Directed Graphs

Announcements

- Homework 2 is due Wednesday at 10pm
    - TAs have solutions to in-class activities
    - We'll go over some of them today as well
- Help hours today: course homepage calendar

- Student announcements?

Quick Review: Trees

Recall (K&T 3.2, page 78)**:** Let $G = (V, E)$ be an undirected graph on n nodes. Any two of the following statements implies the third:

1. G is connected.
2. G does not contain a cycle (equivalently, G is *acyclic*).
3. G has $n - 1$ edges.

Note, this is a stronger version of the claim (K&T 3.1) that every n-node tree has exactly $n - 1$ edges.

Quick Review: Trees

Recall: Let $G = (V, E)$ be an undirected graph on n nodes. Any two of the following statements implies the third (3.2 from K&T, page 78):

1. G is connected.
2. G does not contain a cycle (equivalently, G is *acyclic*).
3. G has $n - 1$ edges.

Prove $(1), (2) \implies (3)$

The proof is by induction on the number of nodes, n.

Let $P(n)$ denote the statement, "Any graph G with n vertices that is connected and acyclic must have $n - 1$ edges."

Base case: $n = 1$.

G is a single node with no edges; G is connected and acyclic.

Inductive hypothesis:

Suppose $P(n)$ holds for all values of n from our base case until some $k \geq 1$: That is, assume that any connected, acyclic graph G that has k vertices has $k - 1$ edges.

continued…

Quick Review: Trees

Recall: Let $G = (V, E)$ be an undirected graph on n nodes. Any two of the following statements implies the third (3.2 from K&T, page 78):

1. G is connected.
2. G does not contain a cycle (equivalently, G is *acyclic*).
3. G has $n - 1$ edges.

Prove $(1), (2) \implies (3)$

Claim 1: G must have some vertex v that is a leaf ($deg(v) = 1$) ✓

G cannot have any vertex u where $deg(u) = 0$ because G is connected.

Every vertex in G cannot have degree ≥ 2 because there would be a cycle: pick some vertex and walk at random until repeating a node. The walk cannot get stuck because every vertex has degree ≥ 2.

Quick Review: Trees

Recall: Let $G = (V, E)$ be an undirected graph on n nodes. Any two of the following statements implies the third (3.2 from K&T, page 78):

1. G is connected.
2. G does not contain a cycle (equivalently, G is *acyclic*).
3. G has $n - 1$ edges.

Prove $(1), (2) \implies (3)$

Now, remove some vertex v, where $deg(v) = 1$, along with its incident edge.

We are left with a graph G' that is still connected and still acyclic, and we can apply our inductive hypothesis to conclude that G' has $k - 1$ edges.

Adding vertex v and its incident edge back to G' does not introduce a cycle. G is connected, acyclic, and has $k + 1$ vertices and k edges.

Quick Review: Finding Connected Components

Algorithm. Given a graph $G = (V, E)$:

- Pick some vertex $v \in V$, and run $BFS(G, v)$. Let S be the set of vertices returned by the breadth-first search from v.
- Add S to the set of connected components, and repeat the process starting with some vertex that has not appeared in any connected component so far.
- When all vertices have been included, all connected components have been found.

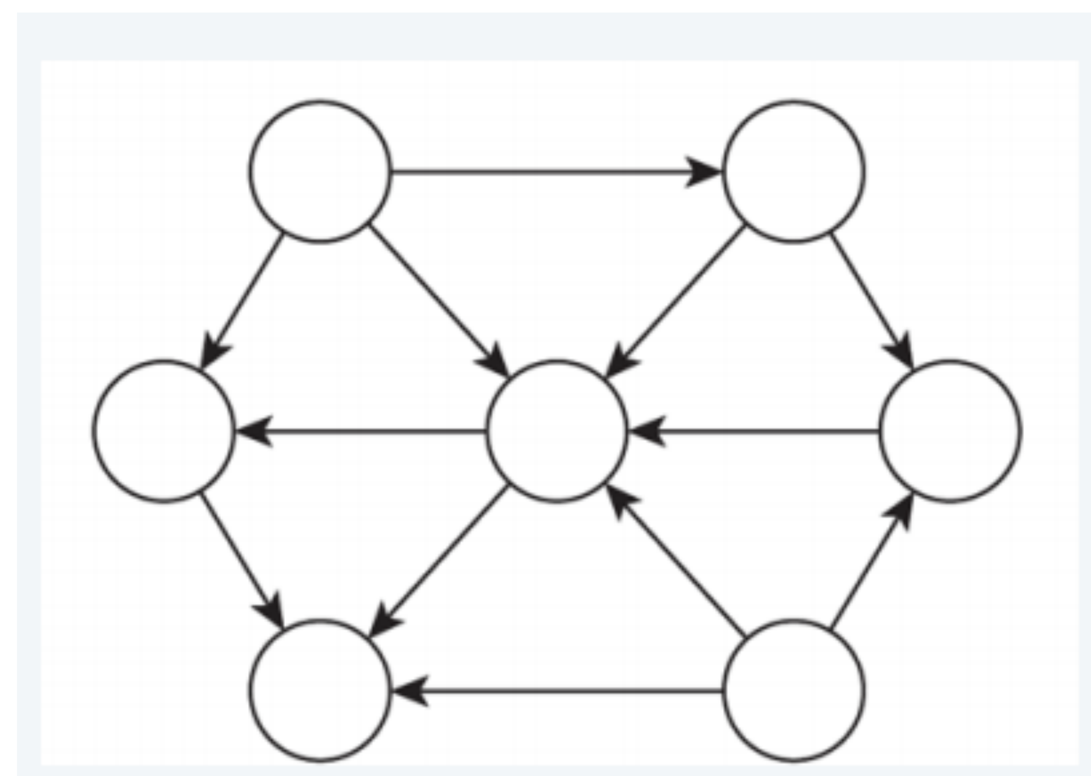
Running time?

Quick Review: Directed Graphs

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

- Edges have "orientation"
  - (u, v) (or sometimes denoted $u \to v$) leaves node u and enters node v
- Vertices have an "in-degree" and an "out-degree"

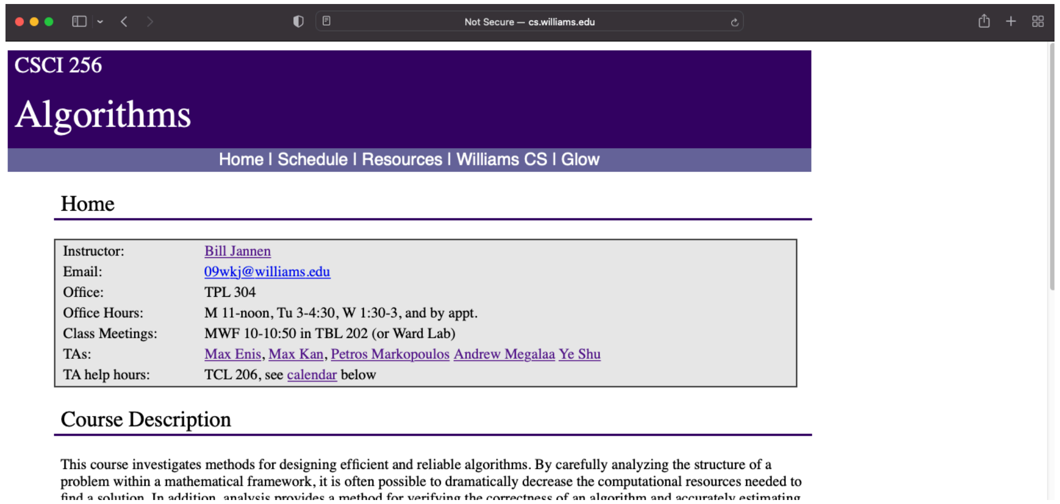
Rest of graph terminology extends to directed graphs: directed paths, cycles, etc.



Directed Graphs Examples

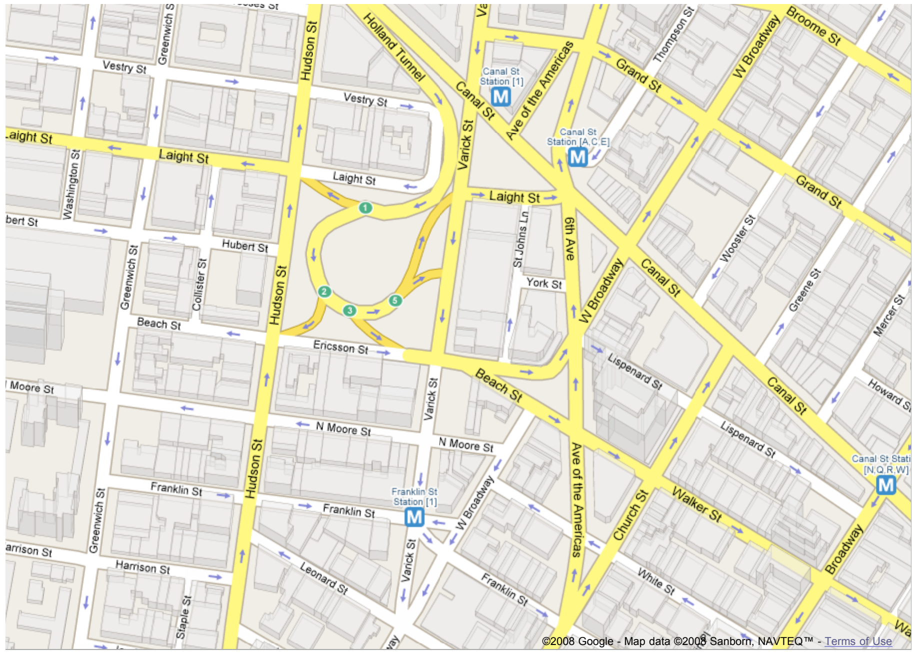
Web graph:

- Nodes: Webpages
- Edges: Hyperlinks
- Orientation of edges is crucial
- Search engines use hyperlink structure to rank web pages



Road network:

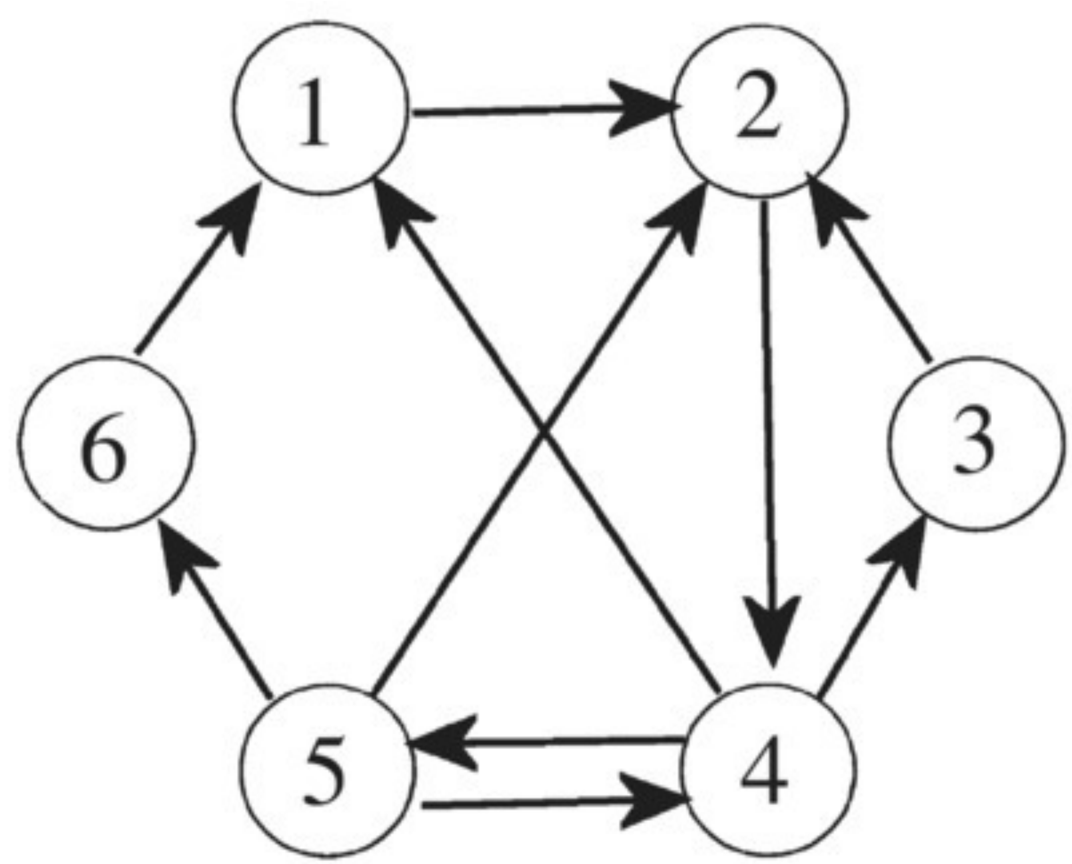
- Vertices: Intersections
- Edges: Streets (one-way)
- Raise your hand if you've navigated (recently) without a GPS app?



Directed Reachability

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

- Can use both BFS and DFS. They both visit exactly the set of nodes reachable from start node s (but perhaps different orders).

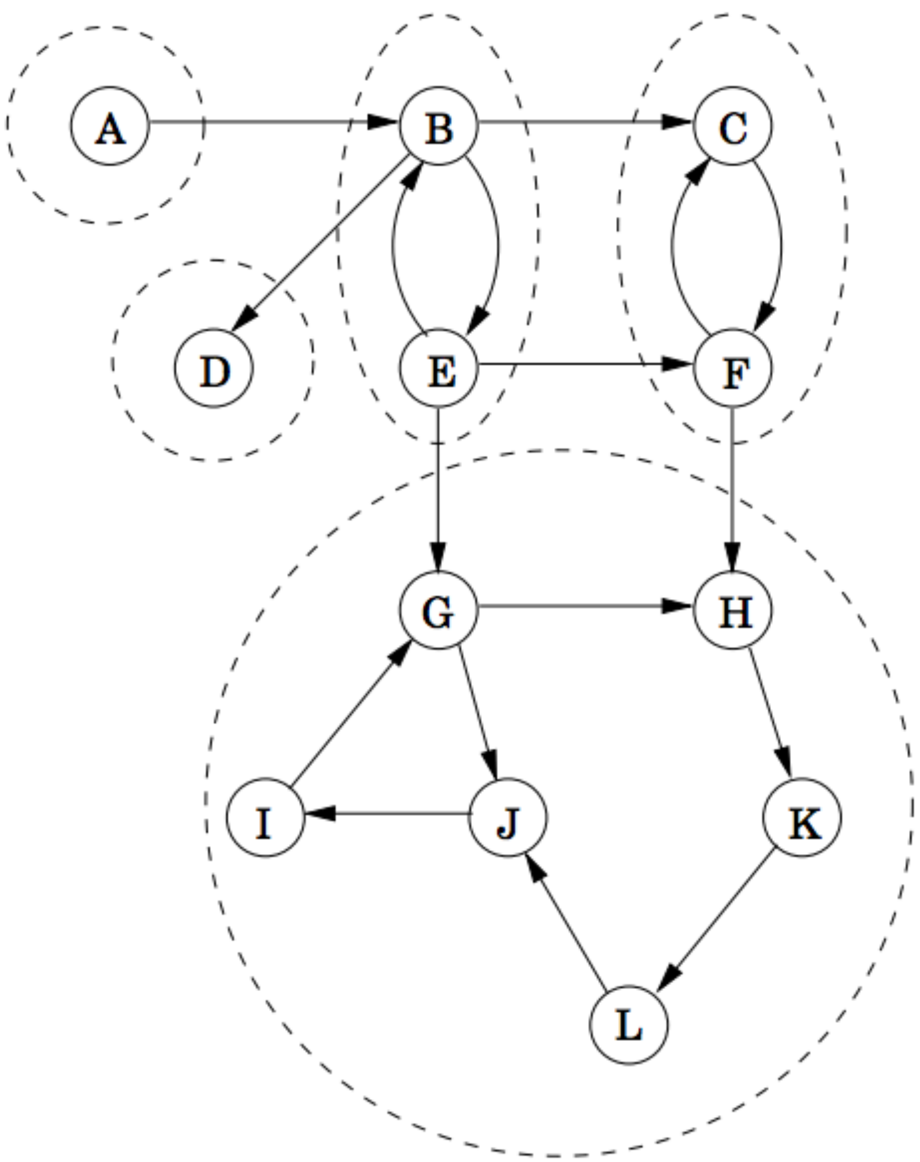


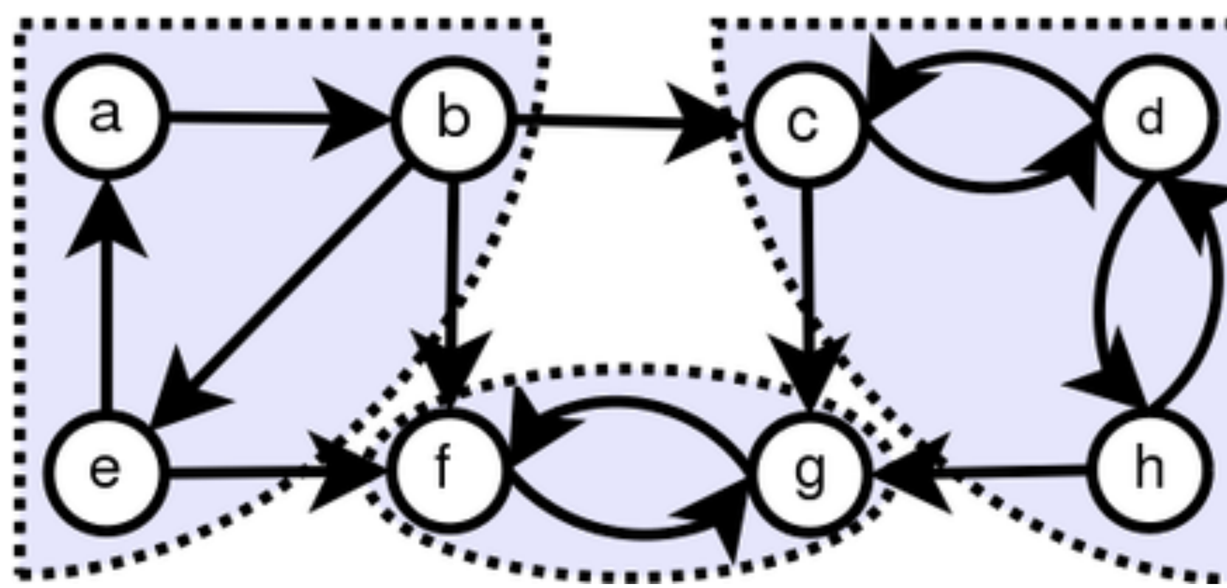
Strong Connectivity

- **Strong connectivity.** Connected components in directed graphs are 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

Strongly Connected Components

- **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 Strong Connectivity

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

Any ideas?

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).

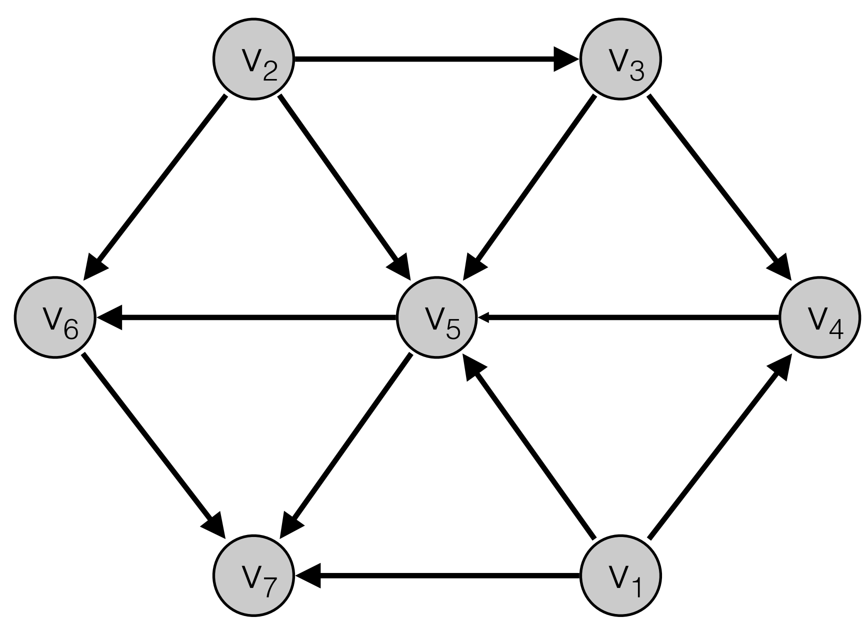
Analysis (Correctness)

- **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.

- DAG is typically pronounced, not spelled out
  - Rhymes with "bag"



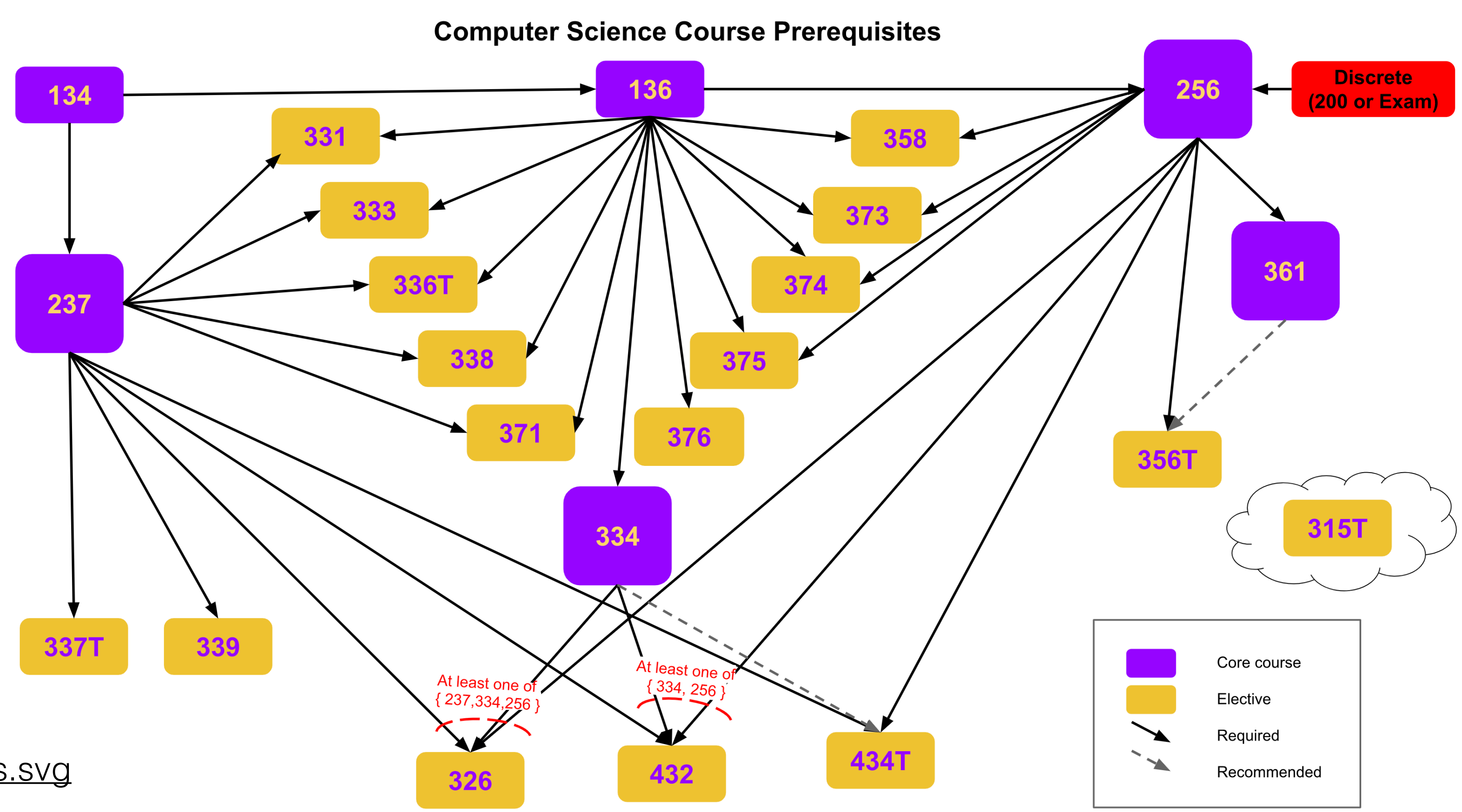
an example DAG

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.

(Said differently, if you number all of the vertices in your sequence of n vertices $v_1, \ldots, v_n$, then any edge that leaving a vertex v_i can only enter a vertex $v_{j>i}$)

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

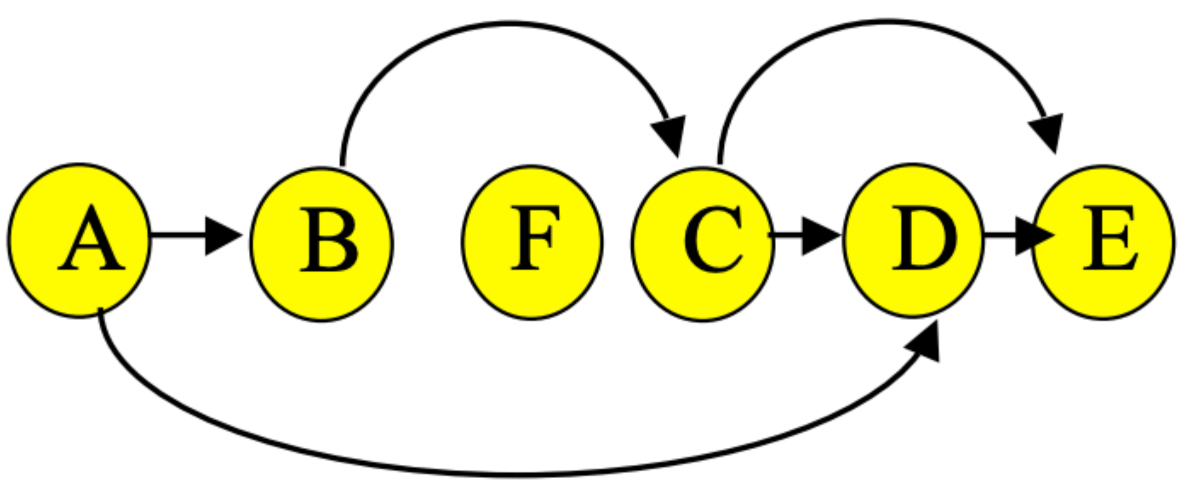


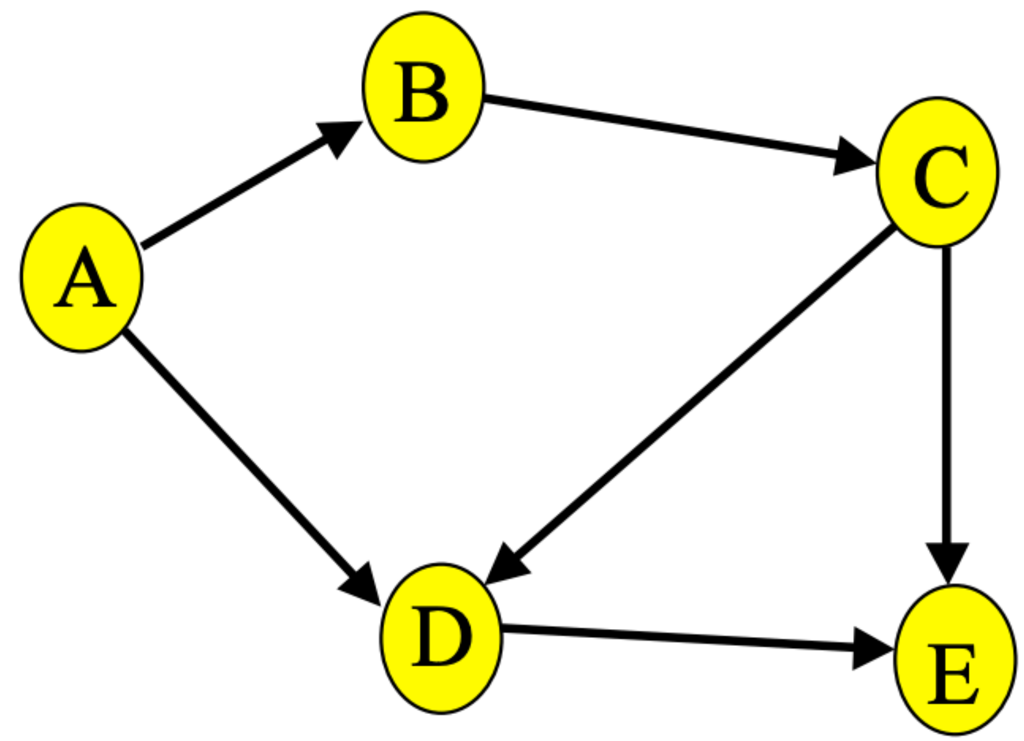
(Mostly) up-to-date

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

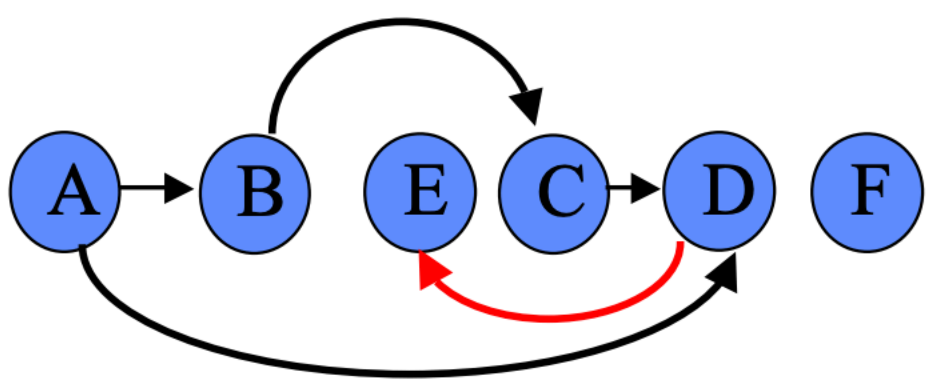




Taken from: https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Ordering: Example

Not a valid topological sort!



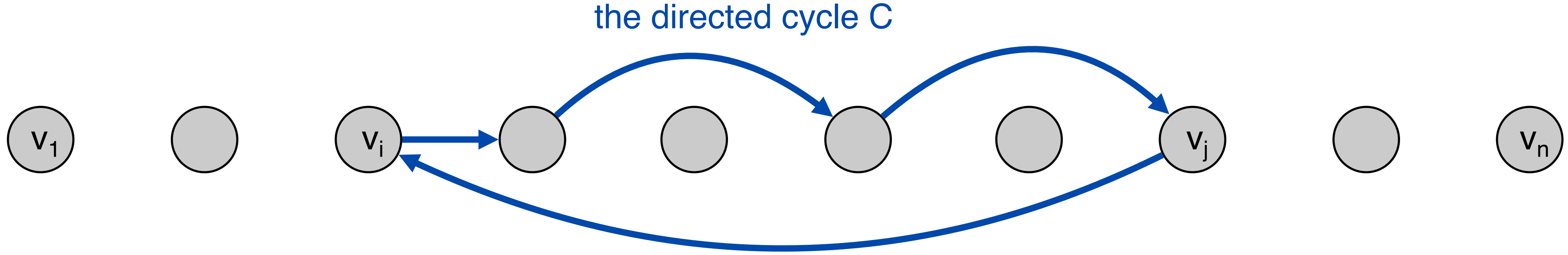
Taken from: https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

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

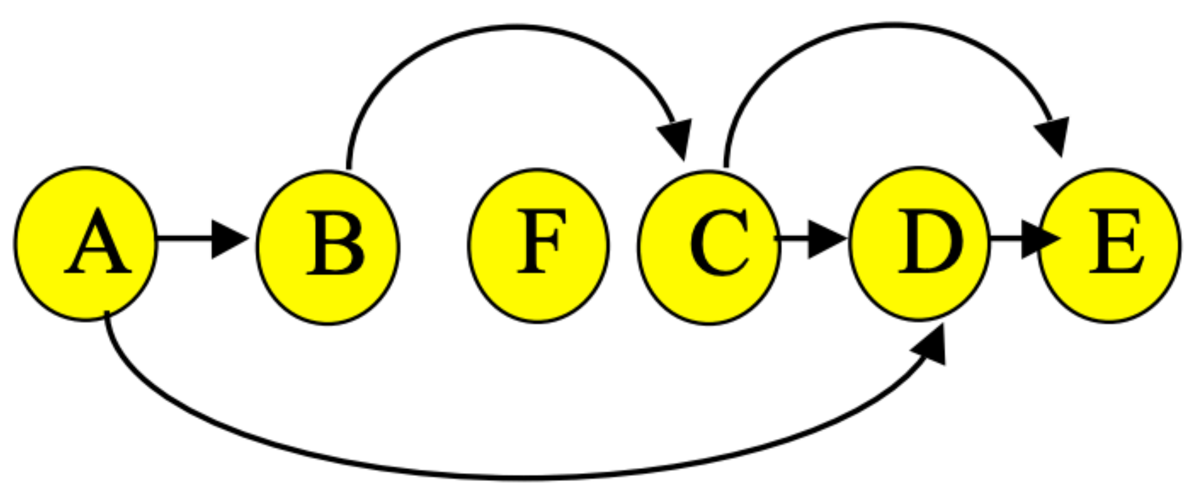
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; because C starts and ends on v_i, (v_j, v_i) is an edge
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, v_2, \ldots, v_n$ is a topological order, we must have $j < i$ ($\Rightarrow\!\Leftarrow$) ∎



the supposed topological order: $v_1, \ldots, v_n$

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 ($\Rightarrow\!\!\!\times\!\!\!\Leftarrow$) ∎

Idea for building a topological ordering: Repeatedly "remove" vertices that have in-degree 0 from the DAG.

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 inductive 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!

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!