CSE 332 Autumn 2026
Lecture 16: Graphs 3

Ruth Anderson

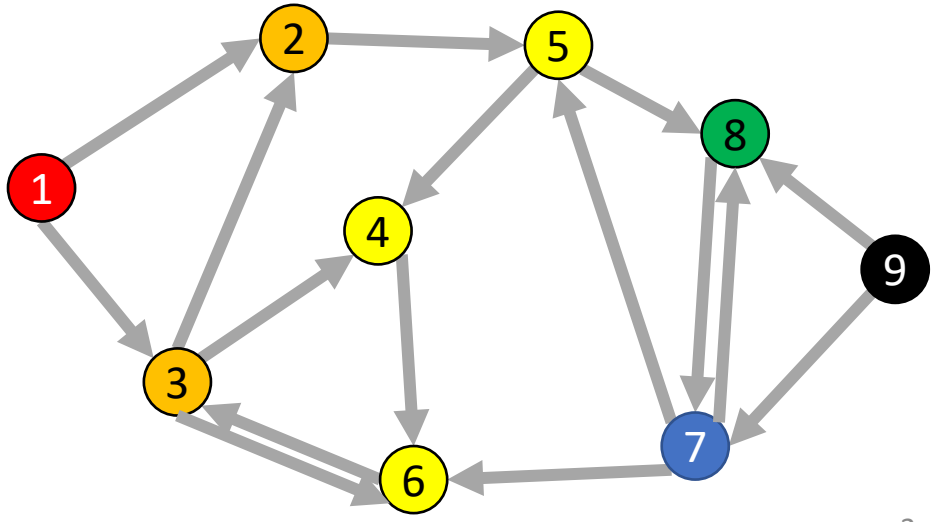
http://www.cs.uw.edu/332

Announcements

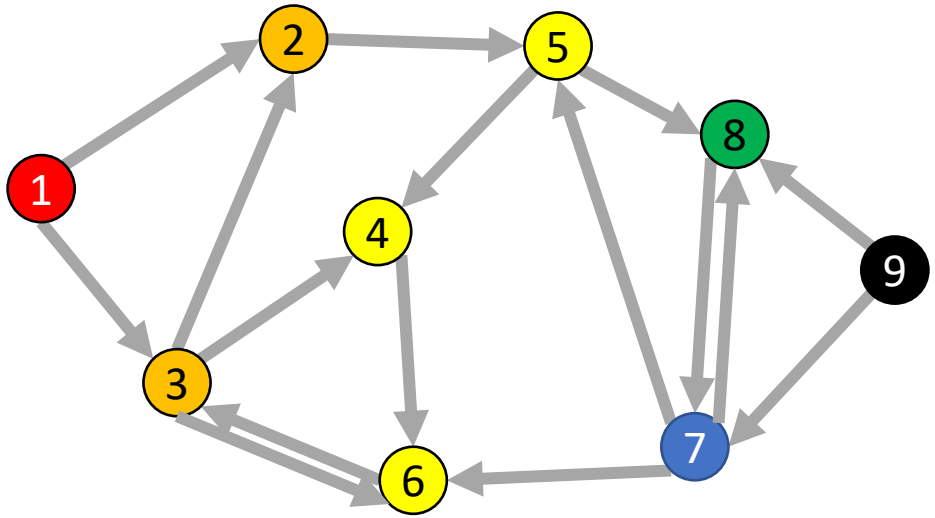
- “Meet the staff” activity – Come meet us in office hours!
  - Will keep this open thru end of week 6 (Nov 6), but will be de-prioritized in OH
- Participate in Section, earn a late day (see Ed Board post)
- Exercise 5 Hashing: Due Friday Nov 6
- Exercise 6 Sorting: Due Friday Nov 13
- Exercise 7 Cycle Detection: Due Monday Nov 16
- Midterm: 5-6pm Monday November 9
  - **Must go to room for Lecture you are Registered for:**
    - **Lecture A: Smith 120 & Lecture B: Smith 205**
  - Review during both lecture sections on Fri Nov 6
  - No lecture on Mon Nov 9
  - Sample exams posted! See exams page.

Breadth-First Search

- Input: a node s
- Behavior: Start with node s, visit all neighbors of s, then all neighbors of neighbors of s, …
- Visits every node reachable from s in order of distance
- Output:
  - How long is the shortest path?
  - Is the graph connected?



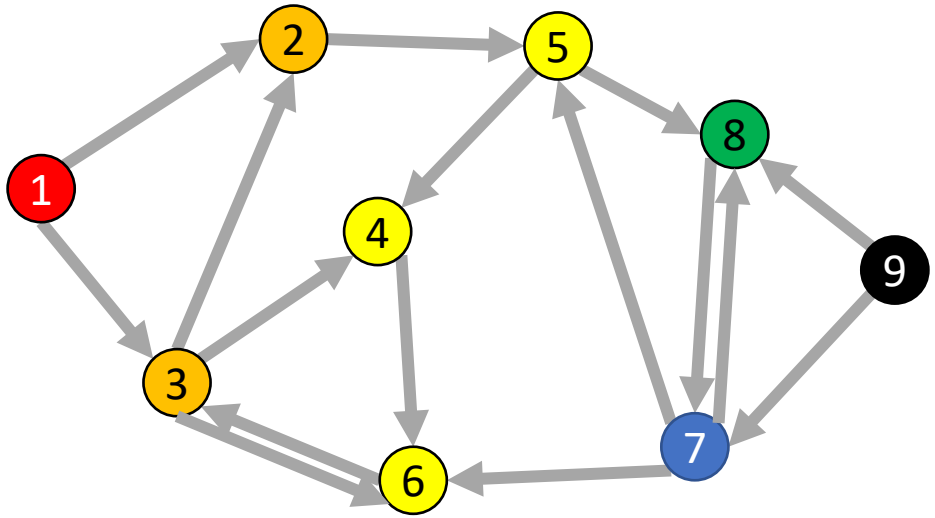
BFS – Pseudocode




```
void bfs(graph, s) {
        found = new Queue();
        found.enqueue(s);
        mark s as “visited”;
        while (!found.isEmpty()) {
                current = found.dequeue();
                for (v : neighbors(current)) {
                        if (! v marked “visited”) {
                                mark v as “visited”;
                                found.enqueue(v);
                        }
                }
        }
}
```

Running time: $\Theta(|V| + |E|)$

BFS(node 1) – Worked Example

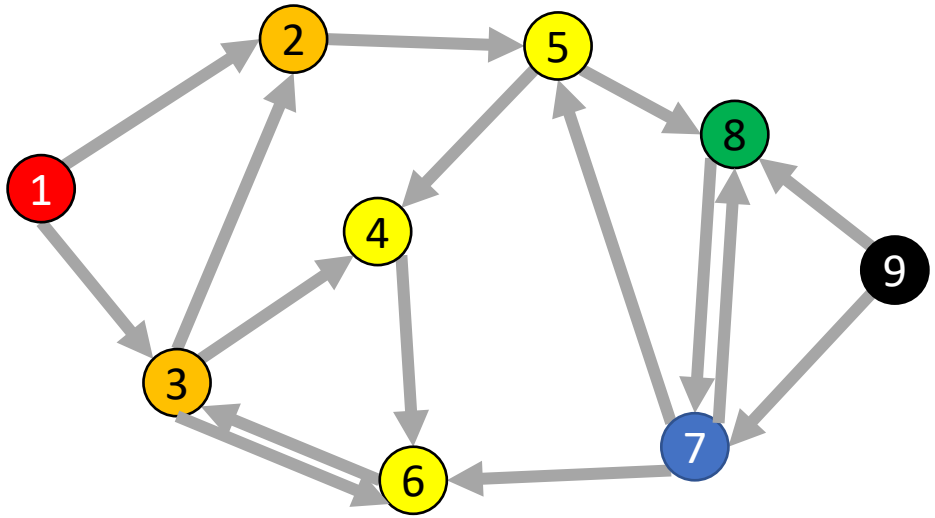


Node	Visited?	Other Info
1	True	
2		
3		
4		
5		
6		
7		
8		
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited

Queue: 1

BFS(node 1) – Step 1: Dequeue 1



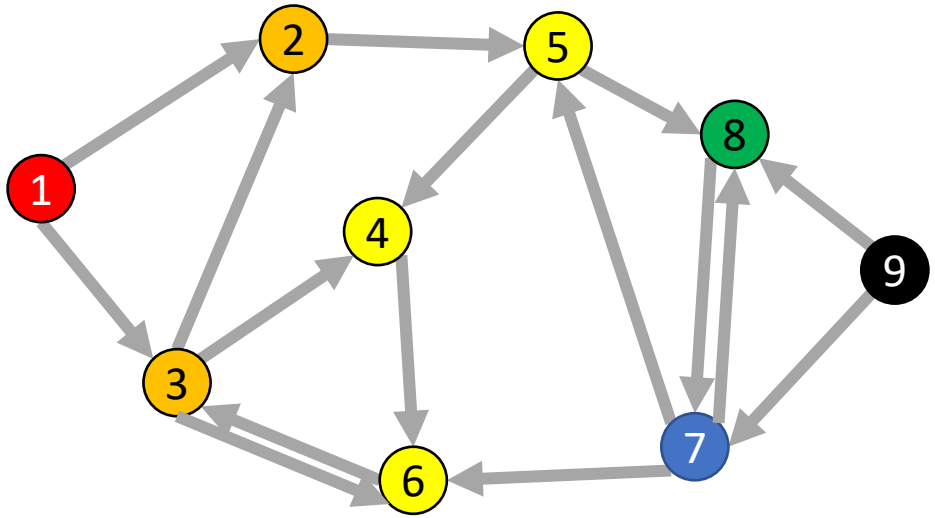
Node	Visited?	Other Info
1	True	
2	True	
3	True	
4		
5		
6		
7		
8		
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited

Queue: ~~1,~~ 2, 3

Note: 2 and 3 could be enqueued in any order

BFS(node 1) – Step 2: Dequeue 2

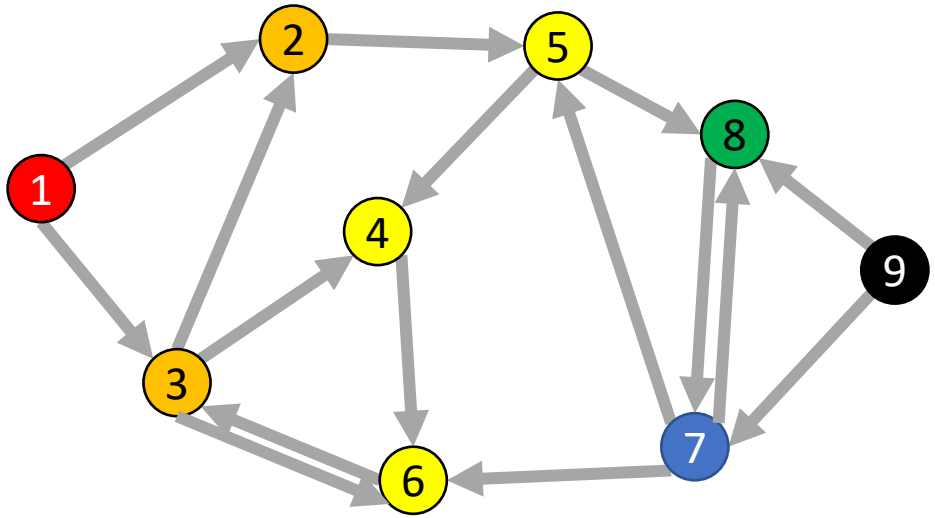


Node	Visited?	Other Info
1	True	
2	True	
3	True	
4		
5	True	
6		
7		
8		
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited

Queue: ~~1, 2,~~ 3, 5

BFS(node 1) – Step 3: Dequeue 3



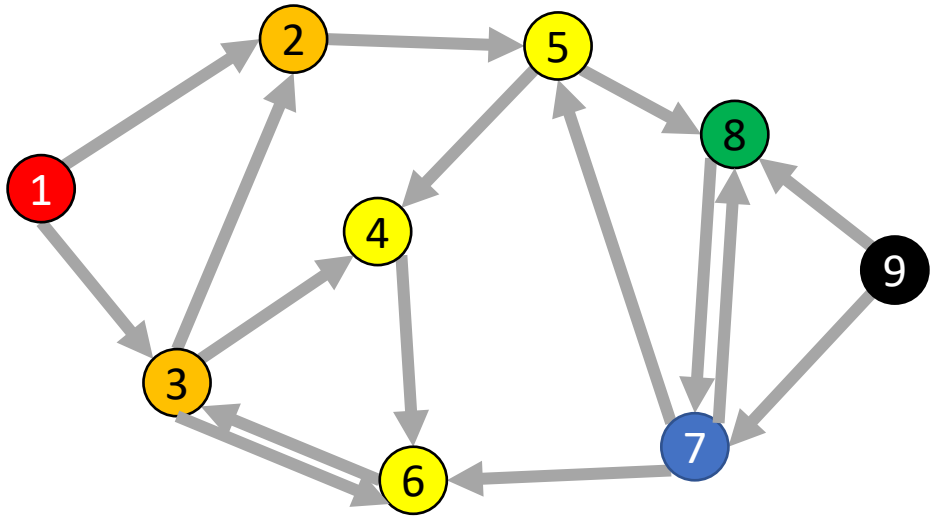
Node	Visited?	Other Info
1	True	
2	True	
3	True	
4	True	
5	True	
6	True	
7		
8		
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited

Queue: ~~1, 2, 3,~~ 5, 4, 6

Note: 4 and 6 could be enqueued in any order

BFS(node 1) – Step 4: Dequeue 5



Node	Visited?	Other Info
1	True	
2	True	
3	True	
4	True	
5	True	
6	True	
7		
8	True	
9		

For each node: (dequeue from queue)

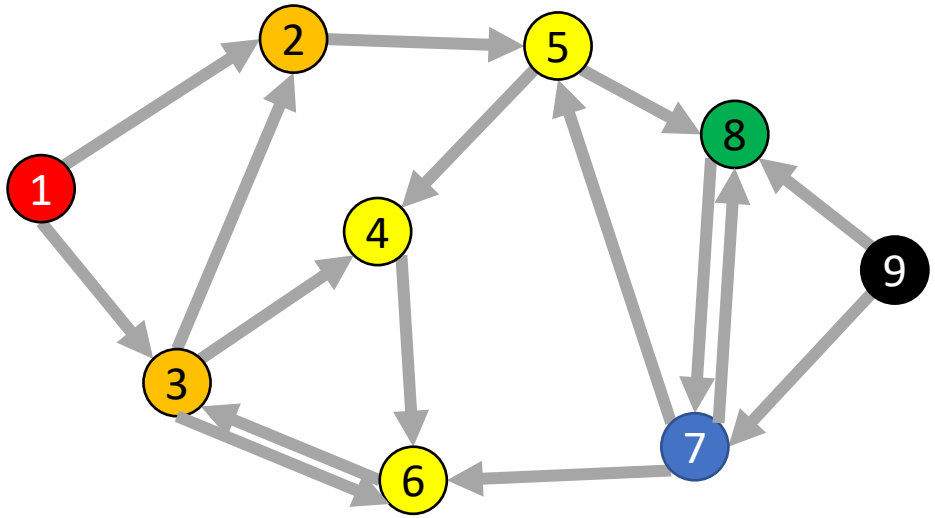
For each unvisited neighbor:

add that neighbor to a queue

mark that neighbor as visited

Queue: ~~1, 2, 3, 5,~~ 4, 6, 8

BFS(node 1) – Step 5: Dequeue 4

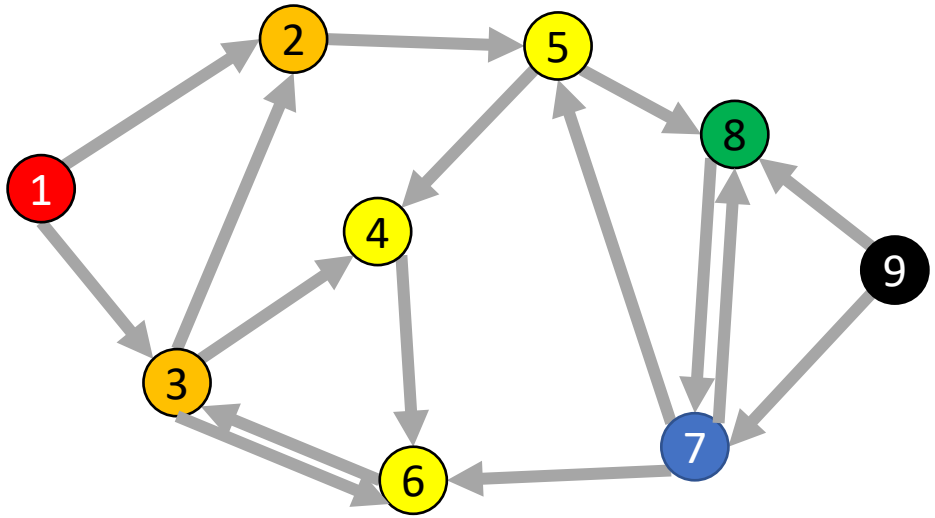


Node	Visited?	Other Info
1	True	
2	True	
3	True	
4	True	
5	True	
6	True	
7		
8	True	
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited

Queue: ~~1, 2, 3, 5, 4~~, 6, 8

BFS(node 1) – Step 6: Dequeue 6

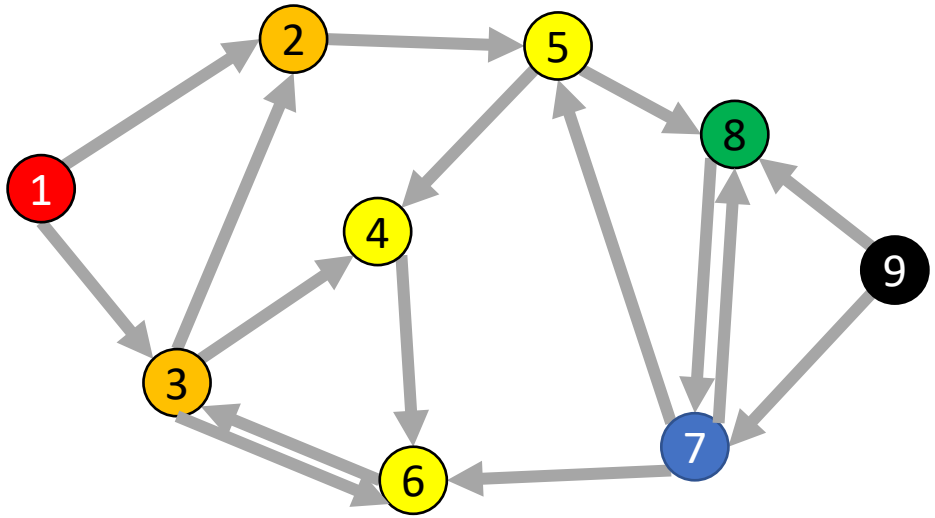


Node	Visited?	Other Info
1	True	
2	True	
3	True	
4	True	
5	True	
6	True	
7		
8	True	
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited

Queue: ~~1, 2, 3, 5, 4, 6~~, 8

BFS(node 1) – Step 7: Dequeue 8

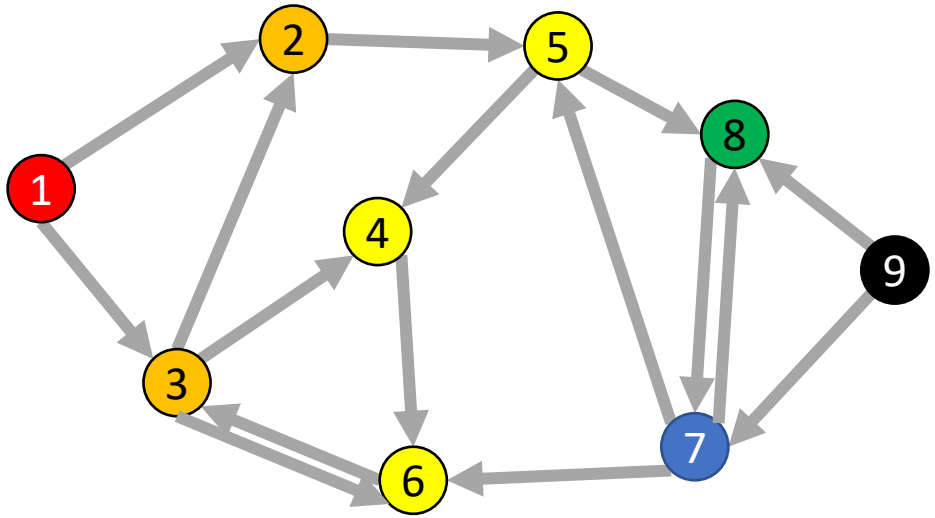


Node	Visited?	Other Info
1	True	
2	True	
3	True	
4	True	
5	True	
6	True	
7	True	
8	True	
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited

Queue: ~~1, 2, 3, 5, 4, 6, 8~~, 7

BFS(node 1) – Step 8: Dequeue 7



Node	Visited?	Other Info
1	True	
2	True	
3	True	
4	True	
5	True	
6	True	
7	True	
8	True	
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited

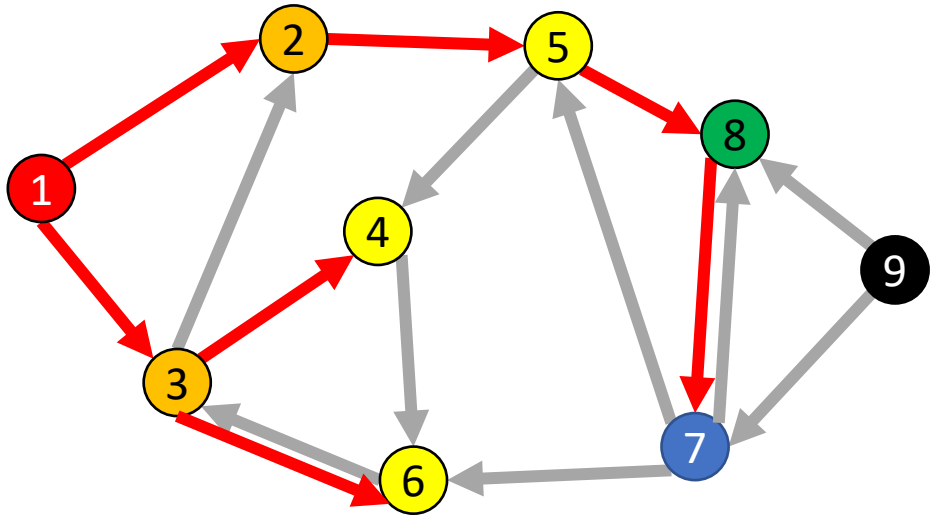
Queue: ~~1, 2, 3, 5, 4, 6, 8, 7~~

Note: 9 is not reachable from node 1

What about Finding the **Distance** from Node 1?

- Distance in terms of number of edges we have to traverse
- We will calculate this as the node's "Layer"

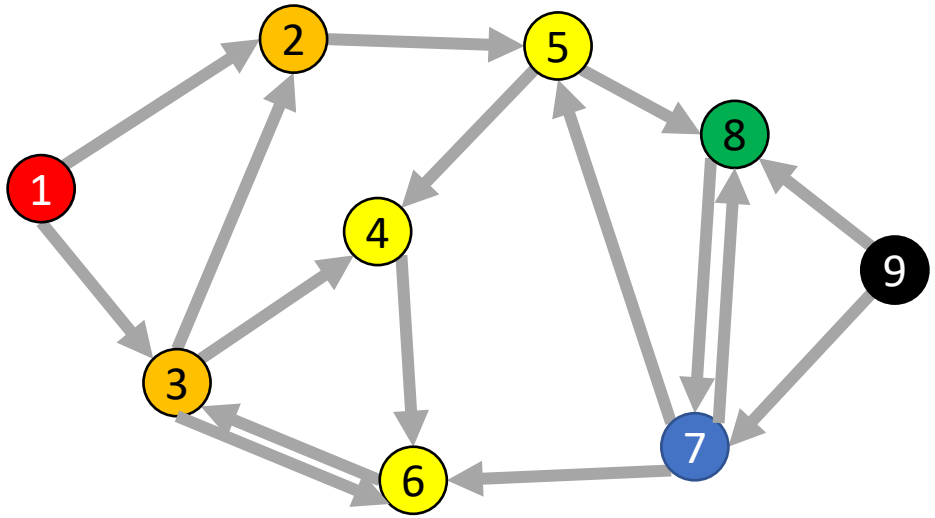
Find Distance (unweighted)



Idea: when it's seen, remember its "layer" depth!

```
int findDistance(graph, s, t) {
        found = new Queue();
        layer = 0;
        found.enqueue(s);
        mark s as "visited";
        while (!found.isEmpty()) {
                current = found.dequeue();
                layer = depth of current;
                for (v : neighbors(current)) {
                        if (! v marked "visited") {
                                mark v as "visited";
                                depth of v = layer + 1;
                                found.enqueue(v);
                        }
                }
        }
        return depth of t;
}
```

Find Distance (node 1) – Worked Example

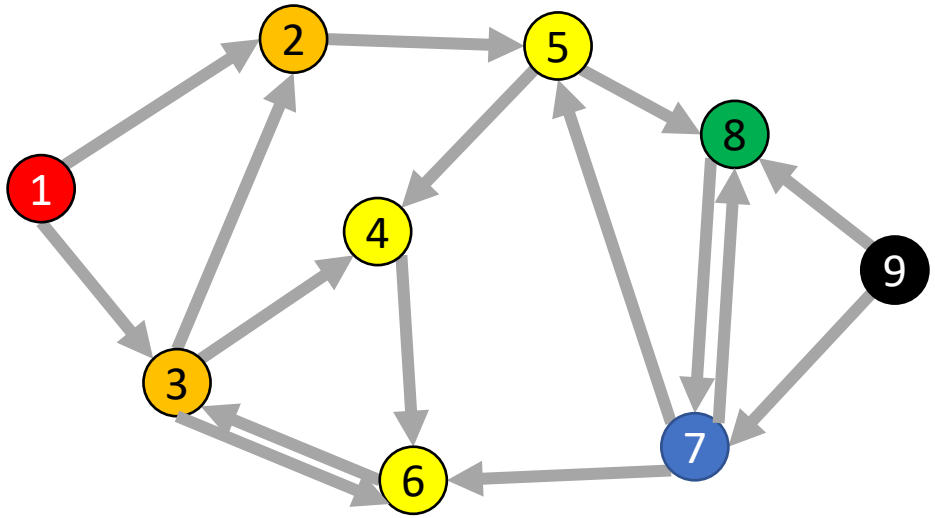


Node	Visited?	Layer
1	True	0
2		
3		
4		
5		
6		
7		
8		
9		

For each node: (dequeue from queue)
- update current layer
- For each unvisited neighbor:
  - add that neighbor to a queue
  - mark that neighbor as visited
  - set neighbor's layer to be current layer + 1

Queue: 1

Find Distance (node 1) - Step 1: Dequeue 1



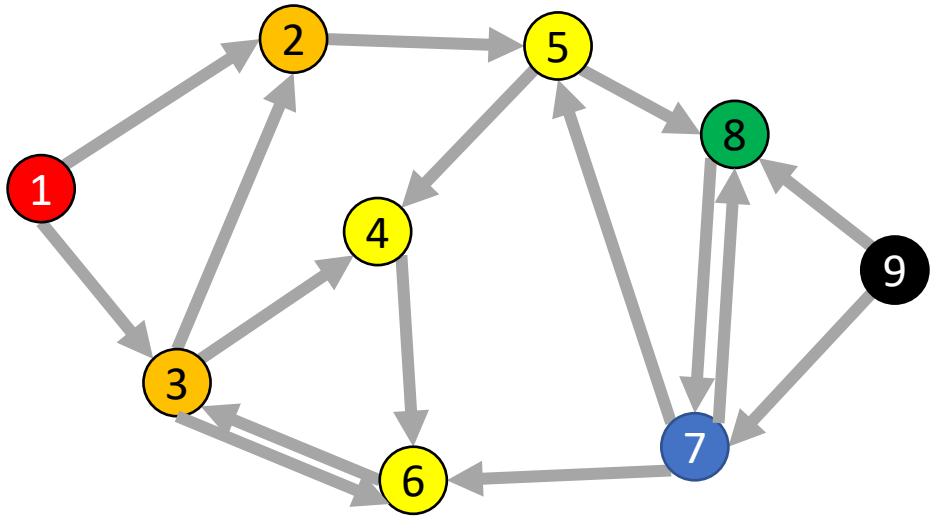
Node	Visited?	Layer
1	True	0
2	True	1
3	True	1
4		
5		
6		
7		
8		
9		

For each node: (dequeue from queue)
- update current layer
- For each unvisited neighbor:
  - add that neighbor to a queue
  - mark that neighbor as visited
  - set neighbor's layer to be current layer + 1

Queue: ~~1~~, 2, 3

Note: 2 and 3 could be enqueued in any order

Find Distance (node 1) - Step 2: Dequeue 2



Node	Visited?	Layer
1	True	0
2	True	1
3	True	1
4		
5	True	2
6		
7		
8		
9		

For each node: (dequeue from queue)

update current layer

For each unvisited neighbor:

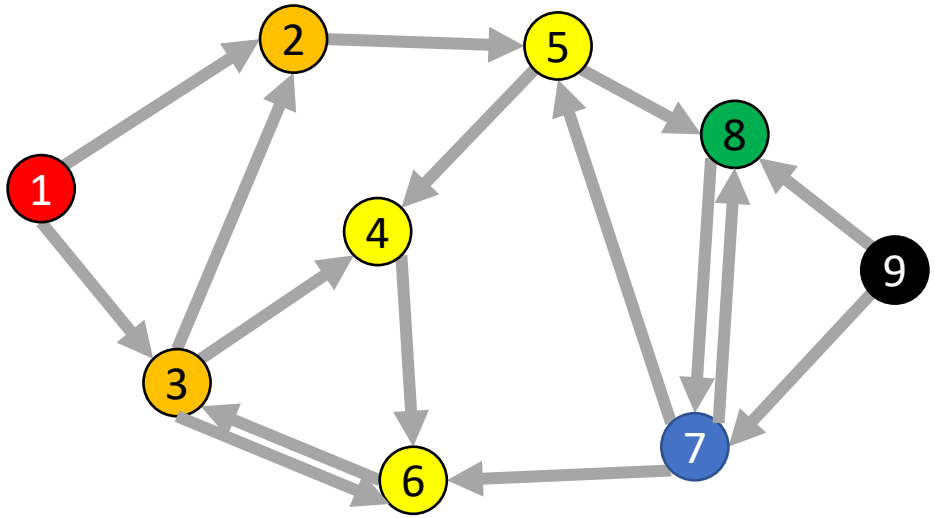
add that neighbor to a queue

mark that neighbor as visited

set neighbor's layer to be current layer + 1

Queue: ~~1, 2~~, 3, 5

Find Distance (node 1) - Step 3: Dequeue 3



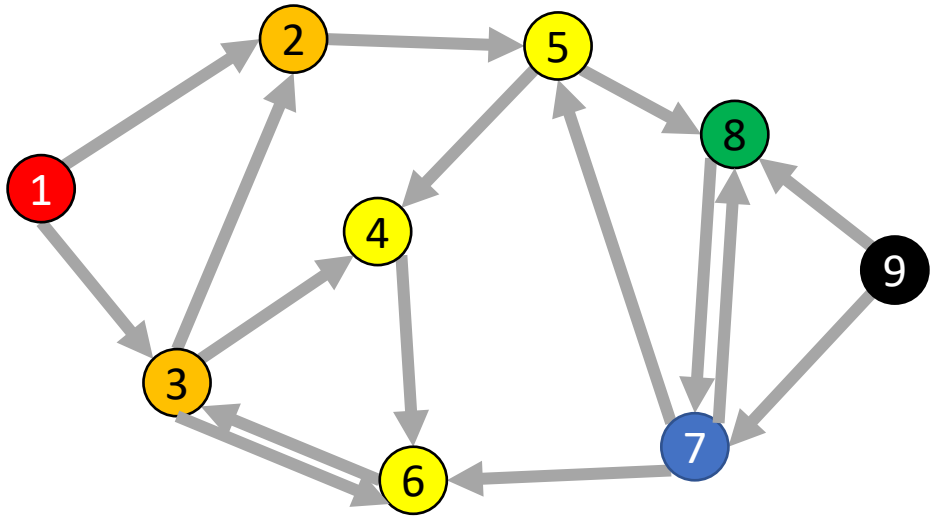
Node	Visited?	Layer
1	True	0
2	True	1
3	True	1
4	True	2
5	True	2
6	True	2
7		
8		
9		

For each node: (dequeue from queue)
- update current layer
- For each unvisited neighbor:
  - add that neighbor to a queue
  - mark that neighbor as visited
  - set neighbor's layer to be current layer + 1

Queue: ~~1, 2, 3~~, 5, 4, 6

Note: 4 and 6 could be enqueued in any order

Find Distance (node 1) - Step 4: Dequeue 5

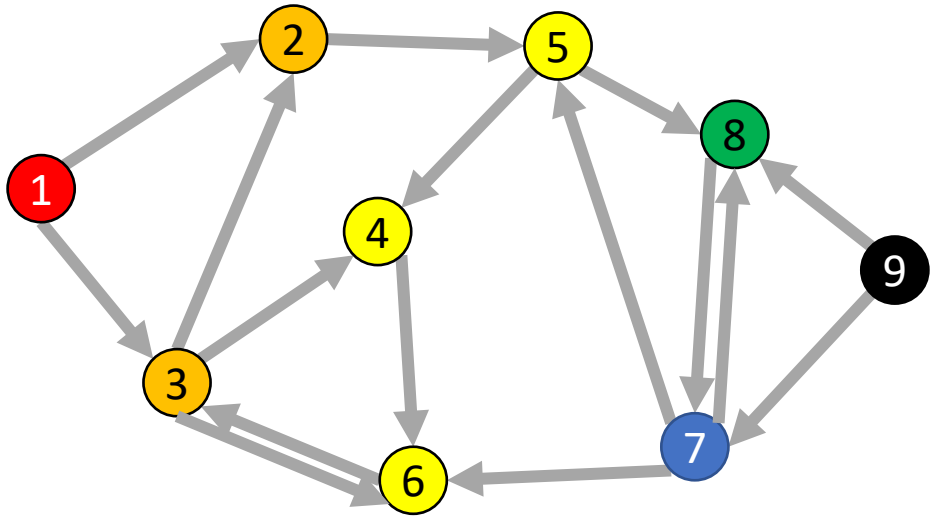


Node	Visited?	Layer
1	True	0
2	True	1
3	True	1
4	True	2
5	True	2
6	True	2
7		
8	True	3
9		

For each node: (dequeue from queue)
- update current layer
- For each unvisited neighbor:
  - add that neighbor to a queue
  - mark that neighbor as visited
  - set neighbor's layer to be current layer + 1

Queue: ~~1, 2, 3, 5~~, 4, 6, 8

Find Distance (node 1) - Step 5: Dequeue 4

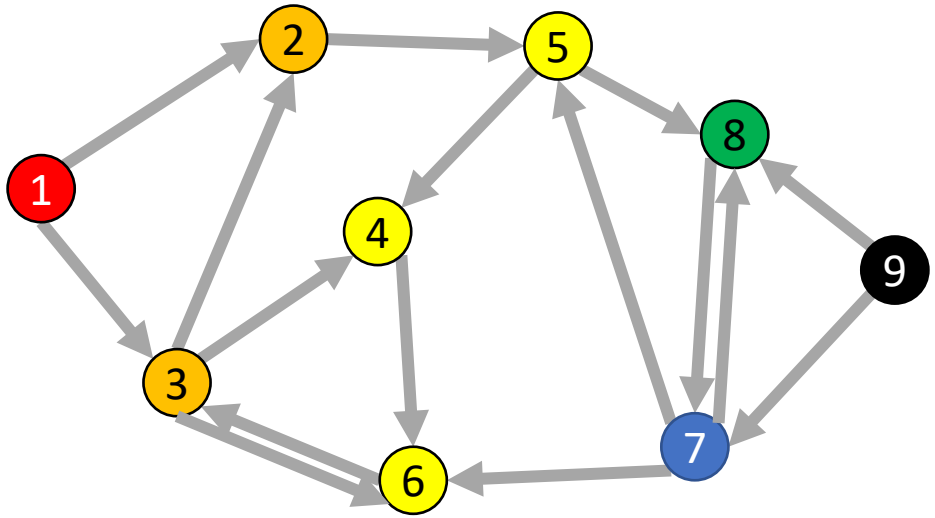


Node	Visited?	Layer
1	True	0
2	True	1
3	True	1
4	True	2
5	True	2
6	True	2
7		
8	True	3
9		

For each node: (dequeue from queue)
- update current layer
- For each unvisited neighbor:
  - add that neighbor to a queue
  - mark that neighbor as visited
  - set neighbor's layer to be current layer + 1

Queue: ~~1, 2, 3, 5, 4,~~ 6, 8

Find Distance (node 1) - Step 6: Dequeue 6

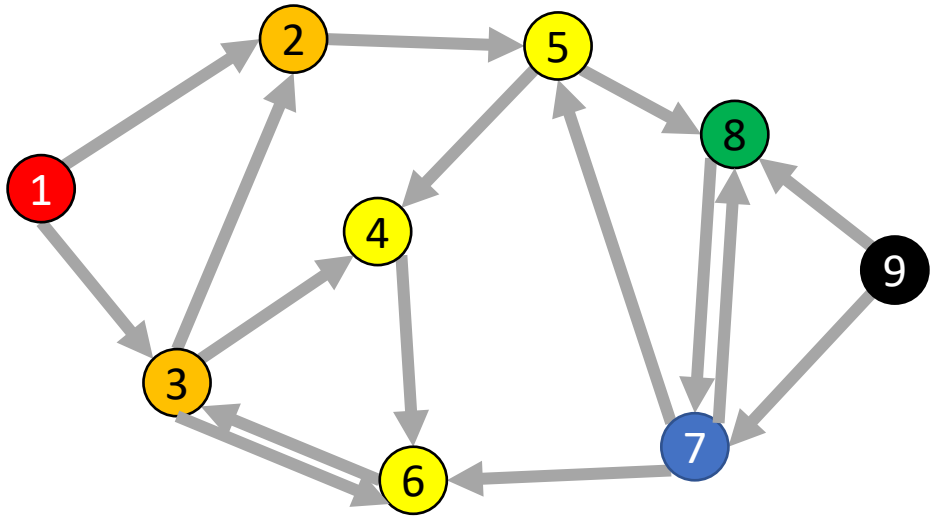


Node	Visited?	Layer
1	True	0
2	True	1
3	True	1
4	True	2
5	True	2
6	True	2
7		
8	True	3
9		

For each node: (dequeue from queue)
- update current layer
- For each unvisited neighbor:
  - add that neighbor to a queue
  - mark that neighbor as visited
  - set neighbor's layer to be current layer + 1

Queue: ~~1, 2, 3, 5, 4, 6~~, 8

Find Distance (node 1) - Step 7: Dequeue 8



Node	Visited?	Layer
1	True	0
2	True	1
3	True	1
4	True	2
5	True	2
6	True	2
7	True	4
8	True	3
9		

For each node: (dequeue from queue)

update current layer

For each unvisited neighbor:

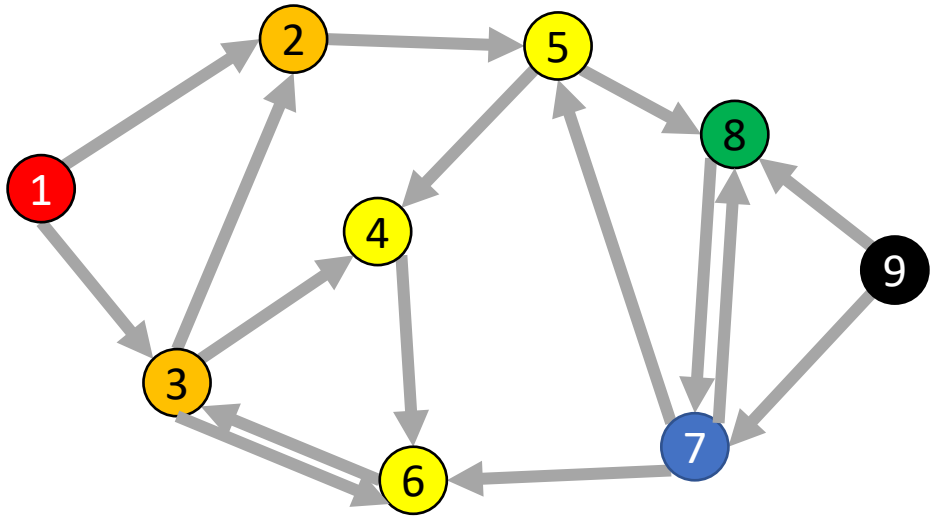
add that neighbor to a queue

mark that neighbor as visited

set neighbor's layer to be current layer + 1

Queue: ~~1, 2, 3, 5, 4, 6, 8, 7~~

Find Distance (node 1) - Step 8: Dequeue 7



Node	Visited?	Layer
1	True	0
2	True	1
3	True	1
4	True	2
5	True	2
6	True	2
7	True	4
8	True	3
9		

For each node: (dequeue from queue)
- update current layer
- For each unvisited neighbor:
  - add that neighbor to a queue
  - mark that neighbor as visited
  - set neighbor's layer to be current layer + 1

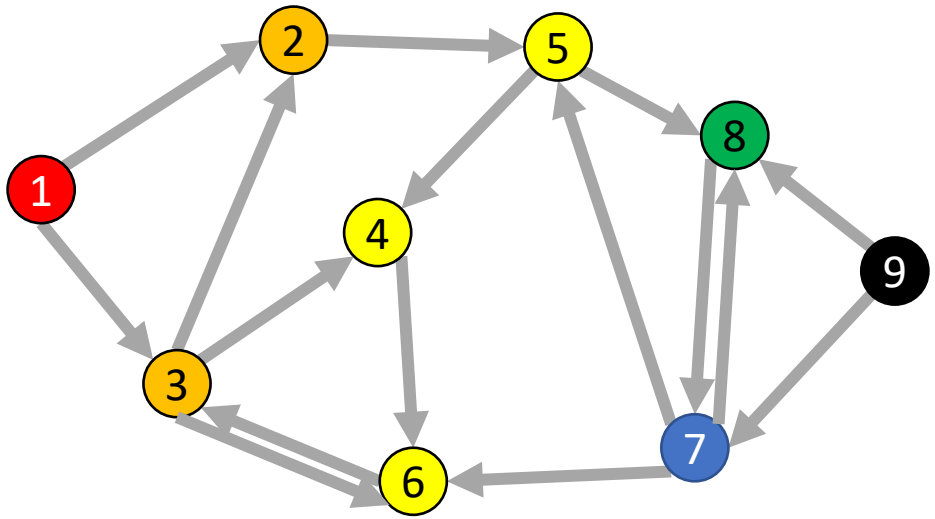
Queue: ~~1, 2, 3, 5, 4, 6, 8, 7~~

Note: 9 is not reachable from node 1

What about Finding the **Path** from Node 1?

- We want to know the sequence of nodes we would traverse to get from Node 1 to another node of interest (using the shortest distance)
- We will calculate this by recording the node that is causing us to enqueue this node as its “Previous Node”

Shortest Path (node 1) – Worked Example



Node	Visited?	Previous Node
1	True	----
2		
3		
4		
5		
6		
7		
8		
9		

For each node: (dequeue from queue)

For each unvisited neighbor:

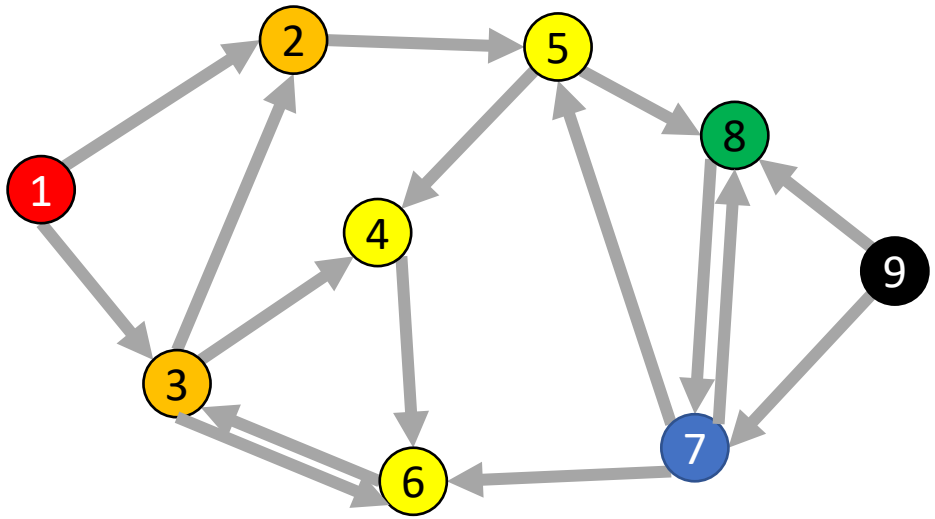
add that neighbor to a queue

mark that neighbor as visited

set neighbor's previous to be the current node

Queue: 1

Shortest Path (node 1) - Step 1: Dequeue 1



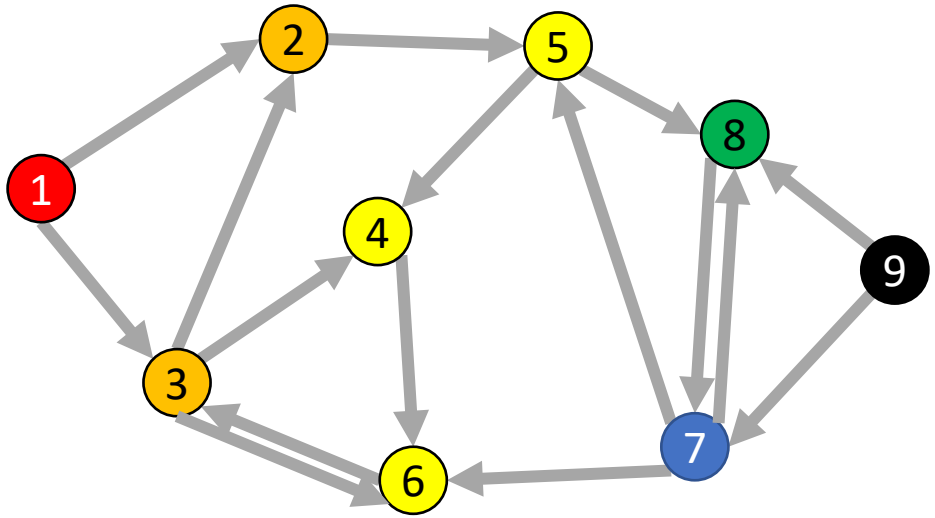
Node	Visited?	Previous Node
1	True	----
2	True	Node 1
3	True	Node 1
4		
5		
6		
7		
8		
9		

For each node: (dequeue from queue)
For each unvisited neighbor:
add that neighbor to a queue
mark that neighbor as visited
set neighbor's previous to be the current node

Queue: ~~1~~, 2, 3

Note: 2 and 3 could be enqueued in any order

Shortest Path (node 1) - Step 2: Dequeue 2

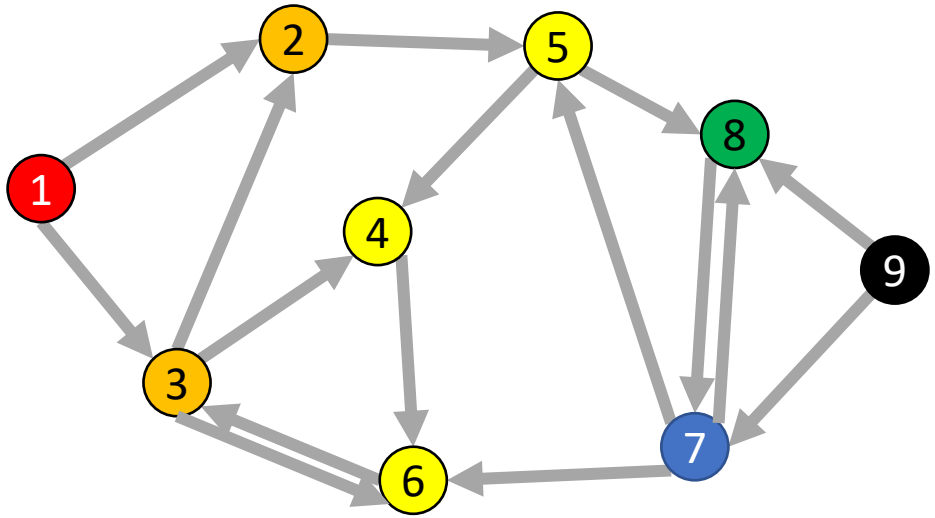


Node	Visited?	Previous Node
1	True	----
2	True	Node 1
3	True	Node 1
4		
5	True	Node 2
6		
7		
8		
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited
    set neighbor's previous to be the current node

Queue: ~~1, 2~~, 3, 5

Shortest Path (node 1) - Step 3: Dequeue 3



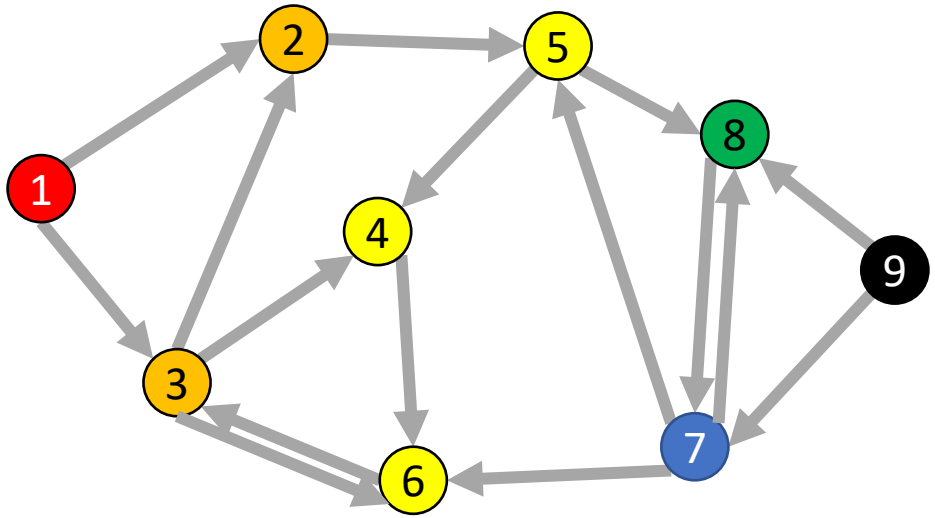
Node	Visited?	Previous Node
1	True	----
2	True	Node 1
3	True	Node 1
4	True	Node 3
5	True	Node 2
6	True	Node 3
7		
8		
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited
    set neighbor's previous to be the current node

Queue: ~~1, 2, 3~~, 5, 4, 6

Note: 4 and 6 could be enqueued in any order

Shortest Path (node 1) - Step 4: Dequeue 5



Node	Visited?	Previous Node
1	True	----
2	True	Node 1
3	True	Node 1
4	True	Node 3
5	True	Node 2
6	True	Node 3
7		
8	True	Node 5
9		

For each node: (dequeue from queue)

For each unvisited neighbor:

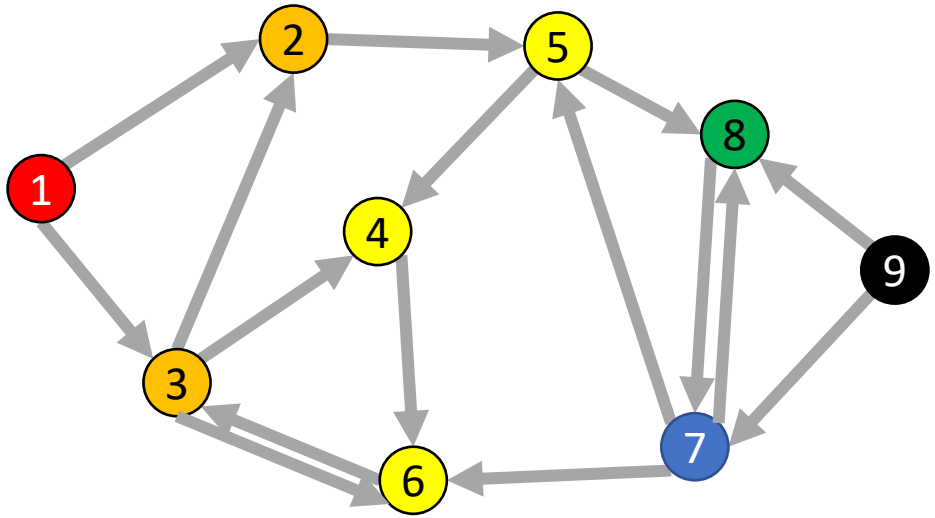
add that neighbor to a queue

mark that neighbor as visited

set neighbor's previous to be the current node

Queue: ~~1, 2, 3, 5~~, 4, 6, 8

Shortest Path (node 1) - Step 5: Dequeue 4

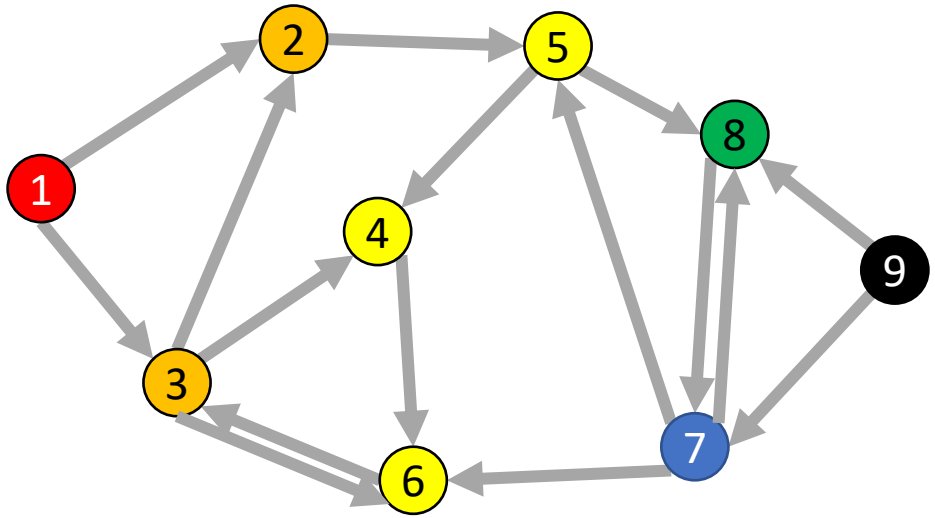


Node	Visited?	Previous Node
1	True	----
2	True	Node 1
3	True	Node 1
4	True	Node 3
5	True	Node 2
6	True	Node 3
7		
8	True	Node 5
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited
    set neighbor's previous to be the current node

Queue: ~~1, 2, 3, 5, 4,~~ 6, 8

Shortest Path (node 1) - Step 6: Dequeue 6

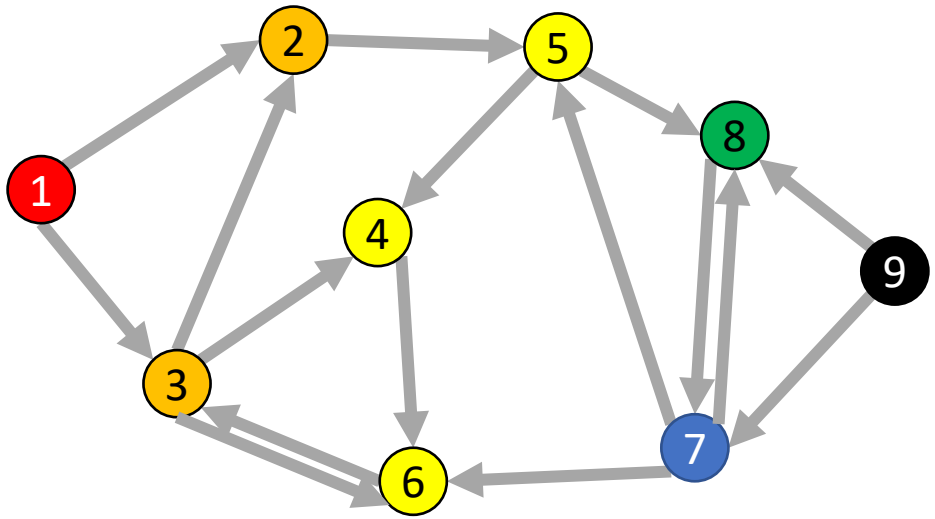


Node	Visited?	Previous Node
1	True	----
2	True	Node 1
3	True	Node 1
4	True	Node 3
5	True	Node 2
6	True	Node 3
7		
8	True	Node 5
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited
    set neighbor's previous to be the current node

Queue: ~~1, 2, 3, 5, 4, 6~~, 8

Shortest Path (node 1) - Step 7: Dequeue 8

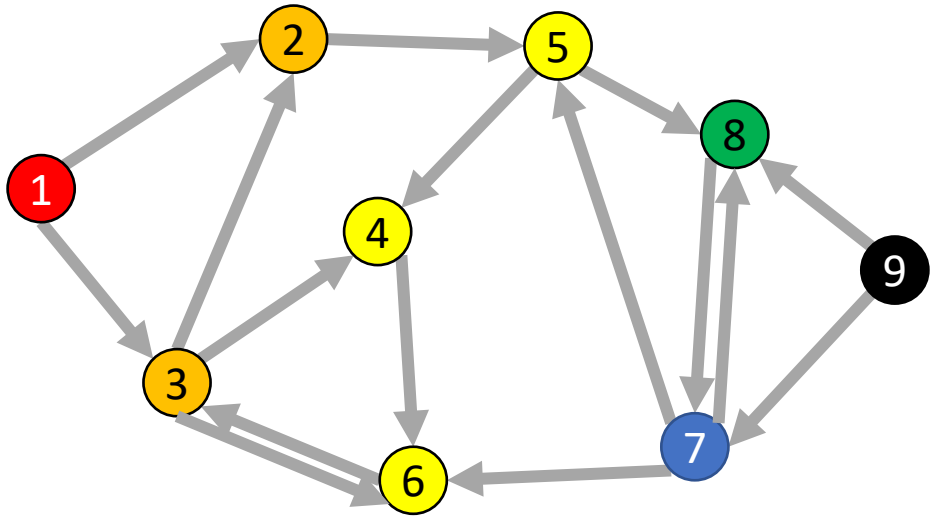


Node	Visited?	Previous Node
1	True	----
2	True	Node 1
3	True	Node 1
4	True	Node 3
5	True	Node 2
6	True	Node 3
7	True	Node 8
8	True	Node 5
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited
    set neighbor's previous to be the current node

Queue: ~~1, 2, 3, 5, 4, 6, 8,~~ 7

Shortest Path (node 1) - Step 8: Dequeue 7



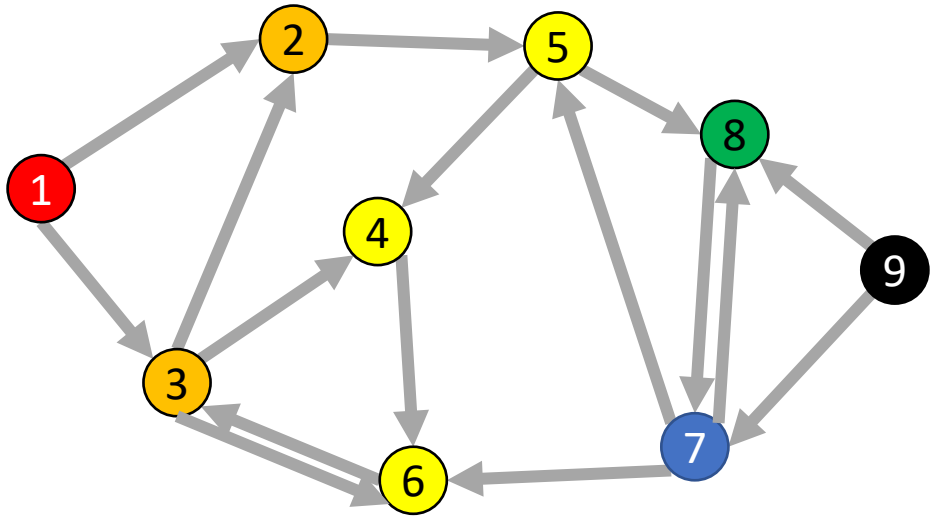
Node	Visited?	Previous Node
1	True	----
2	True	Node 1
3	True	Node 1
4	True	Node 3
5	True	Node 2
6	True	Node 3
7	True	Node 8
8	True	Node 5
9		

For each node: (dequeue from queue)
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited
    set neighbor's previous to be the current node

Queue: ~~1, 2, 3, 5, 4, 6, 8, 7~~

Note: 9 is not reachable from node 1

Shortest Path from node 1 to Node 7?



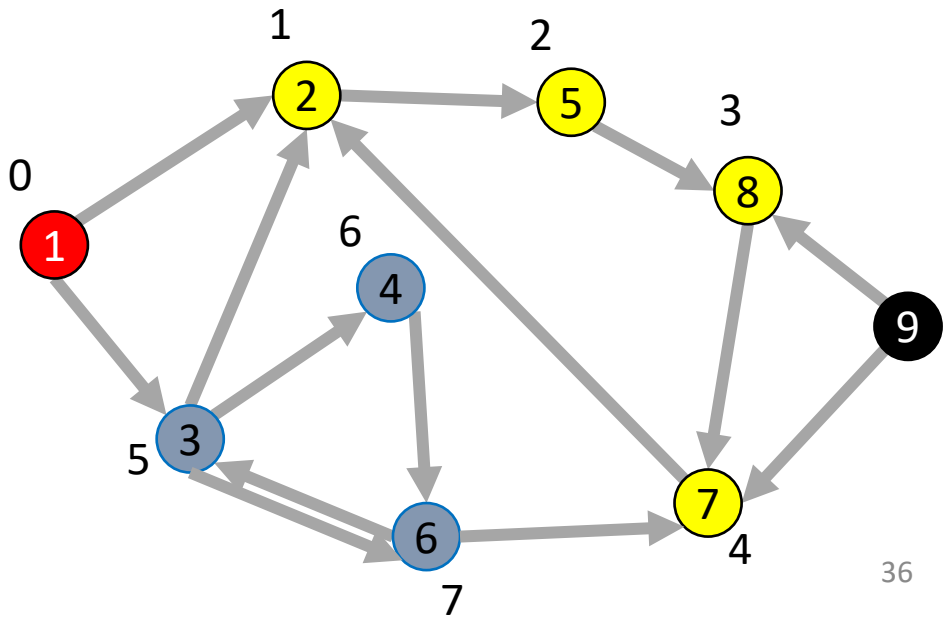
Node	Visited?	Previous Node
1	True	----
2	True	Node 1
3	True	Node 1
4	True	Node 3
5	True	Node 2
6	True	Node 3
7	True	Node 8
8	True	Node 5
9		

Start from destination node and work backwards:

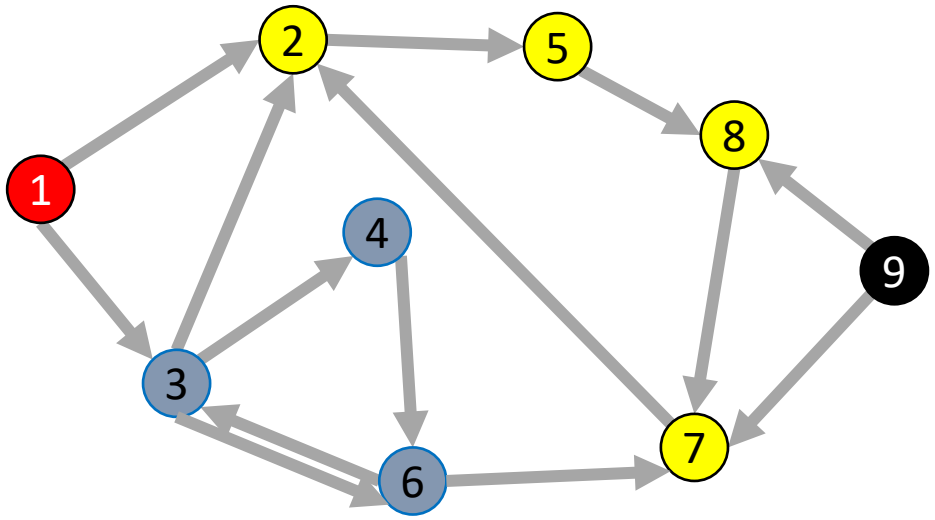
1, 2, 5, 8, 7

Depth-First Search

- Input: a node s
- Behavior: Start with node s, visit one neighbor of s, then all nodes *reachable* from that neighbor of s, then another neighbor of s,…
  - Before moving on to the second neighbor of s, visit everything *reachable* from the first neighbor of s
- Output:
  - Does the graph have a cycle?
  - A **topological sort** of the graph.



DFS (non-recursive)

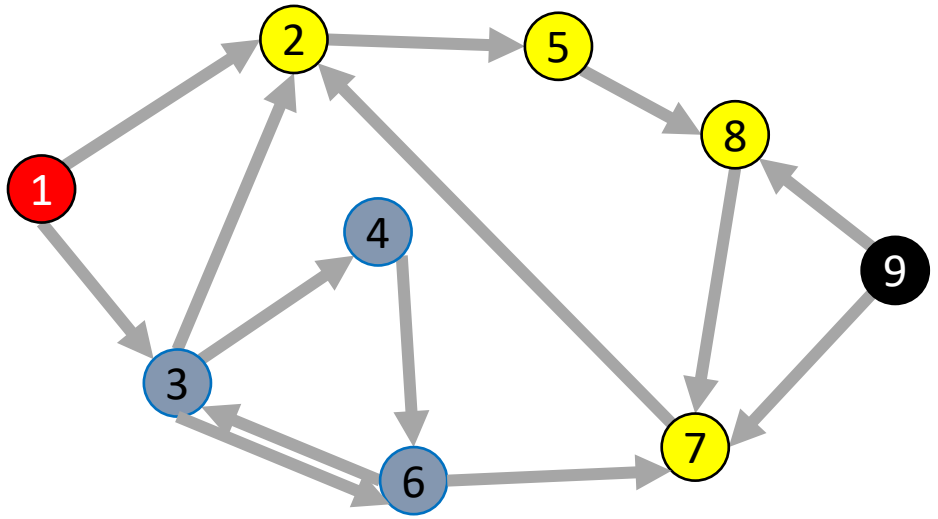


Running time: $\Theta(|V| + |E|)$

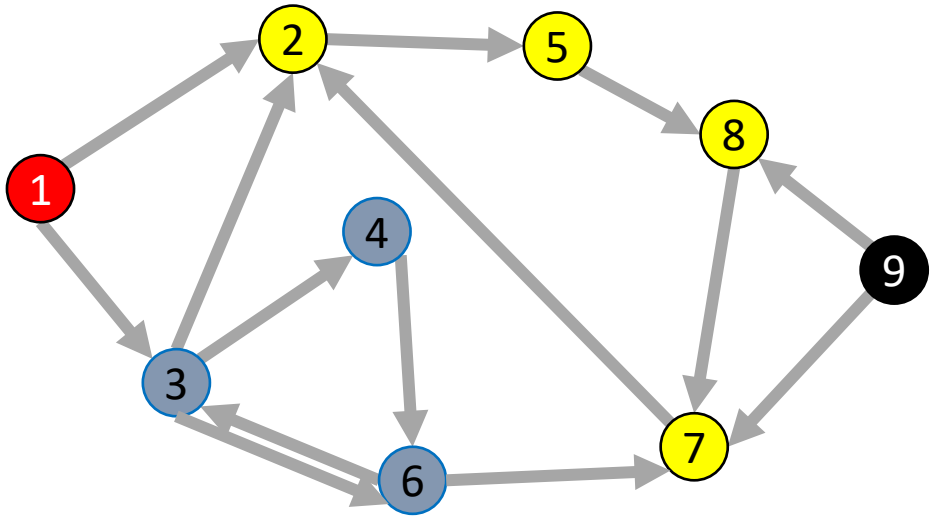
```
void dfs(graph, s) {
        found = new Stack();
        found.pop(s);
        mark s as “visited”;
        while (!found.isEmpty()) {
                current = found.pop();
                for (v : neighbors(current)) {
                        if (! v marked “visited”) {
                                mark v as “visited”;
                                found.push(v);
                        }
                }
        }
}
```

DFS Recursively (more common)

```
void dfs(graph, curr) {
        mark curr as “visited”;
        for (v : neighbors(current)) {
                if (! v marked “visited”) {
                        dfs(graph, v);
                }
        }
        mark curr as “done”;
}
```



DFS (node 1) – Worked Example

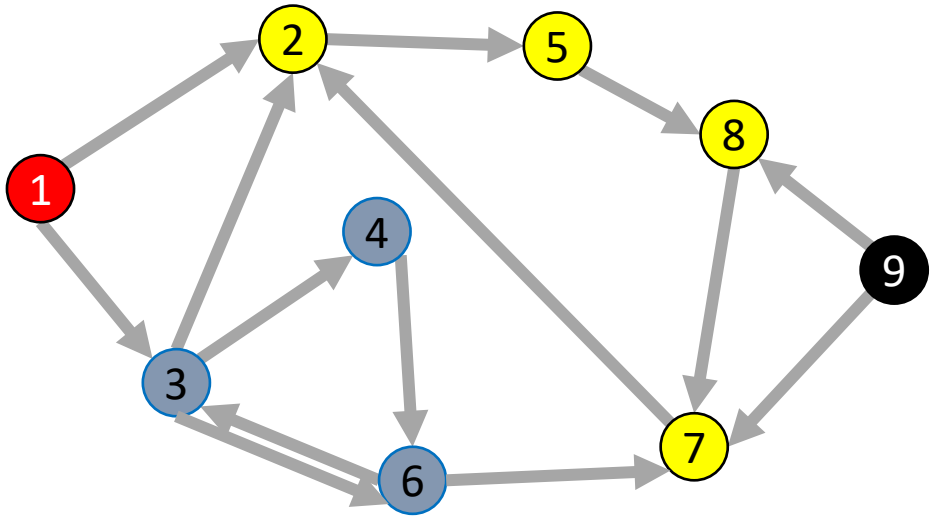


Node	Visited?	Done?	Other Info
1			
2			
3			
4			
5			
6			
7			
8			
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack:

DFS (node 1) – Step 0: Call DFS on 1

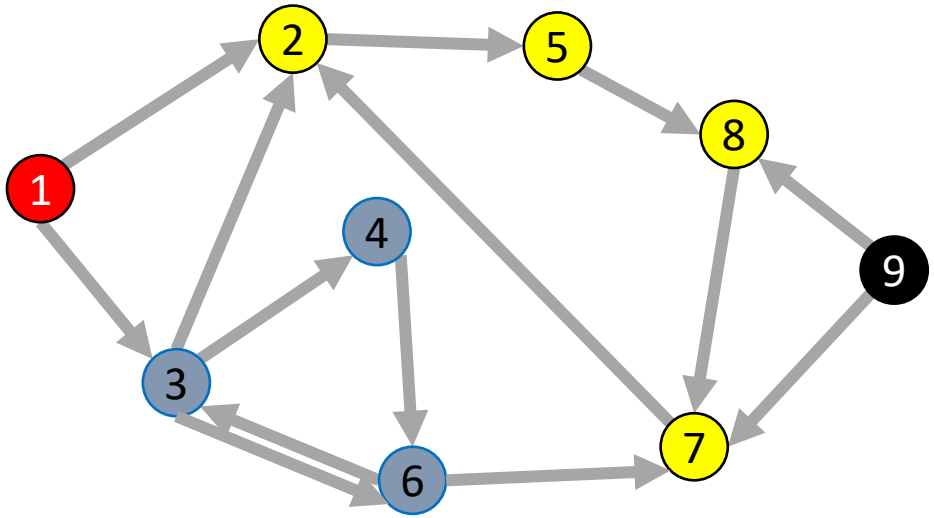


Node	Visited?	Done?	Other Info
1	True		
2			
3			
4			
5			
6			
7			
8			
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1

DFS (node 1) – Step 1: Call DFS on 2

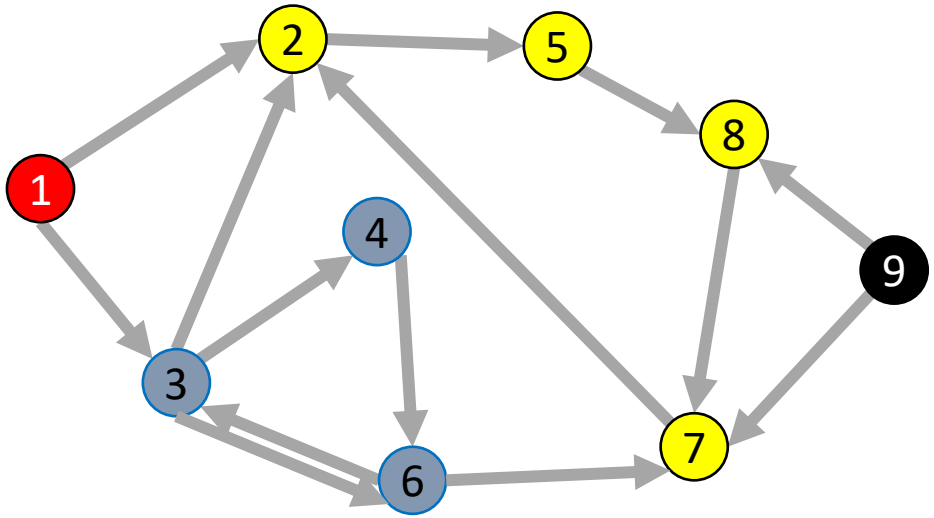


Node	Visited?	Done?	Other Info
1	True		
2	True		
3			
4			
5			
6			
7			
8			
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, 2

DFS (node 1) – Step 2: Call DFS on 5

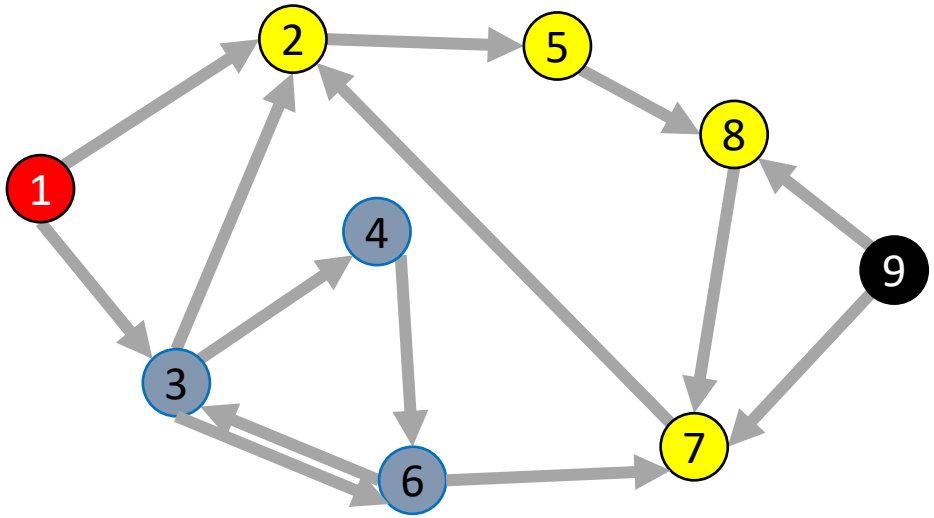


Node	Visited?	Done?	Other Info
1	True		
2	True		
3			
4			
5	True		
6			
7			
8			
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, 2, 5

DFS (node 1) – Step 3: Call DFS on 8

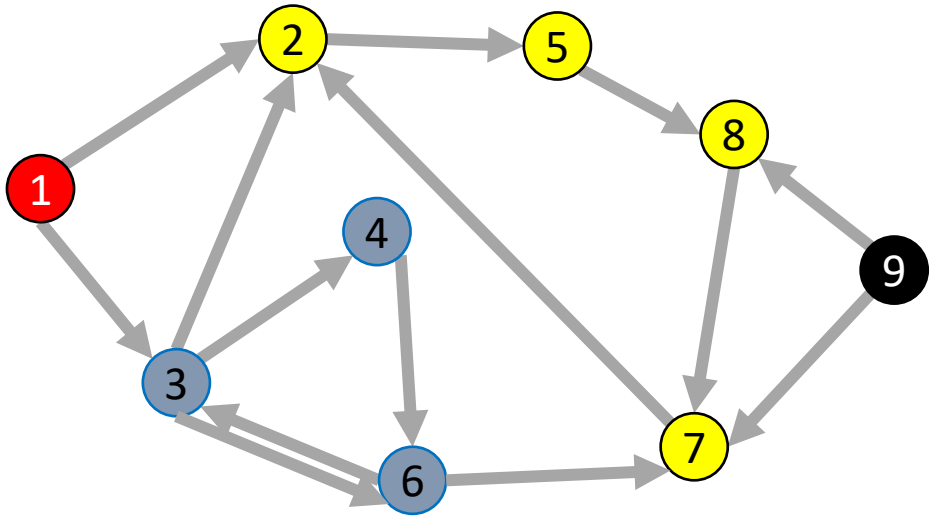


Node	Visited?	Done?	Other Info
1	True		
2	True		
3			
4			
5	True		
6			
7			
8	True		
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, 2, 5, 8

DFS (node 1) – Step 4: Call DFS on 7

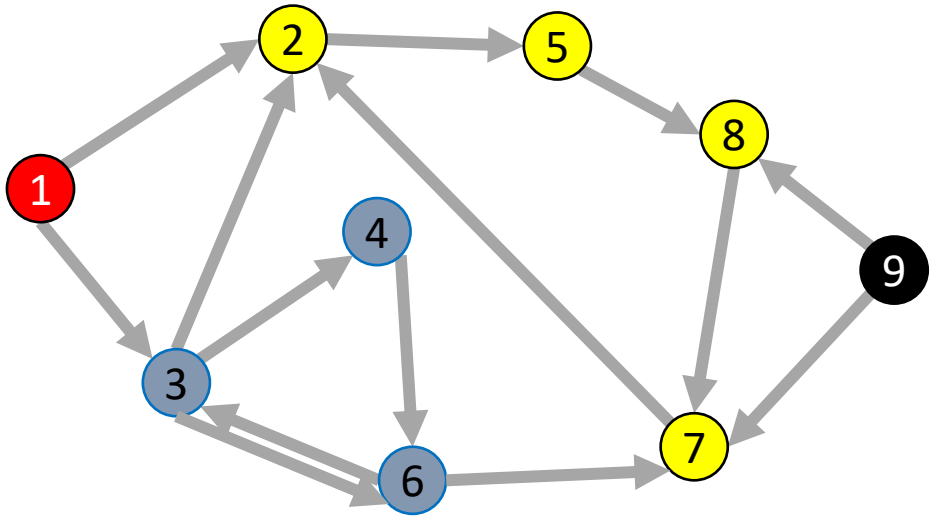


Node	Visited?	Done?	Other Info
1	True		
2	True		
3			
4			
5	True		
6			
7	True		
8	True		
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, 2, 5, 8, 7

DFS (node 1) – Step 5: Return from DFS on 7

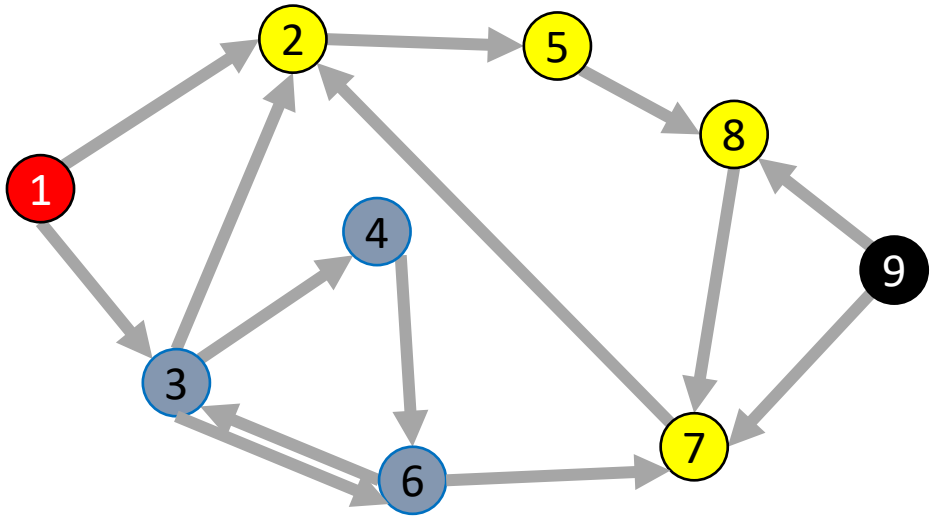


Node	Visited?	Done?	Other Info
1	True		
2	True		
3			
4			
5	True		
6			
7	True	True	
8	True		
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, 2, 5, 8, ~~7~~

DFS (node 1) – Step 6: Return from DFS on 8

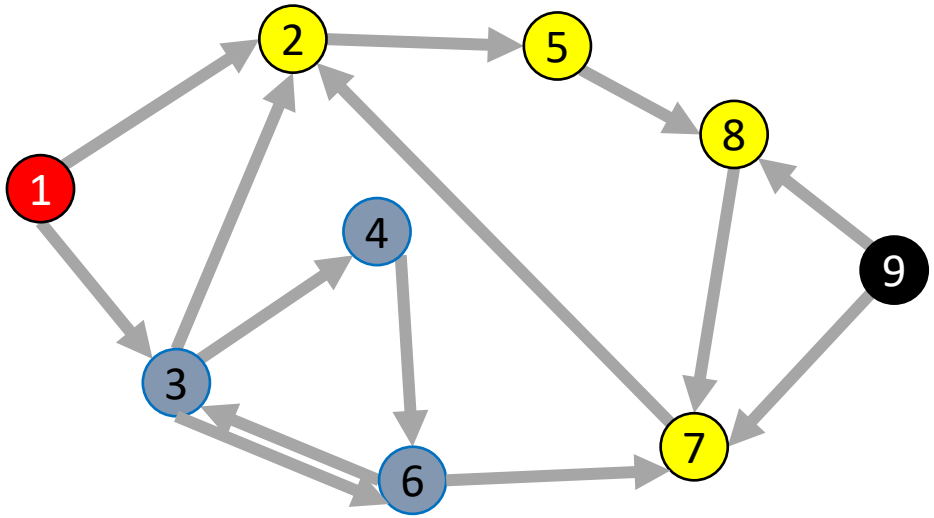


Node	Visited?	Done?	Other Info
1	True		
2	True		
3			
4			
5	True		
6			
7	True	True	
8	True	True	
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, 2, 5, ~~8, 7~~

DFS (node 1) – Step 7: Return from DFS on 5

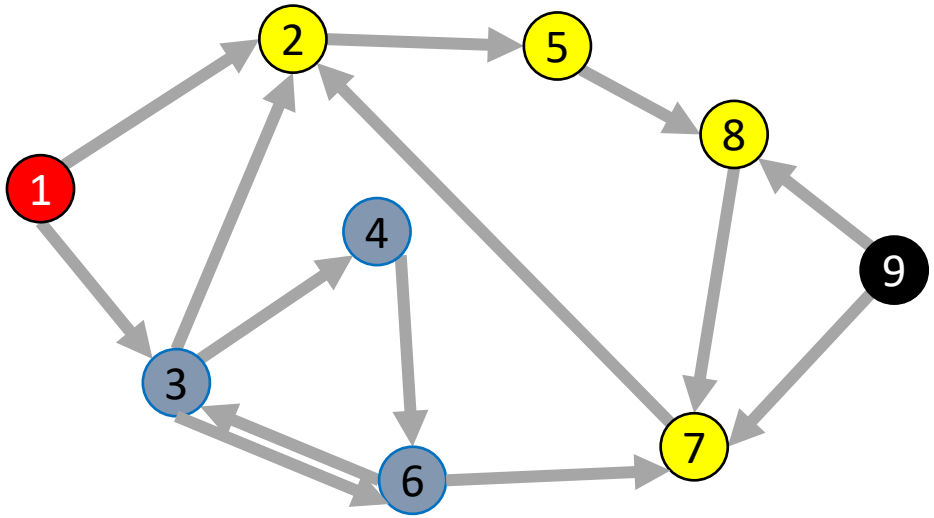


Node	Visited?	Done?	Other Info
1	True		
2	True		
3			
4			
5	True	True	
6			
7	True	True	
8	True	True	
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, 2, ~~5, 8, 7~~

DFS (node 1) – Step 8: Return from DFS on 2

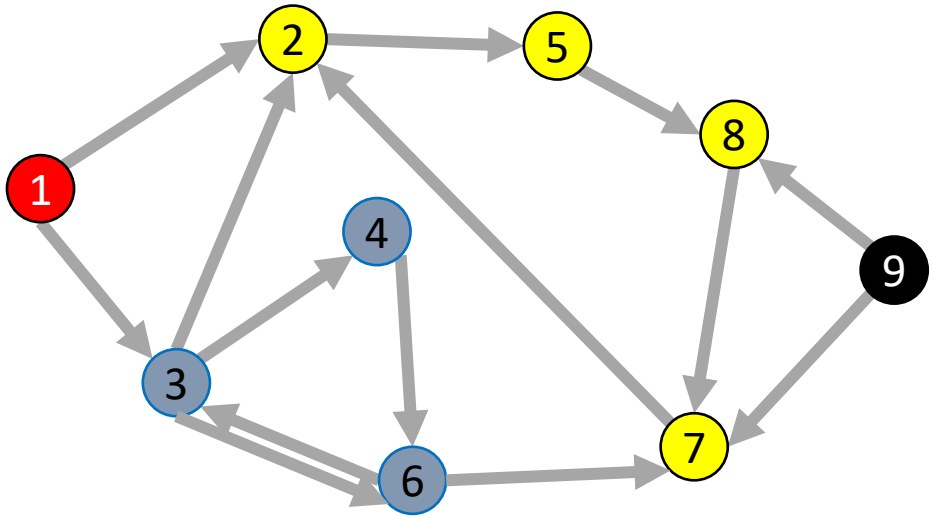


Node	Visited?	Done?	Other Info
1	True		
2	True	True	
3			
4			
5	True	True	
6			
7	True	True	
8	True	True	
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, ~~2, 5, 8, 7~~

DFS (node 1) – Step 9: Call DFS on 3

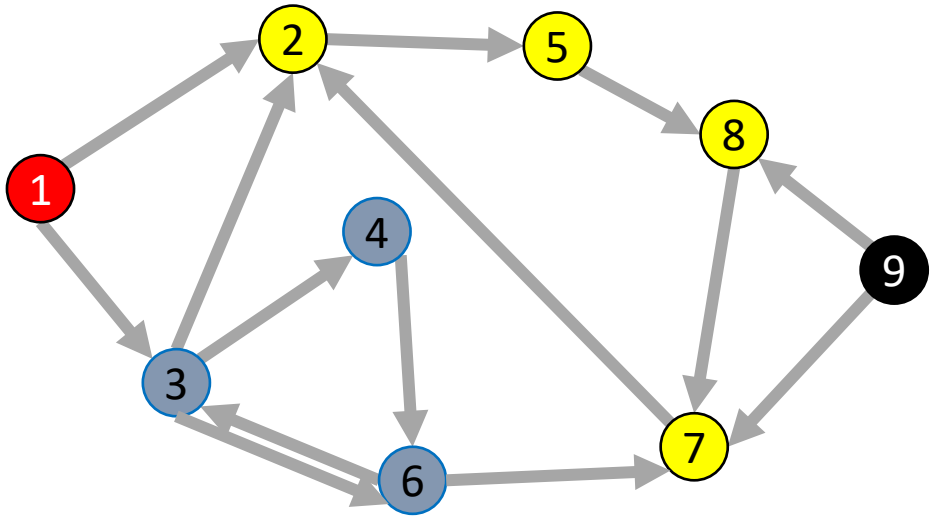


Node	Visited?	Done?	Other Info
1	True		
2	True	True	
3	True		
4			
5	True	True	
6			
7	True	True	
8	True	True	
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, ~~2, 5, 8, 7~~, 3

DFS (node 1) – Step 10: Call DFS on 4

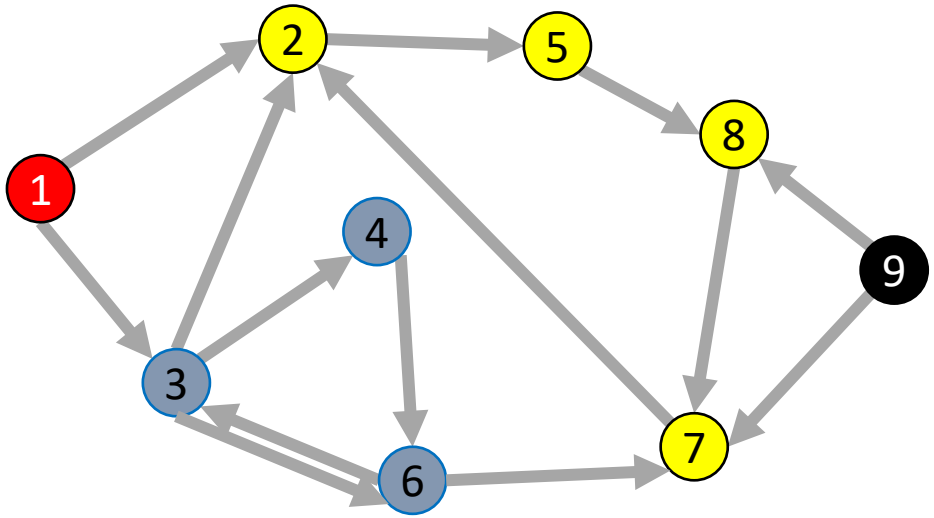


Node	Visited?	Done?	Other Info
1	True		
2	True	True	
3	True		
4	True		
5	True	True	
6			
7	True	True	
8	True	True	
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, ~~2, 5, 8, 7~~, 3, 4

DFS (node 1) – Step 11: Call DFS on 6

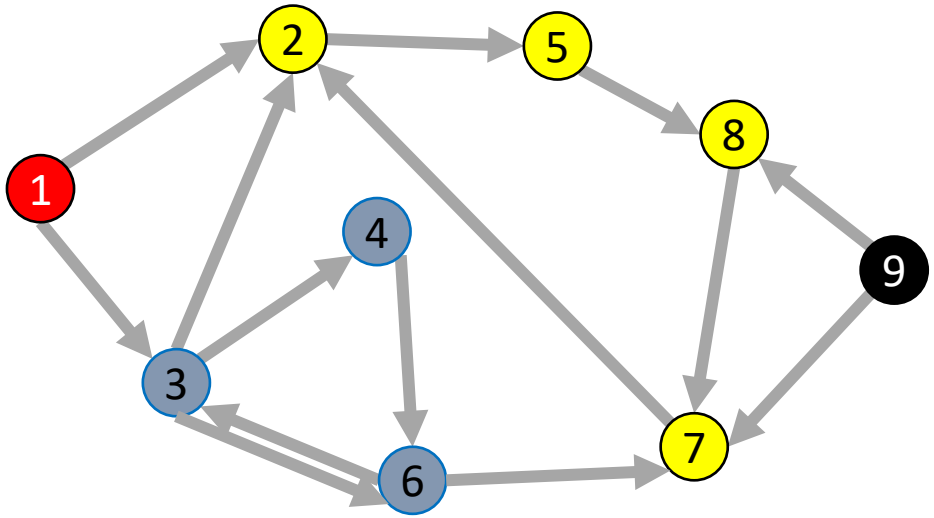


Node	Visited?	Done?	Other Info
1	True		
2	True	True	
3	True		
4	True		
5	True	True	
6	True		
7	True	True	
8	True	True	
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, ~~2, 5, 8, 7~~, 3, 4, 6

DFS (node 1) – Step 12: Return from DFS on 6

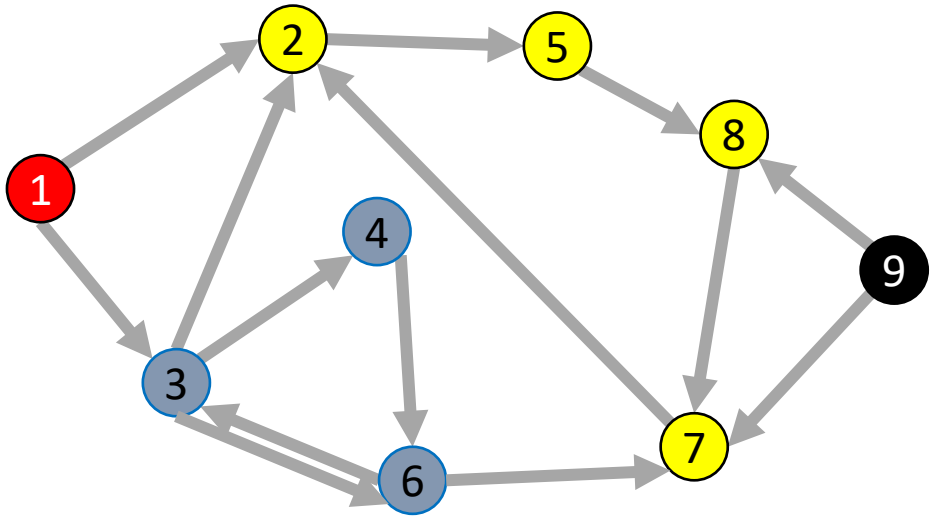


Node	Visited?	Done?	Other Info
1	True		
2	True	True	
3	True		
4	True		
5	True	True	
6	True	True	
7	True	True	
8	True	True	
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, ~~2, 5, 8, 7~~, 3, 4, ~~6~~

DFS (node 1) – Step 13: Return from DFS on 4

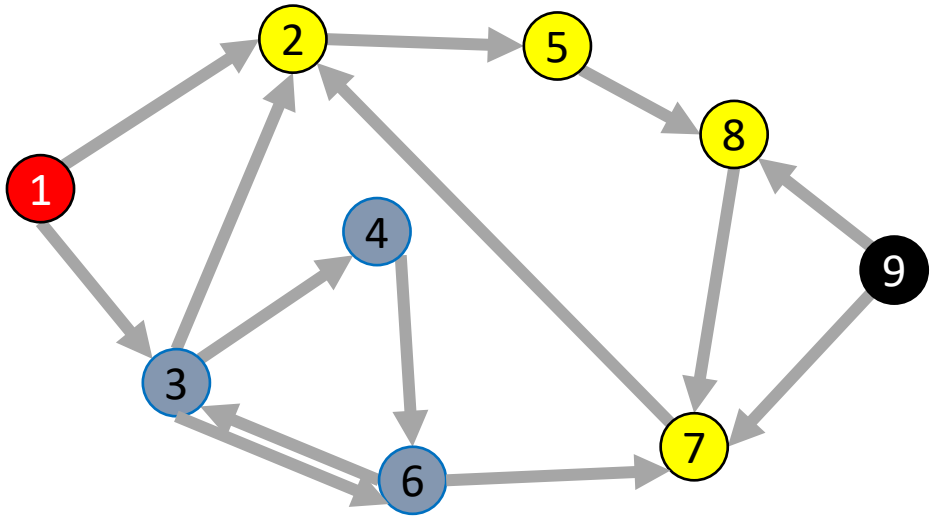


Node	Visited?	Done?	Other Info
1	True		
2	True	True	
3	True		
4	True	True	
5	True	True	
6	True	True	
7	True	True	
8	True	True	
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, ~~2, 5, 8, 7~~, 3, ~~4, 6~~

DFS (node 1) – Step 14: Return from DFS on 3

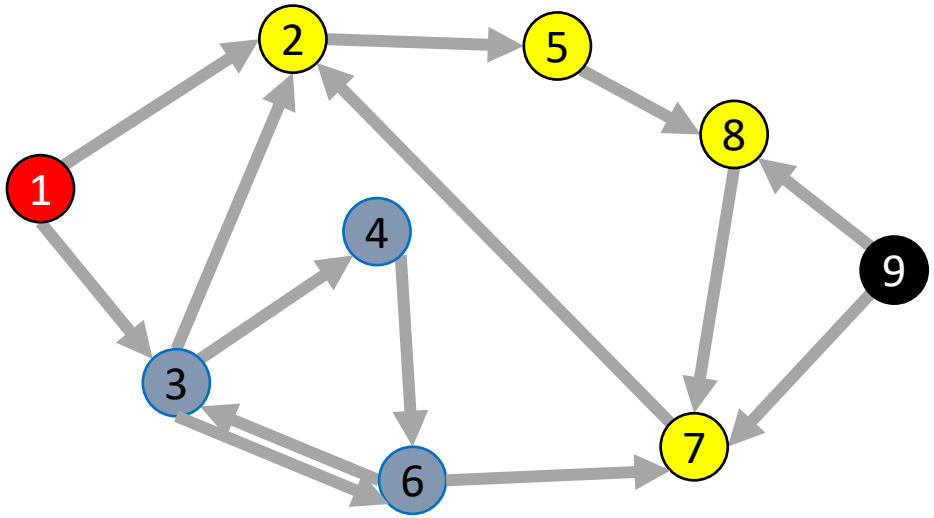


Node	Visited?	Done?	Other Info
1	True		
2	True	True	
3	True	True	
4	True	True	
5	True	True	
6	True	True	
7	True	True	
8	True	True	
9			

Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: 1, ~~2, 5, 8, 7, 3, 4, 6~~

DFS (node 1) – Step 15: Return from DFS on 1



Node	Visited?	Done?	Other Info
1	True	True	
2	True	True	
3	True	True	
4	True	True	
5	True	True	
6	True	True	
7	True	True	
8	True	True	
9			

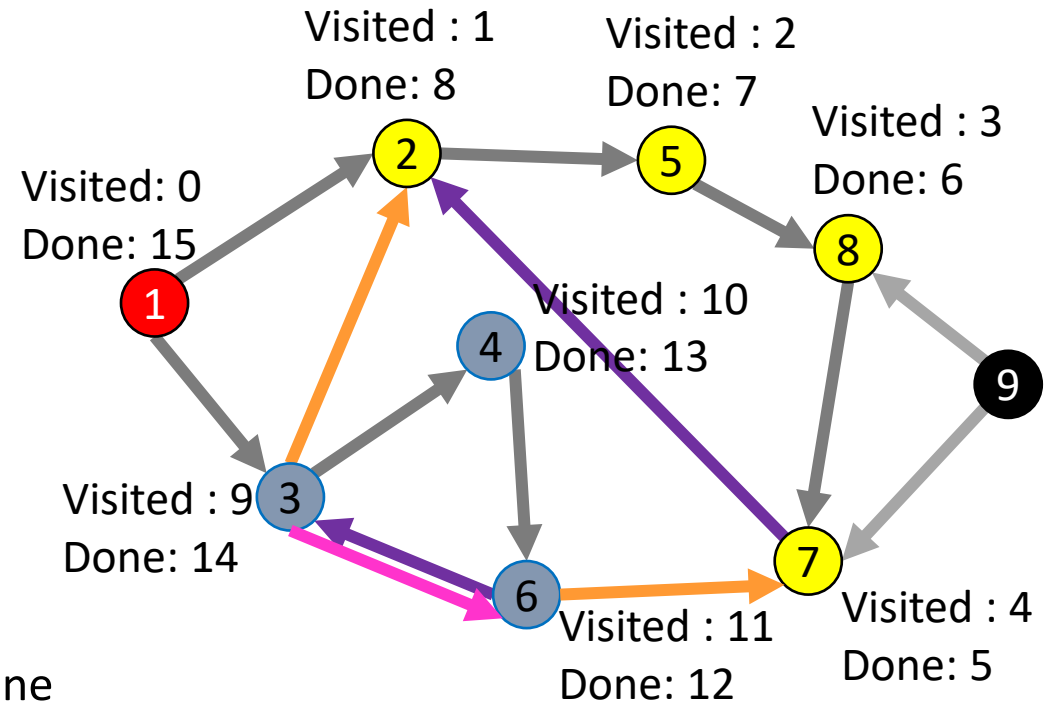
Starting from the current node:
for each unvisited neighbor:
mark the neighbor as visited
do a DFS from the neighbor
mark the current node as done

(Call) Stack: ~~1, 2, 5, 8, 7, 3, 4, 6~~

Note: 9 is not reachable from node 1

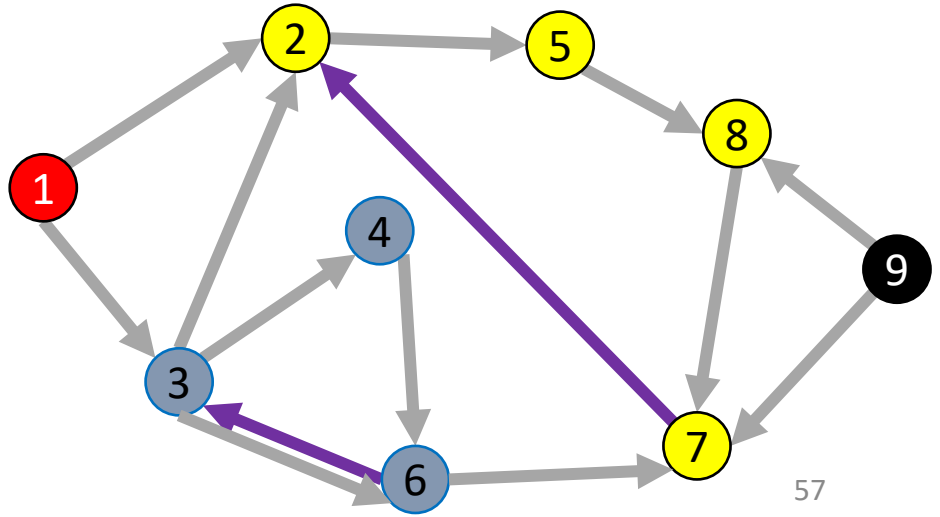
Using DFS

- Consider the "visited times" and "done times"
- Edges can be categorized:
  - Tree Edge
    - (a, b) was followed when pushing
    - (a, b) when b was unvisited when we were at a
  - Back Edge
    - (a, b) goes to an "ancestor"
    - a and b visited but not done when we saw (a, b)
    - $t_{visited}(b) < t_{visited}(a) < t_{done}(a) < t_{done}(b)$
  - Forward Edge
    - (a, b) goes to a "descendent"
    - b was visited and done between when a was visited and done
    - $t_{visited}(a) < t_{visited}(b) < t_{done}(b) < t_{done}(a)$
  - Cross Edge
    - (a, b) goes to a node that doesn't connect to a
    - b was seen and done before a was ever visited
    - $t_{done}(b) < t_{visited}(a)$



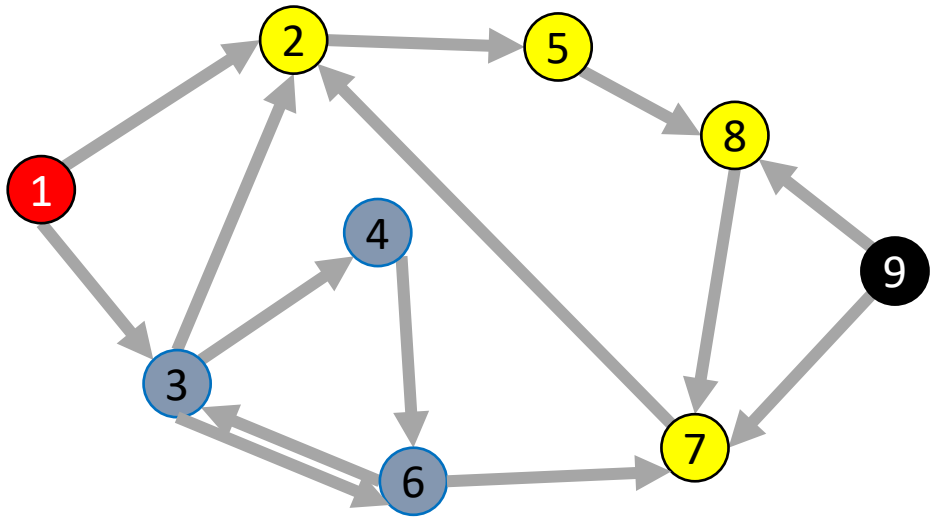
Back Edges

- Behavior of DFS:
  - “Visit everything reachable from the current node before going back”
- Back Edge:
  - The current node’s neighbor is an “in progress” node
  - Since that other node is “in progress”, the current node is reachable from it
  - The back edge is a path to that other node
  - **Cycle!**



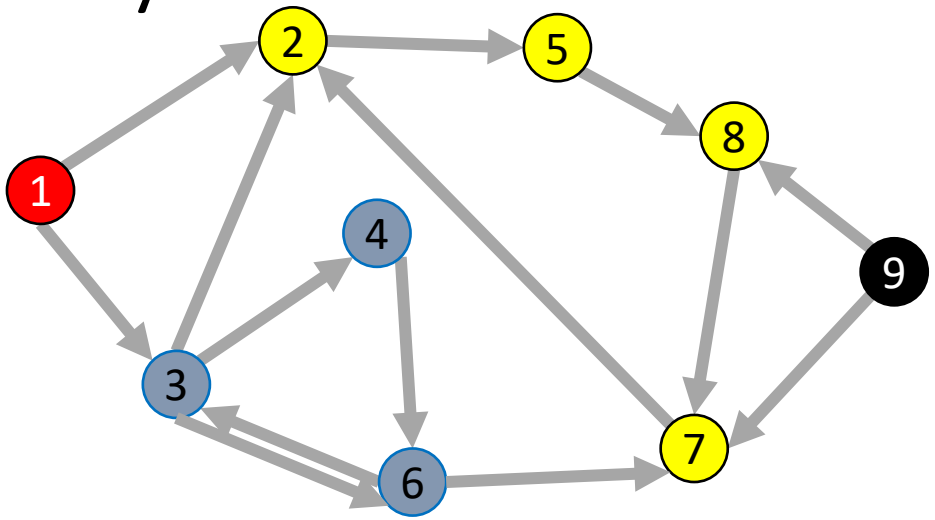
Cycle Detection

Idea: Look for a back edge!




```
Boolean hasCycle(graph) {
        for (v : graph.vertices) {
                if (! v marked “done”) {
                        if (hasCycle(graph, v)) { return true; }
                }
        }
        return false;
}
boolean hasCycle(graph, curr) {
        mark curr as “visited”;
        cycleFound = false;
        for (v : neighbors(current)) {
                if (v marked “visited” && ! v marked “done”) {
                        cycleFound = true;
                }
                if (! v marked “visited” && !cycleFound) {
                        cycleFound = hasCycle(graph, v);
                }
        }
        mark curr as “done”;
        return cycleFound;
}
```

Cycle Detection – Worked Example

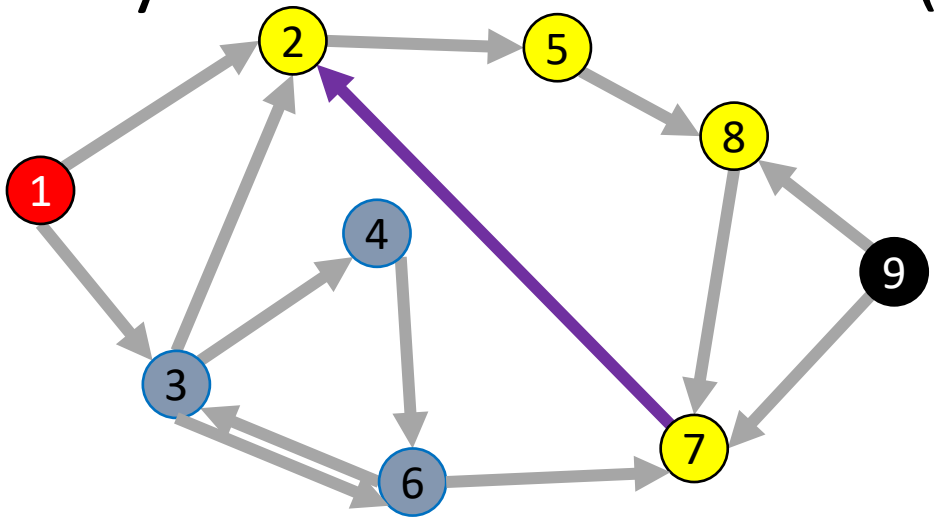


Node	Visited?	Done?	Other Info
1			
2			
3			
4			
5			
6			
7			
8			
9			

Starting from the current node:
  for each non-done neighbor:
    if the neighbor is visited:
      we found a cycle!
    else:
      mark the neighbor as visited
      do a DFS from the neighbor
  mark the current node as done

(Call) Stack:

Cycle Detection – (Skip to) Step 4: Call DFS on 7

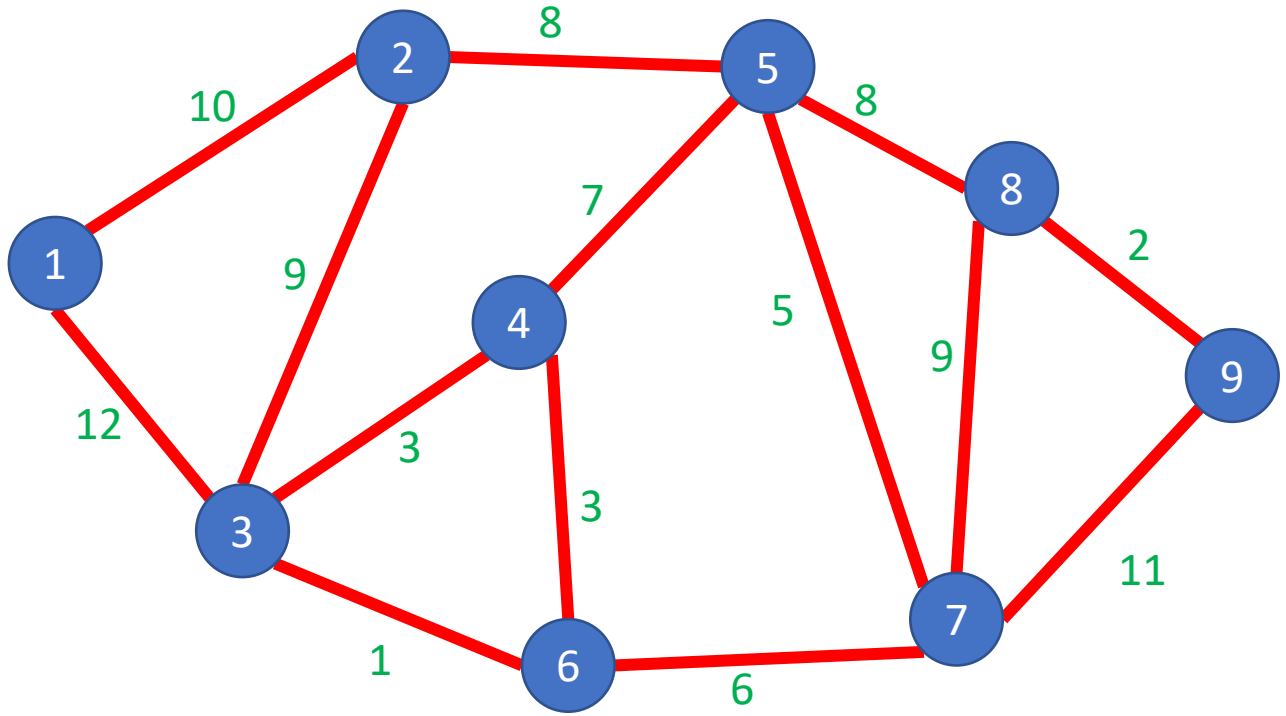


Node	Visited?	Done?	Other Info
1	True		
2	True		
3			
4			
5	True		
6			
7	True		
8	True		
9			

Starting from the current node:
  for each non-done neighbor:
    if the neighbor is visited:
      we found a cycle!
    else:
      mark the neighbor as visited
      do a DFS from the neighbor
  mark the current node as done

(Call) Stack: 1, 2, 5, 8, 7

Single-Source Shortest Path of a Weighted Graph



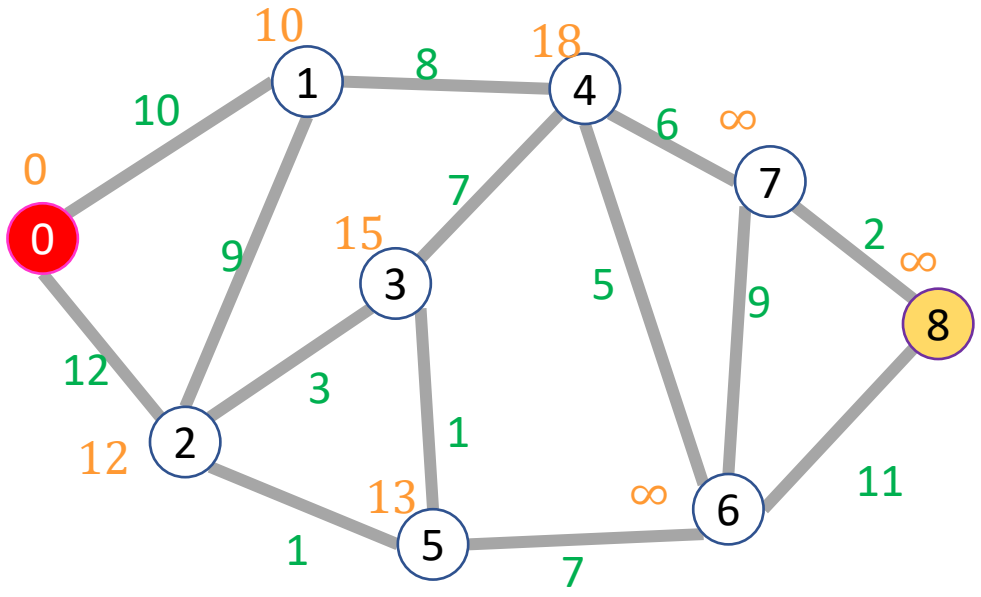
Find the quickest way to get from UW to each of these other places

Given a graph $G = (V, E)$ and a start node $s \in V$, for each $v \in V$ find the **least-weight** path from $s \to v$ (call this weight $\delta(s, v)$)

(assumption: all edge weights are non-negative)

Dijkstra's Algorithm

- Input: graph with **no negative edge weights**, start node s, optional end node t
- Behavior: Start with node s, repeatedly go to the incomplete node "nearest" to s
- Output:
  - Distance from start to end
  - Distance from start to every node



Dijkstra's

```
Distance to start = 0
Add the start node to PQ with priority 0
Mark start as "seen"
While the PQ is not empty:
    curr = PQ.extract();
    mark curr as "done"
    for each neighbor v of curr:
        d = distance to curr + weight of (curr,v)
        if v is not "seen":
            mark v as "seen"
            distance to v = d
            PQ.add(v, d);
        if v is not "done" && d < distance to v:
            distance to v = d
            PQ.decreaseKey(v, d)
```

Loops $|E|$ times

Worst case $\Theta(\log|V|)$ each

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as done
for each not-done neighbor:
    Update its distance if we found a better path

Seen = added to the priority queue
Done = removed from the priority queue
When it's done we've found the shortest path to that node

Running time: $\Theta(|E|\log|V|)$

Dijkstra's Algorithm (1/8)

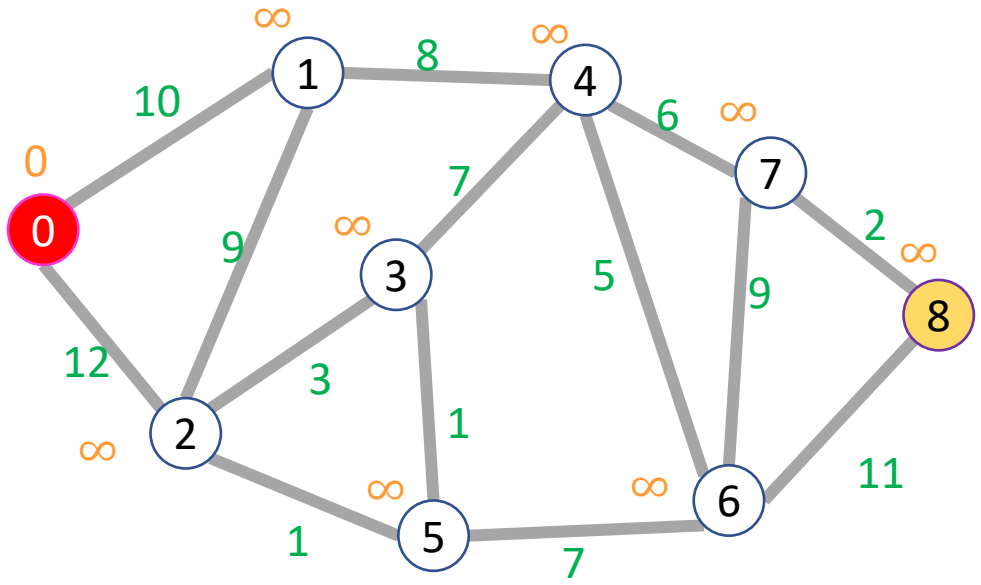
Start: 0

End: 8

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as done
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	True	False	0
1	False	False	∞
2	False	False	∞
3	False	False	∞
4	False	False	∞
5	False	False	∞
6	False	False	∞
7	False	False	∞
8	False	False	∞



Dijkstra's Algorithm (2/8)

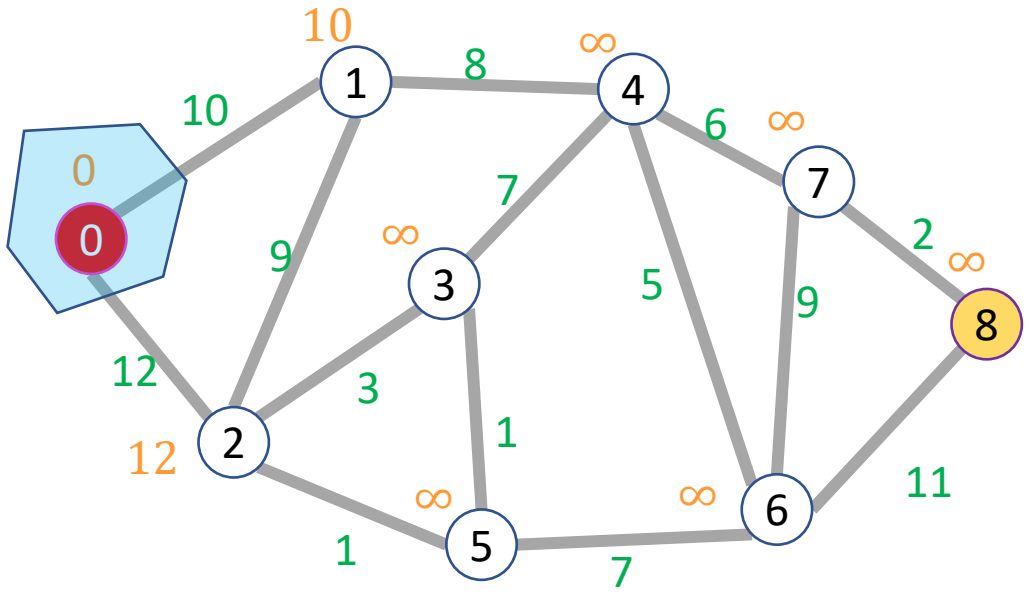
Start: 0

End: 8

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as done
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	True	True	0
1	True	False	0+10 = 10
2	True	False	0+12 = 12
3	False	False	∞
4	False	False	∞
5	False	False	∞
6	False	False	∞
7	False	False	∞
8	False	False	∞



Dijkstra's Algorithm (3/8)

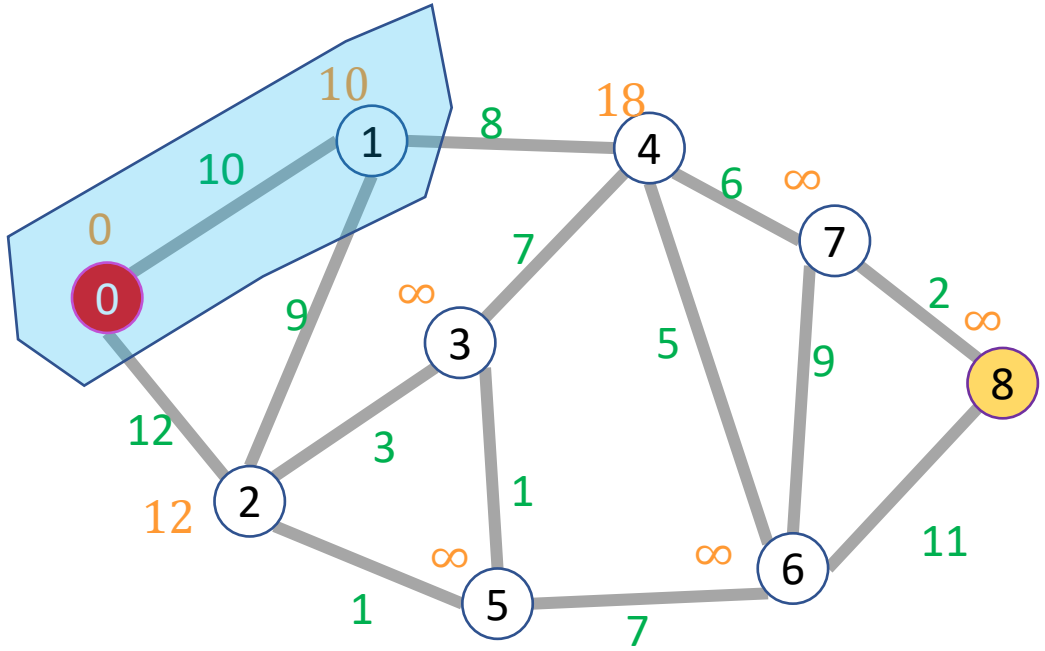
Start: 0

End: 8

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as done
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	True	True	0
1	True	True	10
2	True	False	12
3	False	False	∞
4	True	False	10+8 = 18
5	False	False	∞
6	False	False	∞
7	False	False	∞
8	False	False	∞



Dijkstra's Algorithm (4/8)

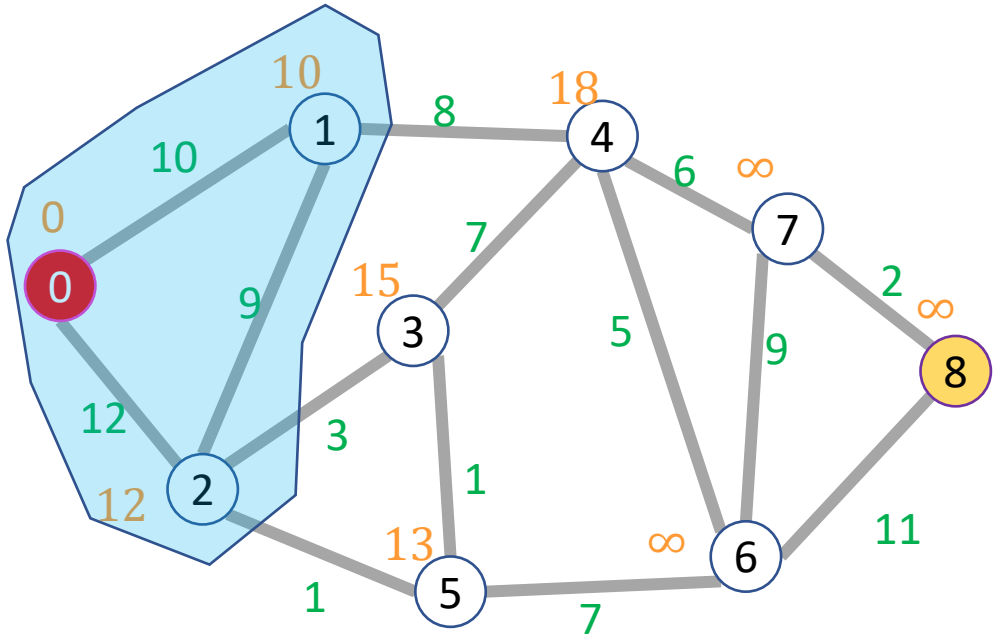
Start: 0

End: 8

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as done
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	True	True	0
1	True	True	10
2	True	True	12
3	True	False	12+3 = 15
4	True	False	18
5	True	False	12+1 = 13
6	False	False	∞
7	False	False	∞
8	False	False	∞



Dijkstra's Algorithm (5/8)

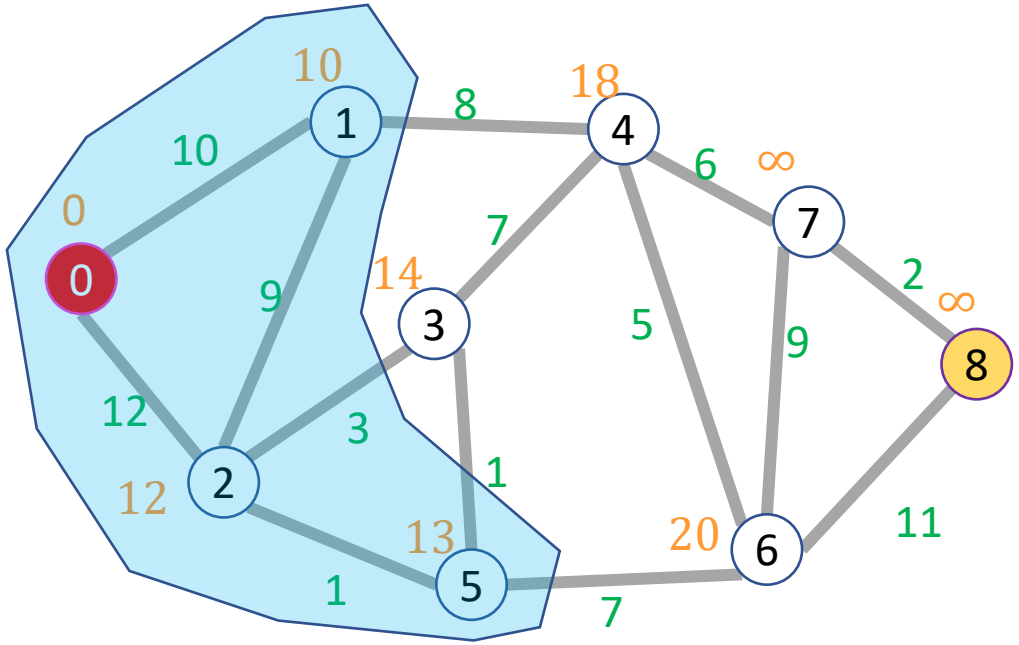
Start: 0

End: 8

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as done
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	True	True	0
1	True	True	10
2	True	True	12
3	True	False	13+1 = 14
4	True	False	18
5	True	True	13
6	True	False	13+7 = 20
7	False	False	∞
8	False	False	∞



Dijkstra's Algorithm (6/8)

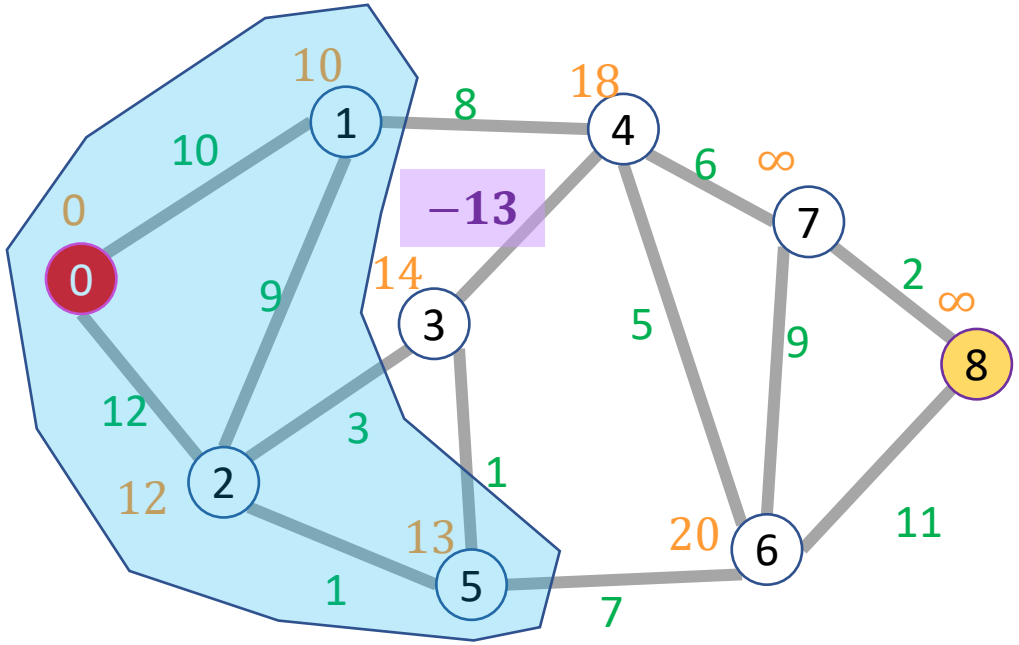
What if we had a negative-weight edge?

Start: 0

End: 8

Node	Seen?	Done?	Distance
0	True	True	0
1	True	True	10
2	True	True	12
3	True	False	14
4	True	False	18
5	True	True	13
6	True	False	20
7	False	False	∞
8	False	False	∞

Extract a node from priority queue (making it "done")
Mark extracted node as done
for each not-done neighbor:
    Update its distance if we found a better path



Dijkstra's Algorithm (7/8)

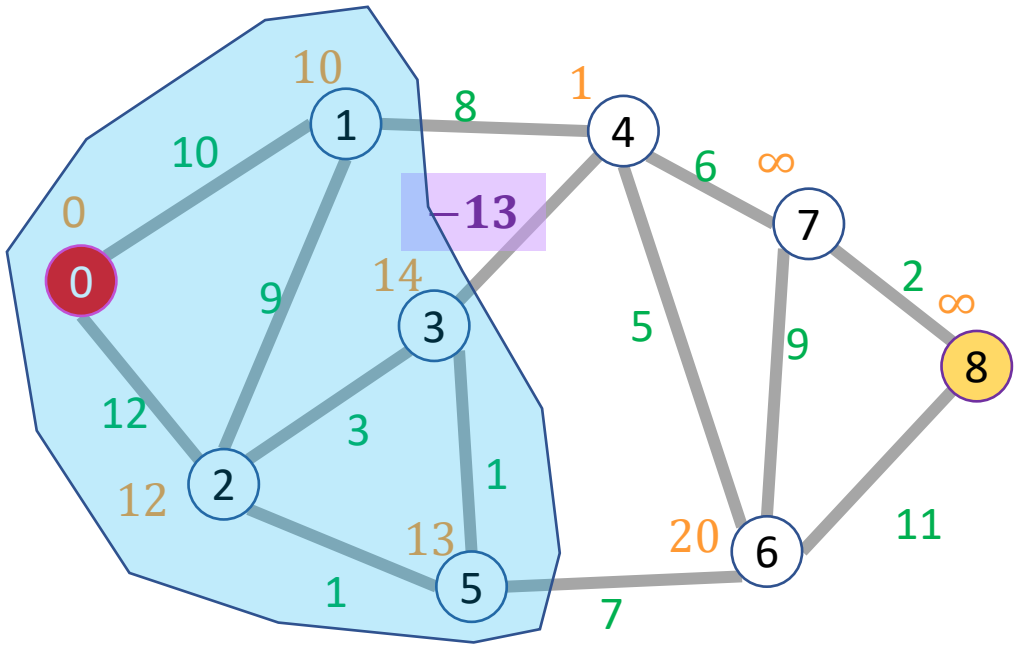
What if we had a negative-weight edge?

Start: 0

End: 8

Node	Seen?	Done?	Distance
0	True	True	0
1	True	True	10
2	True	True	12
3	True	True	14
4	True	False	14-13 = 1
5	True	True	13
6	True	False	20
7	False	False	∞
8	False	False	∞

Extract a node from priority queue (making it "done")
Mark extracted node as done
for each not-done neighbor:
    Update its distance if we found a better path



Dijkstra's Algorithm (8/8)

What if we had a negative-weight edge?

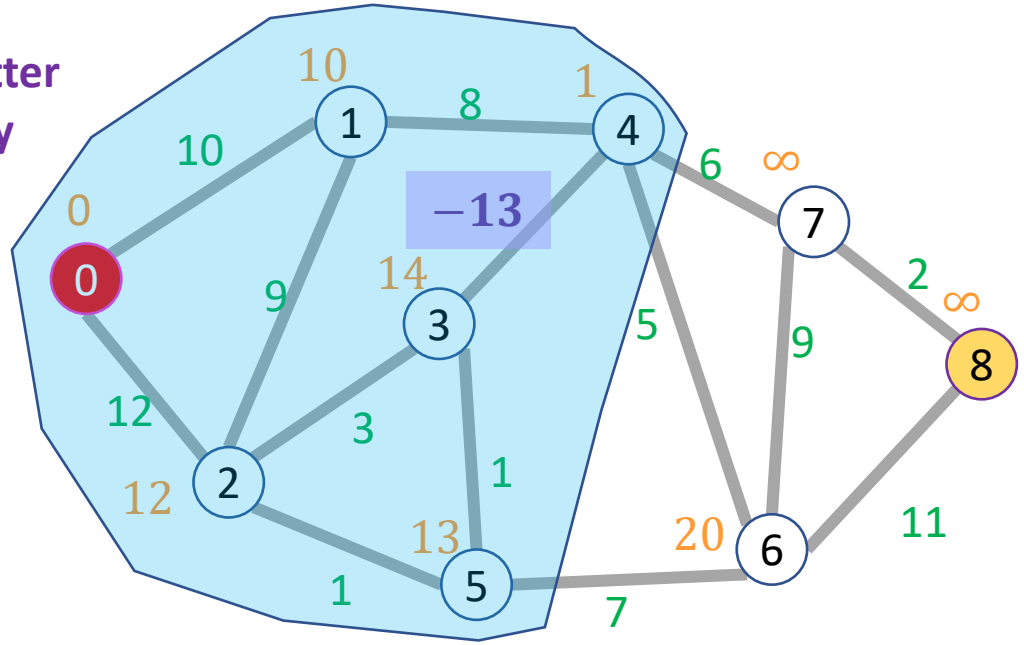
Start: 0

End: 8

Node	Seen?	Done?	Distance
0	True	True	0
1	True	True	10 or 9?
2	True	True	12
3	True	True	14
4	True	True	1
5	True	True	13
6	True	False	20
7	False	False	∞
8	False	False	∞

There's now a better path to node 1, by way of node 4!

Extract a node from priority queue (making it "done")
Mark extracted node as done
for each not-done neighbor:
    Update its distance if we found a better path



Dijkstra's Algorithm - Pseudocode

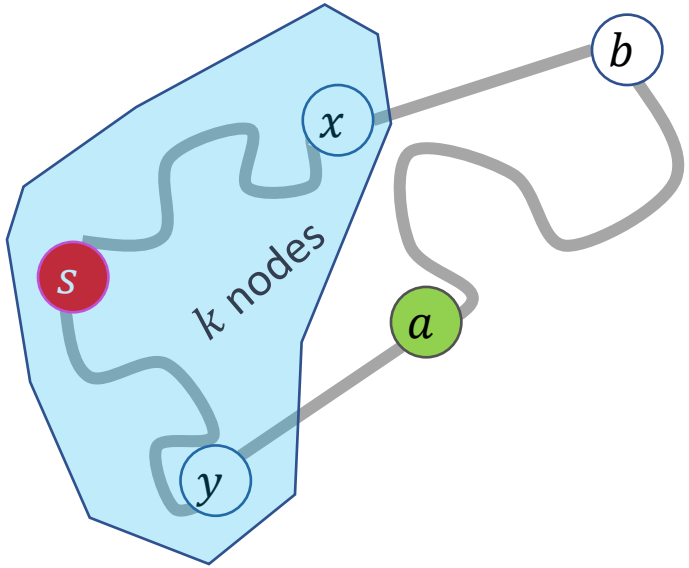
```
int dijkstras(graph, start, end) {
        distances = [∞, ∞, ∞,…];  // one index per node
        seen = [False,False,False,…];  // one index per node
        done = [False,False,False,…];  // one index per node
        PQ = new MinHeap();
        PQ.insert(0, start);  // priority=0, value=start
        distances[start] = 0;
        while (!PQ.isEmpty) {
                current = PQ.extract();
                done[current] = True;
                for (neighbor : current.neighbors) {
                        new_dist = distances[current]+weight(current, neighbor);
                        if (! seen[neighbor]) {
                                seen[neighbor] = True;
                                distances[neighbor] = new_dist;
                                PQ.insert(new_dist, neighbor);
                        }
                        else if (! done[neighbor] && new_dist < distances[neighbor]) {
                                distances[neighbor] = new_dist;
                                PQ.decreaseKey(new_dist, neighbor);
                        }
                }
        }
        return distances[end]
}
```

Dijkstra's Algorithm: Running Time

- How many total priority queue operations are necessary?
  - How many times is each node added to the priority queue?
  - How many times might a node's priority be changed?
- What's the running time of each priority queue operation?
- Overall running time:
  - $\Theta(|E|\log|V|)$

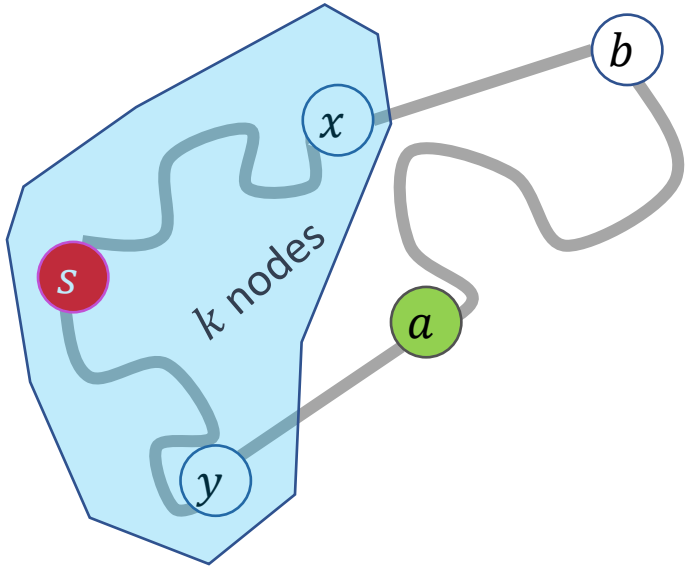
Dijkstra's Algorithm: Correctness

- Claim: when a node is removed from the priority queue, its distance is that of the shortest path
- Induction over number of completed nodes
- Base Case: Only the start node removed
  - It is indeed 0 away from itself
- Inductive Step:
  - If we have correctly found shortest paths for the first k nodes, then when we remove node $k + 1$ we have found its shortest path



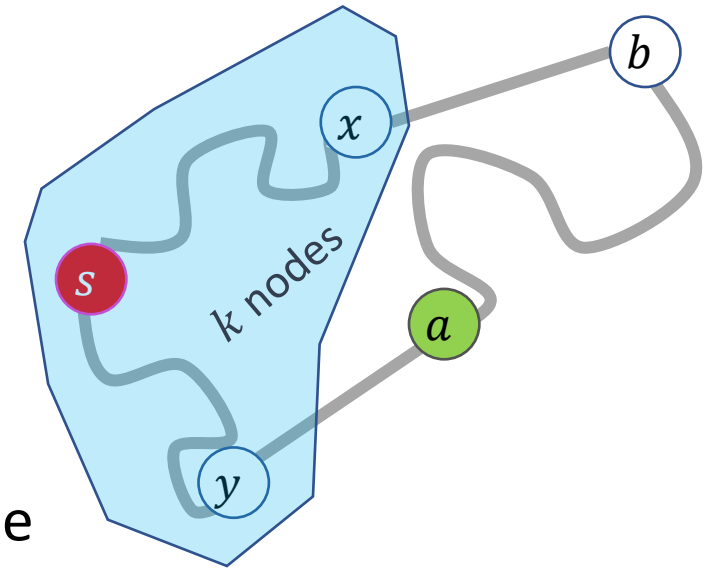
Dijkstra's: Correctness – Properties of a

- Suppose a is the next node removed from the queue. What do we know bout a?



Dijkstra's: Correctness – Why a is correct

- Suppose a is the next node removed from the queue.
  - No other node incomplete node has a shorter path discovered so far
- Claim: no undiscovered path to a could be shorter
  - Consider any other incomplete node b that is 1 edge away from a complete node
  - a is the closest node that is one away from a complete node
  - Thus no path that includes b can be a shorter path to a
  - Therefore the shortest path to a must use only complete nodes, and therefore we have found it already!



Why we need non-negative edge weights

- Suppose a is the next node removed from the queue.
  - No other node incomplete node has a shorter path discovered so far
- Claim: no undiscovered path to a could be shorter
  - Consider any other incomplete node b that is 1 edge away from a complete node
  - a is the closest node that is one away from a complete node
  - No path from b to a can have negative weight
  - Thus no path that includes b can be a shorter path to a
  - Therefore the shortest path to a must use only complete nodes, and therefore we have found it already!

