CSCI-GA.3520-001 Honors Analysis of Algorithms October 6, 2016

Solutions to Problem Set 2

Name: Daniel Wichs *Due: October 6, 2016*

Problem 1

Let $G = (V, E)$ be a connected graph and g, h be two edge-cost functions such that for all $e, e' \in E$, $h(e) \leq h(e') \Leftrightarrow g(r) \leq g(e')$. Show there is a single tree which is an MST with respect to g, h.

Solution: We simply show that any tree constructed by Kruskal's algorithm using the function g, is also a valid output of Kruskal's algorithm using the function h (recall that Kruskal's algorithm does not produce a unique tree). To see this, we notice that the cost-functions in Kruskal's algorithm are only used to select a minimal cost edge. But a minimal cost-edge under the function g *is also* a minimal-cost edge under the function h and hence, by the correctness of Kruskal, if we run it with cost function g, we also get an MST under the function h.

□

Problem 1.1

Assume that the edge-costs are time varying and that the cost of an edge e at time t is given by some degree 2 polynomial $f_e(t)$. Find the time t at which the cost of the MST of G is minimized.

Solution: The main idea is to partition the time-line into intervals so that, for all values of t within an interval, the ordering of the edges with respect to cost is preserved. Using the observation from the first part, we can then find a single candidate MST per interval and this will be an MST for all times t within that interval. Lastly, we take the smallest candidate.

So we are only left with the task of partitioning the time-line. We notice that, if at some time t the ordering of the edge costs changes, then $f_e(t) = f_{e'}(t)$ for some $e, e' \in E$. Since f_e is quadratic, there are at most two solutions to any such equation. Hence, by taking all pairs $e, e' \in E$ and finding the solutions of $f_e(t) = f_{e'}(t)$, we get at most $m = 2|E|(|E|-1)/2$ points $t_1, \ldots t_m$ at which the ordering can change. By sorting the points, we can also assume that $t_1 \leq t_2, \ldots, \leq t_m$ and hence we get our $m+1$ time intervals $[-\infty, t_1], [t_1, t_2], \ldots, [t_{m-1}, t_m][t_m, \infty]$.

The idea is to run Kruskal's algorithm on each interval, get the cost of the MST as a function of t and minimize the function in that interval(Minimising takes constant time). Then we do a linear scan on all the intervals and get the minimum of all the minimum values.

For the run-time analysis, we run $m = \mathcal{O}(|E|^2)$ copies of Kruskal each of which takes $\mathcal{O}(|E| \log(|E|))$ for a total of $\mathcal{O}(|E|^3 \log |E|)$. The other steps are: finding the m intervals, sorting the intervals, finding the smallest of the candidate MSTs. They take $O(|E|^2)$, $\mathcal{O}(|E|^2 \log(|E|))$, $\mathcal{O}(|E|^2)$ respectively and hence the total run time is $\mathcal{O}(|E|^3 \log |E|)$.

□

Problem 2

Let G be an n-vertex connected graph with costs on the edges. Assume that all the edge costs are distinct.

1. Prove that G has a unique minimum cost spanning tree.
2. Give a polynomial time algorithm to find a spanning tree whose cost is the second smallest.
3. Give a polynomial time algorithm to find a cycle in G such that the maximum cost of edges in the cycle is minimum amongst all possible cycles. Assume that the graph has at least one cycle.

Solution:

1. Let T be the MST that is computed by Kruskal and assume that $T' \neq T$ is some other MST. Let us look at the first iteration of Kruskal which adds an edge e to T such that $e \notin T'$ (such an iteration must exist if $T \neq T'$). Then e is the *unique minimal* edge crossing some two different trees C_1, C_2 such that C_1, C_2 are subtrees of both T and T' i.e. $e = (u, v)$ with $u \in C_1$, $v \in C_2$. Since T' is also a tree, there must be some edge $e' = (u', v') \in T'$ crossing between C_1, C_2 i.e. $u' \in C_1, v' \in C_2$. Furthermore the cost of e' is strictly greater than that of e. Now we claim that $\hat{T} := T' \cup \{e\} - \{e'\}$ is also a spanning tree. To see this, we note that C_1, C_2 are subtrees of T', and hence there are paths $u' \longrightarrow u$ and $v' \longrightarrow v$ in T'. Hence for any path crossing $e' = (u', v')$ in T', we can re-route it to a path crossing $e = (u, v)$ in $\hat{T}$. Lastly, it is easy to see that the cost of $\hat{T}$ is strictly smaller than that of T' which contradicts the fact that T' is an MST. Therefore there is no MST $T' \neq T$.
2. The simplest solution to this is to just take each edge $e_i \in E$ and find a candidate minimum spanning tree T_i using Kruskal on the set of edges $E - \{e_i\}$. Also find the minimal spanning tree T on the full graph. Then sort the trees T_i by their cost and take the smallest one whose cost is greater than that of T. To see correctness, let T' be a second smallest spanning tree. Then there must be some edge $e_i \in T - T'$ (otherwise T' contains T and hence has a cycle). When we run Kruskal on $E - \{e_i\}$ we get a tree T_i whose cost is at most that of T' (by correctness of Kruskal) but strictly greater than T (by the uniqueness of the MST T as we saw in part (1)). Therefore, since T' is second-smallest, the cost of T_i is the same as that of T', and hence T_i is second smallest.
3. Run Kruskal's algorithm until the first time it picks an edge (u, v) such that u, v are in the same connected component. Then the path $u \longrightarrow v$ in the component together with the edge (u, v) forms a cycle C which we claim satisfies the desired property. To see this, assume there is some other cycle C' whose maximal edge cost is strictly smaller than C. Then all of edges of C' would have been considered by Kruskal before the edge (u, v) (since Kruskal picks edges in order of increasing cost) and hence (u, v) could not be the *first* edge which causes a cycle.

□

Problem 3

You are given a set of n intervals on a line: $(a_1, b_1], \ldots, (a_n, b_n]$. Design a polynomial time greedy algorithm to select minimum number of intervals whose union is the same as the union of all intervals.

Solution:

First, sort the intervals in increasing order by a_i. Our algorithm will essentially keep track of a right-most frontier s which we try to grow greedily while ensuring that we always cover everything possible to the left of s. Set s to initially be the minimal value of a_i. Then repeat the following: Find the interval with maximal b_i satisfying $a_i \leq s < b_i$.
(1) If such an interval $(a_i, b_i]$ exists then add it to the solution set and update $s := b_i$.
(2) If such an interval does not exist then set s to be the smallest value of a_i which is greater than s.

Correctness:
First, we need to show that our solution set indeed includes the union of all intervals. To do so we analyze the invariant that, after each iteration, our current solution set covers all the points possible to the left of s. This holds in the beginning since s is set to the minimal a_i (i.e. there are no points to the left of s). If it holds in the beginning of some iteration then, during the iteration, s gets updated either:
(1) Because we add an interval $(a_i, b_i]$ satisfying $a_i \leq s < b_i$ and update $s_{new} := b_i$. But then we cover all points between s and s_{new} so the invariant continues to hold.
(2) Because no interval with $a_i \leq s < b_i$ exists and hence we update s_{new} to be the smallest value of a_i which is greater than s. Therefore there are no points between the s and s_{new} in the union of all intervals and, again, the invariant then continues to hold.

Now we need to show that our solution is minimal. Assume contradiction, let S be the set of intervals in our solution of size ℓ, and let some other set I of intervals of size $k < \ell$ be optimal. Sort both S and I in increasing order by the interval start points. Let S_j, I_j denote the union of the first j in intervals in S, I respectively (according to the above sorted order). Then we will show that $I_j \subseteq S_j$ by induction.

This is true for $j = 1$ since we pick the interval that starts at the left most point, and if there are multiple such intervals, we pick the one with maximum b_i value.

Assume it is true for j. Let s_j be the value of s in our algorithm after the first j intervals were added (If the j^{th} interval added was $(a_j, b_j]$, then $s_j \geq b_j$). Let t_j be the largest point contained in I_j so that $t_j \leq b_j \leq s_j$ by the inductive hypothesis.

Let the j^{th} interval in S and I be the intervals $(a_j, b_j]$ and $(a'_j, b'_j]$ respectively. $(a_{j+1}, b_{j+1}]$ is chosen based on the maximal b_{j+1} satisfying $a_{j+1} \leq s_j < b_{j+1}$.

Now, after the $j + 1^{th}$ addition, if I_{j+1} should have a point that is not in S_{j+1}, it CANNOT be the case that $a_{j+1} > s_j$, because that implies I_j must be equal to S_j, and therefore the interval $(a'_{j+1}, b'_{j+1}]$ must be a subset of $(a_{j+1}, b_{j+1}]$ because of construction.

In the other case, when $a_{j+1} \leq s_j$, we need $b'_{j+1} > b_{j+1}$ for there to be a point in I_{j+1} that is not in S_{j+1}, since we already have $a'_{j+1} \leq s_j$, which is not possible based on the construction.

Therefore $I_{j+1} \subseteq S_{j+1}$. This completes the induction and shows that, if $|I| = k$ then S_k covers I. This contradicts our assumption that $|S| = \ell > k$ since we never include an interval unless it adds points. Therefore, $S = S_k$ and $|S| = |I| = k$ is optimal.

Run-Time:
The run-time is polynomial, the sorting takes $\mathcal{O}(n \log(n))$ time and each interval is only looked at once after sorting and hence the algorithm runs in $\mathcal{O}(n \log(n))$ time.

□

Problem 4

Given a list of natural numbers $(d_1, \ldots, d_n)$ decide if this sequence is the degree sequence of *some* n vertex graph.

Solution:

First let us prove the following lemma.

Lemma 1 *A non-increasing sequence $(d_1, \ldots, d_n)$ is a degree sequence of some n- vertex graph if and only if*

$$(d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n) \tag{1}$$

is a degree sequence of some $(n-1)$-vertex graph.

$\Rightarrow$ Assume there is a graph G' with vertices $v_2, \ldots, v_n$ whose degrees follow the sequence (1). Then, by adding a new vertex v_1 with edges to the vertices $v_2, \ldots, v_{d_1+1}$, we get a graph G whose degree sequence is $(d_1, \ldots, d_n)$.

$\Leftarrow$ Assume that the graph G has vertices $v_1, \ldots, v_n$ of degrees $d_1, \ldots, d_n$. Let us first show that there is a graph G' with the same degree-sequence as G and where the d_1 neighbors of v_1 form the set $S = \{v_2, \ldots, v_{d_1+1}\}$. We do this by a series of transformations. Assume that $v_i \in S$ is not neighbor of v_1 while $v' \notin S$ is a neighbor of v_1. Let v'' be an arbitrary neighbor of v_i such that v'' is not connected to v'. Then, by deleting an edges $(v_1, v'), (v_i, v'')$ and adding the edges (v_1, v_2), (v', v'') we preserve the degrees of all vertices and the number of neighbors of v which are also in S increases by 1. Note a very important point here, which is that v' and v'' cannot already be connected. But because the degrees are in non-increasing order, $deg(v_i) \geq deg(v')$, and therefore we can always find such a v''. By repeating this transformation, we end up with a graph G' with the same vertex degrees as G such that the neighbor set of v_1 is exactly S. Now, by removing v_1 and all of its edges, we get a graph G'' whose degree-sequence is given by (1).

Given the above lemma, it becomes clear that we can use the following decision algorithm: (Assume we have spent $\mathcal{O}(n \log n)$ time first to sort the degrees so that the sequence $(d_1, \ldots, d_n)$ is non-increasing.

$$f(d_1, \ldots, d_n) := \begin{cases} true & n = 1, d_1 = 0 \\ false & n = 1, d_1 \neq 0 \\ f(reverse\ sorted(d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)) & n > 1 \end{cases}$$

which runs $\mathcal{O}(n)$ recursive calls each taking at most $O(n)$ time (to sort), so the total time is $\mathcal{O}(n^2)$. Note that in each iteration you don't necessarily have to sort the entire array again, you could simply merge the first d_1 elements and the rest of the array in linear time.

□

Problem 5

Suppose you have an unrestricted supply of pennies, nickels, dimes, and quarters. You wish to give your friend n cents using a minimum number of coins. Describe a greedy strategy to solve this problem and prove its correctness.

Solution: The greedy algorithm is straightforward, first give him as many quarters as you can without exceeding n cents, then repeat for dimes, nickels and pennies in that order.

Proof of correctness: Note that greedy logic won't work for any set of coin denominations, if you want to give someone 8 cents and you have coins worth 1 cent, 4 cents and 6 cents, this algorithm states that you should give him one 6 cent coin, and two 1 cent coins, whereas the optimal strategy would be to give him two 4 cent coins.

Now, let's prove that for the given coin denominations of 1, 5, 10, and 25, this strategy works.

Observation 2 *No optimal solution can have* ≥ 5 *pennies(use nickel instead),* ≥ 2 *nickels(use dime instead),* ≥ 3 *dimes(use a quarter and a nickel instead).*

Claim 3 *Say the denominations, in increasing order, are labelled* a_1, a_2, a_3, a_4*. If we wish to pay* x *cents such that* $a_i \leq x < a_{i+1}$*, both the optimal strategy and the greedy strategy pick* a_i*.*

We prove this using a case by case analysis.

1. $1 \leq x < 5$, this is clearly true as there is no way to pay other than just pennies.
2. $5 \leq x < 10$. In this case, we pick a nickel, as the alternative is to use ≥ 5 pennies.
3. $10 \leq x < 25$. In this case, we pick a dime as the alternative is to use ≥ 5 pennies or ≥ 2 nickels (or both), none of which can be optimal.
4. $x \geq 25$, For this case, notice that based on the observation above, the maximum x we can generate with our constraints on the number of pennies, nickels and dimes is $(4 * 1 + 1 * 5 + 2 * 10) = 29$. So for any $x > 29$, we have to use quarters to get the optimal solution. Also, for any $26 \leq x \leq 29$, notice that we have to use $x - 25$ pennies, and to get 25 cents the optimal way is to pick a quarter.

The proof that greedy algorithm is optimal is concluded by justifying each step of greedy algorithm recursively using the logic above. (Once you justify that the optimal solution picks a_k, consider $x - a_k$ and repeat) □