



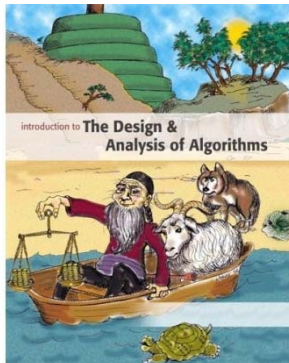
南京大學

NANJING UNIVERSITY

Introduction to

Algorithm Design and Analysis

[13] Undirected Graph



Yu Huang

<http://cs.nju.edu.cn/yuhuang>
Institute of Computer Software
Nanjing University





In the Last Class...

- **Directed Acyclic Graph**
 - Topological Order
 - Critical Path Analysis
- **Strongly Connected Component**
 - Strong Component and Condensation
 - Finding SCC based on DFS



DFS on Undirected Graph

- **Undirected Graph**
 - Symmetric Digraph 
 - Undirected Graph DFS Skeleton
- **Biconnected Components** 
 - *Articulation Points* and 2-point connectedness
 - *Bridge* and 2-edge connectedness
- **Other undirected graph problems**
 - Orientation of an undirected graph
 - Warm up: Minimum Spanning Tree based on DFS



What is Different for “Undirected”

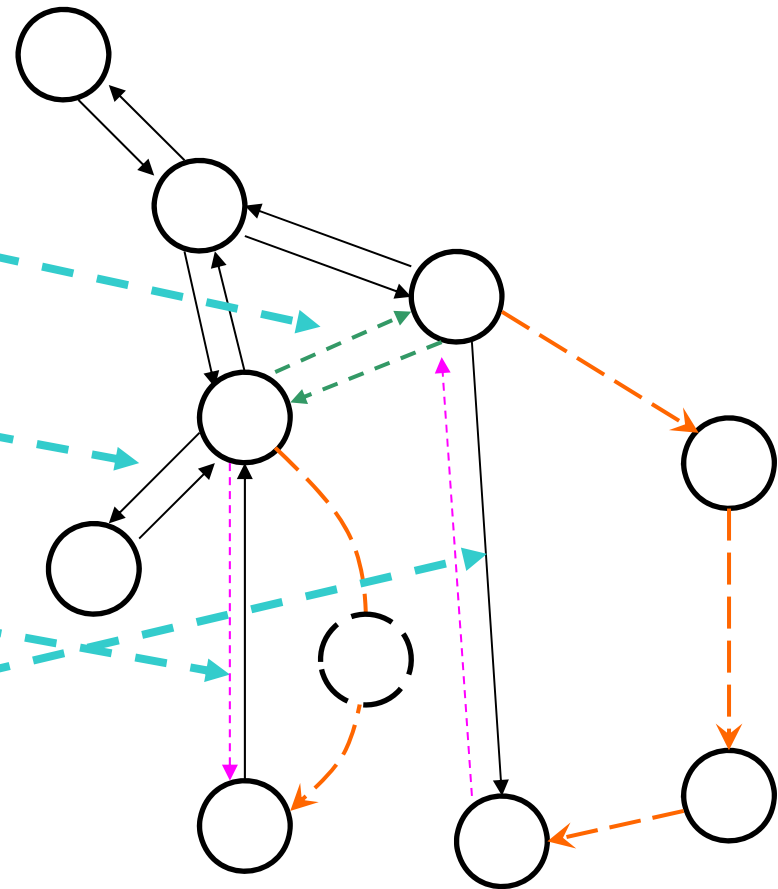


- **Characteristics of undirected graph traversal**
 - One edge may be traversed for **two times** in opposite directions.
- **For an undirected graph, DFS provides an orientation for each of its edges**
 - Oriented in the direction in which they are first encountered.



Edges in DFS

- **Cross edge**
 - Not existing
- **Back edge**
 - Back to the direct parent: **second encounter**
 - Otherwise: **first encounter**
- **Forward edge**
 - Always **second encounter, and first time as back edge**



Modifications to the DFS Skeleton



- All the **second encounter** are **bypassed**.
- So, the *only substantial modification* is for the possible back edges leading to an ancestor, but not direct parent.
- We need know the *parent*, that is, the direct ancestor, for the vertex to be processed.



DFS Skeleton for Undirected Graph

- `int dfsSweep(IntList[] adjVertices, int n, ...)`
- `int ans;`
- **<Allocate color array and initialize to white>**
- For each vertex v of G , in some order
- `if (color[v]==white)`
- `int vAns=dfs(adjVertices, color, v, -1, ...);`
- **<Process vAns>**
- `// Continue loop`
- `return ans;`



Recording the parent

DFS Skeleton for Undirected Graph

- `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;`




Complexity of Undirected DFS


- $\Theta(m+n)$
 - If each inserted statement for specialized application runs in constant time
 - The same with directed graph DFS
- **Extra space $\Theta(n)$**
 - For array *color*, or activation frames of recursion.



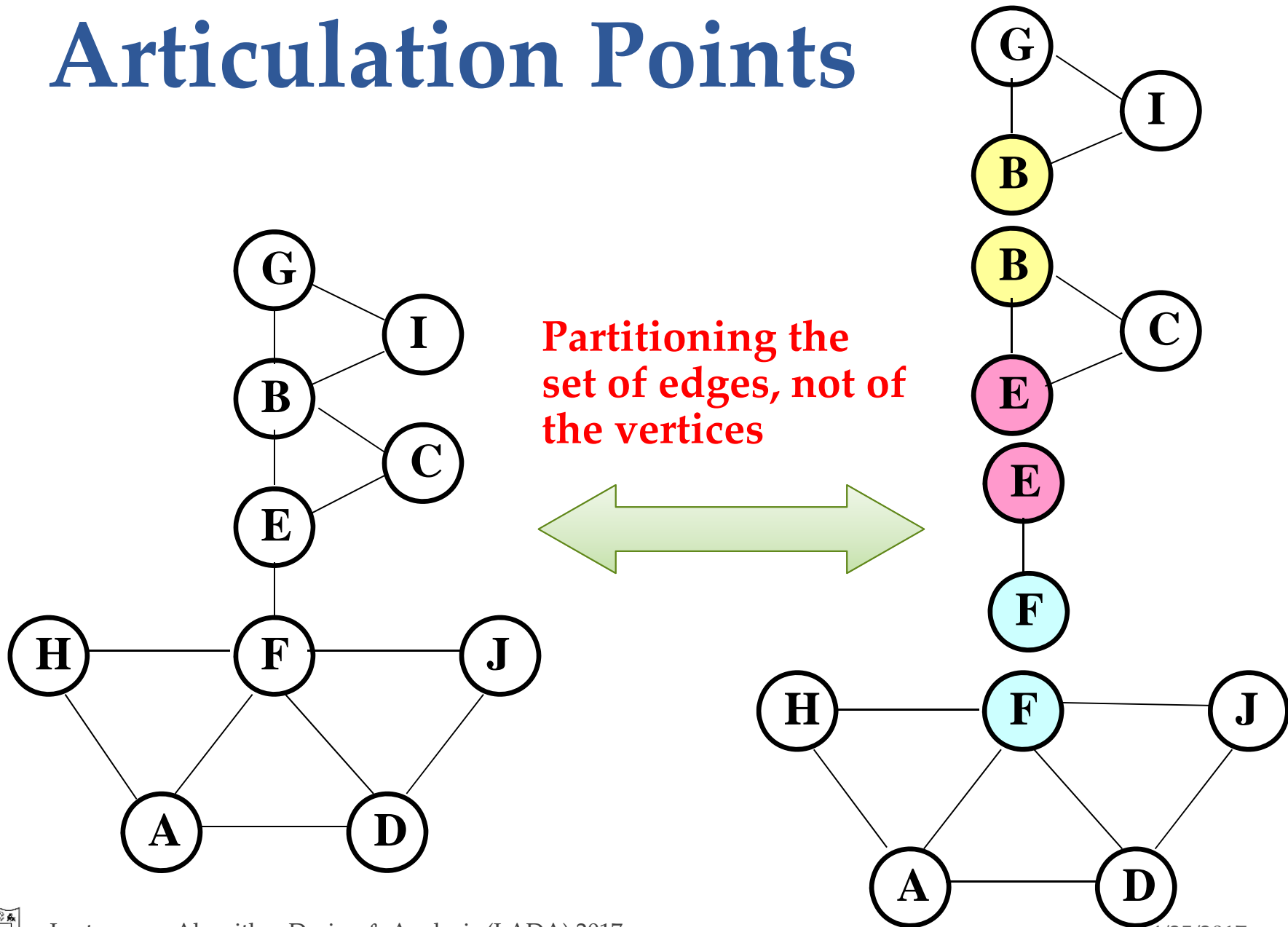
Biconnected Graph

- **Being connected**
 - Tree: acyclic, least (cost) connected
 - Node/edge connected: fault tolerant connection
- **Articulation point (2-node connected)** 
 - v is an articulation point if deleting v leads to disconnection
- **Bridge (2-edge connected)**
 - uv is a bridge if deleting uv leads to disconnection

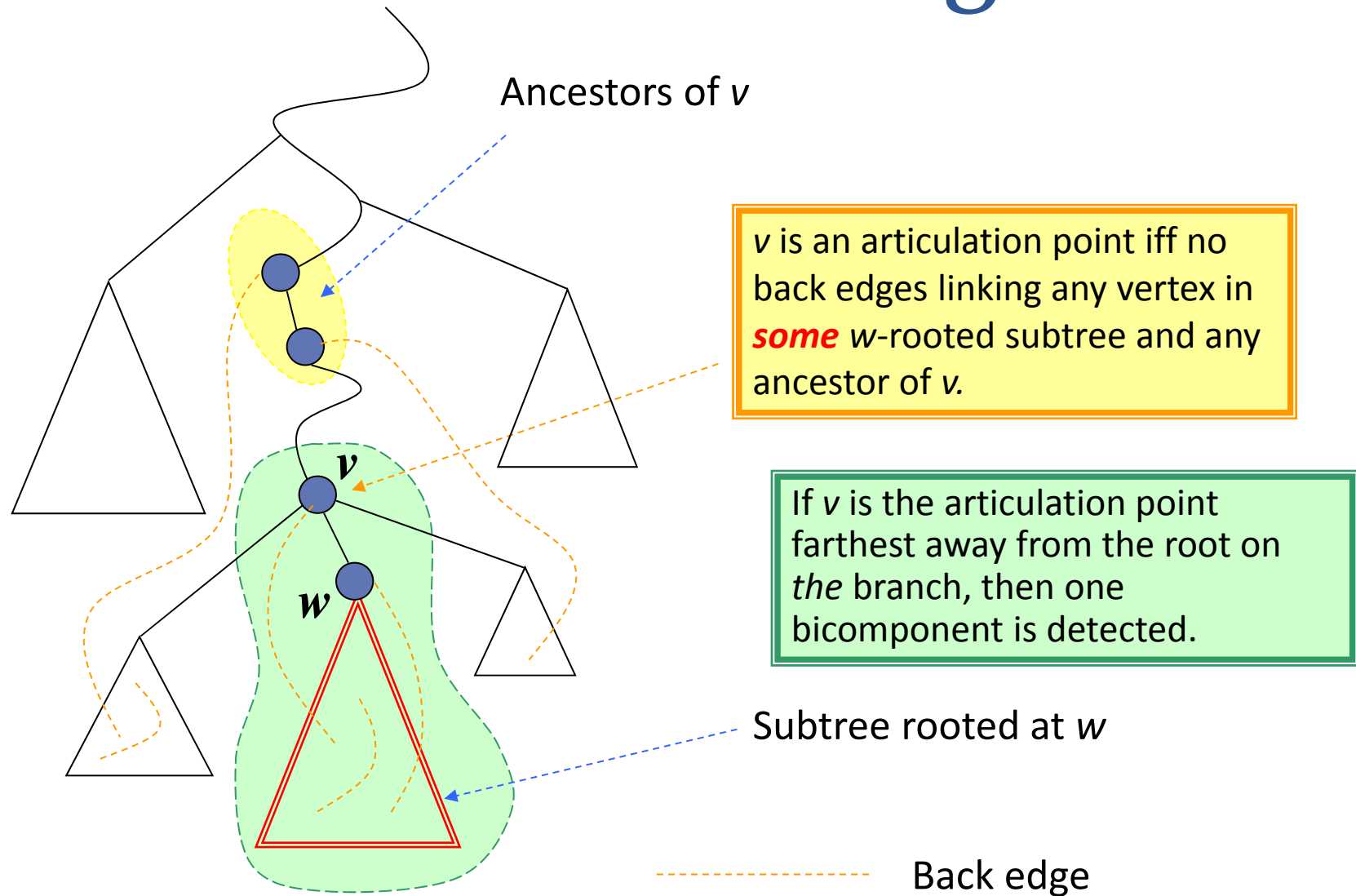
Biconnected Graph

- **Articulation point**
 - v is an articulation point if there **exist** nodes w and x , such that v is in **every** path from w to x (w and x are vertices different from v)
- **Bridge**
 - uv is a bridge if node u and v are connected **only** by edge uv
- **Bicomponent**
 - Maximal biconnected subgraph 

Articulation Points



Articulation Point Algorithm



Updating the value of *back*

- v first discovered

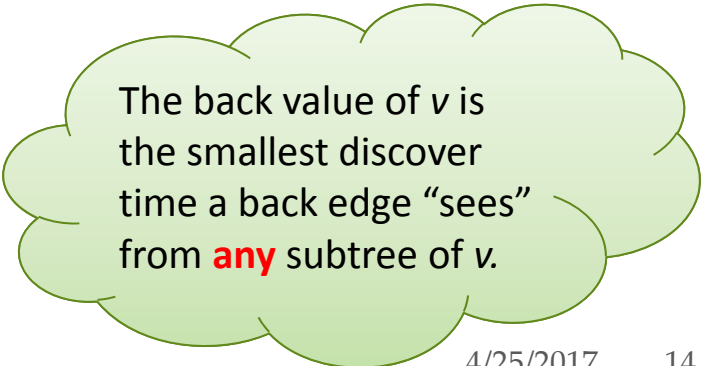
$back = discoverTime(v)$

- Trying to explore, but a back edge vw from v encountered

$back = \min(back, discoverTime(w))$

- Backtracking from w to v

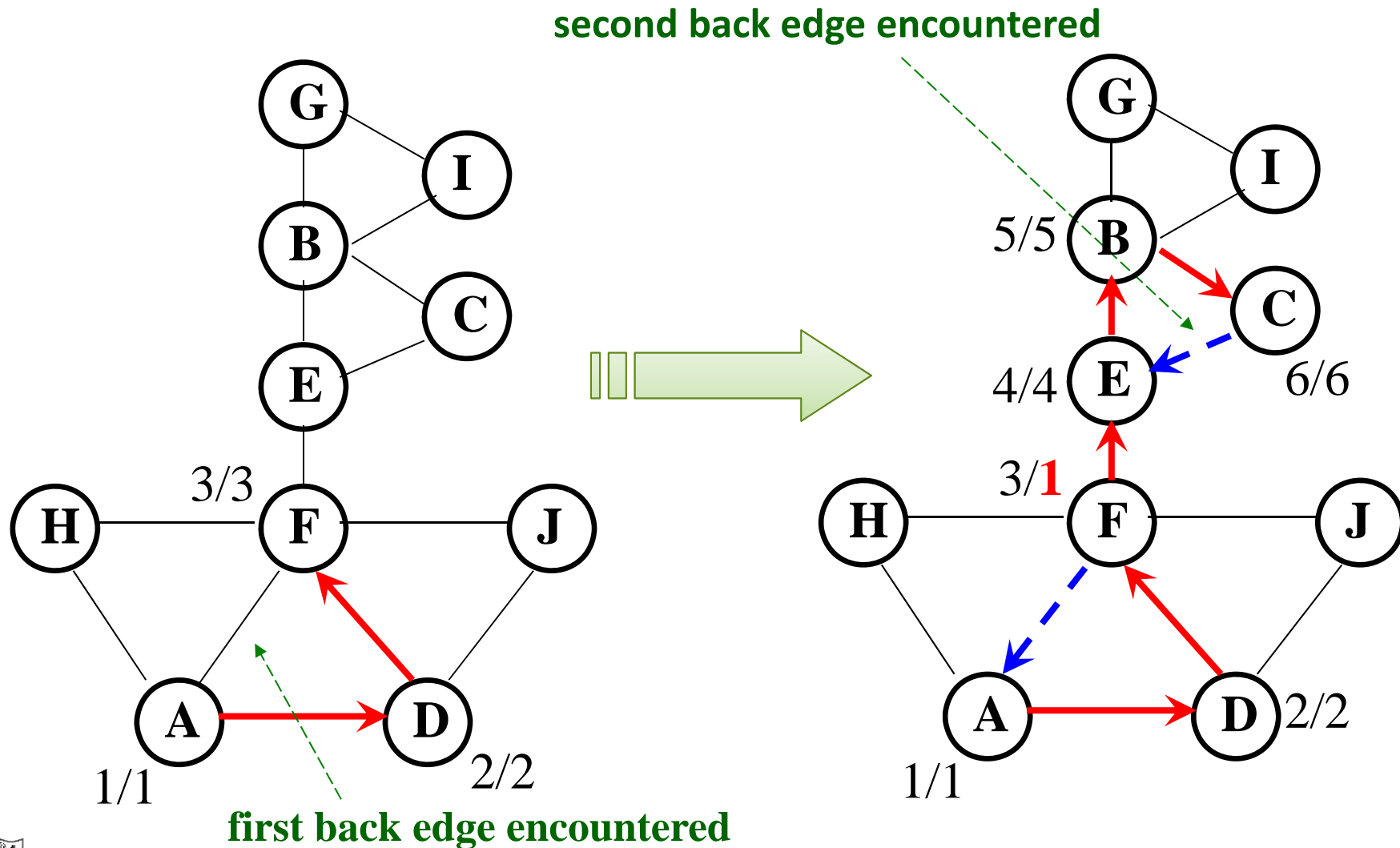
$back = \min(back, wback)$



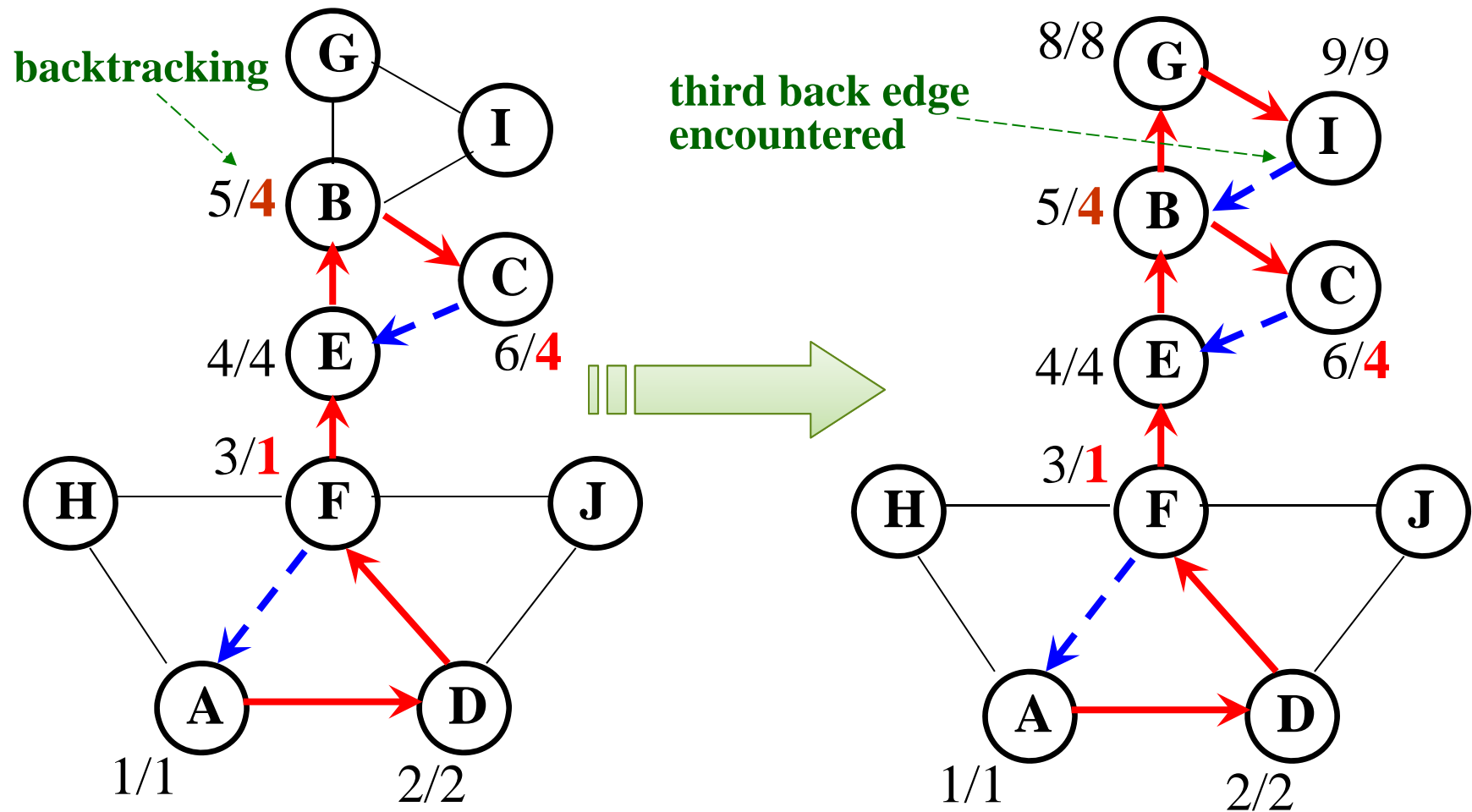
The back value of v is the smallest discover time a back edge “sees” from **any** subtree of v .



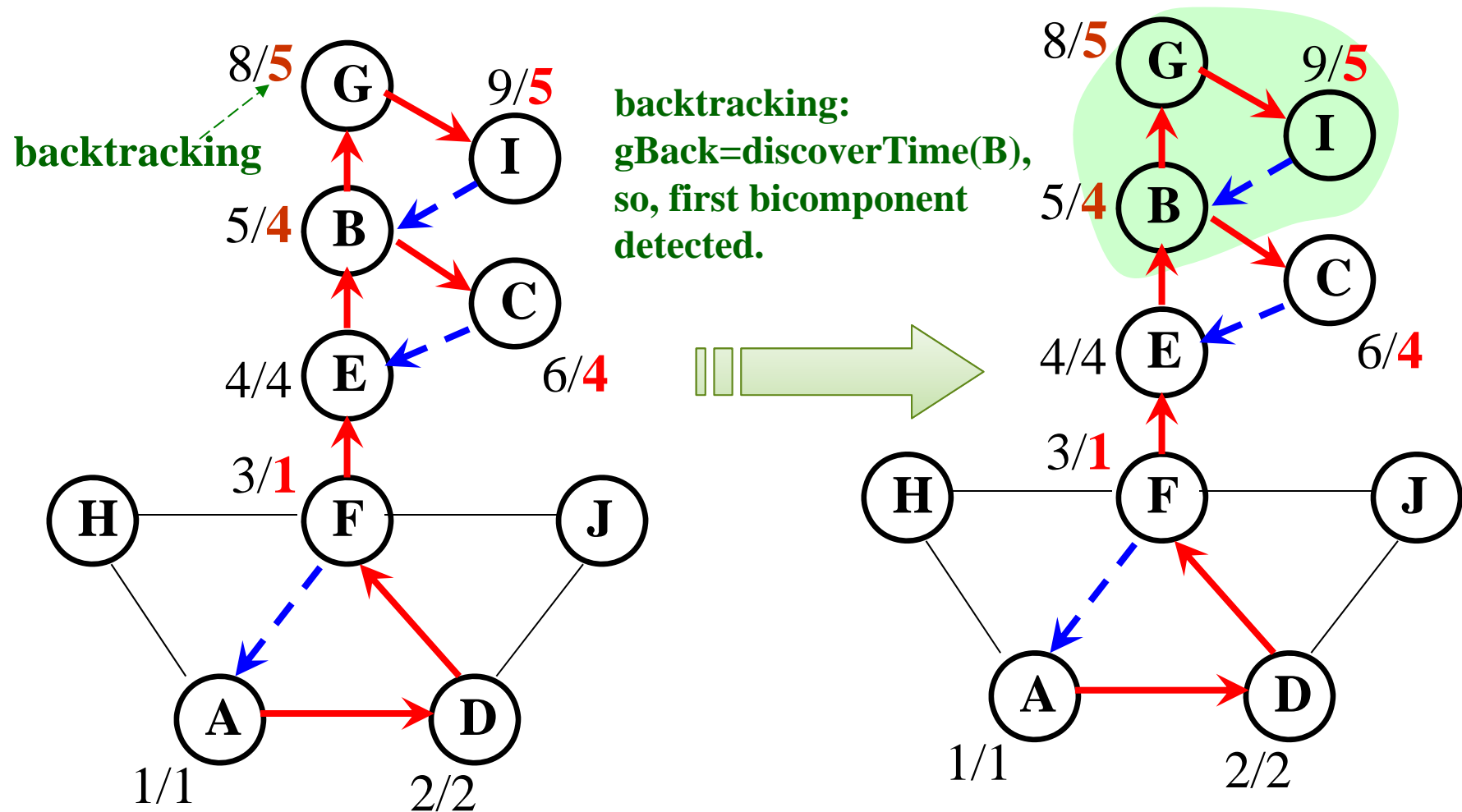
Example



Example



Example



Keeping the Track of Backing

- **Tracking data**
 - For each vertex v , a local variable $back$ is used to store the required information, as the value of *discoverTime* of some vertex.
- **Testing for bicomponent**
 - At backtracking from w to v , the condition implying a bicomponent is:
 - $wBack \geq discoverTime(v)$
(where $wBack$ is the returned back value for w)

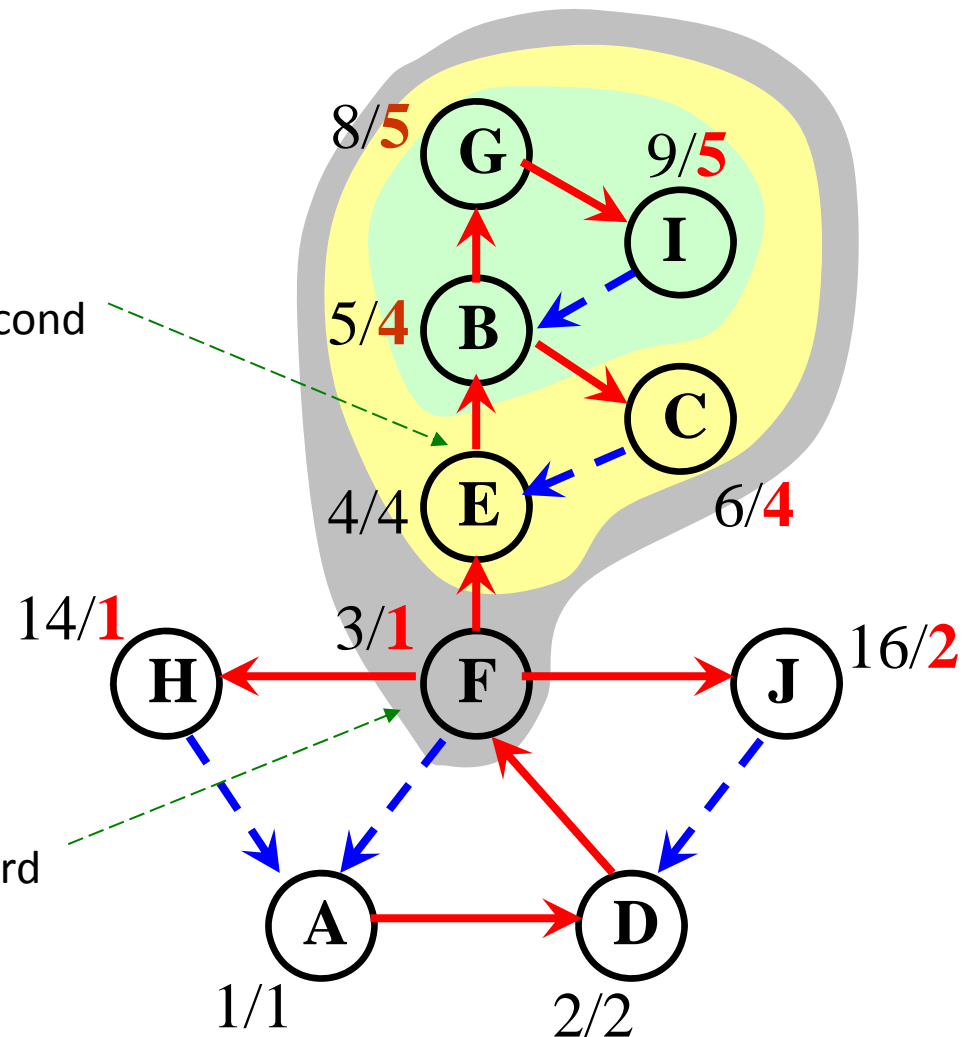
when $back$ is no less than the discover time of v , there is at least one subtree of v connected to other part of the graph only by v .



Example

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



Articulation Point Algorithm

- `int bcompDFS(v)`
- `color[v]=gray; time++; discoverTime[v]=time;`
- `back=discoverTime[v];`
- `while (there is an untraversed edge vw)`
- `<push vw into edgeStack>`
- `if (vw is a tree edge)`
- `wBack=bcompDFS(w);`
- `if (wBack \geq discoverTime[v])`
- `Output a new bicomponent`
- `by popping edgeStack down through vw ;`
- `back=min(back, wBack);`
- `else if (vw is a back edge)`
- `back=min(discoverTime[w], back);`
- `time++; finishTime[v]=time; color[v]=black;`
- `return back;`

Outline of
core procedure



Correctness

- **We have seen that:**
 - If v is the articulation point farthest away from the root on the branch, then one bicomponent is detected.
- **So, we need only prove that:**
 - In a DFS tree, a vertex(not root) v is an articulation point **if and only if** (1) v is not a leaf; (2) **some** subtree of v has **no back edge** incident with a proper ancestor of v .

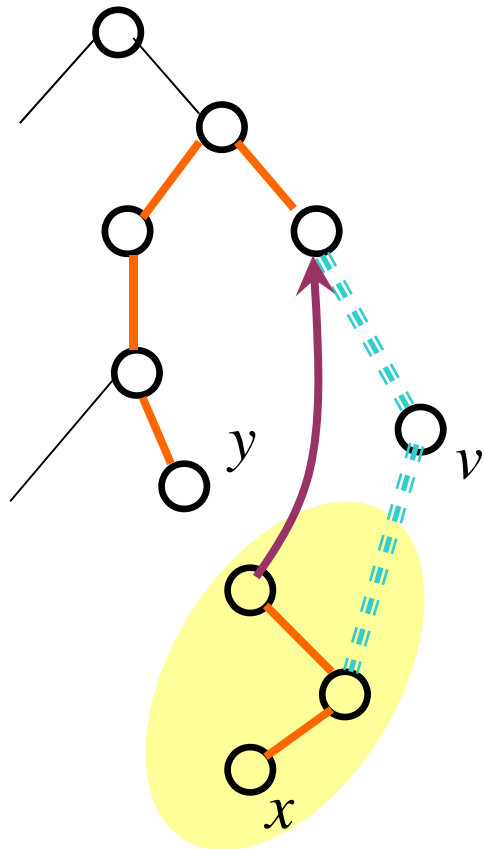


Characteristics of Articulation Point

- In a DFS tree, a vertex(not root) v is an articulation point **if and only if** (1) v is not a leaf; (2) **some** subtree of v has **no back edge** incident with a proper ancestor of v .
- \Leftarrow Trivial
- \Rightarrow
 - By definition, v is on **every** path between some x,y (different from v).
 - At least one of x,y is a proper descendent of v (otherwise, $x \leftrightarrow \text{root} \leftrightarrow y$ not containing v).
 - By **contradiction**, suppose that **every** subtree of v has a back edge to a proper ancestor of v , we can find a xy -path not containing v for all possible cases(only 2 cases)



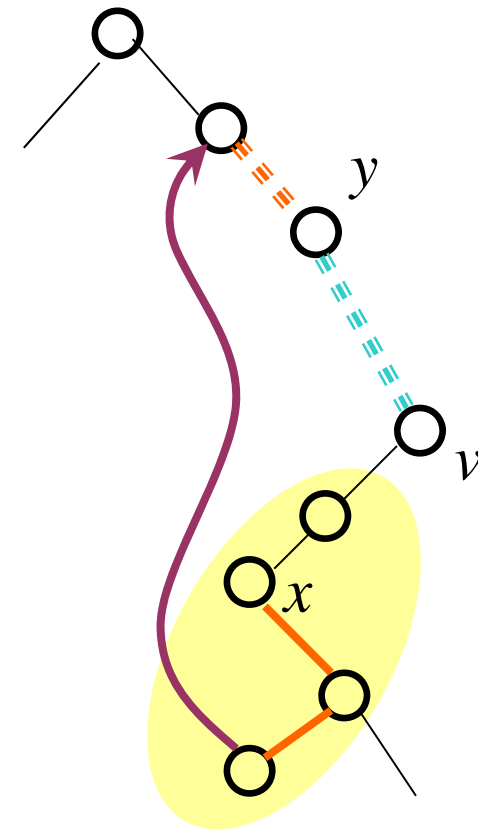
Case 1



Case 1.1: another is not an ancestor of v

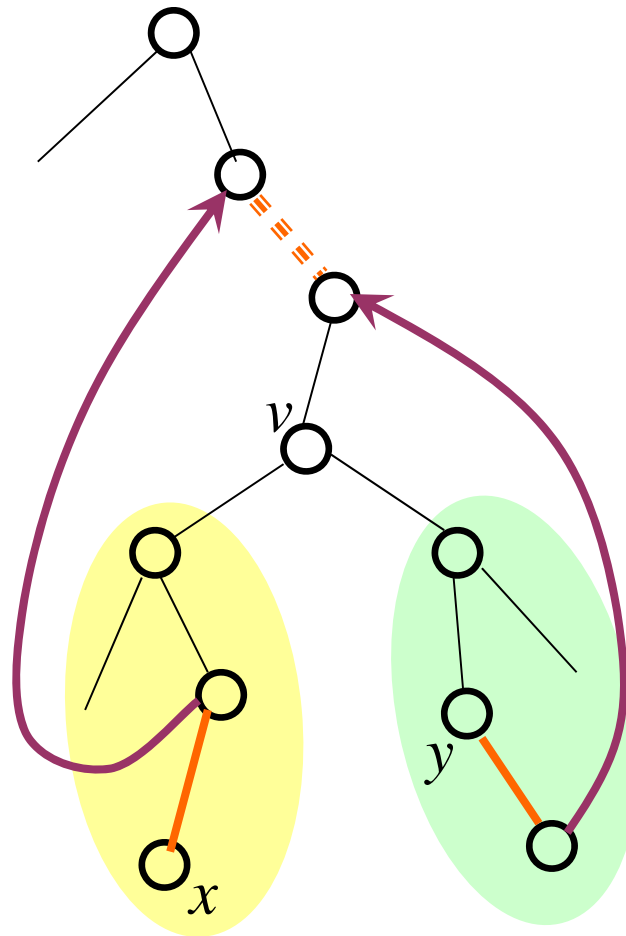
suppose that **every** subtree of v has a back edge to a proper ancestor of v , and, exactly one of x, y is a descendant of v .

Case 1.2: another is an ancestor of v

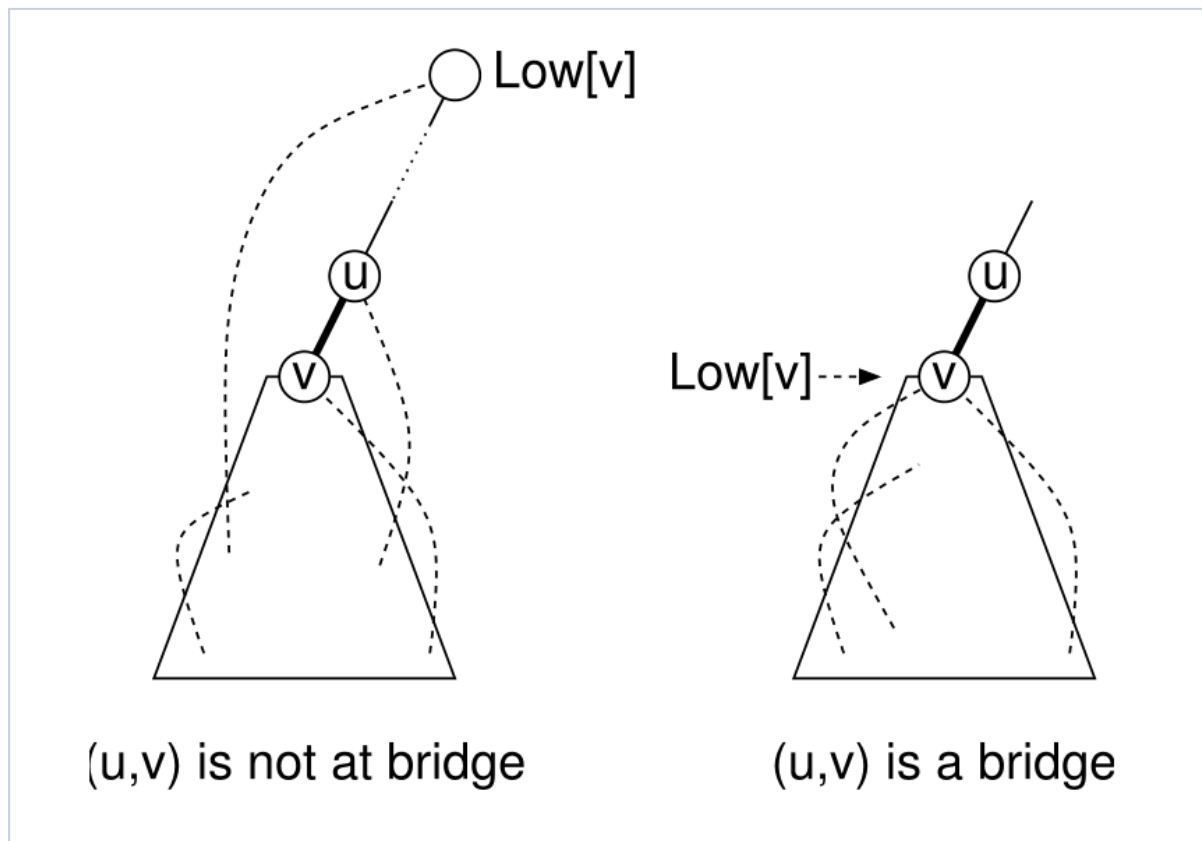


Case 2

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



Finding the Bridge



Finding the Bridge

- Edge uv is a bridge iff
 - (assuming that u is the parent and v is the child)
- a) Edge uv is a tree edge in DFS
- b) There is **no subtree rooted at v** to a proper ancestor of v (including u)



Edge Finding Algorithm

Bridge Finding by DFS

```
findBridges(u) {  
    color[u] = gray;  
    Low[u] = d[u] = ++time;          // set discovery time and init Low  
    for each (v in Adj(u)) {  
        if (color[v] == white) {    // (u,v) is a tree edge  
            pred[v] = u;             // v's parent is u  
            findBridges(v);  
            Low[u] = min(Low[u], Low[v]) // update Low[u]  
            if (Low[v] > d[u]) Report that (u,v) is a bridge  
        }  
        else if (v != pred[u])      // (u,v) is a back edge  
            Low[u] = min(Low[u], d[v]) // update Low[u]  
    }  
}
```






Orientation for Undirected Graphs

- **Orientation**
 - Give each edge a direction
 - Satisfying pre-specified constraints
 - E.g., the “in-degree of each vertex is at least 1”
- **Possible or not?**
 - If possible, how to?
- **As for “in-degree ≥ 1 ”**
 - Orientation possible iff. the graph has at least a circle
 - Find the end point of some back edge
 - A second DFS from this end point



Warm Up for MST

MST: Minimum
Spanning Tree

- **Get MST in $O(m+n)$ time**
 - Given that edges weights are only 1 and 2
- **Graph traversal is sufficient** 
 - DFS over “weight 1 edges” only
 - DFS over “weight 2 edges” only



Thank you!

Q & A

Yu Huang

<http://cs.nju.edu.cn/yuhuang>

