CSE 222 (ADA) Homework Assignment 3 (Theory)

Deadline : April 21 (Friday) 11.59pm.

The theory assignment has to be done in a team of at most two members, as already selected by you. The solutions are to be typed either as a word document or latex-ed and uploaded as pdf on GC. We shall strictly not accept solutions written in any other form. Remember that both team members need to upload the HW solution on GC. Collaboration across teams or seeking help from any sources other than the lectures, notes and texts mentioned on the homepage will be considered an act of plagiarism

Problem 1. (10 points) (Greedy Algorithms) Suppose you are given n tasks $J = \{j_1, j_2, \cdots j_n\}$. Each task j_ℓ has two parts - a preprocessing phase which takes p_ℓ units and a main phase which takes f_ℓ units of time. There are n machines that can execute the main phases of the jobs *in parallel*. However, the preprocessing phases need to be executed sequentially on a special machine. The completion time of any schedule is the earliest time when *all tasks* have finished execution. Design a greedy algorithm which produces a schedule that minimizes the completion time. Here, you need to give formal proof of correctness. Your proof should not exceed one side of an A4 sheet. Anything more than that will not be considered by the evaluator.

Solution. The algorithm is greedy. Execute the pre-processing phases in non-increasing order of f_ℓ.

Proof of Correctness.. We use an exchange argument. Suppose $\sigma = \{j_1, j_2, \cdots j_n\}$ be the ordering of greedy (with renaming of jobs) while $\sigma^\star = \{j_1^\star, j_2^\star, \cdots j_n^\star\}$ be the ordering of an optimal algorithm of the pre-processing phases of the jobs such that the maximum completion time of σ is strictly greater than the maximum completion time of $\sigma^\star$. Now there exists two jobs j_i and j_{i+1} in σ such that their pre-processing phase is executed in the inverse order in $\sigma^\star$ (same argument as done in lectures). Now we create a third schedule σ' where we just exchange the pre-processing phase executions of j_i and j_{i+1} in $\sigma^\star$ - all other jobs remain in the same position. Define $\rho = \sum_{\ell=1}^{i-1} p_\ell$. In $\sigma^\star$, the completion time of j_i is $\rho + p_{i+1} + p_i + f_i$ while the completion time of j_{i+1} is $\rho + p_{i+1} + f_{i+1}$. The maximum of these two quantities is clearly $\rho + p_{i+1} + p_i + f_i$, since $f_i \geq f_{i+1}$ by definition of σ.

In σ', completion time of j_i is $\rho + p_i + f_i$ while the completion time of j_{i+1} is $\rho + p_i + p_{i+1} + f_{i+1}$. Both of these quantities are at most $\rho + p_{i+1} + p_i + f_i$. Hence, the maximum completion time of the jobs j_i, j_{i+1} does not increase while that of every other job remains unchanged in σ' compared to $\sigma^\star$. Repeating this process finitely many times will end up with the schedule σ without increasing the maximum completion time. This contradicts our premise that the maximum completion time of $\sigma^\star$ is smaller than that of σ.

□

Rubric. +2 for figuring out the correct ordering. pseudocode etc are not really needed for this problem.

+8 for the proof : +2 for saying this can be done using exchange argument. +1 for setting up the premise for the exchange argument , that is two schedules etc. +1 for identifying two inverted jobs. +1 for swapping. +3 for finishing the argument. No score deducted if assumed all p's and f's are distinct.

Problem 2. (10 marks) Solve problem 22 Part(a). You can find it here (this is a clickable link). We want linear time, that is $\mathcal{O}(m+n)$ algorithms. No credit for anything that is slower.

Hint 1: The distance array D has nothing to do with the edges in the graph. *Hint 2* : DP !

Follow the usual template for DP solutions. Any non-DP solution will require a formal proof of correctness which is less than one side of an A4 sheet. An algorithm without proof fetches a zero. Moral of the story : Use DP.

Solution. The goal of this problem is compute the length of a longest path from s to v such that $(s,v) \in E(G)$, and the longest path is of the form $s, v_1, v_2, \ldots, v_k (= v)$ such that $D[v_i] \geq D[v_{i+1}]$.

Approach 1 (sketch): Construct an auxiliary directed acyclic graph $D_G = (V, A)$ as follows. For every neighbor u of s, make (s,u) a directed edge. Then, for every u and v, if and only if $D[v] > D[u]$, direct the edge uv from $utov$. (intuitively, this implies that Judy can teleport from u to v as per the given rule).

Observe that the resulting digraph is acyclic. To see this, assume there is a cycle $v_0, v_1, v_2, \cdots v_k, v_0$. But then $D[v_0] < D[v_1] < \cdots < D[v_k] < D[v_0]$ which is clearly a contradiction.

Subproblems: For every vertex v, we denote $OPT(v)$ to be the longest path from s to v in D_G.

Recurrence

$$OPT(v) = 1 + \max_{u:(u,v)\in A} OPT(u)$$

. Also, $OPT(s) = 0$.

Final Solution. $\max_{u:(s,u)\in A} OPT(u)$. This is because the path has to end at one of the neighbors of s so that Judy can take a last hop back home as given in the problem.

Optimal Substructure Property: (not required for credit) Let P be an optimal path from s to v in D. Clearly, the sequence of vertices in P represents a path in G also. Moreover, by construction of D_G, the path P gives a non-increasing sequence of vertices (except for s) in G (with respect to the values in D-array). Let u be the predecessor of v in P. Then, we claim that the subpath P_u from s to u is an optimal solution for $OPT(u)$.

Justification: Suppose that this is not the case. Then, let some other path P'_u is an optimal path from s to u. Then, consider the path P' obtained by appending v at the end of P'_u. As P'_u has more vertices than P_u and it satisfies the criteria of non-increasing D-values, the path P' also satisfies the same. Clearly, P' has more vertices than P. Hence, P is not an optimal path from s to v.

Algorithm: We are now ready to describe an iterative algorithm.

- Construct D_G as described above.
- Initialize $GM[s] = 0$ and for all other vertices $v \in V(G) \setminus \{s\}$, $GM[v] = -\infty$.
- Run a topological sort on D_G. Clearly s is the source of this topological ordering $(s, u_1, u_2, \ldots, u_{n-1})$.
- For every $1 \leq i \leq n-1$, $GM[u_i] = 1 + \max_{j<i:(u_j,u_i)\in A} GM[u_j]$.

- For every $u \in N_G(s)$, output $\max\{GM[u]\}$.

Running time justification: Observe that in this algorithm, every edge of G is checked only once while computing the values of $GM[u]$ for all $u \in V(G)$. Hence, the running time of the algorithm is $O(m+n)$.

Approach 2 (memoization based): We construct a DAG in the same way as before. In addition, we define the subproblems and the optimal substructure in the same way as before.

Algorithm: Construct a DAG $D_G = (V, A)$ as described above. For all $v \in V(G) \setminus \{s\}$, initialize $GM[v] = -\infty$ and $GM[s] = 0$. For every $v \in V(G) \setminus \{s\}$ in any order, invoke $COMPUTE(v)$ as follows.

1. If $GM[v] \neq -\infty$, then return $GM[v]$.
2. Else,
$$q = 1 + \max_{u:(u,v)\in A} COMPUTE(u)$$
.
3. Assign $GM[v] = q$ and Return $GM[v]$.

Finally output $\max_{u:(s,u)\in A} GM[u]$.

Running time:

We first claim that $GM[v] \neq -\infty$ when $COMPUTE(v)$ is called the second time. Suppose this is not the case for the sake of contradiction. This means that Step(3) was not executed before the second call to $COMPUTE(v)$. Let us trace back the recursive calls to $COMPUTE$ starting with the second recursive call to v. Then by construction of the DAG, we shall have a sequence of vertices $v = v_0, v_1, v_2, \cdots v_k = v$ such that there is an edge in A from v_{i+1} to $v_i, \forall i = 0, 1, \cdots, k-1$. This means we have a directed cycle in the DAG leading to a contradiction.

Hence, for every recursive call to v starting with the second one, only Step (1) is executed. The total number of times this can happen is $|outdegree(v)|$. Hence the total number of times Step(1) is executed is at most $\sum_{v\in A} |outdegree(v)| \leq m$. The total number of times steps(2) and (3) are executed for a fixed vertex v is at most one by the first claim. Hence the runtime is $O(m+n)$.

Rubrics: If the solution is Dynamic Programming based, then the following rubric criteria applies.

- (2 point) 1 point for a precise description of the subproblems you want to solve in the dynamic program. Subproblem definition is same for both the approaches. 1 point for either creating the DAG and topological sorting (approach 1) or proving that the graph on which the DP is being run is (explicitly or implicitly) a DAG *Notice this is just the subproblem, not how to solve it, or how it was obtained.*
- (2 points) A recurrence which relates a subproblem to "smaller" (Whatever you define "smaller" to mean) subproblems. This is also same for both the two approaches.

  Notice this is just the recurrence, not the algorithm or why the recurrence is correct.
- (1 points) Identify the subproblem that solves the original problem. Same for both approach-1 and approach-2.

- (3 points) An algorithmic description (or a pseudocode) based on a DP that solves the recurrence in the desired time (linear in this case). *You do not need to prove correctness of the pseudocode*
- (2 points) An argument for the running time of your dynamic programming and the space requirement.

If the solution is non-DP based, then a formal proof of correctness is needed which should be less than a page length. A non-DP solution without proof will result in 0 marks.

Problem 3. (5 points) You are given a graph $G = (V, E)$ where the weights of the edges can have only three possible values - 2, 5 and 7. Design a linear time , that is $\mathcal{O}(E + V)$ time algorithm to find the minimum spanning tree of G. *Hint:* Modify a known algorithm. No proof of correctness required. But argue runtime for credit.

Solution 1:

We can use Prim's algorithm. Prim's algorithm can be implemented in time $\mathcal{O}(|E|t + |V|)$ where t is the time taken to insert or delete an element in a priority queue of size at most $|E|$.

+1 for realizing that this can be done using Prim's algorithm with a modification in priority queue implementation.

In class, we used a min-heap to implement the priority queue, which gave a time complexity of $\mathcal{O}(|E| \log |V|)$. However, if the element values can have only a small number (three in this case) of possibilities, we can implement the priority queue to support insertions and deletions in $O(1)$ time.

Priority queue implementation: maintain 3 lists of elements - each for one possible value of the elements. For insertion, add the element to the list for the corresponding value. To delete the element with smallest value, return (and delete) an arbitrary element from the list of smallest value which is non-empty.

+2 for implementing the priority queue DS properly using constant number of lists, hashmaps etc.

Time complexity of priority queue: If there are k possible values of the elements, the priority queue insertion and deletion take $O(k)$ time each, since insertion and deletion in a list takes $O(1)$ time.

Running time of Prim's implementation: Hence, the running time of Prim's algorithm for k values of edge weights is $O(k|E| + |V|)$ which is $O(|E| + |V|)$ for $k = 3$ as in the question.

+2 for runtime : +1 for writing the content of each of the above paragraphs

We now state and analyze Prim's algorithm for completeness.

Prim's algorithm: Maintain a set X of vertices reachable from a source s using the tree edges. Also maintain a priority queue Q of edges with the invariant that every edge in the cut $(X, V - X)$ is in Q, with value equal to the edge's weight. Initialize Q with the edges incident to s.

At every step, find the cheapest edge crossing the cut $(X, V - X)$ and add it to the tree (and update X). To implement this, repeatedly find (and delete) the cheapest edge (u, v) in Q. If (u, v) crosses the cut $(X, V - X)$, (say with $u \in X$ and $v \notin X$), add (u, v) to the tree, add v to X and insert into Q the edges incident to v.

Analysis of Prim's algorithm: Prim's algorithm is correct due to the cut-lemma which says that the cheapest edge crossing any cut $(A, V - A)$ is in a minimum spanning tree.

The running time is dominated by the number of insertions and deletions in the priority queue. Each edge (u, v) can be inserted into the priority queue at most twice, when either of u or v enters the set X. The number of deletions cannot exceed the number of insertions. Hence, the number of priority queue operations is at most $4|E|$. Hence, the time complexity is $\mathcal{O}(t|E|)$ where t is the time to insert or delete a single element in the priority queue.

Solution 2. Another solution could be modifying Kruskal's algorithm. Let us recall the algorithm first. Sort the list of edges in non-decreasing order of weights. Maintain the connected components of set of edges F picked so far. Start with n trivial components - each a singleton vertex. At any iteration, the algorithm considers the next edge (u, v) in the sorted list. If u and v belong to different components, add (u, v) to the set F and merge the two components of u and of v. Otherwise, discard the edge (u, v). It can be shown that at the end there can be at most one component in F (provided the given graph is connected) and there are no cycles in F. Further, due to the cycle property of MSTs (or equivalently, the cut property) and the fact that we consider edges in non-decreasing order, F is indeed an MST.

+1 for realizing that you can implement Kruskal faster Now, the runtime of this procedure is $\mathcal{O}(E \log V)$. There are two potential sources of the logarithmic factor - firstly the sorting routine. Secondly, testing whether the endpoints of the candidate edge belong to the same component or not requires Union-Find DS which incurs $\mathcal{O}(\log V)$ time per edge on an average. Now suppose there are t different weights of edges in the graph. We shall modify Kruskal in a way so that the overall runtime is $\mathcal{O}(t(E + V))$

Instead of sorting the edges, we run the algorithm in t phases - one phase considers edges of a particular weight only and we run them in increasing order of weights.

+1 for implementing the algorithm in phases - each phase for a specific weight

Let the edge weights be $w_0 < w_1 < \cdots w_t$. We shall define an auxiliary graph G_p for phase $p = 0, 1, 2, \cdots t$. Let C_p be the set of connected components of the set of edges having weight less than w_p. Then, the vertex set V_p of G_p is just C_p. One may think of C_p as *contracted* to a single vertex in V_p.

The edge set E_p is the set of edges $(u, v) \in E$ with weight w_p which connect two vertices in V_p i.e. u and v lie in different components in C_p.

G_0 is the graph with all vertices in G and edges of weight w_0. Notice that C_{p+1} (and hence V_{p+1}) is the set of connected components of G_p.

Algorithm Our algorithm finds a maximal forest of G_p for each p, and output the union of each of these forests.

To implement this, maintain a set of edges F which is initially empty. At any phase $p = 0, 1, 2 \cdots t$, perform the following operations

1. Find the connected components of G_p using BFS/DFS and add to F an arbitrary spanning tree of each component

   1 point for this.

2. Construct G_{p+1} as follows : V_{p+1} is the set of connected components of G_p. Construct an adjacency list representation of E_{p+1} by adding each edge $(u, v) \in E$ of weight w_{p+1} to the

adjacency lists of the connected components of u and v whenever (u, v) belong to different vertices of V_{p+1}.

1 point for constructing this graph.

Explanation. *Correctness:* The above algorithm is essentially a simulation of Kruskal's algorithm while avoiding cycle detection using Union-Find. Kruskal's algorithm considers the edges in non-decreasing order of edge weights, which is the same as the order of our phases. Our algorithm works because of the following crucial property : after any phase p, the connected components of G_{p+1} correspond to the connected components in Kruskal's algorithm. Hence, the total cost of edges in phase p is the same for our algorithm and for Kruskal's, since all edges considered in the phase have the same weight.

We can prove the above property by induction. It is clearly true before the start of the first phase. For the induction step, fix a phase p. Before phase p begins, the connected components in Kruskal's algorithm are V_p. Then, Kruskal's algorithm picks a maximal set edges of E_p that do not form a cycle (i.e. a maximal forest) in V_p. Hence, in each connected component of G_p, Kruskal's algorithm picks a spanning tree, which is also what our algorithm picks. Therefore, Kruskal's and our algorithm have the same connected components after phase p.

Running time: Each phase takes $O(V + E)$ time to find connected components and to contract the connected components of G_p. Since there are t phases, the total time is $\mathcal{O}(t(V + E))$.

1 point for running time.