CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Midterm Exam #2 Solutions

April 18, 2022

Problem 1.

(a) Suppose $G = (V, E)$ is any undirected graph and $(S, V - S)$ is a cut with zero cut edges in G. Prove that if we pick two arbitrary vertices $u \in S$ and $v \in V - S$, and add a new edge (u, v), in the resulting graph, there is no cycle that contains the edge (u, v). **(12.5 points)**

Solution. Suppose towards a contradiction that there is a cycle C that contains the edge (u, v). This means that there is a path P from u to v that does not use the edge (u, v) in this cycle. Since this path starts at $u \in S$ and ends in $v \in V - S$, there should be an edge e in this path that goes from S to $V - S$. This means that the cut $(S, V - S)$ has at least one cut edge, a contradiction.

(b) Suppose $G = (V, E)$ is an undirected graph with weight w_e on each edge e. Additionally, suppose that G has an edge f with weight *strictly larger* than all other edges in G. Prove that if f belongs to a *cycle* in G, then *no* minimum spanning tree (MST) of G can contain the edge f. **(12.5 points)**

Solution (**Solution 1**)**.** Suppose towards a contradiction that there is some MST T that contains the edge f. Consider the graph $T - f$ which contains two connected components S and $V - S$ (we removed an edge from a tree so we get exactly two connected components). Since f belongs to some cycle in G, there is an edge e that also connects S and $V - S$ (because there is another path from endpoints of f to each other in G so together with f, this forms the cycle). Since f has the largest weight in G, we know that $w_e < w_f$.

Now consider the graph $T' = T - f + e$ which we claim is a tree: it has $n - 1$ edges (because we removed one edge from T which had $n - 1$ edges and added another edge to it) and it is connected (because we connected S and $V - S$ via the edge e again). So T' is a spanning tree with weight strictly less than T (because $w_e < w_f$), contradicting the fact that T was a MST. So no MST of G can contain the edge f.

Problem 2. You have a bag of m cookies and a group of n friends. For each friend $1 \leq i \leq n$, you know the "greed factor" of your friend as a number $G[i]$: this is the minimum number of cookies you should give to this friend to make them stop complaining. Of course, you would like to find a way to distribute your cookies in a way to *minimize* the number of your friends that are still complaining.

Design an $O(n \log n)$ time greedy algorithm to find an assignment of the cookies to your friends that minimizes the number of the friends that are still complaining: recall that a friend i stops complaining if we assign them $c_i \geq G[i]$ cookies. **(25 points)**

Assume that the greed factors of the friends are all distinct. The output should be a list of n pairs consisting of a friend identified by their unique greed factor, and the number of cookies assigned to this friend.

Solution. *Algorithm:*

1. Sort the array A in increasing order using merge sort.
2. Let $t \leftarrow m$. Iterate over the array A and if $A[i] \leq t$, allocate $A[i]$ cookies to this friend, output $A([i], t)$, and update $t \leftarrow t - A[i]$; otherwise, if $A[i] > t$, output $(A[i], t)$ and terminate the algorithm (t is the number of remaining cookies).

Proof of Correctness: A simple observation is that minimizing the number of the friends that are complaining is equivalent to maximizing the number of friends that are "satisfied", i.e., no longer complain. So we focus on the second task instead. We now prove the correctness of the algorithm using an exchange argument (there are multiple ways to do an exchange argument for this problem; we simply pick one of them).

Let $G = \{x_1, x_2, \ldots, x_k\}$ denote the name of the friends that were satisfied by the greedy algorithm, sorted in increasing order of their greed factor, so $g_{x_1} \leq g_{x_2} \leq \ldots \leq g_{x_k}$. Let $O = \{o_1, o_2, \ldots, o_\ell\}$ denote the name of the friends that are satisfied in *some optimal* solution, again sorted in increasing order of their greed factor so $g_{o_1} \leq \ldots \leq g_{o_\ell}$ (note that we cannot assume $k = \ell$; this is in fact precisely what we want to prove!).

We now show how to exchange O to G. Let j be the first index where O and G differ, i.e., $x_1 = o_1, \ldots, x_{j-1} = o_{j-1}$ but $x_j \neq o_j$. Since both G and O are sorted in increasing order of their greed, and by definition of the greedy, we know that $g_{x_j} \leq g_{o_j}$. We can thus take the cookies given to o_j in O and instead give them to x_j and make x_j satisfied. This means that $O' = \{x_1, \ldots, x_j, o_{j+1}, \ldots, o_\ell\}$ is also a valid solution and since size of O' is equal to size of O, it is also optimal.

We can thus continue doing the exchange with index $j + 1$ and so on until we entirely exchanged O to G; in particular, note that once we exchanged x_k and o_k, we can be sure that there are no friends left in the resulting solution since by definition of the greedy algorithm, the reason we did not satisfy some friend x_{k+1} was that we ran out of cookie for satisfying *anyone* else and hence this new solution cannot also satisfy anyone after x_k. This means that $\ell = k$ and G is also optimal.

Runtime analysis: Sorting in the first step takes $O(n \log n)$ time and the next line takes $O(n)$ time as we are just going over the array once and spend $O(1)$ for each item. So, the total runtime is $O(n \log n)$.

Problem 3. You are given a directed graph $G = (V, E)$ with blue and red edges, and two vertices s and t. Design and analyze an algorithm that outputs whether there exists a *walk* from s to t in G that contains an *even* number of red edges; the walk can contain an arbitrary number of blue edges. **(25 points)**

A complete solution consists of three part: the algorithm or reduction (**10 points**), the proof of correctness (**10 points**), and the runtime analysis (**5 points**).

Solution. *Algorithm (reduction):*

1. We construct a new directed graph $G' = (V_1 \cup V_2, E')$ with two copies of the vertices in G ($V_1 = V$ is the first copy and $V_2 = V$ is the second copy). For each *blue* edge $(u, v) \in E$, add this edge to both copies of V in G', i.e., add (u_1, v_1) and (u_2, v_2) to E'. For each red edge $(x, y) \in E$, add this from first copy to the second one, i.e., add (x_1, y_2) and (x_2, y_1) to E'.
2. We check if there is a path from s_1 to t_1 in G' using DFS, and return a walk satisfying the requirements exists in G if and only if the path in G' exists.

Proof of Correctness: The first direction: if there is a path from s_1 to t_1 in G' then there is a walk with even number of red edges in G. Let $(s, v_1, v_2, \ldots, v_k, t_1)$ be a path from s_1 to t_1 in G'. We know this path contains an even number of edges crossing between V_1 and V_2, as we start from $s_1 \in V_1$ and end in $t_1 \in V_1$. Thus, the corresponding edges in G form a walk from s to t with even number of red edges.

The second direction: if there is a walk with even number of red edges in G then there is a path from s_1 to t_1 in G'. Let $P = (s, v_1, \ldots, v_k, t)$ be a walk from s to t with even number of red edges in G. If any edge (x, y) in this path is a blue edge, there is an edge (x_1, y_1) or (x_2, y_2) which can be used to stay in the same copy in G' and go from x_i to y_i for $i \in \{1, 2\}$ depending on which copy the walk in G' is in currently. If there is a red edge (x, y) in this walk, we can use either (x_1, y_2) or (x_2, y_1) in G' to move to the other copy of V and not the copy where the edge starts with x. As we have even number of red edges in the walk, we will move back to the first copy when we traverse all these red edges, and we will have a path from s_1 to t_1.

Runtime: Graph G' has $2n$ vertices and $2m$ edges and it takes $O(2n + 2m)$ time to construct it. Also, DFS on this new graph takes $O(2n + 2m) = O(n + m)$ time, so the total runtime is $O(n + m)$.

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 edges in F from G leaves the graph $G - F$ without any cycle. Describe and analyze an $O(m \log m)$ time algorithm to compute the minimum-weight feedback edge set of a given graph $G = (V, E)$ with *positive* weight w_e on each edge $e \in E$.

(25 points)

A complete solution consists of three part: the algorithm or reduction (**10 points**), the proof of correctness (**10 points**), and the runtime analysis (**5 points**).

Hint: Observe that the graph $G - F$ should be a spanning tree. (You have already seen the rest of the solution in your homework!)

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 $F = G - T$ 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). Consider the following different (and less efficient) algorithm for computing an MST of a given undirected and connected graph $G = (V, E)$ with edge weight w_e on each $e \in E$:

1. Sort the edges in decreasing (non-increasing) order of their weights.
2. Let $H = G$ be a copy of the graph G.
3. For $i = 1$ to m (in the sorted ordering of edges):
   (a) If removing e_i from H does not make H disconnected, remove e_i from H.
4. Return H as a minimum spanning tree of G.

Prove the correctness of this algorithm, i.e., that it outputs an MST of any given graph G (we ignore the runtime of this algorithm in this problem). **(+10 points)**

Solution. We first prove the following claim.

Claim: In any graph $G = (V, E)$, if an edge $e \in E$ has the largest weight among edges of some cycle in G, then there exists an MST of G that does *not* contain the edge e.

Proof: Fix any MST T of G. If $e \notin T$ we are already done. Otherwise consider $T - \{e\}$. A tree minus an edge consists of two separate connected components $S, V - S$. Let $e = \{u, v\}$ and since e is maximum weight edge of some cycle, we know there is another path P from u to v in G so that for all $e' \in P, w_{e'} \leq w_e$.

At least one edge of path P should cross the cut $(S, V - S)$ as $u \in S$ and $v \in V - S$ (or vice versa). Let that edge be e' (breaking the ties arbitrarily). Consider the graph $T' = T \cup \{e'\} - \{e\}$. Firstly, $w_{e'} \leq w_e$ as stated earlier and thus the total weight of T' is at most equal to the weight of T. Moreover, T' is connected since we connected two connected components S, $V - S$ by an edge e'. Since T' also has $n - 1$ edges (as T had $n - 1$ edges) and is thus a tree, we have that T' is another MST of G. But T' does not have the edge e as desired. This concludes the proof of the claim. □

We now use this claim to prove the correctness of the algorithm.

Firstly, the algorithm always outputs a tree. This is because the only edges that are not removed from G are the ones that removing them disconnects G. We now argue that at every step of the algorithm, H is a superset of some MST of G. Since at the end, H is itself a tree, we obtain that H should be a MST.

We prove this by induction over index i. At the beginning of the algorithm, H clearly is a superset of some MST of G because $H = G$ (induction base). Suppose H is superset of some MST T of G at the end of iteration i and we prove that this is the case at the end of iteration $i + 1$ also (induction step).

If removing e_{i+1} in this iteration makes H disconnected, we will simply not remove it. Hence H remains the same after this step and by induction hypothesis still contains an MST of G. Otherwise, if removing e_{i+1} does not make H disconnected, we know that there should be a cycle C in H containing this edge. Moreover, this cycle C cannot contain any of the edges e_j for $j < i + 1$ that are still in H as keeping e_j meant that removing it would make H disconnected which means e_j is not part of any cycle in H. As we sorted the edges in decreasing (non-increasing) order of their weights, we know e_{i+1} has the largest weight among the edges of the cycle C. By our claim, we know that there is an MST T of H (before deleting e_{i+1}) that does not use the edge e_{i+1} and by induction hypothesis, T is also an MST of G. As such, T remains a subset of $H - e_{i+1}$ as well after this step, proving the induction step and so H remains a superset of some MST of G.

This concludes the proof of correctness of the algorithm.