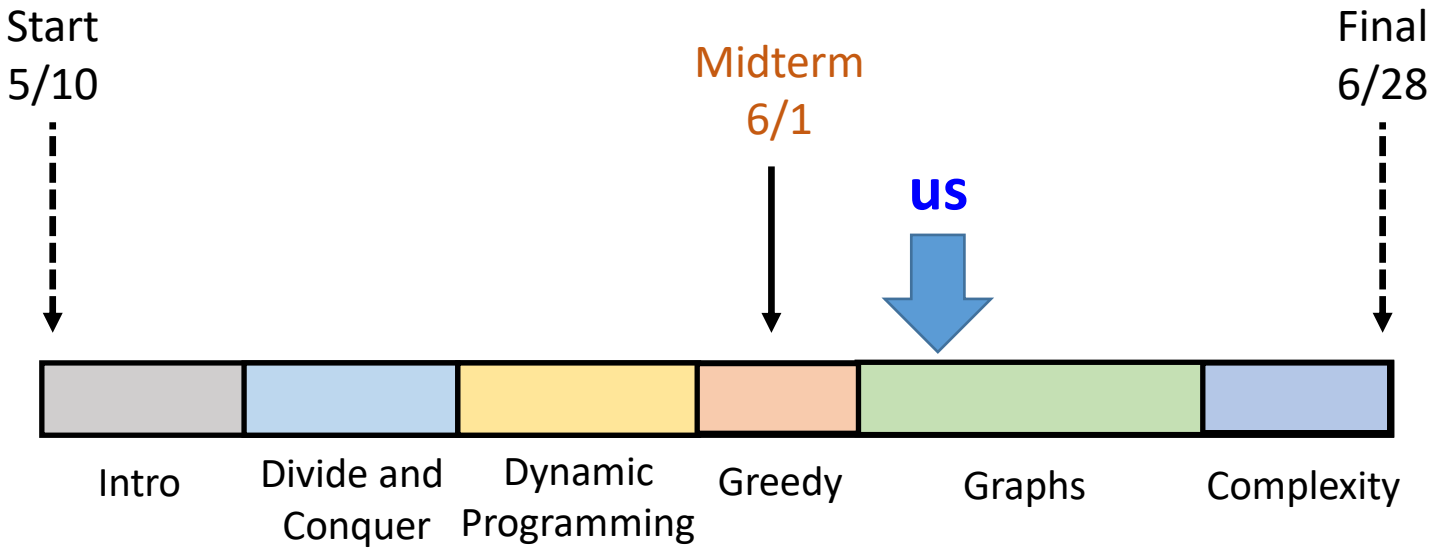
HW 3 due tonight!

CS3000: Algorithms & Data
Drew van der Poel

Lecture 15
- Graph Traversals: DFS
- 2-coloring
- Topological Sorting

June 7, 2021

Outline



Last class: Graphs: BFS, terminology

Next class: Graphs: strongly connected components

BFS Running Time (Adjacency List)

```
BFS(G = (V,E), s):
  Let found[v] ← false ∀v, found[s] ← true
  Let layer[v] ← ∞ ∀v, layer[s] ← 0
  Let i ← 0, L_0 = {s}, T ← ∅

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

O(n)

O(1)

O(n) times in total

deg(u)

Adj. list of node u

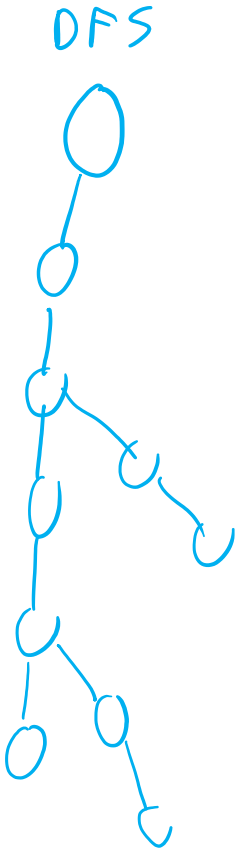
O(1)

O(1)

$$\sum_{u \in V} \deg(u) = O(m)$$

Total: $O(n+m)$

Depth-First Search (DFS)



Exploring a Graph

- **Problem:** Is there a path from s to t?
- **Idea:** Explore all nodes reachable from s.

- Two different search techniques:
  - **Breadth-First Search:** explore nearby nodes before moving on to farther away nodes
  - **Depth-First Search:** follow a path until you get stuck, then go back

Depth-First Search

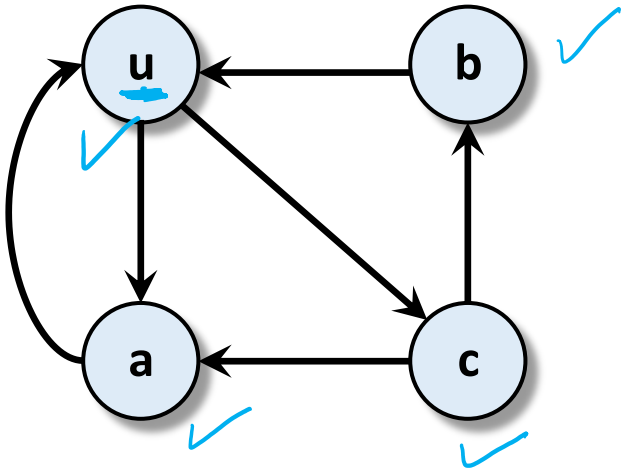
$A_{out}[u]$ (adj. list)

```
G = (V,E) is a graph
explored[u] = 0 ∀u

Let u be a vertex in V
parent[u] = null

DFS(u)

DFS(u):
  explored[u] = 1
  for (v in OUT[u]):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)
Return parent
```

O(n) O(n) O(1) O(n) O(1) (A) (*) O(1)

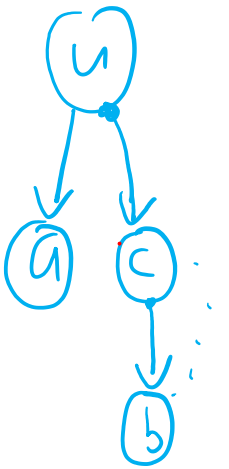
≤ n times, each node is explored ≤ 1 time

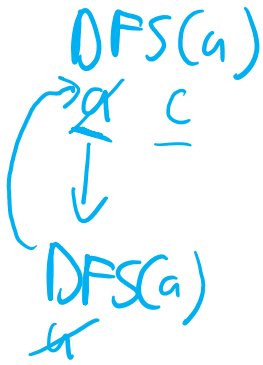
each: Out-deg(u) times, O(m) in total



Parent

u	(null)
a	u
c	u
b	c





$(*) \leq \sum_{u \in V} \text{Out-deg}(u) = m = O(m)$

TOTAL R/T: $O(n+m)$

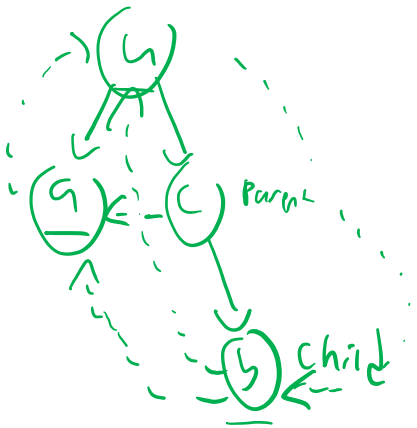
- Problems: counting students, stable matching, sorting, n-digit mulitiplication, array searching, selection, weighted interval scheduling, segmented least squares, knapsack, prefix-free encoding, graph **exploration**

- Alg. techniques: divide & conquer, dynamic programming, greedy

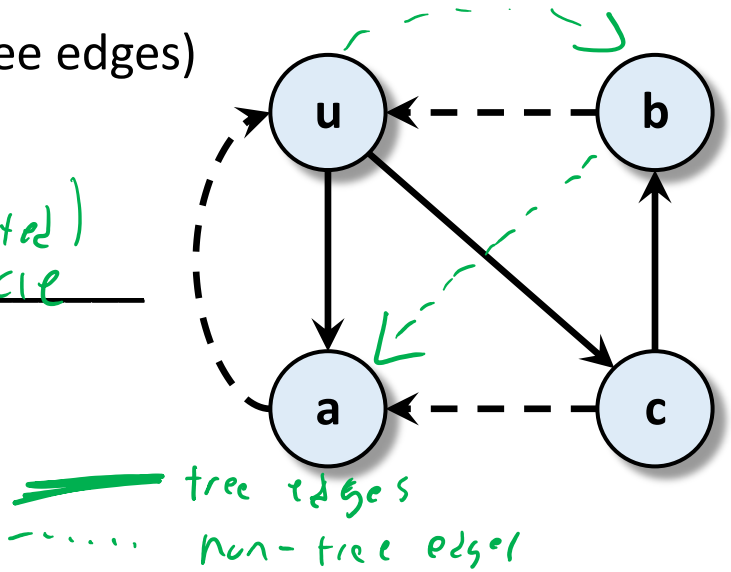
- Analysis: asymptotic analysis, recursion trees, Master Thm., Graph Terminology/representations

- Proof techniques: (strong) induction, contradiction, greedy stays ahead, exchange argument

Depth-First Search

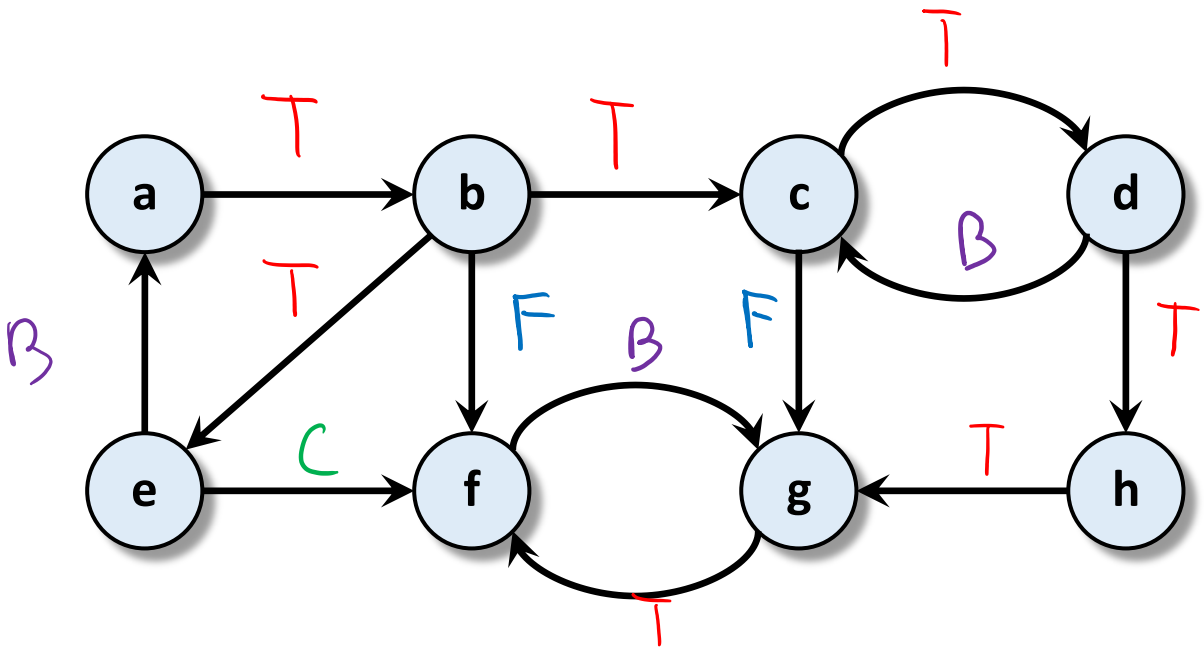


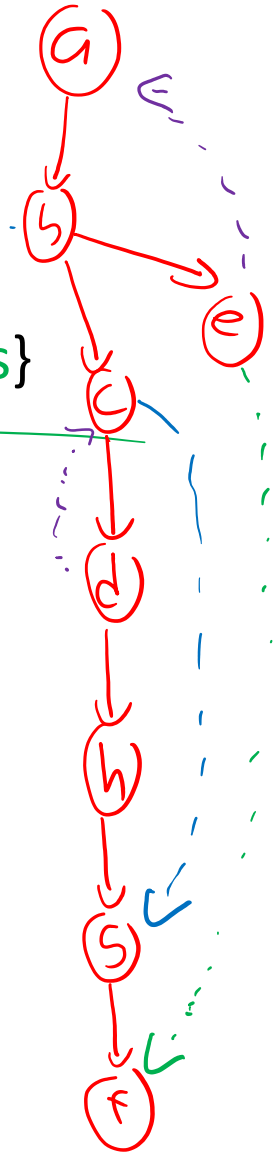
- **Fact:** The parent-child edges form a (directed) tree
- **Each edge has a type:**
  - **Tree edges (parent-child edges):** $(u, a), (u, c), (c, b)$
    - These are the edges that explore new nodes
  - **Forward edges:** (u, b)
    - Ancestor to descendant (excl. tree edges)
  - **Backward edges:** $(a, u), (b, u)$
    - Descendant to ancestor
    - Every back edge lies on a (directed) cycle
  - **Cross edges:** (c, a), (b, a)
    - No ancestral relation



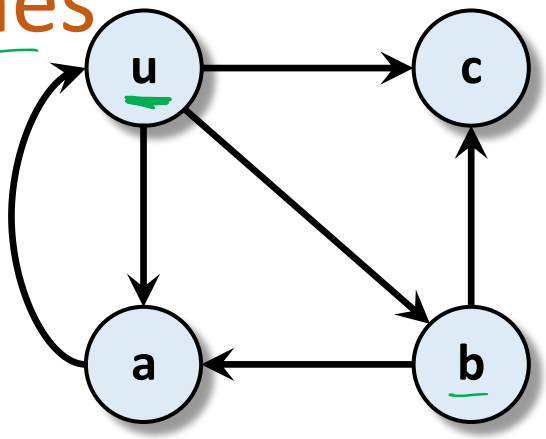
Ask the Audience

- DFS this graph starting from node a
  - Search in alphabetical order
  - Label edges as { tree , forward , backward , cross}





Discovery and Finish Times



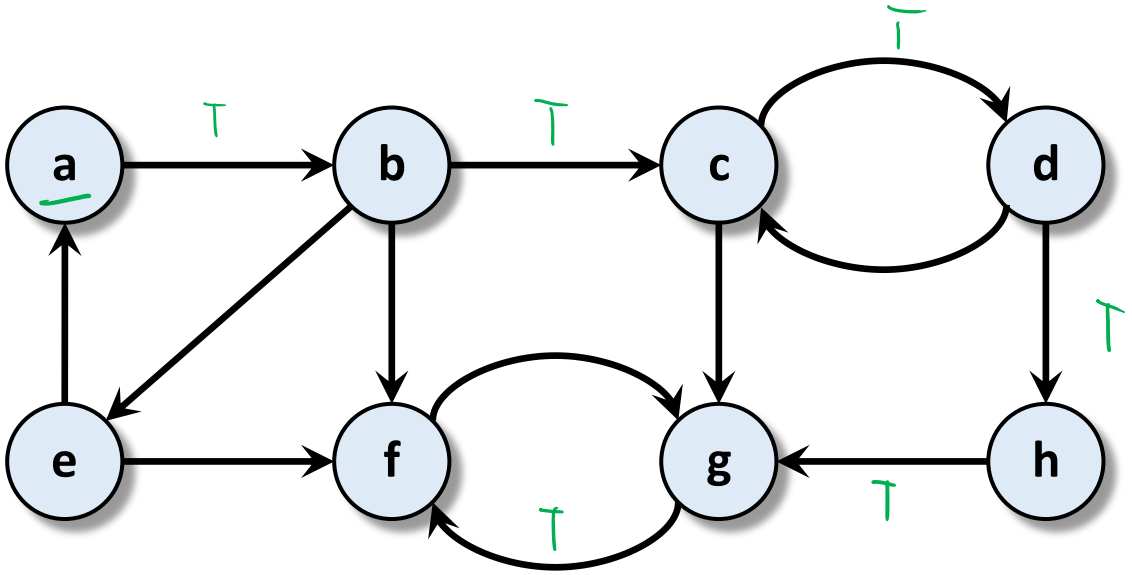

```
G = (V,E) is a graph
discovered[u] = 0 ∀u

DFS(u):
  discovered[u] = 1
  d[u] = clock, clock++

  for (v in OUT[u]):
    if (discovered[v]=0):
      parent[v] = u
      DFS(v)

  f[u] = clock, clock++
```

Vertex	Discovery	Finish
u	1	8
a	2	3
b	4	7
c	5	6

- Maintain a counter **`clock`**, initially set **`clock = 1`**

Ask the Audience

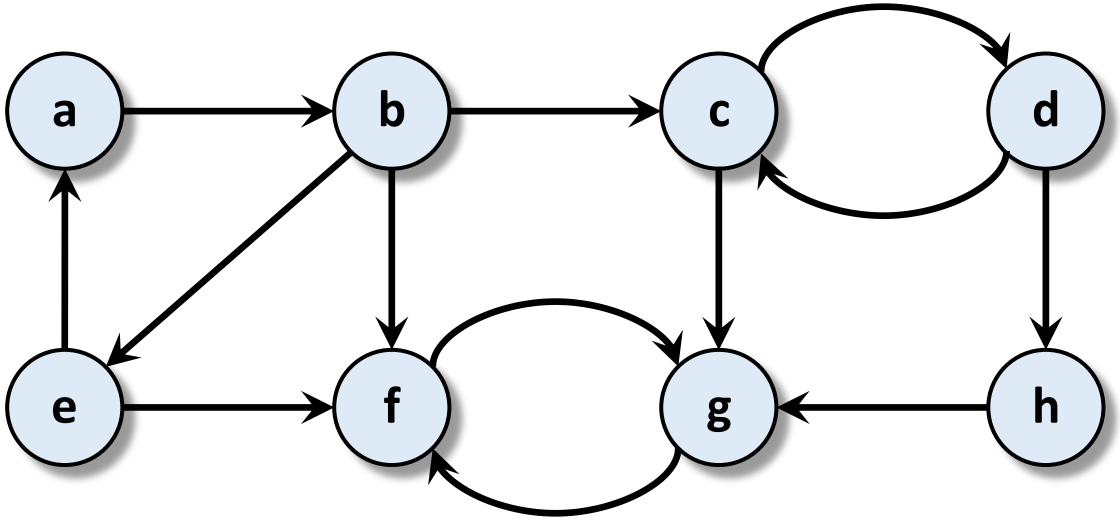
- Compute the **discovery and** finish times for this graph
  - DFS from a, search in alphabetical order



Vertex	a	b	c	d	e	f	g	h
Discovery d[]	1	2	3	4	13	7	6	5
Finish f[]	16	15	12	11	14	8	9	10

Ask the Audience

- Compute the **discovery and** finish times for this graph
  - DFS from a, search in alphabetical order

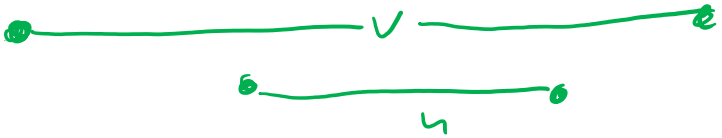


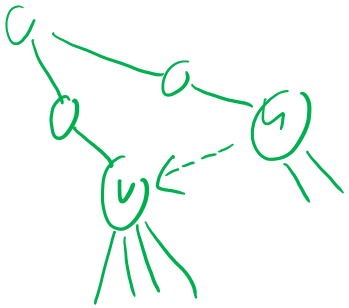
Vertex	a	b	c	d	e	f	g	h
Discovery d[]	1	2	3	4	13	7	6	5
Finish f[]	16	15	12	11	14	8	9	10

Discovery & Finish Times and Edge Types

- For any (directed) edge (u, v):
  - $d[u] < d[v] < f[v] < f[u] \Rightarrow$ tree or forward

  - $d[v] < d[u] < f[u] < f[v] \Rightarrow$ back



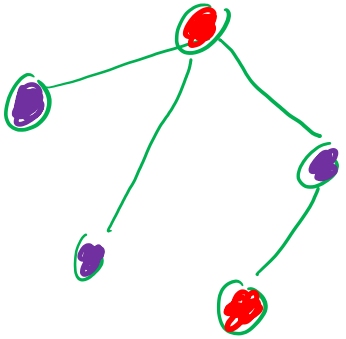


  - $d[v] < f[v] < d[u] < f[u] \Rightarrow$ cross

  - $d[u] < f[u] < d[v] < f[v] \Rightarrow$ **cannot happen**
    - **u cannot finish before v if d[u] < d[v]**

Bipartiteness / 2-Coloring

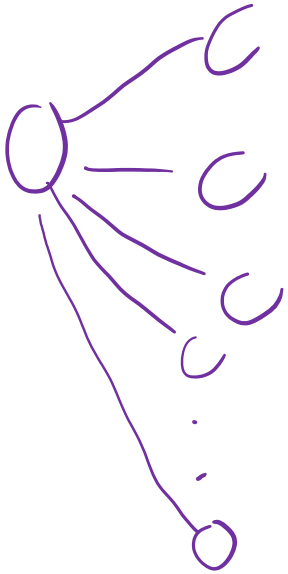
2-Coloring

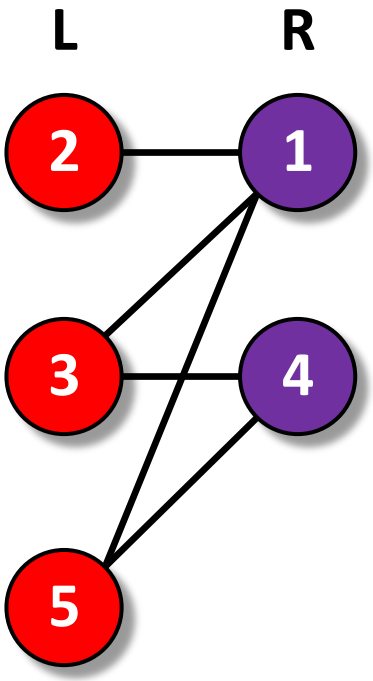


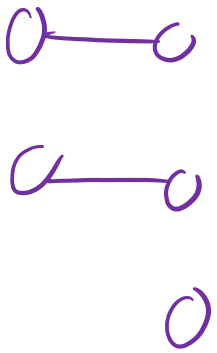
- **Problem:** Tug-of-War
  - Need to form two teams $\boldsymbol{R}$, $\boldsymbol{P}$
  - Some students just don't get along
- **Input:** Undirected graph $G = (V, E)$
  - $(u, v) \in E$ means u, v won't be on the same team
- **Output:** Split V into two sets $\boldsymbol{R}$, $\boldsymbol{P}$ so that 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



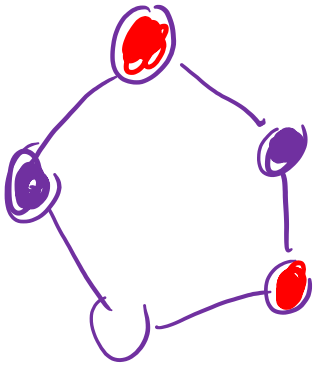


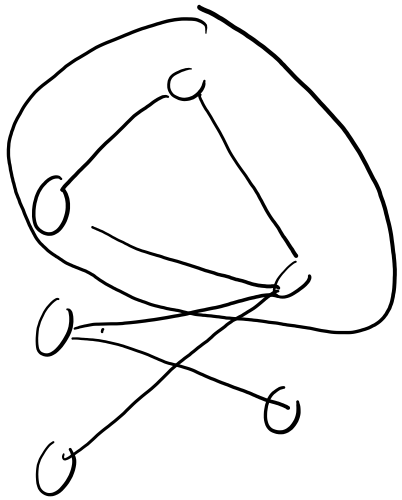


- Problems: counting students, stable matching, sorting, n-digit mulitiplication, array searching, selection, weighted interval scheduling, segmented least squares, knapsack, prefix-free encoding, graph exploration, **bipartiteness**
- Alg. techniques: divide & conquer, dynamic programming, greedy
- Analysis: asymptotic analysis, recursion trees, Master Thm., Graph Terminology/representations
- Proof techniques: (strong) induction, contradiction, greedy stays ahead, exchange argument

Key Fact

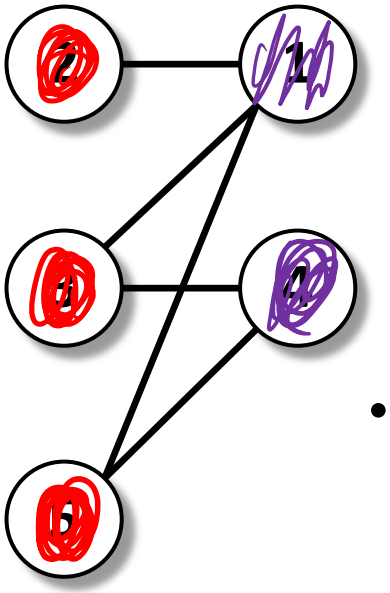
- **Key Fact:** If G contains a cycle of odd length, then G is not 2-colorable/bipartite



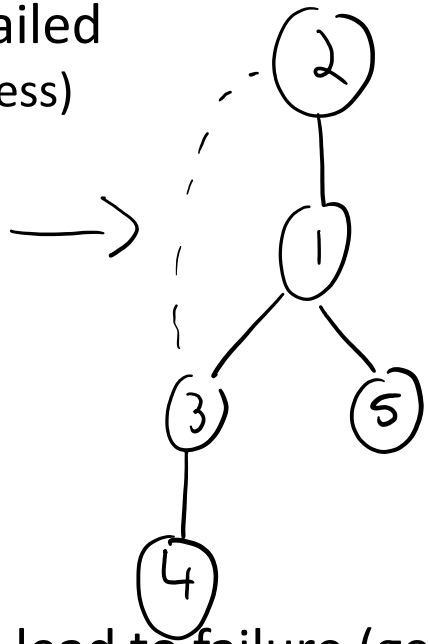


Designing the Algorithm

- **Idea for the algorithm:**
  - BFS the graph, coloring nodes as you find them
  - Color nodes in layer i **purple** if i ~~even~~ odd, **red** if i ~~odd~~ even
  - See if you have succeeded or failed
    - (all edges have 2 colors -> success)



don't need to worry about (can't happen)



Red L_0

Purple L_1

Red L_2

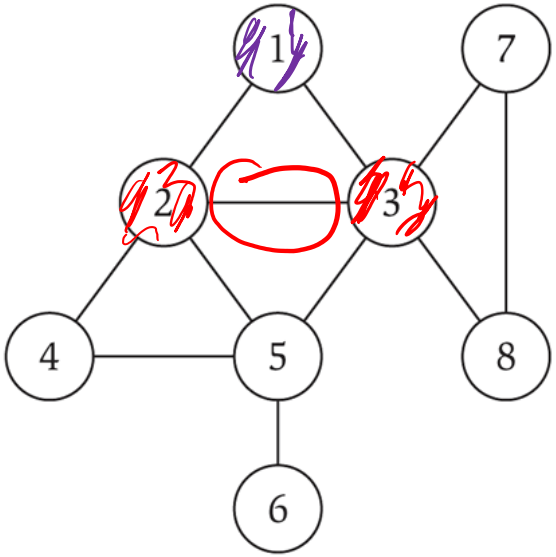
Purple L_3

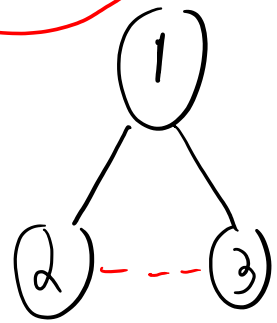
- What edges would lead to failure (generally)?

edges between nodes on same layer

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





L_0 Purple

L_1 Red

No, not bipartite!

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? (i.e. is there a chance of getting a false negative?)

No!

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!

Pf. $\exists$ (u, v) s.t. u and v are on the same layer in the BFS tree

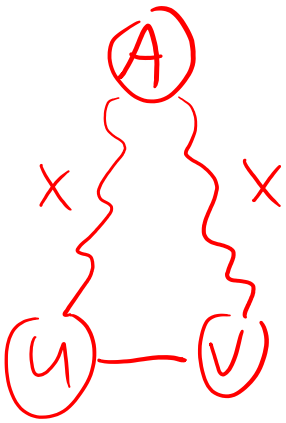
Let A be the closest common ancestor of u and v.

$dist_{BFS}(A, u) = dist_{BFS}(A, v) = x$

$\therefore \exists$ cycle

Of length $2x+1$

$\therefore$ $\exists$ odd length cycle $\square$

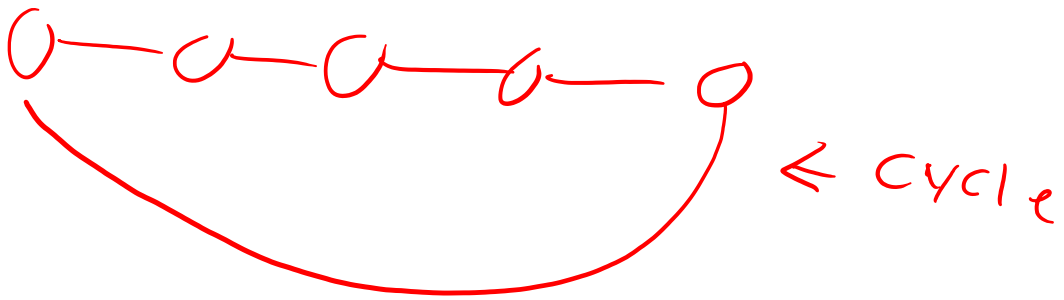


L;

Topological Sort

Acyclic Graphs

- **Acyclic Graph:** has no cycles



Acyclic Graphs

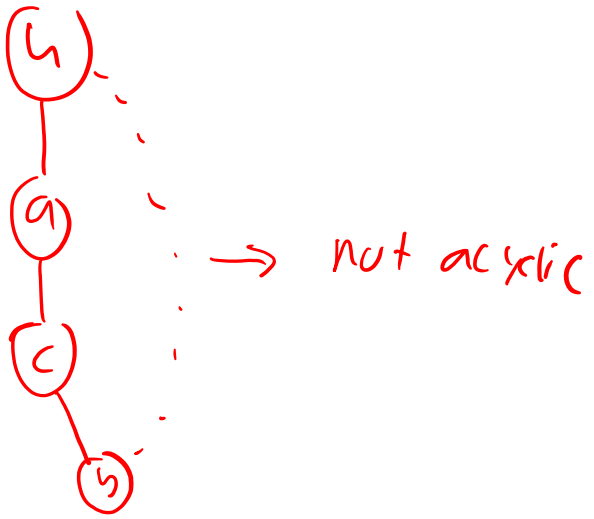
- **Acyclic Graph:** An undirected graph with no cycles
  - Also known as a forest
  - If it's connected then it's known as a tree
- Can test if a graph has a cycle in $O(n + m)$ time
  - How?

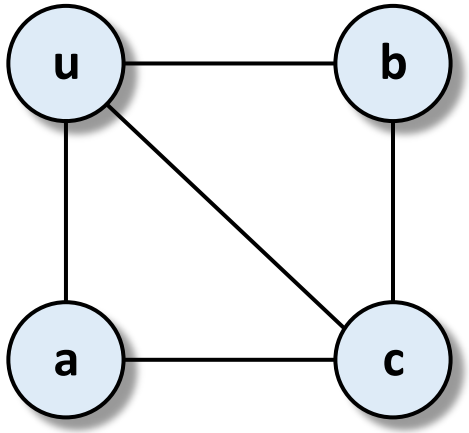
run DFS

Check for back edges

Acyclic Graphs

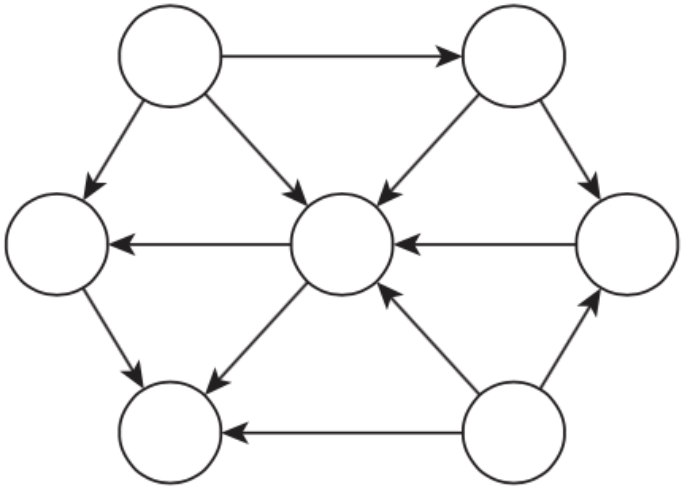
- **Acyclic Graph:** An undirected graph with no cycles
  - Also known as a forest
  - If it's connected then it's known as a tree
- Can test if a graph has a cycle in $O(n + m)$ time
  - Run DFS
  - If there are any edges **back edges**, then they form a cycle





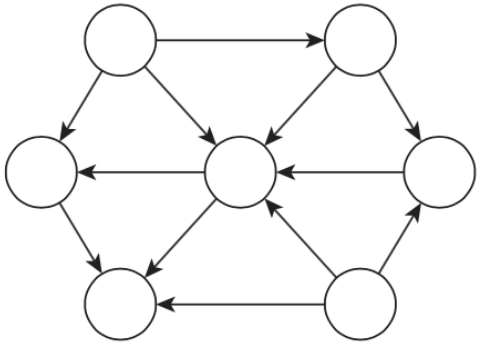
Directed Acyclic Graphs (DAGs)

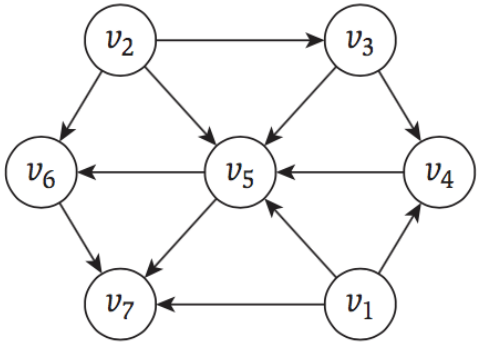
- **DAG:** A directed graph with no directed cycles
- Can be much more complex than a forest

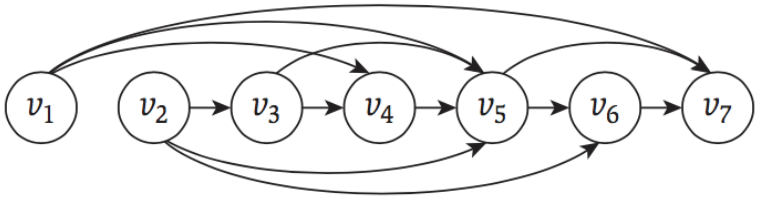


Directed Acyclic Graphs (DAGs)

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





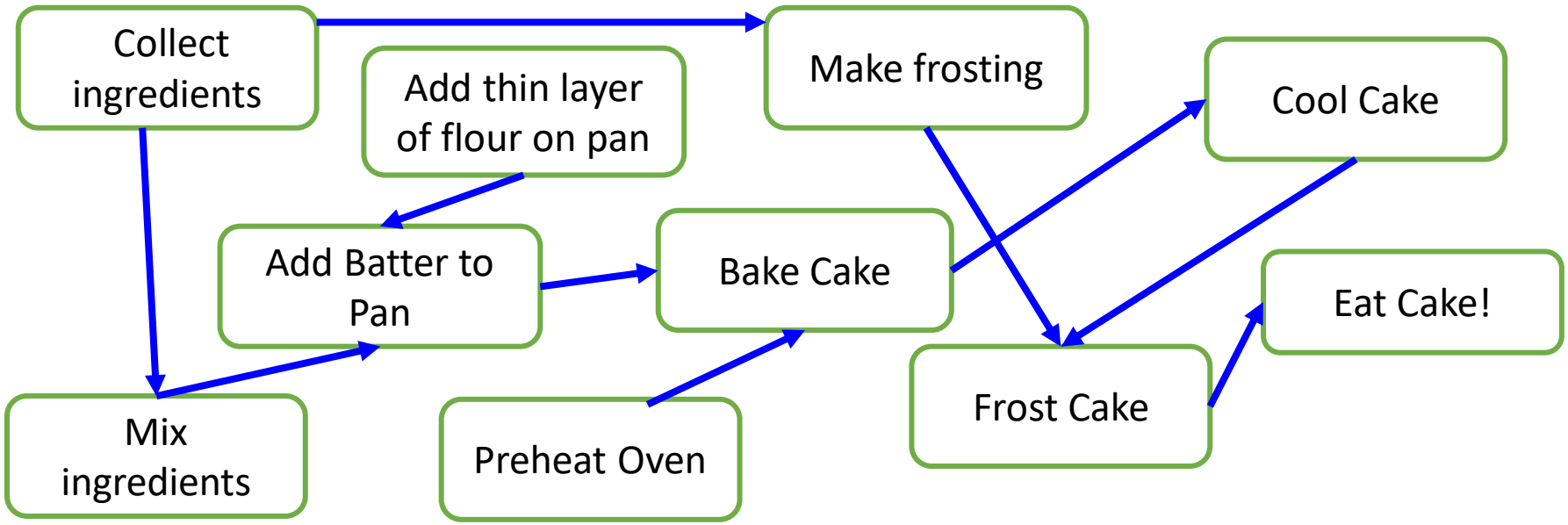


- A topological ordering of a directed graph is a labeling of the nodes from $v_1, \ldots, v_n$ so that all edges go "forwards", that is $(v_i, v_j) \in E \Rightarrow j > i$
  - G has a topological ordering $\Rightarrow$ G is a DAG

**G cannot be top. ordered if _____

Directed Acyclic Graphs (DAGs)

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



- Problems: counting students, stable matching, sorting, n-digit mulitiplication, array searching, selection, weighted interval scheduling, segmented least squares, knapsack, prefix-free encoding, graph exploration, bipartiteness, **topological sorting**

- Alg. techniques: divide & conquer, dynamic programming, greedy

- Analysis: asymptotic analysis, recursion trees, Master Thm., Graph Terminology/representations

- Proof techniques: (strong) induction, contradiction, greedy stays ahead, exchange argument

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?

- **The answers to P1 and P2 are:**
  - Always the same
  - Sometimes different

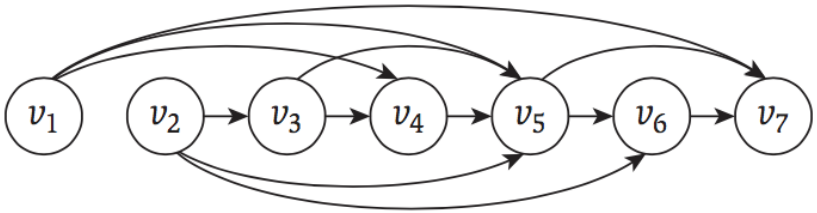
Directed Acyclic Graphs (DAGs)

- **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

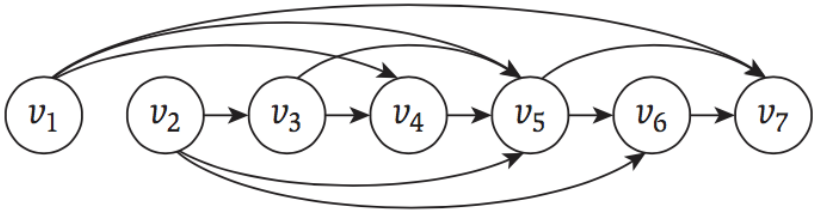
Topological Ordering

- **What can we say about the first node in the top. ordering?**



Topological Ordering

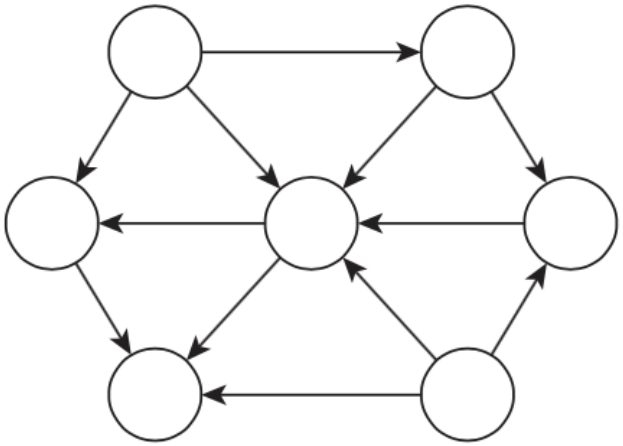
- **Observation:** the first node must have no in-edges



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

Topological Ordering

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



Topological Ordering

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