Lecture 19: Basic Graph Algorithms

Depth First Search and DFS Tree

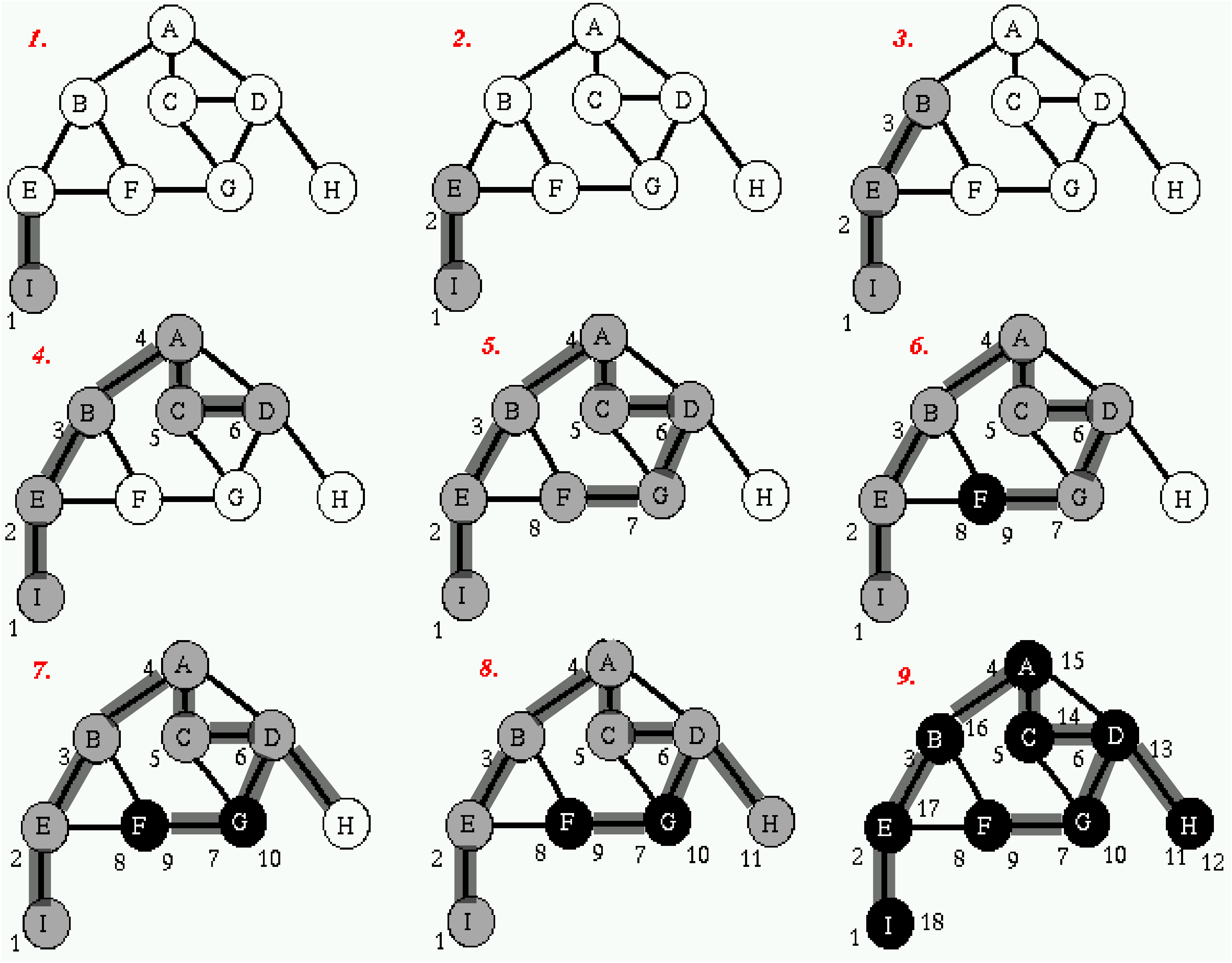
- Breadth first search is "Broad".
  - It builds a wide tree, connecting a node to ALL of the neighbors that have not yet been processed.
  - Once a node starts being processed, it sees ALL of its neighbors before any other node is processed

- There is another procedure, called DEPTH first search.
  - Instead of going broad, it goes DEEP
  - It recursively searches deep into the tree

  - When a node *u* is processed, it looks at each of its neighbors in order
    - At the time u checks a neighbor *v*, DFS starts processing *v* (which starts processing it's children, which start processing their children, etc.).

    - Only after all of *v*'s descendants have been processed does *u* go on to process its next neighbor

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)
………….
```

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

- DFS(G) calls the DFS-visit search on each vertex u
- Before DFS-Visit(u) returns, all nodes in the connected component containing u are turned black (will see later)
- So DFS-Visit will only be called once for each connected component in G

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

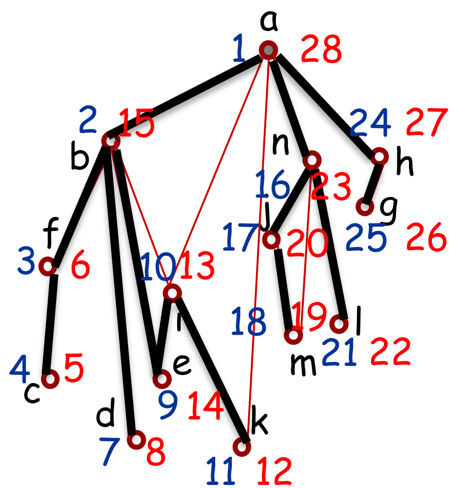
We can add starting and finishing time for each *u*:

Starting time when u.color ← gray

Finishing time when u.color ← black

DFS Worked Example

Adjacency Lists:
a: b, i, k, n, h
b: a, f, d, e, i
c: f
d: b
e: b, i
f: c, b
g: h
h: a, g
i: e, b, k, a
j: n, m
k: i, a
l: n
m: j, n
n: a, j, m, l



- The starting and finishing times are useful for some applications (to be discussed later)
- The bold edges form the DFS tree.
- The rest of the edges (light red) point to ancestors in the tree, and are called back-edges.
- Back edges are also useful for some applications.

Application: Cycle Detection

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

Idea:

- A tree (connected and acyclic) contains **exactly** $V - 1$ edges.
- If it has fewer 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 running BFS or DFS on an undirected graph, all edges can be classified into one of 3 types:

- Tree edges: traversed by the BFS/DFS.
- Back edges: connecting a node with one of its ancestors in the BFS/DFS-tree (other than 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.

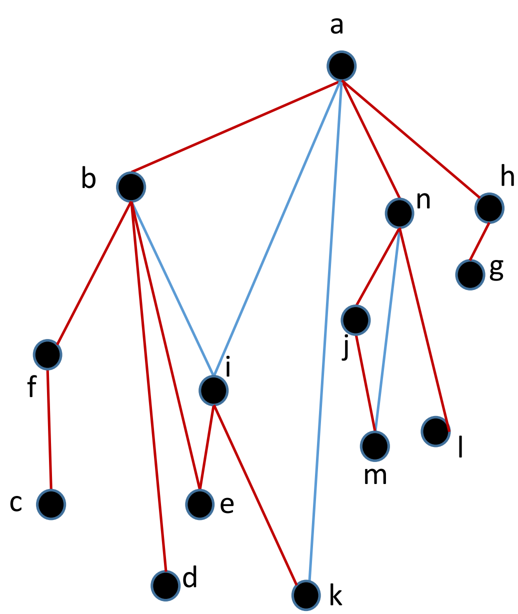
Theorem: In a BFS on an **undirected** graph, there are no back edges.
(Not proven)

DFS for cycle detection

Idea: Run DFS on each connected component of G.

- If (u, v) is a back edge.
  - => 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
  - => v to u plus back edge (u, v) creates a cycle.
- If no back edge exists then it only contains (DFS) tree edges
  - => the graph is a forest, and hence is acyclic.



- In DFS starting at *a*,
  (i,b) was first back edge found
- => *b* was ancestor (not parent) of *i* in tree
- => tree contains path *(b->e->i)* from *b* to *i*
- => this path plus edge *(i ,b)* is the cycle
  b->e->i->b

DFS for cycle detection

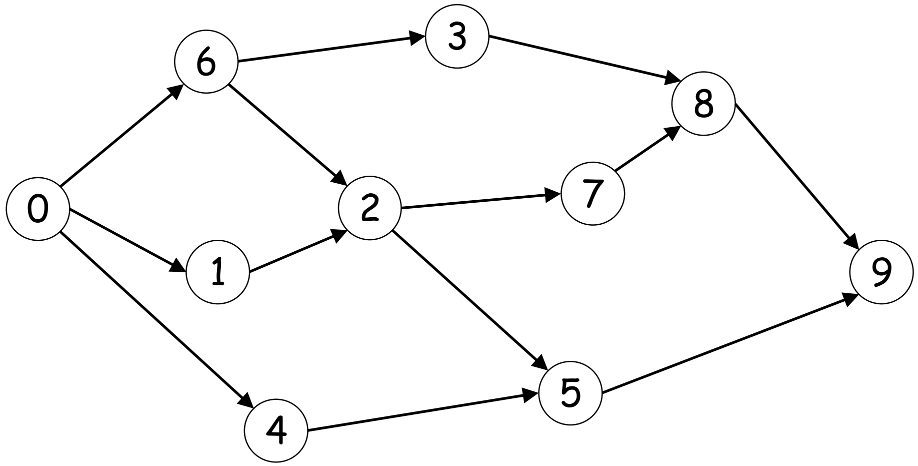
```
CycleDetection(G):
for each vertex u ∈ V do
      u.color ← white
      u.p ← nil
for each vertex u ∈ V do
      if u.color = white then DFS-Visit(u)
return "No cycle"

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 //back edge (u,v)
            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 Graph

A directed graph distinguishes between edge (u, v) and edge (v, u). Directed graphs are often used to represent order-dependent tasks

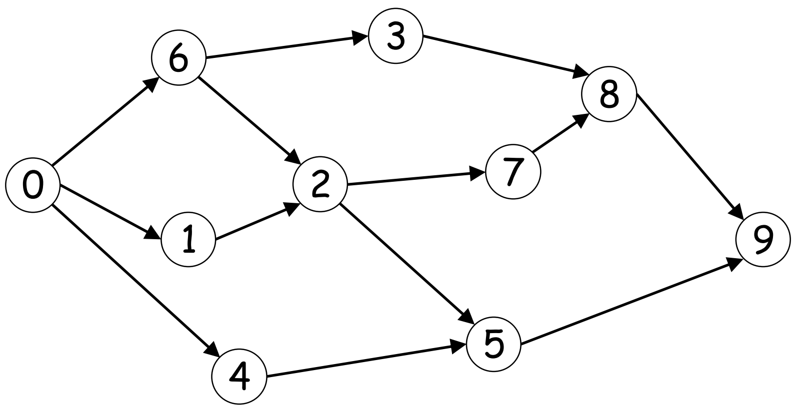


- out-degree of vertex v is the number of edges leaving v
- in-degree of vertex v is the number of edges entering v
- Each edge (u, v) contributes one to the out-degree of u and one to the in-degree of v, so

$$\sum_{v \in V} \text{out-degree}(v) = \sum_{v \in V} \text{in-degree}(v) = |E|$$

Topological Sort

- Directed Acyclic Graph (DAG): Directed graph with no cycles.
- A Topological ordering of a graph is a linear ordering of the vertices of a DAG such that if (u, v) is in the graph, u appears before v in the linear ordering



- Topological ordering may not be unique
- The graph above has many 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
  - A DAG must contain at least one vertex with in-degree zero
- Algorithm: Topological Sort (TS)
  1. Output a vertex u with in-degree zero in current graph.
  2. Remove u and all edges (u, v) from current graph.
  3. If graph is not empty, goto step 1.

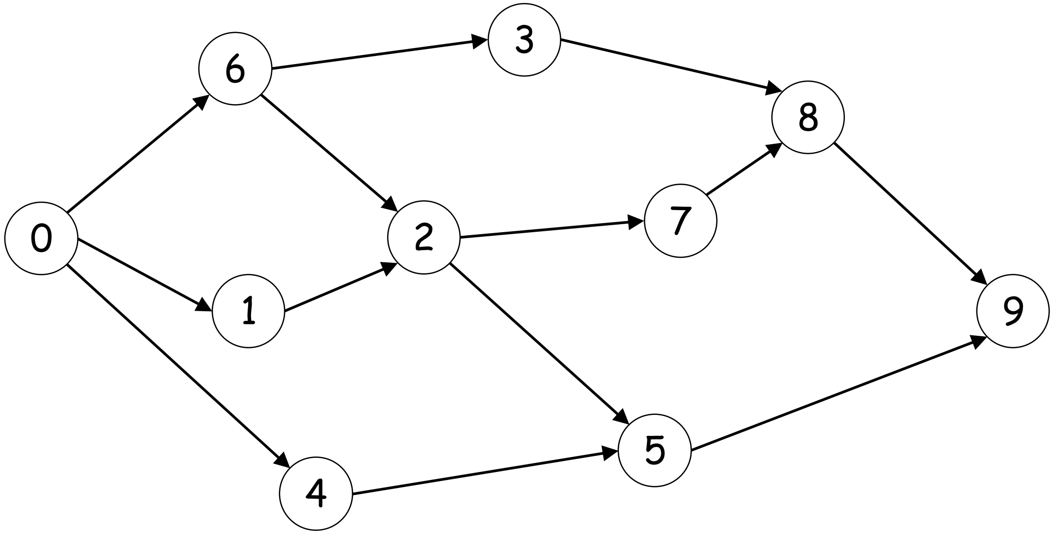
- Correctness
  - At every stage, current graph remains a DAG (why?)
  - Because current graph is always a DAG, TS can always output some vertex. So algorithm outputs all vertices.
  - Suppose output order is **not** a topological order.
    => Then there is some edge (u, v) such that v appears before u in the order. This is impossible, though, because v can not be output until edge (u, v) is removed!

Topological Sort Algorithm

Topological Sort(G)

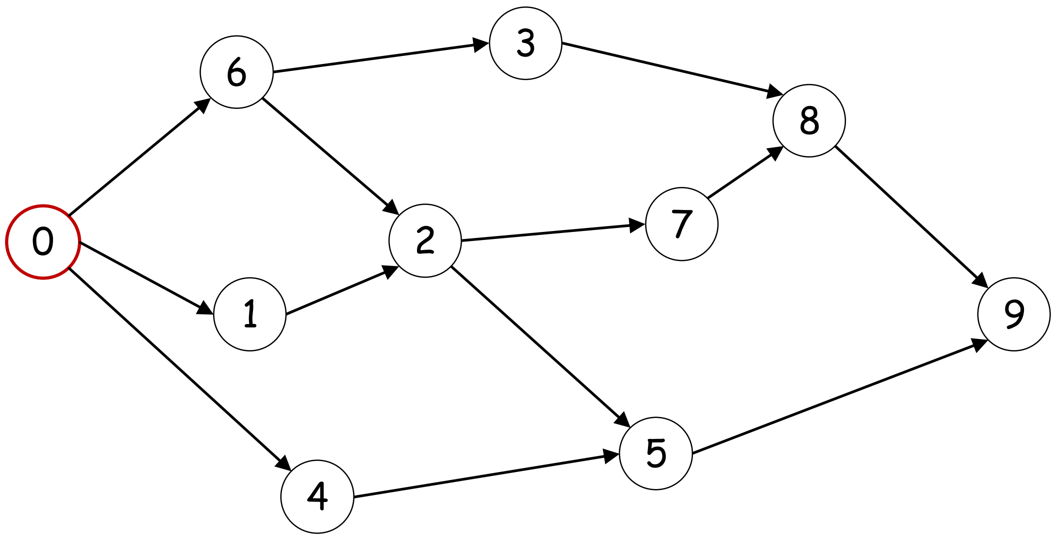
```
Initialize Q to be an empty queue;
foreach u in V do
    If in−degree(u) = 0 then
        // Find all starting vertices
        Enqueue(Q, u);
    end
end
while Q is not empty do
    u = Dequeue(Q);
    Output u;
    foreach v in Adj(u) do
        // remove u's outgoing edges
        in−degree(v) = in−degree(v) − 1
        if in−degree(v) = 0 then
            Enqueue(Q, v);
        end
    end
end
```

Example



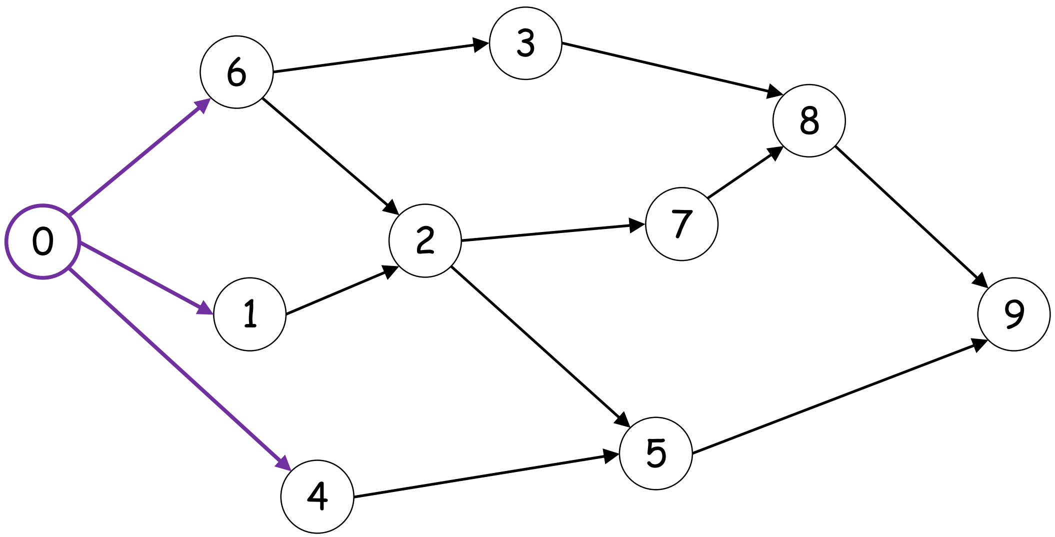
$Q = \{\}$

Example



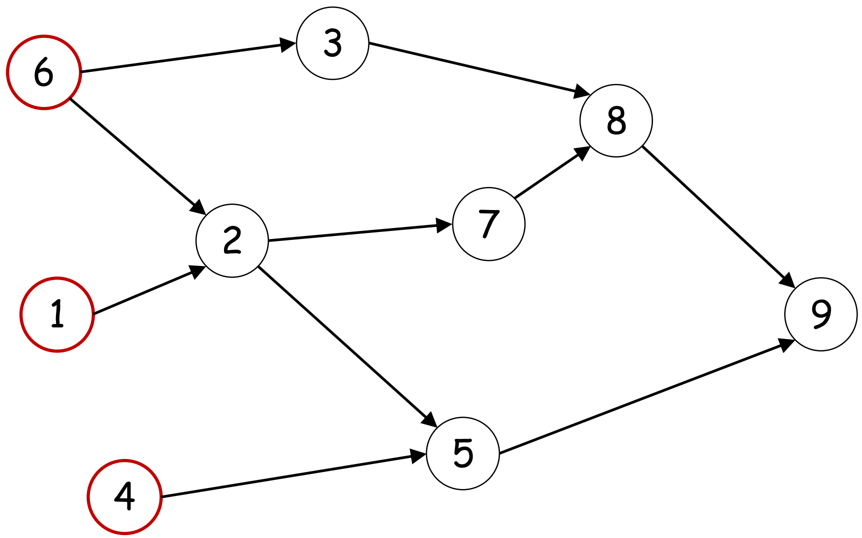
$Q = \{0\}$

Example



$Q = \{0\}$

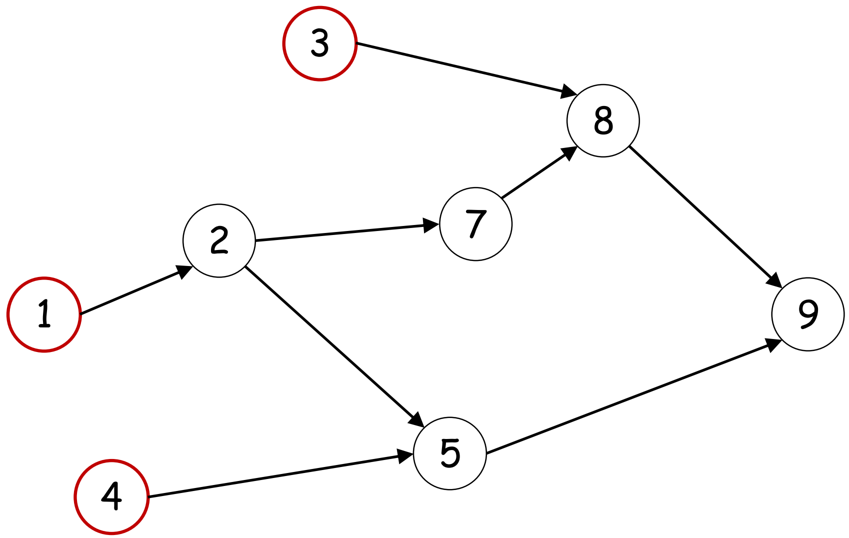
Example



$Q = \{6,1,4\}$

Output: 0

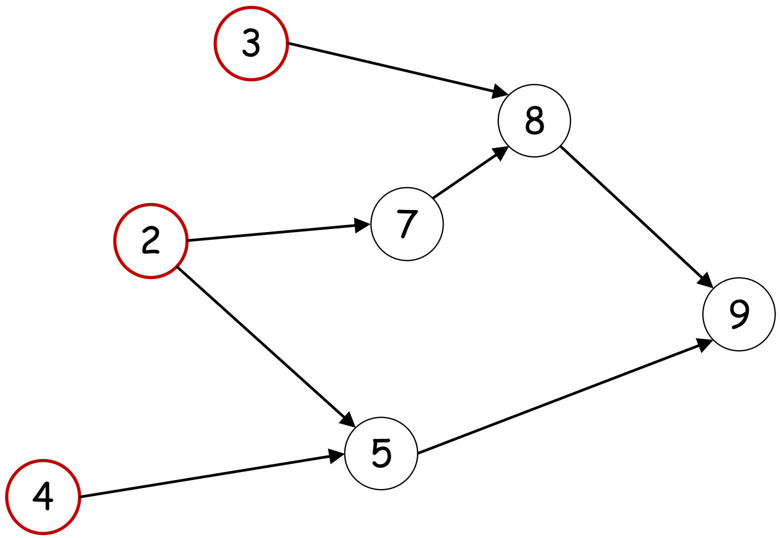
Example



$Q = \{1,4,3\}$

Output: 0,6

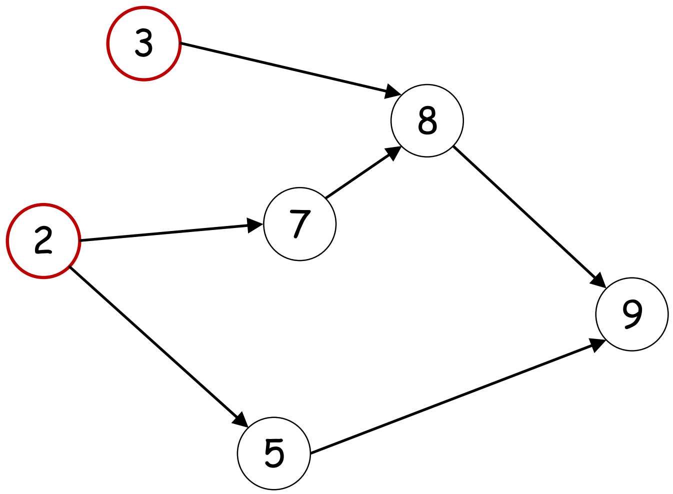
Example



$Q = \{4,3,2\}$

Output: 0,6,1

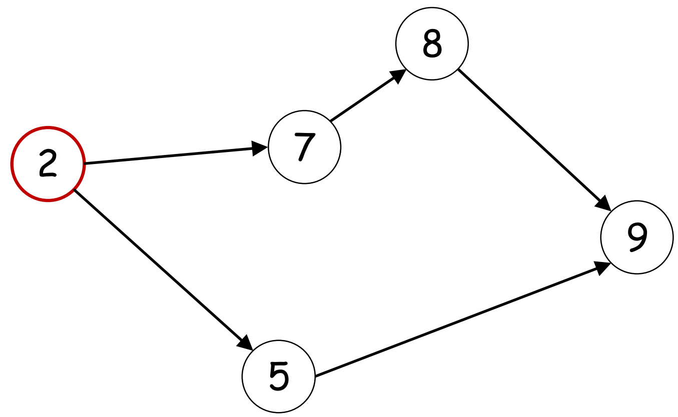
Example



$$Q = \{3,2\}$$

Output: 0,6,1,4

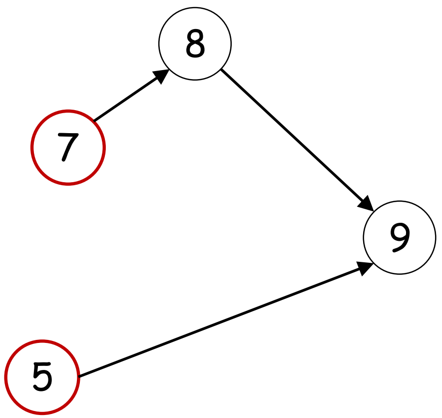
Example



$Q = \{2\}$

Output: 0,6,1,4,3

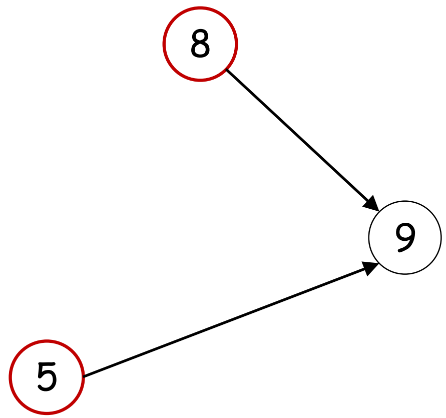
Example



$$Q = \{7,5\}$$

Output: 0,6,1,4,3,2

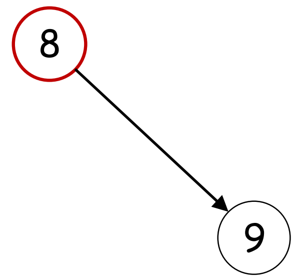
Example



$$Q = \{5,8\}$$

Output: 0,6,1,4,3,2,7

Example



$Q = \{8\}$

Output: 0,6,1,4,3,2,7,5

Example

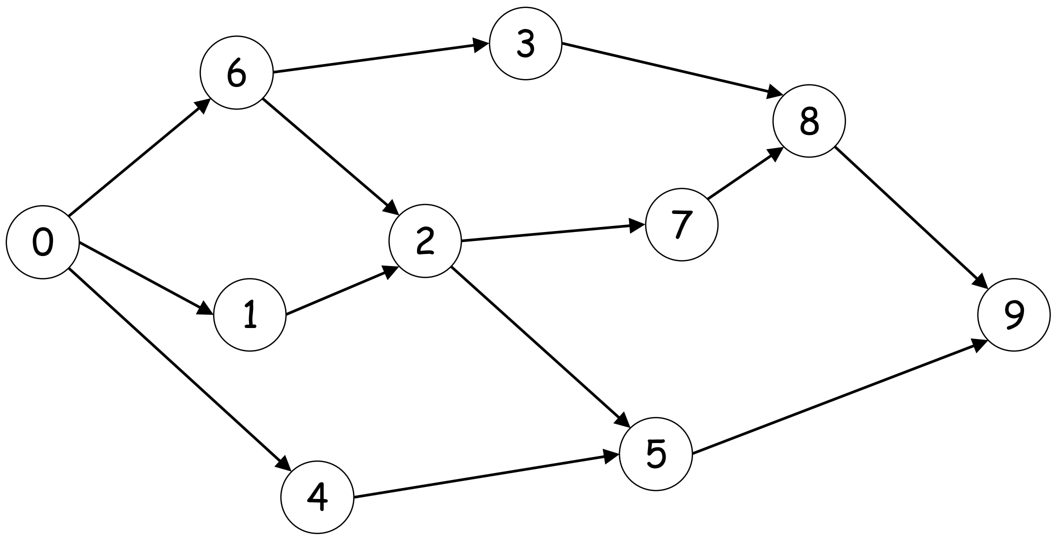
$Q = \{9\}$

Output: 0,6,1,4,3,2,7,5
,8

Example

$Q = \{\}$

Output: 0,6,1,4,3,2,7,5,
8,9



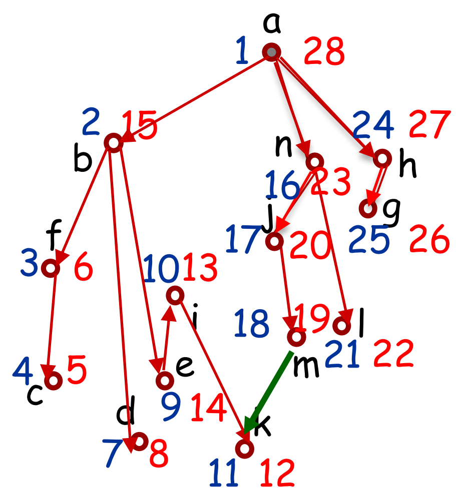
Done!

Topological Sort: Complexity

- We never visit a vertex more than once
- For each vertex, we examine all outgoing edges
  - $\sum_{v \in V}$ out-degree$(v) = E$
- Therefore, the running time is $O(V + E)$

Exercise DFS for Topological Sort

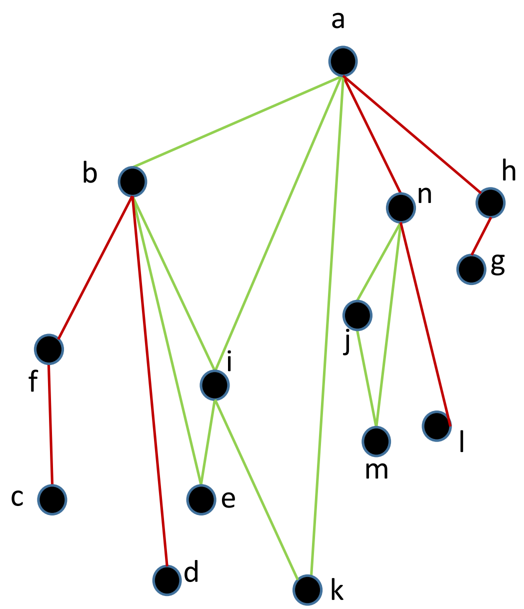
- Q: Can we use DFS to implement topological sort?



- Apply DFS from a node that has in-degree 0
- Output the nodes in decreasing order of finishing time:
- Example:
- a, h, g, n, l, j, m, b, e, ...

Exercise on Bridges

Given a connected undirected graph, a bridge is an edge whose removal disconnects the graph.



Describe a O(E·V) algorithm, to find all bridges in the graph

Remove each edge and start DFS or BFS from any node. If the traversal does not reach all nodes, the removed edge is a bridge.

A traversal has cost O(V), for each removed edge. Total cost: O(E·V)

Can you find all the bridges with a single DFS traversal?

Yes. Main idea similar to cycle detection: if an edge is part of a cycle, then it is not a bridge.