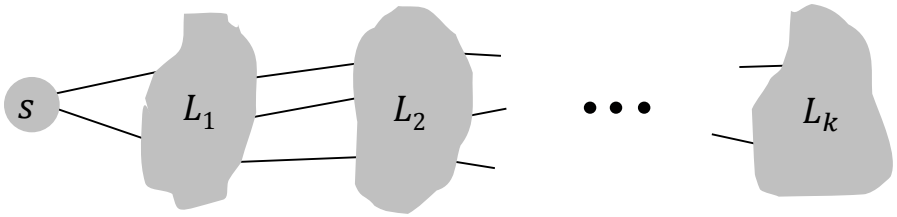
# Lecture 15: Basic Graph Algorithms

# Breadth First Search

BFS idea. Explore outward from $s$ in all possible directions, adding nodes one "layer" at a time.




BFS.

- $L_0 = \{s\}$.
- $L_1$ = all neighbors of $L_0$.
- $L_2$ = all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$.
- $L_{i+1}$ = all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$.

Def: The distance from $u$ to $v$ is the number of edges on the shortest path from $u$ to $v$.

Theorem. For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$. There is a path from $s$ to $t$ iff $t$ appears in some layer.

# BFS Algorithm

```
BFS(G, s):
for each vertex u ∈ V − {s}
    u.color ← white
    u.d ← ∞
    u.p ← nil
s.color ← gray
s.d ← 0
initialize an empty queue Q
Enqueue(Q, s)
while Q ≠ ∅ do
    u ← Dequeue(Q)
    for each v ∈ Adj[u]
        if v.color = white then
            v.color ← gray
            v.d ← u.d + 1
            v.p ← u
            Enqueue(Q, v)
    u.color ← black
```

## Colors:

- white: undiscovered
- gray: discovered, but neighbors not fully explored (these nodes are in $Q$)
- black: discovered and neighbors fully explored

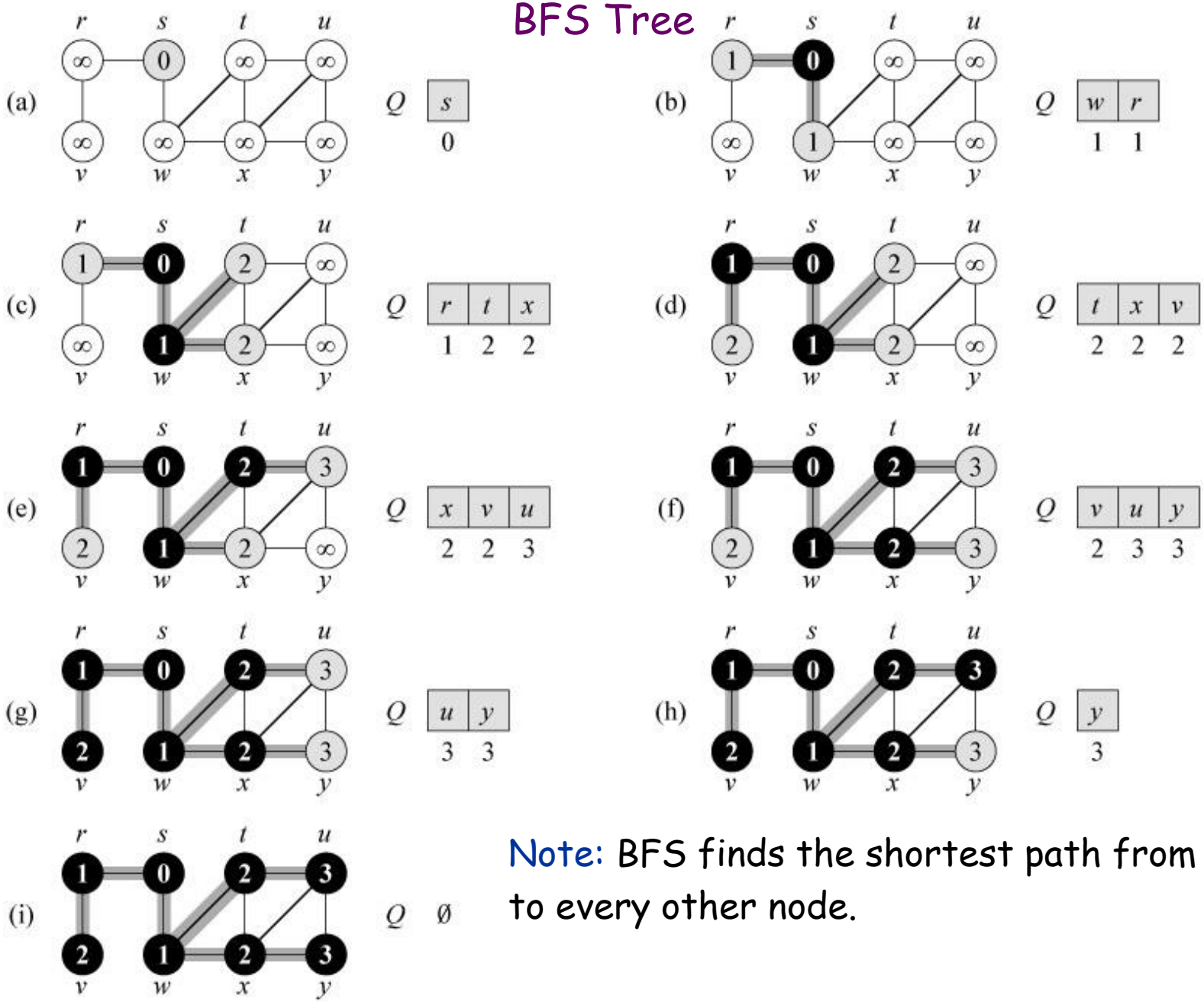
## Parent pointers:

- Pointing to the node that leads to its discovery
- Parent must be in $L_{i-1}$
- Can follow parent pointers to find the actual shortest path
- The pointers form a tree, rooted at $s$

## Running time:

$\sum_u (1 + \deg(u)) = \Theta(V + E)$, which is $\Theta(E)$ if the graph is connected.

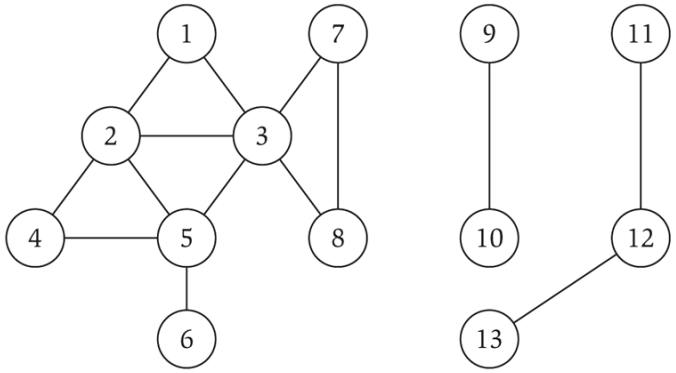
# BFS Tree




Note: BFS finds the shortest path from $s$ to every other node.

# Connected Component

Connected component containing $s$. All nodes reachable from $s$.




Connected component containing node $1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

BFS starting from $s$ finds the connected component containing $s$.

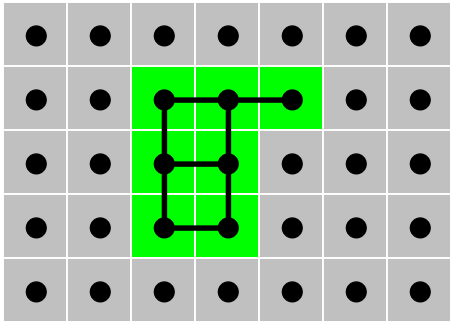
Repeatedly running BFS from an undiscovered node finds all the connected components.

# Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue

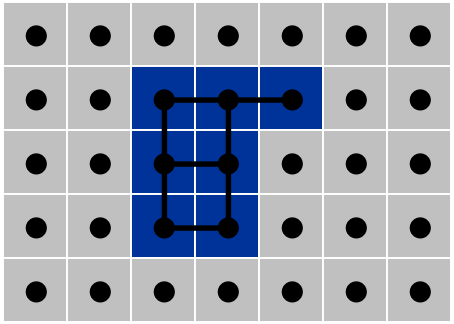



# Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue

# s-t connectivity and shortest path in directed graphs

s-t connectivity (often called reachability for directed graphs). Given two nodes $s$ and $t$, is there a path from $s$ to $t$?

- Undirected graph: $s$ can reach $t \Leftrightarrow t$ can reach $s$
- Directed graph: Not necessarily true

s-t shortest path problem. Given two node $s$ and $t$, what is the length of the shortest path between s and t?

- Undirected graph: $p$ is the shortest path from $s$ to $t \Leftrightarrow p$ is the shortest path from $t$ to $s$
- Directed graph: Not necessarily true

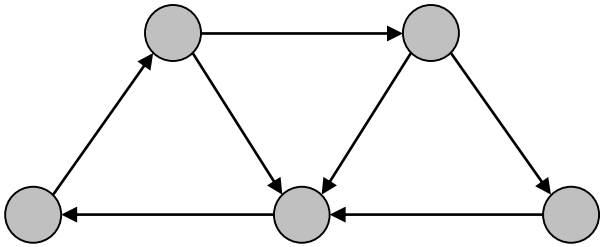
BFS on a directed graph. Same as in undirected case

- Ex: Web crawler. Start from web page $s$. Find all web pages linked from s, either directly or indirectly.

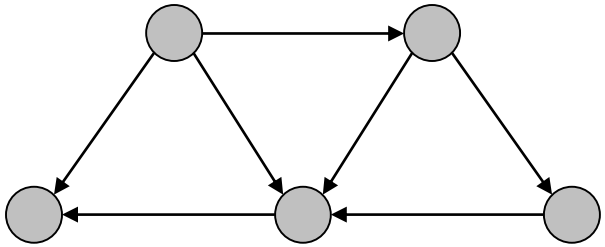
# Strong Connectivity in Directed Graphs

Def. Node $u$ and $v$ are mutually reachable if there is a path from $u$ to $v$ and also a path from $v$ to $u$.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.



strongly connected

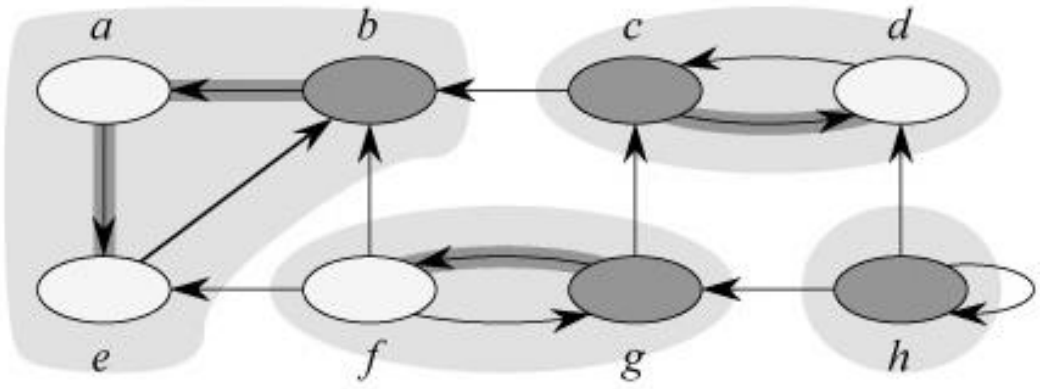


not strongly connected

Algorithm for checking strong connectivity

- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Reverse all edges in $G$, and run BFS from $s$.
- Return true iff all nodes reached in both BFS executions.

# Strongly Connected Components

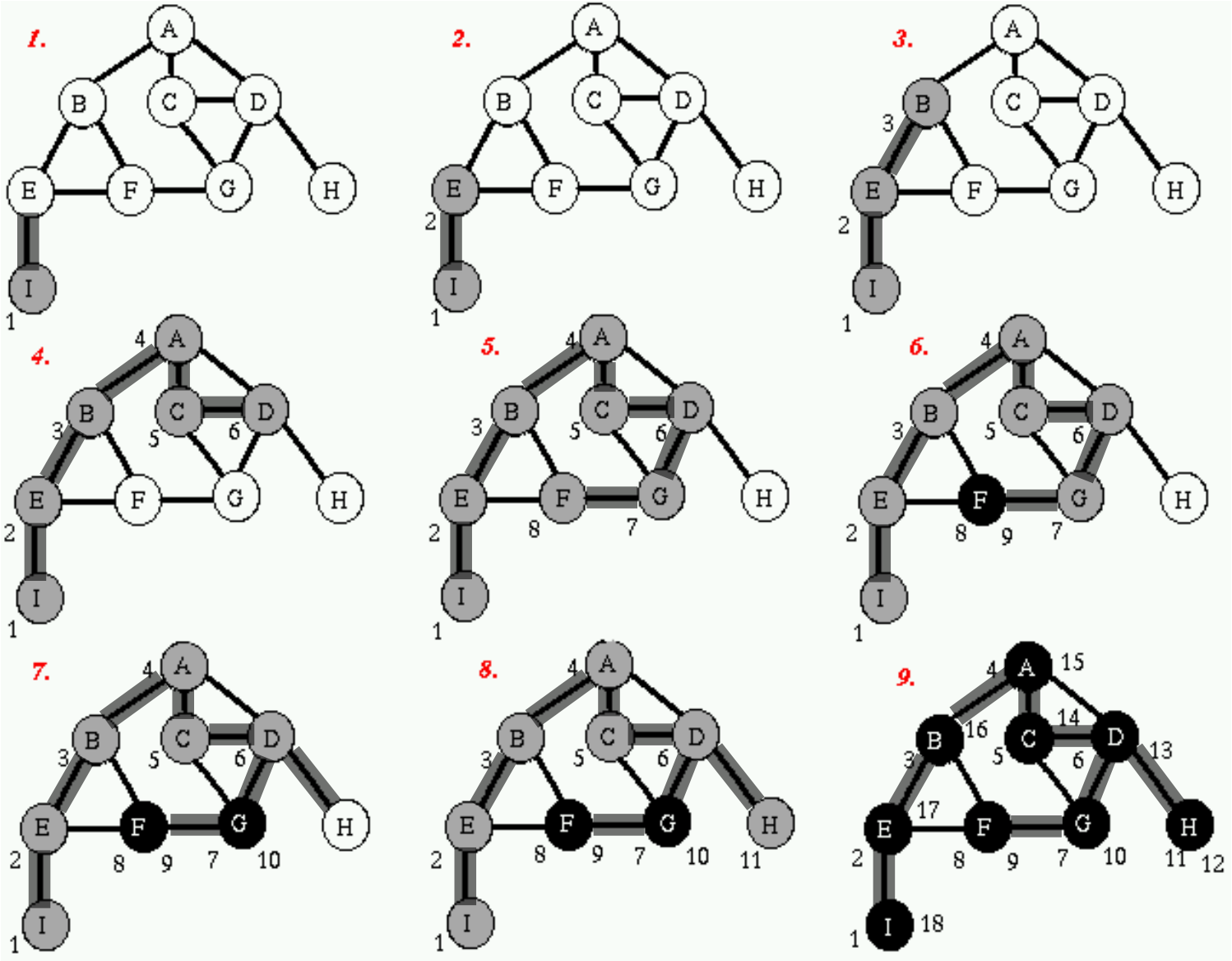



```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V+E)$ algorithm (not required)

# Depth First Search and DFS Tree



# DFS Algorithm

```
DFS(G):
for each vertex u ∈ V do
    u.color ← white
    u.p ← nil
for each vertex u ∈ V do
    if u.color = white then
        DFS-Visit(u)

DFS-Visit(u):
u.color ← gray
for each v ∈ Adj[u] do
    if v.color = white then
        v.p ← u
        DFS-Visit(v)
u.color ← black
```

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

Colors:

- white: undiscovered
- gray: discovered, but neighbors not fully explored (on recursion stack)
- black: discovered and neighbors fully explored

Parent pointers:

- Pointing to the node that leads to its discovery
- The pointers form a tree, rooted at $s$

# Application: Cycle Detection

Problem: Given an undirected graph $G = (V, E)$, check if it contains a cycle.

Idea:

- A tree (connected and acyclic) has exactly $V - 1$ edges.
- If it has less edges, it cannot be connected.
- If it has more edges, it must contain a cycle.

Algorithm:

- Run BFS/DFS to find all the connected components of $G$.
- For each connected component, count the number of edges.
- If # edges $\geq$ # vertices, return "cycle detected".

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

Q: What if we also want to find a cycle (any is OK) if it exists?

# Tree edges, back edges, and cross edges

After we have run BFS or DFS on an undirected graph, the edges can be classified into 3 types:

- Tree edges: traversed by the BFS/DFS.
- Back edges: connecting a node with one of its ancestors in the BFS/DFS-tree except its parent.
- Cross edges: connecting two nodes with no ancestor/descendent relationship.

Theorem: In a DFS on an undirected graph, there are no cross edges.

Pf: Consider any edge $(u, v)$ in $G$.

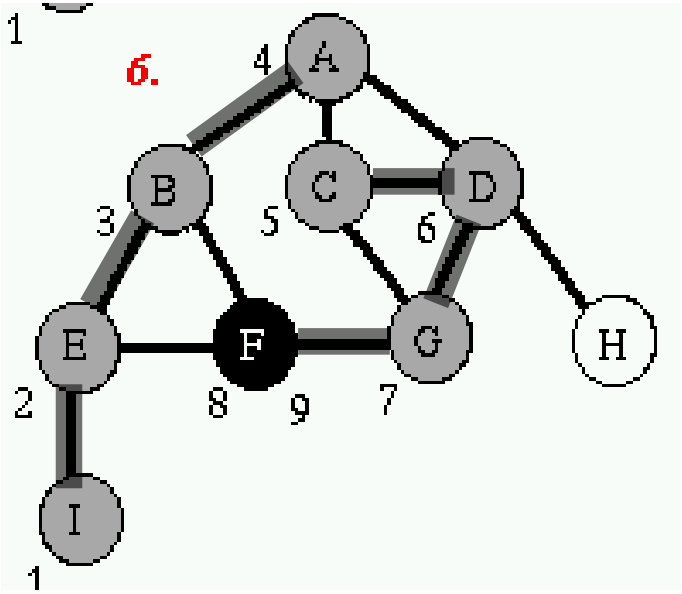
- Without loss of generality, assume $u$ is discovered before $v$.
- Then $v$ is discovered while $u$ is gray (why?).
- Hence $v$ is in the DFS subtree rooted at u.
  - If $v.p = u$, then $(u, v)$ is a tree edge.
  - If $v.p \neq u$, then $(u, v)$ is a back edge.

Theorem: In a BFS on an undirected graph, there are no back edges.

# DFS for cycle detection

Idea: Run DFS on each connected component of $G$.

- If there is a back edge $(u, v)$. Then, $v$ is an ancestor (but not parent) of $u$ in the DFS trees. There is thus a path from $v$ to $u$ in the DFS-tree , and the back edge $(u, v)$ completes a cycle.
- If there is no back edge, then there are only tree edges, so the graph is a forest, and hence is acyclic.



# DFS for cycle detection

```
CycleDetection(G):
for each vertex u ∈ V do
    u.color ← white
    u.p ← nil
for each vertex u ∈ V do
    if u.color = white then DFS-Visit(u)
return "No cycle"

DFS-Visit(u):
u.color ← gray
for each v ∈ Adj[u] do
    if v.color = white then
        v.p ← u
        DFS-Visit(v)
    else if v ≠ u.p then
        output "Cycle found:"
        while u ≠ v do
            output u
            u ← u.p
        output v
        return
u.color ← black
```

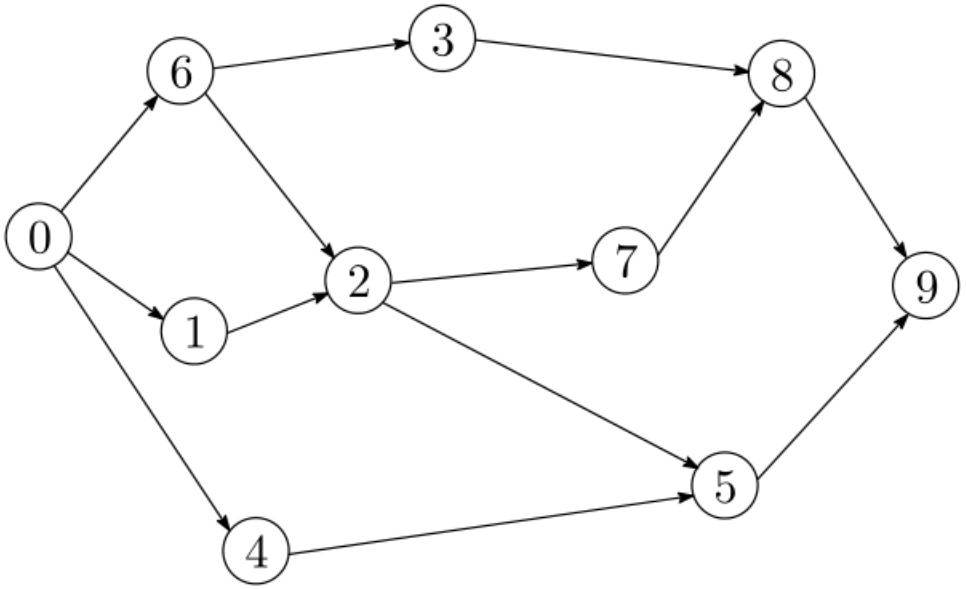
Running time: $\Theta(V)$

- Only traverse DFS-tree edges, until the first non-tree edge is found
- At most $V - 1$ tree edges

# Directed Acyclic Graphs and Topological Ordering

Def. A DAG is a directed graph that contains no (directed) cycles.

Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes such that for every edge $u \to v$, $u$ is ordered before $v$.




Examples of topological orderings:

- 0, 6, 1, 4, 3, 2, 5, 7, 8, 9
- 0, 4, 1, 6, 2, 5, 3, 7, 8, 9
- …

# Topological Sort Algorithm

## Observations

- Starting vertex must have zero in-degree
- If such a vertex doesn't exist, the graph would not be acyclic

## Algorithm

- A vertex with zero in-degree can be output right away.
- If a vertex $u$ is output, then all the edges $(u, v)$ are no longer useful, since $v$ does not need to wait for $u$ anymore
  - Can remove all the edges $(u, v)$
- With vertex $u$ removed, the new graph is still a DAG
  - Repeat step until no vertex is left

```
TopologicalSort(G):
compute deg^in(u) for all u
let Q be an empty queue
for each u ∈ V do
    if deg^in(u) = 0 then
        Enqueue(Q, u)
while Q ≠ ∅ do
    u ← Dequeue(Q)
    output u
    for each v ∈ Adj(u) do
        deg^in(v) ← deg^in(v) − 1
        if deg^in(v) = 0 then
            Enqueue(Q, v)
```

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