CS3000: Algorithms & Data
Jonathan Ullman

Lecture 12:
- Applications of BFS: 2-Coloring, Connected Components, Topological Sort

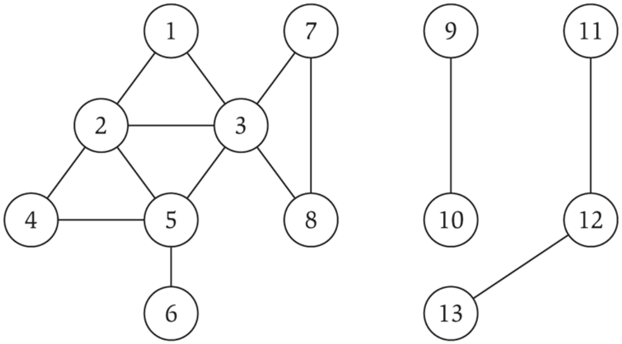
Oct 19, 2018

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"

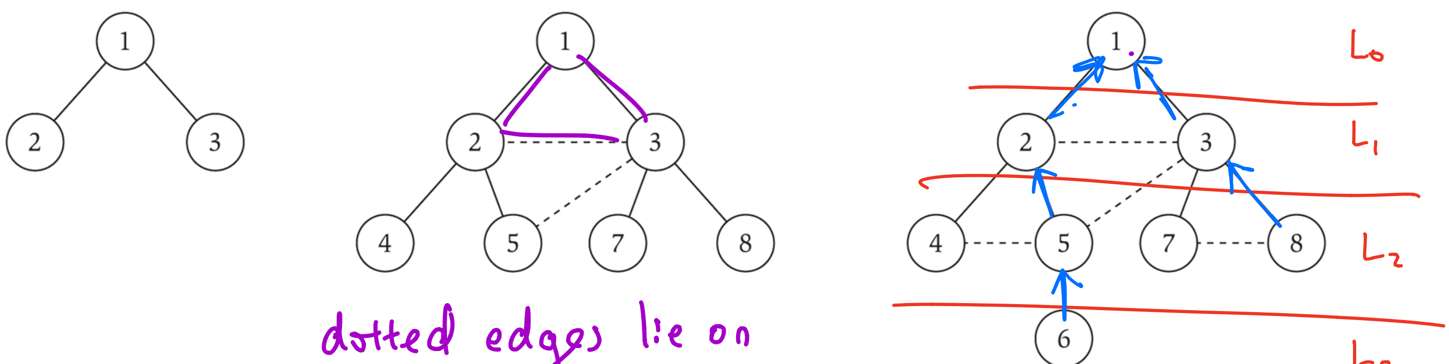
- **Simple Graph:**
  - No duplicate edges
  - No self-loops $e = (u, u)$



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

tree gives the shortest path



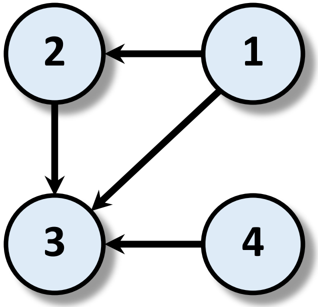
dotted edges lie on cycles

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



<u>Cost</u>

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. $(v, u) \in E$

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

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

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

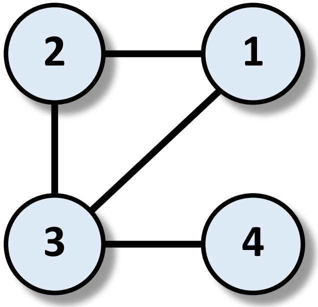
$A[4] = \{3\}$

<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

If n_s is the # of nodes reachable from s
m_s " " edges " "

$\Rightarrow$ time $O(n_s + m_s)$

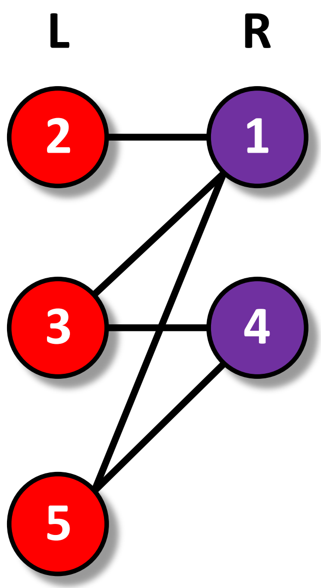
Bipartiteness / 2-Coloring

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

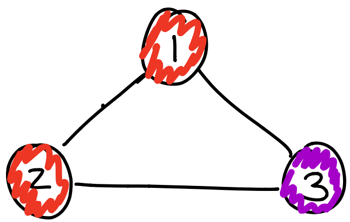


$L \cap R = \emptyset$

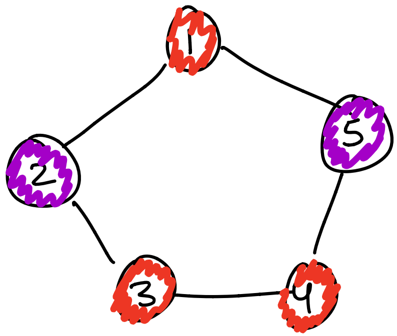
$L \cup R = V$

Designing the Algorithm

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



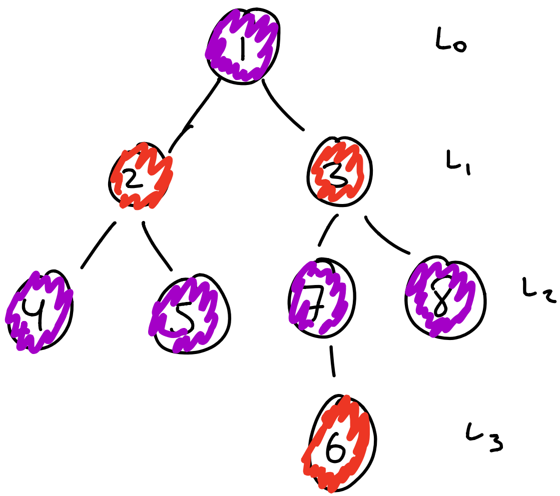
cant be 2-colored



cant be 2-colored

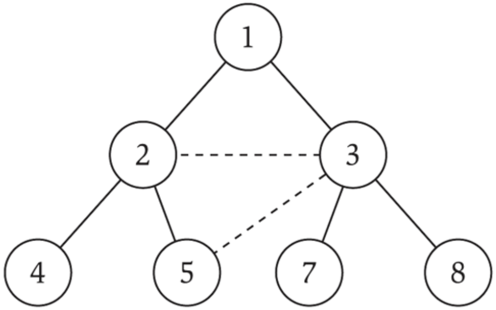
Designing the Algorithm

- **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



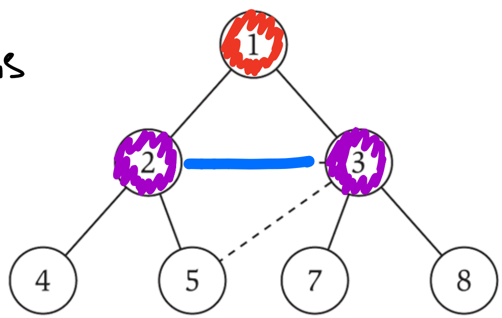
Designing the Algorithm

- **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**
- Every edge in the BFS tree is colored correctly
- Dotted edge from L_i to L_{i+1} are colored correctly
- If the 2-coloring is not correct, then there is an edge from L_i to L_i

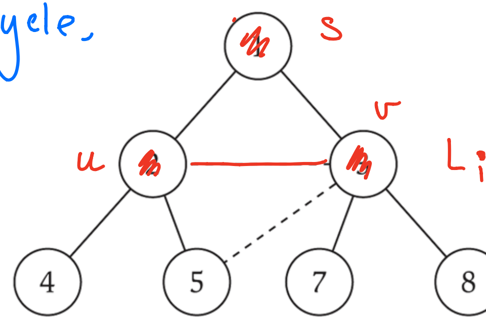


Designing the Algorithm

- **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**

$\boxed{\text{Clm:}}$ If G contains an edge from L_i to itself, then G contains an odd cycle.

- $\exists$ an $s \to u$ path of length i
- $\exists$ an $s \to v$ path of length i
- These paths meet at some node $w \in L_j$ for $j < i$



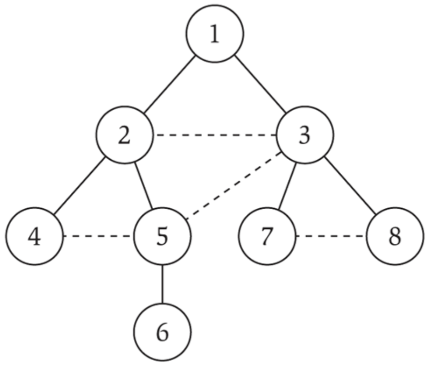
- $w \rightsquigarrow u \to v \rightsquigarrow w$ is an odd cycle

length $i-j$ 1 edge   length $i-j$ $\Rightarrow$ length $= 2(i-j)+1$

Topological Sort

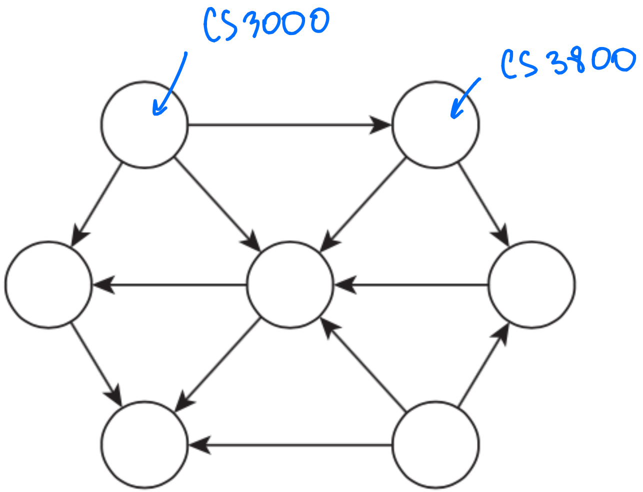
Acyclic Graphs

- **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

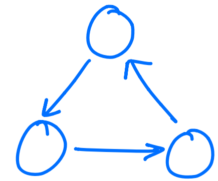


Directed Acyclic Graphs (DAGs)

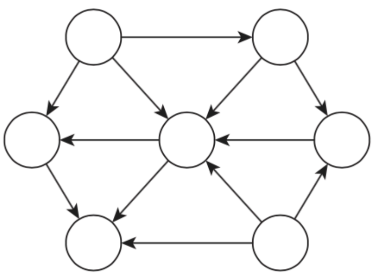
- **DAG:** A directed graph with no directed cycles
- Can be much more complex than a forest

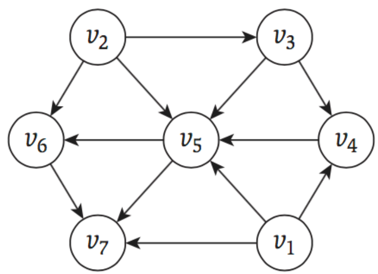


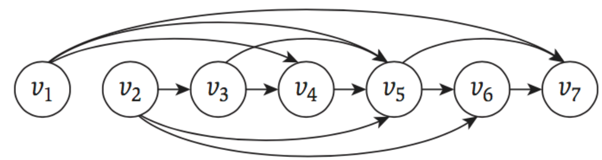
Directed Acyclic Graphs (DAGs)



- **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

G is <u>not</u> a DAG $\Rightarrow$ G <u>cannot</u> be top. ordered

Directed Acyclic Graphs (DAGs)

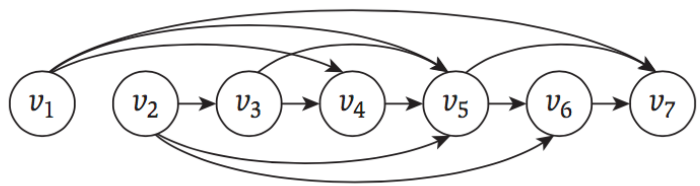
- **Problem 1:** given a digraph G, is it a DAG?
- **Problem 2:** given a digraph G, can it be topologically ordered?

- **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

If every node has ≥ 1 in-edge, then G can't be TO'd.

- **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 ≥ 1 in-edge

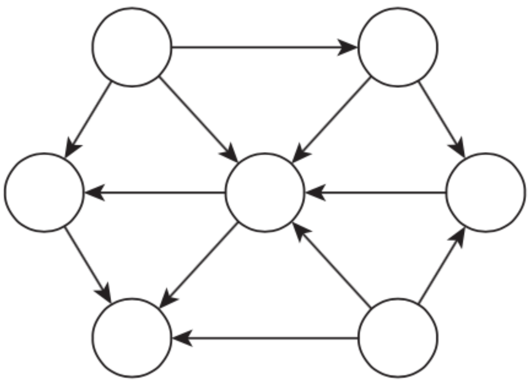
$$a \leftarrow b \leftarrow c \leftarrow d \neq a \;\dots\; \leftarrow v_i \leftarrow v_{i+1} \leftarrow \dots$$

- consider this chain of length $n+1$
- the same node must appear twice

- the node that appears twice starts and ends a directed cycle.

Topological Ordering

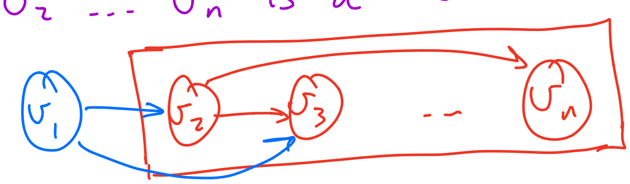
- **Fact:** In any DAG, there is a node with no incoming edges
- **Thm:** Every DAG has a topological ordering
- **Proof (Induction):** $H(n)$: $\forall$ DAG with n nodes, there exists a topological ordering.



- To prove: $\forall n \in \mathbb{N}$ $H(n)$ is true
- Base Case: $H(1)$ is true

Inductive Step:

- To prove : $H(n-1) \Rightarrow H(n)$
- By [Fact] there exists a node w/ no incoming edges, call it v_1
- Consider the graph $G \setminus \{v_1\}$, this graph is a DAG w/ $n-1$ nodes
- By $H(n-1)$, there exists an ordering of $G \setminus \{v_1\}$, call it $v_2, v_3, \ldots, v_n$
- $v_1, v_2 \ldots v_n$ is a TO of G

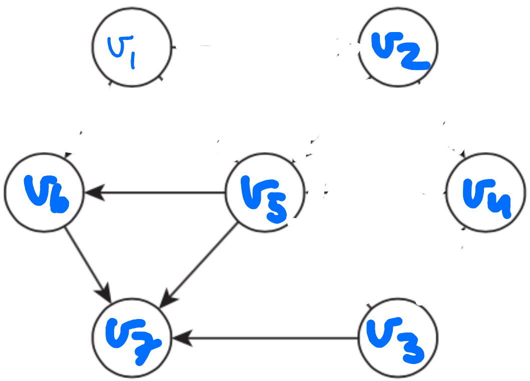


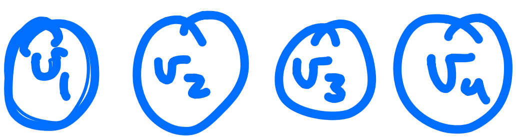
By induction, all edges in the box go left to right

There are no in-edges, so all edges go left to right

Implementing Topological Ordering

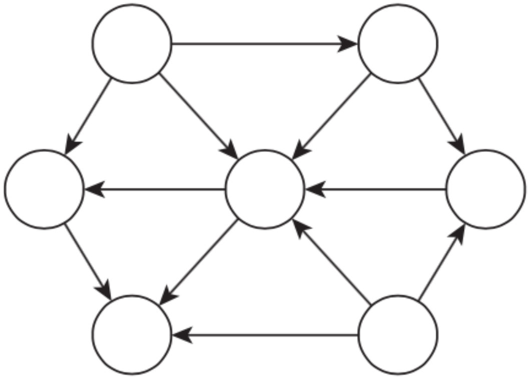
```
SimpleTopOrder(G):
  Set i ← 1
  Until (G has no more nodes):
 ① Find a node u with no incoming edges
 ② Label u as node i, increment i ← i+1
 ③ Remove u and its edges from G
```





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

```
SimpleTopOrder(G):
  Set i ← 1
  Until (G has no more nodes):
  ① Find a node u with no incoming edges
  ② Label u as node i, increment i ← i+1
  ③ Remove u and its edges from G
```

- Go around the loop n times
- Step 1 takes $O(n)$ time
- Step 2 takes $O(1)$ time
- Step 3 takes $O(m)$ time

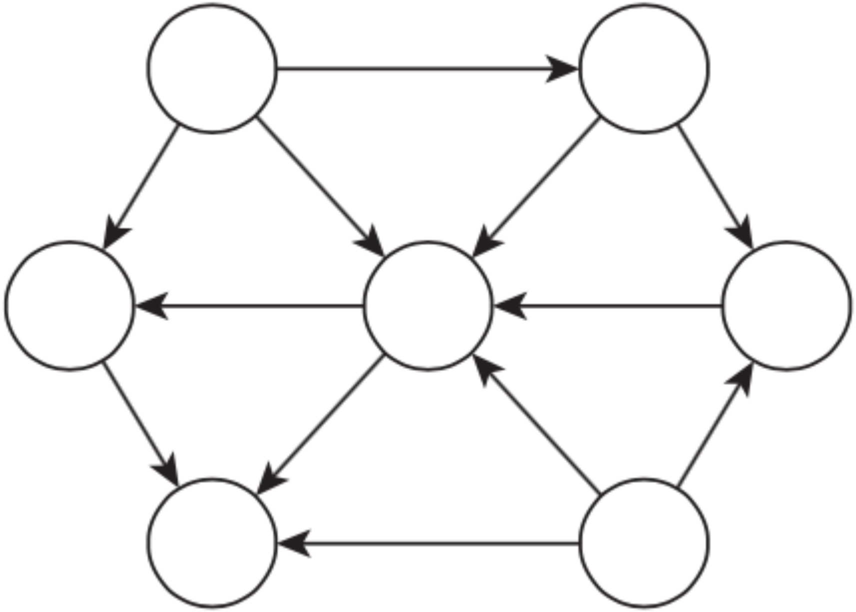
Usually assume $m \geq n-1$

$$n \times O(n+m) = O(n^2+nm) = O(nm)$$

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 $O(n + m)$