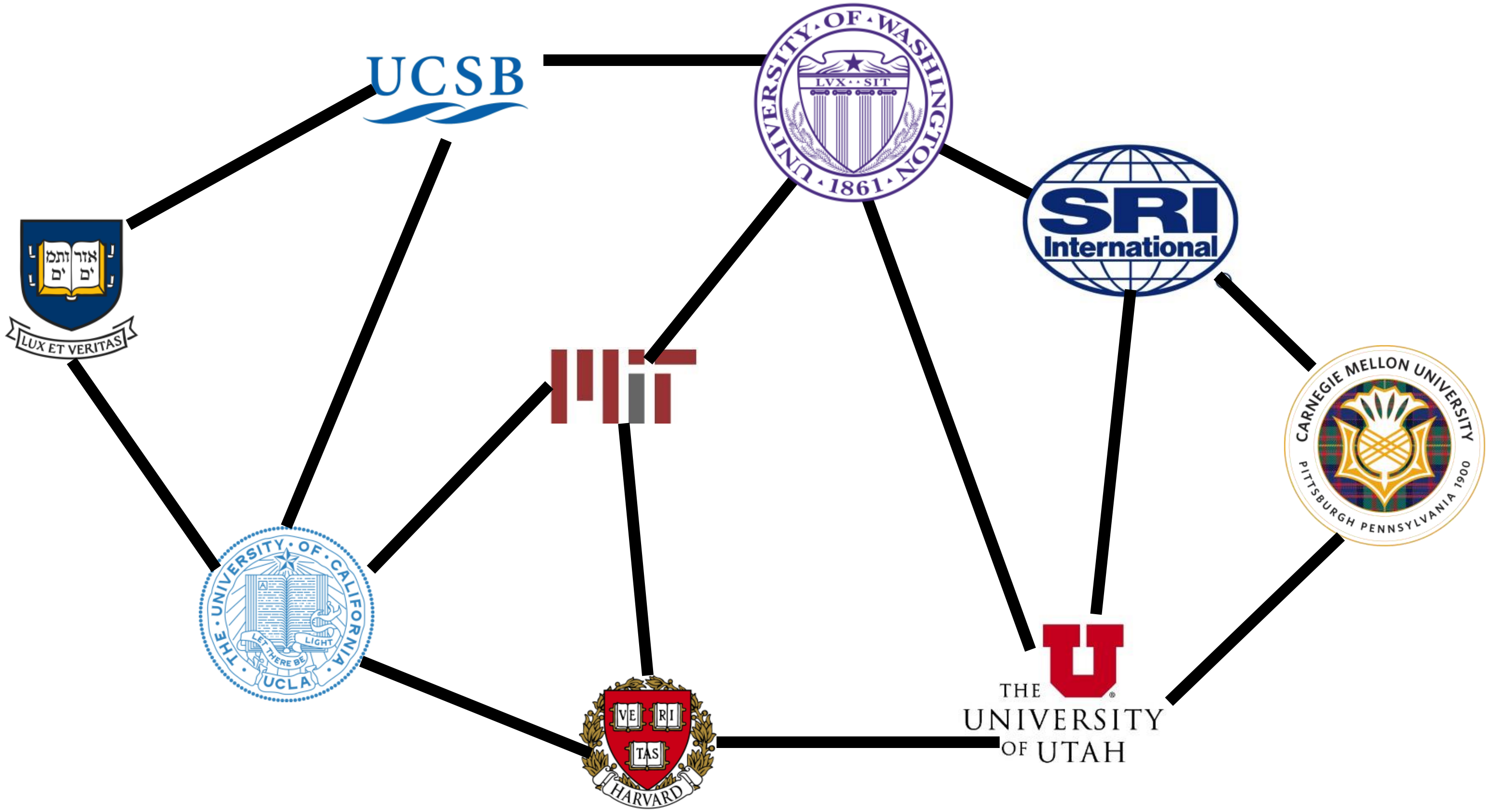
CSE 332 Summer 2024
Lecture 14: Graphs

Nathan Brunelle

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

ARPANET



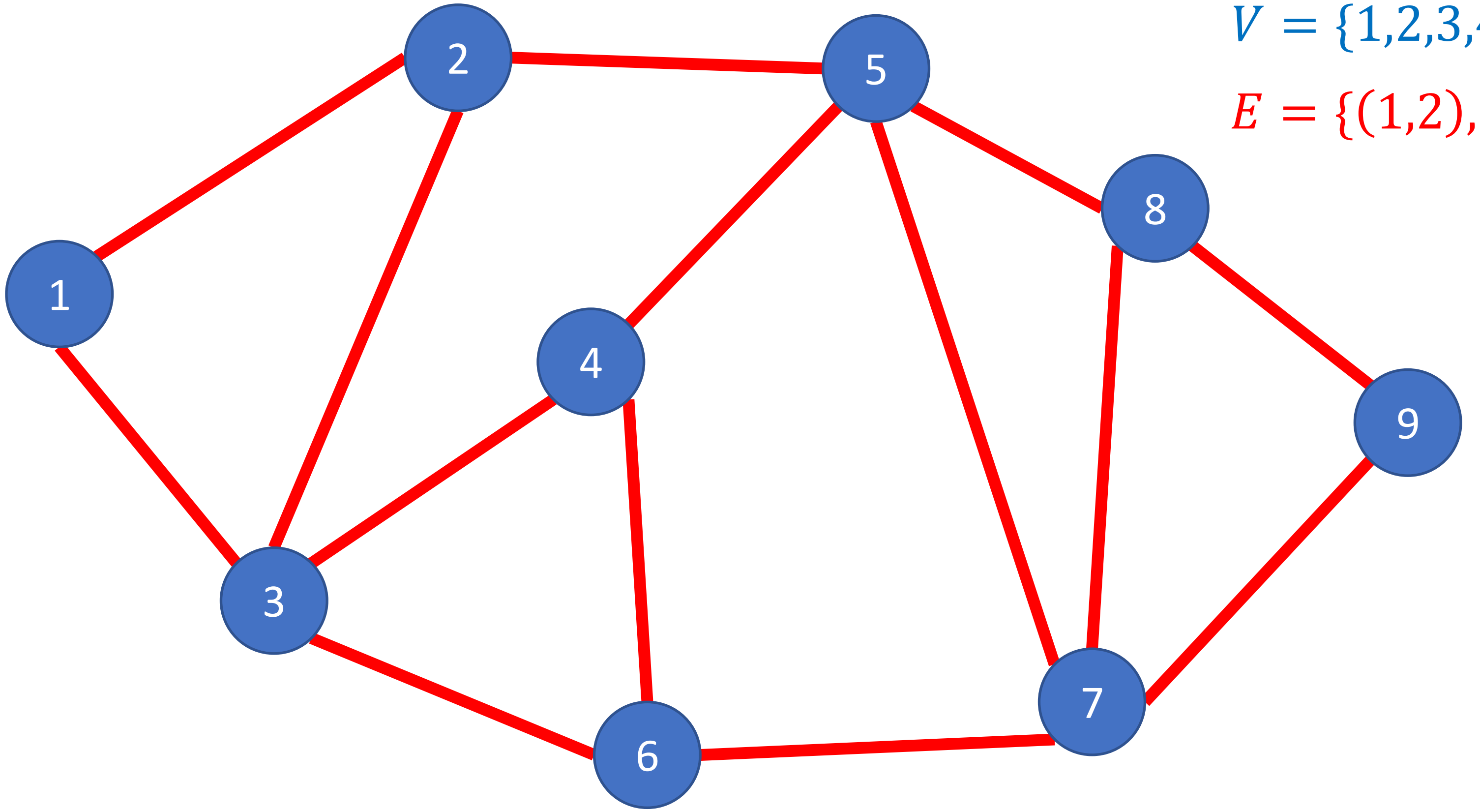
UCSB
UNIVERSITY OF WASHINGTON
LVX SIT
1861
SRI
International
LUX ET VERITAS
THE UNIVERSITY OF CALIFORNIA
LET THERE BE LIGHT
UCLA
VE RI TAS
HARVARD
THE UNIVERSITY OF UTAH
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PENNSYLVANIA 1900

Undirected Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges



$V = \{1,2,3,4,5,6,7,8,9\}$

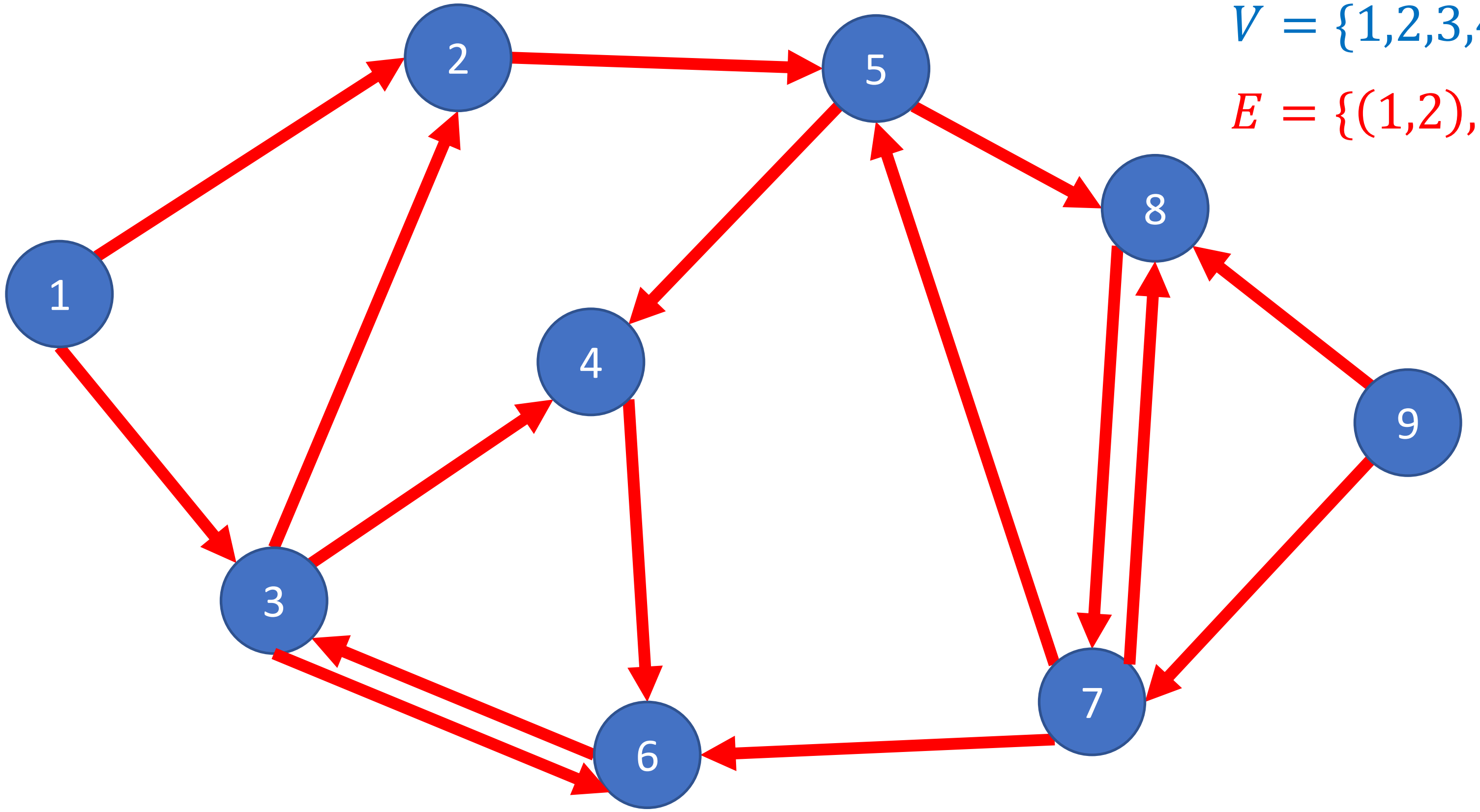
$E = \{(1,2), (2,3), (1,3), \dots\}$

Directed Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges

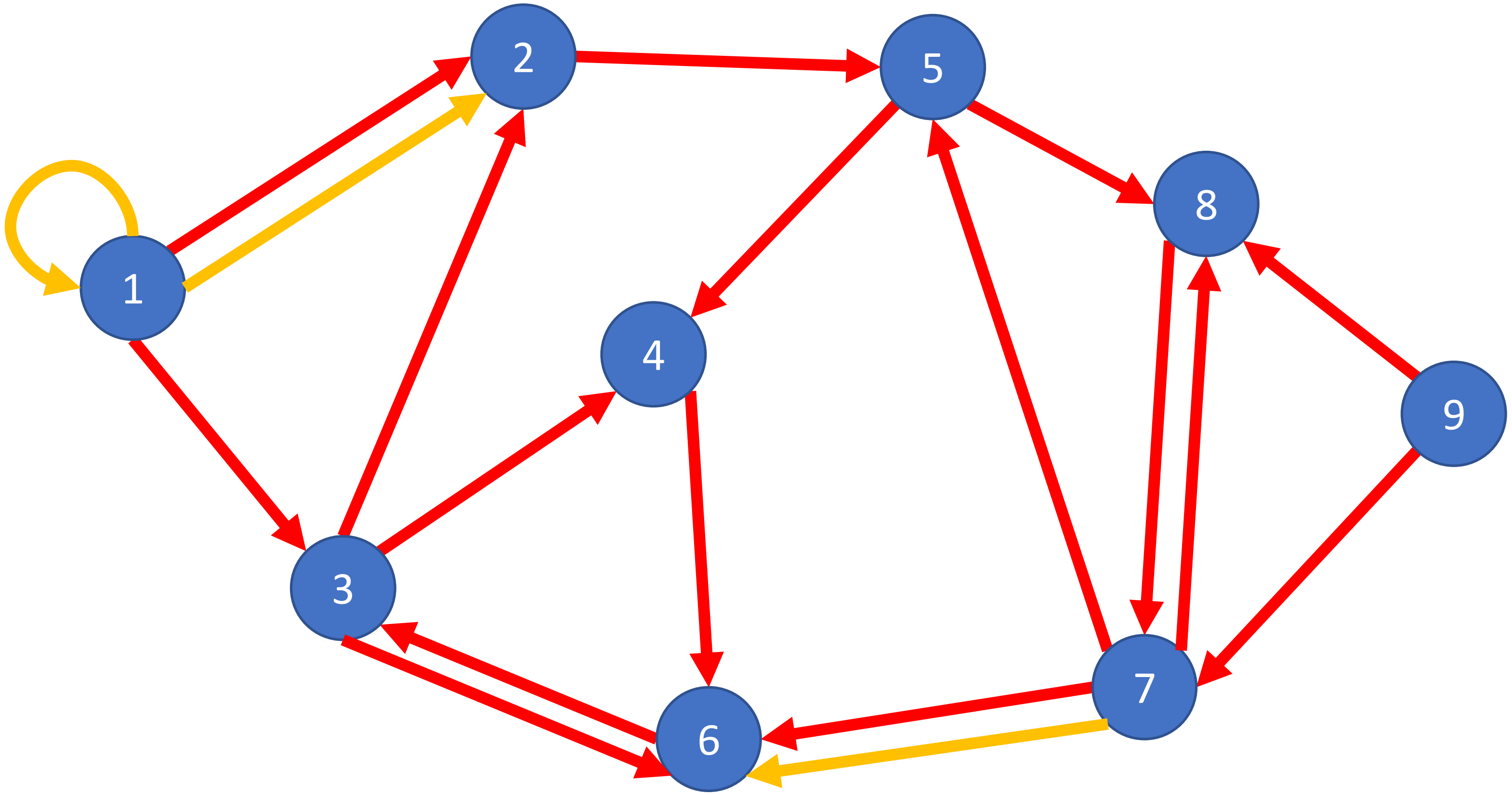


$V = \{1,2,3,4,5,6,7,8,9\}$

$E = \{(1,2), (2,3), (1,3), \ldots\}$

Self-Edges and Duplicate Edges

Some graphs may have duplicate edges (e.g. here we have the edge (1,2) twice).
Some may also have self-edges (e.g. here there is an edge from 1 to 1).
Graph with Neither self-edges nor duplicate edges are called **simple graphs**



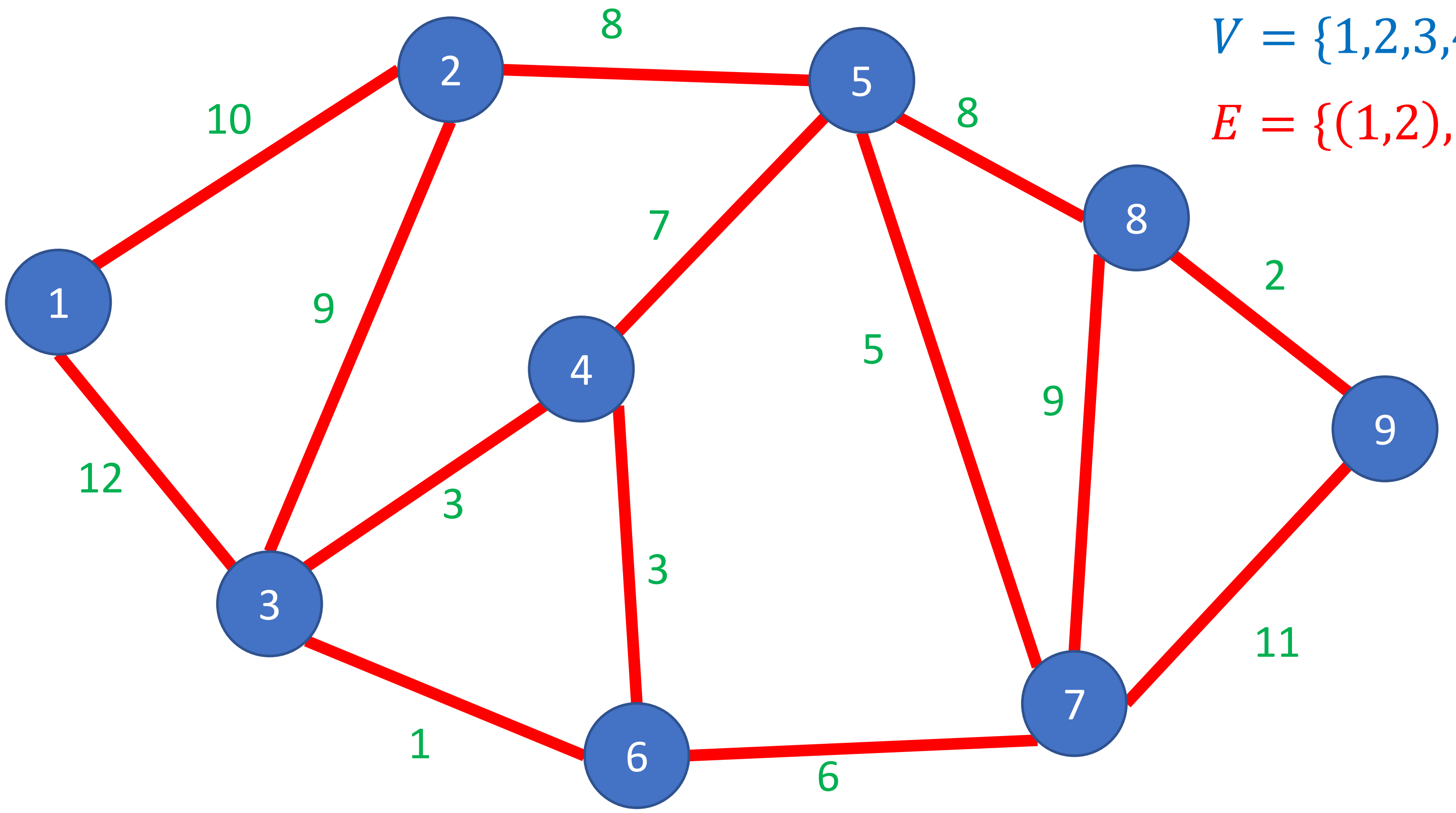
Weighted Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges

$w(e)$ = weight of edge e



$V = \{1,2,3,4,5,6,7,8,9\}$

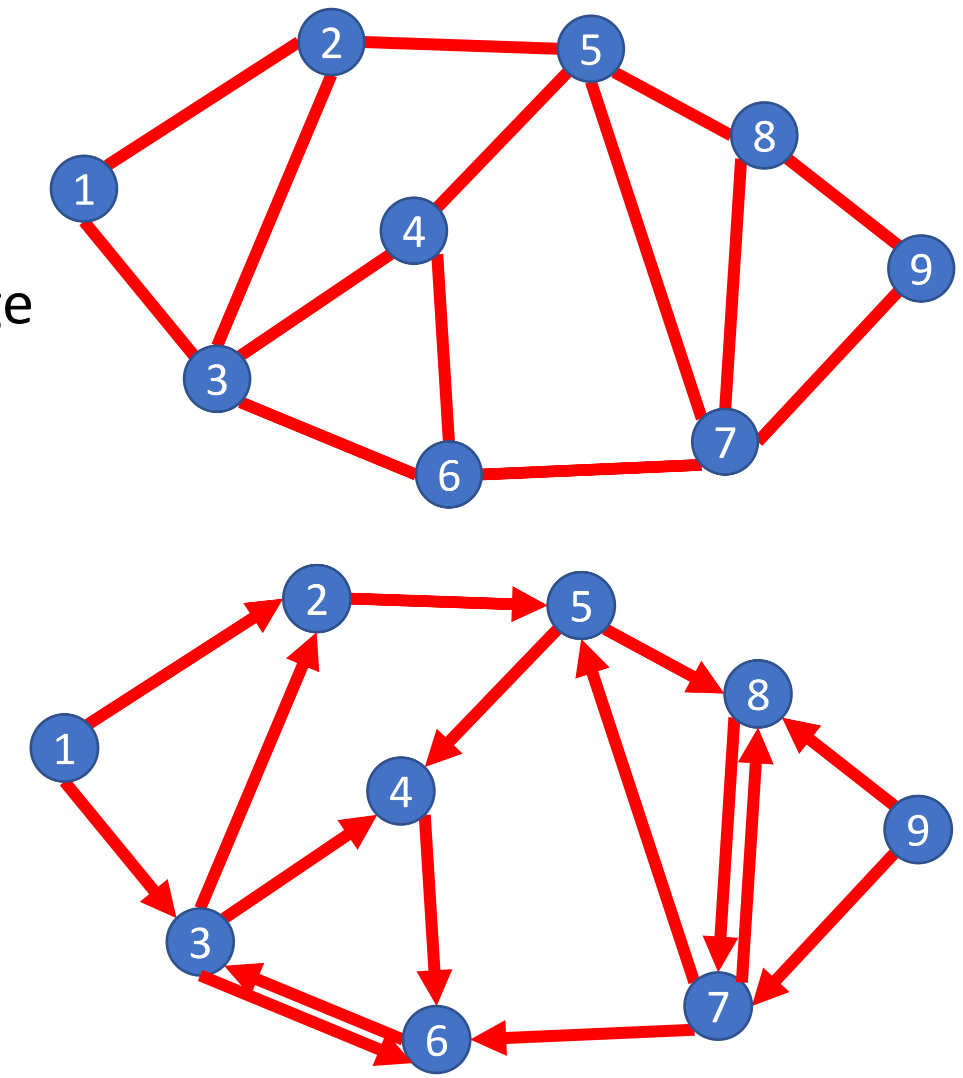
$E = \{(1,2), (2,3), (1,3), \ldots\}$

Graph Applications

- For each application below, consider:
  - What are the nodes, what are the edges?
  - Is the graph directed?
  - Is the graph simple?
  - Is the graph weighted?
- Facebook friends
- Twitter followers
- Java inheritance
- Airline Routes

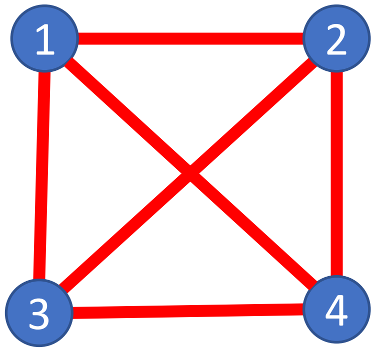
Some Graph Terms

- Adjacent/Neighbors
  - Nodes are adjacent/neighbors if they share an edge
- Degree
  - Number of edges “touching” a vertex
- Indegree
  - Number of incoming edges
- Outdegree
  - Number of outgoing edges

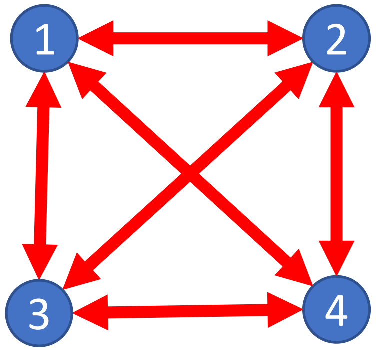


Definition: Complete Graph

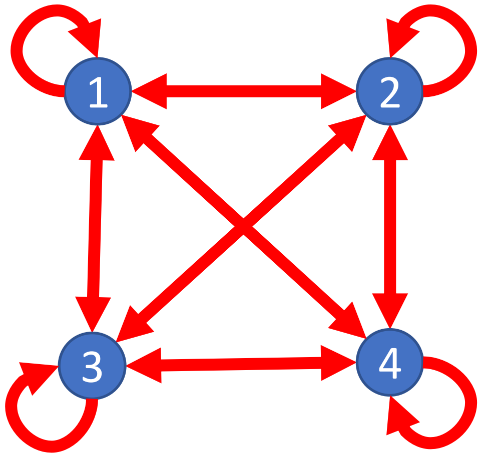
A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is an edge from v_1 to v_2



Complete
Undirected Graph



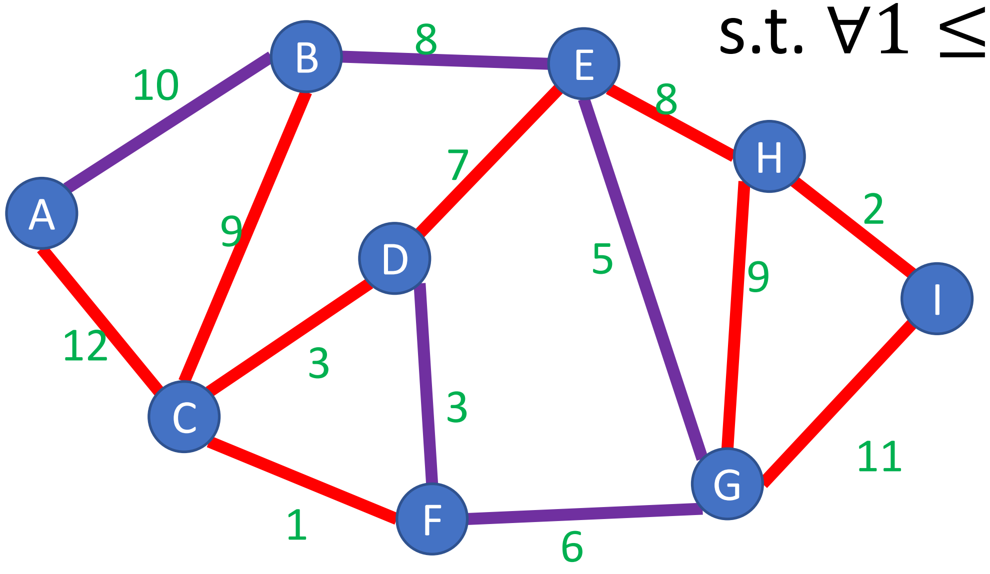
Complete
Directed Graph



Complete Directed
Non-simple Graph

Definition: Path

A sequence of nodes $(v_1, v_2, \dots, v_k)$
s.t. $\forall 1 \leq i \leq k-1,\ (v_i, v_{i+1}) \in E$

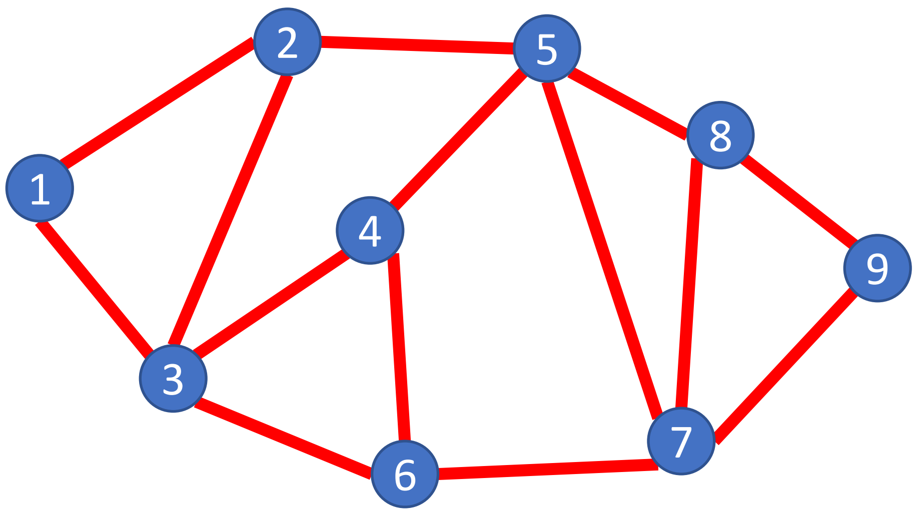


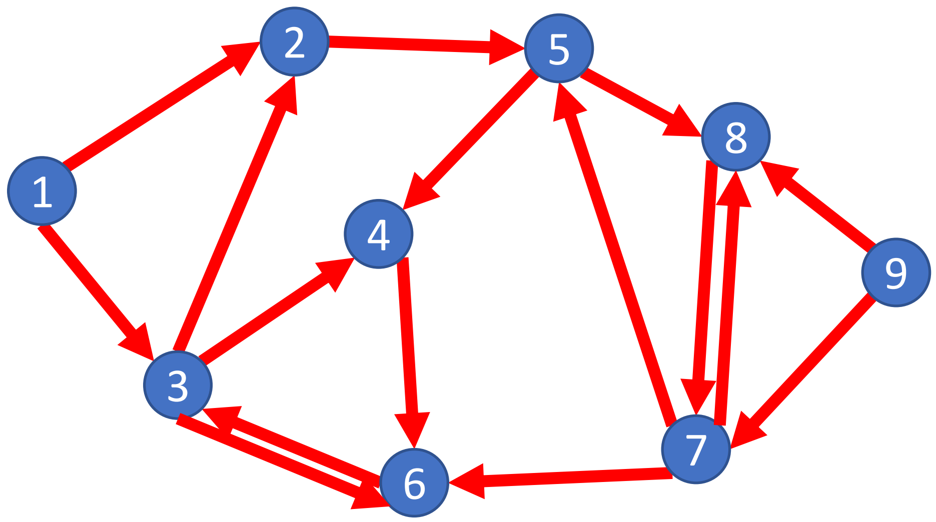
<u>Simple Path:</u>
A path in which each node appears at most once

<u>Cycle:</u>
A path which starts and ends in the same place

Definition: (Strongly) Connected Graph

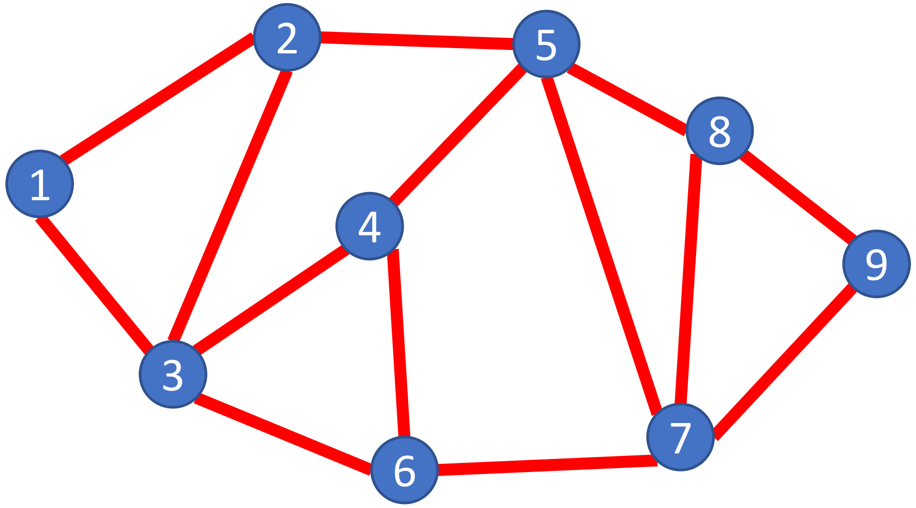
A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from v_1 to v_2



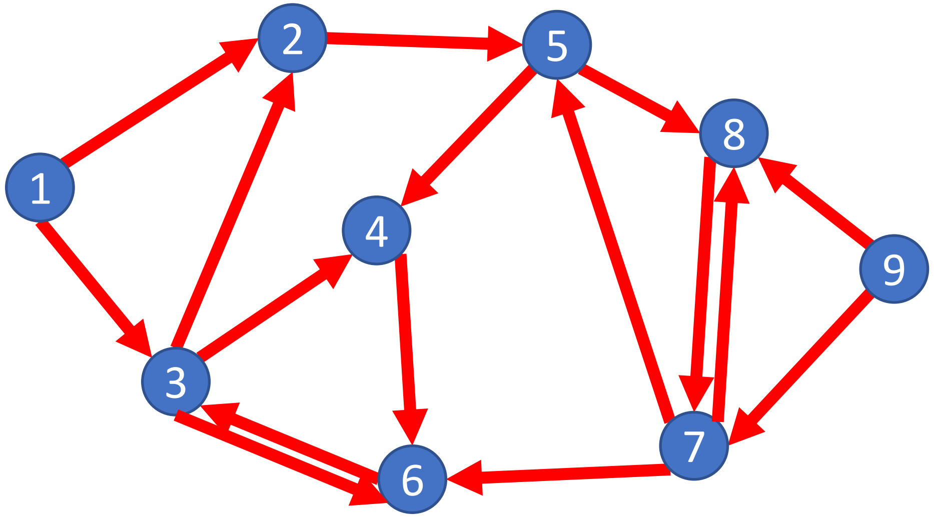


Definition: (Strongly) Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from v_1 to v_2



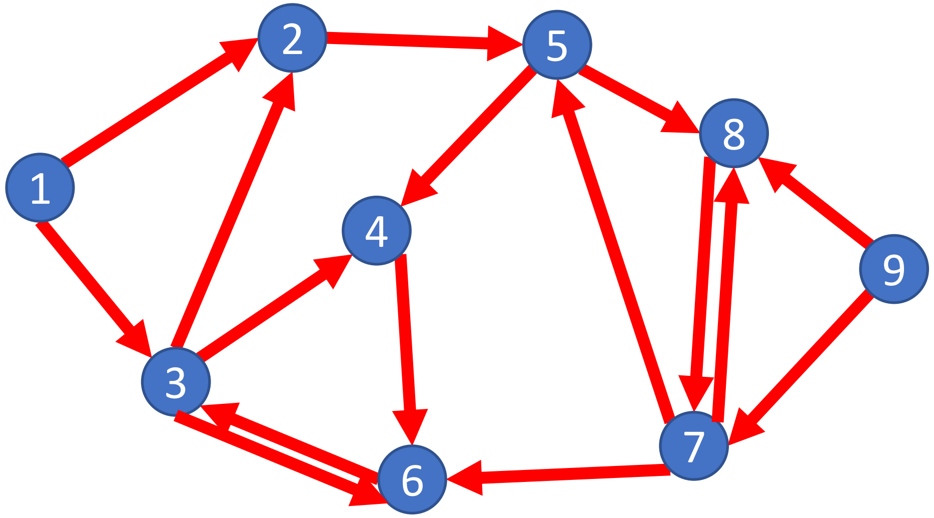
Connected



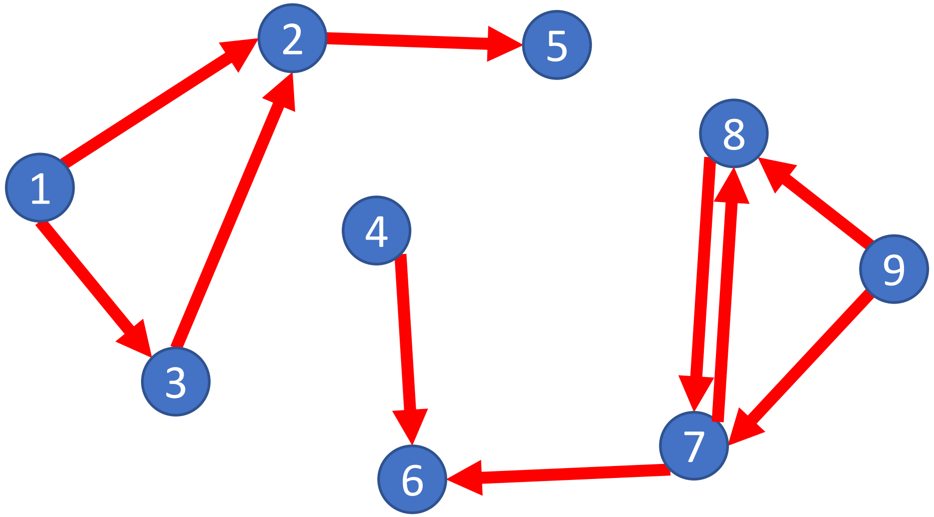
Not (strongly) Connected

Definition: Weakly Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from v_1 to v_2 ignoring direction of edges



Weakly Connected



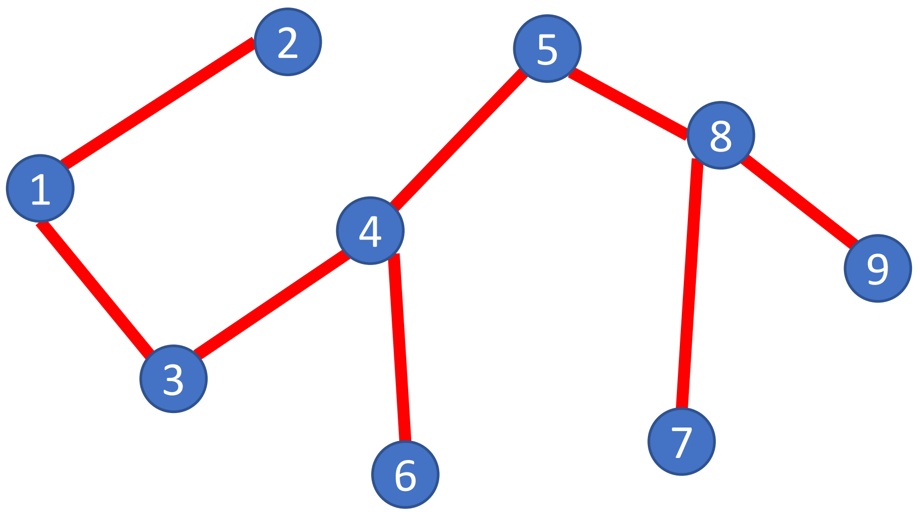
Not Weakly Connected

Graph Density, Data Structures, Efficiency

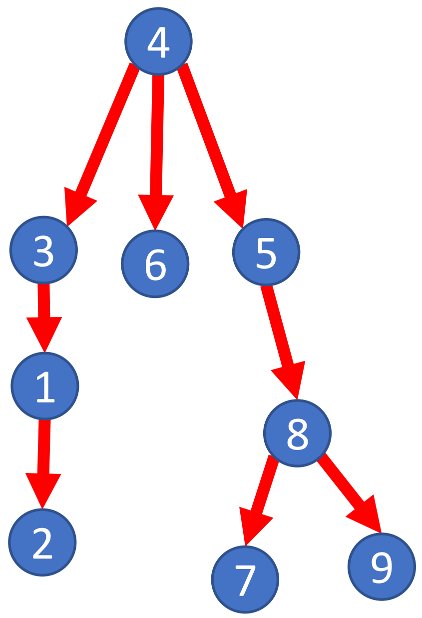
- The maximum number of edges in a graph is $\Theta(|V|^2)$:
  - Undirected and simple: $\frac{|V|(|V|-1)}{2}$
  - Directed and simple: $|V|(|V| - 1)$
  - Direct and non-simple (but no duplicates): $|V|^2$
- If the graph is connected, the minimum number of edges is $|V| - 1$
- If $|E| \in \Theta(|V|^2)$ we say the graph is **dense**
- If $|E| \in \Theta(|V|)$ we say the graph is **sparse**
- Because $|E|$ is not always near to $|V|^2$ we do not typically substitute $|V|^2$ for $|E|$ in running times, but leave it as a separate variable
  - However, $\log(|E|) \in \Theta(\log(|V|))$

Definition: Tree

A Graph $G = (V, E)$ is a tree if it is undirect, connected, and has no cycles (i.e. is acyclic). Often one node is identified as the "root"



A Tree

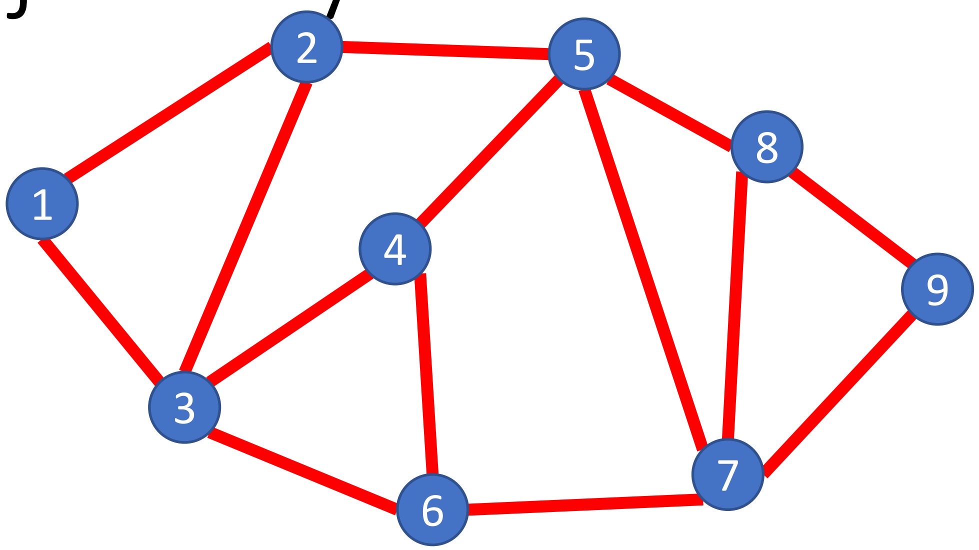


A Rooted Tree

Graph Operations

- To represent a Graph (i.e. build a data structure) we need:
  - Add Edge
  - Remove Edge
  - Check if Edge Exists
  - Get Neighbors (incoming)
  - Get Neighbors (outgoing)

Adjacency List



1	2	3		
2	1	3	5	
3	1	2	4	6
4	3	5	6	
5	2	4	7	8
6	3	4	7	
7	5	6	8	9
8	5	7	9	
9	7	8		

<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(n+m)$

Add Edge (v,w): $\Theta(\deg(v))$

Remove Edge (v,w): $\Theta(\deg(v))$

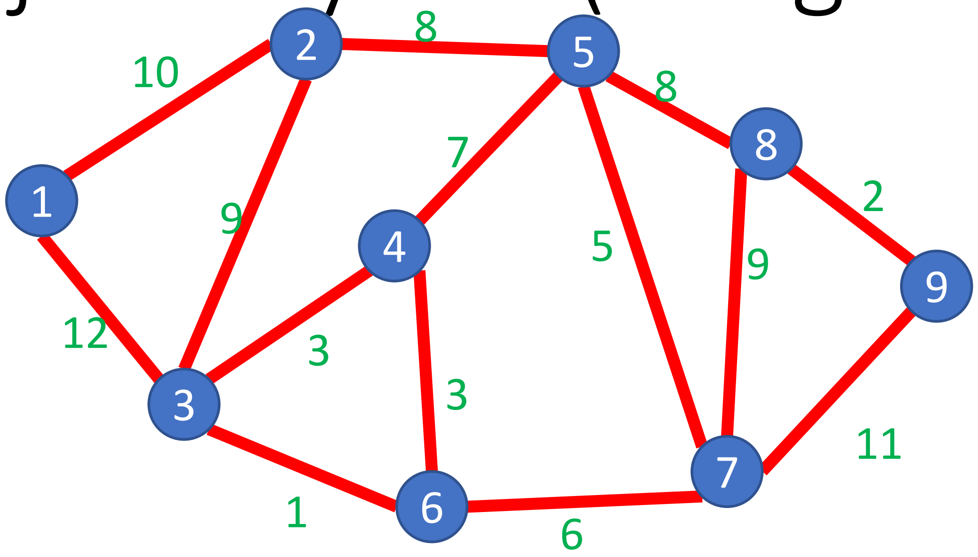
Check if Edge (v,w) Exists: $\Theta(\deg(v))$

Get Neighbors (incoming): $\Theta(n+m)$

Get Neighbors (outgoing): $\Theta(\deg(v))$

$|V| = n$
$|E| = m$

Adjacency List (Weighted)



1	2	3		
2	1	3	5	
3	1	2	4	6
4	3	5	6	
5	2	4	7	8
6	3	4	7	
7	5	6	8	9
8	5	7	9	
9	7	8		

Time/Space Tradeoffs

Space to represent: $\Theta(n+m)$

Add Edge (v,w): $\Theta(\deg(v))$

Remove Edge (v,w): $\Theta(\deg(v))$

Check if Edge (v,w) Exists: $\Theta(\deg(v))$

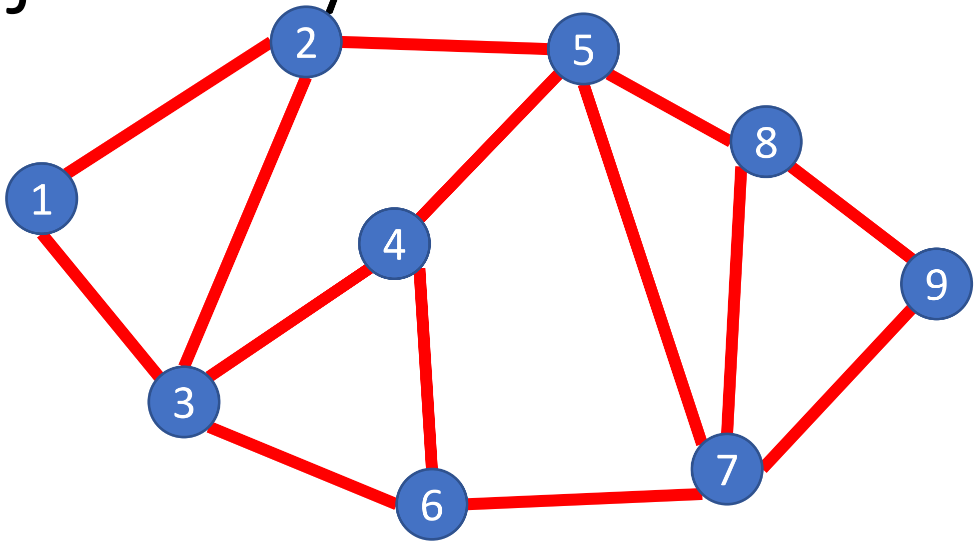
Get Neighbors (incoming): $\Theta(n+m)$

Get Neighbors (outgoing): $\Theta(\deg(v))$

$|V| = n$

$|E| = m$

Adjacency Matrix



	A	B	C	D	E	F	G	H	I
A		1	1						
B	1		1		1				
C	1	1		1		1			
D			1		1	1			
E		1		1			1	1	
F			1	1			1		
G					1	1		1	1
H					1		1		1
I							1	1	

<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(?)$

Add Edge (v, w): $\Theta(?)$

Remove Edge (v, w): $\Theta(?)$

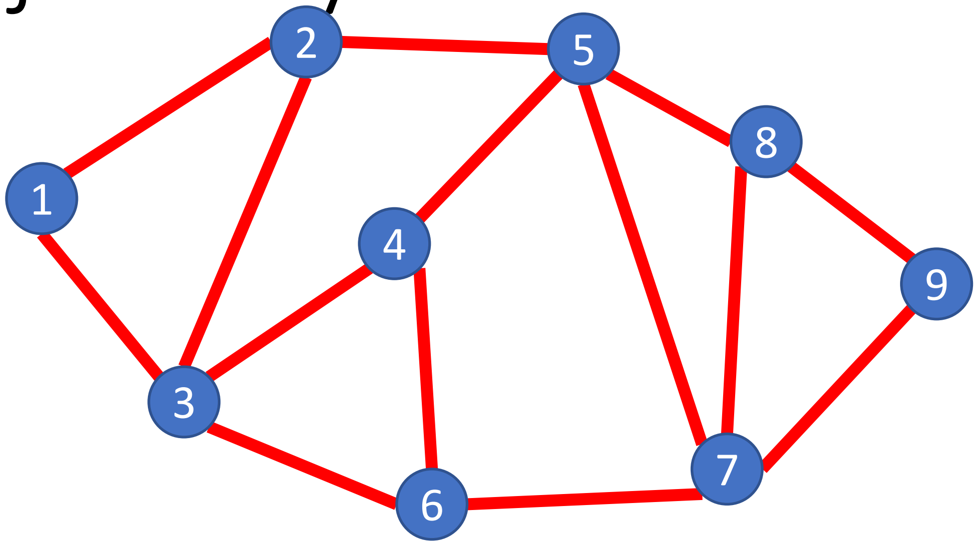
Check if Edge (v, w) Exists: $\Theta(?)$

Get Neighbors (incoming): $\Theta(?)$

Get Neighbors (outgoing): $\Theta(?)$

$|V| = n$
$|E| = m$

Adjacency Matrix



	A	B	C	D	E	F	G	H	I
A		1	1						
B	1		1		1				
C	1	1		1		1			
D			1		1	1			
E		1		1			1	1	
F			1	1			1		
G					1	1		1	1
H					1		1		1
I							1	1	

<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(n^2)$

Add Edge (v, w): $\Theta(1)$

Remove Edge (v, w): $\Theta(1)$

Check if Edge (v, w) Exists: $\Theta(1)$

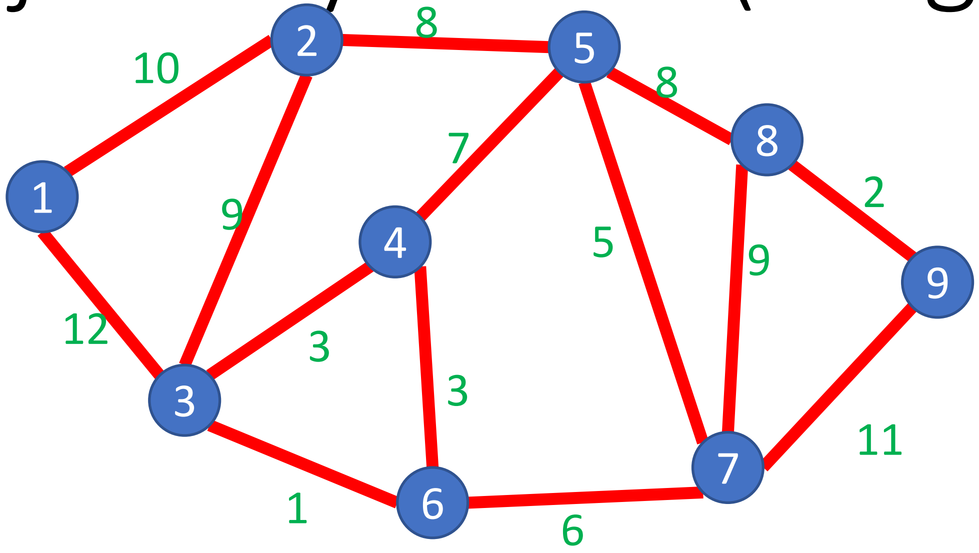
Get Neighbors (incoming): $\Theta(n)$

Get Neighbors (outgoing): $\Theta(n)$

$|V| = n$

$|E| = m$

Adjacency Matrix (weighted)



	A	B	C	D	E	F	G	H	I
A		1	1						
B	1		1		1				
C	1	1		1		1			
D			1		1	1			
E		1		1			1	1	
F			1	1			1		
G					1	1		1	1
H					1		1		1
I							1	1	

<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(n^2)$

Add Edge (v, w): $\Theta(1)$

Remove Edge (v, w): $\Theta(1)$

Check if Edge (v, w) Exists: $\Theta(1)$

Get Neighbors (incoming): $\Theta(n)$

Get Neighbors (outgoing): $\Theta(n)$

$|V| = n$

$|E| = m$

Comparison

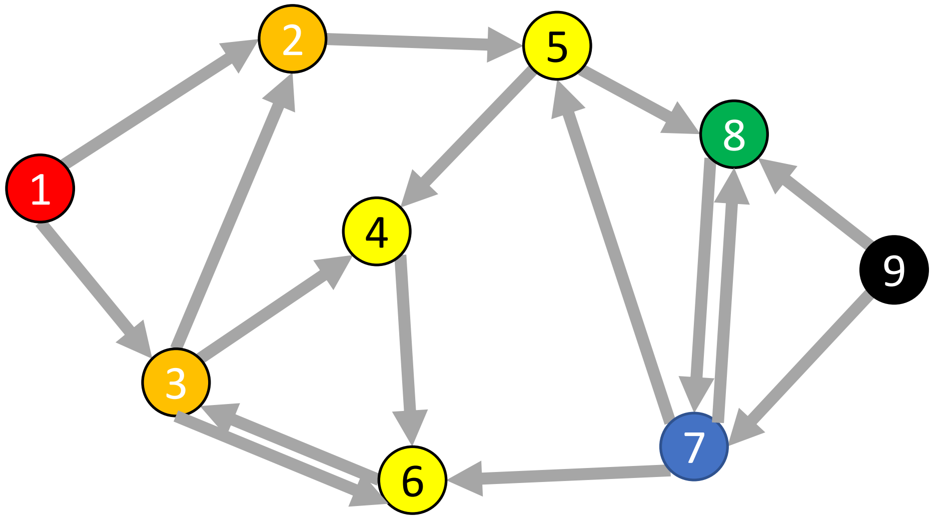
- Adjacency List:
  - Less memory when $|E| < |V|^2$
  - Operations with running time linear in degree of source node
    - Add an edge
    - Remove an edge
    - Check for edge
    - Get neighbors
- Adjacency Matrix:
  - Similar amount of memory when $|E| \approx |V|^2$
  - Constant time operations:
    - Add an edge
    - Remove an edge
    - Check for an edge
  - Operations running with linear time in $|V|$
    - Get neighbors

Adjacency List is more common in practice:

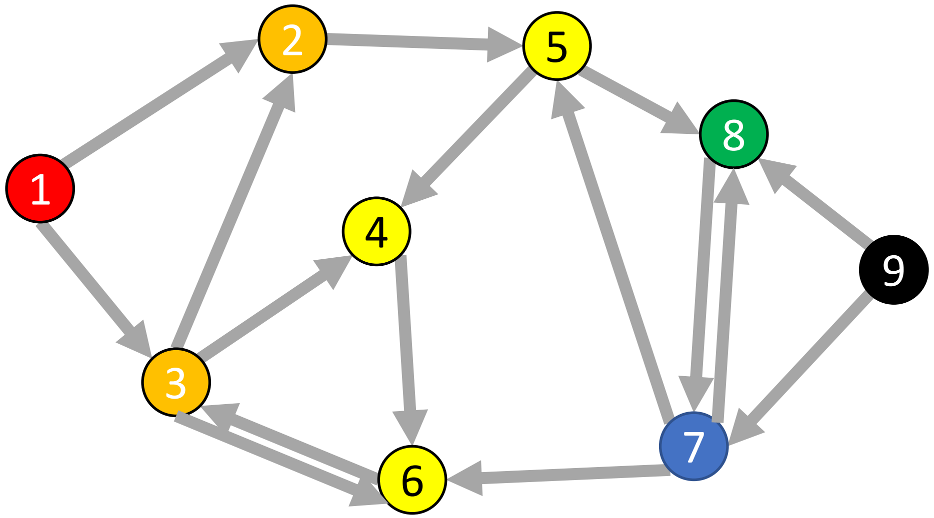
- Most graphs have $|E| \ll |V|^2$
  - Saves memory
  - Most nodes will have small degree
- Getting neighbors is a common operation
- Adjacency Matrix may be better if the graph is "dense" or if its edges change a lot

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?



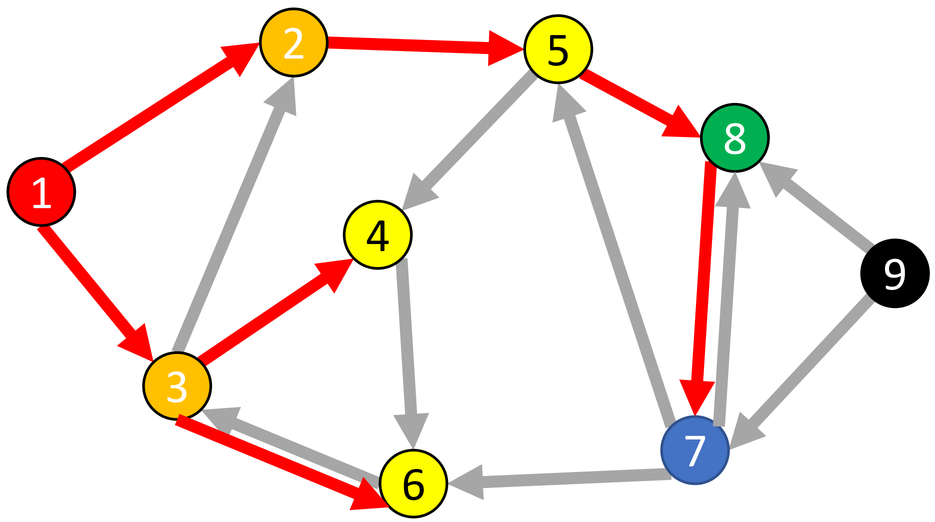
BFS



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);
            }
        }
    }
}
```

Shortest Path (unweighted)



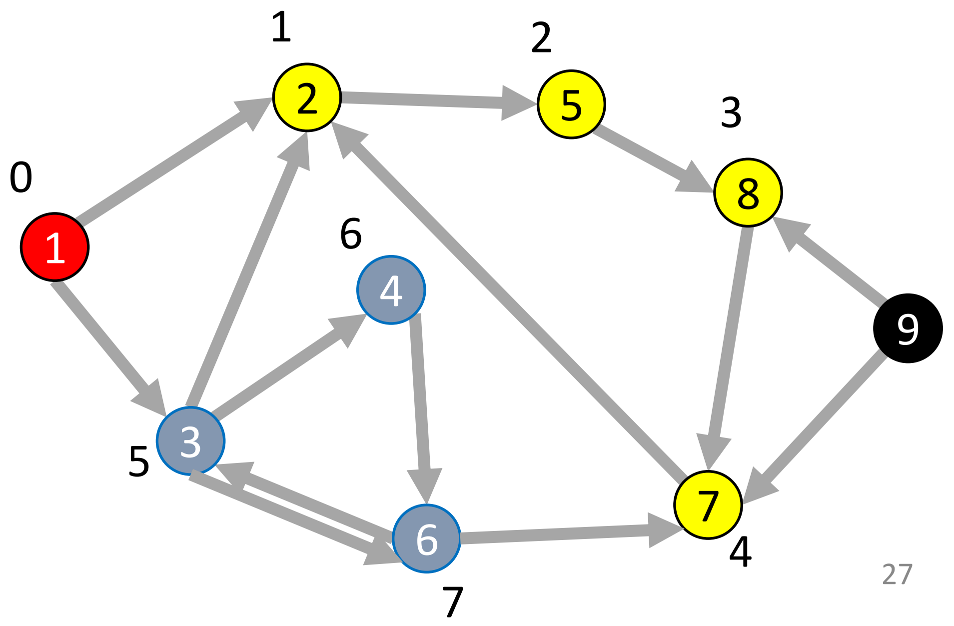
Idea: when it's seen, remember its "layer" depth!

```
int shortestPath(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;
}
```

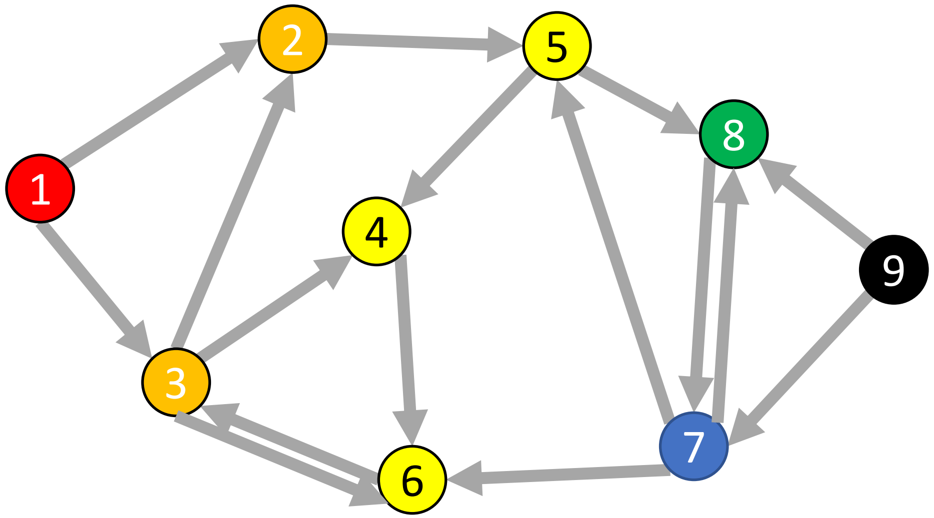
Depth-First Search

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)

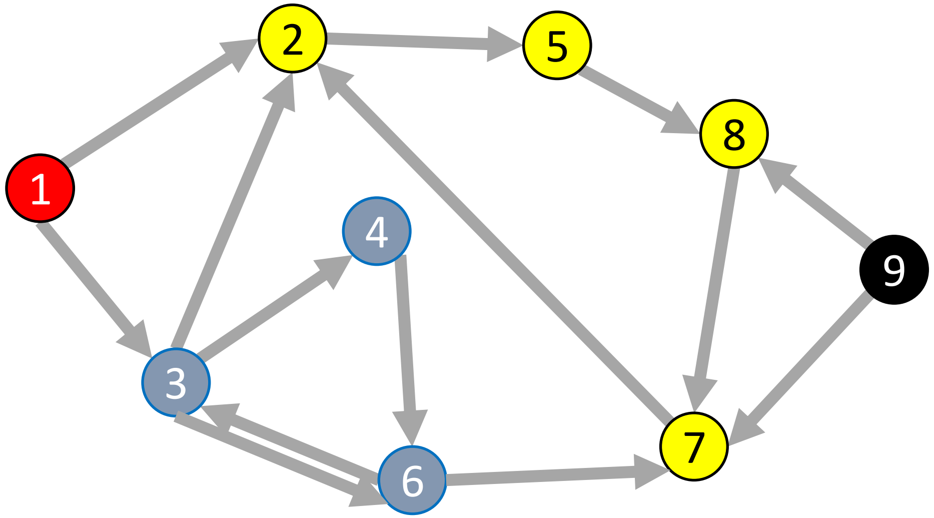


Running time: $\Theta(|V| + |E|)$

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

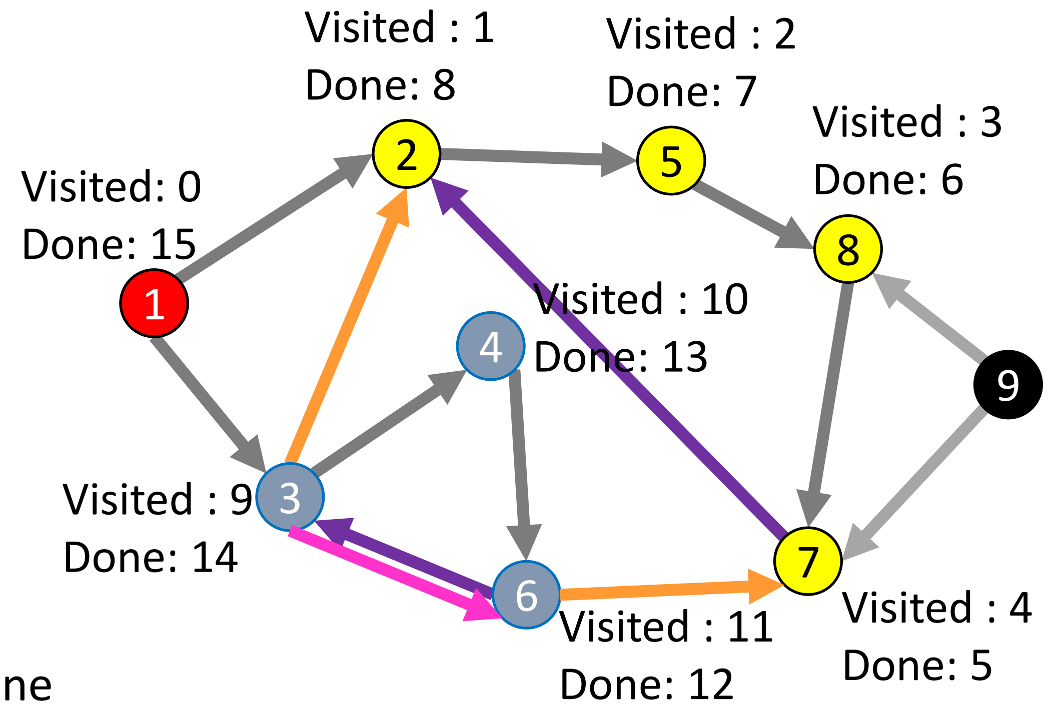
DFS Recursively (more common)

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



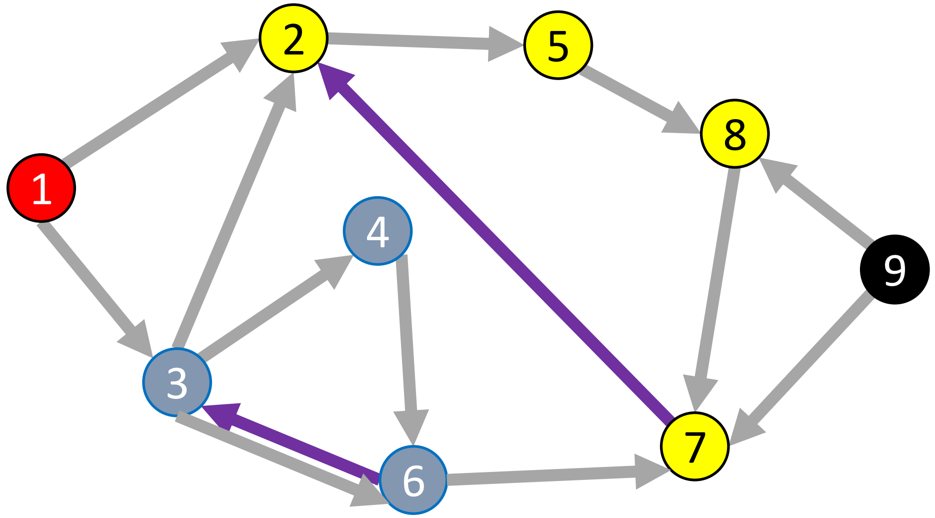
Using DFS

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



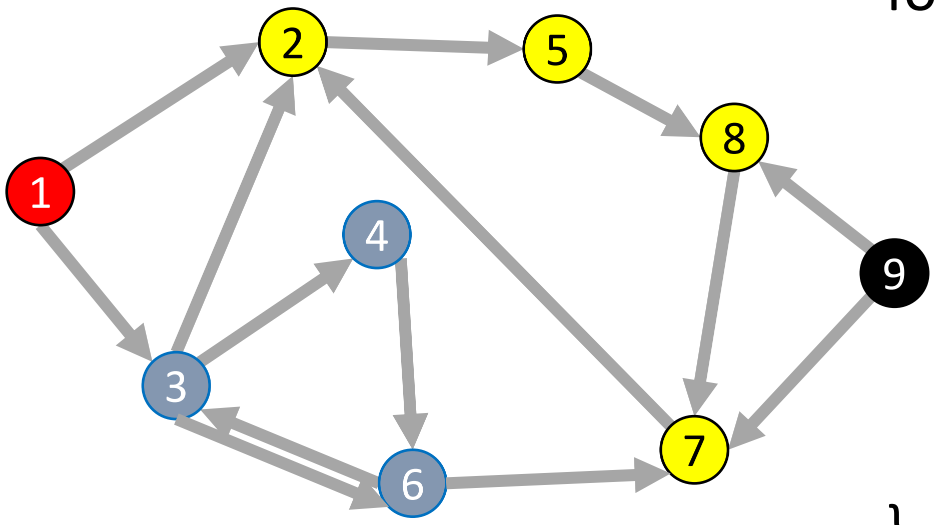
Back Edges

- 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

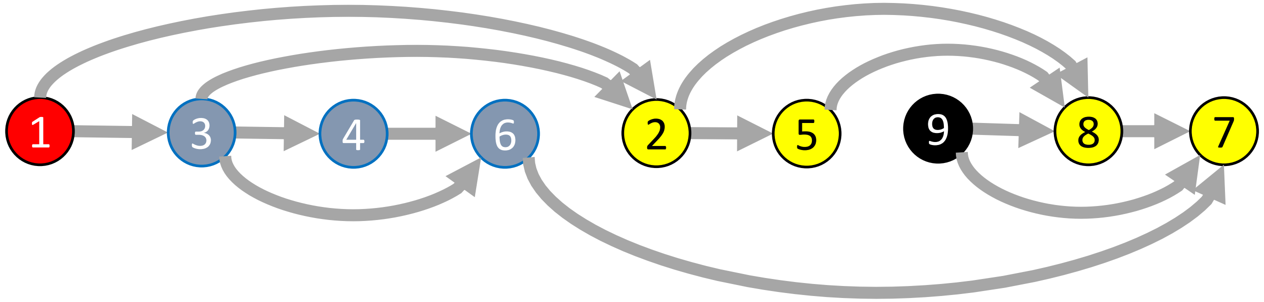
Idea: Look for a back edge!

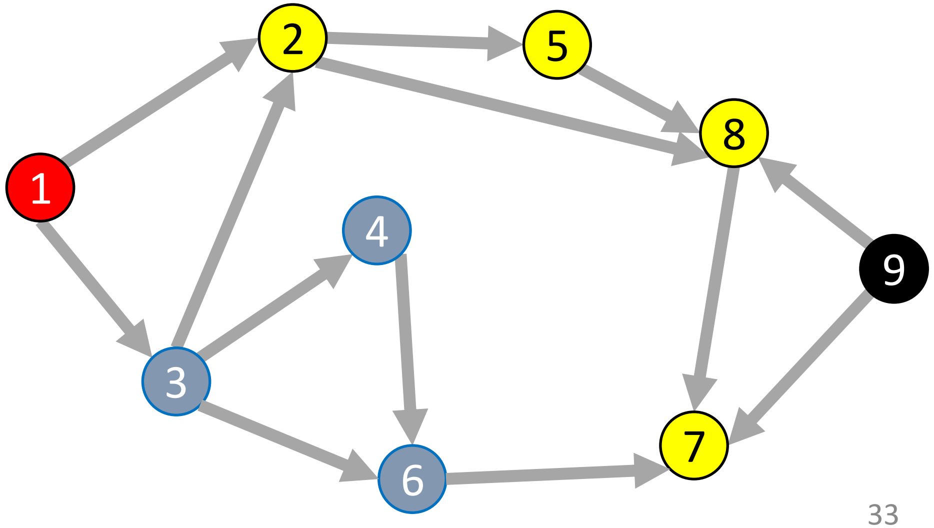



```
boolean hasCycle(graph, curr){
        mark curr as “visited”;
        cycleFound = false;
        for (v : neighbors(current)){
                if (v marked “visited” && ! v marked “done”){
                        cycleFound=true;
                }
                if (! v marked “visited” && !cycleFound){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

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

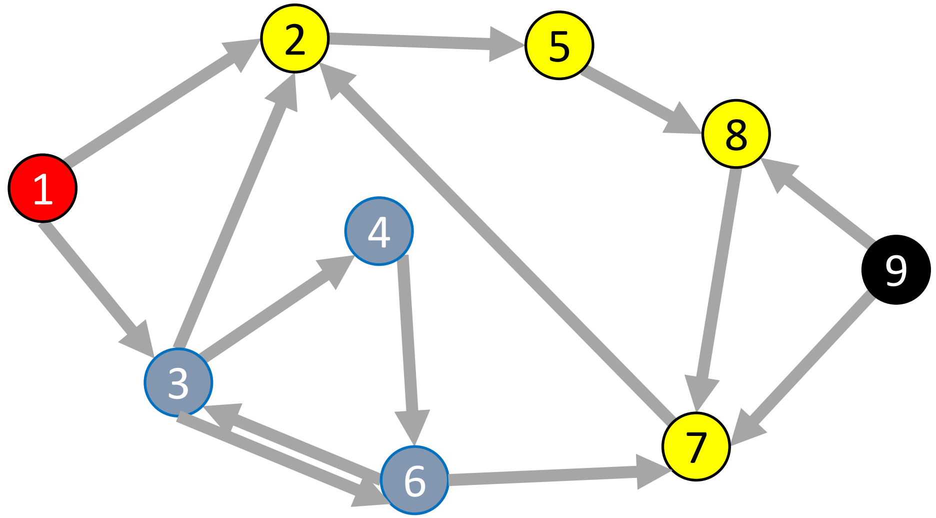




DFS Recursively

```
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

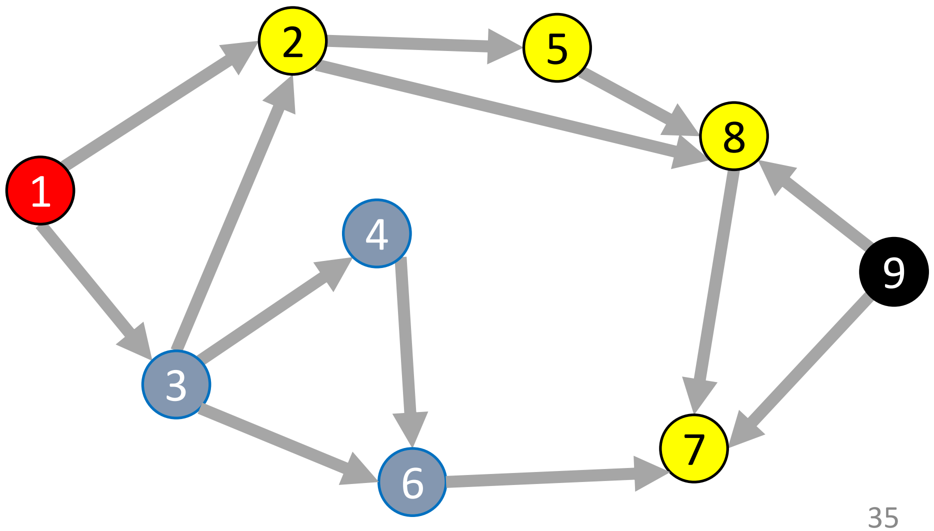


DFS: Topological sort

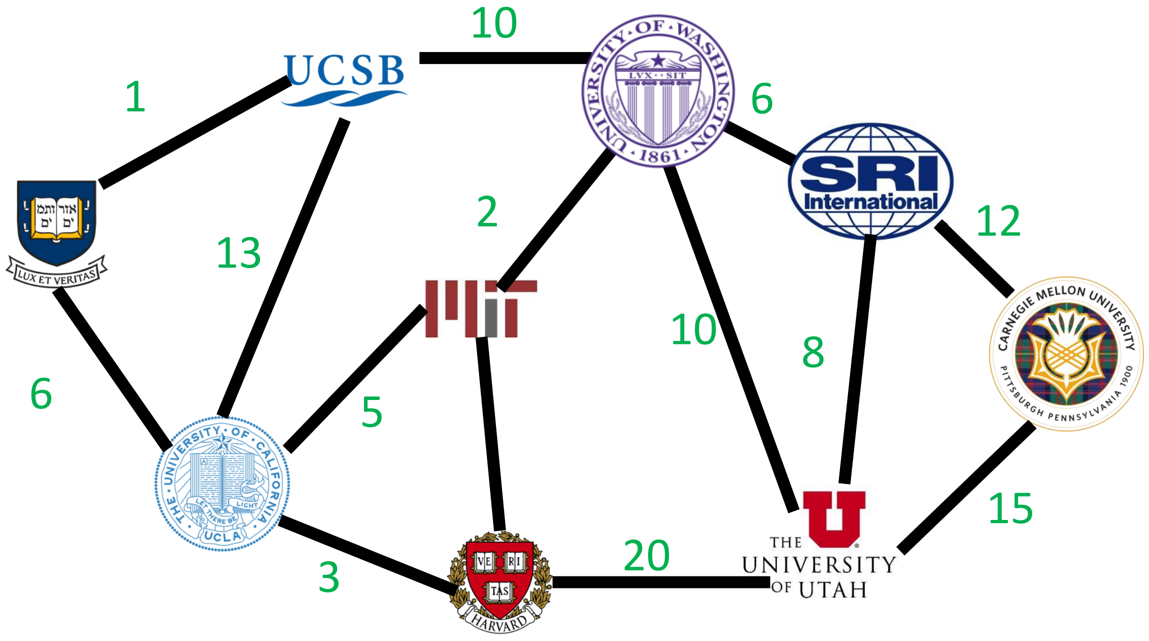
```
List topSort(graph){
        List<Nodes> done = new List<>();
        for (Node v : graph.vertices){
                if (!v.visited){
                        finishTime(graph, v, finished);
                }
        }
        done.reverse();
        return done;
}

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:



Single-Source Shortest Path



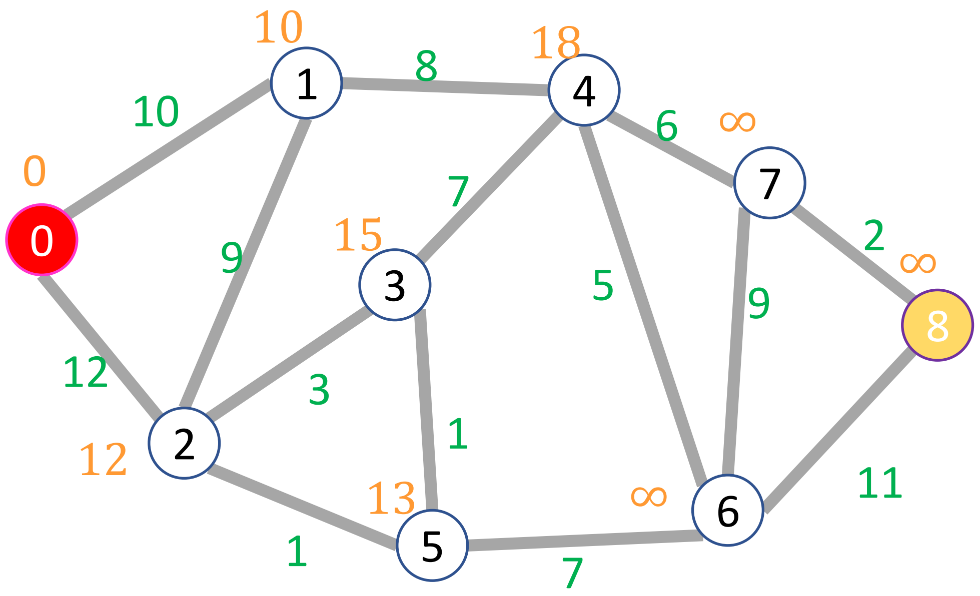
Find the quickest way to get from UVA to each of these other places

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

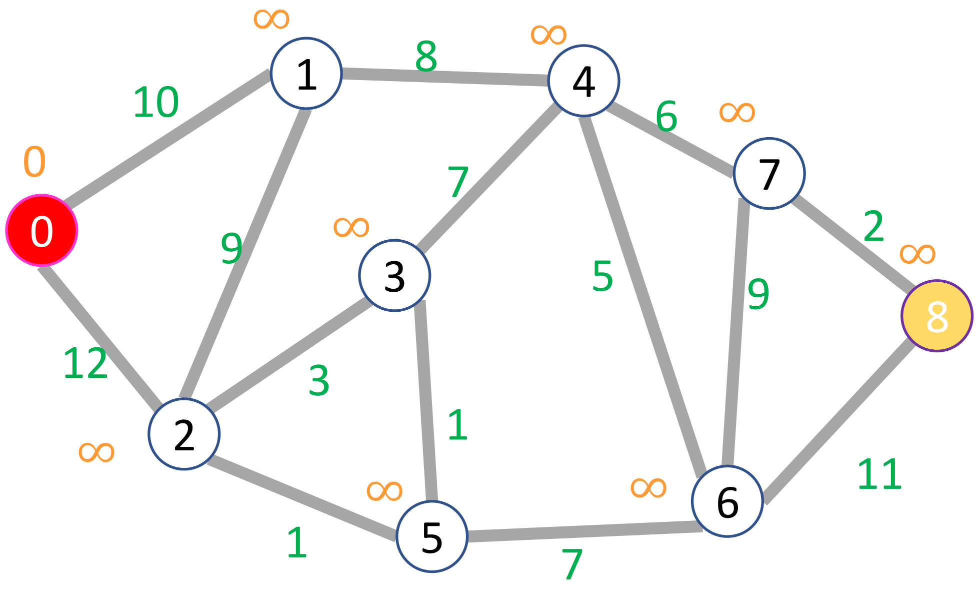
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

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

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



Dijkstra's Algorithm

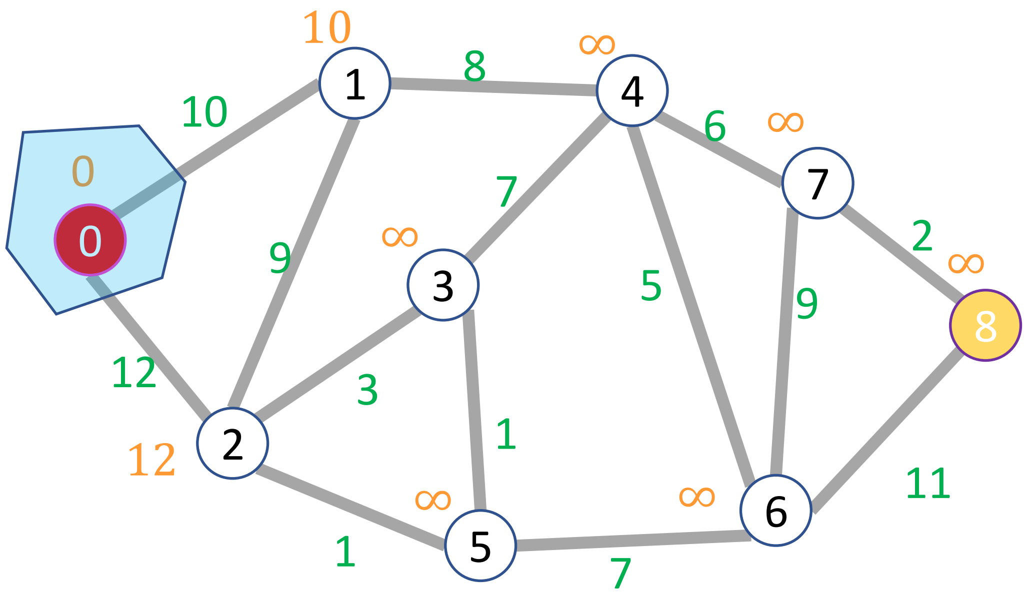
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

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

Node	Distance
0	0
1	10
2	12
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞



Dijkstra's Algorithm

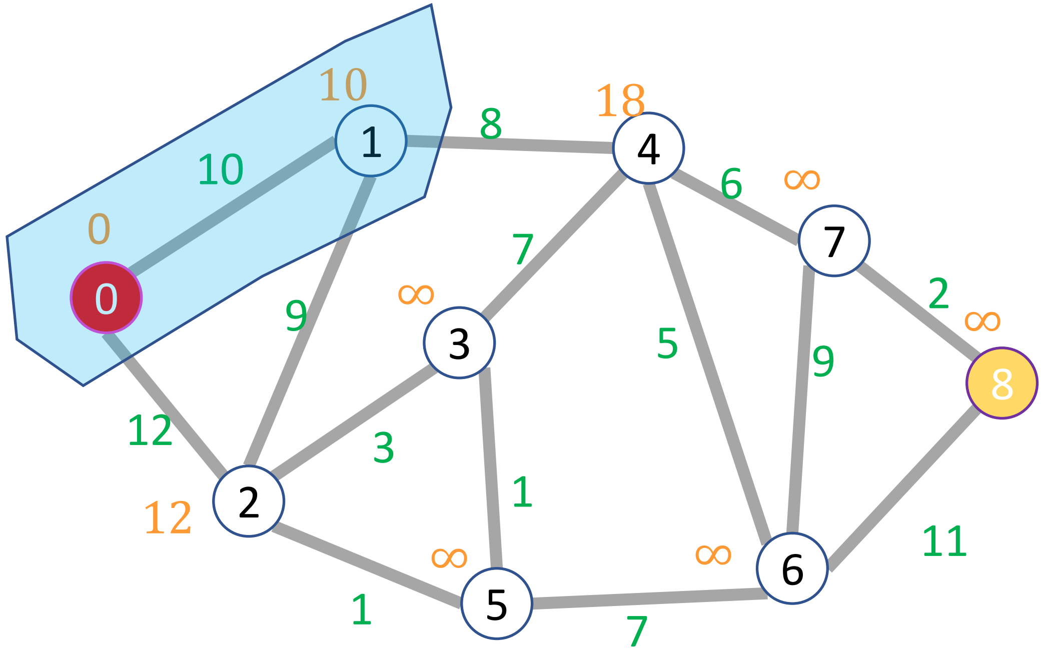
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

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

Node	Distance
0	0
1	10
2	12
3	∞
4	18
5	∞
6	∞
7	∞
8	∞



Dijkstra's Algorithm

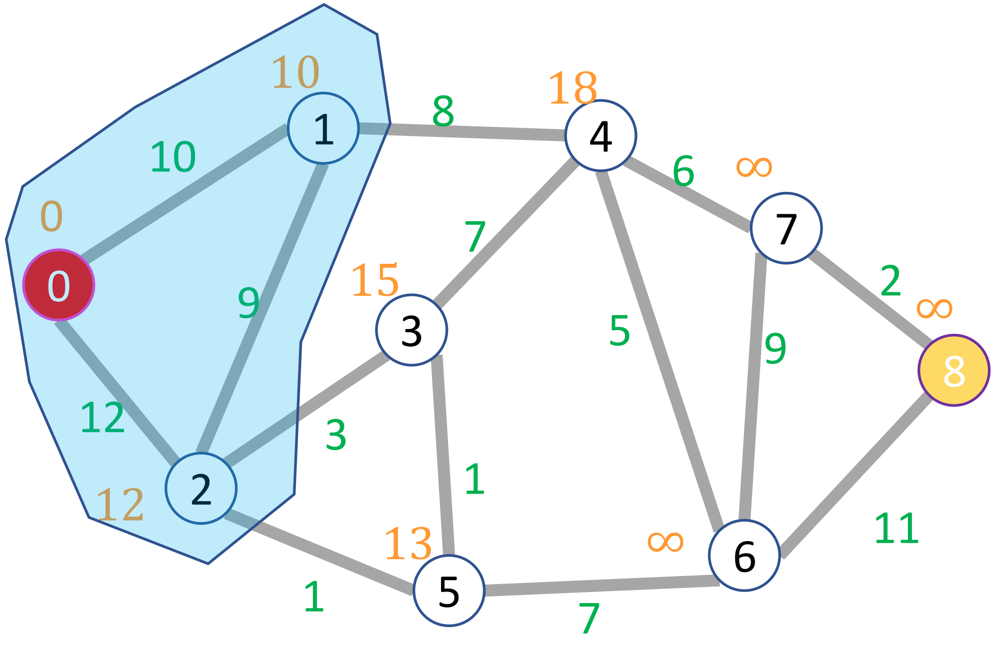
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

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

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



Dijkstra's Algorithm

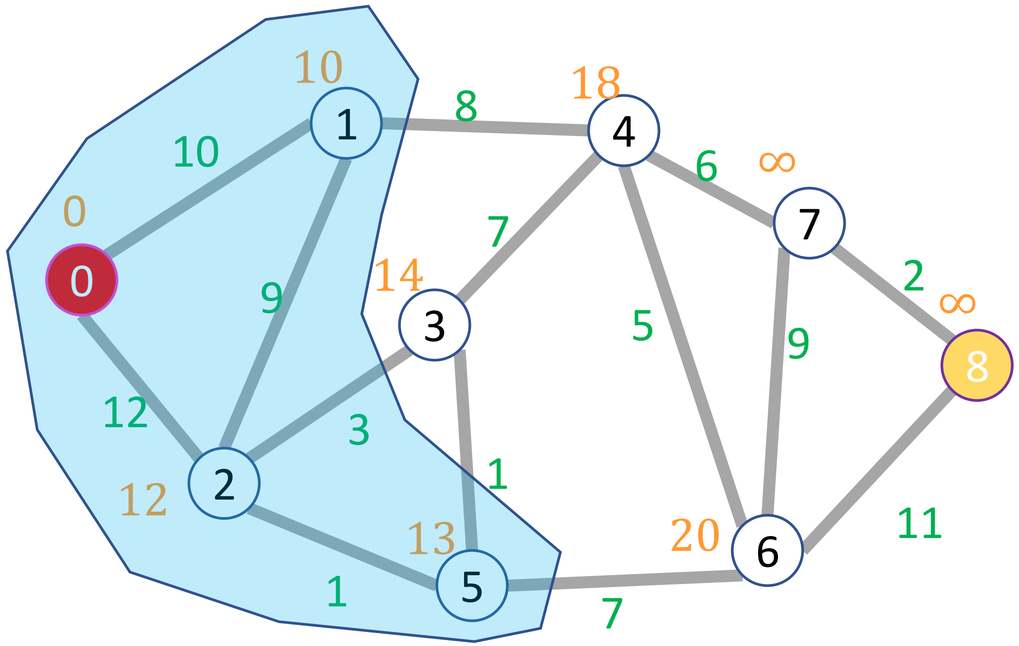
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

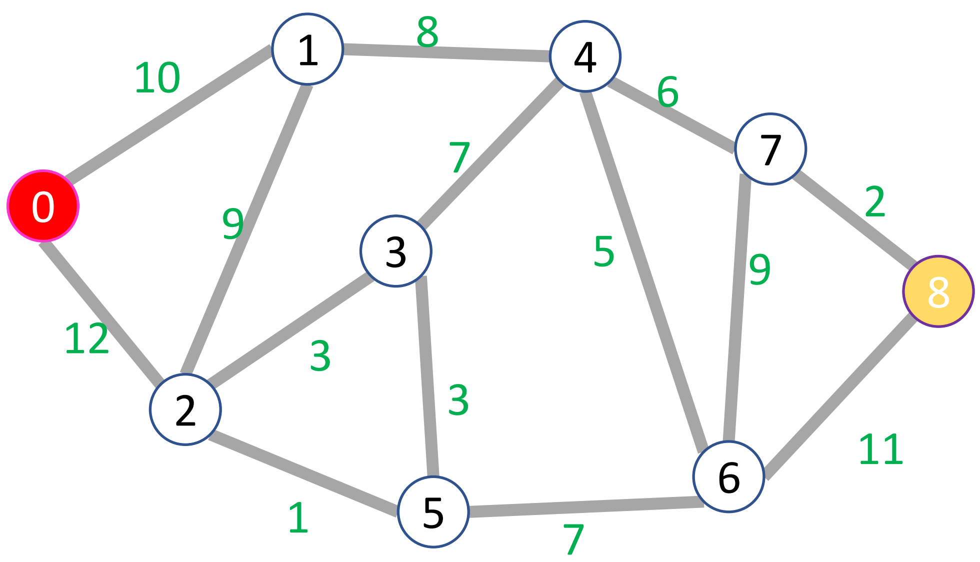
Node	Done?
0	T
1	T
2	T
3	F
4	F
5	T
6	F
7	F
8	F

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



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]
}
```

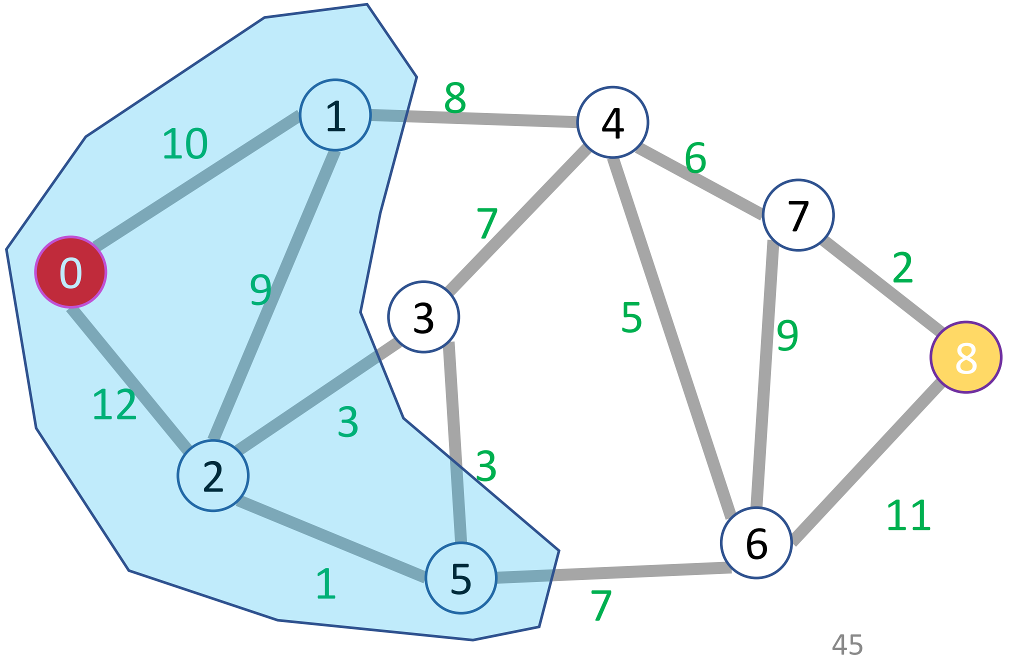


Dijkstra’s Algorithm: Running Time

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

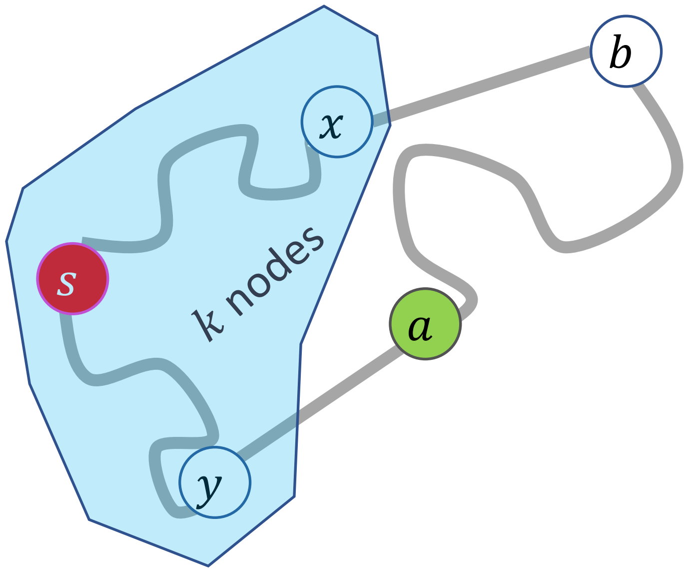
Dijkstra's Algorithm: Correctness

- 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:



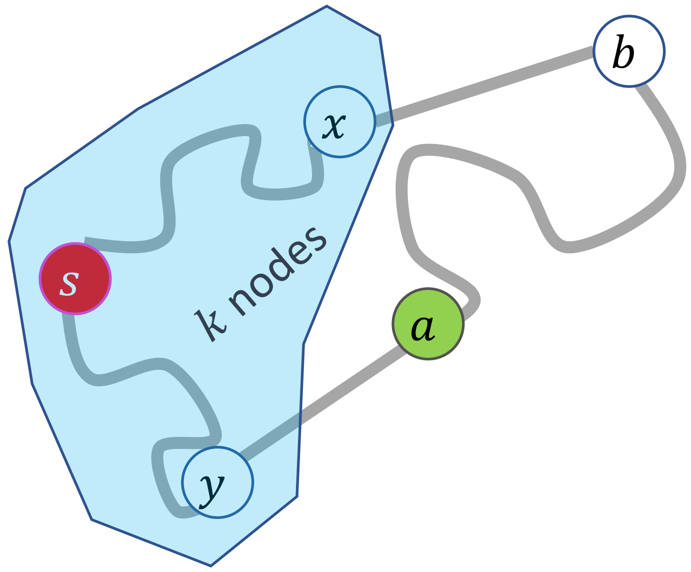
Dijkstra's Algorithm: Correctness

- 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



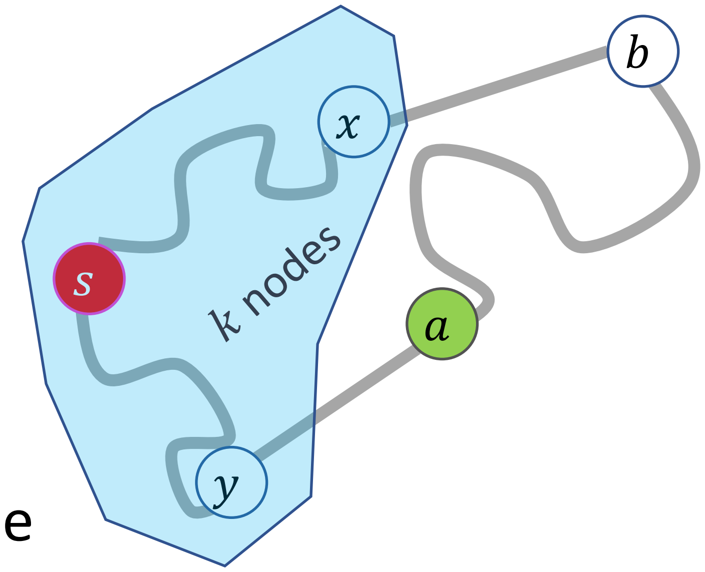
Dijkstra’s Algorithm: Correctness

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



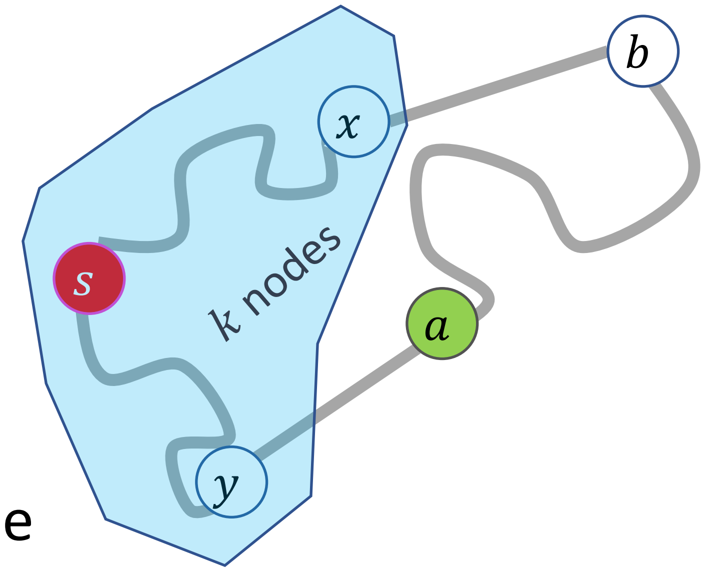
Dijkstra's Algorithm: Correctness

- 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!



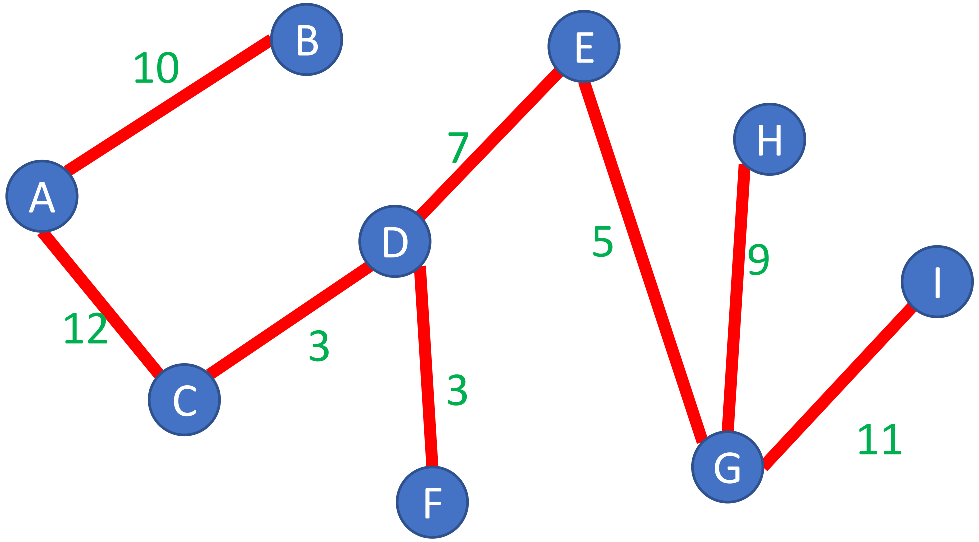
Dijkstra's Algorithm: Correctness

- 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!



Definition: Tree

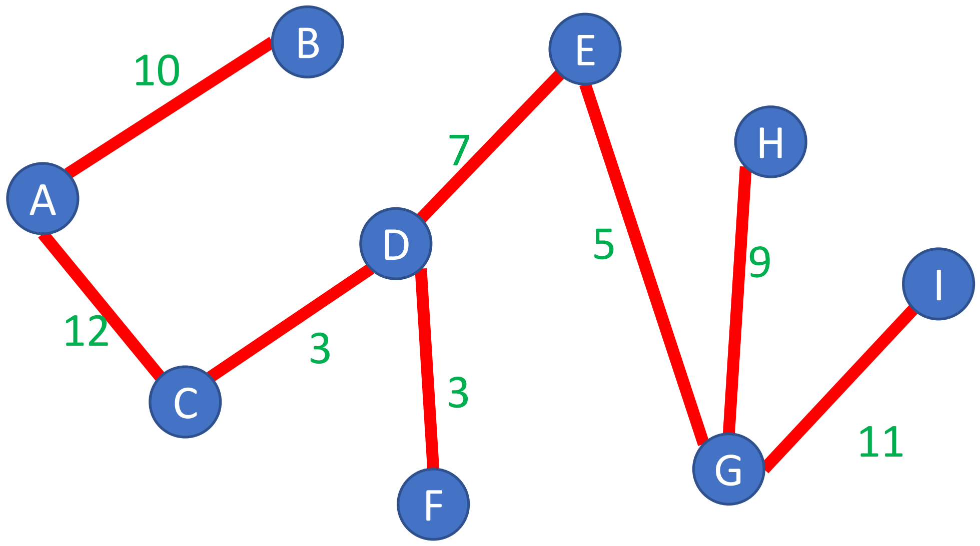
A connected graph with no cycles



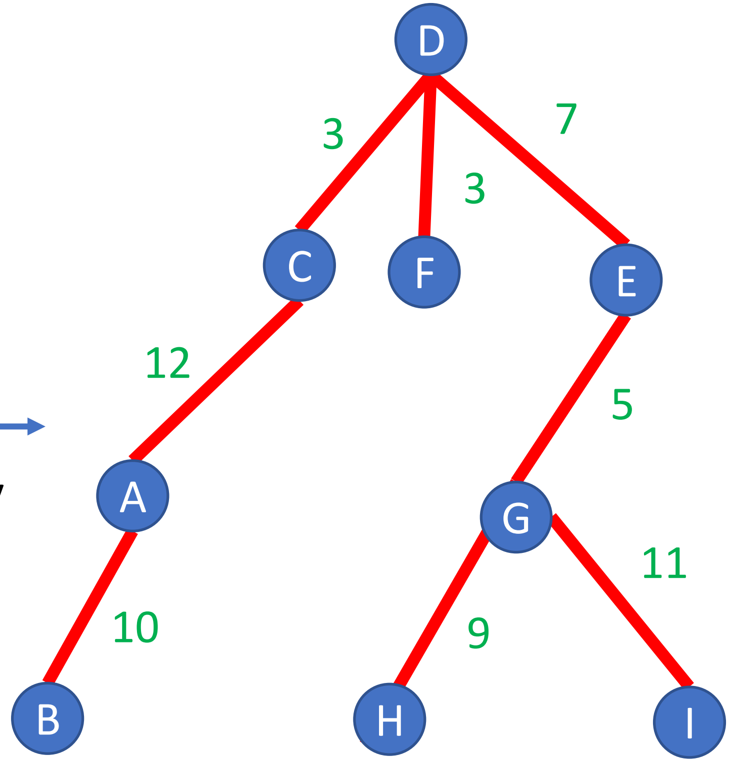
Note: A tree does not need a root, but they often do!

Definition: Tree

A connected graph with no cycles



Pick some arbitrary root node and rearrange tree

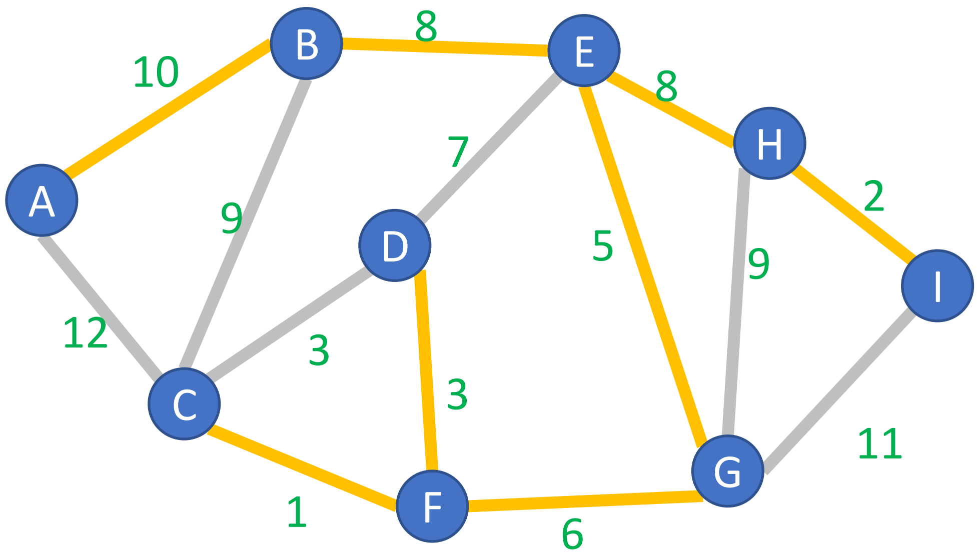


Definition: Spanning Tree

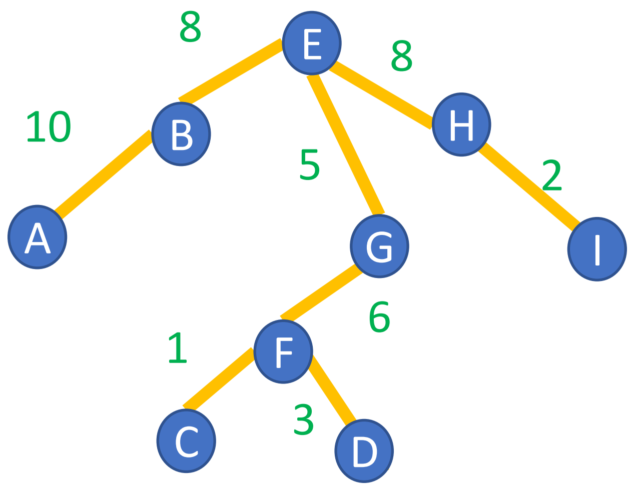
A Tree $T = (V_T, E_T)$ which connects ("spans") all the nodes in a graph $G = (V, E)$

How many edges does T have?

$V - 1$



Pick some arbitrary root node and rearrange tree

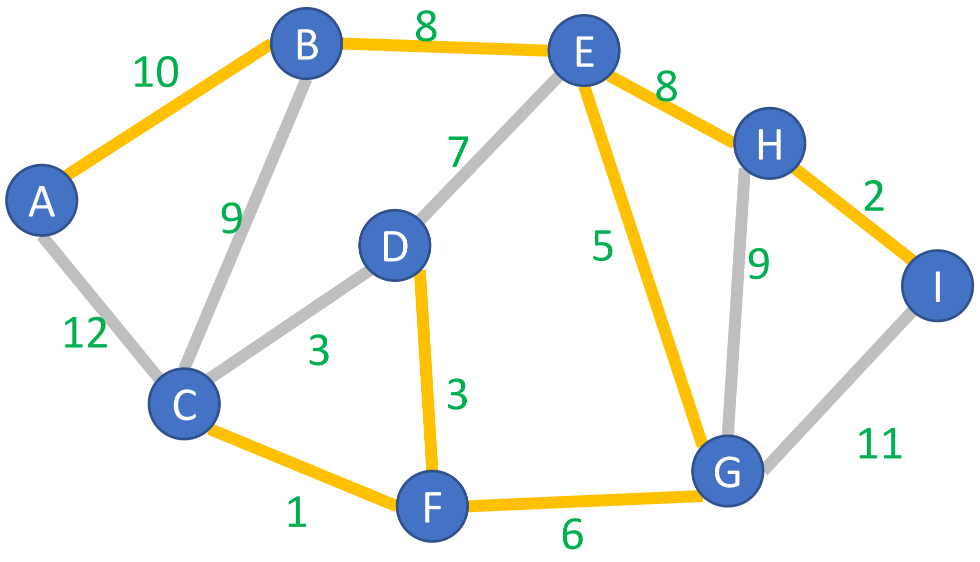


Any set of V-1 edges in the graph that doesn't have any cycles is guaranteed to be a spanning tree!

Any set of V-1 edges that connects all the nodes in the graph is guaranteed to be a spanning tree!

Definition: Minimum Spanning Tree

A Tree $\boldsymbol{T} = (V_T, E_T)$ which connects ("spans") all the nodes in a graph $G = (V, E)$, that has minimal cost

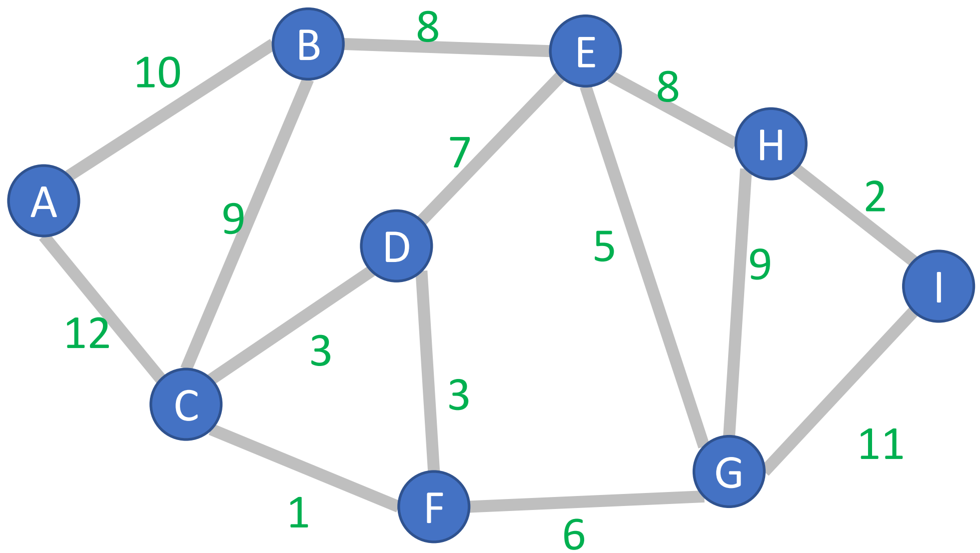


$$Cost(T) = \sum_{e \in E_T} w(e)$$

Kruskal's Algorithm

Start with an empty tree A

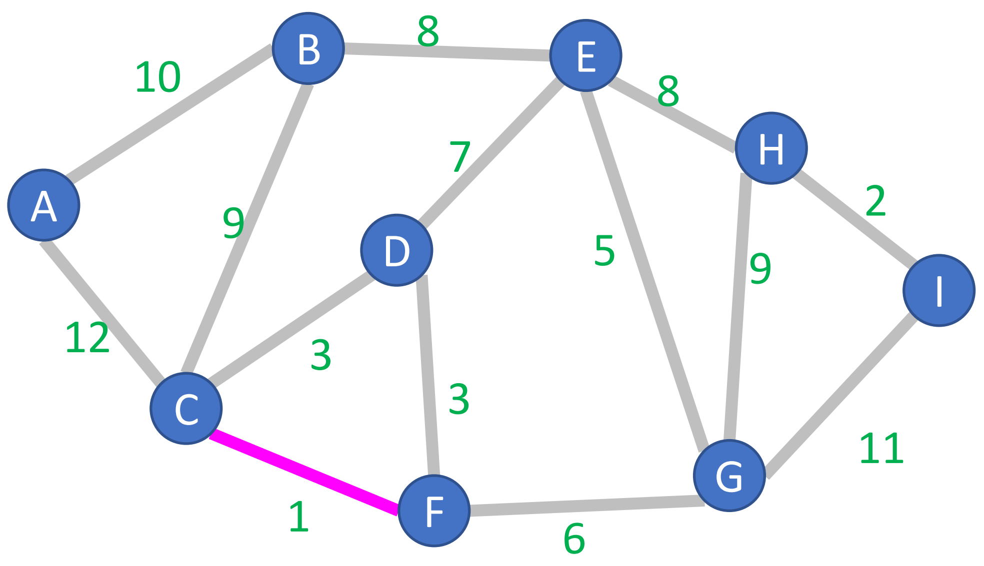
Add to A the lowest-weight edge that does not create a cycle



Kruskal’s Algorithm

Start with an empty tree A

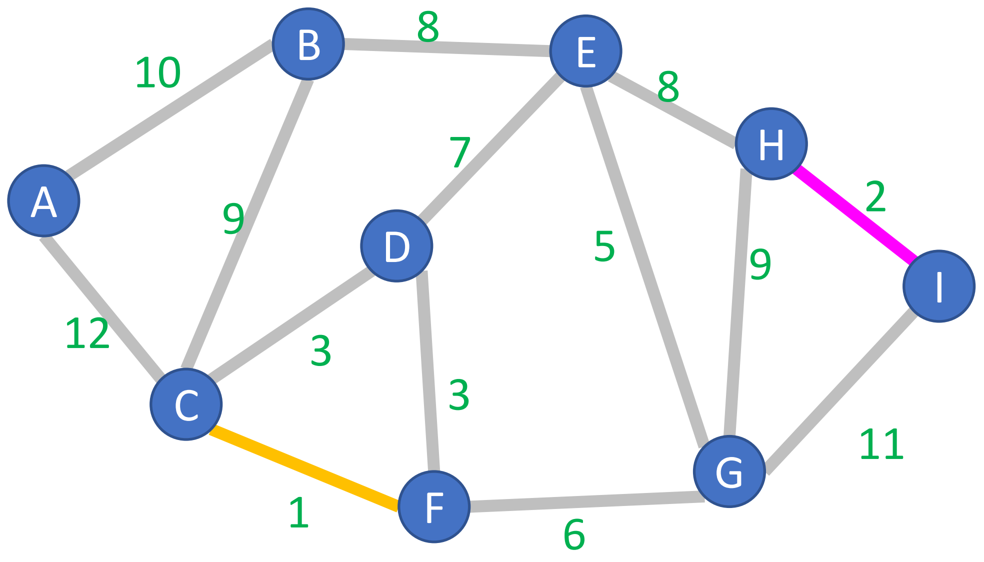
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

Start with an empty tree A

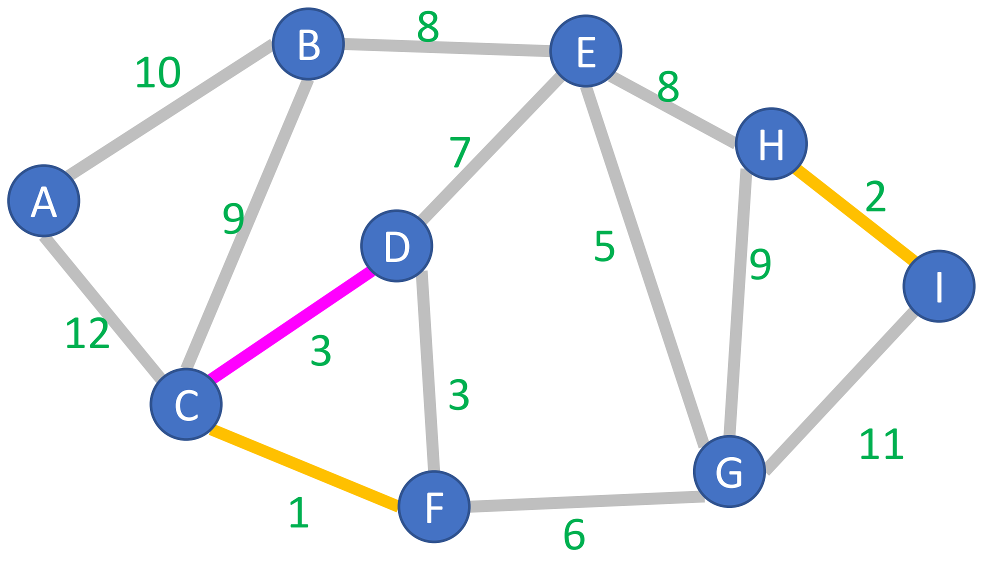
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

Start with an empty tree A

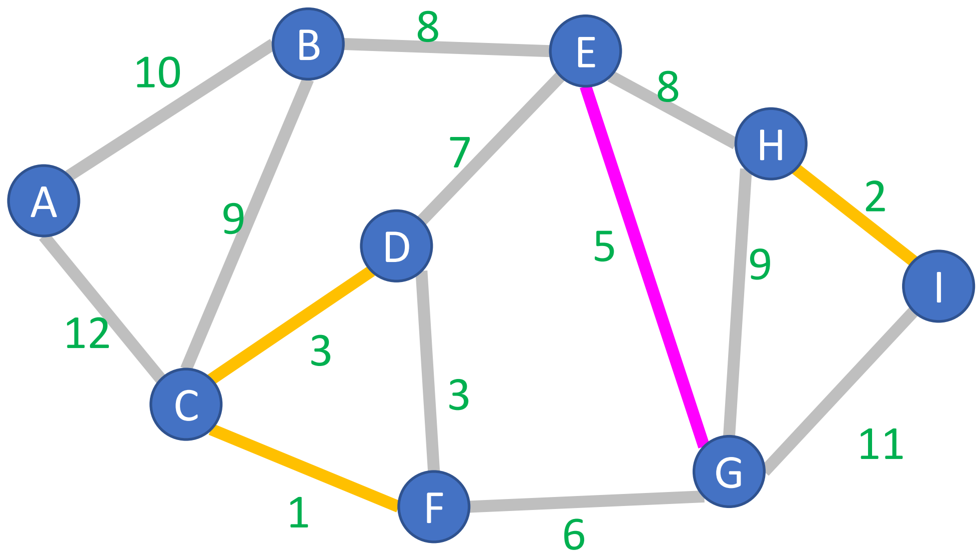
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

Start with an empty tree A

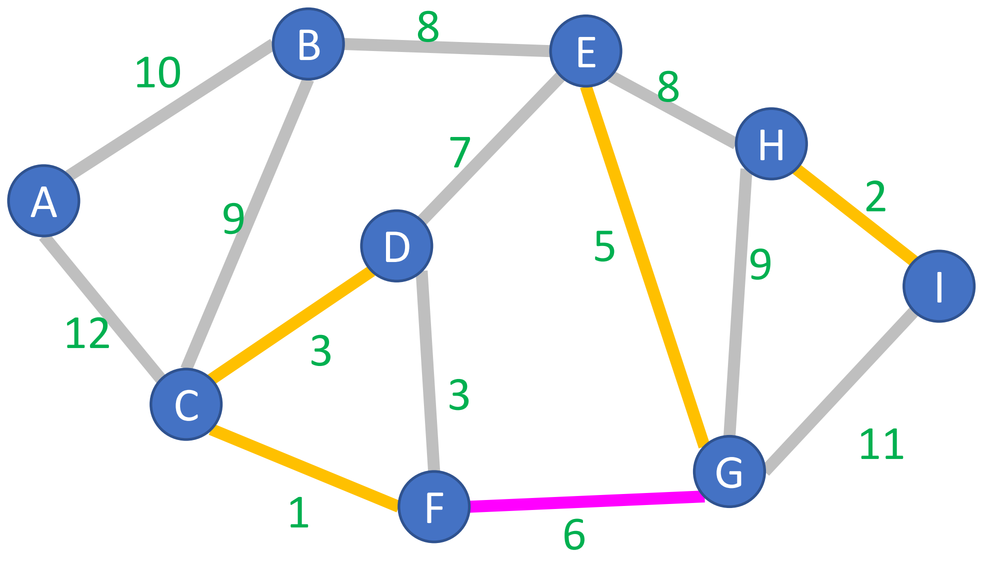
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

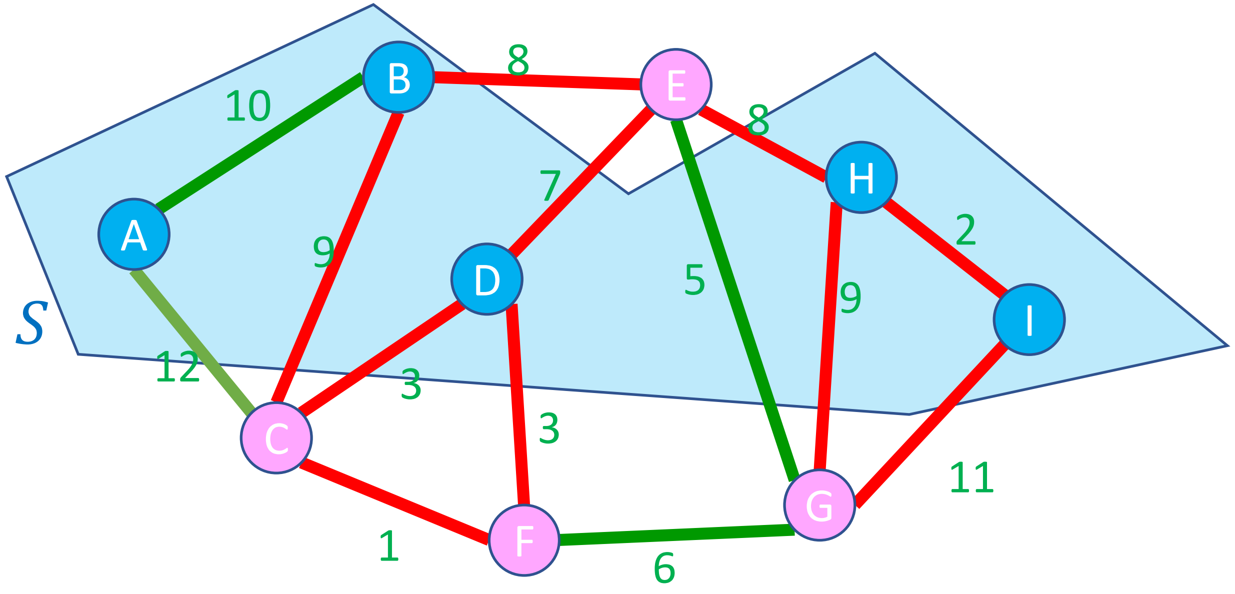
Start with an empty tree A

Add to A the lowest-weight edge that does not create a cycle



Definition: Cut

A Cut of graph $G = (V, E)$ is a partition of the nodes into two sets, S and $V - S$



Edge $(v_1, v_2) \in E$ crosses a cut if $v_1 \in S$ and $v_2 \in V - S$ (or opposite), e.g. (A, C)

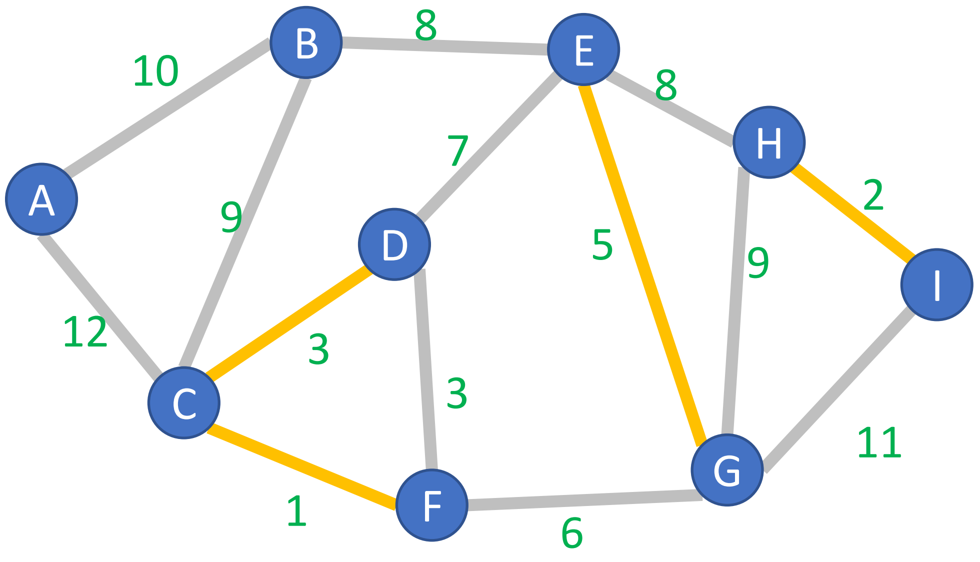
A set of edges R Respects a cut if no edges cross the cut e.g. $R = \{(A, B), (E, G), (F, G)\}$

Cut Theorem

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.

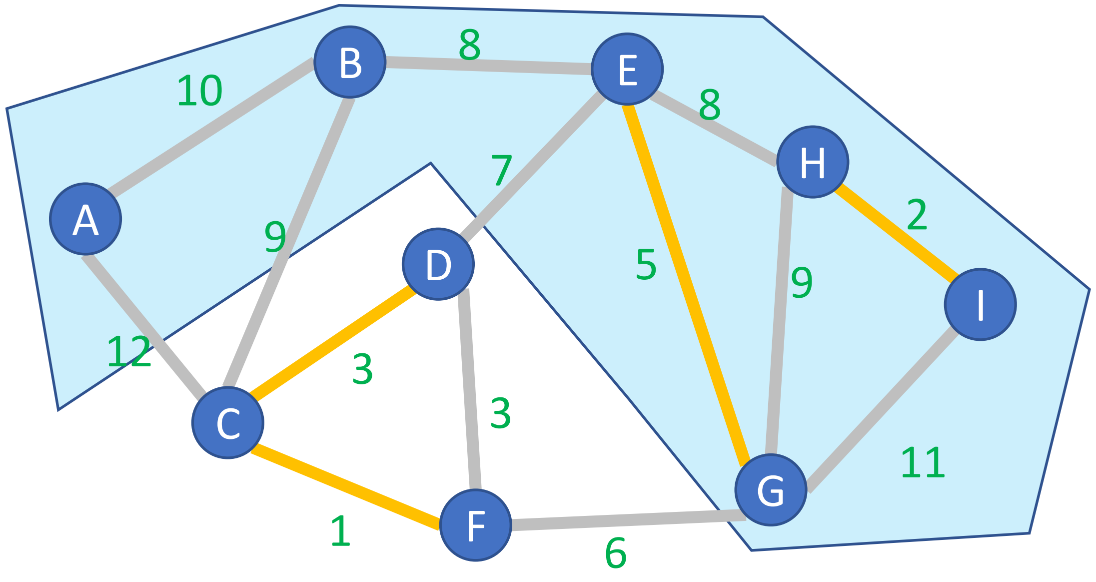
Cut Theorem

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.



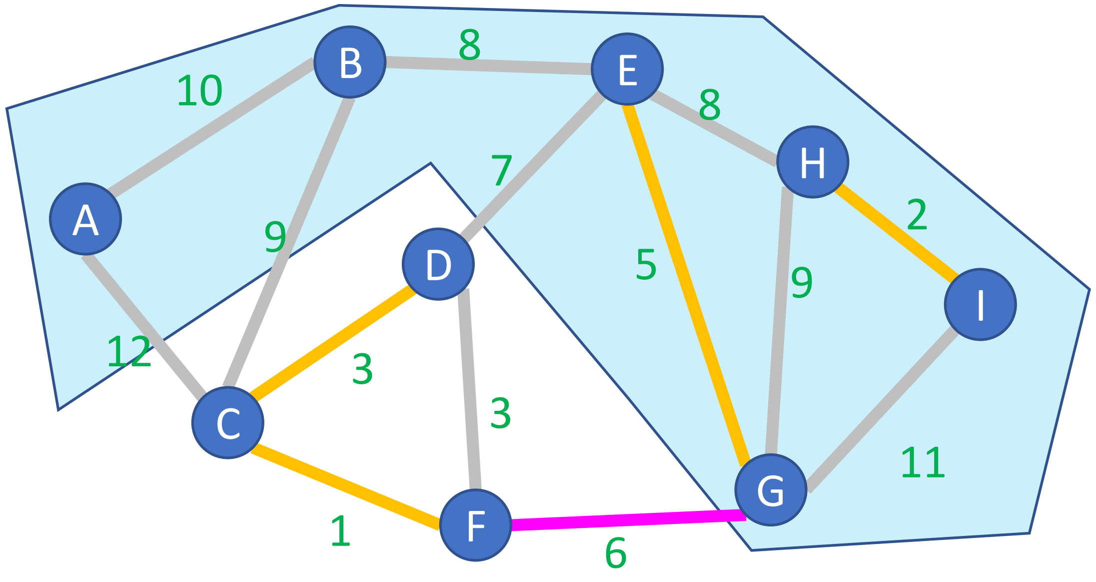
Cut Theorem

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.



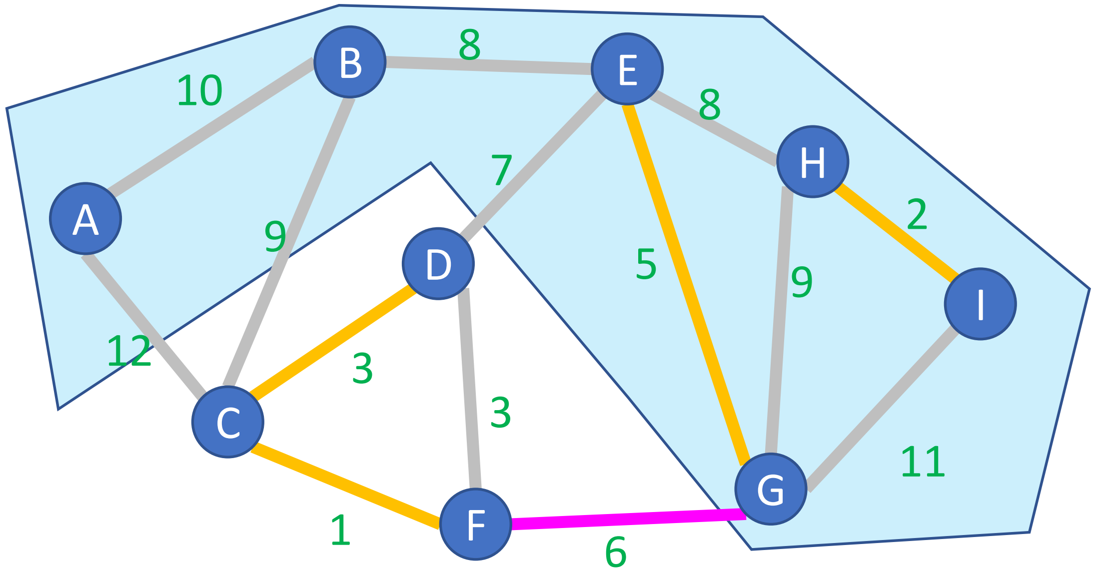
Cut Theorem

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.



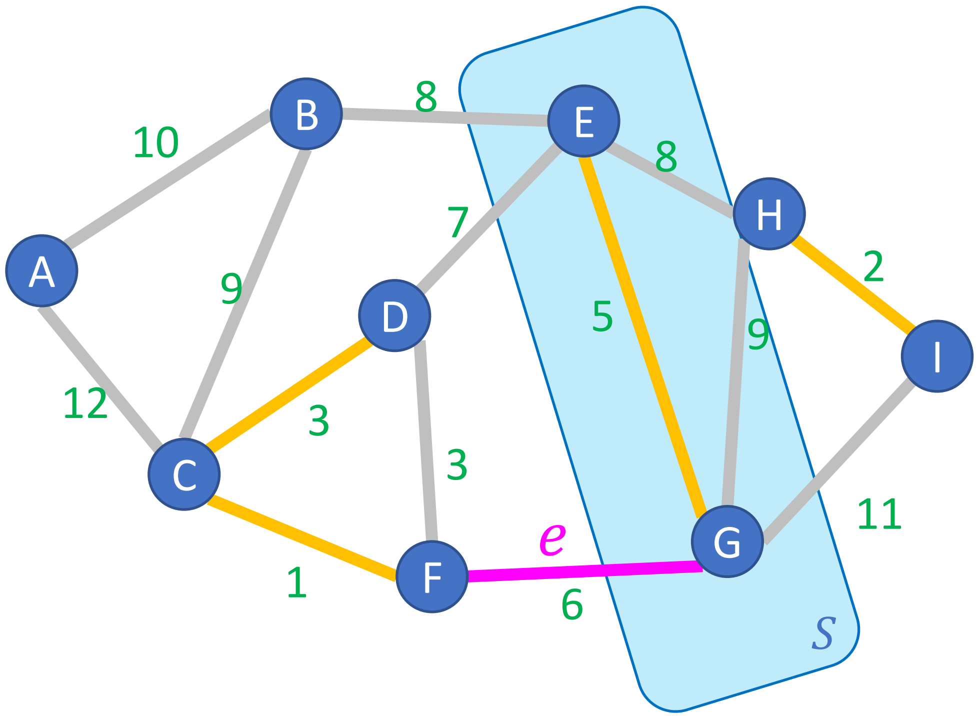
Cut Theorem

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.



Proof of Kruskal's Algorithm

Start with an empty tree A
Repeat $V - 1$ times:
  Add the min-weight edge that doesn't cause a cycle



Proof: Suppose we have some arbitrary set of edges A that Kruskal's has already selected to include in the MST. $e = (F, G)$ is the edge Kruskal's selects to add next

We know that there cannot exist a path from F to G using only edges in A because e does not cause a cycle

We can cut the graph therefore into 2 disjoint sets:

- nodes reachable from G using edges in A
- All other nodes

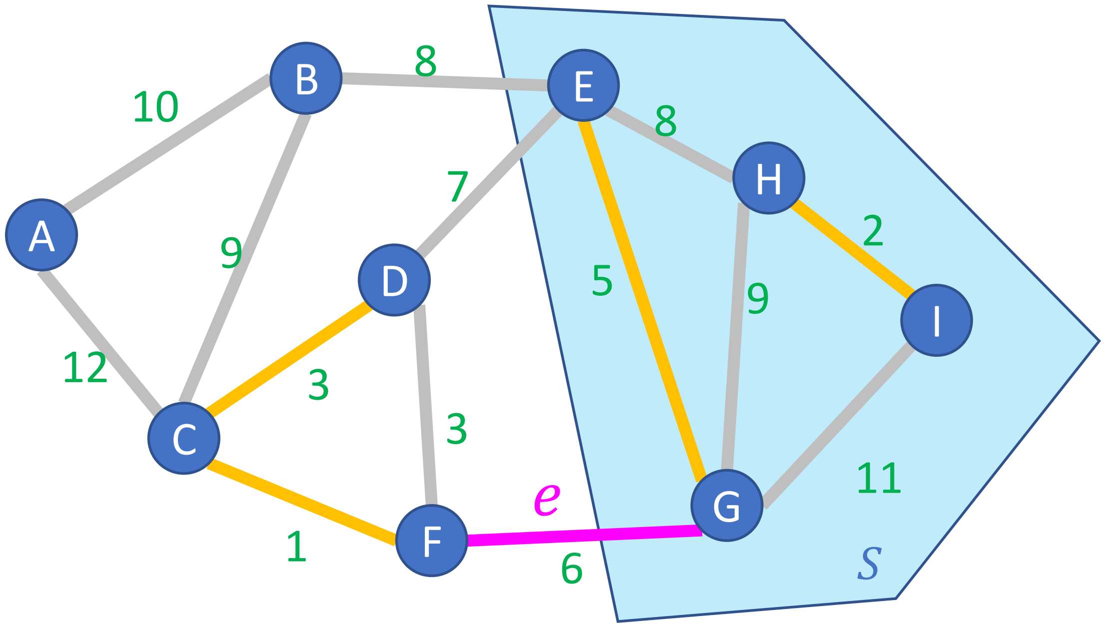
e is the minimum cost edge that crosses this cut, so by the Cut Theorem, Kruskal's is optimal!

Kruskal’s Algorithm Runtime

Start with an empty tree A

Repeat $V - 1$ times:

Add the min-weight edge that doesn’t cause a cycle



Keep edges in a Disjoint-set data structure (very fancy)

$O(E \log V)$

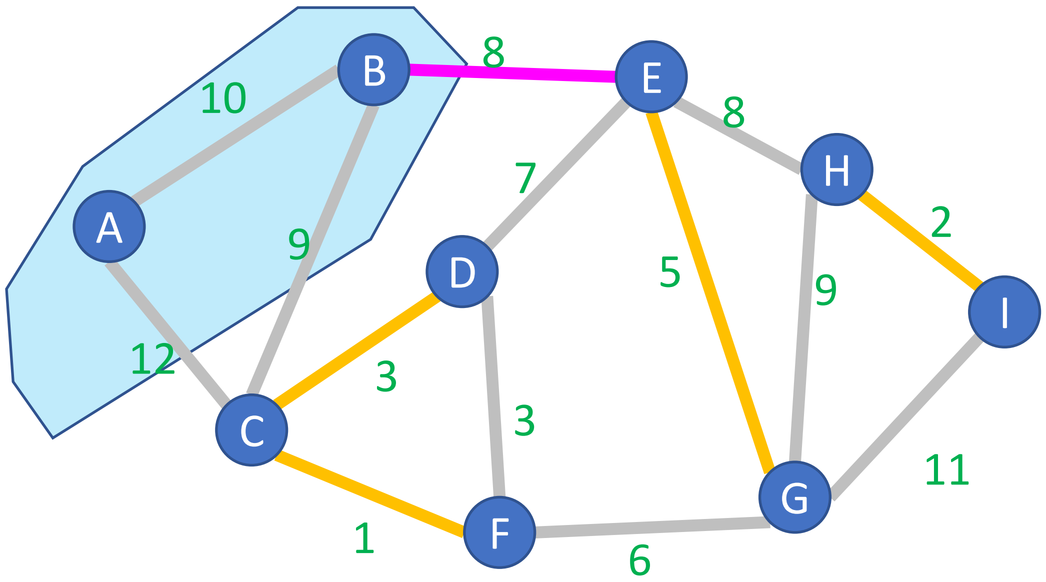
General MST Algorithm

Start with an empty tree A

Repeat $V - 1$ times:

Pick a cut $(S, V - S)$ which A respects (typically implicitly)

Add the min-weight edge which crosses $(S, V - S)$



Prim's Algorithm

Start with an empty tree A

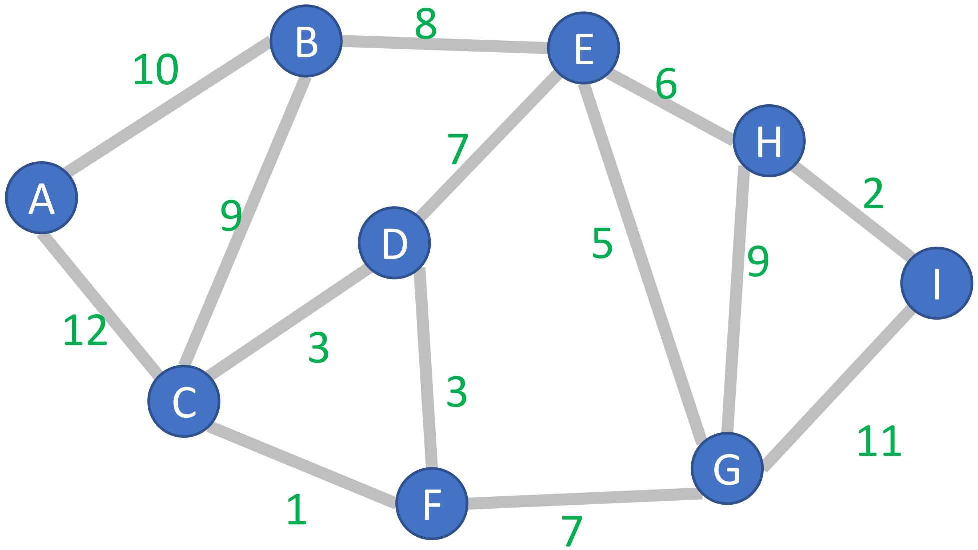
Repeat $V - 1$ times:

Pick a cut $(S, V - S)$ which A respects

Add the min-weight edge which crosses $(S, V - S)$

S is all endpoint of edges in A

e is the min-weight edge that grows the tree



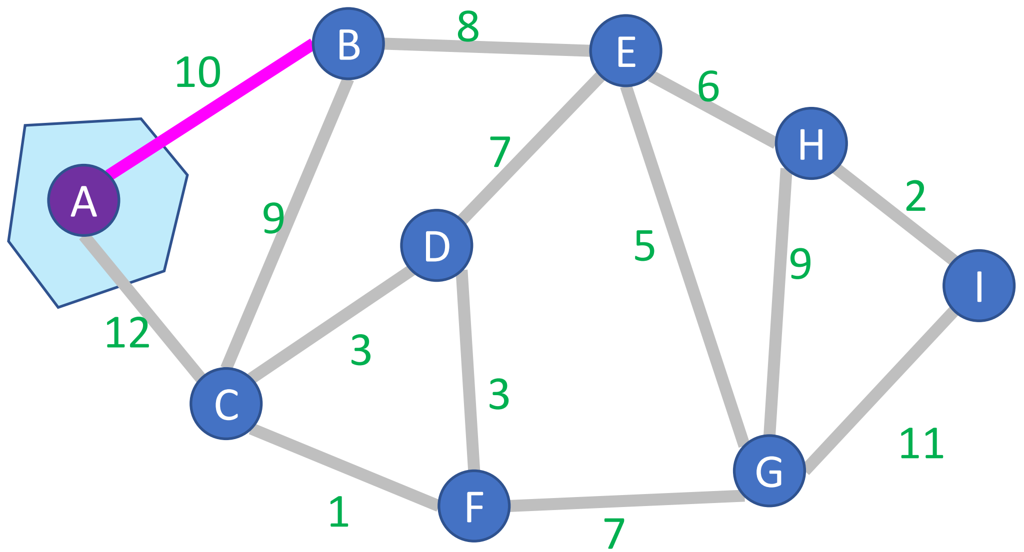
Prim's Algorithm

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



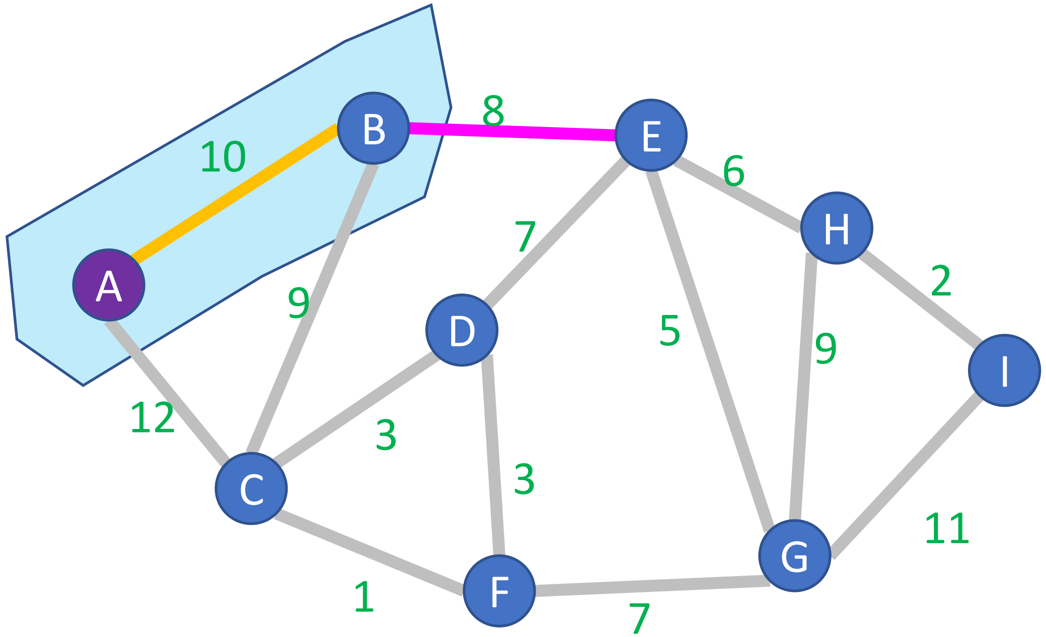
Prim's Algorithm

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



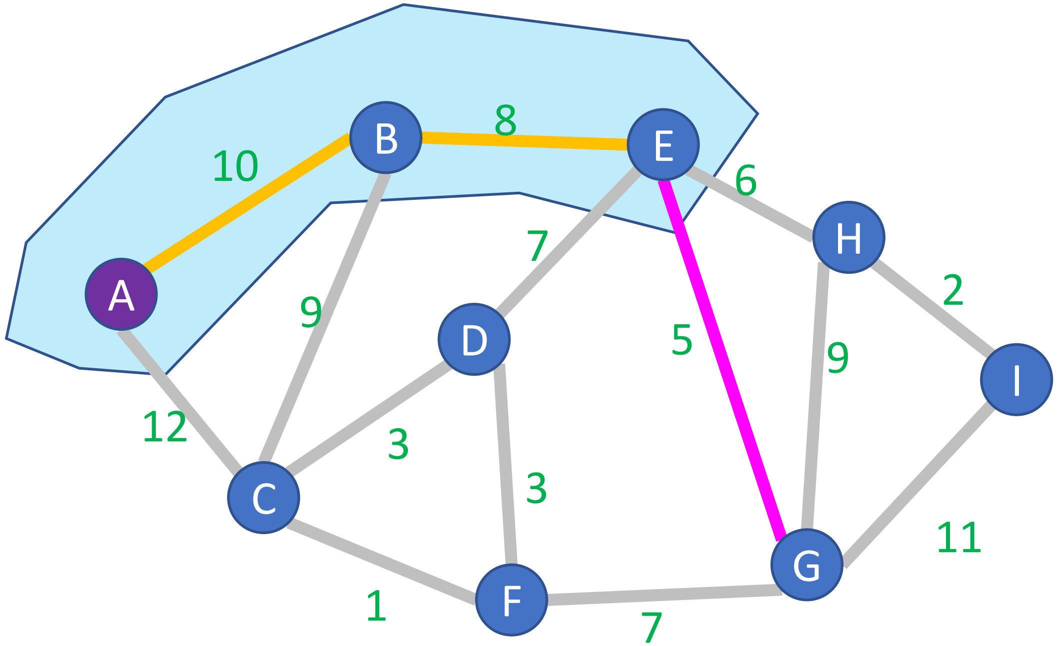
Prim's Algorithm

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



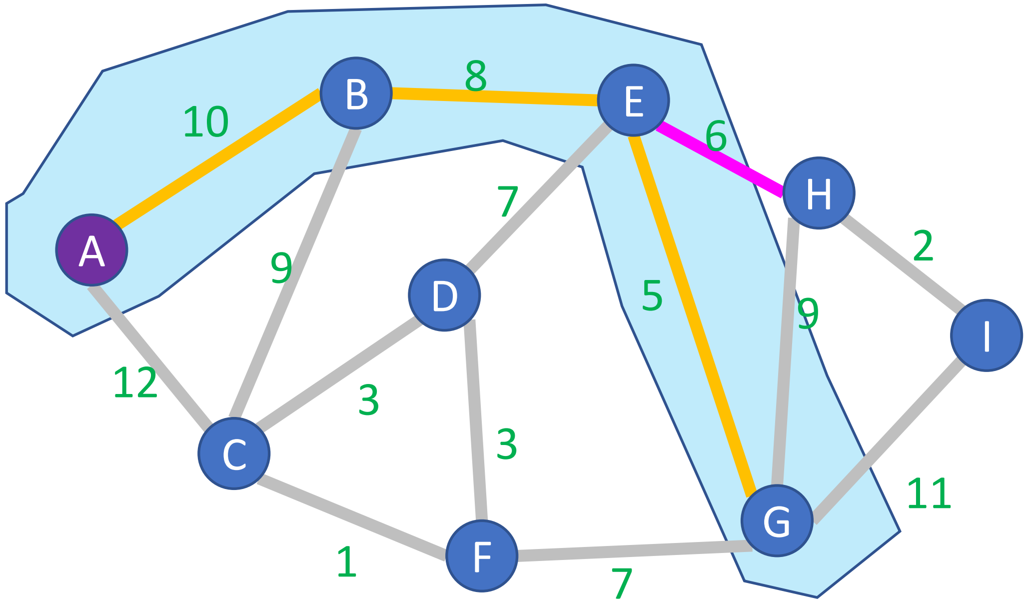
Prim's Algorithm

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



Prim's Algorithm

Start with an empty tree A

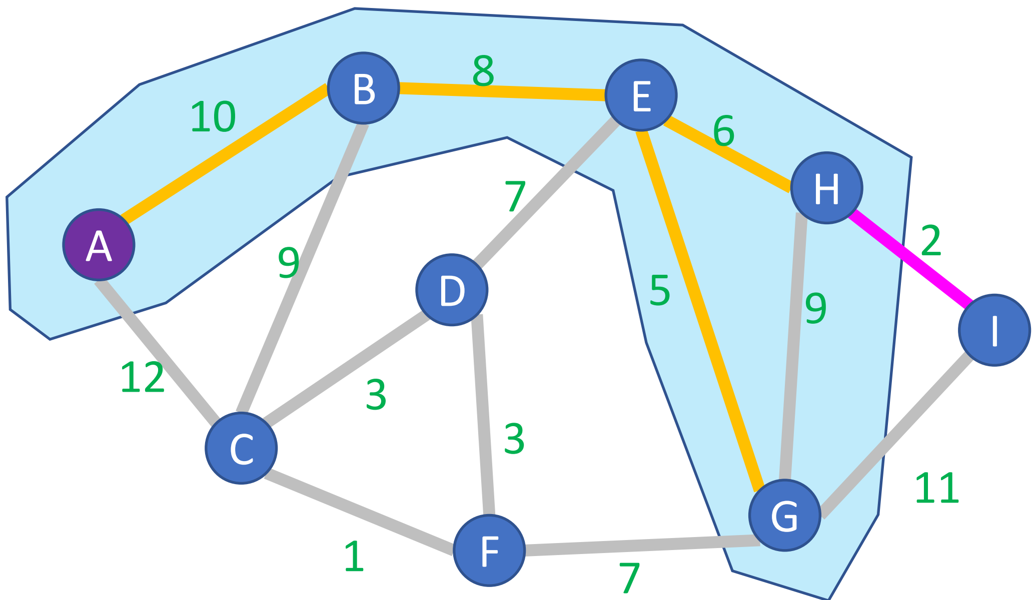
Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A

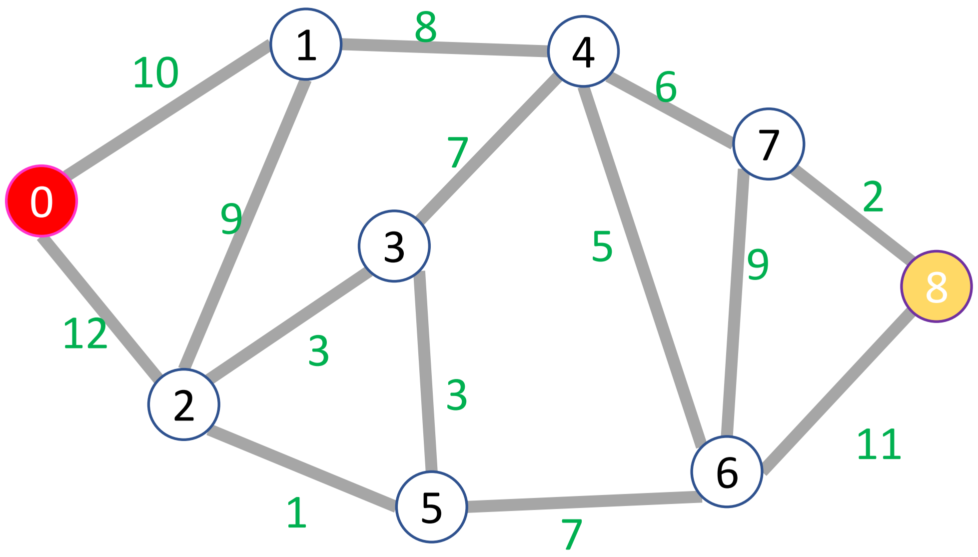
Keep edges in a Heap

$O(E \log V)$



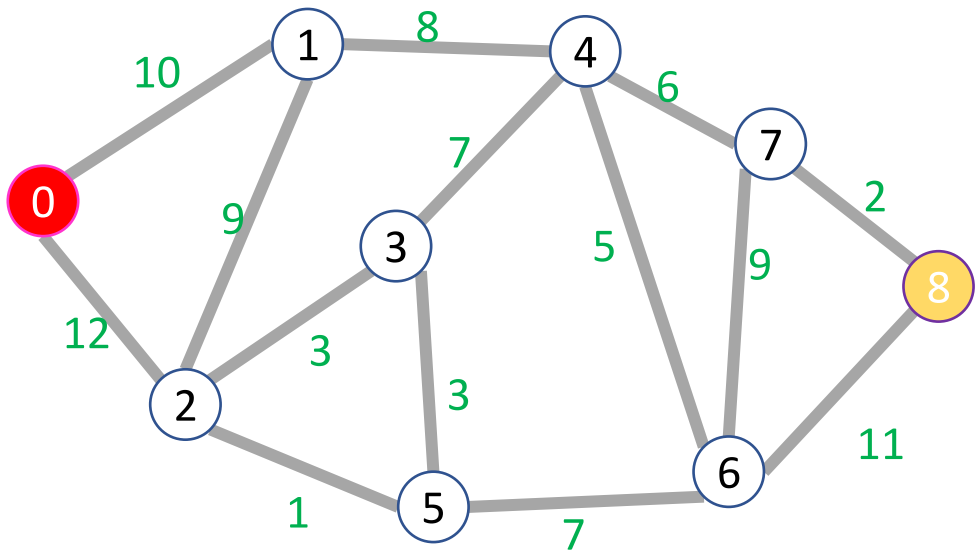
Dijkstra’s Algorithm

```
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]
}
```



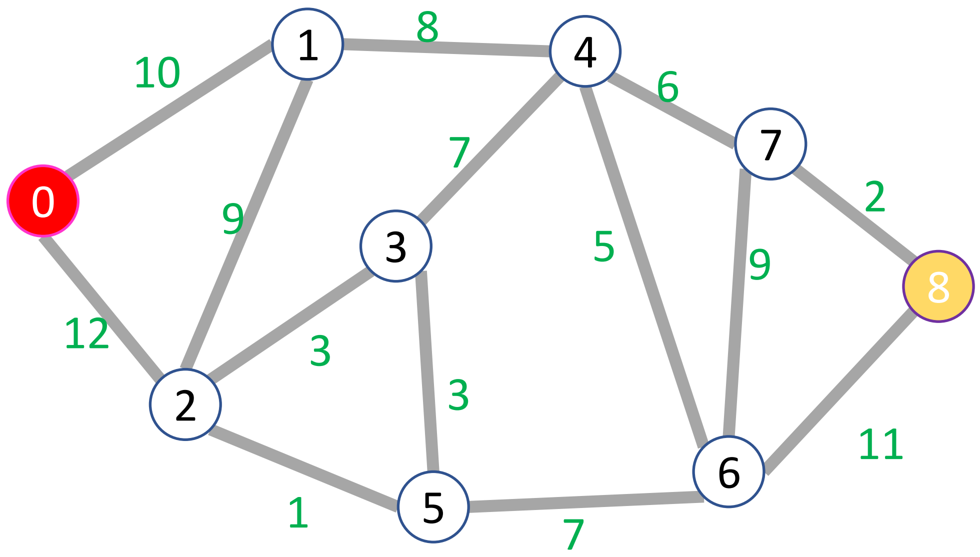
Prims's Algorithm

```
int primss(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 = 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]
}
```



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]
}
```



Prims's Algorithm

```
int primss(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 = 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]
}
```