CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Midterm Exam #2 Solutions

November 14, 2019

Name: _____ *NetID:* _____

Instructions

Problem 1. Prove or disprove the following assertions.

(a) Suppose $G(V, E)$ is a directed acyclic graph (DAG) with a unique source s and a unique sink t. Then the topological ordering of G will be unique. **(10 points)**

Solution. The assertion is **false**. The following DAG has a unique source and a unique sink but two different topological ordering (s, u, v, t) and (s, v, u, t).

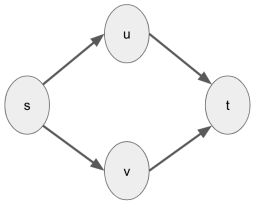

Figure 1: A counter-example to the assertion.

(b) Suppose $G(V, E)$ is an undirected connected graph such that for every cut $(S, V - S)$, there are at least two cut edges in G. Then, if we remove any single edge from G, it will remain connected.

(15 points)

Solution. The assertion is **true**. We proved in the class that a graph G is connected if and only if every cut $(S, V - S)$ has at least one cut edge. We use this to prove the assertion as follows.

For any edge $e \in E$, the graph $G - e$ is such that every cut has at least one cut edge since before removing the edge, there were at least two cut edges, and removing a single edge can only change the number of cut edges by at most one. As such, graph $G - e$ is connected for every choice of $e \in E$, proving the assertion.

Problem 2. You are given a set of n (closed) intervals on a line:

$$[a_1, b_1], [a_2, b_2], ..., [a_n, b_n].$$

Design an $O(n \log n)$ time *greedy* algorithm to select the *minimum* number of points on the line such that any input interval contains at least one of the chosen points. **(25 points)**

Example: If the following 5 intervals are given to you:

$$[2, 5], [3, 9], [2.5, 9.5], [4, 8], [7, 9],$$

then a correct answer is: $\{5, 9\}$ (the first four intervals contain number 5 and the last contains number 9; we also definitely need two points since $[2, 5]$ and $[7, 9]$ are disjoint and no single point can take care of both of them at the same time).

Solution. The solution to this problem is very similar to that of maximum disjoint interval problem studied in the class.

Algorithm:

(i) Sort the intervals based on their *last point* (i.e., b_i for interval $[a_i, b_i]$) in the increasing (non-decreasing) order.

(ii) Pick the last point of the first interval in this ordering (namely, b_i if $[a_i, b_i]$ appears first in the ordering) in the solution. Go over this ordering until you find the next interval that does contain the last chosen point and pick the last point of this interval in the solution. Continue until you iterate over all intervals.

Proof of Correctness: Let $G = \{g_1, \ldots, g_k\}$ denotes the points chosen by the greedy algorithm. The set of points in G clearly *covers* all the intervals, i.e., every interval has at least one point intersecting with G – this is by definition of the greedy algorithm. We now prove that G picks the minimum number of intervals.

Let $O = \{o_1, \ldots, o_\ell\}$ denotes the point in some optimal solution sorted in increasing order. We use an exchange argument. Let j be the first index where O and G differ, namely, $g_1 = o_1, \ldots, g_{j-1} = o_{j-1}$ but $g_j \neq o_j$. We know that $j \leq \ell$ because otherwise, $g_1, \ldots, g_{j-1}$ cover all the intervals and thus G would have not contained g_j by definition of the greedy algorithm. Now consider the solution $O' = \{o_1, \ldots, o_{j-1}, g_j, o_{j+1}, \ldots, o_\ell\}$ obtained by exchanging o_j with g_j in O. We will prove that O' is another optimal solution.

Recall that $o_1, \ldots, o_{j-1}$ is the same as $g_1, \ldots, g_{j-1}$ and by definition of the greedy, we picked g_j to be the smallest last point of an interval, say b_x in the interval $[a_x, b_x]$, which is not covered by $g_1, \ldots, g_{j-1}$. This implies that $o_j \leq g_j$ as otherwise O will not cover the interval $[a_x, b_x]$. As such, if we exchange o_j by g_j, we will still cover any interval that was covered by o_j and thus O' is a valid solution to the problem. Moreover, size of O and O' are the same (as we proved earlier $j \leq \ell$), hence O' is another optimal solution.

We are now done as we can repeat this argument and exchange the entire optimal solution O to the greedy solution G, while maintaining optimality in every step. This implies that G is also optimal.

Runtime Analysis: The first step takes $O(n \log n)$ time (by say using merge sort). The second step also takes $O(n)$ time as we are simply iterating over the intervals. Hence, total runtime is $O(n \log n)$.

Problem 3. You have a list of n cities that you would like to visit. There are also m roads (two-way) between these cities which allow you to travel from one city to another one. Each city $i \in \{1, \ldots, n\}$ has a priority $p_i \in [0, 1]$ and you are originally at a city s with priority $p_s = 1$. Your goal is to travel to *as many cities as possible* by taking the roads between cities subject to the following two constraints:

(1) You should visit the cities in *decreasing* order of their priorities, that is, if you visit a city i at some point and *later* visit city j, it should be the case that $p_i > p_j$;

(2) You are *not* allowed to visit a city more than once.

Design an $O(n + m)$ time algorithm that finds the largest number of cities you can visit. **(25 points)**

Hint: Prove that the goal is to find a certain type of a *path* in a graph in this problem. The easiest way to solve this problem is then to use a reduction to a problem you have already seen.

Solution. The simplest way to solve this problem is by a reduction to longest path over a DAG.

Algorithm:

(i) Create a directed graph G by picking a vertex for every city and adding a directed edge (u, v) if $p_u > p_v$ (here priority of the vertex is taken as the priority of the city this vertex corresponds to).

(ii) Run the longest path (dynamic programming) algorithm over this graph G starting from vertex s and return the name of the cities and their order in this path as the largest number of cities one can visit.

Proof of Correctness: We first prove that G is a DAG (and so we can indeed run the longest path algorithm over it). To do so, it suffices to show a topological ordering of this graph (a directed graph is a DAG if and only if it has a topological ordering). The ordering is simply to write the vertices in decreasing (non-increasing) order of their priorities (breaking the ties arbitrarily). This is a topological ordering since for every edge (u, v) we know $p_u > p_v$ and hence v appears after u in the ordering.

Now note that every path P in G starting from s corresponds to a sequence cities we can visit by taking the roads and following the rules on priorities and vice versa. Hence, the longest path will give us the largest number of cities we could visit.

Runtime Analysis: The first step of creating the graph takes $O(n+m)$ time by traversing the array of roads and cities. The runtime of longest path algorithm on a DAG with n vertices and m edges is also $O(n+m)$, hence making the total runtime $O(n+m)$.

Problem 4. A *feedback edge set* of an undirected connected graph $G(V, E)$ is a set of edges $F \subseteq E$ such that any cycle in G has at least one edge in F. In other words, removing the set of edges in F from G makes the graph G acyclic. Describe and analyze an $O(m \log m)$ time algorithm to compute the minimum-weight feedback edge set of a given $G(V, E)$ with *positive* weight w_e on each edge $e \in E$. **(25 points)**

Hint: Prove that graph $G - F$ should be a tree. What kind of a tree ensures that the total weight of edges *not* in the tree, i.e. in F, is minimized? (You have already seen in your homeworks how to find this tree.)

Solution. The simplest way to solve this problem is by a reduction to the maximum spanning tree problem over a connected undirected graph (which you already solved in homework 4 by another reduction to the minimum spanning tree problem).

Algorithm: Pick a maximum weight spanning tree T of G and return $G - T$ as the answer.

Proof of Correctness: Following the hint, we first prove that for any optimal solution F, $G - F$ should be a tree. Proof by contradiction: suppose not, then either (1) $G - F$ has less than $n - 1$ edges and so is not connected, which means there exists an edge e in F which if added to $G - F$ will not create a cycle, and thus the solution $F - \{e\}$ would have strictly less weight than F, making F not optimal, or (2) $G - F$ has more than $n - 1$ edges and is still connected, which means that there exists a cycle in $G - F$ still (any connected graph with more than $n - 1$ edges has a cycle), and thus F is not a valid solution.

We are now done because minimizing the weight of F is equivalent to maximizing weight of $G - F$ and since $G - F$ should be a tree, this means for the optimal solution F, $G - F$ would be a maximum spanning tree, exactly as found by the algorithm.

Runtime Analysis: We already saw in homework 4 how to find a maximum spanning tree in $O(m \log m)$ time (say using Kruskal's or Prim's algorithms), and we only need to pick the edges not in T which takes another $O(m)$ time. Hence, total runtime is $O(m \log m)$.

Problem 5. [**Extra credit**] You are given a directed unweighted graph $G(V, E)$ with two designated vertices s and t. Some of the edges in G are colored *red*. Design an $O(n + m)$ time algorithm that finds the shortest length path from s to t that uses at most one red edge (you may assume that the shortest length path always involve taking one red edge). **(+10 points)**

Solution. The simplest way to solve this problem is by a reduction to the shortest path problem over an unweighted directed graph. It is worth mentioning that this problem is almost identical to the extra credit problem of practice exam 2.

Algorithm/Reduction: We prove this using reduction to the problem of finding an unweighted shortest path (which we can solve using BFS). Create a new graph $G'(V', E')$ as follows:

- V' contains two disjoint copies of vertices of V, denoted by $V_1 = V$ and $V_2 = V$. We use s_1 to denote the copy of s in V_1 and t_2 to denote the copy of t in V_2.
- We connect every vertex $u \in V_1$ (respectively, $w \in V_2$) to any other vertex $v \in V_1$ (respectively, $z \in V_2$) if and only if there is a normal (not-red) edge (u, v) (respectively, (w, z)) in G. We will then connect every vertex $u \in V_1$ to any vertex $v \in V_2$ if there is a red edge (u, v) in G.

After creating the graph, we simply find the shortest path from s_1 to t_2 in G' (using BFS) and return the name of vertices along the path as the answer to problem.

Proof of Correctness: Any s-t path P in G that uses only one red edge corresponds to a s_1-t_2 path in G' by going in the first copy V_1 until we hit this red edge and then continuing from the endpoint of this red edge in the second copy V_2. Similarly, any s_1-t_2 *shortest* path P' in G' corresponds to a s-t path in G that uses only one red edge by simply mapping the copies of vertices in G' to their original name in G; since P' is a shortest path, we know that two copies of the same vertex are not used in P' as otherwise we could have made it shorter by removing the part between the two copies (using the assumption that shortest paths contain exactly one red edge). Hence this mapping indeed creates a path in G (and not a walk). This proves the correctness of the reduction as it implies the shortest s_1-t_1 map and shortest s-t path with only one red edge are the same (up to the mapping step).

Runtime Analysis: We create a graph with $\Theta(n)$ vertices and $\Theta(m)$ edges in $O(n + m)$ time and run BFS over it which takes another $O(n + m)$ time, leading to $O(n + m)$ time algorithm in total.