



第7章 图

7.1 图的定义及操作

7.2 图的存储结构

7.3 图的遍历

7.4 最小生成树

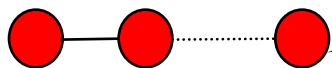
7.5 最短路径问题

7.6 有向无环图的应用

7.7 小结

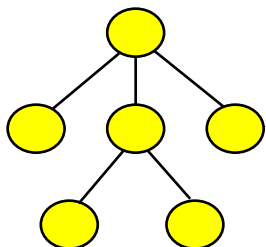
7.1 图的定义及操作

线性表:



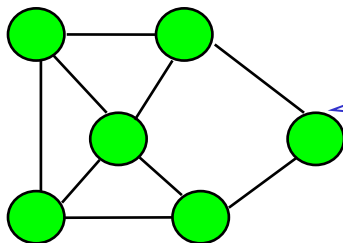
元素之间是线性关系，即表中元素最多一个直接前驱和一个直接后继；

树:



元素之间是层次关系。除根外，元素只有唯一直接前驱，但可以有若干个直接后继；

图:



任意的两个元素都可能相关，即图中任一元素可以有若干个直接前驱和直接后继，属于网状结构类型。



7.1 图的定义及操作

1. 图的定义

图（**Graph**）是一种非线性数据结构，由顶点(**Vertex**)和边(**Edge**)组成。
表示为：

$$Graph = (V, R)$$

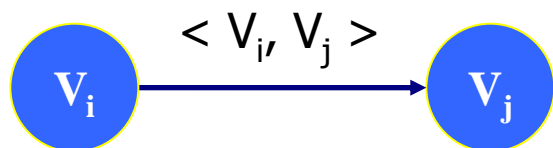
其中, V 是顶点集, R 是边集。

7.1 图的定义及操作

2. 基本术语

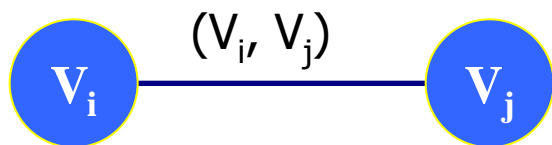
(1) 有向图与无向图

有向图（**Digraph**）：边有方向性。 $\langle V_i, V_j \rangle$ 与 $\langle V_j, V_i \rangle$ 是不同的。



称 V_i 为**弧尾**（起点）， V_j 为**弧头**（终点）。

无向图（**Undigraph**）：边无方向性



7.1 图的定义及操作

(2) 无向完全图与有向完全图

无向完全图：每2个顶点之间都有边，即

边的条数 $e = \frac{1}{2}n(n-1)$ ， n 是顶点数

有向完全图：任意2个顶点之间都有方向相反的2条弧，即

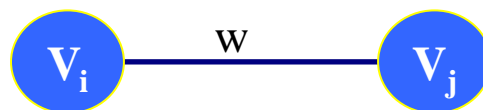
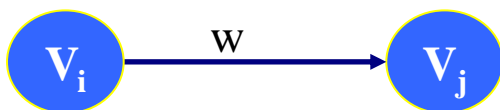
弧的条数 $e = n(n-1)$ ， n 是顶点数

(3) 稀疏图（**Sparse Graph**）和稠密图（**Dense Graph**）

弧或边的条数很少的图称为稀疏图，反之称为稠密图。

(4) 网（**Network**）

带权图（**Weighted Graph**），即弧或边上附加一个权值 w ：



权 w 的含义根据不同应用而定，如顶点表示城市， w 表示两个城市间的距离等。



7.1 图的定义及操作

(5) 子图 (Subgraph)

G的子图：由图**G**中选出顶点集的一个子集**Vs**以及与**Vs**中顶点相关联的一些边所构成的图。

(6) 顶点 V_i 的度、入度、出度

度：连接顶点 V_i 的边（弧）数

入度：以顶点 V_i 为终点的弧的个数，只用于有向图

出度：以顶点 V_i 为起点的弧的个数，只用于有向图

有向图顶点的度 = 入度 + 出度

所有顶点的度之和 = 边（弧）数 * 2



7.1 图的定义及操作

(7) 路径、简单路径、回路、简单回路

路径(Path): 从顶点 V_i 到 V_j 所经过的顶点序列

路径长度: 路径上边（弧）的个数

简单路径: 路径上的顶点不重复出现

回路(Cycle): 路径的起点和终点相同

简单回路: 除了起点和终点外，其他顶点不重复出现

(8) 无向连通图、强连通图

无向连通图: 无向图的任意2个顶点是**连通的**（顶点 V_i 和 V_j 间存在路径）

强连通图: 有向图的任意2个顶点是**连通的**
（顶点 V_i 到 V_j 以及 V_j 到 V_i 存在路径）

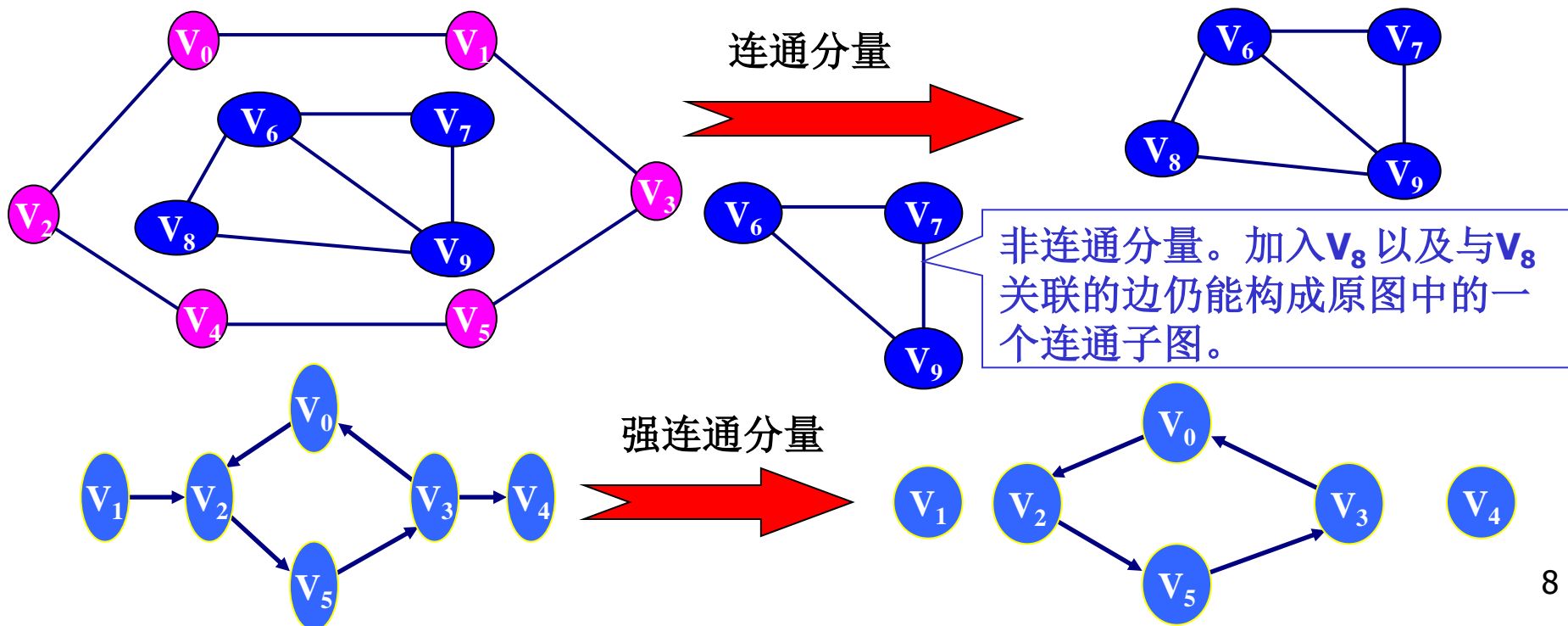
7.1 图的定义及操作

(9) 连通分量、强连通分量

连通分量：无向图的极大连通子图

强连通分量：有向图的极大强连通子图

极大：再加入顶点和边就不连通了



7.1 图的定义及操作

(10) 生成树 (Spanning Tree)

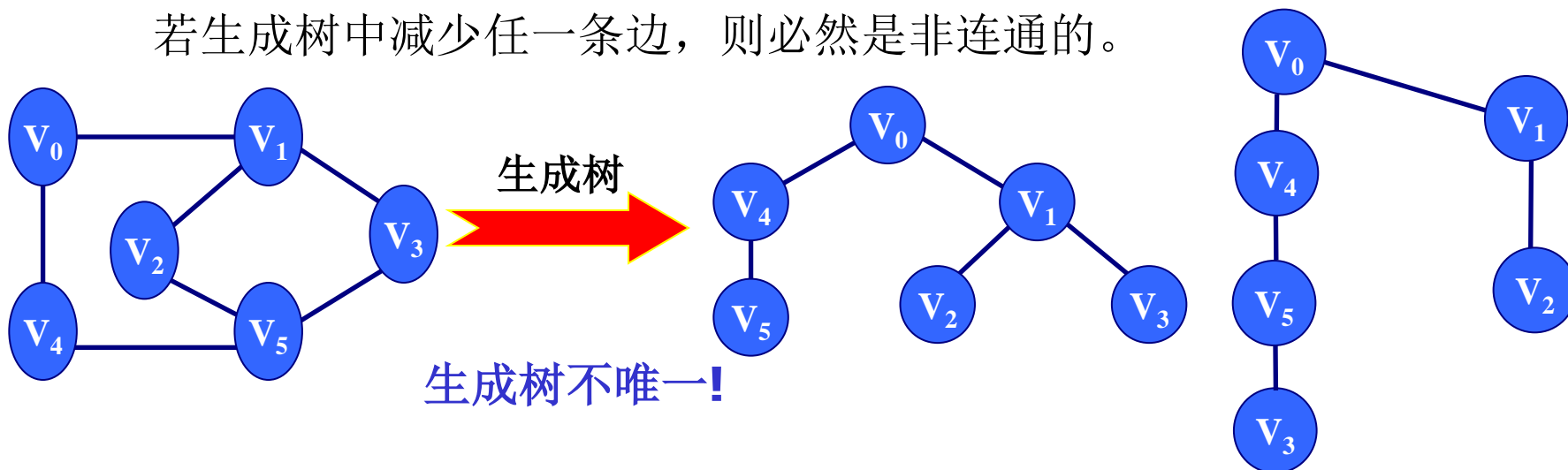
无向连通图的所有 n 个顶点构成的一个极小连通子图

特点:

- ✓ 包含全部 n 个顶点
- ✓ 包含足以构成一棵树的 $n-1$ 条边 (无回路)

若生成树中添加任一条属于原图的边, 则必定形成回路;

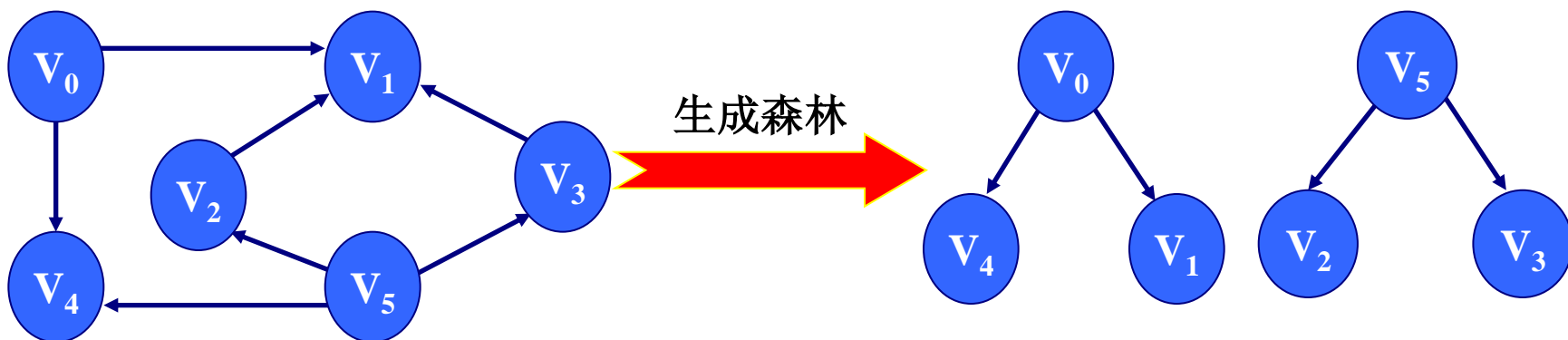
若生成树中减少任一条边, 则必然是非连通的。



7.1 图的定义及操作

(11) 生成森林 (Spanning Forest)

有向图的生成森林 F 由图中若干棵有向树组成。 F 是有向图的一个子图，包含图中全部顶点，但只有足以构成若干棵不相交的有向树的弧。



生成树、生成森林可由后面介绍的“遍历”等方法求出。



7.1 图的定义及操作

3. 图的抽象数据类型

ADT Graph {

数据元素集: $V = \{V_i | V_i \in \text{datatype}, i = 0, 1, 2, \dots, n-1, n \geq 0\}$

数据关系集: $R = \{ \langle V_i, V_j \rangle | V_i, V_j \in V \text{ 且存在 } V_i \text{ 到 } V_j \text{ 的弧或边}, 0 \leq i, j \leq n-1 \}$

基本操作集: P

GraphCreat(&G, V, R)

初始条件: V是图的顶点集, R是相应的关系集。

操作结果: 按照顶点集和关系集, 构造图G。

LocateVex(G, u) //按值查找

初始条件: 图G已存在, u为图中的顶点。//u表示顶点的值

操作结果: 若顶点 $u \in G$, 则返回u在图中的位置(地址或序号); 否则返回表示不在图中的信息(实际中如-1或NULL)。

GetVex(G, i) //按序号查找

初始条件: 图G已存在。

操作结果: 取图G中顶点 V_i , 若G中不存在第i顶点, 则返回“空值”。

图的抽象数据类型表示

FirstAdj(G, u)

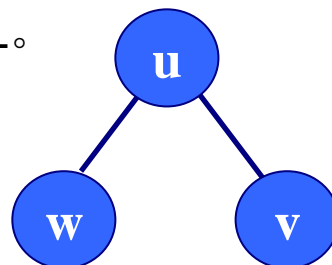
初始条件：图G已存在，u为图中的顶点。

操作结果：返回G中顶点u的第一邻接点位置（序号或地址）。若不存在邻接点，则返回-1或NULL。

NextAdj(G, u, w)

初始条件：图G已存在，u为图中的顶点，w为u的邻接点。

操作结果：返回G中顶点u关于顶点w的下一个邻接点位置（序号或地址）。若w是u的最后一个邻接点，则返回-1或NULL。



GraphTraverse(G)

初始条件：图G存在。

操作结果：依照某种规则对图中的顶点利用visit()函数逐一进行访问（visit()是根据图的具体数据和应用编写的访问函数）。

.

}ADT Graph;

7.2 图的存储结构

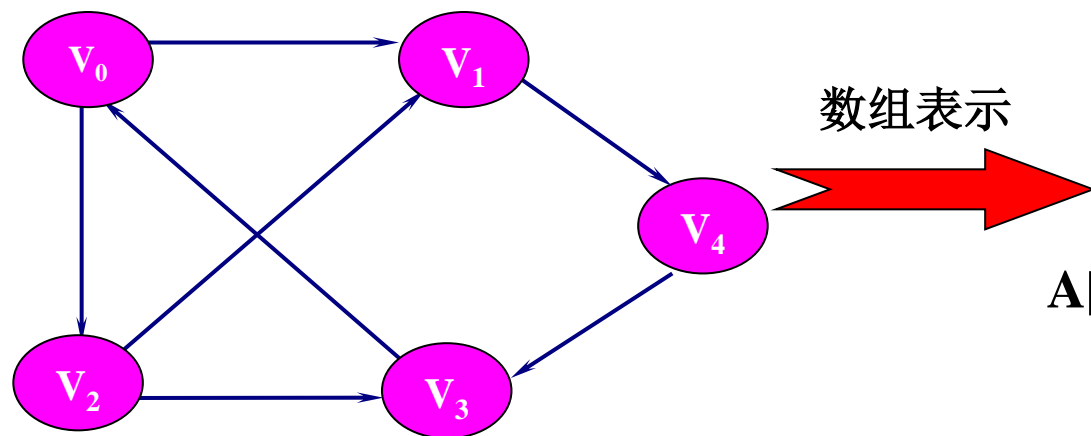
1. 邻接矩阵（数组表示法）

以二维数组表示图中顶点之间的相邻关系（邻接矩阵）

1个一维数组存储各顶点的数据（顶点表）

设有 n 个顶点，则邻接矩阵为 $A[n][n]$:

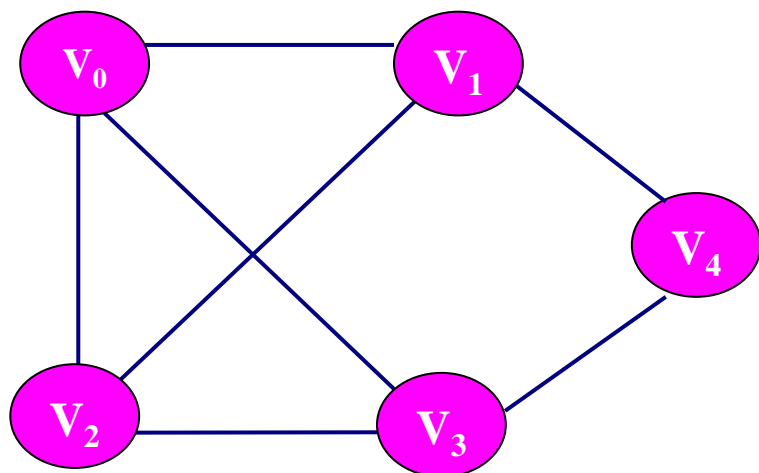
$$A_{ij} = \begin{cases} 1 & \text{若存在弧 } \langle V_i, V_j \rangle \text{ 或边 } (V_i, V_j) \\ 0 & \text{否则} \end{cases}$$



V	V_0	V_1	V_2	V_3	V_4
	0	1	2	3	4

$$A[5][5] = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

邻接矩阵



数组表示

V	V ₀	V ₁	V ₂	V ₃	V ₄
	0	1	2	3	4
A[5][5]=	0	1	1	1	0
	1	0	1	0	1
	1	1	0	1	0
	1	0	1	0	1
	0	1	0	1	0

特点:

- 1) 无向图的邻接矩阵是对称的。
- 2) 容易求出各顶点的度。

无向图: 顶点 V_i 的度 $D(V_i) = \mathbf{A}$ 中第 i 行(或列) 元素之和

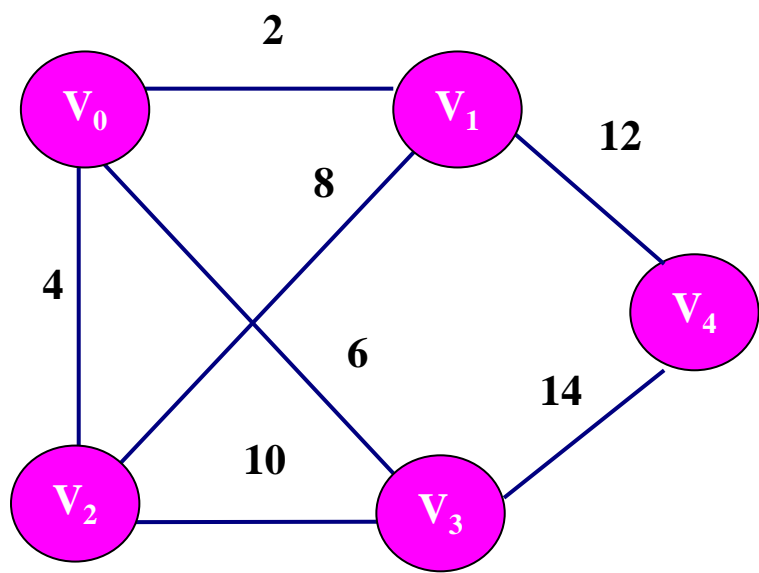
有向图: 顶点 V_i 的出度 $OD(V_i) = \mathbf{A}$ 中第 i 行元素之和

顶点 V_i 的入度 $ID(V_i) = \mathbf{A}$ 中第 i 列元素之和

邻接矩阵

网的邻接矩阵 A 取值为:

$$A_{ij} = \begin{cases} w_{ij} & \text{若存在弧 } \langle V_i, V_j \rangle \text{ 或边 } (V_i, V_j) \text{ 且权值为 } w_{ij} \\ \infty & \text{否则 (具体实现时取什么值?)} \end{cases}$$



数组表示

V	V ₀	V ₁	V ₂	V ₃	V ₄
	0	1	2	3	4

$$A = \begin{bmatrix} \infty & 2 & 4 & 6 & \infty \\ 2 & \infty & 8 & \infty & 12 \\ 4 & 8 & \infty & 10 & \infty \\ 6 & \infty & 10 & \infty & 14 \\ \infty & 12 & \infty & 14 & \infty \end{bmatrix}$$



邻接矩阵

类型描述:

```
#define MAXN 64      //最大顶点数
typedef char vtype;   //设顶点的数据为字符类型
typedef int adjtype;  //设邻接矩阵A中元素 $a_{ij}$ 为整型
typedef struct {
    vtype V[MAXN];      //顶点表
    adjtype A[MAXN][MAXN]; //邻接矩阵
} mgraph;
```


7.2 图的存储结构

2. 邻接表

类似于树的孩子链表示法

顶点表 + 图中每一顶点发出的弧或边构成的若干单链表

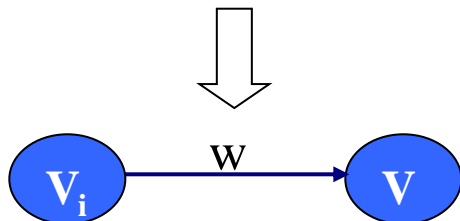
顶点节点:

data	farc
------	------

data: 顶点 V_i 的数据; farc: 指向该顶点发出的第一条弧或边节点的指针

链表节点:

adj	w	next
-----	---	------



adj: 邻接点域, 存放与 V_i 相邻接的顶点 V_j 在顶点表中的序号;

w: 弧或边上的权值; next: 指向与 V_i 相邻接的下一条弧或边节点



邻接表

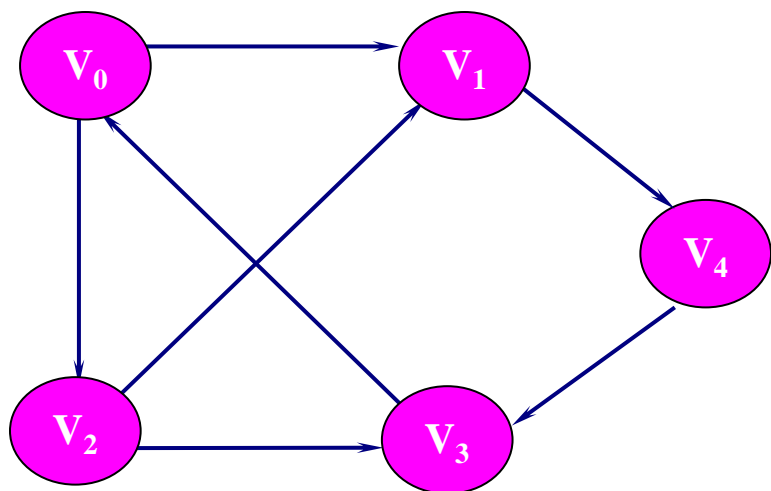
类型描述:

```
typedef struct node { //链表节点类型
    int adj; //邻接点
    int w; //权值
    struct node *next; //指向下一条弧或边
}linknode;

typedef struct { //顶点类型
    vtype data; //顶点值
    linknode *farc; //指向与本顶点关联的第一条弧或边
}Vnode;

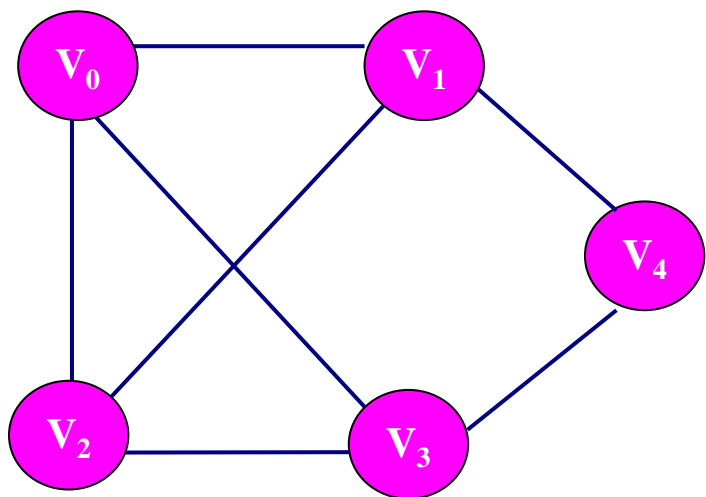
Vnode G[MAXN]; //顶点表
```

邻接表



顶点表

	data	farc
0	V ₀	• → 1 • → 2 ^
1	V ₁	• → 4 ^
2	V ₂	• → 1 • → 3 ^
3	V ₃	• → 0 ^
4	V ₄	• → 3 ^



	data	farc
0	V ₀	• → 1 • → 2 • → 3 ^
1	V ₁	• → 0 • → 2 • → 4 ^
2	V ₂	• → 0 • → 1 • → 3 ^
3	V ₃	• → 0 • → 2 • → 4 ^
4	V ₄	• → 1 • → 3 ^

邻接表

邻接表的特点:

1) 无向图: 邻接表中有重复信息, 当边数为 e 时, 链表节点数为 $2e$

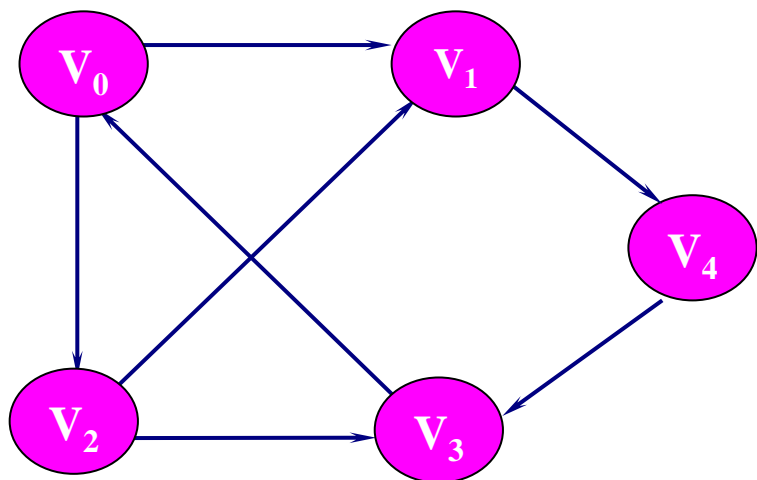
顶点 V_i 的度 $D(V_i) = V_i$ 所指单链表的节点个数

2) 有向图: 容易求顶点的出度

顶点 V_i 的出度 $OD(V_i) = V_i$ 所指单链表的节点个数

顶点 V_i 的入度 $ID(V_i) =$ 所有链表中 $adj=i$ 的节点个数

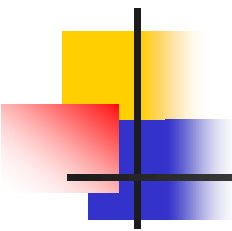
为便于求入度, 可建立逆邻接表



逆邻接表



	data	farc
0	V_0	• → 3 ^
1	V_1	• → 0 • → 2 ^
2	V_2	• → 0 ^
3	V_3	• → 2 • → 4 ^
4	V_4	• → 1 ^



7.2 图的存储结构

3. 十字链表

表示有向图，是邻接表和逆邻接表的结合

两类节点：顶点节点，弧节点

顶点节点：

data	fin	fout
-------------	------------	-------------

 每个顶点1个，构成顶点表

data: 顶点 V_i 的数据;

fin: 相应逆邻接表链指针，指向以 V_i 为终点（弧头）的第一弧节点;

fout: 相应邻接表的链指针，指向以 V_i 为起点（弧尾）的第一弧节点。

弧节点：

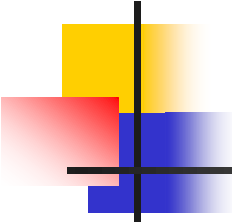
tail	head	hlink	tlink
-------------	-------------	--------------	--------------

 每条弧1个

tail: 弧尾 V_i 的序号(i); **head**: 弧头 V_j 的序号(j);

hlink: 指向终点（弧头）相同的下一个弧节点;

tlink: 指向弧尾（起点）相同的下一个弧节点。



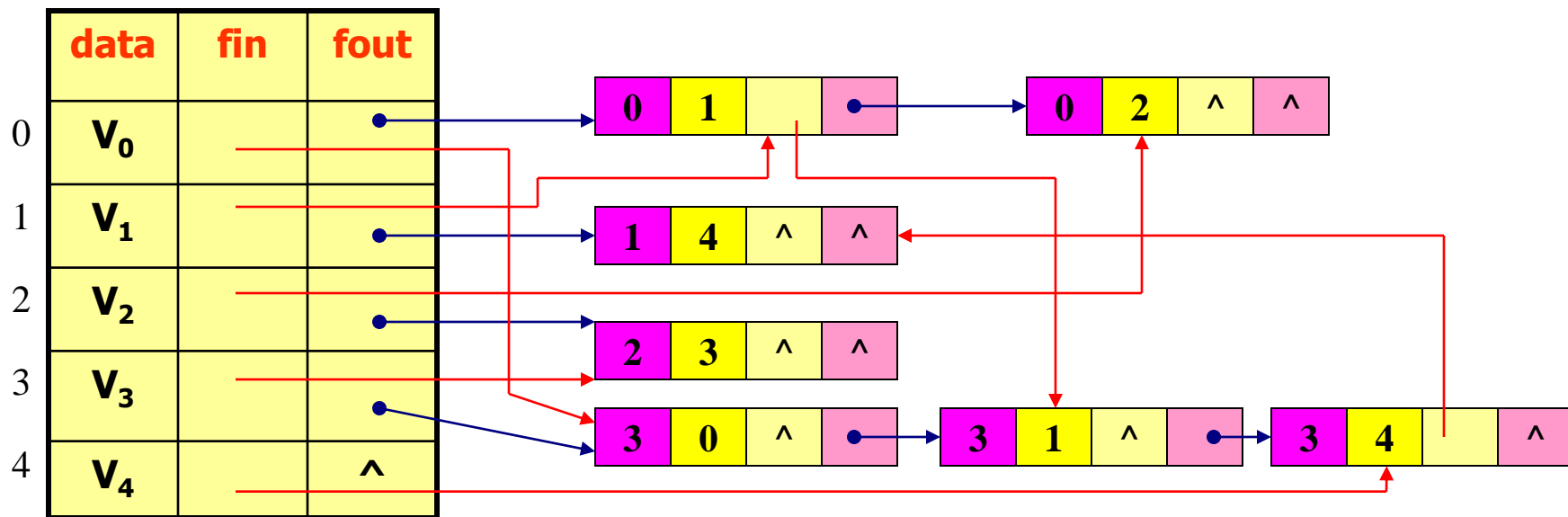
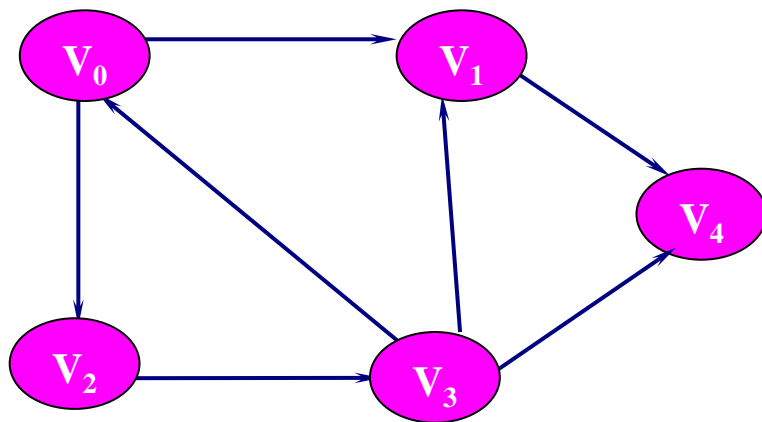
十字链表

类型描述:

```
typedef struct Anode { //弧节点
    int tail, head;
    struct Anode *hlink,*tlink;
}arcnode;
```

```
typedef struct Vnode { //顶点
    vtype data;
    arcnode *fin,*fout;
}vexnode;
vexnode G[MAXN]; //顶点表
```

十字链表



十字链表



7.2 图的存储结构

4. 邻接多重表

表示无向图

两类节点：顶点节点，边节点

顶点节点：

data	fedge
-------------	--------------

 每个顶点1个，构成顶点表

data: 顶点数据;

fedge: 指向与本顶点关联的第一条边节点。

边节点：

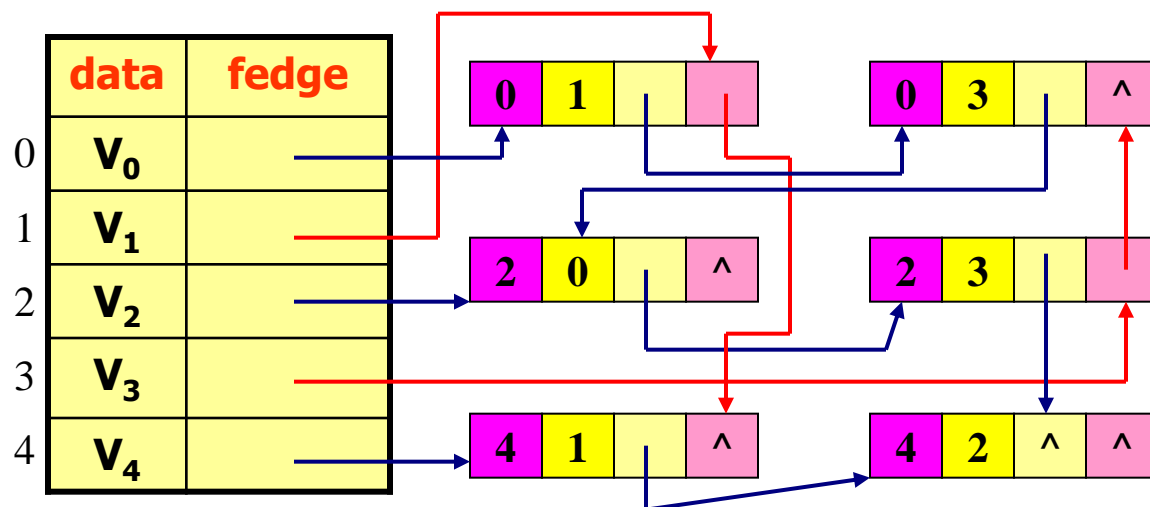
iv	jv	ilink	jlink
-----------	-----------	--------------	--------------

 每条边1个

iv和**jv**分别存放边 (V_i, V_j) 中 V_i 和 V_j 在顶点表的序号(**i**和**j**);

ilink: 指向与 V_i 关联的下一条边节点;

jlink: 指向与 V_j 关联的下一条边节点。



十字链表和邻接多重表的共同特点:

每条边（弧）对应**1**个节点



7.3 图的遍历

从某顶点出发，顺着边（弧）访问所有顶点各1次

1. 两种遍历方法：广度优先，深度优先

(1) 广度优先搜索（**Breadth First Search, BFS**）

类似于树的层次遍历。设从 V_0 出发

- 1) 访问 V_0 ;
- 2) 依次访问 V_0 的所有邻接点 V_1, V_2, \dots, V_t ;
- 3) 依次访问 V_1 的所有邻接点;
依次访问 V_2 的所有邻接点;
.....
依次访问 V_t 的所有邻接点;

直到所有顶点被访问完。

注意：不能重复访问。



7.3 图的遍历

(2) 深度优先搜索 (**Depth First Search, DFS**)

类似于树的前序遍历。

从某个顶点出发，沿着图的某个分支搜索直到头，
然后回溯 (**Backtracking**)，沿着另一分支搜索。

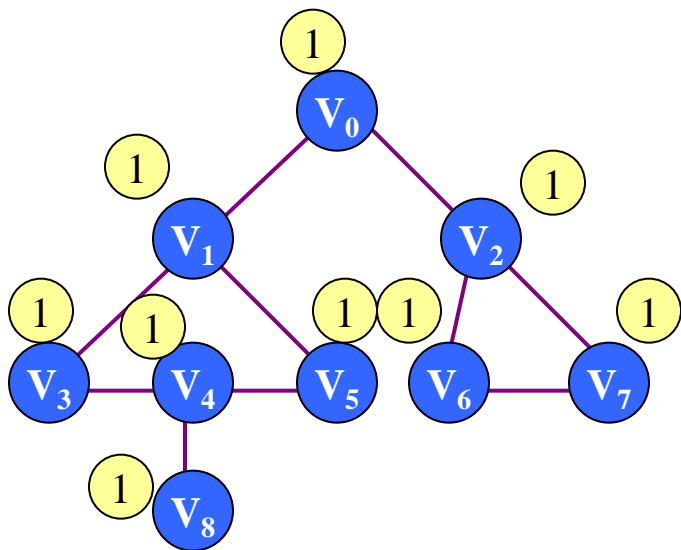
注意：不能重复访问（图可能有回路，避免陷入死循环）

7.3 图的遍历

2. 两种遍历方法的实现

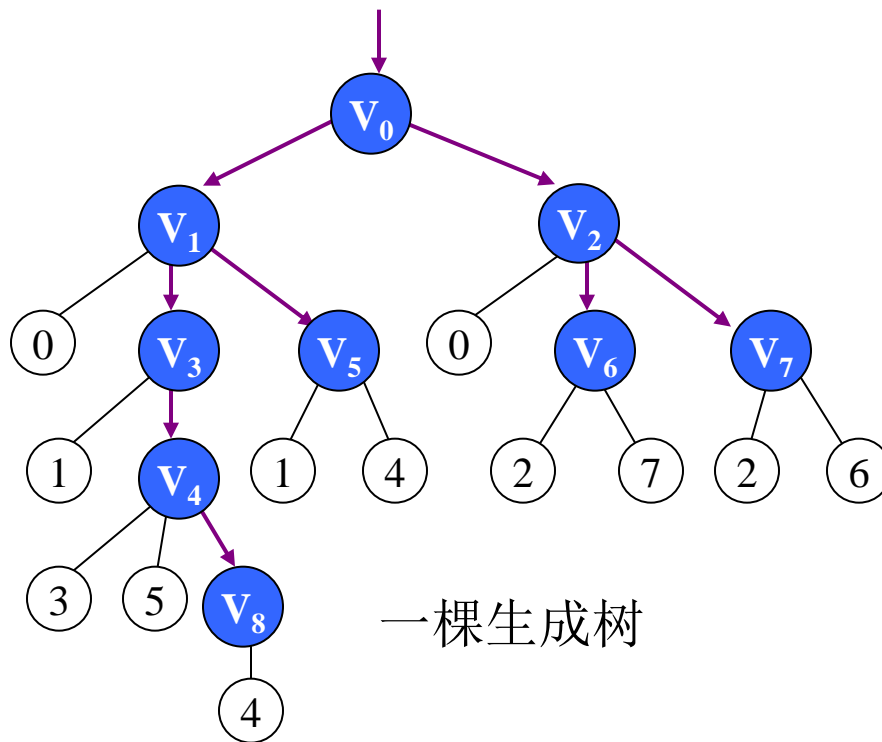
为了避免重复访问，每个顶点设置1个标志visited: (0:未访问, 1:已访问)

(1) 广度优先搜索BFS



BFS: $V_0 V_1 V_2 V_3 V_5 V_6 V_7 V_4 V_8$

实现BFS采用什么数据结构? **队列!**





广度优先搜索算法

算法思路:

采用队列。

每访问1个顶点后，让该顶点的序号进队；

然后队头出队，将未访问过的邻接点访问后再依次进队；

．．．．．

直到队空为止；

广度优先搜索算法

算法描述:

```
void BFS(Vnode G[], int v) //对图G从序号为v的顶点出发, 按BFS遍历
{
```

```
    int u; qtype Q; ClearQueue(Q); //置队Q为空
```

```
    visit(G,v); visited[v] = TRUE; //访问顶点, 置标志为“已访问”
```

```
    EnQueue(Q, v); //v进队
```

```
    while (!EmptyQueue(Q)) { //队非空时
```

```
        v = DeQueue(Q); //出队, 队头送v
```

```
        u = FirstAdj(G, v); //取v的第一邻接点序号
```

```
        while (u >= 0) {
```

```
            if (visited[u] == FALSE) { //u未访问过
```

```
                visit(G, u); visited[u] = TRUE;
```

```
                EnQueue(Q, u);
```

```
            }
```

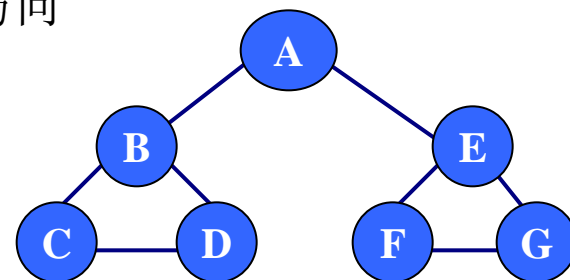
```
            u = NextAdj(G, v, u); //取v关于u的下一邻接点
```

```
        }
```

```
    }
```

```
}
```

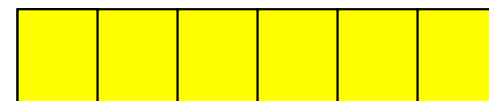
例:



访问:

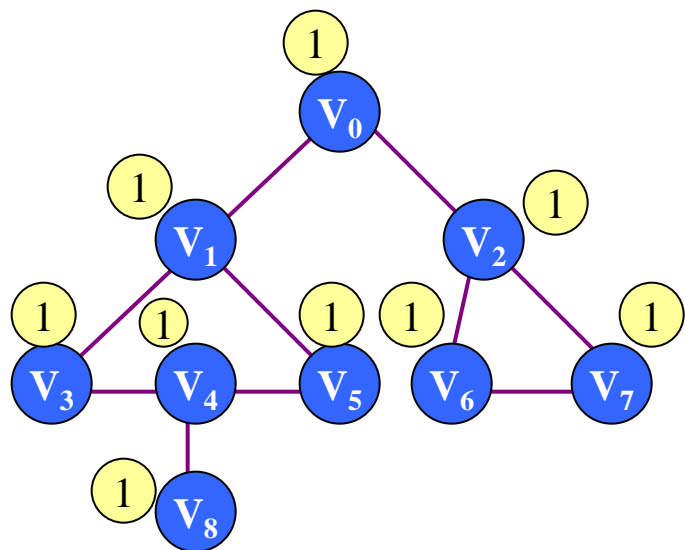


队列:



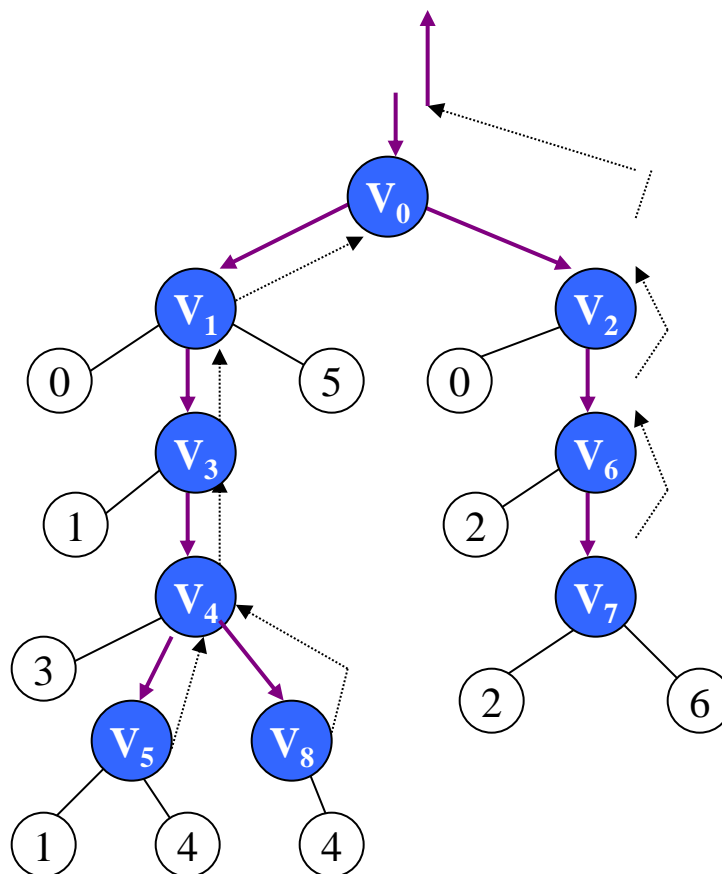
7.3 图的遍历

(2) 深度优先搜索DFS



DFS: $V_0 V_1 V_3 V_4 V_5 V_8 V_2 V_6 V_7$

如何实现？



一棵生成树



深度优先搜索算法

算法思路:

递归。

设从 V_0 出发, V_0 的邻接点为 V_1, V_2, \dots, V_t

1) 访问 V_0 ;

2) 从 V_1 出发, 按DFS遍历;

从 V_2 出发, 按DFS遍历;

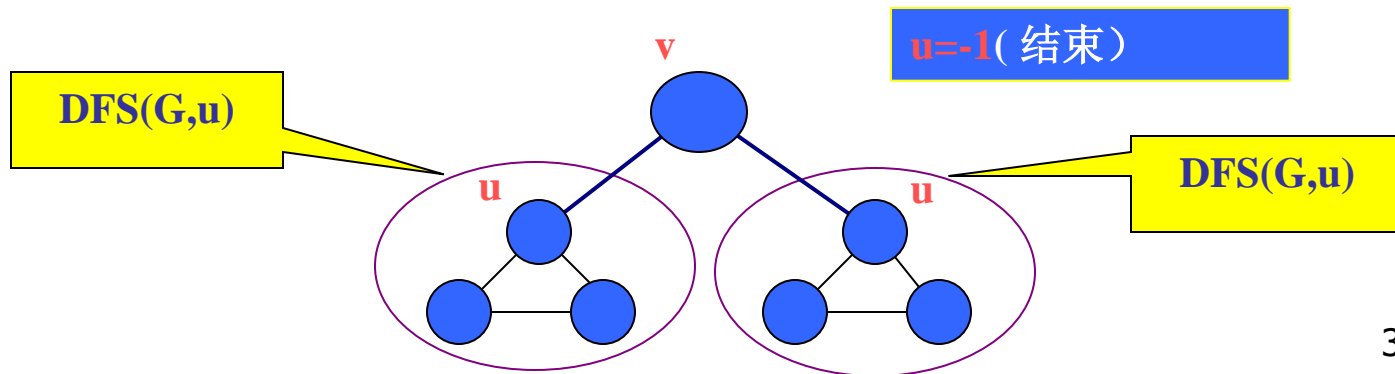
\dots

从 V_t 出发, 按DFS遍历;

深度优先搜索算法

算法描述:

```
void DFS(Vnode G[], int v) //对图G从序号为v的顶点出发，按DFS搜索
{
    int u;
    visit(G, v); //访问v号顶点
    visited[v] = TRUE; //置标志位为”已访问”
    u = FirstAdj(G, v); //取v的第一邻接点序号u
    while (u >= 0) { //当u存在时
        if (visited[u] == FALSE)
            DFS(G, u); //递归调用遍历从u出发的子图
        u = NextAdj(G, v, u); //取v关于当前u的下一邻接点序号
    }
}
```





7.3 图的遍历

说明：

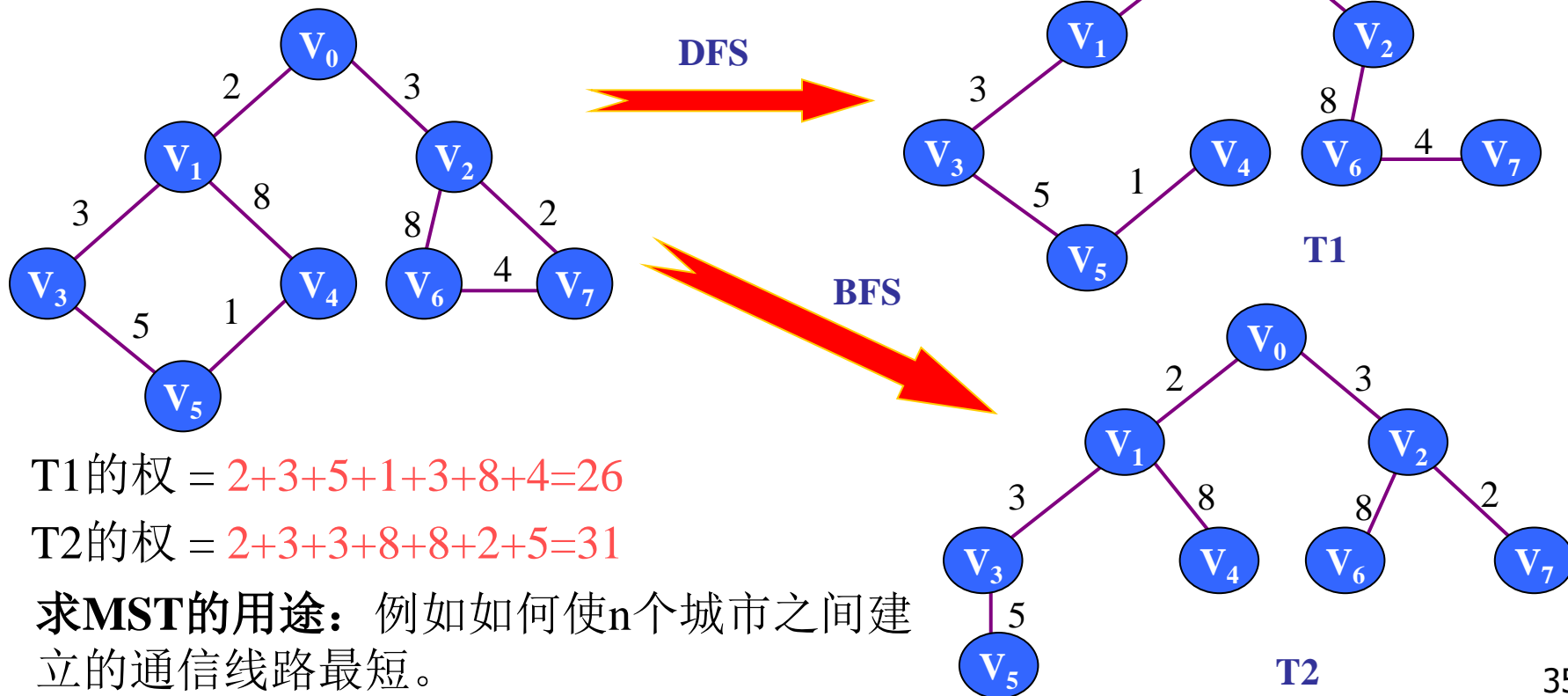
- 1) 遍历结果不是唯一的。但是，**对于确定的存储结构和算法，遍历结果唯一**；
- 2) 对于连通图，**1次BFS或DFS**可访问每个顶点各**1次**；而对于非连通图，**1次BFS或DFS**只能得到**1个**连通分量；
- 3) 对连通图的**1次遍历（ BFS或DFS ）**，可得到**1棵**生成树。

7.4 最小生成树

最小生成树（Minimum Spanning Tree, **MST**）：

边的权值之和最小的生成树。

边的权值之和称为生成树的权。



T_1 的权 = $2+3+5+1+3+8+4=26$

T_2 的权 = $2+3+3+8+8+2+5=31$

求MST的用途：例如如何使n个城市之间建立的通信线路最短。



7.4 最小生成树

构造MST的2个经典算法：Prim算法和Kruskal算法。

1. Prim算法

方法：从图中任一顶点N开始，将顶点N加入MST

- 1) 选出连接N的边中权值最小的1条，设连接到M，将M与边(N, M)加入MST；
- 2) 选出与N或M相连的其他顶点的边中权值最小的1条，将此边与新顶点加入MST；
- ...

直到MST包含 $n-1$ 条边为止。



在连接当前已在MST中的顶点与不在MST中顶点的边中，选出权值最小的边来扩展MST。



Prim 算法

Prim算法:

设无向连通网 $G = (V, R)$, 所求的 $MST = (U, TR)$ (U 、 TR 分别为顶点集和边集)。

初始化 $U = \{V_0\}$, $TR = \Phi$; // V_0 为 G 中任一顶点

while ($U \neq V$) {

 寻找边 (u, v) , 其中 $u \in U$, $v \in V - U$, 且 (u, v) 权值最小;

 将顶点 v 加入 U , 边 (u, v) 加入 TR ;

}

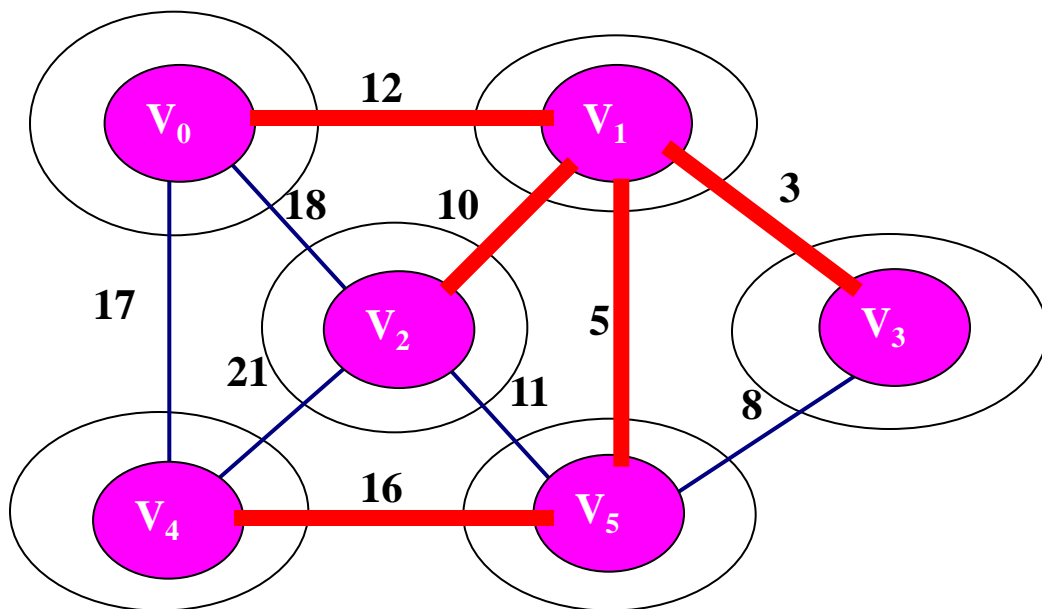
Prim算法属于贪心算法。

定理: Prim算法产生的是最小生成树。

可以用反证法证明。

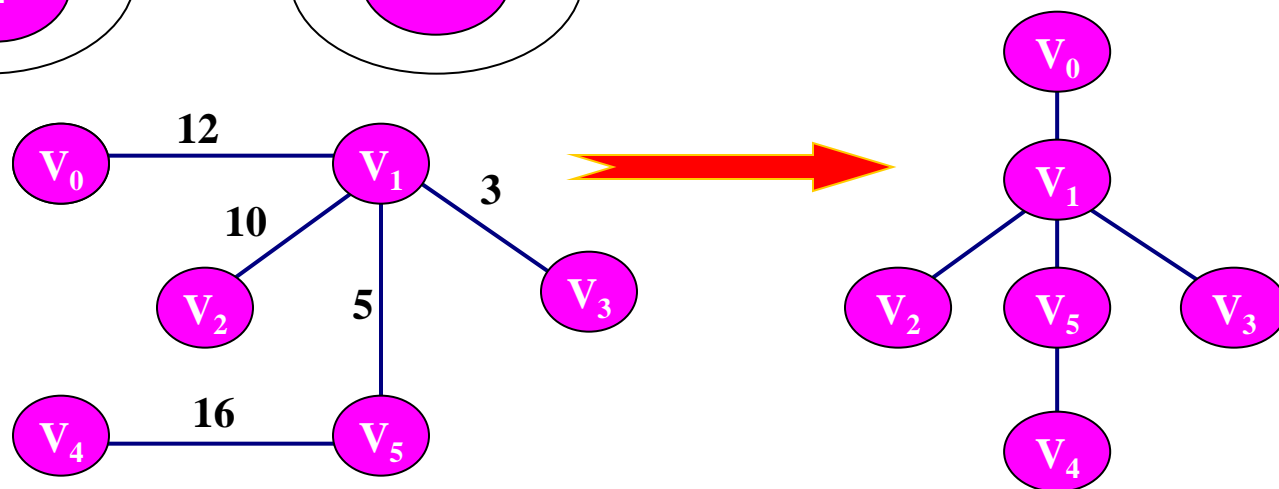
Prim 算法

令 $U = \{V_0\}$



最小生成树:

$T = (U, TR)$



权值之和: $12+3+5+10+16=46$ (最小)

Prim 算法

实现**Prim**算法的一个关键问题：

每次从哪些边中找最小的呢？

设从A开始

候选边集：

(A, B) ∞ , (A, C) 7, (A, D) ∞ , (A, E) 9, (A, F) ∞

调整候选边集：

(A, C), ~~(A, B) ∞ , (A, D) ∞ , (A, E) 9, (A, F) ∞~~

只需保留权小的

MST (C, B) 5, (C, D) 1, ~~(C, E) ∞ , (C, F) 2~~

(A, C), (C, D) (C, B) 5, (A, E) 9, (C, F) 2

MST

~~(D, B) ∞ , (D, E) ∞ , (D, F) 2~~

(A, C), (C, D), (C, F) (C, B) 5, ~~(A, E) 9~~

MST

~~(F, B) 6, (F, E) 1~~

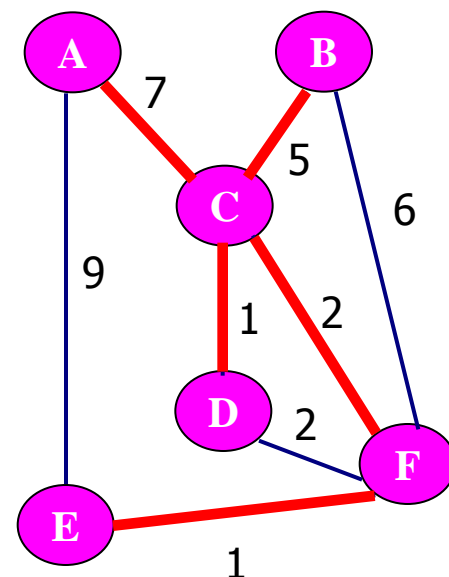
(A, C), (C, D), (C, F), (F, E) (C, B) 5

MST

~~(E, B) ∞~~

(A, C), (C, D), (C, F), (F, E), (C, B)

MST



若用**(D, F)**替代**(C, F)**,
可得到另一个**MST**。

Prim算法描述

设无向连通网用邻接矩阵表示，图G的邻接矩阵为：

$$A = \begin{bmatrix} \infty & 12 & 18 & \infty & 17 & \infty \\ 12 & \infty & 10 & 3 & \infty & 5 \\ 18 & 10 & \infty & \infty & 21 & 11 \\ \infty & 3 & \infty & \infty & \infty & 8 \\ 17 & \infty & 21 & \infty & \infty & 16 \\ \infty & 5 & 11 & 8 & 16 & \infty \end{bmatrix} \quad (\text{顶点数为 } n = 6)$$

2.算法描述

```
typedef struct    // 定义一条边
{ int pre,end; int w; // 边的起点和终点序号、权值 }edge;
edge TR[n-1];    // 存放生成树的n-1条边
typedef struct    // 数组表示法的图结构
{ vtype v[maxn]; // 顶点表
  adjtype A[maxn][maxn]; // 邻接矩阵
}mgraph; // 设当前网的邻接矩阵已建立
```




Prim算法描述

```

void prim(mgraph G, int n) // 从顶点 $v_0$ 开始求网G的最小生成树的Prim算法
{ int i,j,k,m,v,min,d; edge e;
  for (i=1;i<n;i++) // 构造初始候选边
  { TR[i-1].pre=0; // 顶点 $v_0$ （序号为0）为第一个加入树的顶点
    TR[i-1].end=i; TR[i-1].w=G.A[0][i]; }
  for (j=0;j<n-1;j++) // 求出n-1条边
  { min=max; // max为当前机器的最大整数
    for (k=j;k<n-1;k++); // 在候选边中找权值最小的边
    if (TR[k].w<min)
    { min=TR[k].w; m=k; } // TR[m]为当前权最小的边
    e=TR[m]; TR[m]=TR[j]; TR[j]=e; // 将所求最小边放在第j个最小边位置
    v=TR[j].end; // v为新的出发点序号
    for (k=j+1;k<n-1;k++) // 调整候选边集
    { d=G.A[v][TR[k].end];
      if (d<TR[k].w) { TR[k].w=d; TR[k].pre=v; } }
  } }

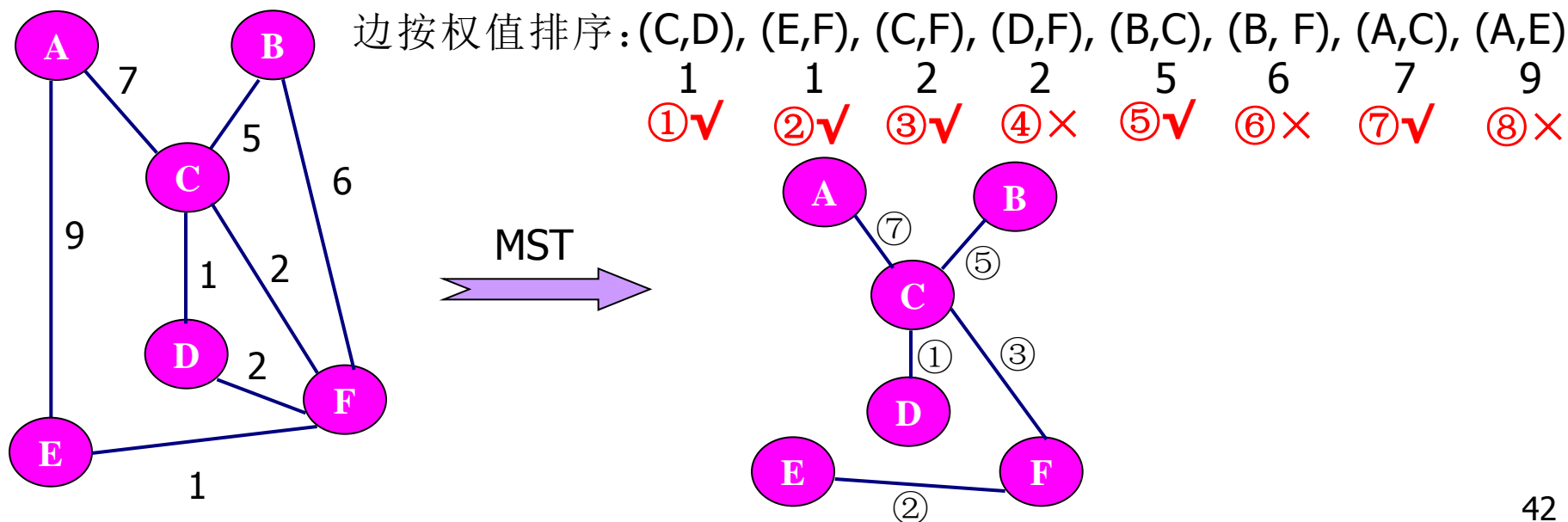
```

7.4 最小生成树

2. Kruskal 算法

方法：设无向连通网 $G=(V, R)$, 所求 $MST=(U, TR)$

- 1) 令MST包含G中所有顶点，边集为空，即 $U = V, TR=\Phi$;
- 2) 将G中所有的边按权值从小到大排序;
- 3) 依次考察排序后的各边 (u, v) , 若 (u, v) 加入MST后不会形成回路，则将其加入, 否则舍去。直到边数 $=n-1$;





Kruskal 算法

实现**Kruskal**算法的一个关键问题：

如何判断所选的边加入后是否形成回路？

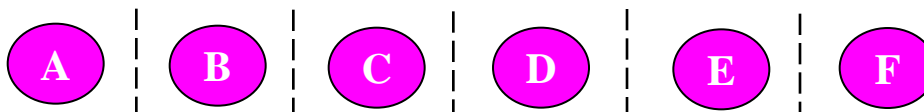
方法：用集合

- ✓ 初始时设置 n 个集合，每个顶点各自属于1个集合；
- ✓ 若将边 (u, v) 加入MST，则将 u 、 v 所属集合合并为1个；
- ✓ 每个集合中的任2个顶点是被已选出的边连通了的，分属不同集合的2个顶点目前在MST中不连通；
- ✓ 若 u 、 v 已在同一个集合，则说明 u 、 v 已被选出的边连通了，若加入边 (u, v) 就会形成回路。

Kruskal 算法

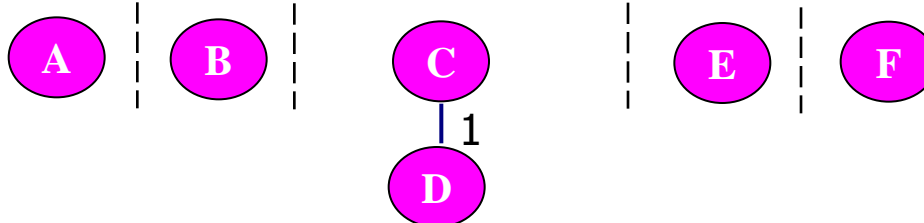
集合的变化:

初始状态:



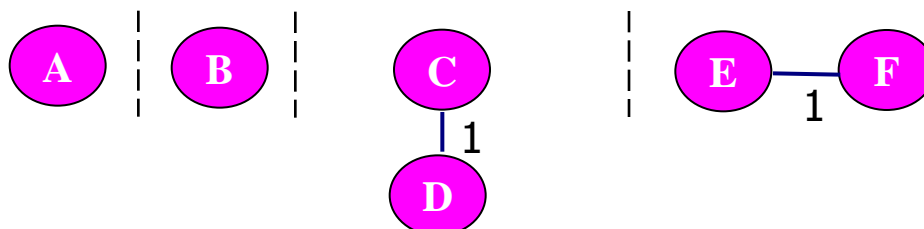
第1步:

处理边(C,D)



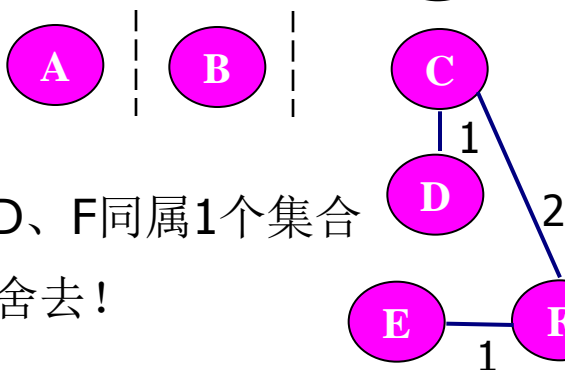
第2步:

处理边(E,F)



第3步:

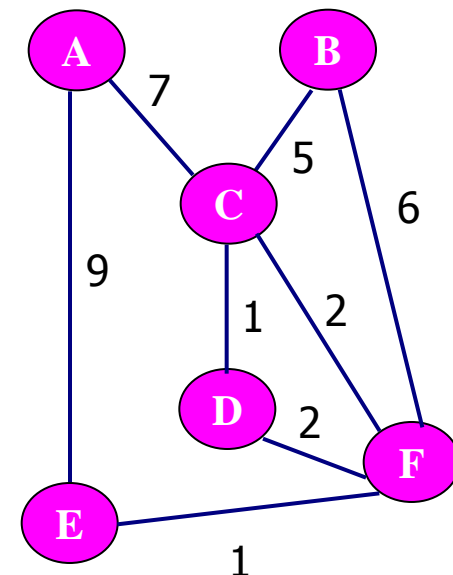
处理边(C,F)



第4步:

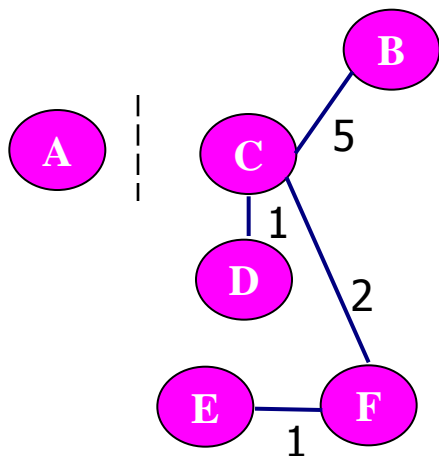
D、F同属1个集合

处理边(D,F) 舍去!



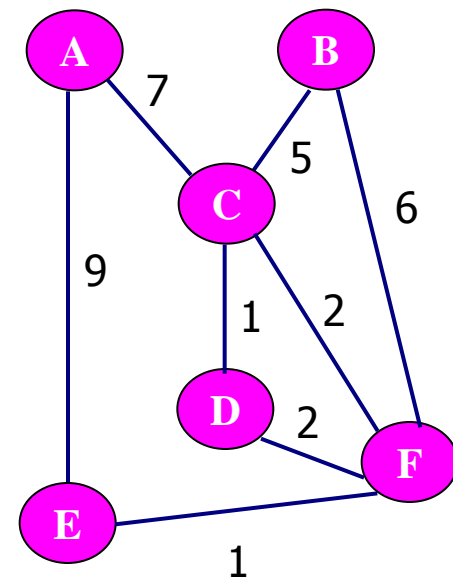
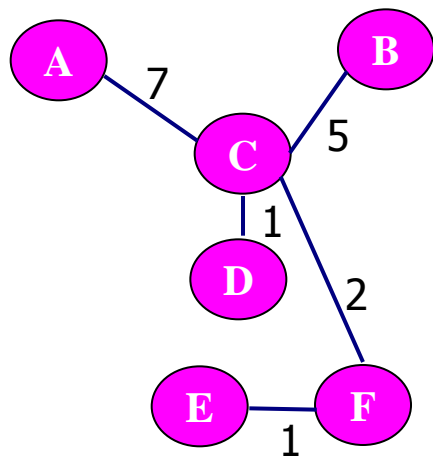
Kruskal 算法

第5步：
处理边(B,C)



第6步：
处理边(B,F) 舍去！
B、F同属1个集合

第7步：
处理边(A,C)





Kruskal算法描述

```
void Kruskal(mgraph G, int n) // 求无向连通网G最小生成树的Kruskal算法
{
    int i,k; edge e,GR[],TR[n-1];
    Createdge(G,GR,k); // 由邻接矩阵G.A[n][n]形成各边, 边集GR, k为边数
    SORT(GR,k);        // 对GR中各边按权值排序
    EmptyTR(TR); // 置TR为空集
    i=0;
    while (i<k) // 考察GR中各边
    {e=Getedge(GR ,i); // 取GR中第i条边 (u,v) 加入e
        if (!cycle(TR,e)) // 若边e加入TR后形成回路,cycle()返回True,
                        // 否则返回False
            Putedge(TR,e); // 边e加入树边集TR
        i++;
    }
}
```



7.5 最短路径问题

若求一个顶点到另一个顶点所含边数最少的路径，可以从指定的顶点出发，对图作广度优先搜索，一旦遇到目标顶点就终止。

本节针对带权图，求两顶点间的**最短路径：路径上边的权值之和最小的那条。**

应用领域：从一个城市到另一个城市的最短距离、时间等；

计算机网络中如何找到一种时间开销最小的方式从一台机器向网上其他机器发送消息？

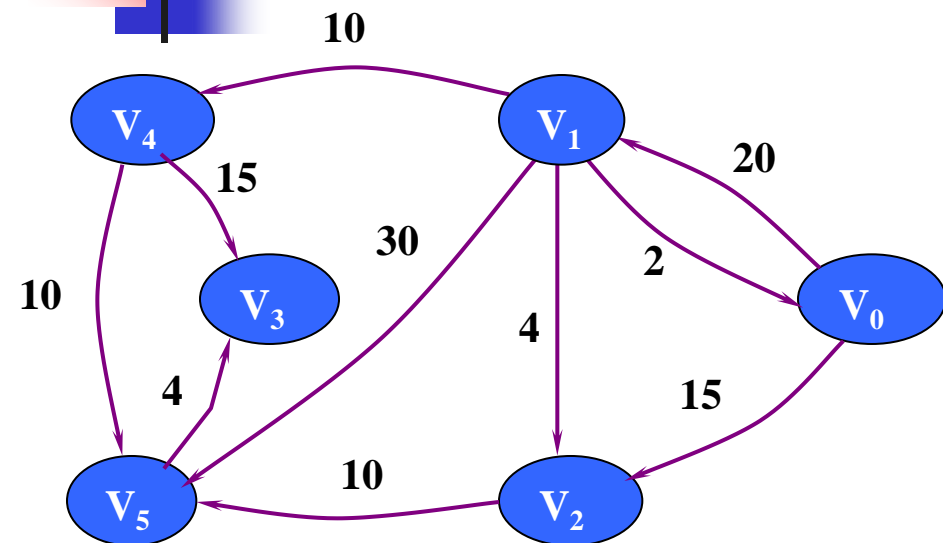
1. 单源点最短路径问题

求某源点 v 到图中其余各顶点的最短路径。

Dijkstra算法：按路径长度递增的次序逐步生成最短路径。

时间复杂度是 $O(n^2)$

Dijkstra 算法



V_0 到其余各顶点的最短路径及长度 L 分别如下(为方便, 各顶点用序号表示):

终点	最短路径	长度
1	0, 1	20
2	0, 2	15
3	0, 2, 5, 3	29
4	0, 1, 4	30
5	0, 2, 5	25

将这些最短路径按长度递增排列:

终点	最短路径	长度
2	0, 2	15
1	0, 1	20
5	0, 2, 5	25
3	0, 2, 5, 3	29
4	0, 1, 4	30

从中可以发现什么求解思路?

可以发现如下性质 (**按递增次序求**): V_0 为源点
 设 S 为已求出了最短路径的顶点集合 (S 初始为空), 则**下一条较短的最短路径 (设终点为 x) 一定是(v_0, x)或者中间经过 S 中顶点到达 x 的路径中最短的那条。**



Dijkstra 算法

算法描述:

设顶点 V_0 为源点, 令 $\text{dist}[n]$ 存放最短路径长度, $\text{dist}[i]$ 为 V_0 到 V_i 的最短路径长度, S 为已求出最短路径的顶点集合

初始化 $S = \Phi$, $\text{dist}[i](0 < i < n)$ 为边 $\langle V_0, V_i \rangle$ 的权(若不存在边 $\langle V_0, V_i \rangle$, 则取 ∞);

for ($k = 1$; $k < n$; $k++$) {

 确定 u , 使得 $\text{dist}[u] = \min(\text{dist}[i] \mid i=1, \dots, n-1 \text{ 且 } V_i \text{ 不在 } S \text{ 中})$; //找最短的

$S = S \cup \{V_u\}$; // V_0 到 V_u 的最短路径已确定, 长度是 $\text{dist}[u]$

 更新 dist ;

}



for (每个不属于 S 的顶点 V_w) {

 if ($\text{dist}[w] > \text{dist}[u] + \langle V_u, V_w \rangle$ 的权值) //说明发现了更短的路径

 置 $\text{dist}[w] = \text{dist}[u] + \langle V_u, V_w \rangle$ 的权值;

}



Dijkstra 算法

相关的数据结构:

1) 带权图的表示

用邻接矩阵 $G.A[n][n]$, n 为顶点数

为方便, 各顶点用其序号 $0, 1, 2, \dots, n-1$ 代替

$$G.A[i][j] = \begin{cases} w_{ij} & \text{若 } \langle v_i, v_j \rangle \in R \text{ 且 } \langle v_i, v_j \rangle \text{ 上的权} = w_{ij} \\ \infty & \text{若 } \langle v_i, v_j \rangle \notin R \\ 0 & \text{若 } i = j \end{cases}$$

2) 集合 S 的表示 (S 为已求出最短路径的顶点集合)

向量 $S[n]$: 其中 $S[i] = \begin{cases} 1 & \text{当源点 } v \text{ 到 } v_i \text{ 的最短路径已求出时;} \\ 0 & \text{否则} \end{cases}$

初始令 $S[v]=1$ (即路径 (v, v) 已求出, $\text{dist}[v]=0$ 。 v 是源点), $S[i]=0$ ($0 \leq i \leq n-1, i \neq v$), 表示 v 到其它顶点的最短路径未求出。

Dijkstra 算法

3) 最短路径长度 \mathbf{dist} 的表示

向量 $\mathbf{dist}[n]$: $\mathbf{dist}[i]$ 存放从 \mathbf{v} 到 \mathbf{v}_i 的最短路径长度($0 \leq i \leq n-1$)。初始为:

$$\mathbf{dist}[i] = \begin{cases} \langle \mathbf{v}, \mathbf{v}_i \rangle \text{ 上的权 } w & \text{若 } \langle \mathbf{v}, \mathbf{v}_i \rangle \in R \\ \infty & \text{若 } \langle \mathbf{v}, \mathbf{v}_i \rangle \notin R \end{cases}$$

若 $\langle \mathbf{v}, \mathbf{v}_i \rangle \in R$, 则 \mathbf{v} 到 \mathbf{v}_i 的路径一定存在, 但不一定是最短的。

4) 最短路径的记录

向量 $\mathbf{path}[n]$: $\mathbf{path}[i]$ 存放从源点 \mathbf{v} 到 \mathbf{v}_i 的最短路径($\mathbf{v}, \dots, \mathbf{v}_i$)。

初始 $\mathbf{path}[i].\mathbf{pi}[0] = \{\mathbf{v}\}$ (表示从 \mathbf{v} 出发)。

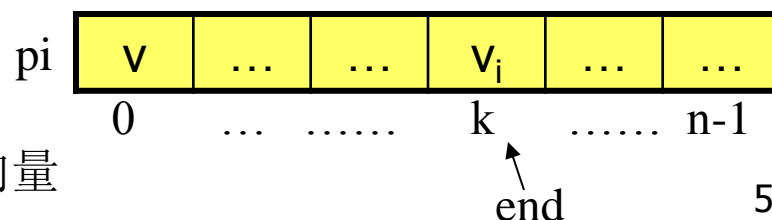
`typedef struct { //最短路径类型`

`int pi[n]; //存放 \mathbf{v} 到 \mathbf{v}_i 的一条最短路径, n 为图中顶点数`

`int end; //pi[0]至pi[end]为最短路径`

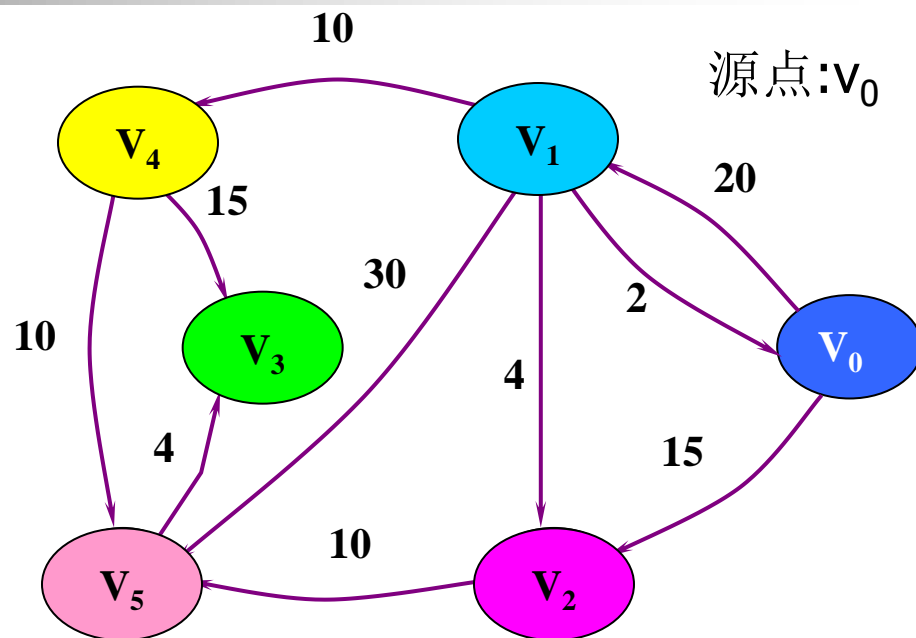
`}pathtype;`

`pathtype path[n]; // \mathbf{v} 到各顶点的最短路径向量`



Dijkstra 算法

	S	dist	path
0	1	0	0
1	1	20	0,1
2	1	15	0,2
3	1	29	0,2,5,3
4	1	30	0,1,4
5	1	25	0,2,5



找满足 $S[u]=0$ 且 $dist[u]$ 最小的，然后修改各向量：

对于每个 $S[u]=0$ 的顶点 w ，若 $dist[w] > dist[u] + \langle u, w \rangle$ 的权，则令 $dist[w] = dist[u] + \langle u, w \rangle$ 的权

邻接矩阵：

$$G.A[n][n] = \begin{bmatrix} 0 & 20 & 15 & \infty & \infty & \infty \\ 20 & 0 & 4 & \infty & 10 & 30 \\ \infty & \infty & 0 & \infty & \infty & 10 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 15 & 0 & 10 \\ \infty & \infty & \infty & 4 & \infty & 0 \end{bmatrix}$$

第1条: $dist[2]=15$ ，最短路径(0, 2)可确定。

第2条: $dist[1]=20$ ，最短路径(0, 1)可确定。

第3条: $dist[5]=25$ ，最短路径(0, 2, 5)可确定。

第4条: $dist[3]=29$ ，最短路径(0, 2, 5, 3)可确定。

第5条: $dist[4]=30$ ，最短路径(0, 1, 4)可确定。

Dijkstra算法描述

```
typedef struct
```

```
{ int pi[n];    // 存放v到 $v_i$ 的一条最短路径, n为图中顶点数  
  int end;  
}pathtype;
```

```
pathtype path[n]; // v到各顶点最短路径向量
```

```
int dist[n]; // v到各顶点最短路径长度向量
```

```
void Dijkstra(mgraph G, pathtype path[], int dist[],int v,int n)
```

```
//求G(用邻接矩阵表示)中源点v到其他各顶点最短路径,n为G中顶点数
```

```
{ int i,count,s[n],m,u,w;
```

```
for(i=0;i<n;i++) //初始化
```

```
{s[i]=0;
```

```
dist[i]=G.A[v][i]; //v到其他顶点的权为当前最短路径,送dist[i]
```

```
path[i].pi[0]=v;
```

```
path[i].end=0;
```

```
}
```

. Dijkstra算法描述

```
s[v]=1; i=1;
while (i<=n-1)  // 求n-1条最短路径
{ m=max;          // max为当前机器表示的最大值
  for (w=0;w<=n-1;w++)  // 找当前最短路径长度
    if (s[w]==0 && dist[w]<m)
      { u=w; m=dist[w]; }
  if (m==max) break; // 最短路径求完（不足n-1条），跳出while循环
  s[u]=1;           // 表示v到vu最短路径求出
  path[u].end++;    // 置当前最短路径
  path[u].pi[end]=u;
  for (w=0;w<=n-1;w++) // u求出后，修改dist和path向量
    if (s[w]==0 && dist[w]>dist[u]+G.A[u][w])
      { dist[w]=dist[u]+G.A[u][w];
        path[w]=path[u]; }
  i++; // 最短路径条数计数
}
```



7.5 最短路径问题

2. 每一对顶点间的最短路径问题

可利用Dijkstra算法，每次以1个顶点为源点，反复执行n次即可。

时间复杂度是 $O(n^3)$ 。

Floyd算法：用试探法求网中任意两顶点间的最短路径。

时间复杂度是 $O(n^3)$ 。



Floyd 算法

算法思路： 设置矩阵 $D[n][n]$ ，以 $D[i][j]$ 表示顶点 V_i 到 V_j 的最短路径。

初始时， $D[n][n]$ 取邻接矩阵， $D[i][j]$ 为边的权(若 $i=j$ 取0；若边不存在取 ∞)，记作 D_{-1} 。当然， D_{-1} 的值不可能都是最短路径。

然后进行 n （ n 为顶点数）次试探：

1) 让路径经过 v_0 。对每一对 v_i 与 v_j ，比较 (v_i, v_j) 和 (v_i, v_0, v_j) 的路径长度，取小者为当前从 v_i 到 v_j 的最短路径，得 D_0 ；

即 D_0 是考虑了各顶点间的路径除了直达外，还可以经过 v_0 到达；

2) 在 D_0 的基础上，考虑让路径经过 v_1 。对每一对 v_i 与 v_j ，比较 (v_i, \dots, v_j) 和 $(v_i, \dots, v_1, \dots, v_j)$ 的路径长度，取小者为当前从 v_i 到 v_j 的最短路径，得 D_1 ；

· · · · ·

依此类推。

Floyd 算法

算法要点:

若从顶点 v_i 到 v_j 的路径经过一个新顶点 v_k 能使路径缩短, 则修改

$$D_k[i][j] = D_{k-1}[i][k] + D_{k-1}[k][j]$$

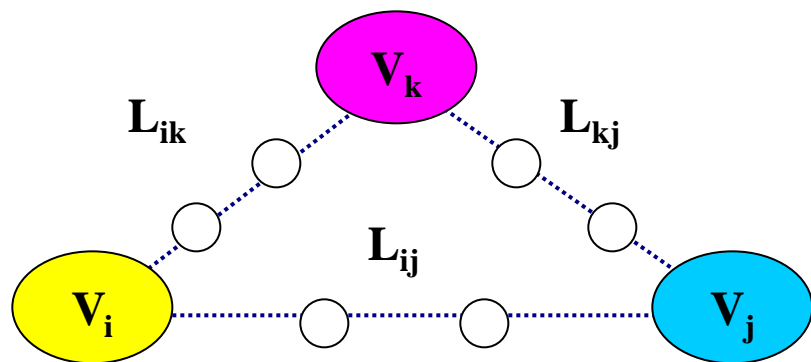
故 $D_k[i][j]$ 就是当前求得的从 v_i 到 v_j 的最短路径, 且其路径的顶点 (除了 v_i 和 v_j) 序号均不大于 k 。

这样, 经过 n 次试探, D_{n-1} 一定是各顶点间的最短路径长度。

设 L_{ij} 为当前 v_i 到 v_j 的路径长度, 即:

在 v_i 与 v_j 之间加进 v_k (k 从 $0 \sim n-1$) 后,

若 $L_{ik} + L_{kj} < L_{ij}$, 则令 $L_{ij} = L_{ik} + L_{kj}$ 。





Floyd 算法

因此，**Floyd算法**是逐步产生 $D_{-1}, D_0, D_1, \dots, D_{n-1}$ 。其递推关系为：
 D_{-1} 为邻接矩阵，

$$D_{-1}[i][j] = \begin{cases} w & \text{若 } \langle v_i, v_j \rangle \in R \text{ 且 } \langle v_i, v_j \rangle \text{ 上的权值为 } w \\ \infty & \text{否则} \end{cases}$$

$$D_k[i][j] = \min(D_{k-1}[i][j], D_{k-1}[i][k] + D_{k-1}[k][j]) \quad 0 \leq i, j, k < n$$

求各顶点间最短路径长度的算法：

初始化 $D[n][n]$ 为邻接矩阵；

for ($k = 0; k < n; k++$) //进行n次试探

 for ($i = 0; i < n; i++$) //对每对 v_i 与 v_j

 for ($j = 0; j < n; j++$)

 if ($D[i][j] > D[i][k] + D[k][j]$) {

$D[i][j] = D[i][k] + D[k][j]$

 }



Floyd 算法

如何在求最短路径长度的同时，记录最短路径？

方法：

设置矩阵 $\text{path}[n][n]$ ， $\text{path}[i][j]$ 记录路径 (v_i, \dots, v_j) 中 v_i 的后继顶点序号。

每当修改 $D_k[i][j]$ 时，令 $\text{path}[i][j] = \text{path}[i][k]$

故 v_i 到 v_j 的最短路径为

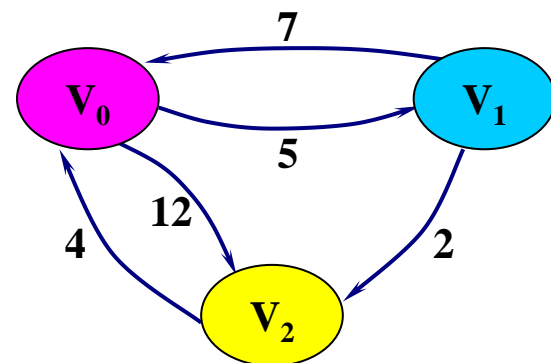
$v_i, \underbrace{\text{path}[i][j]}, \underbrace{v_k \text{ 到 } v_j \text{ 的最短路径}}$
(假设=k) 直到遇 v_j 为止

Floyd 算法

例 7.14

求任意两点间最短路径的过程:

$$D_{-1} = \begin{bmatrix} 0 & 5 & 12 \\ 7 & 0 & 2 \\ 4 & \infty & 0 \end{bmatrix} \quad path = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & -1 & 2 \end{bmatrix}$$



任意两点间试探 v_0 后, 因为 $D_{-1}[2][1] > D_{-1}[2][0] + D_{-1}[0][1] = 4 + 5 = 9$, 故令:

$D_0[2][1] = 9$, $path[2][1] = 0$ (v_0 的序号)。

$$D_0 = \begin{bmatrix} 0 & 5 & 12 \\ 7 & 0 & 2 \\ 4 & 9 & 0 \end{bmatrix} \quad path = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

试探 v_1 后, 因为 $D_0[0][2] > D_0[0][1] + D_0[1][2] = 5 + 2 = 7$, 故令:

$D_1[0][2] = 7$, $path[0][2] = 1$ (v_1 序号)。

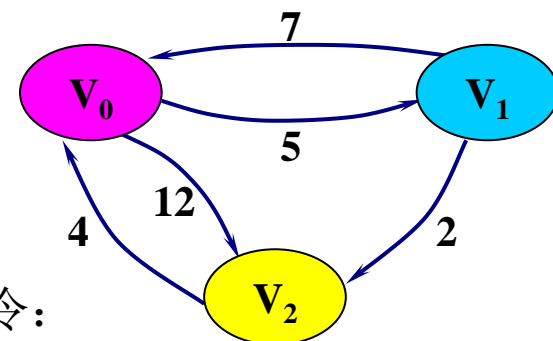
Floyd 算法

$$D_1 = \begin{bmatrix} 0 & 5 & 7 \\ 7 & 0 & 2 \\ 4 & 9 & 0 \end{bmatrix} \quad path = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

试探 v_2 后, 因为 $D_1[1][0] > D_1[1][2] + D_1[2][0] = 2 + 4 = 6$, 故令:

$D_2[1][0] = 6$, $path[1][0] = 2$ (v_2 序号)。

$$D_2 = \begin{bmatrix} 0 & 5 & 7 \\ 6 & 0 & 2 \\ 4 & 9 & 0 \end{bmatrix} \quad path = \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$



输出 (顶点 v_i 用序号 i 代之) 任意两顶点间最短路径及最短路径长度如下:

$(0,0)$	$(0,1)$	$(0,1,2)$	$(1,2,0)$	$(1,1)$	$(1,2)$	$(2,0)$	$(2,0,1)$	$(2,2)$
0	5	7	6	0	2	4	9	0

Floyd算法描述

void Floyd (mgraph G, int n) // 求网G中任意两点间最短路径的Floyd算法

{int i,j,k; int D[][n],path[][n];

// 最短路径长度及最短路径标志矩阵，即path[i][j]存放路径 ($v_i \dots v_j$) 上 v_i 之后继顶点的序号

for (i=0;i<n;i++) // 初始化

for (j=0;j<n;j++)

{ **if** (G.A[i][j]<max) **path[i][j]=j**; // 若 $\langle v_i, v_j \rangle \in R$, v_i 当前后继为 v_j

else path[i][j]=-1; // 否则为-1

D[i][j]=G.A[i][j]; }

for (k=0;k<n;k++) // 进行n次试探

for (i=0;i<n;i++) // 对任意的 v_i , v_j

for (j=0;j<n;j++)

if (D[i][j]>D[i][k]+D[k][j];

{ D[i][j]=D[i][k]+D[k][j]; // 取小者

Path[i][j]=path[i][k]; // 改 v_i 的后继

}

Floyd算法描述

```
for (i=0;i<n;i++)  
  for (j=0;j<n;j++)  
  { printf ("\n %d", D[i][j]); // 输出 $v_i$ 到 $v_j$ 的最短路径长度  
    k=path[i][j]; // 取路径上 $v_i$ 的后继 $v_k$   
    if (k==-1)  
      printf ("%d to %d no path \n",i,j); //  $v_i$ 到 $v_j$ 路径不存在  
    else  
    { printf("(%d",i); // 输出 $v_i$ 的序号i  
      while (k!=j) // k不等于路径终点j时  
      { printf(",%d",k); // 输出k  
        k=path[k][j]; // 求路径上下一顶点序号  
      }  
      printf ("%d) \n",j); // 输出路径终点序号  
    }  
  }  
}
```

输出最短路径



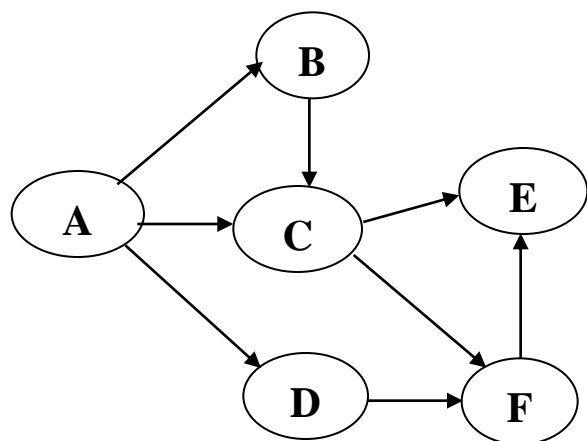
7.6 有向无环图的应用

- **DAG**是描述工程（或系统）进行过程的工具。
- 通常一个大的工程由若干个子工程来构成，子工程又可分为若干个更小的工程，且各子工程之间有一定的约束关系，如某些子工程开始必须等到另一些子工程的结束。

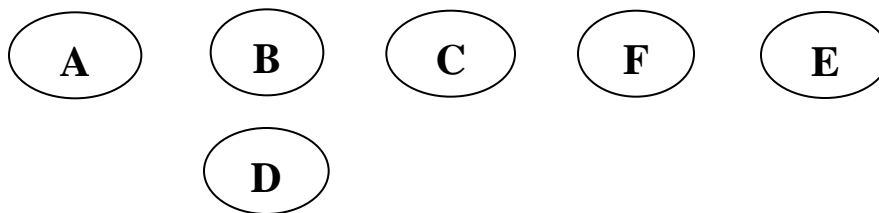
7.6 有向无环图的应用

有向无环图：DAG(Directed Acyclic Graph)

例 设任务T由A、B、C、D、E、F这6个子任务构成，这些子任务之间的优先关系如下：



给出这些子任务的执行次序。



AOV(Activity On Vertex Network)网：由顶点表示活动，用弧表示活动间优先关系的有向图。

7.6 有向无环图的应用

- 对于一项工程而言，人们关心的一般有两点：
 - 工程能否顺序进行；
 - 工程完成所需的时间。

图的拓扑排序问题
(**Topological Sort**)

图的关键路径问题
(**Critical Path**)

若AOV网中出现环，则其表示的任务（工程）无法顺利进行（互相等待）。

7.6 有向无环图的应用

1. 拓扑排序 (Topological Sort)

将**AOV**网中所有顶点在不违反优先关系的前提下排成线性序列

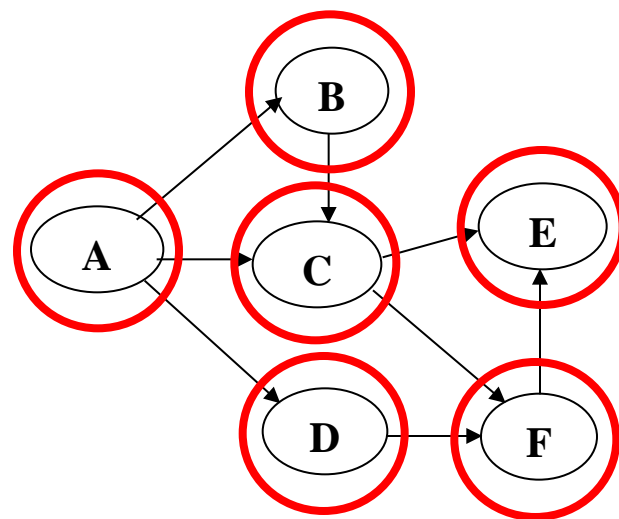
(1) 拓扑排序方法

看这个例子，怎么做？

- ① 在**AOV**网中任选一个无前驱（入度为0）的顶点输出；
- ② 删去输出过的顶点和由它发出的所有弧；
- ③ 重复①、②，直到网中不存在无前驱的顶点为止。

结果：若全部顶点都输出，则**AOV**网是一个**DAG**，得到的顶点序列是一个拓扑序列，否则该图存在环。

说明：拓扑序列不唯一，因为某时刻可能有多个无前驱的顶点。



一种拓扑序列：

A, D, B, C, F, E



拓扑排序

(2) 拓扑排序算法

采用什么数据结构表示**AOV**网?

AOV网用十字链表表示, 便于求入度。

算法:

将所有入度为0的顶点进栈;

while (栈非空) {

 出栈送 v_j , 输出 v_j 并计数输出个数;

 将 v_j 的所有直接后继入度减1, 并将入度变为0的顶点进栈;

}

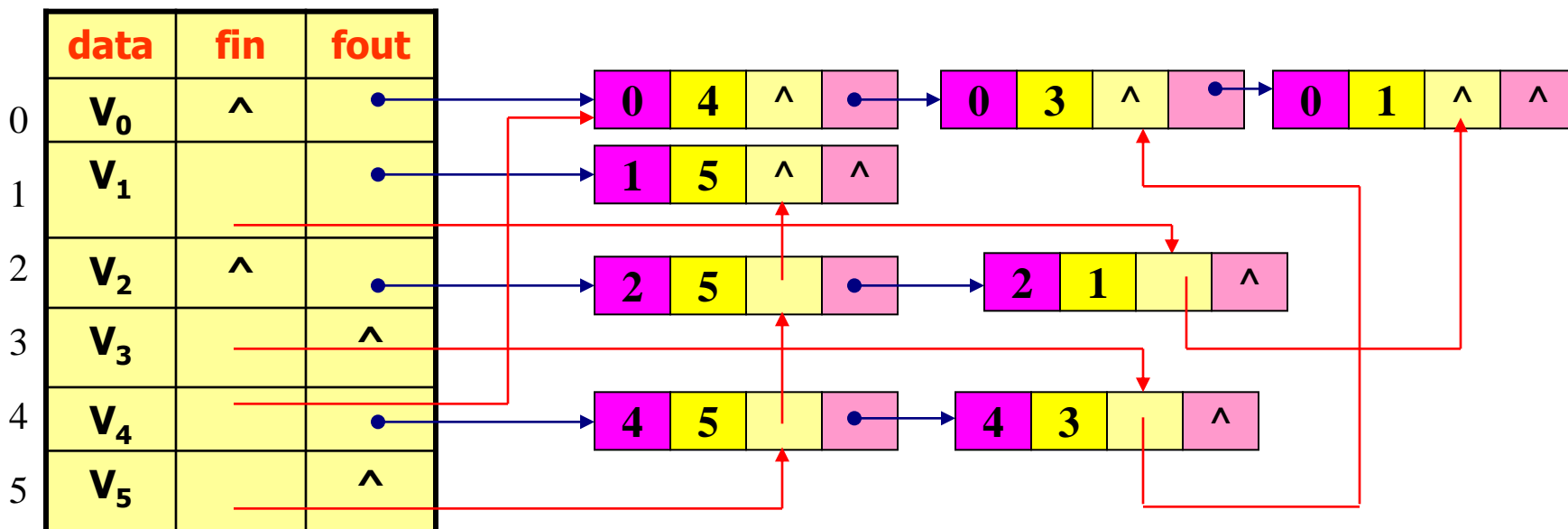
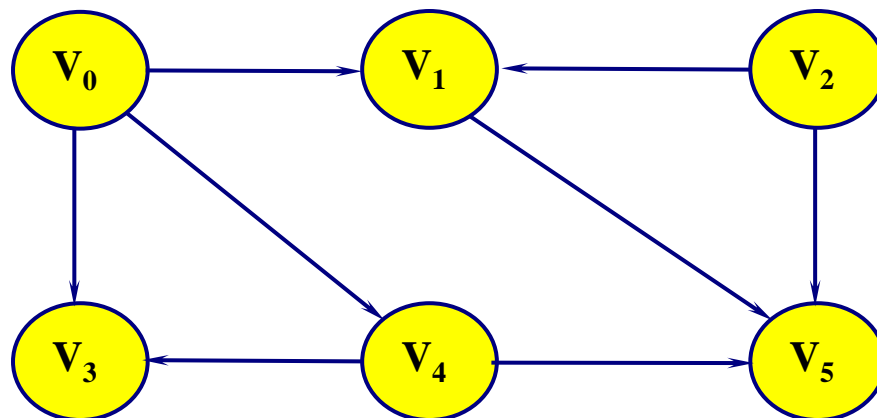
若输出顶点数为AOV网中的顶点数 n , 则网中不存在环, 输出序列为拓扑序列, 否则网中存在环;

这里采用了栈, 采用队列实现是否可以?

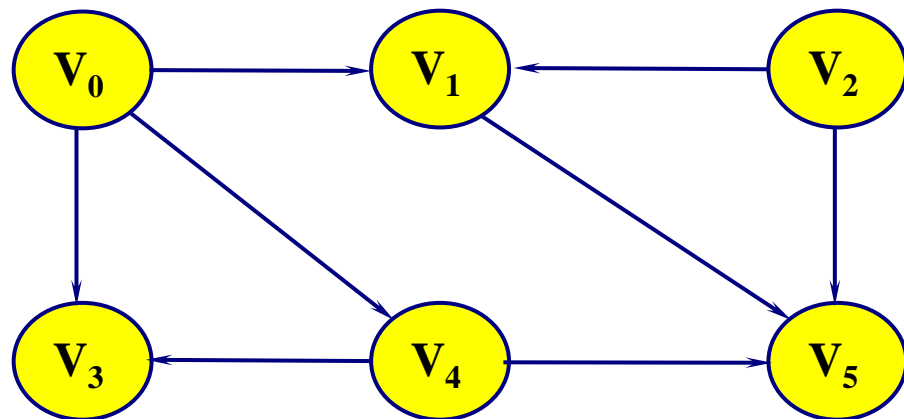
拓扑排序

例 7.16 设AOV网如右图：

十字链表如下：

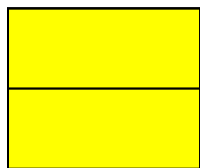


拓扑排序



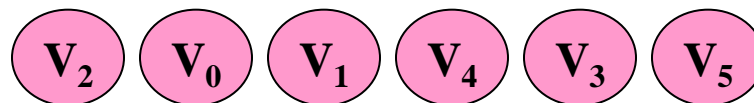
入度表:

0	0	0	0	0	0
0	1	2	3	4	5



栈

输出顶点:



count=6

不存在环

7.6 有向无环图的应用

2. 关键路径 (Critical Path, CP)

(1) AOE网： 以边表示活动的网

AOE: Activity On Edge

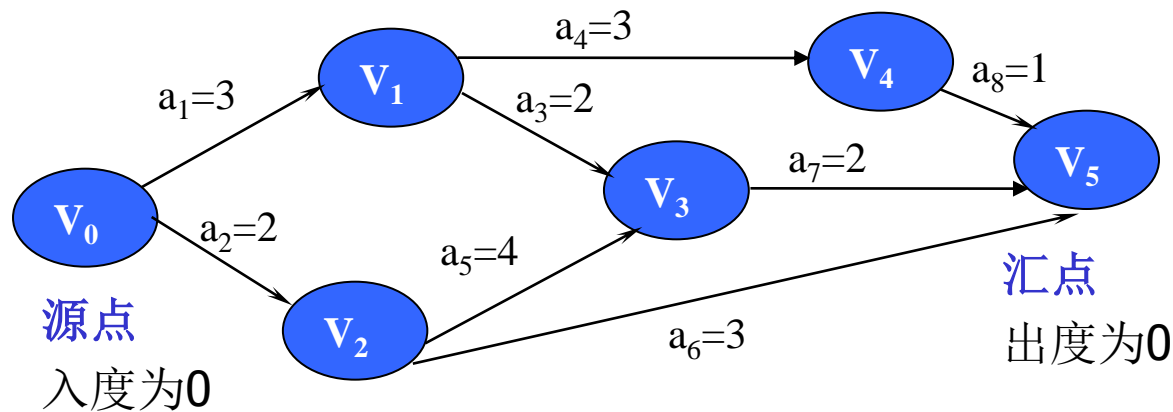
特点：顶点表示“状态”(或事件)，

边表示“活动”(如权值表示活动的持续时间)

例 AOE网：

研究的2个问题：

- ① 完成该工程至少需要多长时间？
- ② 哪些活动是影响整个工程进度的关键？



关键路径

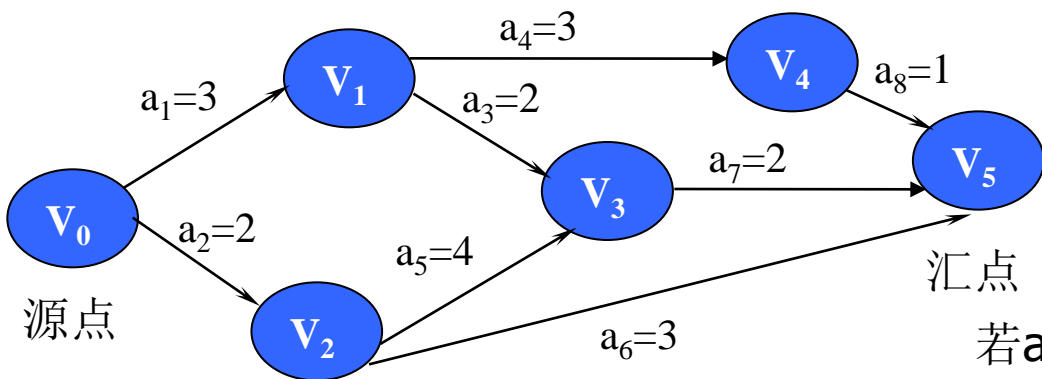
(2) 关键路径CP

从源点到汇点最长的路径，其长度就是工程所需时间

关键活动： CP上的所有活动

关键活动的重要性：

- ✓ 若关键活动延期完成，则影响整个工程进度；
- ✓ 提高关键活动的速度通常可缩短整个工程的工期。



左图的关键路径：(V₀, V₂, V₃, V₅)

长度=2+4+2=8

关键活动：a₂, a₅, a₇

若a₅由4天减为3天，则整个工程可缩短1天。

若a₅变为2，整个工程可否在6天内完成？不能！因为若a₅=2，则关键路径就变了。



关键路径

若存在多条关键路径，则需同时提高这些关键路径上某些关键活动的速度，才能缩短整个工程的工期。

提高非关键活动的速度是不能加快整个工程进度的。

为了求出关键活动，首先定义几个变量：

事件的最早发生时间 $Ve(j)$

事件的最晚发生时间 $Vl(i)$

活动（边）的最早开始时间 $Ee(i)$

活动（边）的最晚开始时间 $El(i)$



关键路径

① 事件的最早发生时间**Ve(j)**: **V_j**的最早发生时间

从源点V₀到V_j的最长路径长度

决定了所有以V_j为起点的弧所表示的活动能开始的最早时间

Ve(j)的计算方法（递推）：

$$Ve(0) = 0$$

$$Ve(j) = \max_i \{Ve(i) + \underbrace{w_{ij}}_{\text{弧} \langle V_i, V_j \rangle \text{的权值}}\}, 1 \leq j \leq n-1, n \text{为顶点数}$$

即从所有以V_j为终点的弧的集合中，找一个**Ve(i)+w_{ij}**最大的作为**Ve(j)**

显然，必须在V_j的所有前驱顶点的**Ve**计算出来后，才能求出**Ve(j)**。

因此，需对**AOE**网进行拓扑排序，按拓扑序列逐个求出各顶点事件的**最早发生时间**。

关键路径

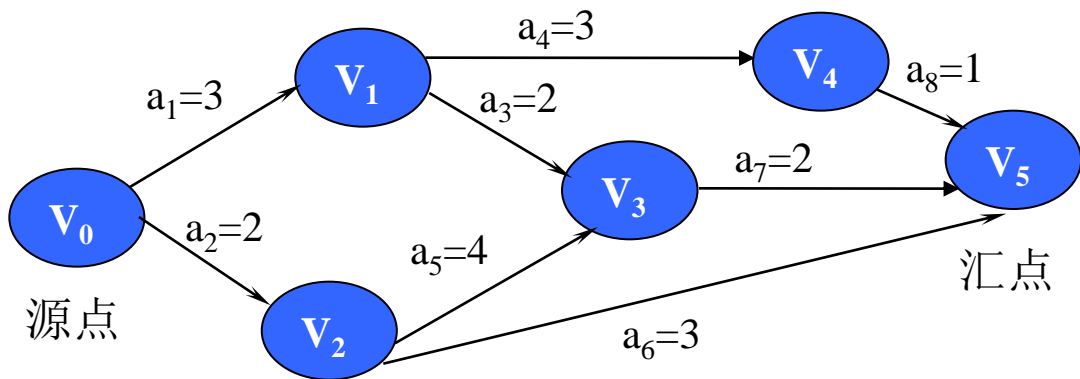
$$Ve(0) = 0$$

$$Ve(1) = 3$$

$$Ve(2) = 2$$

$$Ve(3) = 6$$

.





关键路径

② 事件的最晚发生时间 $VI(i)$

在不推迟整个工程完成日期的前提下，事件 V_i 所允许的最晚发生时间 $VI(i)$ 的计算公式（从汇点 V_{n-1} 开始向源点方向递推）：

$$VI(n-1) = Ve(n-1)$$

$$VI(i) = \min_j \{VI(j) - w_{ij}\}, 0 \leq i \leq n-2, n \text{ 为顶点数}$$

即从所有以 V_i 为起点的弧的集合中，找一个 $VI(j)-w_{ij}$ 最小的作为 $VI(i)$

因此，需对AOE网进行逆拓扑排序，然后按逆拓扑序列逐个求出各顶点事件的最晚发生时间 VI 。



关键路径

③ 活动（边）的最早开始时间**Ee(i)**

设活动 a_i 由弧 $\langle V_j, V_k \rangle$ 表示，则 $Ee(i) = Ve(j)$

即活动 a_i 的最早开始时间 = 事件 V_j 的最早发生时间

④ 活动（边）的最晚开始时间**El(i)**

设活动 a_i 由弧 $\langle V_j, V_k \rangle$ 表示，则 $El(i) = Vl(k) - w_{jk}$

对于活动 a_i ，若 **$El(i) = Ee(i)$** ，则 a_i 是关键活动。不能延误！



关键路径

(3) 求关键路径的算法

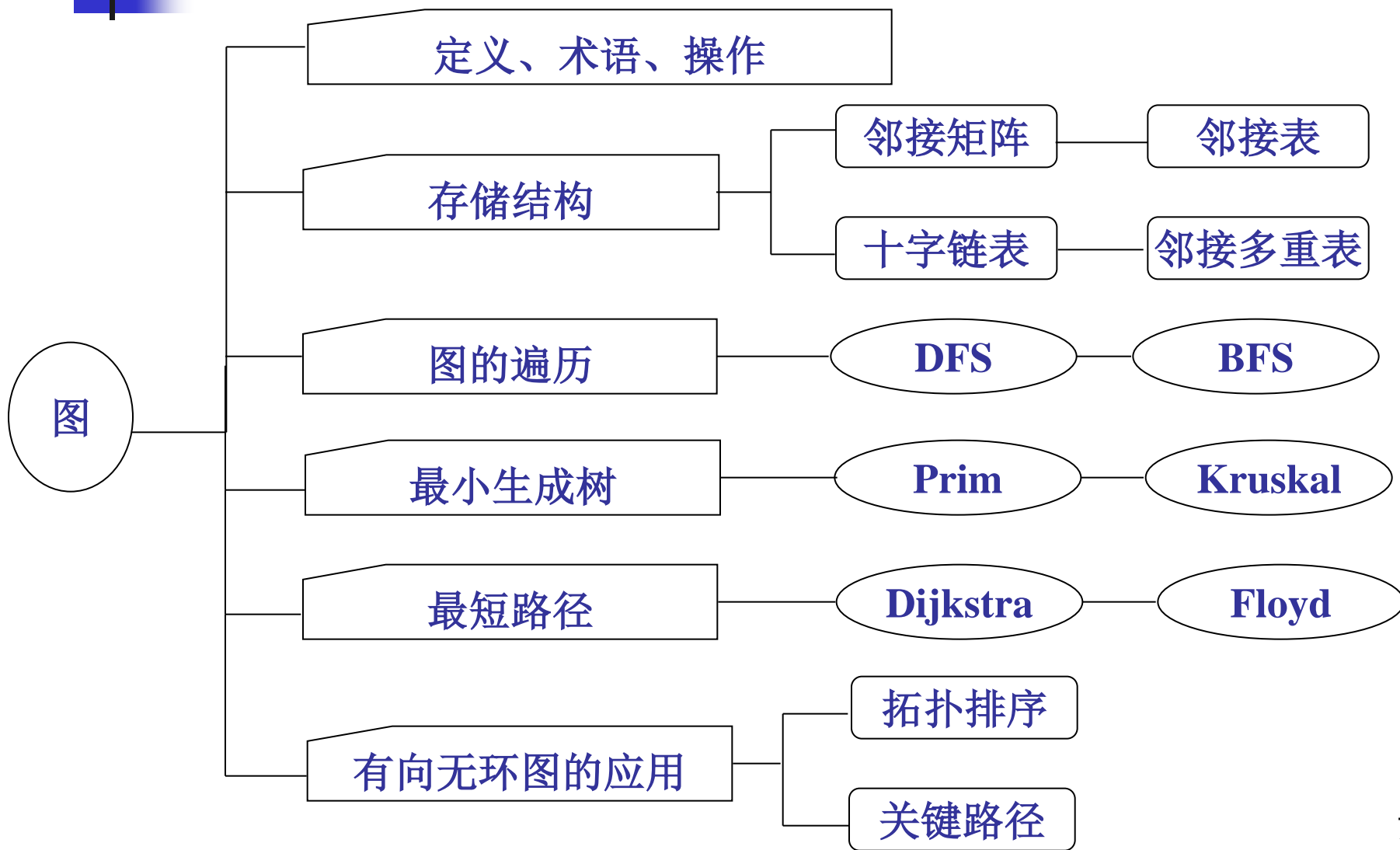
AOE 网用什么数据结构表示合适？怎么求关键路径？

AOE 网用十字链表表示

算法步骤：

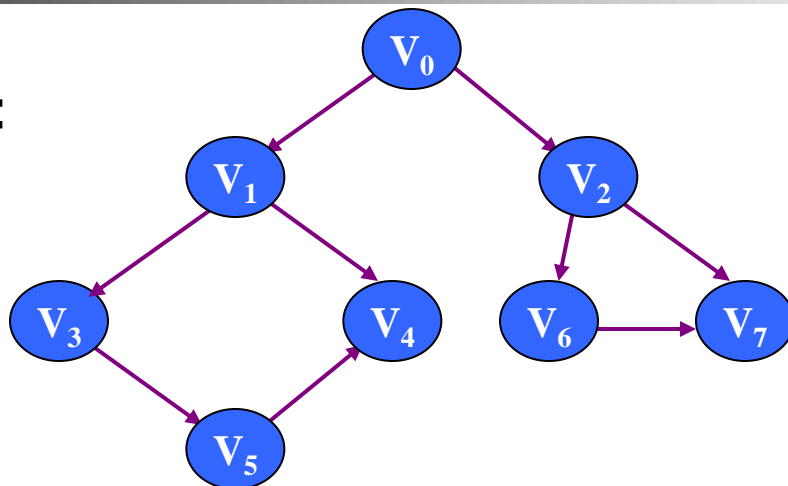
- ① 从源点 V_0 出发, 令 $Ve(0)=0$, 按拓扑排序的序列依次计算其余顶点的 $Ve(j)$ ($1 \leq j \leq n-1$);
- ② 从汇点 V_{n-1} 出发, 令 $VI(n-1)=Ve(n-1)$, 按逆拓扑序列求其余顶点的 $VI(i)$ ($0 \leq i \leq n-2$);
- ③ 根据各顶点的 Ve 和 VI , 求每个活动 a_i 的 $Ee(i)$ 和 $El(i)$ 。若 $Ee(i)=El(i)$, 则 a_i 为关键活动。依次输出这些关键活动可得关键路径;

7.7 小结



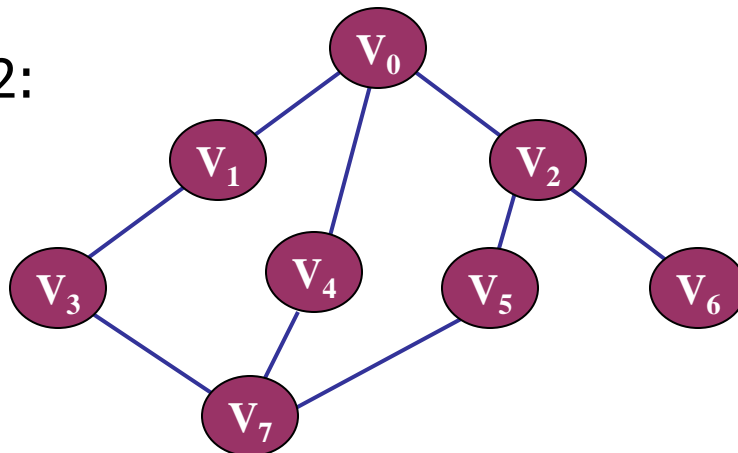
第7章 作业

1. 设有向图G1:



试构造出G1的“邻接矩阵”、“邻接表”和“十字链表”结构。

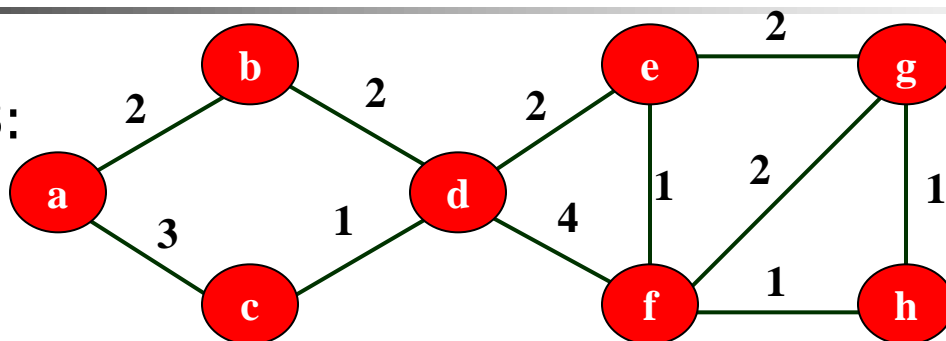
2. 设无向图G2:



写出从顶点 V_0 出发，按“DFS”和“BFS”方法遍历G2所得到的顶点序列。 80

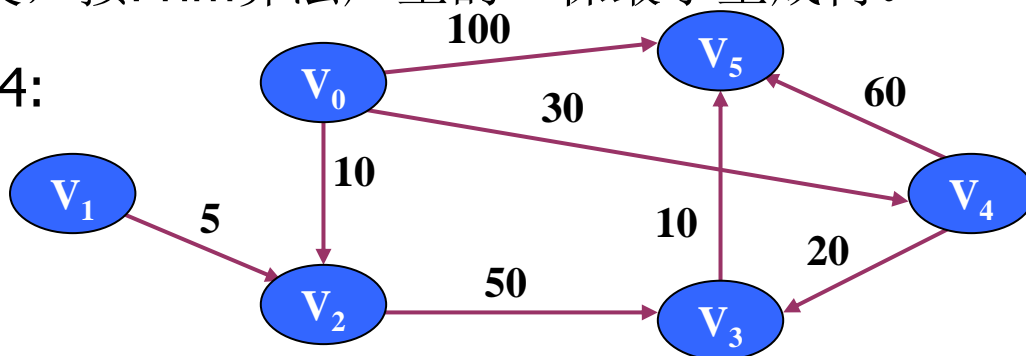
第7章 作业

3. 设无向网G3:



画出从a出发, 按Prim算法产生的一棵最小生成树。

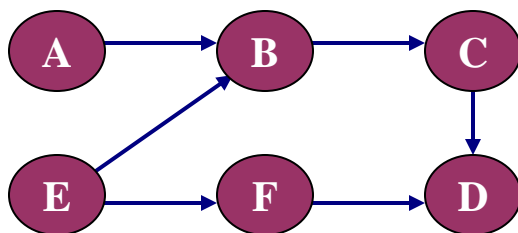
4. 设有向网G4:



用Floyd算法求出G4中任意两点间最短路径长度的矩阵,即:

$D_{-1}, D_0, D_1, D_2, D_3, D_4, D_5 = ?$

5. 设有向图G5:



写出G5的所有拓扑序列。