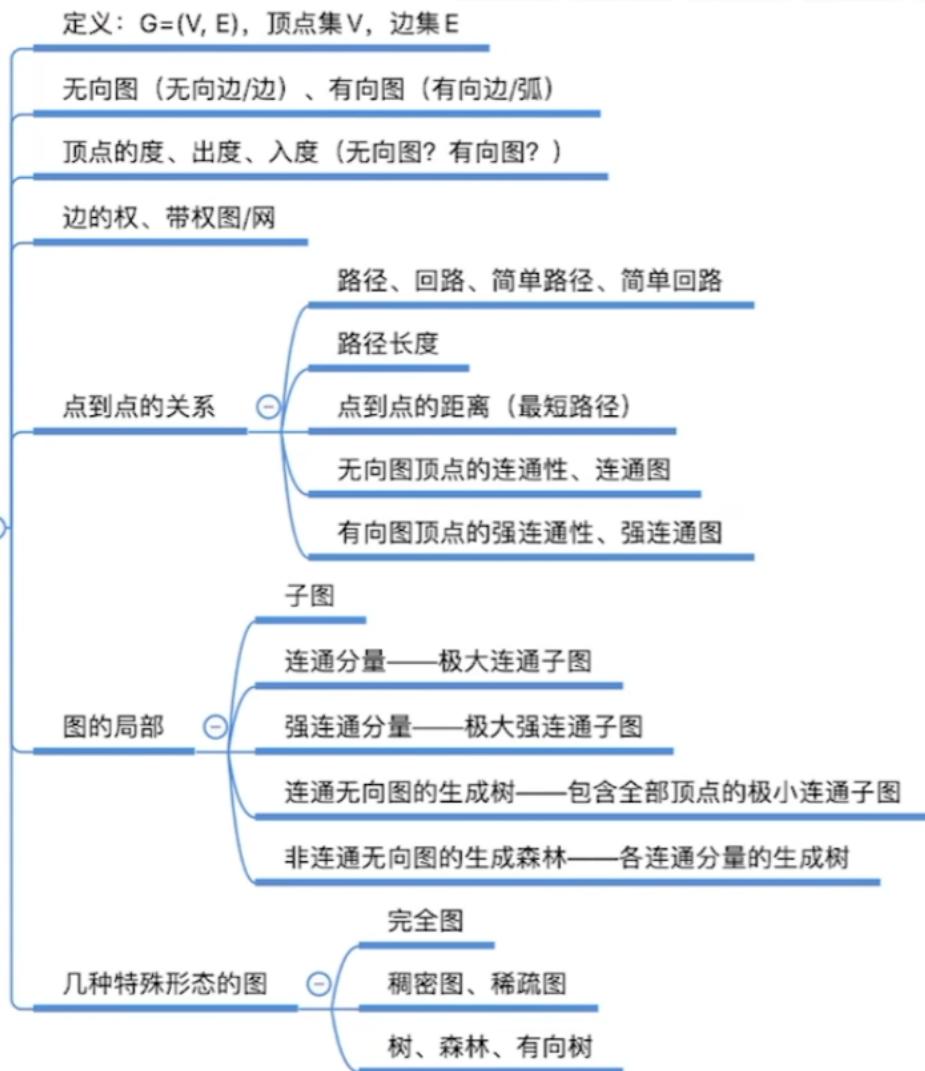


图的基本概念

图的基本概念



常见考点:

对于n个顶点的无向图G

- 所有顶点的度之和= $2|E|$
- 若G是连通图, 则最少有 $n-1$ 条边 (树), 若 $|E|>n-1$, 则一定有回路
- 若G是非连通图, 则最多可能有 c_{n-1}^2 条边
- 无向完全图共有 C_n^2 条边

对于n个顶点的有向图G

- 所有顶点的出度之和=入度之和= $|E|$
- 所有顶点的度之和= $2|E|$
- 若G是强连通图, 则最少有n条边 (形成回路)
- 有向完全图共有 $2C_n^2$ 条边

图的定义

$G : Graph$

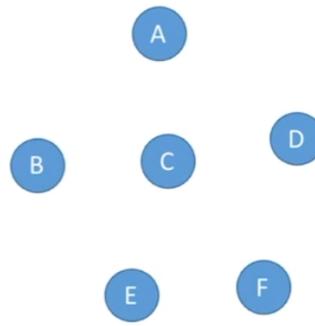
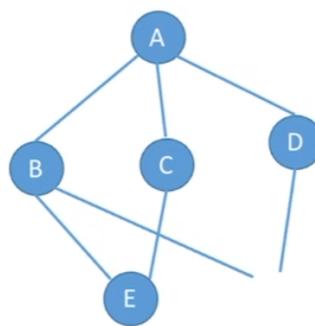
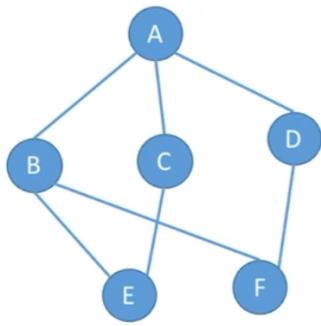
$V : Vertex$

$E : Edge$

图G由顶点集V和边集E组成，记为 $G=(V,E)$ ，其中 $V(G)$ 表示图G中顶点的有限非空集； $E(G)$ 表示图G中顶点之间的关系（边）集合。若 $V=\{v_1, v_2, \dots, v_n\}$ ，则用 $|V|$ 表示图G中顶点的个数，也称图G的阶，用 $|E|$ 表示图G中边的条数。

注意：线性表可以是空表，树可以是空树，但图不可以是空，即V一定是非空集

$$E = \{(u, v) | u \in V, v \in V\}$$



图逻辑结构的应用

- 铁路
- 公路交通地图
- 微信好友关系 边是没有方向的
- 微博粉丝关系 边是有方向的

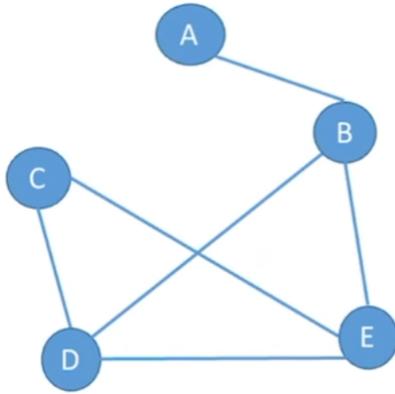
无向图 有向图

若E是无向边（简称边）的有限集合时，则图G为无向图。边是顶点的无序对，记为 (v,w) 或 (w,v) ，因为 $(v,w)=(w,v)$ ，其中 v, w 是顶点。可以说顶点 w 和顶点 v 互为邻接点。边 (v,w) 依附于顶点 w 和 v ，或者说边 (v,w) 和顶点 v, w 相关联。

$$G_2 = (V_2, E_2)$$

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

$$E_2 = \{(A, B), (B, D), (B, E), (C, D), (C, E), (D, E)\}$$

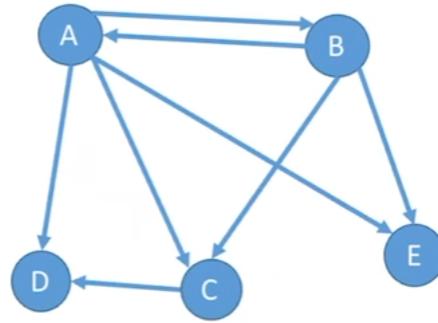


若 E 是有向边（也称弧）的有限集合时，则图 G 为有向图。弧是顶点的有序对，记为 $\langle v, w \rangle$ ，其中 v 、 w 是顶点， v 称为弧尾， w 称为弧头， $\langle v, w \rangle$ 称为从顶点 v 到顶点 w 的弧，也称 v 邻接到 w ，或 w 邻接自 v 。 $\langle v, w \rangle \neq \langle w, v \rangle$

$$G_1 = \{V_1, E_1\}$$

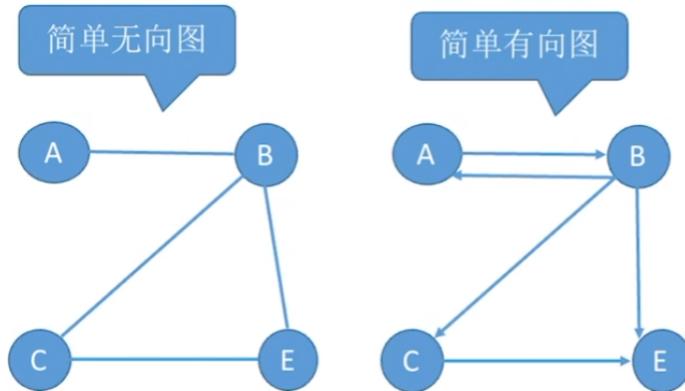
$$V_1 = \{A, B, C, D, E\}$$

$$E_1 = \{\langle A, B \rangle, \langle A, C \rangle, \langle A, D \rangle, \langle A, E \rangle, \langle B, A \rangle, \langle B, C \rangle, \langle B, E \rangle, \langle C, D \rangle\}$$



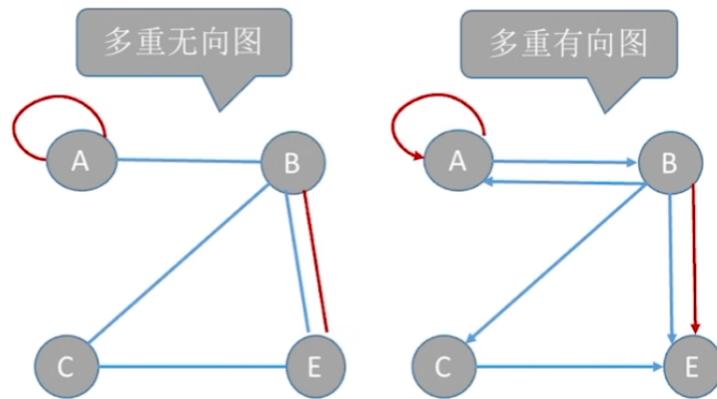
简单图 多重图

简单图



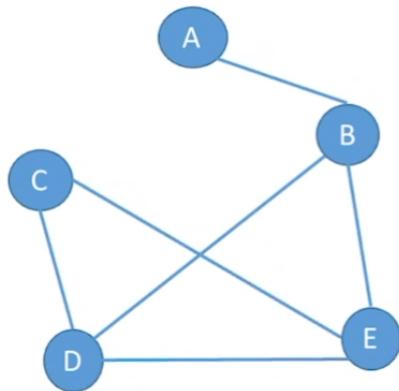
1. 不存在重复边
2. 不存在顶点到自身的边

多重图

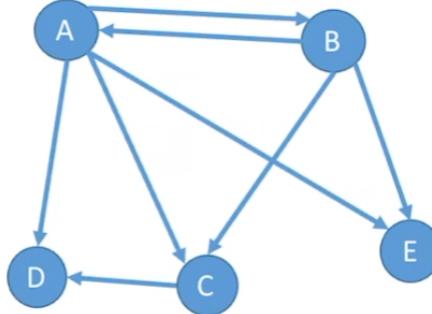


图G中某两个结点之间的边数多于1条，又允许顶点通过同一条边和自己关联，则G为多重图

顶点的度、入度、出度



对于无向图：顶点v的度是指依附于该顶点的边的条数，记为TD(v)。



对于有向图：

入度是以顶点v为终点的有向边的数目，记为ID(v)

出度是以顶点v为起点的有向边的数目，记为OD(v)

顶点v的度等于其入度和出度之和，即 $TD(v)=ID(v)+OD(v)$

顶点-顶点的关系描述

路径

顶点 v_p 到顶点 v_q 之间的一条路径是指顶点序列, $v_p, v_{i_1}, v_{i_2}, \dots, v_{i_m}, v_q$

顶点之间可能存在路径

有向图的路径也是有向的

回路

第一个顶点和最后一个顶点相同的路径称为回路或环

简单路径

在路径序列中, 顶点不重复出现的路径称为简单路径

简单回路

除第一个顶点和最后一个顶点外, 其余顶点不重复出现的回路称为简单回路

路径长度

路径上边的长度

点到点的距离

从顶点 u 出发到顶点 v 的最短路径若存在, 则此路径的长度称为从 u 到 v 的距离。

若从 u 到 v 根本不存在路径, 则记该距离为无穷(∞)

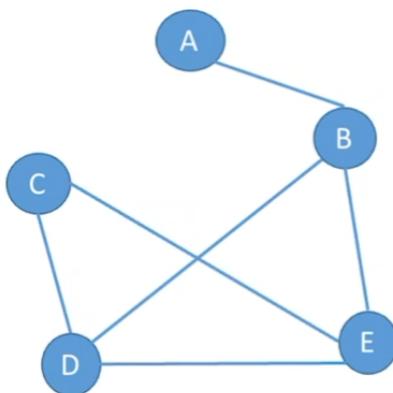
连通

无向图中, 若从顶点 v 到顶点 w 有路径存在, 则称 v 和 w 是连通的。

强连通

有向图中, 若从顶点 v 到顶点 w 和从顶点 w 到顶点 v 之间都有路径, 则称这两个顶点是强连通的。

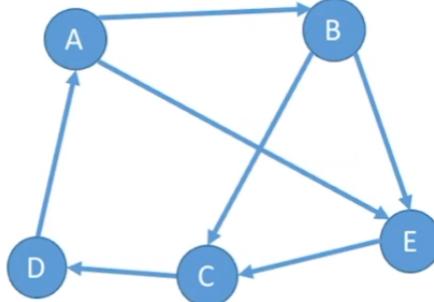
连通图、强连通图



若图G中任意两个顶点都是连通的, 则称图G为连通图, 否则称为非连通图。

常见考点：

对于n个顶点的无向图G



若G是连通图，则最少有 $n-1$ 条边

若G是非连通图，则最多可能有 C_{n-1}^2 条边

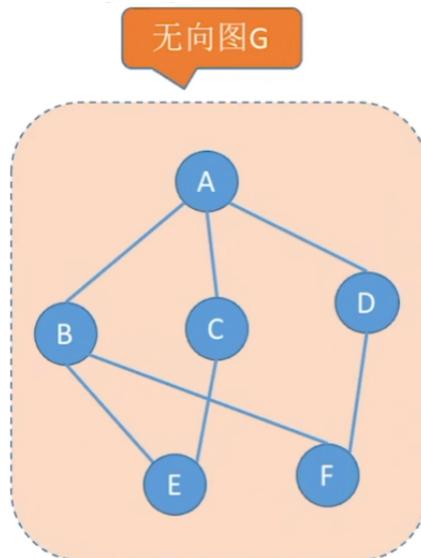
若图中任何一对顶点都是强连通的，则称此图为强连通图。

常见考点：

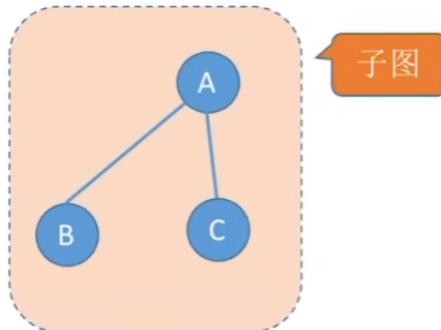
对于n个顶点的有向图G

若G是强连通图，则最少有n条边（形成回路）

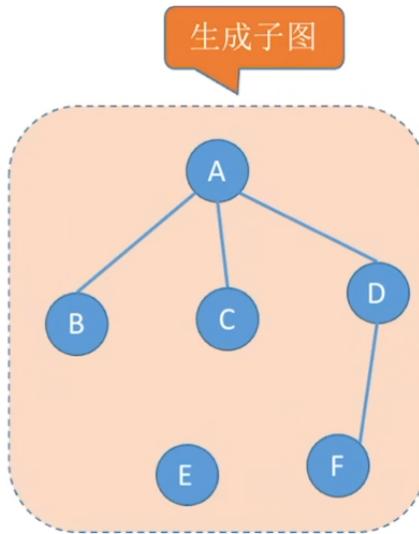
研究图的局部子图



设有两个图 $G = (V, E)$ 和 $G' = (V', E')$, 若 V' 是 V 的子集, 且 E' 是 E 的子集, 则称 G' 是 G 的子图

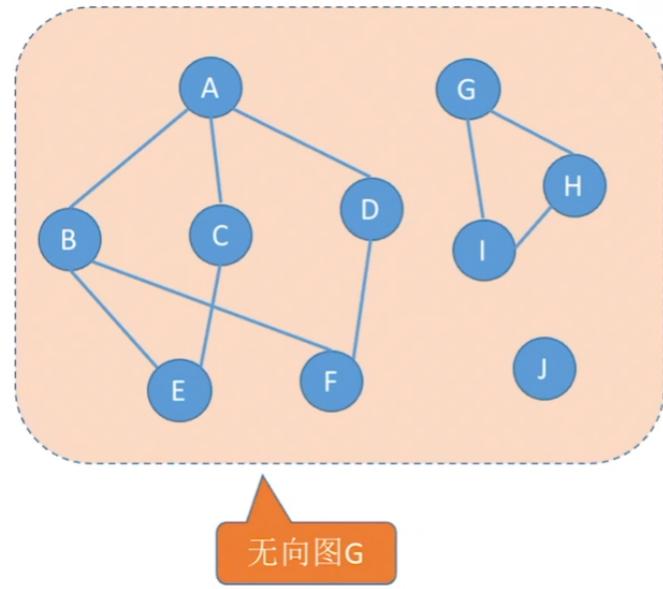


并非任意挑几个点、几条边都能构成子图



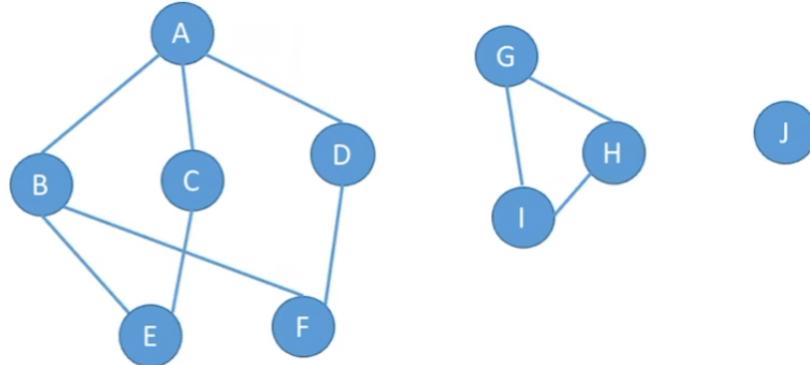
若有满足 $V(G') = V(G)$ 的子图 G' , 则称其为 G 的生成子图

连通分量



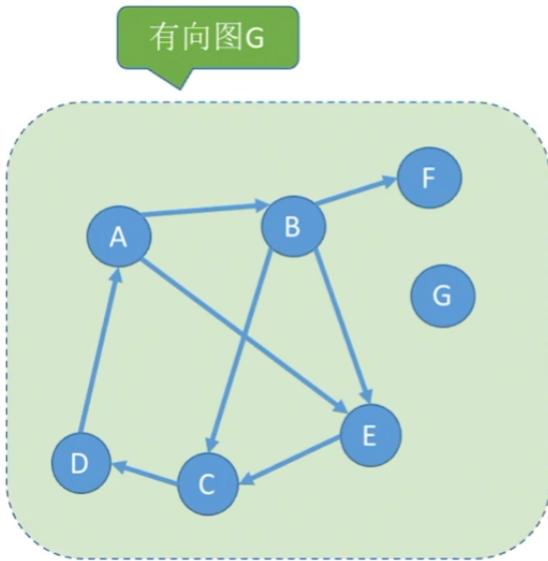
无向图中的极大连通子图称为连通分量。

子图必须连通，且包含尽可能多的顶点和边



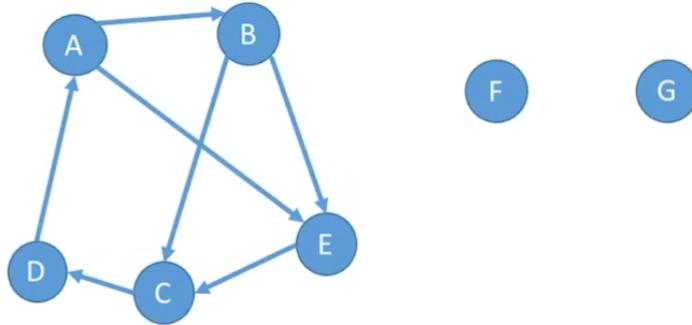
G的三个连通分量

强连通分量



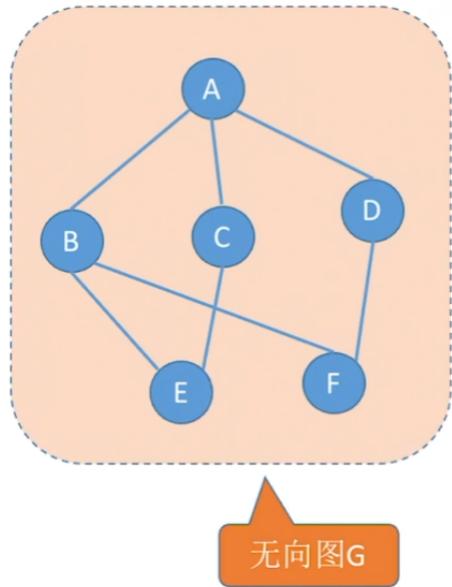
有向图中的极大强连通子图称为有向图的强连通分量

子图必须强连通，同时保留尽可能多的边



G的三个强连通分量

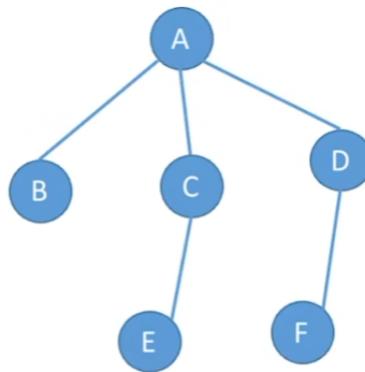
生成树



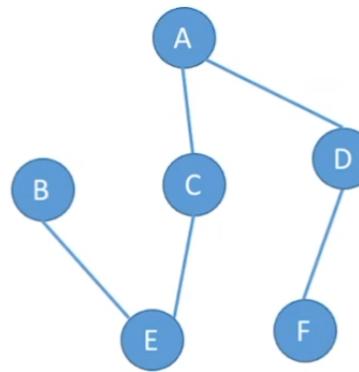
连通图的生成树是包含图中全部顶点的一个极小连通子图。

边尽可能的少，但要保持连通

若图中顶点数为n，则它的生成树含有n-1条边。对生成树而言，若砍去它的一条边，则会变成非连通图，若加上一条边则会形成一个回路。



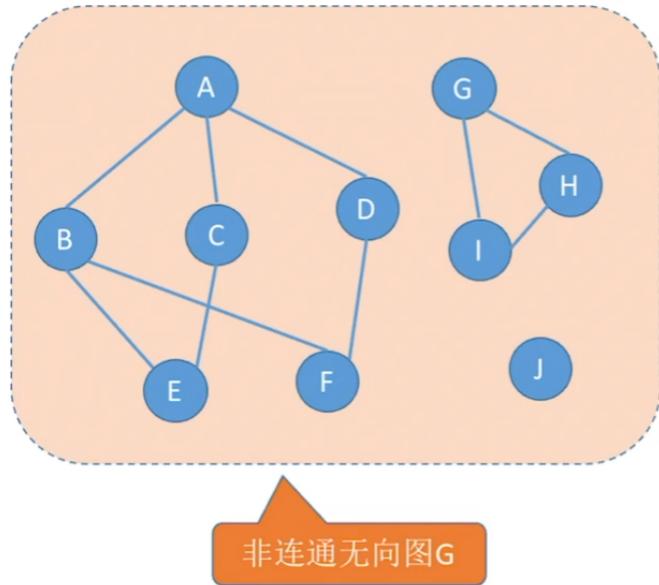
G的生成树1



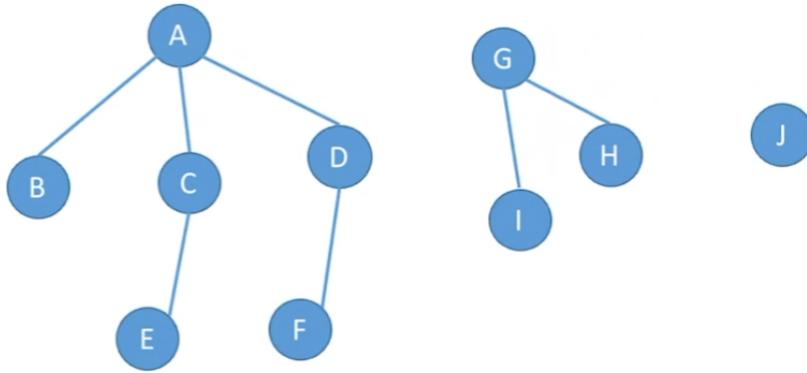
G的生成树2

.....

生成森林

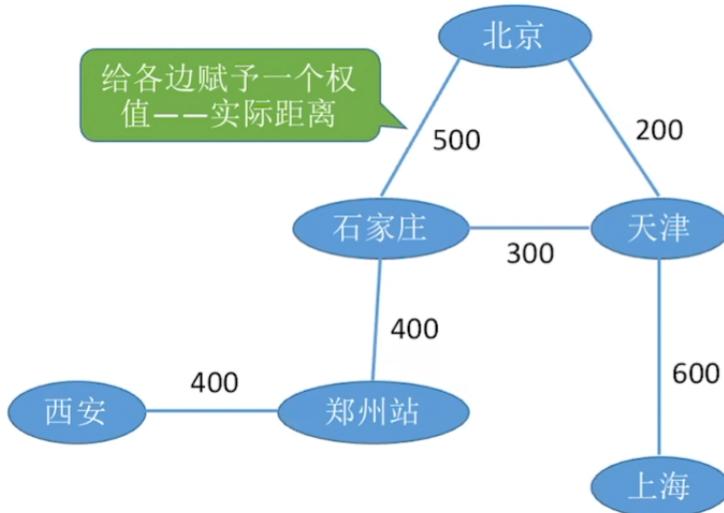


在非连通图中，连通分量的生成树构成了非连通图的生成森林。

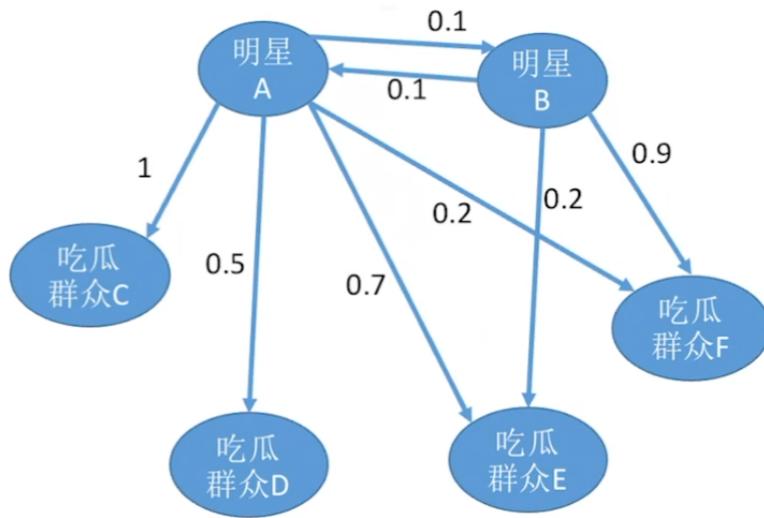


G的连通分量 -> G的生成森林

边的权、带权图/网



给各边赋予一个权值——实际距离



给各边赋予一个数值——转发概率

边的权

在一个图中，每条边都可以标上具有某种含义的值，该数值称为该边的权值。

带权图/网

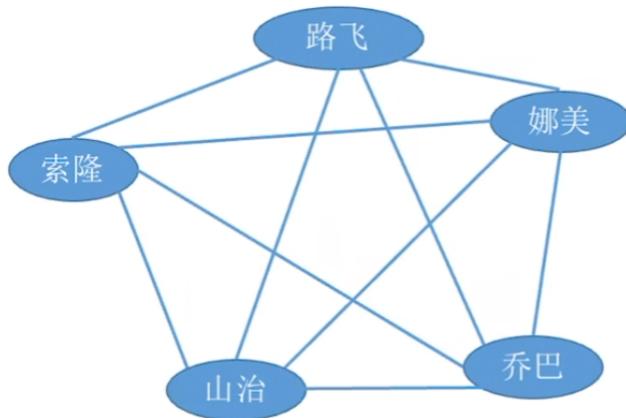
边上带有权值的图称为带权图，也称网。

带权路径长度

当图是带权图时，一条路径上所有边的权值之和，称为该路径的带权路径长度

几种特殊形态的图

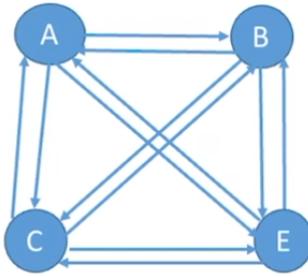
无向完全图



无向图中任意两个顶点都存在边

若无向图的顶点数 $|V| = n$, 则 $|E| \in [0, C_n^2] = [0, n(n - 1)/2]$

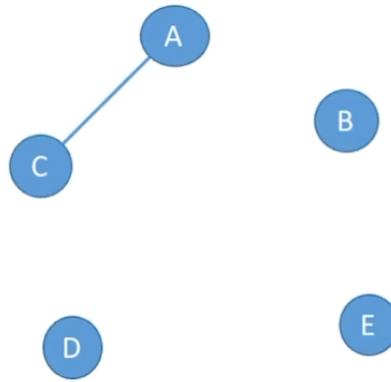
有向完全图



有向图中任意两个顶点之间都存在方向相反的两条弧

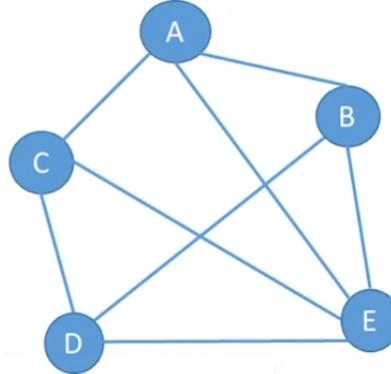
若有向图的顶点数 $|V| = n$, 则 $|E| \in [0, 2C_n^2] = [0, n(n - 1)]$

稀疏图



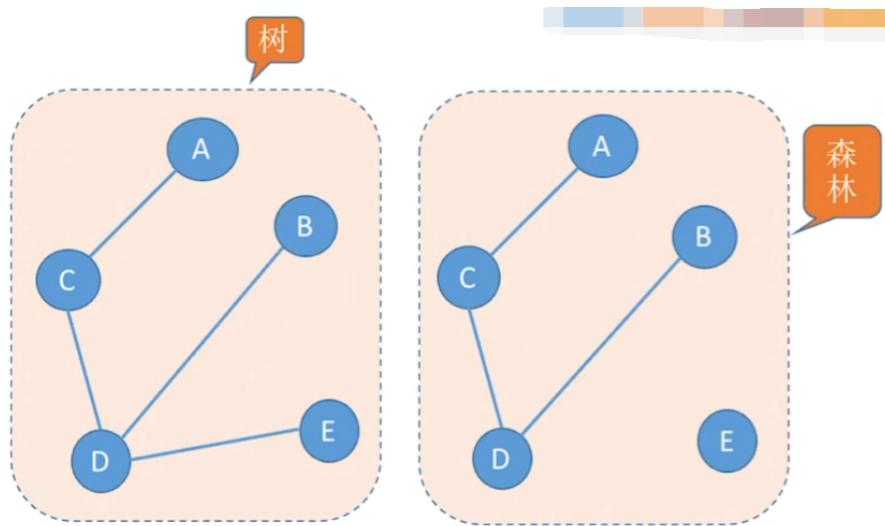
边数很少的图称为稀疏图

稠密图



反之称为稠密图

树

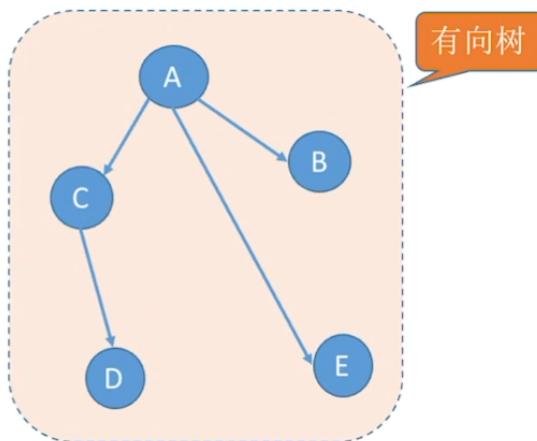


不存在回路，且连通的无向图

n个顶点的树，必有n-1条边。

常见考点：n个顶点的图，若 $|E|>n-1$ ，则一定有回路

有向树



一个顶点的入度为0，其余顶点的入度均为1的有向图，称为有向树。

图的存储

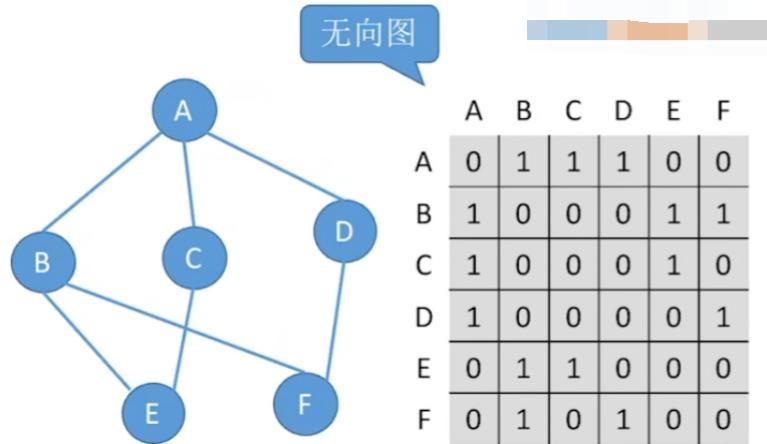
邻接矩阵法

```
#define MaxVertexNum 100 //顶点数目的最大值
typedef struct{
    char Vex[MaxVertexNum]; //顶点表
    int Edge[MaxVertexNum][MaxVertexNum]; //邻接矩阵，边表
    int vexnum, arcnum; //图的当前顶点数和边数/弧数
}MGraph;
```

只要确定了顶点编号，图的邻接矩阵表示方式唯一

数组实现的顺序存储，空间复杂度高，不适合存储稀疏图

- 如何计算指定顶点的度、入度、出度（分无向图、有向图来考虑）？时间复杂度如何？
- 如何找到与顶点相邻的边（入边、出边）？时间复杂度如何？
- 如何存储带权图？
- 空间复杂度 $O(|V|^2)$, 适合存储稠密图
- 无向图的邻接矩阵为对称矩阵，如何压缩存储？
- 设图G的邻接矩阵为A（矩阵元素为0/1），则 $A_{n \times n}$ 的元素 $A_{n \times n}[i][j]$ 等于由顶点i到顶点j的长度为n的路径的数目



顶点中可以存更复杂的信息

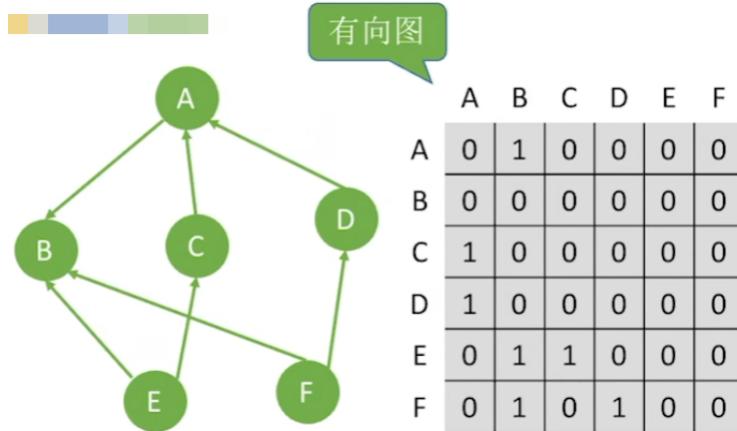
可以用bool型或枚举型变量表示边

结点数为 n 的图 $G = (V, E)$ 的邻接矩阵 A 是 $n \times n$ 的。将 G 的顶点编号为 v_1, v_2, \dots, v_n ，则

$$A[i][j] = \begin{cases} 1, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \text{ 是 } E(G) \text{ 中的边} \\ 0, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \text{ 不是 } E(G) \text{ 中的边} \end{cases}$$

思考：如何求顶点的度、入度、出度？

第*i*个结点的度 = 第*i*行（或第*i*列）的非零元素个数



第*i*个结点的出度 = 第*i*行的非零元素个数

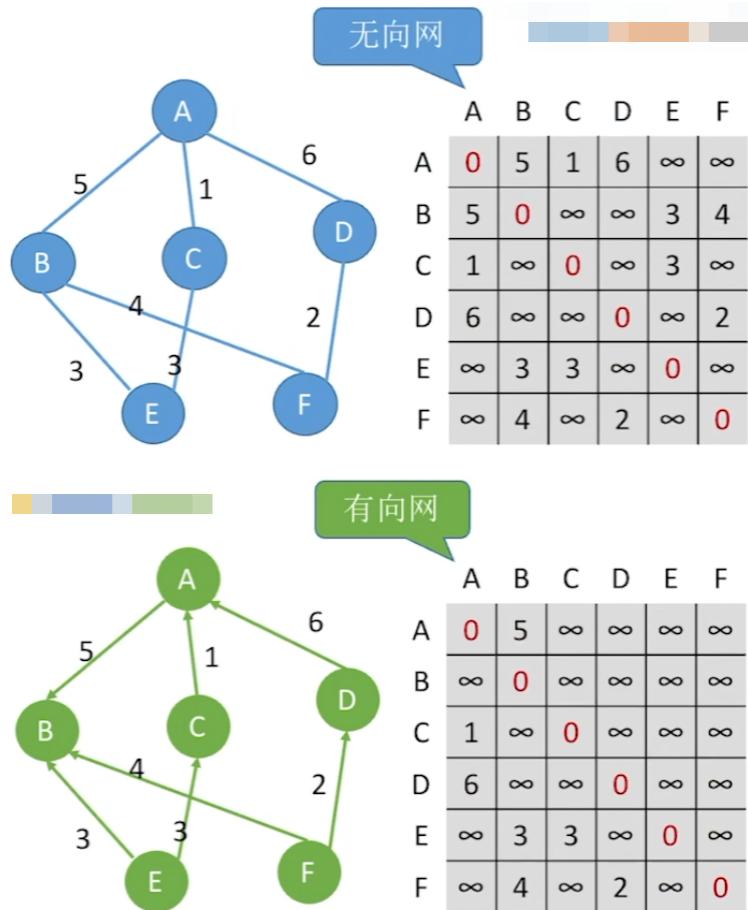
第*i*个结点的入度 = 第*i*列的非零元素个数

第*i*个结点的度 = 第*i*行、第*i*列的非零元素个数之和

邻接矩阵法求顶点的度/出度/入度的时间复杂度为 $O(|V|)$

邻接矩阵法存储带权图 (网)

```
#define MaxVertexNum 100 //顶点数目的最大值
#define INFINITY 最大的int值 //宏定义常量“无穷”，可用int的上限值表示“无穷”
typedef struct{
    char Vex[MaxVertexNum]; //顶点表
    int Edge[MaxVertexNum][MaxVertexNum]; //邻接矩阵，边表
    int vexnum, arcnum; //图的当前顶点数和边数/弧数
}MGraph;
```



邻接矩阵法的性能分析

空间复杂度： $O(|V|^2)$ ，只和顶点数相关，和实际的边数无关

适合用于存储稠密图

无向图的邻接矩阵是对称矩阵，可以压缩存储（只存储上三角区/下三角区）

邻接矩阵法的性质

设图 G 的邻接矩阵为 A （矩阵元素为0/1），则 A^n 的元素 $A^n[i][j]$ 等于由顶点 i 到顶点 j 的长度为 n 的路径的数目

0	1	0	0
1	0	1	1
0	1	0	1
0	1	1	0

*

0	1	0	0
1	0	1	1
0	1	0	1
0	1	1	0

$$A^2[1][4] = a_{1,1}a_{1,4} + a_{1,2}a_{2,4} + a_{1,3}a_{3,4} + a_{1,4}a_{4,4} = 1$$

$$A^2[2][2] = a_{2,1}a_{1,2} + a_{2,2}a_{2,2} + a_{2,3}a_{3,2} + a_{2,4}a_{4,2} = 3$$

1	0	1	1
0	3	1	1
1	1	2	1
1	1	1	2

邻接表法

```
//顶点
typedef struct VNode{
    VertexType data; //顶点信息
    ArcNode *first; //第一条边/弧
}VNode,AdjList[MaxVertexNum];
```

```
//用邻接表存储的图
typedef struct{
    AdjList vertices;
    int vexnum,arcnum;
}ALGraph;
```

```
//边/弧
typedef struct ArcNode{
    int adjvex; //边/弧指向哪个结点
    struct ArcNode *next; //指向下一条弧的指针
    //InfoType info; //边权值
}ArcNode;
```

顺序+链式存储

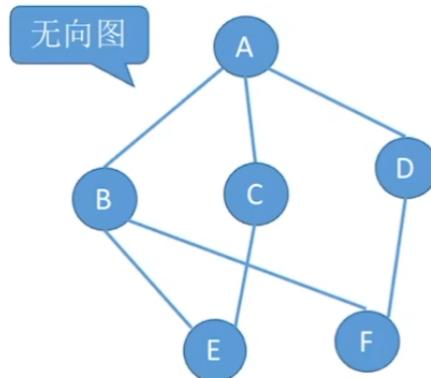
思考：如何求顶点的度、入度、出度？

如何找到与一个顶点相连的边/弧？

图的邻接表表示方式并不唯一

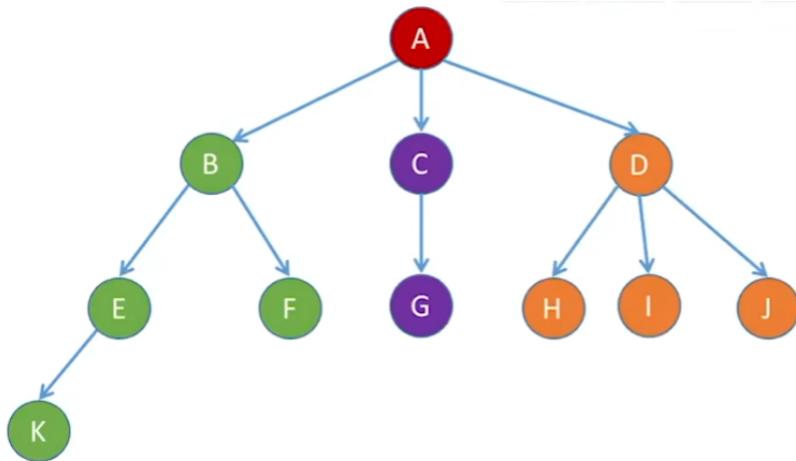
无向图

边结点的数量是 $2|E|$, 整体空间复杂度为 $O(|V|+2|E|)$



	data	*first	指向第一条边
0	A		1 → 2 → 3 → ^
1	B		0 → 4 → 5 → ^
2	C		0 → 4 → ^
3	D		0 → 5 → ^
4	E		1 → 2 → ^
5	F		1 → 3 → ^

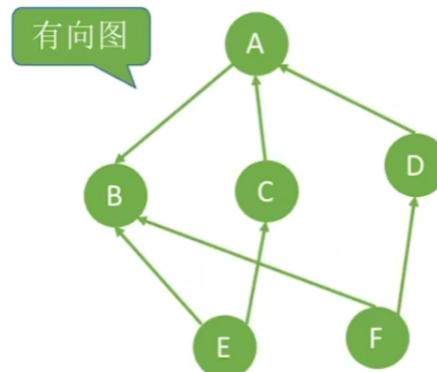
对比：树的孩子表示法



	data	*firstChild
0	A	→ 1 → 2 → 3 ^
1	B	→ 4 → 5 ^
2	C	→ 6 ^
3	D	→ 7 → 8 → 9 ^
4	E	→ 10 ^
5	F	^
6	G	^
7	H	^
8	I	^
9	J	^
10	K	^

孩子表示法：顺序存储各个结点，每个结点中保存孩子链表头指针

有向图



	data	*first
0	A	→ 1 ^
1	B	^
2	C	→ 0 ^
3	D	→ 0 ^
4	E	→ 1 → 2 ^
5	F	→ 1 → 3 ^

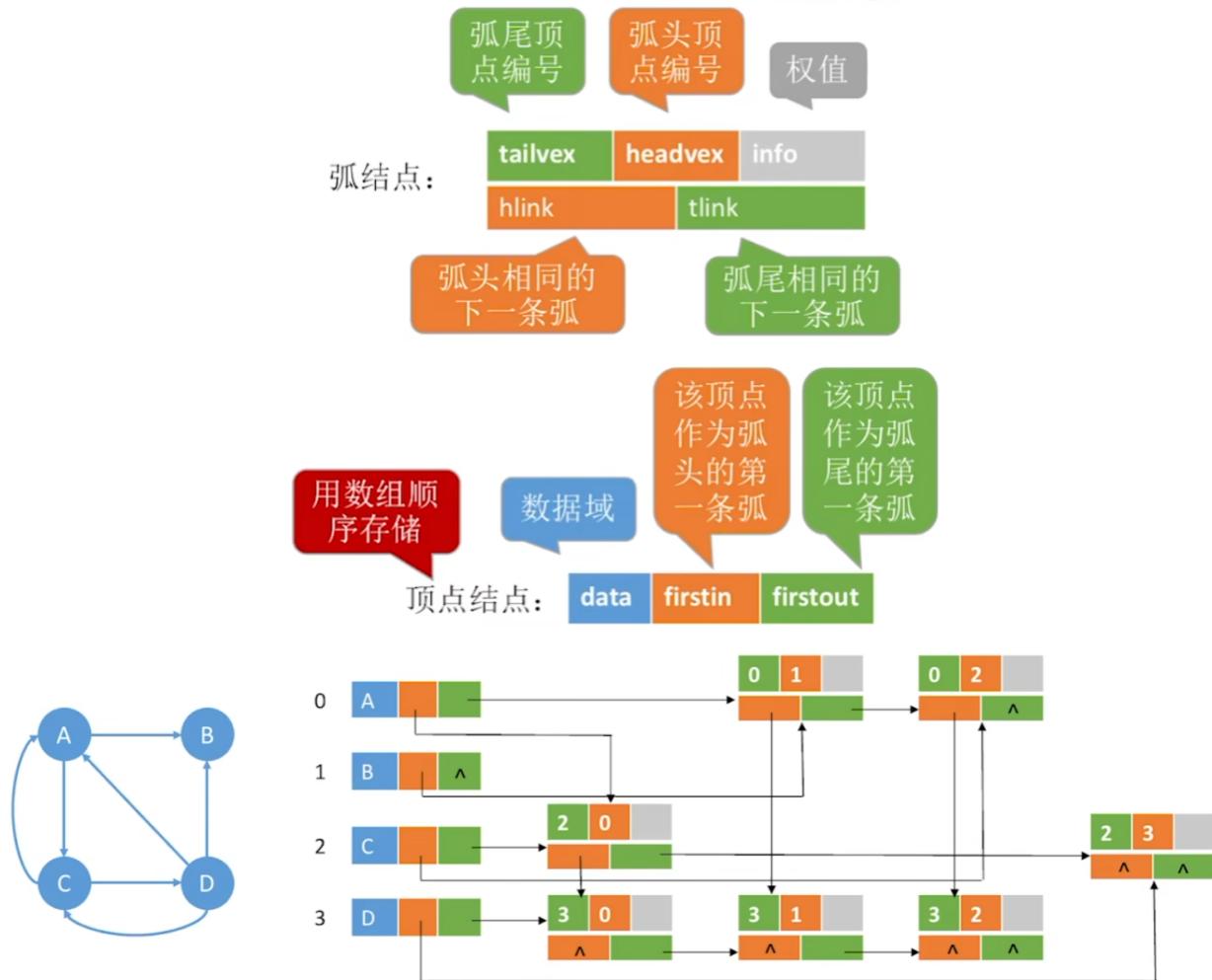
边结点的数量是 $|E|$ ，整体空间复杂度为 $O(|V| + |E|)$

邻接表	邻接矩阵
-----	------

	邻接表	邻接矩阵
空间复杂度	无向图 $O(V +2 E)$ 有向图 $O(V + E)$	$O(V ^2)$
适合用于	存储稀疏图	存储稠密图
表示方式	不唯一	唯一
计算度/出度/入度	计算有向图的度、入度不方便，其余很方便	必须遍历对应行或列
找相邻的边	找有向图的入边不方便，其余很方便	必须遍历对应行或列

十字链表

存储有向图



十字链表法性能分析

空间复杂度: $O(|V|+|E|)$

如何找到指定顶点的所有出边? ——顺着绿色线路找

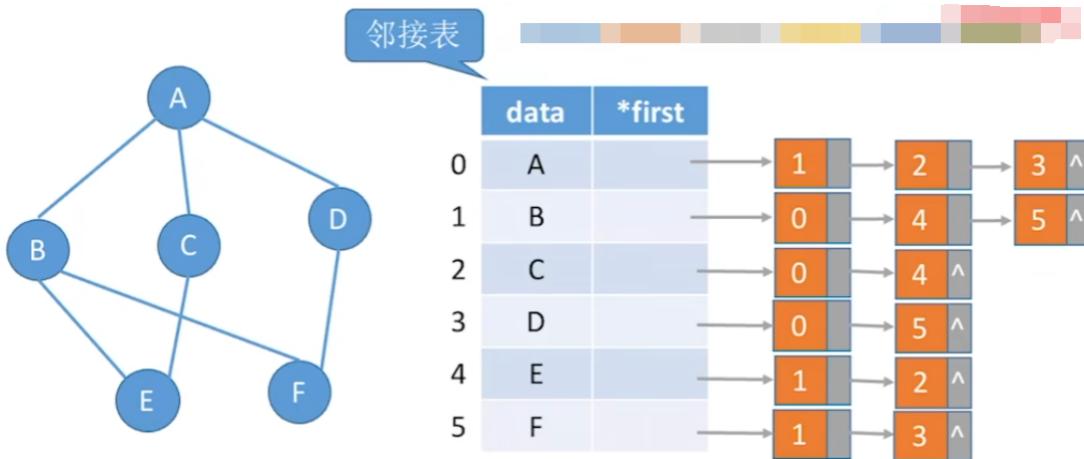
如何找到指定顶点的所有入边? ——顺着橙色线路找

邻接多重表

存储无向图

邻接矩阵、邻接表存储无向图

邻接表



每条边对应两份冗余信息，删除顶点、删除边等操作时间复杂度高

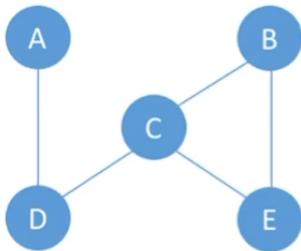
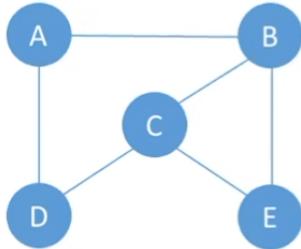
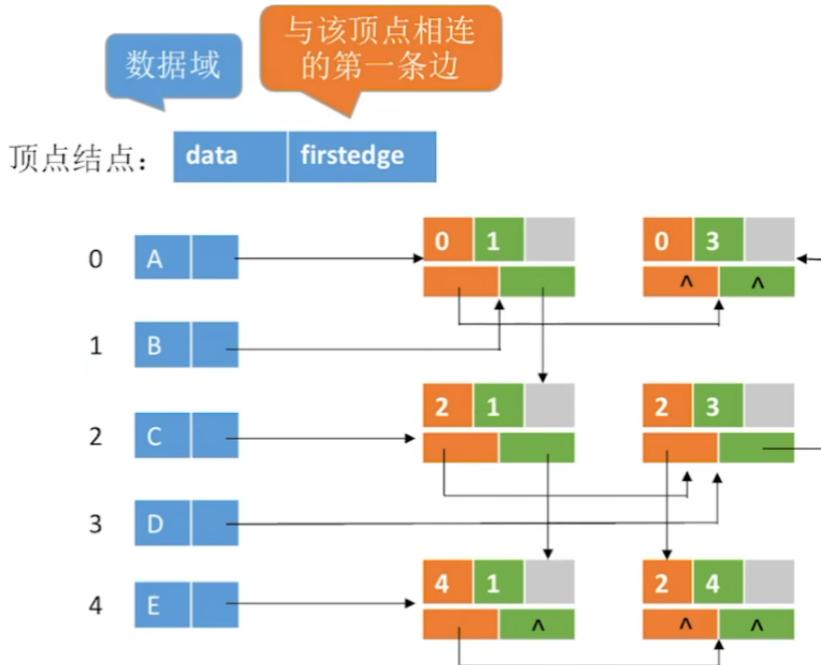
邻接矩阵

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	1	0
D	1	0	0	0	0	1
E	0	1	1	0	0	0
F	0	1	0	1	0	0

空间复杂度高 $O(|V|^2)$

邻接多重表存储无向图





每条边只对应一份数据

空间复杂度: $O(|V| + |E|)$

删除边、删除结点等操作很方便

注意: 邻接多重表只适用于存储无向图

	邻接矩阵	邻接表	十字链表	邻接多重表
空间复杂度	$O(V ^2)$	无向图 $O(V +2 E)$ 有向图 $O(V + E)$	$O(V + E)$	$O(V + E)$
找相邻边	遍历对应行或列 时间复杂度为 $O(V)$	找有向图的入边必须遍历整个邻接表	很方便	很方便
删除边或顶点	删除边很方便, 删除顶点数据需要大量移动数据	无向图中删除边或顶点都不方便	很方便	很方便

	邻接矩阵	邻接表	十字链表	邻接多重表
适用于	稠密图	稀疏图和其他	只能存有向图	只能存无向图
表示方式	唯一	不唯一	不唯一	不唯一

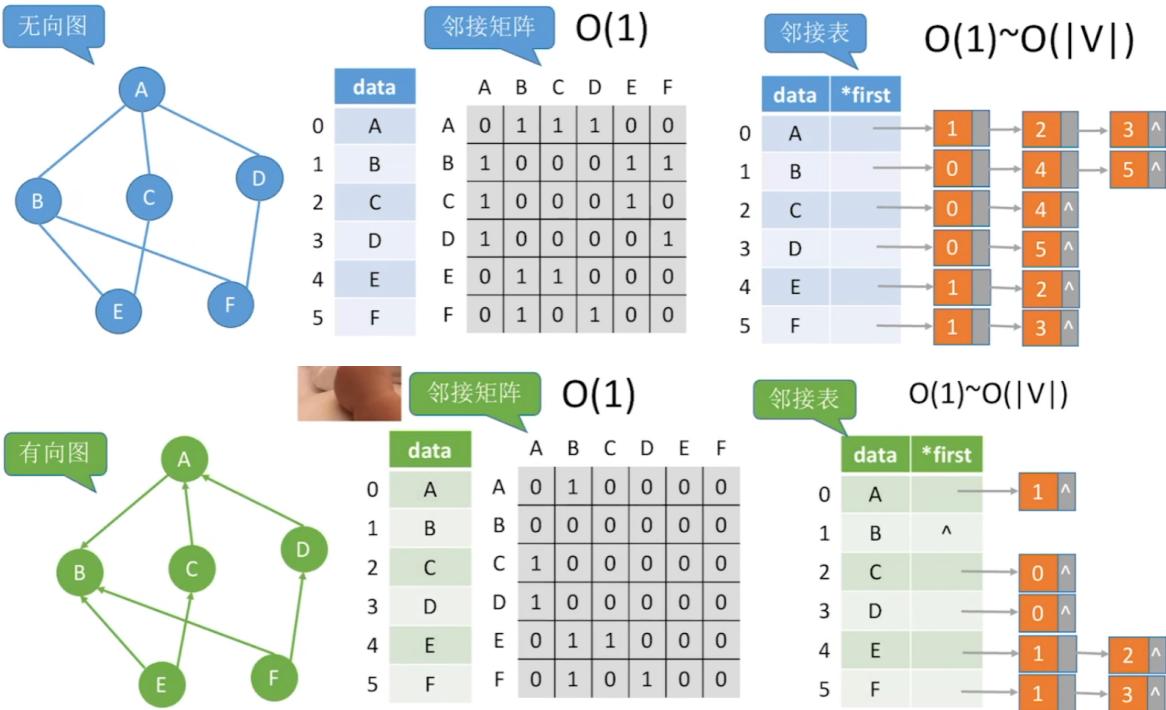


图的基本操作

图的基本操作

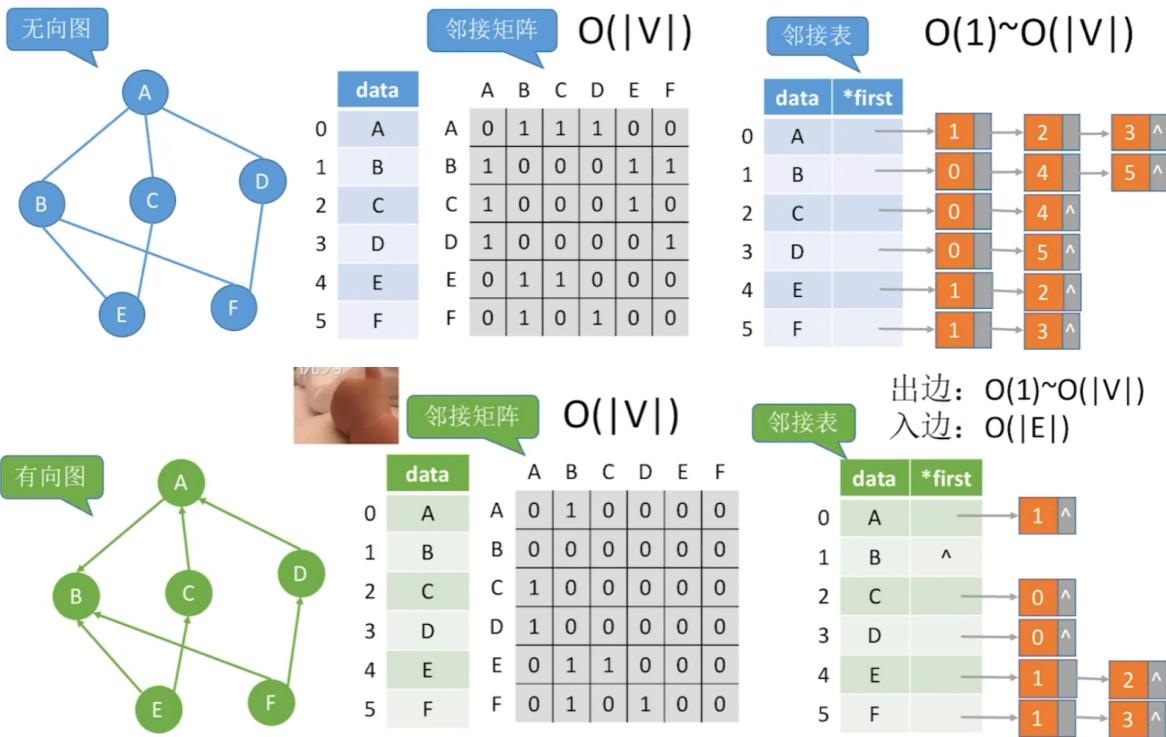
- Adjacent(G,x,y)

判断图G是否存在边<x,y>或(x,y)



- Neighbors(G,x)

列出图G中与结点x邻接的边



- InsertVertex(G, x)

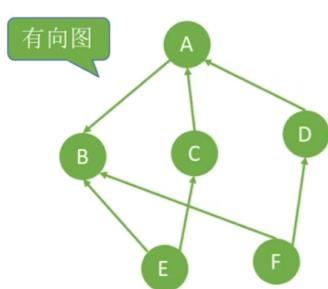
在图G中插入顶点x



- DeleteVertex(G, x)

从图G中删除顶点x





邻接矩阵 $O(|V|)$

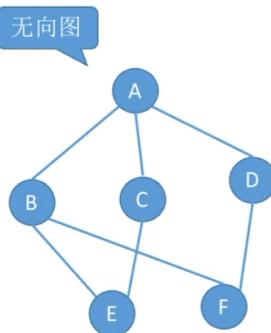
	data	A	B	C	D	E	F
0	A	A	0	1	0	0	0
1	B	B	0	0	0	0	0
2	C	C	1	0	0	0	0
3	D	D	1	0	0	0	0
4	E	E	0	1	1	0	0
5	F	F	0	1	0	1	0

邻接表 删出边: $O(1) \sim O(|V|)$

	data	*first
0	A	1 ^
1	B	^
2	C	0 ^
3	D	0 ^
4	E	1 ^
5	F	1 ^

- AddEdge(G,x,y)

若无向边(x,y)或有向边<x,y>不存在，则向图G中添加该边



邻接矩阵 $O(1)$

	data	A	B	C	D	E	F	
0	A	A	0	1	1	1	0	0
1	B	B	1	0	0	0	1	1
2	C	C	1	0	0	0	1	0
3	D	D	1	0	0	0	0	1
4	E	E	0	1	1	0	0	0
5	F	F	0	1	0	1	0	0

邻接表 $O(1)$

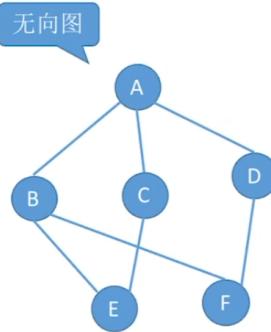
	data	*first
0	A	1 ^
1	B	0 ^
2	C	0 ^
3	D	0 ^
4	E	1 ^
5	F	1 ^

- RemoveEdge(G,x,y)

若无向图(x,y)或有向边<x,y>存在，则从图G中删除该边

- FirstNeighbor(G,x)

求图G中顶点x的第一个邻接点，若有则返回顶点号。若x没有邻接点或图中不存在x，则返回-1

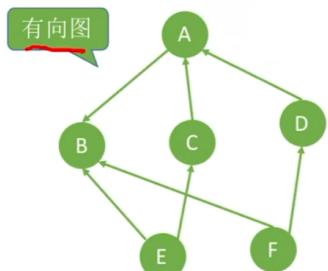


邻接矩阵 $O(1) \sim O(|V|)$

	data	A	B	C	D	E	F	
0	A	A	0	1	1	1	0	0
1	B	B	1	0	0	0	1	1
2	C	C	1	0	0	0	1	0
3	D	D	1	0	0	0	0	1
4	E	E	0	1	1	0	0	0
5	F	F	0	1	0	1	0	0

邻接表 $O(1)$

	data	*first
0	A	1 ^
1	B	0 ^
2	C	0 ^
3	D	0 ^
4	E	1 ^
5	F	1 ^



邻接矩阵 $O(1) \sim O(|V|)$

	data	A	B	C	D	E	F
0	A	A	0	1	0	0	0
1	B	B	0	0	0	0	0
2	C	C	1	0	0	0	0
3	D	D	1	0	0	0	0
4	E	E	0	1	1	0	0
5	F	F	0	1	0	1	0

邻接表 找出边邻接点: $O(1)$ 找入边邻接点: $O(1) \sim O(|E|)$

	data	*first
0	A	1 ^
1	B	^
2	C	0 ^
3	D	0 ^
4	E	1 ^
5	F	1 ^

- NextNeighbor(G,x,y)

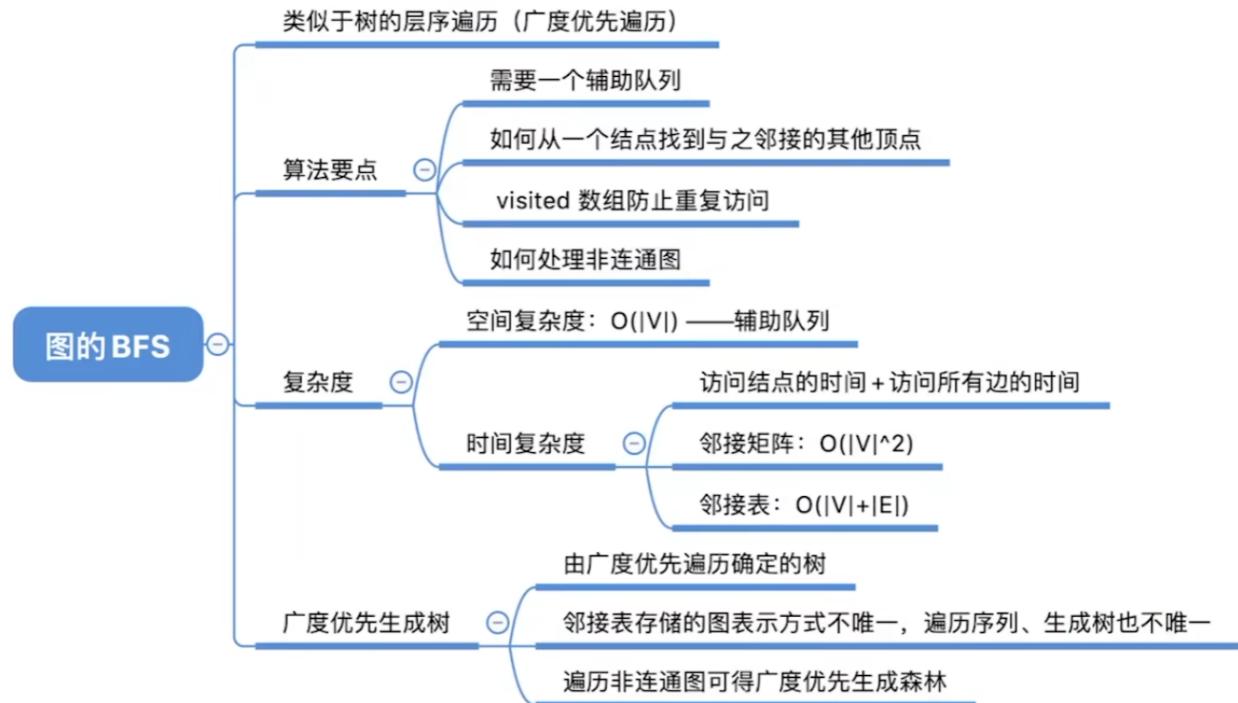
假设图G中顶点y是顶点x的一个邻接点，返回除y之外顶点x的下一个邻接点的顶点号，若y是x的最后一一个邻接点，则返回-1



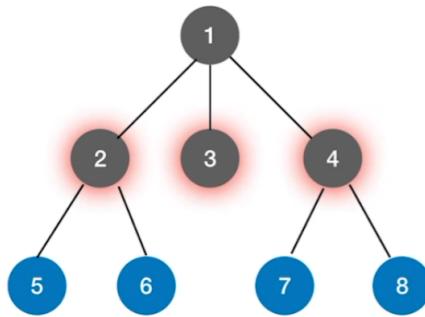
- `Get_edge_value(G,x,y)`
获取图G中边(x,y)或<x,y>对应的权值
 - 核心在于找到边
`Adjacent(G,x,y)`: 判断图G是否存在边<x,y>或(x,y)
- `Set_edge_value(G,x,y,v)`
设置图G中边(x,y)或<x,y>对应的权值为v
 - 核心在于找到边
`Adjacent(G,x,y)`: 判断图G是否存在边<x,y>或(x,y)

图的遍历算法

图的广度优先遍历 BFS



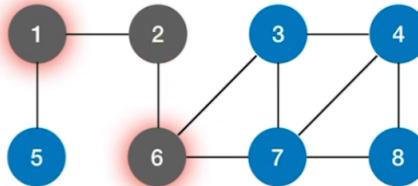
树vs图



不存在“回路”，搜索相邻的结点时，不可能搜到已经访问过的结点

树的广度优先遍历（层序遍历）：

1. 若树非空，则根结点入队
2. 若队列非空，队头元素出队并访问，同时将该元素的孩子依次入队
3. 重复2直到队列为空



搜索相邻的顶点时，有可能搜到已经访问过的顶点

代码实现

广度优先遍历（Breadth-First-Search, BFS）要点：

1. 找到与一个顶点相邻的所有顶点
2. 标记哪些顶点被访问过
3. 需要一个辅助队列

```
bool visited[MAX_VERTEX_NUM]; //访问标记数组，初始都为false
//广度优先遍历
void BFS(Graph G, int v) //从顶点v出发，广度优先遍历图G
{
    visit(v); //访问初始顶点v
    visited[v] = TRUE; //对v做已访问标记
    Enqueue(Q, v); //顶点v入队列Q
    while (!isEmpty(Q))
    {
        DeQueue(Q, v); //顶点v出队列
        for (w = FirstNeighbor(G, v); w >= 0; w = NextNeighbor(G, v, w))
            //检测v所有邻接点
            if (!visited[w]) //w为v的尚未访问的邻接顶点
            {
                visit(w); //访问顶点w
                visited[w] = TRUE; //对w做已访问标记
            }
    }
}
```

```

        EnQueue(Q,w); //顶点w入队列
    }
}
}

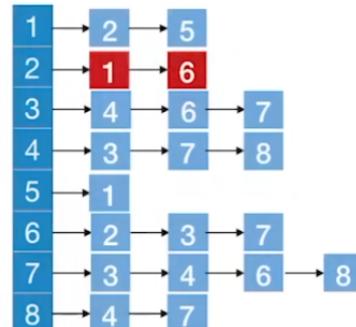
```

广度优先遍历序列

遍历序列的可变性

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

邻接矩阵



邻接表

同一个图的邻接矩阵表示方式唯一，因此广度优先遍历序列唯一

同一个图的邻接表表示方式不唯一，因此广度优先遍历序列不唯一

算法存在的问题

结论：对于无向图，调用BFS函数的次数=连通分量数

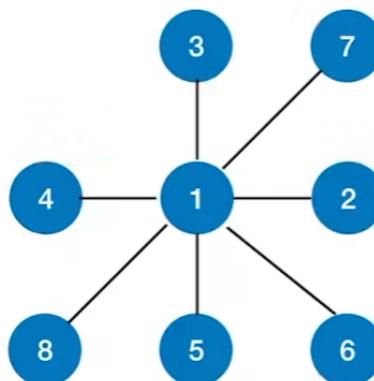
如果是非连通图，则无法遍历完所有结点

```

void BFSTraverse(Graph G)//对图G进行广度优先遍历
{
    for(i=0;i<G.vexnum;++i)
        visited[i]=FALSE;//访问标记数组初始化
    InitQueue(Q);//初始化辅助队列Q
    for(i=0;i<G.vexnum;++i)//从0号顶点开始遍历
        if(!visited[i])//对每个连通分量调用依次BFS
            BFS(G,i);//vi未访问过,从vi开始BFS
}

```

复杂度分析



空间复杂度：最坏情况，辅助队列大小为 $O(|V|)$

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

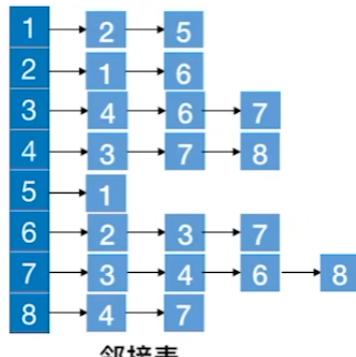
邻接矩阵

邻接矩阵存储的图：

访问 $|V|$ 个顶点需要 $O(|V|)$ 的时间

查找每个顶点的邻接点都需要 $O(|V|)$ 的时间，而总共由 $|V|$ 个顶点

时间复杂度= $O(|V|^2)$



邻接表

邻接表存储的图：

访问 $|V|$ 个顶点需要 $O(|V|)$ 的时间

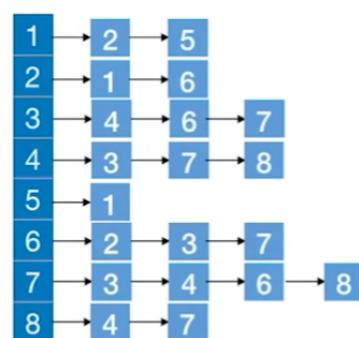
查找各个顶点的邻接点共需要 $O(|E|)$ 的时间，

时间复杂度= $O(|V| + |E|)$

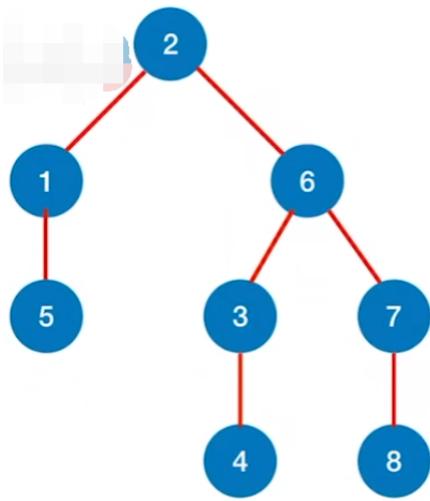
广度优先生成树

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

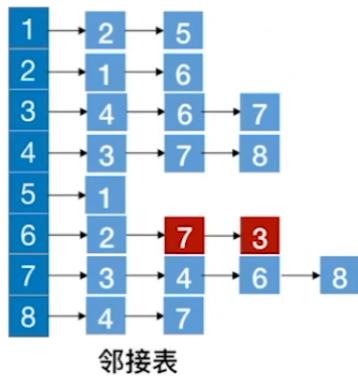
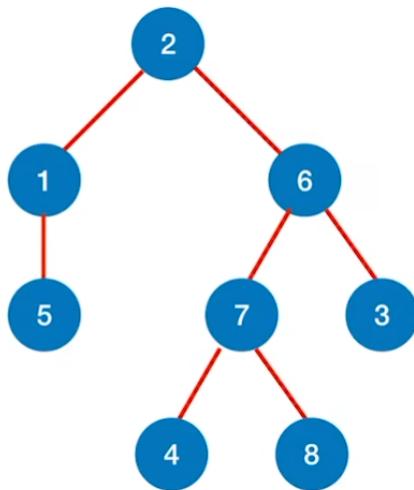
邻接矩阵



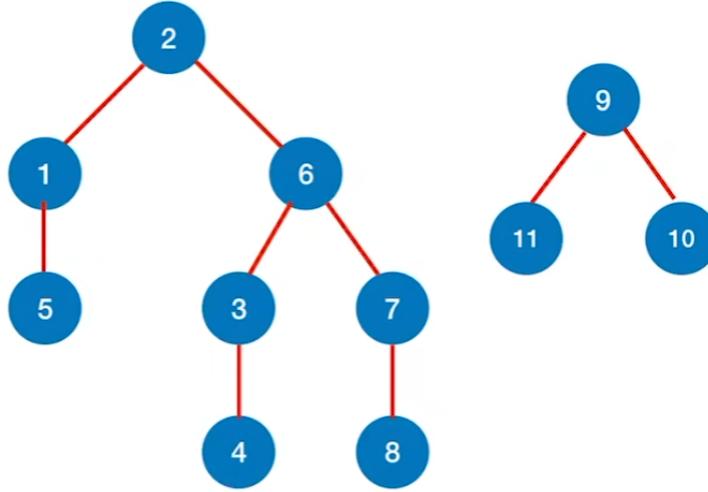
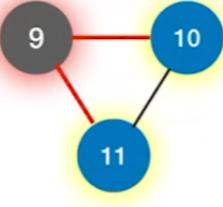
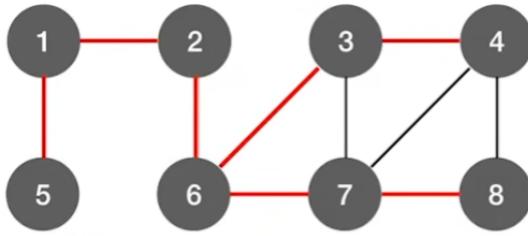
邻接表



广度优先生成树由广度优先遍历过程确定。由于邻接表的表示方式不唯一，因此基于邻接表的广度优先生成树也不唯一。



广度优先生成森林

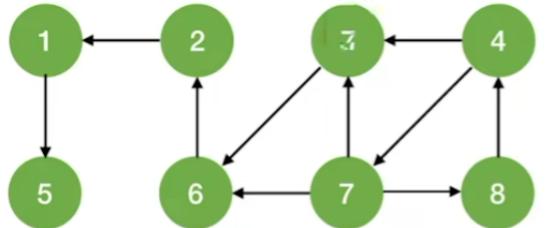


对非连通图的广度优先遍历，可得到广度优先生成森林

练习：有向图的BFS过程

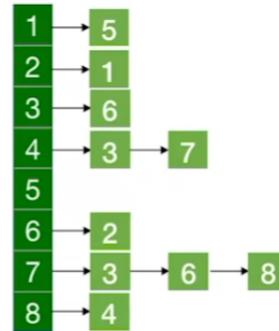
思考：

1. 从1出发，需要调用几次BFS函数？
2. 从7出发，需要调用几次BFS函数？



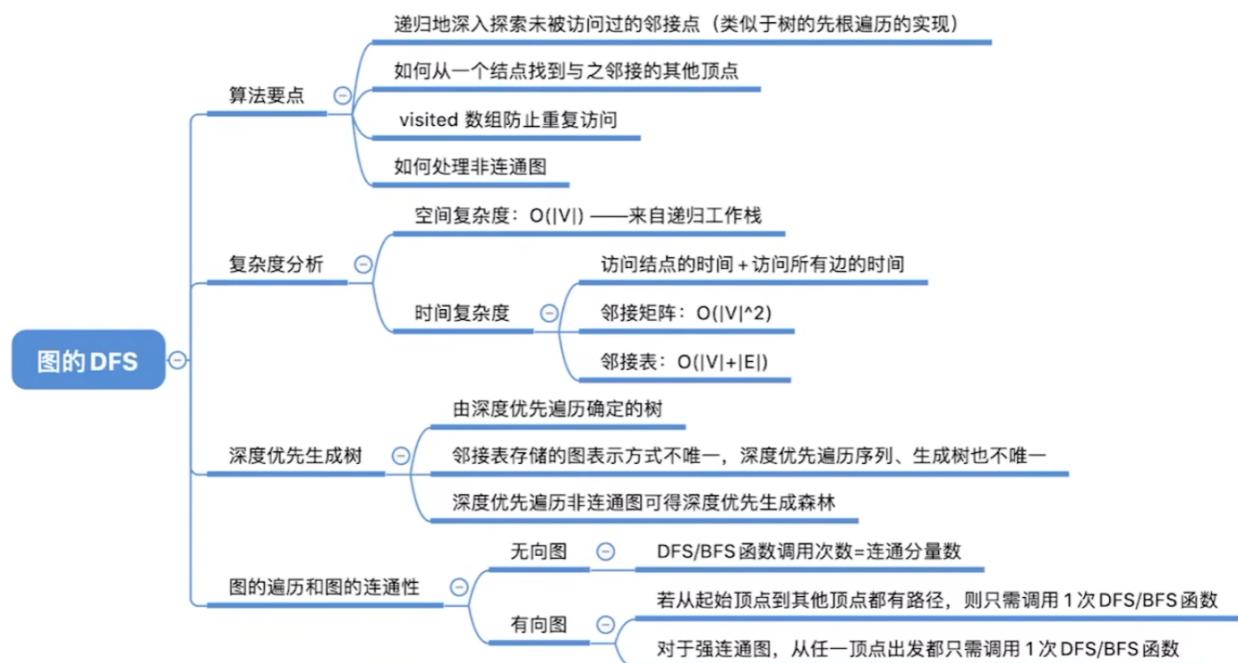
	1	2	3	4	5	6	7	8
1	0	0	0	0	1	0	0	0
2	1	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0
4	0	0	1	0	0	0	1	0
5	0	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0
7	0	0	1	0	0	1	0	1
8	0	0	0	1	0	0	0	0

邻接矩阵

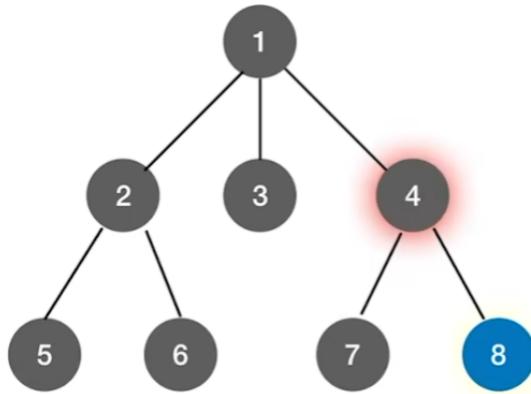


邻接表

图的深度优先遍历 DFS



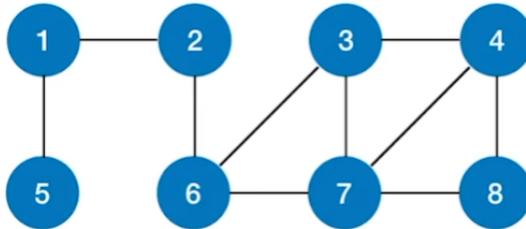
树的深度优先遍历



```
//树的先根遍历
void PreOrder(TreeNode *R)
{
    if(R!=NULL)
    {
        visit(R); //访问根结点
        while(R还有下一个子树T)
            PreOrder(T); //先根遍历下一棵子树
    }
}
```

新找到的相邻结点一定是没有访问过的

图的深度优先遍历



```
bool visited[MAX_VERTEX_NUM]; //访问标记数组
void DFS(Graph G, int v)//从顶点v出发,深度优先遍历图G
{
    visit(v); //访问顶点v
    visited[v]=TRUE; //设已访问标记
    for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w))
    {
        if(!visited[w]) //w为u的尚未访问的邻接顶点
        {
            DFS(G,w);
        }
    }
}
```

算法存在的问题

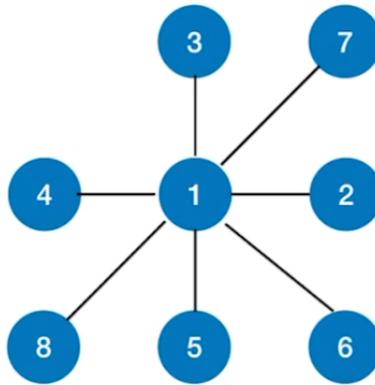
如果是非连通图，则无法遍历完所有结点

```
void DFSTraverse(Graph G) //对图G进行深度优先遍历
{
    for(v=0; v<G.vexnum; ++v)
        visited[v] = FALSE; //初始化已访问标记数据
    for(v=0; v<G.vexnum; ++v) //本代码中是从v=0开始遍历
        if(!visited[v])
            DFS(G, v);
}
```

复杂度分析

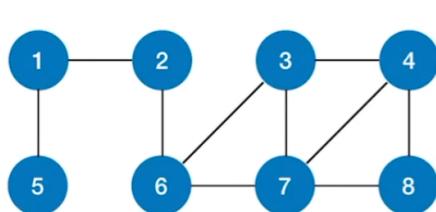


空间复杂度：来自函数调用栈，最坏情况，递归深度为 $O(|V|)$



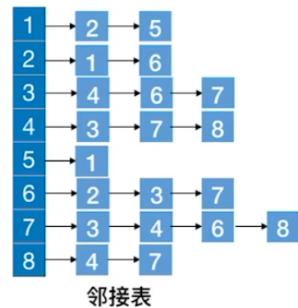
空间复杂度：最好情况， $O(1)$

时间复杂度 = 访问各结点所需时间 + 探索各条边所需时间



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

邻接矩阵



邻接矩阵存储的图：

访问 $|V|$ 个顶点需要 $O(|V|)$ 的时间

查找每个顶点的邻接点都需要 $O(|V|)$ 的时间，而总共由 $|V|$ 个顶点

时间复杂度 = $O(|V|^2)$

邻接表存储的图：

访问 $|V|$ 个顶点需要 $O(|V|)$ 的时间

查找各个顶点的邻接点共需要 $O(|E|)$ 的时间，

时间复杂度= $O(|V|+|E|)$

深度优先遍历序列

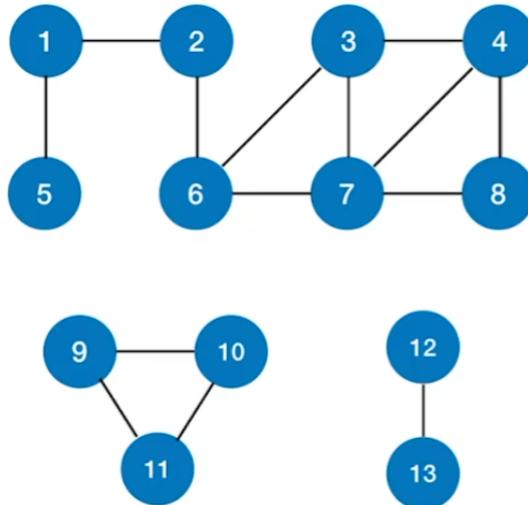
同一个图的邻接矩阵表示方式唯一，因此深度优先遍历序列唯一

同一个图邻接表表示方式不唯一，因此深度优先遍历序列不唯一

深度优先生成树

深度优先生成森林

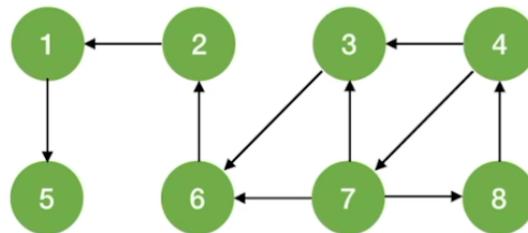
图的遍历与图的连通性



对无向图进行BFS/DFS遍历

调用BFS/DFS函数的次数=连通分量数

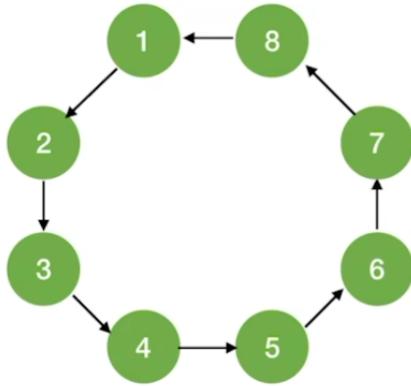
对于连通图，只需调用1次BFS/DFS



对有向图进行BFS/DFS遍历

调用BFS/DFS函数的次数要具体问题具体分析

若起始顶点到其他各顶点都有路径，则只需调用1次BFS/DFS函数



对于强连通图，从任一结点出发都只需调用1次BFS/DFS

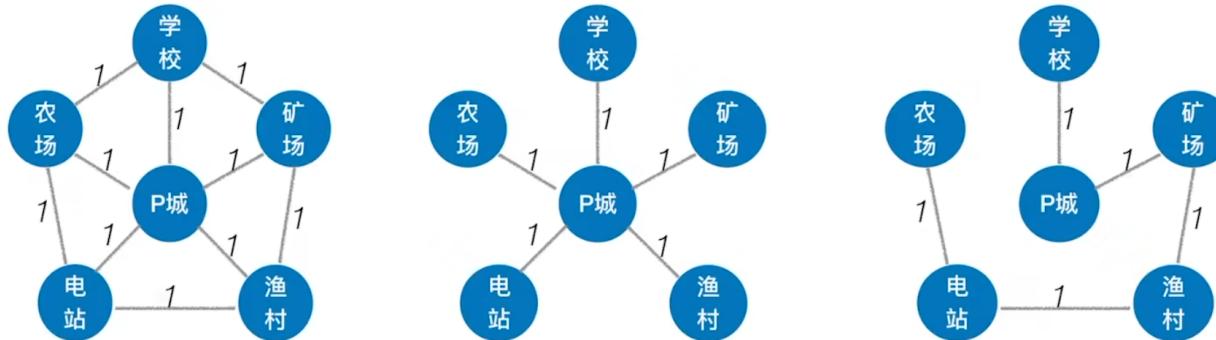
图的应用

最小生成树

最小生成树（最小代价树）

道路规划要求：所有地方都连通，且成本尽可能的低

对于一个带权连通无向图 $G=(V,E)$ ，生成树不同，每棵树的权（即树中所有边上的权值之和）也可能不同。设 R 为 G 的所有生成树的集合，若 T 为 R 中边的权值之和最小的生成树，则 T 称为 G 的最小生成树（Minimum-Spanning-Tree, MST）。

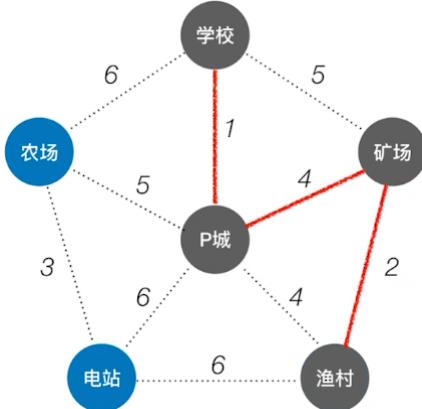


- 最小生成树可能有多个，但边的权值之和总是唯一且最小的
- 最小生成树的边数=顶点数-1。砍掉一条则不连通，增加一条边则会出现回路
- 如果一个连通图本身就是一棵树，则其最小生成树就是它本身
- 只有连通图才有生成树，非连通图只有生成森林

Prim算法（普里姆）

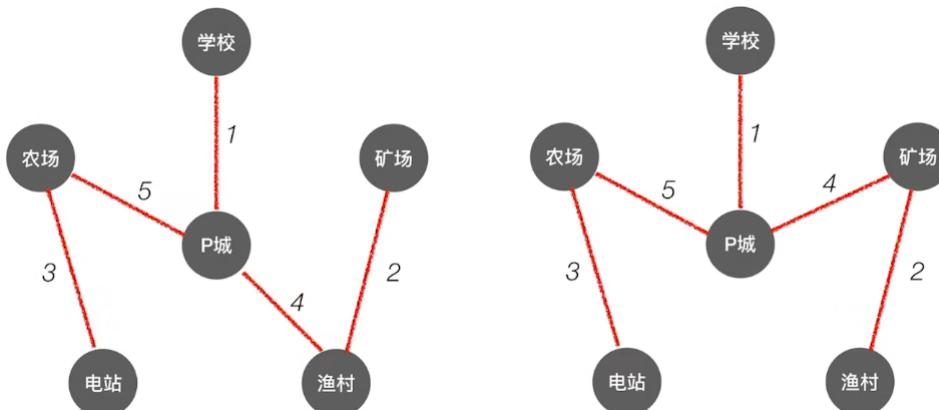
时间复杂度： $O(|V|^2)$

适用于边稠密图



从某一个顶点开始构建生成树；

每次将代价最小的新顶点纳入生成树，直到所有顶点都纳入为止。



Kruskal算法（克鲁斯卡尔）

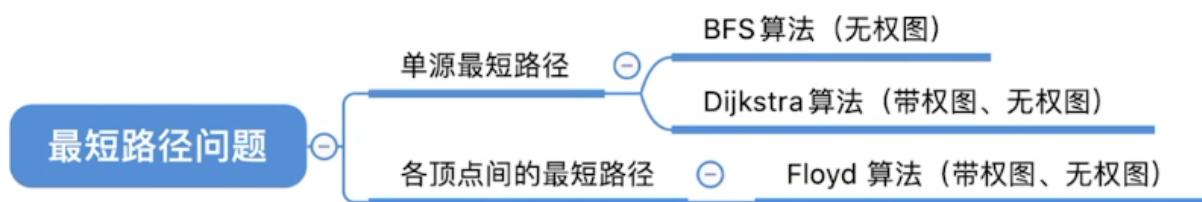
时间复杂度： $O(|E|\log_2|E|)$

适用于边稀疏图

每次选择一条权值最小的边，使这条边的两头连通（原本已经连通的就不选）

直到所有结点都连通

最短路径问题



"G港"是个物流集散中心，经常需要往各个城市运东西，怎么运送距离最近？——单源最短路径

各个城市之间也需要互相往来，相互之间怎么走距离最近——没对顶点间的最短路径

	BFS 算法	Dijkstra 算法	Floyd 算法
无权图	✓	✓	✓
带权图	✗	✓	✓
带负权值的图	✗	✗	✓
带负权回路的图	✗	✗	✗
时间复杂度	$O(V ^2)$ 或 $O(V + E)$	$O(V ^2)$	$O(V ^3)$
通常用于	求无权图的单源最短路径	求带权图的单源最短路径	求带权图中各顶点间的最短路径

注：也可用Dijkstra算法求所有顶点间的最短路径，重复 $|V|$ 次即可，总的时间复杂度也是 $O(|V|^3)$

单源最短路径

BFS算法（无权图）

注：无权图可以视为一种特殊的带权图，只是每条边的权值都为1

代码实现

```

bool visited[MAX_VERTEX_NUM];//访问标记数组
//广度优先遍历
void BFS(Graph G, int v)//从顶点v出发，广度优先遍历图G
{
    visit(v);//访问初始顶点v
    visited[v]=TRUE;//对v做已访问标记
    Enqueue(Q,v);//顶点v入队列Q
    while(!isEmpty(Q))
    {
        DeQueue(Q,v);//顶点v出队列
        for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w))
            //检测v所有邻接点
            if(!visited[w])//w为v的尚未访问的邻接顶点
            {
                visit(w);//访问顶点w
                visited[w]=TRUE;//对w做已访问标记
                Enqueue(Q,w);//顶点w入队列
            }
    }
}
//广度优先遍历
void BFS_MIN_Distance(Graph G, int u)//从顶点v出发，广度优先遍历图G
{
    //d[i]表示从u到i结点的最短路径
    for(i=0;i<G.vexnum;++i)
    {
        d[i]=∞;//初始化路径长度
        path[i]=-1;//最短路径从哪个顶点过来
    }
}

```

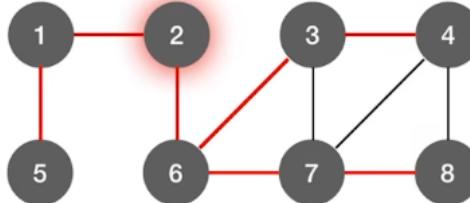
```

d[u]=0;
visited[u]=TRUE;
Enqueue(Q,u);
while(!isEmpty(Q))//BFS算法主过程
{
    DeQueue(Q,u);//队头元素u出队
    for(w=FirstNeighbor(G,u);w>=0;w=NextNeighbor(G,u,w))
        //检测v所有邻接点
        if(!visited[w])//w为u的尚未访问的邻接顶点
    {
        d[w]=d[u]+1;//路径长度加1
        path[w]=u;//访问顶点w
        visited[w]=TRUE;//对已访问标记
        EnQueue(Q,w);//顶点w入队列
    }
}
}

```

	1	2	3	4	5	6	7	8
d[]	∞							
path[]	-1	-1	-1	-1	-1	-1	-1	-1

	1	2	3	4	5	6	7	8
visited	false							



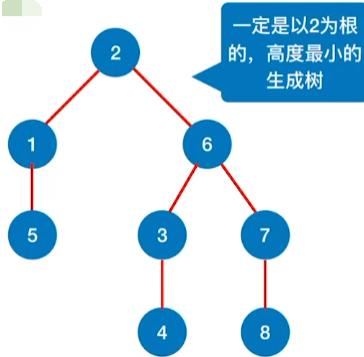
	1	2	3	4	5	6	7	8
d[]	1	0	2	3	2	1	2	3
path[]	2	-1	6	3	1	2	6	7

对BFS的小修改，在visit一个顶点时，修改其最短路径长度d[]并在path[]记录前驱结点

2到8的最短路径长度=d[8]=3

通过path数组可知，2到8的最短路径为：8762

广度优先生成树一定是以2为根，高度最小的生成树



Dijkstra算法 (带权图、无权图)

Dijkstra

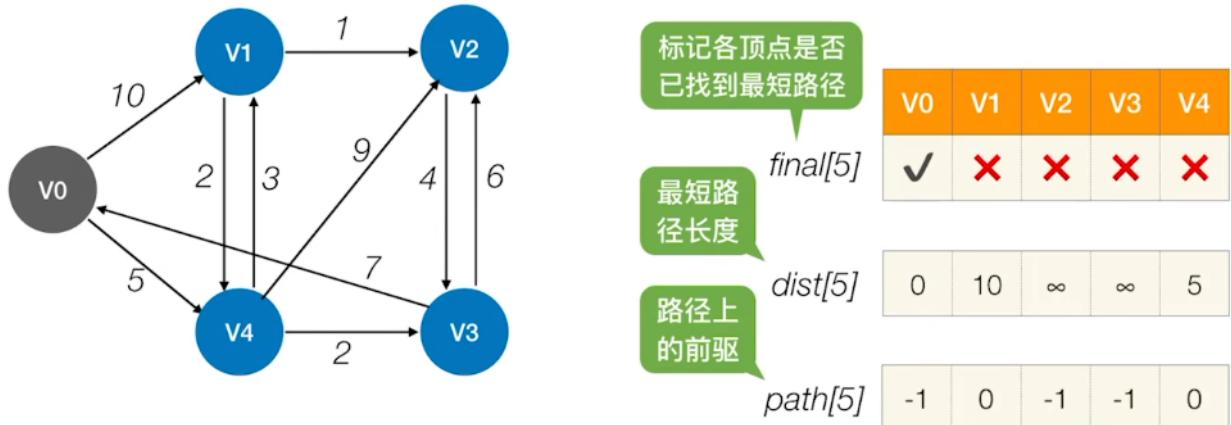
- 提出"goto 有害理论"——操作系统，虚拟存储技术
- 信号量机制PV原语——操作系统，进程同步
- 银行家算法——操作系统，死锁
- 解决哲学家进餐问题——操作系统，死锁
- Dijkstra最短路径算法——数据结构大题、小题

BFS算法的局限性

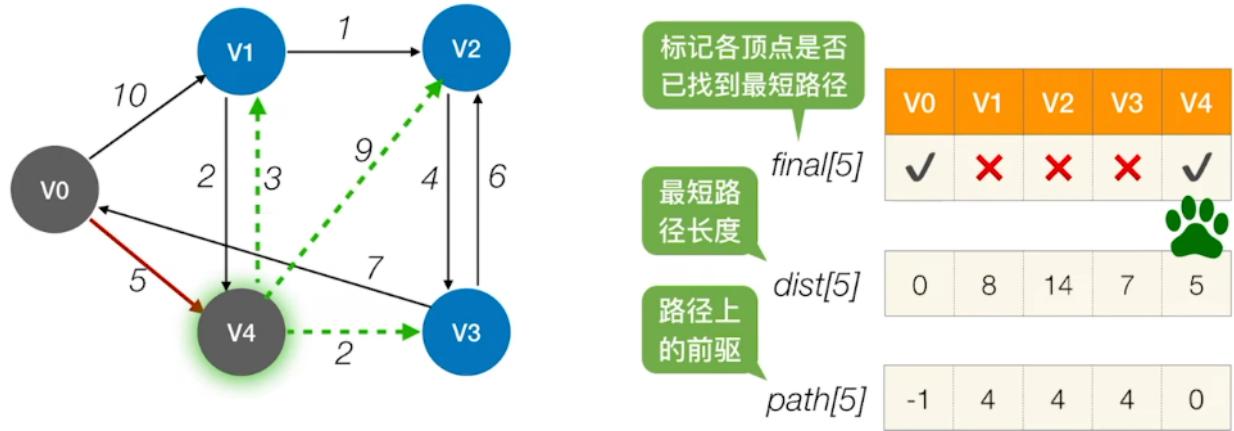
BFS算法求单源最短路径只适用于无权图，或所有边的权值都相同的图

带权路径长度——当图是带权图时，一条路径上所有边的权值之和，称为该路径的带权路径长度

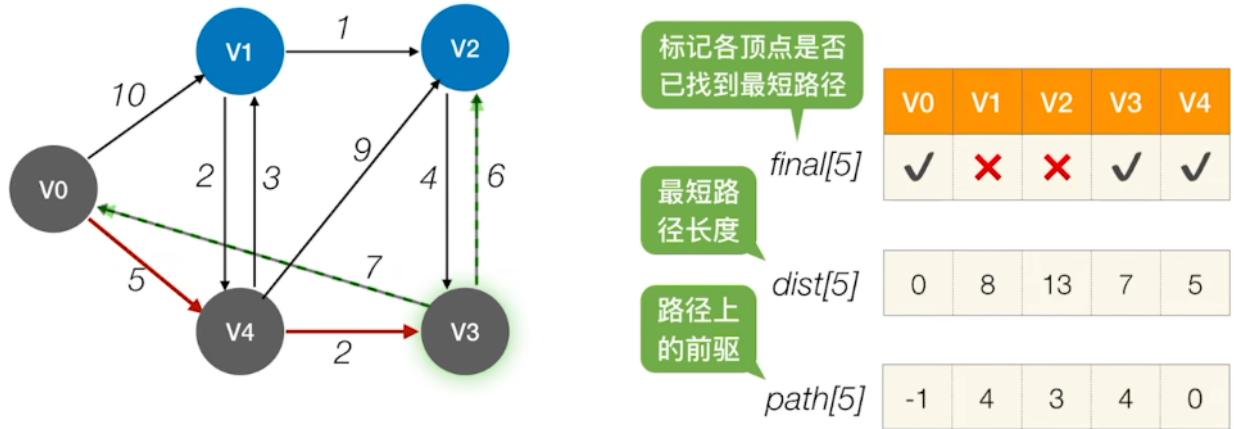
Dijkstra算法



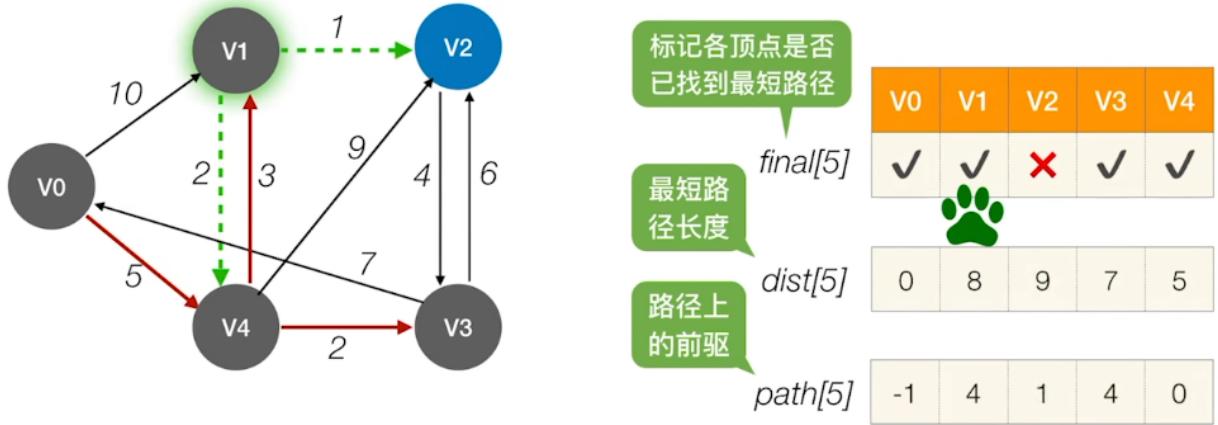
初始：从V0开始，初始化三个数组信息



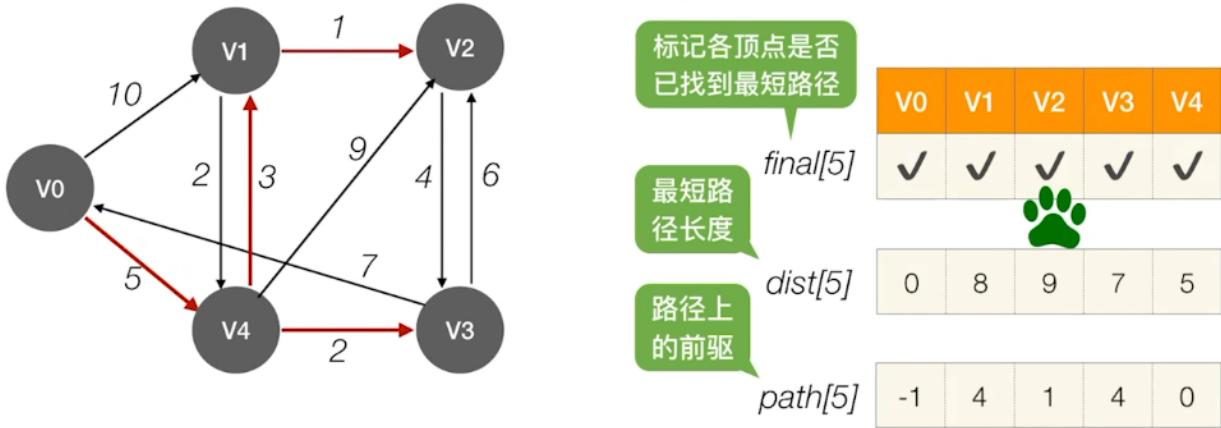
第1轮：循环遍历所有结点，找到还没有确定最短路径，且dist最小的顶点Vi，令final[i]=true。检查所有邻接自Vi的顶点，若其final值为false，则更新dist和path信息



第2轮：循环遍历所有结点，找到还没确定最短路径，且dist最小的顶点Vi，令final[i]=true。



第3轮：循环遍历所有结点，找到还没确定最短路径，且dist最小的顶点Vi，令final[i]=true。



第4轮：循环遍历所有结点，找到还没确定最短路径，且dist最小的顶点Vi，令final[i]=true。

如何使用数组信息

V0到V2的最短（带权）路径长度为：dist[2]=9

通过path[]可知，V0到V2的最短（带权）路径：V2<-V1<-V4<-V0

Dijkstra算法的时间复杂度

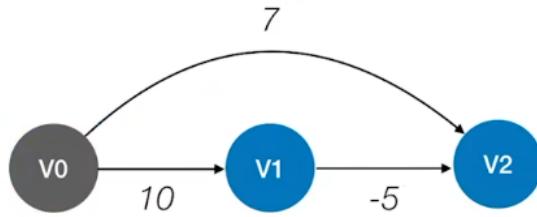
```
//初始
//从v0开始
final[0]=True;
dist[0]=0;
path[0]=-1;
final[k]=False;
//其余顶点
final[k]=False;
dist[k]=arcs[0][k];
path[k]=(arcs[0][k]==∞)?-1:0
```

```
//n-1轮处理：
//循环遍历所有顶点，找到还没确定最短路径，且dist最小的顶点vi，令final[i]=true
//并检查所有邻接自vi的顶点，对于邻接自vi的顶点vj，若final[j]==false且dist[i]+arcs[i][j]
<dist[j]，
//则令dist[j]=dist[i]+arcs[i][j];path[j]=i
//注：arcs[i][j]表示vi到vj的弧的权值
```

时间复杂度： $O(n^2)$ 即 $O(|V|^2)$

对比：Prim算法的实现思想

对于负权值带权图



标记各顶点是否已找到最短路径
final[3]

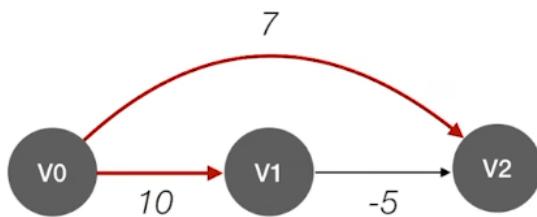
V0	V1	V2
✓	✗	✗

最短路径长度
dist[3]

0	10	7
---	----	---

路径上的前驱
path[3]

-1	0	0
----	---	---



标记各顶点是否已找到最短路径
final[3]

V0	V1	V2
✓	✓	✓

最短路径长度
dist[3]

0	10	7
---	----	---

路径上的前驱
path[3]

-1	0	0
----	---	---

事实上V0到V2的最短带权路径长度为5

结论：Dijkstra算法不适用于有负权值的带权图

各顶点间的最短路径

Floyd算法（带权图、无权图）

求出每一对顶点之间的最短路径

使用动态规划思想，将问题的求解分为多个阶段

对于n个顶点的图G，求任意一对顶点Vi->Vj之间的最短路径可分为如下几个阶段：

#初始：不允许在其他顶点中转，最短路径是？

#0：若允许在V0中转，最短路径是？

#1：若允许在V0、V1中转，最短路径是？

#2：若允许在V0、V1、V2中转，最短路径是？

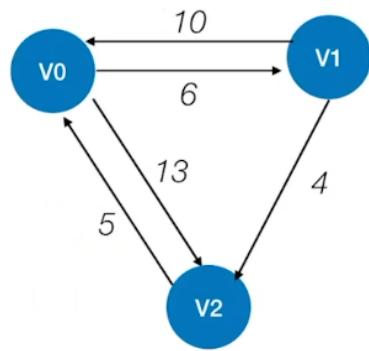
...

#n-1：若允许在V0、V1、V2...Vn-1中转，最短路径是？

Robert W. Floyd

- Floyd算法（Floyd-Warshall算法）
- 堆排序算法

Floyd算法



目前来看，各顶点间的最短路径长度

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

	V0	V1	V2
V0	0	6	13
V1	10	0	4
V2	5	∞	0

两个顶点之间的中转点

$$path^{(-1)} =$$

	V0	V1	V2
V0	-1	-1	-1
V1	-1	-1	-1
V2	-1	-1	-1

#初始：不允许在其他顶点中转，最短路径是？

目前来看，各顶点间的最短路径长度

$$A^{(0)} =$$

	V0	V1	V2
V0	0	6	13
V1	10	0	4
V2	5	11	0

两个顶点之间的中转点

$$path^{(0)} =$$

	V0	V1	V2
V0	-1	-1	-1
V1	-1	-1	-1
V2	-1	0	-1

若 $A^{(k-1)}[i][j] > A^{(k-1)}[i][k] + A^{(k-1)}[k][j] + A^{(k-1)}[k][j]$

则 $A^{(k)}[i][j] = A^{(k-1)}[i][k] + A^{(k-1)}[k][j]$

$$path^{(k)}[i][j] = k$$

否则 $A^{(k)}$ 和 $path^{(k)}$ 保持原值

#0：若允许在V0中转，最短路径是？

求 $A^{(0)}$ 和 $path^{(0)}$

$$A^{(-1)}[2][l] > A^{(-1)}[2][0] + A^{(-1)}[0][1] = 11$$

$$A^{(0)}[2][1] = 11$$

$$path^{(0)}[2][1] = 0$$

#1：若允许在V0、V1中转，最短路径是？

求 $A^{(1)}$ 和 $path^{(1)}$

$$A^{(0)}[0][2] > A^{(0)}[0][1] + A^{(0)}[1][2] = 10$$

$$A^{(1)}[0][2] = 10$$

$$path^{(1)}[0][2] = 1$$

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

	V0	V1	V2
V0	0	6	10
V1	10	0	4
V2	5	11	0

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

	V0	V1	V2
V0	-1	-1	1
V1	-1	-1	-1
V2	-1	0	-1

#2：若允许在V0、V1、V2中转，最短路径是？

$$A^{(1)}[1][0] > A^{(1)}[1][2] + A^{(1)}[2][0] = 9$$

$$A^{(2)}[1][0] = 9$$

$$path^{(2)}[1][0] = 2$$

	V0	V1	V2
V0	0	6	10
V1	9	0	4
V2	5	11	0

	V0	V1	V2
V0	-1	-1	1
V1	2	-1	-1
V2	-1	0	-1

从 $A^{(-1)}$ 和 $path^{(-1)}$ 开始, 经过n轮递推, 得到 $A^{(n-1)}$ 和 $path^{(n-1)}$

根据 $A^{(2)}$ 可知, $V1$ 到 $V2$ 最短路径长度为4

根据 $path^{(2)}$ 可知, 完整路径信息为 $V1_V2$

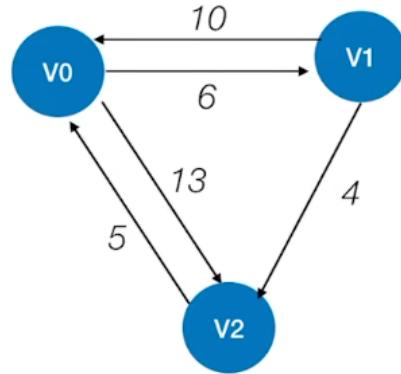
根据 $A^{(2)}$ 可知, $V0$ 到 $V2$ 最短路径长度为10

根据 $path^{(2)}$ 可知, 完整路径信息为 $V0_V1_V2$

根据 $A^{(2)}$ 可知, $V1$ 到 $V0$ 最短路径长度为9

根据 $path^{(2)}$ 可知, 完整路径信息为 $V1_V2_V0$

Floyd算法核心代码



	V0	V1	V2
V0	0	6	13
V1	10	0	4
V2	5	∞	0

	V0	V1	V2
V0	-1	-1	-1
V1	-1	-1	-1
V2	-1	-1	-1

```

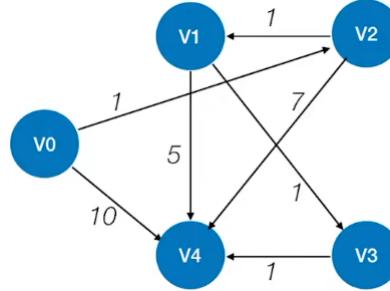
//准备工作，根据图的信息初始化矩阵A和path
for(int k=0;k<n;k++) //考虑以vk作为中转点
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
    {
        if(A[i][j]>A[i][k]+A[k][j]) //以vk为中转点的路径更短
        {
            A[i][j]=A[i][k]+A[k][j]; //更新最短路径长度
            path[i][j]=k; //中转点
        }
    }
}

```

时间复杂度, $O(|V|^3)$

空间复杂度, $O(|V|^2)$

Floyd算法实例

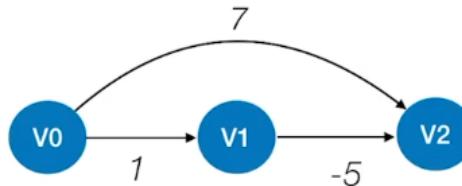


```

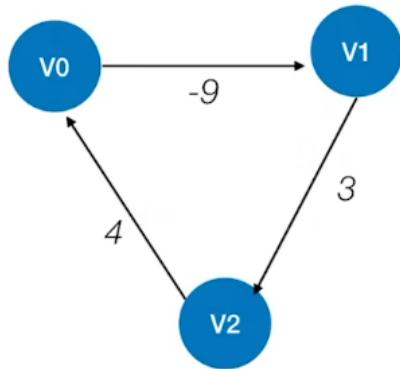
for(int i=0;i<n;i++) //遍历整个矩阵，i为行号，j为列号
    for(int j=0;j<n;j++)
{
    if(A[i][j]>A[i][k]+A[k][j]) //以vk为中转点的路径更短
    {
        A[i][j]=A[i][k]+A[k][j]; //更新最短路径长度
        path[i][j]=k; //中转点
    }
}

```

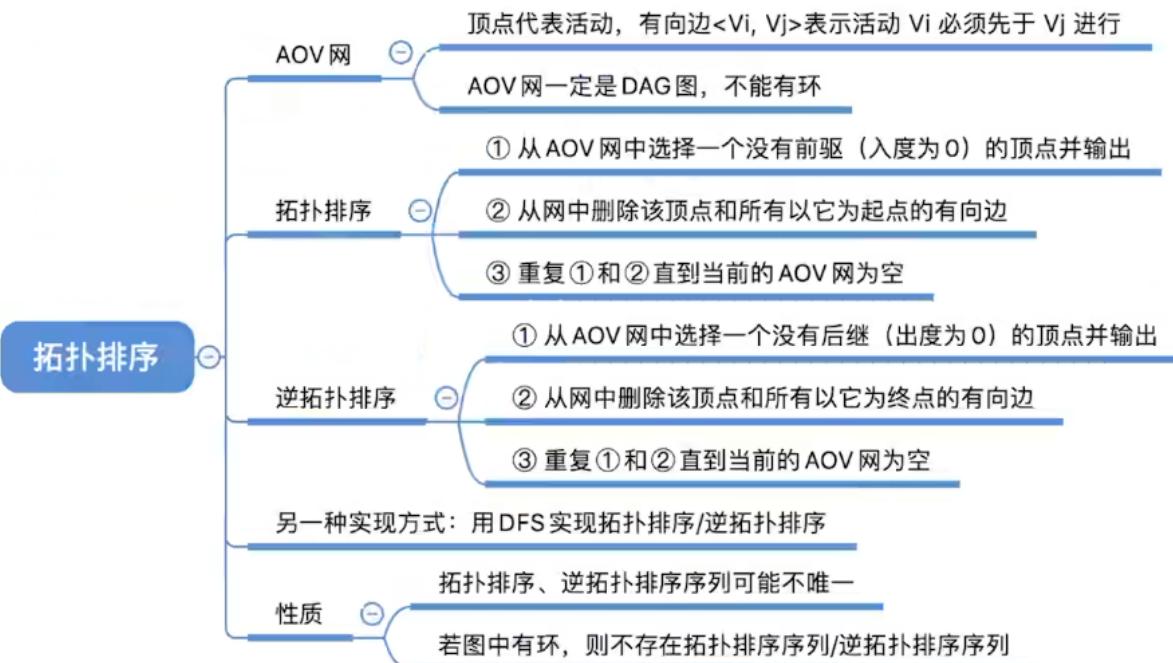
- Floyd算法可以用于负权值带权图



- Floyd算法不能解决带有“负权回路”的图（有负权值的边组成回路），这种图有可能没有最短路径

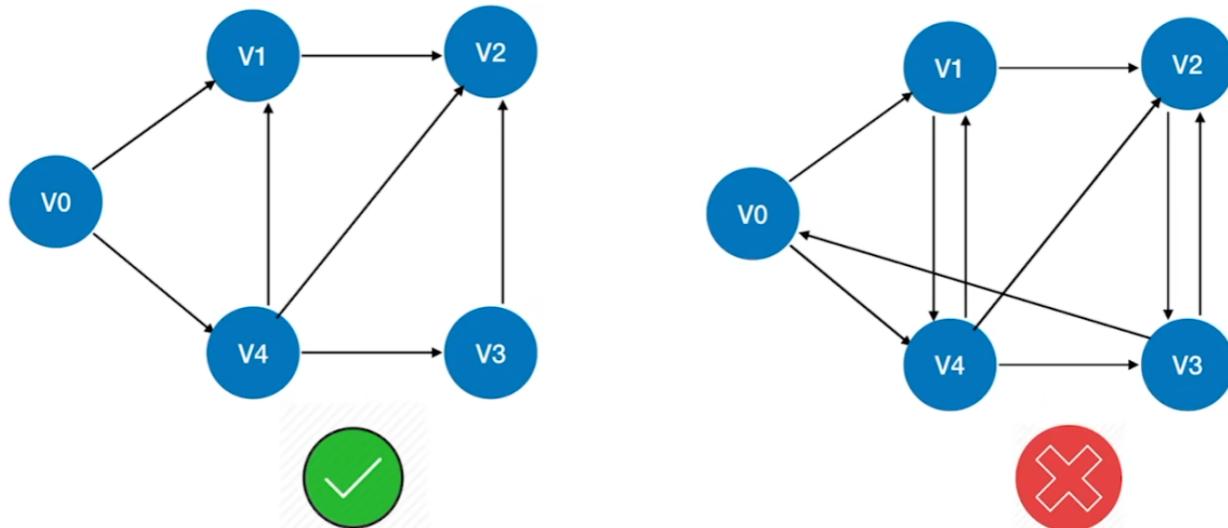


有向无环图 (DAG)



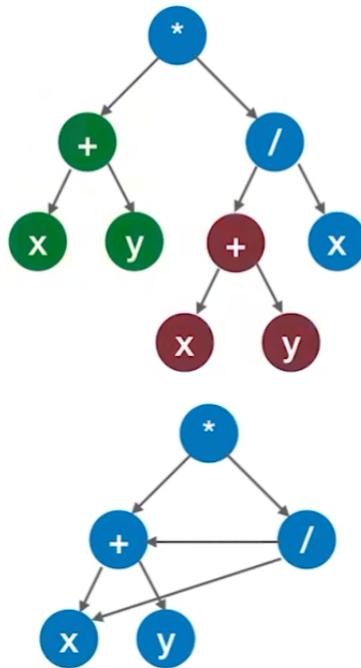
有向无环图 (DAG)

若一个有向图中不存在环，则称为有向无环图，简称DAG图 (Directed Acyclic Graph)



有向无环图 (DAG) 描述表达式

$$(x + y)((x + y)/x)$$



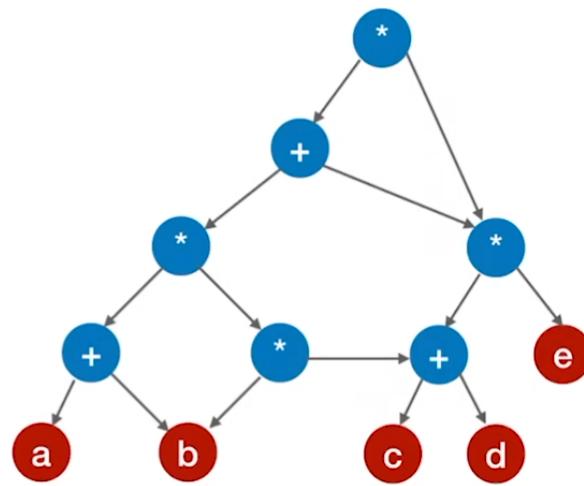
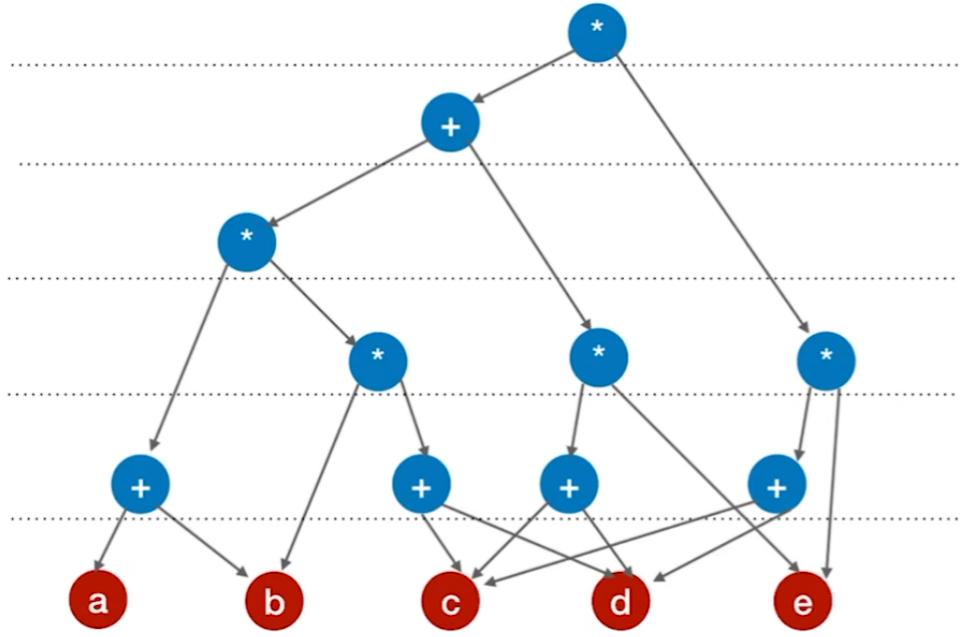
顶点中不可能出现重复的操作数

解题方法

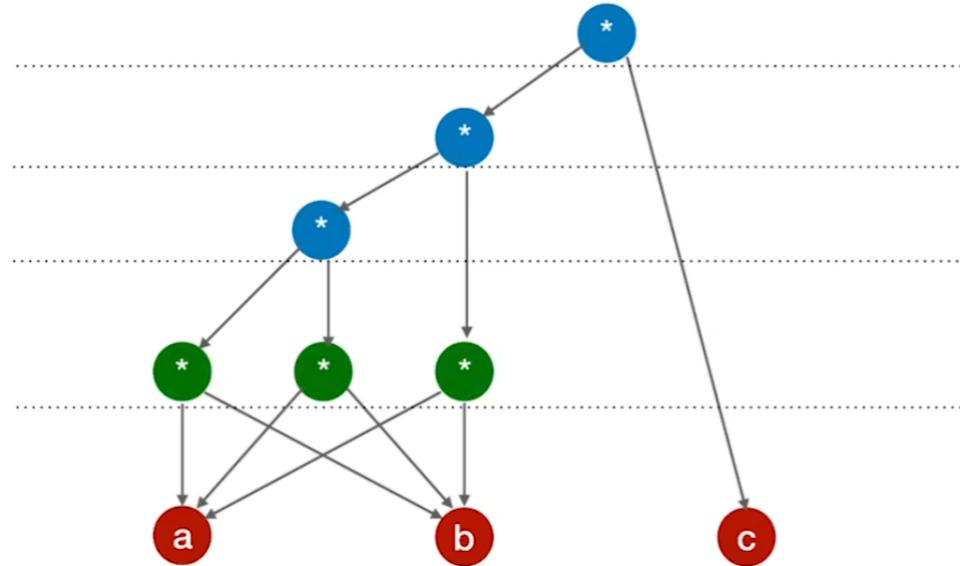
$$((a + b) * (b * (c + d)) + (c + d) * e) * ((c + d) * e)$$

1. 把各个操作数不重复地排成一排
2. 标出各个运算符的生效顺序 (先后顺序有点出入无所谓)
3. 按顺序加入运算符, 注意“分层”
4. 从底向上逐层检查同层的运算符是否可以合体

$$((a + b) * (b * (c + d)) + (c + d) * e) * ((c + d) * e)
① ④ ③ ② ⑦ ⑤ ⑥ ⑩ ⑧ ⑨$$



$$(a * b) * (a * b) * (a * b) * c$$



拓扑排序

拓扑排序：在图论中，由一个有向无环图的顶点组成的序列，当且仅当满足下列条件时，称为该图的一个拓扑排序：

1. 每个顶点出现且只出现一次。
2. 若顶点A在序列中排在顶点B的前面，则在图中不存在从顶点B到顶点A的路径。

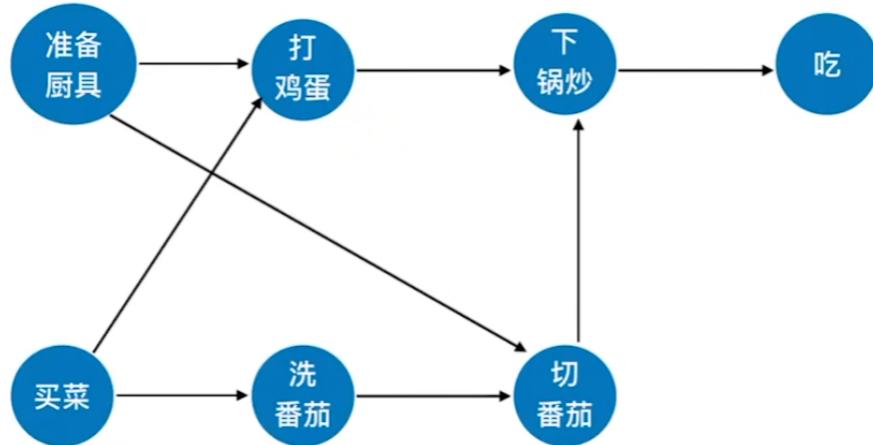
或定义为：拓扑排序是对有向无环图的顶点的一种排序，它使得若存在一条从顶点A到顶点B的路径，则在排序中顶点B出现在顶点A的后面。每个AOV网都有一个或多个拓扑排序序列。

AOV网

AOV网（Activity On Vertex NetWork, 用顶点表示活动的网）

用DAG（有向无环图）表示一个工程。

顶点表示活动，有向边 $< V_i, V_j >$ 表示活动 V_i 必须先于活动 V_j 进行



拓扑排序

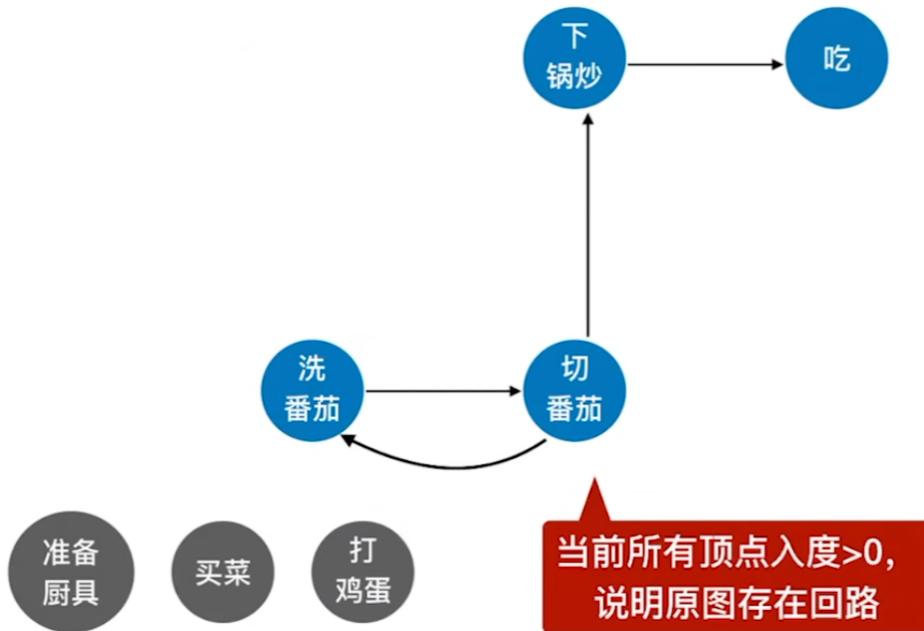
拓扑排序：找到做事的先后顺序



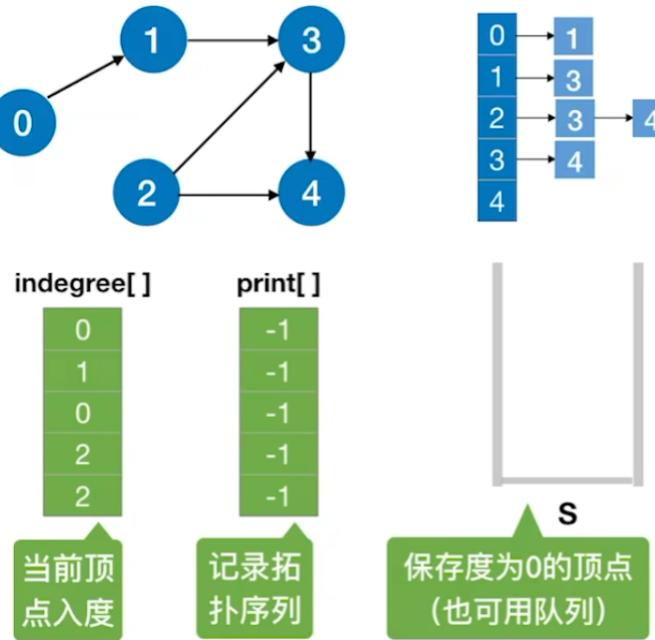
拓扑排序的实现：

1. 从AOV网中选择一个没有前驱（入度为0）的顶点并输出
2. 从网中删除该顶点和所有以它为起点的有向边
3. 重复1和2直到当前的AOV网为空或当前网中不存在无前驱的顶点为止（说明有回路）

对有回路的图进行拓扑排序



代码实现



```
#define MaxVertexNum 100 //图中顶点数目的最大值
typedef struct ArcNode{//边表结点
{
    int adjvex; //该弧所指向的顶点的位置
    struct ArcNode *nextarc; //指向第一条依附该顶点的弧的指针
    //InfoType info;//网的边权值
}ArcNode;
typedef struct VNode{//顶点表结点
{
    VertexType data; //顶点信息
    ArcNode *firstarc; //指向第一条依附该顶点的弧的指针
}VNode,AdjList[MaxVertexNum];
typedef struct
```

```

{
    AdjList vertices; //邻接表
    int vexnum, arcnum; //图的顶点数和弧数
}Graph; //Graph是以邻接表存储的图类型

```

```

bool TopologicalSort(Graph G)
{
    InitStack(S); //初始化栈,存储入度为0的顶点
    for(int i=0;i<G.vexnum;i++)
        if(indegree[i]==0)
            Push(S,i); //将所有入度为0的顶点进栈
    int count=0; //计数,记录当前已经输出的顶点数
    while(!IsEmpty(S)) //栈不空,则存在入度为0的顶点
    {
        Pop(S,i); //栈顶元素出栈,每个顶点都需要处理一次
        print[count++]=i; //输出顶点i
        for(p=G.vertices[i].firstarc;p;p=p->nextarc)
        {
            //将所有i指向的顶点的入度减为1,并且将入度减为0的顶点压入栈S
            v=p->adjvex; //每条边都需要处理一次
            if(!(--indegree[v]))
                Push(S,v); //入度为0,则入栈
        }
    }
    if(count<G.vexnum)
        return false; //排序失败,有向图中有回路
    else
        return true; //拓扑排序成功
}

```

时间复杂度： $O(|V| + |E|)$

若采用邻接矩阵, 则需 $O(|V|^2)$

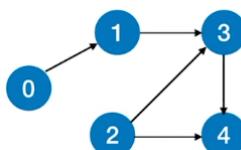
逆拓扑排序

对一个AOV网, 如果采用下列步骤进行排序, 则称之为逆拓扑排序:

1. 从AOV网中选择一个没有后继 (出度为0) 的顶点并输出。
2. 从网中删除该顶点和所有以它为终点的有向边。
3. 重复1和2直到当前的AOV网为空。

使用不同的存储结构对时间复杂度的影响

逆拓扑排序的实现 (DFS算法)





```

void DFSTraverse(Graph G) //对图进行深度优先遍历
{
    for(v=0;v<G.vexnum;++v)
        visited[v]=FALSE;//初始化已访问标记数据
    for(v=0;v<G.vexnum;++v)//本代码中是从v=0开始遍历
        if(!visited[v])
            DFS(G,v);
}

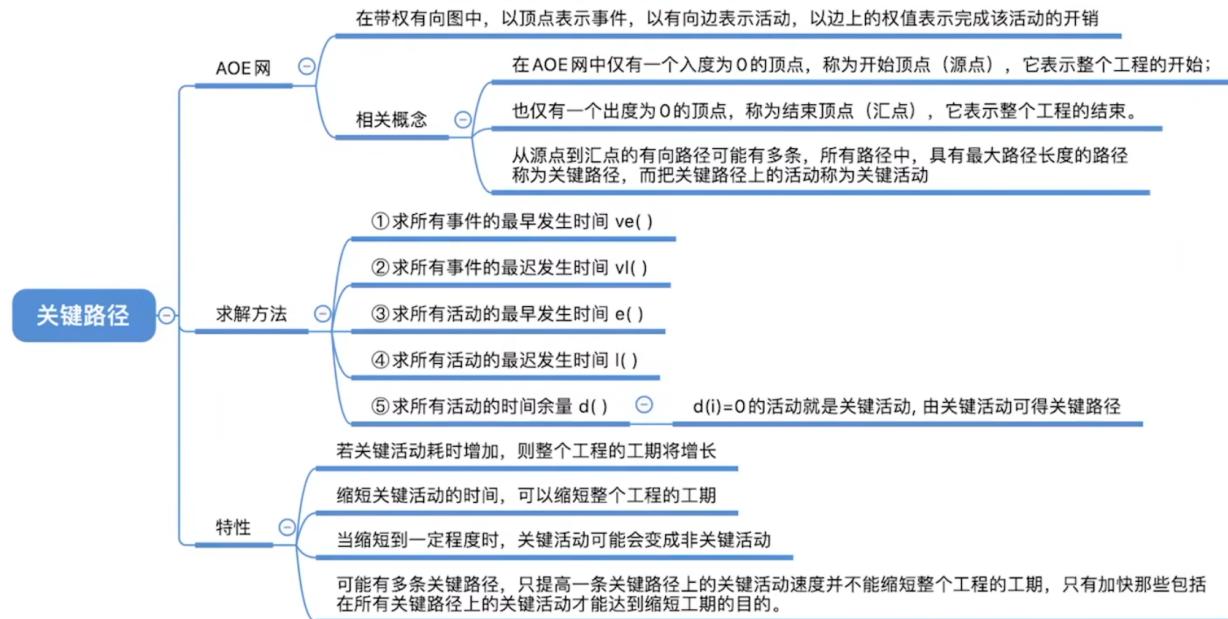
void DFS(Graph G, int v) //从顶点v出发,深度优先遍历图G
{
    visited[v]=TRUE;//设已访问标记
    for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w))
        if(!visited[w])//w为u的尚未访问的邻接顶点
            DFS(G,w);
    print(v);//输出顶点
}

```

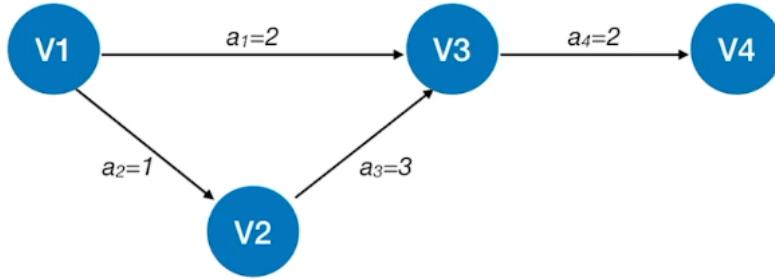
思考：如果存在回路，则不存在逆拓扑排序序列，如何判断回路？

DFS实现逆拓扑排序在顶点退栈前输出

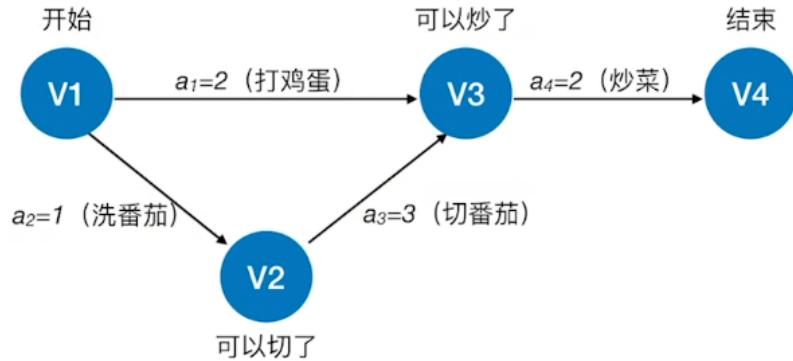
关键路径



AOE网



在带权有向图中，以顶点表示时间，以有向边表示活动，以边上的权值表示完成该活动的开销（如完成活动所需的时间），称之为用边表示活动的网络，简称AOE网（Activity On Edge Network）



AOE网具有以下两个性质：

- 只有在某顶点所代表的事件发生后，从该顶点出发的各有向边所代表的活动才能开始；
- 只有在进入某顶点的各有向边代表的活动都已结束时，该顶点所代表的事件才能发生。另外，有些活动是可以并行进行的

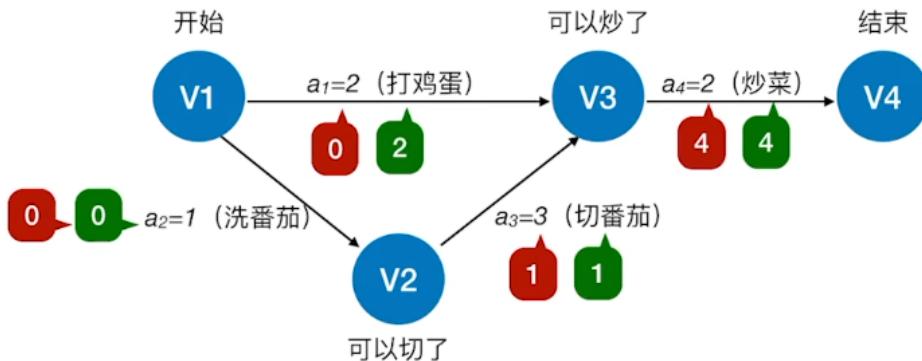
在AOE网中仅有一个入度为0的顶点，称为开始顶点（源点），它表示整个工程的开始；

也仅有一个出度为0的顶点，称为结束顶点（汇点），它表示整个工程的结束。

关键路径

从源点到汇点的有向路径可能有多条，所有路径中，具有最大路径长度的路径称为关键路径，而把关键路径上的活动称为关键活动

完成整个工程的最短时间就是关键路径的长度，若关键活动不能按时完成，则整个工程的完成时间就会延长



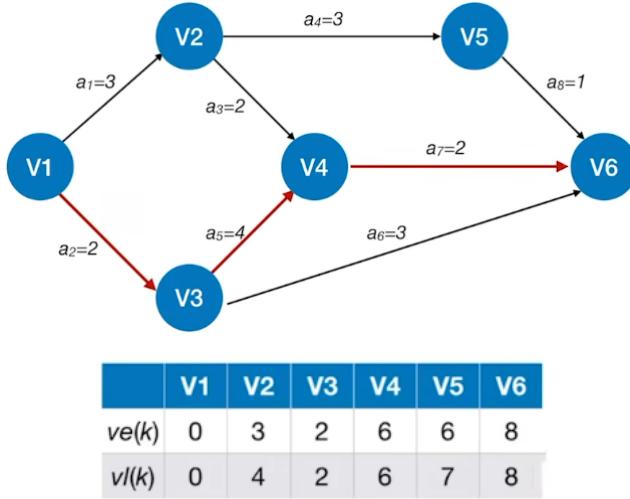
事件 v_k 的最早发生时间 $ve(k)$: 决定了所有从 v_k 开始的活动能够开工的最早时间
活动 a_i 的最早开始时间 $e(i)$: 指该活动弧的起点所表示的事件的最早发生时间

事件 v_k 的最迟发生时间 $vl(k)$: 它是指在不推迟整个工程完成的前提下, 该事件最迟必须发生的时间
活动 a_i 的最迟开始时间 $l(i)$: 它是指该活动弧的终点所表示事件的最迟发生时间与该活动所需时间之差

活动 a_i 的时间余量 $d(i) = l(i) - e(i)$, 表示在不增加完成整个工程所需总时间的情况下, 活动 a_i 可以拖延的时间
若一个活动的时间余量为零, 则说明该活动必须要如期完成, $d(i) = 0$ 即 $l(i) = e(i)$ 的活动 a_i 是关键活动

由关键活动组成的路径就是关键路径

求关键路径的步骤



	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$e(k)$	0	0	3	3	2	2	6	6
$l(k)$	1	0	4	4	2	5	6	7
$d(k)$	1	0	1	1	0	3	0	1

1. 求所有事件的最早发生时间 $ve()$

按拓扑排序序列, 依次求各个顶点的 $ve(k)$:

$$ve(\text{源点}) = 0$$

$$ve(k) = \max\{ve(j) + Weight(v_j, v_k)\}, v_j \text{ 为 } v_k \text{ 的任意前驱}$$

2. 求所有事件的最迟发生时间 $vl()$

按逆拓扑排序序列, 依次求各个顶点的 $vl(k)$:

$$vl(\text{汇点}) = ve(\text{汇点})$$

$$vl(k) = \min\{vl(j) - Weight(v_k, v_j)\}, v_j \text{ 为 } v_k \text{ 的任意后继}$$

3. 求所有活动的最早发生时间 $e()$

若边 $< v_k, v_j >$ 表示活动 a_i , 则有 $e(i) = ve(k)$

4. 求所有活动的最迟发生时间 $l()$

若边 $< v_k, v_j >$ 表示活动 a_i , 则有 $l(i) = vl(j) - Weight(v_k, v_j)$

5. 求所有活动的时间余量 $d()$

$$d(i) = l(i) - e(i)$$

$d(k)=0$ 的活动就是关键活动，由关键活动可得关键路径

关键活动: a2 a5 a7

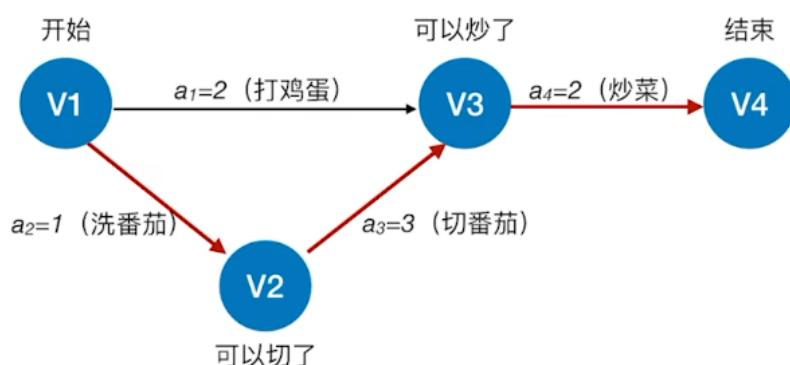
关键路径: v1->v3->v4->v6

关键活动、关键路径的特性

若关键活动耗时增加，则整个工程的工期将增长

缩短关键活动的时间，可以缩短整个工程的工期

当缩短到一定程度时，关键活动可能会变成非关键活动



可能有多条关键路径，只提高一条关键路径上的关键活动速度并不能缩短整个工程的工期，只有加快那些包括所有关键路径上的关键活动才能达到缩短工期的目的。