增的次序求得各条路径

设置辅助数组**Dist[n-1]**

- ◆ **Dist[k]** 表示从源点**V0**到顶点**Vk**最短路径的长度

Dist[n-1]

V1	V2	V3	V4	V5
10	$\infty$	50	45	$\infty$



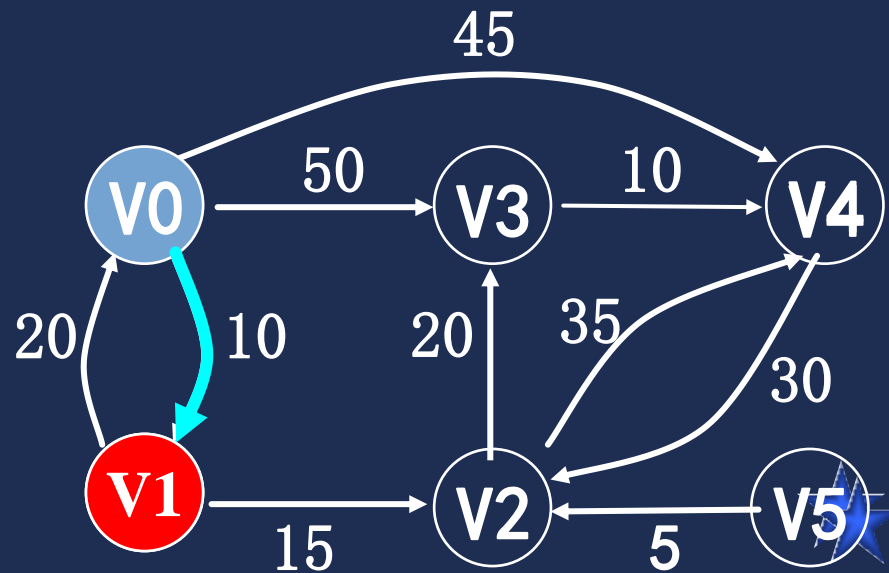
## 7.6 迪杰斯特拉(Dijkstra)算法

在 $\text{Dist}[n-1]$ 中，第一条长度最短的路径的特点：

— 必定只含一条弧，并且这条弧在始于 $V_0$ 的弧中的权值最小。

$\text{Dist}[n-1]$

V1	V2	V3	V4	V5
10	25	50	45	$\infty$



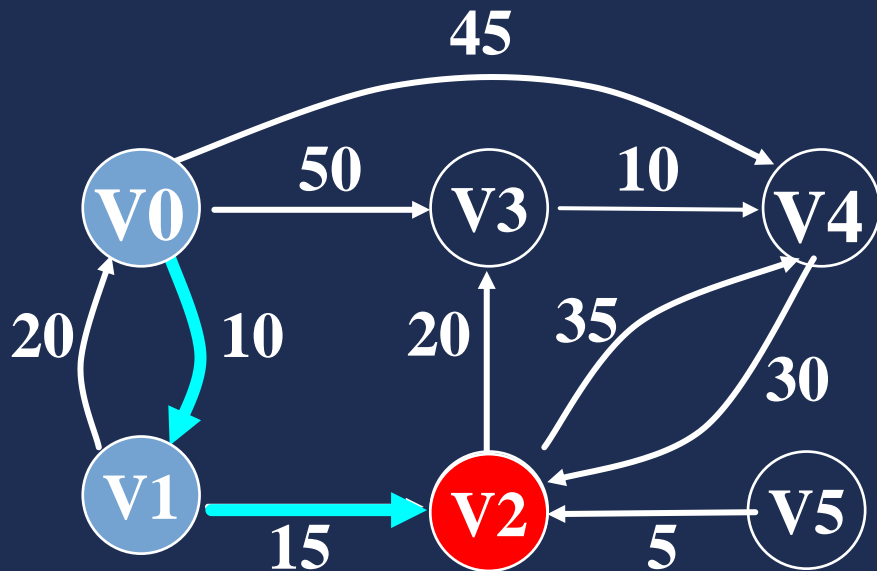
## 7.6 迪杰斯特拉(Dijkstra)算法

下一条路径长度次短的最短路径的特点:

它只可能有两种情况:

或者是从源点经过顶点 $v_1$ , 再到达该顶点 (由两条弧组成)。

或者是直接从源点到该点 (只含一条弧) ;



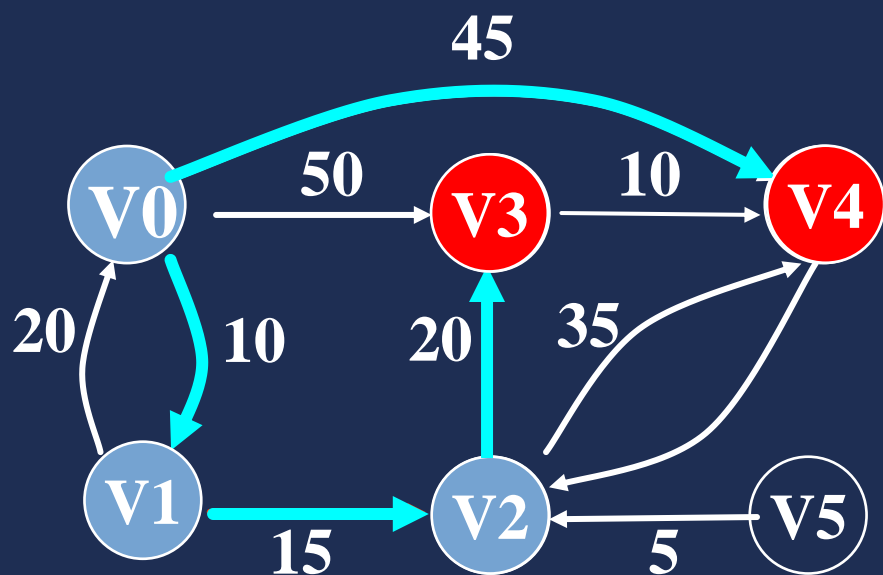
V1	V2	V3	V4	V5
10	25	45	45	$\infty$



## 7.6.1 迪杰斯特拉(Dijkstra)算法

再下一条路径长度次短的最短路径的特点：  
它可能有三种情况：

- 或者是**直接从源点到该点**（只含一条弧）；
- 或者是**从源点经过顶点v1，再到达该顶点**（由两条弧组成）；
- 或者是**从源点经过顶点v2，再到达该顶点**。



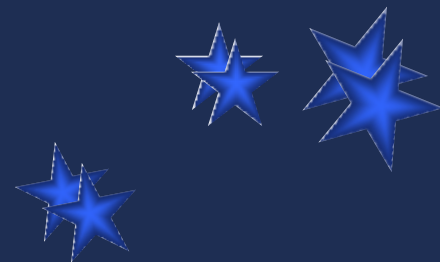
V1	V2	V3	V4	V5
10	25	45	45	$\infty$

## 7.6.1 迪杰斯特拉(Dijkstra)算法

其余最短路径的特点:

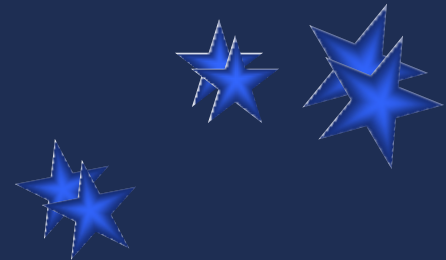
它或者是**直接从源点到该点** (只含一条弧)

或者是从源点经过已求得最短路径的顶点,  
再到达该顶点。



## 7.6 最短路径

- 迪杰斯特拉算法基本思想：按长度递增的顺序求解最短路径
- 把图中所有顶点分成两组
  - ◆ 第1组包括已求得最短路径的顶点
  - ◆ 第2组包括尚未求得最短路径的顶点；
- 每次从第2组中选择与源点距离最小的顶点，加入第1组，直至把图的所有顶点都加到进第1组。



## 7.6.1 迪杰斯特拉算法

辅助集合S:

- ◆ 当前已经得到最短路径的顶点集合
- ◆ 初始时,  $S=\{V_0\}$

辅助数组Dist

- ◆  $\text{Dist}[k]$  表示 “当前” 所求得的从源点到顶点  $k$  的最短路径
  - ◆  $\text{Dist}[k] = \langle \text{源点到顶点 } k \text{ 的弧上的权值} \rangle$  或者  
= 沿着 “当前” 最短路径到顶点  $k$  的路径长度
- 假设 “当前” 最短路径 为 源点到顶点  $j$  的路径  
则,  $\text{Dist}[k] = \text{“当前” 最短路径长度} + \langle \text{顶点 } j \text{ 到顶点 } k \text{ 的弧上的权值} \rangle$

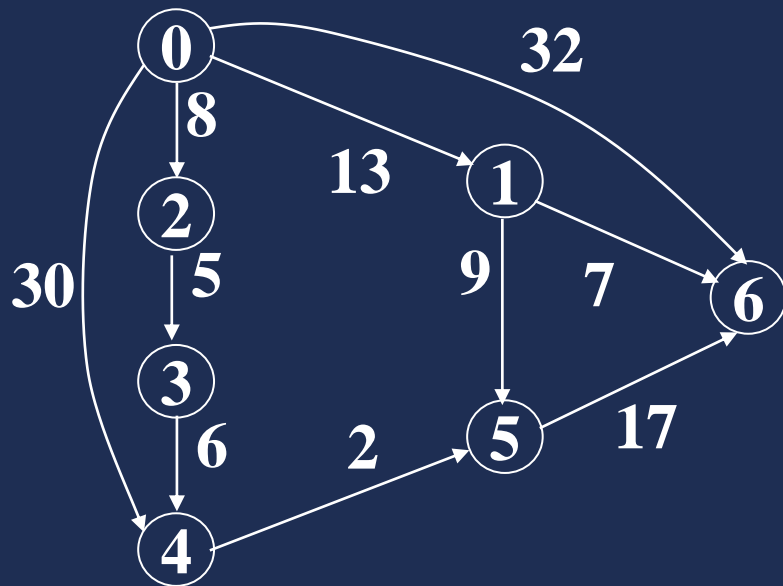
## 7.6.1 迪杰斯特拉算法

- 1) 在所有从源点出发的弧中选取一条权值最小的弧，即为第一条最短路径。
  - ◆ **V0和k之间存在弧：**  $\text{Dist}[k] = G.\text{arcs}[v0][k]$
  - ◆ **V0和k之间不存在弧：**  $\text{Dist}[k] = \text{无穷}$
- 2) 依次修改其它尚未确定最短路径的顶点  $\text{Dist}[k]$  值。
  - ◆ 假设求得最短路径的顶点为  $u$ ，则  $\text{Dist}[k] = \min(\text{Dist}[k], \text{Dist}[u] + G.\text{arcs}[u][k])$

## 7.6 最短路径

### ● 求最短路径步骤

- ◆ 初始时令  $S = \{V_0\}$ ,  $T = \{\text{其余顶点}\}$ ,  $T$ 中顶点对应的距离值
  - 若存在  $\langle V_0, V_i \rangle$ , 为  $\langle V_0, V_i \rangle$  弧上的权值
  - 若不存在  $\langle V_0, V_i \rangle$ , 为  $\infty$



$ad[ ][ ] =$

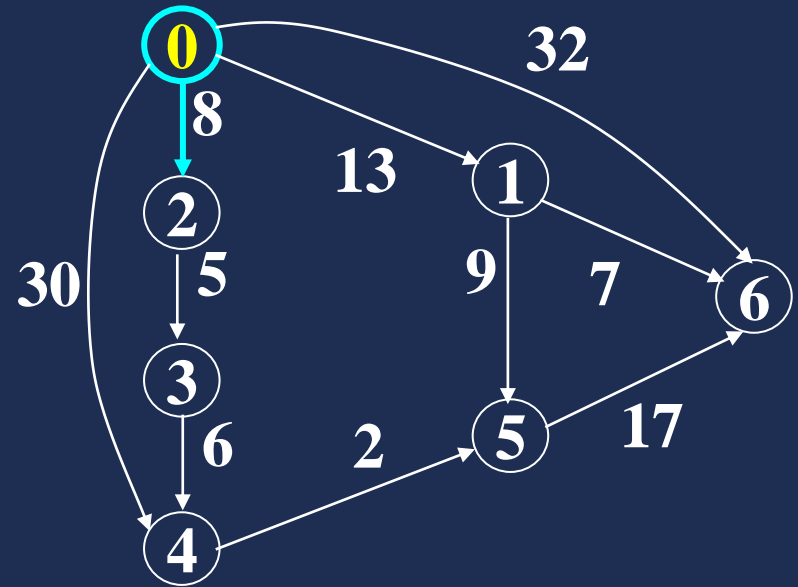
1	13	8	$\infty$	30	$\infty$	32
$\infty$	0	$\infty$	$\infty$	$\infty$	9	7
$\infty$	$\infty$	0	5	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	0	6	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	0	2	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0	17
$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

## 7.6 最短路径

### ● 求最短路径步骤

辅助数组 $\text{Dist}[n-1]$ ,  $\text{Dist}[k]$  表示源点 $V_0$ 到顶点 $V_k$ 最短路径的长度

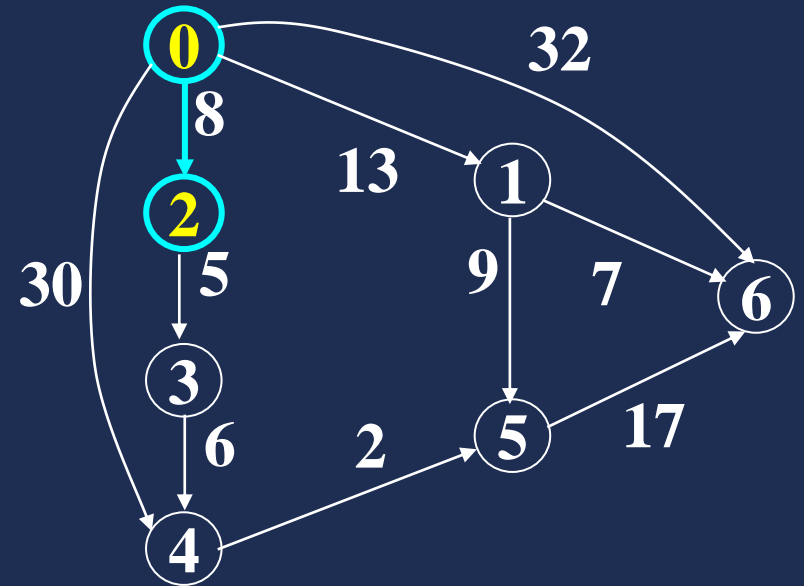
终点	从 $V_0$ 到各终点的最短路径及其长度			
$V_1$	13 < $V_0, V_1$ >			
$V_2$	8 < $V_0, V_2$ >			
$V_3$	$\infty$			
$V_4$	30 < $V_0, V_4$ >			
$V_5$	$\infty$			
$V_6$	32 < $V_0, V_6$ >			
$S$	$V_2:8$ < $V_0, V_2$ >			



# 7.6 最短路径

## ● 求最短路径步骤

终点	从V0到各终点的最短路径及其长度		
V1	13 <V0,V1>	13 <V0,V1>	
V2	8 <V0,V2>	-----	
V3	$\infty$	13 <V0,V2,V3>	
V4	30 <V0,V4>	30 <V0,V4>	
V5	$\infty$	$\infty$	
V6	32 <V0,V6>	32 <V0,V6>	
S	V2:8 <V0,V2>	V1:13 <V0,V1>	

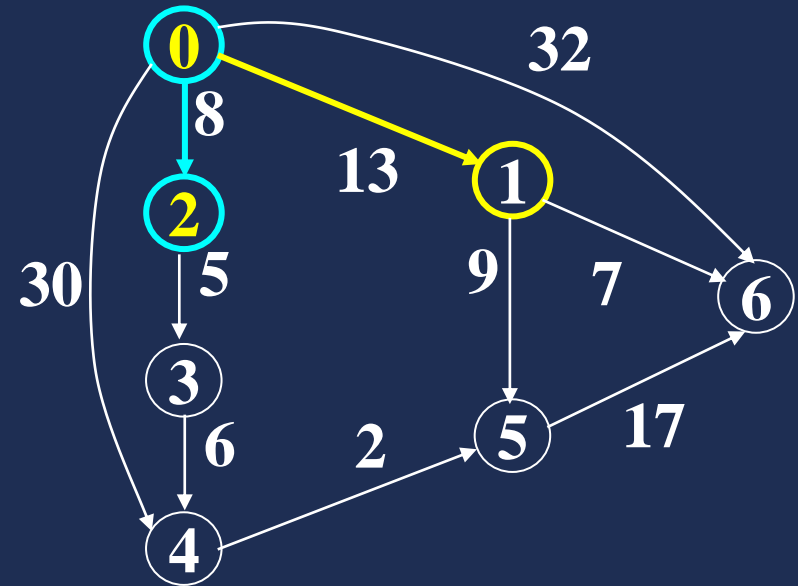




# 7.6 最短路径

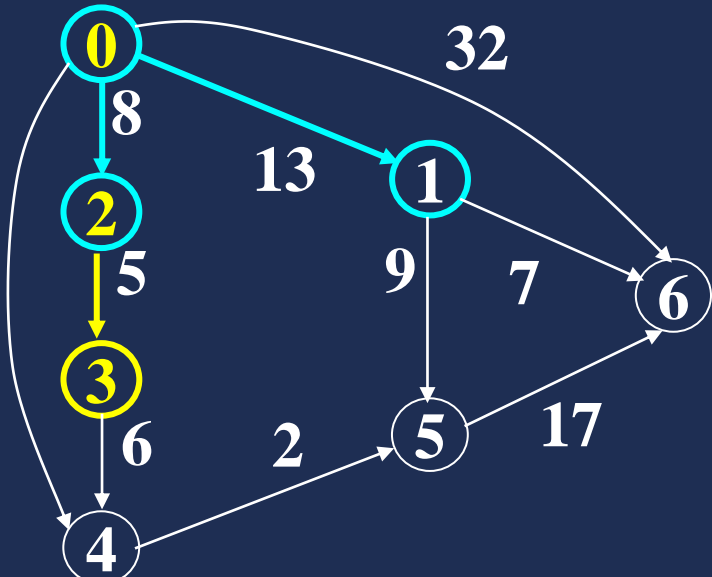
## ● 求最短路径步骤

终点	从V0到各终点的最短路径及其长度		
V1	13 <V0,V1>	13 <V0,V1>	-----
V2	8 <V0,V2>	-----	-----
V3	$\infty$	13 <V0,V2,V3>	13 <V0,V2,V3>
V4	30 <V0,V4>	30 <V0,V4>	30 <V0,V4>
V5	$\infty$	$\infty$	22 <V0,V1,V5>
V6	32 <V0,V6>	32 <V0,V6>	20 <V0,V1,V6>
Vj	V2:8 <V0,V2>	V1:13 <V0,V1>	V3:13 <V0,V2,V3>



# 7.6 最短路径

## ● 求最短路径步骤

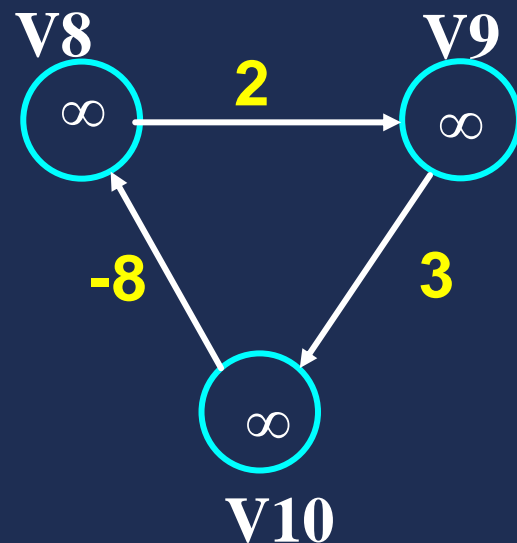
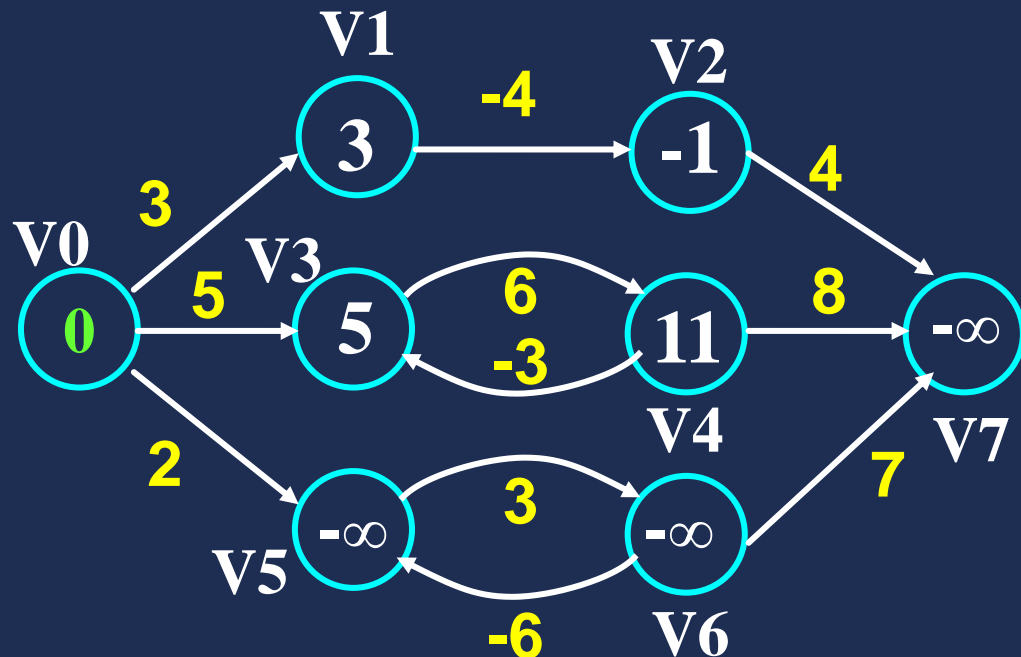
终点	从V0到各终点的最短路径及其长度			
V1	13	13		
				
				19 <V0,V2,V3,V4>
				22 <V0,V1,V5>
				20 <V0,V1,V6>
Vj	V2:8 <V0,V2>	V1:13 <V0,V1>	V3:13 <V0,V2,V3>	V4:19 <V0,V2,V3,V4>

# 7.6 最短路径

## ● 求最短路径步骤

终点	从V0到各终点的最短路径及其长度				
V1	13	13			
Vj	V2:8	V1:13	V3:13	V4:19	V6:20
	<V0,V2>	<V0,V1>	<V0,V2,V3>	<V0,V2,V3,V4>	<V0,V1,V6>

# 有负权重边的情况

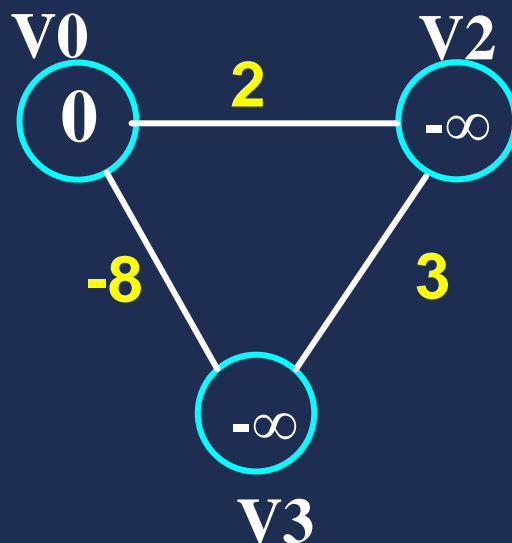


- 最短路径上可以包括环吗？
  - 负值回路（路径长度 $<0$ ）——最短路径无定义
  - 包含正值回路的路径不可能是最短路径
  - 可以把零值回路直接删除，得到简单路径

# 有负权重边的情况

## ● 无向图

- ◆ 如果有权重 $<0$ ，最短路径无定义

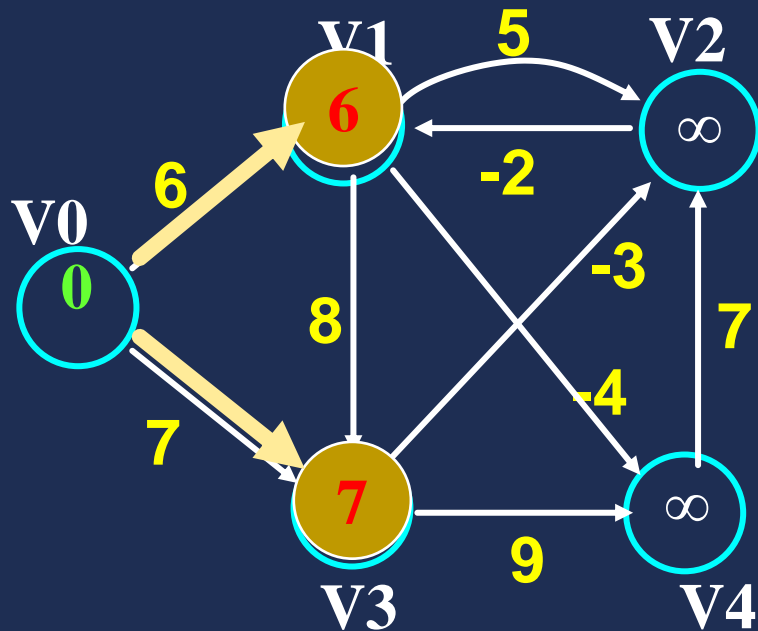


## ● 有向图

- ◆ 负值回路（路径长度 $<0$ ）——最短路径无定义
- ◆ 最短路径一定是简单路径
- ◆ 广度优先搜索

# Bellman-Ford算法

- 单源最短路径问题，顶点V0到其它顶点间最短路径
- 基本思想：逐条边试探法

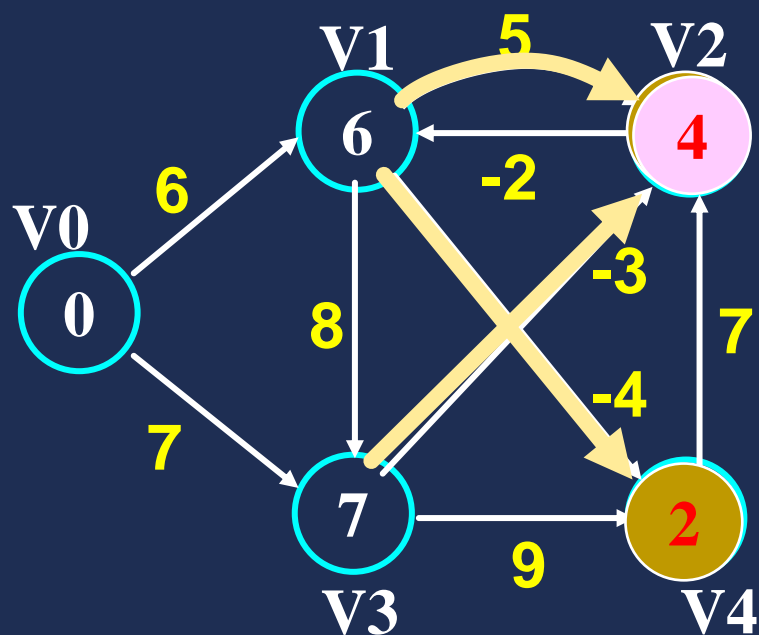


V0	V1	V2	V3	V4
0	∞	∞	∞	∞
0	6	∞	7	∞

依次考察边V1V2, V1V3, V1V4, V2V1, V3V2, V3V4,  
V4V2, **V0V1, V0V3**

# Bellman-Ford算法

- 单源最短路径问题，顶点V0到其它顶点间最短路径
- 基本思想：逐条边试探法



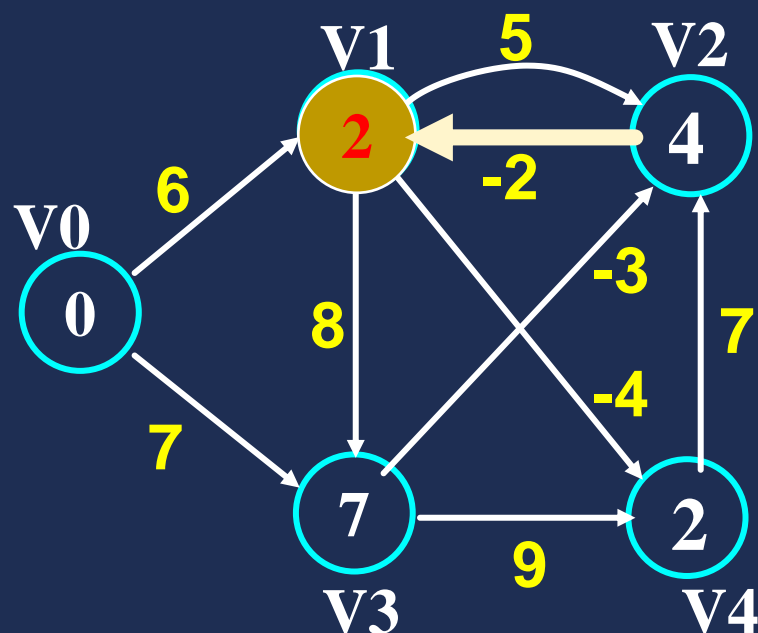
V0	V1	V2	V3	V4
0	$\infty$	$\infty$	$\infty$	$\infty$
0	6	$\infty$	7	$\infty$
0	6	4	7	2

第2次考察边 V1V2, V1V3, V1V4, V2V1, V3V2, V3V4,  
V4V2, V0V1, V0V3



# Bellman-Ford算法

- 单源最短路径问题，顶点**V0**到其它顶点间最短路径
- 基本思想：逐条边试探法



V0	V1	V2	V3	V4
0	$\infty$	$\infty$	$\infty$	$\infty$
0	6	$\infty$	7	$\infty$
0	6	4	7	2
0	6	2	7	2

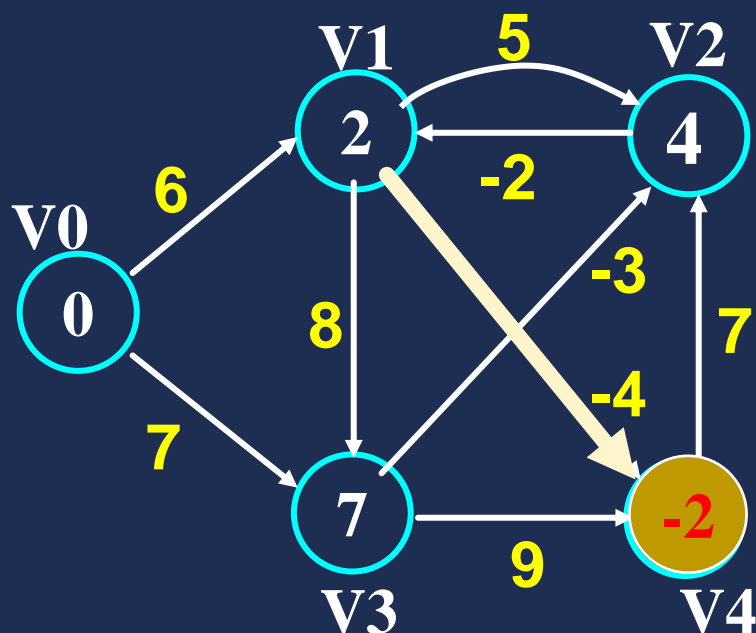
第3次考察边V1V2, V1V3, V1V4, **V2V1**, V3V2, V3V4,  
V4V2, V0V1, V0V3





# Bellman-Ford算法

- 单源最短路径问题，顶点V0到其它顶点间最短路径
- 基本思想：逐条边试探法 处理过程类似动态规划

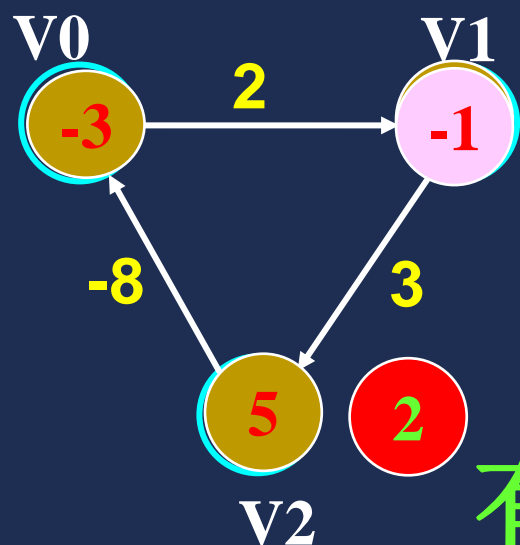


V0	V1	V2	V3	V4
0	$\infty$	$\infty$	$\infty$	$\infty$
0	6	$\infty$	7	$\infty$
0	6	4	7	2
0	6	2	7	2
0	6	2	7	-2

第4次考察边V1V2, V1V3, **V1V4**, V2V1, V3V2, V3V4,  
V4V2, V0V1, V0V3

# Bellman-Ford算法

## ● 若图中有负环



V0, V1, V2
0, $\infty$ , $\infty$
0, 2, $\infty$
-3, -1, 5

有负环！

依次考察边V1V2, V2V0, V0V1

再次考察边V1V2, V2V0, V0V1

判断是否有负环– 依次考察边V1V2, V2V0, V0V1



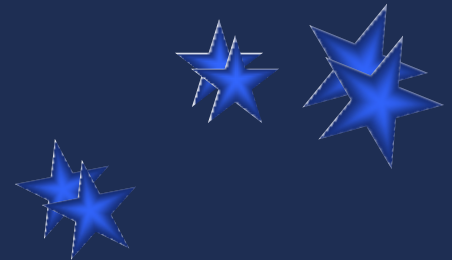
# Bellman-Ford算法

- 算法过程:
- 1. 初始化所有节点到源点距离为 $\infty$ ;
- 2. **for**( $i=1$ ;  $i < n$ ;  $i++$ )
- 对每一条边( $u, v$ )依次进行下列判断（松弛操作）
  - ◆ 如果  $\text{Dist}[v] > \text{Dist}[u] + w(u, v)$
  - ◆ 则  $\text{Dist}[v] = \text{Dist}[u] + w(u, v)$  ;
- 3. 判断图中是否有负环：对每一条边( $u, v$ )依次进行下列判断：
  - ◆ 如果  $\text{Dist}[v] > \text{Dist}[u] + w(u, v)$
  - ◆ 则有负环，**return false**;

时间复杂度?

邻接表

$O(n \times E)$

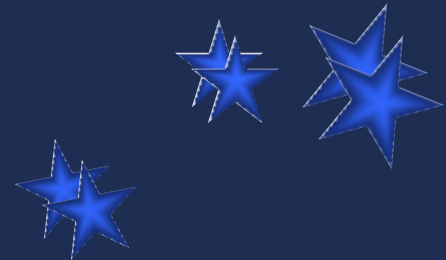


## 7.6 最短路径

- 求每一对顶点之间的最短路径
- 方法1：每次以一个顶点为源点，重复执行Dijkstra算法n次—— $T(n)=O(n^3)$

- 方法2：弗洛伊德(Floyd)算法

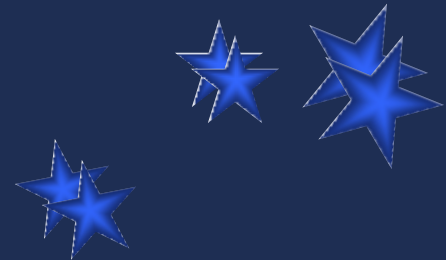
从 $v_i$ 到 $v_j$ 的所有可能存在的路径中，选出一条长度最短的路径。



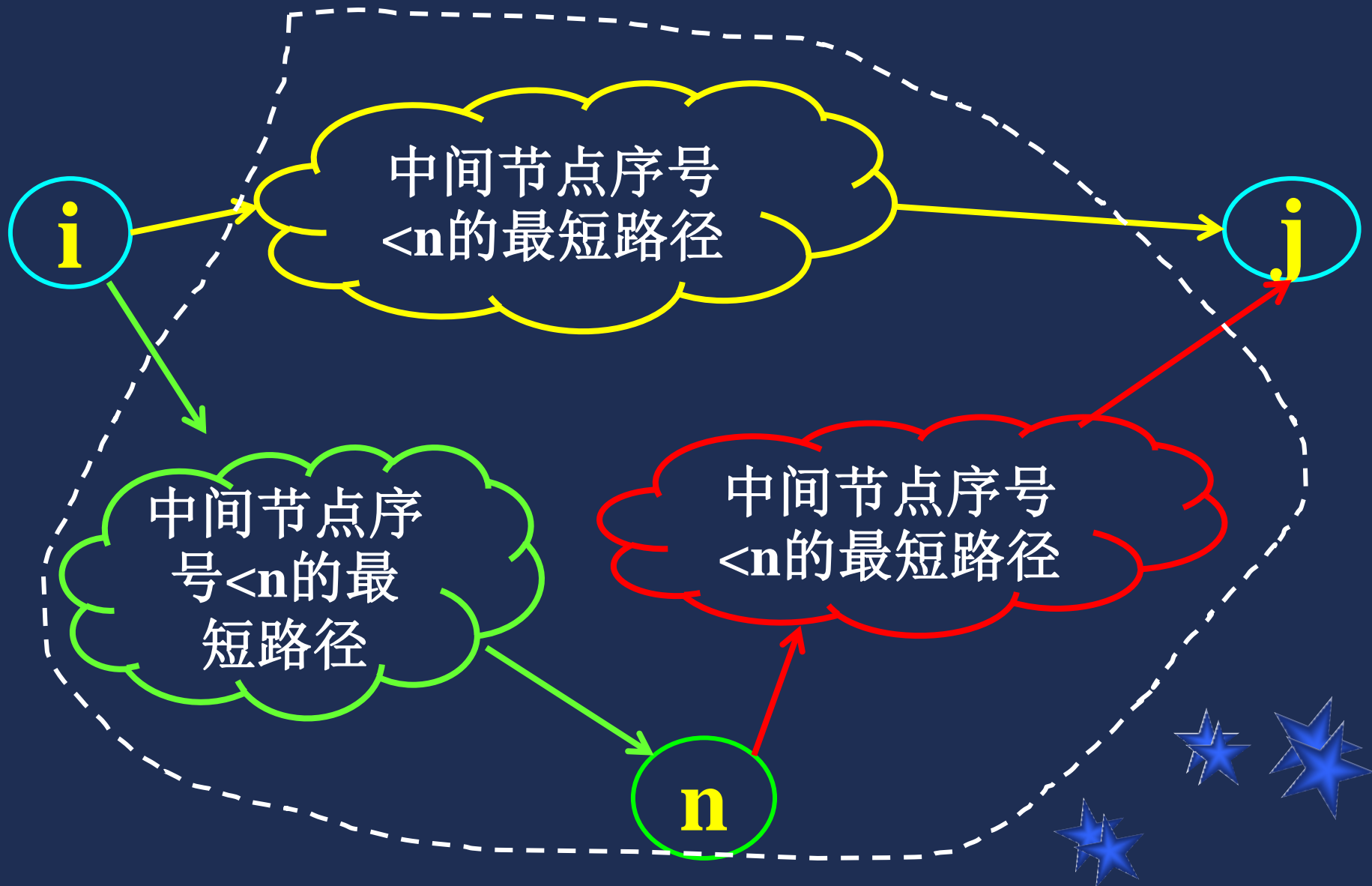
## 7.6.2 弗洛伊德算法

求每一对顶点之间的最短路径

- 即要找出从  $v_i$  到  $v_j$  的最短路径，从  $v_i$  到  $v_j$  所有可能的路径中，选出一条长度最短的路径
  - ◆ 若弧  $\langle v_i, v_j \rangle$  存在，则存在路径  $\{v_i, v_j\}$
  - ◆ 若弧  $\langle v_i, v_1 \rangle, \langle v_1, v_j \rangle$  存在，则存在路径  $\{v_i, v_1, v_j\}$
  - ◆ 若  $\{v_i, \dots, v_2\}, \{v_2, \dots, v_j\}$  存在，则存在一条路径  $\{v_i, \dots, v_2, \dots, v_j\}$
  - ◆ 依次类推，则  $v_i$  至  $v_j$  的最短路径应是上述这些路径中，路径长度最小者



## 7.6 最短路径




## 7.6 最短路径

对于任意顶点 $v_i, v_j$ ,

假设 $D_{i,j,k}$ 表示所经节点序号不超过 $k$ 的一条路径长度,  $\min\{D_{i,j,k}\}$ 表示从 $v_i$ 到 $v_j$ 所经节点序号不超过 $k$ 的最短路径长度,

$v_i, v_j$ 之间的最短路径为 $\min\{D_{i,j,n}\}$

假设对于任意顶点 $v_i, v_j$ 已经求出所经节点序号不超过 $k-1$ 的最短路径长度, 则有:

$$\min\{D_{i,j,k}\} = \min\left\{\min\{D_{i,j,k-1}\}, \min\{D_{i,k,k-1}\} + \min\{D_{k,j,k-1}\}\right\}$$


## 7.6 最短路径

### 算法思想:

按照顶点序号逐个试探，假设为任意2个顶点已计算出中间节点最大序号为 $K-1$ 的最短路径，在此基础上进一步计算出任意2个顶点中间节点最大序号为 $K$ 的最短路径。

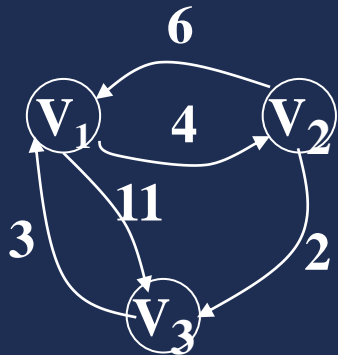
### ◆求最短路径步骤

- 初始时设置一个  $n$  阶方阵，对角线元素为 0，若存在弧  $\langle v_i, v_j \rangle$ ，对应元素为权值；否则为 $\infty$ 。
- 逐步试着在原直接路径中增加中间顶点，若加入中间点后路径变短，则修改之；否则，维持原值。
- 所有顶点试探完毕，算法结束。



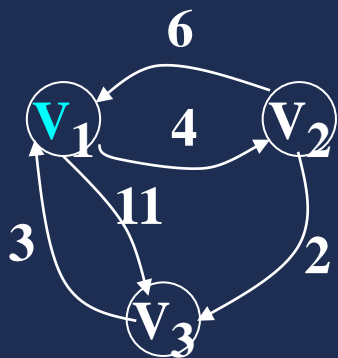
# 7.6 最短路径

例



初始:  $\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$  路径:

	$V_1V_2$	$V_1V_3$
$V_2V_1$		$V_2V_3$
$V_3V_1$		



加入  $V_1$  点

考察:  $\langle v_2, v_3 \rangle = 2$

$\langle v_2, v_1 \rangle \langle v_1, v_3 \rangle = 17$

$\langle v_3, v_2 \rangle = \infty$

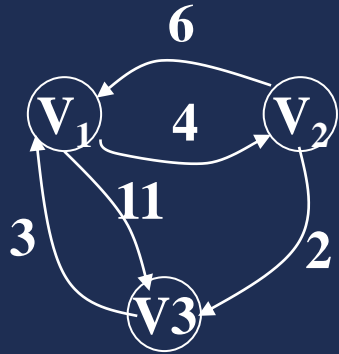
$\langle v_3, v_1 \rangle \langle v_1, v_2 \rangle = 7$

加入  $V_1$ :  $\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$  路径:

	$V_1V_2$	$V_1V_3$
$V_2V_1$		$V_2V_3$
$V_3V_1$	$V_3V_1V_2$	

# 7.6 最短路径

例



加入  $V_1$ :  $\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$  路径:

	$V_1V_2$	$V_1V_3$
$V_2V_1$		$V_2V_3$
$V_3V_1$	$V_3V_1V_2$	

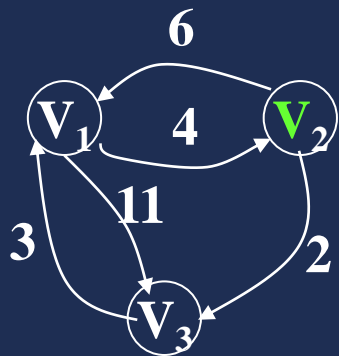
加入  $V_2$  点

考察:  $\langle v1, v3 \rangle = 11$

$\langle v1, v2 \rangle \langle v2, v3 \rangle = 6$

$\langle v3, v1 \rangle = 3$

$\langle v3, v2 \rangle \langle v2, v1 \rangle = 13$

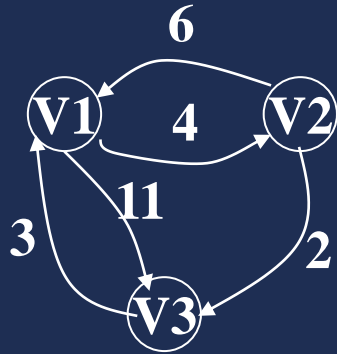


加入  $V_2$ :  $\begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$  路径:

	$V_1V_2$	$V_1V_2V_3$
$V_2V_1$		$V_2V_3$
$V_3V_1$	$V_3V_1V_2$	

# 7.6 最短路径

例



加入V2:  $\begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$  路径:

	$V_1V_2$	$V_1V_2V_3$
$V_2V_1$		$V_2V_3$
$V_3V_1$	$V_3V_1V_2$	

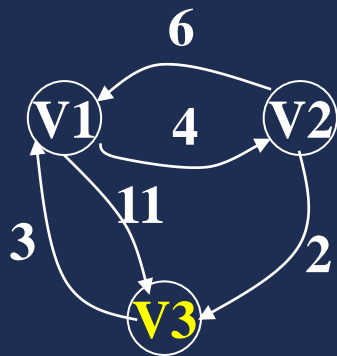
加入V3点

考察:  $\langle v1, v2 \rangle = 4$

$\langle v1, v3 \rangle \langle v3, v2 \rangle = 13$

$\langle v2, v1 \rangle = 6$

$\langle v2, v3 \rangle \langle v3, v1 \rangle = 5$



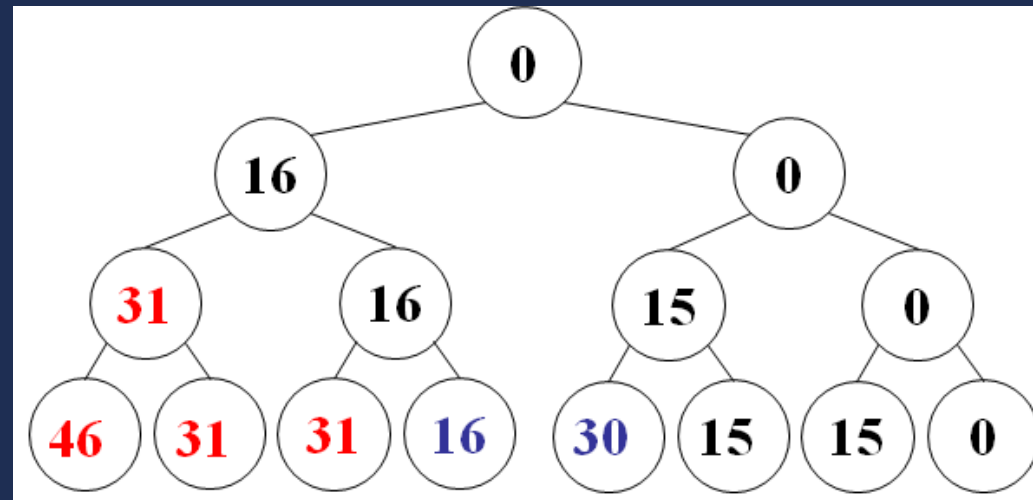
加入V3:  $\begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$  路径:

	$V_1V_2$	$V_1V_2V_3$
$V_2V_3V_1$		$V_2V_3$
$V_3V_1$	$V_3V_1V_2$	

## ● 面向过程决策的最优化求解

◆ 多步骤

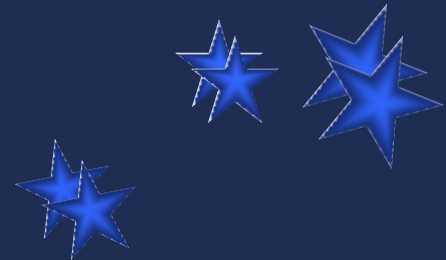
◆ 多选项



## ●贪心算法基本要素

最优子结构——子问题的最优解包含在整体最优解中

◆贪心选择性——整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。



# 最小生成树算法正确性



## 1) 证明具有贪心选择性质

即证明贪心选择的边属于某棵最小生成树

- ◆ 设  $G=(V, E)$  是一个连通无向图，边上定义了实数权重函数
- ◆ 设  $A$  是  $E$  的一个子集，在某棵最小生成树  $T$  中。
- ◆ 设  $(S, V-S)$  是  $G$  的一个切割，且  $A$  的边要么在  $S$  的子图中，要么在  $V-S$  的子图中。
- ◆ 设  $(u, v)$  是横跨切割  $(S, V-S)$  的一条权重最小的边。

● 那么  $(u, v)$  包含在  $G$  的某棵最小生成树中

# 最小生成树算法正确性

证明:

设  $T$  是包含  $A$  的最小生成树, 若  $T$  包含边  $(u, v)$ , 则得证

若  $T$  不包含边  $(u, v)$ , 则  $T$  中必定包含一条边连通  $S$  和  $V-S$ , 设为  $(x, y)$ .

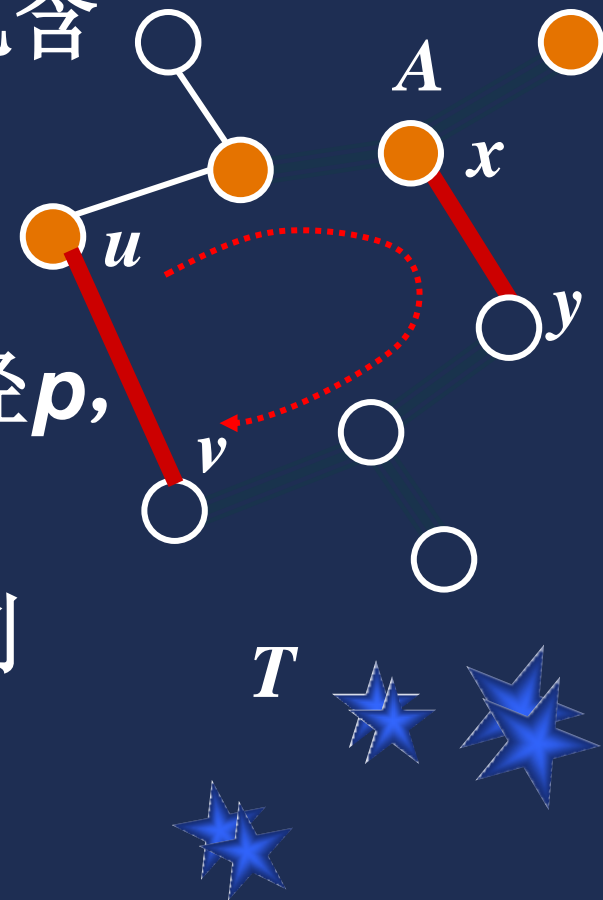
显然  $w(u, v) \leq w(x, y)$

则  $T$  中必定存在一条从  $u$  到  $v$  的路径  $p$ ,  $p$  包含边  $(x, y)$ 。

令树  $T' = T - \{(x, y)\} \cup \{(u, v)\}$ , 则

$$w(T') \leq w(T)$$

所以  $T'$  也是  $G$  的最小生成树



## 4 最小生成树

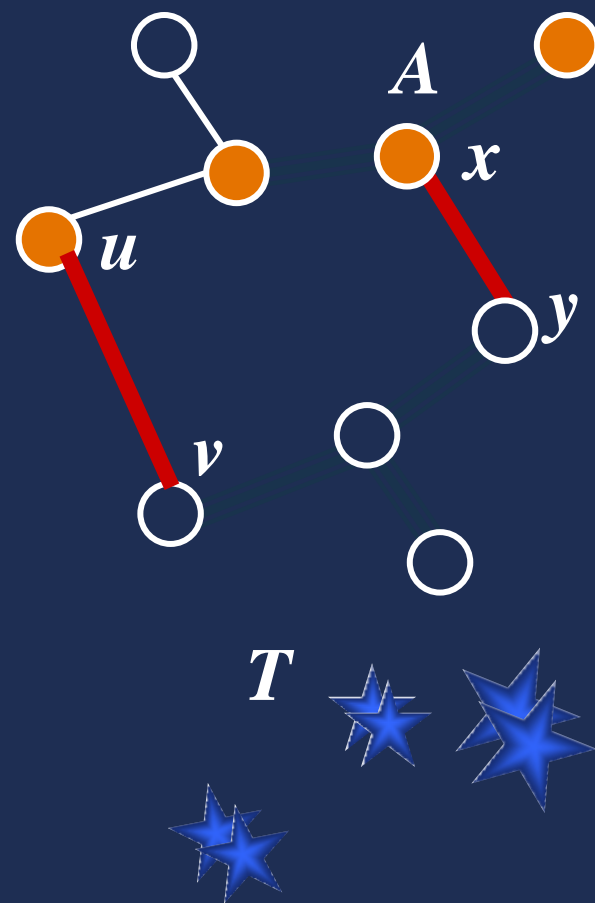
- 2) 证明具有最优子结构性质
- 即证明加入新选择的边形成的部分是最小生成树的一部分。

- 证明:

- ◆  $(u, v) \in A, A \subseteq T$

- ◆  $A \cup \{(u, v)\} \subseteq T$

- ◆  $A \cup \{(u, v)\} \subseteq T'$





# 思考题

- 1、判断一个有向图是否有环（回路）的方法是

A) 求结点的度

B) 拓扑排序

C) 求关键路径

D) 求最短路径

- 答案：B

- 2、在有向图的邻接表存储结构中，顶点v在出边表中出现的次数是

A) 顶点的v的度

B) 顶点v的出度

C) 顶点v的入度

D) 依附于顶点v的边数

- 答案：C

- 3、用邻接矩阵表示图时，若图中有100个顶点，100条边，则形成的矩阵有多少元素？有多少非零元素？

- 答案：邻接矩阵中的元素有 $100^2 = 10000$ 个。  
它有100个非零元素（对于有向图）或200个非零元素（对于无向图）。

# 思考题

一个 $n$ 个顶点的连通无向图，其边的个数至少为（  $n-1$  ）。

要连通具有 $n$ 个顶点的有向图，至少需要（  $n$  ）条边

在一个无向图中，所有顶点的度数之和等于所有边数（  $2$  ）倍，在一个有向图中，所有顶点的入度之和等于所有顶点出度之和的（  $1$  ）倍。



1. 下列说法不正确的是（ C ）。

A. 图的遍历是从给定的源点出发每一个顶点仅被访问一次

B. 遍历基本算法有两种：深度遍历和广度遍历

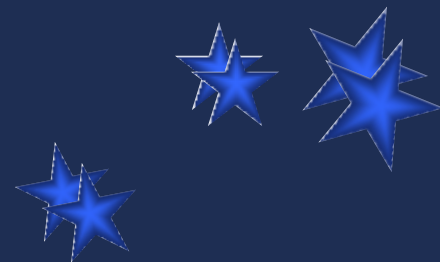
C. 图的深度遍历不适用于有向图

D. 图的深度遍历是一个递归过程

2、无向图 $G=(V,E)$ ,其中:  $V=\{a,b,c,d,e,f\}$ ,  
 $E=\{(a,b),(a,e),(a,c),(b,e),(c,f),(f,d),(e,d)\}$ , 对该图进行深度优先遍历, 得到的顶点序列正确的是（ C ）。

A. a,b,e,c,d,f    B. a,c,f,e,b,d

C. a,e,b,d,f,c    D. a,e,d,b,f,c



1. 下面哪一方法可以判断出一个有向图是否有环:

- A. 深度优先遍历    B. 拓扑排序  
C. 求最短路径      D. 求关键路径

B

2、在用邻接表表示图时，拓扑排序算法时间复杂度为:

- A.  $O(n)$       B.  $O(n+e)$   
C.  $O(n*n)$     D.  $O(n*n*n)$

B

3、当各边上的权值( )时，BFS（广度优先遍历）算法可用来解决单源最短路径问题。

- A. 均相等    B. 均互不相等  
C. 不一定相等

A



- 判断题:

- F 1. 有 $e$ 条边的无向图, 在邻接表中有 $e$ 个结点。
- T 2. 强连通图的各项点间均可达
- F 3. 邻接多重表是无向图和有向图的链式存储结构。
- T 4. 无向图的邻接矩阵可用一维数组存储。
- F 5. 需要借助于一个队列来实现DFS算法
- T 6. 无环有向图才能进行拓扑排序
- T 7. 在图 $G$ 的最小生成树 $G_1$ 中, 可能会有某条边的权值超过未选边的权值。
- F 8. 不同求最小生成树的方法得到的生成树是相同的。
- F 9. 当改变网上某一关键路径上任一关键活动后, 必将产生不同的关键路径
- F 10. 网络的最小生成树是唯一的

