CMPSC 465 Data Structures & Algorithms
Fall 2025 Ke Chen and Yana Safonova

HW 4 Solution

Due October 06, 10:00 pm

Instructions: You are encouraged to solve the problem sets on your own, or in groups of three to five people, but you must write your solutions strictly by yourself. You must explicitly acknowledge in your write-up all your collaborators, as well as any books, papers, web pages, etc. you got ideas from.

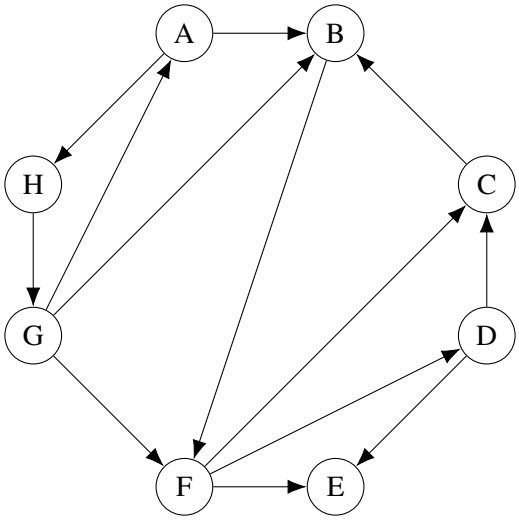
Formatting: Each part of each problem should begin on a new page. Each page should be clearly labeled with the problem number and the problem part. The pages of your homework submissions must be in order. When submitting in Gradescope, make sure that you assign pages to problems from the rubric. You risk receiving no credit for it if you do not adhere to these guidelines.

Late homework will not be accepted. Please, do not ask for extensions since we will provide solutions shortly after the due date. Remember that we will drop your lowest three scores.

This homework is due Monday, October 6, at 10:00 pm electronically. You need to submit it via Gradescope. Please ask on Canvas about any details concerning Gradescope.

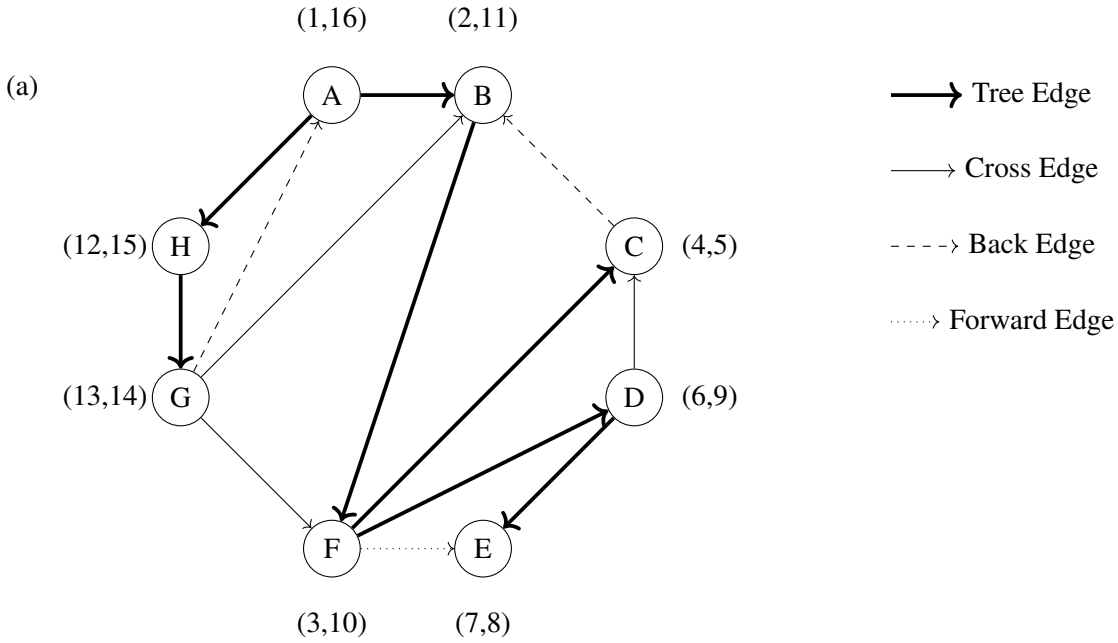
1. (20 pts.) **Depth-First Search.**

(a). Perform depth-first search on the following graph; whenever there's a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge, or cross edge, and give the pre and post number of each vertex.



(b). Either prove or give a counterexample: if $\{u, v\}$ is an edge in an undirected graph, and during depth-first search $post(u) < post(v)$, then v is an ancestor of u in the DFS tree.

Solution:



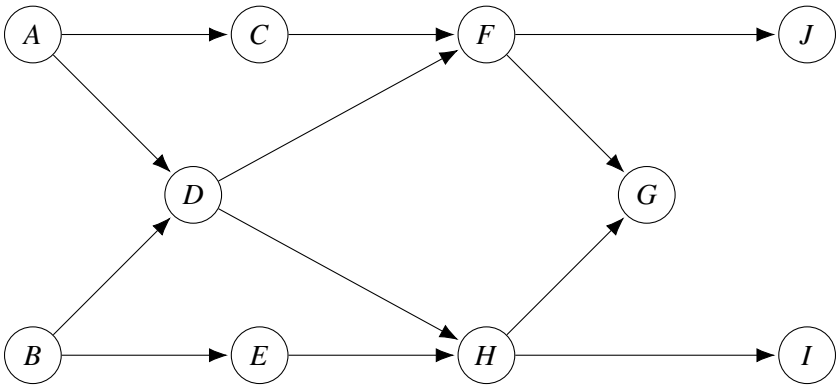
(b) There are two cases possible: $\texttt{pre}(u) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(u)$ or $\texttt{pre}(v) < \texttt{post}(v) < \texttt{pre}(u) < \texttt{post}(u)$. In the first case, u is an ancestor of v. In the second case, v was popped off the stack without looking at u. However, since there is an edge between them and we look at all neighbors of v, this cannot happen. So, the given statement is true.

2. (20 pts.) **Topological Sort** Consider the following directed graph.

(a) What are the sources and sinks of the graph?

(b) Run the topological sort algorithm we learned in class on this graph, use alphabetical order whenever there is a choice

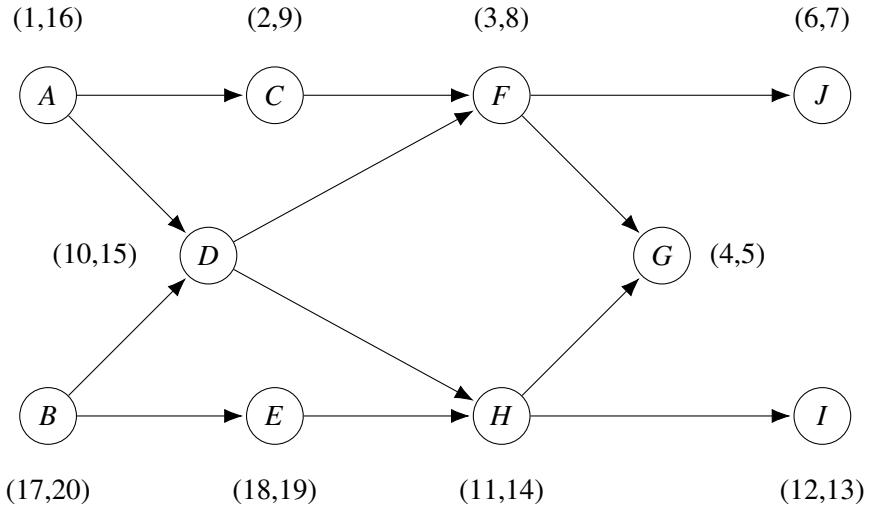
(c) Prove or disprove that the ordering ACBDFJEHIG can be obtained by the algorithm we learned [hint: can you find a valid DFS forest with pre- and post-numbers that yields the given ordering?]



Solution:

(a). Sources of a graph do not have any in-edges. So, A and B are the sources. Sinks do not have any out-edges. So J, G and I are the sinks.

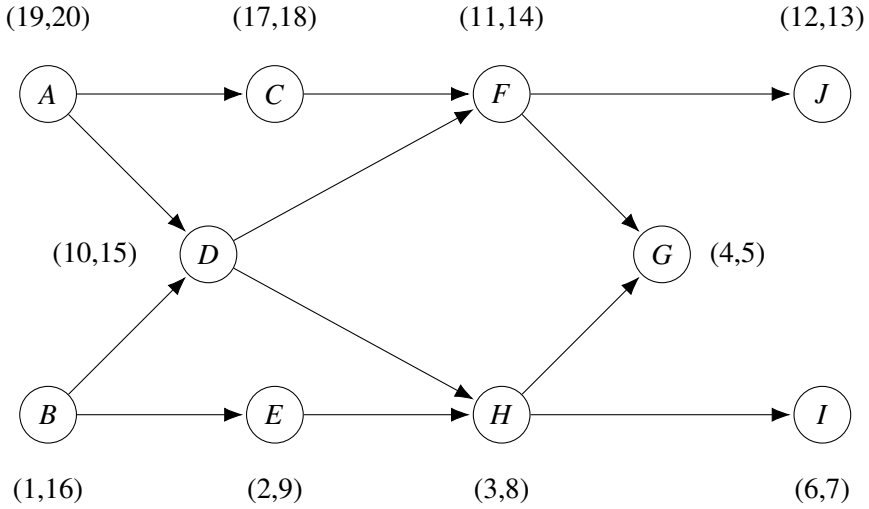
(b). First, we run DFS with timing to obtain the following graph:



Note that we have broken ties based on the alphabetic order. Next, we sort the nodes in descending order of *post time* of DFS to obtain the following topological order:

$$B, E, A, D, H, I, C, F, J, G$$

(c). We can choose to start DFS from node B initially. After it is done with exploration we choose C and finally A. That results in the following pre and post numbers. If we arrange the nodes in the descending order of the post numbers, we obtain the target topological order. So, this ordering can be obtained by our algorithm.



3. (20 pts.) **Ancient Cities.** You are given a set of ancient cities in a desert, along with the pattern of only roads between them, in the form of an undirected graph $G = (V, E)$. Each stretch of the road $e \in E$ connects two of the cities, and you know its length in miles, ℓ_e. You want to get from city s to city t. There's one problem: your bottle can only hold enough water to cover L miles. There are fountains in each city to refill your bottle, but not between cities. Therefore, you can only take a route if every one of its edges has length $\ell_e \leq L$. (**Note:** Only use the algorithms that have been covered in class so far to solve the below problems.)

(a) Given the limitation on your water bottle's capacity, show how to determine in linear time $O(|V| + |E|)$

whether there is a feasible route from s to t. State your algorithm clearly, prove that it is correct and analyze its running time

(b) You are now planning to buy a new bottle, and you want to know the minimum capacity that is needed to travel from s to t. Give an $O((|V|+|E|)\log|V|)$ algorithm to determine this. State your algorithm clearly, prove that it is correct and analyze its running time. (Hint: Consider all the possible values for the minimum bottle capacity. How many are there? What would be the running time of the algorithm that tries them all? Then, try to improve this algorithm.)

Solution:

(a) **Algorithm.** Define the predicate $P(L)$ as "t is reachable from s using only edges of length $\leq L$." To test $P(L)$ run a DFS from s that only traverses edges e satisfying $\ell_e \leq L$. Concretely, one may implement DFS on G but treat any adjacency edge whose length exceeds L as if it were absent. If the DFS visit reaches t then return `YES` (a feasible route exists); otherwise return `NO`.

This can be implemented in two equivalent ways: (i) first form the induced subgraph

$$G_L = (V, E_L), \qquad E_L = \{e \in E : \ell_e \leq L\},$$

and run a standard DFS on G_L; or (ii) run DFS directly on the adjacency representation of G and ignore edges with $\ell_e > L$ when exploring neighbors. The second method avoids explicitly constructing a new edge set.

Correctness. An $s-t$ path is feasible under capacity L if and only if every edge on the path has length at most L. The subgraph G_L (or the DFS that skips edges longer than L) contains exactly those edges permitted under capacity L. Thus there exists a feasible $s-t$ path if and only if s and t lie in the same connected component of G_L. A DFS from s explores precisely the vertices in s's component, so reaching t during DFS is both necessary and sufficient for feasibility.

Running-time analysis. Assume G is stored in an adjacency-list representation. In method (ii), each vertex is marked visited at most once and every adjacency edge is examined at most once during neighbor iteration. Each examination involves a constant-time comparison $\ell_e \leq L$ and possibly a recursive/stack operation. Hence the total cost is $O(|V|+|E|)$. If one explicitly constructs G_L first, forming E_L requires scanning all edges once ($O(|E|)$) and running DFS on G_L costs $O(|V|+|E_L|) \leq O(|V|+|E|)$, so the total remains $O(|V|+|E|)$.

(b) We want the smallest capacity

$$L^\star = \min_{\text{paths } P \text{ from } s \text{ to } t} \max_{e \in P} \ell_e,$$

i.e. the minimum bottle size that permits some feasible $s-t$ route.

Algorithm. Let S be the multiset of edge lengths $\{\ell_e : e \in E\}$. Form the sorted list of distinct edge lengths

$$a_1 < a_2 < \cdots < a_k,$$

where $k \leq |E|$. (These are the only values that could possibly equal $L^\star$.) Perform a binary search on the index set $\{1, \ldots, k\}$. For a candidate index m (value a_m) construct the subgraph

$$G_{a_m} = (V, E_{a_m}), \qquad E_{a_m} = \{e \in E : \ell_e \leq a_m\}.$$

Use DFS (starting at s) in G_{a_m} to test whether t is reachable from s. If t is reachable, then $L^\star \leq a_m$ and the binary search continues on the lower half; otherwise $L^\star > a_m$ and the search continues on the upper half. After the binary search terminates, output the smallest a_m for which DFS found t reachable (or report that no such a_m exists if s, t are disconnected in G at all).

Correctness. The algorithm relies on the monotone (strictly non-decreasing) property of the reachability predicate with respect to the allowed maximum edge length. Let the predicate

$$P(L) \;=\; t \text{ is reachable from } s \text{ in the subgraph of edges with length} \le L.$$

If $P(L_0)$ is true for some L_0, then for every $L \ge L_0$ we have $P(L)$ true as well, because larger L can only add edges. Thus $P(L)$ is monotone nondecreasing in L. The minimum capacity $L^\star$ is exactly the smallest L for which $P(L)$ is true. Because any optimal capacity must equal one of the distinct edge lengths, we can search over the sorted list $a_1, \ldots, a_k$. Binary search finds the smallest index m with $P(a_m) =$ true by repeatedly testing $P(a_m)$ via DFS; monotonicity guarantees that the binary search decision (go left if reachable, go right if not) is correct.

Running-time analysis. Sorting the edge lengths and deduplicating to obtain $a_1, \ldots, a_k$ takes $O(|E|\log|E|)$ time; since $\log|E| = O(\log|V|)$ this is $O(|E|\log|V|)$. Each DFS reachability test on a subgraph G_{a_m} runs in $O(|V|+|E_{a_m}|) \le O(|V|+|E|)$ time. Binary search performs $O(\log k) = O(\log|E|) = O(\log|V|)$ iterations, so the total time for all DFS tests is $O\big((|V|+|E|)\log|V|\big)$. Combining with the sorting cost yields an overall runtime of

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

Space usage is $O(|V|+|E|)$ to store the graph and the list of distinct edge lengths.

4. (20 pts.) **Cycle with an Edge.** Given an undirected graph $G = (V,E)$ and a specific edge $e = \{u,v\} \in E$, the task is to determine if there exists a cycle in G that includes the edge e. Design an algorithm and analyze its runtime.

Solution:
To check for a cycle containing the edge $e = \{u,v\}$, the algorithm first temporarily removes this edge from the graph. After removing the edge, Depth-First Search (DFS) is initiated from vertex u to determine if a path to vertex v still exists in the modified graph. If such a path is found, it implies that an alternate route between u and v exists, and when combined with the original edge e, forms a cycle. Therefore, the algorithm returns `True`. If no path from u to v can be found after removing the edge, then no cycle containing e exists, and the algorithm returns `False`. The final step is to restore the edge e to the graph to revert it to its original state.

Runtime: Removing an edge takes $O(1)$(adjacency matrix) or $O(E)$(adjacency list). So, the time complexity of this algorithm is $O(V+E)$, where V is the number of vertices and E is the number of edges in the graph. This is dominated by the graph traversal (DFS) which operates on the modified graph. The space complexity is $O(V)$, primarily due to the storage required for the visited vertices during the traversal.

5. (20 pts.) **Prerequisites.** Suppose a CS curriculum consists of n courses, all of them mandatory. The prerequisite graph G has a node for each course, and an edge from course v to course w if and only if v is a prerequisite for w.

(a) Design an algorithm to check if it is possible to complete all courses.

(b) Design an algorithm to output a valid sequence in which to take all courses (if your algorithm from part (a) determines that this is possible).

(c) Suppose all courses are offered each semester. A student can take any number of courses in a semester, as long as no two are prerequisites of each other. Design an algorithm to find the minimum number of semesters needed to complete the curriculum.

Solution:

(a) We essentially check if there is a cycle in the graph. If there is a cycle in the graph, then it is not possible to complete all courses in any order.

(b) We need to find a topological order of the nodes using DFS. Any topological order is a valid answer.

(c) The prerequisite graph $G = (V, E)$ is a directed acyclic graph (DAG). Any directed path in G represents a sequence of courses that must be taken in distinct semesters. Therefore, the minimum number of semesters required is at least the length of the longest directed path in G. Conversely, if each course is placed in a layer equal to the length of the longest path ending at it, then all prerequisite constraints are satisfied and no additional semesters are needed. Thus the length of the longest path is both a lower bound and an achievable upper bound, and hence equals the minimum number of semesters required.

Algorithm.

(i) Compute a topological ordering of G : $v_1, \ldots, v_n$.

(ii) Initialize an array level$[v]$ for all $v \in V$ with level$[v] := 1$. Here level$[v]$ denotes the earliest semester in which course v can be taken.

(iii) Process vertices in topological order. For each vertex u (in the order encountered) and for every outgoing edge $(u \to w)$ update

$$\text{level}[w] := \max\{\text{level}[w],\ \text{level}[u] + 1\}.$$

(iv) After processing all vertices, the minimum number of semesters required is

$$S^\star \ = \ \max_{v \in V} \ \text{level}[v].$$

Correctness. *Feasibility.* By construction, whenever there is an edge $u \to w$ we ensure that level$[w] \geq$ level$[u] + 1$. Thus prerequisites of a course always occur strictly earlier in the schedule, so assigning each course to its level produces a valid semester assignment.

Optimality. Let $P = v_1 \to v_2 \to \cdots \to v_k$ be any directed path in G. From the recurrence we obtain level$[v_k] \geq k$. Hence $\max_v$ level$[v]$ is at least the length of any path, i.e. at least the length of a longest path. Conversely, the algorithm constructs a schedule with exactly $\max_v$ level$[v]$ semesters. Therefore the minimum number of semesters equals the length of the longest directed path in G.

Running time. Topological sorting takes $O(|V| + |E|)$ time, and the dynamic programming updates require one pass over all edges, also $O(|E|)$. Thus the total running time is $O(|V| + |E|)$, with $O(|V| + |E|)$ space for the graph plus $O(|V|)$ for the level array.

Rubric:

Problem 1, ? pts

?

Problem 2, ? pts

?

Problem 3, ? pts

?

Problem 4, ? pts

?

Problem 5, ? pts

?