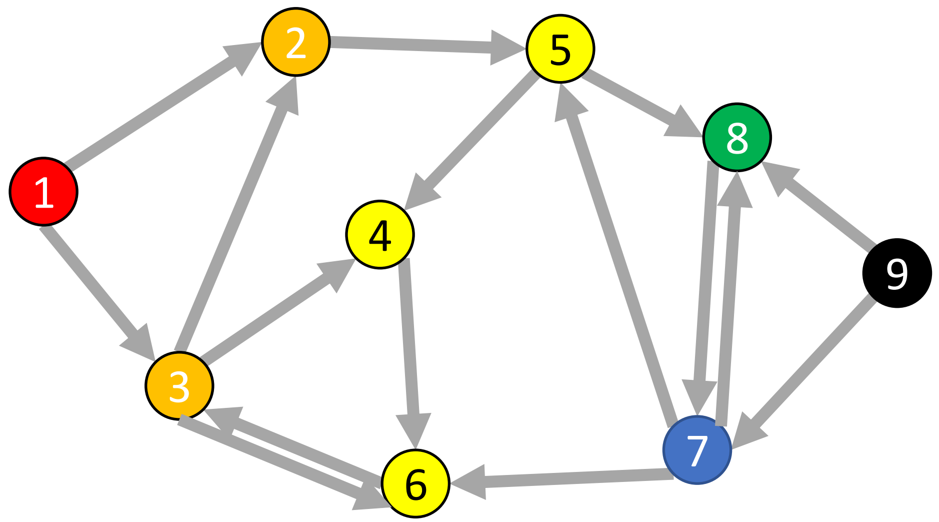
CSE 332 Winter 2026
Lecture 16: Graphs 3

Nathan Brunelle

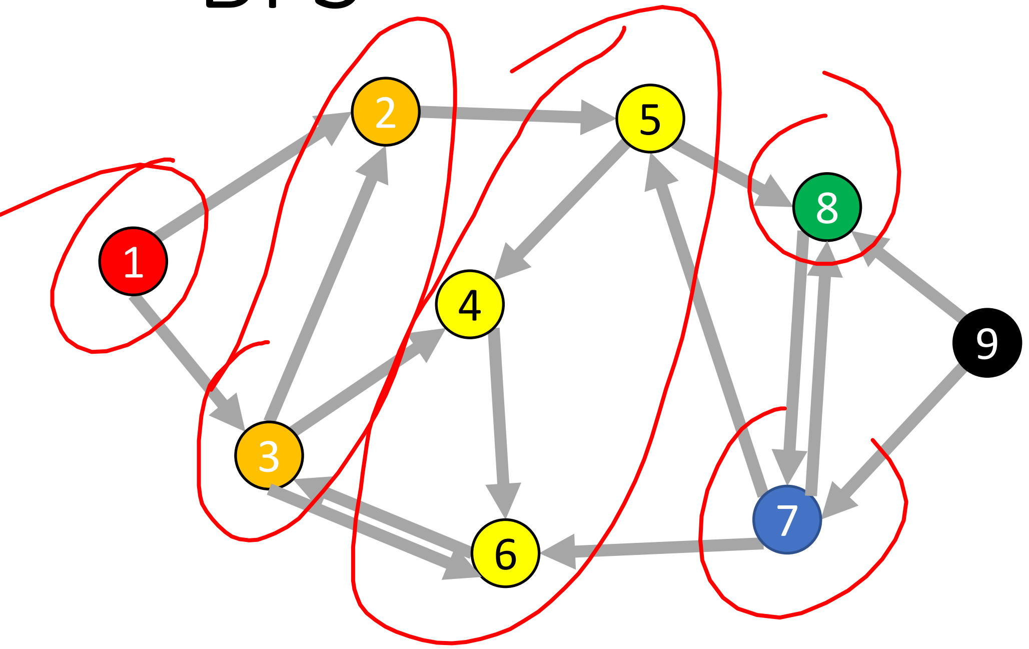
http://www.cs.uw.edu/332

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?



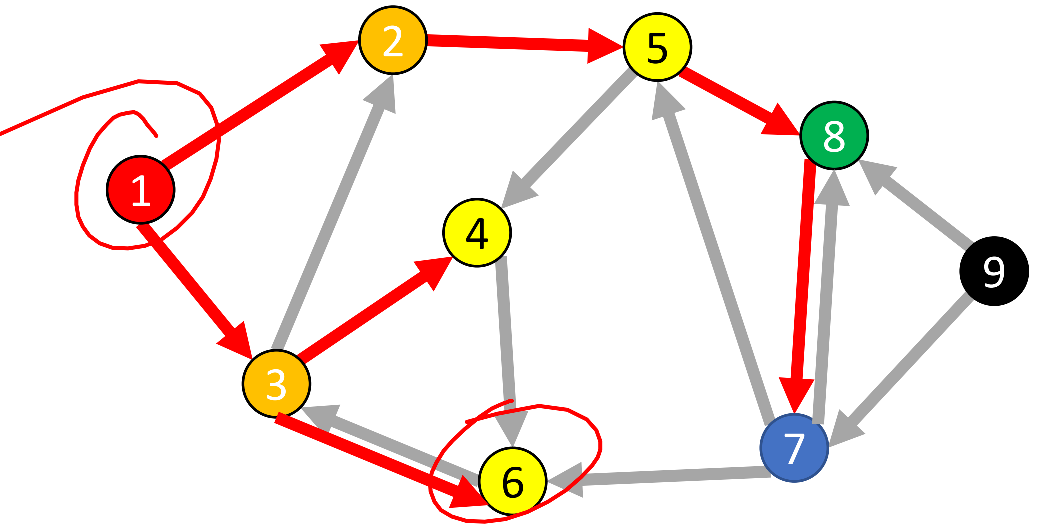
BFS



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);
                        }
                }
        }
}
```

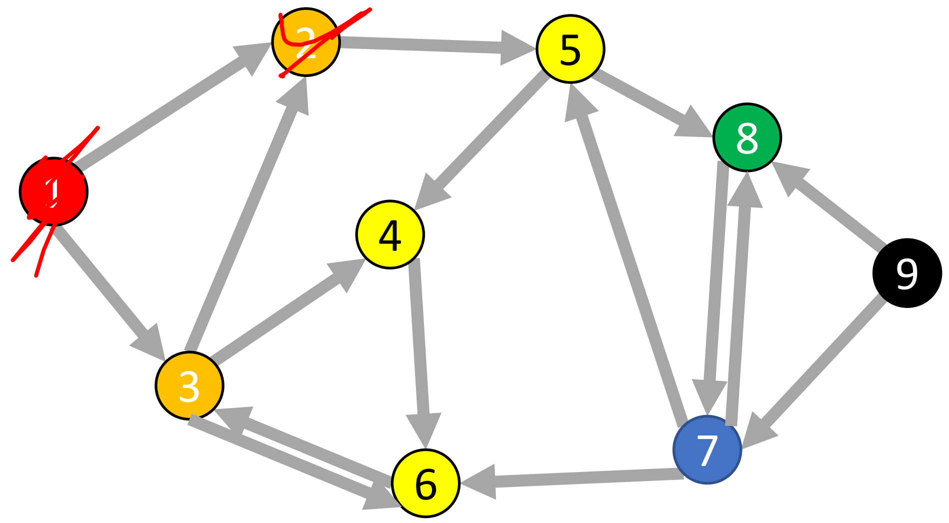
Find Distance (unweighted)



Idea: when it’s seen, remember its “layer” depth!

```
int findDistance(graph, s, t){
    found = new Queue();
    ~~layer = 0;~~
    depth of s = 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 – Worked Example



```
For each node:
    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
```

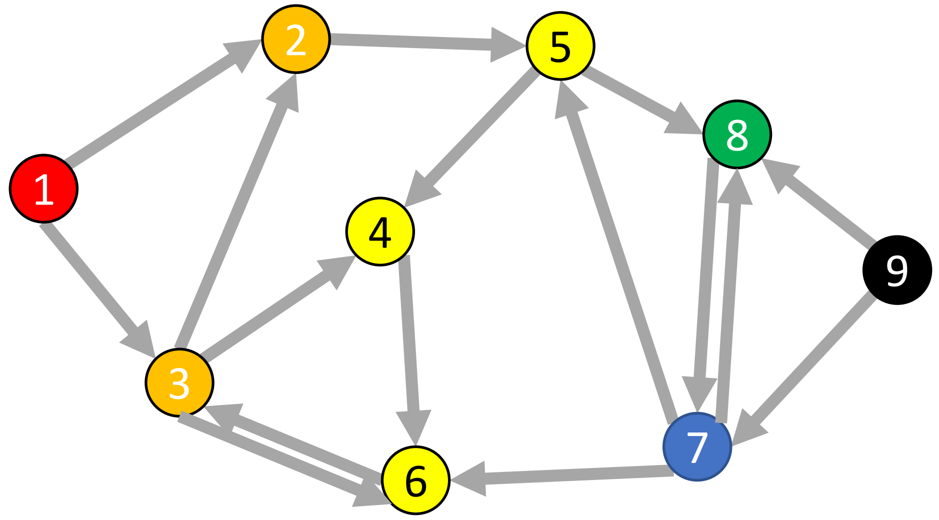
Node	Visited?	Depth
1	T	0
2	T	1
3	T	1
4	T	2
5	T	2
6	T	2
7		
8	T	3
9		

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

Shortest Path - Idea

1, 2, 5, 8

4



Node	Visited?	Previous
1	T	[illegible]
→ 2	T	1
3	T	1
4	T	3
→ 5	T	2
6	T	3
7		
→ 8	T	5
9		

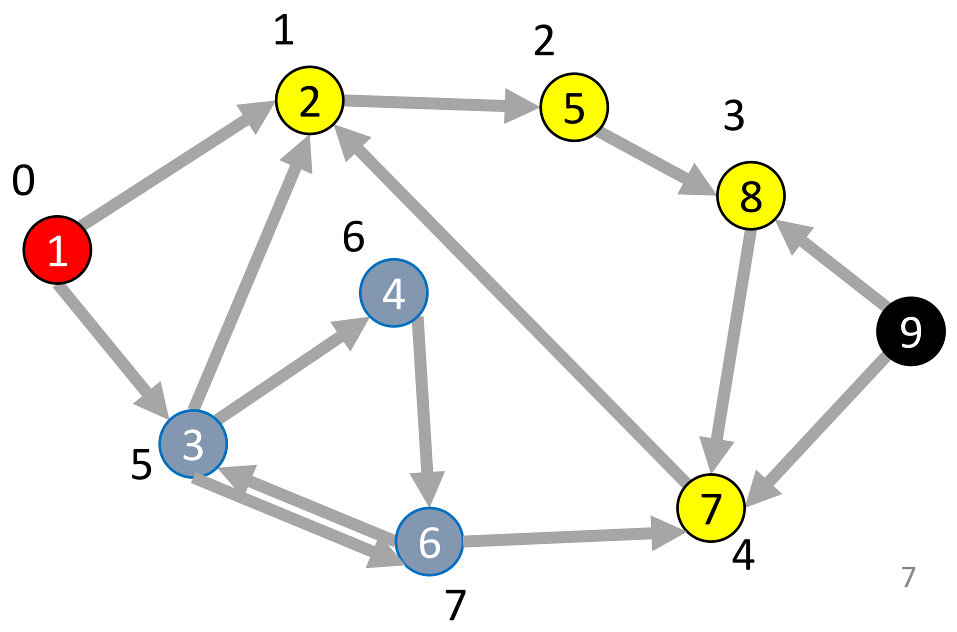
Queue: ~~1~~ ~~2~~ ~~3~~ ~~5~~ 4 6

For each node:

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

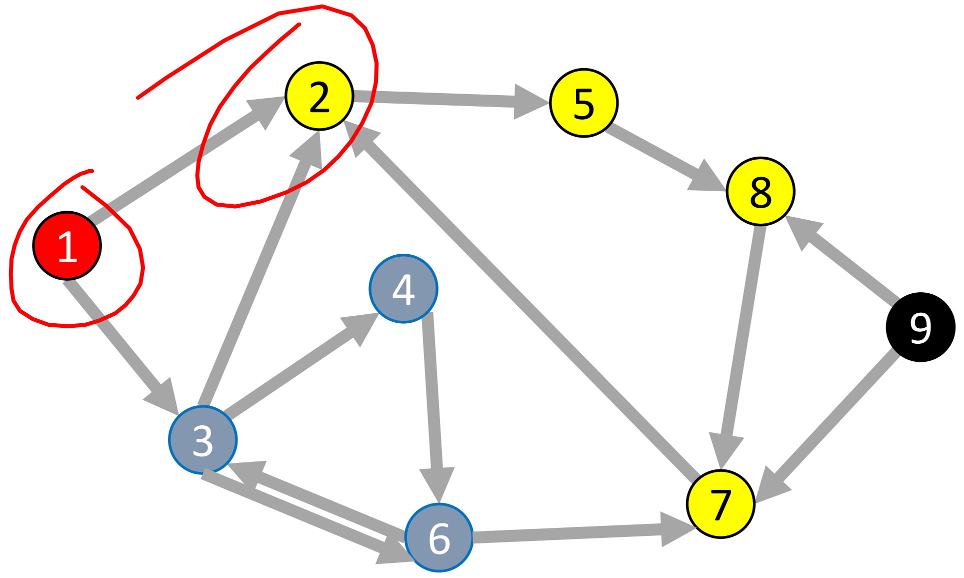
Depth-First Search

- 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.

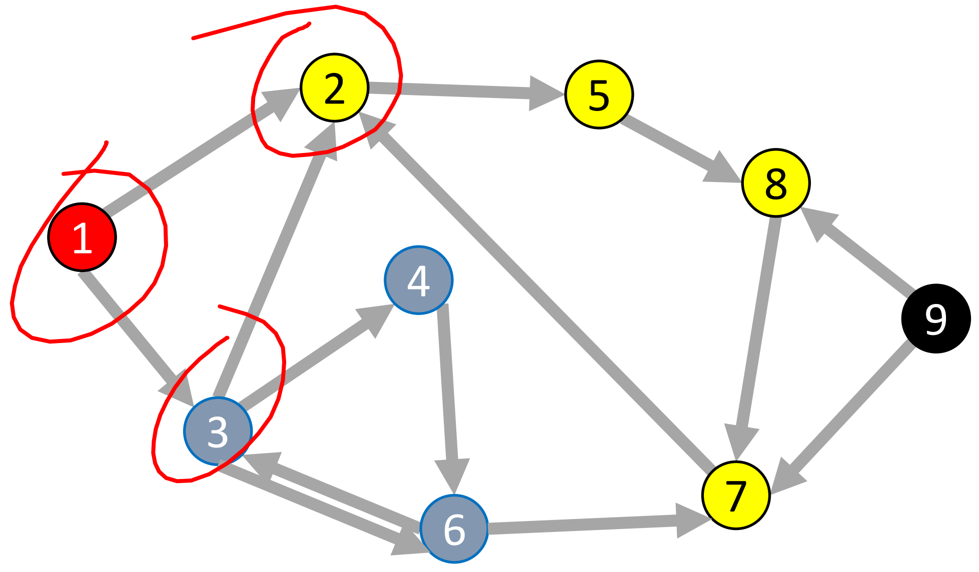


DFS Recursively (more common)

```
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



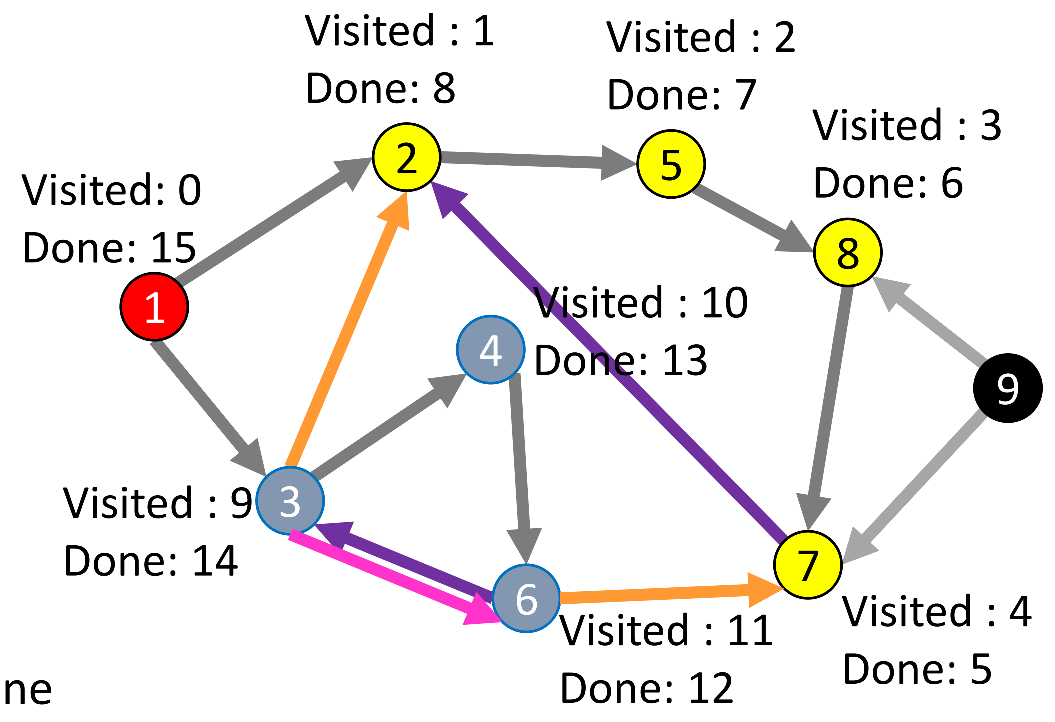
Node	Visited?	Done?	Other Info
1	T		
2	T	T	
3	T		
4			
5	T	T	
6			
7	T	T	
8	T	T	
9			

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

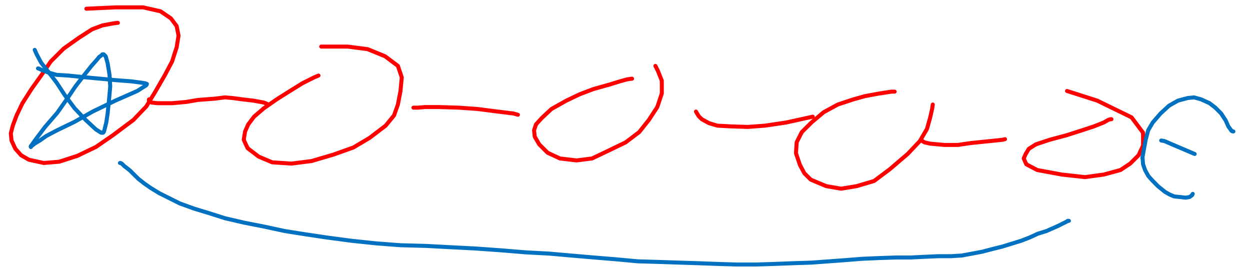
(Call) Stack: | 1

Using DFS

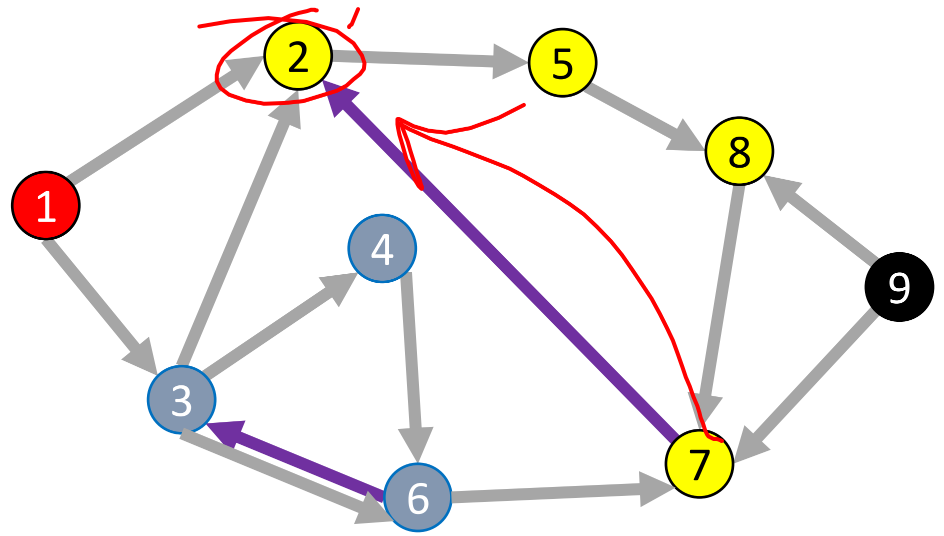
- Consider the "visited times" and "done times"
- Edges can be categorized:
  - Tree Edge
    - (a, b) was followed when pushing
    - (a, b) when b was unvisited when we were at a
  - Back Edge
    - (a, b) goes to an "ancestor"
    - a and b visited but not done when we saw (a, b)
    - $t_{visited}(b) < t_{visited}(a) < t_{done}(a) < t_{done}(b)$
  - 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

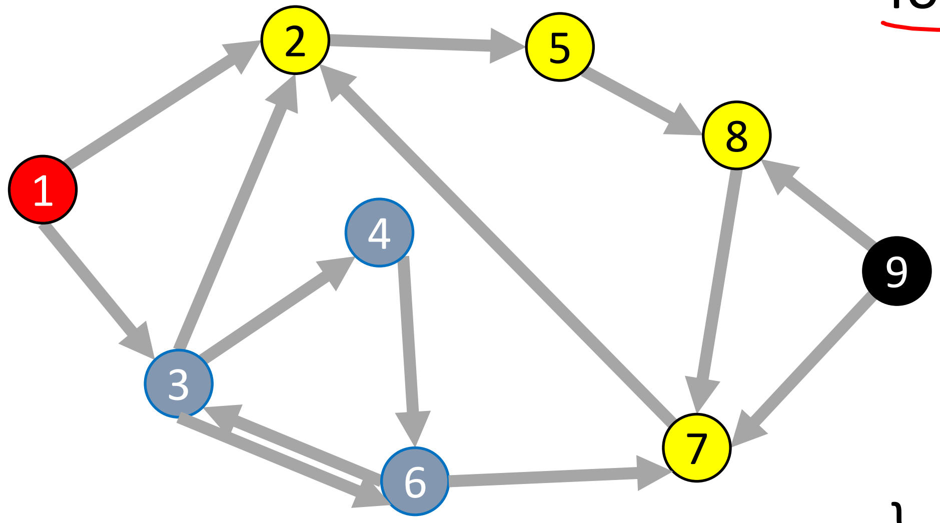


- Behavior of DFS:
  - "Visit everything reachable from the current node before going back"
- Back Edge:
  - The current node's neighbor is an "in progress" node
  - Since that other node is "in progress", the current node is reachable from it
  - The back edge is a path to that other node
  - **Cycle!**



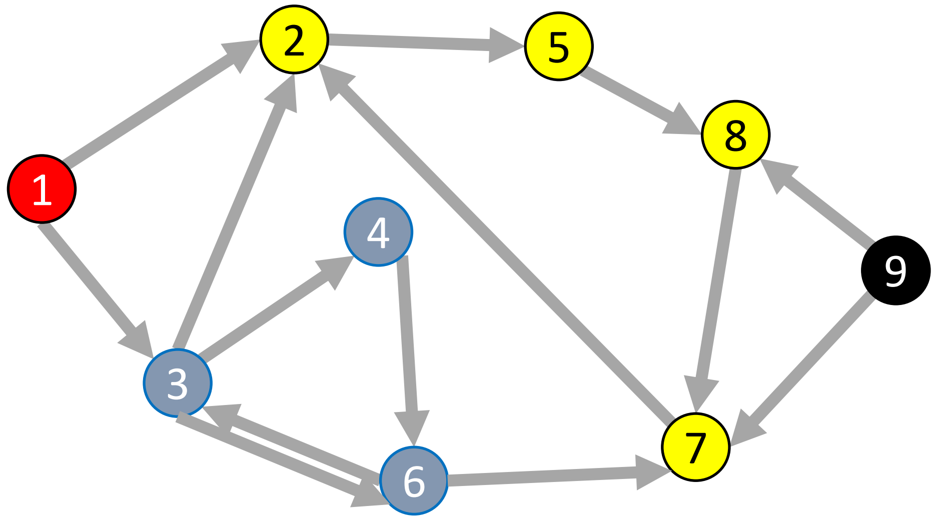
Cycle Detection

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

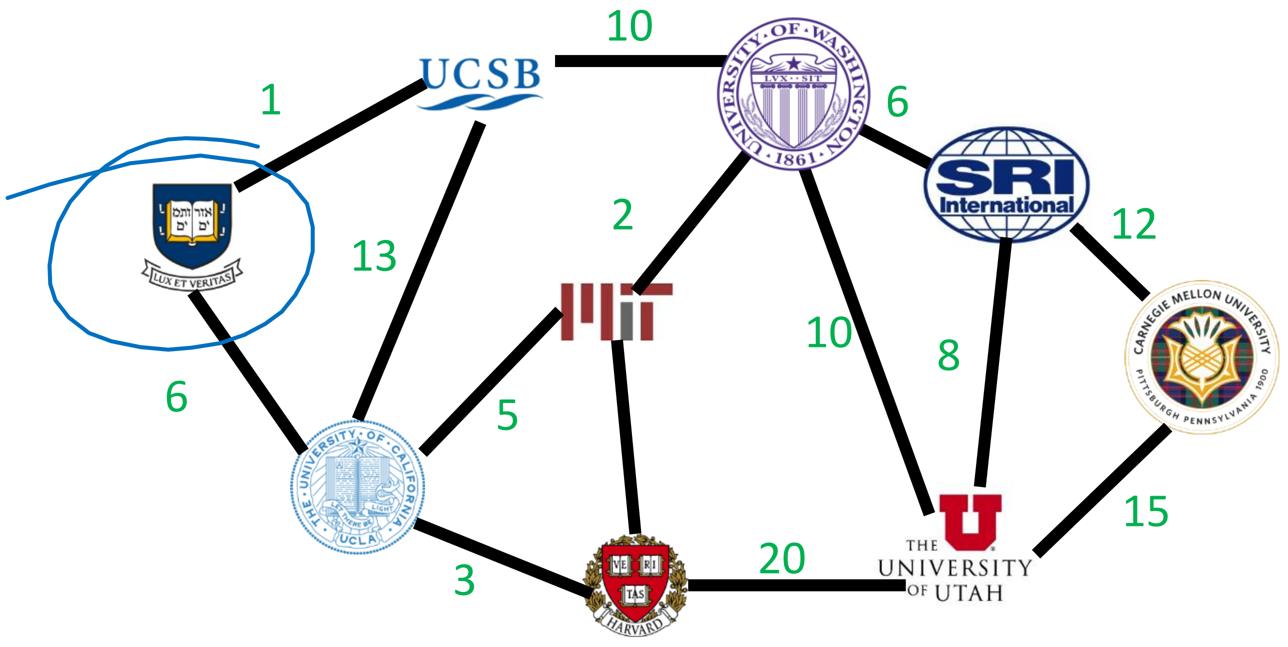


```
Starting from the current node:
    for each non-done neighbor:
        if the neighbor is visited:
            we found a cycle!
        else:
            mark the neighbor as visited
            do a DFS from the neighbor
    mark the current node as done
```

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

(Call) Stack:

Single-Source Shortest Path



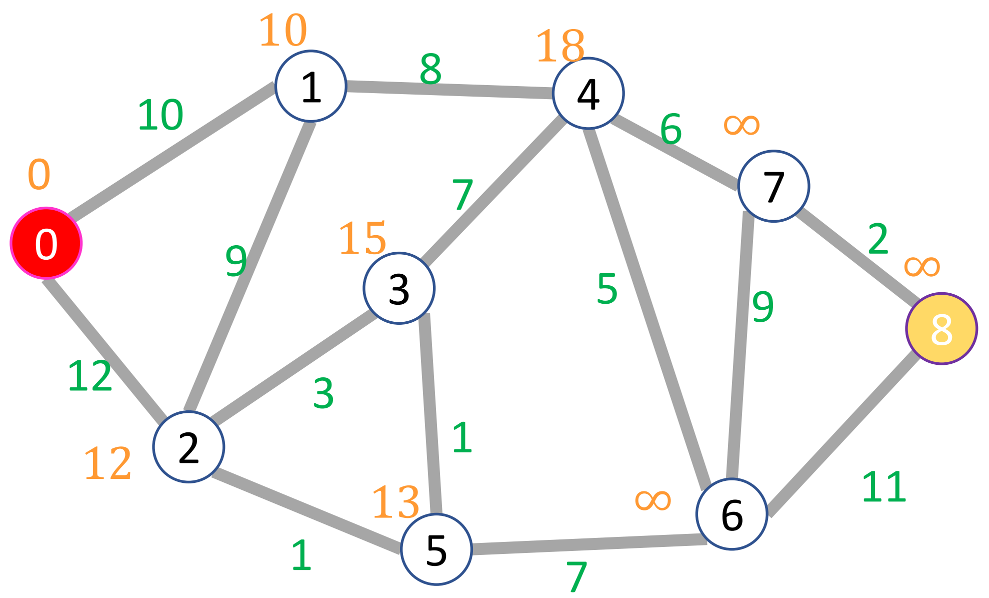
Find the quickest way to get from UVA to each of these other places

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, 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 seen
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

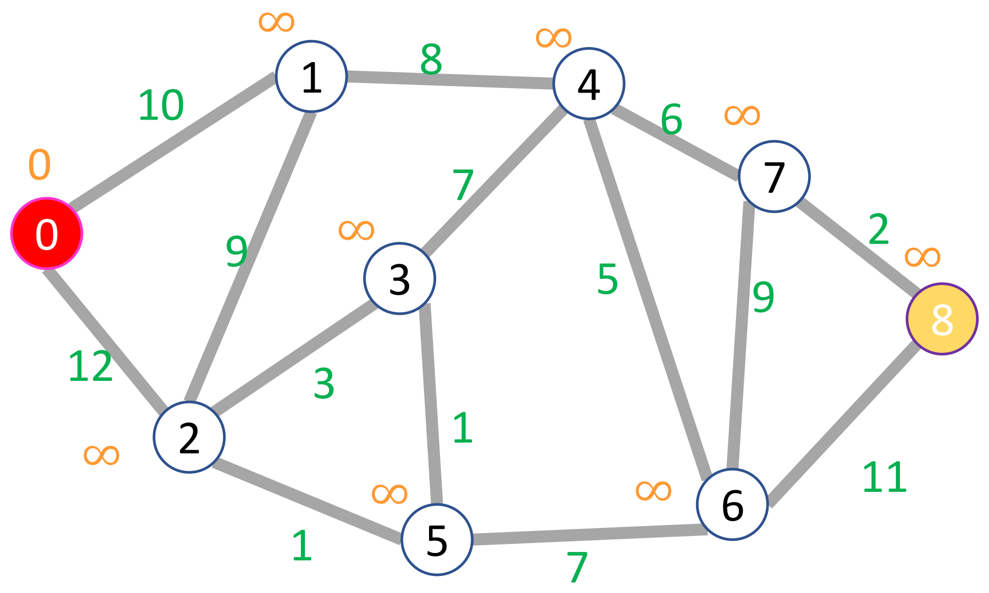
Start: 0

End: 8

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

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 seen
for each not-done neighbor:
    Update its distance if we found a better path
```



Dijkstra’s Algorithm

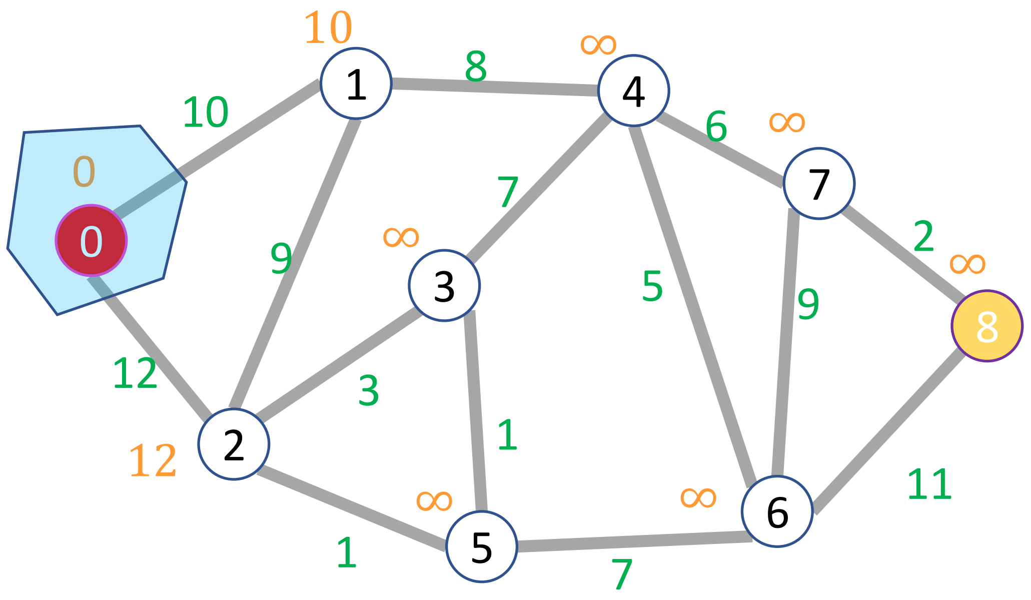
Start: 0

End: 8

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

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 seen
for each not-done neighbor:
        Update its distance if we found a better path
```



Dijkstra’s Algorithm

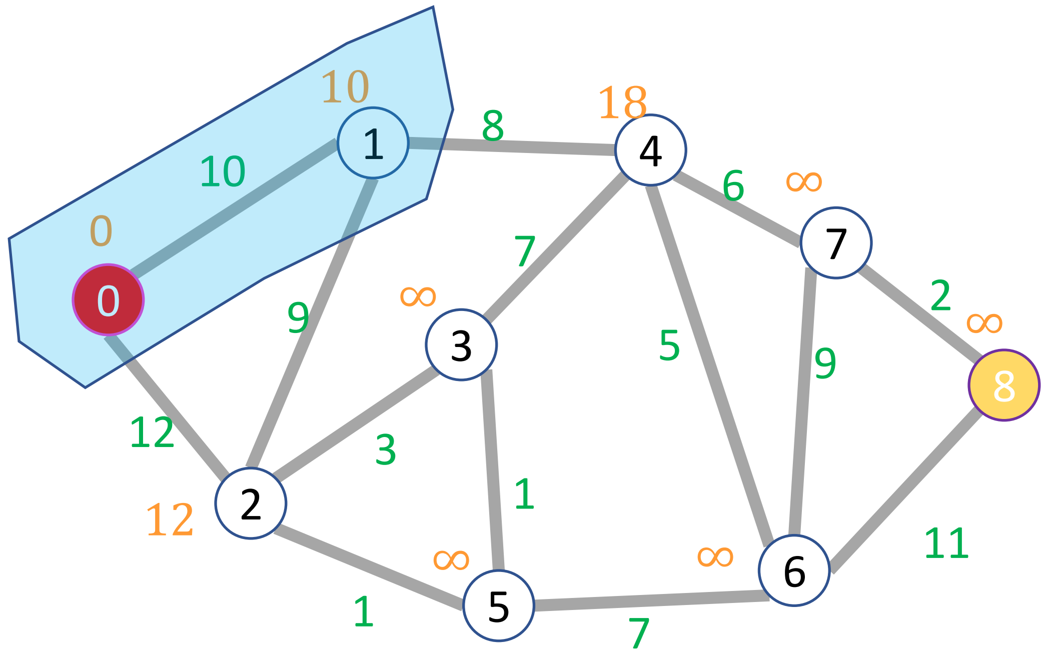
Start: 0

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 seen
for each not-done neighbor:
    Update its distance if we found a better path

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



Dijkstra’s Algorithm

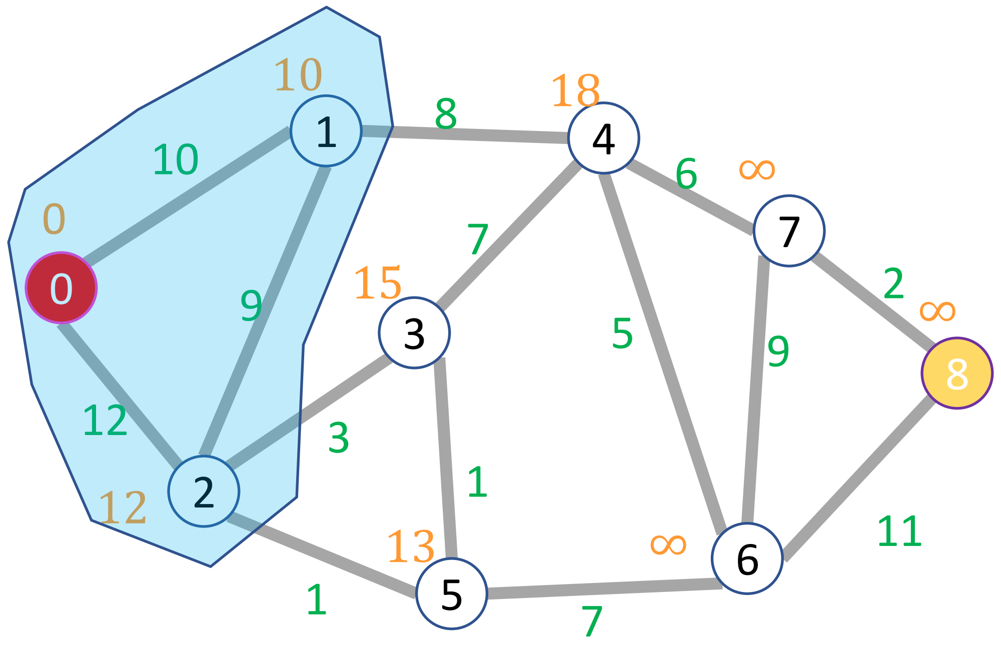
Start: 0

End: 8

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

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 seen
for each not-done neighbor:
    Update its distance if we found a better path



Dijkstra's Algorithm

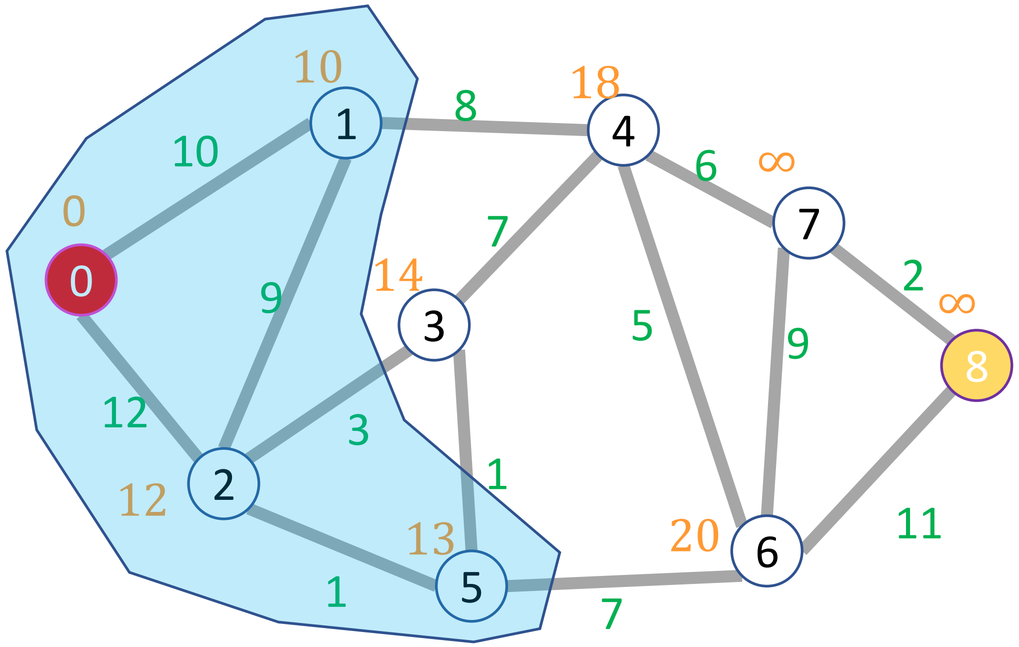
Start: 0

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 seen
for each not-done neighbor:
    Update its distance if we found a better path

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



Dijkstra's Algorithm

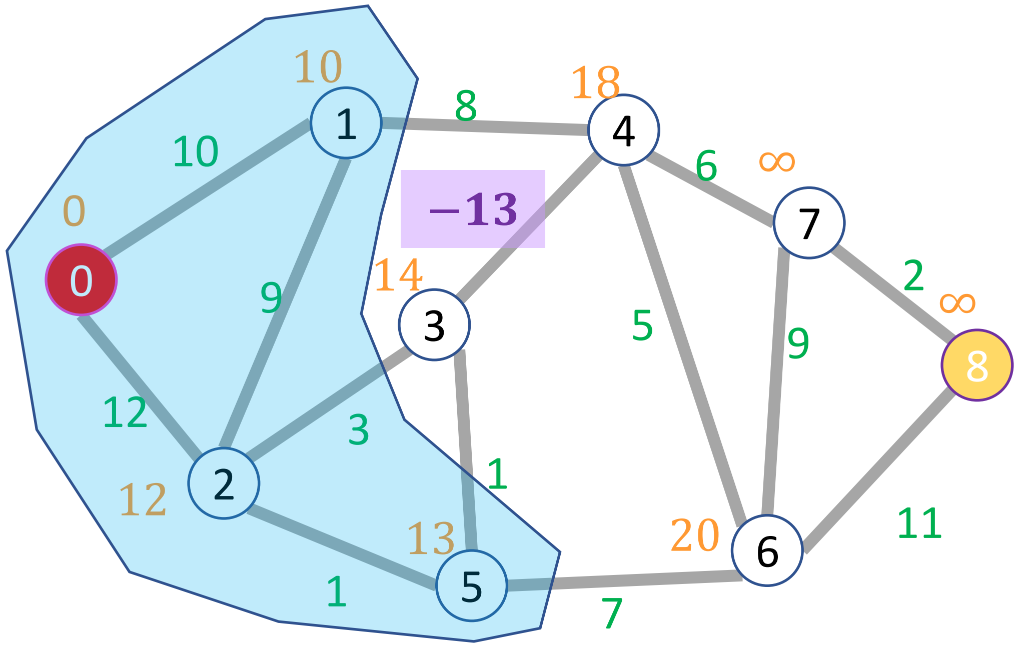
Start: 0

End: 8

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

What if we had a negative-weight edge?

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



Dijkstra’s Algorithm

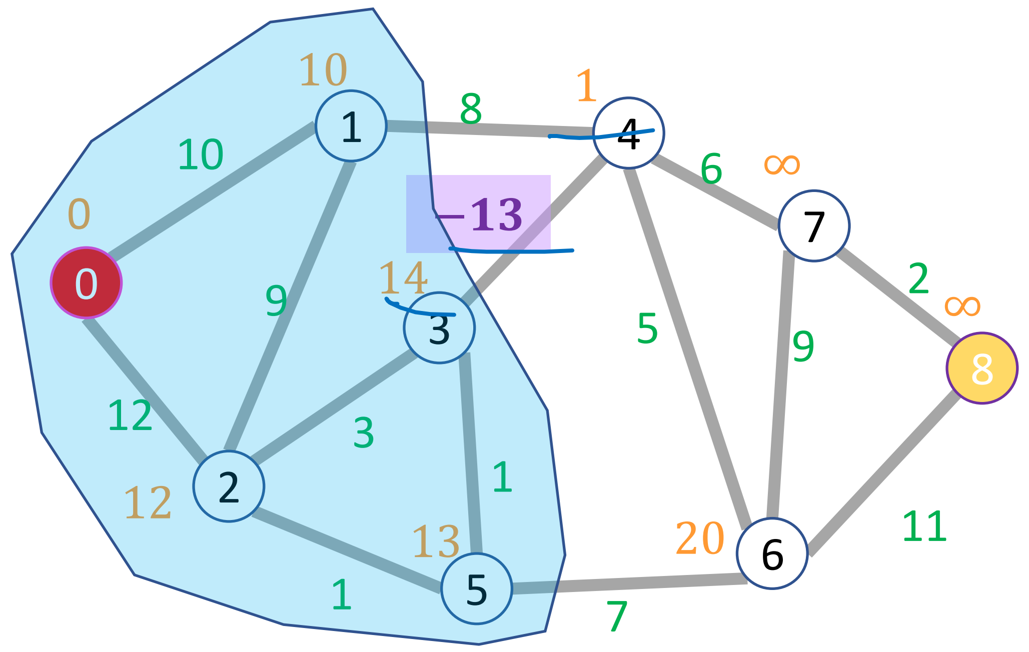
Start: 0

End: 8

What if we had a negative-weight edge?

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

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



Dijkstra’s Algorithm

Start: 0

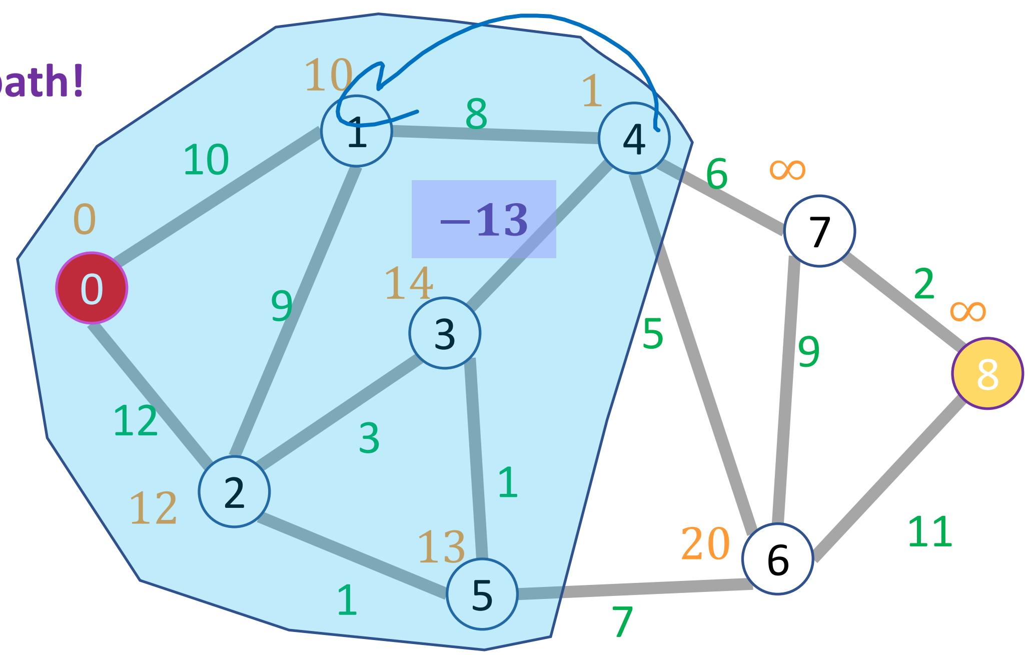
End: 8

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

There’s a better path!

What if we had a negative-weight edge?

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



Dijkstra’s Algorithm

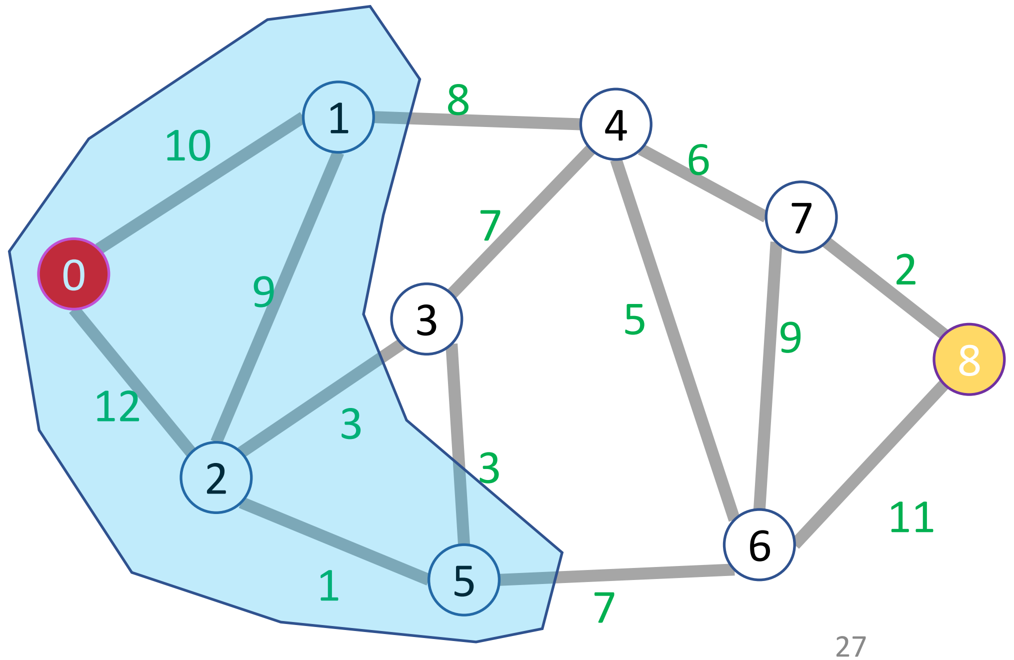
```
int dijkstras(graph, start, end){
    distances = [∞, ∞, ∞,…]; // one index per node
    seen = [False,False,False,…]; // one index per node
    done = [False,False,False,…]; // one index per node
    PQ = new MinHeap();
    PQ.insert(0, start); // priority=0, value=start
    distances[start] = 0;
    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:
  - $\Theta(|E| \log|V|)$

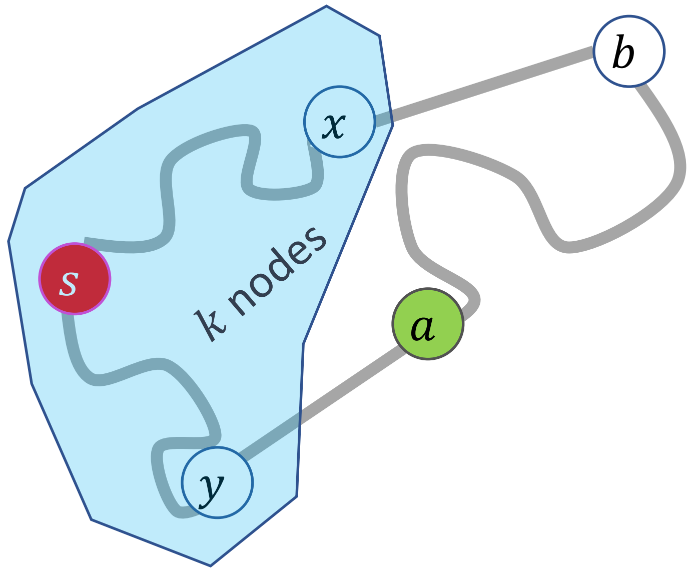
Dijkstra’s Algorithm: Correctness

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



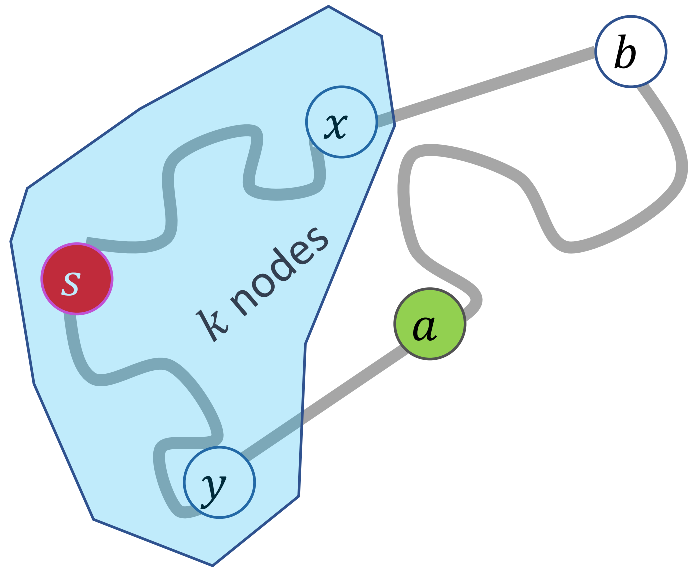
Dijkstra’s Algorithm: Correctness

- 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



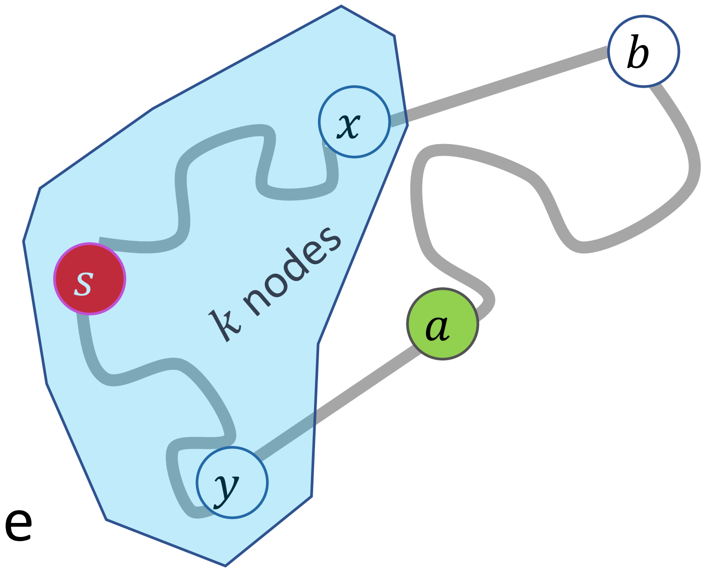
Dijkstra’s Algorithm: Correctness

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



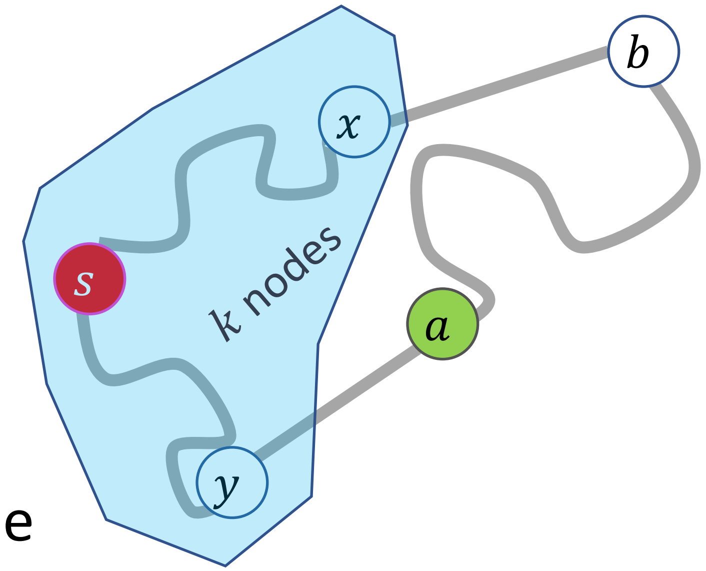
Dijkstra's Algorithm: Correctness

- 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!


b
x
s
k nodes
a
y