算法设计与分析

LECTURE 7

Lecture 7

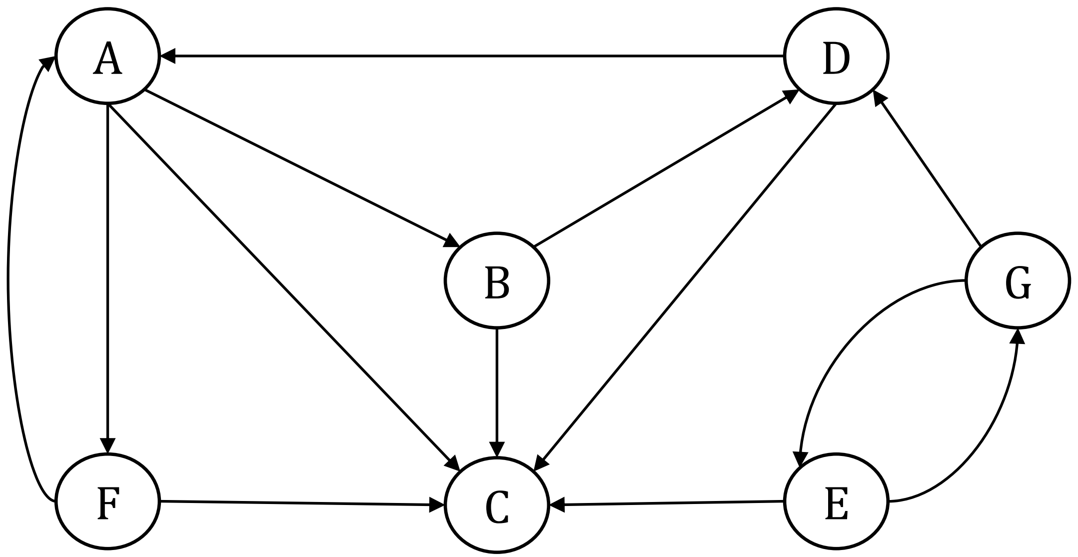
图遍历

Outline

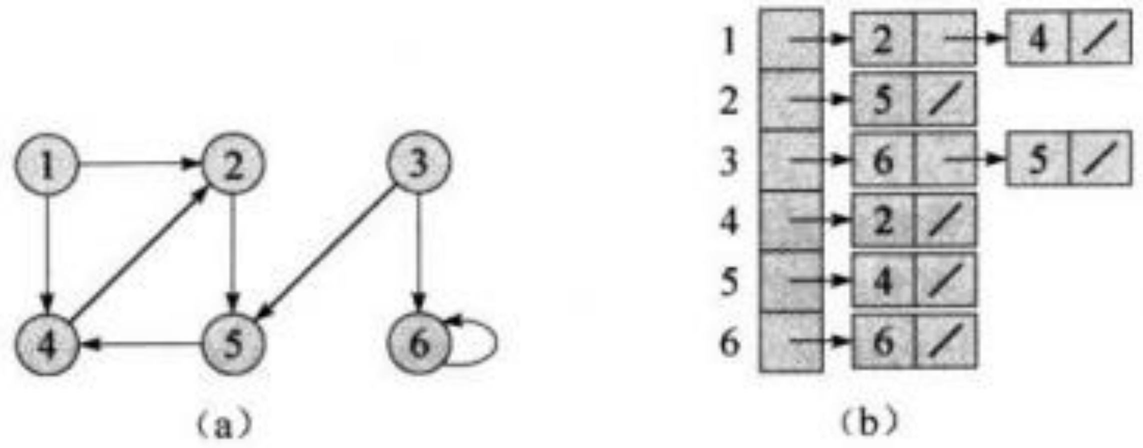
- DFS 和 BFS
- 遍历框架
- 遍历树
- 活动区间

图和图遍历

- 图 G=(V,E)
- 顶点集 V(G)
- 边集 E(G)∈ V × V



图的表示



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

图 22-2　有向图的两种表示。(a)一个有 6 个结点和 8 条边的有向图 G。(b)G 的邻接链表表示。(c)G 的邻接矩阵表示

图的表示

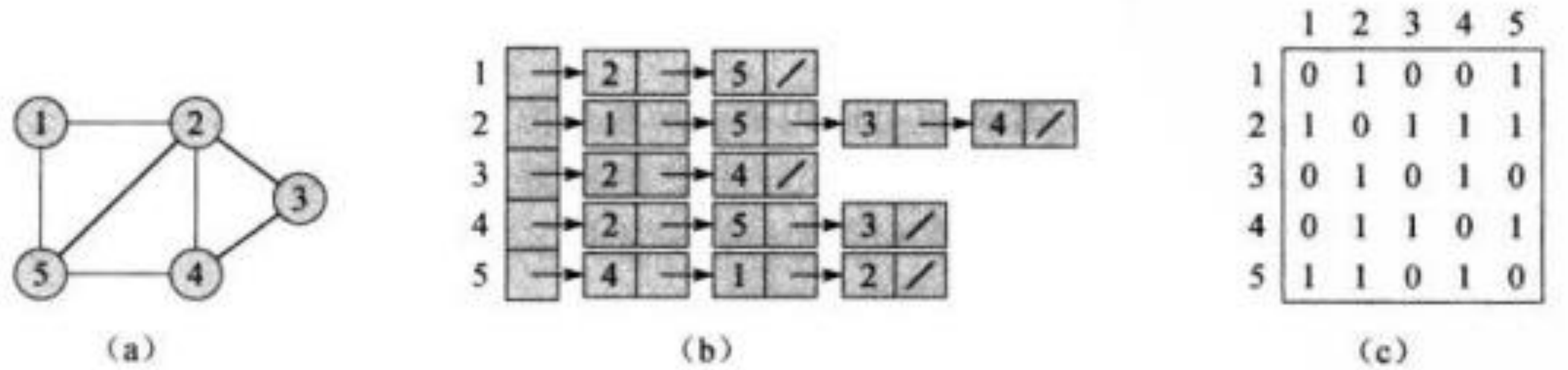
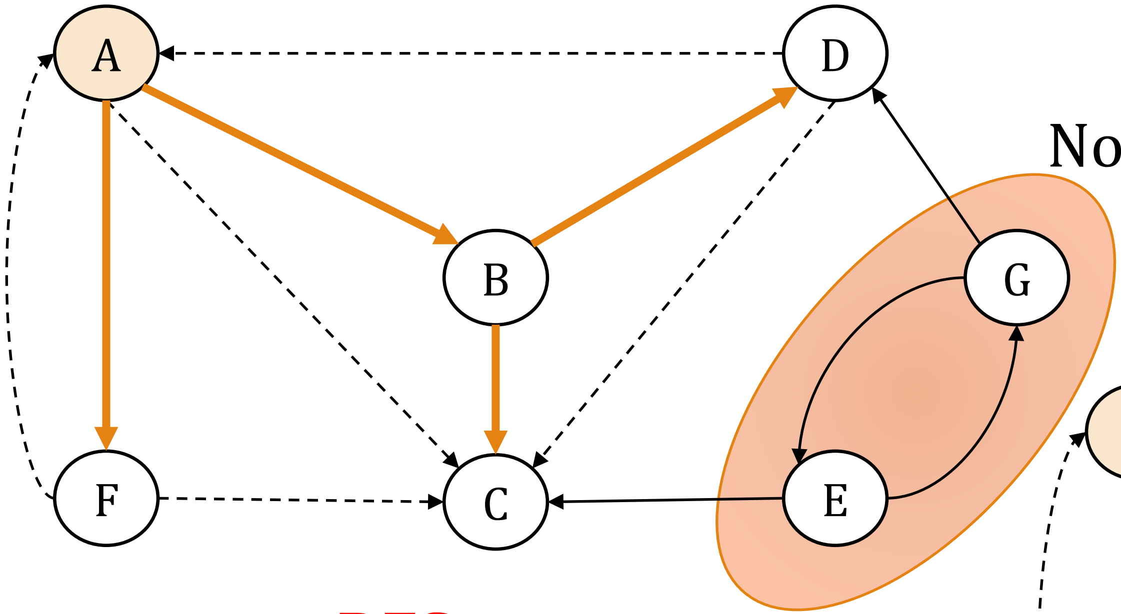


图 22-1　无向图的两种表示。(a)一个有 5 个结点和 7 条边的无向图 G。(b)G 的邻接链表表示。(c)G 的邻接矩阵表示

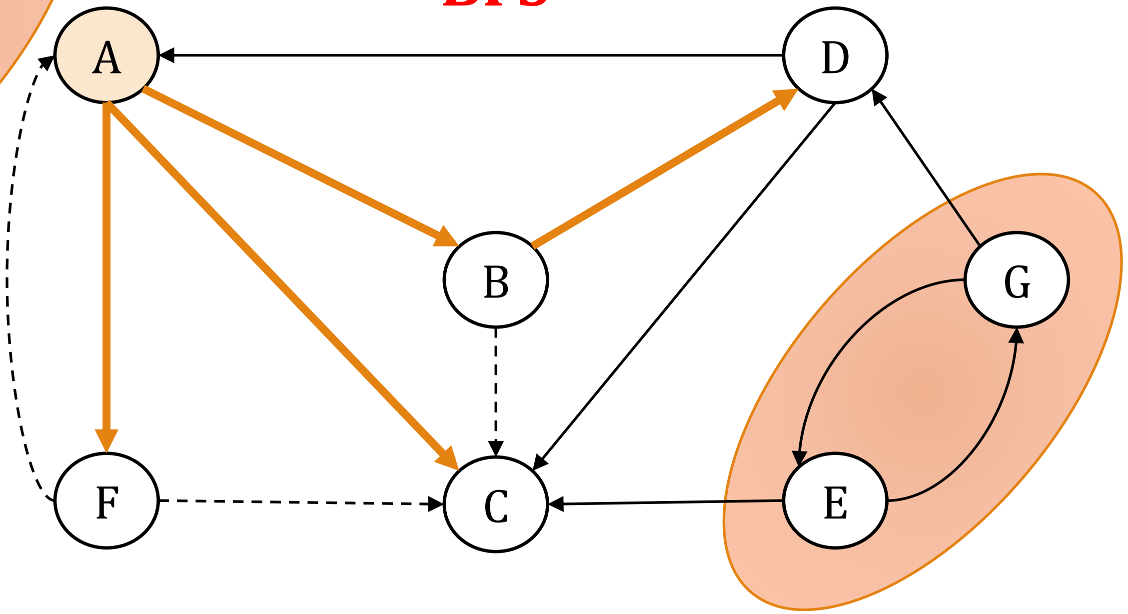
图遍历



Not reachable

DFS

BFS



DFS 框架

- dfs(G,v)
- Mark v as "discovered".
- For each vertex w that edge vw is in G:
- If w is undiscovered:
- dfs(G,w)
- Otherwise:
- "Check" vw without visiting w.
- Mark v as "finished".

BFS 框架

```
• Bfs(G,s)
•     Mark s as "discovered";
•     enqueue(pending,s);

•     while (pending is nonempty)
•         dequeue(pending, v);
•         For each vertex w that edge vw is in G:
•             If w is "undiscovered"
•                 Mark w as "discovered" and
                  enqueue(pending, w)
•         Mark v as "finished";
```

例题

200. 岛屿数量

难度 中等 1405 收藏 分享 切换为英文 接收动态 反馈

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

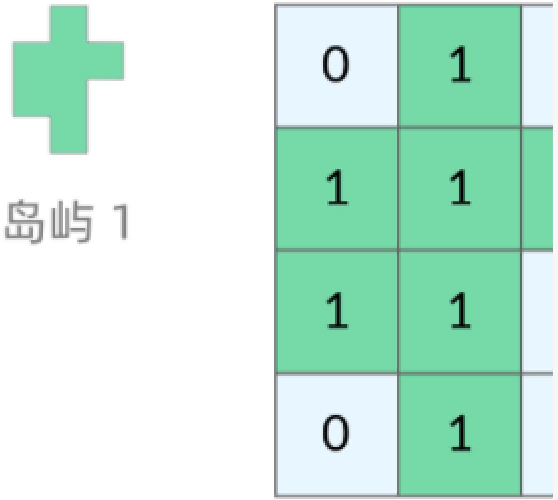
岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1:

```
输入：grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
输出：1
```

岛屿数量




```java
void dfs(int[][] grid, int r, int c) {
    // 判断 base case
    // 如果坐标 (r, c) 超出了网格范围，直接返回
    if (!inArea(grid, r, c)) {
        return;
    }
    // 访问上、下、左、右四个相邻结点
    dfs(grid, r - 1, c);
    dfs(grid, r + 1, c);
    dfs(grid, r, c - 1);
    dfs(grid, r, c + 1);
}

// 判断坐标 (r, c) 是否在网格中
boolean inArea(int[][] grid, int r, int c) {
    return 0 <= r && r < grid.length
            && 0 <= c && c < grid[0].length;
}
```

岛屿数量

在 LeetCode 中，「岛屿问题」是一个系列系列问题，比如：

- L200. 岛屿数量 （Easy）
- 463. 岛屿的周长 （Easy）
- 695. 岛屿的最大面积 （Medium）
- 827. 最大人工岛 （Hard）

我们所熟悉的 DFS（深度优先搜索）问题通常是在树或者图结构上进行的。而我们今天要讨论的 DFS 问题，是在一种「网格」结构中进行的。岛屿问题是这类网格 DFS 问题的典型代表。网格结构遍历起来要比二叉树复杂一些，如果没有掌握一定的方法，DFS 代码容易写得冗长繁杂。

[ref] https://leetcode-cn.com/problems/number-of-islands/solution/dao-yu-lei-wen-ti-de-tong-yong-jie-fa-dfs-bian-li-/

连通片个数

- void connectedComponents(Intlist[ ] *adjVertices*, int *n*,
  int[ ] *cc*) // *This is a wrapper procedure*
- int[ ] color=new int[*n*+1];
- int *v*;
- <Initialize color array to white for
- for (*v*=1; *v*≤*n*; *v*++)
- if (color[*v*]==white)
- ccDFS(*adjVertices*, *color*, *v*, *v*,
- return

- void ccDFS(IntList[ ] *adjVertices*, int[ ] *color*, int *v*, int *ccNum*, int [ ] *cc*)//*v* as the code of current connected component
- int *w*;
- IntList remAdj;
- color[v]=gray;
- cc[v]=ccNum;
- remAdj=adjVertices[v];
- while (remAdj≠nil)
- w=first(remAdj);
- if (color[w]==white)
- ccDFS(adjVertices, color, w, ccNum, cc);
- remAdj=rest(remAdj);
- color[v]=black;
- return

The elements of *remAdj* are neighbors of *v*

Processing the next neighbor, if existing, another depth-first search to be incurred

v finished

遍历框架

- 遍历过程中节点分类
  - 白色：一个节点尚未被遍历到
  - 灰色：一个节点已经被遍历到，但是尚未结束
  - 黑色：一个节点的所有邻居节点已经完成遍历，其自身遍历已经结束

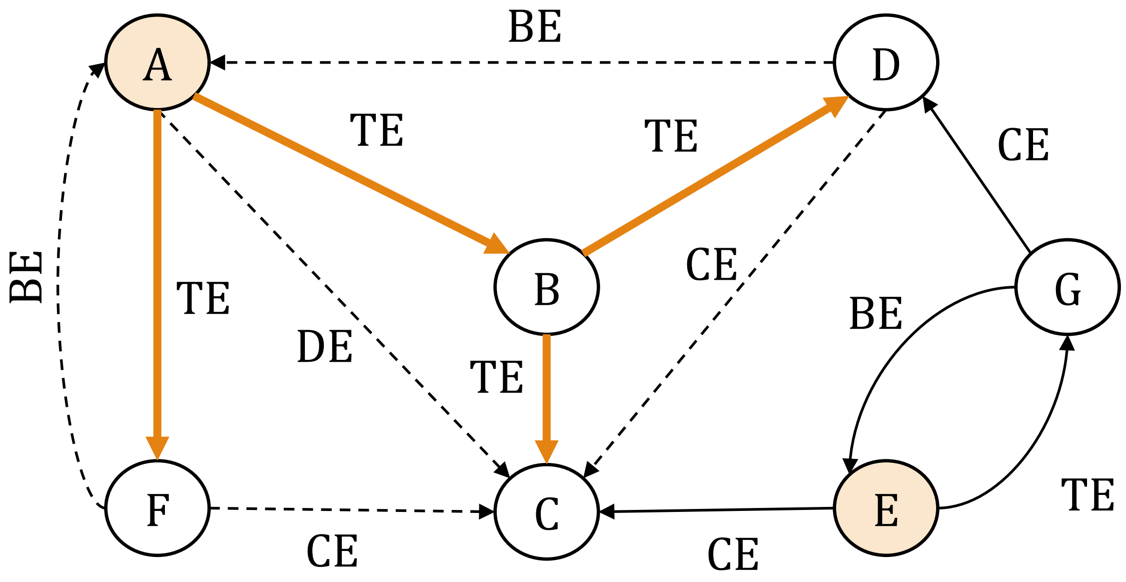
- 遍历过程中处理操作
  - 遍历前处理
  - 遍历中处理
  - 遍历后处理

遍历框架

```
int dfs(IntList[] adjVertices, int[] color, int v, ...)
  int w;
  IntList remAdj;
  int ans;
  color[v]=gray;
  <Preorder processing of vertex v>
  remAdj=adjVertices[v];
  while (remAdj≠nil)
    w=first(remAdj);
    if (color[w]==white)
      <Exploratory processing for tree edge vw>
      int wAns=dfs(adjVertices, color, w, ...);
      < Backtrack processing for tree edge vw , using wAns>
    else
      <Checking for nontree edge vw>
    remAdj=rest(remAdj);
  <Postorder processing of vertex v, including final computation of ans>
  color[v]=black;
```

If partial search is used for a application, tests for termination may be inserted here.

遍历树



TE: Tree Edge
BE: Back Edge
DE: Desendant Edge
CE: Cross Edge

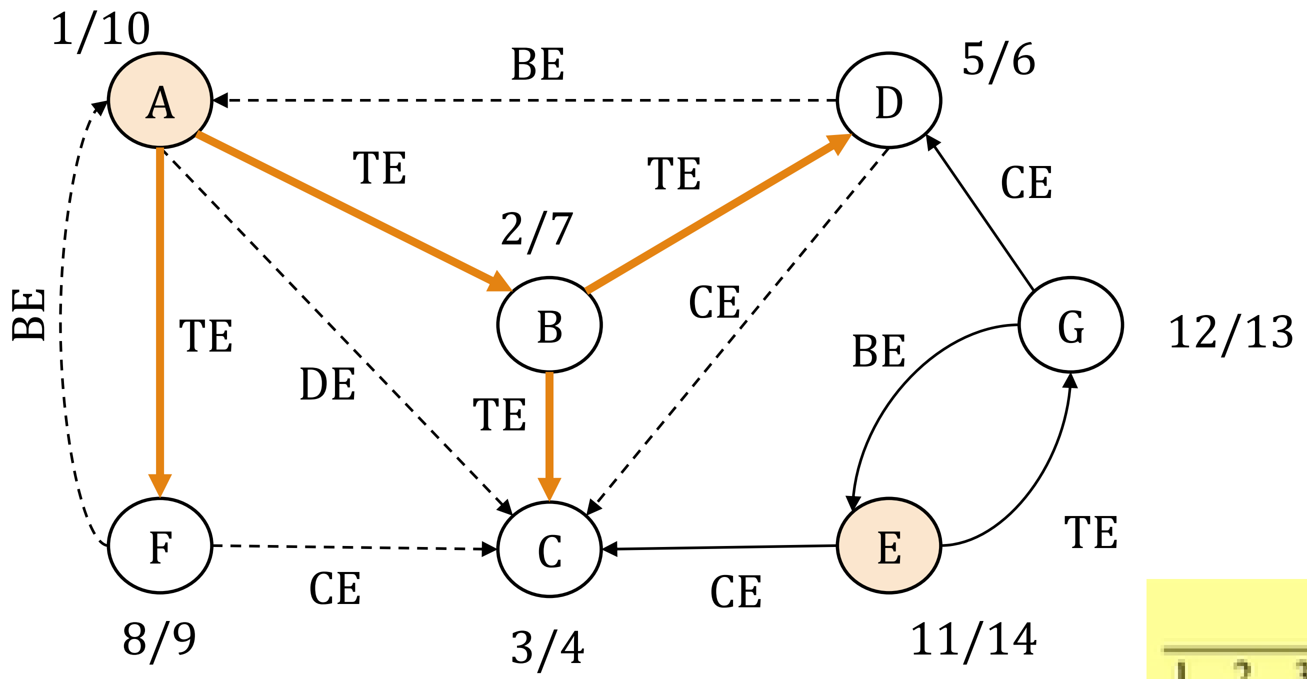
活动区间

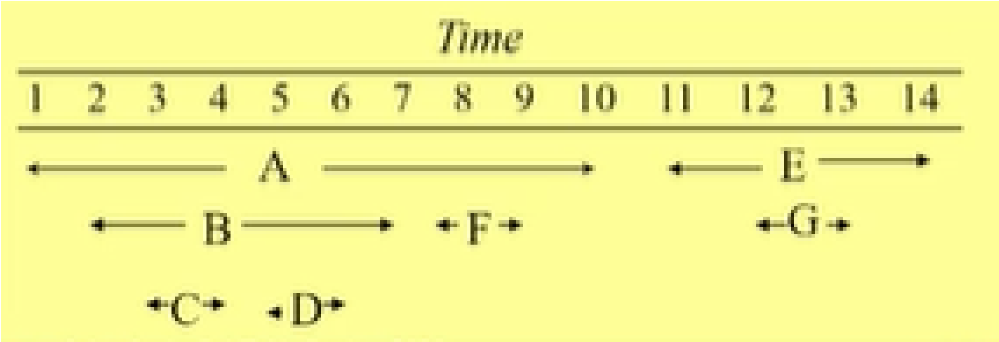
- 遍历时间
  - discoverTime：被发现的时间
  - finishTime：遍历结束的时间

- 一个节点被发现到遍历结束，中间的遍历活动可以由其后继节点的发现与结束事件来刻画。

- 活动区间
  - active(v)=[discoverTime, finishTime]

活动区间


1/10
A
BE
D
5/6
TE
TE
CE
2/7
BE
TE
B
CE
G
12/13
DE
TE
BE
F
C
E
TE
CE
CE
8/9
3/4
11/14


Time
1 2 3 4 5 6 7 8 9 10 11 12 13 14
A
E
B
F
G
C
D

活动区间

```
• int dfsTrace(intList[ ] adjVertices, int[ ] color, int v,  int[ ] discoverTime,
•                  int[ ] finishTime, int[ ] parent  int time)
•    int w; IntList remAdj; int ans;
•    color[v]=gray; time++; discoverTime[v]=time;
•    remAdj=adjVertices[v];
•    while (remAdj≠nil)
•       w=first(remAdj);
•       if (color[w]==white)
•          parent[w]=v;
•          int wAns=dfsTrace(adjVertices, color, w, discoverTime, finishTime,
                             parent, time);
•       else  <Checking for nontree edge vw>
•       remAdj=rest(remAdj);
•    time++; finishTime[v]=time; color[v]=black;
•    return ans;
```

活动区间

- 图遍历活动区间的时序关系等价地转换成整数区间之间的先后/包含关系

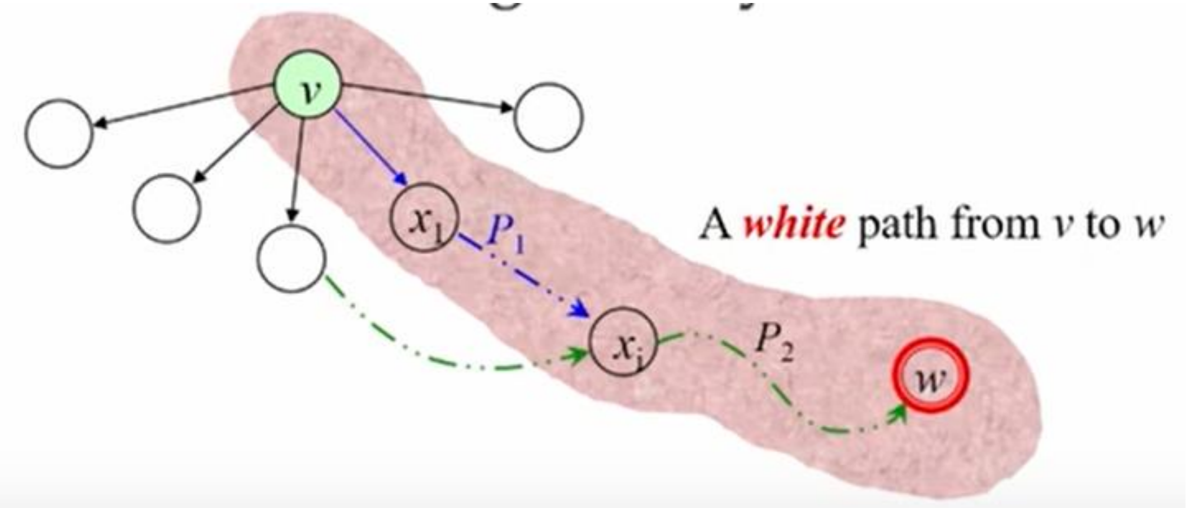
定理 22.7(括号化定理) 在对有向或无向图 $G=(V,E)$ 进行的任意深度优先搜索中，对于任意两个结点 u 和 v 来说，下面三种情况只有一种成立：

- 区间 $[u.d, u.f]$ 和区间 $[v.d, v.f]$ 完全分离，在深度优先森林中，结点 u 不是结点 v 的后代，结点 v 也不是结点 u 的后代。
- 区间 $[u.d, u.f]$ 完全包含在区间 $[v.d, v.f]$ 内，在深度优先树中，结点 u 是结点 v 的后代。
- 区间 $[v.d, v.f]$ 完全包含在区间 $[u.d, u.f]$ 内，在深度优先树中，结点 v 是结点 u 的后代。

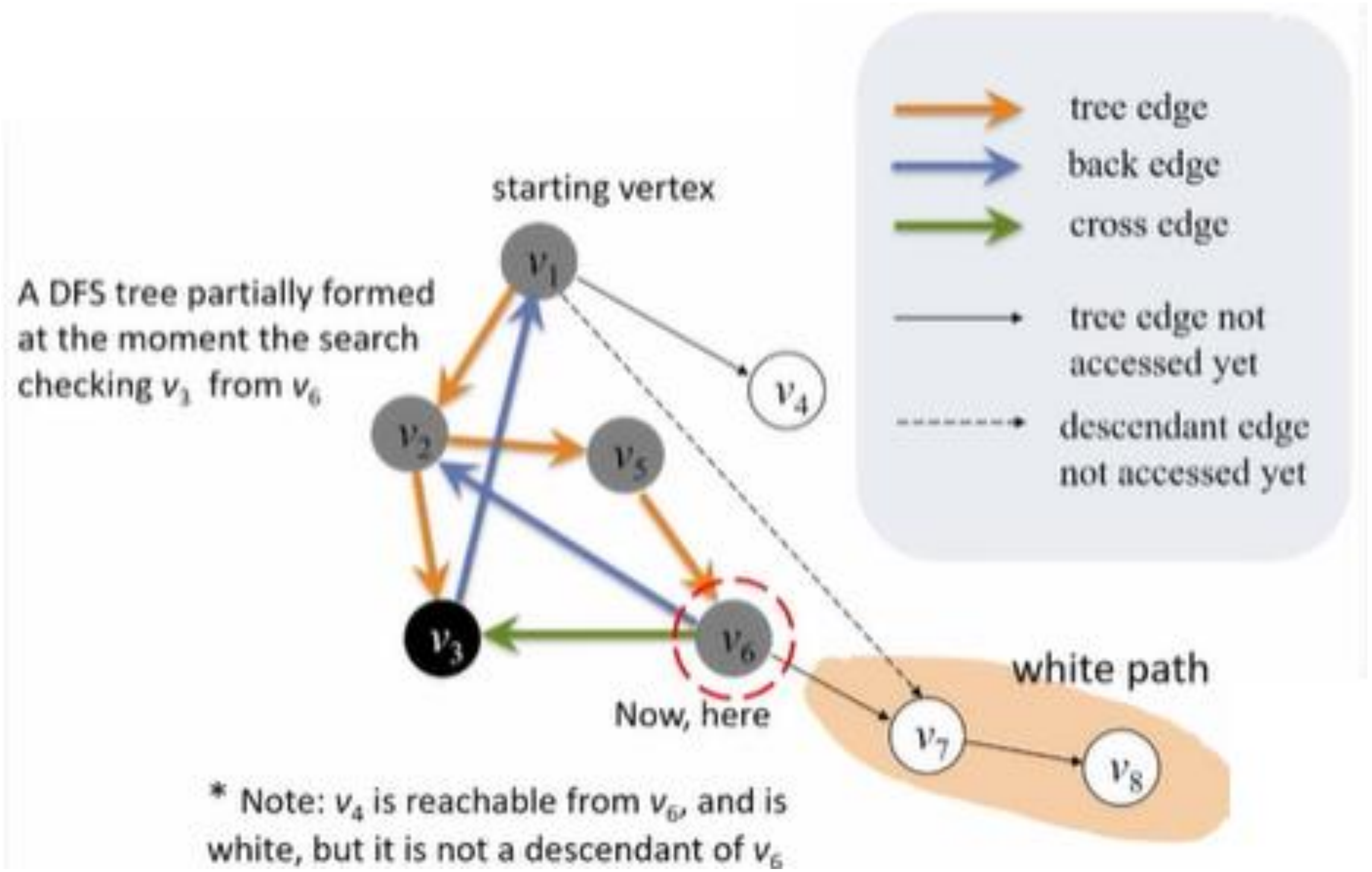
白色路径

- 遍历树中有祖先后继关系 => 在图中有路径相连
- 在图中有路径相连 => 遍历树中有祖先后继关系 ???

- 白色路径定理：在深度优先遍历树中，节点 v 是 w 的祖先，当且仅当在遍历过程中刚刚发现点 v 的时刻，存在一条从 v 到 w 的全部由白色节点组成的路径。



回顾



starting vertex
A DFS tree partially formed at the moment the search checking v_3 from v_6
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
Now, here
white path
* Note: v_4 is reachable from v_6, and is white, but it is not a descendant of v_6
tree edge
back edge
cross edge
tree edge not accessed yet
descendant edge not accessed yet

Lecture 7

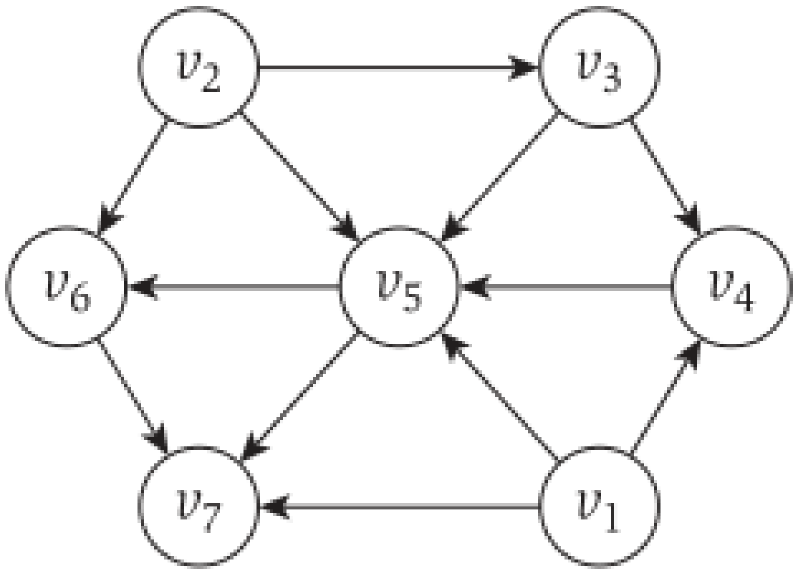
图遍历

Outline

- 有向无环图
  - 拓扑排序
  - 关键路径
- 强连通片

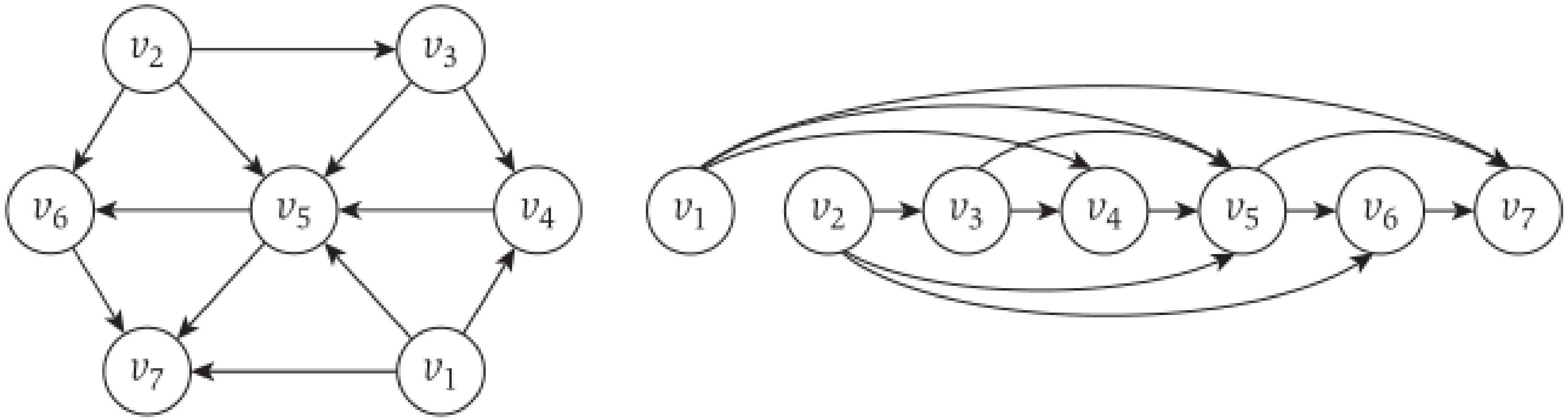
有向无环图 DAG

- Directed Acyclic Graph (DAG)



拓扑排序

- 如果为图中的每个顶点$v_1, v_2, \dots, v_n$分配一个序号$\tau_1, \tau_2, \dots, \tau_n$,满足:
  - 所有序号为正整数1到n的某个排列
  - 对任意有向边 i->j，满足$\tau_i < \tau_j$



拓扑排序的存在性

- 引理：如果有向图 G=(V,E) 中有环，则图 G 不存在拓扑排序。
  - 反证法

- 引理：如果有向图 G=(V,E) 为有向无环图，则 G 必然存在拓扑排序。
  - 构造式证明：存在正确的拓扑排序算法

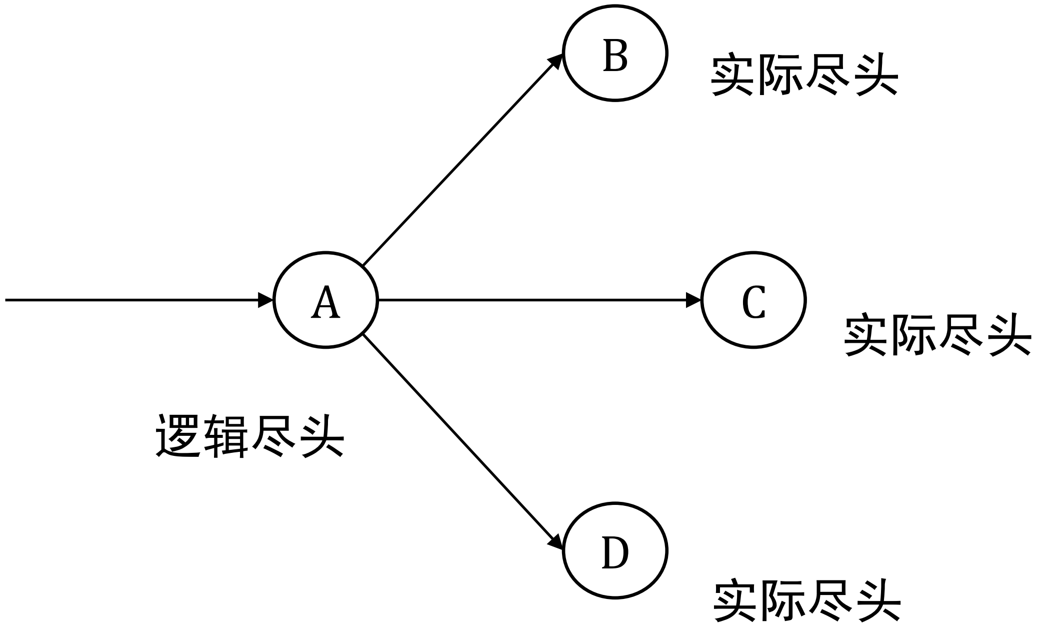
拓扑排序

```
void dfsTopo(IntList[] adjVertices, int[] color, int v, int[ ]
    topo, int topoNum)
    int w; IntList remAdj; color[v]=gray;
    remAdj=adjVertices[v];
    while (remAdj≠nil)
        w=first(remAdj);
        if (color[w]==white)
            dfsTopo(adjVertices, color, w, topo, topoNum);
        remAdj=rest(remAdj);
    topoNum++; topo[v]=topoNum
    color[v]=black;
    return;
```

拓扑排序算法

- 引入概念：“尽头”

- 有向无环
  - 尽头一定存在



关键路径

- 调度问题：
  - 假设有充分多的机器
  - 任务之间有依赖关系
  - 每个任务有自己的执行时长

- 调度目标：所有任务尽早执行完
  - 需要调度的任务 $\{a_i\}(1 \le i \le n)$
  - 每个任务执行时长 l_i
  - 每个任务的最早开始时间 est_i
  - 每个任务的最早结束时间 $eft_i = est_i + l_i$

关键路径

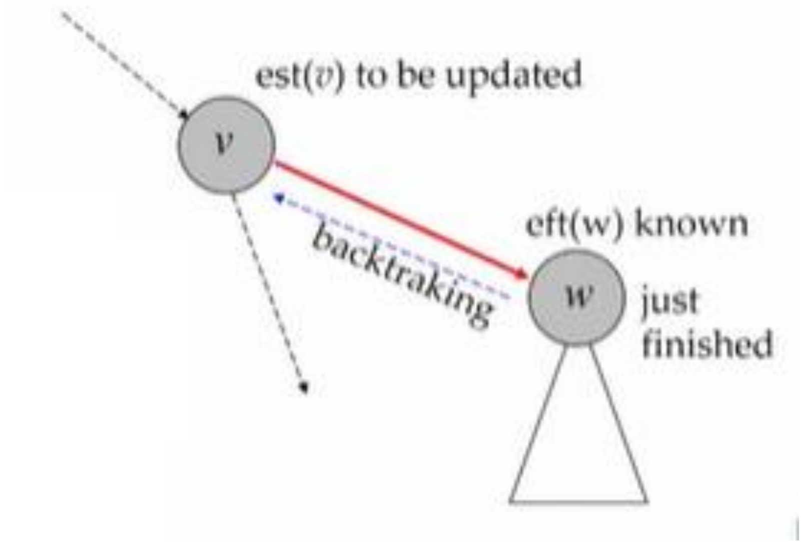
- 最早开始时间、最早结束时间的递归定义：
  - 如果一个任务 a_i不依赖任何其他任务，则 $est_i = 0$
  - 如果一个任务 a_i 的est_i已经被确定，则 $eft_i = est_i + l_i$
  - 如果一个任务 a_i 依赖若干其他任务，则 est_i 为它所依赖的所有任务的最早结束时间中的最大值： $est_i = \max\{eft_j \mid a_i \to a_j\}$

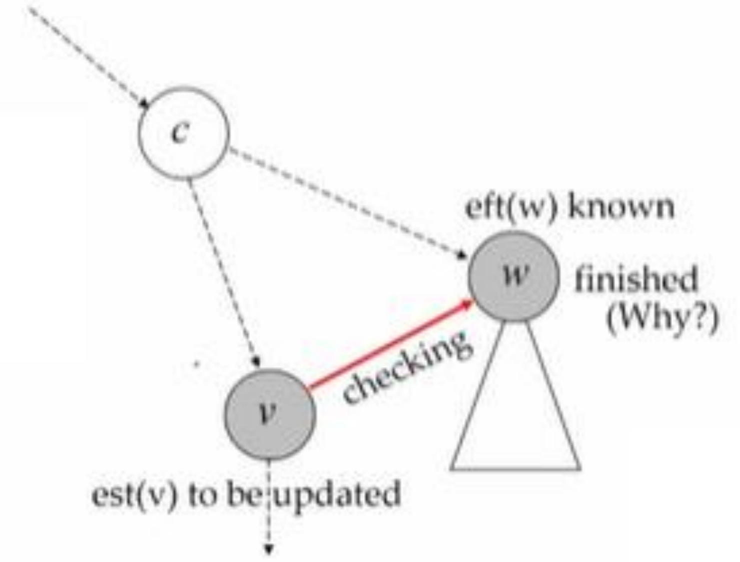
关键路径

- 任务调度中的关键路径是一组任务 $v_0, v_1, \dots, v_k$，满足：
  - 任务 v_0不依赖任何其他任务
  - 对任意 $1 \le i \le k$: $v_i \rightarrow v_{i-1}, est_i = eft_{i-1}$
  - 任务 v_k 的最早结束时间是所有任务的最早结束时间中最大的

关键路径算法

- 遍历前处理：初始化
- 遍历中处理：检查是否需要更新当前节点的 est
  - 处理完一条 TE 返回的时候
  - 处理完一条非TE返回的时候
- 遍历后处理：计算 eft

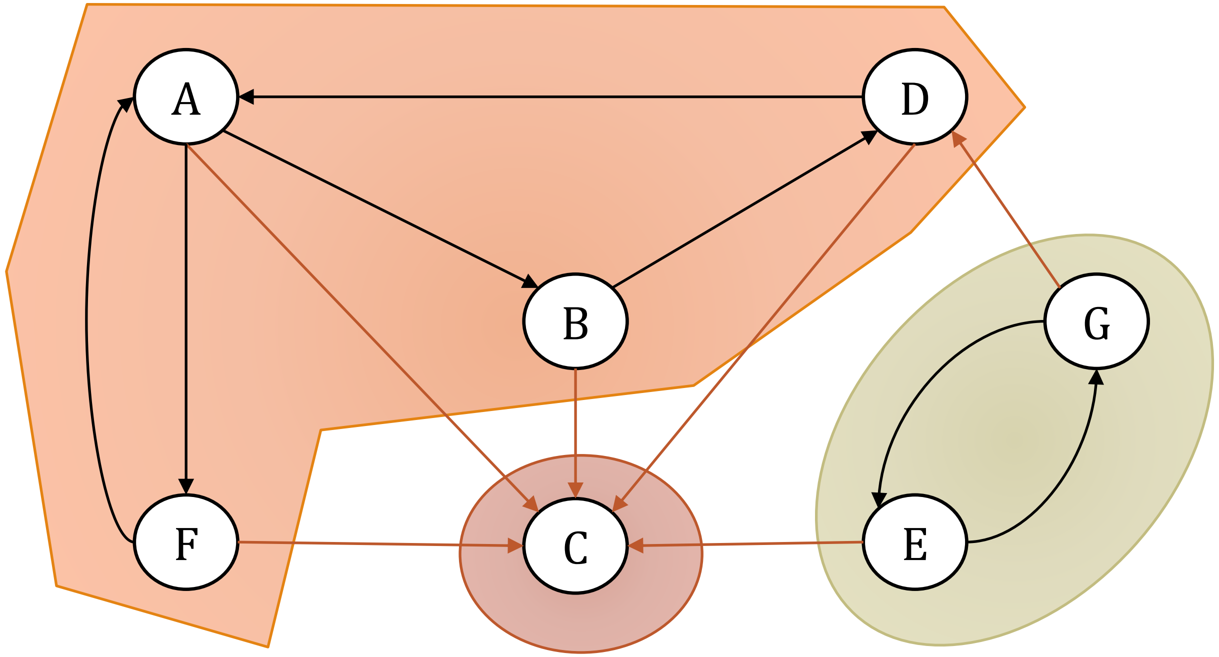


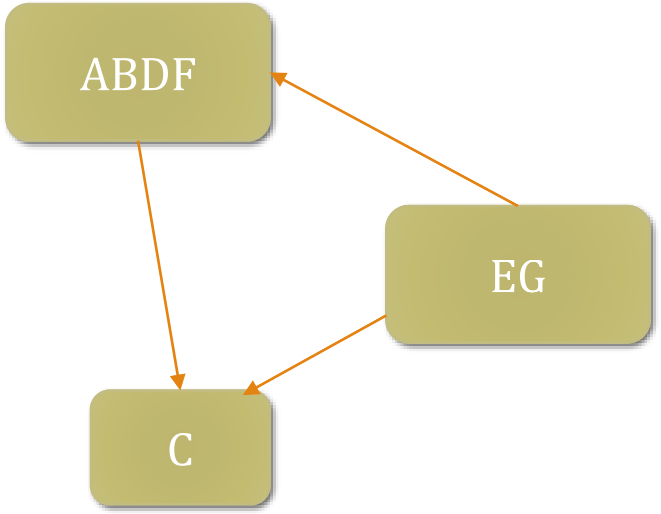


关键路径算法

```
void dfsCrit(.. adjVertices,  .. color, .. v, int[ ] duration, int[ ] critDep,
int[ ] eft)
    int w; IntList remAdj; int est=0;
    color[v]=gray; critDep[v]=-1; remAdj=adjVertices[v];
    while (remAdj≠nil) w=first(remAdj);
        if (color[w]==white)
            dfsCrit(adjVertices, color, w, duration, critDep, efs);
            if (eft[w]≥est) est=eft[w]; critDep[v]=w
        else//checking for nontree edge
            if (eft[w]≥est) est=eft[w]; critDep[v]=w
        remAdj=rest(remAdj);
    eft[v]=est+duration[v]; color[v]=black;
    return;
```

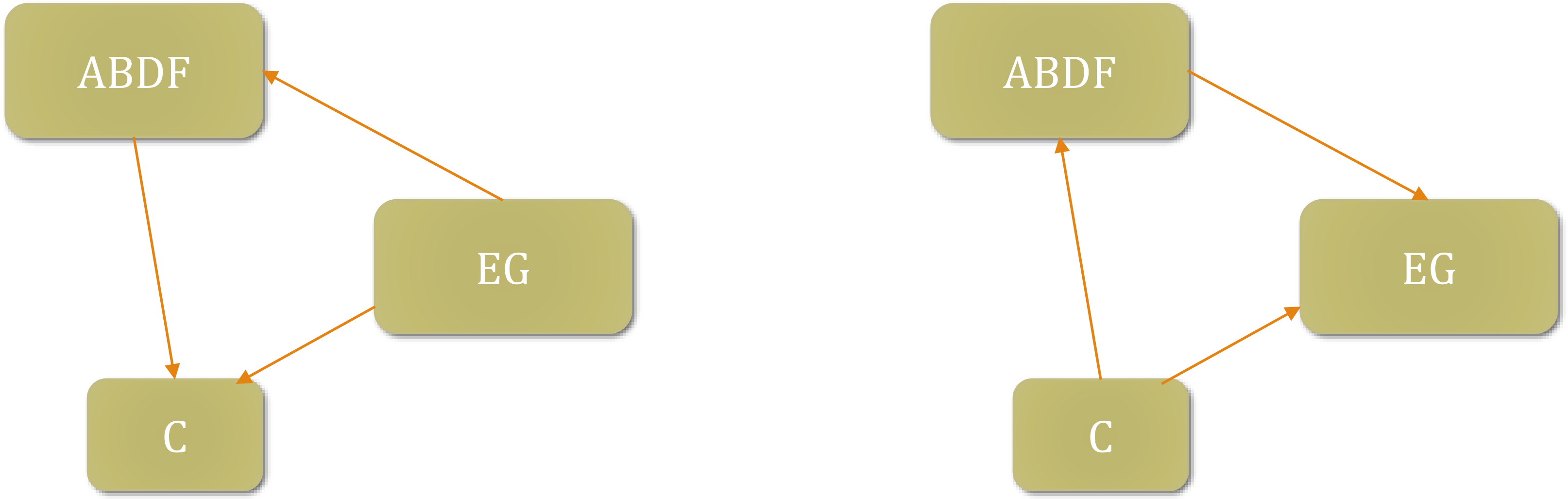
强连通片





收缩图

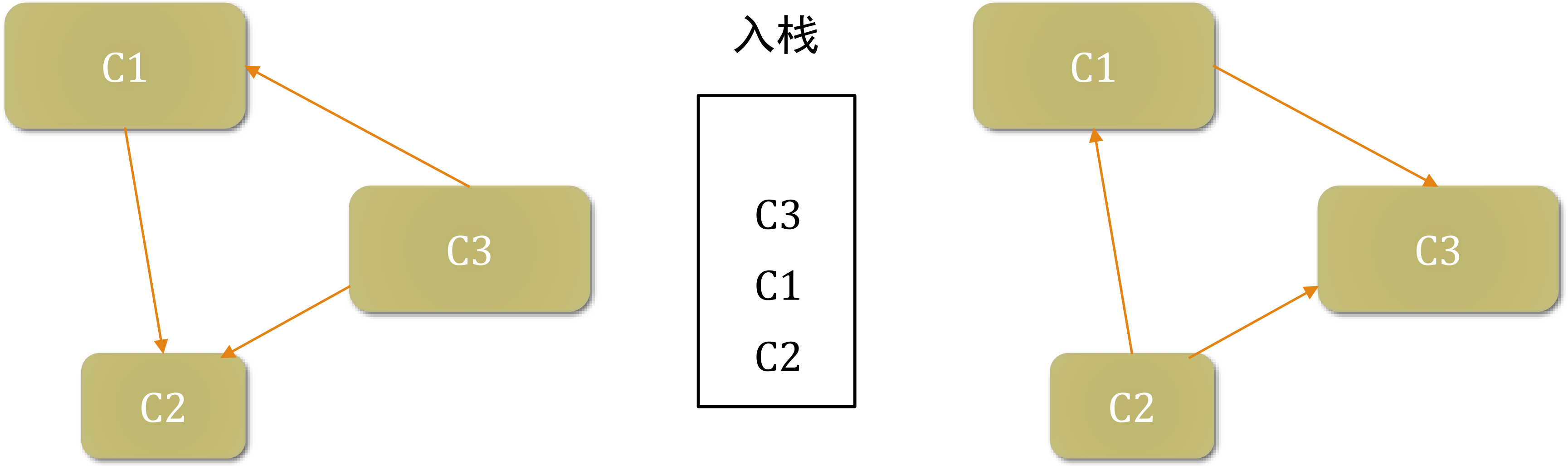
有向图的转置



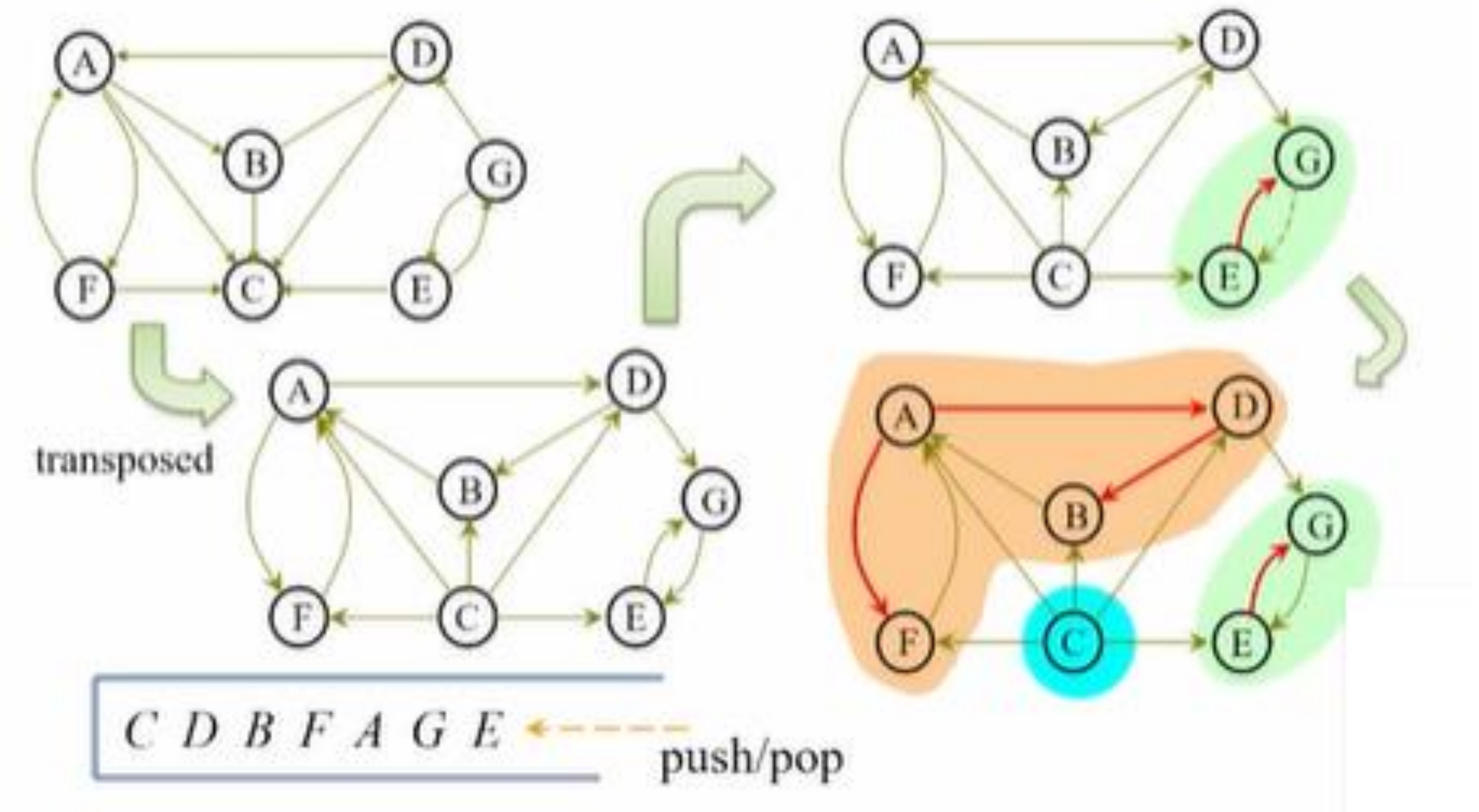
图G和G^T的收缩图

强连通片算法

- 遍历结束时标记尽头
- 先进后出的栈结构：每个节点完成处理时进栈（先结束，先入栈）
- 图的转置：按出栈的顺序开始第二轮遍历



强连通片算法



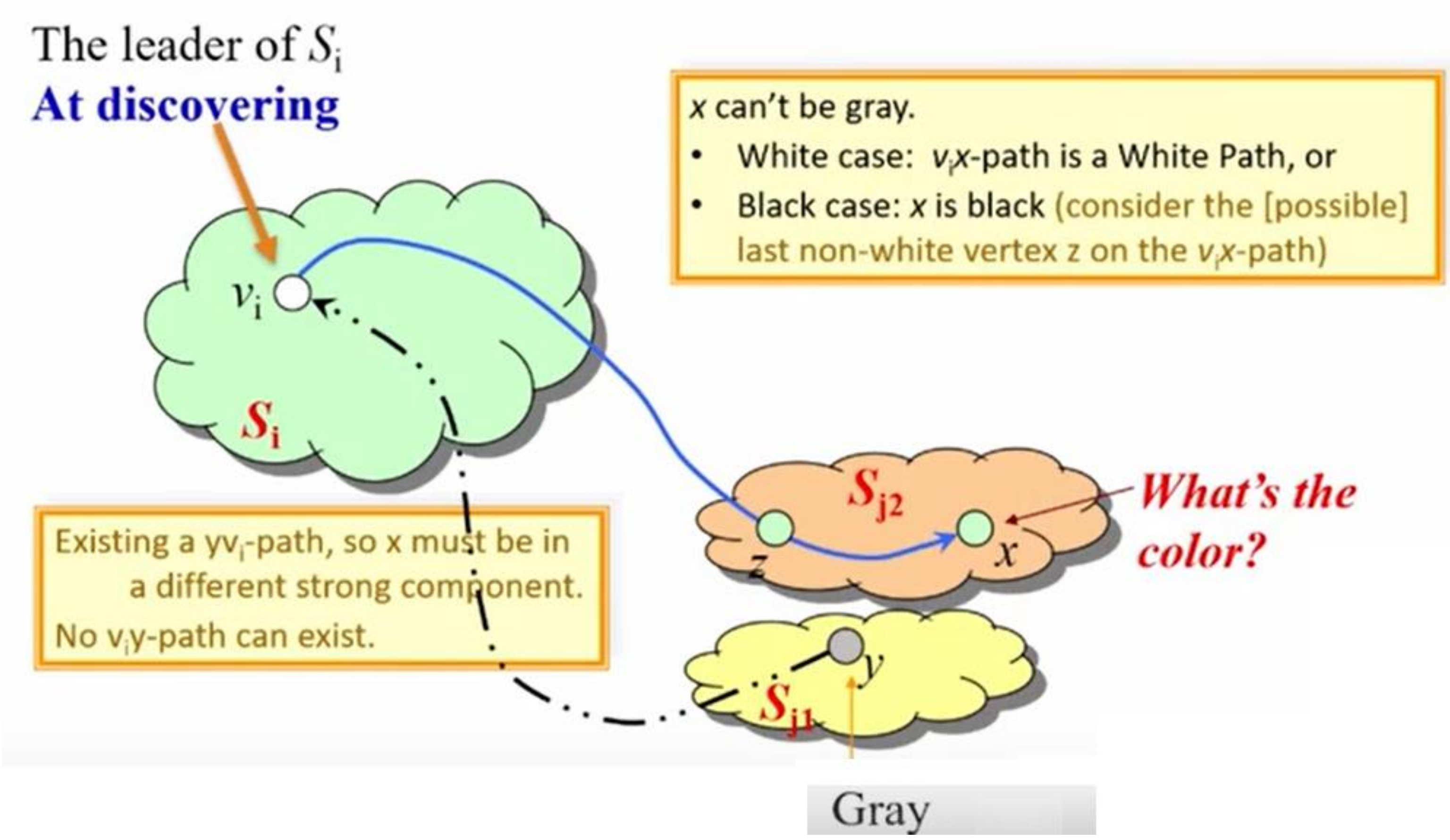
算法正确性

- 强连通片的首节点：在第一轮深度优先遍历过程中，定义每个强连通片中第一个被发现的节点为该强连通片的首节点。

- 推论：首节点的活动区间包含同一个强连通片中其他所有节点的活动区间。

- 引理：第一轮遍历的遍历树中，可能包括一个或多个强连通片的节点。也就是说，一个强连通片中的节点不可能一部分在某棵遍历树中，一部分不在。

强连通片的首节点



The leader of S_i
At discovering
x can't be gray.
• White case: v_ix-path is a White Path, or
• Black case: x is black (consider the [possible] last non-white vertex z on the v_ix-path)
v_i
S_i
S_{j2}
z
x
What's the color?
Existing a yv_i-path, so x must be in a different strong component.
No v_iy-path can exist.
y
S_{j1}
Gray

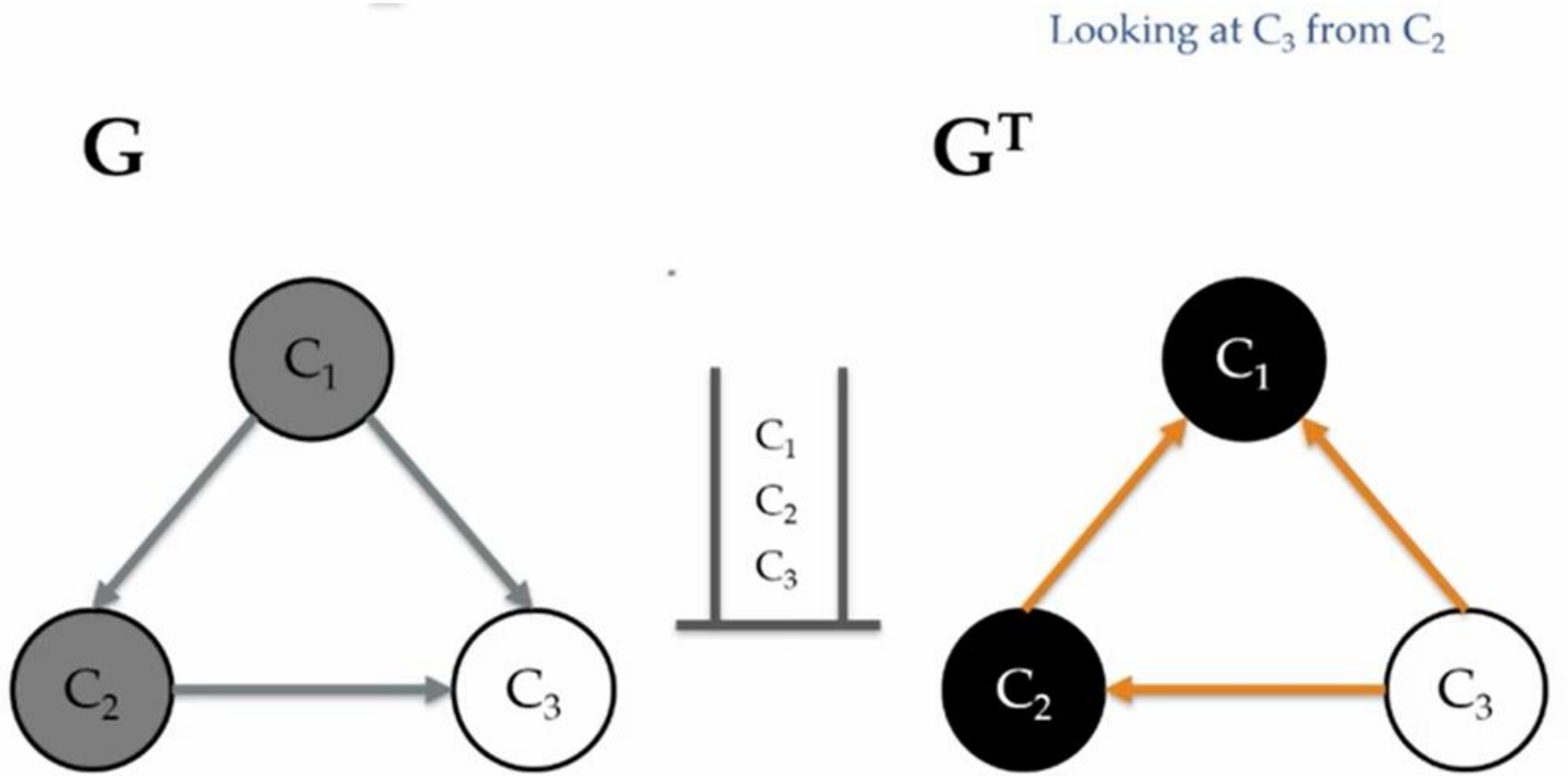
强连通片

- 引理8.4：当某个强连通片的首节点在第一轮遍历中被发现时（刚刚被处理，即将被染为灰色的时候），不可能有路径通向某个灰色节点。

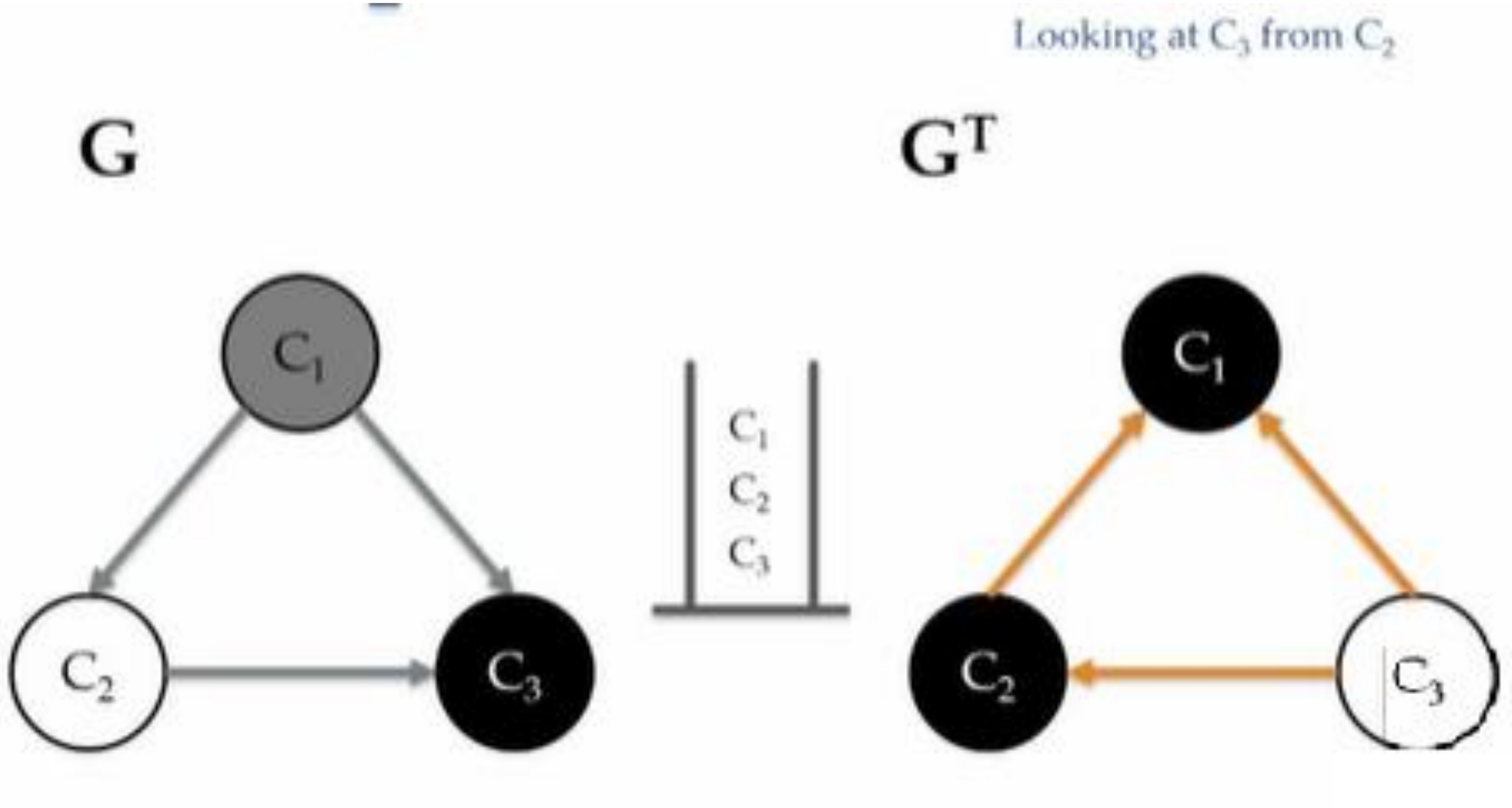
- 引理8.5：假设l是某个强连通片的首节点，x是另一个强连通片的节点，并且存在l通向x的路径，则在第一轮遍历中x比l先结束遍历。

- 引理8.6：在第二轮深度优先遍历过程中，当一个白色节点从栈中被POP出来时，它一定是其所在强连通片的首节点。

白色情况



黑色情况



Lecture 7

图遍历

Outline

- 无向图
  - DFS 框架
  - 割点
  - 桥
- 无向图问题

无向图特点

- 无向图遍历的特点
  - 计算机用一对有向边来表示无向边
  - 一条边可能被遍历两次
- 对于一个无向图，DFS 为每条边做了定向，遍历推进的方向
  - 有向遍历

Edges in DFS

- CE
  - 不存在
- BE
  - 回到直接父亲：第二次遍历
  - 否则：第一次遍历
- DE
  - 总是二次遍历

DFS 框架

```
int dfs(IntList[] adjVertices, int[] color, int v, int p, ...)
    int w; IntList remAdj; int ans;
    color[v]=gray;
    <Preorder processing of vertex v>
    remAdj=adjVertices[v];
    while (remAdj≠nil)
        w=first(remAdj);
        if (color[w]==white)
            <Exploratory processing for tree edge vw>
            int wAns=dfs(adjVertices, color, w, v ...);
            < Backtrack processing for tree edge vw , using wAns>
        else if (color[w]==gray && w≠p)
            <Checking for nontree edge vw>
        remAdj=rest(remAdj);
    <Postorder processing of vertex v, including final computation of ans>
    color[v]=black;
    return ans;
```

无向图

- 连通
  - 冗余连通 / 容错连通
  - k 连通：k-点连通、k-边连通

- 对于连通的无向图G，如果其中任意去掉 k-1 个点，图 G 仍然连通，则称图 G 是 k-点连通。类似地，如果图中任意去掉 k-1条边，图 G 仍然连通，则称图 G 是 k-边连通的。

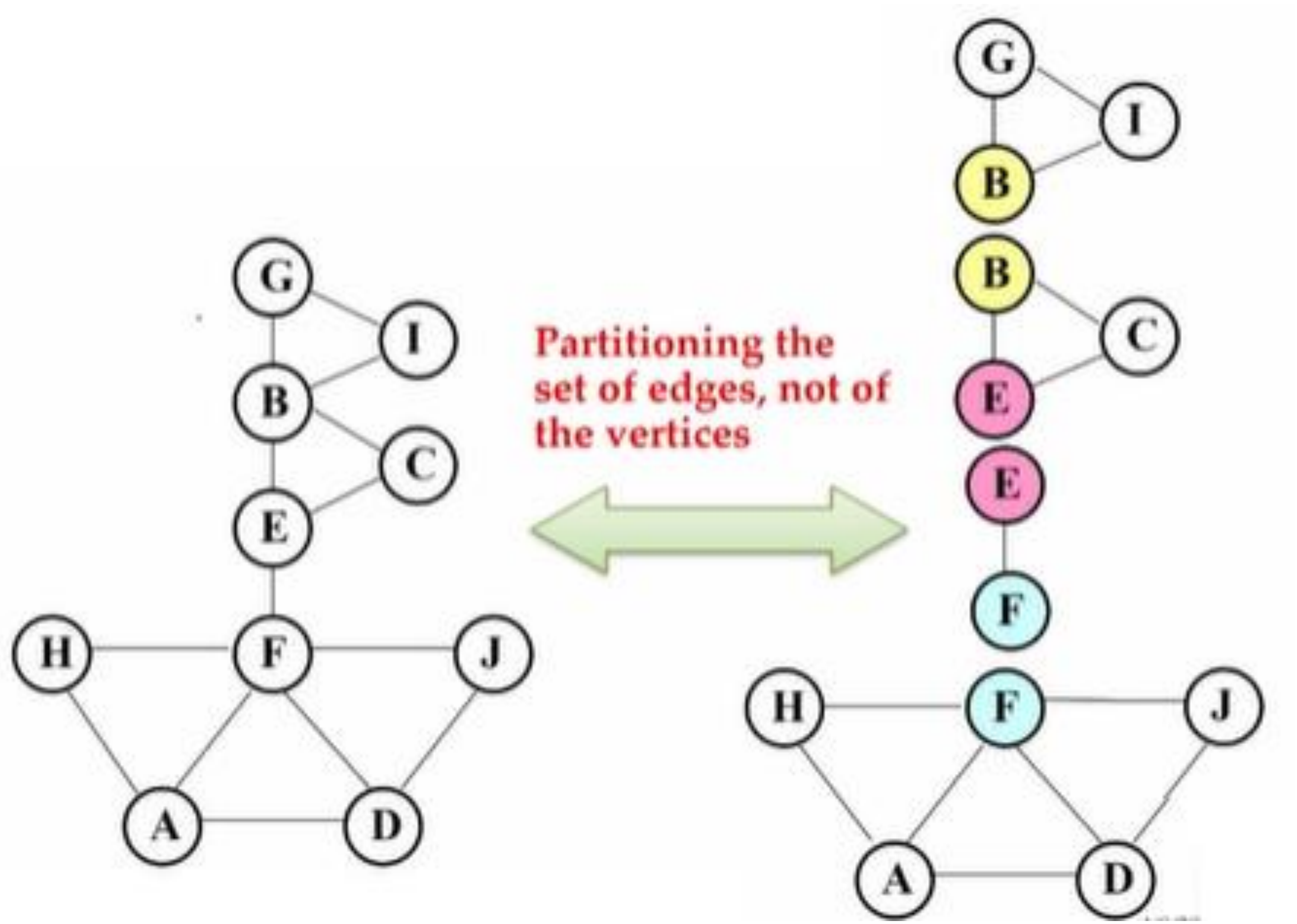
无向图

- 对于一个连通的无向图 G，称节点 v 为割点，如果去掉点 v 之后，图 G 不再连通；称边 uv 为桥，如果去掉边 uv 之后，图 G 不再连通。

- 割点（2-node connected）
  - 删除该节点导致不连通

- 桥（2-edge connected）
  - 删除该边导致不连通

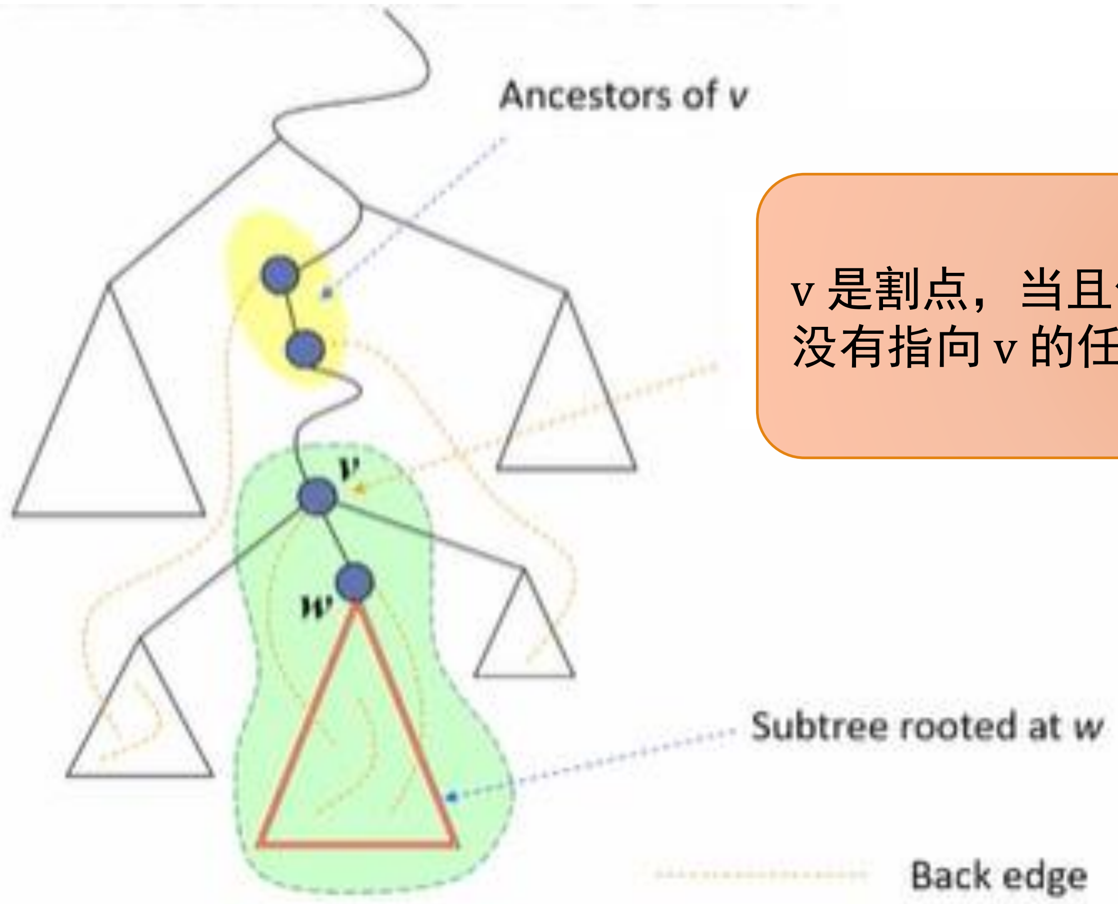
割点


G
I
B
C
E
H
F
J
A
D
Partitioning the
set of edges, not of
the vertices
G
I
B
B
C
E
E
F
H
F
J
A
D

割点定义

- 定义
  - 删除 v 导致不连通
- 基于路径的定义
  - 存在节点 w 和 x，v 在w和x的每条路径上
- 基于 DFS 定义
  - 存在一颗子树（w-rooted），没有 back edge 指向v的祖先

割点算法



v 是割点，当且仅当存在 w-rooted 子树，没有指向 v 的任意祖先的 back edge

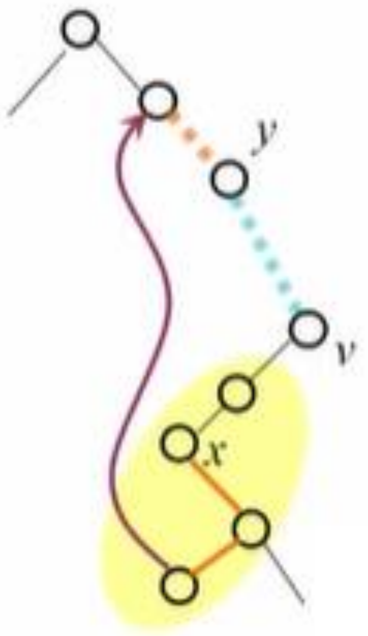
正确性

- 在 DFS 树中，一个节点（非根节点）为割点，当且仅当：
  - 1）v不是叶子节点
  - 2）存在v 的某个子树没有 BE 指向v的祖先节点

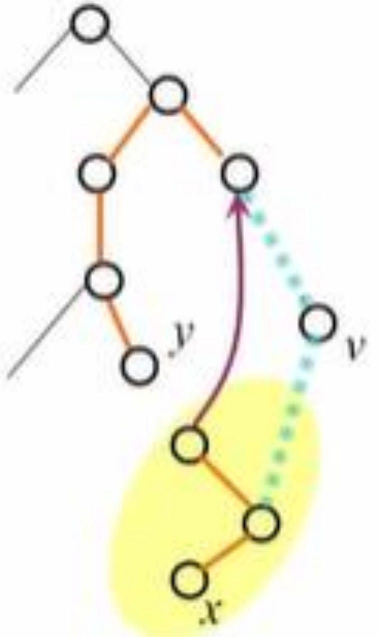
- 证明：
  - 存在x 和 y 满足 v 在从 x 到 y 的每一条路径上
  - 至少有一个节点是 v 的后继节点
  - 通过反证法，假设任意子树均有BE指向v的祖先节点，均可构造一条从x到y且不经过点 v的路径（2 cases）

Case 1

Case 1.2: another is an ancestor of v



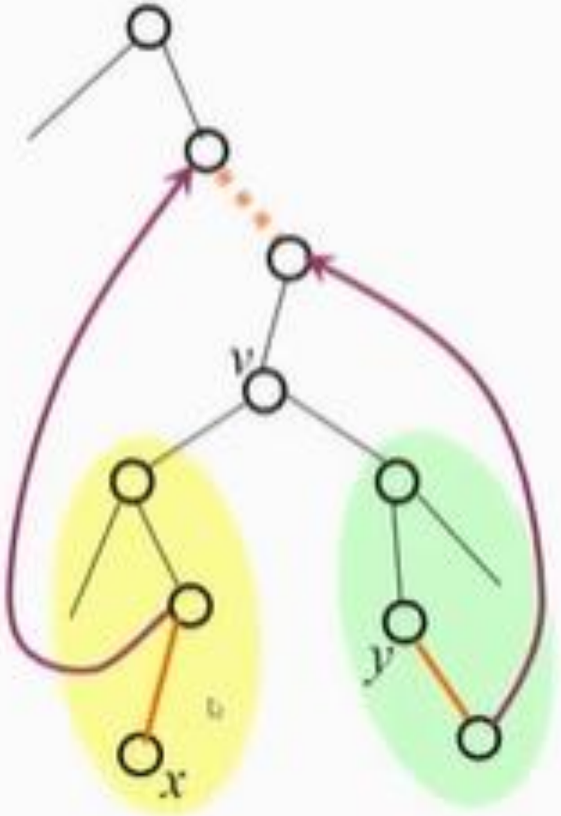
suppose that **every** subtree of *v* has a back edge to a proper ancestor of *v*, and, exactly one of *x*, *y* is a descendant of *v*.



Case 1.1: another is not an ancestor of *v*

Case 2

suppose that **every** subtree of v has a back edge to a proper ancestor of v, and, both x, y are descendants of v.



更新 back 值

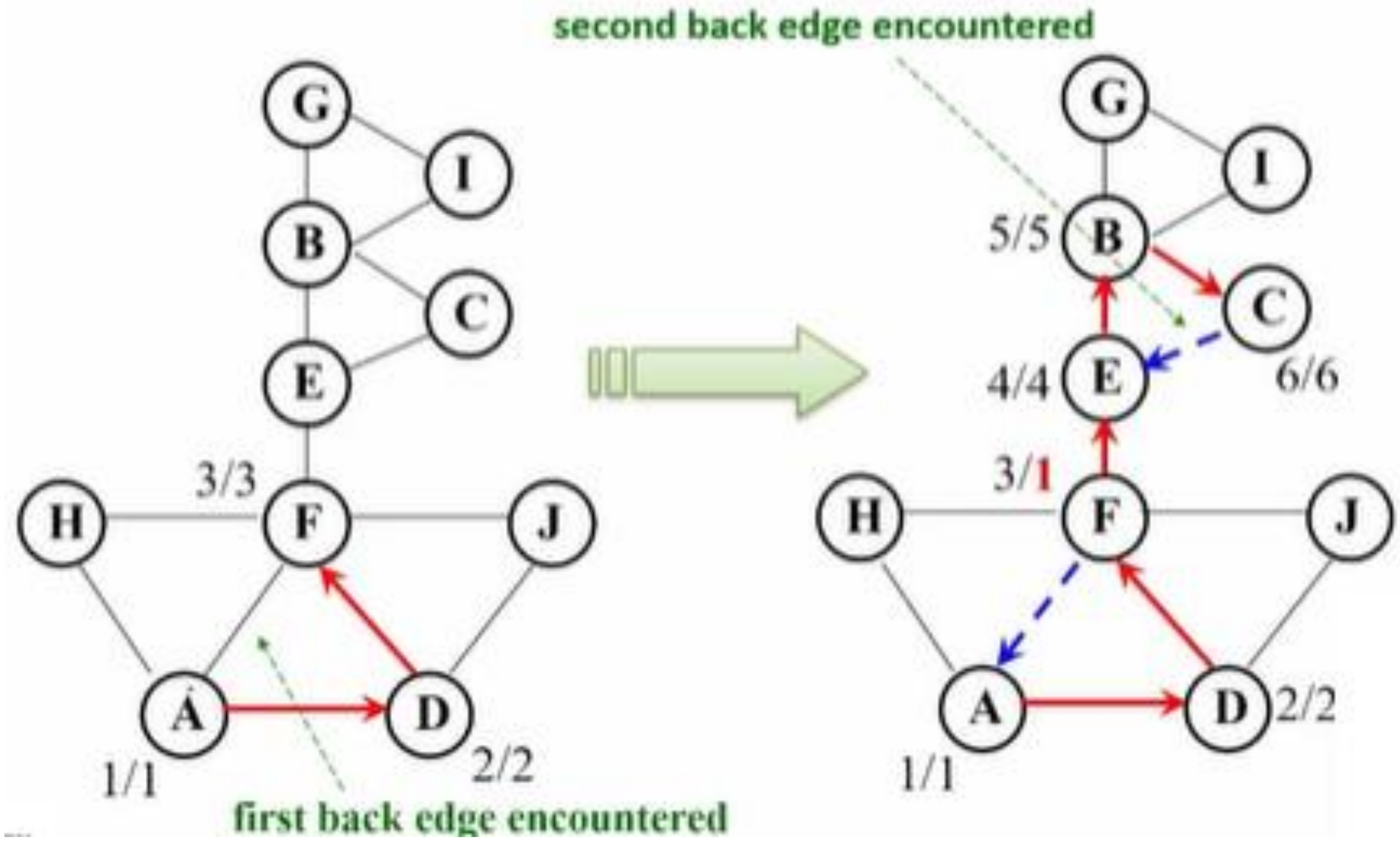
- v 第一次发现
  - back = discoverTime(v)
- 发现 back edge vw
  - back = min(back, discoverTime(w))
- backtracking from w to v
  - back = min(back, wback)

维护back值

- 为每一个顶点维护一个变量 back 来判断它是否是割点

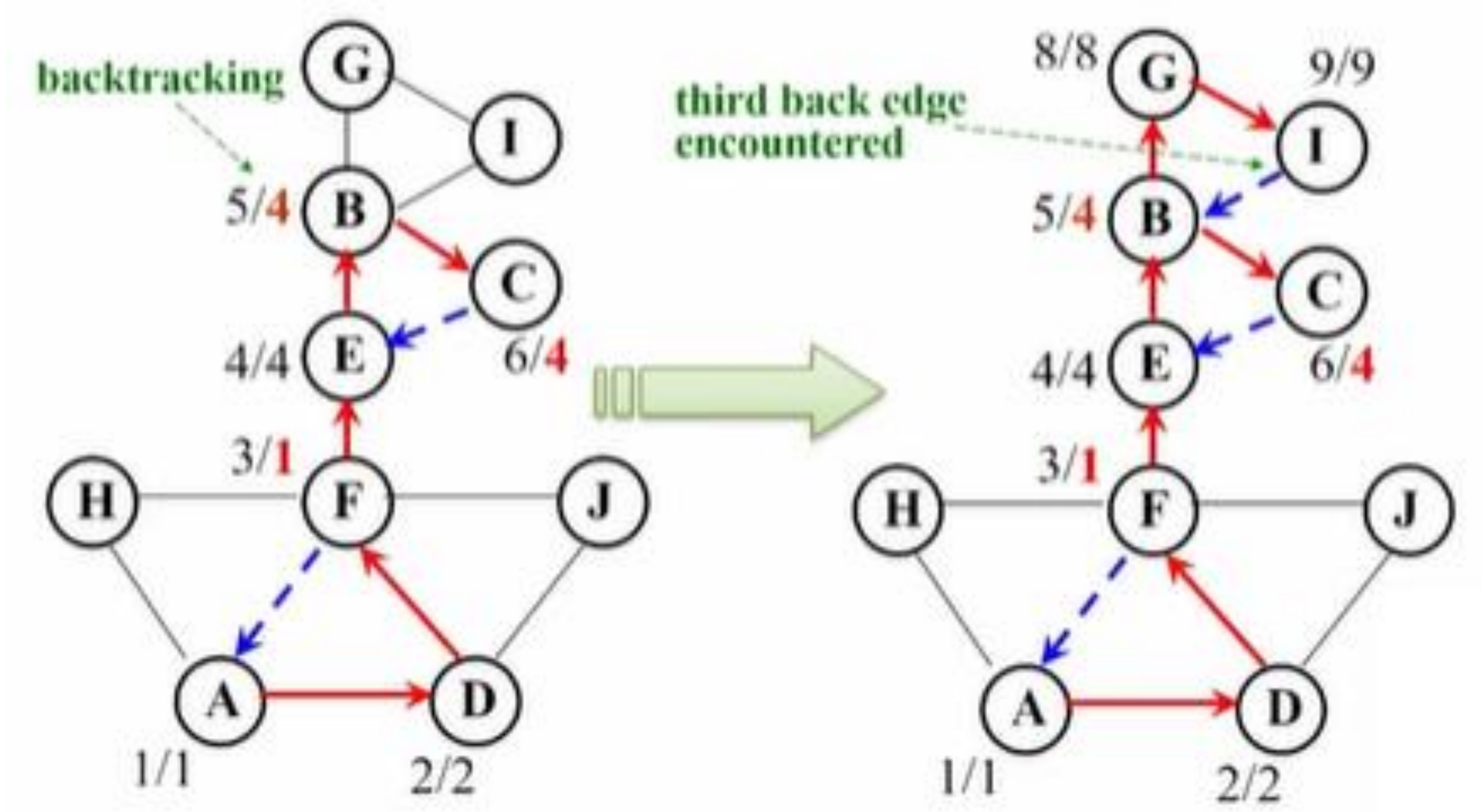
- 判断是否是割点
  - 当从 TE vw 回退时，即w 到 v 回溯时，判断
  - wBack >= discoverTime(v)

例子

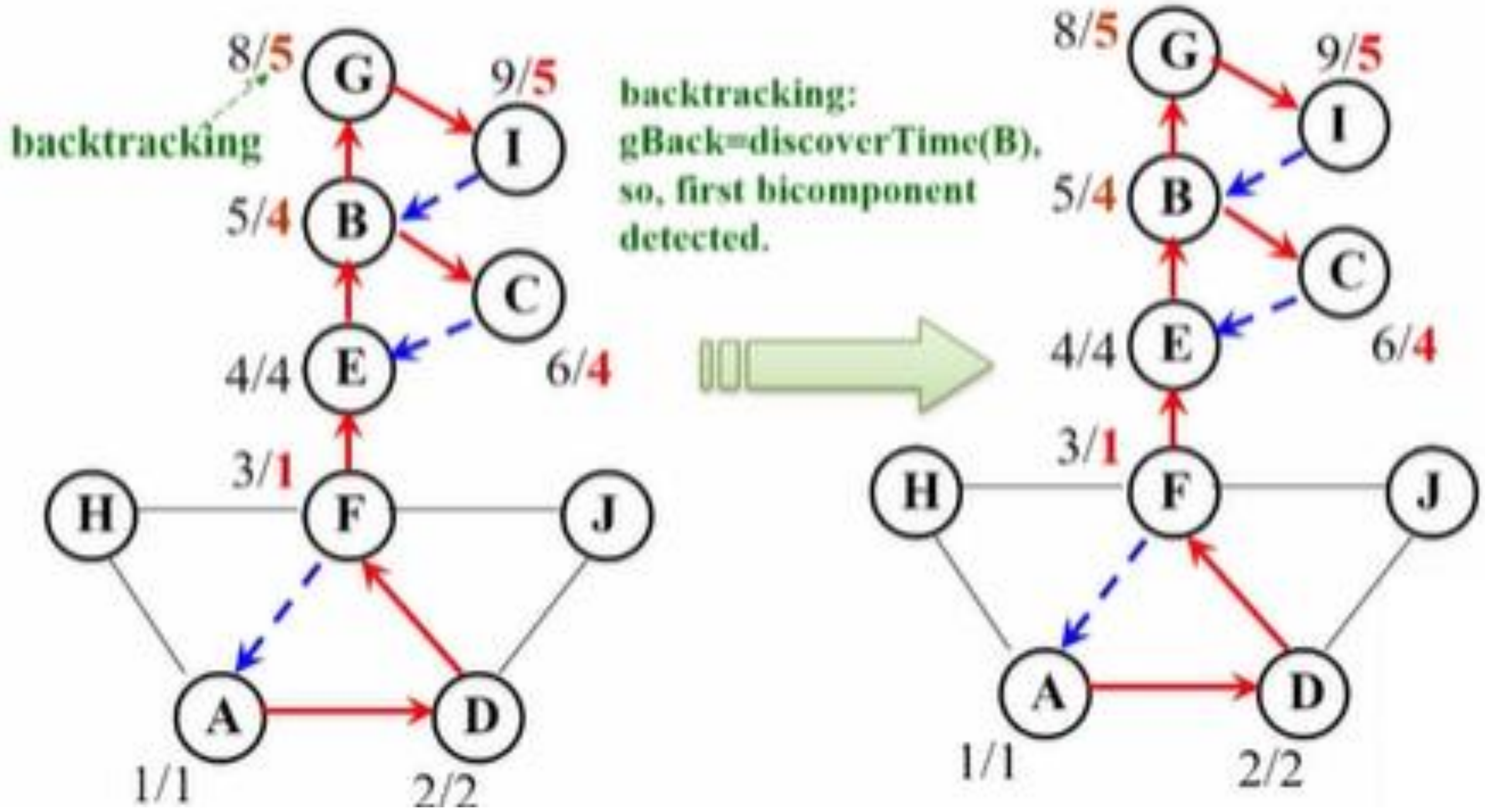


second back edge encountered
G
I
B
C
E
3/3
H
F
J
A
D
1/1
2/2
first back edge encountered
5/5
4/4
6/6
3/1

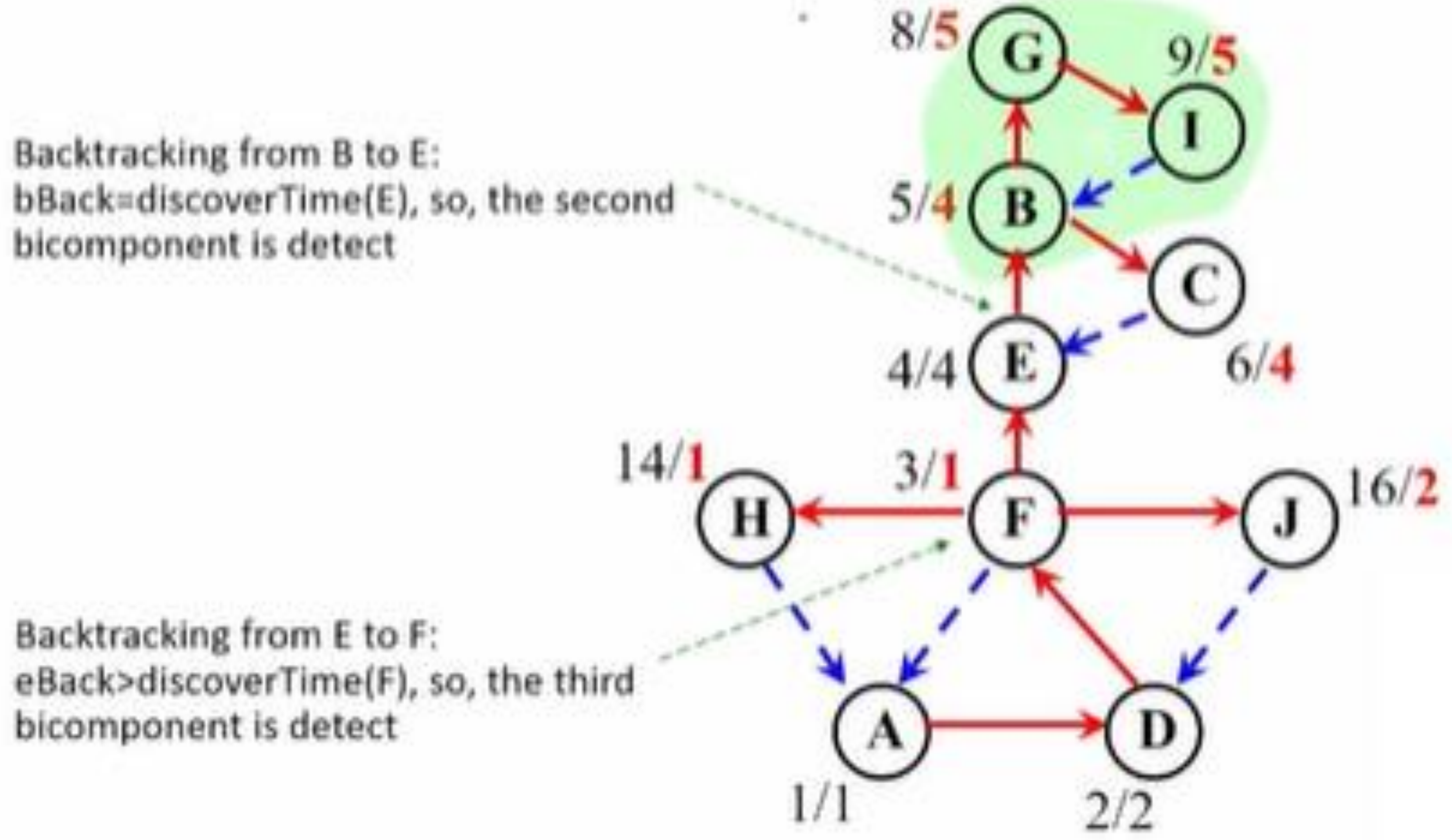
例子


backtracking
G
I
5/4
B
C
E
4/4
6/4
3/1
H
F
J
A
D
1/1
2/2
8/8
9/9
third back edge
encountered
G
I
5/4
B
C
4/4
E
6/4
3/1
H
F
J
A
D
1/1
2/2

例子


8/5
G
9/5
I
backtracking
5/4
B
C
4/4
E
6/4
3/1
H
F
J
A
D
1/1
2/2
backtracking:
gBack=discoverTime(B),
so, first bicomponent
detected.
8/5
G
9/5
I
5/4
B
C
4/4
E
6/4
3/1
H
F
J
A
D
1/1
2/2

例子



Backtracking from B to E:
bBack=discoverTime(E), so, the second bicomponent is detect
Backtracking from E to F:
eBack>discoverTime(F), so, the third bicomponent is detect
8/5
G
9/5
I
5/4
B
C
4/4
E
6/4
14/1
3/1
H
F
J
16/2
A
D
1/1
2/2

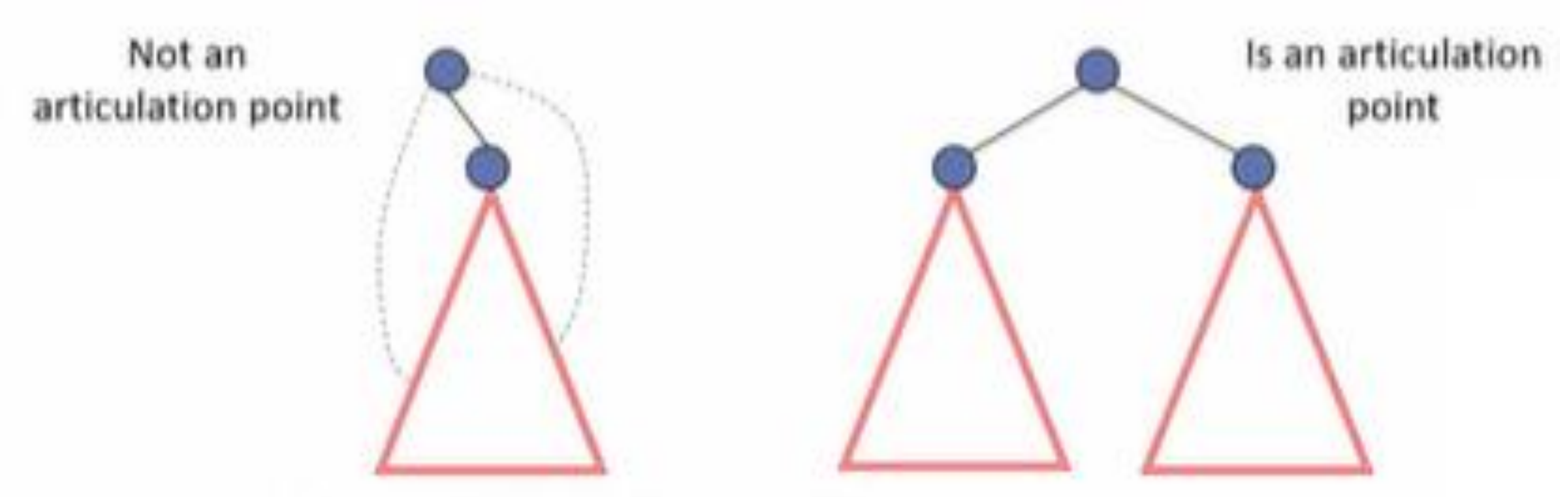
割点算法

Algorithm 12: ARTICULATION-POINT-DFS(v)

```
v.color := GRAY ;
time := time + 1 ;
v.discoverTime := time ;
v.back := v.discoverTime ;
foreach neighbor w of v do
    if w.color = WHITE then
        w.back := ARTICULATION-POINT-DFS(w) ;
        if w.back ≥ v.discoverTime then
            Output v as an articulation point ;
        v.back := min{v.back, w.back} ;
    else
        if vw is BE then                        /* w 是 v 非父节点的祖先节点 */
            v.back := min{v.back, w.discoverTime} ;
return back ;
```

Root 判断

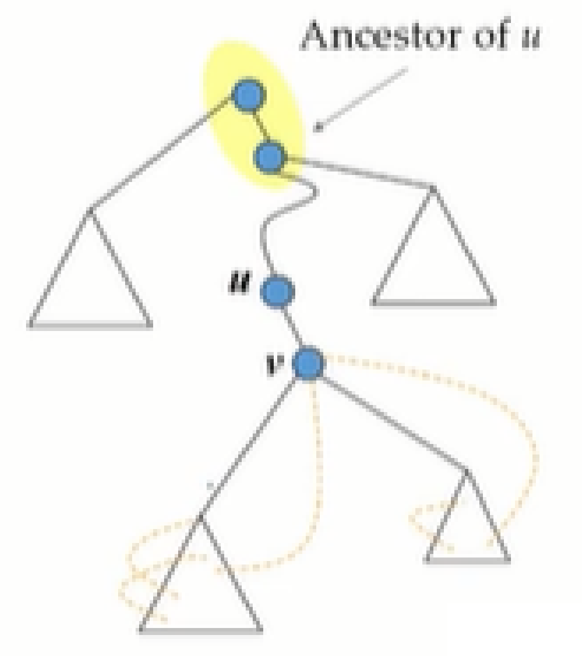
- 考虑一个连通片的单个 DFS tree
- Root 是割点 当且仅当 有两个或更多的子树



桥的定义

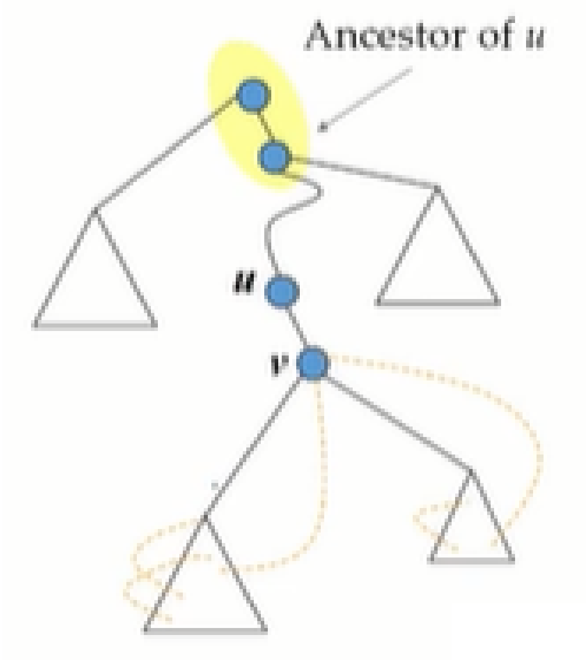
- 基于DFS的定义：给定遍历树中的TE边uv(u是v的父节点)，uv是桥，当且仅当以 v 为根的所有遍历树子树中没有 BE 指向 v 的祖先。

- v 第一次发现
  - vback = discoverTime(v)
- 遍历 BE vw 时
  - vback = min(vback, discoverTime(w))
- 遍历 w 结束，回退到 v 的时候
  - vback = min(vback, wback)



桥的判断

- 对于TE uv，当算法遍历完节点v，向节点u回退时，如果 v.back>u.discoverTime, 则uv是桥。



桥—算法

Algorithm 11: BRIDGE-DFS(u)

```
u.color := GRAY ;
time := time + 1 ;
u.discoverTime := time ;
u.back := u.discoverTime ;
foreach neighbor v of u do
    if v.color = WHITE then
        BRIDGE-DFS(v) ;
        u.back := min{u.back, v.back} ;
        if v.back > u.discoverTime then
            Output uv as a bridge ;
    else
        if uv is BE then                    /* v 是 u 非父节点的祖先节点 */
            u.back := min{u.back, v.discoverTime} ;
```

Thanks