

CSCI 256 Lecture 7:
Directed Graphs & Topsort

Shikha Singh

Announcements

- Hand in **Exercise #5**, Pick up **Exercise #6**
- **Assignment 2** due Wed Sept 30 at 10 pm
 - Assigned partners are available on GLOW
- First in-class test is on Mon, Oct 5
 - Will cover topics up until Assignment 2
 - Chapter 1.1, 2.1-2.3 and 3 in KT
 - Will release practice midterm next week

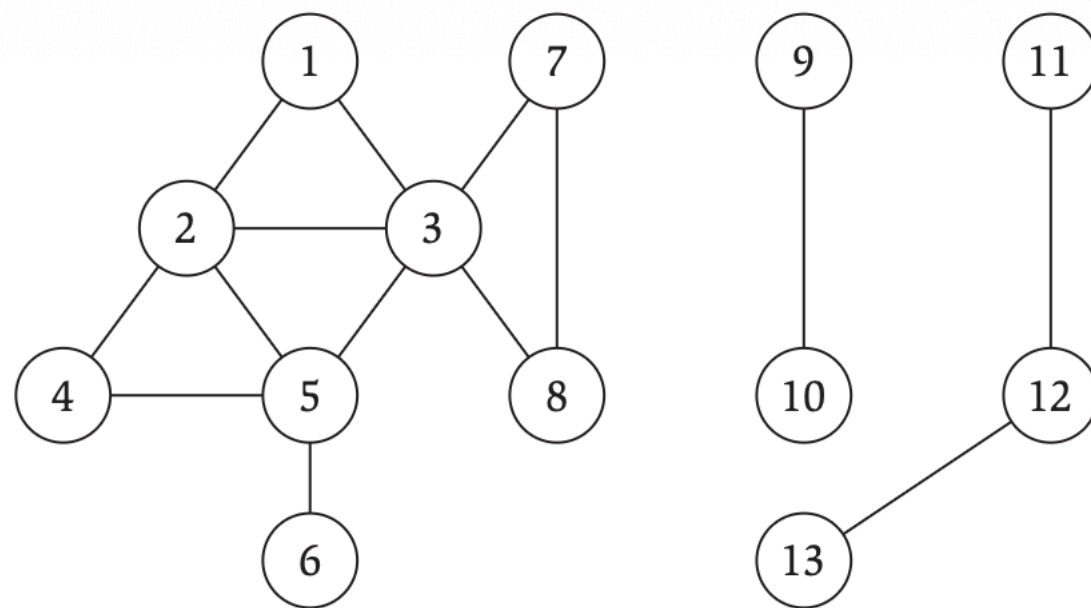
Last Time and Today

- Discussed depth-first search
 - Recursive and iterative (stack) implementation
 - Correctness & run time
- Applications of BFS & DFS for connectivity
- Directed graphs

BFS & DFS as Building Blocks

Connectivity

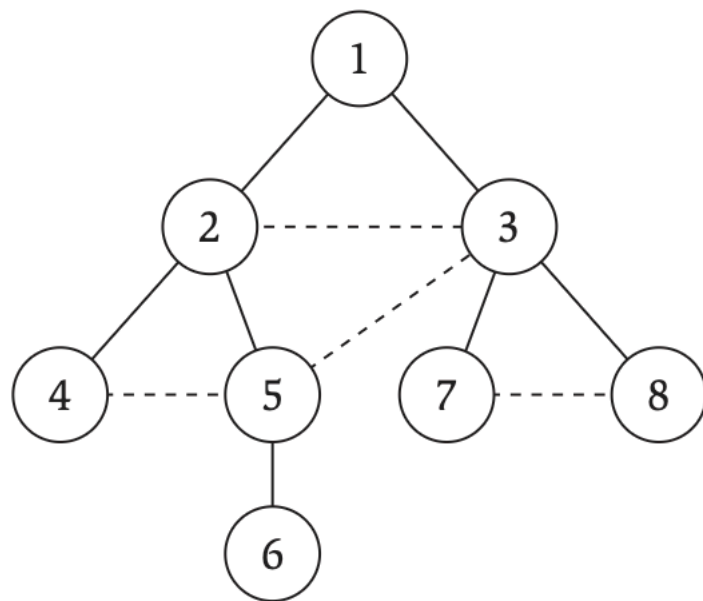
- A graph is **connected** if there is a path between every pair of nodes
- If a graph is disconnected: we consider its connected components which are maximally connected subgraphs



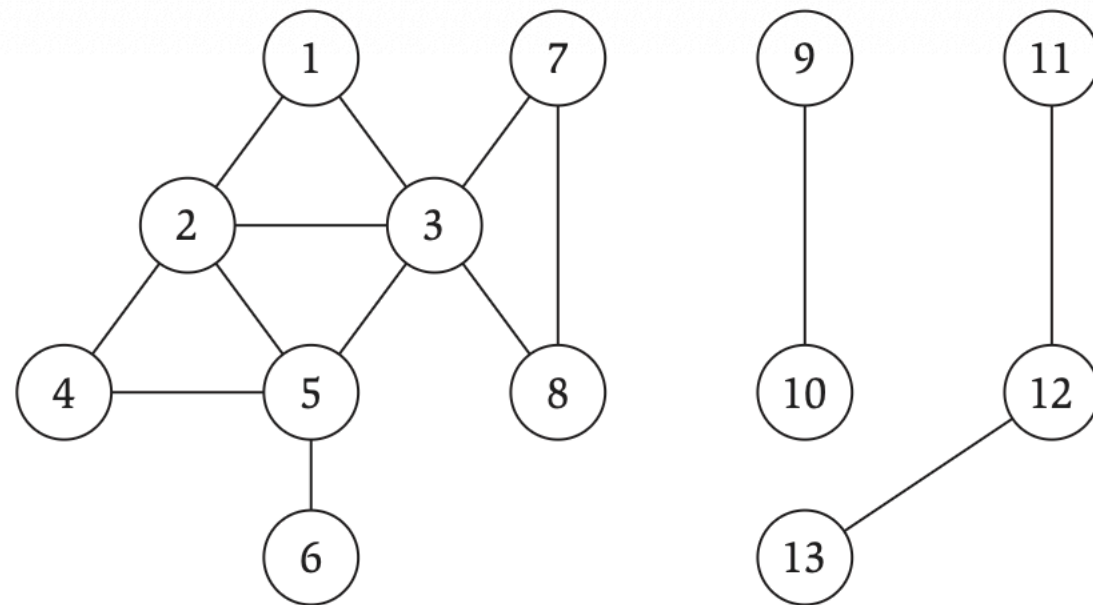
Connected components of an undirected graph

Spanning Trees & Components

- **Definition.** A **spanning tree** of an undirected graph G is a connected acyclic subgraph of G that contains every node of G .
- The tree produced by the BFS algorithm (with $((u, \text{parent}(u)))$ as edges) is a spanning tree of the component containing s .



BFS Spanning Tree of 1 (solid edges)

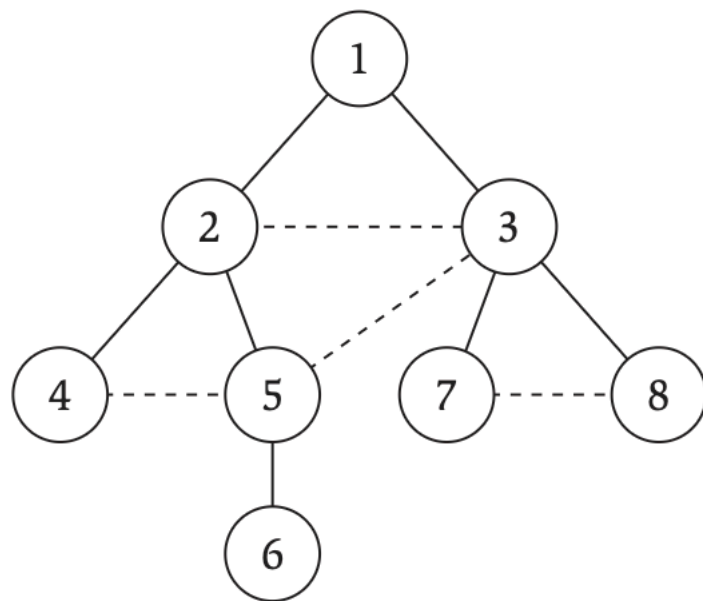


Graph G containing more than one component

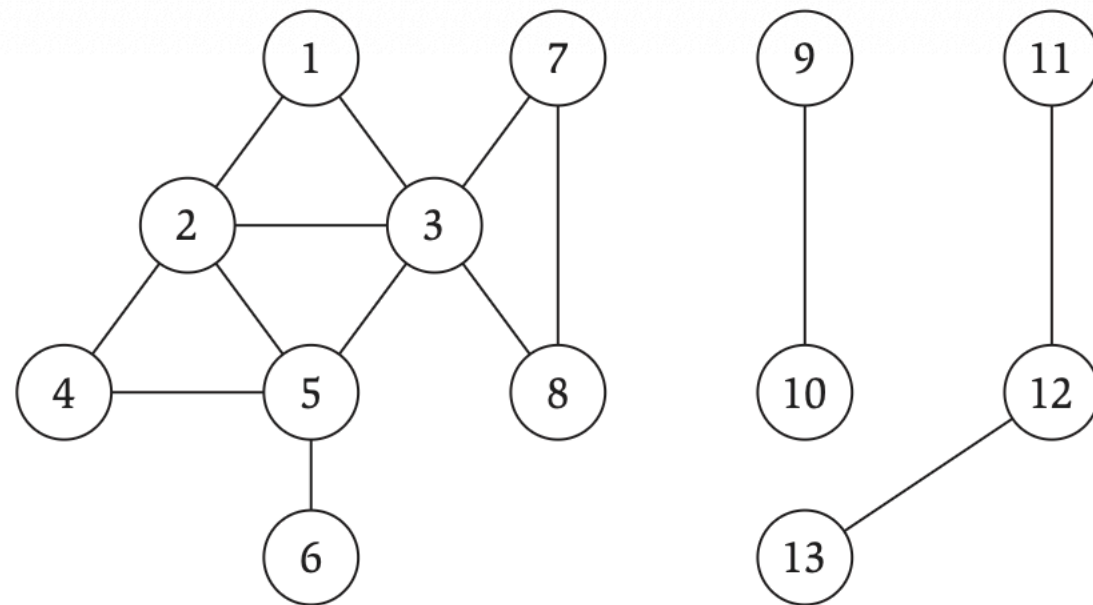
Finding Connected Components

Problem. Given an undirected graph G , determine if it is connected.

Observation. For any two nodes s and t , their components are either identical or disjoint.



BFS Spanning Tree of 1 (solid edges)



Graph G containing more than one component

Finding Connected Components

- How to determine whether a graph is connected using traversals?
 - If BFS tree contains all nodes, then the graph is connected
- **Question.** If the graph is not connected, how can we find all connected components?
 - Start BFS/DFS with any node s : all nodes in the BFS tree of s are one component
 - Pick another node that is not visited and repeat
 - Number of **trees** in resulting **forest** is the number of components
 - $O(n + m)$ time overall

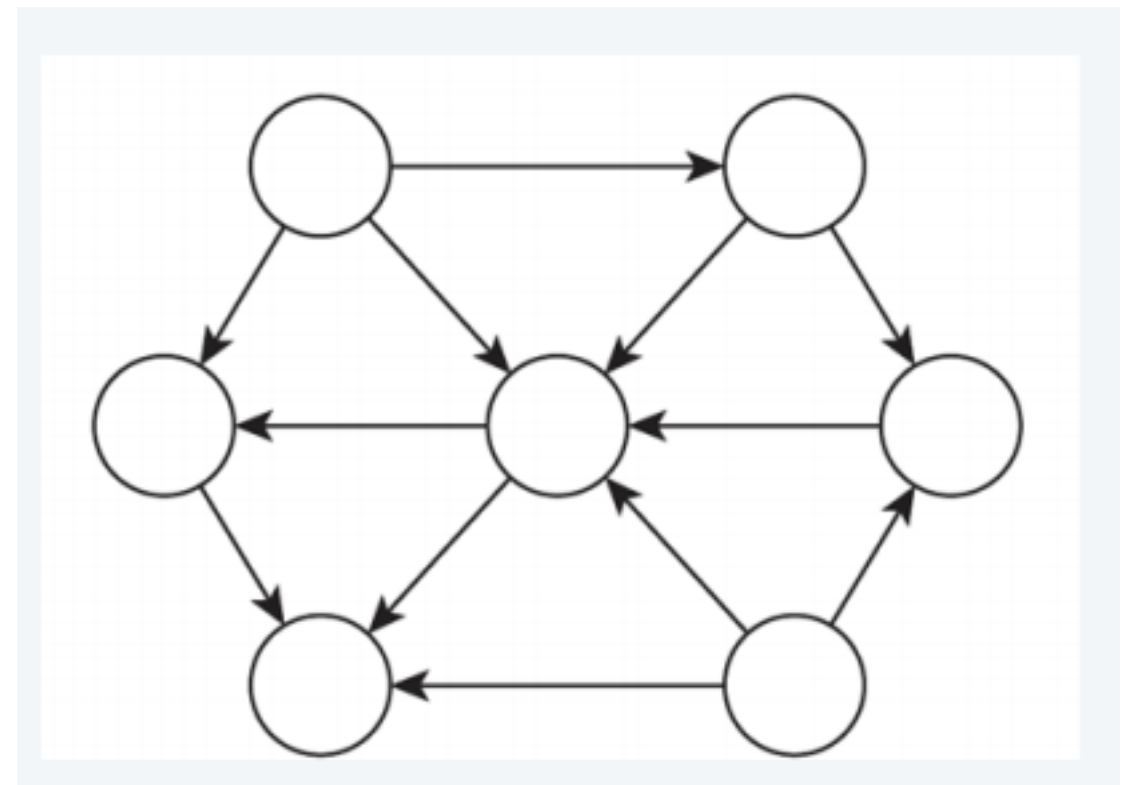
Directed Graphs

Directed Graphs

Notation. $G = (V, E)$.

- Edges have “orientation”
- Edge (u, v) or sometimes denoted $u \rightarrow v$, leaves node u and enters node v
- Nodes have “in-degree” and “out-degree”

Terminology of graphs extend to directed graphs: directed paths, cycles, etc.



Directed Graphs

Adjacency List. For each vertex $v \in V$

- Standard BFS/DFS: sufficient to just store outgoing neighbors
- Sometimes an algorithm needs finding incoming neighbors or in-degree, in which case can store both

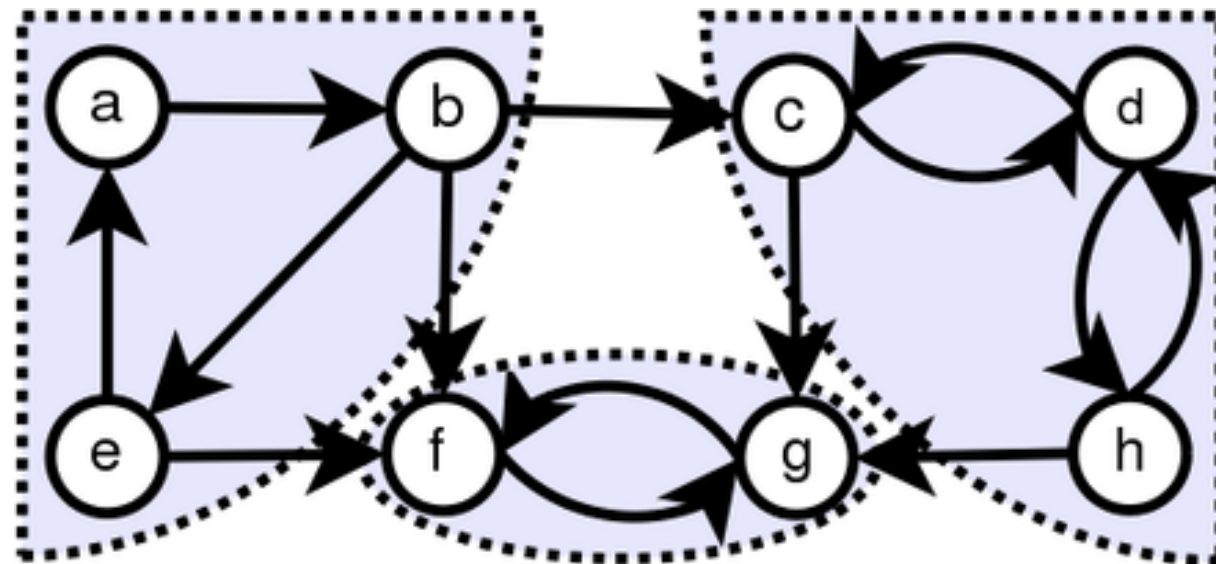
Most of the algorithms apply in a straightforward way:

Reachability. Given a node s find all nodes reachable from s .

- Can use both BFS and DFS. Both visit exactly the set of nodes reachable from start node s .

Strong Connectivity

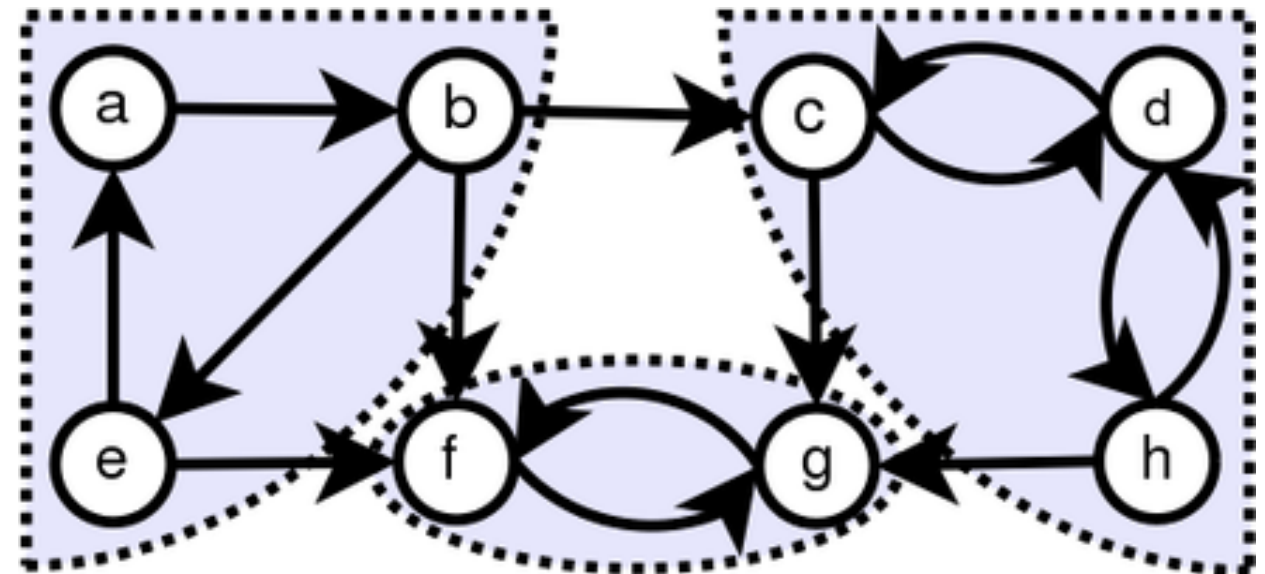
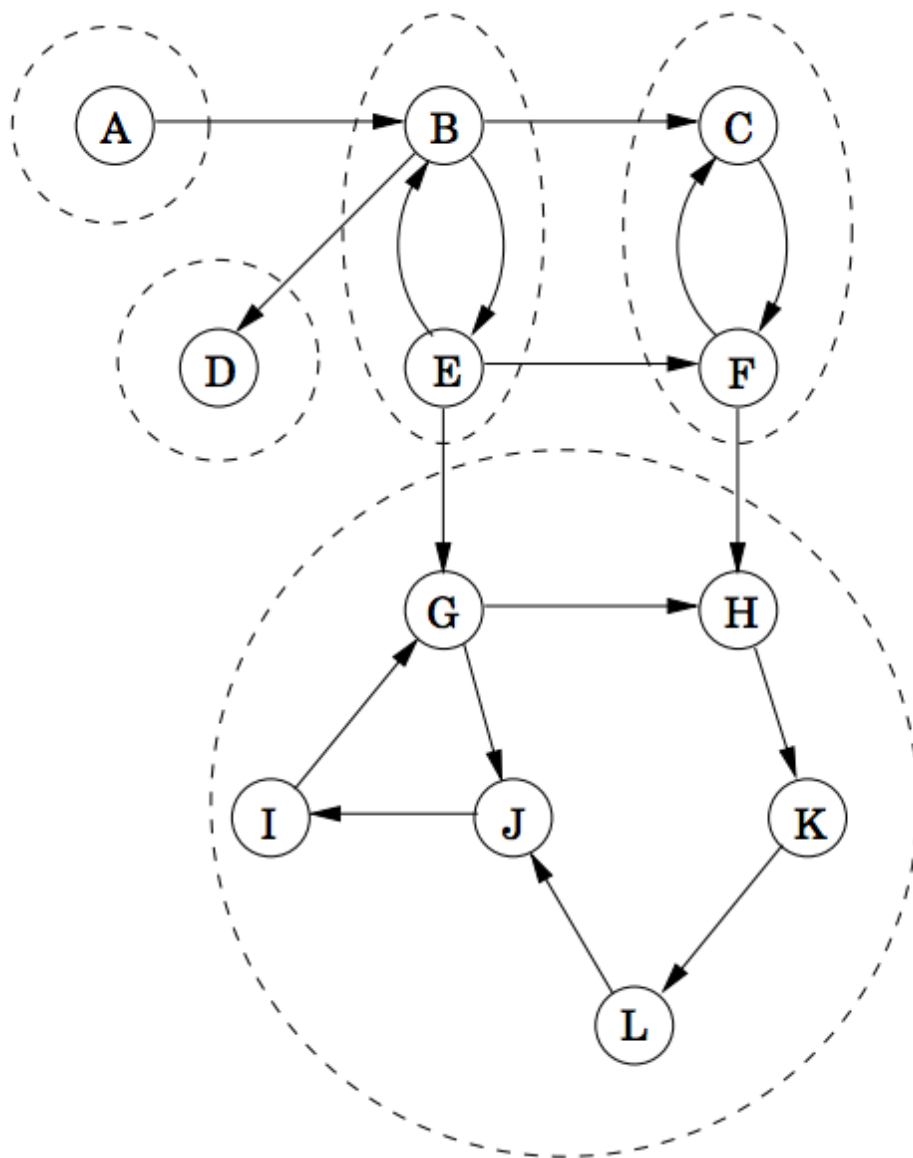
- **Strong connectivity.** Connected components in directed graphs defined based on **mutual reachability**. Two vertices u, v in a directed graph G are mutually reachable if u is reachable from v and v is reachable from u . A graph G is **strongly connected** if every pair of vertices are mutually reachable.



Wikipedia: Each shaded region is strongly connected

Strong Connectivity

- **Strongly-connected components.** For each $v \in V$, the set of vertices mutually reachable from v , defines the strongly-connected component of G containing v .



Deciding Strongly Connected

Problem. Given a directed graph G , determine if it is strongly connected.

Observation. For any two nodes s and t in a directed graph, their strong components are either identical or disjoint.

- So enough to find the strongly connected component of one vertex and check if it contains every node
- In an undirected graph, we performed a single BFS/DFS to determine this
- What changes now for a directed graph?

Deciding Strongly Connected

First idea. Pick a vertex s .

- Run BFS/DFS starting from s to see if every other vertex is reachable from s (was visited during it)
- Now see whether s is reachable from every other vertex: run BFS from every other vertex and check if s is visited

Analysis (Runtime).

- First step: one call to BFS: $O(n + m)$ time
- Second step: $n - 1$ calls to BFS: $O(n(n + m))$ time
- Can we do better?

Deciding Strongly Connected

Idea. Flip the direction of edges of G and create G_{rev}

- There is a directed path from v to u in G_{rev} iff there is a directed path from u to v in G
- Call **BFS**(G_{rev}, v): Every vertex is reachable from v (in G_{rev}) if and only if v is reachable from every vertex (in G).

Analysis (Runtime)

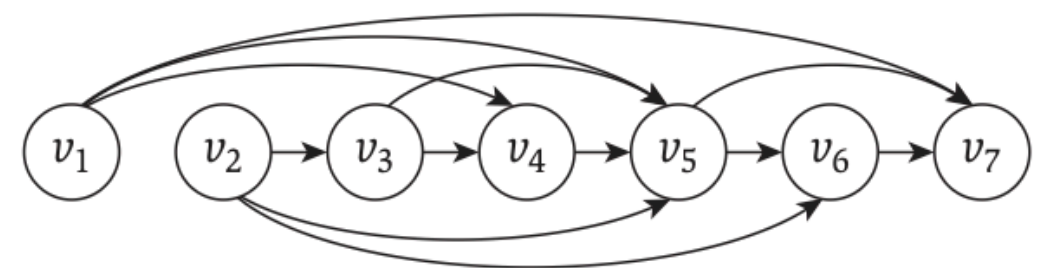
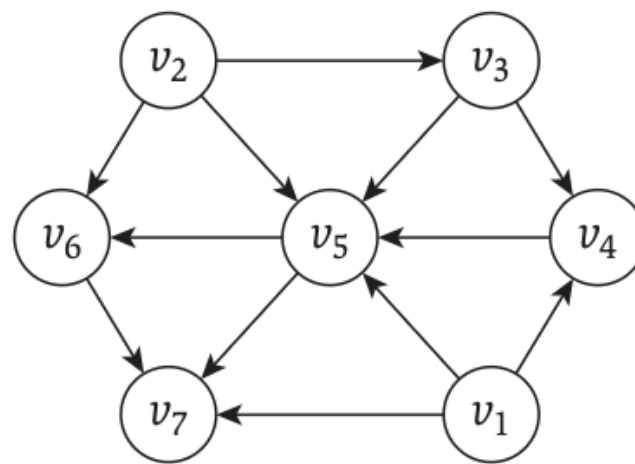
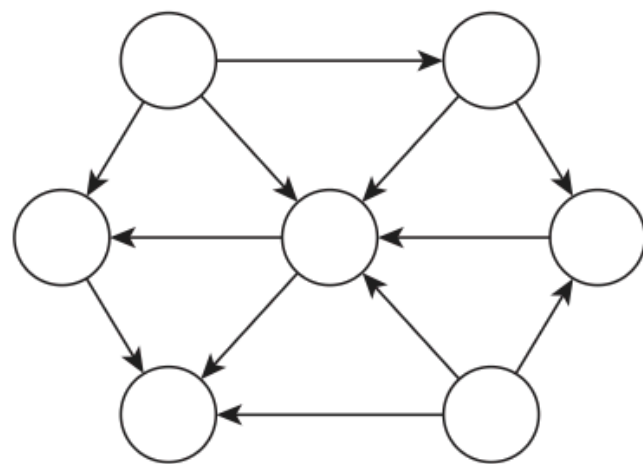
- **BFS**(G, v): $O(n + m)$ time
- Build G_{rev} : $O(n + m)$ time
- **BFS**(G_{rev}, v): $O(n + m)$ time
- Overall, linear time algorithm!

Kosaraju's Algorithm

Directed Acyclic Graphs (DAGs)

Definition. A directed graph is acyclic (or a DAG) if it contains no (directed) cycles.

- Very useful to model dependency graphs

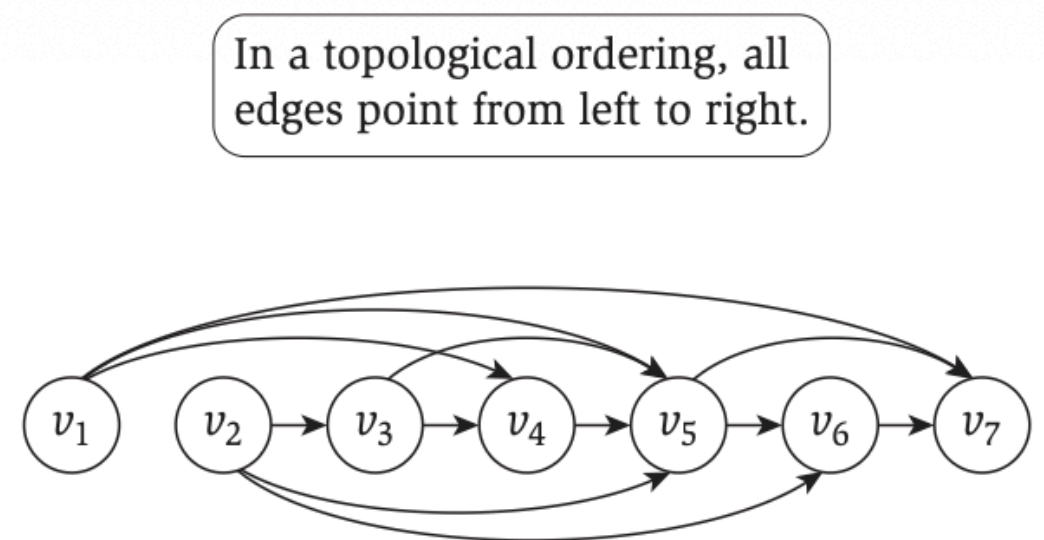
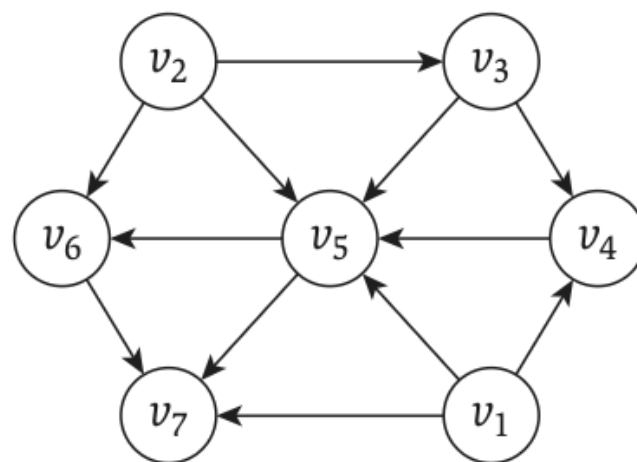
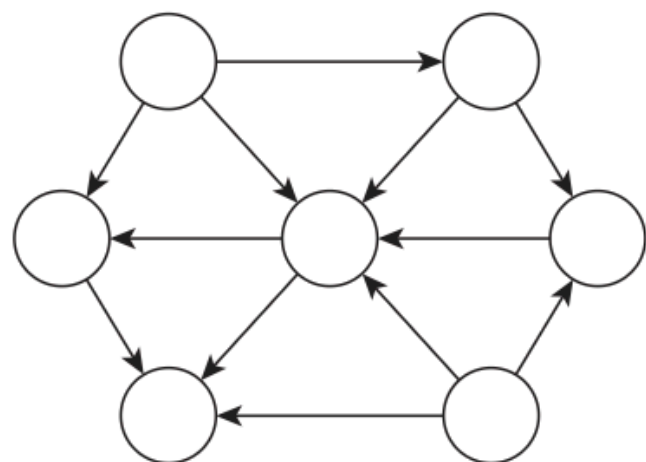


In a topological ordering, all edges point from left to right.

Directed Acyclic Graphs (DAGs)

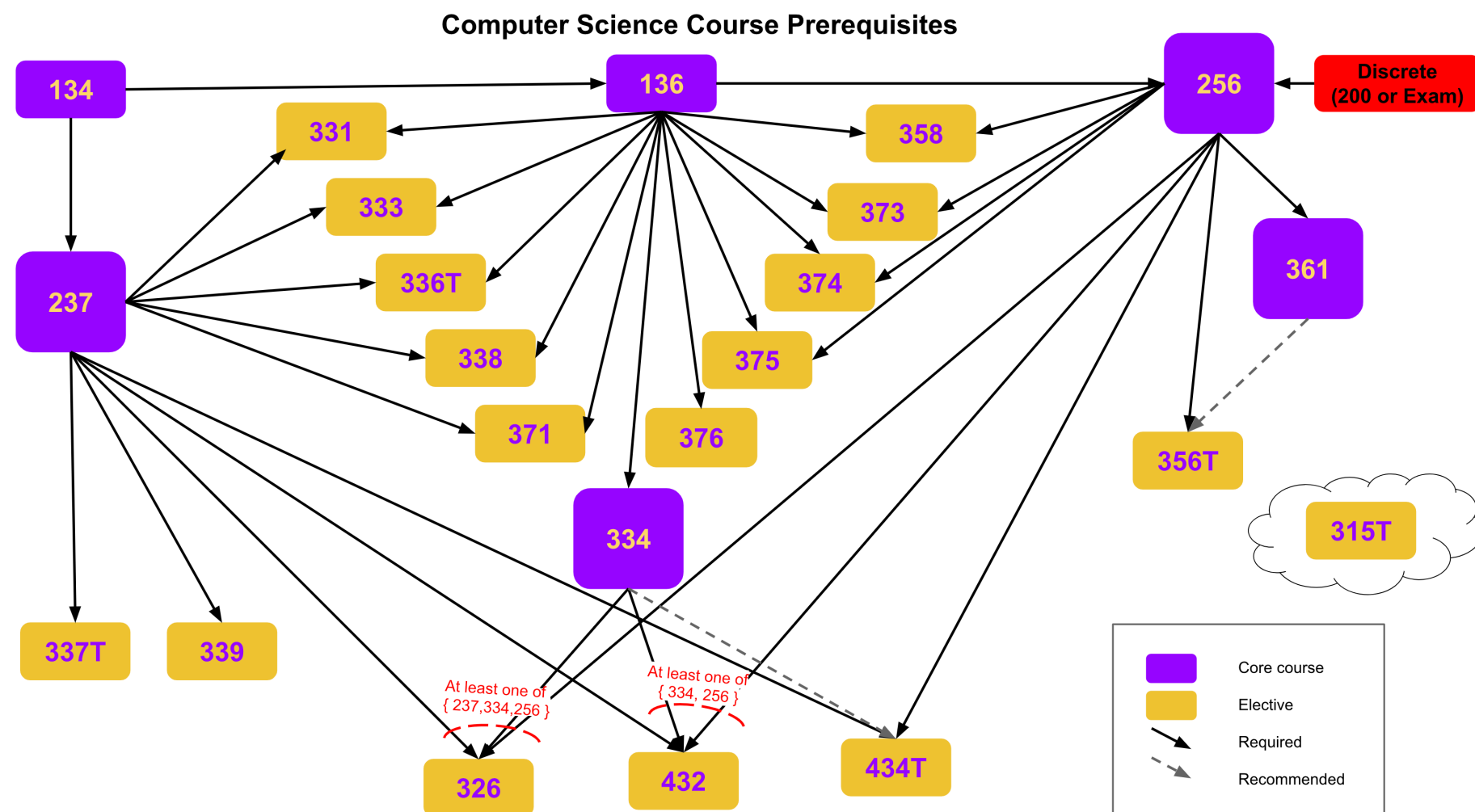
Definition. A directed graph is acyclic (or a DAG) if it contains no (directed) cycles.

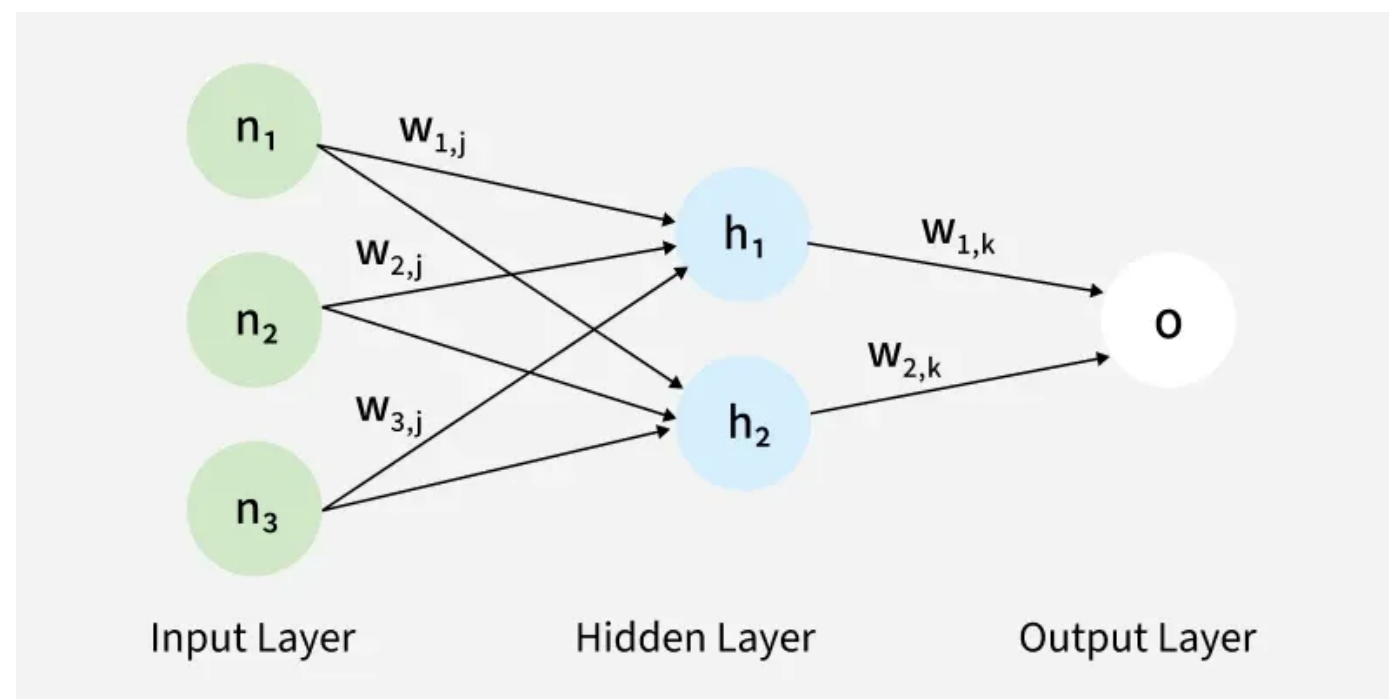
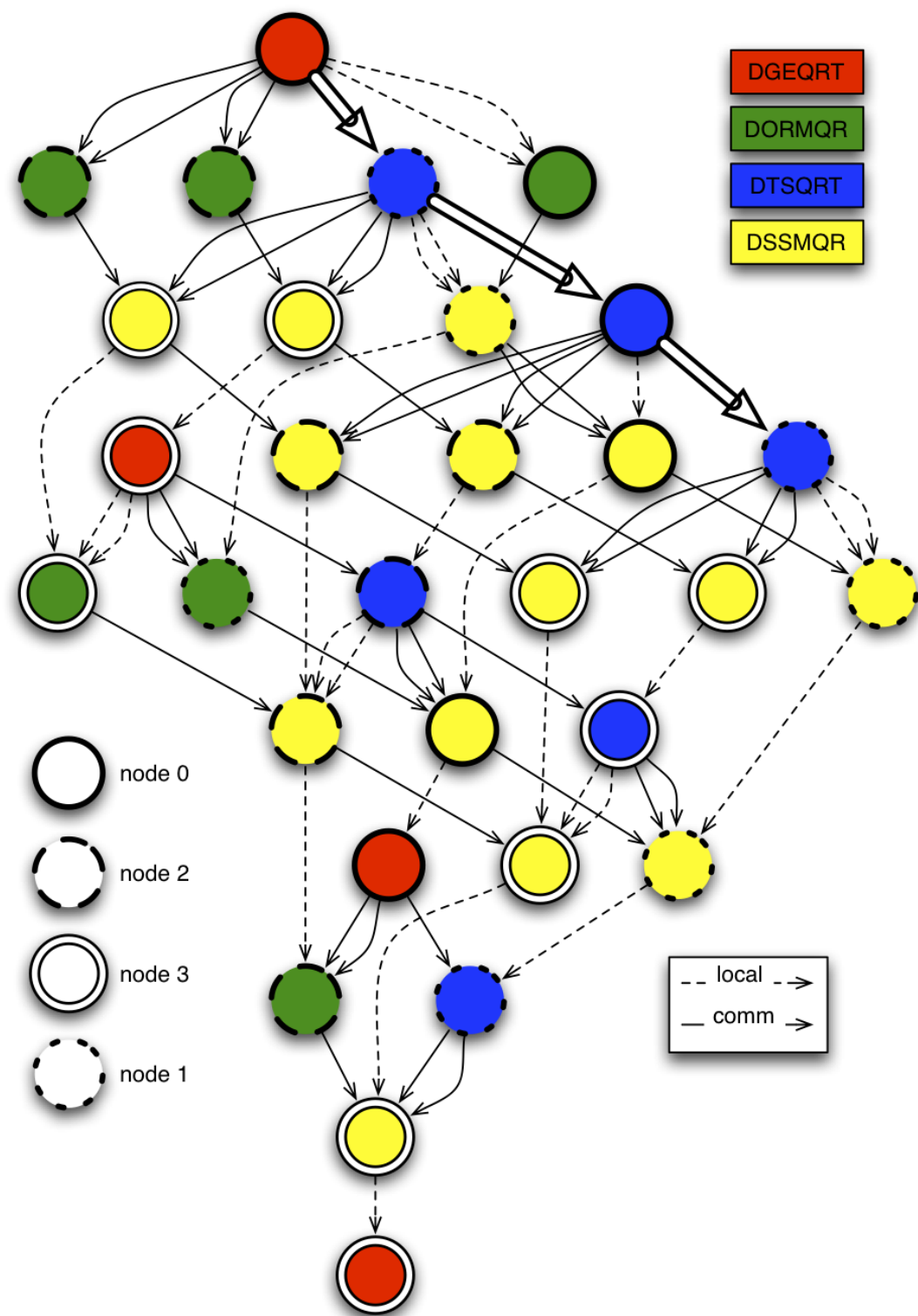
- Very useful to model dependency graphs
- Unlike undirected graphs, being acyclic does not mean a tree-like structure: DAGs can have lots of edges without any cycles



Directed Acyclic Graphs (DAGs)

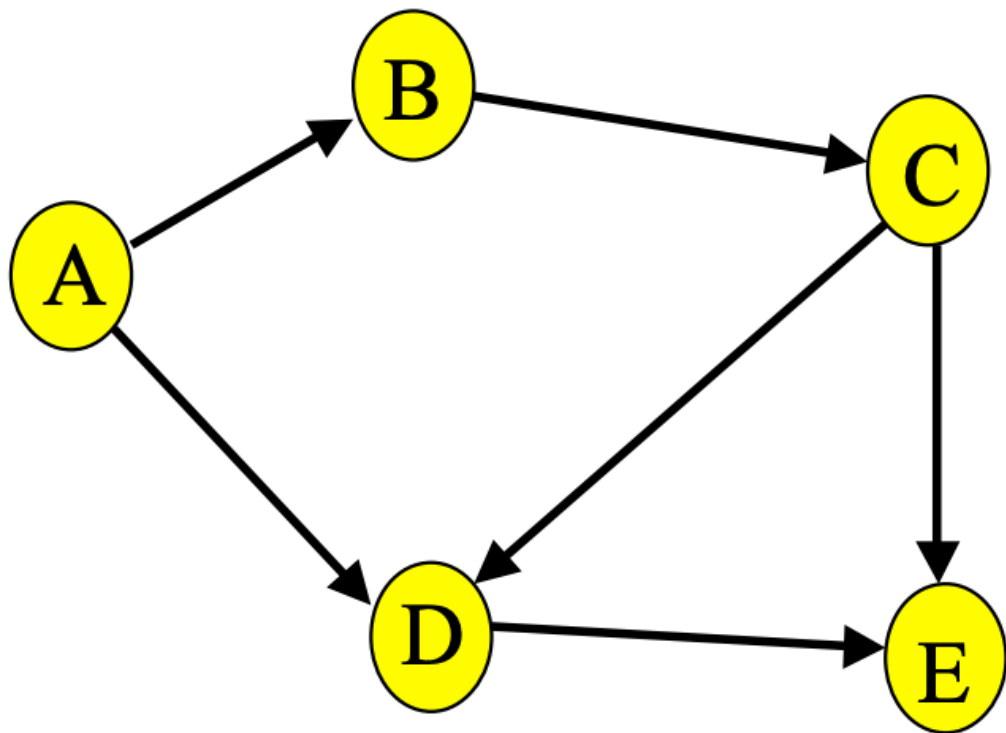
- Many applications need a way to "organize" the nodes of a DAG that preserves dependency structure



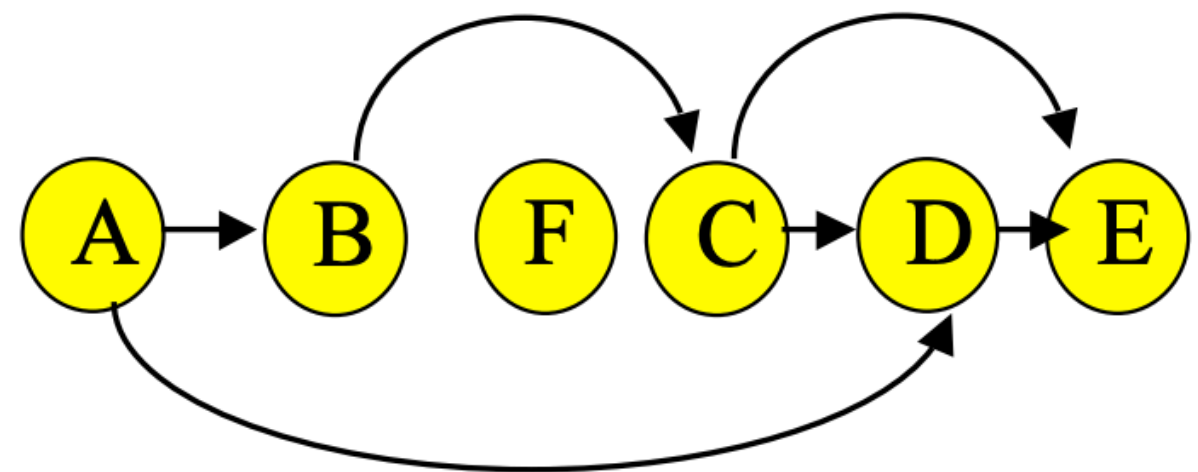


Topological Ordering

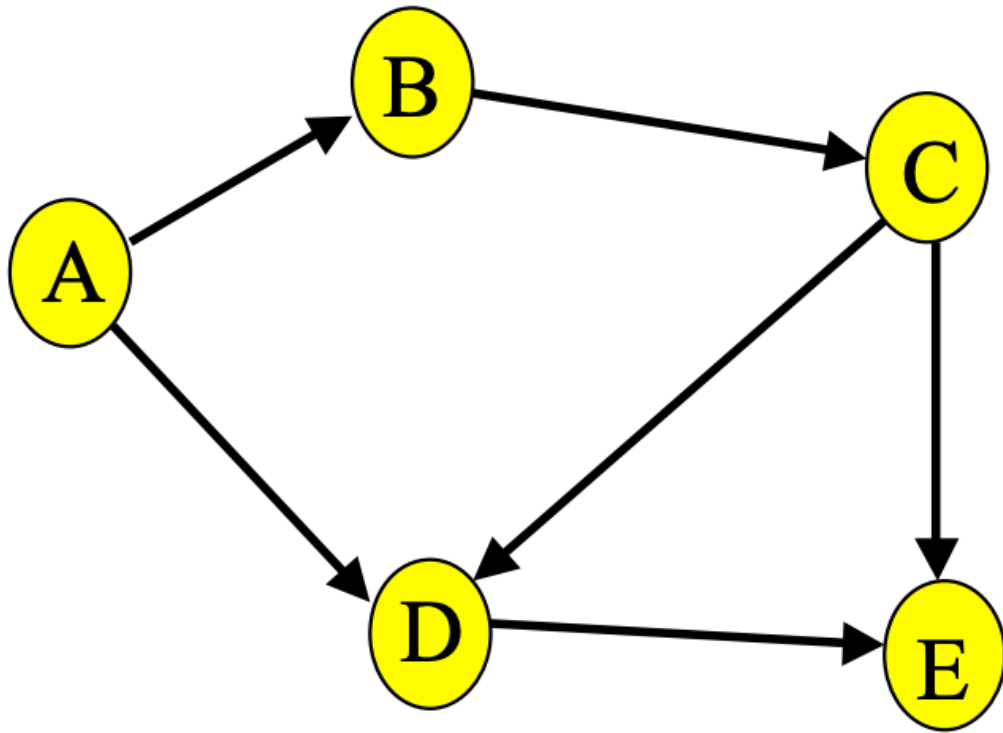
Definition. Given a DAG $G = (V, E)$ a topological ordering is an ordering of all the vertices such that for any edge $(v, w) \in E$, v appears before w in the ordering.



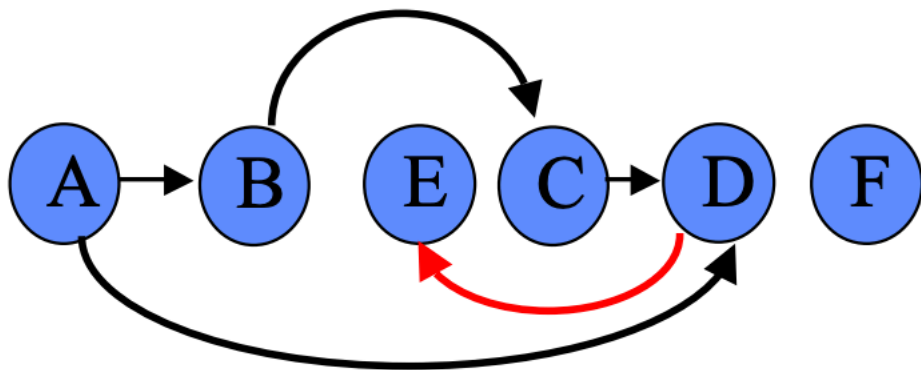
Any linear ordering in which all the arrows go to the right is a valid solution



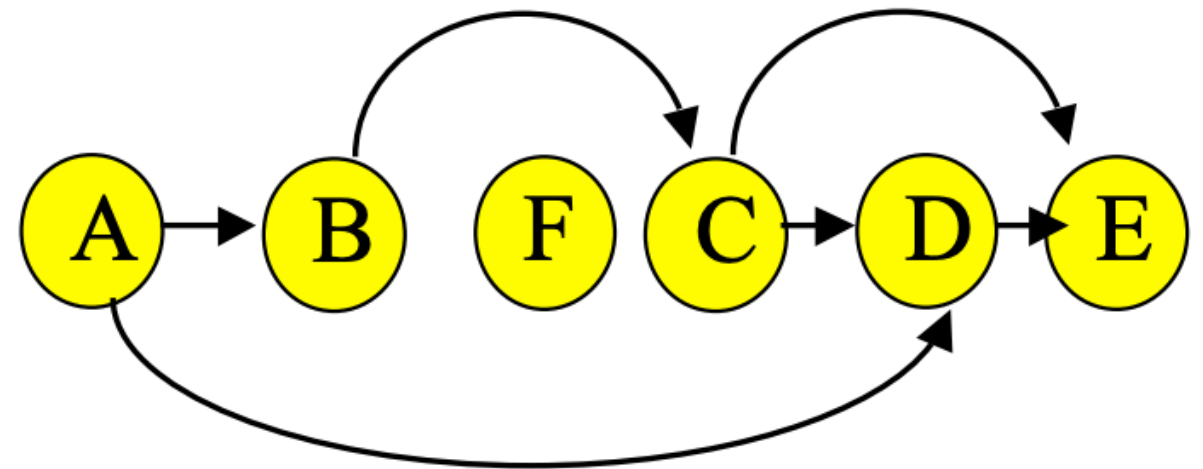
Topological Ordering: Example



Not a valid topological sort!



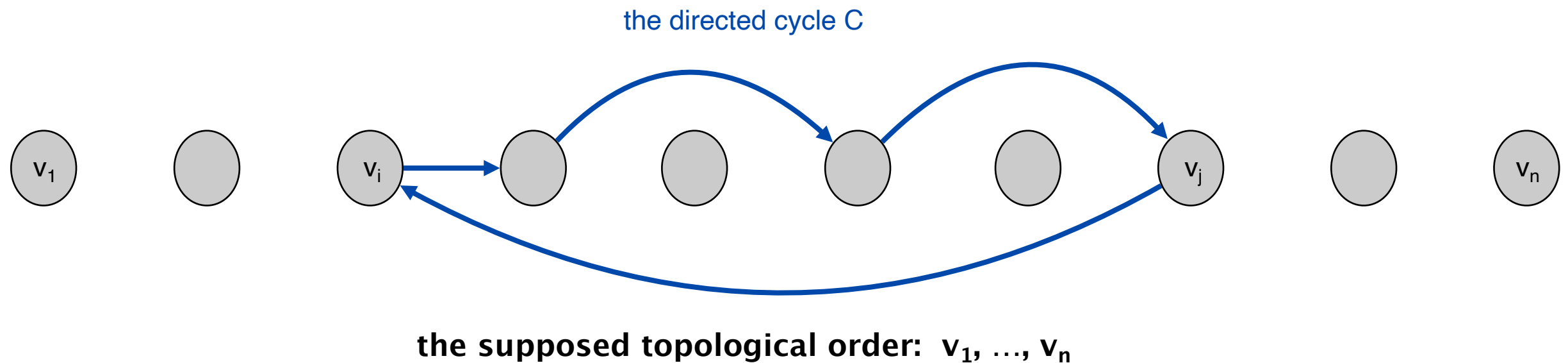
Any linear ordering in which all the arrows go to the right is a valid solution



Topological Ordering and DAGs

Lemma. If G has a topological ordering, then G is a DAG.

Proof. [By contradiction] Suppose G has a cycle C . Let $v_1, v_2, \dots, v_n$ be the topological ordering of G

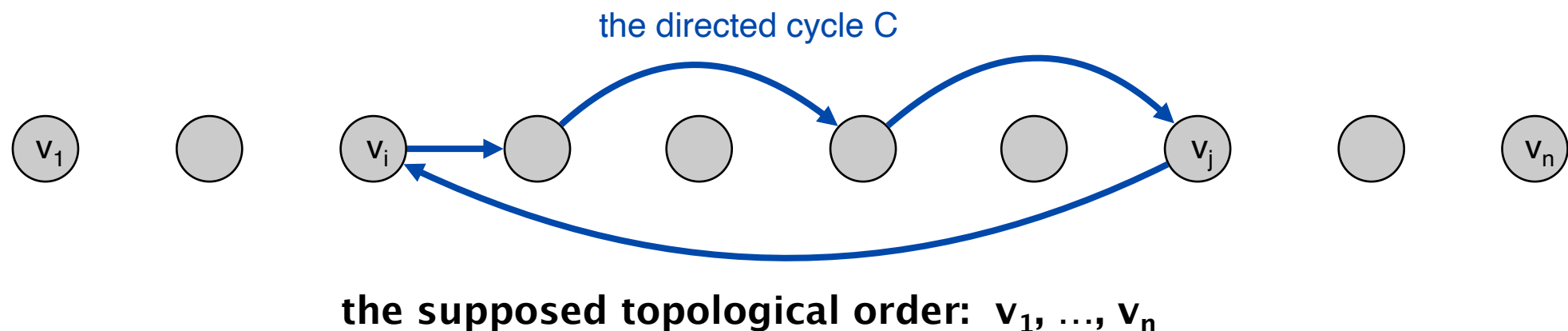


Topological Ordering and DAGs

Lemma. If G has a topological ordering, then G is a DAG.

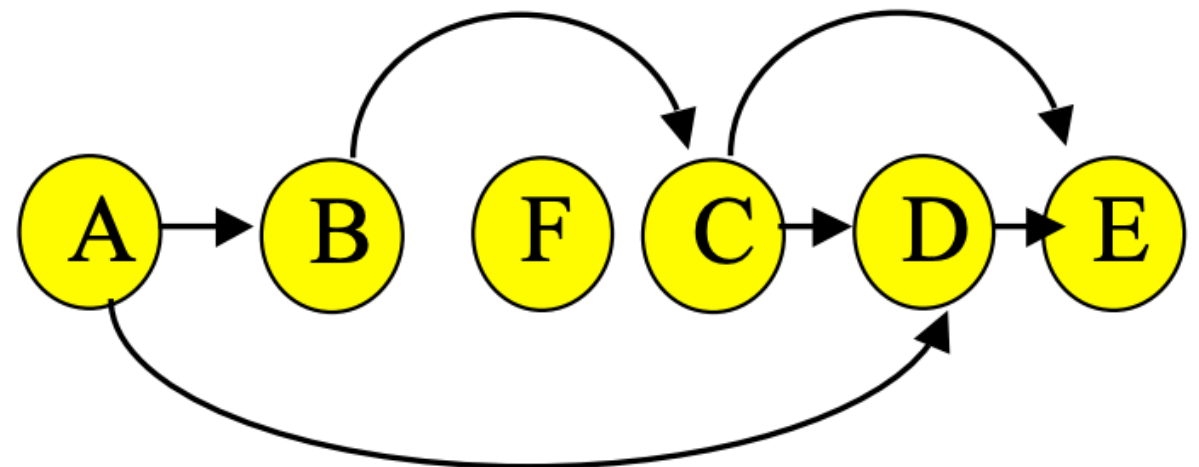
Proof. [By contradiction] Suppose G has a cycle C . Let $v_1, v_2, \dots, v_n$ be the topological ordering of G

- Let v_i be the lowest-indexed node in C , and let v_j be the node just before v_i ; thus (v_j, v_i) is an edge
- By our choice of i , we have $i < j$.
- But (v_j, v_i) is a back edge w.r.t. ordering $v_1, v_2, \dots, v_n$ ($\Rightarrow \Leftarrow$) ■



Topological Ordering and DAGs

- No directed **cyclic** graph can have a topological ordering
- Does every DAG have a topological ordering?
- To show this, let's try ordering the vertices
- Which vertex should go first?



Finding a Topological Ordering

Claim. Every DAG has a vertex with in-degree zero.

Proof. [By contradiction] Suppose every vertex has an incoming edge.

- Pick any vertex v , there must be an edge (u, v) .
- Walk backwards following these incoming edges for each vertex
- After $n + 1$ steps, we must have visited some vertex w twice (why?)
- Nodes between two successive visits to w form a cycle ($\Rightarrow \Leftarrow$) ■

Idea for finding topological ordering. Build order by repeatedly removing a vertex of in-degree 0 from G .

Topological Sorting Algorithm

TopologicalSorting(G) $\triangleleft G = (V, E)$ is a DAG

Initialize $T[1..n] \leftarrow \emptyset$ and $i \leftarrow 0$

while V is not empty do

$i \leftarrow i + 1$

 Find a vertex $v \in V$ with $\text{indeg}(v) = 0$

$T[i] \leftarrow v$

 Delete v (and its edges) from G

Analysis:

- Correctness, any ideas how to proceed?
- Running time

Topological Sorting Algorithm

Analysis (Correctness). Base case: $n = 1$, no edges, the vertex itself forms topological ordering

- IH: Suppose algorithm is correct for any DAG with less than n vertices
- Consider an arbitrary DAG on n vertices
 - Must contain a vertex v with in-degree 0 (we proved it)
 - Delete v and its outgoing edges to create G'
 - G' is a DAG with less than n vertices
 - By IH, G' has a topological ordering (say $u_1, u_2, \dots, u_{n-1}$)
- Then $v, u_1, u_2, \dots, u_{n-1}$ is a valid topological ordering of G ■

Topological Sorting Algorithm

Running time:

- (Initialize) In-degree array **ID**[1..*n*] of all vertices
 - $O(n + m)$ time
- Find a vertex with in-degree zero
 - $O(n)$ time
 - Need to keep doing this till we run out of vertices! $O(n^2)$
- Reduce in-degree of vertices adjacent to a vertex
 - $O(\text{outdegree}(v))$ time for each *v*: $O(n + m)$ time
- **Bottleneck step:** finding vertices with in-degree zero

Can we do better?

Linear-Time Algorithm

- Need a faster way to find vertices with in-degree 0 instead of searching through entire in-degree array!
- **Idea:** Maintain a queue (or stack) S of in-degree 0 vertices
- Update S : When v is deleted, decrement $ID[u]$ for each neighbor u ; if $ID[u] = 0$, add u to S :
 - $O(\text{outdegree}(v))$ time
- Total time for previous step over all vertices:
 - $\sum_{v \in V} O(\text{outdegree}(v)) = O(n + m)$ time
- Topological sorting takes $O(n + m)$ time and space!

