Midterms back after class.

Make sure I stop at 11:15

CS4800: Algorithms & Data
Jonathan Ullman

Lecture 12:
- Graph Search: BFS Applications, DFS

Feb 20, 2018

BFS Review

Given a graph $G=(V,E)$ and a "source" $s \in V$

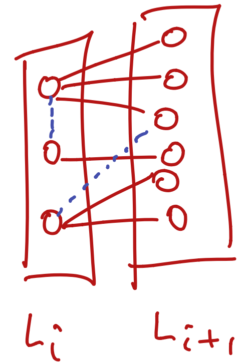
- BFS Algorithm:
  - Input: source node s
  - $L_0 = \{s\}$
  - $L_1 =$ all neighbors of L_0
  - $L_2 =$ all neighbors of L_1 that are not in L_0, L_1
  - …
  - $L_d =$ all neighbors of L_{d-1} that are not in $L_0, \ldots, L_{d-1}$

Algorithm makes sense for directed or undirected

BFS Review

time: $\sum_{u \in V} O(\deg(u) + 1) = O(n+m)$

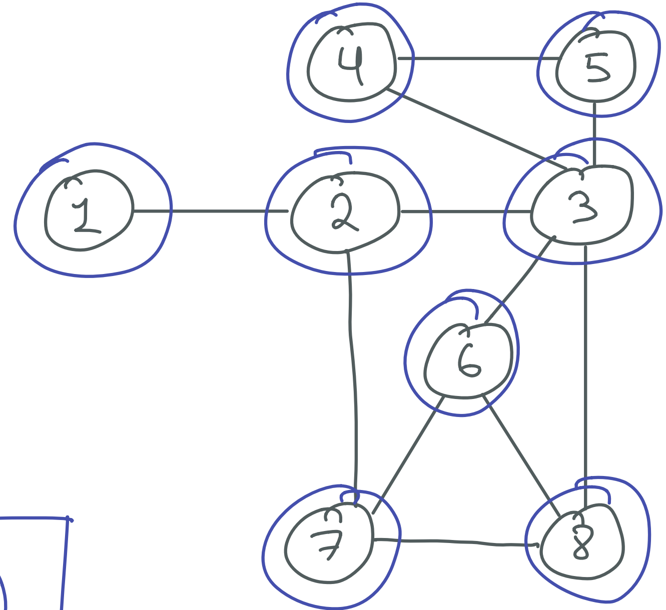
Set found$[v]$ ← false for every node v
BFS(s):
  Initialize found$[s]$ ← true
  Initialize $i \leftarrow 0$, $L_0 \leftarrow \{s\}$, $T \leftarrow \emptyset$
  While (L_i is not empty):
    Initialize a new layer $L_{i+1} \leftarrow \emptyset$
    For ($u \in L_i$):
      For ($(u, v) \in E$):
        If (found$[v]$ = false) Then:
          Set found$[v]$ ← true
          Add (u, v) to T, add v to L_{i+1}
    Increment the layer $i \leftarrow i + 1$

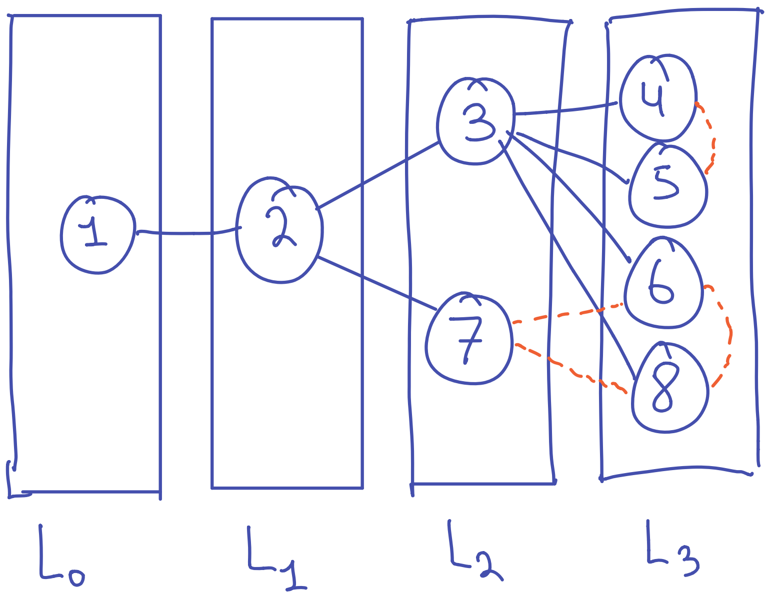


Running time is mostly by time to explore each node

BFS Review

- BFS this graph from s=1





- Partition into layers
- Blue edges form a tree
- Any orange edge makes a cycle (undirected)

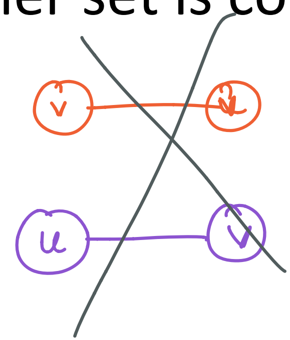
BFS Review

- Last time we saw that BFS can…
  - … find the set of nodes that are reachable from s
  - … find the distances from s to all other nodes t
  - … find shortest paths from s to all other nodes t
  - … find a cycle in an undirected graph
  - … identify connected components in undirected graphs
- Today:
  - Using BFS to…
    - … split the nodes into two "teams" (2-coloring/bipartiteness)
    - … find strongly connected components in directed graphs
  - Topological Sort / Order
  - DFS (Depth-First Search)

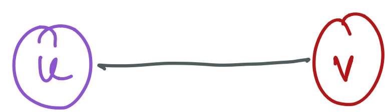
2-Coloring/Bipartiteness

2-Coloring

- Problem: Tug-of-War Rematch
  - Need to form two teams R, P
  - Some students are still mad from last time…
- Input: Undirected graph $G = (V, E)$
  - $(u, v) \in E$ means u, v can't be on then same team
- Output: Split V into two sets R, P so that no pair in either set is connected by an edge, or say not possible



no

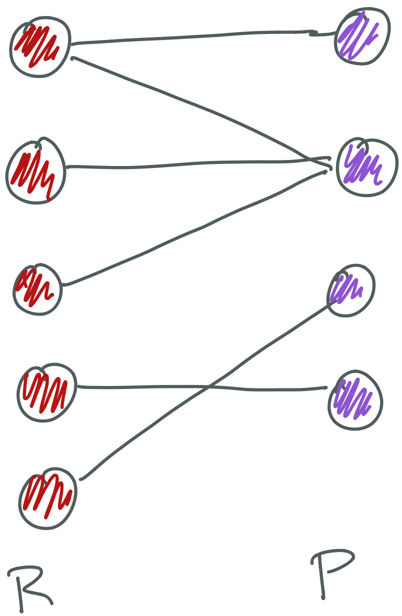


yes

2-Coloring/Bipartiteness

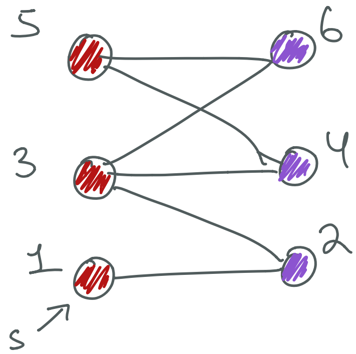
- Alternative Phrasing: Is the graph G bipartite?

Bipartite Graph:

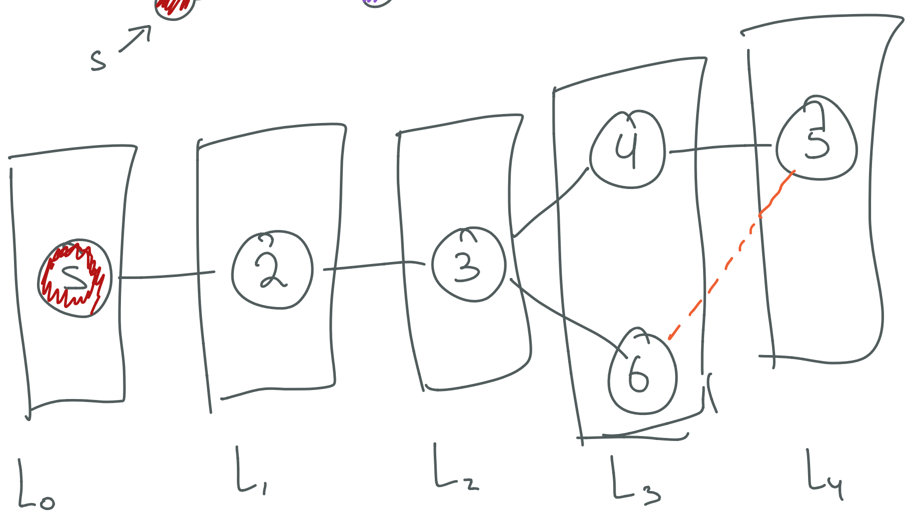


$V = R \cup P$ s.t. no edge (u,v) has $u,v \in R$ or $u,v \in P$

Suppose G <u>is</u> bipartite / G <u>has</u> a 2-coloring



- BFS (from any node) will correctly 2-color

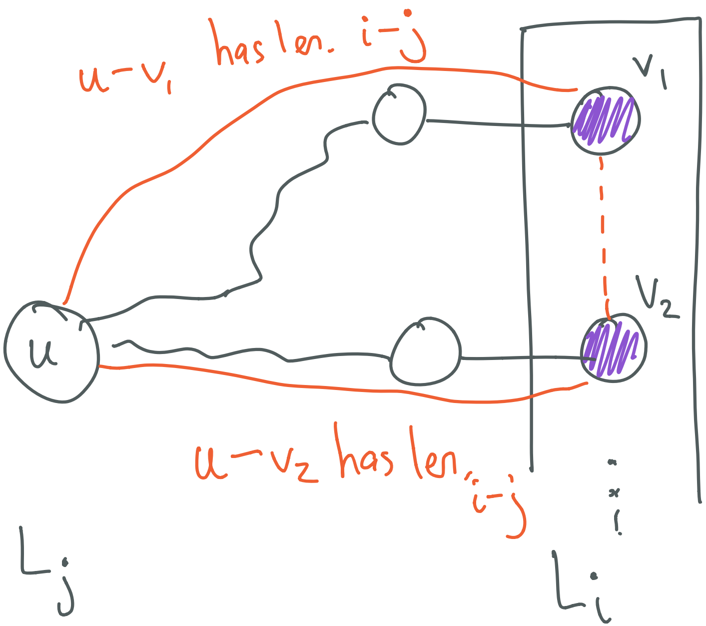


Designing the Algorithm

- Optimistic Algorithm: $O(n+m)$ time
  1. Pick an arbitrary start node s
  2. BFS the graph from s, coloring nodes as you find them
  3. Color nodes in layer i purple if i even, red if i odd
  4. See if you found a legal coloring

Correctness?

- If you 2-colored the graph successfully, the graph can be 2-colored successfully
- If you have not 2-colored the graph successfully, maybe you should just try harder?



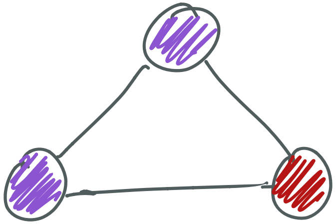
$$u \overset{i-j}{\rightsquigarrow} v_1 \overset{1}{\text{---}} v_2 \overset{i-j}{\rightsquigarrow} u$$

$2i - 2j + 1$ odd

Correctness?

- Key Fact: If G has an odd-length cycle then there is no legal 2-coloring

proof-by-picture



Summary of 2-Coloring

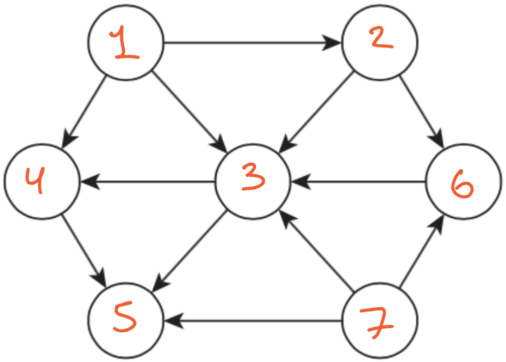
- Can 2-color or find an odd cycle in $O(n+m)$ time.

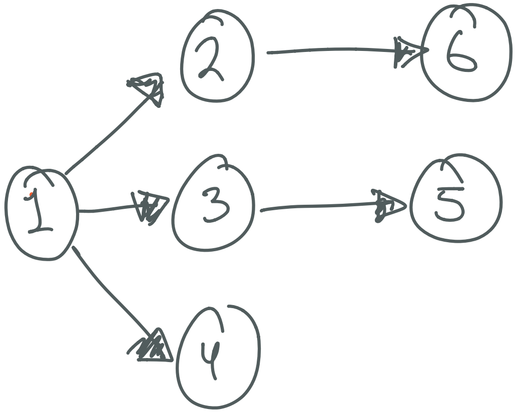
- A graph can be 2-colored iff it has no odd cycles.

BFS in Directed Graphs
(Strongly) Connected Components

BFS in Directed Graphs

- BFS works in directed graphs

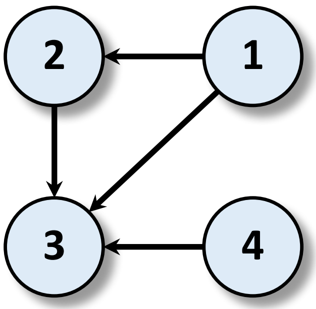




BFS still finds all nodes reachable from s and the shortest path

Adjacency Lists for Directed Graphs

- The adjacency list of a vertex $v \in V$ is the list $A_{out}[v]$ of all edges $(v, u) \in E$ and the list $A_{in}[v]$ of all edges $(u, v) \in E$



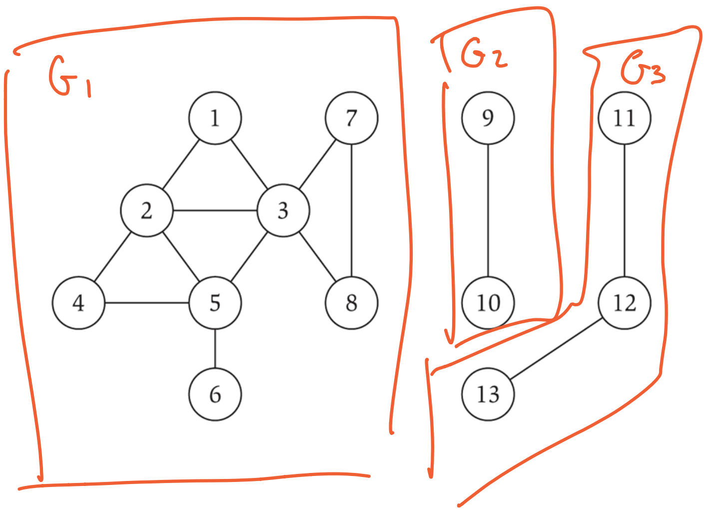
$$A_{out}[1] = \{2,3\} \qquad A_{in}[1] = \{\}$$
$$A_{out}[2] = \{3\} \qquad A_{in}[2] = \{1\}$$
$$A_{out}[3] = \{\ \} \qquad A_{in}[3] = \{1,2,4\}$$
$$A_{out}[4] = \{3\} \qquad A_{in}[4] = \{\}$$

Connected Components

- An undirected graph G is connected if for every pair of nodes $u, v \in V$, u is reachable from v
- The connected component of s is the set of nodes reachable from s

$v \in CC(s)$ then $s \in CC(v)$

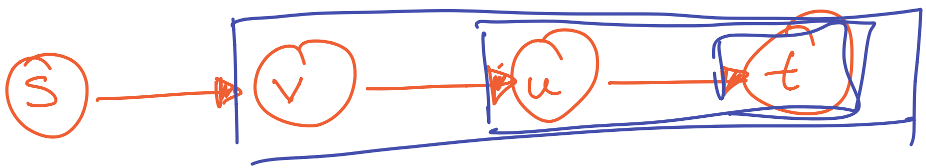
- Can partition G into connected components



Strongly Connected Components

- A directed graph G is strongly connected if for every pair $u, v \in V$, u, v are mutually reachable
- The strongly connected component of s is the set of nodes t such that s, t are mutually reachable

scc(s) = nodes in cc(s) s.t. s ∈ cc(t)



cc(s) = {v, u, t}

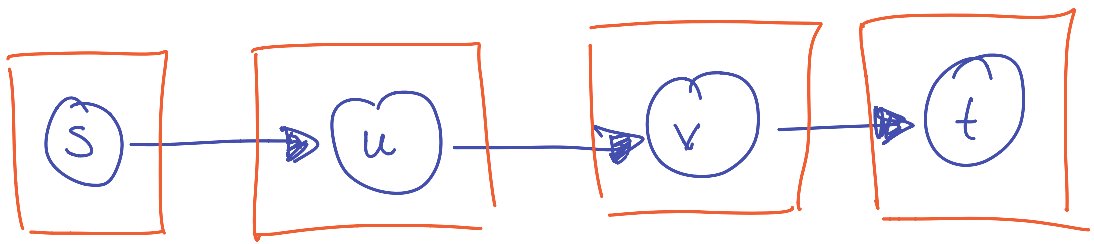
cc(v) = {u, t}

cc(u) = {t}

Strongly Connected Components

- A directed graph G is strongly connected if for every pair $u, v \in V$, u, v are mutually reachable
- The strongly connected component of s is the set of nodes t such that s, t are mutually reachable

scc(s) = nodes in cc(s) s.t. s ∈ cc(t)

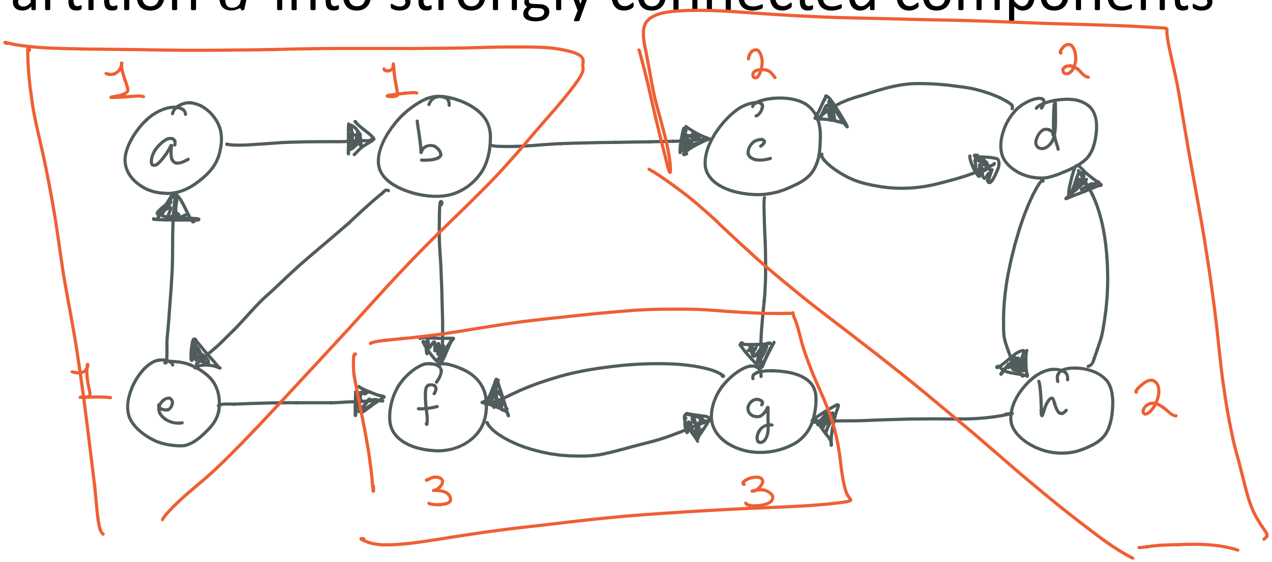


If t ∈ scc(s) then s ∈ scc(t)

· Can partition into strongly connected components

Ask the Audience

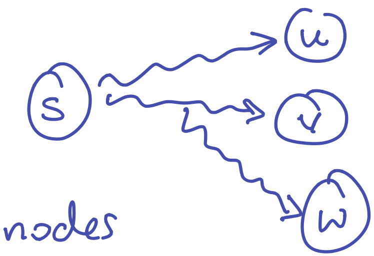
- Partition G into strongly connected components



Strongly Connected Components

- Problem: Given s find $SCC(s)$
- Algorithm:

1: Use BFS to find all nodes reachable from s in G



2: Let G^{back} be a graph w/ the same nodes and $(u,v) \in E \iff (v,u) \in E^{back}$

a path from u to s
$\Rightarrow$ a way to go from s to u following edges "backwards"

3: Use BFS to find all nodes reachable from s in G^{back}

4: Output the set of nodes v reachable in both.

Connected Components Recap

- Partition an undirected graph into connected components in $O(n+m)$ time
- Test if a directed graph is strongly connected in $O(n+m)$ time
  - Find the strong component of s in $O(n+m)$ time two BFSs
  - Can partition in $O(n+m)$ with more cleverness

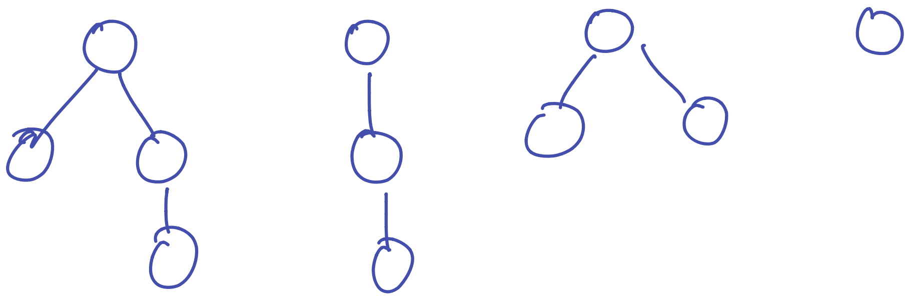
Partition into SCCs in $O(n(n+m))$ time

- Upshot: we tend to assume graphs are connected

Topological Sort

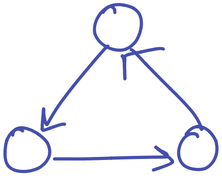
Acyclic Graphs

- Acyclic Graph: An undirected graph with no cycles.
- Can test if a graph has a cycle in $O(n + m)$ time
- An acyclic undirected graph is called a forest

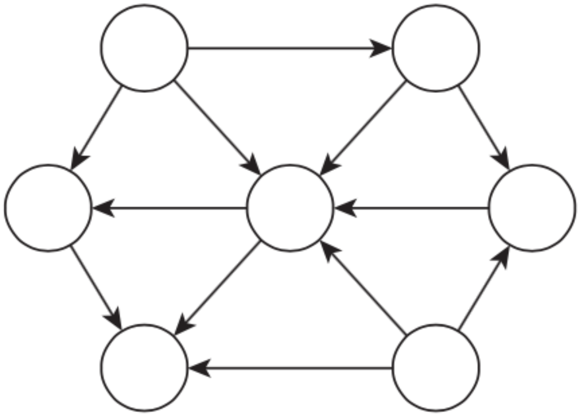


if connected
it's a tree

Directed Acyclic Graphs (DAGs)



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

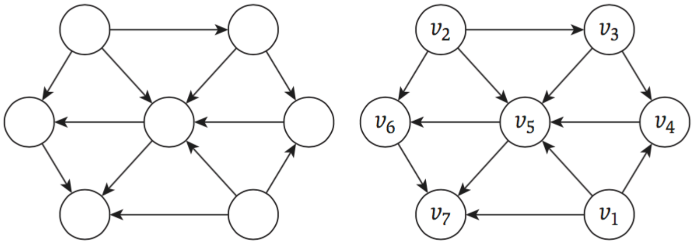


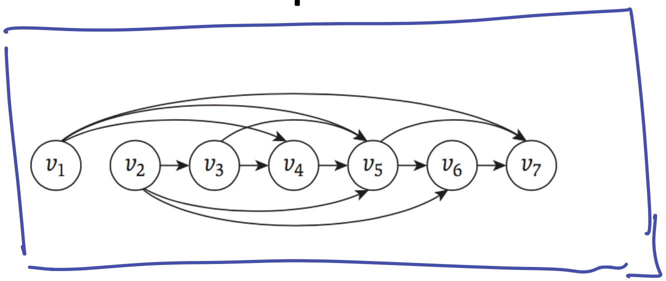
a set of edges

$(v_1, v_2)(v_2, v_3) \cdots (v_{k-1}, v_1)$

Directed Acyclic Graphs (DAGs)

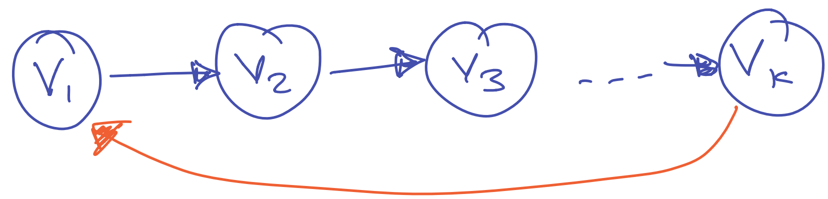
- DAG: A directed graph with no directed cycles
- DAGs represent precedence relationships





- 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" $(v_i, v_j) \in E \Rightarrow j > i$
  - G has a topological ordering $\Rightarrow$ G is a DAG

If G has a top. ordering, it's a DAG



If a graph has a directed cycle it cannot be top. ordered.

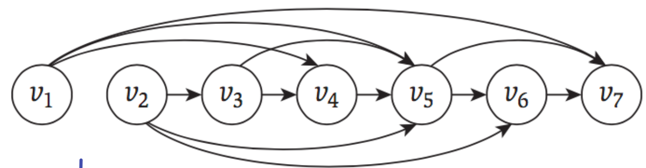
Directed Acyclic Graphs (DAGs)

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

- Theorem: G has a top. ordering $\Leftrightarrow$ G is a DAG
- We will design one algorithm that either outputs a topological ordering or finds a directed cycle

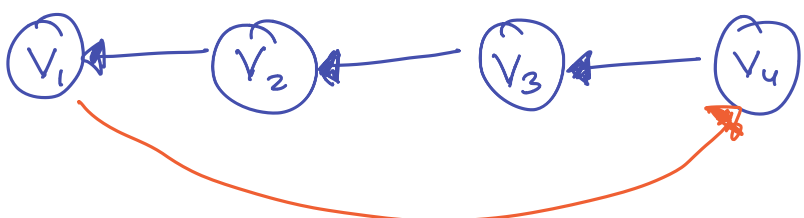
Topological Ordering

- Simple Observation: the first node must have no incoming edges



at least one

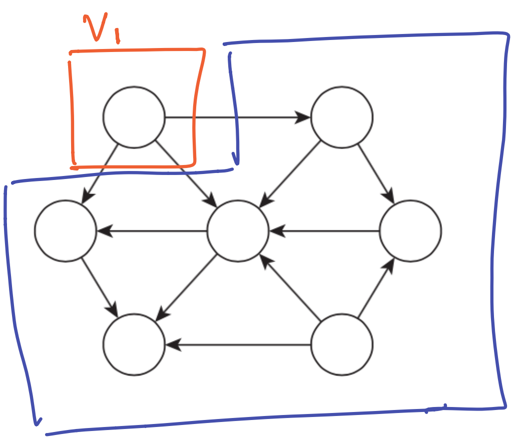
- In any DAG, there is a node with no incoming edges

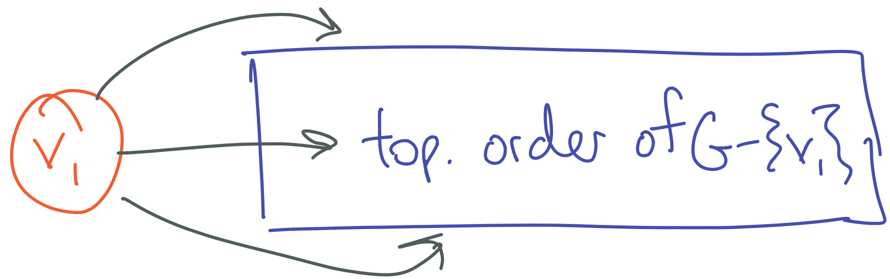


Keep following incoming edges until you either find a directed cycle or run out of nodes

Topological Ordering

- In any DAG, there is a node with no incoming edges
- Theorem: Every DAG has a topological ordering
- Proof by Induction: (by induction on n)
  - Base Case (n=1): Trivial
  - Inductive Step:



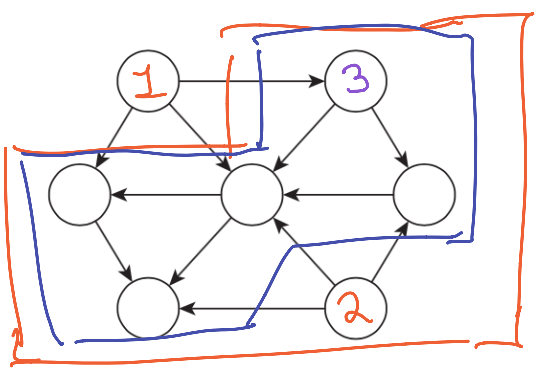


Implementing Topological Ordering

- Simple Algorithm:
  1. Set $i \leftarrow 1$
  2. Until we're out of nodes
     a) Find a node w with no incoming edges, label it v_i
     b) Eliminate w and all its edges

do loop n times

O(n) time



V_1 V_2 V_3

time is $O(n^2)$

can be improved to $O(n+m)$

Fast Topological Ordering

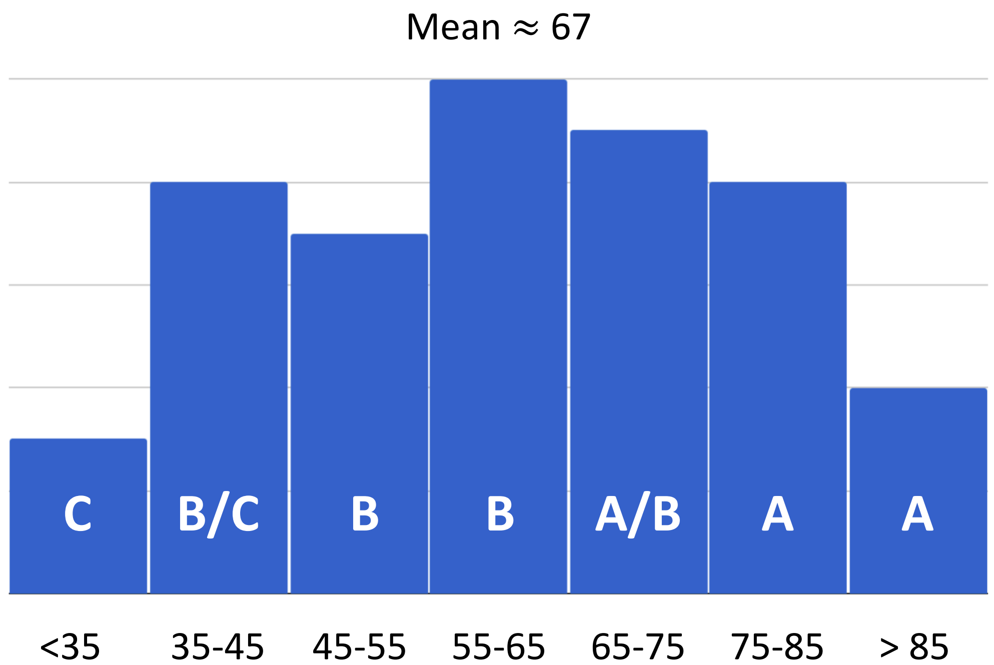
1. Set all nodes to active
2. Label nodes with # of incoming edges from active nodes
3. Let S be a set of active nodes with label 0
4. Find a node w with label 0 and add it to S
5. Set $i \leftarrow 1$
6. Until we're out of nodes
   a) Choose a node in $w \in S$ call it v_i
   b) For every edge (w, u), decrease u's label, if u's label drops to 0 then add it to the set S
   c) Increment $i \leftarrow i + 1$

Allow me to geek out for a minute

- We saw the first example of two amazing themes:
  - Using algorithms to prove mathematical facts
    - G is bipartite $\Leftrightarrow$ G contains no odd cycles
  - "Duality"
    - An odd cycle is an obvious obstruction to 2-coloring
    - Odd cycles are the only obstructions to 2-coloring
- This theme is ubiquitous in algorithms
  - MaximumFlow/MinimumCut
  - BipartiteMatching/VertexCover
  - LinearProgramming
  - ZeroSumGames

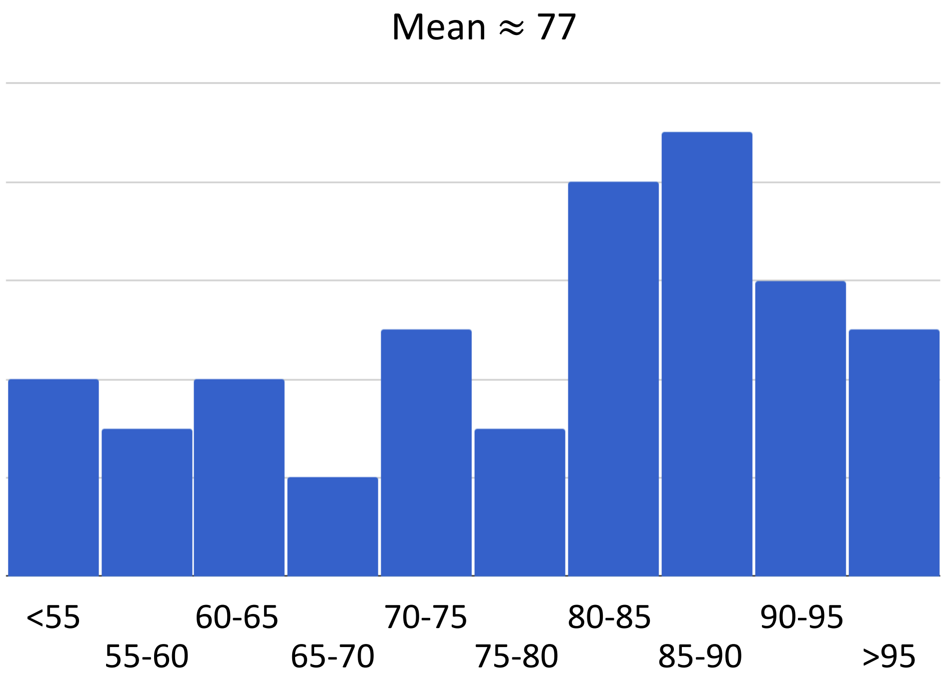
Midterms

Midterm Grade Distribution



- Letter grades are highly approximate
- Approximate letter grades consider MT1 only

HW Grade Distribution



- Chart does not reflect dropping the lowest HW