Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 7: this Friday @ 11:59 pm
  - Lab 6: next Monday 10/31 @ 11:59 pm
  - **Assignment 2: Friday 11/4 @ 11:59 pm**
  - Lab 7: Monday 11/7 @ 11:59 pm
- Live Support Session for Assignment 2
  - This Friday 7-8 pm (https://pitt.zoom.us/my/khattab)
- Weekly Live QA Session on Piazza
  - Friday 4:30-5:30 pm

Previous lecture

- ADT Graph
  - definitions
  - representations
    - two-arrays
    - adjacency matrix
    - adjacency lists
  - traversals
    - BFS
      - shortest paths based on number of edges
      - connected components

This Lecture

- ADT Graph
  - traversals
    - DFS
      - finding articulation points of a graph
  - representation
    - Graph compression

Problem of previous lecture

- **Input**: A file containing LinkedIn (LI) accounts and their connections
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - e.g., 1^{st} connection?, 2^{nd} connection?, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - For each connected component, are there certain accounts that if removed, the remaining accounts become ***partitioned***?

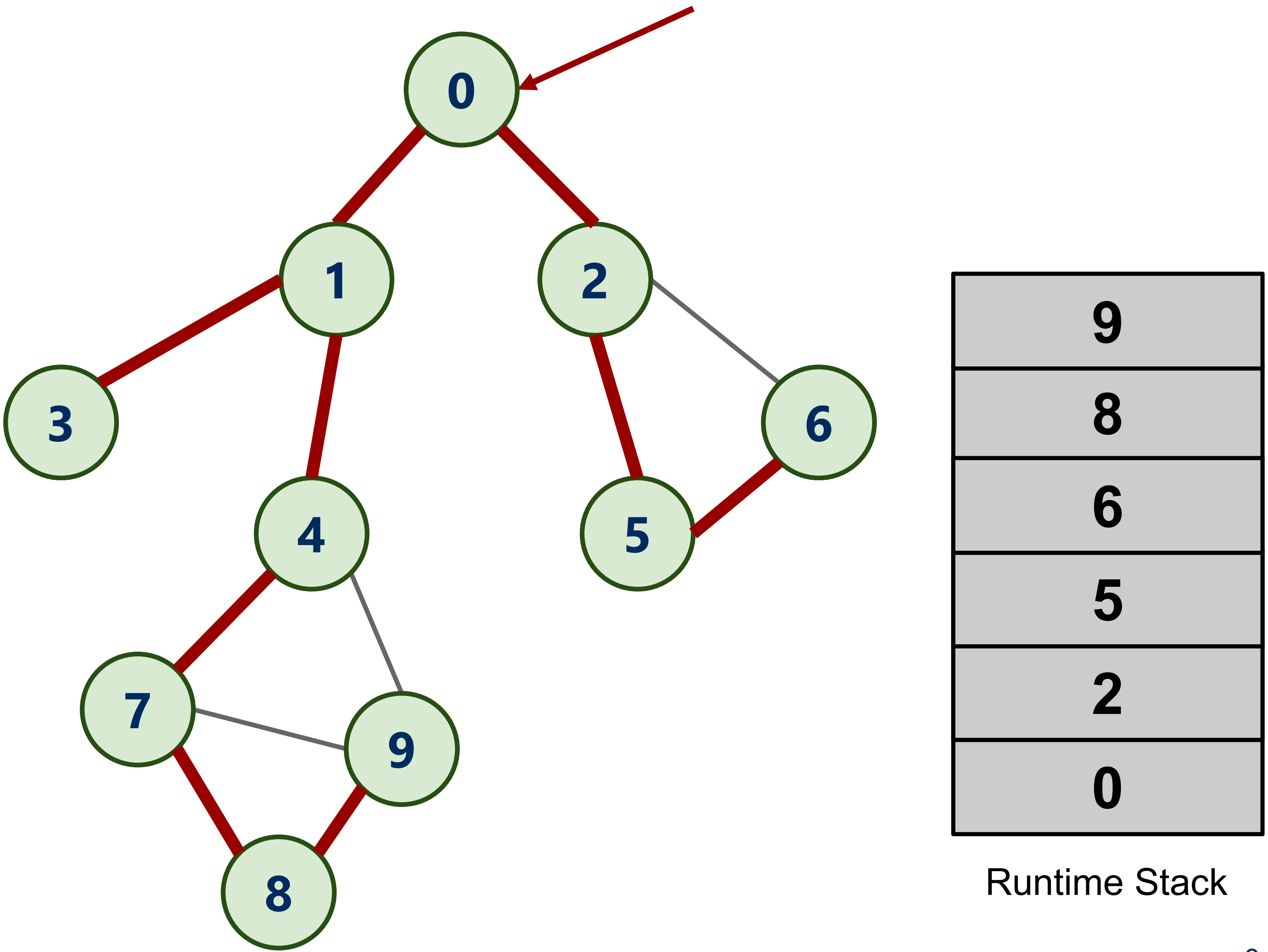
DFS – Depth First Search

- Already seen and used this throughout the term
  - For Huffman encoding…
    - as we build the codebook out of the Huffman Trie
- Can be easily implemented recursively
  - For each vertex, visit *first* unseen neighbor
  - Backtrack at deadends (i.e., vertices with no unseen neighbors)
    - Try *next* unseen neighbor after backtracking
  - An arbitrary order of neighbors is assumed

DFS Pseudo-code

```
DFS(vertex v) {
  seen[v] = true //mark v as seen
  for each unseen neighbor w
    parent[w] = v
    DFS(w)
}
```

DFS example



When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
   visit v //pre-order DFS
   for each unseen neighbor w
      parent[w] = v
      DFS(w)
}
```

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
for each unseen neighbor w
      parent[w] = v
      DFS(w)
 visit v //post-order DFS
}
```

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
for each unseen neighbor w
      parent[w] = v
      DFS(w)
      (re)visit v //in-order DFS
}
```

Runtime Analysis of BFS

- Each vertex is added to the queue exactly once and removed exactly once
  - *v* add/remove operations
    - *O(v)* time for vertex processing
- Edges are checked when adding the list of neighbors to the queue
- Each edge is checked at most twice, one per edge endpoint
  - *O(e)* time for edge processing
- Total time: vertex processing time + edge processing time
  - *O(v + e)*

Runtime Analysis for DFS

- For Adjacency Matrix representation, BFS checks each *possible* edge!
  - $O(v^2)$ time for edge processing with Adjacency Matrix
- Total time: $O(v^2 + v) = O(v^2)$

Runtime Analysis of DFS

- Each vertex is seen then visited exactly once
  - $O(v)$ time for vertex processing
  - except when (re)visiting a vertex after each child
    - vertex processing happens inside edge processing in that case
- Edges are checked when finding the list of neighbors
- Each edge is checked at most twice, one per edge endpoint
  - $O(e)$ time for edge processing
- Total time: vertex processing time + edge processing time
  - $O(v + e)$

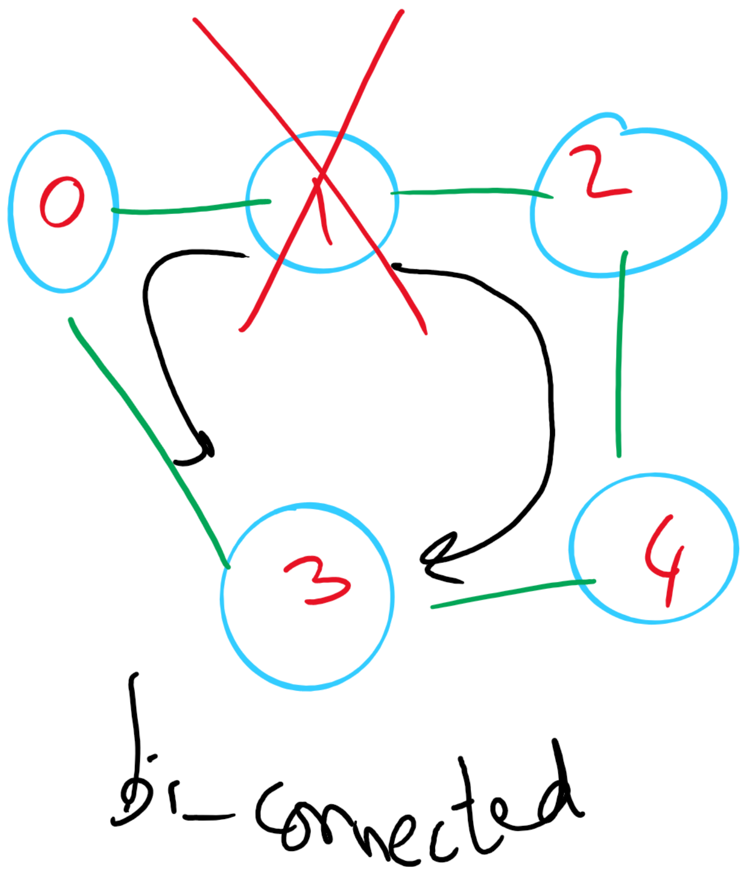
Runtime Analysis of BFS and DFS

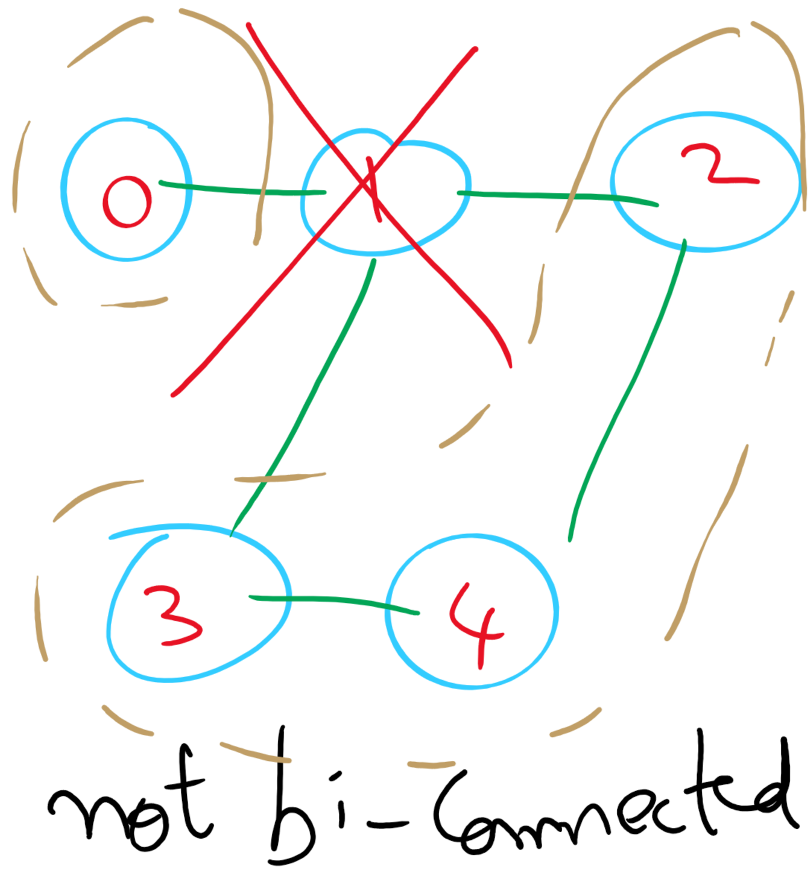
- At a high level, DFS and BFS have the same runtime
  - Each vertex must be seen and then visited, but the order will differ between these two approaches
- The representation of the graph affect the runtimes of of these traversal algorithms?
  - $O(v + e)$ with Adjacency Lists
  - $O(v^2)$ with Adjacency Matrix
  - Note that for a dense graph, $v + e = O(v^2)$

Biconnected graphs

- A *biconnected graph* has at least 2 distinct paths between all vertex pairs
  - a distinct path shares no common edges or vertices with another path except for the start and end vertices
- A graph is biconnected graph iff it has zero *articulation points*
  - Vertices, that, if removed, will separate the graph

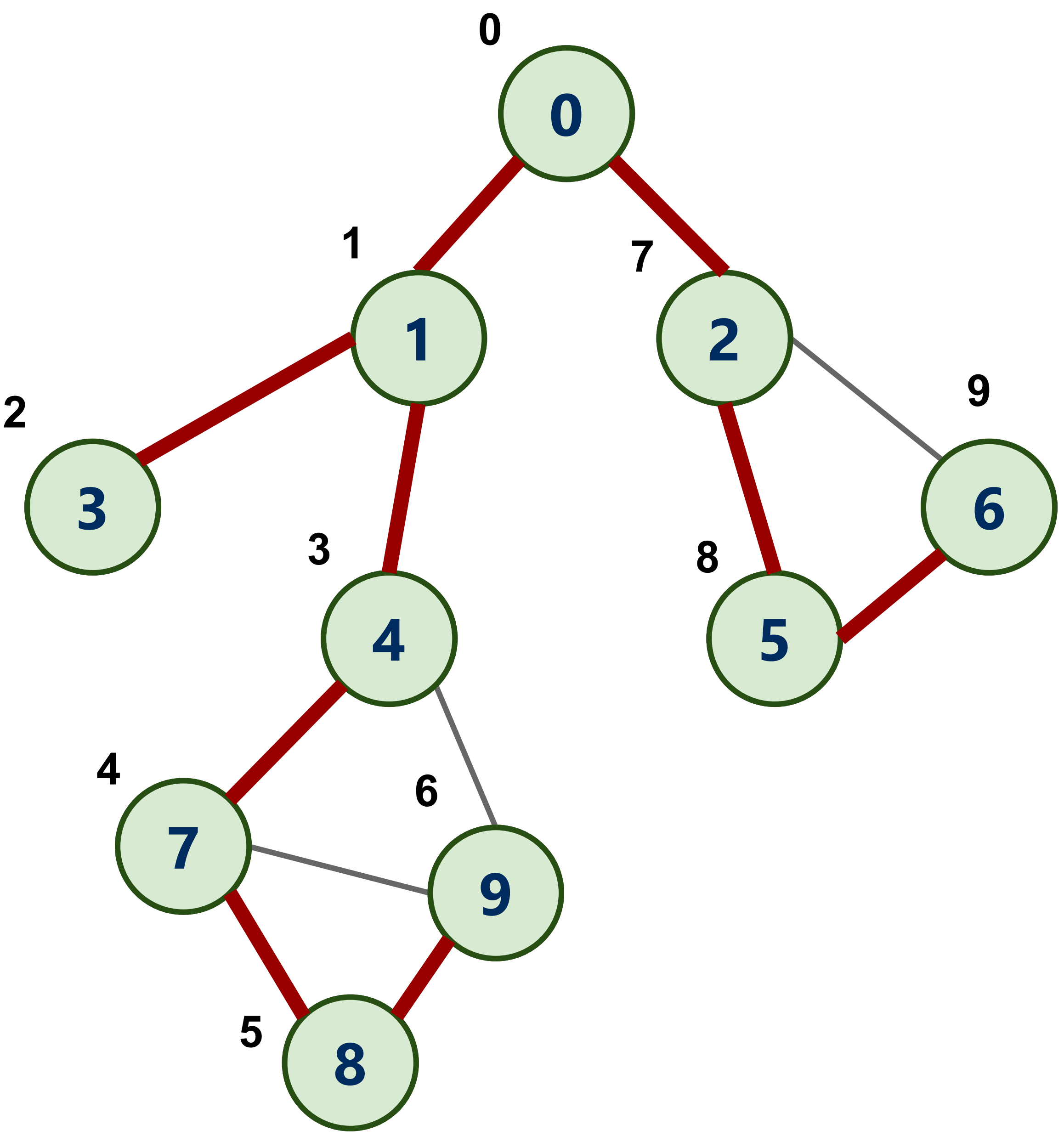
Biconnected Graph





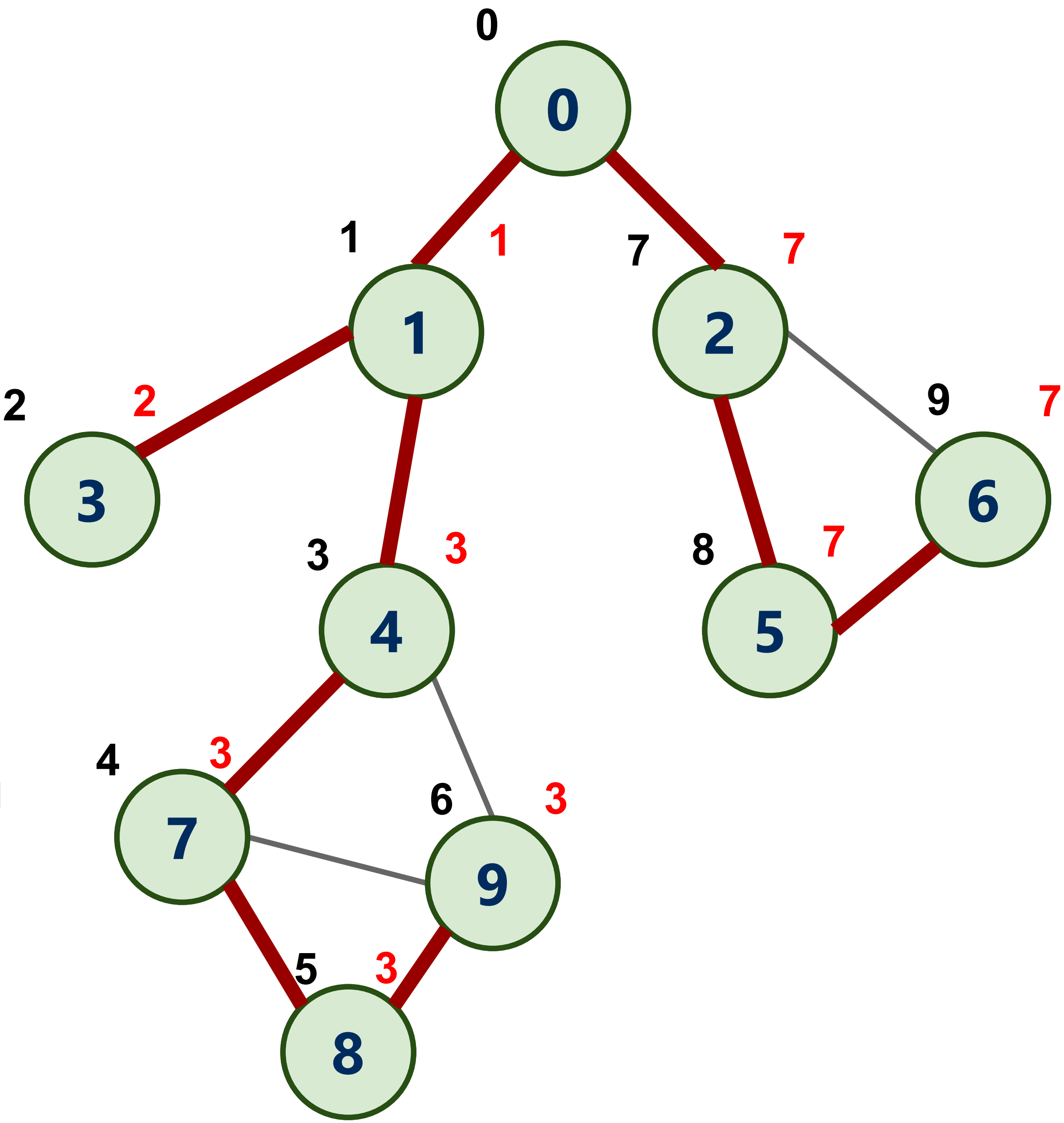
Finding articulation points of a graph

- Edges not included in the spanning tree are called **back edges**
  - e.g., (4, 9) and (2, 6)
- A pre-order DFS traversal visits the vertices in some order
  - let's number the vertices with their traversal order
  - num(v)



Finding articulation points of a graph

- For each non-root vertex v, find the lowest numbered vertex reachable from v
  - not through v's parent
  - using 0 or more tree edges then at most one back edge
- move down the tree looking for a back edge that goes backwards the furtheset

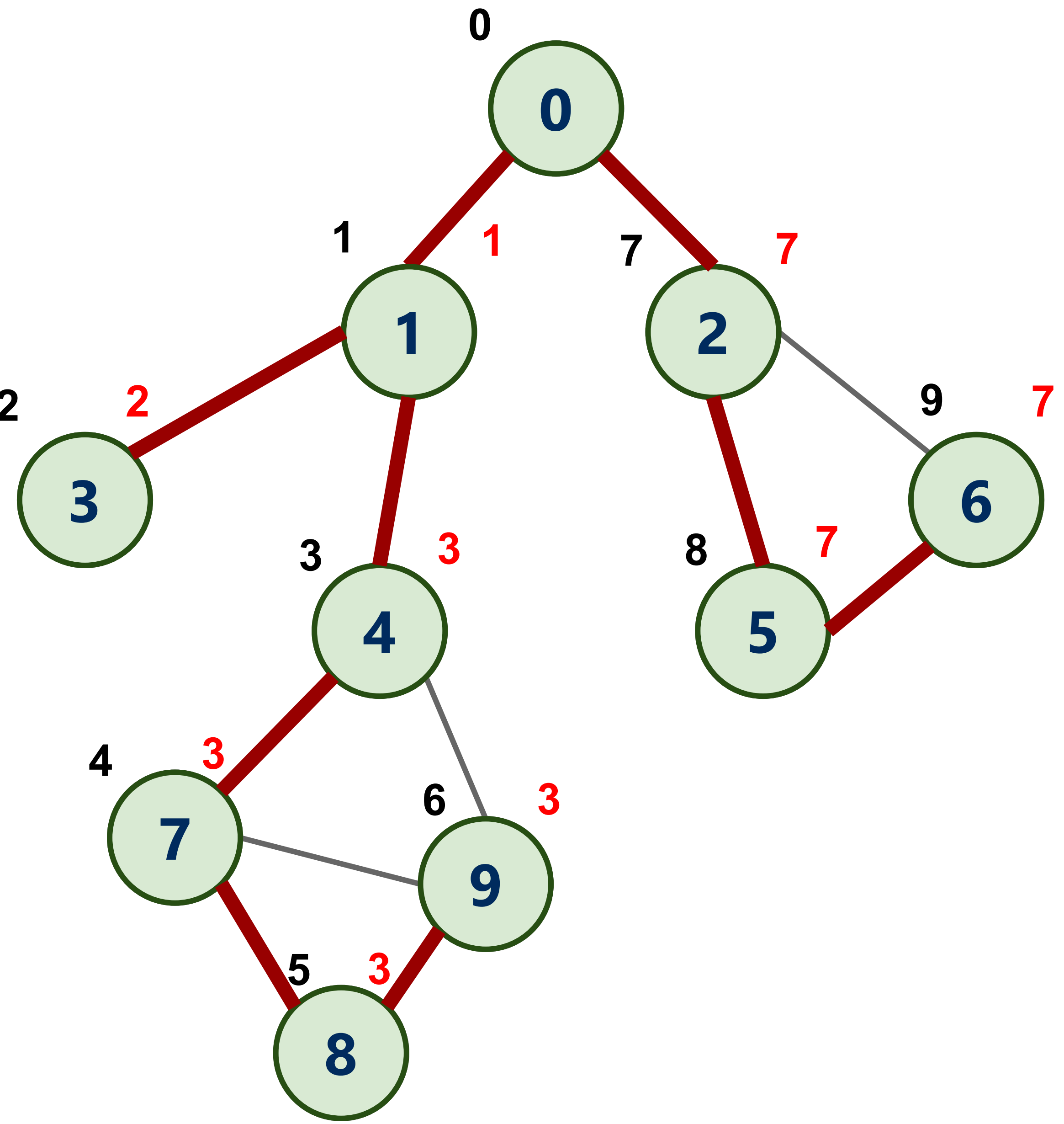


low(v)

- low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then **at most one** back edge
  - Min of:
    - num(v) (the vertex is reachable from itself)
    - Lowest num(w) of all back edges (v, w)
    - Lowest low(w) of all children of *v* (the lowest-numbered vertex reachable through a child)

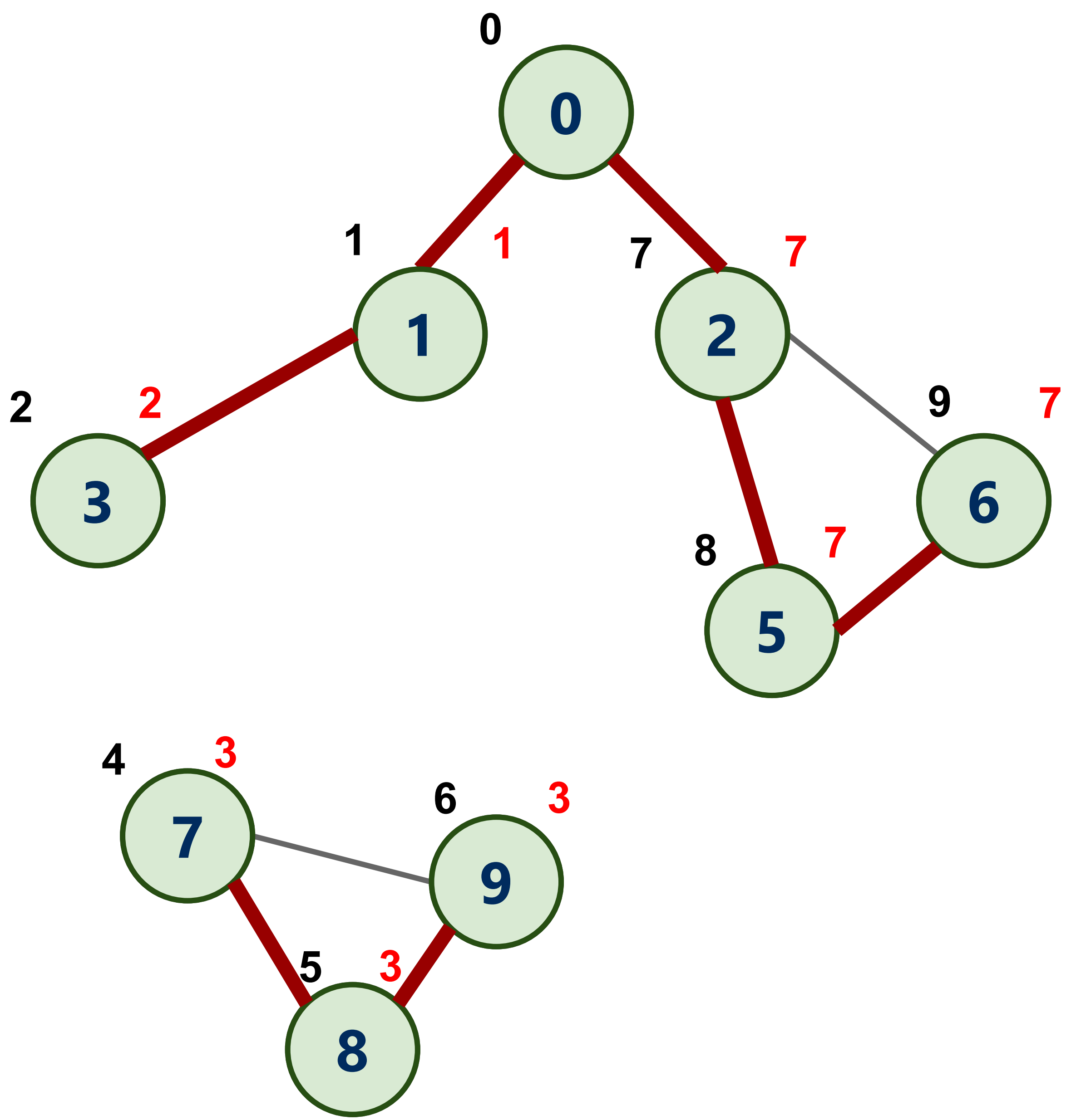
Finding articulation points of a graph

- What does it mean if a vertex v has a child w such that
  - low(w) >= num(v)?
- e.g., 4 and 7
- It means the child has no other way except through its parent to reach at least one other vertex
- e.g., 7 cannot reach 0, 1, and 3 except through 4
- So, the parent is an articulation point!



Finding articulation points of a graph

- So, the parent is an articulation point!
  - e.g., if 4 is removed, the graph becomes disconnected
- Each non-root vertex v that has a child w with low(w) >= num(v) is an articulation point



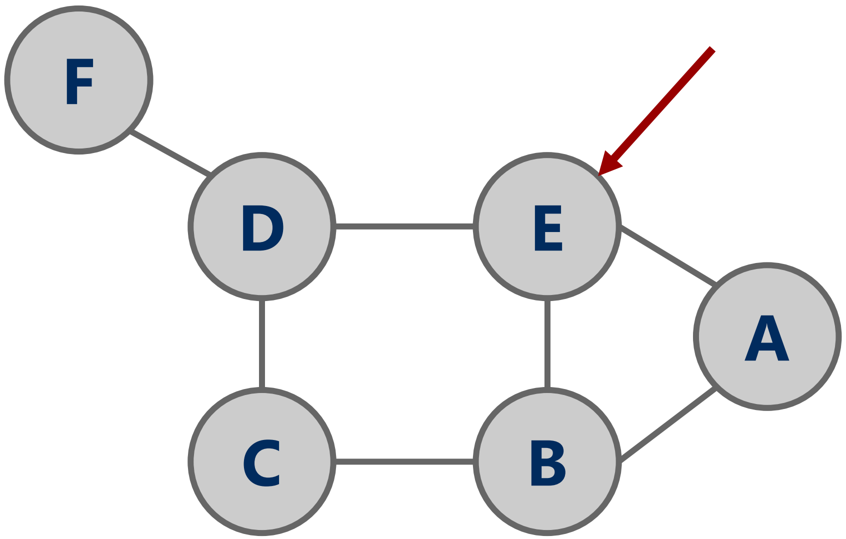
Finding articulation points of a graph: The Algorithm

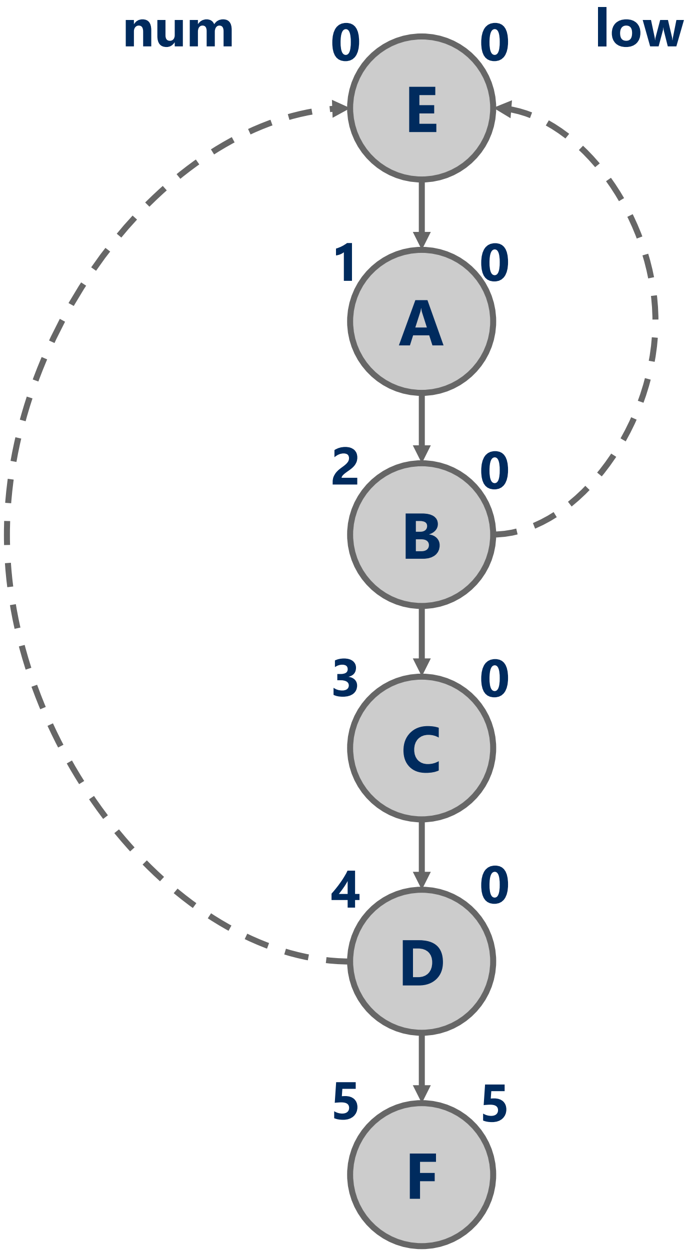
- Build a DFS spanning tree
  - Think of it as directed
  - Create back edges when considering a vertex that has already been visited
  - Label each vertex v with with two numbers:
    - num(v) = pre-order traversal order
    - low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then at most one back edge

when to compute num(v) and low(v)

- num(v) is computed as we move down the tree
    - pre-order DFS
- low(v) is computed as we move up the tree
- Recursive DFS is convenient to compute both
    - why?

Finding articulation points example





Using DFS to find the articulation points of a connected undirected graph

```
int num = 0
DFS(vertex v) {
    num[v] = num++
    low[v] = num[v] //initially
    seen[v] = true //mark v as seen
    for each neighbor w
      if(w unseen){
          parent[w] = v
          DFS(w) //after the call returns low[w] is computed, why?
          low[v] = min(low[v], low[w])
      } else { //seen neighbor
          if(w != parent[v]) //and not the parent, so back edge
            low[v] = min(low[v], num[w])
      }
}
```

What about the root of the spanning tree?

- What if we start DFS at an articulation point?
  - The starting vertex becomes the root of the spanning tree
  - If the root of the spanning tree has more than one child, the root is an articulation point