CS-3510:
Design and Analysis of Algorithms

Exam 2: Review

Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

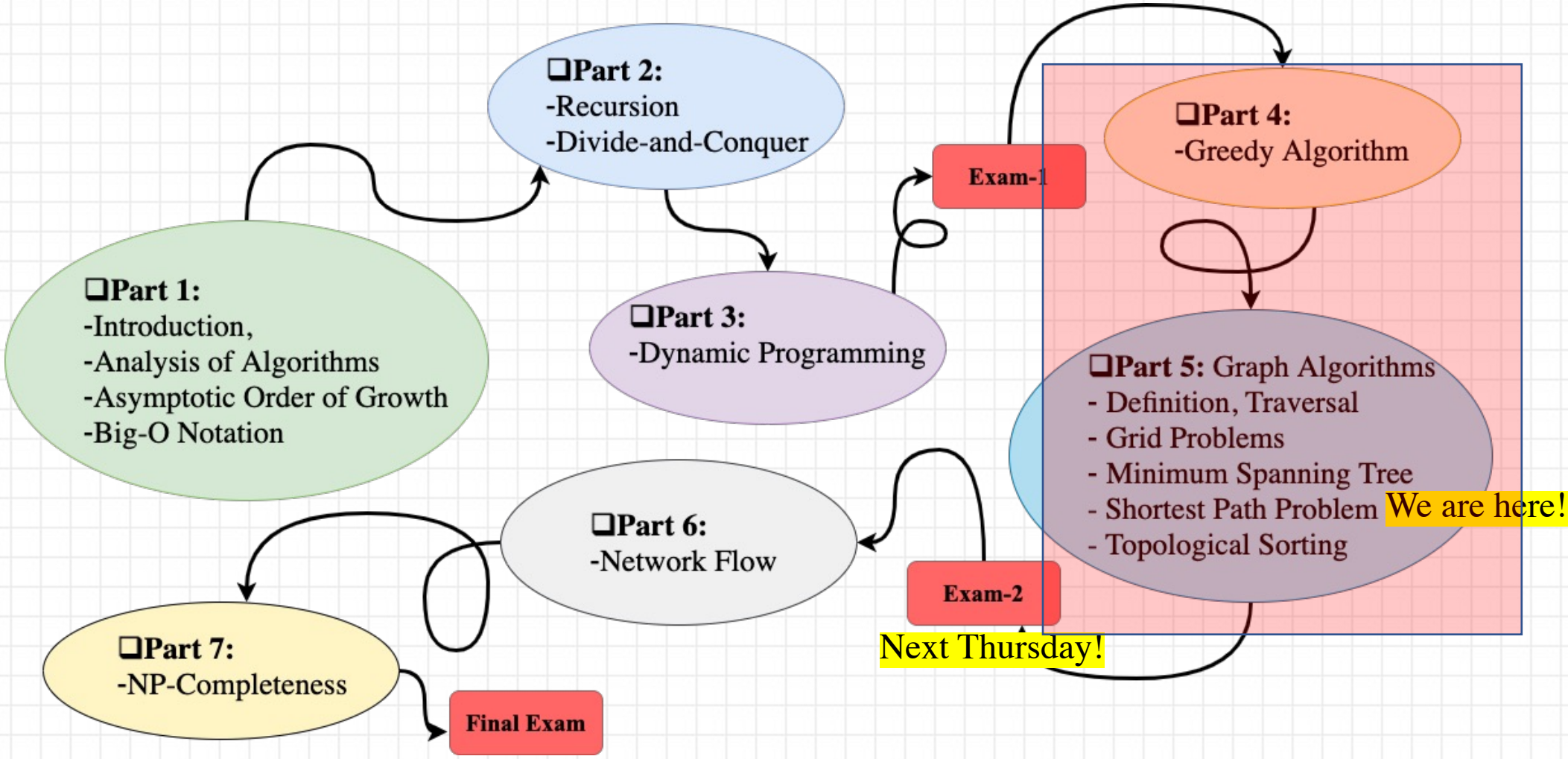
Exam 2

- Date: Thursday, July 07, 2022
- Time: 03:30 pm – 05:00 pm
- Location: Klaus 2443

No class on Tuesday (07/05)!

- Closed book; No calculator
- One page sheet of notes
  - Letter size
  - Both sides
  - Typed or hand-written

Roadmap


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

Exam 2

- Contents:
  - Greedy algorithms
  - Graph algorithms
    - Definition and representation
    - Graph traversal (BFS, DFS)
    - Graph traversal applications
    - Minimum spanning tree
    - ~~Shortest path in weighted graph~~ (→ Final exam)

Exam 2: Greedy Algorithms

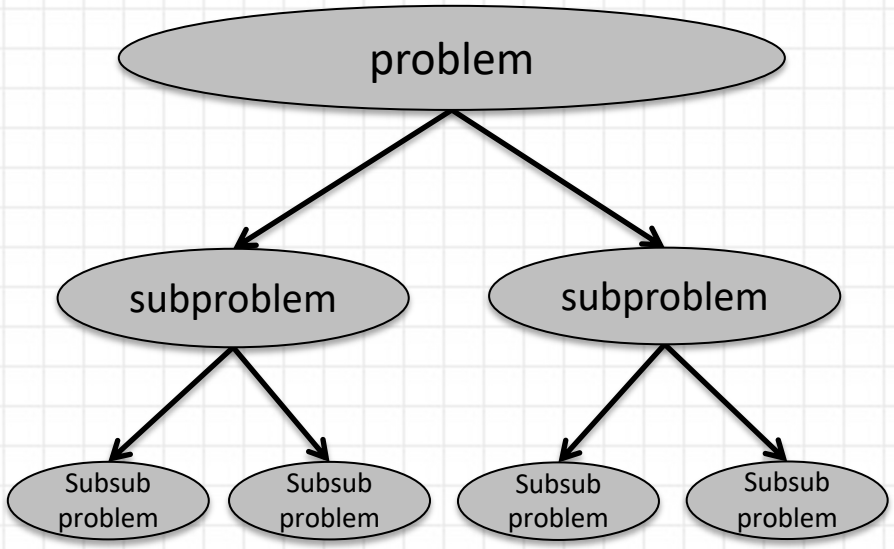
- Build the solution step-by-step
- At each step, make a decision that is locally optimal
- Never look back and hope for the best!
- Do NOT always yield optimal solutions, but for many problems they do

Exam 2: Greedy Choice Property

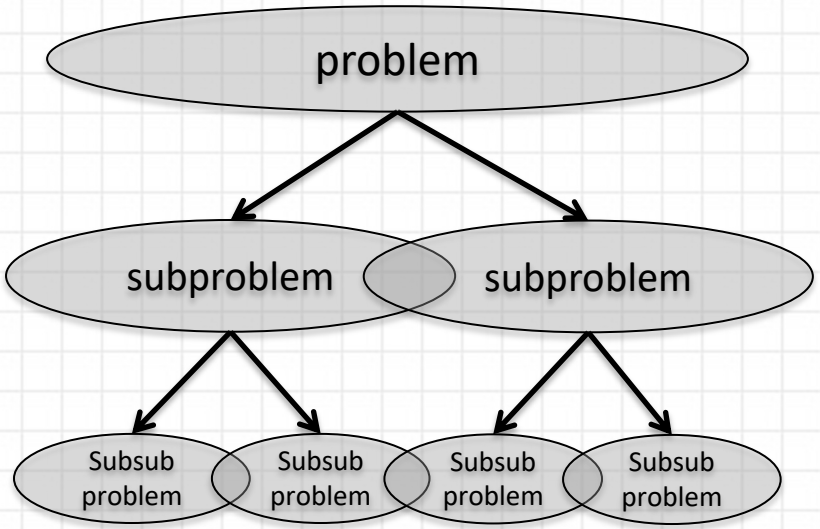
- Greedy choice = locally optimal choice
- Greedy-choice property: we can assemble a globally optimal solution by making locally optimal choices.
- In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.
  (The main difference with dynamic programming)
- Make whatever choice seems best at the moment and then solve the subproblem that remains.
- Makes its first choice before solving any subproblems.

Exam 2: Greedy Algorithms

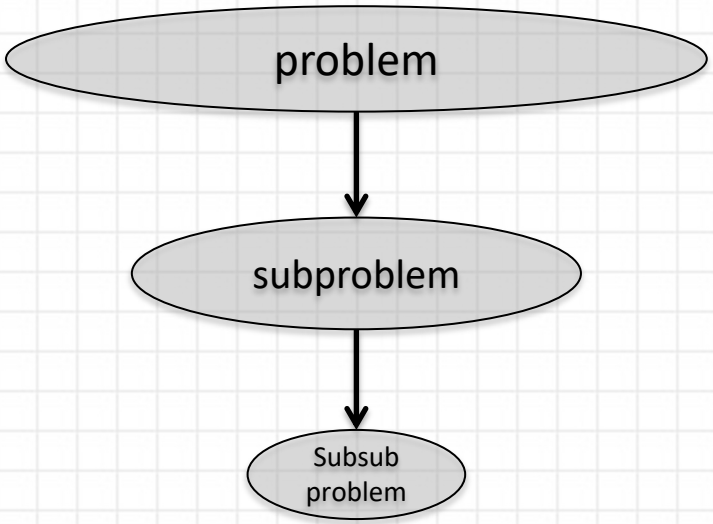
Divide-and-Conquer



Dynamic Programming

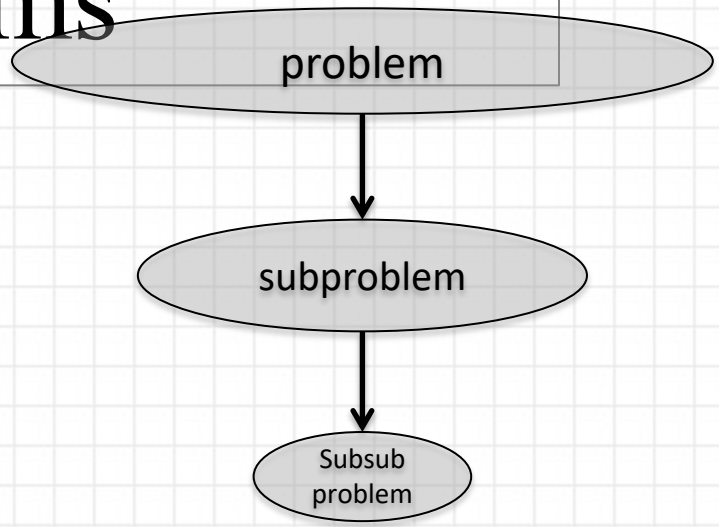


Greedy Approach



Optimal substructure
But only one subproblem

Exam 2: Greedy Algorithms



- Seems "easier" than dynamic programming?
- Two major "questions/problems":
  - What is the best/correct greedy choice to make?
  - How can we prove that the greedy algorithm yields an optimal solution?
- When is using the greedy approach a good idea?
  - Greedy can be optimal when the problem shows an especially nice optimal substructure.

Exam 2: Greedy Algorithms

- Examples
  - Interval scheduling (activity selection)
  - Interval partitioning
  - Schedule to minimize lateness
  - …

Type of Questions in Exam-2:
- Short answers
- Definition
- True/False questions

- Applications in Graph (next week)
  - Kruskal's algorithm (minimum spanning tree)
  - Prim's algorithm (minimum spanning tree)
  - Dijkstra's algorithm (shortest path)

Exam 2: Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list

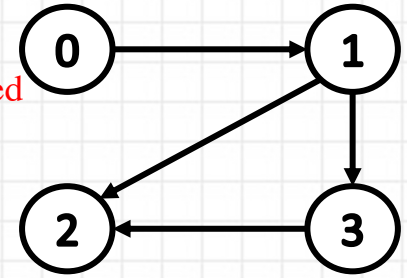
- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected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

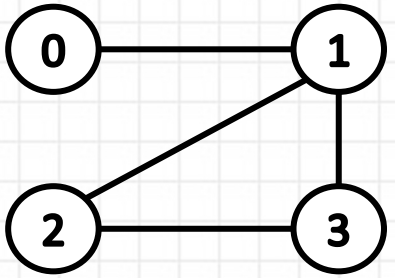
- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem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

Graph Properties and Terminology Review

- Notation. $G = (V, E)$
  - V = nodes (or vertices). {0, 1, 2, … n-1}
  - E = edges (or arcs) between pairs of nodes. $\{e_1, e_2, \ldots e_m\}$ where $e_i = (v_i, v_j)$
  - Captures pairwise relationship between objects.
- Directed vs. undirected
  - Directed graph = digraph

Directed
Unweighted

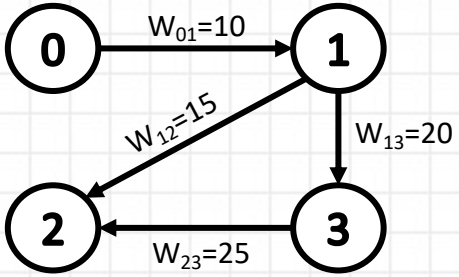


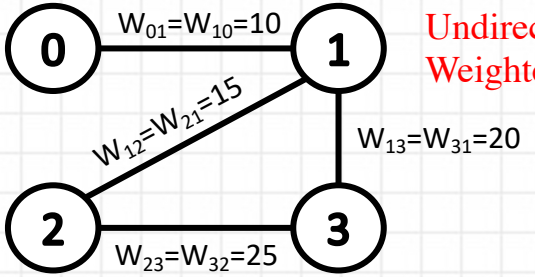


Undirected
Unweighted

- Weighted vs. unweighted

Directed
Weighted





Undirected
Weighted

Graph Representation: Summary

- Two common ways to represent graphs
  - Adjacency matrix
  - Adjacency list
- Adjacency matrix
  - Space: n^2 elements for n vertices
  - Easy to check if a link exists between two vertices
- Adjacency list
  - More common representation: most large real-world graphs are sparse
  - Space: Number of edges [2*(number of edges) if undirected] + number of vertices, i.e., (m+n) or (2m+n)
  - Linked list implementation is typically used

Graph Definitions and Terminology: Summary

- Paths and connectivity
- Connected graph, connected component
- Cycle
- DAG
- Bipartiteness
- Trees
- …

Type of Questions in Exam-2:
- Short answers
- Definition
- True/False questions
- Graph representation

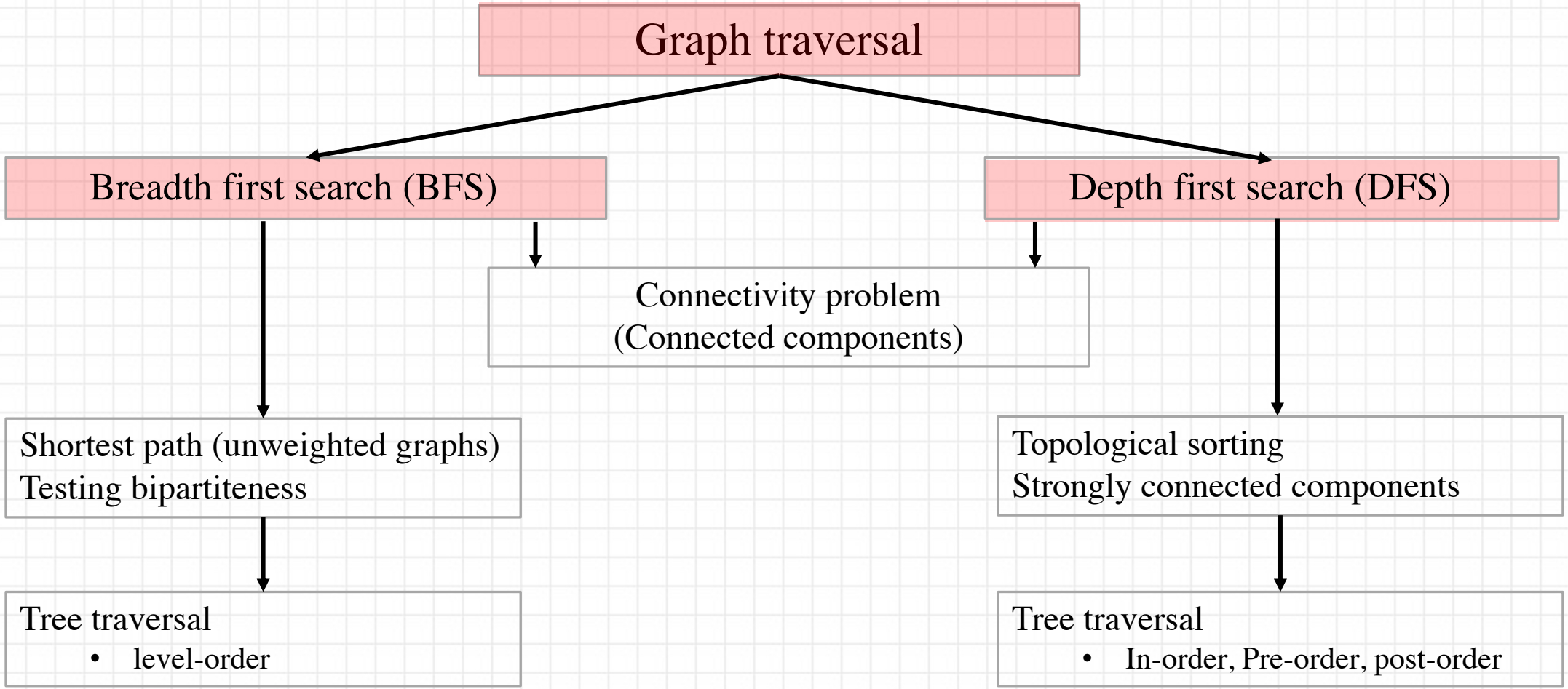
Exam 2: Graph

- 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 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 Traversal

- Traversal = Exploring = Searching
- A graph needs to be traversed in order to determine some properties

- 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

	Implementation	**Data Structure**
BFS	Iterative	Queue (FIFO)
DFS	Recursive	(not explicitly required → execution stack)
	Iterative	Stack (LIFO)

Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v
- Iterative implementation.
- Needs queue data structure

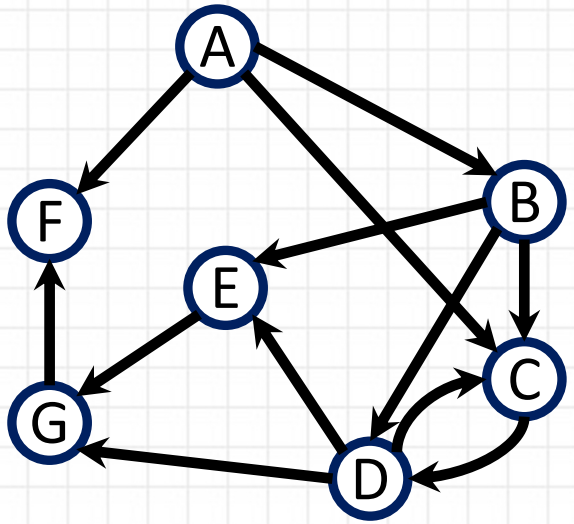
- Traversal = Exploring = Searching
  (visiting vertices one-by-one)

Graph Traversal: BFS

- BFS runs in O(|V| + |E|) time
- The worst case is when the graph is connected.
  - Each vertex is added to the queue and removed from it exactly once
  - Each adjacency list is used exactly once

Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex "s"
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v




```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

white := unvisited node

distance from source

parent

gray := visited node

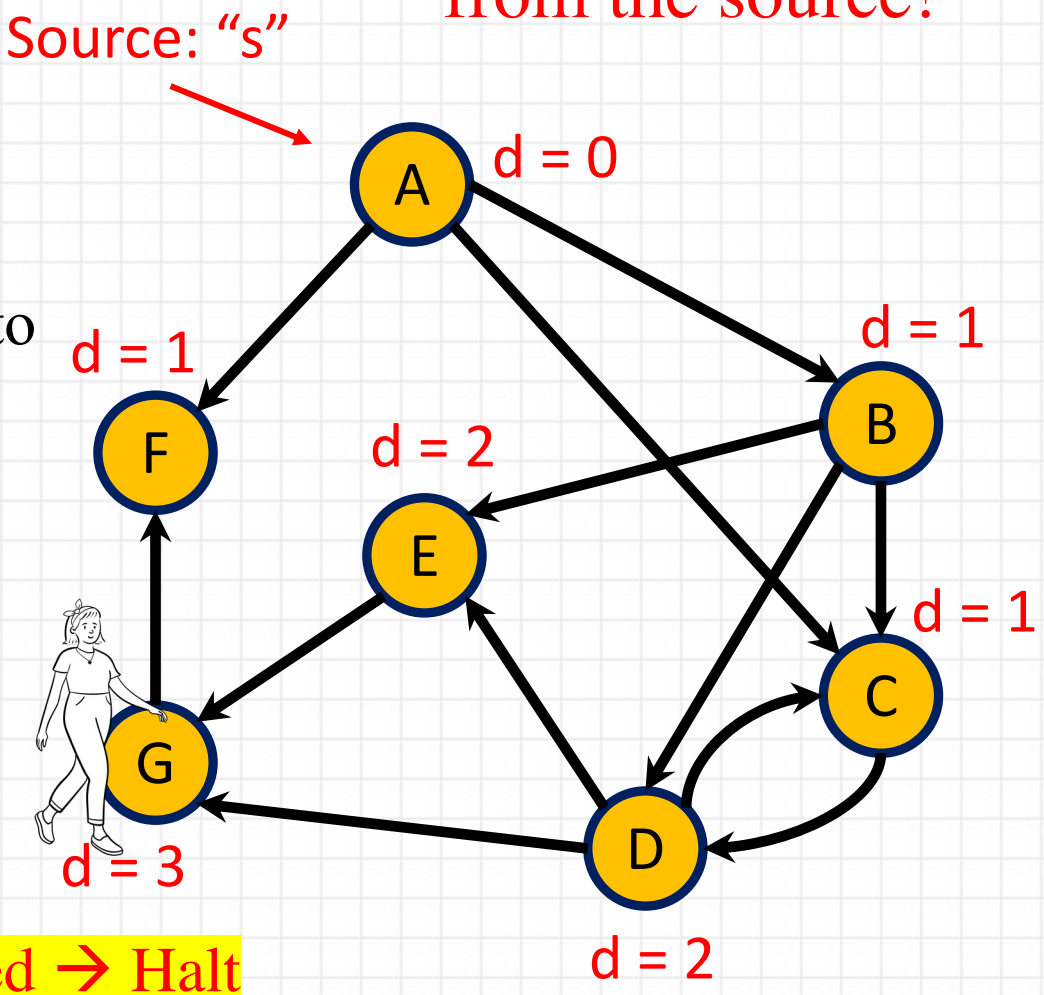
black := visited & all unvisited neighbors added to the queue

Graph Traversal: BFS

Note d always presents the "shortest distance" from the source!

- An efficient graph traversal procedure
- BFS starts from a source vertex "s"
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

- Queue = {~~A~~, ~~B~~, ~~C~~, ~~F~~, ~~D~~, ~~E~~, ~~G~~}
- Visited = {A, B, C, F, D, E, G}



Nothing left in the queue → All nodes are visited → Halt

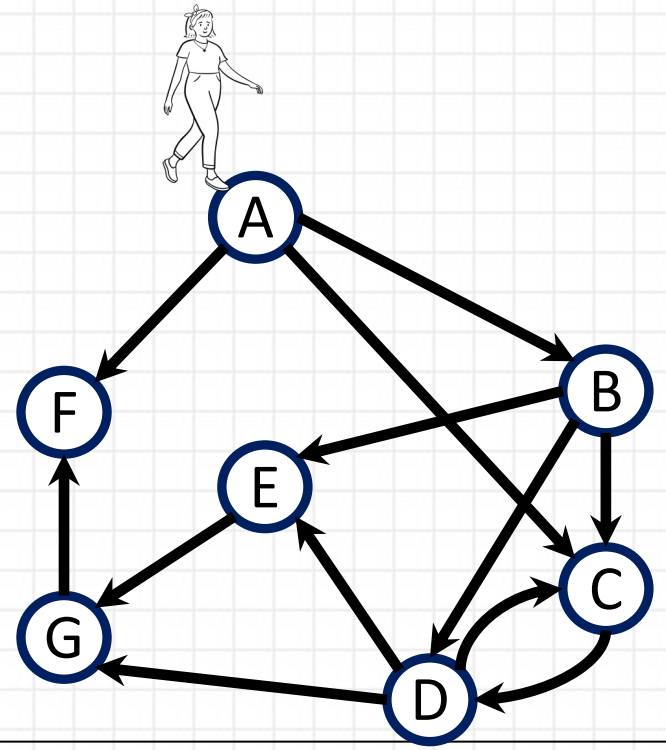
Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path.
- No explicit storage of vertices is required (BFS needs a queue)
- However, calls for each vertex build up on the execution stack (recursive implementation)
- An iterative implementation is possible using an explicit stack data structure.

- Traversal = Exploring = Searching
  (visiting vertices one-by-one)

Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path




```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

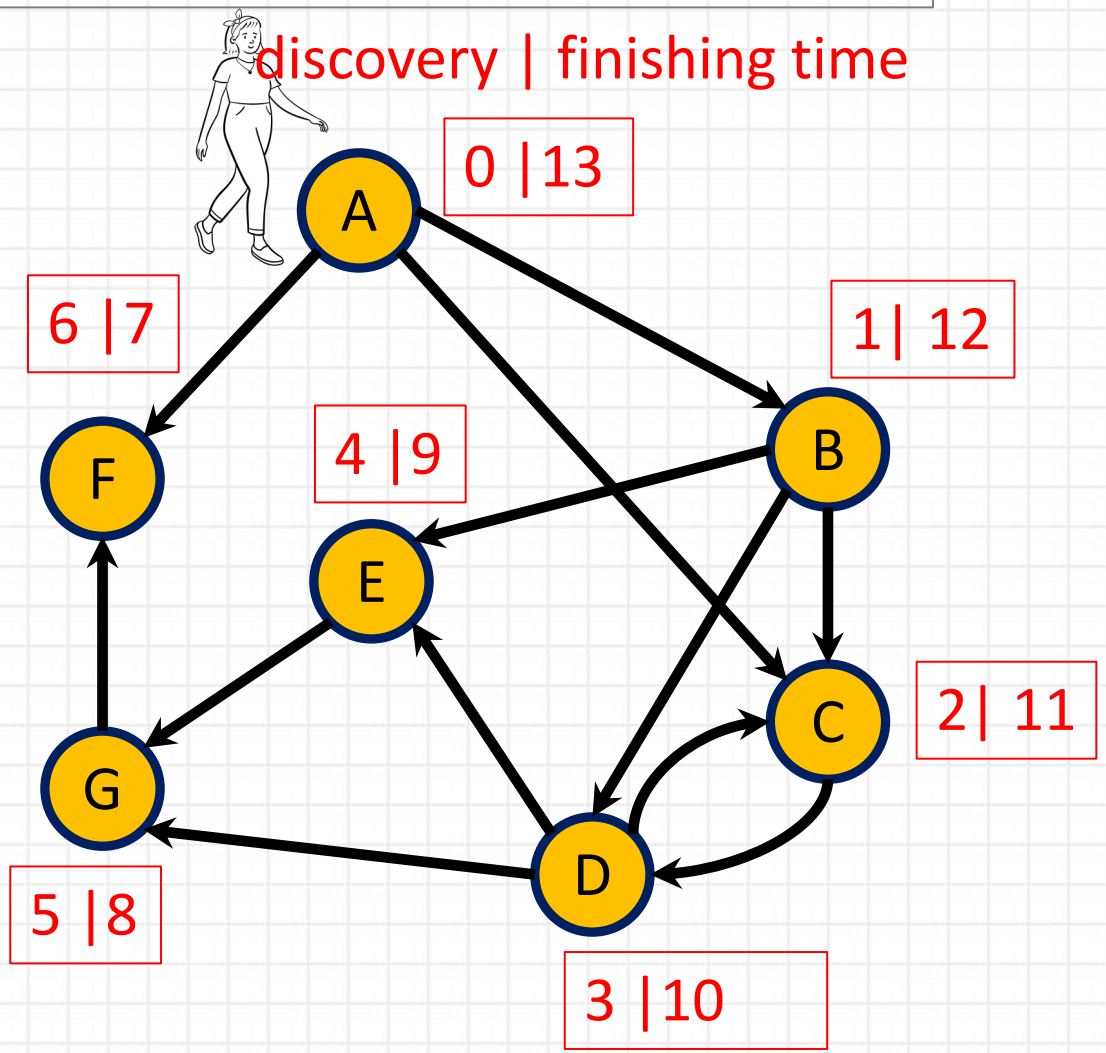
```
DFS-VISIT(G, u)
1   time = time + 1                  // white vertex u has just been discovered
2   u.d = time
3   u.color = GRAY
4   for each v ∈ G.Adj[u]            // explore edge (u, v)
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   u.color = BLACK                  // blacken u; it is finished
9   time = time + 1
10  u.f = time
```

Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

Pop

- Stack = {~~A, B, C, D, E, G, F~~}
- Visited = {A, B, C, D, E, G, F}
- No more path to explore→ backtrack
- No more element in the stack → Halt



Graph Traversal: DFS

- DFS also runs in O(|V| + |E|) time
- DFS is called exactly once per vertex
- Each adjacency list is used exactly once

	Implementation	**Data Structure**	**Running Time**
BFS	Iterative	Queue (FIFO)	O(|V| + |E|)
DFS	Recursive Iterative	(not explicitly required → execution stack) Stack (LIFO)	O(|V| + |E|)

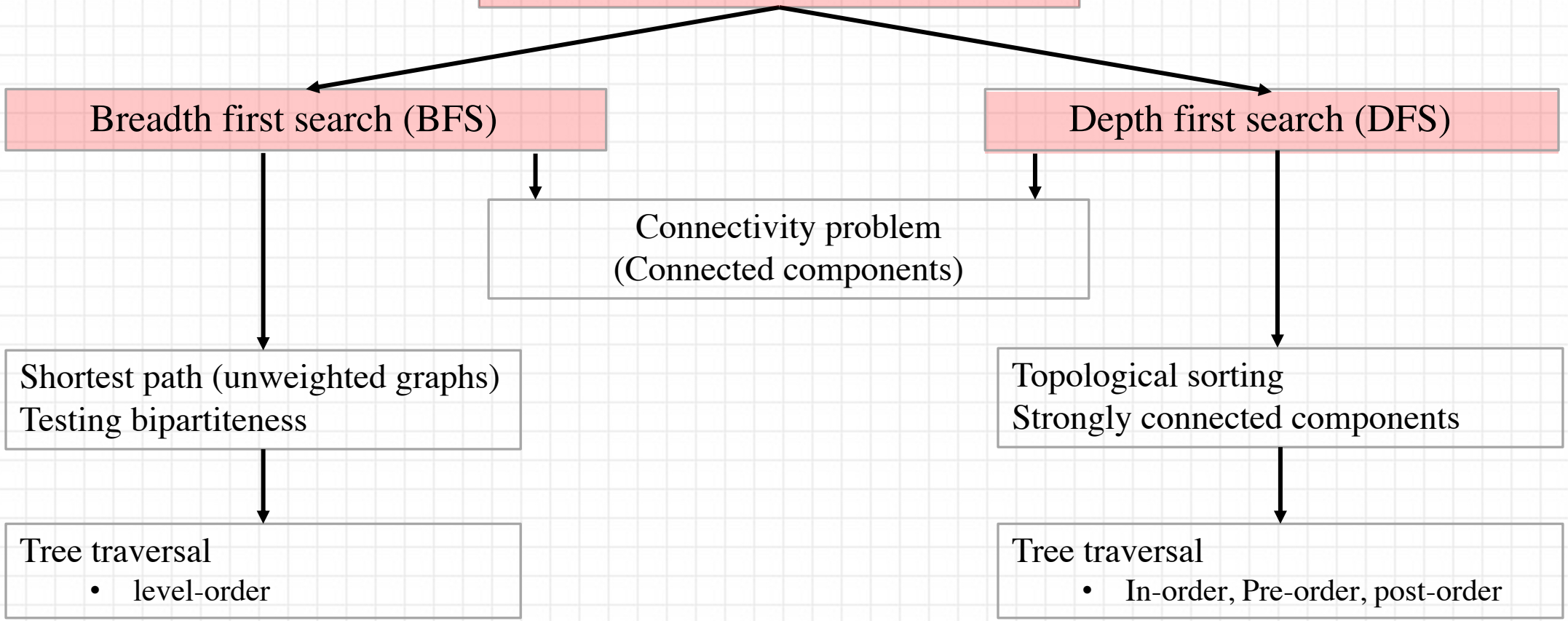
BFS and DFS

- Both are graph traversal algorithms

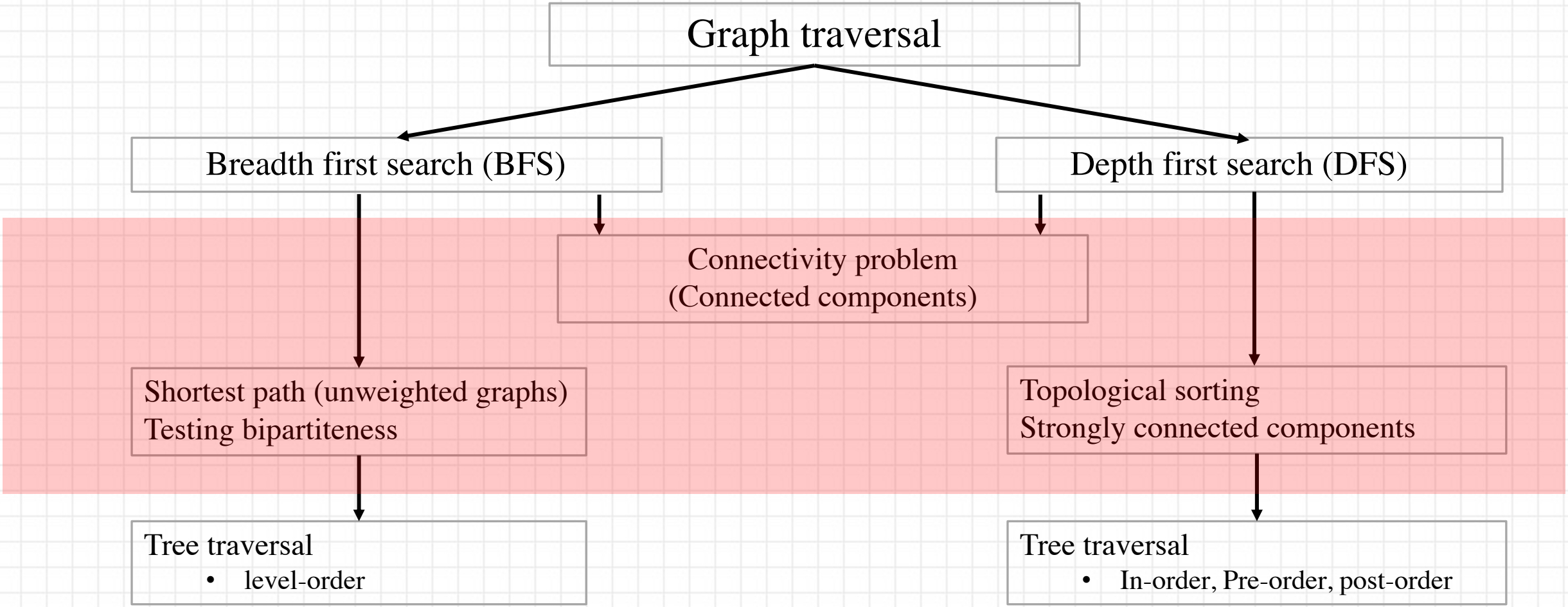
BFS	DFS
Iterative: Queue (FIFO), Time:O(|V| + |E|), Space: O(|V|)	Recursive: (execution stack), Iterative: Stack(LIFO) Time:O(|V| + |E|), Space: O(|V|)
➢ **BFS builds a breadth-first tree as it searches the graph.** ➢ **We can print out the vertices on a shortest path from s to v, using the BFS tree** ➢ **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.**

Type of Questions in Exam-2:

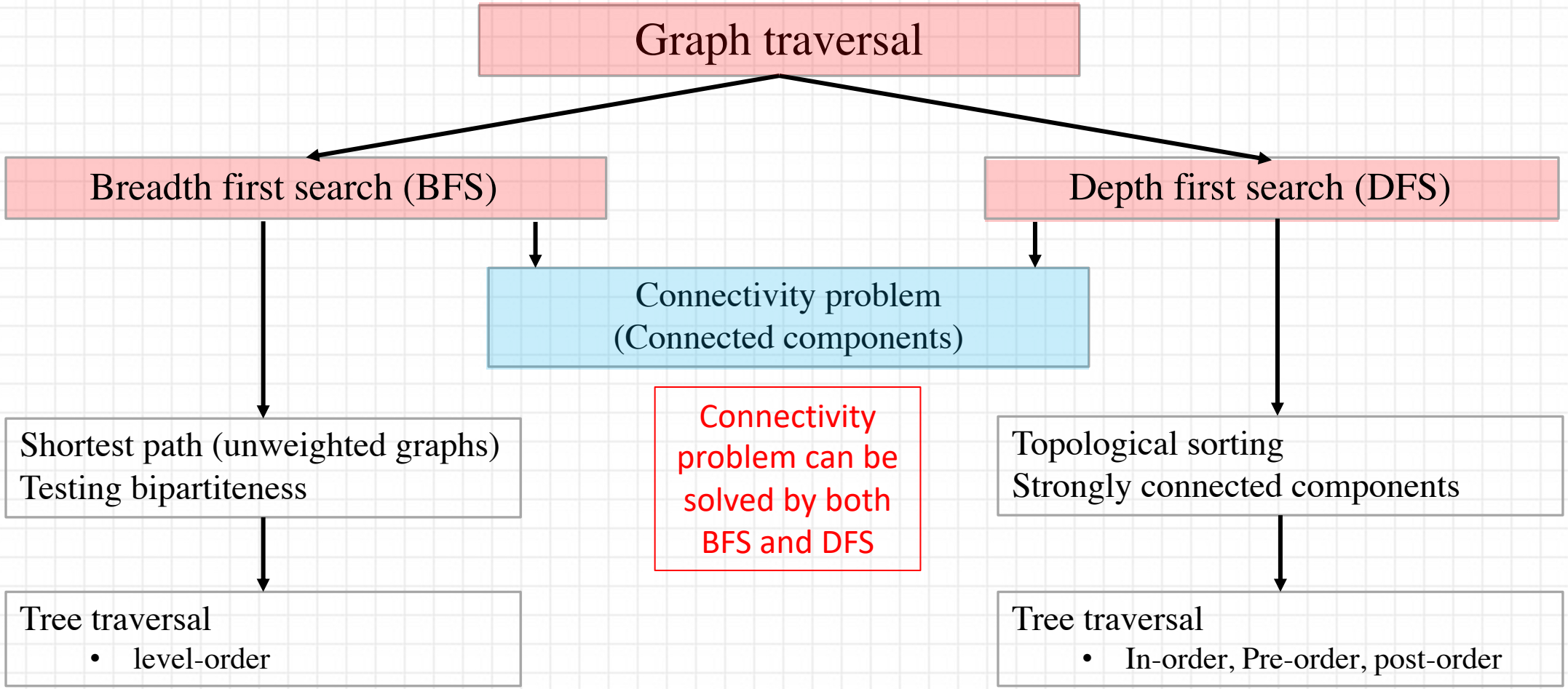
- Short answers /Definition/ True/False
- Running BFS/DFS on a given graph (show your steps)
- BFS/DFS trees, discovery/finishing times, …



Exam 2: Graph traversal applications



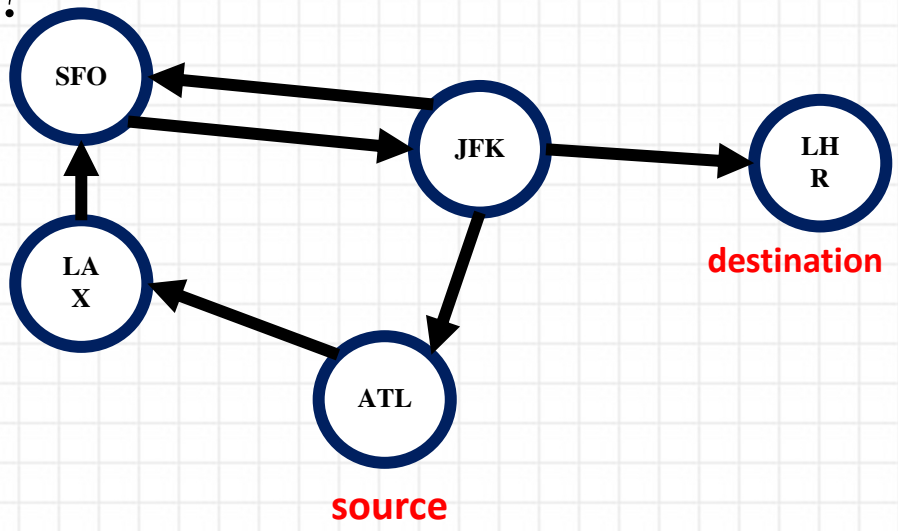
Exam 2: Graph


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.

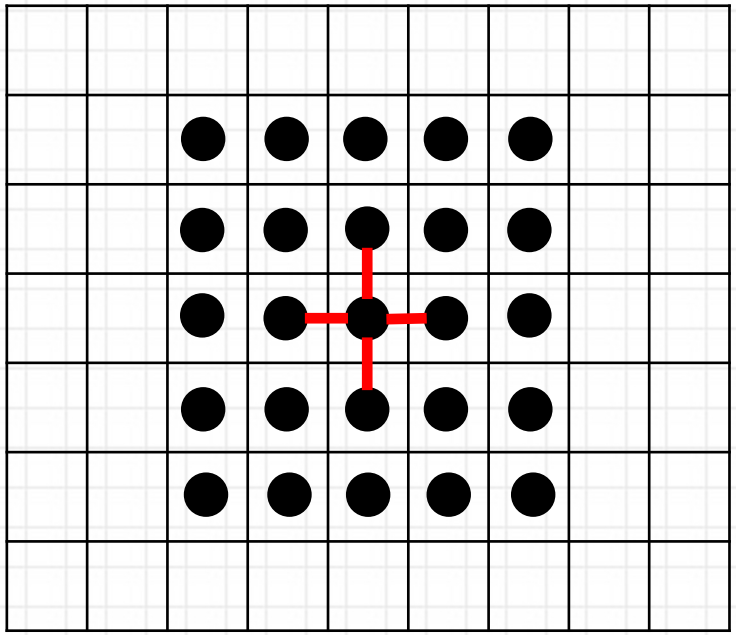
We know the nodes (= grid cells) and we know the neighbors (the relationship), so we can skip the graph definition part!

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]

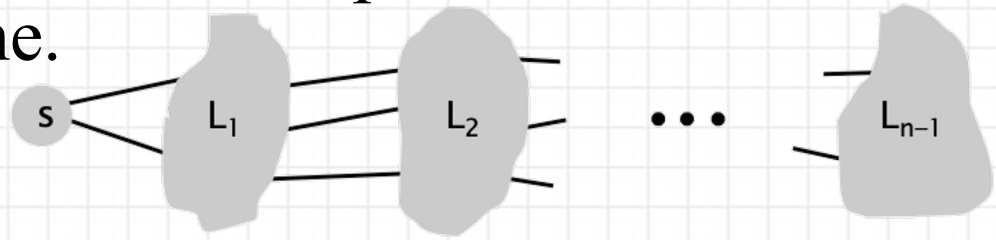
grid =



 Demo code!

BFS: Shortest paths

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



BFS algorithm.

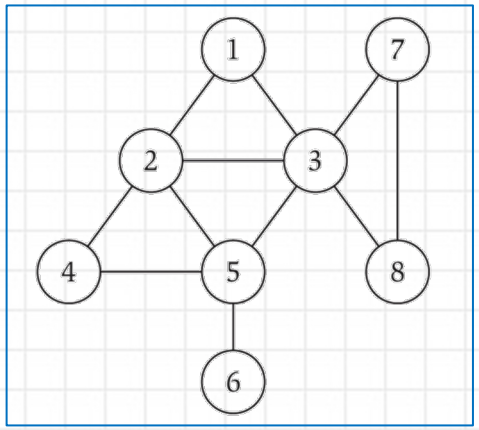
- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

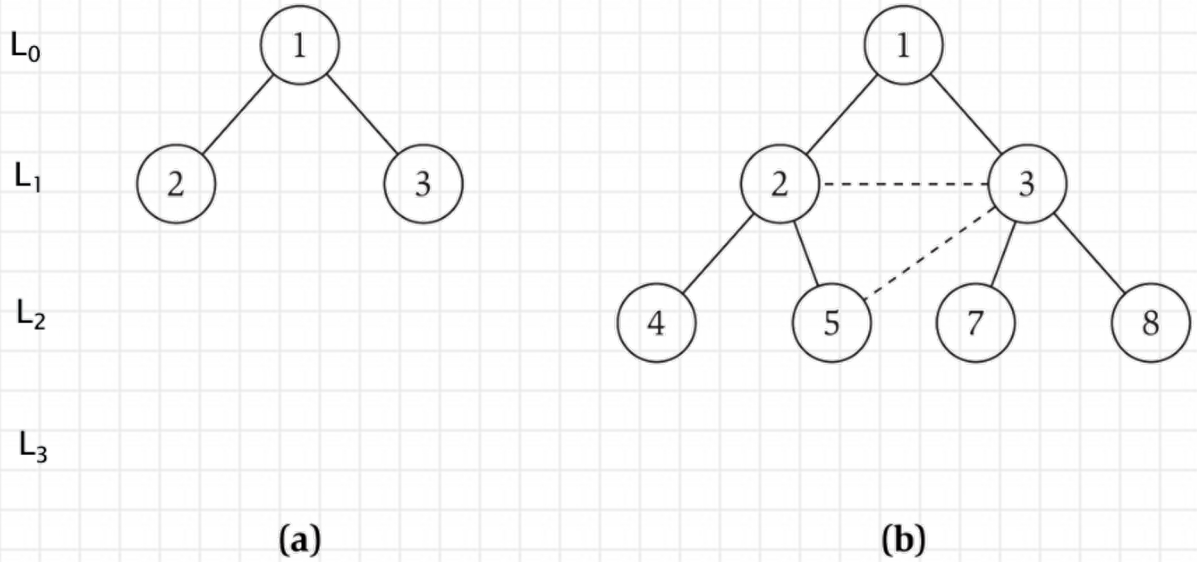
- Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t if and only if t appears in some layer.

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.





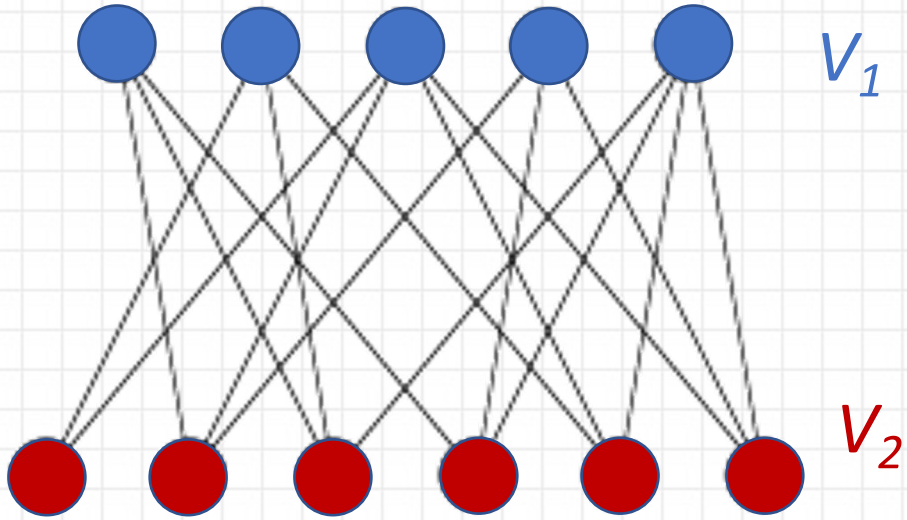
BFS: Shortest paths

- **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: 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.

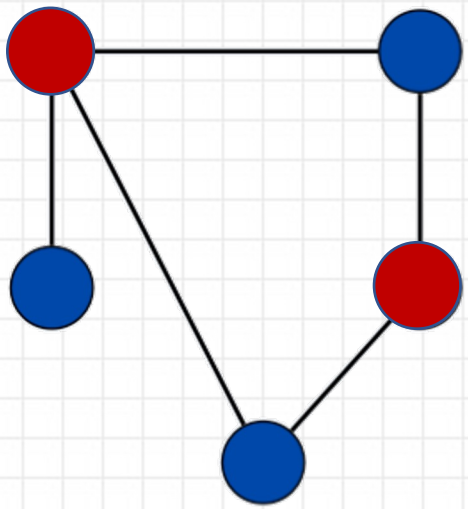
bipartite graph = 2-colorable graph



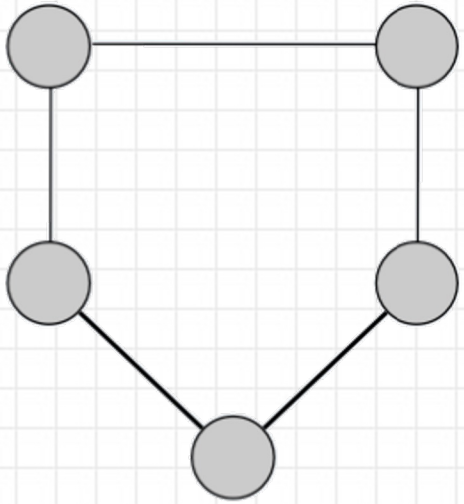
- Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored blue or red such that every edge has one blue and one red end.

BFS: Testing Bipartiteness (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.



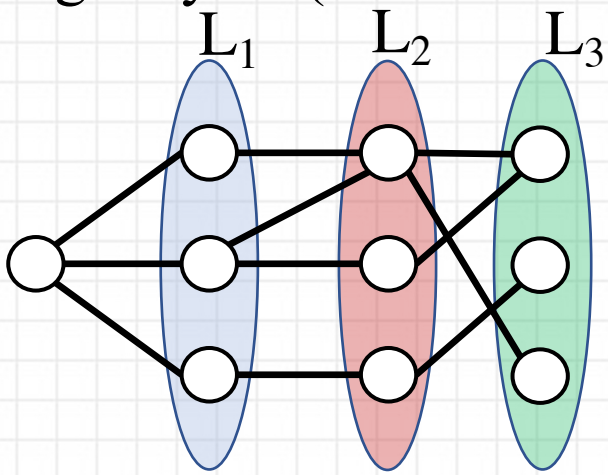
bipartite
(2-colorable)



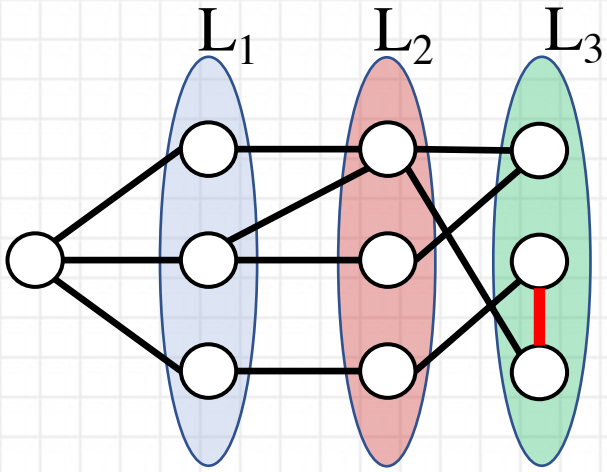
not bipartite
(not 2-colorable)

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

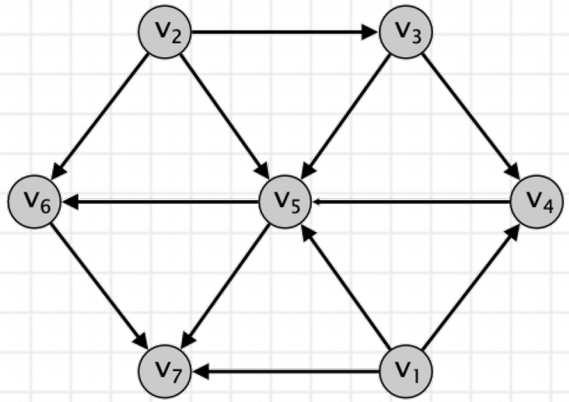
- We can modify the BFS algorithm to color each neighbor with the opposite color when it explores a node.
- If a neighbor has already been colored (i.e., visited), and has the same color, then return false.
- If the BFS can traverse the entire graph and color all nodes, then return true.

Directed Acyclic Graphs (DAG)

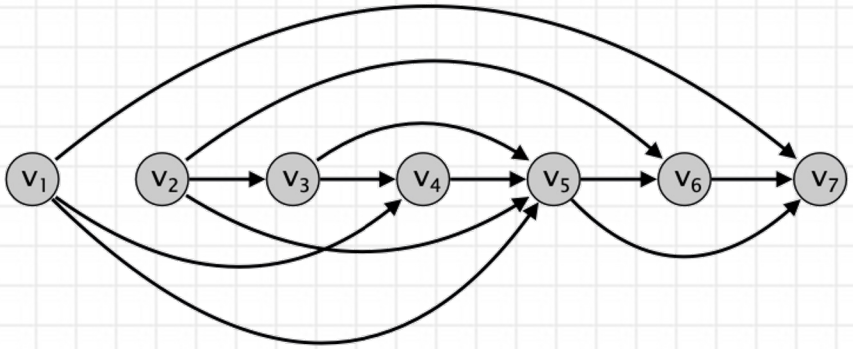
- Def. A directed acyclic graphs (DAG) is a directed graph that contains no directed cycles.

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 (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`

- 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.

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}.

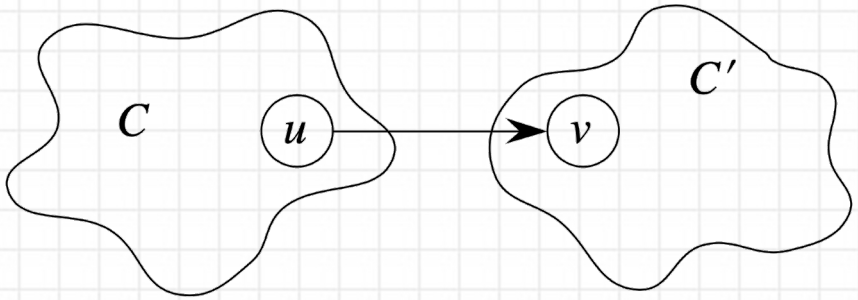
- $\mathrm{G}^{\mathrm{T}} = (V, E^T)$, where $E^T = \{(u, v) | (v, u) \in E\}$
  In other words, same graph except all edges are reversed.
- Adjacency list representation: G^T can be obtained in $\mathrm{O}(|V| + |E|)$

- **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)

```
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
```

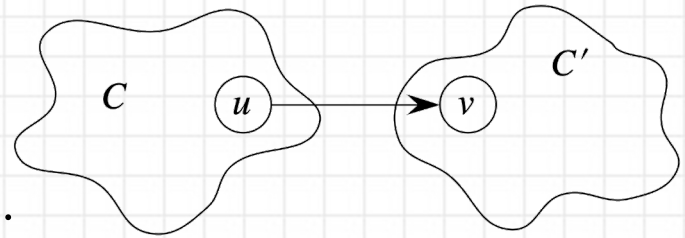
- **Lemma**
  Let C and C' be distinct SCCs in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.



- **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.

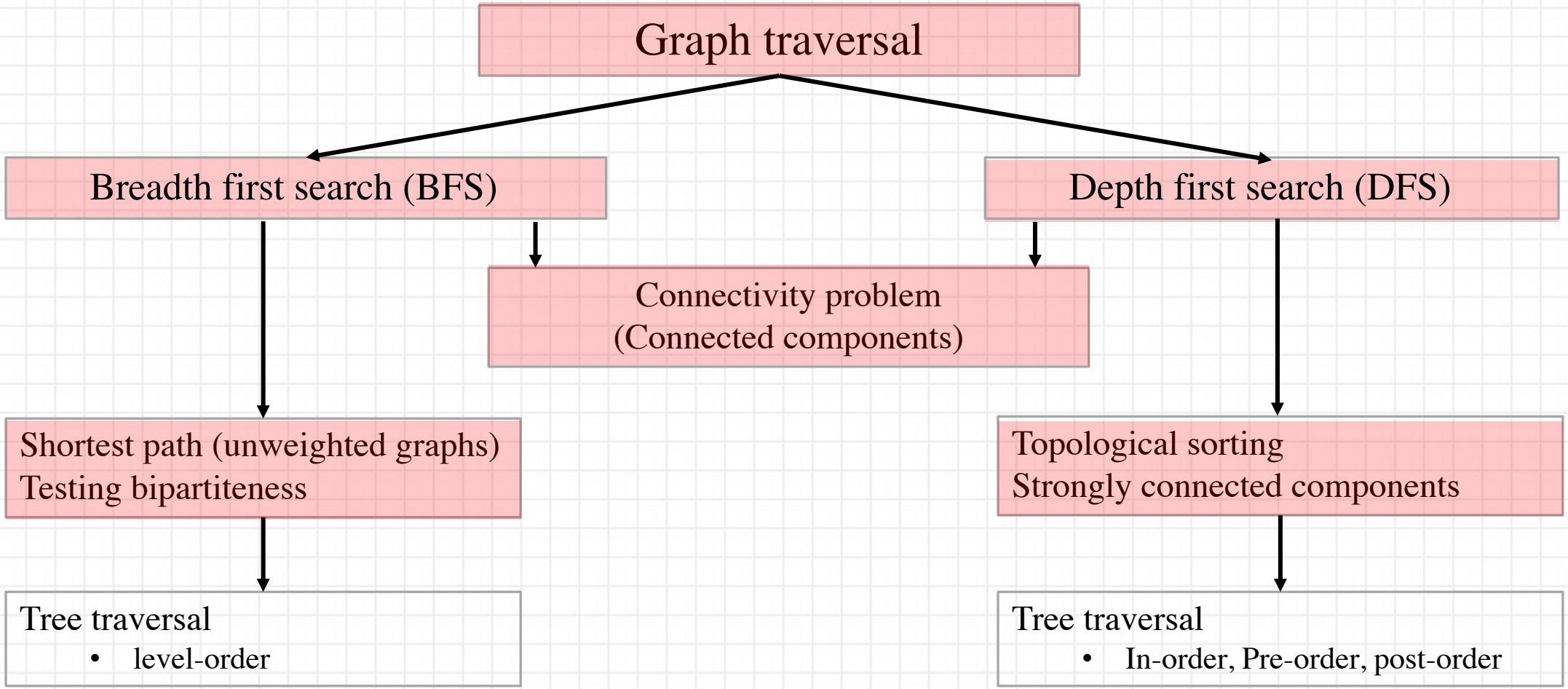
Strongly Connected Component (SCC)

- **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.

Exam 2: 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

Exam 2: 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

Type of Questions in Exam-2:

- Short answers /Definition/ True/False
- Designing (explaining) an algorithm for a graph-related problem
  - Detecting that the problem is a graph-related problem (Explain how we can formulate the given problem as a graph problem, and how the given information can be represented as a graph, i.e., adjacency list, adjacency matrix, etc.
  - Which one of the discussed algorithms (DFS, BFS, testing bipartiteness, SCC, topology ordering can be employed to solve the given problem and justify (explain) the correctness of your approach.
  - Discuss the overall time complexity. (The running time takes to create the corresponding graph and the running time takes to solve the problem)

- Tree traversal (level-order)
- Connected components

- Depth first search (DFS)
  - Topological sorting
  - Tree traversal (in-order, pre-order, post-order)
  - Connected components

- Flow network
  - Max-flow min-cut theorem
  - Ford-Fulkerson algorithm

Exam 2: 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

Minimum Spanning Tree

- Weighted graphs
  - Each edge has an associated weight, cost, or distance.
  - Edge (u, v) → w(u, v)

- Spanning tree
  - Given graph G = (V, E), a tree $T = (V, E_T)$ such that $E_T \subseteq E$ is a spanning tree of G.
  - Tree T spans the graph G

Minimum Spanning Tree (MST)

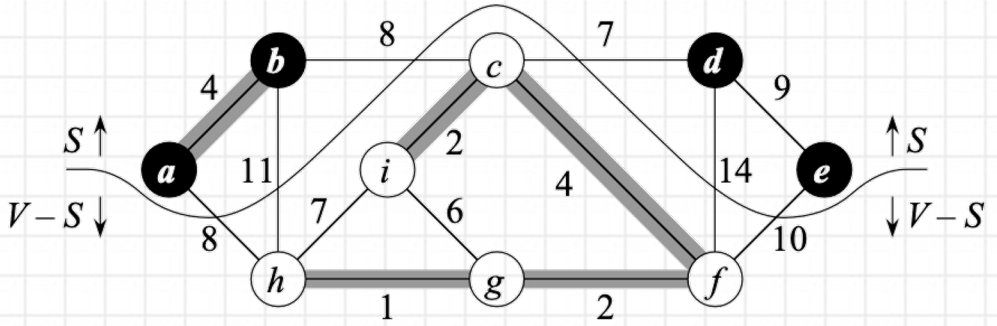
- Spanning tree
  - Given graph G = (V, E), a tree $T = (V, E_T)$ such that $E_T \subseteq E$ is a spanning tree of G.
  - Tree T spans the graph G
- Minimum spanning tree = Minimum-weight spanning tree
- Spanning tree T for G such that the sum $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized
- Approach: "Greedy choice"
- Algorithms:
  - Kruskal
  - Prim

Growing a Minimum Spanning Tree

- This greedy strategy is captured by the following generic method, which grows the minimum spanning tree one edge at a time.
- The generic method manages a set of edges A, maintaining the following loop invariant:
  - **Prior to each iteration, A is a subset of some minimum spanning tree**.
- At each step, we determine an edge (u, v) that we can add to A without violating this invariant $A \cup \{(u, v)\}$ is also a subset of an MST
- An edge is safe edge if adding it to A will not violate the invariant.

Some Definitions

- Cut
  - A cut $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V .



- With this definition, we say
  - An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of this endpoints is in S, and the other in $V - S$
  - A cut respects a set A of edges if no edge in A crosses the cut.
  - An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

Generic-MST

- Theorem:

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing this cut. Then edge (u, v) is safe for A.

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

Generic-MST

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3        find an edge (u, v) that is safe for A
4        A = A ∪ {(u, v)}
5  return A
```

- Notes
  - The set A is always acyclic.
  - At any point $G_A = (V, A)$ is a forest
  - At first when $A = \phi$, we have |V| trees in the forest G_A, each a tree of one vertices
  - At each iteration, the number of trees is reduced by one.
  - While loop (line 2-4) runs for |V|-1 times to find the edges required to form the minimum spanning tree.
  - The method terminates when we have one tree (clearly, with |V|-1 edges).

Generic-MST

- Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G,
- [**Theorem**:] let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing this cut. Then edge (u, v) is safe for A.
- [**Corollary**:] let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, Then edge (u, v) is safe for A.
  - Pf. Cut $(V_C, V - V_C)$ respects A, and (u, v) is a light edge for this cut → safe

MST Algorithms

- Kruskal's algorithm
  - The set A is a forest whose vertices are all those of the given graph.
  - The safe edge added to A is always a least-weight edge in the graph that connects two distinct components. (so it is not creating a loop)

- Prim's algorithm
  - The set A forms a single tree.
  - The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

MST: Summary

- Spanning tree
  - Given graph G = (V, E), a tree $T = (V, E_T)$ such that $E_T \subseteq E$ is a spanning tree of G.
  - Tree T spans the graph G
- Minimum spanning tree
  - Spanning tree T for G such that the sum $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized

Algorithm	Paradigm	Data Structure Used	Running Time
Kruskal	Greedy	Disjoint-Set (Union-Find)	O(|E| log|V|)
Prim	Greedy	Priority Queue (Binary Min-Heap)	O(|E| log|V|)

Type of Questions in Exam-2:

- Short answers /Definition/ True/False
- Running Kruskal's/Prim's algorithms on a given graph (show your steps)
- Proving some properties of minimum spanning trees.
  - Similar to HW4/Q3
  - Proof by contradiction (assume the given statement is not correct, then show this assumption will cause some contradictions → Thus, the given statement is true.)

- Spanning tree T for G such that the sum $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized

Algorithm	Paradigm	Data Structure Used	Running Time
Kruskal	Greedy	Disjoint-Set (Union-Find)	O(|E| log|V|)
Prim	Greedy	Priority Queue (Binary Min-Heap)	O(|E| log|V|)

Exam 2: Practice Problems

Course website

CS-3510 | Algorithms

home policies lectures assignments resources

#	Assignment	Release Date	Deadline
1	**hw1:** [pdf | tex | solution]	05/20	05/27
2	**hw2:** [pdf | tex | solution]	05/27	06/03
3	**hw3:** [pdf | tex | solution]	06/03	06/17
4	**hw4:** [pdf | tex | solution]	06/21	07/02
5	**hw5:** [pdf | tex | solution]	07/08	07/16

You can use this LaTeX template file to prepare your solutions on the cloud-based LaTeX editor OverLeaf.

#	Exam	Date (mm/dd)	Time (EST)	Location
1	**Exam 1:** Complexity, Devide-and-Conquer, Dynamic Programming [practice | pdf | solution]	06/09 Thursday	03:30 pm	Klaus 2443
2	**Exam 2:** Greedy Algorithms, Graph Algorithms [practice | pdf | solution]	07/07 Thursday	03:30 pm	Klaus 2443
3	**Final Exam:** Inclusive (including all discussed topics) [practice | solution]	07/28 Thursday	03:00 pm	Klaus 2443