# CSE 332 Winter 2026
# Lecture 15: Graphs 2

Nathan Brunelle

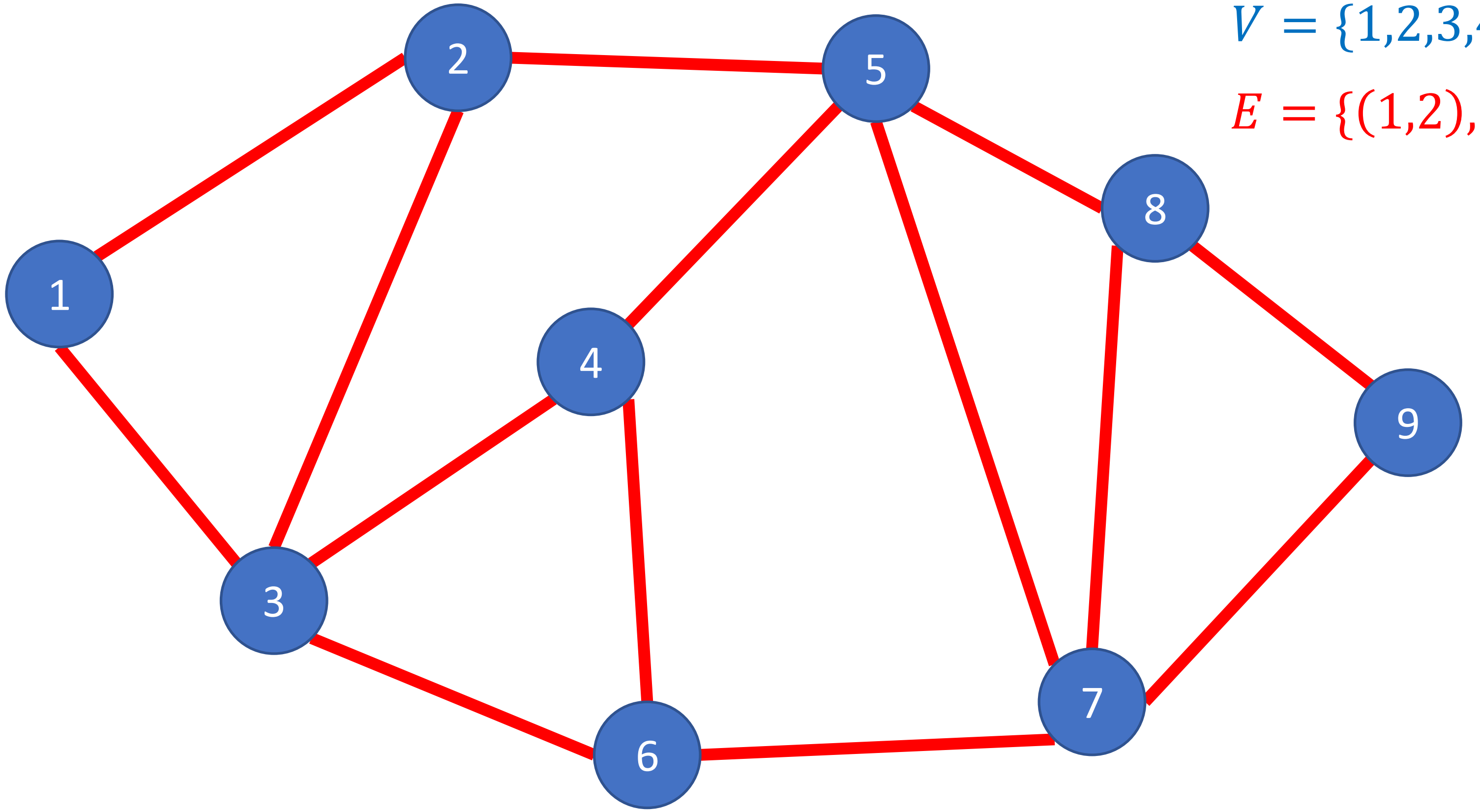
http://www.cs.uw.edu/332

# Undirected Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges




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

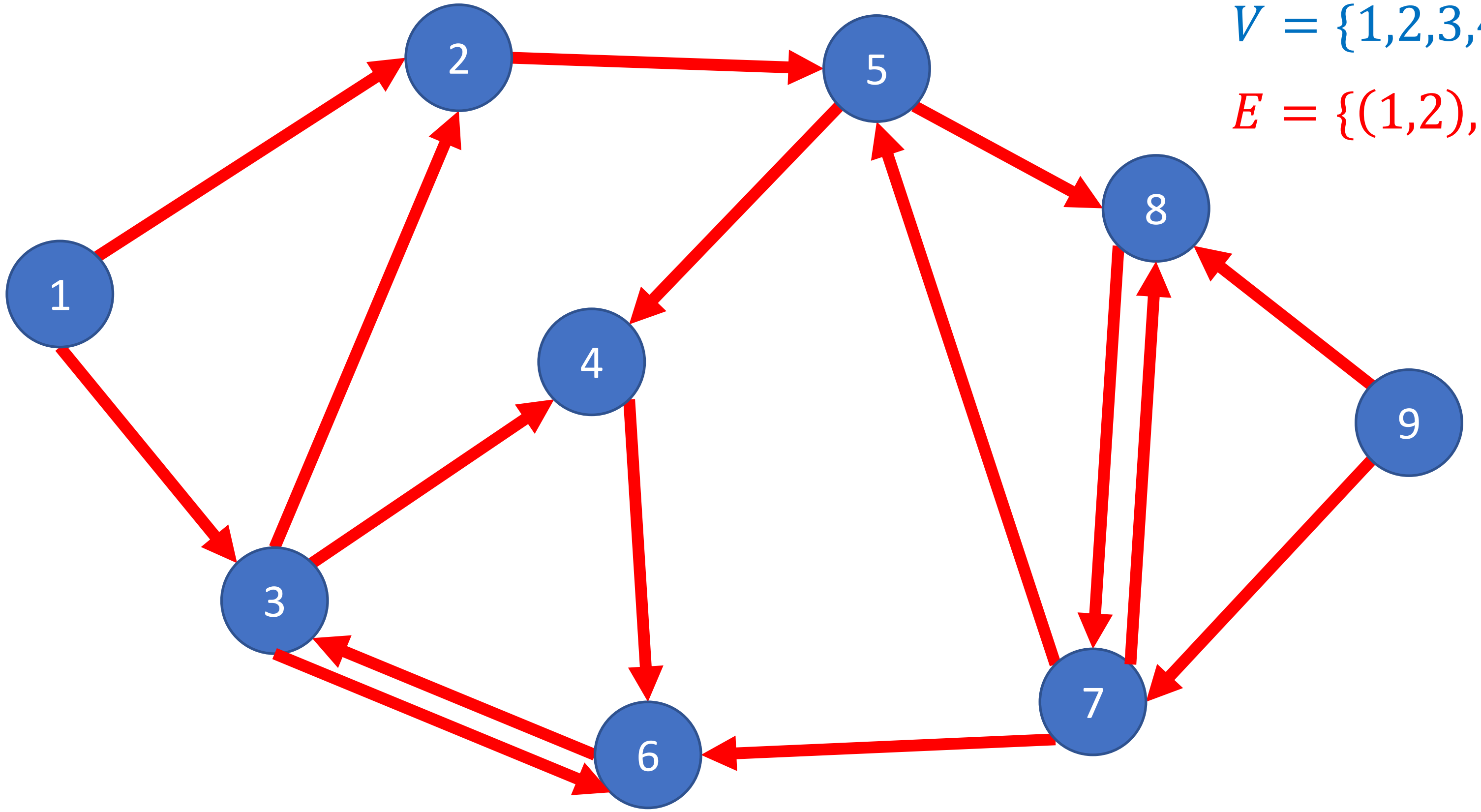
$E = \{(1,2), (2,3), (1,3), \dots\}$

# Directed Graphs

Vertices/Nodes

$$\text{Definition: } G = (V, E)$$

Edges




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

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

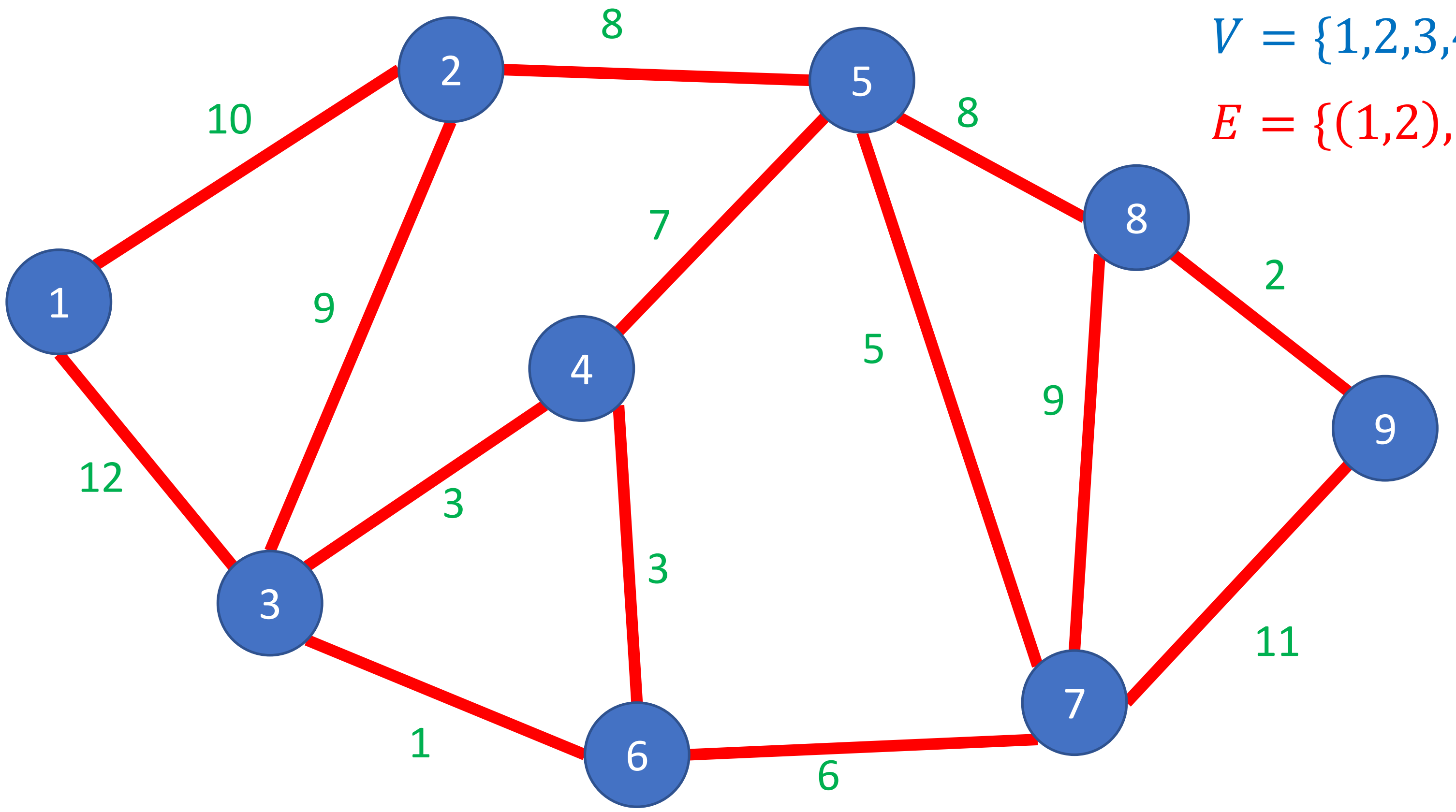
# Weighted Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges

$w(e)$ = weight of edge $e$

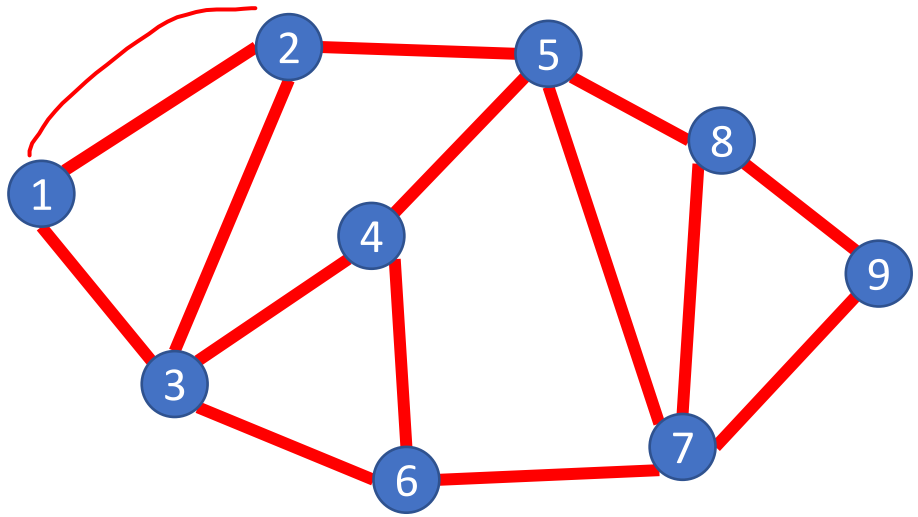


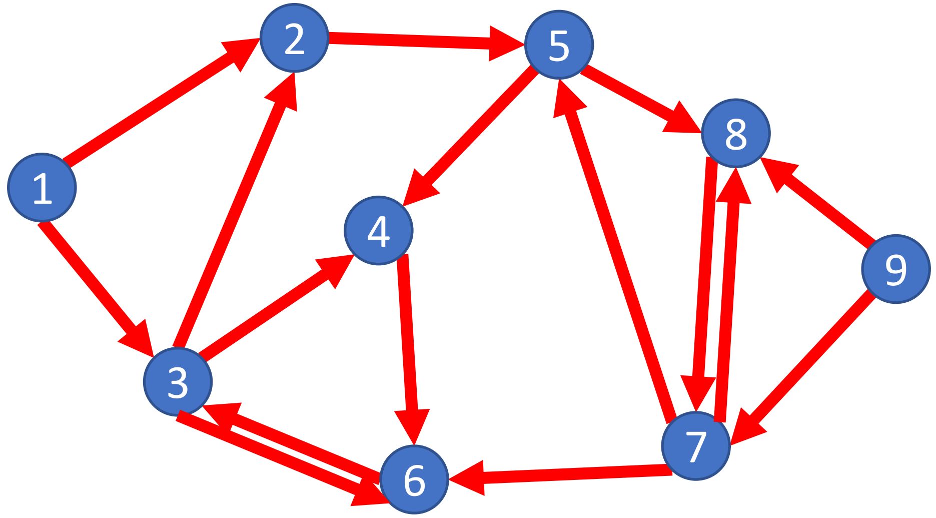

$V = \{1,2,3,4,5,6,7,8,9\}$

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

# Some Graph Terms

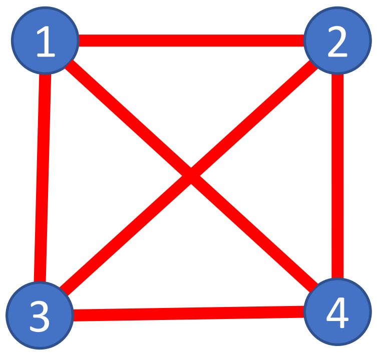
- Adjacent/Neighbors
  - Nodes are adjacent/neighbors if they share an edge
- Degree
  - Number of edges “touching” a vertex
- Indegree
  - Number of incoming edges
- Outdegree
  - Number of outgoing edges



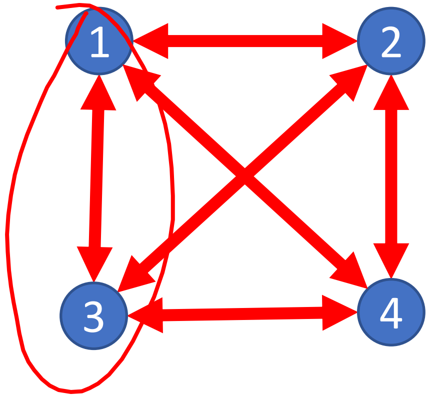



# Definition: Complete Graph

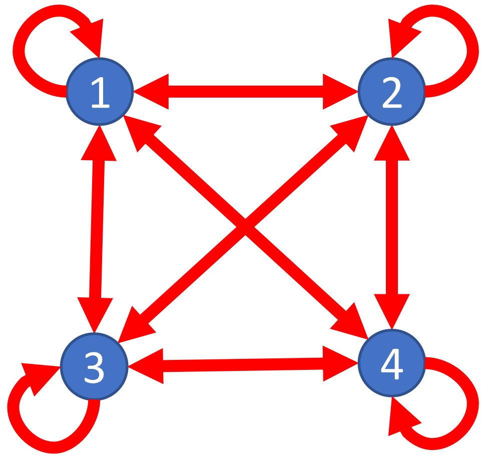
A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is an edge from $v_1$ to $v_2$




Complete Undirected Graph

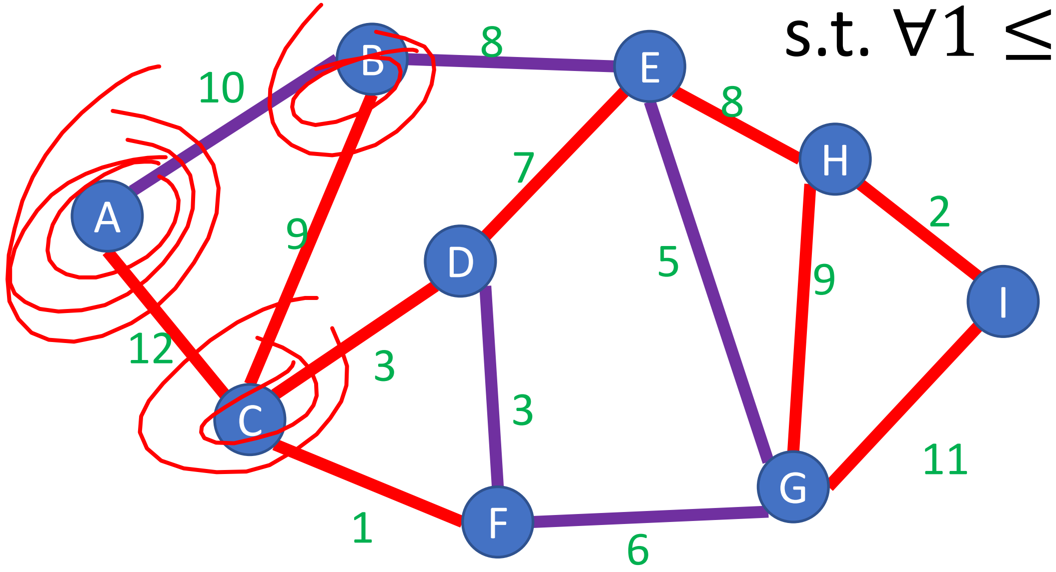



Complete Directed Graph




Complete Directed Non-simple Graph

# Definition: Path



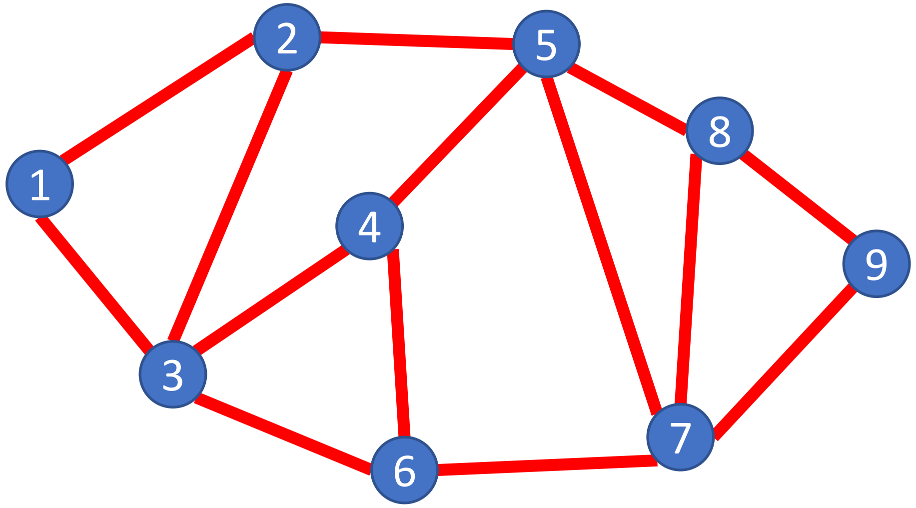

A sequence of nodes $(v_1, v_2, \ldots, v_k)$
s.t. $\forall 1 \leq i \leq k-1, (v_i, v_{i+1}) \in E$

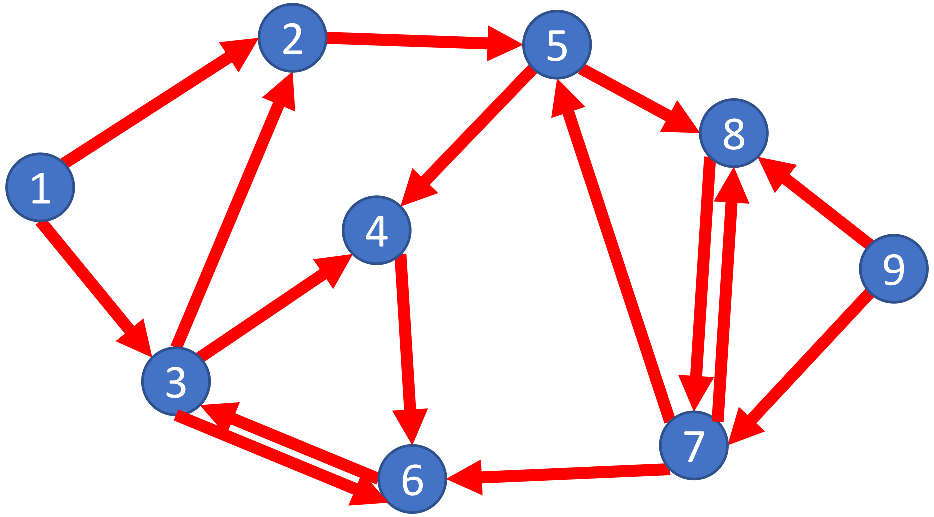
<u>Simple Path:</u>
A path in which each node appears at most once

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

# Definition: (Strongly) Connected Graph

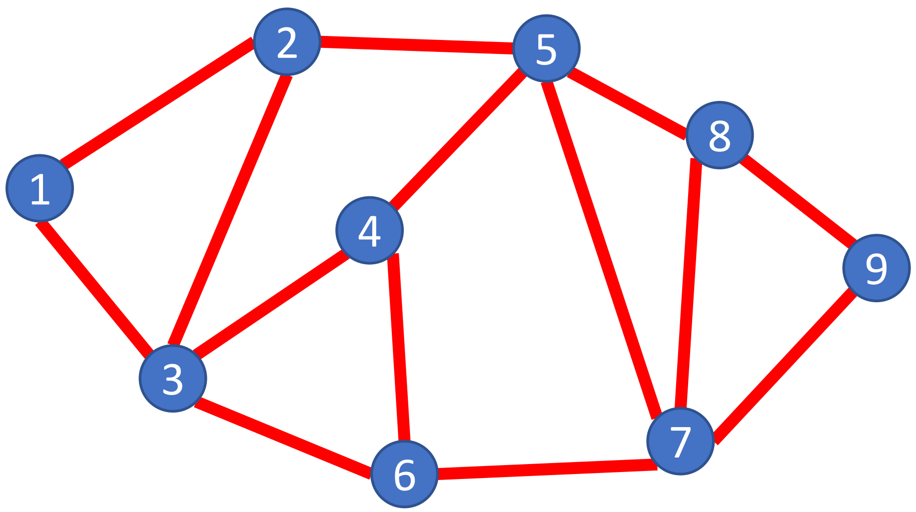
A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$



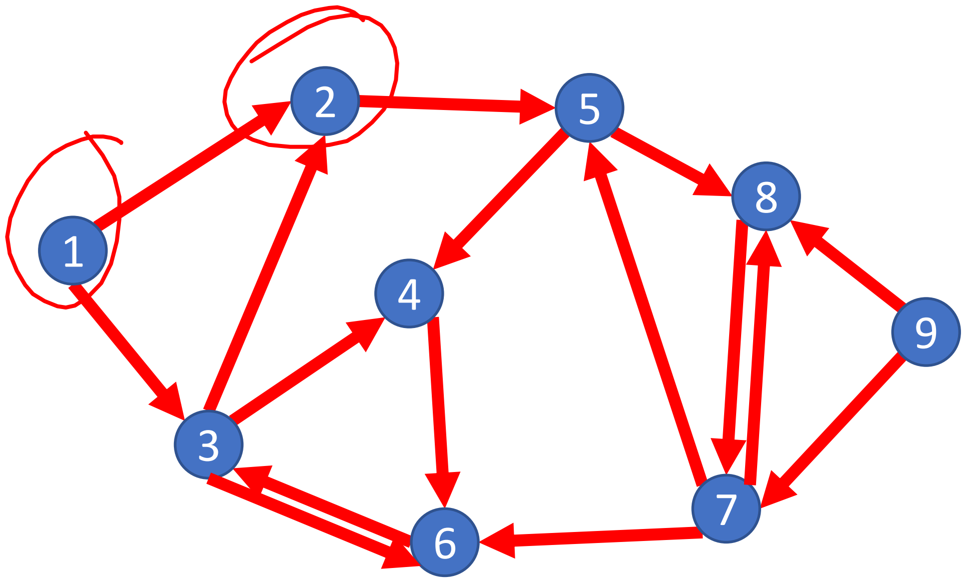



# Definition: (Strongly) Connected Graph

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 $v_2$



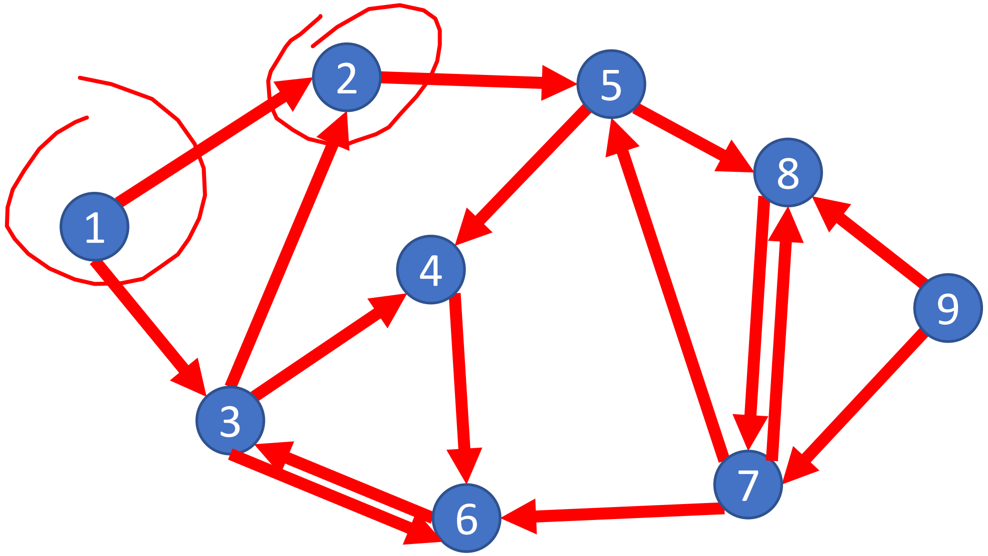

Connected



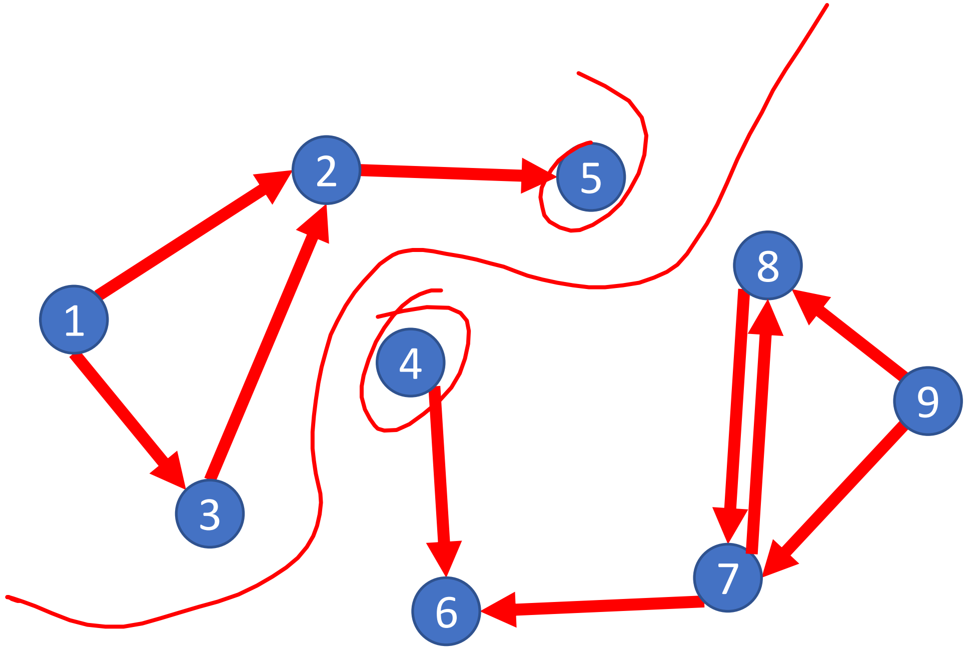

Not (strongly) Connected

# 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



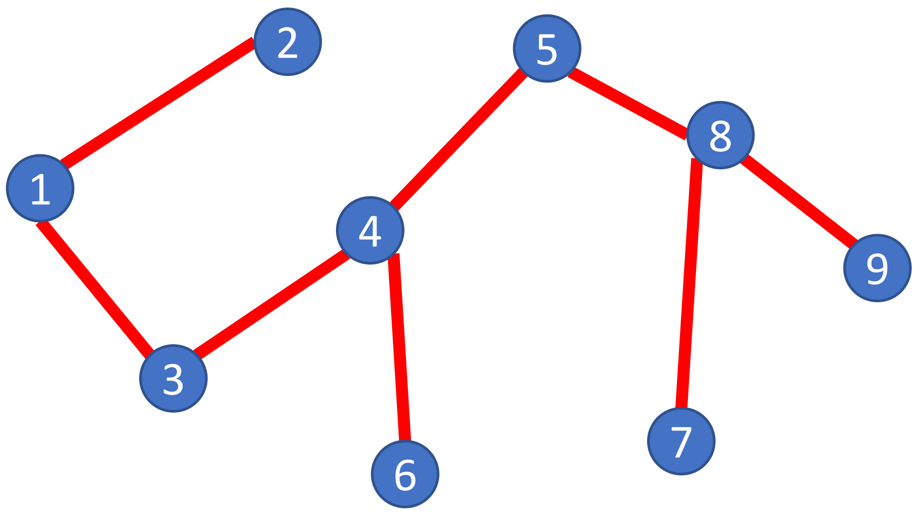

Not Weakly Connected

# Graph Density, Data Structures, Efficiency

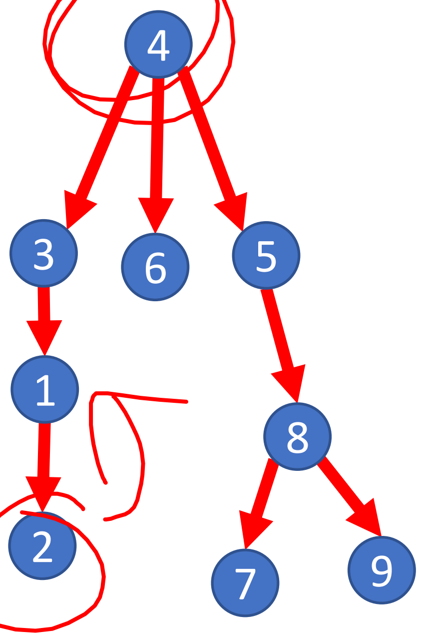
- The maximum number of edges in a graph is $\Theta(|V|^2)$:
  - Undirected and simple: $\frac{|V|(|V|-1)}{2}$
  - Directed and simple: $|V|(|V|-1)$
  - Direct and non-simple (but no duplicates): $|V|^2$
- If the graph is connected, the minimum number of edges is $|V|-1$
- If $|E| \in \Theta(|V|^2)$ 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 undirected, connected, and has no cycles (i.e. is acyclic). Often one node is identified as the "root"




A Tree

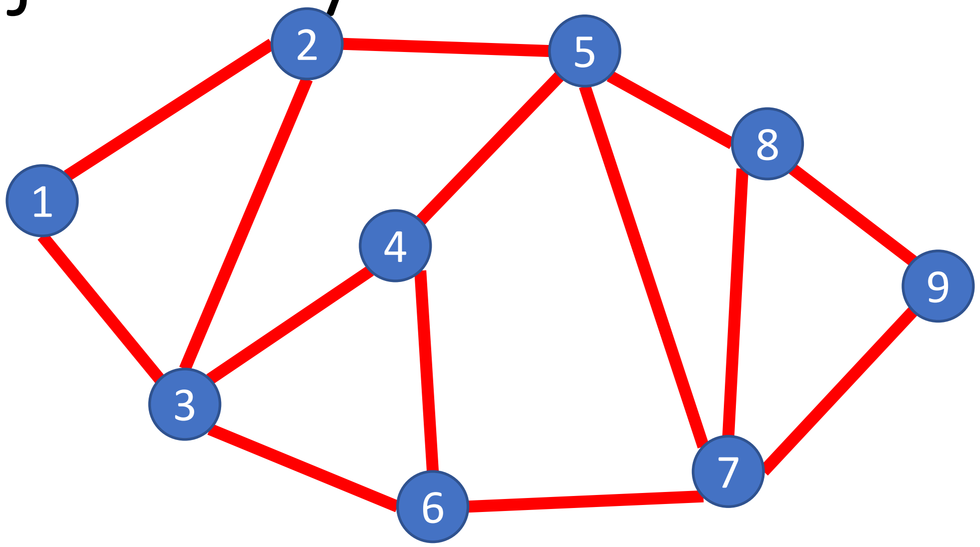



A Rooted Tree

# Graph ADT

- Idea: Nodes with edges between them
  - Directed or undirected
  - Weighted or unweighted
- Operations we'll need:
  - addEdge: an in a new edge between preexisting nodes
  - removeEdge: remove an edge
  - exists: Check if a particular edge exists
  - getNeighbors: give a list of all neighbors of a given node
    - For a directed graph, we also might want getNeighborsIncoming

# Adjacency List Data Structure




| | | | | |
|---|---|---|---|---|
| 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 | | |

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

<u>Time/Space Tradeoffs</u>

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

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

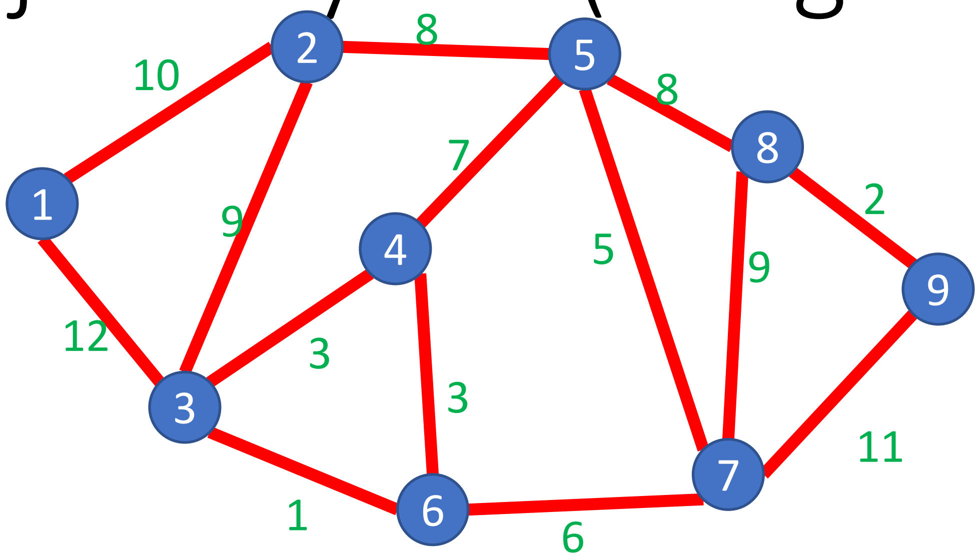
Remove Edge $(v, w)$: $\Theta(\deg(v))$

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

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

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

# Adjacency List (Weighted)




| | | | | |
|---|---|---|---|---|
| 1 | 2 (10) | 3 (12) | | |
| 2 | 1 (10) | 3 (9) | 5 (8) | |
| 3 | 1 (12) | 2 (9) | 4 (3) | 6 (1) |
| 4 | 3 (3) | 5 (7) | 6 (3) | |
| 5 | 2 (8) | 4 (7) | 7 (5) | 8 (8) |
| 6 | 3 (1) | 4 (3) | 7 (6) | |
| 7 | 5 (5) | 6 (6) | 8 (9) | 9 (11) |
| 8 | 5 (8) | 7 (9) | 9 (2) | |
| 9 | 7 (11) | 8 (2) | | |

<u>Time/Space Tradeoffs</u>

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

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

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

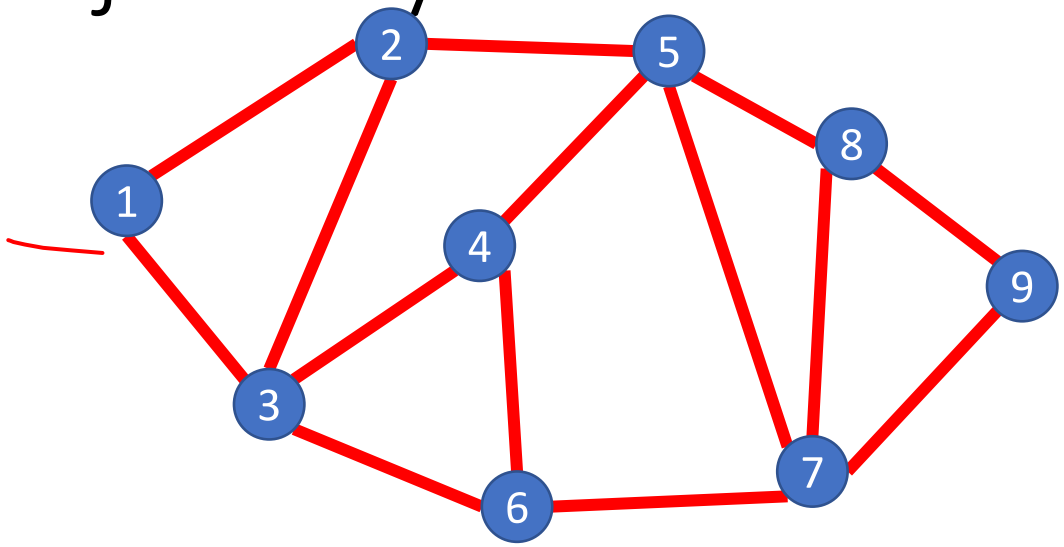
Check if Edge $(v,w)$ Exists: $\Theta(\deg(v))$

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

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

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

# Adjacency Matrix




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

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

<u>Time/Space Tradeoffs</u>

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

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

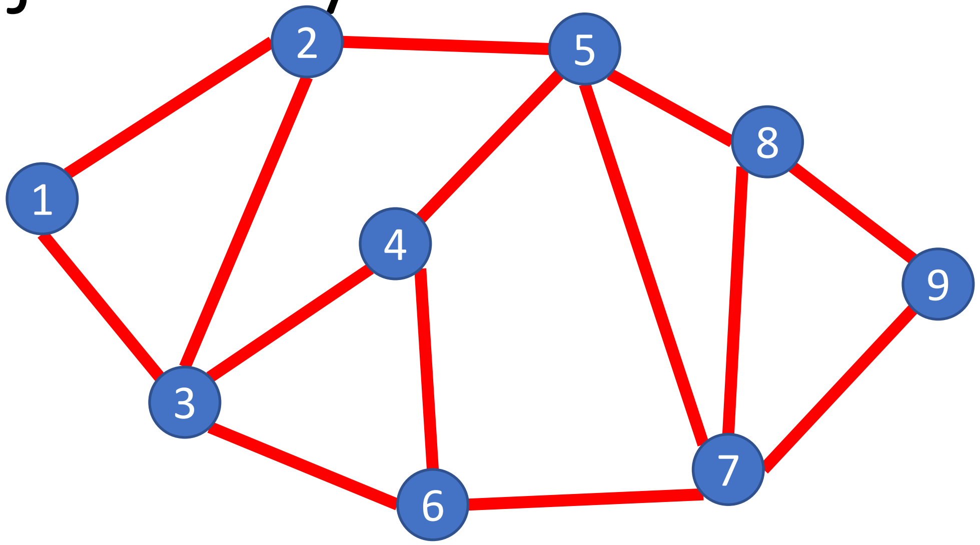
Remove Edge $(v, w)$: $\Theta(?)$ 1

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

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

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

# Adjacency Matrix




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

<u>Time/Space Tradeoffs</u>

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

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

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

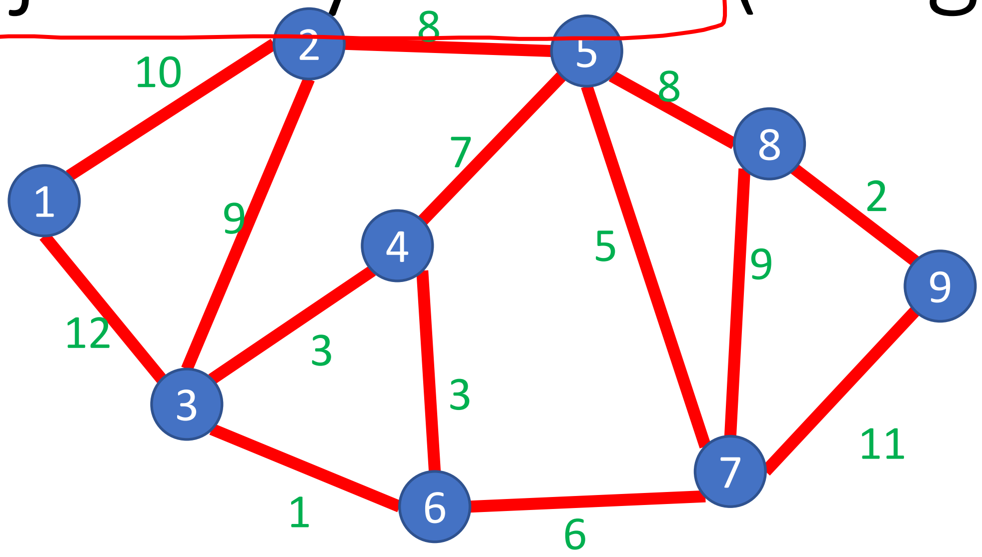
Check if Edge $(v, w)$ Exists: $\Theta(1)$

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

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

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

# Adjacency Matrix (weighted)




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | 10 | 12 | | | | | | |
| 2 | 10 | | 9 | | 8 | | | | |
| 3 | 12 | 9 | | 3 | | 1 | | | |
| 4 | | | 3 | | 7 | 3 | | | |
| 5 | | 8 | | 7 | | | 5 | 8 | |
| 6 | | | 1 | 3 | | | 1 | | |
| 7 | | | | | 5 | 1 | | 9 | 11 |
| 8 | | | | | 8 | | 9 | | 2 |
| 9 | | | | | | | 11 | 2 | |

<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

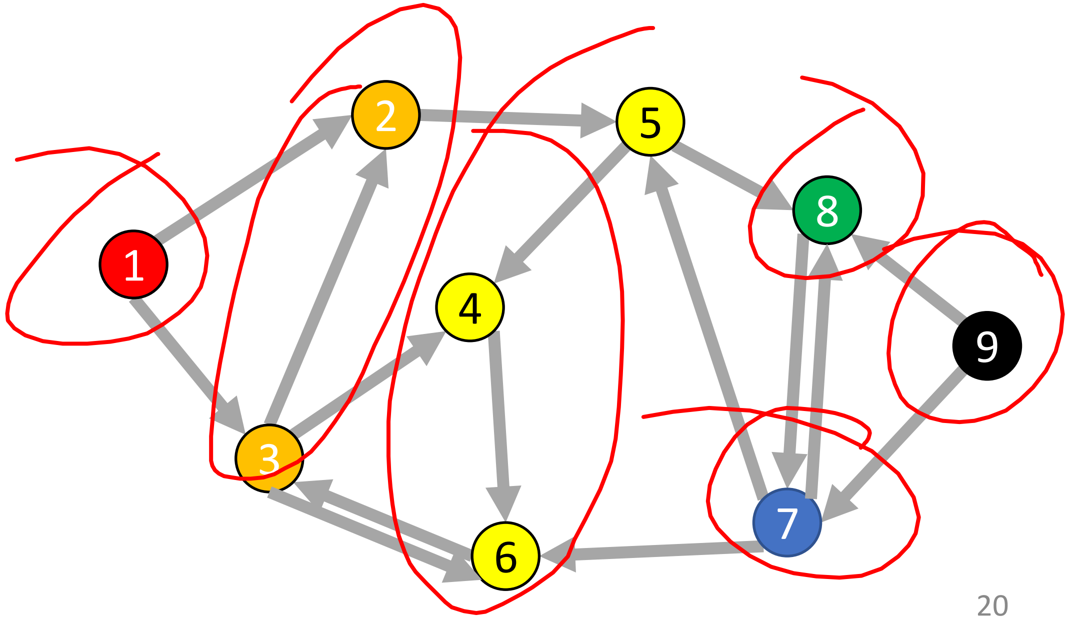
- Adjacency List:
  - Less memory when $|E| < |V|^2$
  - Operations with running time linear in degree of source node
    - Add an edge
    - Remove an edge
    - Check for edge
    - Get neighbors
- Adjacency Matrix:
  - Similar amount of memory when $|E| \approx |V|^2$
  - Constant time operations: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

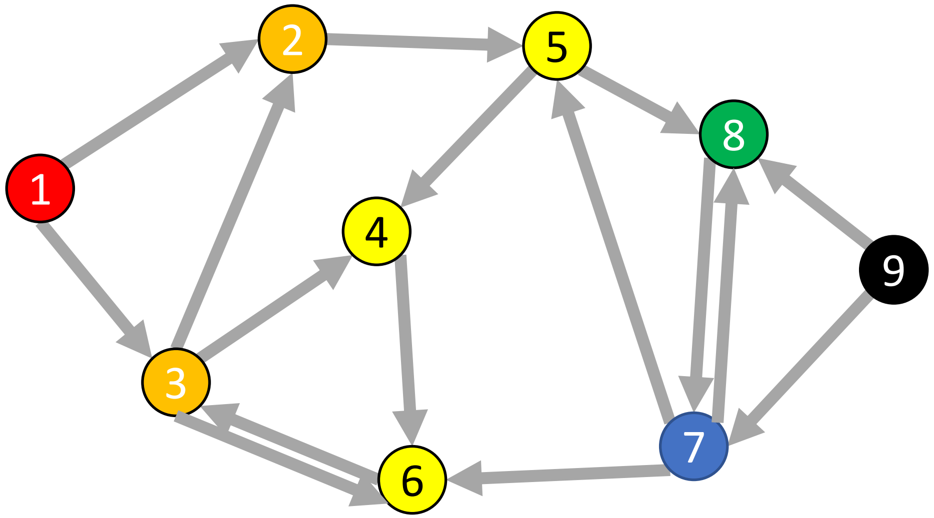
- Most graphs have $|E| \ll |V|^2$
  - Saves memory
  - Most nodes will have small degree
- Getting neighbors is a common operation
- Adjacency Matrix may be better if the graph is "dense" or if its edges change a lot

# 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

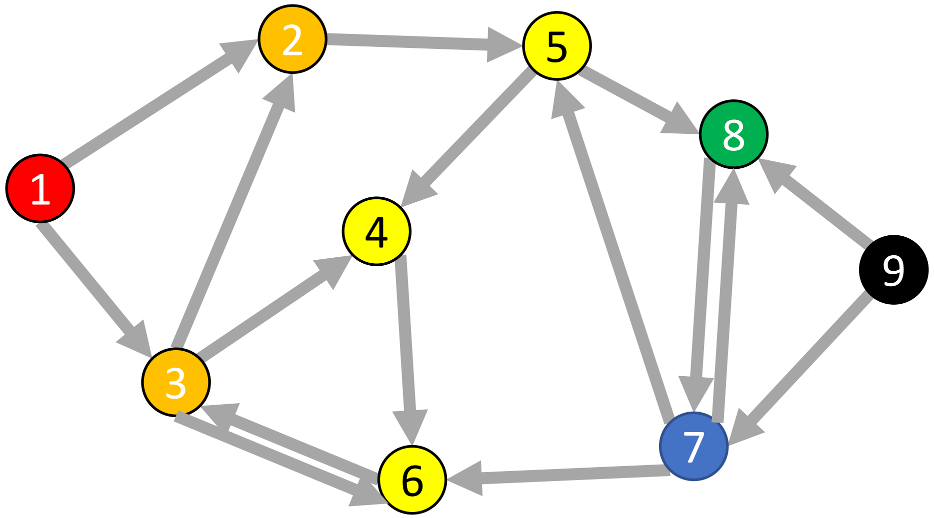


```
void bfs(graph, s){
        found = new Queue();
        found.enqueue(s);
        mark s as "visited";
        While (!found.isEmpty()){
                current = found.dequeue();
                for (v : neighbors(current)){
                        if (! v marked "visited"){
                  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

deg(v)

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

# BFS – Worked Example

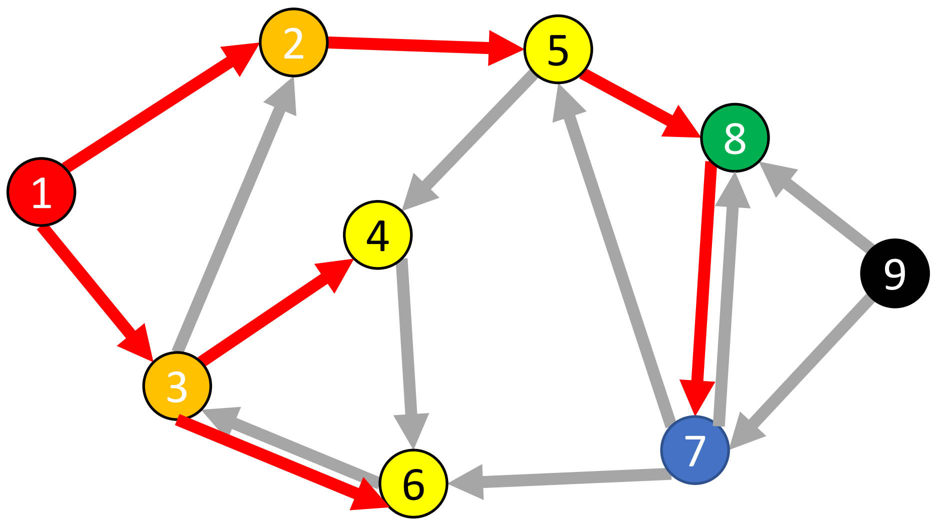



| Node | Visited? | Other Info |
|---|---|---|
| 1 | True | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |

For each node:
  For each unvisited neighbor:
    add that neighbor to a queue
    mark that neighbor as visited

Queue:

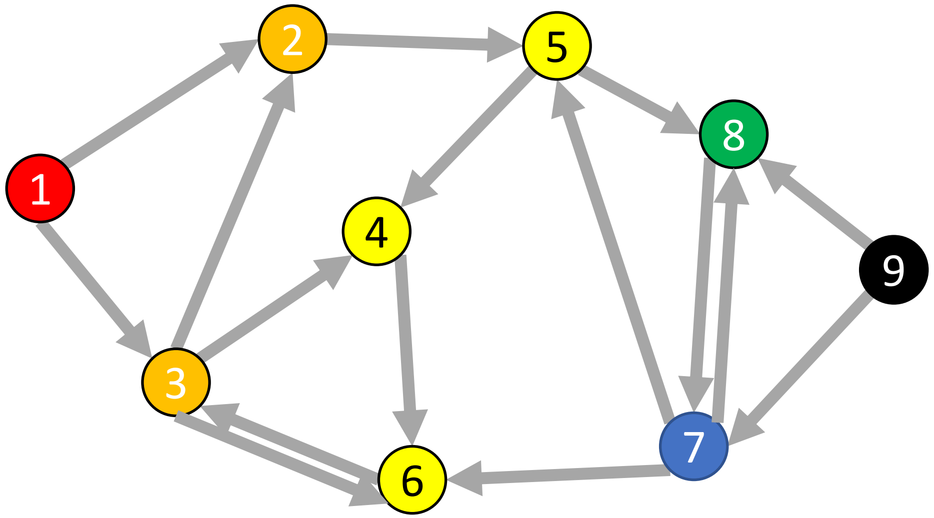
# Find Distance (unweighted)




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 – Worked Example

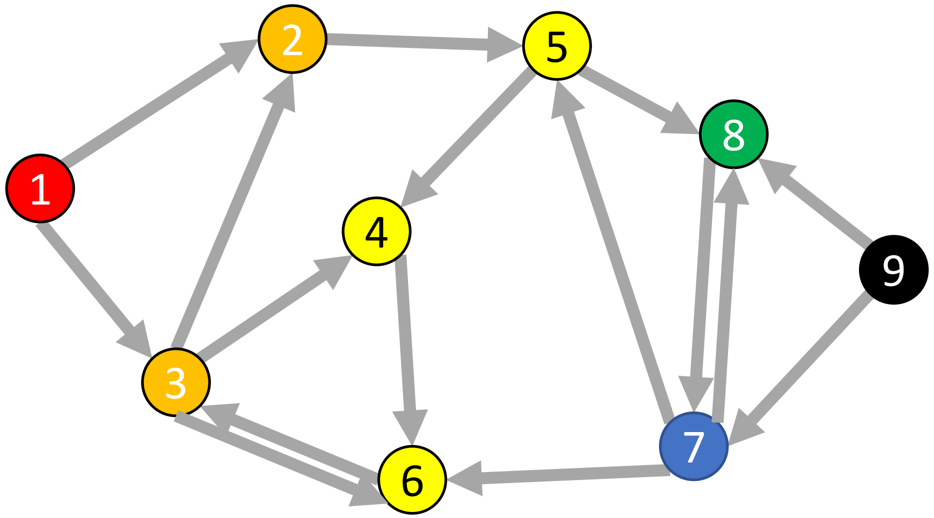



For each node:
- update current layer
- For each unvisited neighbor:
  - add that neighbor to a queue
  - mark that neighbor as visited
  - set neighbor's layer to be current layer + 1

| Node | Visited? | Layer |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |

Queue:

# Shortest Path - Idea




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

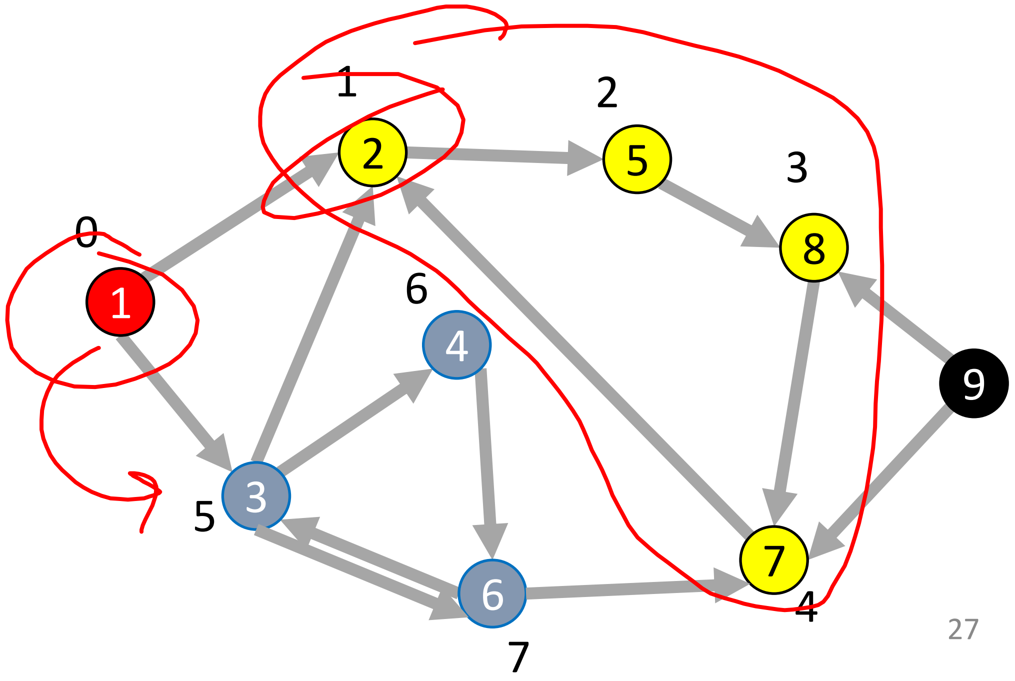
Queue:

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

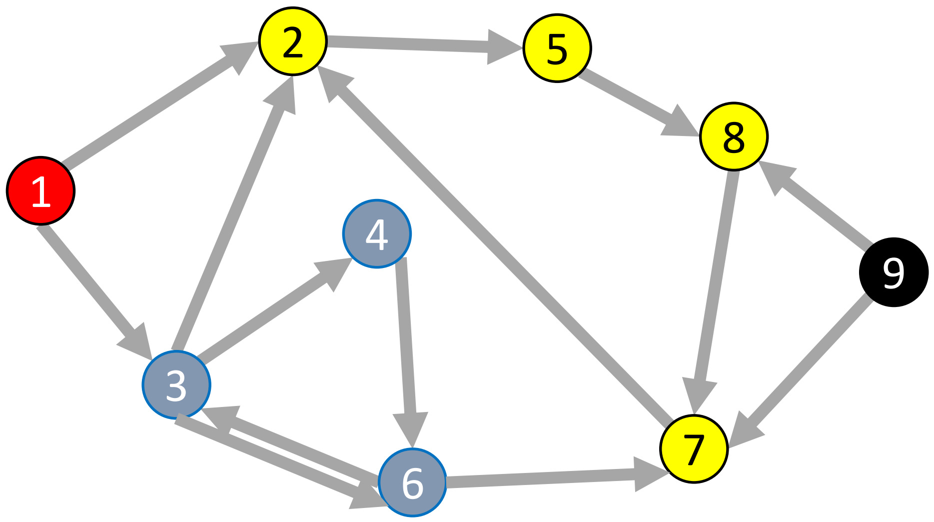
# Depth-First Search

# 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)



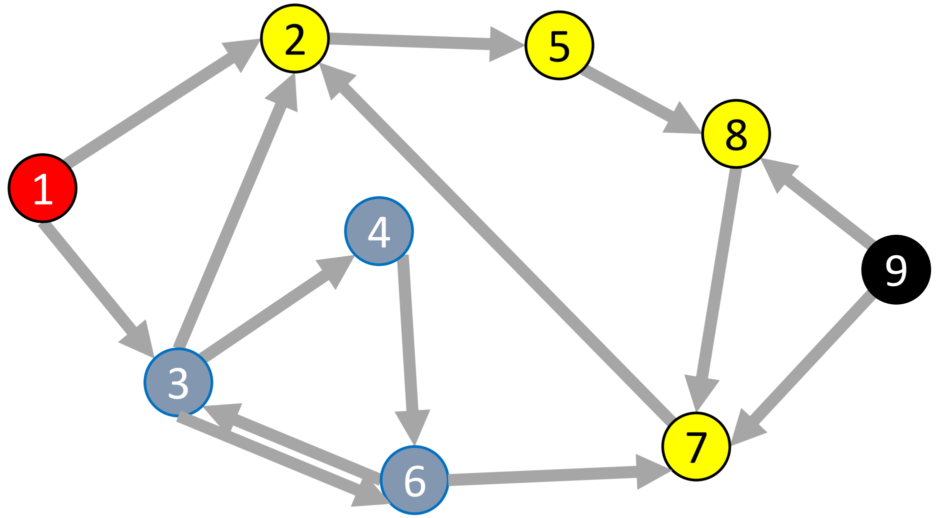

Running time: $\Theta(|V| + |E|)$

```
void dfs(graph, s){
        found = new Stack();
        found.~~pop~~ push(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);
                        }
                }
        }
}
```

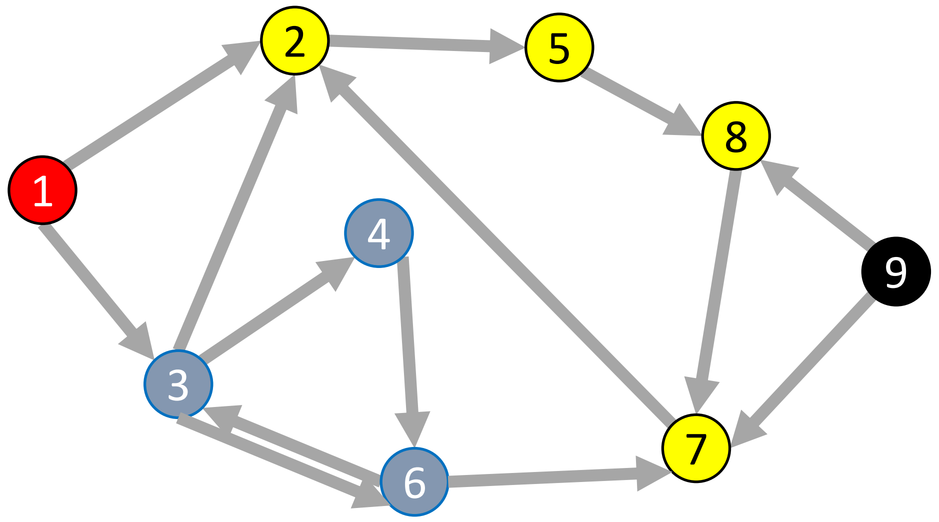
# DFS Recursively (more common)

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

|V|+|E|



# DFS – Worked Example



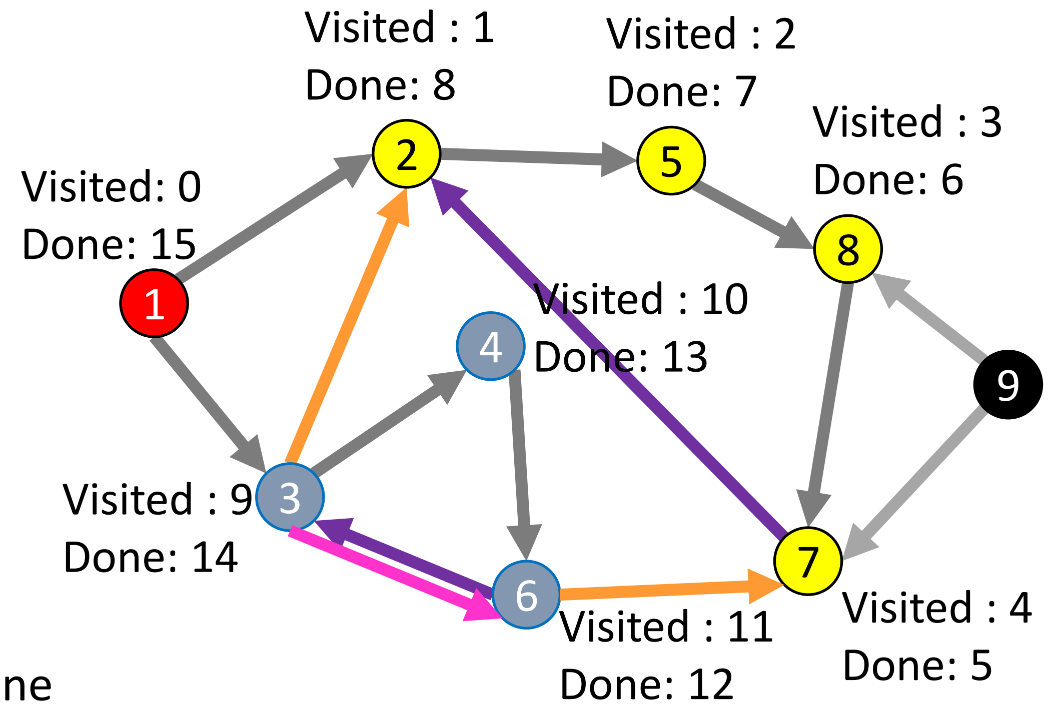

| Node | Visited? | Done? | Other Info |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |

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:

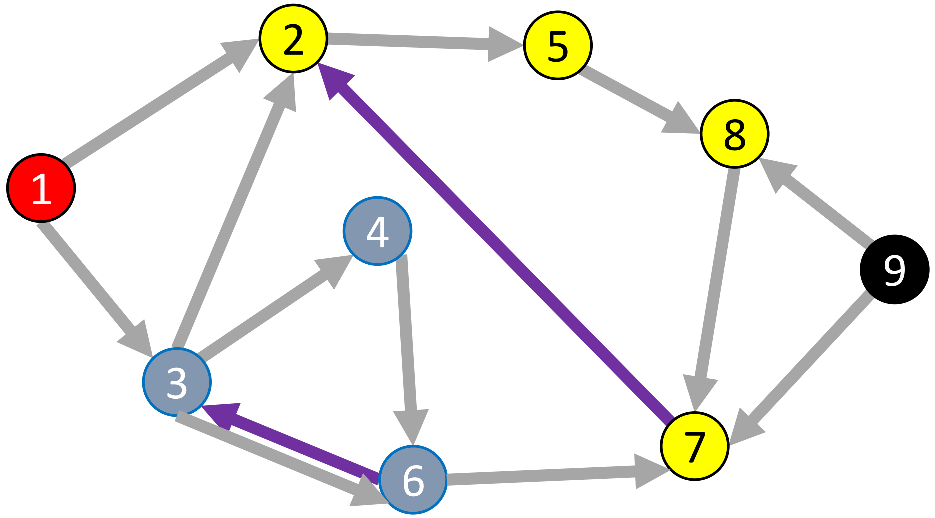
# Using DFS

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



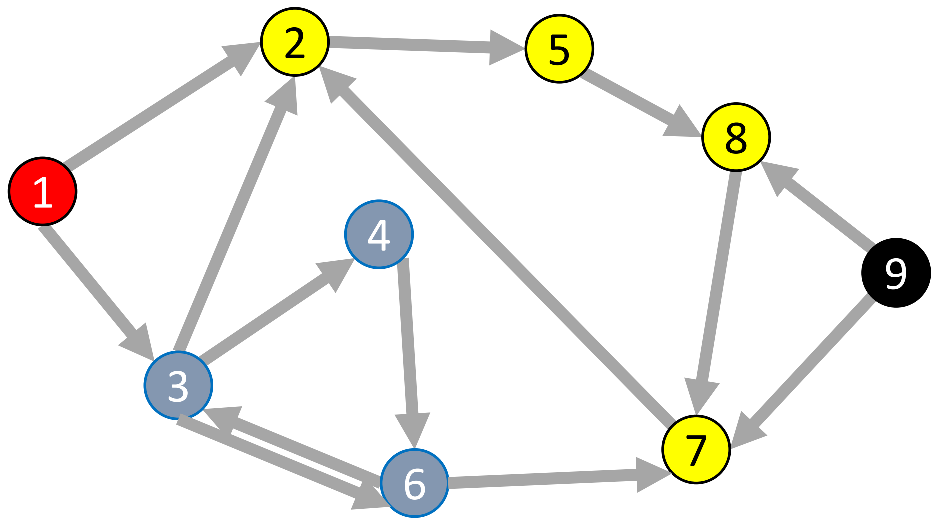
# Back Edges

- 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!**



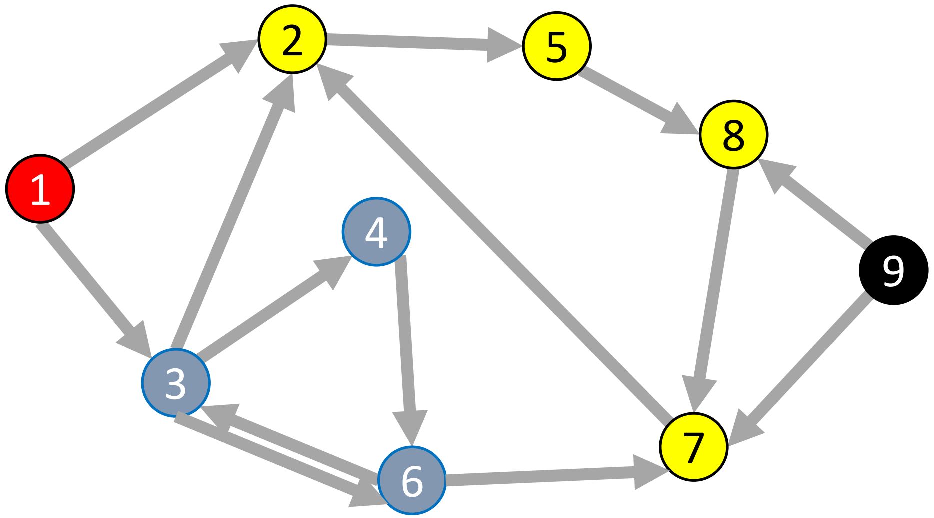
# Cycle Detection

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



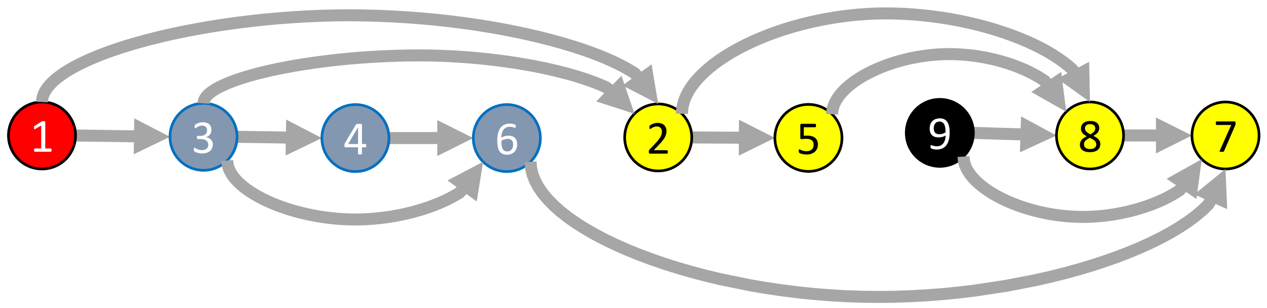

Starting from the current node:
  for each non-done neighbor:
    if the neighbor is visited:
      we found a cycle!
    else:
      mark the neighbor as visited
      do a DFS from the neighbor
  mark the current node as done

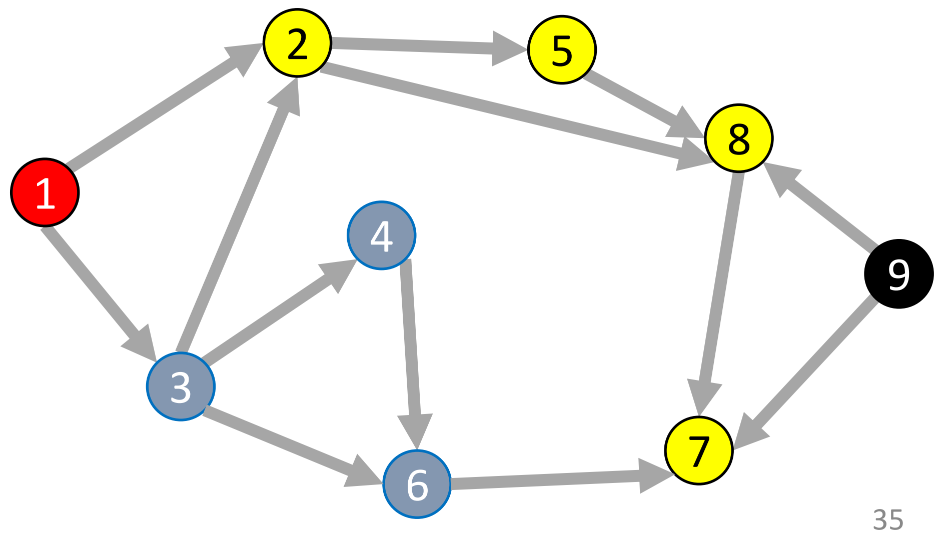
| Node | Visited? | Done? | Other Info |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |

(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

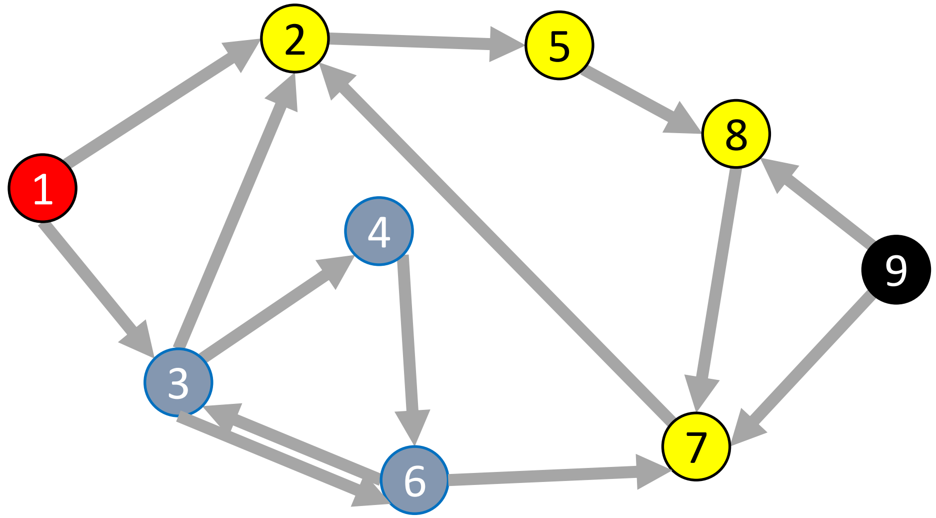


# DFS Recursively

```
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

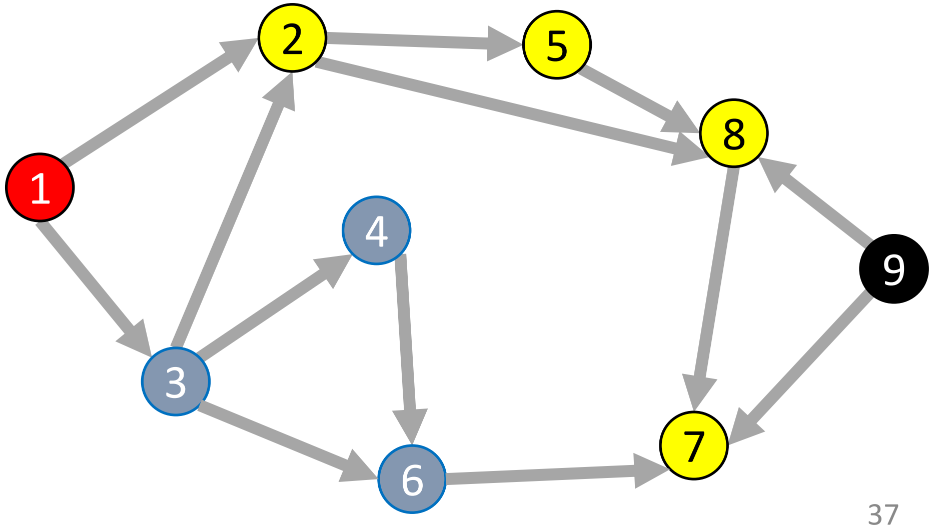
```
List topSort(graph){
        List<Nodes> done = new List<>();
        for (Node v : graph.vertices){
                if (!v.visited){
                        finishTime(graph, v, finished);
                }
        }
        done.reverse();
        return done;
}

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

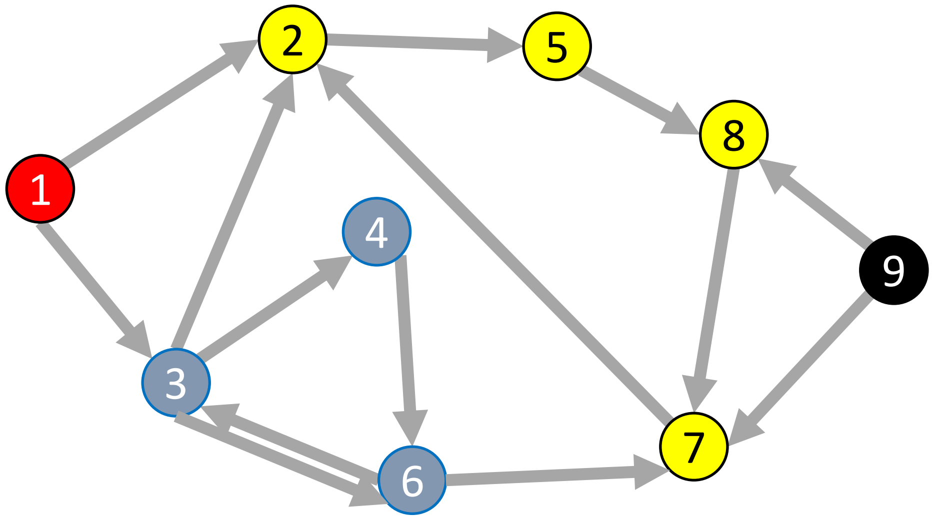
Idea: List in reverse order by "done" time

finished:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |



# 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:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |