

# 拓扑排序

# 预备知识

- 全序: Total order
  - 集合  $X$  上的全序关系 (Total order) , 简称全序、又名线性序 (linear order) 、简单序 (simple order) , 或 (非严格) 排序 ((non-strict) ordering) , 是在  $X$  上的反对称的、传递的和完全的任何二元关系。
  - 集合中的任何一对元素都是可相互比较的。

若  $X$  满足全序关系, 则下列陈述对于  $X$  中的所有  $a, b$  和  $c$  成立:

- 反对称性: 若  $a \leq b$  且  $b \leq a$  则  $a = b$
- 传递性: 若  $a \leq b$  且  $b \leq c$  则  $a \leq c$
- 完全性:  $a \leq b$  或  $b \leq a$

# 预备知识

- 偏序: Partial order

## 严格偏序, 反自反偏序 [编辑]

给定集合 $S$ , “ $<$ ”是 $S$ 上的二元关系, 若“ $<$ ”满足:

1. **反自反性**:  $\forall a \in S$ , 有 $a \not< a$ ;
2. **非对称性**:  $\forall a, b \in S$ ,  $a < b \Rightarrow b \not< a$ ;
3. **传递性**:  $\forall a, b, c \in S$ ,  $a < b$ 且 $b < c$ , 则 $a < c$ ;

则称“ $<$ ”是 $S$ 上的严格偏序或反自反偏序。

- 全序 $T$ 是偏序 $P$ 的线性扩展, 只要 $x \leq y$ 在 $P$ 中成立则 $x \leq y$ 在 $T$ 中也成立。
- 在计算机科学中, 找到偏序的线性扩展的算法叫做拓扑排序

# 预备知识

- 偏序:

VS

- 全序:

## 严格偏序, 反自反偏序 [编辑]

给定集合 $S$ , “ $<$ ”是 $S$ 上的二元关系, 若“ $<$ ”满足:

1. **反自反性**:  $\forall a \in S$ , 有 $a \not< a$ ;
2. **非对称性**:  $\forall a, b \in S$ ,  $a < b \Rightarrow b \not< a$ ;
3. **传递性**:  $\forall a, b, c \in S$ ,  $a < b$ 且 $b < c$ , 则 $a < c$ ;

则称“ $<$ ”是 $S$ 上的严格偏序或反自反偏序。

若 $X$ 满足全序关系, 则下列陈述对于 $X$ 中的所有 $a, b$ 和 $c$ 成立:

- **反对称性**: 若 $a \leq b$ 且 $b \leq a$ 则 $a = b$
- **传递性**: 若 $a \leq b$ 且 $b \leq c$ 则 $a \leq c$
- **完全性**:  $a \leq b$ 或 $b \leq a$

- 全序 $T$ 是偏序 $P$ 的线性扩展, 只要 $x \leq y$ 在 $P$ 中成立则 $x \leq y$ 在 $T$ 中也成立。
- 在计算机科学中, 找到偏序的线性扩展的算法叫做**拓扑排序**

# 应用

```
static std::vector<std::string> staticData;
std::vector<std::string> initializeStaticData ()
{
    std::vector<std::string> vec;
    vec.push_back ("initialize");

    return vec;
}

void foo()
{
    static std::once_flag oc;
    std::call_once(oc, [] { staticData = initializeStaticData ();});
}
```

- 在多线程的环境下，有些时候我们不需要某给函数被调用多次或者某些变量被初始化多次，它们仅仅只需要被调用一次或者初始化一次即可。
- 若该调用正常返回（这种对 call\_once 的调用被称为返回），则翻转 flag，并保证以同一 flag 对 call\_once 的其他调用为消极
- 同一 flag 上的所有积极调用组成单独全序，它们由零或多个异常调用后随一个返回调用组成

# 6. 图

## 深度优先搜索DFS

**KISS:** Keep it simple, stupid.

- K. Johnson

邓俊辉

deng@tsinghua.edu.cn

## 算法

❖  $\text{DFS}(s)$  //始自顶点s的深度优先搜索 (Depth-First Search)

访问顶点s

若s尚有未被访问的邻居，则任取其一u，递归执行 $\text{DFS}(u)$

否则，返回

❖ 若此时图中尚有顶点未被访问 //何时出现这一情况？

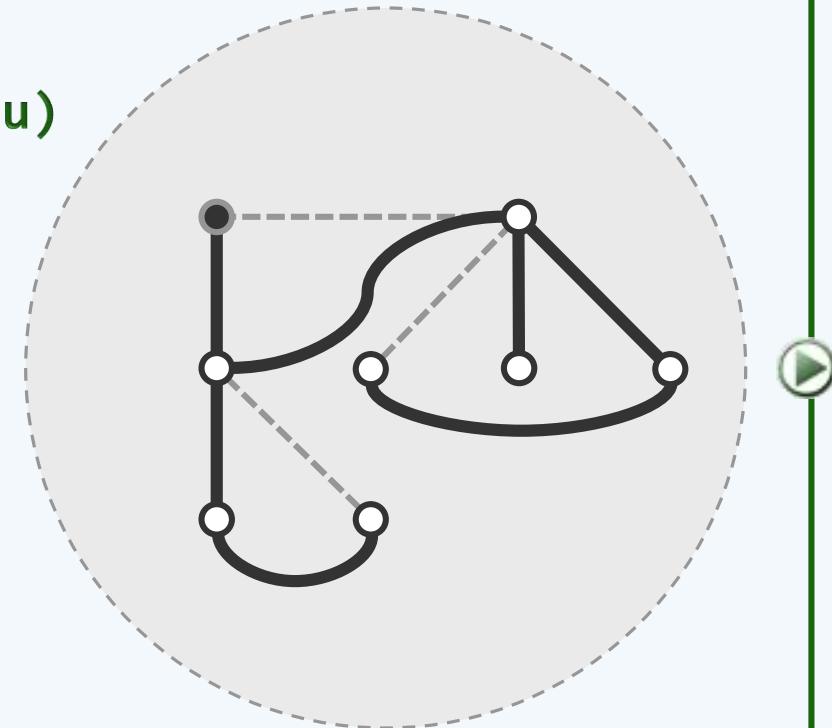
任取这样的一个顶点作起始点

重复上述过程

直至所有顶点都被访问到

❖ 等效于树的先序遍历

事实上，DFS也的确会构造出原图的一棵支撑树 (DFS tree)



## Graph::DFS()

❖ template <typename Tv, typename Te> //顶点类型、边类型

```
void Graph<Tv, Te>::DFS( int v, int & clock ) {
```

```
dTime(v) = ++clock; status(v) = DISCOVERED; //发现当前顶点v
```

```
for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v的每一邻居u
```

```
/* ... 视u的状态，分别处理 ... */
```

```
/* ... 与BFS不同，含有递归 ... */
```

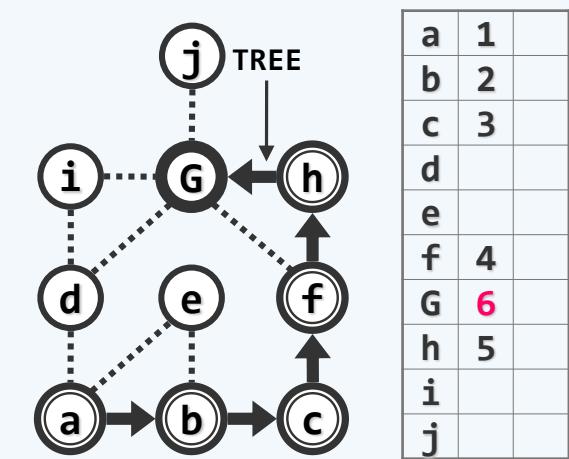
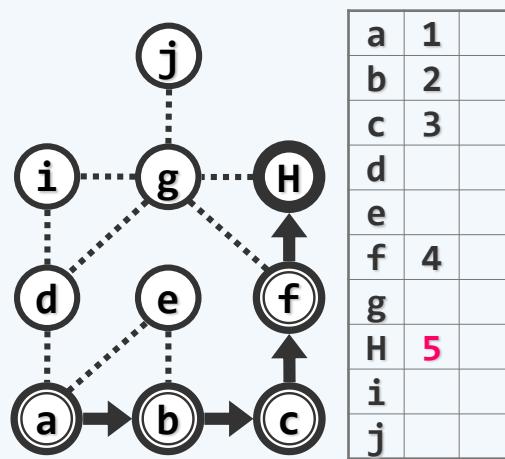
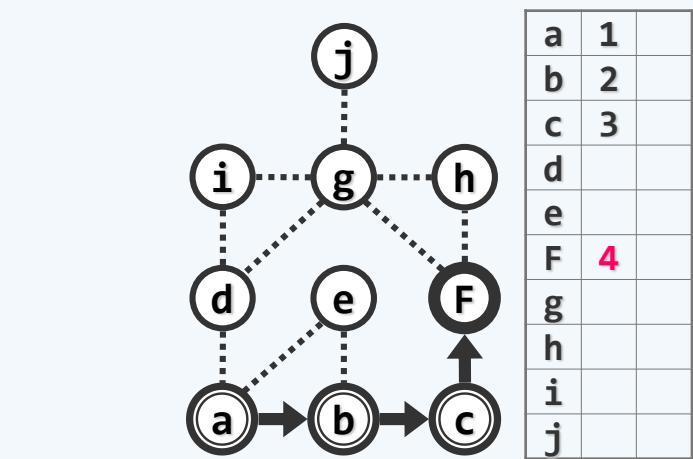
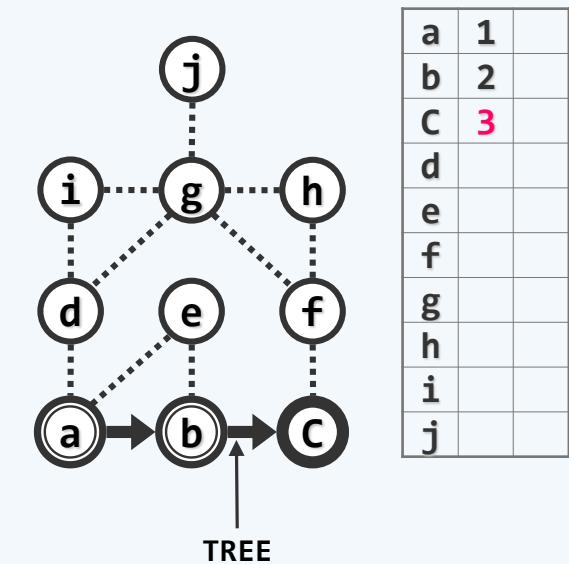
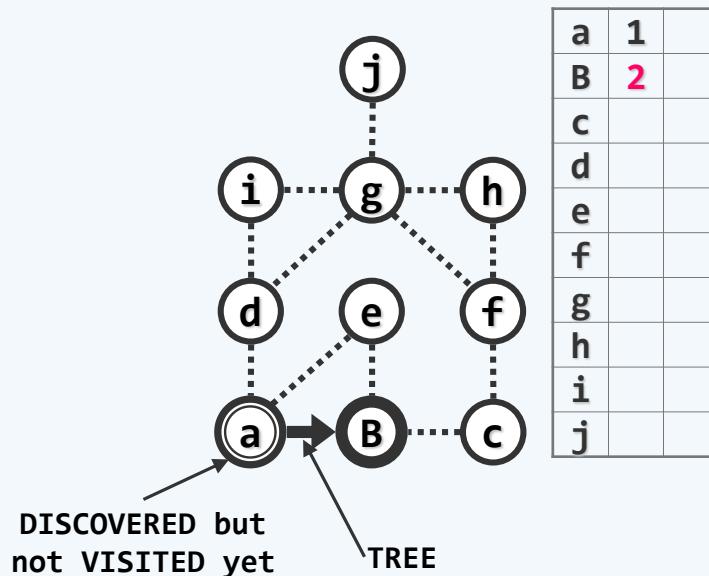
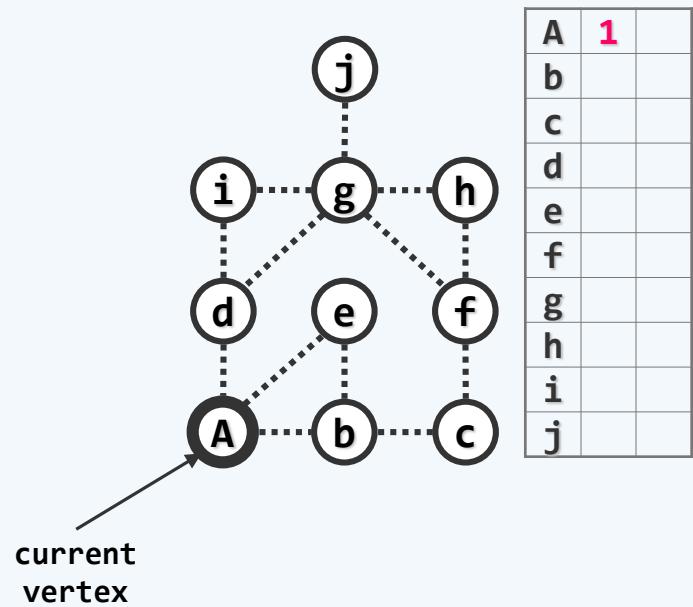
```
status(v) = VISITED; fTime(v) = ++clock; //至此，当前顶点v方告访问完毕
```

```
}
```

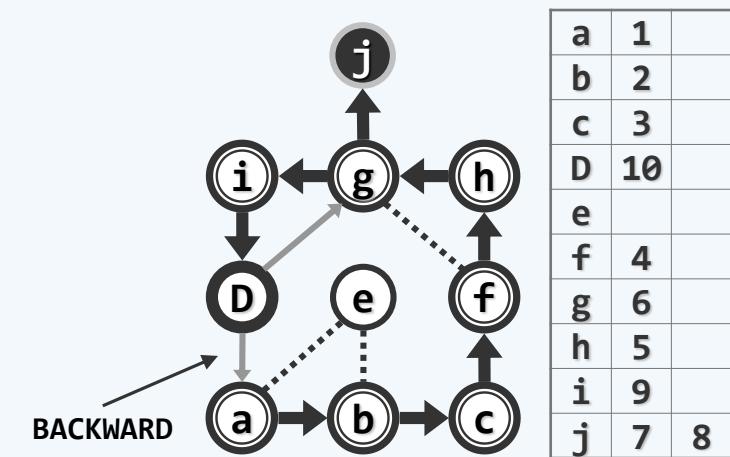
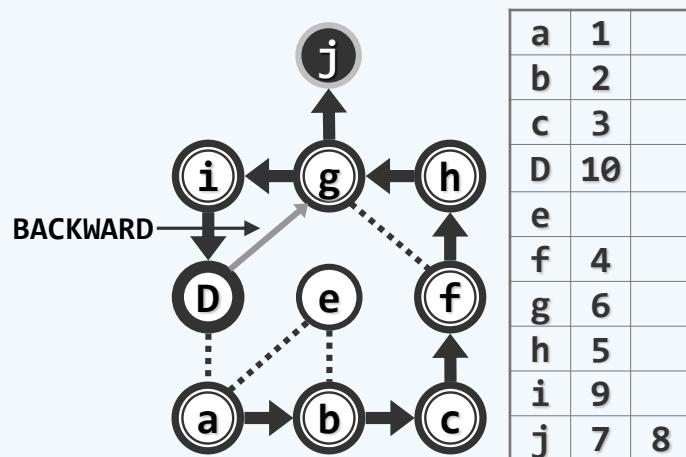
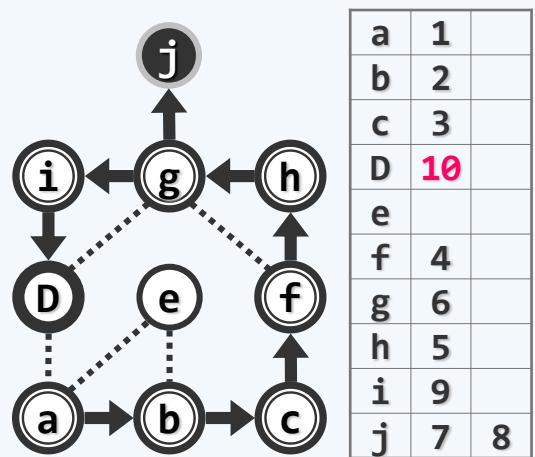
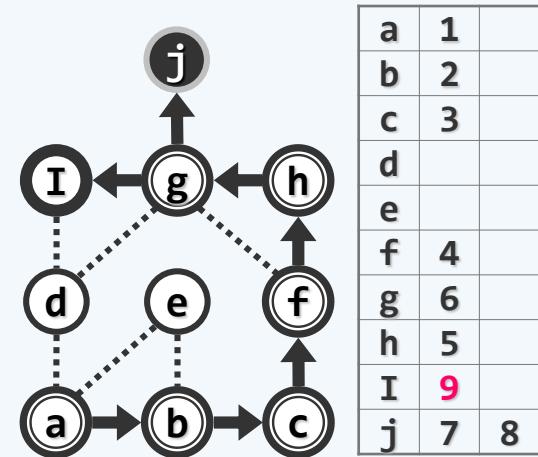
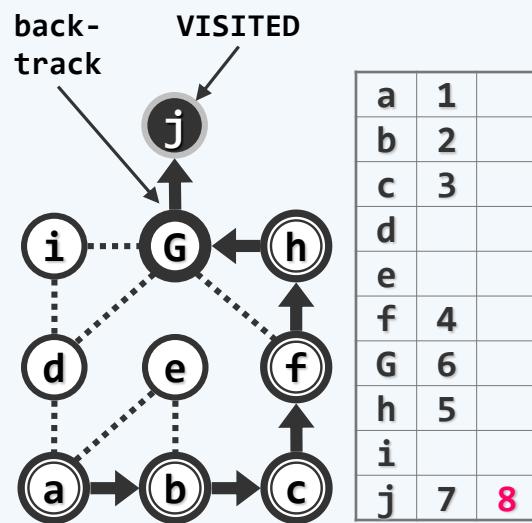
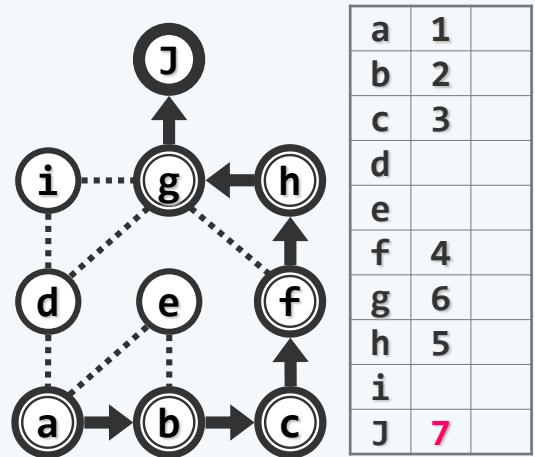
## Graph::DFS()

```
❖ for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v所有邻居u  
switch ( status(u) ) { //并视其状态分别处理  
    case UNDISCOVERED: //u尚未发现, 意味着支撑树可在此拓展  
        type(v, u) = TREE; parent(u) = v; DFS( u, clock ); break; //递归  
    case DISCOVERED: //u已被发现但尚未访问完毕, 应属被后代指向的祖先  
        type(v, u) = BACKWARD; break;  
    default: //u已访问完毕 (VISITED, 有向图), 则视承袭关系分为前向边或跨边  
        type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS; break;  
} //switch
```

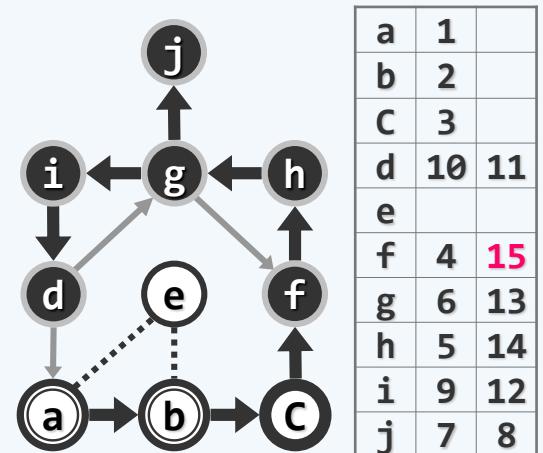
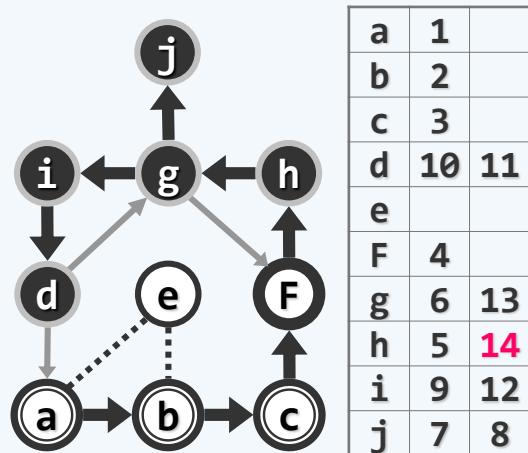
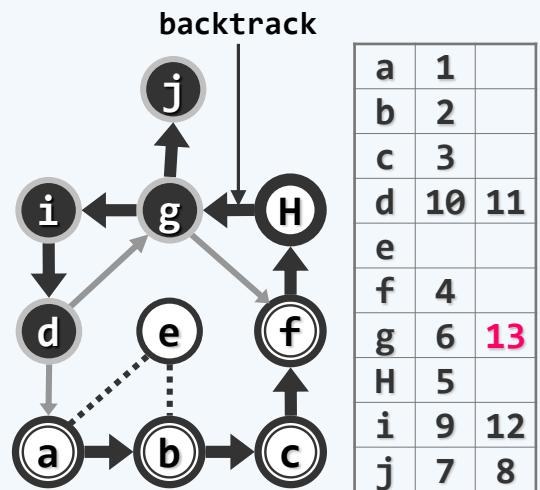
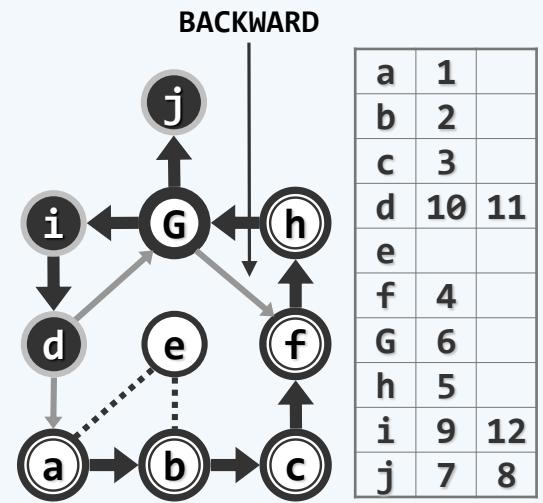
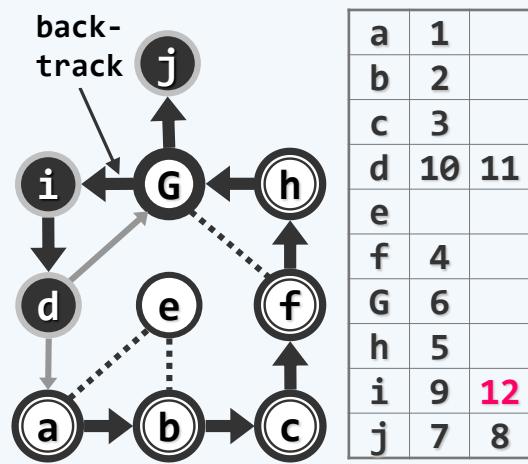
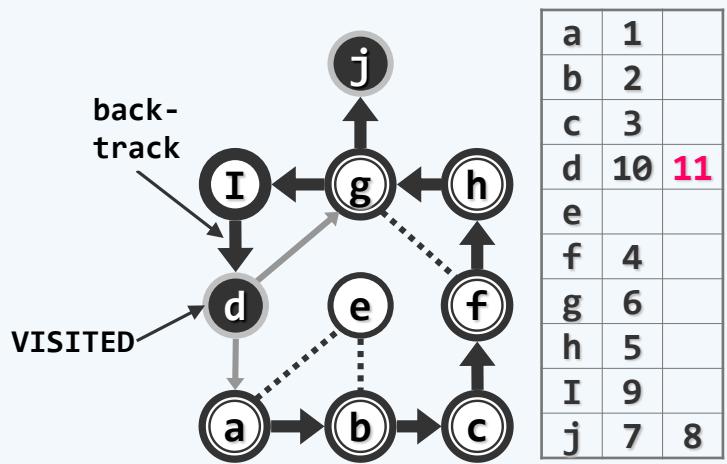
## 实例 (无向图)



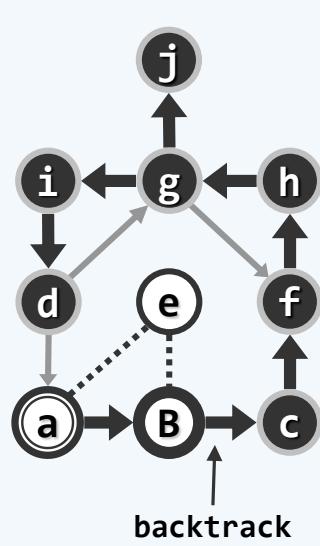
## 实例 (无向图)



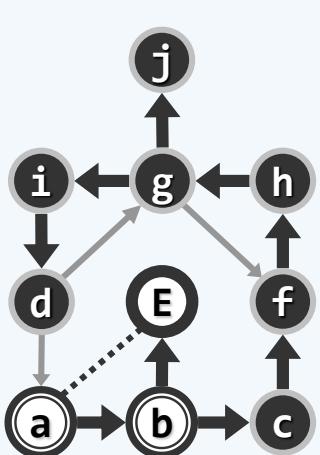
## 实例 (无向图)



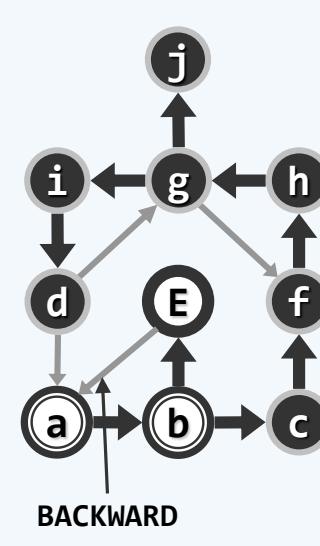
## 实例 (无向图)



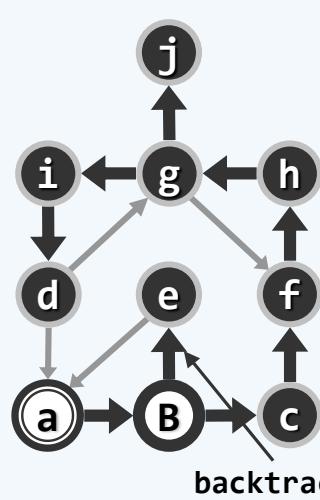
|   |    |    |
|---|----|----|
| a | 1  |    |
| B | 2  |    |
| c | 3  | 16 |
| d | 10 | 11 |
| e |    |    |
| f | 4  | 15 |
| g | 6  | 13 |
| h | 5  | 14 |
| i | 9  | 12 |
| j | 7  | 8  |



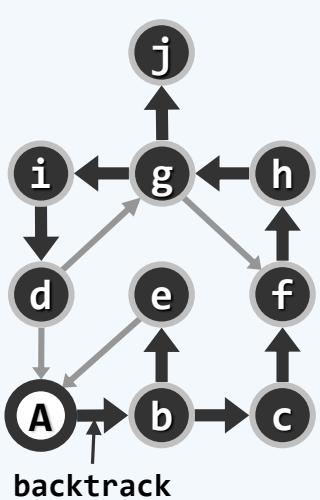
|   |    |    |
|---|----|----|
| a | 1  |    |
| b | 2  |    |
| c | 3  | 16 |
| d | 10 | 11 |
| E | 17 |    |
| f | 4  | 15 |
| g | 6  | 13 |
| h | 5  | 14 |
| i | 9  | 12 |
| j | 7  | 8  |



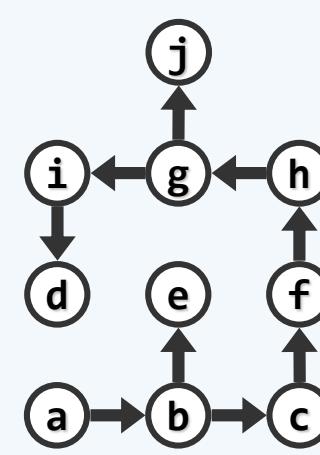
|   |    |    |
|---|----|----|
| a | 1  |    |
| b | 2  |    |
| c | 3  | 16 |
| d | 10 | 11 |
| E | 17 |    |
| f | 4  | 15 |
| g | 6  | 13 |
| h | 5  | 14 |
| i | 9  | 12 |
| j | 7  | 8  |



|   |    |    |
|---|----|----|
| a | 1  |    |
| B | 2  |    |
| c | 3  | 16 |
| d | 10 | 11 |
| e | 17 | 18 |
| f | 4  | 15 |
| g | 6  | 13 |
| h | 5  | 14 |
| i | 9  | 12 |
| j | 7  | 8  |



|   |    |    |
|---|----|----|
| A | 1  |    |
| b | 2  | 19 |
| c | 3  | 16 |
| d | 10 | 11 |
| e | 17 | 18 |
| f | 4  | 15 |
| g | 6  | 13 |
| h | 5  | 14 |
| i | 9  | 12 |
| j | 7  | 8  |

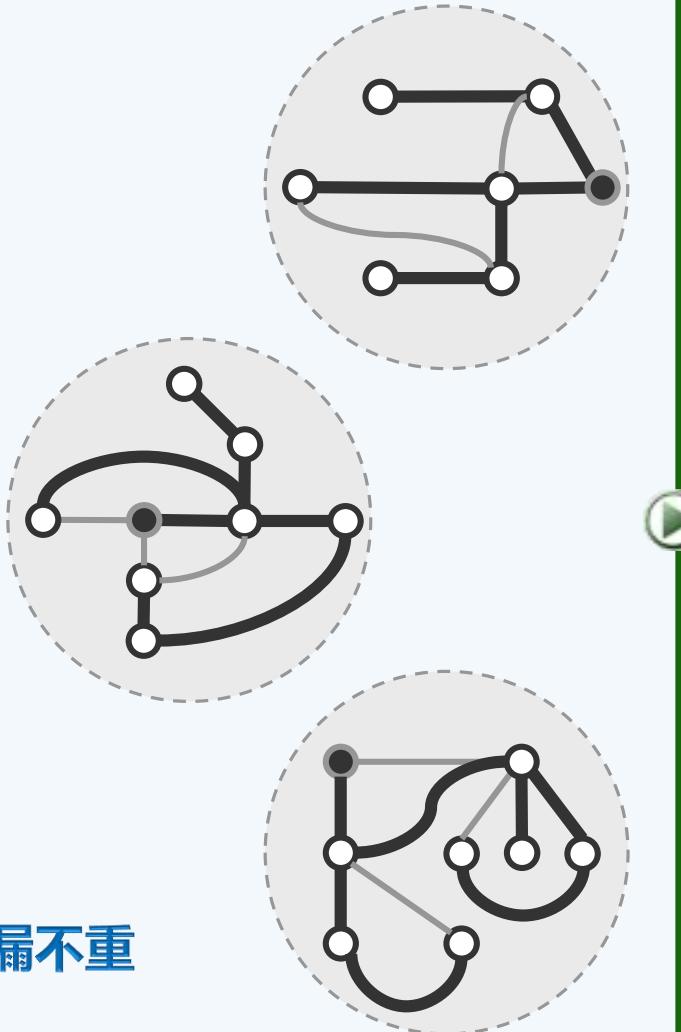


|   |    |    |
|---|----|----|
| a | 1  | 20 |
| b | 2  | 19 |
| c | 3  | 16 |
| d | 10 | 11 |
| e | 17 | 18 |
| f | 4  | 15 |
| g | 6  | 13 |
| h | 5  | 14 |
| i | 9  | 12 |
| j | 7  | 8  |

## Graph::dfs()

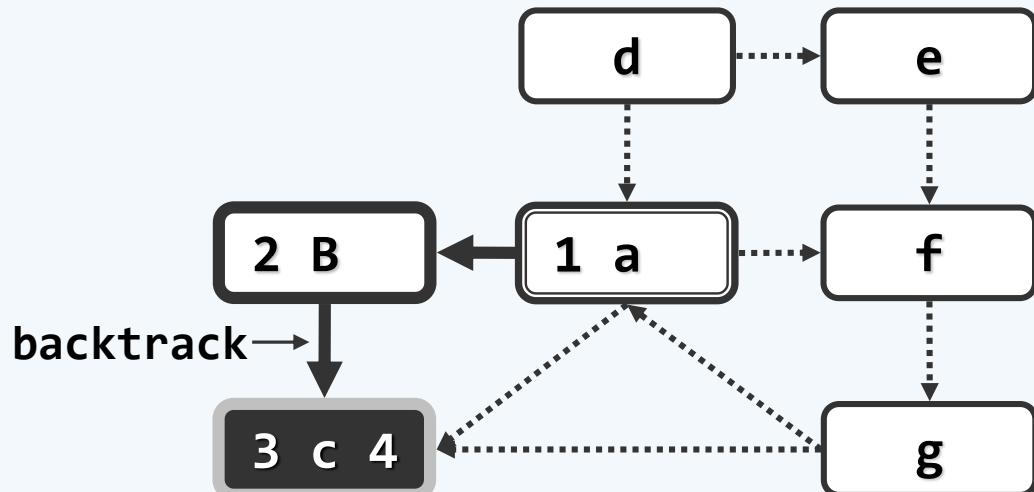
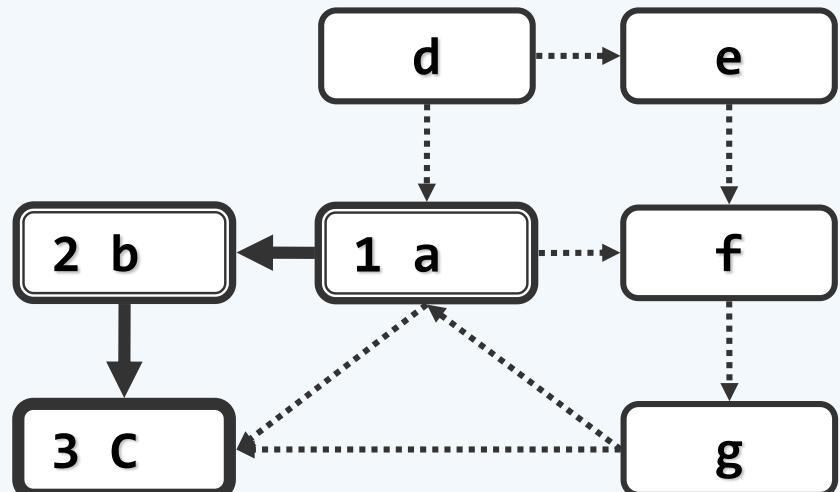
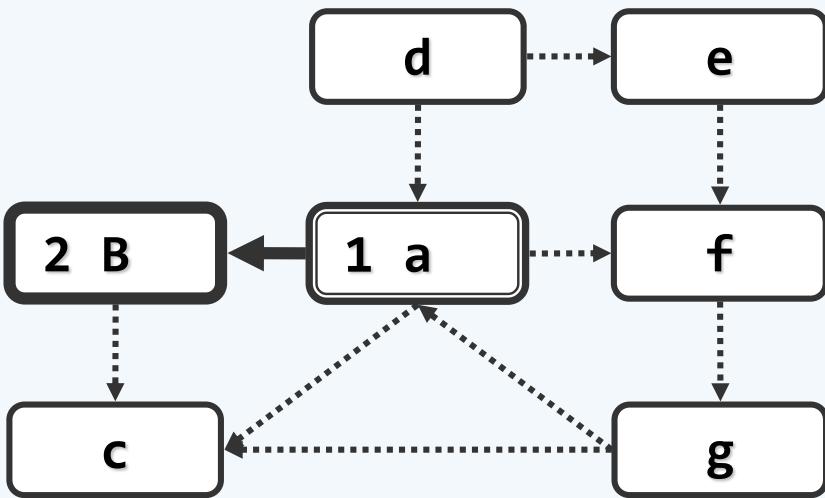
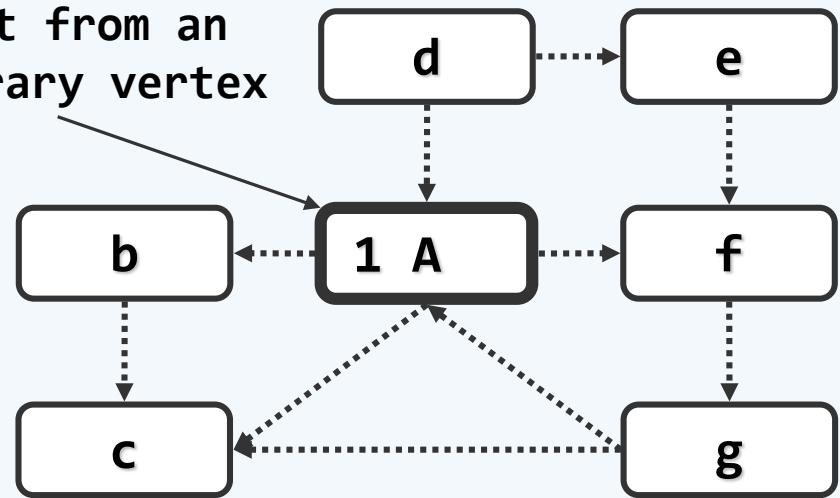
- ❖ 与BFS( $v$ )类似，DFS( $v$ )也可遍历 $v$ 所属分量——若含多个分量呢？
- ❖ 与bfs( $s$ )类似（采用邻接表）， $\text{dfs}(s)$ 也可在累计  $O(n + e)$  时间内  
对于每一连通/可达分量，从其起始顶点 $v$ 进入DFS( $v$ )恰好1次，并  
最终生成一个DFS森林（包含  $c$  棵树、 $n - c$  条树边）
- ❖ 

```
template <typename Tv, typename Te> //顶点类型、边类型
void Graph<Tv, Te>::dfs( int s ) { //s为起始顶点
    reset(); int clock = 0; int v = s; //初始化
    do //逐一检查所有顶点，一旦遇到尚未发现的顶点
        if ( UNDISCOVERED == status(v) )
            DFS( v, clock ); //即从该顶点出发启动一次DFS
    while ( s != ( v = ( ++v % n ) ) ); //按序号访问，故不漏不重
}
```

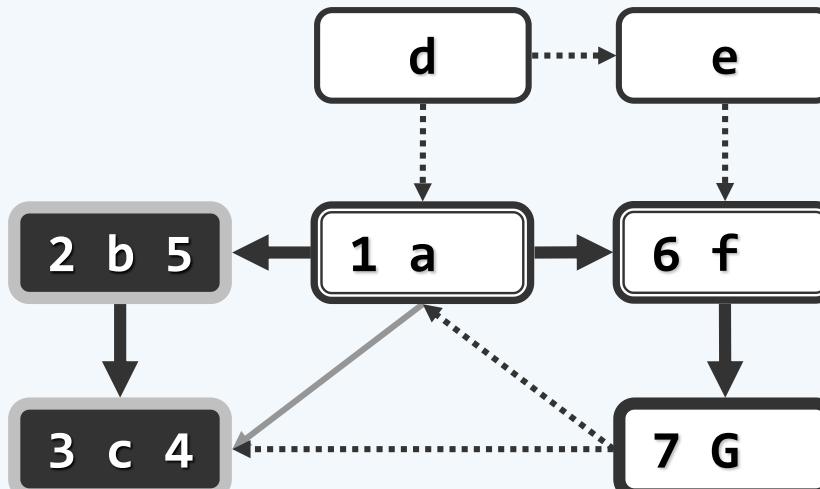
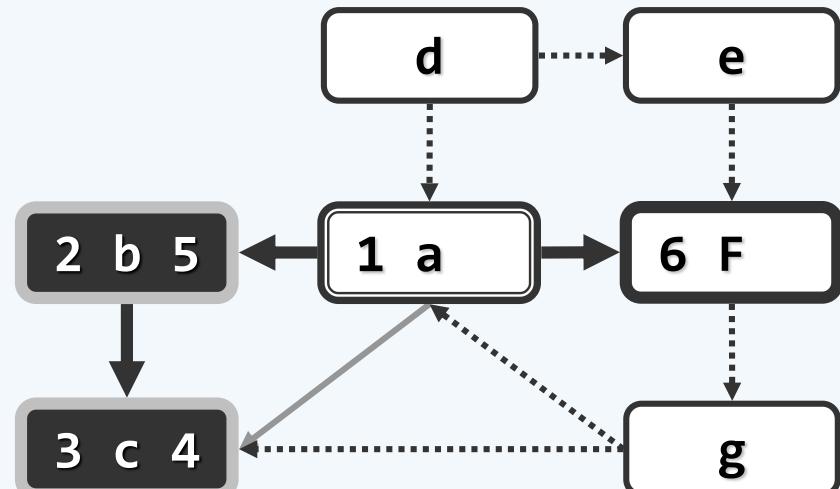
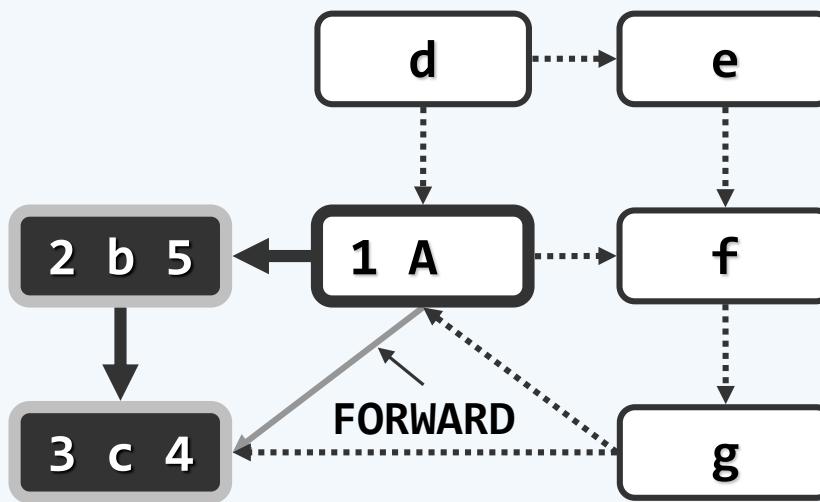
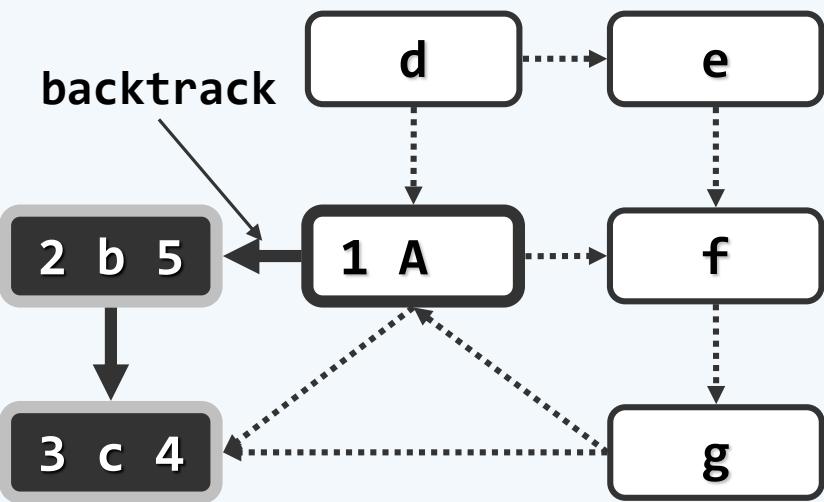


## 实例 (有向图)

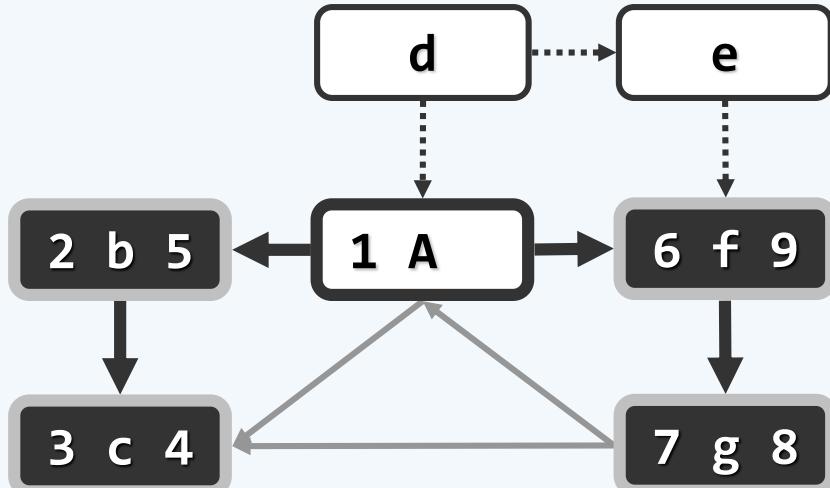
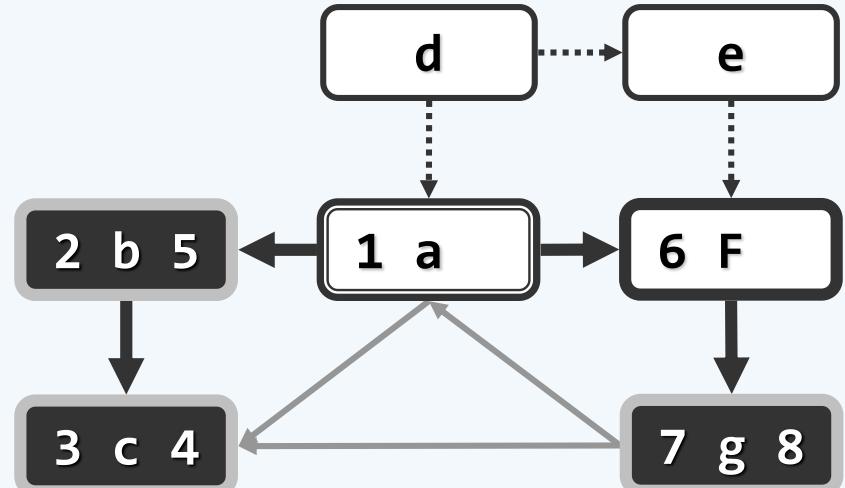
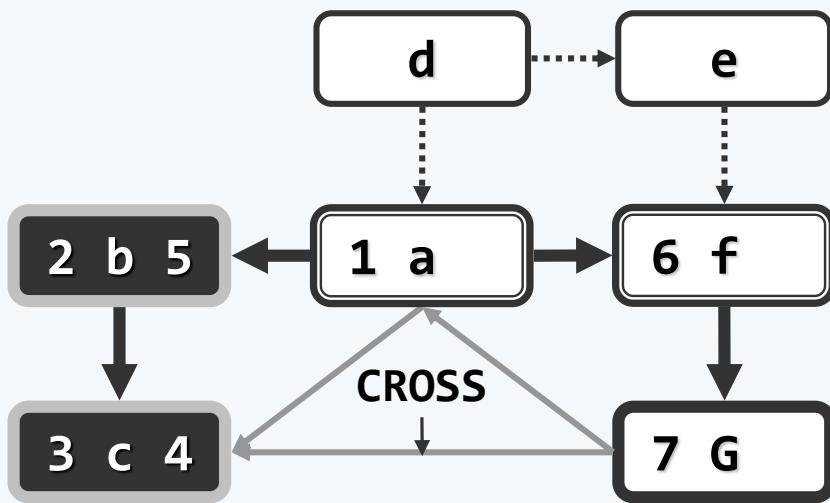
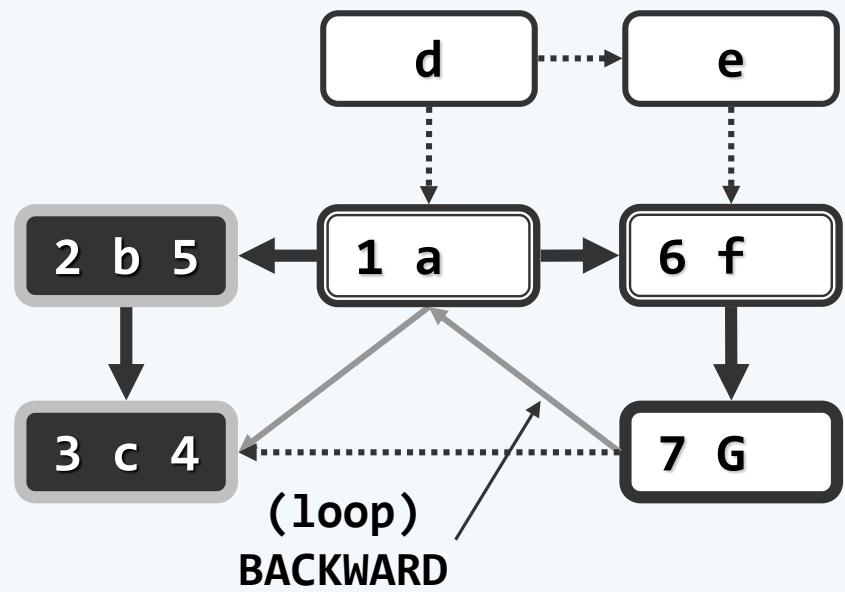
start from an arbitrary vertex



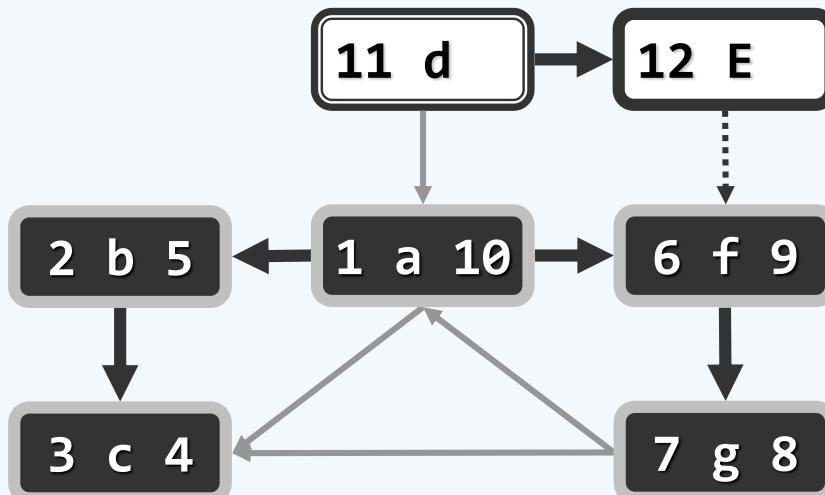
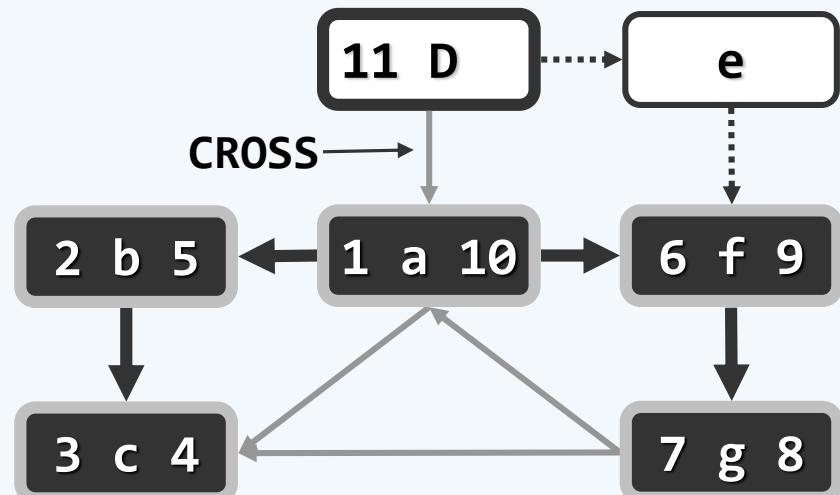
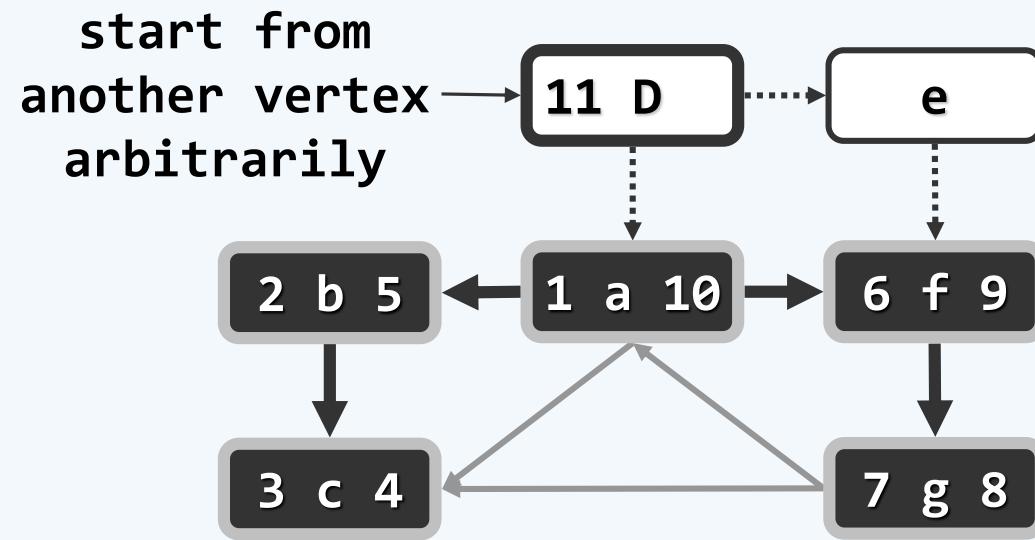
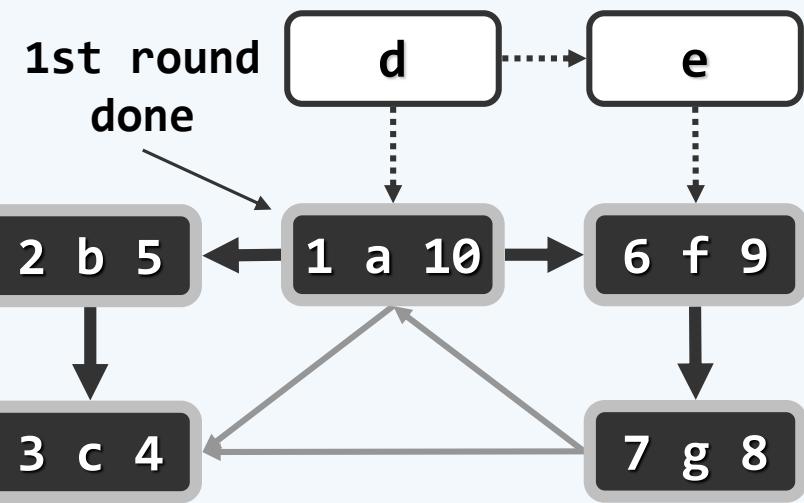
## 实例 (有向图)



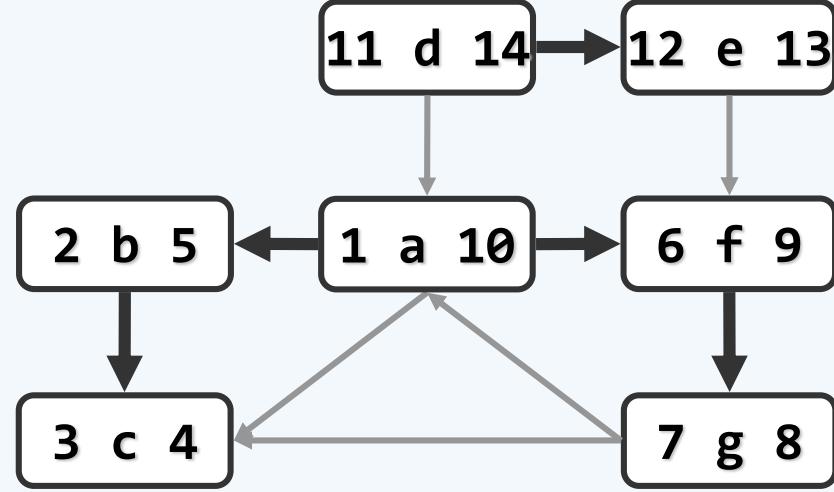
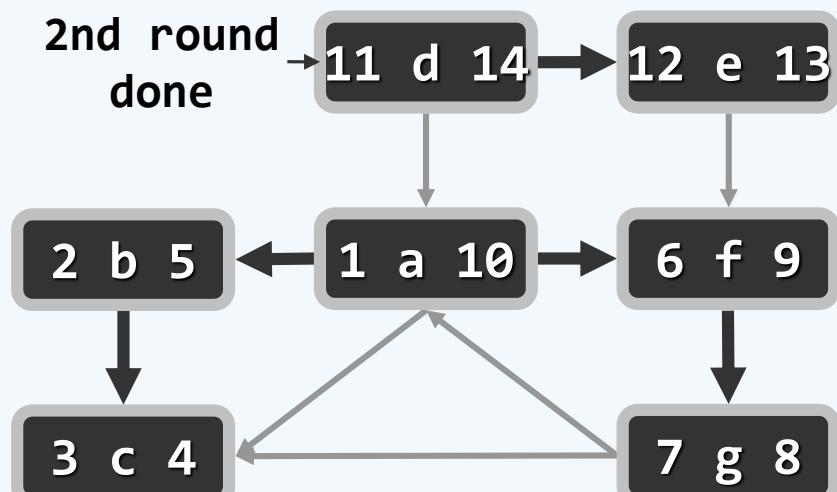
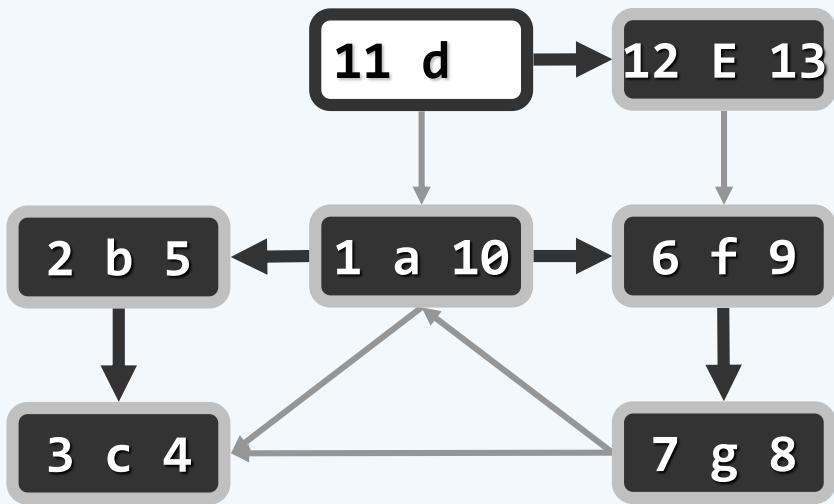
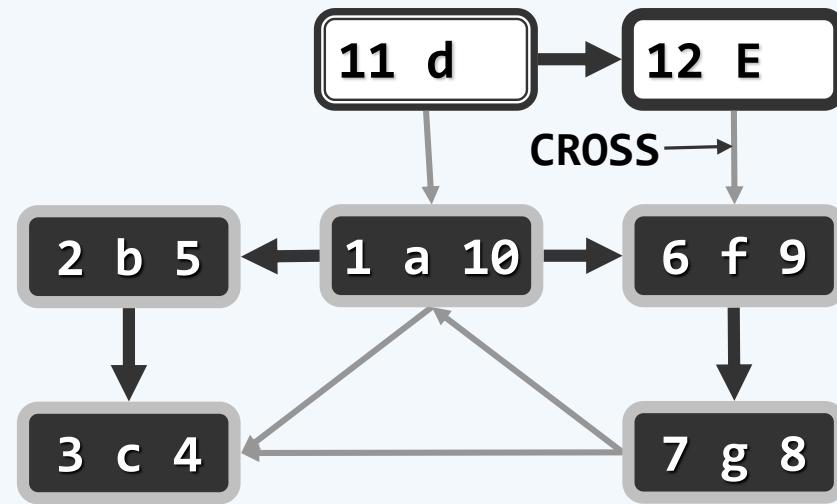
## 实例 (有向图)



## 实例 (有向图)



## 实例 (有向图)



- ❖ 从顶点s出发的DFS

在无向图中将访问与s连通的所有顶点 connected component

在有向图中将访问由s可达的所有顶点 reachable component

- ❖ 经DFS确定的树边，不会构成回路

- ❖ 从s出发的DFS，将以s为根生成一棵DFS树；所有DFS树，进而构成DFS森林

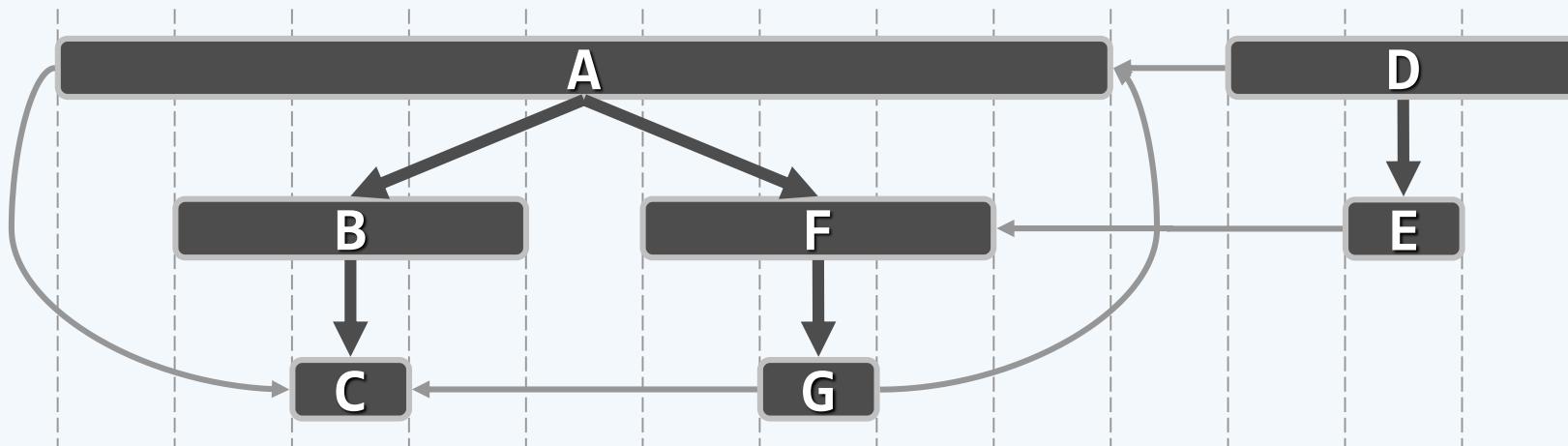
- ❖ DFS树及森林由parent指针描述（只不过所有边取反向）

- ❖ DFS之后，我们已经知道森林乃至原图的全部信息了吗？

就某种意义而言，是的...

## 括号引理

- ❖ 活跃期:  $\text{active}[u] = (\text{dTime}[u], \text{fTime}[u])$
- ❖ Parenthesis Lemma: 给定有向图  $G = (V, E)$  及其任一DFS森林, 则
  - $u$ 是 $v$ 的后代 iff  $\text{active}[u] \subseteq \text{active}[v]$
  - $u$ 是 $v$ 的祖先 iff  $\text{active}[u] \supseteq \text{active}[v]$
  - $u$ 与 $v$ “无关” iff  $\text{active}[u] \cap \text{active}[v] = \emptyset$
- ❖ 仅凭  $\text{status}[]$ 、 $\text{dTime}[]$  和  $\text{fTime}[]$ , 即可对各边分类...



## 边分类

❖ **TREE(v, u)**:   可从当前v进入处于`UNDISCOVERED`状态的u

❖ **BACKWARD(v, u)**:   试图从当前v进入处于`DISCOVERED`状态的u

DFS发现后向边 iff 存在回路

//后向边数 == 回路数?

❖ **FORWARD(v, u)**:  

试图从当前顶点v进入处于`VISITED`状态的u, 且v更早被发现

❖ **CROSS(v, u)**:  

试图从当前顶点v进入处于`VISITED`状态的u, 且u更早被发现

❖ 无向图中, 后向边与前向边不予区分, 跨边没有

//为什么?

## 遍历算法的应用

|                        |         |
|------------------------|---------|
| 连通图的支撑树 (DFS/BFS Tree) | DFS/BFS |
| 非连通图的支撑森林              | DFS/BFS |
| 连通性检测                  | DFS/BFS |
| 无向图环路检测                | DFS/BFS |
| 有向图环路检测                | DFS     |
| 顶点之间可达性检测/路径求解         | DFS/BFS |
| 顶点之间的最短距离              | BFS     |
| 直径                     | BFS     |
| Eulerian tour          | DFS     |
| 拓扑排序                   | DFS     |
| 双连通分量、强连通分量分解          | DFS     |
| ...                    | ...     |

6. 图

拓扑排序

邓俊辉

deng@tsinghua.edu.cn

# 有向无环图

## ❖ DAG

(Directed Acyclic Graph)

## ❖ 应用

类派生和继承关系图中，是否存在循环定义

操作系统中，相互等待的一组线程可否调度，如何调度

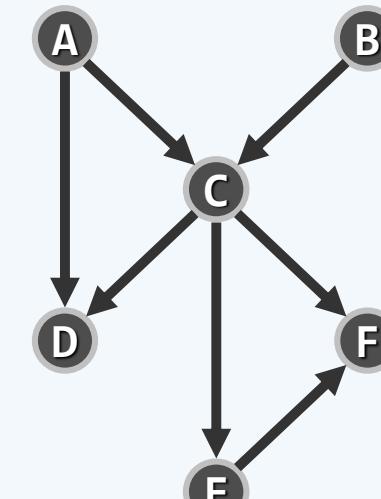
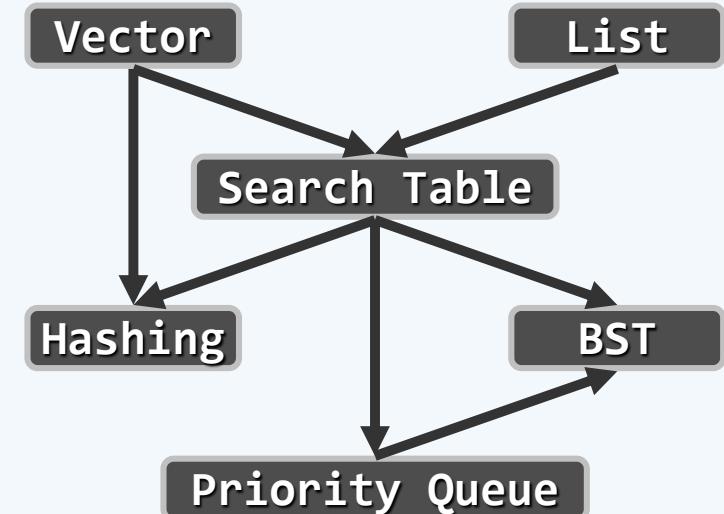
给定一组相互依赖的课程，是否存在可行的培养方案

给定一组相互依赖的知识点，是否存在可行的教学进度方案

项目工程图中，是否存在可串行施工的方案

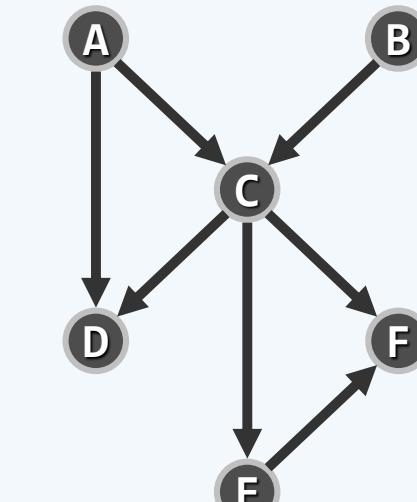
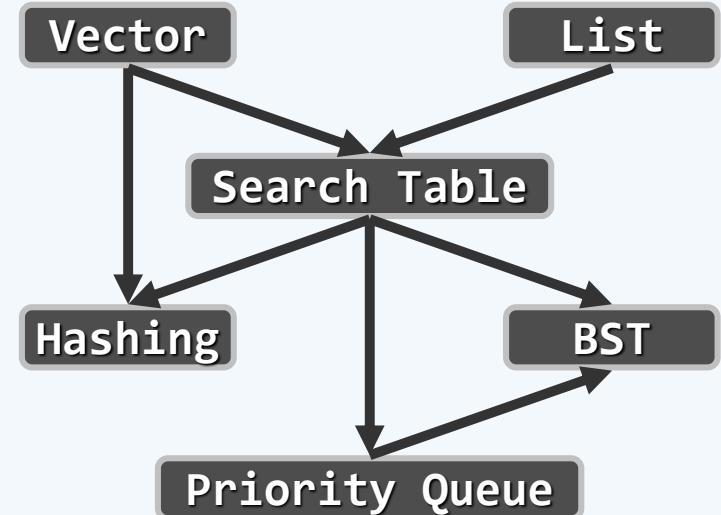
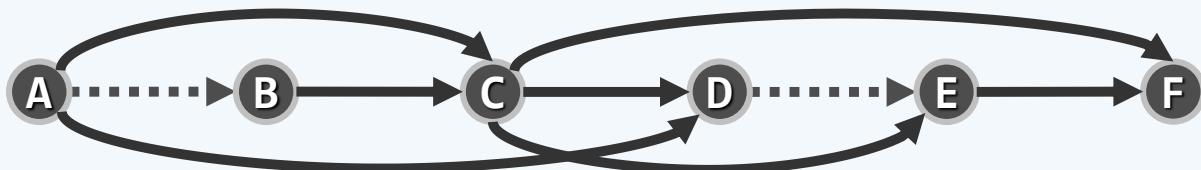
email系统中，是否存在自动转发或回复的回路

...



## 拓扑排序

- ❖ 任给有向图G，不一定是DAG
- ❖ 尝试将其中顶点排成一个线性序列
  - 其次序须与原图相容，亦即
  - 每一顶点都不会通过边指向前驱顶点
- ❖ 算法要求
  - 若原图存在回路（即并非DAG），检查并报告
  - 否则，给出一个相容的线性序列



## 存在性

- ❖ 每个DAG对应于一个偏序集；拓扑排序对应于一个全序集

所谓的拓扑排序，即构造一个与指定偏序集相容的全序集

- ❖ 可以拓扑排序的有向图，必定无环 //反之  
任何DAG，都存在（至少）一种拓扑排序？是的！ //为什么…

- ❖ 有限偏序集必有极大/极大元素

任何DAG都存在（至少）一种拓扑排序

- ❖ 可归纳证明，并直接导出一个算法…

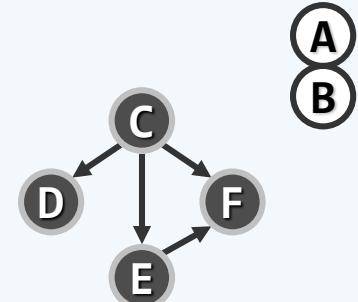
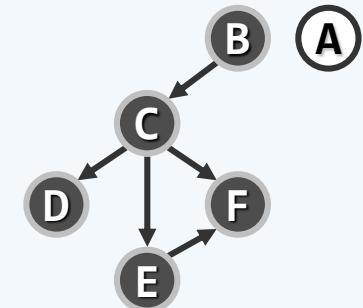
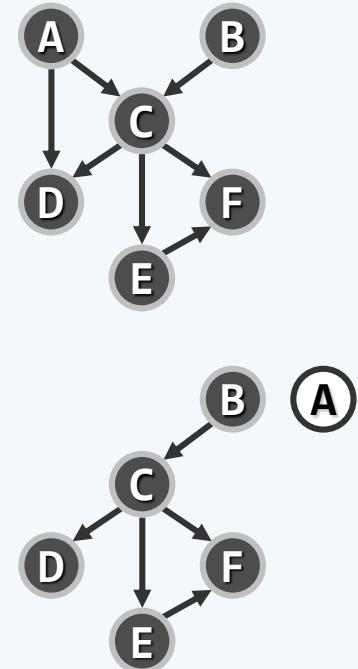
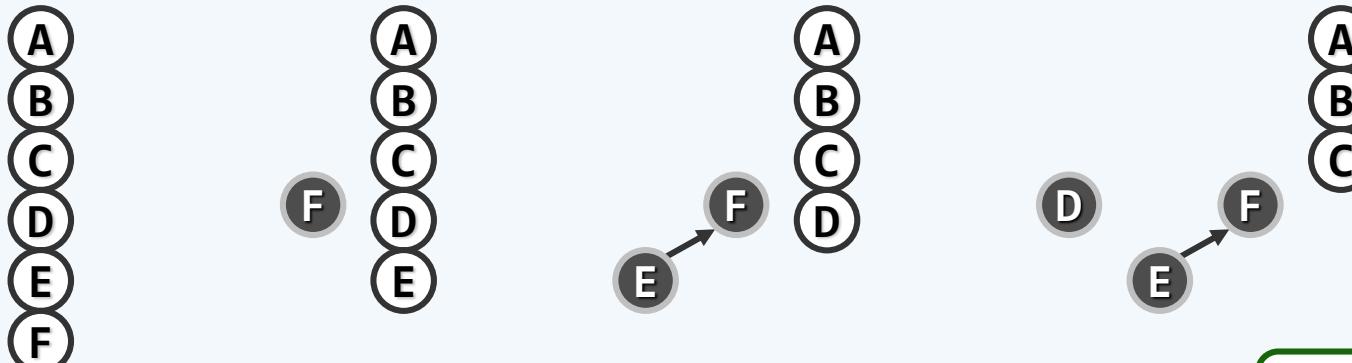
## 存在性

1. 任何DAG，必有（至少一个）顶点入度为零 //记作 $m$
  2. 若  $DAG \setminus \{m\}$  存在拓扑排序  $S = \langle u_{k_1}, \dots, u_{k(n-1)} \rangle$  //subtraction  
则  $S' = \langle m, u_{k_1}, \dots, u_{k(n-1)} \rangle$  即为DAG的拓扑排序 //DAG子图亦为DAG
- ❖ 只要 $m$ 不唯一，拓扑排序也应不唯一 //反之呢？

## 算法A：顺序输出零入度顶点

将所有入度为零的顶点存入栈S，取空队列Q //O(n)

```
while ( ! S.empty() ) { //O(n)  
    Q.enqueue( v = S.pop() ); //栈顶v转入队列  
    for each edge(v, u) //v的邻接顶点u若入度仅为1  
        if ( inDegree(u) < 2 ) S.push(u); //则入栈  
        G = G \ {v}; //删除v及其关联边 (邻接顶点入度减1)  
} //总体O(n + e)  
return |G| ? Q : "NOT_DAG"; //残留的G空，当且仅当原图可拓扑排序
```



## 算法B：逆序输出零出度顶点

❖ /\* 基于DFS，借助栈S \*/

对图G做DFS，其间 //得到组成DFS森林的一系列DFS树

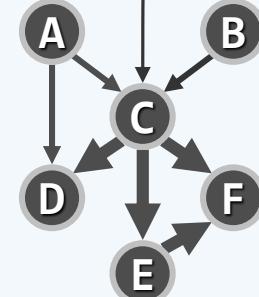
每当有顶点被标记为VISITED，则将其压入S

一旦发现有后向边，则报告非DAG并退出

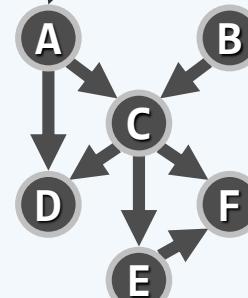
DFS结束后，顺序弹出S中的各个顶点

❖ 复杂度与DFS相当，也是 $\theta(n + e)$

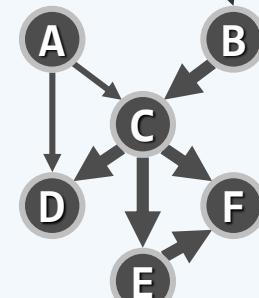
DFS(G, 'C')



DFS(G, 'A')



DFS(G, 'B')



## 算法B：实现

```
❖ template <typename Tv, typename Te> //顶点类型、边类型  
  
bool Graph<Tv, Te>::TSort(int v, int & clock, Stack<Tv>* S) {  
  
    dTime(v) = ++clock; status(v) = [DISCOVERED]; //发现顶点v  
  
    for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v所有邻居u  
  
        /* ... 视u的状态，分别处理 ... */  
  
    status(v) = [VISITED]; S->push( vertex(v) ); //顶点被标记为VISITED时入栈  
  
    return true;  
}
```

## 算法B：实现

```
❖ for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v所有邻居u
    switch ( status(u) ) { //并视u的状态分别处理
        case UNDISCOVERED:
            parent(u) = v; type(v, u) = TREE; //树边(v, u)
            if ( !TSort(u, clock, S) ) return false; break; //从顶点u处深入
        case DISCOVERED: //一旦发现后向边 (非DAG)
            type(v, u) = BACKWARD; return false; //则退出而不再深入
        default: //VISITED (digraphs only)
            type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS; break;
    }
```

# Next

- 最短路径
- 数据结构（C++语言版）第三版 Chapter 6.12