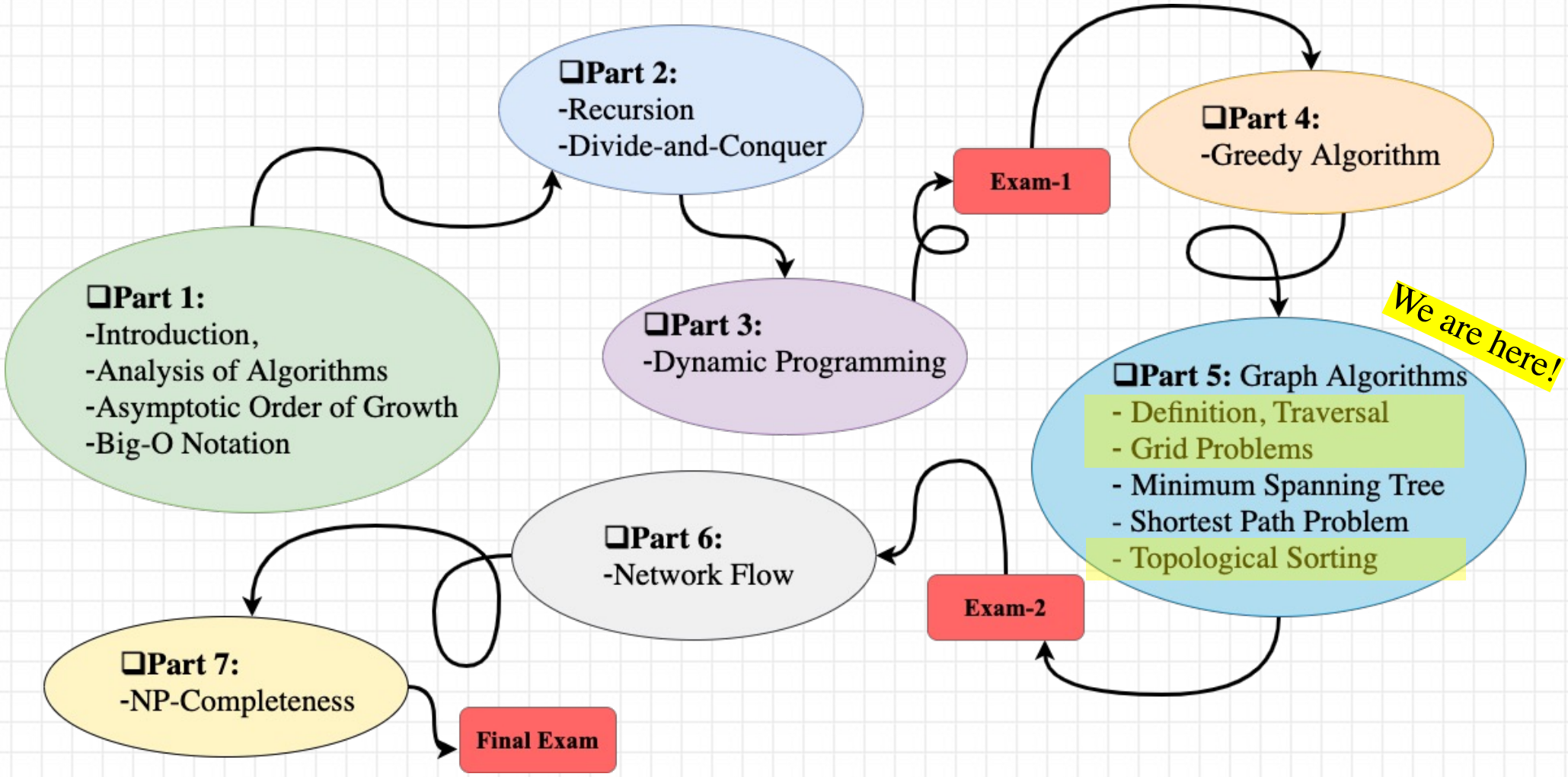
CS-3510:
Design and Analysis of Algorithms

Graph Algorithms:
Traversal Applications II

Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

Roadmap


❑Part 2:
-Recursion
-Divide-and-Conquer
❑Part 1:
-Introduction,
-Analysis of Algorithms
-Asymptotic Order of Growth
-Big-O Notation
❑Part 3:
-Dynamic Programming
Exam-1
❑Part 4:
-Greedy Algorithm
We are here!
❑Part 5: Graph Algorithms
- Definition, Traversal
- Grid Problems
- Minimum Spanning Tree
- Shortest Path Problem
- Topological Sorting
Exam-2
❑Part 6:
-Network Flow
❑Part 7:
-NP-Completeness
Final Exam

Announcements

1. A few updates to earlier lecture notes/slides:
   - Lecture 08 – Slide # 18:
     - A **path** in a directed/undirected graph…
   - Lecture 09 – DFS Traversal Example Slides #17-#53
     - The stack representation is updated to demonstrate the exact behavior when pop operations happen.
   - Lecture 09 – DFS Forest Definition and Example
     - Slide # 81 added to present a more acceptable definition of DFS forest.

1. HW4 is released. Due "Saturday" 07/02
2. Algorithm visualizations (http://www.cs3510.com/resources/)

Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components
- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list

- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components

- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

Graph

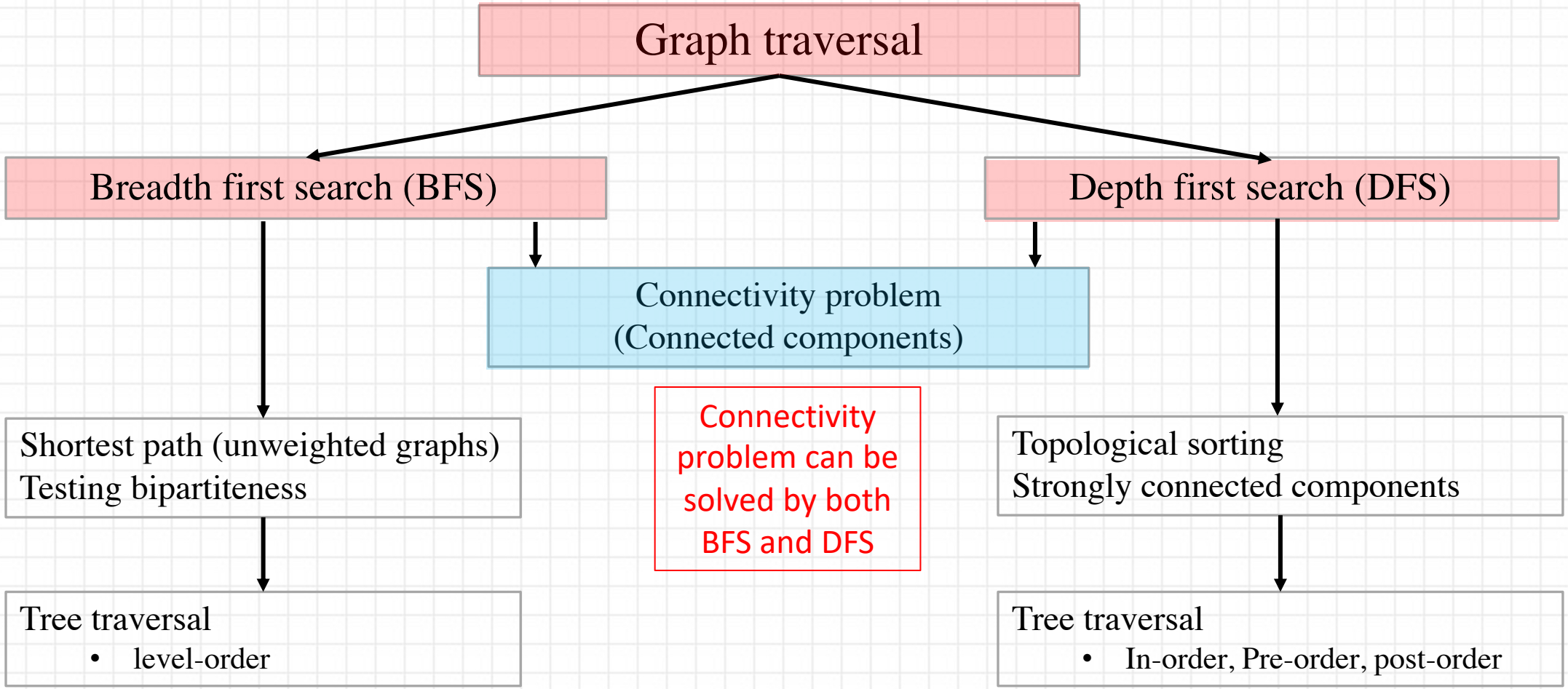
- Graph definition and representation
  - Adjacency matrix
  - Adjacency list

- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components

Now, we know how to run BFS and DFS from a given source node.

- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

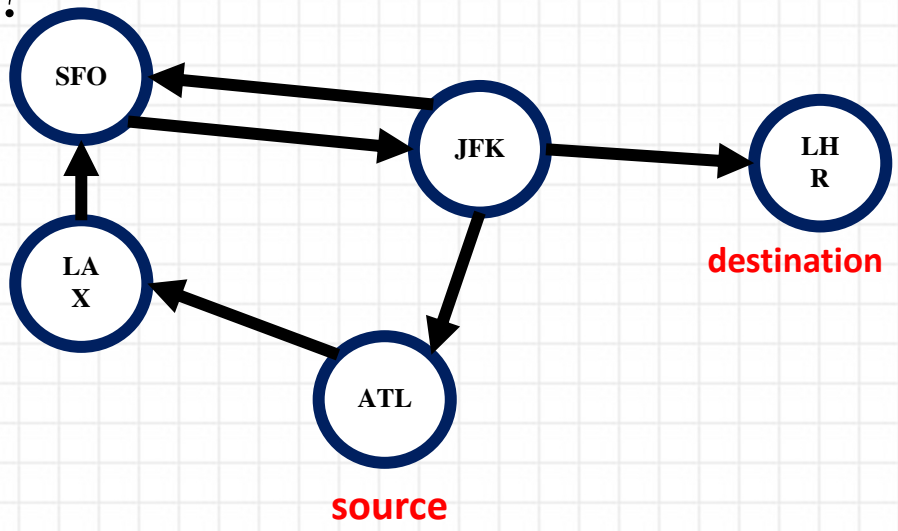
Graph


Graph traversal
Breadth first search (BFS)
Depth first search (DFS)
Connectivity problem
(Connected components)
Shortest path (unweighted graphs)
Testing bipartiteness
Tree traversal
• level-order
Connectivity problem can be solved by both BFS and DFS
Topological sorting
Strongly connected components
Tree traversal
• In-order, Pre-order, post-order

Ex. 1

Graph Traversal: Connected Component

- Ex1: Given a set of flight plans, can we travel from Atlanta (ATL) to London (LHR)?
- Flights:
  - (JFK, ATL)
  - (ATL, LAX)
  - (LAX, SFO)
  - (JFK, SFO)
  - (SFO, JFK)
  - (JFK, LHR)



- Define the corresponding graph
- Run BFS or DFS from the source node, i.e., the node associated with ATL
- During the traversal check if the destination (LHR) is a neighbor of the current node

Demo code time!

Ex. 2

Graph Traversal: Connected Component

- Ex2 [Grid problems]: Given an m-by-n 2D binary matrix in which 0 represent water and 1 represent land, design an algorithm computing the number islands. An island includes one or more horizontally or vertically cells surrounded by water.

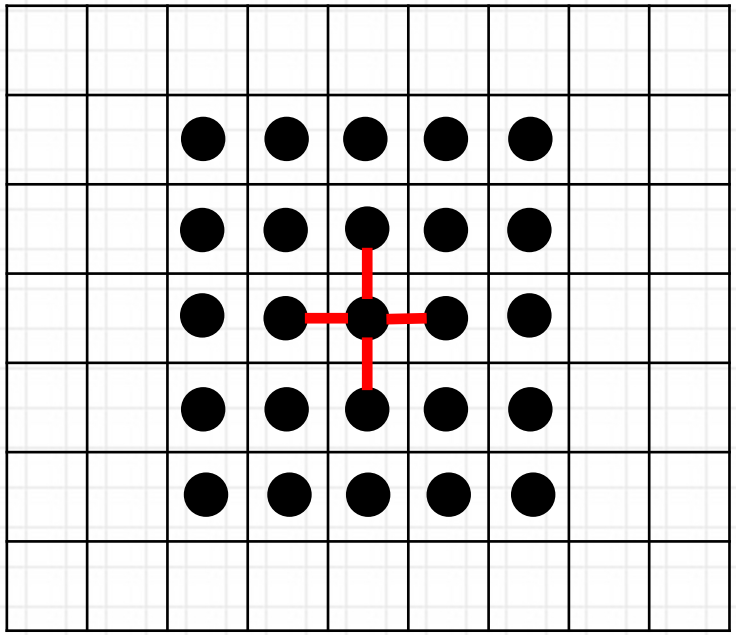
We know the nodes (= grid cells) and we know the neighbors (the relationship), so we can skip the graph definition part!

Each cell = graph node

Neighbors of grid[i][j]:

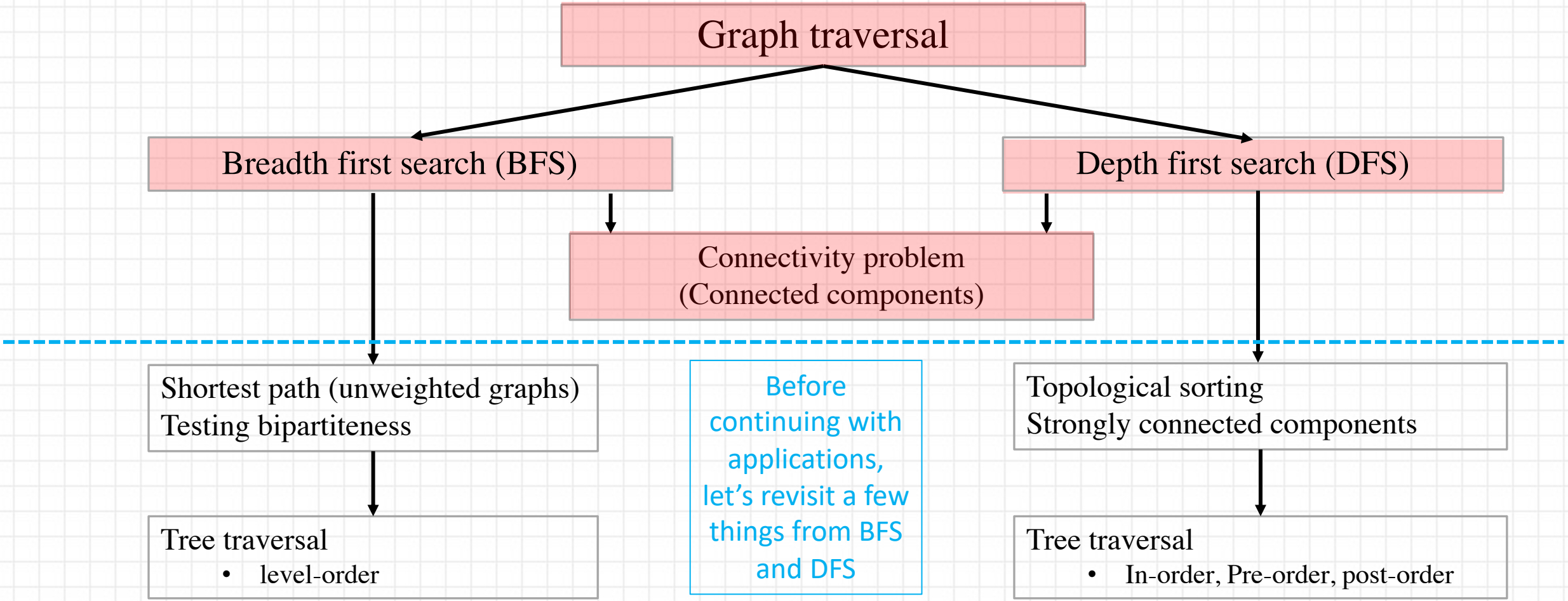
- grid[i-1][j]
- grid[i+1][j]
- grid[i][j-1]
- grid[i][j+1]

grid =



 Demo code!

Graph

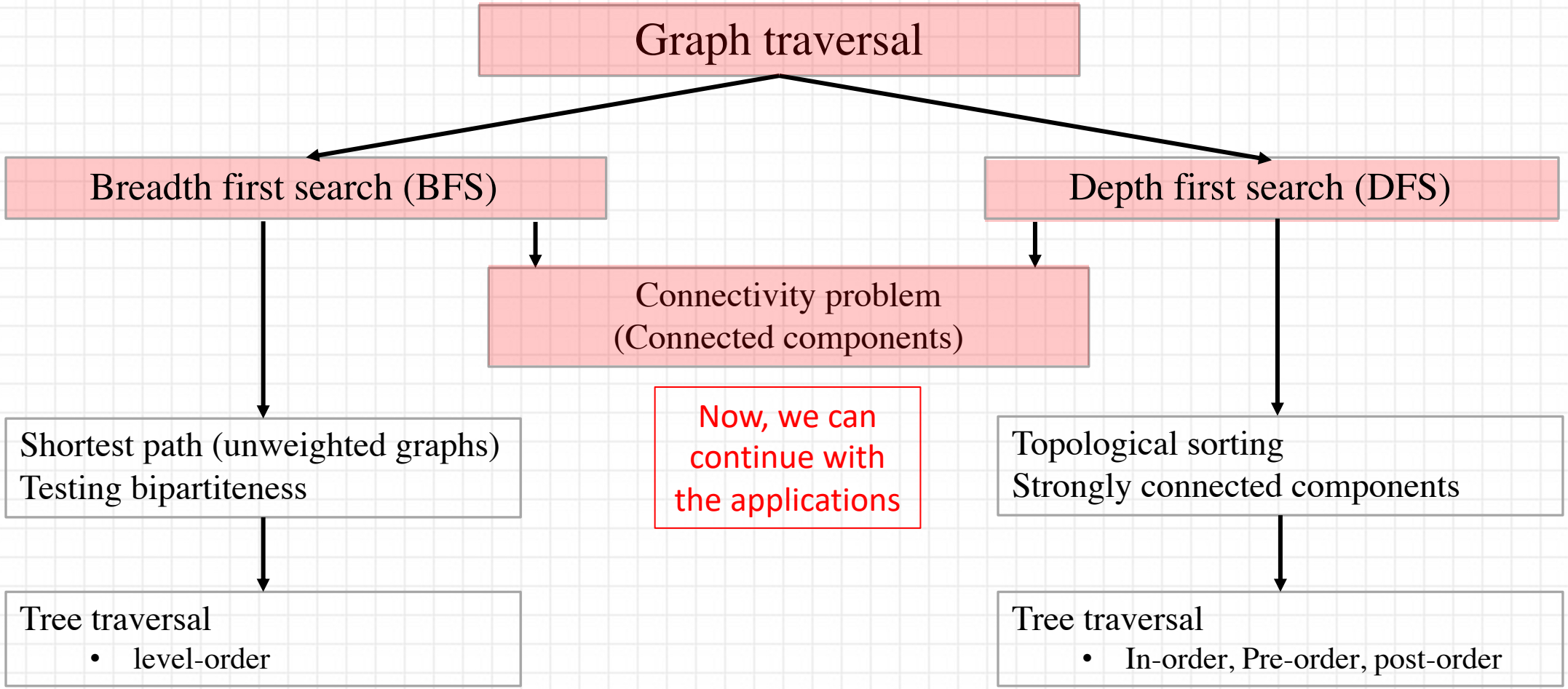

Graph traversal
Breadth first search (BFS)
Depth first search (DFS)
Connectivity problem
(Connected components)
Shortest path (unweighted graphs)
Testing bipartiteness
Tree traversal
• level-order
Before continuing with applications, let's revisit a few things from BFS and DFS
Topological sorting
Strongly connected components
Tree traversal
• In-order, Pre-order, post-order

BFS and DFS

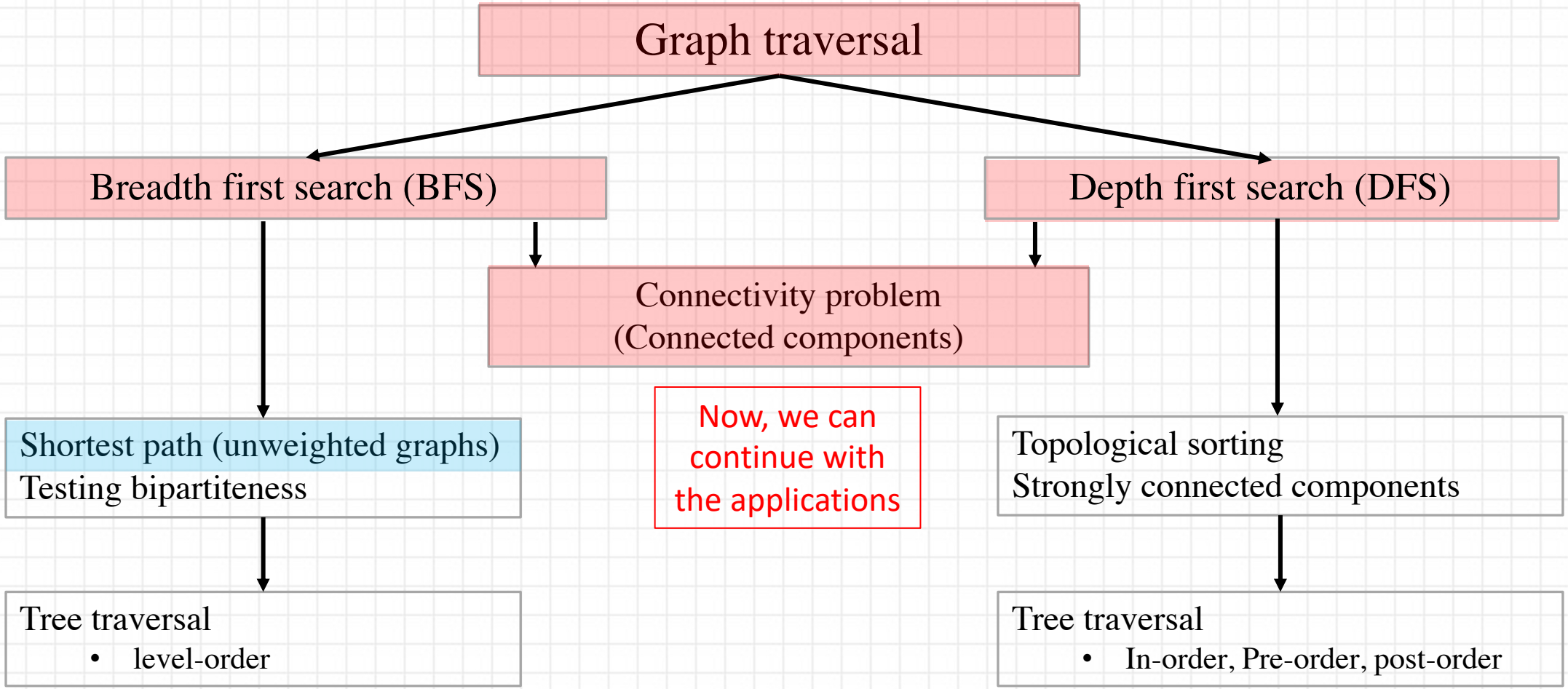
- Both are graph traversal algorithms

BFS	DFS
Iterative: Queue (FIFO), Time:O(|V| + |E|), Space: O(|V|)	Recursive: (execution stack), Iterative: Stack(LIFO) Time:O(|V| + |E|), Space: O(|V|)
➢ **BFS builds a breadth-first tree as it searches the graph.** ➢ **We can print out the vertices on a shortest path from s to v, using the BFS tree** ➢ **We only have one distance measure (timestamp), denoted by d, assigned to each node, i.e., the time that a node visited for the first (and last) time.**	➢ **The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees.** ➢ **DFS timestamps each node with two numbers;** **d (discovery time) and f (finishing time).** ➢ **The timestamps have parenthesis structure.**

Graph

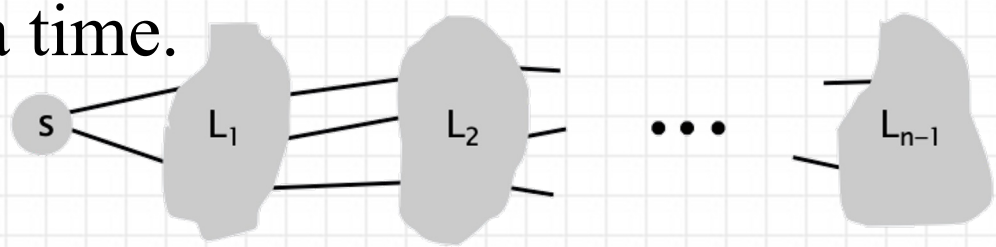

Graph traversal
Breadth first search (BFS)
Depth first search (DFS)
Connectivity problem
(Connected components)
Now, we can continue with the applications
Shortest path (unweighted graphs)
Testing bipartiteness
Topological sorting
Strongly connected components
Tree traversal
• level-order
Tree traversal
• In-order, Pre-order, post-order

Graph


Graph traversal
Breadth first search (BFS)
Depth first search (DFS)
Connectivity problem
(Connected components)
Shortest path (unweighted graphs)
Testing bipartiteness
Now, we can continue with the applications
Topological sorting
Strongly connected components
Tree traversal
• level-order
Tree traversal
• In-order, Pre-order, post-order

BFS: Shortest paths

- We claimed that BFS finds the shortest distance to each reachable vertex in a graph $G = (V, E)$ from a given source vertex $s \in V$.
- BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.

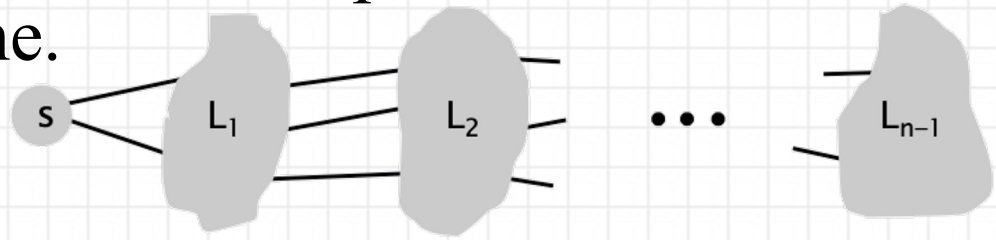


BFS algorithm.

- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

BFS: Shortest paths

- BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



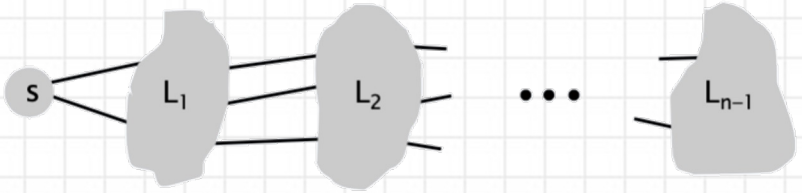
BFS algorithm.

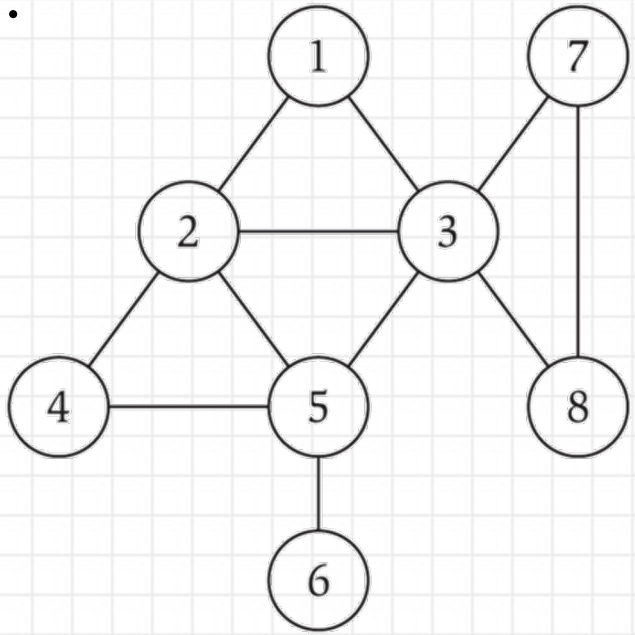
- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

- Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t if and only if t appears in some layer.

BFS: Shortest paths

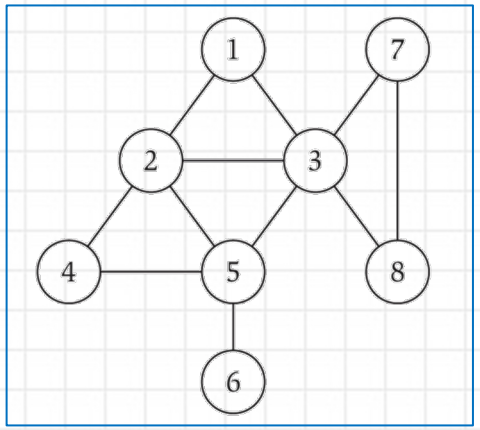
- BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.
- Property. Let T be a BFS tree of $G = (V, E)$, and let (u, v) be an edge of G. Then, the levels of u and v differ by at most 1.

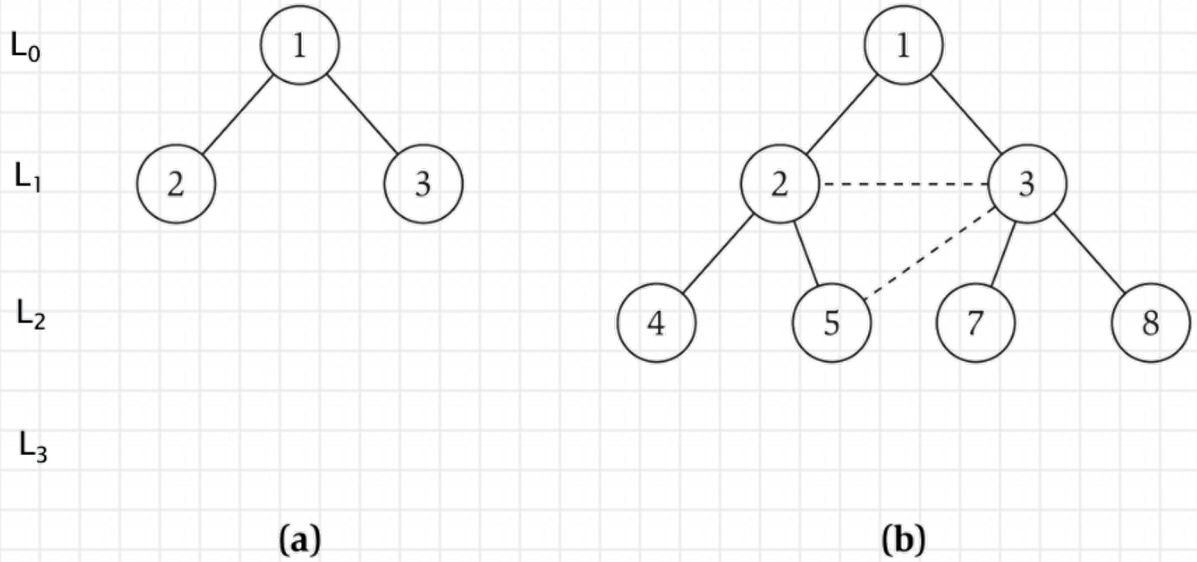




BFS: Shortest paths

- Property. Let T be a BFS tree of $G = (V, E)$, and let (u, v) be an edge of G. Then, the levels of u and v differ by at most 1.

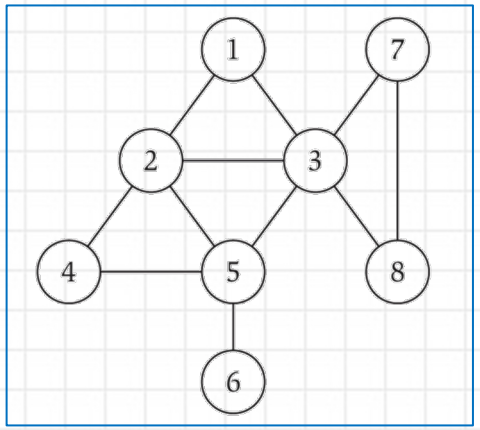


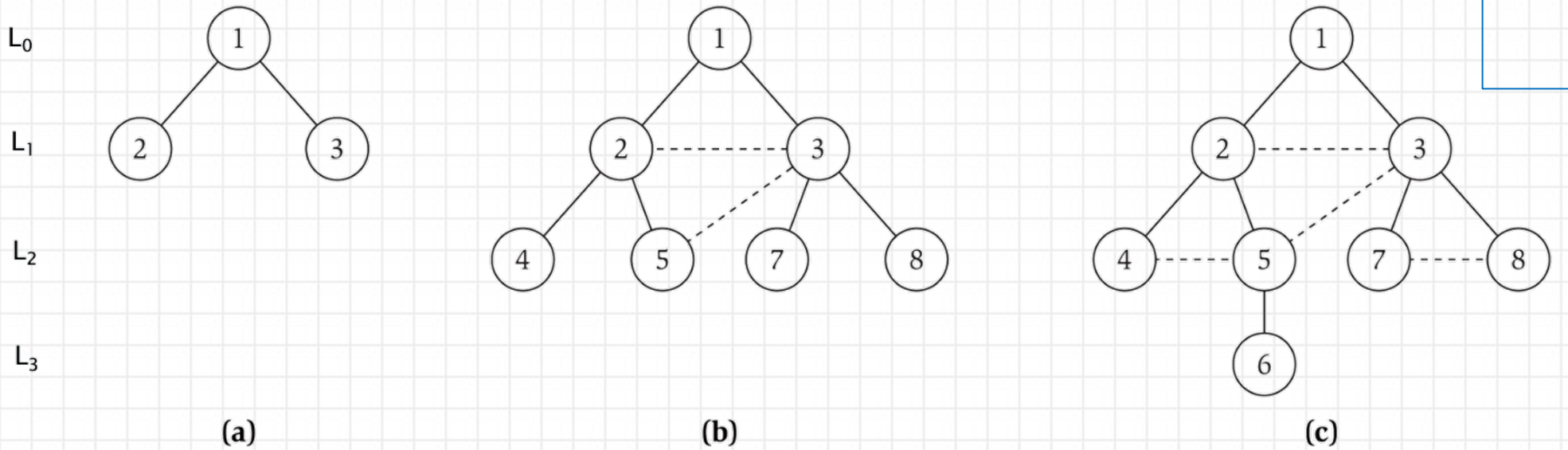


Lemma 1

BFS: Shortest paths

- Property. Let T be a BFS tree of $G = (V, E)$, and let (u, v) be an edge of G. Then, the levels of u and v differ by at most 1.





BFS: Shortest paths (CLRS 22.2)

- We claimed that BFS finds the shortest distance to each reachable vertex in a graph G $= (V, E)$ from a given source vertex s $\in V$.
- Define $\delta(s, v)$ shortest path distance from $s \rightsquigarrow v$.
  - $\delta(s, v) = \infty$ if no path from s to v.

- To prove BFS gives the shortest path to each node u, we need to show the distance d obtained by BFS is equal to $\delta(s, u)$
- Let's see some proofs!

BFS: Shortest paths

- **Lemma 1**
  Let $G = (V, E)$ be a directed/undirected graph, and $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.
- Pf.
  - If u is reachable from s, then so is v. In this case, the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u, v). Thus, the inequality holds.
  - If u is not reachable from s, then $\delta(s, u) = \infty$, and the inequality holds.

BFS: Shortest paths

- **Lemma 2**
  Let G $= (V, E)$ be a directed/undirected graph, and BFS is run on G from a given source s $\in V$. Then, upon termination, for each vertex v $\in V$ the distance d computed by BFS satisfies $d \geq \delta(s, v)$. (d is the upper bound of shortest distance)

BFS: Shortest paths

- **Lemma 2**
  Let G $= (V, E)$ be a directed/undirected graph, and BFS is run on G from a given source s $\in V$. Then, upon termination, for each vertex v $\in V$ the distance d computed by BFS satisfies $d \geq \delta(s, v)$. (d is an upper bound for shortest distance)
- Pf.
  - Induction on the number of nodes added to the queue (Enqueue operations)
  - Inductive base is when we add s to the queue.
    - $d = 0 \geq \delta(s, s)$ and $d = \infty \geq \delta(s, v)\ \forall\ v \in V - \{s\}$.
  - Inductive hypothesis: $d_v \geq \delta(s, v)$ for $\forall\ v \in V$.
  - Inductive step: consider an unvisited node v that is discovered from node u
    - From hypothesis: $d_u \geq \delta(s, u)$
    - From BFS algorithm: $d_v = d_u + 1 \geq \delta(s, u) + 1 \overset{\text{Lemma 1}}{\geq} \delta(s, v)$

BFS: Shortest paths

- **Lemma 3**
  Let $G = (V, E)$ be a directed/undirected graph, and BFS is run on G from a given source $s \in V$. During the execution Queue $= [v_1, \ldots, v_r]$, where v_1 is the head and v_r is the tail. Then,

$$d_{v_r} \leq d_{v_1} + 1 \text{ and}$$

$$d_{v_i} \leq d_{v_{i+1}} \quad \text{for } i = 1, 2, \ldots, r-1$$

BFS: Shortest paths

- **Lemma 3**
  Let $G = (V, E)$ be a directed/undirected graph, and BFS is run on G from a given source $s \in V$. During the execution Queue $= [v_1, \ldots, v_r]$, where v_1 is the head and v_r is the tail. Then,

$$d_{v_r} \leq d_{v_1} + 1 \text{ and}$$

$$d_{v_i} \leq d_{v_{i+1}} \quad \text{for } i = 1, 2, \ldots, r-1$$

- Pf.
  - Induction on the number of queue operations
  - Holds when the queue = [s]
  - Inductive step
    - Must prove the lemma holds after both dequeuing and enqueuing a vertex.

BFS: Shortest paths

- **Lemma 3**
  BFS is run on G from $s \in V$. Execution Queue $= [v_1, \ldots, v_r]$, where v_1 is the head and v_r is the tail. Then, $d_{v_r} \leq d_{v_1} + 1$ and $d_{v_i} \leq d_{v_{i+1}}$ for $i = 1, 2, \ldots, r-1$
- Pf.
  - Induction on the number of queue operations. Holds when the queue = [s]
  - Inductive step
    1. **Dequeuing:** If the head v_1 is dequeued → v_2 new head
       By the inductive hypothesis: $d_{v_1} \leq d_{v_2}$. But then we have $d_{v_r} \leq d_{v_1} + 1 \leq d_{v_2} + 1$ and the remaining inequalities remain unaffected. So, the lemma follows with v_2 as the new head: $d_{v_r} \leq d_{v_2} + 1$

BFS: Shortest paths

- **Lemma 3**
  BFS is run on G from $s \in V$. Execution Queue $= [v_1, \ldots, v_r]$, where v_1 is the head and v_r is the tail. Then, $d_{v_r} \leq d_{v_1} + 1$ and $d_{v_i} \leq d_{v_{i+1}}$ for $i = 1, 2, \ldots, r-1$
- Pf.
  - Induction on the number of queue operations. Holds when the queue = [s]
  - Inductive step
    2. **Enqueuing:** If we enqueue a vertex v → new node as v_{r+1} in the tail
       We can assume, this enqueuing happens during exploring the neighbors of node u which has already been removed from the queue.
       Therefore, by the inductive hypothesis: $d_u \leq d_{v_1}$ (Note v_1 is the current head). Thus,
       $d_{v_{r+1}} = d_v = d_u + 1 \leq d_{v_1} + 1$.
       From the inductive hypothesis, we also have $d_{v_r} \leq d_u + 1$. So,
       $$d_{v_r} \leq d_u + 1 = d_v = d_{v_{r+1}}$$
       → Thus, the lemma follows, when v is queued.

BFS: Shortest paths

- **Lemma 3**
  Let $G = (V, E)$ be a directed/undirected graph, and BFS is run on G from a given source $s \in V$. During the execution Queue $= [v_1, \ldots, v_r]$, where v_1 is the head and v_r is the tail. Then,

$$d_{v_r} \leq d_{v_1} + 1 \text{ and}$$

$$d_{v_i} \leq d_{v_{i+1}} \quad \text{for } i = 1, 2, \ldots, r-1$$

- **Corollary**: Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j . Then $d_{v_i} \leq d_{v_j}$ at the time that v_j is enqueued.
- Pf. Immediate from **Lemma 3**.

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
  Let G $= (V, E)$ be a directed/undirected graph, and BFS is run on G from a given source s $\in V$. Then, during the execution,
  - BFS discovers every vertex v $\in V$ that is reachable from the source s, and
  - Upon termination, $d = \delta(s, v)$ for all v $\in V$, where d is the distance computed by BFS.
  - Moreover, for any vertex v $\neq$ s that is reachable from s, one of the shortest paths from s to v is a shortest path from s to π(v) followed by edge (π(v), v).

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume d is not equal to the shortest distance.
  - So, let v be such a vertex, i.e., $d_v \neq \delta(s, v)$. By Lemma 2: $d_v > \delta(s, v)$. Vertex v must be reachable from s. If it is not, then $d_v \ngeq \delta(s, v) = \infty$.
  - Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, v) > \delta(s, u)$ and according to how we chose v, we have $d_u = \delta(s, u)$.

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume d is not equal to the shortest distance.
  - So, let v be such a vertex, i.e., $d_v \neq \delta(s, v)$. By Lemma 2: $d_v > \delta(s, v)$. Vertex v must be reachable from s. If it is not, then $d_v \ngeq \delta(s, v) = \infty$.
  - Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, v) > \delta(s, u)$ and according to how we chose v, we have $d_u = \delta(s, u)$. So,

$$d_v > \delta(s, v) = \delta(s, u) + 1 = d_u + 1$$

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume d is not equal to the shortest distance.
  - So, let v be such a vertex, i.e., $d_v \neq \delta(s, v)$. By Lemma 2: $d_v \geq \delta(s, v)$. Vertex v must be reachable from s. If it is not, then $d_v \ngeq \delta(s, v) = \infty$.
  - Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, v) > \delta(s, u)$ and according to how we chose v, we have $d_u = \delta(s, u)$. So, $d_v > \delta(s, v) = \delta(s, u) + 1 = d_u + 1$

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume d is not equal to the shortest distance.
  - So, let v be such a vertex, i.e., $d_v \neq \delta(s, v)$. By Lemma 2: $d_v \geq \delta(s, v)$. Vertex v must be reachable from s. If it is not, then $d_v \ngeq \delta(s, v) = \infty$.
  - Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, v) > \delta(s, u)$ and according to how we chose v, we have $d_u = \delta(s, u)$. So, $d_v > \delta(s, v) = \delta(s, u) + 1 = d_u + 1$
  - Now consider the time when BFS chooses to dequeue vertex u from the queue. At this time, vertex v is either white (unvisited), gray (visited, but the neighbors are not completely explored), or black (visited and neighbors are completely explored). We show each of these case contradicting the obtained inequality.

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume $d_v \neq \delta(s, v)$.
  - We obtained: $d_v > \delta(s, v) = \delta(s, u) + 1 = d_u + 1$
  - Now consider the time when BFS chooses to dequeue vertex u from the queue.
    - If v is white (unvisited) →by BFS: $d_v = d_u + 1$ Contradicting the inequality
    - If v is gray (visited, but the neighbors are not completely explored),
    - If v is black (visited and neighbors are completely explored).

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume $d_v \neq \delta(s, v)$.
  - We obtained: $d_v > \delta(s, v) = \delta(s, u) + 1 = d_u + 1$
  - Now consider the time when BFS chooses to dequeue vertex u from the queue.
    - If v is white (unvisited)
      - Contradicting the inequality
    - If v is gray (visited, but the neighbors are not completely explored),
      - It is visited (became gray), it happened when some parent node "w" is dequeued.
      - "w" has been removed from the queue earlier than "u", so, $d_v = d_w + 1$.
      - However, by corollary: $d_w \leq d_u$, and so $d_v = d_w + 1 \leq d_u + 1$ Contradicting the inequality
    - If v is black (visited and neighbors are completely explored).

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume $d_v \neq \delta(s, v)$.
  - We obtained: $d_v > \delta(s, v) = \delta(s, u) + 1 = d_u + 1$
  - Now consider the time when BFS chooses to dequeue vertex u from the queue.
    - If v is white (unvisited)
      - Contradicting the inequality
    - If v is gray (visited, but the neighbors are not completely explored),
      - Contradicting the inequality
    - If v is black (visited and neighbors are completely explored).
      - It means it was already removed from the queue, and by the corollary: $d_v \leq d_u$
      - Contradicting the inequality

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume $d_v \neq \delta(s, v)$.
  - We obtained: $d_v > \delta(s, v) = \delta(s, u) + 1 = d_u + 1$
  - Now consider the time when BFS chooses to dequeue vertex u from the queue.
    - Contradicting the inequality

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume $d_v \neq \delta(s, v)$.
  - We obtained: $d_v > \delta(s, v) = \delta(s, u) + 1 = d_u + 1$ Contradiction!

  - Therefore, $d_v = \delta(s, v)$ for all v $\in V$
  - All vertex v must be reachable from s. Otherwise, $d_v = \infty > \delta(s, v)$.
  - If $\pi(\mathrm{v}) = \mathrm{u}$, then $d_v = d_u + 1$. Thus, we can obtain the shortest path from s to v by taking a shortest path from s to $\pi(\mathrm{v})$ and then traversing the edge $(\pi(\mathrm{v}), \mathrm{v})$.

BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- Pf.
  - For the sake of contradiction assume $d_v \neq \delta(s, v)$.
  - We obtained: $d_v > \delta(s, v) = \delta(s, u) + 1 = d_u + 1$ Contradiction!

  - Therefore, $d_v = \delta(s, v)$ for all v $\in V$
  - All vertex v must be reachable from s. Otherwise, $d_v = \infty > \delta(s, v)$.
  - If $\pi(\mathrm{v}) = \mathrm{u}$, then $d_v = d_u + 1$. Thus, we can obtain the shortest path from s to v by taking a shortest path from s to $\pi(\mathrm{v})$ and then traversing the edge $(\pi(\mathrm{v}), \mathrm{v})$.

  - **So, BFS gives the shortest path from each reachable node to the source node in an unweighted graph (where all edges have the same unit weight).**

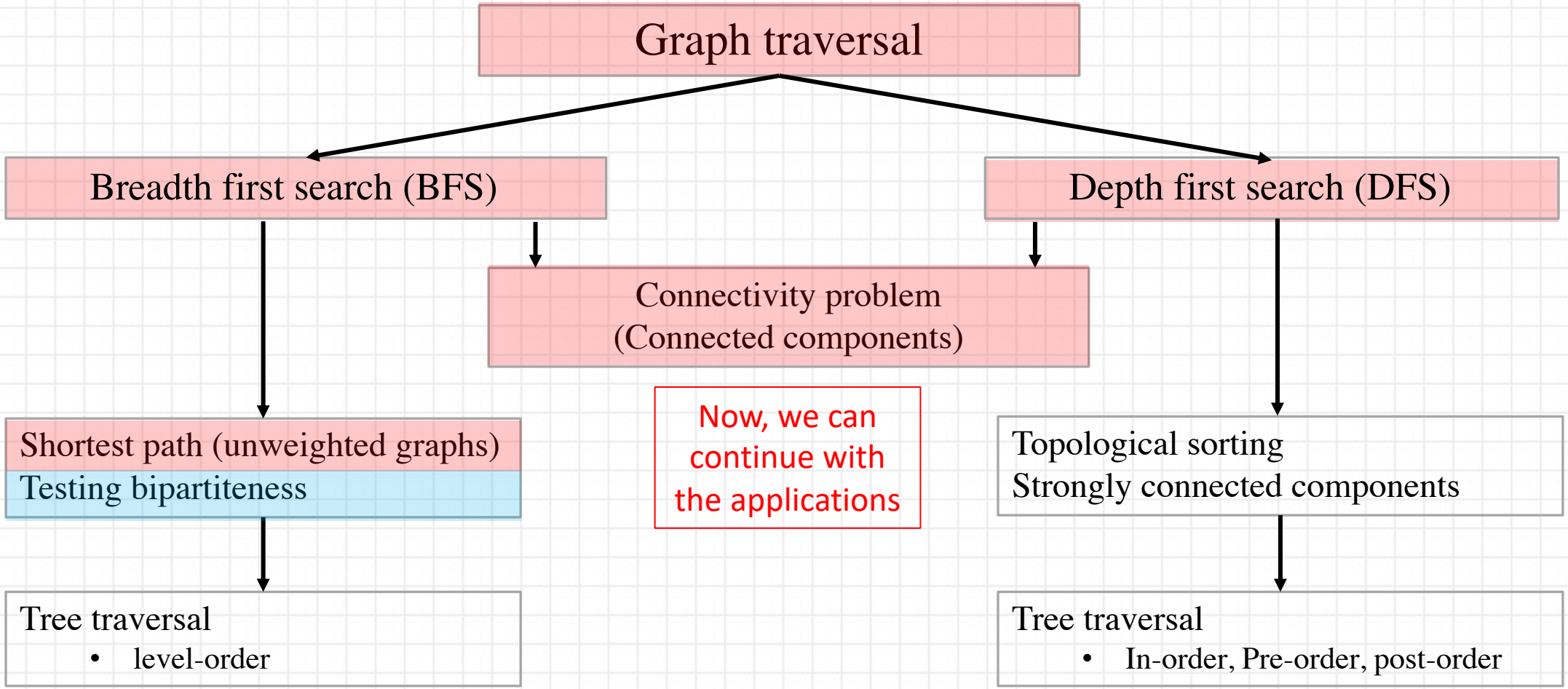
BFS: Shortest paths

- **Theorem: Correctness of BFS; Shortest Paths**
- **BFS gives the shortest path from each reachable node to the source node in an unweighted graph (where all edges have the same unit weight).**

- Check the demo codes one more time for the implementations…

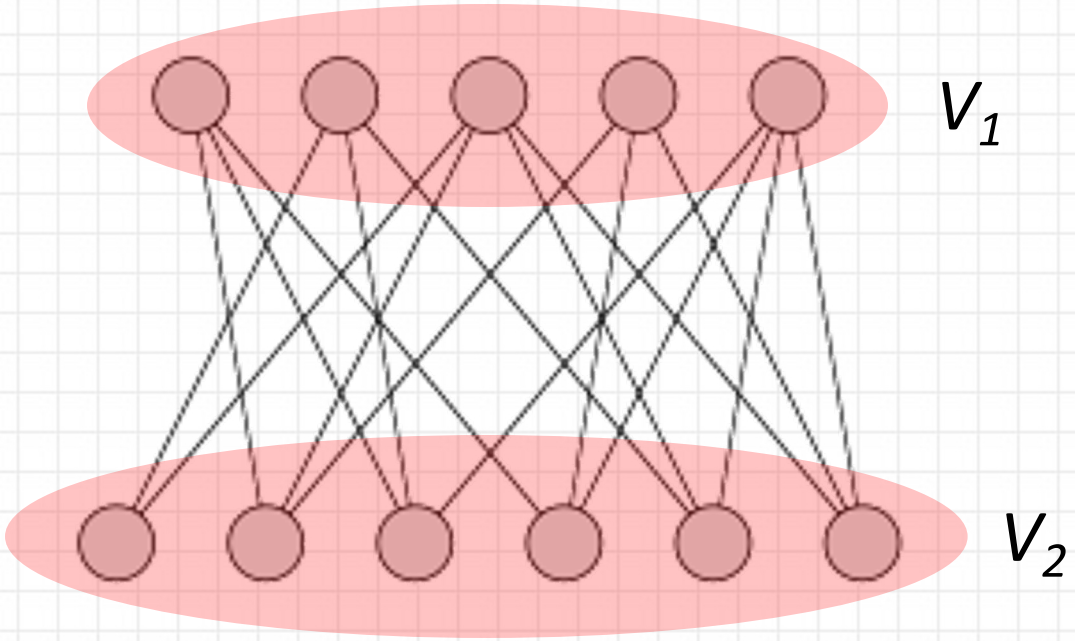
```
1 from collections import deque
2 def bfs(graph, s):
3   q = deque([s])
4   visited = {s: 0}
5
6   while q:
7     curr = q.popleft()
8
9     for nei in graph[curr]:
10      if nei not in visited:
11        visited[nei] = visited[curr] + 1
12        q.append(nei)
13
14  return visited
```

Graph


Graph traversal
Breadth first search (BFS)
Depth first search (DFS)
Connectivity problem
(Connected components)
Shortest path (unweighted graphs)
Testing bipartiteness
Now, we can continue with the applications
Topological sorting
Strongly connected components
Tree traversal
• level-order
Tree traversal
• In-order, Pre-order, post-order

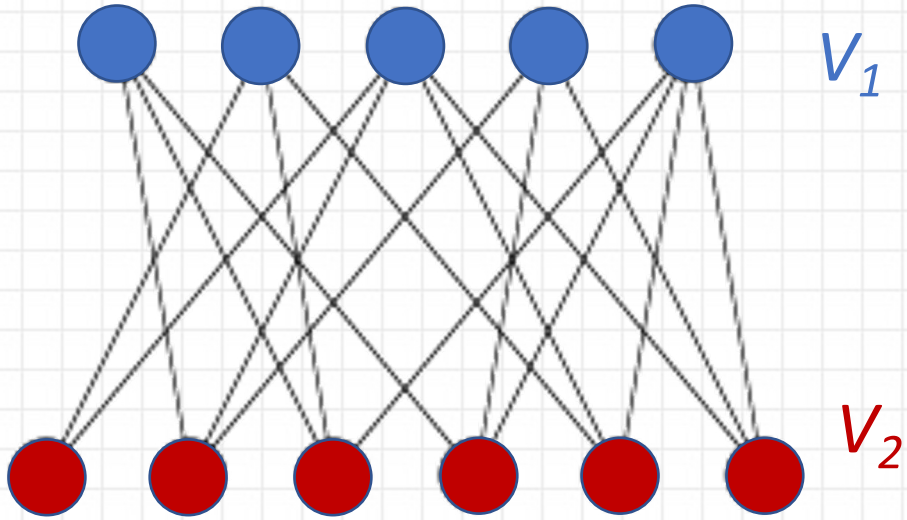
BFS: Testing Bipartiteness

- Def. A bipartite graph is an undirected graph G = (V, E) in which V can be partitioned into two sets V_1 and V_2 such that (u, v) ∈ E implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2.



BFS: Testing Bipartiteness

- Def. A bipartite graph is an undirected graph G = (V, E) in which V can be partitioned into two sets V_1 and V_2 such that (u, v) ∈ E implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2.

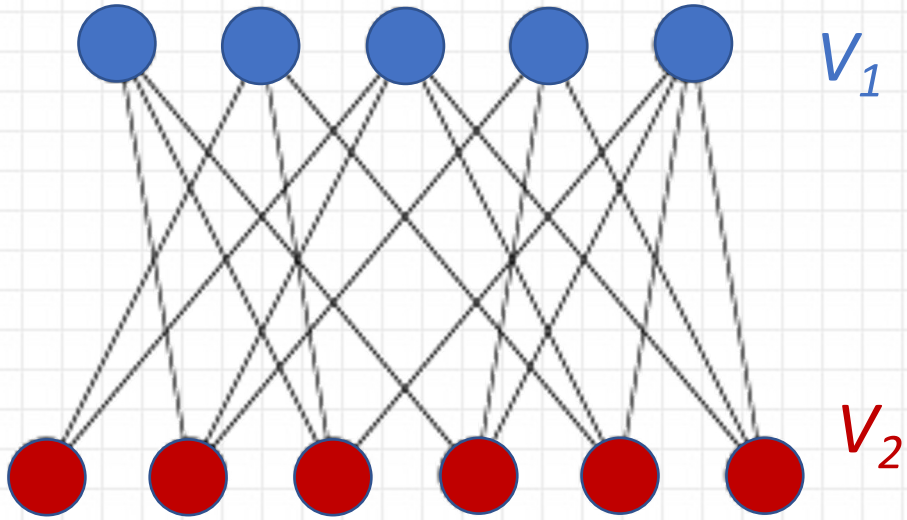


- Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored blue or red such that every edge has one blue and one red end.

BFS: Testing Bipartiteness

- Def. A bipartite graph is an undirected graph G = (V, E) in which V can be partitioned into two sets V_1 and V_2 such that (u, v) ∈ E implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2.

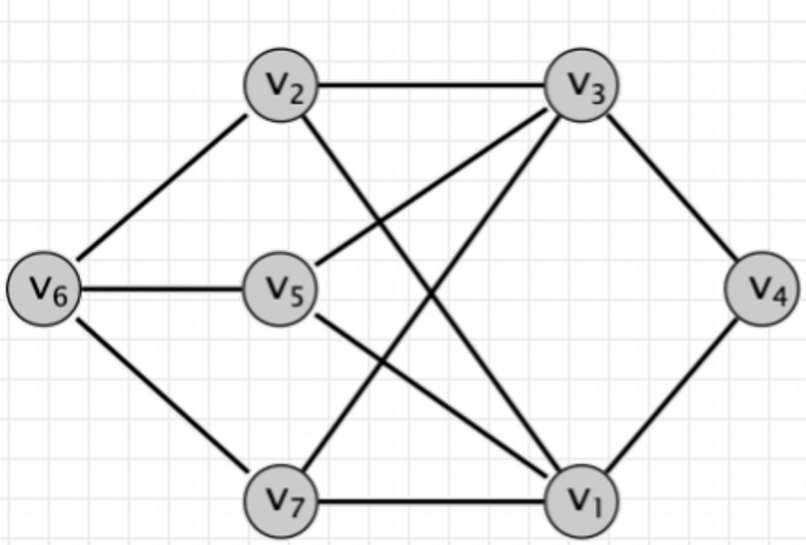
bipartite graph = 2-colorable graph



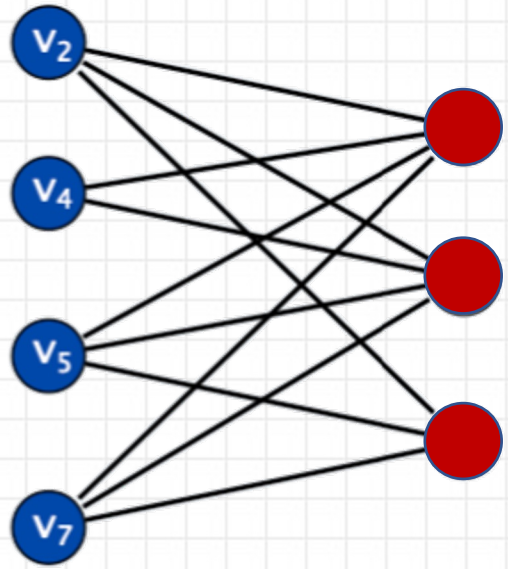
- Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored blue or red such that every edge has one blue and one red end.

BFS: Testing Bipartiteness

- Bipartite graph = 2-colorable graph



a bipartite graph G



another drawing of G

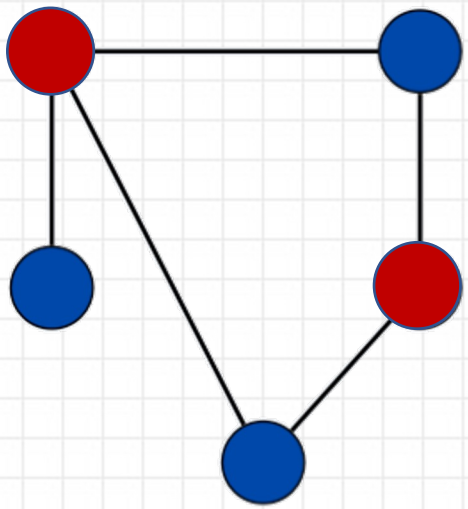
BFS: Testing Bipartiteness

- Bipartite graph = 2-colorable graph
- Applications
  - Stable matching
  - Scheduling: machines = blue nodes, jobs = red nodes

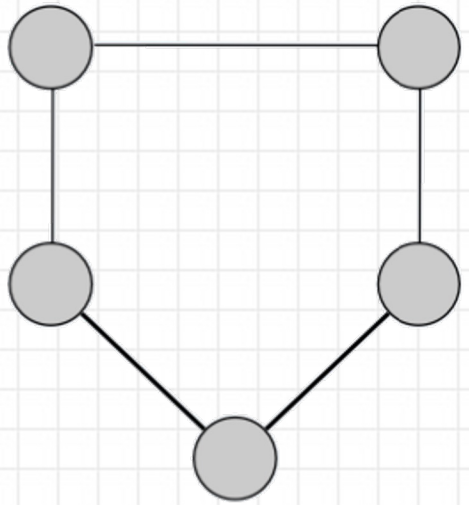
- Many graph problems become:
  - Easier if the underlying graph is bipartite (matching).
  - Tractable if the underlying graph is bipartite (independent set).

BFS: Testing Bipartiteness (KT 3.4)

- Lemma. If a graph G is bipartite, it cannot contain an odd-length cycle.
- Proof. Not possible to 2-color the odd-length cycle, let alone G.



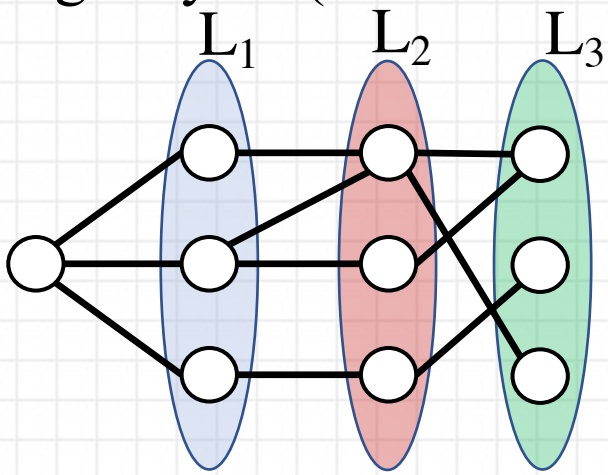
bipartite
(2-colorable)



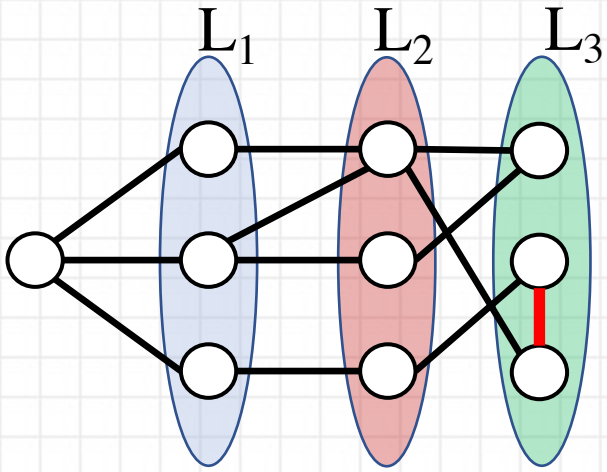
not bipartite
(not 2-colorable)

BFS: Testing Bipartiteness

- Lemma. Let G be a connected graph, and let L_0, ..., L_k be the layers produced by BFS starting at node s. Exactly one of the following holds:
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



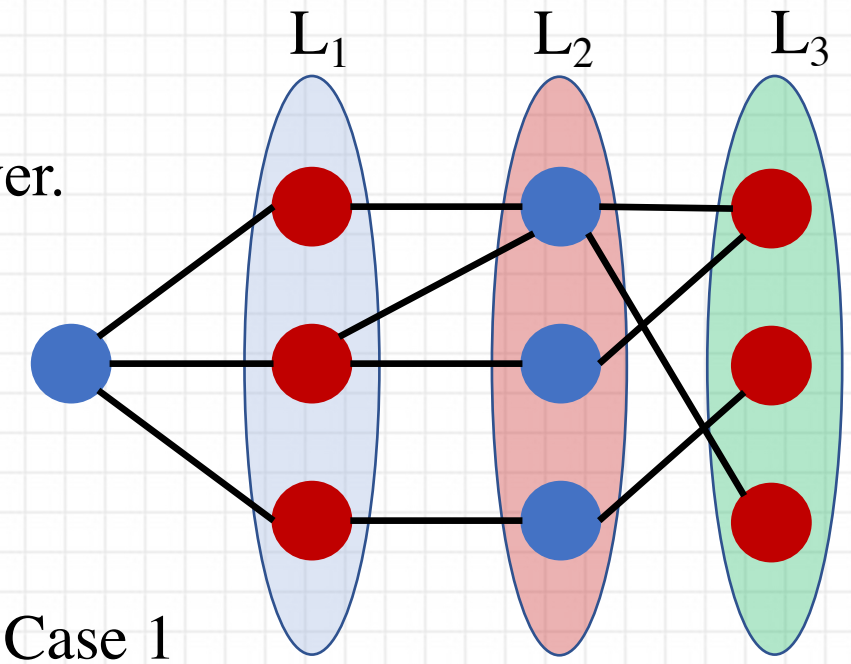
Case 1



Case 2

BFS: Testing Bipartiteness

- Lemma. Let G be a connected graph, and let L_0, ..., L_k be the layers produced by BFS starting at node s. Exactly one of the following holds:
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).
- Proof (1)
  - Suppose no edge joins two nodes in same layer.
  - By BFS property, each edge joins two nodes in adjacent levels.
  - Bipartition:
    - red = nodes on odd levels,
    - blue = nodes on even levels.

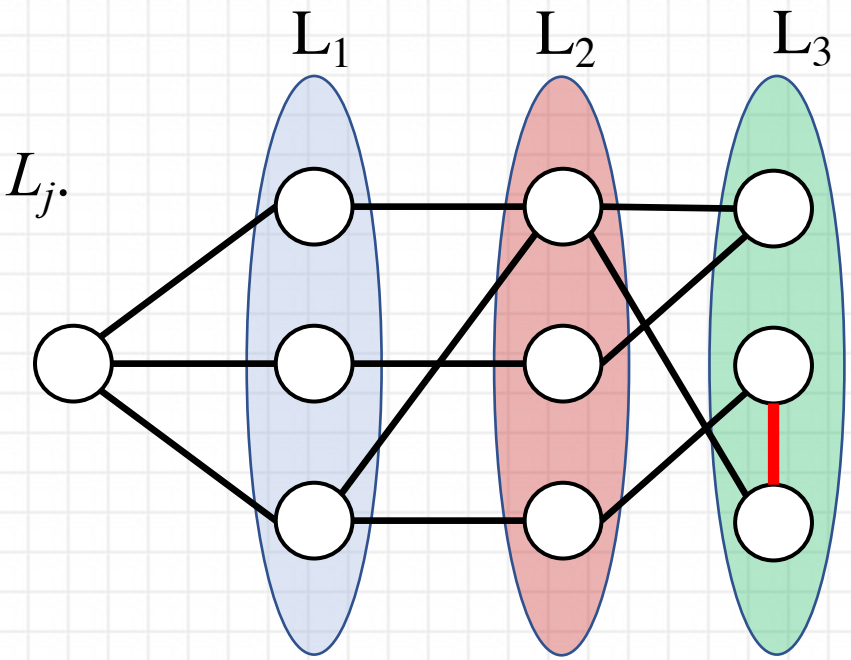


Case 1

BFS: Testing Bipartiteness

- Lemma. Let G be a connected graph, and let L_0, ..., L_k be the layers produced by BFS starting at node s. Exactly one of the following holds:
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

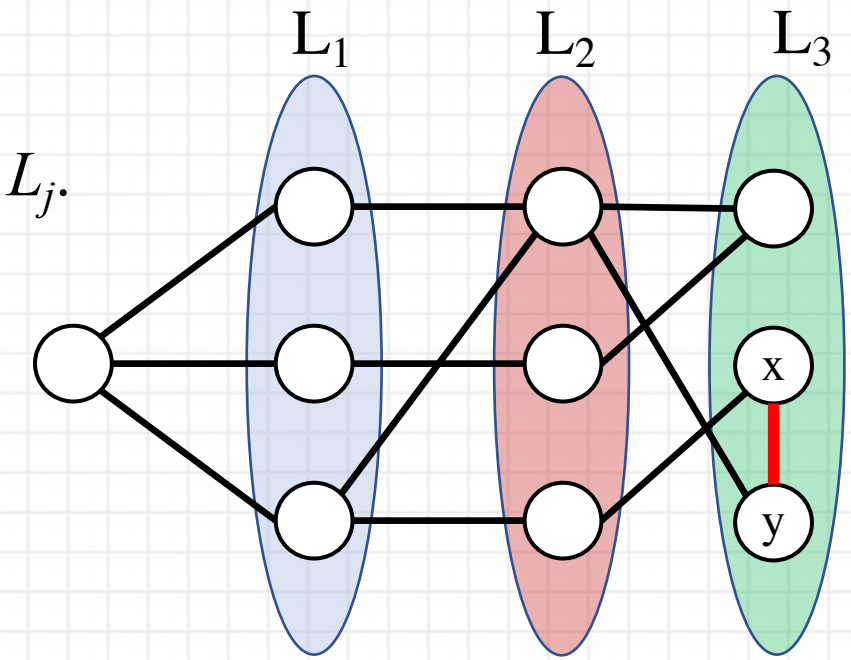
- Proof (2)
  - Suppose (x, y) is an edge with x, y in same level L_j.
  - Let $z = lca(x, y)$ = lowest common ancestor.
  - Let L be level containing z.
  - Consider cycle that takes edge from x to y,
  - then path from y to z, then path from z to x.
  - Its length is $1 + (j - i) + (j - i)$, which is odd.



BFS: Testing Bipartiteness

- Lemma. Let G be a connected graph, and let L_0, ..., L_k be the layers produced by BFS starting at node s. Exactly one of the following holds:
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

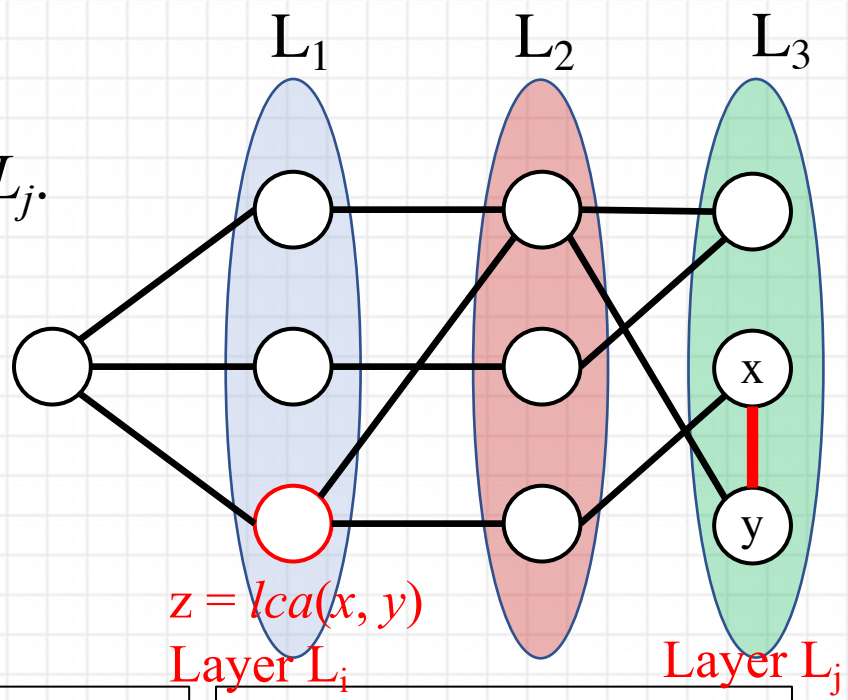
- Proof (2)
  - Suppose (x, y) is an edge with x, y in same level L_j.
  - Let $z = lca(x, y)$ = lowest common ancestor.
  - Let L be level containing z.
  - Consider cycle that takes edge from x to y,
  - then path from y to z, then path from z to x.
  - Its length is $1 + (j - i) + (j - i)$, which is odd.



BFS: Testing Bipartiteness

- Lemma. Let G be a connected graph, and let L_0, ..., L_k be the layers produced by BFS starting at node s. Exactly one of the following holds:
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

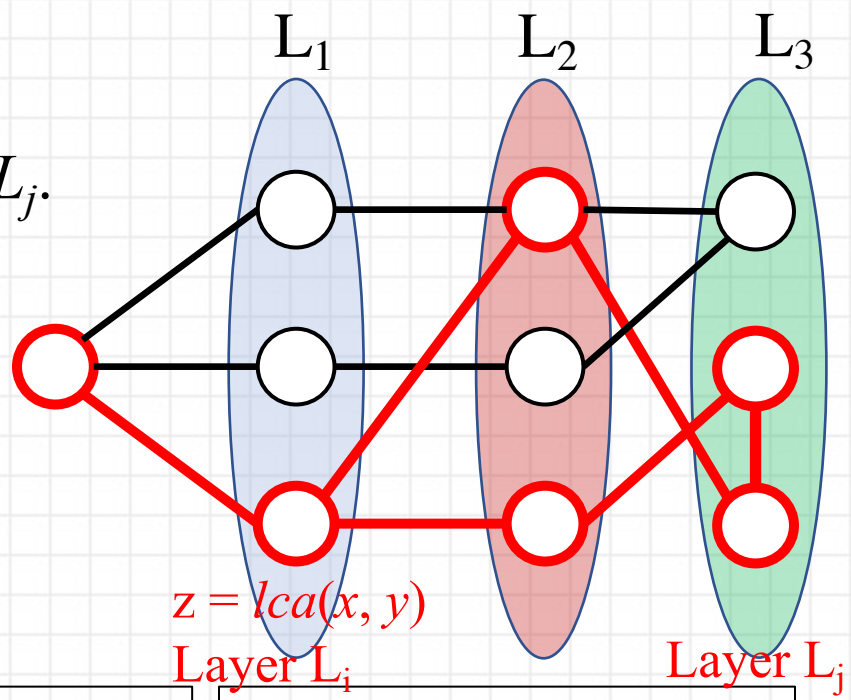
- Proof (2)
  - Suppose (x, y) is an edge with x, y in same level L_j.
  - Let $z = lca(x, y)$ = lowest common ancestor.
  - Let L be level containing z.
  - Consider cycle that takes edge from x to y,
  - then path from y to z, then path from z to x.
  - Its length is $1 + (j - i) + (j - i)$, which is odd.



BFS: Testing Bipartiteness

- Lemma. Let G be a connected graph, and let L_0, ..., L_k be the layers produced by BFS starting at node s. Exactly one of the following holds:
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

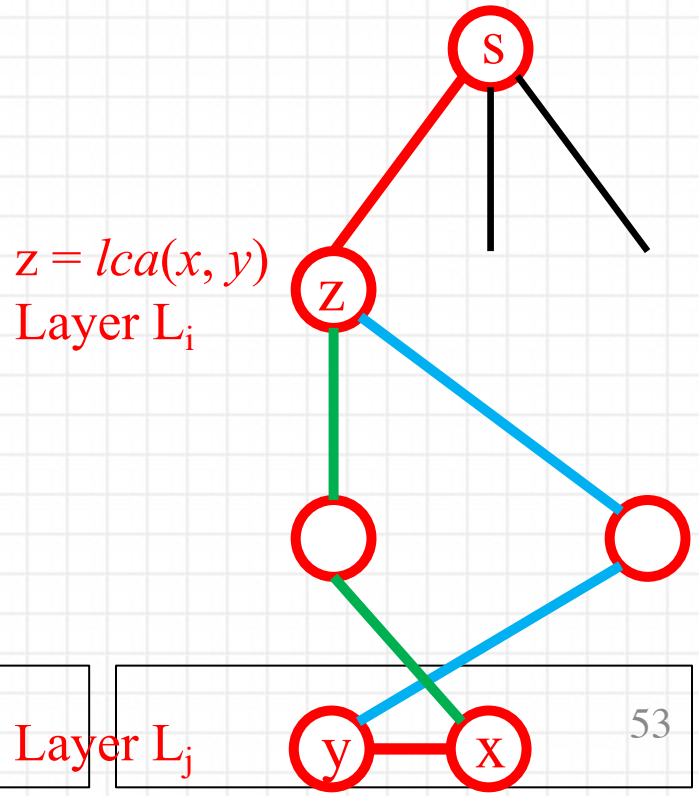
- Proof (2)
  - Suppose (x, y) is an edge with x, y in same level L_j.
  - Let $z = lca(x, y)$ = lowest common ancestor.
  - Let L be level containing z.
  - Consider cycle that takes edge from x to y,
  - then path from y to z, then path from z to x.
  - Its length is $1 + (j - i) + (j - i)$, which is odd.



BFS: Testing Bipartiteness

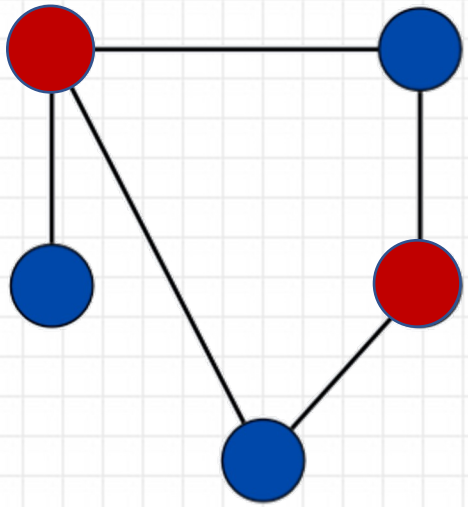
- Lemma. Let G be a connected graph, and let L_0, ..., L_k be the layers produced by BFS starting at node s. Exactly one of the following holds:
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

- Proof (2)
  - Suppose (x, y) is an edge with x, y in same level L_j.
  - Let z = $lca(x, y)$ = lowest common ancestor.
  - Let L be level containing z.
  - Consider cycle that takes edge from x to y,
  - then path from y to z, then path from z to x.
  - Its length is $1 + (j - i) + (j - i) = 2k+1$, which is odd.

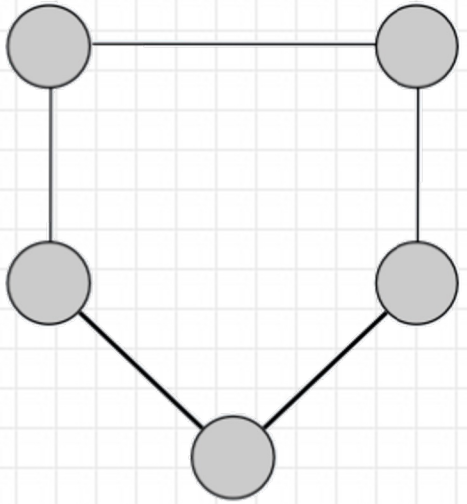


BFS: Testing Bipartiteness

- Corollary. A graph G is bipartite iff it contains no odd-length cycle.



bipartite
(2-colorable)

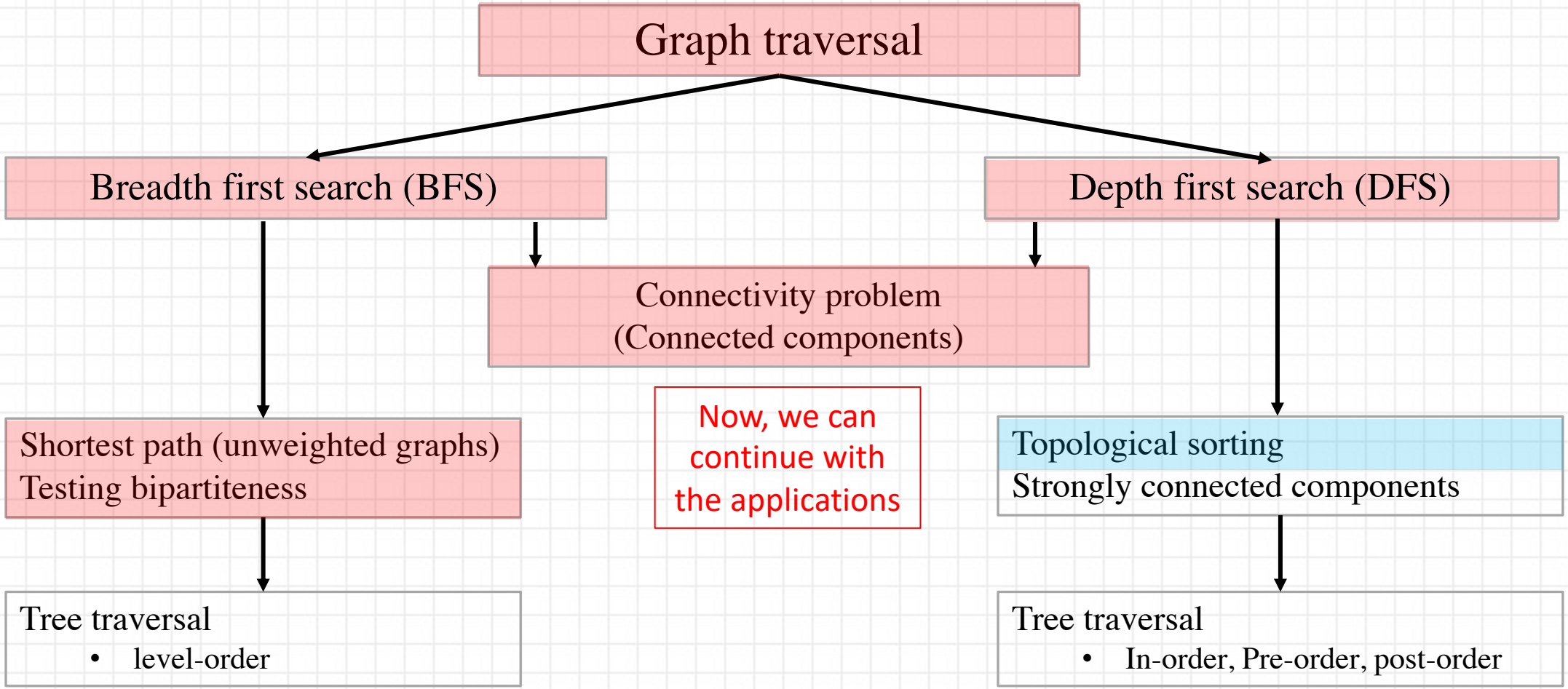


not bipartite
(not 2-colorable)

BFS: Testing Bipartiteness

- We can modify the BFS algorithm to color each neighbor with the opposite color when it explores a node.
- If a neighbor has already been colored (i.e., visited), and has the same color, then return false.
- If the BFS can traverse the entire graph and color all nodes, then return true.

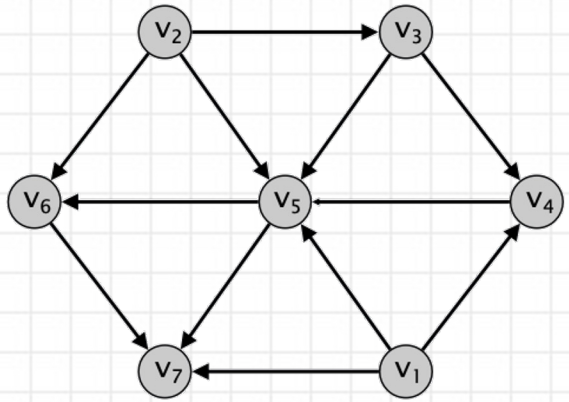
Graph


Graph traversal
Breadth first search (BFS)
Depth first search (DFS)
Connectivity problem
(Connected components)
Shortest path (unweighted graphs)
Testing bipartiteness
Now, we can continue with the applications
Topological sorting
Strongly connected components
Tree traversal
• level-order
Tree traversal
• In-order, Pre-order, post-order

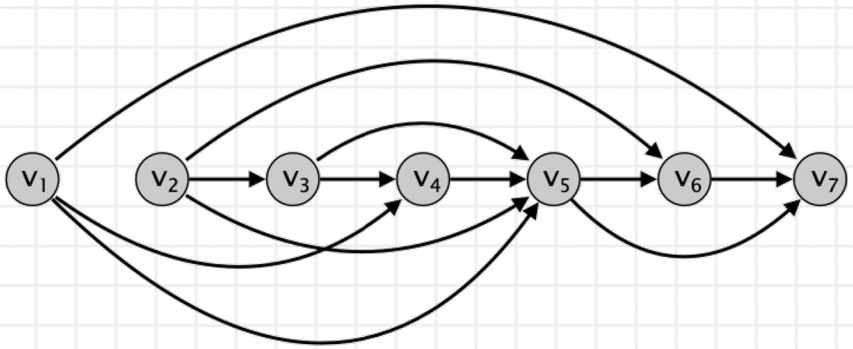
Directed Acyclic Graphs (DAG)

- Def. A directed acyclic graphs (DAG) is a directed graph that contains no directed cycles.

- Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



a DAG



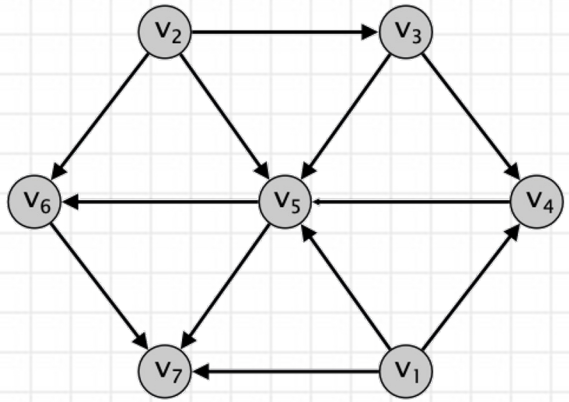
a topological ordering

Directed Acyclic Graphs (DAG)

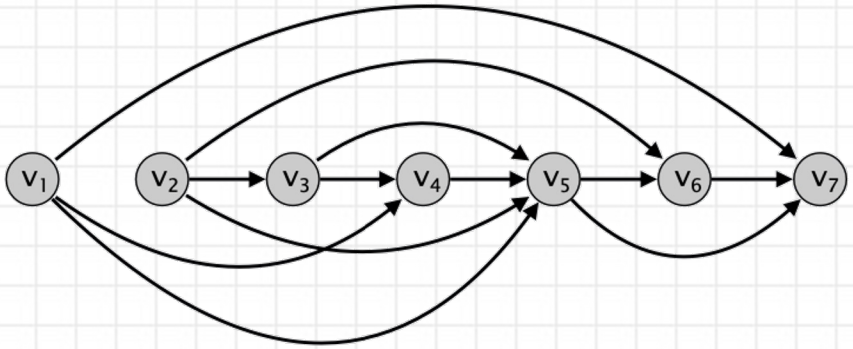
- Def. A directed acyclic graphs (DAG) is a directed graph that contains no directed cycles.

Topological ordering = Topological sort (Top-Sort)

- Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



a DAG



a topological ordering

Directed Acyclic Graphs (DAG)

- Def. A directed acyclic graphs (DAG) is a directed graph that contains no directed cycles.
- Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.
- Topological Ordering → Precedence Constraints
  - Precedence constraints: edge (v_i, v_j) means task v_i must occur before v_j.
- Applications
  - Course prerequisite graph: course v_i must be taken before v_j
  - Compilation: module v_i must be compiled before v_j
  - Pipeline of computing jobs: output of job v_i needed to determine input of job v_j

Directed Acyclic Graphs (DAG)

- If G has a topological order, then G is a DAG.
  - Q. Does every DAG have a topological ordering?
  - Q. If so, how do we compute one?

- If G is a DAG, then G has a topological ordering.
  - If G is a DAG, then G has a node with no entering edges.

G is a DAG $\Leftrightarrow$ G has a topological ordering

- Algorithm finds a topological order in $O(m + n)$ time.

Directed Acyclic Graphs (DAG)

- If G has a topological order, then G is a DAG.
  - Q. Does every DAG have a topological ordering?
  - Q. If so, how do we compute one?

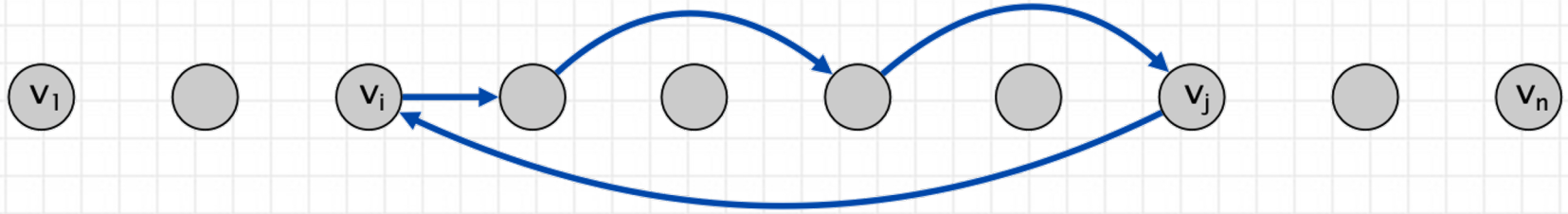
- If G is a DAG, then G has a topological ordering.
  - If G is a DAG, then G has a node with no entering edges.

G is a DAG $\Leftrightarrow$ G has a topological ordering

- Algorithm finds a topological order in $O(m + n)$ time.

Directed Acyclic Graphs (DAG)

- If G has a topological order, then G is a DAG.
- Proof by contradiction
  - Suppose that G has a topological order $v_1, v_2, ..., v_n$ and that G also has a directed cycle C. Let's see what happens.
  - Let v_i be the lowest-indexed node in C and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
  - By our choice of i, we have $i < j$.
  - On the other hand, since (v_j, v_i) is an edge and $v_1, v_2, ..., v_n$ is a topological order, we must have $j < i$, a contradiction.



Directed Acyclic Graphs (DAG)

- If G has a topological order, then G is a DAG.
  - Q. Does every DAG have a topological ordering?
  - Q. If so, how do we compute one?

- If G is a DAG, then G has a topological ordering.
  - If G is a DAG, then G has a node with no entering edges.

G is a DAG $\Leftrightarrow$ G has a topological ordering

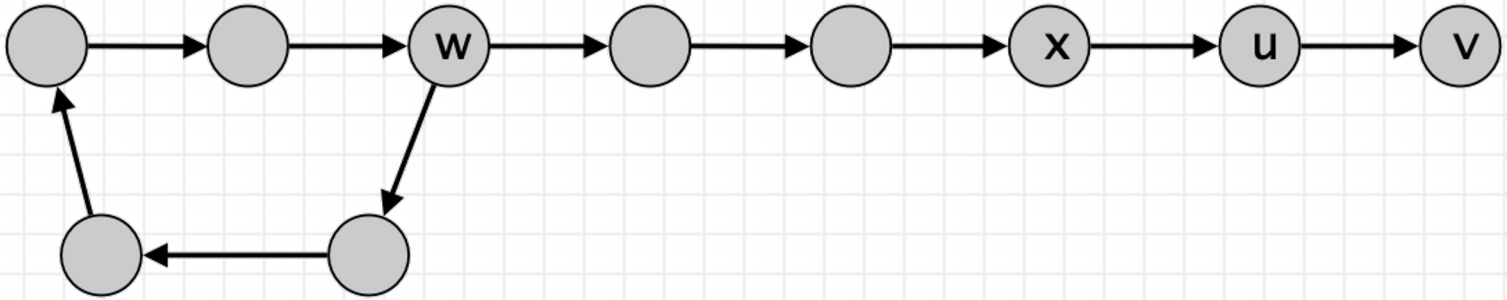
- Algorithm finds a topological order in $O(m + n)$ time.

Directed Acyclic Graphs (DAG)

- If G is a DAG, then G has a topological ordering.
  - If G is a DAG, then G has a node with no entering edges. (first, we need this!)
  - Proof by contradiction
    - Suppose that G is a DAG, and every node has at least one entering edge. Let's see what happens.
    - Pick any node v, and begin following edges backward from v. Since v has at least one entering edge (u, v) we can walk backward to u.
    - Then, since u has at least one entering edge (x, u), we can walk backward to x.
    - Repeat until we visit a node, say w, twice.
    - Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. Contradiction!

Directed Acyclic Graphs (DAG)

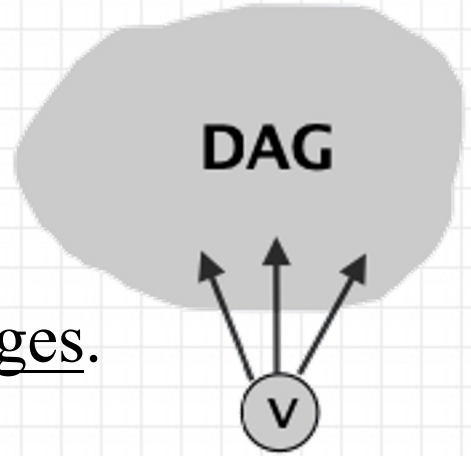
- If G is a DAG, then G has a topological ordering.
  - If G is a DAG, then G has a node with no entering edges. (first, we need this!)



  - Proof by contradiction
    - Suppose that G is a DAG, and <u>every</u> node <u>has at least one</u> entering edge. Let's see what happens.
    - Pick any node v, and begin following edges backward from v. Since v has at least one entering edge (u, v) we can walk backward to u.
    - Then, since u has at least one entering edge (x, u), we can walk backward to x.
    - Repeat until we visit a node, say w, twice.
    - Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. <u>Contradiction!</u>

Directed Acyclic Graphs (DAG)

- If G is a DAG, then G has a topological ordering. (Now, we can prove this!)
  - ✓If G is a DAG, then G has a node with no entering edges



- Proof by induction
  - Base case: true if $n = 1$.
  - Given DAG on $n > 1$ nodes, find a node v with no entering edges.
  - $G - \{ v \}$ is a DAG, since deleting v cannot create cycles.
  - By inductive hypothesis, $G - \{ v \}$ has a topological ordering.
  - Place v first in topological ordering; then append nodes of $G - \{ v \}$ in topological order. This is valid since v has no entering edges.

Directed Acyclic Graphs (DAG)

- If G has a topological order, then G is a DAG.
  - Q. Does every DAG have a topological ordering?
  - Q. If so, how do we compute one?

- If G is a DAG, then G has a topological ordering.
  - If G is a DAG, then G has a node with no entering edges.

G is a DAG $\Leftrightarrow$ G has a topological ordering

- Algorithm finds a topological order (topological sort) in $O(m + n)$ time.

Directed Acyclic Graphs (DAG)

- If G has a topological order, then G is a DAG.
  - Q. Does every DAG have a topological ordering?
  - Q. If so, how do we compute one?

- If G is a DAG, then G has a topological ordering.
  - If G is a DAG, then G has a node with no entering edges.

G is a DAG $\Leftrightarrow$ G has a topological ordering

- Algorithm finds a topological order (topological sort) in $O(m + n)$ time.

Directed Acyclic Graphs (DAG)

- Algorithm finds a topological order in $O(m + n)$ time

- `TOPOLOGICAL-SORT`
  - `Call DFS to compute finishing times for each vertex v.`
  - `As each vertex is finished, insert it onto the front of a linked list`
  - `Return the linked list of vertices`
    `(Output vertices in order of decreasing finish times)`
- Intuition:
  - Ensures if (u,v) ∈ E, then f[v] < f[u]

Directed Acyclic Graphs (DAG)

- Algorithm finds a topological order in $O(m + n)$ time

- `TOPOLOGICAL-SORT`
  - `Call DFS to compute finishing times for each vertex v.`
  - `As each vertex is finished, insert it onto the front of a linked list`
  - `Return the linked list of vertices`

- Pf. (CLRS, Theorem 22.12)

Directed Acyclic Graphs (DAG)

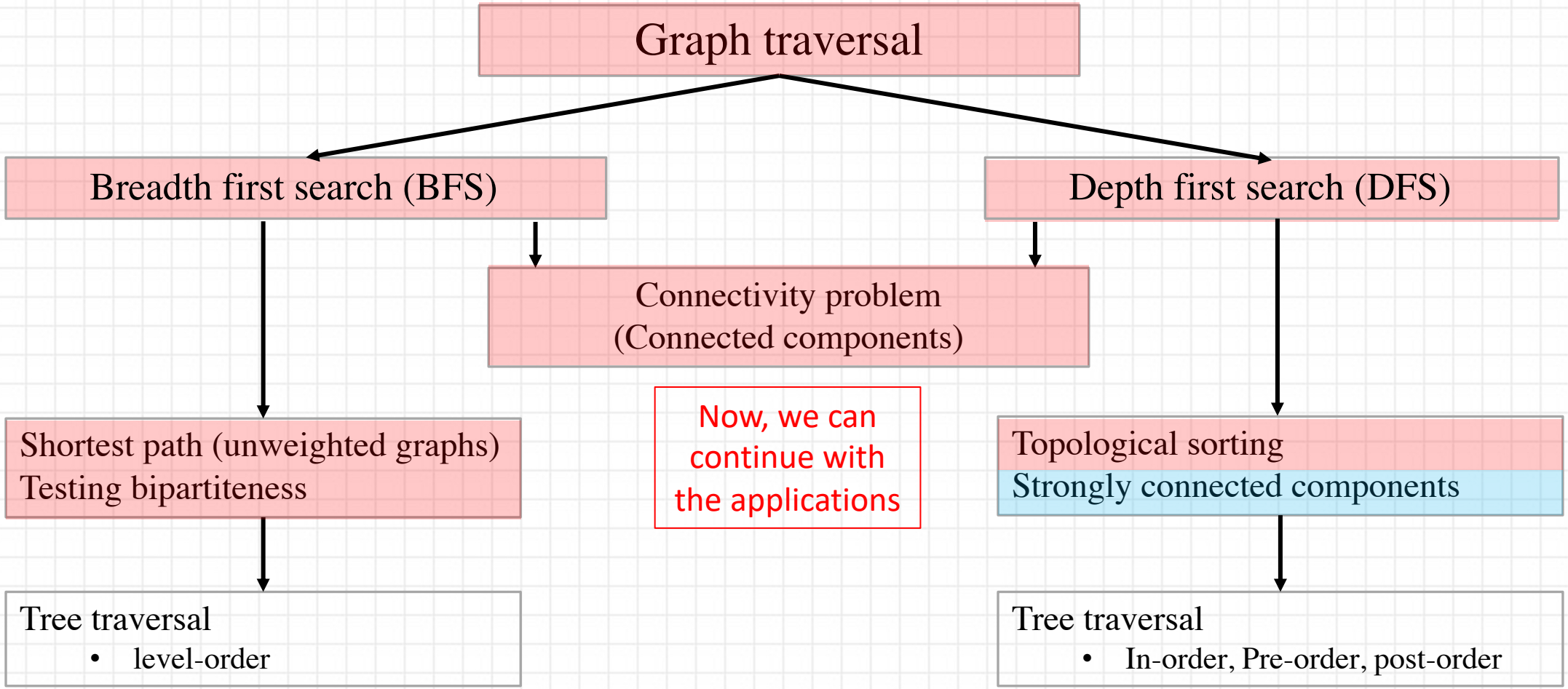
- Algorithm finds a topological order in $O(m + n)$ time

- `TOPOLOGICAL-SORT`
  - `Call DFS to compute finishing times for each vertex v.`
  - `As each vertex is finished, insert it onto the front of a linked list`
  - `Return the linked list of vertices`

- Pf. (CLRS, Theorem 22.12)

- Note topological ordering can also be obtained using "**Kahn's algorithm**", which is BFS approach starting from a node with no entering edge, in $O(m + n)$ time.

Graph


Graph traversal
Breadth first search (BFS)
Depth first search (DFS)
Connectivity problem
(Connected components)
Now, we can continue with the applications
Shortest path (unweighted graphs)
Testing bipartiteness
Topological sorting
Strongly connected components
Tree traversal
• level-order
Tree traversal
• In-order, Pre-order, post-order

Strongly Connected Component (SCC)

- Problem: Decomposing a directed graph into its strongly connected components
- A classic application of DFS.
  - The strongly connected components of a directed graph are the equivalence classes of vertices under the "are mutually reachable" relation.
  - Given directed graph G=(*V, E*) an SCC is a maximal set of vertices C ⊆ V such that for every pair of vertices u and v in C , we have both u⇝v and v⇝u; that is, vertices u and v are reachable from each other.
  - A directed graph is strongly connected if it has only one strongly connected component.

Strongly Connected Component (SCC)

- Linear-time ($\Theta(|V| + |E|)$) algorithm to compute the strongly connected components of a directed graph $\mathrm{G} = (V, E)$ using two depth-first searches, one on G and one on G^{T}.

- $\mathrm{G}^{\mathrm{T}} = (V, E^T)$, where $E^T = \{(u, v) | (v, u) \in E\}$
  In other words, same graph except all edges are reversed.
- Adjacency list representation: G^T can be obtained in $\mathrm{O}(|V| + |E|)$

Strongly Connected Component (SCC)

- Linear-time ($\Theta(|V| + |E|)$) algorithm to compute the strongly connected components of a directed graph $\mathrm{G} = (V, E)$ using two depth-first searches, one on G and one on G^{T}.

- $\mathrm{G}^{\mathrm{T}} = (V, E^T)$, where $E^T = \{(u, v) | (v, u) \in E\}$
  In other words, same graph except all edges are reversed.
- Adjacency list representation: G^T can be obtained in $\mathrm{O}(|V| + |E|)$

- **Observation:** G and G^{T} have the same SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^{T}.)

Strongly Connected Component (SCC)

- Linear-time ($\Theta(|V| + |E|)$) algorithm to compute the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^T.

```
STRONGLY-CONNECTED-COMPONENTS (G)
1. Call DFS(G) to compute finishing times for each vertex u (u.f)
2. Compute G^T
3. Call DFS(G^T) but in the main loop of DFS, consider the vertices
   in order of decreasing u.f (as computed in line 1)
4. Output the vertices of each tree in the depth-first forest
   formed in line 3 as a separate strongly connected component
```

Strongly Connected Component (SCC)

- Linear-time ($\Theta(|V| + |E|)$) algorithm to compute the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^T.

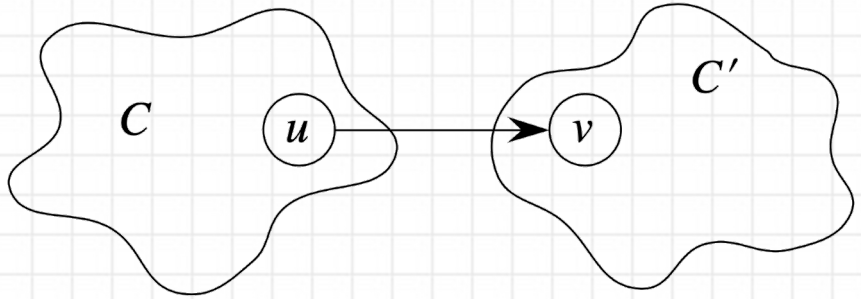
```
STRONGLY-CONNECTED-COMPONENTS (G)
1. Call DFS(G) to compute finishing times for each vertex u (u.f)
2. Compute G^T
3. Call DFS(G^T) but in the main loop of DFS, consider the vertices
   in order of decreasing u.f (as computed in line 1)
4. Output the vertices of each tree in the depth-first forest
   formed in line 3 as a separate strongly connected component
```

Why does this algorithm work?

Strongly Connected Component (SCC)

```
STRONGLY-CONNECTED-COMPONENTS (G)
1. DFS(G) to compute f(u)
2. Compute G^T
3. Call DFS(G^T) in the order of decreasing f(u) (topology ordering of G)
4. Each tree in the depth-first forest formed in line 3 is a strongly connected component
```

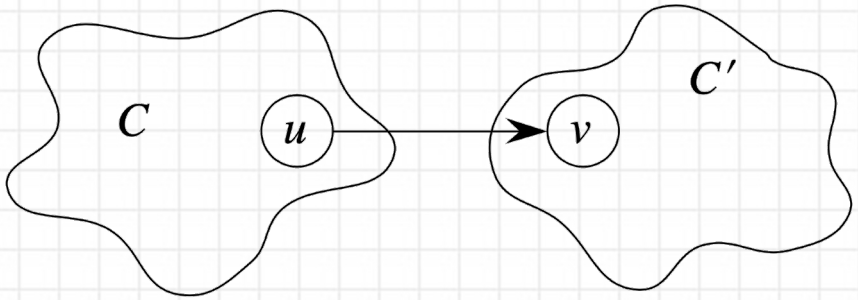
- Considering vertices in second DFS in decreasing order of finishing times from first DFS means we are visiting vertices of the component graph in topological sort order.
- **Lemma 1**
  Let C and C' be distinct SCCs in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.



Strongly Connected Component (SCC)

```
STRONGLY-CONNECTED-COMPONENTS (G)
1. DFS(G) to compute f(u)
2. Compute G^T
3. Call DFS(G^T) in the order of decreasing f(u) (topology ordering of G)
4. Each tree in the depth-first forest formed in line 3 is a strongly connected component
```

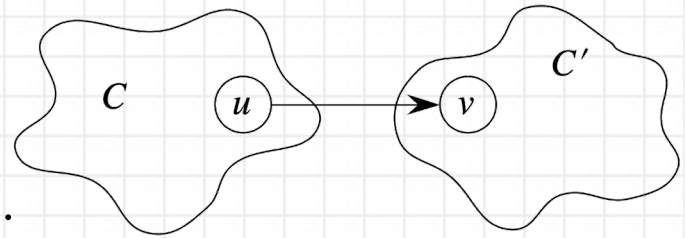
- **Lemma**
  Let C and C' be distinct SCCs in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.



- **Corollary-1** Suppose there is an edge $(u,v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.
- **Corollary-2** Suppose $f(C) > f(C')$. Then, there cannot be an edge from C' to C in G^T.

Strongly Connected Component (SCC)

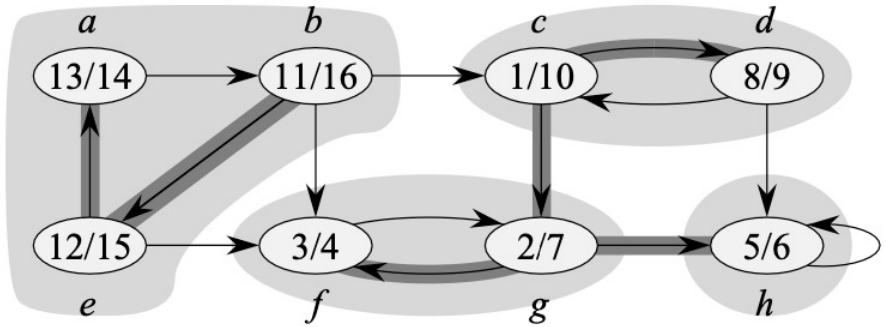
- **When we start the second DFS on G^T:**
  - We begin with SCC C such that f(C) is maximum.
  - So, the second DFS starts from some $x \in C$, which visits all C vertices.
  - Corollary-2 says that since $f(C) > f(C')$ for all $C' \neq C$, there are no edges from C to C' in G^T. Therefore, the second DFS only visits vertices in C, i.e., the depth-first tree rooted at x contains *exactly* the vertices of C.

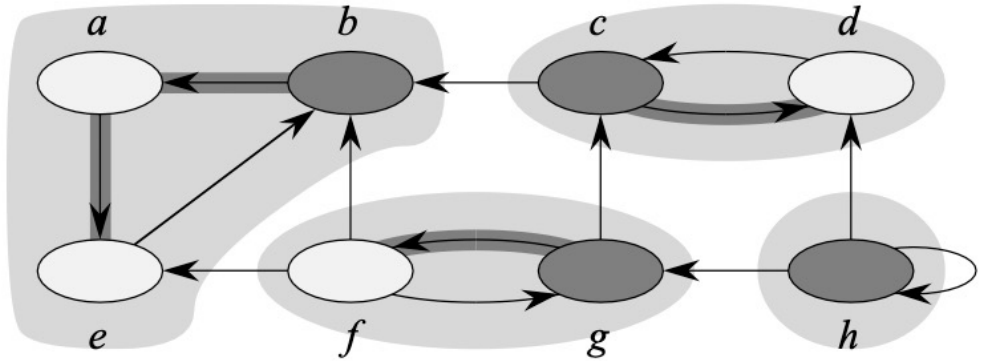


- **The next root chosen in the second DFS** is in SCC C' such that $f(C')$ is maximum over all SCCs other than C. DFS visits all vertices in C', but the only edges out of C' go to C, which we have already visited. Therefore, the only tree edges will be to vertices in C'.
- **We can continue the process**. Each time we choose a root based on the topological order, where we have only edges to the current SCC nodes (and the earlier ones but they are already visited), and there is no edge to the next SCC (Corollary-2); therefore, the DFS only visits the current SCC nodes.

Strongly Connected Component (SCC)

- Ex.



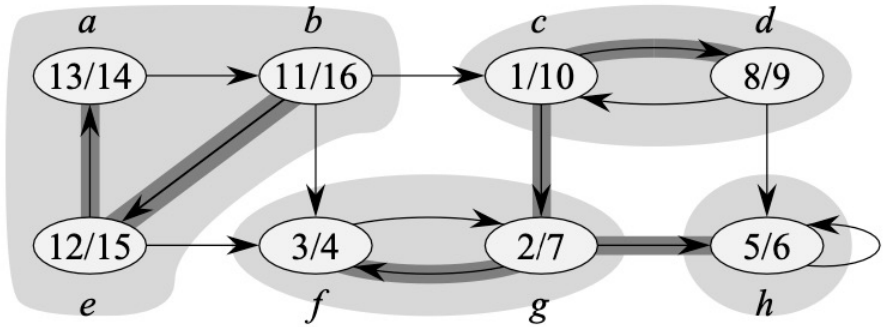


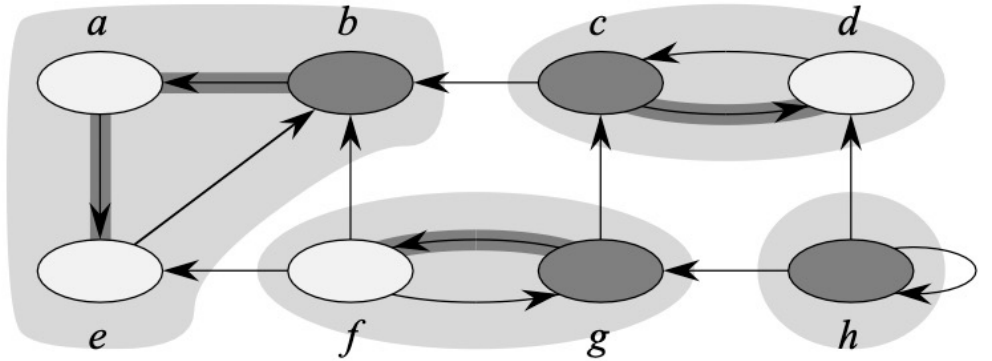
- A directed graph G.
- Each shaded region is a strongly connected component of G.
- Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded.

- The graph G^T, the transpose of G, with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded.
- Each strongly connected component corresponds to one depth-first tree.
- Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of GT.

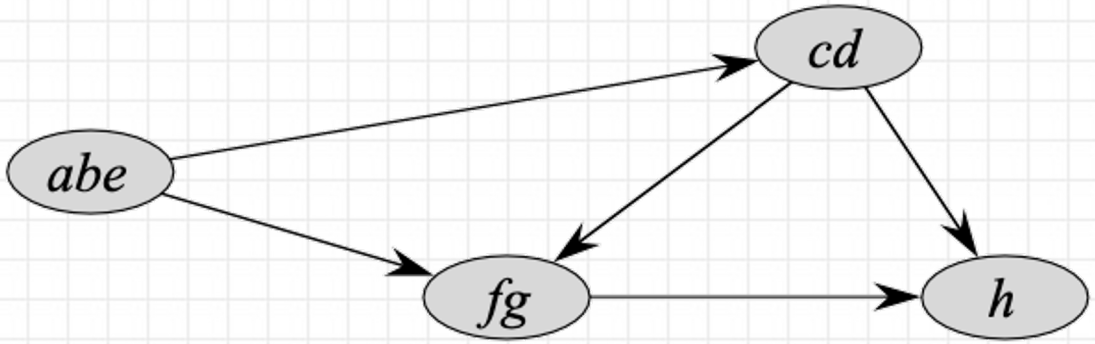
Strongly Connected Component (SCC)

- Ex.

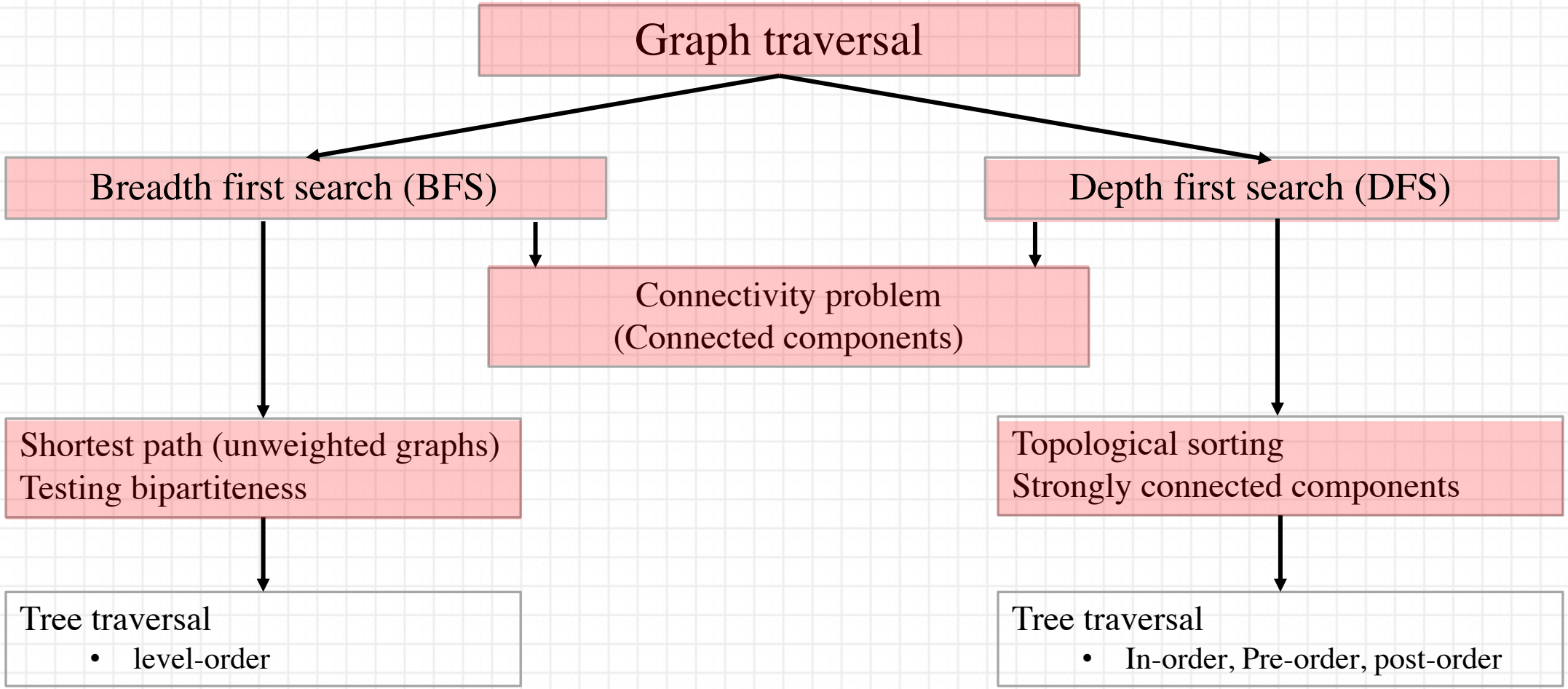




The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.



Graph


Graph traversal
Breadth first search (BFS)
Depth first search (DFS)
Connectivity problem
(Connected components)
Shortest path (unweighted graphs)
Testing bipartiteness
Topological sorting
Strongly connected components
Tree traversal
• level-order
Tree traversal
• In-order, Pre-order, post-order

Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list

- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components

- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components
- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

References

- The lecture slides are mainly based on the suggested textbooks and the corresponding published lecture notes:
  - CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
  - KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.
  - DPV: Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms, McGraw-Hill Higher Education., 2008.
  - Slides by Kevin Wayne. Copyright © 2005 Pearson-Addison Wesley.