CS-3510:
Design and Analysis of Algorithms

Final Review

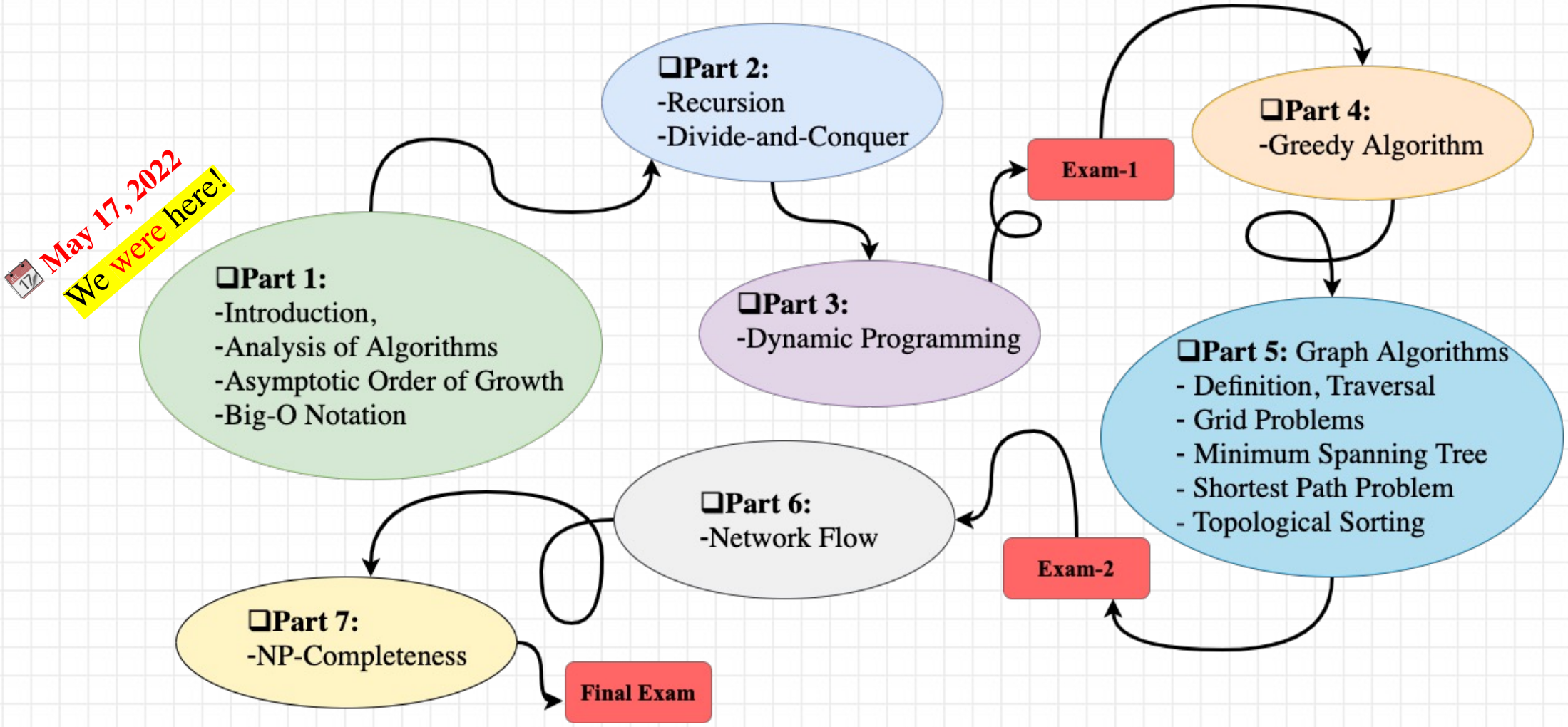
Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

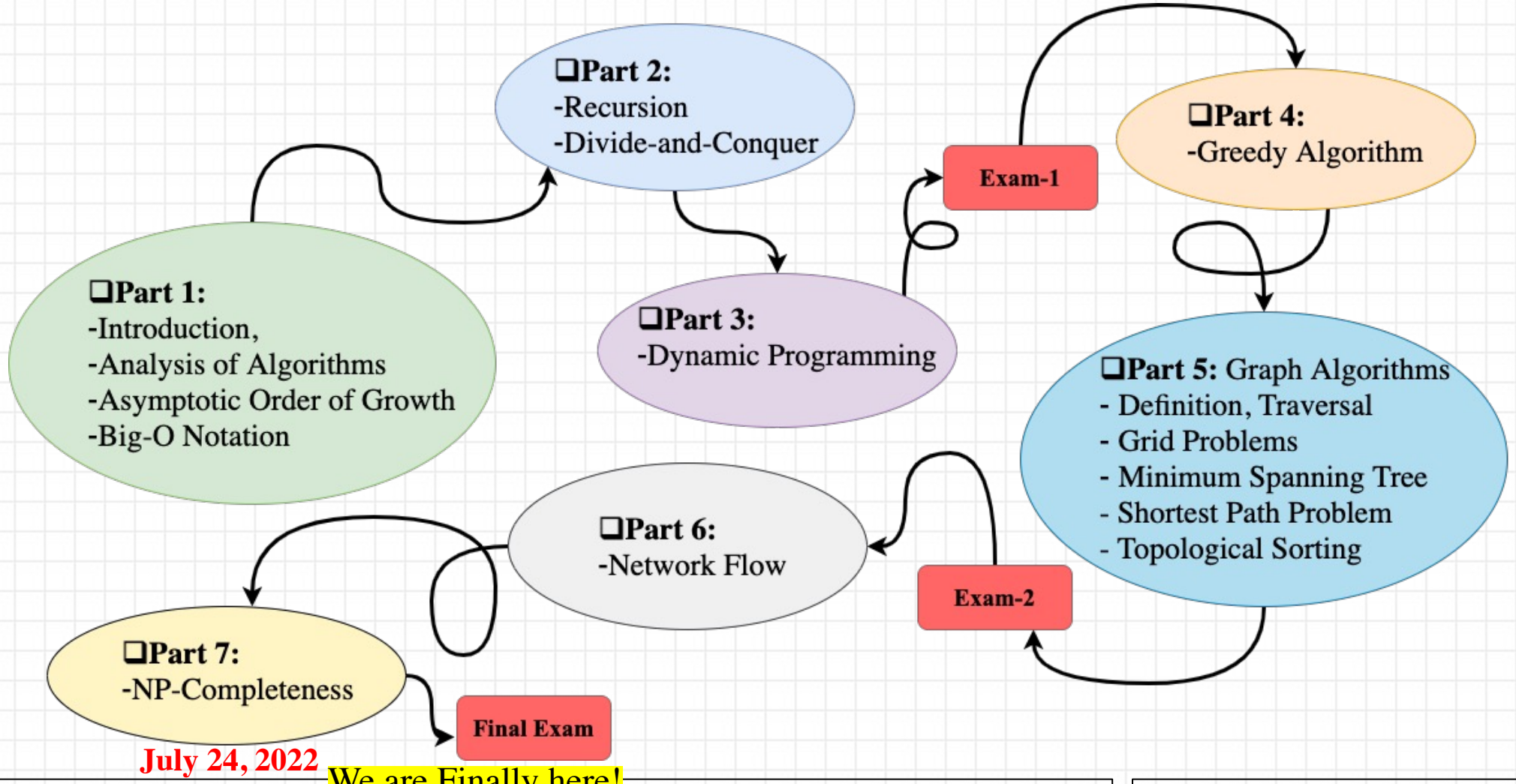
Overview

- General course review
  - What we have accomplished
- Final exam:
  - Problem/topic distribution
  - Preparation
- Final remarks

Roadmap


May 17, 2022
We were here!
❑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
Exam-2
❑Part 6:
-Network Flow
❑Part 7:
-NP-Completeness
Final Exam

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
Exam-2
❑Part 6:
-Network Flow
❑Part 7:
-NP-Completeness
Final Exam
July 24, 2022
We are Finally here!

Summary

Design and Analysis of Algorithms

- What?
  - Algorithms
  - Algorithmic paradigms; design and correctness
  - Performance analysis
- Why?
  - Fundamental to all areas of computer science
    - Operating systems, Networks and distributed systems, Machine learning, Data science, Numerical computation, Cryptography, Computational biology, etc.
  - Inseparable part of every technical interview
    - Internship, Part-time, Full-time
  - Useful and Fun!
    - Problem solving skills
    - Competitive programming, Hackathons, etc.

Final Exam

- Date: **Thursday, July 28, 2022**
- Time: **03:00 pm – 05:00 pm**
- Location: Klaus 2443

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

Final Exam

- Contents:  Inclusive (including all discussed topics)
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming
  - Graph algorithms (Traversal, applications, MST, shortest path, flow network)
  - NP-completeness

Final Exam

- Contents:  Inclusive (including all discussed topics)
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming
  - Graph algorithms (Traversal, applications, MST, shortest path, flow network)
  - NP-completeness

Final Review

Time Complexity

- Asymptotic Order of Growth
  - It is easier to talk about the lower bound and upper bound of the running time.
  - To practically deal with time complexity analysis, we use asymptotic notations.
  - The asymptotic growth of a function (in this case T(n)) is specified using Θ, O, and Ω notations.
  - Asymptotic means for "very large" input size, as n grows without bound or "asymptotically".

Final Review

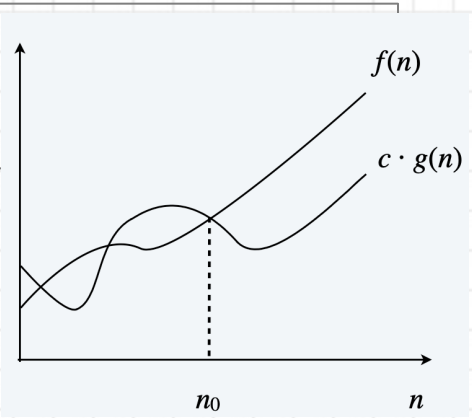
Time Complexity

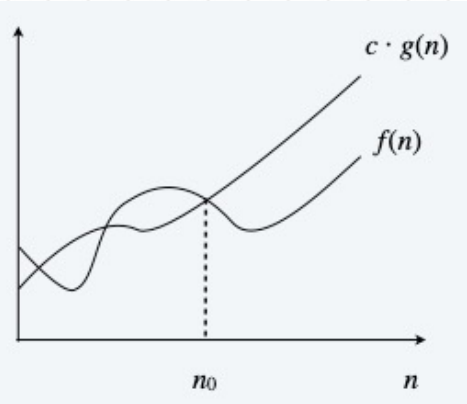
- Asymptotic Order of Growth
  - In general, the asymptotic notations define bounds on the growth of a function. Informally, a function $f(n)$ is:
    - $\Omega(g(n))$ if $g(n)$ is an asymptotic lower bound for $f(n)$
    - $O(g(n))$ if $g(n)$ is an asymptotic upper bound for $f(n)$
    - $\Theta(g(n))$ if $g(n)$ is an asymptotic tight bound for $f(n)$

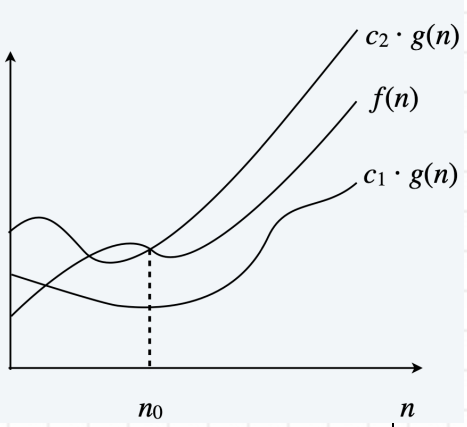
Final Review

Time Complexity

- Asymptotic Order of Growth (Formal definition):
  - **Big Omega (lower bound):**
    $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq cg(n) \geq 0$ for all $n \geq n_0$.
  - **Big O (upper bound):**
    $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$
  - **Big Theta (tight bound):**
    $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
  - Note: $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$







Final Review

Time Complexity

- Big O Notation Properties

Reflexivity	f is $O(f)$
Constants	If f is $O(g)$ and $c > 0$, then cf is $O(g)$
Products	If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$
Sums (Additivity)	If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max\{g_1, g_2\})$ Ex. If $f_1 \in O(n^2)$ and $f_2 \in O(n^4)$. Then, $f_1 + f_2 \in O(n^4)$
Transitivity	If f is $O(g)$ and g is $O(h)$, then f is $O(h)$

- So, we can ignore the lower terms and constants:
  - Ex. $f = 2n^3 + 4n^2 - 5n + 1 \in O(n^3)$
  - Ex. $f = 4n^5 \in O(n^5)$

Final Review

Time Complexity

- Asymptotic Bounds for Some Common Functions

Polynomials	$f(n) = a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ and thus, $O(n^d)$ if $a_d > 0$.
Logarithms	$\log_a n$ is $\Theta(\log_b n)$ for every $a>1$ and $b>1$. Note: $O(\log_a n) = O(\log_b n)$ (Recall $\log_b n = \log_b a \times \log_a n$)
Logarithms vs polynomials	$\log_a n$ is $O(n^d)$ for every $a>1$ and $d>0$. Logarithms grow slower than every polynomial regardless of how small d is.
Exponential vs Polynomials	n^d is $O(r^n)$ for every $d>0$ and $r>1$. Exponentials grow faster than every polynomial regardless of how big d is.

Final Review

Asymptotic Order of Growth Hierarchy

n^n

$n!$

3^n

2^n

$n^3, 2n^3, (n-1000)^3, n^3 - n^2$

$n^2, 2n^2, -1000n^2 + n, 100n^2 + \log n$

$n \log n$

$n, 2n, 1000n, 10^{10}n + 1000$

$\sqrt{n}$

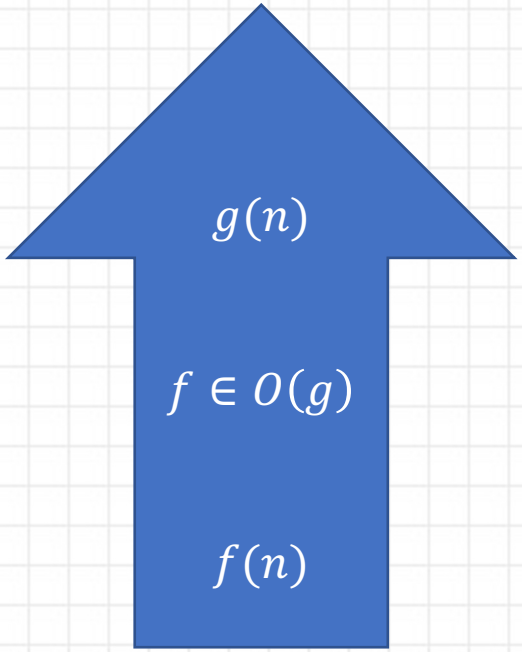
$\log^3 n$

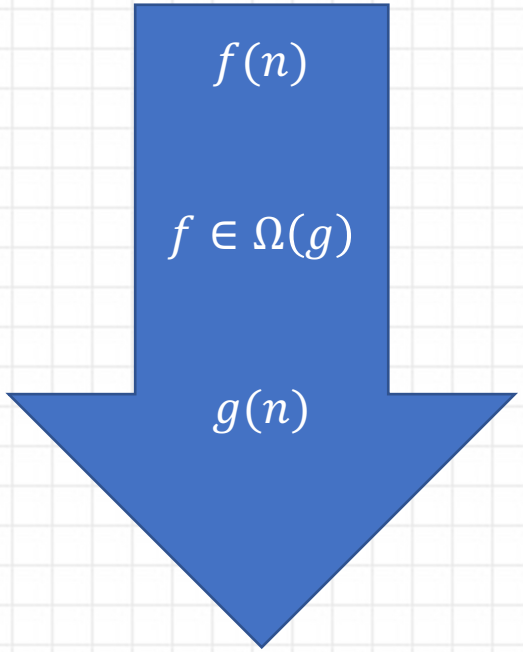
$\log^2 n$

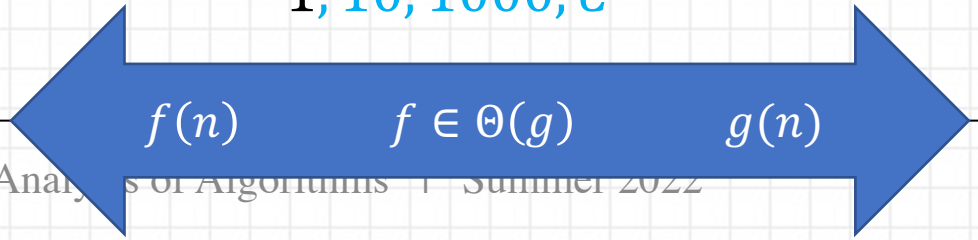
$\log n, \log_2 n, \log_3 n, \log_{10000} n$

$\log \log n$

$1, 10, 1000, C$



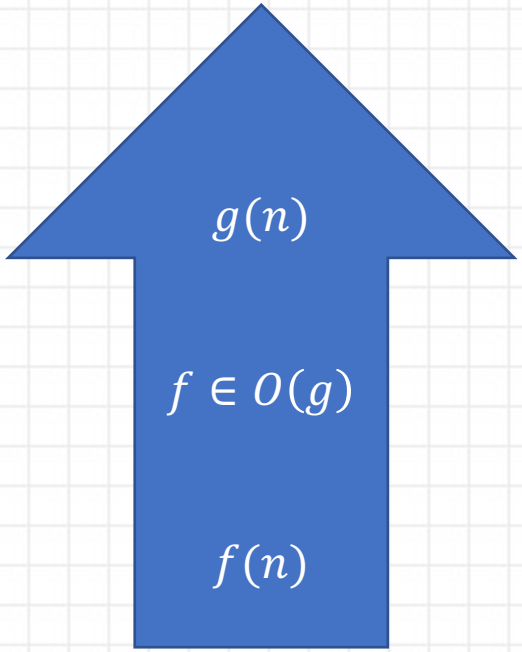




Final Review

Asymptotic Order of Growt

Type of questions for this part in the Final Exam:
- Short answers
- Definition
- True/False questions



n^n

$n!$

3^n

2^n

$n^3, 2n^3, (n-1000)^3, n^3 - n^2$

$n^2, 2n^2, -1000n^2 + n, 100n^2 + \log n$

$n \log n$

$n, 2n, 1000n, 10^{10}n + 1000$

$\sqrt{n}$

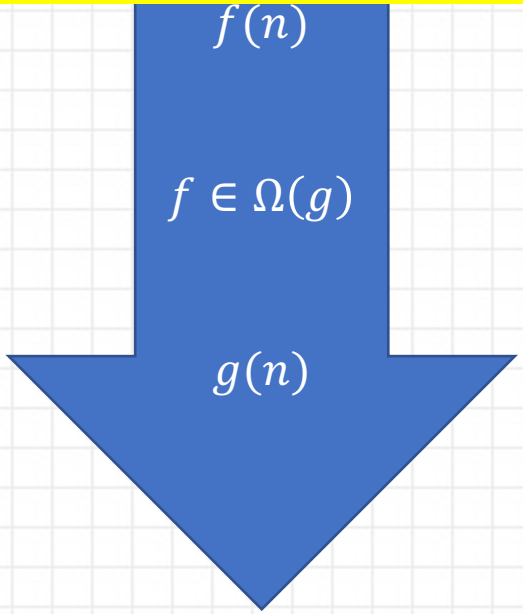
$\log^3 n$

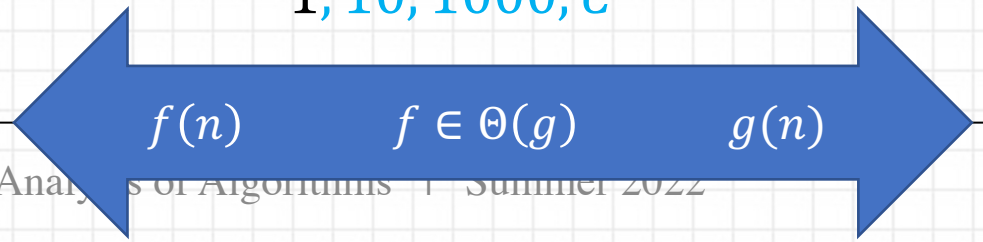
$\log^2 n$

$\log n, \log_2 n, \log_3 n, \log_{10000} n$

$\log \log n$

$1, 10, 1000, C$





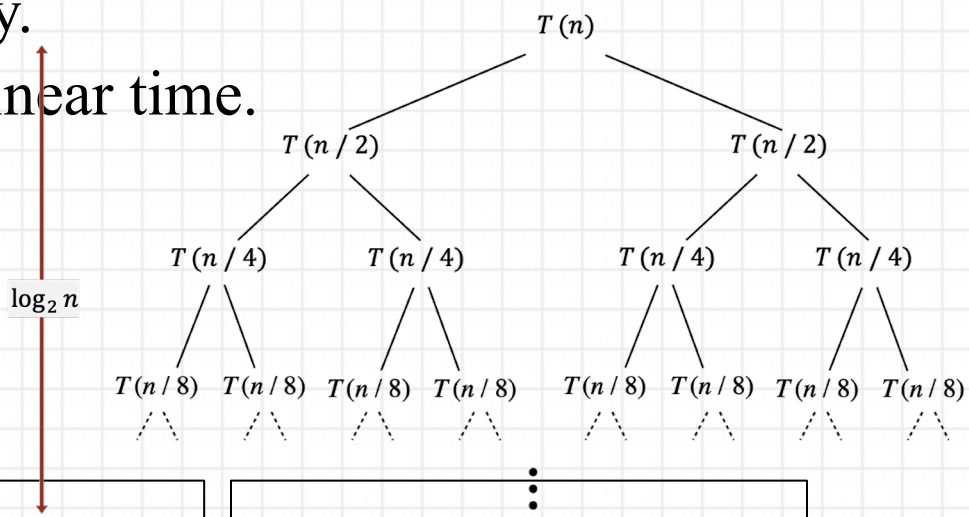
Final Exam

- Contents:  Inclusive (including all discussed topics)
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming
  - Graph algorithms (Traversal, applications, MST, shortest path, flow network)
  - NP-completeness

Final Review

Divide-and-Conquer (D&C)

- Main steps
  - Divide up problems into several subproblems (of the same type).
  - Solve (conquer) each subproblem (usually recursively).
  - Combine the solutions.
- Most common framework
  - Divide the problem of size n into two subproblems of size $n/2$ in linear time
  - Solve (conquer) the two subproblems recursively.
  - Combine two solutions into overall solution in linear time.



Final Review

Divide-and-Conquer (D&C)

- Discussed examples:
  - **Binary-search**
    → Variant/applications of binary search
  - **Merge-sort**
    → Variant/applications of merge-sort
  - **Quick-sort**
    → Variant/applications of quick-sort
  - **Matrix multiplication**
  - **Closest pair of points**

Search Algorithm

Sorting Algorithm

Sorting Algorithm

Type of questions:
- Variant (Design) /applications /parts of **binary search**, merge-sort, or quick-sort
- True/False questions
- Worst case/best case
- Time and space complexity

Final Review

Master Theorem

- Goal. Recipe for solving common divide-and-conquer recurrences,

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right), & \text{if } a > b^d \quad \text{(case 1)} \\ \Theta\left(n^d \log n\right), & \text{if } a = b^d \quad \text{(case 2)} \\ \Theta\left(n^d\right), & \text{if } a < b^d \quad \text{(case 3)} \end{cases}$$

- Limitation. Master theorem cannot be used if
  - $T(n)$ is not monotone, e.g., $T(n) = \sin(n)$
  - $f(n)$ is not polynomial, e.g., $T(n) = 2\,T\left(\frac{n}{2}\right) + 2^n$
  - b cannot be expressed as a constant, e.g., $T(n) = a\,T(\sqrt{n}) + f(n)$

Application of Master Theorem

- The recurrence relation is given → direct
- Dominated by root/leaves/evenly distributed
- An algorithm (D&C) is given, you need to find the recurrence first. Then, apply the Master Theorem → indirect
- (Very similar to Exam-1)

Final Exam

- Contents:  Inclusive (including all discussed topics)
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming
  - Graph algorithms (Traversal, applications, MST, shortest path, flow network)
  - NP-completeness

Final Review

Dynamic Programming (DP)

- Dynamic Programming vs. Divide-and-Conquer

Divide-and-Conquer:

- Divide problem into subproblems
- Recursively solve the subproblems and aggregate solutions

Dynamic Programming

- Divide problem into subproblems, recursively solve them
- Subproblems overlap
- When a subproblem has been solved, remember its solution and reuse that solution rather than resolving it later (memoization)

Final Review

Dynamic Programming

- Top-down vs. Bottom-up Approach
  - "Top-down" dynamic programming
    - Begin with problem description
    - i.e., begin at root of tree and work downwards
    - Recursively subdivide problem into subproblems
  - "Bottom-up" dynamic programming
    - Start at the leaf nodes of tree, i.e., the base case(s).
    - Build up solution to larger problem from solutions of the simpler subproblems

Final Review

DP Examples

- One-dimensional
  1. Fibonacci sequence
  2. Staircase climbing
  3. Rod-cutting
  4. Red-black game

- Two-dimensional
  5. Longest common subsequence (LCS)
  6. Coin-changing
  7. Knapsack

Type of questions in Final Exam:
- Design a DP algorithm (1D or 2D)
- Discuss the optimal substructure
- Write the recurrence relation/base case
- Top-down / bottom-up
- Time and space complexity
- Very similar to the example solved in class and the assignments.

Final Exam

- Contents:  Inclusive (including all discussed topics)
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming
  - Graph algorithms (Traversal, applications, MST, shortest path, flow network)
  - NP-completeness

What about the Greedy algorithms?

Final Review

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

Final Review

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.

Final Review

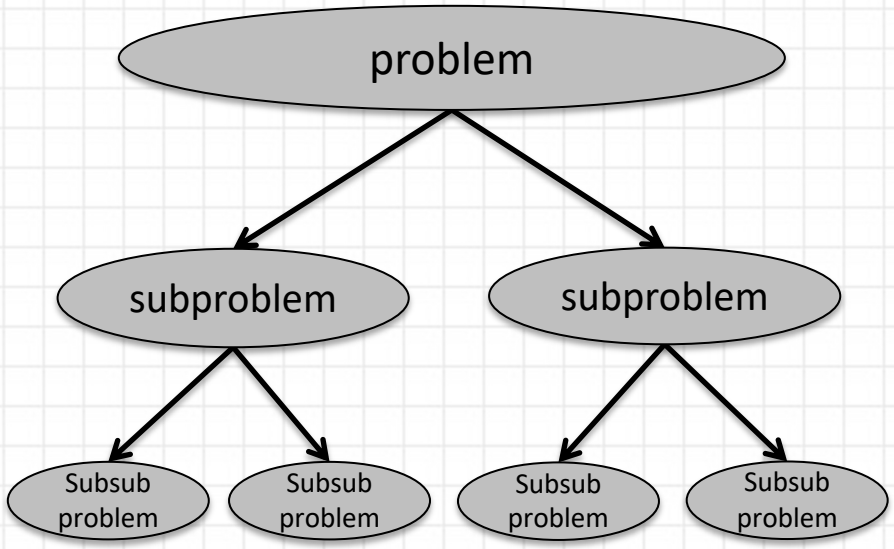
Difference with Dynamic Programming

- Dynamic programming:
  - Make a choice at each step, but the choice usually depends on the solutions to subproblems.
  - Consequently, we typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems.
  - Even in top-down approach, we use memoizing. So, even though the code works top down, we still solve the subproblems before making a choice.
  - Solves the subproblems before making the first choice.

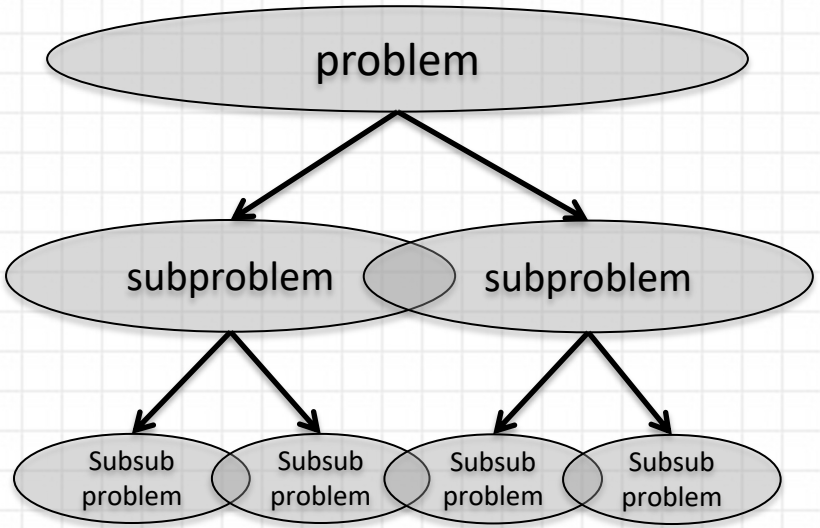
Final Review

Greedy Algorithms

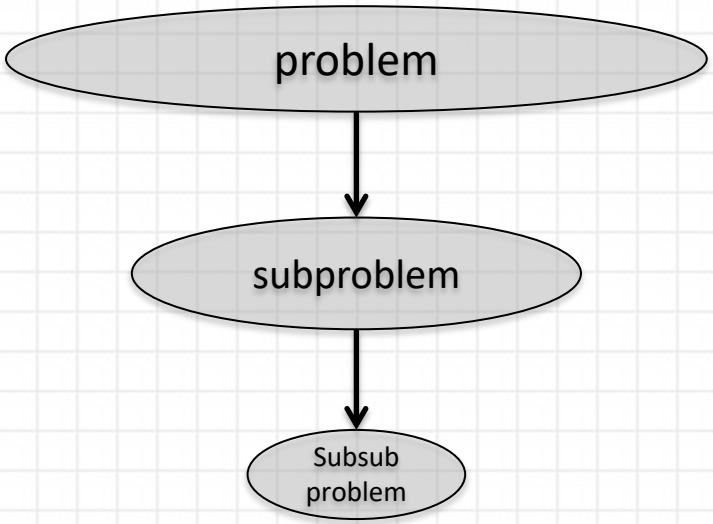
Divide-and-Conquer



Dynamic Programming



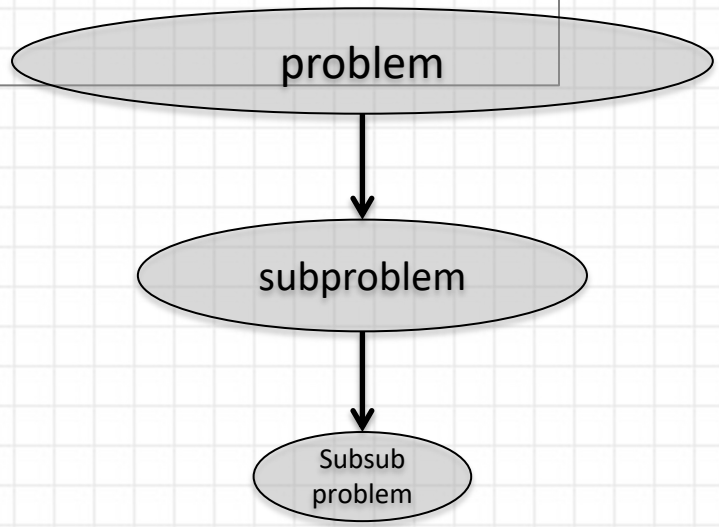
Greedy Approach



Optimal substructure
But only one subproblem

Final Review

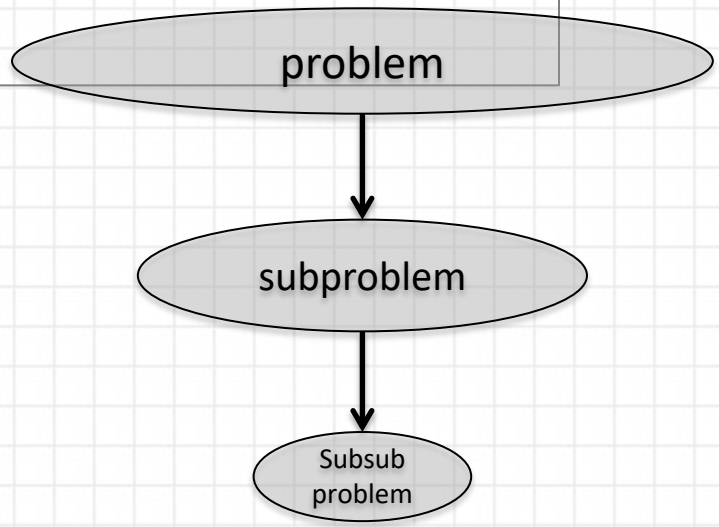
Greedy Algorithms



- 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.

Final Review

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 a substructure.

Type of questions in Final Exam:
- "Perhaps" T/F, short answer, or definition

Final Exam

- Contents:  Inclusive (including all discussed topics)
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming
  - Graph algorithms (Traversal, applications, MST, shortest path, flow network)
  - NP-completeness

Final Review

Graph Algorithms

- 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

Final Review

Graph Definitions and Terminology: Summary

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

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

Final Review

Exam 2: Graph

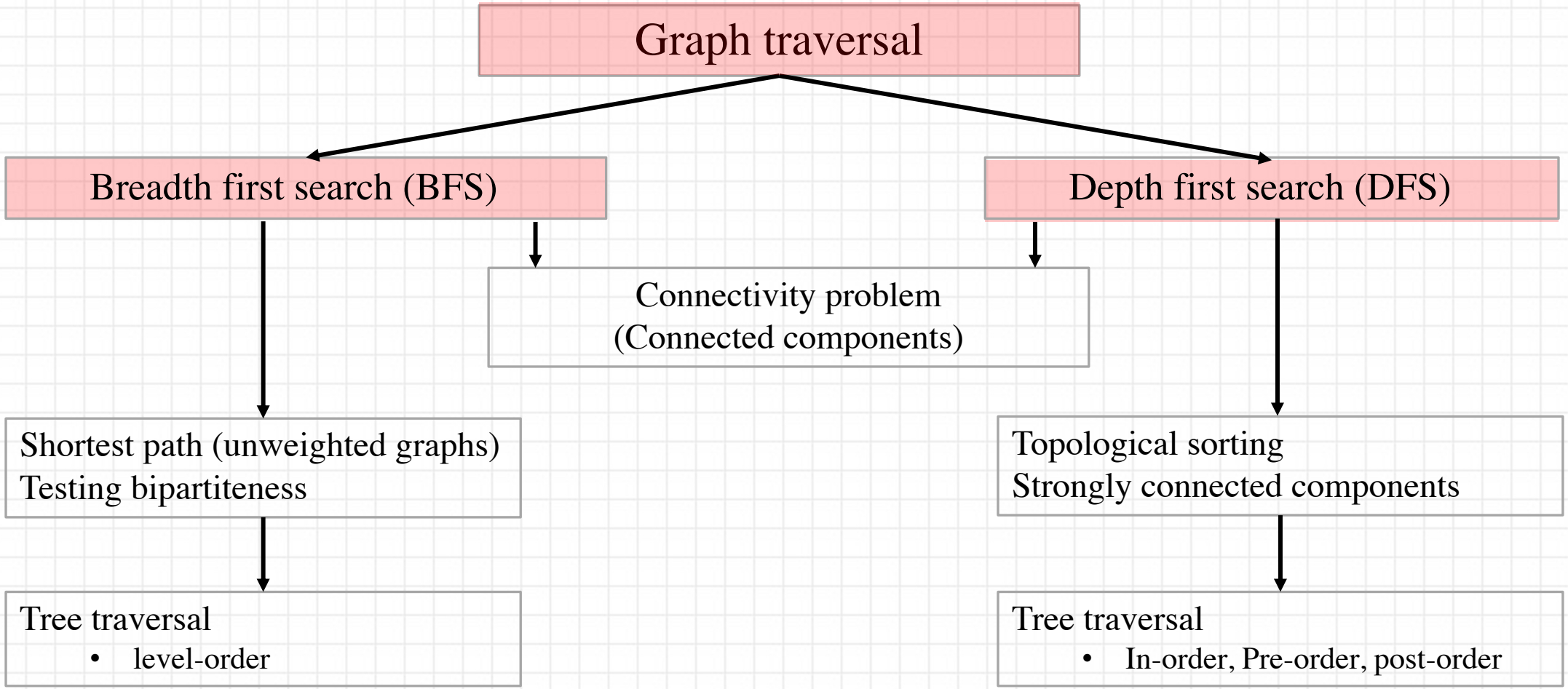
- Graph definition and representation
  - Adjacency matrix
  - Adjacency list

- 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

Final Review

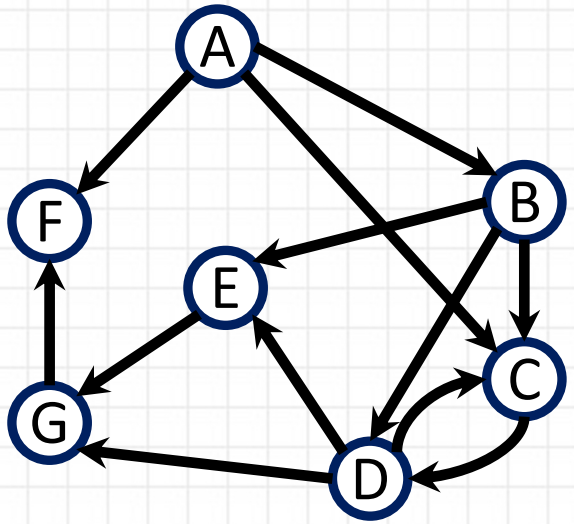
Graph



Final Review

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

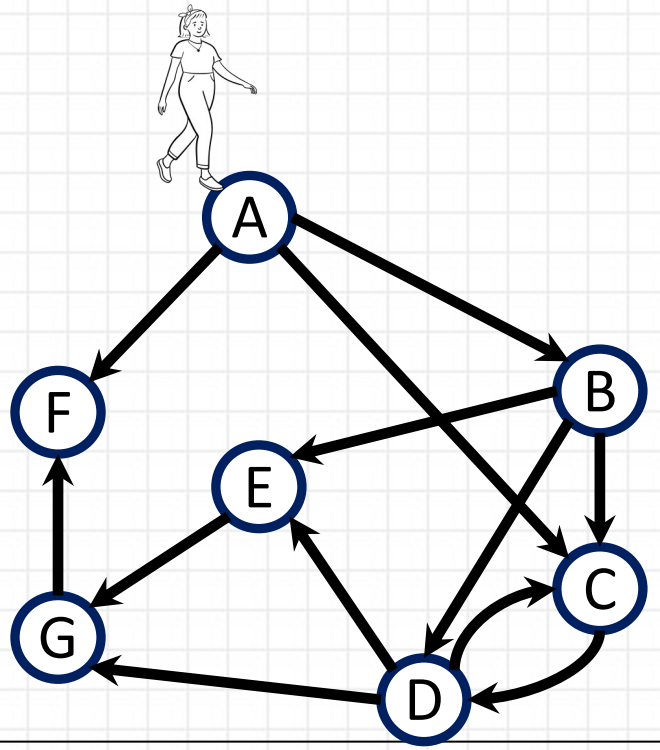
gray := visited node

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

Final Review

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
```

Final Review

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|)

Final Review

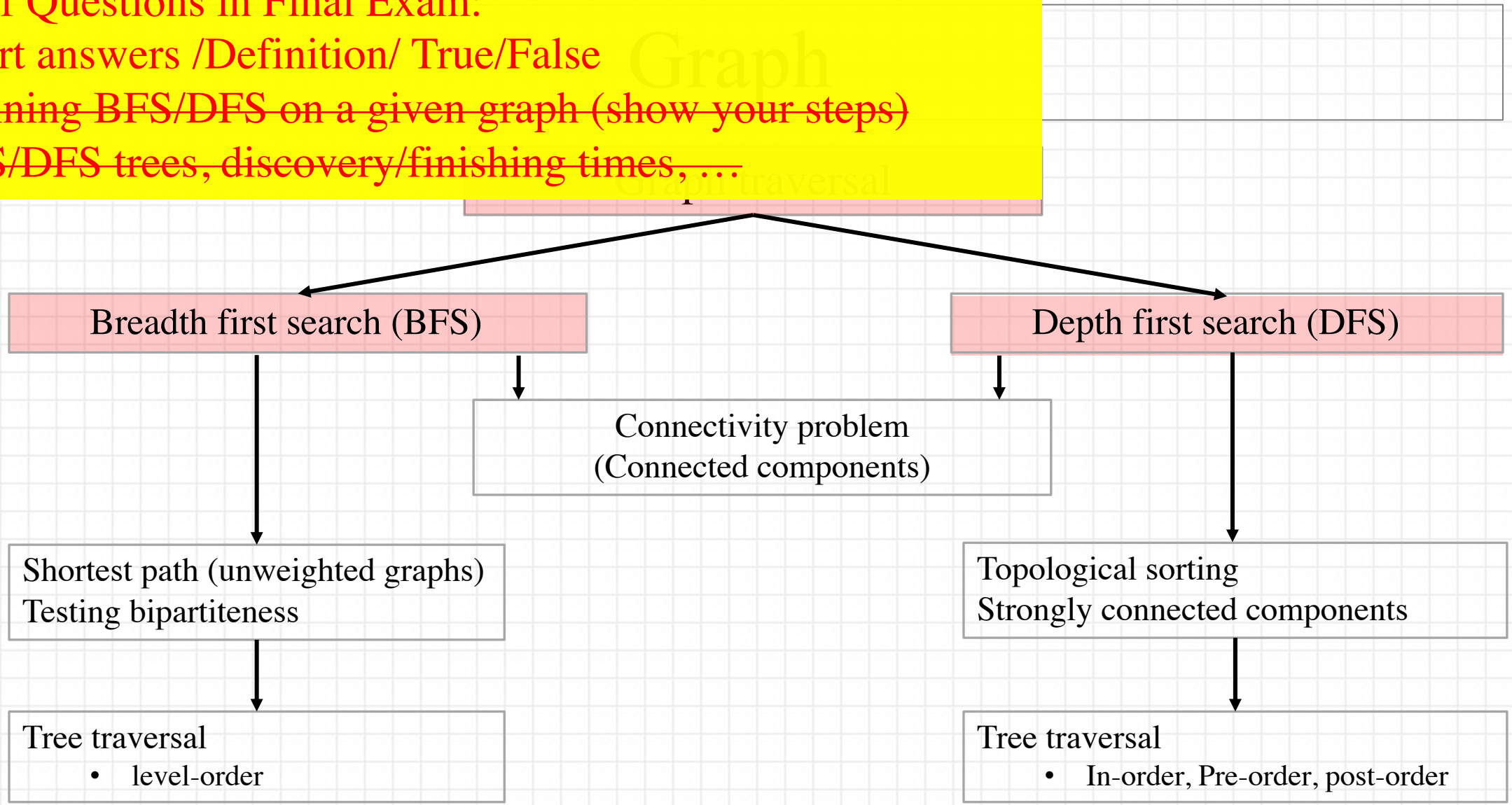
BFS and DFS

- 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.**

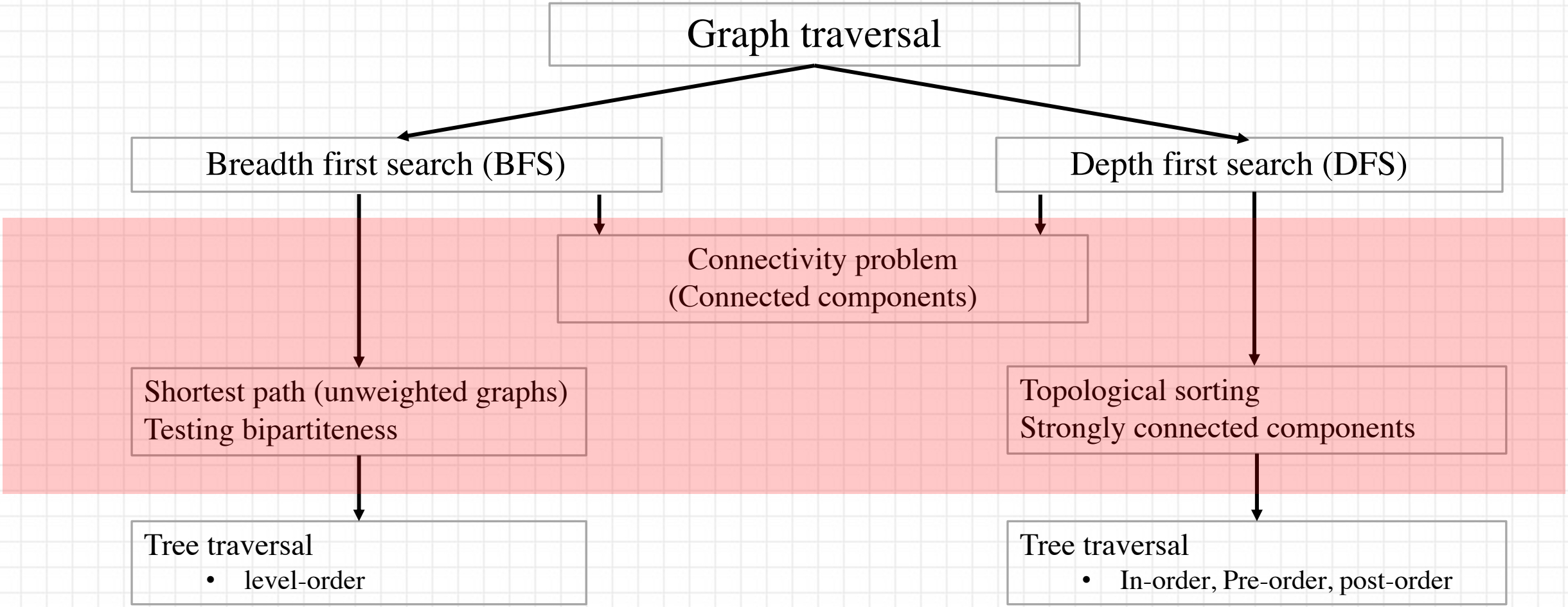
Type of Questions in Final Exam:

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



Final Review

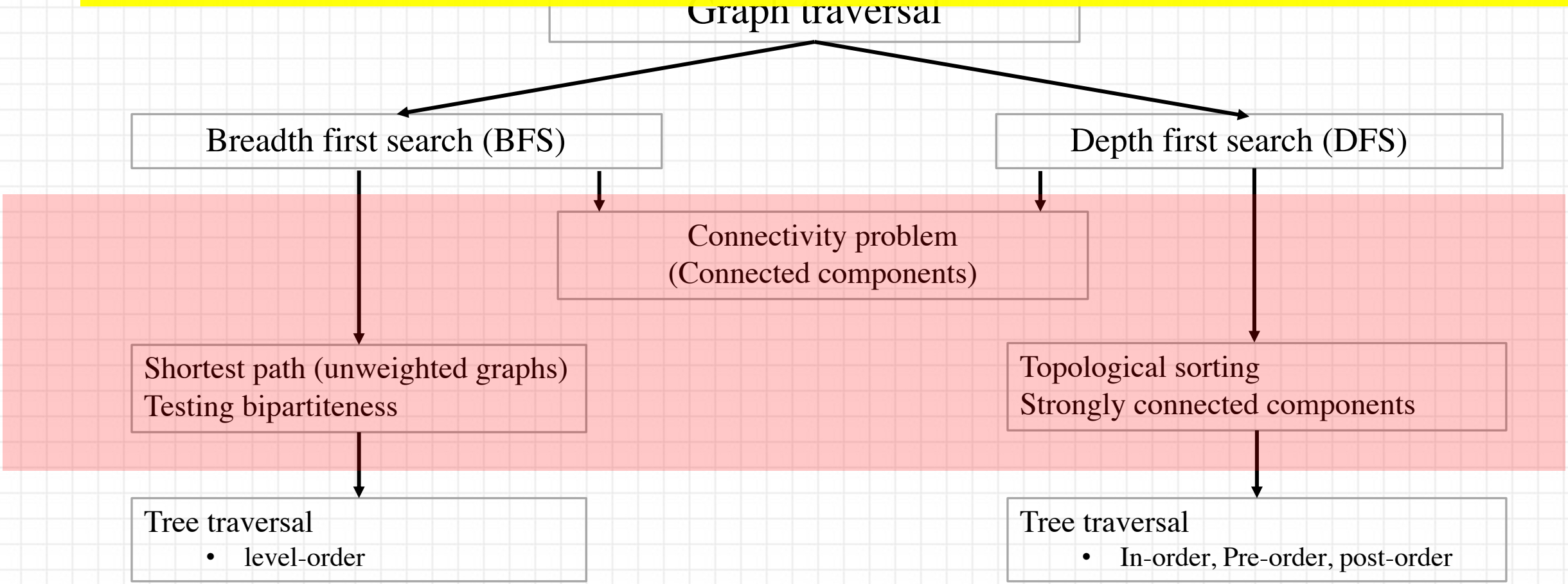
Exam 2: Graph traversal applications



Final Review

Type of Questions in Final Exam for graph-related problems:

- **Short answers /Definition/ True/False**
- Designing (explaining) an algorithm for a graph-related problem



Final Review

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

Final Review

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

Final Review

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).

Final Review

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.

Final Review

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 Final Exam:

- 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.
  - Proof by contradiction (assume the given statement is not correct, then show this assumption will cause some contradictions → Thus, the given statement is true.)

- 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)	$O(\|E\| \log\|V\|)$

Final Exam

- Contents:  Inclusive (including all discussed topics)
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming
  - Graph algorithms (Traversal, applications, MST, shortest path, flow network)
  - NP-completeness

Final Review

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

Final Review

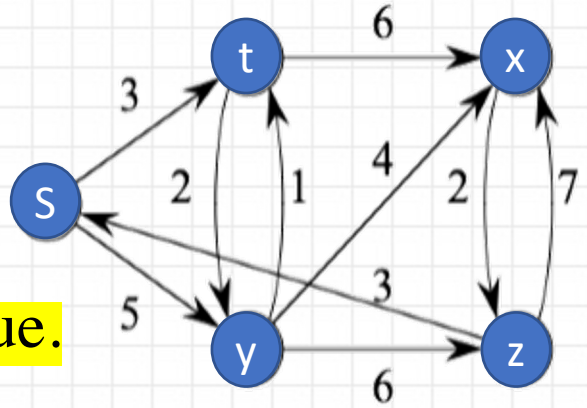
Shortest Paths (Weighted Graphs)

- Problem description
  - Given graph $G = (V, E)$, and a weight function $w: E \rightarrow \mathbb{R}$
  - Weight of a path $p=[v_0, v_1, \ldots, v_k]$ $= \sum$ edge_weights on path p
    $= \sum_{i=1}^{k} w(v_{i-1}, v_i)$
  - Shortest-path weight from $u \rightsquigarrow v$ $= \delta(u, v) = \begin{cases} \min_{p(u \rightsquigarrow v)} w(p), & \text{if there exists a path } u \rightsquigarrow v \\ \infty, & \text{otherwise} \end{cases}$

Final Review

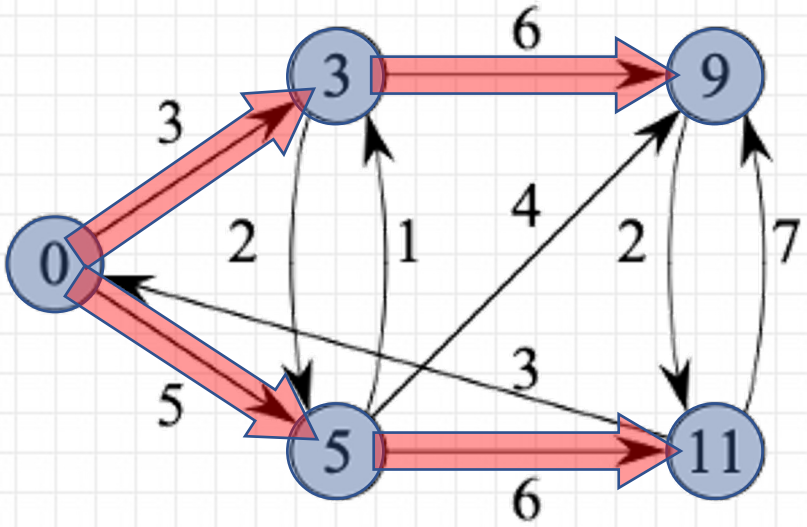
Shortest Paths (Weighted Graphs)

- Example
  - Weighted, directed graph
  - Shortest path from source s to other vertices

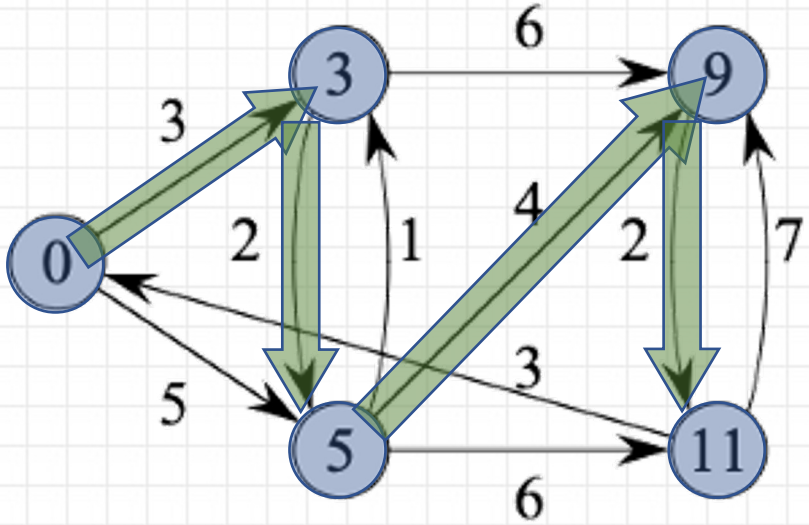


Shortest path does not have to be unique.

Shortest path 1



Shortest path 2



Final Review

Shortest Paths (Weighted Graphs)

- Variants of shortest path problem:

SSSP

1. Single-source: Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$.

   Bellman-Ford, Dijkstra

2. Single-destination: Find shortest paths to a given destination vertex.

3. Single-pair: Find shortest path from u to v. In the worst case is the same as solving single-source.

APSP

4. All-pairs: Find shortest path from u to v for all $u, v \in V$.

   Floyd-Warshall

Final Review

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

1. Optimal substructure
   - Any sub-path of a shortest path is a shortest path.
2. Cycles
3. Outputs the SSSP problem
   - Shortest path distance d[v]
   - Shortest path predecessors/ Shortest path tree
4. Initialization
5. Relaxation

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
  then d[v] ← d[u] + w(u, v)
       π[v] ← u
```

Final Review

Shortest Paths (Weighted Graphs)

- Shortest path properties
  1. Triangle inequality
  2. Upper-bound property
  3. No-path property
  4. Convergence property
  5. Path relaxation property

Final Review

SSSP: Bellman-Ford

- Bellman-Ford
  - Dynamic programming approach
  - Allows negative-weight edges.
  - Computes d[v] and π[v] for all v $\in V$.
  - Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.
  - Running time: $\Theta(|V||E|)$

  - Proof of correctness using path-relaxation property (CLRS 24.1)

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Final Review

SSSP: Bellman-Ford

- Bellman-Ford
  - So, in short,
  - The algorithm iterates at most |V|-1 times.
  - At each iteration, it updates (relaxes) along all edges.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3        for each edge (u, v) ∈ G.E
4              RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6        if v.d > u.d + w(u, v)
7              return FALSE
8  return TRUE
```

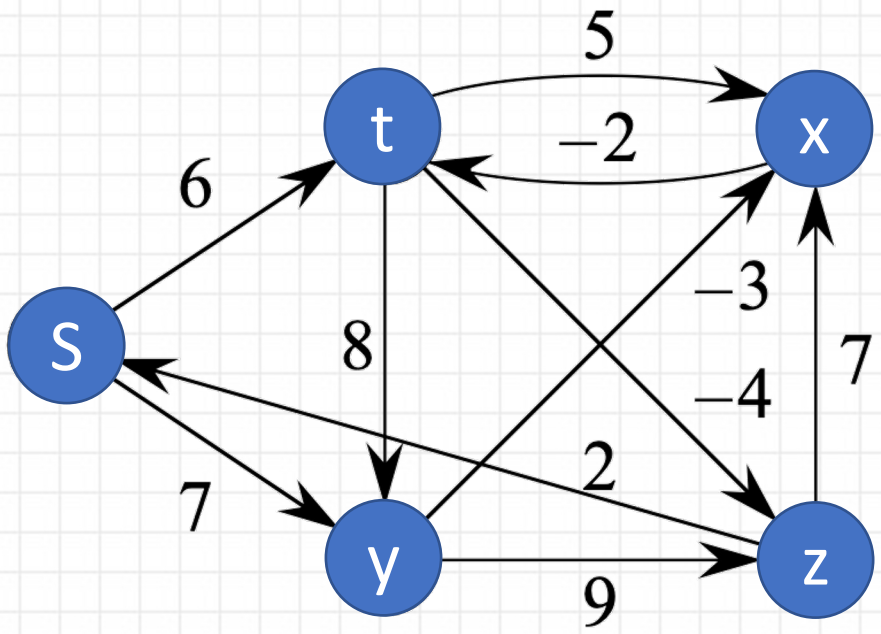
Final Review

SSSP: Bellman-Ford

- Example
  - 5 vertices → 4 iterations

Parents (path)

v	π(v)
s	
t	
y	
x	
z	




```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

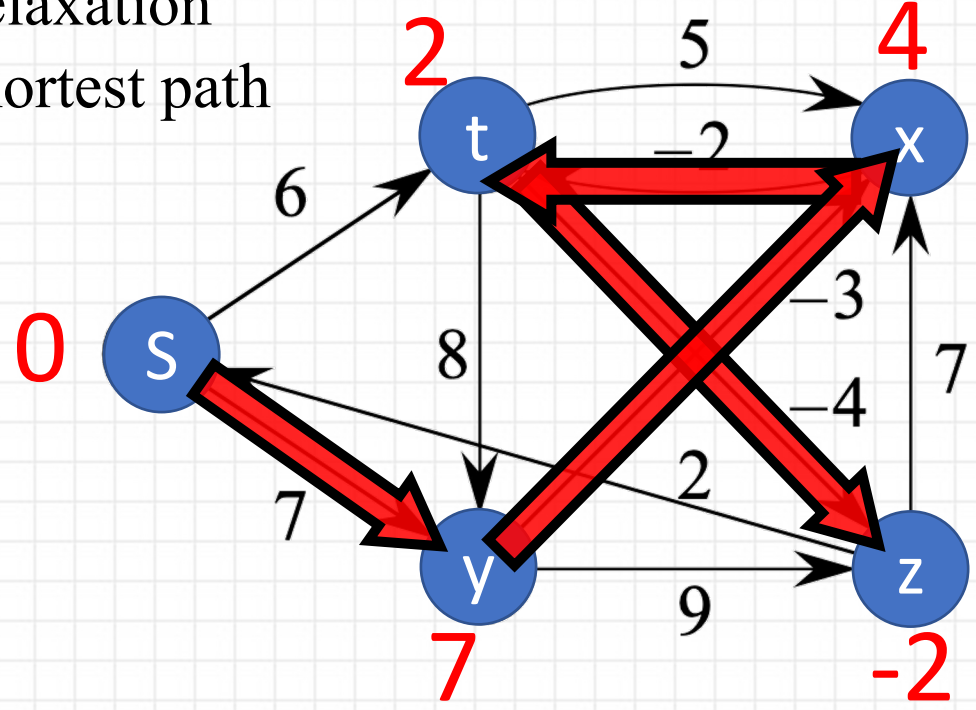
i	s	t	y	x	z
0					
1					
2					
3					
4					

Final Review

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation
  - Shortest path

No negative-weight cycle
Return TRUE



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7	4	-2
4	0	2	7	4	-2

Final Review

SSSP: Bellman-Ford Summary

- Bellman-Ford
  - Dynamic programming approach
  - How to apply? (Main steps)

1. Create two tables (both can be implemented using 1D arrays)
   - One for "d", and
   - Another for "π(v)"
2. Initialize the tables

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Shortest distance

	s	t	y	x	z
i	0	∞	∞	∞	∞

Final Review

SSSP: Bellman-Ford Summary

- Bellman-Ford
  - Dynamic programming approach
  - How to apply? (Main steps)
  1. Create two tables (both can be implemented using 1D arrays)
     - One for "d", and
     - Another for "π(v)"
  2. Initialize the tables
  3. Iterate over each node and each time iterate over all edges and update the tables if necessary (relaxation)

$\Theta(|V||E|)$

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

Shortest distance

	s	t	y	x	z
i	0	∞	∞	∞	∞

Final Review

SSSP: Dijkstra's Algorithm

- Dijkstra
  - Greedy approach
  - No negative-weight *edges*.
  - Essentially a weighted version of BFS
  - Instead of a FIFO queue, uses a priority queue. Keys are shortest-path weights (d[v]).
  - Have two sets of vertices:
    - S = vertices whose final shortest-path weights are determined,
    - Q = priority queue = $V - S$.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Min priority queue
e.g., binary heap

Final Review

SSSP: Dijkstra's Algorithm

- Dijkstra's algorithm running time:
  - Like Prim's algorithm, depends on implementation of priority queue.
  - If binary heap, each operation takes $O(\log |V|)$ time $\Rightarrow$ $O(|E| \log |V|)$.
  - If a Fibonacci heap:
    - Each Extract-Min takes $O(1)$ amortized time.
    - There are $O(|V|)$ other operations, taking $O(\log |V|)$ amortized time each.
    - Therefore, time is $O(|V| \log |V| + |E|)$.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Running time?

Final Review

SSSP: Dijkstra's Algorithm

- Dijkstra
  - So, in short,
  - The algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
  - The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.
  - Greedy strategy:
    Always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
   then d[v] ← d[u] + w(u, v)
        π[v] ← u
```

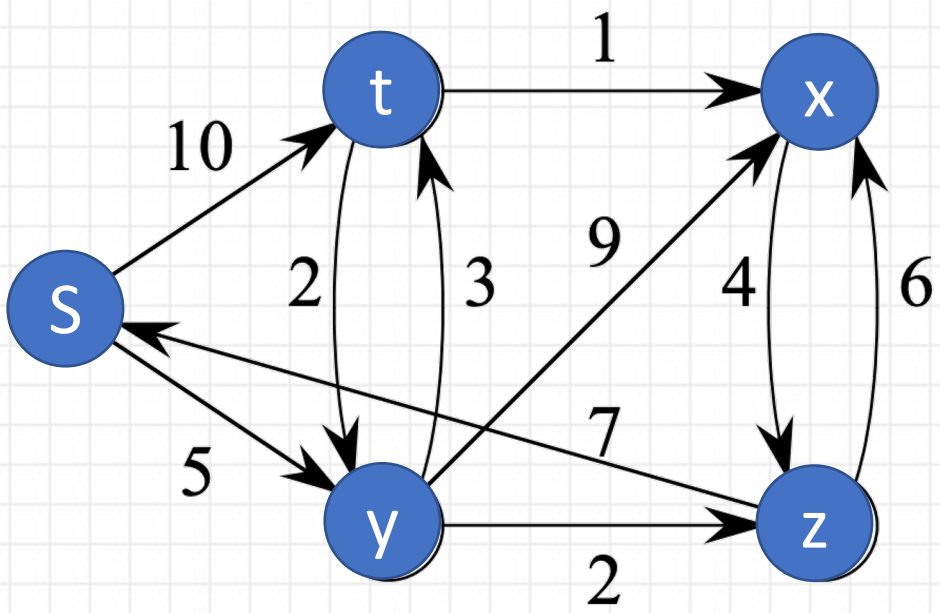
Final Review

SSSP: Dijkstra's Algorithm

- Example

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```



Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

Shortest distance

s	t	y	x	z
0	∞	∞	∞	∞

Final Review

SSSP: Dijkstra's Algorithm

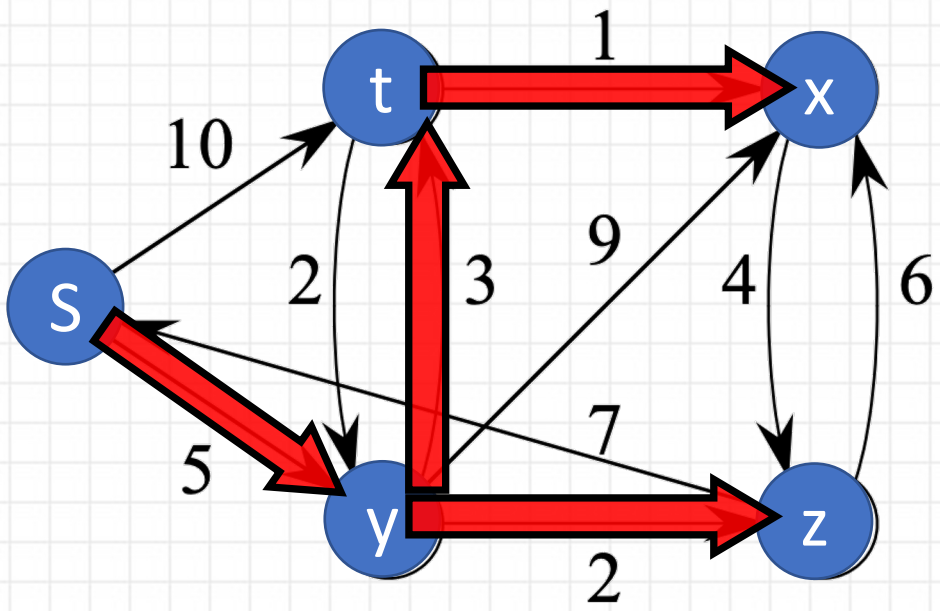
- Example
  - S = {s, y, z, t, x}
  - Q = {~~s: 0~~, ~~t: 8~~, ~~y: 5~~, ~~x: 9~~, ~~z: 7~~}

unvisited nodes (priority queue)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	t
z	y



Shortest distance

s	t	y	x	z
0	8	5	9	7

Final Review

SSSP: Dijkstra's Algorithm Summary

- Dijkstra
  - Greedy approach
  - No negative-weight edges.
  - Essentially a weighted version of BFS
  - Instead of a FIFO queue, uses a priority queue. Keys are shortest-path weights (d[v]).
  - If binary heap, each operation takes $O(\log |V|)$ time $\Rightarrow O(|E| \log |V|)$.

  - How to apply? (Main steps)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Final Review

SSSP: Dijkstra's Algorithm Summary

- Dijkstra
  - Greedy approach
  - No negative-weight edges.
  - How to apply? (Main steps)

1. Create three data structure
   - One for the priority queue Q (usually min binary heap)
   - One for "d" shortest path weight estimation
   - Another for "π(v)"
2. Initialize them

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Q = {s: 0, t: ∞, y: ∞, x: ∞, z: ∞}

Shortest distance

s	t	y	x	z
0	∞	∞	∞	∞

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

Final Review

SSSP: Dijkstra's Algorithm Summary

- Dijkstra
  - Greedy approach
  - No negative-weight edges.
  - How to apply? (Main steps)

1. Create three data structure
   - One for the priority queue Q
   - One for "d" shortest path weight estimation
   - Another for "π(v)"
2. Initialize them
3. At each step deque the min-distance not chosen node u from the priority queue, and update the neighbors (relax) and the key of the priority Q.

$O(|E| \log |V|)$.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Q = {s: 0, t: ∞, y: ∞, x: ∞, z: ∞}

Shortest distance

s	t	y	x	z
0	∞	∞	∞	∞

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

Final Review

All-Pairs Shortest Path (APSP)

- Problem description
  - Given graph $G = (V, E)$, and a weight function w: $E \rightarrow \mathbb{R}$
  - Output: An $n \times n$ matrix of shortest path distances $\delta(u, v)$.

- Can we use Bellman-Ford or Dijkstra's algorithms?
  - Running Bellman-Ford once from each vertex:
    - $O(|V|^2|E|)$ which is $O(|V|^4)$ if the graph is *dense* ($|E| = \Theta(|V|^2)$).
  - If non-negative weights, then we can run Dijkstra's algorithm once from each vertex:
    - $O(|V||E| \log |V|)$ with binary heap – $O(|V|^3)$ if dense,
    - $O(|V|^2 \log|V| + |V||E|)$ with Fibonacci heap – $O(|V|^3)$ if dense.

Final Review

APSP: Floyd-Warshall

- Floyd-Warshall algorithm
  - Dynamic programming approach
  - We will use a weight matrix W which is defined as:

$$W_{ij} = \begin{cases} 0 & i = j \\ w(i,j) & i \neq j \text{ and } (i,j) \in E \\ \infty & i \neq j \text{ and } (i,j) \notin E \end{cases}$$

  - Recurrence relation:

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & k = 0 \\ \min \begin{cases} d_{ij}^{(k-1)} \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} & k \geq 1 \end{cases}$$

We want $D^{(n)} = d_{ij}^{(n)}$

Final Review

APSP: Floyd-Warshall

- Floyd-Warshall algorithm

We want $D^{(n)} = d_{ij}^{(n)}$

  - Recurrence relation:

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & k = 0 \\ \min \begin{cases} d_{ij}^{(k-1)} \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} & k \geq 1 \end{cases}$$

  - Implementation:
    - Bottom-up (iterative)

```
FLOYD-WARSHALL(W, n)
D^(0) ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

  - Running time?

Final Review

APSP: Floyd-Warshall

- Floyd-Warshall algorithm
  - Implementation:
    - Bottom-up (iterative)
  - Running time?
    - $O(|V|^3)$
  - Memory required?
    - $O(|V|^3)$
    - But we only use the computations from the previous step (k-1). So, we can only store the last step computations → $O(|V|^2)$

```
FLOYD-WARSHALL(W, n)
D^(0) ← W
for k ← 1 to n
    do for i ← 1 to n
           do for j ← 1 to n
                  do d_ij^(k) ← min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1))
return D^(n)
```

Type of Questions in Final Exam for graph-related problems:

- Short answers /Definition/ True/False
- Run a particular shortest path algorithm (Bellman-Ford, Dijkstra, Floyd-Warshall) on a given graph (step-by-step solution)
- Designing (explaining) an algorithm for a graph-related problem
  - [Detecting that the problem is a graph-related problem]
  - Which one of the discussed algorithms (in this case, shortest path algorithms) can be used to solve the current problem and how? (Describe your algorithm and justify the correctness…)
  - Discuss the overall time complexity. (The running time takes to create the corresponding graph and the running time takes to solve the problem)

  - IMPORTANT REMINDER: You can use the algorithms that we discussed in class (e.g., Bellman-Ford, Dijkstra's, …) without explaining how these algorithms work or proving their correctness. (See HW5/Q1 solution)

Final Exam

- Contents:  Inclusive (including all discussed topics)
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming
  - Graph algorithms (Traversal, applications, MST, shortest path, flow network)
  - NP-completeness

Final Review

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

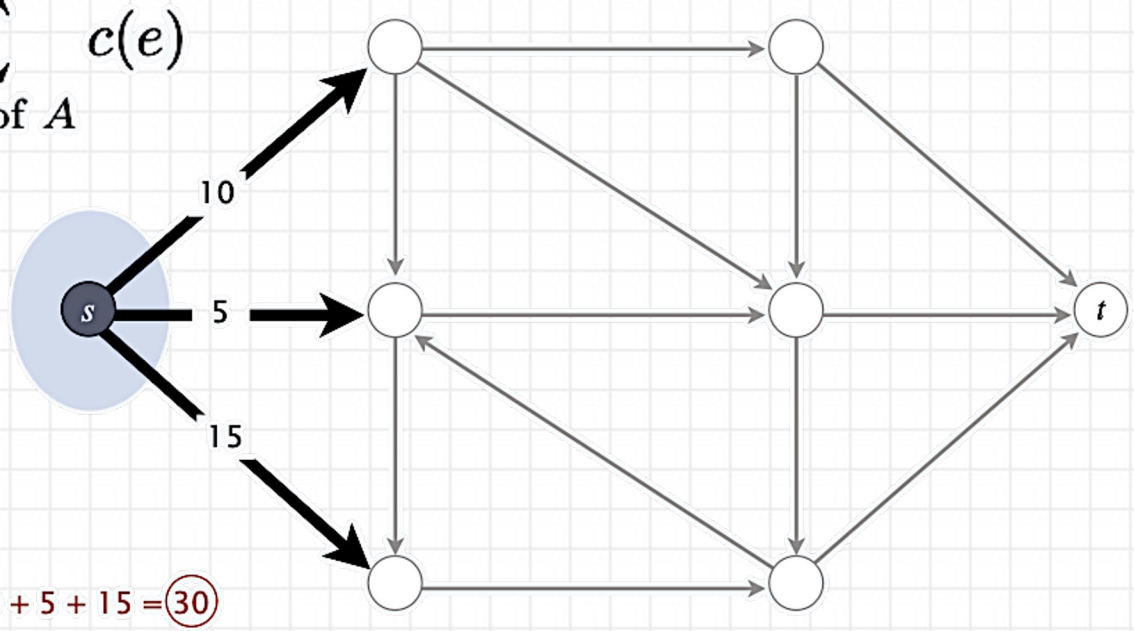
Final Review

Flow Network: Min-Cut Problem

Def. An *st*-cut (cut) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$



Cut notations: $(A, B) \equiv (A, V\text{-}A) \equiv (A, V \backslash A)$

Final Review

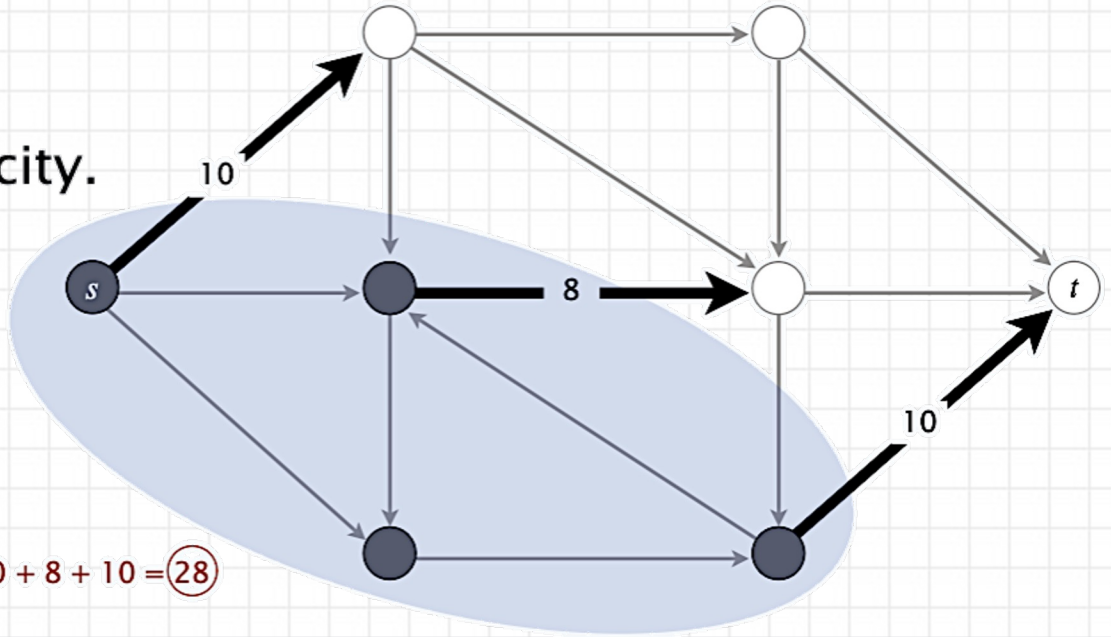
Flow Network: Min-Cut Problem

Def. An st-cut (cut) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$

Min-cut problem. Find a cut of minimum capacity.



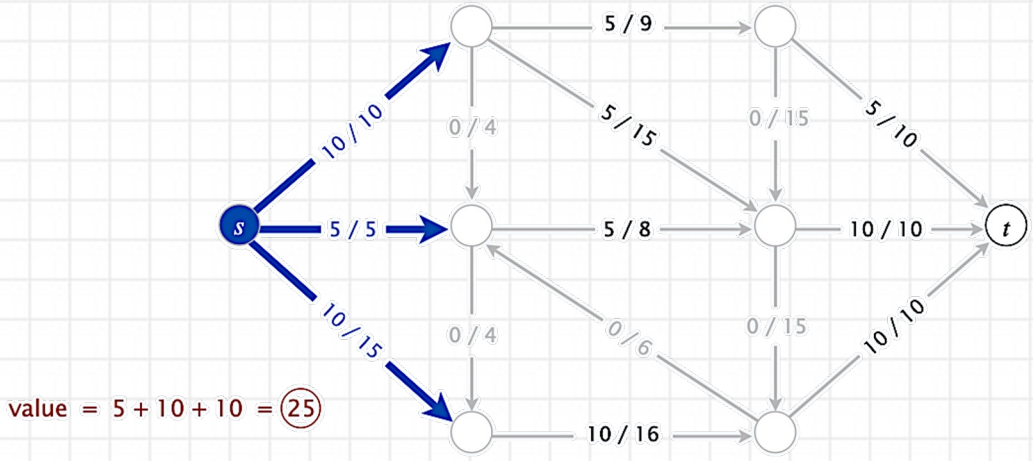
Final Review

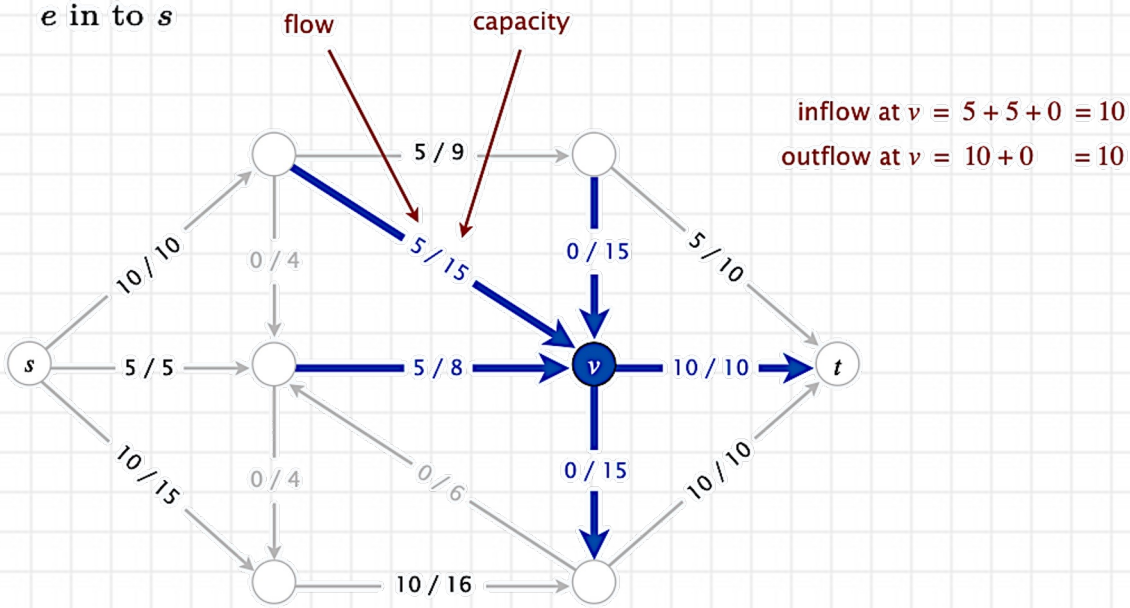
Flow Network: Max-Flow Problem

Def. An *st*-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)$

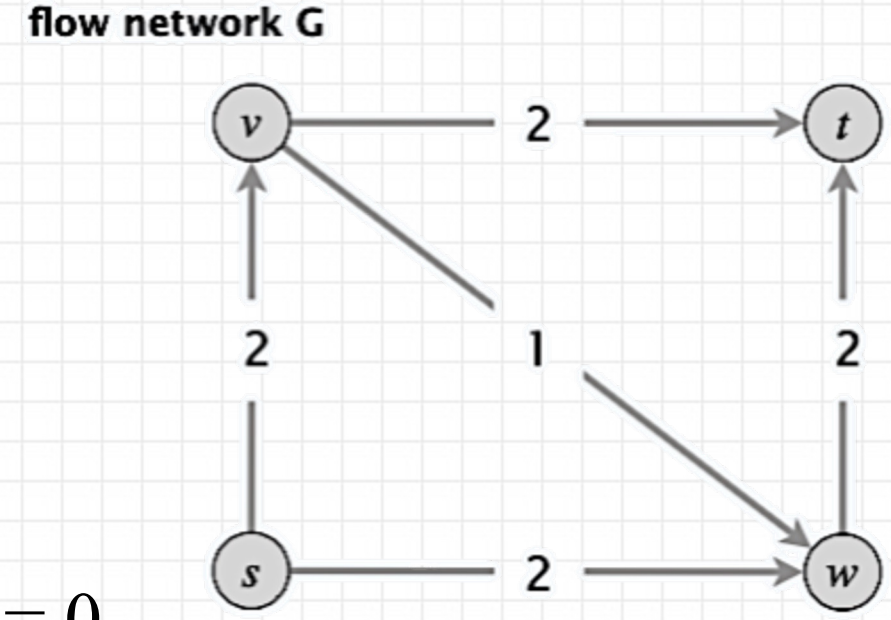




Final Review

Ford–Fulkerson Algorithm

- Q. Why does the greedy algorithm fail?
- A. Once greedy algorithm increases flow on an edge, it never decreases it.
- Bottom line.
  Need some mechanism to "undo" a bad decision.



- Ex.
  Consider flow network G .
  The unique max flow f^* has $f^*(v, w) = 0$.
  Greedy algorithm could choose $s \rightarrow v \rightarrow w \rightarrow t$ as first path.

Final Review

Residual Network

Original edge. $e = (u, v) \in E$.

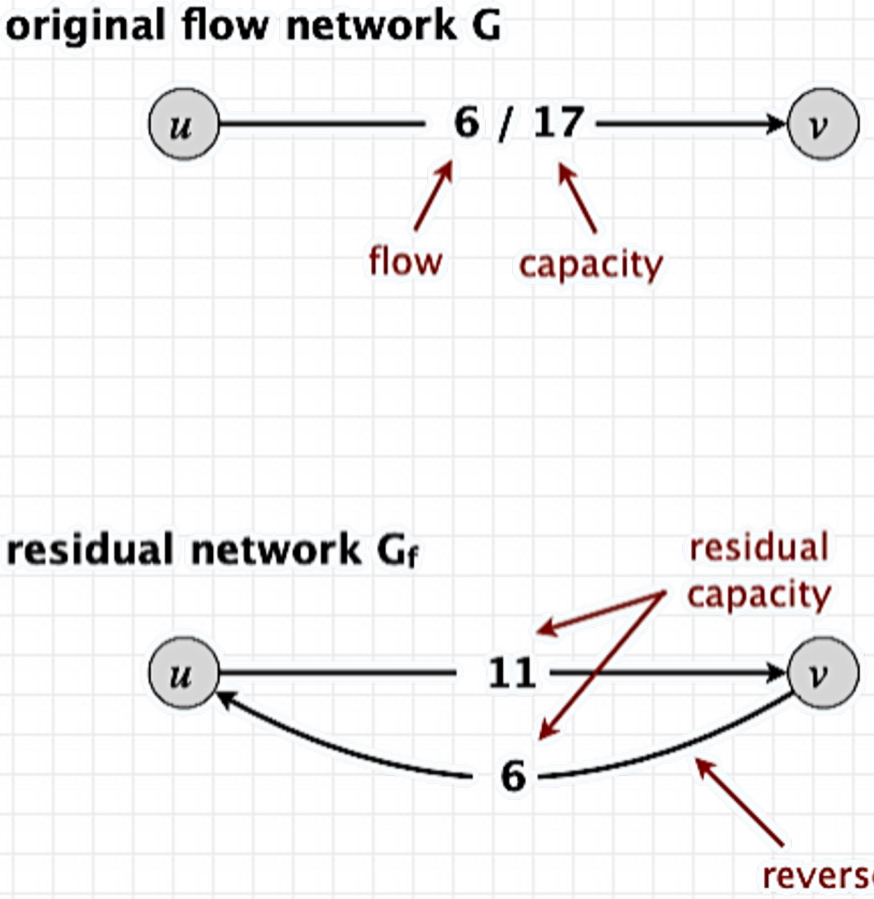
- Flow $f(e)$.
- Capacity $c(e)$.

Reverse edge. $e^{\text{reverse}} = (v, u)$.

- "Undo" flow sent.

Residual capacity.

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e^{\text{reverse}}) & \text{if } e^{\text{reverse}} \in E \end{cases}$$



Final Review

Augmenting Path

- Def. An augmenting path is a simple $s \rightsquigarrow t$ path in the residual network G_f
- Def. The bottleneck capacity of an augmenting path P is the minimum residual capacity of any edge in P.

- Key property. Let f be a flow and let P be an augmenting path in G_f . Then, after calling $f' \leftarrow$ AUGMENT(f, c, P), the resulting f' is a flow and $val(f') = val(f) + bottleneck(G_f, P)$.

AUGMENT(f, c, P)

$\delta \leftarrow$ bottleneck capacity of augmenting path P.

FOREACH edge $e \in P$:

  IF ($e \in E$) $f(e) \leftarrow f(e) + \delta$.

  ELSE $f(e^{reverse}) \leftarrow f(e^{reverse}) - \delta$.

RETURN f.

Final Review

Ford–Fulkerson Algorithm

- Ford–Fulkerson augmenting path algorithm
  - Start with $f(e) = 0$ for each edge $e \in E$.
  - Find an $s \rightsquigarrow t$ path P in the residual network G_f.
  - Augment flow along path P.
  - Repeat until you get stuck.

FORD–FULKERSON(G)

FOREACH edge $e \in E$: $f(e) \leftarrow 0$.

$G_f \leftarrow$ residual network of G with respect to flow f.

WHILE (there exists an $s \rightsquigarrow t$ path P in G_f)

  $f \leftarrow$ AUGMENT(f, c, P).

  Update G_f.

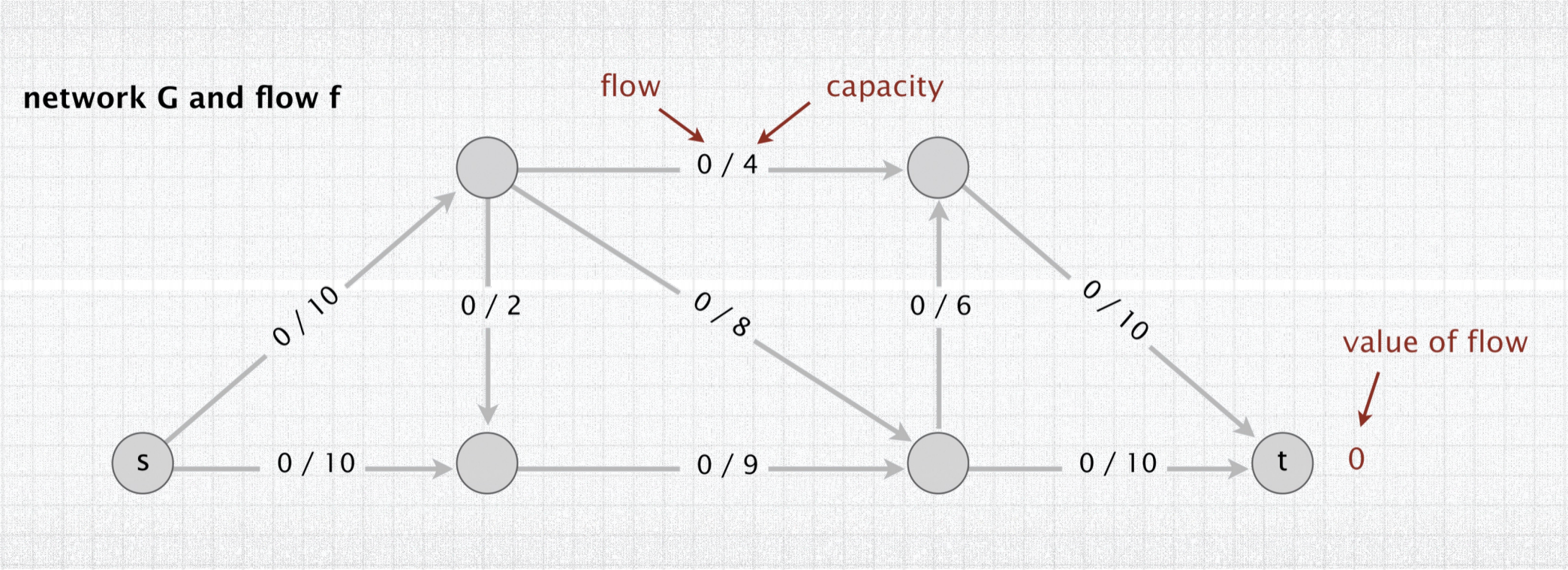
RETURN f.

augmenting path

Final Review

Ford–Fulkerson Algorithm

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P in the residual network G_f .
- Augment flow along path P.
- Repeat until you get stuck.



Final Review

Max-Flow Min-Cut Theorem

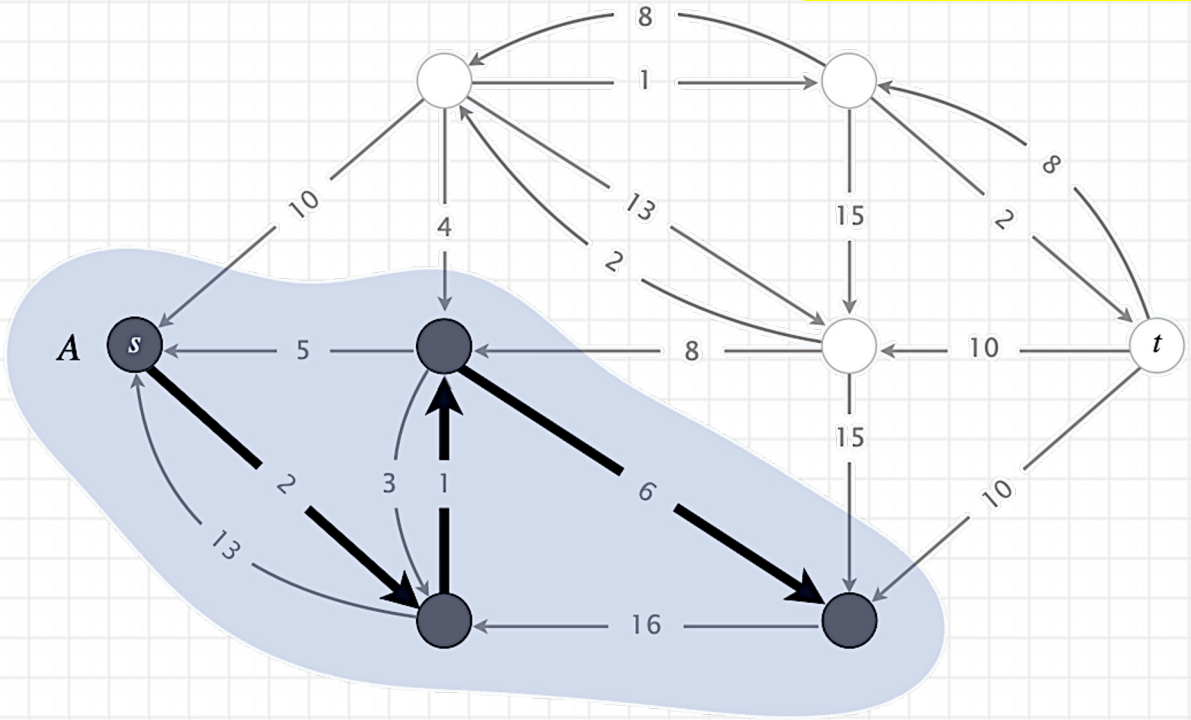
- Max-flow min-cut theorem: Value of a max flow = capacity of a min cut
- Augmenting path theorem: A flow f is a max flow iff no augmenting paths.
- Proof : The following three conditions are equivalent for any flow f :
  1. There exists a cut (A, B) such that $cap(A, B) = val(f)$.
  2. f is a max flow.
  3. There is no augmenting path with respect to f.

if Ford–Fulkerson terminates, then f is max flow

Final Review

Max-Flow Min-Cut Theorem

- Computing a minimum cut from a maximum flow
- Theorem. Given any max flow f, can compute a min cut (A, B) in $O(|E|)$ time.
- Proof. Let A = set of nodes reachable from s in residual network G_f. ▪



Final Review

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 of flow network in Final Exam:

- Short answers /Definition/ True/False
- A flow network with the values of flow/capacity of each edge is given
  - Short answer questions regarding the given flow
  - Performing one/two step(s) of the Ford-Fulkerson algorithm to find the maxflow
  - Give a min-cut
  - Similar to HW5/Q3

- 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

- 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

Final Exam

- Contents:  Inclusive (including all discussed topics)
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming
  - Graph algorithms (Traversal, applications, MST, shortest path, flow network)
  - NP-completeness

Final Review

NP-Completeness

- So far, all algorithms we have seen could solve the given problem in polynomial time → complexity class "P"
  - Problems in P are considered tractable
  - Problems not in P are intractable

- NP-completeness is a form of bad news!
  - There exist many important problems that cannot be solved quickly.

Final Review

Optimization vs. Decision Problems

- **Decision problems**
  - Given an input and a question regarding a problem, determine if the answer is yes or no
- **Optimization problems**
  - Find a solution with the "best" value
- Optimization problems can be cast as decision problems that are easier to study

Final Review

Class "P"

- **Class P** consists of [decision] problems that are solvable in polynomial time
- Recall from the first lecture:
  - [slide #36] Polynomial time → Running time is $O(n^k)$ for some constant $k > 0$.
  - Examples
    - Linear search O(n)
    - Dynamic programming solutions (O(n), O(n^2) , O(n^3), …)
    - Sorting (O(n^2), O(nlogn))
    - Divide-and-conquer solutions
    - Graph algorithms O(n+m), O(mlogn), …
  - Non-polynomial time → $O(2^n), O(a^n), O(n!), O(n^n)$, ...

Problems in P are Considered/called tractable

Problem not in P are intractable

Final Review

Class "NP"

- First of all: **NP does NOT stand for not-P!**

NP = Nondeterministic Polynomial

- NP is the class of problems for which a candidate solution can be verified in polynomial time.

- P is a subset of NP ($P \subseteq NP$)

Final Review

Class "NP"

- Nondeterministic algorithms entail a two-stage procedure:

Note in NP algorithms the verification step is polynomial

1. Nondeterministic "guessing" stage
   - Generate randomly an arbitrary candidate solution (≡ "certificate")
2. Deterministic "verifying" stage
   - Take the certificate and the instance to the problem and returns YES if the certificate represents a solution (verifying in polynomial time)

Final Review

P vs. NP

- Is P = NP?
- Mentioned earlier that any problem in P is also in NP. So, P is a subset of NP ($P \subseteq NP$)
- But the big (and open) question is whether $NP \subseteq P$, and so P=NP.
  - It means if it is always easy to check a candidate solution, should it also be easy to find a solution?
  - Answer? Most computer scientists believe that this is false, but we do not have a proof

Final Review

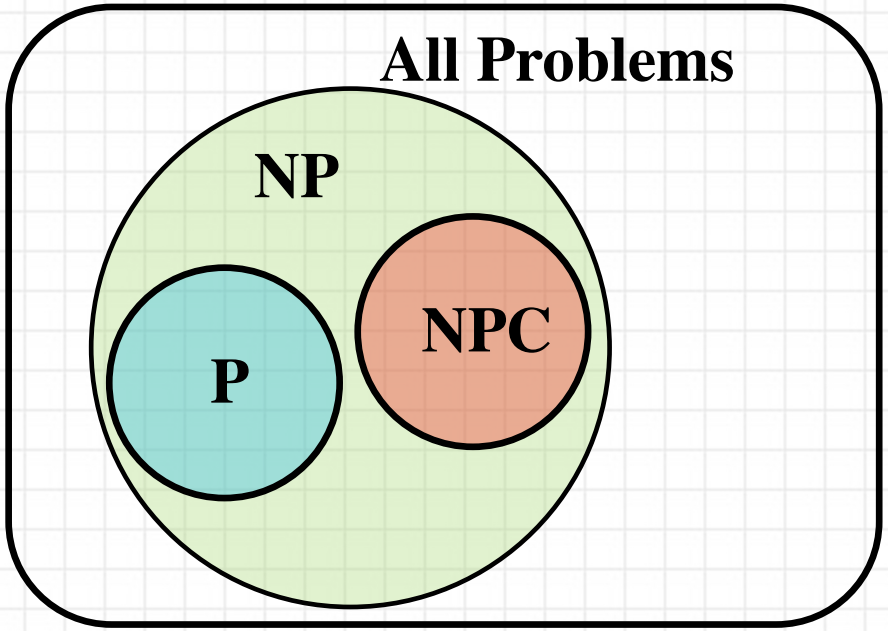
NP-Complete (NPC)

- NP-complete problems are a class of "hardest" problems in NP.
- If you can solve an NP-complete problem, then you can solve all NP problems (show later).
- Hence, if any NP-complete problem can be solved in polynomial time, then all problems in NP can be, and thus P = NP.
- Precise/formal definition coming later...

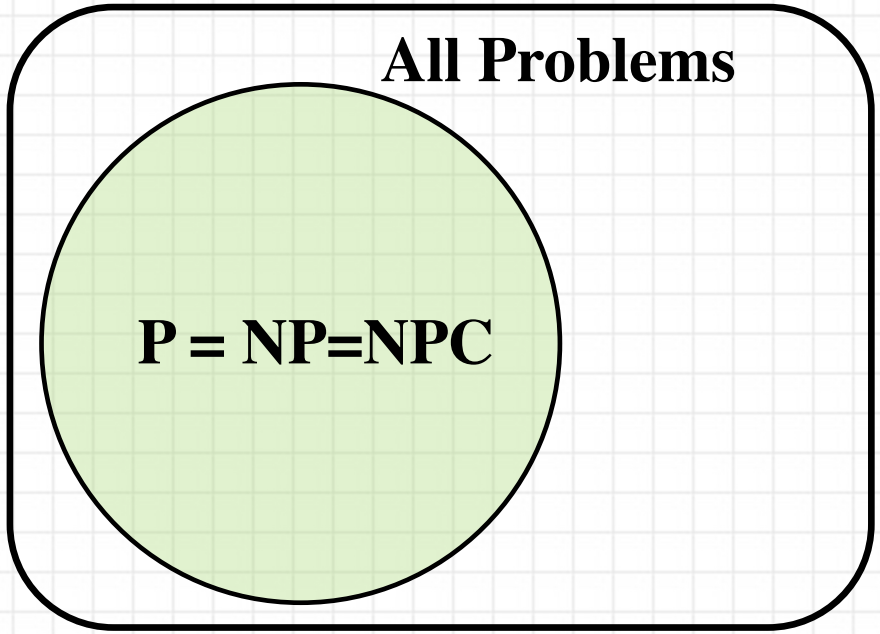
Final Review

Possible Worlds

- Therefore, there are two possibilities:



P≠NP

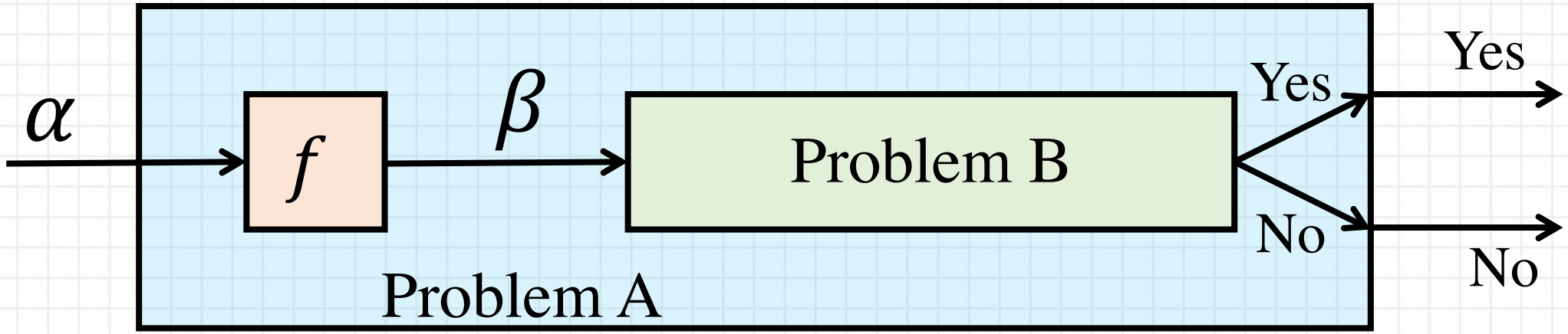


P=NP

Final Review

Reductions

- $A \leq B$: Reduction from A to B is showing that we can solve A using the algorithm that solves B
- If we have an oracle for solving B, then we can solve A by making polynomial number of computations and polynomial number of calls to the oracle for B
- We can transform the inputs of A to inputs of B



Final Review

Polynomial Reductions

- Given two problems, A and B, we say that A is polynomially reducible to B, and write it as $A \leq_p B$ if:

1. There exists a function f that converts the input of A **to** inputs of B in polynomial time

2. $A(i) = \text{YES} \iff B(f(i)) = \text{YES}$

Final Review

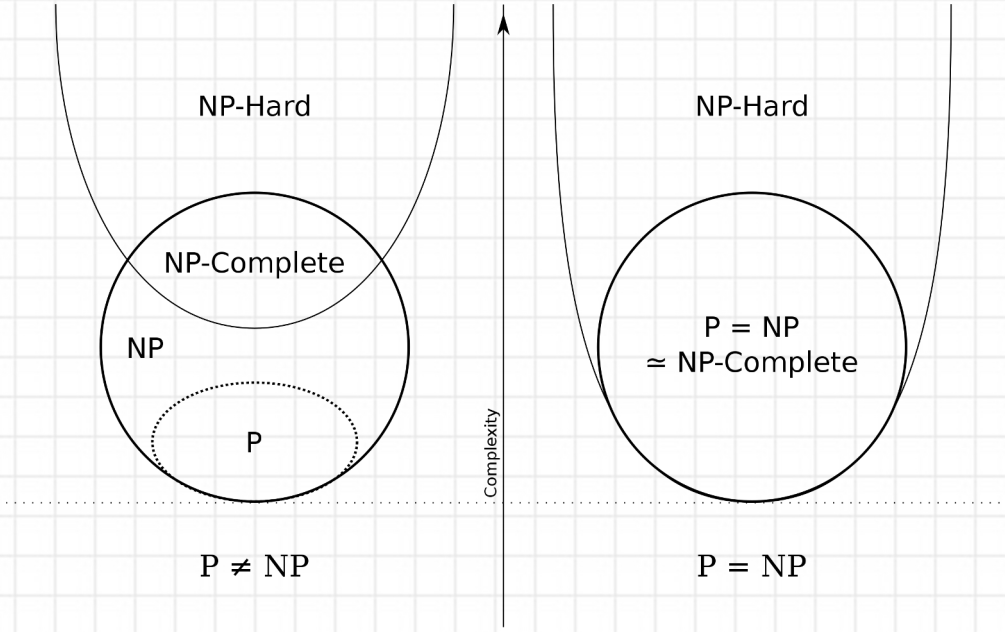
Implications of Polynomial-Time Reductions

- **Purpose**. Classify problems according to relative difficulty.
- **Design algorithms**. If $X \leq_p Y$ and Y can be solved in polynomial-time, then X can also be solved in polynomial time.
- **Establish intractability**. If $X \leq_p Y$ and X cannot be solved in polynomial-time, then Y cannot be solved in polynomial time.
- **Establish equivalence**. If $X \leq_p Y$ and $Y \leq_p X$, we use notation $X \equiv_p Y$.
- **Transitivity**. If $X \leq_p Y$ and $Y \leq_p Z$,then $X \leq_p Z$.

Final Review

NP-Completeness (Formal Definition)

- A problem Y is **NP-hard** if $X \leq_p Y$ for all $X \in$ **NP**
  - A problem is NP-hard if and only if a polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NP
  - NP-hard problems are at least as hard as any NP problem
- A problem Y is **NP-complete** if:
  1. $Y \in$ **NP**
  2. Y is **NP-hard**



https://en.wikipedia.org/wiki/P_versus_NP_problem

Final Review

Establishing NP-Completeness

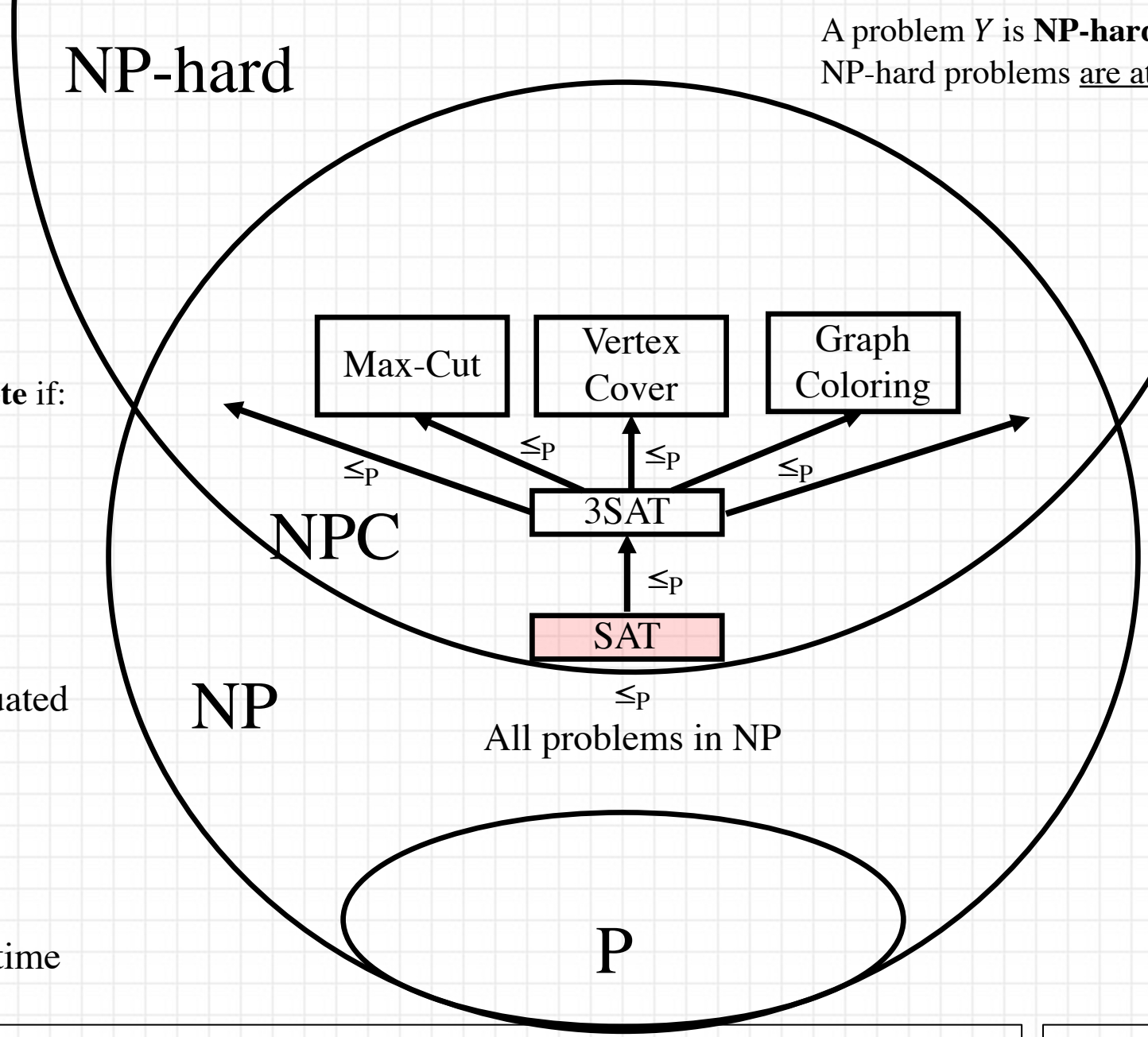
- Recipe to establish NP-completeness of problem Y.
  - Step1. Show that Y is in NP. ($Y \in$ **NP**)
    - Describe how a potential solution will be represented
    - Describe a procedure to check whether the potential solution is a correct solution to the problem instance, and argue that this procedure takes polynomial time
  - Step 2. Choose an NP-complete problem X.
  - Step 3. Prove that $X \leq_p Y$ (X is **poly-time reducible** to Y).
    - Describe a procedure f that converts the inputs i of X to inputs of Y in polynomial time
    - Show that the reduction is correct by showing that $X(i) = \text{YES} \iff Y(f(i)) = \text{YES}$

      Note this is an "if and only if" condition, so proofs are needed for both directions.

Final Review

A problem Y is **NP-hard** if $X \leq_p Y$ for all $X \in$ **NP**
NP-hard problems are at least as hard as any NP problem

A problem Y is **NP-complete** if:
1. Y ∈ **NP**
2. Y is **NP-hard**



Once we establish the first "natural" NPC problem, others fall like dominoes!

All problems in NP can polynomially be reduced to SAT

SAT is at least as hard as all problems in NP

Candidate can be evaluated in poly-time

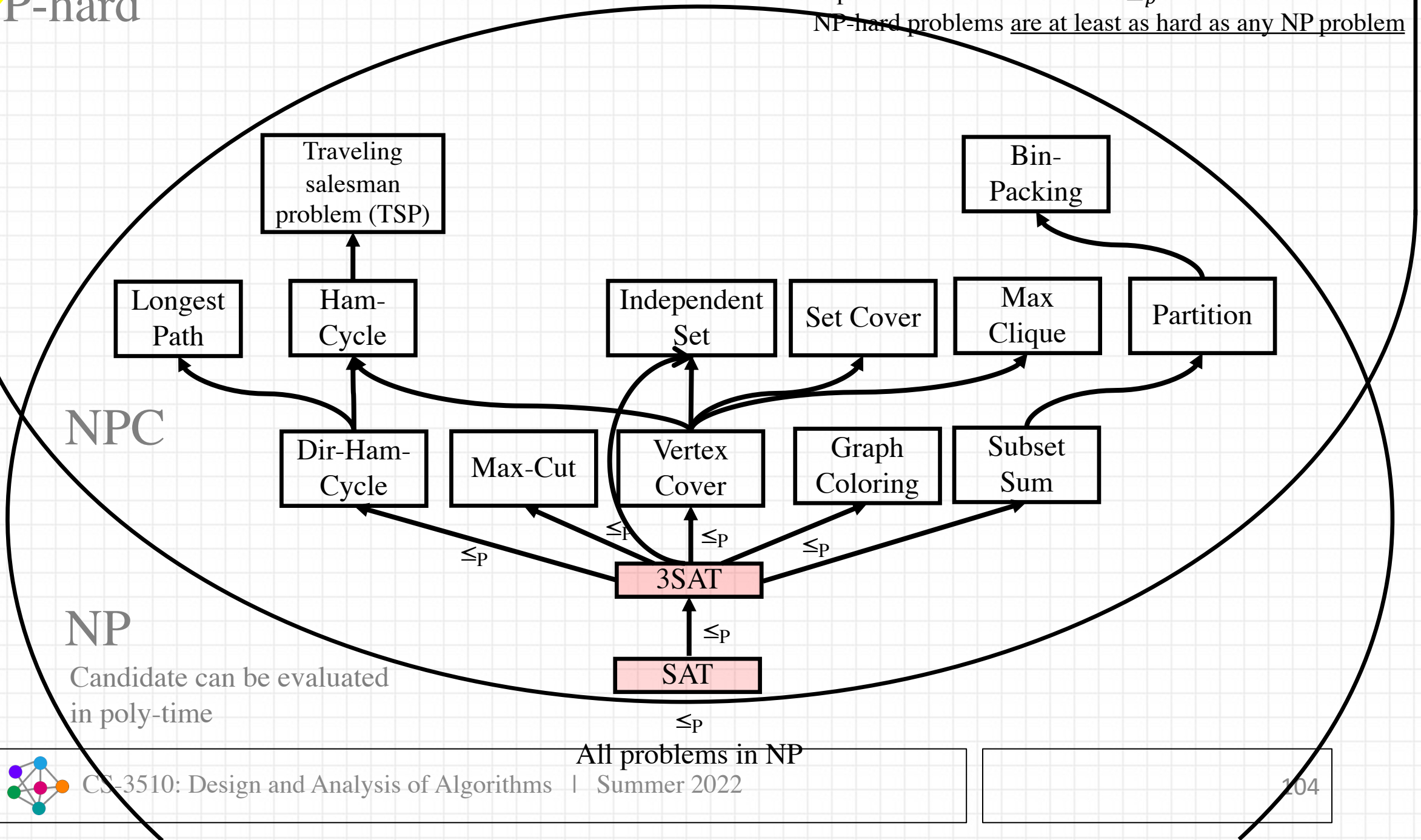
Solvable in poly-time

Final Review

NP-hard

A problem Y is **NP-hard** if $X \leq_p Y$ for all $X \in$ **NP**

NP-hard problems are at least as hard as any NP problem



Final Review

Genres of NP-complete problems

- **Six basic genres of NPC problems and paradigmatic examples.**

1. Constraint satisfaction problems: SAT, 3-SAT.
2. Packing problems: SET-PACKING, INDEPENDENT SET.
3. Covering problems: SET-COVER, VERTEX-COVER.
4. Sequencing problems: HAMILTONIAN-CYCLE, TSP.
5. Partitioning problems: 3-COLOR, 3D-MATCHING.
6. Numerical problems: 2-PARTITION, SUBSET-SUM, KNAPSACK.

Final Review

3SAT Problem

- We want to show that 3SAT is an NP-complete problem.

Recipe to establish <u>NP-completeness of problem Y.</u>

Step1. Show that Y is in NP. ($Y \in$ **NP**)

Describe how a potential solution will be represented

Describe a procedure to check whether the potential solution is a correct solution to the problem instance, and argue that this procedure takes polynomial time

Y → 3SAT

Step 2. Choose <u>an</u> NP-complete problem X.

X → SAT

Step 3. Prove that $X \leq_p Y$ (X is **poly-time reducible** to Y).

Describe a procedure f that converts the inputs i of X to inputs of Y in polynomial time

Show that the reduction is correct by showing that $\boxed{X(i) = \text{YES} \iff Y(f(i)) = \text{YES}}$

Note this is an "if and only if" condition, so proofs are needed for both directions.

SAT $\leq_p$ 3SAT

Final Review

Independent Set Problem

- We want to show Independent Set (IS) problem is an NPC problem

Recipe to establish <u>NP-completeness of problem Y.</u>

Step1. Show that Y is in NP. ($Y \in$ **NP**) — **Y → IS**

Describe how a potential solution will be represented

Describe a procedure to check whether the potential solution is a correct solution to the problem instance, and argue that this procedure takes polynomial time

Step 2. Choose <u>an</u> NP-complete problem X. — **X → 3SAT**

Step 3. Prove that $X \leq_p Y$ (X is **poly-time reducible** to Y). — **3SAT $\leq_p$ IS**

Describe a procedure f that converts the inputs i of X to inputs of Y in polynomial time

Show that the reduction is correct by showing that $\boxed{X(i) = \text{YES} \iff Y(f(i)) = \text{YES}}$

Note this is an "if and only if" condition, so proofs are needed for both directions.

Final Review

3-Colorability Problem

- `3-COLOR`: Given an undirected graph G does there exists a way to color the nodes using at most three colors (e.g., red, green, and blue) so that no adjacent nodes have the same color?
- We want to show `3-COLOR` problem is an NPC problem.

Recipe to establish NP-completeness of problem Y.

Step1. Show that Y is in NP. ($Y \in$ **NP**)

Step 2. Choose an NP-complete problem X.

Step 3. Prove that $X \leq_p Y$ (X is **poly-time reducible** to Y).

Describe a procedure f that converts the inputs i of X to inputs of Y in polynomial time

Show that the reduction is correct by showing that $X(i) = \text{YES} \iff Y(f(i)) = \text{YES}$

Note this is an "if and only if" condition, so proofs are needed for both directions.

Y → 3-COLOR

X → 3SAT

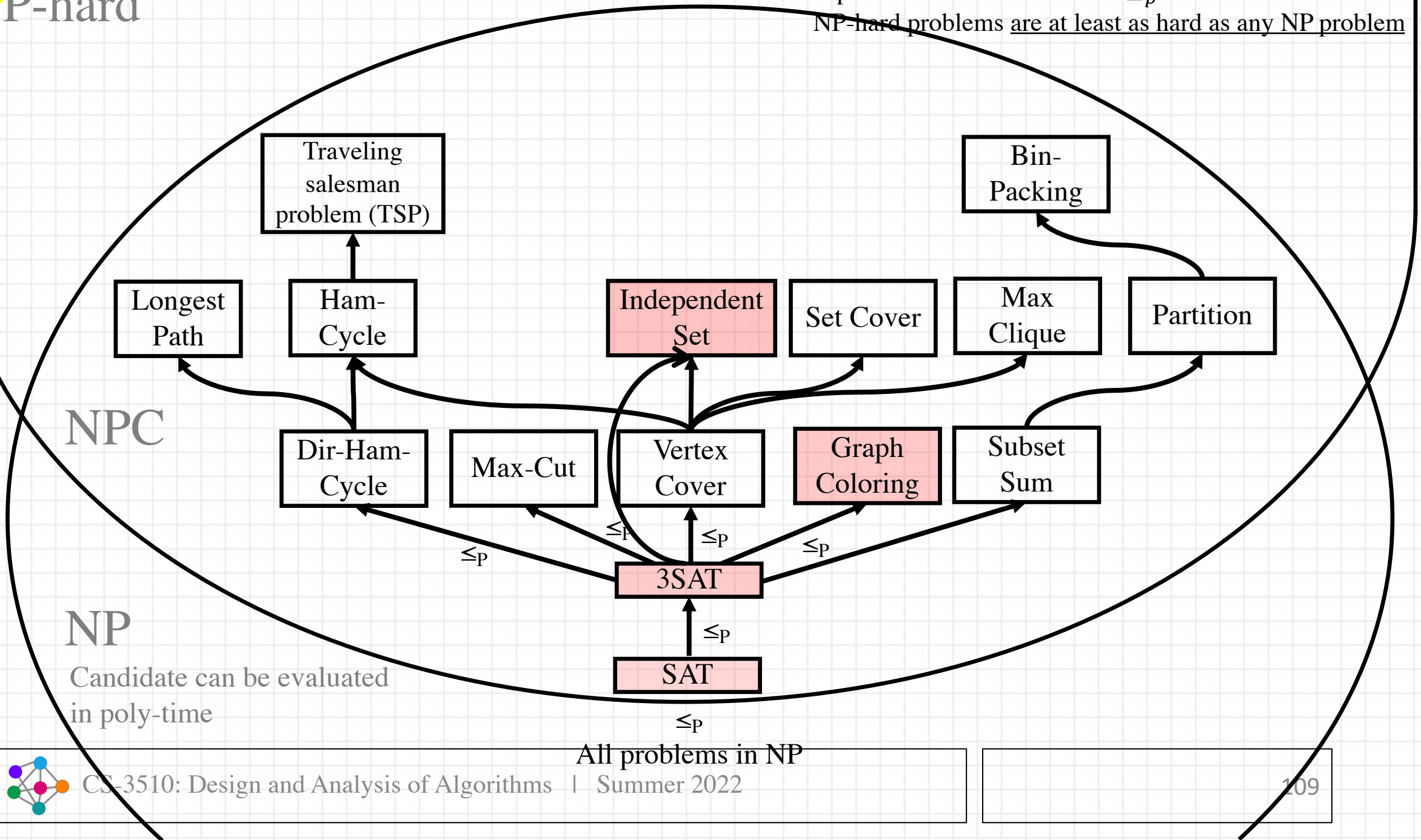
3SAT $\leq_p$ 3-COLOR

Final Review

NP-hard

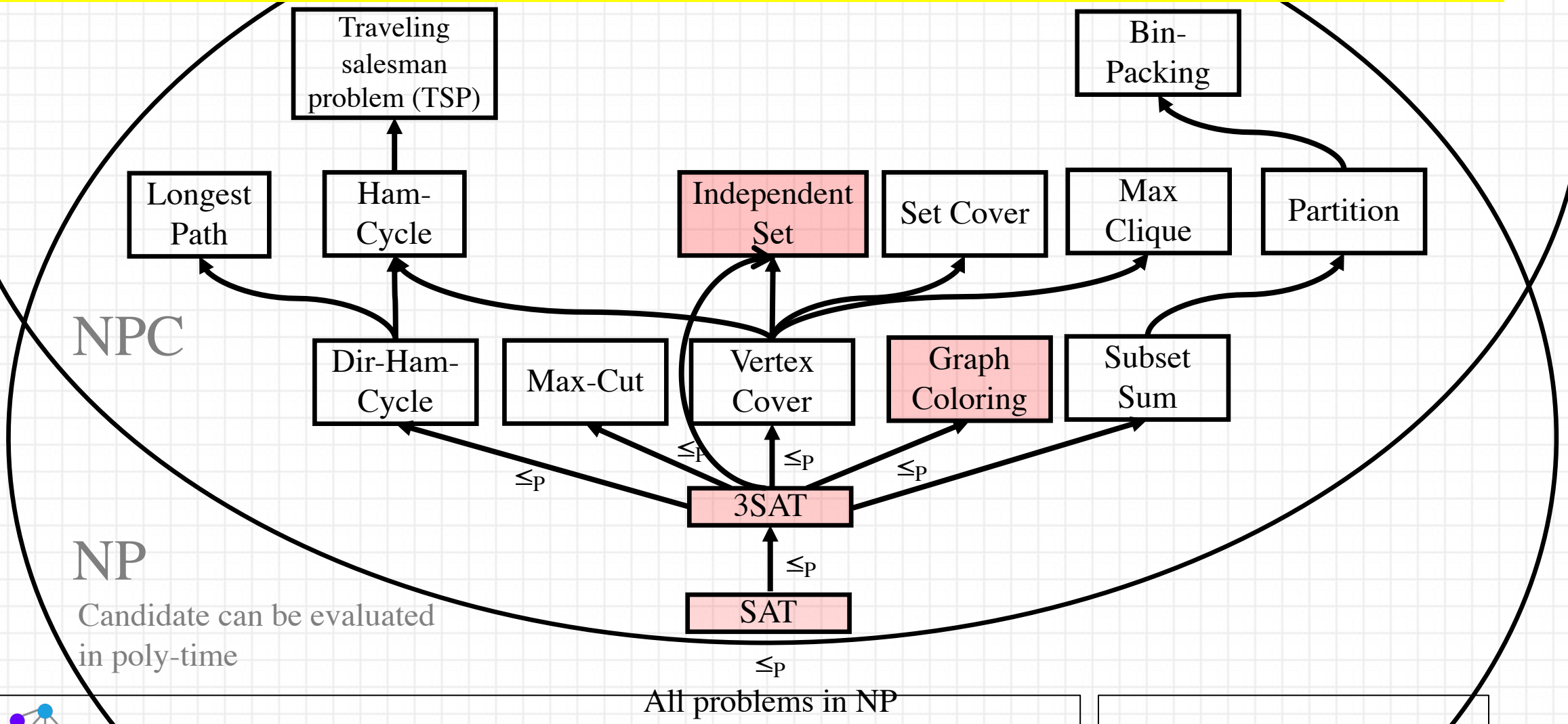
A problem Y is **NP-hard** if $X \leq_p Y$ for all $X \in$ **NP**

NP-hard problems are at least as hard as any NP problem



Type of Questions of NP-completeness in Final Exam:

- Short answers /Definition/ True/False
- ~~NPC establishment problems~~



Preparation for Exam

- Suggested preparation steps:
  - Start with lecture slides, comprehend step-by-step solutions/algorithms.
    - Make sure you have downloaded the latest version of slides. (Minor updates in Lec8, 9, and 10)
  - Textbook suggested readings.
  - Run the demo codes and print step-by-step computations/results
    - Particularly, helpful for graph-related algorithms.
  - Homework assignments → HW5
  - Practice problems

Final Exam: 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]	07/12	07/22

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

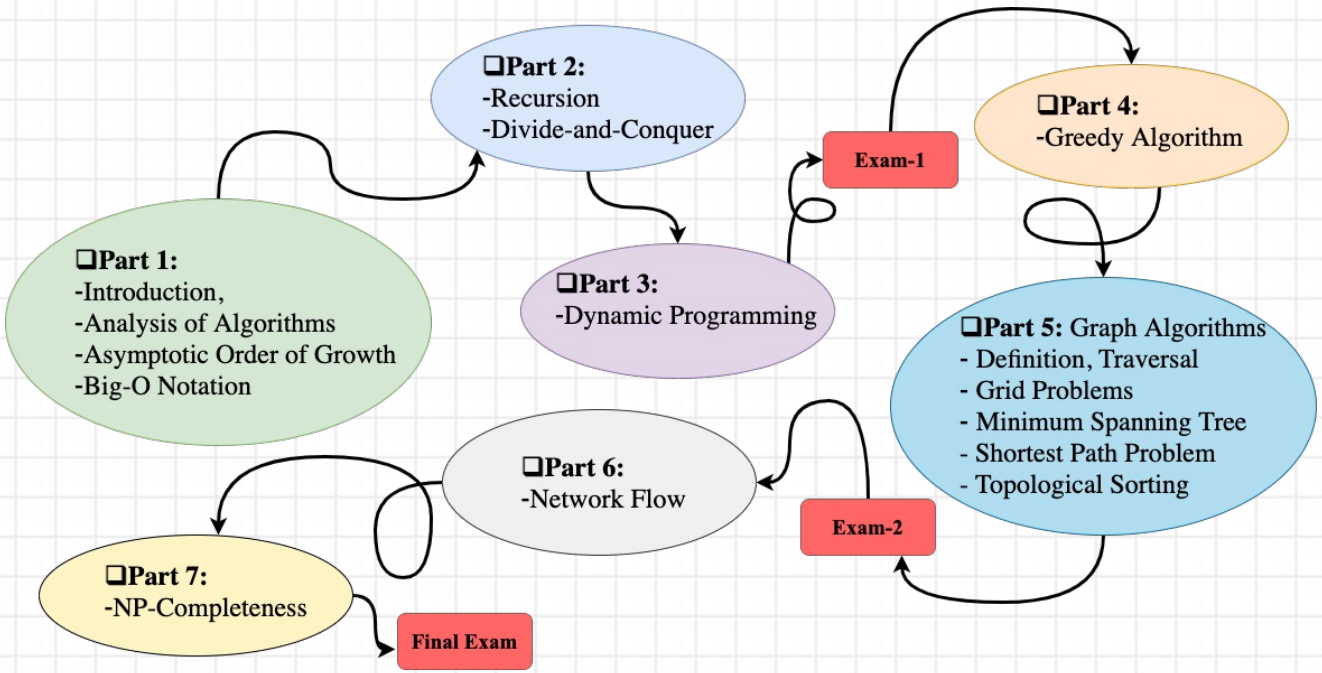
Final Exam

- Problem 1: Short Answer Questions
- Problem 2: Divide-and-Conquer
- Problem 3: Dynamic Programming
- Problem 4: Shortest Path Algorithms
- Problem 5: Graph-related (Traversal, MST, Shortest Path)
- Problem 6: Flow Network

- 80 points (+ bonus perhaps)
- Time ~120 minutes

Final Words …

- Congratulations! We made it!



Design and Analysis of Algorithms

What?

Algorithms, Algorithmic paradigms; design and correctness

Performance analysis

Why?

Fundamental to all areas of computer science

Operating systems, Networks and distributed systems, Machine learning, Data science, Numerical computation, Cryptography, Computational biology, etc.

Inseparable part of every technical interview

Internship, Part-time, Full-time

Useful and Fun!

Problem solving skills

Competitive programming, Hackathons, etc.

Final Words …

- Congratulations! We made it!
- I hope you have enjoyed the course as much as I enjoyed teaching it.
- Course materials
  - Lecture notes and recordings, demo codes, assignments, and exams
  - The order of the topic presented
  - Covered future paths
    - Theory, graduate studies, …
    - Internship, fulltime jobs (SWE, PM, ML, …)
- Course website remains available through the same address.
  - http://www.cs3510.com
  - A summary/gist is added for future use and reference

Final Words …

- More importantly, I hope this course has made you more interested in algorithm design and related topics
- Lecture notes and recording
  - Course plan and roadmap
  - The recording option was not provided by CoC, but as requested by the majority, we did it anyway with some technical difficulties (zoom meeting, recording, YouTube upload, …).
  - No mandatory class attendance
- Exam and assignments
  - Reduced number of assignments / adjusted workload for a summer semester
  - Reduced number of problems → More practice problems
  - Consistency between lecture notes, assignments, and exam problems
  - Clear exam/evaluation plan
  - Flexible deadlines, exam times, …

Learning and enjoying the concepts is what matters most…!

Final Words …

- Please let us know what you think about the course in general, and if you have any comments and/or suggestions about the course materials, presentation, etc.

- Course Instructor Opinion Surveys (CIOS)
  - Available from 07/18 until 08/07

Thank you!

- **Algorithms are fun!**
  - Amazing experience teaching this class
  - Thanks to all of you and TAs
  - Hope you've had fun and learned useful material

- **My own favorite topics in computer science**
  - Algorithms [along with Data Structures]
  - Machine Learning/Deep Learning

- **Stay in touch!** Do not hesitate to contact me after this course
  - Questions, codes, interview experiences/questions, …
  - Email, personal website, LinkedIn, GitHub

- **Wish you all the best for the future!**