Graphs and Graph Traversals

a. Introduction to Graphs
b. Graph Traversals: DFS
c. Topological Ordering
d. Breadth-First Search

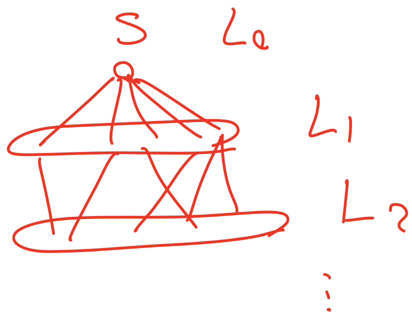
Exploring a Graph

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

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

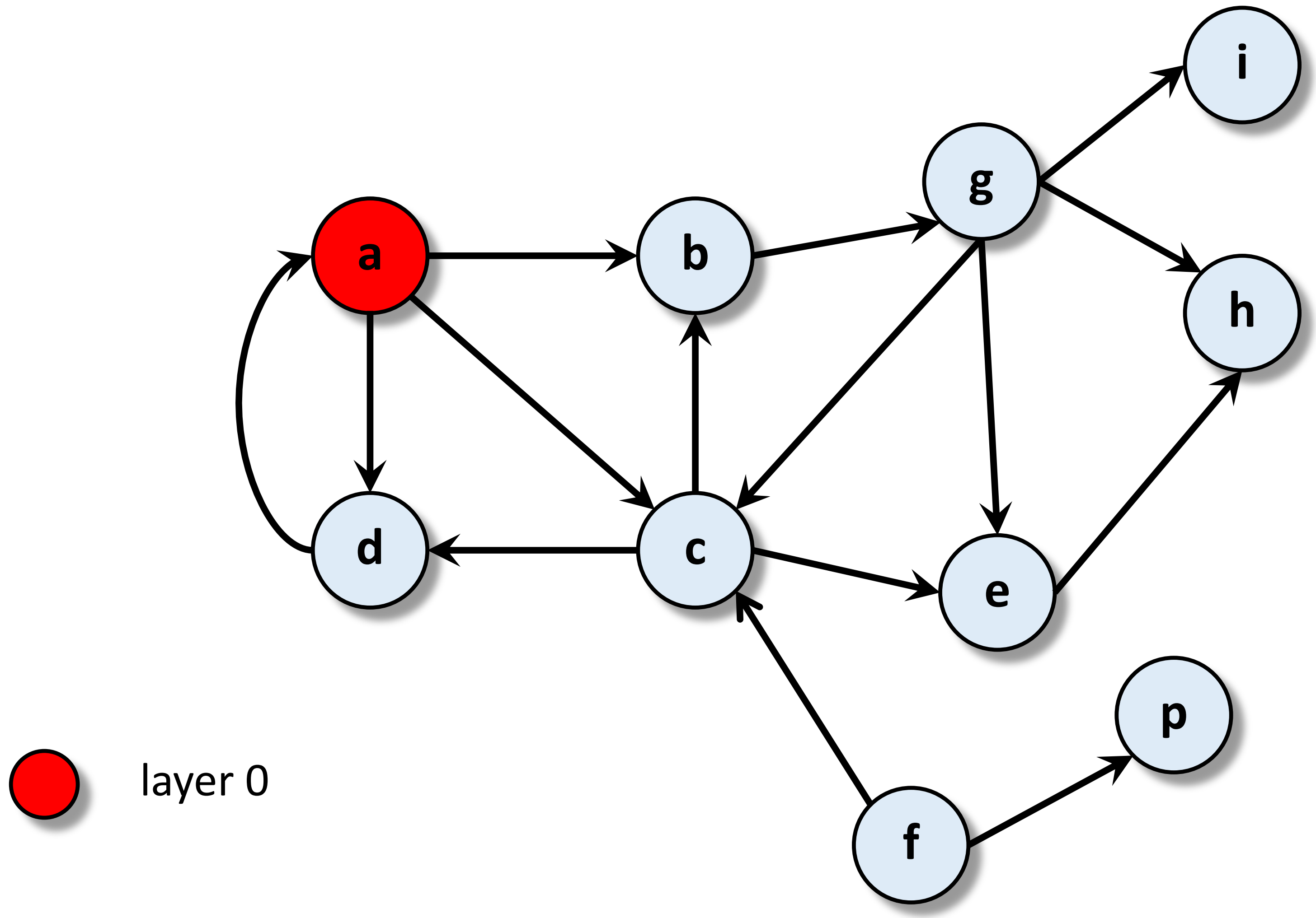
Breadth-First Search (BFS)

- **Informal Description:** start at s, find neighbors of s, find neighbors of neighbors of s, and so on...

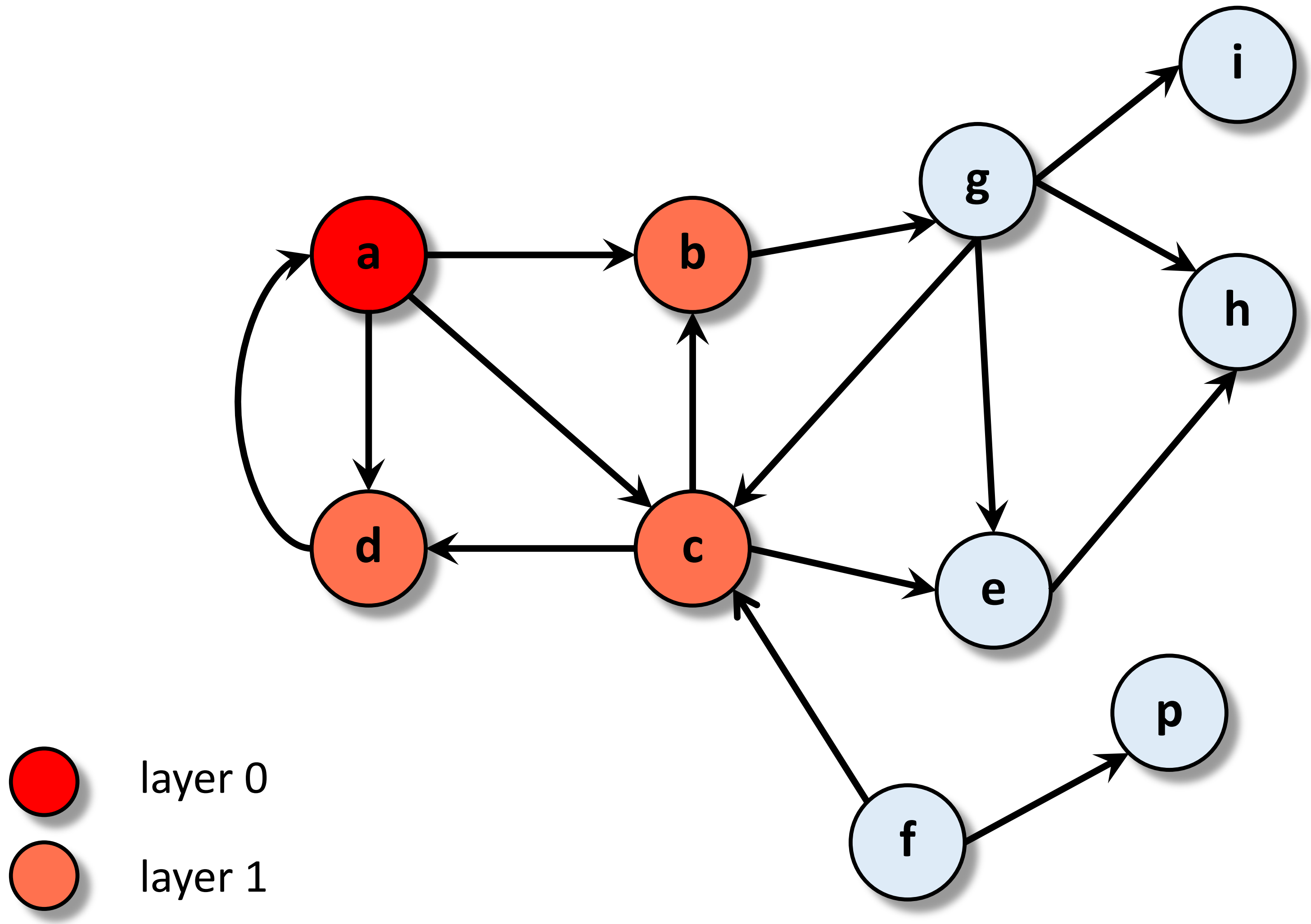


- BFS Tree:
  - $L_0 = \{s\}$
  - $L_1 =$ all neighbors of L_0
  - $L_2 =$ all neighbors of L_1 that are not in L_0, L_1
  - $L_3 =$ all neighbors of L_2 that are not in L_0, L_1, L_2
  - …
  - $L_d =$ all neighbors of L_{d-1} that are not in $L_0, \dots, L_{d-1}$
  - Stop when L_{d+1} is empty

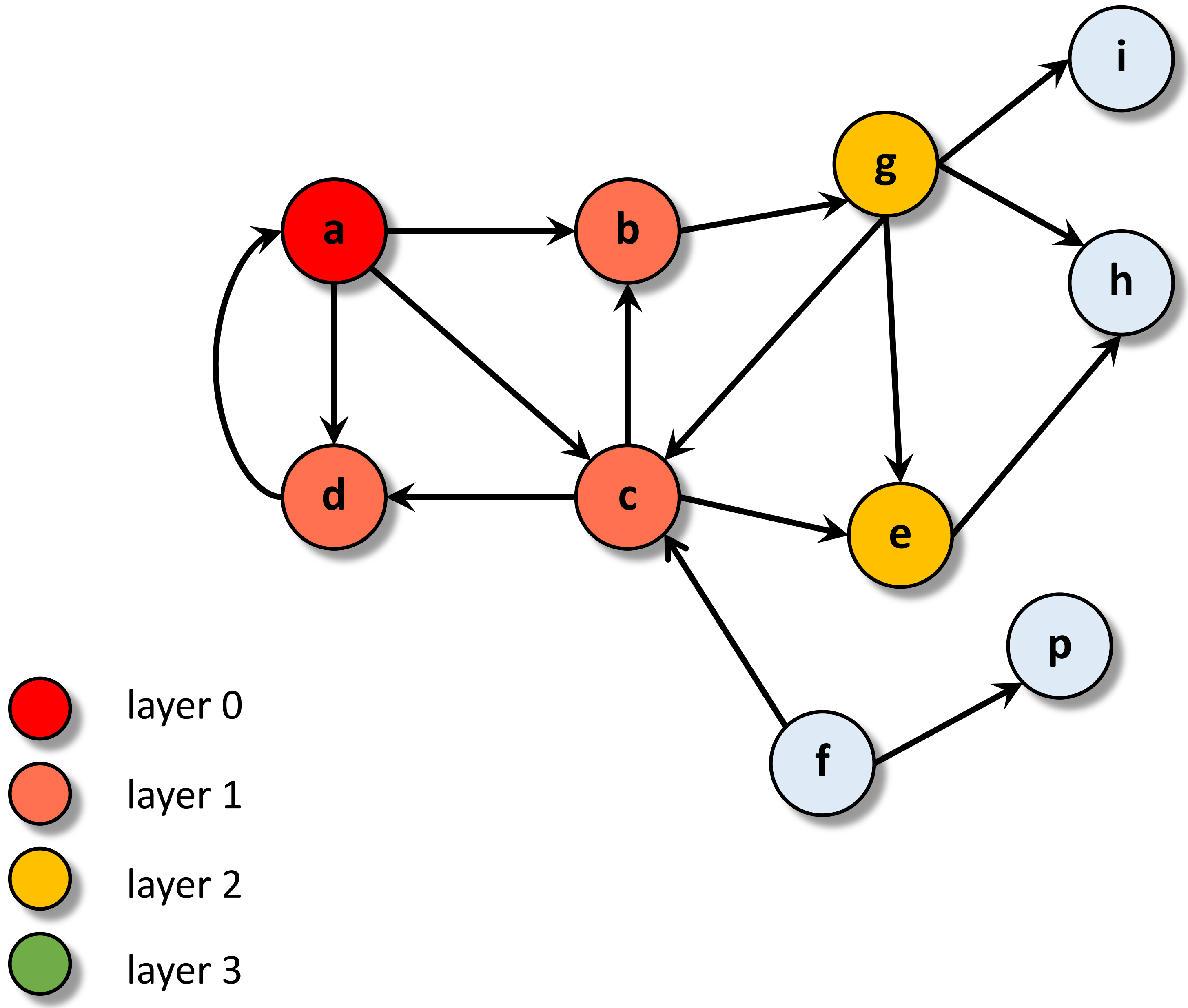
Breadth-First Search in Directed Graphs



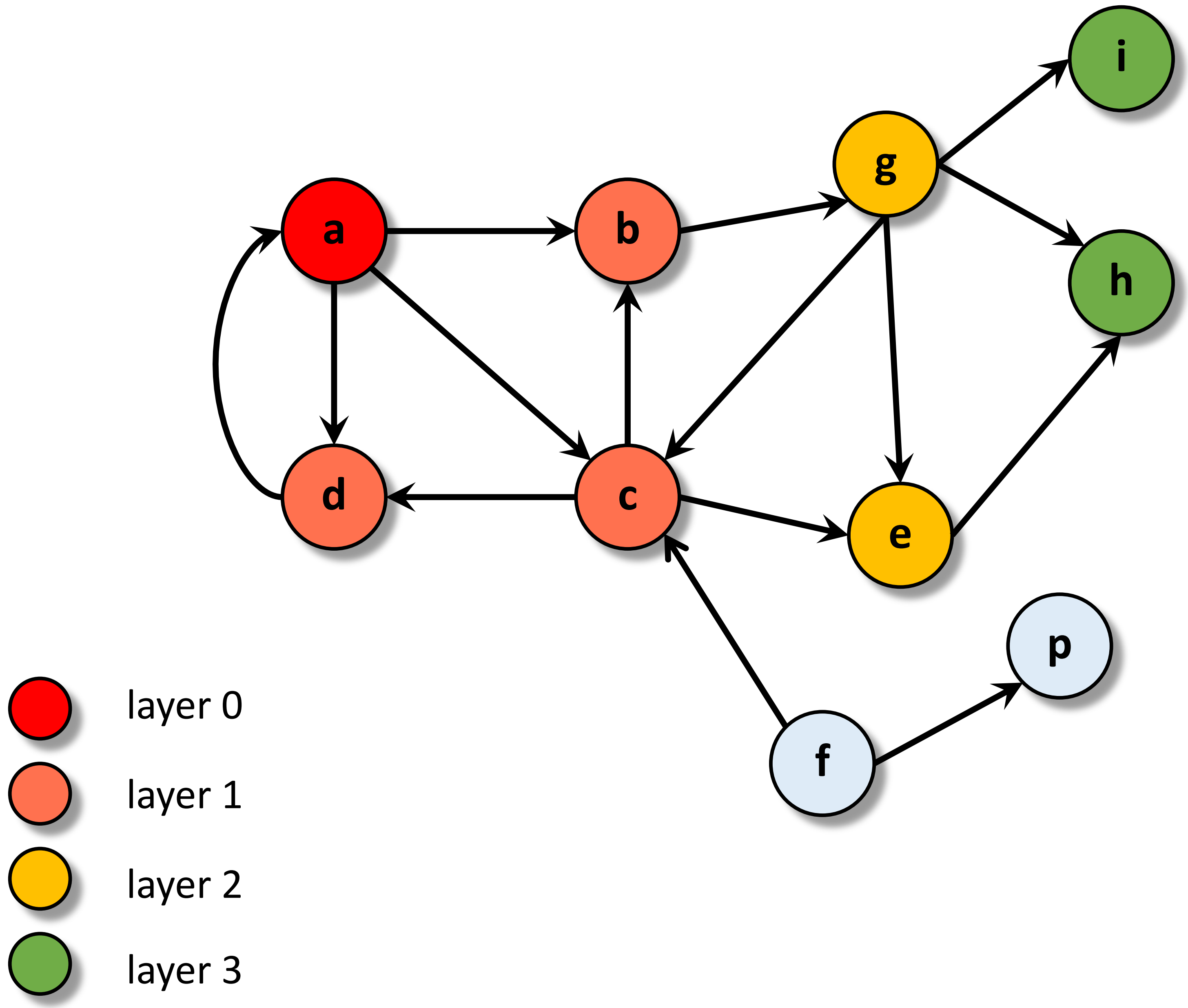
Breadth-First Search in Directed Graphs



Breadth-First Search in Directed Graphs

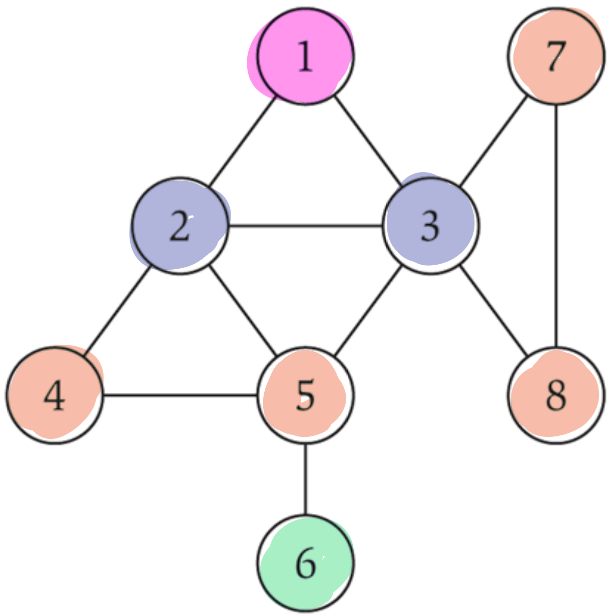


Breadth-First Search in Directed Graphs



Ask the Audience

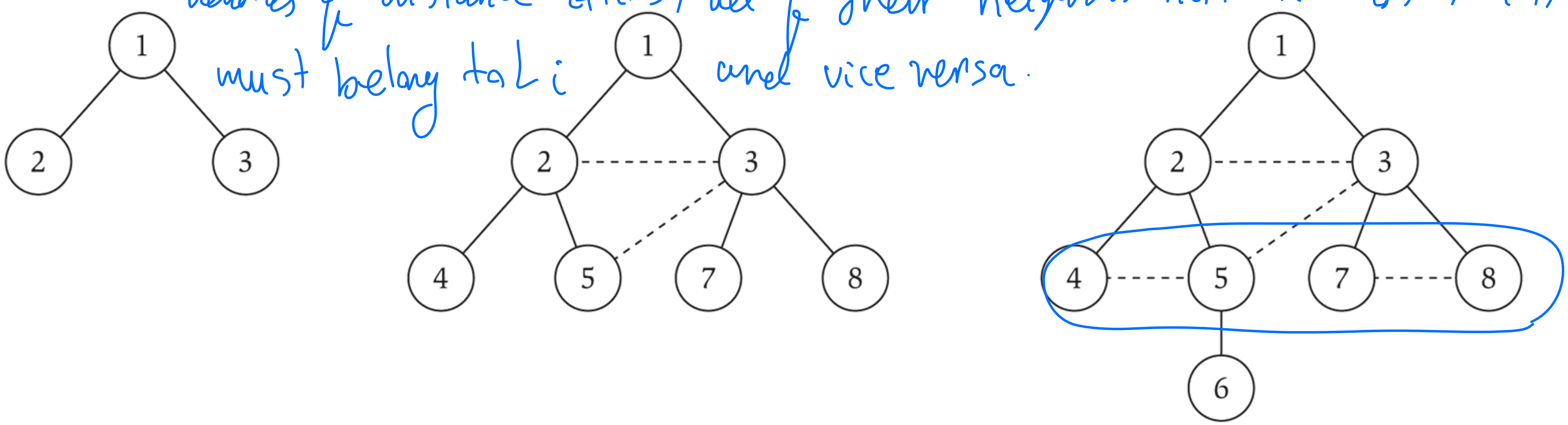
- BFS this graph from $s = 1$



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

Proof by induction on i. Base case $i=0$. Assuming that L_{i-1} includes all vertices of distance $i-1$ to s, all of their neighbors not in $L_0, \dots, L_{i-1}$, must belong to L_i and vice versa.



L_{i-1}

BFS Implementation (Adjacency List)

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

  layer[s] = 0
  discovered[s] = true

  while (L_i is not empty):
    Initialize new layer L_{i+1}
    For (u in L_i):
      For ((u,v) in E):
        If (discovered[v] = false):
          discovered[v]= true,
          layer[v] = i+1
          parent[v] = u
          Add v to L_{i+1}
    i= i+1
```

Graphs and Graph Traversals

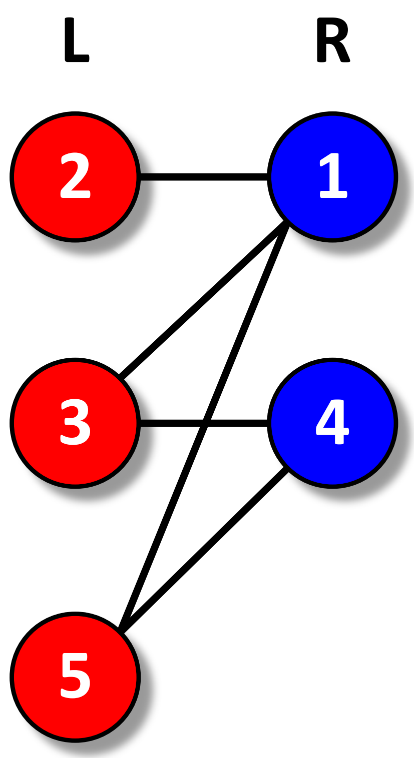
a. Introduction to Graphs
b. Graph Traversals: DFS
c. Topological Ordering
d. Breadth-First Search
e. Bipartite Graphs and Graph Traversals Recap

Practice Problem: 2-Coloring

- **Problem:** Tug-of-War
  - Need to form two teams $\boldsymbol{R}$, $\boldsymbol{B}$
  - Some students just don't get along
- **Input:** Undirected graph $G = (V, E)$
  - $(u, v) \in E$ means u, v will not be on the same team
- **Output:** Split V into two sets $\boldsymbol{R}$, $\boldsymbol{B}$ so that no pair in either set is connected by an edge

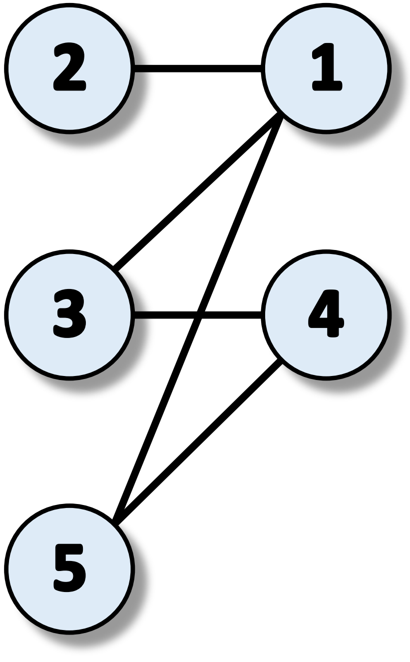

2-Coloring (Bipartiteness)

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



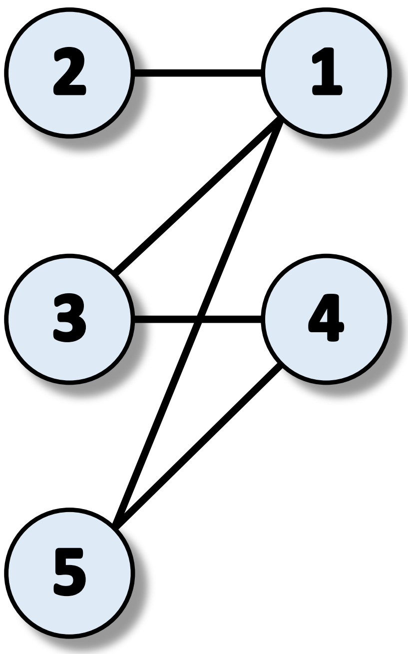
Designing the Algorithm

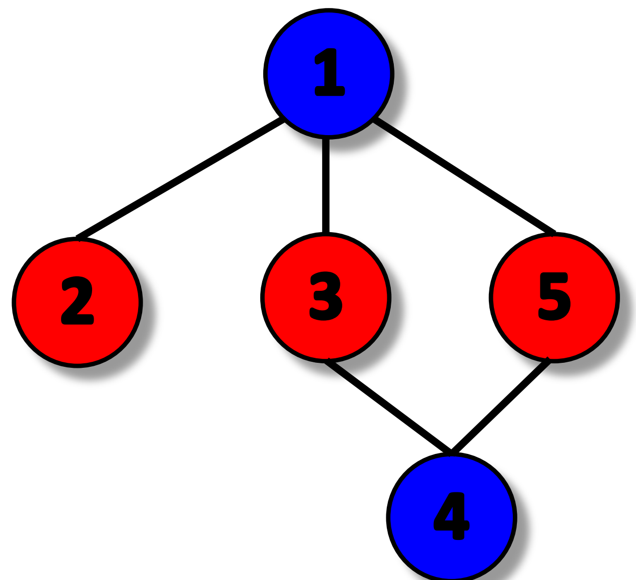
- **Idea for the algorithm:**
  - BFS the graph, coloring nodes as you find them
  - Color nodes in layer i **blue** if i even, **red** if i odd
  - Go over all edges and check if their endpoints have received different colors.



BFS 2-Coloring Success Implies Bipartite

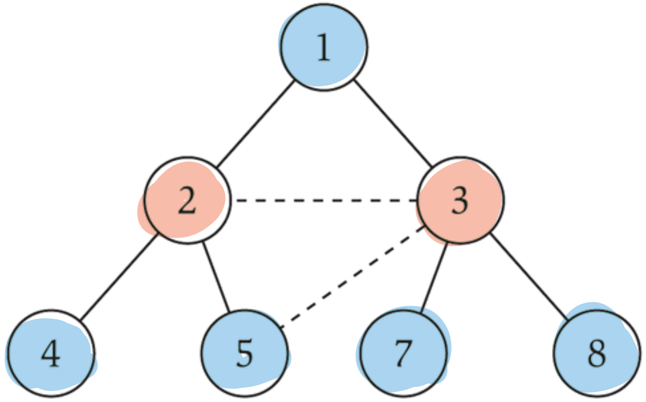
- **Claim:** If our algorithm succeeds, the graph is bipartite (i.e., can be 2-colored)
- **Proof:** Immediate since our algorithm checks validity of the coloring at the end.





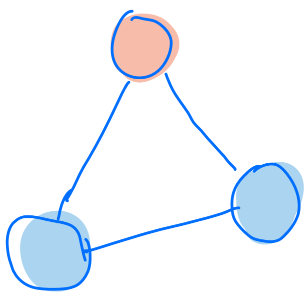
BFS 2-Coloring Failure Implies Not Bipartite

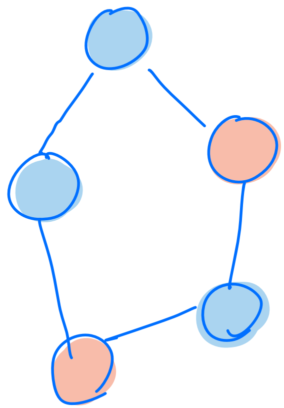
- **Claim:** If our algorithm did not succeed, the graph is not bipartite (i.e., cannot be 2-colored)
- **Question:** Suppose you have not 2-colored the graph successfully, maybe someone else can do it?



Key Fact

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





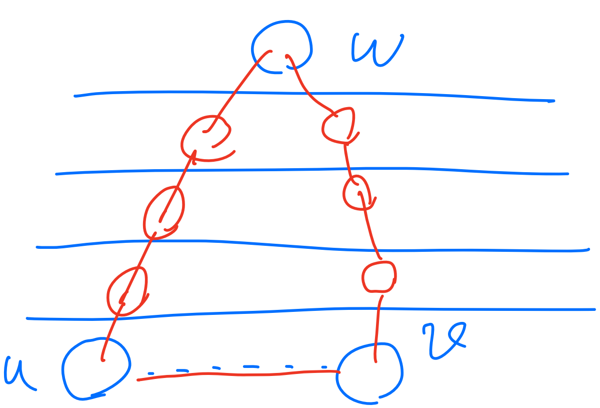
BFS 2-Coloring Failure Implies Not Bipartite

- **Claim:** If BFS did not 2-color the graph, the graph is not bipartite (i.e., cannot be 2-colored)
- **Proof:** If BFS fails, then G contains an odd cycle

Since BFS fails, there is an edge (u,v) such that u and v are assigned the same colors. First u and v must be in the same layer, as otherwise whichever discovered first would add the other to the next layer.

Let w be a common ancestor of u and v furthest from the source.

We claim that $(P(w,u), (u,v), P(v,w))$ is an odd cycle. This is because, $P(w,u)$ and $P(v,w)$ have the same lengths.



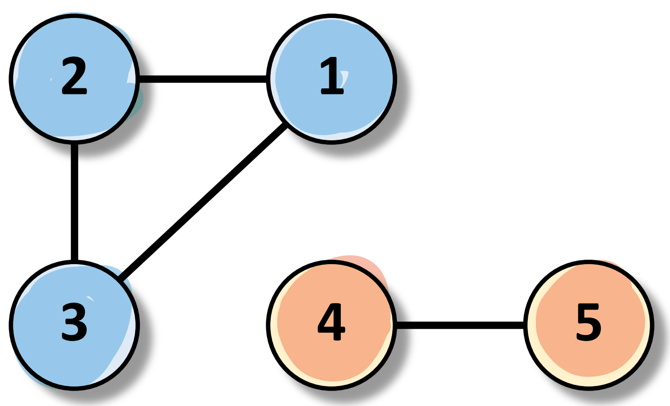
Graphs and Graph Traversals

a. Introduction to Graphs
b. Graph Traversals: DFS
c. Topological Ordering
d. Breadth-First Search
e. Bipartite Graphs and Graph Traversals Recap
f. Connected Components

Connected Components (Undirected)

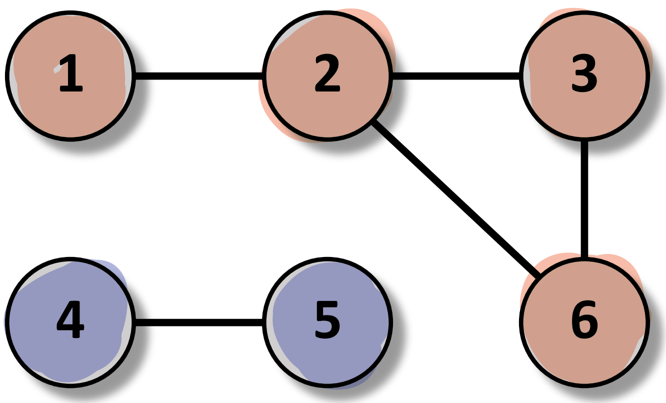
- An undirected graph is connected if for every two vertices $u, v \in V$, there is a path from u to v

- **connected component:** a maximal subset of vertices which are all connected in G



Connected Components (Undirected)

- **Algorithm:**
  - Pick a node v
  - Use DFS or BFS to find all nodes reachable from v
  - Labels those as one connected component
  - Repeat until all nodes are in some component



Connected Components (Undirected)

```
CC(G):
  // Initialize an empty array and a counter
  let comp[1:n] = ⊥, c = 1

  // Iterate through nodes
  for (u = 1,…,n):
    // Ignore this node if it already has a comp.
    // Otherwise, explore it using DFS
    if (comp[u] = ⊥):
      run DFS(G,u)
      let comp[v] = c for every v found by DFS
      let c = c + 1

  output comp[1:n]
```

Running Time

DFS takes $O(n+m')$ time

initialization

m' is the # of reachable edges from the source.

```
CC(G):
  let comp[1:n] = ⊥, c ← 1

  for (u = 1,…,n):
    if (comp[u] = ⊥):
      run DFS(G,u)
      let comp[v] = c for every v
          found by DFS
      let c = c + 1

  output comp[1:n]
```

$O($

Therefore, overall this takes $O(n+m)$ times when run on all connected components.

Connected Components (Undirected)

- **Problem:** Given an undirected graph G, split it into connected components
- **Algorithm:** Can split a graph into connected components in time $\Theta(n + m)$ using DFS
- **Punchline:** Usually assume graphs are connected
  - Implicitly assume that we have already broken the graph into CCs in $\Theta(n + m)$ time

Graph Traversals Recap

- **Basic Graph Theory:**
  - Degrees, paths, cycles, trees
- **Graph representations:**
  - Adjacency list and adjacency matrix
- **Depth-First Search:**
  - Discovery and finish times
  - Tree, forward, back, and cross edges
  - DFS from any node takes $O(n + m)$ time
- **DAGs:**
  - No directed cycles, no back edges
  - Topological ordering in $\Theta(n + m)$ time

Graphs and Graph Traversals Recap

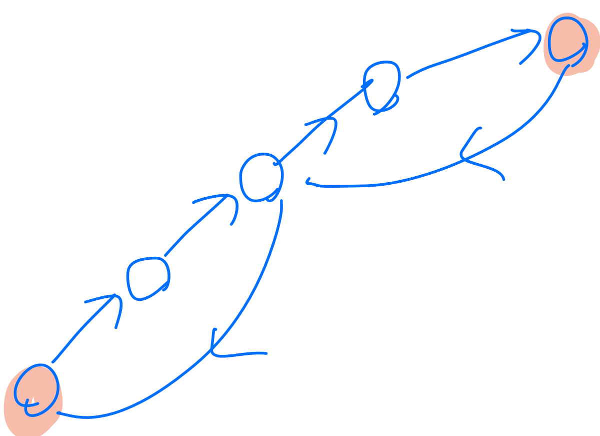
- **Breadth-First Search:**
  - Explores from start node, splits nodes into layers
  - Min-hop path from start to all other reachable nodes
  - BFS from any node takes $O(n+m)$ time
  - Can be used to determine if undirected graph is bipartite
- **Connected components:**
  - Splits graph into its connected components
  - Straightforward $\Theta(n+m)$ time application of DFS and BFS

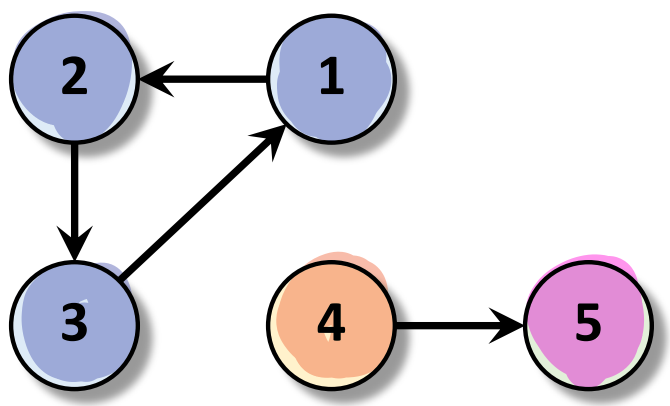
Graphs and Graph Traversals

a. Introduction to Graphs
b. Graph Traversals: DFS
c. Topological Ordering
d. Breadth-First Search
e. Bipartite Graphs and Graph Traversals Recap
f. Connected Components
g. Strongly Connected Components

Strongly Connected Components

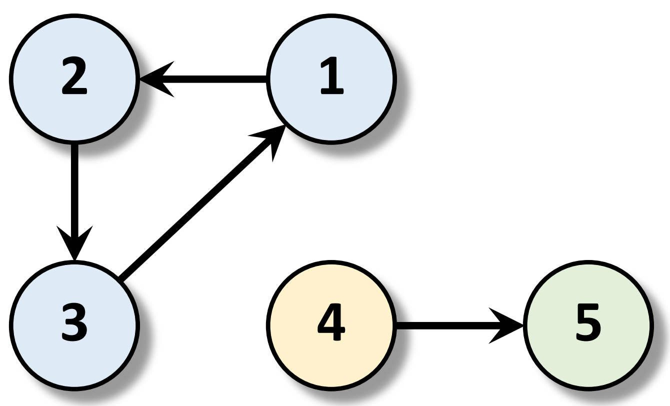
- **Definition:** In a directed graph, we say two vertices u and v are strongly connected if there is a path from v to u and a path from u to v.
- A strongly connected component is a maximal set of vertices all two of which are strongly connected.





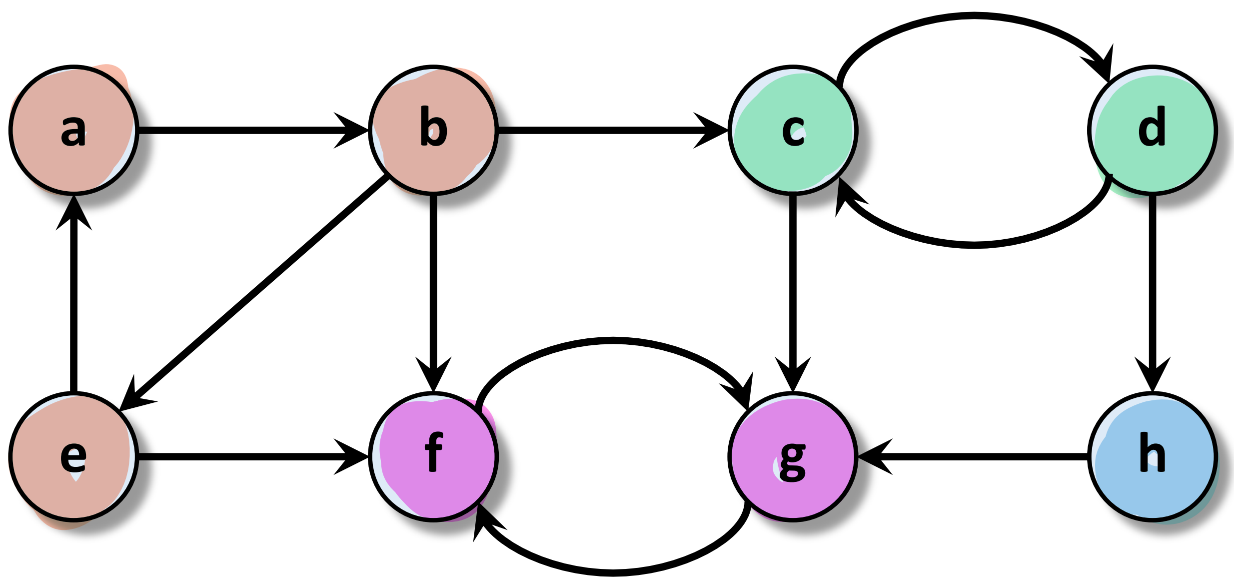
Strongly Connected Components

- **Problem:** Given a directed graph G, split it into strongly connected components
- **Input:** Directed graph $G = (V, E)$
- **Output:** A labeling of the vertices by their strongly connected component



Ask the Audience

- Find all the strongly connected components (SCCs) of this directed graph



Strongly Connected Components

- **Observation:** $\text{SCC}(s)$ is all nodes $v \in V$ such that v is reachable from s and vice versa
  - Can find all nodes reachable from s using DFS
  - How do we find all nodes that *can reach* s?
    - DFS(s) in reverse of the graph!

SCCs by DFS

```
SCC-Slow():
  Gᴿ = G with all edges "reversed"

  // Initialize an array and counter
  comp[1:n] = ⊥, c = 1

  for (u = 1,…,n):
    // If u has not been explored
    if (comp[u] = ⊥):
      S = set of nodes found by DFS(G,u)
      T = set of nodes found by DFS(Gᴿ,u)
      // S ∩ T contains SCC(u)
      label S ∩ T with c
      c = c + 1

  return comp
```

Runs in $n(n+m)$ time

$O(n+m)$

$O(n^2+nm)$ time