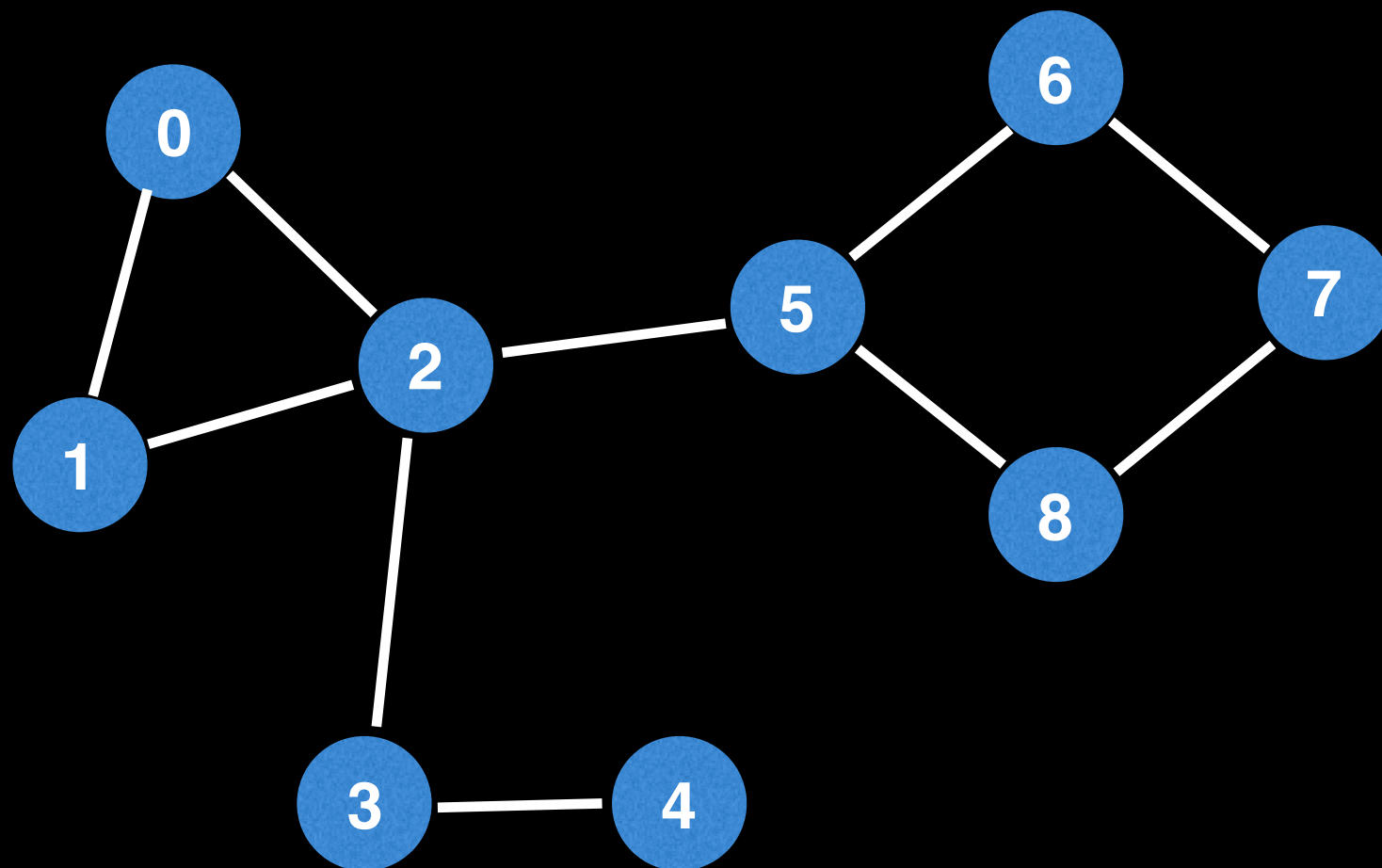


Algorithm to Find Bridges and Articulation Points

William Fiset

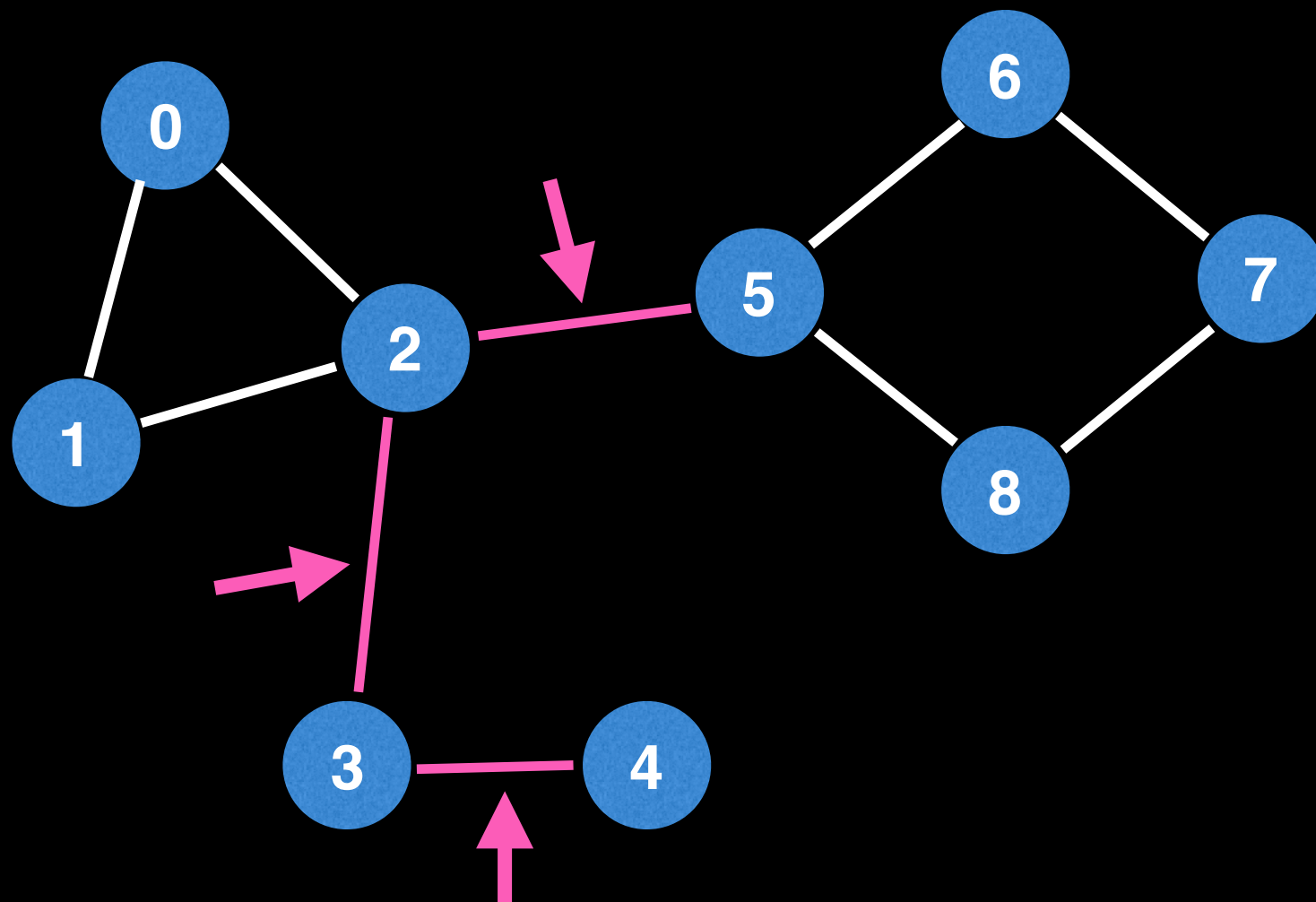
What are bridges & articulation points?

A **bridge / cut edge** is any edge in a graph whose removal increases the number of connected components.



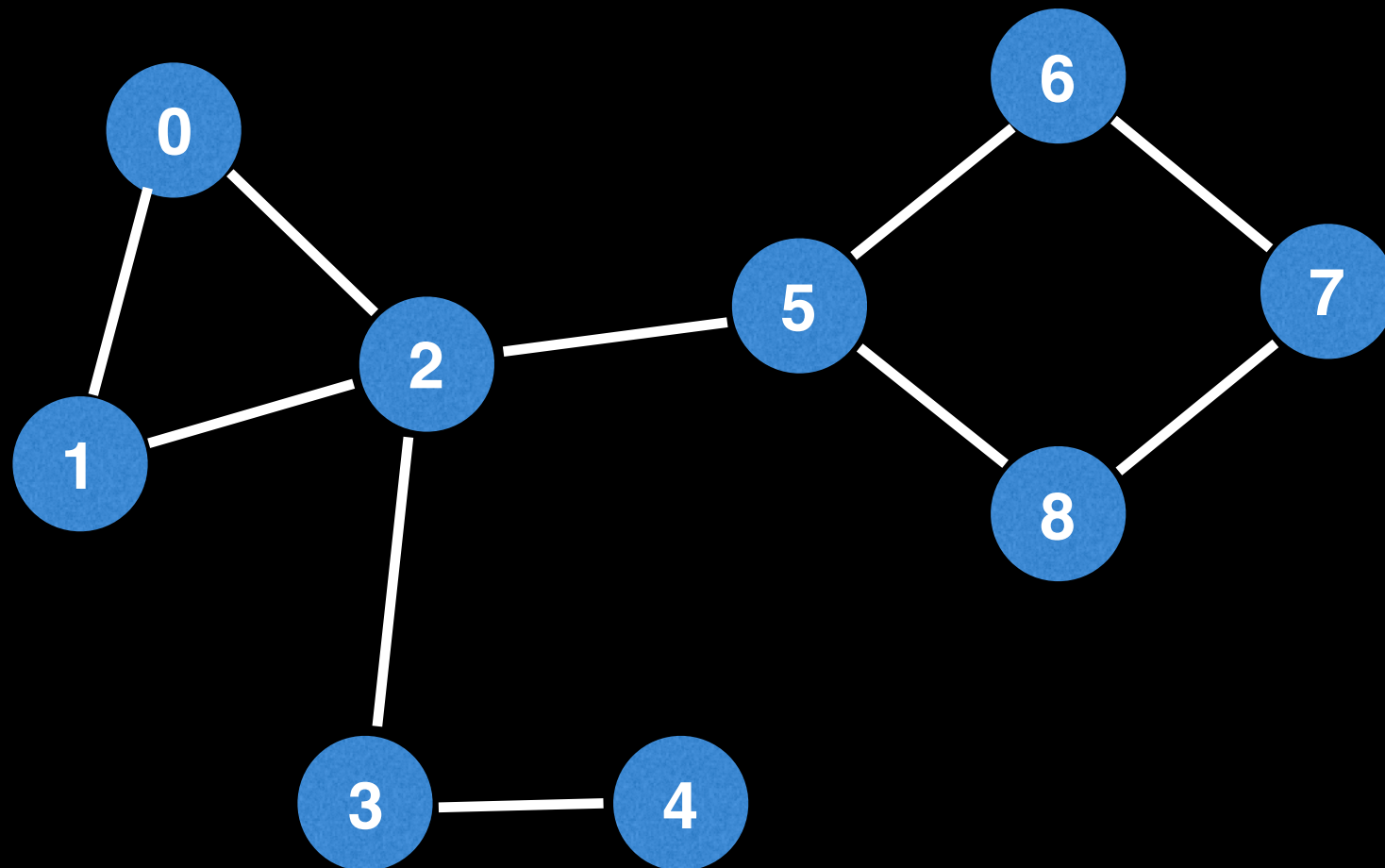
What are bridges & articulation points?

A **bridge / cut edge** is any edge in a graph whose removal increases the number of connected components.



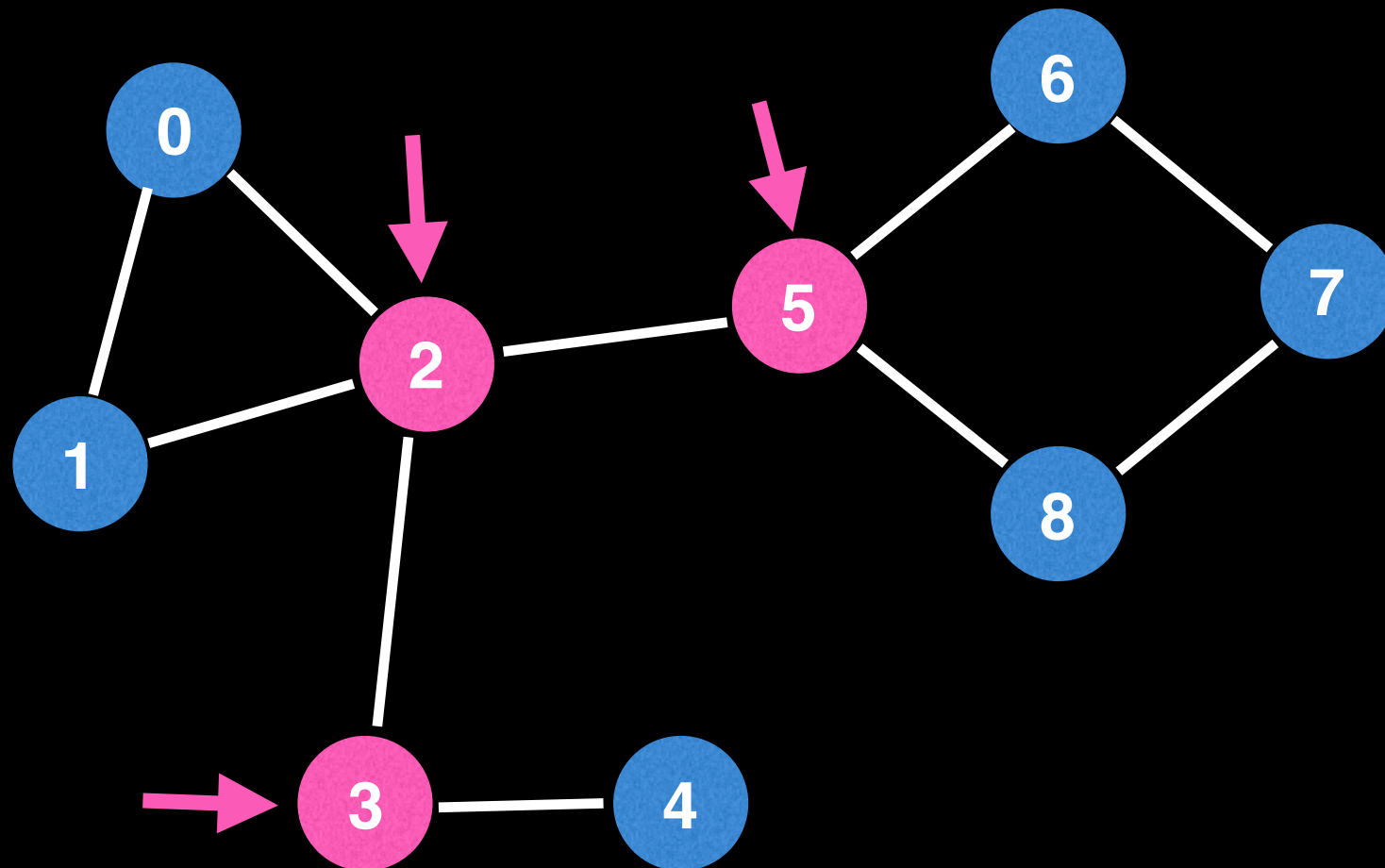
What are bridges & articulation points?

An **articulation point** / **cut vertex** is any node in a graph whose removal increases the number of connected components.



What are bridges & articulation points?

An **articulation point** / **cut vertex** is any node in a graph whose removal increases the number of connected components.



What are bridges & articulation points?

Bridges and articulation points are important in graph theory because they often hint at **weak points, bottlenecks or vulnerabilities in a graph**. Therefore, it's important to be able to quickly find/detect when and where these occur.

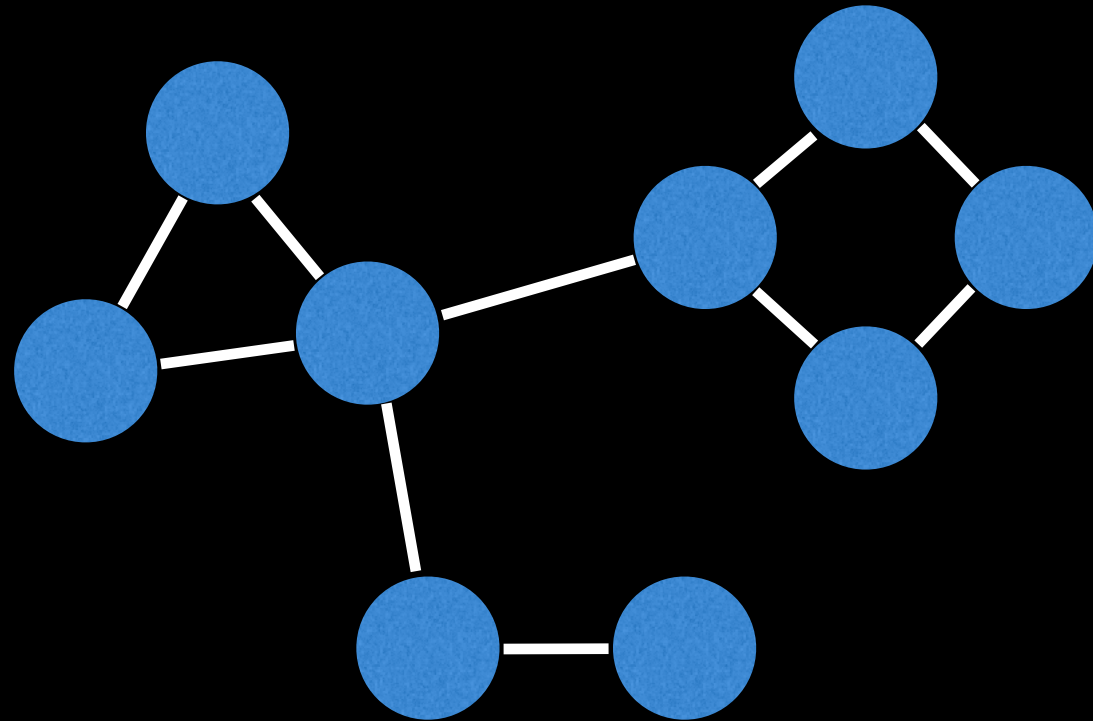
Both problems are related so we will develop an algorithm to find bridges and then modify it slightly to find articulation points.

Bridges algorithm

Start at any node and do a Depth First Search (DFS) traversal labeling nodes with an increasing id value as you go. Keep **track the id of each node** and the **smallest low-link** value. During the DFS, bridges will be found where the id of the node your edge is coming from is less than the low link value of the node your edge is going to.

NOTE: The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node when doing a DFS (including itself).

DFS traversal

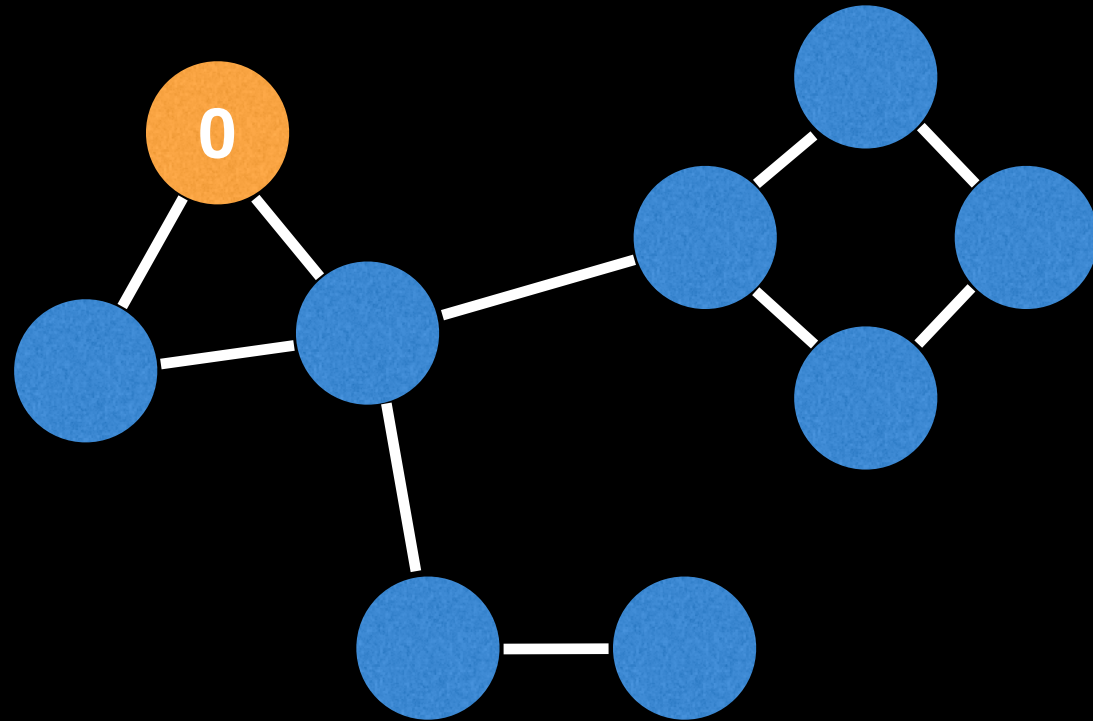


Undirected edge



Directed edge

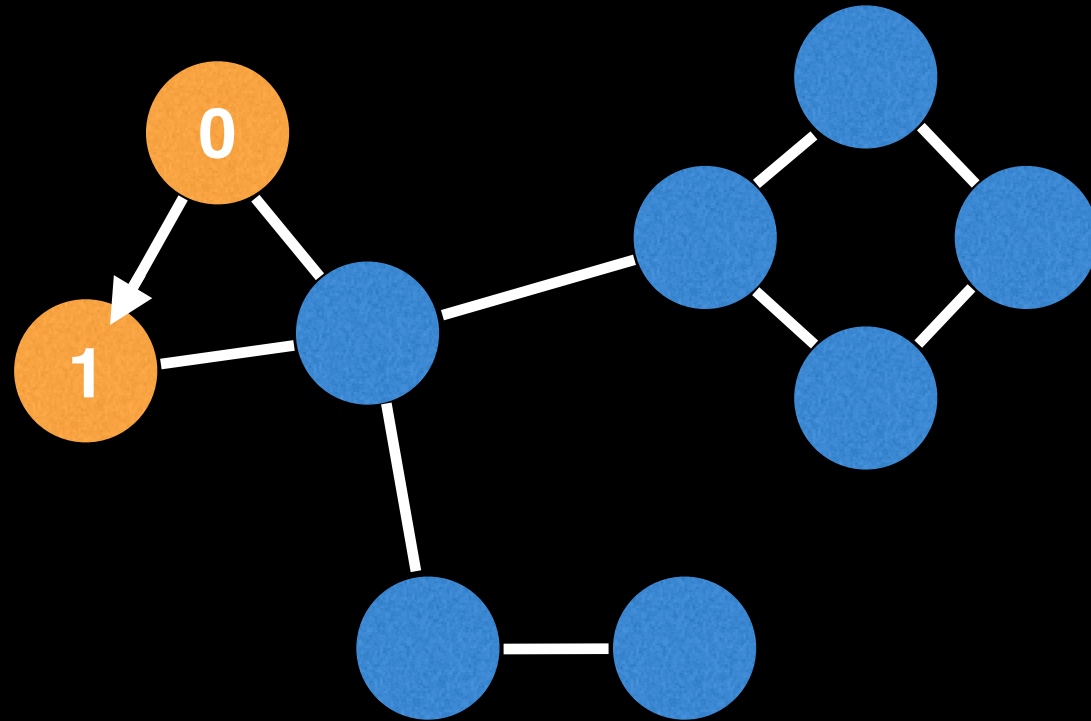
DFS traversal



Undirected edge

Directed edge

DFS traversal

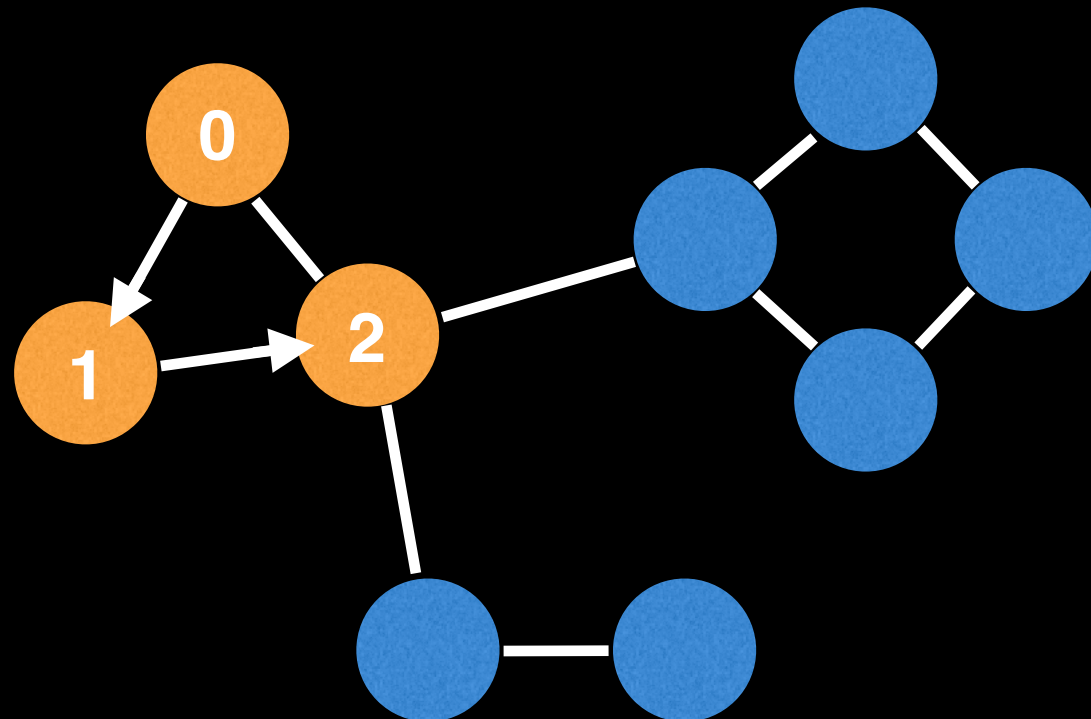


Undirected edge



Directed edge

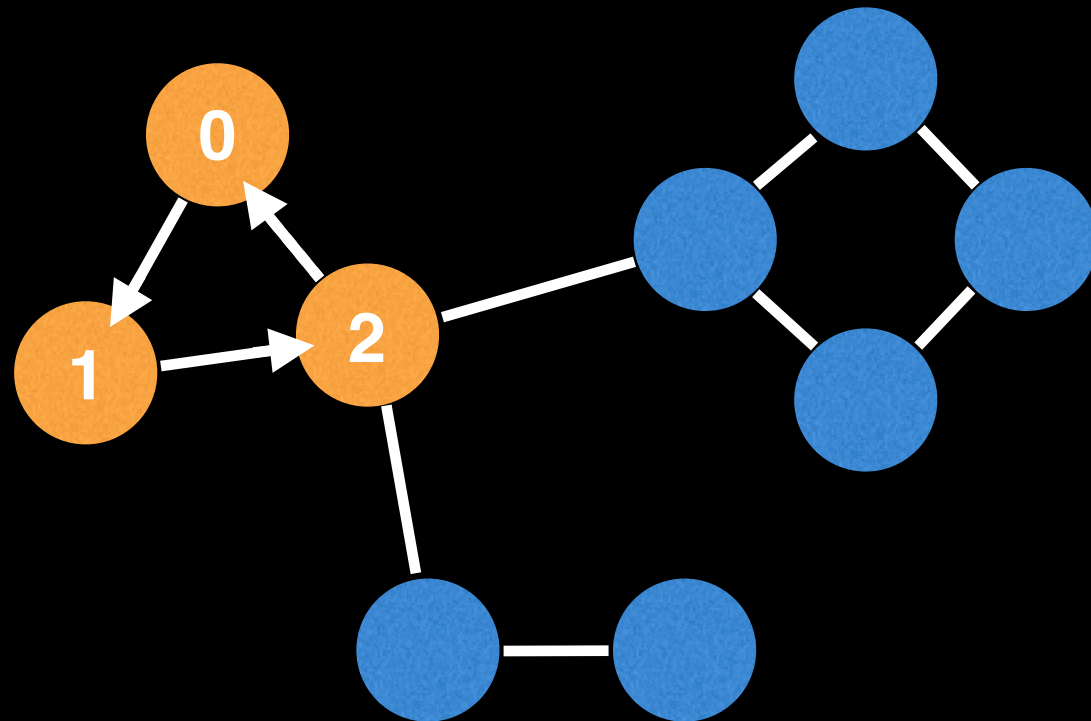
DFS traversal



Undirected edge

Directed edge

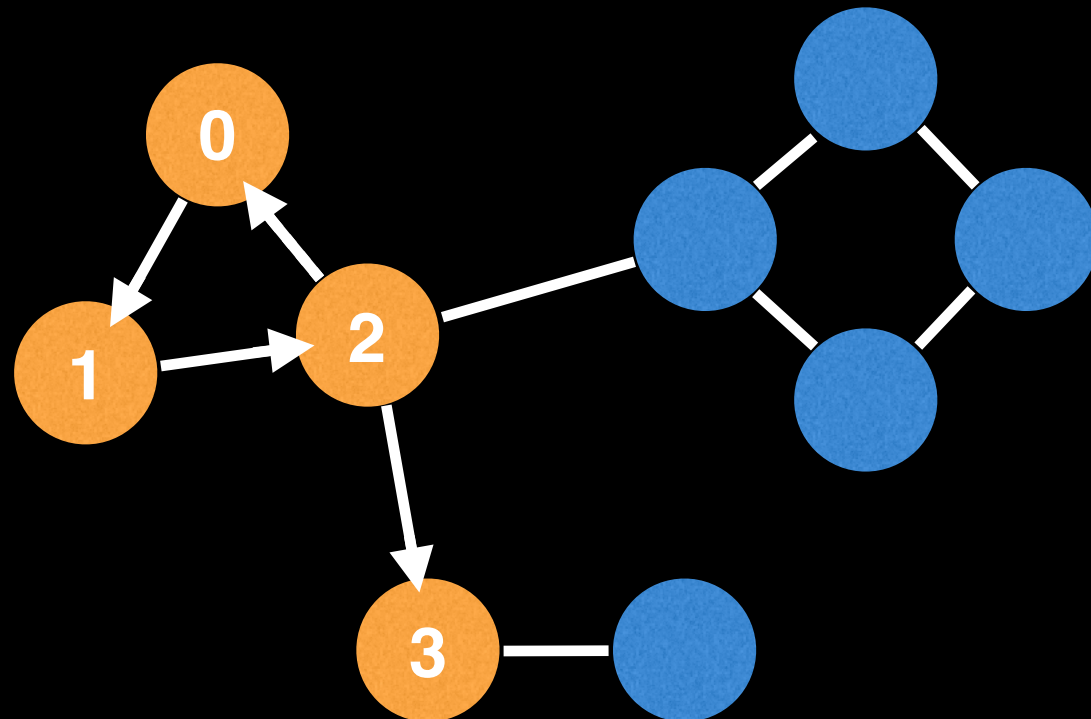
DFS traversal



Undirected edge

Directed edge

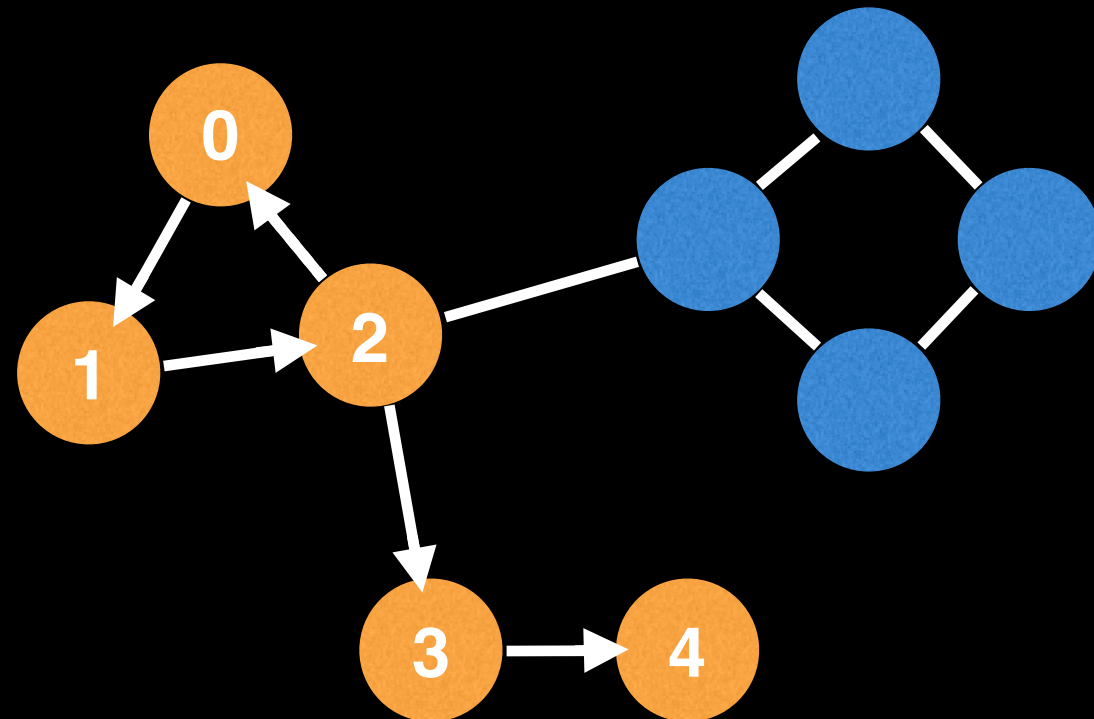
DFS traversal



Undirected edge

Directed edge

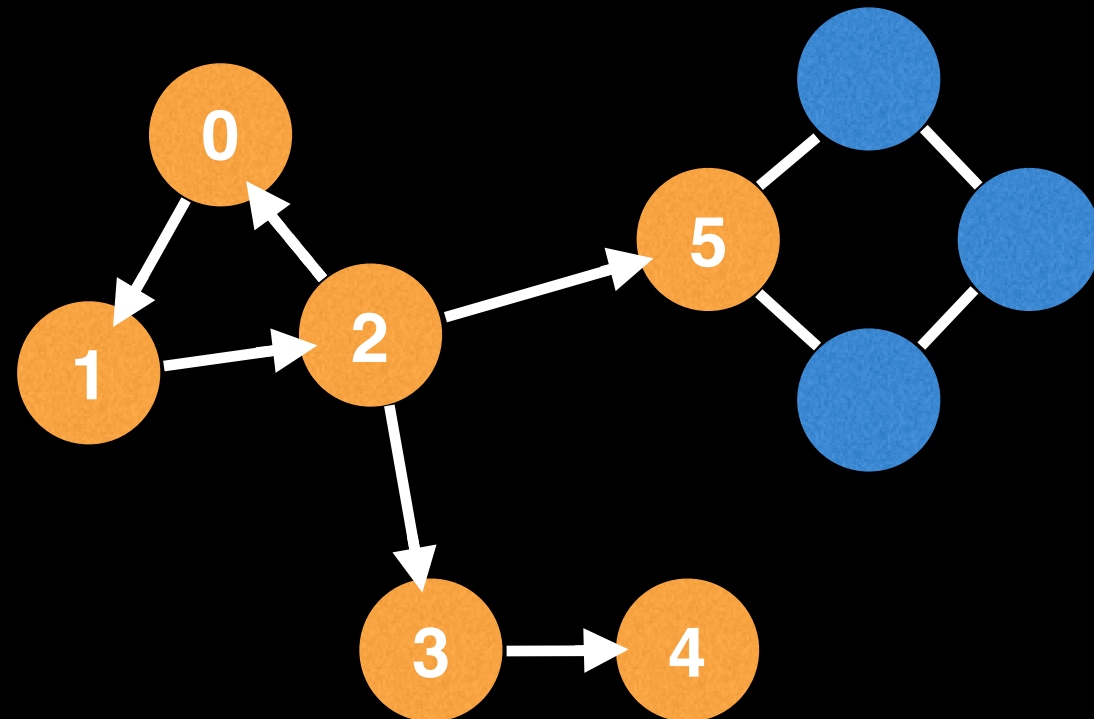
DFS traversal



Undirected edge

Directed edge

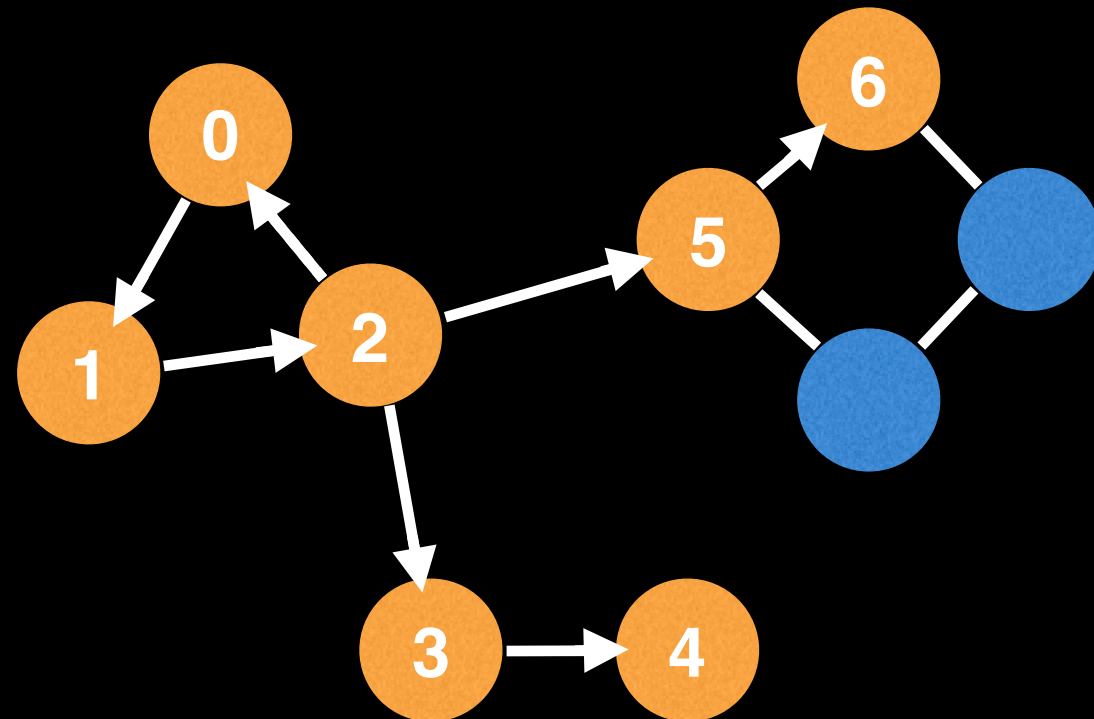
DFS traversal



Undirected edge

Directed edge

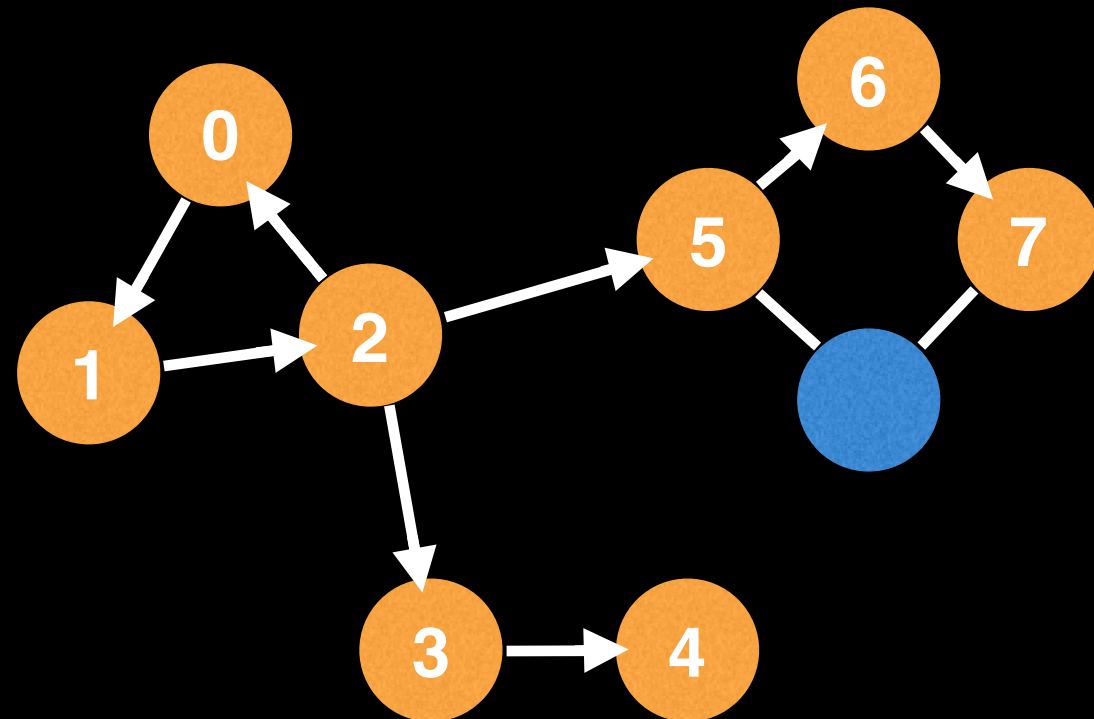
DFS traversal



Undirected edge

Directed edge

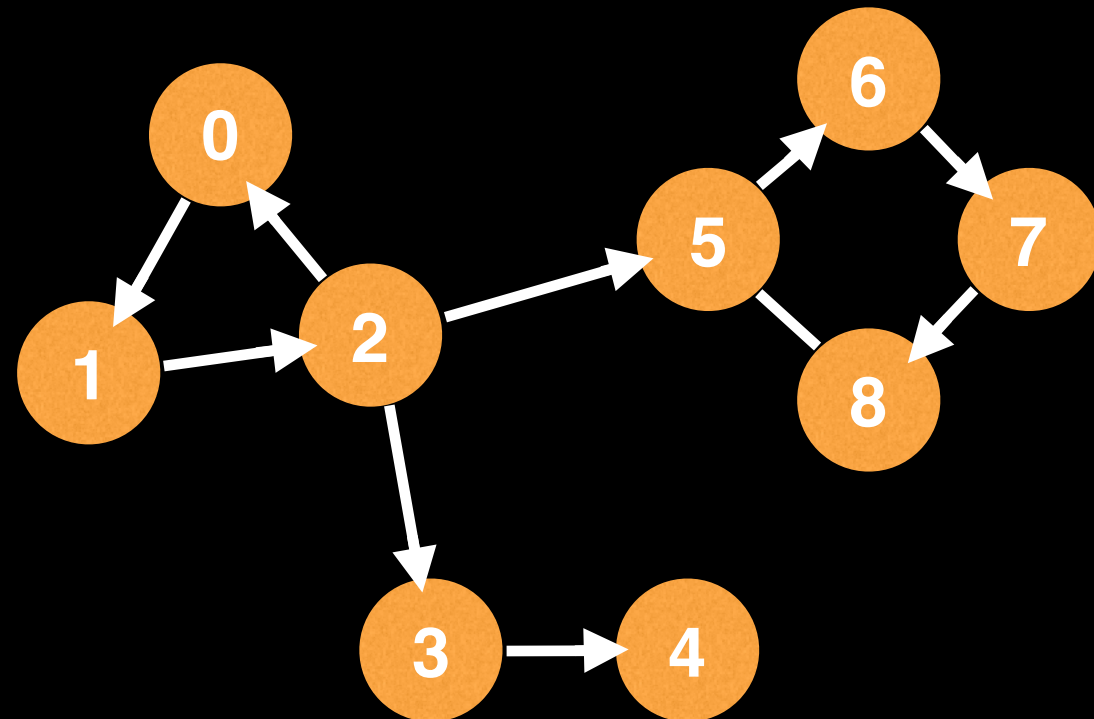
DFS traversal



Undirected edge

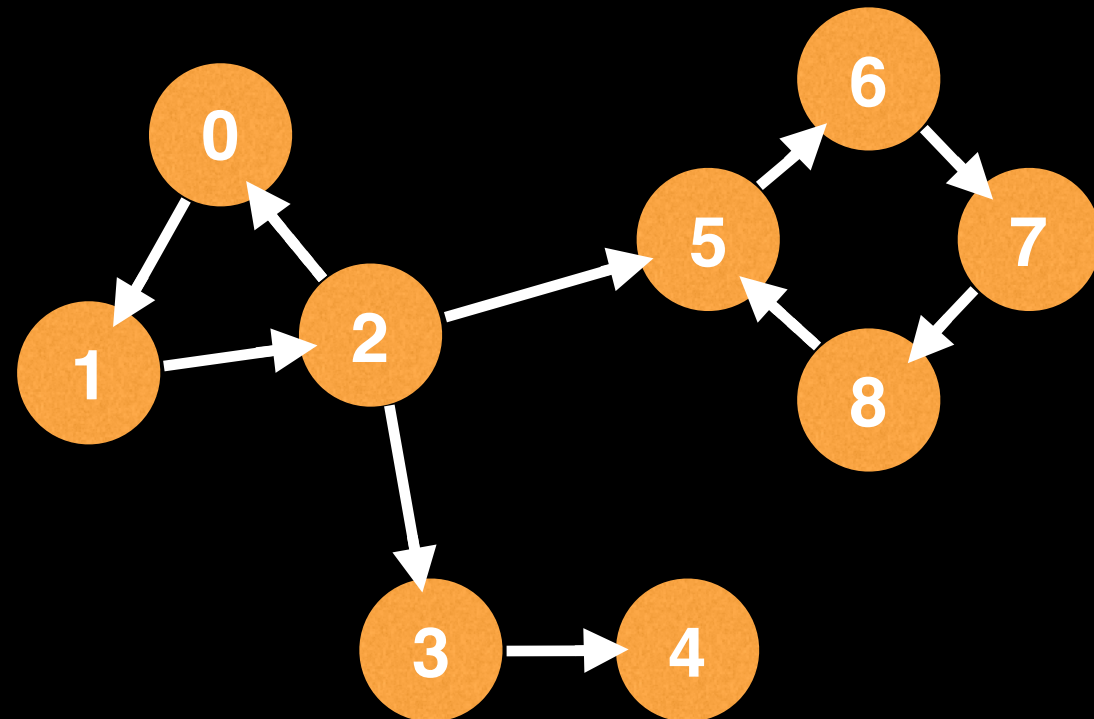
Directed edge

DFS traversal



Undirected edge

Directed edge

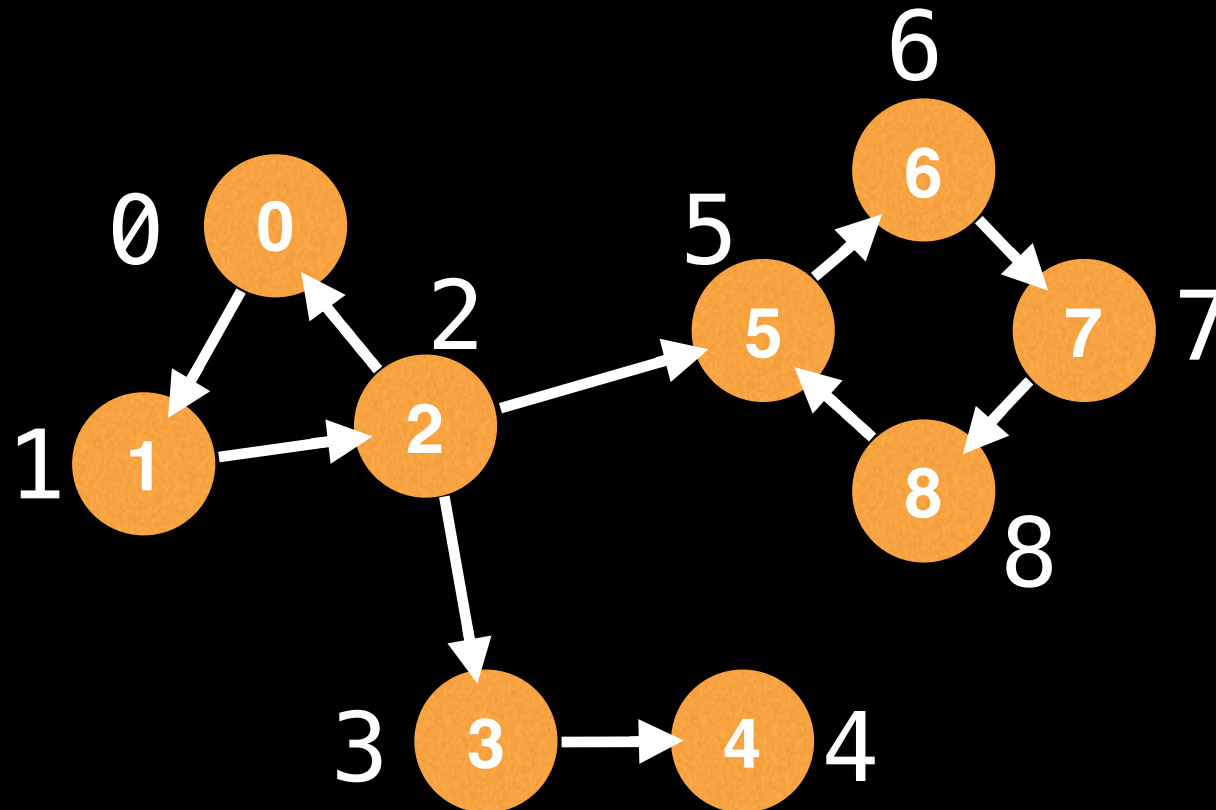


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Initially all low-link values can be initialized to the node ids.

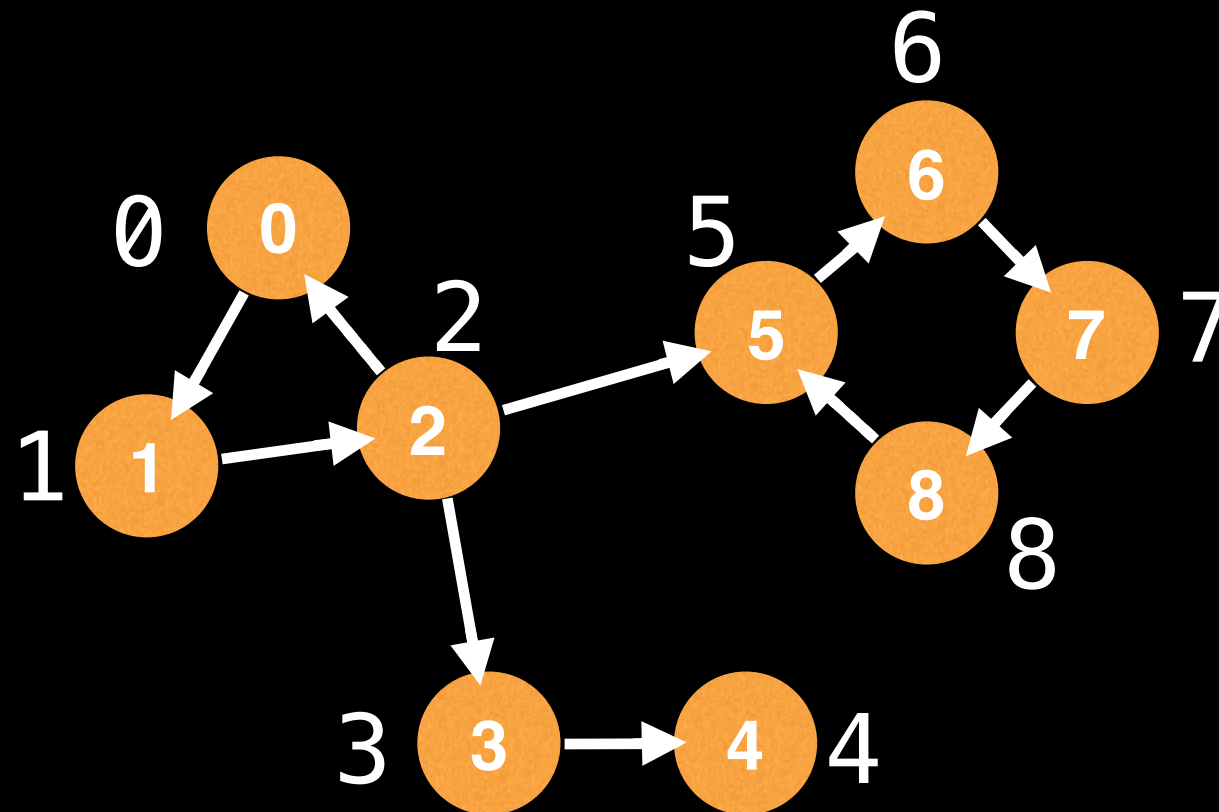


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

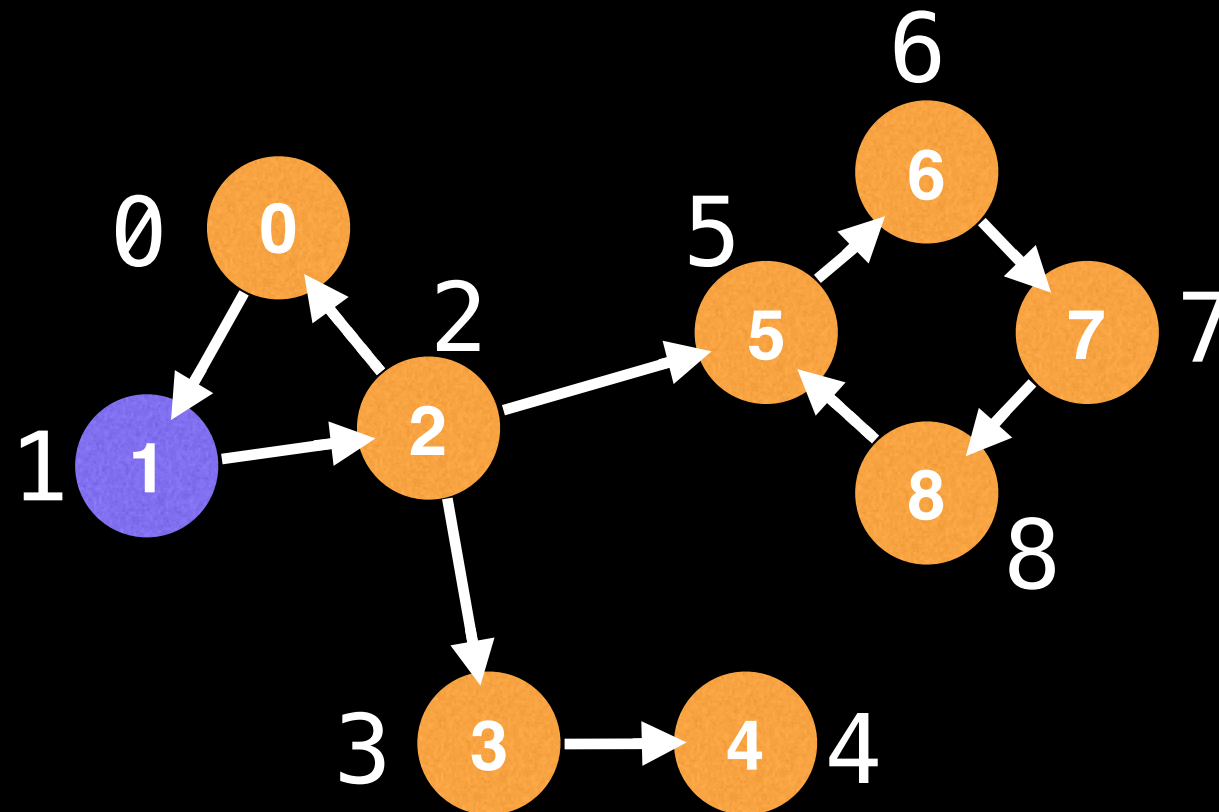


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

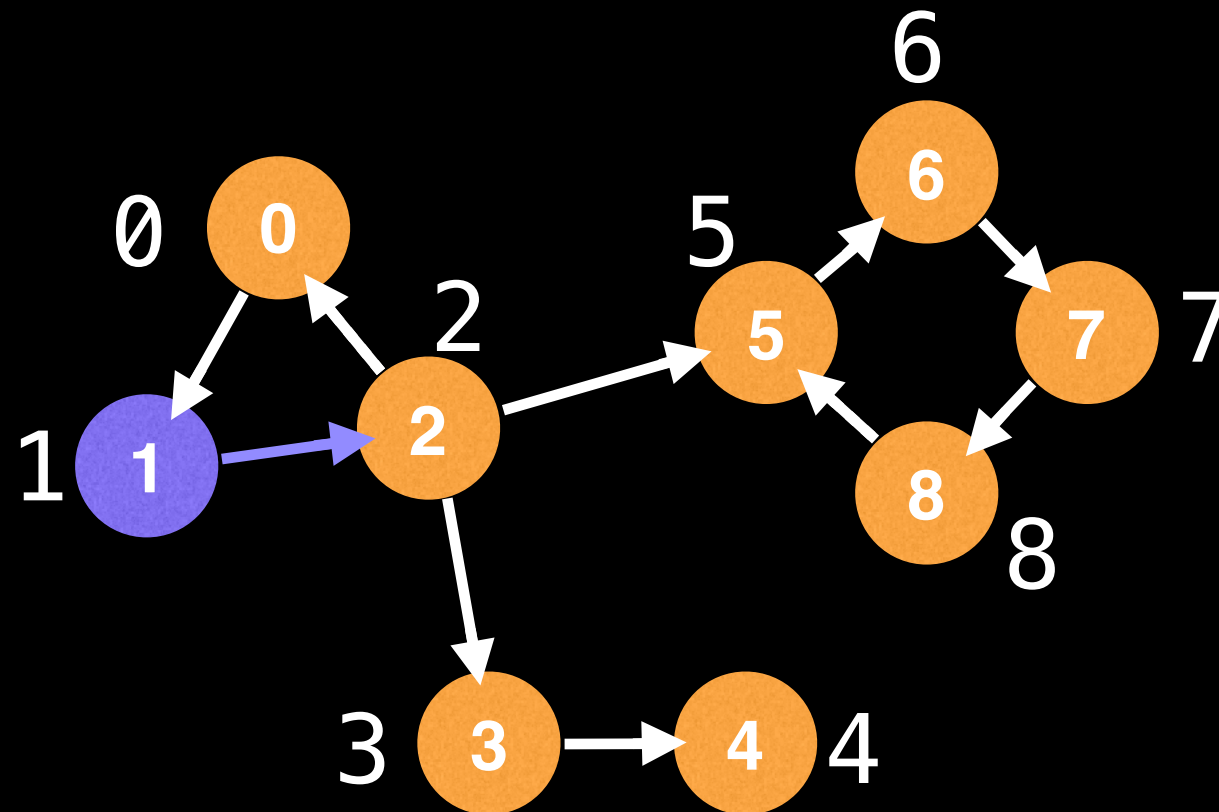


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

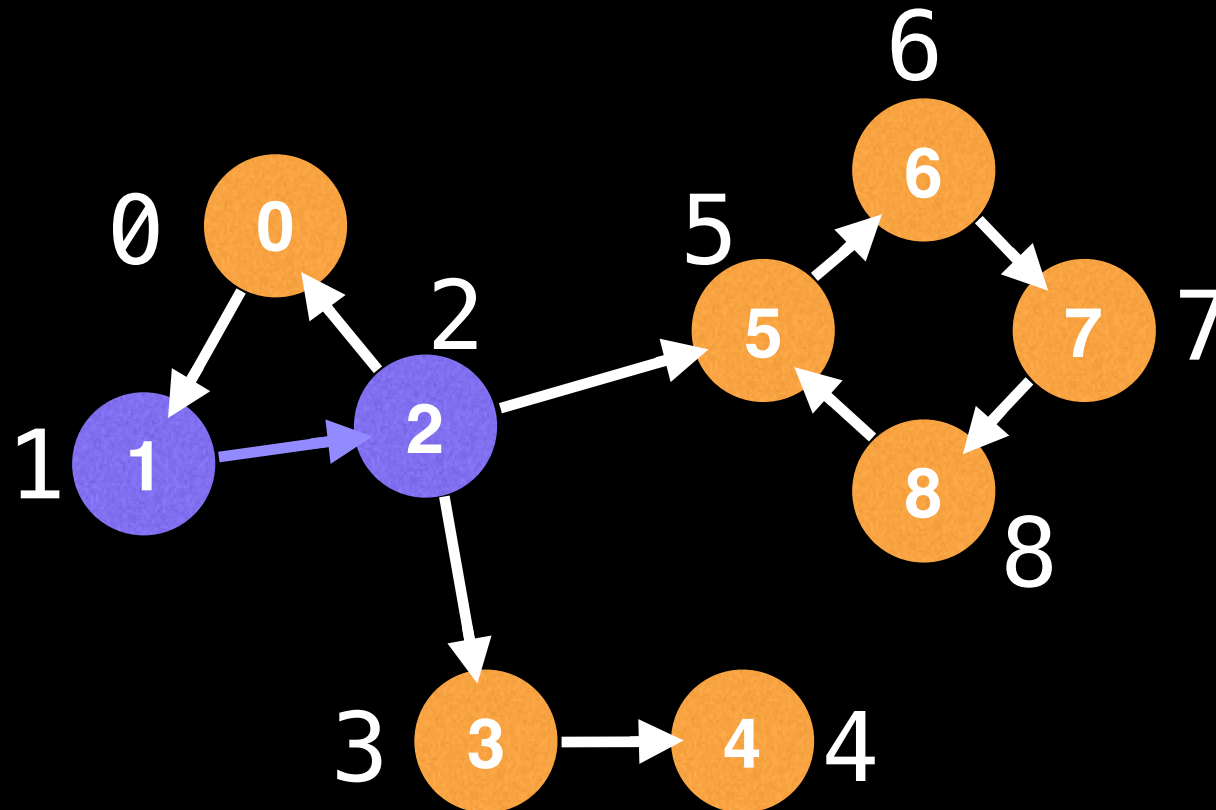


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

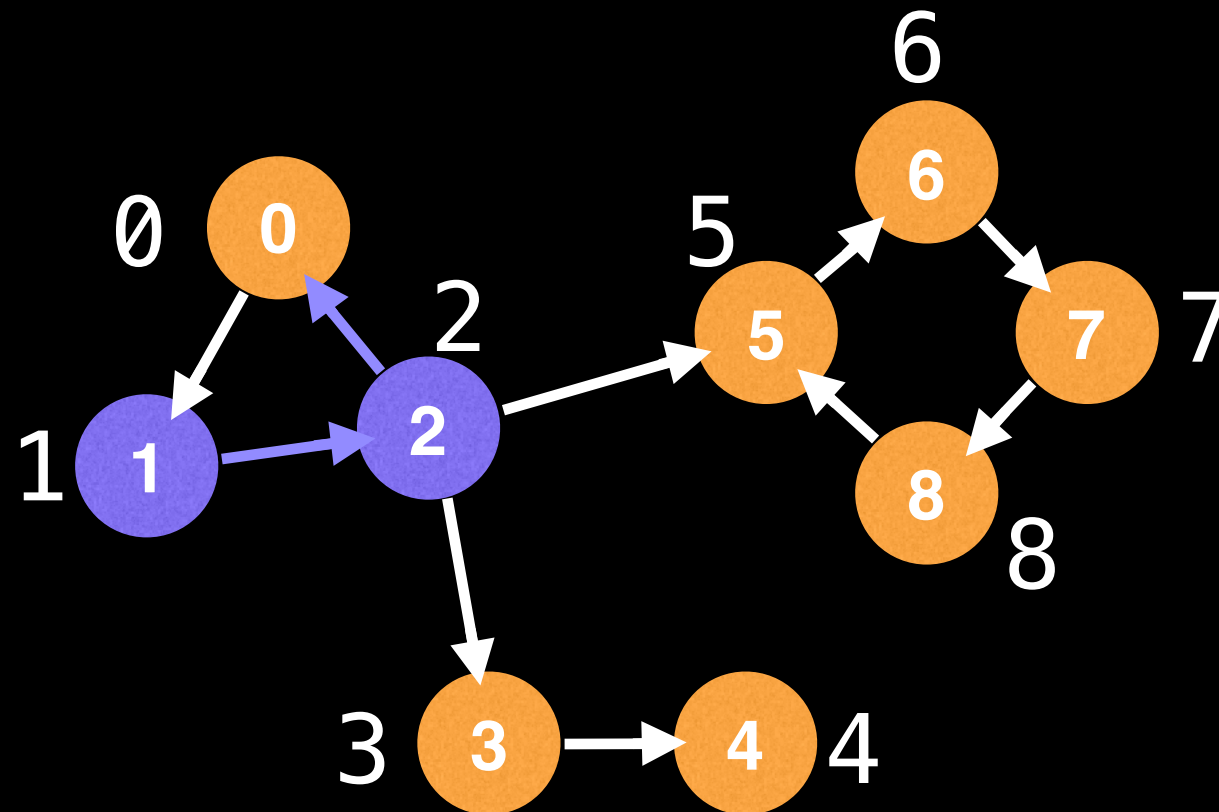


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

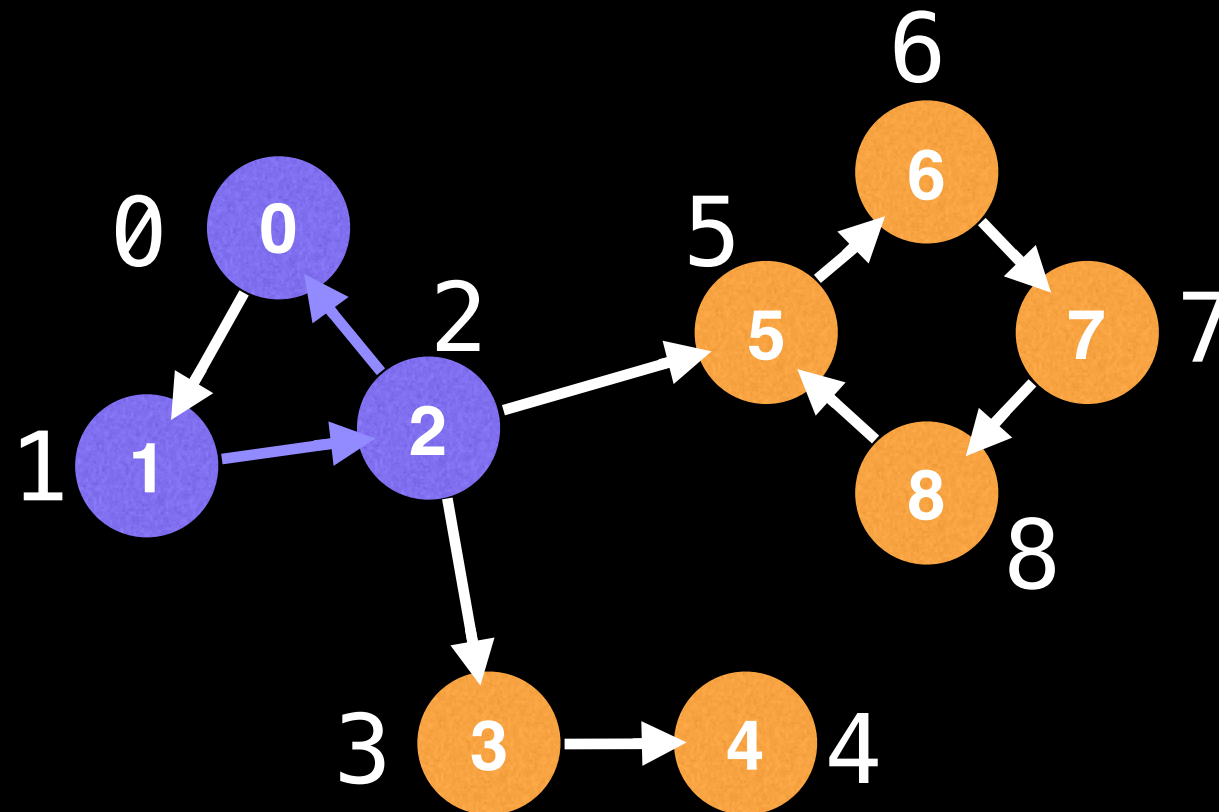


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

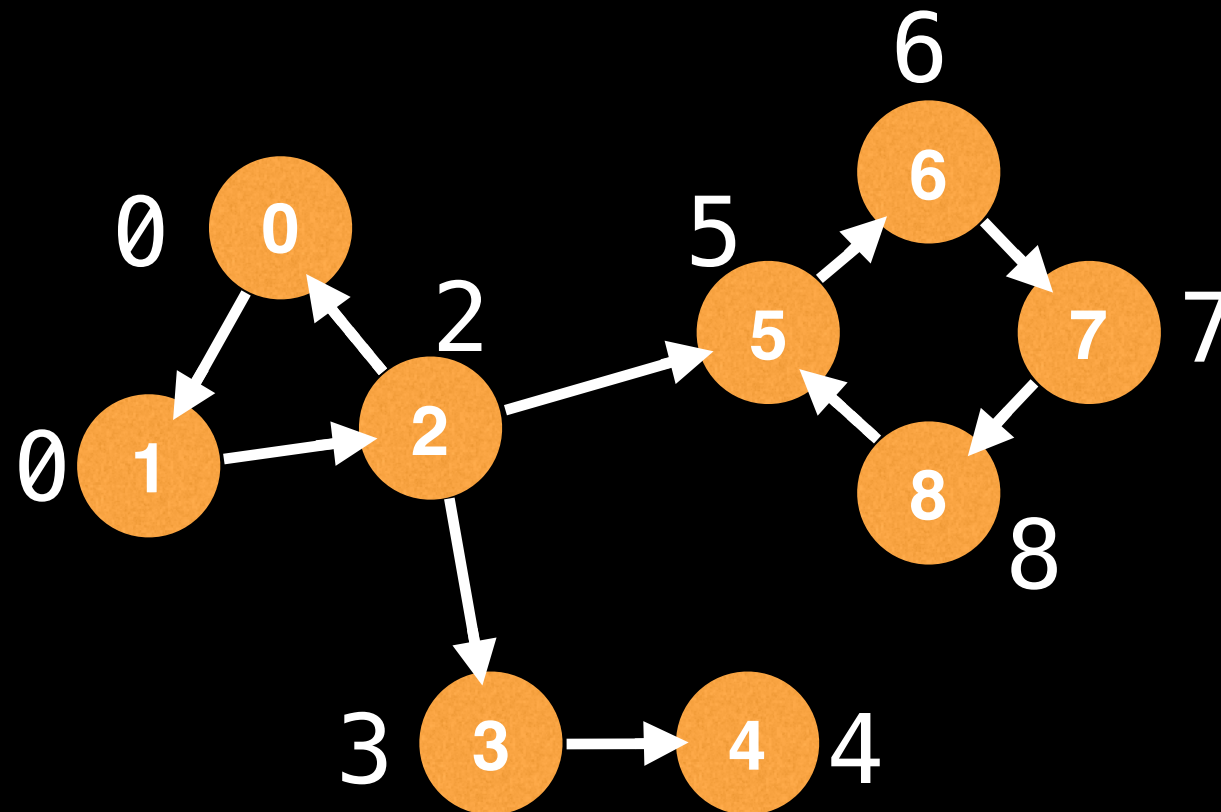


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.


Undirected edge


Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

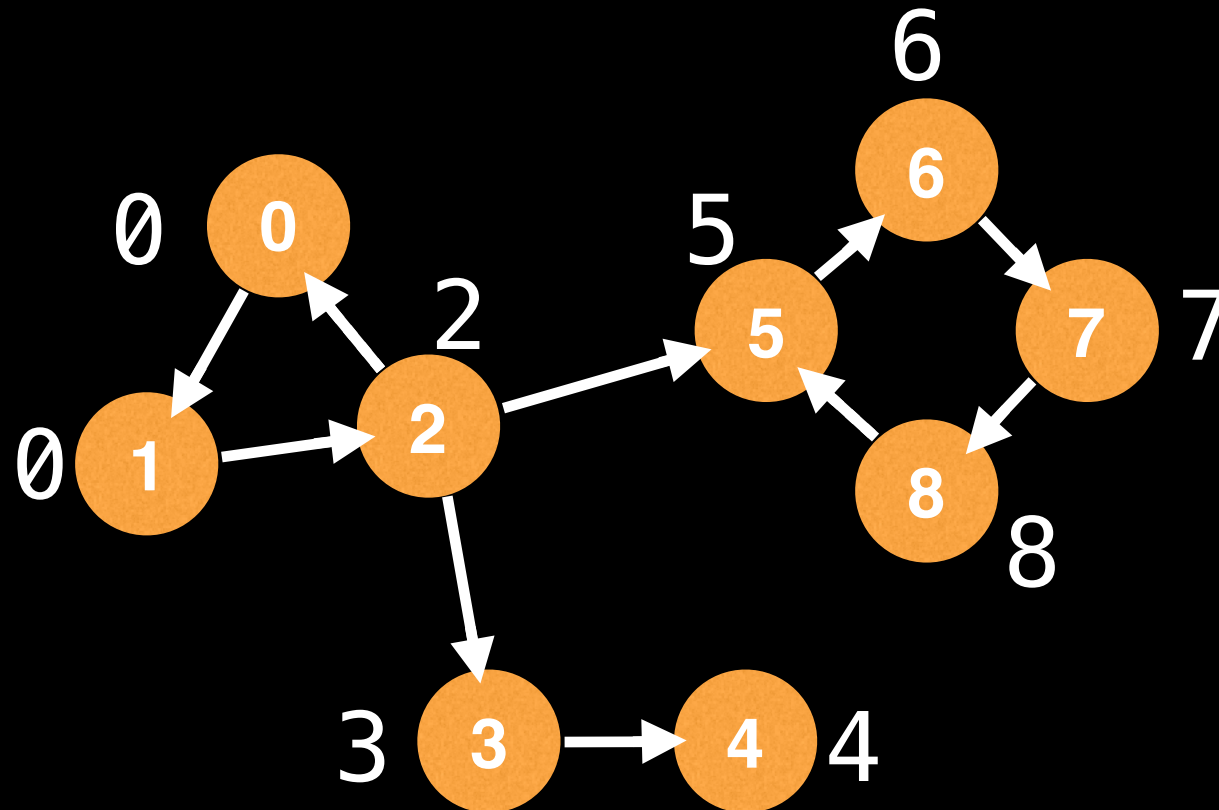


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

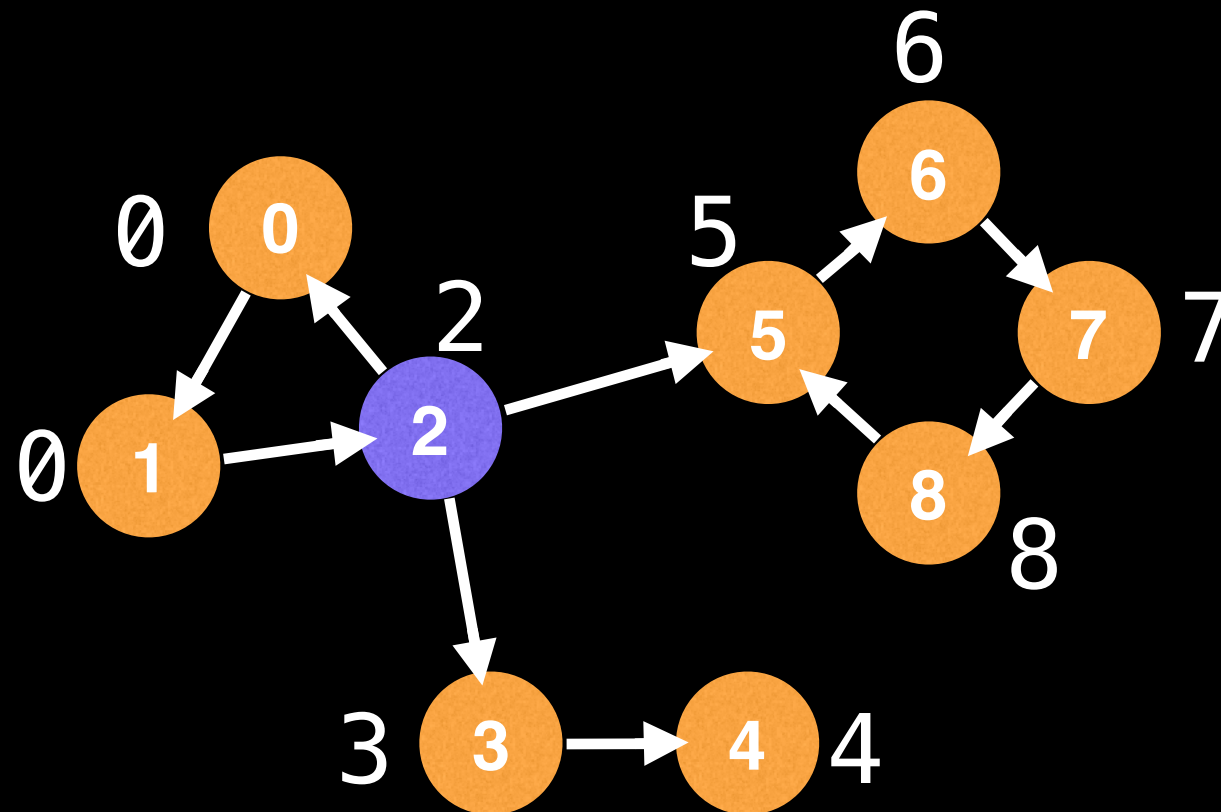


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

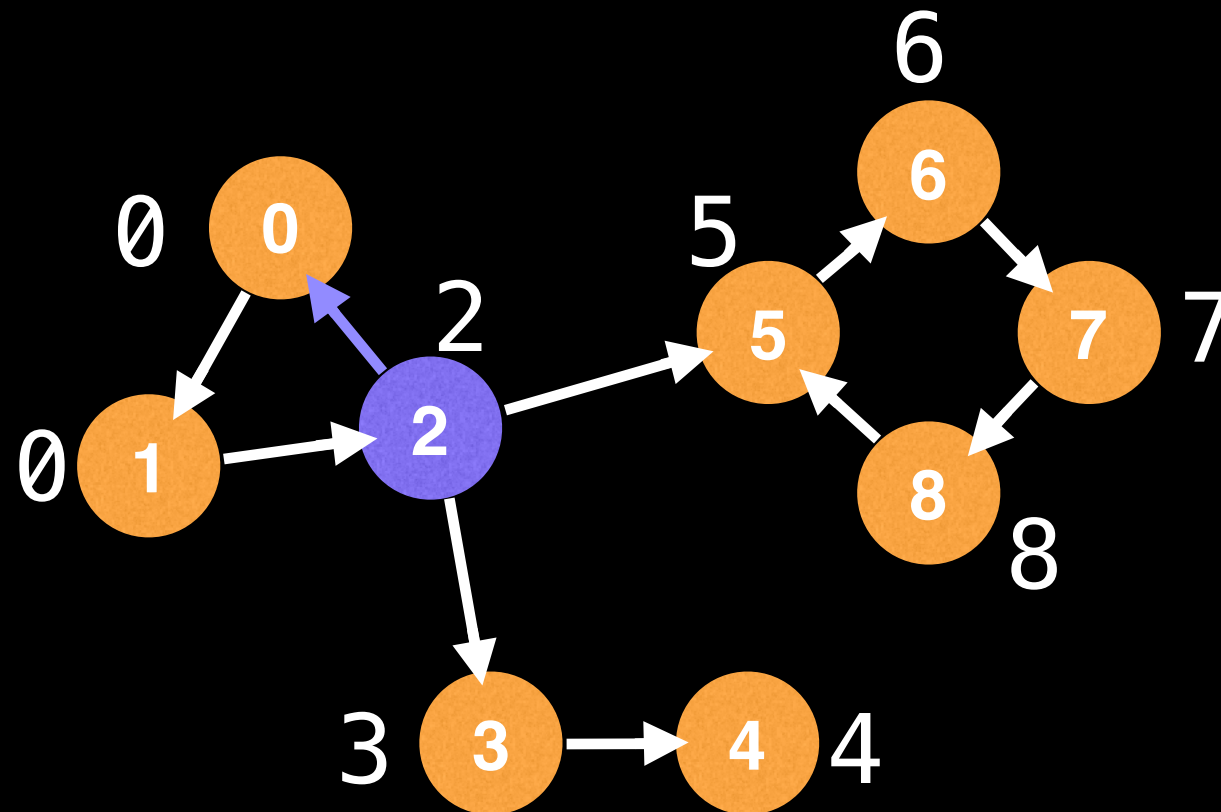


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

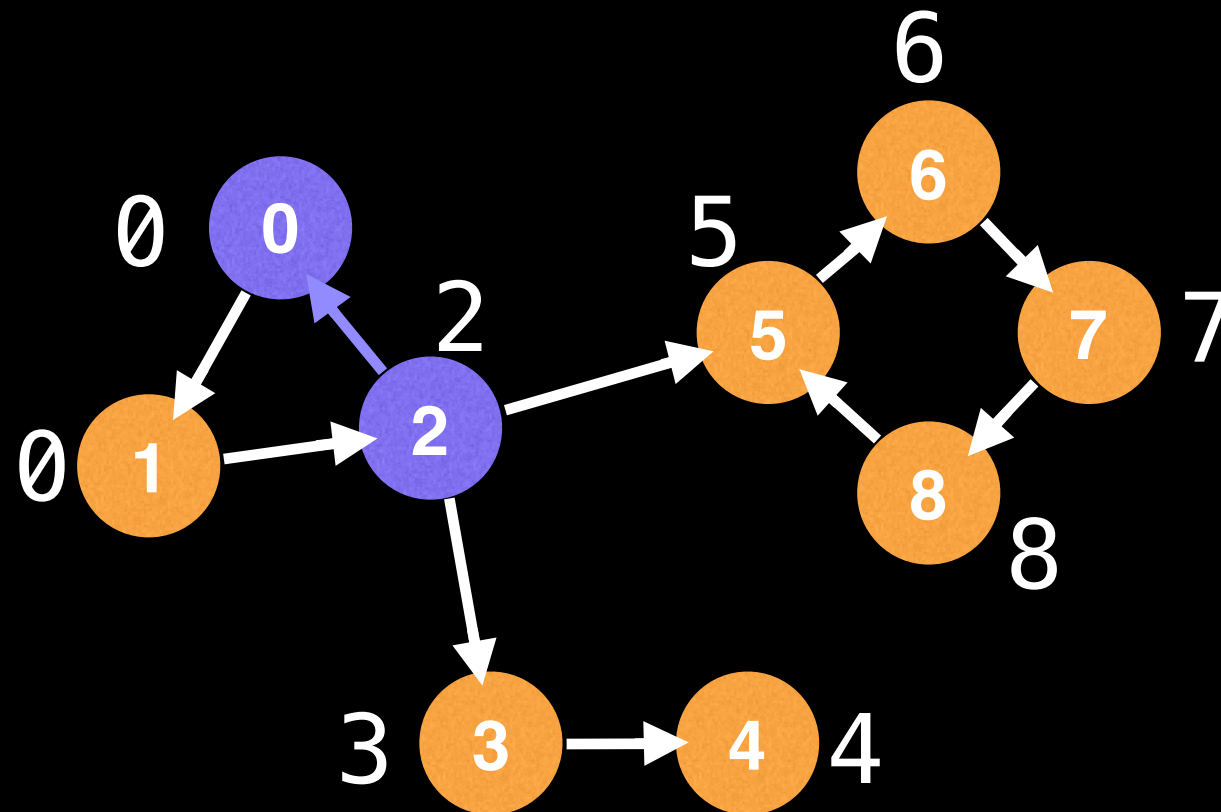


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

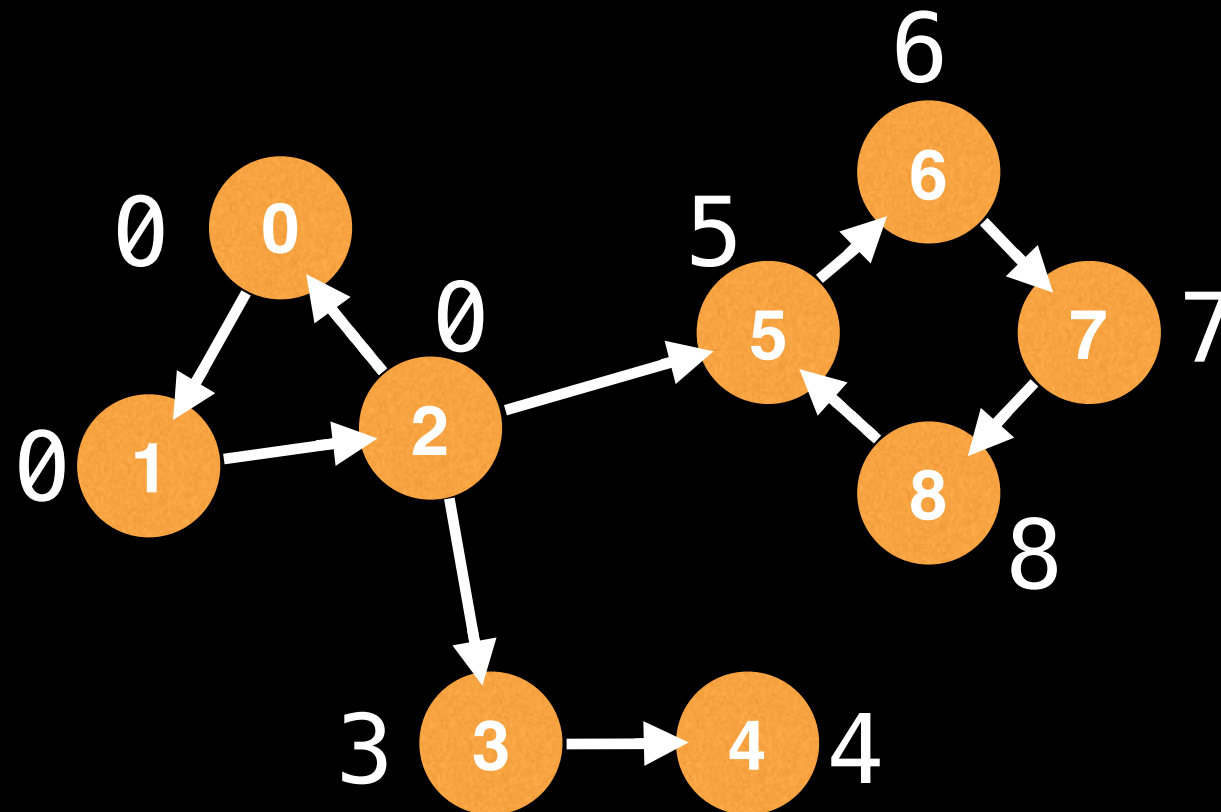


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

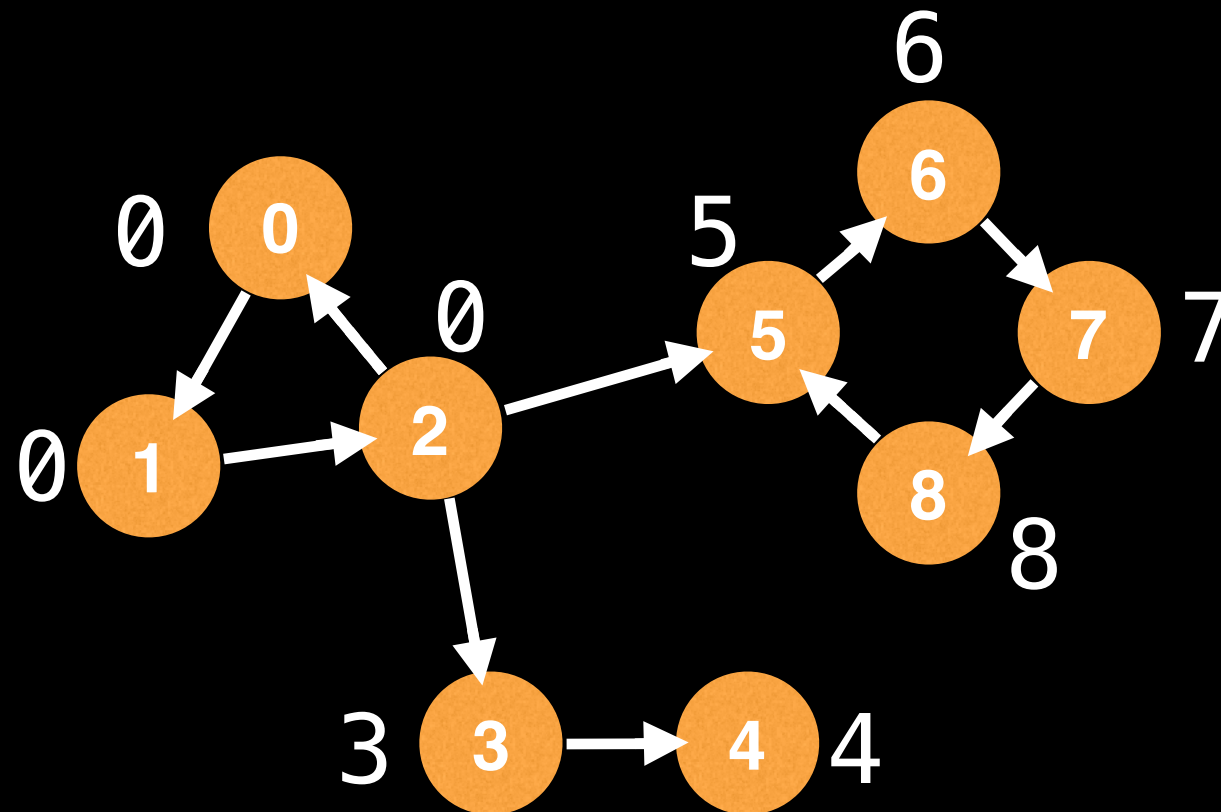


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Cannot update low-link values for nodes
3, 4 and 5.

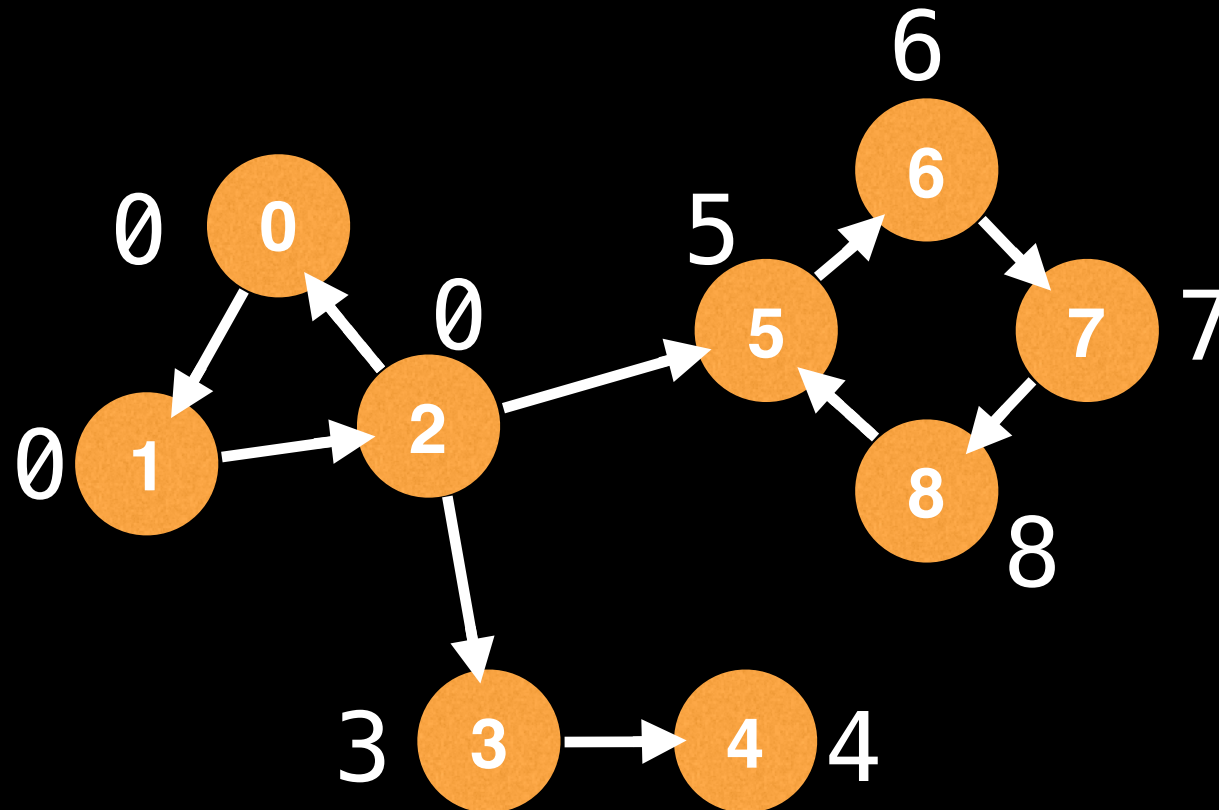


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

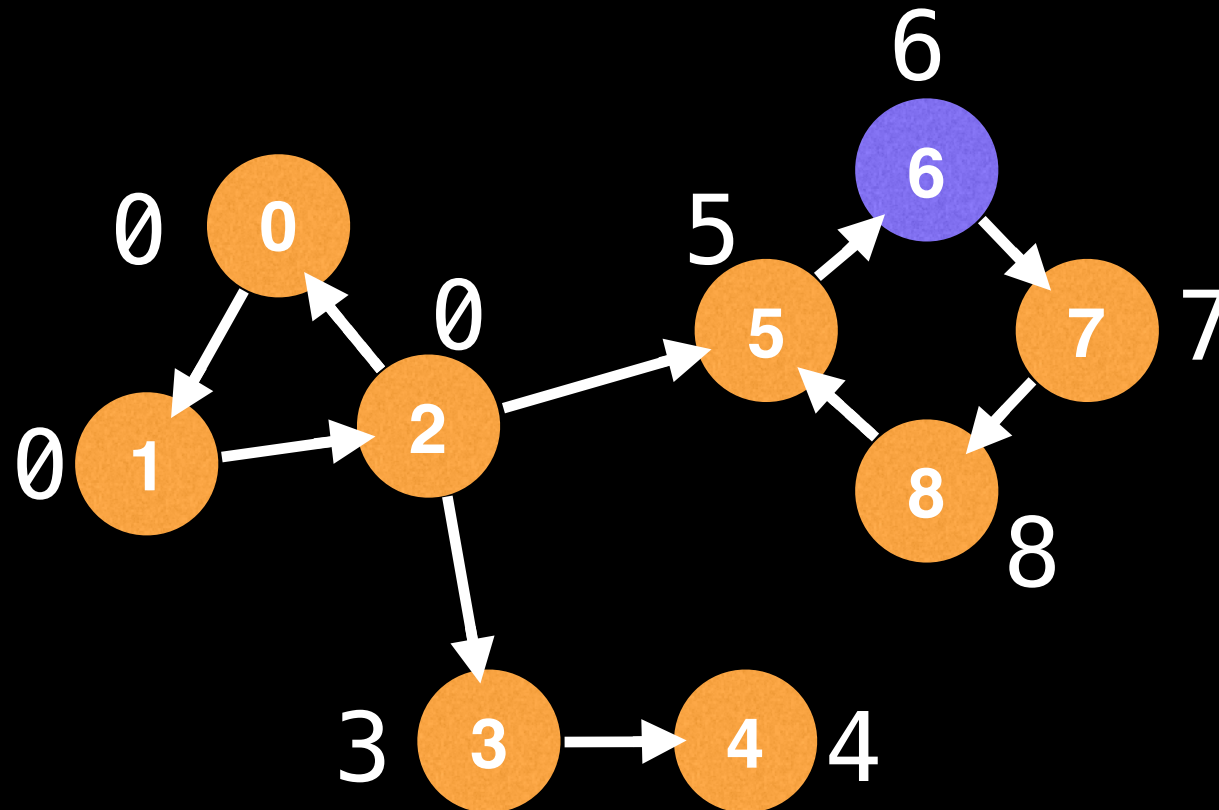


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

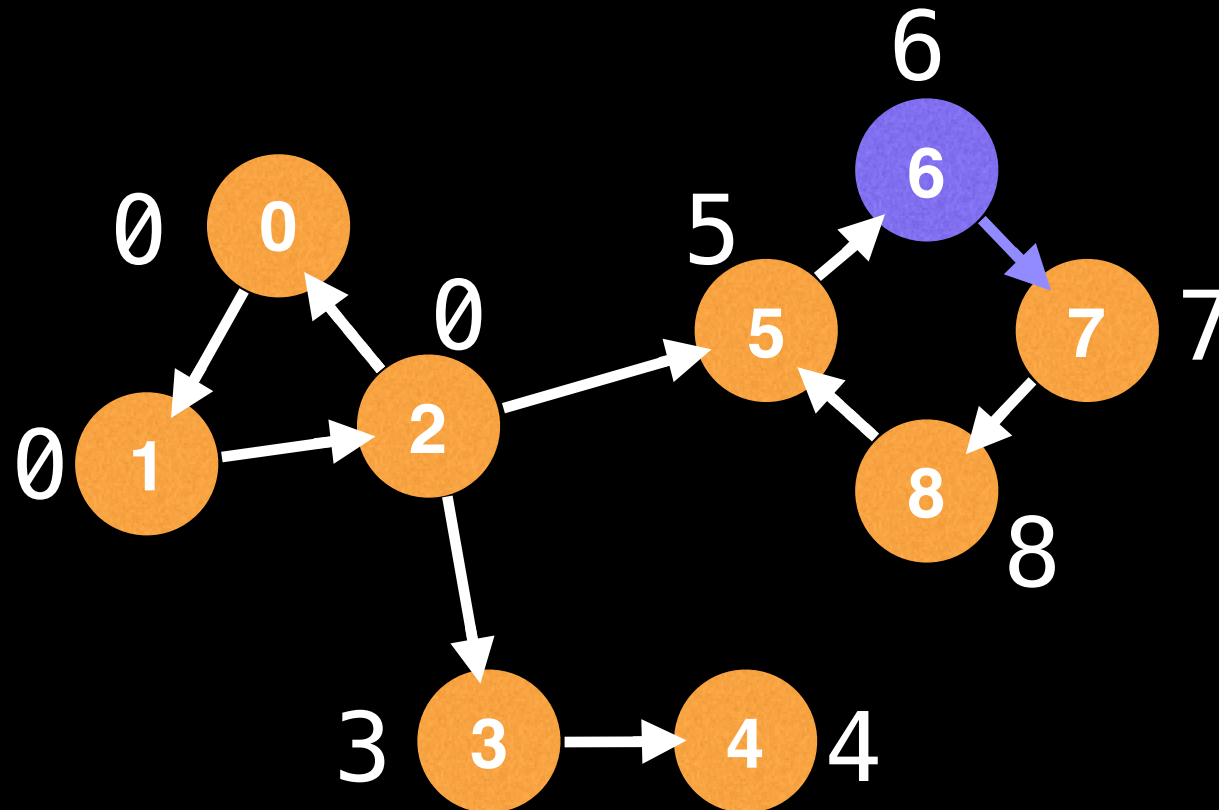


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

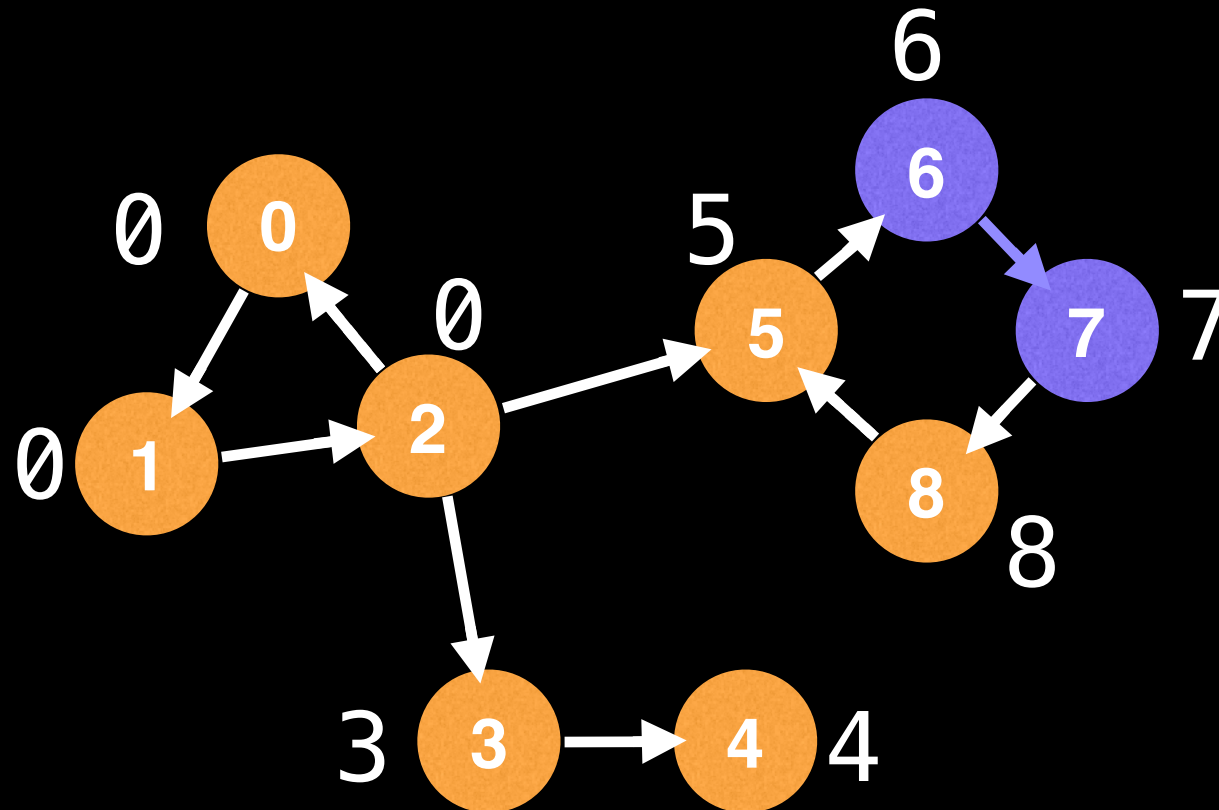


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

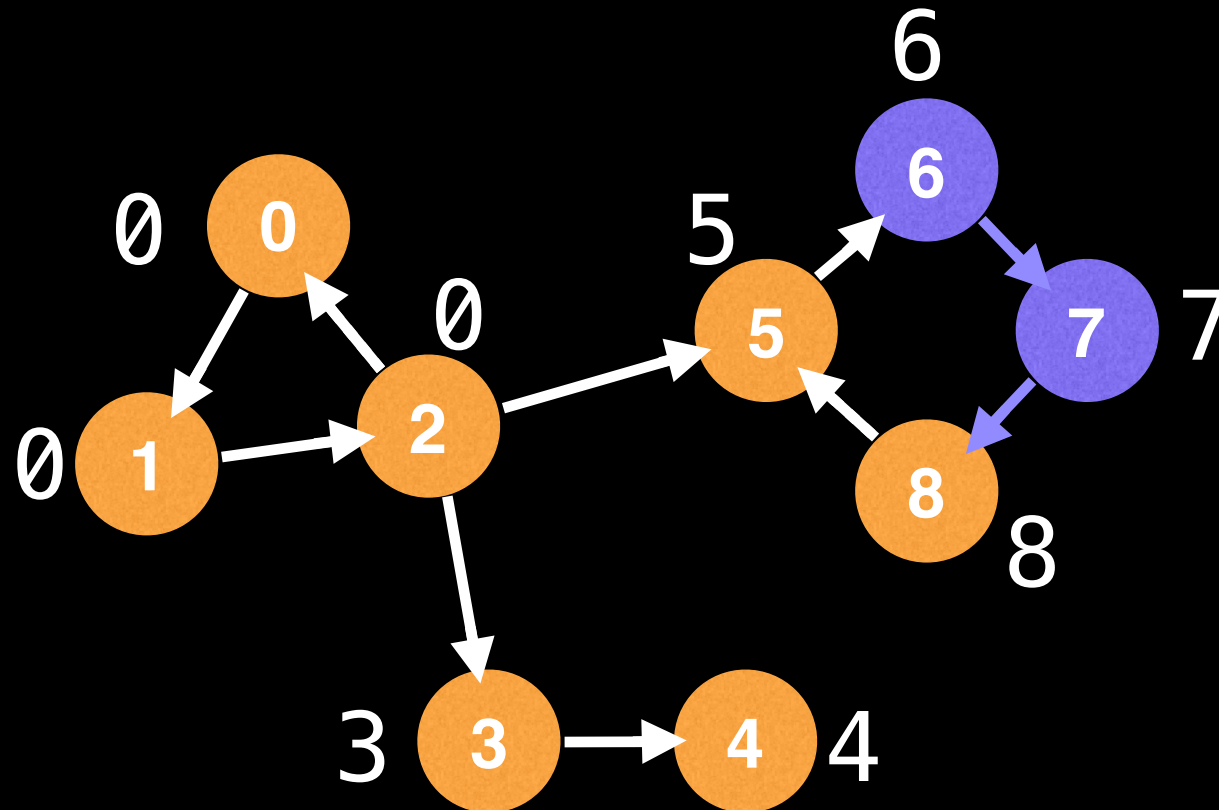


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

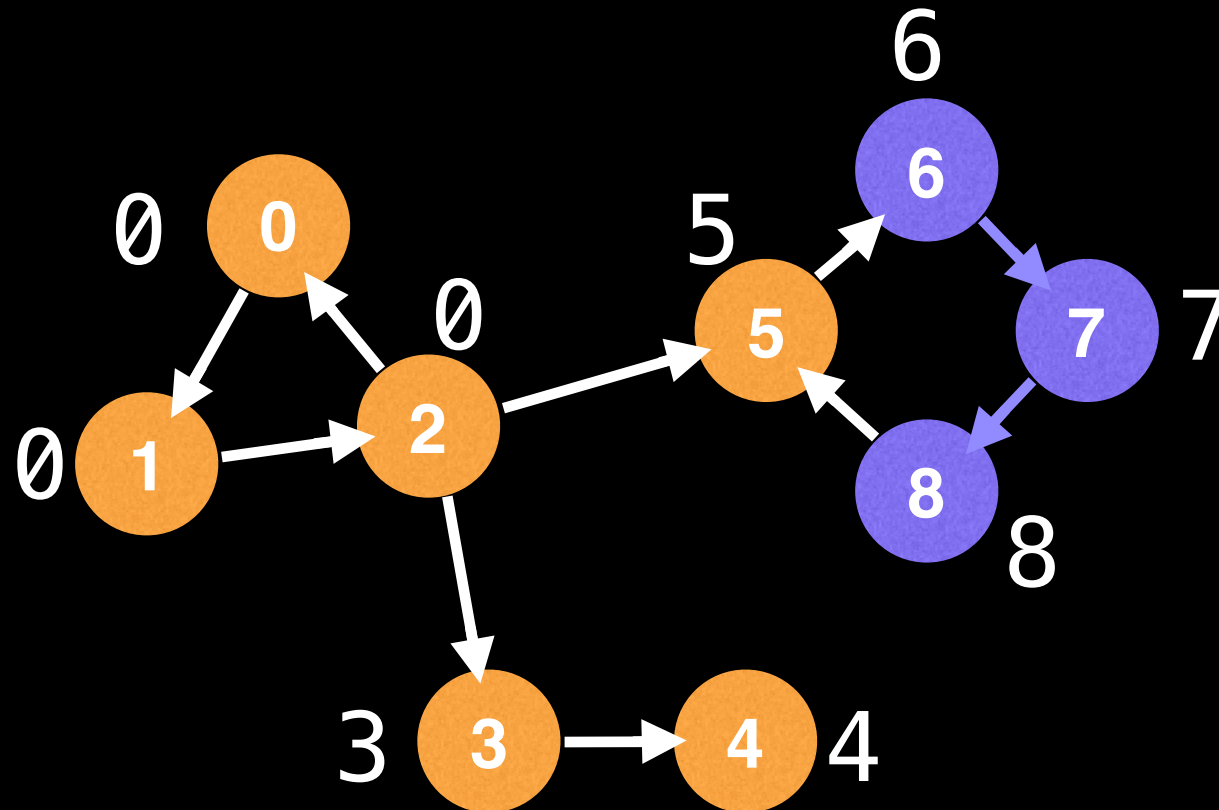


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

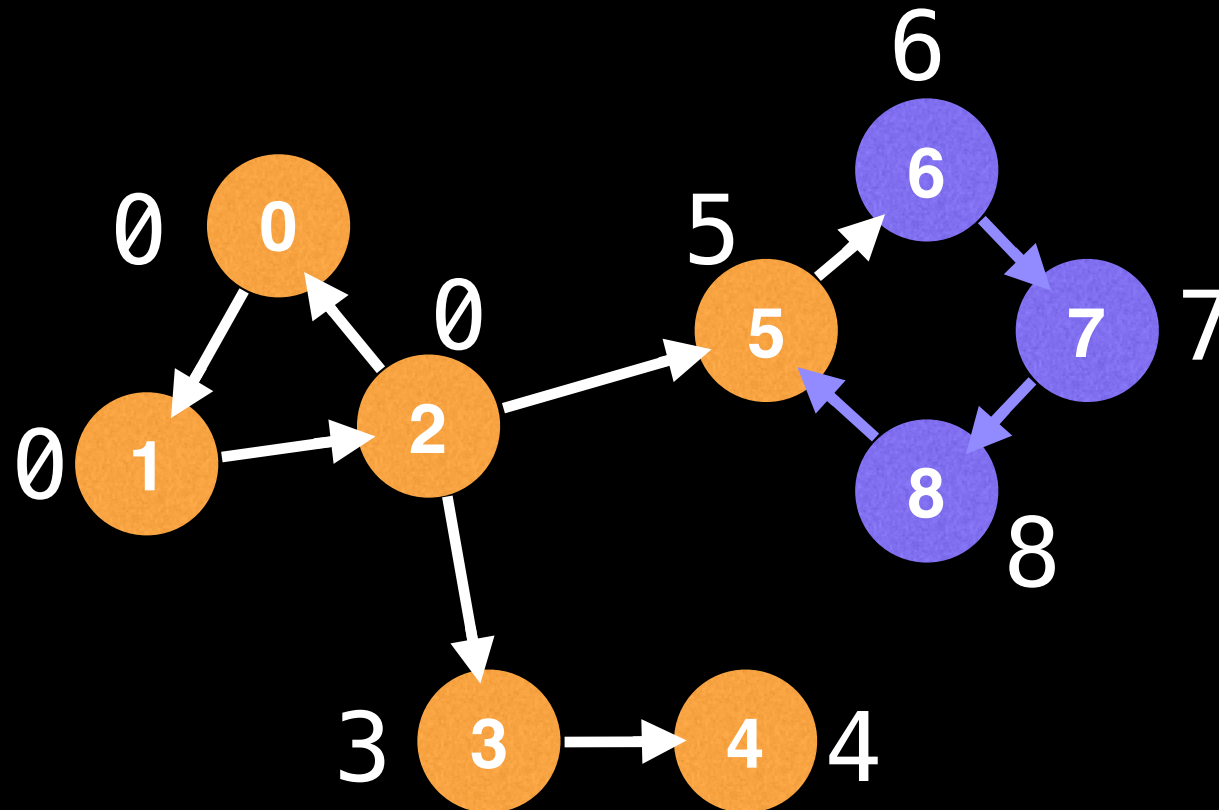


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

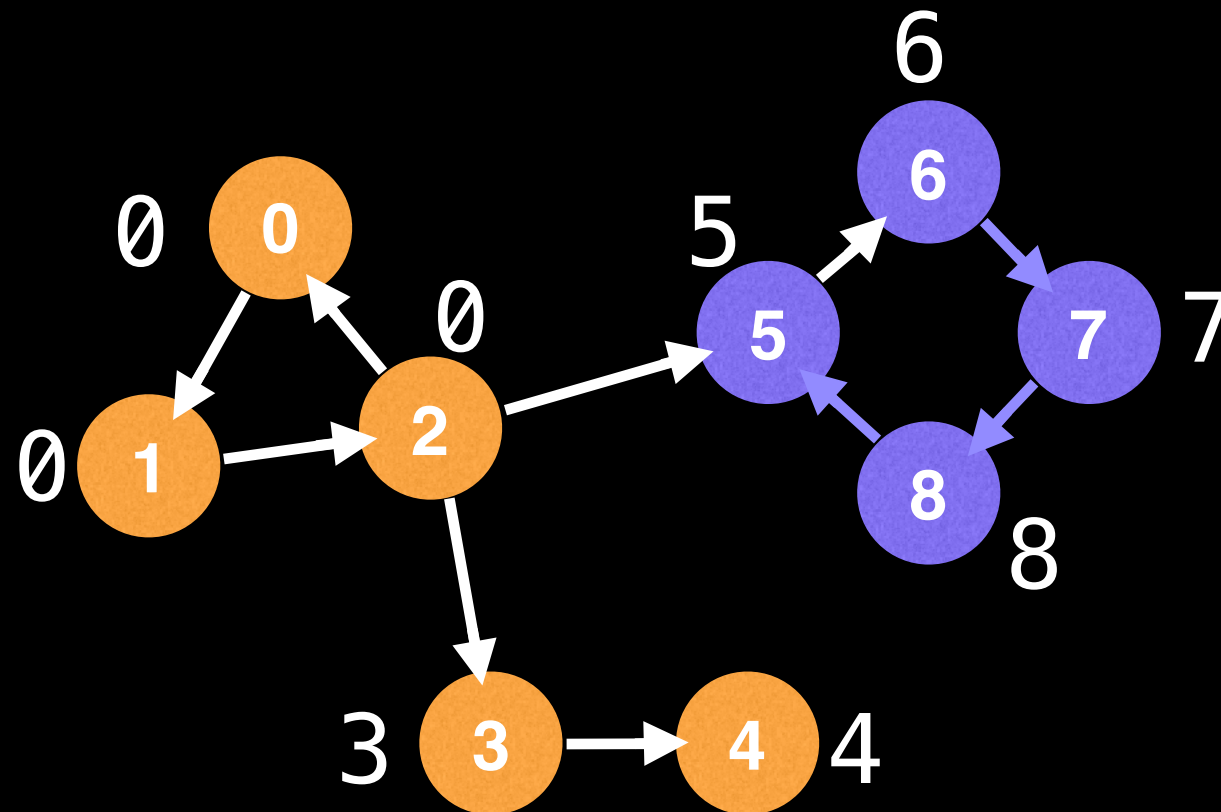


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

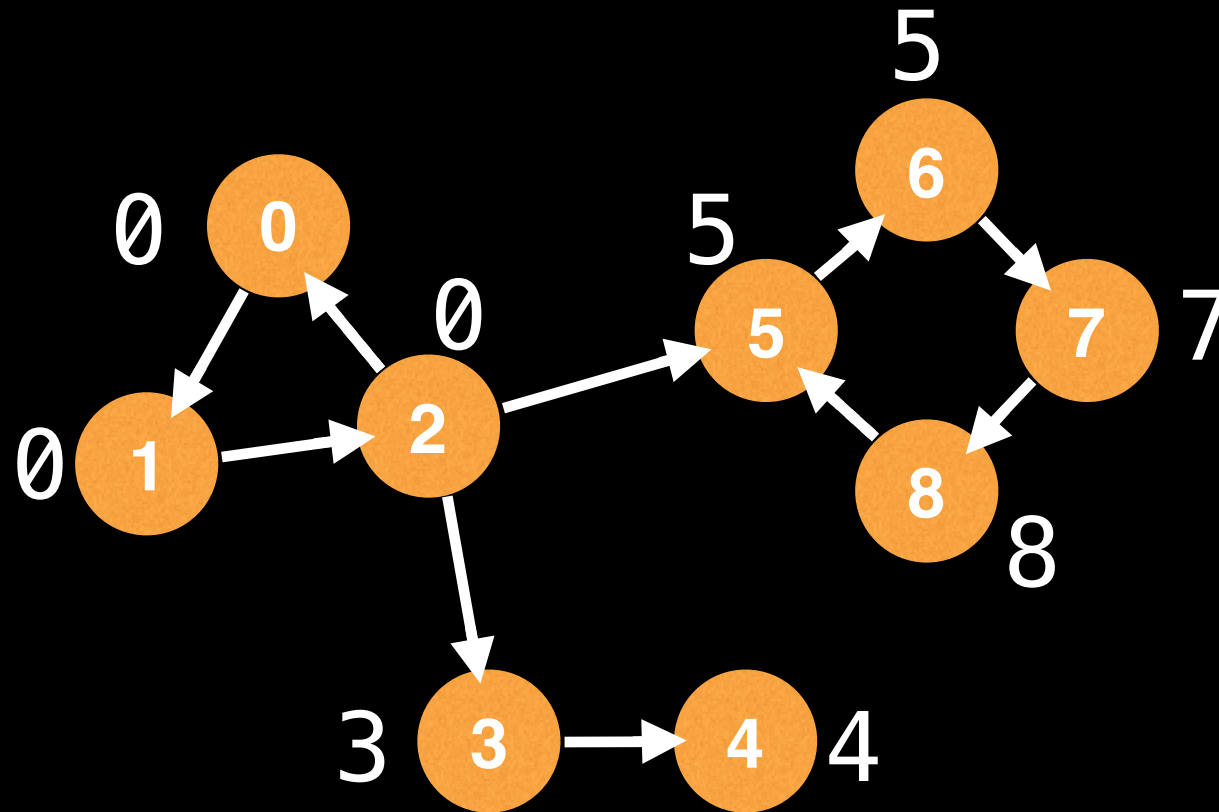


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

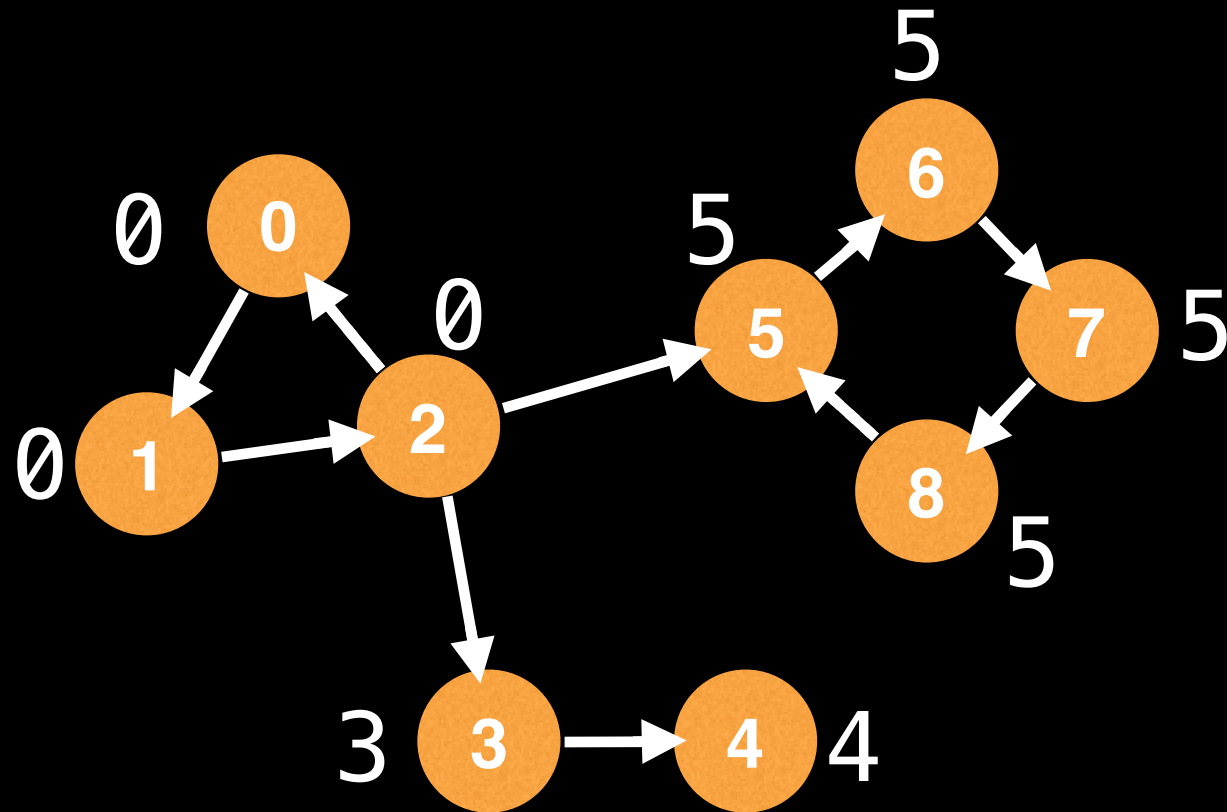


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Node 6's low-link value can be updated to 5 since node 5 is reachable from node 6.

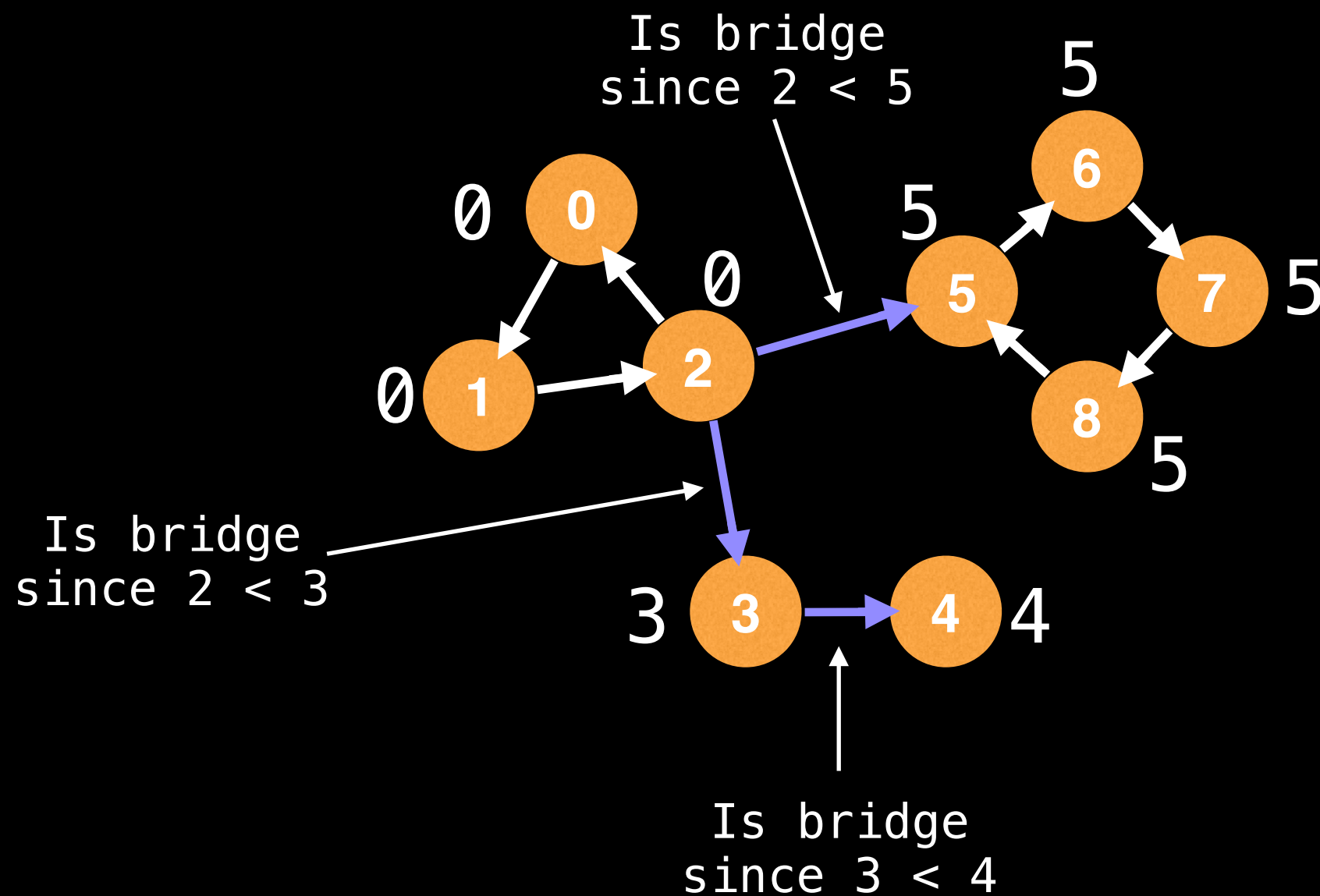


The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

—————
Undirected edge

—————→
Directed edge

Now notice that the condition for a directed edge 'e' to have nodes that belong to a bridge is when the $\text{id}(\text{e.from}) < \text{lowlink}(\text{e.to})^*$

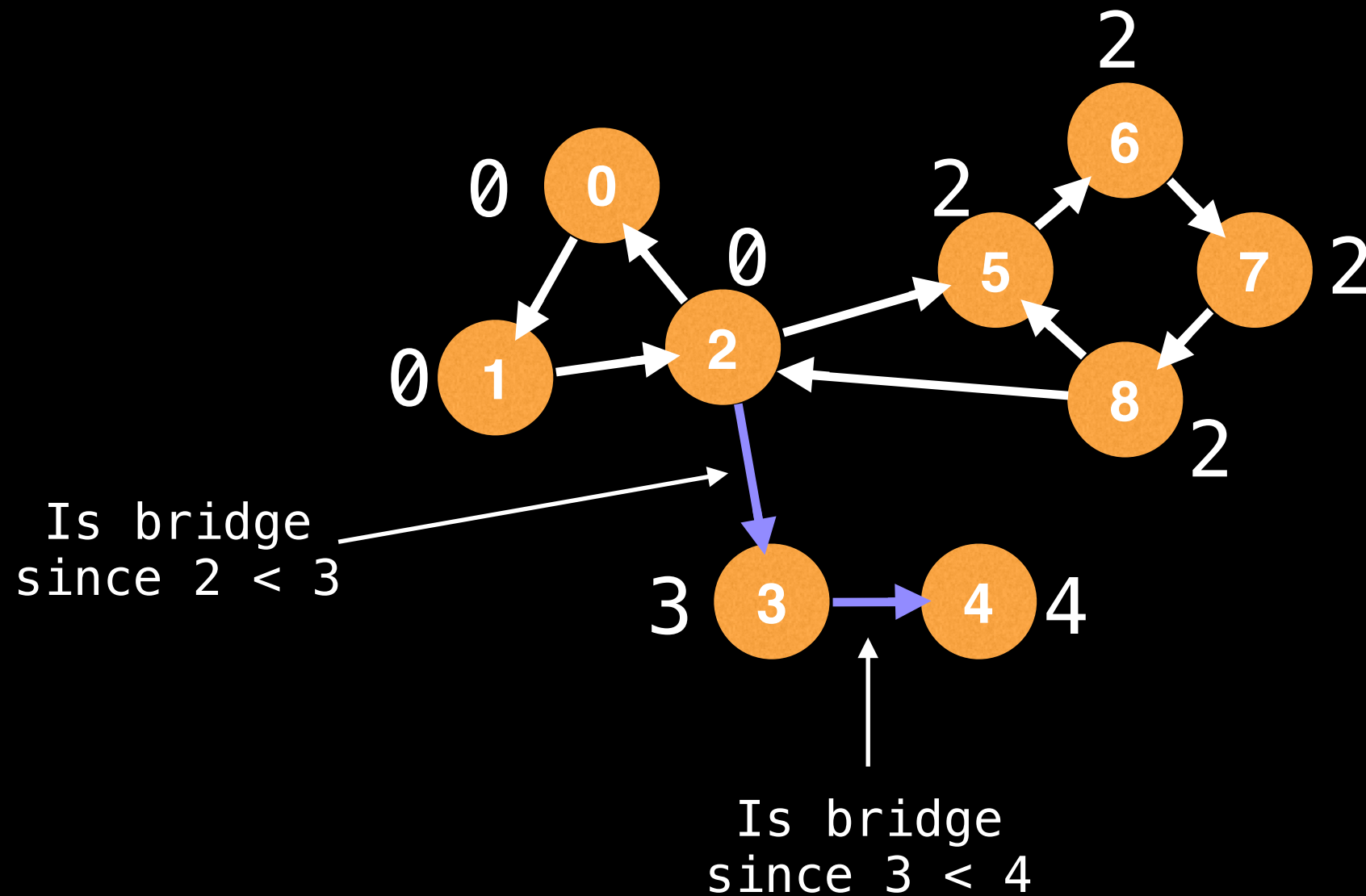


* Where e.from is the node the directed edge starts at and e.to is the node the directed edge ends at.

Undirected edge

Directed edge

Now notice that the condition for a directed edge 'e' to have nodes that belong to a bridge is when the $\text{id}(\text{e.from}) < \text{lowlink}(\text{e.to})^*$



* Where e.from is the node the directed edge starts at and e.to is the node the directed edge ends at.

Undirected edge

Directed edge

Complexity

What's the runtime of our algorithm to find bridges? Right now we're doing one DFS to label all the nodes plus V more DFSs to find all the low-link values, giving us roughly:

$$O(V(V+E))$$

Fortunately, we are able to do better by updating the low-link values in one pass for

$$O(V+E)$$

```
id = 0
g = adjacency list with undirected edges
n = size of the graph
```

```
# In these arrays index i represents node i
ids = [0, 0, ... 0, 0]          # Length n
low = [0, 0, ... 0, 0]          # Length n
visited = [false, ..., false]  # Length n
```

```
function findBridges():
    bridges = []
    # Finds all bridges in the graph across
    # various connected components.
    for (i = 0; i < n; i = i + 1):
        if (!visited[i]):
            dfs(i, -1, bridges)
    return bridges
```

```
id = 0
g = adjacency list with undirected edges
n = size of the graph
```

```
# In these arrays index i represents node i
ids = [0, 0, ... 0, 0]          # Length n
low = [0, 0, ... 0, 0]          # Length n
visited = [false, ..., false]  # Length n
```

```
function findBridges():
    bridges = []
    # Finds all bridges in the graph across
    # various connected components.
    for (i = 0; i < n; i = i + 1):
        if (!visited[i]):
            dfs(i, -1, bridges)
    return bridges
```



```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
    visited[at] = true
    id = id + 1
    low[at] = ids[at] = id

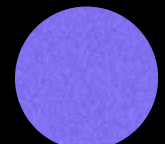
    # For each edge from node 'at' to node 'to'
    for (to : g[at]):
        if to == parent: continue
        if (!visited[to]):
            dfs(to, at, bridges)
            low[at] = min(low[at], low[to])
            if (ids[at] < low[to]):
                bridges.add(at)
                bridges.add(to)
        else:
            low[at] = min(low[at], ids[to])
```

```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
    visited[at] = true
    id = id + 1
    low[at] = ids[at] = id

    # For each edge from node 'at' to node 'to'
    for (to : g[at]):
        if to == parent: continue
        if (!visited[to]):
            dfs(to, at, bridges)
            low[at] = min(low[at], low[to])
            if (ids[at] < low[to]):
                bridges.add(at)
                bridges.add(to)
        else:
            low[at] = min(low[at], ids[to])
```

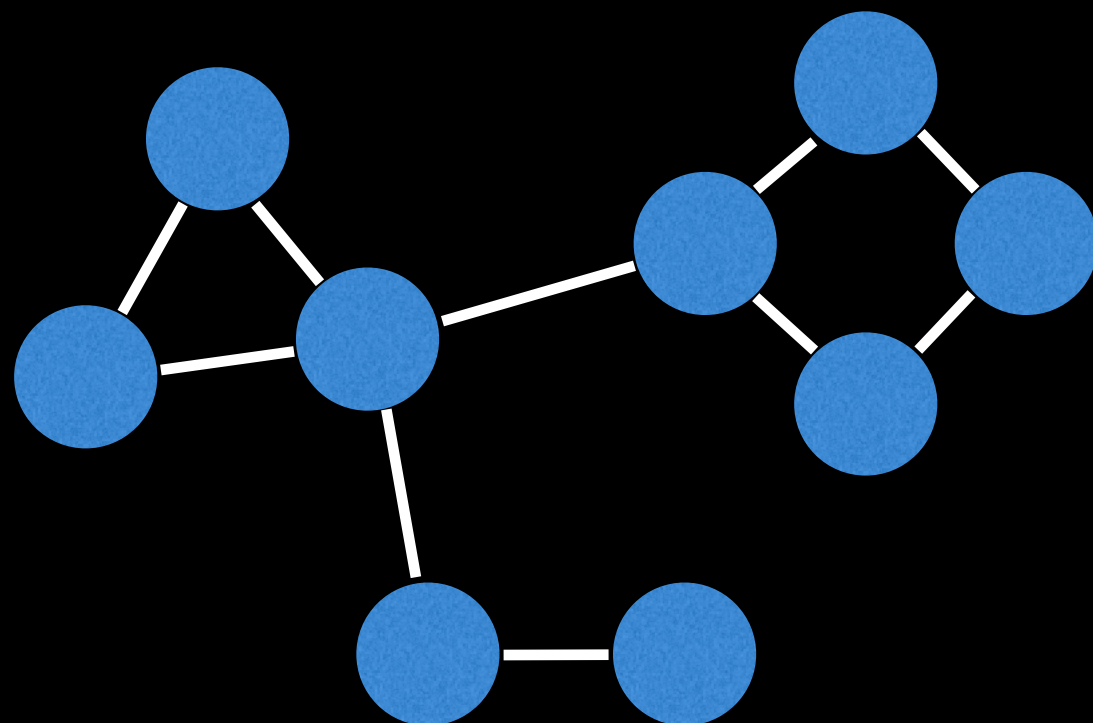
```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
    visited[at] = true
    id = id + 1
    low[at] = ids[at] = id

    # For each edge from node 'at' to node 'to'
    for (to : g[at]):
        if to == parent: continue
        if (!visited[to]):
            dfs(to, at, bridges)
            low[at] = min(low[at], low[to])
            if (ids[at] < low[to]):
                bridges.add(at)
                bridges.add(to)
        else:
            low[at] = min(low[at], ids[to])
```

 Current

 Visited

 Unvisited



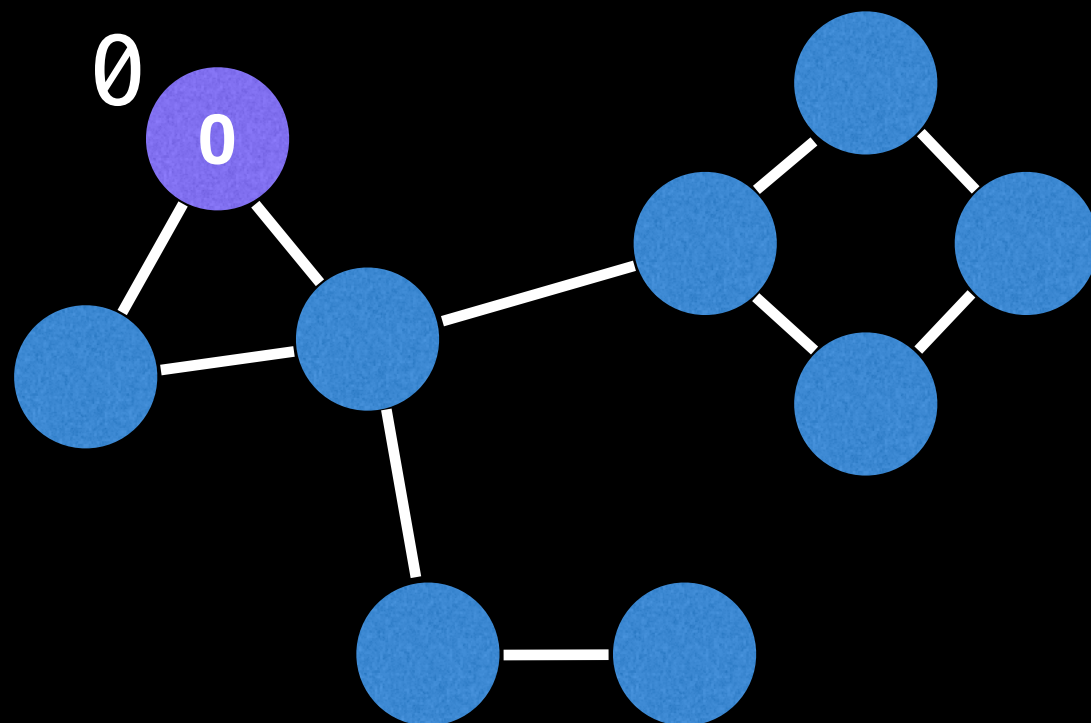

Undirected edge


Directed edge

● Current

● Visited

● Unvisited



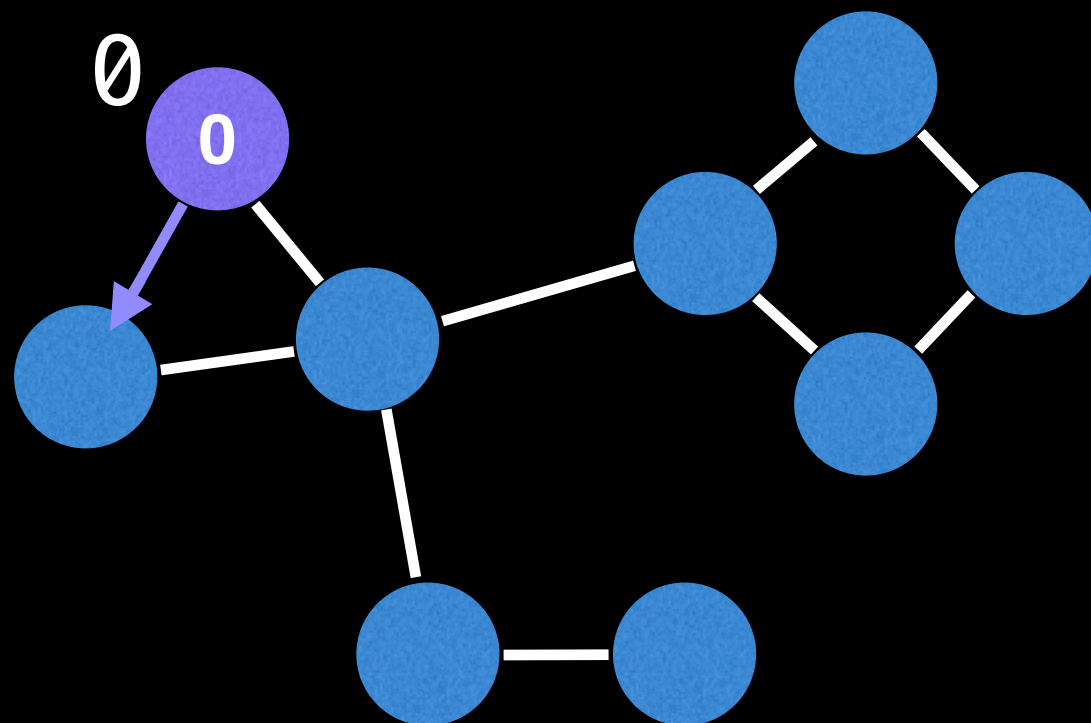
—————
Undirected edge

—————→
Directed edge

● Current

● Visited

● Unvisited



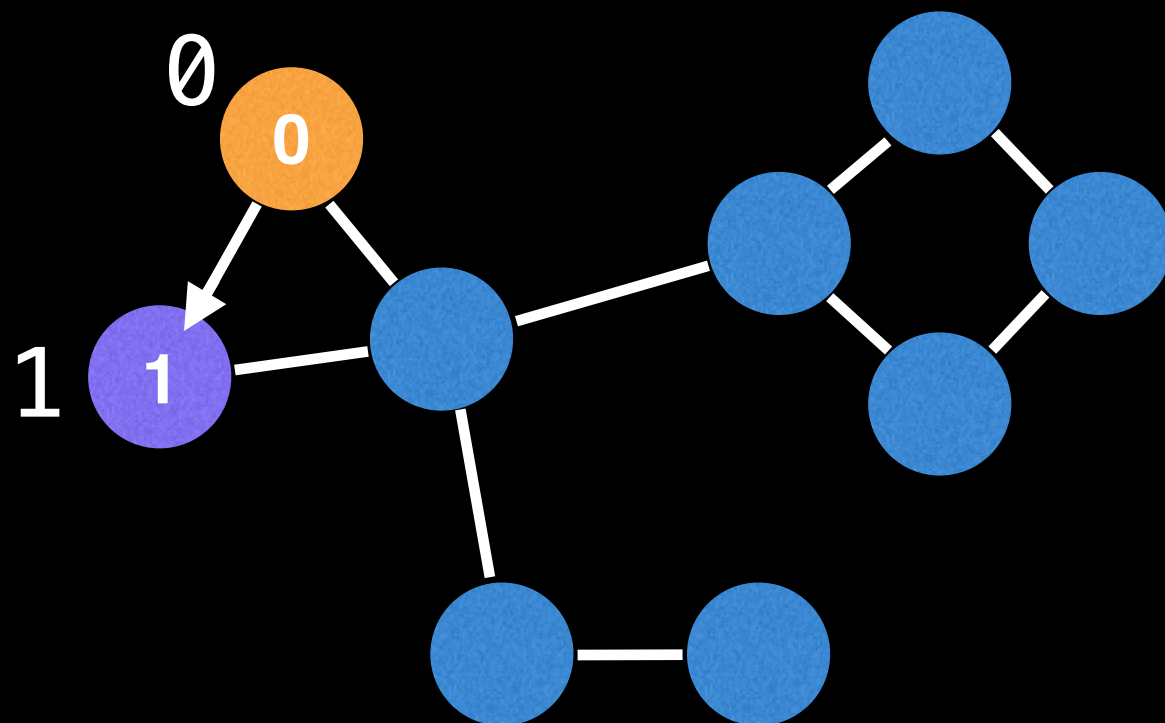
Undirected edge

Directed edge

● Current

● Visited

● Unvisited



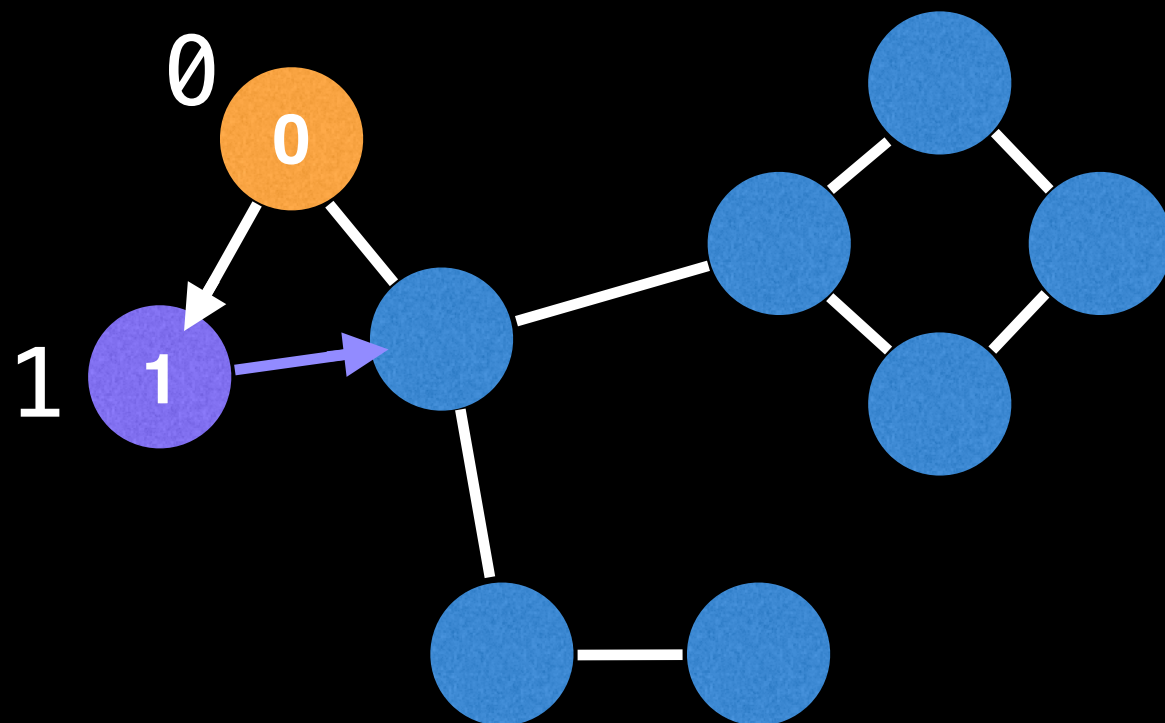
—————
Undirected edge

—————→
Directed edge

● Current

● Visited

● Unvisited



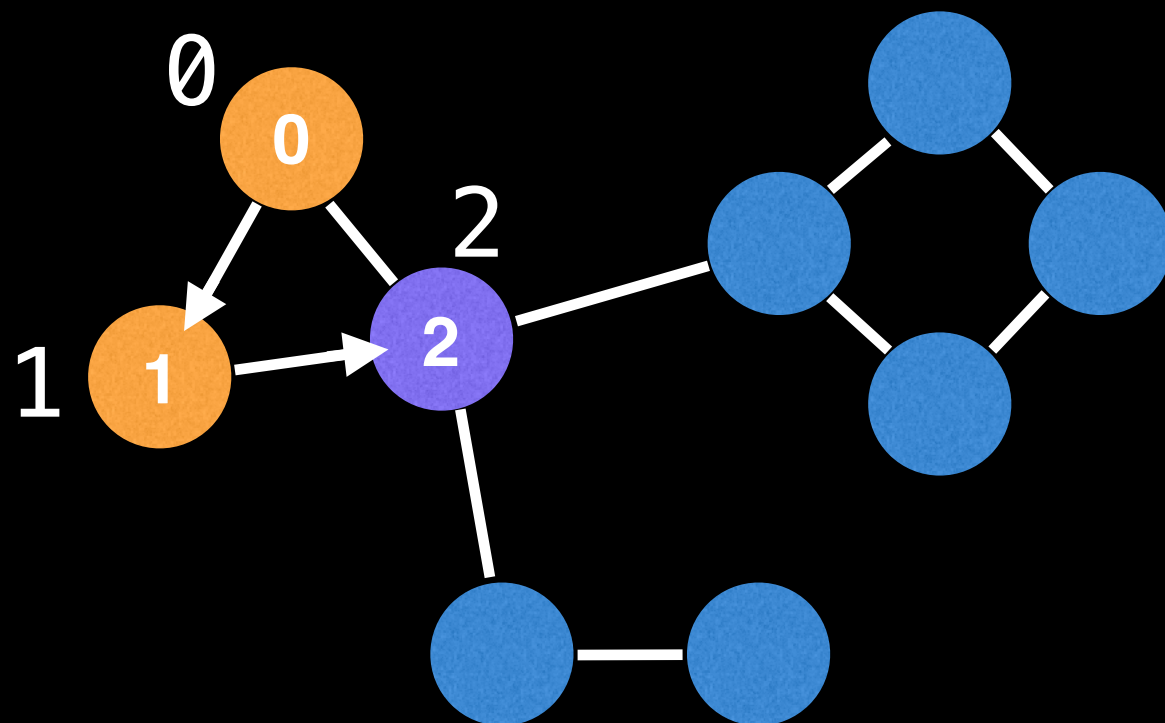
Undirected edge

Directed edge

● Current

● Visited

● Unvisited



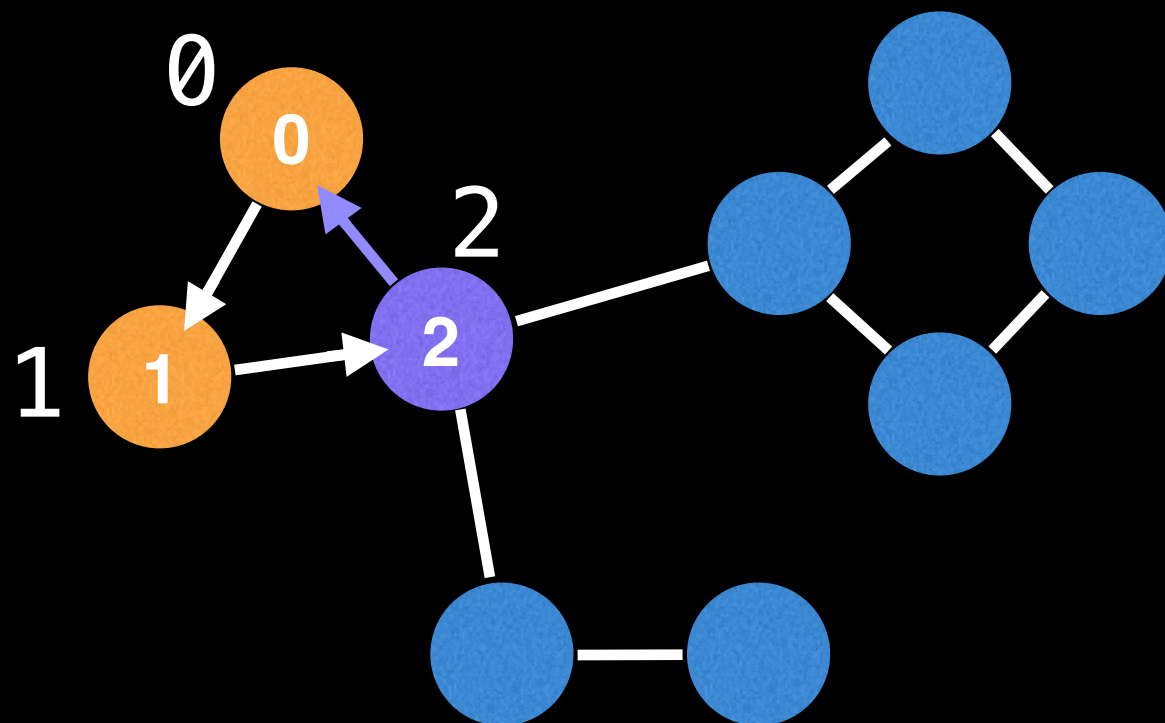
—————
Undirected edge

—————→
Directed edge

● Current

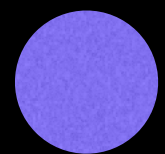
● Visited

● Unvisited



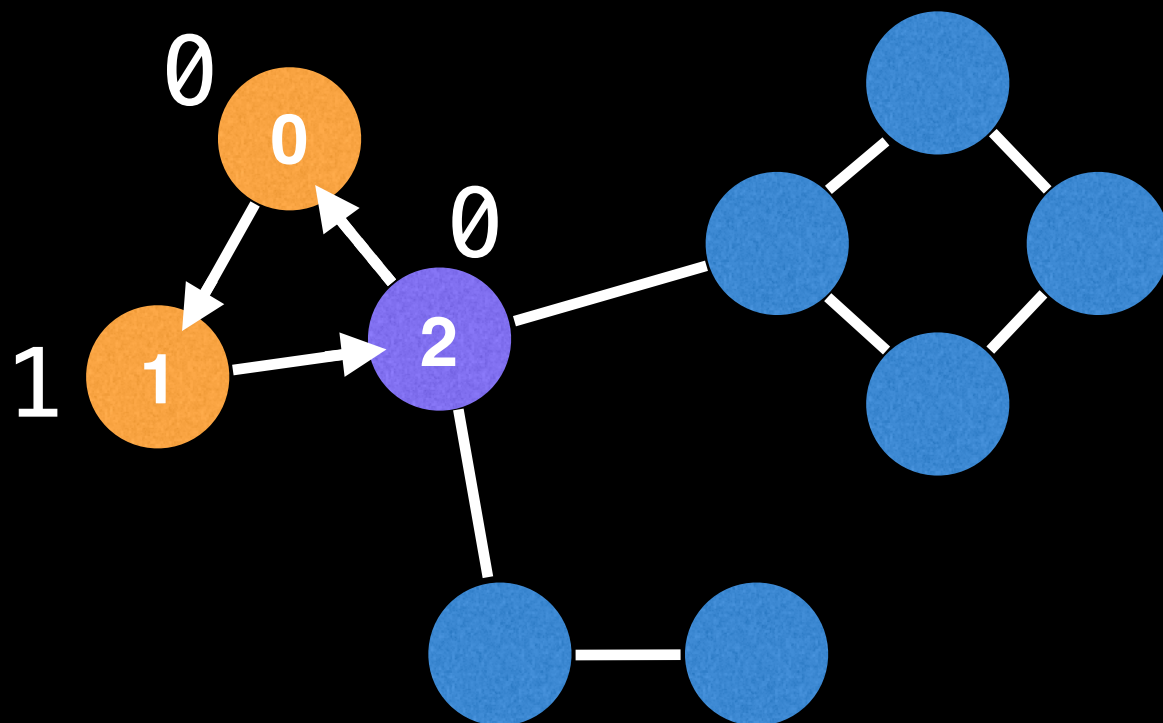
—————
Undirected edge

—————→
Directed edge

 Current

 Visited

 Unvisited

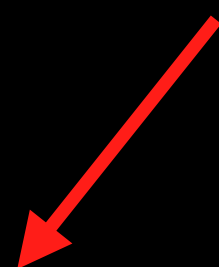



Undirected edge


Directed edge

```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
    visited[at] = true
    id = id + 1
    low[at] = ids[at] = id

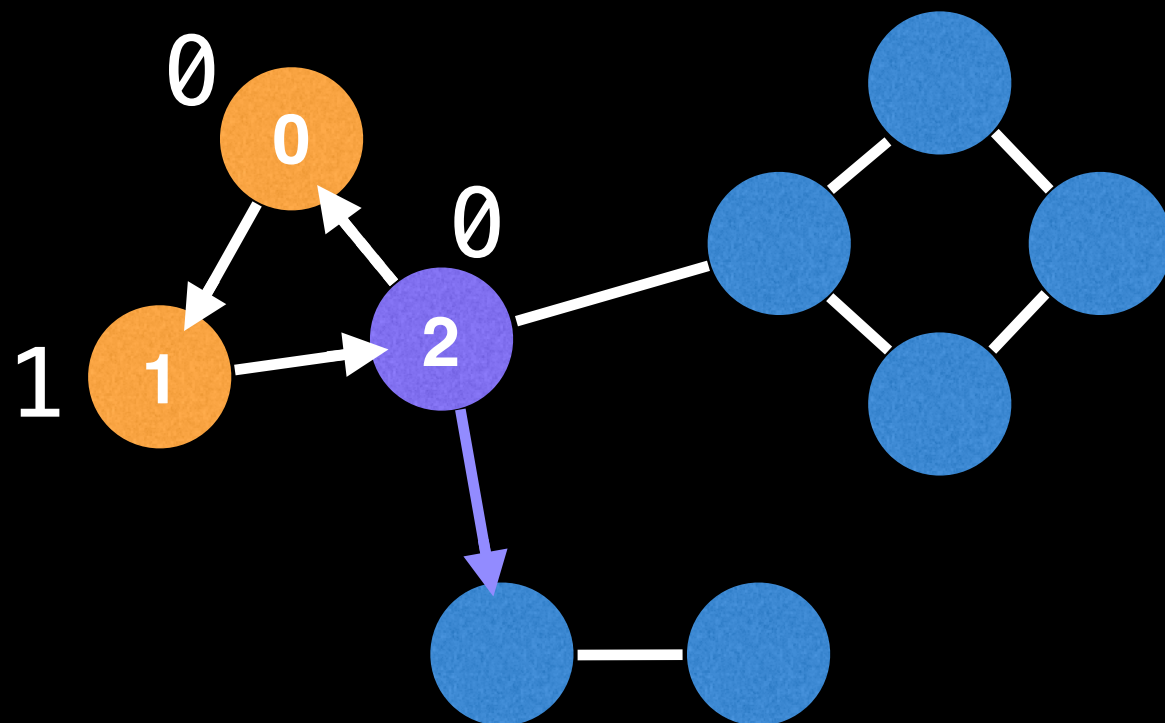
    # For each edge from node 'at' to node 'to'
    for (to : g[at]):
        if to == parent: continue
        if (!visited[to]):
            dfs(to, at, bridges)
            low[at] = min(low[at], low[to])
            if (ids[at] < low[to]):
                bridges.add(at)
                bridges.add(to)
        else:
            low[at] = min(low[at], ids[to])
```



● Current

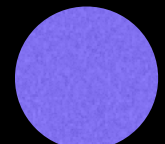
● Visited

● Unvisited



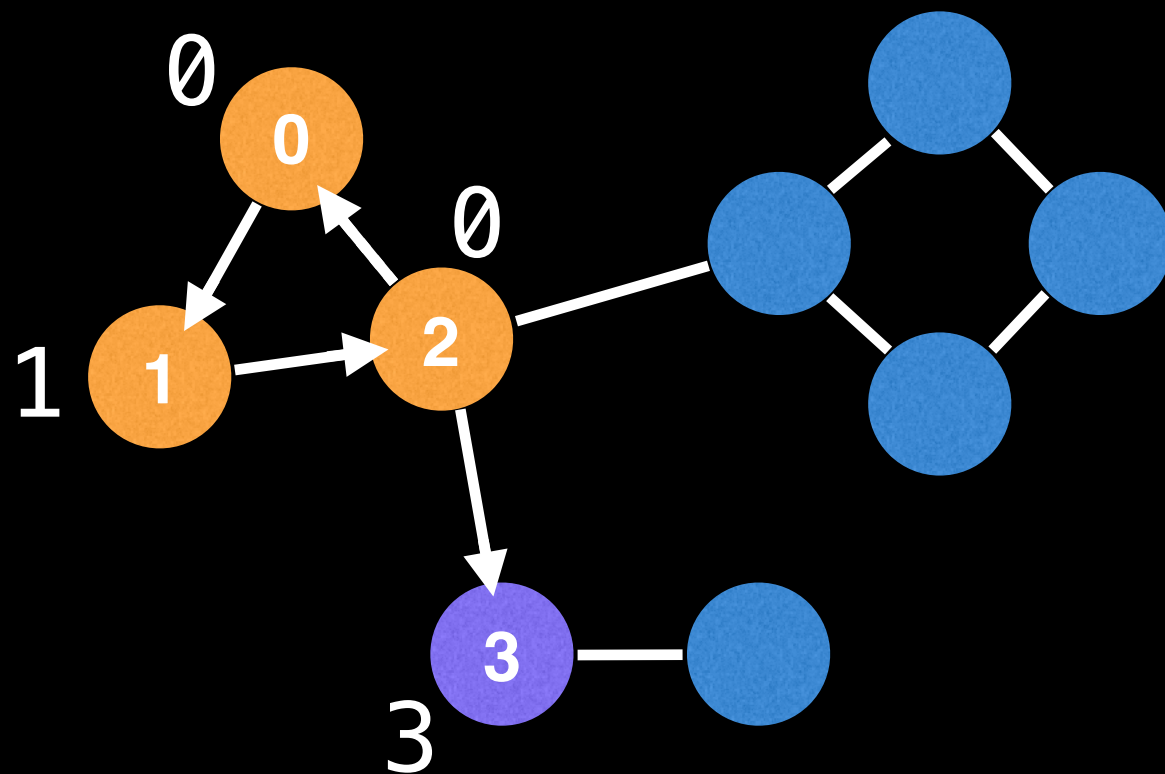
—————
Undirected edge

—————→
Directed edge

 Current

 Visited

 Unvisited



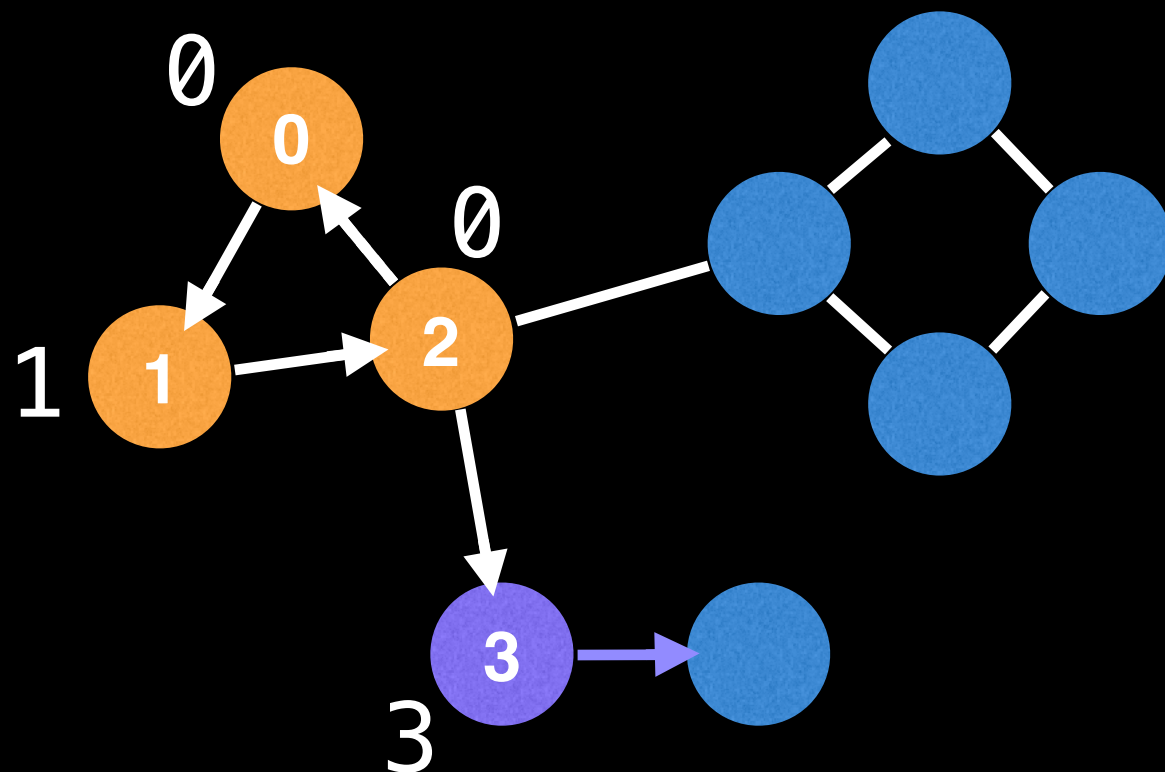

Undirected edge


Directed edge

● Current

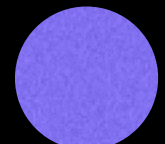
● Visited

● Unvisited



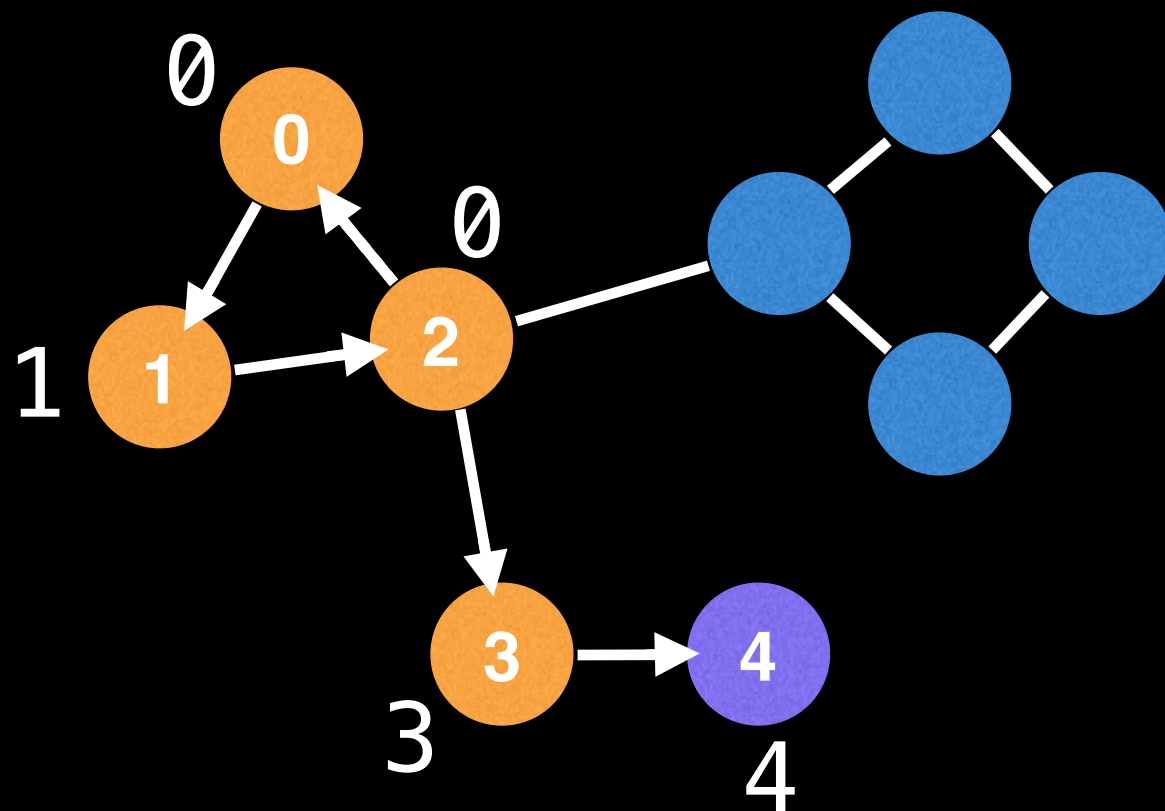
—————
Undirected edge

—————→
Directed edge

 Current

 Visited

 Unvisited



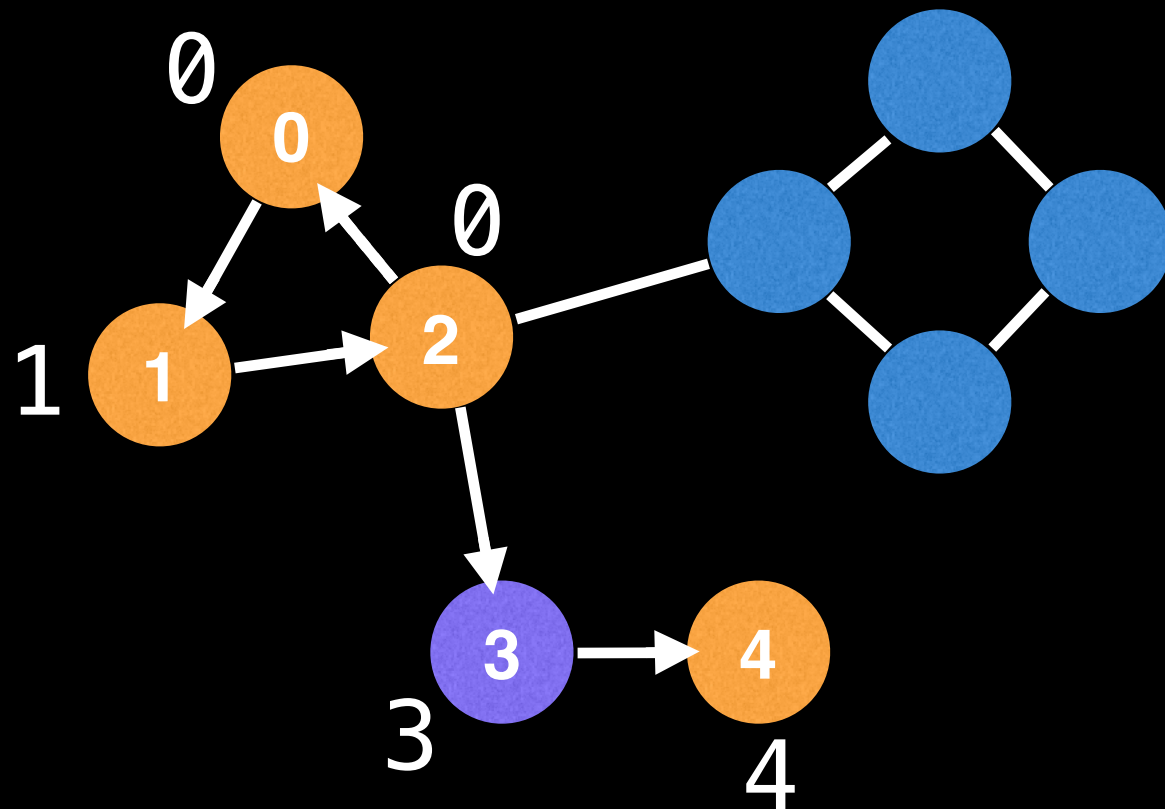

Undirected edge


Directed edge

● Current

● Visited

● Unvisited



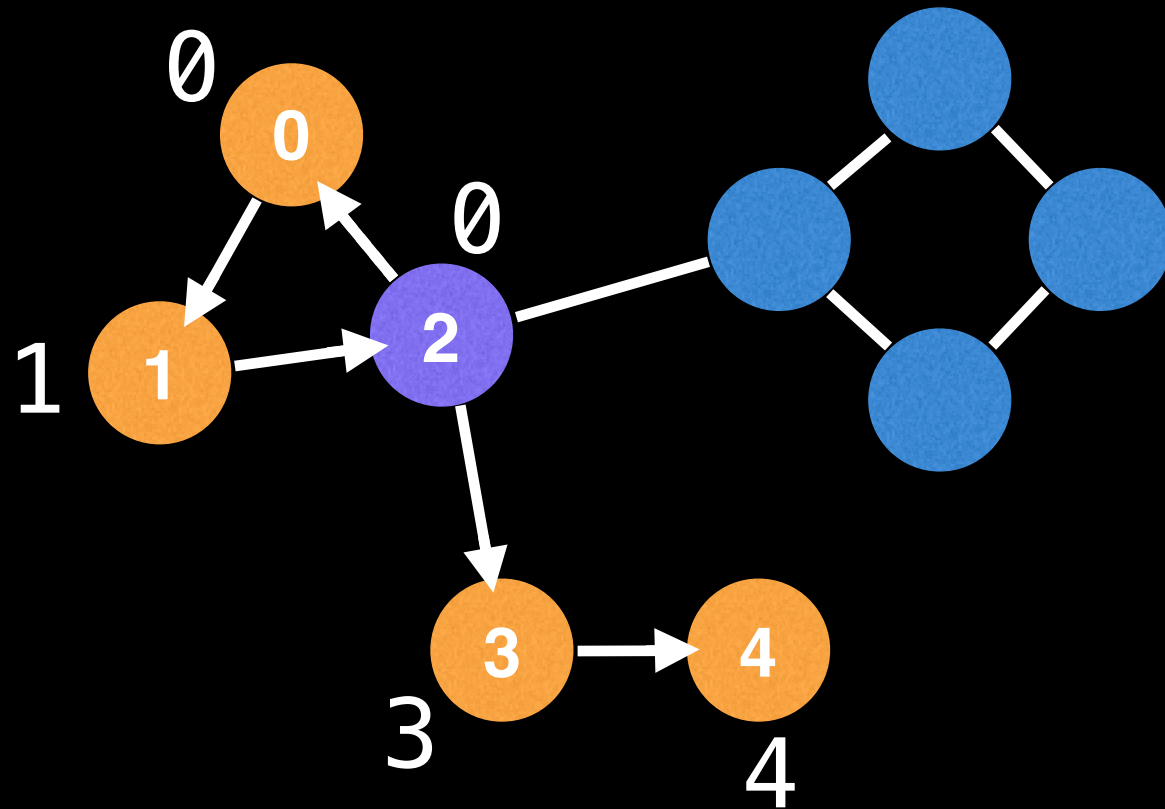
—————
Undirected edge

—————→
Directed edge

● Current

● Visited

● Unvisited



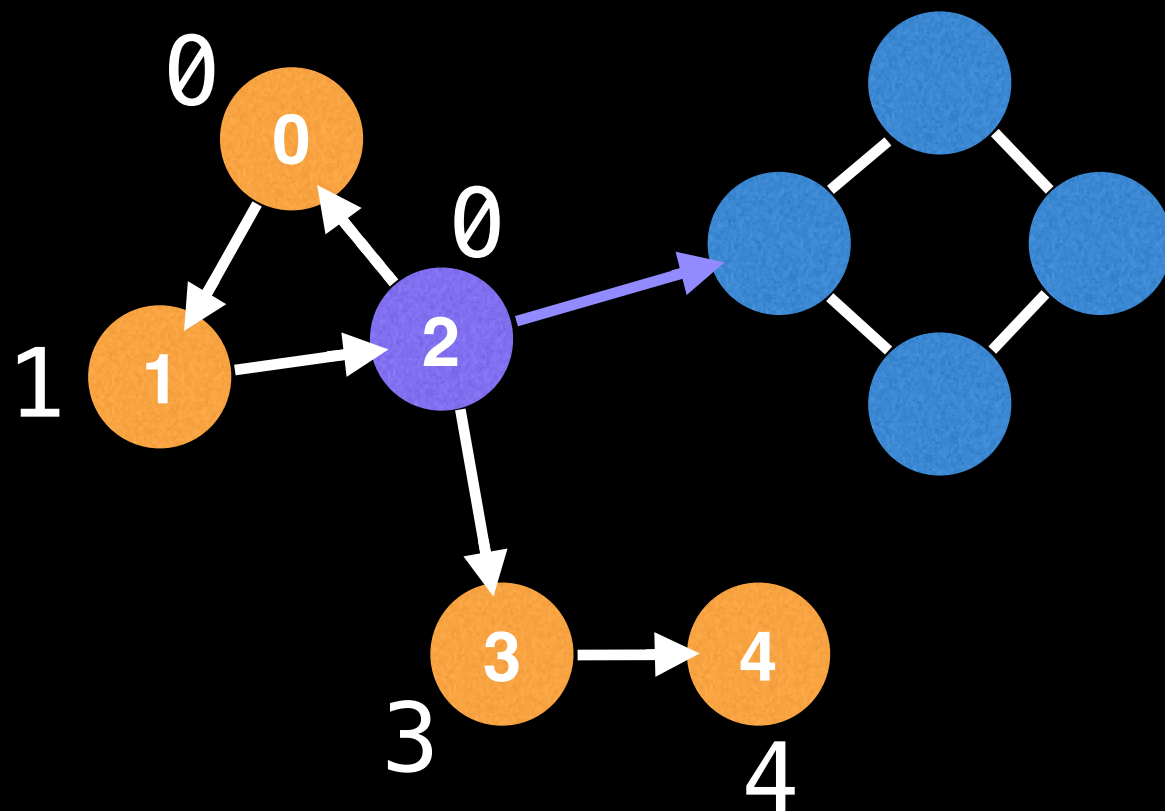
Undirected edge

Directed edge

● Current

● Visited

● Unvisited



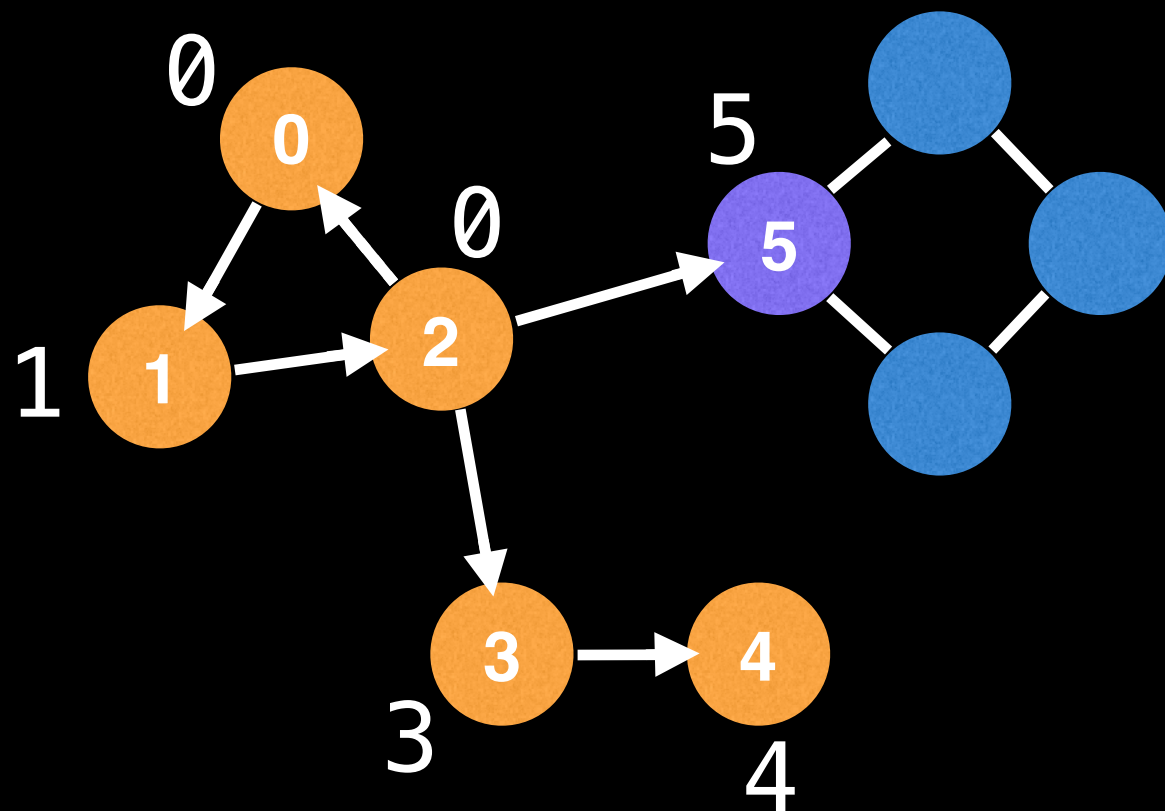
Undirected edge

Directed edge

● Current

● Visited

● Unvisited



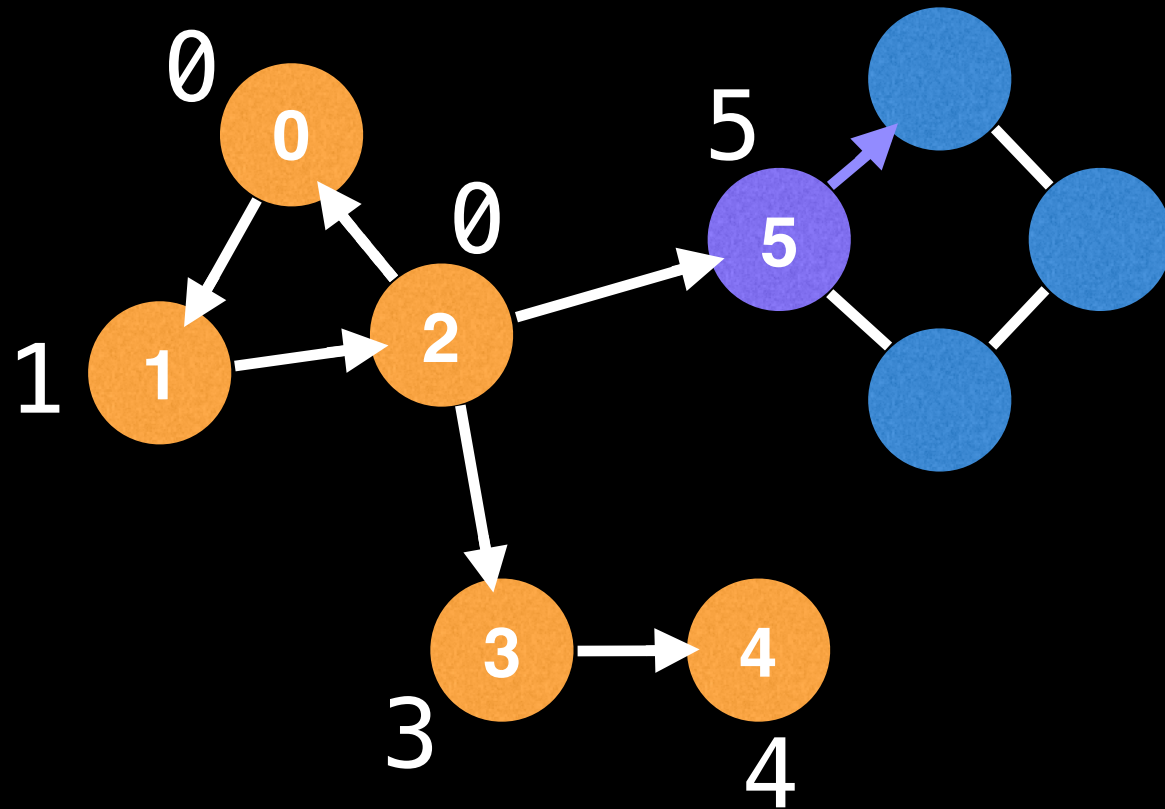
—————
Undirected edge

—————→
Directed edge

● Current

● Visited

● Unvisited



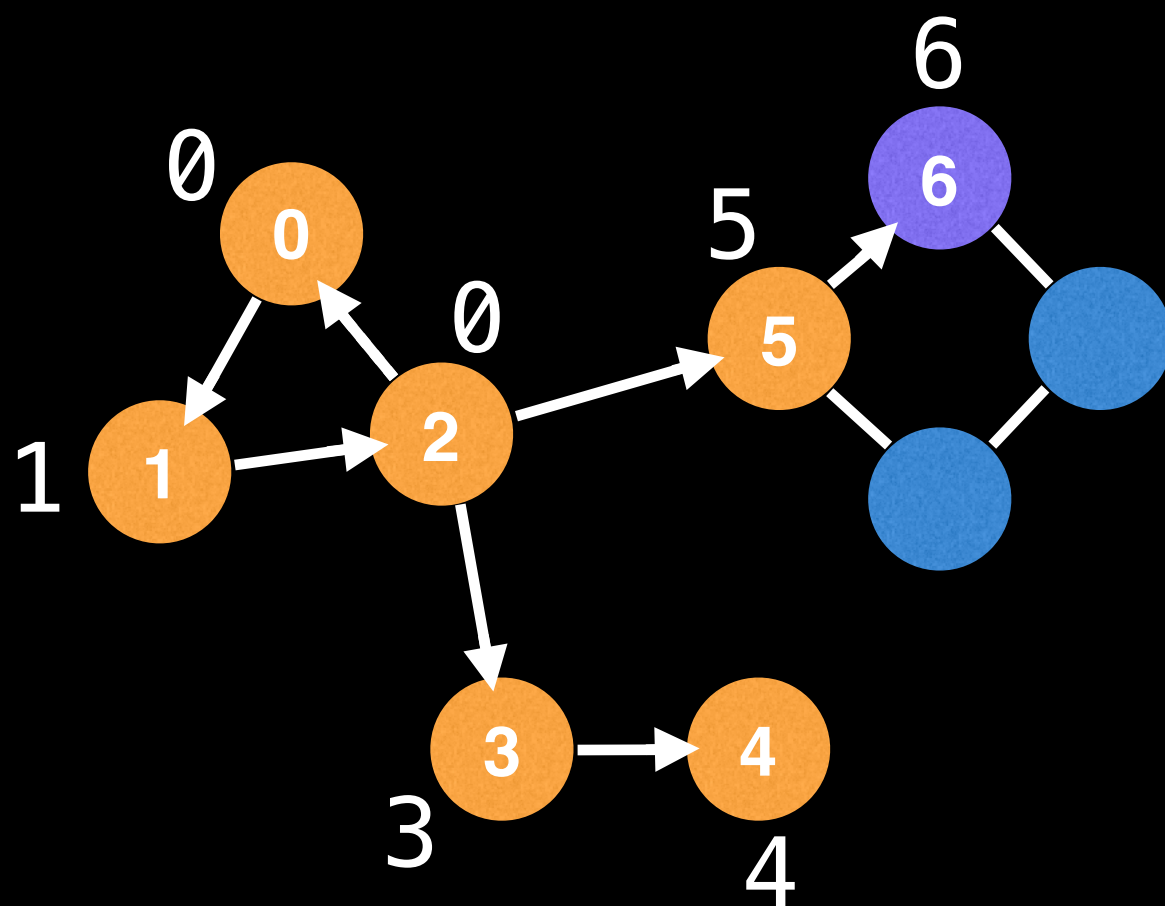
—————
Undirected edge

—————→
Directed edge

● Current

● Visited

● Unvisited



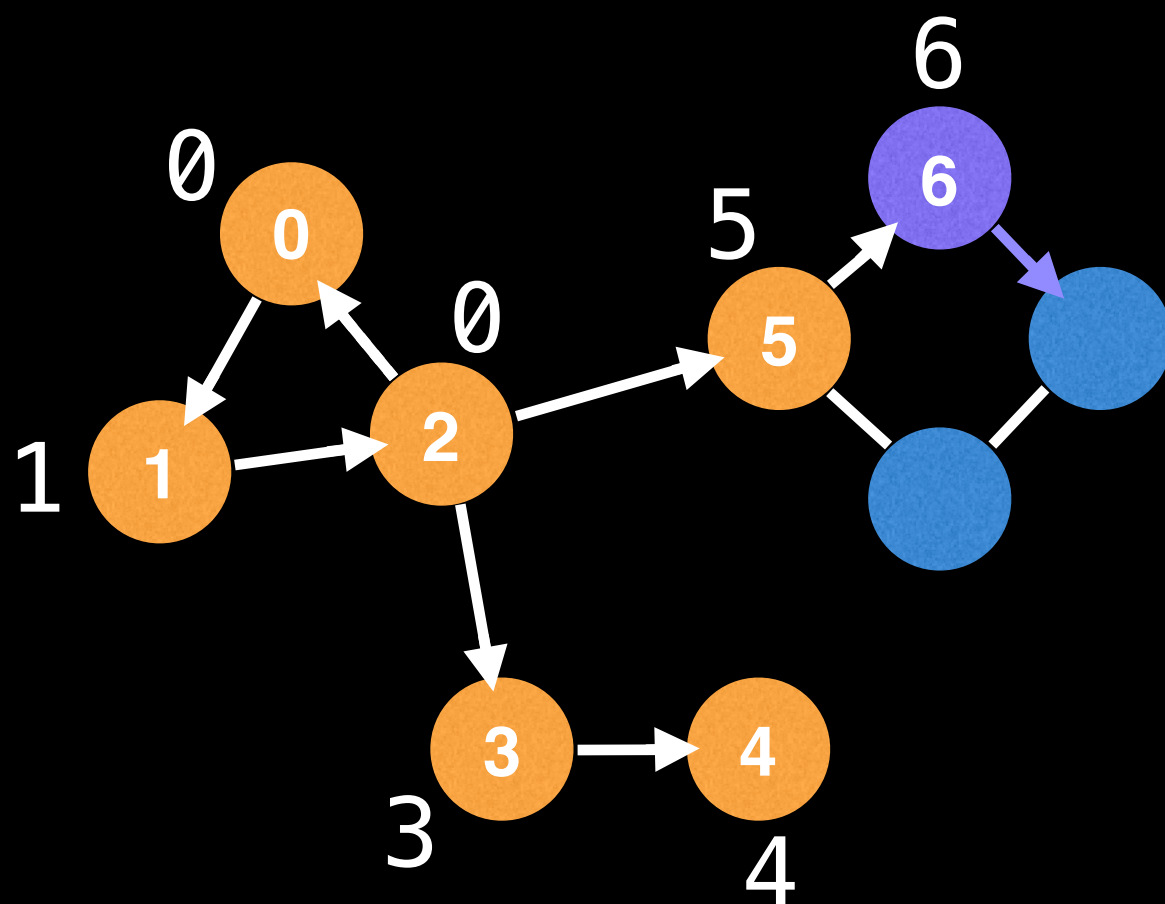
—————
Undirected edge

—————→
Directed edge

● Current

● Visited

● Unvisited



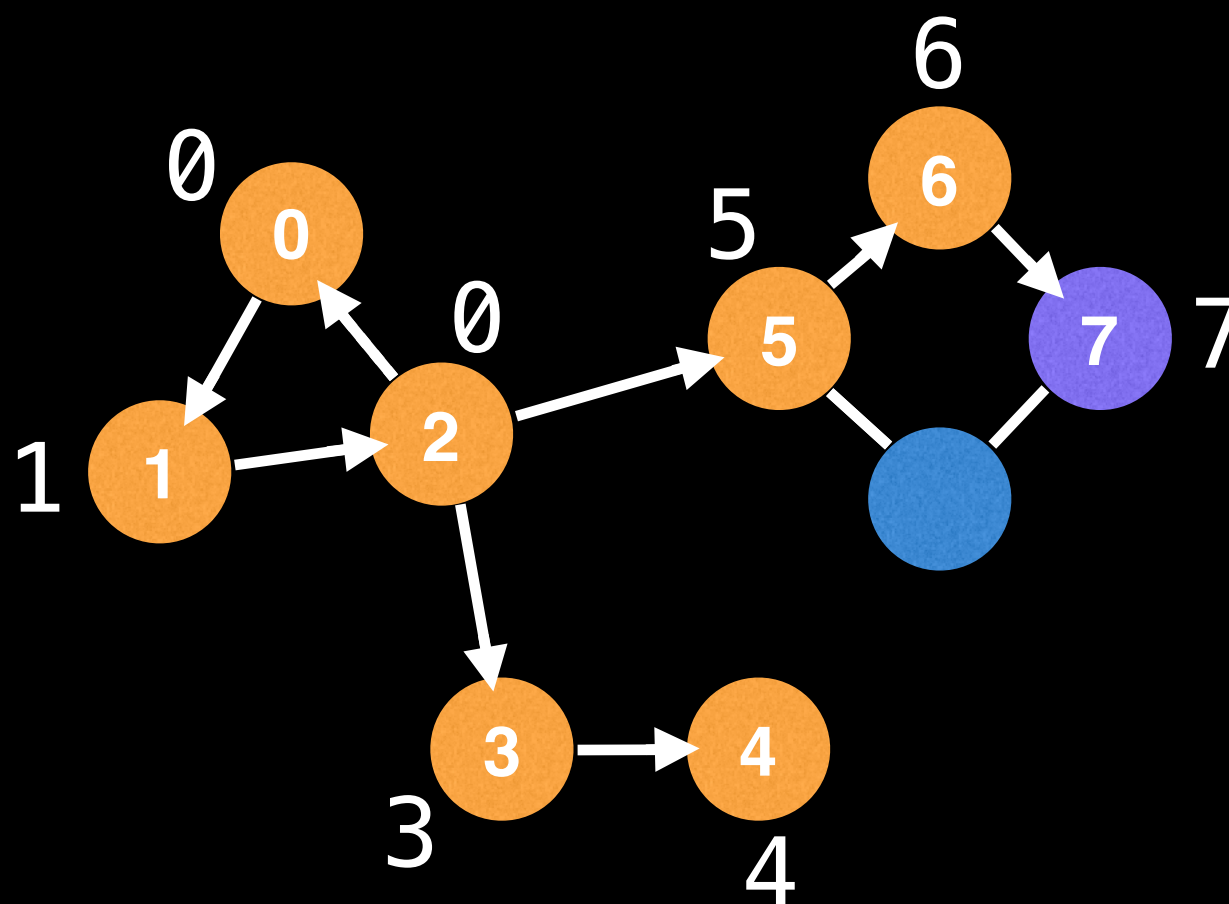
—————
Undirected edge

—————→
Directed edge

● Current

● Visited

● Unvisited



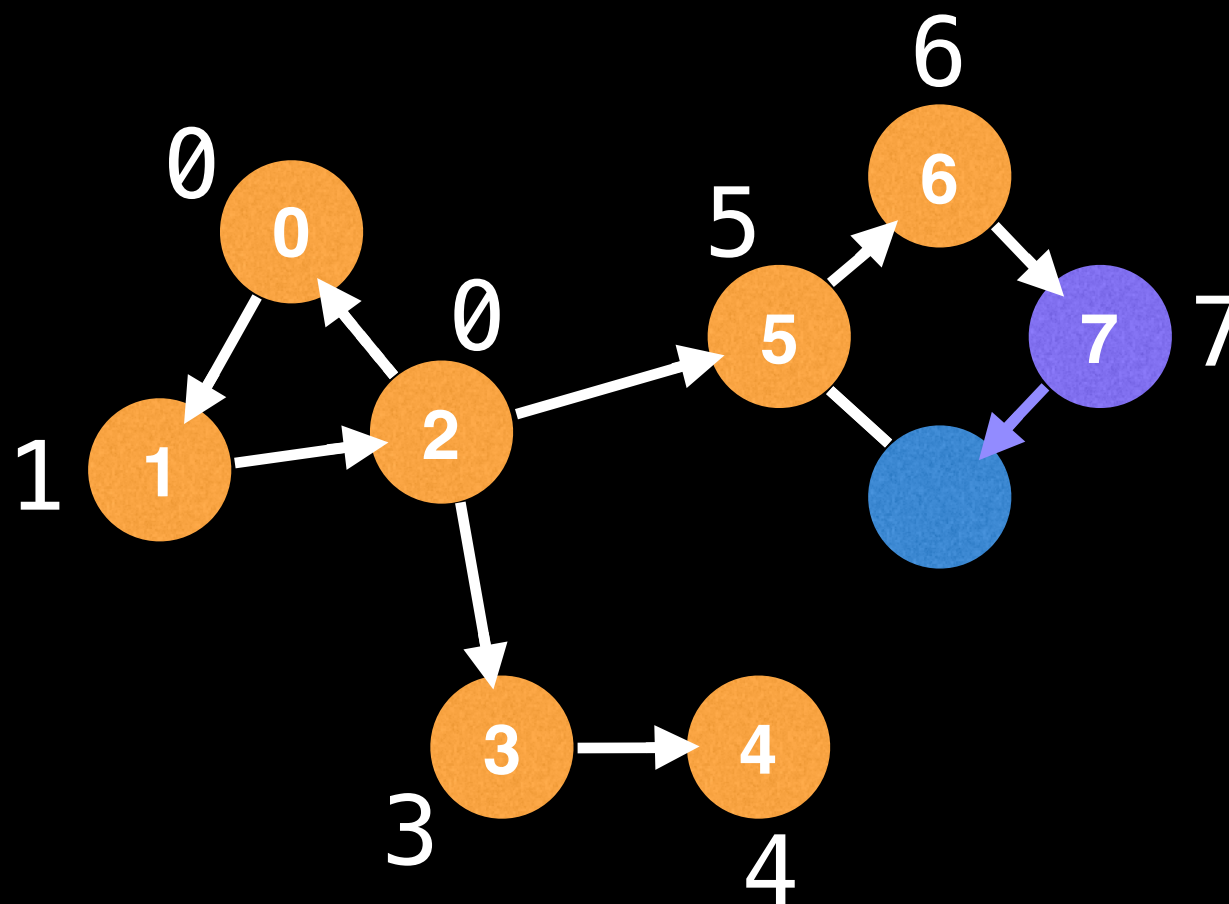
—————
Undirected edge

—————→
Directed edge

● Current

● Visited

● Unvisited



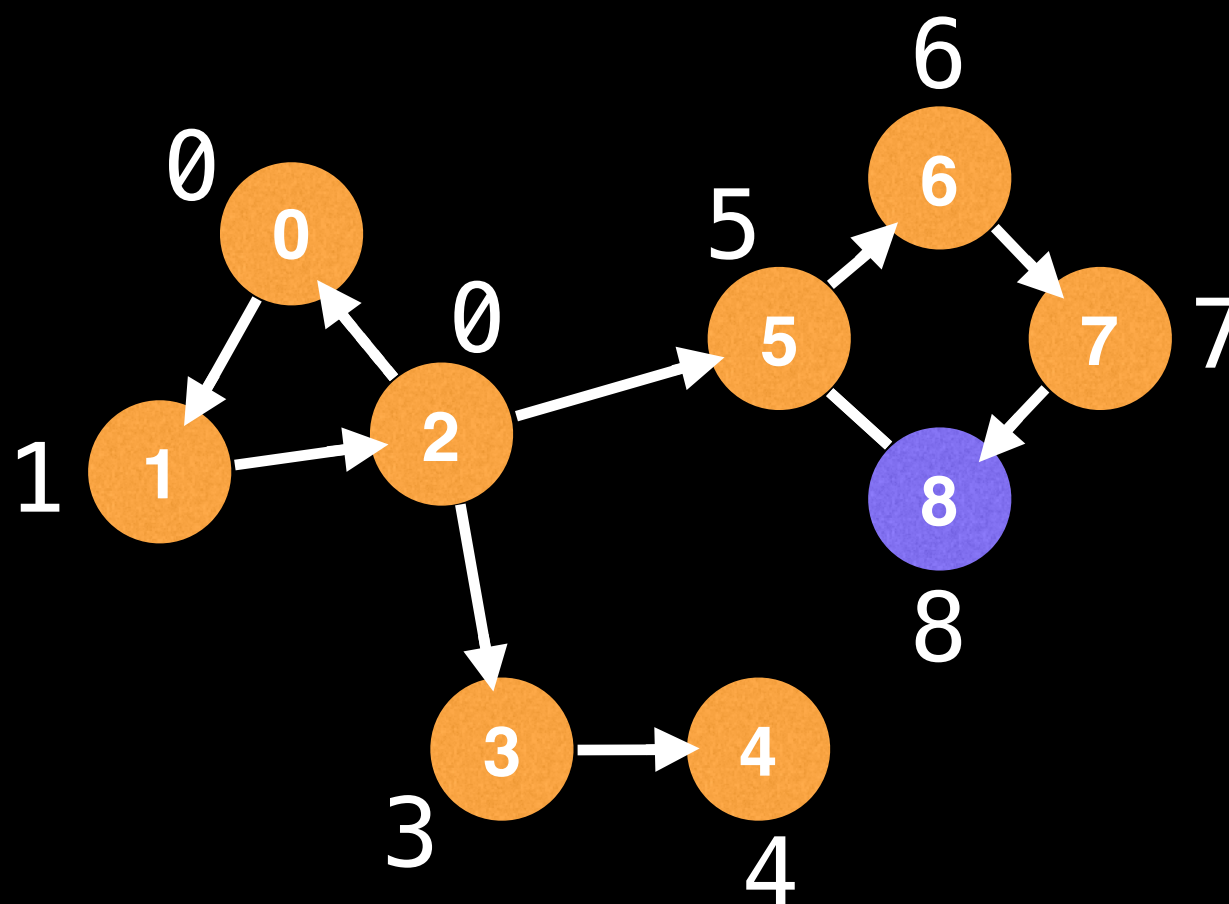
Undirected edge

Directed edge

● Current

● Visited

● Unvisited



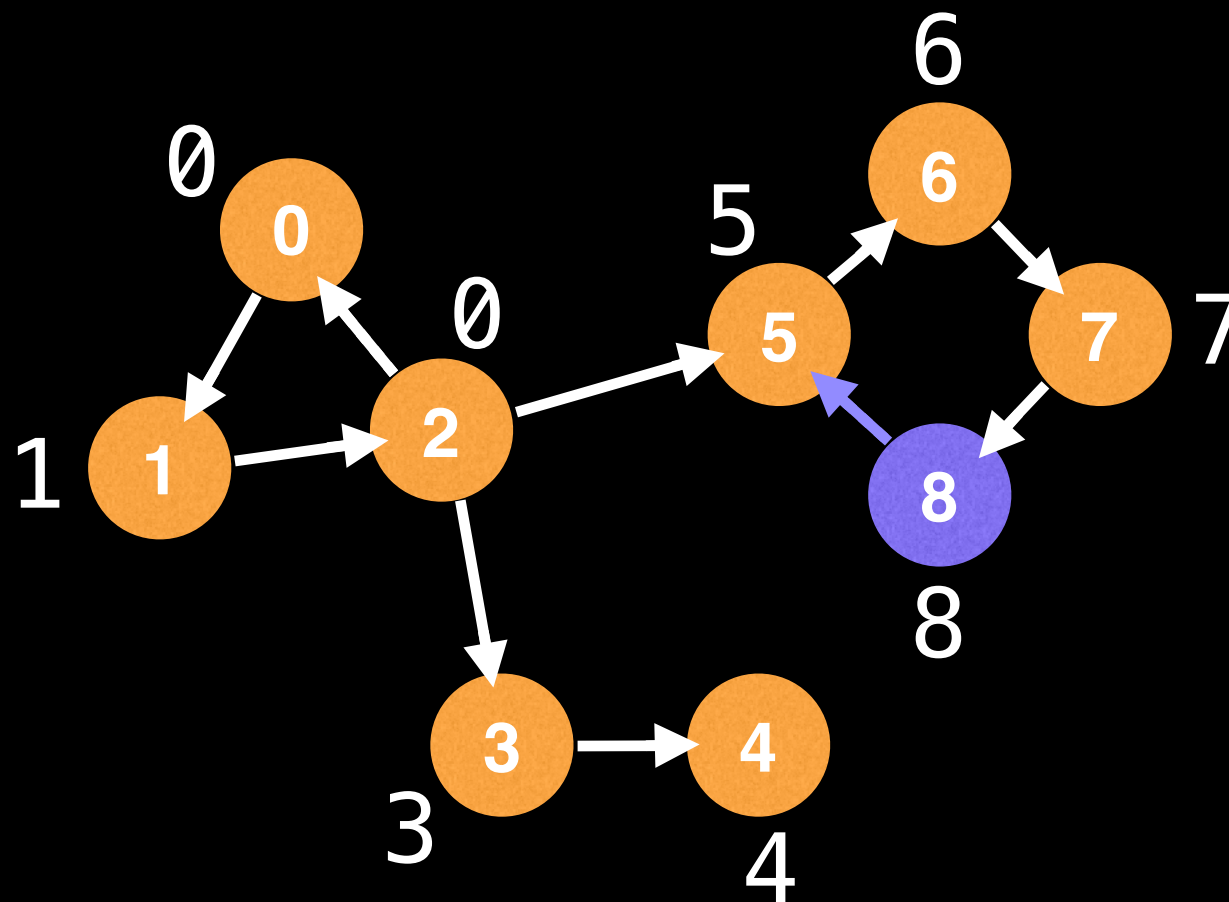
—————
Undirected edge

—————→
Directed edge

● Current

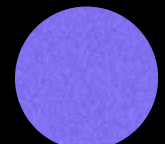
● Visited

● Unvisited



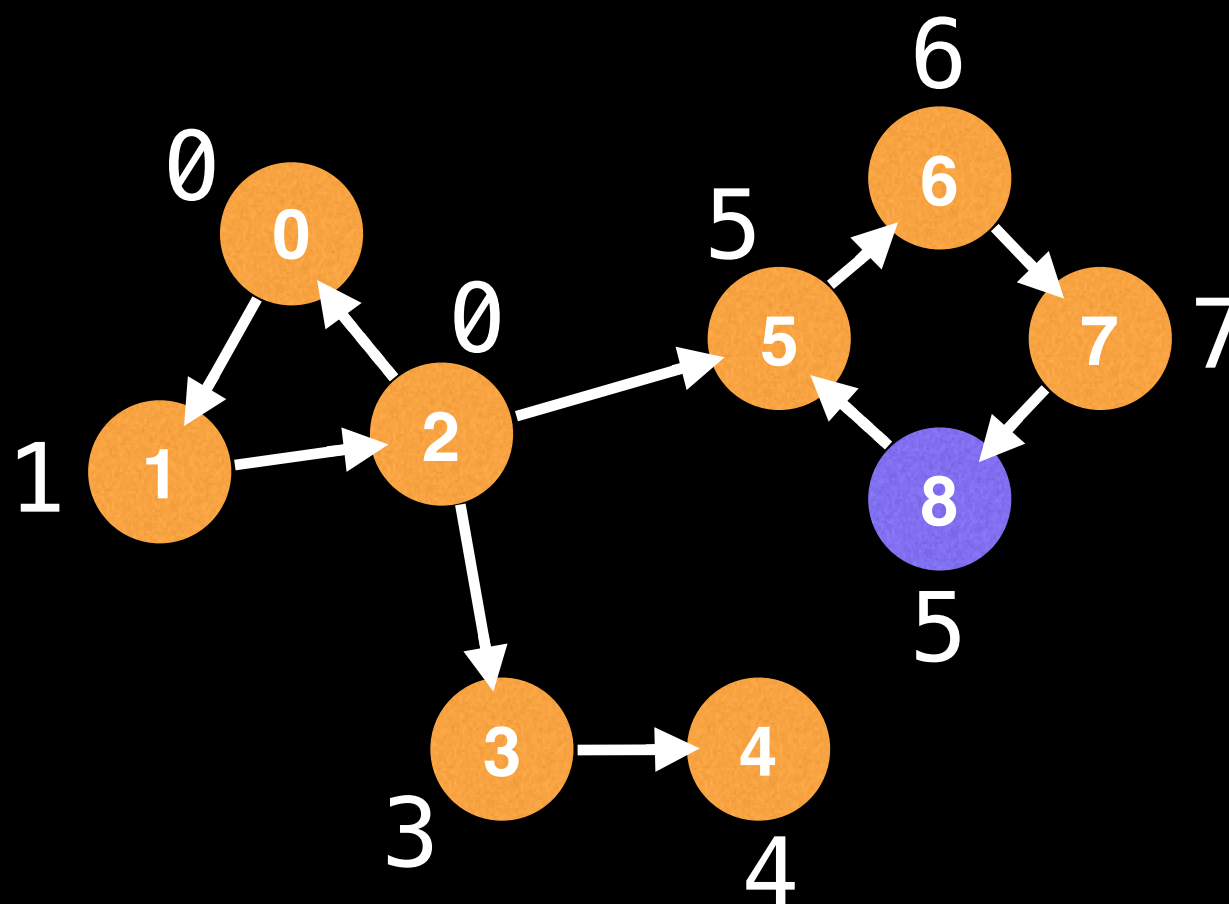
Undirected edge

Directed edge

 Current

 Visited

 Unvisited



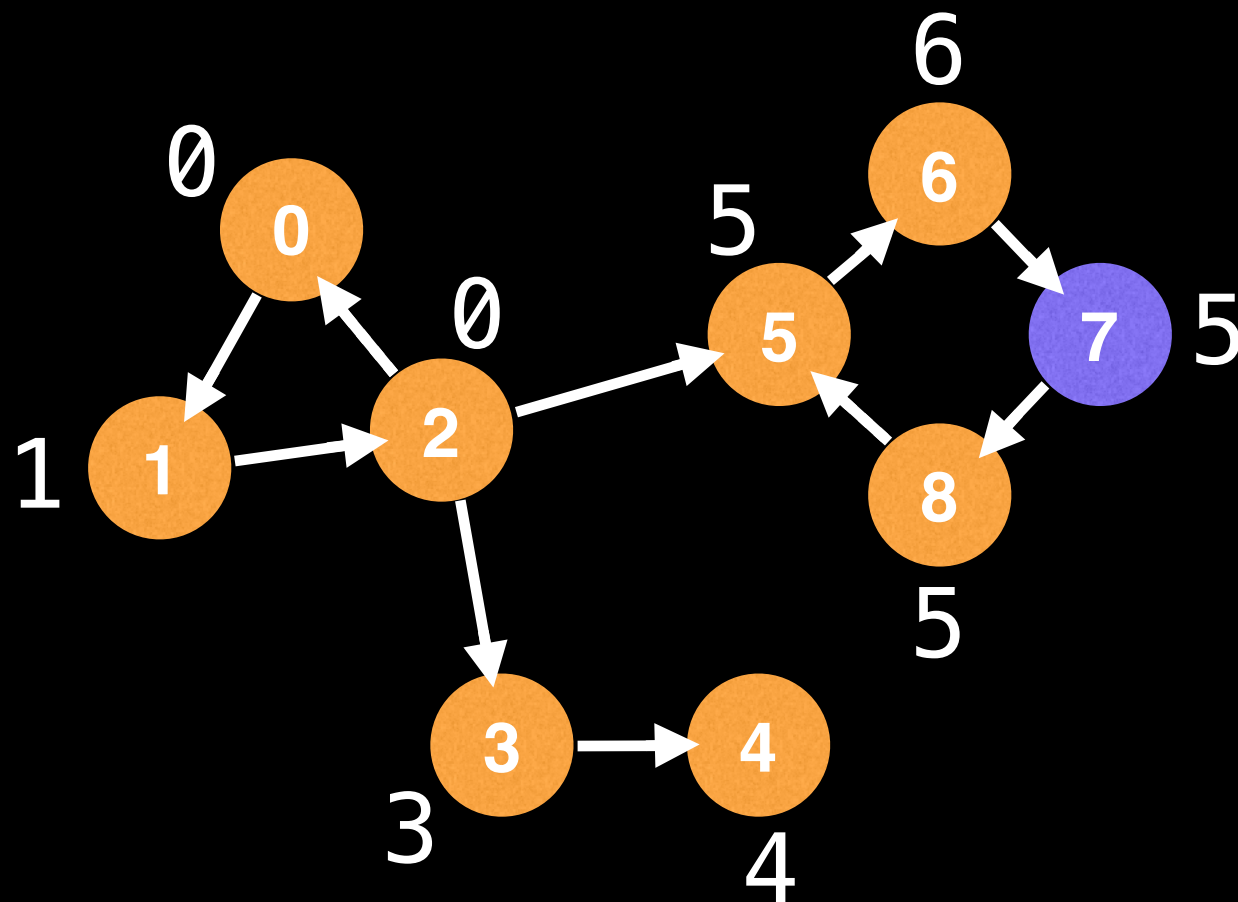

Undirected edge


Directed edge

● Current

● Visited

● Unvisited




—————
Undirected edge

—————→
Directed edge

```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
    visited[at] = true
    id = id + 1
    low[at] = ids[at] = id

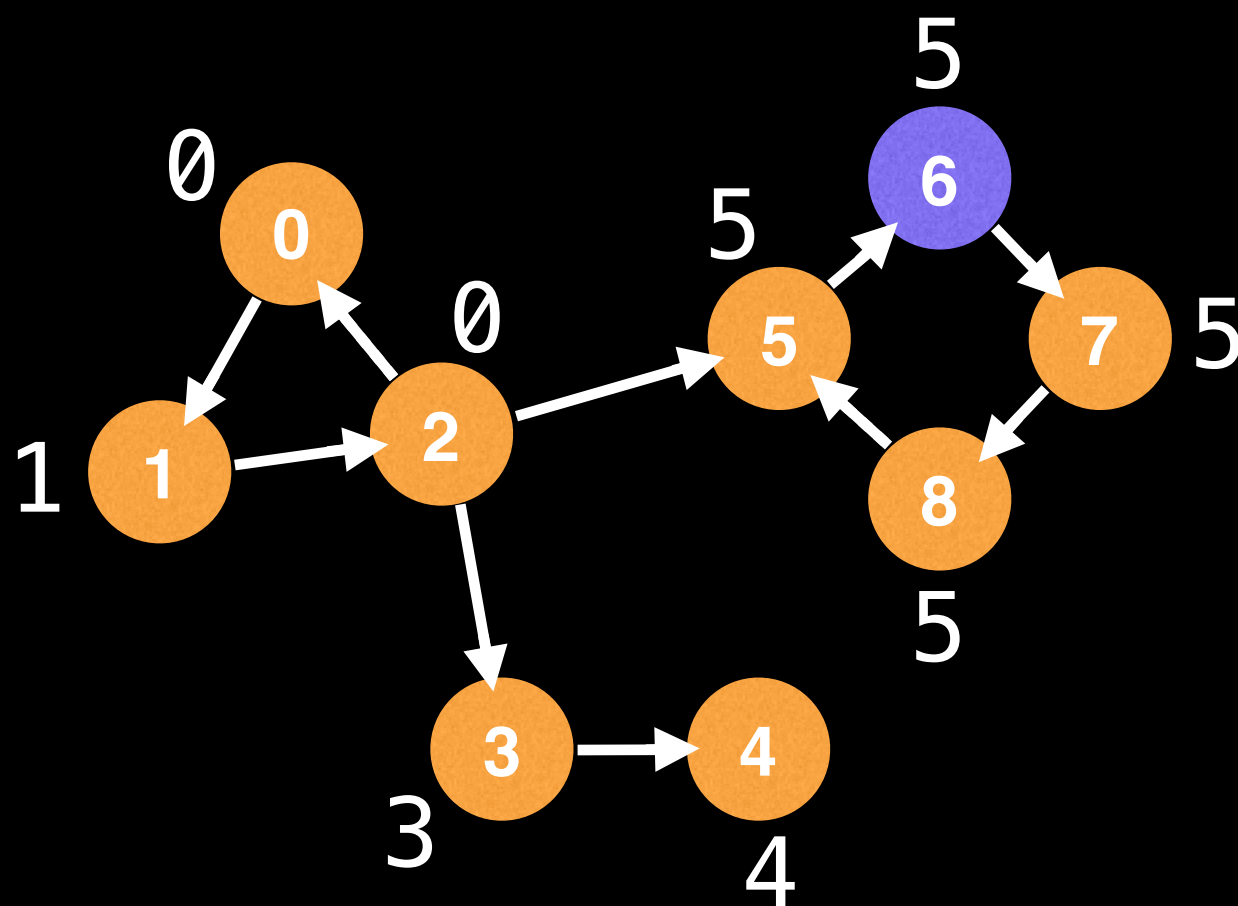
    # For each edge from node 'at' to node 'to'
    for (to : g[at]):
        if to == parent: continue
        if (!visited[to]):
            dfs(to, at, bridges)
            low[at] = min(low[at], low[to])
            if (ids[at] < low[to]):
                bridges.add(at)
                bridges.add(to)
        else:
            low[at] = min(low[at], ids[to])
```



● Current

● Visited

● Unvisited



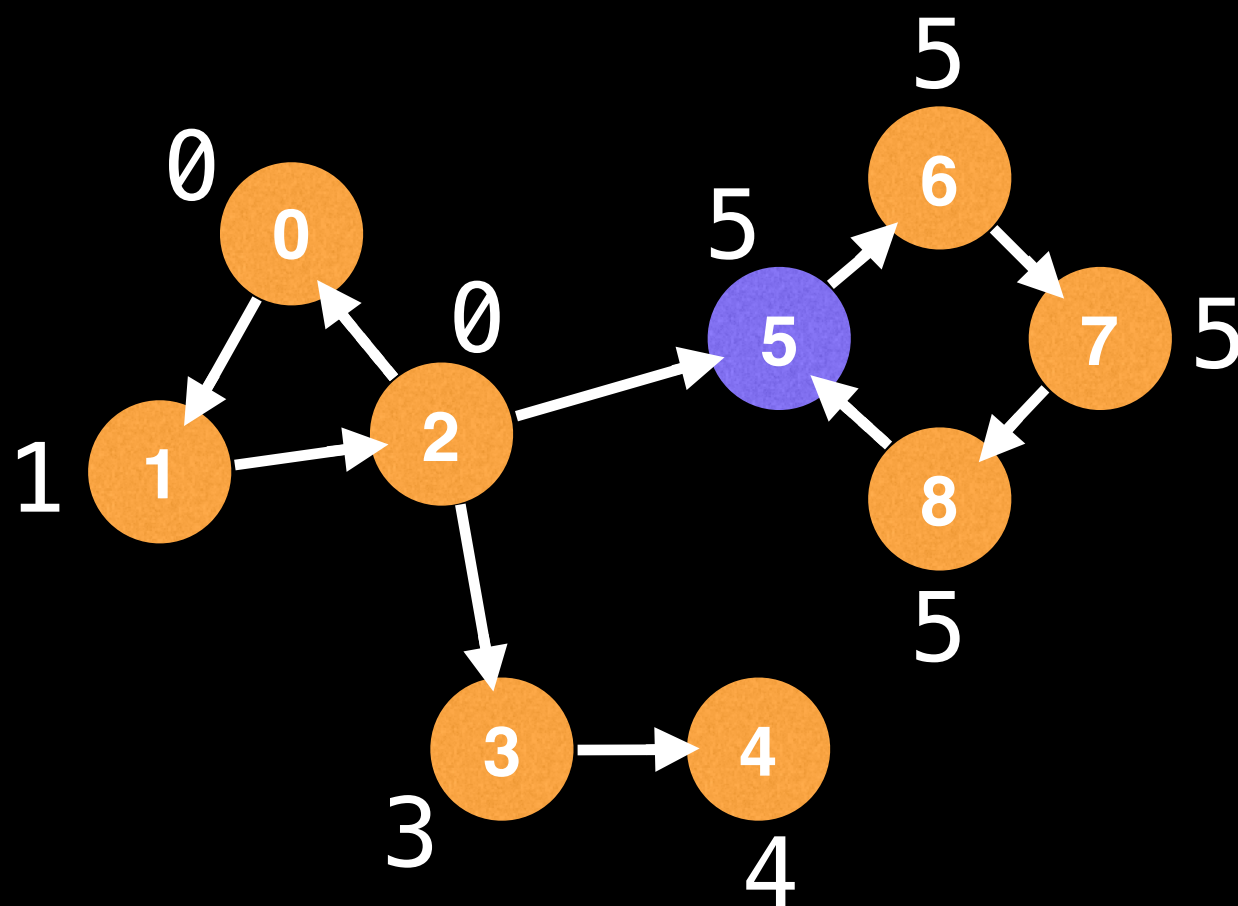
Undirected edge

Directed edge

● Current

● Visited

● Unvisited



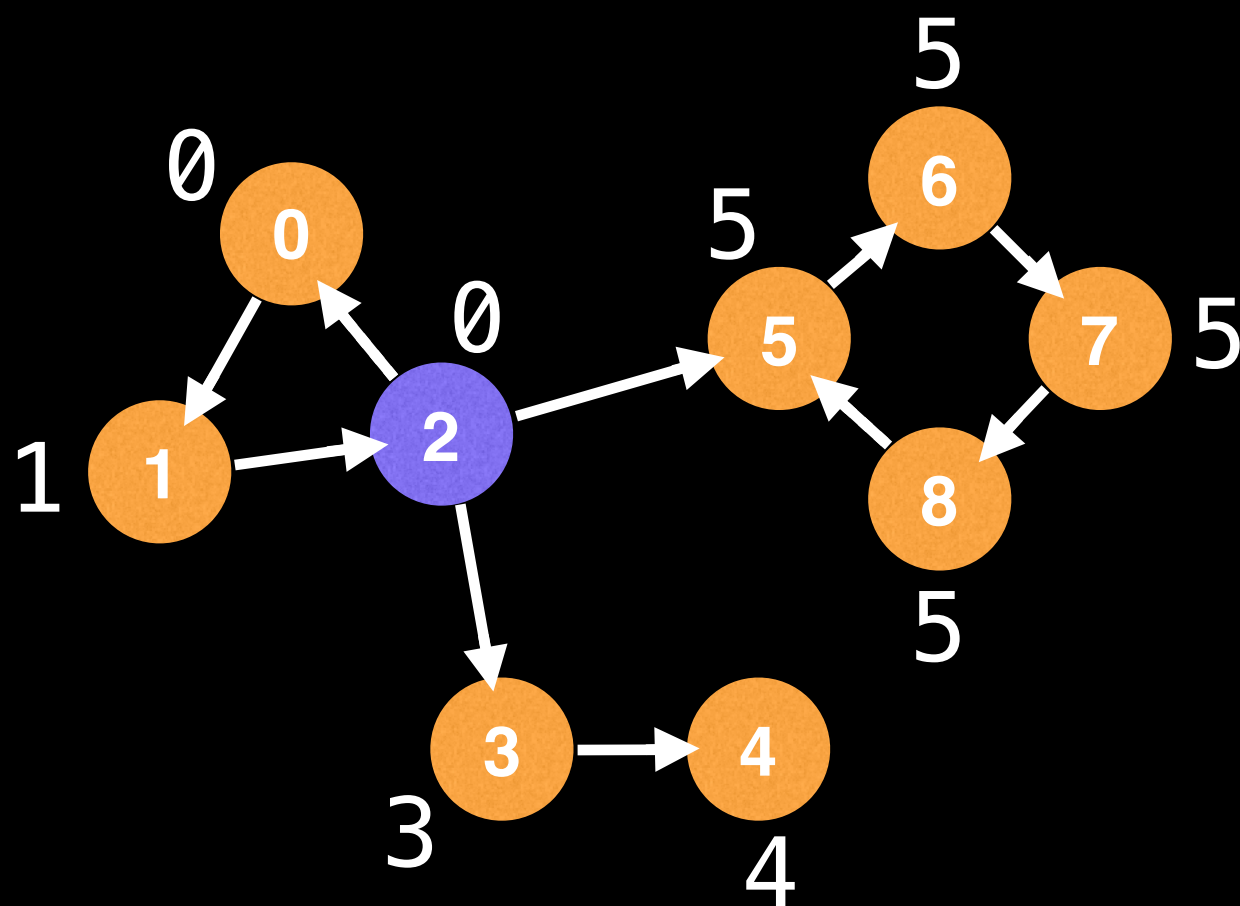
—————
Undirected edge

—————→
Directed edge

● Current

● Visited

● Unvisited



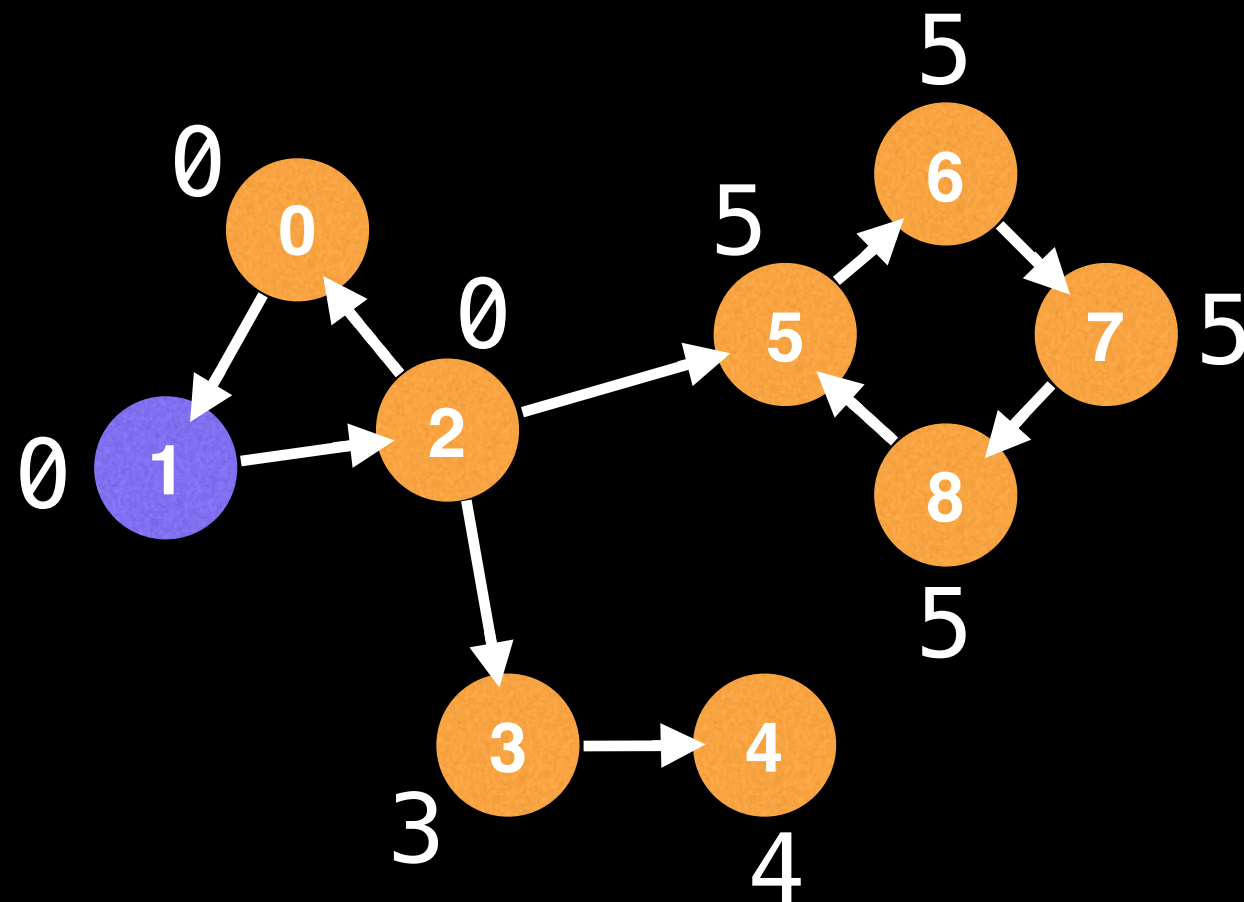
Undirected edge

Directed edge

● Current

● Visited

● Unvisited



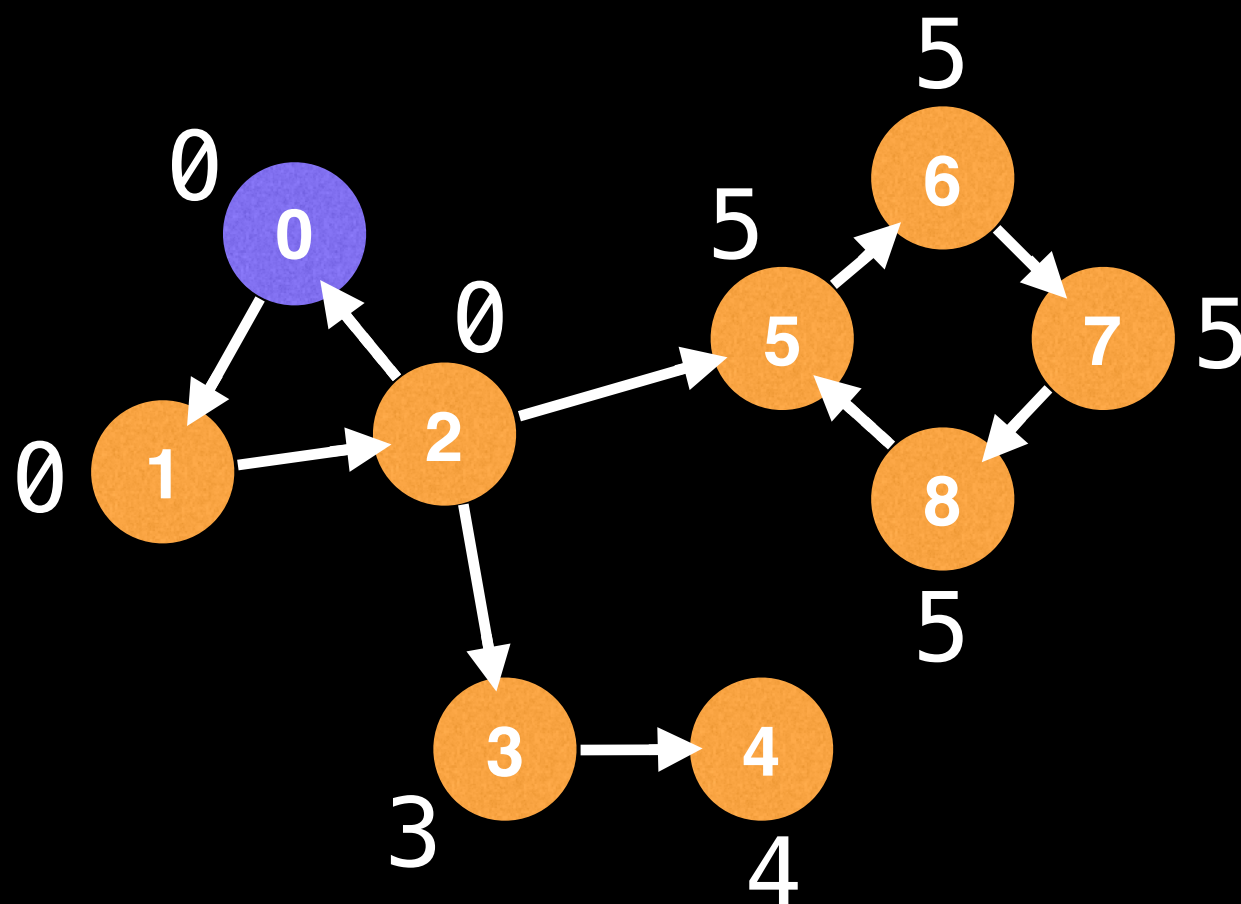
Undirected edge

Directed edge

● Current

● Visited

● Unvisited



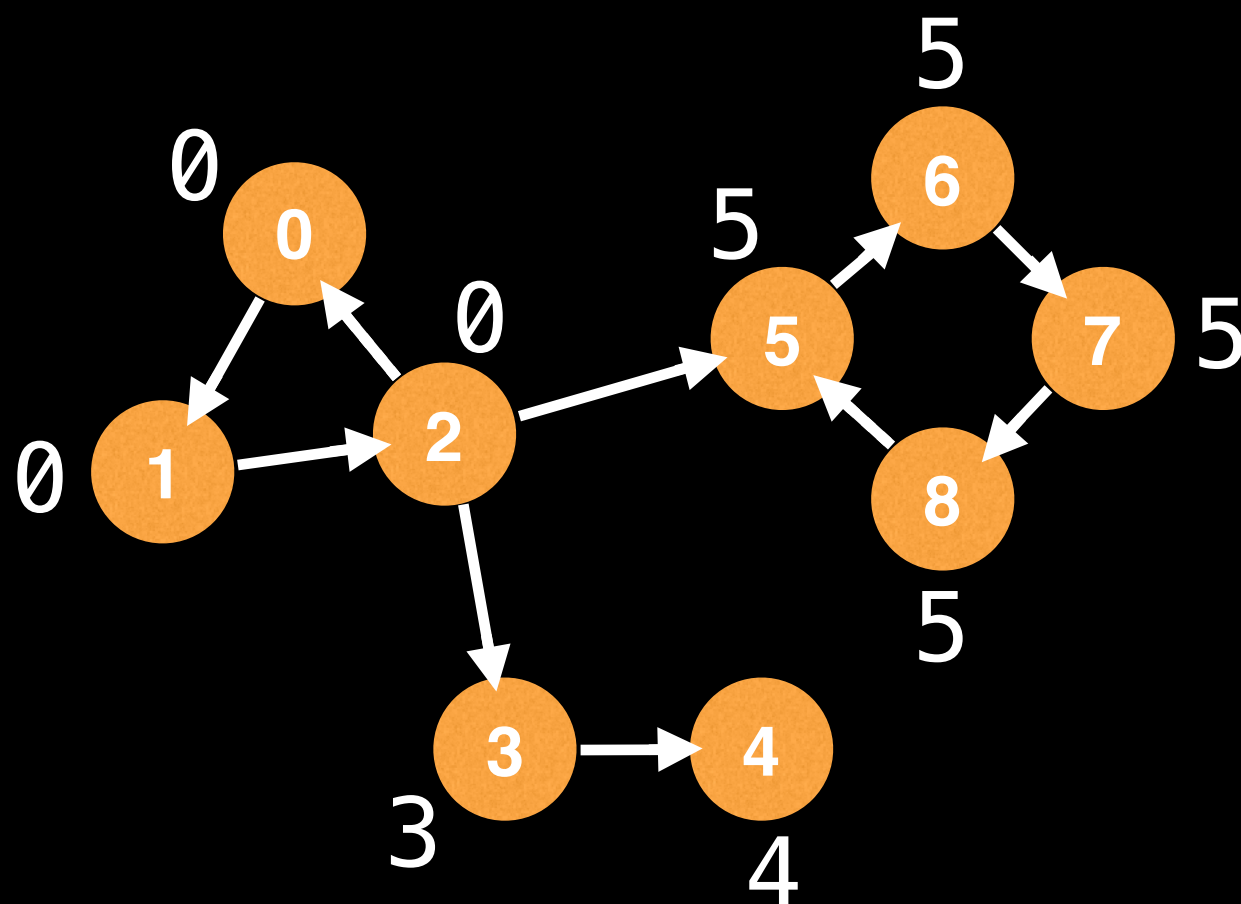
Undirected edge

Directed edge

● Current

● Visited

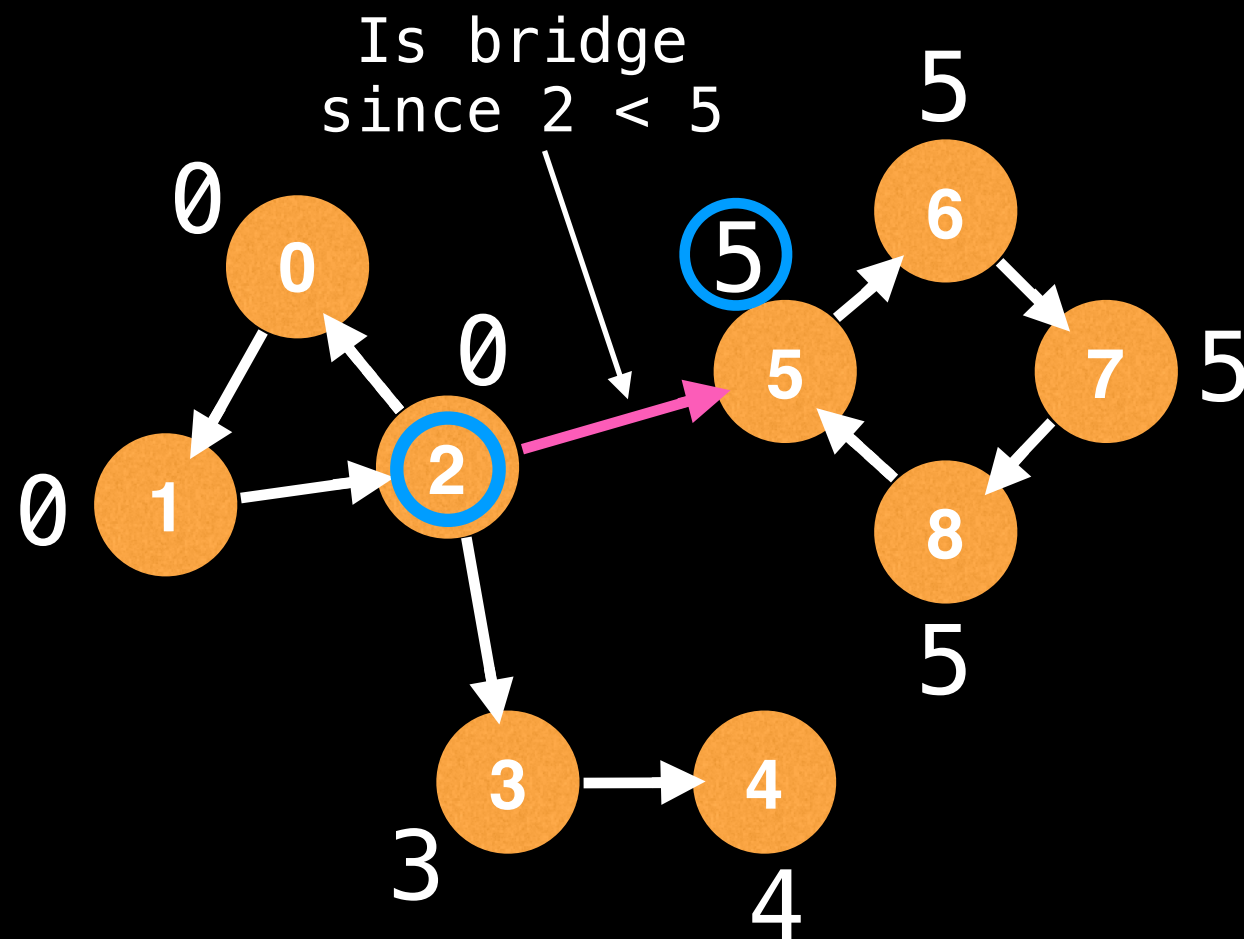
● Unvisited



Undirected edge

Directed edge

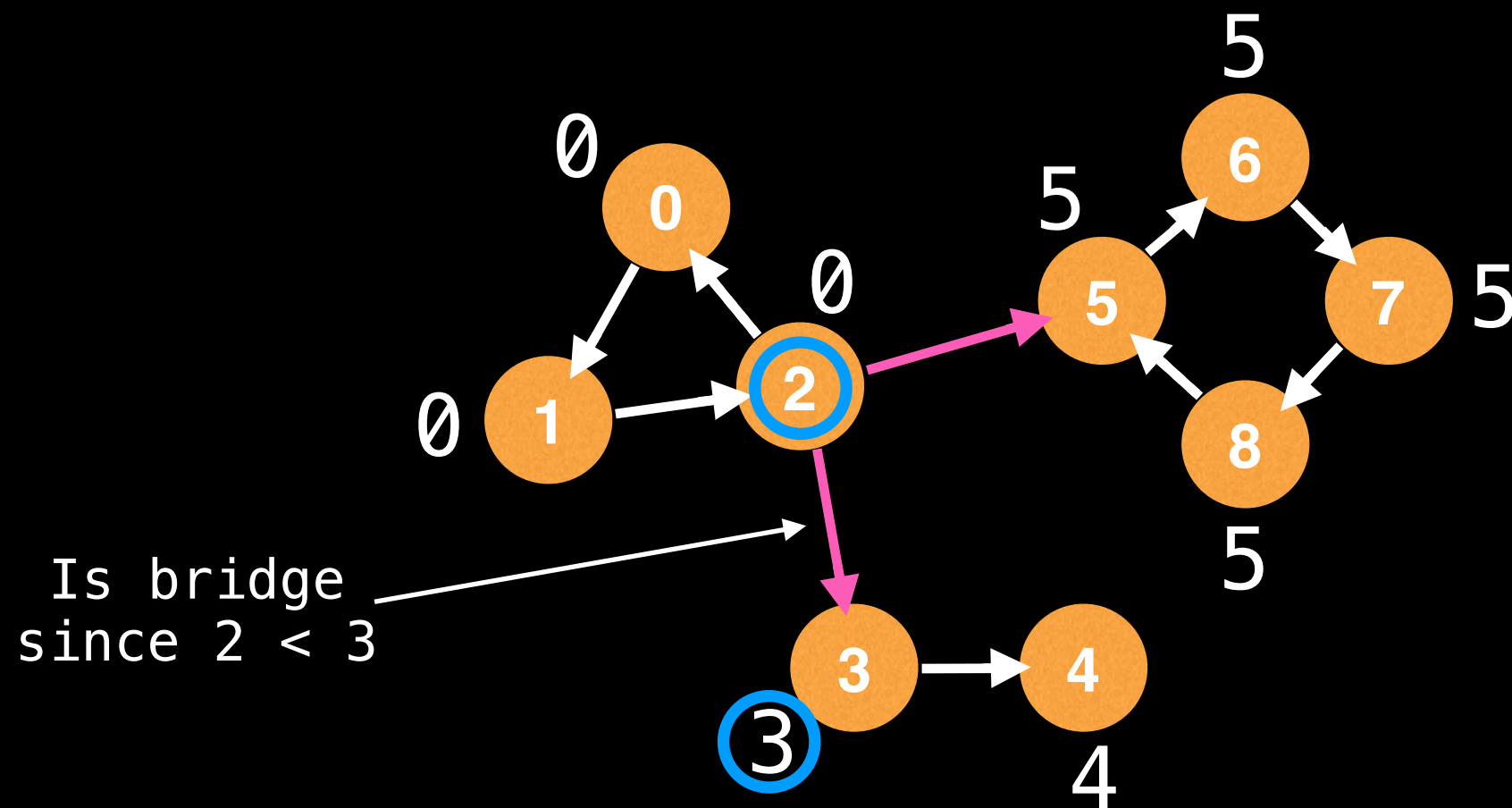
The condition for a directed edge 'e' to have nodes that belong to a bridge is when the
 $\text{id}(\text{e.from}) < \text{lowlink}(\text{e.to})$



—————
Undirected edge

—————→
Directed edge

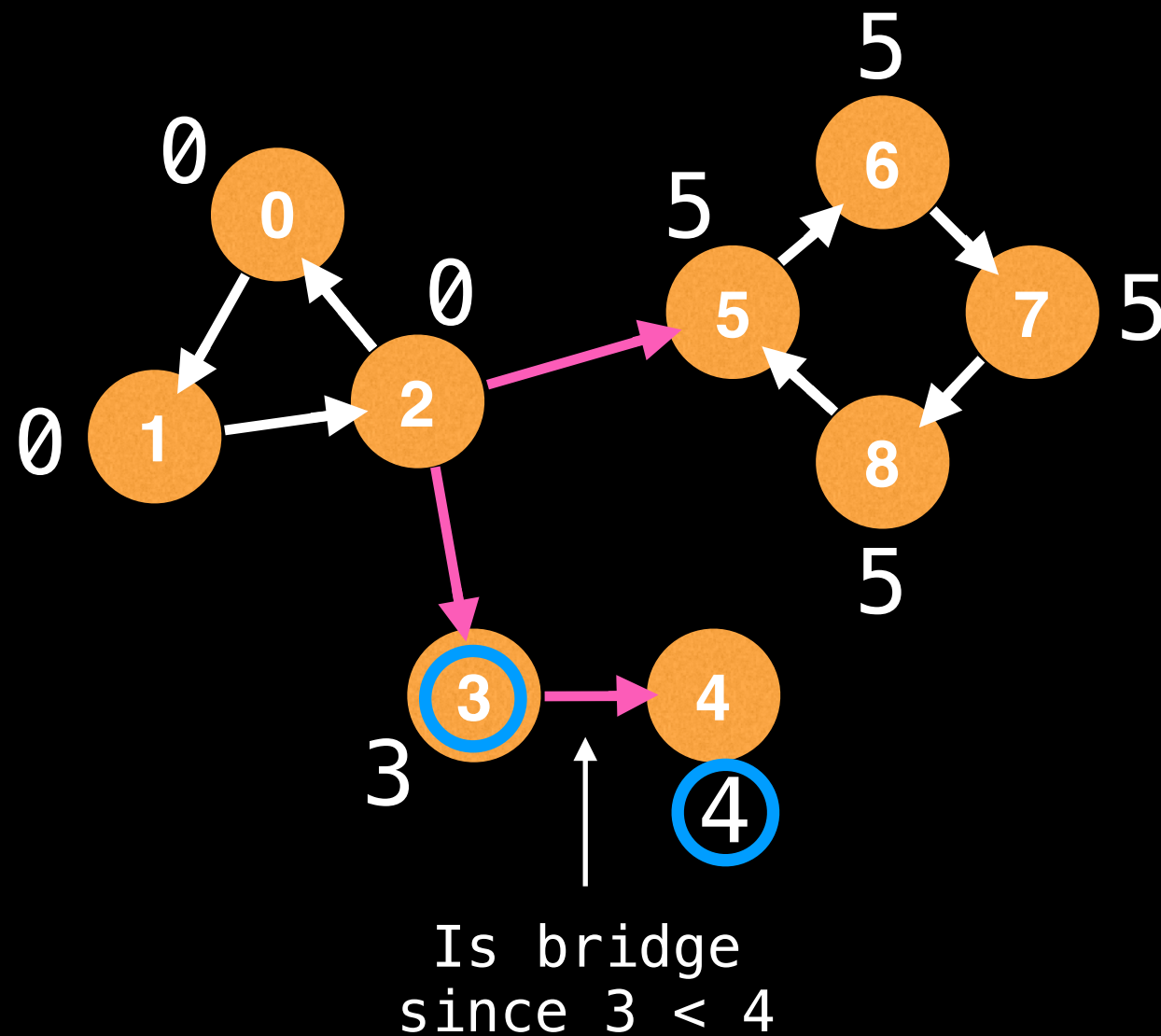
The condition for a directed edge 'e' to have nodes that belong to a bridge is when the
 $\text{id}(\text{e.from}) < \text{lowlink}(\text{e.to})$



—————
Undirected edge

—————→
Directed edge

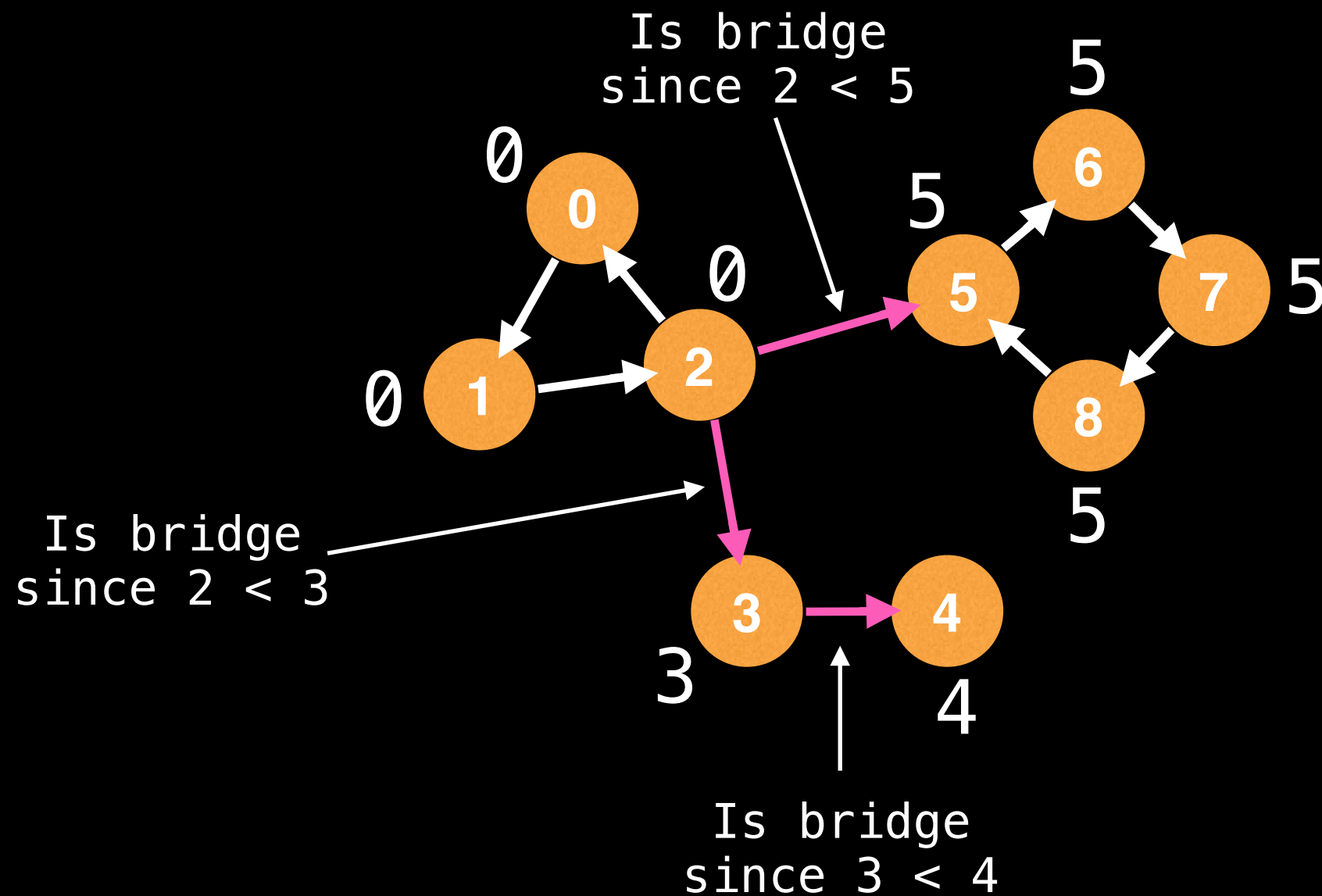
The condition for a directed edge 'e' to have nodes that belong to a bridge is when the
 $\text{id}(\text{e.from}) < \text{lowlink}(\text{e.to})$



—————
Undirected edge

—————→
Directed edge

The condition for a directed edge 'e' to have nodes that belong to a bridge is when the
 $\text{id}(\text{e.from}) < \text{lowlink}(\text{e.to})$



—————
Undirected edge

—————→
Directed edge

Articulation points

Articulation points are related very closely to bridges. It won't take much modification to the finding bridges algorithm to find articulation points.

Articulation points

Simple observation about articulation points:

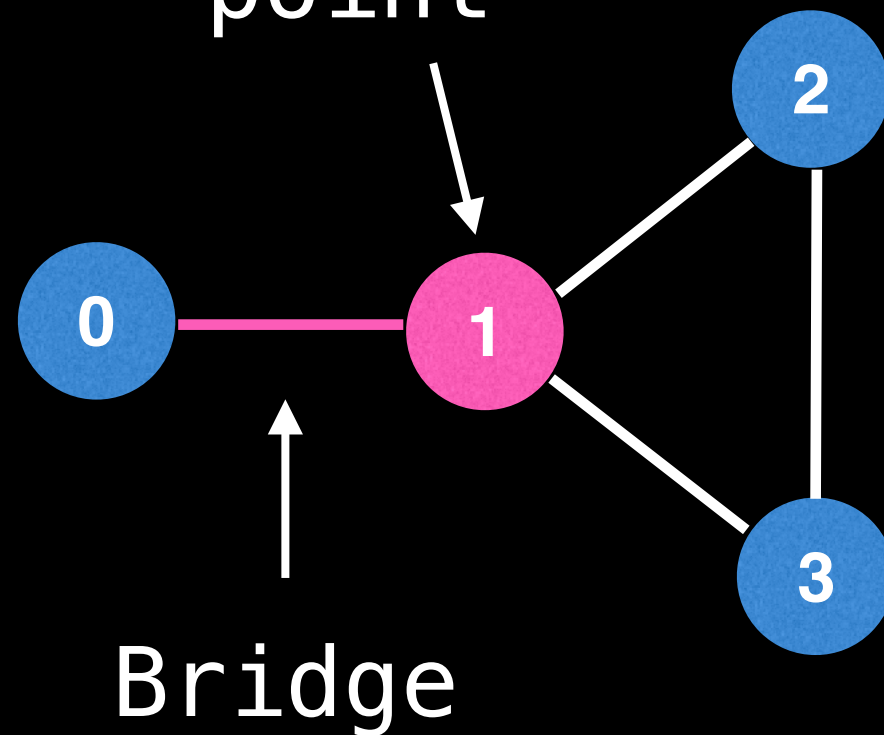
On a connected component with three or more vertices if an edge (u, v) is a bridge then either u or v is an articulation point.

Articulation points

Simple observation about articulation points:

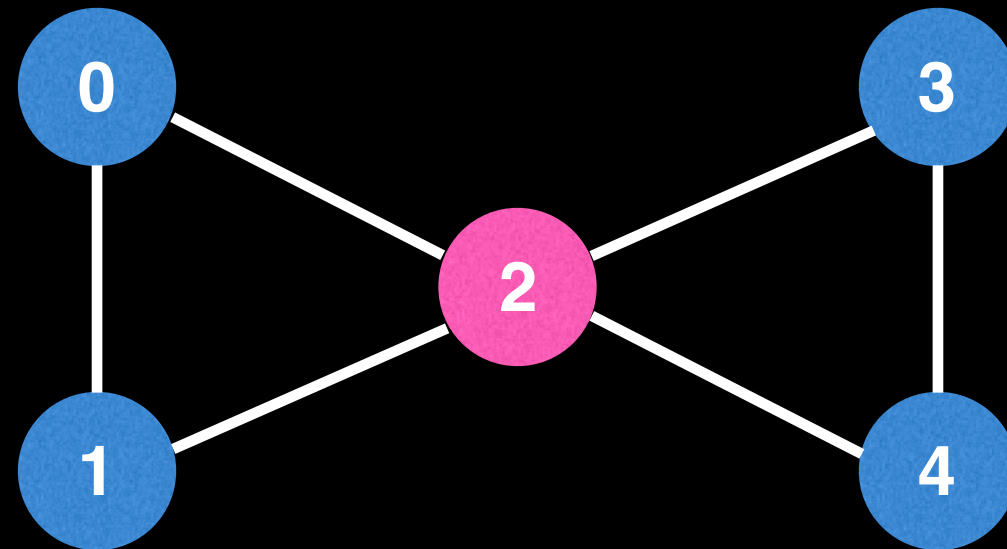
On a connected component with three or more vertices if an edge (u, v) is a bridge then either u or v is an articulation point.

Articulation
point



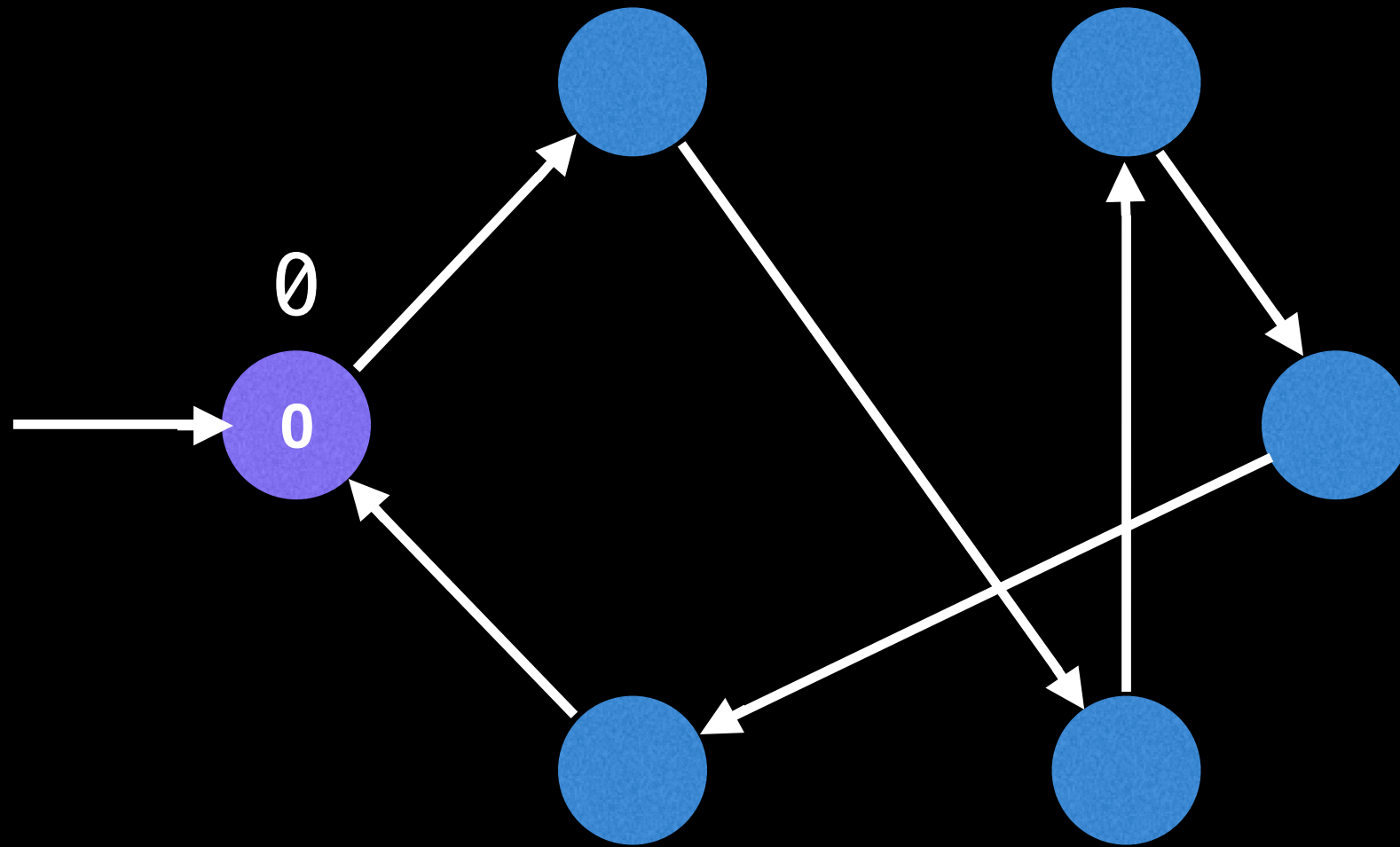
Articulation points

However, this condition alone is not sufficient to capture all articulation points. There exist cases where there is an articulation point without a bridge:

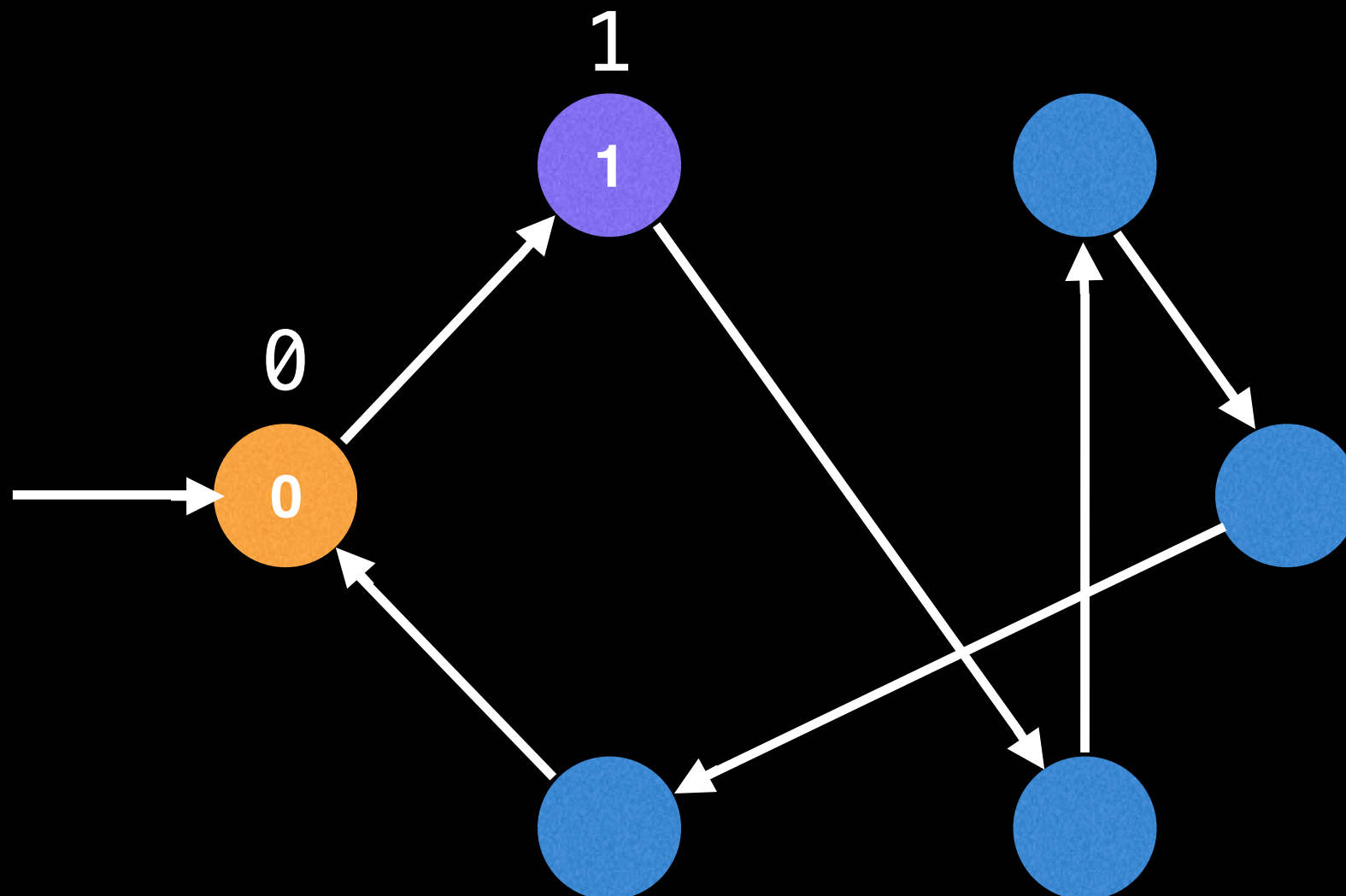


There are no bridges but node 2 is an articulation point since its removal would cause the graph to split into two components.

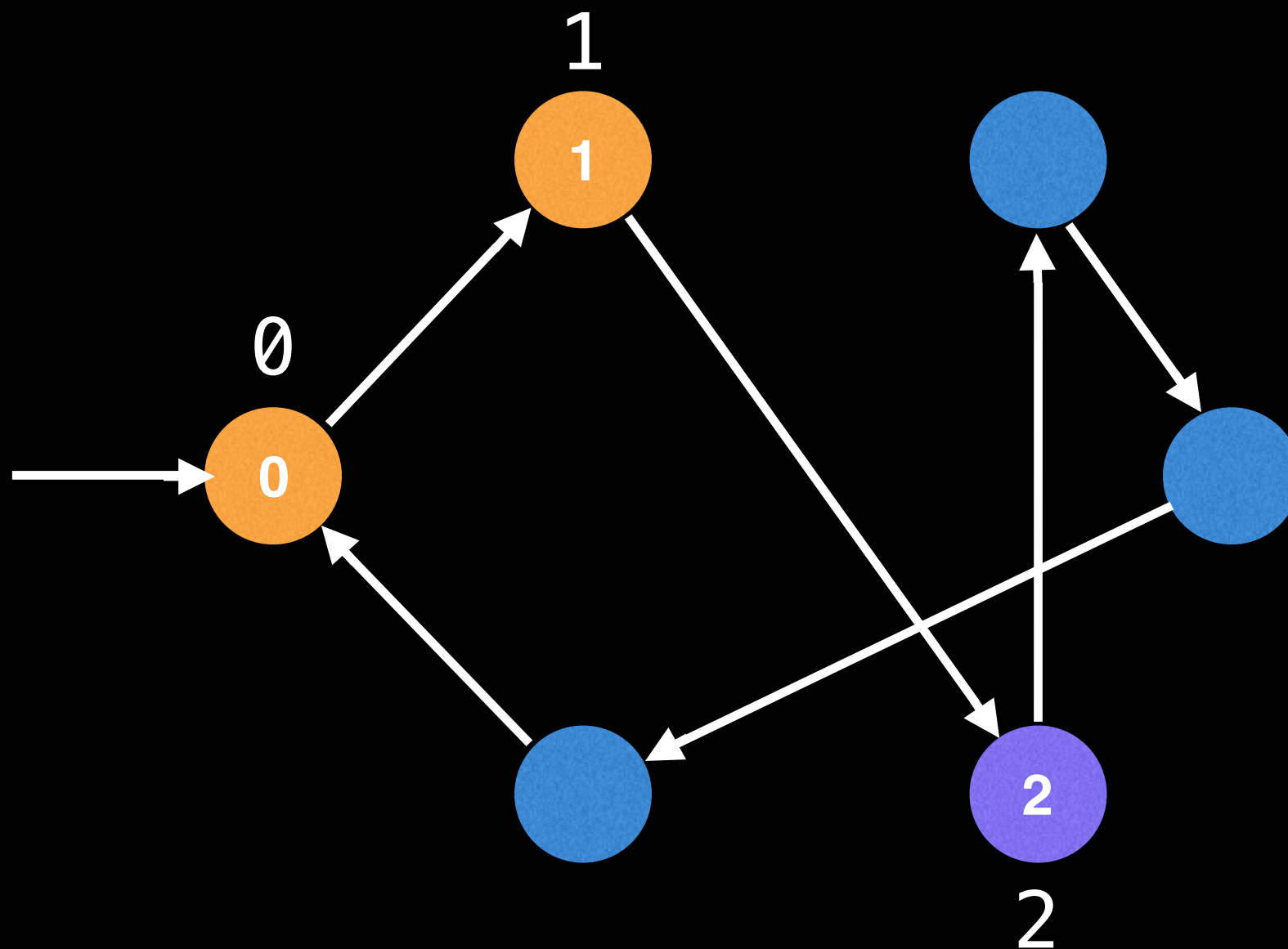
Articulation points



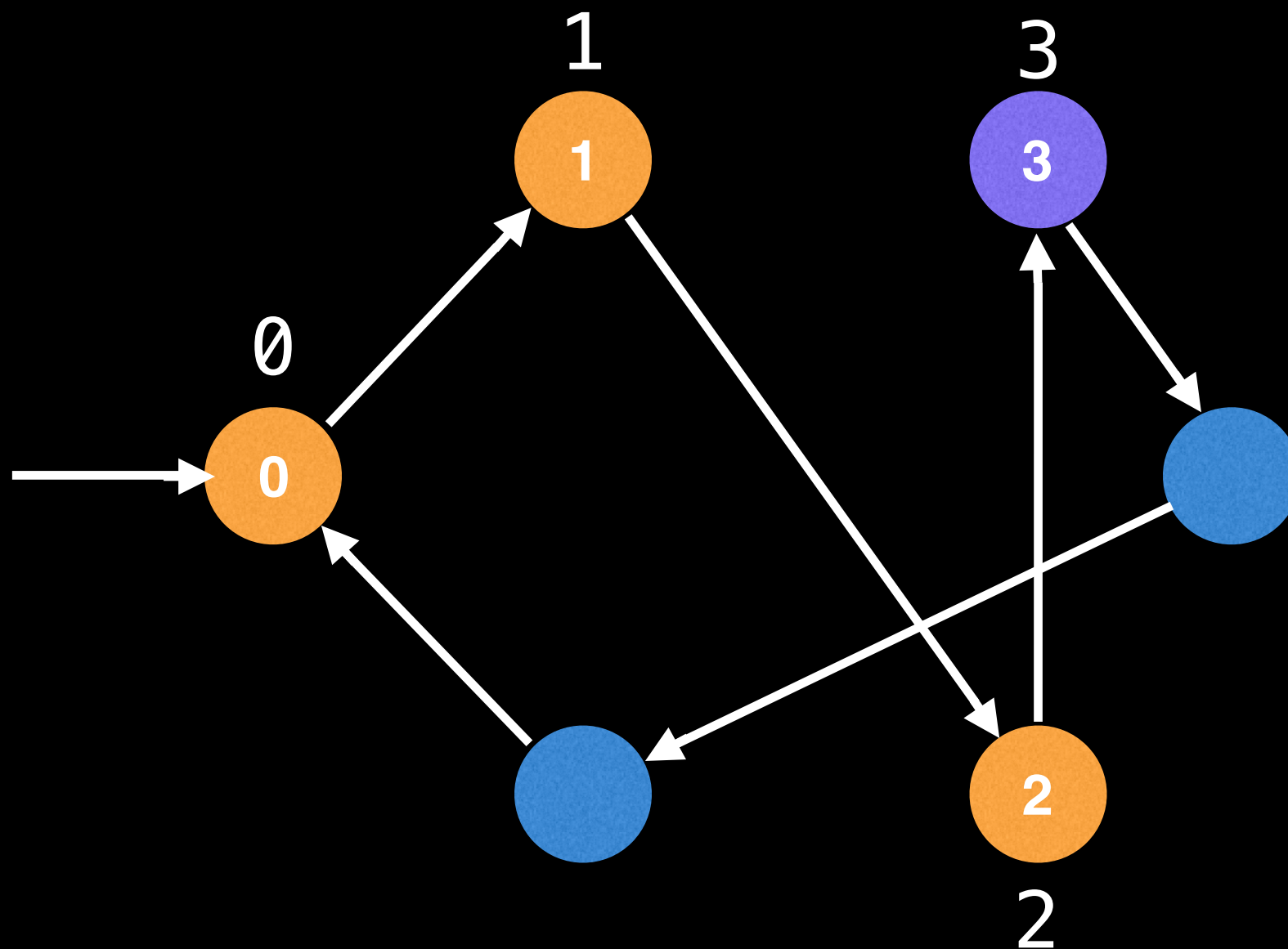
Articulation points



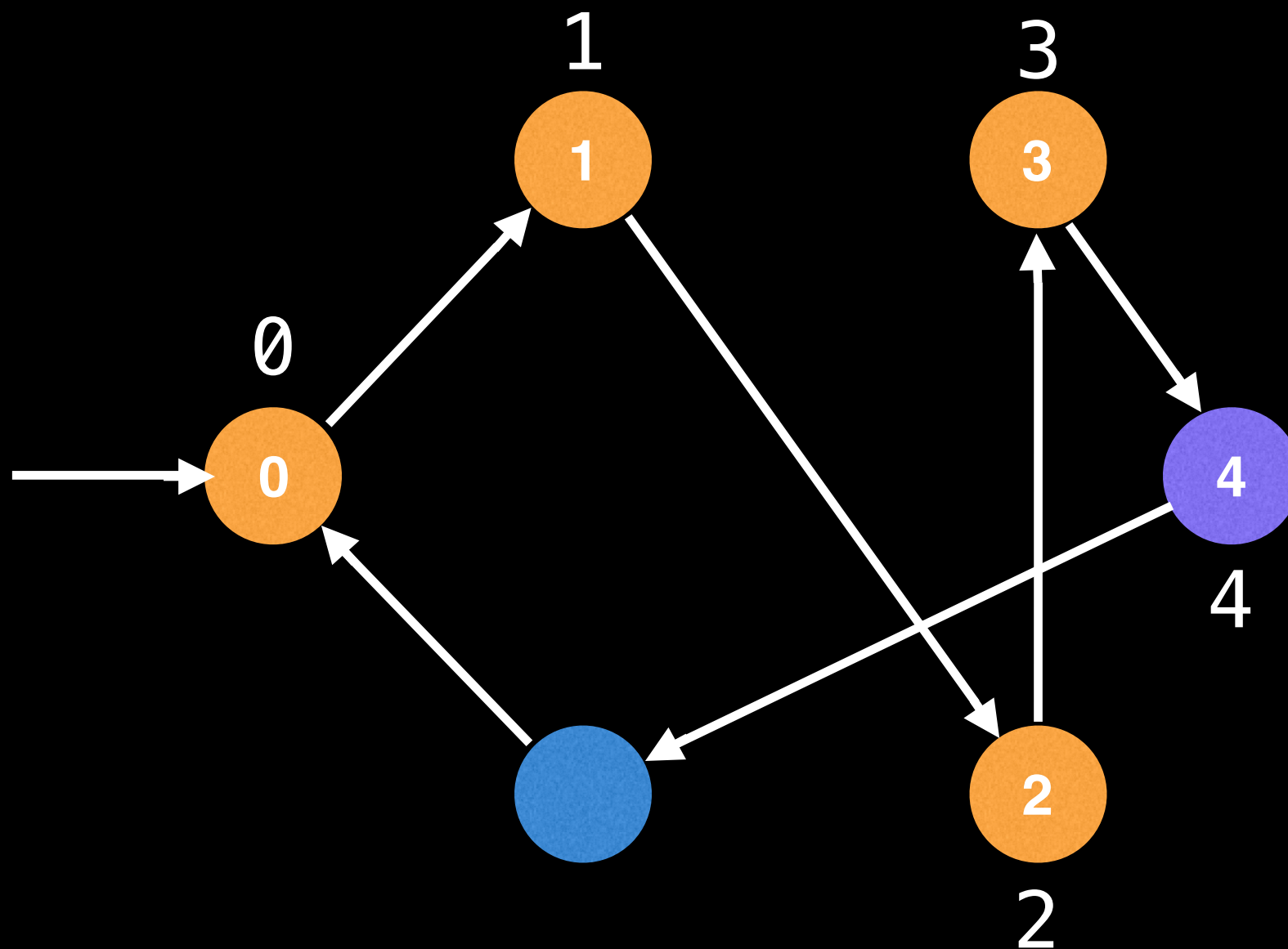
Articulation points



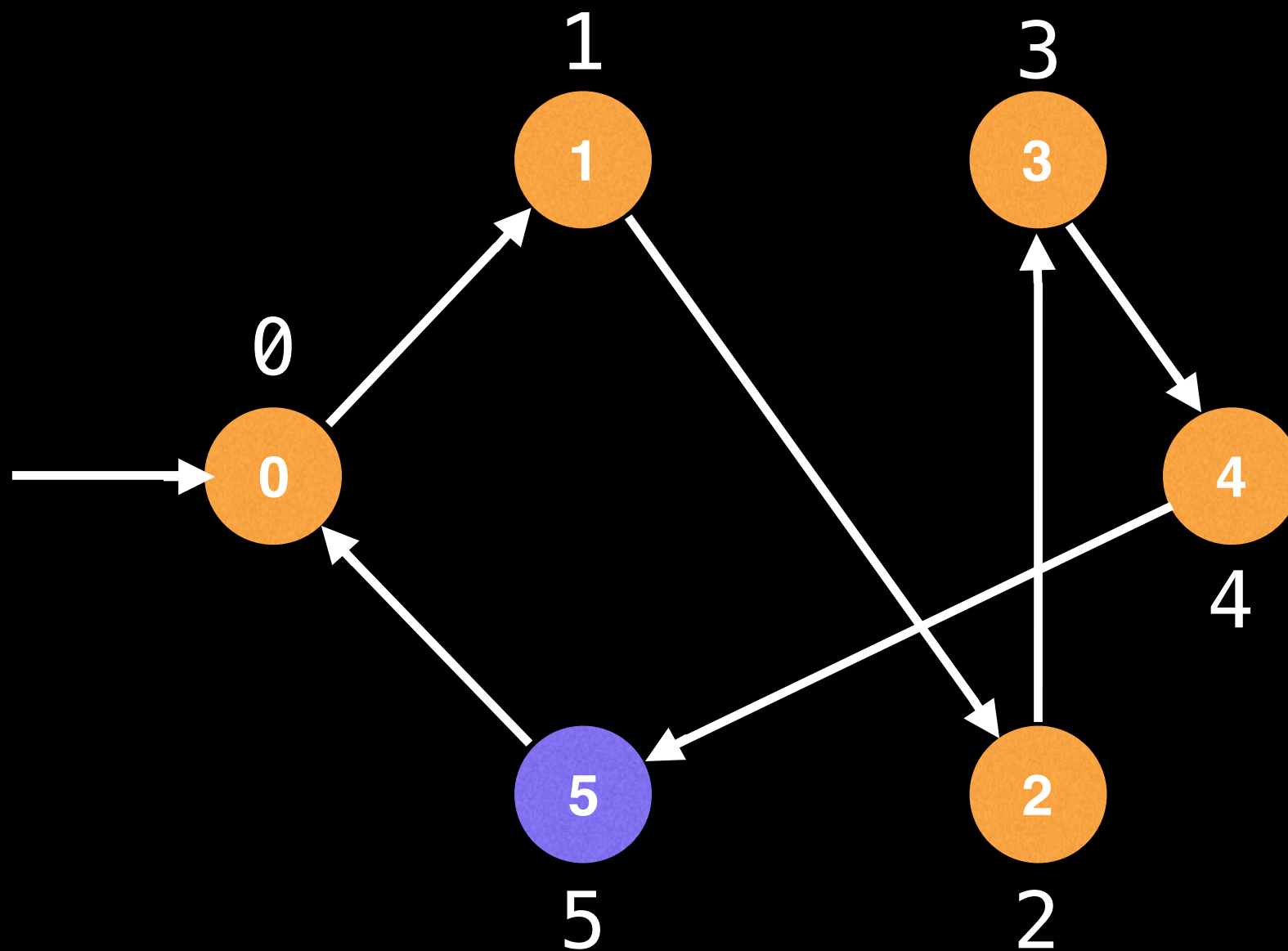
Articulation points



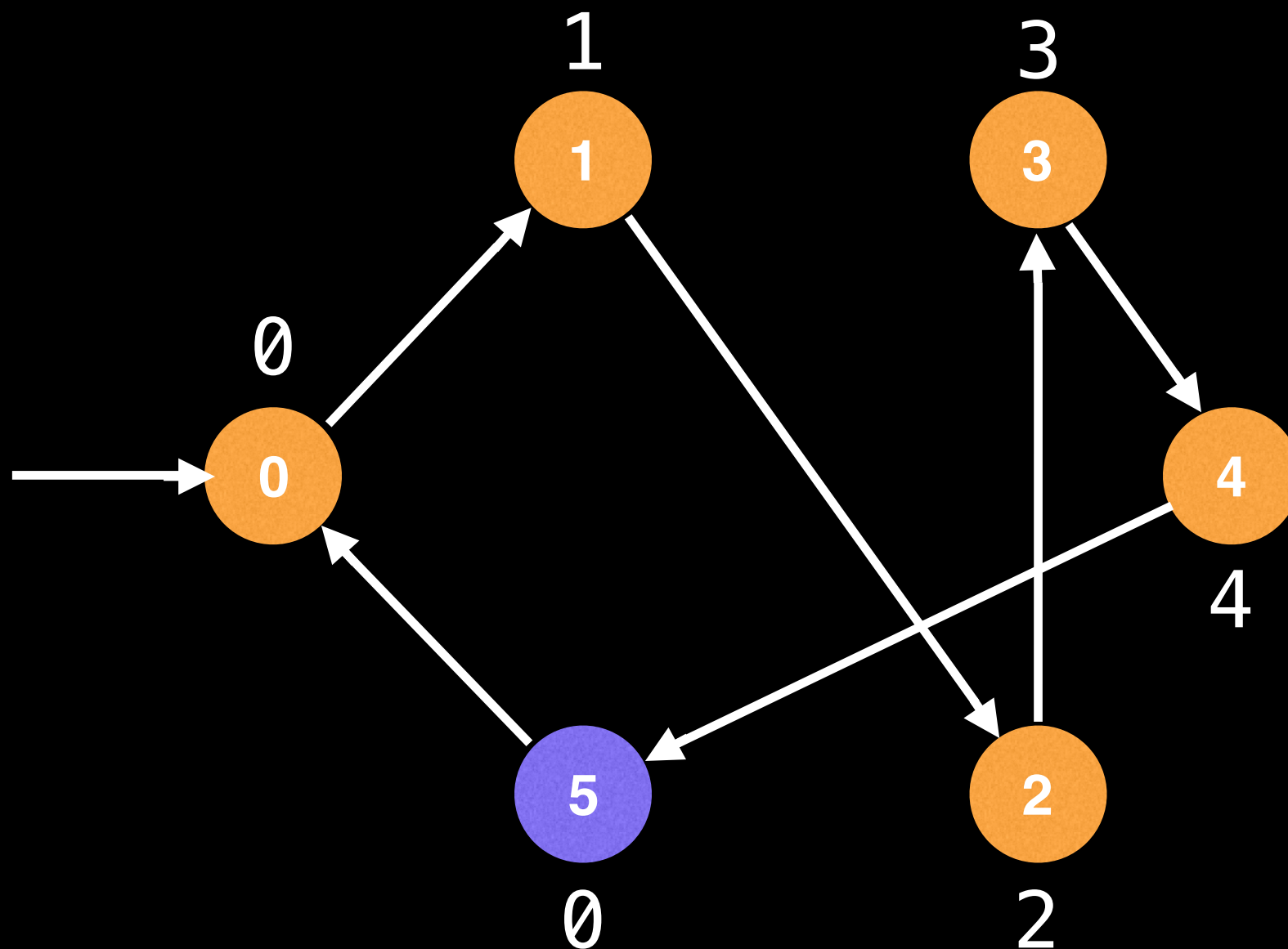
Articulation points



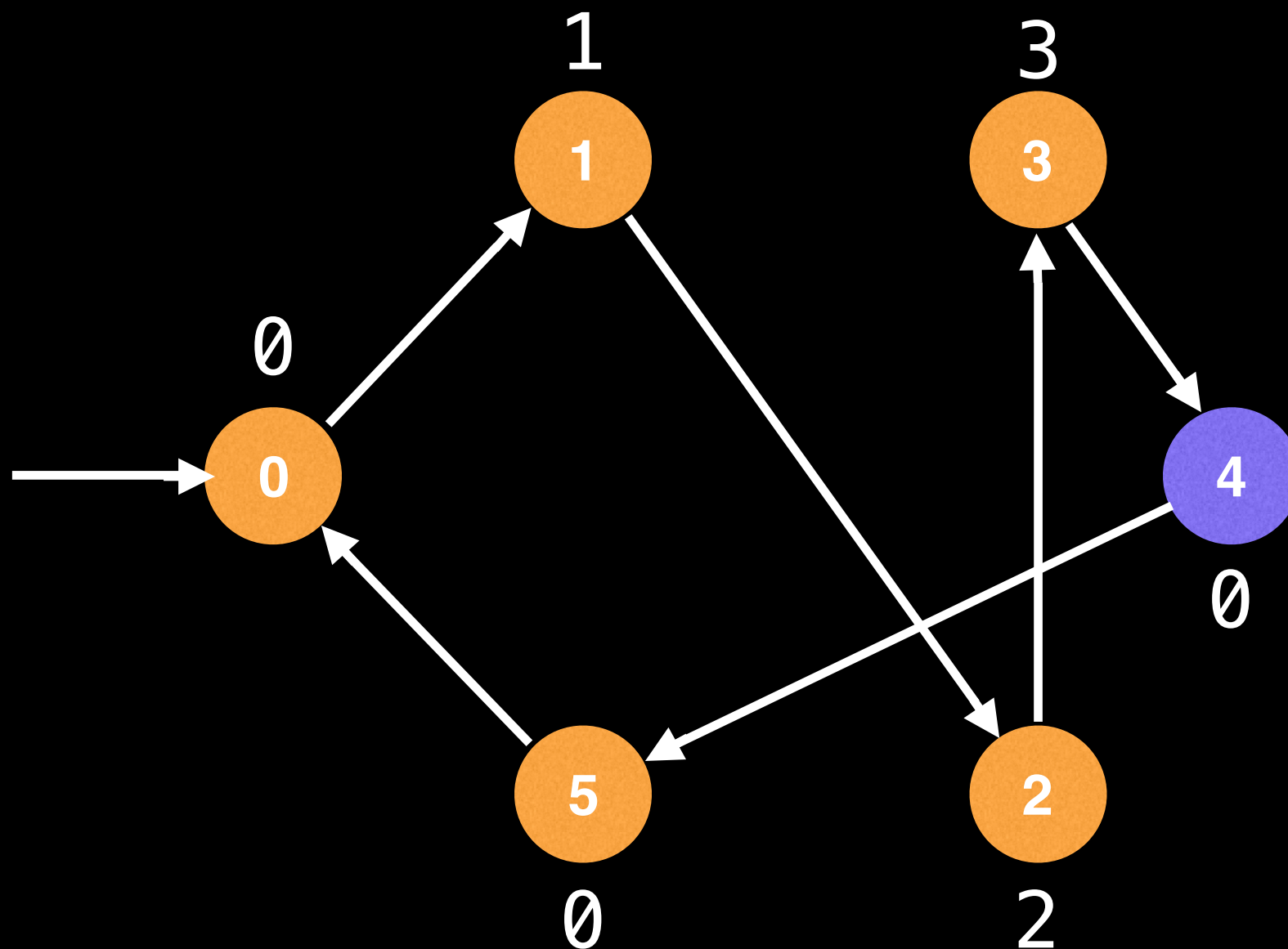
Articulation points



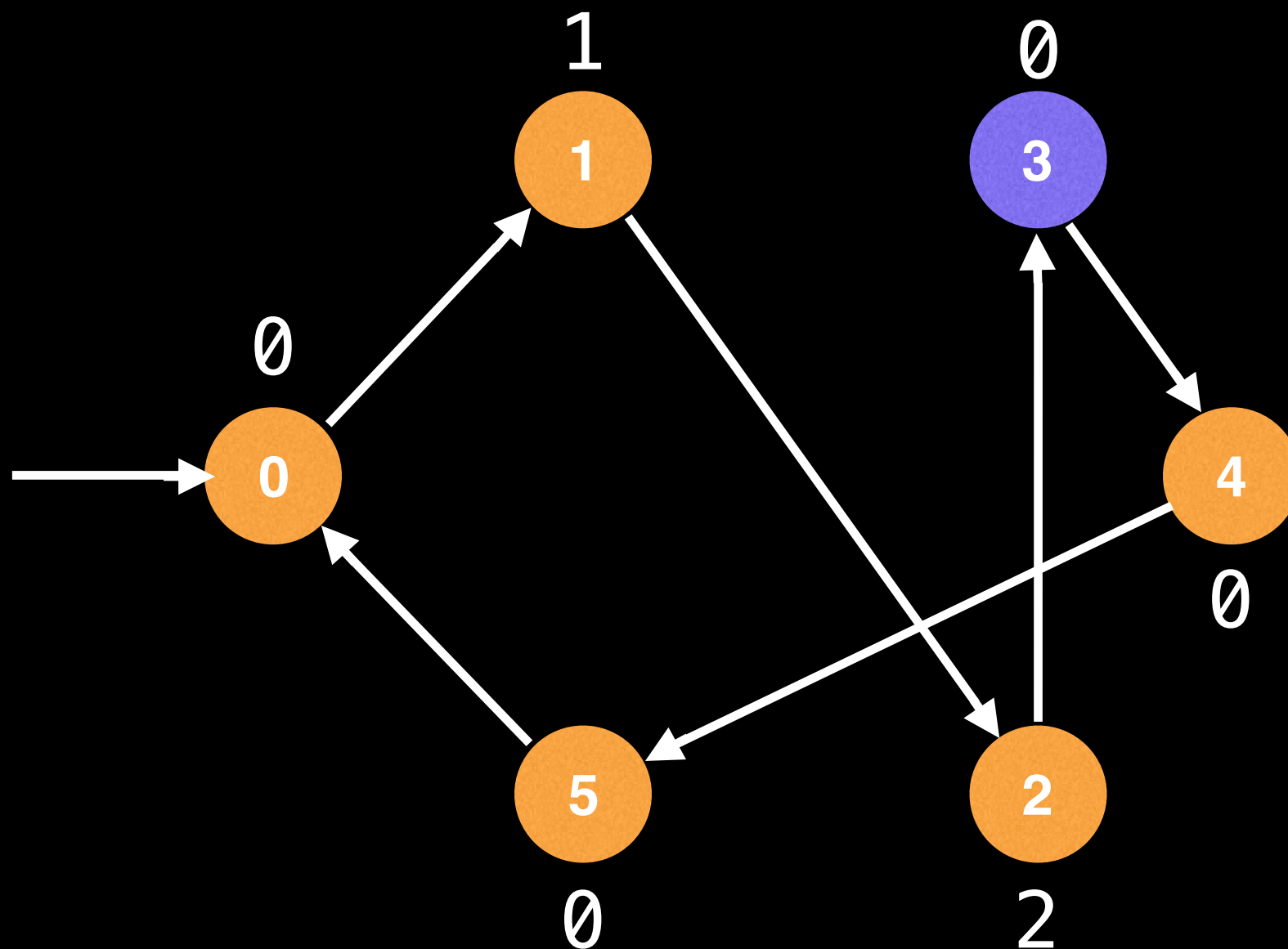
Articulation points



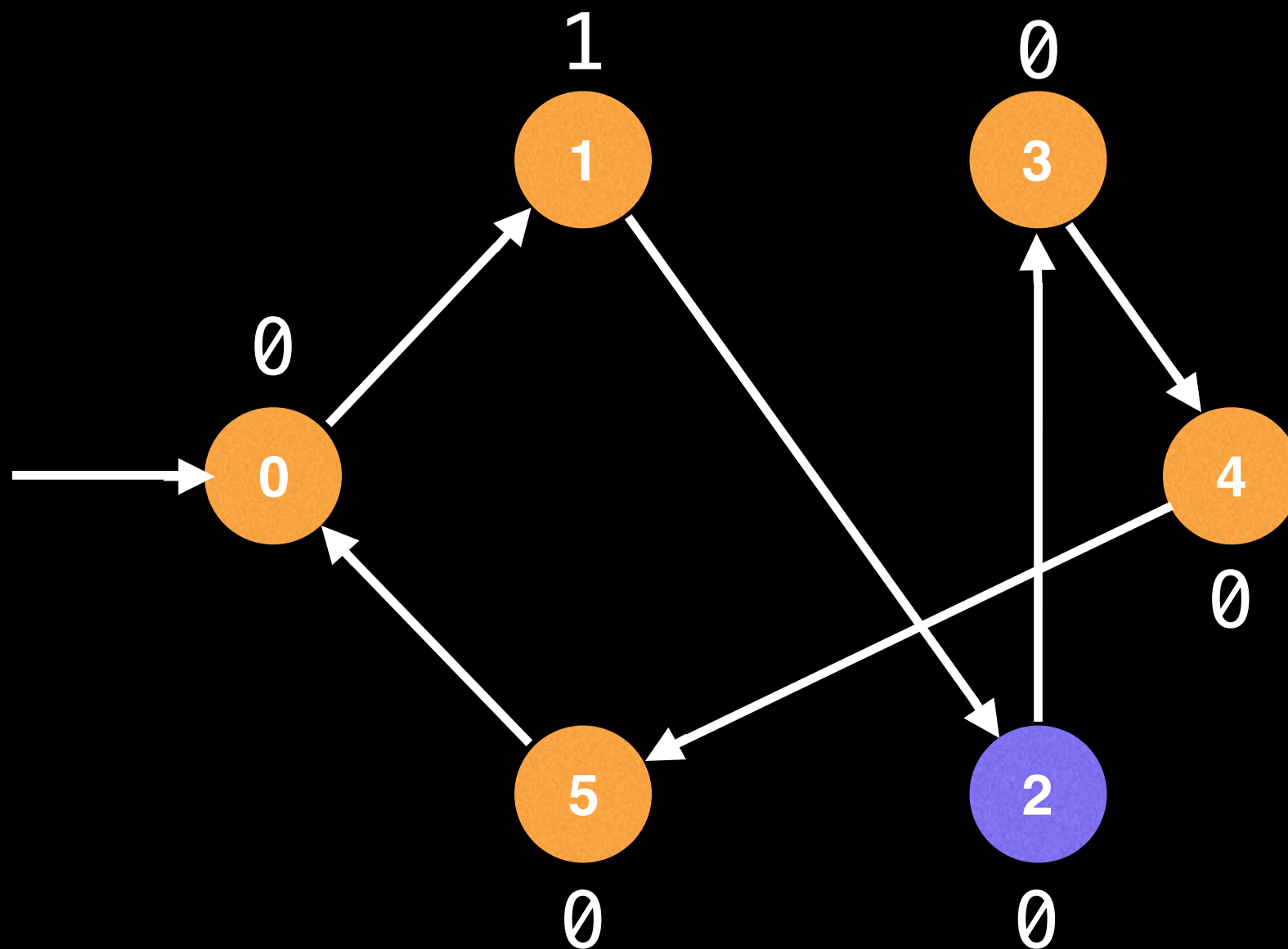
Articulation points



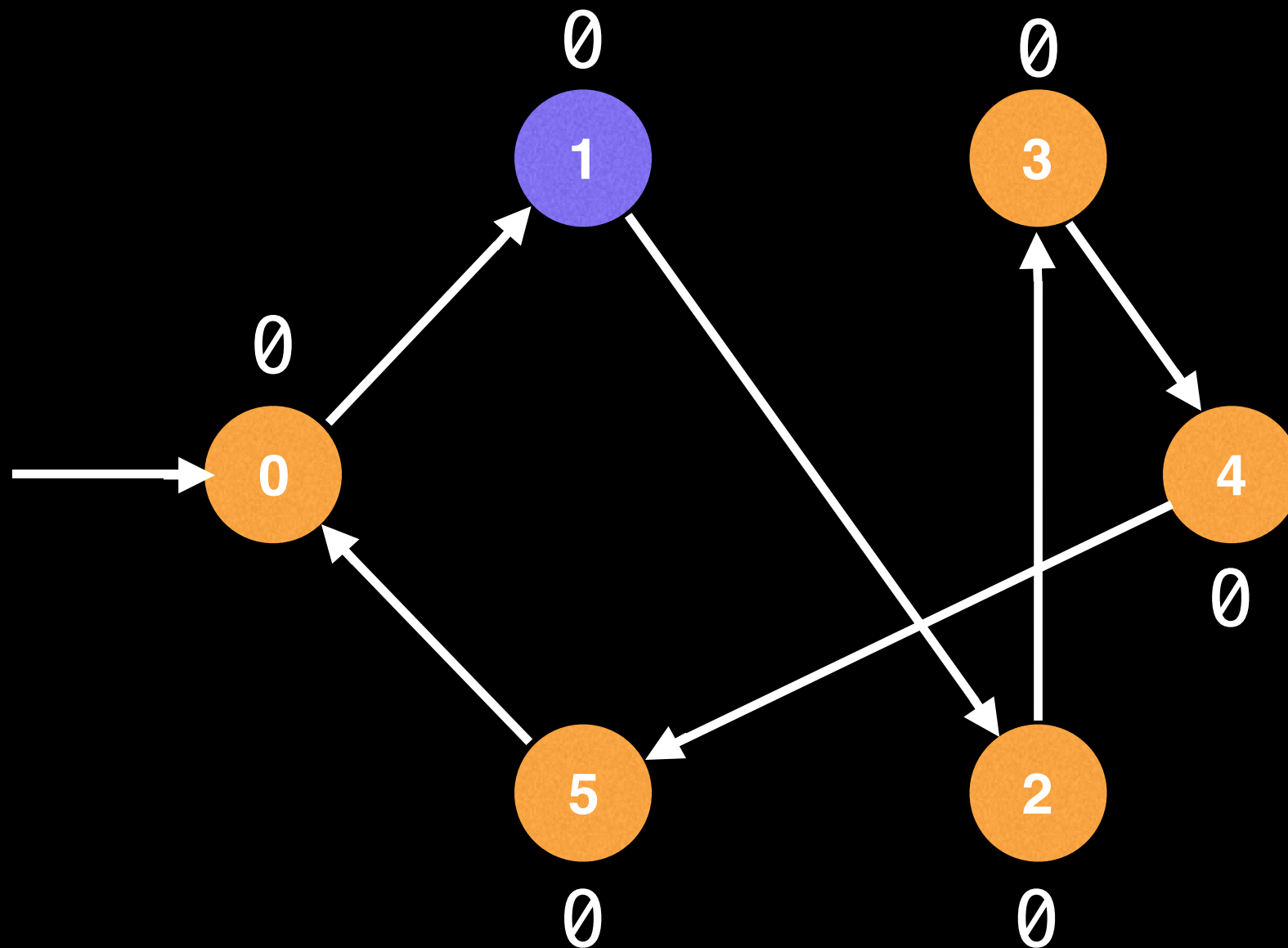
Articulation points



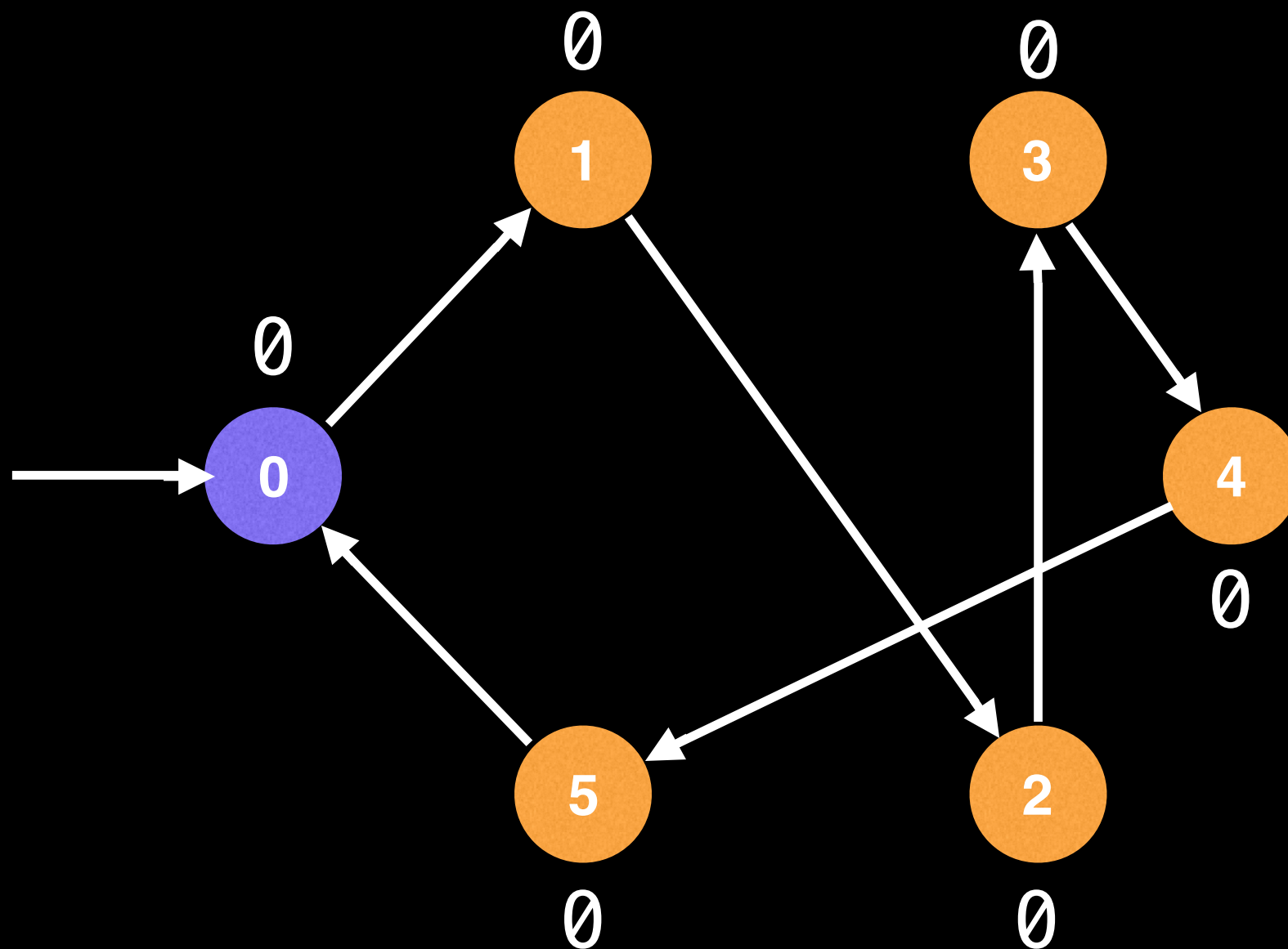
Articulation points



Articulation points

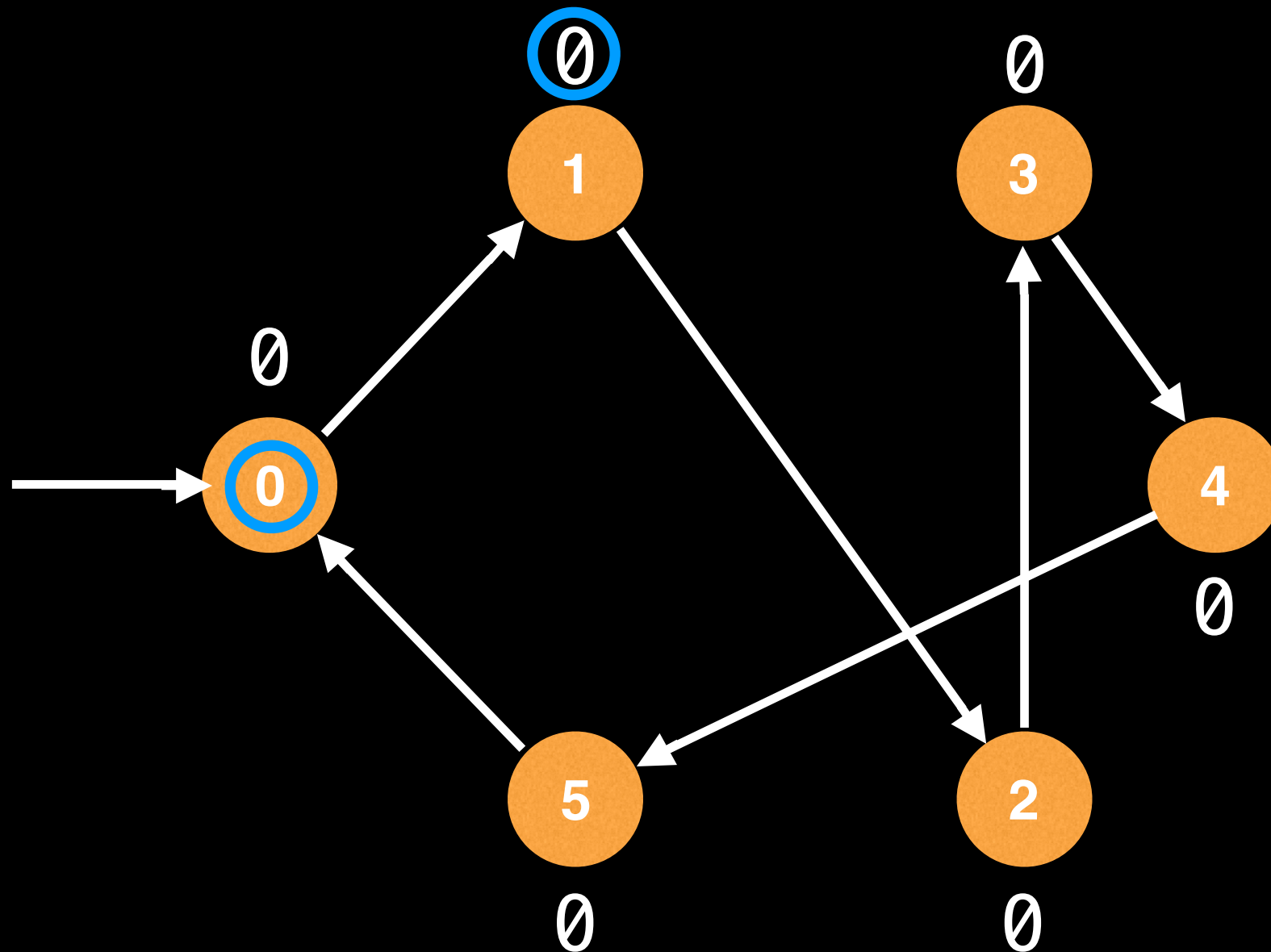


Articulation points



Articulation points

On the callback, if `id(e.from) == lowlink(e.to)`
then there was a **cycle**.

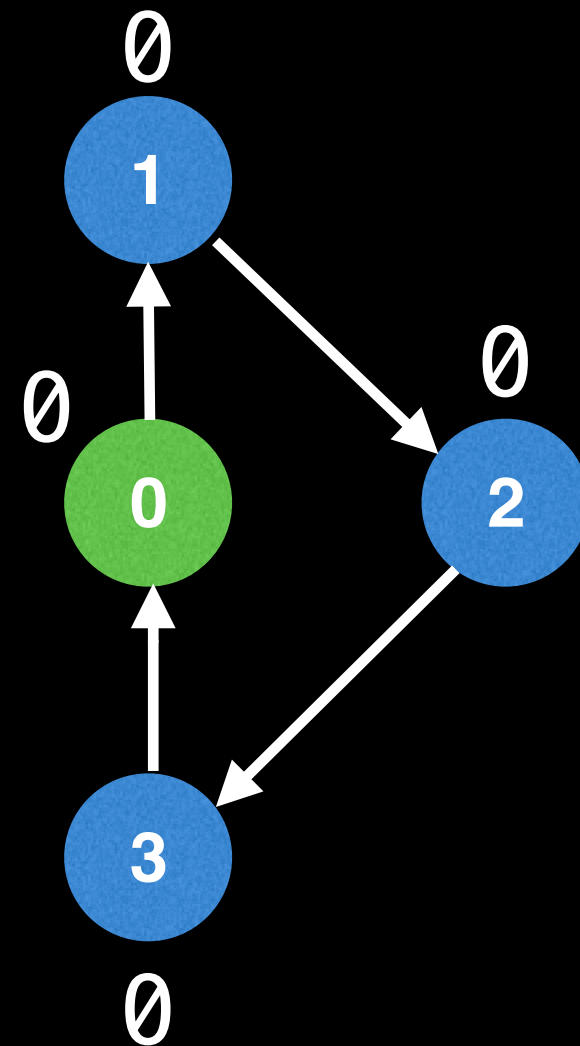


The indication of a cycle back to the original node implies an articulation point.

Articulation points

The only time `id(e.from) == lowlink(e.to)` fails is when the **starting node** has 0 or 1 outgoing directed edges. This is because either the node is a singleton (0 case) or the node is trapped in a **cycle** (1 case).

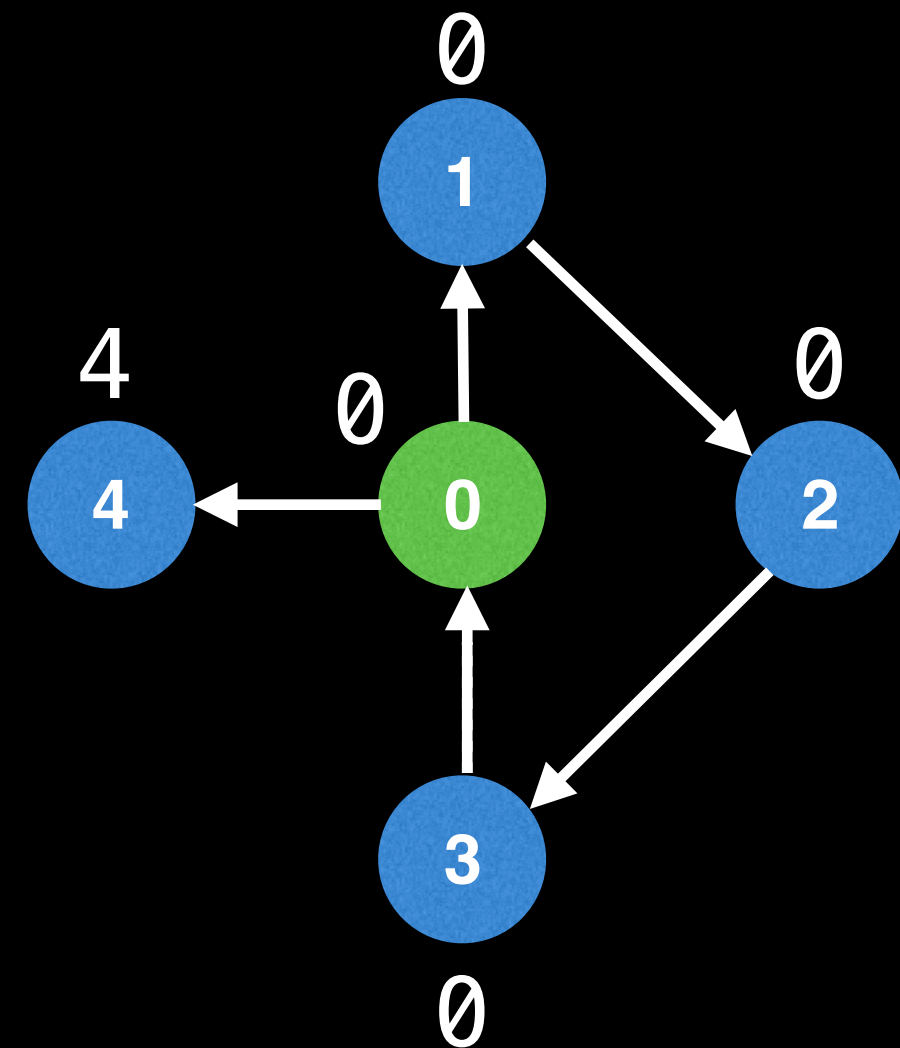
Here the condition is met, but the starting node only has 1 outgoing edge. Therefore, the start node is not an articulation point.



Articulation points

The only time `id(e.from) == lowlink(e.to)` fails is when the **starting node** has 0 or 1 outgoing directed edges. This is because either the node is a singleton (0 case) or the node is trapped in a **cycle** (1 case).

However, when there are more than 1 outgoing edges the starting node can escape the cycle and thus becomes an articulation point!



 Start node

```
id = 0
g = adjacency list with undirected edges
n = size of the graph
outEdgeCount = 0
```

```
# In these arrays index i represents node i
low = [0, 0, ... 0, 0] # Length n
ids = [0, 0, ... 0, 0] # Length n
visited = [false, ..., false] # Length n
isArt = [false, ..., false] # Length n
```

```
function findArtPoints():
    for (i = 0; i < n; i = i + 1):
        if (!visited[i]):
            outEdgeCount = 0 # Reset edge count
            dfs(i, i, -1)
            isArt[i] = (outEdgeCount > 1)
    return isArt
```

```
# Perform DFS to find articulation points.
```

```
function dfs(root, at, parent):
```

```
    if (parent == root): outEdgeCount++
```

```
    visited[at] = true
```

```
    id = id + 1
```

```
    low[at] = ids[at] = id
```

```
# For each edge from node 'at' to node 'to'
```

```
for (to : g[at]):
```

Being explicit here.

```
    if to == parent: continue
```

However, this could just be a \leq clause.

```
    if (!visited[to]):
```

```
        dfs(root, to, at)
```

```
        low[at] = min(low[at], low[to])
```

```
# Articulation point found via bridge
```

```
if (ids[at] < low[to]):
```

```
    isArt[at] = true
```

```
# Articulation point found via cycle
```

```
if (ids[at] == low[to]):
```

```
    isArt[at] = true
```

```
else:
```

```
    low[at] = min(low[at], ids[to])
```