Portland State University

W'21 CS 584/684

Algorithm Design & Analysis

Fang Song

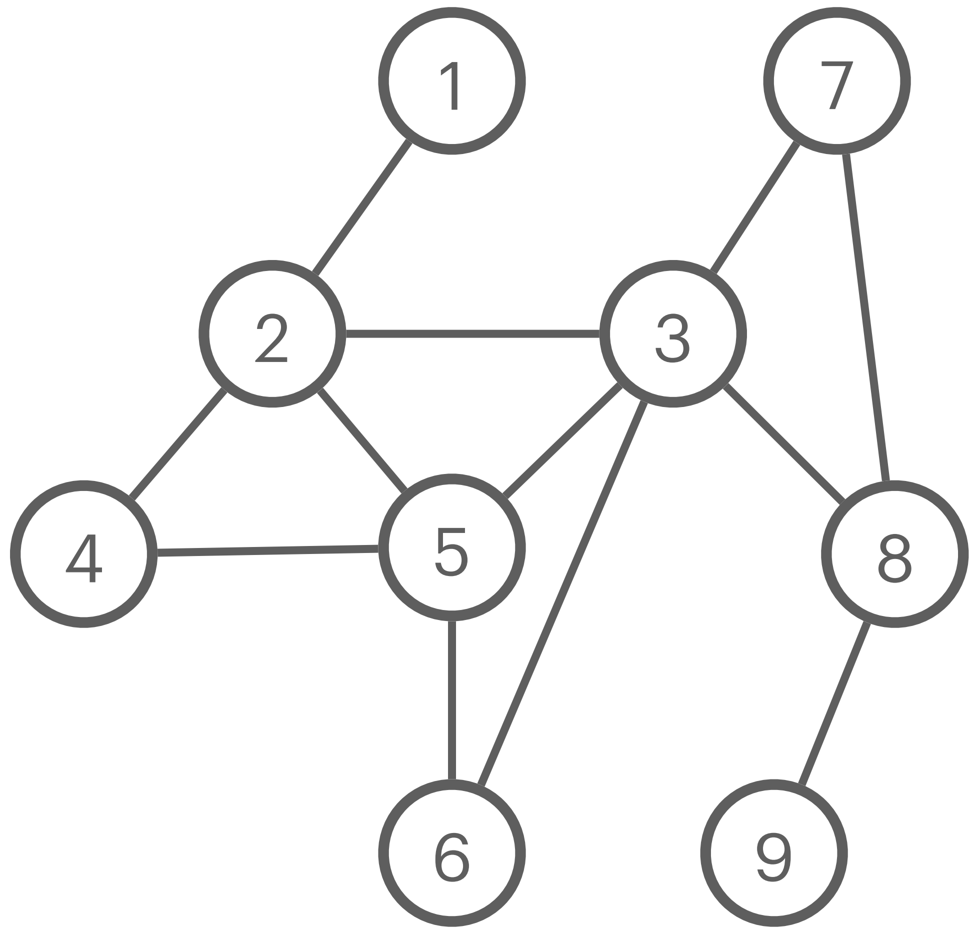
Lecture 5

- Connected components
- DAG & Topological order

Credit: based on slides by A. Smith & K. Wayne

Warm-up exercises

- True or False. A tree of n vertices can have n edges.
- Run BFS and DFS starting at node s, and form BFS/DFS trees. Decide if nodes 1 and 9 are connected.



Recap: BFS running time

Theorem. BFS takes $O(m+n)$ time (linear in input size).

Why not $n \cdot m$?

BFS(s):
// Discoverd[1,...,n] array of bits (explored or not), initialized to all zeros.
// Queue $Q \leftarrow \varnothing$
1. Set Discovered[s] = 1
2. EnQ(s) // add s to Q
3. While Q not empty DeQ(u)
    For each (u,v) incident to u
        If Discovered[v]=0 then
            Set Discovered[v]=1
            Add edge (u,v) to T
            EnQ(v)

$O(1)$, run once for all

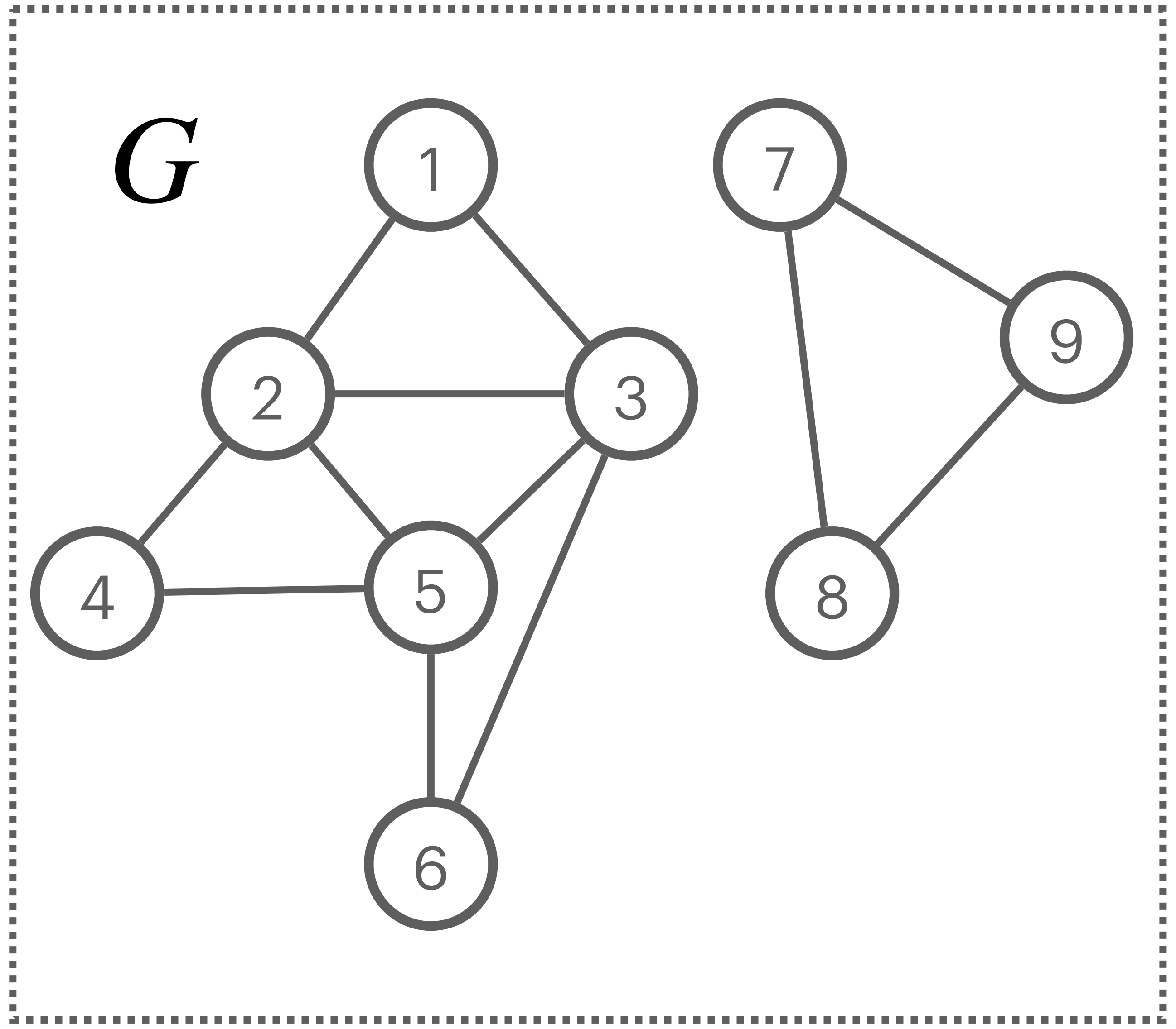
$O(1)$, run once per vertex

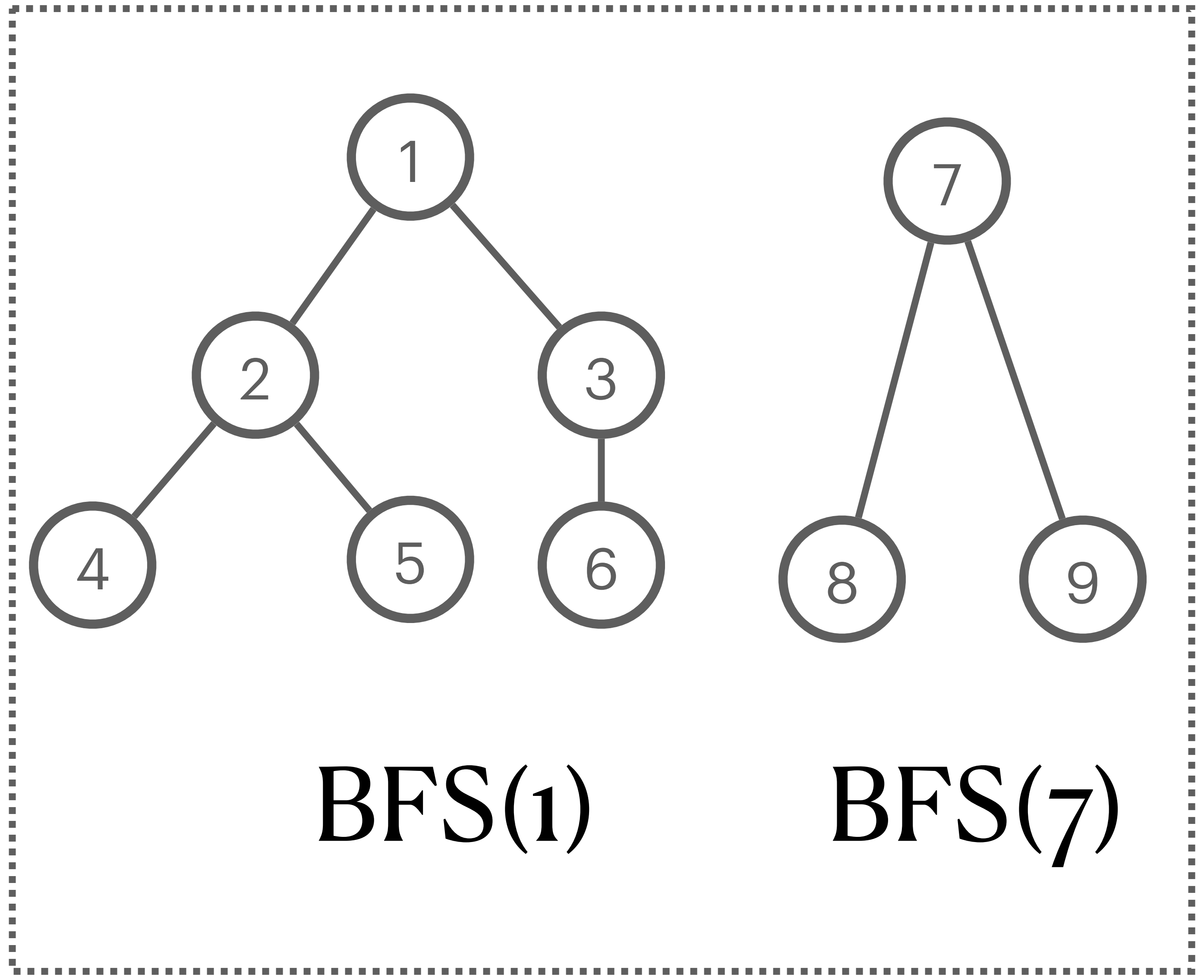
$O(1)$, run $\leq$ twice per edge

Connected components

- B/DFS tell more than s-t connectivity.

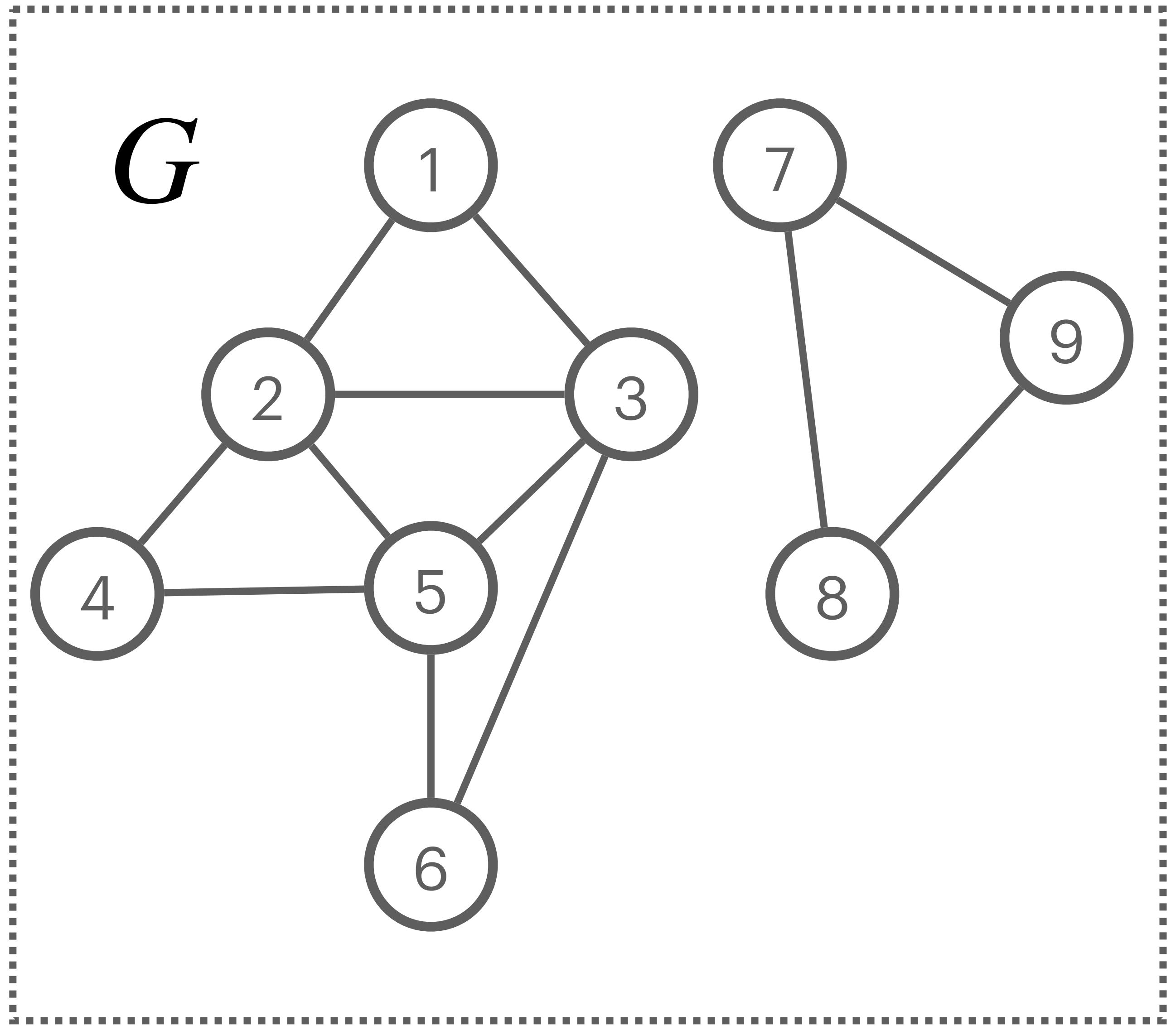
Connected component of G containing s: all nodes reachable from s.

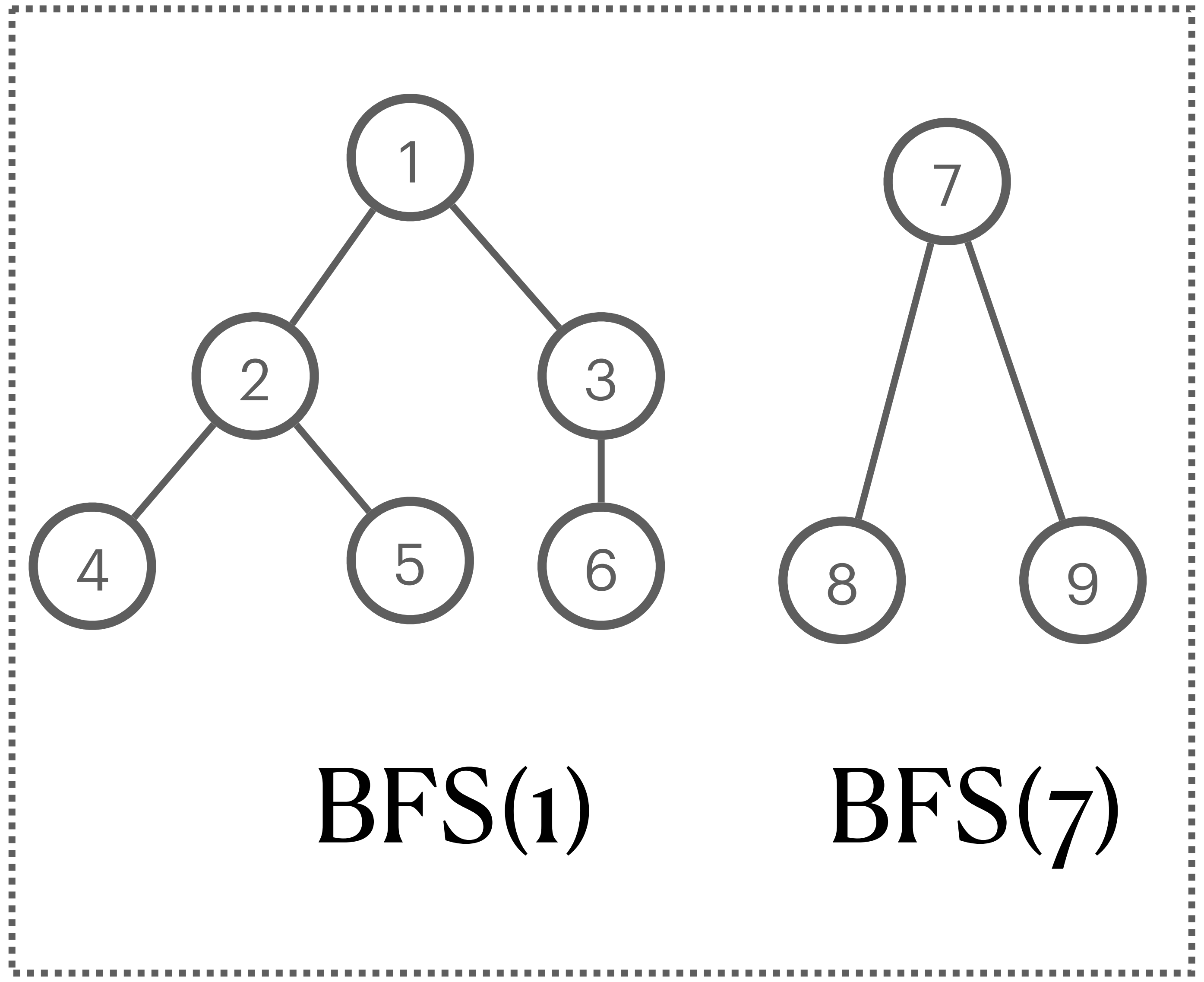




- Claim. For any tow nodes s and t, their connected components are either identical or disjoint.

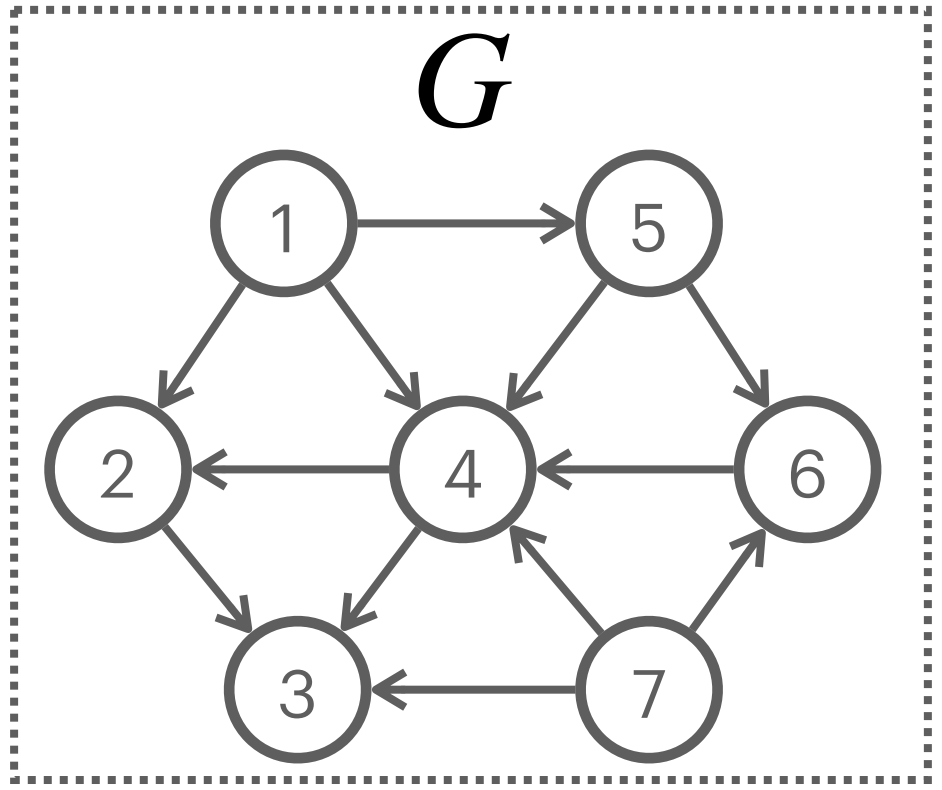
The set of all connected components





- **How to find all?**
  - Iterate over V, run B/DFS.
- **How fast?**
  - $\sum_i n_i + m_i = O(m+n)$.
- **Why care?**
  - Basic topology about G.

Directed graphs



- A directed graph $G = (V, E)$
  - Edge $u \to v$ leaves node u and enters node v.
  - Adjacency matrix: asymmetric
  - Adjacency list: track outgoing edges (or two for in and out)

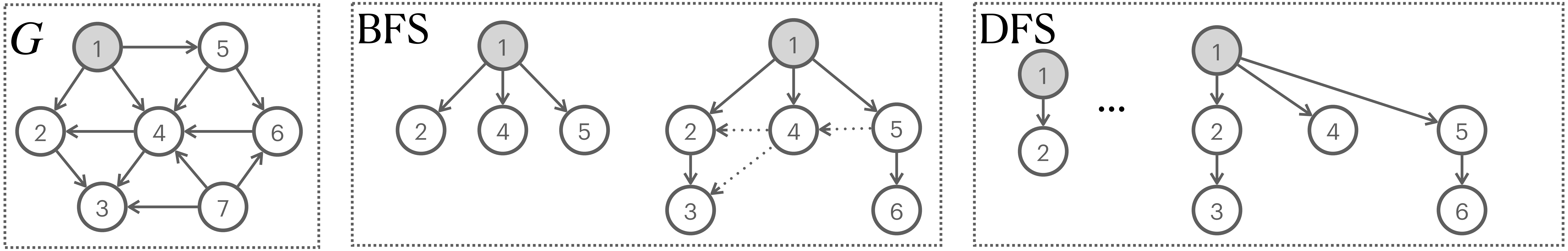
… $Adj_{\text{out}}[2] = \{3\}, Adj_{\text{in}} = \{1,4\}$

- Examples.

Directed graph	Node	Directed edges
Transportation	Intersections	One-way street
Social network	People	Following
Web	Webpage	Hyperlink
Citation	Article	Citing

Connectivity in directed graphs

- **Directed reachability.** **Find all nodes reachable from a node s.**
  - BFS/DFS apply.
  - $s \rightsquigarrow t$: there is a path from s to t. Need not be $t \rightsquigarrow s$.



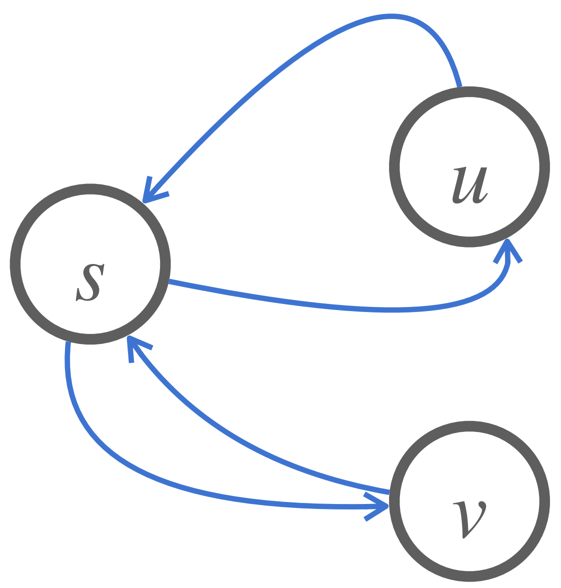
- **Application: web crawler.**
  - Start from web page s. Find all web pages linked from s, via one or more hops.

Strong connectivity

- **Def.** u and v are mutually reachable ($u \leftrightsquigarrow v$)
- **Observation.** If $u \leftrightsquigarrow v$ and $v \leftrightsquigarrow w$, then $u \leftrightsquigarrow w$.

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

Lemma. Let s be any node. G is strongly connected iff. every node is reachable from s, and s is reachable from every node.



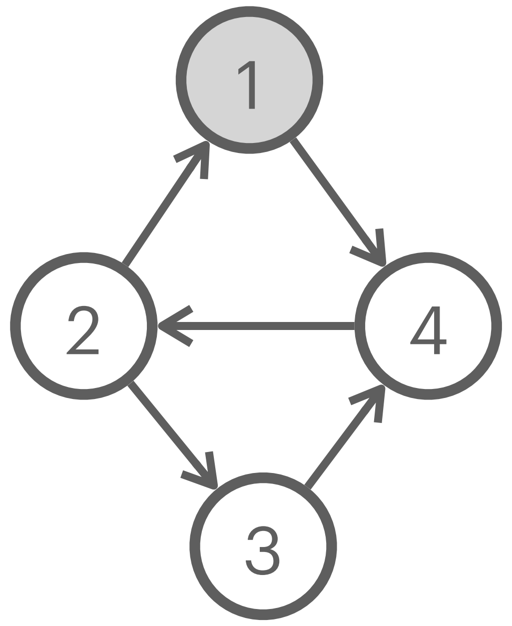
- **Proof.** [Show both "if" and "only if"]
  - $\Rightarrow$ (only if) By definition of "strongly connected".
  - $\Leftarrow$ (if) for any two nodes u, v: $u \rightsquigarrow v$ by following $u \rightsquigarrow s$ then $s \rightsquigarrow v$.
    $v \rightsquigarrow u$ by following $v \rightsquigarrow s$ then $s \rightsquigarrow u$.

Testing strong connectivity

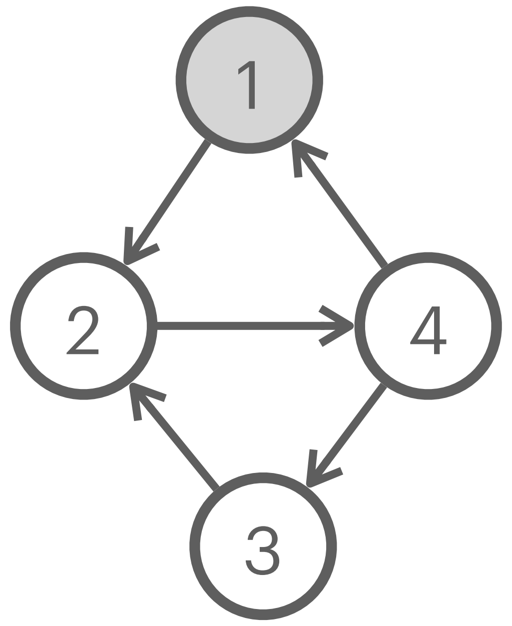
Theorem. Theres is an $O(m + n)$ time algorithm that determines if G is strongly connected.

- **Proof.** [construction of an algorithm. Fill in the analysis on your own.]

G



G^{rev}: reverse orientation of all edges in G.

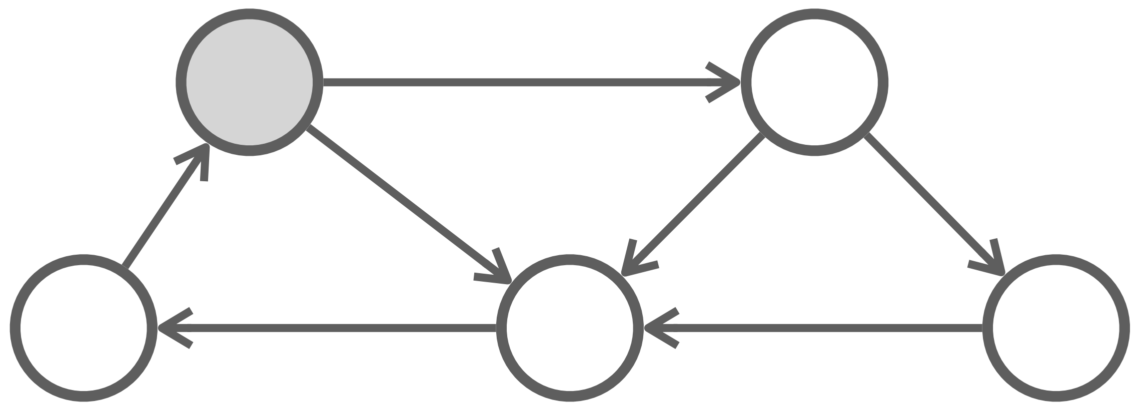


1. Pick any node s.
2. Run **BFS** from s on G.
3. Run **BFS** from s on G^{rev}.
4. Return true if all nodes reached in both **BFS** runs.

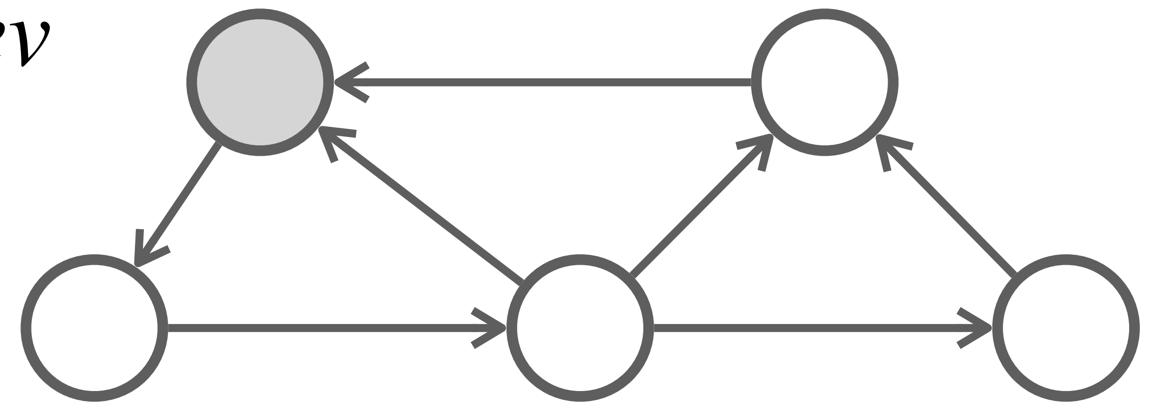
Exercise

- Determine if the graph is strongly connected.

G



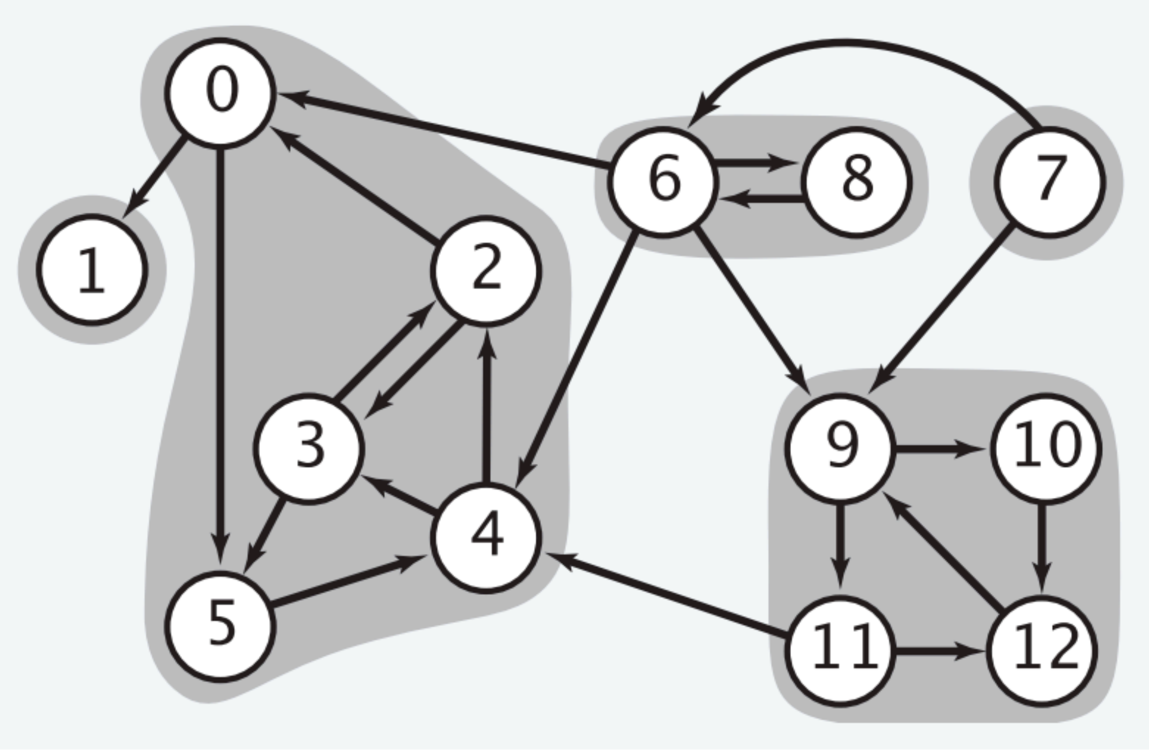
G^{rev}



Strong (connected) components

- Def. A strong component is a maximal subset of mutually reachable nodes.
- Obs. For any two nodes s and t in a directed graph, their strong components are either identical or disjoint.

Theorem. Theres is an $O(m+n)$ time algorithm that finds all strong components.



SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

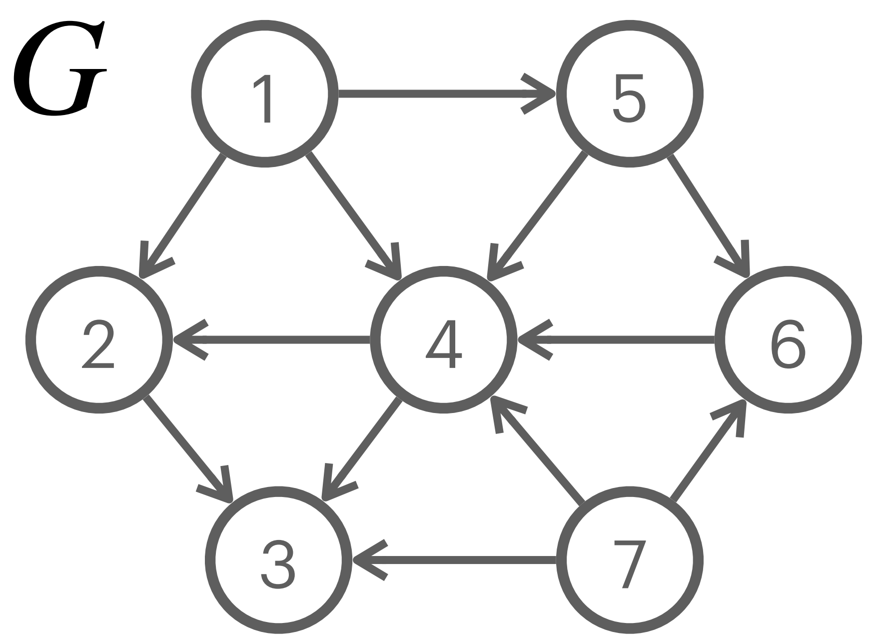
DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

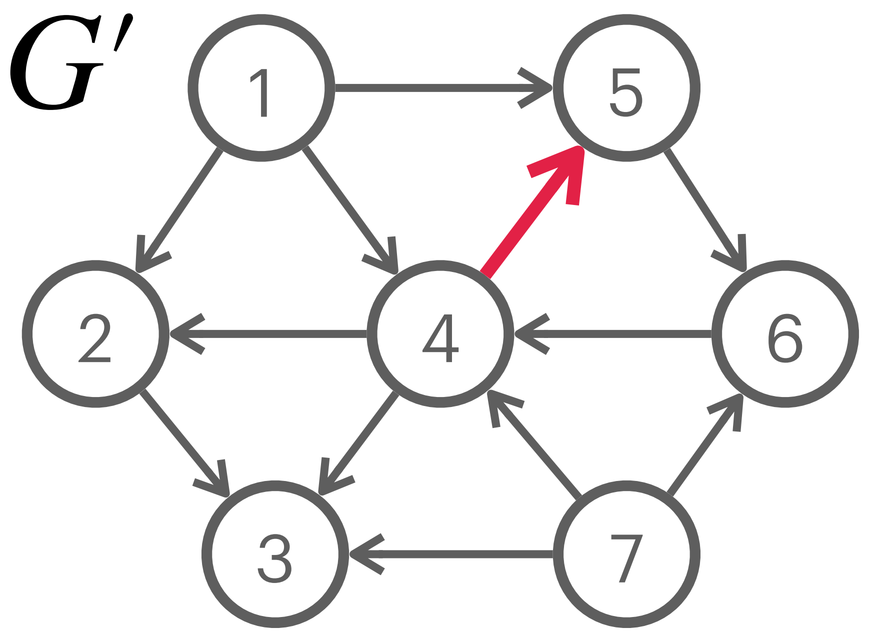
ROBERT TARJAN†

Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an un-direct graph are presented. The space and time requirements of both algorithms are bounded by

Directed acyclic graphs (DAG)

- **Def. A DAG is a directed graph that contains no directed cycles.**

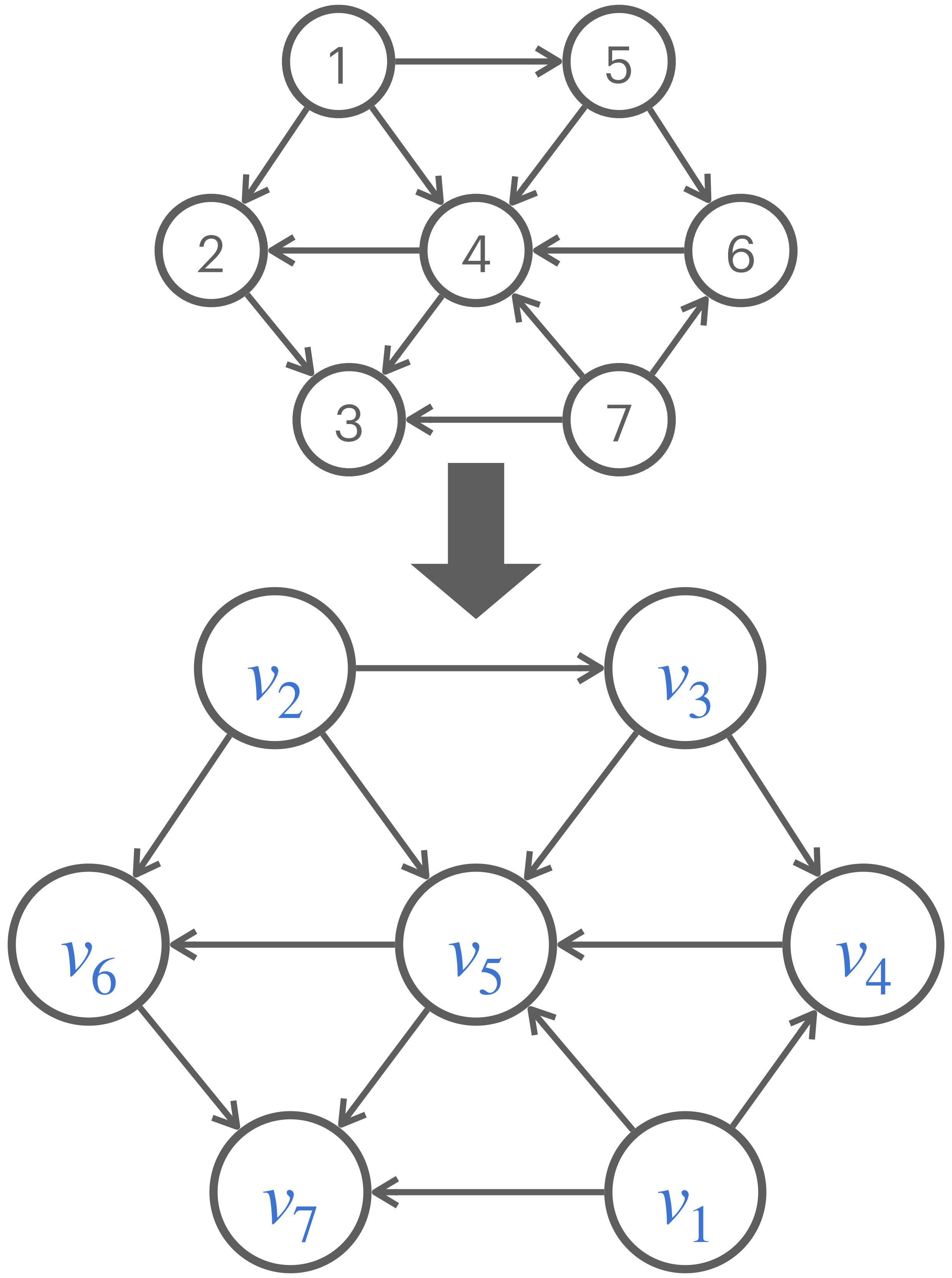




- **Application: precedence constraints.**
  - Course prerequisite: 350 must be taken before 584/684.
  - Compilation: module i must be complied before j.
  - Pipeline of computing jobs: output of job i determines input of job j.

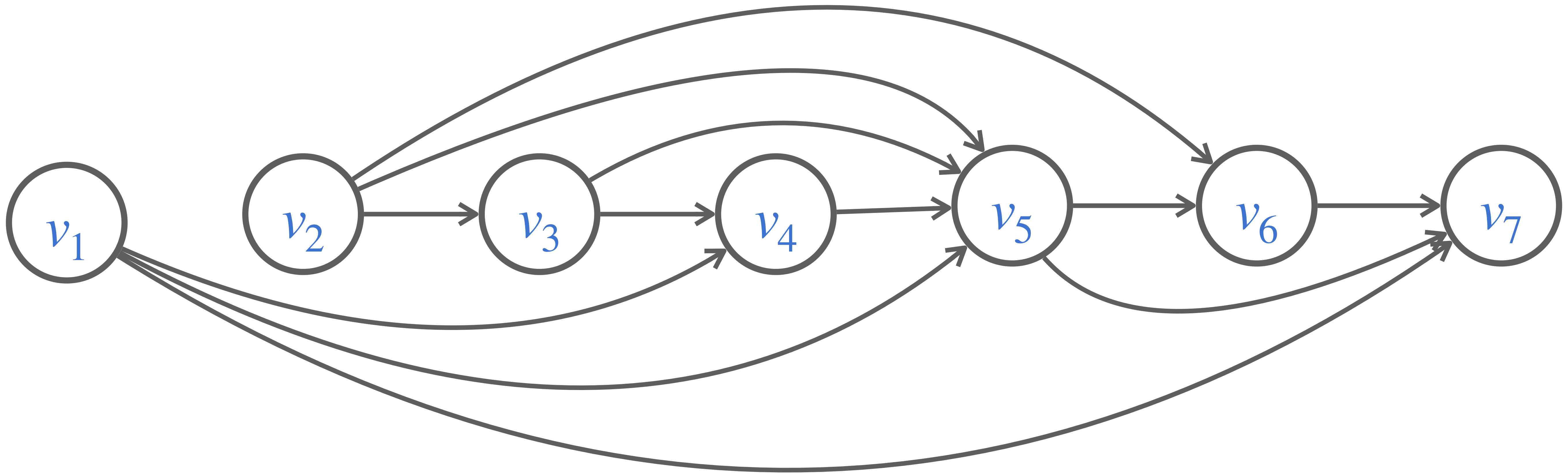
Topological order

- **Def.** A topological order of a directed graph is an ordering of its nodes $v_1, \ldots, v_n$, so that for every edge $v_i \to v_j$ we have $i < j$.



A topological order

All edges go from left to right

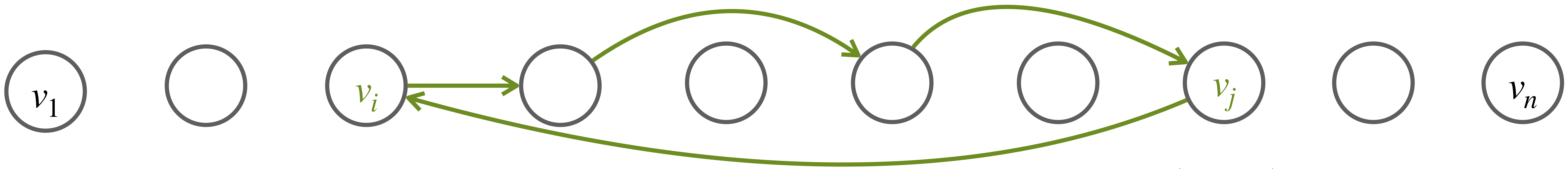


1. If G has a topological order, is G necessarily a DAG?
2. Does every DAG have a topological order?

Q1: If G has a topological order, is G necessarily a DAG?

Lemma 1. If G has a topological order, then G is a DAG.

- **Proof** [by contradiction]
  - Suppose G has topological order $v_1, \ldots, v_n$; and G also has a directed cycle C.
  - Let v_i be the lowest-indexed node in C, v_j be the node just before v_i in C.
  - Then $v_j \to v_i$ is an edge & by our choice $i < j$.
  - But since $v_1, \ldots, v_n$ is a topological order, if $v_j \to v_i$ is an edge, then $j < i$.
  - Contradiction!



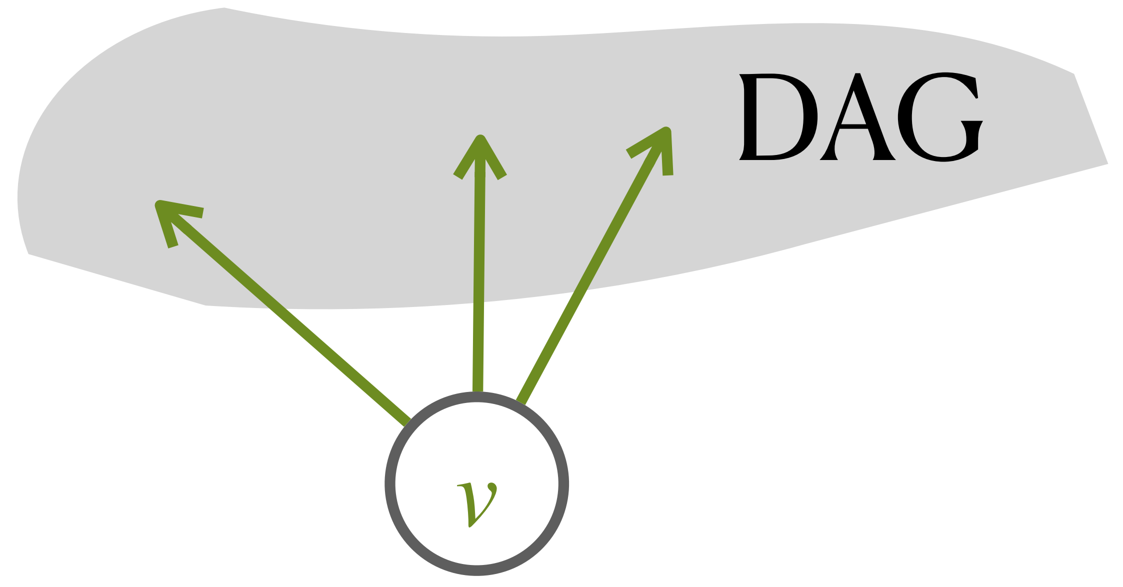
Directed cycle C

Q2: Dose every DAG have a topological order?

Lemma 2. A DAG G has a node with no entering edges.

Corollary. If G is a DAG, then G has a topological order.

- **Proof** of corollary given Lemma 1 [by induction on number of nodes]
  - Base case: true if $n = 1$.
  - Given a DAG on $n > 1$ nodes, find a node v with no entering edges [Lemma 1].
    - $G - \{v\}$ is a DAG, since deleting v cannot create cycles.
  - Induction hypothesis, $G - \{v\}$ (with $n - 1$ nodes) has a topological order.
  - Place v first then append nodes of $G - \{v\}$ in topological order [valid because v has no entering edges].



Topological sorting algorithm

TopSort(G):
// count(w)= remaining number of incoming edges
// S = set of remaining nodes with no incoming edges
// $V[1,\ldots,n]$ topological order
1. Initialize S and $Count(\cdot)$ for all nodes
2. For $v \in S$
   Append v to V
   For all w with $v \to w$ // delete v from G
   $Count(w) --$
   If $Count(w) == 0$ add w to S

$O(n+m)$, a single scan of adjacency list

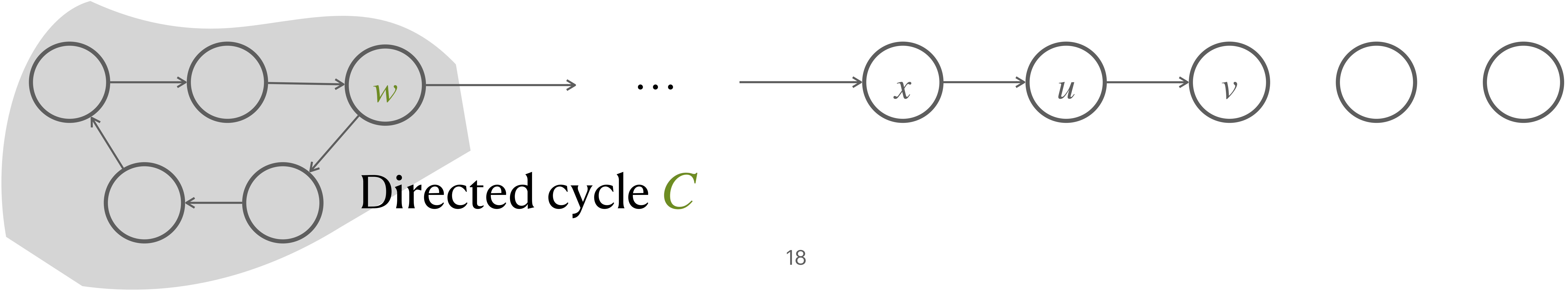
$O(1)$, run once per edge

Theorem. TopSort computes a topological order in $O(n+m)$ time.

Completing the proof

Lemma 1. A DAG G has a node with no entering edges.

- **Proof [by contradiction]**
  - Suppose G is a DAG, and every node has at least one entering edge.
  - Pick any node v, and follow edges backwards from v. Repeat till we visit a node, say w, twice. ($v \leftarrow u \leftarrow x \ldots \leftarrow w \ldots \leftarrow w$)
  - Let C be the sequence of nodes between successive visits to w.
  - C is a cycle. Contradiction!



Scratch