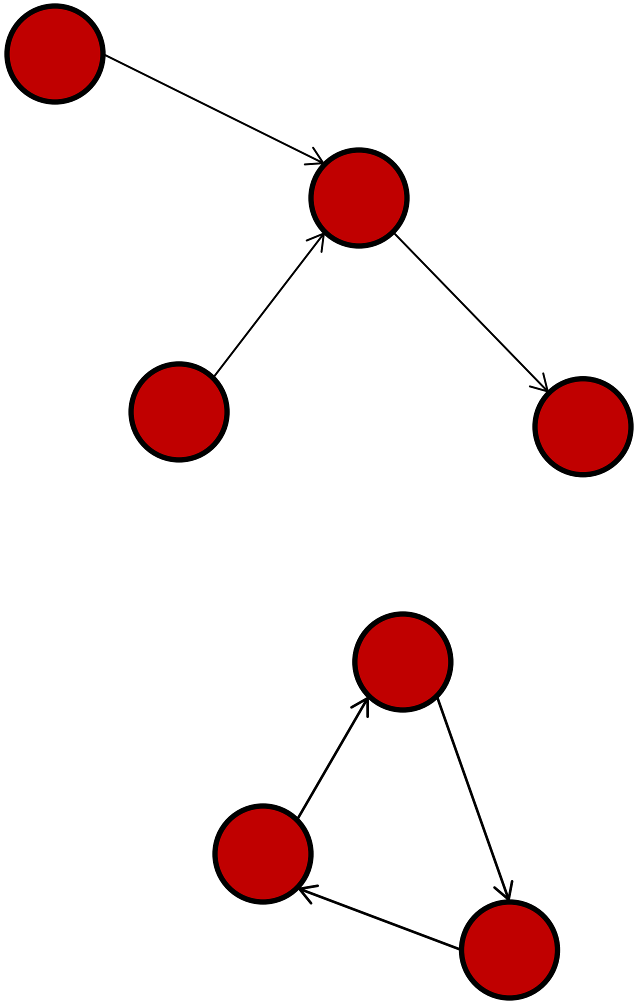
Lecture 12
Graphs and graph search

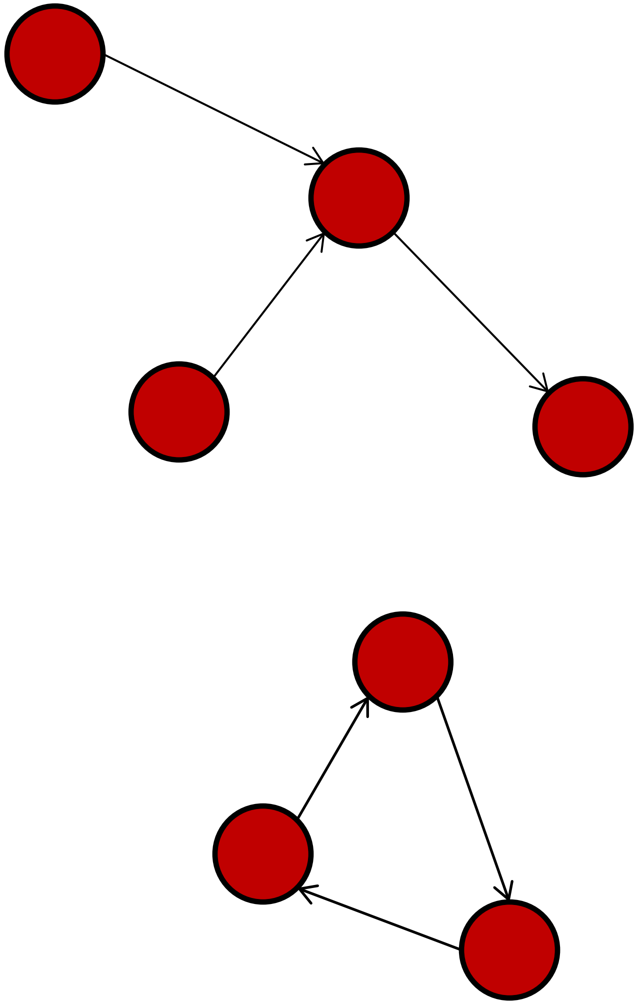
EECS-214

Graphs

- Informally, a **graph** is a network of objects connected by lines or arrows
- The objects are called **nodes** or **vertices** and the lines are called **edges**



Graphs

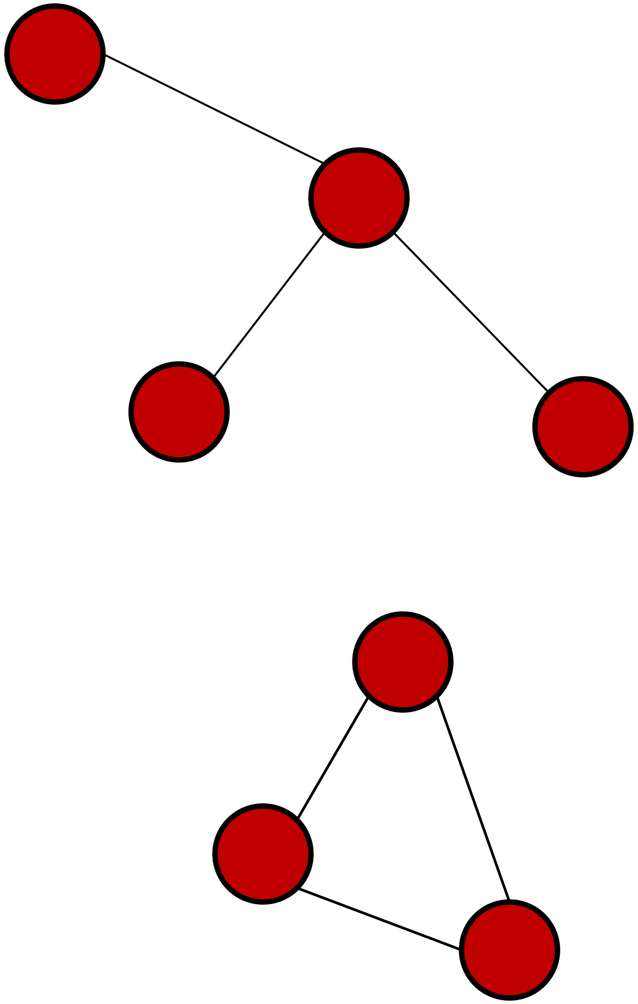


Formally, a **directed graph** (or **digraph**) is a pair, $G = (V, E)$ where

- V is the set of vertices of the graph
- $E \subseteq V \times V$ is the set of edges of the graph
- $(v_1, v_2) \in E$ iff the graph has an edge (arrow) from v_1 to v_2

Note: you can also think of a digraph as a relation on the vertices

Graphs



An **undirected graph** is the same except that

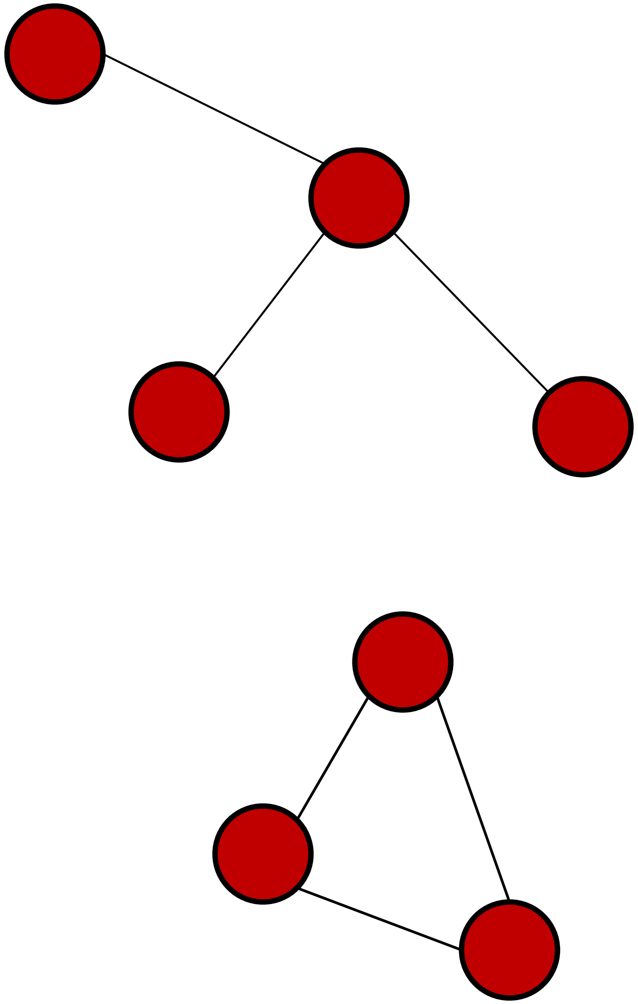
- E is a set of **two-element subsets** of V rather than a set of pairs
- $\{ v_1, v_2\} \in E$ iff the graph has an edge (line) v_1 between v_2

- Undirected graphs don't distinguish between an edge from v_1 to v_2 and an edge from v_2 to v_1
  - Hence they're drawn with lines rather than arrows

Note: you can also think of an undirected graph as a symmetric relation on the vertices

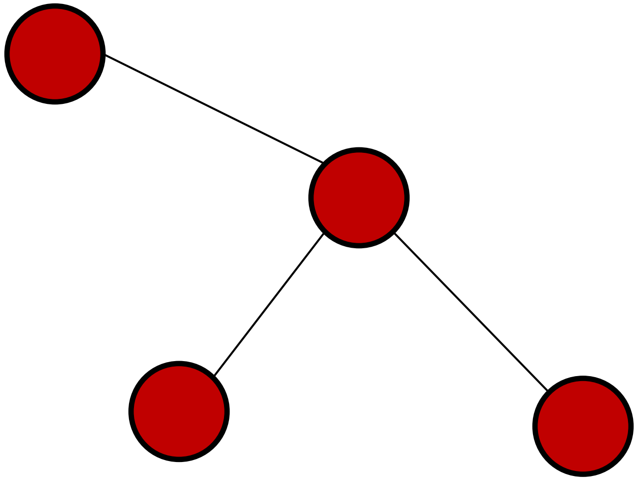
Subgraphs

A **subgraph** is just a subset of the vertices and/or edges of some other graph



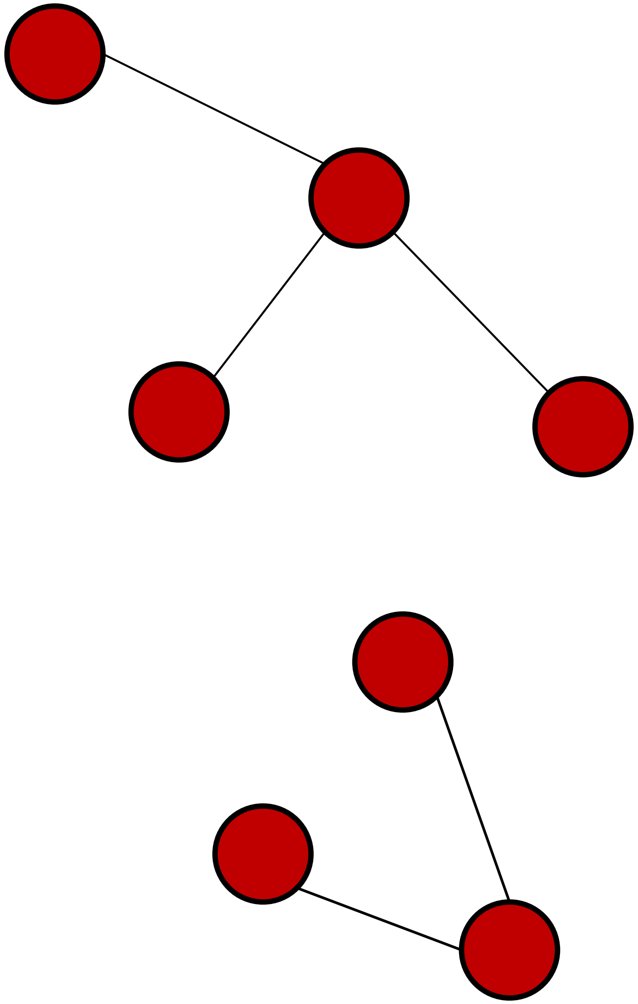
Subgraphs

A **subgraph** is just a subset of the vertices and/or edges of some other graph



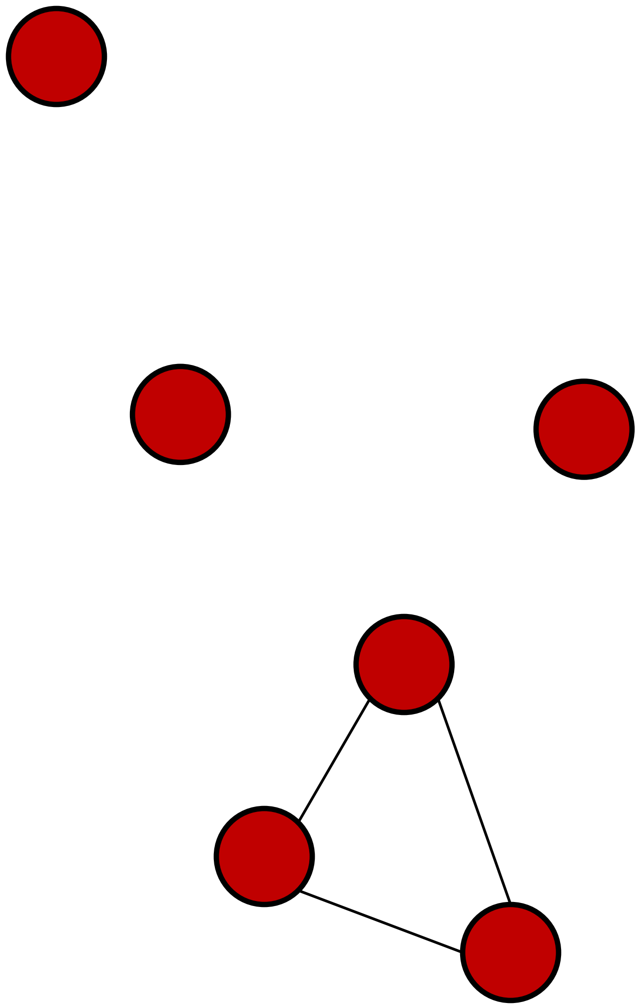
Subgraphs

A **subgraph** is just a subset of the vertices and/or edges of some other graph



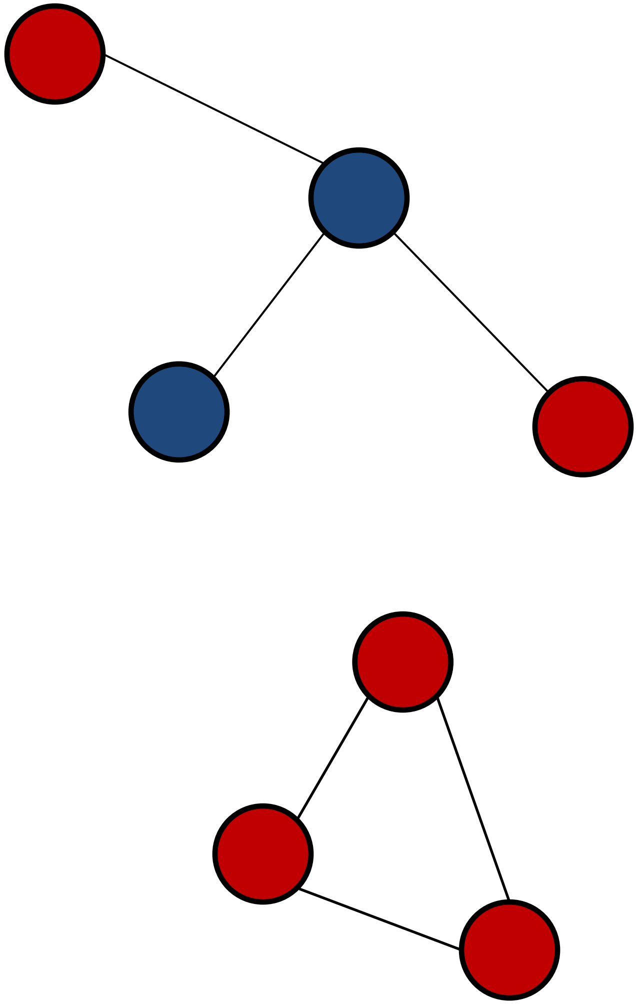
Subgraphs

A **subgraph** is just a subset of the vertices and/or edges of some other graph



Connectivity

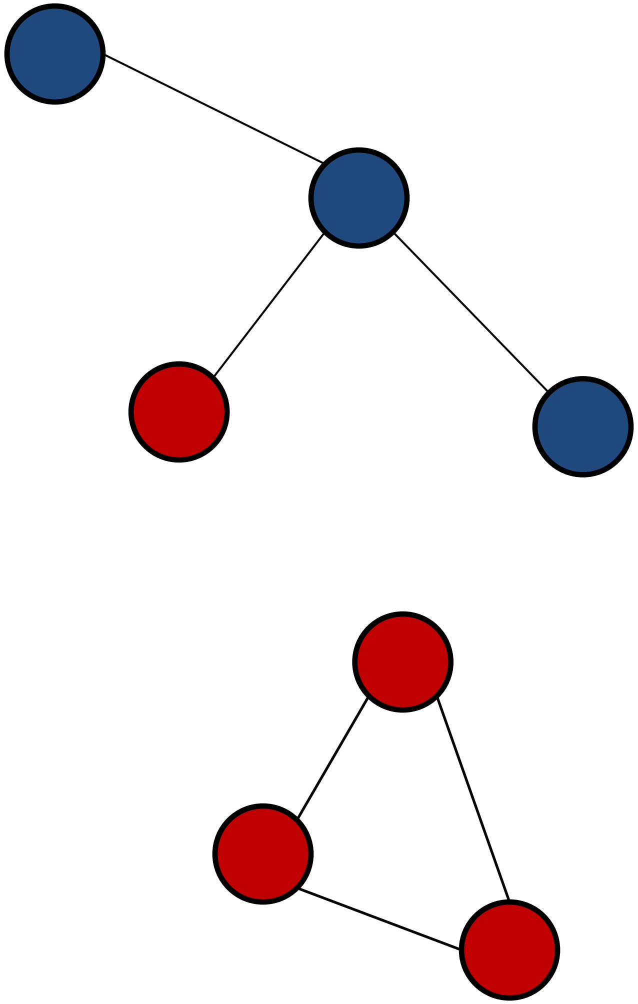
- Two vertices linked by an edge are said to be **adjacent**



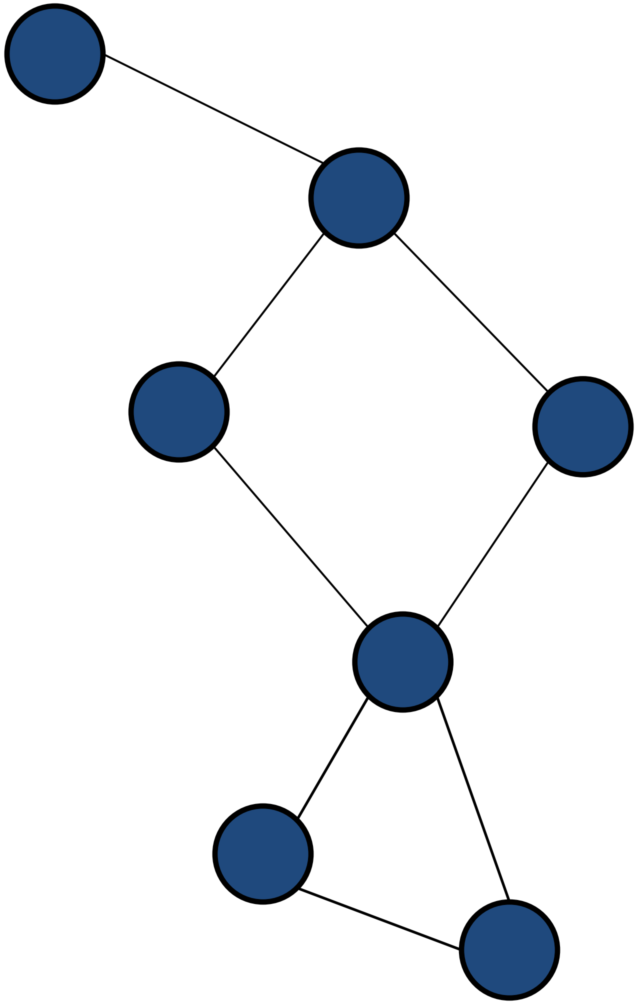
Paths

- A **path** is a series of adjacent vertices that doesn't repeat the same vertex
  - For a directed graph, the path has to follow the direction of the edges

- Two vertices are **connected**, if there is a path between them



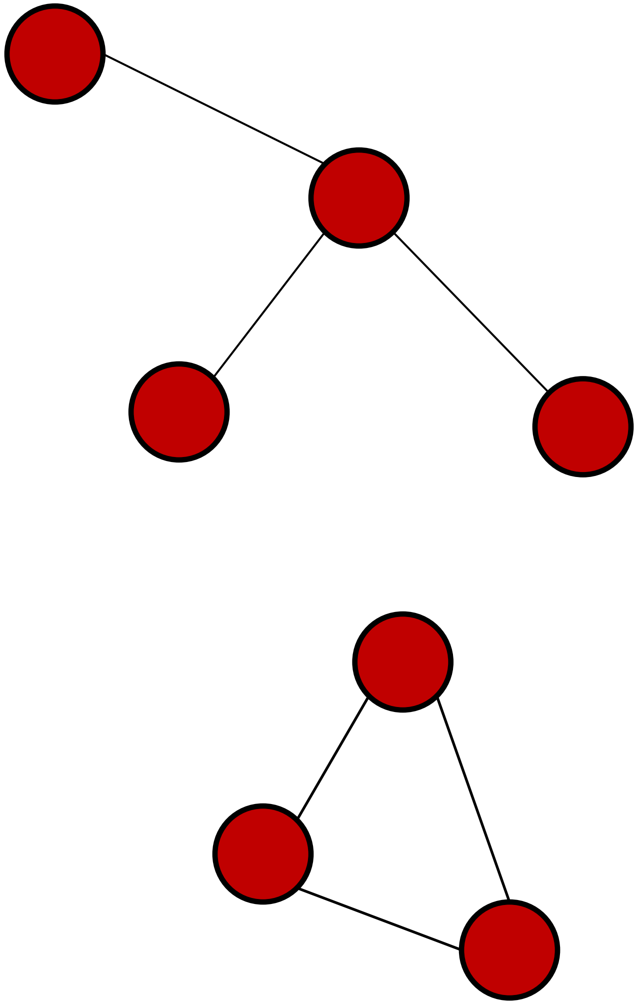
Connectivity



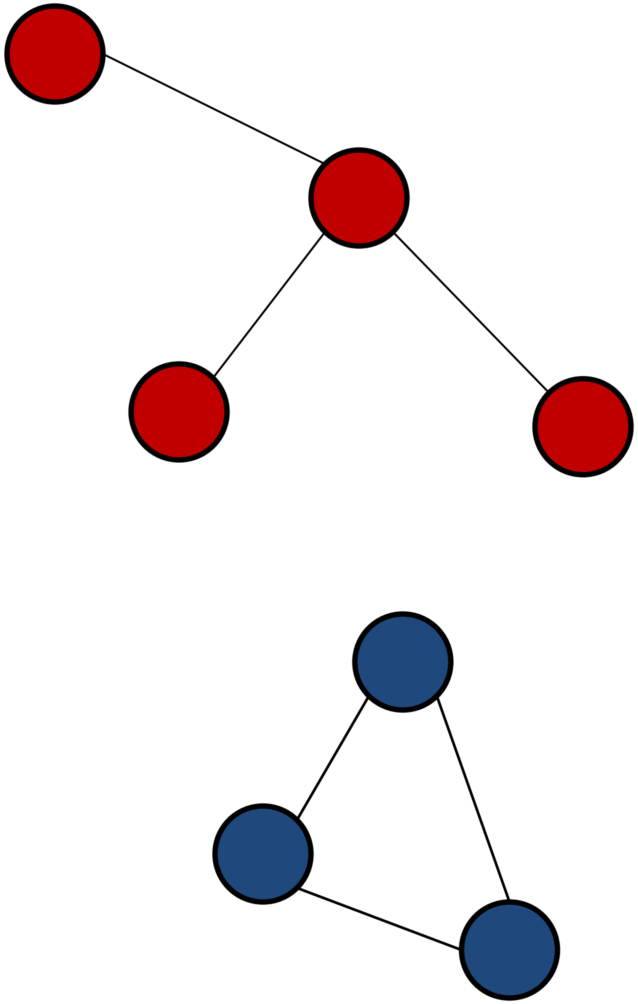
A graph is **connected** if every vertex in the graph is connected to every other vertex

Connected components

- Not all graphs are connected



Connected components

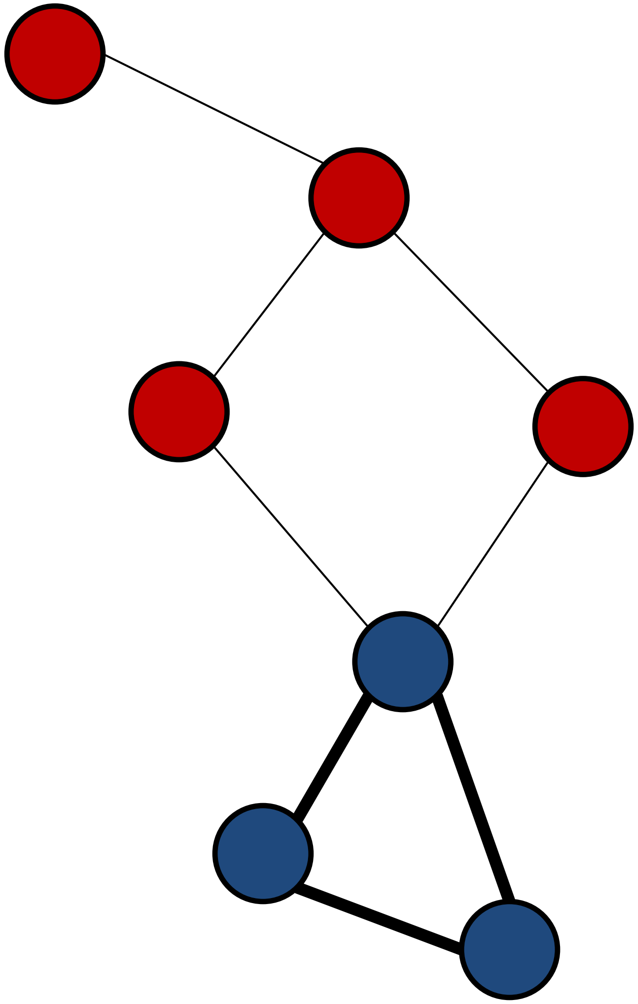


- Not all graphs are connected
- The connected components of a graph of the **largest subgraphs** of the graph you can form that are themselves connected

Cycles

- A **cycle** is a path that starts and ends with the same vertex
  - (Okay, I know we said paths can't repeat the same vertex, but we'll allow the first and the last to be the same)

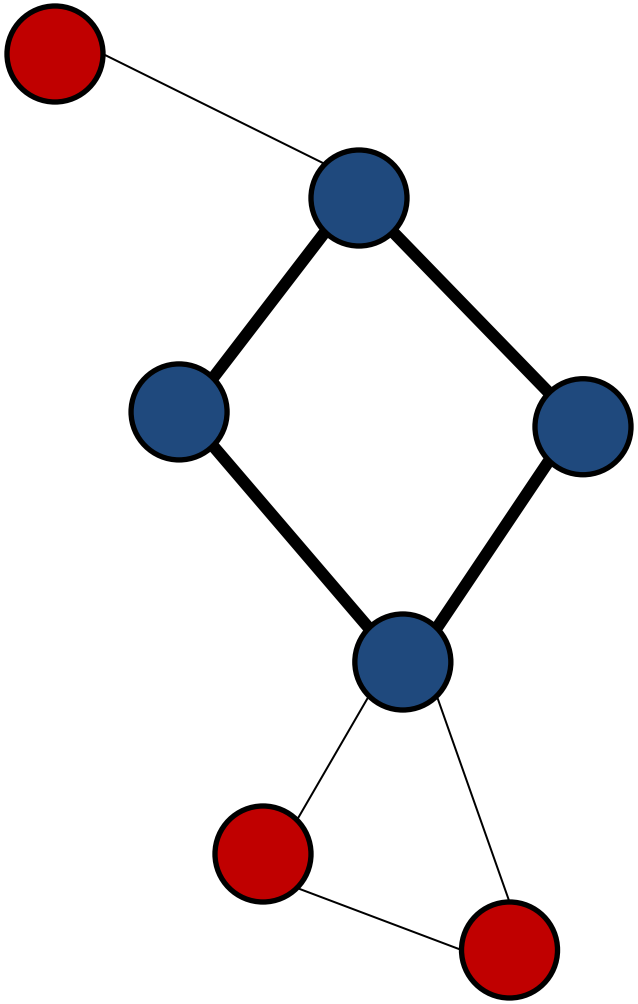
- A graph is **cyclic** if it contains at least one cycle



Cycles

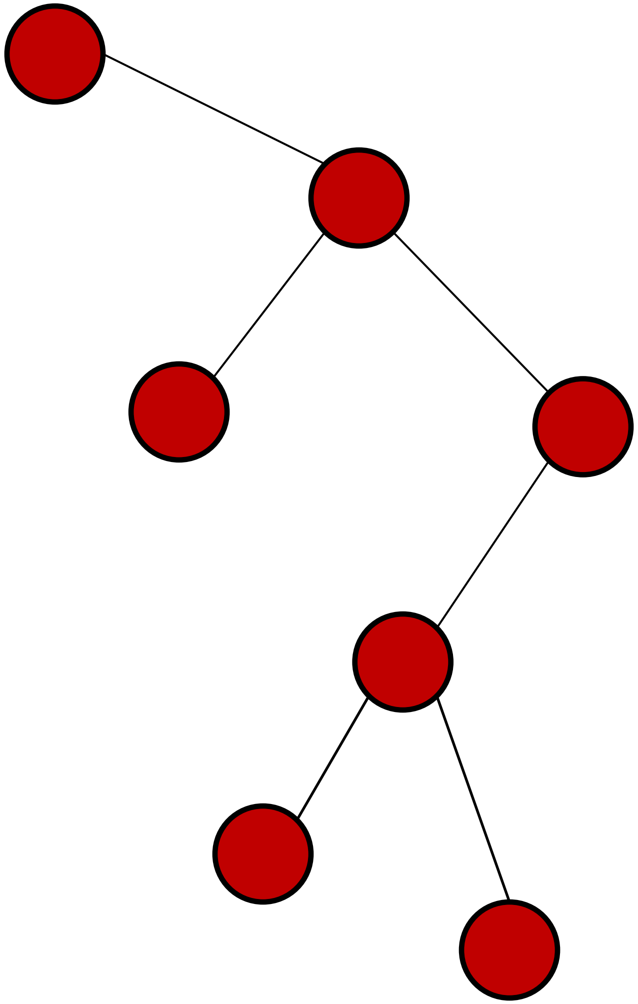
- A **cycle** is a path that starts and ends with the same vertex
  - (Okay, I know we said paths can't repeat the same vertex, but we'll allow the first and the last to be the same)
- A graph is **cyclic** if it contains at least one cycle

(This graph has 2 cycles)



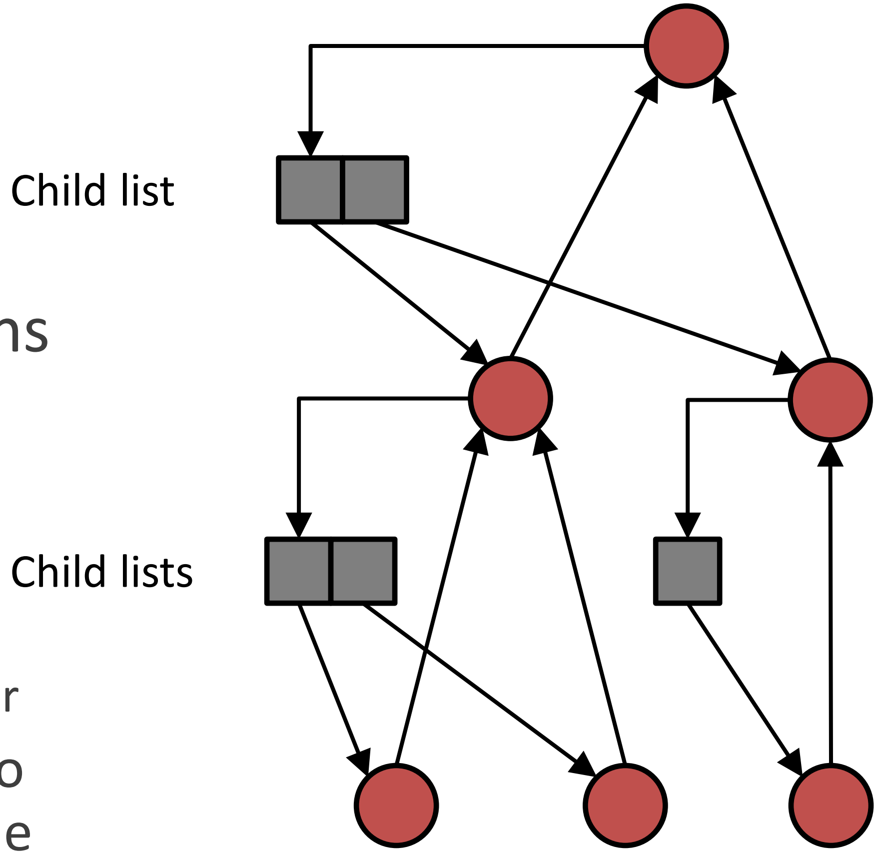
Trees

- A **tree** is a graph in which any pair of nodes has exactly one path between them



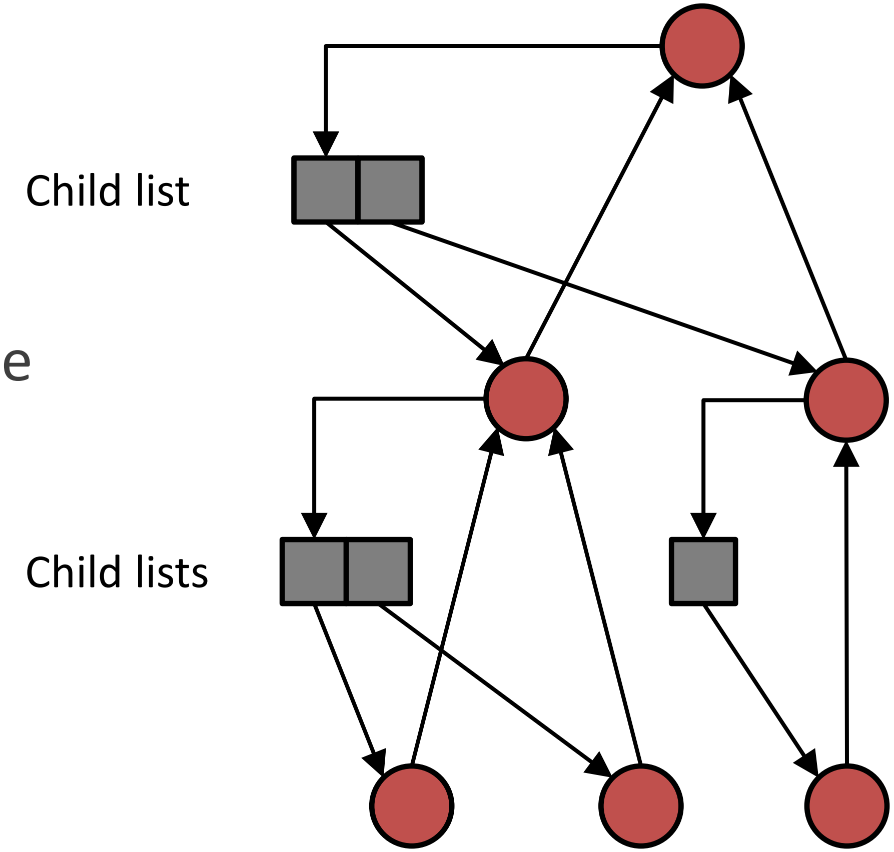
General representation of trees

- Each tree **node is an object**
  - (Red circles)
- Each node object contains
  - **Parent**
    - (Upward arrows)
  - List of **children**
    - (Grey boxes)
    - linked list, array, whatever
  - **Anything else** you want to remember about the node



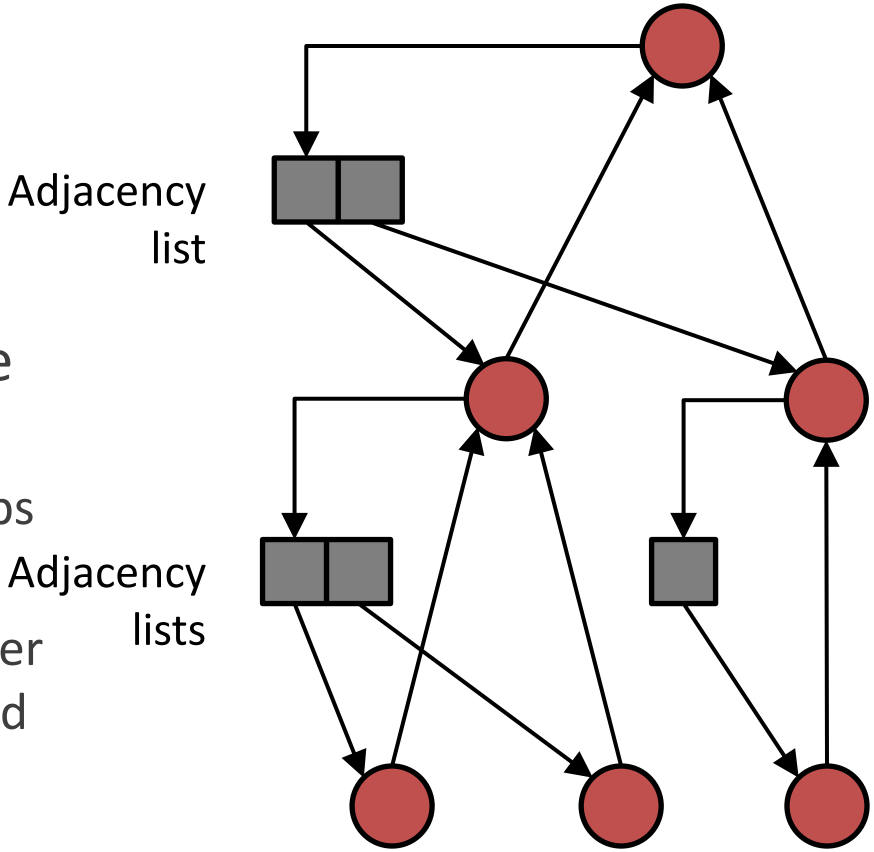
General representation of graphs

- Graphs don’t have the **special properties of trees**
  - Graph doesn’t have a distinguished **root** node
    - So there aren’t **parent/child relationships**



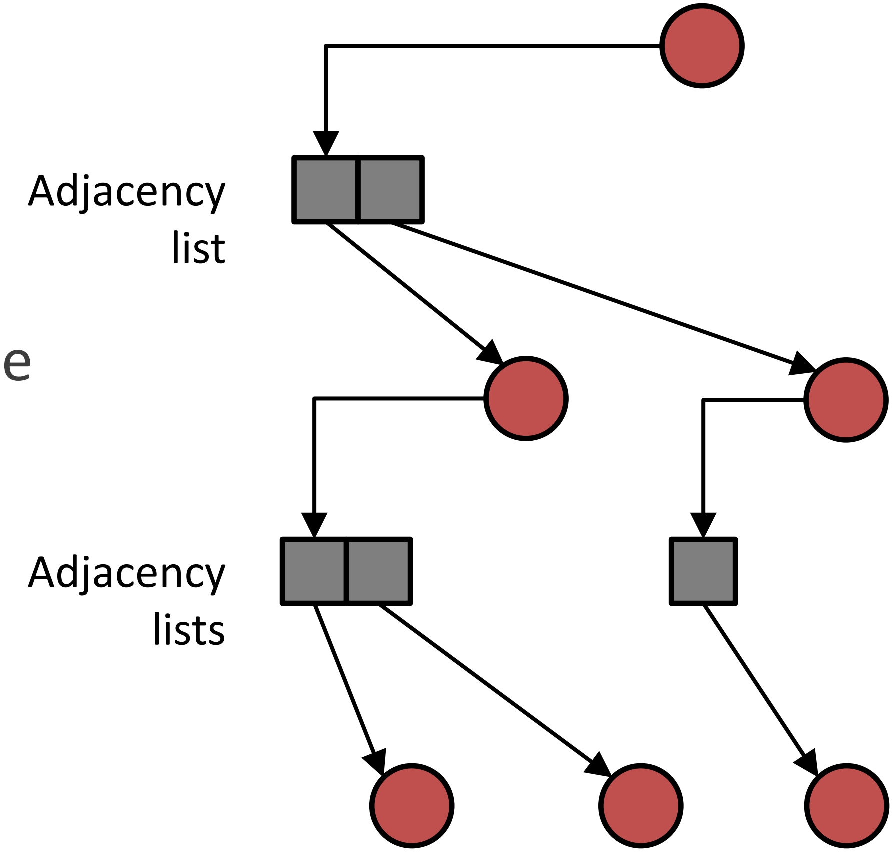
General representation of graphs

- Graphs don't have the **special properties of trees**
  - Graph doesn't have a distinguished **root** node
    - So there aren't parent/child relationships
    - Just "**adjacency**" relationships, i.e. whether two nodes are connected



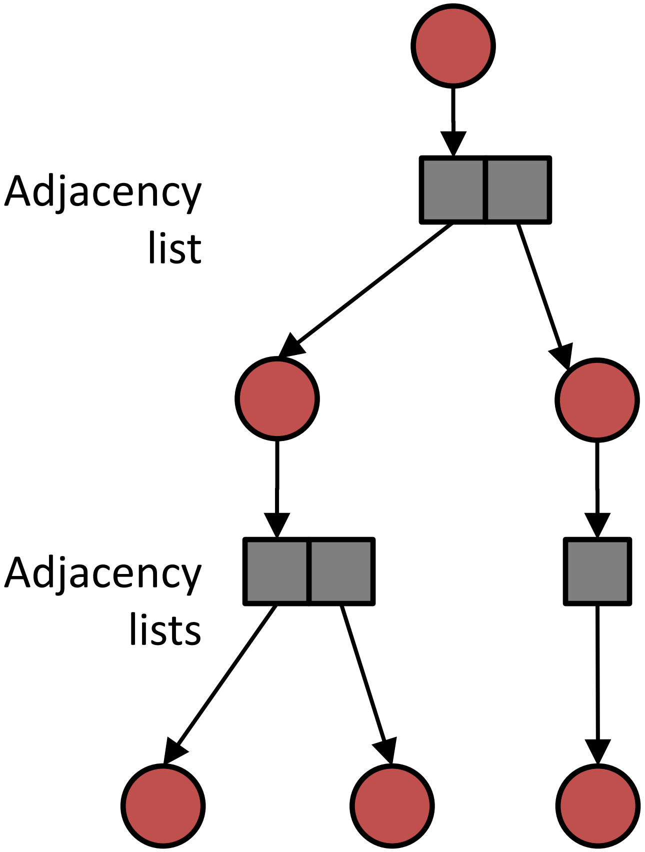
General representation of graphs

- Graphs don't have the **special properties of trees**
  - Graph doesn't have a distinguished **root** node
  - Nodes don't have a distinguished **parent** node



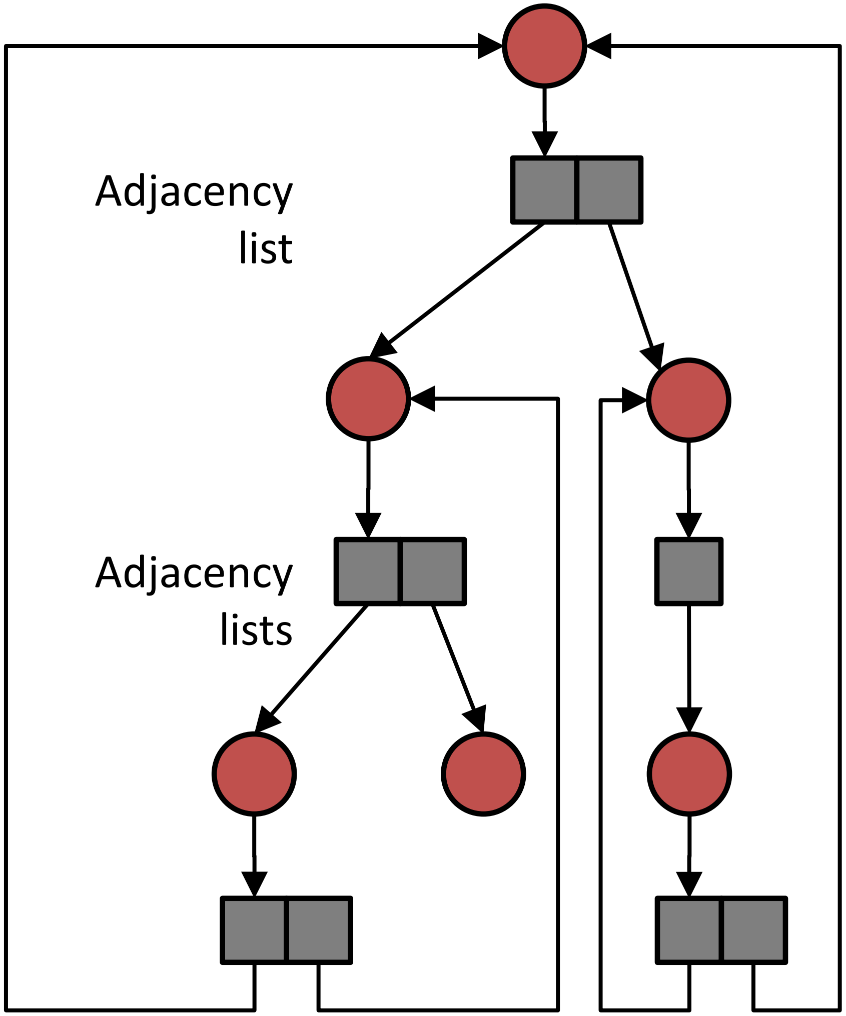
General representation of graphs

- Graphs don’t have the **special properties of trees**
  - Graph doesn’t have a distinguished **root** node
  - Nodes don’t have a distinguished **parent** node



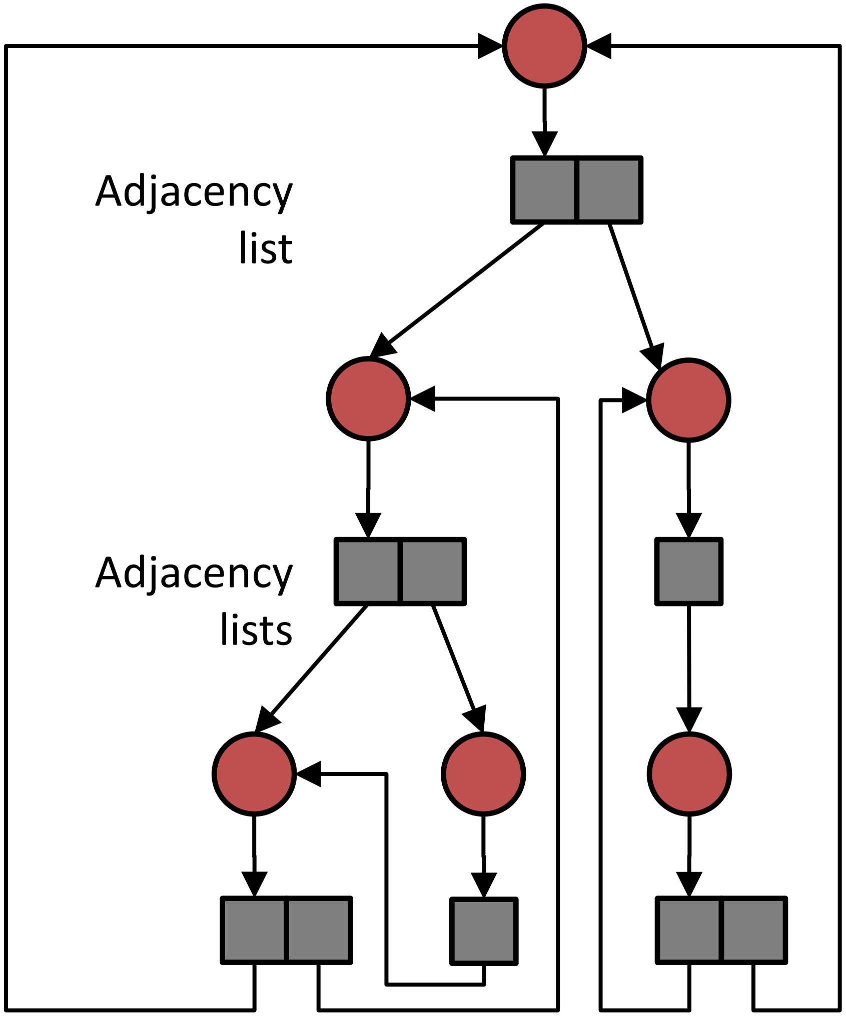
General representation of graphs

- Graphs don't have the **special properties of trees**
  - Graph doesn't have a distinguished **root** node
  - Nodes don't have a distinguished **parent** node
  - Can have **cycles**



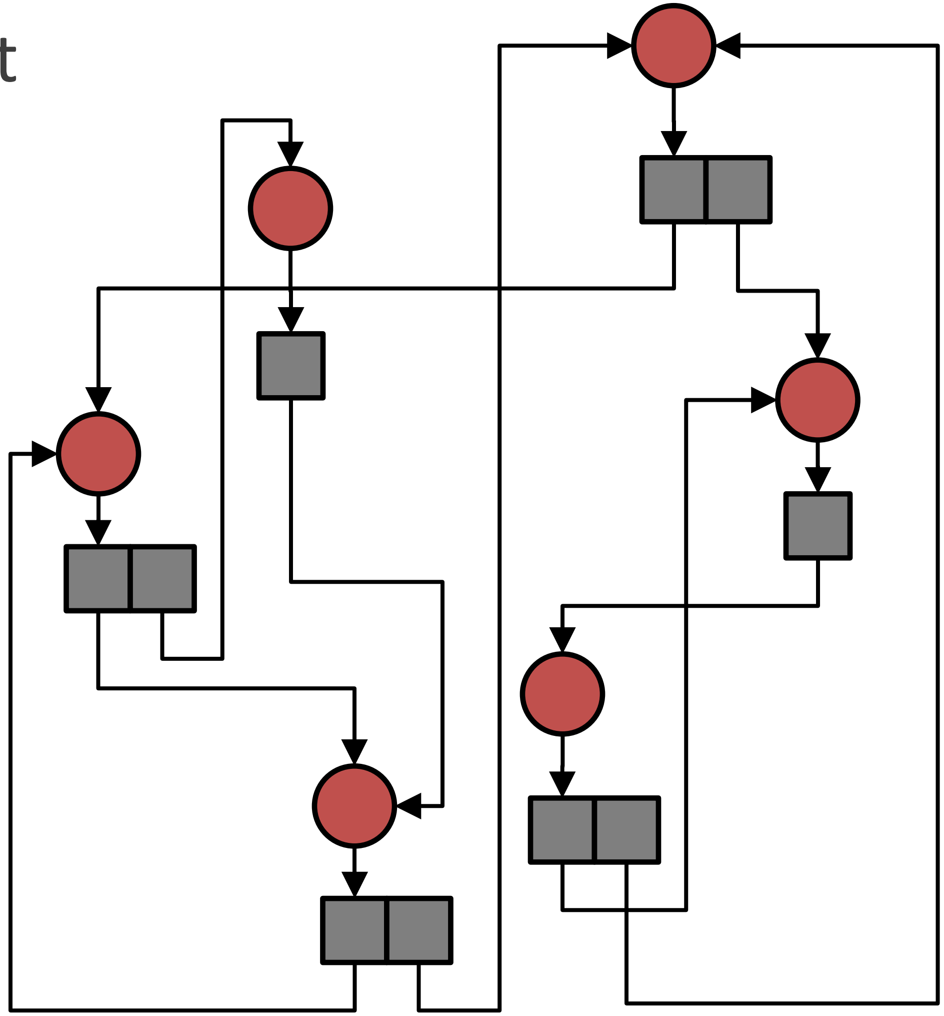
General representation of graphs

- Graphs don’t have the **special properties of trees**
  - Graph doesn’t have a distinguished **root** node
  - Nodes don’t have a distinguished **parent** node
  - Can have **cycles**
  - Or other **complicated topologies**



General representation of graphs

- And we don’t think of it as being organized into **levels**

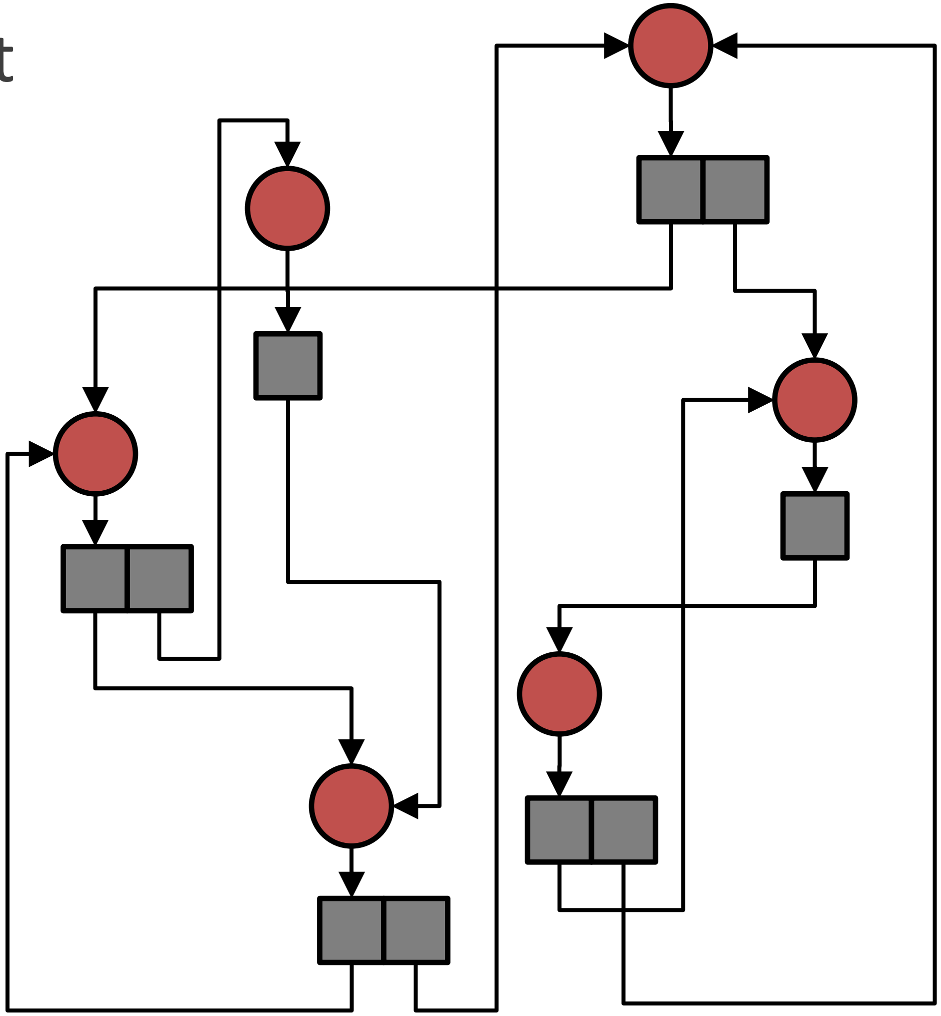


Same graph

General representation of graphs

- And we don't think of it as being organized into **levels**

- But we otherwise tend to represent them very **similarly** to trees



Same graph

Adjacency list representation

- For the most part, graph representations should be **pretty obvious**
  - Each **node lists the nodes its connected to**
- This is called an **adjacency list** representation
  - **Linked lists** are often used

```
class GraphNode {
  AdjListCell first;
}

class AdjListCell {
  GraphNode node;
  AdjListCell next;
}
```

Adjacency list representation

- For the most part, graph representations should be **pretty obvious**
  - Each **node lists the nodes its connected to**

- This is called an **adjacency list** representation
  - **Linked lists** are often used
  - But **arrays** or any other sequence representation can be used for the adjacency lists themselves

```
class GraphNode {
  GraphNode[ ] adjacent;
}
```

Array of adjacency lists

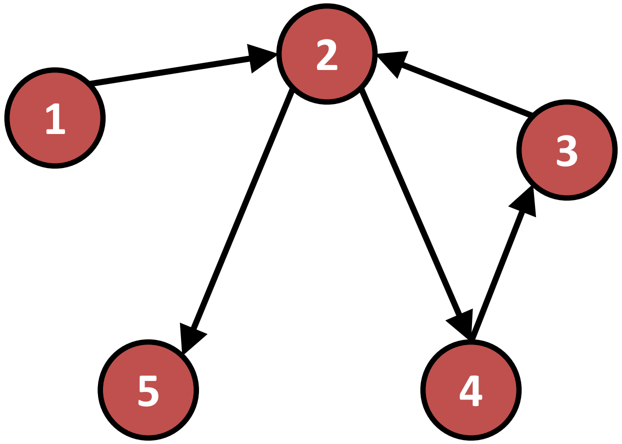
- The CLR book uses a specific graph representation
  - **Number all the nodes**
  - **Array of adjacency lists**, indexed by node number
- Adjacency list of node i is stored in adj[i]

```
class Graph {
   AdjListCell[] adj;
}


class AdjListCell {
   int nodeNumber;
   AdjListCell next;
}
```

Adjacency matrix representation

- Finally, we can store the edges for a graph in a **matrix**
  - Again, **number the nodes**
  - Define the matrix $E = (e_{i,j})$ by the rule:

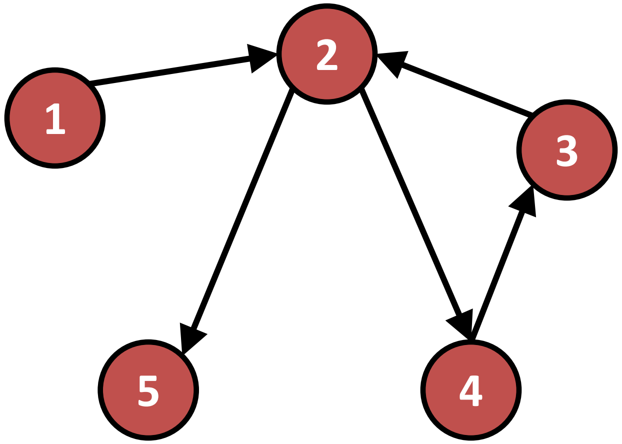
$$e_{i,j} = \begin{cases} 1, & \text{if edge from i to j} \\ 0, & \text{otherwise} \end{cases}$$



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Adjacency matrix representation

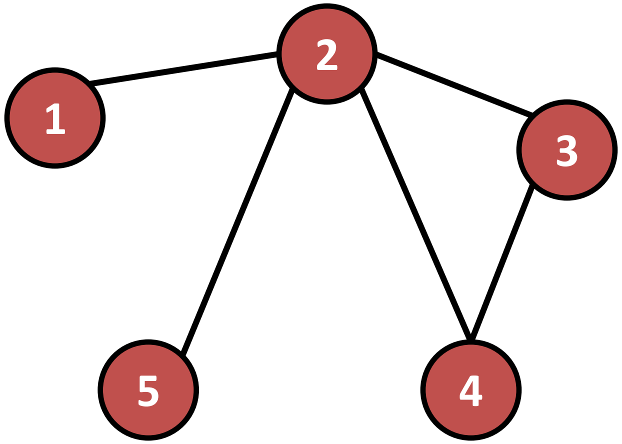
- Adjacency matrices are especially good for representing **dense** graphs
  - Graphs in which most nodes have edges to most other nodes



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Undirected graphs

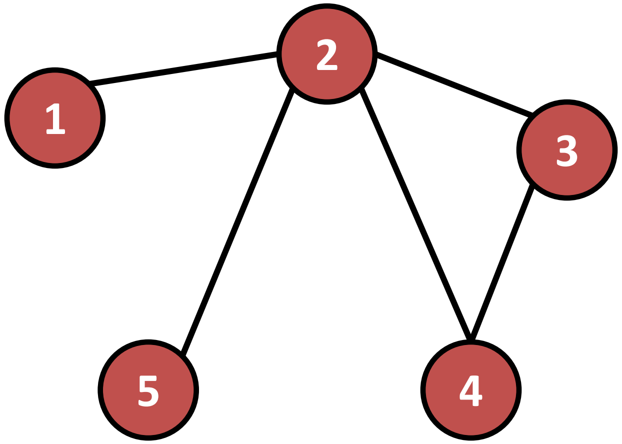
- Recall that **undirected** graphs don't distinguish between
  - An edge from a to b, and
  - An edge from b to a

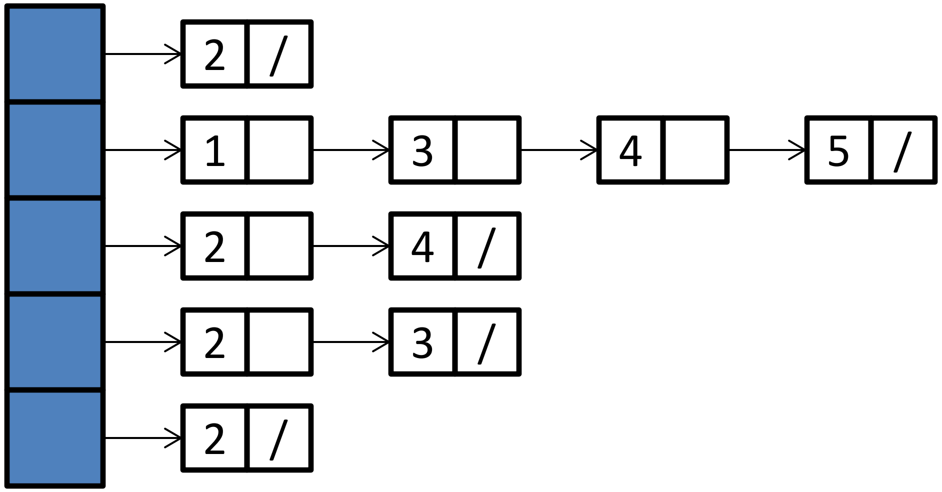


$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Undirected graphs

- These are easily handled by both representations
  - Adjacency lists: link a to b but also link b to a
  - Adjacency matrix: matrix is just a **symmetric** matrix

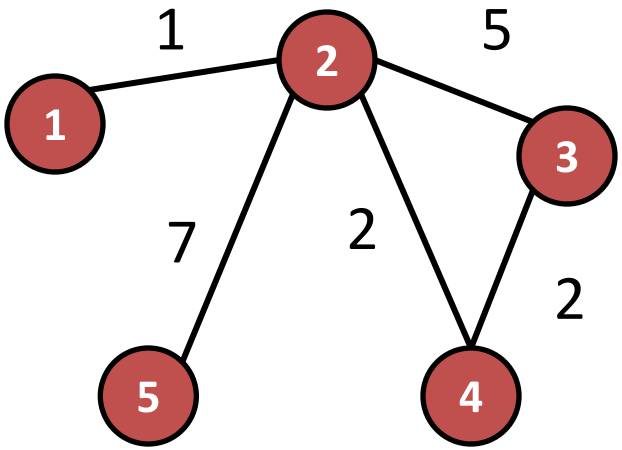


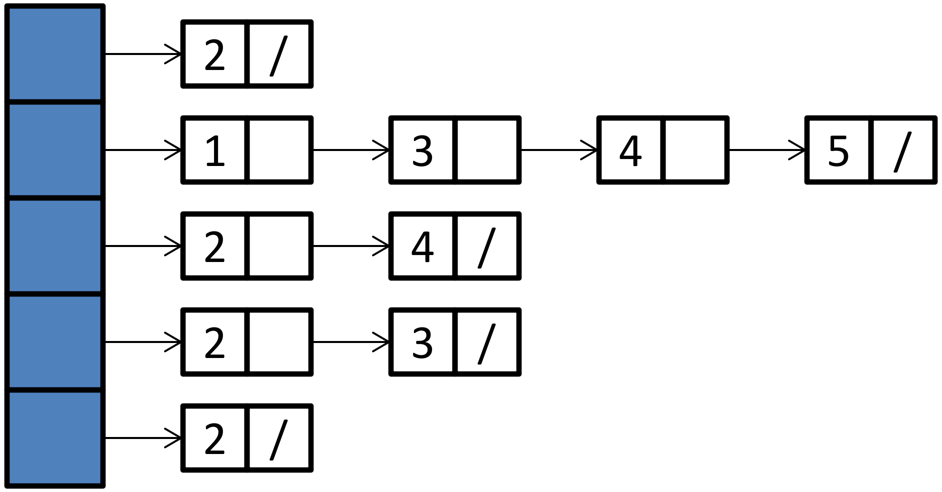


$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Weighted graphs

- Edges are labeled with **numerical values** (weights)
  - Usually represents some kind of **distance or abstract “cost”**

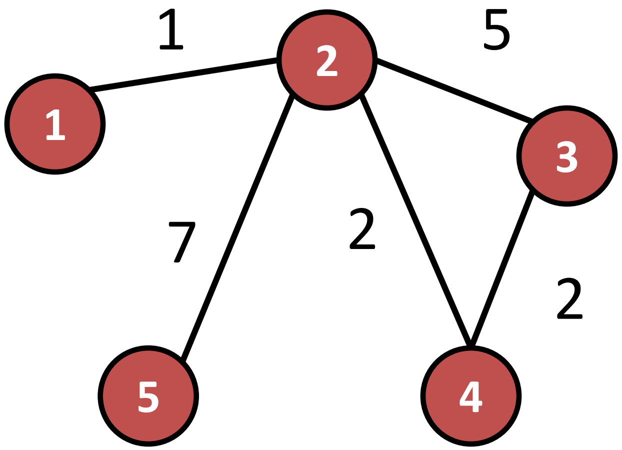


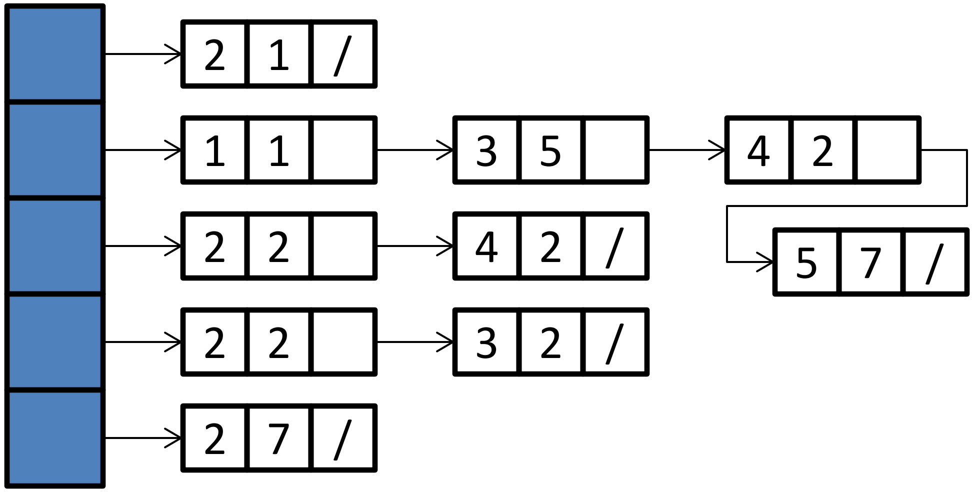


$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Representing weighted graphs

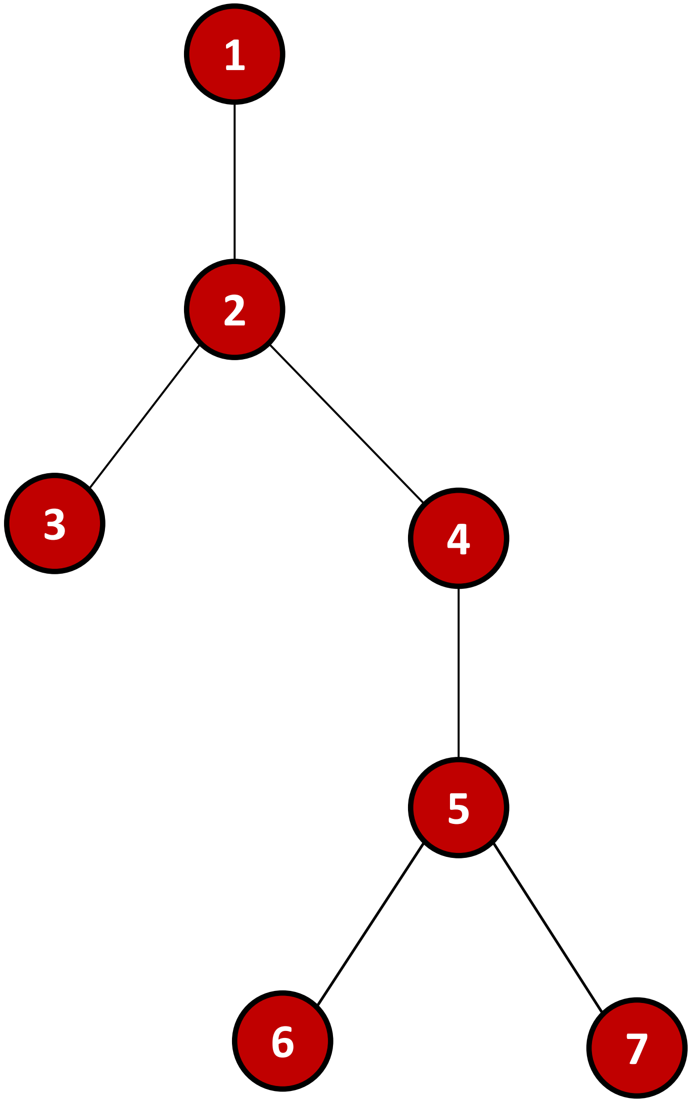
- Adjacency **lists**
  - Just **add** another **field** to the list cells to hold the weight
- Adjacency **matrix**
  - Use the weight as the **matrix entry**
  - Or 0 (or ∞) for non-adjacent nodes





$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 5 & 2 & 7 \\ 0 & 5 & 0 & 2 & 0 \\ 0 & 2 & 2 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 \end{bmatrix}$$

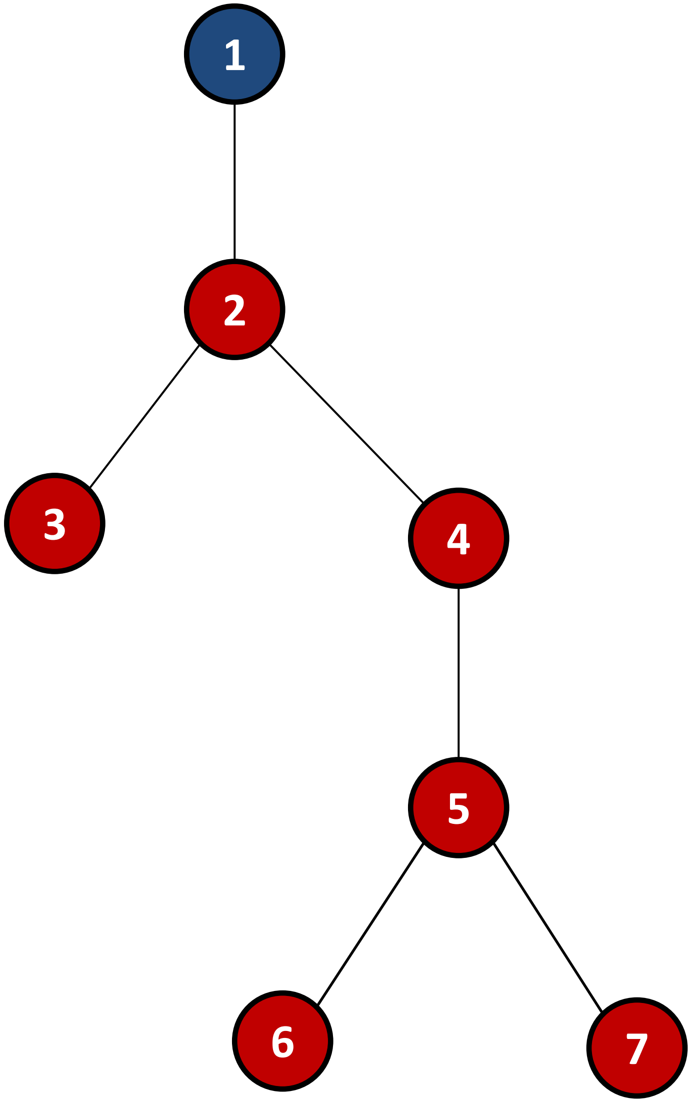
Breadth-first tree walk



Queue: 1

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

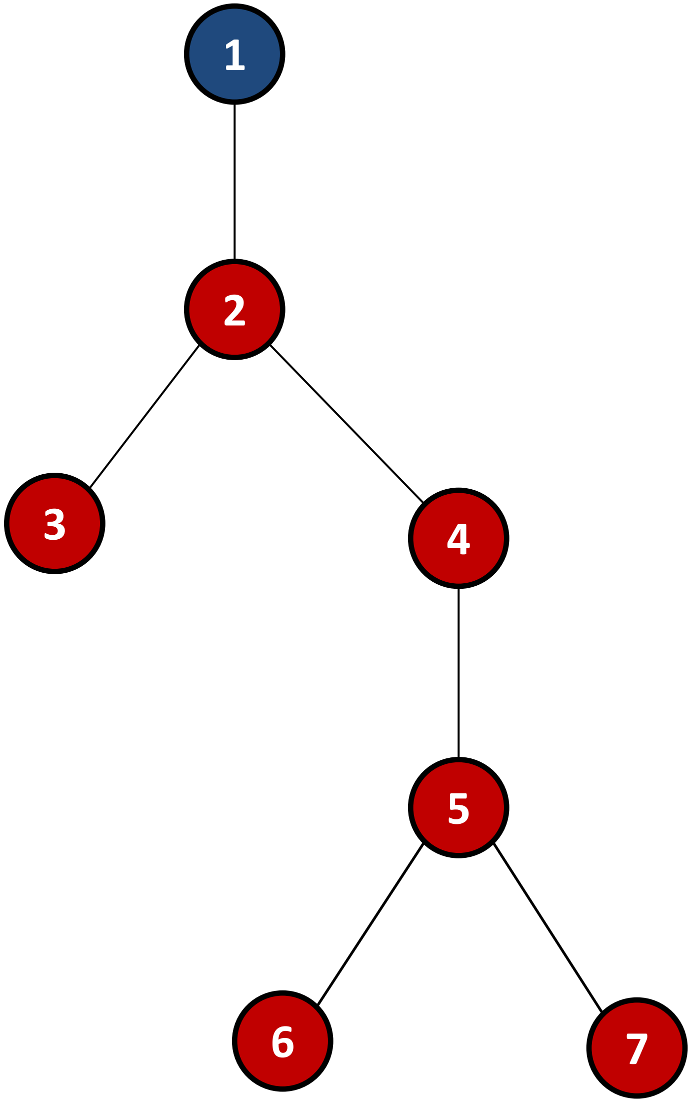
Breadth-first tree walk



Queue:

```
BreadthFirst(root) {
  q = empty queue
  q.Enqueue(root)
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      q.Enqueue(c)
  }
}
```

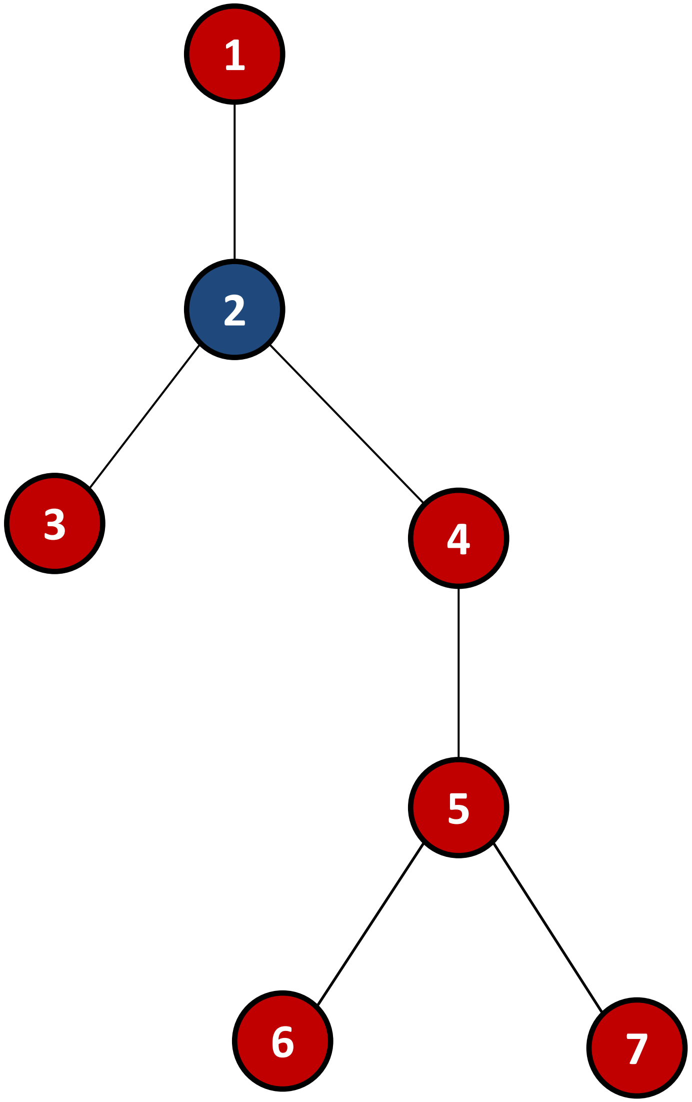
Breadth-first tree walk



Queue: 2

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

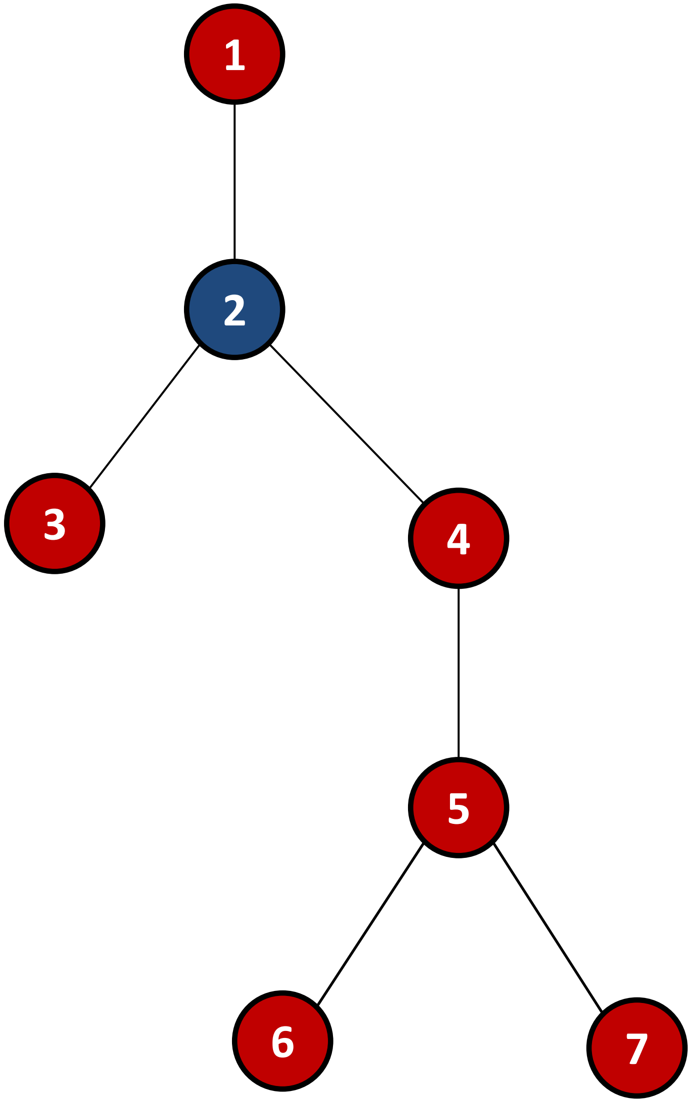
Breadth-first tree walk



Queue:

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

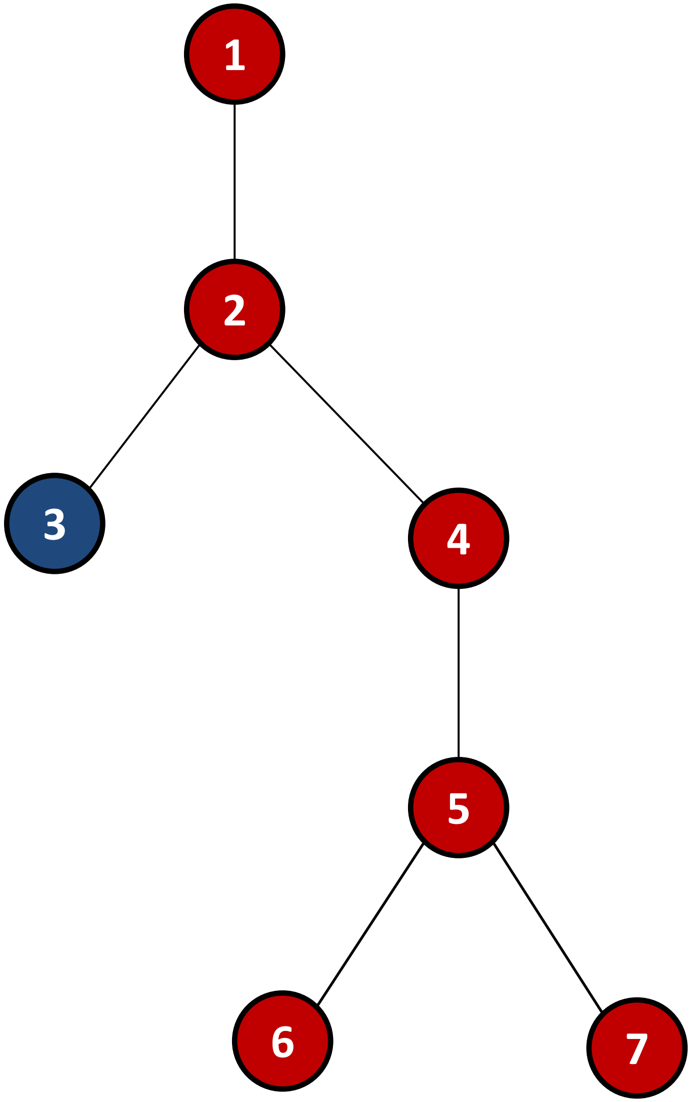
Breadth-first tree walk



Queue: 3 4

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

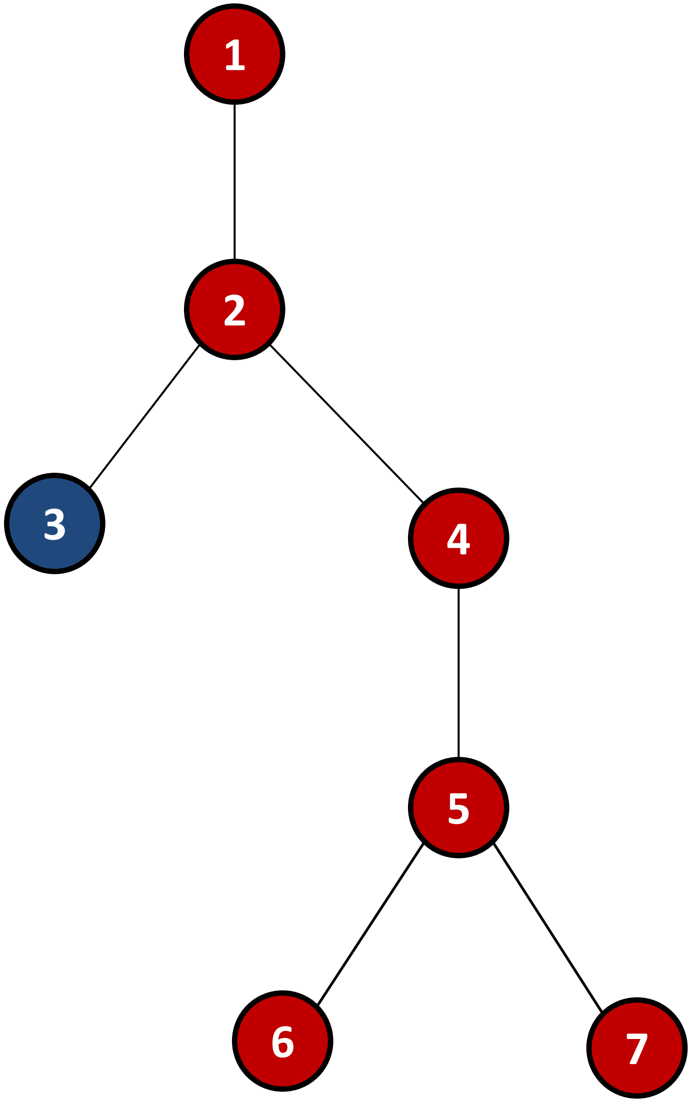
Breadth-first tree walk



Queue: 4

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

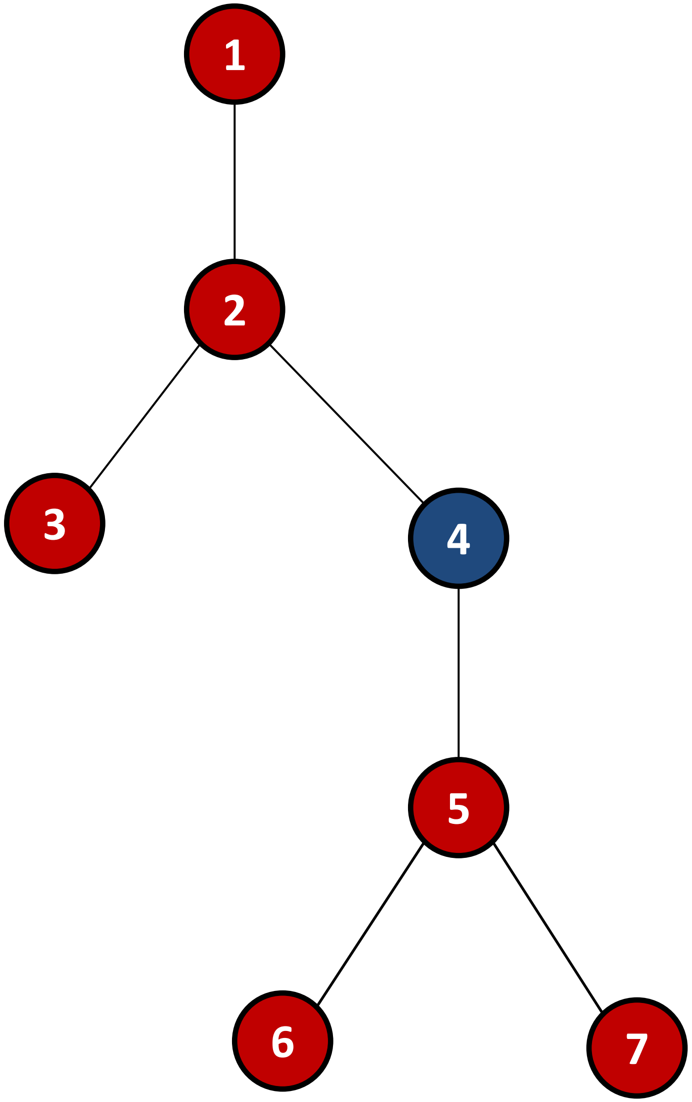
Breadth-first tree walk



Queue: 4

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

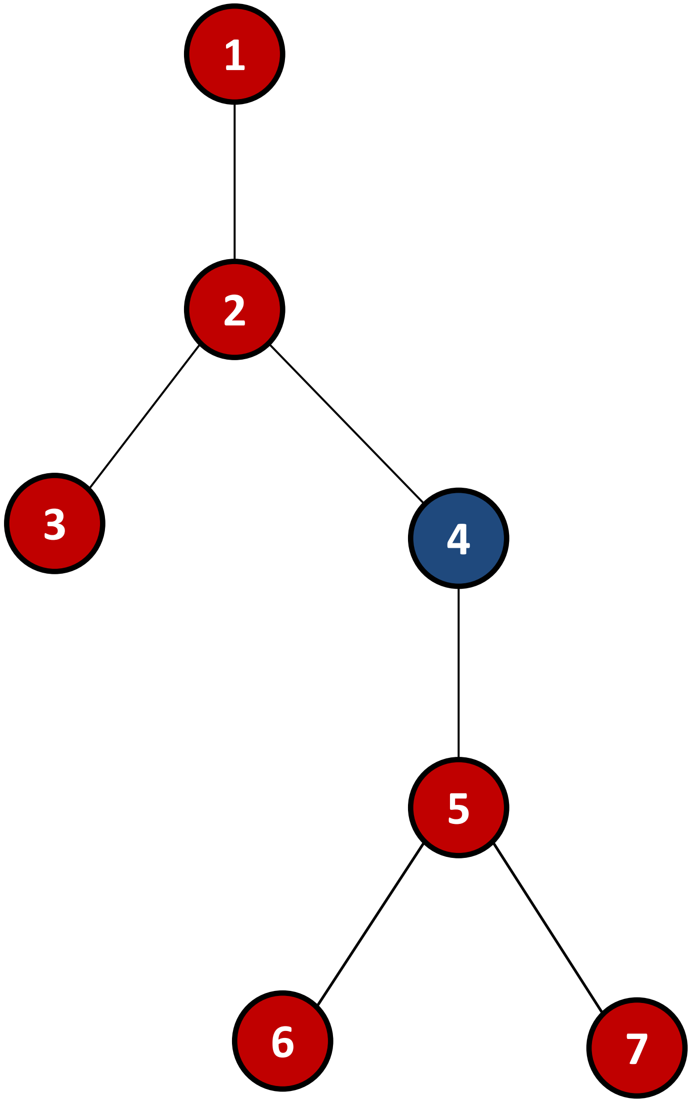
Breadth-first tree walk



Queue:

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

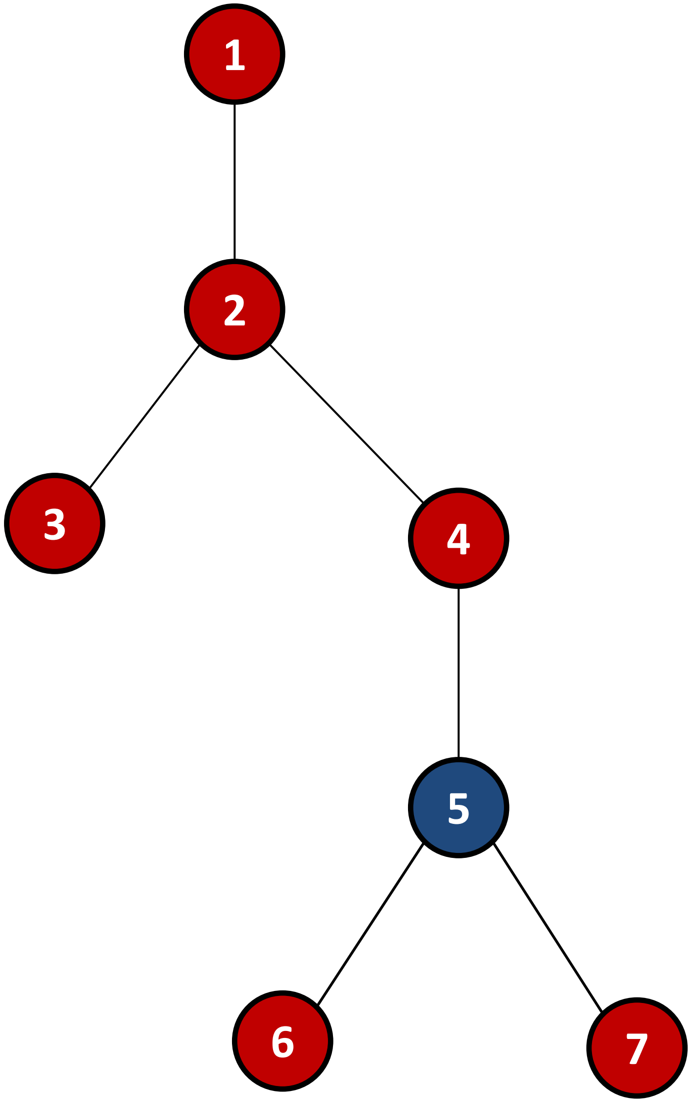
Breadth-first tree walk



Queue: 5

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

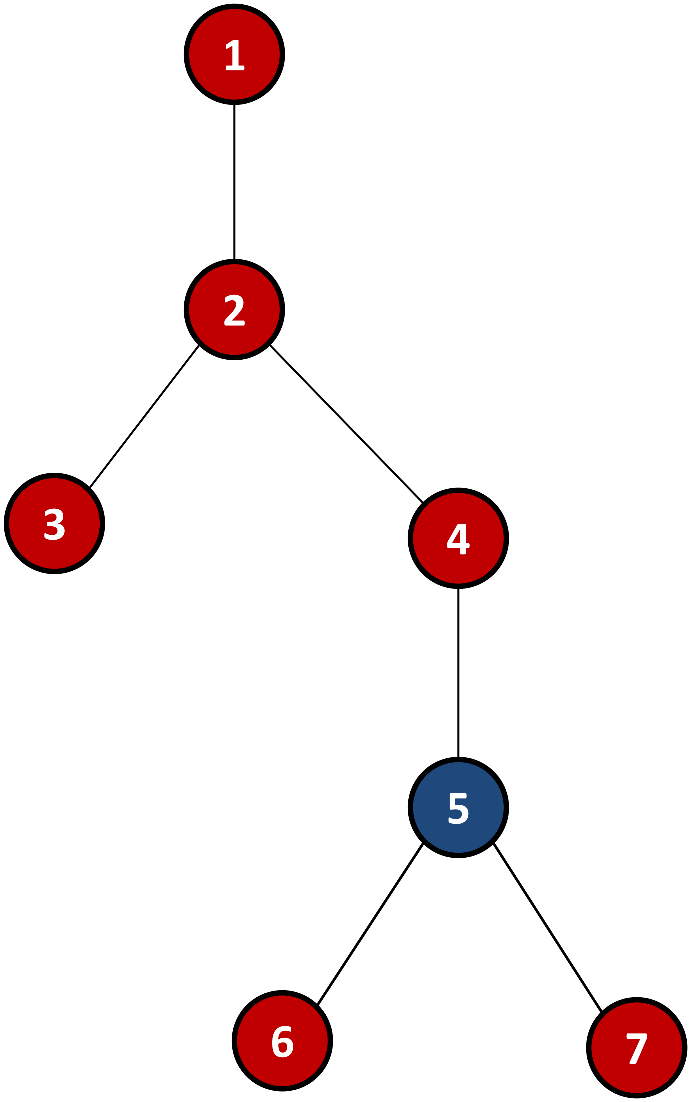
Breadth-first tree walk



Queue:

```
BreadthFirst(root) {
  q = empty queue
  q.Enqueue(root)
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      q.Enqueue(c)
  }
}
```

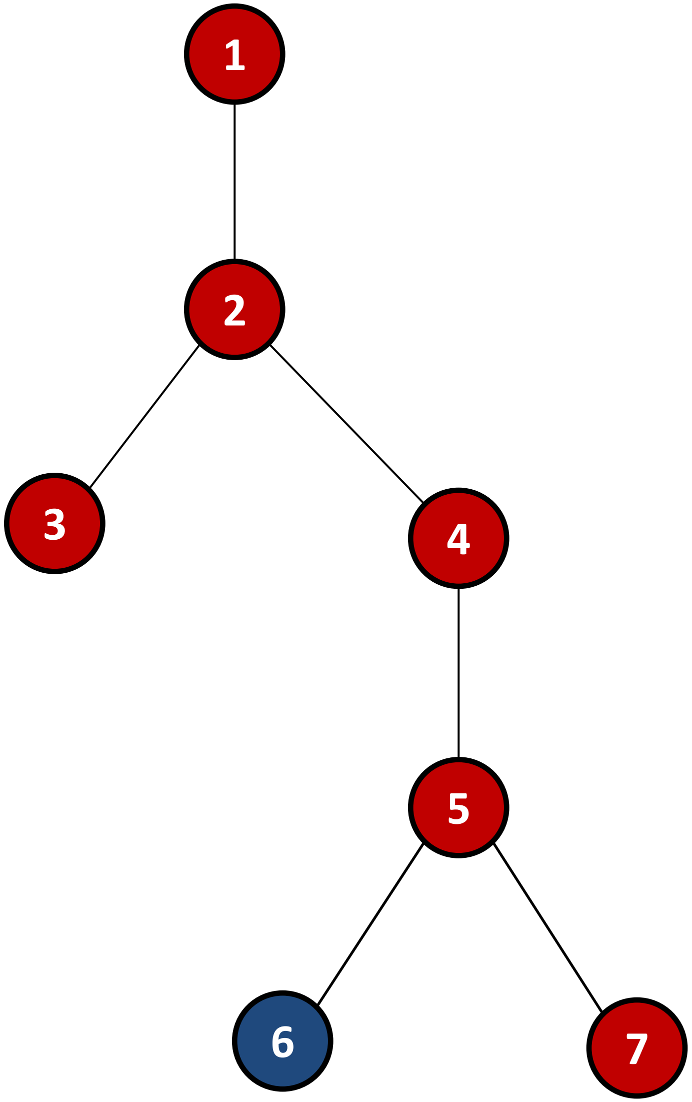
Breadth-first tree walk



Queue: 6 7

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

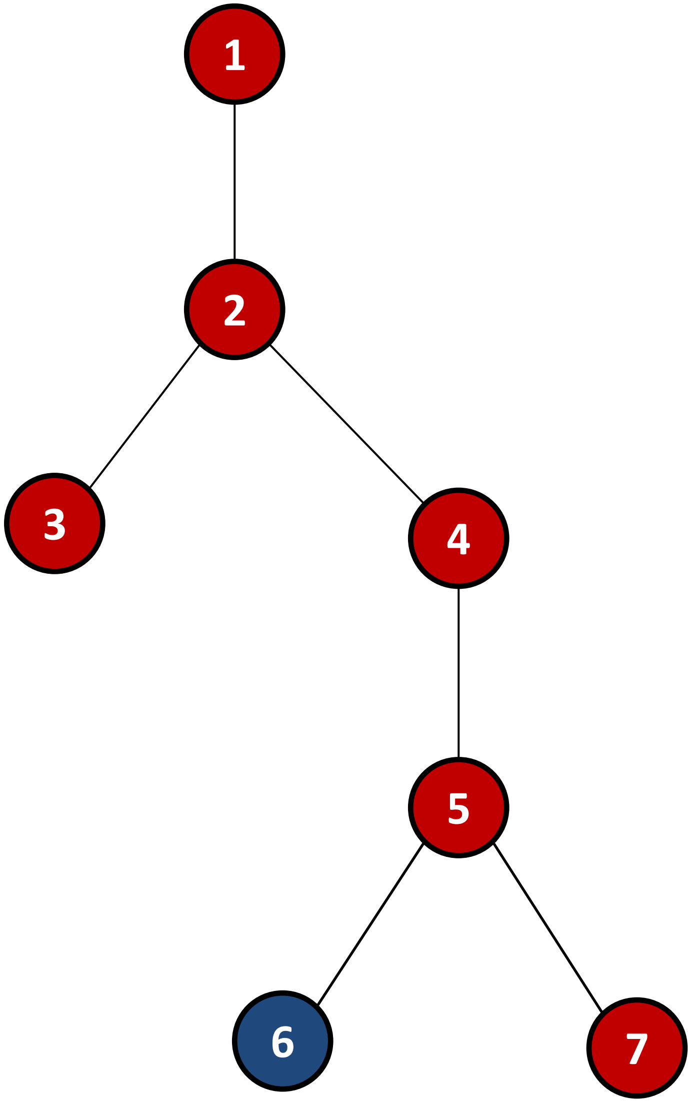
Breadth-first tree walk



Queue: 7

```
BreadthFirst(root) {
  q = empty queue
  q.Enqueue(root)
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      q.Enqueue(c)
  }
}
```

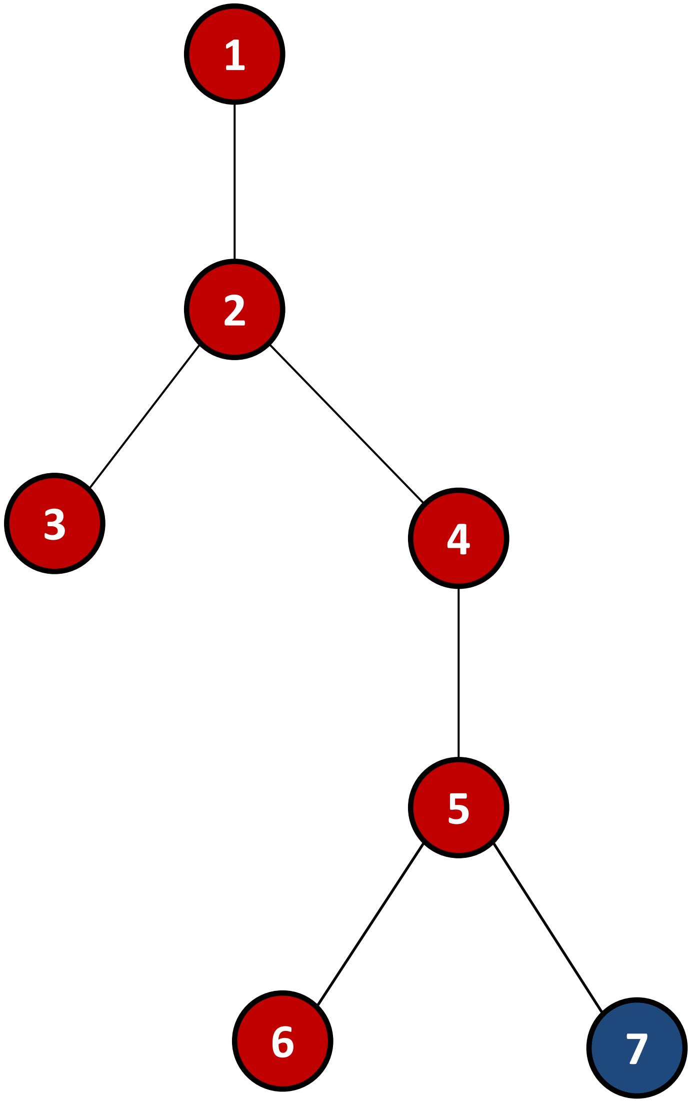
Breadth-first tree walk



Queue: 7

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

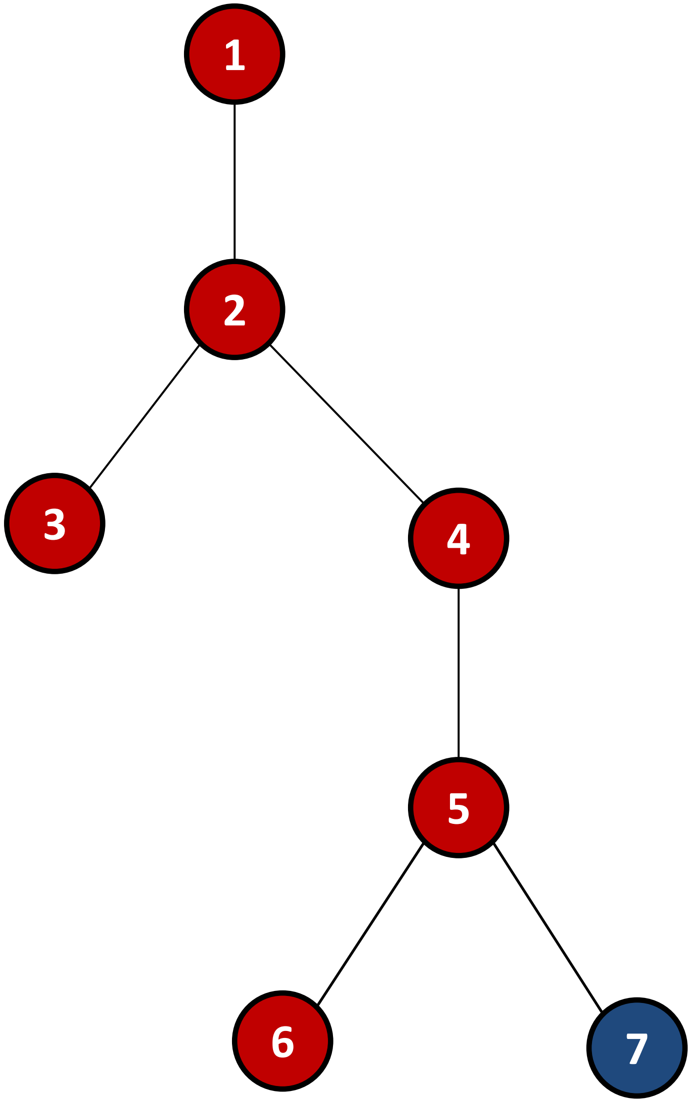
Breadth-first tree walk



Queue:

```
BreadthFirst(root) {
  q = empty queue
  q.Enqueue(root)
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      q.Enqueue(c)
  }
}
```

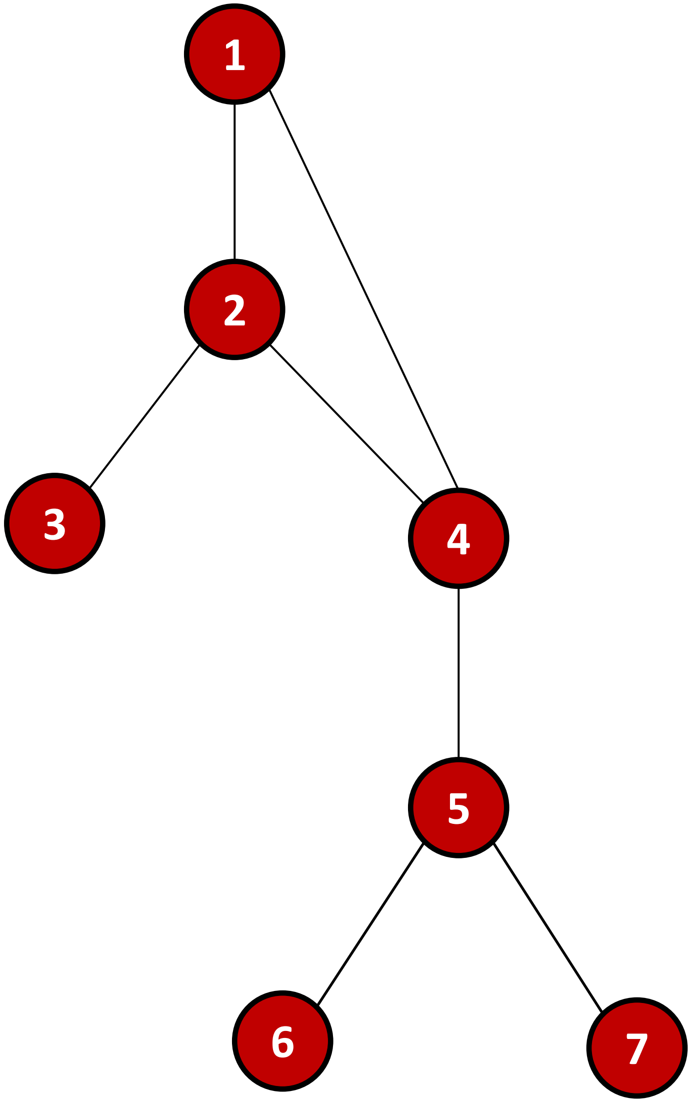
Breadth-first tree walk



Queue: 7

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

Breadth-first ***graph*** walk

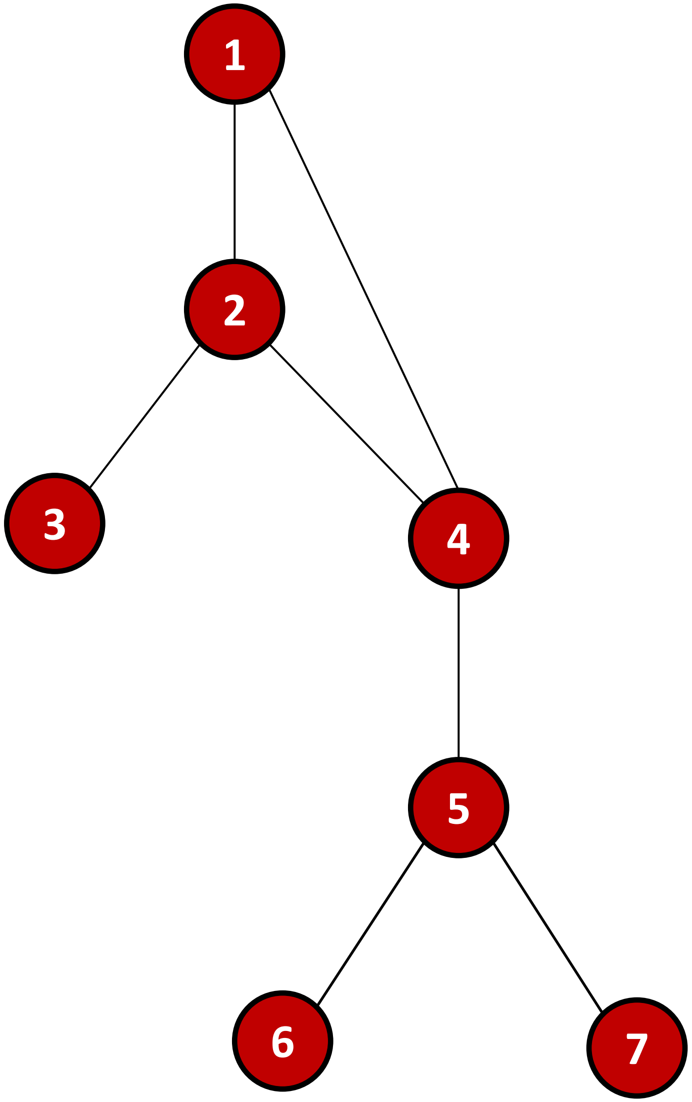


Pseudocode:

```
BreadthFirst(root) {
  q = empty queue
  q.Enqueue(root)
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      q.Enqueue(c)
  }
}
```

Will this code work for a general graph?

Breadth-first ***graph*** walk

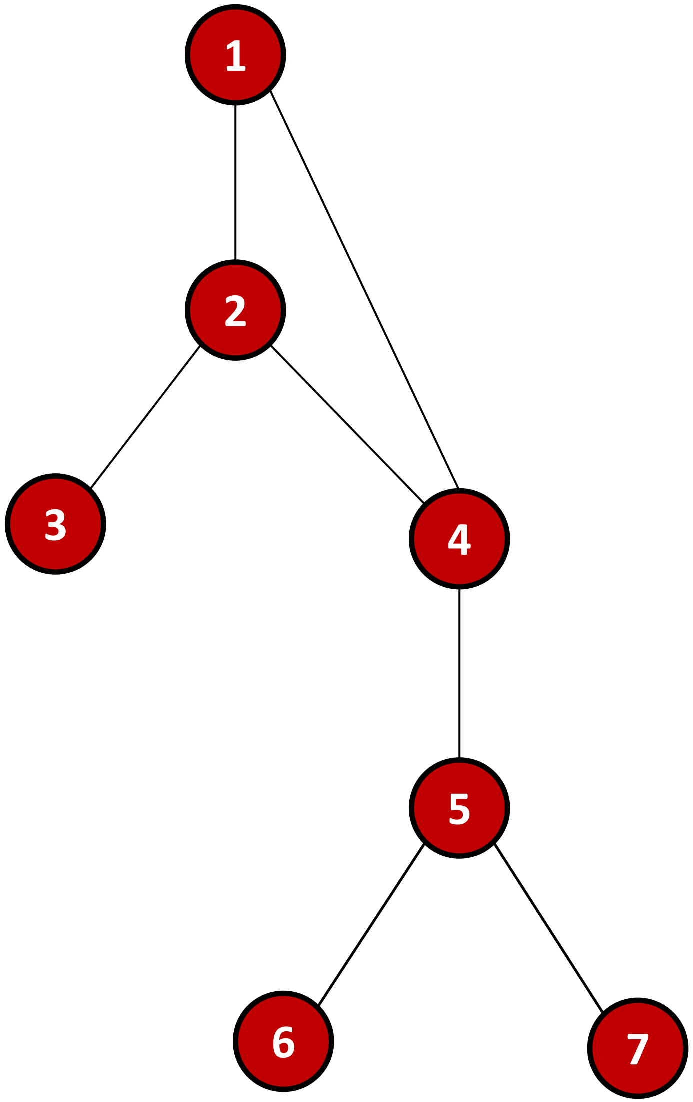


Pseudocode:

```
BreadthFirst(start) {
   q = empty queue
   q.Enqueue(start)
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         q.Enqueue(c)
   }
}
```

Well, first, there are no children *per se*

Breadth-first ***graph*** walk

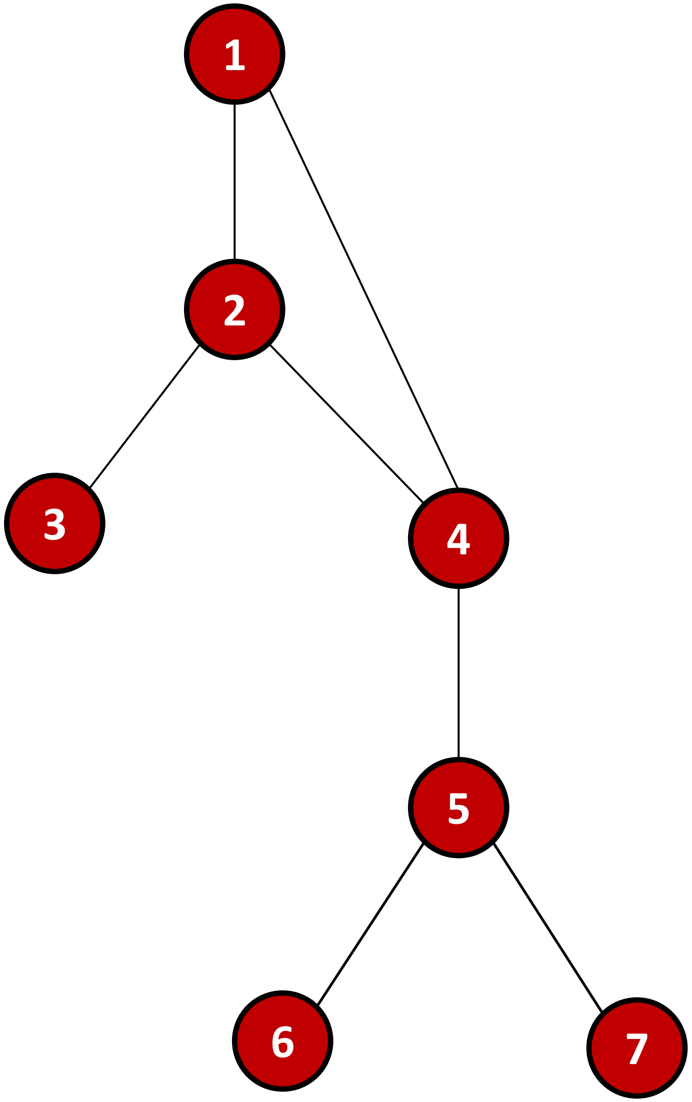


Pseudocode:

```
BreadthFirst(start) {
   q = empty queue
   q.Enqueue(start)
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         q.Enqueue(c)
   }
}
```

What happens if we just change it to neighbor?

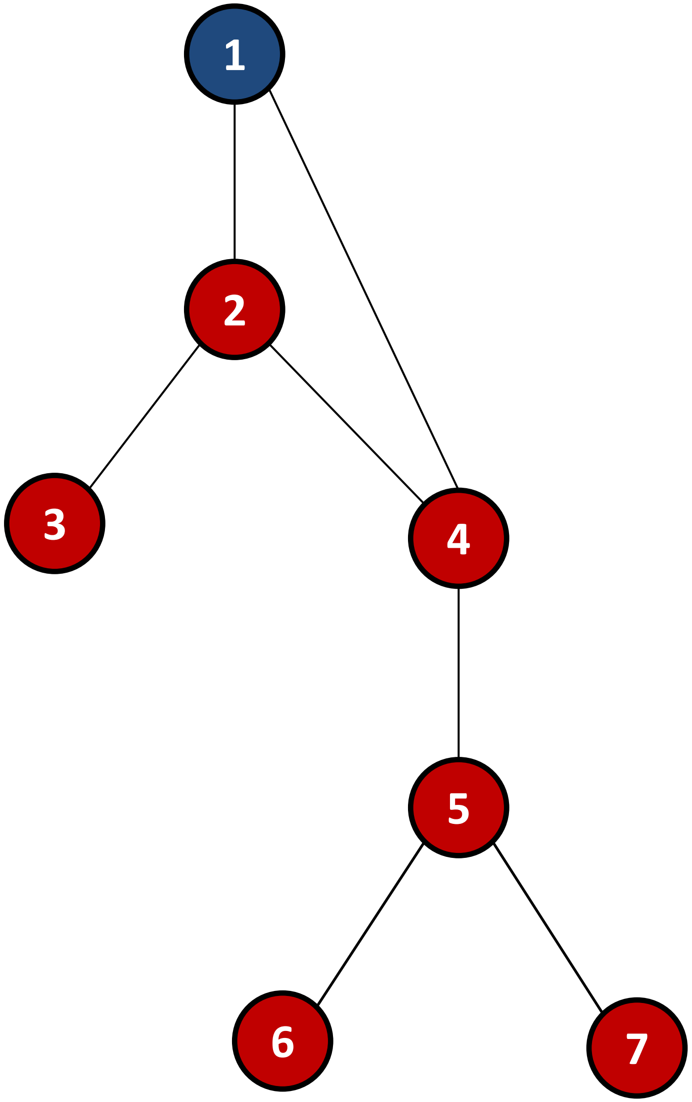
Breadth-first ***graph*** walk



Queue: 1

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      q.Enqueue(c)
  }
}
```

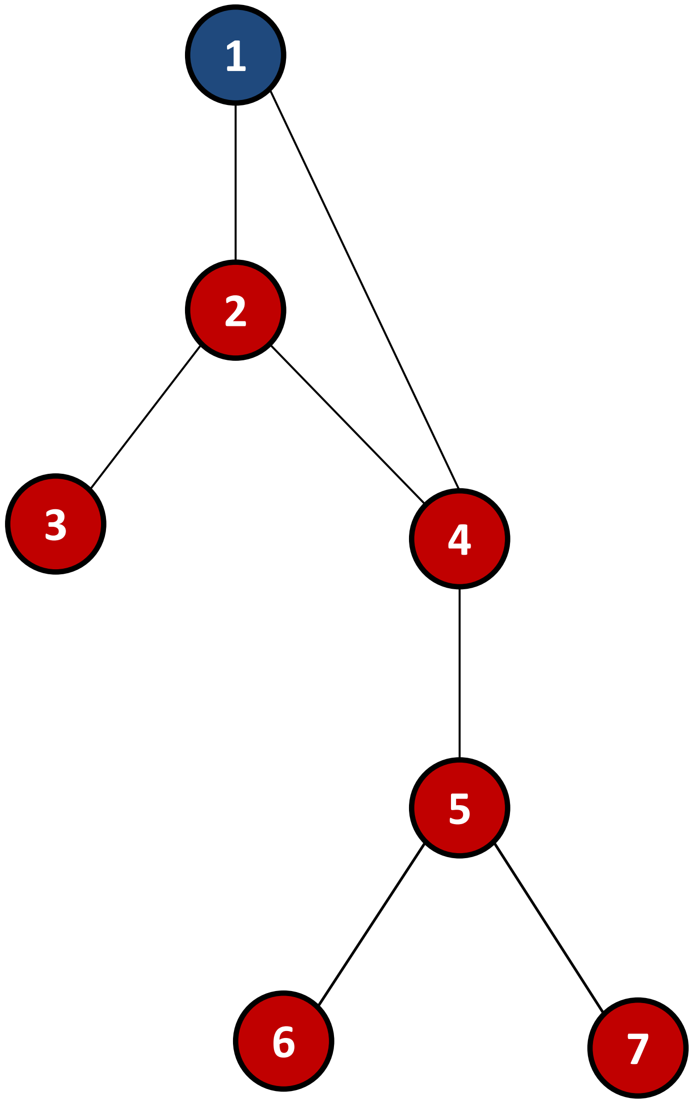
Breadth-first ***graph*** walk



Queue:

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  while q not empty {
    **node = q.Dequeue()**
    for each neighbor c
      q.Enqueue(c)
  }
}
```

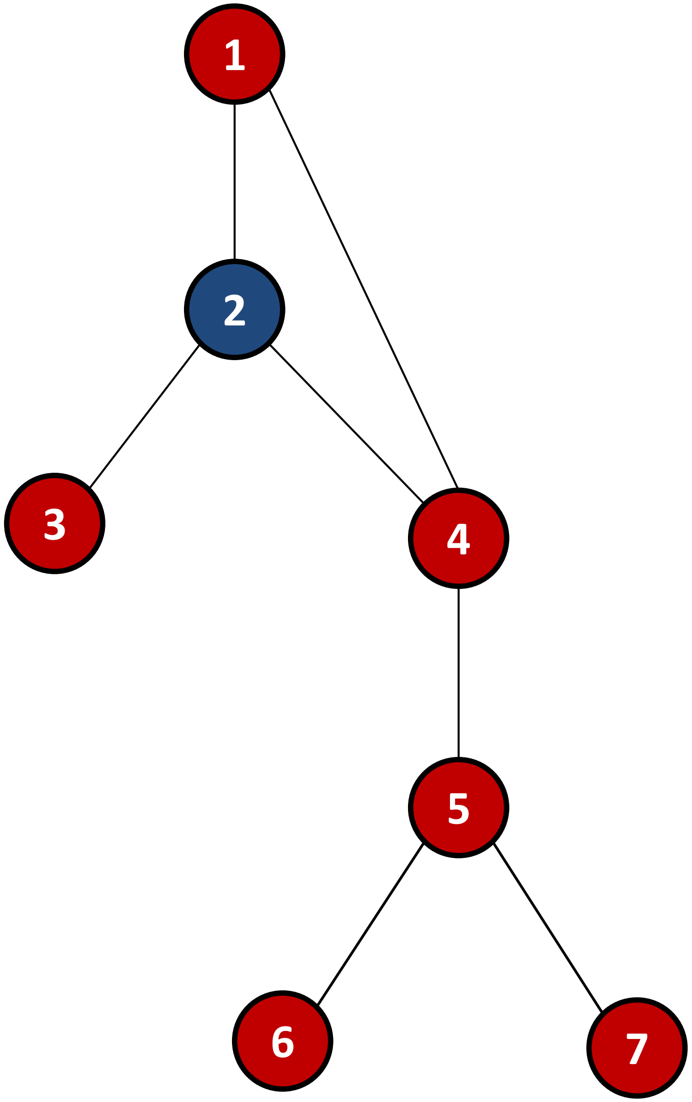
Breadth-first ***graph*** walk



Queue: 2 4

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      q.Enqueue(c)
  }
}
```

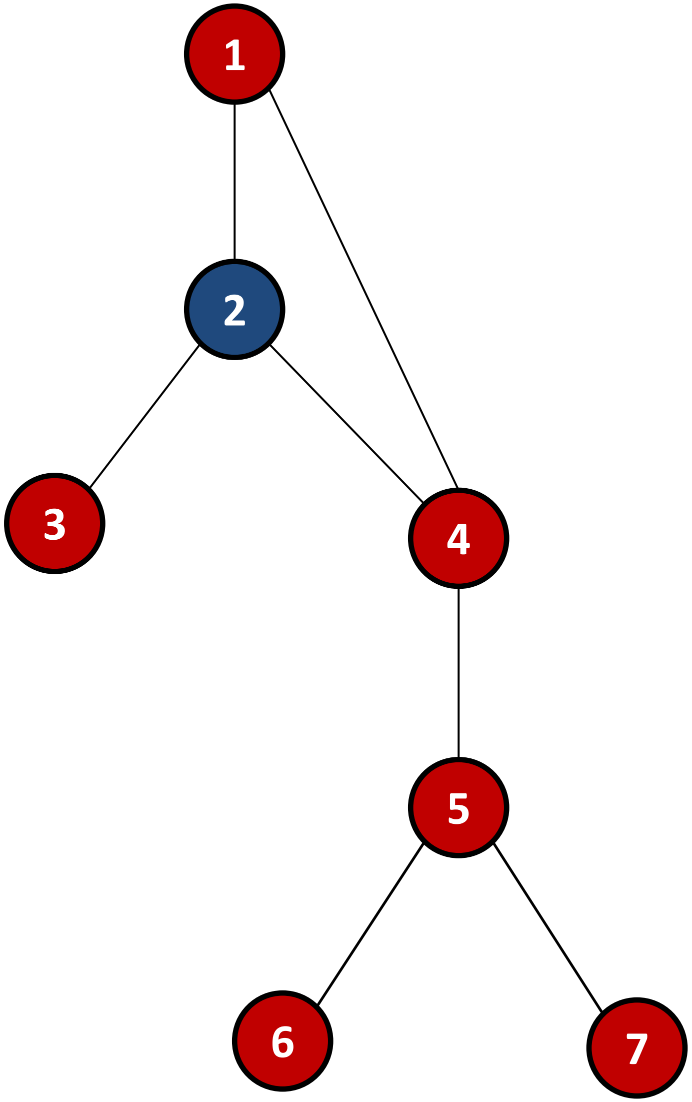
Breadth-first ***graph*** walk



Queue: 4

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      q.Enqueue(c)
  }
}
```

Breadth-first ***graph*** walk

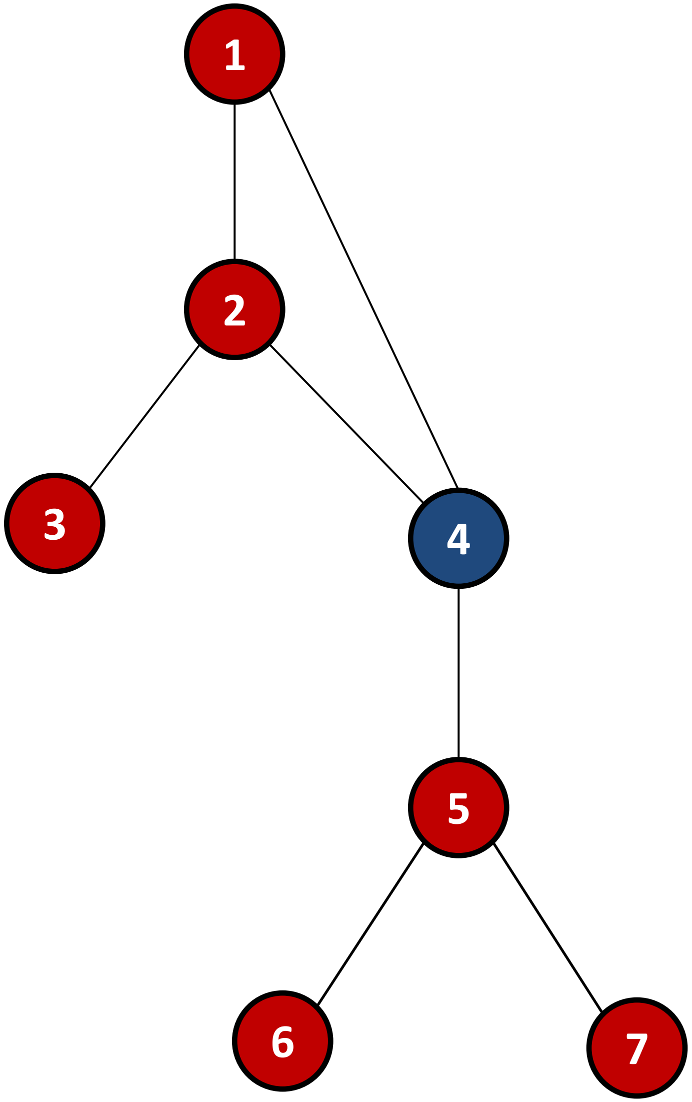


Queue: 4 3 4 1

```
BreadthFirst(start) {
   q = empty queue
   q.Enqueue(start)
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         q.Enqueue(c)
   }
}
```

uh oh …

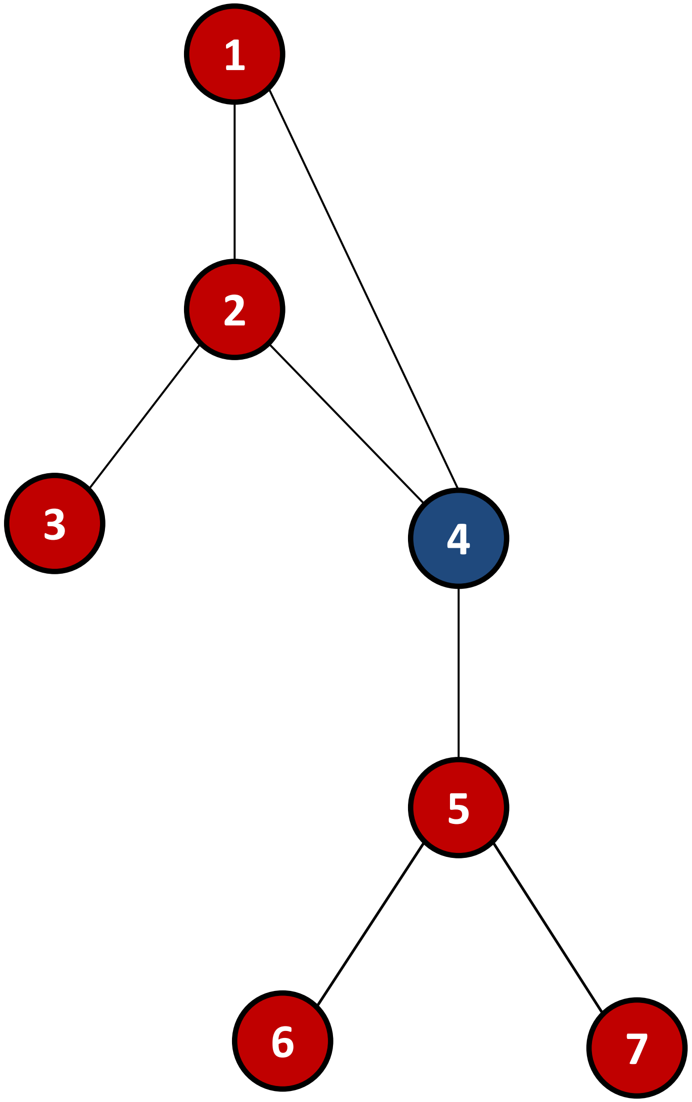
Breadth-first ***graph*** walk



Queue: 3 4 1

```
BreadthFirst(start) {
   q = empty queue
   q.Enqueue(start)
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         q.Enqueue(c)
   }
}
```

Breadth-first ***graph*** walk



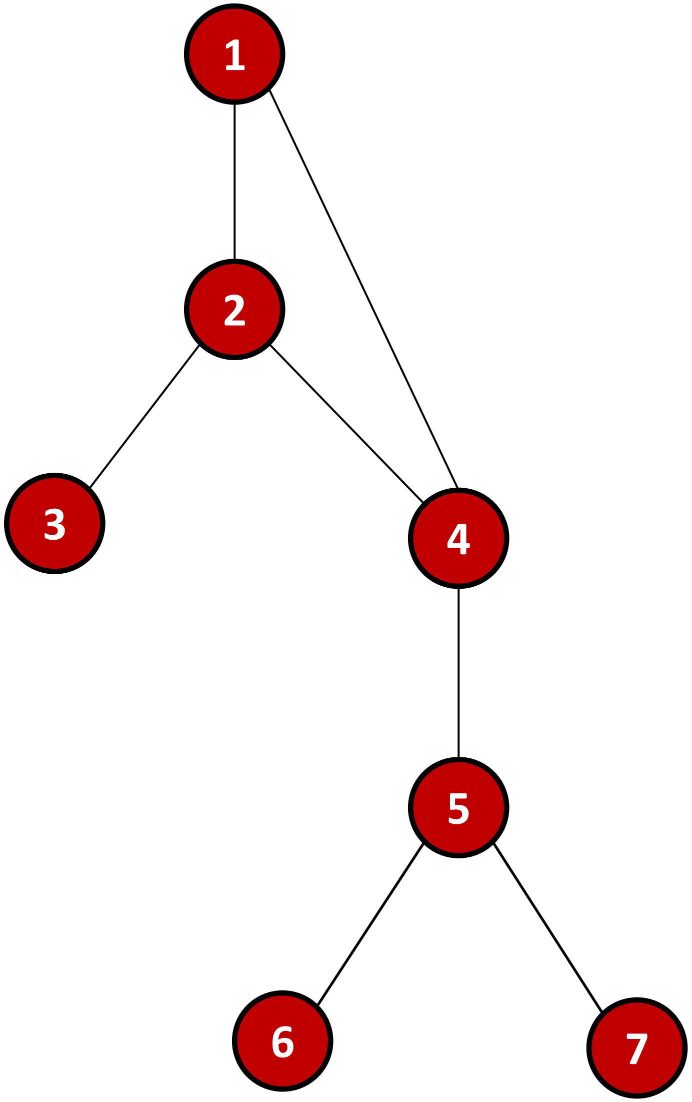
Queue: 3 4 1 1 5

```
BreadthFirst(start) {
   q = empty queue
   q.Enqueue(start)
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         q.Enqueue(c)
   }
}
```

this is not going to end well ...

actually,
it’s not going to end at all …

Breadth-first ***graph*** walk

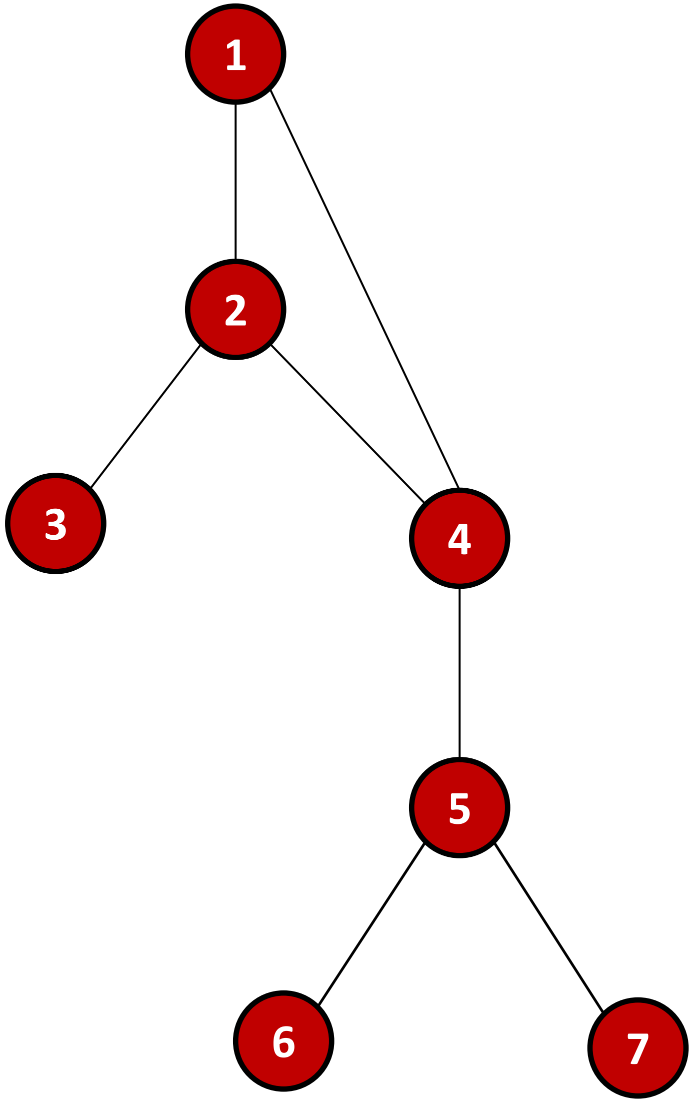


Pseudocode:

```
BreadthFirst(start) {
   q = empty queue
   q.Enqueue(start)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

We need to keep from revisiting nodes

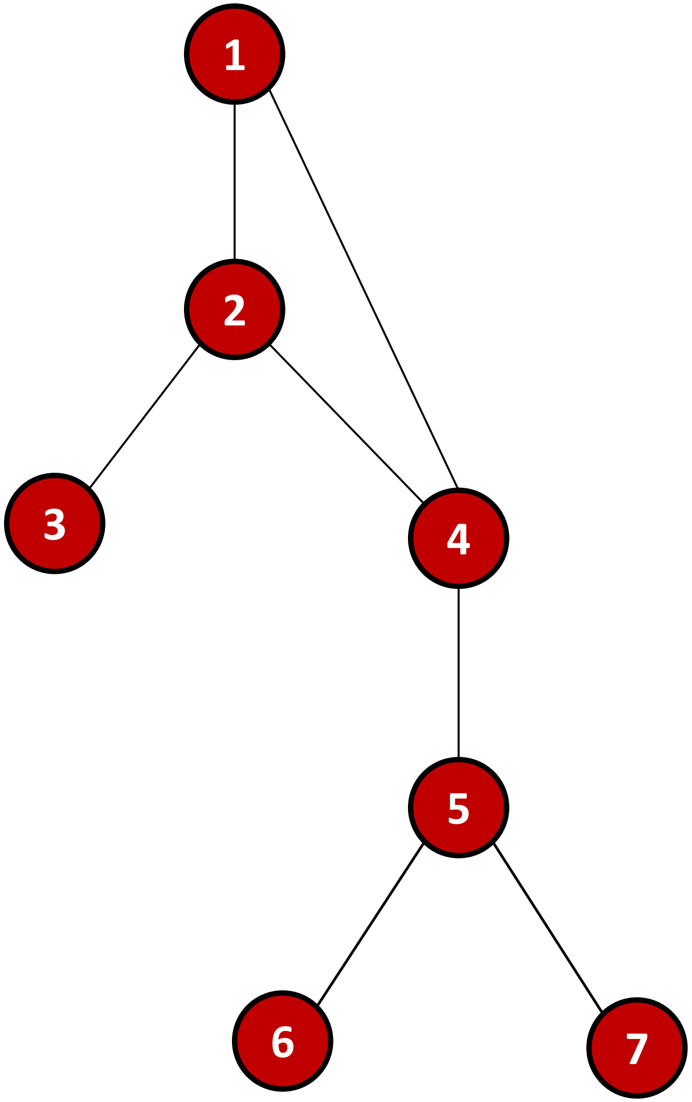
Breadth-first ***graph*** walk



Pseudocode:

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

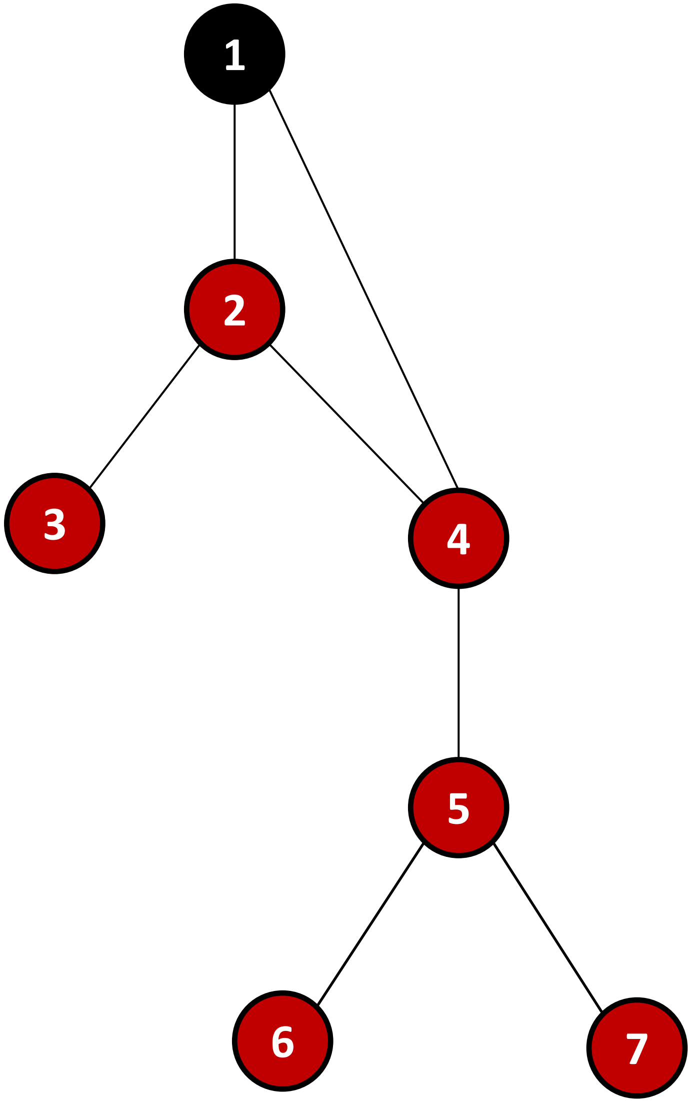
Breadth-first ***graph*** walk



Queue:

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

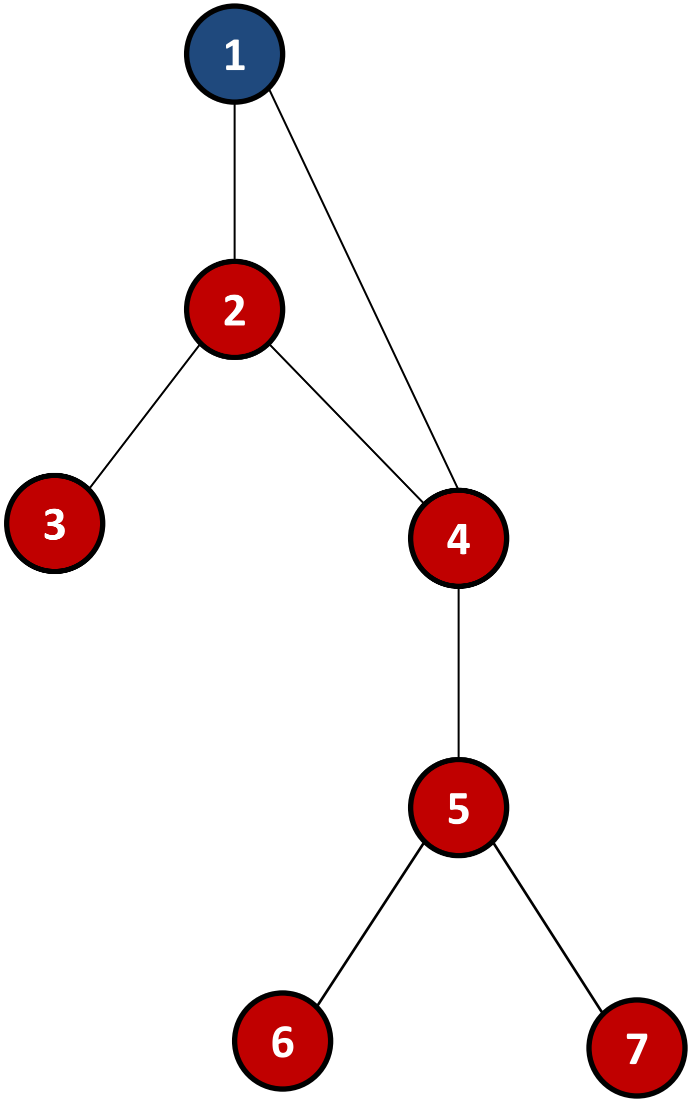
Breadth-first ***graph*** walk



Queue: 1

```
BreadthFirst(start) {
  q = empty queue
  **q.Enqueue(start)**
  **mark start visited**
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

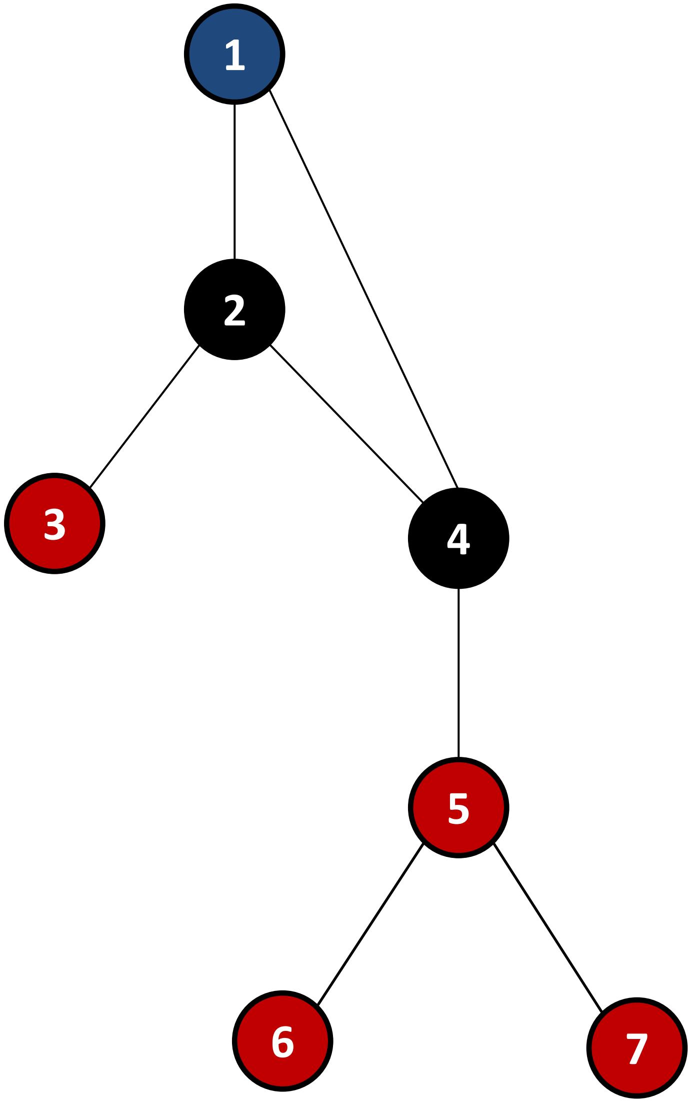
Breadth-first ***graph*** walk



Queue:

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

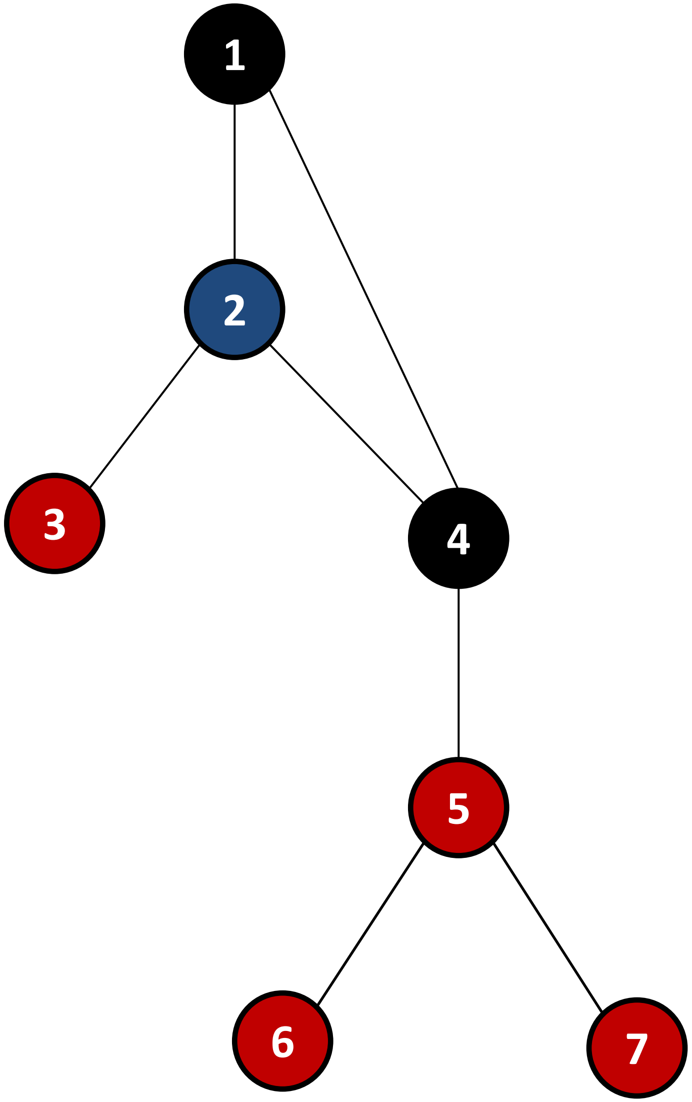
Breadth-first ***graph*** walk



Queue: 2 4

```
BreadthFirst(start) {
   q = empty queue
   q.Enqueue(start)
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         if c not visited already
            q.Enqueue(c)
            mark c visited
   }
}
```

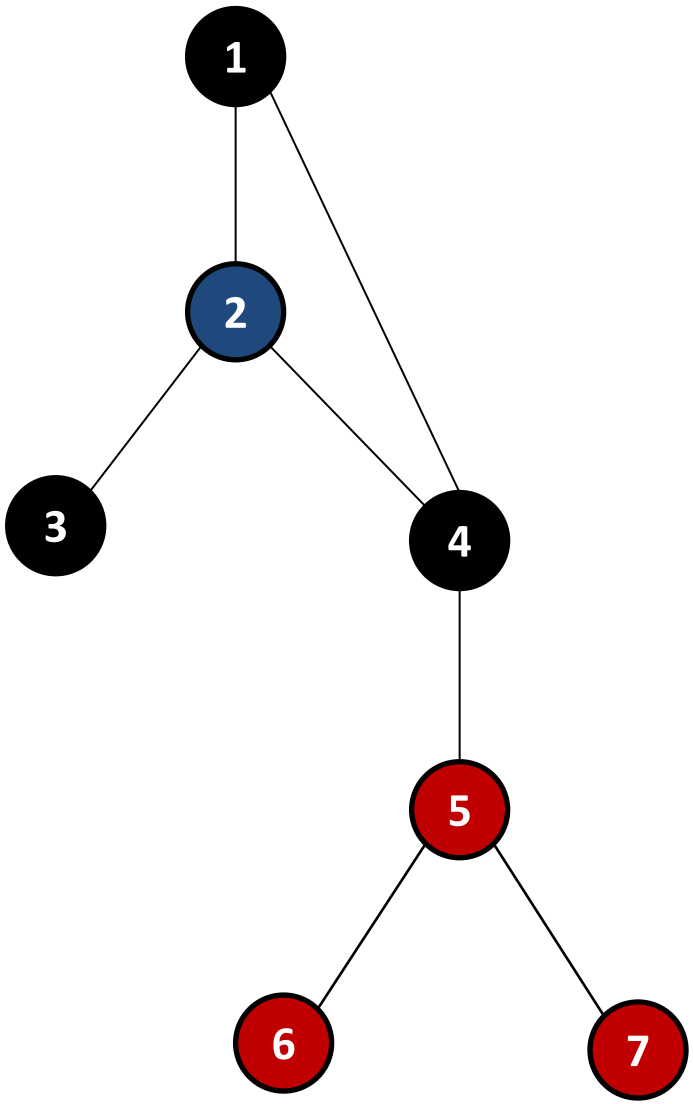
Breadth-first *graph* walk



Queue: 4

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

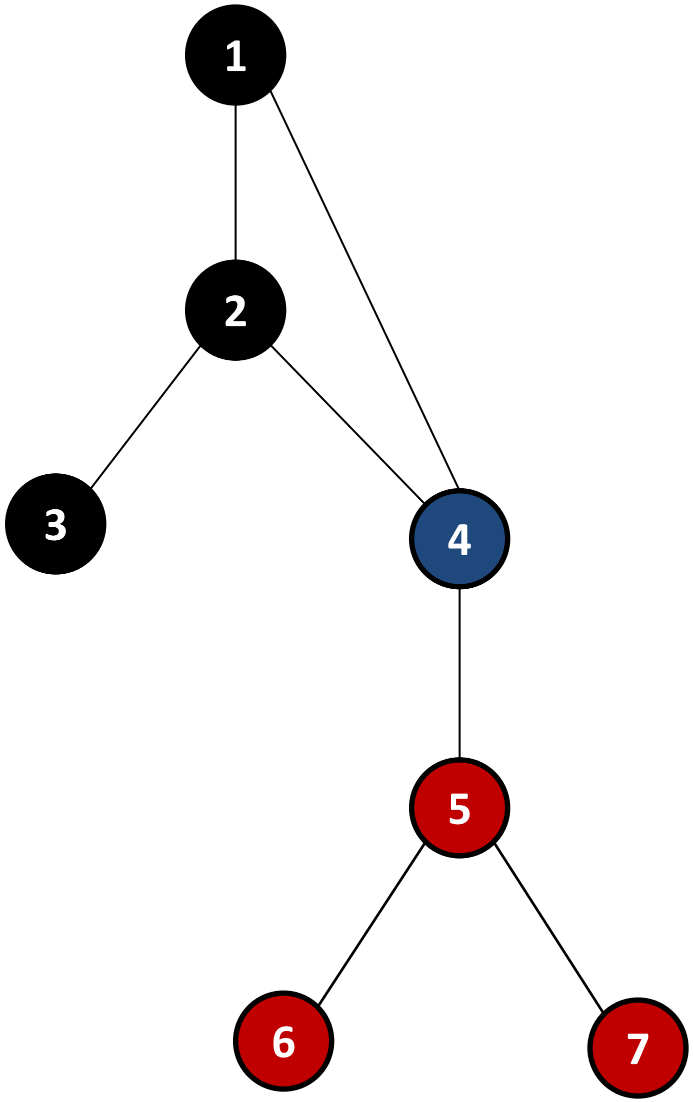
Breadth-first *graph* walk



Queue: 4 3

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

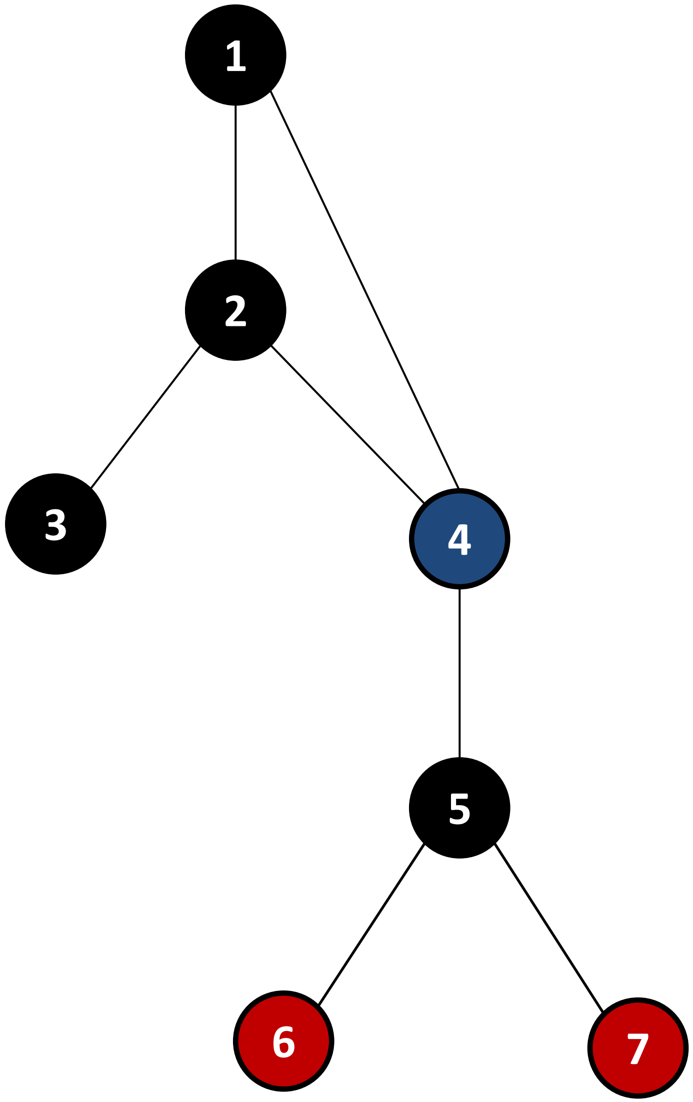
Breadth-first *graph* walk



Queue: 3

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    **node = q.Dequeue()**
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

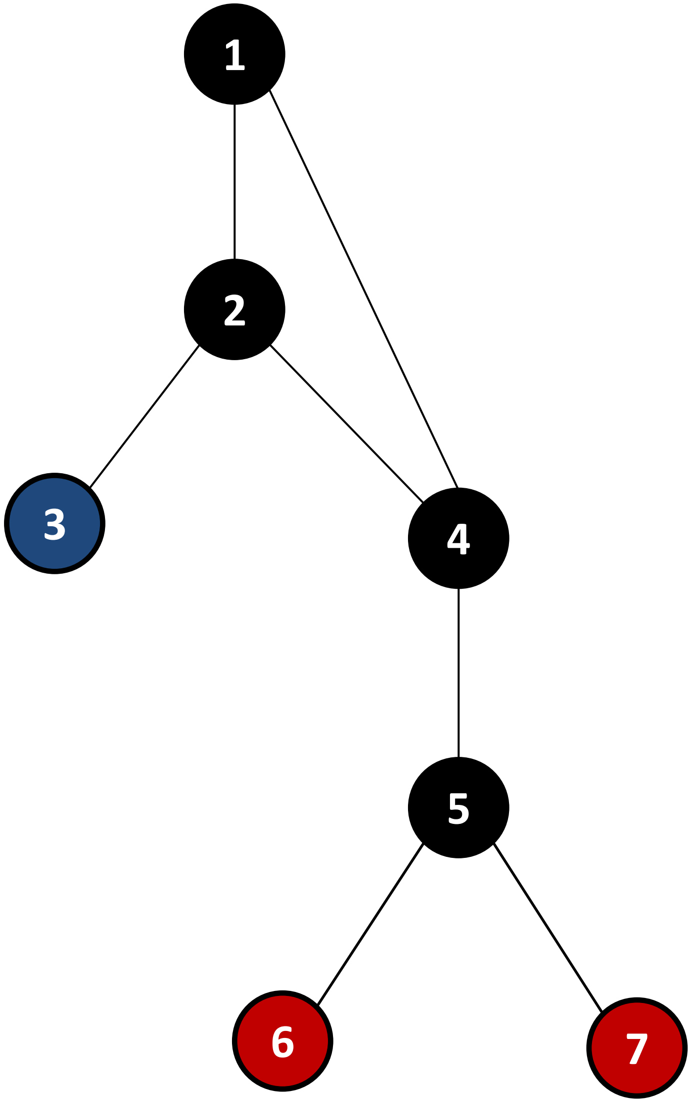
Breadth-first ***graph*** walk



Queue: 3 5

```
BreadthFirst(start) {
   q = empty queue
   q.Enqueue(start)
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         if c not visited already
            q.Enqueue(c)
            mark c visited
   }
}
```

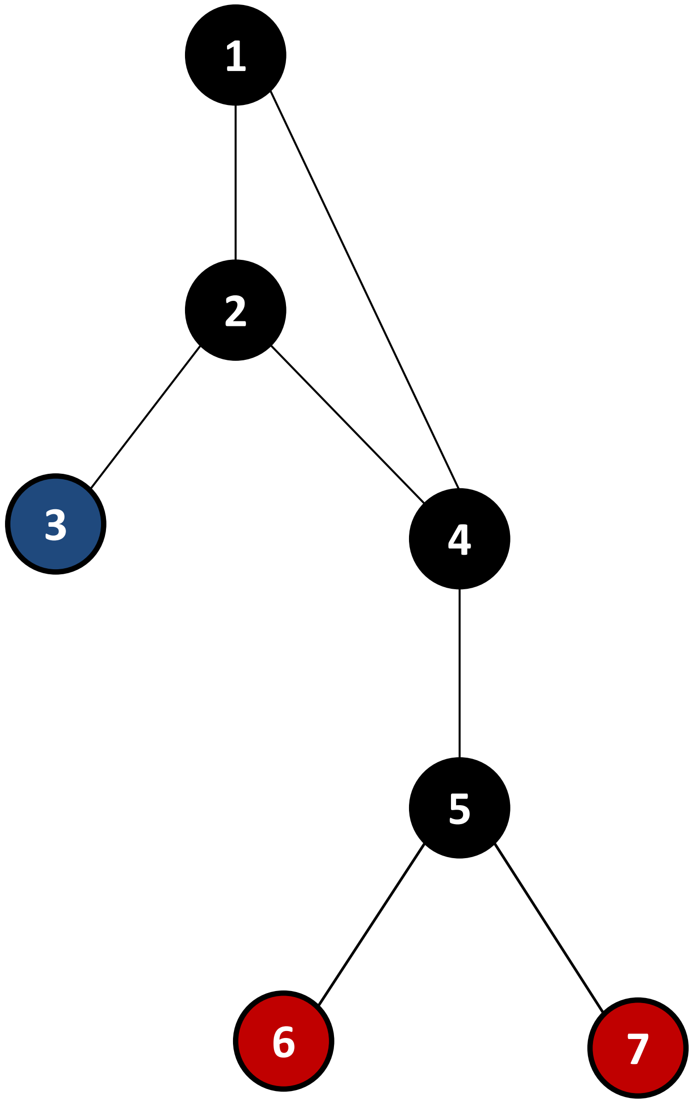
Breadth-first ***graph*** walk



Queue: 5

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

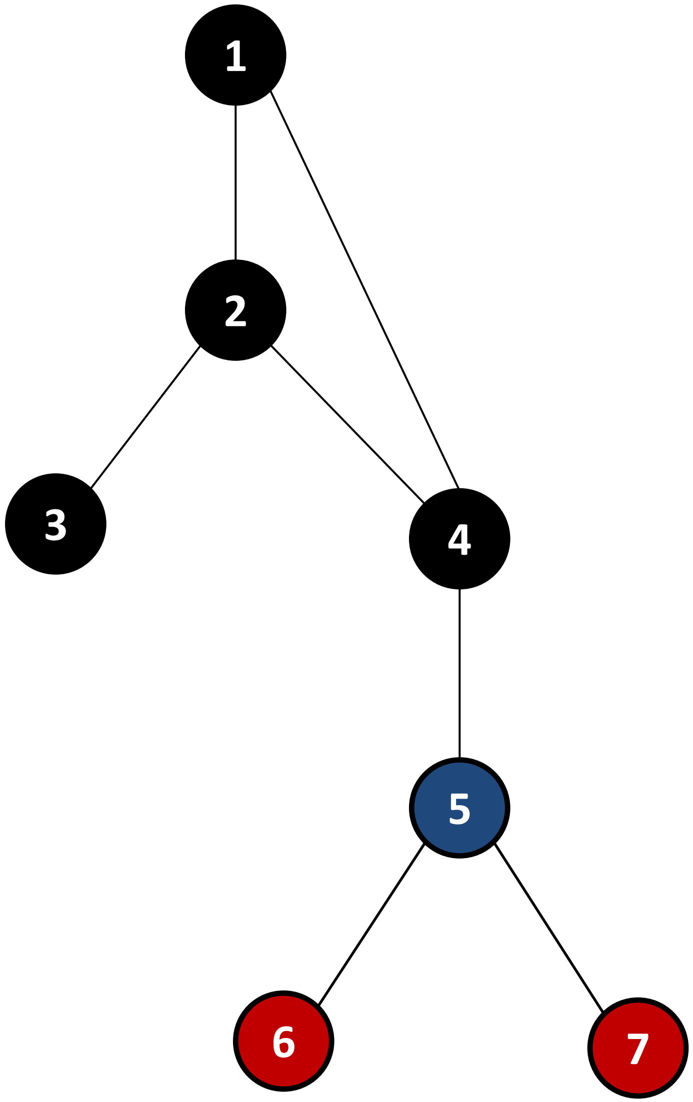
Breadth-first *graph* walk



Queue: 5

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

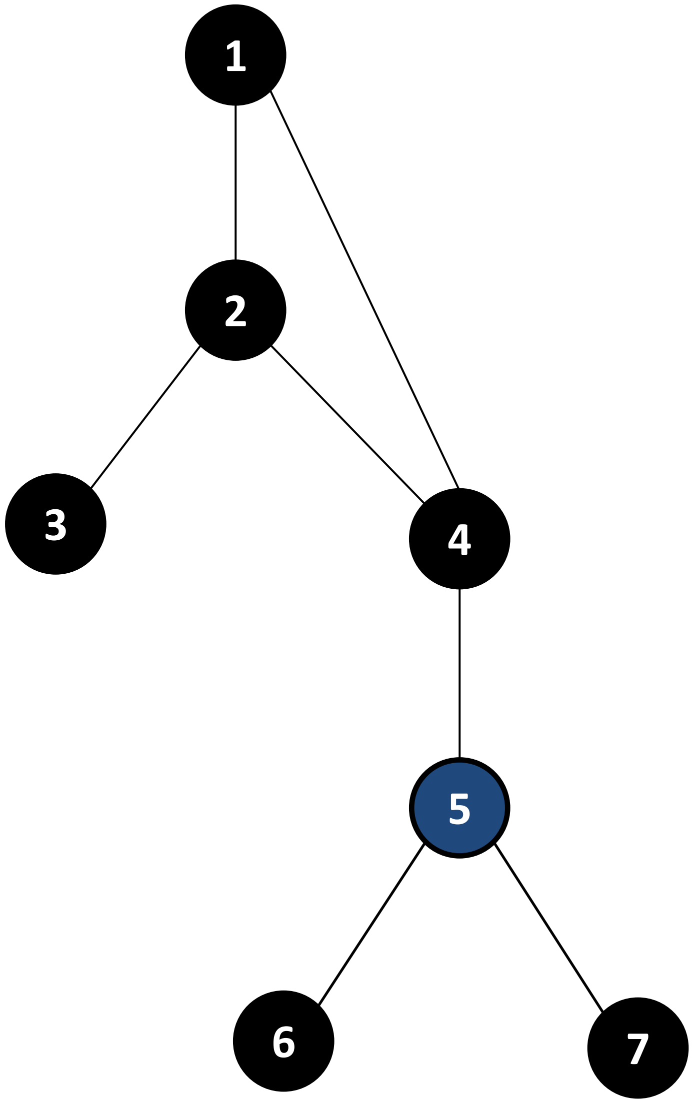
Breadth-first *graph* walk



Queue:

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

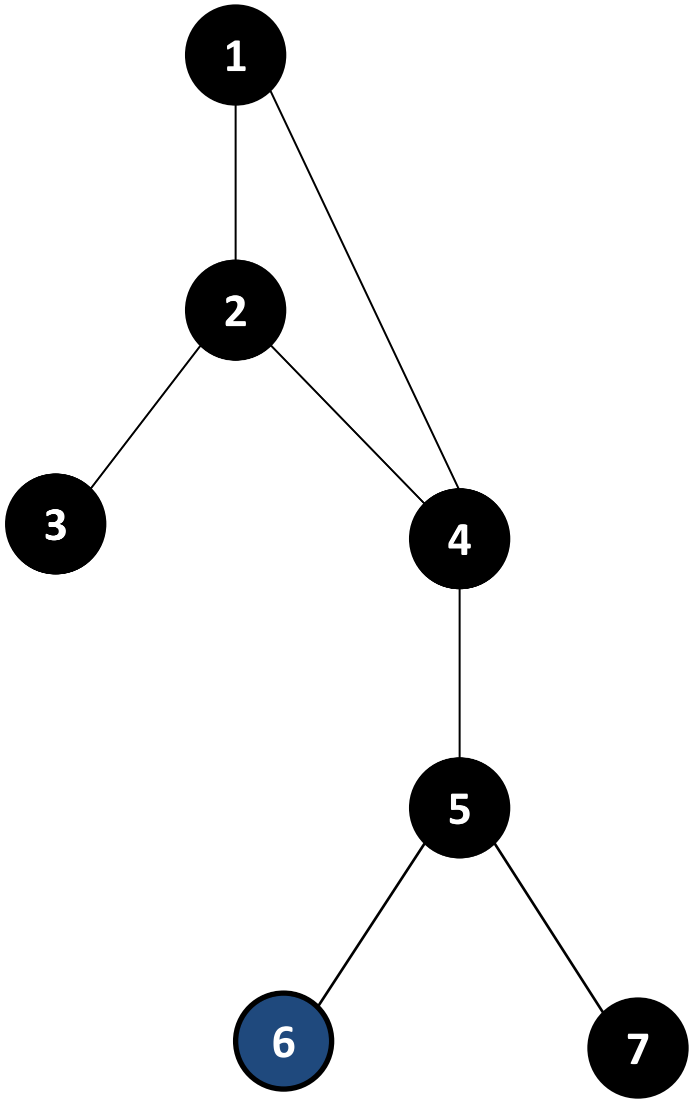
Breadth-first *graph* walk



Queue: 6 7

```
BreadthFirst(start) {
   q = empty queue
   q.Enqueue(start)
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         if c not visited already
            q.Enqueue(c)
            mark c visited
   }
}
```

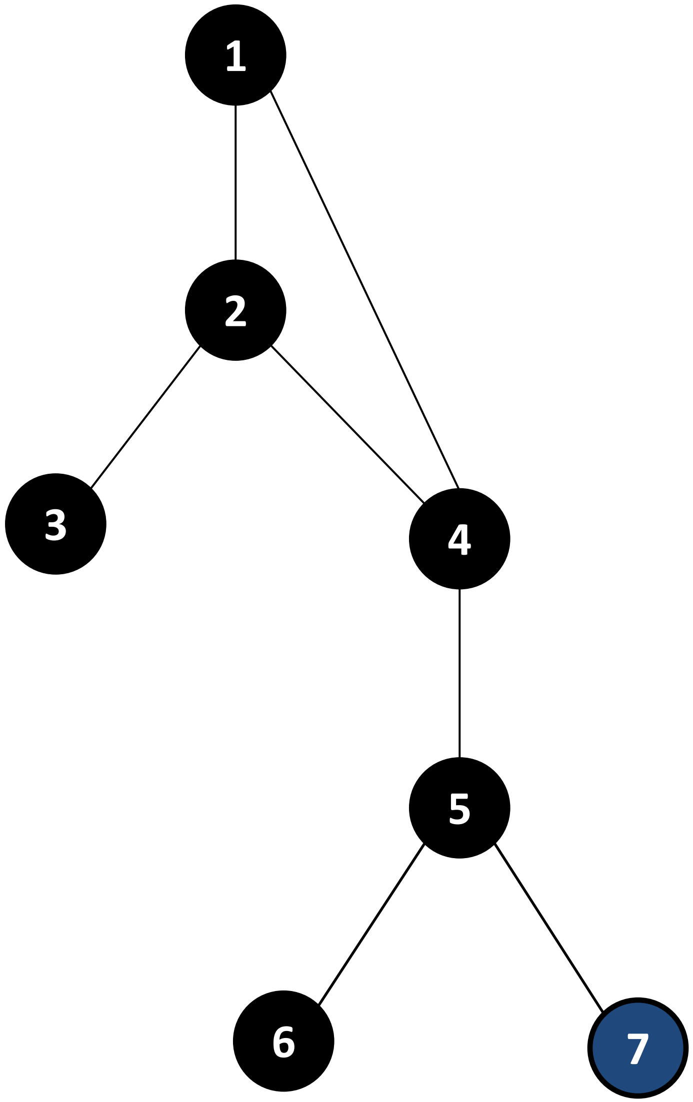
Breadth-first ***graph*** walk



Queue: 7

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

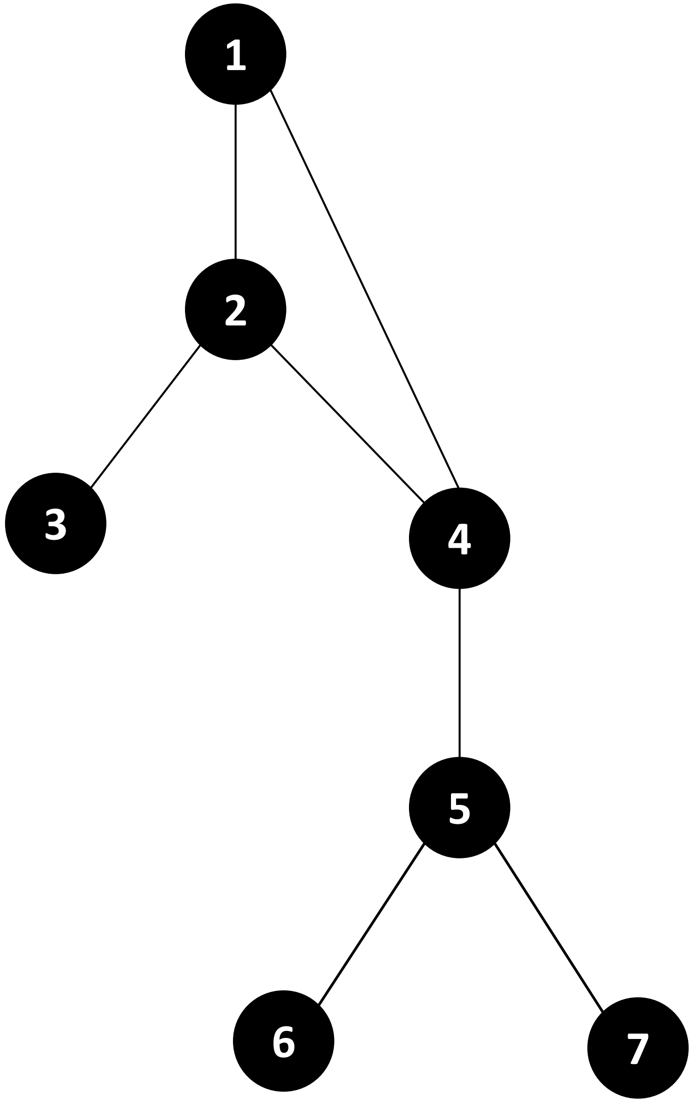
Breadth-first ***graph*** walk



Queue: 7

```
BreadthFirst(start) {
  q = empty queue
  q.Enqueue(start)
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each child c of node
      if c not visited already
        q.Enqueue(c)
        mark c visited
  }
}
```

Breadth-first ***graph*** walk



- Tracking which nodes have been visited is sufficient to fix the problem

- Great! How do we keep track of which nodes have been visited

Visit information in the nodes

- One solution is to **add a field** to the node structure
- Advantages
  - **Fast** and efficient
- Disadvantages
  - You need to **know in advance** that you'll be walking the graph
  - Doesn't work well if you're using **someone else's graph implementation**
  - Uses memory even when you aren't walking the graph

```
class GraphNode {
  GraphNode[ ] adjacent;
  bool visited = false;
}
```

```
node.visited = true;
```

Parallel array structure

- Use a **separate array**
  - Indexed by node number
- Advantages
  - **Fast**
  - Can **deallocate** the array when it's not being used
- Disadvantages
  - Requires nodes to be **numbered**
    - But you often do that for other reasons anyway.

bool[ ] visited;

visited[node.number] = true;

Hash table
(or other dictionary structure)

- If nothing else, you can use a hash table
  - Use nodes as keys

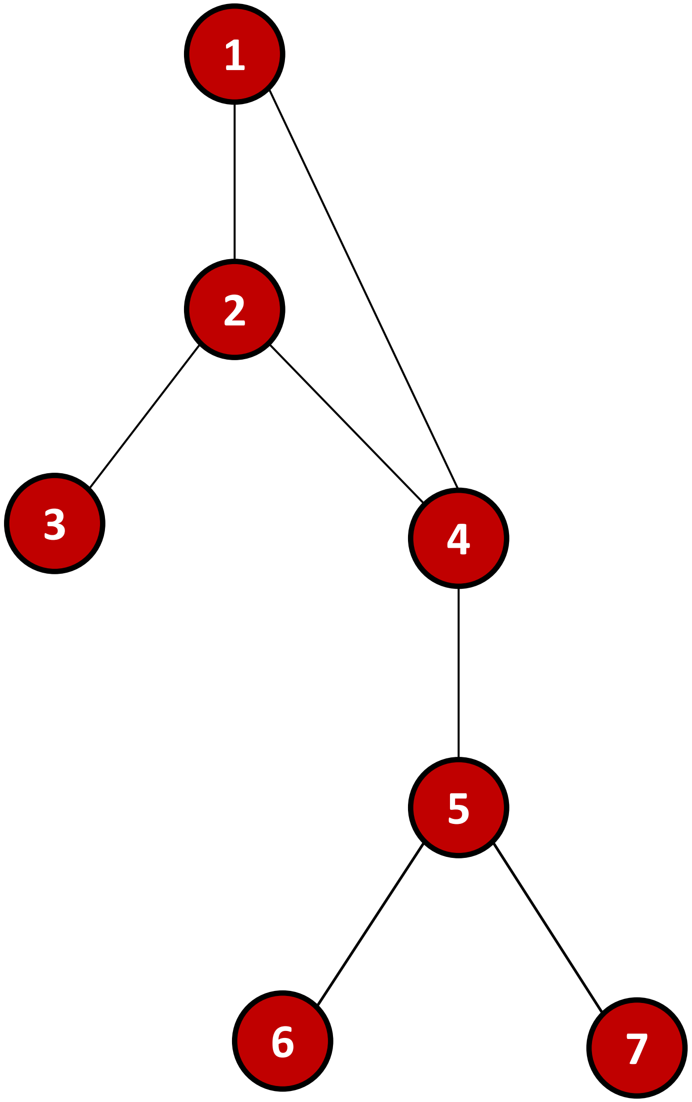
Hashtable visited
= new Hashtable();

- Advantages
  - **Easy**
  - Works with **any node structure**

visited.Store(node, true);

- Disadvantages
  - **Slower** than other approaches

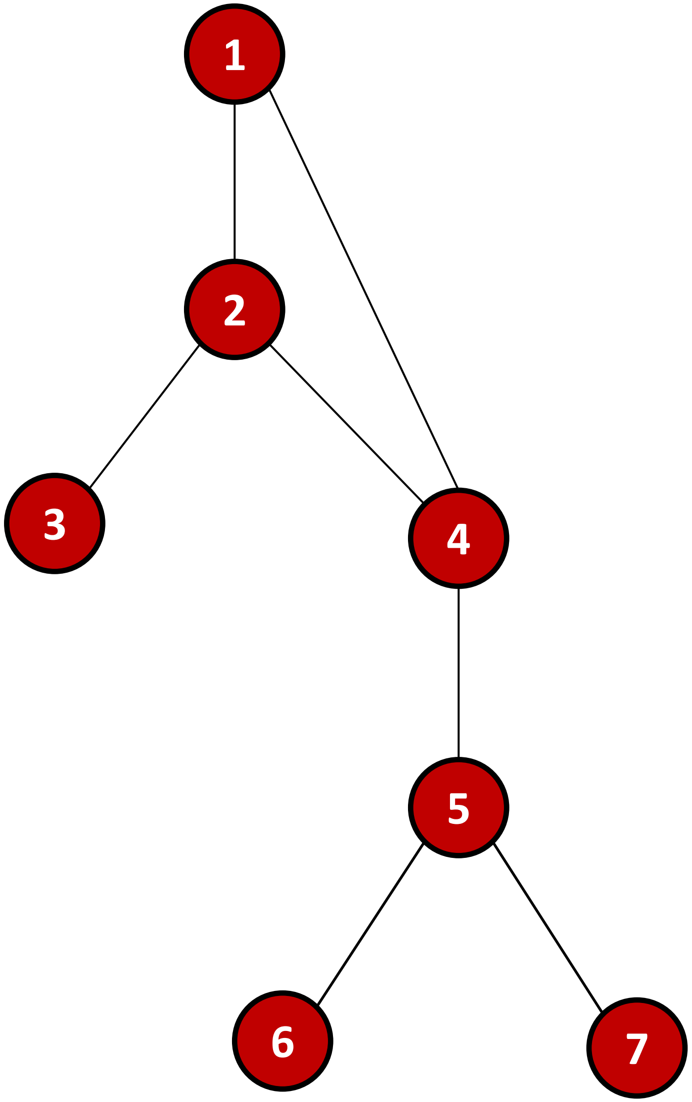
Breadth-first search (from CLR)



- Breadth-first walks are often called **breadth-first search**es

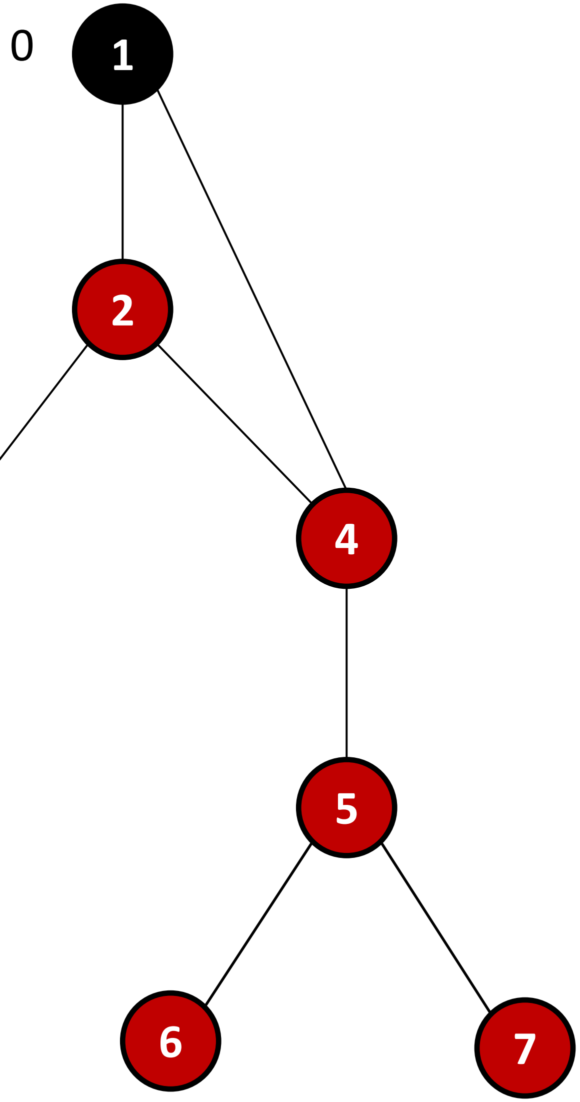
- The CLR book defines a **fancier version of BFS**
- It tracks **additional information**
  - **Distances** of nodes from the start node
  - **Predecessors** of nodes in the walk
    - Which of its neighbors was searched first

Breadth-first search (from CLR)



```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

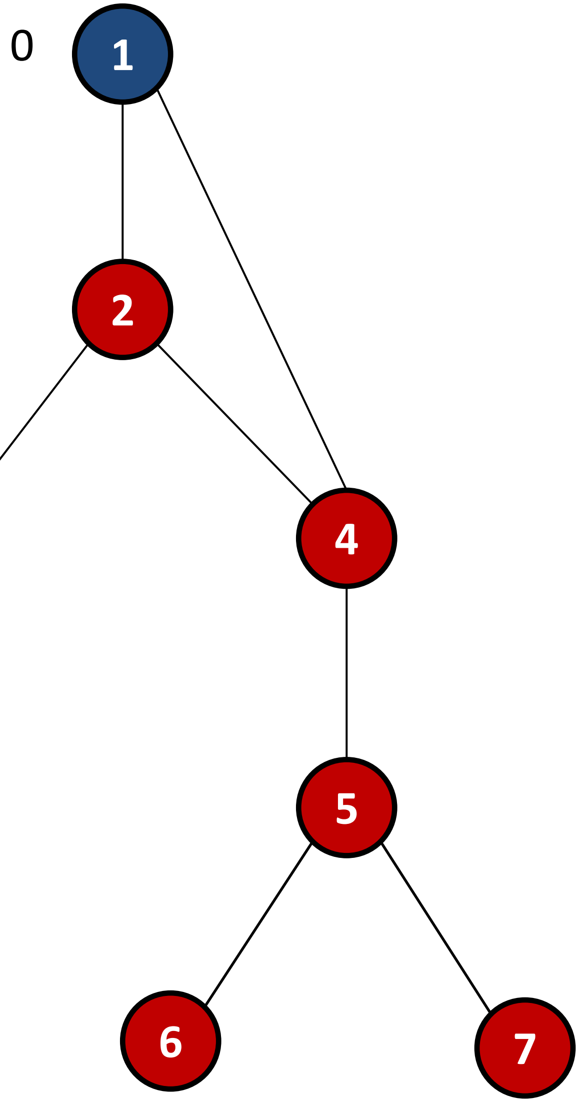
Breadth-first search (from CLR)



Queue: 1

```
BreadthFirstSearch(start) {
   q = empty queue
   q.Enqueue(start)
   start.distance = 0
   start.predecessor = null
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         if c not visited {
            q.Enqueue(c)
            c.distance = node.distance+1
            c.predecessor = node
         }
   }
}
```

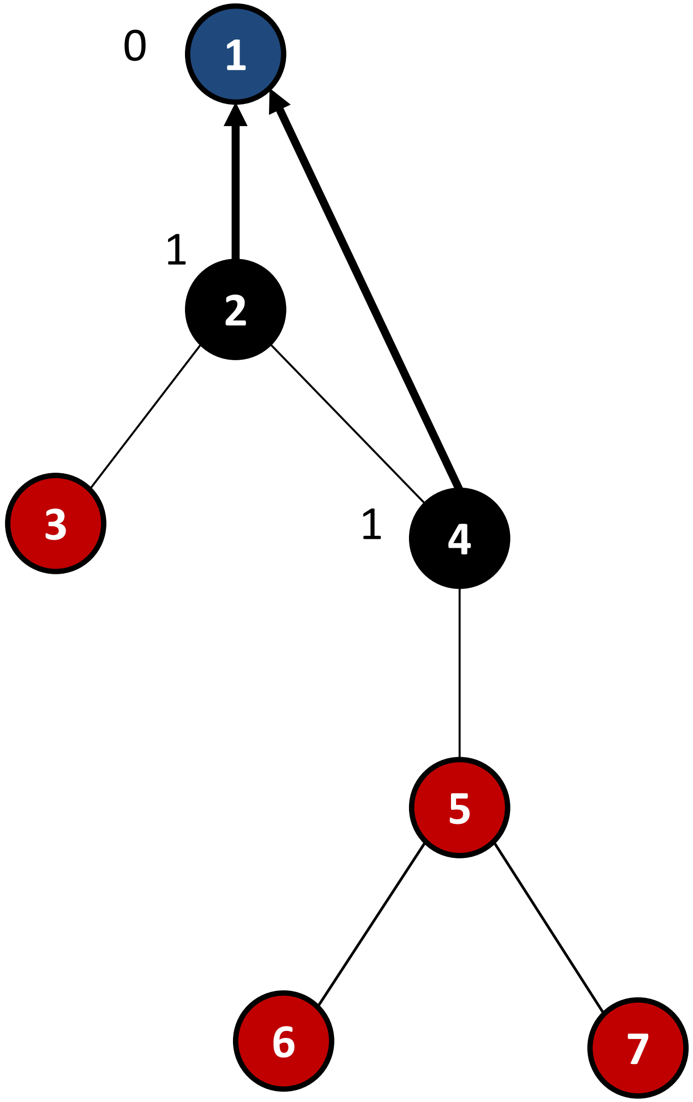
Breadth-first search (from CLR)



Queue:

```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

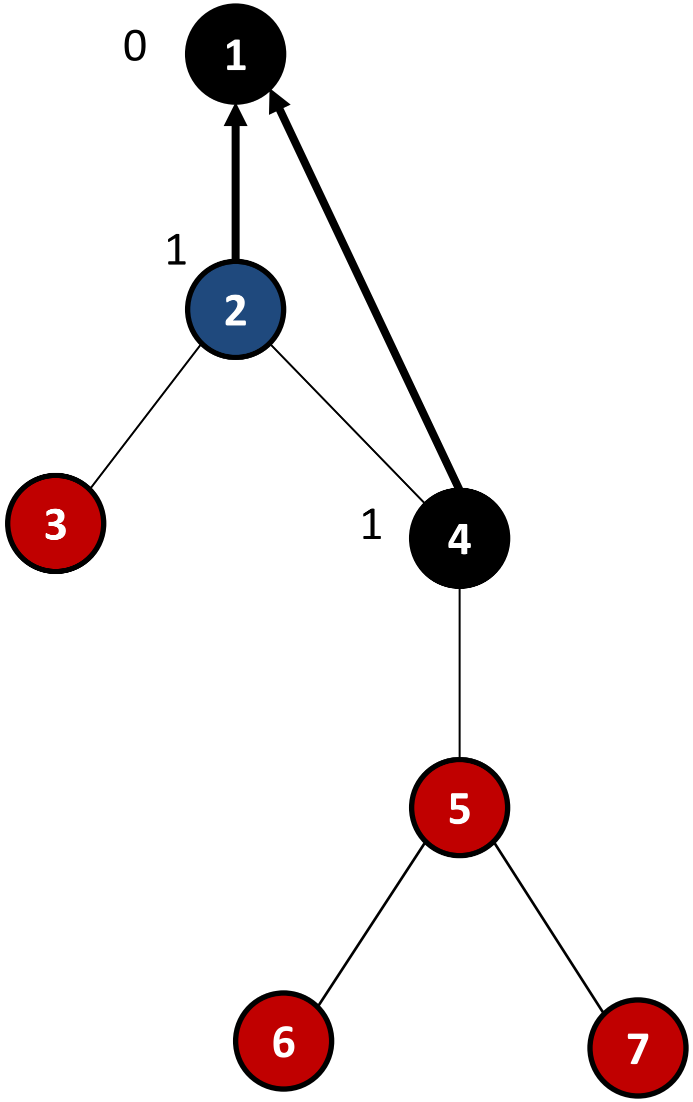
Breadth-first search (from CLR)



Queue: 2 4

```
BreadthFirstSearch(start) {
   q = empty queue
   q.Enqueue(start)
   start.distance = 0
   start.predecessor = null
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         if c not visited {
            q.Enqueue(c)
            c.distance = node.distance+1
            c.predecessor = node
         }
   }
}
```

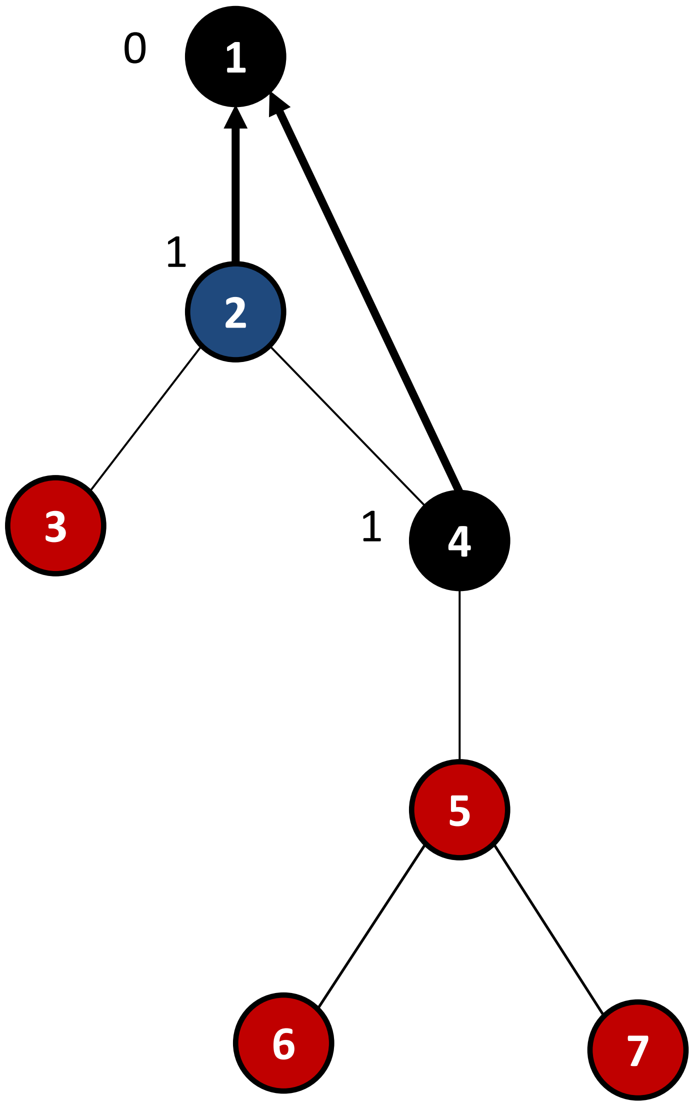
Breadth-first search (from CLR)



Queue: 4

```
BreadthFirstSearch(start) {
   q = empty queue
   q.Enqueue(start)
   start.distance = 0
   start.predecessor = null
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         if c not visited {
            q.Enqueue(c)
            c.distance = node.distance+1
            c.predecessor = node
         }
   }
}
```

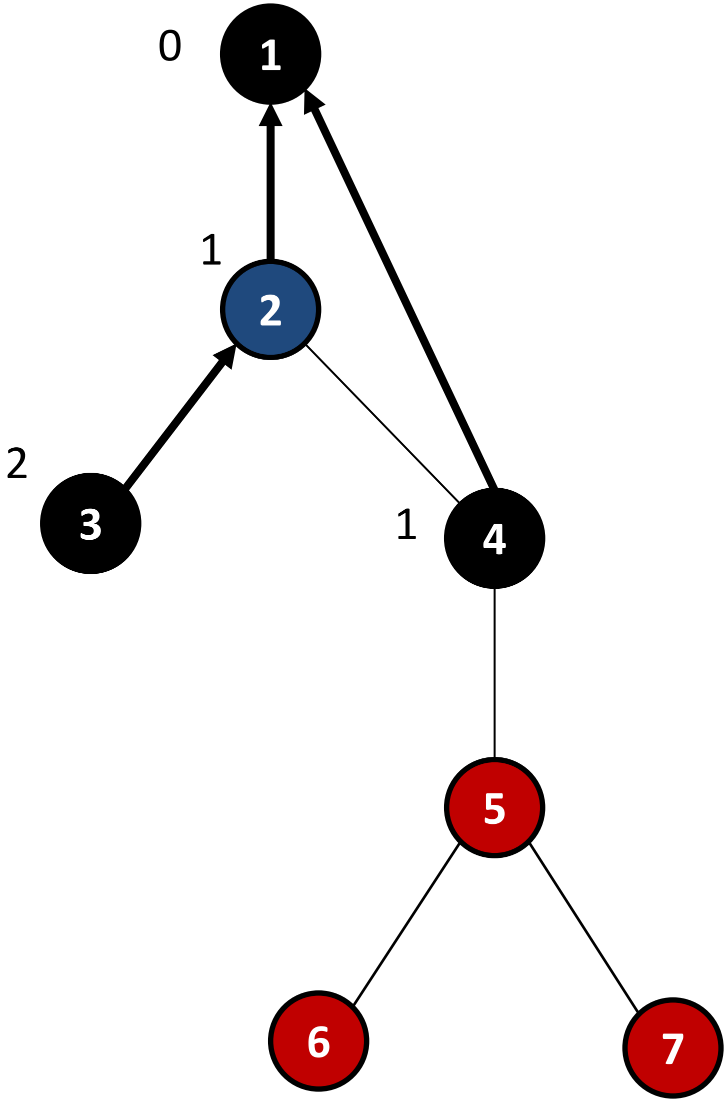
Breadth-first search (from CLR)



Queue: 4

```
BreadthFirstSearch(start) {
   q = empty queue
   q.Enqueue(start)
   start.distance = 0
   start.predecessor = null
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         if c not visited {
            q.Enqueue(c)
            c.distance = node.distance+1
            c.predecessor = node
         }
   }
}
```

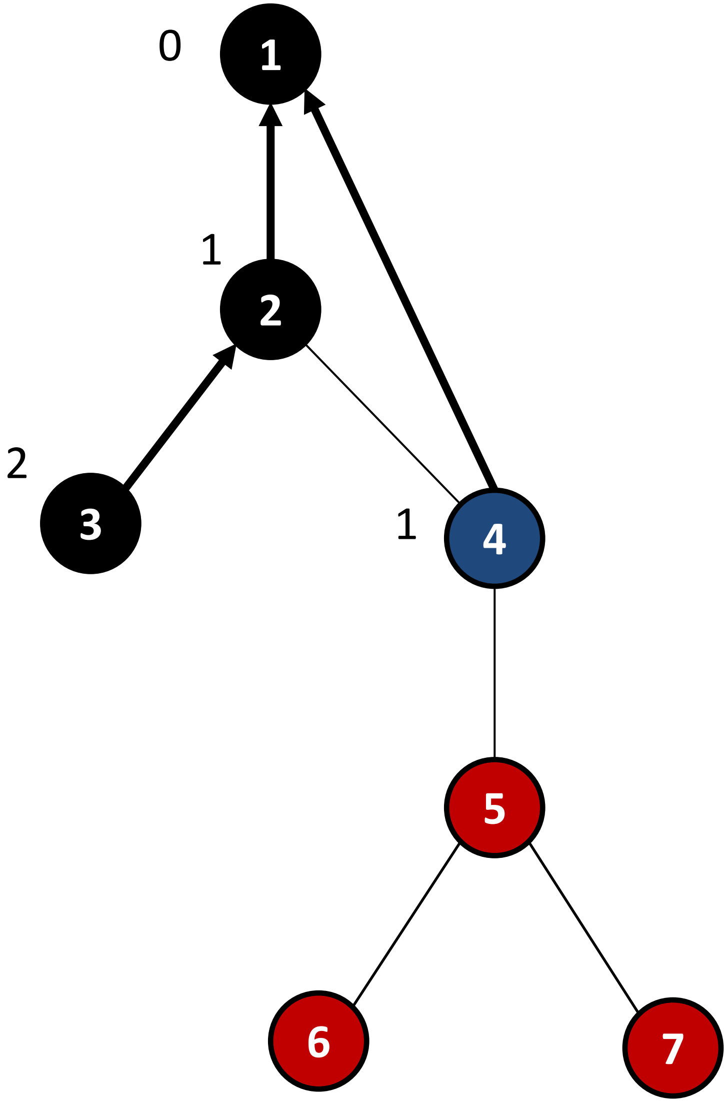
Breadth-first search (from CLR)



Queue: 4 3

```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

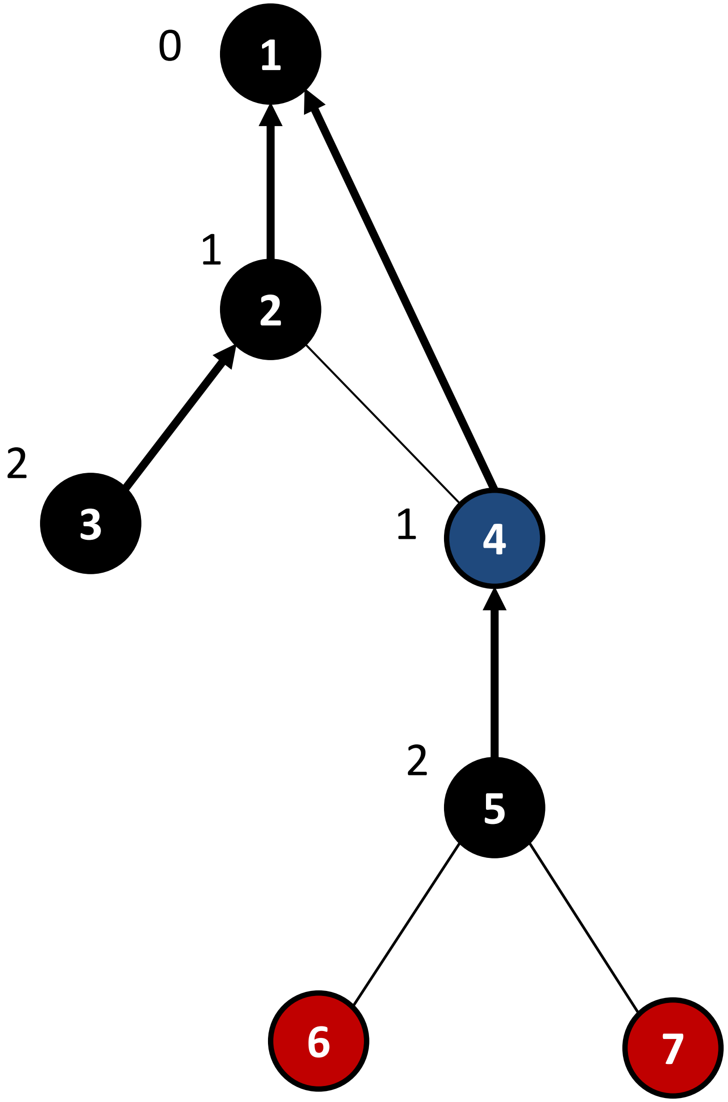
Breadth-first search (from CLR)



Queue: 3

```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

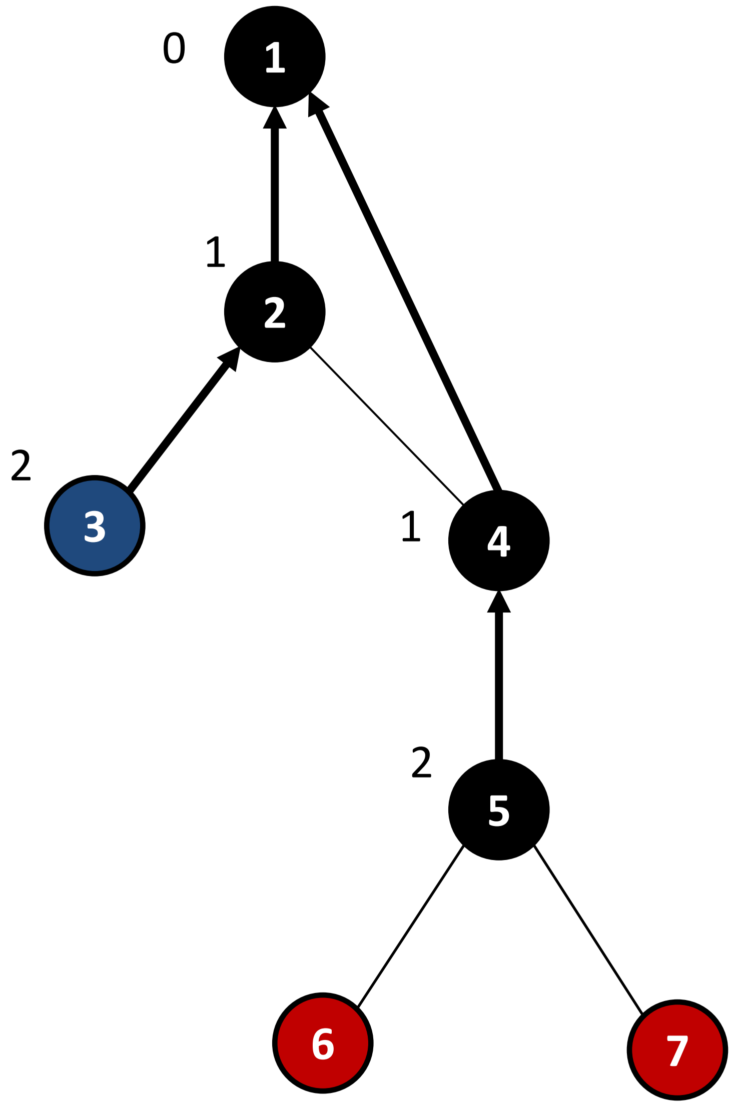
Breadth-first search (from CLR)



Queue: 3 5

```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

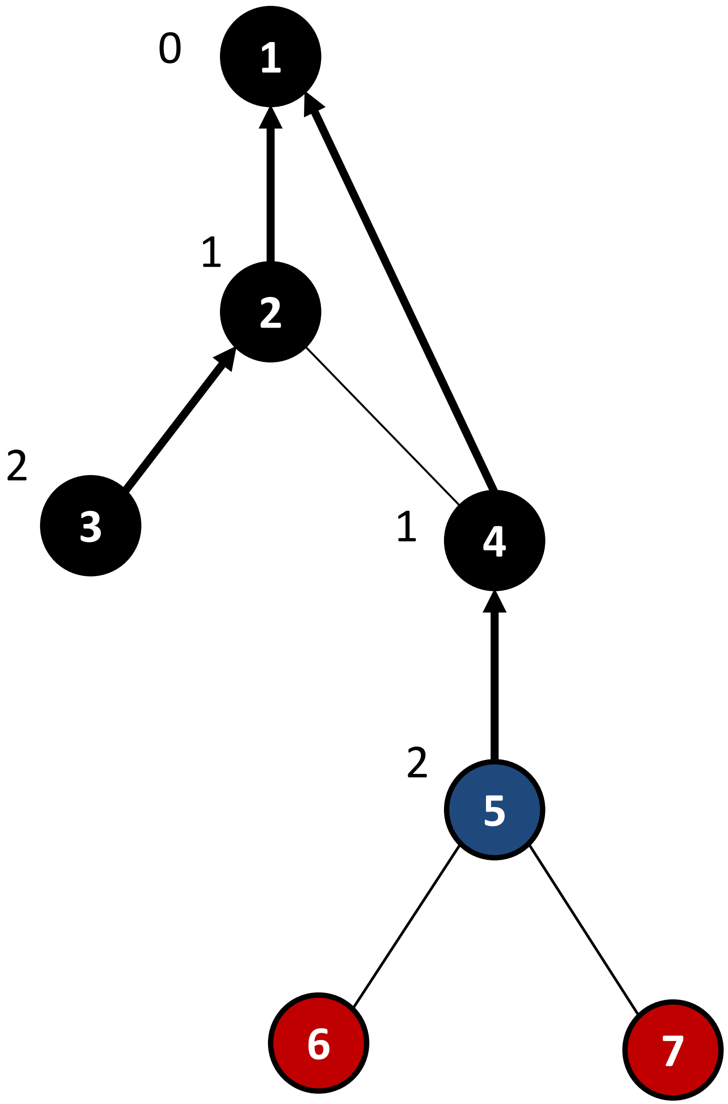
Breadth-first search (from CLR)



Queue: 5

```
BreadthFirstSearch(start) {
   q = empty queue
   q.Enqueue(start)
   start.distance = 0
   start.predecessor = null
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         if c not visited {
            q.Enqueue(c)
            c.distance = node.distance+1
            c.predecessor = node
         }
   }
}
```

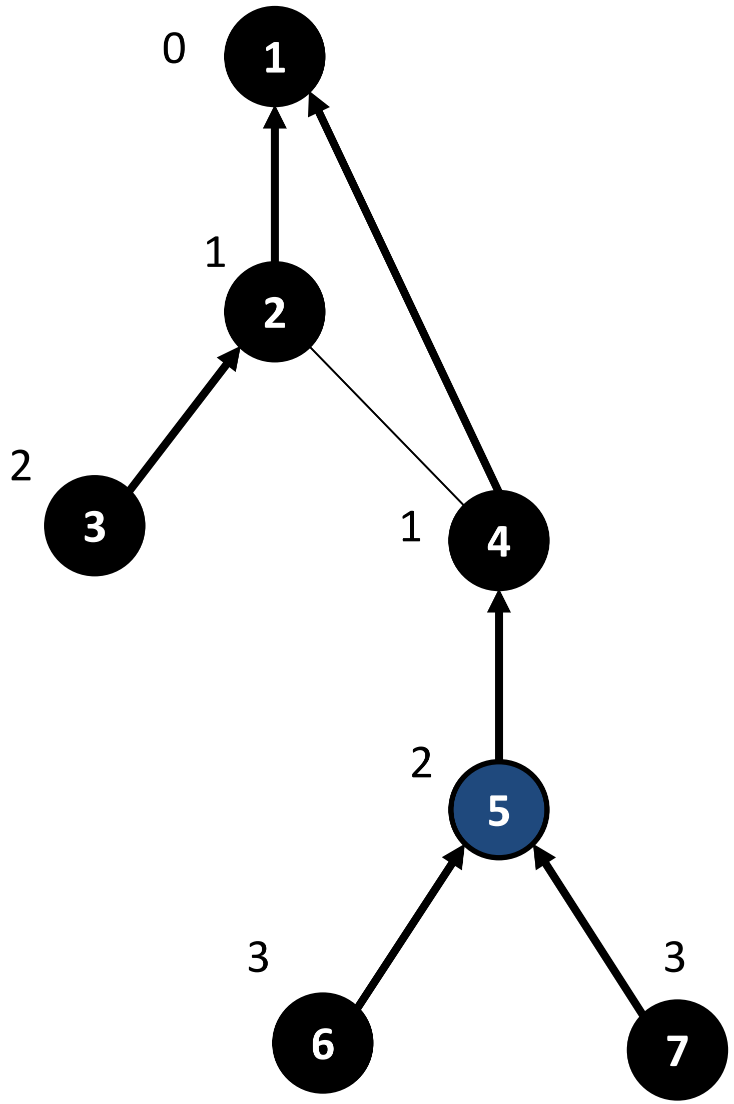
Breadth-first search (from CLR)



Queue:

```
BreadthFirstSearch(start) {
   q = empty queue
   q.Enqueue(start)
   start.distance = 0
   start.predecessor = null
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         if c not visited {
            q.Enqueue(c)
            c.distance = node.distance+1
            c.predecessor = node
         }
   }
}
```

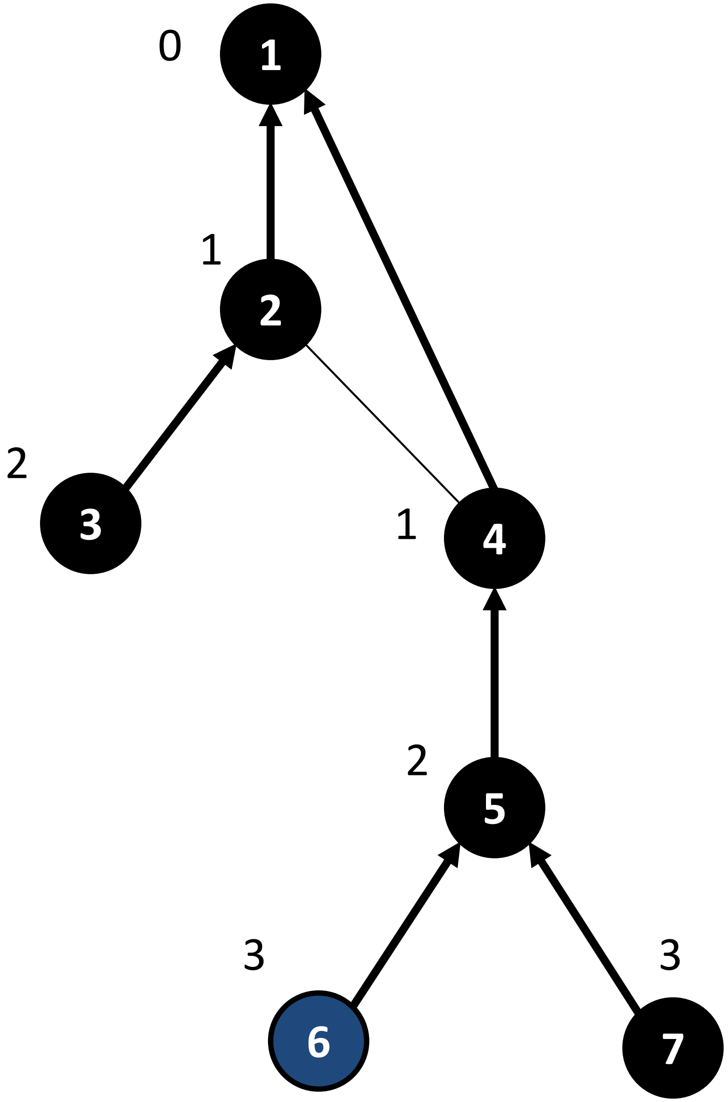
Breadth-first search (from CLR)



Queue: 6 7

```
BreadthFirstSearch(start) {
   q = empty queue
   q.Enqueue(start)
   start.distance = 0
   start.predecessor = null
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         if c not visited {
            q.Enqueue(c)
            c.distance = node.distance+1
            c.predecessor = node
         }
   }
}
```

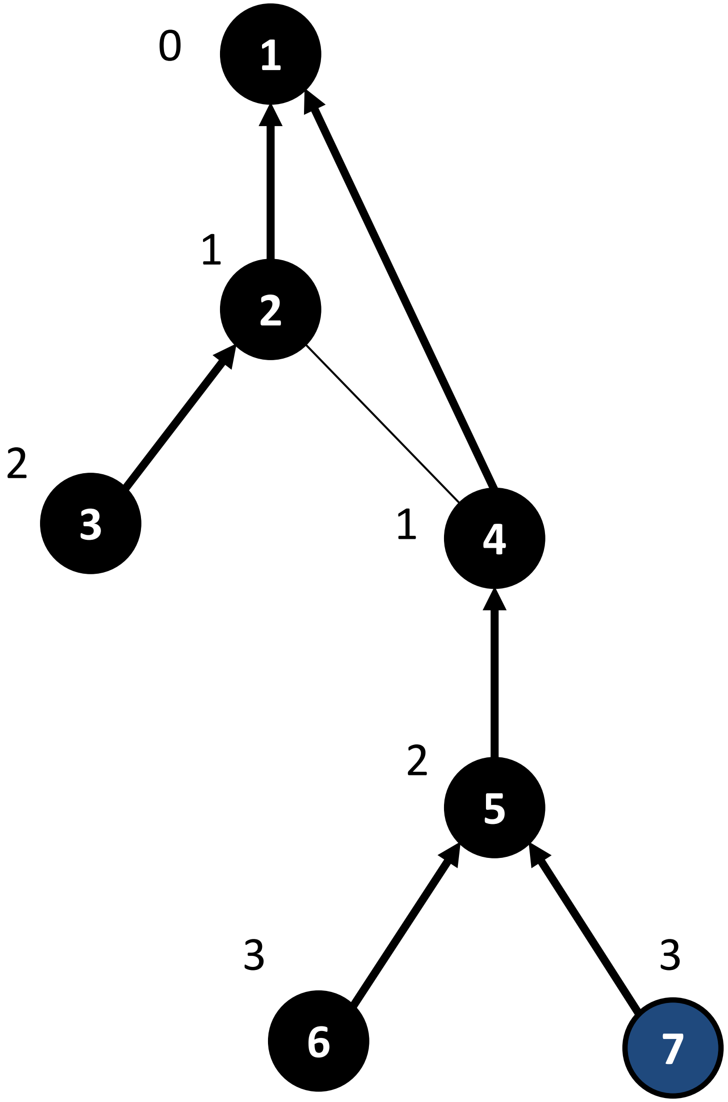
Breadth-first search (from CLR)



Queue: 7

```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

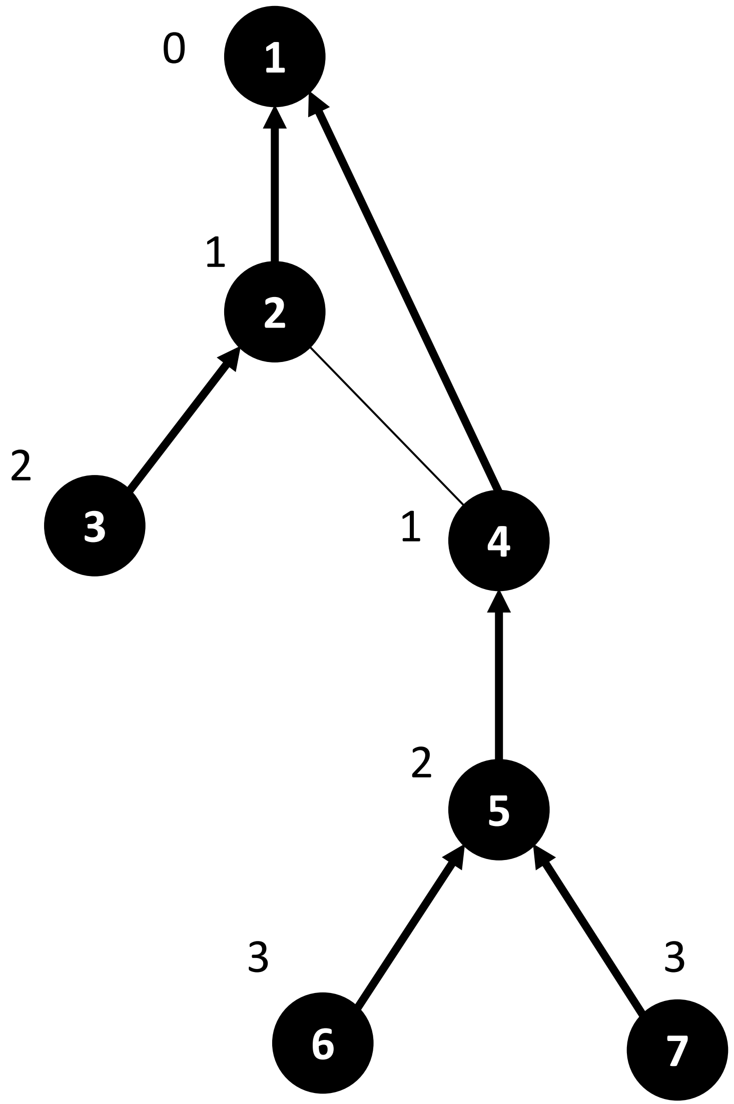
Breadth-first search (from CLR)



Queue:

```
BreadthFirstSearch(start) {
   q = empty queue
   q.Enqueue(start)
   start.distance = 0
   start.predecessor = null
   mark start visited
   while q not empty {
      node = q.Dequeue()
      for each neighbor c
         if c not visited {
            q.Enqueue(c)
            c.distance = node.distance+1
            c.predecessor = node
         }
   }
}
```

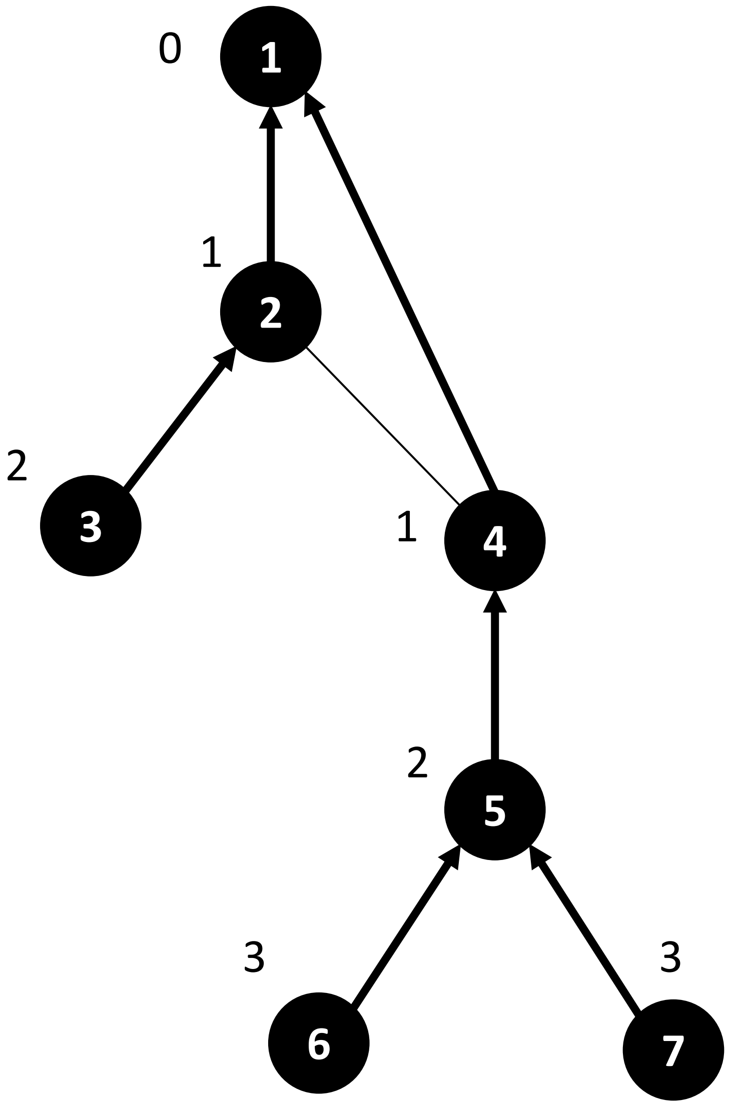
Breadth-first search (from CLR)



Queue:

```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

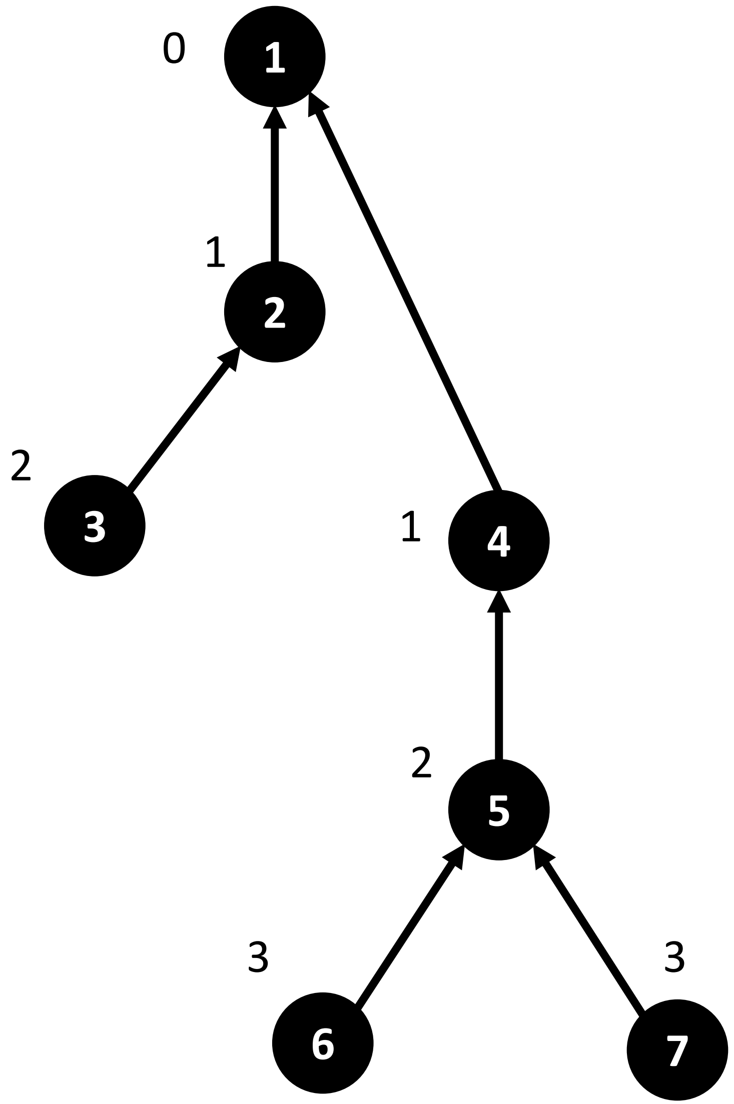
Breadth-first search (from CLR)



Notice

- Every node is labeled with its distance from the start node

Breadth-first search (from CLR)

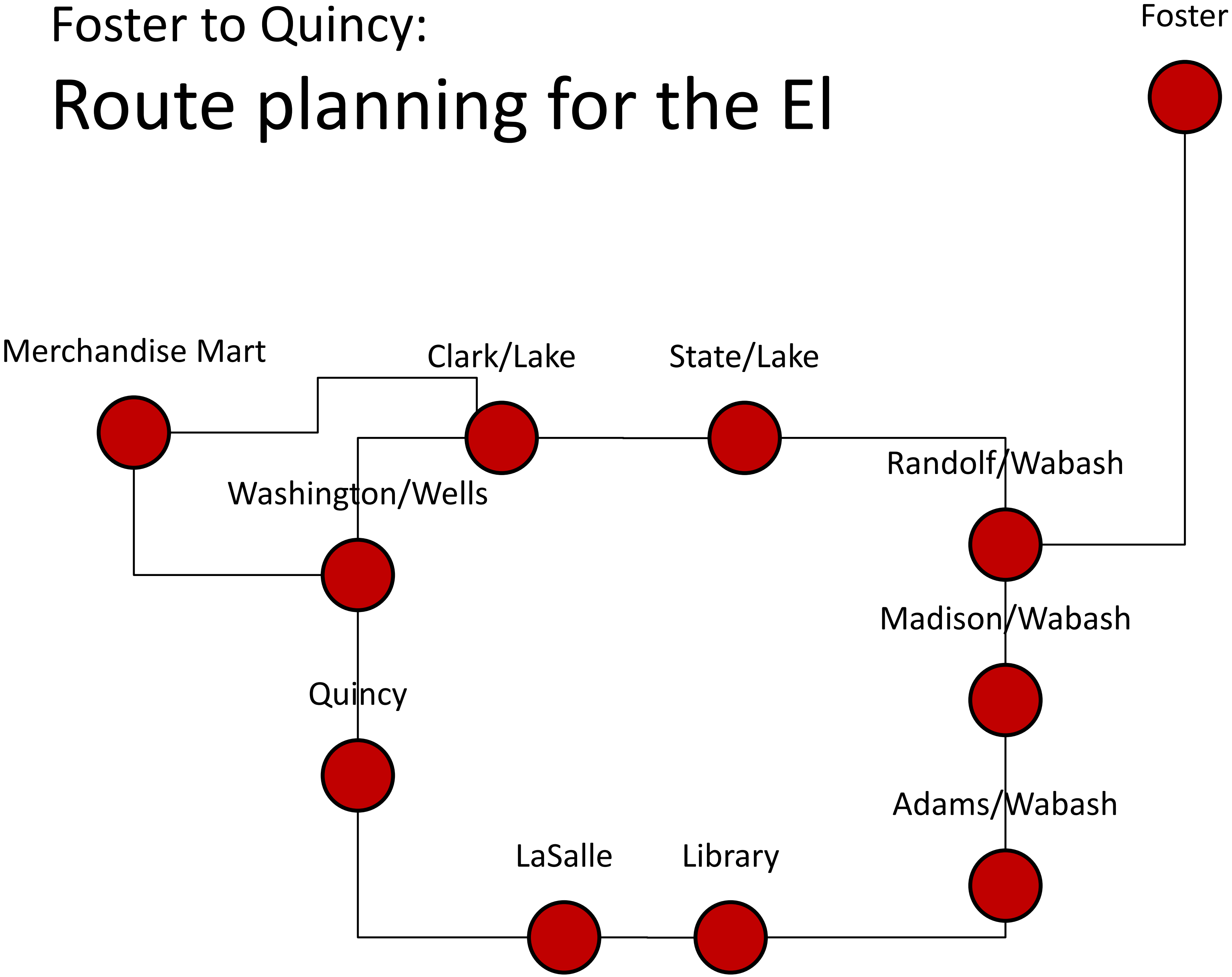


Notice

- Every node is labeled with its distance from the start node
- The **predecessor subgraph** (formed using only the edges corresponding to predecessor links) forms a tree
  - Called a **breadth-first tree**
- The predecessor links give you the **shortest path** from any node back to the start node
  - Technically, it's **a** shortest path
  - Since there might be other paths in the original graph of the same length

Foster to Quincy:

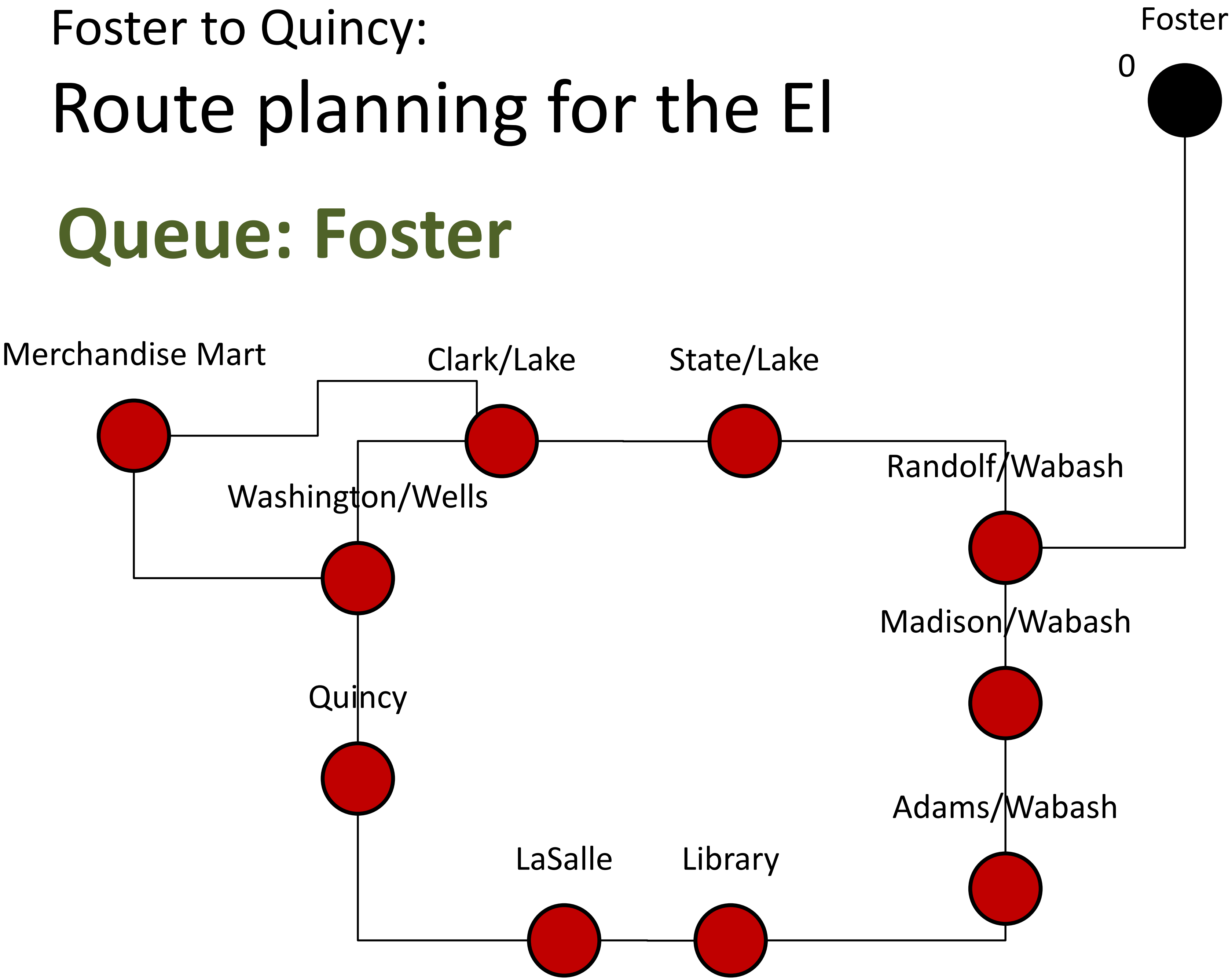
Route planning for the El



Foster to Quincy:

Route planning for the El

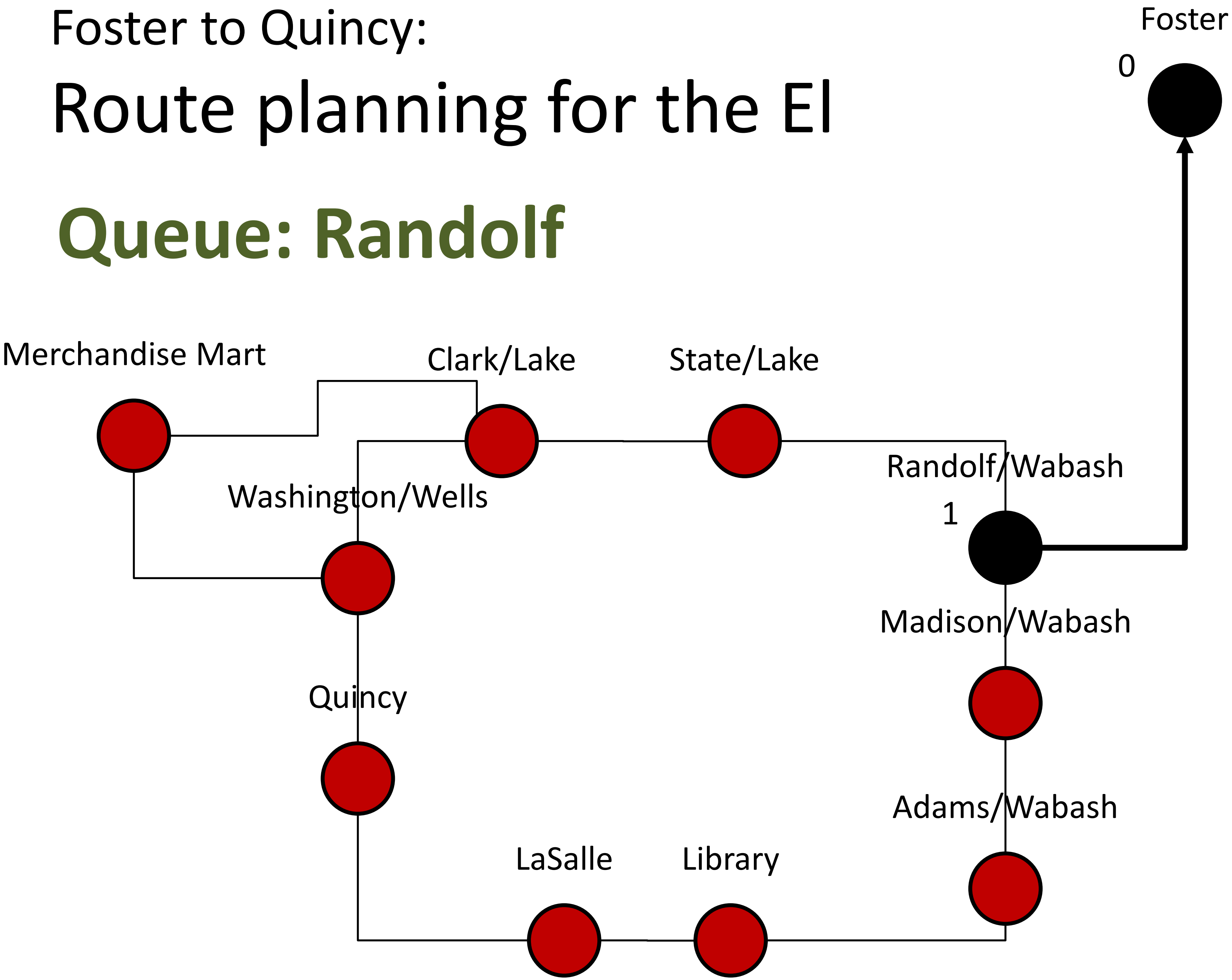
Queue: Foster



Foster to Quincy:

Route planning for the El

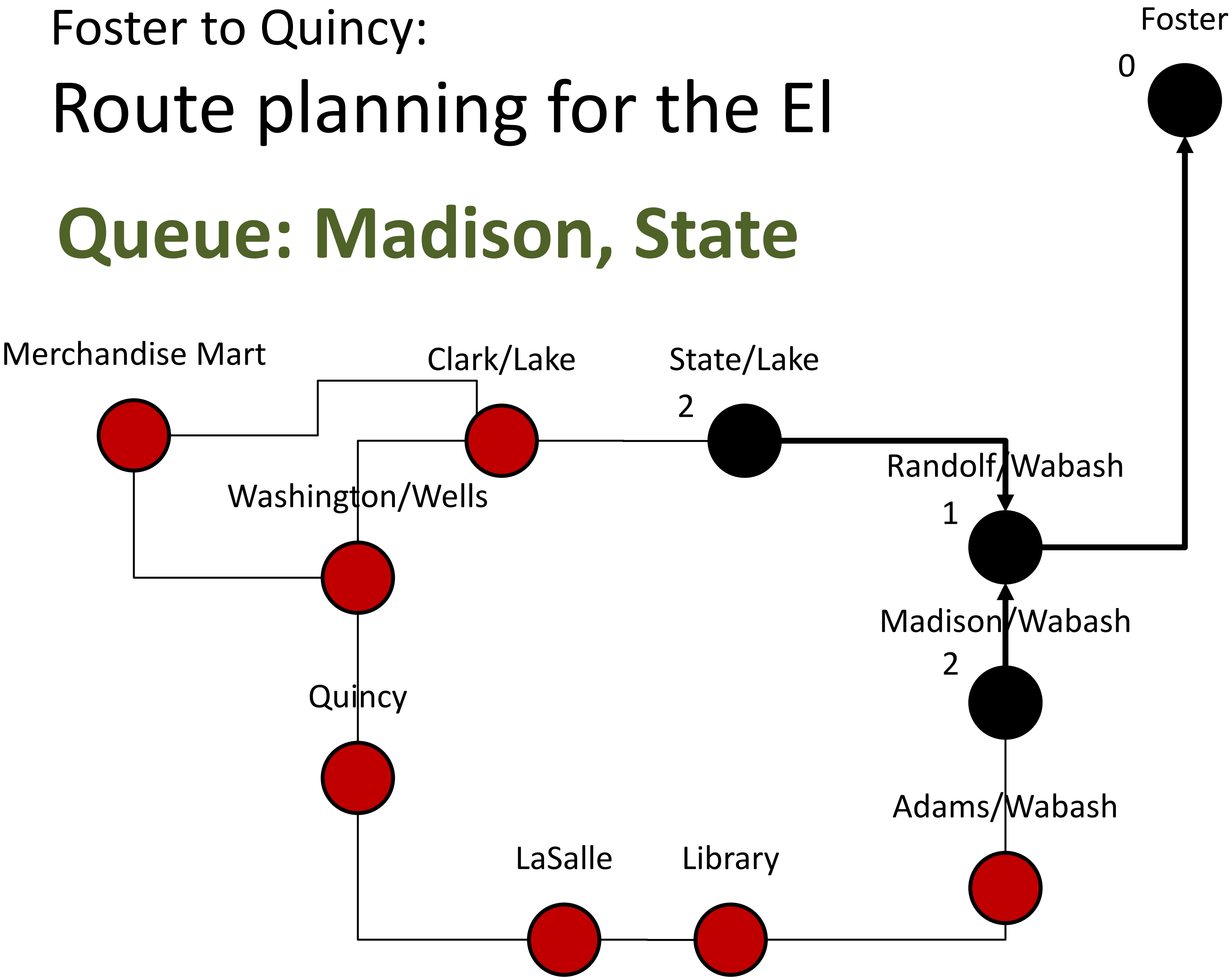
Queue: Randolf



Foster to Quincy:

Route planning for the El

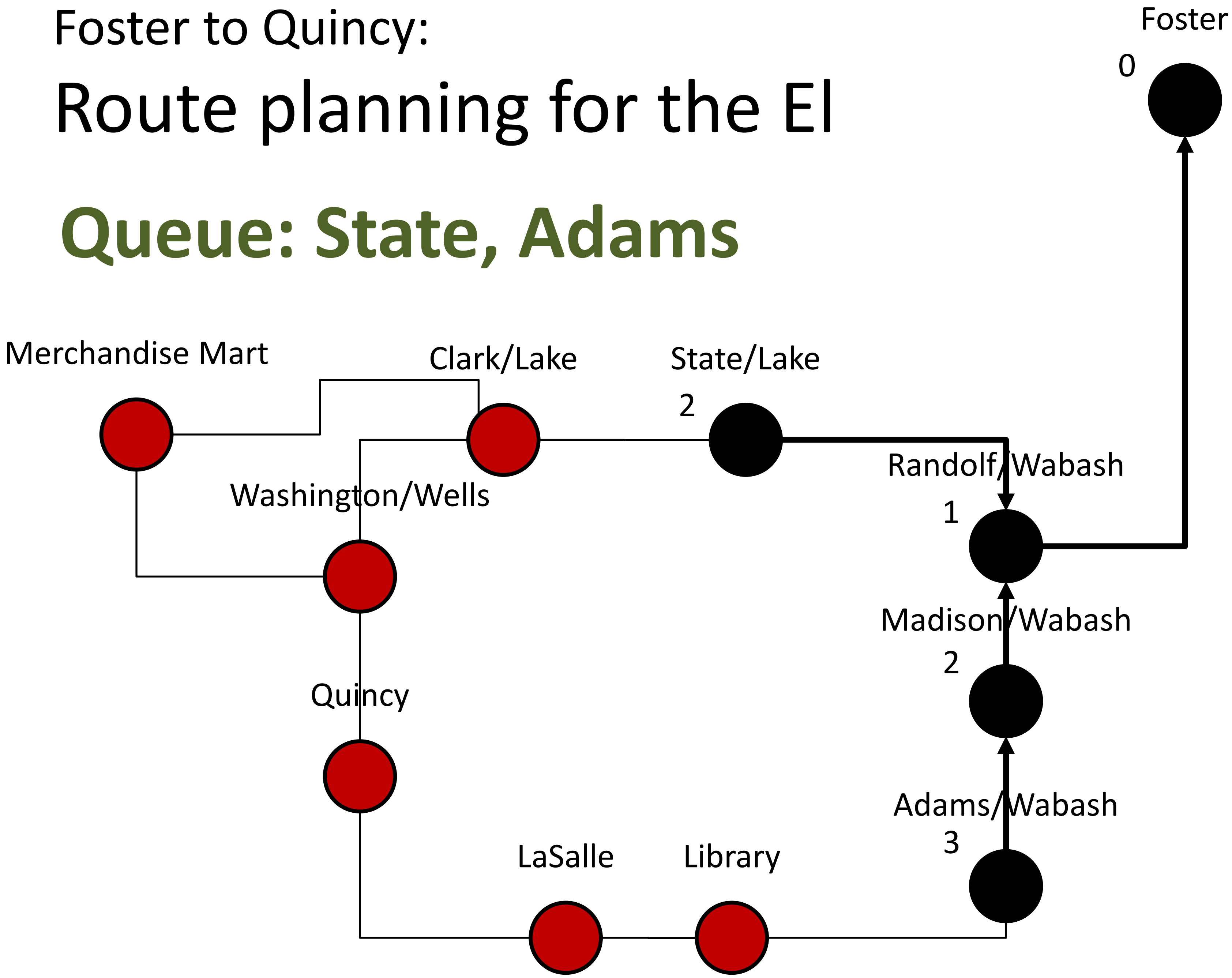
Queue: Madison, State



Foster to Quincy:

Route planning for the El

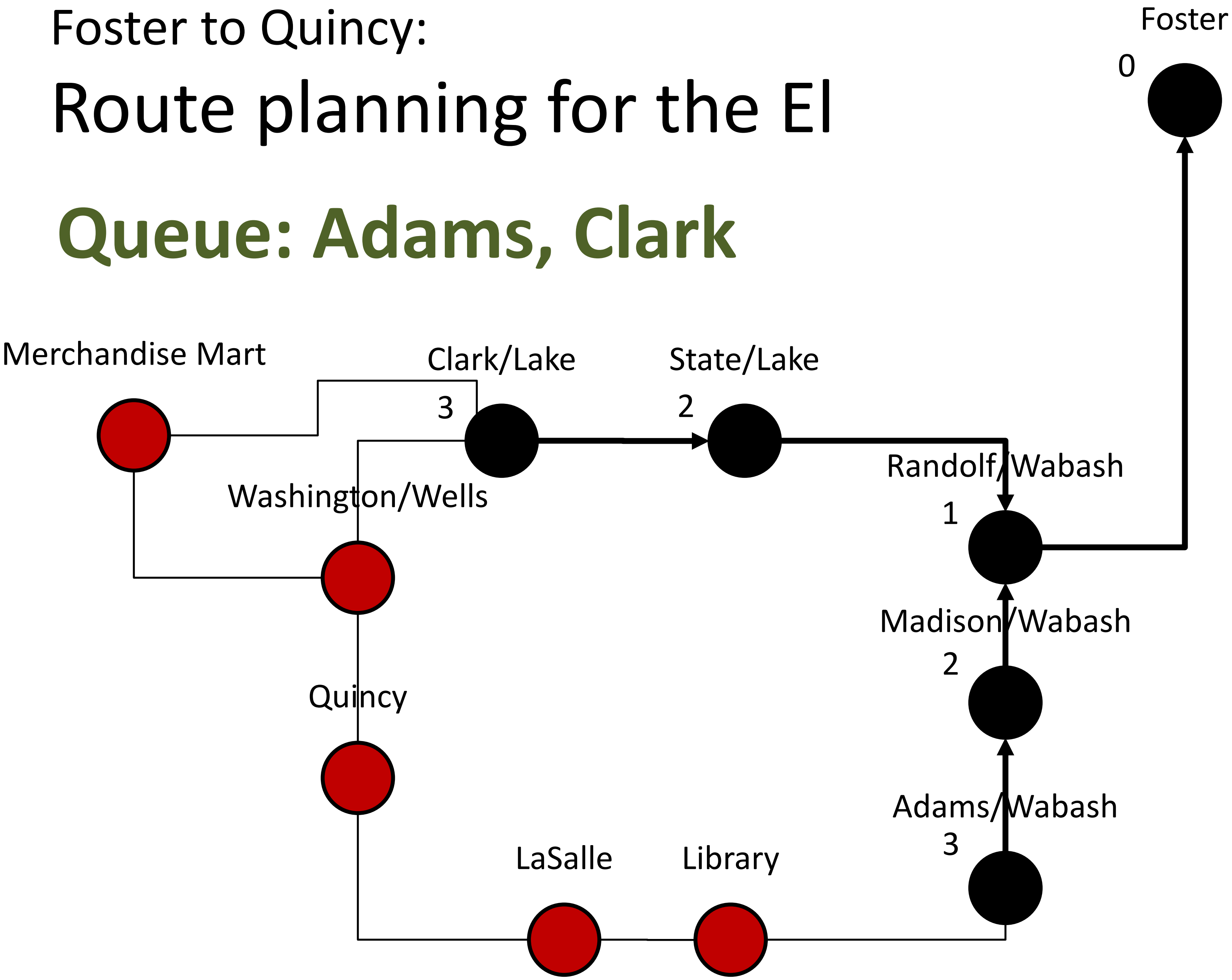
Queue: State, Adams



Foster to Quincy:

Route planning for the El

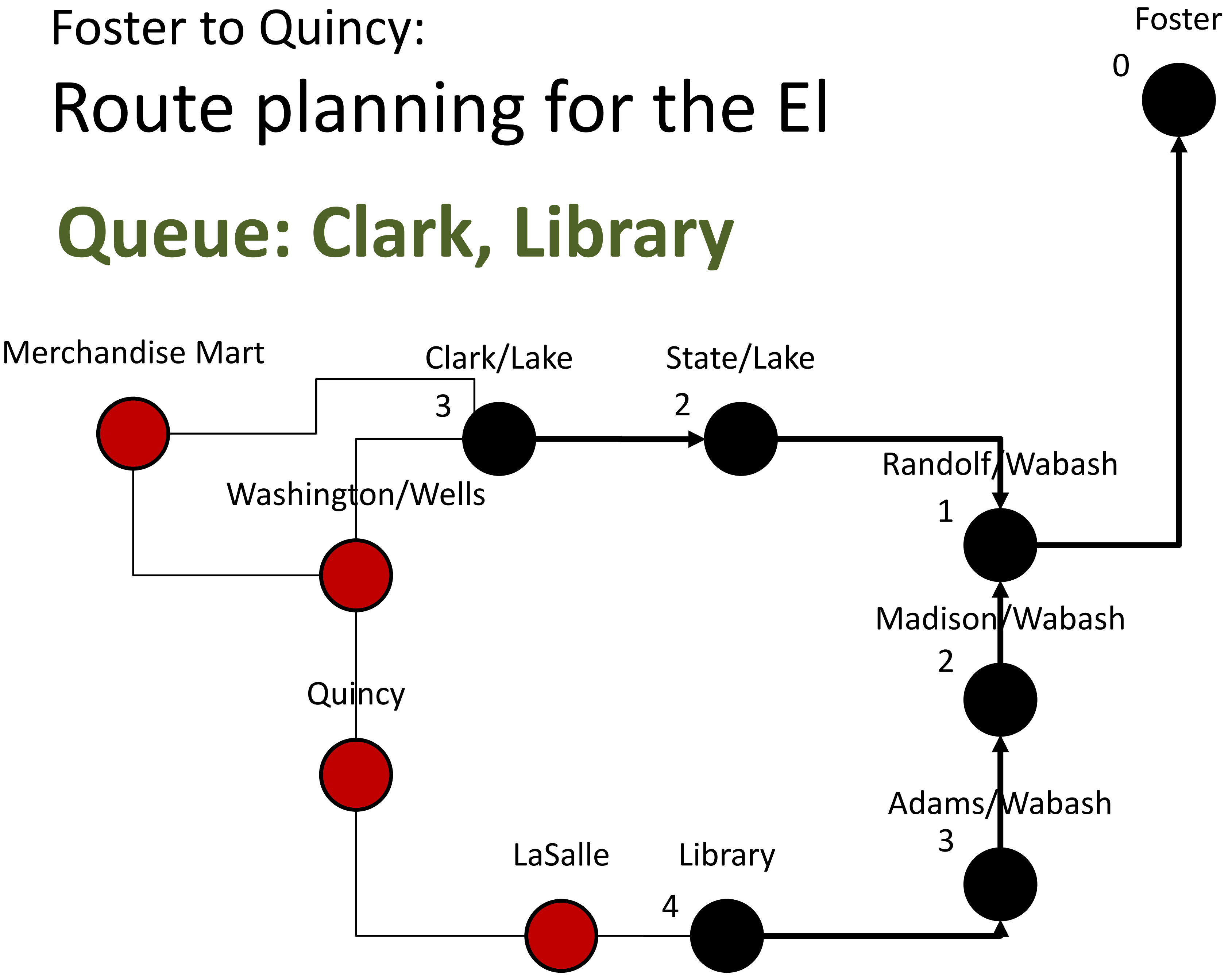
Queue: Adams, Clark



Foster to Quincy:

Route planning for the El

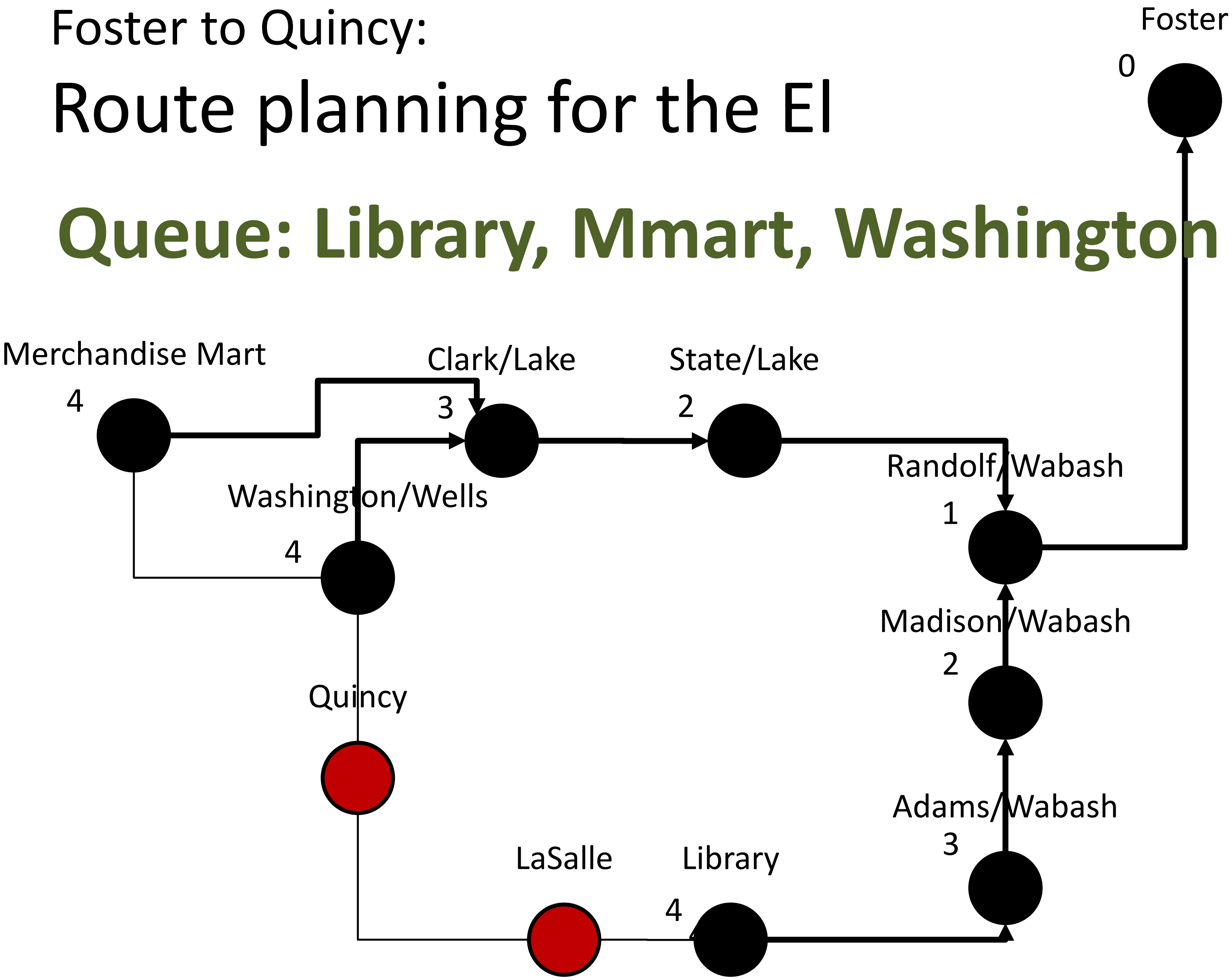
Queue: Clark, Library



Foster to Quincy:

Route planning for the El

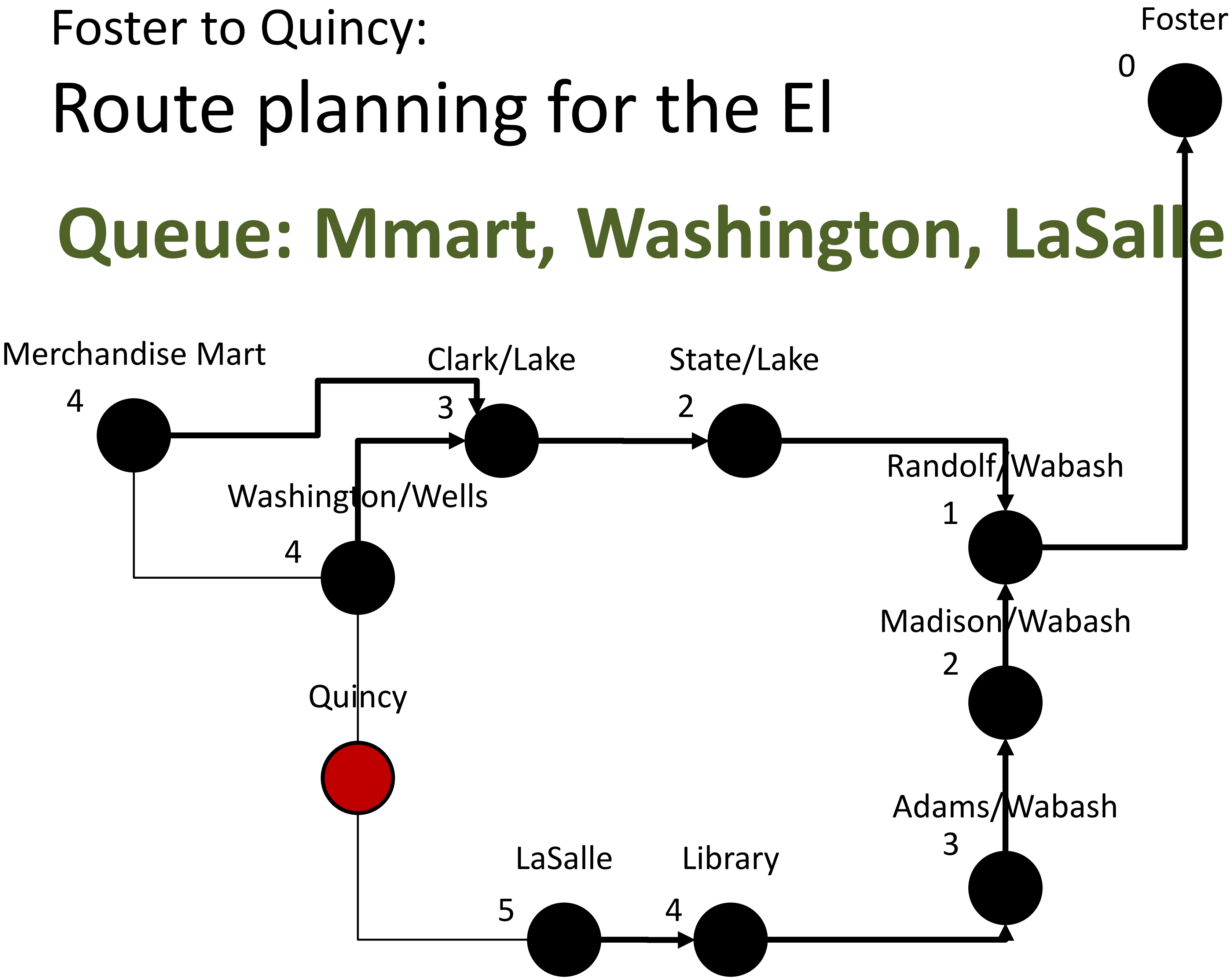
Queue: Library, Mmart, Washington



Foster to Quincy:

Route planning for the El

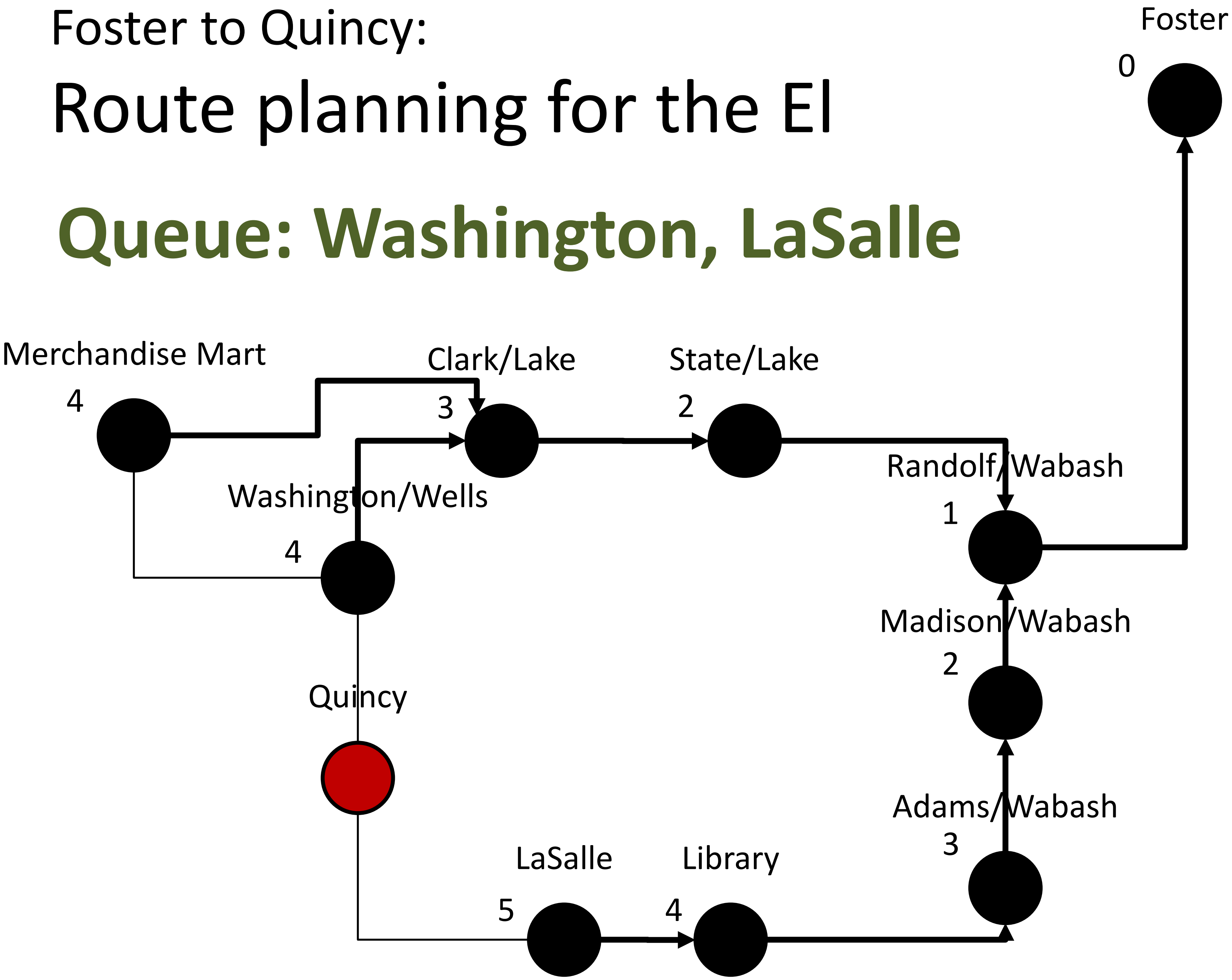
Queue: Mmart, Washington, LaSalle



Foster to Quincy:

Route planning for the El

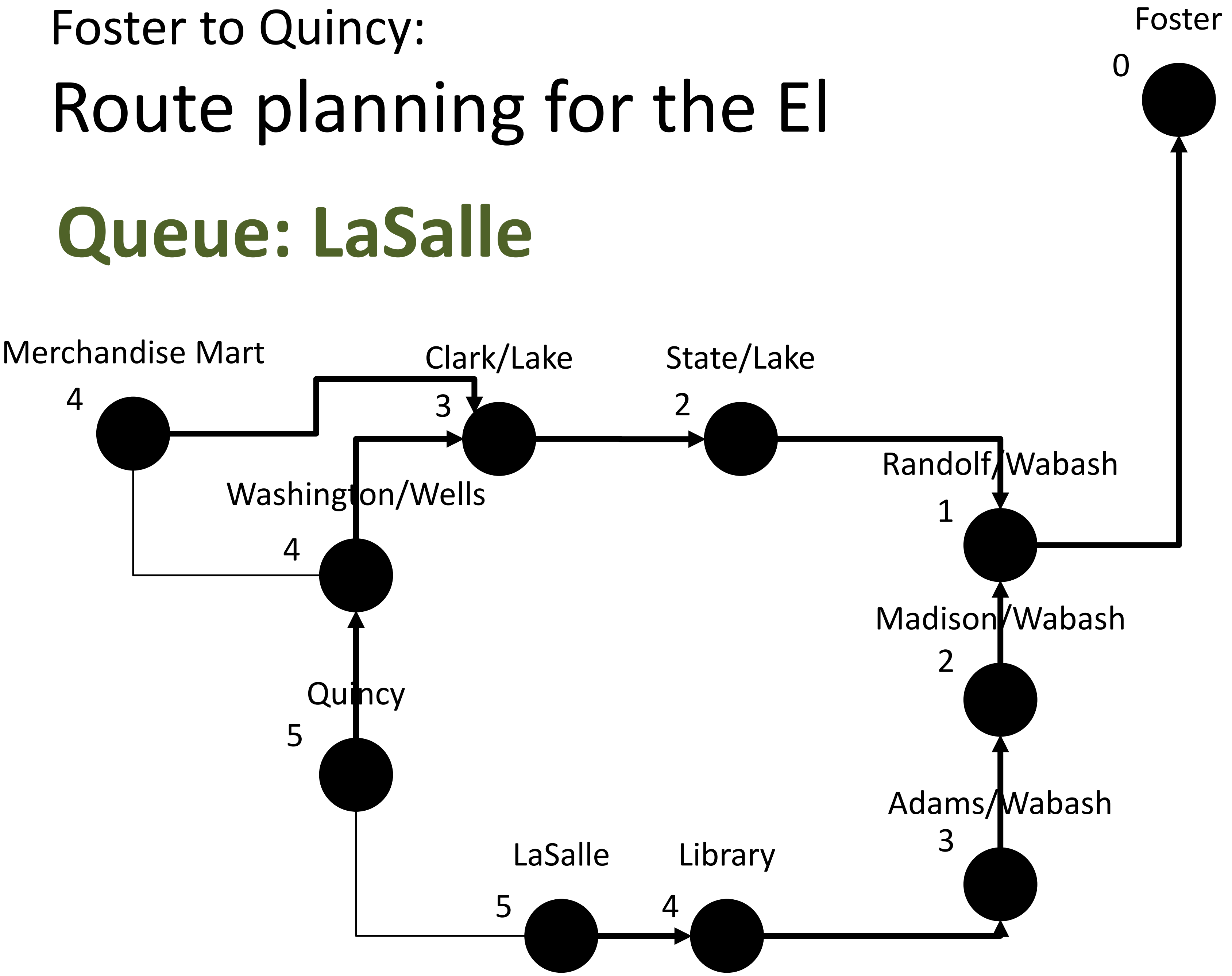
Queue: Washington, LaSalle



Foster to Quincy:

Route planning for the El

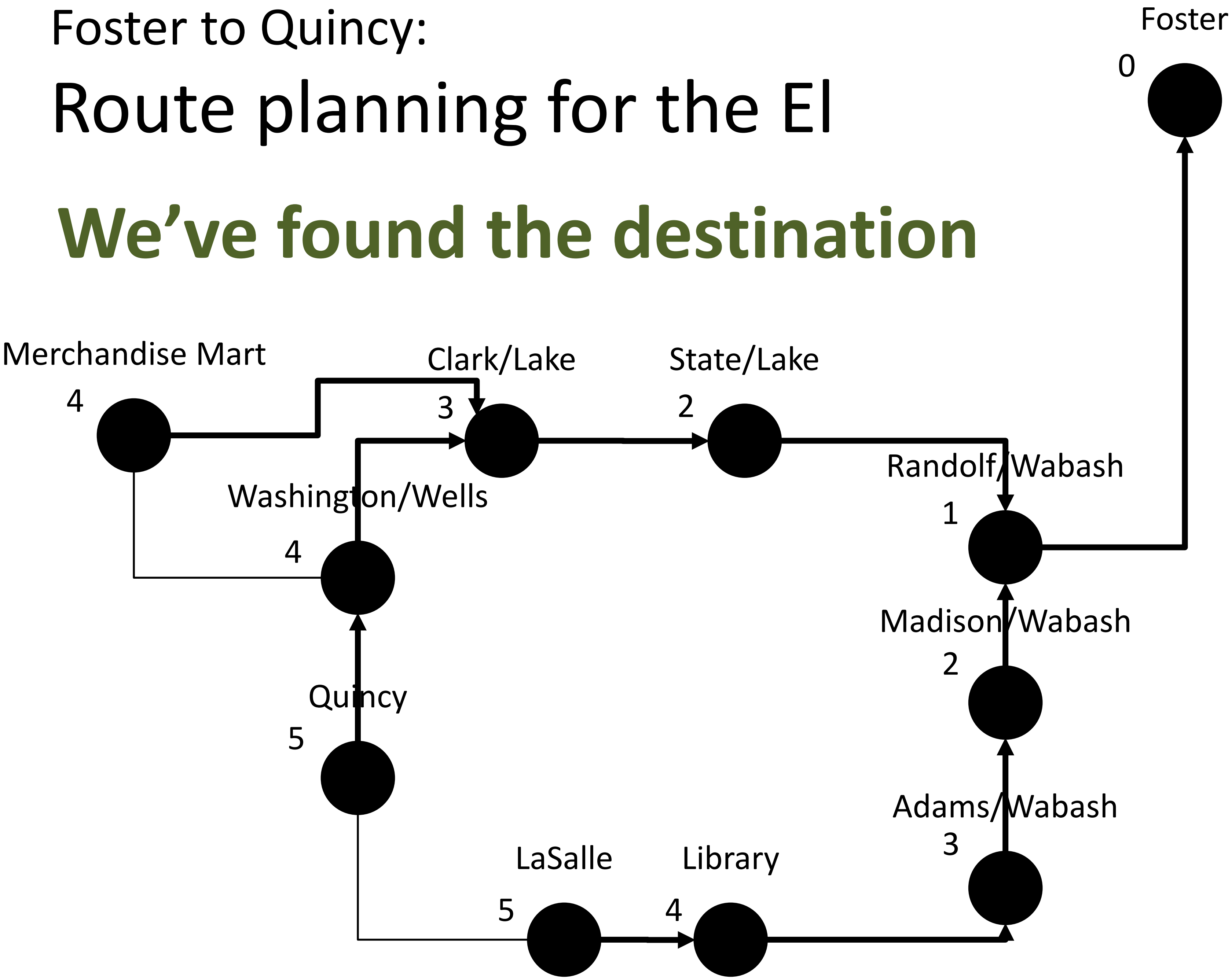
Queue: LaSalle



Foster to Quincy:

Route planning for the El

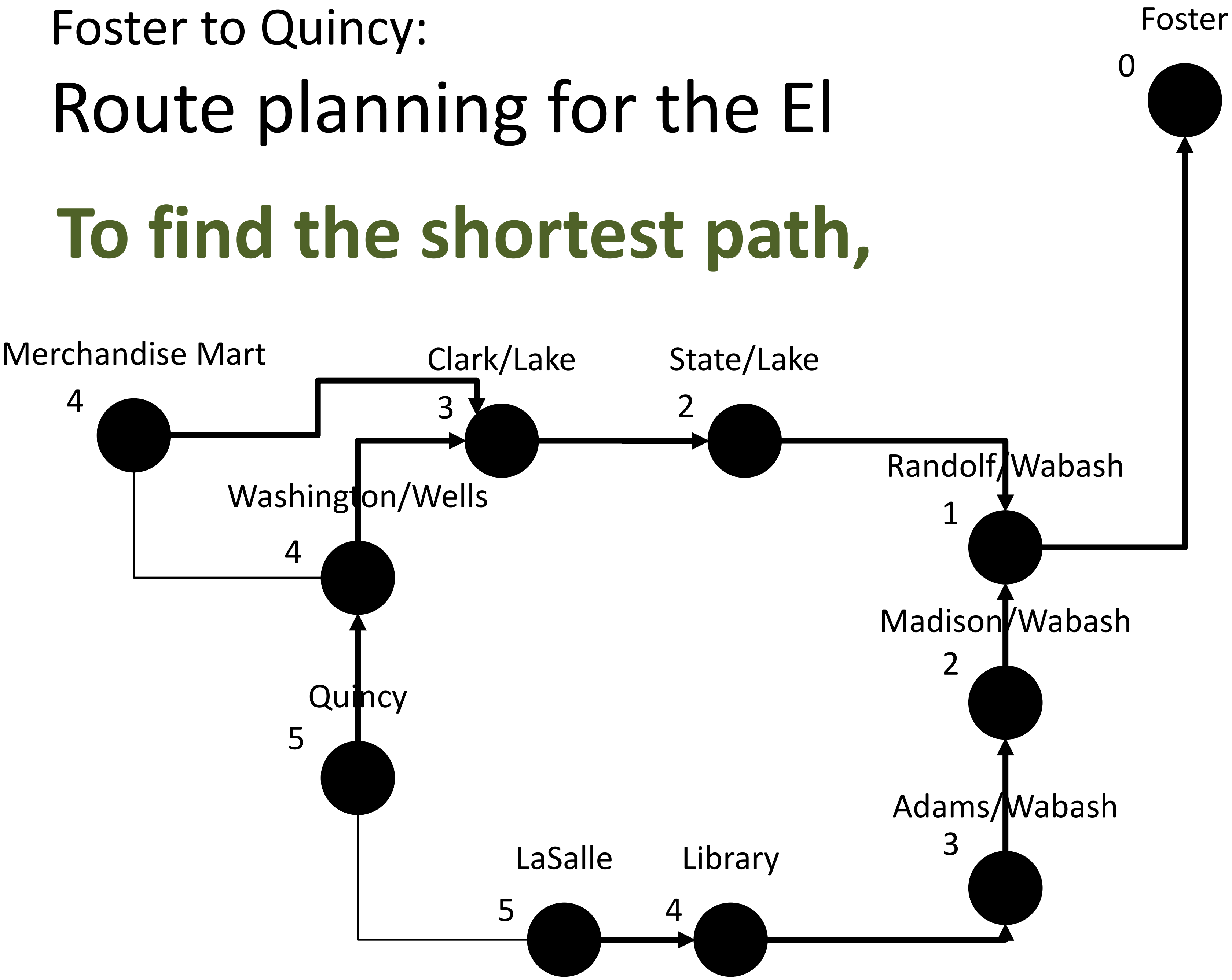
We’ve found the destination



Foster to Quincy:

Route planning for the El

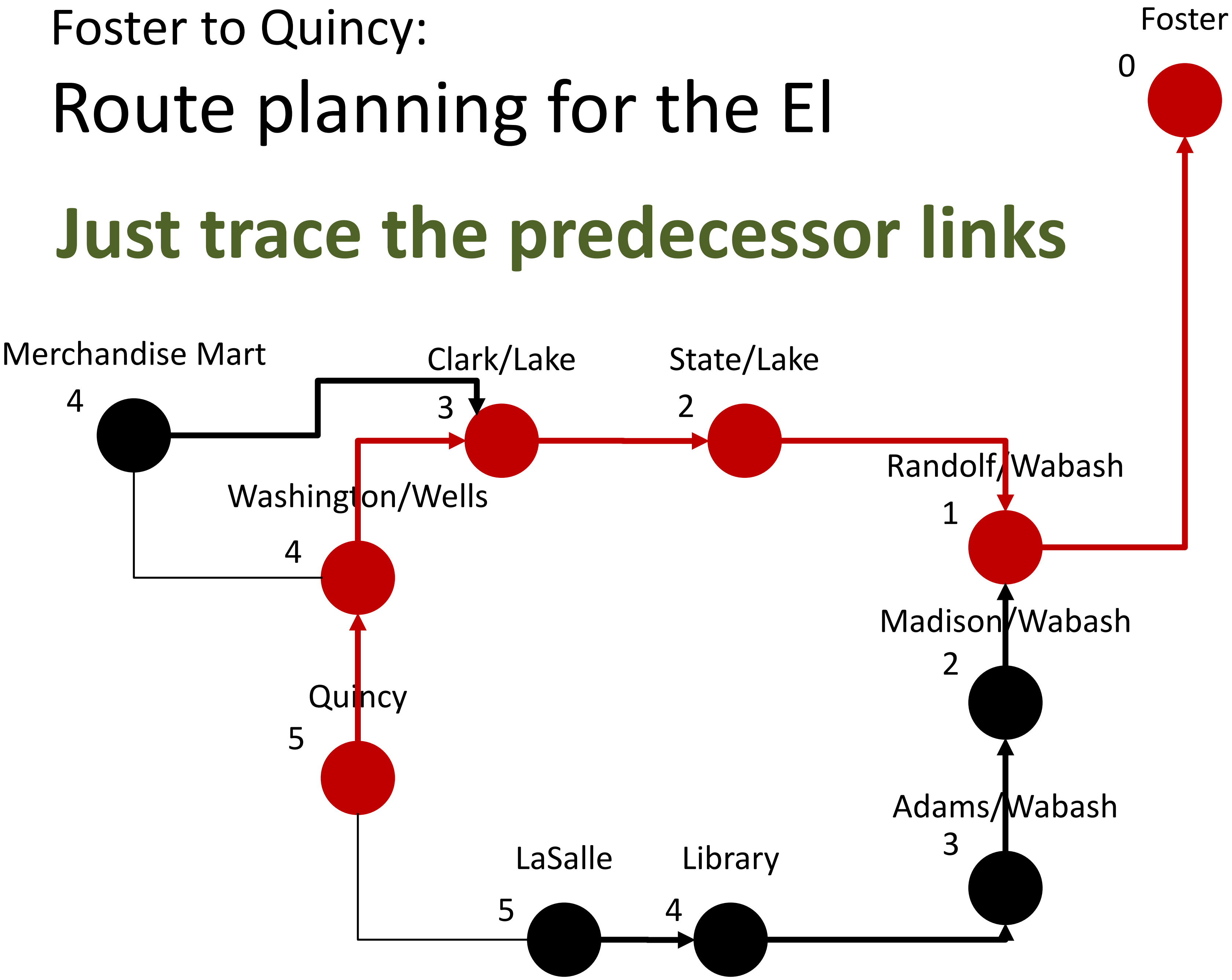
To find the shortest path,



Foster to Quincy:

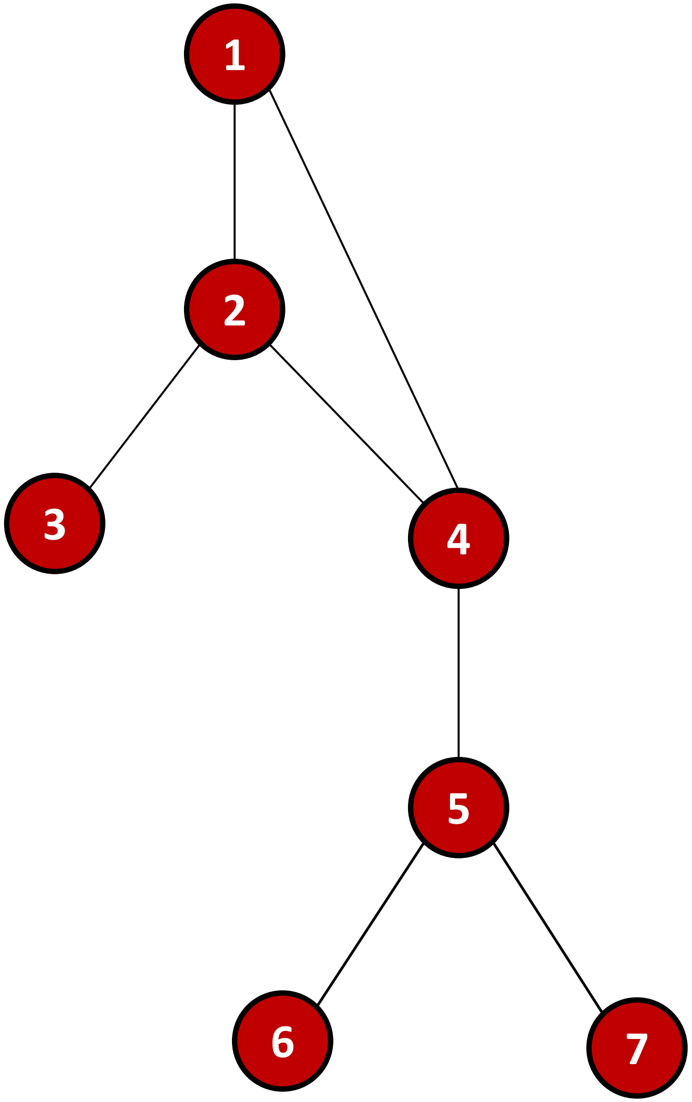
Route planning for the El

Just trace the predecessor links



Complexity analysis

How many times does each line run?



```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

Complexity analysis

How many times does each line run?

```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

Complexity analysis

How many times does each line run?

One time only

Once per vertex

Once per edge

Once per vertex

```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

Complexity analysis

How many times does each line run?

$$O(1)$$

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

$$+ O(V)$$

$$= O(V + E)$$

```
BreadthFirstSearch(start) {
  q = empty queue
  q.Enqueue(start)
  start.distance = 0
  start.predecessor = null
  mark start visited
  while q not empty {
    node = q.Dequeue()
    for each neighbor c
      if c not visited {
        q.Enqueue(c)
        c.distance = node.distance+1
        c.predecessor = node
      }
  }
}
```

BFS algorithm in the books

- Assumes **array-of-adjacency-lists** representation
- Extra information stored in **arrays**, indexed by node number
  - dist[node] = distance
    - d[node] in CLR
  - pred[node] = predecessor
    - π[node] in CLR
- “**Colors**” the nodes rather than marking them visted
  - Three colors
    - White = unvisited
    - Gray = in the queue or being processed now
    - Black = finshed processing
  - Stored in color[node]

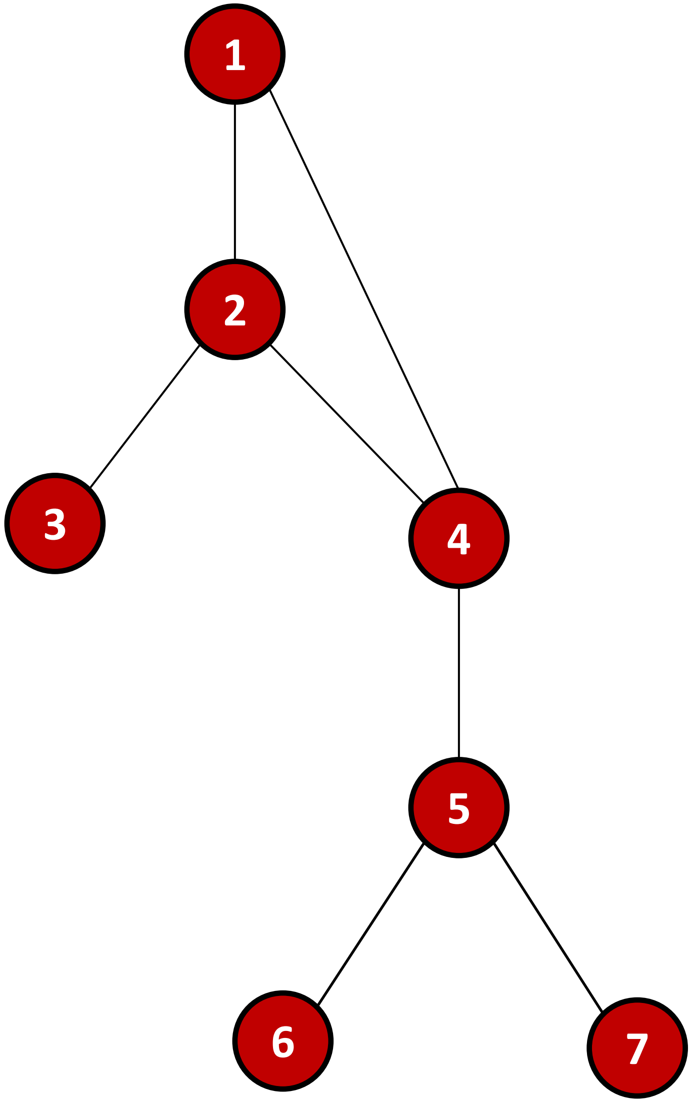
The actual code

```
BFS(start)
  for each vertex v
    pred[v] = -1
    dist[v] = ∞
    color[v] = white
  color[start] = gray
  dist[start] = 0
  Q = empty queue
  enqueue(Q, start)
  while Q not empty
    u = head of Q
    for each neighbor v of u
      if color[v] = white
        dist[v] = dist[u]+1
        pred[v] = u
        color[v] = gray
        Q.Enqueue(v)
    Q.Dequeue()
    color[u] = black
```

Depth-first search of a tree

```
DepthFirst(node)
    for each child c of node
        DepthFirst(c)
```

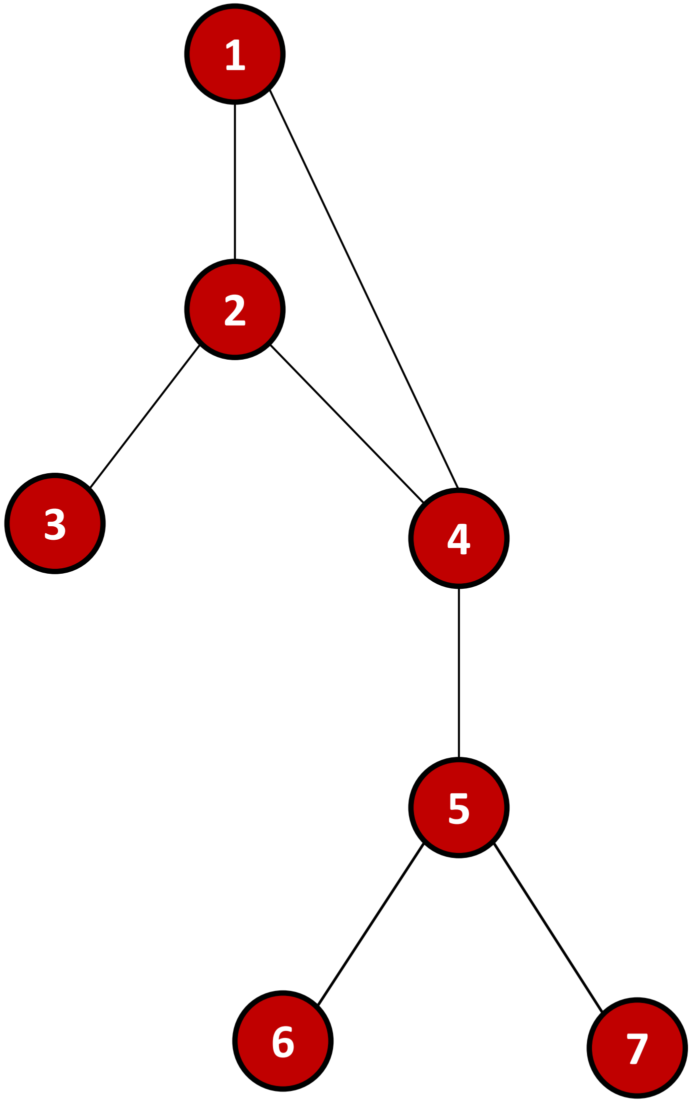
- Again, this **doesn't quite work**
  - No children per se
  - Need to keep from re-visiting nodes



Depth-first search of a graph

```
DepthFirst(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DepthFirst(c)
```

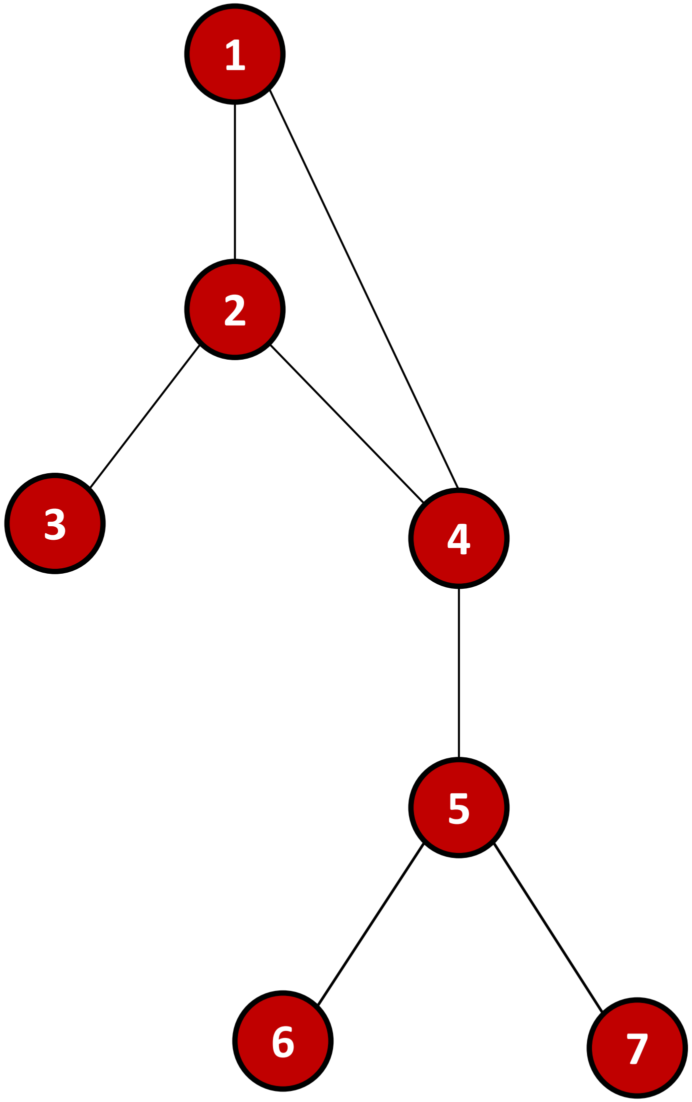
- We fix it by keeping track of what nodes have **already** been **visited**



Depth-first search of a graph

```
DepthFirst(node)
  mark node visited
  for each unvisited
            neighbor, c, of node
    DepthFirst(c)
```

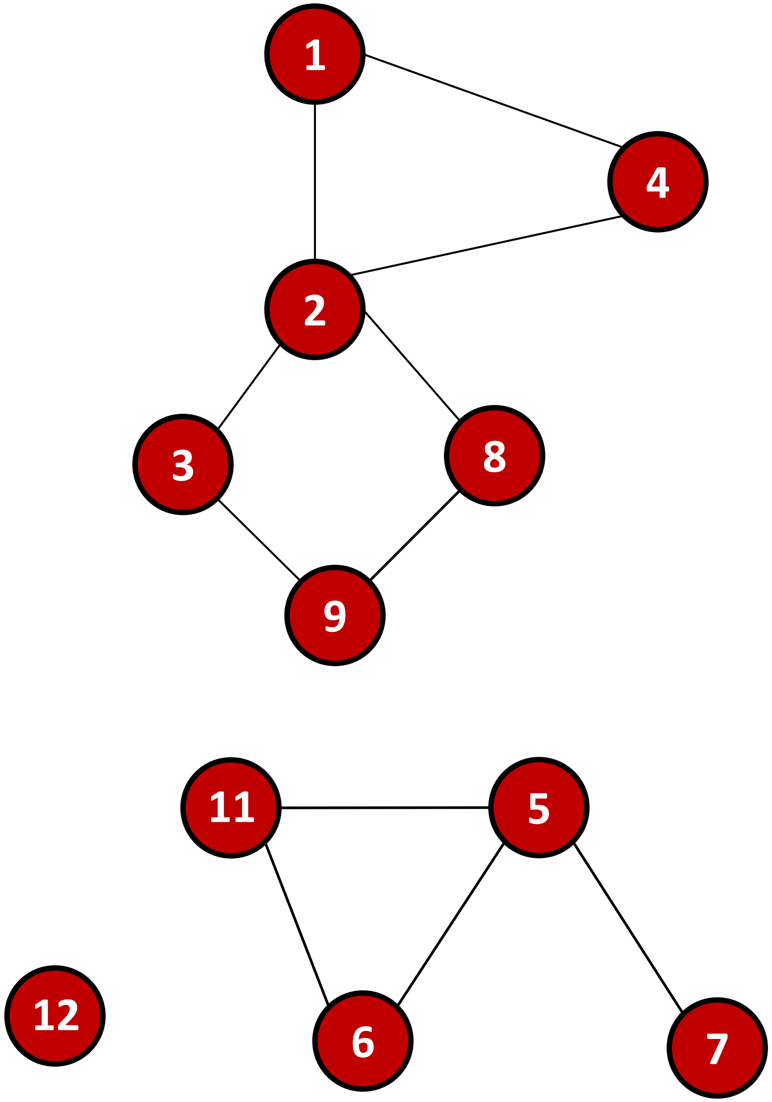
- However, in practice, one of the things DFS is most often used for is analyzing the **connectivity** of a graph



Depth-first search of a graph

```
DepthFirst(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DepthFirst(c)
```

- So it’s often run on an **unconnected graph**
- Which means that **not every node will be visited**

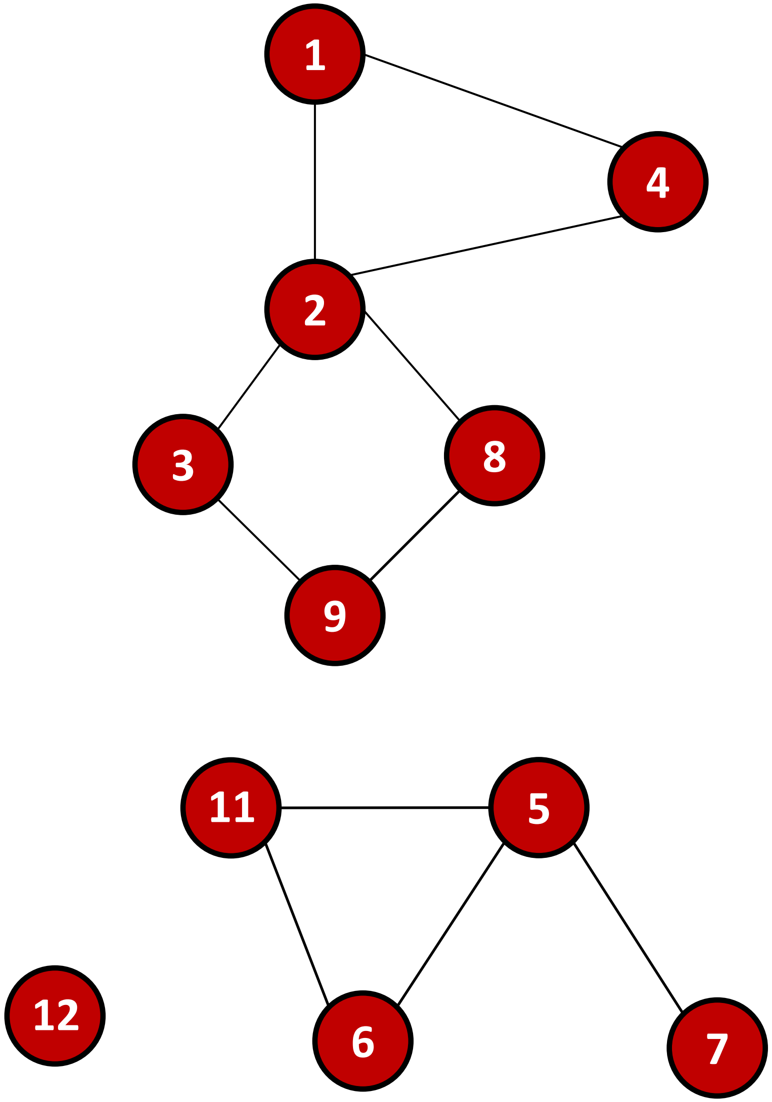


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
          DFSVisit(node)
```

```
DFSVisit(node)
  mark node visited
  foreach unvisited
              neighbor, c, of node
    DFSVisit(c)
```

- So we modify it to **run it on every node**



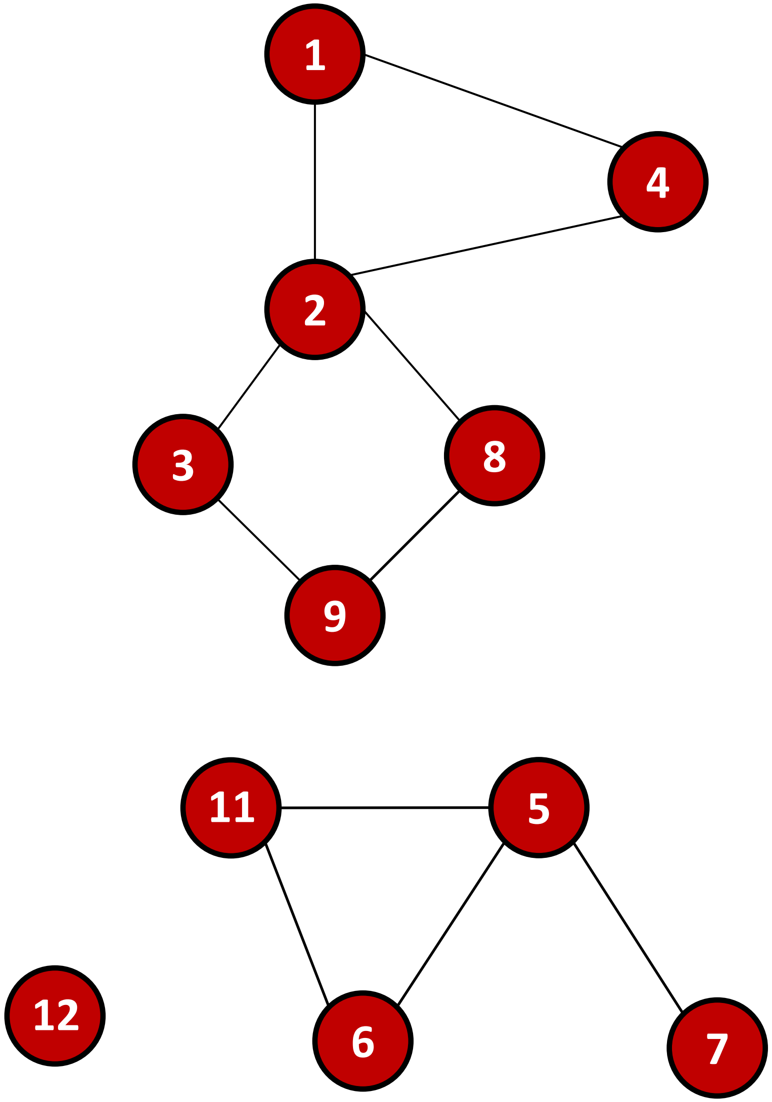
Depth-first search of a graph

```
DepthFirst()
   for each node in graph
        if node not visited
            DFSVisit(node)
```

```
DFSVisit(node)
   mark node visited
   for each unvisited neighbor c
      DFSVisit(c)
```

Call Stack:

- DepthFirst()

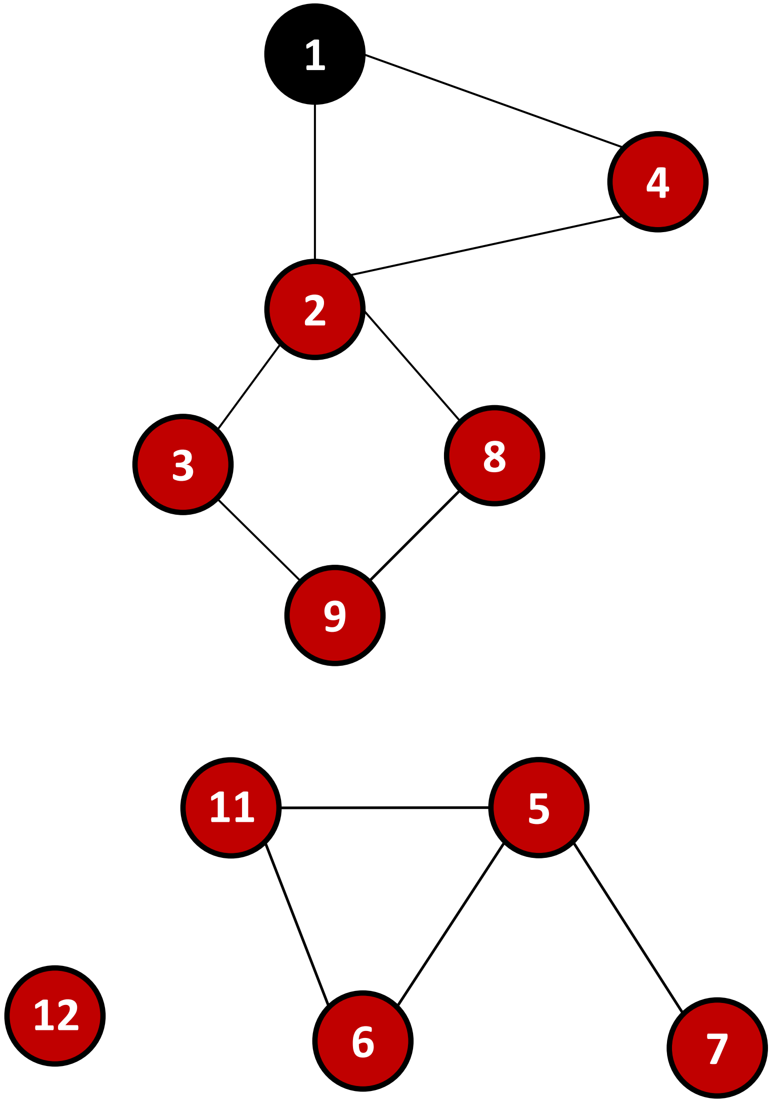


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
     if node not visited
       DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor c
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)

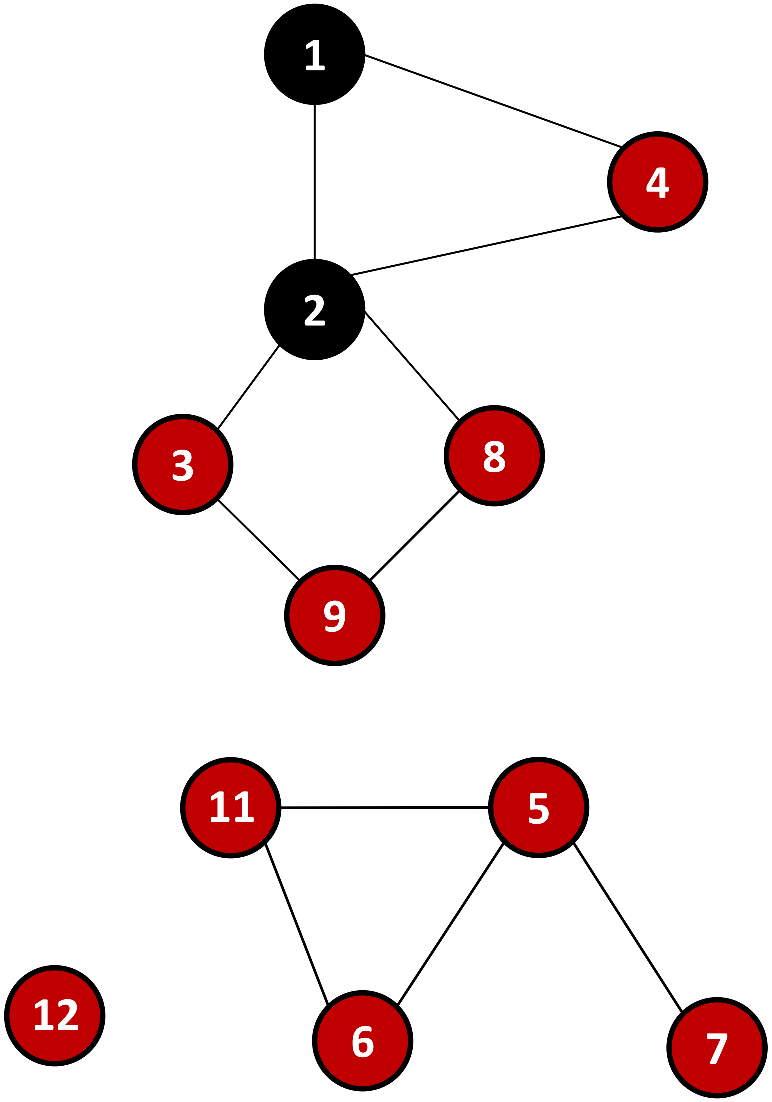


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(2)



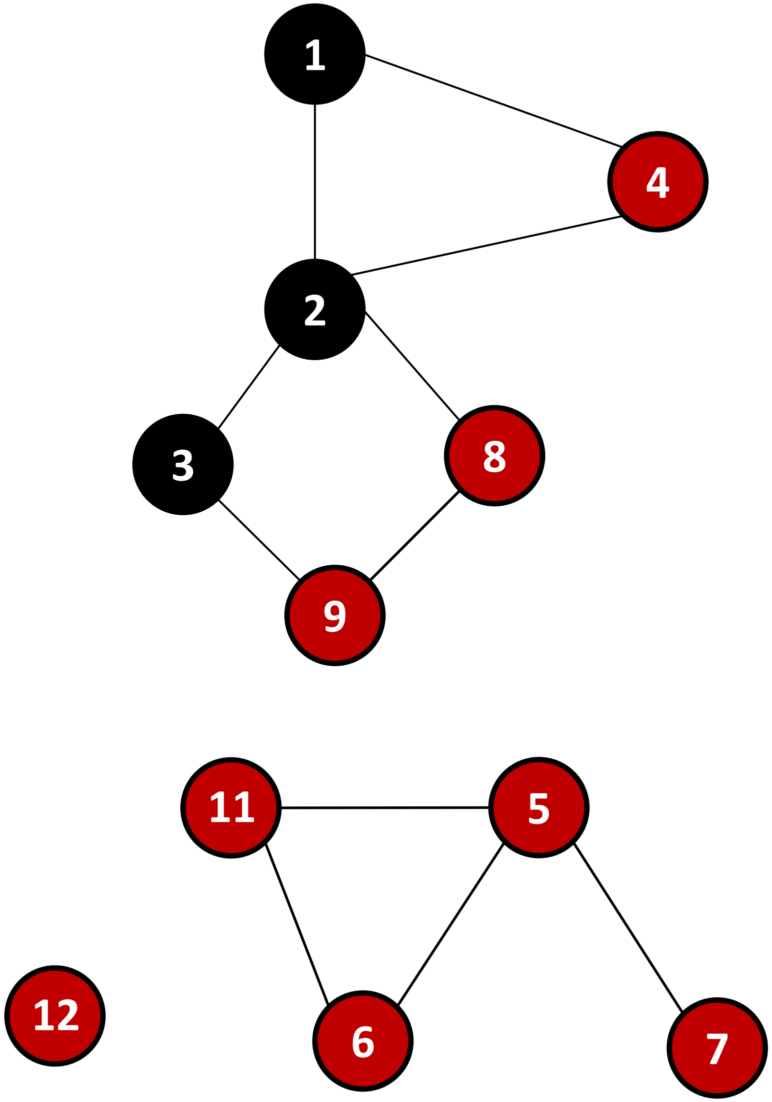
Depth-first search of a graph

```
DepthFirst()
   for each node in graph
        if node not visited
            DFSVisit(node)
```

```
DFSVisit(node)
   mark node visited
   for each unvisited neighbor, c, of node
      DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(2)
- DFSVisit(3)

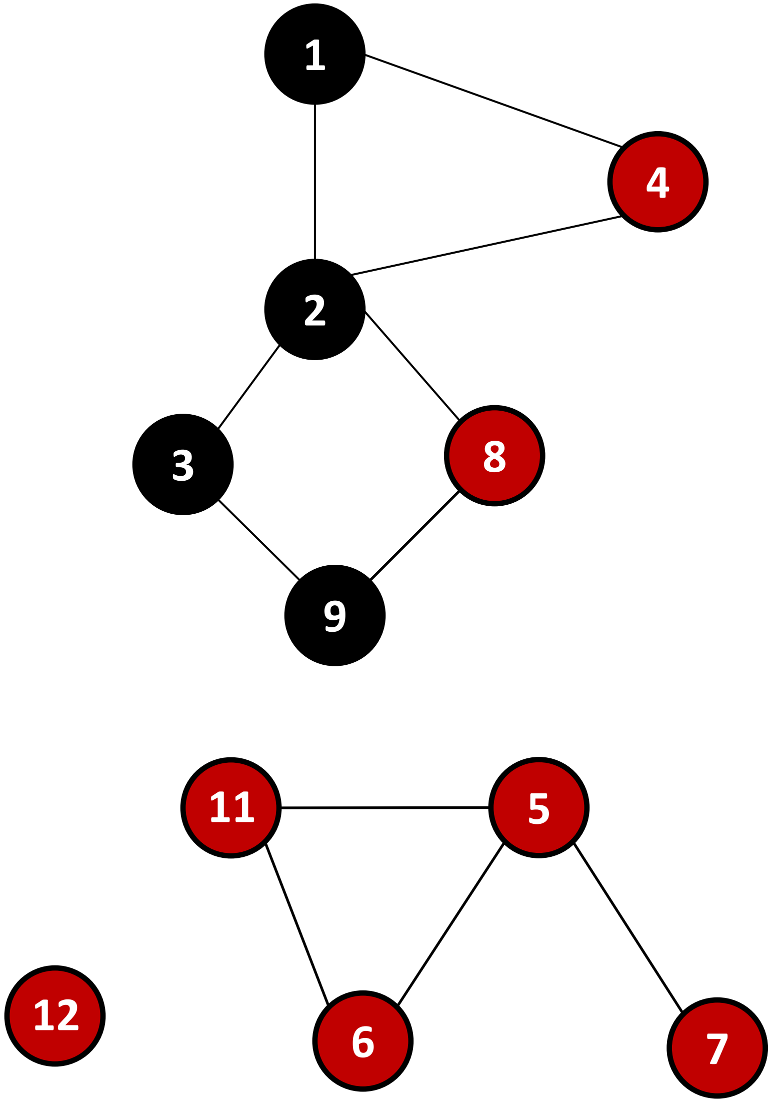


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(2)
- DFSVisit(3)
- DFSVisit(9)

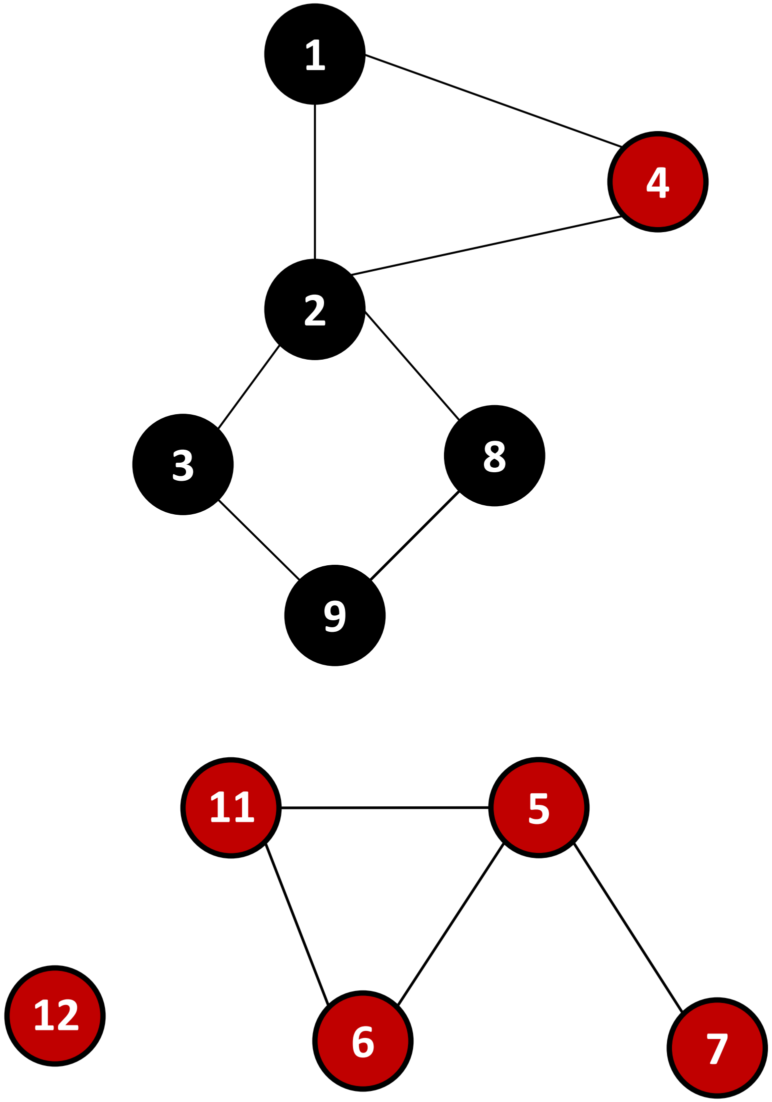


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(2)
- DFSVisit(3)
- DFSVisit(9)
- DFSVisit(8)

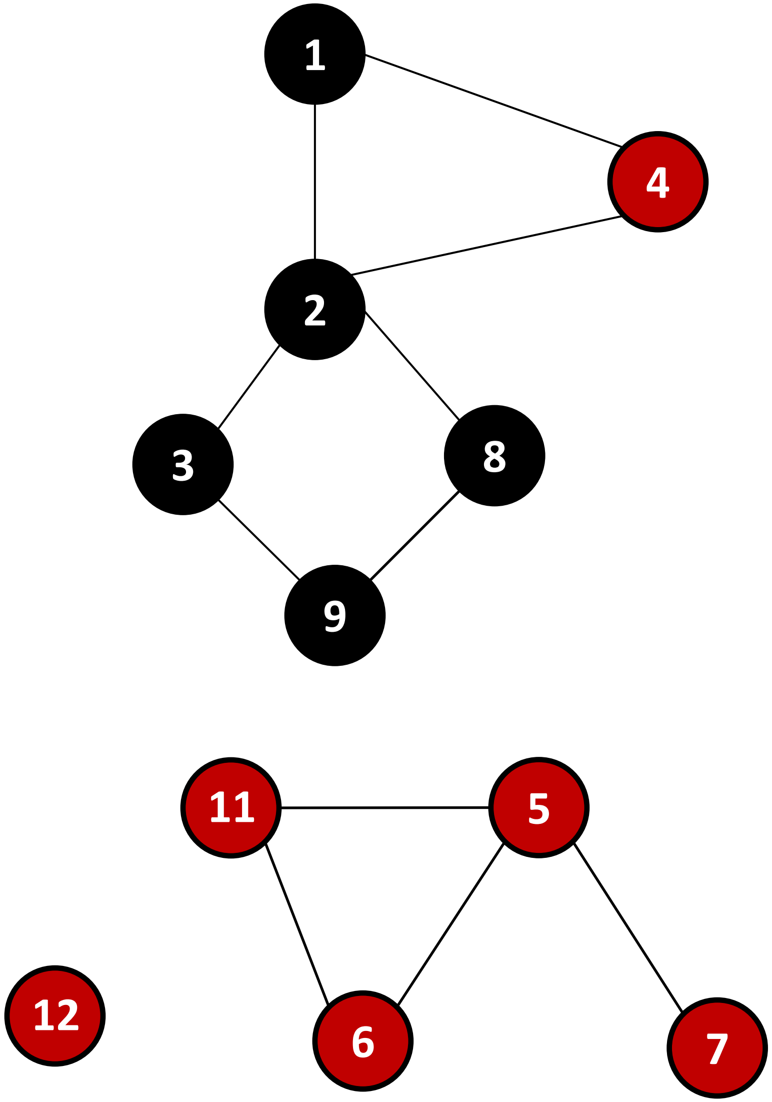


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
     if node not visited
        DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(2)
- DFSVisit(3)
- DFSVisit(9)

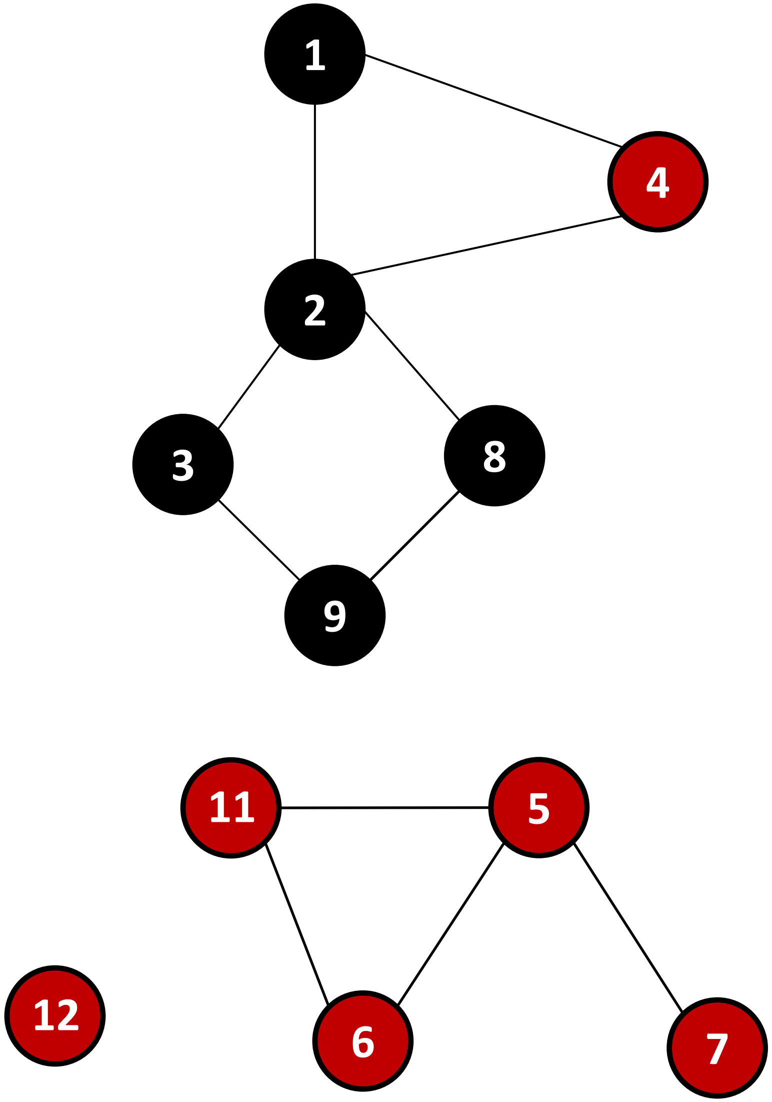


Depth-first search of a graph

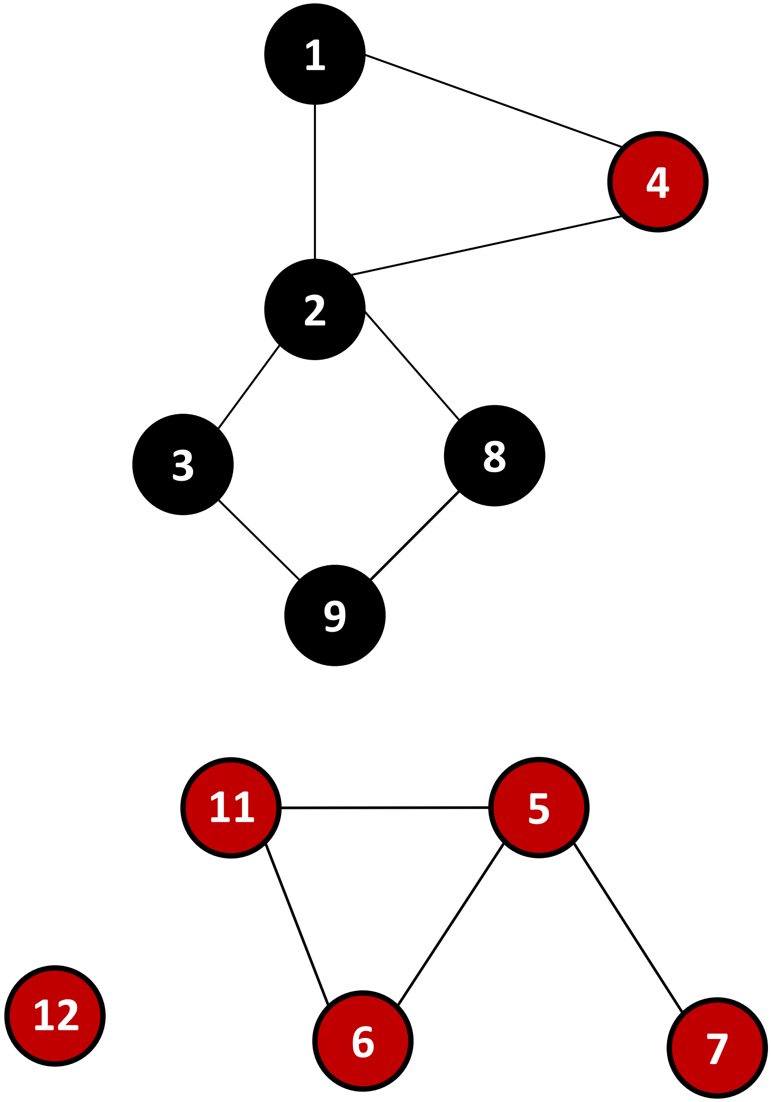
```
DepthFirst()
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(2)
- DFSVisit(3)



Depth-first search of a graph



```
DepthFirst()
   for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
   mark node visited
   for each unvisited neighbor, c, of node
      DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(2)

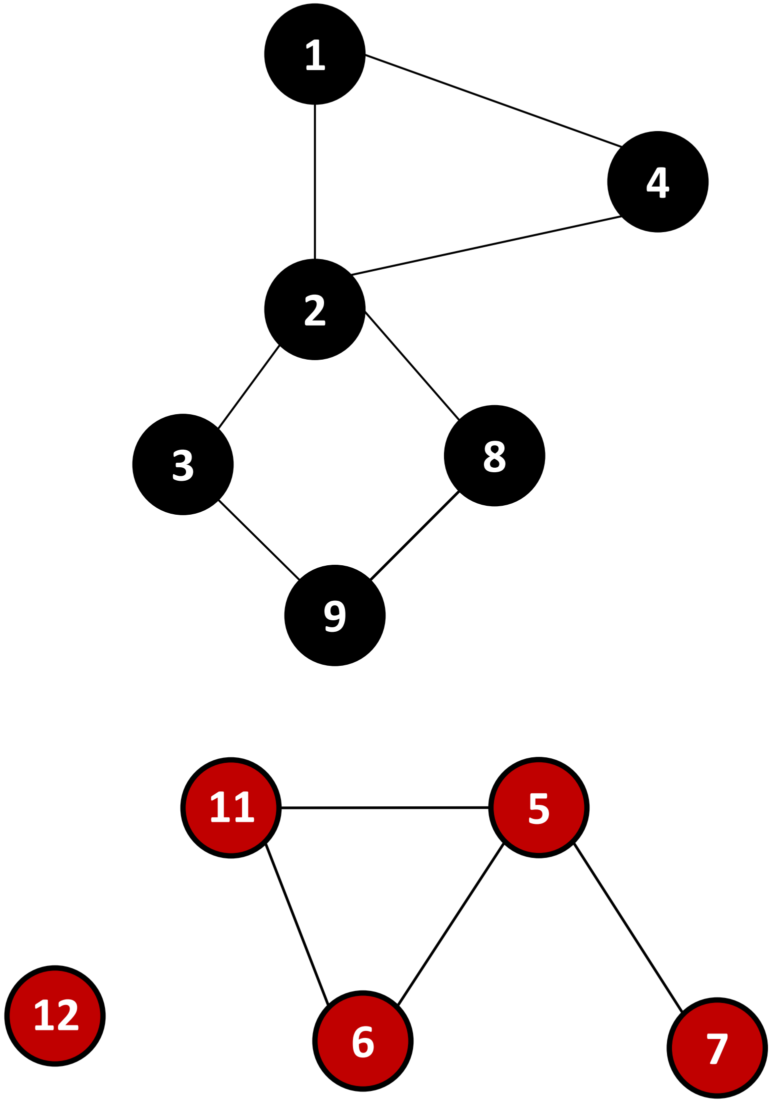
Depth-first search of a graph

```
DepthFirst()
   for each node in graph
       if node not visited
           DFSVisit(node)
```

```
DFSVisit(node)
   mark node visited
   for each unvisited neighbor, c, of node
      DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(2)
- DFSVisit(4)

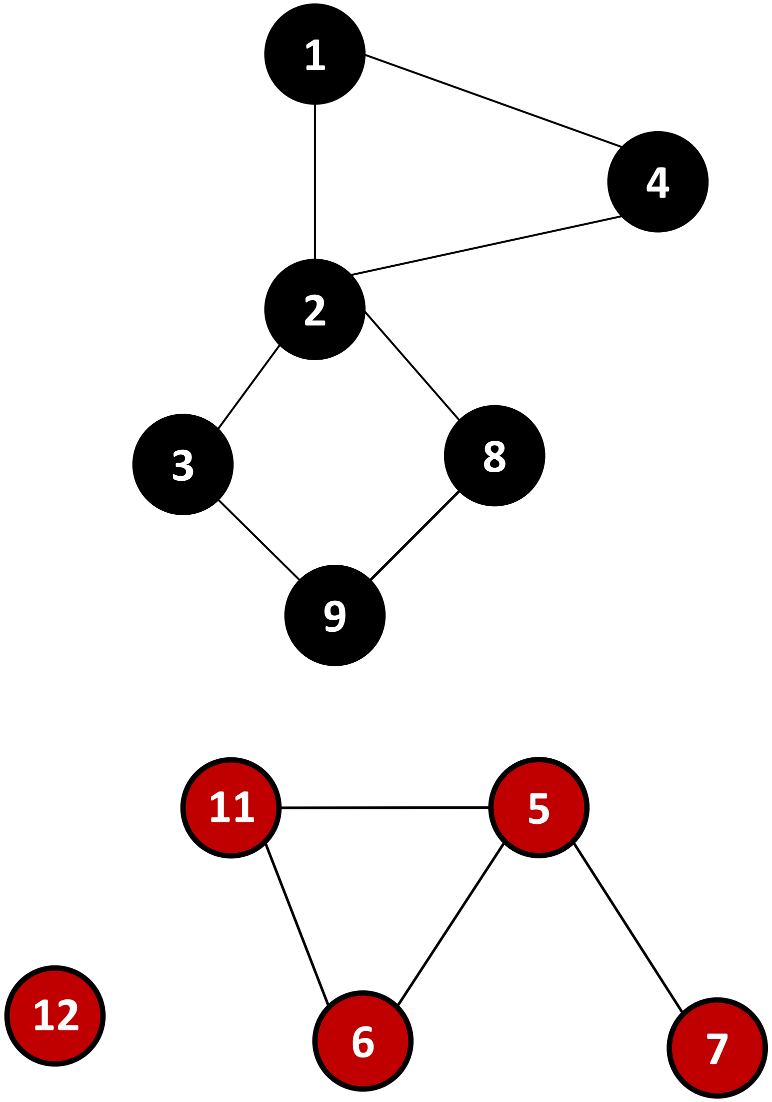


Depth-first search of a graph

```
DepthFirst()
   for each node in graph
        if node not visited
            DFSVisit(node)

DFSVisit(node)
   mark node visited
   for each unvisited neighbor, c, of node
      DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(2)



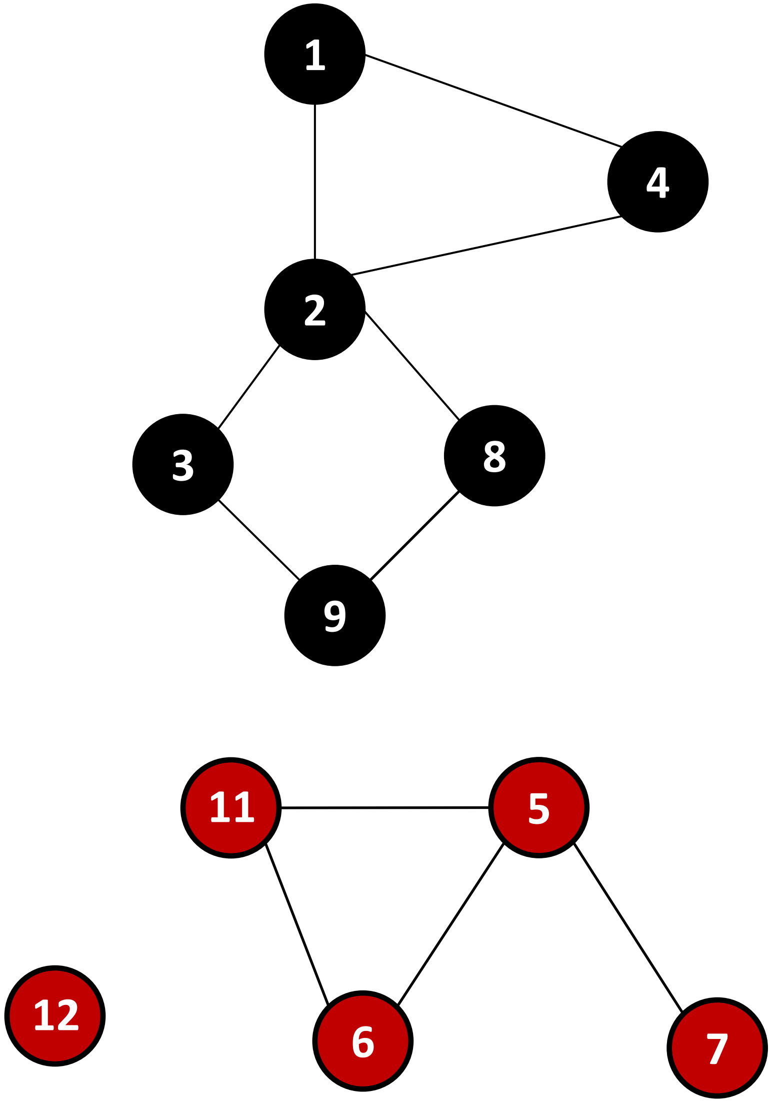
Depth-first search of a graph

```
DepthFirst()
  for each node in graph
     if node not visited
        DFSVisit(node)
```

```
DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)



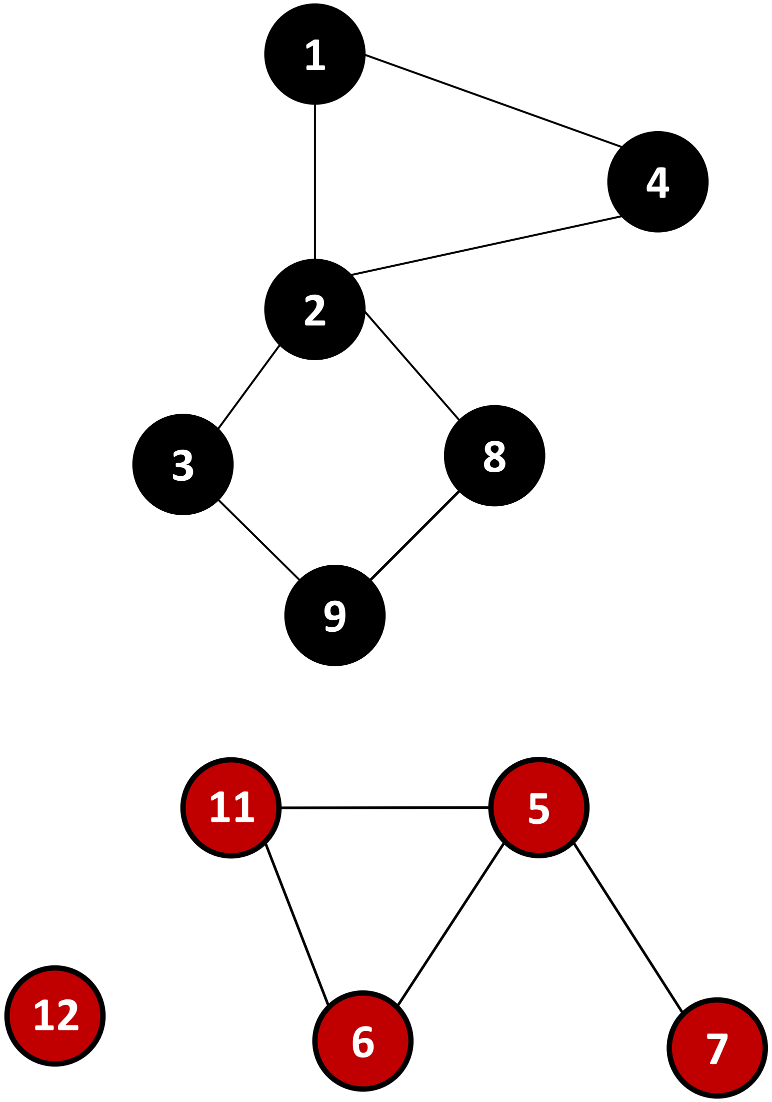
Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
         DFSVisit(node)
```

```
DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)
- DFSVisit(4)



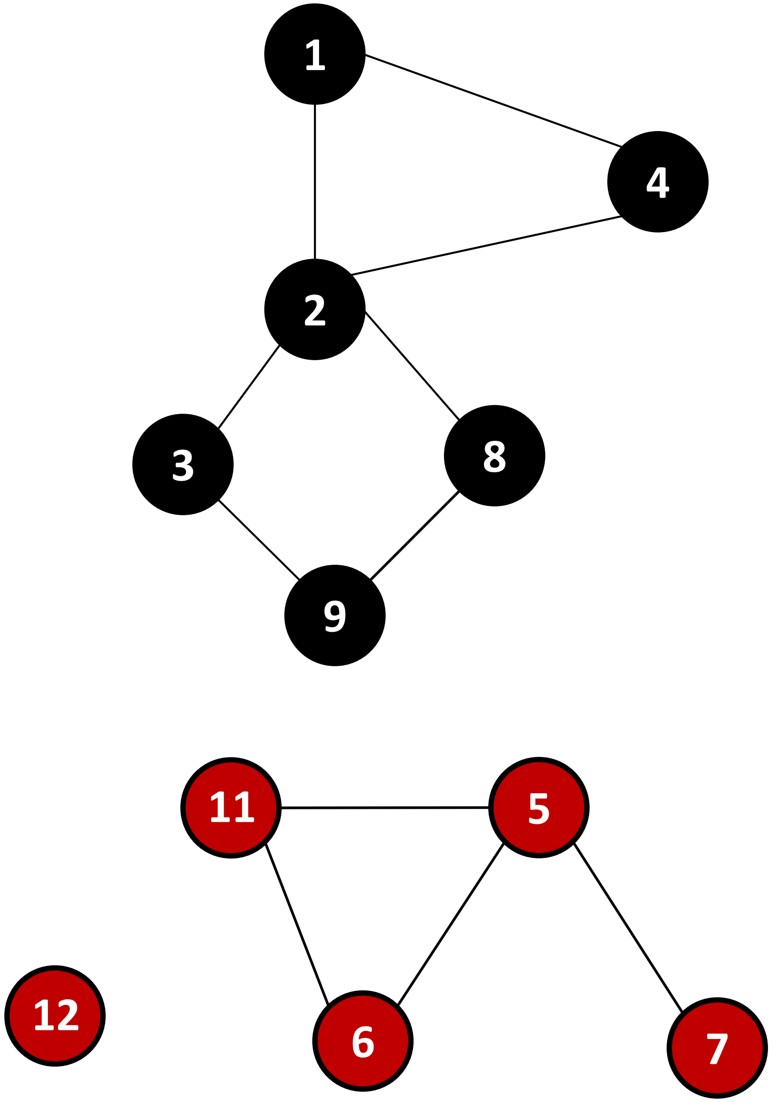
Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
         DFSVisit(node)
```

```
DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(1)

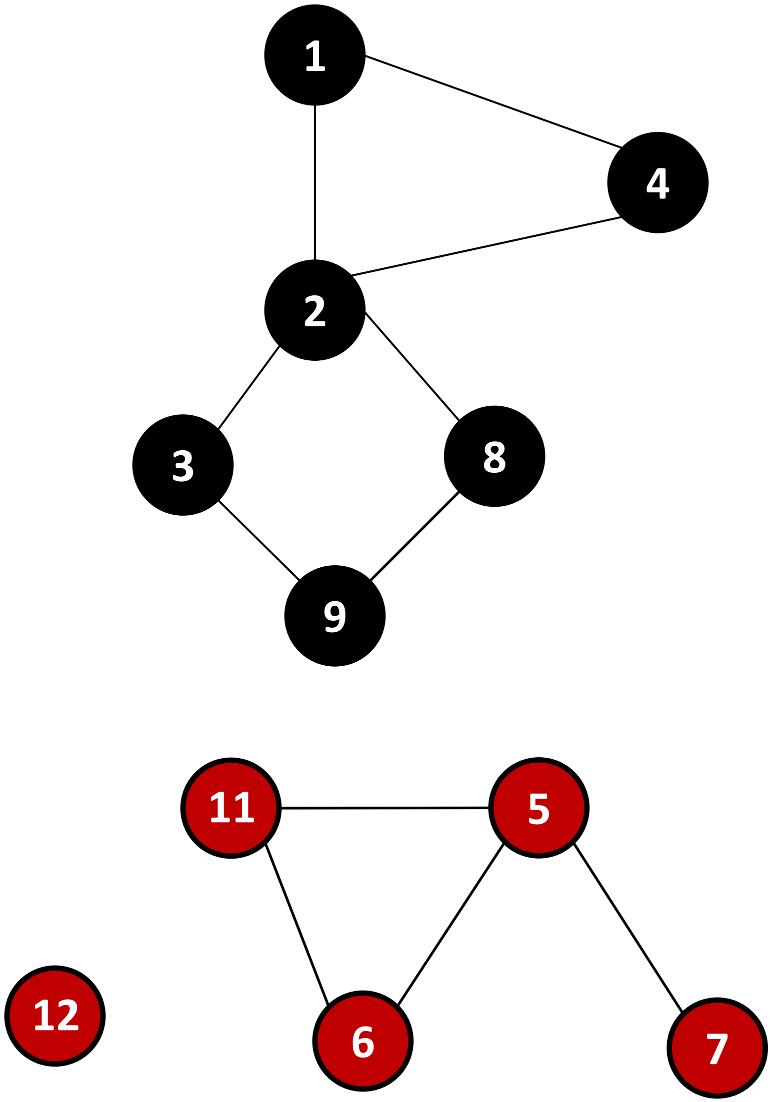


Depth-first search of a graph

```
DepthFirst()
   for each node in graph
        if node not visited
            DFSVisit(node)

DFSVisit(node)
   mark node visited
   for each unvisited neighbor, c, of node
      DFSVisit(c)
```

Call Stack:

- DepthFirst()

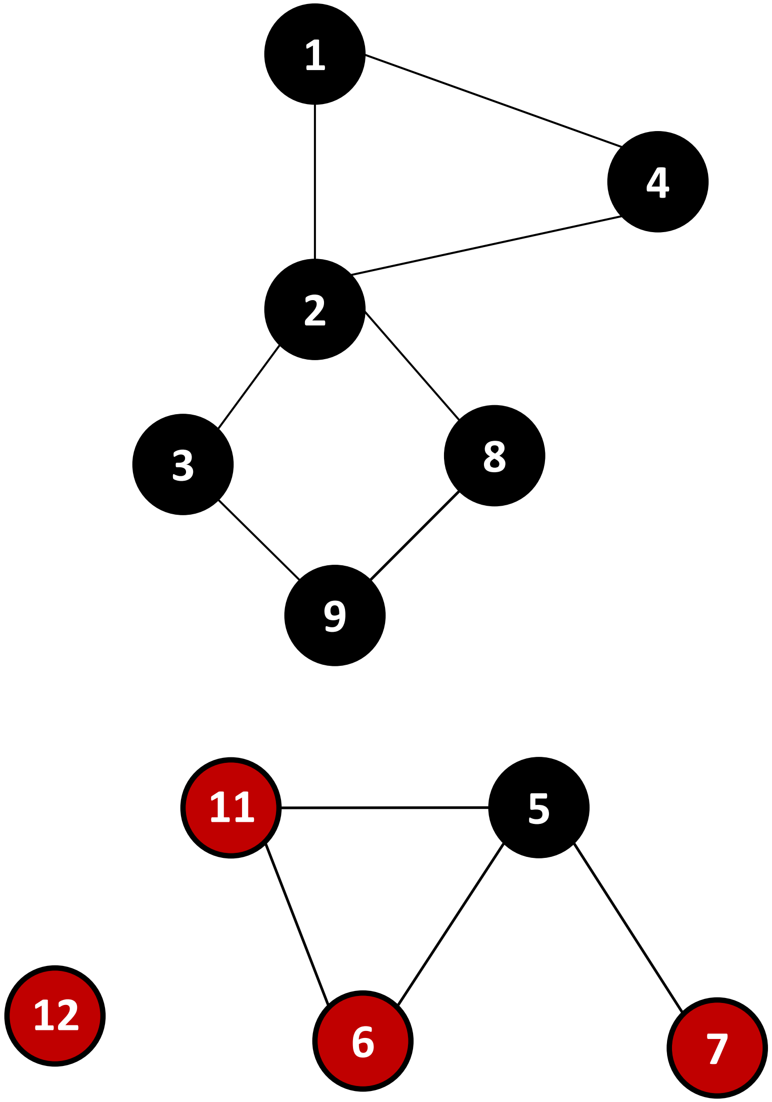


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(5)

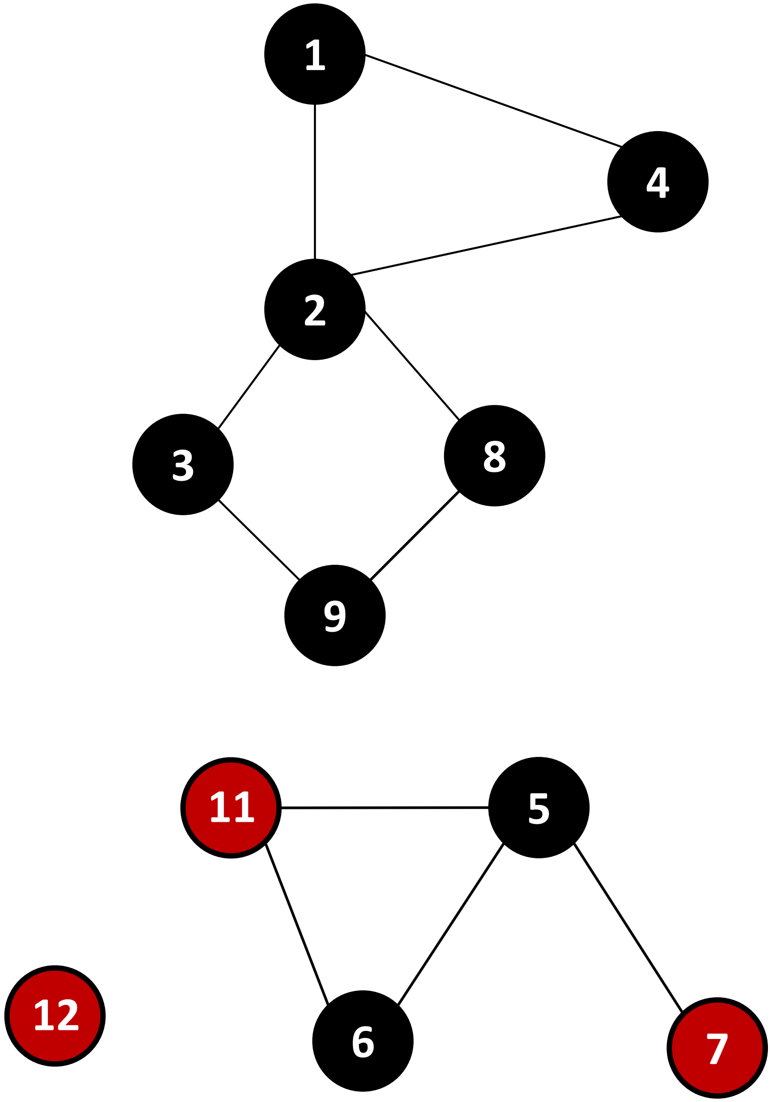


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(5)
- DFSVisit(6)



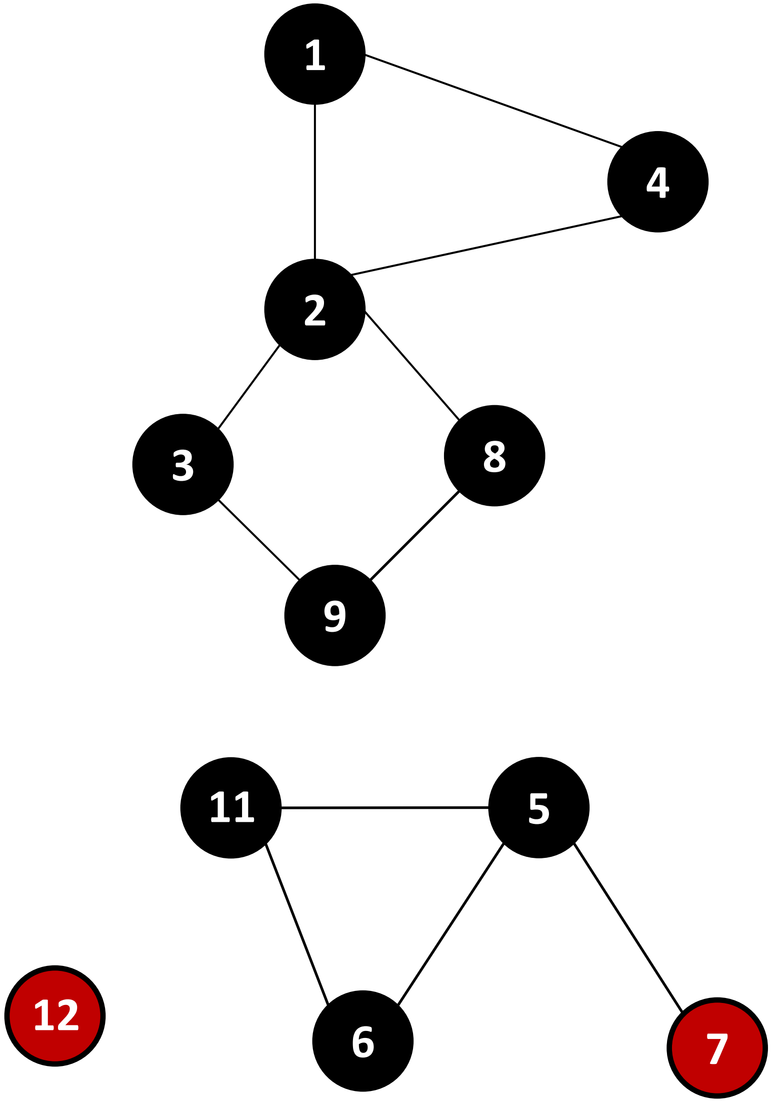
Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
         DFSVisit(node)
```

```
DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(5)
- DFSVisit(6)
- DFSVisit(11)

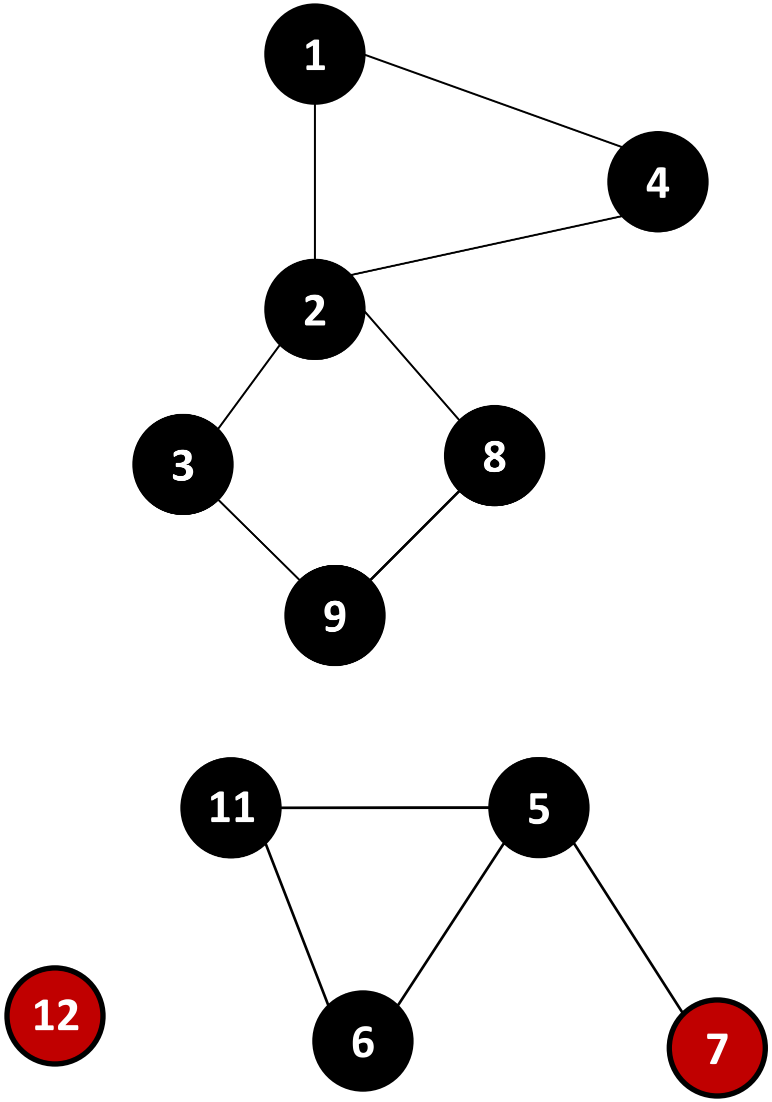


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(5)
- DFSVisit(6)

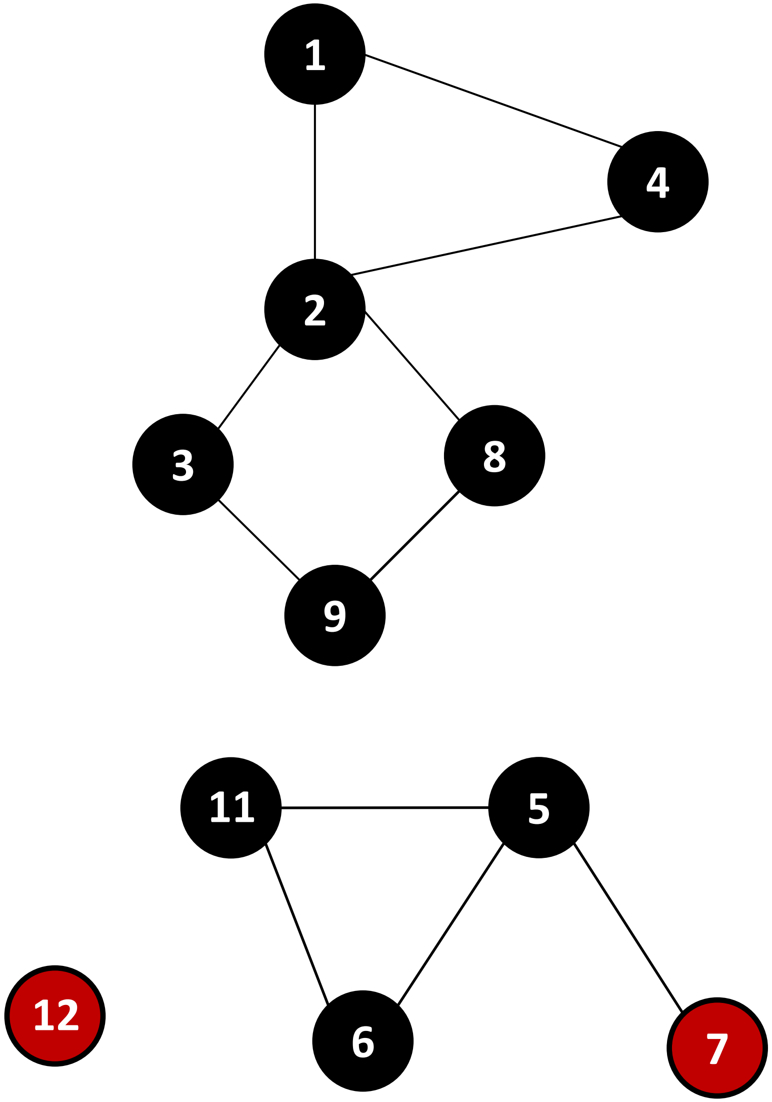


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(5)

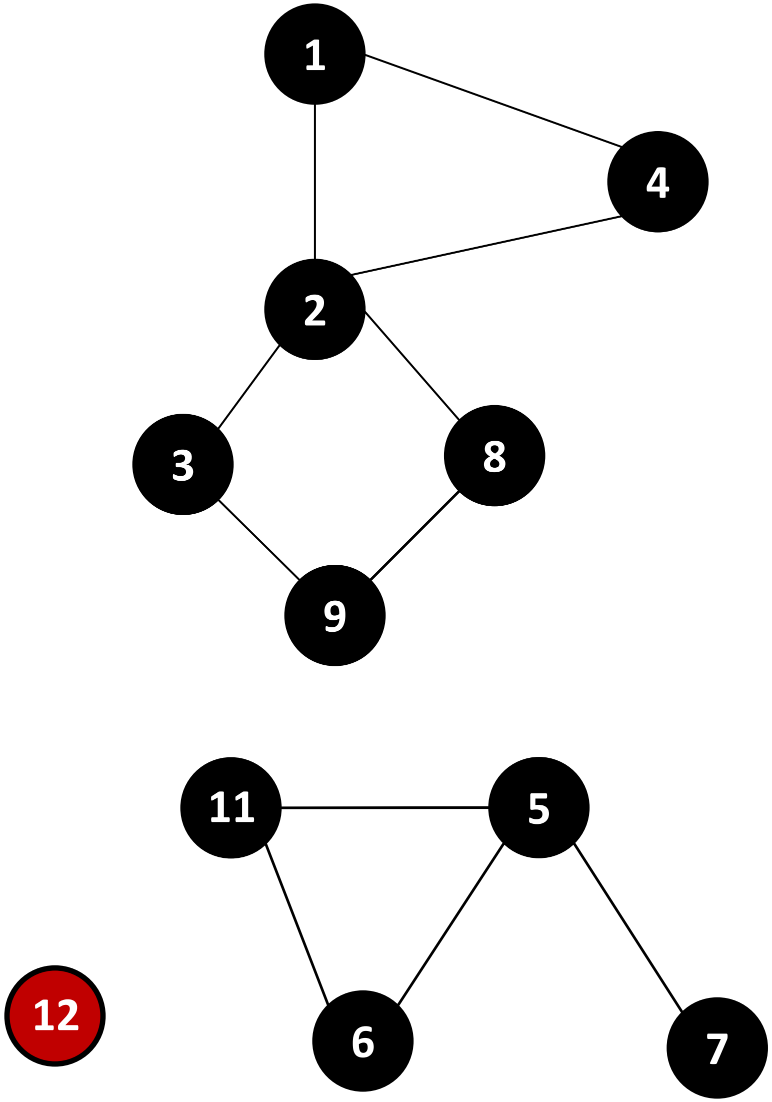


Depth-first search of a graph

```
DepthFirst()
    for each node in graph
        if node not visited
            DFSVisit(node)

DFSVisit(node)
    mark node visited
    for each unvisited neighbor, c, of node
        DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(5)
- DFSVisit(7)

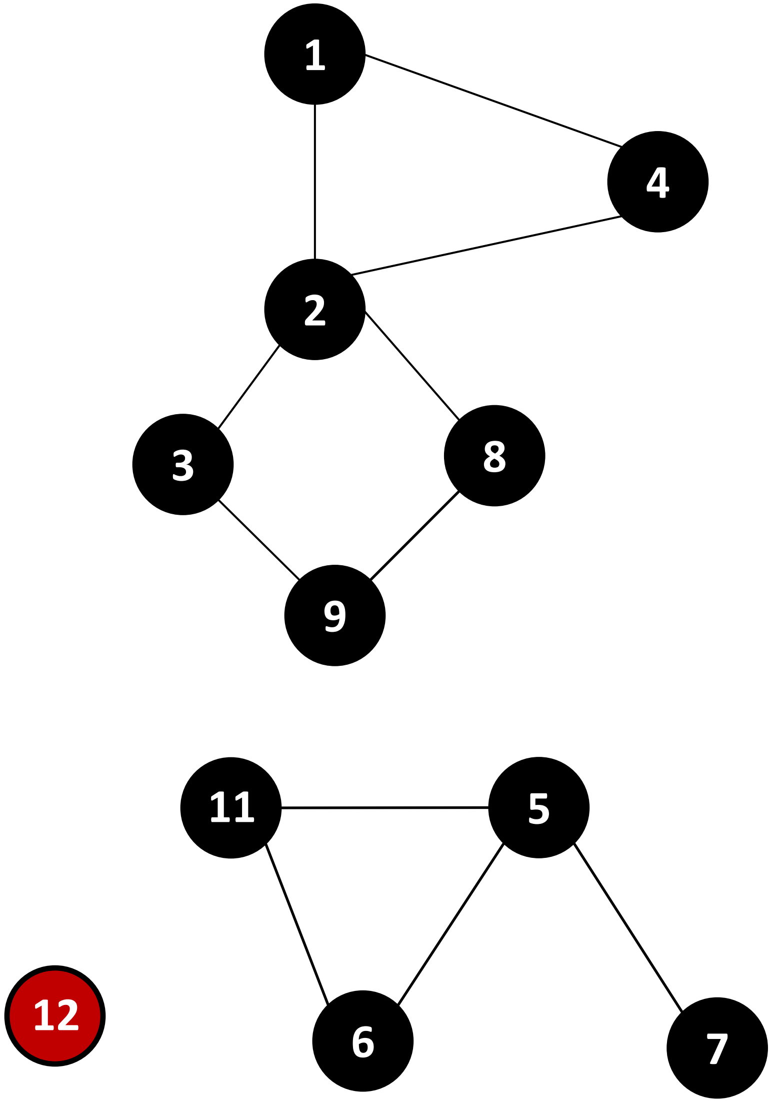


Depth-first search of a graph

```
DepthFirst()
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(5)

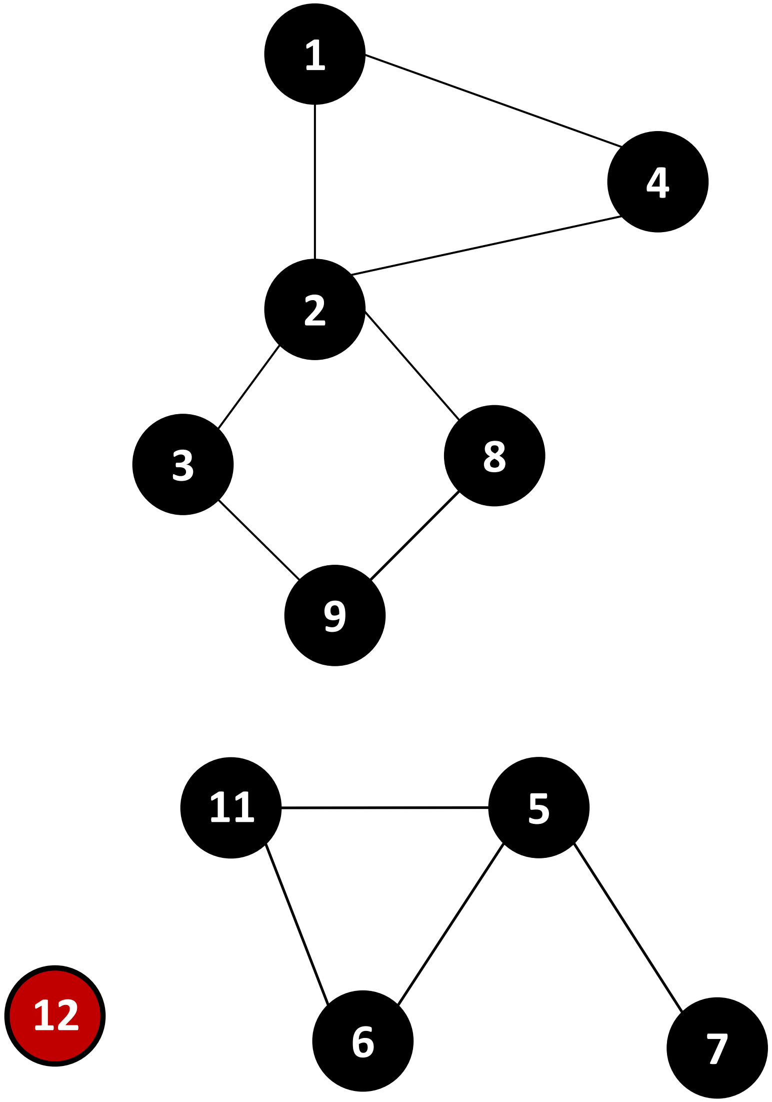


Depth-first search of a graph

```
DepthFirst()
   for each node in graph
        if node not visited
            DFSVisit(node)

DFSVisit(node)
   mark node visited
   for each unvisited neighbor, c, of node
      DFSVisit(c)
```

Call Stack:

- DepthFirst()



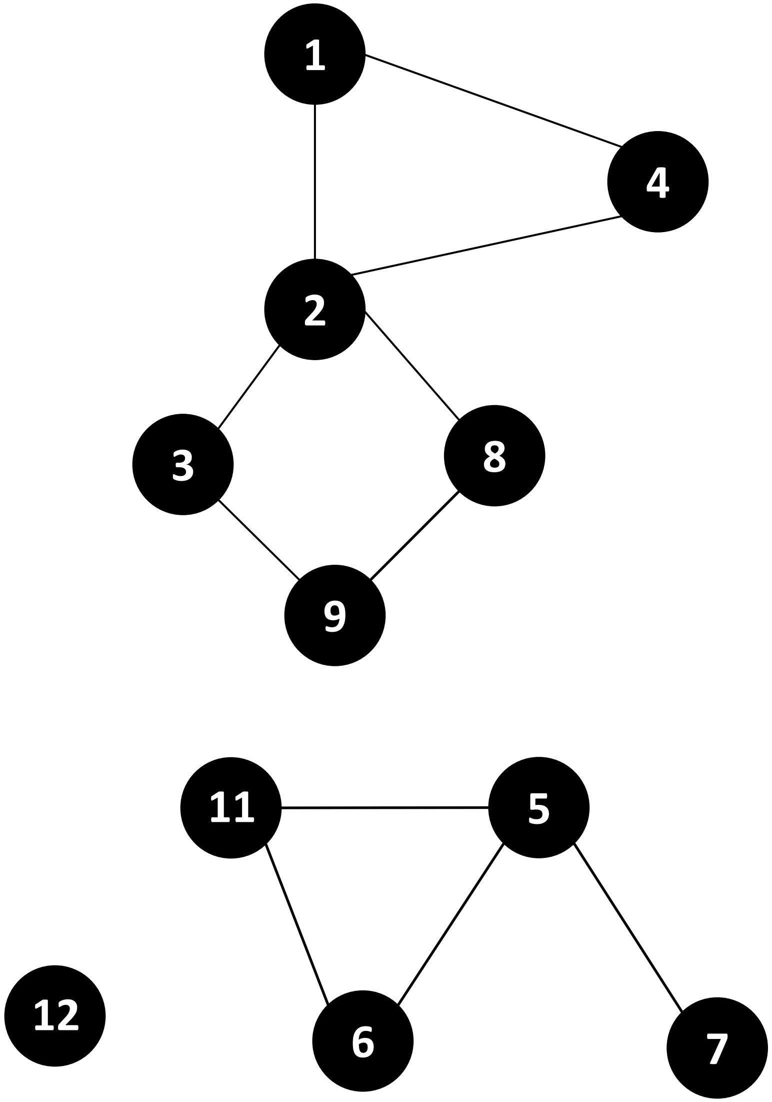
Depth-first search of a graph

```
DepthFirst()
  for each node in graph
     if node not visited
        DFSVisit(node)
```

```
DFSVisit(node)
  mark node visited
  for each unvisited neighbor, c, of node
    DFSVisit(c)
```

Call Stack:

- DepthFirst()
- DFSVisit(12)

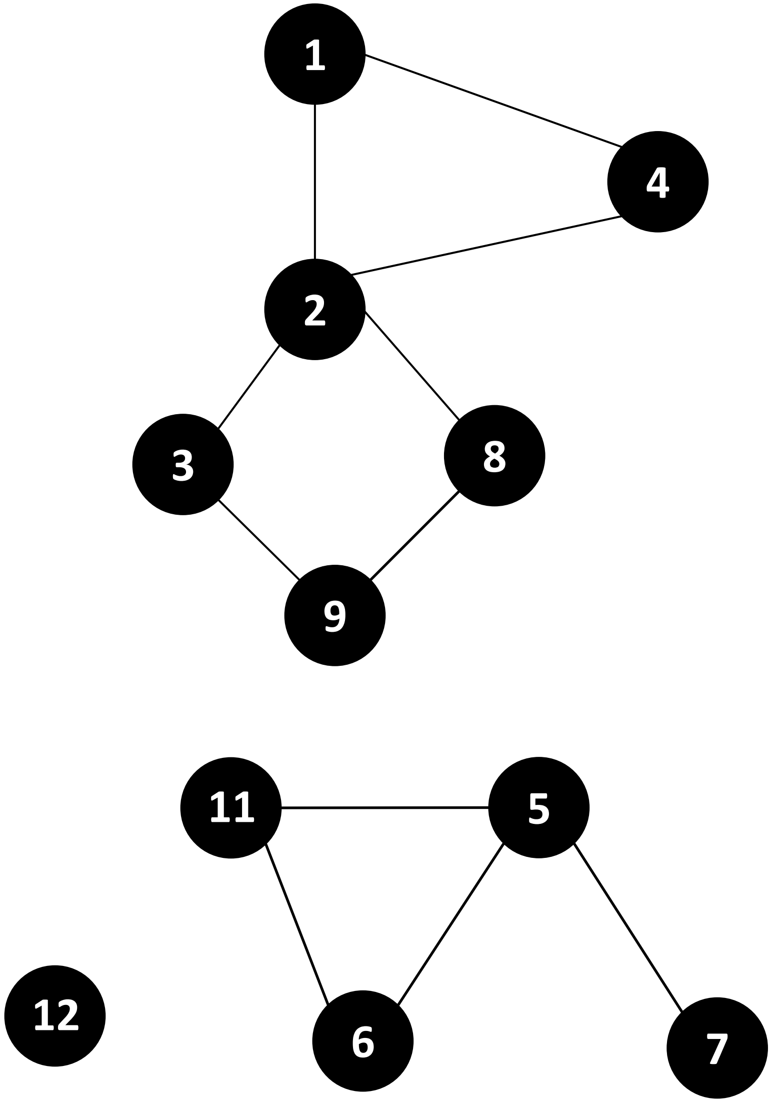


Depth-first search of a graph

```
DepthFirst()
   for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
   mark node visited
   for each unvisited neighbor, c, of node
      DFSVisit(c)
```

Call Stack:

- DepthFirst()

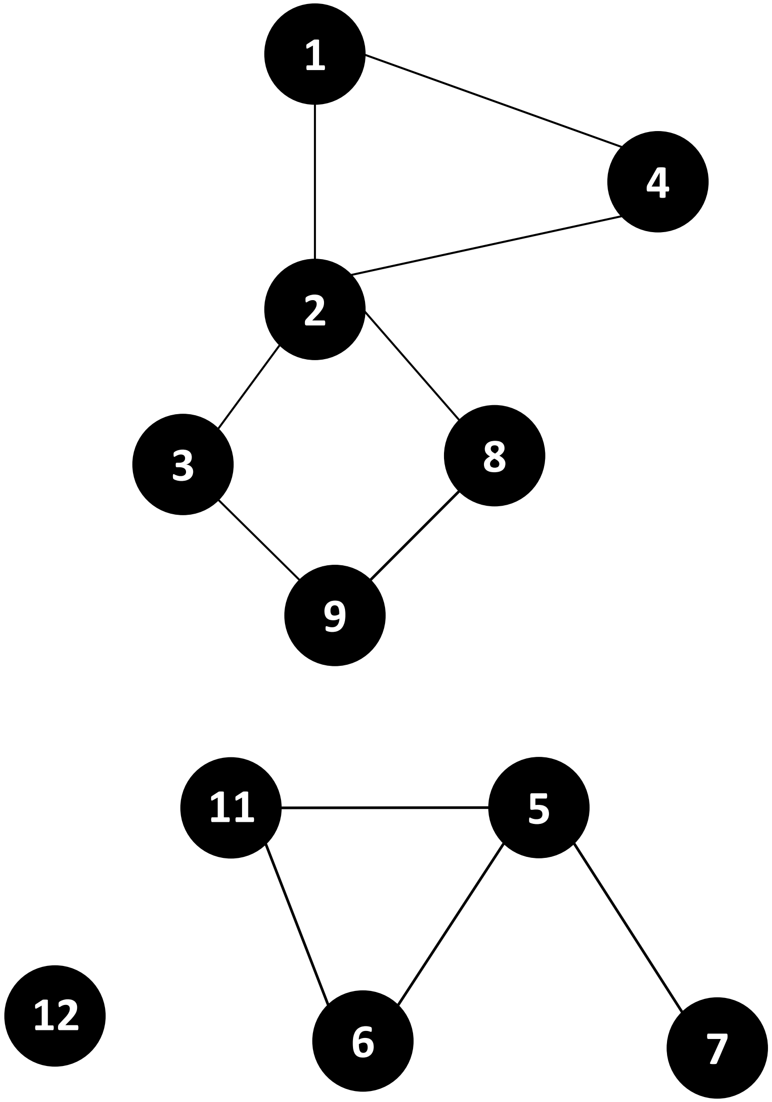


Depth-first search of a graph

```
DepthFirst()
   for each node in graph
        if node not visited
            DFSVisit(node)

DFSVisit(node)
   mark node visited
   for each unvisited neighbor, c, of node
      DFSVisit(c)
```

Call Stack:

- Done!



Analysis:

how many times are these lines run?

```
DepthFirst()
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  for each neighbor, c, of node
    if c not visited
       DFSVisit(c)
```

Analysis

```
DepthFirst()
  for each node in graph
      if node not visited
          DFSVisit(node)                    |  Once per vertex


DFSVisit(node)
  mark node visited                         |  Once per vertex
  for each neighbor, c, of node
    if c not visited                        |  Once per edge
        DFSVisit(c)                         |  Once per vertex
```

Analysis

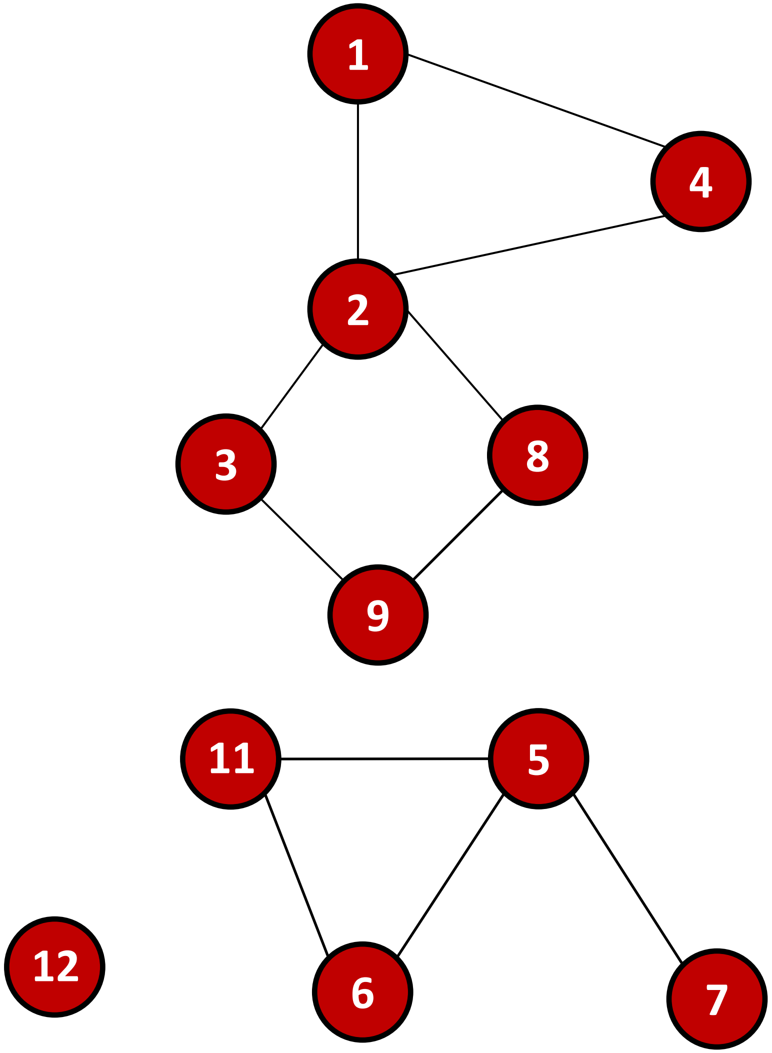
```
DepthFirst()
  for each node in graph
      if node not visited
         DFSVisit(node)                    |  $O(V)$

DFSVisit(node)
  mark node visited                        |  $O(V)$
  for each neighbor, c, of node
    if c not visited                       |  $O(E)$
       DFSVisit(c)                         |  $O(V)$
```

Analysis: $O(V + E)$ time

```
DepthFirst()
  for each node in graph
     if node not visited
        DFSVisit(node)                          |   $O(V)$

DFSVisit(node)
  mark node visited                             |   $O(V)$
  for each neighbor, c, of node
    if c not visited                            |   $O(E)$
       DFSVisit(c)                              |   $O(V)$
```

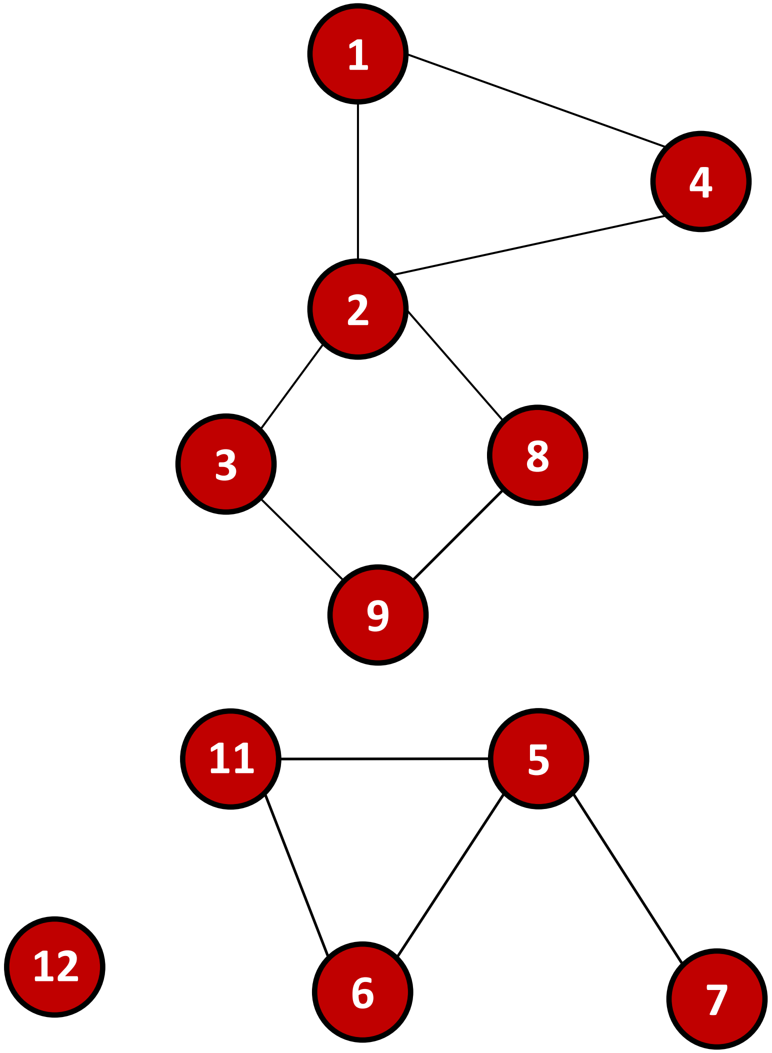
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
            neighbor, n, of node
      LCCVisit(n, c)
```



Example application: connected component analysis

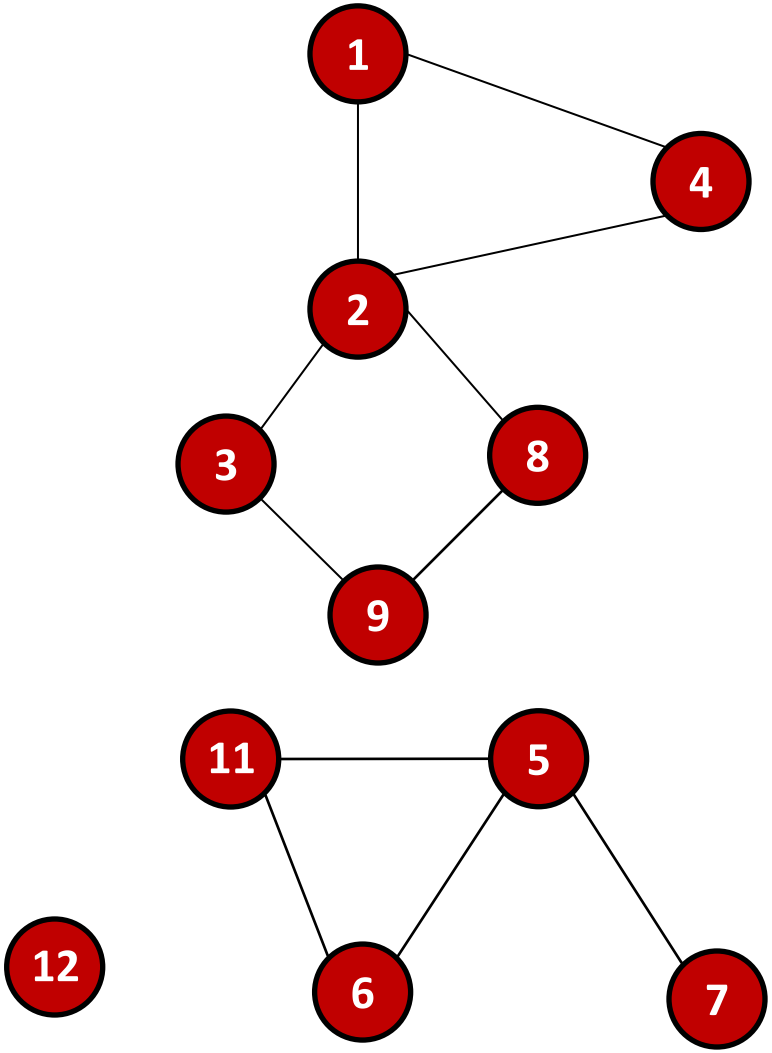
```
LabelConnectedComponents()
   component = 0
   for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
            neighbor, n, of node
      LCCVisit(n, c)
```



Example application:
connected component analysis

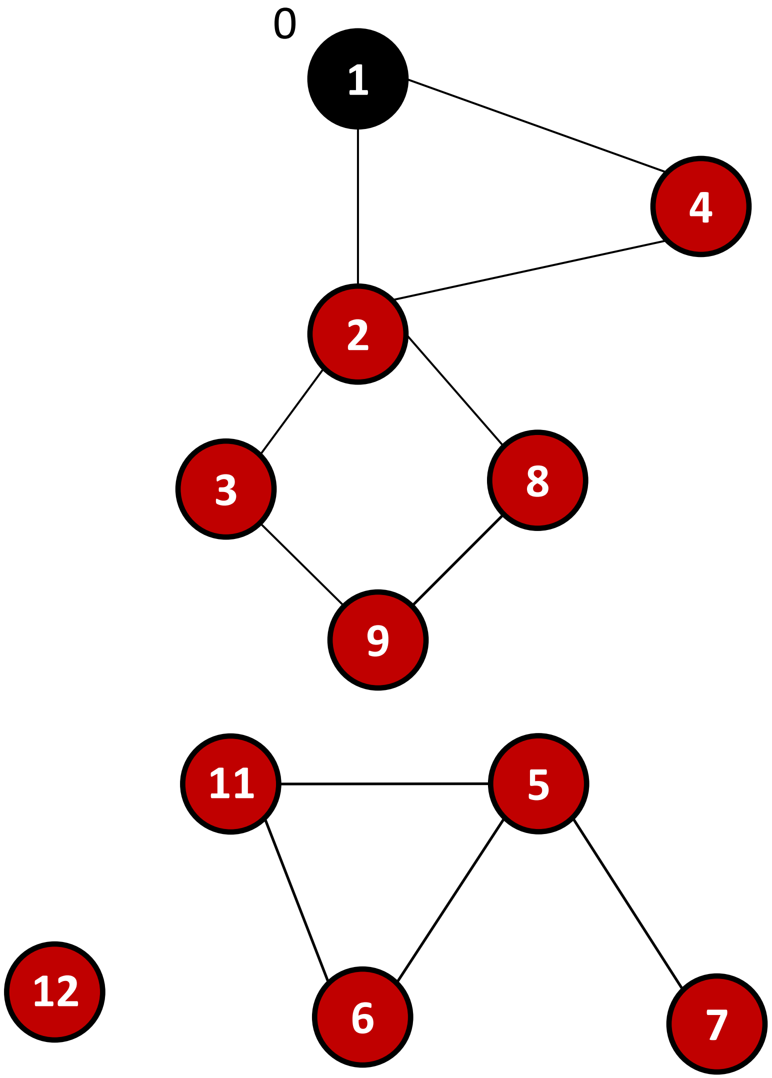
```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
          **LCCVisit(node, component)**
          component++
```

```
LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
            neighbor, n, of node
    LCCVisit(n, c)
```



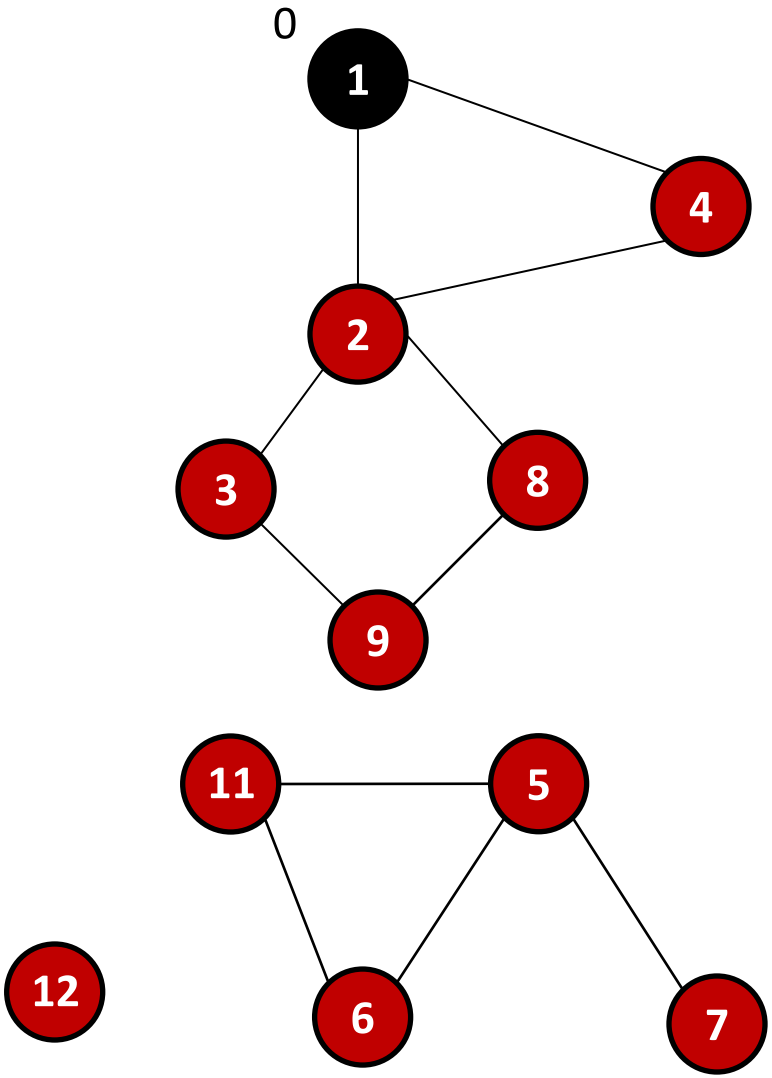
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
            neighbor, n, of node
      LCCVisit(n, c)
```



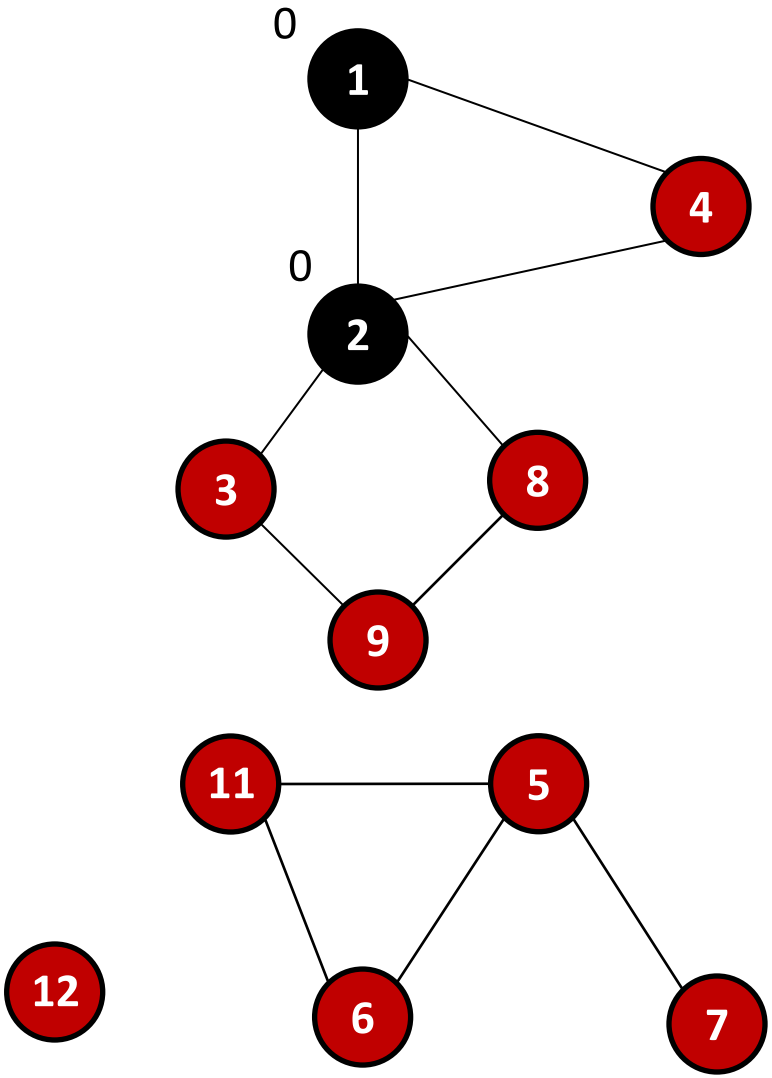
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
              neighbor, n, of node
      LCCVisit(n, c)
```



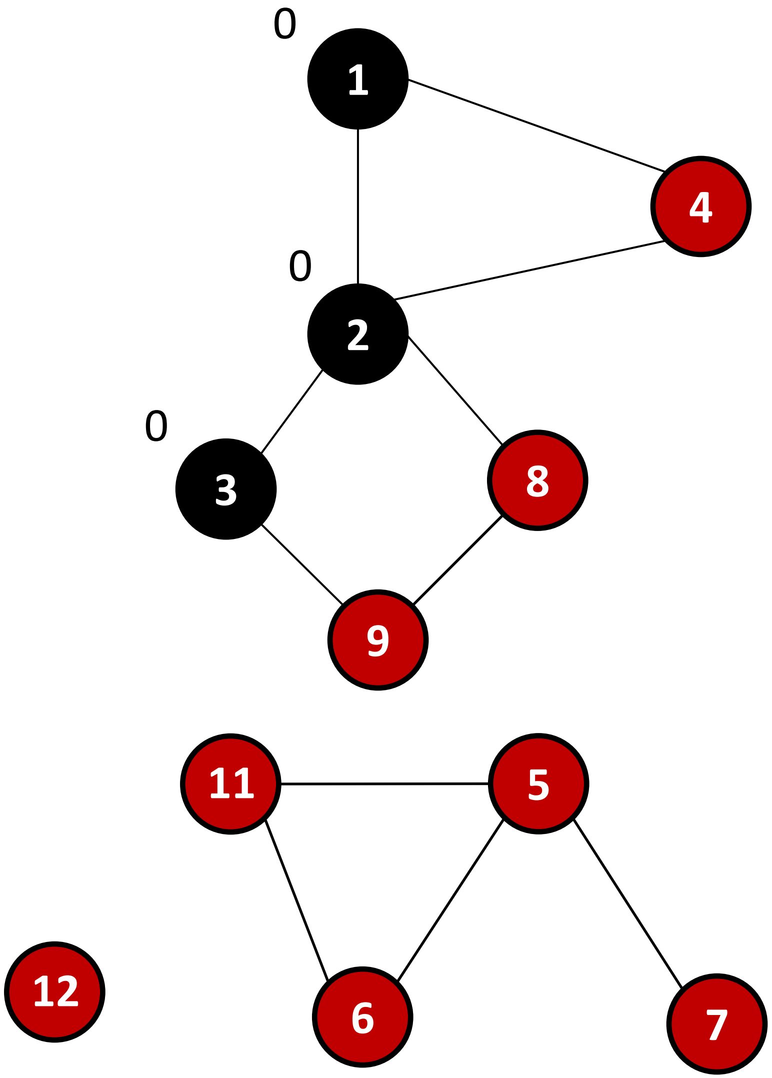
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   **mark node visited**
   **node.component = c**
   for each unvisited
              neighbor, n, of node
      LCCVisit(n, c)
```



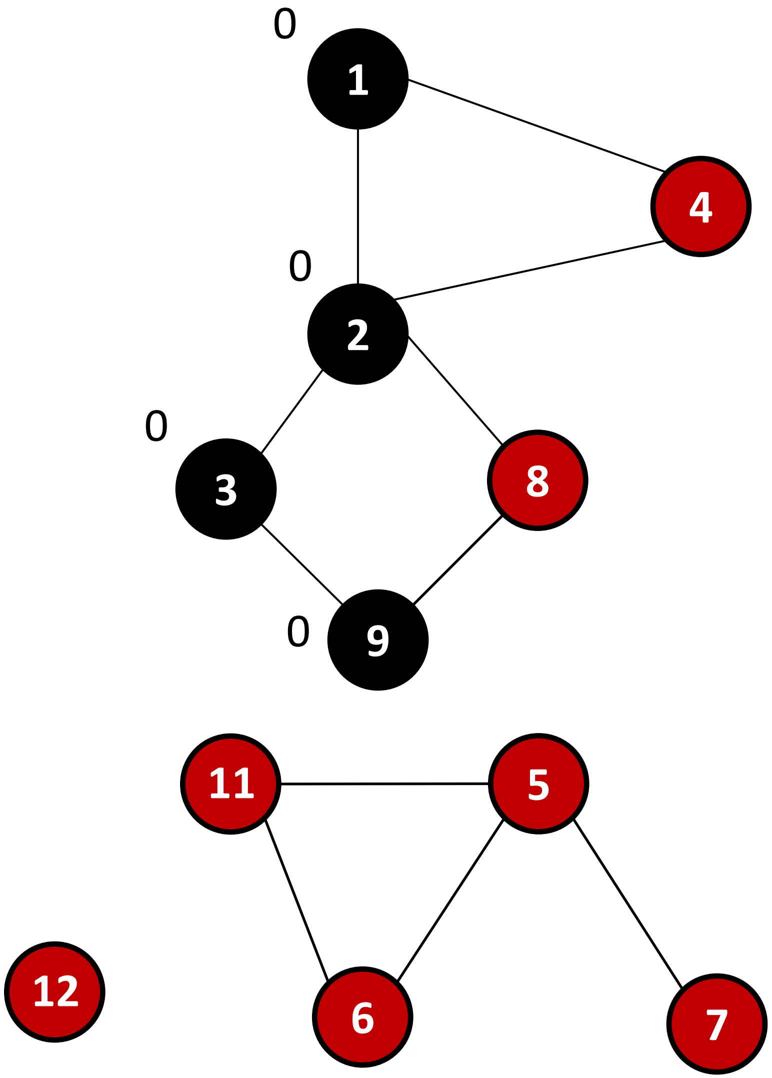
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
               neighbor, n, of node
      LCCVisit(n, c)
```



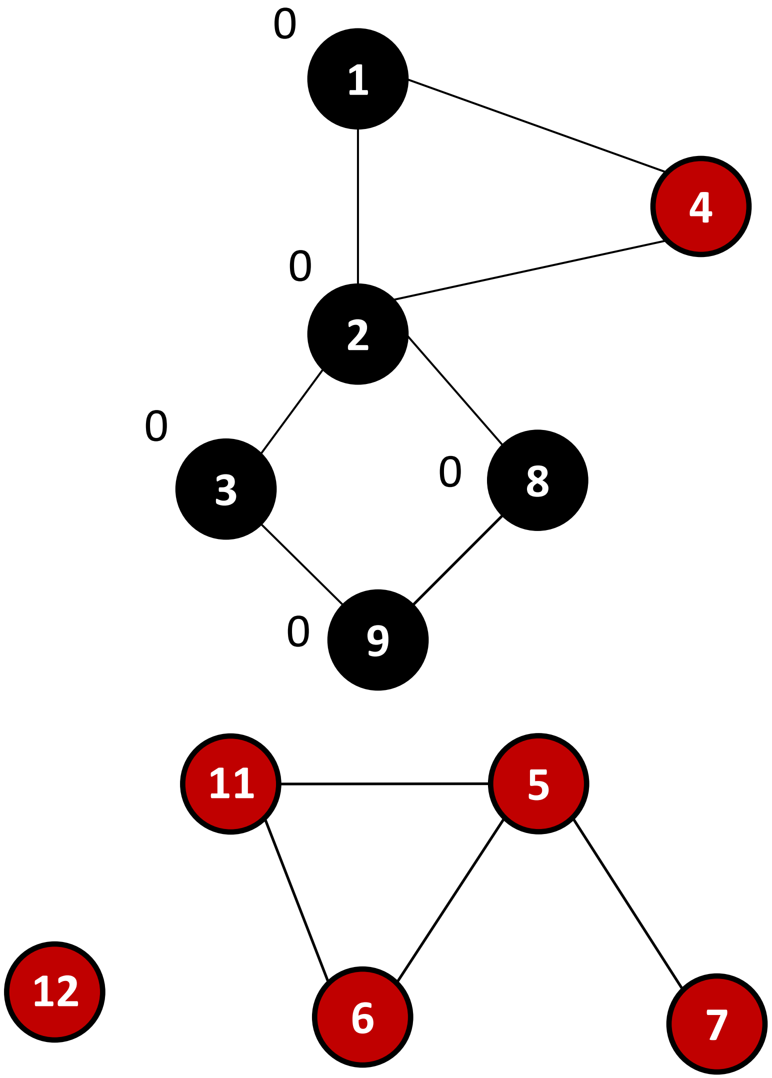
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
              neighbor, n, of node
      LCCVisit(n, c)
```



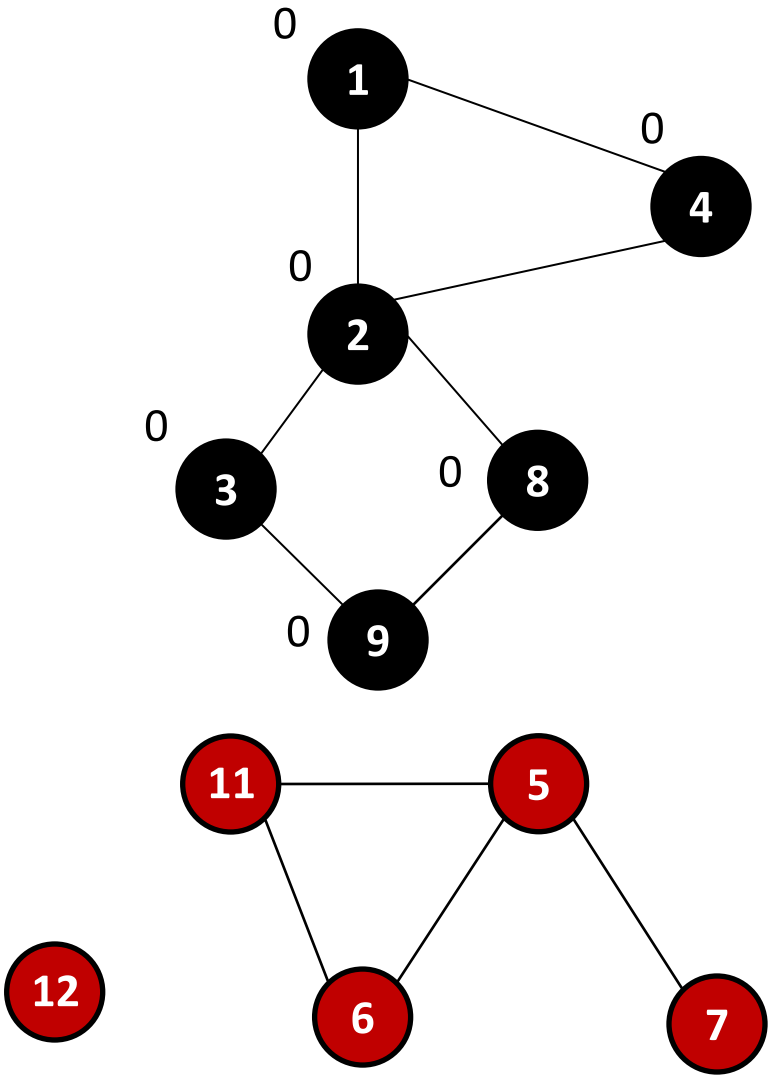
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
               neighbor, n, of node
      LCCVisit(n, c)
```



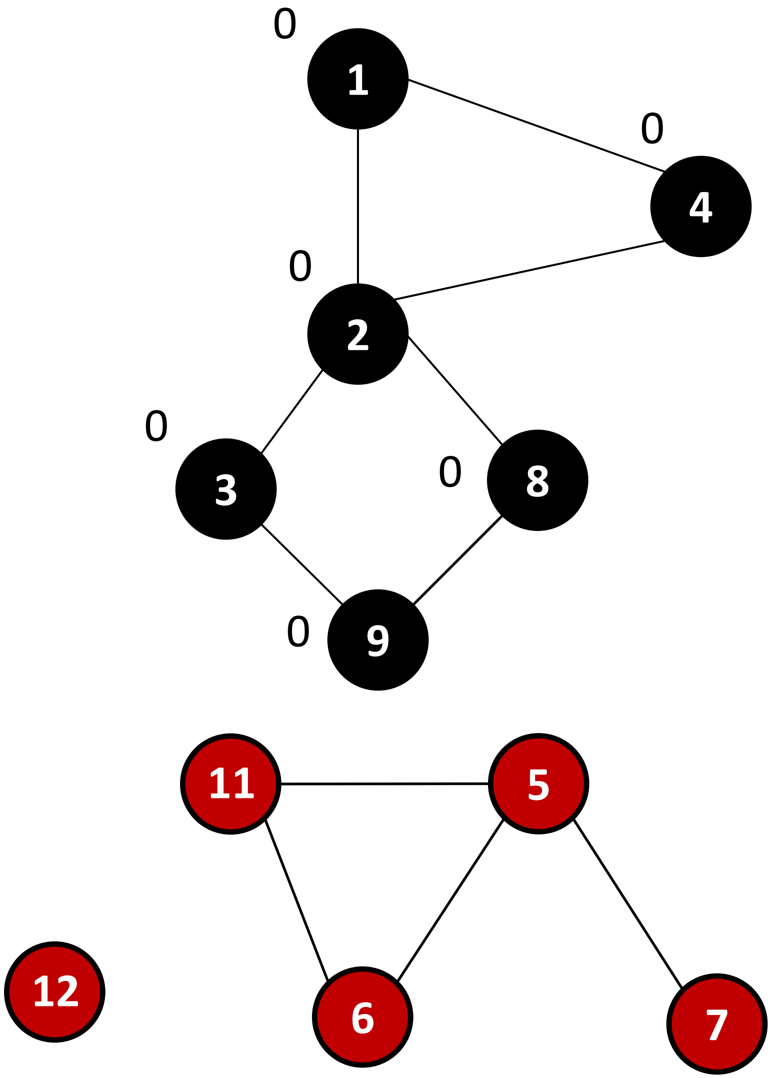
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
              neighbor, n, of node
      LCCVisit(n, c)
```



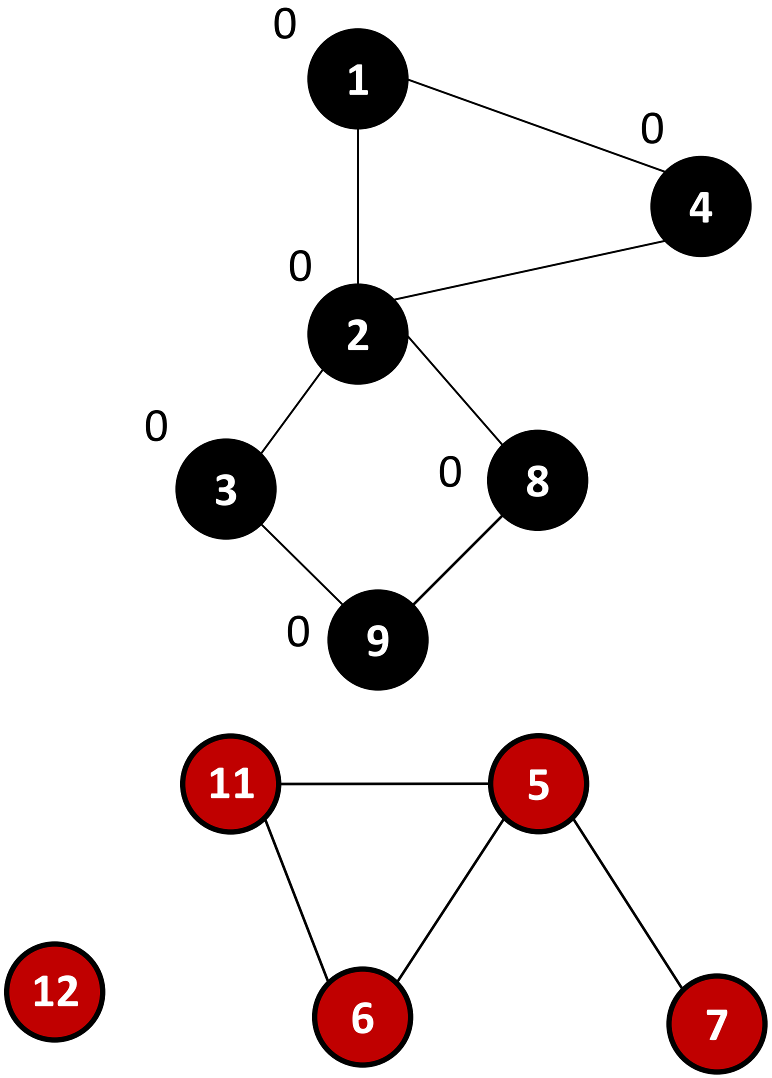
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
              neighbor, n, of node
      LCCVisit(n, c)
```



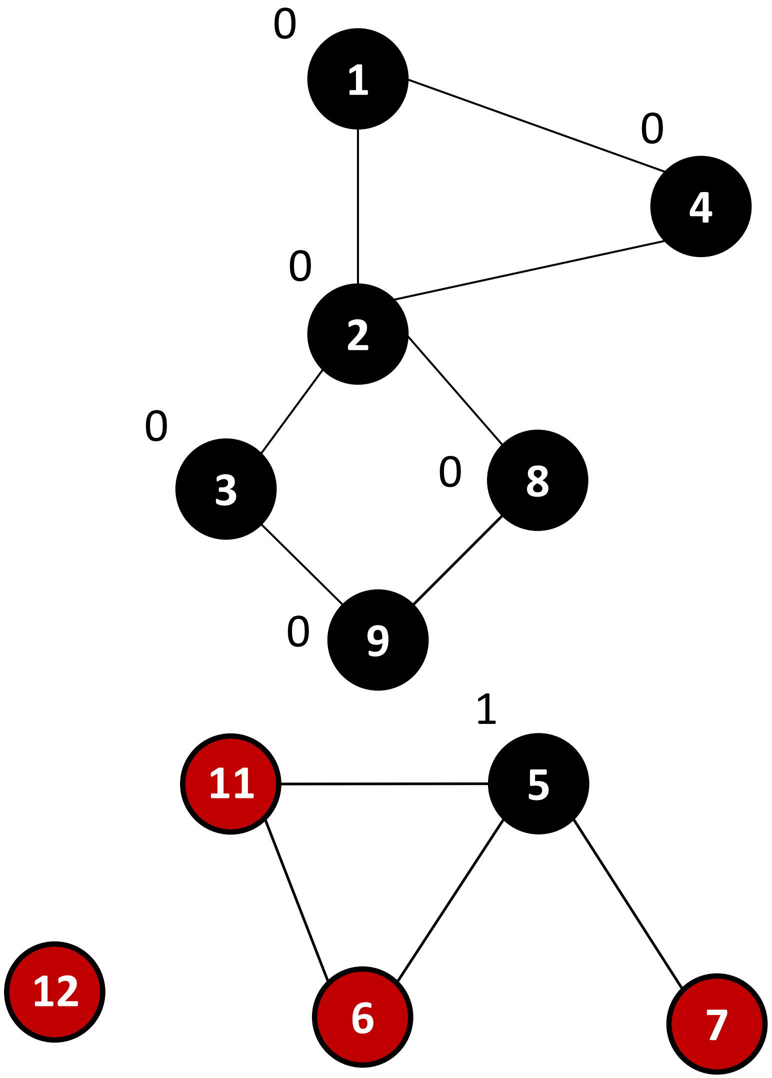
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
            neighbor, n, of node
      LCCVisit(n, c)
```



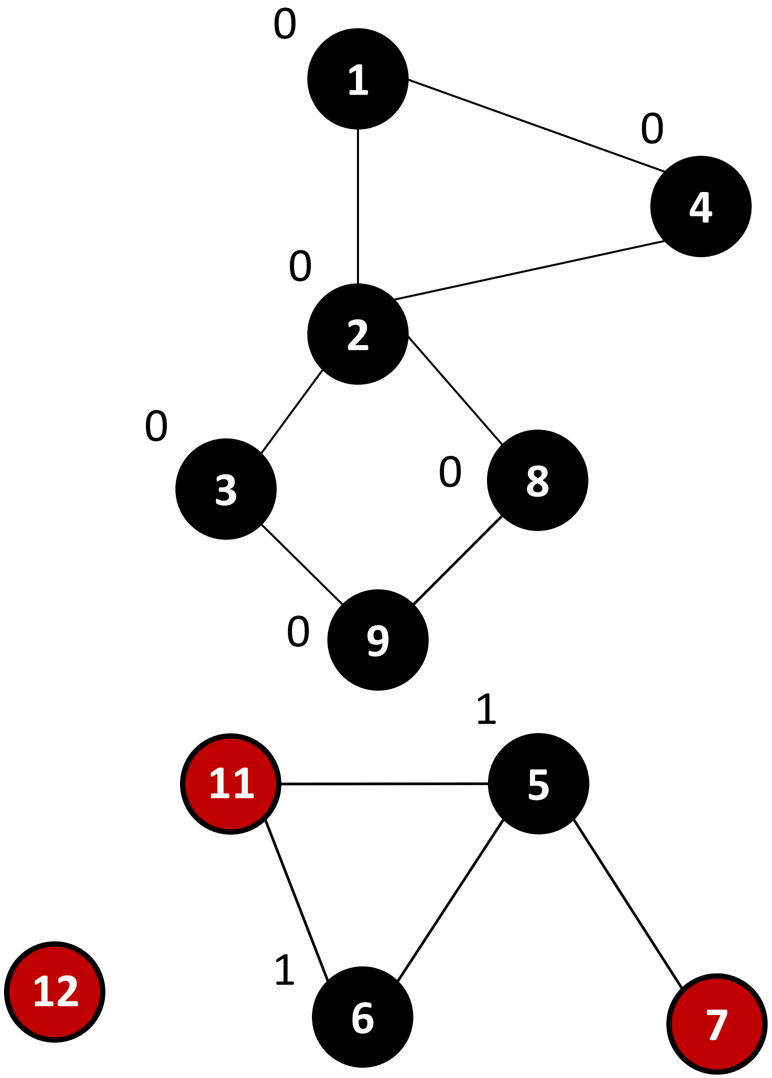
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++


LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
             neighbor, n, of node
      LCCVisit(n, c)
```



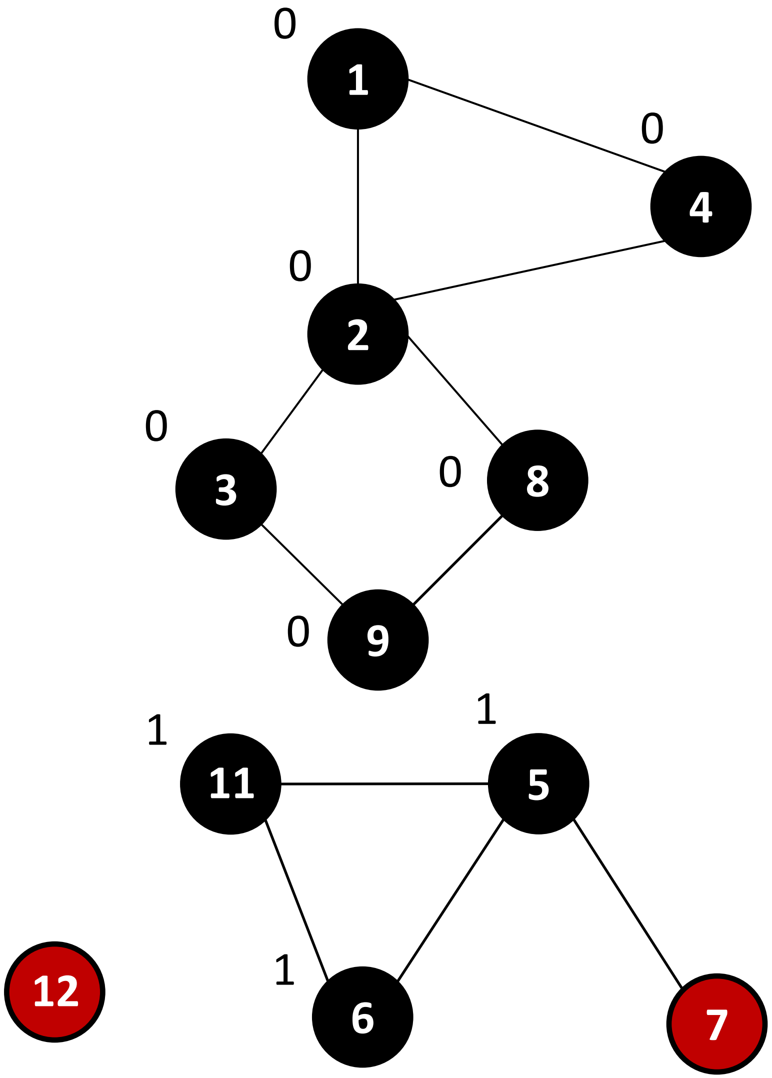
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
             neighbor, n, of node
      LCCVisit(n, c)
```



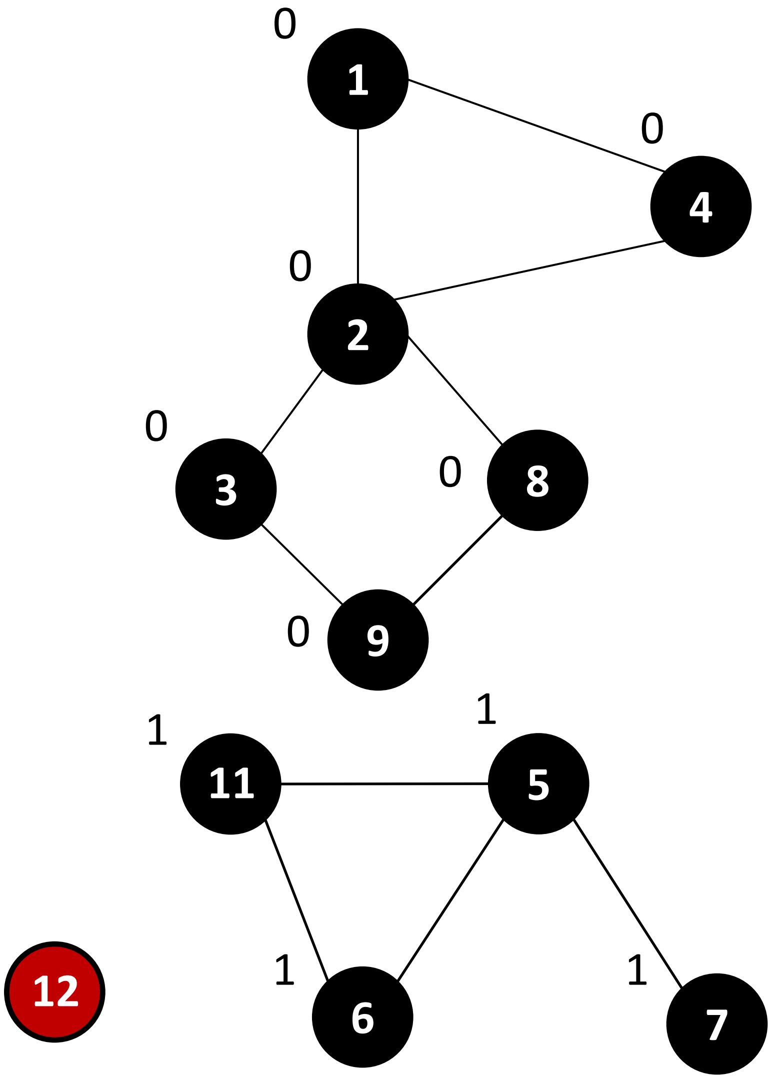
Example application: connected component analysis

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
          LCCVisit(node, component)
          component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
          neighbor, n, of node
    LCCVisit(n, c)
```



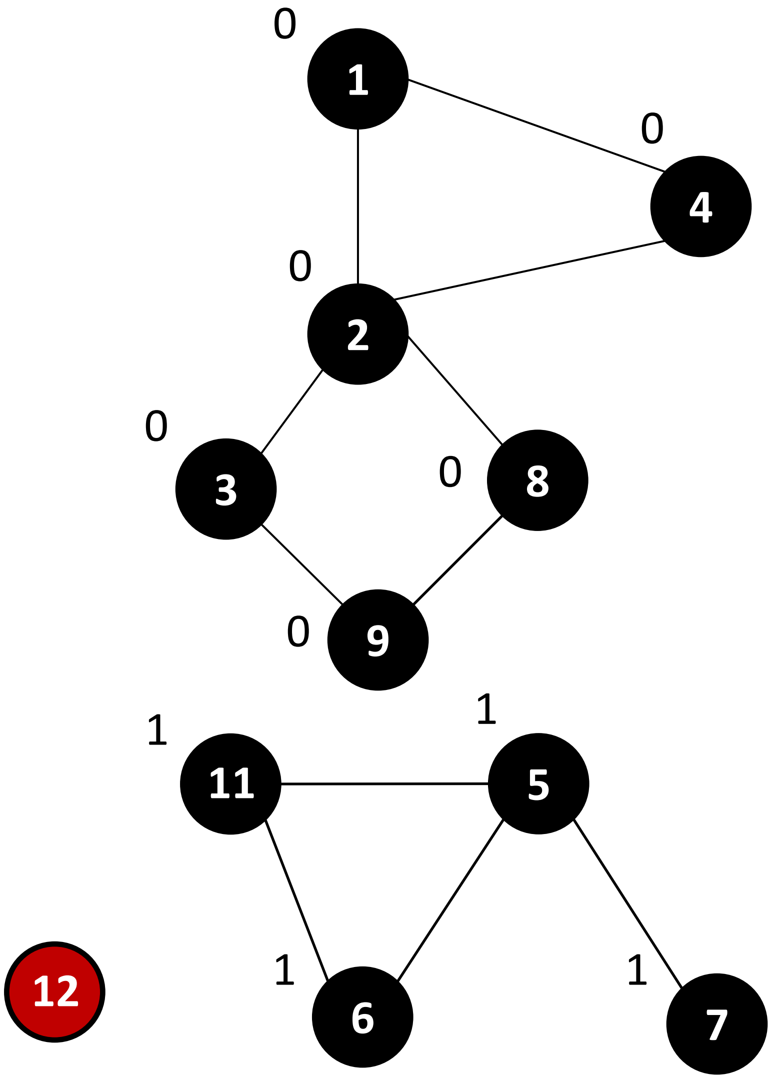
Example application: connected component analysis

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
               neighbor, n, of node
      LCCVisit(n, c)
```



Example application: connected component analysis

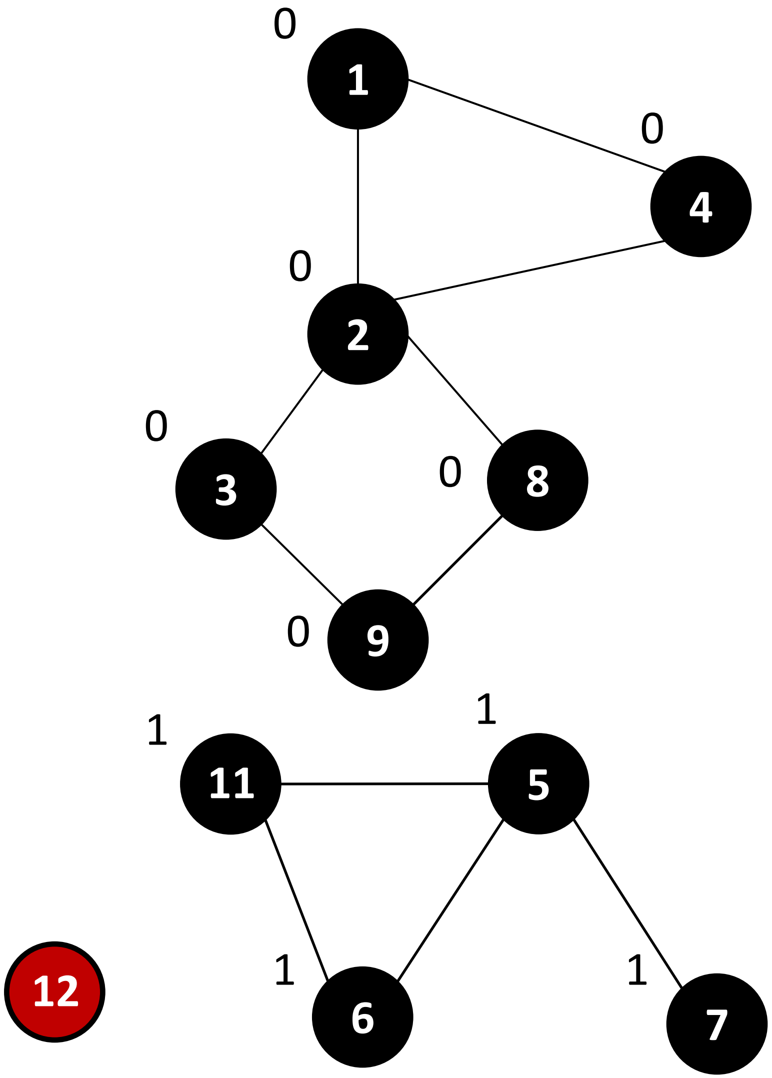
```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
            neighbor, n, of node
     LCCVisit(n, c)
```



Example application: connected component analysis

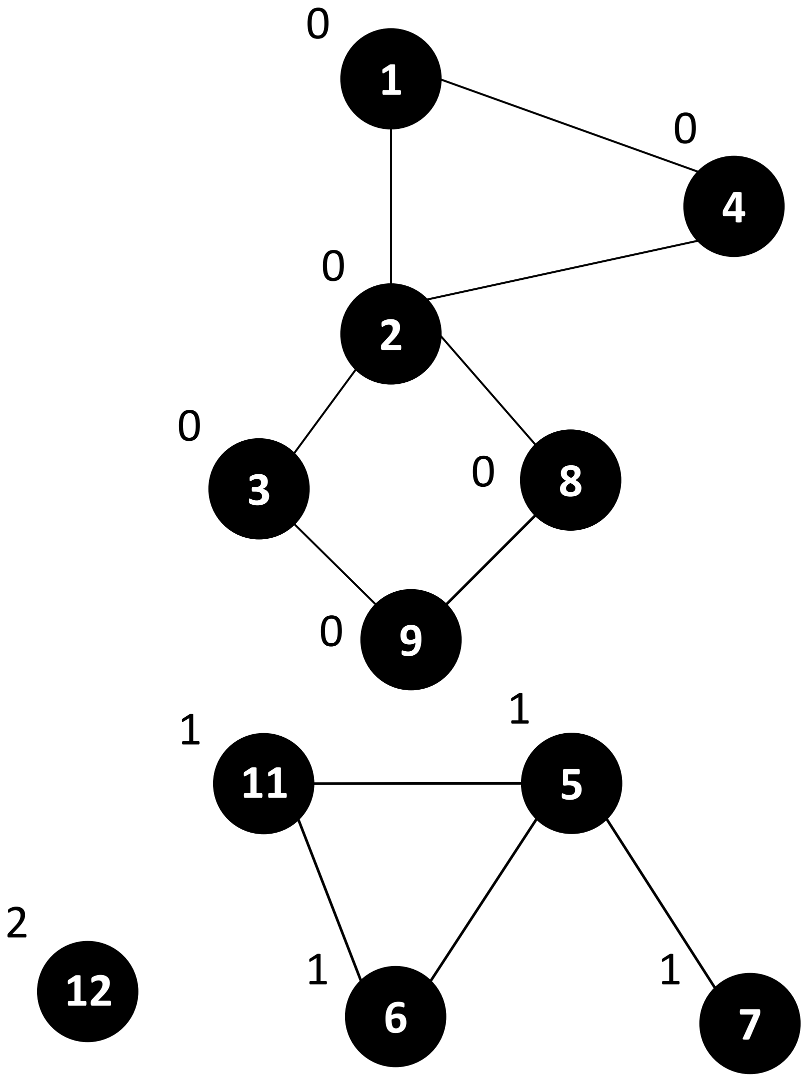
```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            **LCCVisit(node, component)**
            component++
```

```
LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
            neighbor, n, of node
      LCCVisit(n, c)
```



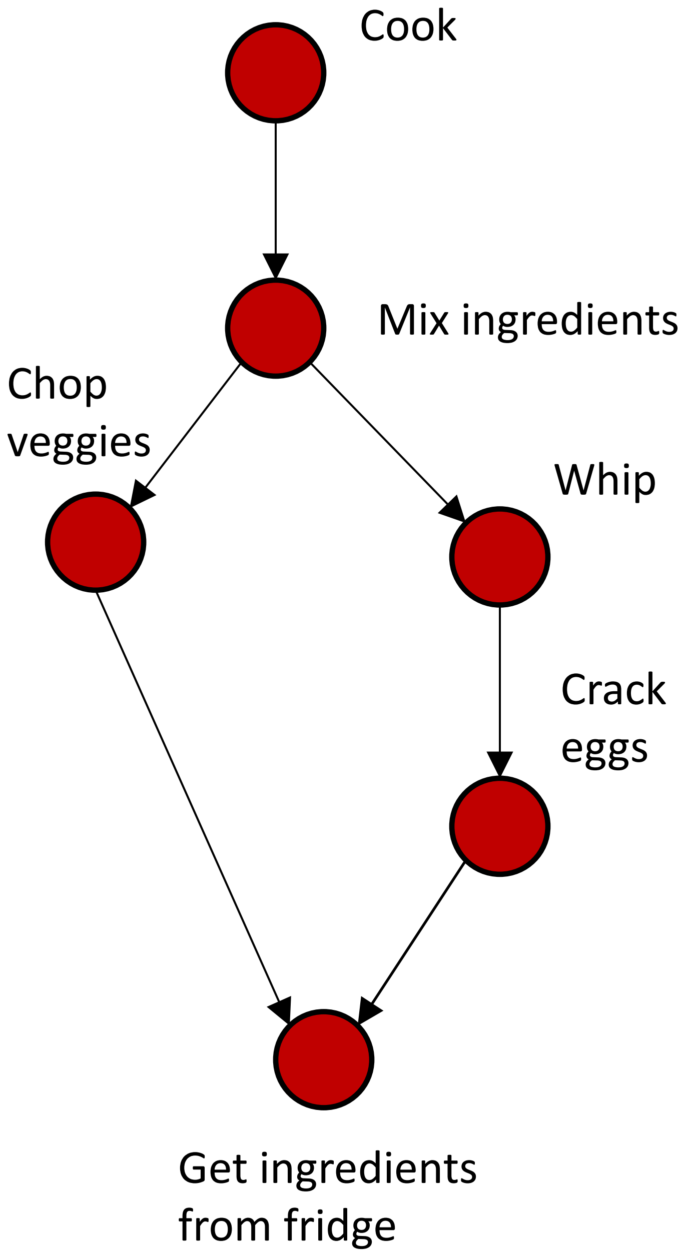
Example application: connected component analysis

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
          LCCVisit(node, component)
          component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
              neighbor, n, of node
    LCCVisit(n, c)
```



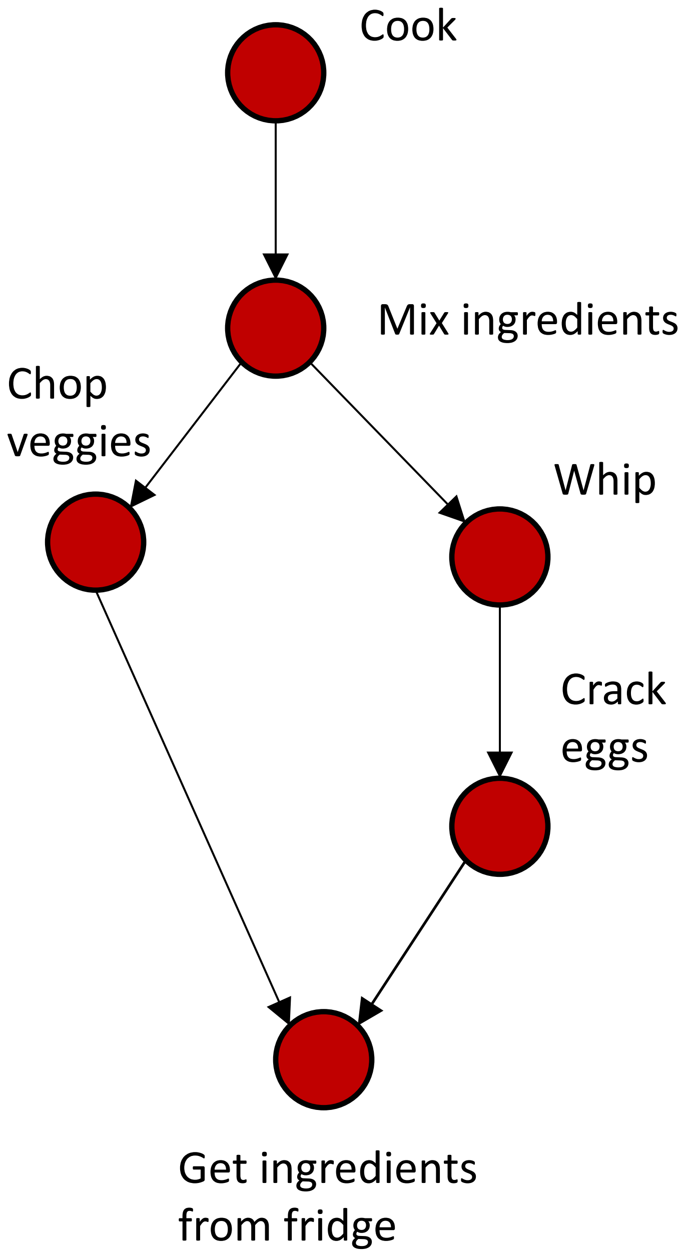
Example application: topological sort

- Given a **connected, directed acyclic graph (DAG)**
- Find an **ordering** of the nodes such that for any nodes a, b
  - If there is an **edge** from a to b
  - b **comes after** a in the ordering
  - Alternate version: b comes before a in the order
- Useful for problems such as **scheduling** tasks
  - Make a graph of the **dependencies** between **subtasks**
  - Topologically sort it



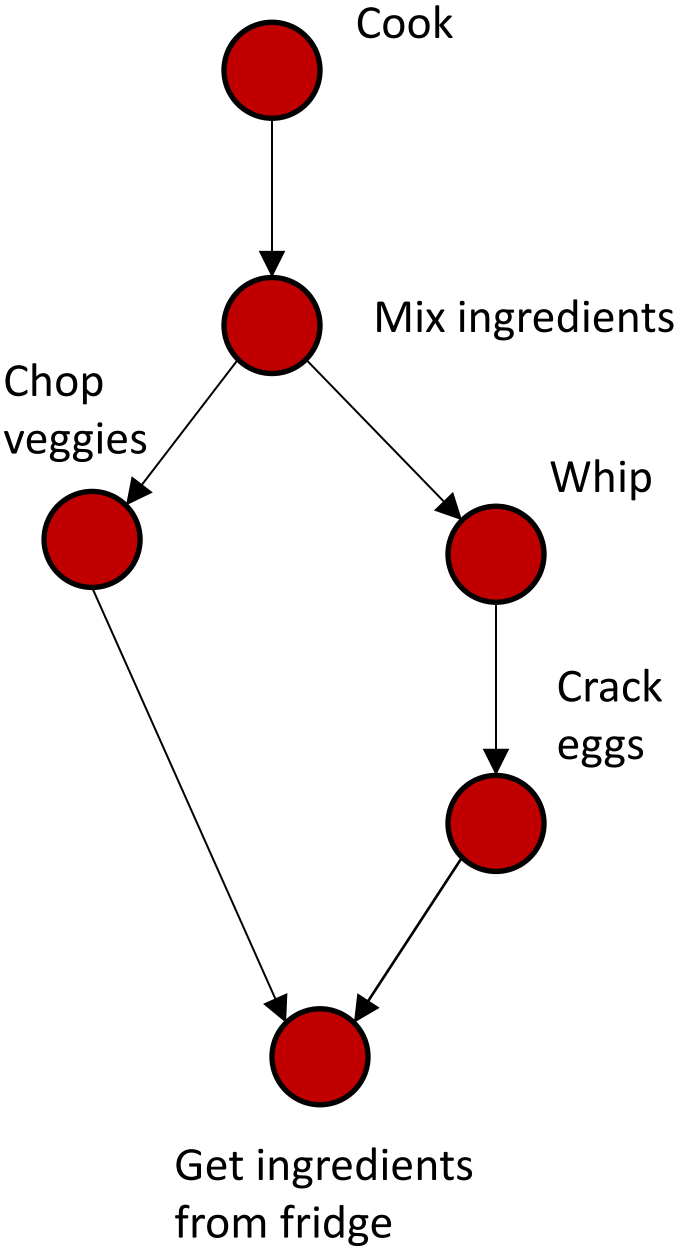
Example application: topological sort

- Essentially a fancy **post-order** traversal
- Used to find **total orderings** from **partial orderings**
- If you've used the **make** program on linux, it's essentially doing a topological sort to decide what order to compile things in



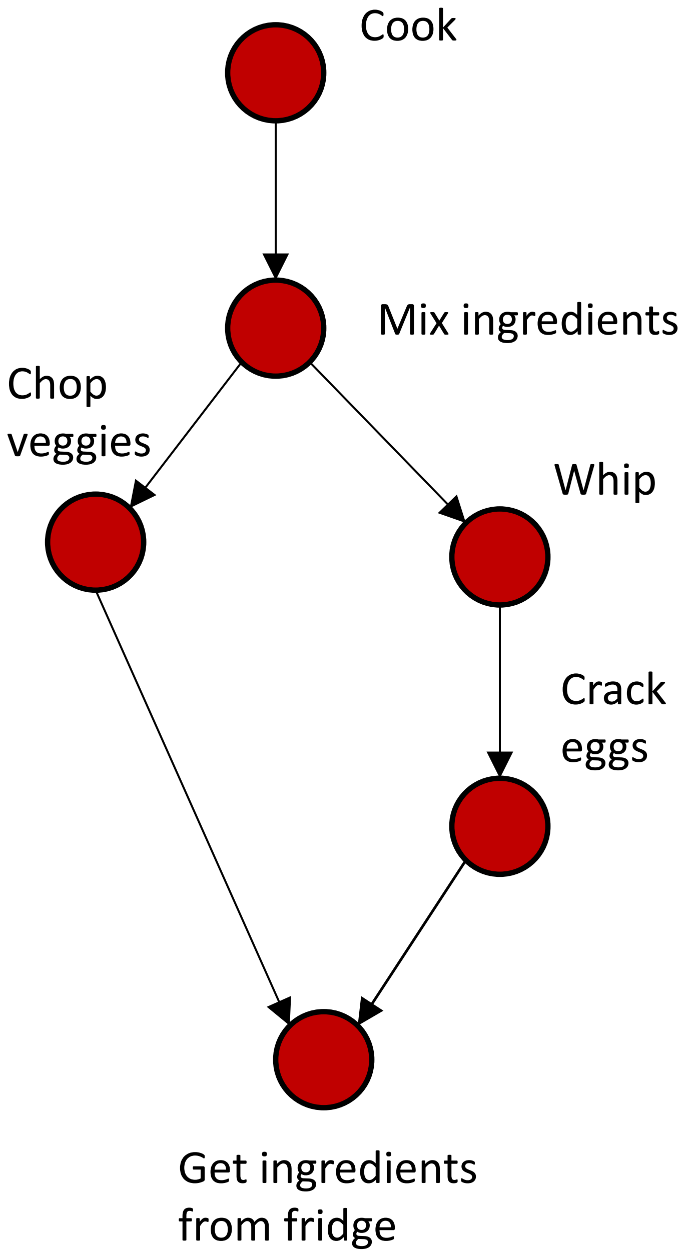
Example application: topological sort

```
TopologicalSort(start)
    list = new empty list
    TopologicalVisit(start)

TopologicalVisit(node)
    for each node n with an
            edge from node to n
            if n not visited
                TopologicalVisit(n)
    mark node visited
    add node to list
```



Example application: topological sort

TopologicalVist(cook)

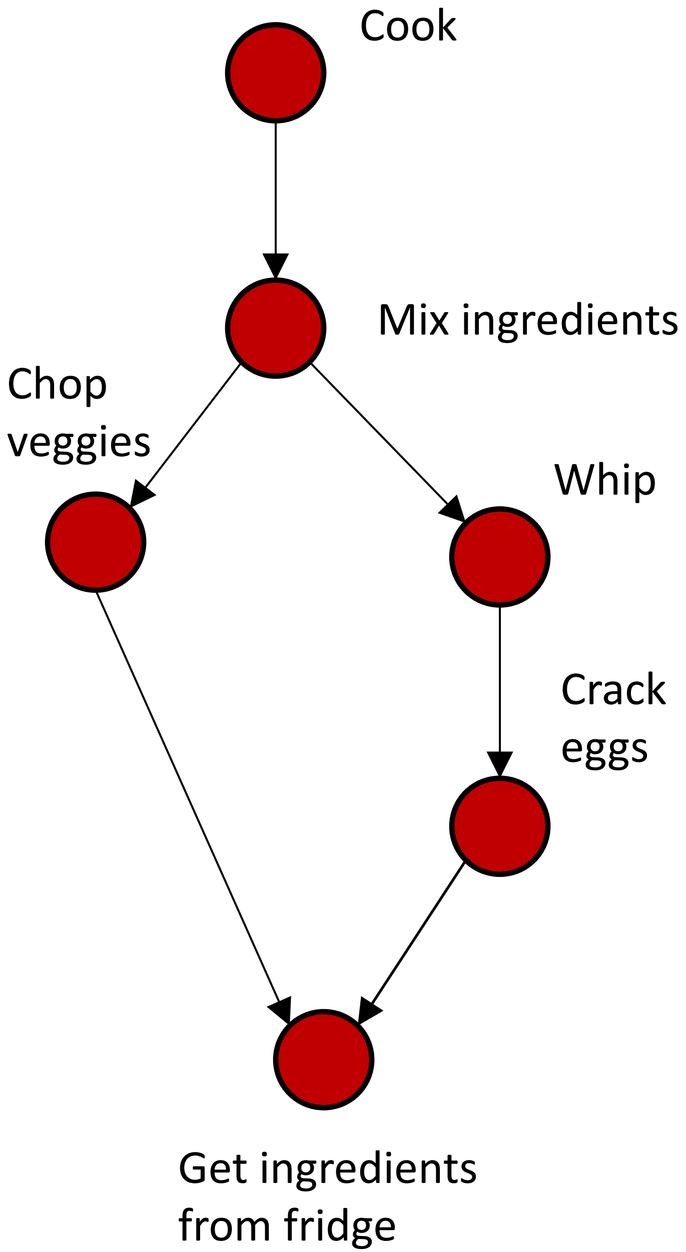


Output list: <empty>

Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)



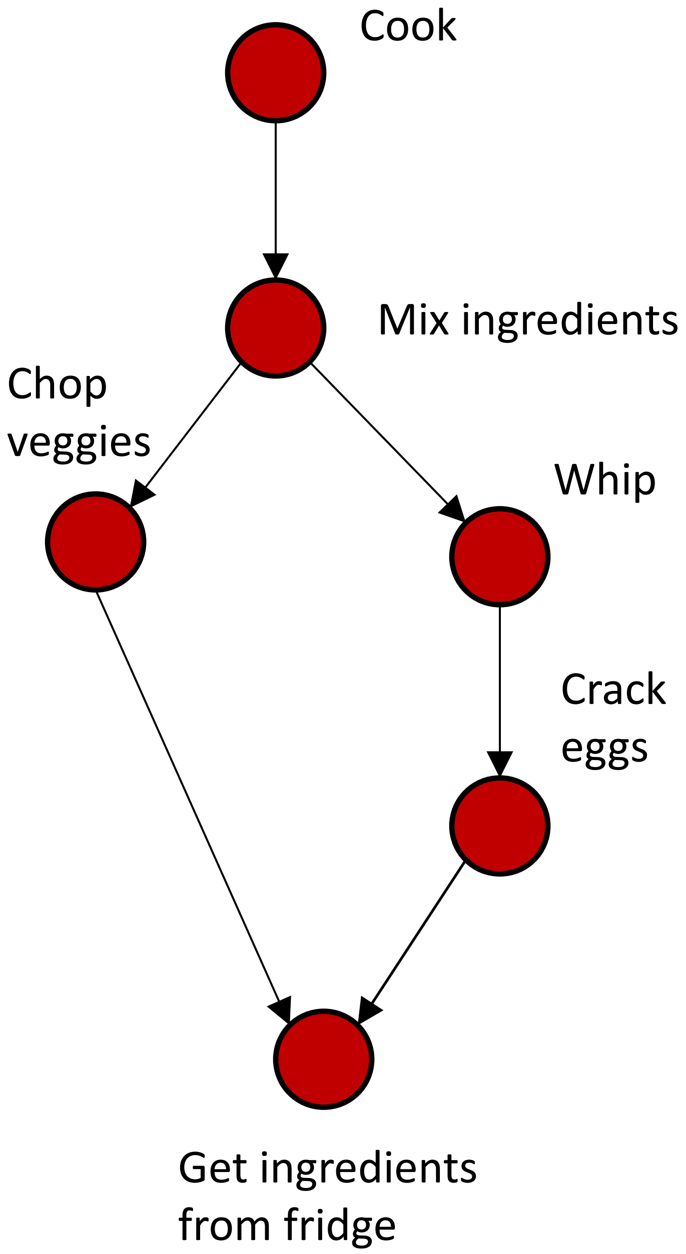
Output list: <empty>

Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)

TopologicalVist(chop)



Output list: <empty>

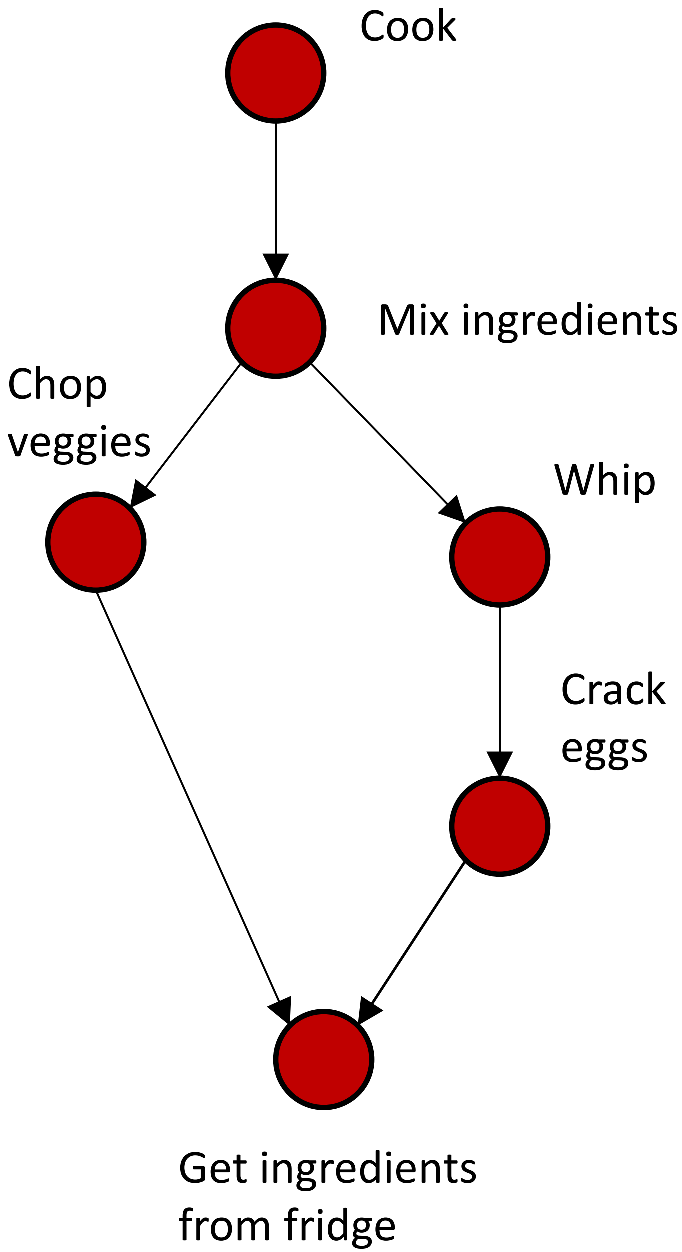
Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)

TopologicalVist(chop)

TopologicalVist(get)



Output list: <empty>

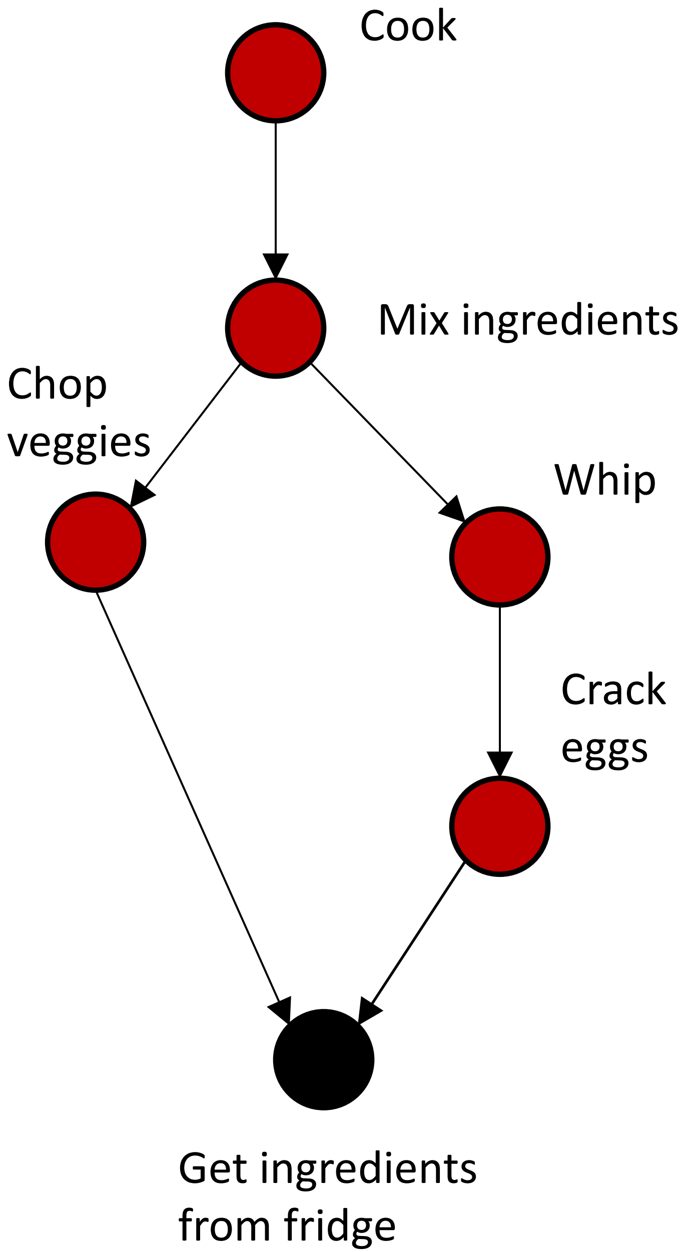
Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)

TopologicalVist(chop)

TopologicalVist(get)



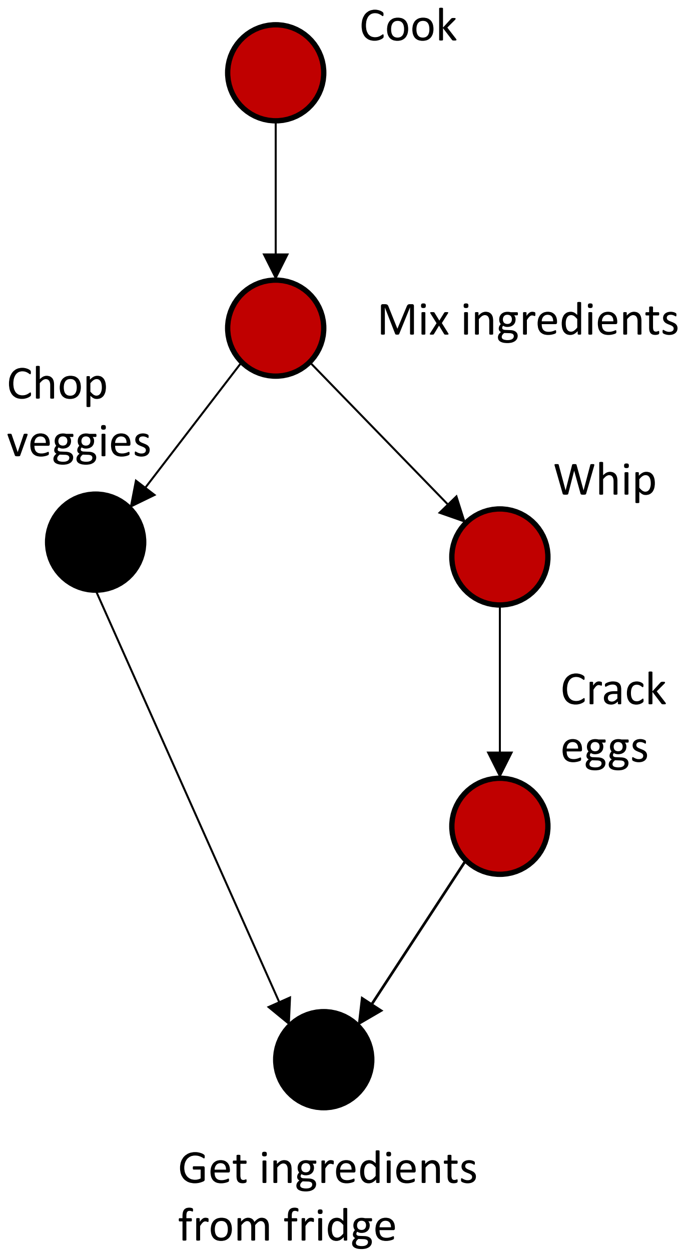
Output list: get

Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)

TopologicalVist(chop)

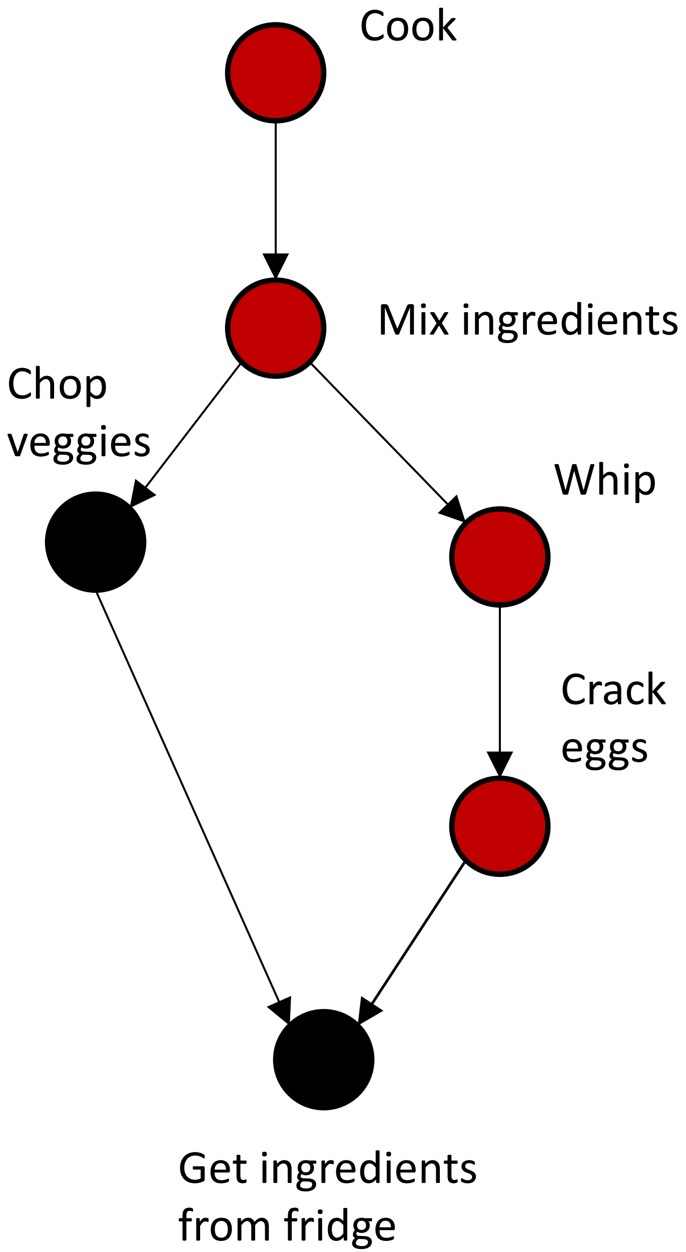


Output list: get, chop

Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)



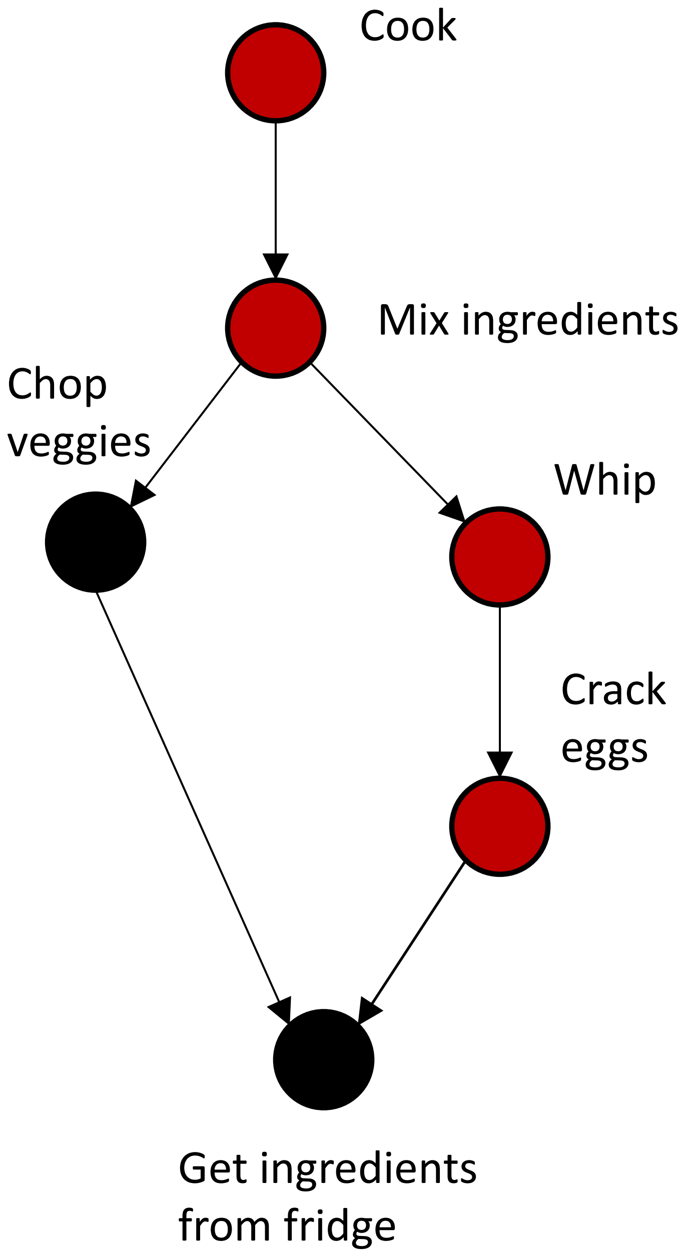
Output list: get, chop

Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)

TopologicalVist(whip)



Output list: get, chop

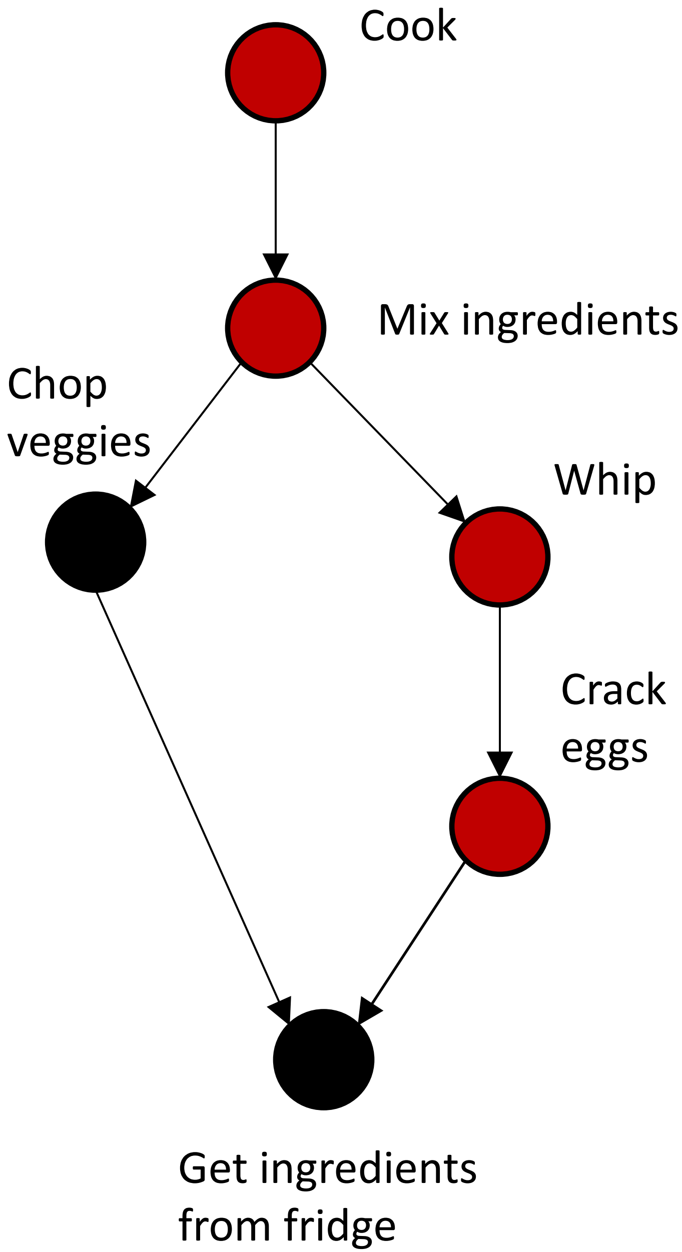
Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)

TopologicalVist(whip)

TopologicalVist(crack)



Output list: get, chop

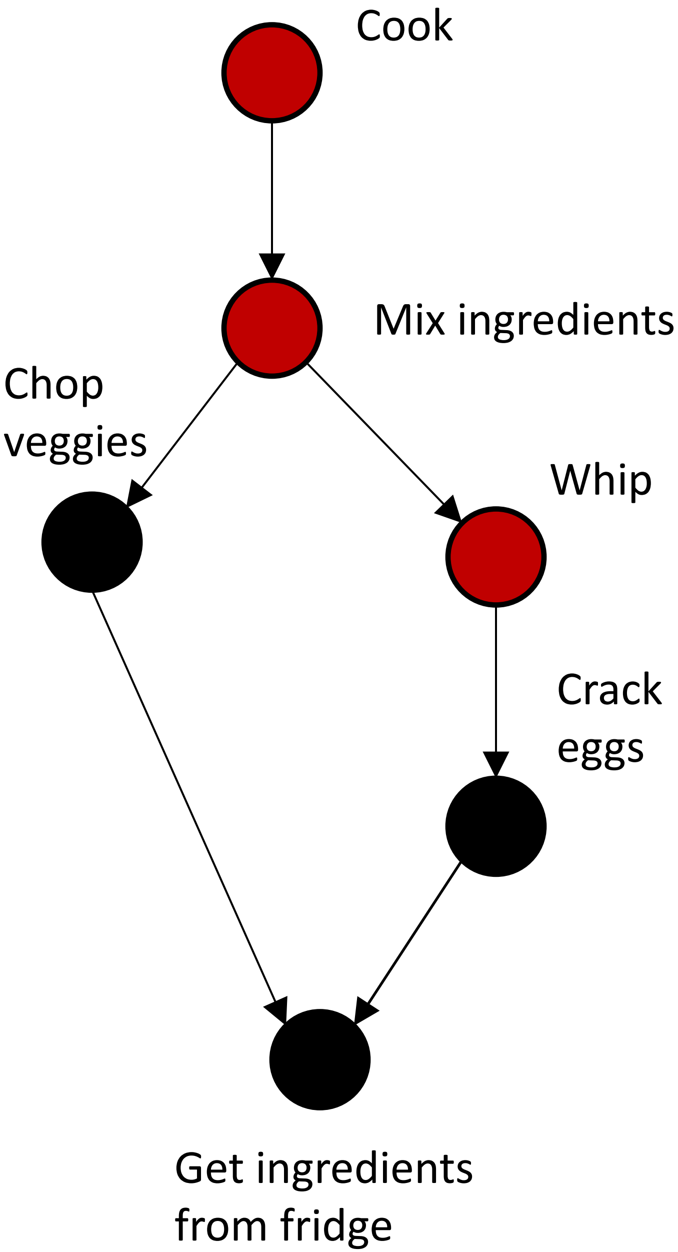
Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)

TopologicalVist(whip)

TopologicalVist(crack)



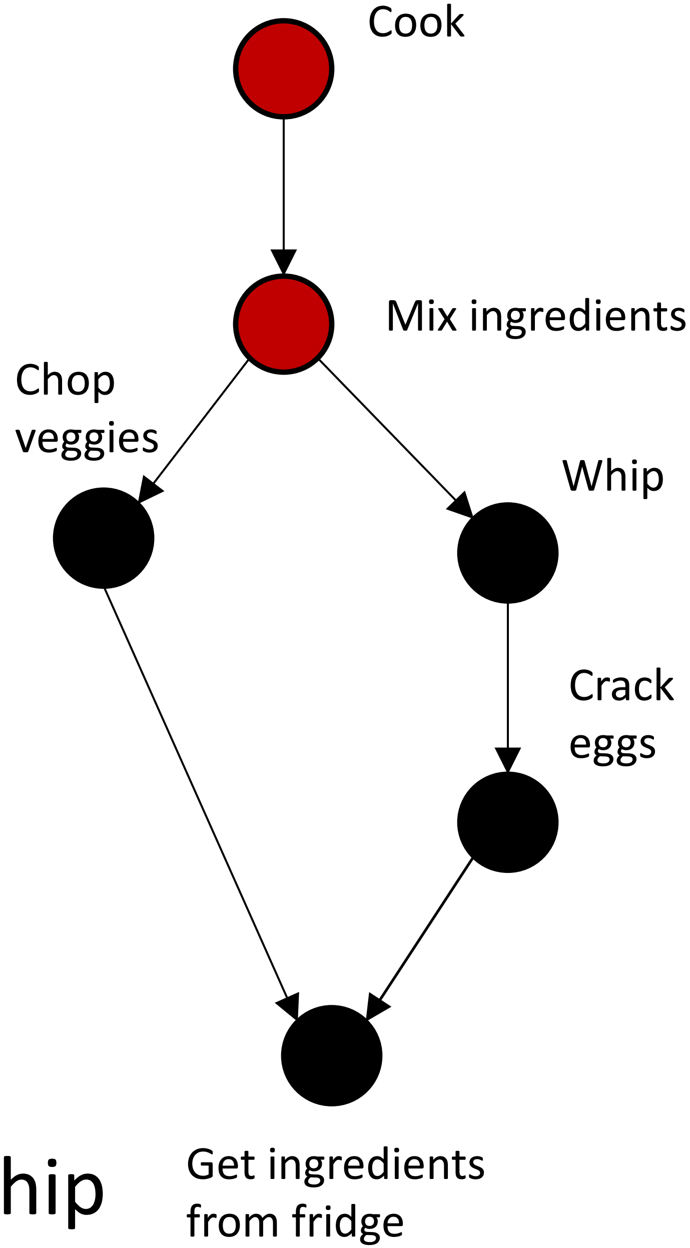
Output list: get, chop, crack

Example application: topological sort

TopologicalVist(cook)

TopologicalVist(mix)

TopologicalVist(whip)

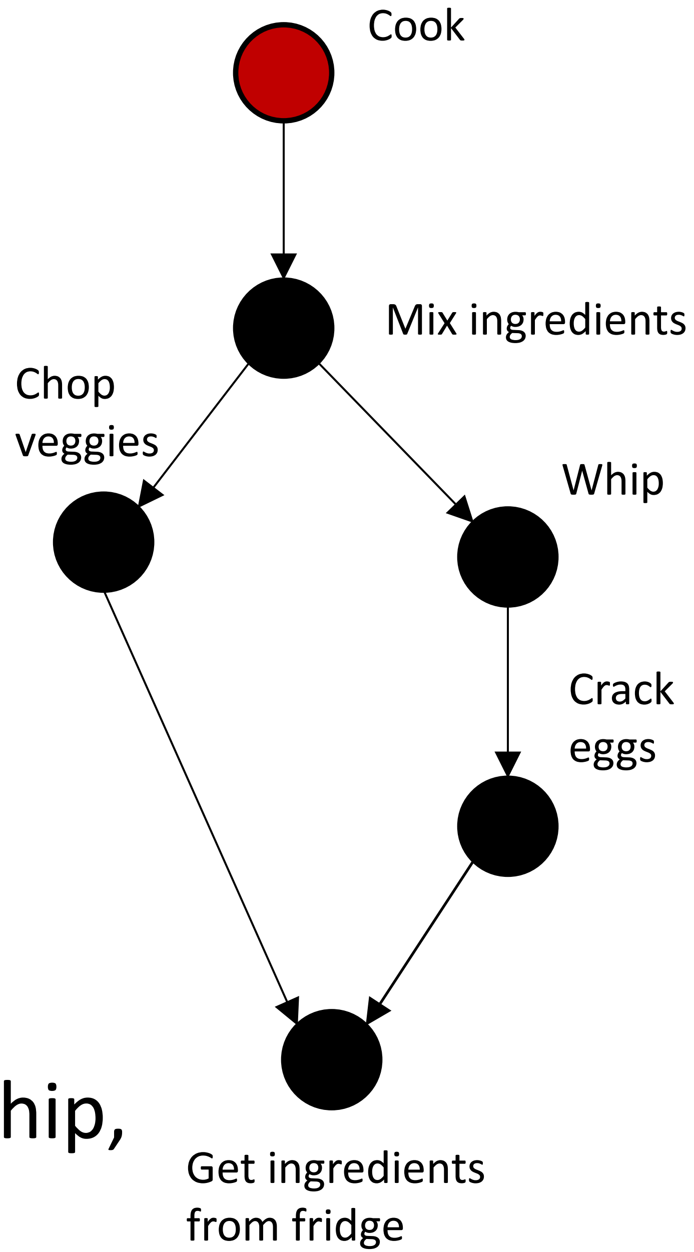


Output list: get, chop, crack, whip

Example application: topological sort

TopologicalVist(cook)

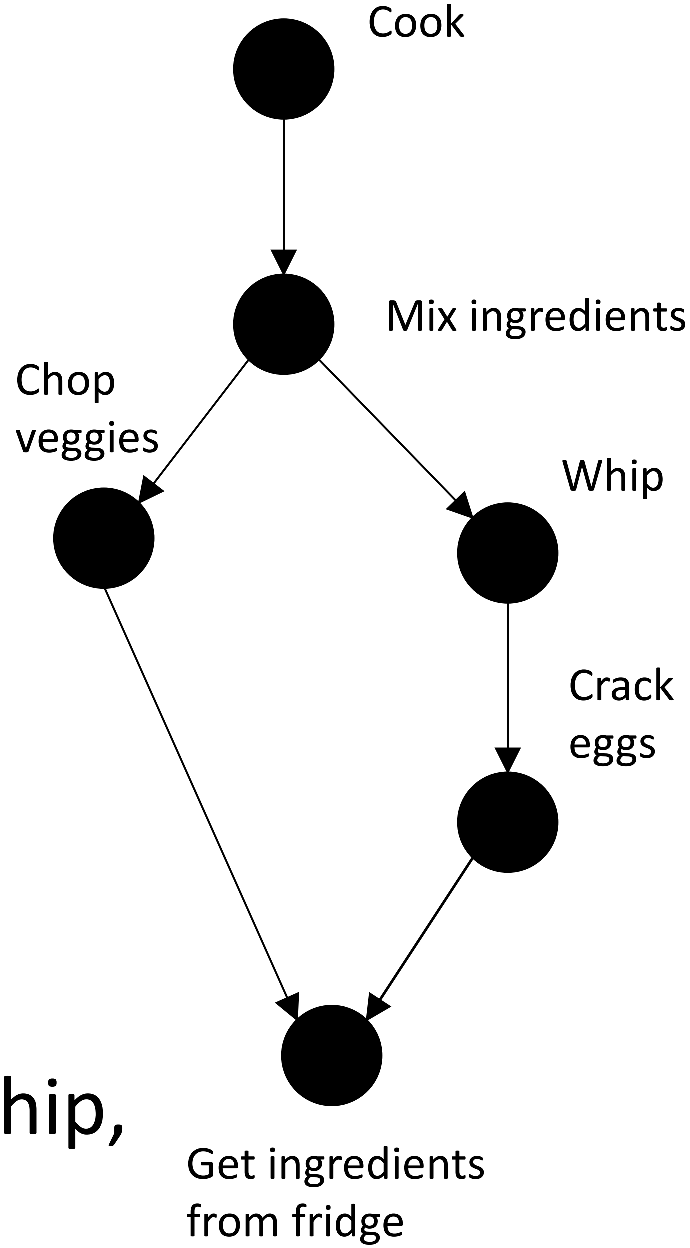
TopologicalVist(mix)



Output list: get, chop, crack, whip, mix

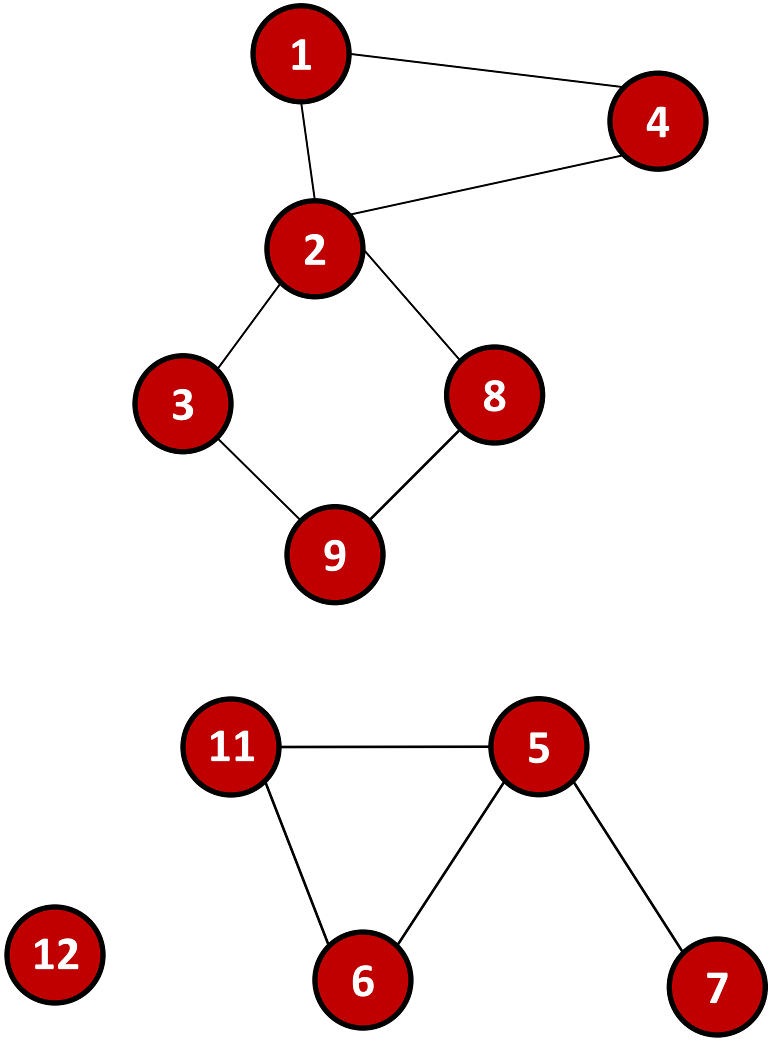
Example application: topological sort

TopologicalVist(cook)



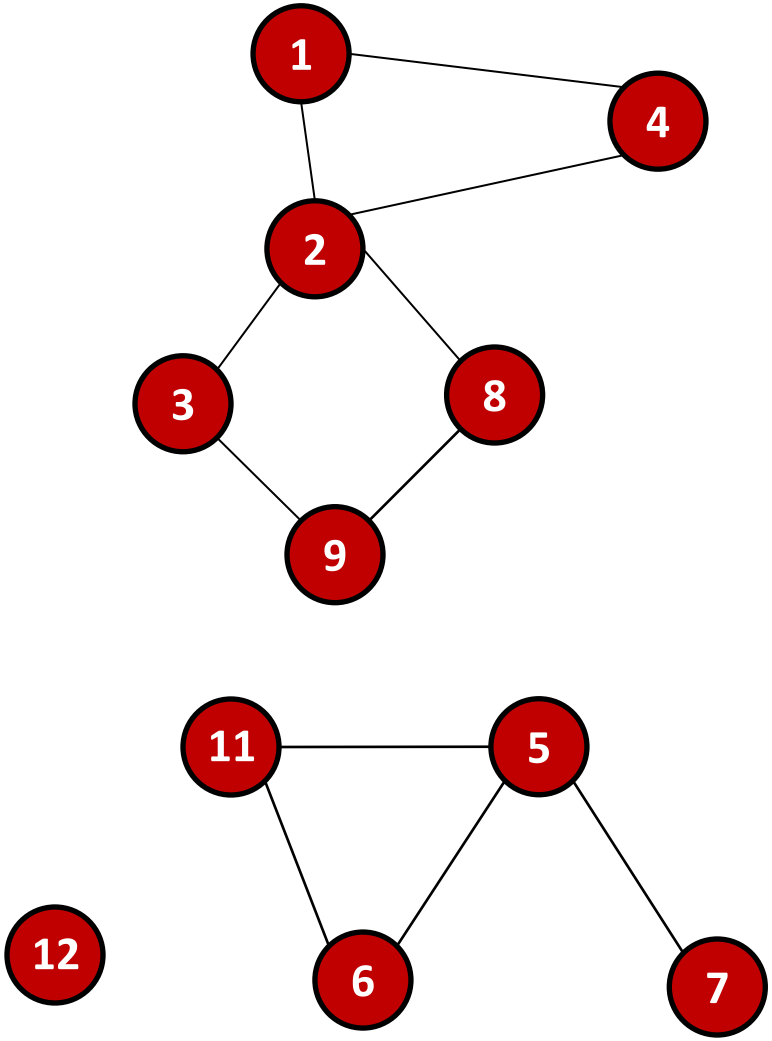
Output list: get, chop, crack, whip, mix, cook

Timestamping nodes



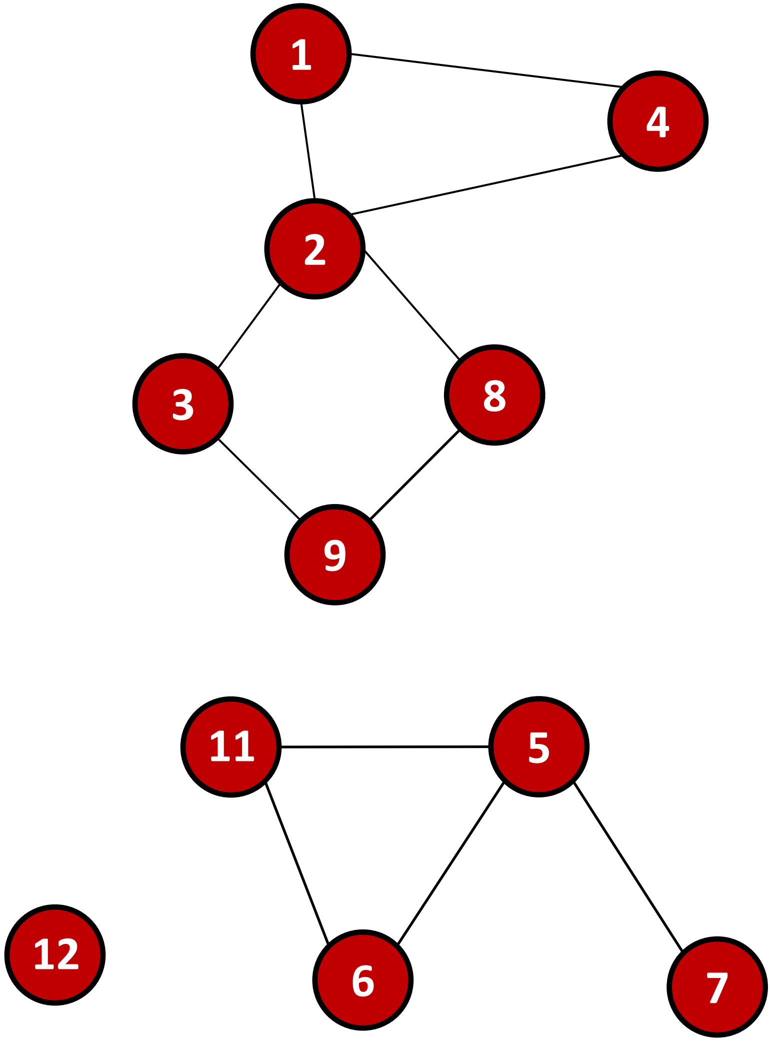
- It turns out to be useful to keep track of **when** nodes are accessed
- We keep a **counter** variable that acts as a kind of **clock**
  - Increment it every time we call DFSVisit
- **Record** the value of the counter every time we
  - Start a call to DFSVisit
  - End a call to DFSVisit

Timestamping nodes



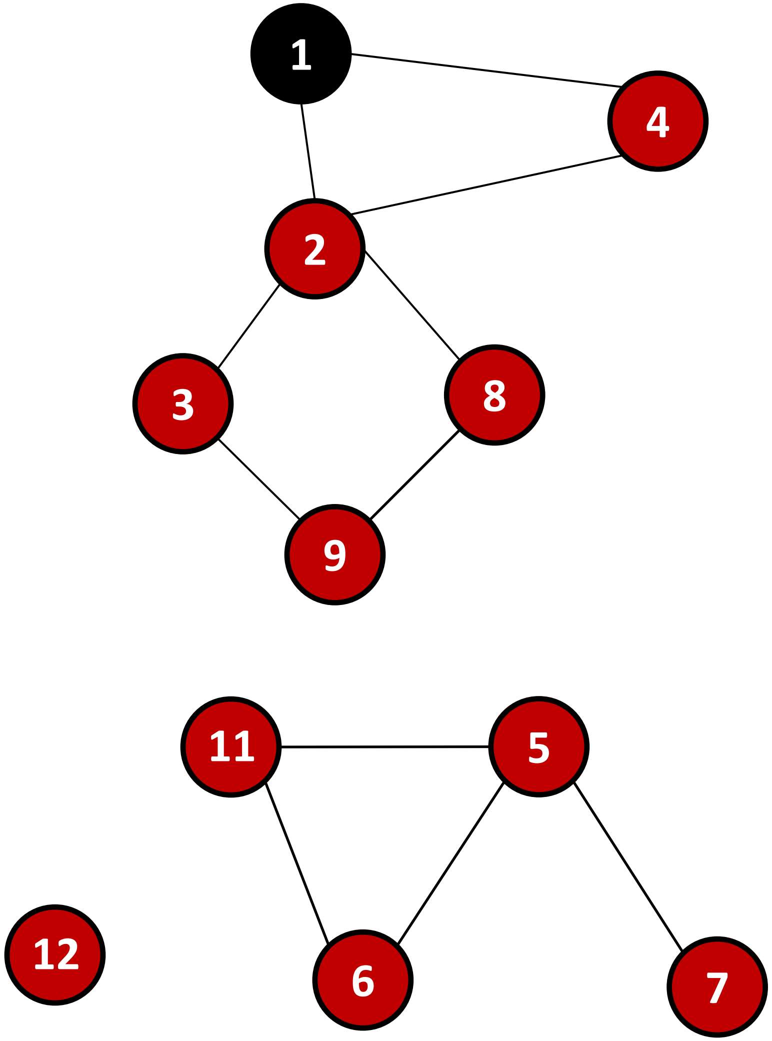
- And as long as we’re at it, we’ll also keep track of **predecessor** nodes
  - As with the fancy version of breadth-first search

Timestamping nodes



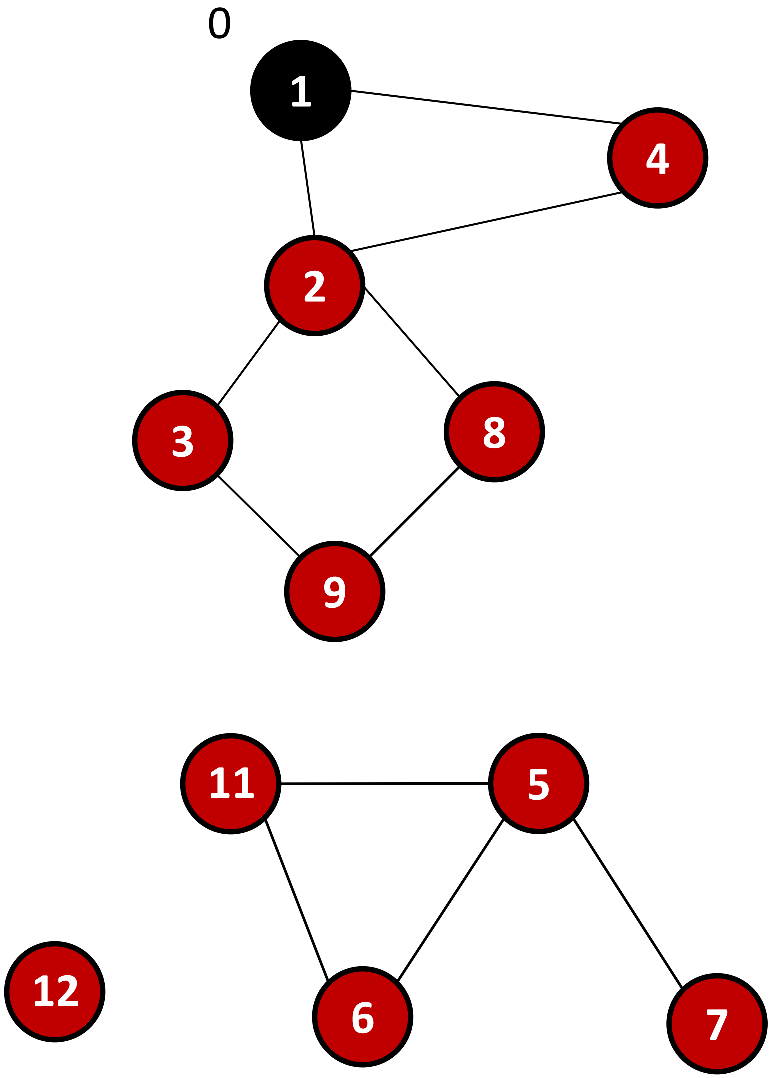
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



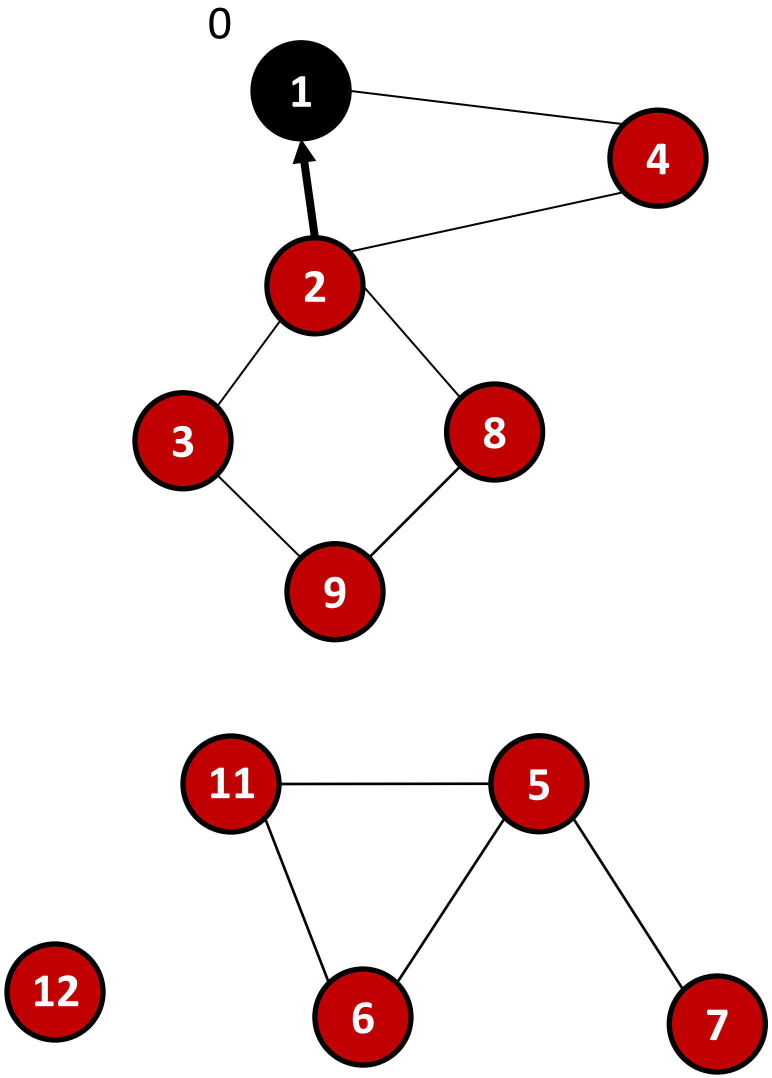
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



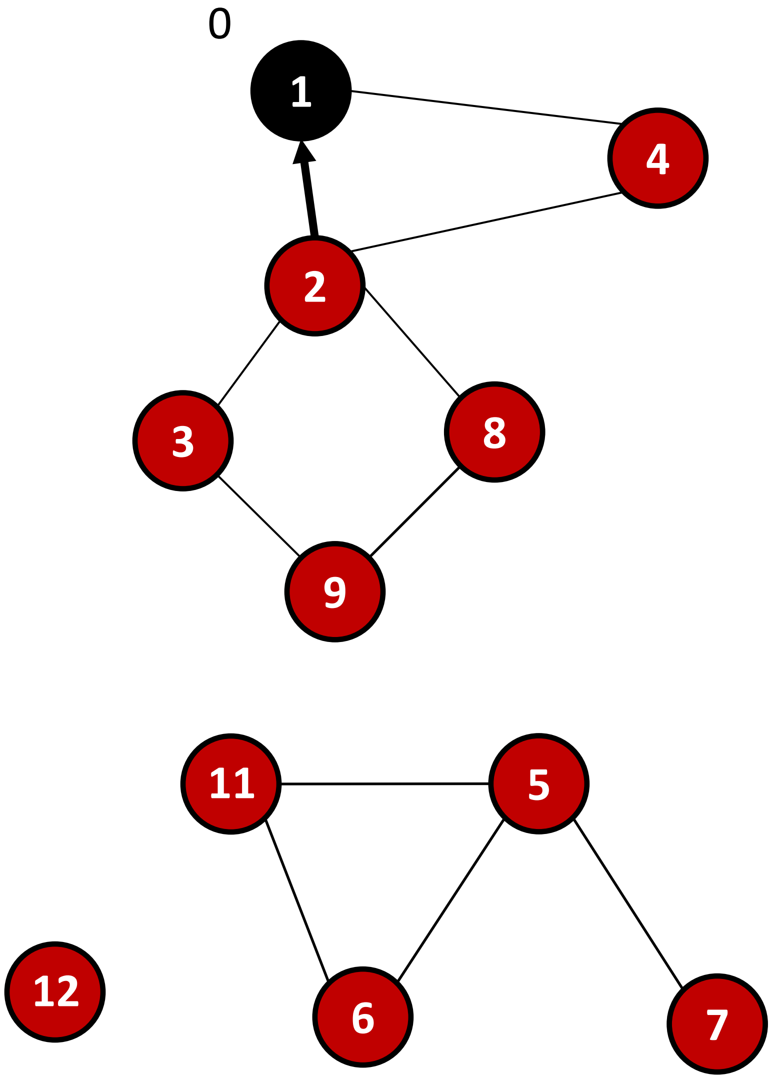
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



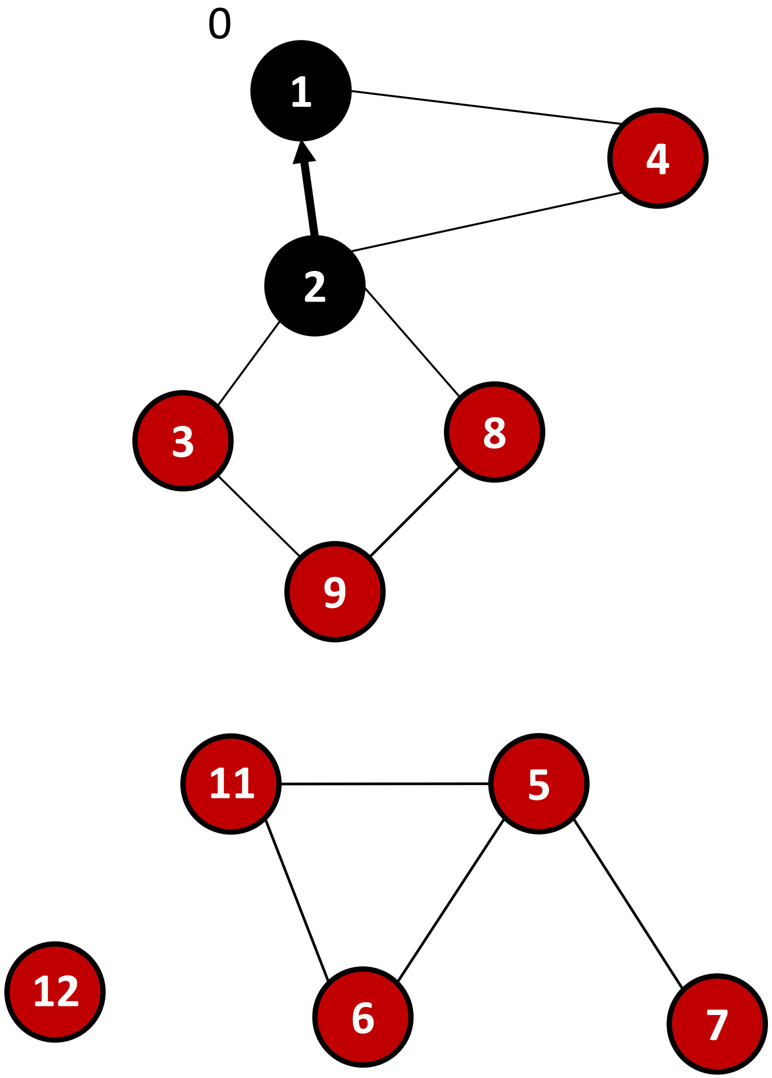
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



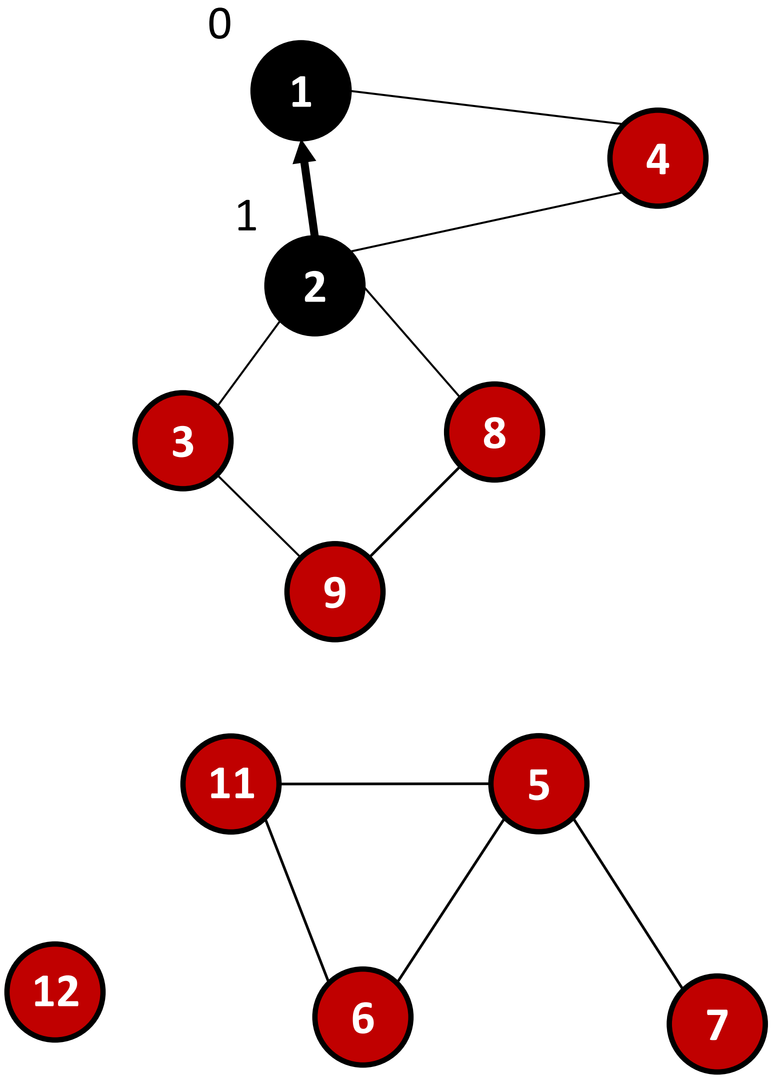
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



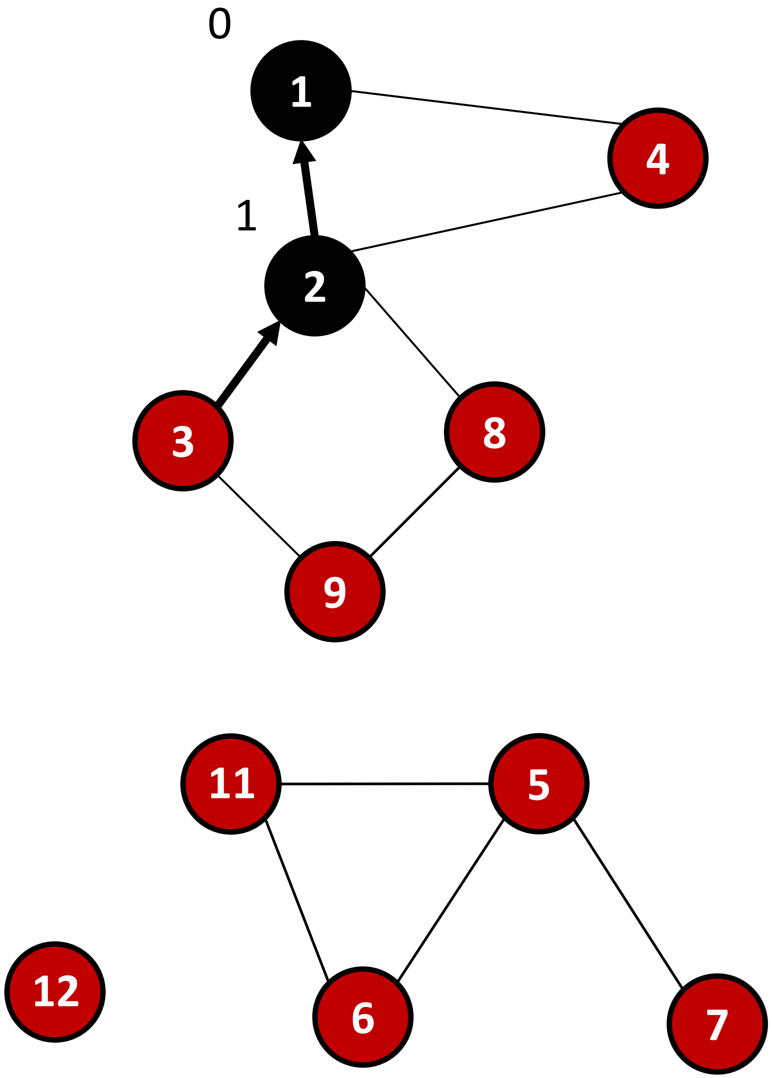
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



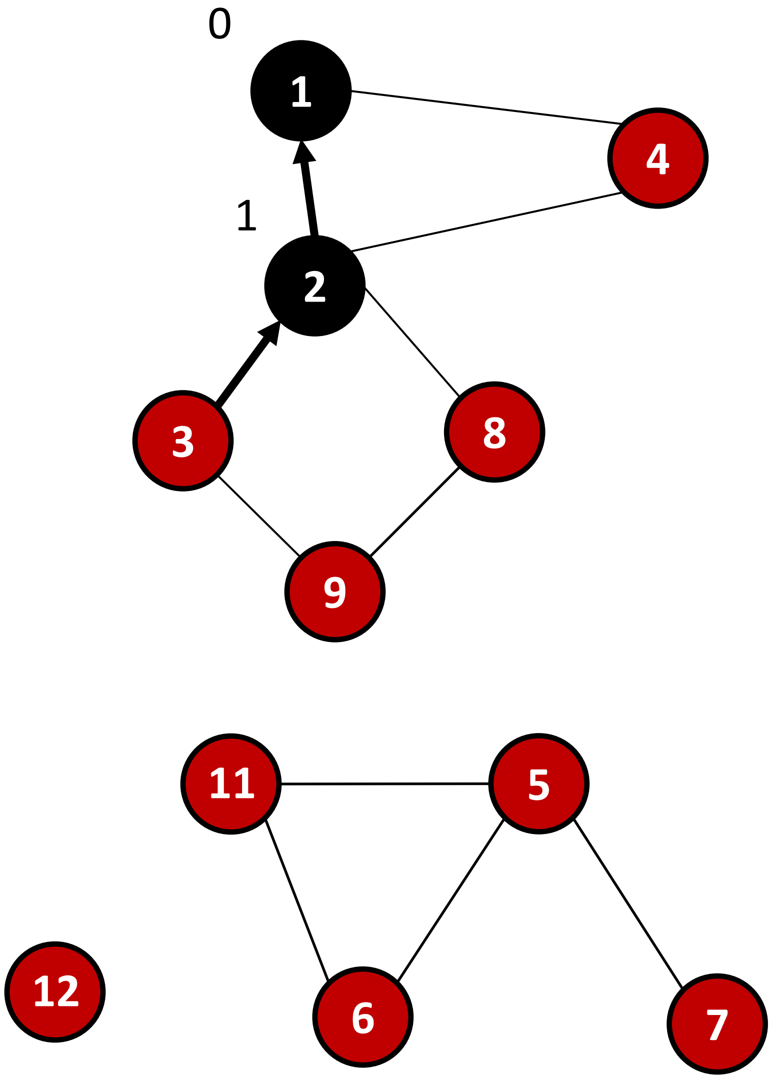
```
DepthFirst()
   time = 0
   for each node in graph
        if node not visited
            DFSVisit(node)

DFSVisit(node)
   mark node visited
   node.discovered = time++
   for each unvisited neighbor,
              c, of node
      c.predecessor = node
      DFSVisit(c)
   node.finished = time++
```

Timestamping nodes



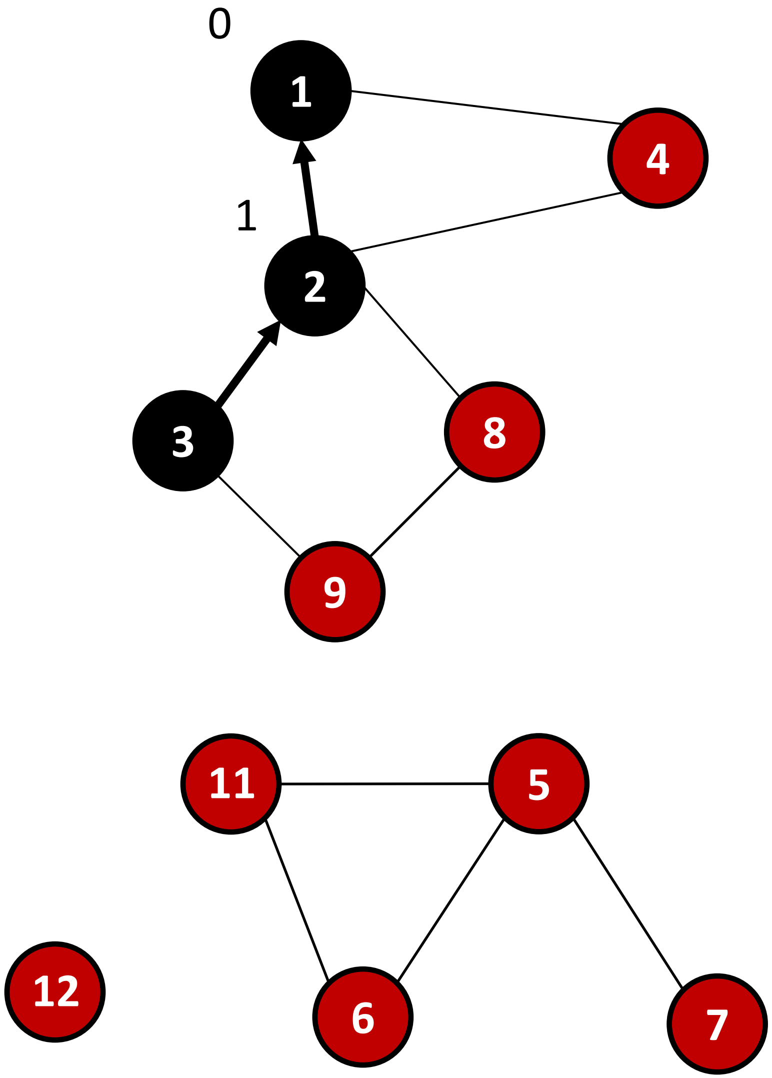
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



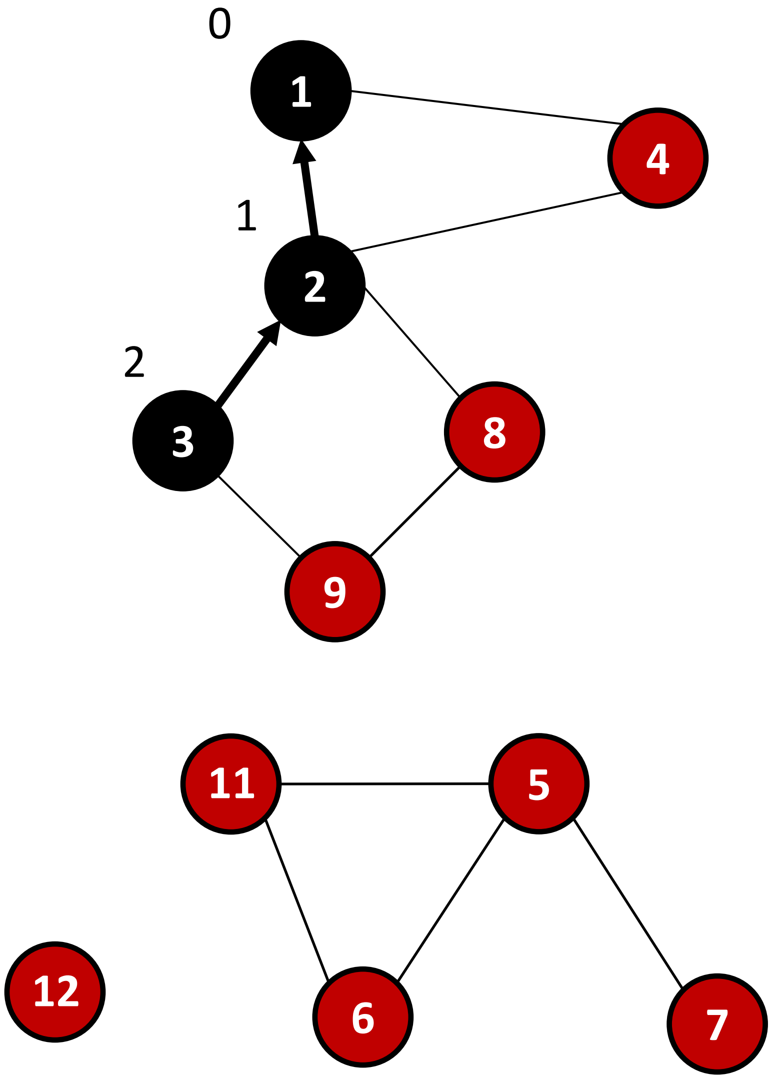
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



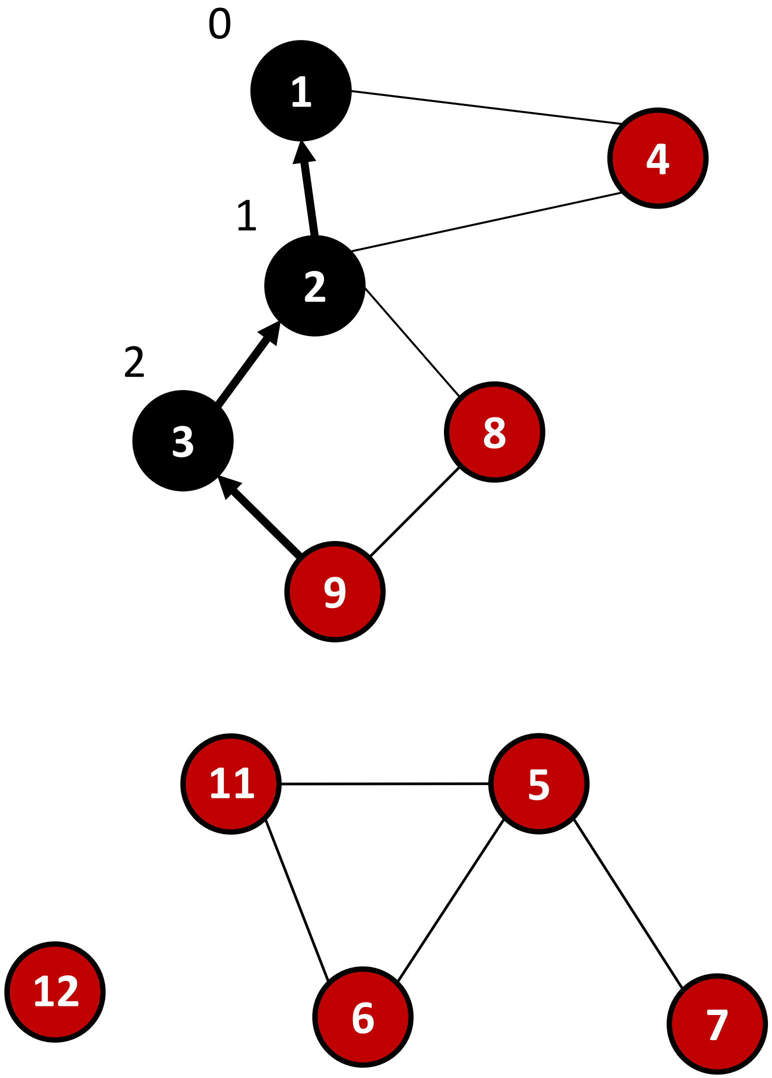
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



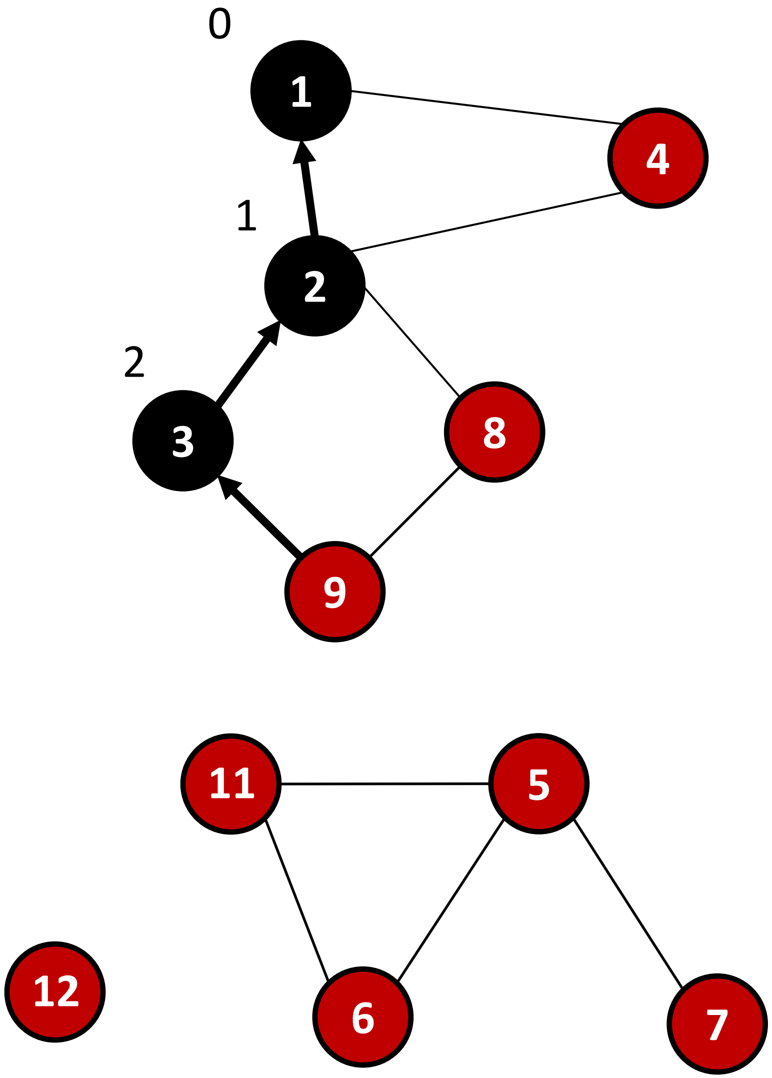
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



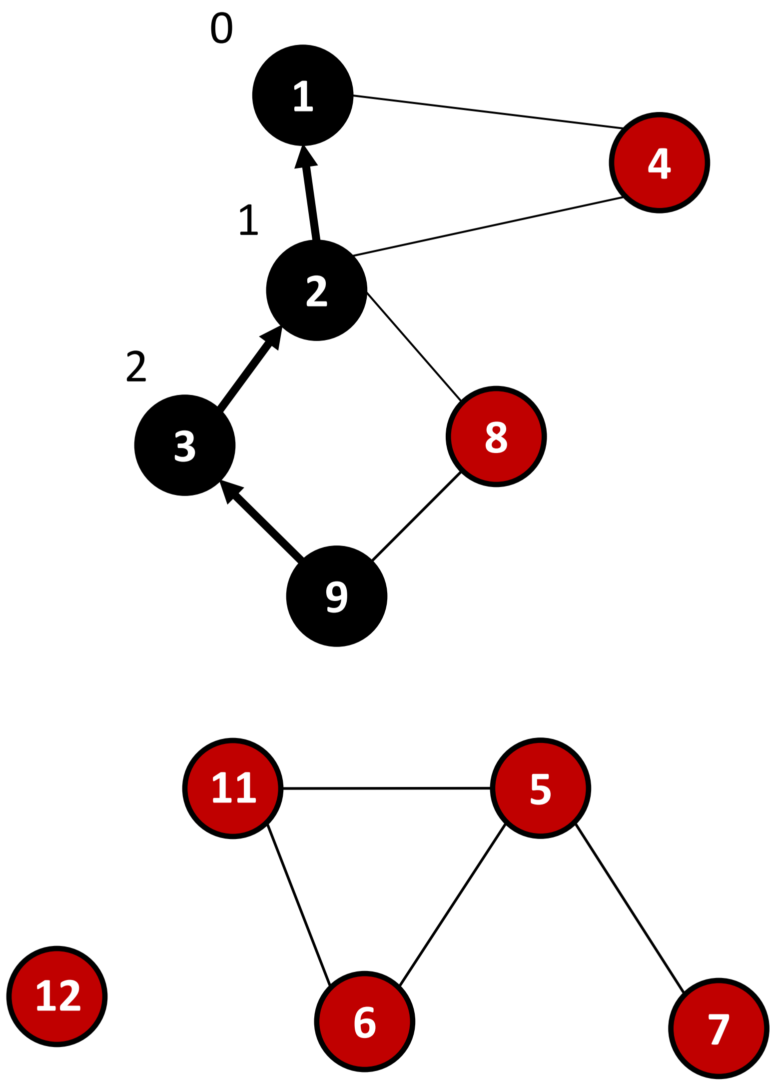
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



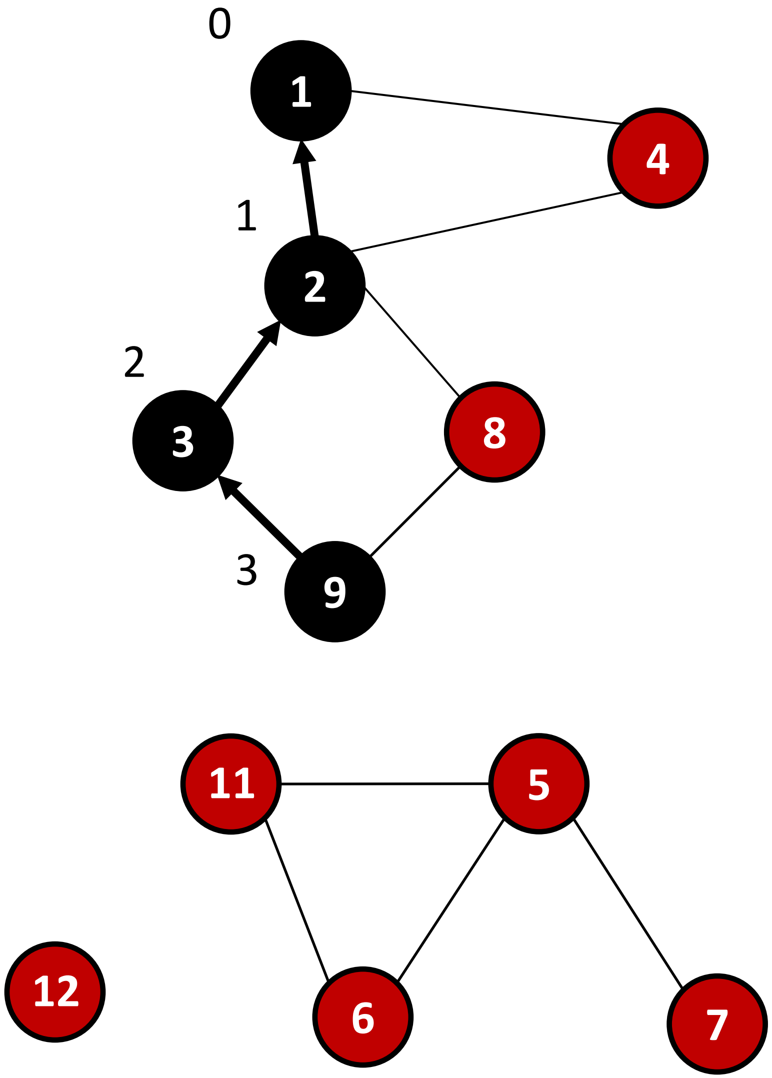
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
            c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



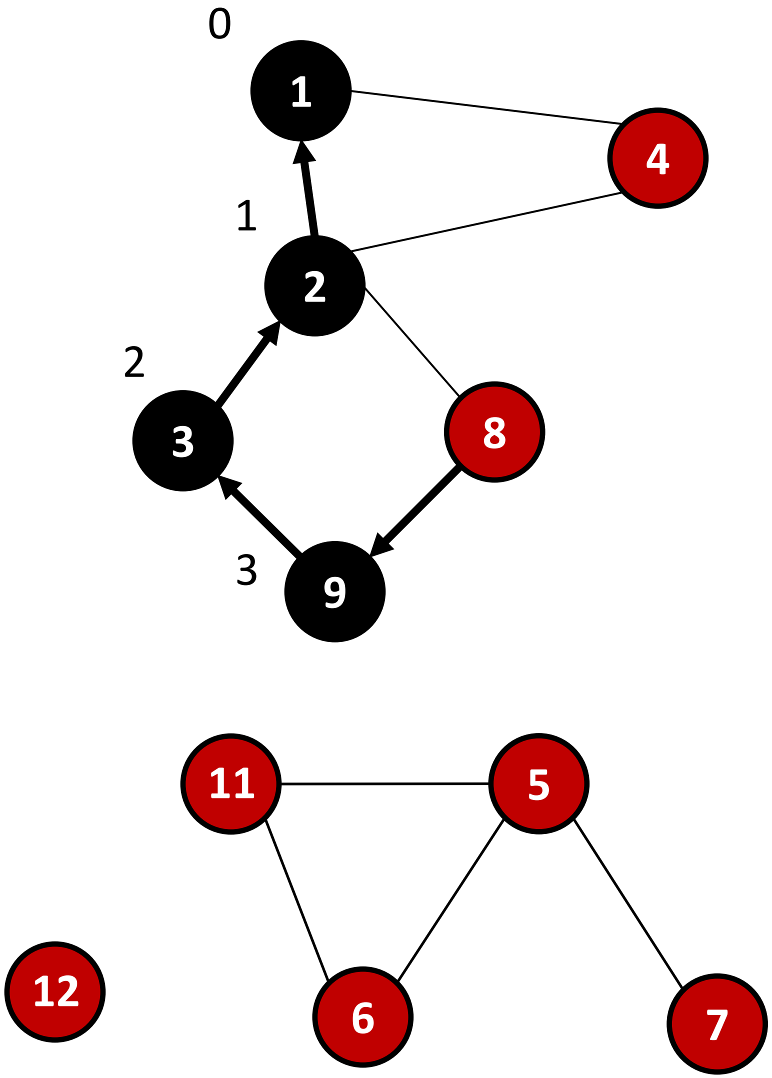
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
              c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



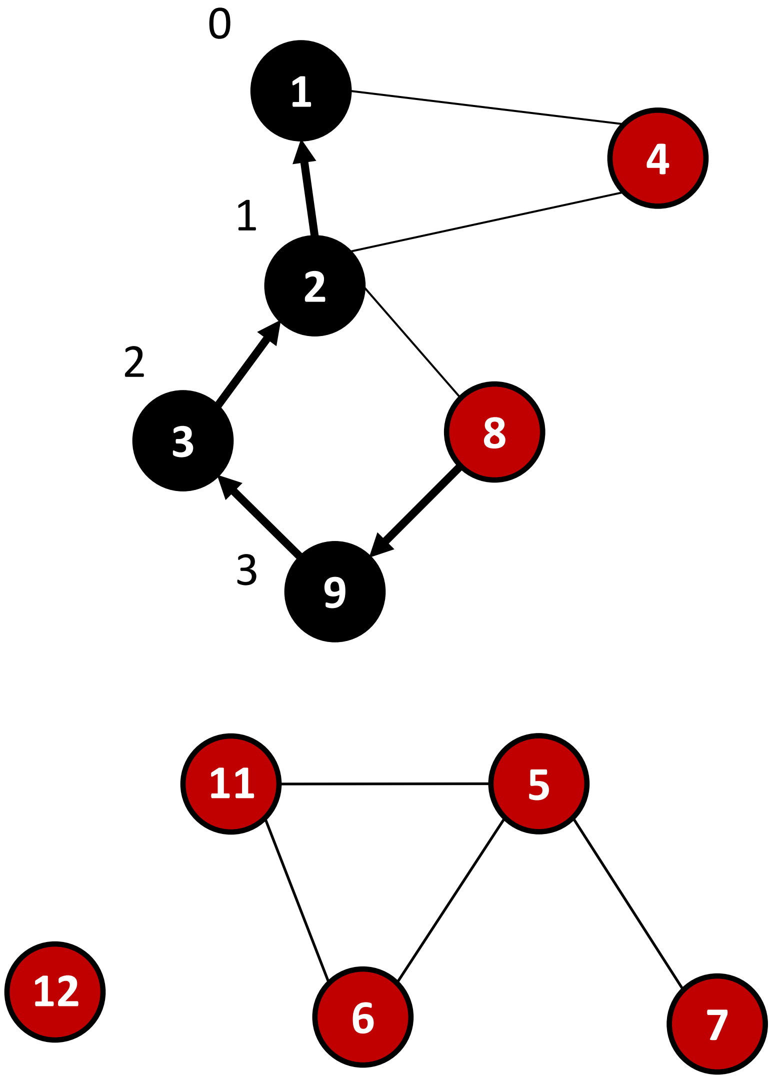
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



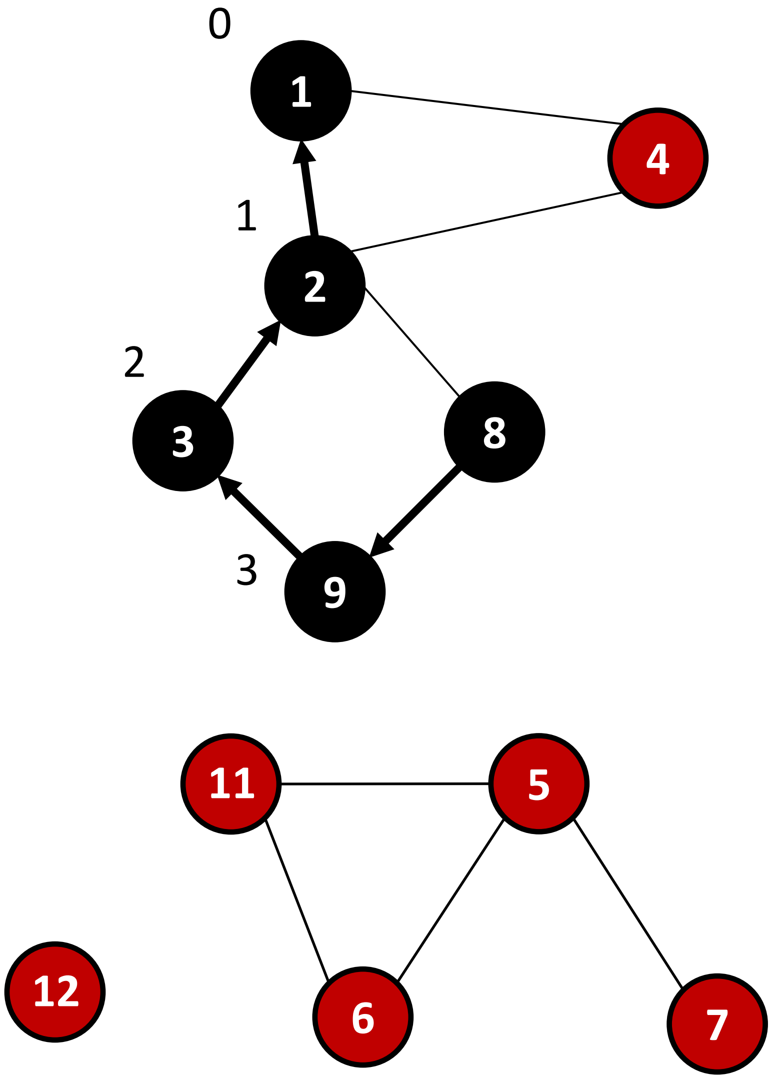
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



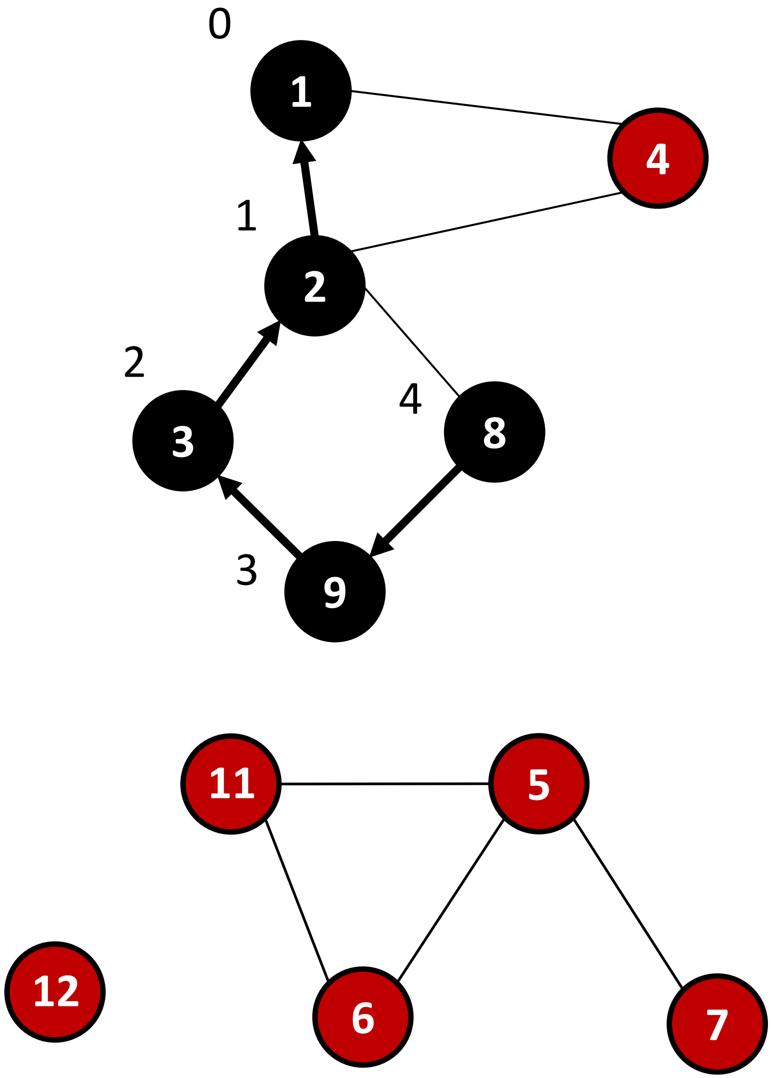
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    **DFSVisit(c)**
  node.finished = time++
```

Timestamping nodes



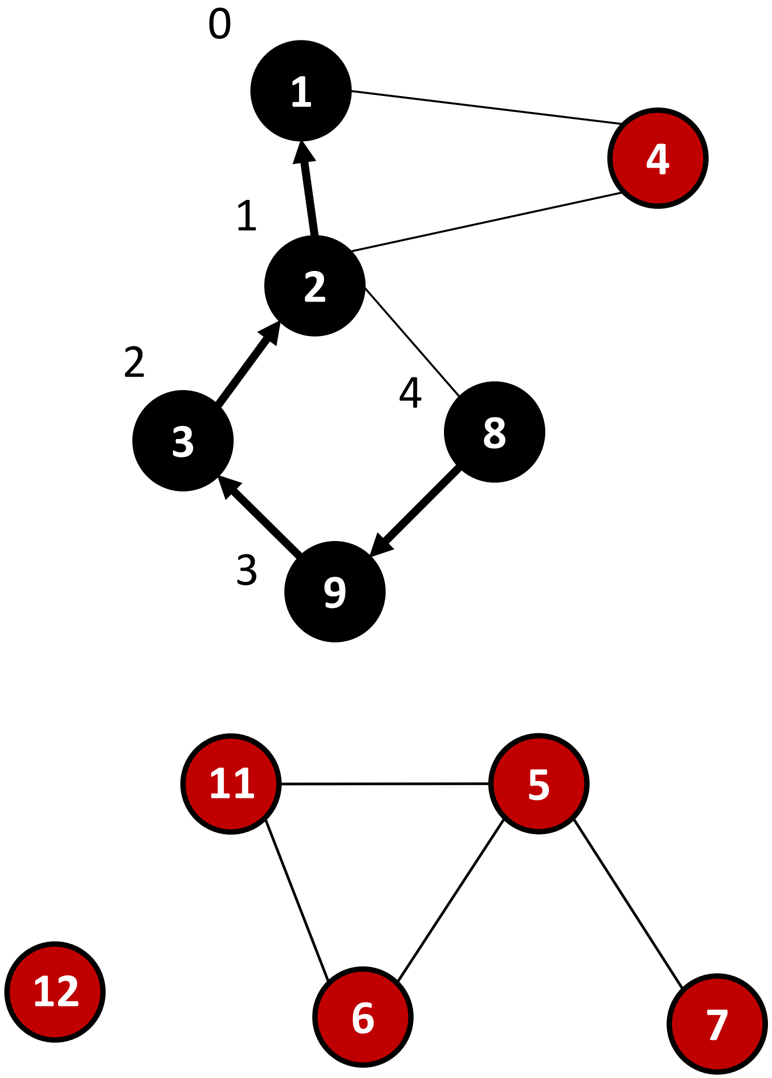
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
         c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



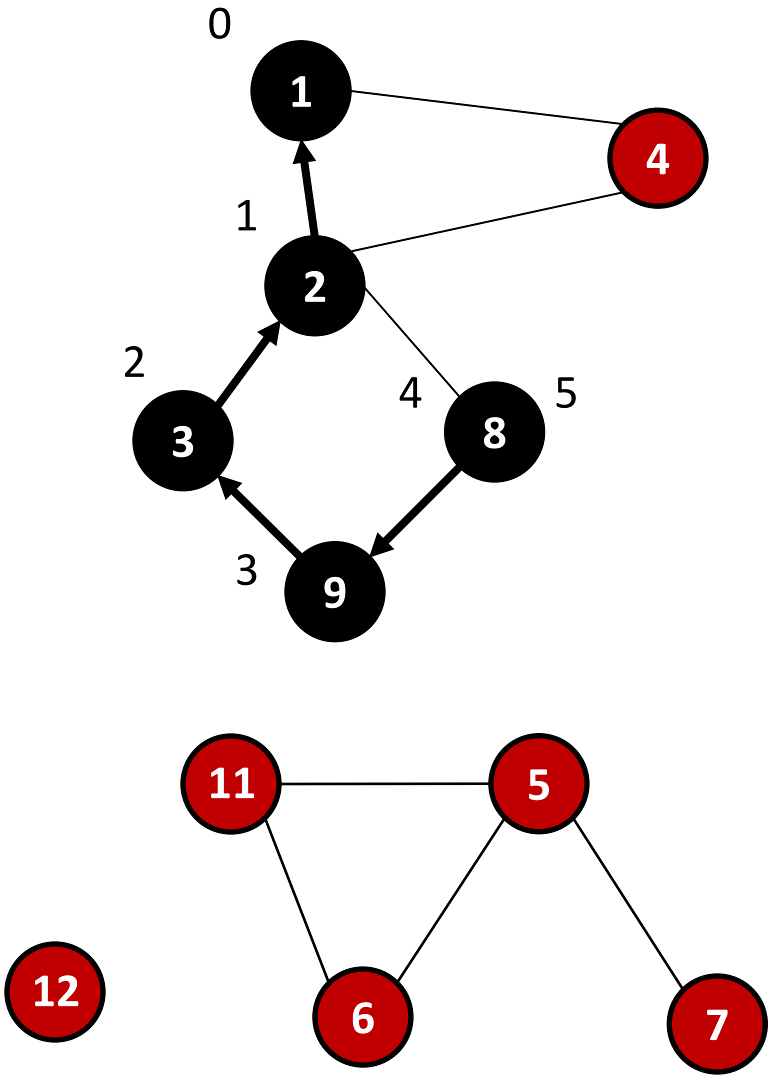
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
        DFSVisit(node)

DFSVisit(node)
  mark node visited
  **node.discovered = time++**
  for each unvisited neighbor,
            c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

What’s next?



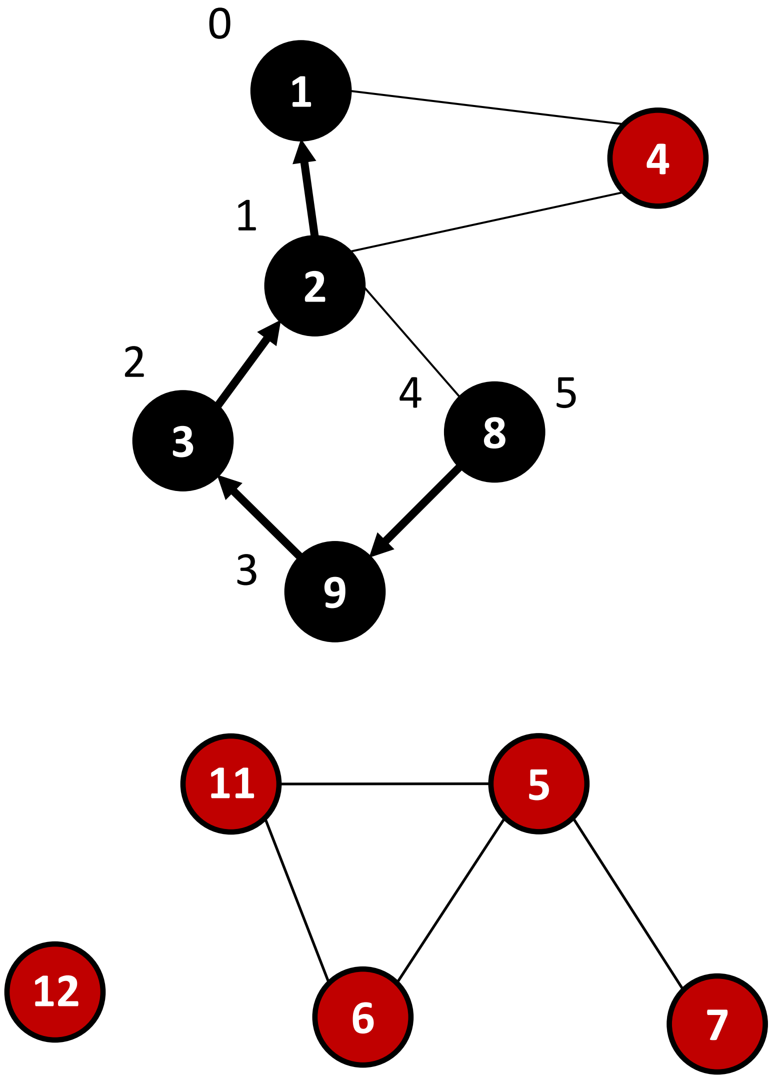
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Timestamping nodes



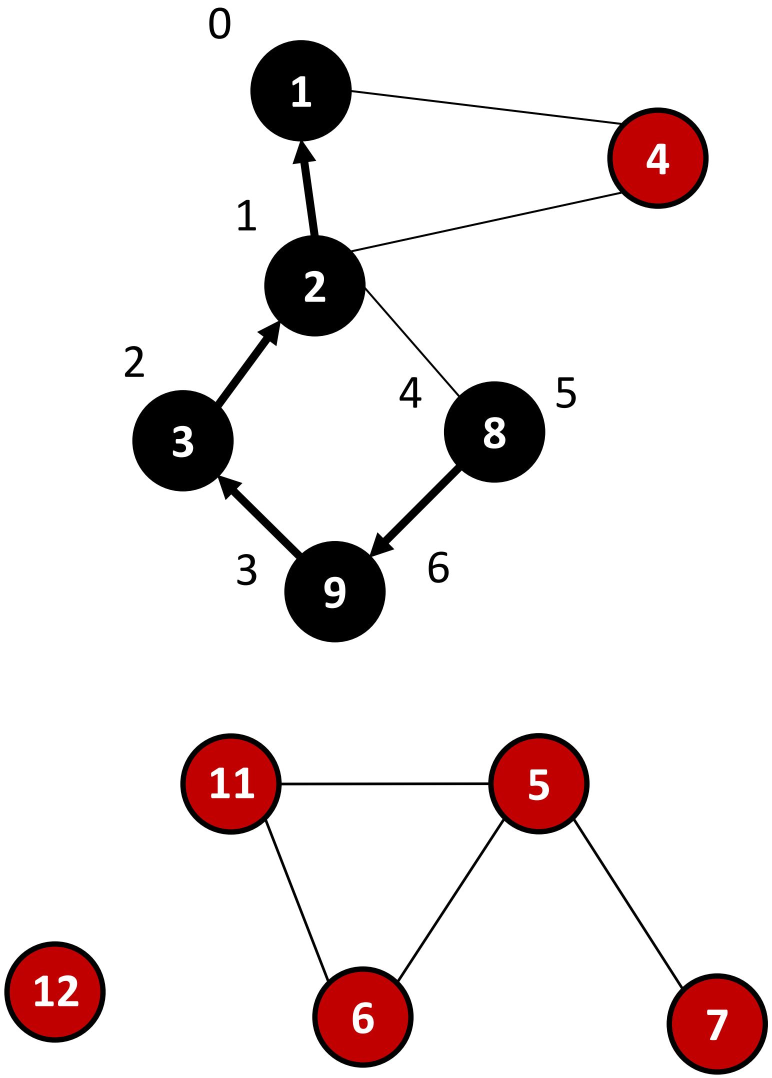
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

What’s next?



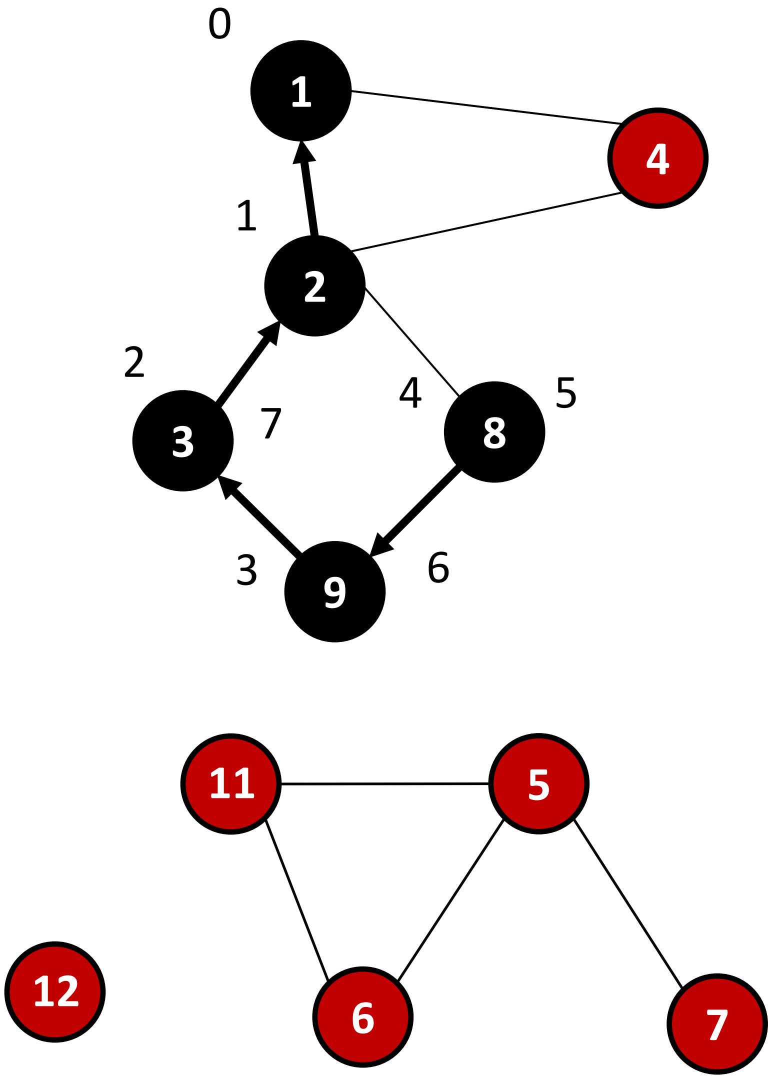
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

8 was called from 9



```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
            c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

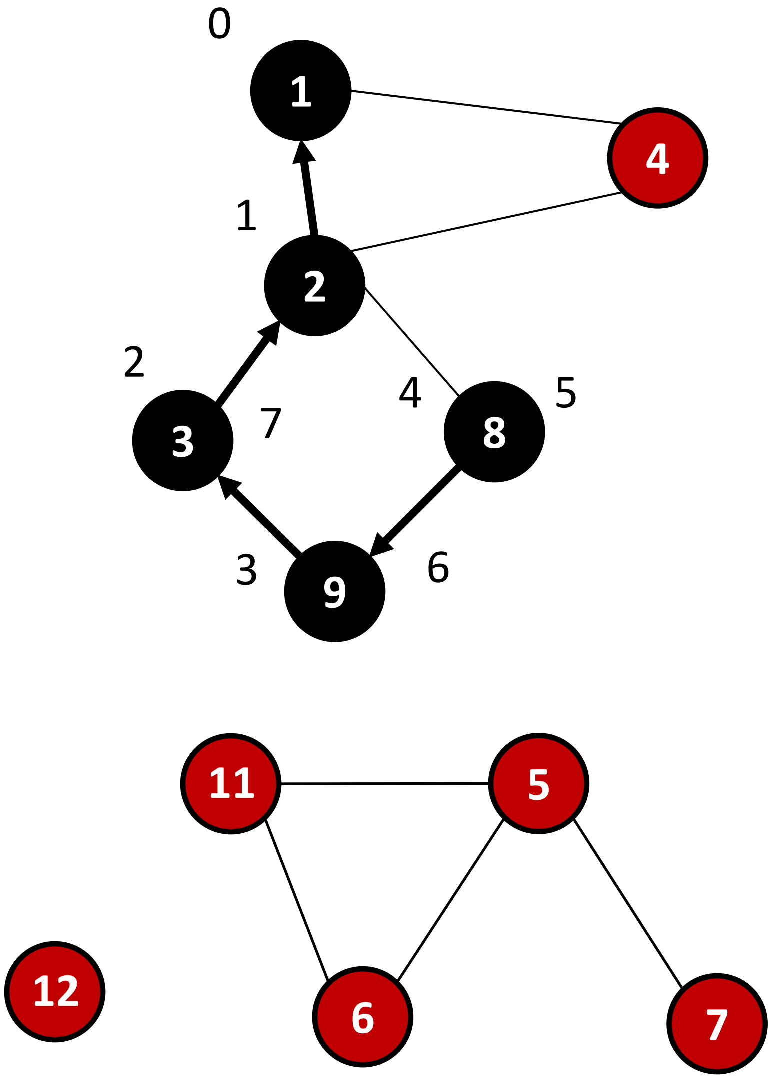
9 was called from 3



```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

3 was called from 2

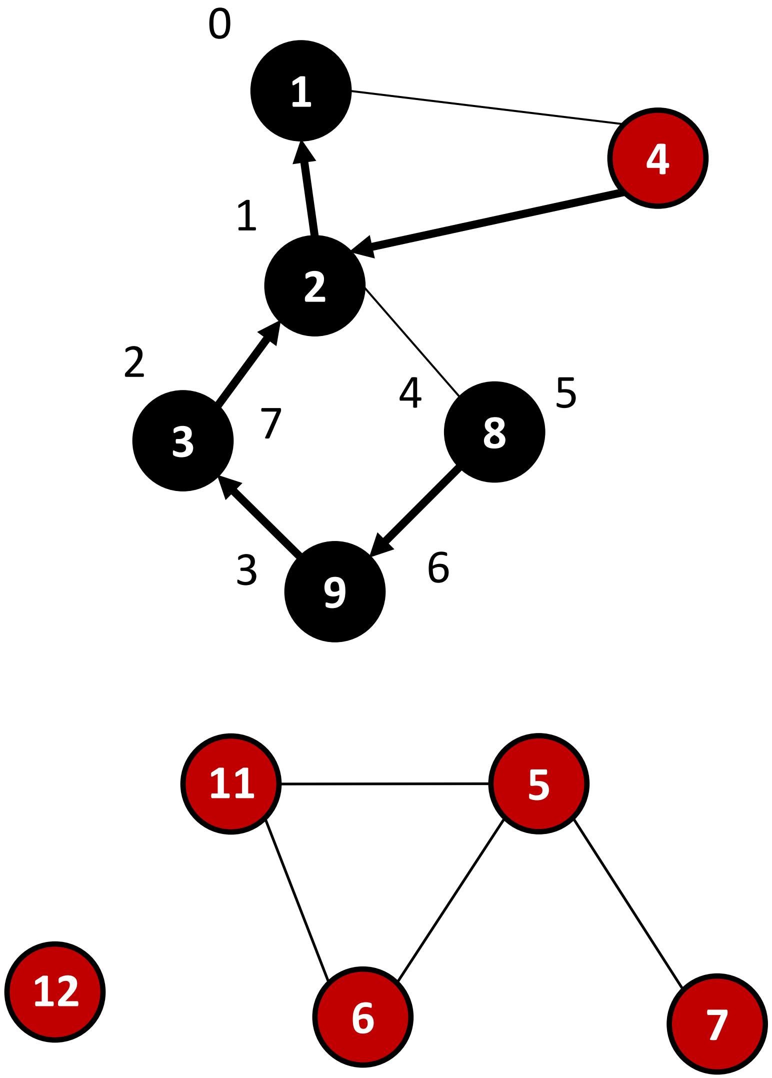
(who still has an unvisited neighbor)



```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
        DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

3 was called from 2

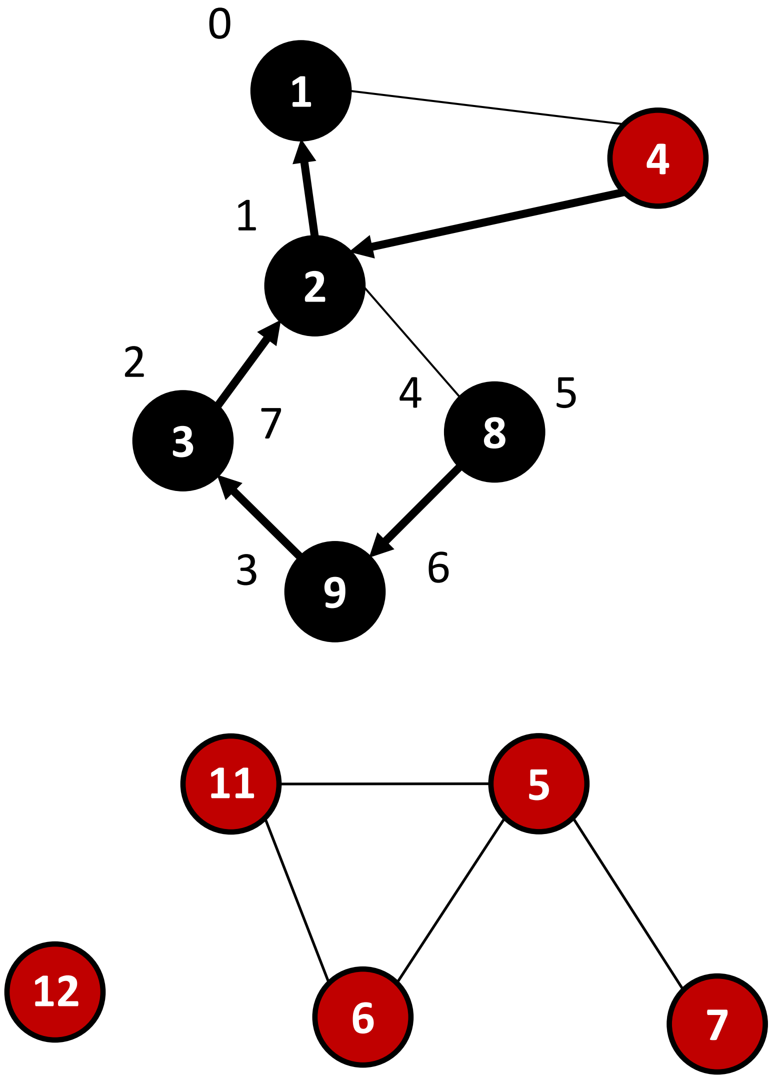
(who still has an unvisited neighbor)



```
DepthFirst()
   time = 0
   for each node in graph
        if node not visited
           DFSVisit(node)

DFSVisit(node)
   mark node visited
   node.discovered = time++
   for each unvisited neighbor,
              c, of node
      c.predecessor = node
      DFSVisit(c)
   node.finished = time++
```

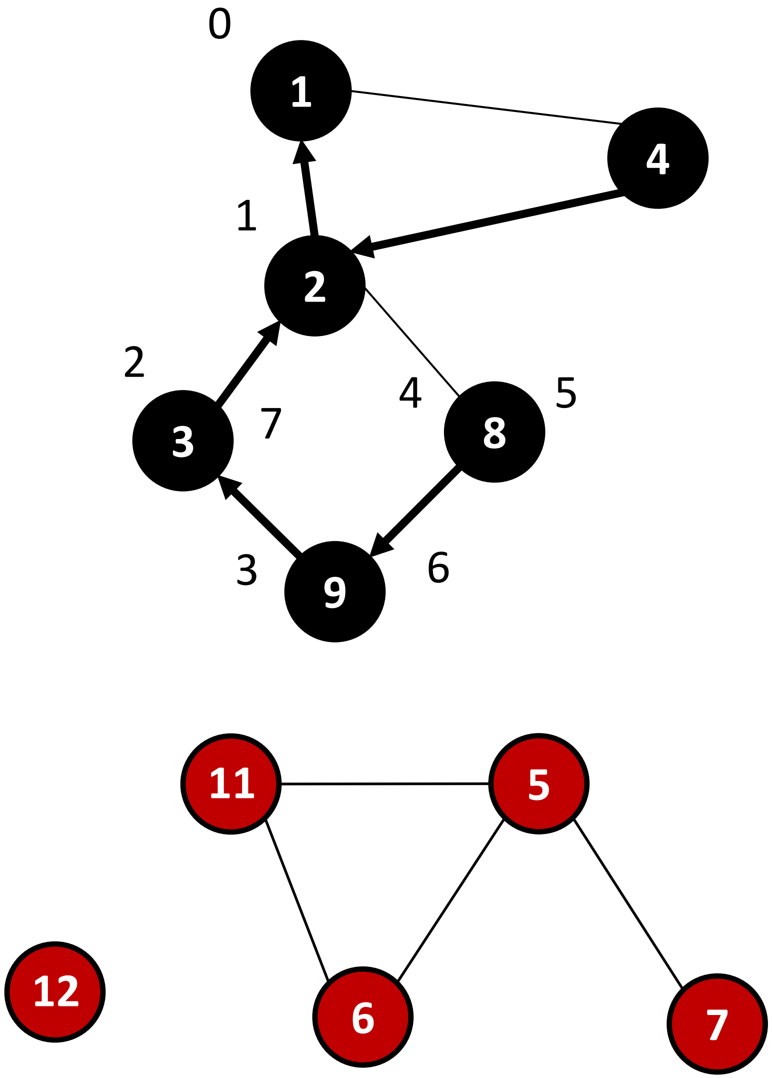
3 was called from 2

(who still has an unvisited neighbor)



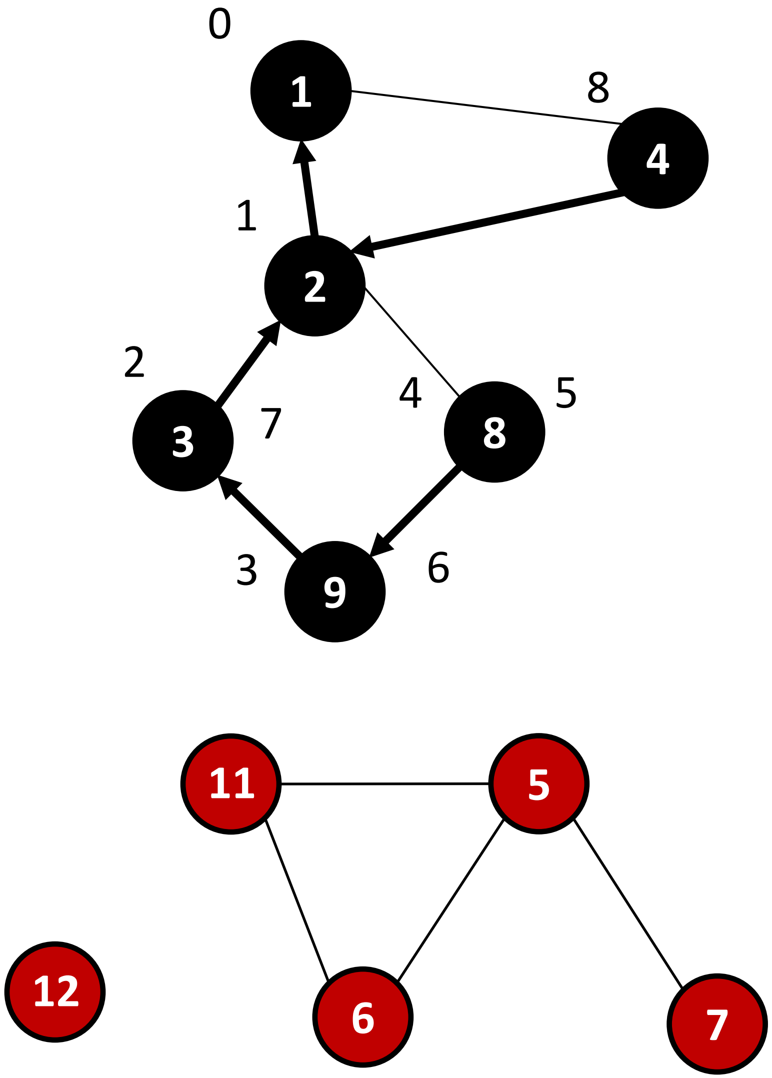
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
        DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
              c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

So now we visit 4



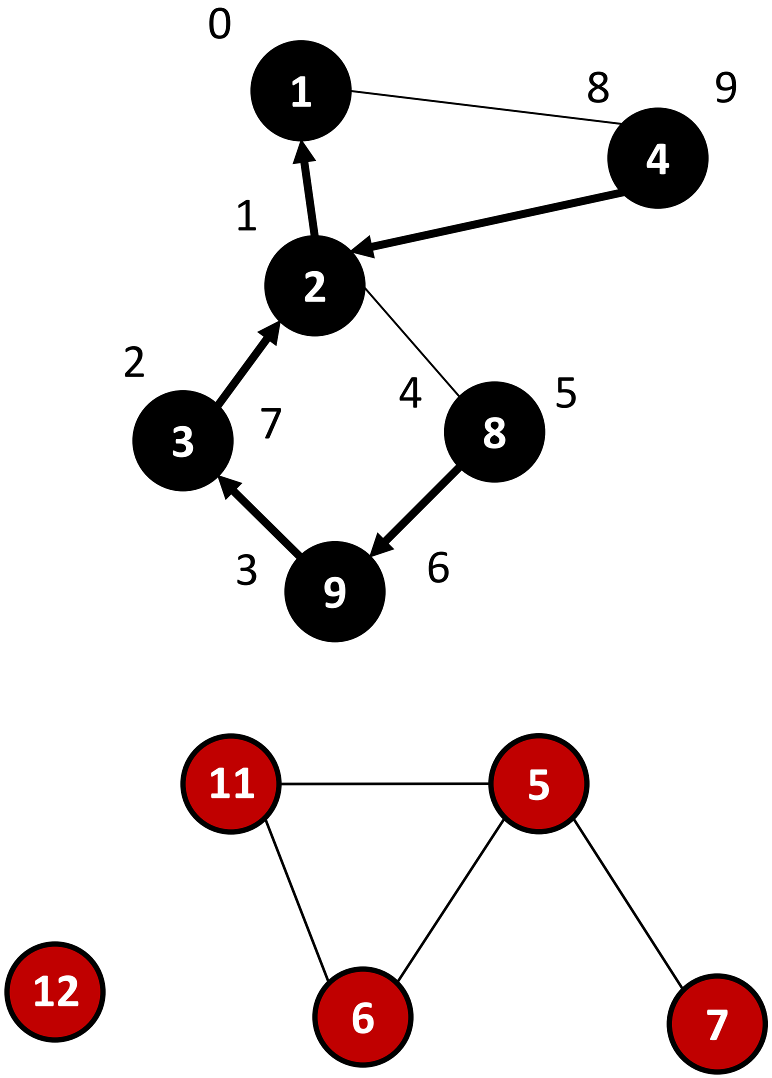
```
DepthFirst()
  time = 0
  for each node in graph
    if node not visited
      DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
      c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

So now we visit 4



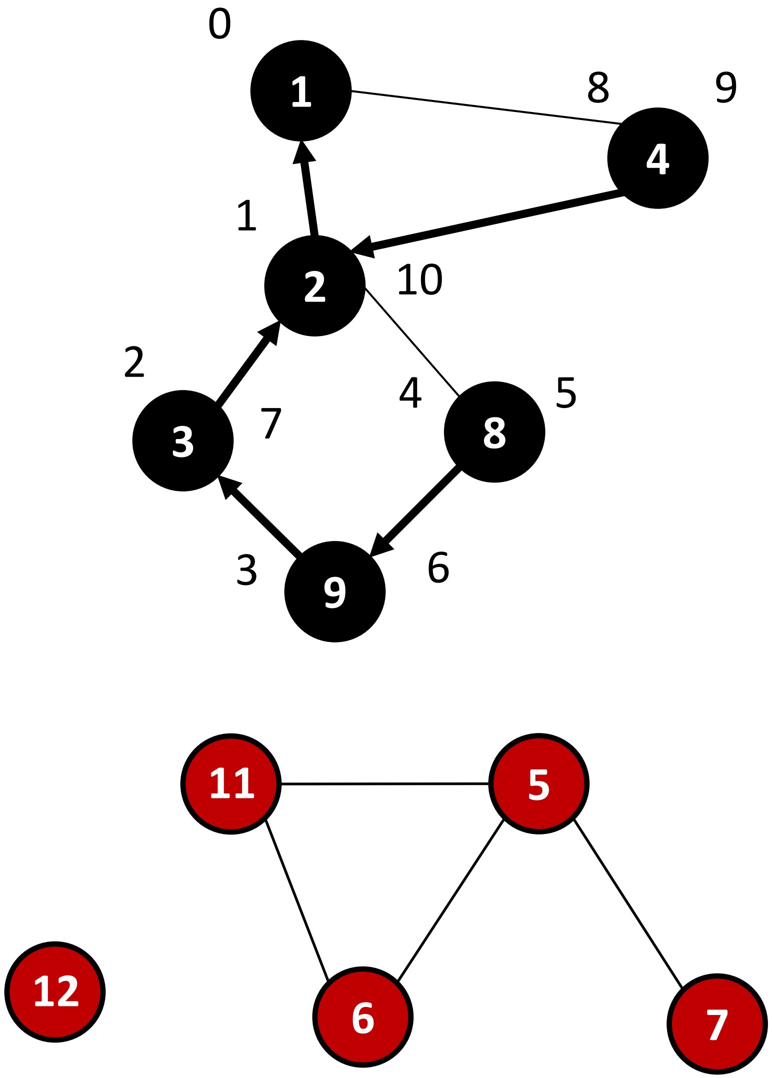
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
          DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

So now we visit 4



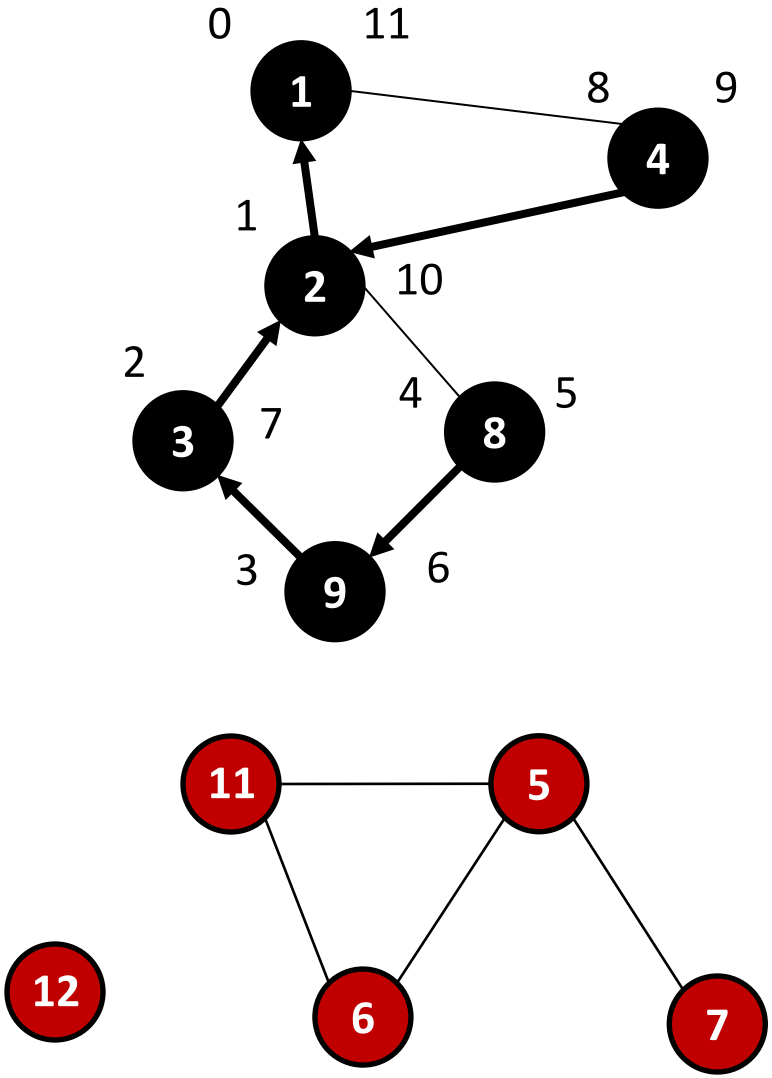
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And now we can finish off 2



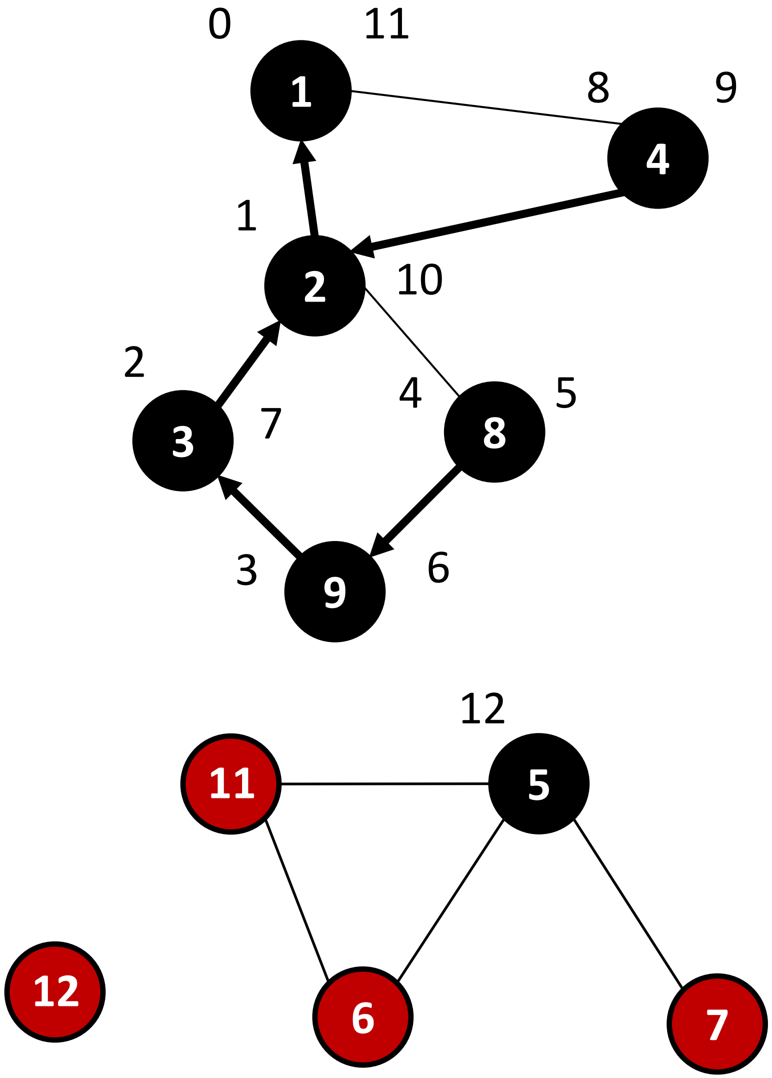
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
            c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And 1…



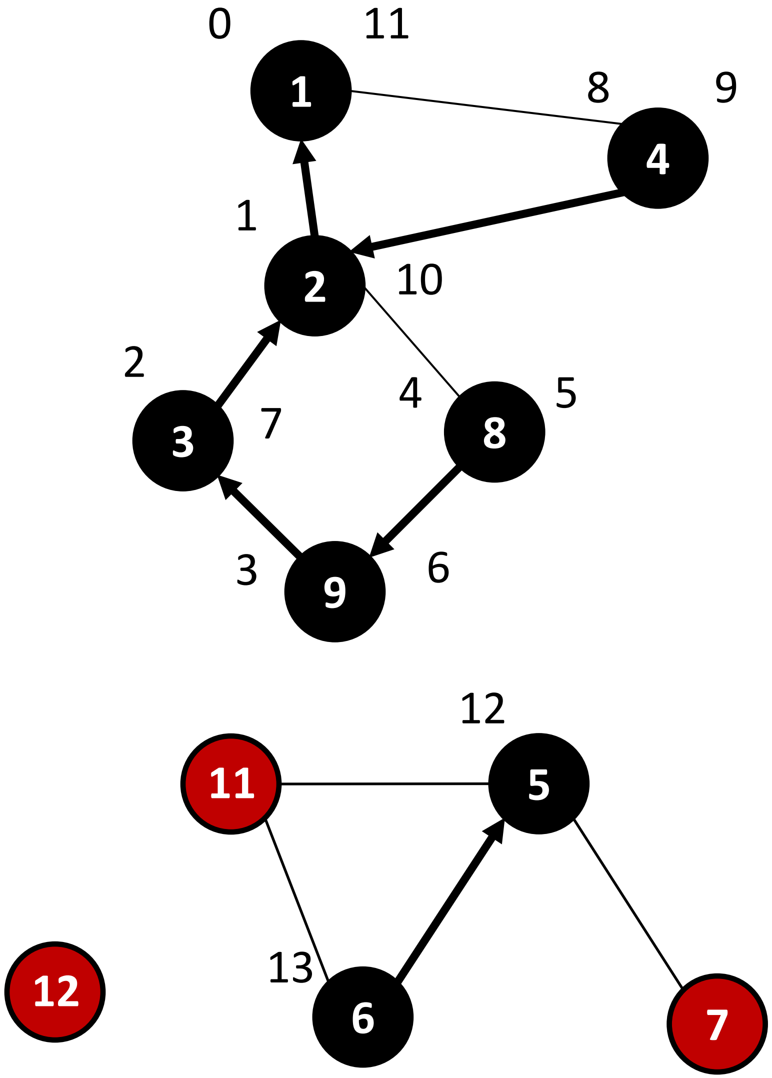
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And then we do the rest …



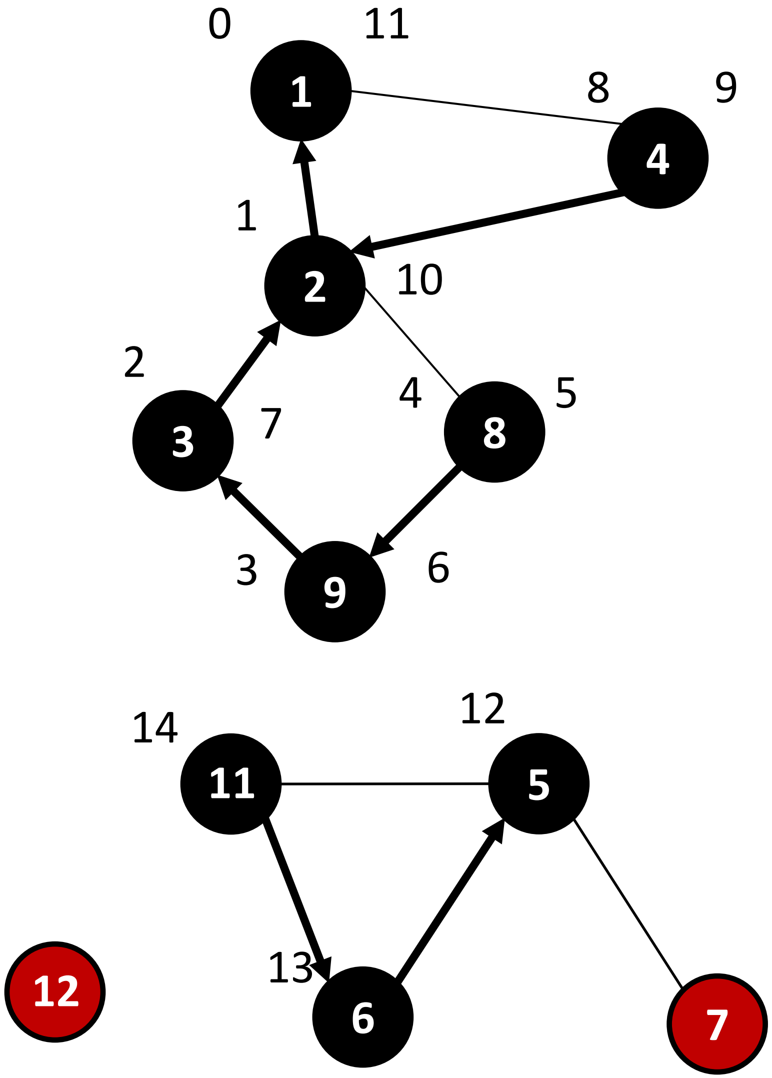
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And then we do the rest …



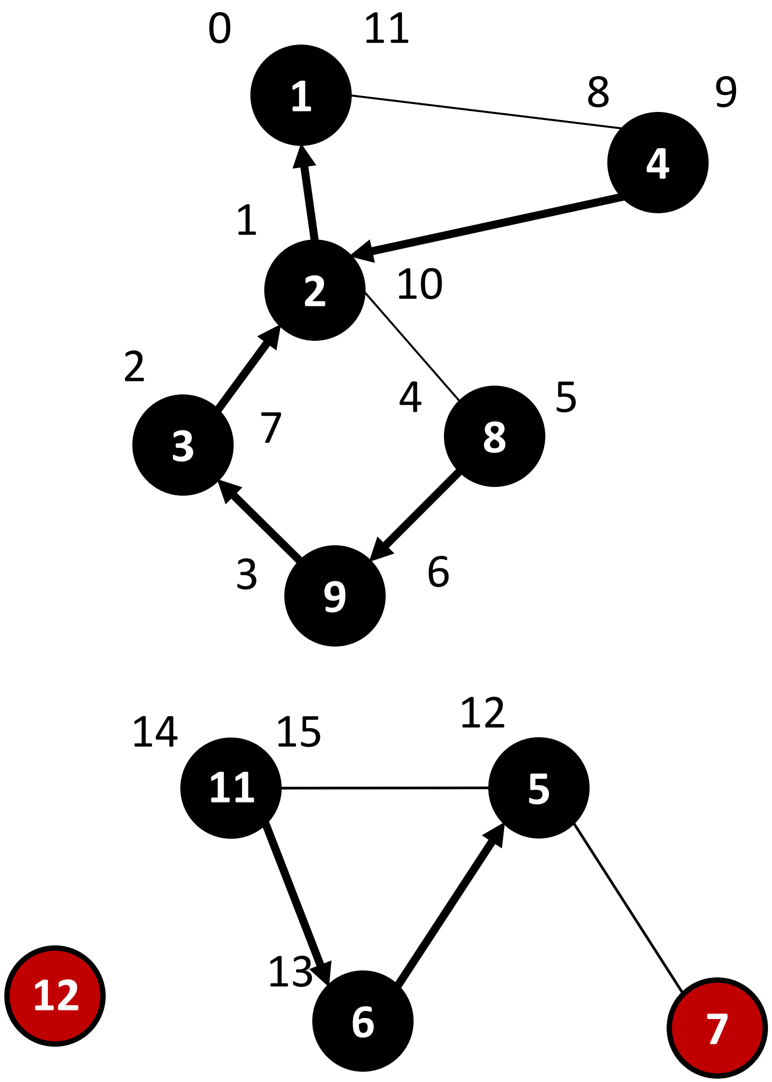
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
        DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And then we do the rest …



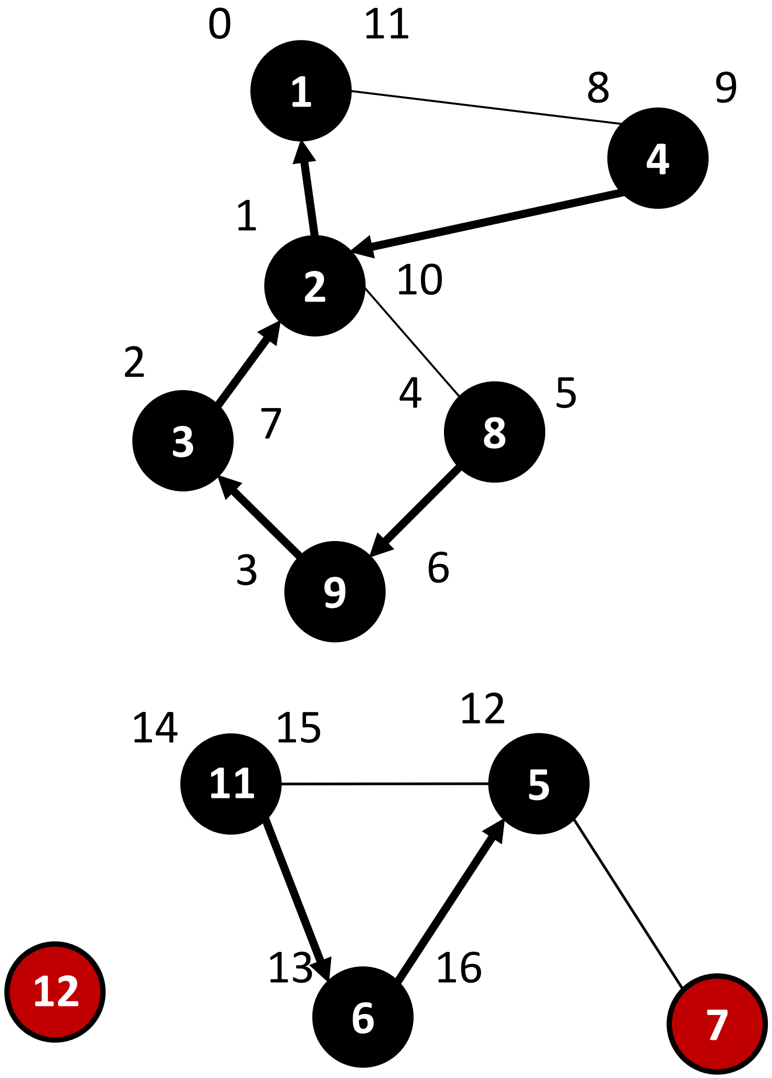
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And then we do the rest …



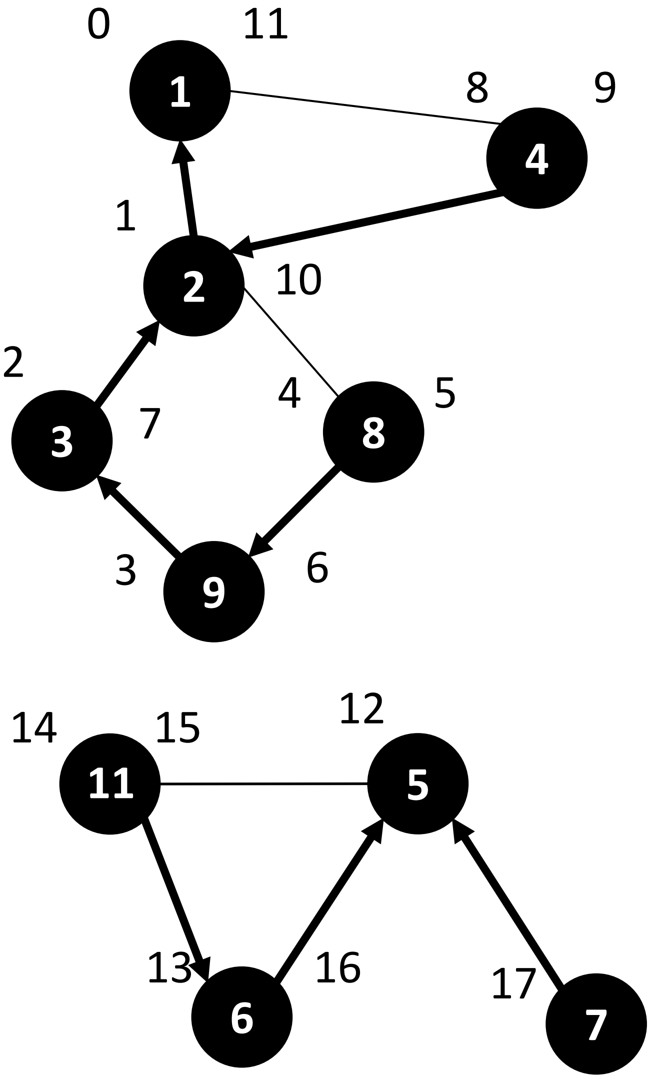
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
        DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
            c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And then we do the rest …



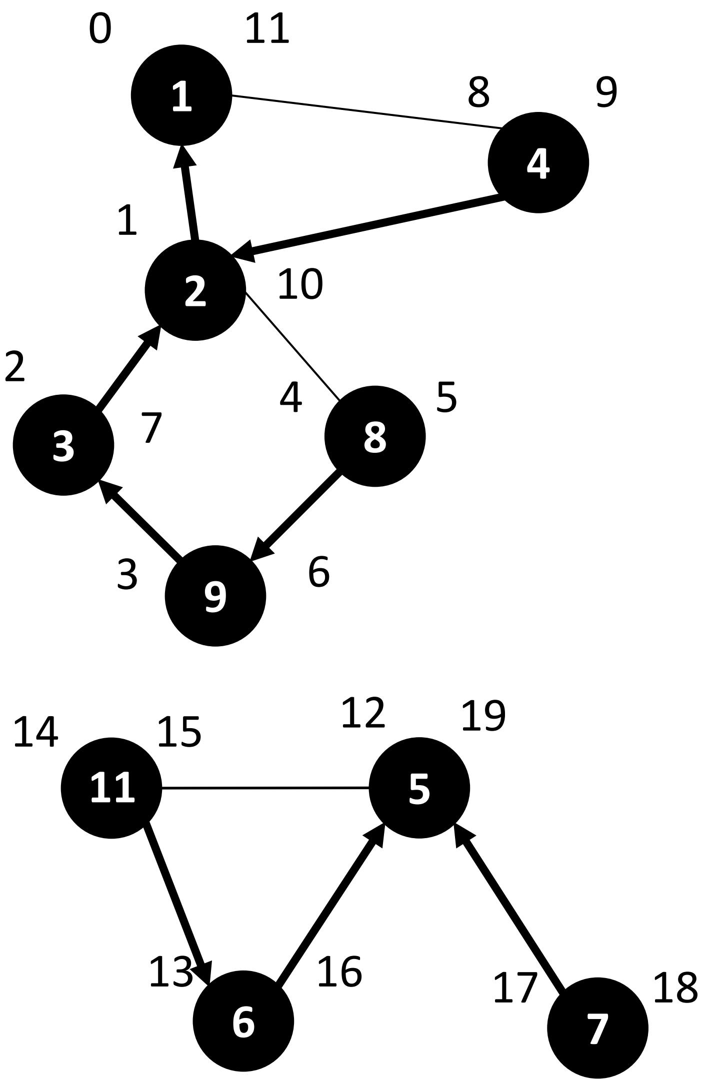
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
        DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
            c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And then we do the rest …



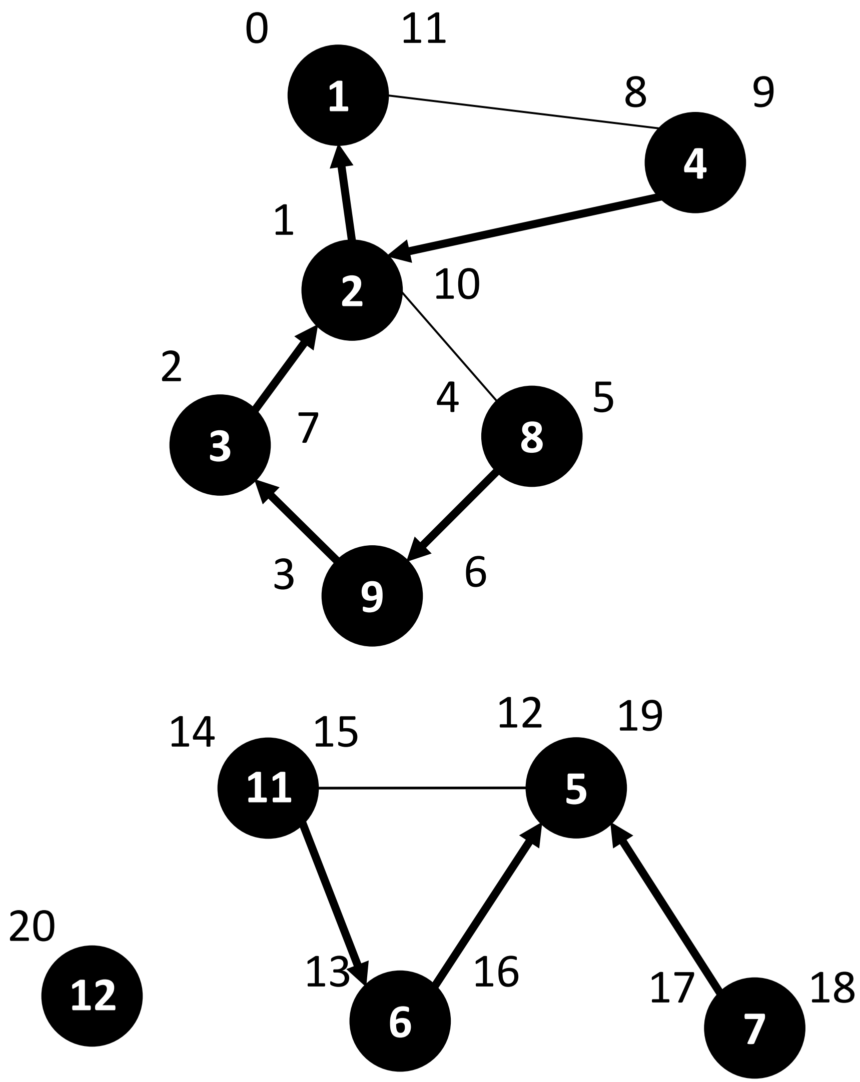
```
DepthFirst()
  time = 0
  for each node in graph
     if node not visited
       DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And then we do the rest …



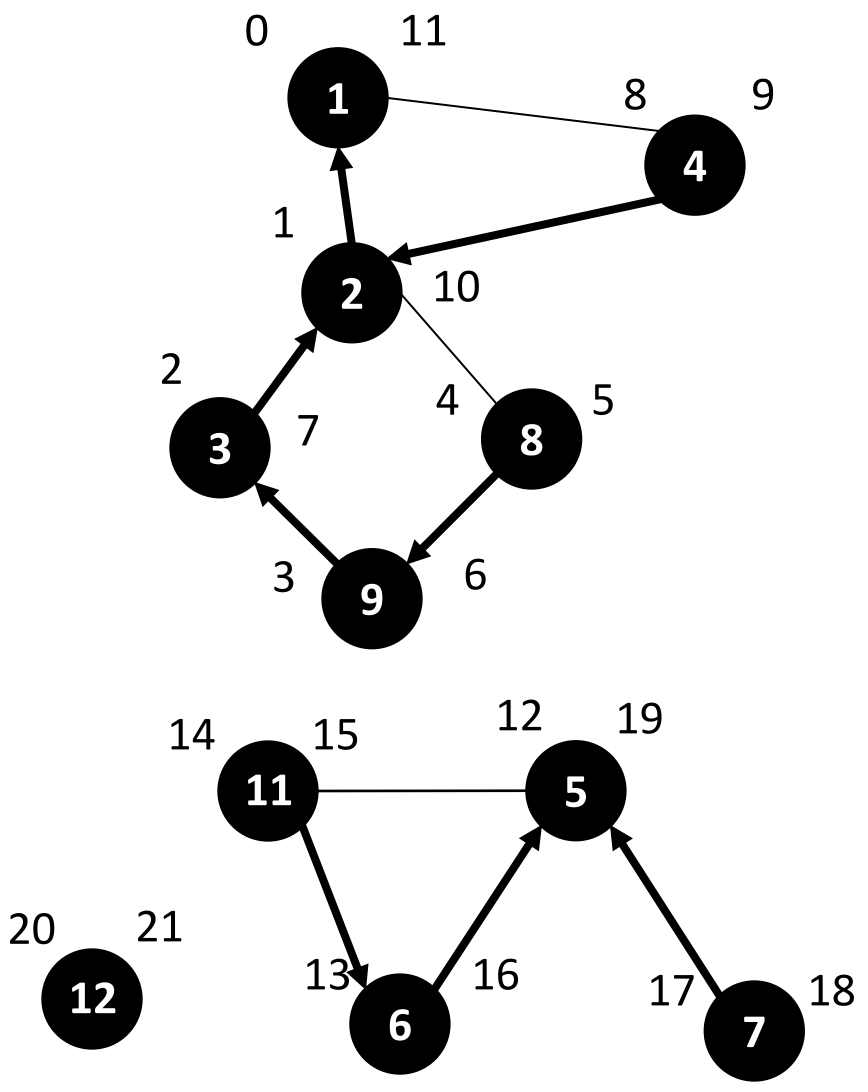
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
        DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
          c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

And then we do the rest …



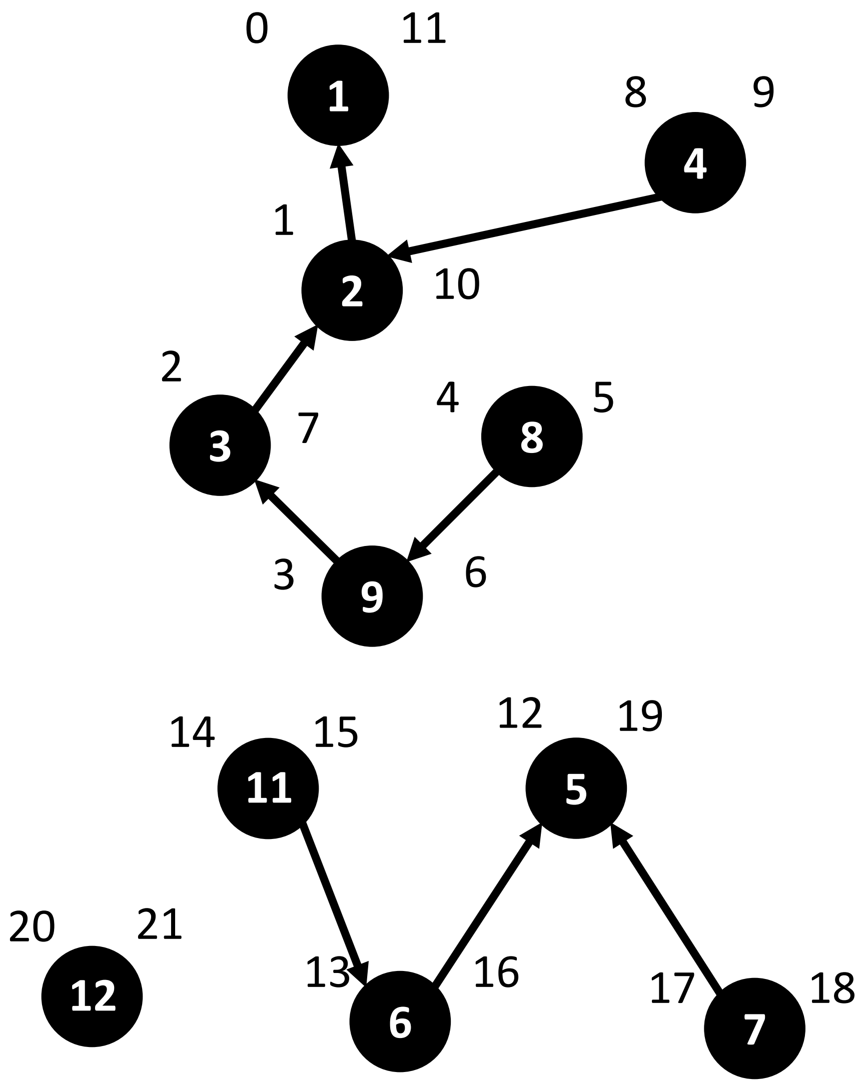
```
DepthFirst()
  time = 0
  for each node in graph
      if node not visited
         DFSVisit(node)

DFSVisit(node)
  mark node visited
  node.discovered = time++
  for each unvisited neighbor,
           c, of node
    c.predecessor = node
    DFSVisit(c)
  node.finished = time++
```

Depth-first forest



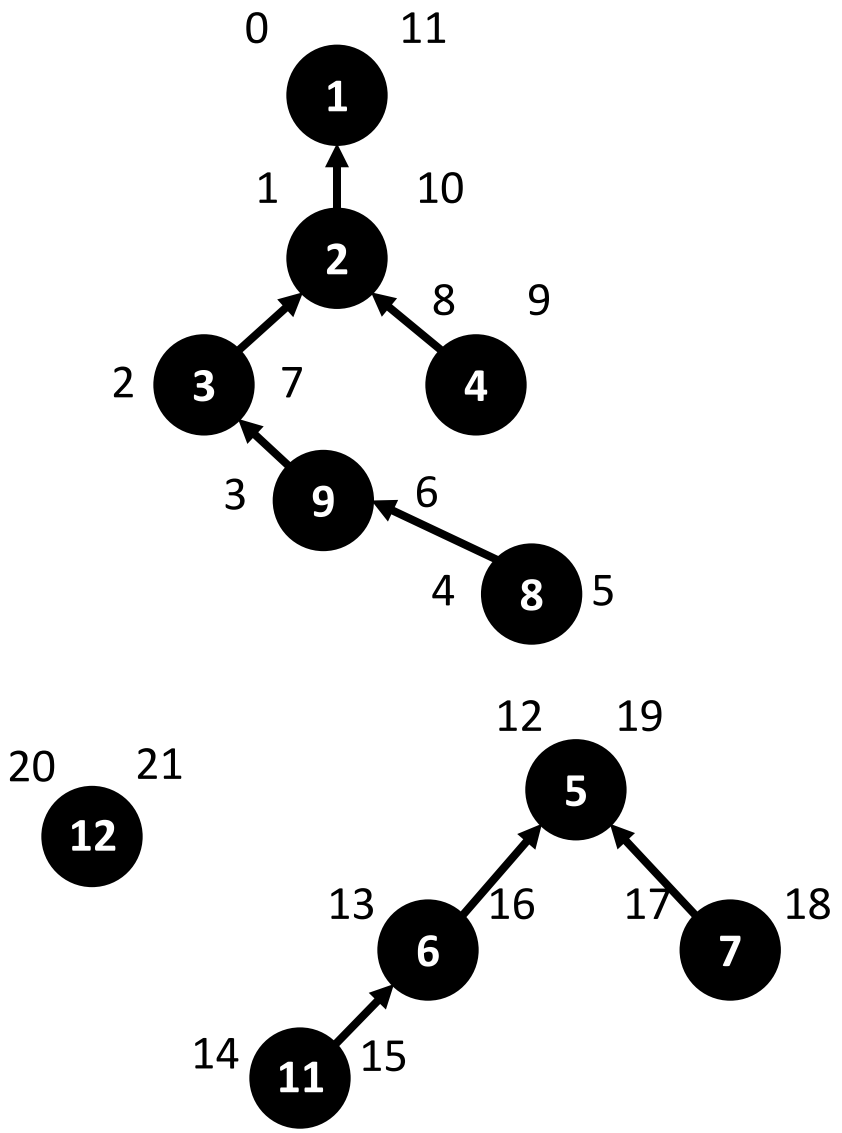
- Once again, the **predecessor links form a tree**
  - Rooted at the point where the DFSVist was started
  - Called a **depth-first tree**
- Except, technically there are **several trees**
  - One for every connected component
- So this is called a **depth-first forest**

Depth-first forest



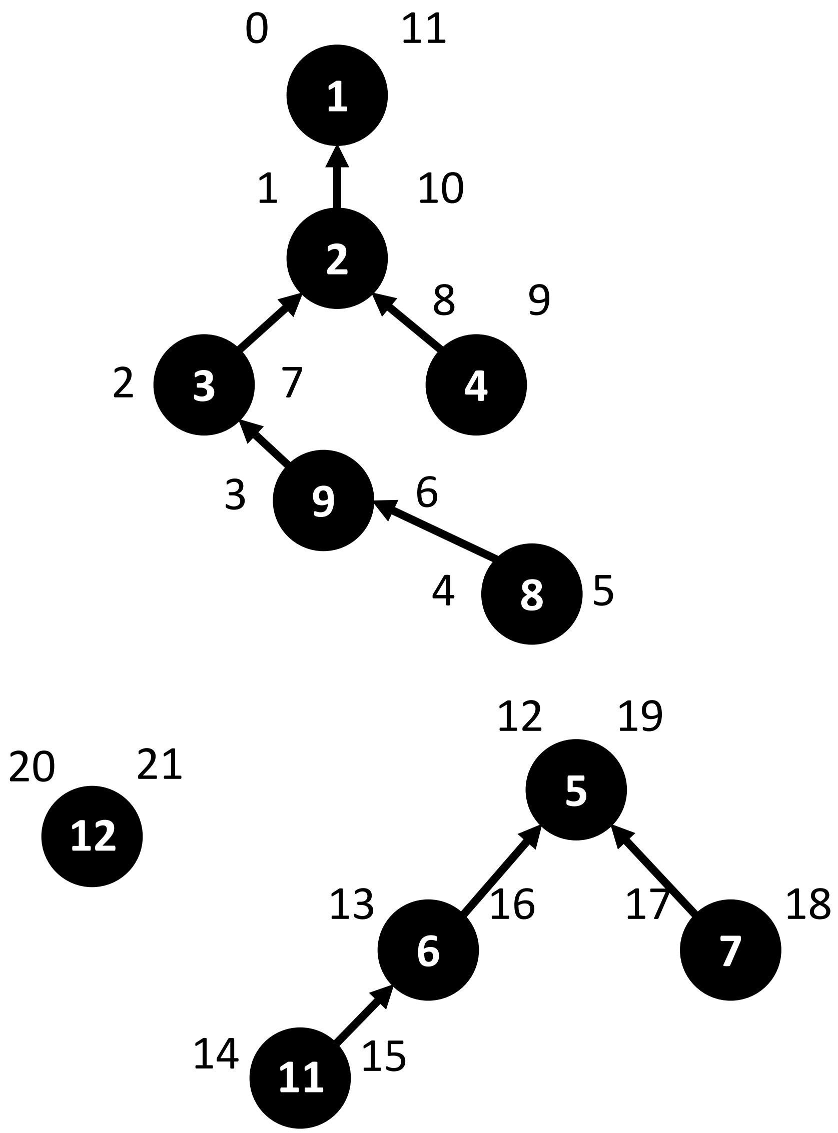
- Once again, the **predecessor links form a tree**
  - Rooted at the point where the DFSVist was started
  - Called a **depth-first tree**
- Except, technically there are **several trees**
  - One for every connected component
- So this is called a **depth-first forest**

Depth-first forest



- Once again, the **predecessor links form a tree**
  - Rooted at the point where the DFSVist was started
  - Called a **depth-first tree**
- Except, technically there are **several trees**
  - One for every connected component
- So this is called a **depth-first forest**

Properties of depth-first forests



- For any vertex v
  - v.discovered < v.finished

- If c is a child of p

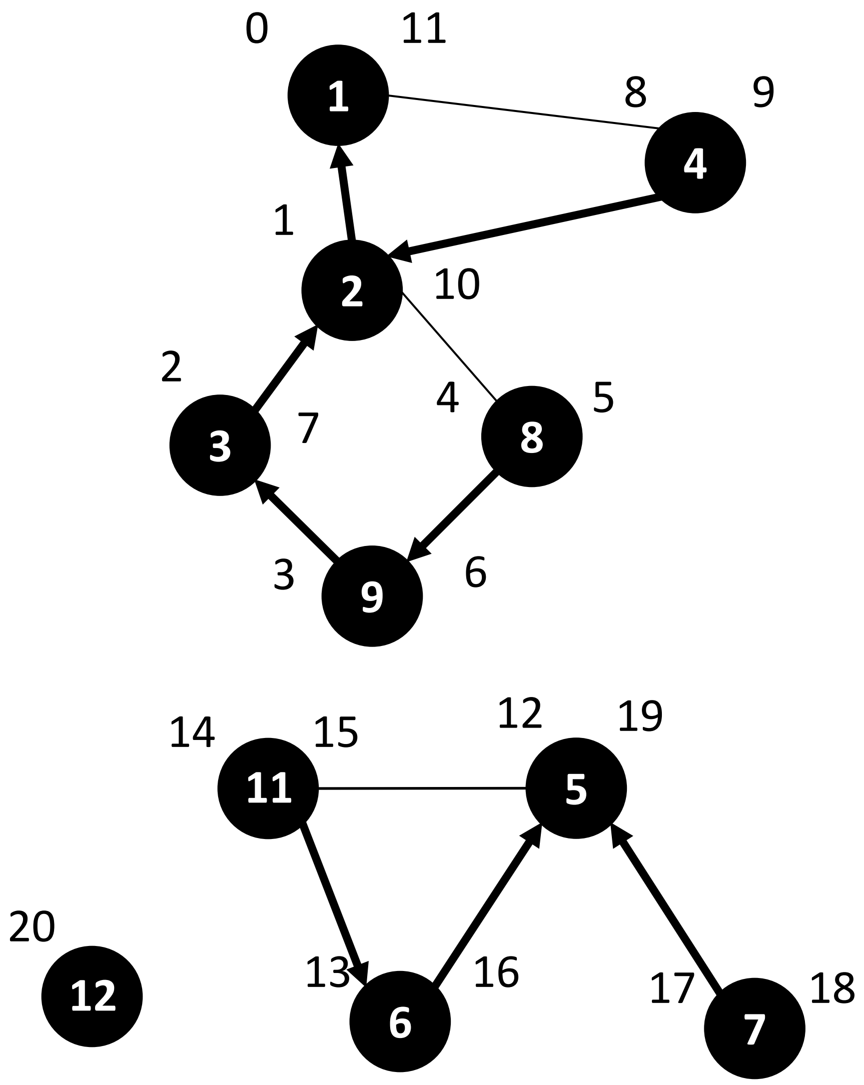
  p.discovered

  < c.discovered

  < c.finished

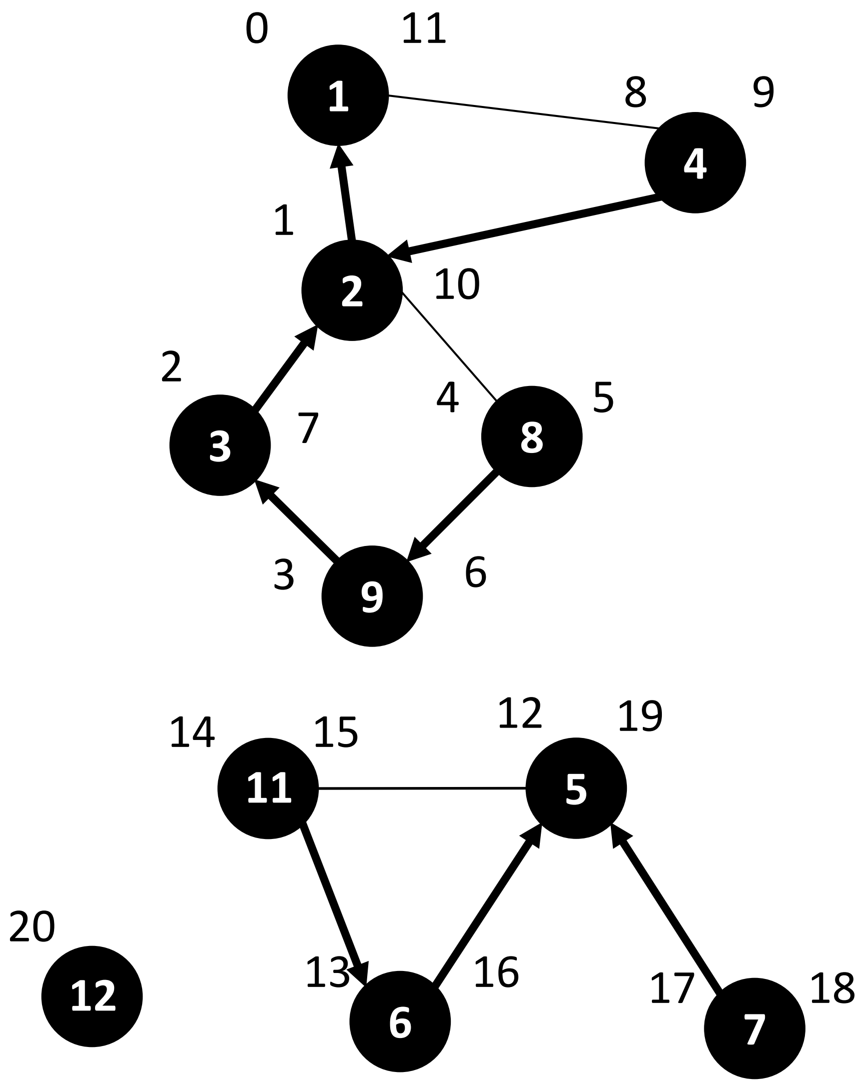
  < p.finished

Time intervals



- **Definition:** Let v be a vertex. Its **time interval** is the range of times v.discovered through v.finished
  - A.k.a.: [v.discovered, v.finished]
  - A.k.a.: the set of times during which DFSVisit(v) was running

Parenthesis theorem



- For any vertices u and v, **one of the following** three conditions must hold:
  - Their intervals are **disjoint**
  - u’s interval is a **subset** of v’s
    - And so **u is a descendant of v** in the depth-first forest
  - v‘s interval is a **subset** of u’s
    - And so **u is a ancestor of v**

Reading

- "**Elementary graph algorithms**" from CLR
    - Fairly theoretical
    - Read the proofs, but don't stress out about them