CS 381 – FALL 2019

Week 9.1, Monday, Oct 14

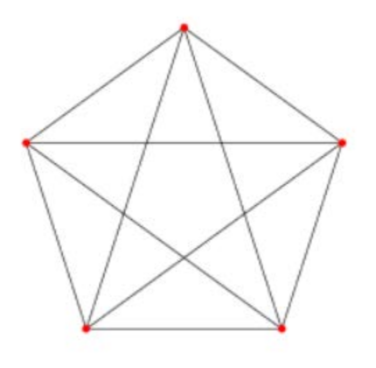
Homework 4 Due Tonight! October 14th @ 11:59PM (Gradescope)
Reminder: You must include a collaborator/resource (CR) statement for every problem.
Homework 5: Planning to released on Wednesday (October 16th)

Depth First Search (22.3+22.5; 251 text)

- **DFS performs a recursive exploration of a graph**
  - DFS follows a path until it is forced to back up – "backtracking"
- DFS operates on an adjacency list representation
- Most uses of DFS result in O(n+m) time
- DFS be used on directed and undirected graphs
- **Undirected graph**
  - DFS partitions the edges into tree and back edges
  - Assigns numbers to the vertices during exploration (e.g. DFS number, discovery number, finish number)

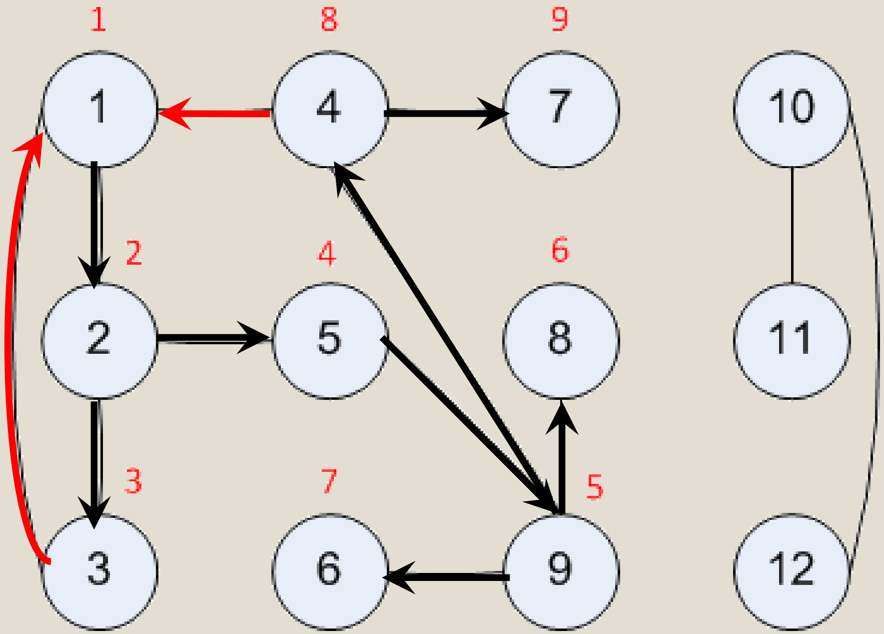
Depth First Search

- DFS can be used to solve
  - Determine *connected components* in an undirected graph (easy)
  - Test for stronger forms of connectivity: is the graph still connected if any edge/vertex is removed? *Bi-connected and bridge-connected.*
  - Determine *strongly connected components* in a directed graph
  - Test for *planarity* of a graph (can the graph be drawn without edges crossing?)



DFS in an undirected graph

DFS partitions edges into tree edges and back edges;
One often puts a direction on tree and back edges indicating who explored whom



numbers indicate the order in which vertices are explored

Generic version of DFS

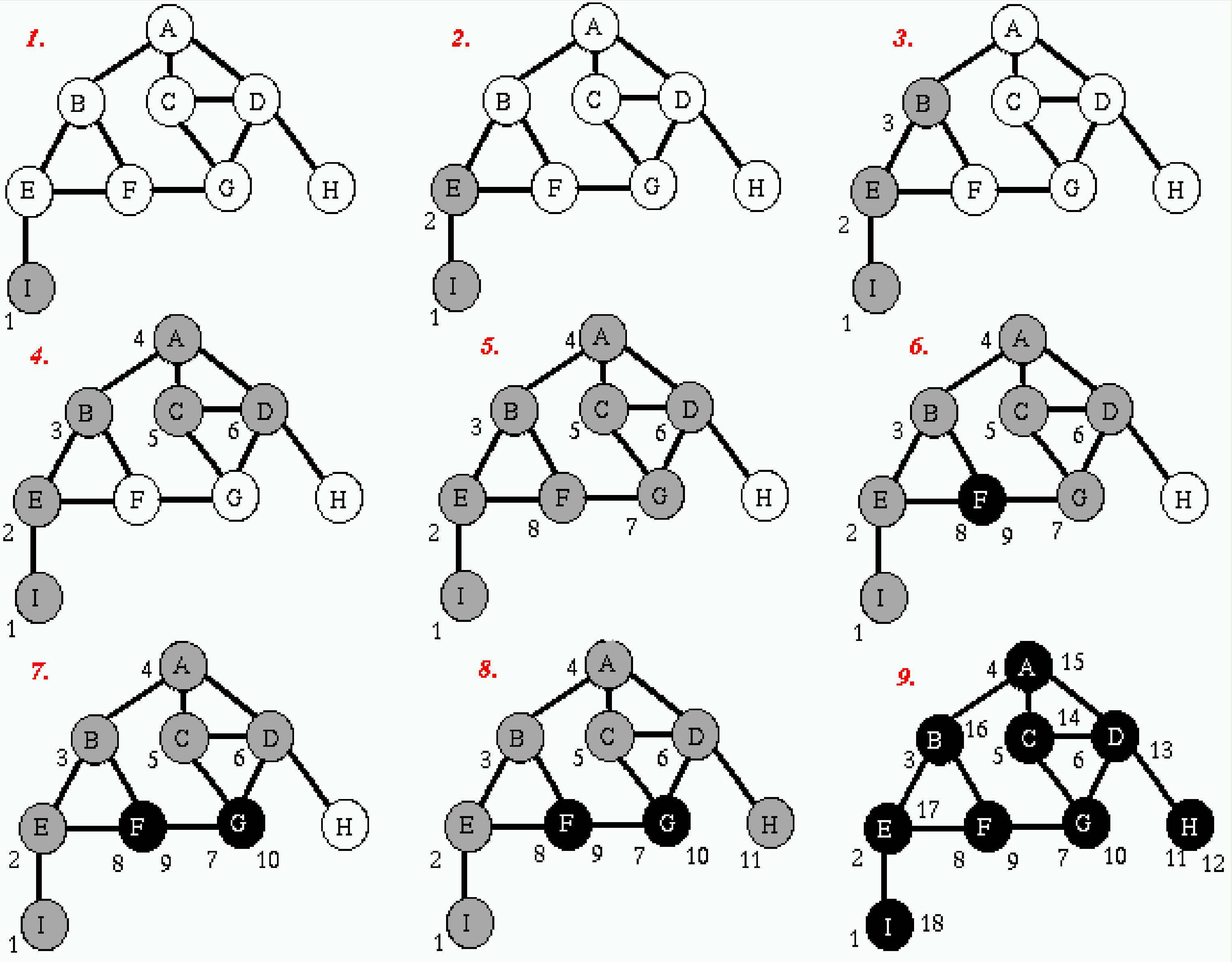
```
DFS(v)
    mark vertex v visited
    for each vertex w adjacent to v
      if w is unvisited
        DFS(w)
        add edge(v,w) to tree T
```

DFS is invoked at least once with an unvisited vertex

Book-keeping collects information:

- Vertices have a discovery and a finish time (d(v) resp. f(v))
  - Discovery time is also called the DFS number
- Vertices are white, gray, and black during exploration in text

Vertex u has discovery time d(u) and finish time f(u)



Generic version of DFS

```
Global Counter c = 1
DFS(v)
    mark vertex v visited (set d(v)=c and update c=c+1)
    for each vertex w adjacent to v
       if w is unvisited
          DFS(w)
          add edge(v,w) to tree T
    set f(v) = c and update c=c+1
```

DFS is invoked at least once with an unvisited vertex

Book-keeping collects information:

- Vertices have a discovery and a finish time (d(v) resp. f(v))
  - Discovery time is also called the DFS number
- Vertices are white, gray, and black during exploration in text

Generic version of DFS

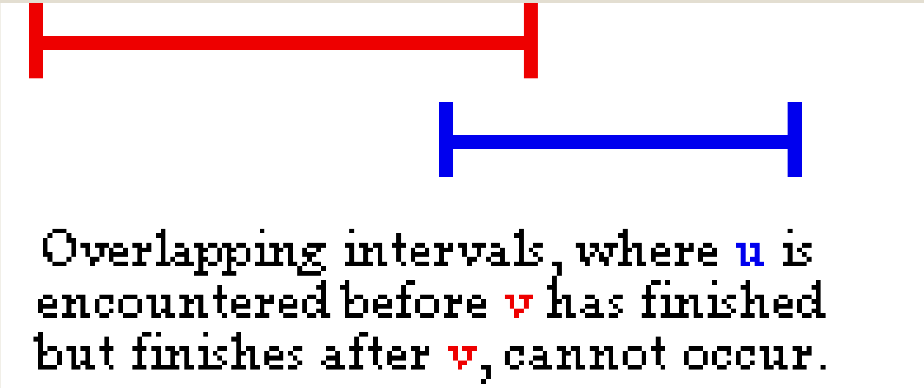
```
Global Counter c = 1
DFS(v)
    mark vertex v visited
    (set d(v)=c and update c=c+1)
    for each vertex w adjacent to v
      if w is unvisited
         DFS(w)
         add edge(v,w) to tree T
    set f(v) = c and update c=c+1
(Optional: Label All Nodes)
For each vertex v
    if v is unvisited
       DFS(v)
```

Properties of DFS numbers

For any two vertices u and v, one of the following must hold:

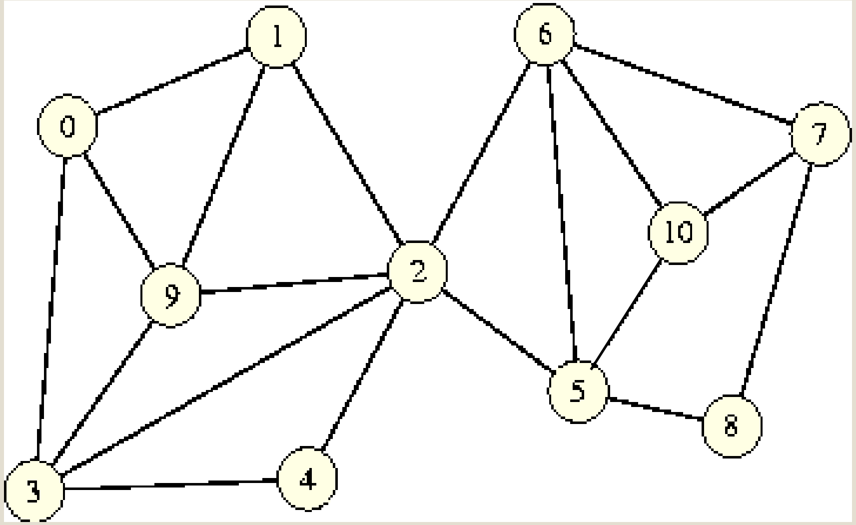
1. intervals [d(u), f(u)] and [d(v), f(v)] are disjoint; vertex u is discovered and finished before v
2. [d(u), f(u)] contains [d(v), f(v)]; v is a descendent of u
3. [d(v), f(v)] contains [d(u), f(u)]; u is a descendent of v

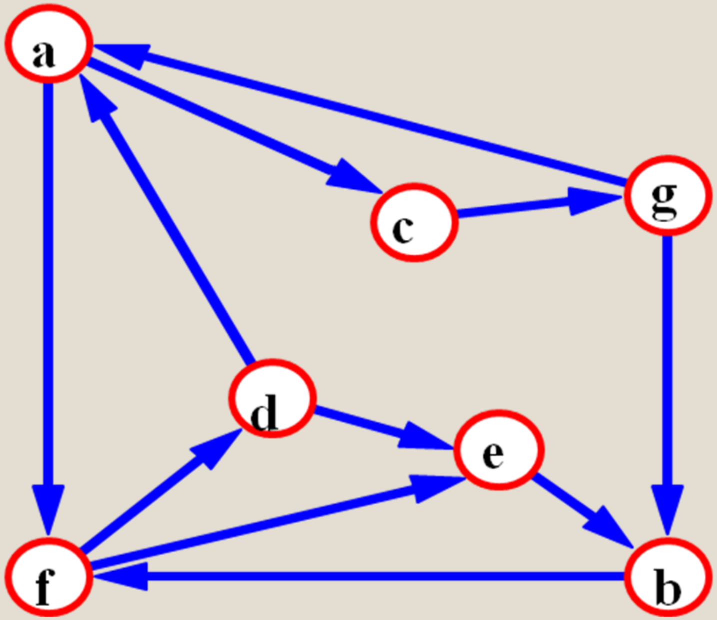
Not possible: two intervals overlap without containment:


Overlapping intervals, where u is encountered before v has finished but finishes after v, cannot occur.

O(n+m) time algorithms using DFS

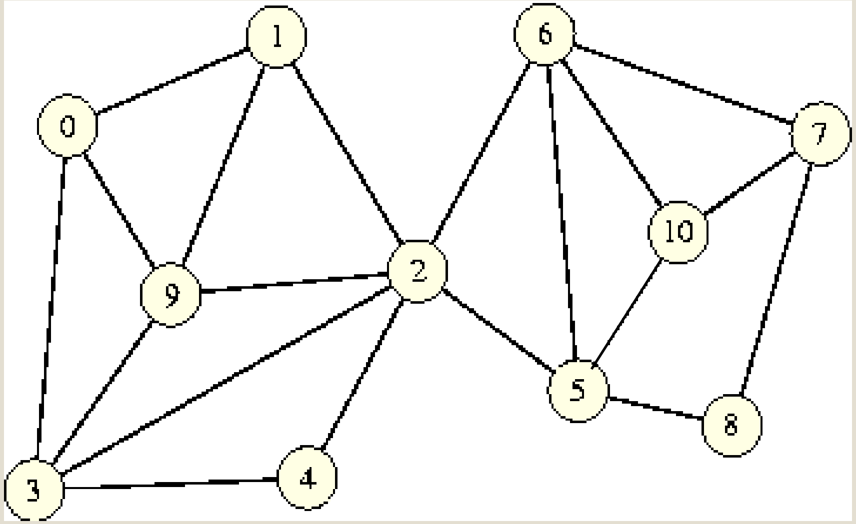
- Finding the **biconnected components** in an *undirected* graph
- Finding the **strongly connected components** in a *directed* graph





Clicker Question

Suppose we run DFS(v) starting at node 0 to assign discovery numbers (d(u)) and finish numbers (f(u)) to each node in G (see below)



No further information about order of adjacency lists e.g. we could have

AdjList(0) = 1,3,9 or

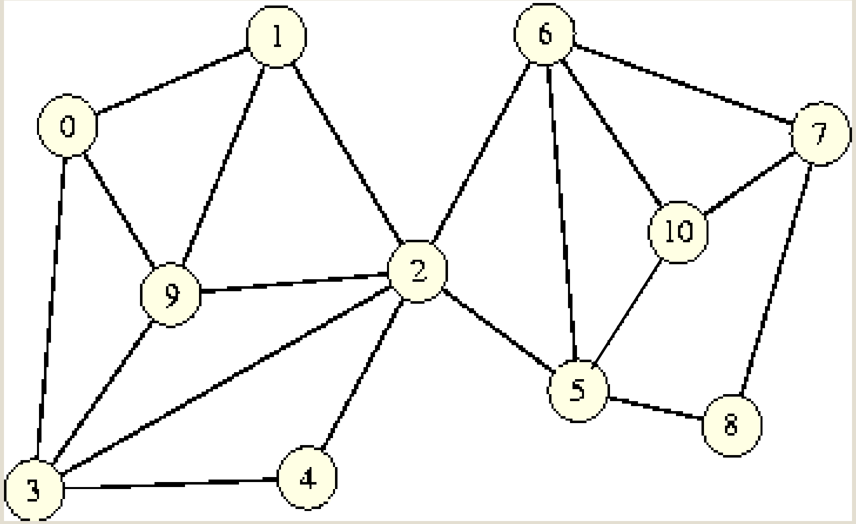
AdjList(0) = 9,3,1

Which of the following claims must be false?

A. d(0) < f(0) B. f(2) > d(4) C. f(0) > f(9) D. f(5) > f(2) E. d(2) < d(10)

Clicker Question

Suppose we run DFS(v) starting at node 0 to assign discovery numbers (d(u)) and finish numbers (f(u)) to each node in G (see below)



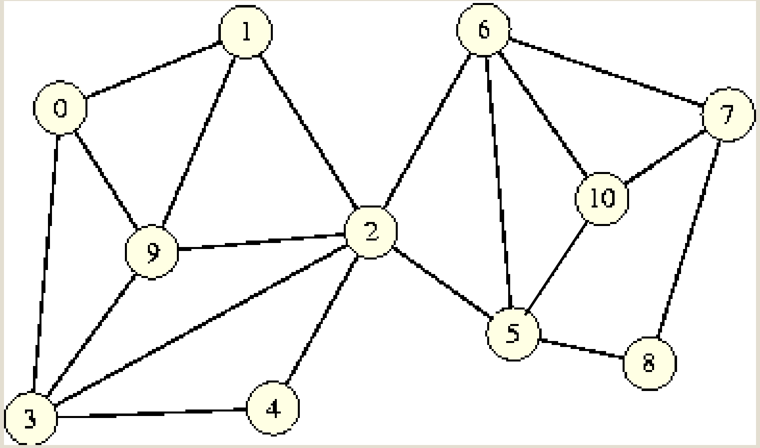
Which of the following claims must be false?

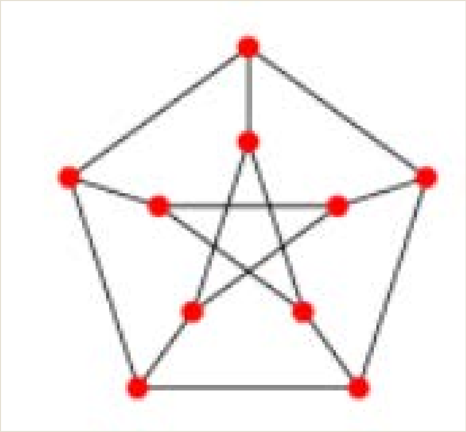
A. d(0) < f(0) B. f(2) > d(4) C. f(0) > f(9) D. f(5) > f(2) E. d(2) < d(10)

Biconnectivity in undirected, connected graphs

Vertex v is an **articulation point** if its removal results in a graph with more than one connected component.

If v is an articulation point, then there exist distinct vertices *w* and *x* such that *v* is in every path from *w* to *x*.



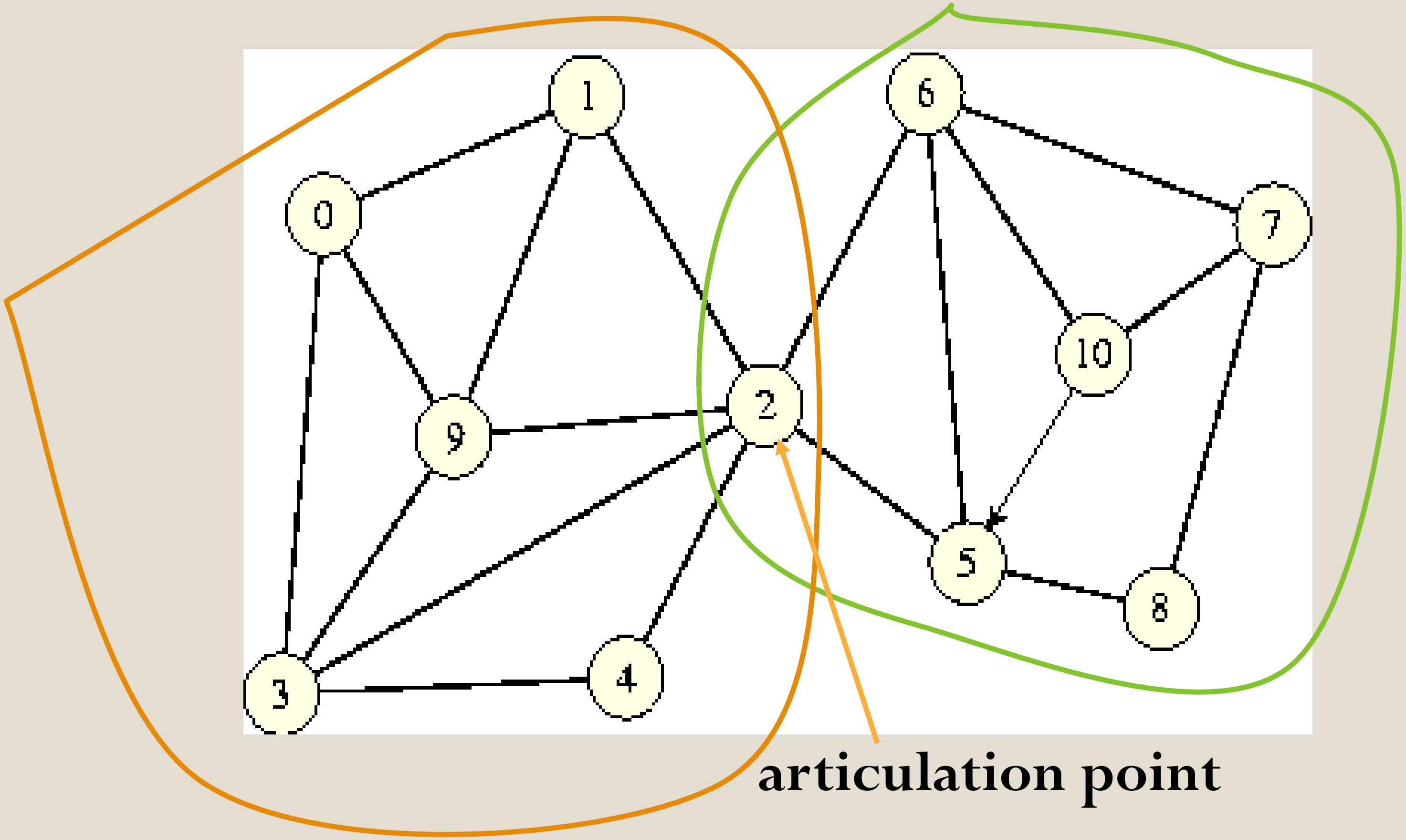


Biconnectivity

articulation point = removal disconnects the graph

A graph is **biconnected** if it contains no articulation points

- In a biconnected graph, there exist at least two vertex-disjoint paths between any pair of vertices
- The biconnected components are the largest subgraphs that are biconnected (partitions the edges, not the vertices)
- Connectivity definition can be generalized to *k*-connected graphs



articulation point

Brute-Force algorithm

- Delete a vertex and test for connectivity; repeat n times
- O(n (n+m)) time to find all articulation points and the biconnected components

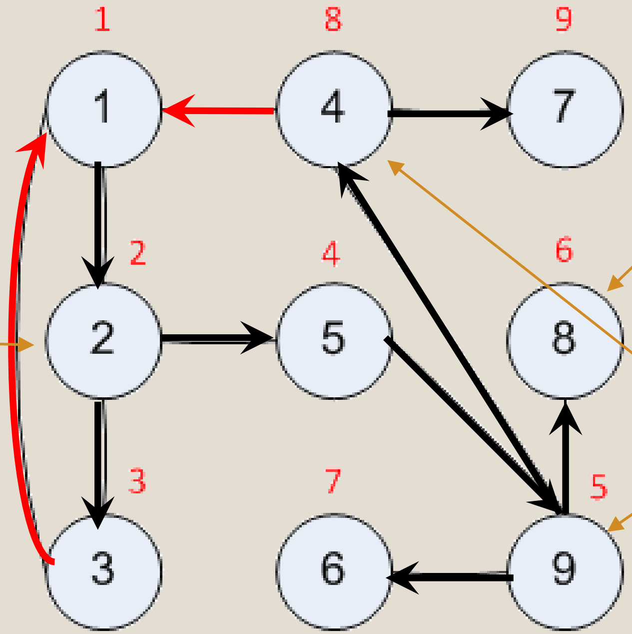
Biconnected component algorithm using DFS

Consider a DFS tree of an undirected graph G.

- The root is an articulation point if it has two or more children.
- A vertex v (other than the root) is an articulation point if and only if
  - v is not a leaf in the DFS tree and
  - some subtree rooted at a child of *v* has no back edge to a proper ancestor of *v*

Node v=2 has 2 children: 3 & 5

- Subtree rooted at 3 has back edge (3,1)
- Subtree rooted at 5 has back edge (4,1)
- Node v=2 is not an articulation point



Leaves cannot be articulation points

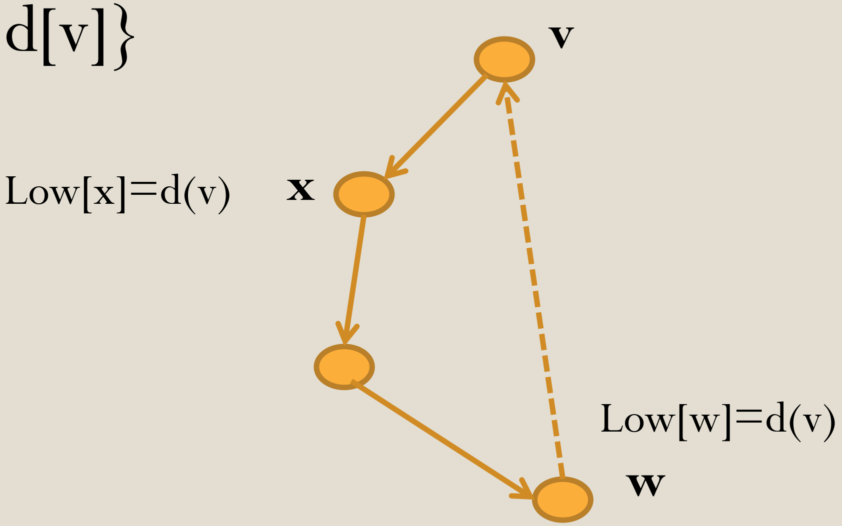
Articulation points!

Idea *During DFS keep track on how "far back up in the tree" one can get from each vertex by following tree and back edges.*

Let low [v] keep track of how "far back up the tree" one can get from a descendent of v (via back edges)

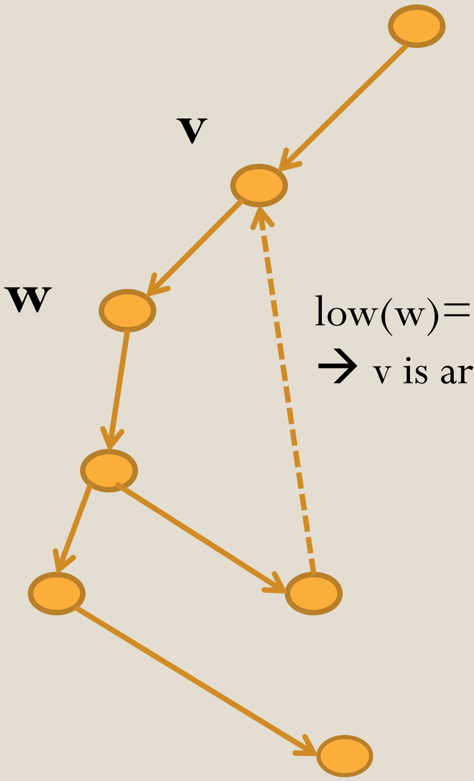
- low[v] is initialized to d[v]
- When encountering a back edge (w,v),

  low[w] = min {low[w], d[v]}



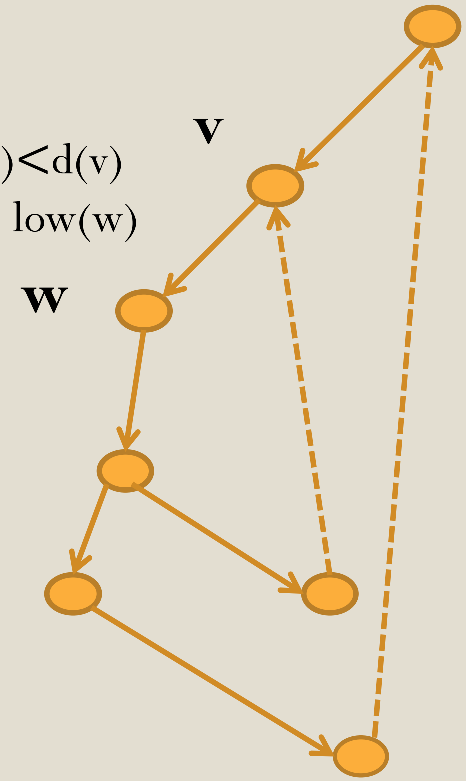
When vertex w is completely explored, w’s DFS parent v updates its low information:

if low [w] < d[v] **then** low[v] = min {low[v], low[w]}
if low [w] ≥ d[v] and v is not the root **then**
    v is an articulation point

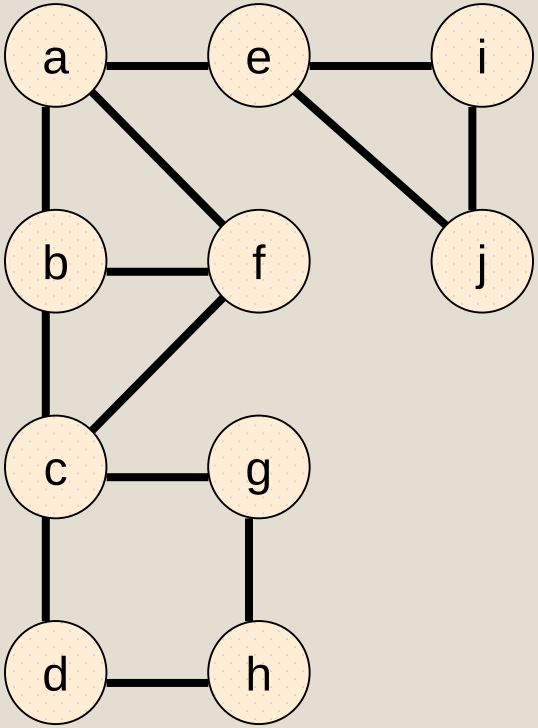


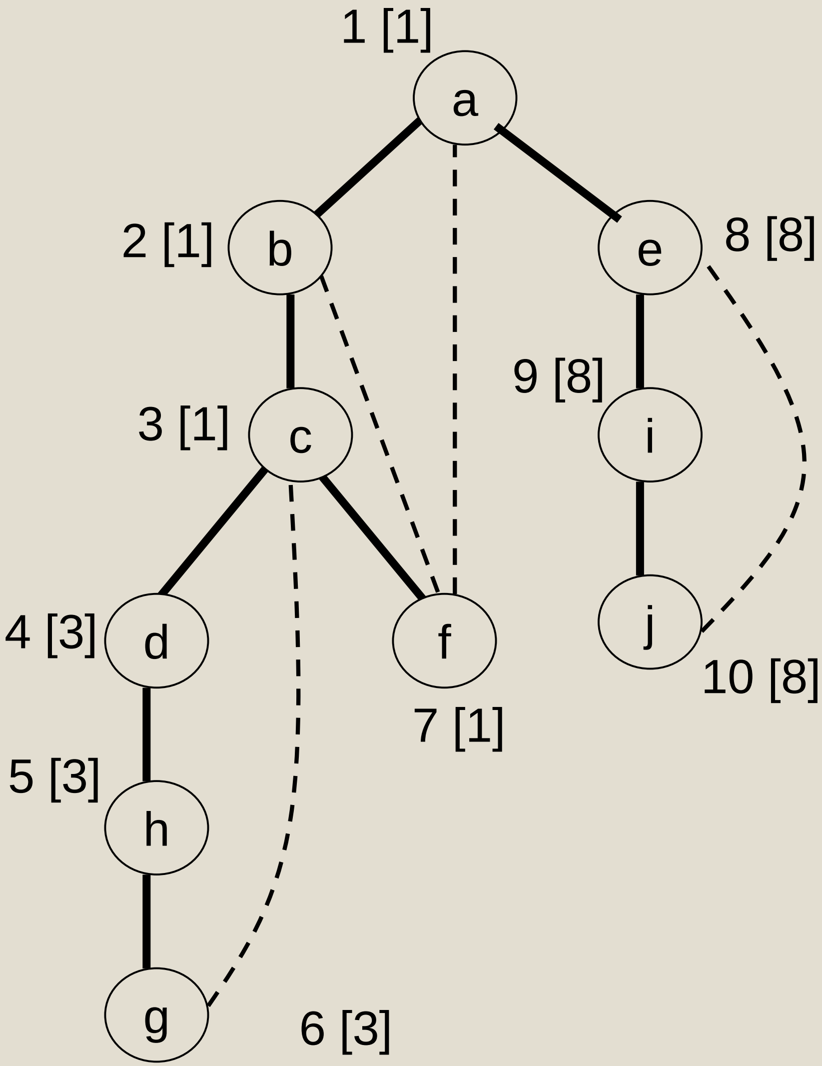
low(w)=d(v)≥d(v)
→ v is articulation point

low(w)=d(u)<d(v)
→ low(v) ≤ low(w)



d(v)[low(v)]





Articulation points: a, c, e

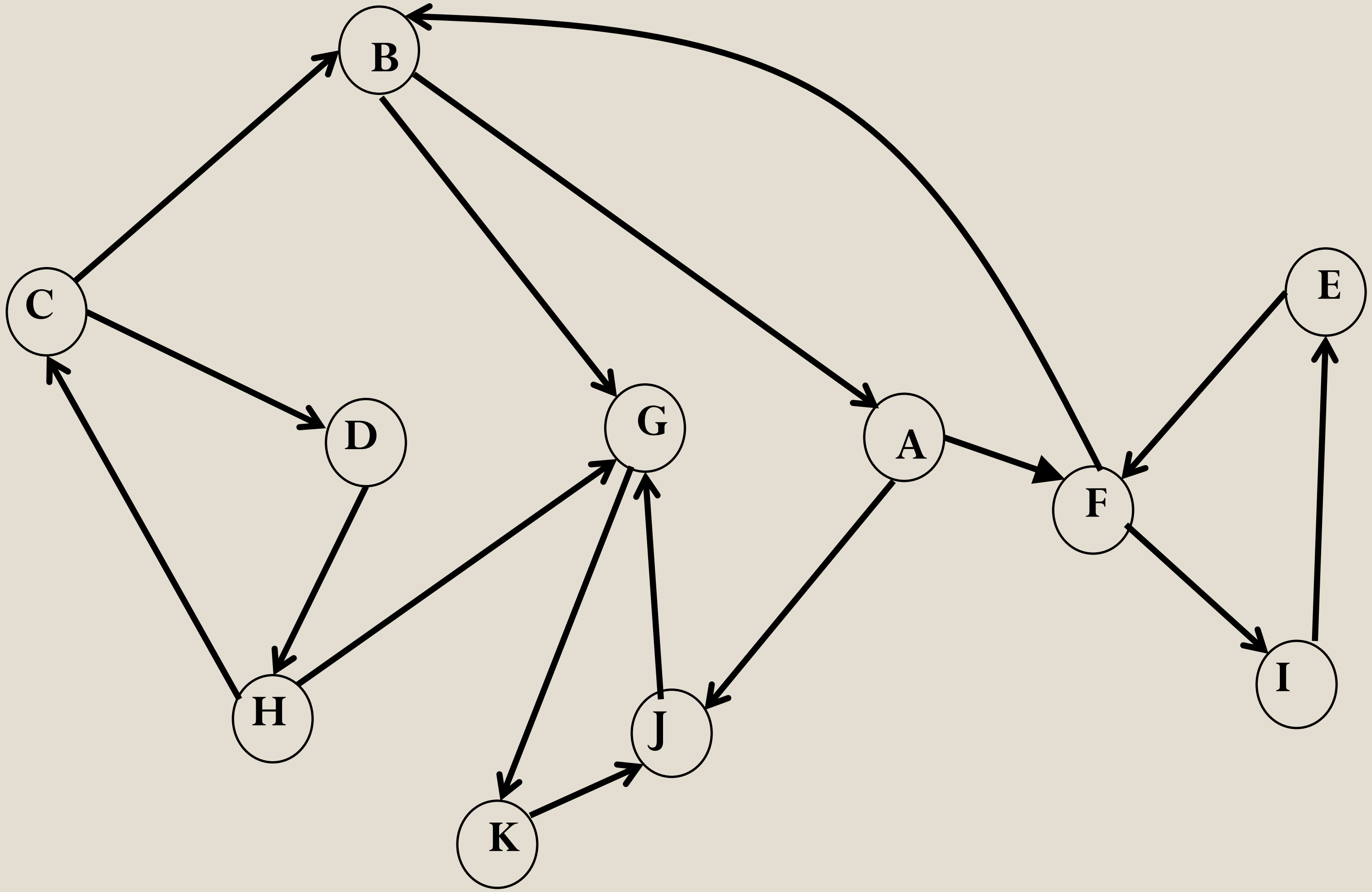
Separate Check for Root: a

Strongly Connected Components (22.5)

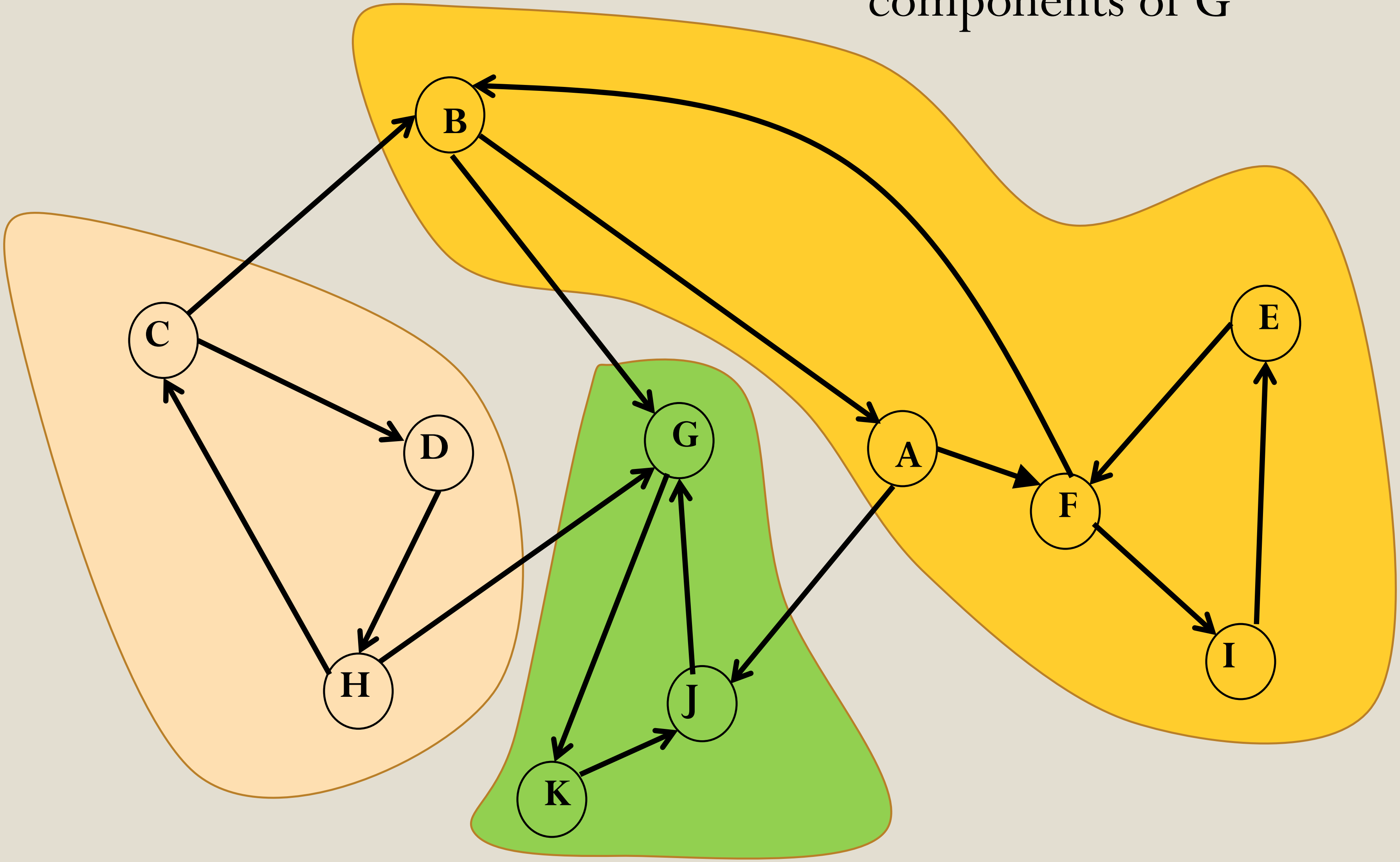
Let G be a directed graph.

- G is **strongly connected** if there exists a path between any pair of vertices.
- If G is not strongly connected, decompose G into *strongly* connected components:
  - sets of vertices in which any two vertices are *mutually reachable*
  - each vertex set cannot be enlarged by adding more vertices without destroying this property.

Directed Graph G



Strongly connected components of G



Determine the strongly connected components (stcc) in O(n+m) time

Perform 2 DFS's
On what graphs?

- G^T is the transpose of G generated by reversing the direction of every edge
- G^T and G have the same strongly connected components

Record discovery and finish times

Sketch of algorithm finding the stcc

1. call DFS on G to compute f[u] for each vertex u
    A. Sort nodes in decreasing order of **f[u]**
    B. (Only requires time O(n) since $1 \le f[u] \le 2n$)
2. find G^T, the transpose of G
3. call DFS on G^T
    - consider the vertices in order of **decreasing f[u]**
4. the second DFS generates one or more tree
    - the vertices in each tree form one strongly connected component

Why does the algorithm find the stcc?

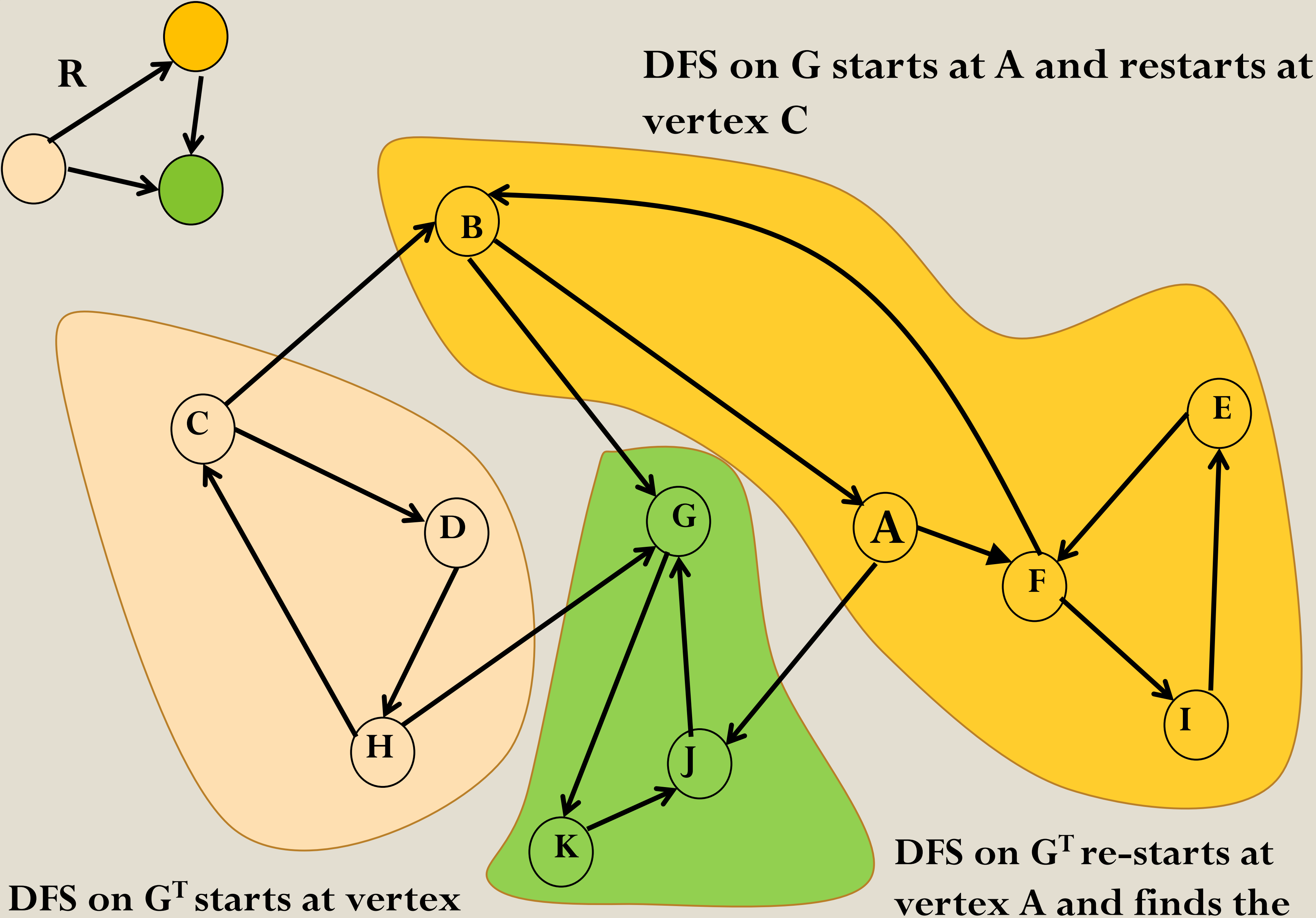
Not obvious.

Create the following "reduced" graph $R=(V_R,E_R)$

- Shrink every stcc into a single vertex.
- Put edges not in a stcc into graph R and remove duplicate edges.

Graph R is a dag

There must exist at least one "vertex" that has no incoming edges and at least one vertex with no outgoing edges.



DFS on G starts at A and restarts at vertex C

DFS on G^T starts at vertex C and finds the first stcc

DFS on G^T re-starts at vertex A and finds the second stcc

Main Idea - Summary

Second DFS on G^T

- we start with the component C whose f(C) is the biggest (actually we start with x in C where f(x) is the biggest).
- No edges go from inside C to any other component.
- The tree rooted at x contains exactly the vertices in C and we generated one strongly connected component.

Repeat the argument for the next sink in graph R until all strongly connected components have been generated.

Hence, the strongly connected components can be found in O(n+m) time by doing two DFS's.

Let U be a set of vertices of directed graph G

- d(U) is the smallest discovery time of any vertex in U
- f(U) is the largest finishing time of any vertex in U

Assume C and C′ are two strongly connected components of G.

Claim 1: If there is an edge (u, v) in G with u in C and v in C′, then f(C) > f(C′).
Claim 2: If there is an edge (v,u) in the *transpose* of G with v in C’ and u in C, then f(C’) < f(C).