CS3000: Algorithms & Data
Jonathan Ullman

Lecture 12:

- Applications of BFS: 2-Coloring, Connected Components, Topological Sort

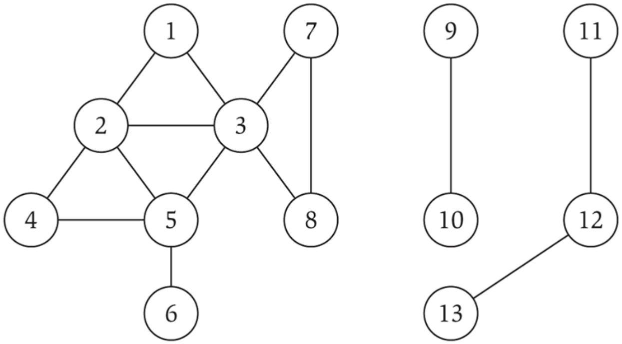
Oct 19, 2018

Recap: Graphs/BFS

Graphs: Key Definitions

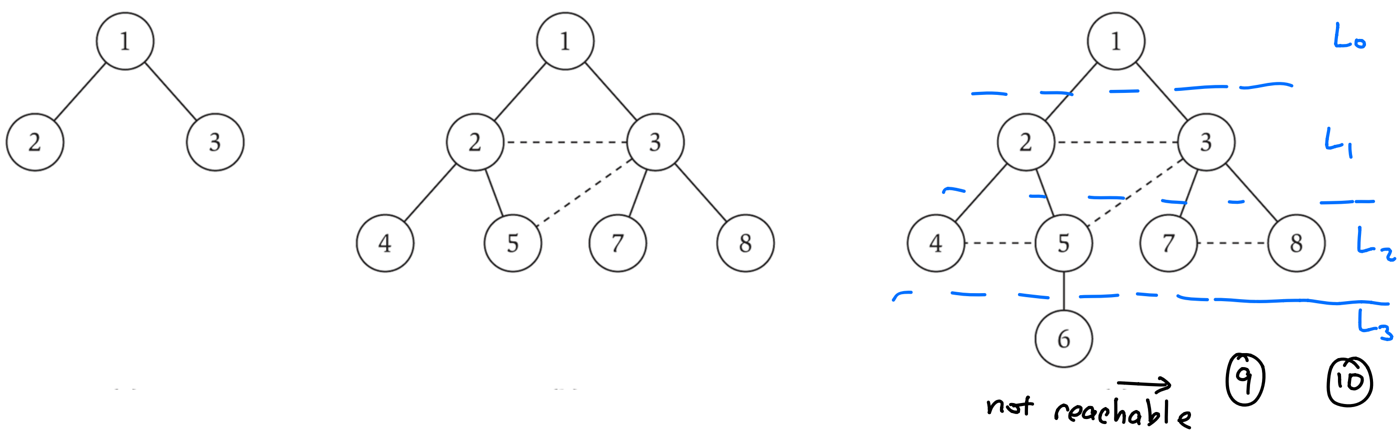
- **Definition:** A directed graph $G = (V, E)$
  - V is the set of nodes/vertices, $|V| = n$
  - $E \subseteq V \times V$ is the set of edges, $|E| = m$
  - An edge is an ordered $e = (u, v)$ “from u to v”
- **Definition:** An undirected graph $G = (V, E)$
  - Edges are unordered $e = (u, v)$ “between u and v”

- **Simple Graph:**
  - No duplicate edges
  - No self-loops $e = (u, u)$



Breadth-First Search (BFS)

- **Definition:** the distance between s, t is the number of edges on the shortest path from s to t
- **Thm:** BFS finds distances from s to other nodes
  - L_i contains all nodes at distance i from s
  - Nodes not in any layer are not reachable from s

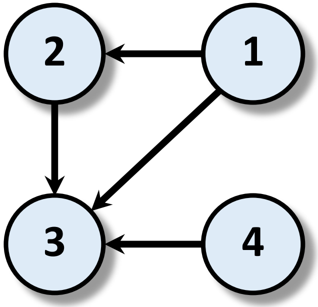


Adjacency Matrices

- The adjacency matrix of a graph $G = (V, E)$ with n nodes is the matrix $A[1{:}n\,, 1{:}n]$ where

$$A[i,j] = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$$

A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0



Cost

Space: $\Theta(n^2)$

Lookup: $\Theta(1)$ time

List Neighbors: $\Theta(n)$ time

Adjacency Lists (Undirected)

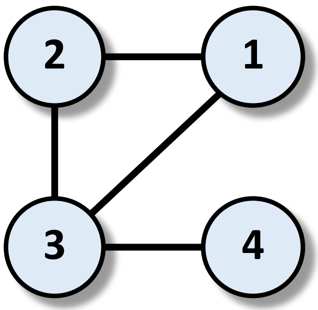
- The adjacency list of a vertex $v \in V$ is the list $A[v]$ of all u s.t. $(v, u) \in E$

$A[1] = \{2,3\}$

$A[2] = \{1,3\}$

$A[3] = \{1,2,4\}$

$A[4] = \{3\}$



<u>Cost</u>

Space: $\Theta(n + m)$

Lookup: $\Theta(\deg(u) + 1)$ time

List Neighbors: $\Theta(\deg(u) + 1)$ time

Breadth-First Search Implementation

```
BFS(G = (V,E), s):
  Let found[v] ← false ∀v, found[s] ← true
  Let layer[v] ← ∞ ∀v, layer[s] ← 0
  Let i ← 0, L0 = {s}, T ← ∅

  While (Li is not empty):
    Initialize new layer Li+1
    For (u in Li):
      For ((u,v) in E):
        If (found[v] = false):
          found[v] ← true, layer[v] ← i+1
          Add (u,v) to T and add v to Li+1
    i ← i+1
```

Implements BFS in $O(n+m)$ time

Time is really $O(n_s + m_s)$ where n_s = # of nodes reachable from s, m_s = " " edges

Bipartiteness / 2-Coloring

2-Coloring

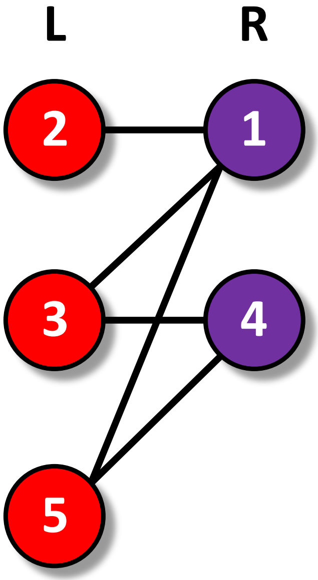
- **Problem:** Tug-of-War Rematch
  - Need to form two teams $\boldsymbol{R}, \boldsymbol{P}$
  - Some students are still mad from last time
- **Input:** Undirected graph $G = (V, E)$
  - $(u, v) \in E$ means u, v wont be on the same team
- **Output:** Split V into two sets $\boldsymbol{R}, \boldsymbol{P}$ so that no pair in either set is connected by an edge


2-Coloring (Bipartiteness)

- **Equivalent Problem:** Is the graph G bipartite?
  - A graph G is bipartite if I can split V into two sets L and R such that all edges $(u, v) \in E$ go between L and R

$L \cap R = \emptyset$

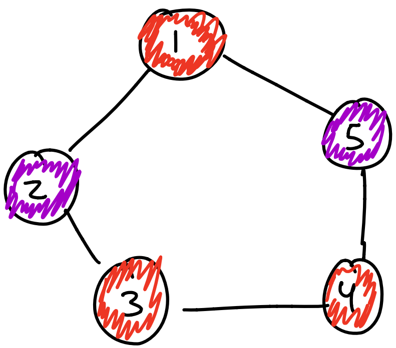
$L \cup R = V$



Designing the Algorithm

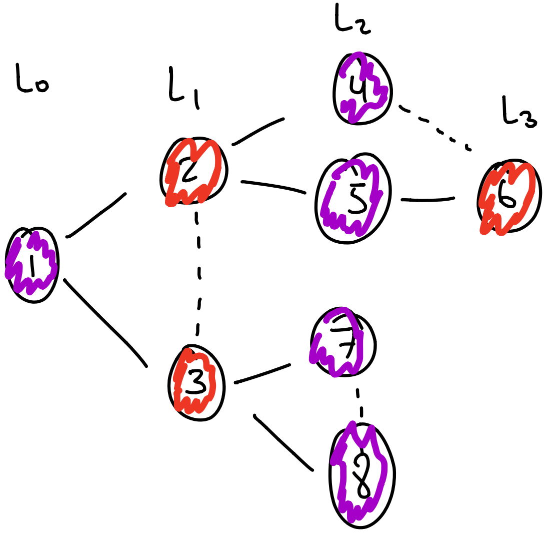
- **Key Fact:** If G contains a cycle of odd length, then G is not 2-colorable/bipartite

Proof by Picture:



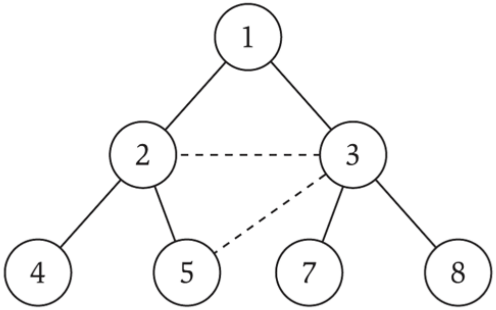
Designing the Algorithm

- **Idea for the algorithm:**
  - BFS the graph, coloring nodes as you find them
  - Color nodes in layer i **purple** if i even, **red** if i odd
  - See if you have succeeded or failed



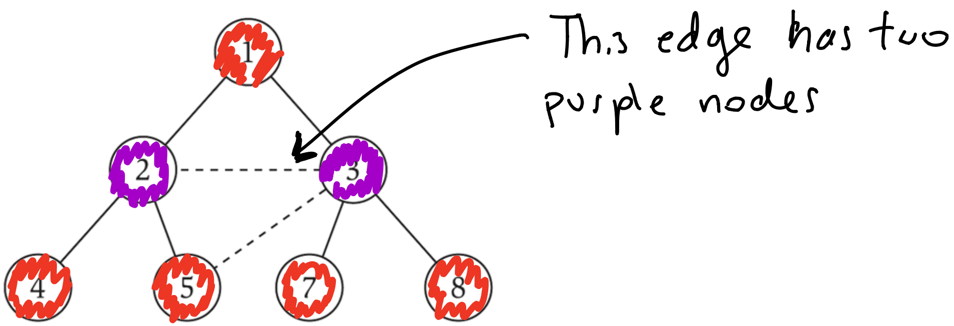
Designing the Algorithm

- **Claim:** If BFS 2-colored the graph successfully, the graph has been 2-colored successfully
- **Key Question:** Suppose you have not 2-colored the graph successfully, maybe someone else can do it?



Designing the Algorithm

- **Claim:** If BFS fails, then G contains an odd cycle
  - If G contains an odd cycle then G can't be 2-colored!
  - Example of a phenomenon called **duality**



Designing the Algorithm

If BFS colors incorrectly ⇒ ∃ odd cycle

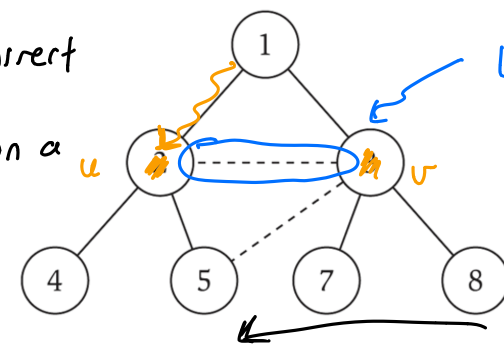
- **Claim:** If BFS fails, then G contains an odd cycle
  - If G contains an odd cycle then G can't be 2-colored!
  - Example of a phenomenon called **duality**

① Black edges are colored correctly (b/c they go from E to O)

② Dotted edge are ethe $L_i \leftrightarrow L_{i+1}$ or $L_i \leftrightarrow L_i$

(2a) $L_i \leftrightarrow L_{i+1}$ edges are correct

(2b) Any $L_i \leftrightarrow L_i$ edge is on a cycle of length $2i+1$, which is an odd #



$L_i \leftrightarrow L_i$ edge

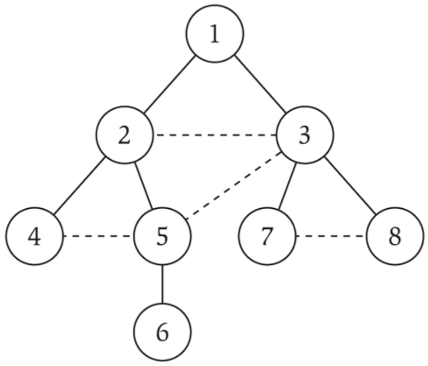
- There is a path of length i from (s) ⇝ (u)
- There is a path of length i from (s) ⇝ (v)
- Cycle (s) ⇝ (u) → (v) ⇝ (s)

$i + 1 + i$

Topological Sort

Acyclic Graphs

- **Acyclic Graph:** An undirected graph with no cycles
  - Also known as a forest
  - If it's connected then it's known as a tree
- Can test if a graph has a cycle in $O(n + m)$ time
  - Run BFS
  - If there are any edges that are **not** in the BFS tree, then they form a cycle

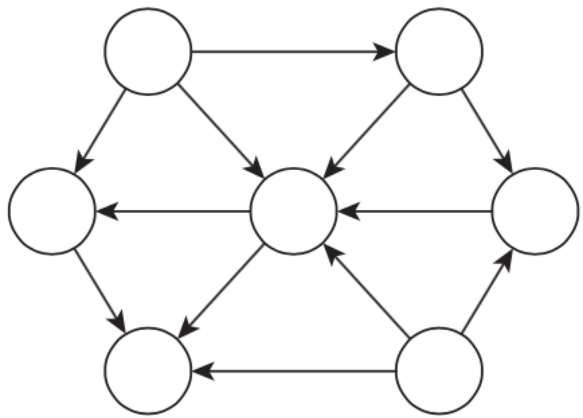


Directed Acyclic Graphs (DAGs)

A path

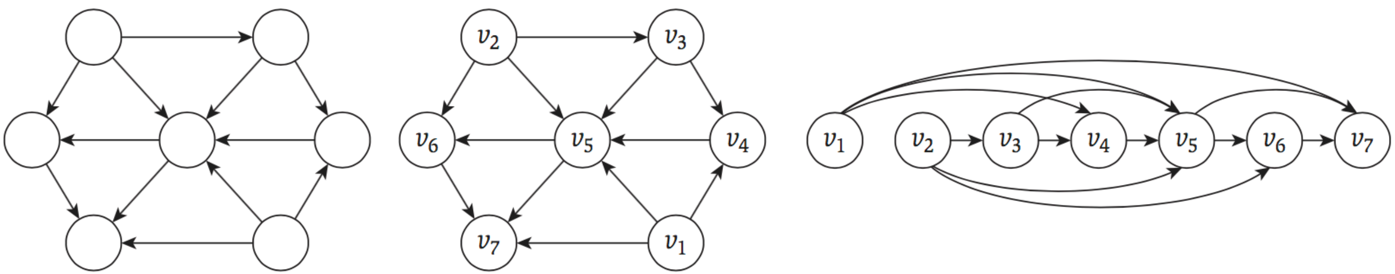
$u \to v_1 \to v_2 \to \dots \to v_k \to u$

- **DAG:** A directed graph with no directed cycles
- Can be much more complex than a forest



Directed Acyclic Graphs (DAGs)

- **DAG:** A directed graph with no directed cycles
- DAGs represent precedence relationships



- A topological ordering of a directed graph is a labeling of the nodes from $v_1, \ldots, v_n$ so that all edges go "forwards", that is $(v_i, v_j) \in E \Rightarrow j > i$
  - G has a topological ordering $\Rightarrow$ G is a DAG

Any directed cycle means G cannot be top. ordered

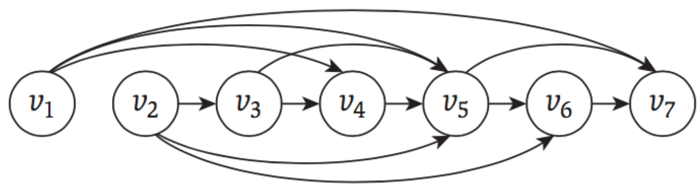
Directed Acyclic Graphs (DAGs)

- **Problem 1:** given a digraph G, is it a DAG?
- **Problem 2:** given a digraph G, can it be topologically ordered?

- **Thm:** G has a topological ordering $\Leftrightarrow$ G is a DAG
  - We will design one algorithm that either outputs a topological ordering or finds a directed cycle
  - Another example of duality

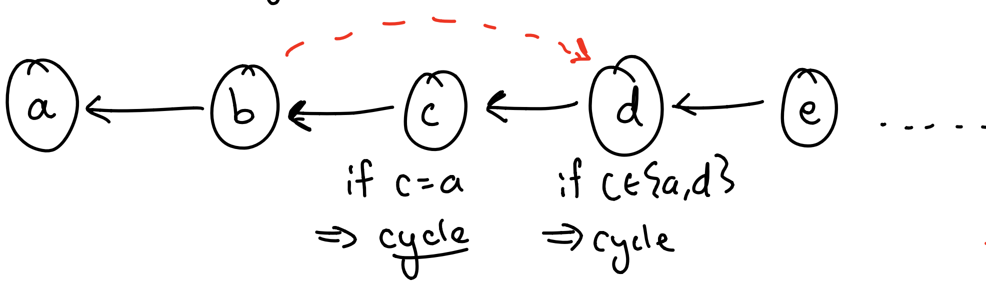
Topological Ordering

- **Observation:** the first node must have no in-edges



- **Observation:** In any DAG, there is always a node with no incoming edges

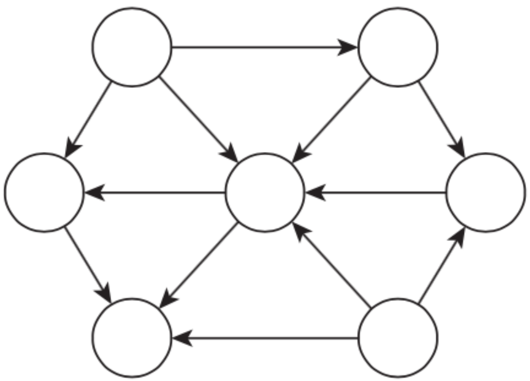
Proof: Suppose every node has an in-edge. Follow in-edges until either we get a directed cycle or run out of nodes



If every node has an in-edge, we can repeat for n+1 nodes ⇒ cycle

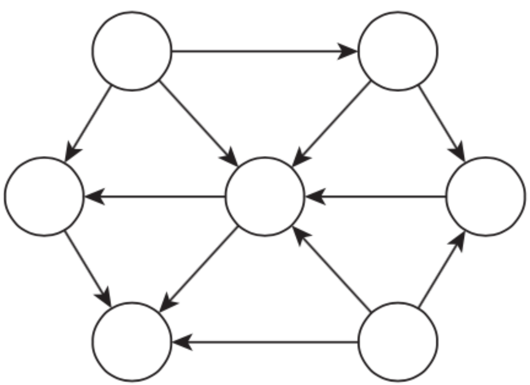
Topological Ordering

- **Fact:** In any DAG, there is a node with no incoming edges
- **Thm:** Every DAG has a topological ordering
- **Proof (Induction):**
  - $H(n)$ = Every DAG w/ n nodes has a topological ordering
  - Goal: prove $\forall n \in \mathbb{N}$, $H(n)$ is true
  - Base Case: $n = 1$ (trivial)



Topological Ordering

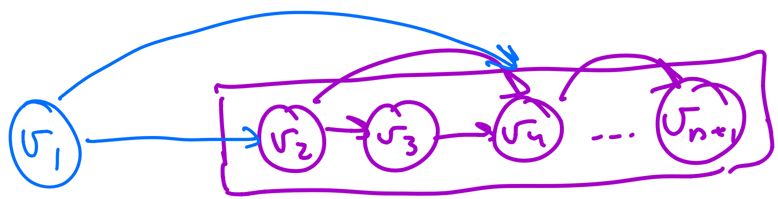
- **Fact:** In any DAG, there is a node with no incoming edges
- **Thm:** Every DAG has a topological ordering
- **Proof (Induction):**



Inductive Step: To show $H(n) \Rightarrow H(n+1)$

① Let v_1 be a node with no in-edges

② Let ◯ be the ordering of $V \setminus \{v_1\}$



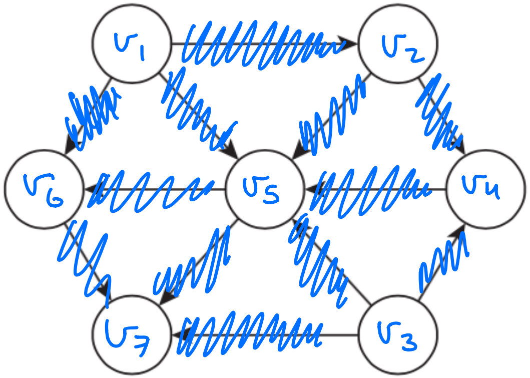
③ Use the ordering $v_1 \rightarrow$ ◯ for the whole graph.

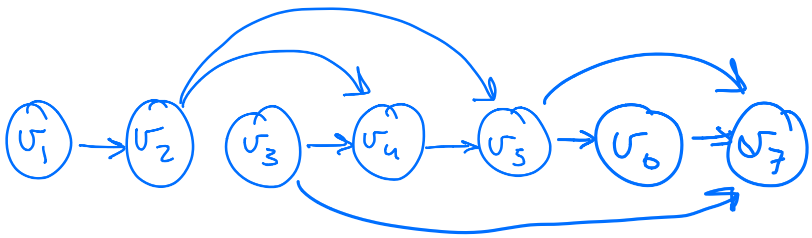
Implementing Topological Ordering

```
SimpleTopOrder(G):
  Set i ← 1
  Until (G has no more nodes):
    Find a node u with no incoming edges
    Label u as node i, increment i ← i+1
    Remove u and its edges from G
```

Guaranteed to exist

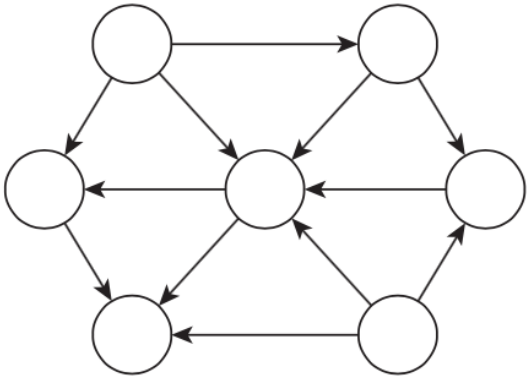
Modify adjacency list in time O(m)





Implementing Topological Ordering

```
SimpleTopOrder(G):
  Set i ← 1
  Until (G has no more nodes):
    Find a node u with no incoming edges
    Label u as node i, increment i ← i+1
    Remove u and its edges from G
```



Implementing Topological Ordering

Iterate n times

```
SimpleTopOrder(G):
  Set i ← 1
  Until (G has no more nodes):
    Find a node u with no incoming edges      O(1)
O(1) Label u as node i, increment i ← i+1
    Remove u and its edges from G
```

Can implement in $O(m)$ time (if I represent w/ both A_{in} and A_{out})

Overall time is $O(nm)$

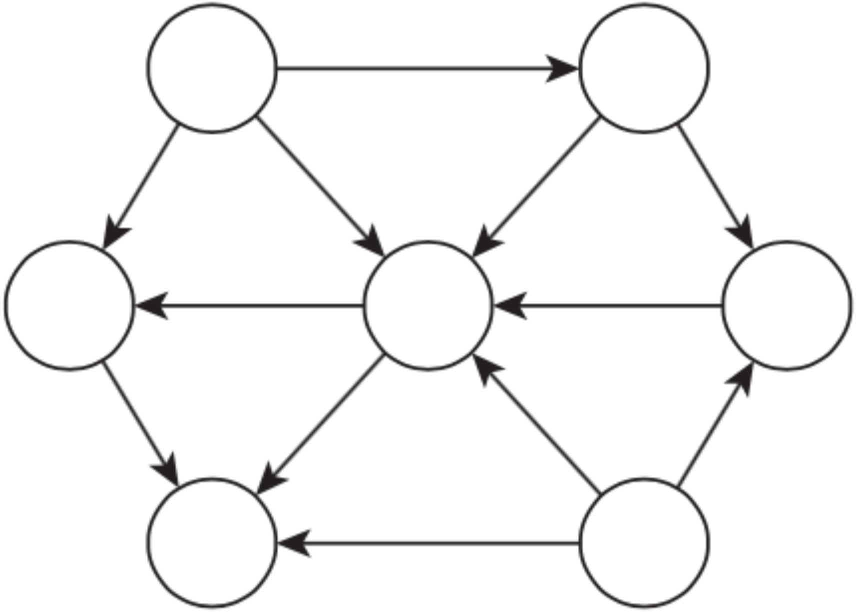
- If we only keep outgoing edges in the list then
  - ... final step is $O(1)$
  - ... first step is $O(n)$

Overall time is $O(n^2)$

Fast Topological Ordering

```
FastTopOrder(G):
  Mark all nodes with their # of in-edges
  Call a node INACTIVE if it's mark is 0
  Call a node ACTIVE otherwise
  Let i = 1
  Until (all node are INACTIVE):
    Let u be an INACTIVE
    Label u as node i in the top. order
    Let i = i+1
    For (every (u,v) in E):
      Decrease v's mark by 1
```

Fast Topological Ordering Example



Topological Ordering Summary

- **DAG:** A directed graph with no directed cycles
- Any DAG can be **toplogically ordered**
  - There is an algorithm that either outputs a topological ordering or finds a directed cycle in time $O(n+m)$

next tuesday