CS101 Algorithms and Data Structures
Fall 2023
Final Exam

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

Time: Jan 22nd 10:30-12:30

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.

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

1. (20 points) True or False

Each of the following statements is true (T) or false (F). Write your answers in the **answer sheet**.

(a) (2') You need $\Theta(|V|^3)$ space to store the adjacency matrix for graph $G = (V, E)$.

○ True [✓ **False**]

Solution: $\Theta(|V|^2)$

(b) (2') In a DFS traversal of a directed graph, if u is visited before v is visited, then there must exist a path from u to v.

○ True [✓ **False**]

Solution: Obviously.

(c) (2') Both Kruskal's and Prim's algorithms cannot correctly find the MST when there are negative edges in the graph.

○ True [✓ **False**]

Solution: Both Kruskal's and Prim's algorithm could work when there are negative edges.

(d) (2') Given a directed acyclic graph G. If u appears before v in a topological sort, there will not exist a path from v to u in G.

[✓ **True**] ○ False

Solution: Obviously.

(e) (2') In A* graph search algorithm with a consistent heuristic function, if vertex u is marked visited before v, then $d(u)+h(u) \leq d(v)+h(v)$, where $d(u)$ is the distance from the start vertex to u.

[✓ **True**] ○ False

Solution: The correction of A* is based on this. The proof is similar to Dijkstra.

(f) (2') Dijkstra's algorithm can work correctly on any graphs with negative edges.

○ True [✓ **False**]

Solution: Obviously.

(g) (2') The run time complexity of Floyd-Warshall algorithm can be reduced to $O(|V|^2)$ if we only need to compute the shortest path between a given pair of nodes instead of the shortest paths between all pairs of nodes.

○ True [✓ **False**]

Solution: We still need $\Theta(|V|^3)$ time even we only need to compute the shortest path between a given pair of nodes.

(h) (2') We want to count the number of connected components in an undirected graph using disjoint set. We union the two vertices of each edge. Then the number of vertices such that $\texttt{find}(v) = v$ is the answer.

[✓ **True**] ◯ False

Solution: In disjoint sets, v is the root node of a tree/set if and only if $\texttt{find}(v) = v$. And the number of connected components in the graph is identical to the number of trees/sets in disjoint sets.

(i) (2') If problem A polynomial-time reduces to problem B, then $B \in P$ **only if** $A \in P$.

[✓ **True**] ◯ False

Solution:
As mentioned in the review class, $B \in P$ **only if** $A \in P$ means $B \in P \implies A \in P$. By HW12 question 2(b) choice C, this statement is true.

(j) (2') If you can prove that there exists an NP problem that is not an NP-Complete problem, then you've proved $P \neq NP$.

[✓ **True**] ◯ False

Solution:
"There exists an NP problem that is not an NP-Complete problem" means $NP \neq NPC$. Meanwhile, recall that HW12 question 2(d) choice D stated that "$P = NP$ if and only if $NP = \text{NP-Complete}$.", whose contrapositive is "$P \neq NP$ if and only if $NP \neq \text{NP-Complete}$." Combine these two facts together, we may deduce that this statement is true.

2. (12 points) Single Choice

Each question has **exactly one** correct answer. Write your answers in the **answer sheet**.

(a) (3') Consider a simplest disjoint set of size 6 - with **no** path compression and **no** union by rank. The disjoint set has 2 operations:

- `find(int i)`: Find the root element of the tree that contains `i`.
- `union(int i, int j)`: Find the root elements of `i` and `j`, update `i`'s root to be `j`'s root.

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

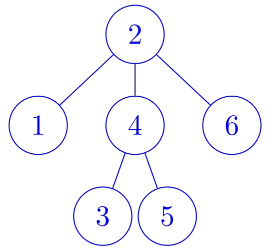
A. 2

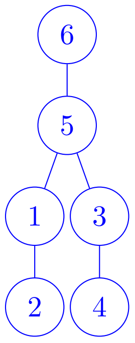
B. 3

C. 4

D. 5

Solution: You can get points if you choose A or B.
"Update `i`'s root to be `j`'s root" is ambiguous.
If `parent[find(i)]=find(j)` then choose A.
If `parent[find(j)]=find(i)` then choose B.

A. 

B. 

(b) (3') If we use Dijkstra's Algorithm with binary heap optimization to find the shortest path from s to t on a positive-weighted graph, what is the **earliest** time that we ensure `dist[t]` is the real length of the shortest path?

A. When t is pushed into the heap for the first time.

B. When the heap is empty.

C. When we modify `dist[t]` for the first time.

D. When t is popped from the heap for the first time.

Solution: The order that these choices happen is $A \approx C < D < B$. Before D, `dist[t]` may be updated to be smaller from another vertex. After D, it is fixed and ensured to be the shortest path.

(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 $(u \rightarrow v)$ 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') The statement "2-Color is in NP-Complete." is ______?

   A. True regardless of whether P equals to NP or not.

   B. False regardless of whether P equals to NP or not.

   C. True if and only if $\mathsf{P} = \mathsf{NP}$.

   D. True if and only if $\mathsf{P} \neq \mathsf{NP}$.

Solution:
As mentioned in solution of HW12 question 2(d) choice B, 2-Color $\in \mathsf{P}$. The reason is that a graph can be 2-Colored if and only if it is a bipartite graph while we are able to determine whether the graph is a bipartite one by BFS in polynomial time. Then:

1. If $\mathsf{P} = \mathsf{NP}$, then by HW12 question 2(d) choice D, $\mathsf{NP\text{-}Complete} = \mathsf{NP} = \mathsf{P}$. Hence, in this case 2-Color $\in$ NP-Complete.
2. If $\mathsf{P} \neq \mathsf{NP}$, then by HW12 question 2(d) choice C, $\mathsf{P} \cap \mathsf{NP\text{-}Complete} = \emptyset$. Hence, in this case 2-Color $\in$ NP-Complete.

3. (20 points) Multiple Choices

Each question has <u>**one or more**</u> 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 2.5 points if you select a non-empty subset of the correct answers. Write your answers in the **answer sheet**.

(a) (5') Let G be a weighted undirected graph with positive weights where edge e has weight $w_e \in \mathbb{R}^+$ for all $e \in E$. And G' is a copy of G except that edge e has weight $f(w_e)$. Which of the following statements is/are true?

A. If $f(w_e) = w_e + \frac{4}{w_e}$, then any MST in G is also an MST in G'.

B. If $f(w_e) = \exp(w_e + w_e^3)$, then any MST in G is also an MST in G'.

C. If $f(w_e) = |w_e \log(w_e)|$, then any MST in G is also an MST in G'.

D. If $f(w_e) = 1 - \frac{1}{1+w_e^8}$, then any MST in G is also an MST in G'.

Solution: $f(\cdot)$ should satisfy for any $\text{Rank}_x < \text{Rank}_y$, $\text{Rank}_{f(x)} \leq \text{Rank}_{f(y)}$ always holds. An easier version is for any $0 < x < y$, $0 \leq f(x) \leq f(y)$ always holds. B and D satisfy this property. Other counterexamples:

A. $|V| = 3$, $w_{12} = 0.1$, $w_{23} = w_{31} = 1$. Obviously, w_{12} is not in the MST in G'.

C. $|V| = 3$, $w_{12} = 1$, $w_{23} = w_{31} = \frac{1}{e}$. Obviously, $\{w_{23}, w_{31}\}$ is not an MST in G'.

(b) (5') Which of the following statements about shortest path algorithms is/are true?

A. If we modify the outer loop of Bellman-Ford algorithm to execute $|V|$ iterations instead of $|V| - 1$ iterations, the algorithm can still find the shortest path on a directed graph with negative-weight edges but no negative cycles.

B. If we modify the outer loop of Floyd-Warshall algorithm to execute $|V| - 1$ iterations instead of $|V|$ iterations, the algorithm can still find all pairs of shortest paths on a directed graph with negative-weight edges but no negative cycles.

C. We can modify the Bellman-Ford algorithm to find the longest path in a directed graph with positive-weight edges but no positive cycles.

D. We can modify Floyd-Warshall algorithm to detect whether there exists a negative cycle or not in a directed graph.

Solution:

A. $\text{dist}[v]$ will remain the same in the $|V|$-th iteration if the graph has no negative cycles.

B. If only $|V| - 1$ iterations are executed, then every pair of shortest paths that must contain the $|V|$-th vertex is not computed correctly.

C. To find the longest path is to find the shortest path in the graph where all edge weights are negated.

D. There exists a negative cycle if and only if $\exists v, \text{dist}[v][v] < 0$.

(c) (5') Given an undirected graph $G = (V, E)$ whose adjacency matrix is A ($\forall\, u, v \in V$, $A_{uv} = 1$ if $\{u, v\} \in E$ and 0 otherwise). Which of the following statements is/are true?

A. If G is connected, then $|E| \geq |V| - 1$.

B. If G is acyclic, then $|E| \leq |V| - 1$.

C. $\sum_{u,v\in V} A_{uv} = |E|$.

D. $\forall\, u \in V,\ (A^2)_{uu} = \mathtt{degree}(u)$.

Solution:

A. Intuitively, trees are connected graphs with minimum edges since removing any one edge in a tree will make it disconnected. (Page 24 of the lecture 16.2 slides, .pptx file)

Formally, we first state a theorem as follows:

Theorem: For a unconnected and acyclic graph $G' = (V', E')$, if we add a new edge to it, we either decrease the number of connected components by 1 or create a cycle in it.

Hence, consider the process of getting $G = (V, E)$, which is given in the question, by adding edges to the empty graph $G = (V, \emptyset)$ (here the V is the set of vertices of G). Since the empty graph $G = (V, \emptyset)$ has $|V|$ connected components while a connected graph has only 1 connected components, we may deduce that if G is connected, we must add at least $|V| - 1$ vertices during the process mentioned above. Hence, if G is connected, it must contains at least $|V| - 1$ edges.

B. Intuitively, trees are acyclic graphs with maximum edges since adding any one more edge to a tree must create a cycle. (Page 24 of the lecture 16.2 slides, .pptx file)

Formally, we first state the theorem mentioned above again as follows:

Theorem: For a unconnected and acyclic graph $G' = (V', E')$, if we add a new edge to it, we either decrease the number of connected components by 1 or create a cycle in it.

Hence, again consider the process of getting $G = (V, E)$, which is given in the question, by adding edges to the empty graph $G = (V, \emptyset)$ (here the V is the set of vertices of G). Since the empty graph $G = (V, \emptyset)$ has $|V|$ connected components while a connected graph has only 1 connected components, we may deduce that during the process of adding edges, we can only decrease the number of connected components by at most $|V| - 1$ times. Hence, we may deduce that if $|E| \geq |V|$, then G must be cyclic. Thus, consider the contrapositive of this statement, we may conclude that if G is acyclic, then $|E| \leq |V| - 1$.

C. $\sum_{u,v\in V} A_{uv} = \sum_{u\in V} \mathtt{deg}(u) = 2\,|E|$.

D. $\forall\, u \in V,\ (A^2)_{uu} = \sum_{v\in V} A_{uv} \cdot A_{vu} = \sum_{v\in V} \mathbb{1}^2\{\{u,v\} \in E\} = \sum_{v\in V} \mathbb{1}\{\{u,v\} \in E\} = \mathtt{degree}(u)$.

(d) (5') Which of the following statements is/are true?

A. 3-SAT can be reduced to 3-Color in polynomial time, and vice versa.

B. A problem A is NP-Complete if every problem in NP can be reduced to A in polynomial time.

C. If you prove 3-SAT can be reduced to 2-SAT in polynomial time, then you've proved $P = NP$.

D. If you prove that any two problems in NP can be reduced to each other in polynomial time, then you've proved $P = NP$.

Solution:

A. First, both 3-SAT and 3-Color are in NP-Complete. Second, as mentioned in solution of HW12 question 2(d) choice B, any two problems in NP-Complete reduces to each other in poly-time. Combine these two facts together, we may deduce that this statement is true.

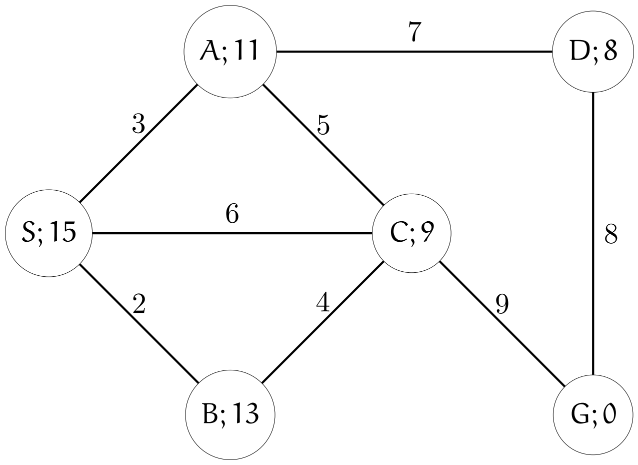
B. As mentioned in solution of HW12 question 2(b) choice B, if a problem A is to be in NP-Complete, then A must first be in NP. However, in this case, since A is not necessarily in NP, you can't conclude that $A \in$ NP-Complete.

C. First, as mentioned in solution of HW12 question 2(c) choice C, 2-SAT $\in$ P (lecture slides page 5). Second, as mentioned in solution of HW12 question 2(d) choice B, proving that 3-SAT can be reduced to 2-SAT indicates that you can solve 3-SAT within polynomial number of standard computational steps as well as calling the subroutine that solves 2-SAT, which is also of polynomial time because 2-SAT $\in$ P. In this way, you have found a polynomial time algorithm for 3-SAT, an NP-Complete problem, which implies that P $=$ NP.

D. By definition of NP-Complete, "Any two problems in NP can be reduced to each other in polynomial time" means NP $=$ NP-Complete. Meanwhile, recall that HW12 question 2(d) choice D stated that "P $=$ NP if and only if NP $=$ NP-Complete.", we may deduce that this statement is true.

4. (12 points) Graph theory benchmark

Consider the weighted undirected graph shown in the figure where G is the goal vertex. When choosing which vertex to visit next, always visit in **alphabetical** order if there is a tie. For A* Search algorithm, the heuristic function is written inside the node. For example, $h(S) = 15$.



(a) **Fill in the blanks**

i. (1') The degree of vertex C is **4**.

ii. (1') Is (S, C, A, D, G, C, B) a simple path? Yes/No: **No**.

iii. (2') Is the heuristic function admissible? **Yes**. Is it consistent? **No**.

(b) **MST**

i. (2') If we run Kruskal's algorithm on this graph, what are the edges in the MST? **Write their weights** in the order we added them.

Solution: 2, 3, 4, 7, 8

ii. (2') If we run Prim's algorithm on this graph starting from B, what is the order of vertices we mark visited?

Solution: B, S, A, C, D, G

(c) **Shortest Path**

Note: In this question, we only care about the shortest path from S to G, so we will terminate the algorithm once we find the shortest path.

i. (2') Suppose we run Dijkstra's algorithm on this graph starting from S. Write down the order of vertices we marked visited.

Solution: S, B, A, C, D, G

ii. (2') If we run A* graph search algorithm on this graph from S, write down the order of vertices we marked visited.

Solution: S, A, B, C, G

5. (8 points) Another house coloring problem

You need to paint n houses in a row with m colors.

You will have a cost $c_{i,j}$ if you paint the i-th house with the j-th color.

You will have an additional cost b_j for each two consecutive houses painted with the same j-th color.

For example, if you paint $n = 6$ houses with colors 122233, your total cost is

$$c_{1,1} + c_{2,2} + c_{3,2} + c_{4,2} + c_{5,3} + c_{6,3} + 2b_2 + b_3$$

Please design a **dynamic programming** algorithm that returns the minimum total cost.

(a) (2') How will you define the subproblems?

Solution: $\mathsf{OPT}(i, j)$ = the minimum total cost after painting the first i houses and the i-th house is painted with the j-th color.

(b) (1') What is the answer to this question in terms of your subproblems?

Solution: $\min\limits_{j\in[1,m]} \{\mathsf{OPT}(n, j)\}$.

(c) (4') Give your Bellman equation to solve the subproblems.

Solution:

$$\mathsf{OPT}(i,j) = \begin{cases} 0 & \text{if } i = 0, j \in [1,m] \\ c_{i,j} + \min\{\mathsf{OPT}(i-1,j) + b_j, \min\limits_{k\neq j}\{\mathsf{OPT}(i-1,k)\}\} & \text{otherwise} \end{cases}$$

Explanation: (NOT Required)

- The base case is $\mathsf{OPT}(0, j) = 0$ for all j.
- For the subproblem $\mathsf{OPT}(i, j)$, if we paint the i-th house with the same color as the $(i-1)$-th house, we will have an additional cost b_j, so the minimal cost besides $c_{i,j}$ is $\mathsf{OPT}(i-1, j) + b_j$.

  Otherwise we paint with a different color k, we won't have additional cost, so the minimal cost besides $c_{i,j}$ is $\min\limits_{k\neq j}\{\mathsf{OPT}(i-1,k)\}$.

(d) (1') What is the runtime complexity of your algorithm? (answer in $\Theta(\cdot)$)
Prefix/suffix optimization is not required.

Solution: $\Theta(nm^2)$ (You can't get points here if your design a worst-case $\omega(nm^2)$ algorithm, but you can get full points if you optimize it to $\Theta(nm)$.)

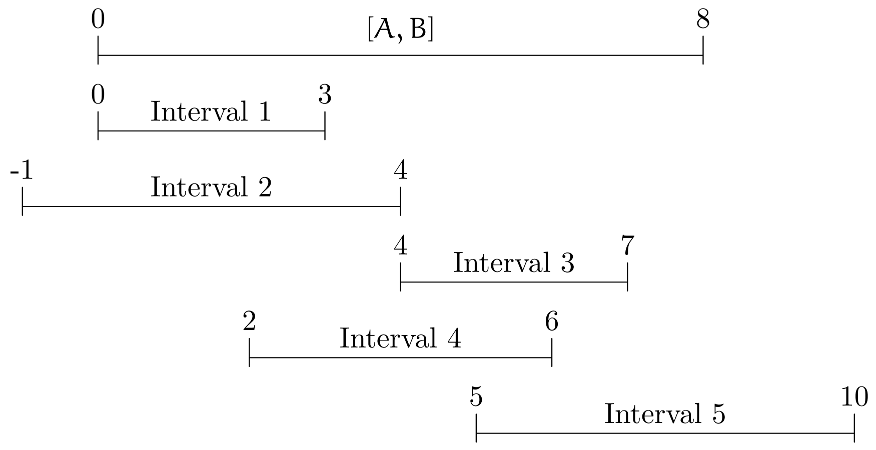
6. (8 points) Interval Covering Problem

There are n intervals. The i-th interval is $[l_i, r_i]$. Your target is to fully cover the range $[A, B]$ with as few intervals as you can.

It is promised that $\forall x \in [A, B], \exists i \in [1, n], l_i \leqslant x \leqslant r_i$, which means that a solution that covers $[A, B]$ always exists.

Example: You need at least 3 intervals to fully cover $[A, B] = [0, 8]$ by these $n = 5$ intervals: $[0, 3], [-1, 4], [4, 7], [2, 6], [5, 10]$, and there are multiple solutions that uses 3 intervals:

- $[0, 3], [2, 6], [5, 10]$
- $[-1, 4], [2, 6], [5, 10]$
- $[-1, 4], [4, 7], [5, 10]$



Please design a **greedy algorithm** to find one solution with the fewest intervals.

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

Solution:
Natural language:
(0.5') Sort the intervals by l_i in ascending order.
Then iterate greedily:

- (0.5') In the 1st iteration, choose the interval S_1 such that $l_{S_1} \leqslant A$ and r_{S_1} is maximized.
- (1') In the 2nd iteration, choose the interval S_2 such that $l_{S_2} \leqslant r_{S_1}$ and r_{S_2} is maximized.
- $\cdots$

(1') Stop iteration when $r_{S_j} \geqslant B$.
Pseudocode:

```
function INTERVAL-COVERING($A, B, l_1, \cdots, l_n, r_1, \cdots, r_n$)
    Sort intervals by left endpoints, i.e. $l_1 \leqslant l_2 \leqslant \cdots \leqslant l_n$
    $S \leftarrow \{\}$
    $h(0) \leftarrow A$
    $i \leftarrow 1$
    for $j = 1$ to $n$ do
        $h(j) \leftarrow A$
        while $i \leqslant n$ and $l_i \leqslant h(j-1)$ do
            if $h(j) < r_i$ then
                $h(j) \leftarrow r_i$
                $S_j \leftarrow i$
            end if
            $i \leftarrow i + 1$
        end while
        if $h(j) \geqslant B$ then
            return $S$
        end if
    end for
end function
```

(b) (1') The time complexity of this algorithm is $\underline{O(n \log n)}$ (in terms of $O(f(n))$).

Solution: You should sort the intervals by l_i in ascending order.
If you answer $O(n^2)$, you can get 0.5 points because it is correct but slower.

(c) (4') Prove the correctness of your greedy algorithm by **exchanging arguments**.

Solution:
(1') Definition of optimality / Key Observation:
The optimal solution of the subproblem $g(j)$ means the sequence of j intervals that can fully cover the range $[A, R]$ and the right endpoint R is maximized.
And we also define some notations in the solution:

- $h(j)$: the max right endpoint R according to $g(j)$ (also in **pseudocode**).
- $S = \langle S_1, S_2, \cdots, S_j \rangle$: the sequence of intervals you select in greedy solution of subproblem $g(j)$ (also in **pseudocode**).
- $T = \langle T_1, T_2, \cdots, T_j \rangle$: any other sequence/any optimal sequence of intervals of subproblem $g(j)$.

Then we are going to prove that greedy solution is always optimal by exchanging arguments, so with minimal j such that $h(j) \geqslant B$, we can prove that the greedy solution correctly find a solution with minimal intervals.
(1') Basic structure of exchanging arguments: if you try to demonstrate some steps to exchange some element, and each step doesn't make it worse.

(1') The way of exchanging:
(0.5) Sort T by the left endpoints, i.e. $l_{T_1} \leqslant l_{T_2} \leqslant \cdots \leqslant l_{T_j}$. **Common mistake: T not sorted! Sort by r may be OK but very difficult to prove then.**
(0.5) Let k be the first index that T is different to S, i.e. $T_1 = S_1, \cdots, T_{k-1} = S_{k-1}, T_k \neq S_k$. Then we change T_k to S_k.
Do iterative changes until T is the same as S.
(1') Proof that such exchanging does not worsen the solution:

- (0.5) If $r_{T_{k-1}} < l_{T_k}$, it means the range $(r_{T_{k-1}}, l_{T_k})$ is not covered before changing because $l_{T_k} \leqslant \cdots \leqslant l_{T_j}$, so the cover range will become larger after changing since $l_{S_k} \leqslant r_{T_{k-1}} < r_{S_k}$. (It contradicts to T's optimality if you define T as any optimal sequence)

  (You are not required to prove $r_{T_{k-1}} < r_{S_k}$ i.e. $r_{S_{k-1}} < \max\limits_{\substack{i\in[1,n] \\ l_i \leq r_{S_{k-1}}}} \{r_i\}$.

  It can be proved by contradiction: suppose $r_{S_{k-1}} = \max\limits_{\substack{i\in[1,n] \\ l_i \leqslant r_{S_{k-1}}}} \{r_i\}$, then

  - If $\forall i \in [1, n], l_i \leqslant r_{S_{k-1}}$, then the range $(r_{S_{k-1}}, B]$ is not covered by any interval, which contradicts to the promise $\forall x \in [A, B], \exists i \in [1, n], l_i \leqslant x \leqslant r_i$.
  - Otherwise $\exists i \in [1, n], l_i > r_{S_{k-1}}$, then the range $(r_{S_{k-1}}, \min\limits_{l_i > r_{S_{k-1}}} \{l_i\})$ is not covered by any interval, which also contradicts to the promise.)

- (0.5) Otherwise $l_{T_k} \leqslant r_{T_{k-1}} = h(k-1)$, we have $r_{T_k} \leqslant r_{S_k}$ since $S_k \in \arg\max\limits_{\substack{i\in[1,n] \\ l_i \leqslant h(k-1)}}\{r_i\}$, so the cover range will at least not become smaller. **Common mistake: presume $r_{T_k} \leqslant r_{S_k}$ always holds!**

- (For the corner case $k - 1 = 0$ above, we can define $r_{T_0} = h(0) = A$).

Notes for other definitions of subproblems:

- If you don't restrict the length of a solution like the proof of interval scheduling on lecture slides or 2022F final, then you need to prove S can't be longer than T by contradiction additionally. It doesn't account for points in this problem because this additional proof can be avoided.

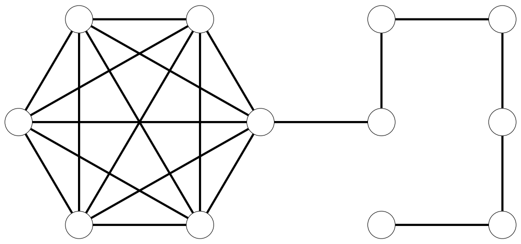
- If you define a solution as a set of intervals instead of a sequence of intervals, the advantage is that sets are unordered, so you can naturally compare S and T in the l_i-ascending order instead of sorting T by l_i first. However, the drawback is that sets doesn't allow duplicate elements, so you can't only change T_k to S_k because the S_k-th interval may also appears in T. In this case you should swap T_k and S_k.

- If you restrict a solution $\langle T_1, \cdots, T_j \rangle$ to be $l_{T_1} \leqslant \cdots \leqslant l_{T_j}$ and $\forall k \in [1, j], l_{T_k} \leqslant r_{T_{k-1}}$, the advantage is that you don't need to analyze the case $l_{T_k} > r_{T_{k-1}}$ compare to the solution above. However, the drawback is that you need to prove feasibility like HW9 minimum refueling problem (d): prove any set of fewest intervals which can cover [A,B] can be reordered to satisfy $\langle T_1, \cdots, T_j \rangle$ to be $l_{T_1} \leqslant \cdots \leqslant l_{T_j}$ and $\forall k \in [1, j], l_{T_k} \leqslant r_{T_{k-1}}$. This can be proved by contradiction: If it doesn't satisfy, then it's not the fewest one.

7. (8 points) Kite is in NP-Complete
In this question, we will prove that Kite is in NP-Complete.

Recall that a **clique** of size k is a graph on k vertices such that any two of them are adjacent (i.e. all the k vertices are connected to each other).

Based on this, we define that a **kite** of size k is a graph on 2k vertices such that k of the vertices form a clique and the remaining k vertices are connected in a "tail" that consists of a path joined to one of the vertices of the clique. Below is a kite of size 6.



Then consider the following problem:

Kite: Given a undirected graph $G = (V, E)$ and a positive integer $k \geq 3$, determine G contains a kite of size at least k. (To avoid corner cases here, we assume $k \geq 3$.)

The yes-instances of Kite is:

$$\text{Kite} = \left\{ \langle G, k \rangle \;\middle|\; \begin{array}{l} k \geq 3 \text{ and } G = (V, E) \text{ is an undirected graph} \\ \text{that contains a kite of size at least } k. \end{array} \right\}$$

(a) (2') Prove that Kite is in NP. (Show your certificate and certifier.)

Solution: Our certificate and certifier for Kite goes as follows:

1. Certificate: A sub-graph G with 2k vertices, whose size is polynomial of the input size.
2. Certifier: Check whether the sub-graph mentioned above is a kite of size k, whose run-time is polynomial of the input size.

(b) (0') We choose Clique to reduce from. Recall that the yes-instance of Clique is:

$$\text{Clique} = \left\{ \langle G', k' \rangle \;\middle|\; \begin{array}{l} k' \geq 3 \text{ and } G' = (V', E') \text{ is an undirected graph that contains} \\ k' \text{ vertices such that they are connected to each other.} \end{array} \right\}$$

(Note that here we also assume $k' \geq 3$ to avoid corner cases.)

(c) (3') Construct your **correct** polynomial-time many-one reduction f that maps instances of Clique to instances of Kite.

Solution:
Given an undirected graph $G' = (V', E')$ and an positive integer $k' \geq 3$, we construct $f(\langle G', k' \rangle) = \langle G, k \rangle$ as follows:

1. $k = k'$.

2. Intuitively, we add a tail of length k **for every** original vertex on G' to get G. Formally, define $m = |V'|$ and label the elements in V' as $V' = \{v_1, v_2, \ldots, v_m\}$. Then for each vertex v_i in G', we add unique k new vertices $v_{i,1}, v_{i,2}, \ldots, v_{i,k}$ that forms a tail leading to v_i (i.e. we add the edges along the path $v_{i,1} - v_{i,2} - \cdots - v_{i,k}$) to get G.

Our newly constructed graph G is of polynomial size, with at most $|V| \cdot (k+1) \leq |V| \cdot (|V|+1)$ vertices (since $k \leq |V|$). Hence, Our reduction takes polynomial time because all computation including adding new vertices and adding new edges can be done in polynomial time.

(d) Prove the correctness of your reduction by showing:

i. (1') x is a yes-instance of Clique $\Rightarrow$ $f(x)$ is a yes-instance of Kite.

Solution:
Let $\langle G', k' \rangle$ be a yes-instance of Clique and let $V^* = \{v_1, \ldots, v_{k'}\}$ be the choice of k' vertices that form the clique. Then the clique formed by these vertices in addition to the tail leading to v_1 (or formally the path $v_1 - v_{1,1} - v_{1,2} - \cdots - v_{1,k}$) form a kite of size $k = k'$ in G. Hence, we claim that $\langle G, k \rangle$ is a yes-instance of Kite.

ii. (2') x is a yes-instance of Clique $\Leftarrow$ $f(x)$ is a yes-instance of Kite.

Solution:
Let $\langle G, k \rangle$ be a yes-instance of Kite and $S \sqcup T$ be the choice of $2k$ vertices that form the kite where S is the set of k vertices that form the clique while T is the set of k vertices that form the tail leading to some vertex $v \in S$. Then we claim that S contains only vertices from the original graph G' because if S contains a newly added vertex $v_{i,j}$, then this new vertex can be connected to only 2 other vertices (since it's on a single path) while these 2 vertices are not connected, which indicates that S cannot be a clique of size at least 3. In this way, the original graph G' contains a clique of size $k' = k$, which indicates that $\langle G', k' \rangle$ is a yes-instance of Clique.

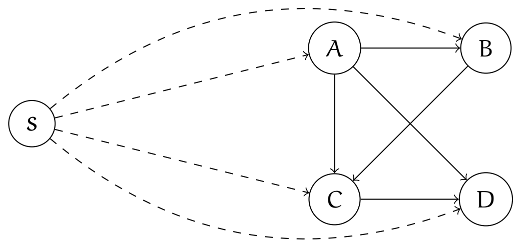
Solution:
Some additional remarks regarding this question:

1. Please keep in mind that proving the correctness of a wrong reduction makes no sense. Hence, once your reduction (question 7(c)) is wrong, you won't get the points regarding proving the correctness of the reduction (question 7(d)).

2. The reduction given by some students only adds one "tail" to the original graph randomly, which turns out to be wrong since it is not guaranteed that this reduction will attach the "tail" to the clique in the original graph accurately when the original graph is a yes-instance of Clique.

8. (12 points) All-Pairs Shortest Paths on Sparse Graphs

Let $G = (V, E)$ be a simple directed graph that does not contain a negative cycle. Let $w(u, v) \in \mathbb{R}$ be the weight of the edge $(u, v) \in E$. We make a new graph $G' = (V', E')$ where $V' = V \cup \{s\}$ for a new vertex $s \notin V$, and $E' = E \cup \{(s, v) \mid v \in V\}$. The construction of the new graph is demostrated as follows.



Let $w'(u, v)$ be the weight of the edge $(u, v) \in E'$ on the new graph, defined as

$$w'(u,v) = \begin{cases} w(u,v), & \text{if } (u,v) \in E, \text{ i.e. } u \neq s, \\ 0, & \text{if } u = s. \end{cases}$$

Let $h(v)$ be the length of the shortest path on G' from s to v, where $v \in V$.

(a) (2') What is the most efficient algorithm to compute $h(v)$ for all $v \in V$? Choose only one choice. **B**.

A. Dijkstra B. Bellman-Ford C. Floyd-Warshall D. Prim

Solution: Computing $h(v)$ is a single-source shortest-path problem, in which the edge weights $w(u, v) \in R$ may be negative, so Bellman-Ford is the best choice.

(b) (2') Prove that $h(u) + w(u, v) \geqslant h(v)$ for every $(u, v) \in E$.

Solution: If $h(u) + w(u, v) < h(v)$ for some $(u, v) \in E$, we can still relax the vertex v by the edge (u, v), which contradicts the fact that $h(v)$ is the length of the shortest path from s to v.

(c) (4') For each edge $(u, v) \in E$ on the original graph G, we define a new weight function as

$$\hat{w}(u,v) = w(u,v) + h(u) - h(v)$$

so that $\hat{w}(u, v) \geqslant 0$. Let $p = \langle v_0, v_1, \cdots, v_k \rangle$ be any simple path from v_0 to v_k, where $v_0, \cdots, v_k \in V$. Prove that if p is a shortest path from v_0 to v_k with weight function $\hat{w}$, then it is also a shortest path from v_0 to v_k with weight function w.

Hint: Let $w(p)$ and $\hat{w}(p)$ be the length of p with weight function w and $\hat{w}$ respectively. Consider $\hat{w}(p) = \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i)$.

Solution: Let $w(p)$ and $\hat{w}(p)$ be the length of p with weight function w and $\hat{w}$ respectively. We start by showing that

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k).$$

We have that

$$\begin{aligned}\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} \left(w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)\right) \\ &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\ &= w(p) + h(v_0) - h(v_k).\end{aligned}$$

Therefore, any path p from v_0 to v_k has $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$. Since the terms $h(v_0)$ and $h(v_k)$ do not depend on the path, if one path from v_0 to v_k is shorter than another using weight function $\hat{w}$, then it is also shorter using w. Note: In fact this is a sufficient and necessary condition, but we only require the students to prove the sufficiency. Moreover, p contains a negative cycle with weight function w **if and only if** it contains one with weight function $\hat{w}$, but we only consider graphs with no negative cycles in this problem.

(d) (4') Now we want to find the lengths of shortest paths between all pairs of vertices in G. If $|E| = \Theta(|V|)$ (which means that the graph is *sparse*), can you come up with an algorithm that solves this problem with time complexity asymptotically better than the Floyd-Warshall algorithm? Briefly describe your algorithm in **natural language** or **pseudocode**, and give its time complexity.

Hint: With weight function $\hat{w}$, the graph does not contain negative weight edges. How can you make use of this property?

Solution:

Algorithm 1 All-Pairs Shortest Paths on Sparse Graphs

```
1: Run Bellman-Ford algorithm on the graph $G'$ starting from $s$ to compute $h(v)$ for every $v \in V$.
2: Let $\hat{w}(u,v) = w(u,v) + h(u) - h(v)$ for every edge $(u,v) \in E$.
3: for each vertex $u \in V$ do
4:     Run Dijkstra's algorithm on $G$ starting from $u$ with weight function $\hat{w}$, to compute $\hat{\delta}(u,v)$ for every $v \in V$, which represents the length of the shortest path from $u$ to $v$ with weight function $\hat{w}$.
5:     For every $v \in V$, $d(u,v) = \hat{\delta}(u,v) + h(v) - h(u)$ is the length of the shortest path from $u$ to $v$ with weight function $w$.
6: end for
```

G' contains $|V|+1$ vertices and $|V|+|E|$ edges, so the Bellman-Ford algorithm needs $\Theta\left(|V|^2 + |V||E|\right)$ time. The Dijkstra's algorithm takes $O\left(|E| + |V|\log|V|\right)$ time if optimized using a Fibonacci-heap. So the entire algorithm has time complexity

$$O\left(|V|^2 + |V||E| + |V|\left(|E| + |V|\log|V|\right)\right) = O\left(|V|^2\log|V| + |V||E|\right).$$

When $|E| = \Theta(|V|)$, the time complexity is $O\left(|V|^2\log|V|\right)$, which is better than the Floyd-Warshall algorithm that takes $O\left(|V|^3\right)$ time.