Data Structures in Java

Lecture 19: Applications of DFS

11/30/2015

Daniel Bauer

Contents

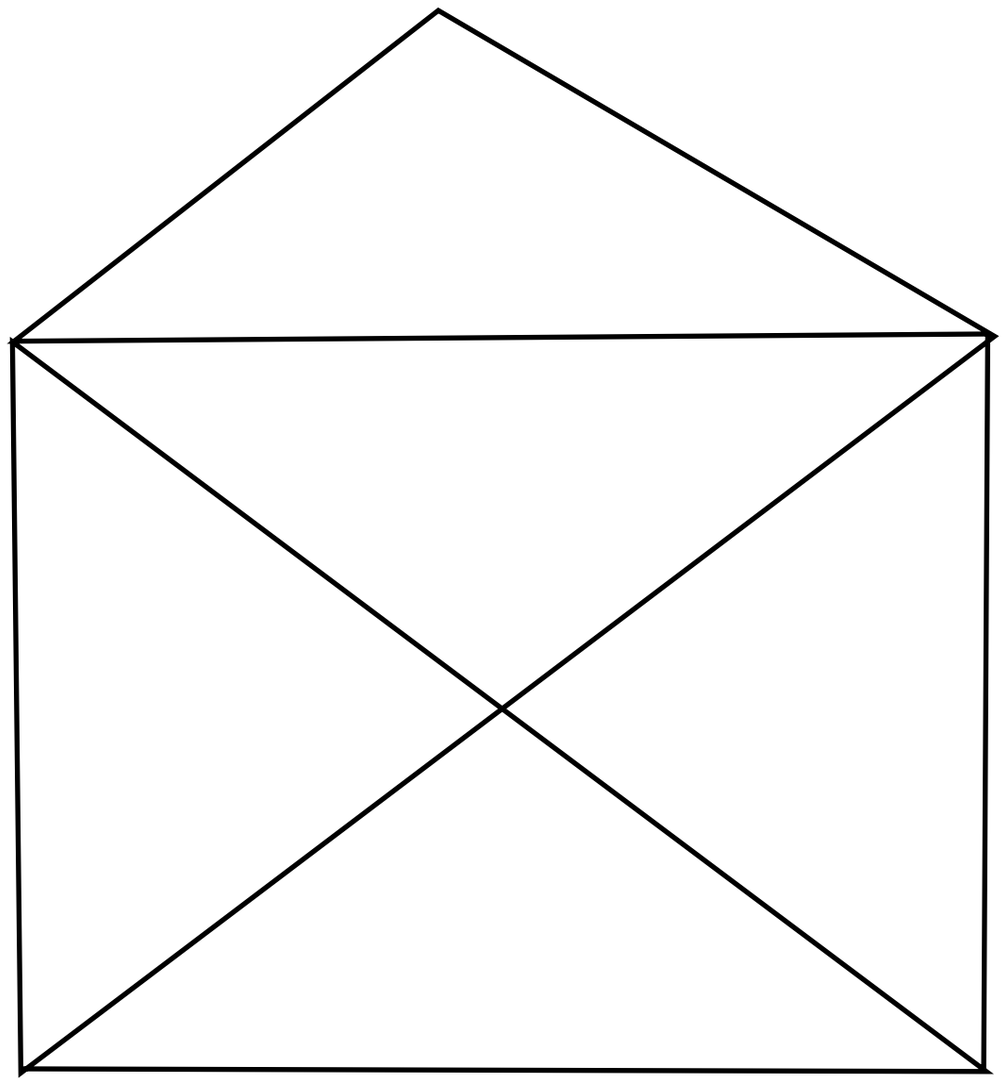
- Applications of DFS
- Euler Circuits
- Biconnectivity in Undirected Graphs.
- Finding Strongly Connected Components for Directed Graphs.

Contents

- Applications of DFS
- **Euler Circuits**
- Biconnectivity in Undirected Graphs.
- Finding Strongly Connected Components for Directed Graphs.

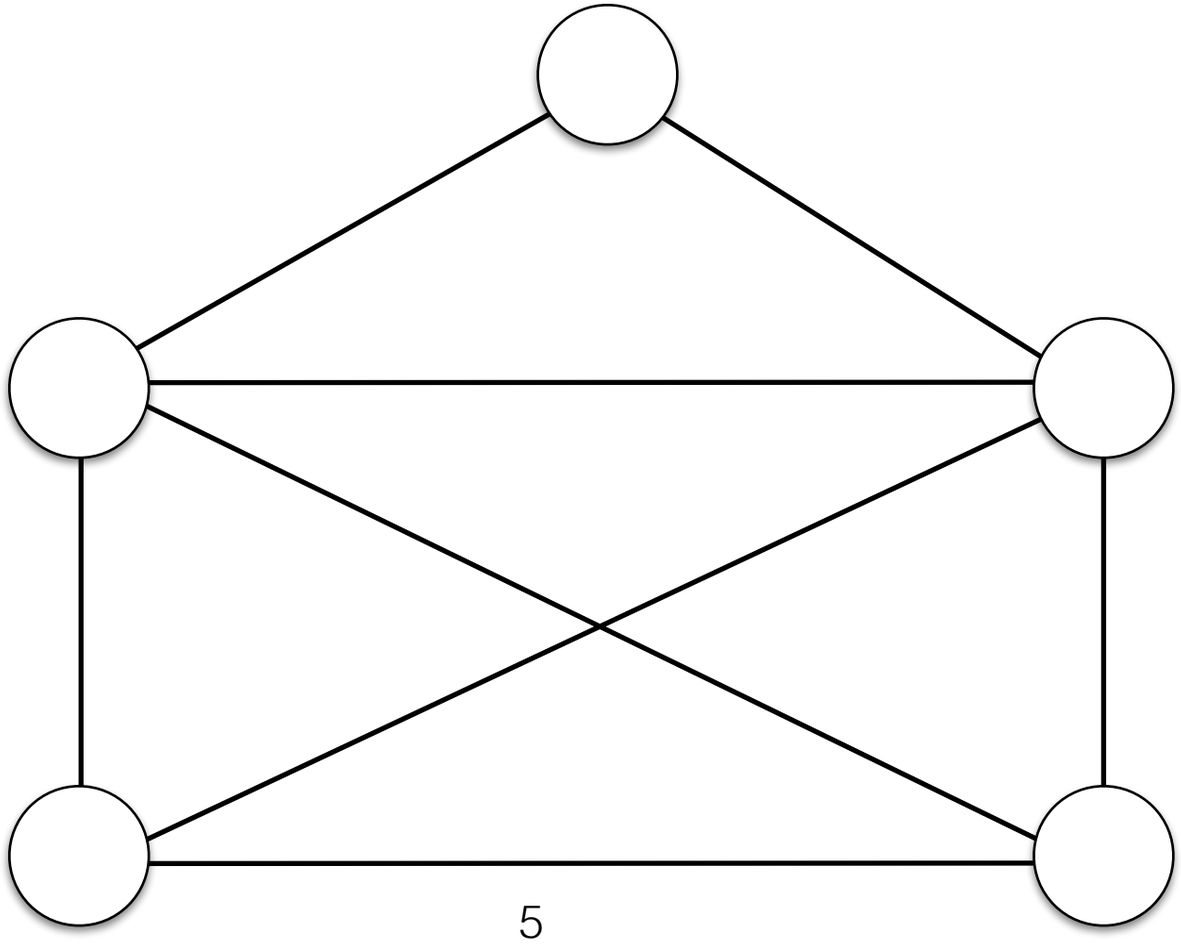
Draw this Figure

Without lifting your pen off the paper.



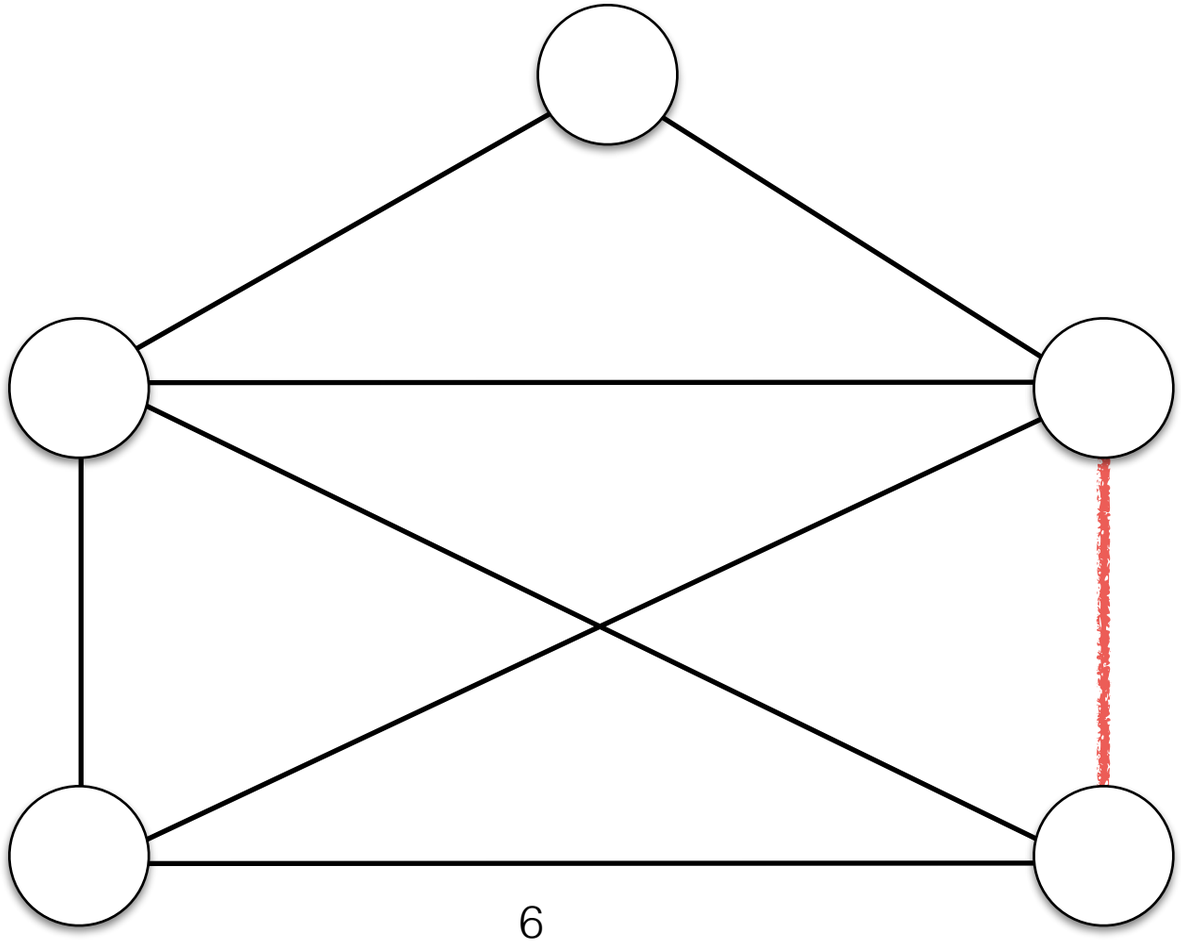
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



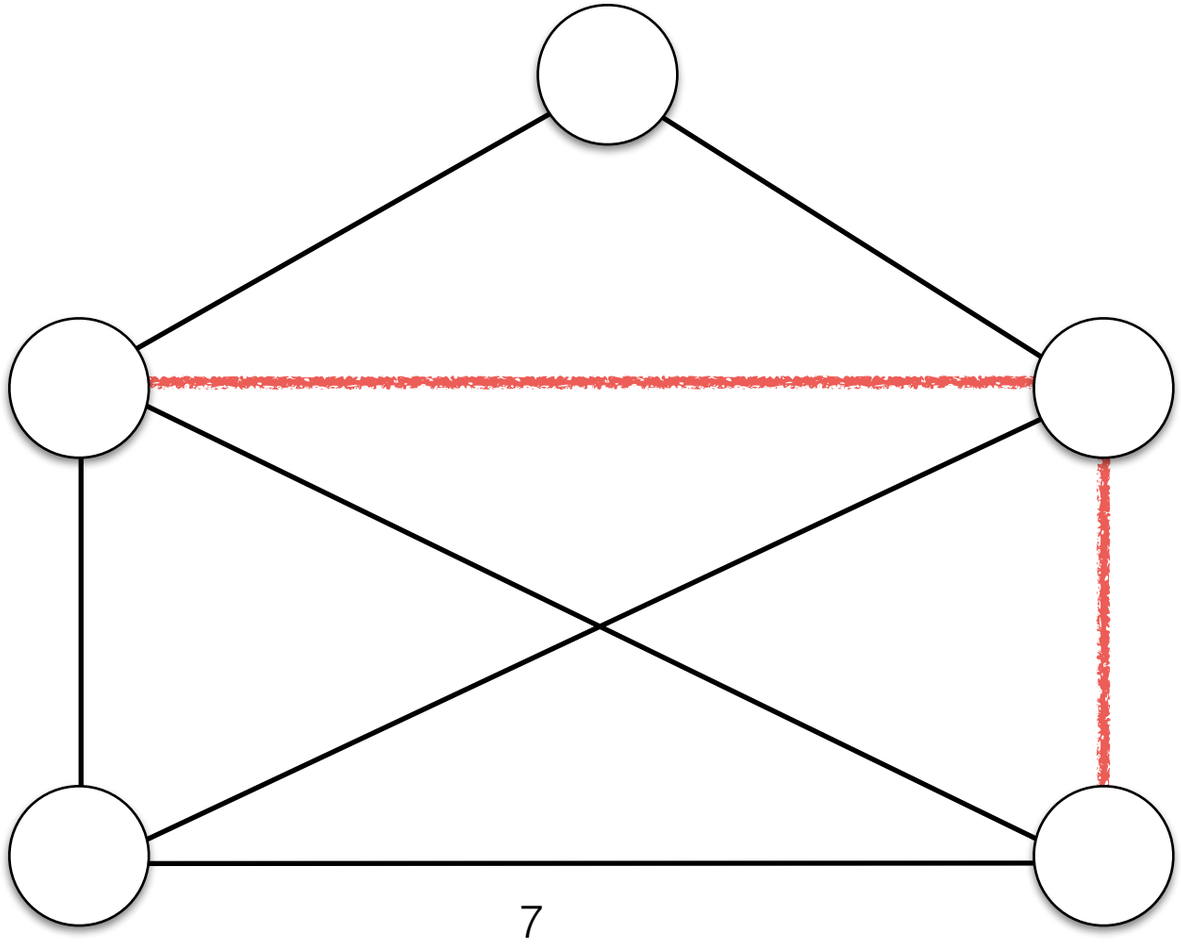
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



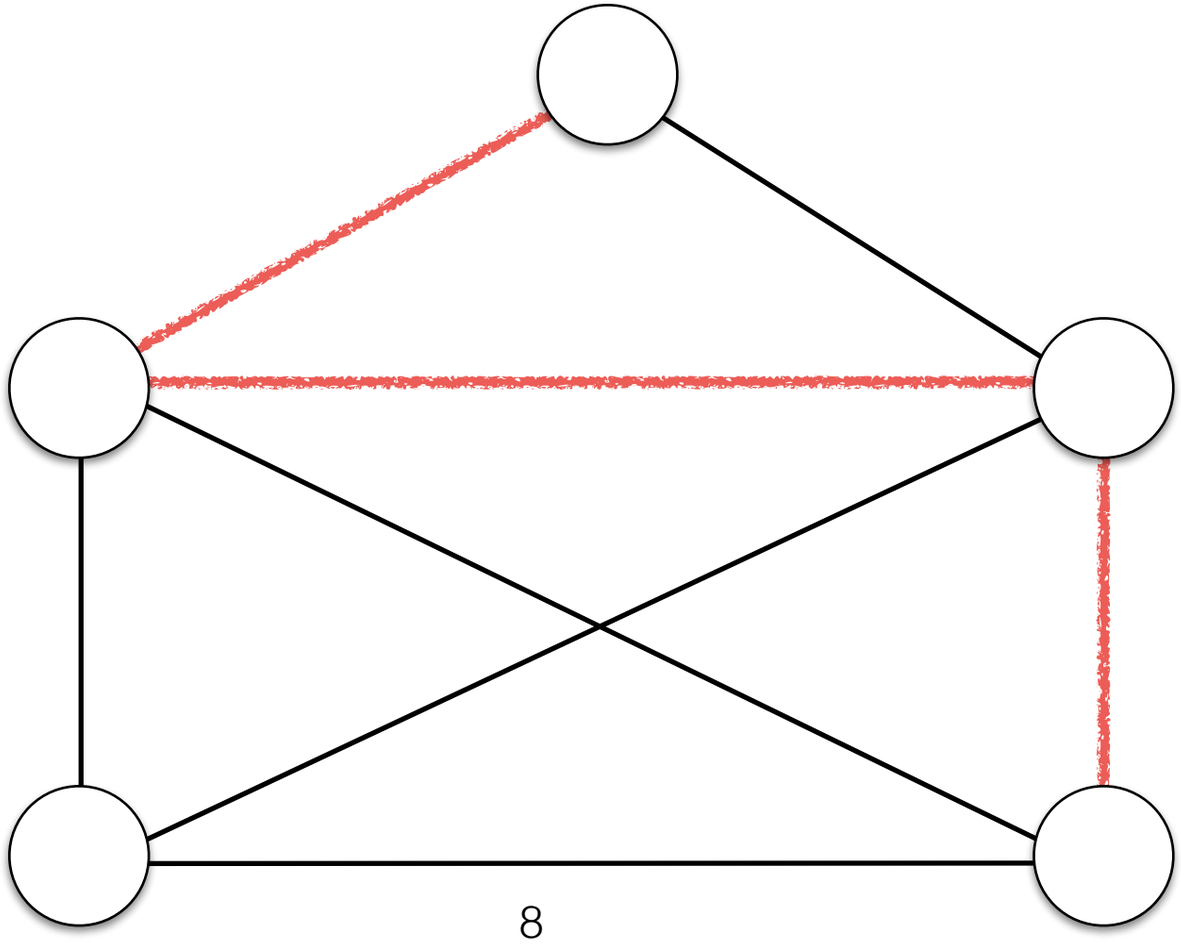
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



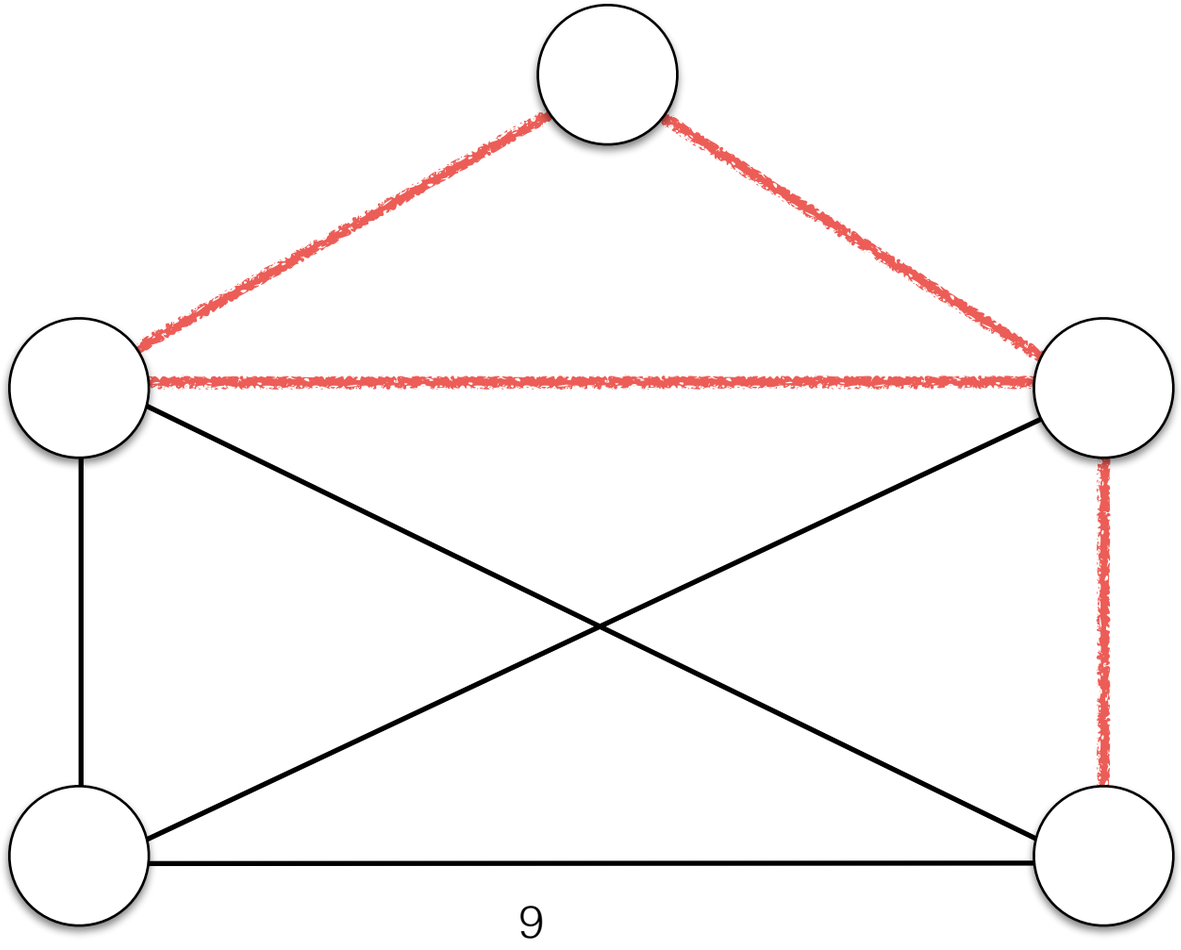
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



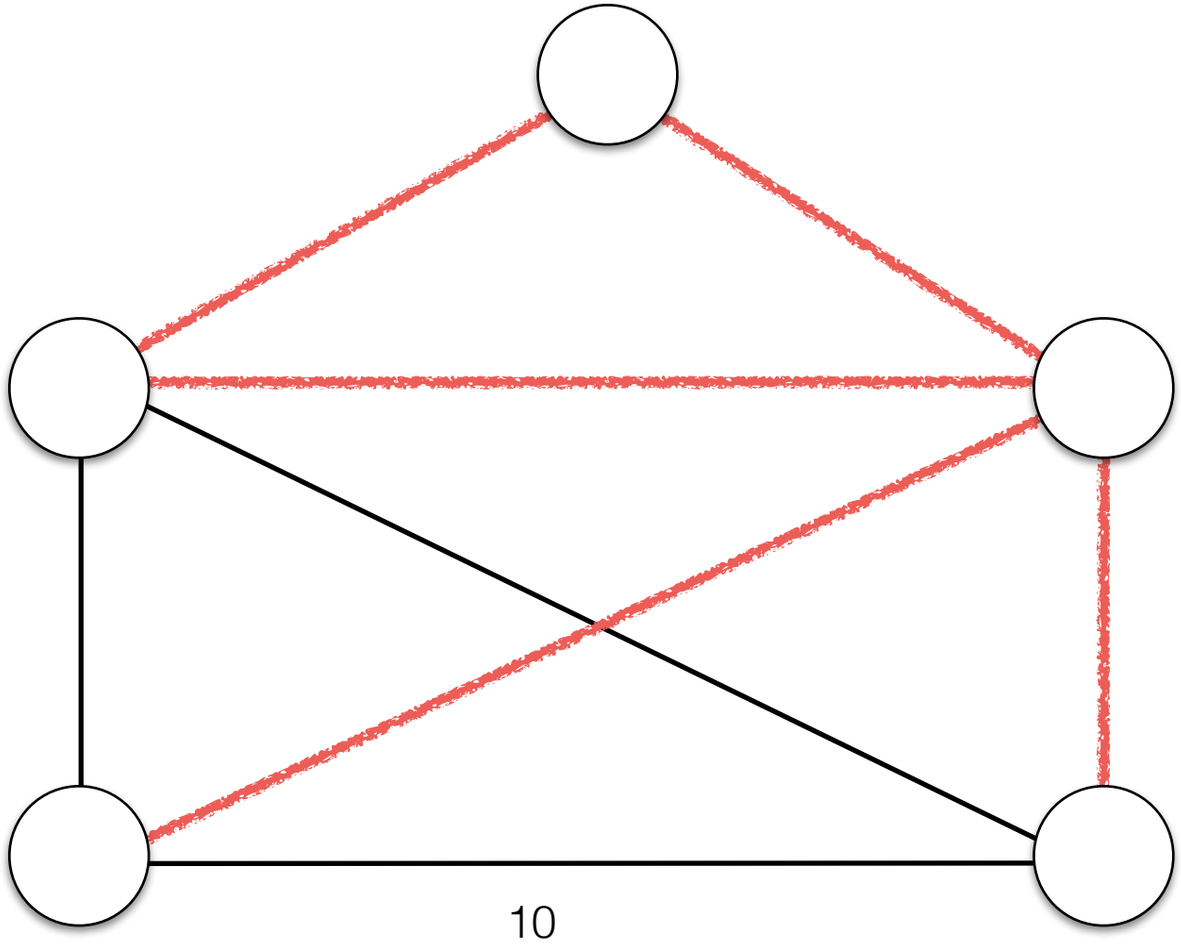
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



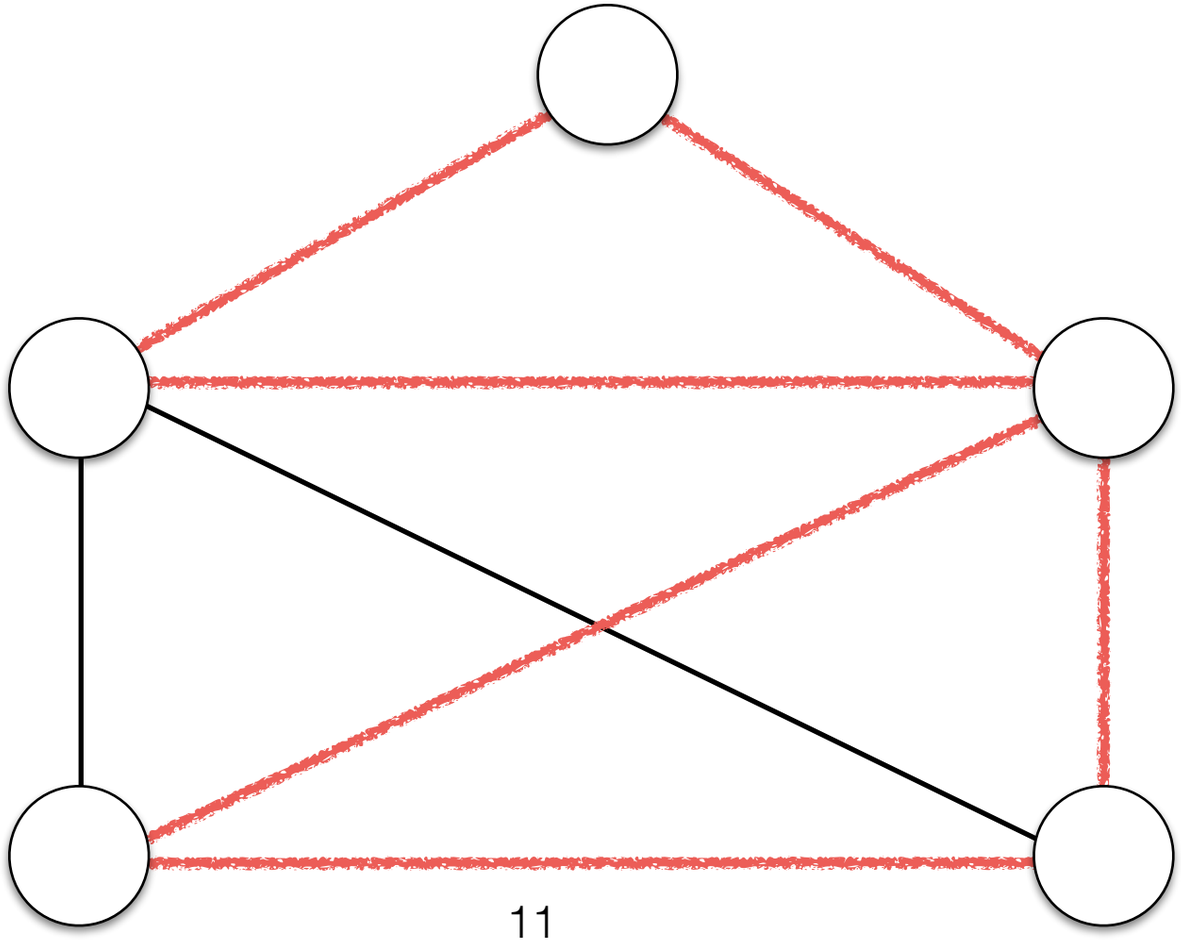
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



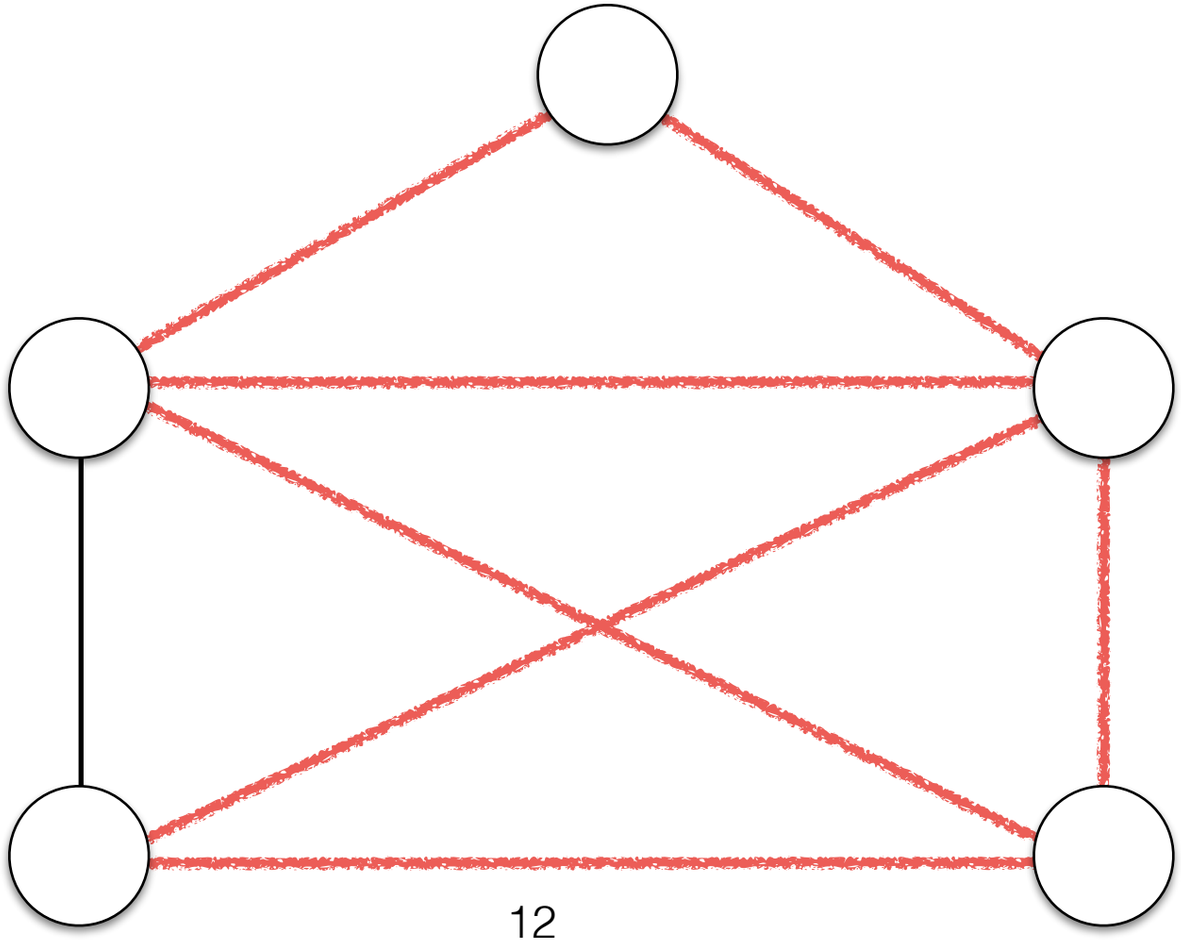
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



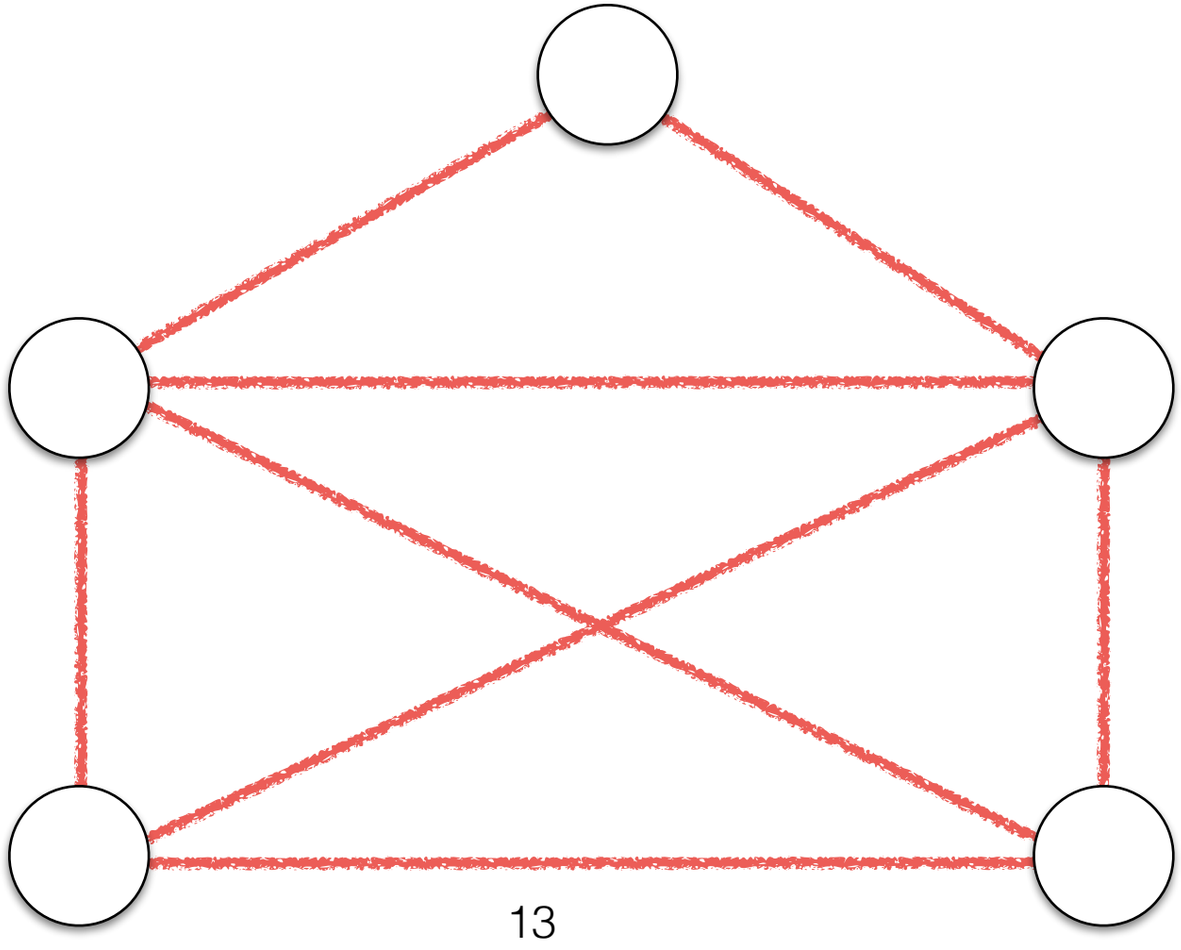
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



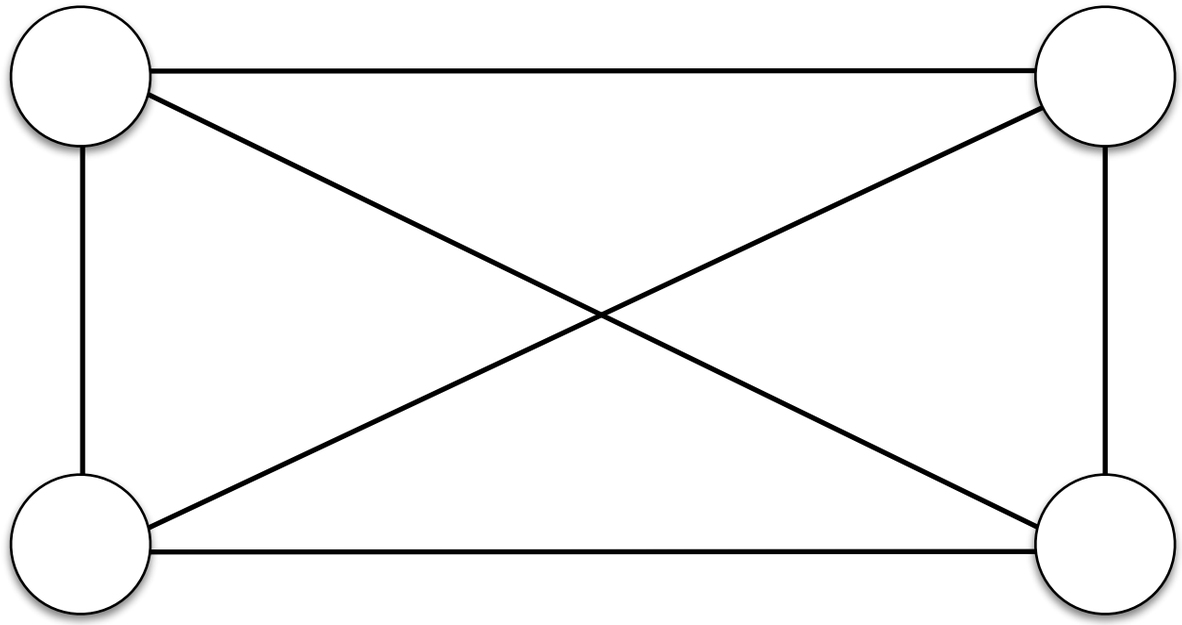
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



Euler Paths

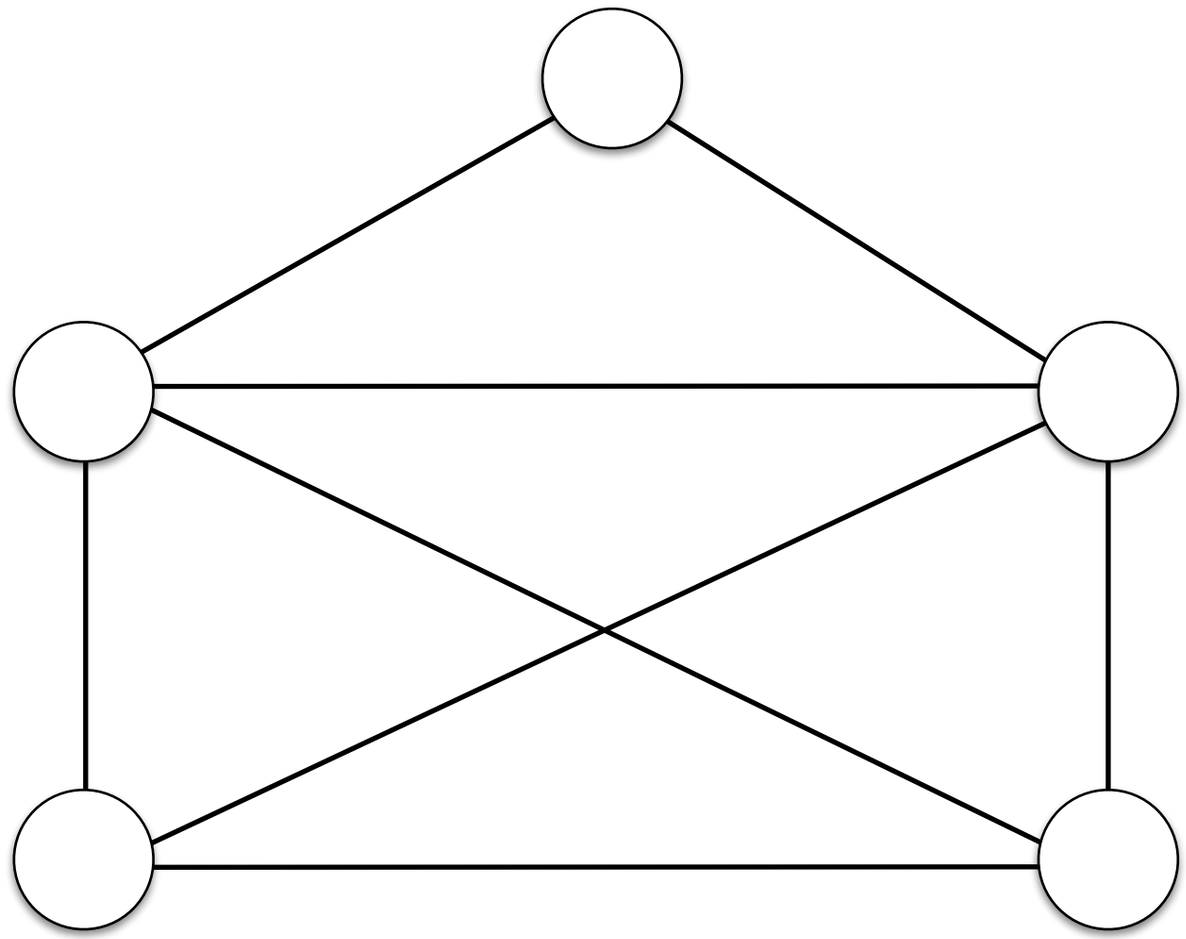
- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



This graph does not have an Euler Path.

Euler Circuit

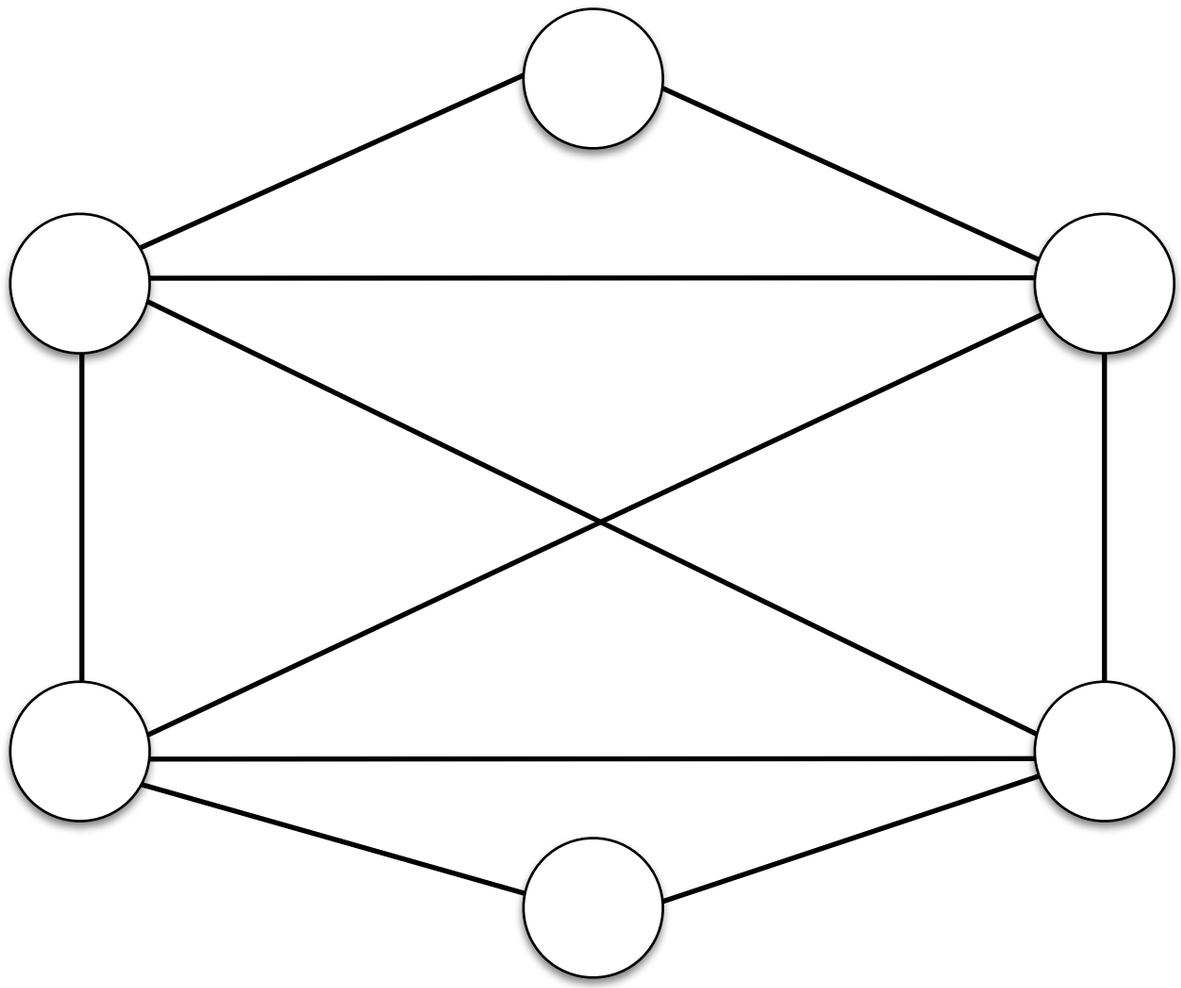
- An Euler Circuit is an Euler path that begins and ends at the same node.



This graph does NOT have an Euler Circuit

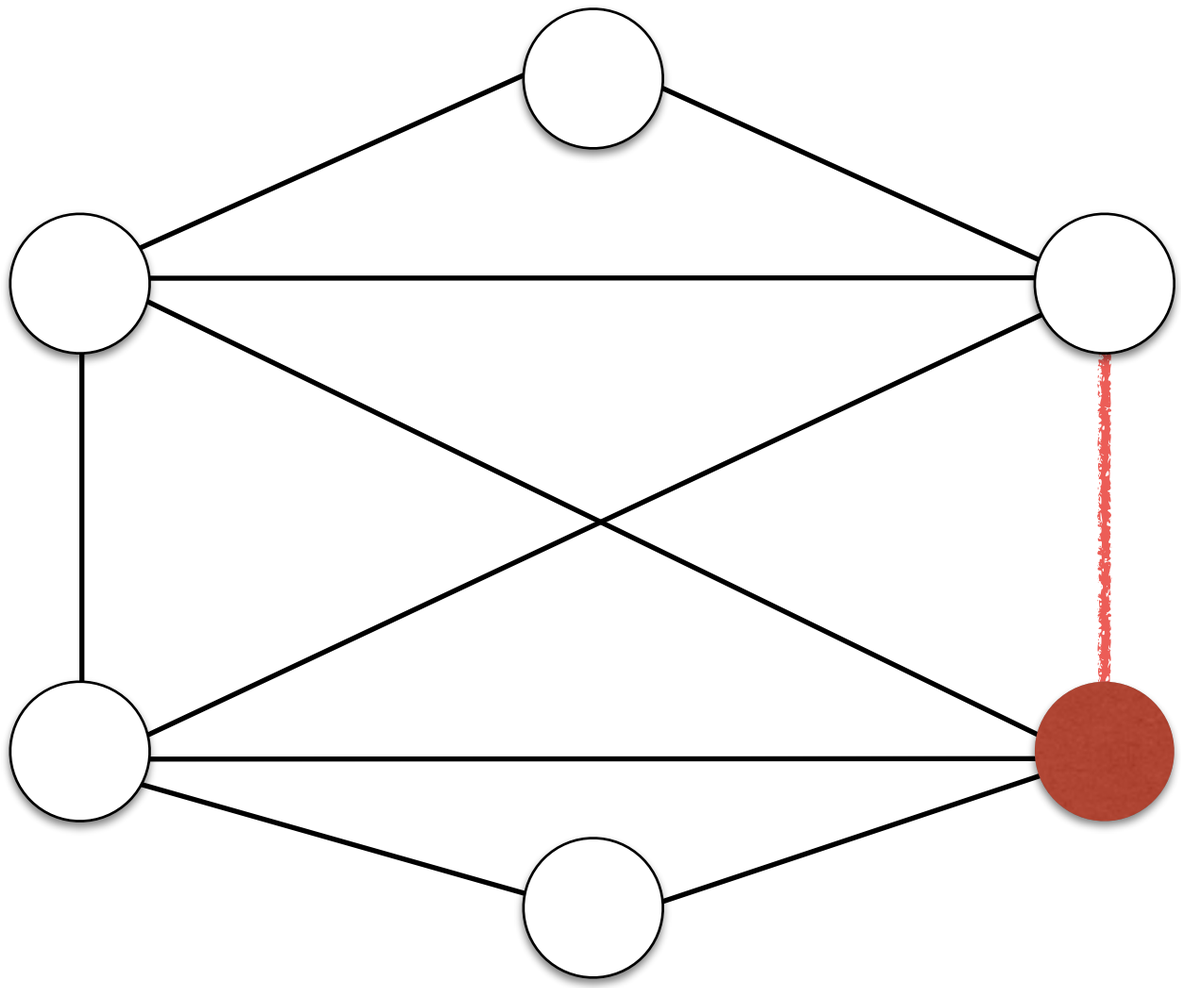
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



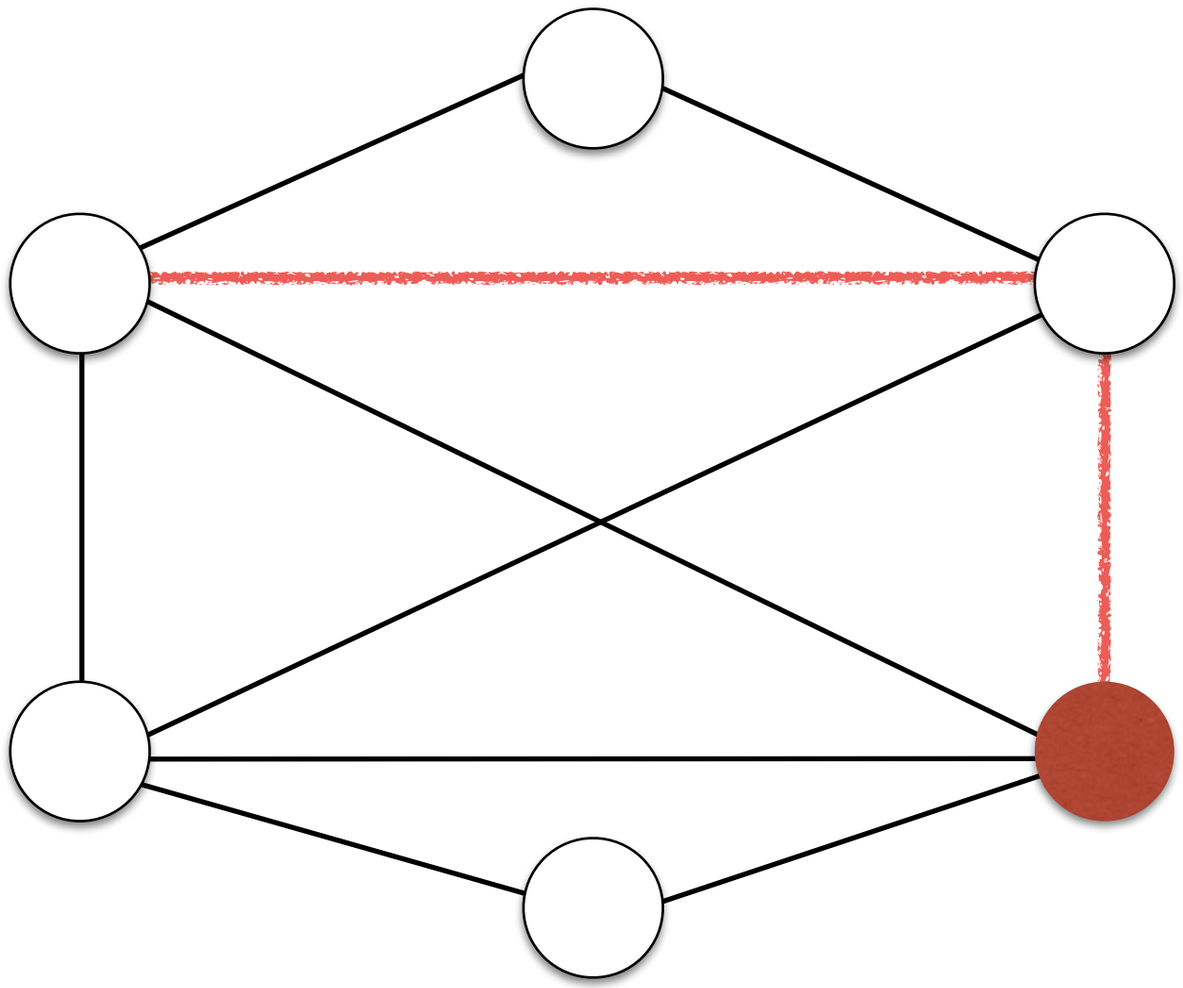
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



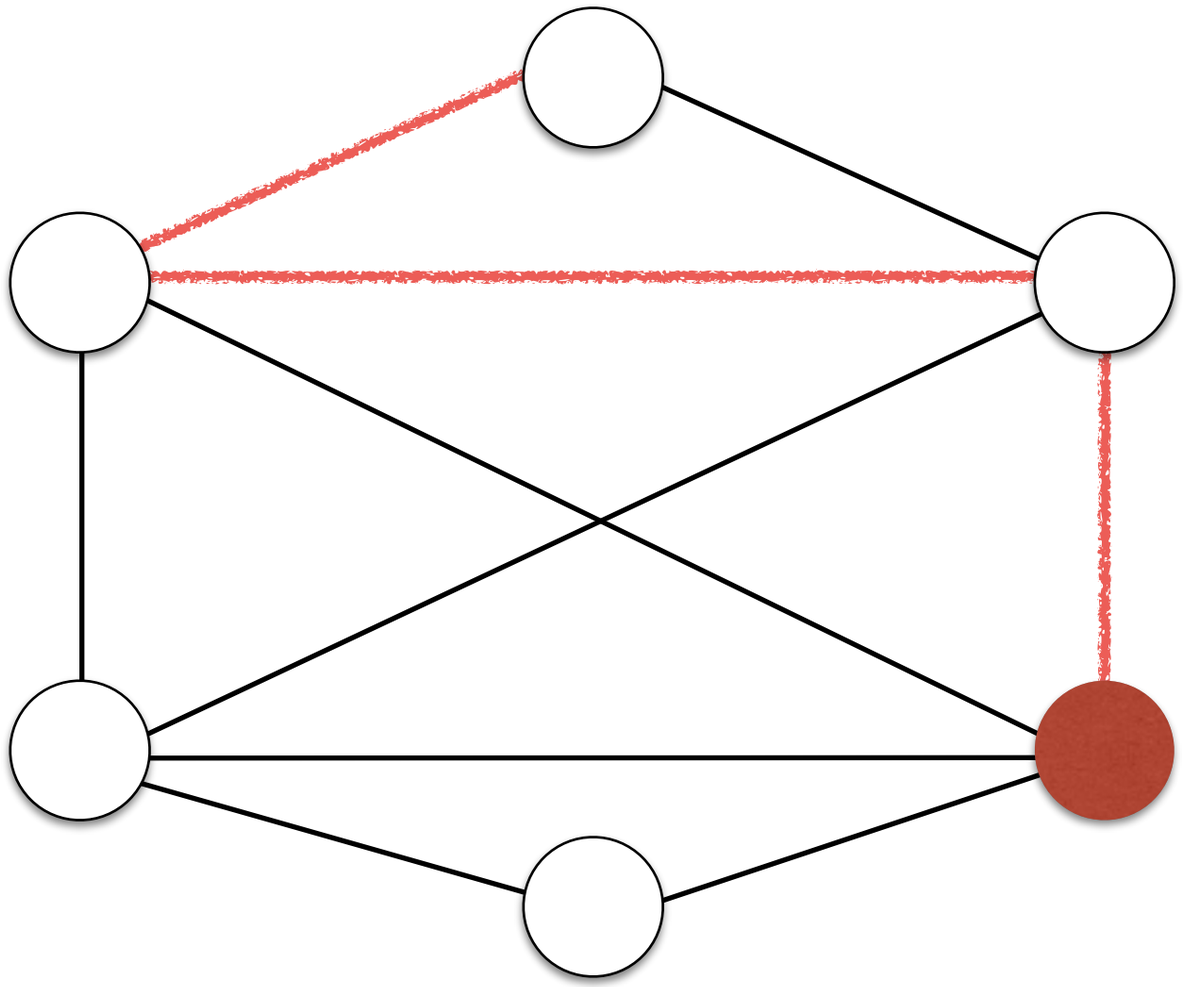
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



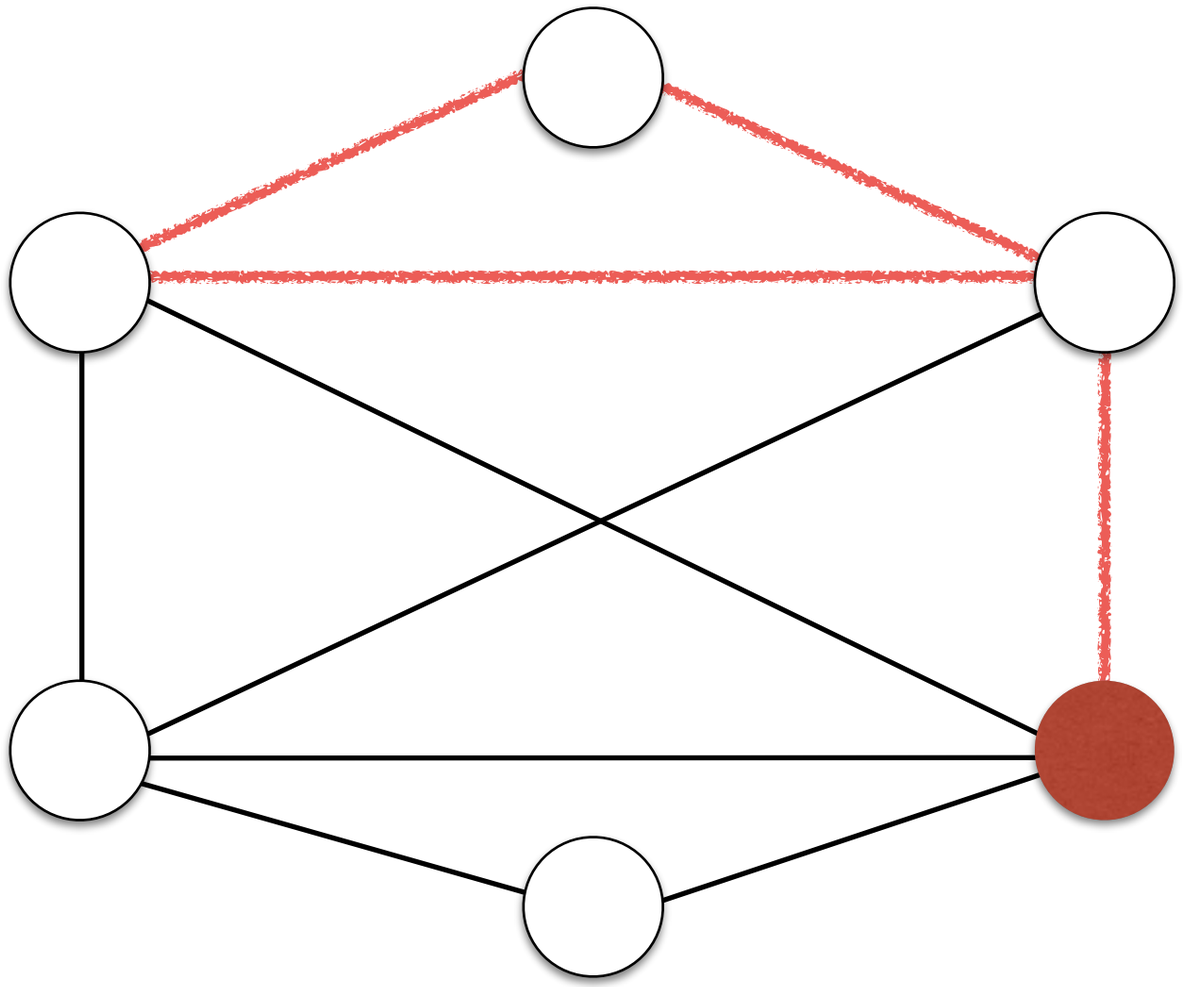
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



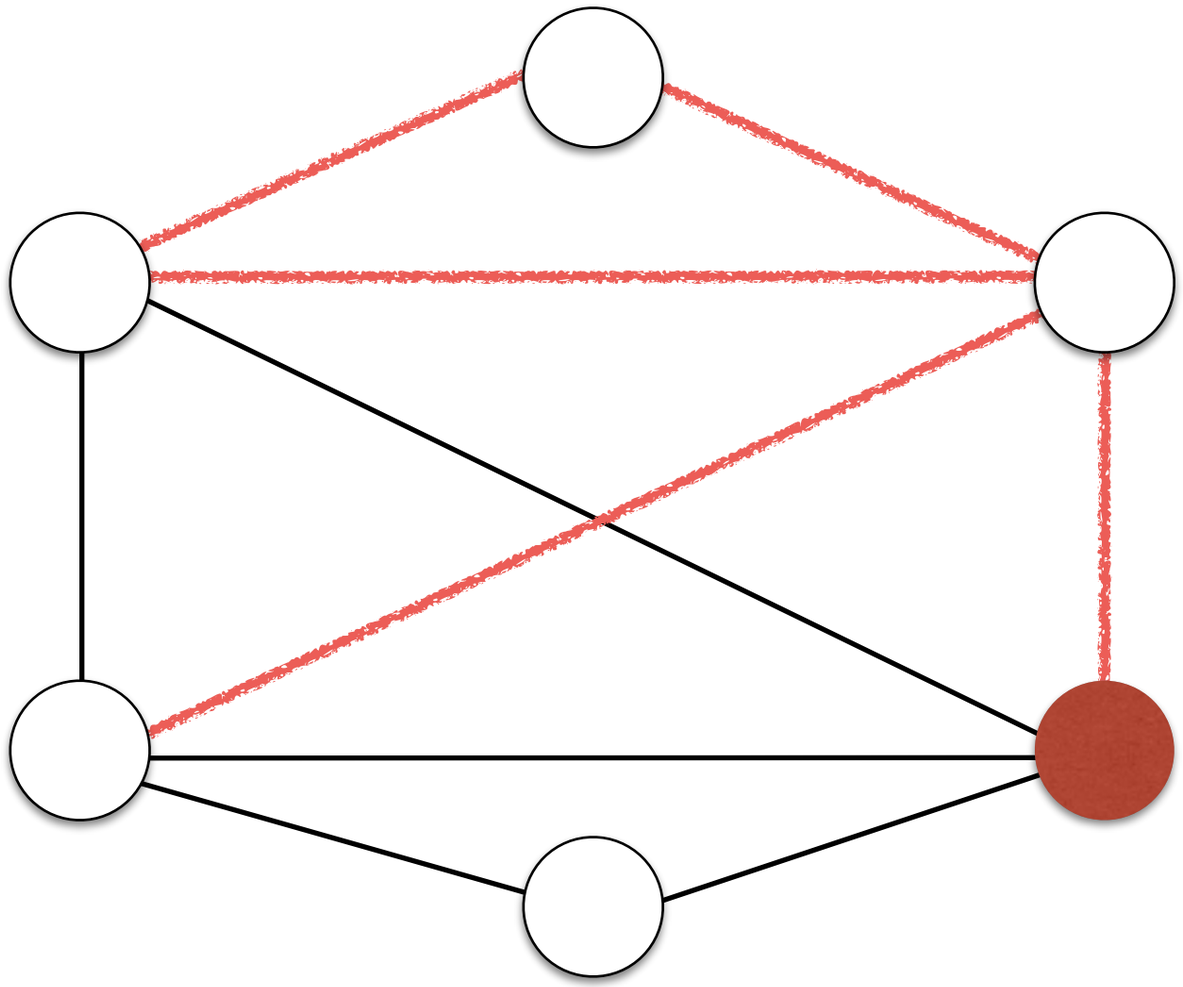
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



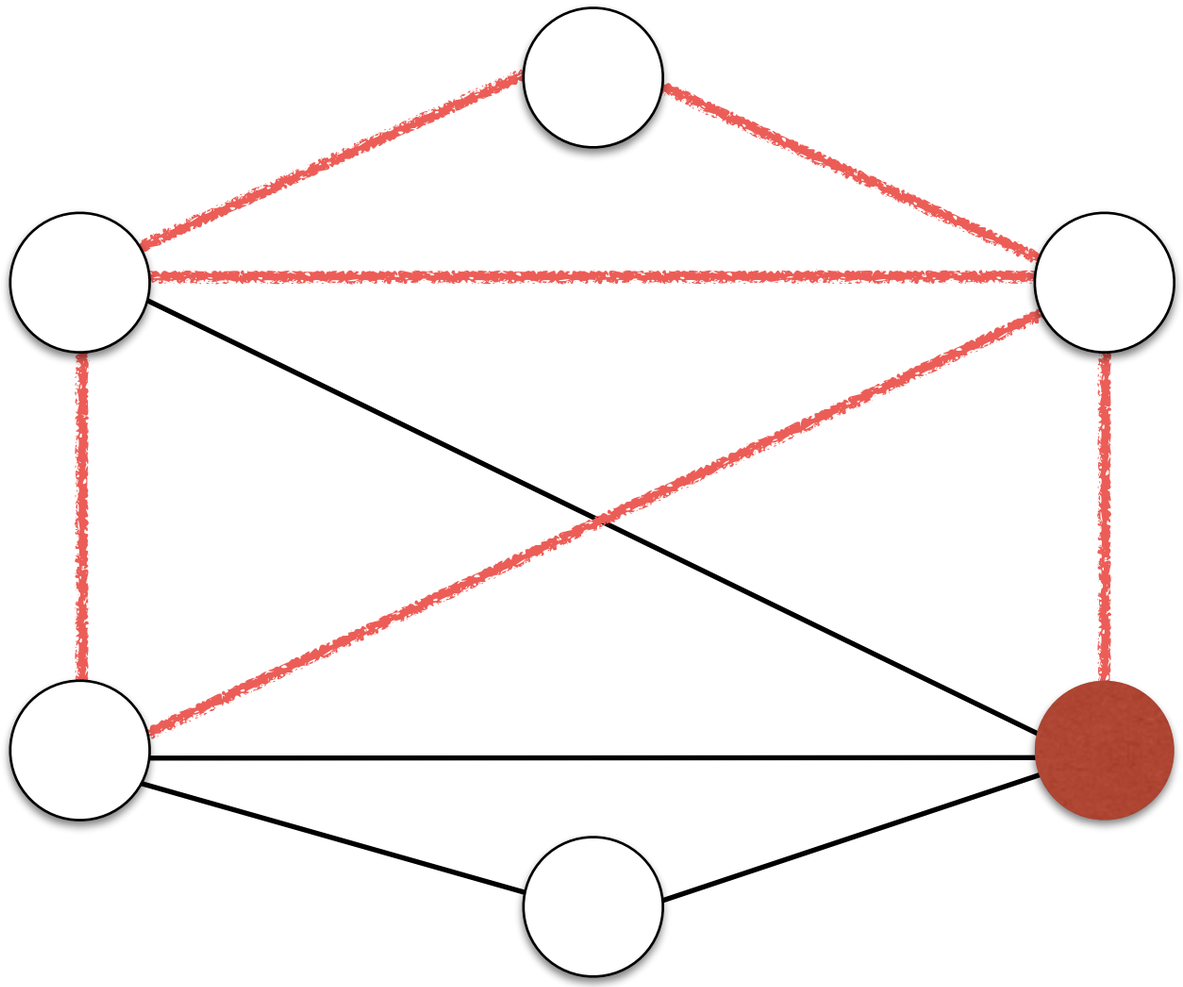
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



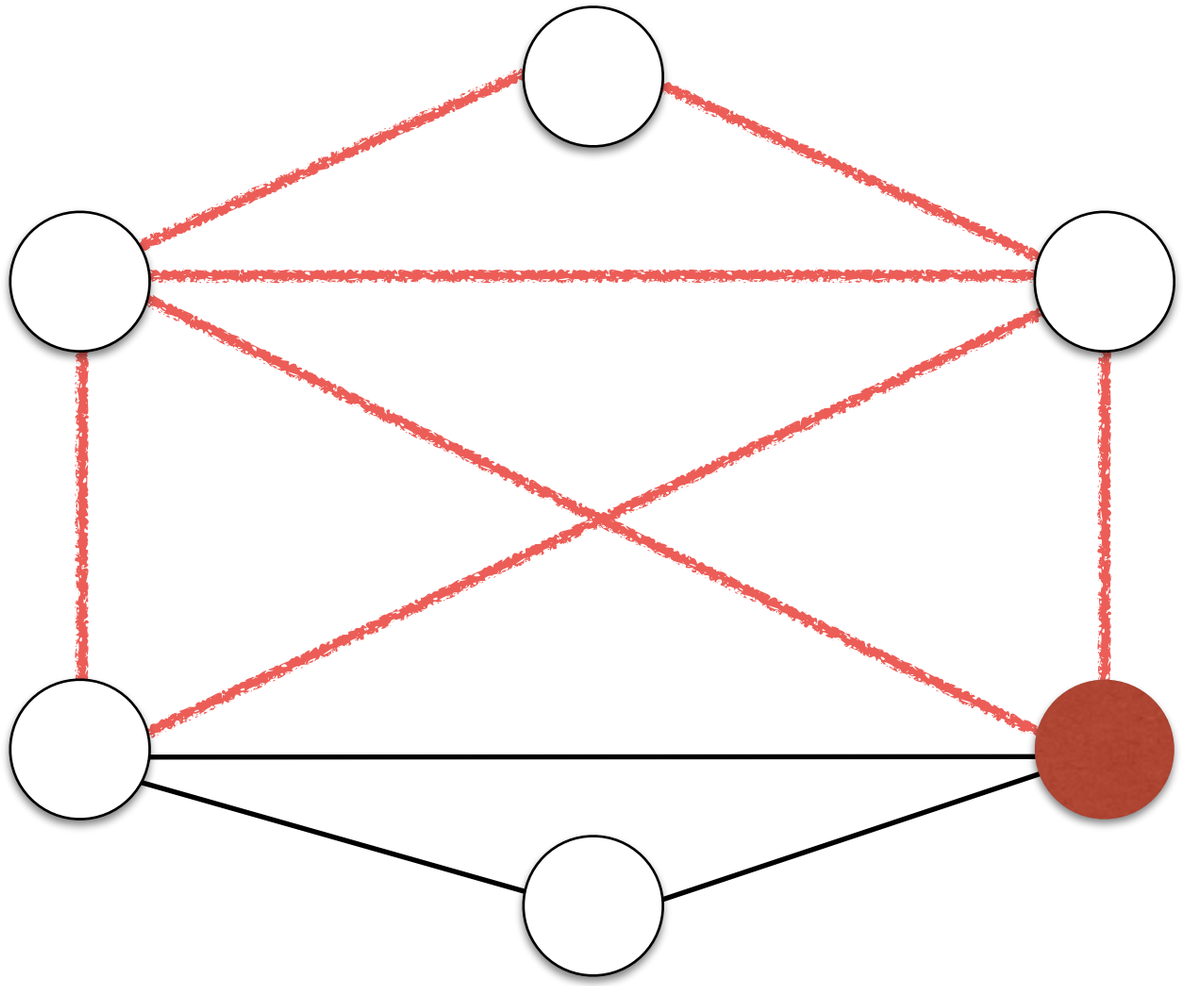
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



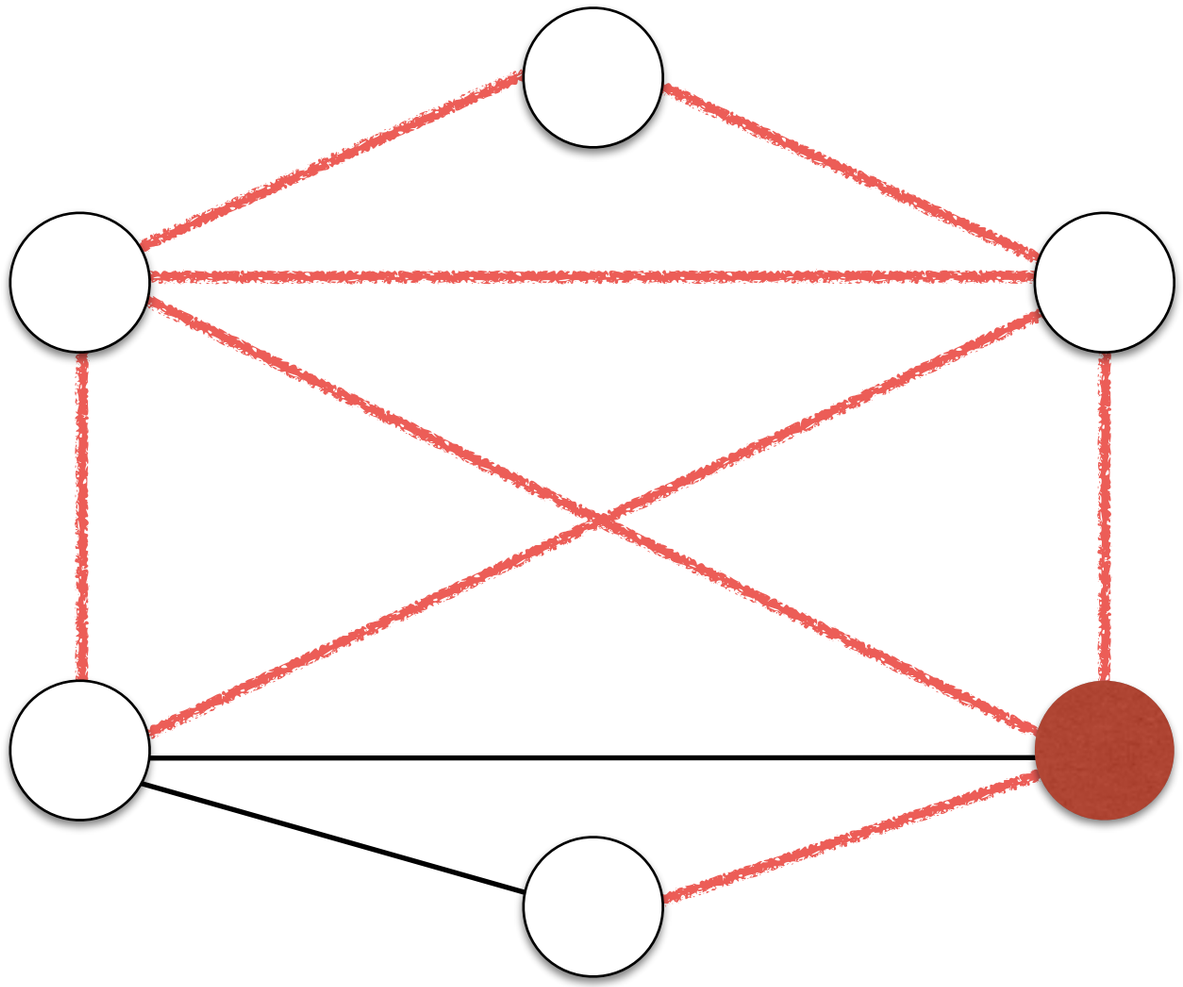
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



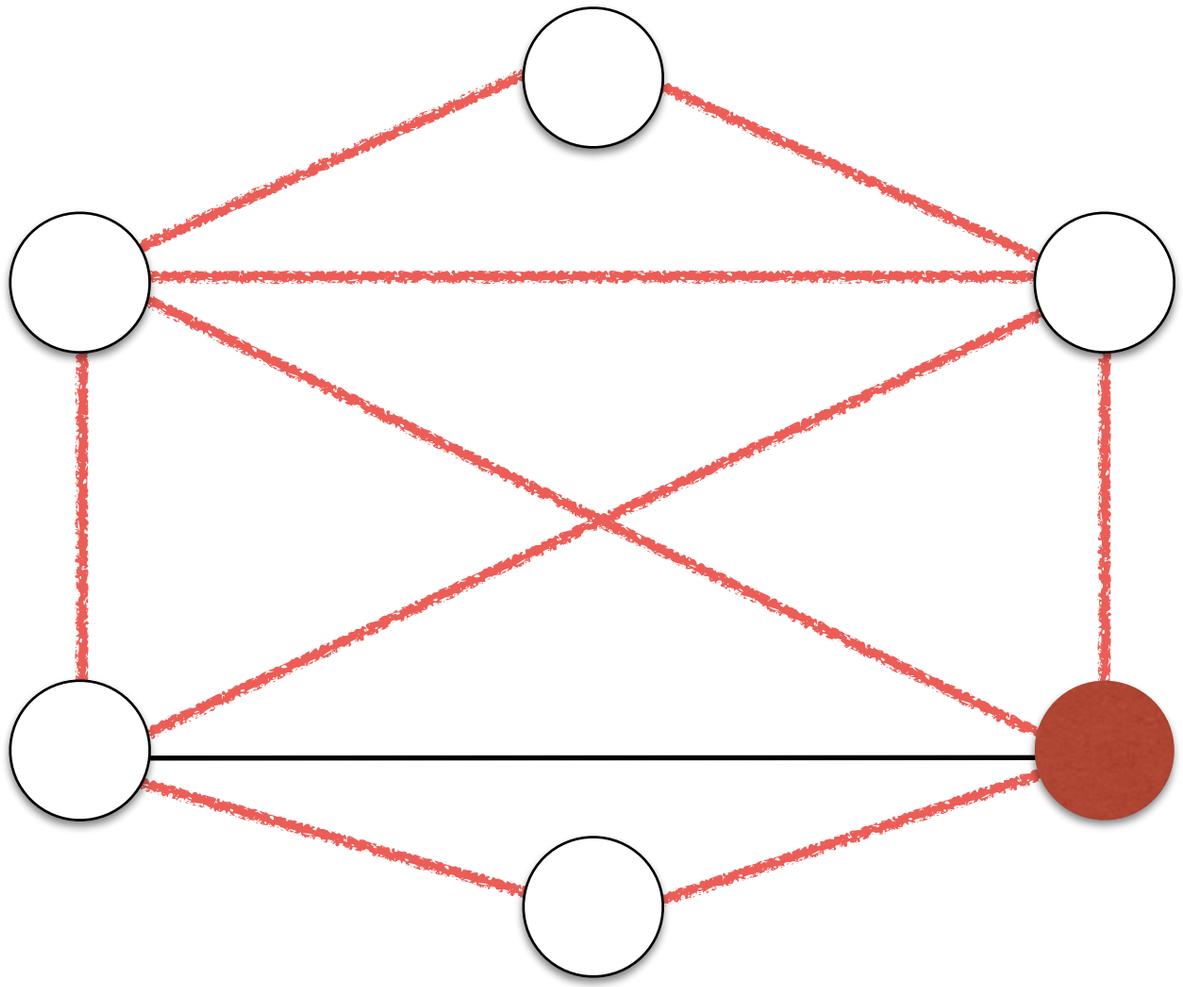
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



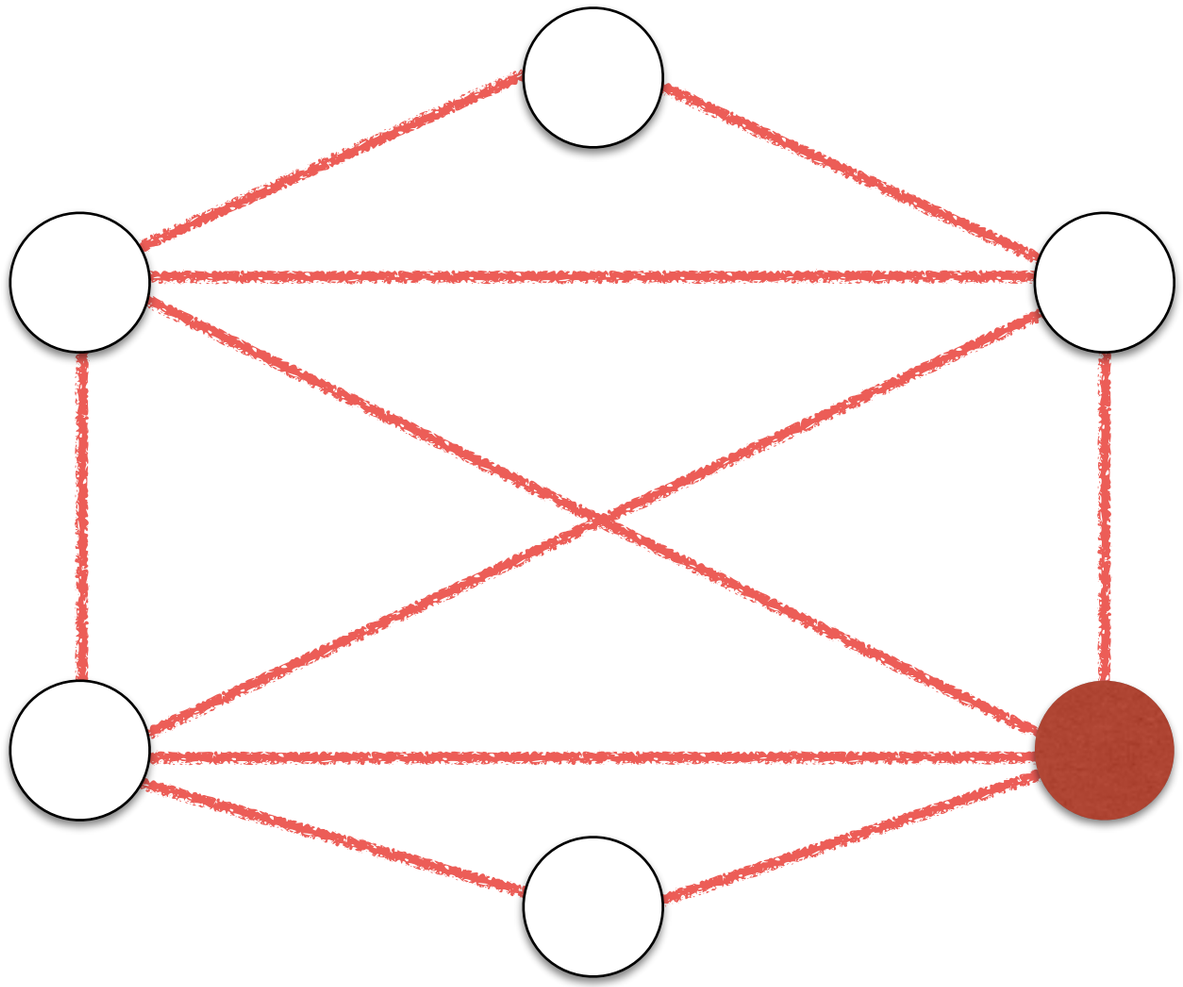
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



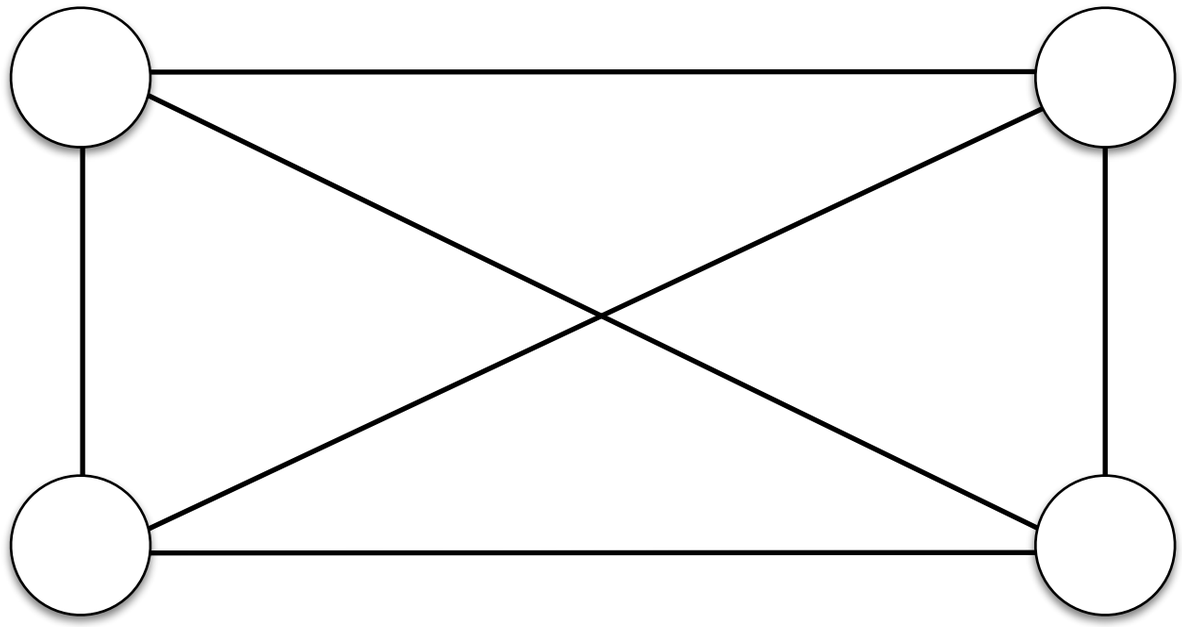
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



Euler Circuit

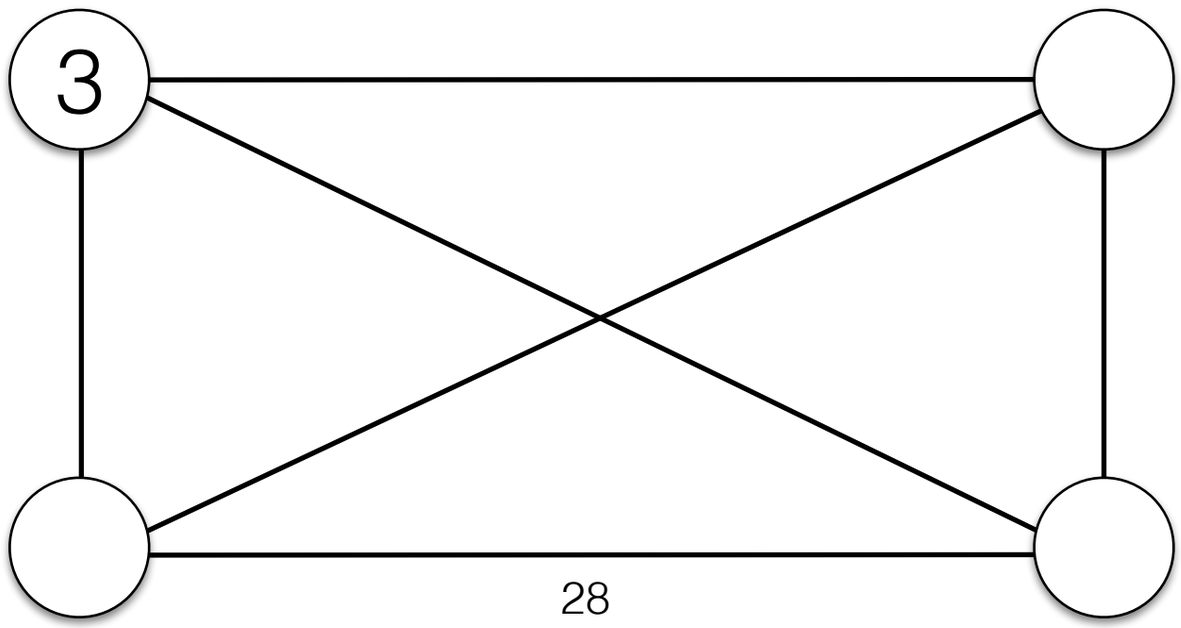
- An Euler Circuit is an Euler path that begins and ends at the same node.



This graph does not have an Euler Path.

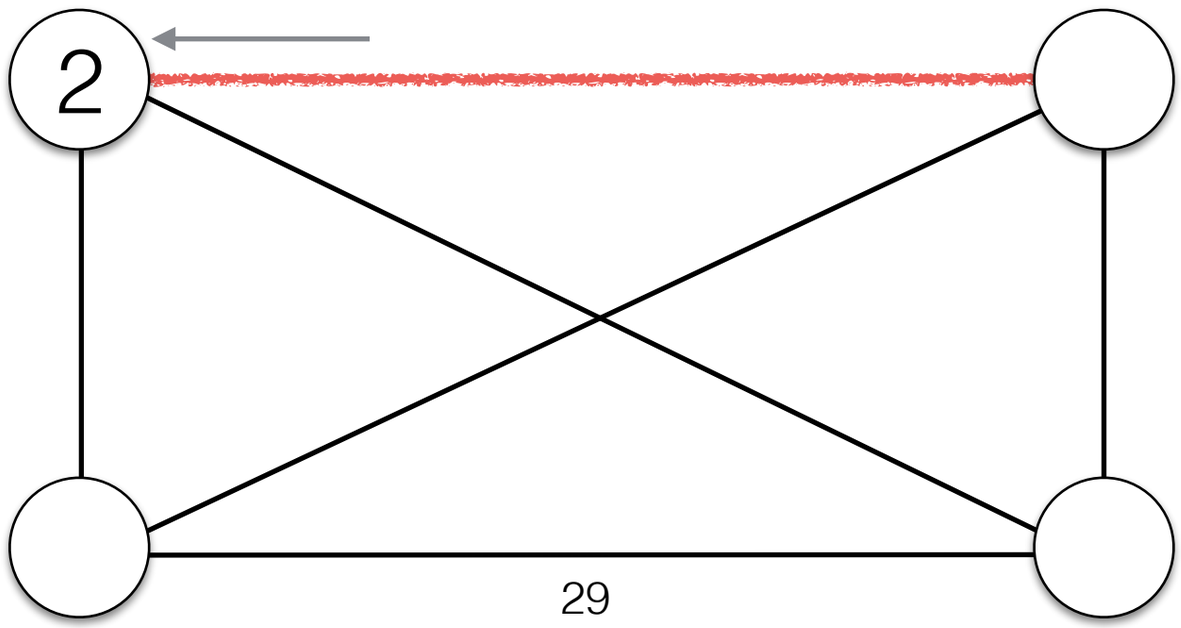
Necessary Condition for Euler Circuits

- Observation:
  - Once we enter v, we need another edge to leave it.
  - If v has an odd number of edges, the last time we enter v, we will be stuck!



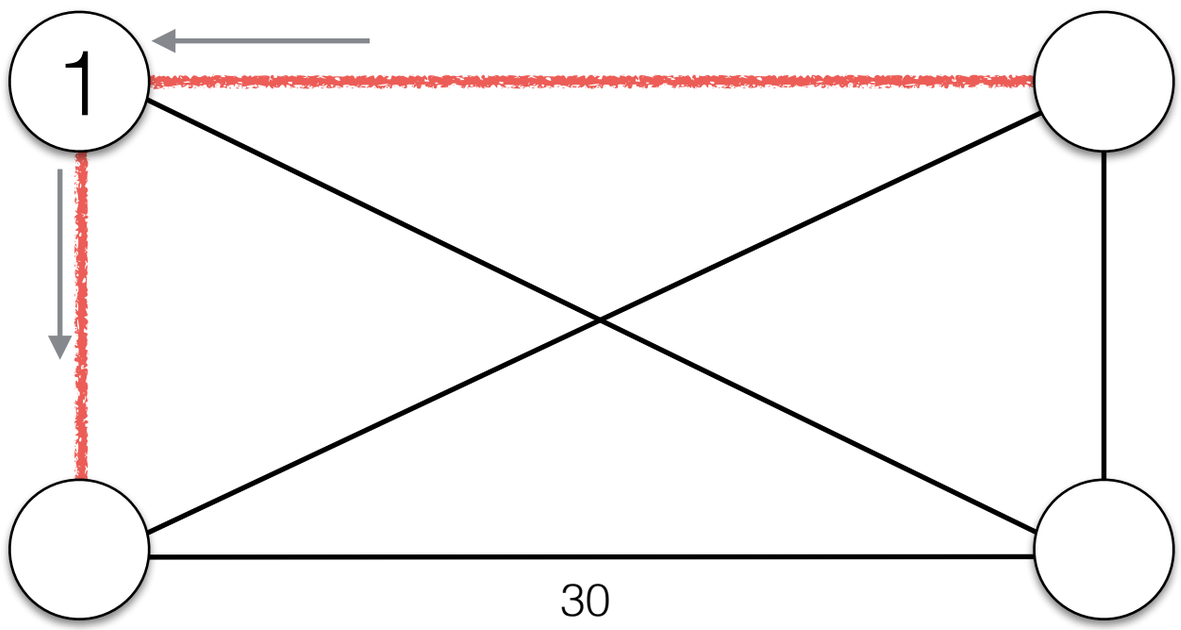
Necessary Condition for Euler Circuits

- Observation:
  - Once we enter v, we need another edge to leave it.
  - If v has an odd number of edges, the last time we enter v, we will be stuck!



Necessary Condition for Euler Circuits

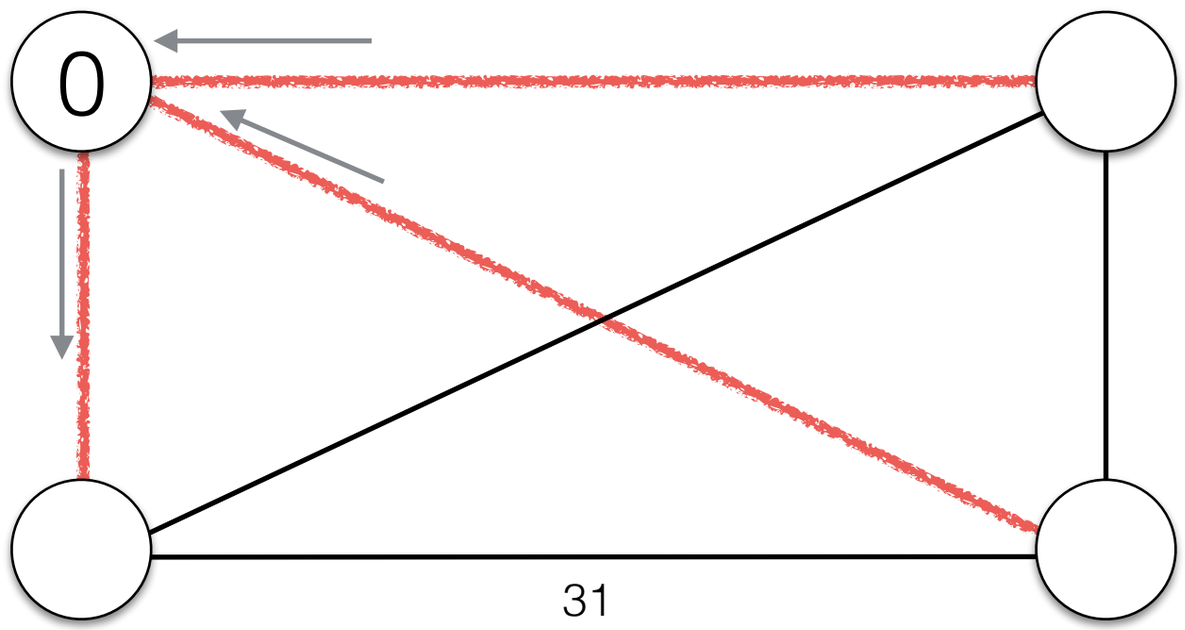
- Observation:
  - Once we enter v, we need another edge to leave it.
  - If v has an odd number of edges, the last time we enter v, we will be stuck!



Necessary Condition for Euler Circuits

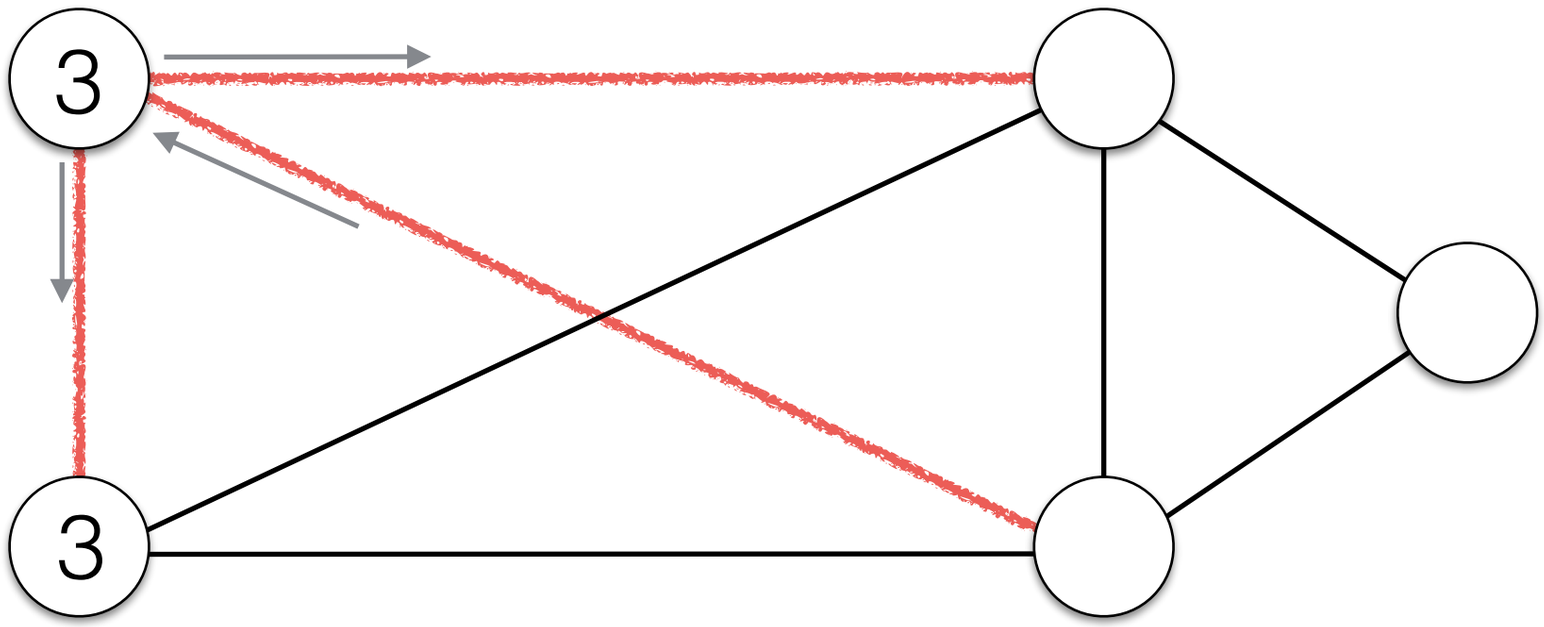
- Observation:
  - Once we enter v, we need another edge to leave it.
  - If v has an odd number of edges, the last time we enter v, we will be stuck!

- For an Euler Circuit to exist in a graph, all vertices need to have even degree (even number of edges).



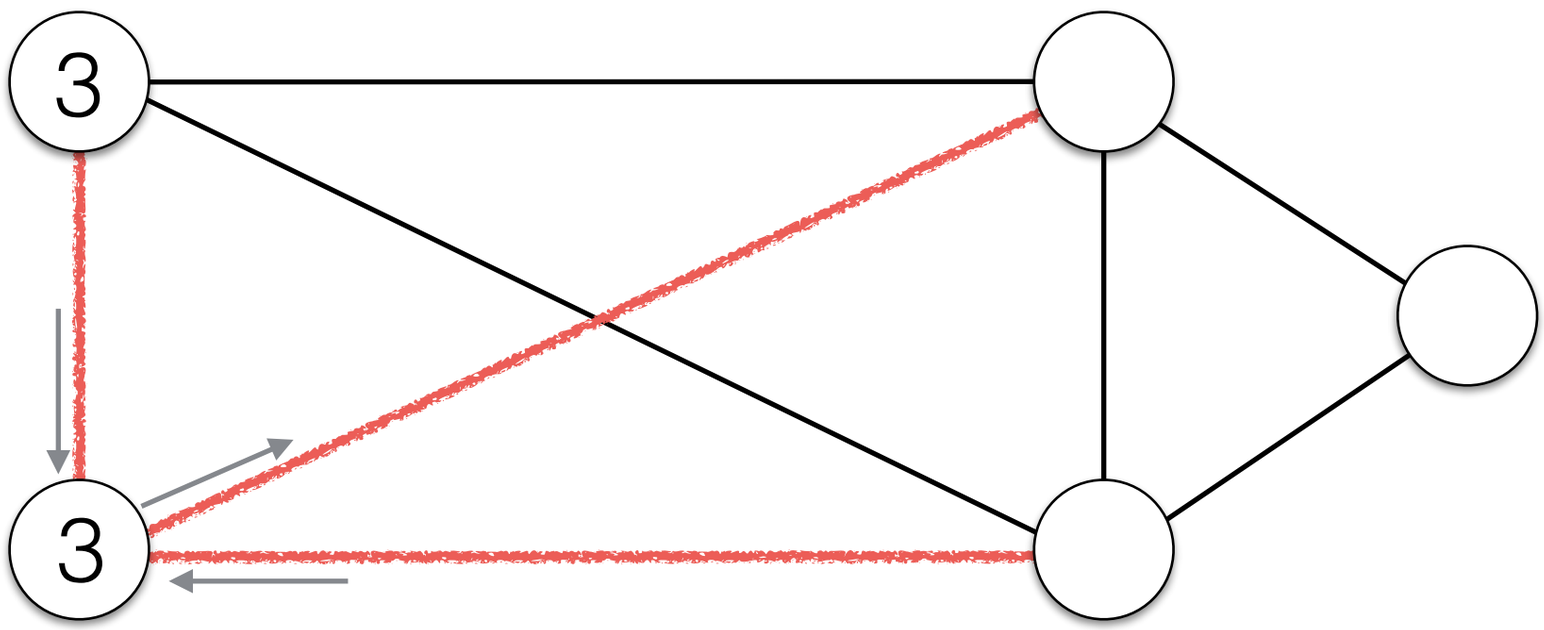
Necessary Condition for Euler Paths

- For an Euler Path to exist in a graph, exactly 0 or 2 vertices must have odd degree.
    - Start with one of the odd vertices.
    - End in the other one.



Necessary Condition for Euler Paths

- For an Euler Path to exist in a graph, exactly 0 or 2 vertices must have odd degree.
    - Start with one of the odd vertices.
    - End in the other one.



Conditions for Euler Paths and Circuits

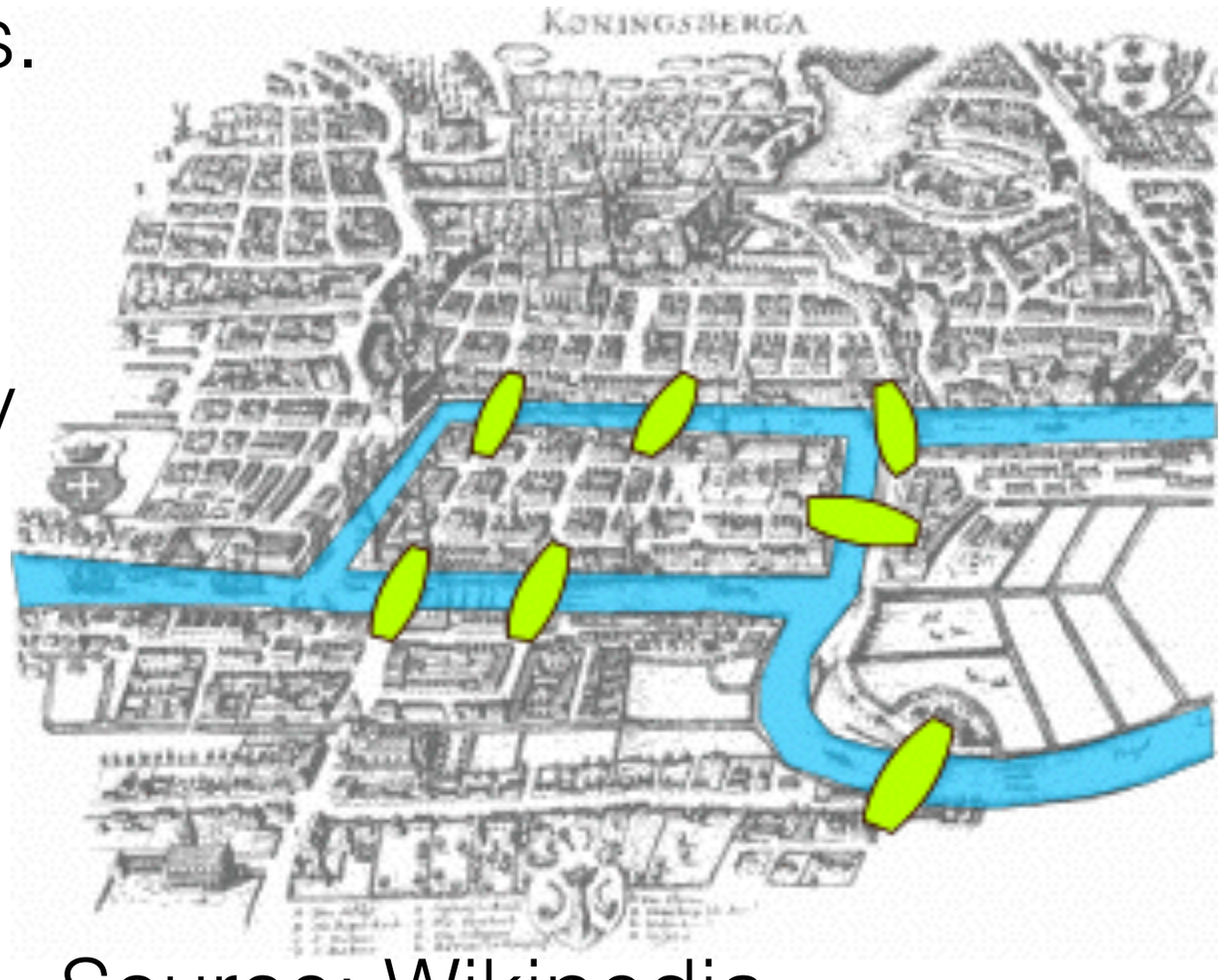
- For an Euler Circuit to exist in a graph, all vertices need to have even degree (even number of edges).
- For an Euler Path to exist in a graph, exactly 0 or 2 vertices need to have odd degree.

- These conditions are also sufficient!
  (i.e. every graph that contains only vertices of even degree has an Euler circuit). *(Hierholzer, 1873)*

Seven Bridges of Königsberg

Leonhard Euler, 1735

The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands which were connected to each other and the mainland by seven bridges.

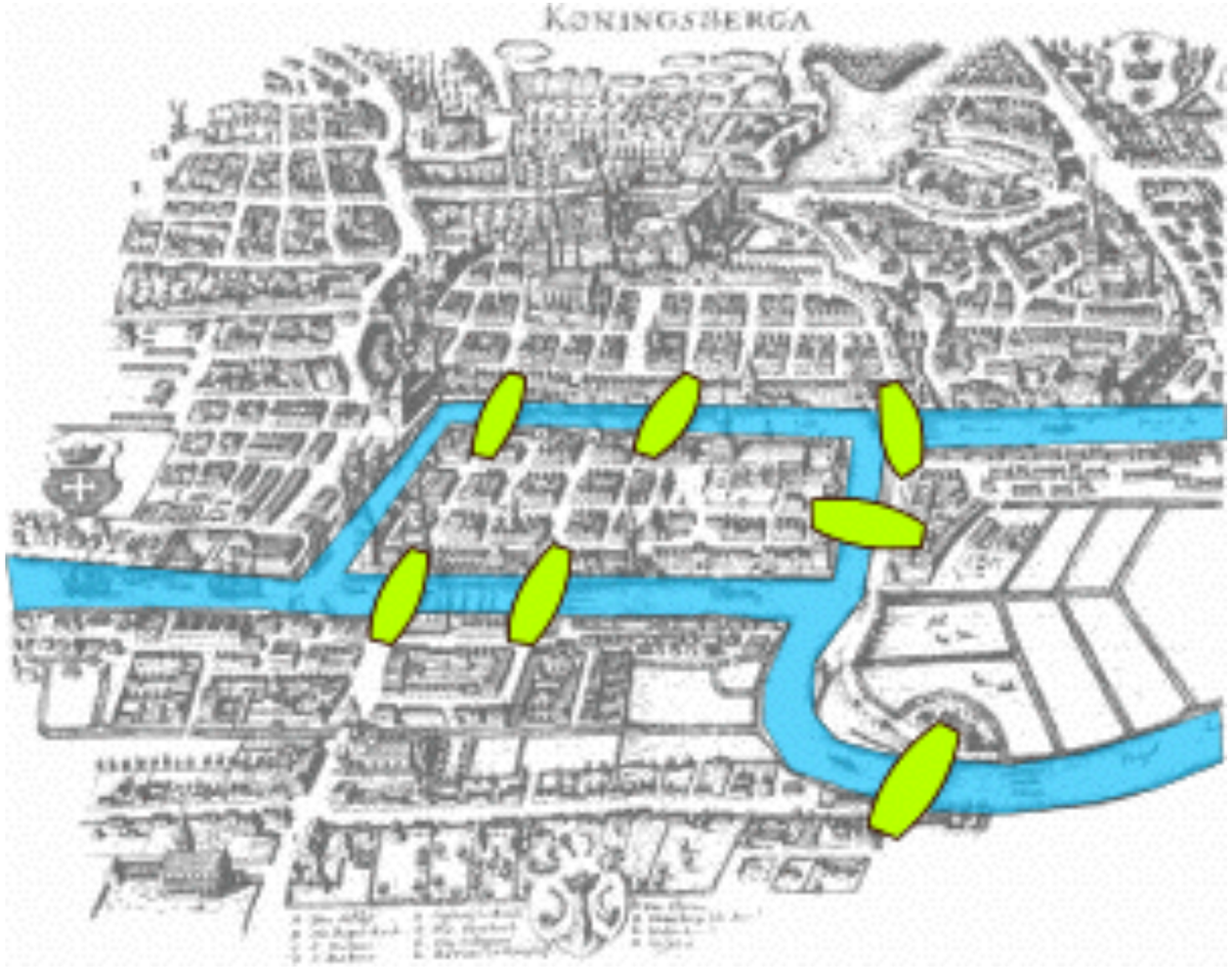


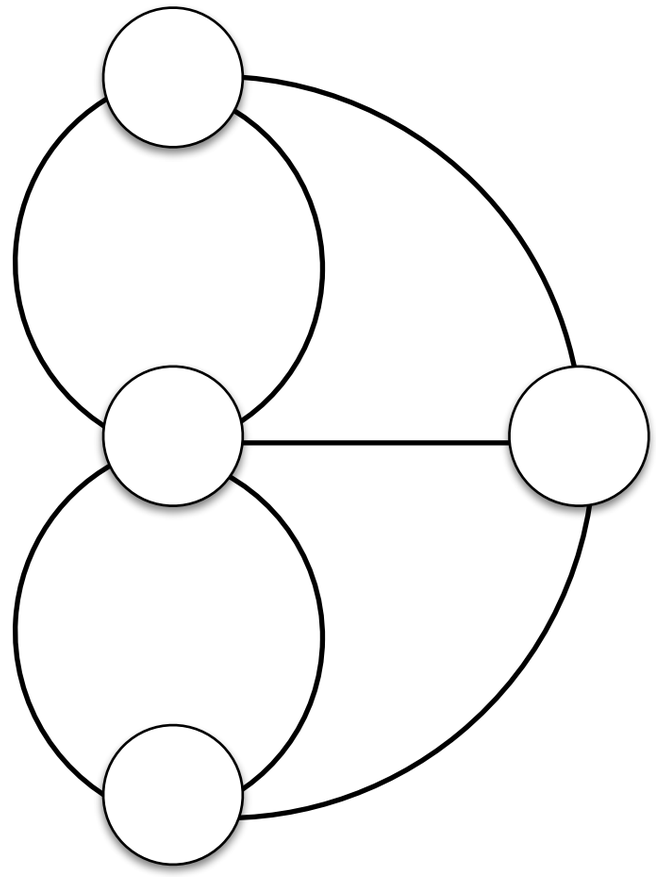
Euler's Problem:
Find a walk through the city that crosses every bridge exactly once!

Source: Wikipedia

Seven Bridges of Königsberg

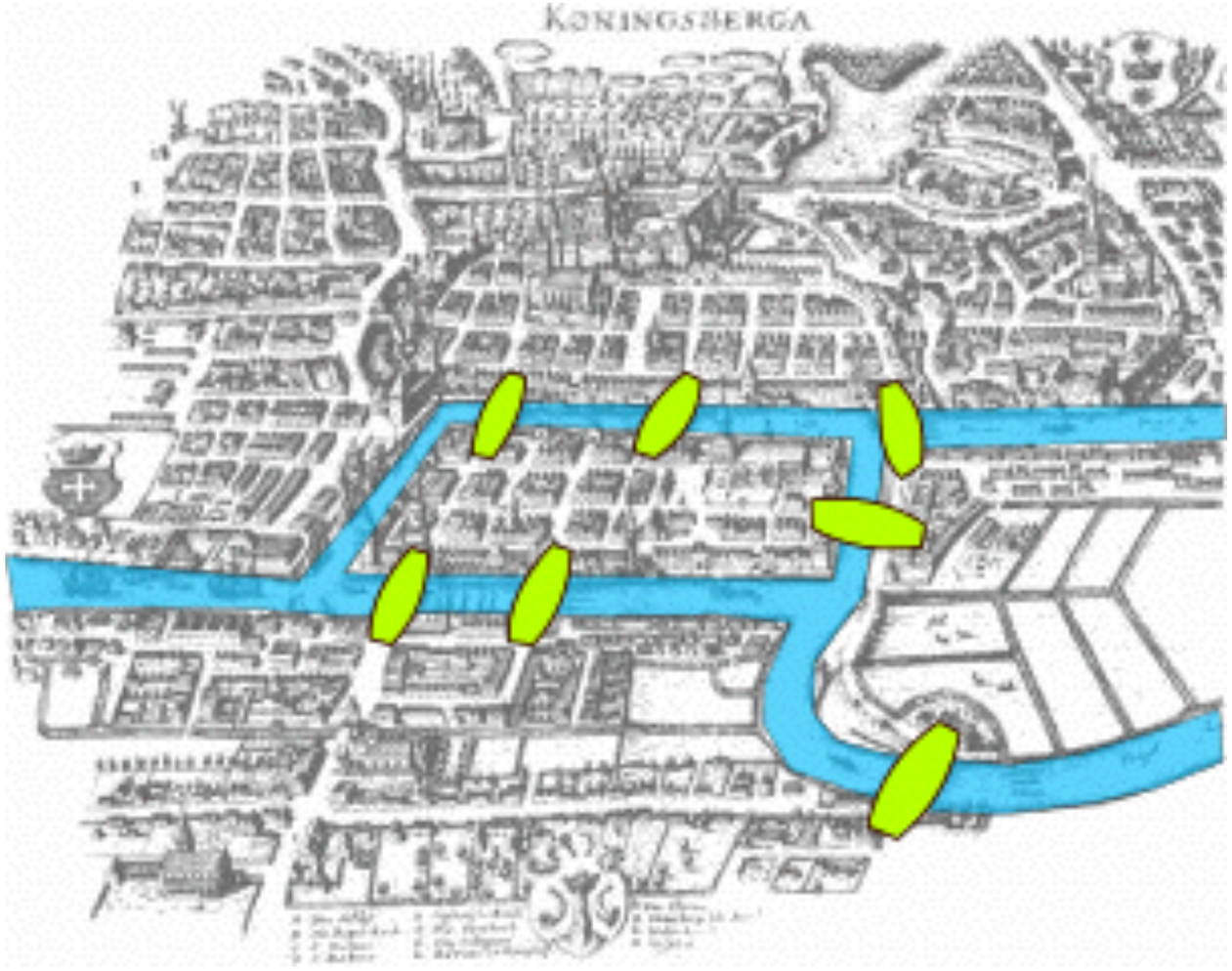
Leonhard Euler, 1735

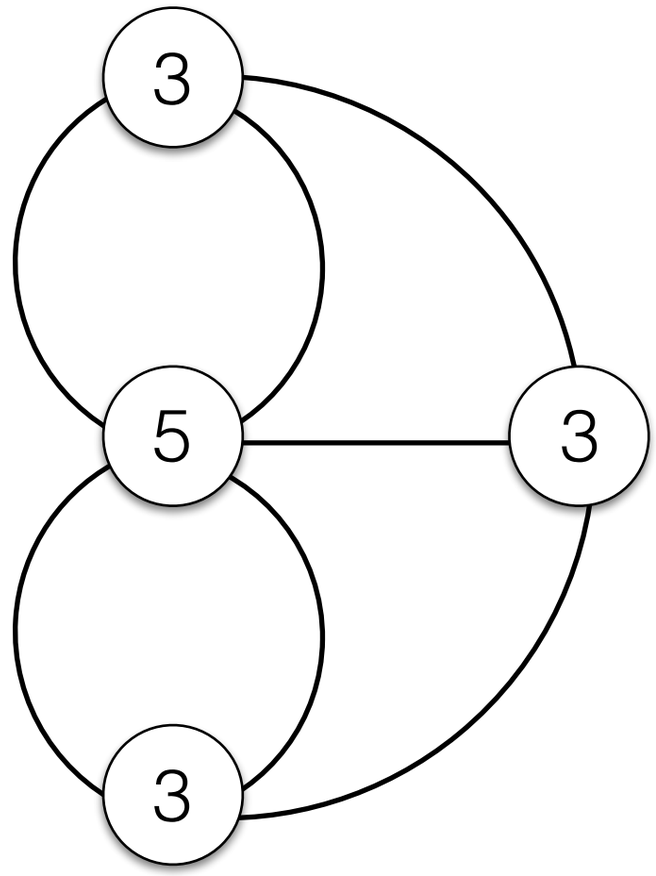




Seven Bridges of Königsberg

Leonhard Euler, 1735



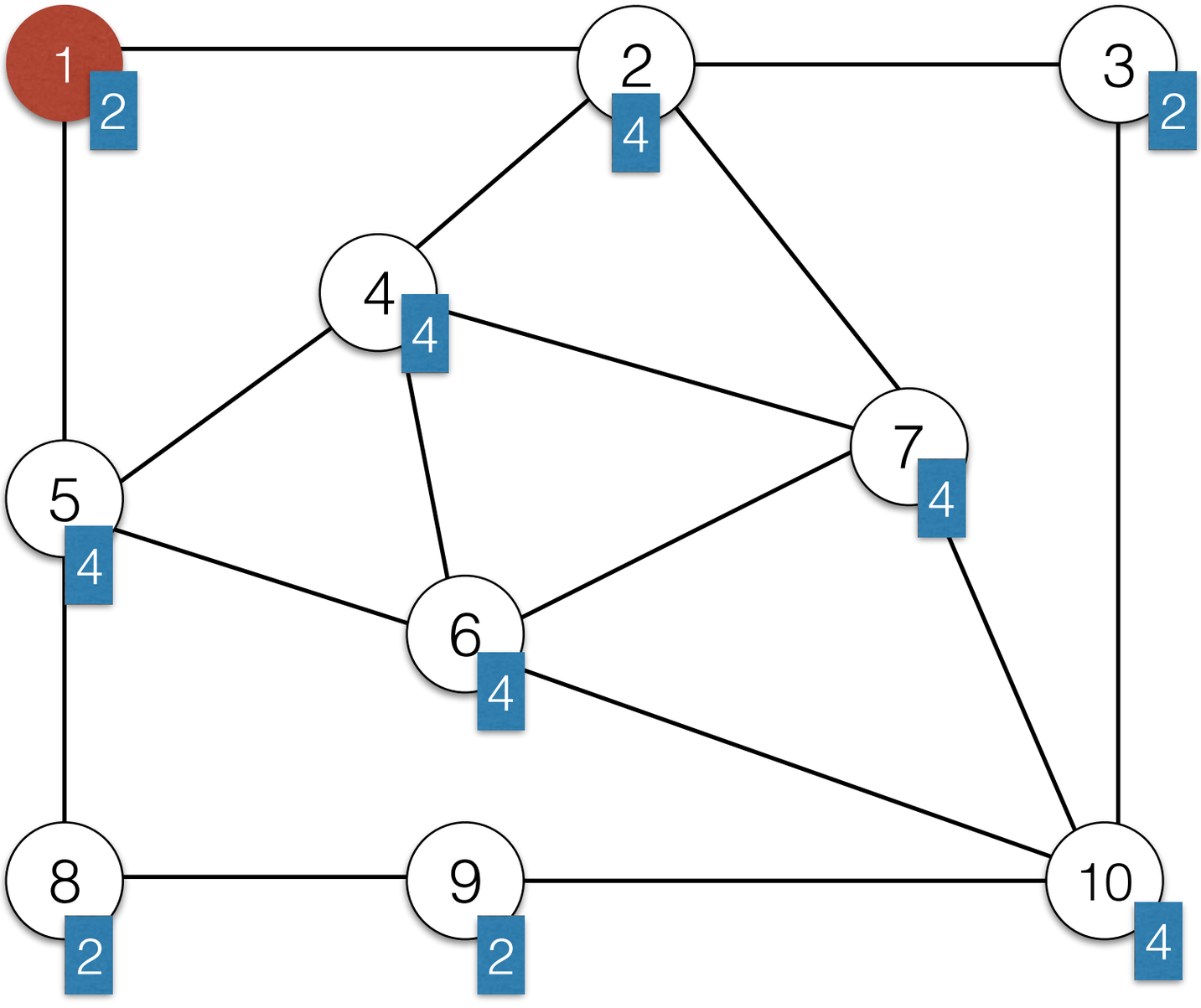


There is no Euler Path in this graph.

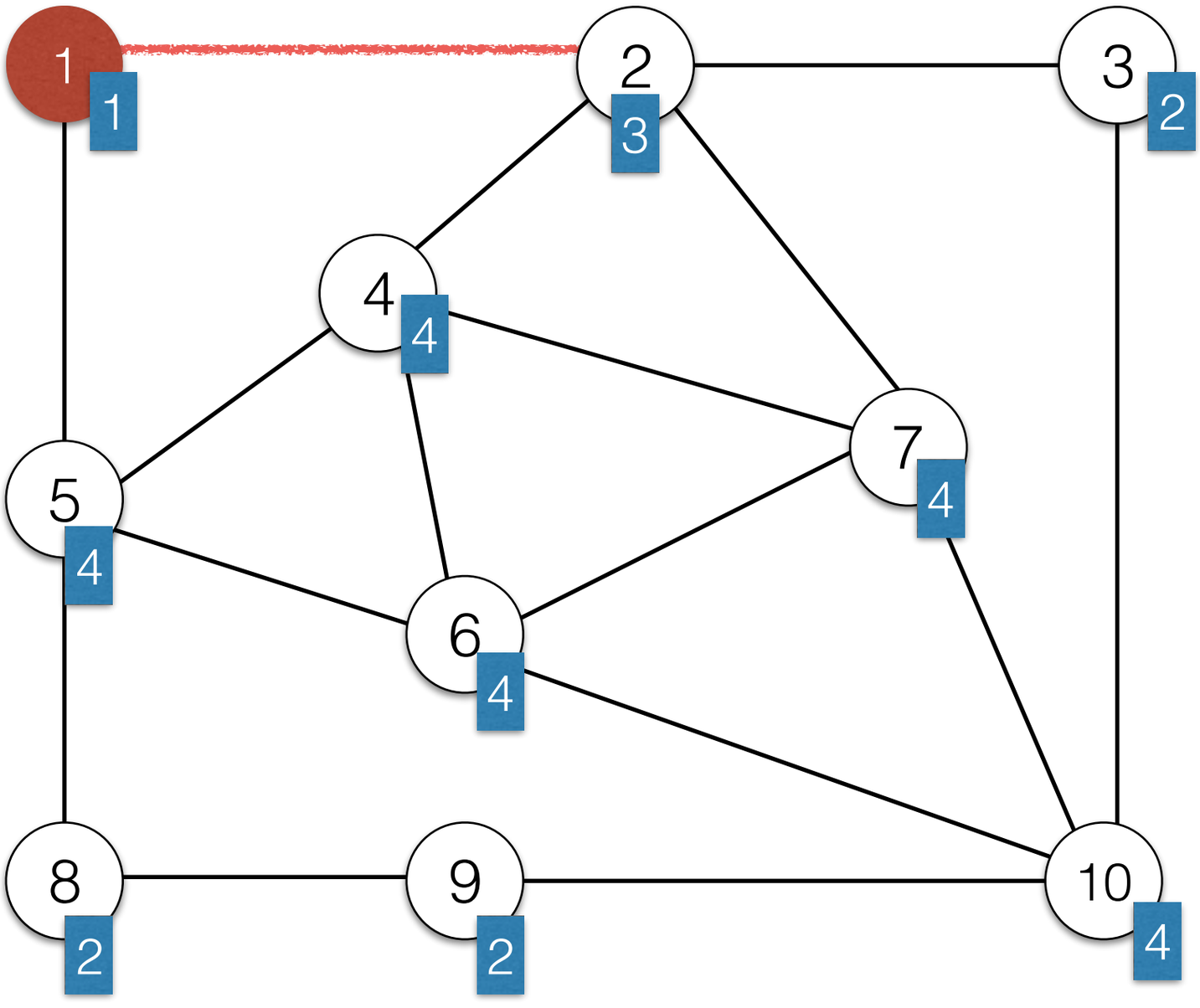
Finding Euler Circuits

- Start with any vertex *s*. First, using DFS find *any* circuit starting and ending in *s*. Mark all edges on the circuit as visited.
    - Because all vertices have even degree, we are guaranteed to not get stuck before we arrive at *s* again.
- While there are still edges in the graph that are not market visited:
    - Find the first vertex v on the circuit that has unvisited edges.
    - Find a circuit starting in v and *splice* this path into the first circuit

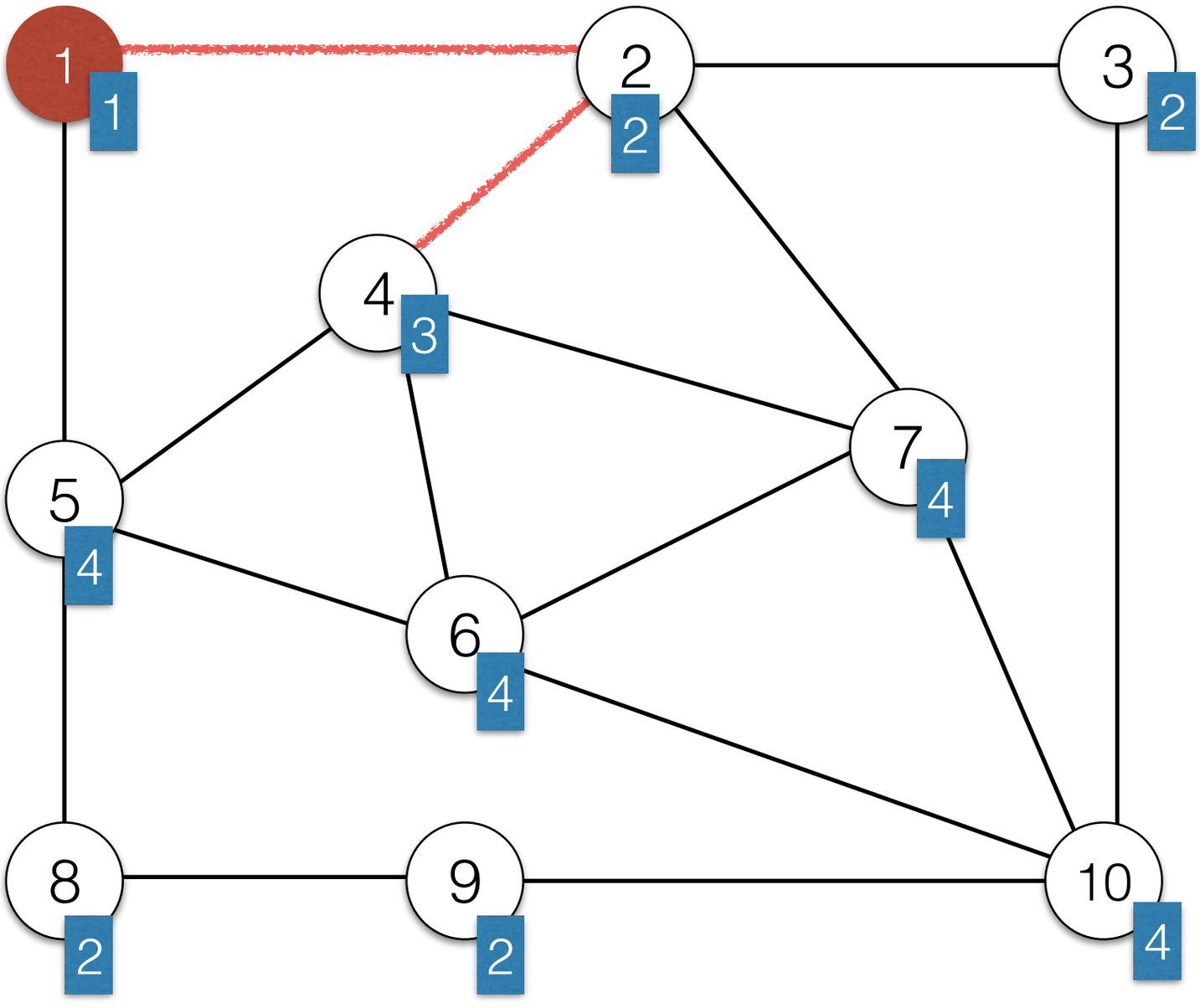
Finding Euler Circuits



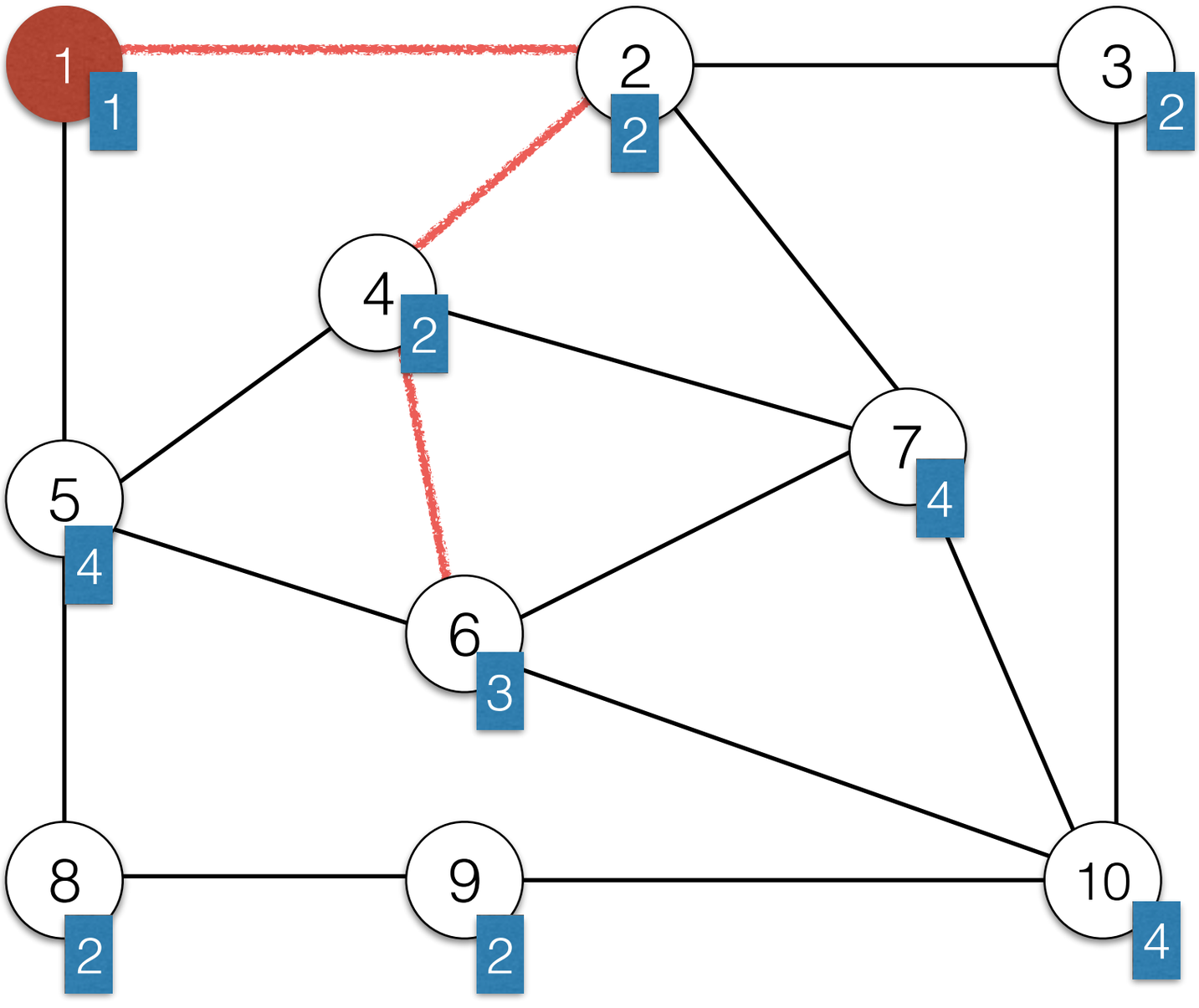
Finding Euler Circuits



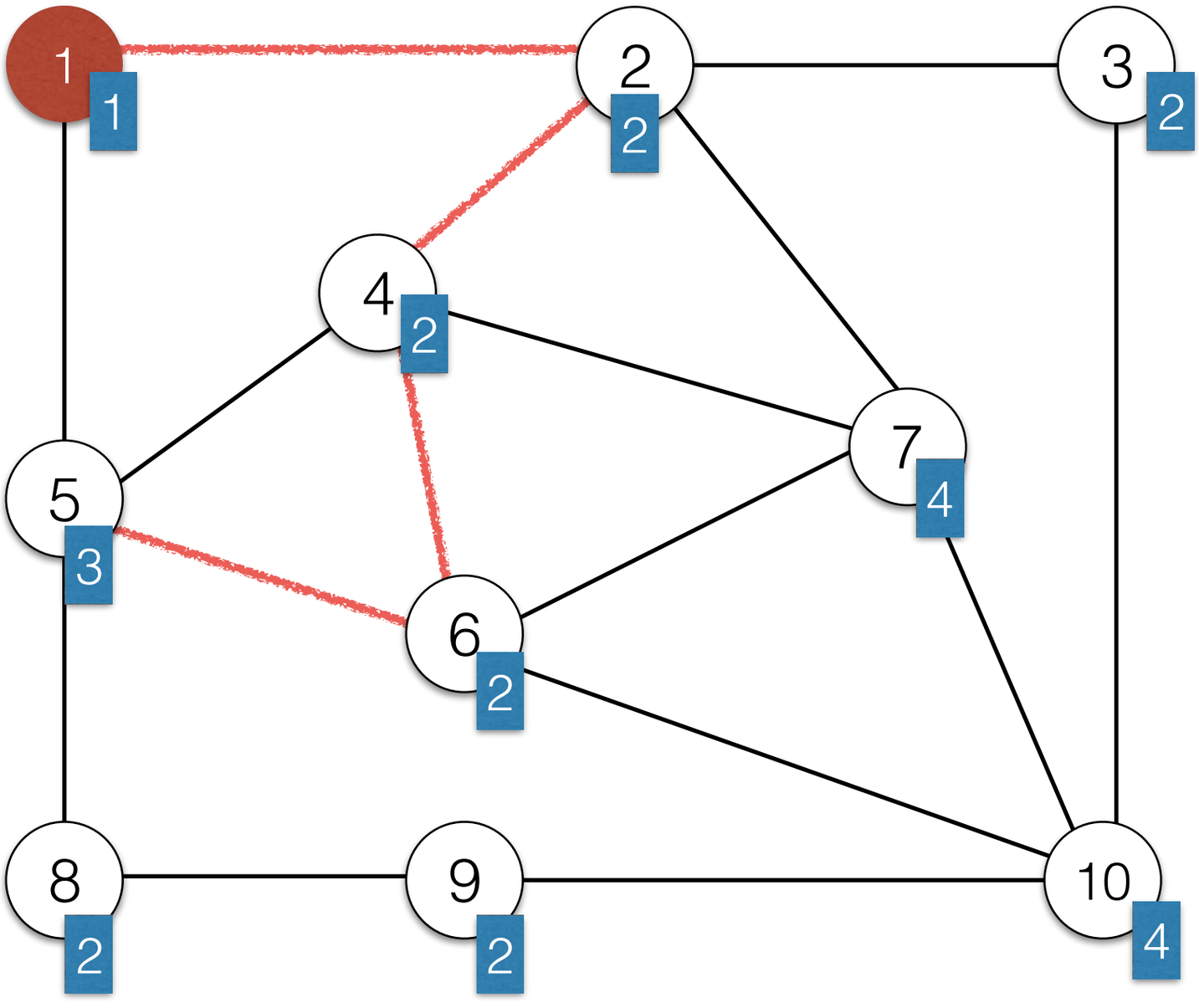
Finding Euler Circuits



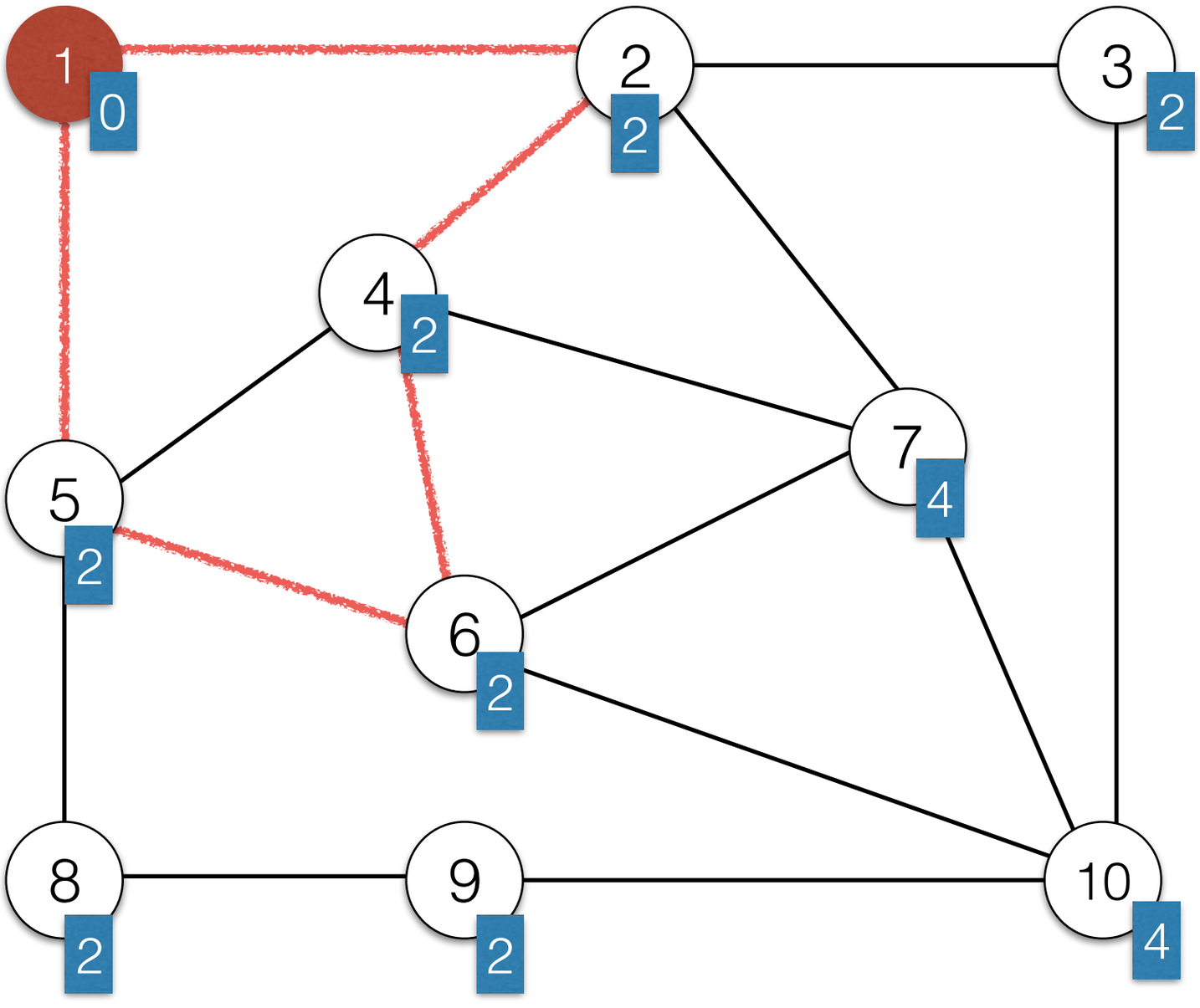
Finding Euler Circuits



Finding Euler Circuits

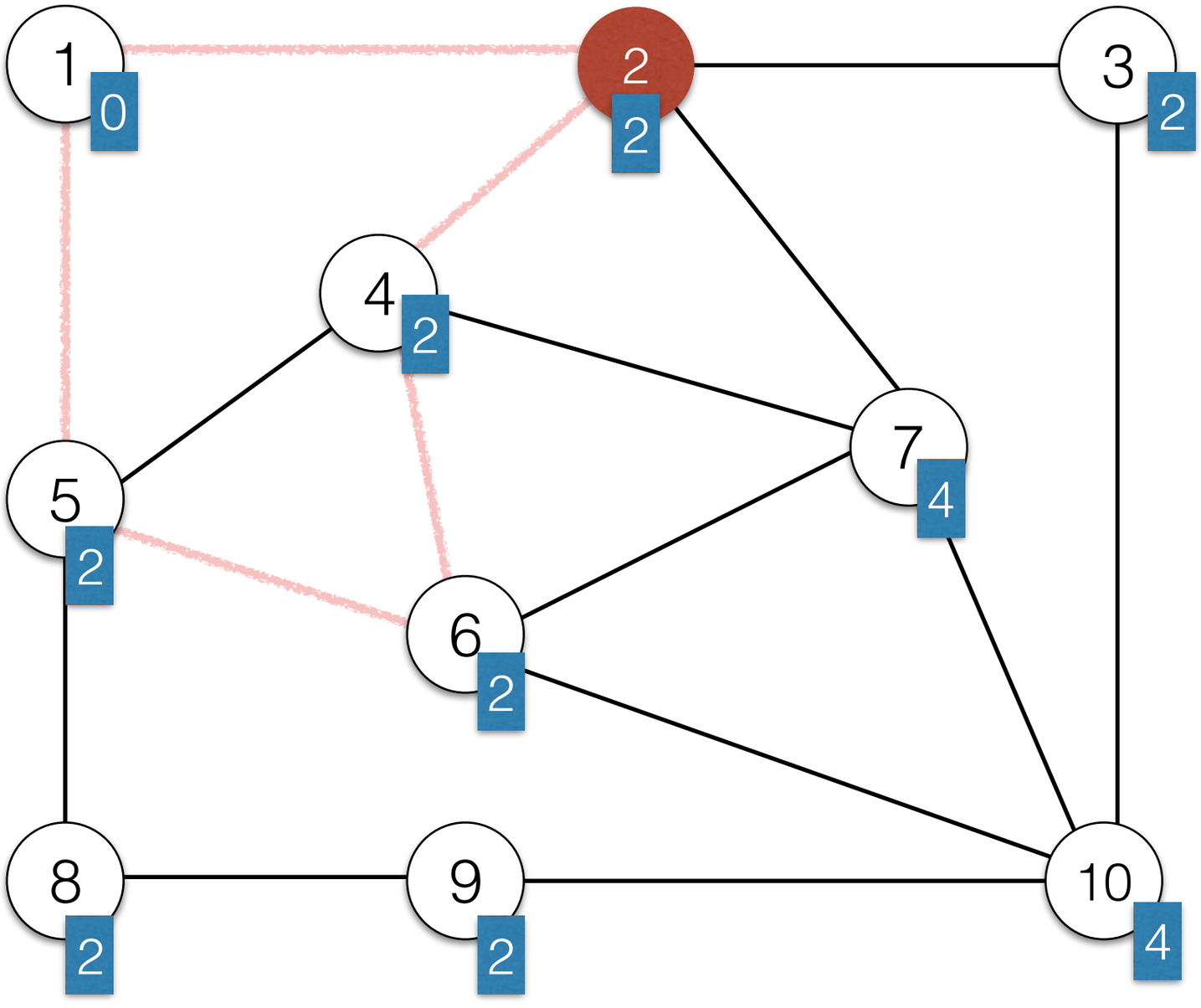


Finding Euler Circuits



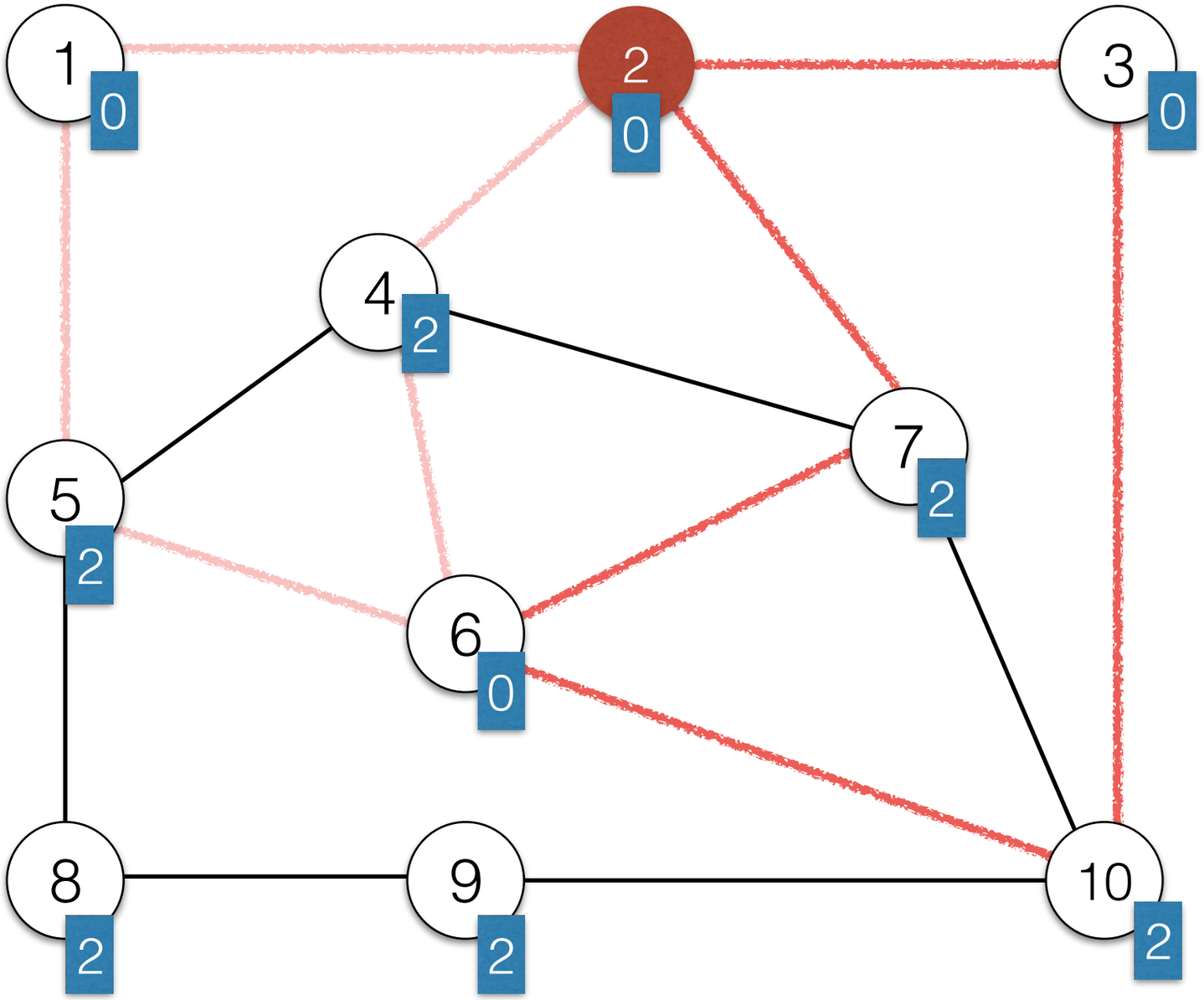
1 2 4 6 5 1

Finding Euler Circuits



1 2 4 6 5 1

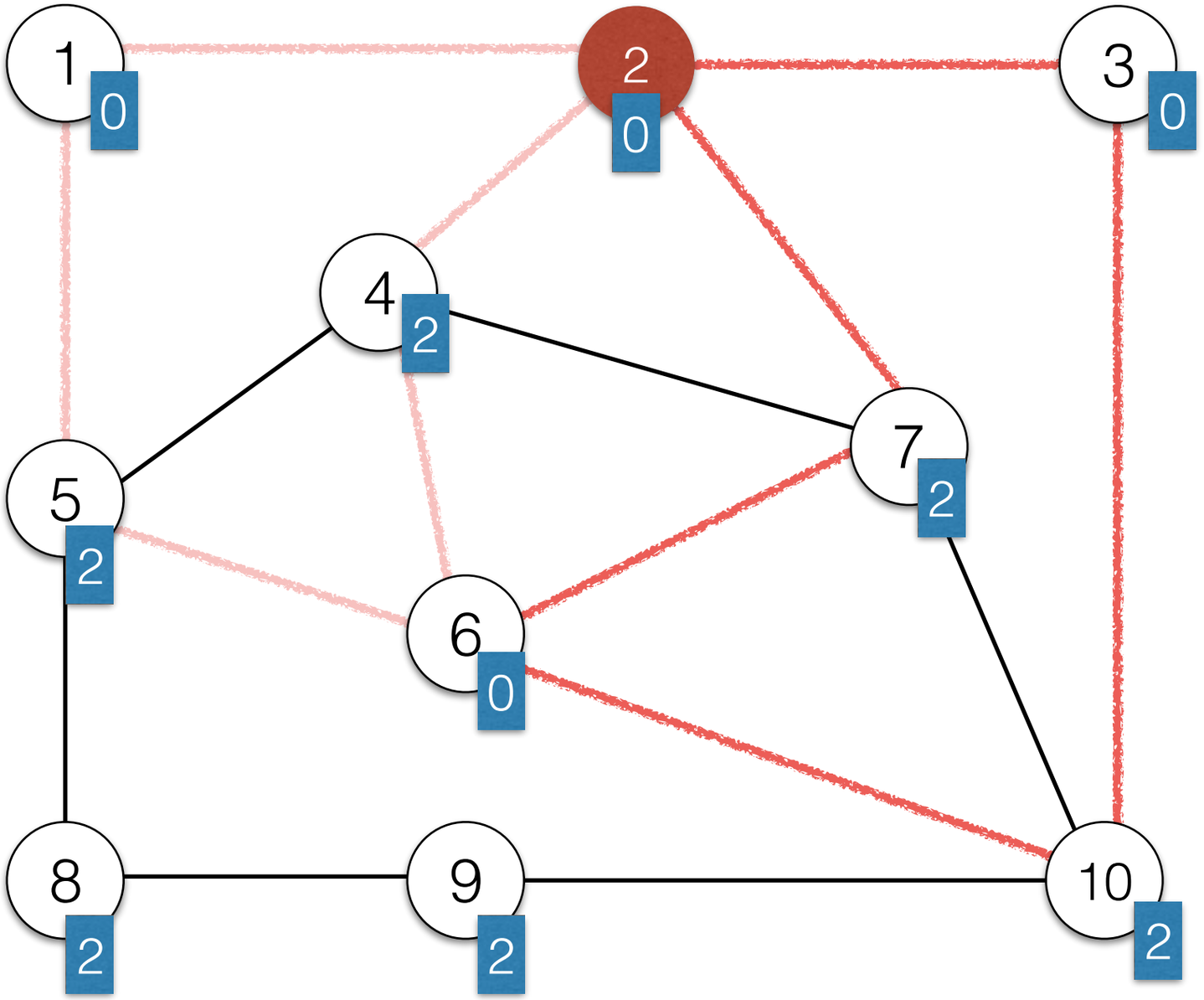
Finding Euler Circuits



2 7 6 10 3 2

1 2 4 6 5 1

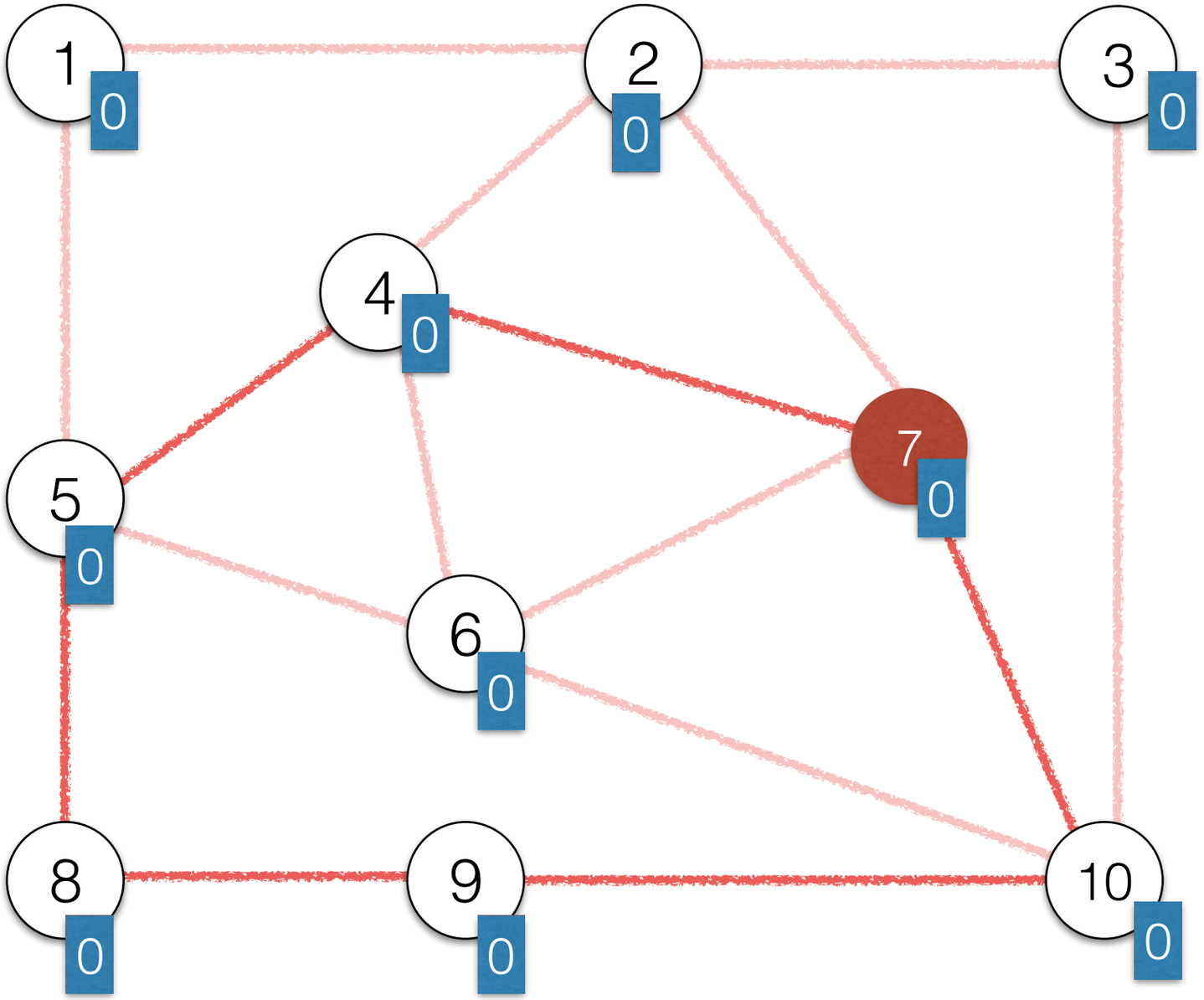
Finding Euler Circuits



2 7 6 10 3 2

1 4 6 5 1

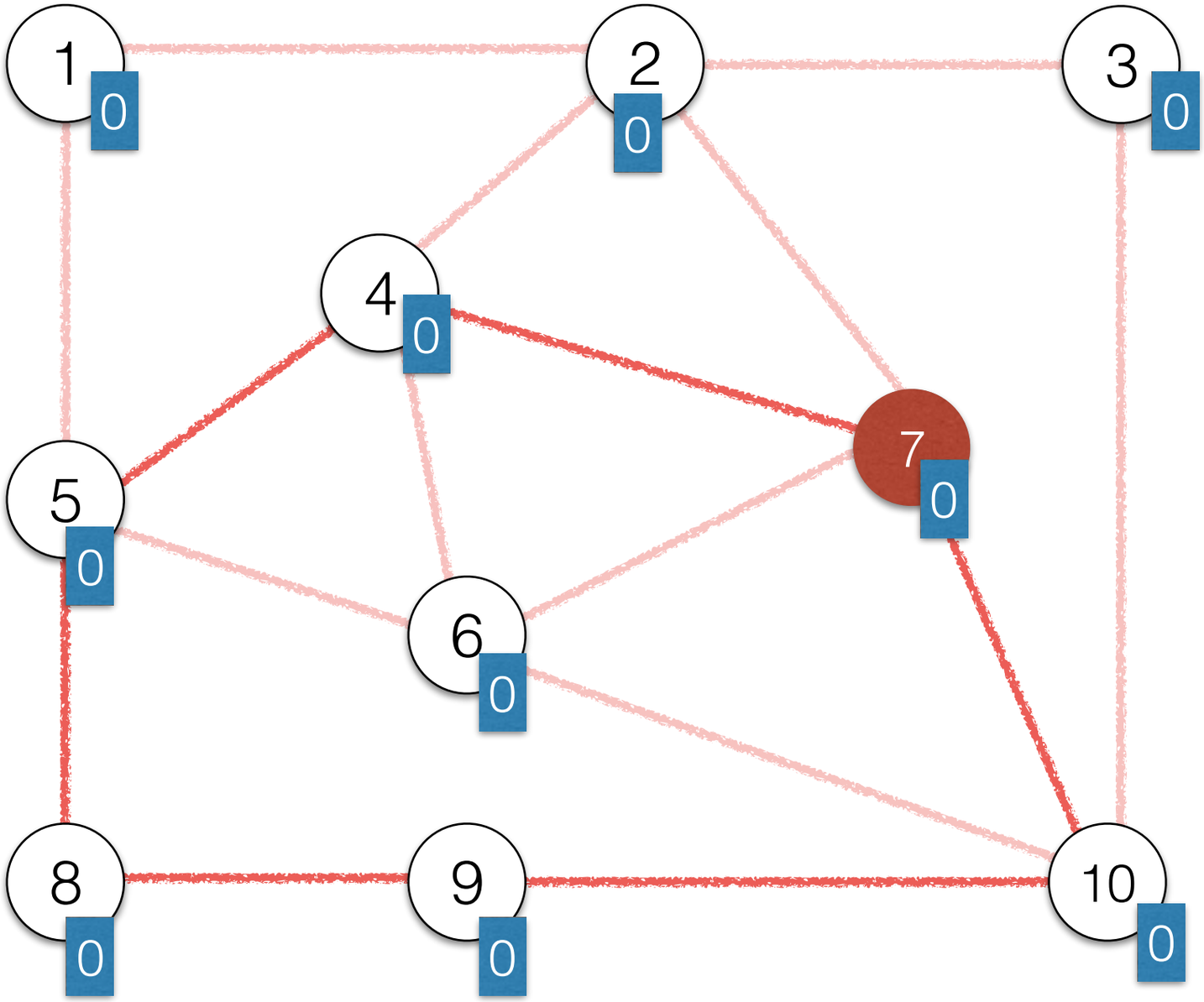
Finding Euler Circuits



7 4 5 8 9 10 7

1 2 7 6 10 3 2 4 6 5 1

Finding Euler Circuits

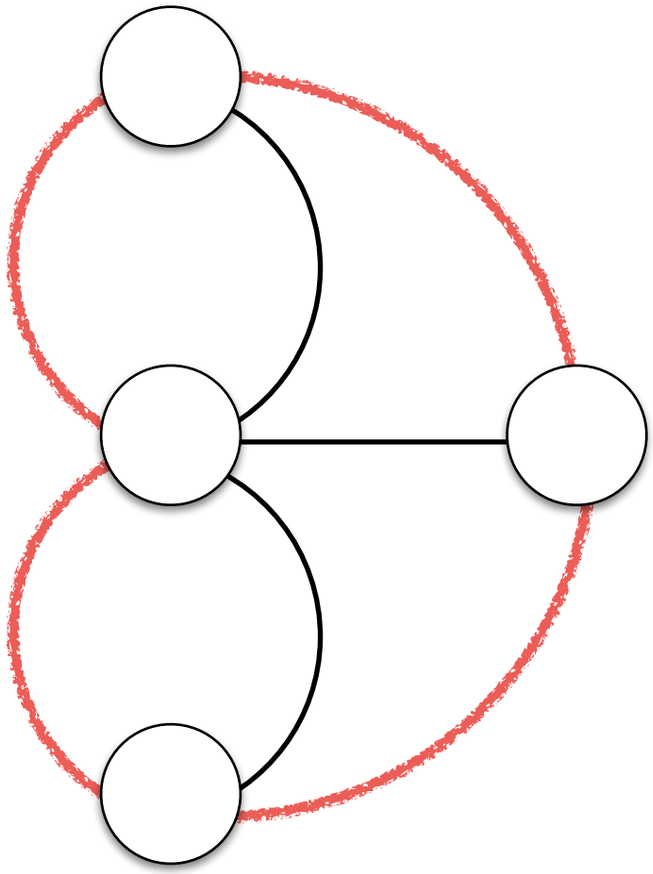


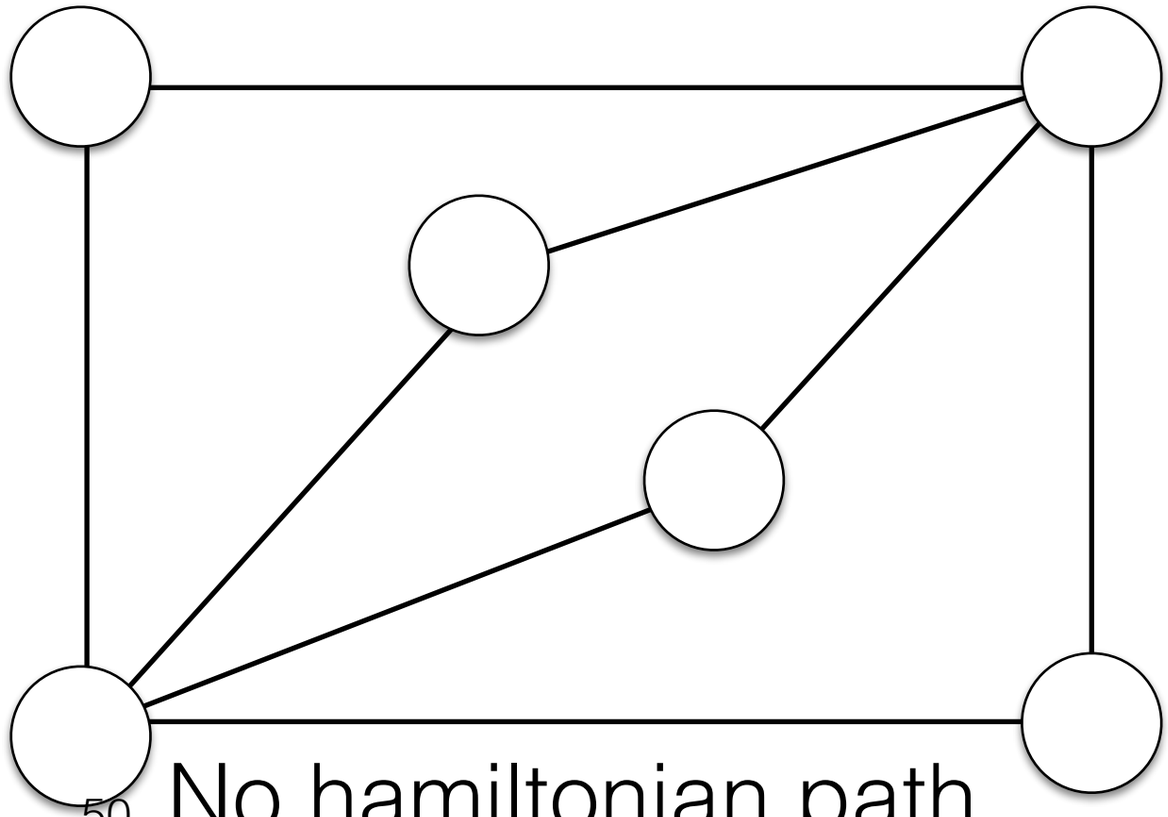
7 4 5 8 9 10 7

1 2 6 10 3 2 4 6 5 1

Hamiltonian Cycle

- A Hamiltonian Path is a path through an undirected graph that visits ***every vertex*** exactly once (except that the first and last vertex may be the same).
- A Hamiltonian Cycle is a Hamiltonian Path that starts and ends in the same node.

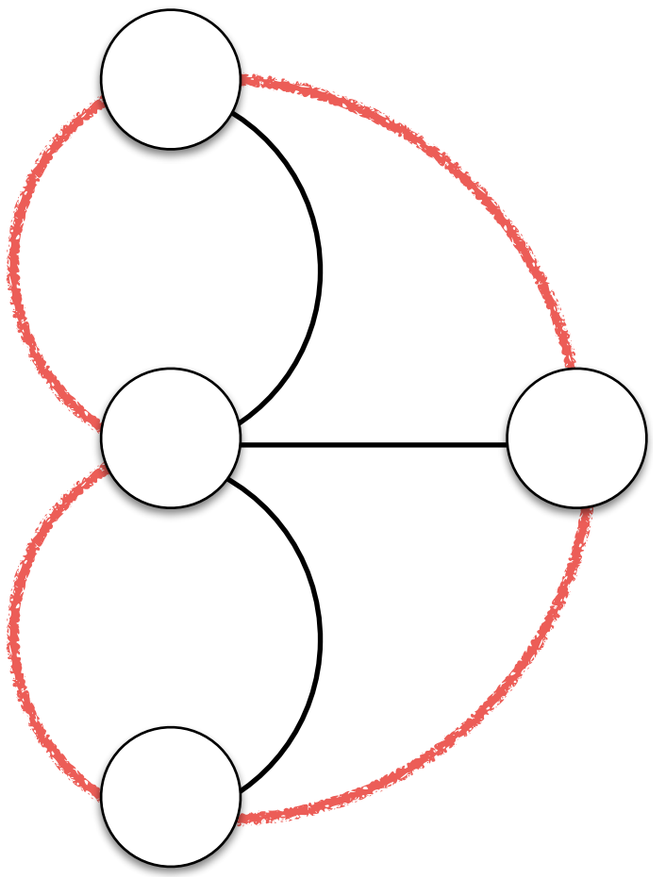


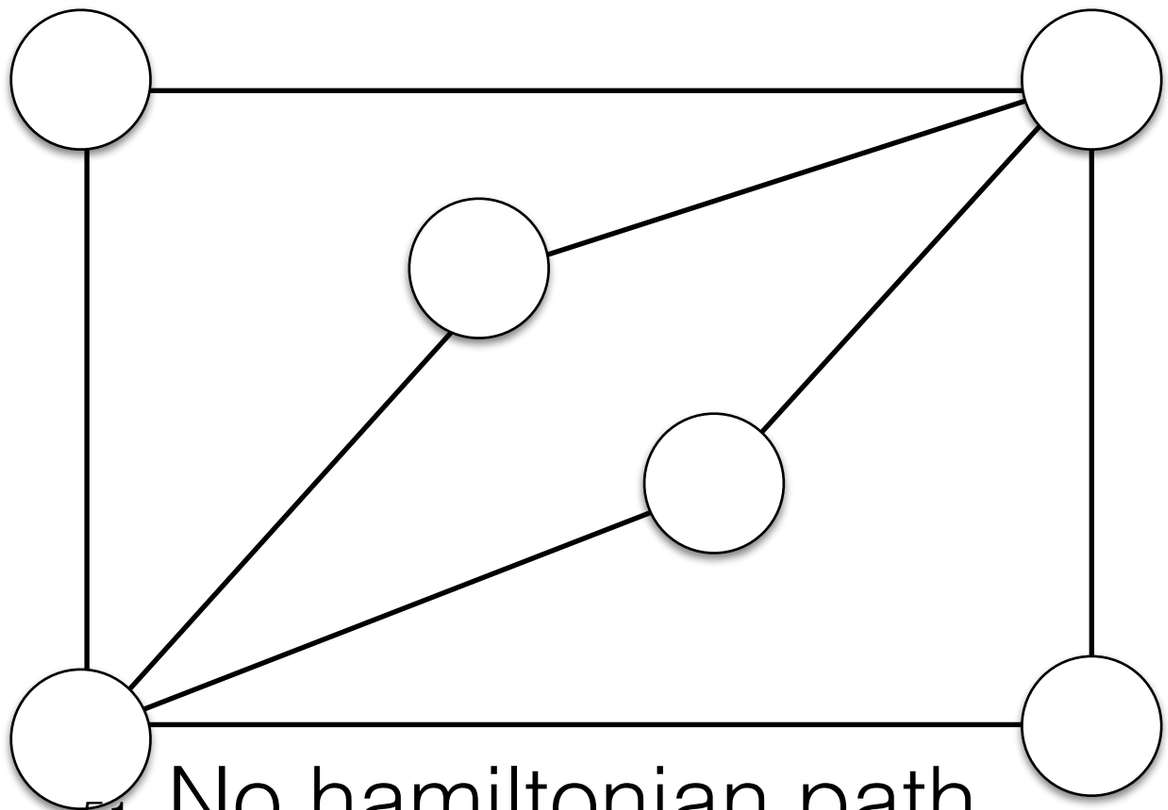


No hamiltonian path.

Hamiltonian Cycle

- We can check if a graph contains an Euler Cycle in linear time.
- Surprisingly, checking if a graph contains a Hamiltonian Path/Cycle is much harder!
- No polynomial time solution (i.e. $O(N^k)$) is known.





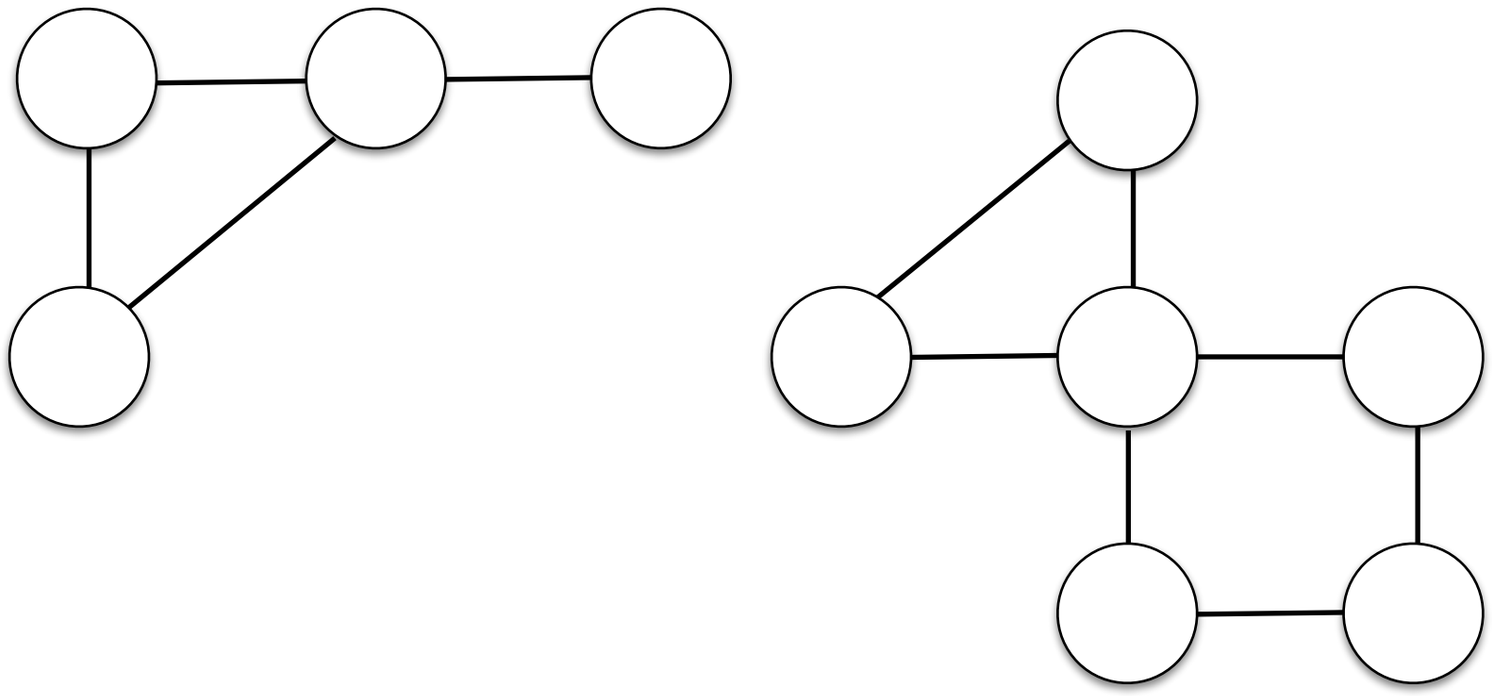
No hamiltonian path.

Contents

- Applications of DFS
- Euler Circuits
- **Biconnectivity in Undirected Graphs.**
- Finding Strongly Connected Components for Directed Graphs.

Connectivity

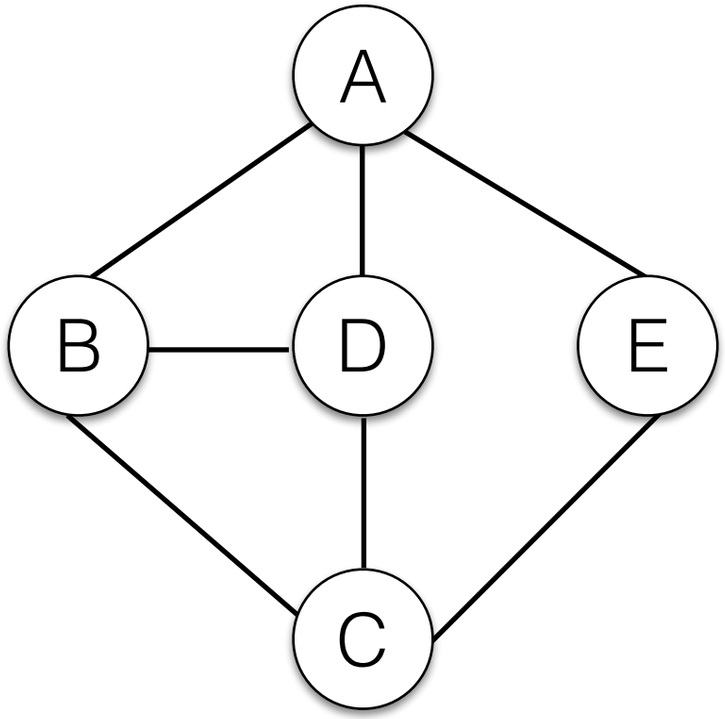
- An undirected graph is **connected** if there is a path from every vertex to every other vertex.
- Test for connectivity: See if DFS can reach all vertices.



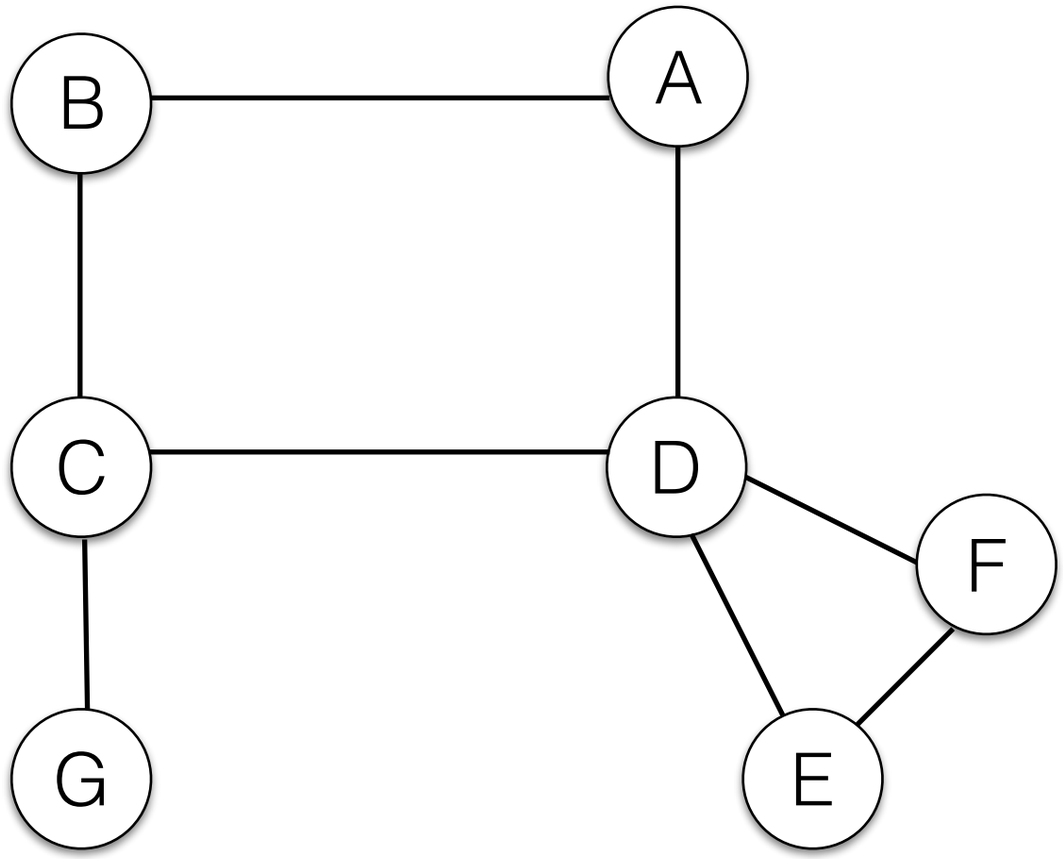
unconnected graph

Biconnectivity

- A graph is biconnected if there is no single vertex *v*, such that removing *v* will disconnect the remaining graph.



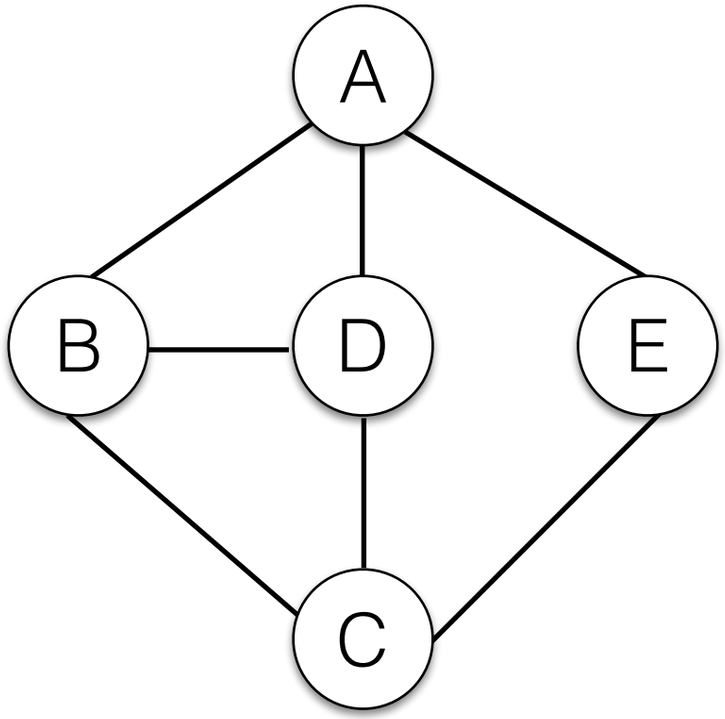
biconnected



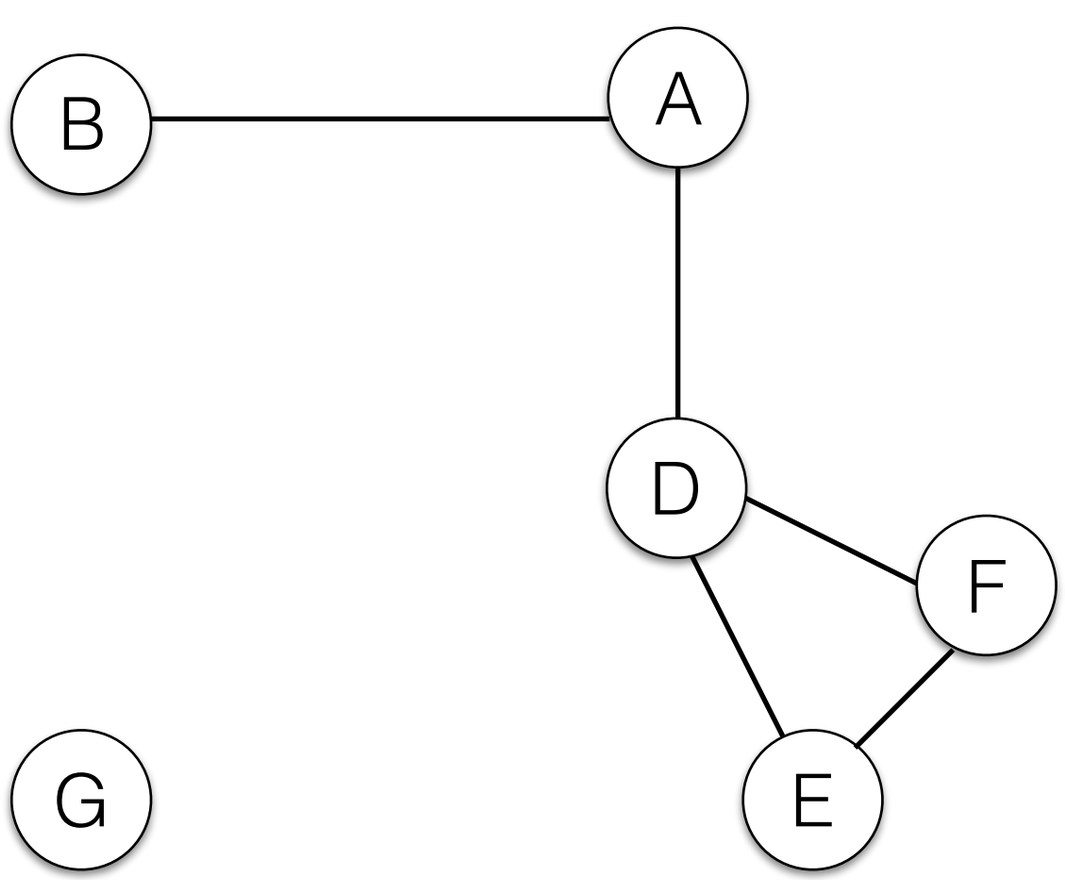
not biconnected

Biconnectivity

- A graph is biconnected if there is no single vertex *v*, such that removing *v* will disconnect the remaining graph.



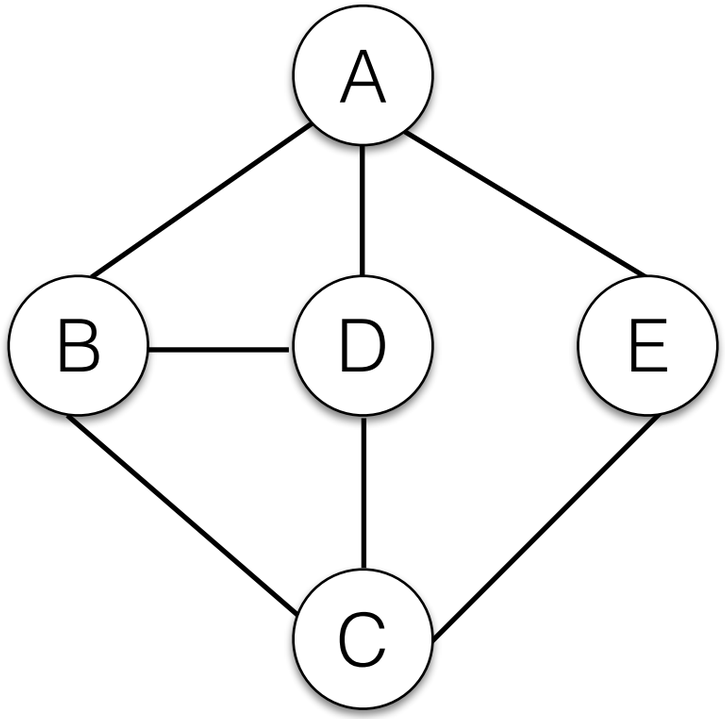
biconnected



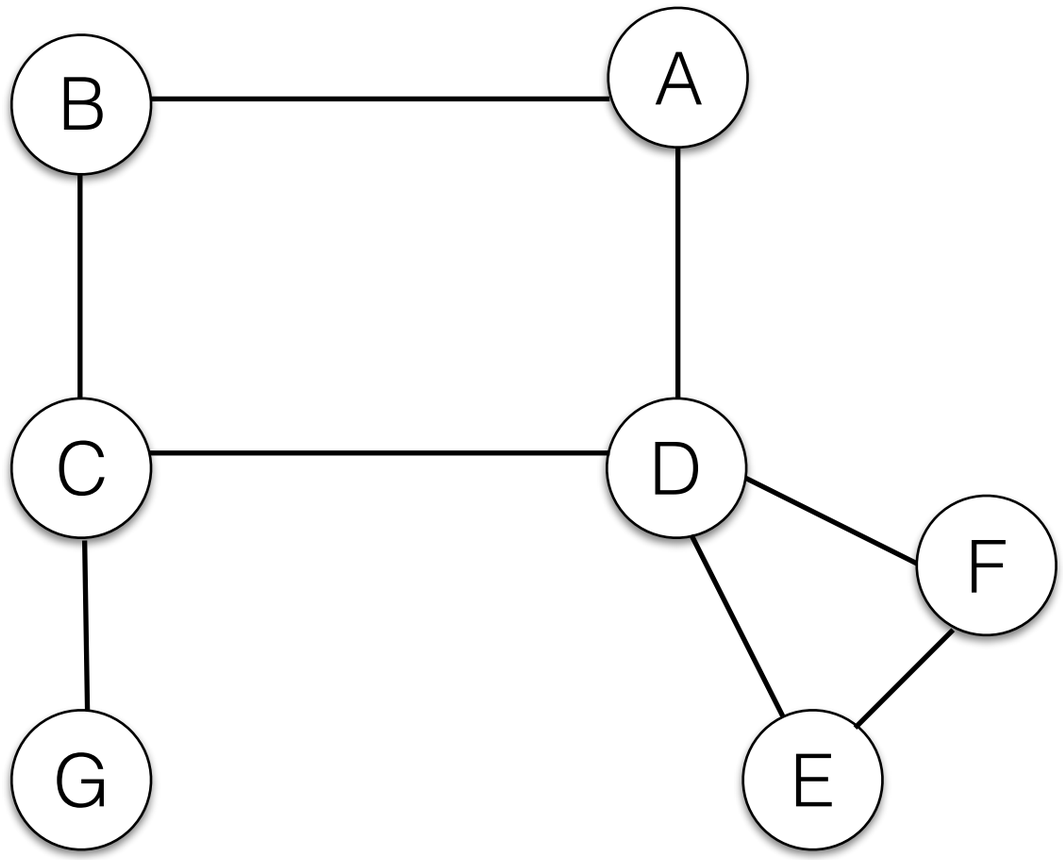
not biconnected

Biconnectivity

- A graph is biconnected if there is no single vertex *v*, such that removing *v* will disconnect the remaining graph.



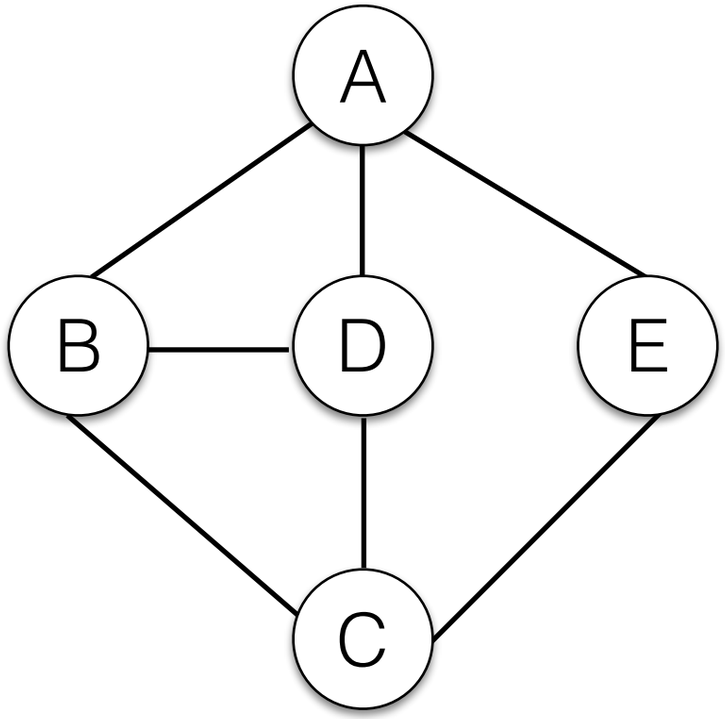
biconnected



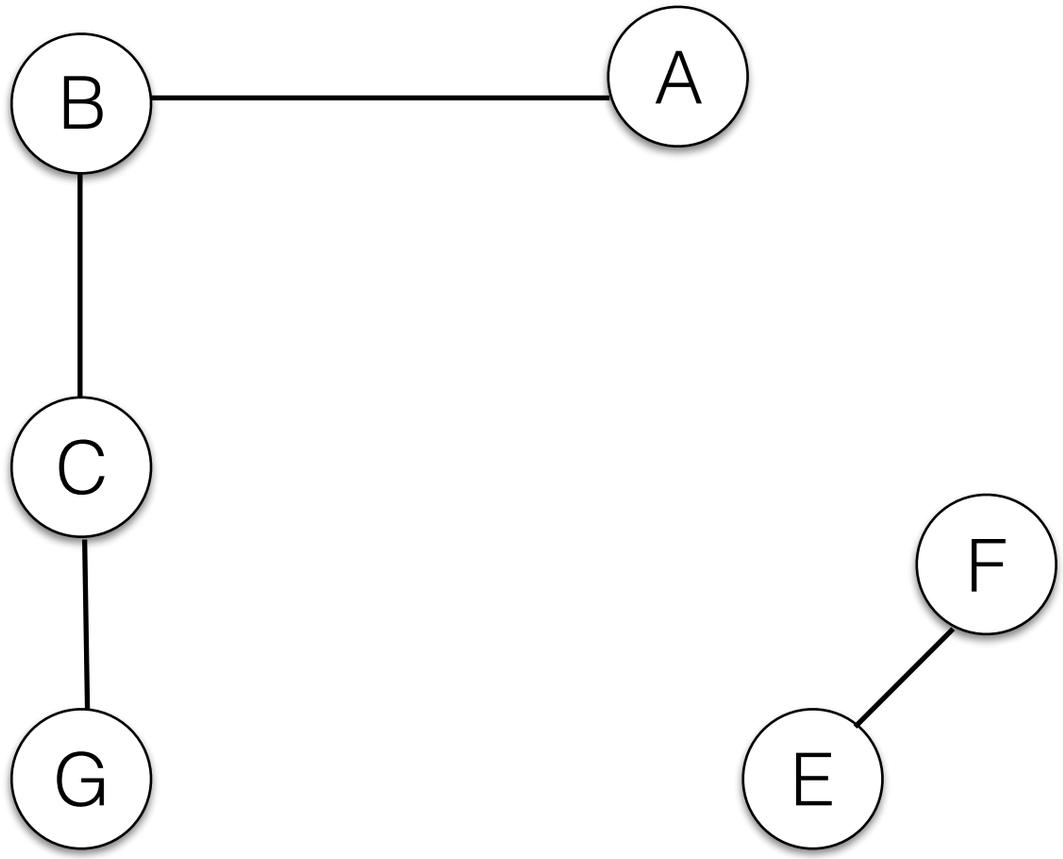
not biconnected

Biconnectivity

- A graph is biconnected if there is no single vertex *v*, such that removing *v* will disconnect the remaining graph.



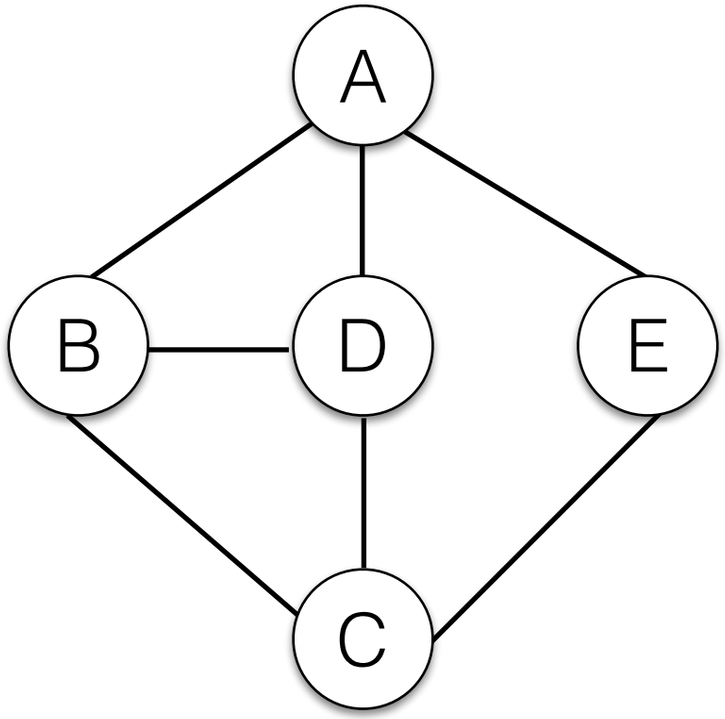
biconnected



not biconnected

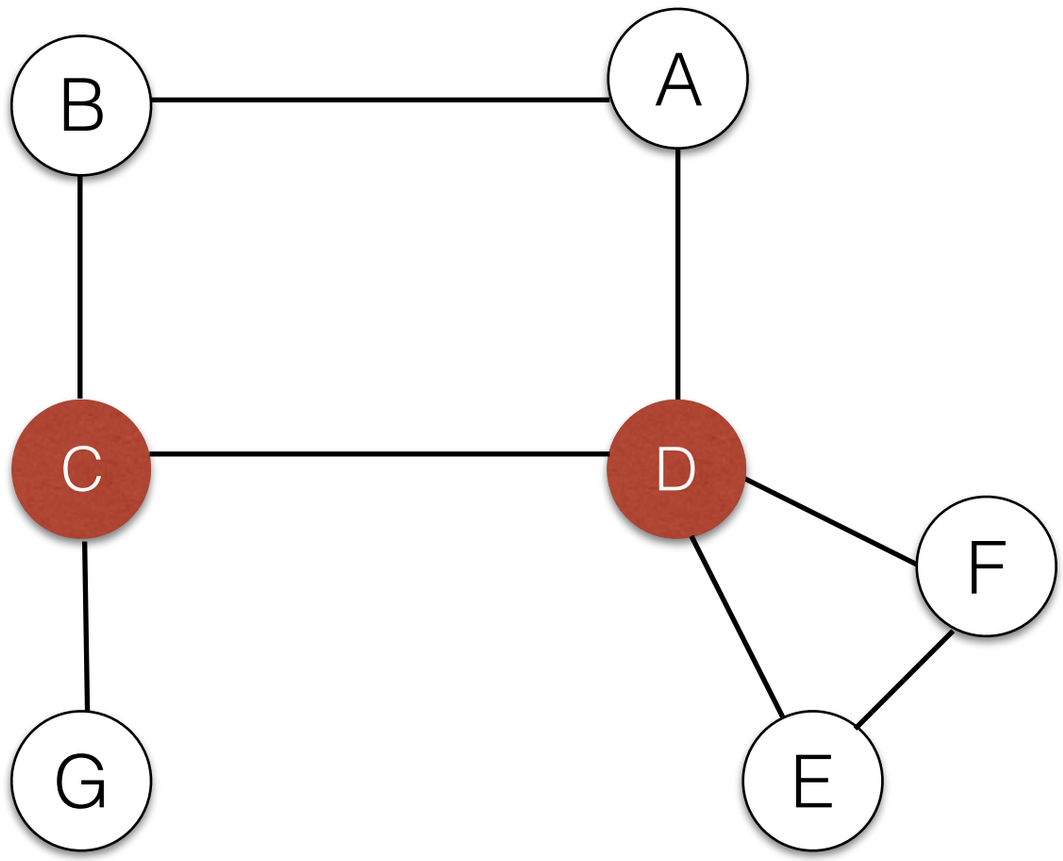
Biconnectivity

- A graph is biconnected if there is no single vertex *v*, such that removing *v* will disconnect the remaining graph.



biconnected

C and D are *articulation points.*



not biconnected

Testing for Biconnectivity

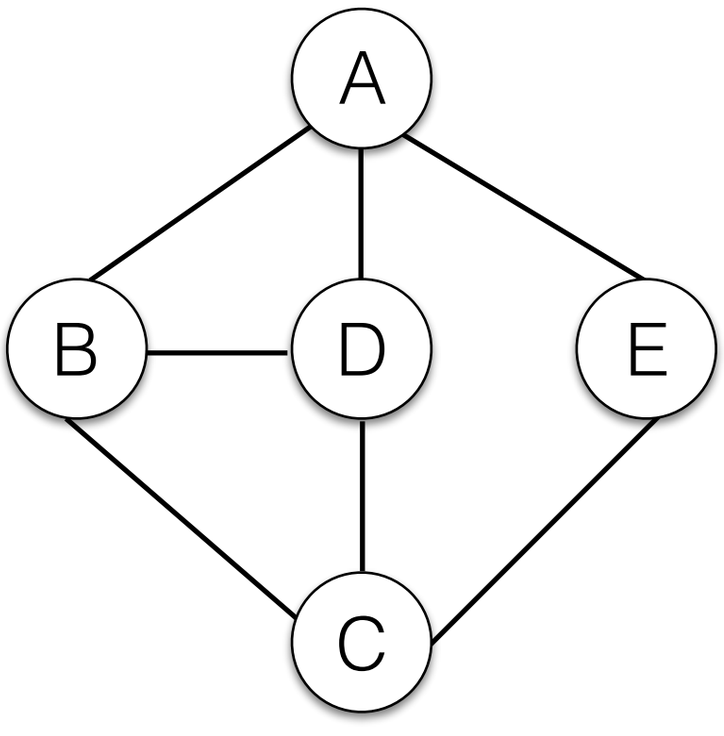
- A graph G is biconnected if:
  - G is connected.
    - G does not contain any articulation points.

- Naive approach:
  - Remove each vertex. Test if the resulting graph is still connected.

$$|V| \cdot O(|V|+|E|) = O(|V|^2 + |V| \cdot |E|)$$

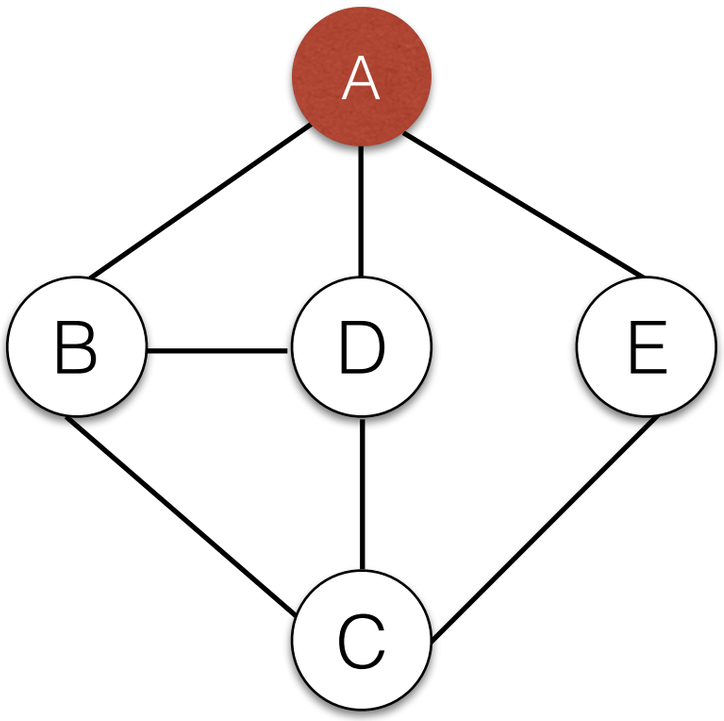
Depth First Spanning Tree

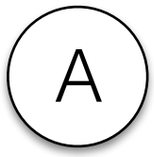
- The steps taken by DFS can be illustrated as a (directed) spanning tree.
  - Add a **tree edge** for every graph edge taken by DFS.
  - Add a ***back edge*** for every skipped edge.



Depth First Spanning Tree

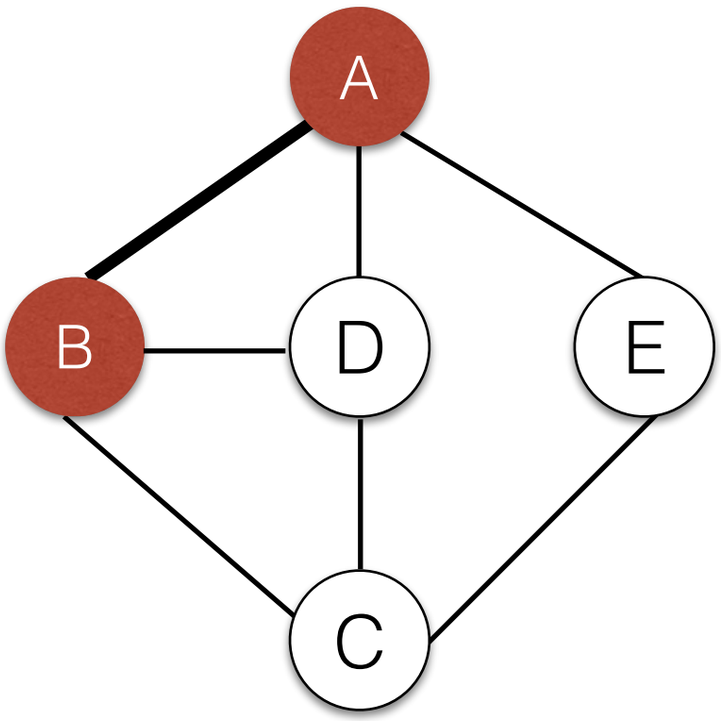
- The steps taken by DFS can be illustrated as a (directed) spanning tree.
  - Add a **tree edge** for every graph edge taken by DFS.

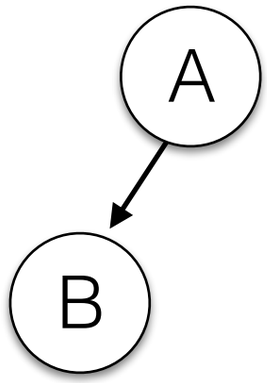




Depth First Spanning Tree

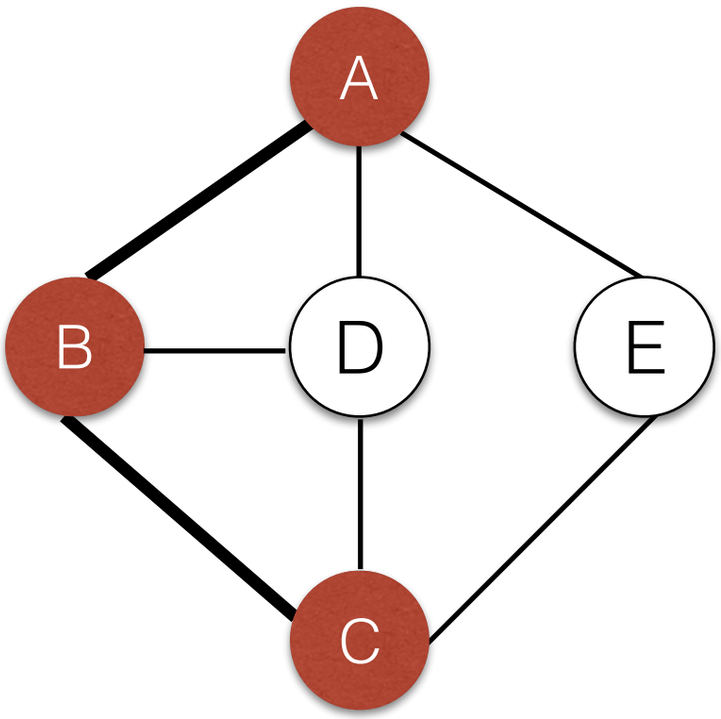
- The steps taken by DFS can be illustrated as a (directed) spanning tree.
  - Add a **tree edge** for every graph edge taken by DFS.

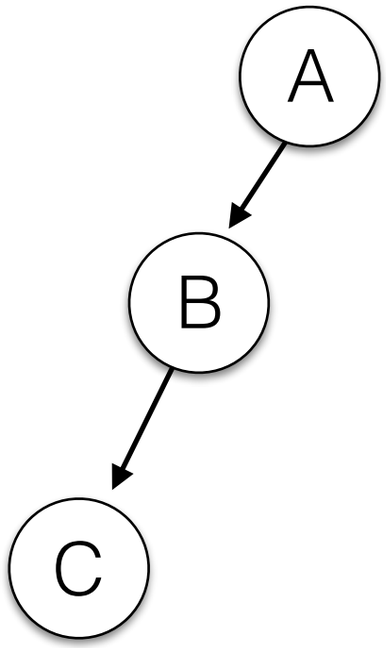




Depth First Spanning Tree

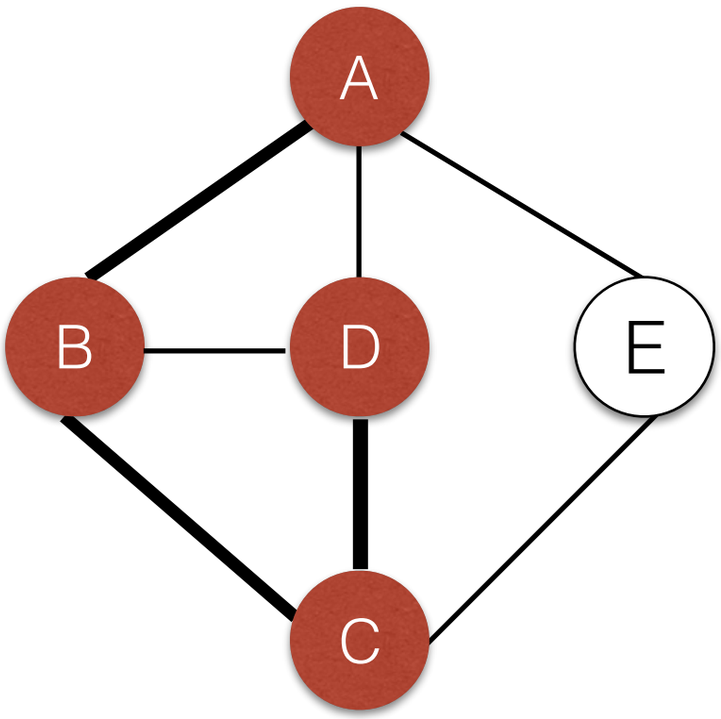
- The steps taken by DFS can be illustrated as a (directed) spanning tree.
  - Add a **tree edge** for every graph edge taken by DFS.

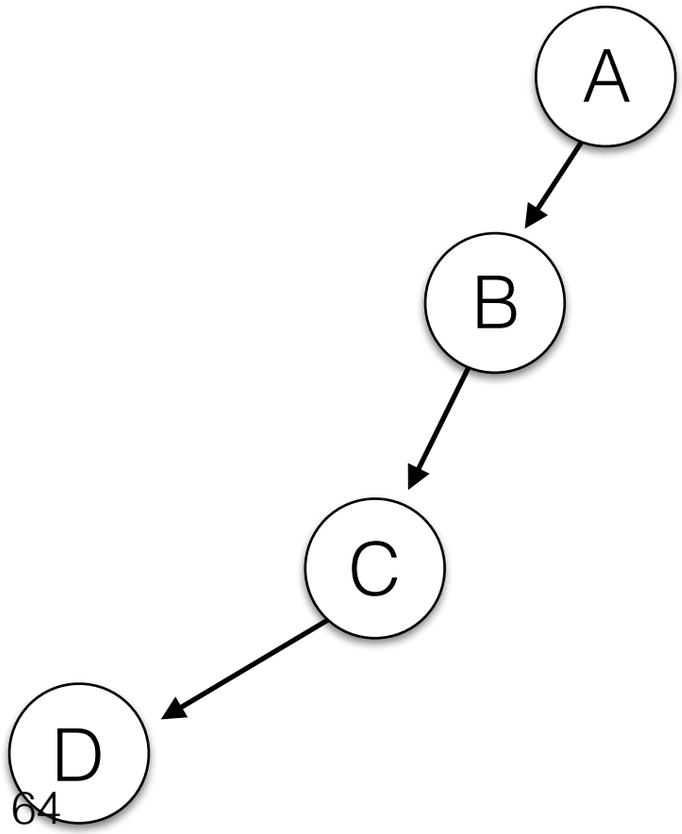




Depth First Spanning Tree

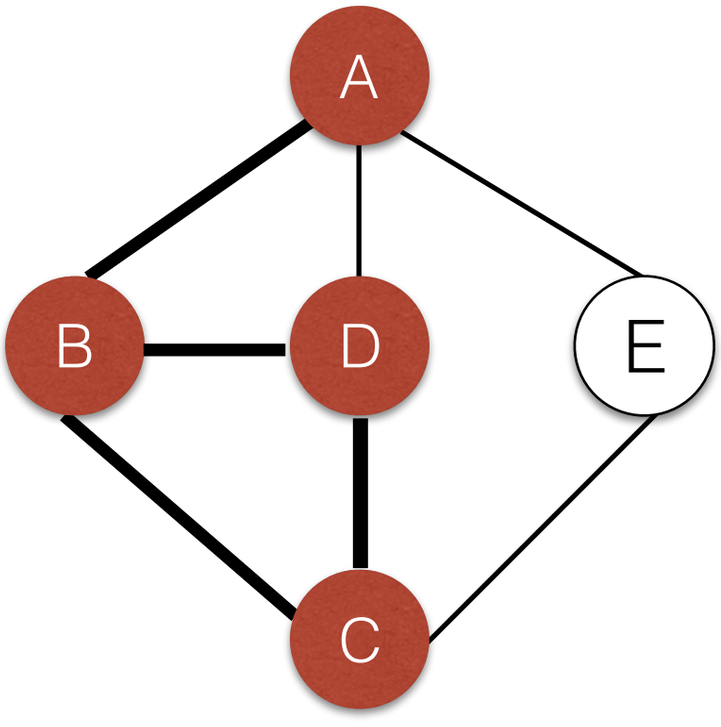
- The steps taken by DFS can be illustrated as a (directed) spanning tree.
  - Add a **tree edge** for every graph edge taken by DFS.

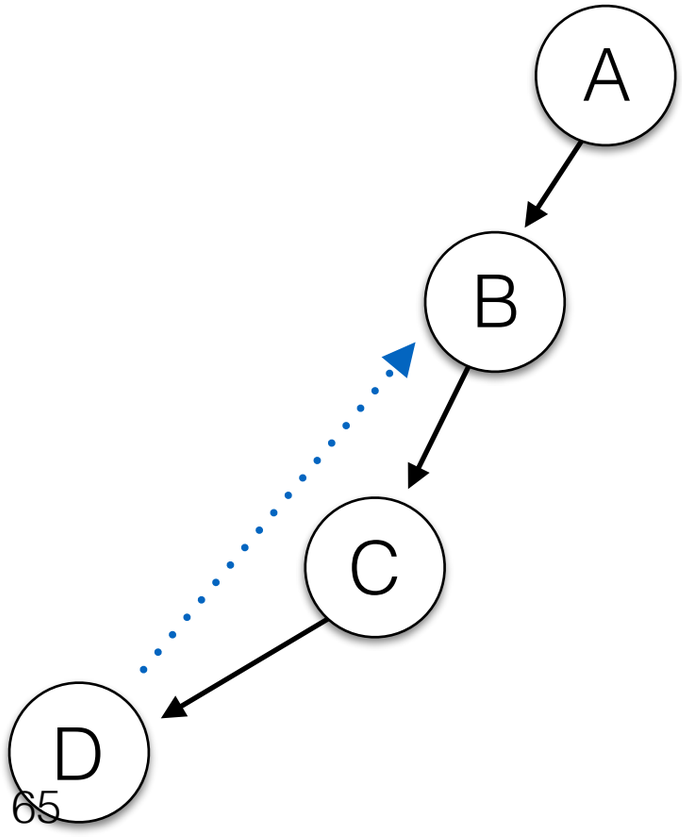




Depth First Spanning Tree

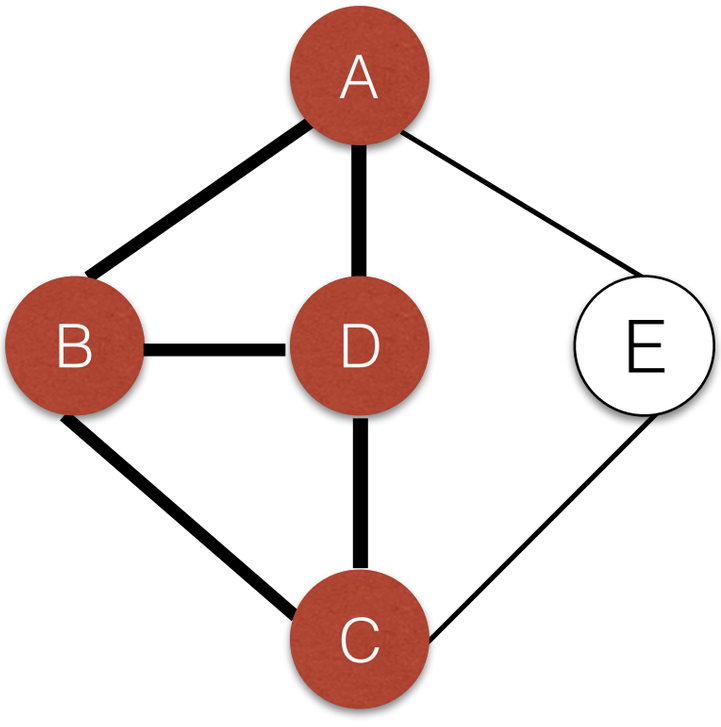
- The steps taken by DFS can be illustrated as a (directed) spanning tree.
  - Add a **tree edge** for every graph edge taken by DFS.
  - Add a ***back edge*** for every skipped edge.

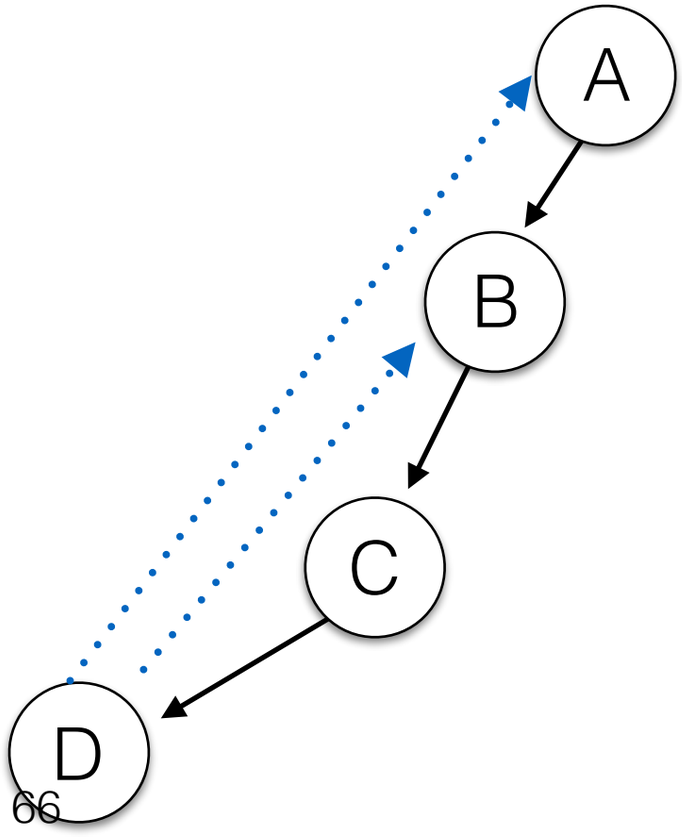




Depth First Spanning Tree

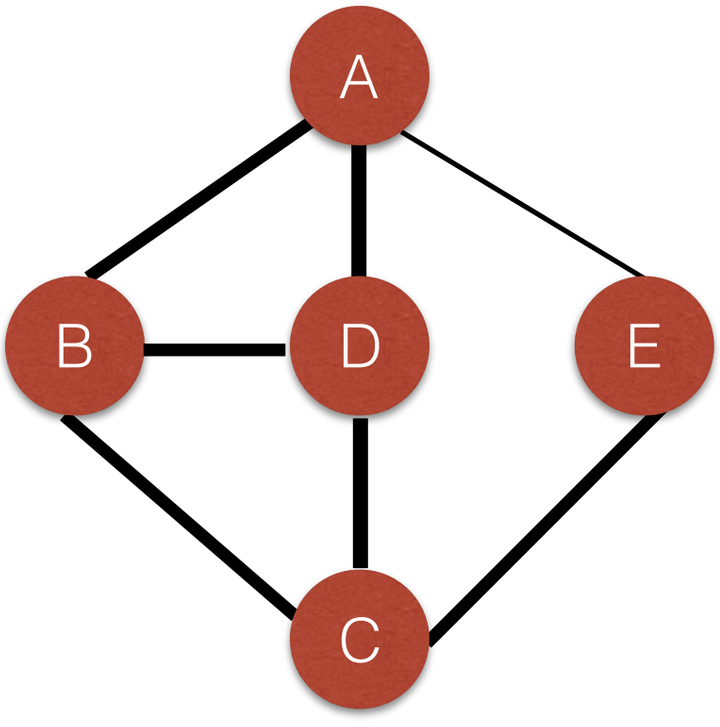
- The steps taken by DFS can be illustrated as a (directed) spanning tree.
  - Add a **tree edge** for every graph edge taken by DFS.
- Add a ***back edge*** for every skipped edge.

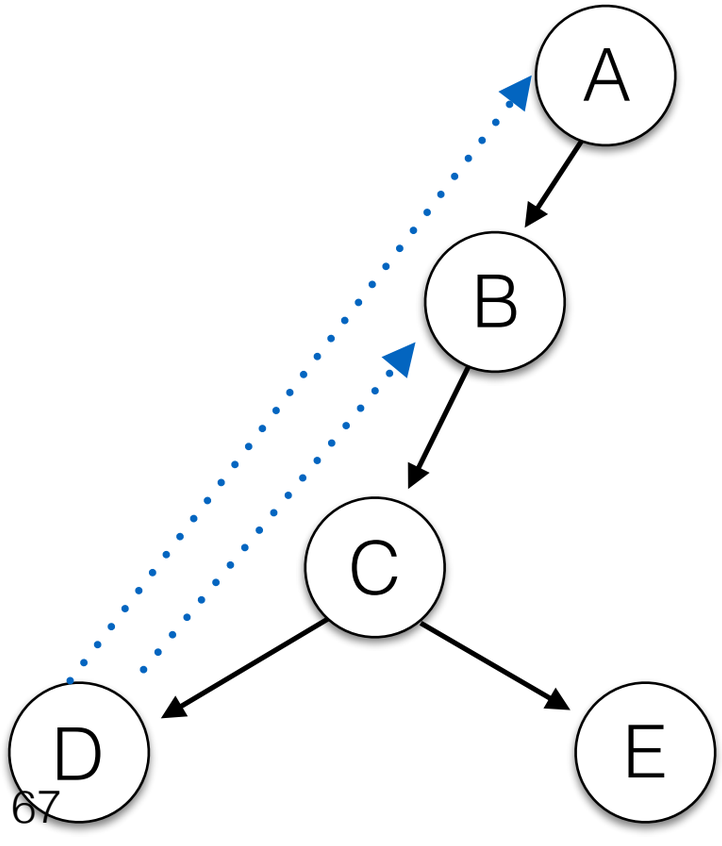




Depth First Spanning Tree

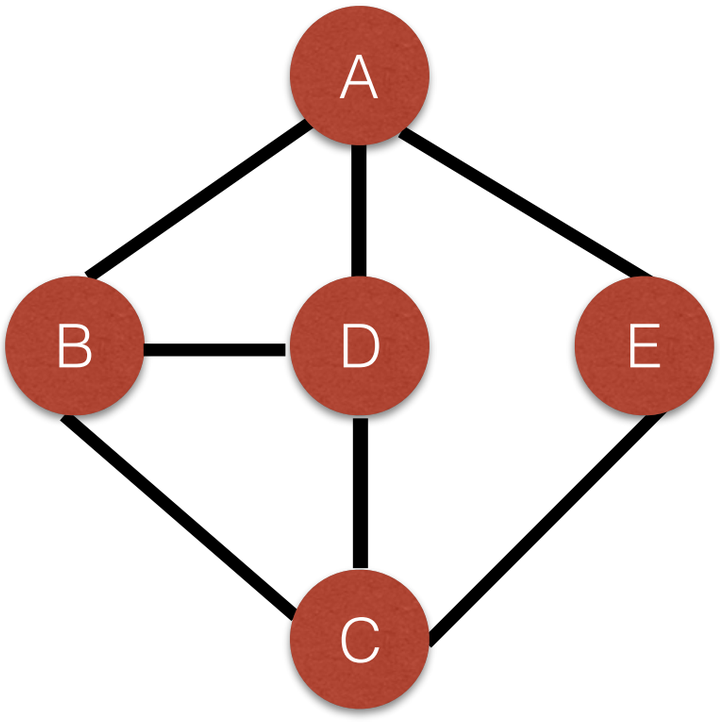
- The steps taken by DFS can be illustrated as a (directed) spanning tree.
  - Add a **tree edge** for every graph edge taken by DFS.
- Add a ***back edge*** for every skipped edge.

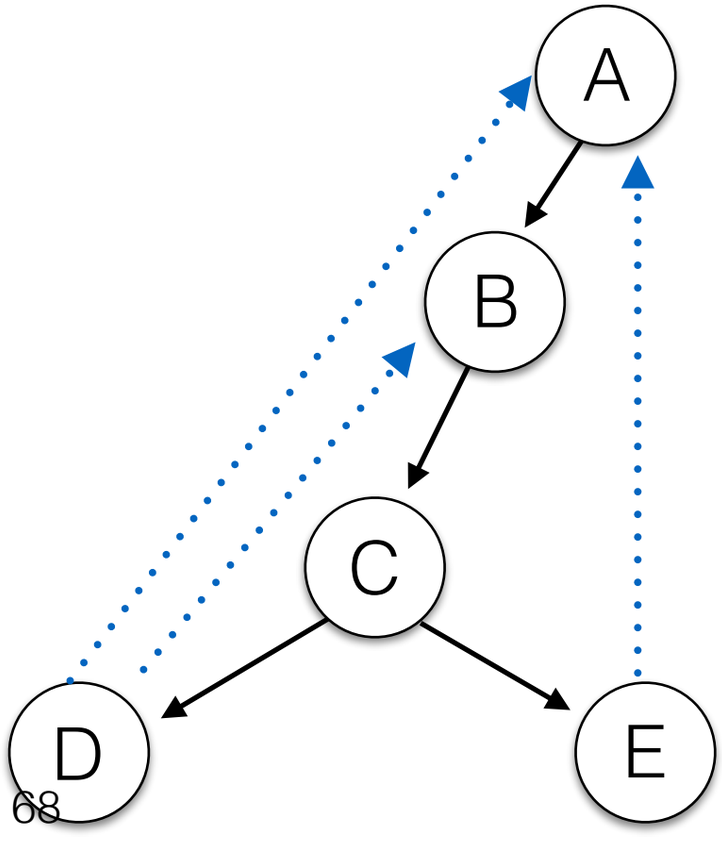




Depth First Spanning Tree

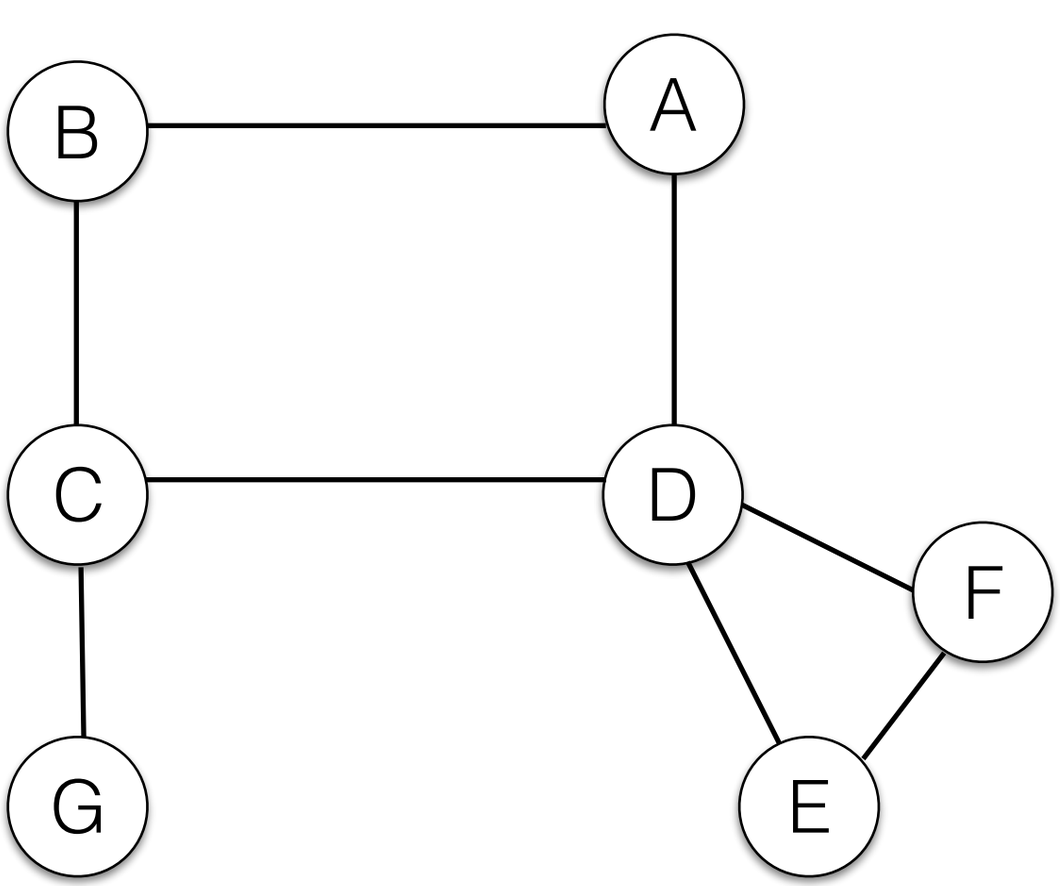
- The steps taken by DFS can be illustrated as a (directed) spanning tree.
  - Add a **tree edge** for every graph edge taken by DFS.
- Add a ***back edge*** for every skipped edge.

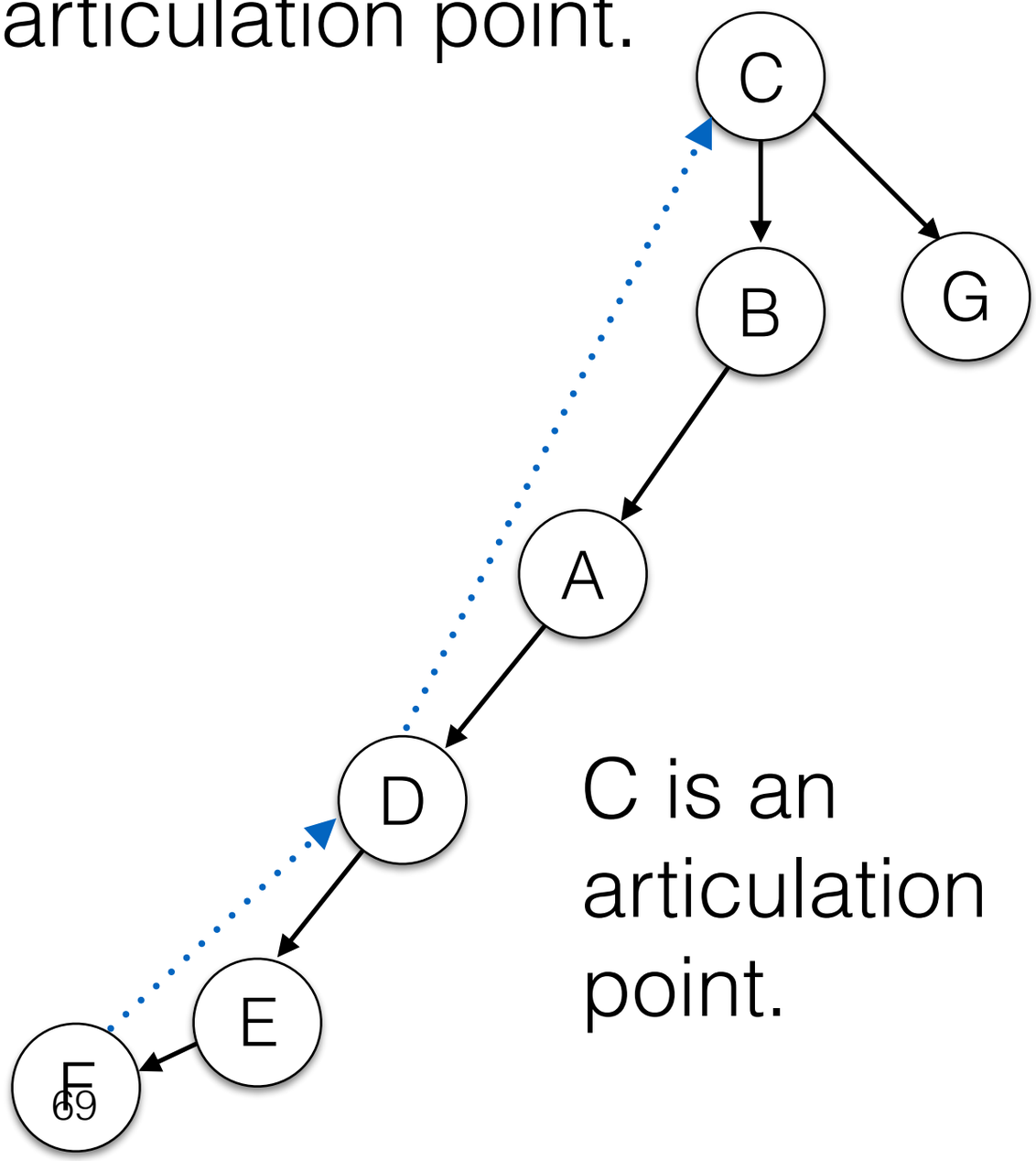




Identifying Articulation Points (1)

- If the root of the DFS spanning tree has two outgoing tree edges, the root is an articulation point.

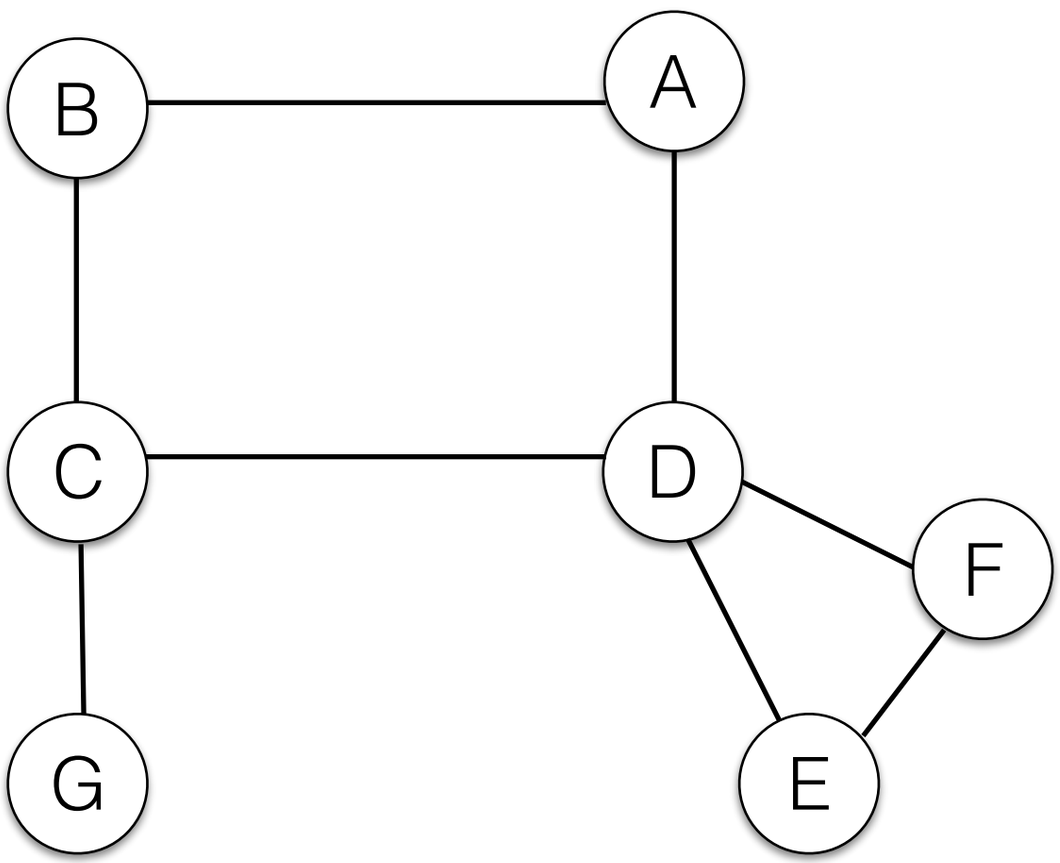


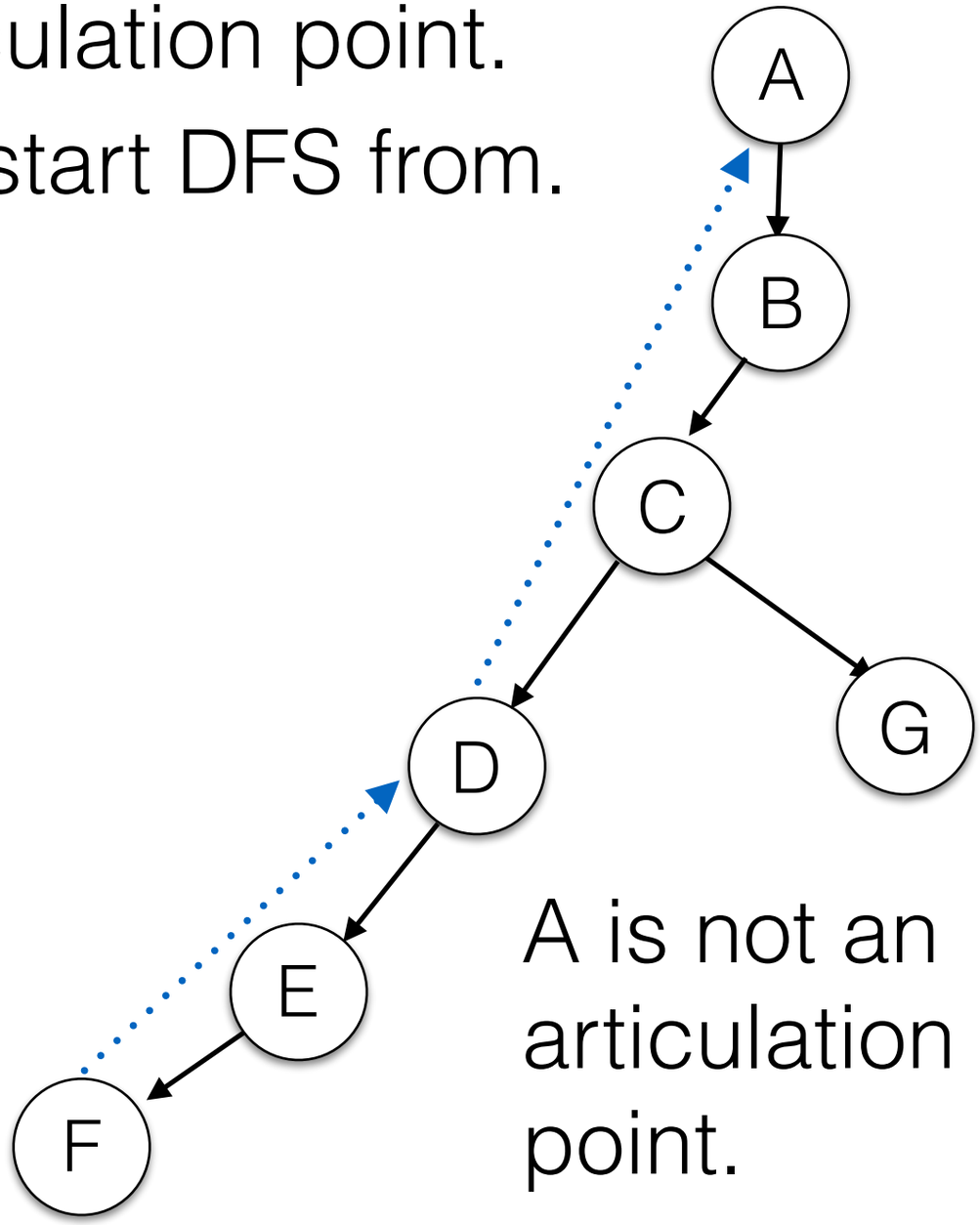


C is an articulation point.

Identifying Articulation Points (1)

- If the root of the DFS spanning tree has two outgoing tree edges, the root is an articulation point.
- Depends on which vertex we start DFS from.

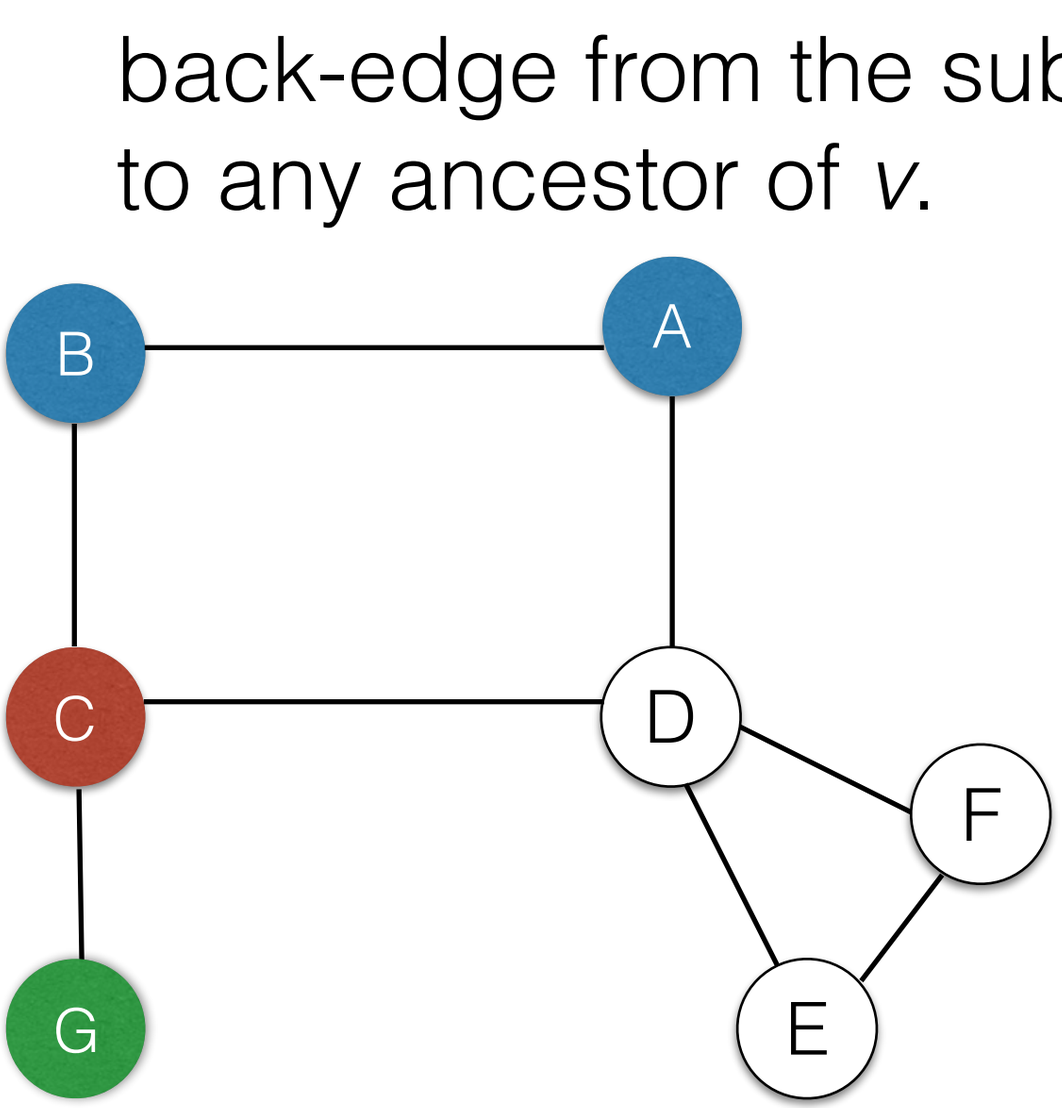


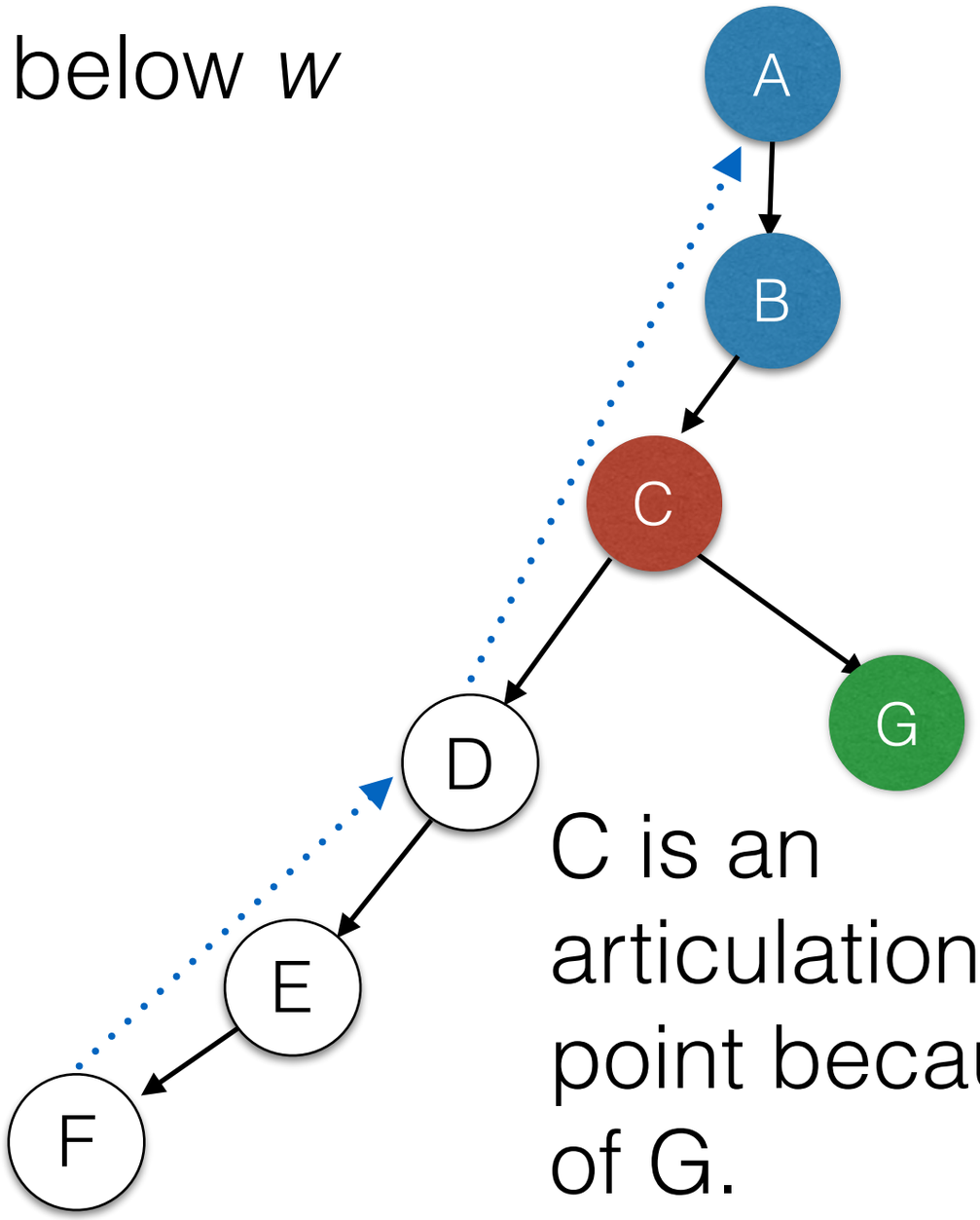


A is not an articulation point.

Identifying Articulation Points (2)

- Any non-root vertex *v* is an articulation point iff
  - *v* has a child *w* such that there is no back-edge from the subtree below *w* to any ancestor of *v*.

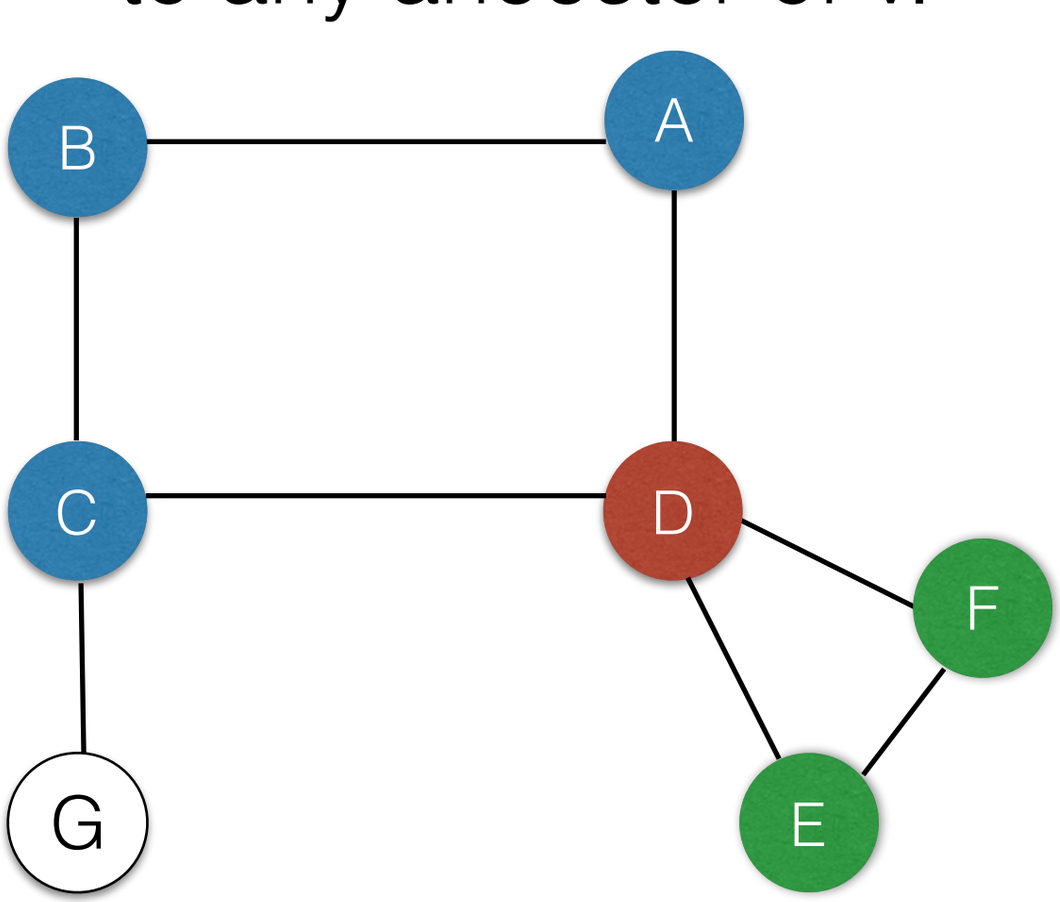


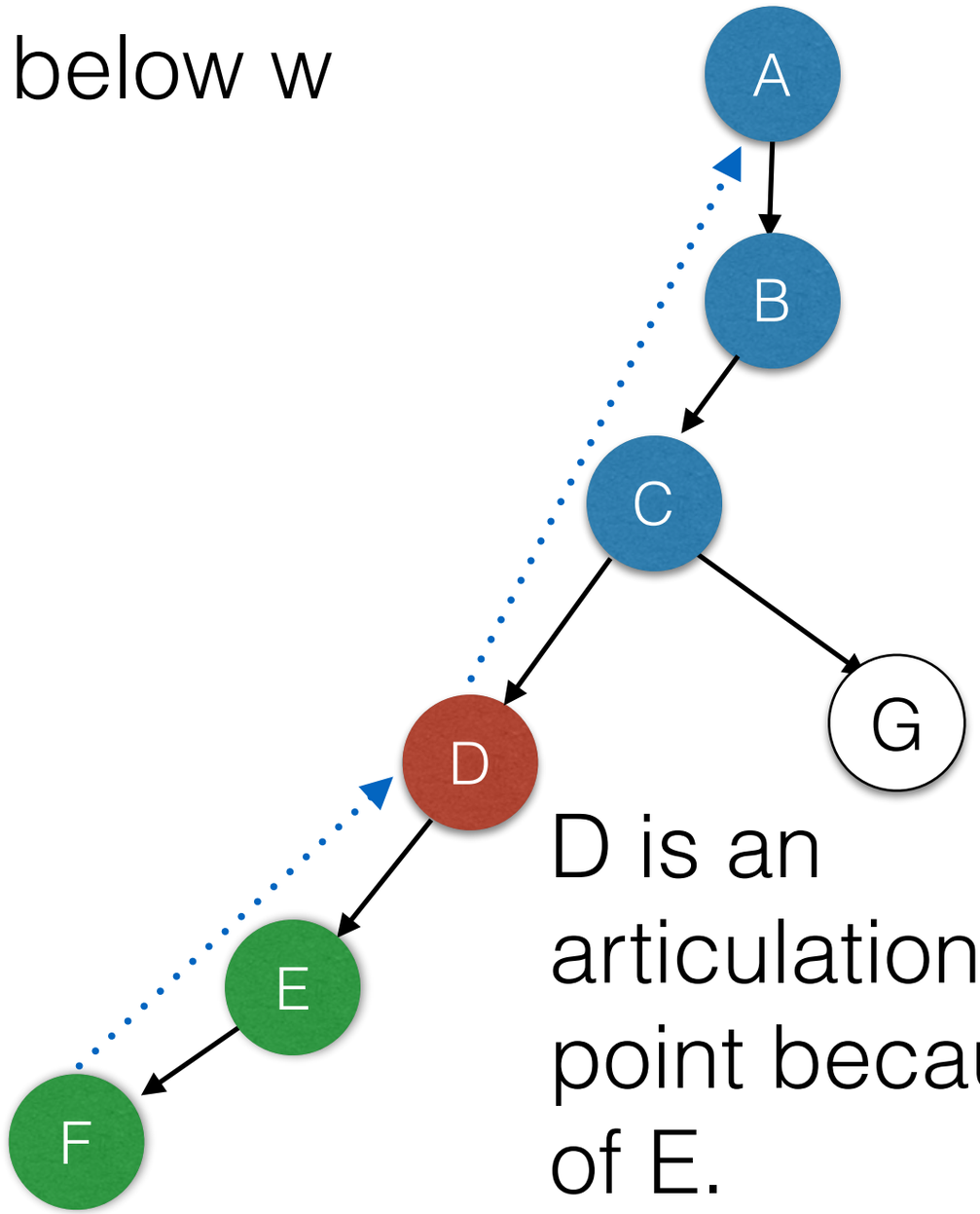


C is an articulation point because of G.

Identifying Articulation Points (2)

- Any non-root vertex *v* is an articulation point iff
  - v has a child w such that there is no back-edge from the subtree below w to any ancestor of v.



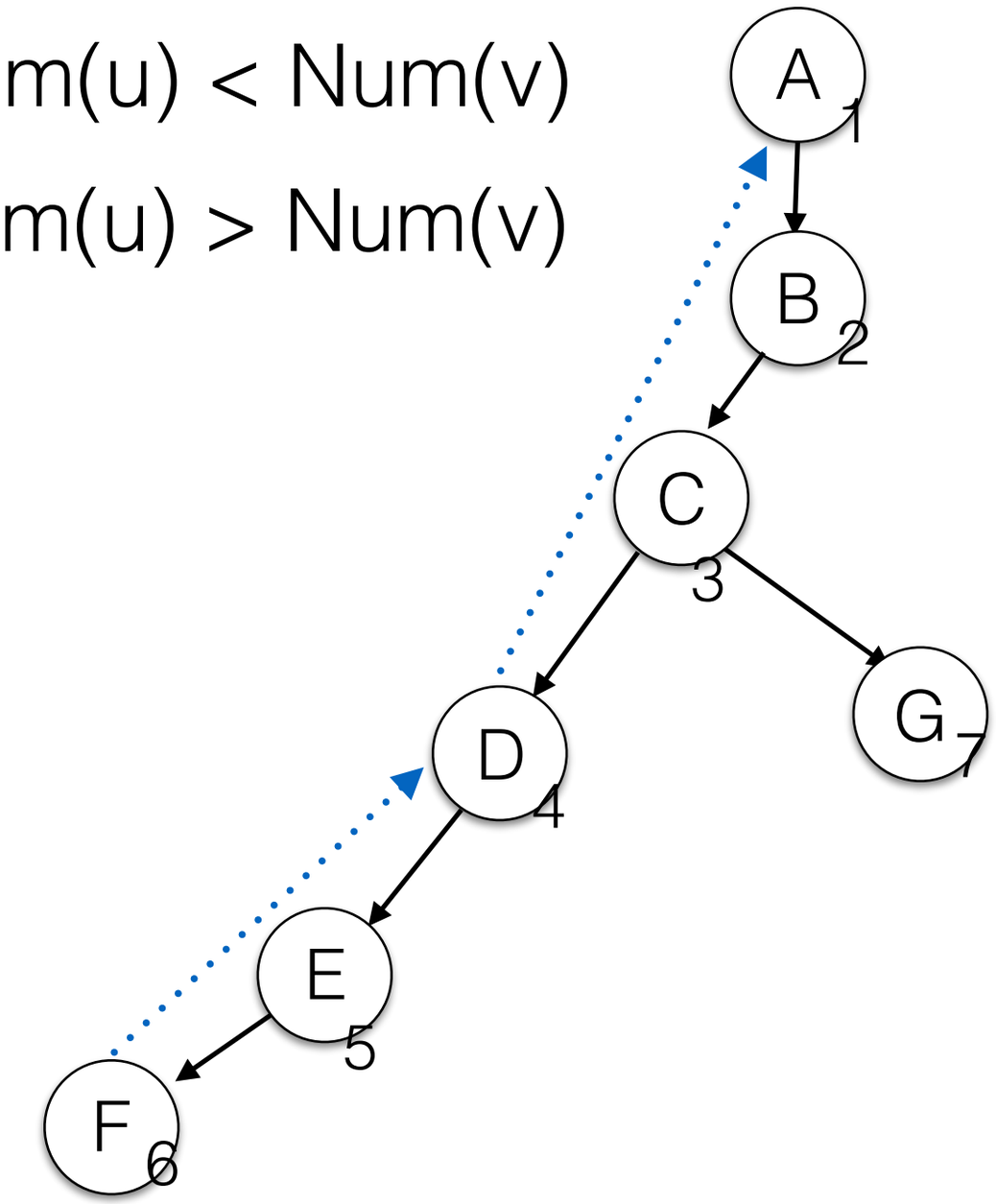


D is an articulation point because of E.

Preorder Numbers

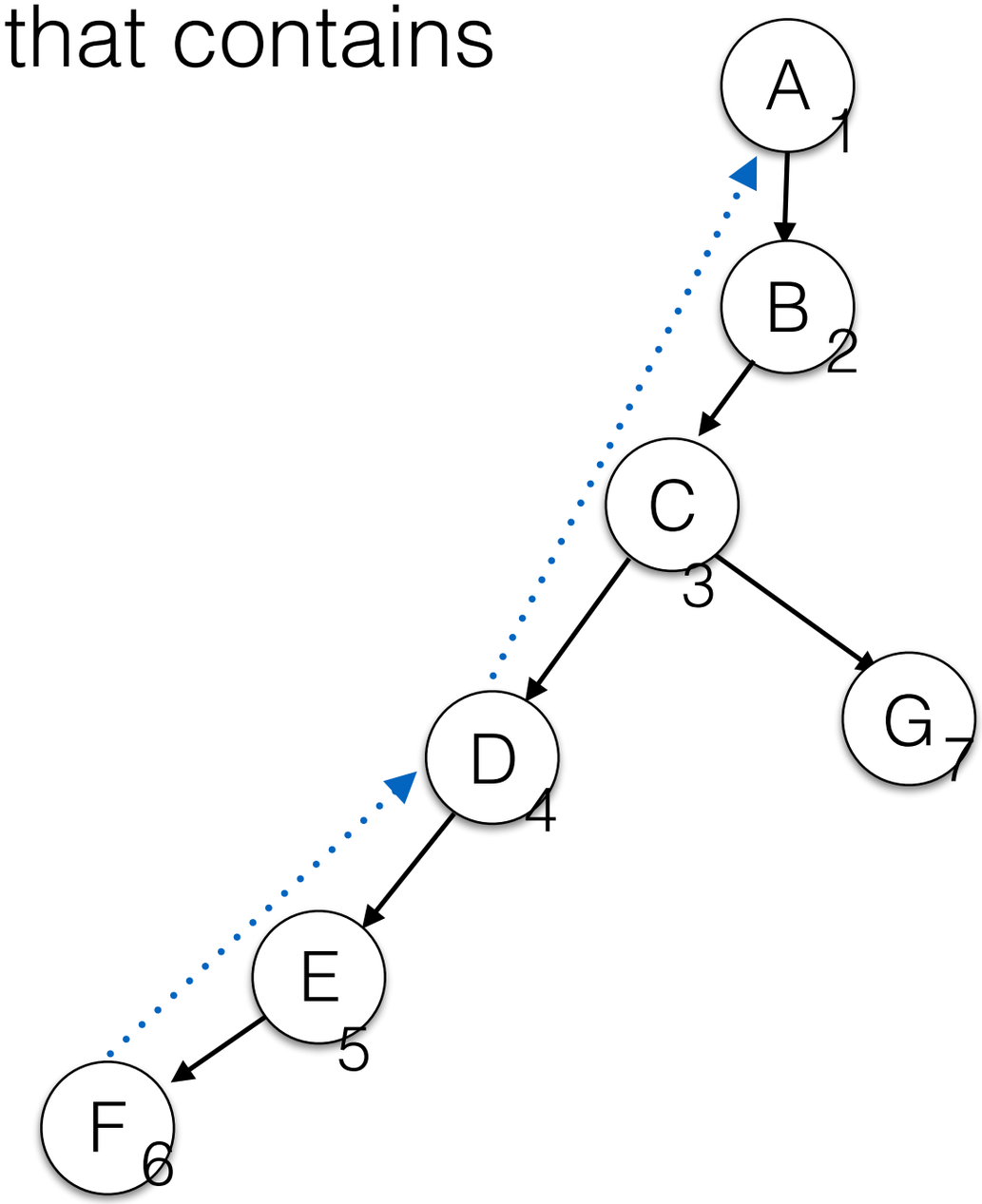
- Assign numbers to each vertex in the order in which they are visited by DFS.
- For every tree edge (u,v): Num(u) < Num(v)
- For every back edge (u,v): Num(u) > Num(v)

v	*Num(v)*
A	1
B	2
C	3
D	4
E	5
F	6
G	7



Low numbers

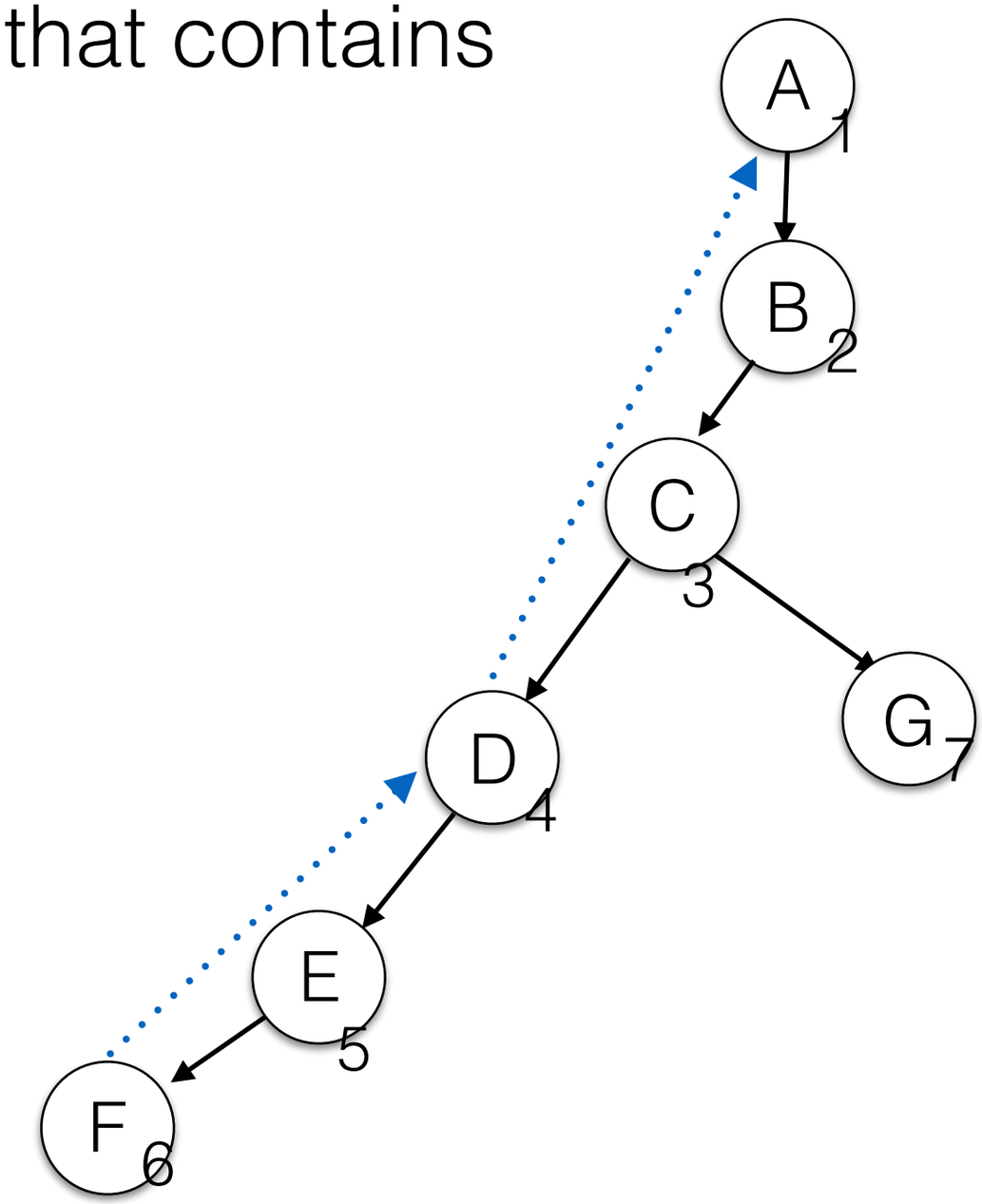
- For each vertex, find the lowest numbered vertex that is reachable by following a path that contains *at most one back edge.*



v	*Num(v)*	*Low(v)*
A	1	1
B	2	
C	3	
D	4	
E	5	
F	6	
G	7	

Low numbers

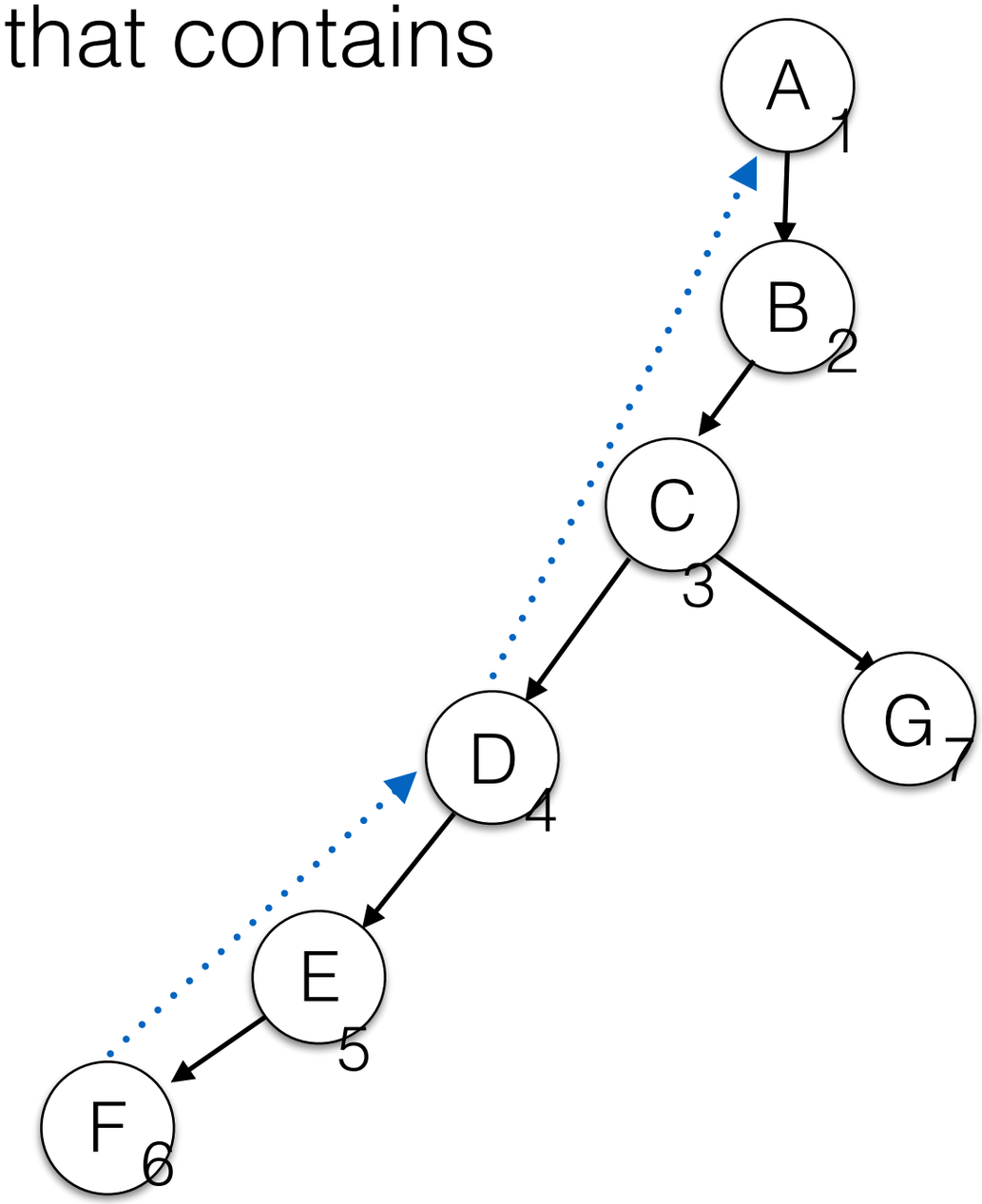
- For each vertex, find the lowest numbered vertex that is reachable by following a path that contains *at most one back edge.*



v	*Num(v)*	*Low(v)*
A	1	1
B	2	1
C	3	1
D	4	1
E	5	
F	6	
G	7	

Low numbers

- For each vertex, find the lowest numbered vertex that is reachable by following a path that contains *at most one back edge.*

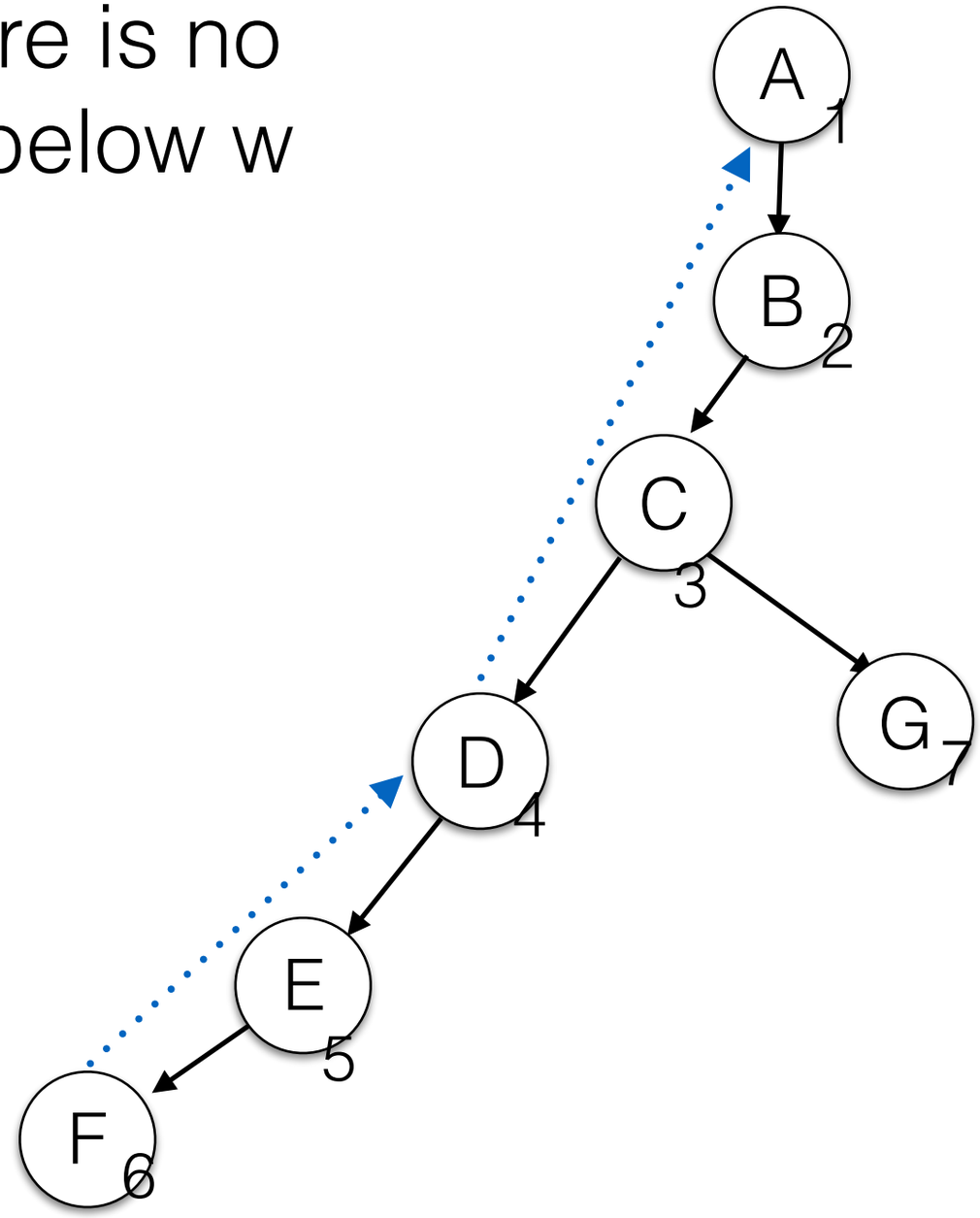


v	*Num(v)*	*Low(v)*
A	1	1
B	2	1
C	3	1
D	4	1
E	5	4
F	6	4
G	7	7

Identifying Articulation Points

- Any non-root vertex *v* is an articulation point iff
  - v has a child w such that there is no back-edge from the subtree below w to any ancestor of v.

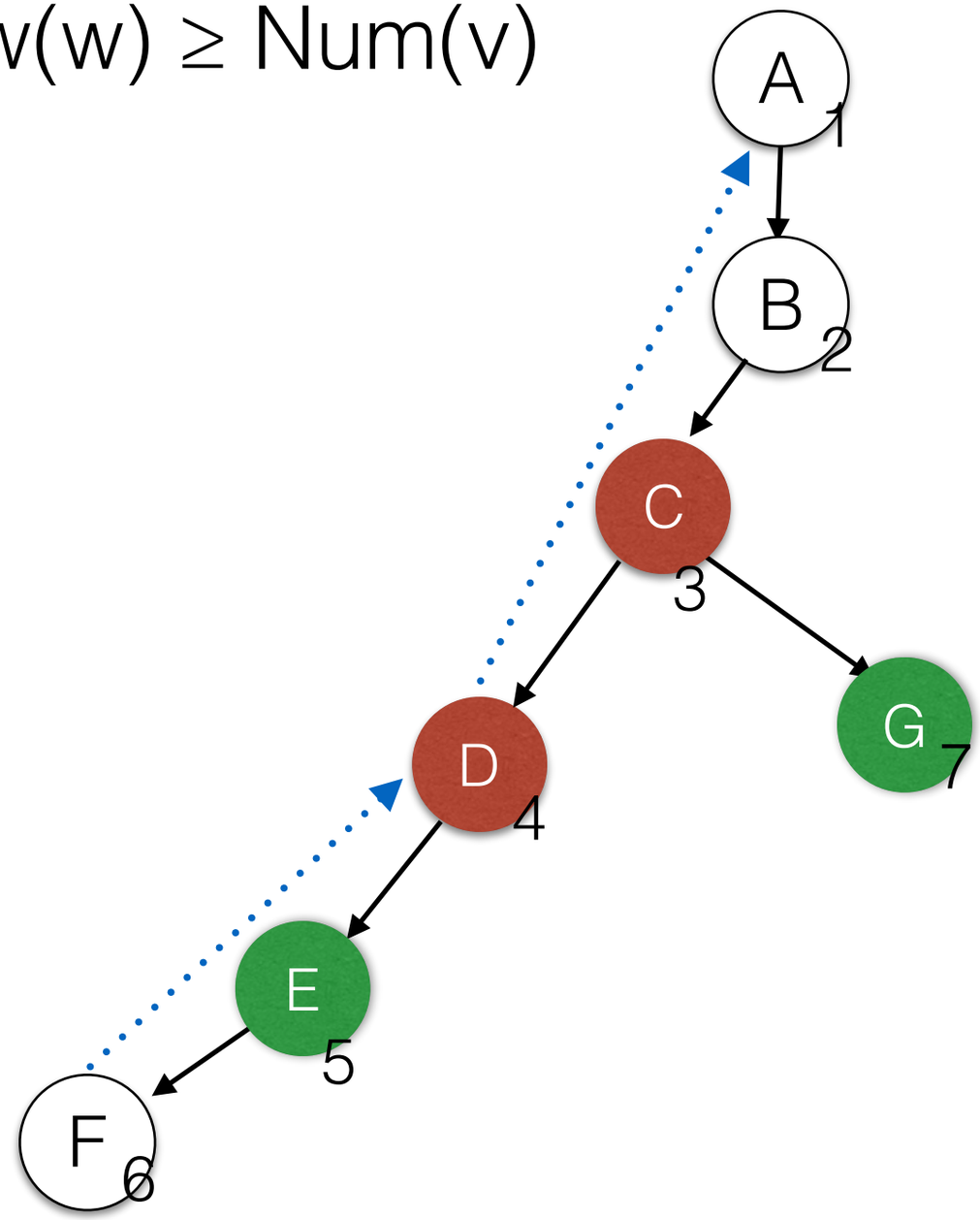
v	*Num(v)*	*Low(v)*
A	1	1
B	2	1
C	3	1
D	4	1
E	5	4
F	6	4
G	7	7



Identifying Articulation Points

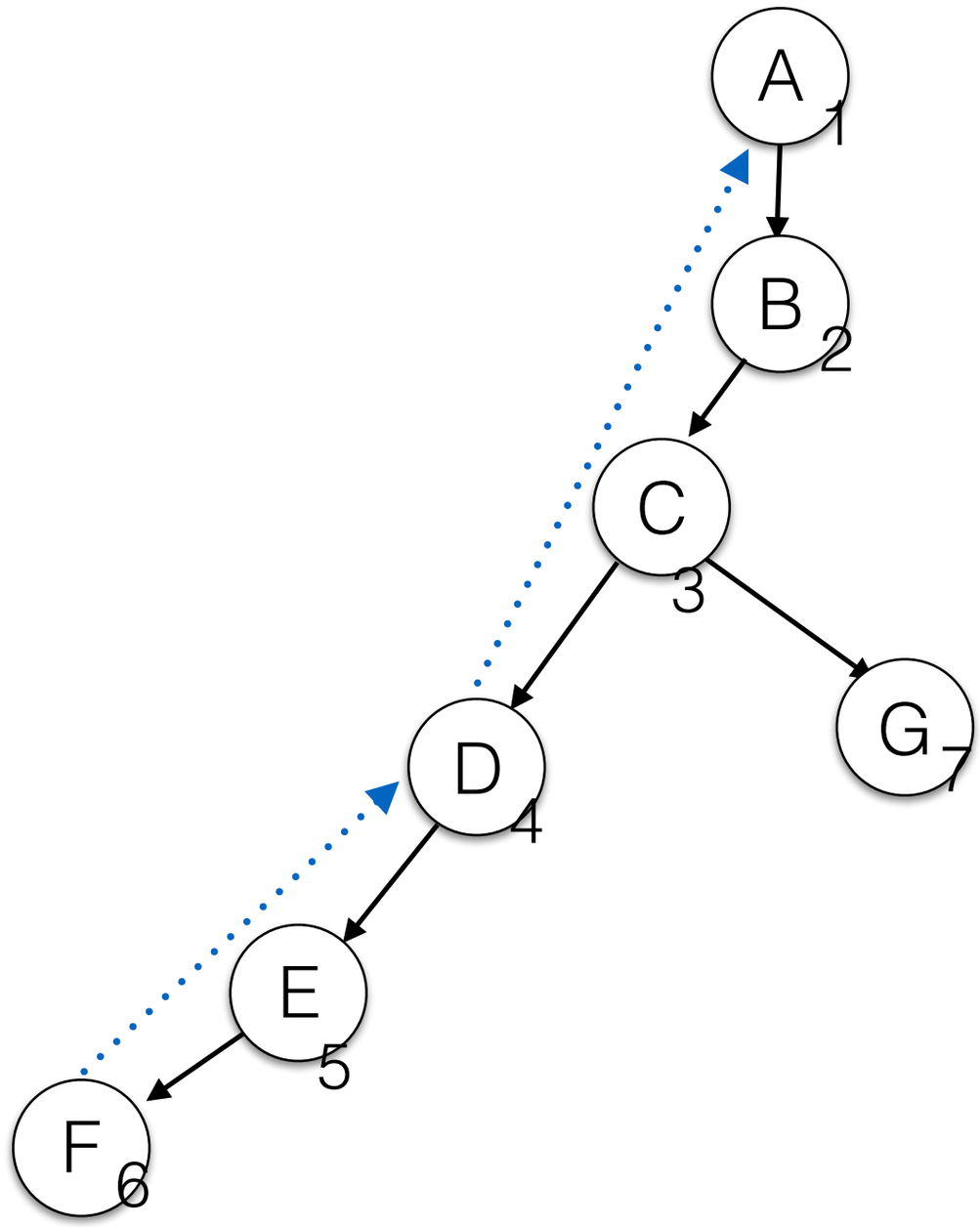
- Any non-root vertex *v* is an articulation point iff
  - v has a child w such that Low(w) ≥ Num(v)

v	*Num(v)*	*Low(v)*
A	1	1
B	2	1
C	**3**	1
D	**4**	1
E	5	**4**
F	6	4
G	7	**7**



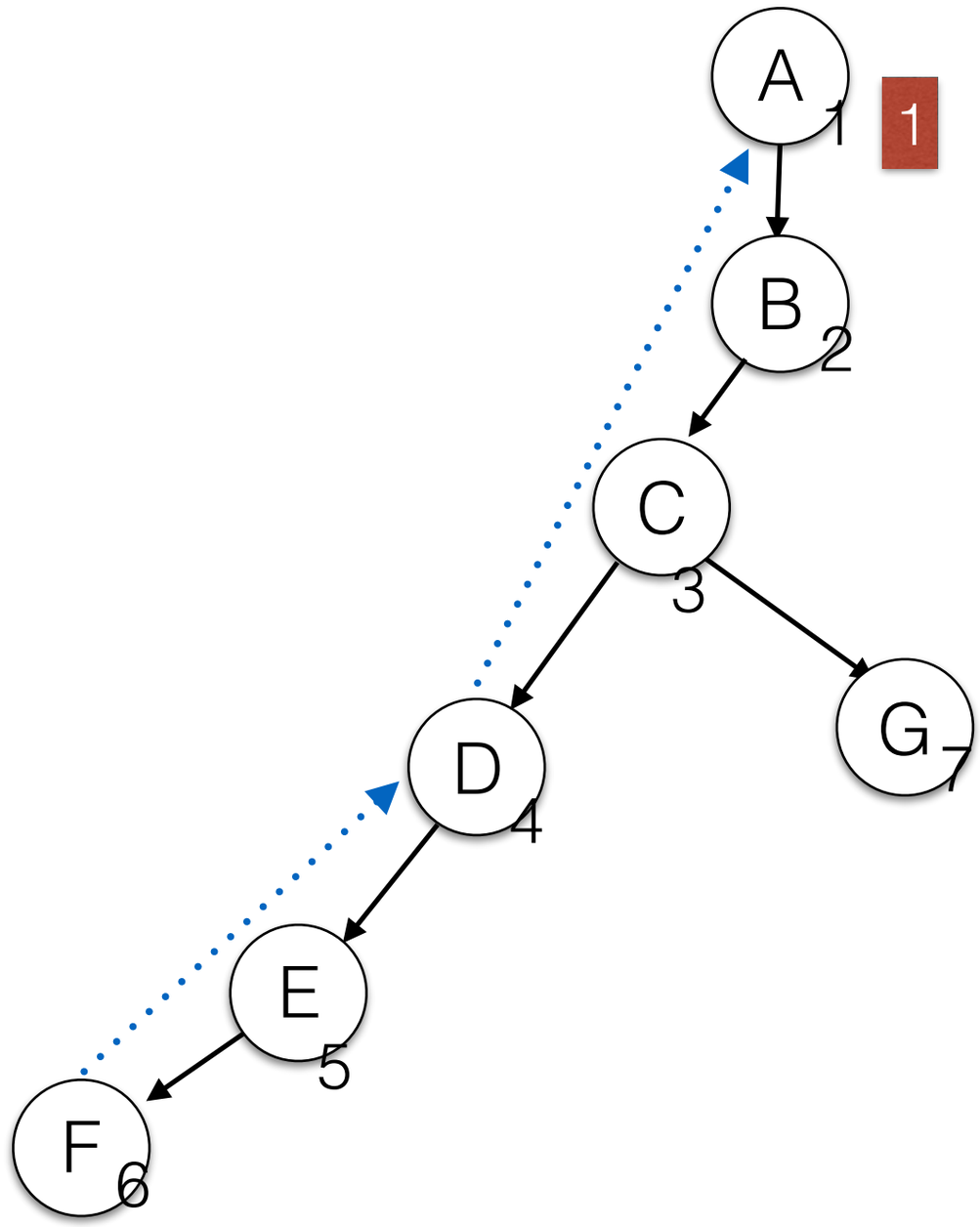
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

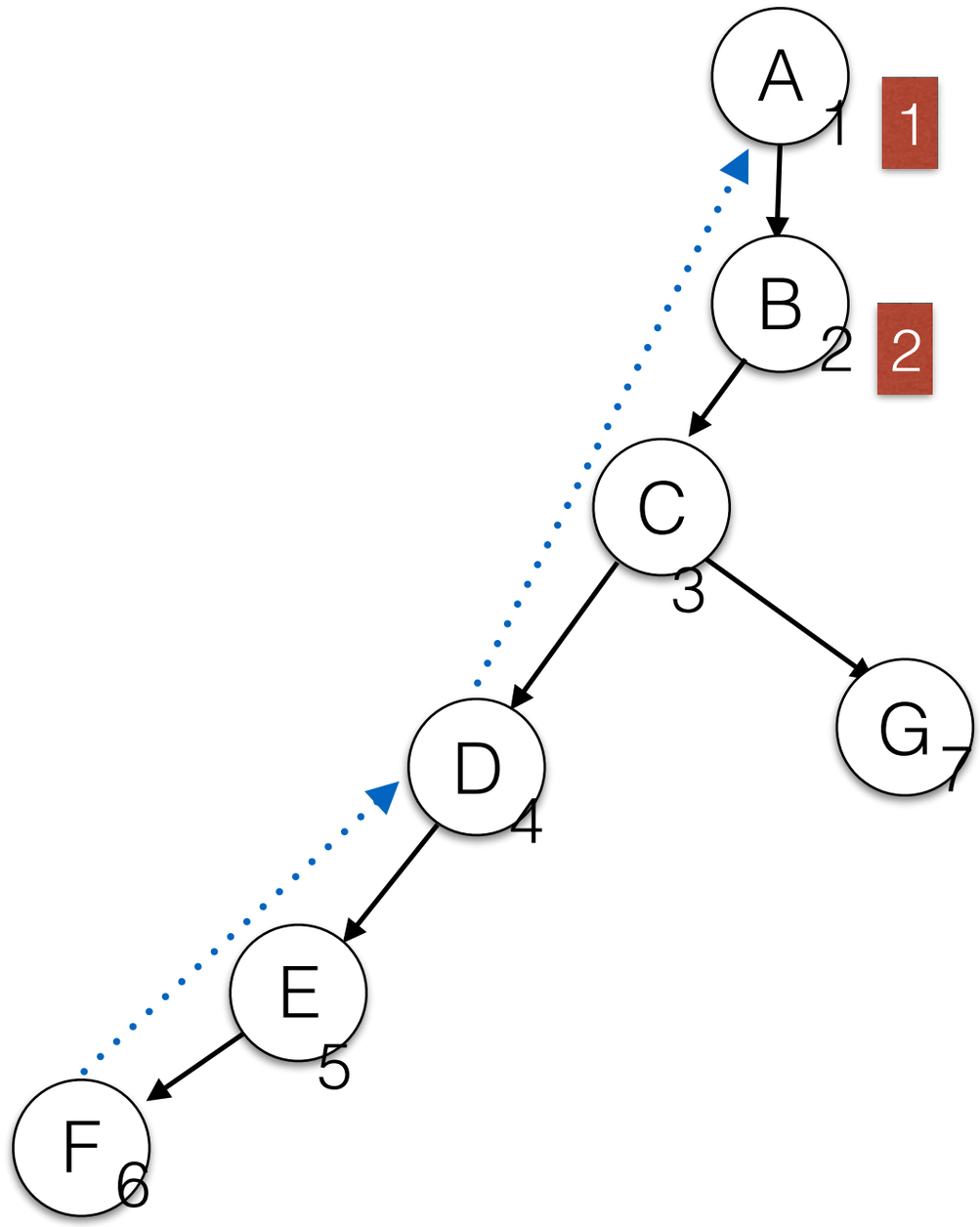
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

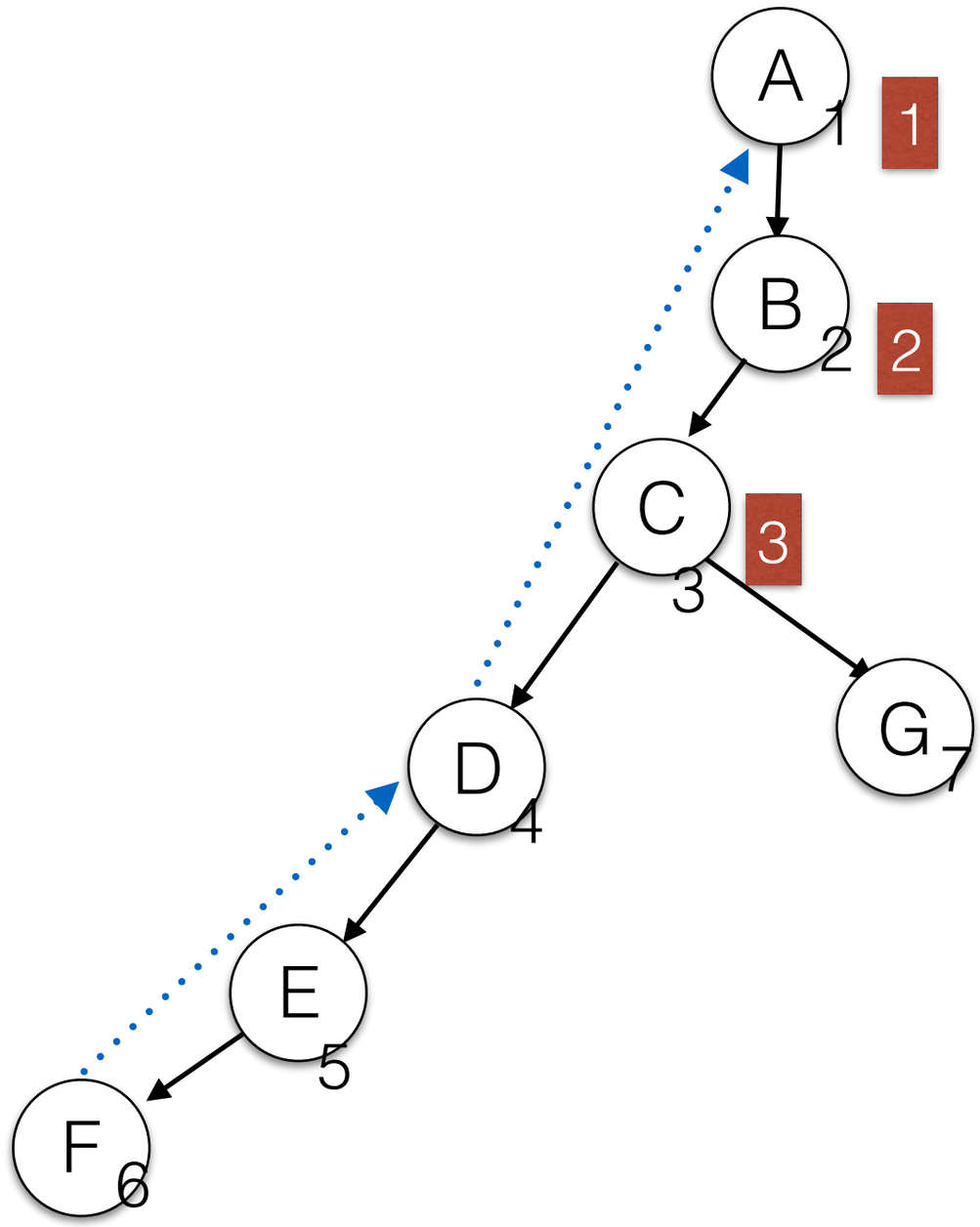
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

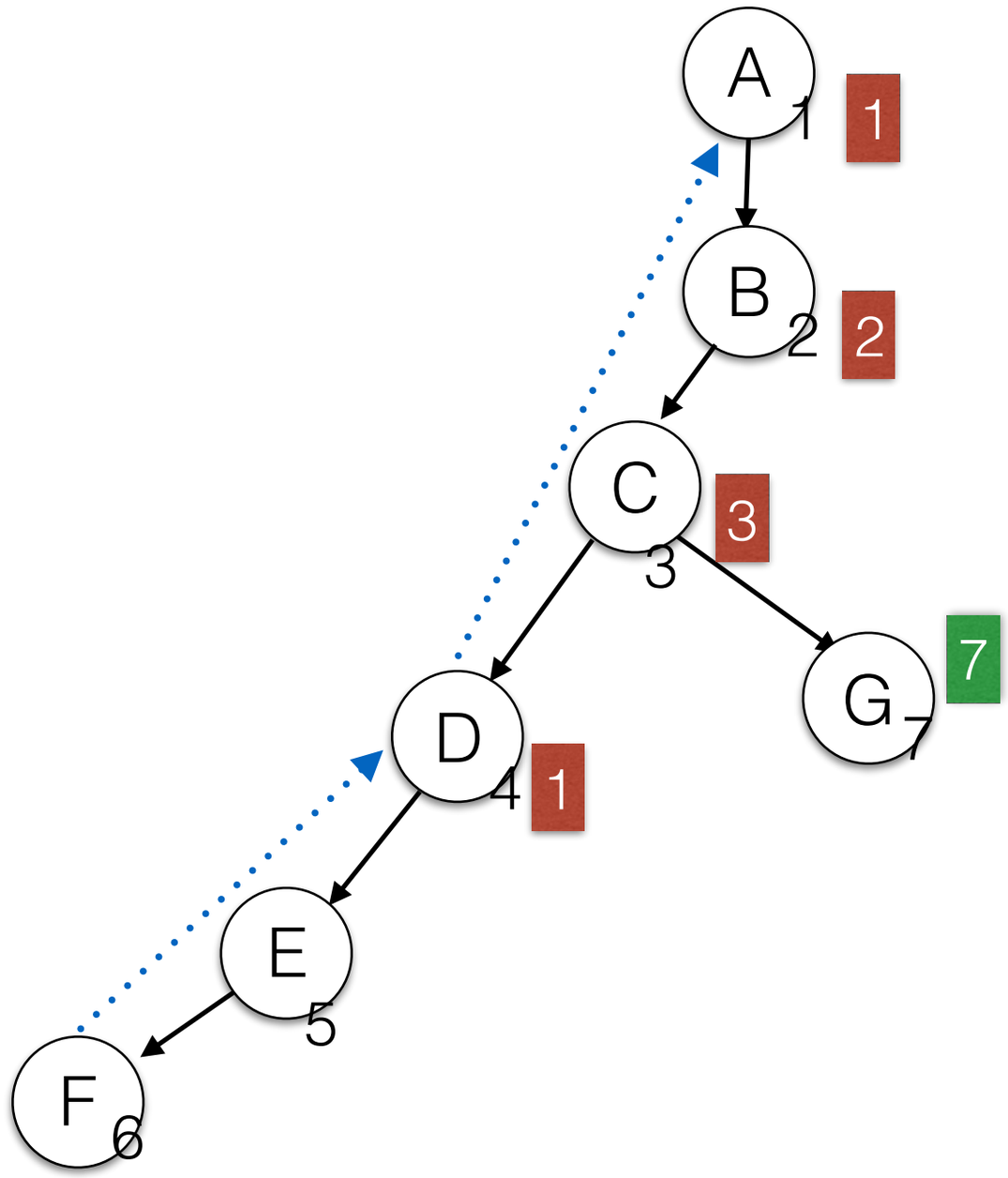
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

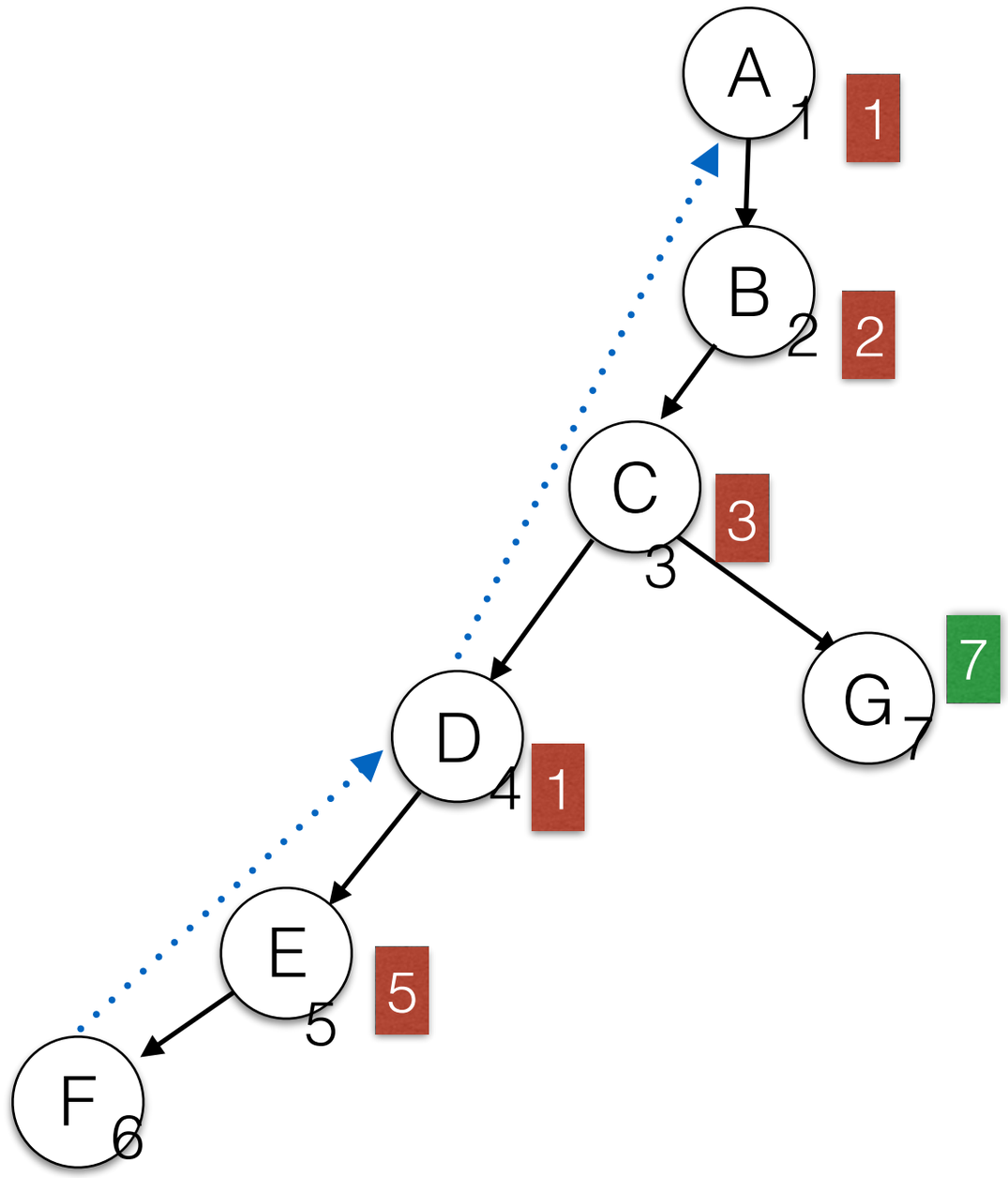
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

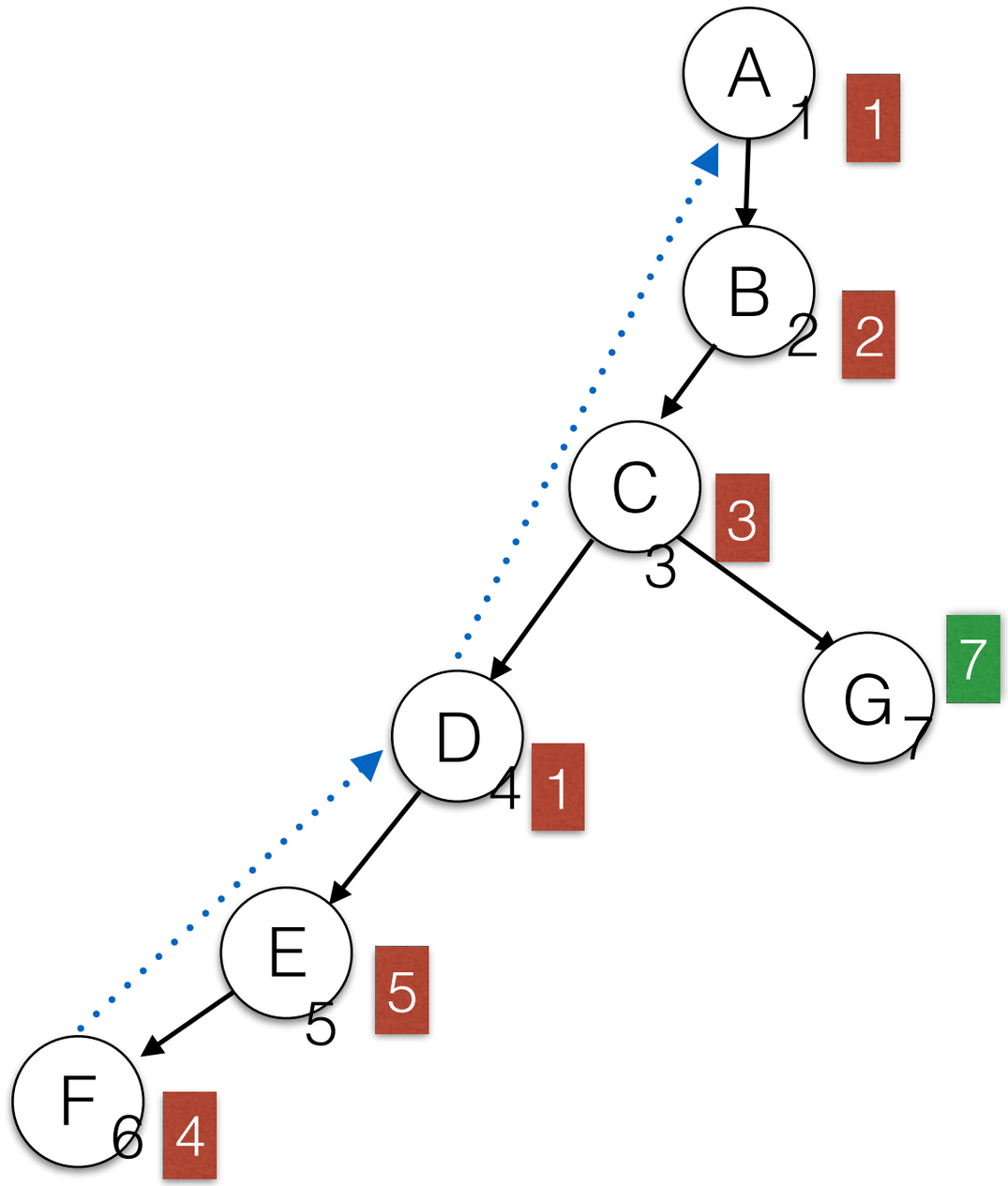
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

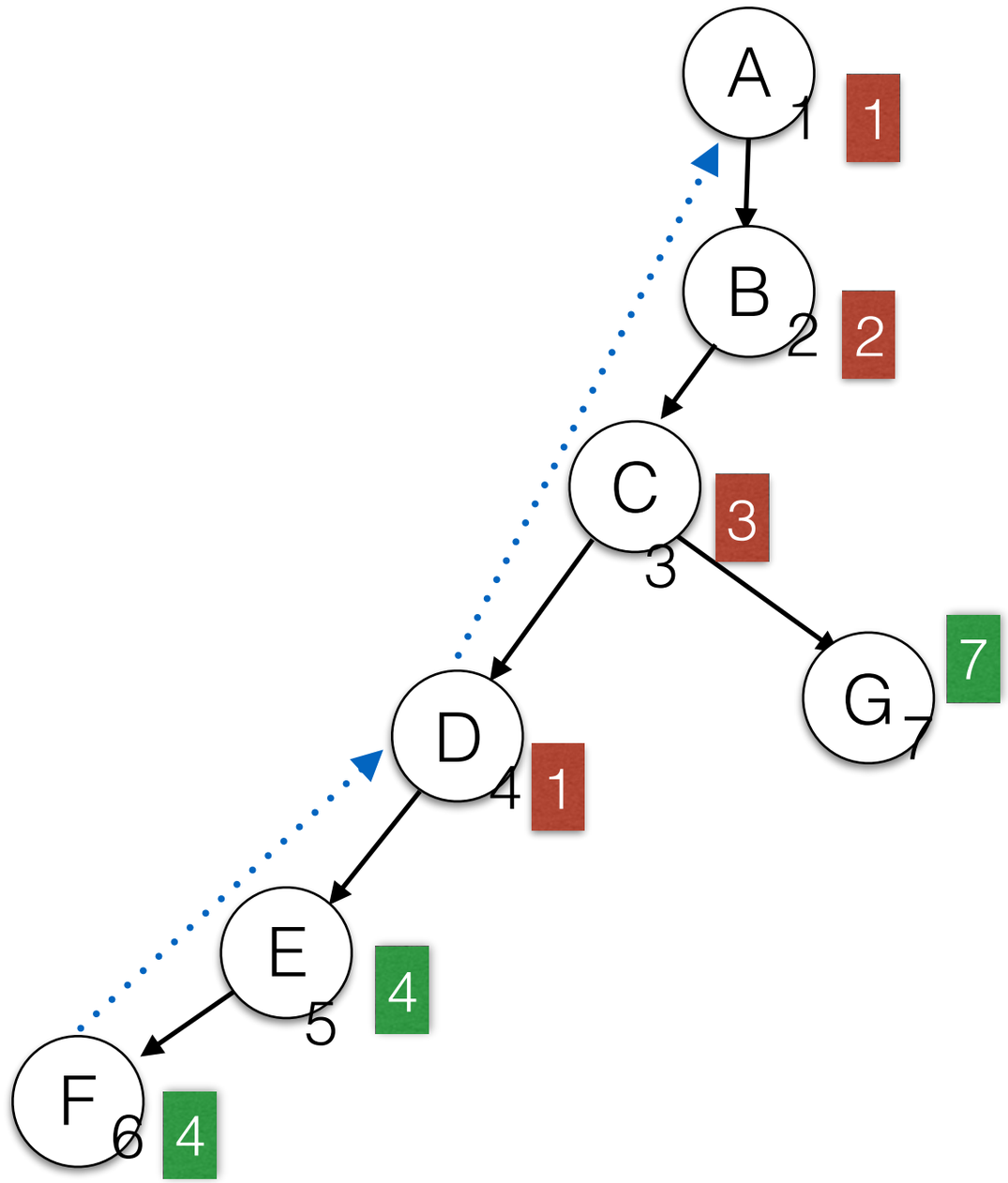
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

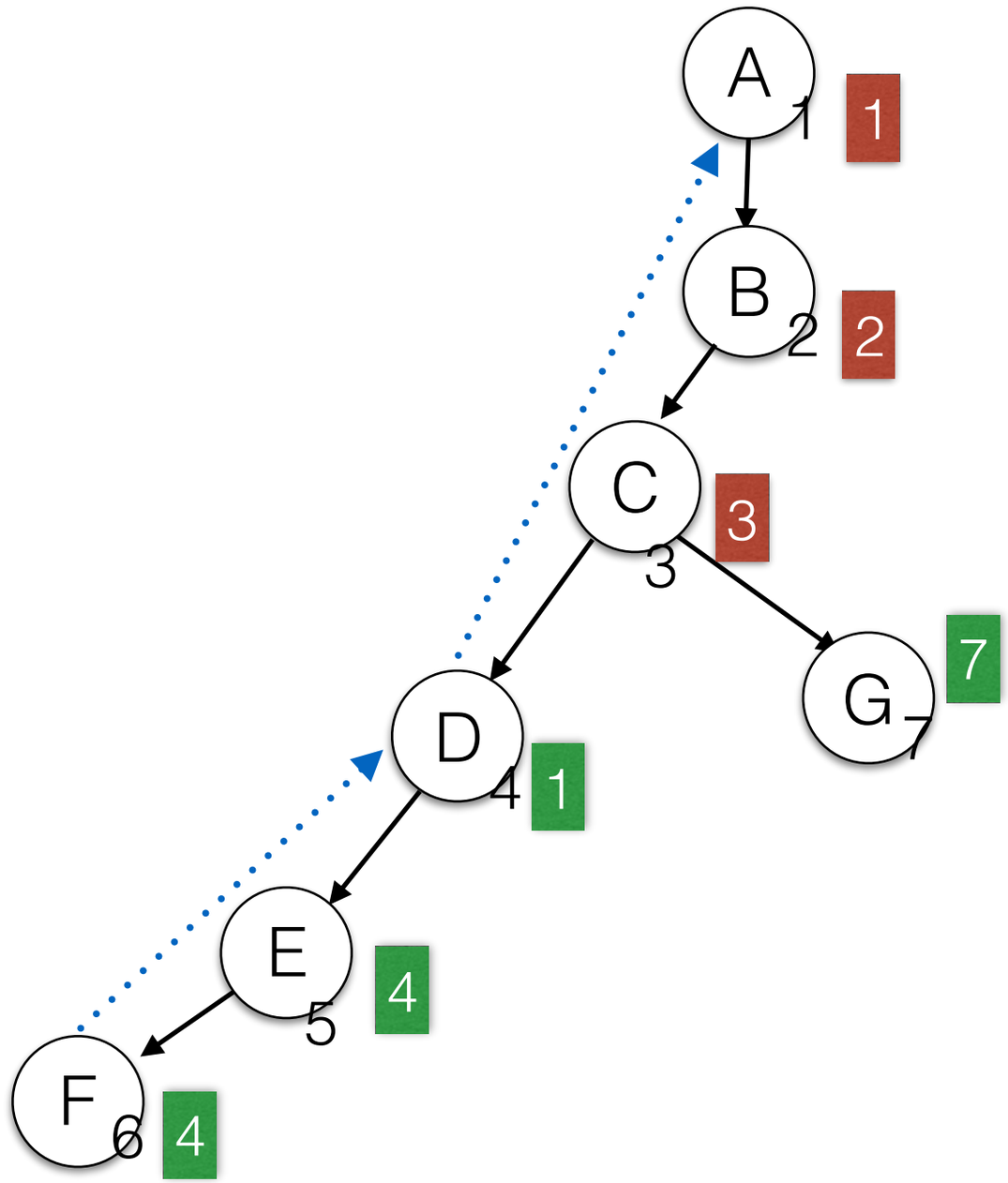
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

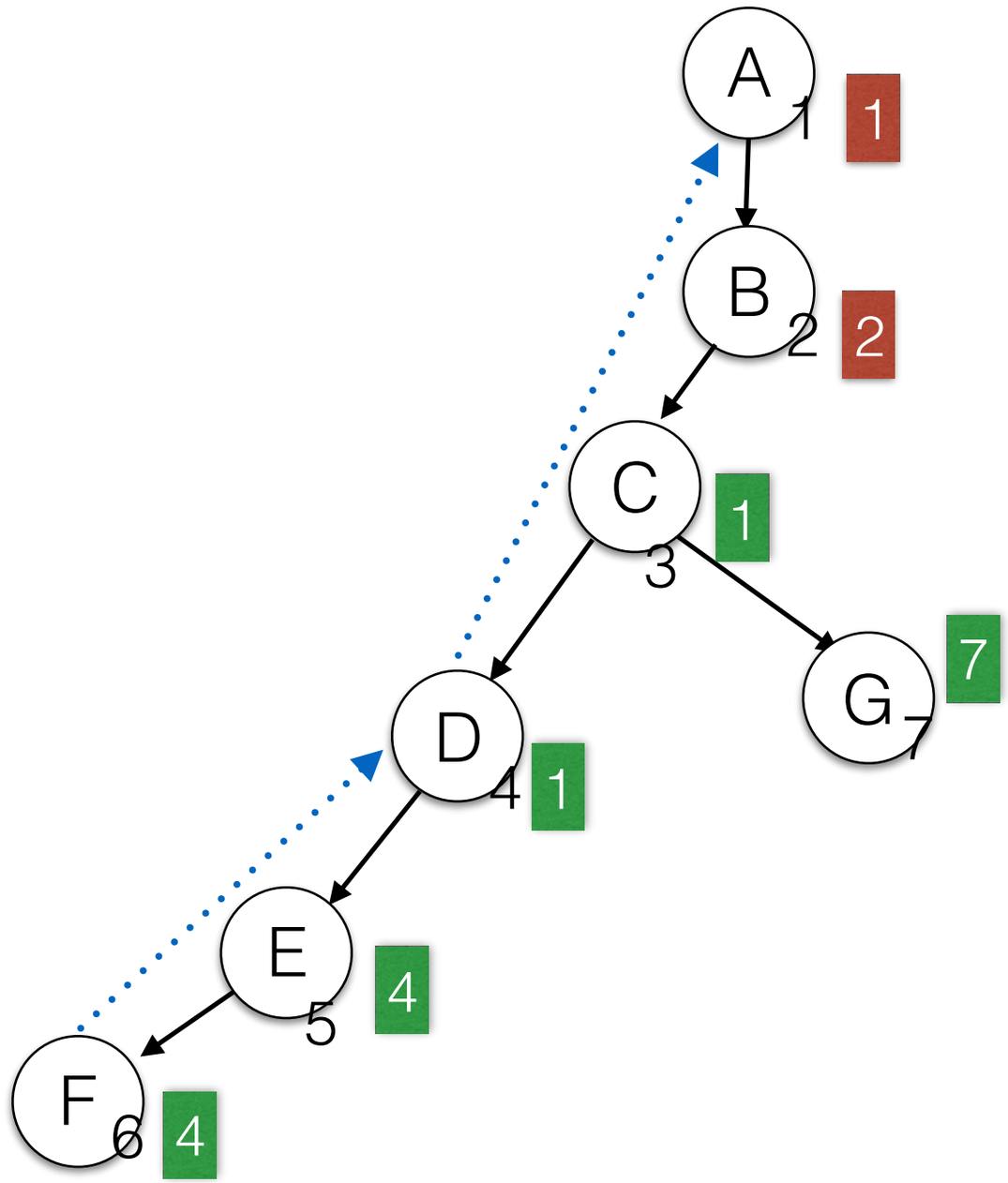
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

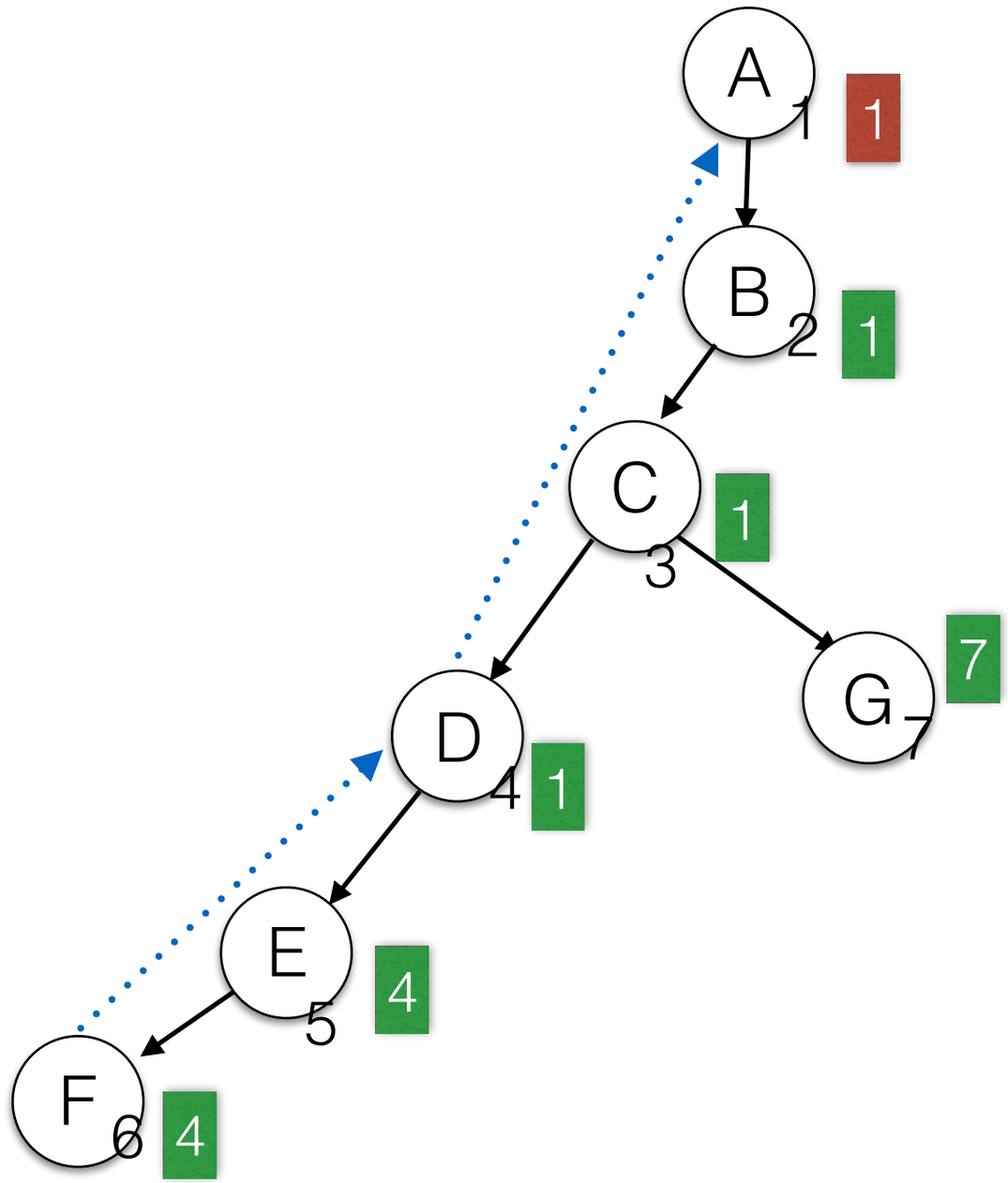
Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

Computing *Low* Numbers Recursively

```
compute_low(v)  {
        Low(v) =  Num(v)
        for all back edges (v,u) {
            if ( Num(u) < Low(v) )
                Low(v) = Num(u);
        }
        for all tree edges (v,u) {
            compute_low(u);
            if ( Low(u) < Low(v) )
                Low(v) = Low(u);
        }
}
```

Computing *Low* Numbers

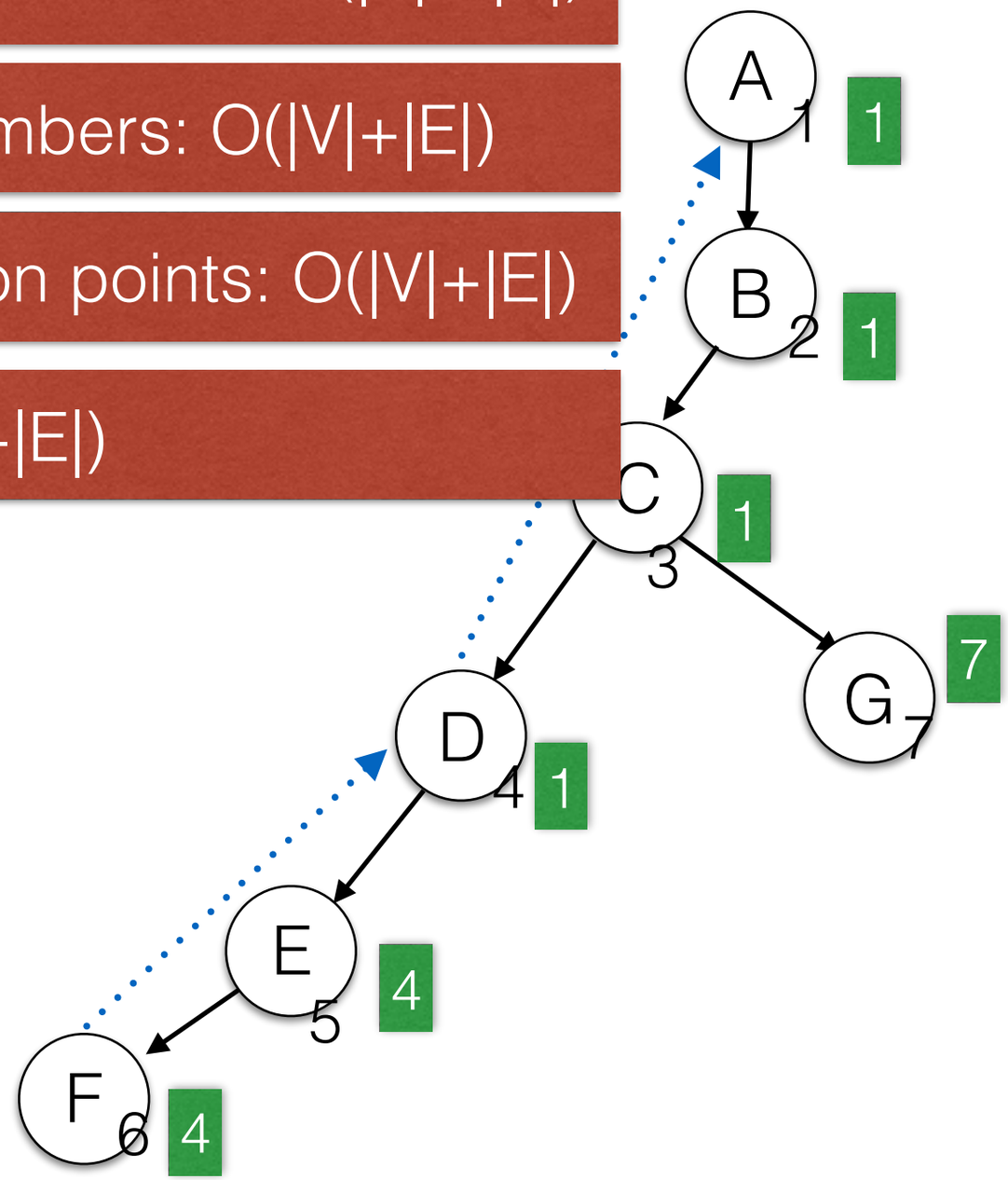
Time to compute preorder numbers: O(|V| +|E|)

Time to compute low numbers: O(|V|+|E|)

Time to check for articulation points: O(|V|+|E|)

Total: O(|V|+|E|)

```
compute
    Lo
    for
        i
    }
    for all tree edges (v,u) {
        compute_low(u);
        if ( Low(u) < Low(v) )
            Low(v) = Low(u);
    }
}
```

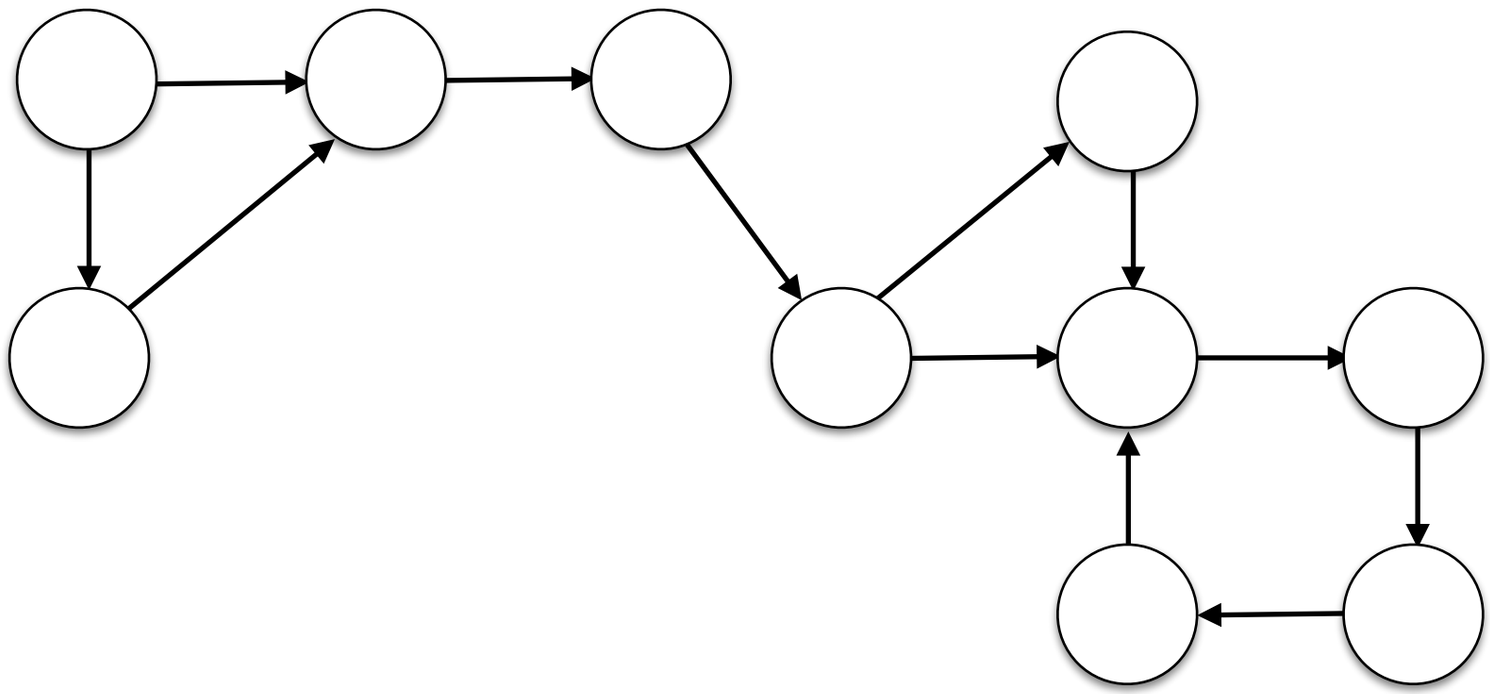


Contents

- Applications of DFS
- Euler Circuits
- Biconnectivity in Undirected Graphs.
- **Finding Strongly Connected Components for Directed Graphs.**

Connectivity in Directed Graphs

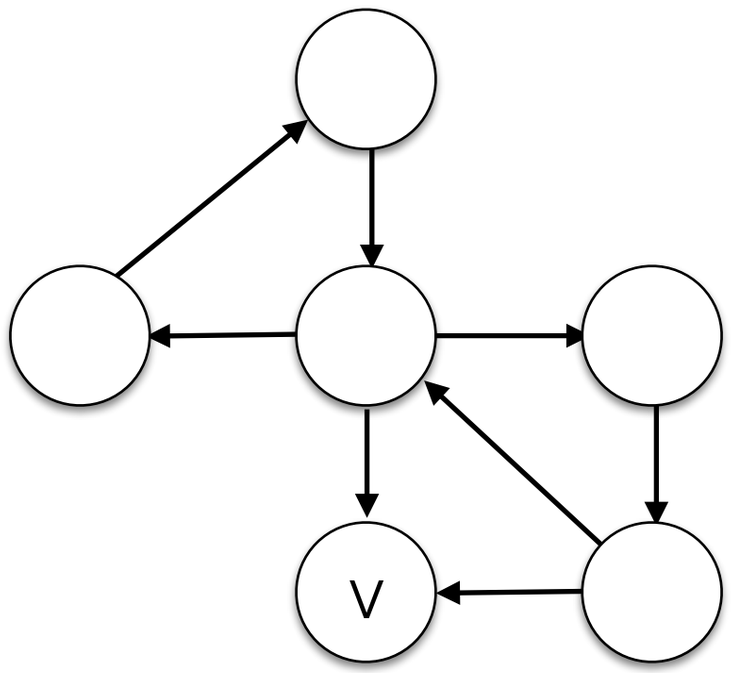
- A directed graph is **weakly connected** if there is an *undirected* path from every vertex to every other vertex.



weakly connected graph

Strongly Connected Graphs

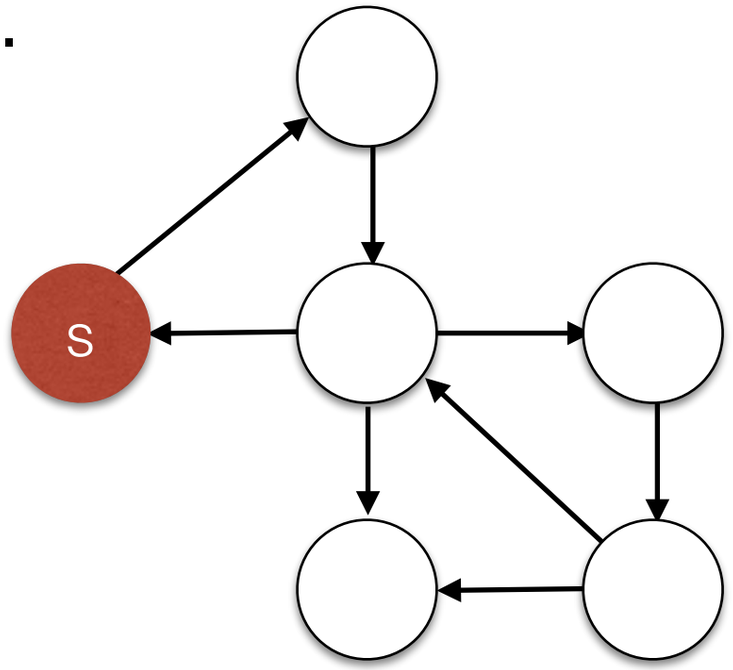
- A directed graph is **strongly connected** if there is a path from every vertex to every other vertex.



Weakly connected, but not strongly connected (no other vertex can be reached from v).

Testing if a Graph is Strongly Connected

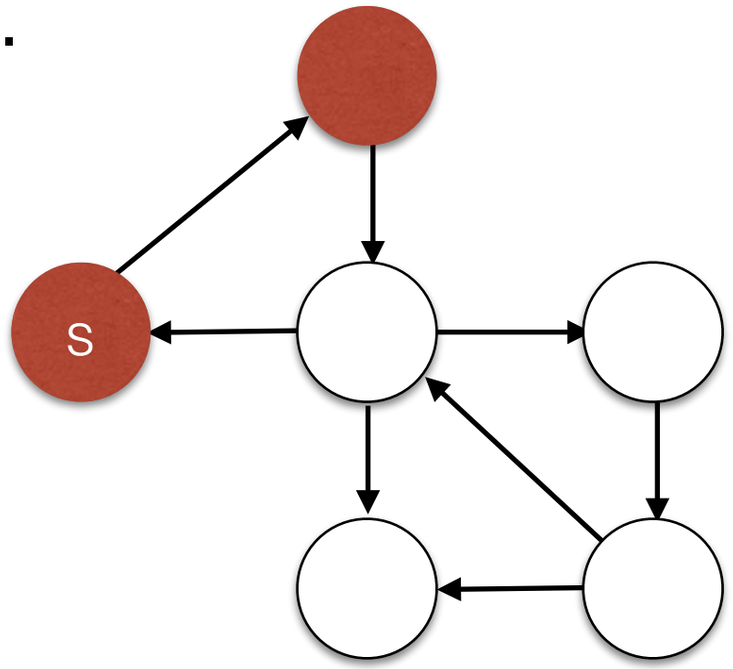
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

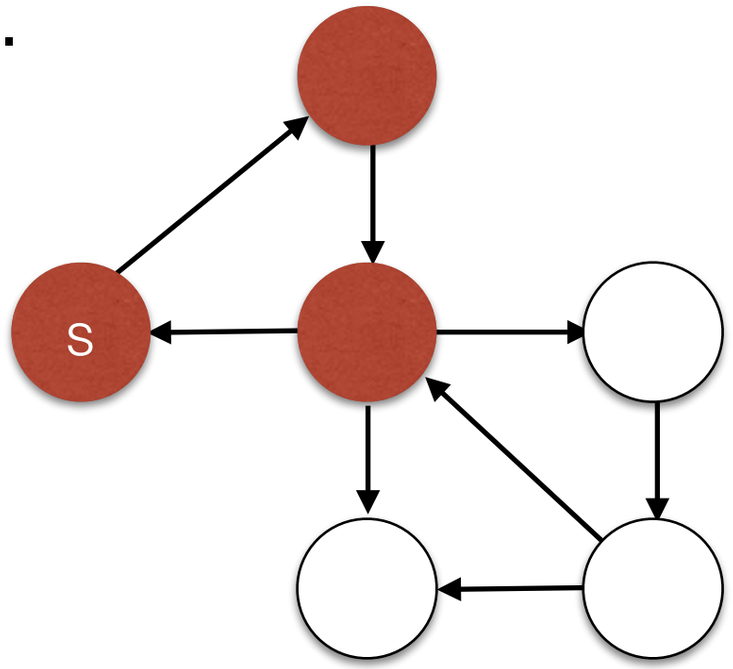
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

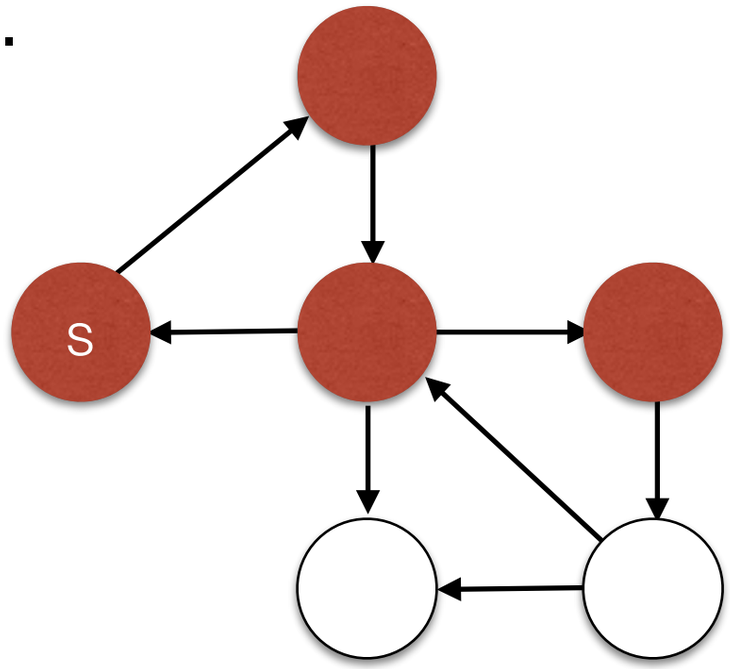
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

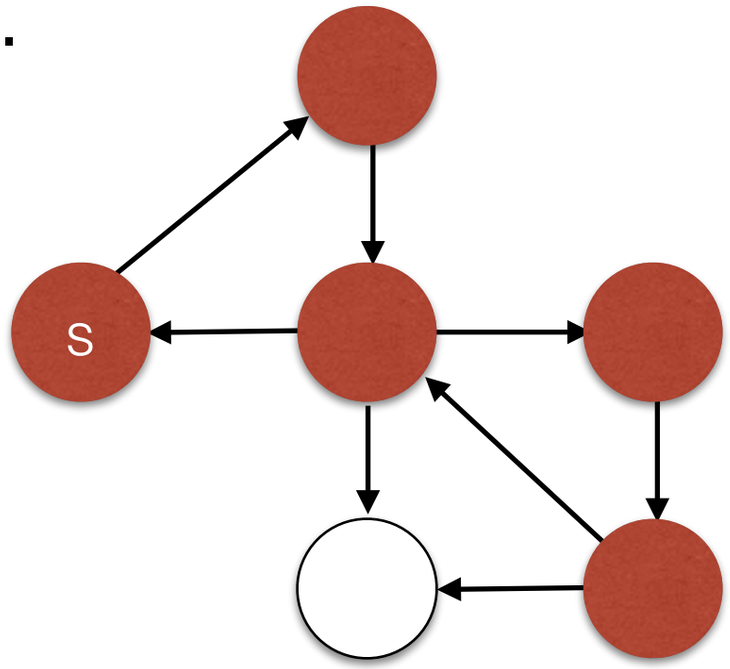
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

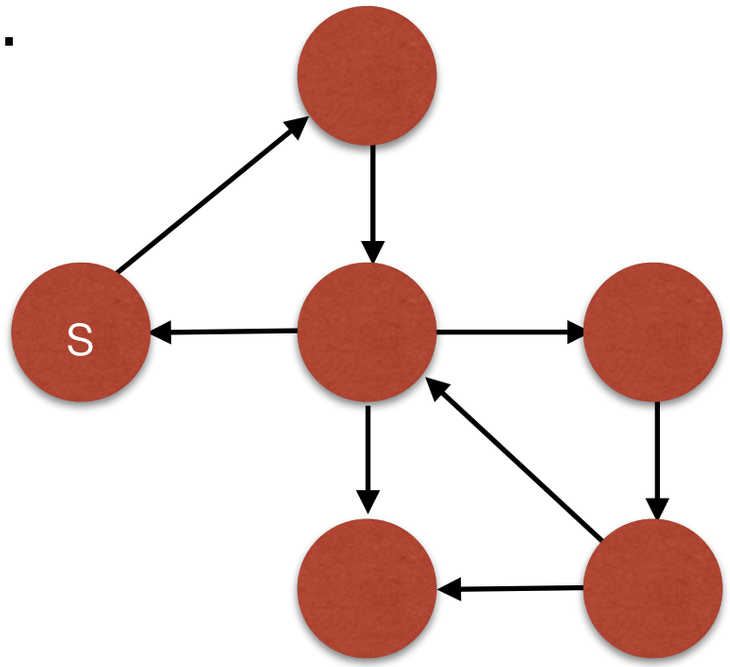
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

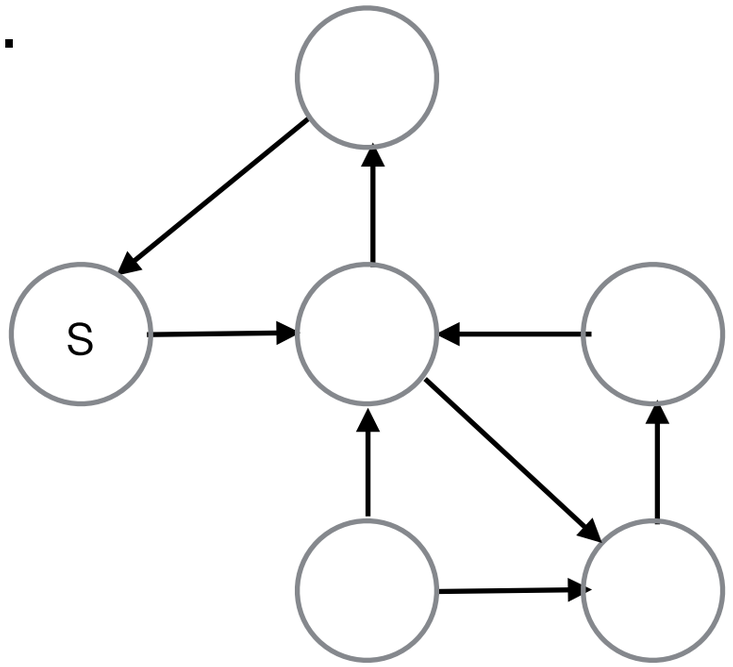
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

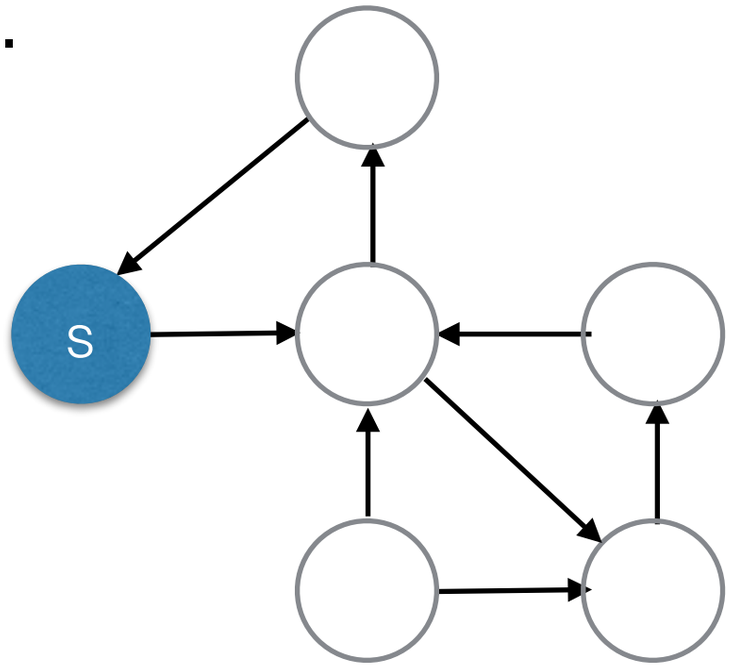
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

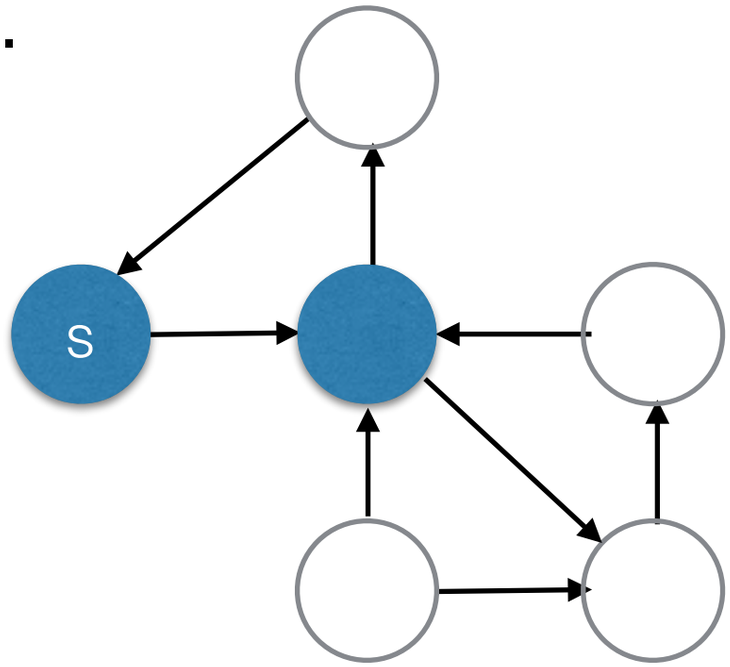
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

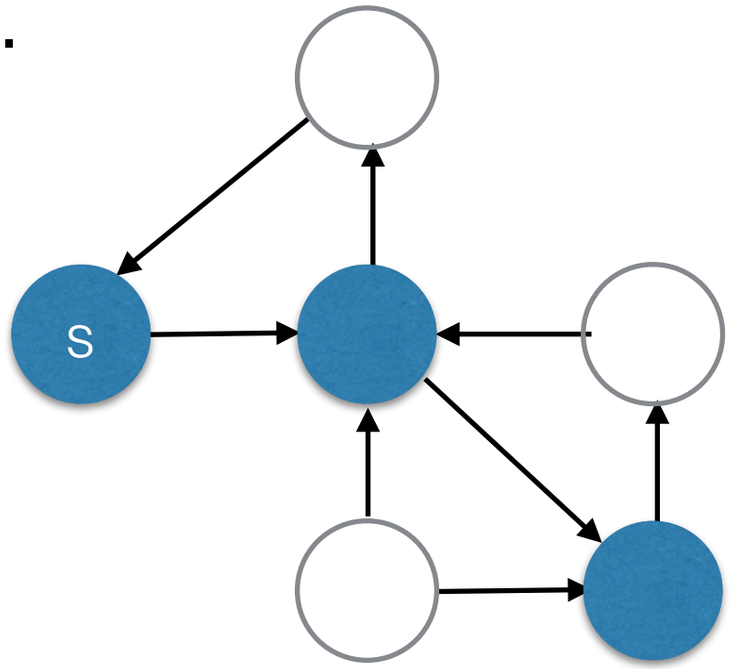
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

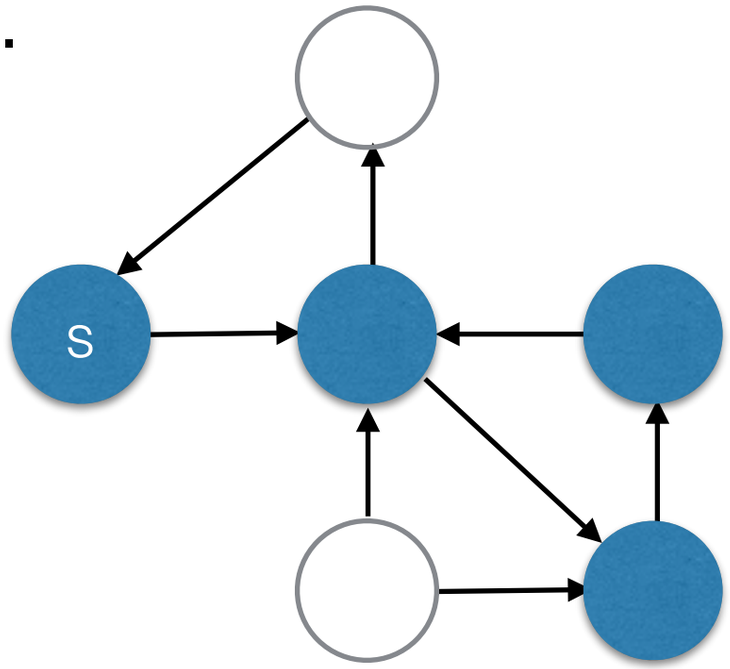
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

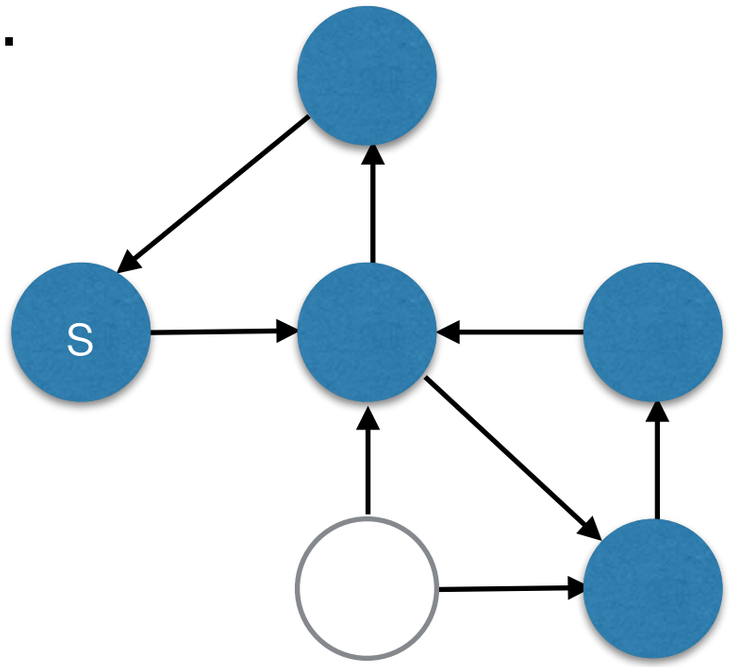
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

Testing if a Graph is Strongly Connected

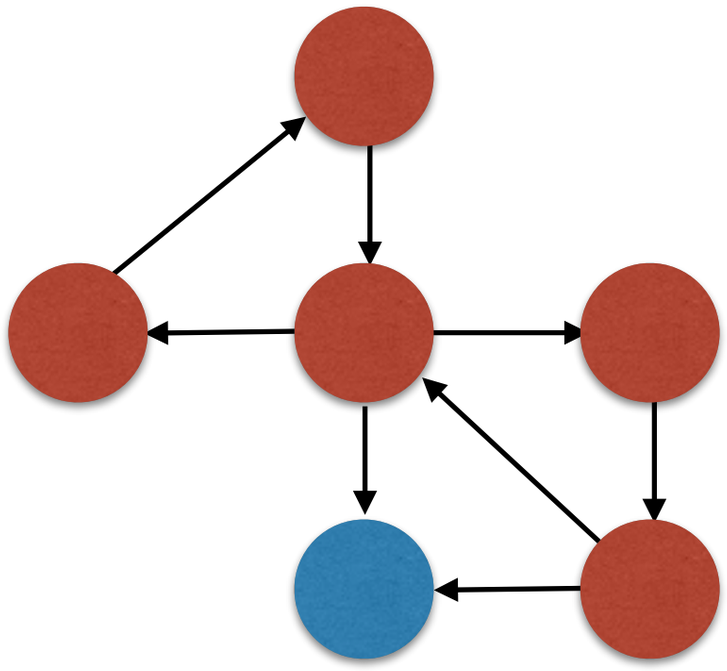
- Run DFS to see if all vertices are reachable from some start node s.



- Reverse direction of edges and run DFS again.

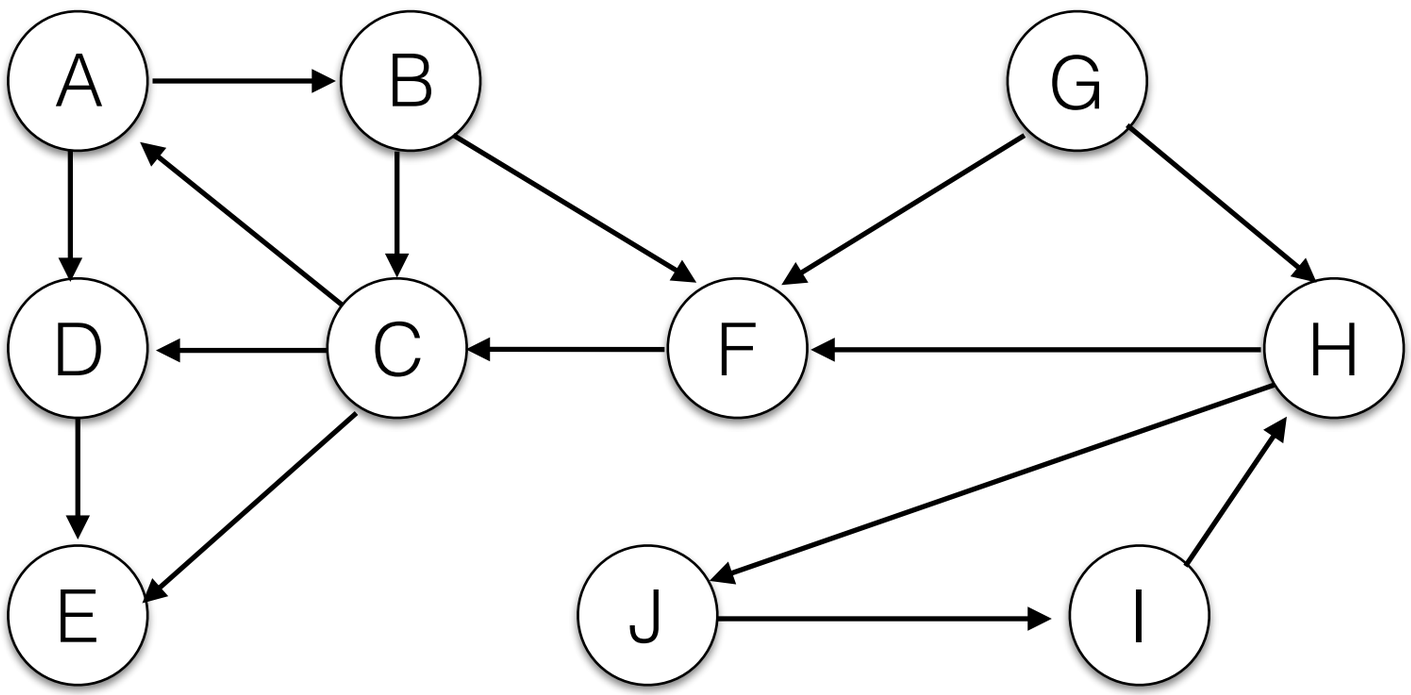
Strongly Connected Components

- Goal: Partition the graph into subgraphs such that each partition is strongly connected.



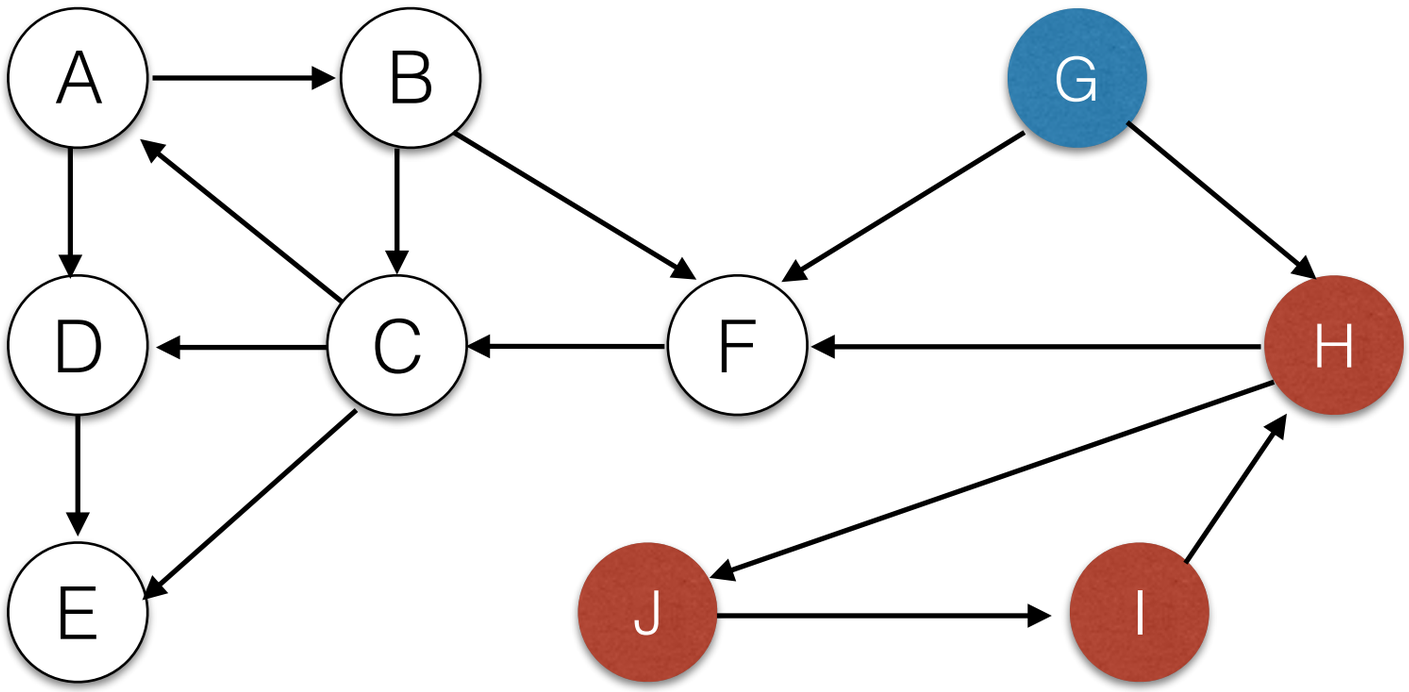
Strongly Connected Components

- Goal: Partition the graph into subgraphs such that each partition is strongly connected.



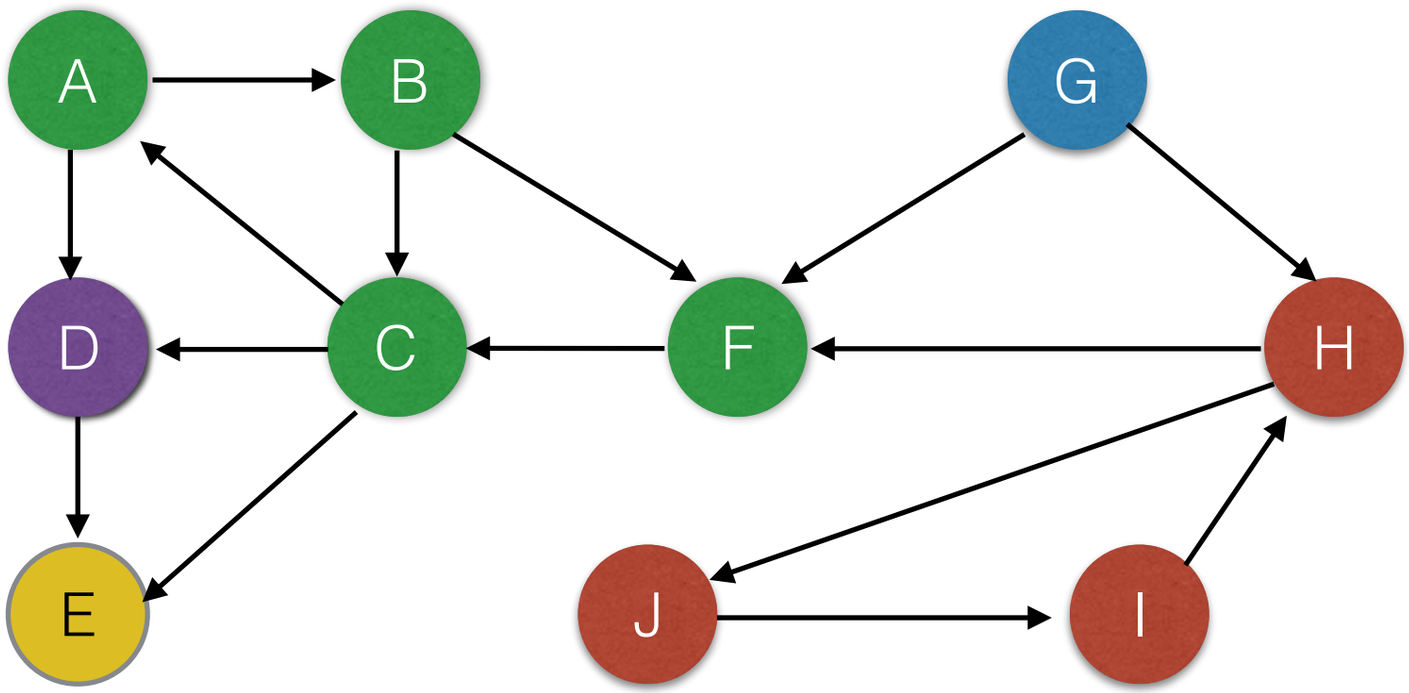
Strongly Connected Components

- Goal: Partition the graph into subgraphs such that each partition is strongly connected.



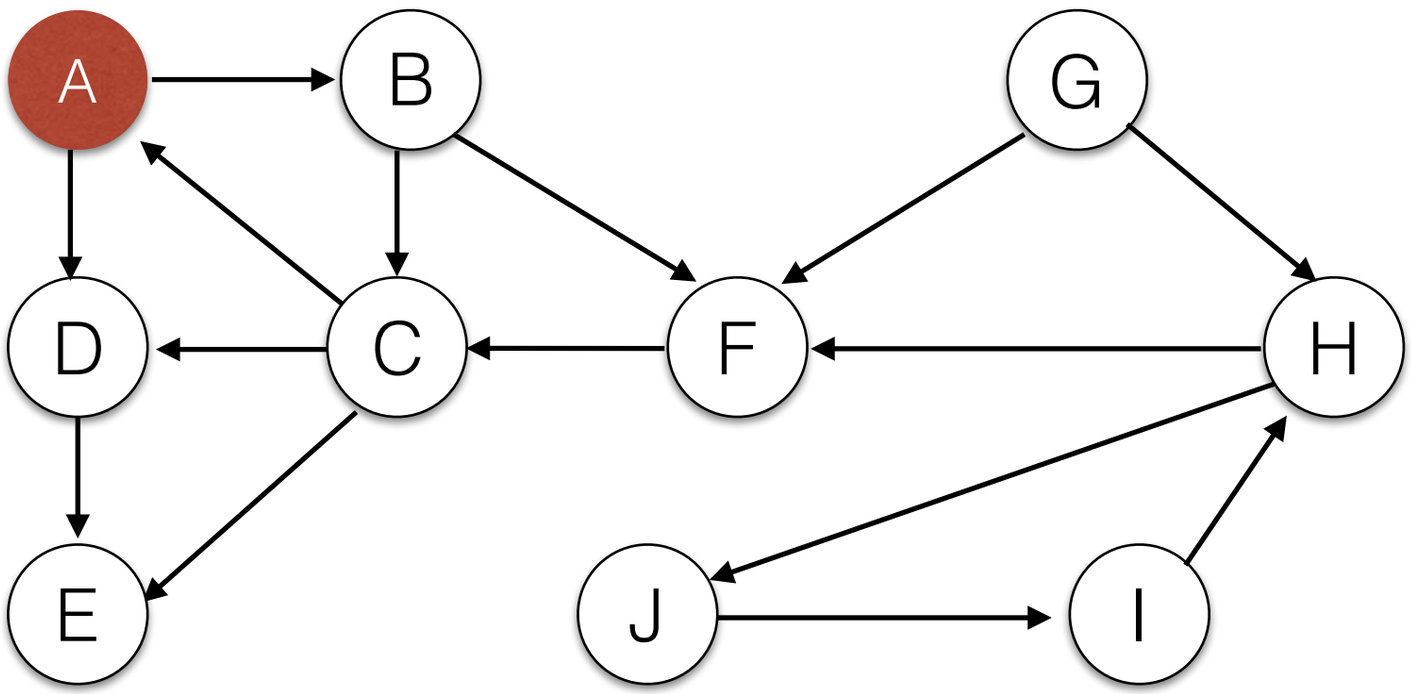
Strongly Connected Components

- Goal: Partition the graph into subgraphs such that each partition is strongly connected.



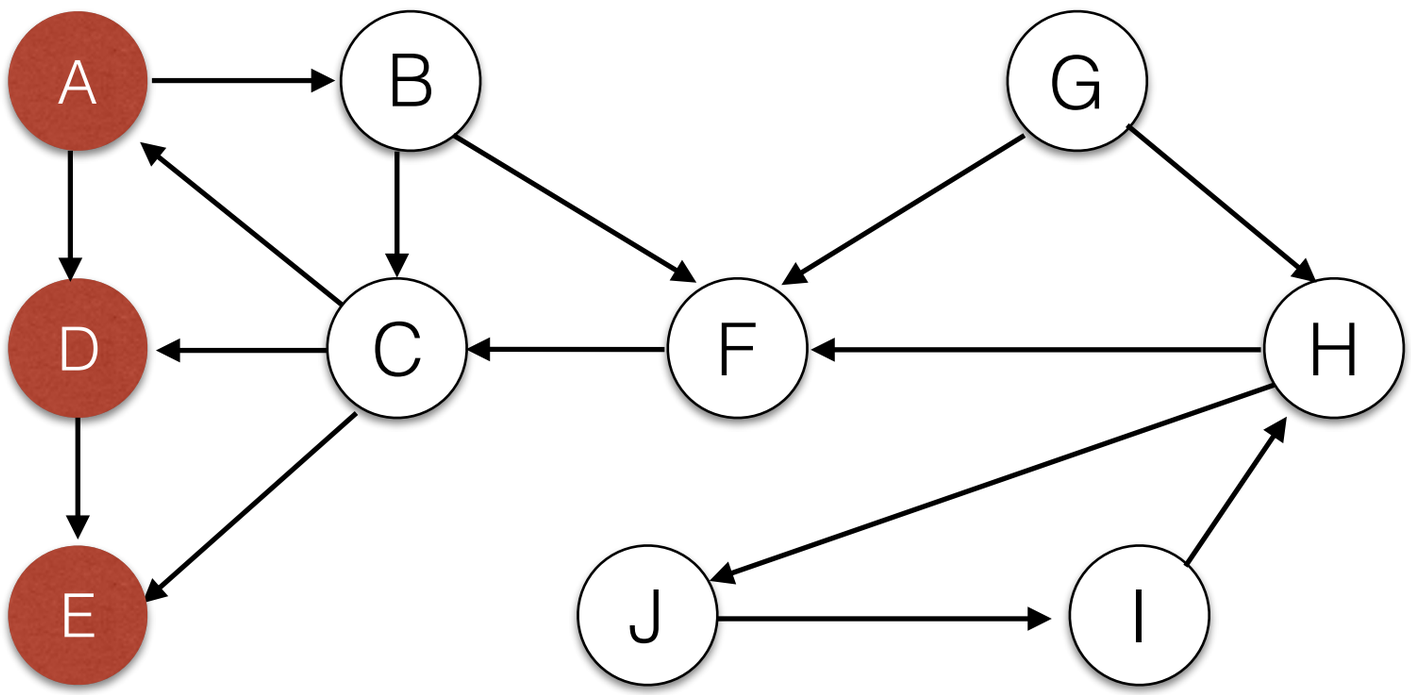
Finding Strongly Connected Components

- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



Finding Strongly Connected Components

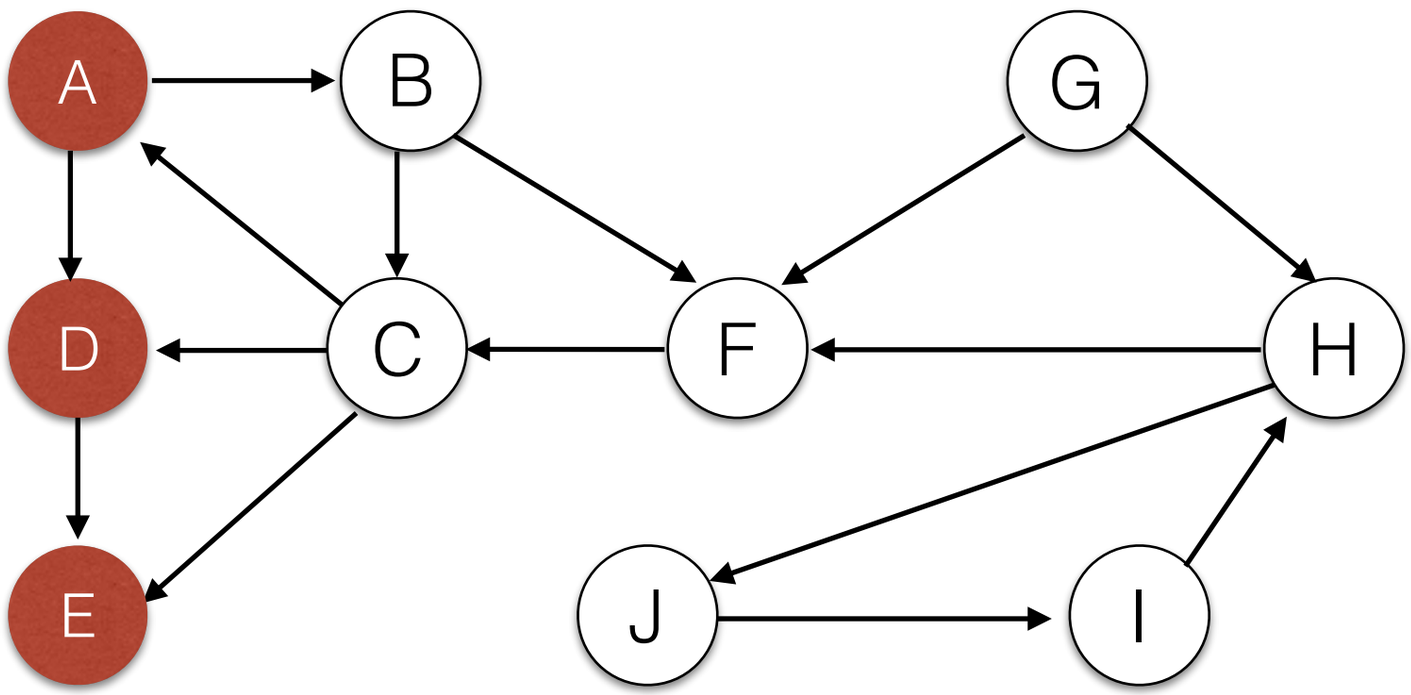
- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



E

Finding Strongly Connected Components

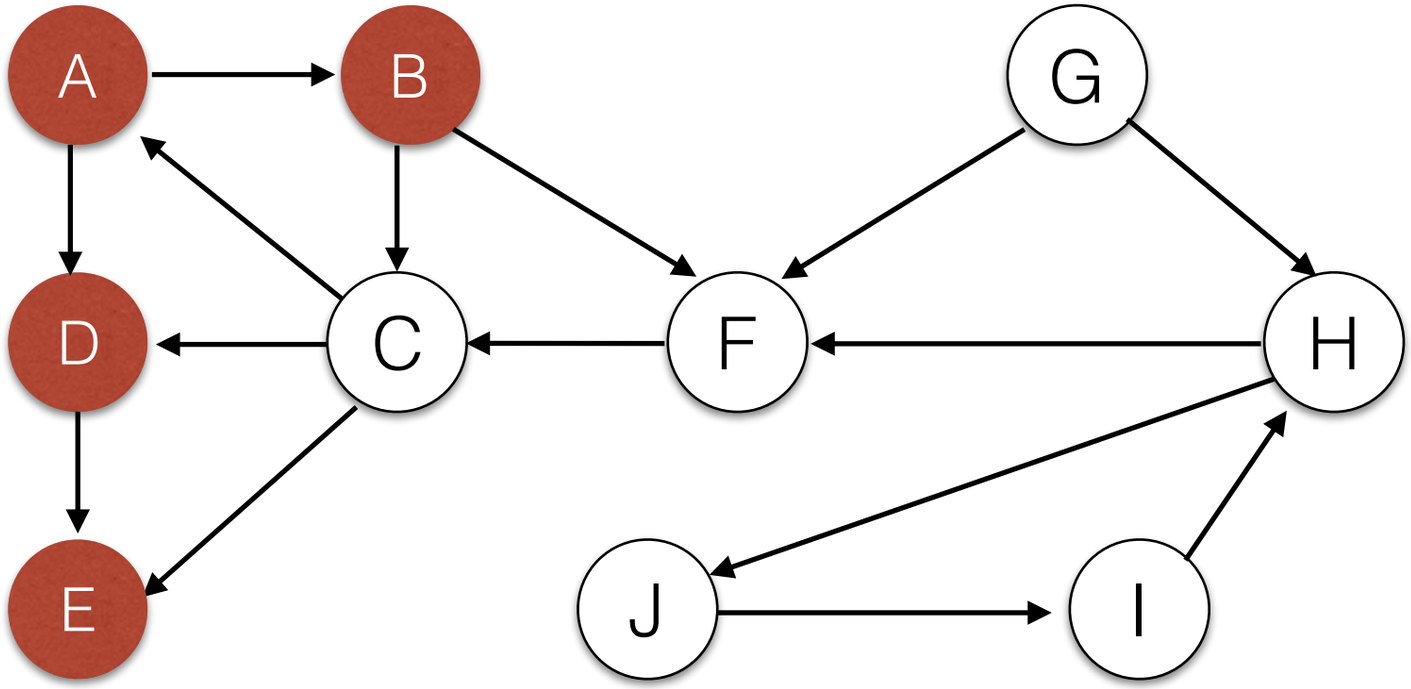
- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



D
E

Finding Strongly Connected Components

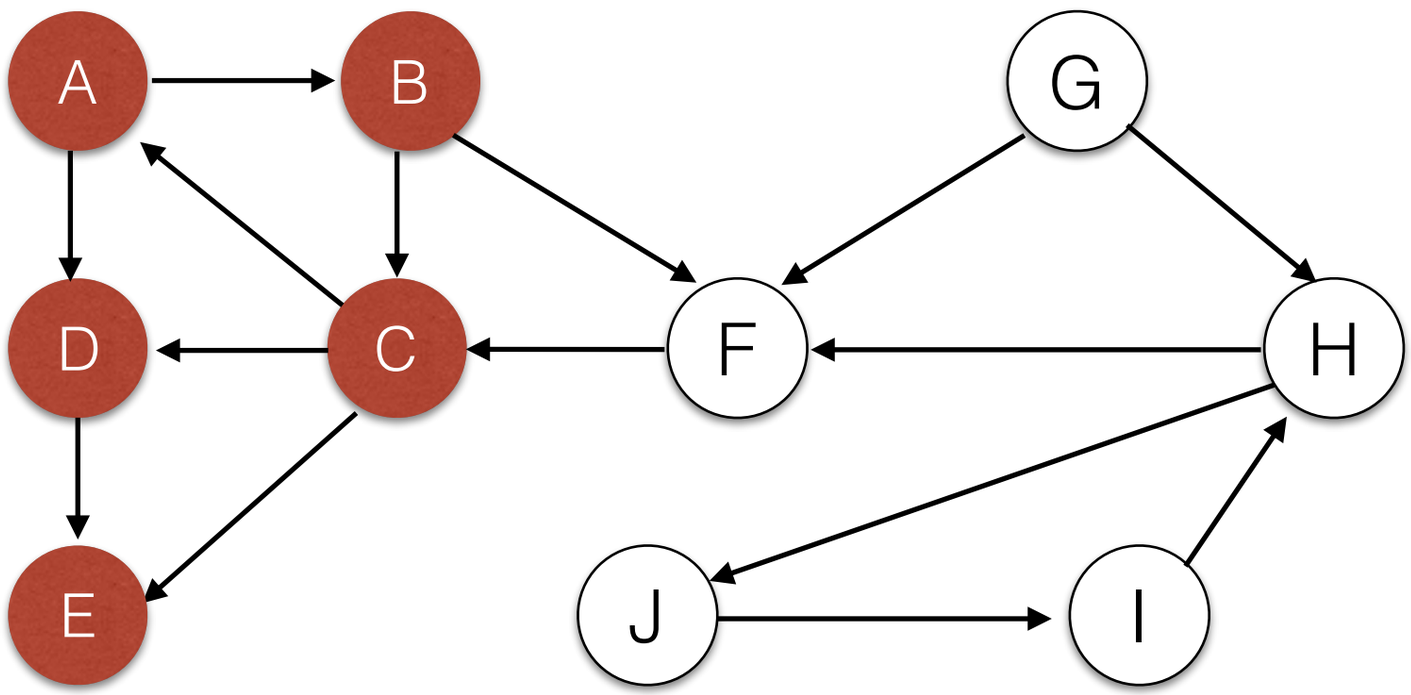
- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



A
D
E

Finding Strongly Connected Components

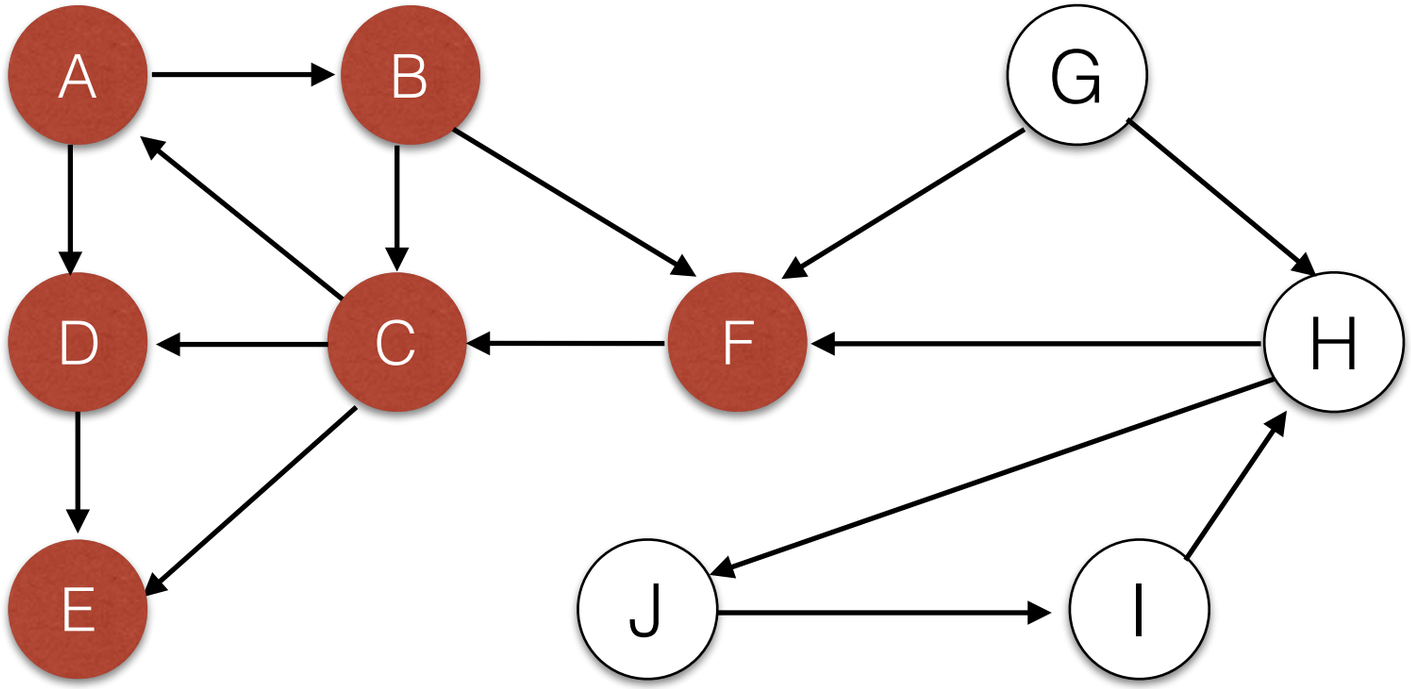
- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



C
A
D
E

Finding Strongly Connected Components

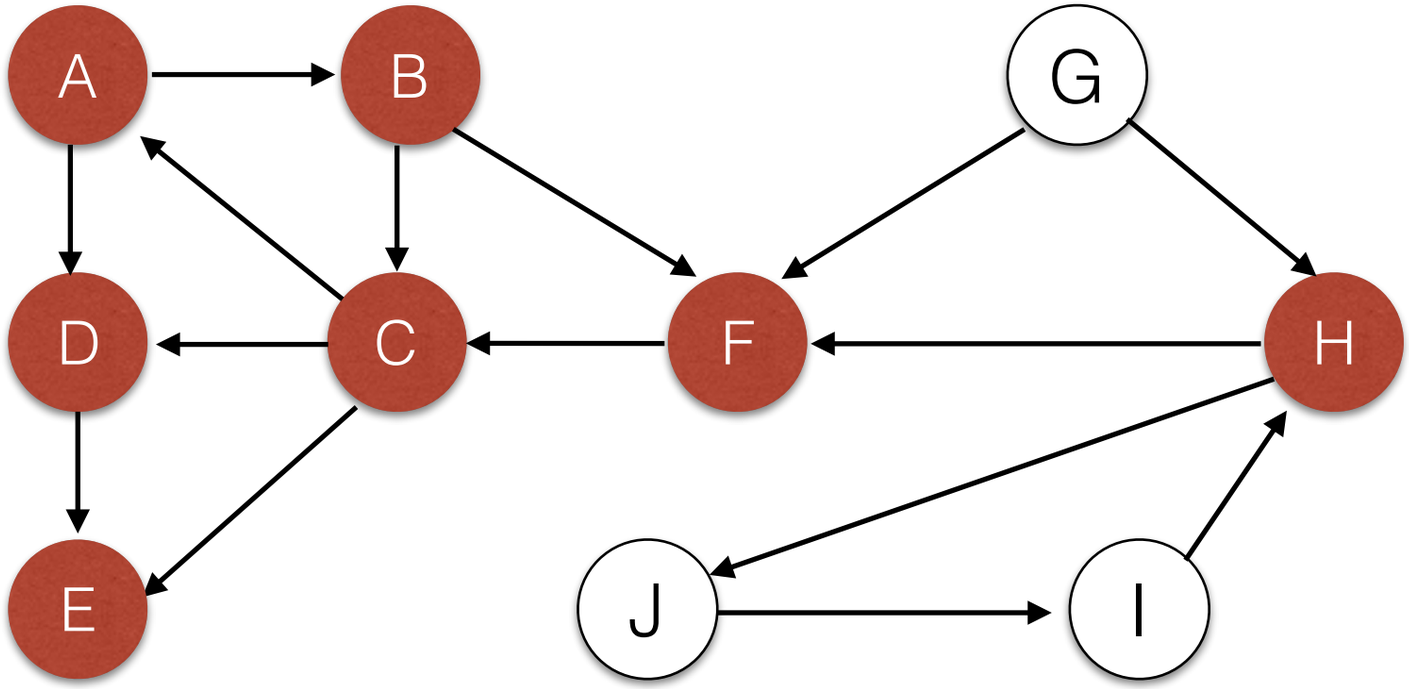
- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



F
B
C
A
D
E

Finding Strongly Connected Components

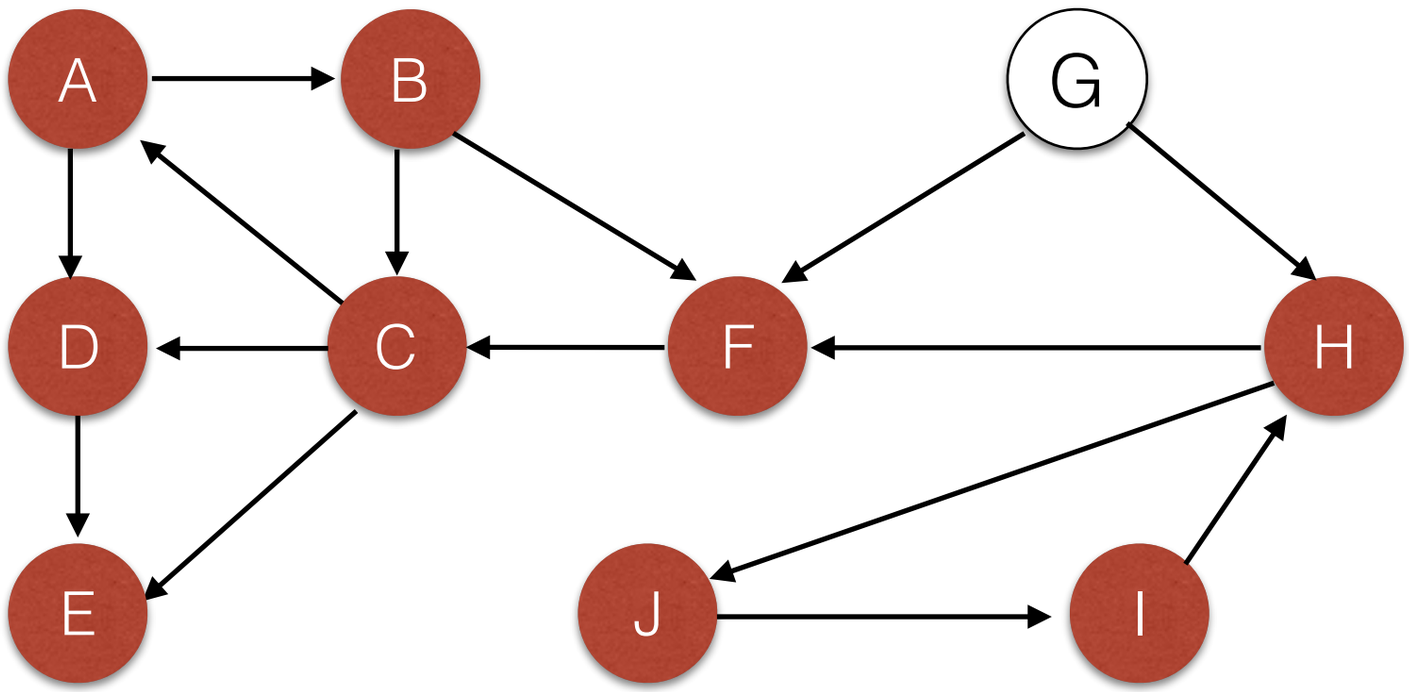
- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



F
B
C
A
D
E

Finding Strongly Connected Components

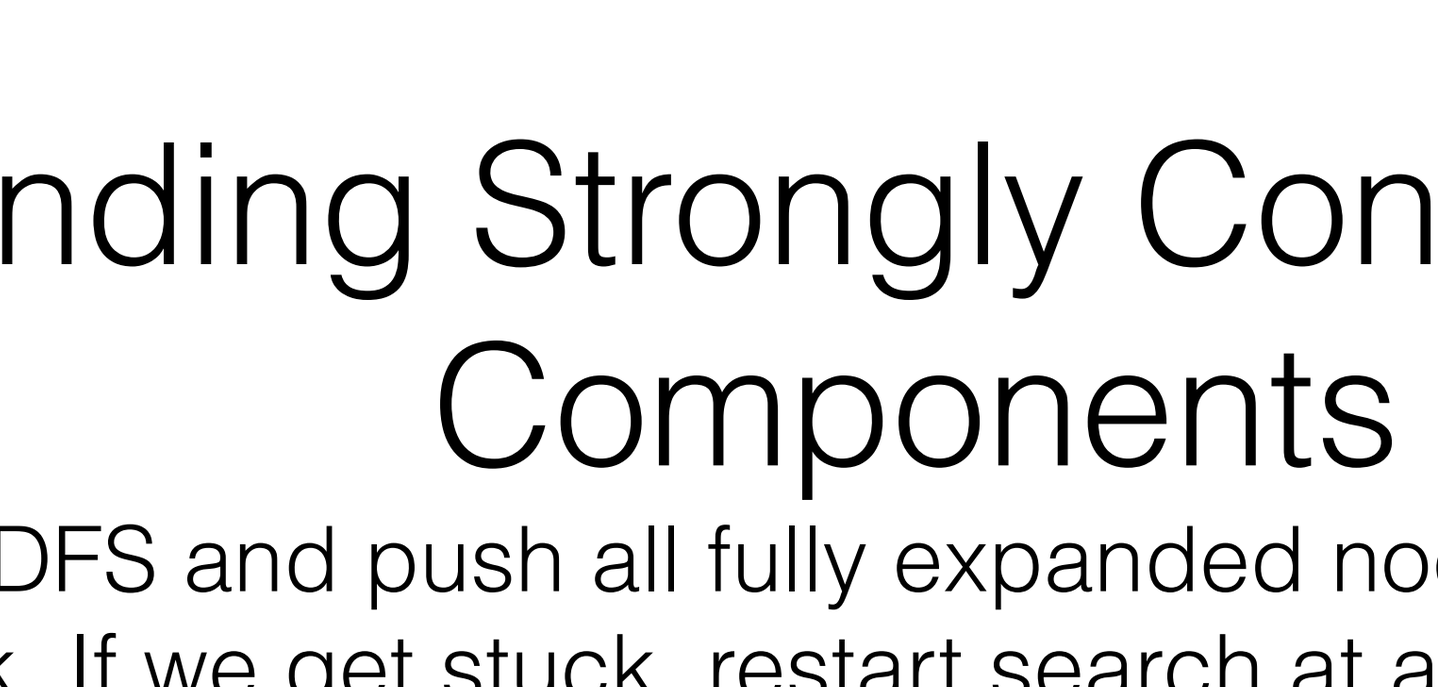
- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



J
F
B
C
A
D
E

Finding Strongly Connected Components

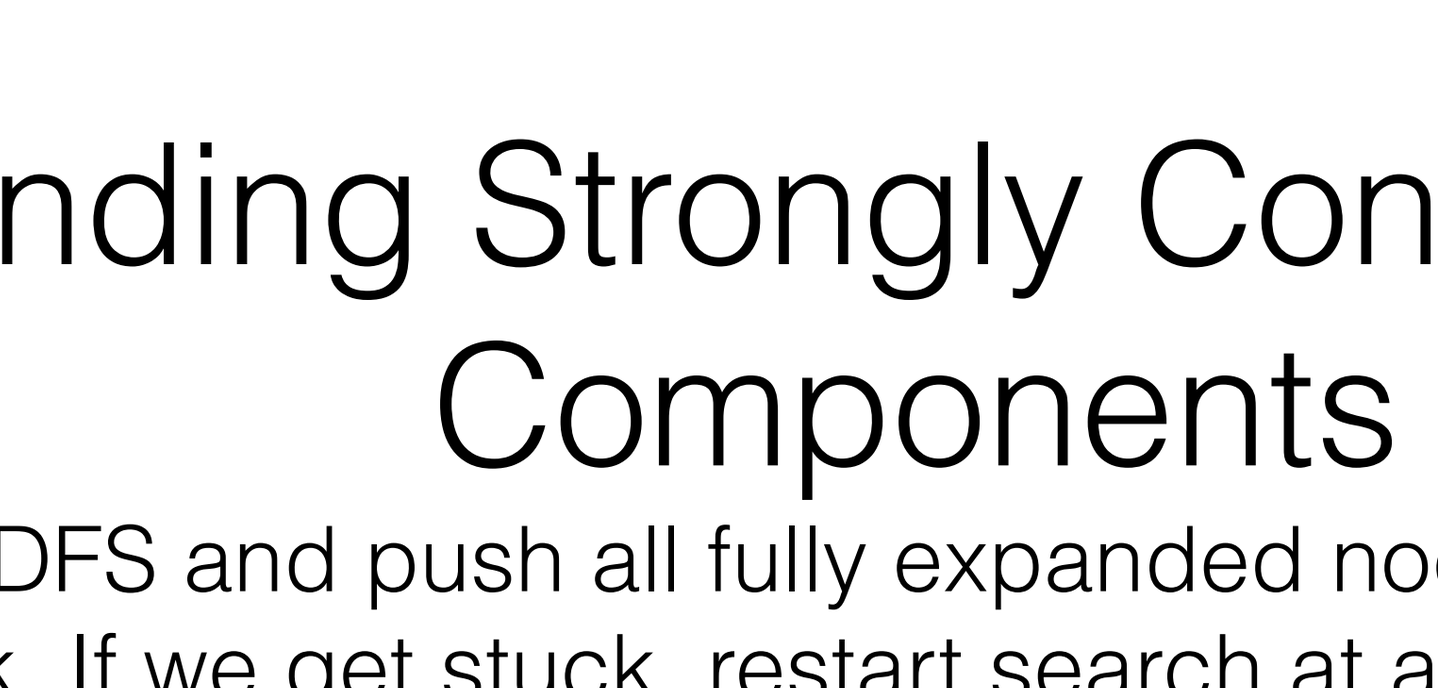
- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



H
I
J
F
B
C
A
D
E

Finding Strongly Connected Components

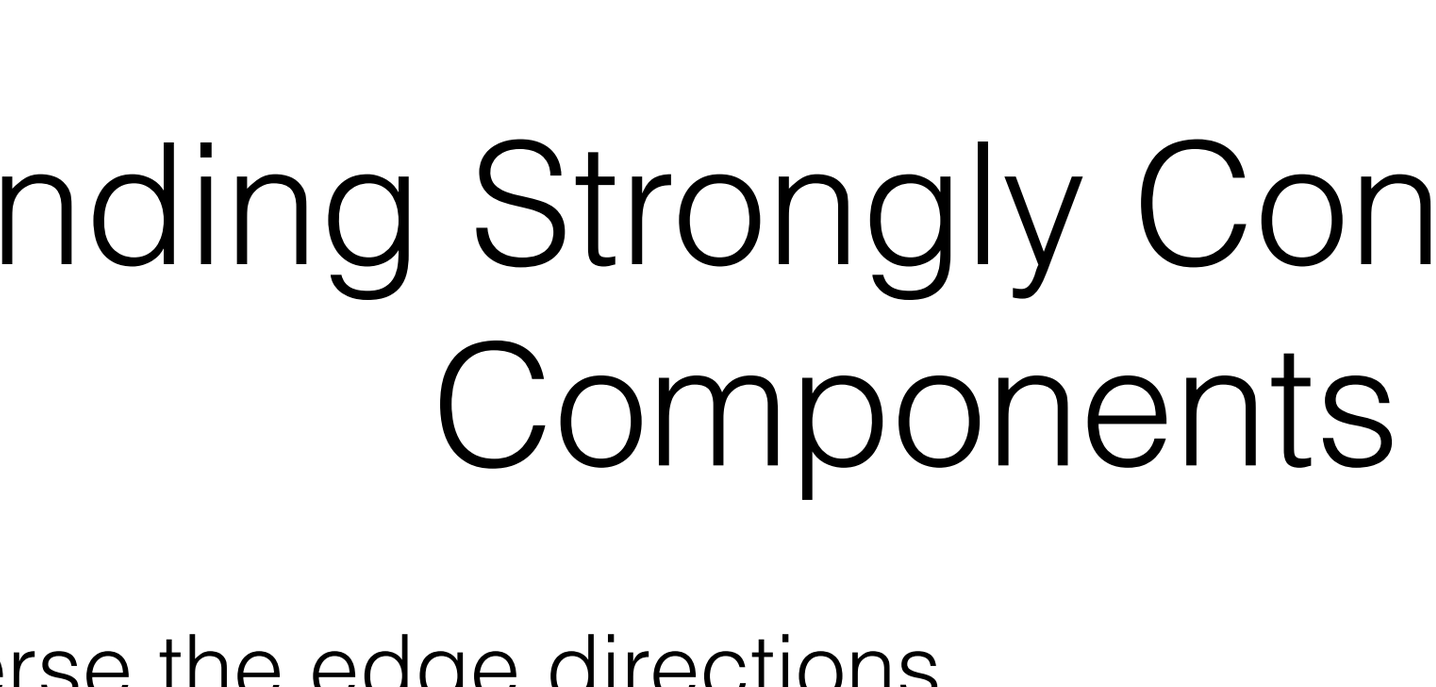
- Run DFS and push all fully expanded nodes on a stack. If we get stuck, restart search at an arbitrary unvisited node.



G
H
I
J
F
B
C
A
D
E

Finding Strongly Connected Components

- Reverse the edge directions.



G
H
I
J
F
B
C
A
D
E

Finding Strongly Connected Components

- Pop the top vertex off the stack and run DFS. The set of visited nodes are a strong component. Remove this component from the graph and stack.

- Continue until stack is empty.

G
H
I
J
F
B
C
A
D
E

Finding Strongly Connected Components

- Pop the top vertex off the stack and run DFS. The set of visited nodes are a strong component. Remove this component from the graph and stack.

- Continue until stack is empty.

H
I
J
F
B
C
A
D
E

Finding Strongly Connected Components

- Pop the top vertex off the stack and run DFS. The set of visited nodes are a strong component. Remove this component from the graph and stack.

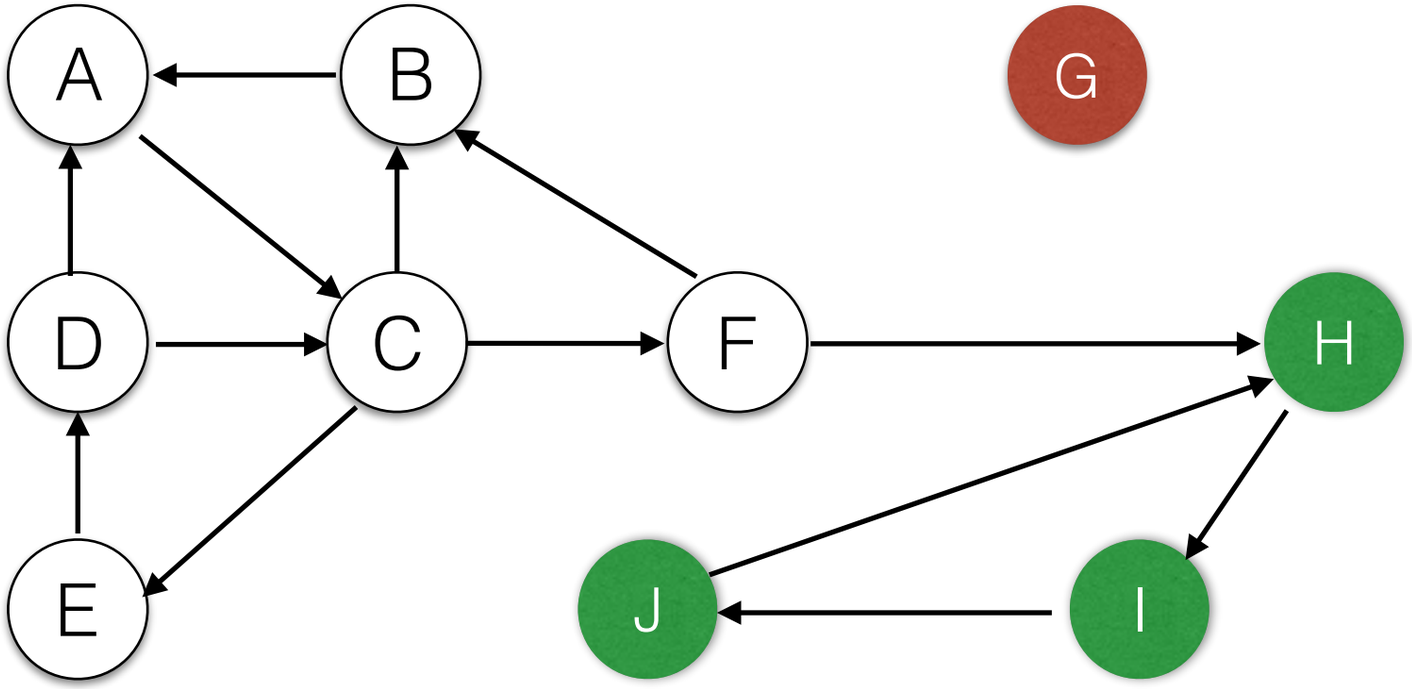
- Continue until stack is empty.

H
I
J
F
B
C
A
D
E

Finding Strongly Connected Components

- Pop the top vertex off the stack and run DFS. The set of visited nodes are a strong component. Remove this component from the graph and stack.

- Continue until stack is empty.

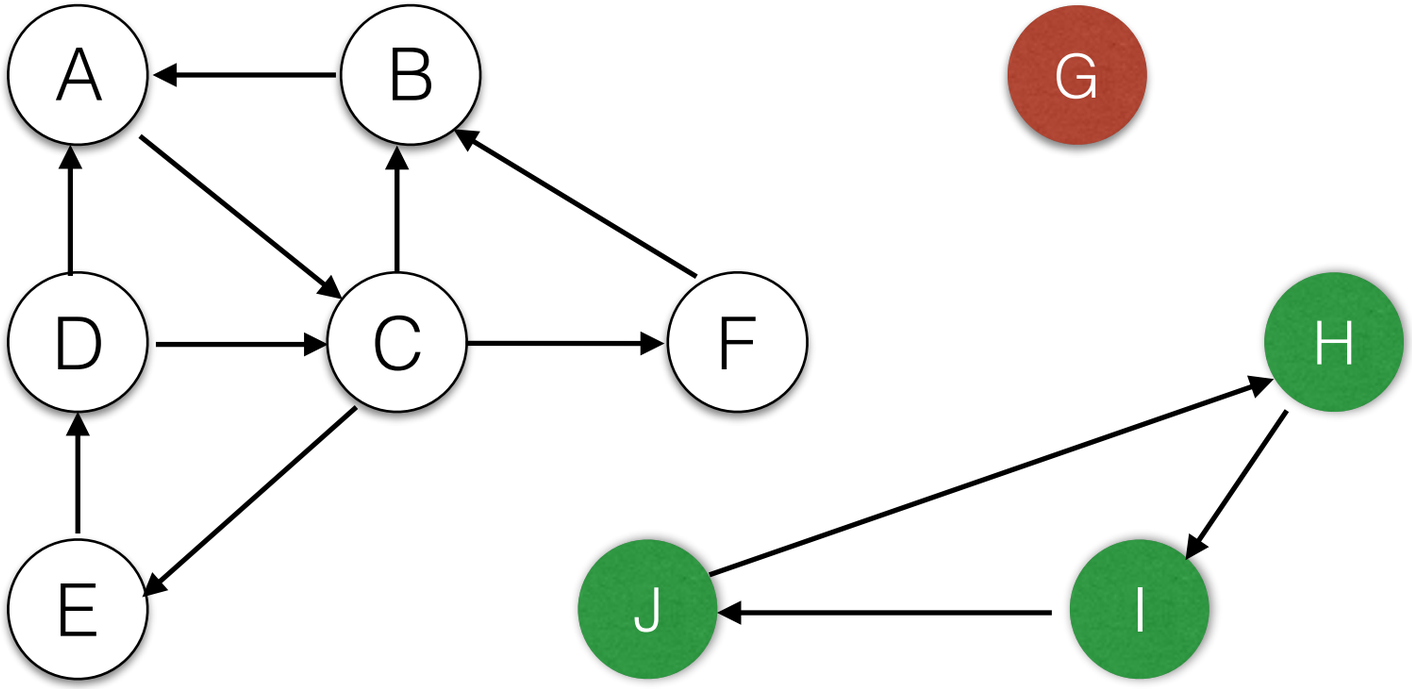


I
J
F
B
C
A
D
E

Finding Strongly Connected Components

- Pop the top vertex off the stack and run DFS. The set of visited nodes are a strong component. Remove this component from the graph and stack.

- Continue until stack is empty.

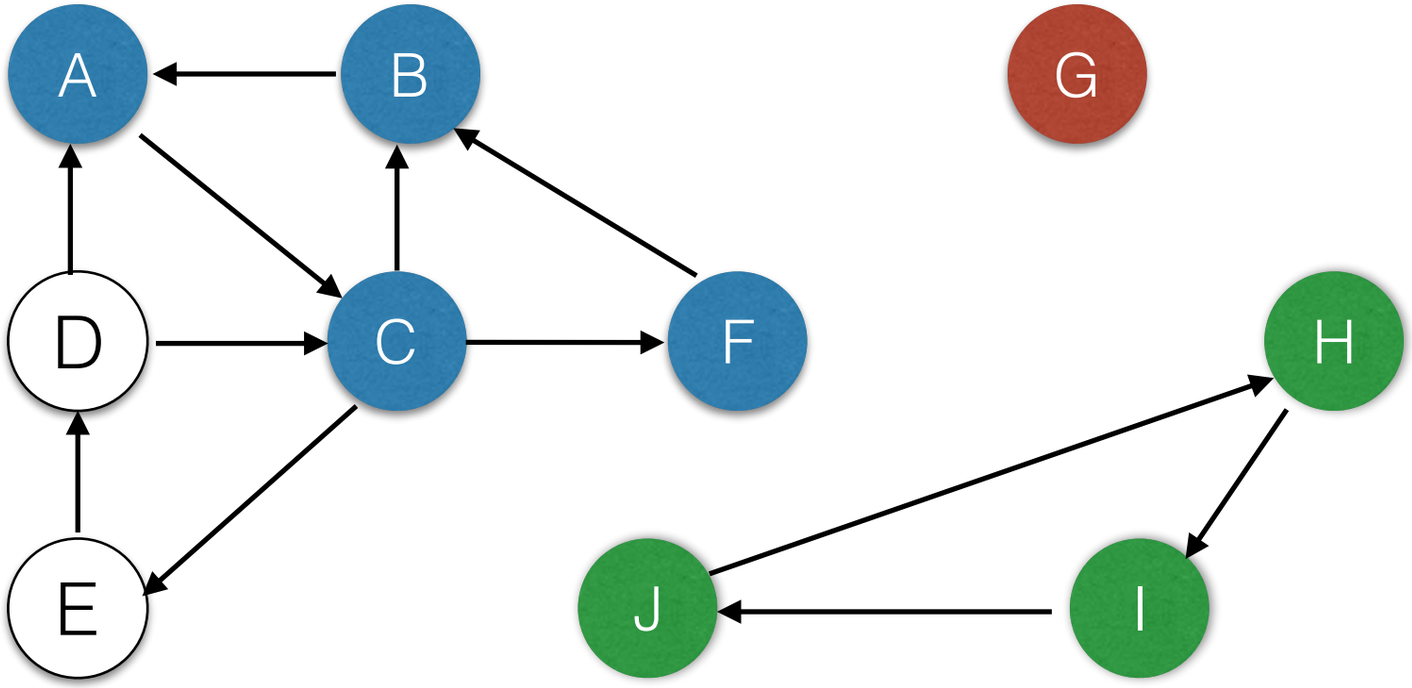


F
B
C
A
D
E

Finding Strongly Connected Components

- Pop the top vertex off the stack and run DFS. The set of visited nodes are a strong component. Remove this component from the graph and stack.

- Continue until stack is empty.

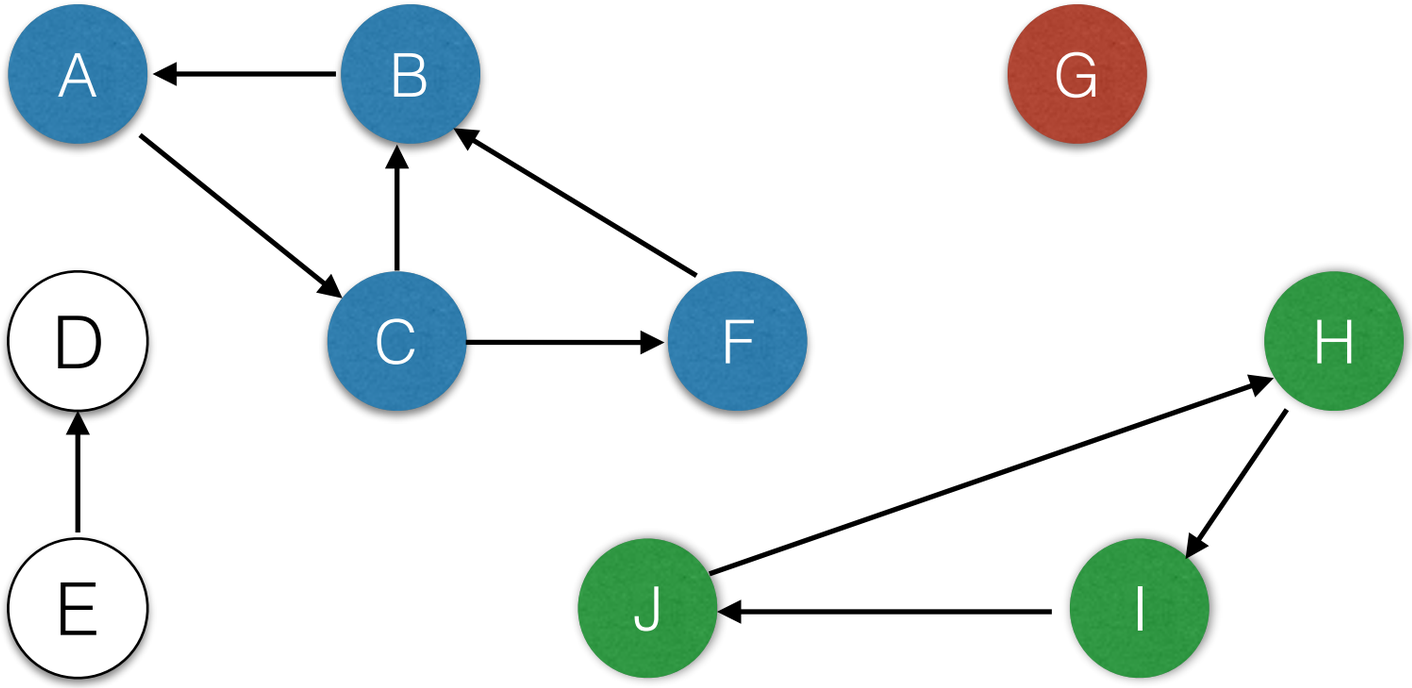


B
C
A
D
E

Finding Strongly Connected Components

- Pop the top vertex off the stack and run DFS. The set of visited nodes are a strong component. Remove this component from the graph and stack.

- Continue until stack is empty.

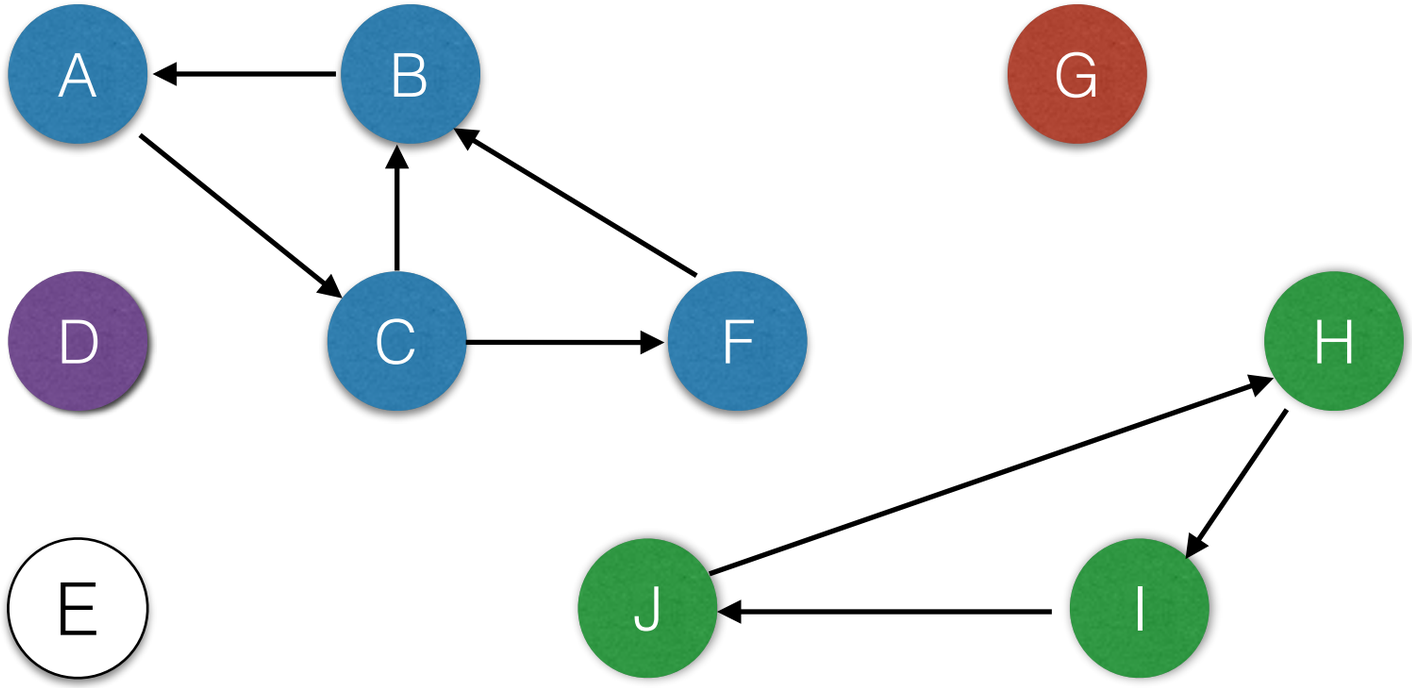


D
E

Finding Strongly Connected Components

- Pop the top vertex off the stack and run DFS. The set of visited nodes are a strong component. Remove this component from the graph and stack.

- Continue until stack is empty.



E