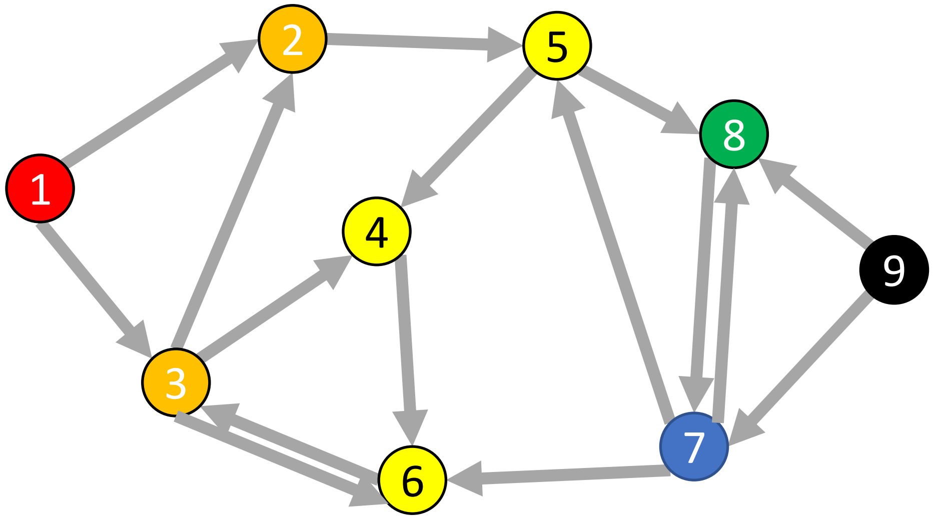
CSE 332 Autumn 2024
Lecture 19: Graphs 3

Nathan Brunelle

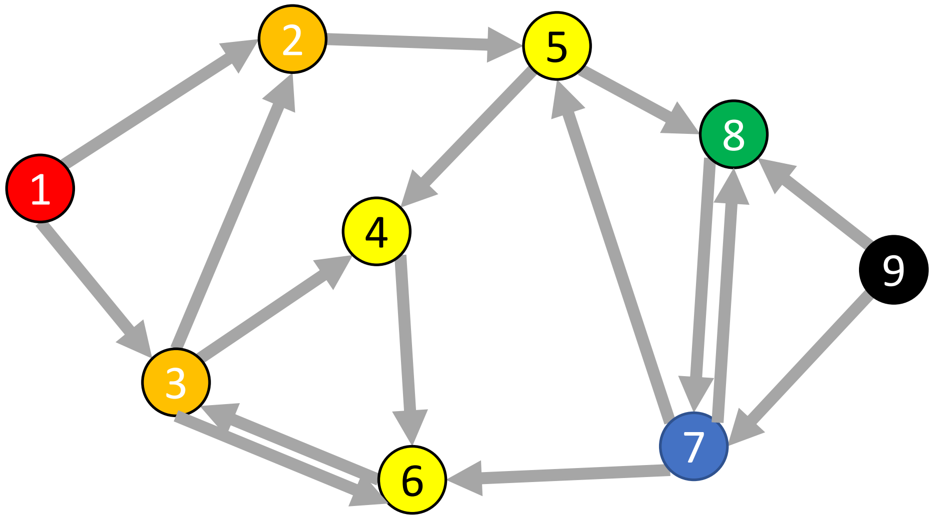
http://www.cs.uw.edu/332

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?



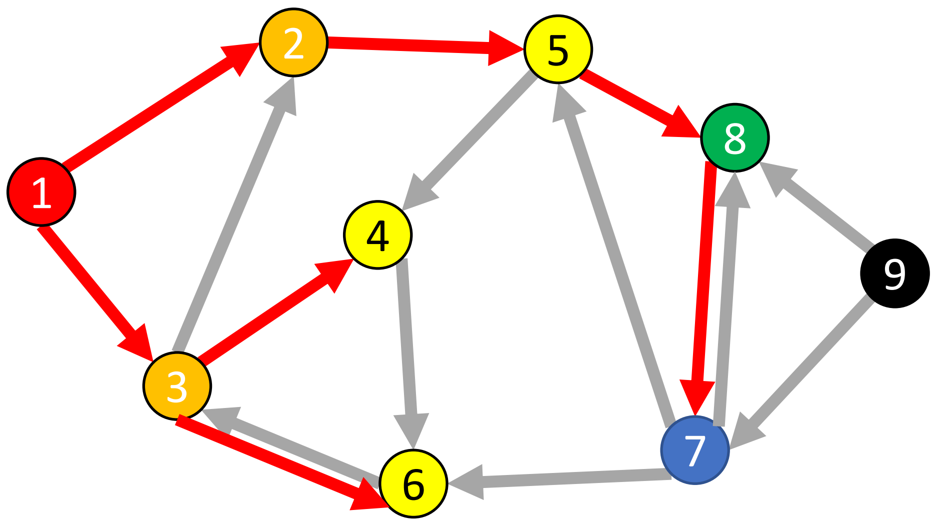
BFS



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

```
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);
                        }
                }
        }
}
```

Find Distance (unweighted)

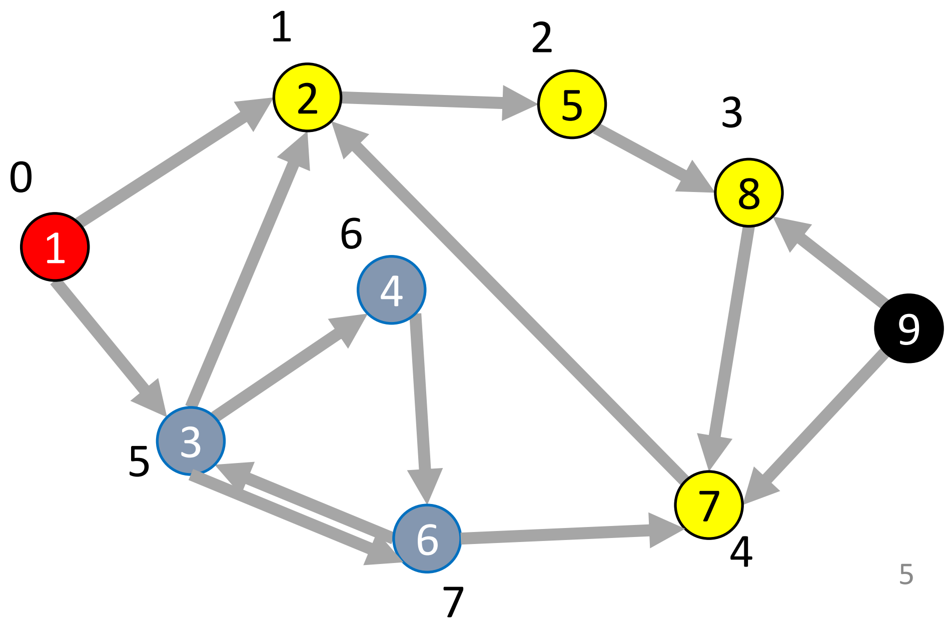


Idea: when it's seen, remember its "layer" depth!

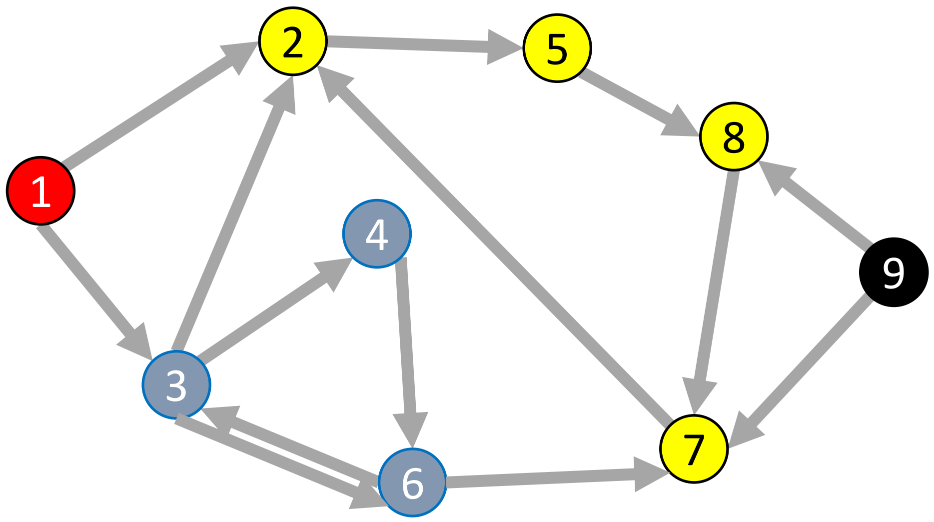
```
int findDistance(graph, s, t){
        found = new Queue();
        layer = 0;
        found.enqueue(s);
        mark s as "visited";
        While (!found.isEmpty()){
                current = found.dequeue();
                layer = depth of current;
                for (v : neighbors(current)){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

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)

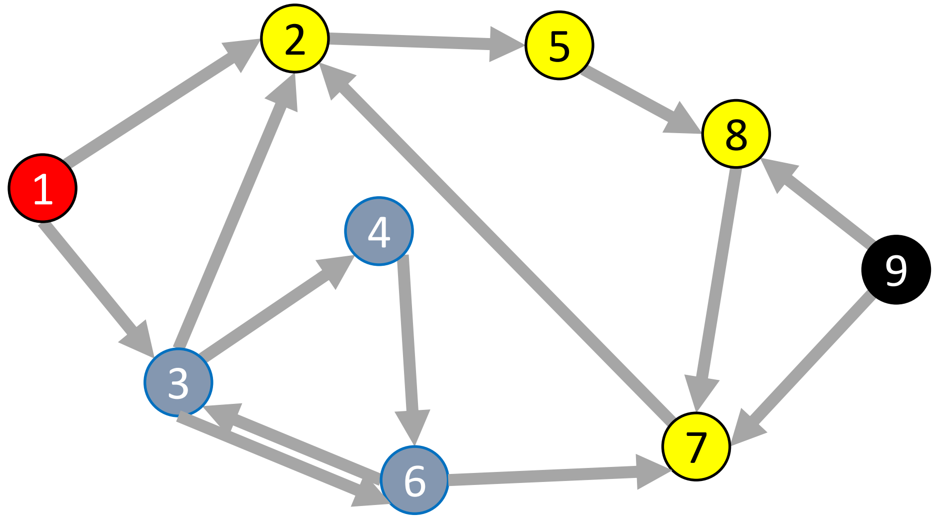


Running time: $\Theta(|V| + |E|)$

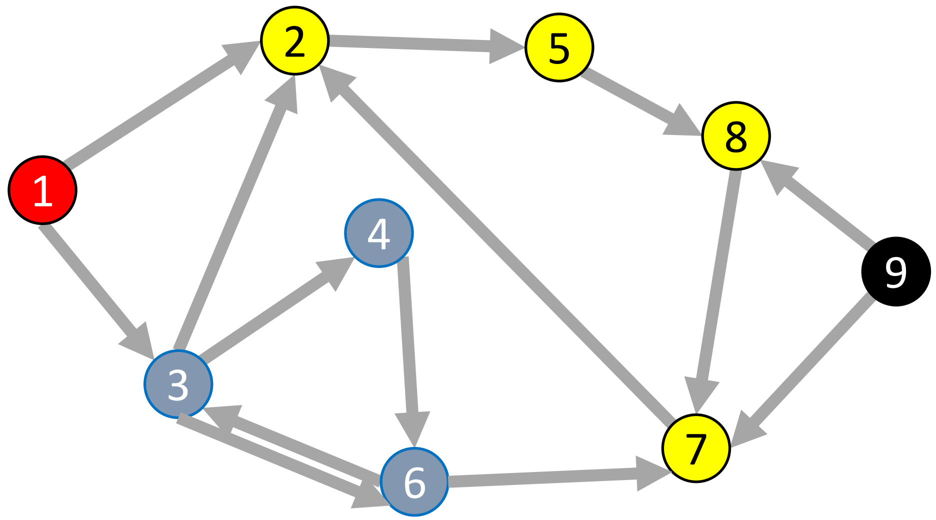
```
void dfs(graph, s){
        found = new Stack();
        found.pop(s);
        mark s as “visited”;
        While (!found.isEmpty()){
                current = found.pop();
                for (v : neighbors(current)){
                        if (! v marked “visited”){
                          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



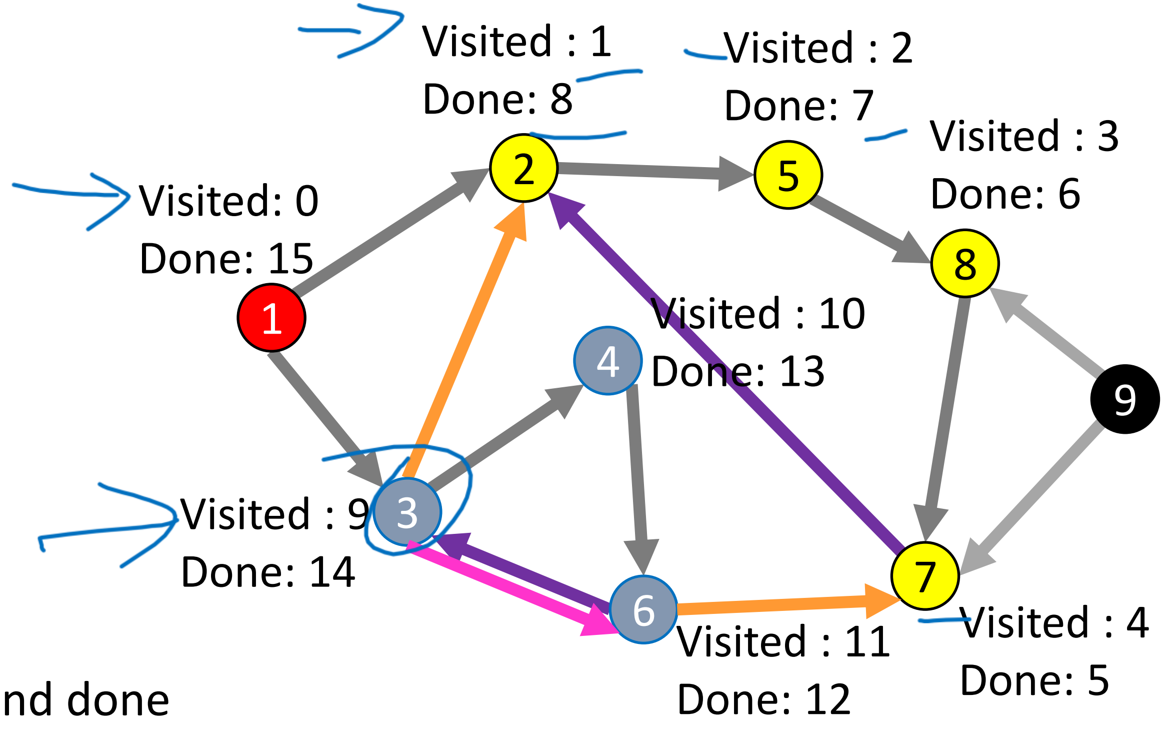
Node	Visited?	Done?	Other Info
1	T		
2	T		
3			
4			
5	T		
6			
7	T	T	
8	T		
9			

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 |

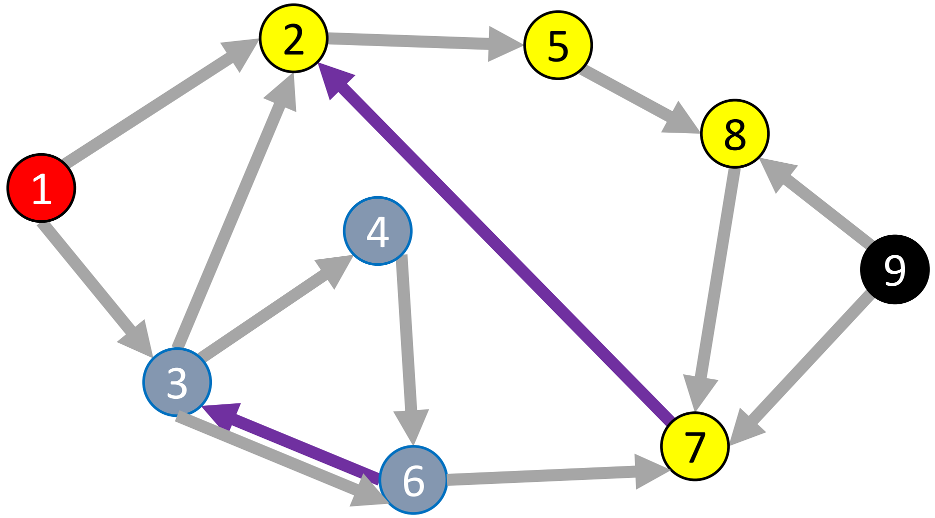
Using DFS

- 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
    - $t_{done}(b) < t_{visited}(a)$



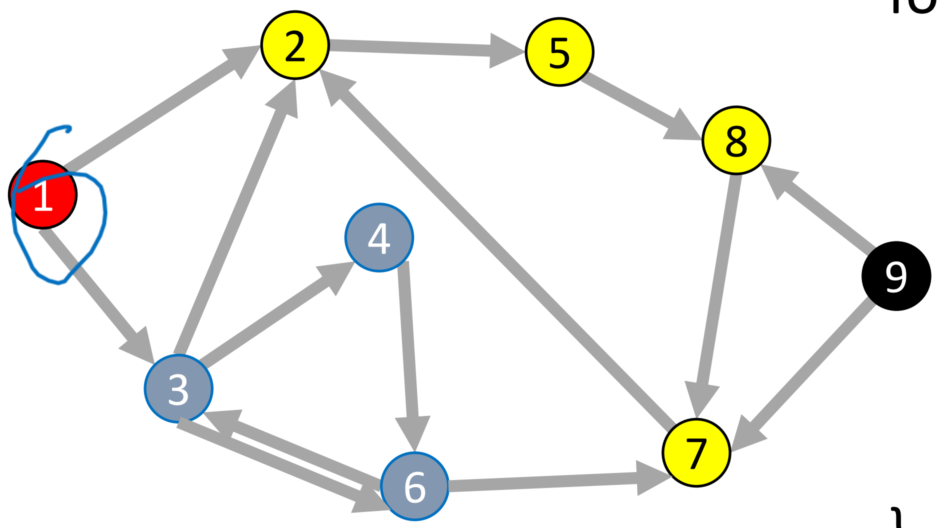
Back Edges

- 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!**



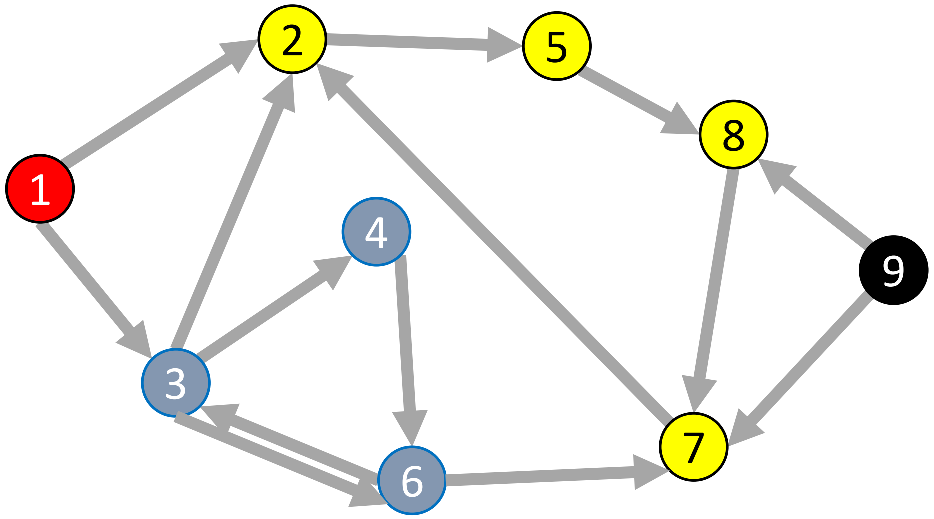
Cycle Detection

Idea: Look for a back edge!




```
boolean hasCycle(graph, 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



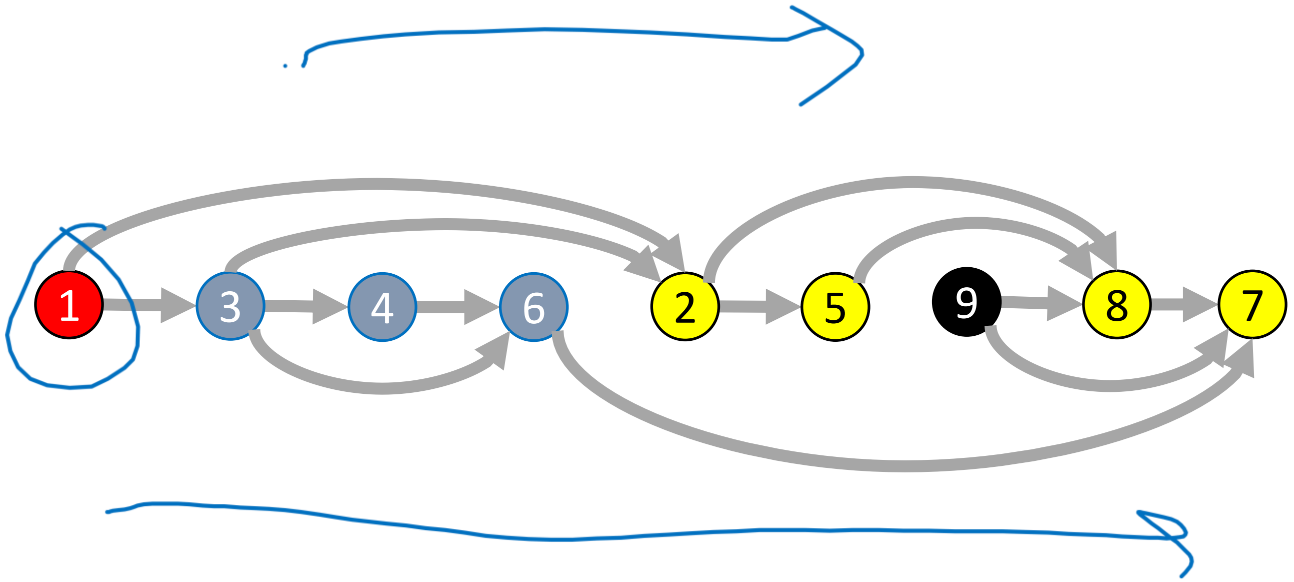
Starting from the current node:
    for each non-done neighbor:
        if the neighbor is visited:
            we found a cycle!
        else:
            mark the neighbor as visited
            do a DFS from the neighbor
    mark the current node as done

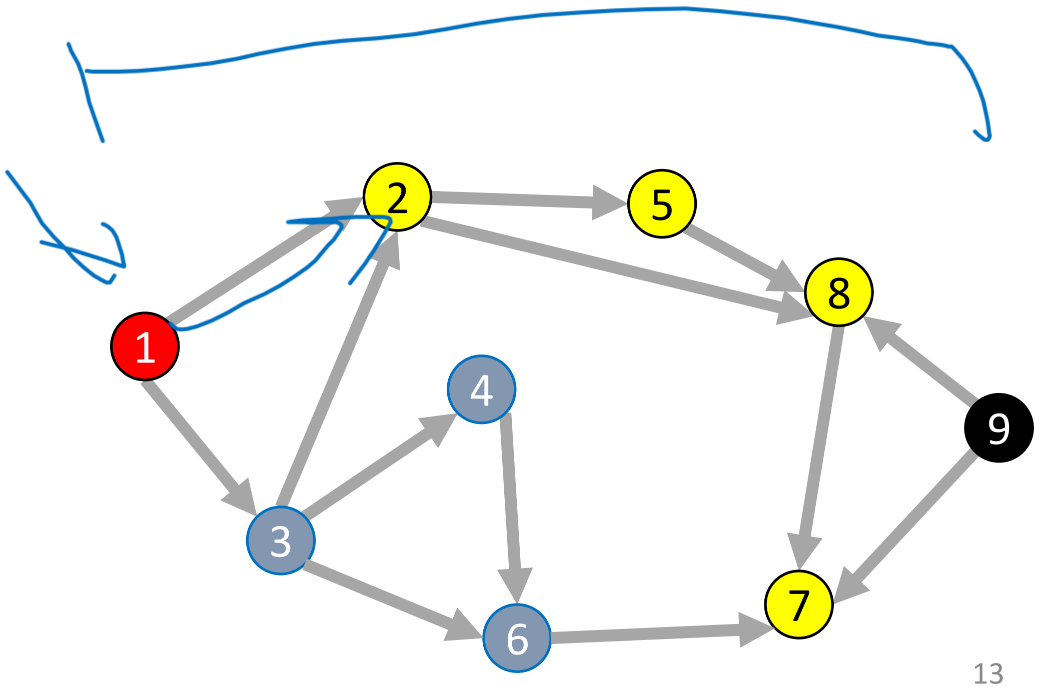
Node	Visited?	Done?	Other Info
1			
2			
3			
4			
5			
6			
7			
8			
9			

(Call) Stack:

Topological Sort

- A Topological Sort of a **directed acyclic graph** $\boldsymbol{G = (V, E)}$ is a permutation of V such that if $(u, v) \in E$ then u is before v in the permutation

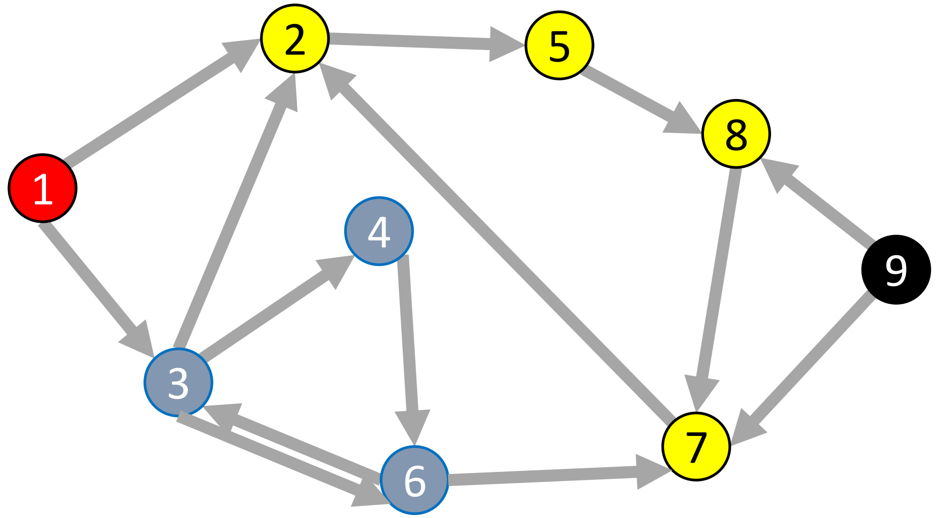




DFS Recursively

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

Idea: List in reverse order by "done" time



DFS: Topological sort

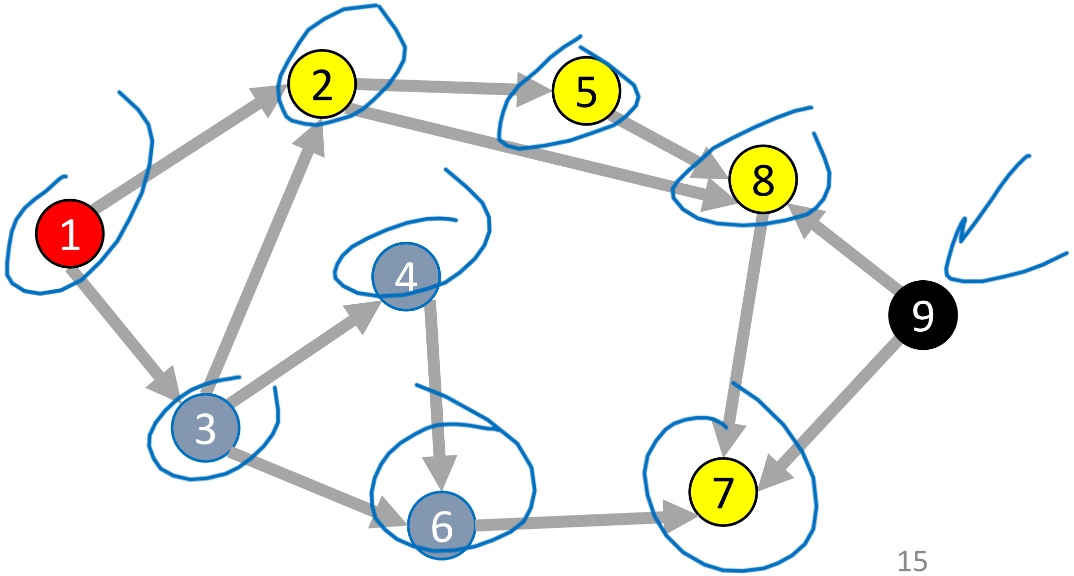
```
List topSort(graph){
        List<Nodes> done = new List<>();
        for (Node v : graph.vertices){
                if (!v.visited){
                        finishTime(graph, v, finished);
                }
        }
        done.reverse();
        return done;
}

void finishTime(graph, curr, finished){
        curr.visited = true;
        for (Node v : curr.neighbors){
                if (!v.visited){
                        finishTime(graph, v, finished);
                }
        }
        done.add(curr)
}
```

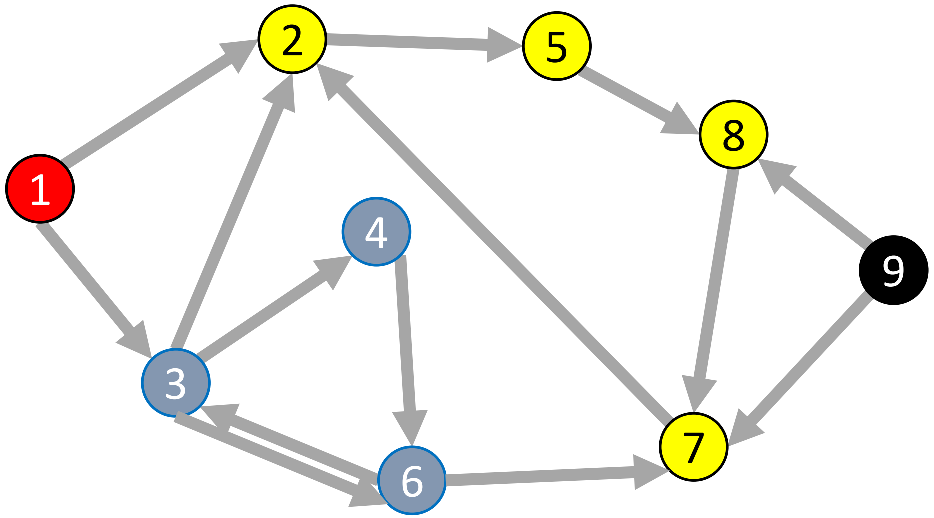
Idea: List in reverse order by “done” time

finished:

9	1	3	4	6	2	5	8	7



Topological Sort– Worked Example



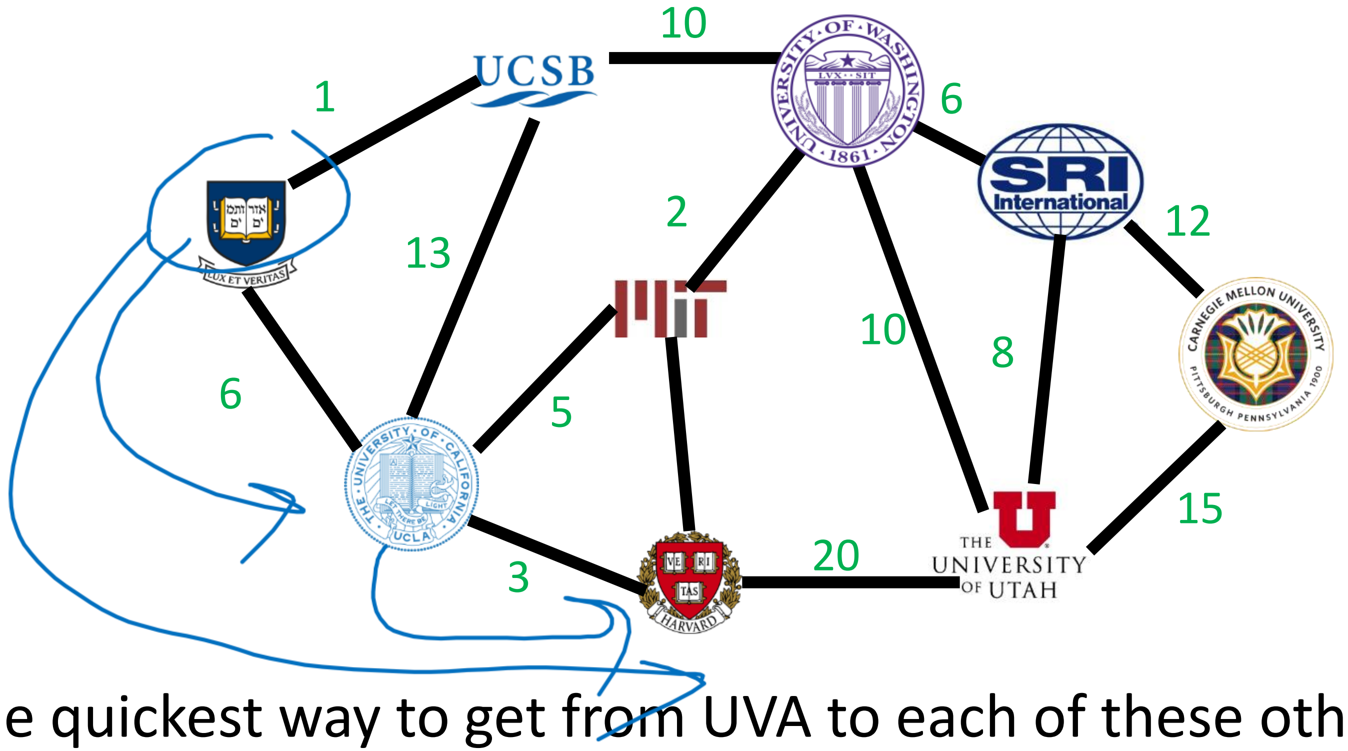
Starting from the current node:
  for each non-done neighbor:
    if the neighbor is visited:
      we found a cycle!
    else:
      mark the neighbor as visited
      do a DFS from the neighbor
  mark the current node as done
  add current node to finished

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

(Call) Stack:

finished:

Single-Source Shortest Path



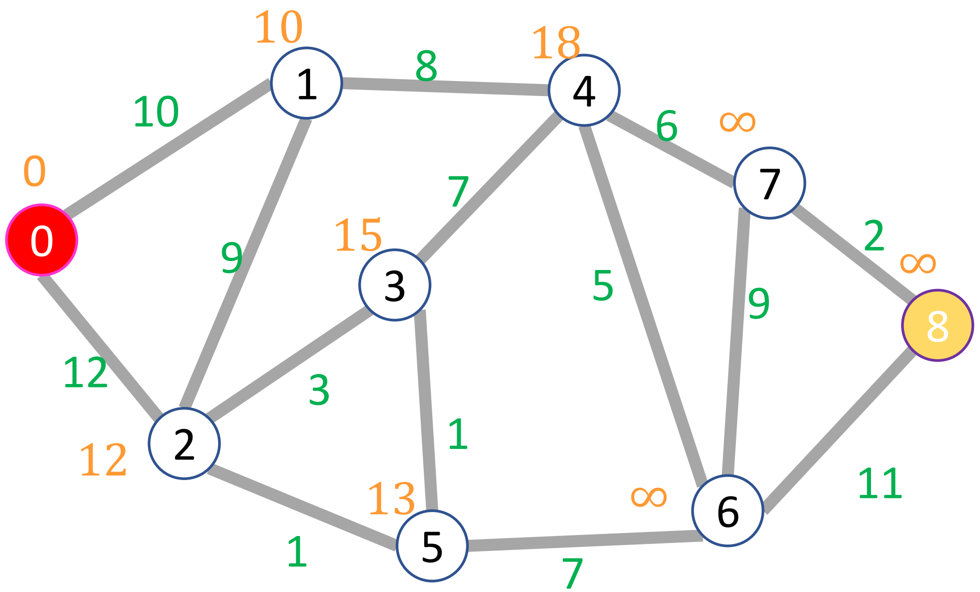
Find the quickest way to get from UVA to each of these other places

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 positive)

Dijkstra's Algorithm

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



Dijkstra's Algorithm

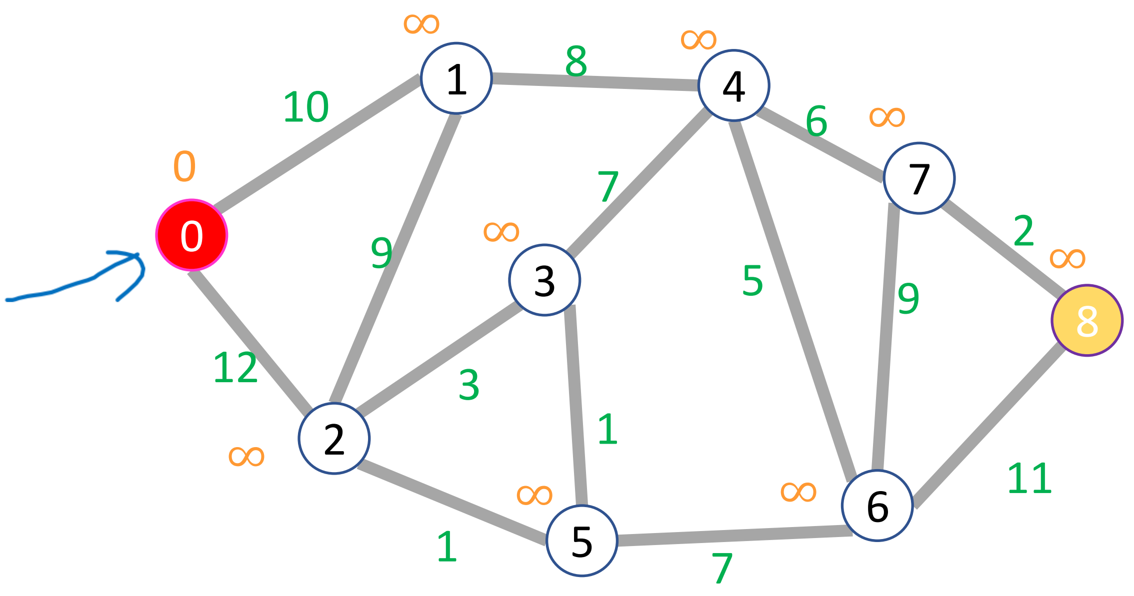
Start: 0

End: 8

Node	Done?	Distance
0	F	0
1	F	∞
2	F	∞
3	F	∞
4	F	∞
5	F	∞
6	F	∞
7	F	∞
8	F	∞

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

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



Dijkstra's Algorithm

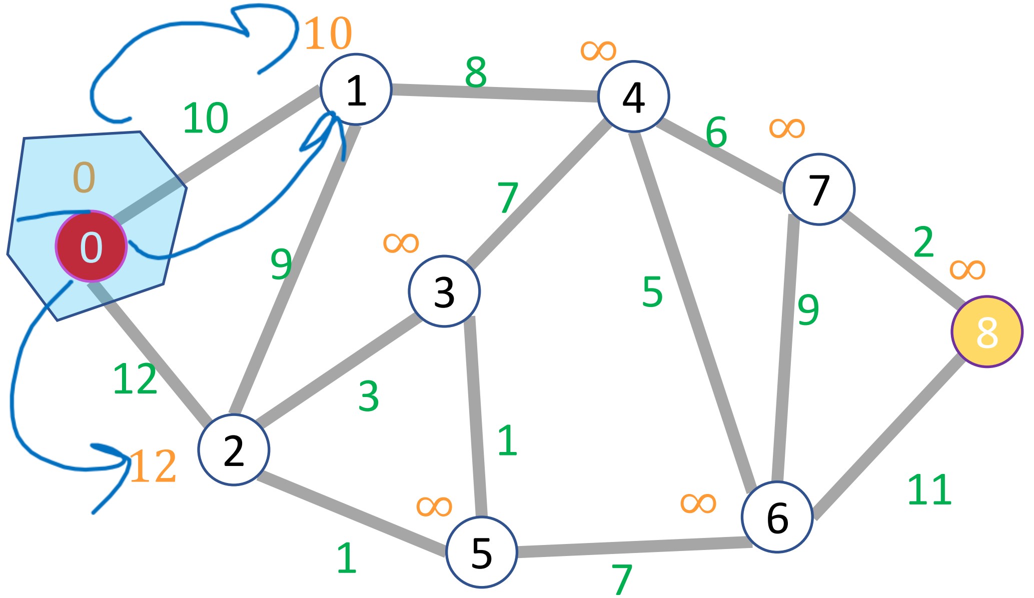
Start: 0

End: 8

Node	Done?	Distance
0	T	0
1	F	10
2	F	12
3	F	∞
4	F	∞
5	F	∞
6	F	∞
7	F	∞
8	F	∞

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

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



Dijkstra's Algorithm

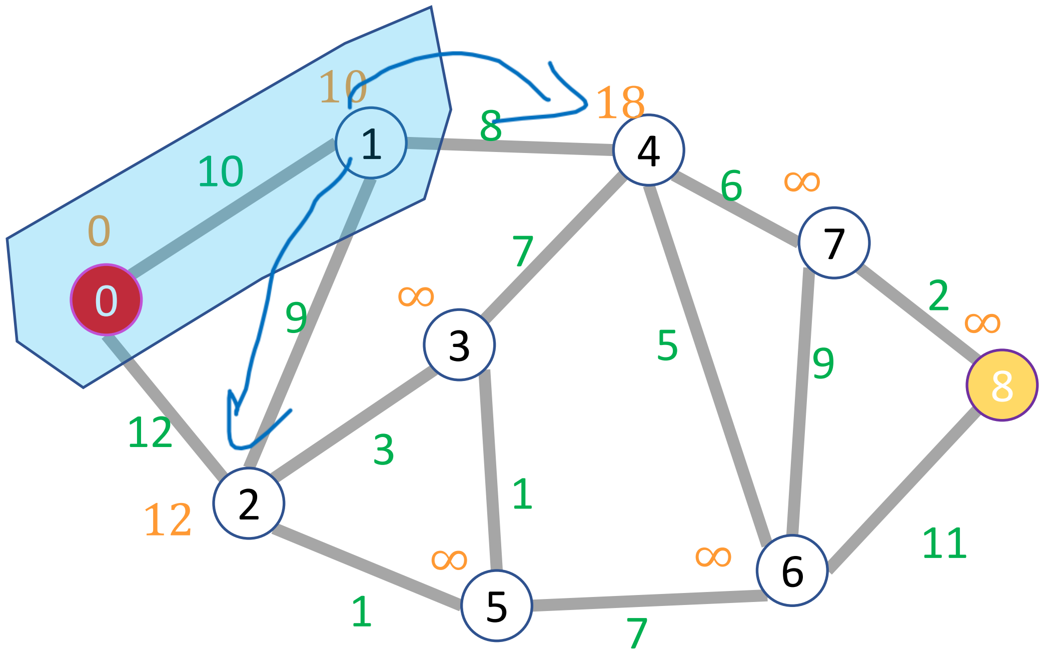
Start: 0

End: 8

Node	Done?	Distance
0	T	0
1	T	10
2	F	12
3	F	∞
4	F	18
5	F	∞
6	F	∞
7	F	∞
8	F	∞

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

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



Dijkstra's Algorithm

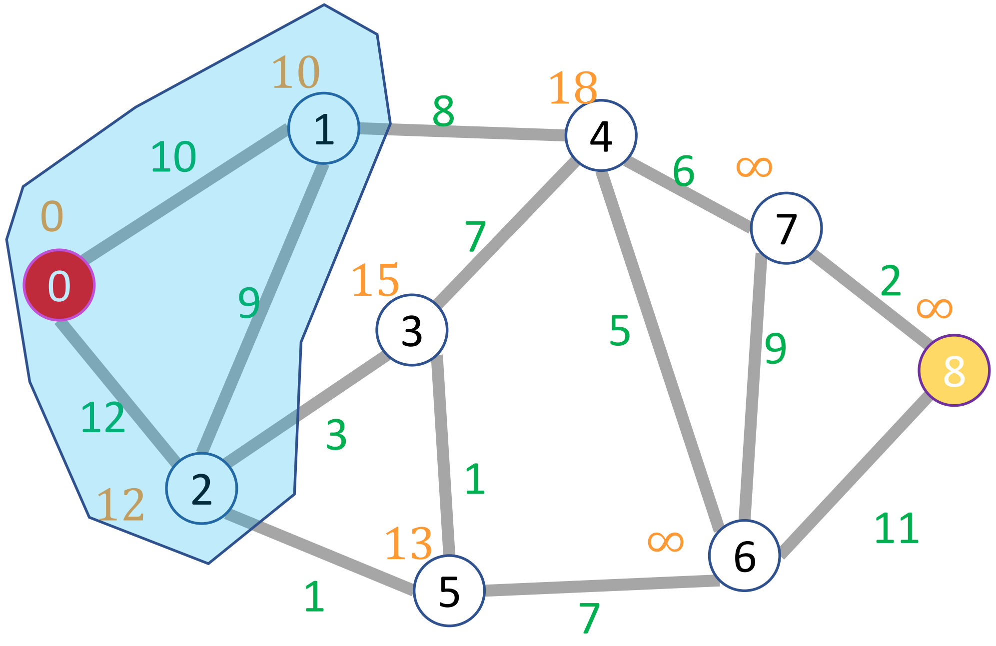
Start: 0

End: 8

Node	Done?	Distance
0	T	0
1	T	10
2	T	12
3	F	15
4	F	18
5	F	13
6	F	∞
7	F	∞
8	F	∞

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

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



Dijkstra's Algorithm

Start: 0

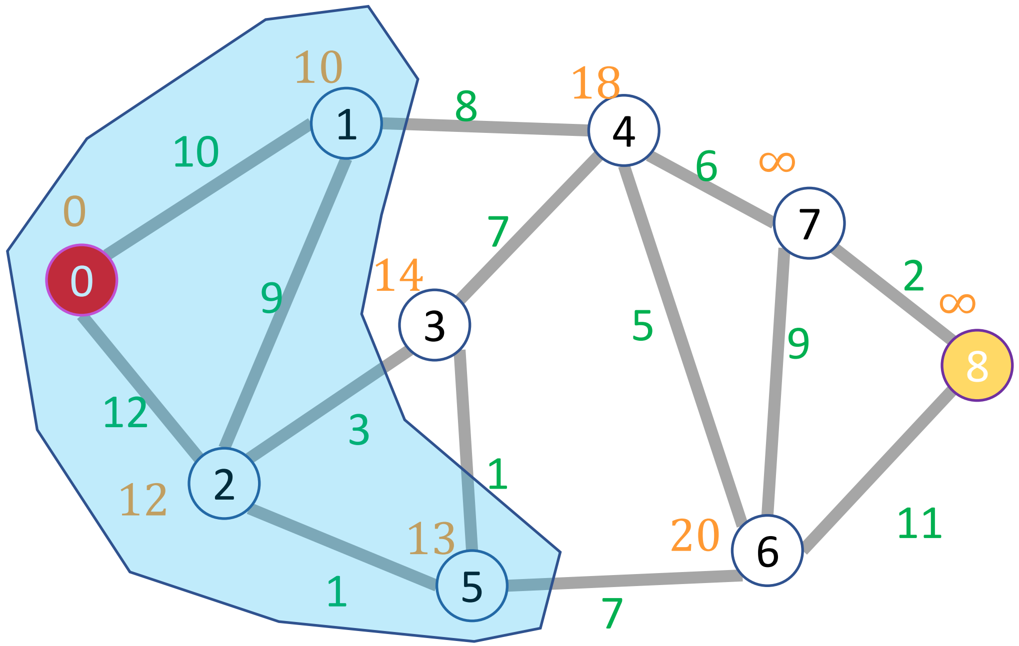
End: 8

← prev

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

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

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



Dijkstra's Algorithm

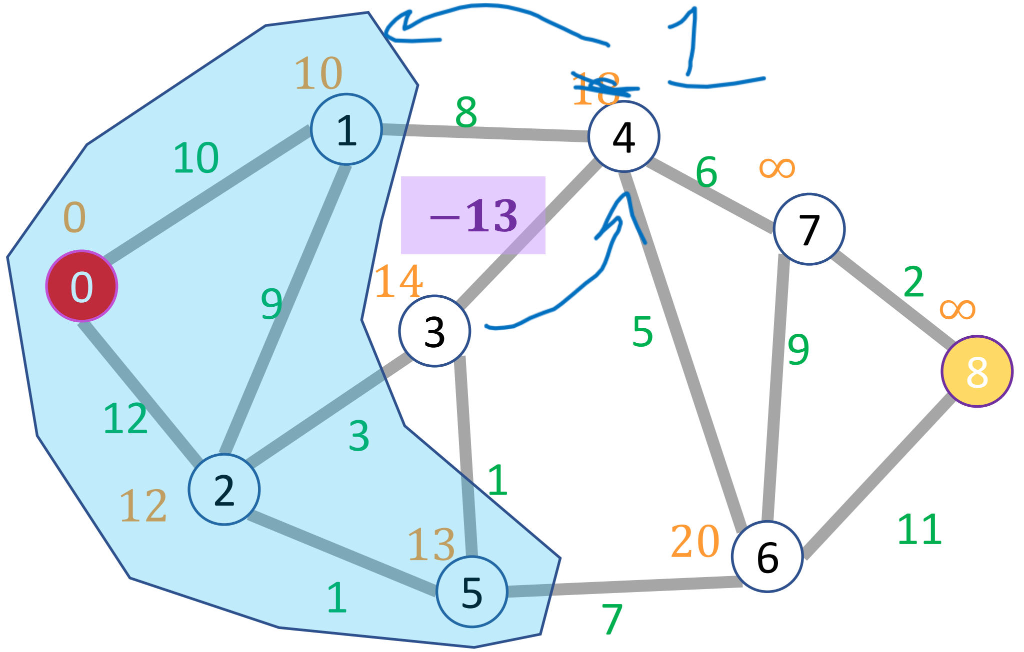
Start: 0

End: 8

What if we had a negative-weight edge?

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

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



Dijkstra’s Algorithm

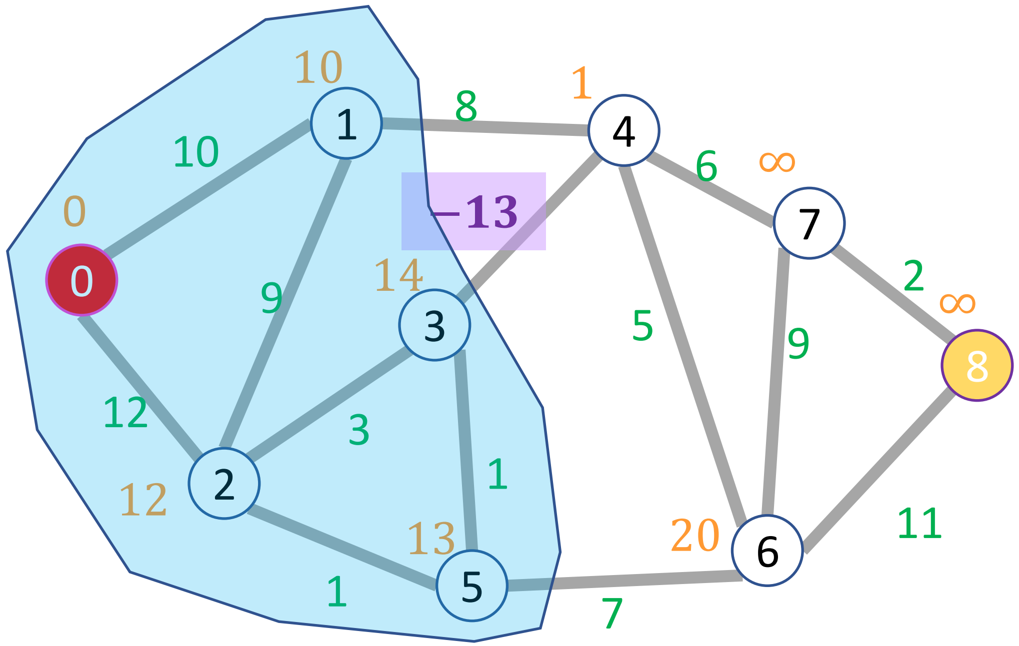
Start: 0

End: 8

Node	Done?	Distance
0	T	0
1	T	10
2	T	12
3	T	14
4	F	1
5	T	13
6	F	20
7	F	∞
8	F	∞

What if we had a negative-weight edge?

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



Dijkstra's Algorithm

Start: 0

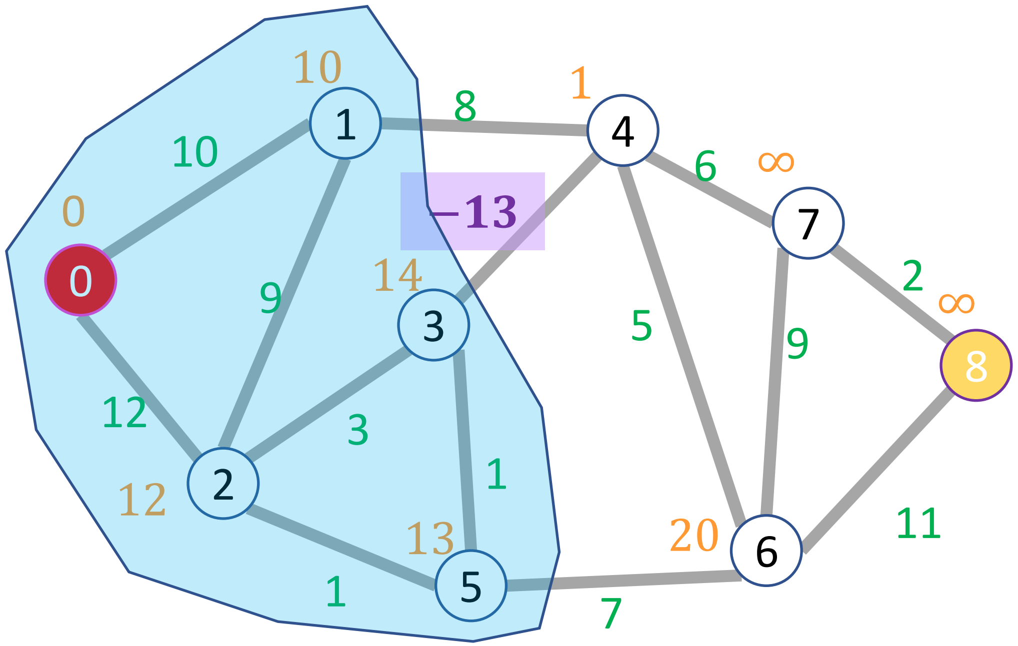
End: 8

What if we had a negative-weight edge?

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

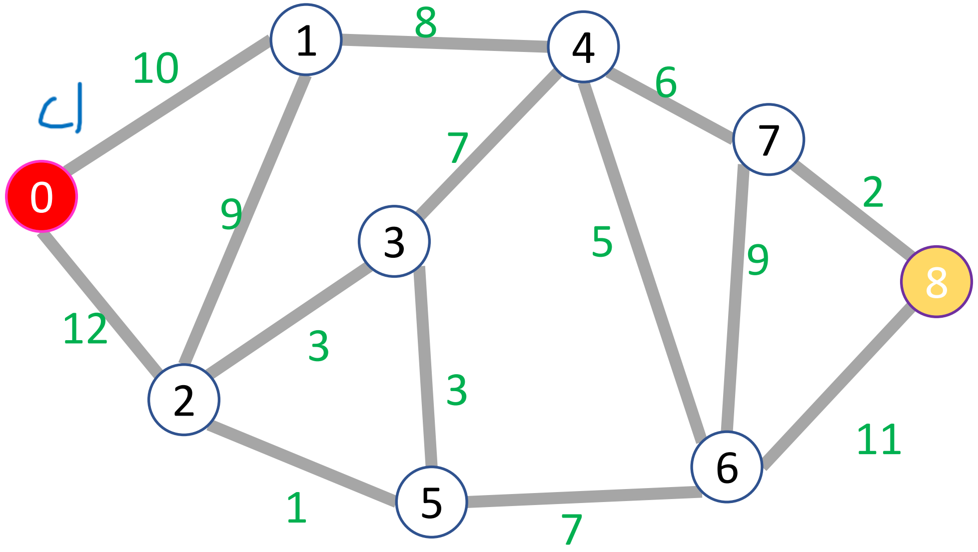
Node	Done?	Distance
0	T	0
1	T	10
2	T	12
3	T	14
4	F	1
5	T	13
6	F	20
7	F	∞
8	F	∞

There's a better path!



Dijkstra's Algorithm

```
int dijkstras(graph, start, end){
        distances = [∞, ∞, ∞,…];  // 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.deleteMin();
                done[current] = true;
                for (neighbor : current.neighbors){
                        if (!done[neighbor]){
                                new_dist = distances[current]+weight(current,neighbor);
                                if(distances[neighbor] == ∞){
                                        distances[neighbor] = new_dist;
                                        PQ.insert(new_dist, neighbor);
                                }
                                if (new_dist < distances[neighbor]){
                                        distances[neighbor] = new_dist;
                                        PQ.decreaseKey(new_dist,neighbor); }
                        }
                }
        }
        return distances[end]
}
```



log V

V

E

log V

O(E log V)

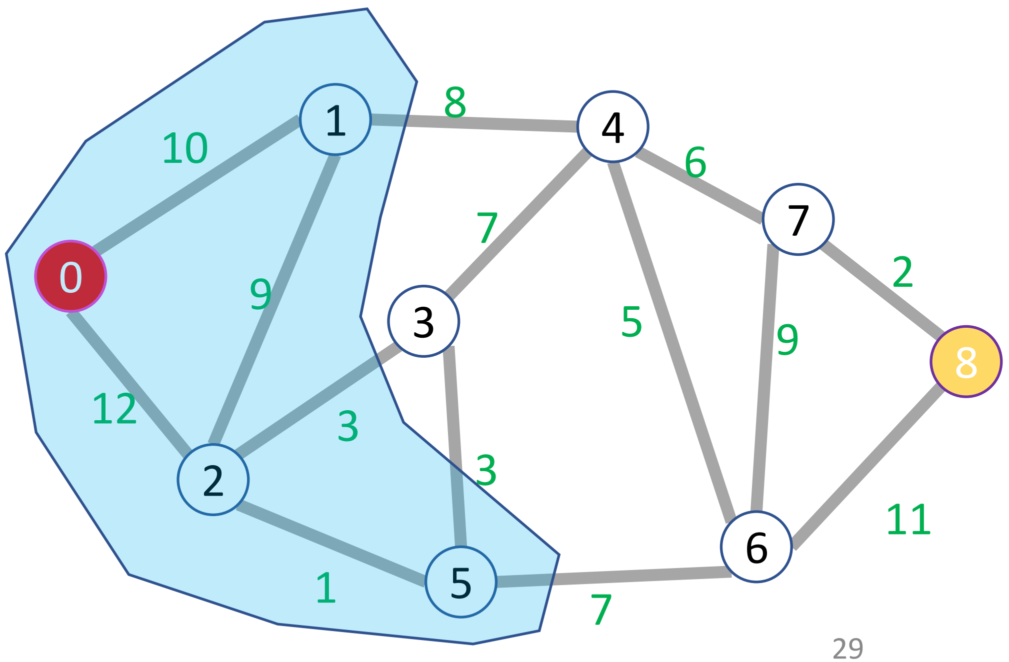
(V log V + E log V)

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:
  - $\Theta(|E| \log|V|)$

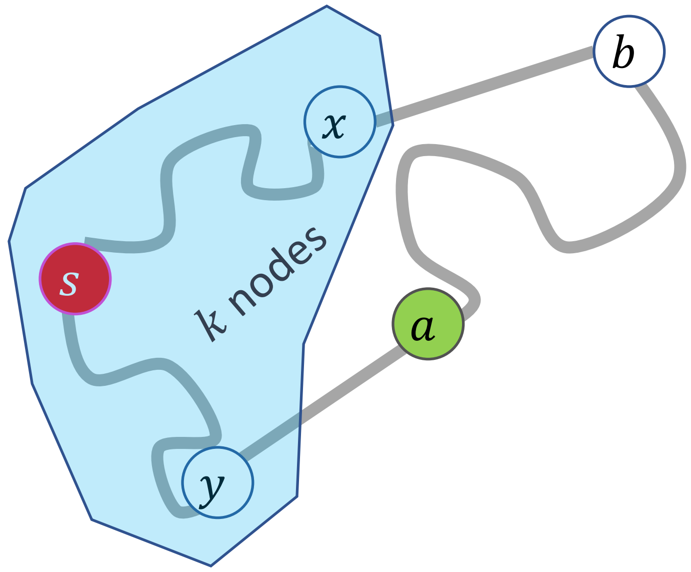
Dijkstra's Algorithm: Correctness

- Claim: when a node is removed from the priority queue, we have found its shortest path
- Induction over number of completed nodes
- Base Case:
- Inductive Step:



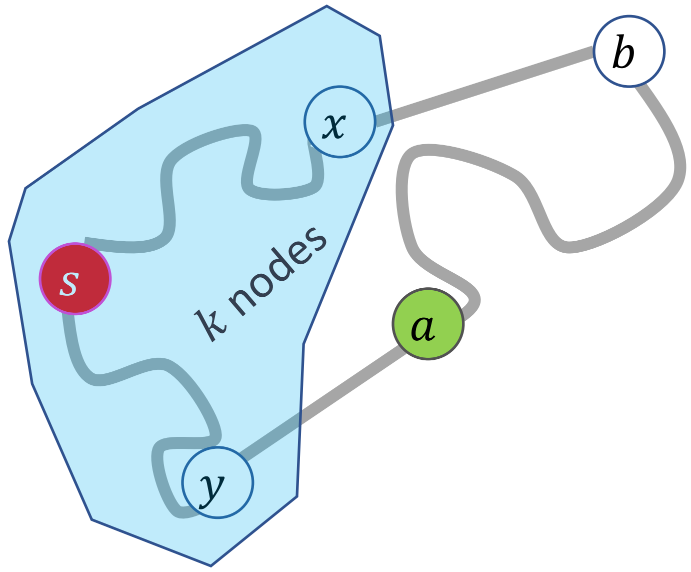
Dijkstra's Algorithm: Correctness

- 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



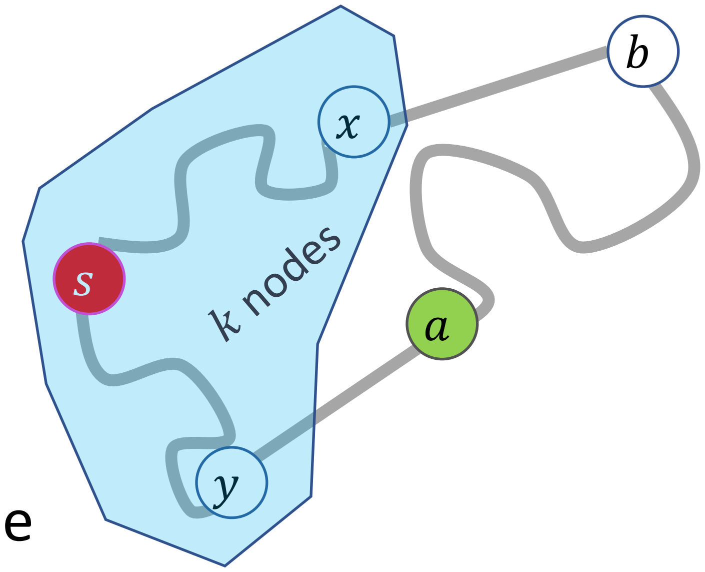
Dijkstra's Algorithm: Correctness

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



Dijkstra’s Algorithm: Correctness

- 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!



Dijkstra's Algorithm: Correctness

- 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!