CS3100 DSA2

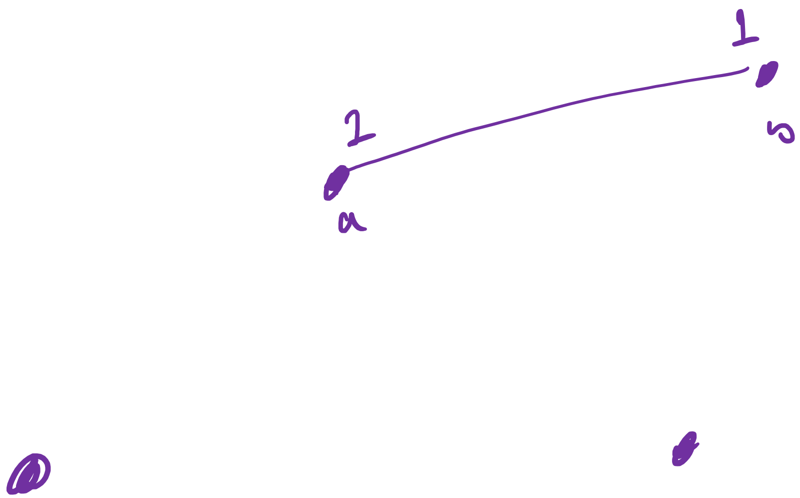
Spring 2024

Warm up:

Show that the sum of degrees of all nodes in any undirected graph is even

Show that for any graph $G = (V, E)$, $\sum_{v \in V} \deg(v)$ is even

$\sum_{v \in V} \deg(v)$ is even



(a, b)

$2E$

CS 3100
Data Structures and Algorithms 2
Lecture 4: Depth First Search

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

Readings in CLRS 4th edition:

- Chapter 20: Sections 20-3, 20-4, and 20-5

Announcements

- PS1 due tomorrow
- PA1 available, Gradescope submission coming later this week
- Office hours
    - Prof Hott Office Hours: Mondays 11a-12p, Fridays 10-11a and 2-3p
    - Prof Pettit Office Hours: Mondays and Wednesdays 2:30-4:00p
    - TA office hours posted on our website

Breadth First Search

Traversing Graphs

"Traversing" means processing each vertex edge in some organized fashion by following edges between vertices

- We speak of *visiting* a vertex. Might do something while there.

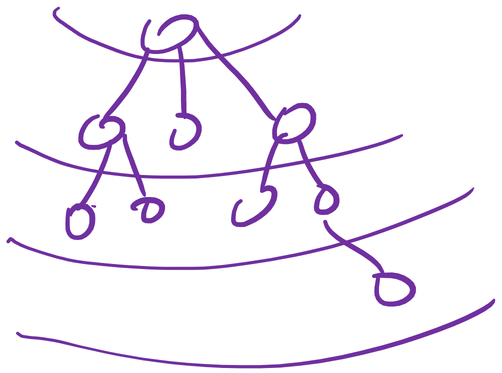
Recall traversal of binary trees:

- Several strategies: In-order, pre-order, post-order
- Traversal strategy implies an <u>order</u> of visits
- We used recursion to describe and implement these

Graphs can be used to model interesting, complex relationships

- Often traversal used just to process the set of vertices or edges
- Sometimes traversal can identify interesting properties of the graph
- Sometimes traversal (perhaps modified, enhanced) can answer interesting questions about the problem-instance that the graph models

BFS: Specific Input/Output



Input:

- A graph ***G***
- single start vertex ***s***

Output:

- Distance from ***s*** to each node in ***G*** (distance = number of edges)
- Breadth-First Tree of ***G*** with root ***s***

Strategy:

Start with node ***s***, visit all neighbors of ***s***, then all neighbors of neighbors of ***s***, …

Important: The paths in this BFS tree represent the **shortest paths** from s to each node in G

- But edge weight's (if any) not used, so "short" is in terms of number of edges in path

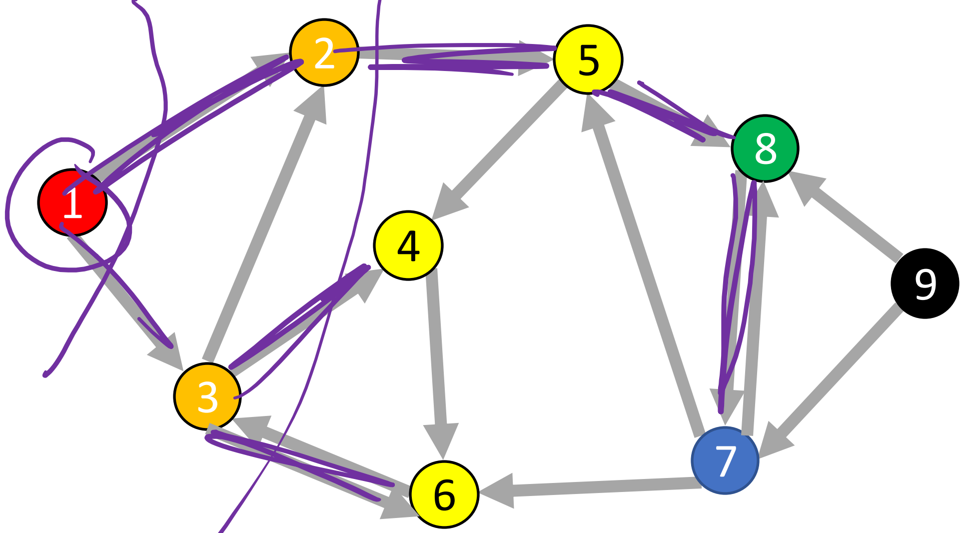
BFS

```
def bfs(graph, s):
        toVisit.enqueue(s)
        mark s as "seen"
        While toVisit is not empty:
                current = toVisit.dequeue()
                for v in neighbors(current):
                        if v not seen:
                                mark v as seen
                                toVisit.enqueue(v)
```

G , 1

toVisit Q :

Seen

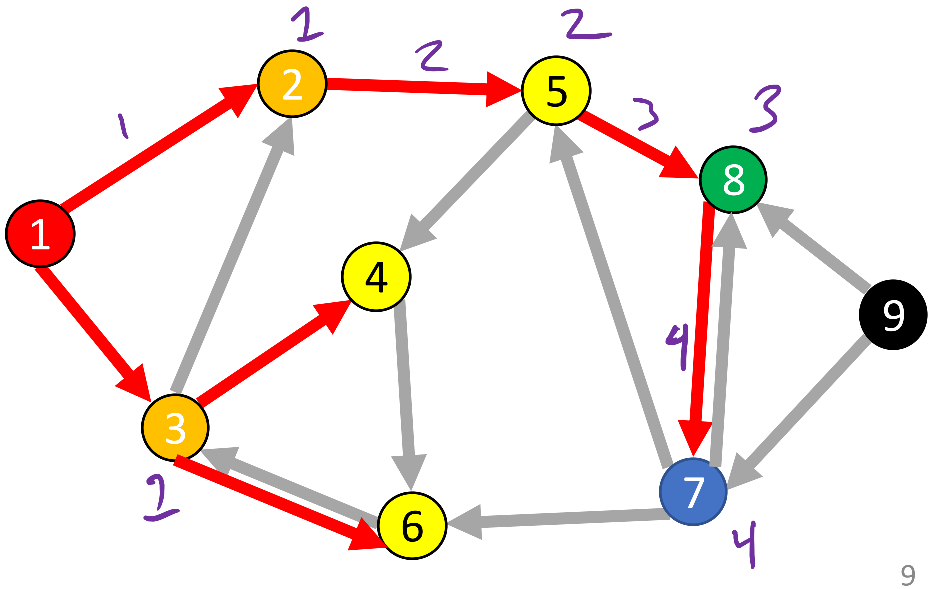


BFS: Shortest Path

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

```
                             1  7
def shortest_path(graph, s, t):
        toVisit.enqueue(s)              depth[] = [-1,-1,-1, ... -1]
        ~~mark s as "seen"~~
        depth[s] = 0
        While toVisit is not empty:
                current = toVisit.dequeue()
                layer = depth[current]
                for v in neighbors(current):
                        ~~if v not seen:~~ If depth[v] == -1:
                                ~~mark v as seen~~
                                depth[v] = layer + 1
                                toVisit.enqueue(v)

        return depth[t];
```

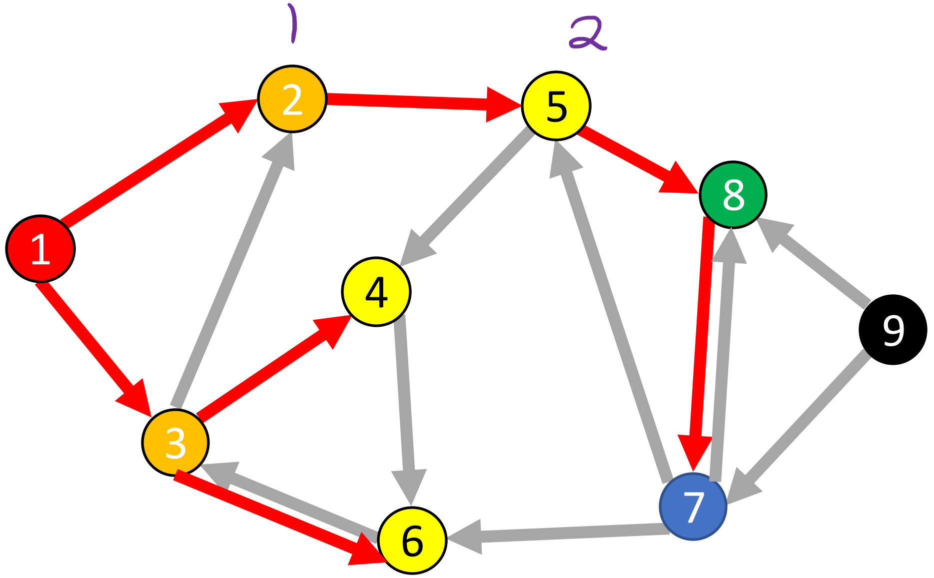


BFS: Shortest Path

```
def shortest_path(graph, s, t):
        toVisit.enqueue(s)
        depth[s] = 0
        While toVisit is not empty:
                current = toVisit.dequeue()
                layer = depth [current]
                for v in neighbors(current):
                        if v does not have a depth:
                                depth[v]=layer+1
                                toVisit.enqueue(v)
        return depth[t]
```

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

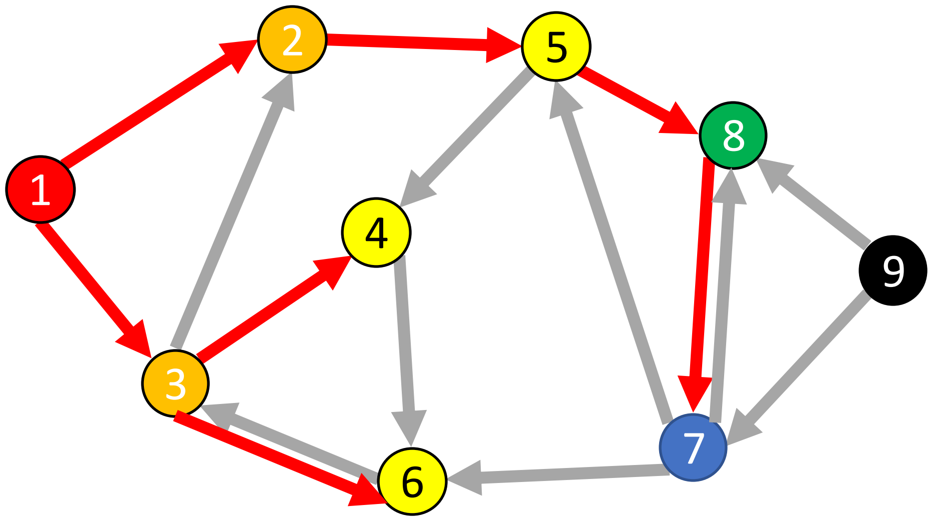
depth[5] = 2 (depth)
parent[5] = 2 (node)



BFS: Shortest Path

```
def shortest_path(graph, s, t):
        layer = 0
        depth = [-1,-1,-1,…]  # Length matches |V|
        toVisit.enqueue(s)
        mark a as "seen"
        depth[s] = 0
        While toVisit is not empty:
                current = toVisit.dequeue()
                layer = depth[current]
                if current == t:
                        return layer
                for v in neighbors(current):
                        if v not seen:
                                mark v as seen
                                toVisit.enqueue(v)
                                depth[v] = layer + 1
```

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



Breadth-first search from CLRS 20.2

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

From CLRS

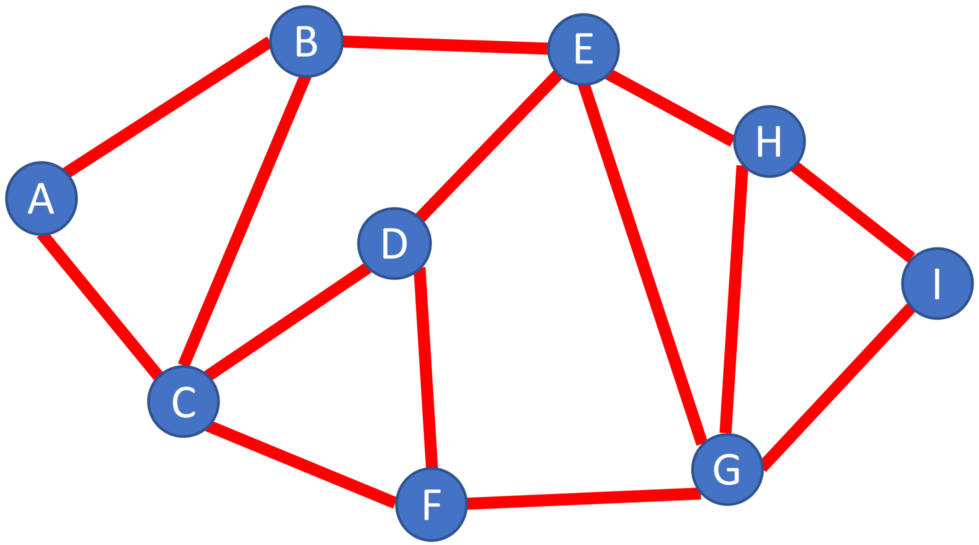
Vertices here have some properties:

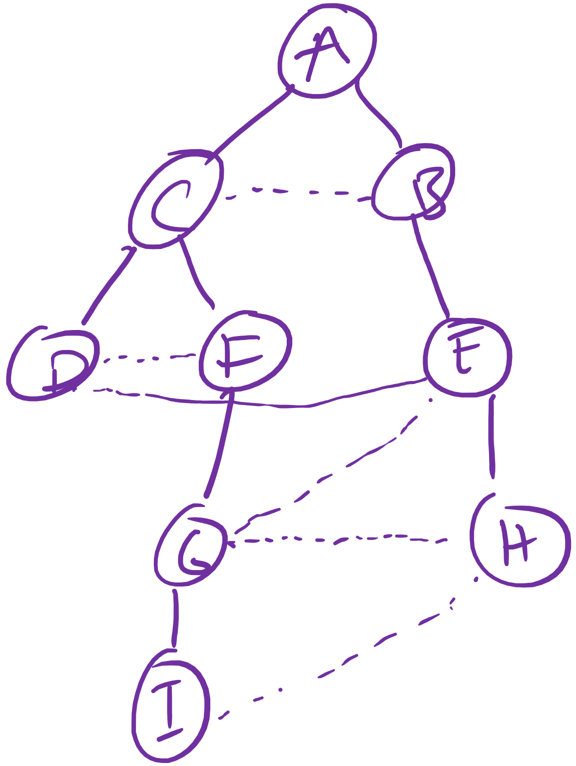
- *color = white/gray/black*
- *d = distance from start node*
- *pi = parent in tree, i.e. v.pi is vertex by which v was* connected to BFS tree

Color meanings here:

- White: haven't seen this vertex yet
- Gray: vertex has been seen and added to the queue for processing later
- Black: vertex has been removed from queue and its neighbors seen and added to the queue

Tree View of BFS Search Results

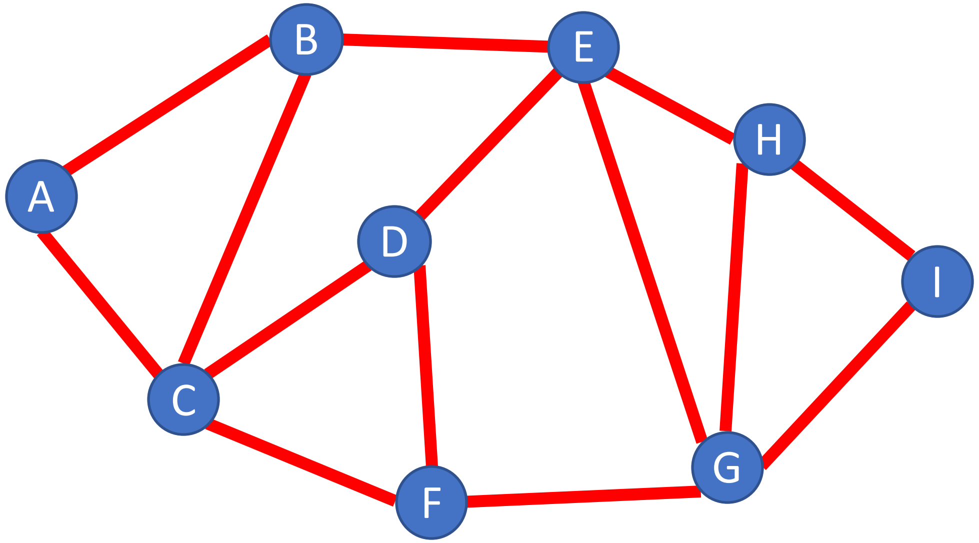


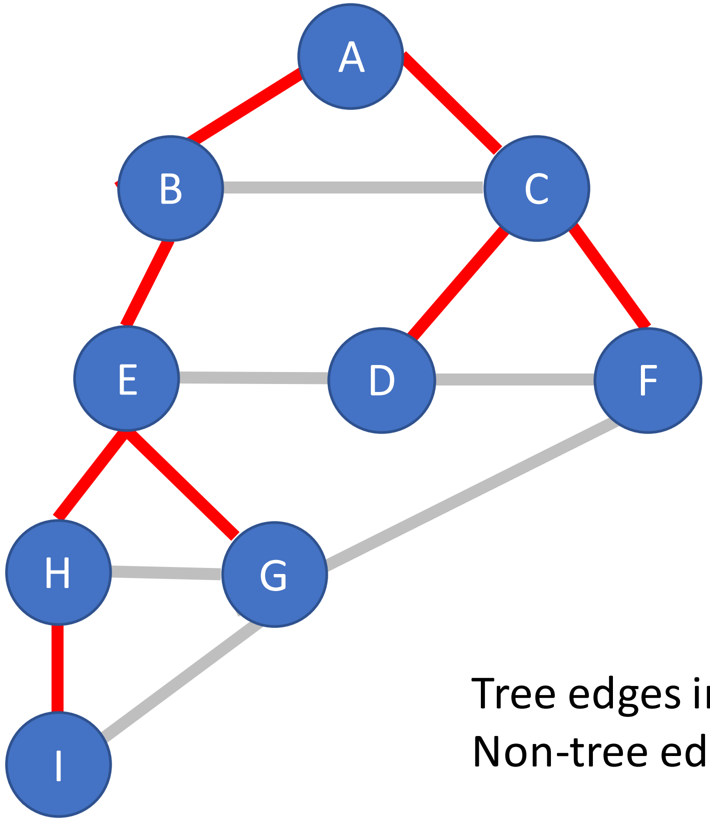


Draw BFS tree starting at A

Q: ~~A~~ ~~C~~ ~~B~~ ~~D~~ ~~F~~ ~~E~~ ~~G~~ ~~H~~ ~~I~~

Tree View of BFS Search Results





Tree edges in red
Non-tree edges in gray

Analysis for Breadth-first search

For a graph having V vertices and E edges

- Each edge is processed once in the while loop for a cost of $\Theta(E)$
- Each vertex is put into the queue once and removed from the queue and processed once, for a cost $\Theta(V)$
  - Also, cost of initializing colors or depth arrays is $\Theta(V)$

Total **time-complexity**: $\Theta(V + E)$

- For graph algorithms this is called “linear”

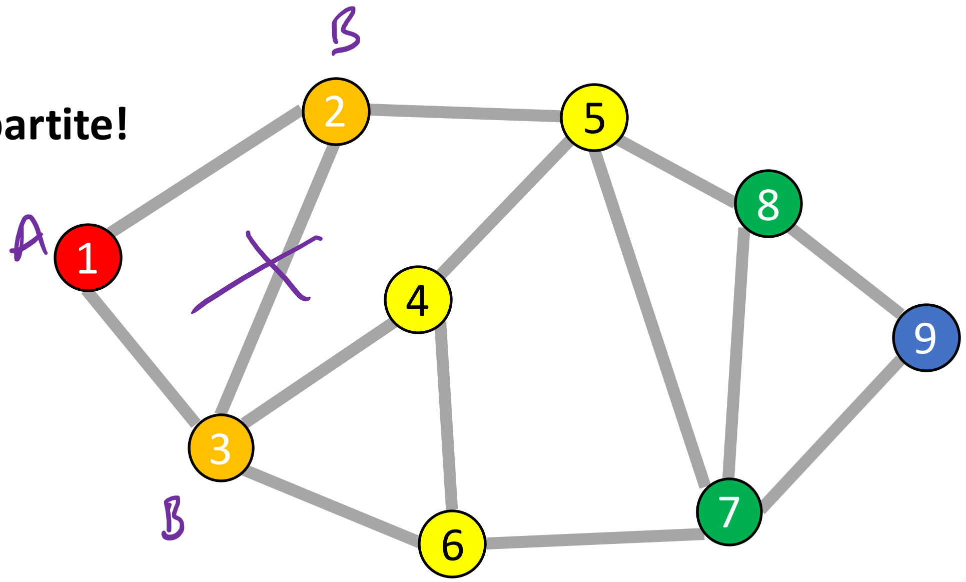
Space complexity: extra space is used for queue and also depth/color arrays, so $\Theta(V)$

Definition: Bipartite

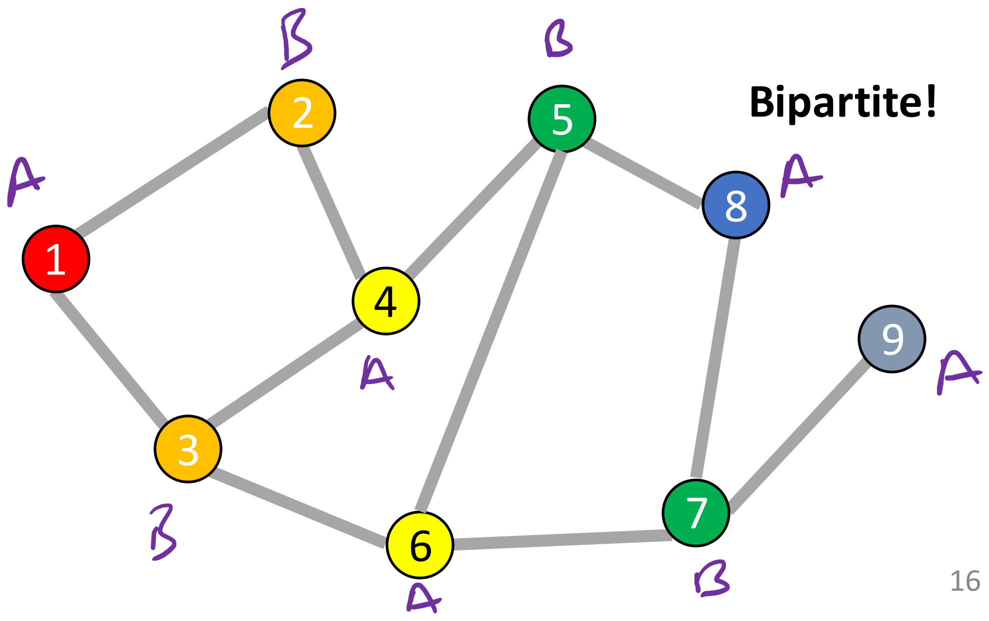
A (undirected) graph is Bipartite provided every vertex can be assigned to one of two teams such that every edge “crosses” teams

- Alternative: Every vertex can be given one of two colors such that no edges connect same-color nodes

Not Bipartite!



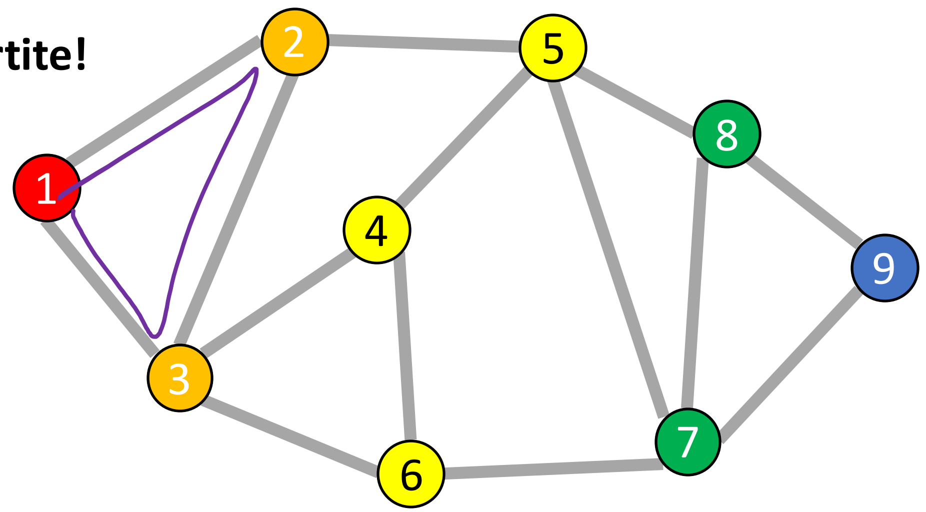
Bipartite!



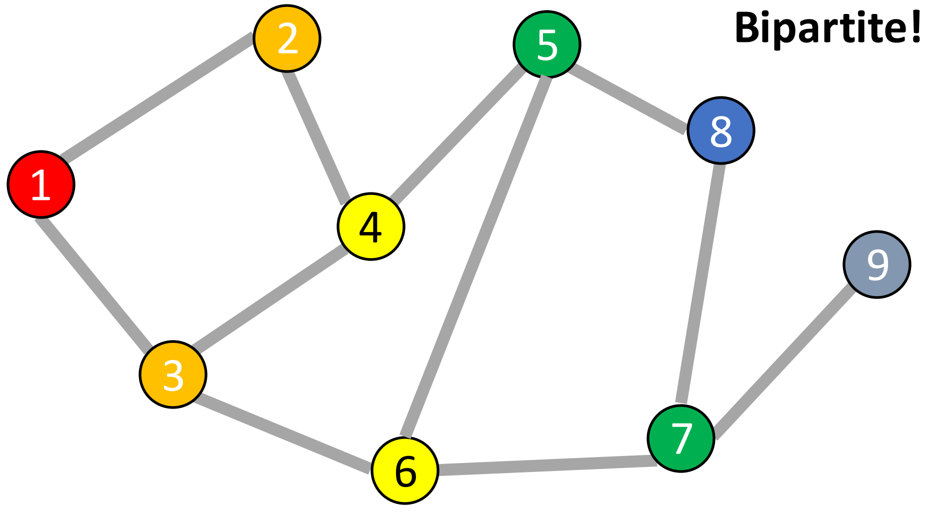
Odd Length Cycles

A graph is bipartite if and only if it has no odd length cycles

Not Bipartite!



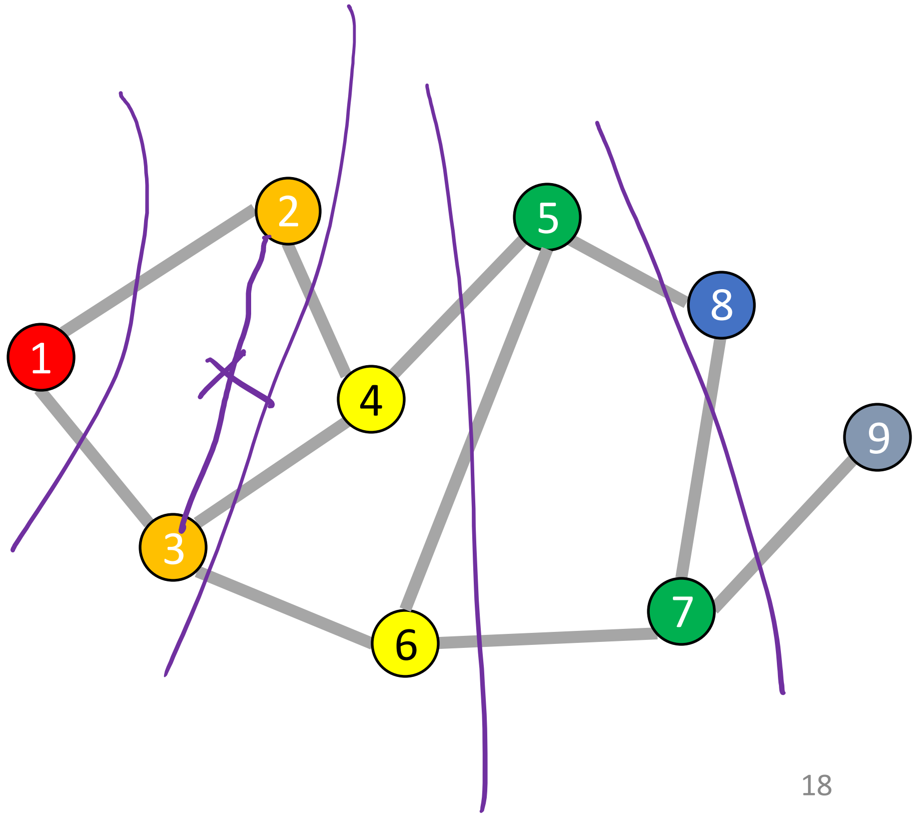
Bipartite!



BFS: Bipartite Graph?

```
def bfs(graph, s):
        toVisit.enqueue(s)
        depth[s] = 0
        depth = [-1,-1,-1,…]  # Length matches |V|
        While toVisit is not empty:
                current = toVisit.dequeue()
                layer = depth [current]
                for v in neighbors(current):
                        if v does not have a depth:
                                depth[v]=layer+1
                                toVisit.enqueue(v)
```

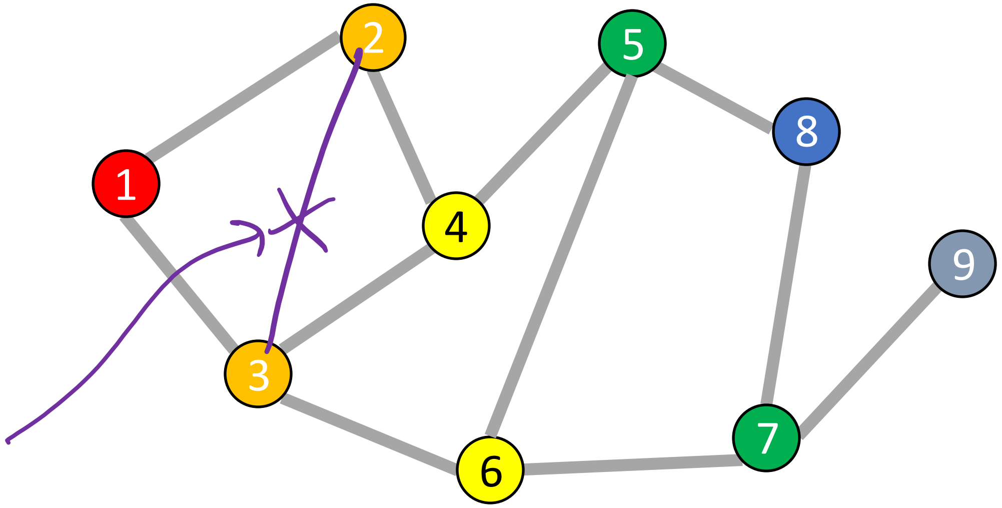
Idea: Check for edges in the same layer!



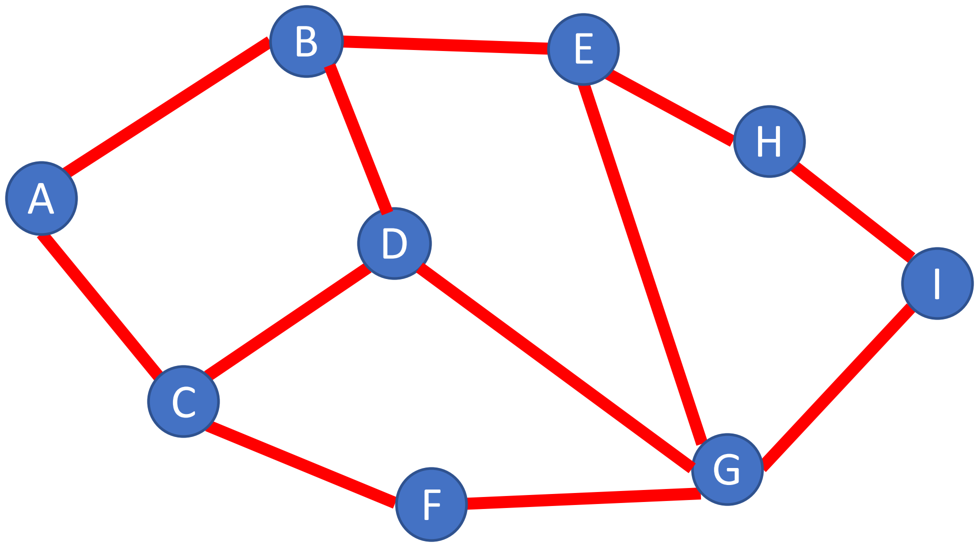
BFS: Bipartite Graph?

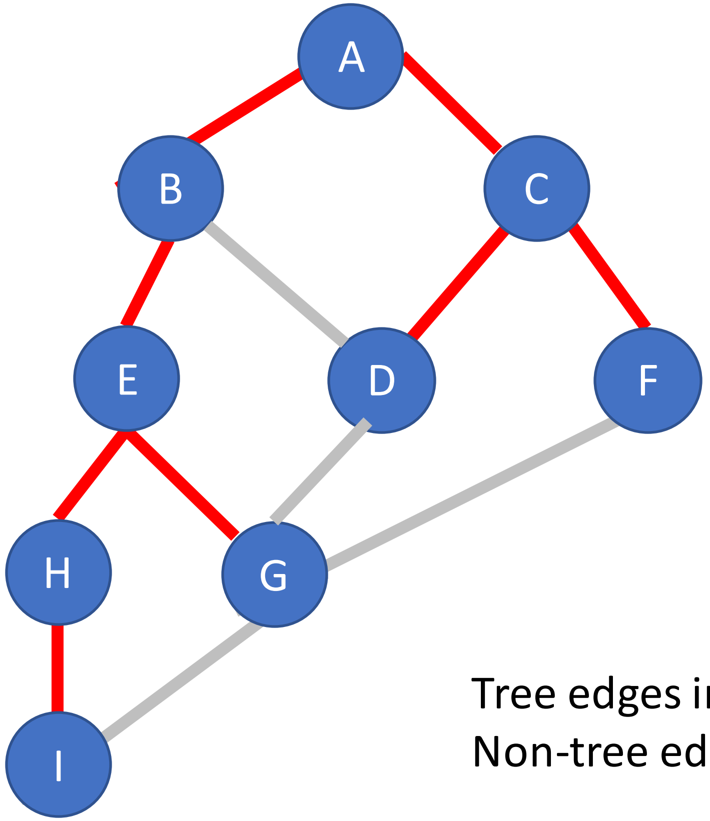
```
def bfs(graph, s):
        toVisit.enqueue(s)
        depth[s] = 0
        depth = [-1,-1,-1,…]  # Length matches |V|
        While toVisit is not empty:
                current = toVisit.dequeue()
                layer = depth [current]
                for v in neighbors(current):
                        if v does not have a depth:
                                depth[v]=layer+1
                                toVisit.enqueue(v)
                        elif depth[v] == depth[current]:
                                return False
        return True
```

Idea: Check for edges in the same layer!



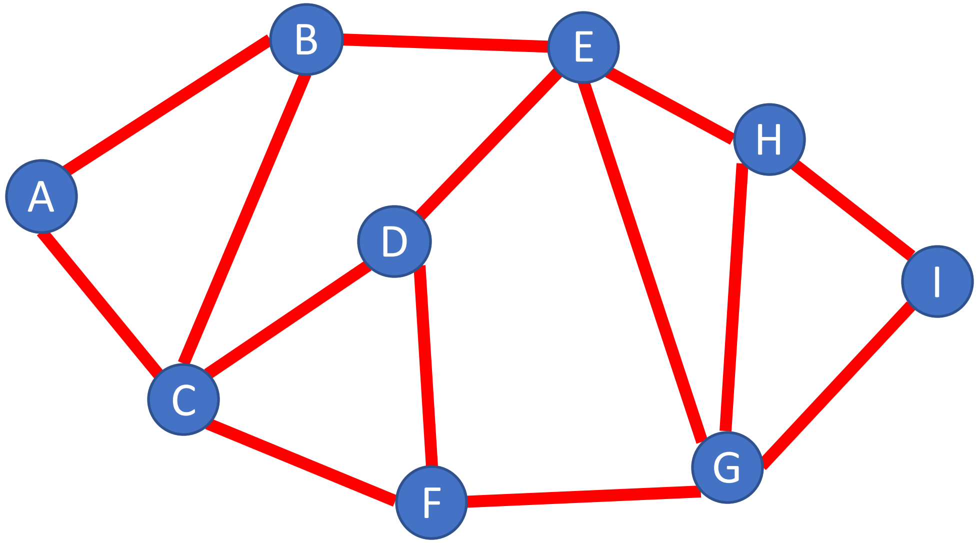
BFS Tree for a Bipartite Graph

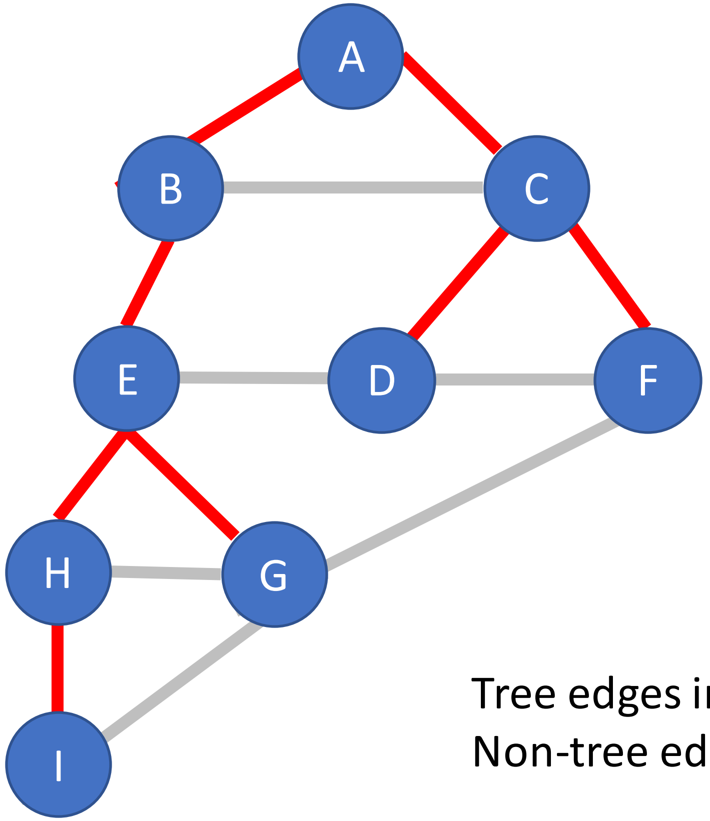




Tree edges in red
Non-tree edges in gray

BFS Tree for a Non-Bipartite Graph





Tree edges in red
Non-tree edges in gray

Depth-First Search

DFS: the Strategy in Words

Depth-first search strategy

- Go as deep as can visiting un-visited nodes
  - Choose any un-visited vertex when you have a choice
- When stuck at a dead-end, backtrack as little as possible
  - Back up to where you could go to another unvisited vertex
- Then continue to go on from that point
- Eventually you'll return to where you started
  - Reach all vertices? Maybe, maybe not

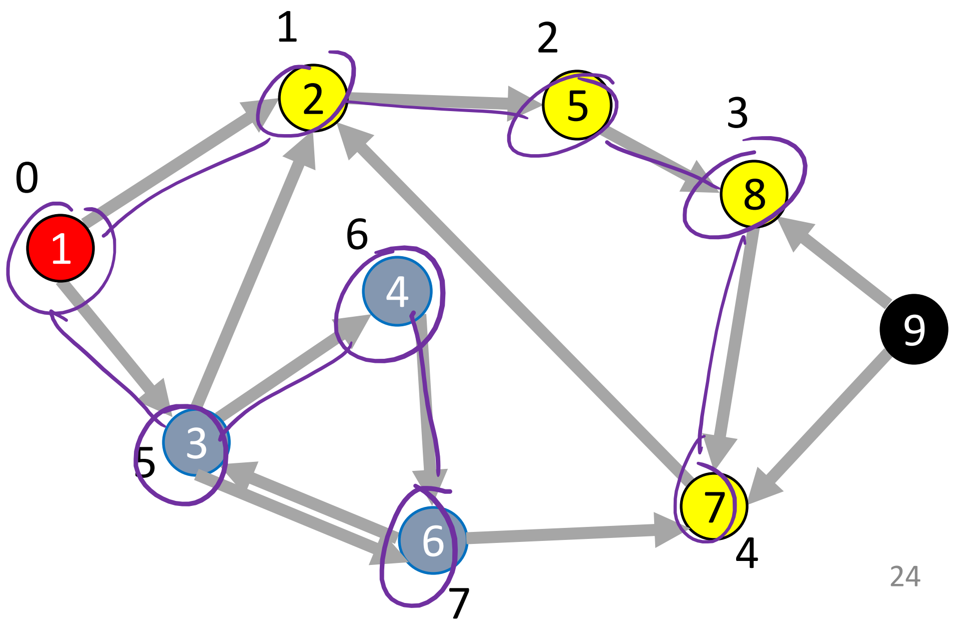
Depth-First Search

Input: a node s , Graph G

Behavior: Start with node s, visit one neighbor of s, then all nodes reachable from that neighbor of s, then another neighbor of s,…

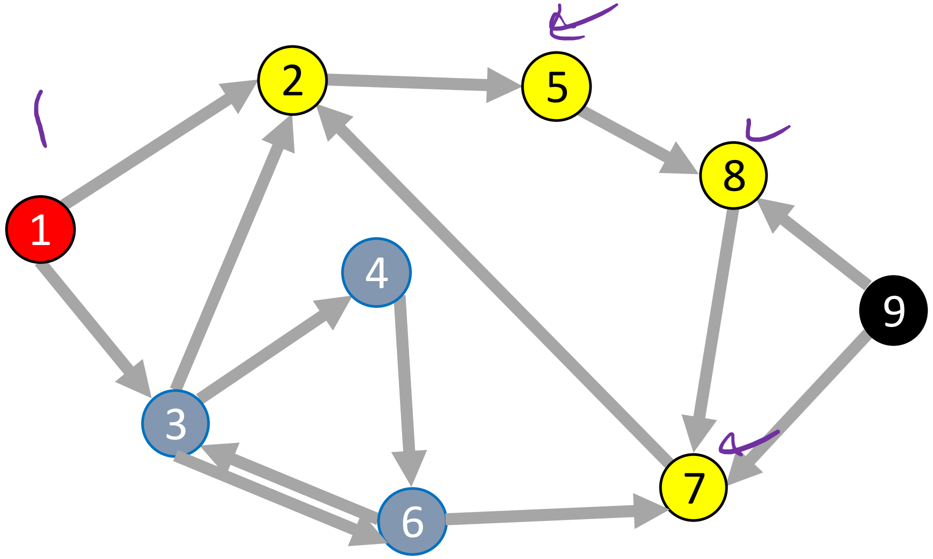
Output:

- Does the graph have a cycle?
- A topological sort of the graph.



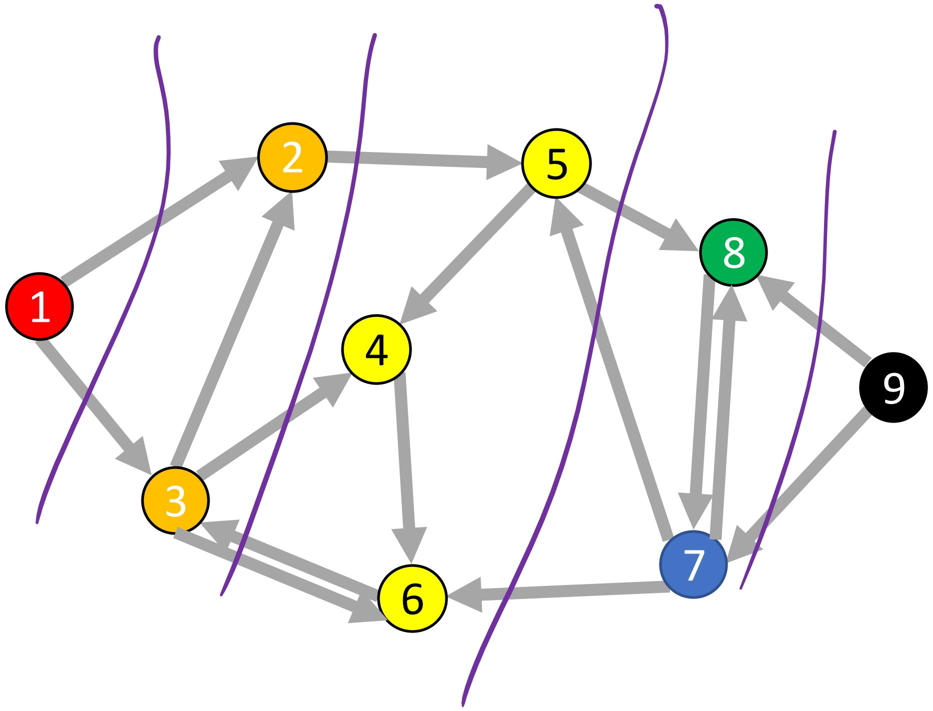
DFS: Non-recursively (less common)

```
def dfs(graph, s):
        toVisit.push(s)
        mark s as "seen"
        While toVisit is not empty:
                current = toVisit.pop()
                for v in neighbors(current):
                        if v not seen:
                                mark v as seen
                                toVisit.push(v)
```



Remember: BFS

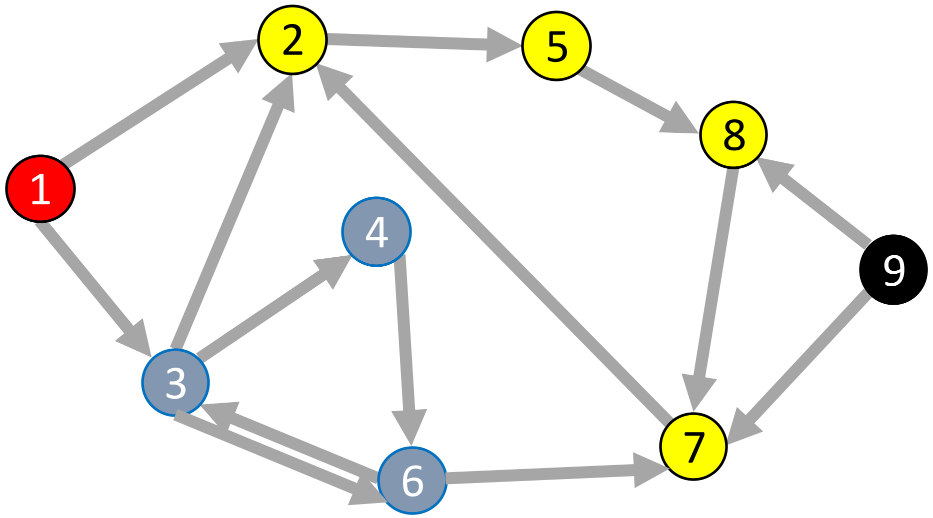
```
def bfs(graph, s):
        toVisit.enqueue(s)
        mark s as "seen"
        While toVisit is not empty:
                current = toVisit.dequeue()
                for v in neighbors(current):
                        if v not seen:
                                mark v as seen
                                toVisit.enqueue(v)
```



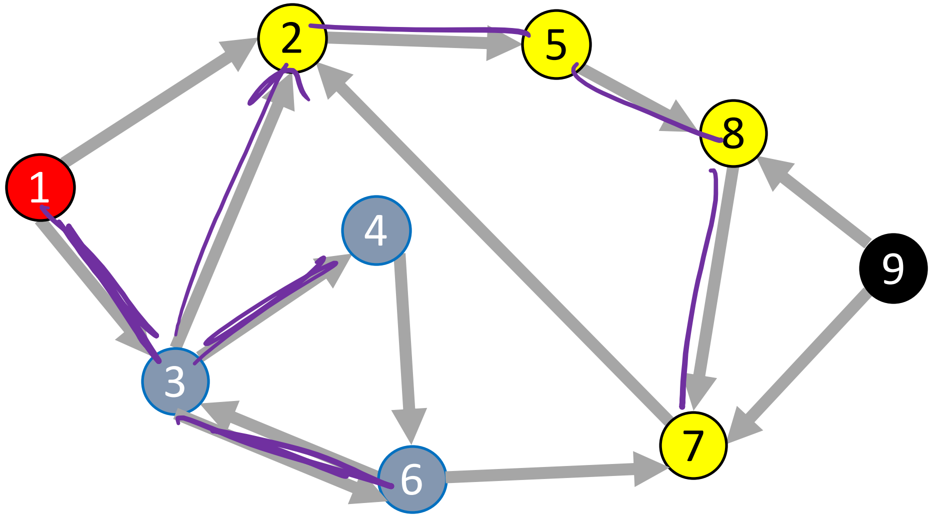
DFS: Recursively

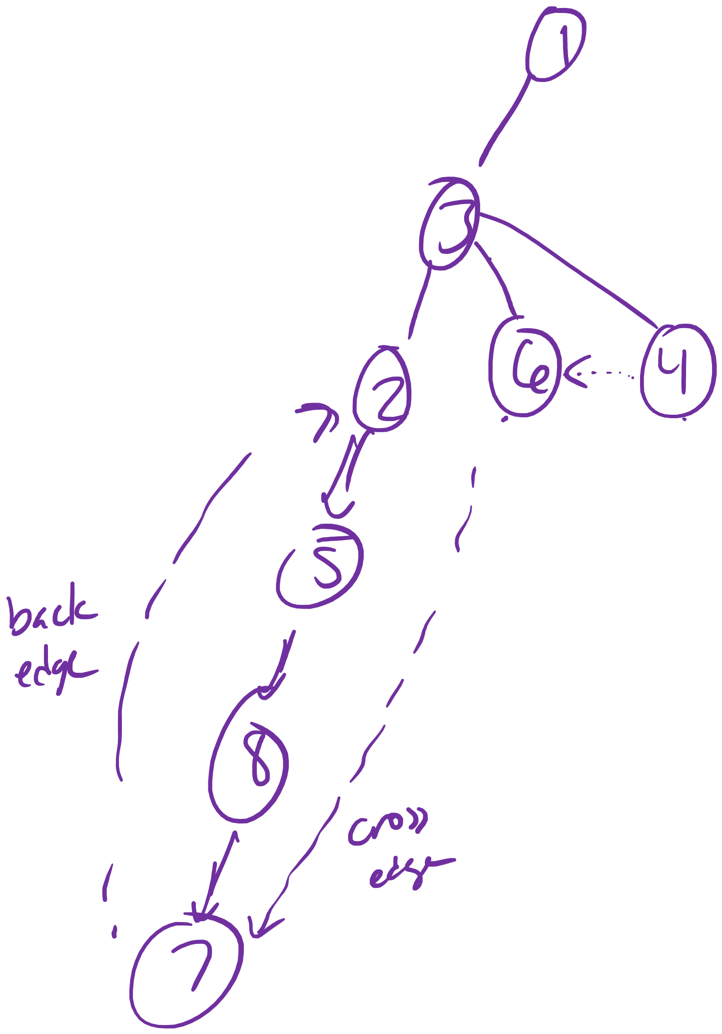
```
def dfs(graph, s):
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        dfs_rec(graph, s, seen, done)

def dfs_rec(graph, curr, seen, done)
        mark curr as seen
        for v in neighbors(current):
                if v not seen:
                        dfs_rec(graph, v, seen, done)
        mark curr as done
```



View of DFS Results as a Tree

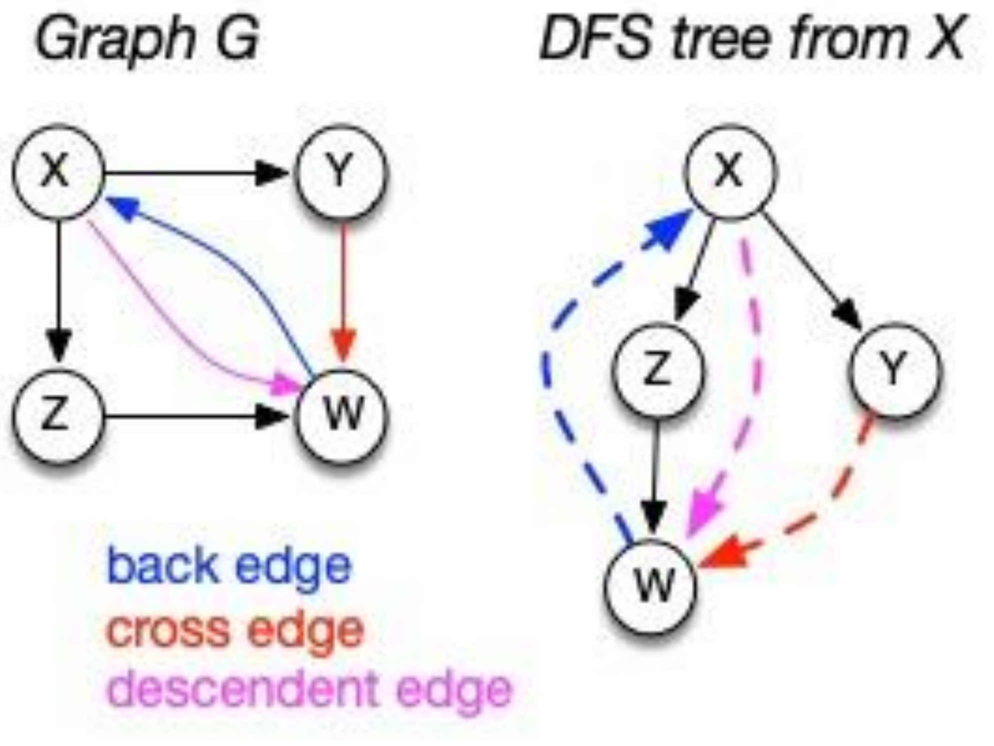


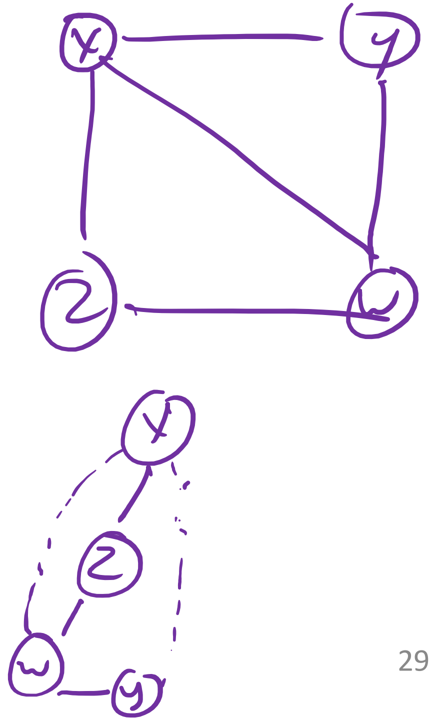


Depth-first search tree

As DFS traverses a digraph, edges classified as:

- tree edge, back edge, descendant edge, or cross edge
- If graph undirected, do we have all 4 types?



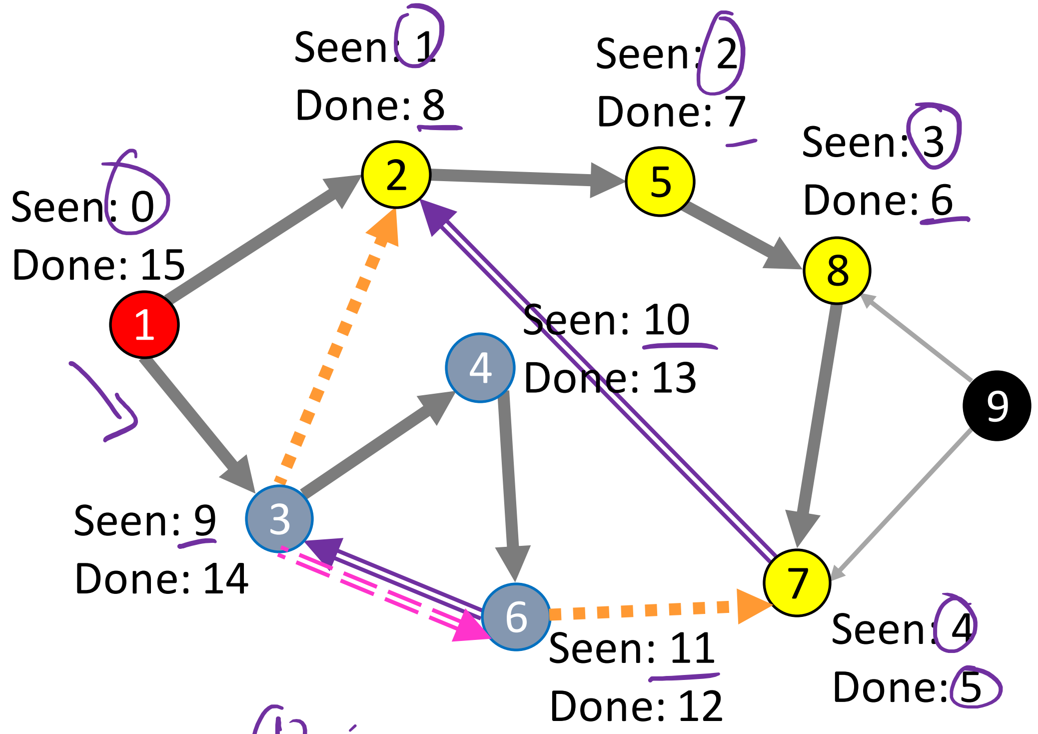


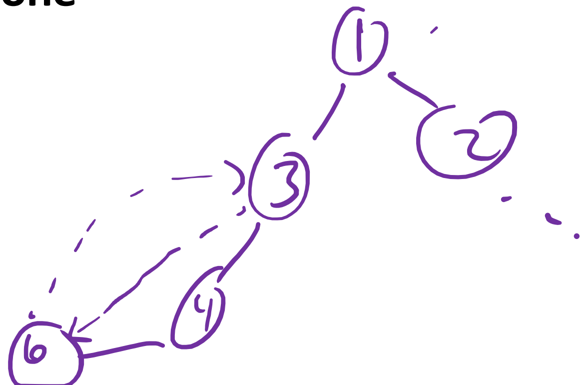
Using DFS

Consider the "seen times" and "done times"

Edges can be categorized:

- Tree Edge
  - (a, b) was followed when pushing
  - (a, b) when b was **unseen** when we were at a
- Back Edge
  - (a, b) goes to an "ancestor"
  - a and b **seen** but not **done** when we saw (a, b)
  - $t_{seen}(b) < t_{seen}(a) < t_{done}(a) < t_{done}(b)$
- Forward Edge
  - (a, b) goes to a "descendent"
  - b was **seen** and **done** between when a was **seen** and **done**
  - $t_{seen}(a) < t_{seen}(b) < t_{done}(b) < t_{done}(a)$
- Cross Edge
  - (a, b) connects "branches" of the tree
  - b was **seen** and **done** before a was ever **seen**
  - (a, b) when $t_{done}(b) > t_{seen}(a)$ and



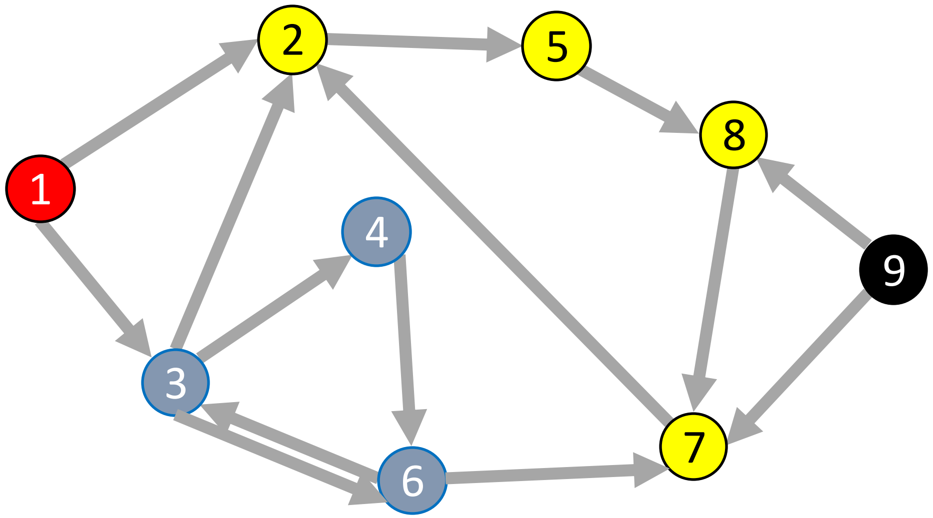


DFS: Cycle Detection

Idea: Look for a back edge!

```
def dfs(graph, s):
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        dfs_rec(graph, s, seen, done)

def dfs_rec(graph, curr, seen, done)
        mark curr as seen
        for v in neighbors(current):
                if v not seen:
                        dfs_rec(graph, v, seen, done)
        mark curr as done
```



DFS: Cycle Detection

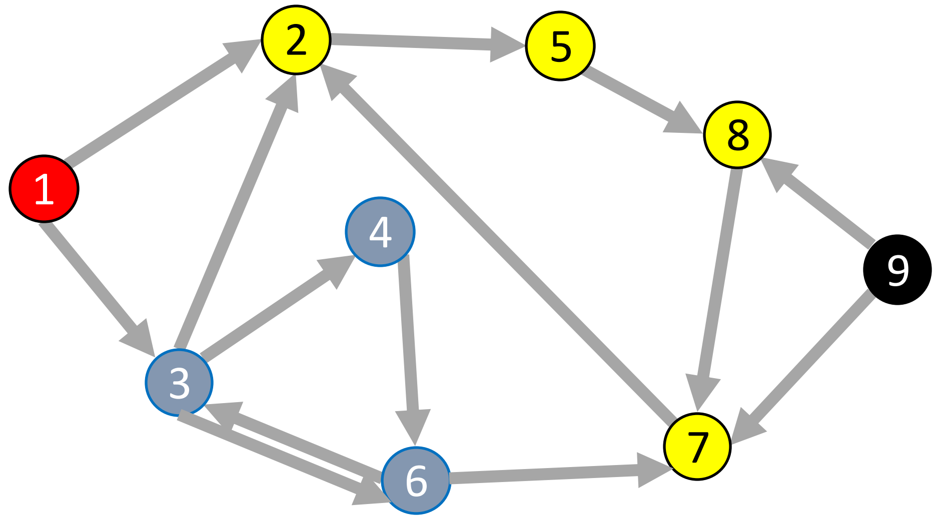
Idea: Look for a back edge!

```
def hasCycle(graph, s):
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        dfs_rec(graph, s, seen, done)

def hasCycle_rec(graph, curr, seen, done)

        mark curr as seen
        for v in neighbors(current):


                if v not seen:
                        dfs_rec(graph, v, seen, done)
        mark curr as done
```

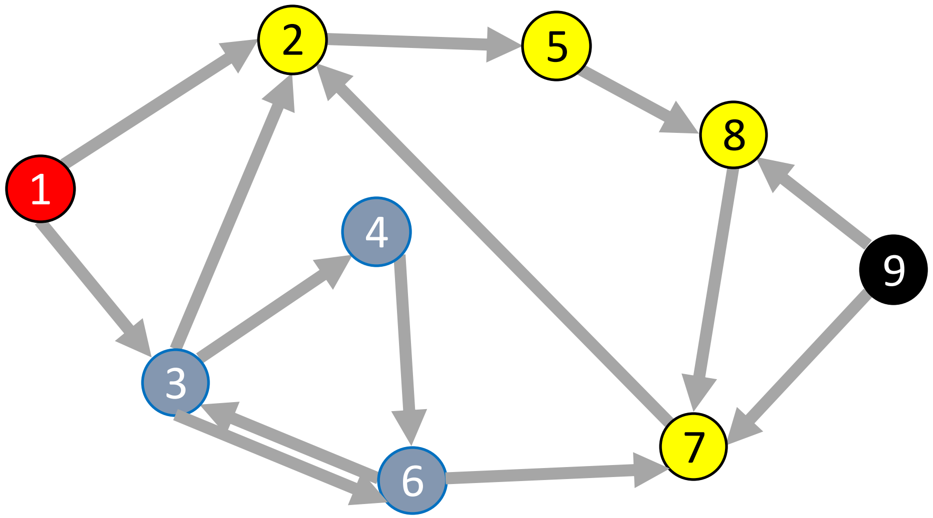


DFS: Cycle Detection

Idea: Look for a back edge!

```
def hasCycle(graph, s):
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        return hasCycle_rec(graph, s, seen, done)

def hasCycle _rec(graph, curr, seen, done):
        cycle = False
        mark curr as seen
        for v in neighbors(current):
                if v seen and v not done:
                        cycle = True
                elif v not seen:
                        cycle = dfs_rec(graph, v, seen, done) or cycle
        mark curr as done
        return cycle
```



Back Edges in Undirected Graphs

Finding back edges for an undirected graph is not **quite** this simple:

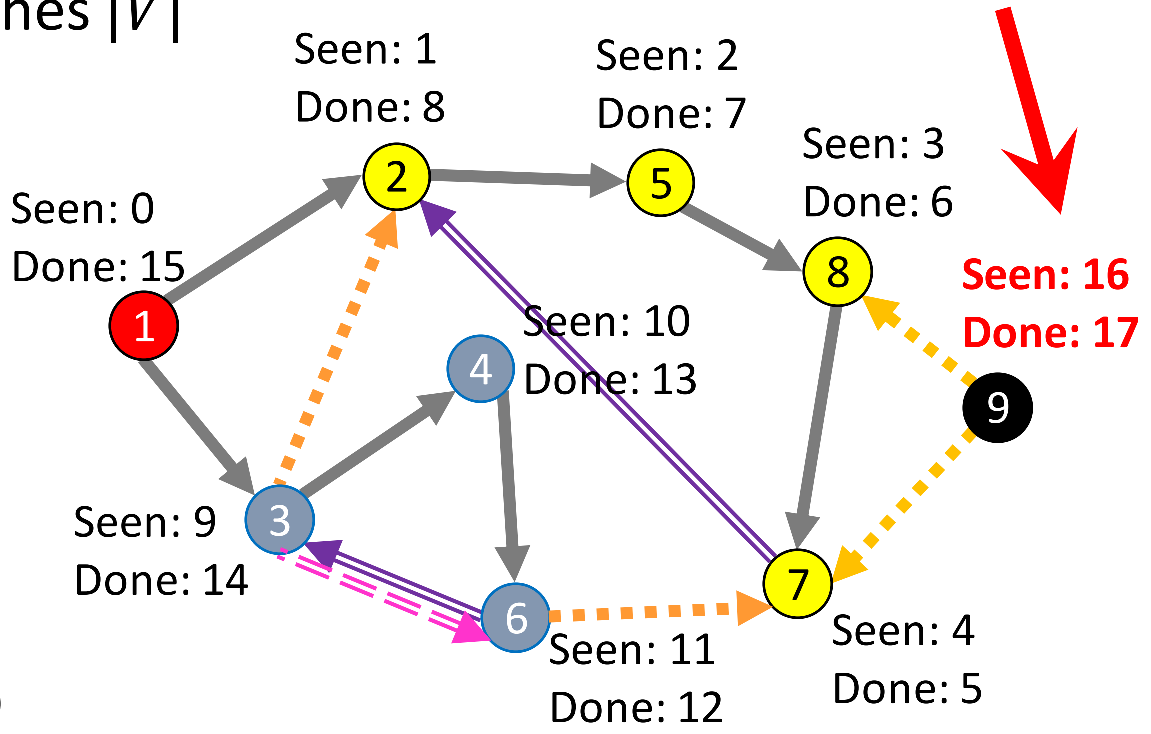
- The parent node of the current node is **seen** but not **done**
- Not a cycle, is it? It's the same edge you just traversed

Question: how would you modify our code to recognize this?

DFS "Sweep" to Process All Nodes

```
def dfs_sweep(graph):  # no start node given
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        for s in graph :  # do DFS at every vertex
            if s not seen:
                dfs_rec(graph, s, seen, done)

def dfs_rec(graph, curr, seen, done) # unchanged
        mark curr as seen
        for v in neighbors(current):
                if v not seen:
                        dfs_rec(graph, v, seen, done)
        mark curr as done
```



Time Complexity of DFS

For a digraph having V vertices and E edges

- Each edge is processed once in the while loop of dfs_rec() for a cost of $\Theta(E)$
  - Think about *adjacency list* data structure.
  - Traverse each list exactly once. (Never back up)
  - There are a total of **E** nodes in all the lists
- The non-recursive dfs() algorithm will do $\Theta(V)$ work even if there are no edges in the graph
- Thus over all time-complexity is $\Theta(V + E)$
  - Remember: this means the larger of the two values
  - Reminder: This is considered “linear” for graphs since there are two size parameters for graphs.
- Extra space is used for seen/done (or color) array.
  - Space complexity is $\Theta(V)$