SCHOOL OF ENGINEERING

VANDERBILT UNIVERSITY

CS 3250
Algorithms
Graphs: Depth-First Search

Articulation Points
Tarjan's Algorithm

VANDERBILT UNIVERSITY

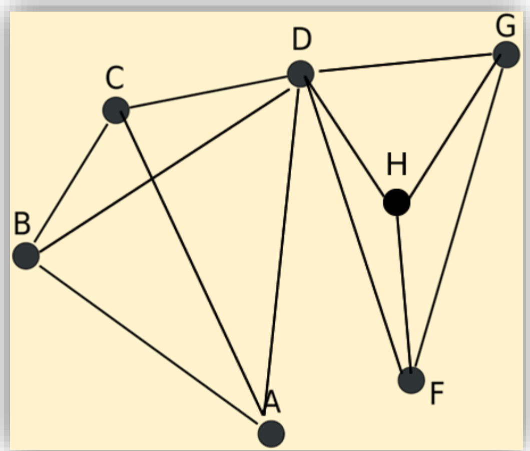
Announcements

- **HW2** is due Wednesday, February 7th by 9 AM.
- There are two parts:

1. **Brightspace Hashing Quiz**. Formative assessment. Graded but not timed.
2. **Gradescope Questions.**
    - For the first Gradescope question, you will need to reference the graph generator quiz on Brightspace to generate your random graph for Exercise #1.
    - You should be able to do Exercise #1 after Wednesday's lecture.

Recap: Articulation Points

- An **articulation point** in an **undirected** graph is a vertex whose deletion breaks the graph into separate pieces or components.
- A graph is said to be **biconnected** if it has no articulation points.
- **Connectivity** is critical to the design of any network -- road networks, social networks, computer networks, etc.

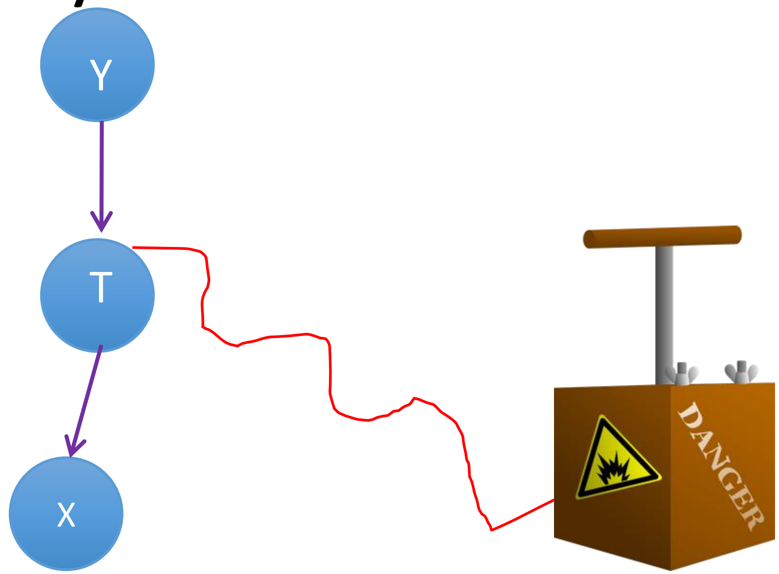


Recap: A Better Articulation Point Algorithm

- It turns out we can do better by making use of the information provided by DFS in the tree of discovery.
- In the tree of discovery for a graph G, all edges are shown as directed even when the graph is undirected.
- Think of the **back edges** in a tree as lifelines, or safety cables that link some vertex x safely back to one of its ancestors y.

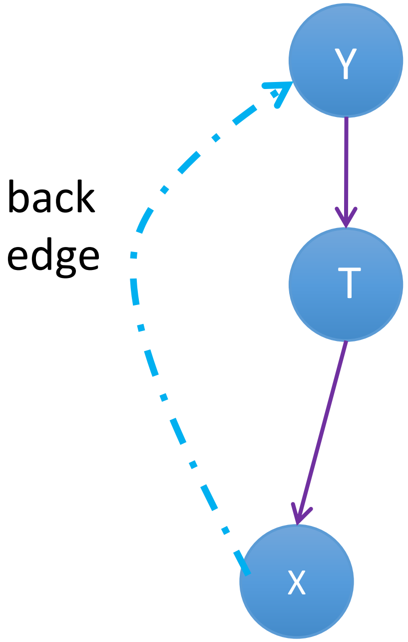
Recap: A Better Articulation Point Algorithm

- Given the DFS tree of discovery below, if we blow up (i.e., delete) vertex T, vertex x will have no way to get back to vertex Y.
- This means T is an **articulation point** whose deletion breaks the graph into two or more pieces, separating x from y.



Recap: A Better Articulation Point Algorithm

- Suppose the tree of discovery contains a back edge from x to y.
- If T is destroyed, x still has a way to safely reach vertex y.
- In this case T would **not** be an articulation point.



Graphs: Articulation Points

- **Question:** The root of the DFS tree of discovery is an articulation point when it has two or more children.

  A. Always

  B. Sometimes

  C. Never

Graphs: Articulation Points

- **Question:** The root of the DFS tree of discovery is an articulation point when it has two or more children.

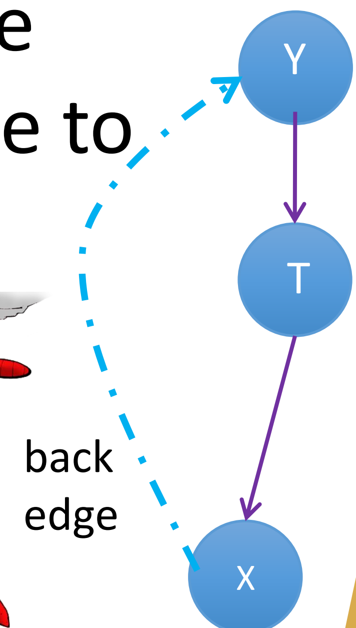
  A. Always

  B. Sometimes

  C. Never

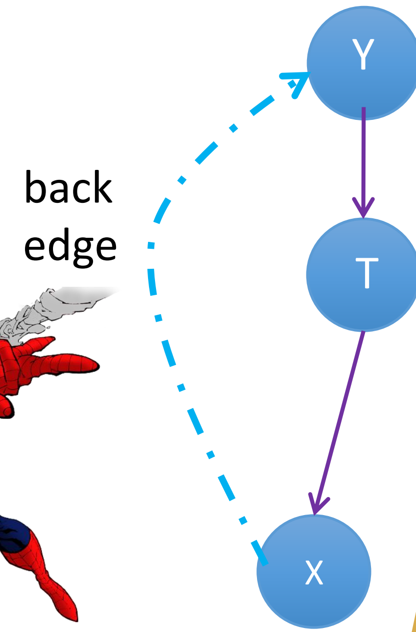
A Better Articulation Point Algorithm

- **Observations:** A vertex x in a DFS tree of discovery is an articulation point iff…
  1. x is the root of the DFS tree of Discovery and x has at least two children.
  2. x is not root of DFS tree of Discovery and has a child v where no vertex in the subtree rooted at v has a back edge to one of the ancestors of x.



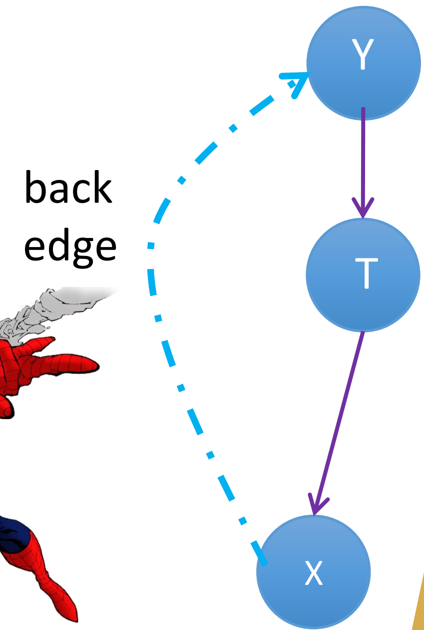
Question: Articulation Points

- **Question:** Why do we need to specify that the second observation does not apply to the root x of the DFS tree of Discovery?
- **Answer:** It is not possible in a subtree of x for one of the nodes to have a backedge to one of x's ancestors since x is the root.



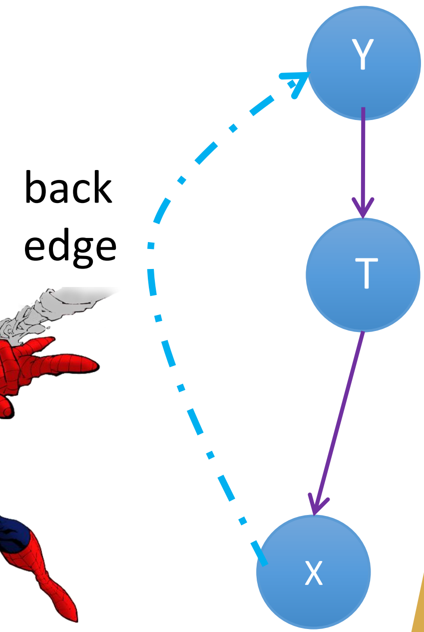
Question: Articulation Points

- **Question:** True/False. It’s not possible for a leaf in the tree of discovery to be an articulation point?
- True
- False



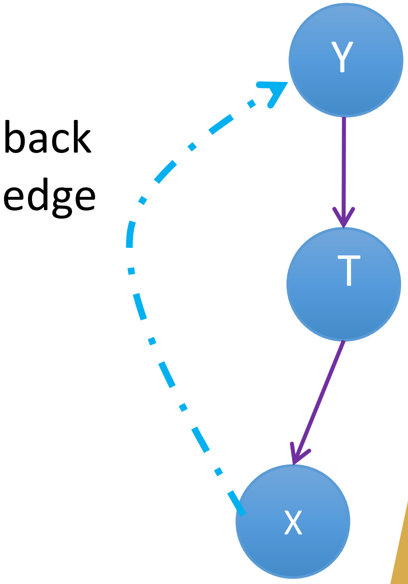
Question: Articulation Points

- **Question:** True/False. It's not possible for a leaf in the tree of discovery to be an articulation point?
- **Answer: True.** By definition, a leaf has no children so eliminating it cannot break the graph into pieces that separate the descendants of the leaf from the ancestors of the leaf.



A Better Articulation Point Algorithm

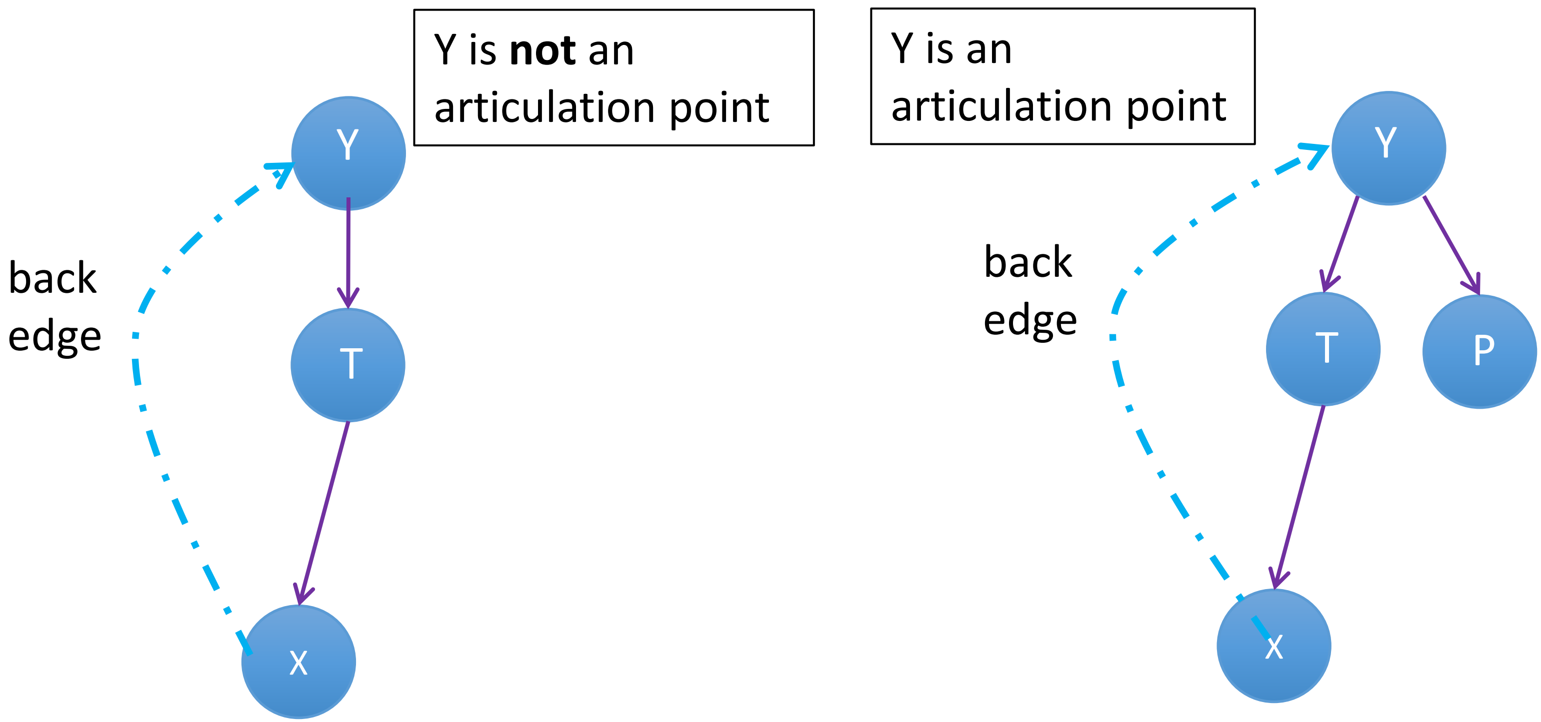
- The key to writing a smart algorithm to locate articulation points is understanding how reachability affects whether any vertex x is an articulation point.
- Provided we maintain the parent of each discovered node and the entry time from DFS, we have enough information to determine "reachability."
- Enter **Robert** "Spiderman" **Tarjan**.



VANDERBILT UNIVERSITY

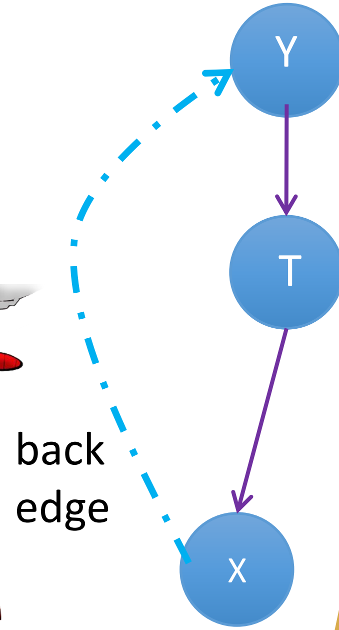
Tarjan's Articulation Point Algorithm

- **The root in the Tree of Discovery (easy) -** For the root x, if parent[x] is null) and x has two or more children in the tree of discovery, x is an articulation point, and the graph is **not bi-connected**.



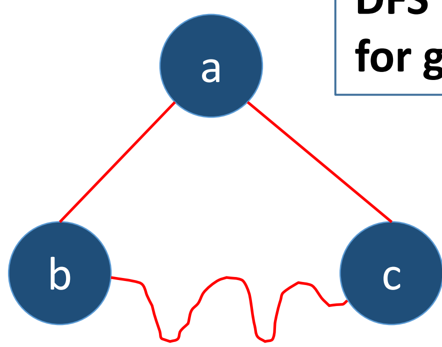
Tarjan’s Articulation Point Algorithm

- **Suppose x is not the root of the DFS tree of Discovery.**
- If x has a child v where no vertex in the subtree rooted at v has a back edge to one of the ancestors of x, then x is an articulation point.



Tarjan's Articulation Point Algorithm

- **Question:** What if we have a graph where the children of the root have edge/paths between them further down the in the tree of discovery? How can we claim the root of the DFS tree is an articulation point?

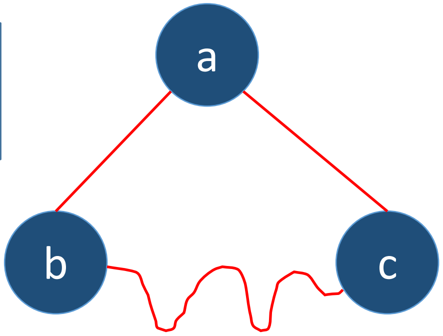


DFS tree of discovery for graph G?

Tarjan’s Articulation Point Algorithm

- What if we have a graph where the children of the root have edge/paths between them further down the in the tree of discovery? How can we claim the root of the DFS tree is an articulation point?
- **Answer:** A DFS would have explored those nodes first. In the graph below, the DFS would have a tree edge from B to C, meaning A would not have two children.

DFS tree of discovery for graph G?

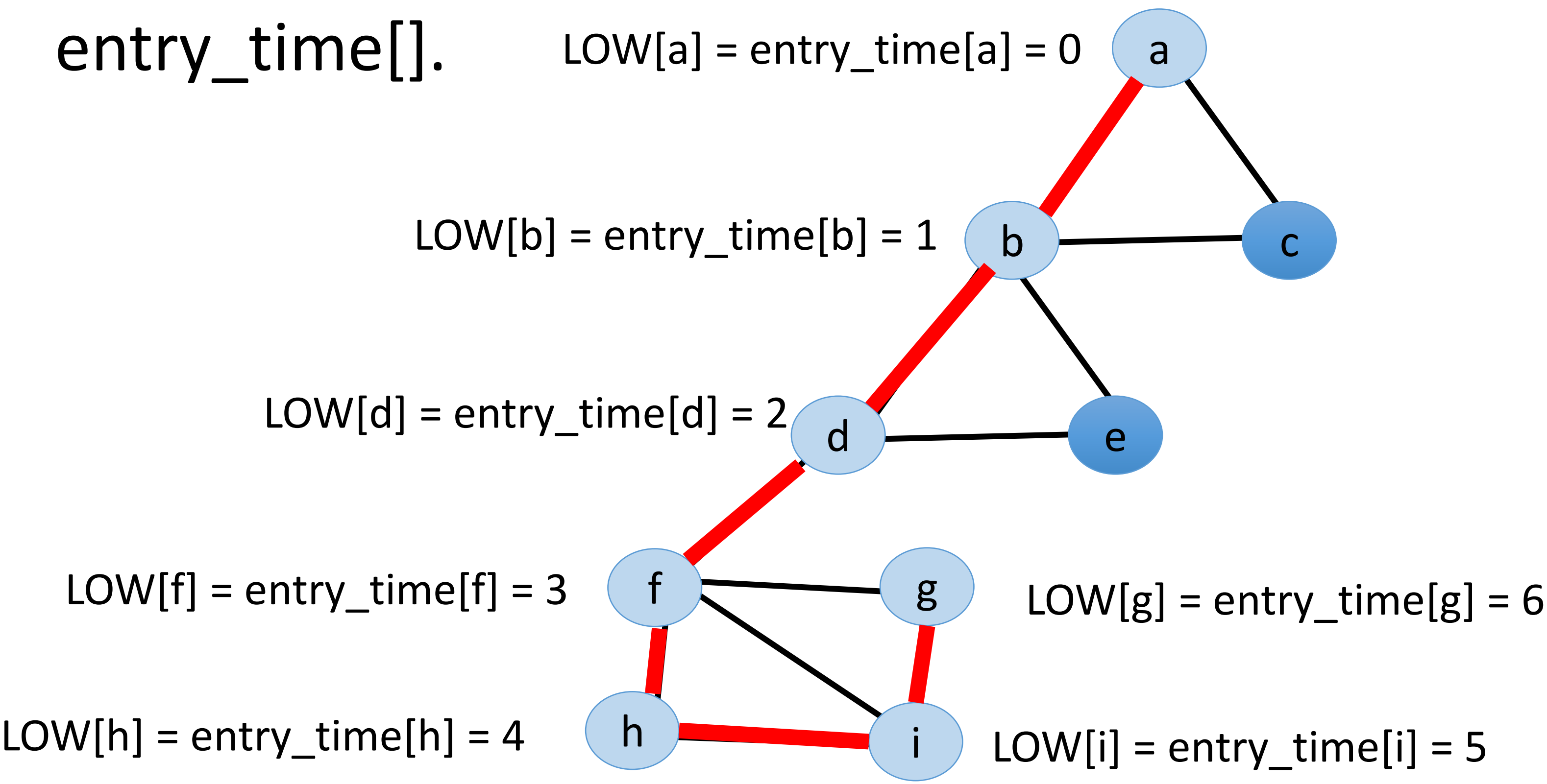


Tarjan's Articulation Point Algorithm

1. Time stamp each vertex during DFS with its entry_time[] (we don't use exit times here).
2. For every node x, determine the **earliest discovered vertex** that can be reached from the subtree rooted at x. This can be accomplished with housekeeping
   - Maintain an additional array LOW[]. During DFS discovery, initialize the LOW[x] to the entry_time[x].
   - During DFS, depending on the status of the neighbor of x being visiting set low of x to either:
     - LOW[x] = min(LOW[x], LOW[neigh])
     - LOW[x] = min(LOW[x], entry_time[neigh])

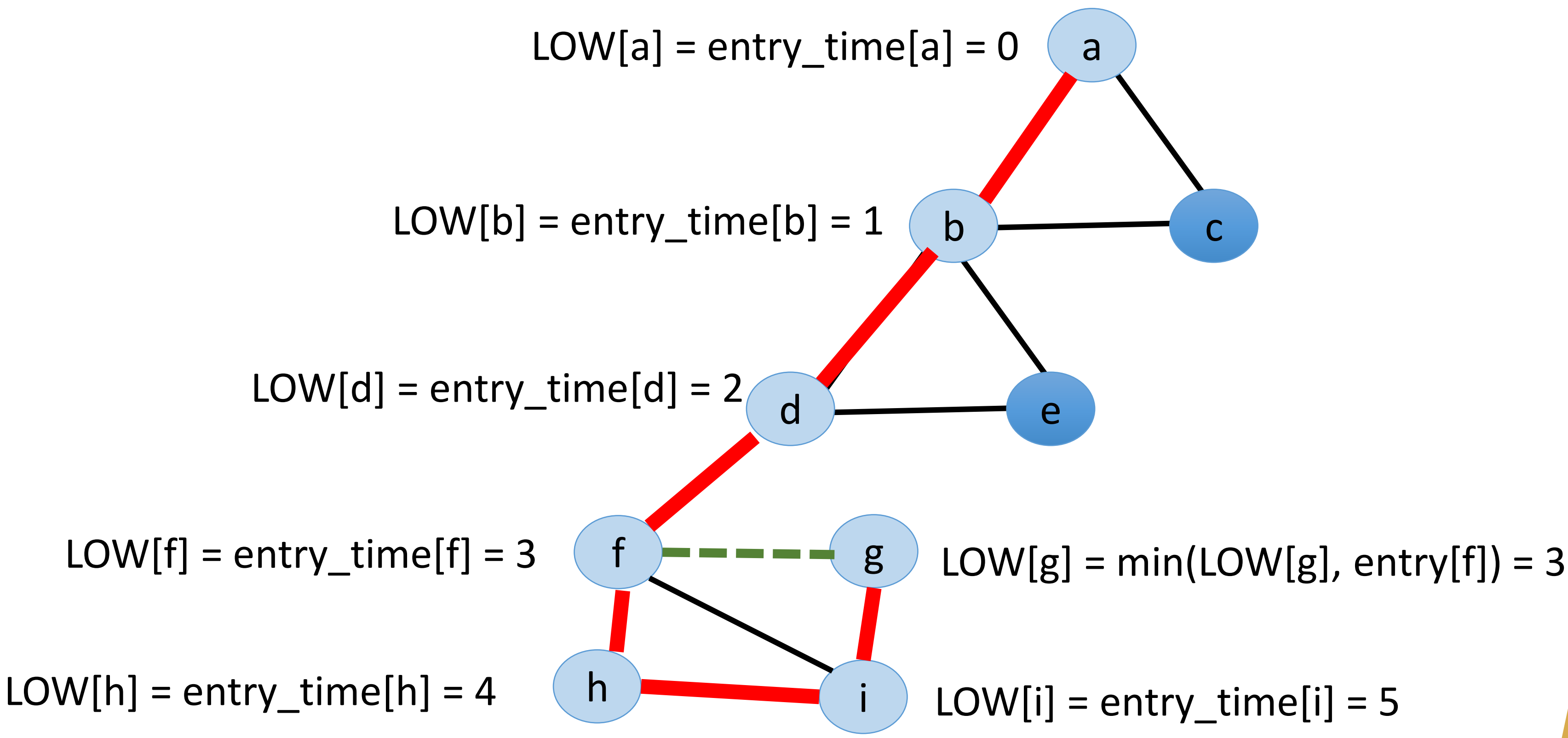
Tarjan's Articulation Point Algorithm

- **Walkthrough Step 1:** During DFS, let's imagine we discover the vertices in this order: a, b, d, f, h, i, g (I know, it's not lexicographic). As we **discover** each of those vertices, we initialize the LOW of each newly discovered vertex to its entry_time[].



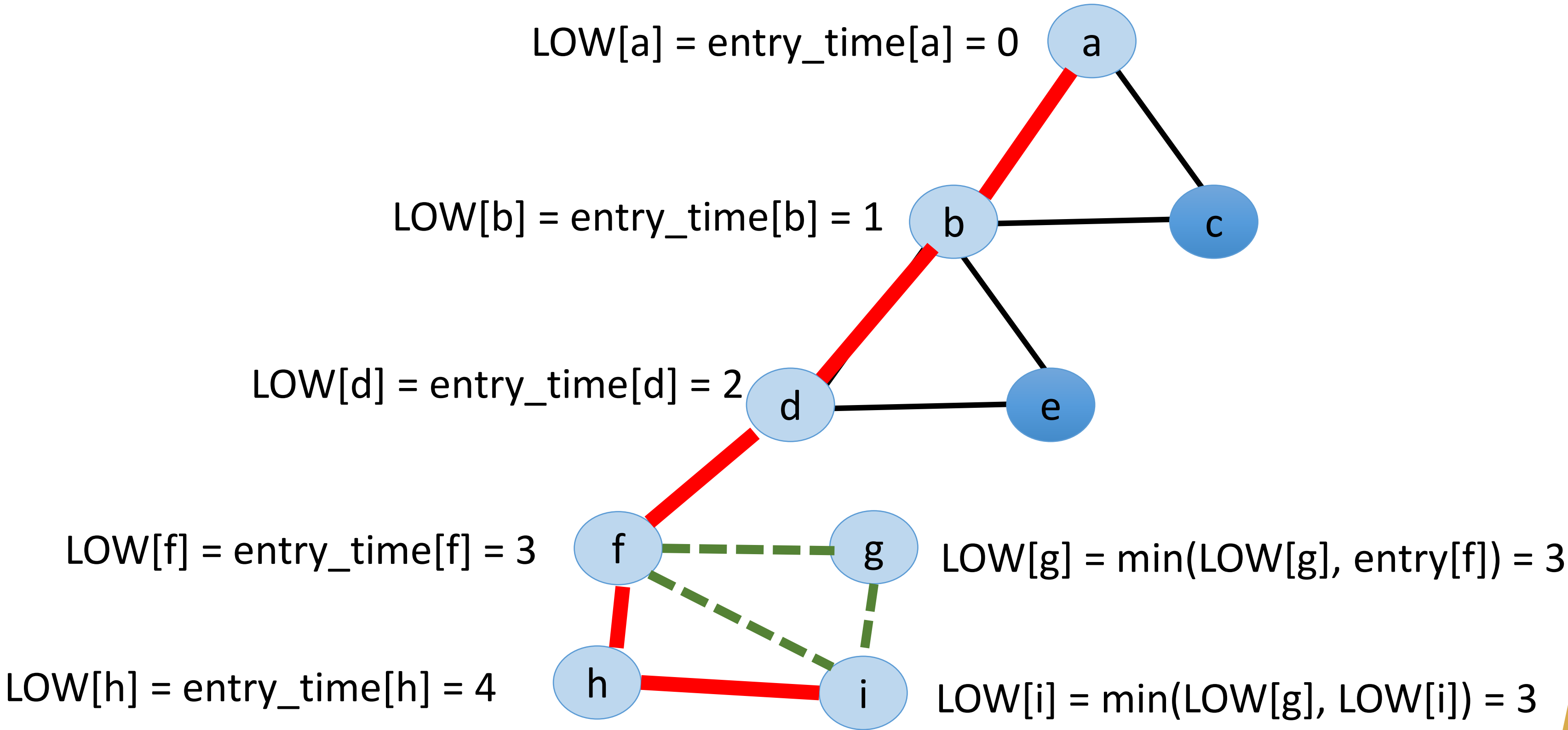
Tarjan's Articulation Point Algorithm

Walkthrough Step 2: We see an already discovered vertex f. We re-examine the low[g] and adjust its low (f's low/entry time indicates its better for g's low as we now know g can get to f).



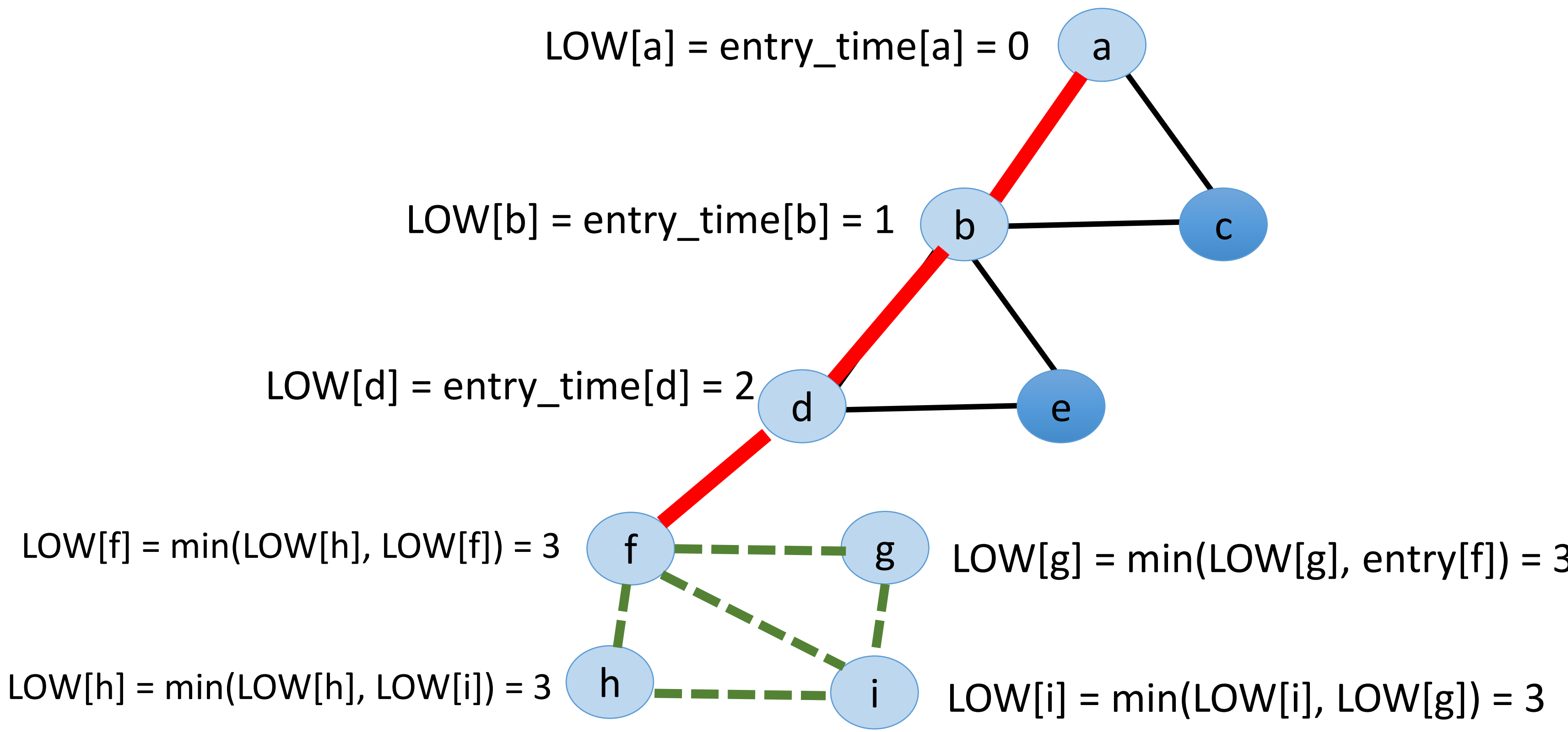
Tarjan's Articulation Point Algorithm

Walkthrough Step 3: We backtrack to i and re-examine the low[i] and adjust it since g's low is better for us (and we know vertex i can get to g). Vertex i will then head to f which is already discovered. That's not any better for vertex i.



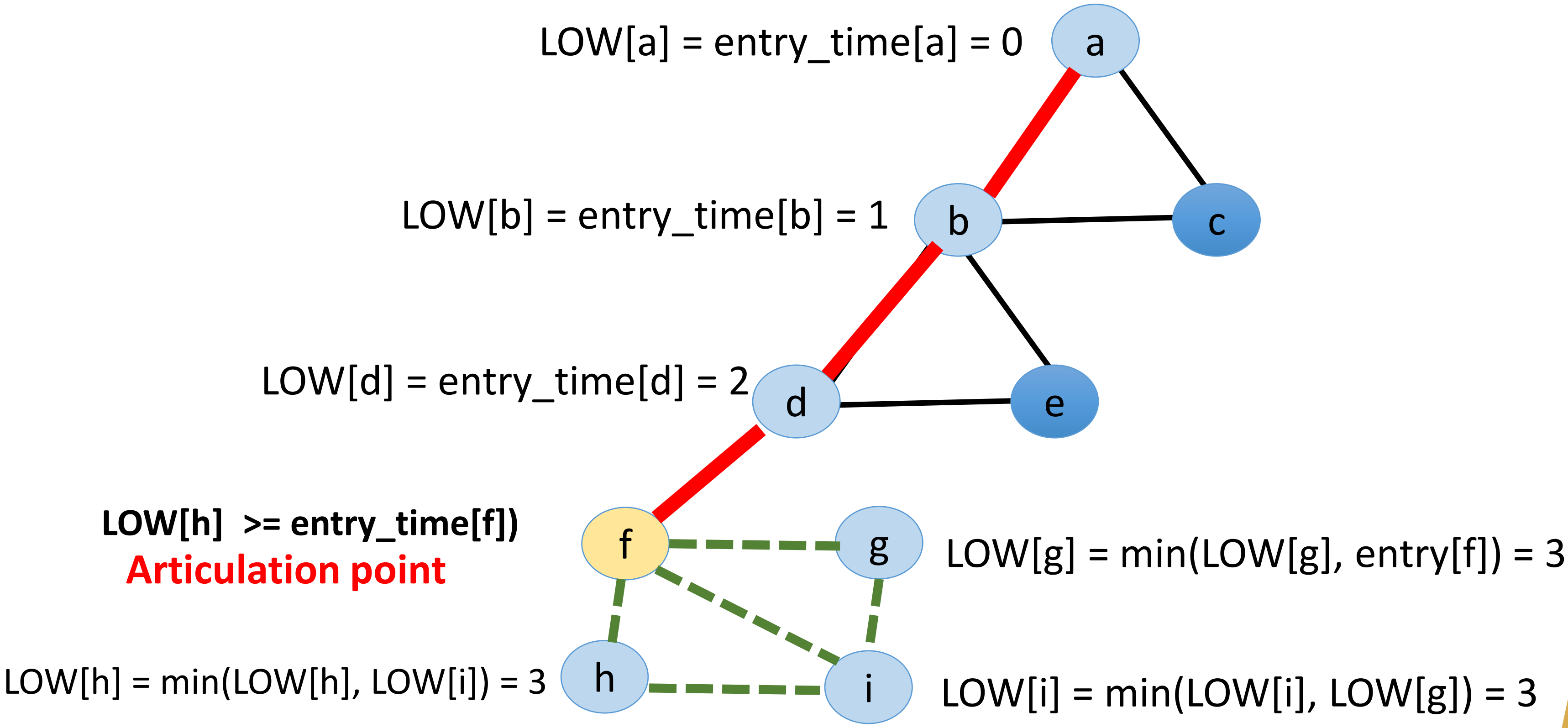
Tarjan's Articulation Point Algorithm

Walkthrough Step 4: We backtrack to h and re-examine the low[h] and adjust it if i's low if it is better for us (which it is). Upon backtracking to f, no further adjustment is needed. The low of f remains 3. Hmmm….



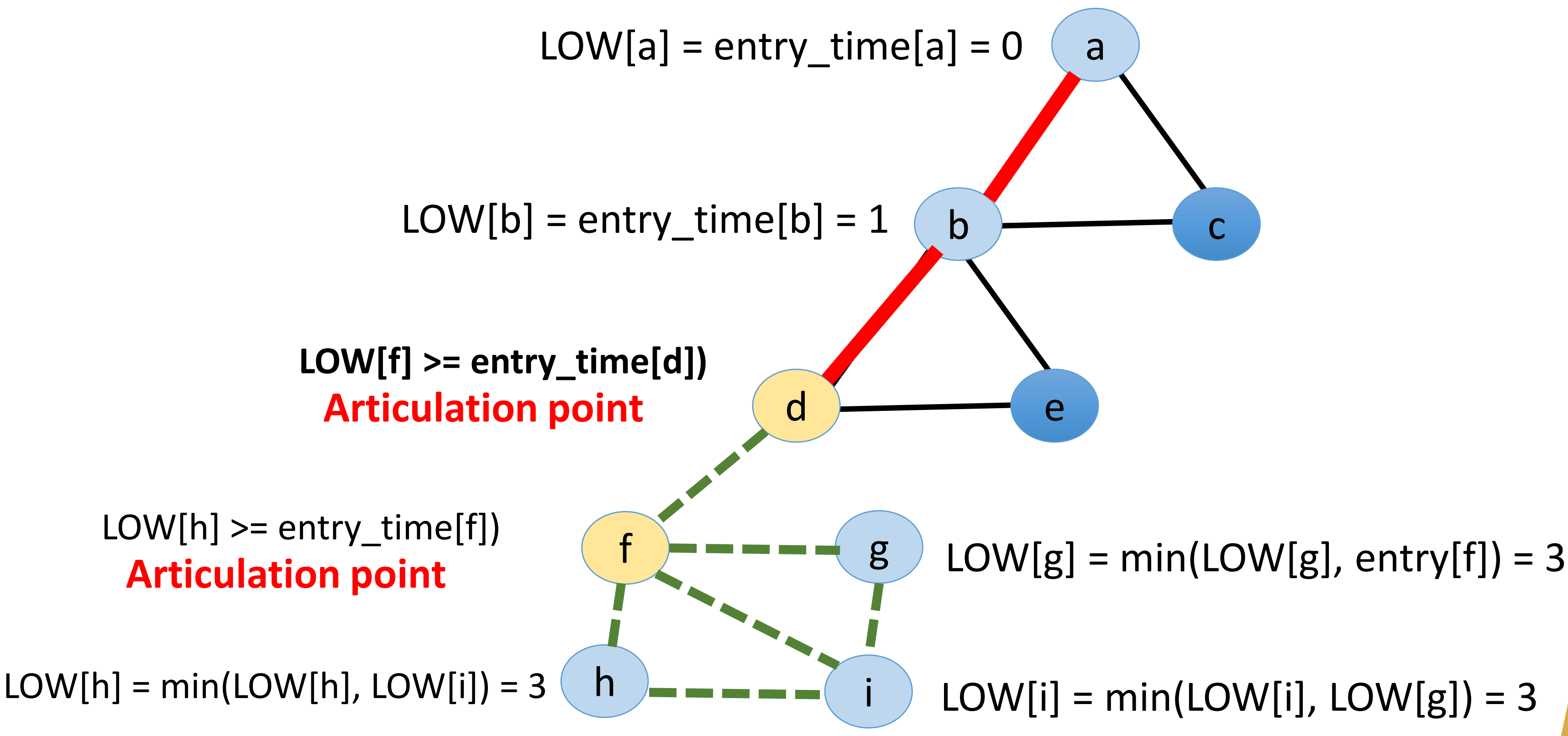
Tarjan’s Articulation Point Algorithm

Walkthrough Step 5: Since no descendant of f has an escape route to someplace better than f, we deem vertex f to be an **articulation point**.



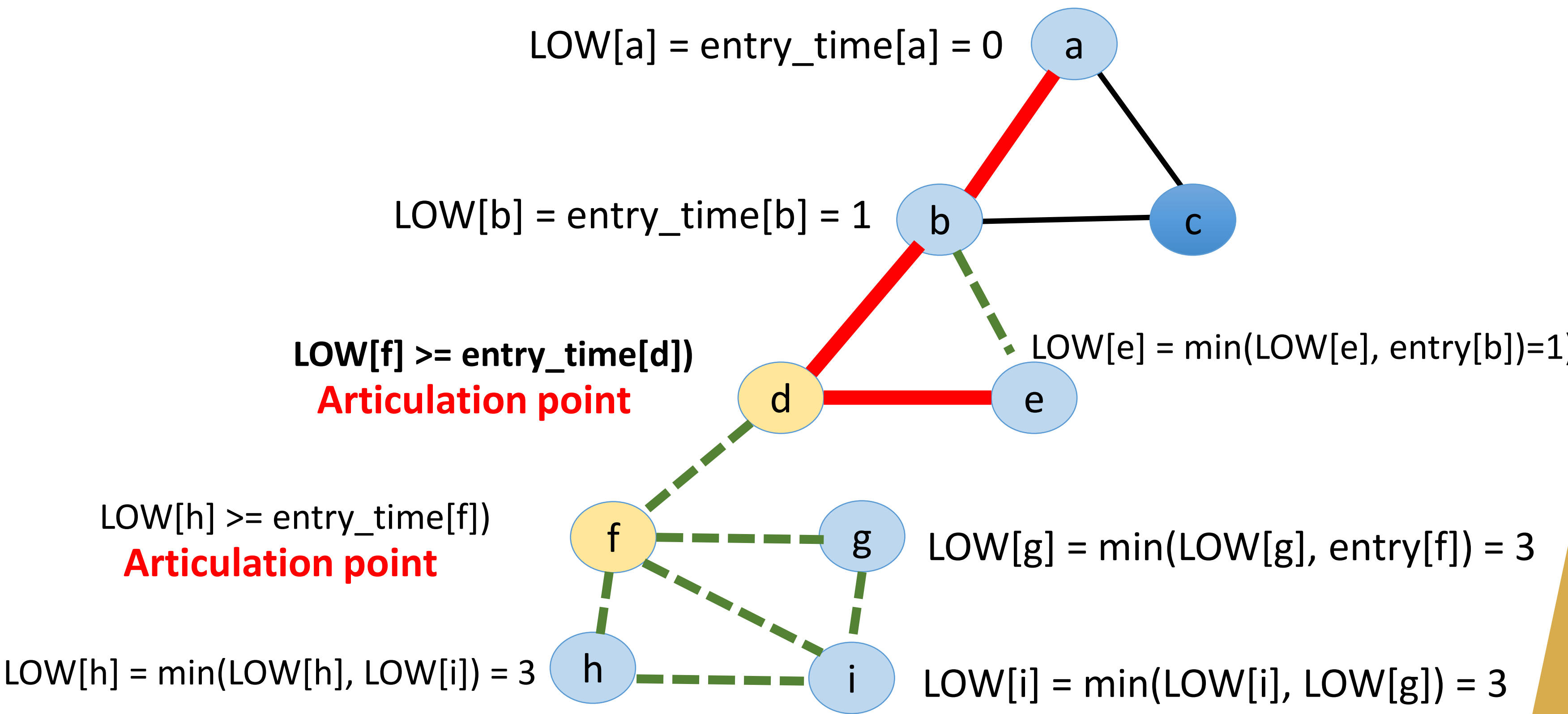
Tarjan's Articulation Point Algorithm

Walkthrough Step 6: Backtrack to d and we are in the same situation. No vertex in a subtree of d can escape to anyone better than d. We can deem vertex d to be an **articulation point**.



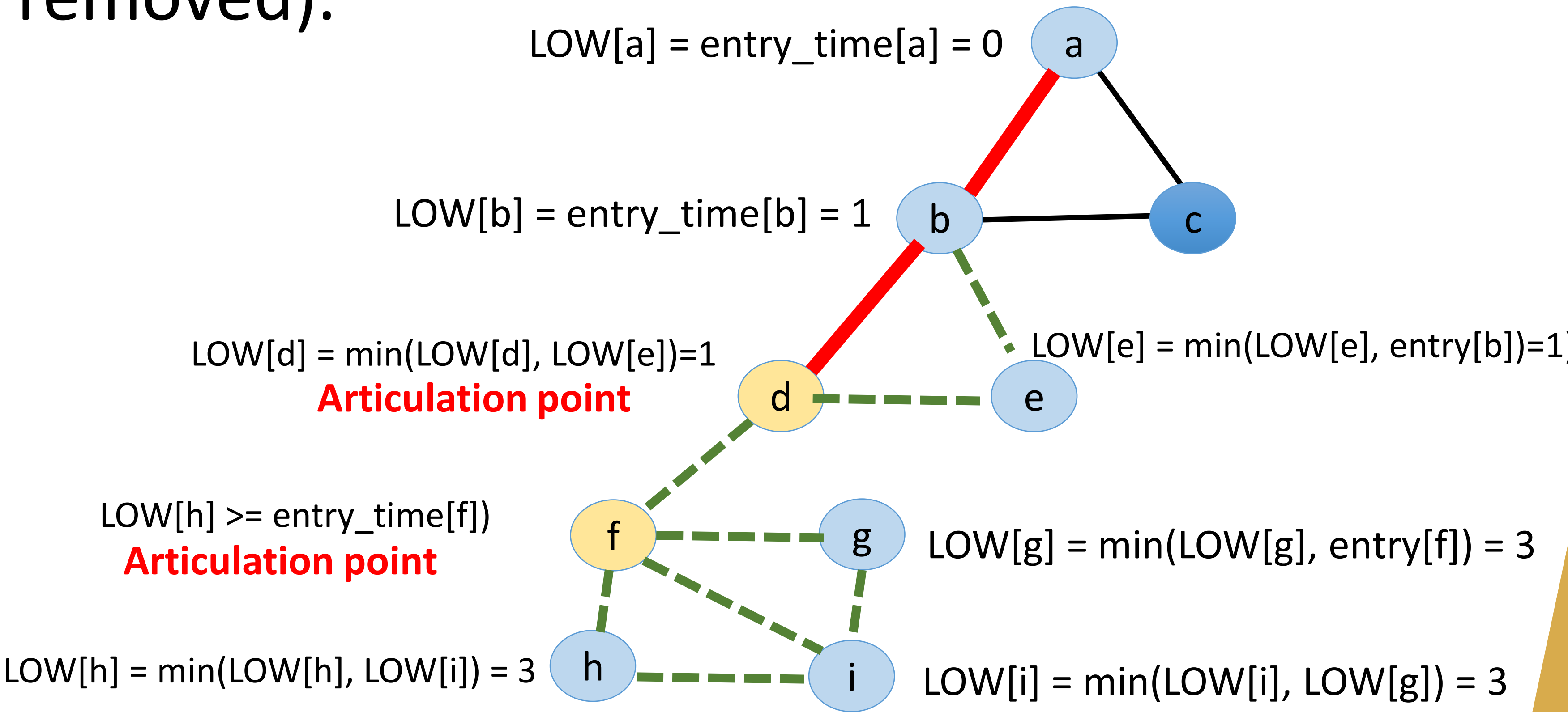
Tarjan's Articulation Point Algorithm

Walkthrough Step 7: Deep dive from d and discover e. From e, we hit the already discovered b, so we backtrack and adjust the LOW[e].



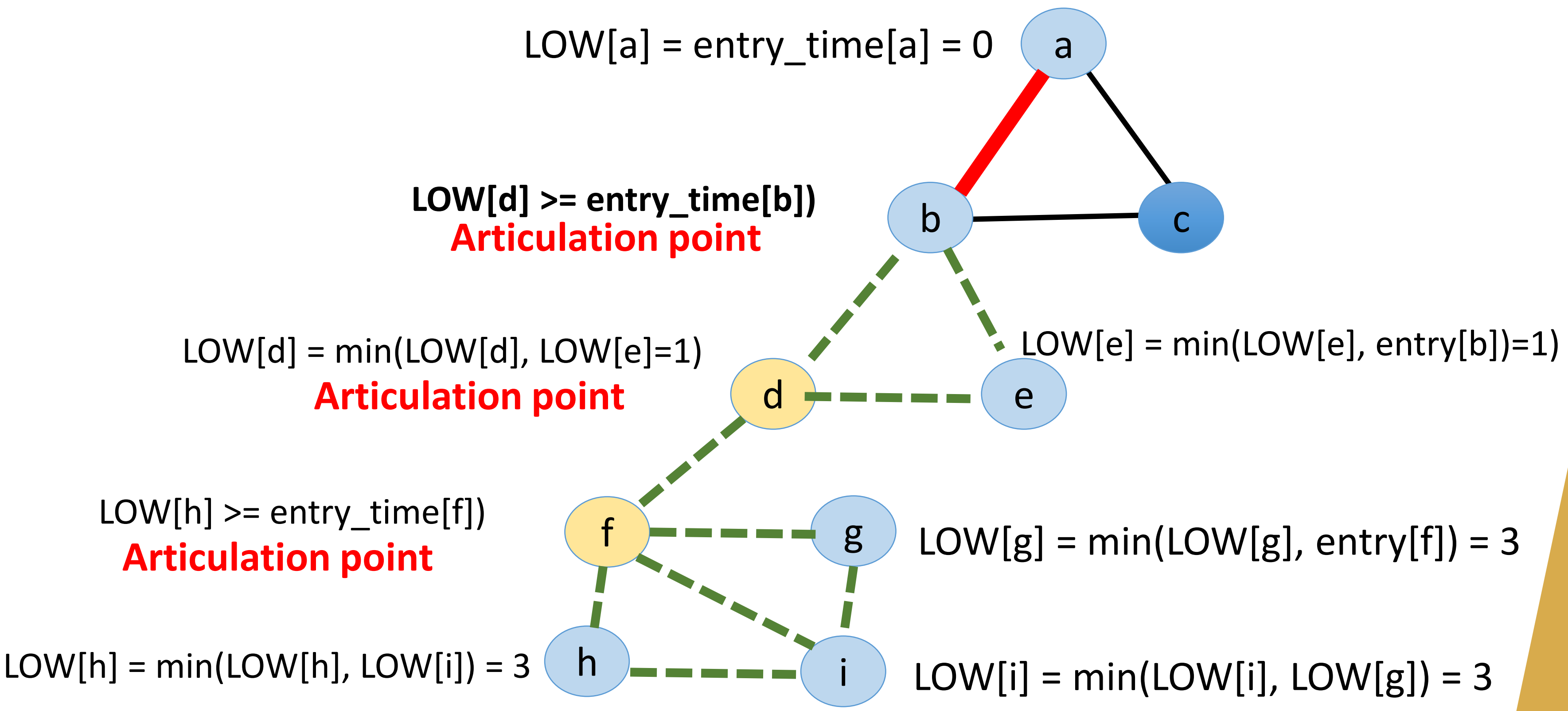
Tarjan’s Articulation Point Algorithm

Walkthrough Step 8: Backtrack from e and adjust the LOW[d] since e has an escape to b (in other words, d won’t cut vertex e off from escaping if vertex d is removed).



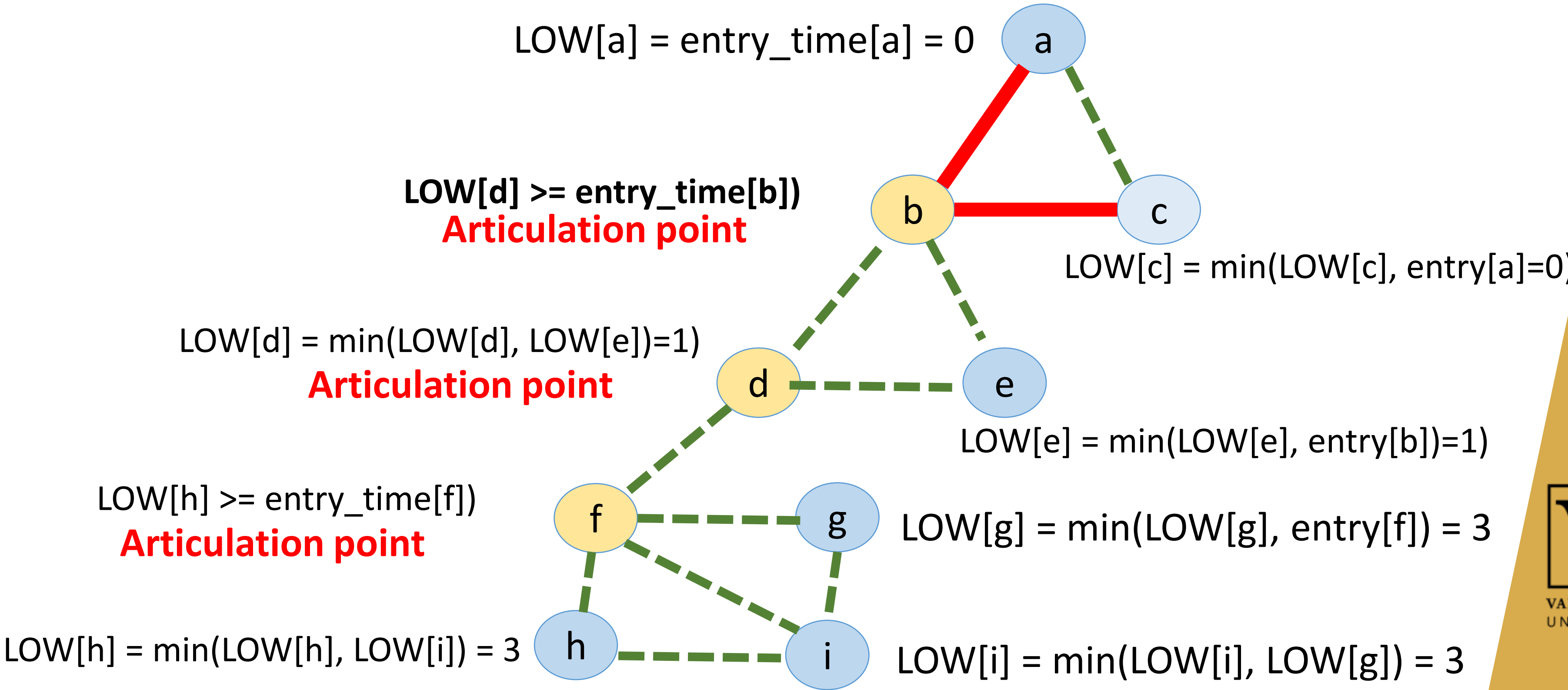
Tarjan’s Articulation Point Algorithm

Walkthrough Step 9: Backtrack to b and recognize that no vertex below b has an escape to anyone better than b. This means b is an **articulation point**.



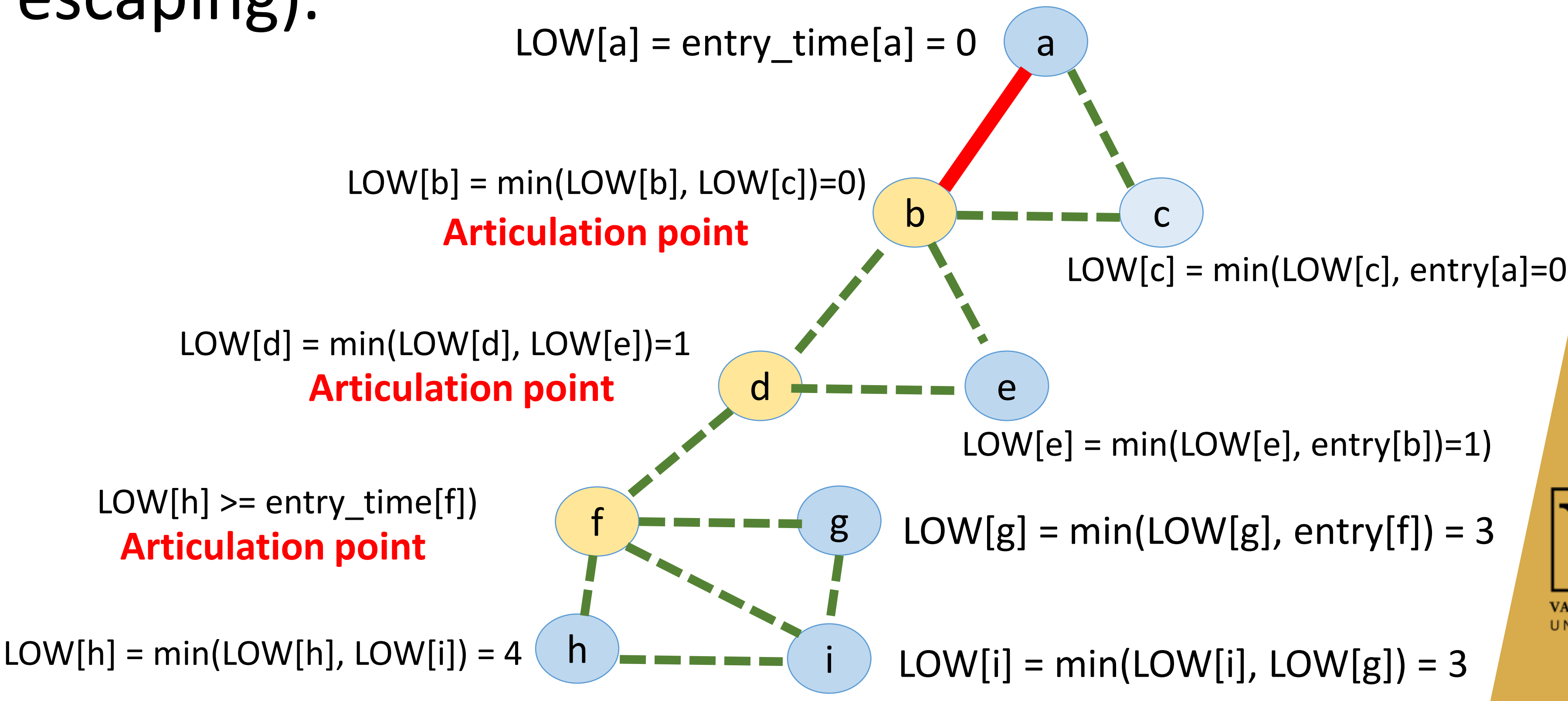
Tarjan’s Articulation Point Algorithm

Walkthrough Step 10: Deep dive from b and discover c. From c, we hit the already discovered a, so we adjust the LOW[c].



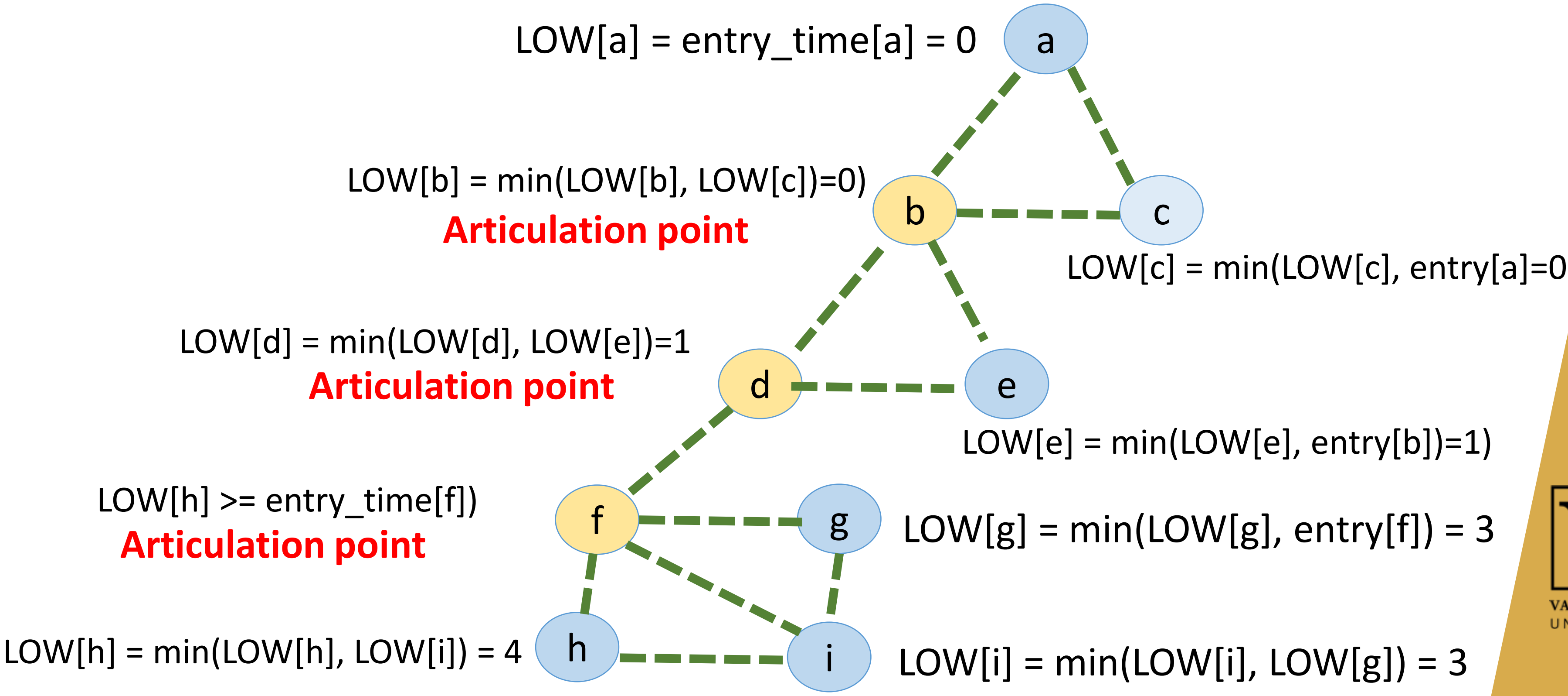
Tarjan’s Articulation Point Algorithm

Walkthrough Step 11: Backtrack from c and adjust the LOW[b] since vertex c has an escape to vertex a (meaning that removing b won’t cut c off from escaping).



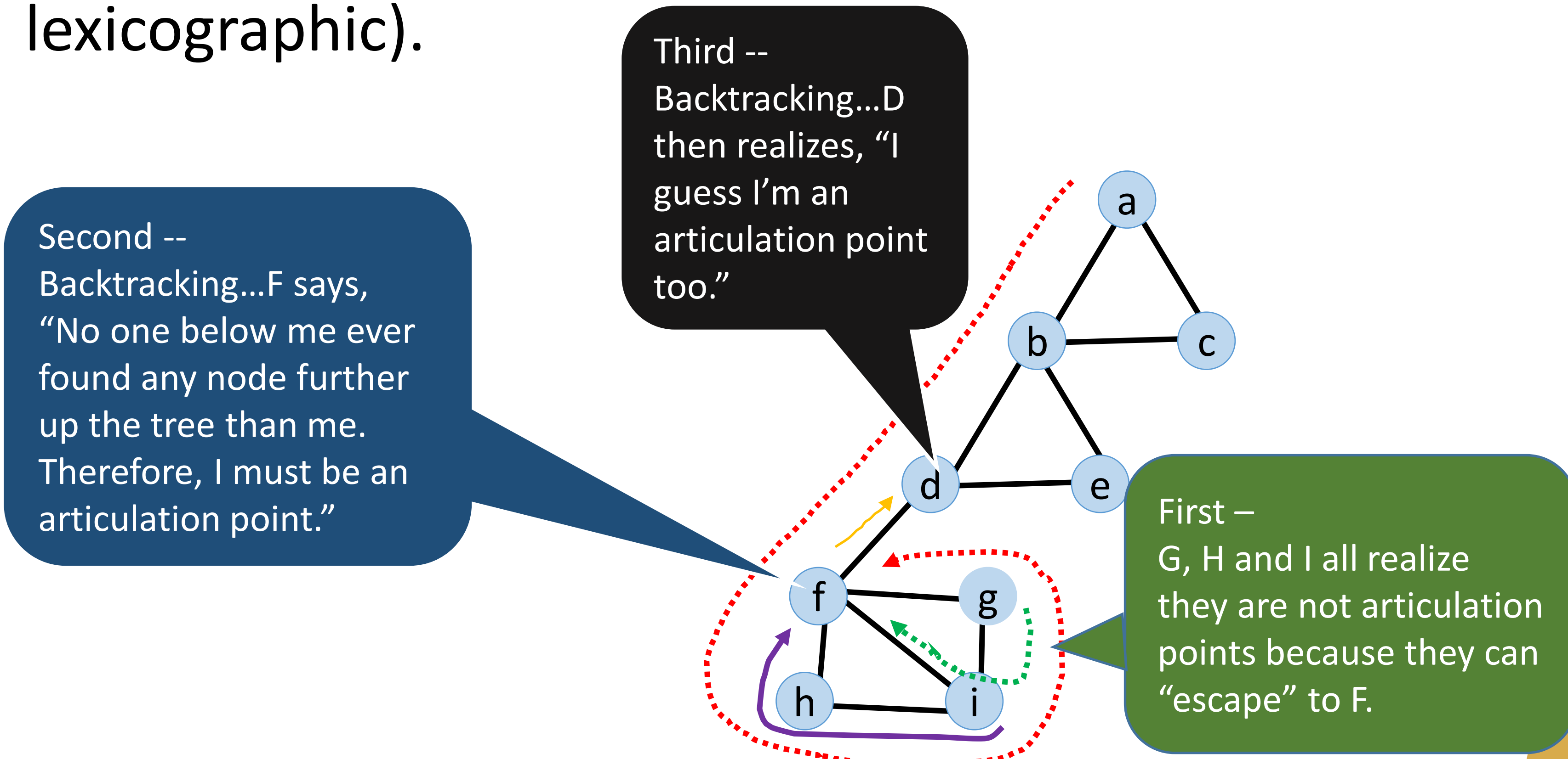
Tarjan's Articulation Point Algorithm

Walkthrough Step 12: Backtrack from b and recognize that vertex a is the root of the DFS tree of discovery with only one child. Therefore vertex "a" is not an articulation point. DFS now complete and we have located all articulation points.



Tarjan's Articulation Point Algorithm

- **Summary:** For every node x, we need to determine the **earliest** discovered vertex that can be reached from the subtree rooted at x. Consider a DFS on the graph below and say the order of vertices discovered is a, b, d, f, h, i, g (not lexicographic).



Tarjan's Articulation Point Algorithm

Execute DFS(v)

When vertex v is discovered, LOW[v] = entry_time[v] = time

time = time + 1

For each of v's neighbors, neigh

 if neigh is undiscovered

  execute DFS(neigh)

  LOW[v] = min of {LOW[v], LOW[neigh]}

  if LOW[neigh] ≥ entry_time[v], v is articulation point

 else if (neigh is not v's parent but has been discovered)

  LOW[v] = min of {LOW[v] and entry_time[neigh]}

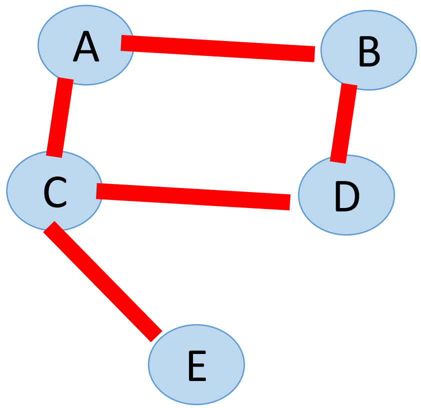
Total = O(V+E)

More informative: Θ(V+E)

VANDERBILT UNIVERSITY

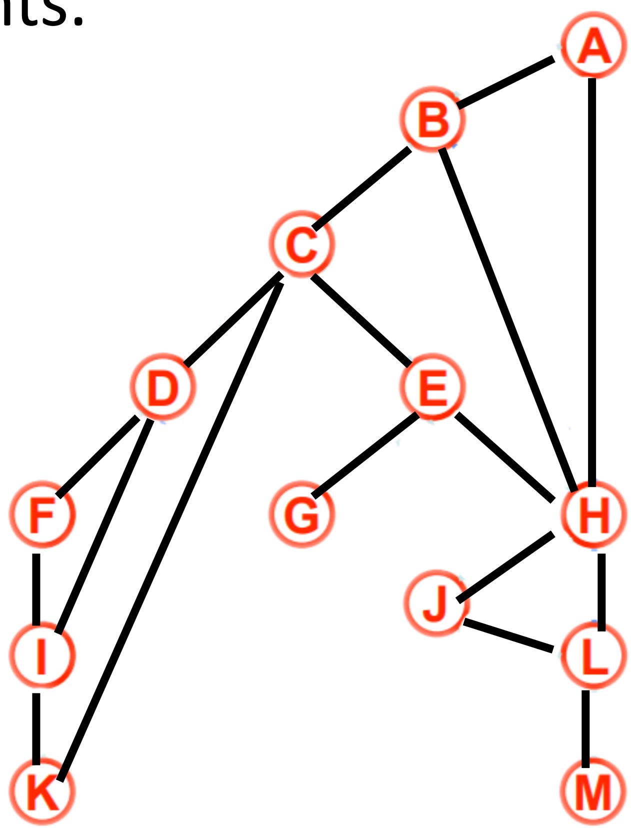
Walkthrough: Tarjan’s Algorithm

- **Your Turn:** Perform Tarjan’s algorithm on the graph below with three different starting vertices.
- Draw the Tree of Discovery and identify the articulation points.
- Are they the same or different when you start at a different vertex?



On Your Own: Tarjan's Algorithm

- Walk through and explain Tarjan's algorithm using the graph below starting at vertex A. When given a choice between 2 paths, choose the one that comes first numerically. Identify all articulation points.



VANDERBILT UNIVERSITY

That's All For Now…

- Coming to a Slideshow Near You Soon…
  1. Kosaraju's Algorithm for SCCs
  2. Minimal Spanning Tree
  3. Prim's Algorithm

That's All For Now