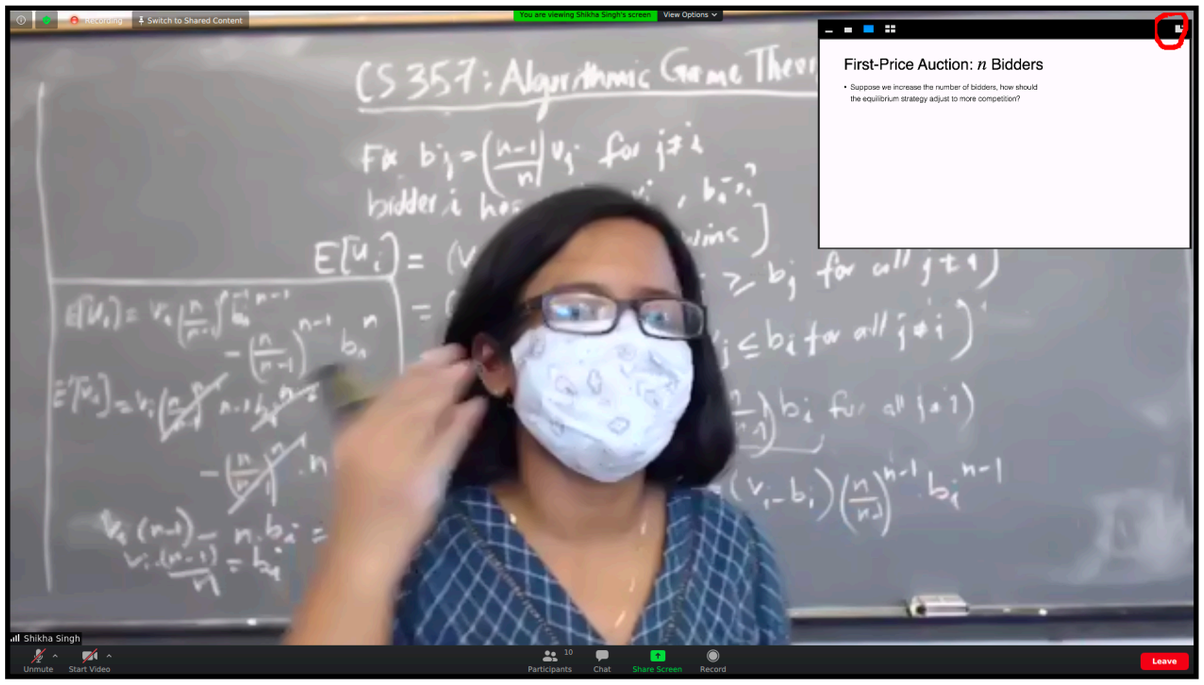
Directed Graphs

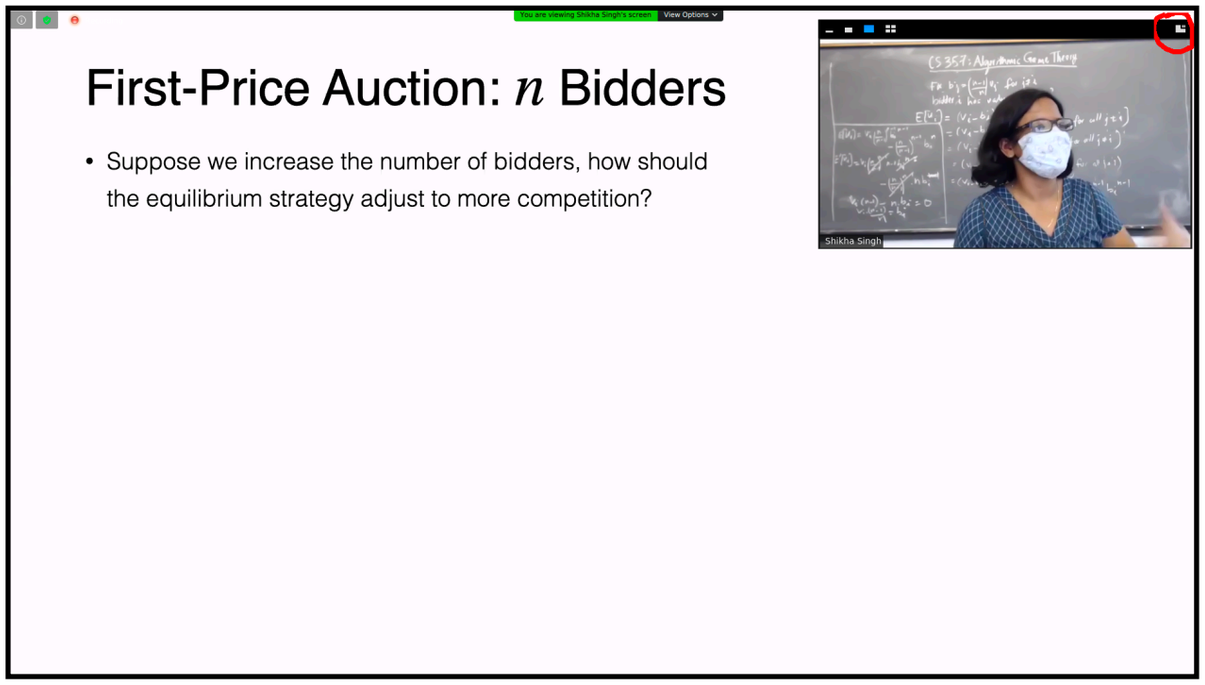
Reminders

- Homework 1 is due tonight by 11 pm
    - Question 1: Must identify $\Theta()$ or $O()$ relationship between every adjacent pair in ordering
    - Can use any results proved in class
- Homework 2 will be released this afternoon
- Help hours today:
    - Me: 1.30-3 pm
    - TAs: 3-5 pm, 7-11 pm
- If you cannot make any office/TA hours, let me know

Slides/Board on Zoom

- Reminder: **switch** between **speaker view** and **screen view in Zoom**
- Can also switch in lecture recordings on GLOW
- From now on, only one section will be recorded



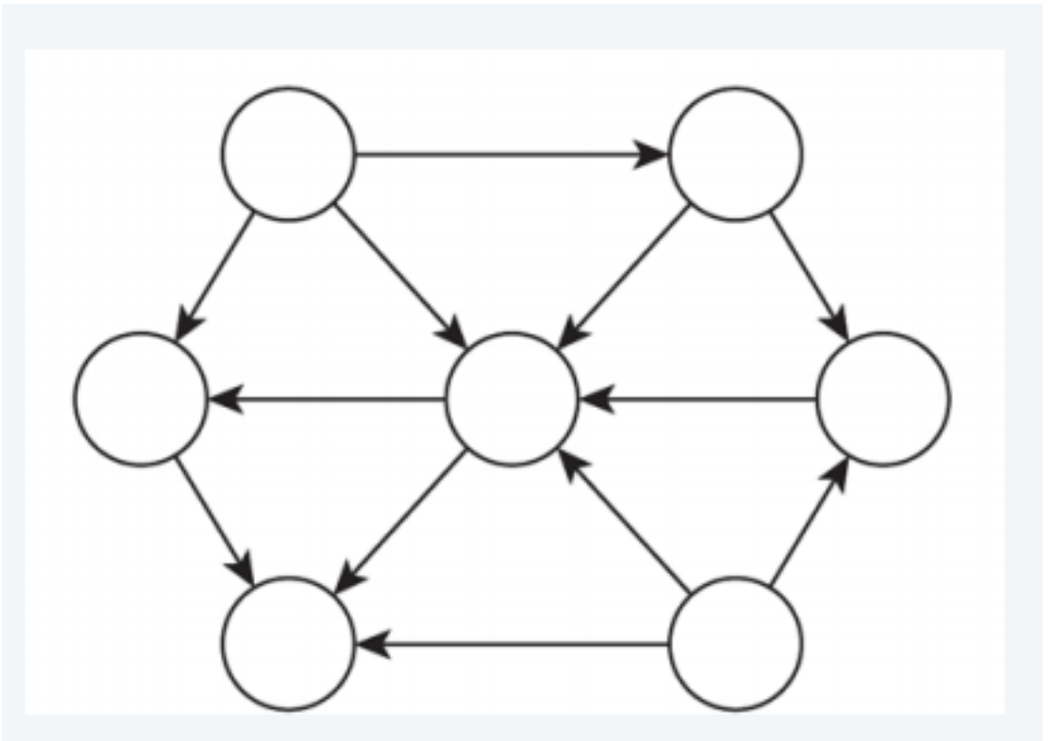


Directed Graphs

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

- Edges have “orientation”
- Edge (u, v) or sometimes denoted $u \to v$, 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.



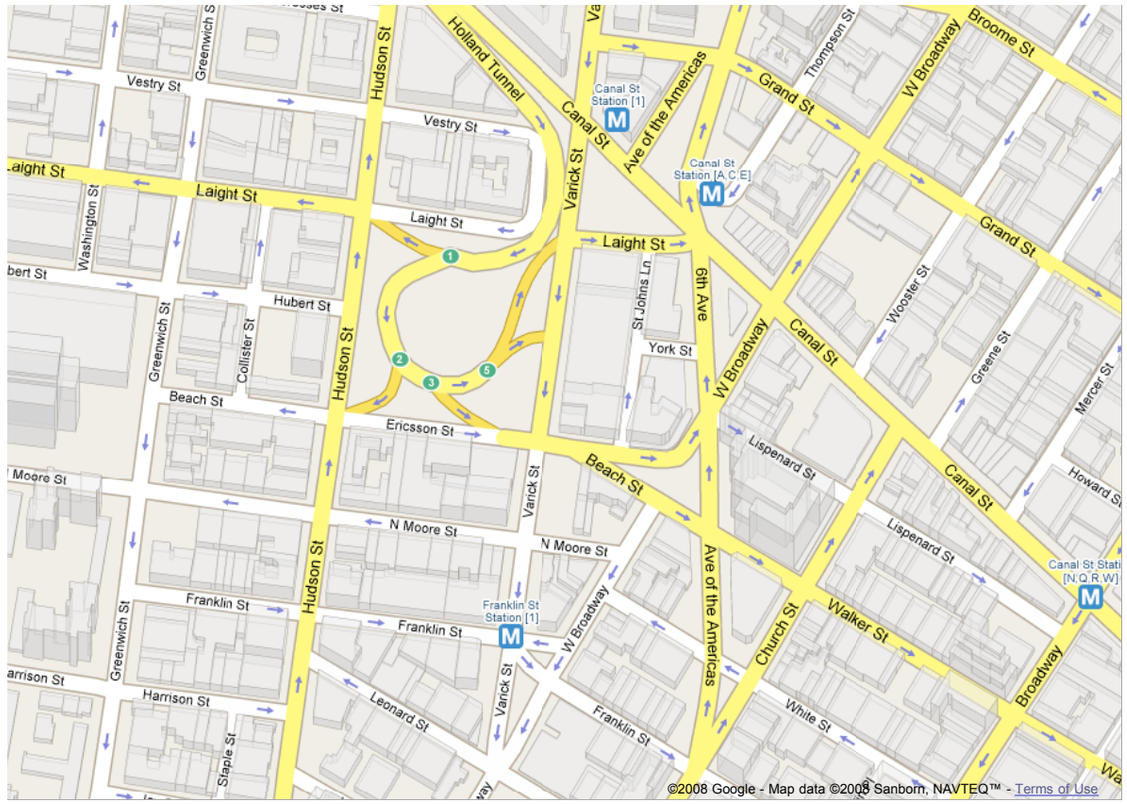
Directed Graphs in Practice

Web graph:

- Webpages are nodes, hyperlinks are edges
- Orientation of edges is crucial
- Search engines use hyperlink structure to rank web pages

Road network

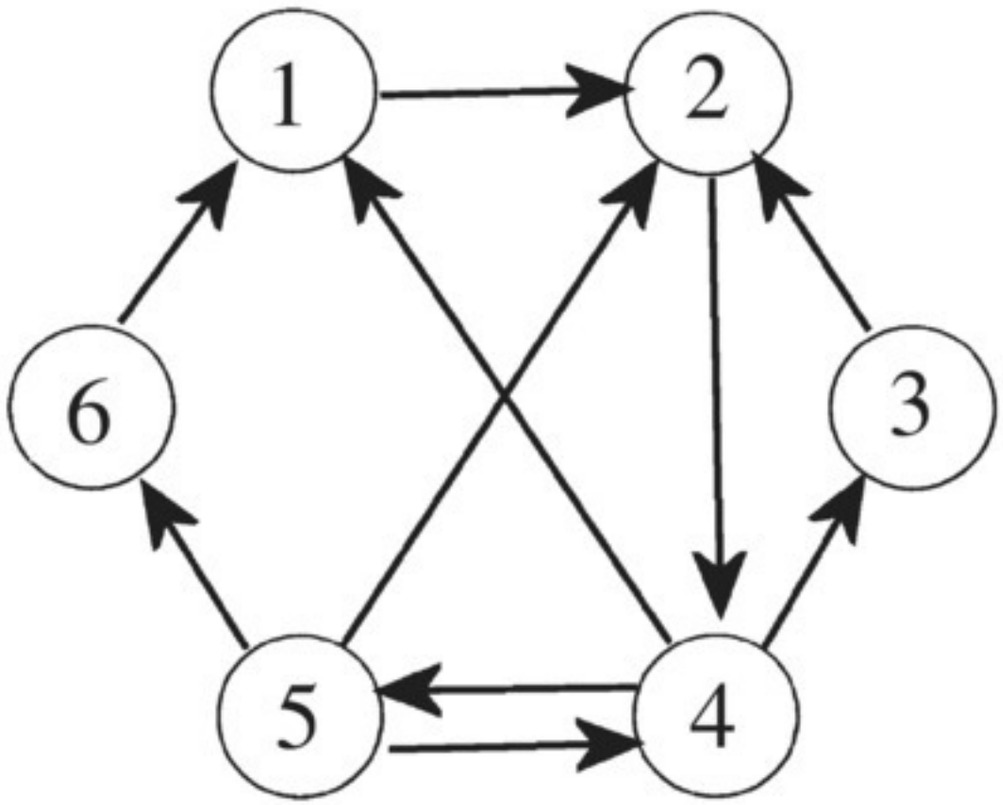
- Road: nodes
- Edge: one-way street



Directed Reachability

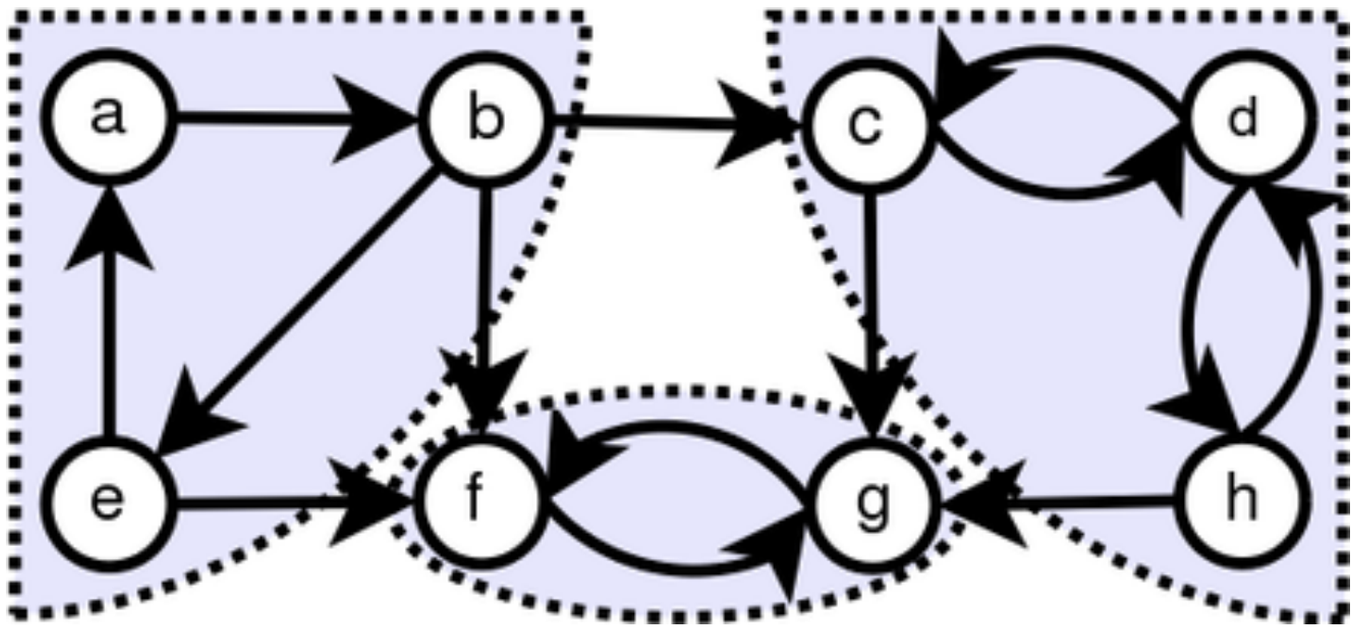
Directed reachability. Given a node s find all nodes reachable from s.

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



Strong Connectivity

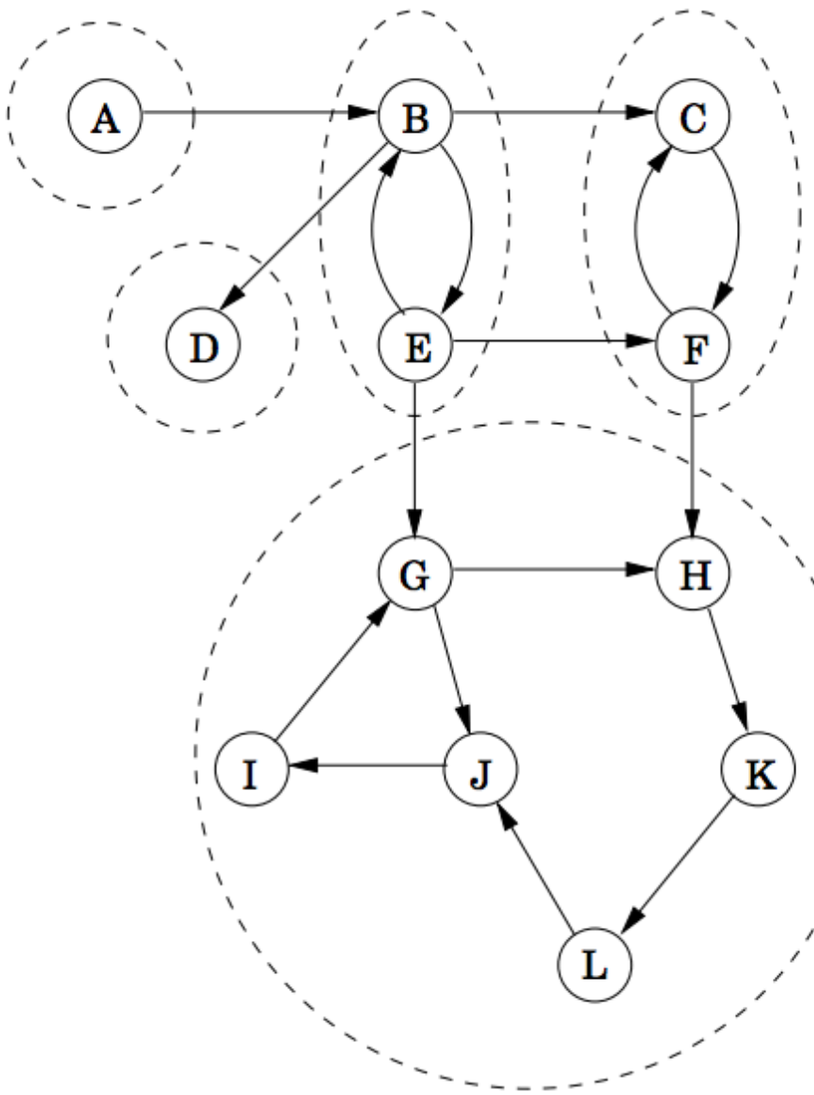
- **Strong connectivity.** Connected components in directed graphs defined based on mutual reachability. Two vertices u, v in a directed graph G are mutually reachable if there is a directed path from u to v and from from v to u. A graph G is **strongly connected** if every pair of vertices are mutually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

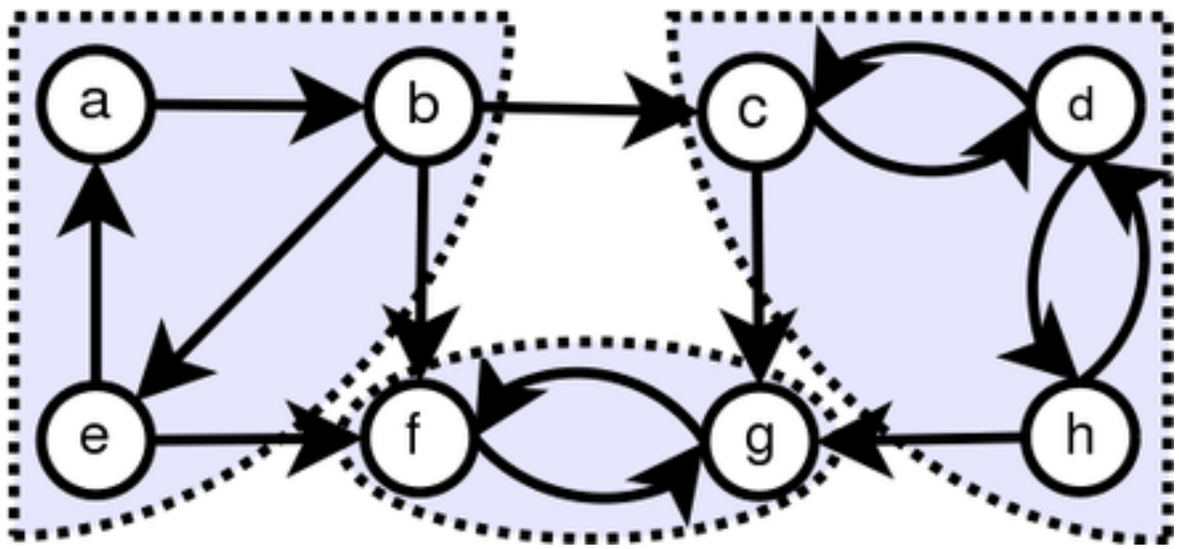


Wikipedia: Each shaded region is strongly connected

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.

Deciding Strongly Connected

First idea. How can we use BFS/DFS to determine strong connectivity? Recall: BFS/DFS on graph G starting at v will identifies all vertices reachable from v by directed paths

- Pick a vertex v. Check to see whether every other vertex is reachable from v;
- Now see whether v is reachable from every other vertex

Analysis

- 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?*

Testing Strong Connectivity

Idea. Flip the edges of G and do a BFS on the new graph

- Build $G_{\text{rev}} = (V, E_{\text{rev}})$ where $(u, v) \in E_{\text{rev}}$ iff $(v, u) \in E$
- 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 (Performance)

- 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

Testing Strong Connectivity

Idea. Flip the edges of G and do a BFS on the new graph

- Build $G_{\text{rev}} = (V, E_{\text{rev}})$ where $(u, v) \in E_{\text{rev}}$ iff $(v, u) \in E$
- 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).

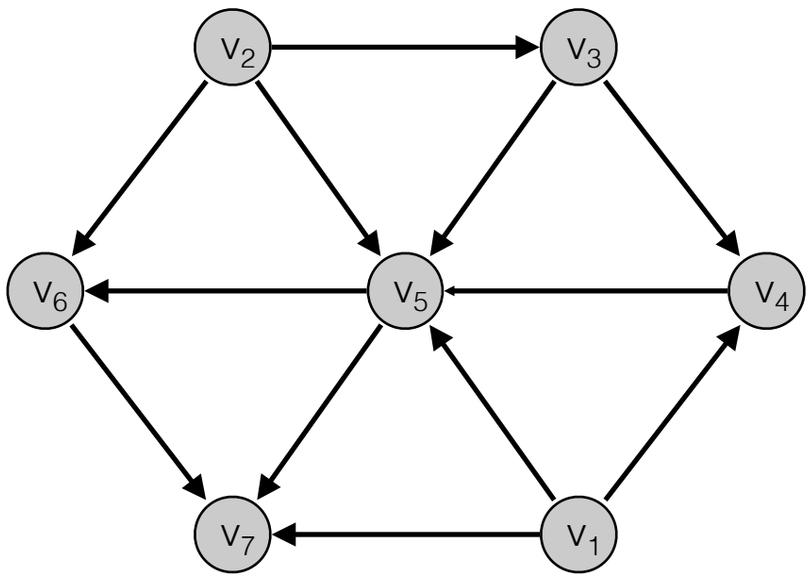
Analysis (Correctness)

- **Claim.** If v is reachable from every node in G and every node in G is reachable from v then G must be strongly connected
- **Proof.** For any two nodes $x, y \in V$, they are mutually reachable through v, that is, $x \rightsquigarrow v \rightsquigarrow y$ and $y \rightsquigarrow v \rightsquigarrow z$ ∎

Directed Acyclic Graphs (DAGs)

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

Question. Given a directed graph G, can you detect if it has a cycle in linear time?

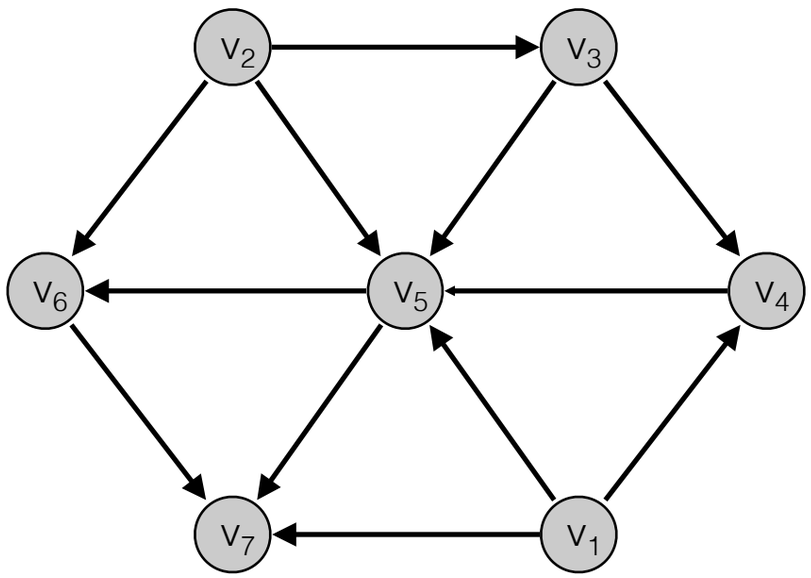


a DAG

Directed Acyclic Graphs (DAGs)

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

Question. What goes wrong if we use our approach for cycle detection in undirected graphs?



a DAG

Cycle Detection: Directed

Idea. Need to keep track of not only "marked/visited" vertices but also "finished" ones

```
Cycle-Detection-Directed-DFS(u):
    Set status of u to marked # discovered u
for each edges (u, v):
    if v's status is unmarked:
        DFS(v)
    else if v is marked but not finished
        report a cycle!
mark u finished
# done exploring neighbors of u
```

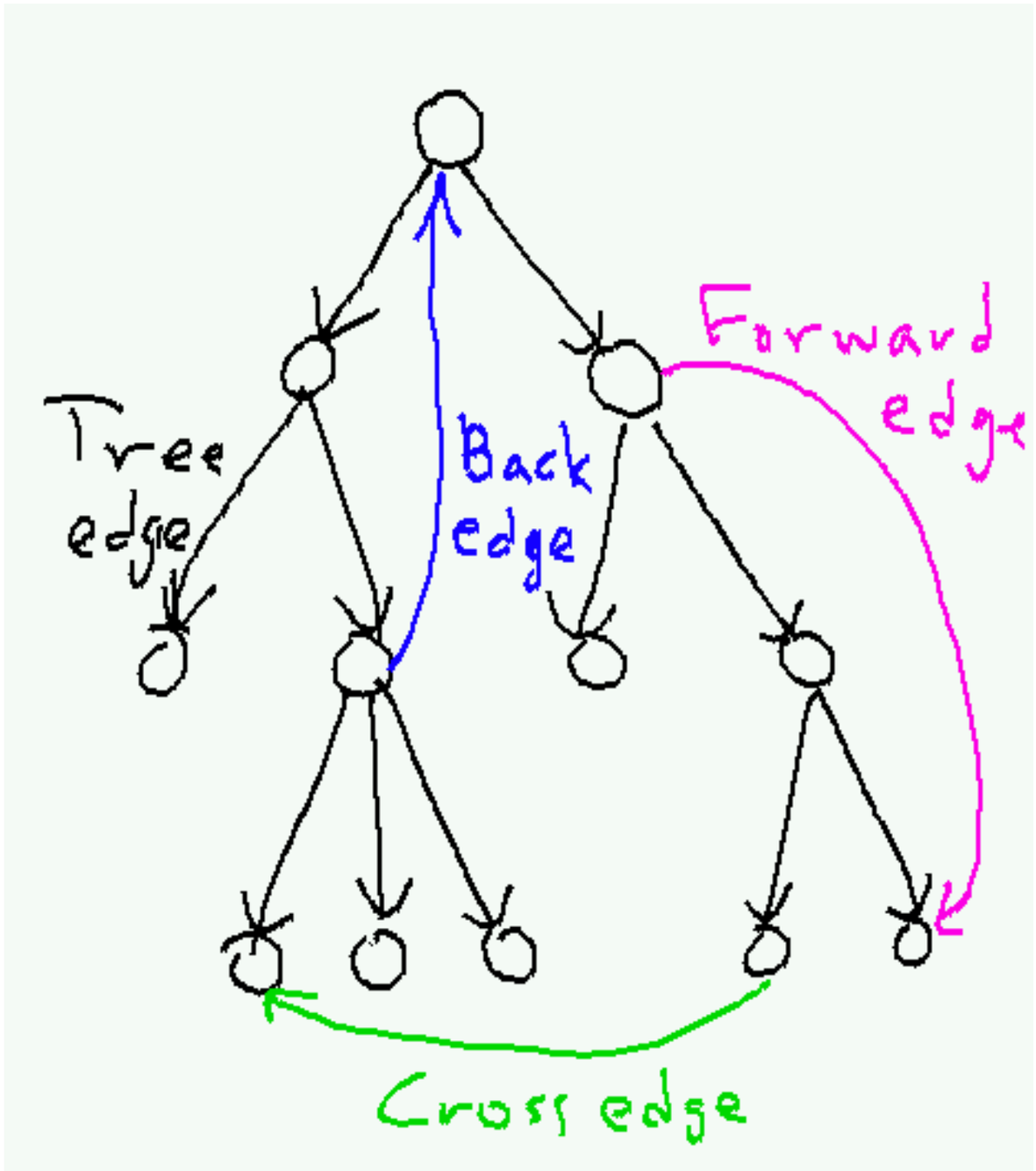
Classifying Edges: DFS Directed

- Call a node u **inactive,** if DFS(u) has not been called yet
- Call a node u **active**, if DFS(u) has been called but has not returned
- Call a node u **finished**, if DFS(u) has returned
- We can keep track of when a node is activated and finished and use it to classify every edge $u \rightarrow v$ in the directed input graph G

Classifying Edges: DFS Directed

- **Tree edge** (u, v)**:** the edge is in the DFS tree
    - (If v is activated just after u and finished before u)
- The remaining edges fall into three categories:
- **Forward edge:** (u, v) where v is a proper descendant of u in tree
    - (If v is activated after u and finished before u)
- **Back edge**: (u, v) where v is an ancestor of u in tree
    - v is active when DFS(u) begins
- **Cross edge:** (u, v) where u and v are not related in tree (are not ancestors or descendants of one another)
    - v is finished when DFS(u) begins

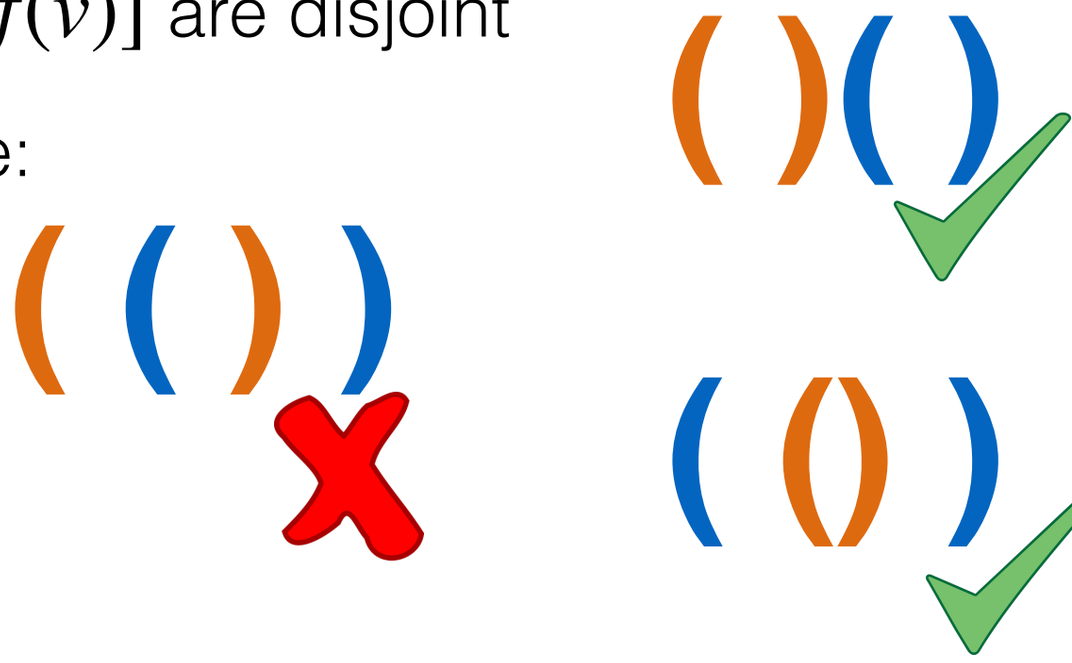
Classifying Edges: DFS Directed



http://www.cs.yale.edu/homes/aspnes/pinewiki/DepthFirstSearch.html

Parenthesis Structure: DFS Directed

- Let $d[u]$ denote the time when u is discovered, $f[u]$ denote the time when u is finished
- **Lemma.** Given a connected directed graph $G = (V, E)$ and any DFS tree for G and vertices $u, v \in V$
  - v is a descendant of u in the DFS tree if and only if $d[u] < d[v] < f[v] < f[u]$
- u, v are unrelated (no ancestor, descendant relation in the tree) if and only if $[d(u), f(u)]$ and $[d(v), f(v)]$ are disjoint
- Both of the following are not possible:
  - $d[u] < d[v] < f[u] < f[v]$
  - $d[v] < d[u] < f[v] < f[u]$



Parenthesis Structure: DFS Directed

- **Claim.** Given a directed graph G, it is acyclic if and only if any DFS tree of G has no back edges.
- **Proof** ($\Leftarrow$) : Suppose there are no back edges
- Then all edges in G go from a vertex of higher finish time to a vertex of lower finish time (parenthesis structure)
- Hence there can be no cycles (need a way to go back to an active node from an active node)
- ($\Rightarrow$) Assume G has no cycles, and suppose a DFS tree has a back edge $v \rightarrow u$, what is the contradiction?
- There is a path $u \rightsquigarrow v$ following tree edges and there is a edge from v to u (this is our cycle!). ∎

Directed Acyclic Graphs (DAGs)

Analysis. (Correctness) A directed (undirected) graph has a cycle iff we can identify a back edge during a DFS traversal.

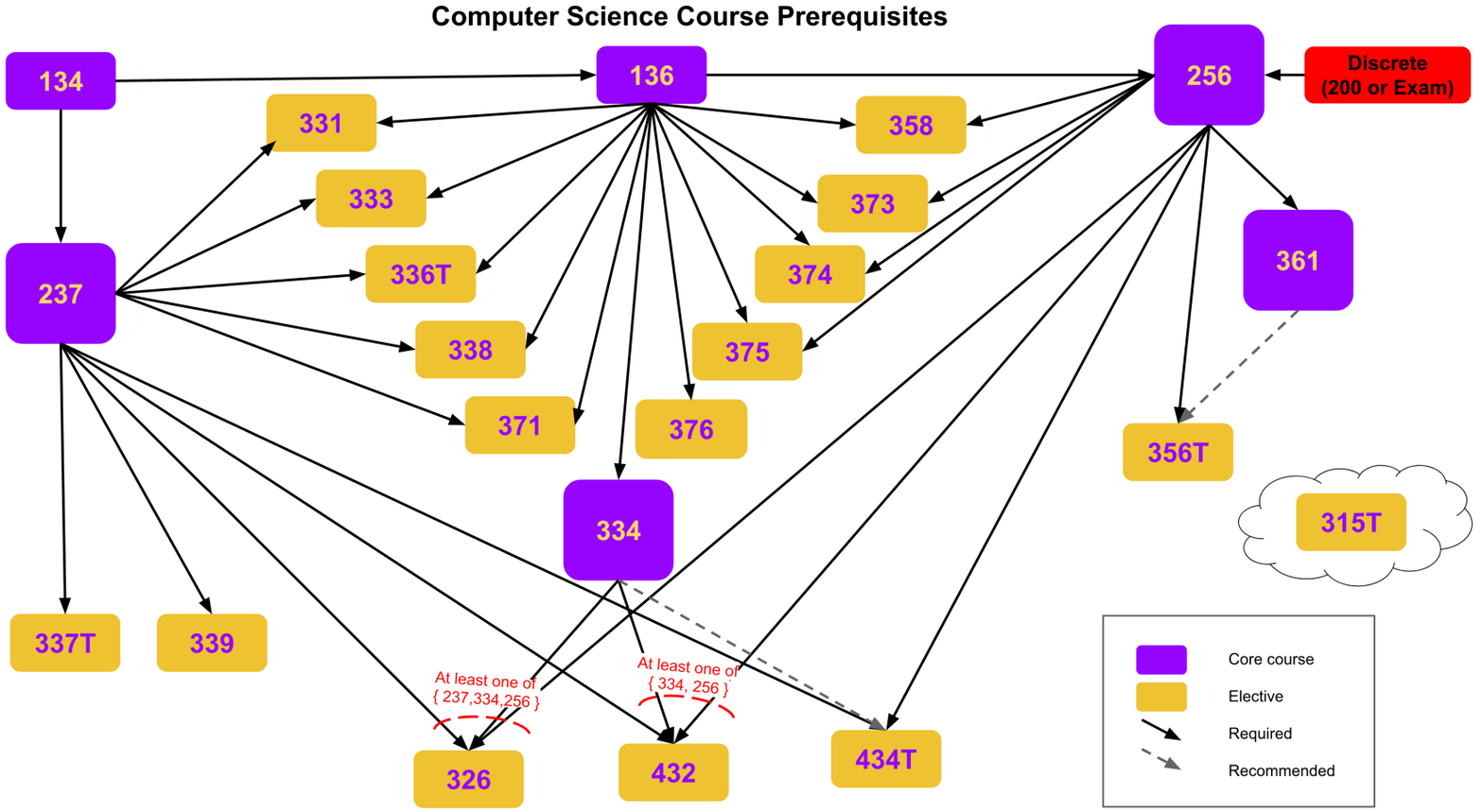
The following code finds and reports back edges correctly.

```
Cycle-Detection-Directed-DFS(u):
   Set status of u to marked # discovered u
   for each edges (u, v):
      if v's status is unmarked:
         DFS(v)
      else if v is marked but not finished # active
         report a cycle!
   mark u finished
   # done exploring neighbors of u
```

Topological Ordering

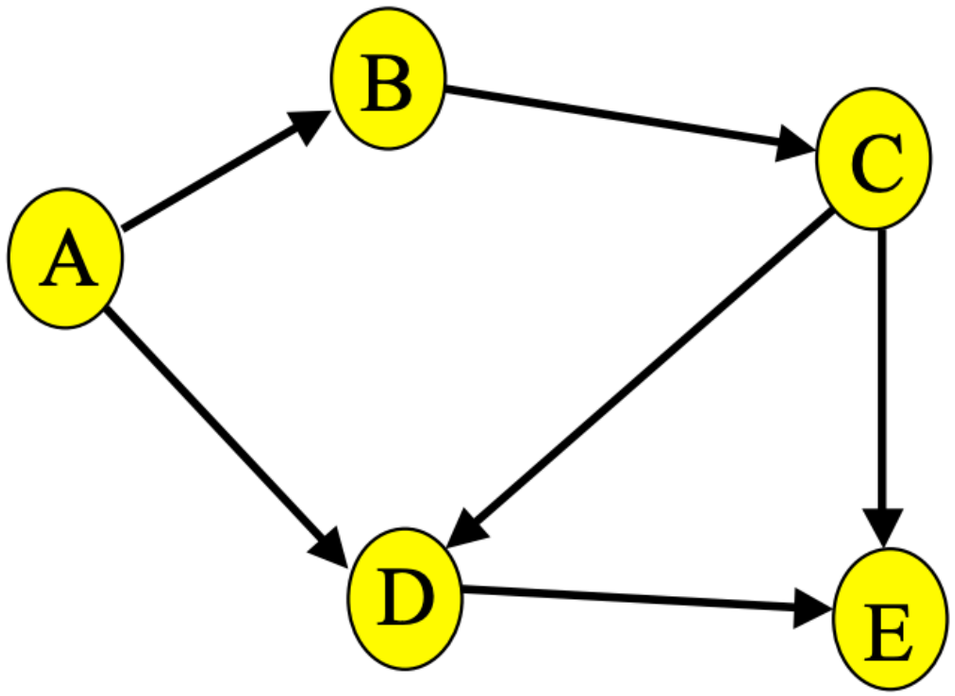
Problem. Given a DAG $G = (V, E)$ find a linear ordering of the vertices such that for any edge $(v, w) \in E$, v appears before w in the ordering.

Example. Find an ordering in which courses can be taken that satisfies prerequisites.

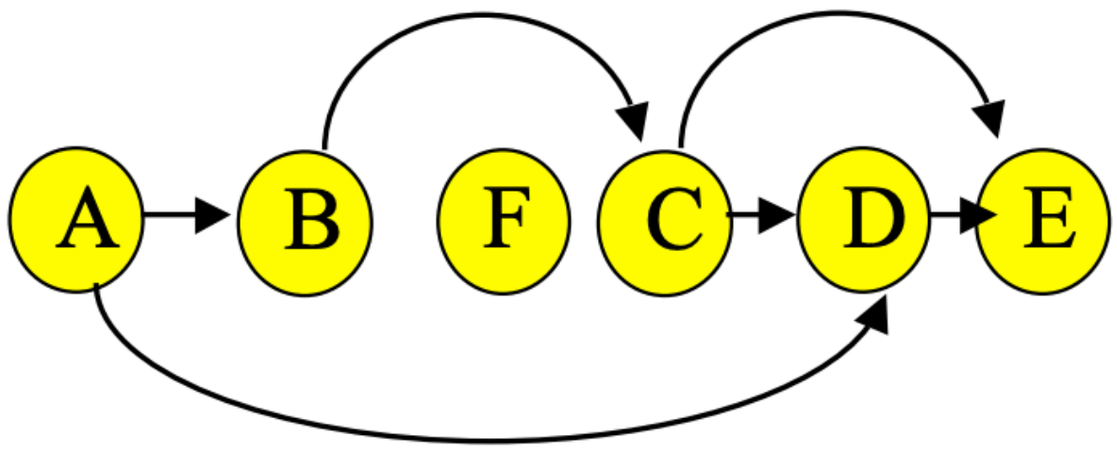


Credits: Bill Jannen, http://www.cs.williams.edu/~jannen/teaching/cs-prereqs.svg

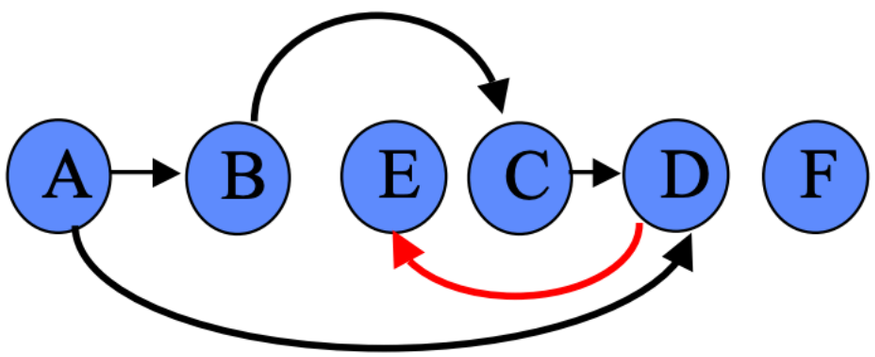
Topological Ordering: Example



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



Not a valid topological sort!



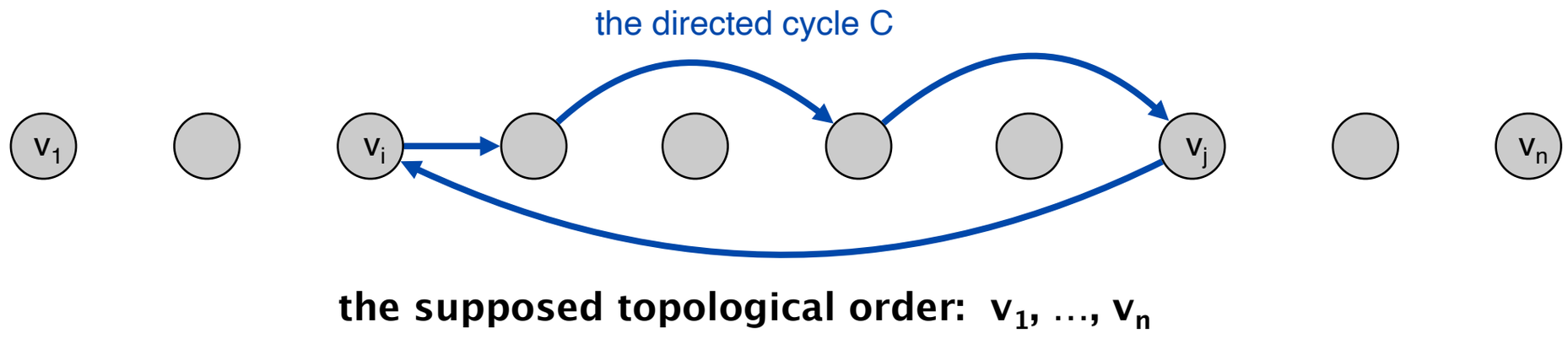
Taken from: https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Ordering and DAGs

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

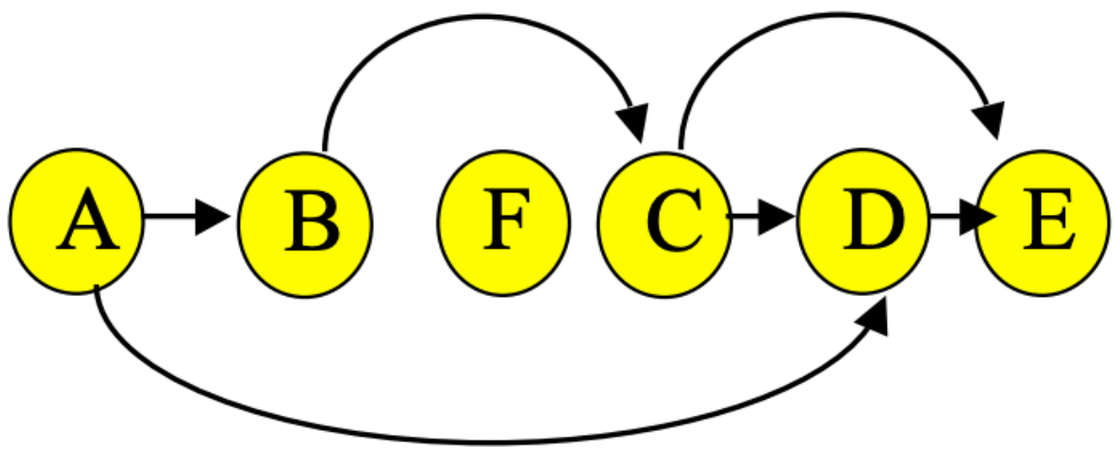
Proof. [By contradiction] Suppose G has a cycle C. Let $v_1, v_2, \ldots, v_n$ be the topological ordering of G

- 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 $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, v_2, \ldots, v_n$ is a topological order, we must have $j < i$ $(\Rightarrow\Leftarrow)$ ∎



Topological Ordering and DAGs

- No directed **cyclic** graph can have a topological ordering
- Does every DAG have a topological ordering?
    - Yes, can prove by induction (and construction)
- How do we compute a topological ordering?
    - What property should the first node in any topological ordering satisfy?
        - Cannot have incoming edges, i.e., indegree = 0
    - Can we use this idea repeatedly?



Finding a Topological Ordering

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

Proof. [By contradiction] Suppose every vertex has an incoming edge. Show that the graph must have a cycle.

- 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) ◃ G = (V,E) is a DAG
    Initialize T[1..n]← 0 and i ← 0
    while V is not empty do
        i←i+1
        Find a vertex v ∈ V with indeg(v) = 0
        T[i] ← v
        Delete v (and its edges) from G
```

Analysis:

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

Topological Sorting Algorithm

Analysis (Correctness). Proof by induction on number of vertices n:

- $n = 1$, no edges, the vertex itself forms topological ordering
- Suppose our algorithm is correct for any graph with less than n vertices
- Consider an arbitrary DAG on n vertices
  - Must contain a vertex v with in-degree 0 (we proved it)
  - Deleting that vertex and all outgoing edges gives us a graph G' with less than n vertices that is still a DAG
  - Can invoke induction hypothesis on G' !
- Let $u_1, u_2, \ldots, u_{n-1}$ be a topological ordering of G', then $v, u_1, u_2, \ldots, u_{n-1}$ must be a 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(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!

Topological Ordering by DFS

- Call DFS and maintain finish times of all vertices
  - Finish(u): time DFS(v) completed for all neighbors of u
- Return the list of vertices **in reverse order of finish times**
    - Vertex finished last will be first in topological ordering
- **New.** This generates the topological ordering all all nodes reachable from the root of the DFS

- **Claim.** If a DAG G contains an edge $u \rightarrow v$, then the finish time of u must be larger than the finish time of v.
    - u is finished only after all its neighbors are finished

Traversals: Many More Applications

BFS and/or DFS can also be used to solve many other problems

- Find a (directed) cycle in a (directed) graph (or a cycle containing a specified vertex v)
- Find all cut vertices of a graph (A cut vertex is one whose removal increases the number of connected components)
- Find all bridges of a graph (A bridge is an edge whose removal increases the number of connected components
- Find all biconnected components of a graph (A biconnected component is a maximal subgraph having no cut vertices)

All of this can be done in $O(|V| + |E|)$ space and time!