ShanghaiTech University

CS101 Algorithms and Data Structures

Fall 2024

Final Exam

Instructors: Dengji Zhao, Yuyao Zhang, Xin Liu, Hao Geng

Time: Jan 8th 8:00-10:00

INSTRUCTIONS

Please read and follow the following instructions:

- You have 120 minutes to answer the questions.
- You are not allowed to bring any papers, books, or electronic devices including regular calculators.
- You are not allowed to discuss or share anything with others during the exam.
- You should write the answer to every problem in the dedicated box **clearly**.
- You should write **your name and your student ID** as indicated on the top of **each page** of the exam sheet.
- If you need more space, write "Continued on Page #" and continue your solution on the referenced scratch page at the end of the exam sheet.

Name	
Student ID	
Exam Classroom Number	
Seat Number	
(please copy this and sign)	All the work on this exam is my own.

THIS PAGE INTENTIONALLY LEFT BLANK.

DO NOT WRITE ANY ANSWER IN THIS PAGE!

1. (18 points) Single Choice

Each question has **exactly one** correct answer. Fill your answers **in the box below**.

Notice: Fulfill your answer or we may take it unspecified.

(a)	◯ A	◯ B	◯ C	◯ D	(b)	◯ A	◯ B	◯ C	◯ D
(c)	◯ A	◯ B	◯ C	◯ D	(d)	◯ A	◯ B	◯ C	◯ D
(e)	◯ A	◯ B	◯ C	◯ D	(f)	◯ A	◯ B	◯ C	◯ D

(a) (3') Consider a disjoint set of size 6 - with path compression but **no** union by rank. The disjoint set is initialized by `parent[i] = i` for $i \in [1, 6]$, and has 2 operations:

- `find(int i)`: Return the root of the tree that contains `i`, while do path compression.
- `union(int i, int j)`: `parent[find(i)] = find(j)`.

Now given a sequence of operations, `union(1, 2)`, `union(3, 4)`, `union(5, 4)`, `union(4, 2)`, `union(6, 5)`. What is the height of the disjoint-set tree that contains 2?

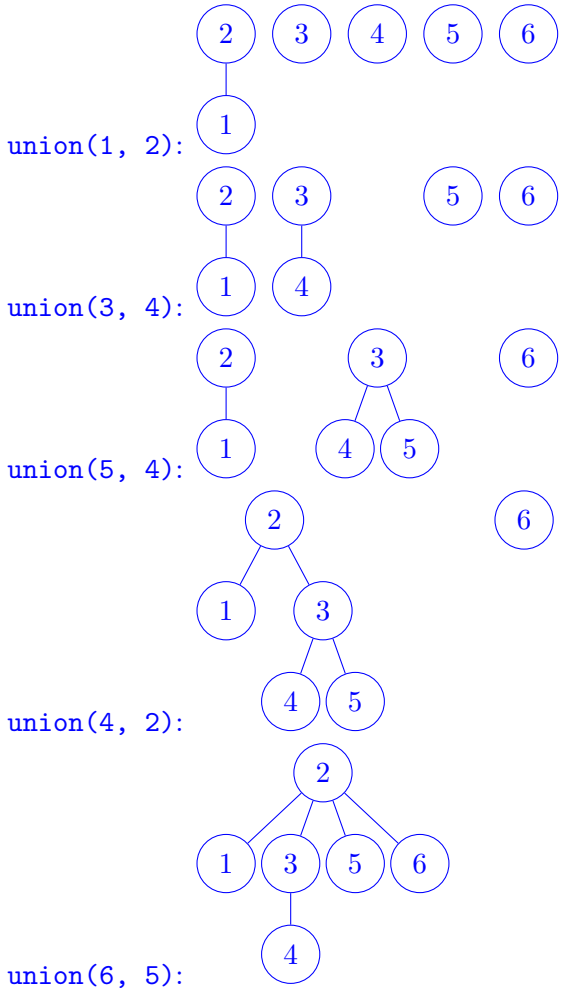
A. 1

B. 2

C. 3

D. 4

Solution:



(b) (3') Given an undirected graph $G = (V, E)$ whose adjacency matrix is A ($\forall v_i, v_j \in V$, $A_{ij} = 1$ if $(v_i, v_j) \in E$ and 0 otherwise). Which of the following statements is false?

A. $A = A^T$ i.e. A is symmetric.

B. $\forall\, v_i \in V$, $(A^2)_{ii} = \deg(v_i)$.

C. $tr(A^2) = 2|E|$.

D. $\sum_{v_i, v_j \in V} A_{ij} = |E|$.

Solution:

D. $2|E|$.

(c) (3') Which of the following statements about topological sort is correct?

1. Topological sort can be applied to any connected graph.
2. Topological sort is used to find (one of the) shortest paths in a weighted graph.
3. Topological sort is a linear ordering of vertices in a directed acyclic graph, where for every directed edge $(u \to v)$ from vertex u to vertex v, u comes before v in the ordering.
4. Topological sort can result in multiple valid orderings for the same graph.

A. Only 1 and 2

B. Only 3 and 4

C. Only 1, 2 and 4

D. All of the above

Solution:

1. No. Topological sort can only be applied to any DAG.
2. No. Topological sort is used to find a topological order. Only on a DAG you can find a shortest path by a DP algorithm based on a topological order.
3. Yes. That's the definition.
4. Yes. Many examples.

(d) (3') To find the minimum number of moves for the N-puzzle problem, the fastest one of the following algorithms is

	3	2
8	7	1
4	5	6

$\Longrightarrow$

1	2	3
4	5	6
7	8	

A. BFS

B. DFS

C. Dijkstra

D. A*

Solution: A* is informed search, which can reduce the number of searched states.

(e) (3') Given n points $\{p_i = (x_i, y_i)\}$ in a plane, we can obtain a weighted complete graph $G = (V, E)$, where the vertices correspond to points in the plane and the weights of the edges $e = (v_i, v_j)$ are equal to the distance between pairs of points p_i and p_j i.e. $w(e) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Which of the following statements is true?
Hint: $|V| = n, |E| = \frac{n(n-1)}{2}$.

A. Using Kruskal's algorithm can find the MST more efficiently, which takes at most $\Theta(n^2 \log n)$ time.

B. Using Prim's algorithm can find the MST more efficiently, which takes at most $\Theta(n^2 \log n)$ time.

C. We can use L-∞ Distance i.e. $\max(|x_i - x_j|, |y_i - y_j|)$ as a consistent heuristic function to do A^* graph search on this graph.

D. We can use Manhattan Distance i.e. $|x_i - x_j| + |y_i - y_j|$ as a consistent heuristic function to do A^* graph search on this graph.

Solution:

A.B. Prim is $\Theta(n^2)$ without heap or using Fibonacci heap, while Kruskal is $\Theta(n^2 \log n)$ which is less efficient.

C. We can prove consistency, i.e. suppose t is the target vertex, for any edge $u \to v$:

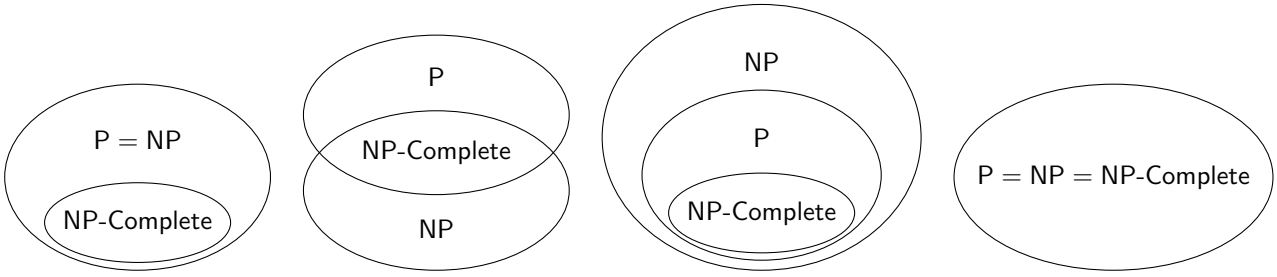
$$\max(|x_u - x_t|, |y_u - y_t|) \le \max(|x_v - x_t|, |y_v - y_t|) + \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}$$

W.l.o.g. suppose $|x_u - x_t| \ge |y_u - y_t|$, then

$$\begin{aligned}\max(|x_u - x_t|, |y_u - y_t|) =& |x_u - x_t| \\ \le& |x_v - x_t| + |x_u - x_v| \\ \le& |x_v - x_t| + \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} \\ \le& \max(|x_v - x_t|, |y_v - y_t|) + \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}\end{aligned}$$

D. Counterexample: $(x_t, y_t) = (0, 0)$, then $h(u) = |x_u| + |y_u| \ge \sqrt{x_u^2 + y_u^2}$, which overestimates the distance, which is not admissible, therefore impossible to be consistent.

(f) (3') Suppose zsc2003 has found an algorithm that correctly solves the Subset-Sum problem in polynomial time. Which one of the following Venn diagrams can correctly represent the relationship among P, NP, and NP-Complete under this circumstance? (Suppose all problems involved both have yes-instances and no-instances.)



A. B. C. D.

Solution: D. Subset − Sum ∈ P ⇒ 3-SAT ∈ P ⇒ $\forall A \in$ P, 3-SAT $\leq_p A \Rightarrow \forall A \in$ P, $A \in$ NP-Complete. $\forall A \in$ NP-Complete, $A \leq_p$ Subset − Sum $\in P \Rightarrow \forall A \in$ NP-Complete, $A \in P$.

2. (20 points) Multiple Choices

Each question has **one or more** correct answer(s). Select all the correct answer(s). For each question, you will get 0 points if you select one or more wrong answers, but you will get 2 points if you select a non-empty subset of the correct answers. Fill your answers **in the box below**.

Notice: Fulfill your answer or we may take it unspecified.

(a)	◯ A	◯ B	◯ C	◯ D	(b)	◯ A	◯ B	◯ C	◯ D
(c)	◯ A	◯ B	◯ C	◯ D	(d)	◯ A	◯ B	◯ C	◯ D

(a) (5') Which of the following statements about the minimum spanning tree is correct?

A. If a connected graph has $|V|$ nodes and $|E|$ edges, all edges have non-negative weights, then we can use the Dijkstra algorithm with a binary heap in $O(|E| \log |V|)$ time to get the minimum spanning tree.

B. If we do Prim's algorithm from one vertex and get a unique MST, we can also get the shortest path from the vertex to other vertices.

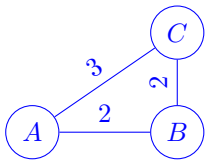
C. We can get the maximum spanning tree of a graph by negating the edge weights and running any minimum spanning tree algorithm on it.

D. After we pick up all the edges in the shortest path i.e. $d(u) + w(u, v) = d(v)$, they will form a tree if the graph is connected and undirected.

Solution:

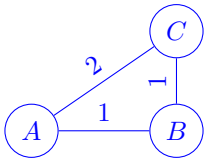
A. Dijkstra is shortest path algorithm.

B. Counterexample:

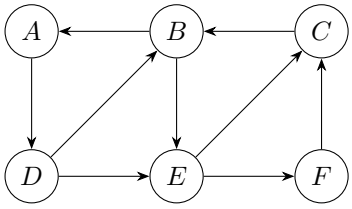


The shortest path between A, C, which is not on MST.

D. They may form a graph instead of a tree. Counterexample:



(b) (5') Suppose we have the following graph, and we perform a Depth-First Search (DFS) or Breadth-First Search (BFS) starting from a specified vertex. Below are some possible orders in which the vertices might be visited during the traversal. Which of these orders could actually occur, based on the rules of DFS or BFS?



A. DFS starting from A, order: ADEFBC

B. DFS starting from D, order: DBAECF

C. BFS starting from B, order: BEAFCD

D. BFS starting from E, order: EFCBAD

Solution:

A. C should be visited before B.

(c) (5') For a simple positive-weighted directed graph, if the heuristic $h(\cdot)$ in A* algorithm is consistent, which of the following statements is/are TRUE? Here are some notations:

- $w(u, v)$ is weight of edge $u \to v$.
- $d(u, v)$ is the actual length of shorted path from u to v.
- $p(u) = d(s, u) + h(u)$ is the priority in GRAPH-SEARCH, where s is the source vertex.

A. $h(u) \leq w(u, v) + h(v)$ for any edge $u \to v$.

B. $h(u) \leq d(u, v) + h(v)$ for any two vertices u, v such that there is a path from u to v.

C. $p(u)$ is non-decreasing during the iteration of u in GRAPH-SEARCH.

D. $h(\cdot)$ is admissible.

Solution:

A. It's the definition.

B. Suppose the actual shortest path between u, v is $(u, a_1, a_2, \cdots, a_k, v)$, then the following inequalities can be implied by A.:

$$\begin{aligned} h(u) &\leq w(u, a_1) + \cancel{h(a_1)} \\ \cancel{h(a_1)} &\leq w(a_1, a_2) + \cancel{h(a_2)} \\ &\cdots \\ \cancel{h(a_k)} &\leq w(a_k, v) + h(v) \end{aligned}$$

By adding them we can prove that $h(u) \leq d(u, v) + h(v)$.

C. Proved in lecture slides.

D. Consistent $\implies$ admissible. Also can be implied by B. because $h(u) \leq d(u, t) + h(t) = d(u, t)$ where t is the target vertex.

(d) (5') Which of the following statements must be true?

A. Consider a problem where the input is an integer $x \geq 0$. If the time complexity of this problem is $O(x)$, then it's in P.

B. Consider two problems X and Y. Suppose that $X \leq_p Y$, then X is in NP-Complete only if Y is in NP-Complete.

C. Consider two problems X and Y. Suppose that $X \leq_p Y$, then X is in P only if Y is in P.

D. If $X \in$ NP-Complete, $Y \in$ NP, $X \leq_p Y$, then $Y \notin$ P can prove P $\neq$ NP.

Solution:

A. No.

B. X, Y may not be in NP.

C. From Lecture 26-28 Page 9(D).

D. Lecture 25-27 Page 69(B).

3. (12 points) Adjacency matrix

One basic usage of an adjacency matrix is to represent a graph. However, it also has other applications in graph theory. In this question, we only consider **simple undirected unweighted graph**. We use G to denote a graph, and A to denote its adjacency matrix.

By using dynamic programming, we can know that $(\mathbf{A^n})_{\mathbf{ij}}$ **is equal to the number of different paths from** $\mathbf{v_i}$ **to** $\mathbf{v_j}$ **with length n**. You can use this conclusion in this question without any proof.

(a) **Basic properties**

i. (2') The i_{th} diagonal element of A^2 represents ___$deg(v_i)$___.

(b) **Counting triangles**

Let's start with a simple task! You are required to compute the number of triangles in a graph. In a graph, a triangle is defined as a cycle with **exactly** 3 vertices in it.

i. (3') Since there is no self-loop in this graph, a path with length 3 from a vertex v_i to itself must be a contour on a triangle which includes v_i, and a triangle will include **exactly** ___**3**___ different vertices. So the total number of triangles in this graph is $\frac{1}{3} \cdot \sum_{i=1}^{|V|} \frac{t_i}{2}$, where $t_i =$ ___$(A^3)_{ii}$___.

(c) **Counting pentagons**

Now we want to calculate the number of pentagons in a graph. In a graph, a pentagon is defined as a cycle with **exactly** 5 vertices in it.

i. (2') For a graph with no triangle in it, write a simple expression to calculate the number of pentagons in that graph. You can use A to represent the adjacency matrix of the given graph. (Hint: $tr(M) = \sum_{i=1}^{k} M_{ii}$, where M is an $k \times k$ matrix)

Solution: $\frac{tr(A^5)}{10}$

ii. (3') Prove the correctness of your solution.

Solution:
Let $P = A^5$. Then P_{ii} represents the number of paths from v_i to v_i with length 5. These paths can only be a contour on a pentagon which includes v_i, because there is no triangle in the graph. And for one such pentagon, there will be exactly 2 paths from v_i to v_i. So P_{ii} is equal to $2p_i$, where $2p_i$ is the number of pentagon which includes v_i. Through a single pentagon, there will be exactly 5 vertices which can contour to itself with 5 steps. So the answer is $\frac{\sum_{i=1}^{n} \frac{P_{ii}}{2}}{5} = \frac{tr(A^5)}{10}$.
1' for mentioning no triangle, 1' for mentioning 5 inclusions per pentagon, 1' for mentioning 2 paths per inclusion

(d) **Not always easy...**

For a graph with triangles in it, it is not so easy to calculate the number of all pentagons. But it's relatively easy to calculate the number of pentagons which includes some certain nodes.

i. (2') Suppose you are given a graph G with n vertices and its adjacency matrix A. Given a node $v_i \in G$ which is not included in any triangle, what is the number of pentagons that includes v_i? ___**B**___

A. $\frac{(A^5)_{ii}}{2}$ **B.** $\frac{(A^5)_{ii}-\sum_{k=1}^{n} A_{ik}\cdot(A^3)_{kk}}{2}$ C. $\frac{(A^5)_{ii}-\sum_{k=1}^{n} (A^2)_{kk}\cdot(A^3)_{kk}}{2}$

Solution:

For a general vertex v, a path of length 5 must fall in one of the 3 cases:

- Case 1: contouring a pentagon
- Case 2: stepping into an 'adjacent' triangle, contour it, then step back
- Case 3: stepping into(then immediately step back) an 'reachable' vertex of a triangle which includes v while contouring the triangle

For the given situation, only Case 1 and Case 2 can exist. For case one, one such pentagon will generate 2 paths. For Case 2, consider how many triangles include each of its adjacent vertex. One such 'inclusion' will add 2 paths. And remember that for a vertex v_j, $(A^3)_{jj}$ represents two times of the number of triangles that includes v_j. So, we will have:

$$(A^5)_{ii} = 2 \times \text{number of pentagons} + \sum_{k=1}^{n} A_{ik} \cdot (A^3)_{kk}$$

The final solution:

$$\frac{(A^5)_{ii} - \sum_{k=1}^{n} A_{ik} \cdot (A^3)_{kk}}{2}$$

4. (11 points) Boruvka's Algorithm

Let $G = (V, E)$ be a weighted connected graph. Let $w(u, v) \in \mathbb{R}$ be the weight of the edge $(u, v) \in E$. The goal of this problem is to invent a new algorithm to find one MST of a graph.

(a) Review:

i. (2') The time complexity of finding one MST of a connected graph in general is ____**D**____ for Kruskal's algorithm (optimized by disjoint sets, taking $\alpha(n)$ as a constant) and ____**D**____ for Prim's algorithm (optimized by binary heaps).

A. $\Theta(|V|^2)$ B. $\Theta(|E|)$ C. $\Theta(|V|\log|E|)$ D. $\Theta(|E|\log|E|)$

ii. (0') Recall **Cut property**: Given $G = (V, E)$ and $S \subset V$ is a non-empty proper subset of V. The minimum weighted edge in $\{(u, v)|u \in S, v \in V - S\}$ must be part of at least one MST. Prim's algorithm uses the property.

(b) Inspiration: Consider how Prim's find one MST of a graph. We only find one edge at a time, what if we find multiple edges at a time?

i. (1') If the graph has been divided into m disjoint connected components, design a method to find $\Theta(m)$ edges in one MST (No need to justify your answer in this subproblem.)

> **Solution:** Algorithm: For each connected component, find an edge with a minimum weight that connects with other connected components.

ii. (2') Prove the correctness of the algorithm. You may use the Cut property to show that the selected edges belong to at least one Minimum Spanning Tree (MST). Additionally, denote x as the number of edges selected by the algorithm. Verify that x satisfies the inequality: $am \leq x \leq bm$ where a and b are constants, and m is the number of connected components in the graph so that $x = \Theta(m)$.

> **Solution:** For a connected component, denote its vertex set as C and the selected edge as $e(C)$. Then using the cutting property for C and $V \backslash C$, $e(C)$ is a cutting edge with the minimum weight due to the select method. So $e(C)$ is in one MST. Note that an edge is at most involved in 2 components, so at least $\frac{m}{2}$ edges are selected i.e. $\frac{1}{2}m \leq x < m$.

(c) Details: After we find edges in MST, we should merge the connected components.

i. (1') Which data structure is best for efficiently merging the connected components?

i. ____**B**____

A. Binary Heap **B. Disjoint sets** C. Binary Search Tree D. Hash Table

ii. (1') What is the time complexity of all m connected components into one component?

ii. ____**C**____

A. $\Theta(|V|)$ B. $\Theta(|E|)$ C. $\Theta(m)$ D. $\Theta(m\log|V|)$

(d) Integration: From all those above, we can finally design a new algorithm.

i. (3') If we use the trivial method to find edges each time i.e. determine the edges by traversing all of them. Describe the whole algorithm and analyze the time complexity of them (Make sure your upper bound is tight otherwise you may lose your points).

Solution: Algorithm:

1. Initialize the connected component set $S = \{\{v\}|v \in V\}$ and denote $e(C)/e(v)$ as the selected edge for connected component $C/$ the connected component containing v.
2. Traversing all edges, for one edge $e = (u, v)$, use disjoint set to find whether u and v are in the same connected component. If so, skip this edge. Otherwise, respectively compare the weight $w(u, v)$ to the minimum outside edge recorded for the component containing u and one containing v i.e. $e(u)$ and $e(v)$ and if $w(u, v)$ is less then substitute it(them).
3. After traversing all edges, merge the connected components with selected edges: Denote $E' = \{e(C)|\forall C \in S\}$, then the new connected component are: u and v are in the same connected components iff $\exists\{C_1, \cdots, C_n\}$, where C_i distinct, satisfies: $u \in C_1, v \in C_i, \forall 1 \leq i < n, \exists e_i = (u_i, v_i) \in E, u_i \in C_i, v_i \in C_{i+1}$ i.e. can find a path in E' that passes edges in only E' or inside the components. Denote the new connected component set as S', and add edges in E' into the MST.
4. If $|S'| = 1$, then end the algorithm. Otherwise, use S' as the connected component set and return to step 2. Note that $|S|$ decreases in each iteration, so the algorithm will end in finite steps.

ii. (1') Determine the time complexity of your algorithm (with analysis).

Solution: In each iteration, we traverse each edge once, and for each edge, we compare it twice, which takes $\Theta(1)$ time. So the iteration costs $\Theta(|E|)$ time each time. Then determine the time of iterations. In (b), we show that we will pick at least $\frac{m}{2}$ edges, where m is the number of connected components. So at least, we decrease $|S|$ for half. Therefore, the number of iterations is $O(\log |V|)$, since there are $|V|$ components at first. And it will happen in a graph whose MST is a full binary tree. To sum up, the time complexity of $O(|E| \log |V|)$.

5. (10 points) Minimal dot product of two arrays

Given two arrays a and b of length n, we define the dot product of the two arrays as $\sum_{i=1}^{n} a_i b_i$. Now you are allowed to reorder the elements of both arrays and your task is to minimize the dot product of two arrays after reordering. Mathematically, a reorder of an array is defined by a permutation i.e. a bijective mapping $p : [n] \to [n]$ and the dot product after reordering is $\sum_{i=1}^{n} a_i b_{p_i}$.

For example, if $a = \langle 1, 2, 3 \rangle, b = \langle 4, -2, -1 \rangle$, then the minimal dot product is

$$1 \times 4 + 2 \times (-1) + 3 \times (-2) = -4$$

Please design a **greedy algorithm** as efficient as you can that returns the minimal dot product.

(a) (3') Describe your algorithm in **natural language** or **pseudocode**.

Solution:

Natural language:

(1') Sort a in ascending order.

(1') Sort b in descending order.

(1') Compute $\sum_{i=1}^{n} a_i b_i$ and return.

Pseudocode:

```
1: function MINIMAL-DOT-PRODUCT(a, b)
2:     Sort a in ascending order
3:     Sort b in descending order
4:     return Σ_{i=1}^{n} a_i b_i
5: end function
```

(b) (1') The time complexity of this algorithm is $O(\underline{\quad n \log n \quad})$ (give the most precise upper bound).

(c) (6') Prove the correctness of your greedy algorithm by **Exchange Arguments**.

i. (1pt) **Definition of a solution**

ii. (1pt) **Definition of optimality**

iii. (2pt) **How to change any optimal solution X^* to greedy solution X iteratively**

iv. (2pt) **Why such change doesn't make X^* worse**

Solution:

Definition of a solution

Assume a is sorted in ascending order and b in descending order. Then a solution X can be defined as a permutation of $[1, n]$, where we match a_i and b_{X_i} for every $i \in [1, n]$.

Definition of optimality

$$X^* \in \operatorname*{argmax}_X \left\{ \sum_{i=1}^{n} a_i b_{X_i} \right\}$$

How to change any optimal solution X^* to greedy solution X iteratively

The greedy solution X is $(1, 2, 3, \cdots, n)$, i.e. $X_i = i$ for every $i \in [1, n]$.

We can iteratively check i from 1 to n that, if $X_i^* \neq i$, suppose $X_k^* = i$, then swap X_i^* and X_k^* to make $X_i^* = i$.

Why such change doesn't make X^* worse

The difference of $\sum_{i=1}^{n} a_i b_{X_i}$ before and after such change is $(a_i b_i + a_k b_t) - (a_i b_t + a_k b_i)$ (suppose $X_i^* = t$ before such change).

Because a, b are already sorted, we have $\Delta a = a_k - a_i \geq 0, \Delta b = b_i - b_t \geq 0$.

$$\begin{aligned}
&(a_i b_i + a_k b_t) - (a_i b_t + a_k b_i) \\
=&a_i(b_t + \Delta b) + (a_i + \Delta a)b_t - a_i b_t - (a_i + \Delta a)(b_t + \Delta b) \\
=&-\Delta a \Delta b \leq 0
\end{aligned}$$

Which means such change doesn't make the sum of product larger.

6. (9 points) Conclusion about the variation of shortest path problems

We have a directed weighted graph $G = (V, E)$ that $|V| = n$ and $|E| = \Theta(n)$ (it is a sparse graph).

For each problem, select the **most precise option** that describes the best algorithm to solve this problem.

A. Can be solved in $O(n)$ time.

B. Can be solved in $O(n \log n)$ time.

C. Can be solved in $O(n^2)$ time.

D. Can be solved in $O(n^2 \log n)$ time.

E. Can be solved in $O(n^3)$ time.

F. Can be solved, but no polynomial solution so far.

G. Can not be solved, because it may not exist.

(a) (1') Find the shortest path between two given vertices. Edge weights are nonnegative.

◯ A √ **B** ◯ C ◯ D ◯ E ◯ F ◯ G

Solution: Dijkstra.

(b) (1') Find the shortest path between two given vertices. Edge weights could be negative, but there are no negative cycles.

◯ A ◯ B √ **C** ◯ D ◯ E ◯ F ◯ G

Solution: Bellman-Ford.

(c) (1') Find the shortest path between two given vertices. Edge weights could be negative, and there could exist negative cycles.

◯ A ◯ B ◯ C ◯ D ◯ E ◯ F √ **G**

Solution: Unsolvable because the path can repeat through the negative cycle and become infinitely short.

(d) (1') Find the shortest path between two given vertices. Edge weights are nonnegative and **identical**.

√ **A** ◯ B ◯ C ◯ D ◯ E ◯ F ◯ G

Solution: BFS.

(e) (1') Find the shortest path between two given vertices. The graph is a **DAG**.

√ **A** ◯ B ◯ C ◯ D ◯ E ◯ F ◯ G

Solution: Topological sort (similar to Critical Path).

(f) (1') Find the shortest path between **every pair** of vertices. Edge weights are nonnegative.

◯ A ◯ B ◯ C √ **D** ◯ E ◯ F ◯ G

Solution: Run Dijkstra n times starting from every vertex once.
Or, Johnson algorithm (not introduced in CS101).

(g) (1') Find the shortest **simple path** between two given vertices. Edge weights are nonnegative.

◯ A √ **B** ◯ C ◯ D ◯ E ◯ F ◯ G

Solution: Dijkstra.

(h) (1') Find the shortest **simple path** between two given vertices. Edge weights could be negative, but there are no negative cycles.

◯ A ◯ B √ **C** ◯ D ◯ E ◯ F ◯ G

Solution: Bellman-Ford.

(i) (1') Find the shortest **simple path** between two given vertices. Edge weights could be negative, and there could exist negative cycles.

◯ A ◯ B ◯ C ◯ D ◯ E √ **F** ◯ G

Solution: A simple path cannot be infinitely short, so it must exist. However, it is even more difficult than NP-Complete because there is a simple reduction from Hamiltonian-Path: edge weights are -1 and find if the length of the shortest simple path is $-(n-1)$.

7. (10 points) Gokemon Po

Bob is playing "Gokemon Po", an augmented reality game where he needs to catch n monsters located at specific places in his town. The monsters must be caught in a specific sequence: Bob can only catch monster m_i after catching all previous monsters m_j where $j < i$.

To catch a monster m_i, Bob has two options:

- Buy the monster in the game for c_i dollars.
- Catch it for free at its location.

If Bob is not already at the monster's location, he must pay a ride-share service to transport him there. The minimum cost to travel from the location of monster m_i to m_j is given by $s(i, j)$.

The task is to develop an $O(n^2)$-time algorithm to find the minimum total cost Bob needs to spend to catch all n monsters, starting from the location of the first monster, m_1.

(a) (2') How will you define the sub-problems?

Solution: Define $OPT(i, j)$ as the minimum cost to catch monsters from m_i to m_n, starting at the location of monster m_j (where $j \leq i$).

(b) (5') Describe and justify the Bellman Equation for computing the solution to sub-problems.

Solution: If Bob is already at the location of monster m_i, she can catch it for free. Otherwise, she has two options:

- Purchase the monster m_i for c_i dollars.
- Use a ride-sharing service to travel from m_j to m_i, then catch it for free.

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = n+1 \text{(base case)} \\ OPT(i+1,j) & \text{if } j = i \\ \min\{c_i + OPT(i+1,j), s(j,i) + OPT(i,i)\} & \text{otherwise} \end{cases}$$

Here we define $OPT(n+1, j) = 0$ for any j, since there are no monsters left to catch beyond m_n. The subproblems $OPT(i, j)$ depend only on subproblems with strictly larger indices $i + j$, ensuring an acyclic dependency structure.

(c) (2') What is the solution (minimum total cost) in terms of your sub-problems?

Solution: The solution to the original problem is given by $OPT(1, 1)$, which represents the minimum cost to catch all monsters starting from the location of m_1.

(d) (1') What is the runtime complexity of your algorithm? (answer in $\Theta(\cdot)$ and in the most simplified form)

Solution: As the number of sub-problems goes on $O(n^2)$, and each sub-problem requires $O(n)$ time to compute, the final runtime complexity should be $\Theta(n^2)$

Solution:

Method 2:

Define $OPT(i)$ as the minimum cost to catch monsters from m_1 to m_i if we catch m_i for free.

$$OPT(i) = \min_{j\in[1,i-1]} \left\{ OPT(j) + s(j,i) + \sum_{k=j+1}^{i-1} c_k \right\}$$

The base case is $OPT(1) = 0$.

The solution (minimum total cost) is

$$\min_{j\in[1,n]} \left\{ OPT(j) + \sum_{k=j+1}^{n} c_k \right\}$$

The complexity is originally $\Theta(n^3)$.

However, we can optimize it to $\Theta(n^2)$ by precomputing the cumulative sum of c in $\Theta(n)$ time:

Define the cumulative sum as $C_i = \sum_{j=1}^{i} c_i$, then it can be computed by $C_i = C_{i-1} + c_i$.

Then we can compute $\sum_{k=j+1}^{i-1} c_k$ by $C_{i-1} - C_j$ in $\Theta(1)$ time, which reduces the total time from $\Theta(n^3)$ to $\Theta(n^2)$.

Method 3: (similar to Method 2)

$$OPT(i) = \min_{j\in[i+1,n+1]} \left\{ OPT(j) + s(i,j) + \sum_{k=i+1}^{j-1} c_k \right\}$$

The base case is $OPT(n+1) = 0$.

The solution (minimum total cost) is $OPT(1)$.

8. (10 points) A world with SAT

The goal of this problem is to decide which type of k-SAT problem is in P or NP-Complete.

Recall a **SAT formula** is a logical formula in conjunctive normal form (CNF). Specifically, a **k-SAT formula** ϕ is a conjunction (AND) of clauses $C_1, C_2, \ldots, C_m$, and each clause C_i is a disjunction (OR) of k literals (variables or their negations) i.e. from $X \cup \{\neg x_1, \neg x_2, \ldots, \neg x_n\}$.

k-SAT: Given a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$ and a **k-SAT formula** ϕ on X, determine whether there exists at least one truth assignment τ that makes the formula ϕ evaluate to true. The yes-instance of k-SAT is:

$$\text{k-SAT} = \left\{ \langle \phi \rangle \middle| \begin{array}{l} \phi \text{ is a } k\text{-SAT formula with variables } X = x_1, x_2, \ldots, x_n \\ \text{and clauses } C_1, C_2, \ldots, C_m \text{ such that there exists a truth} \\ \text{assignment } \tau : X \to \{\text{true}, \text{false}\} \text{ such that } \tau(\phi) = \text{true}. \end{array} \right\}$$

Notation: The clause of the original question is denoted as C_i, and the jth literal in the ith clause can be denoted as $l_{i,j}$. Solve the problems below:

(a) (2') Prove $\forall k \geq 2$, k-SAT is in NP by giving out **polynomial-size certificate** and **polynomial-time certifier**.

Solution:

- Certificate: An assignment $\tau : X \to \{true, false\}$.
- Certifier: Assign all the variables with values given by τ and check all m clauses, which takes $O(n + km)$time.

(b) (2') If (k+1)-SAT $\in$ NP-Complete($k \geq 3$), show that k-SAT $\in$ NP-Complete by giving your polynomial-time many-to-one reduction f.
Hint: Consider partitioning the (k+1)-SAT clauses into parts and introduce new variables to connect them.

Solution: Given a (k+1)-SAT formula, for every clause $C_i = l_{i,1} \vee \ldots \vee l_{i,k+1}$, we transform it into two clauses with a new variable x_i since the following boolean formula holds:

$$(l_{i,1} \vee \ldots \vee l_{i,k+1}) \Leftrightarrow (l_{i,1} \vee \ldots \vee l_{i,k-1} \vee x_i) \wedge (l_{i,3} \vee \ldots \vee l_{i,k+1} \vee \neg x_i)$$

Explanation: Always assign $x_i = \neg(l_{i,1} \vee l_{i,2})$.
The reduction is for any k-SAT yes-instance $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, it can transformed into a (k+1)-SAT yes-instance $D_1 \wedge E_1 \wedge \ldots \wedge D_n \wedge E_n$ with n new variables x_i, where $D_i = (C_i \vee x) = (l_{i,1} \vee \ldots \vee l_{i,k-1} \vee x_i), E_i = (C_i \vee \neg x) = (l_{i,3} \vee \ldots \vee l_{i,k+1} \vee \neg x_i)$.

(c) (2') Prove that x is a *yes*-instance of (k+1)-SAT $\Leftrightarrow$ $f(x)$ is a *yes*-instance of k-SAT.

Solution:

$\Rightarrow$: For each clause C_i: $l_{i,1} \vee \ldots \vee l_{i,k+1}$, x is a yes-instance of (k+1)-SAT and $l_{i,1} \vee \ldots \vee l_{i,k+1} \Rightarrow (l_{i,1} \vee l_{i,2}) \vee (l_{i,3} \vee \ldots \vee l_{i,k+1})$, assign $x_i = \neg(l_{i,1} \vee l_{i,2})$. $D_i =$

$l_{i,1} \vee \ldots \vee l_{i,k-1} \vee x_i = (l_{i,1} \vee l_{i,2}) \vee \neg(l_{i,1} \vee l_{i,2}) \vee l_{i,3} \vee \ldots \vee l_{i,k-1}$ =true, $E_i = l_{i,3} \vee \ldots \vee l_{i,k+1} \vee (l_{i,1} \vee l_{i,2}) = C_i$, which indicates $f(x)$ is a yes-instance of k-SAT.

$\Leftarrow$: For each yes-instance of k-SAT $f(x)$, consider each pair of clause D_i and E_i, since $D_i \wedge E_i = (l_{i,1} \vee \ldots \vee l_{i,k-1} \vee x_i) \wedge (l_{i,3} \vee \ldots \vee l_{i,k+1} \vee \neg x_i) = l_{i,1} \vee \ldots \vee l_{i,k+1} = C_i$, each clause of C_i is assigned true. So x is a *yes*-instance of (k+1)-SAT.

Hint for (d) and (e): 2-SAT $\in$ P, while 3-SAT $\in$ NP-Complete.

(d) (2') Briefly show whether your reduction will hold when $k = 2$ and why.

Solution: It won't hold when $k = 2$, since $k = 2$ won't have $l_{i,3}$.
If it holds, then 3-SAT $\leq_p$ 2-SAT, it shows P = NP, which is unknown to be proved now.

(e) (2') Here is a polynomial-time many-to-one reduction from k-SAT to (k+1)-SAT .
Given a k-SAT formula, for every k-SAT clause $C_i = l_{i,1} \vee \ldots \vee l_{i,k}$, we transform it into two (k+1)-SAT clauses with a new variable x since the following boolean formula holds:

$$(l_{i,1} \vee \ldots \vee l_{i,k}) \Leftrightarrow (l_{i,1} \vee \ldots \vee l_{i,k} \vee x) \wedge (l_{i,1} \vee \ldots \vee l_{i,k} \vee \neg x)$$

The reduction is for any k-SAT yes-instance $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, it can transformed into a (k+1)-SAT yes-instance $D_1 \wedge E_1 \wedge \ldots \wedge D_n \wedge E_n$ with a new variable x, where $D_i = (C_i \vee x) = (l_{i,1} \vee \ldots \vee l_{i,k} \vee x), E_i = (C_i \vee \neg x) = (l_{i,1} \vee \ldots \vee l_{i,k} \vee \neg x)$.
Briefly explain whether it will hold when $k = 2$ and why.

Solution: It holds when $k = 2$. When $k = 2$, $D_i = l_{i,1} \vee l_{i,2} \vee x$, $E_i = l_{i,1} \vee l_{i,2} \vee \neg x$.
This indicates 2-SAT $\leq_p$ 3-SAT. Since 2-SAT $\in$ P, the conclusion always holds.

THIS PAGE INTENTIONALLY LEFT BLANK.

Label it clearly when you need to continue your solution on the scratch page.

THIS PAGE INTENTIONALLY LEFT BLANK.

THIS PAGE INTENTIONALLY LEFT BLANK.

THIS PAGE INTENTIONALLY LEFT BLANK.

THIS PAGE INTENTIONALLY LEFT BLANK.

THIS PAGE INTENTIONALLY LEFT BLANK.