CS 5084 - Algorithms - Design and Analysis

Assignment 4: Kleinberg and Tardos - Chapter 04 - Greedy Algorithms
Exercises 4.1, 4.2, 4.3, 4.5, 4.8, 4.9, 4.21

For full credit, please adhere to the following:

- Unsupported answers receive no credit.
- All answers can be typed or handwritten, and should be readable.
- Submit the assignment in one pdf file

1. (10 points) **Exercise 4.1**: Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

   Let G be an arbitrary connected, undirected graph with a distinct cost, $c(e)$ on every edge, e. Suppose e^* is the cheapest edge in G; that is, $c(e^*) < c(e)$ for every edge, $e \neq e^*$. Then, there is a minimum spanning tree T of G that contains the edge, e^*.

   Solution:

   This statement is true.

   It was shown in the lectures that Kruskal's algorithm produces a minimum spanning tree for a given graph, $G = (V, E)$. Kruskal's algorithm is as follows:

   1. Start without any edges at all.
   2. Build a tree by successively inserting edges from E in order of increasing cost.
   3. As we move through the edges in this order, we insert each edge e as long as it does *not* create a cycle when added to the edges we've already inserted. (If it does create a cycle, we discard it and continue to the next edge.)

   Since e^* is the cheapest edge in G, it would be the first edge selected by Kruskal's algorithm, and is thus part of the minimum spanning tree for G.

2. (15 points) **Exercise 4.2**: For each of the following two statements, decide whether it is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

(a) (8 points) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph, G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, ${c_e}^2$ thereby creating a new instance of the problem with the same graph but different costs.

True or false? T must still be a minimum spanning tree for this new instance.

Solution: This statement is true. Consider Kruskal's alglrithm, which produces a minimum spanning tree according to the following steps:

1. Start without any edges at all.
2. Build a tree by successively inserting edges from E in order of increasing cost.
3. As we move through the edges in this order, we insert each edge e as long as it does not create a cycle when added to the edges we've already inserted. (If it does create a cycle, we discard it and continue to the next edge.)

Squaring the cost of each edge will not change the order in which the edges are added in this algorithm as the relative costs will remain the same. Thus, T will still be a minimum spanning tree.

(b) (7 points) Suppose we are given an instance of the Shortest $s-t$ Path Problem on a directed graph, G. We assume that all edge costs are positive and distinct. Let P be a minimum-cost $s-t$ path for this instance. Now suppose we replace each edge cost c_e by its square, ${c_e}^2$ thereby creating a new instance of the problem with the same graph but different costs.

True or false? P must still be a minimum-cost $s-t$ path for this new instance.

Solution: This statement is false. Let G have edges $(s,v),(v,t)$, and (s,t), where the first two edges have cost 3 and the third has cost 5. Then, the shortest path from node s to node t is the single edge, (s,t). However, if the edge costs are all squared, then the shortest path would be the one through node v - namely, $(s,v),(v,t)$.

3. (15 points) **Exercise 4.3**: You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two

locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight, w_i. The trucking station is quite small so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of the greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

Solution:

Say n boxes arrive in the order $b_1, \ldots, b_n$. Say each box b_i has a positive weight w_i, and the maximum weight each truck can carry is W. To pack the boxes into N trucks *preserving the order* is to assign each box to one of the trucks $1, \ldots, N$ so that:

- No truck is overloaded: the total weight of all boxes in each truck is less or equal to W.
- The order of arrivals is preserved: if the box b_i is sent before the box b_j (i.e. box b_i is assigned to truck x, box b_j is assigned to truck y, and $x < y$) then it must be the case that b_i has arrived to the company earlier than b_j (i.e. $i < j$).

We prove that the greedy algorithm uses the fewest possible trucks by showing that it "stays ahead" of any other solution. Specifically, we consider any other solution and show the following. If the greedy algorithm fits boxes $b_1, b_2, \ldots, b_j$ into the first k trucks, and the other solution fits $b_1, \ldots, b_i$ into the first k trucks, then $i \leq j$. Note that this implies the optimality of the greedy algorithm, by setting k to be the number of trucks used by the greedy algorithm.

We will prove this claim by induction on k. The case $k = 1$ is clear; the greedy algorithm fits as many boxes as possible into the first truck. Now, assuming it holds for $k - 1$: the greedy algorithm fits j' boxes into the first $k - 1$, and the other solution fits $i' \leq j'$. Now, for the k^{th} truck, the alternate solution packs in $b_{i'+1}, \ldots, b_i$. Thus, since $j' \geq i'$, the greedy algorithm is able at least to fit all the boxes $b_{j'+1}, \ldots, b_i$ into the k^{th} truck, and it can potentially fit more. This completes the induction step, the proof of the claim, and hence the proof of optimality of the greedy algorithm.

4. (15 points) **Exercise 4.5**: Let's consider a long, quiet country road with houses scat-

tered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

Solution:

Consider the following greedy algorithm for this problem:

1. Start at the western end of the road and begin moving east until we hit the first house, h.
2. From location, h, move east four miles.
3. Place the first base station at this location.
4. Delete all houses with four miles of of this station both in the eastward and westward directions.
5. Repeat this algorithm on the remaining houses.

Here's a slightly more formal way to express this algorithm:

1. For any position on the road, define its *position* as the number of miles east from the western end of the road.
2. Place the first base station at the largest position (i.e. the furthest east) on the road so that all houses to the west are covered.
3. Remove all houses to the east that are also covered by the first station.
4. Repeat this algorithm on the remaining houses.

The output of this algorithm will be the set of positions ,$S = \{s_1, s_2, \ldots s_m\}$, listed in increasing order of position, such that for $i = 1, \ldots m$, station $i + 1$ will be placed at a location, s_{i+1}, as large as possible, (i.e., as far east as possible) so that all houses are covered between station, i at location, s_i and station, $i + 1$ at location s_{i+1}.

We prove this algorithm is optimal using the *stay ahead* approach and using induction.

Suppose we have the output of this algorithm, $S = \{s_1, s_2, \ldots s_m\}$, and we also have the output of some other algorithm that is optimal, $T = \{t_1, t_2, \ldots t_n\}$ where the

stations in T are also listed in increasing order of position. We need to show that the number of stations produced by our algorithm is equal to that of the the optimal algorithm, i.e., that $m = n$.

We claim that for $s_i >= t_i$ for each i, and prove this by **induction**.

Base case: First, we show this to be true for k = 1. Since s_1 is as large as possible by how our algorithm, it must be the case that $s_1 >= t_1$. Thus, all houses starting from the western end that are covered by the station at t_1 are also covered by the one at s_1.

Inductive Hyposthesis: Now, we make the assumption that our postulate is true for some $k \geqslant 1$. That is, we assume that $s_k \geqslant t_k$ for some $k \geqslant 1$.

Inductive Step: Now we show show our postulate is true for $k + 1$. The inductive hypothesis implies that for the first k stations, the houses covered up through station k at s_k by our algorithm cover those houses up through station k at t_k of the optimal algorithm. This means that when we place station, $k + 1$ at location s_{k+1} in our algorithm, it will be at least as large (i.e., as far east) as station $k + 1$ at location t_{k+1} of the optimal algorithm. Thus, $s_{k+1} \geqslant t_{k+1}$.

Thus, by induction, we have that for each i, $s_i >= t_i$.

Now, let's suppose $m > n$. This means then, that s_n fails to cover all the houses. But since we've shown that it must be the case that $s_n > t_n$, it must also be the case that the optimal algorithm fails to cover all the houses, as well, in which case our optimal algorithm does not solve the problem and is thus, not optimal at all. Thus, under this assumption, we would have a contradition.

Thus, it must be the case that $S = \{s_1, s_2, \ldots s_m\}$ covers all the houses and that m is as small as possible, i.e., $m = n$. Thus, our algorithm is optimal.

5. (15 points) **Exercise 4.8**: Suppose you are given a connected graph, G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Solution: We will show this to be true using induction for any connected graph, $G_n = (V_n, E_n)$ with a number of nodes, n, for all integers, $n \geqslant 2$.

Base Case: First we show this is true for $n = 2$. Suppose we have a graph, $G_2 = (V_2, E_2)$ where V_2 is a set of nodes of size $n = 2$. Then $T_2 = (V_2, E'_2)$ is a minimum spanning tree of G_2 where E' consists of a single edge, $e \in E$ connecting the two nodes with minimum cost. Since all edge costs are distinct, there can be no other edge in E with the same edge cost as e, and thus, the minimum spanning tree, T_2, for G_2 must be unique.

Inductive Hypothesis: Now, let's assume that for any graph, $G_k = (V_k, E_k)$ with $k \geqslant 1$ nodes, there is a unique minimum spanning tree, $T_k = (V_k, E_k{}')$.

Inductive Step: Now, we must show that based on our inductive hypothesis, our assertion must be true for $k + 1$, that is we must show that for a graph, $G_{k+1} = (V_{k+1}, E_{k+1})$, G_{k+1} must also have a unique spanning tree that is unique.

Well, let's take a graph, $G_k = (V_k, E_k)$, such that $V_{k+1} = V_k \cup v_{k+1}$ and $E_{k+1} = E_k \cup E(k+1)$ where $E(k+1) = \{\text{edges for which an end point is } v_{k+1}\}$. We know from the inductive hypothesis that G_k has a unique minimum spanning tree, $T_k = (V_k, E_k{}')$. Now, for G_{k+1}, we build our minimum spanning tree, $T_{k+1} = (V_{k+1}, E_{k+1}{}')$ as follows:

- $V_{k+1} = V_k \cup v_{k+1}$, and
- $E_{k+1}{}' = E_k{}' \cup \min(E(k+1))$

Since all edge costs are distinct, both V_{k+1} and $E_{k+1}{}'$ are unique. Thus, T_{k+1} is unique. And so we have established the proof by induction that for any connected graph, $G_n = (V_n, E_n)$ with a number of nodes, n, for all integers, $n \geqslant 2$, the minimum spanning tree for G_n is unique.

6. (15 points) **Exercise 4.9**: One of the motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum *total* cost. Here we explore another type of objective: designing a spanning network for which the *most expensive* edge is as cheap as possible.

   Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with greatest cost.

   A spanning tree T of G is a *minimum-bottleneck spanning tree* if there is no spanning tree T' of G with a cheaper bottleneck edge.

(a) (7 points) Is every minimum-bottleneck tree of G a minimum spanning tree of G? Prove or give a counterexample.

Solution: This is false. We provide a counterexmaple: Let G have vertices $\{v_1, v_2, v_3, v_4\}$, with edges between each pair of vertices, and with weight on each edge from v_i to v_j equal to $i + j$. Then, every tree has a bottleneck edge of weight at least 5.

Consider the tree consisting of a path through the vertices v_3, v_2, v_1, v_4. This is a minimum bottleneck tree for G. However, it is *not* a minimum spanning tree since its total weight is greather than the weight of the tree with edges from v_1 to every other node.

(b) (7 points) Is every minimum spanning tree of G a minimum-bottleneck tree of G? Prove or give a counterexample.

Solution: This is true. We will show this using proof by contradiction. Let's rephrase the statement as an implication in the form of $P \implies Q$.

- Let $P \equiv T$ is a minimum spanning tree of G.
- Let $Q \equiv T$ is a minimum-bottleneck tree of G.

We shall assume $P \wedge \neg Q$. That is we assume T is a minimum spanning tree of G and T is *not* a minimum bottleneck tree of G. This assumption implies that there is some spanning tree, T' of G such that there exists an edge, e in T that is more costly than *every edge* in T'. Now, let's add edge, e to T'. With the addition of e, T' now has a cycle for which e must be the most costly edge (since e is more costly than every edge in T').

The Cycle Property tells us it is impossible for the most costly edge in a cycle to be in a minimum spanning tree. Thus, it must be the case that e, the most expensive edge in the cycle created by its addition to T', cannot be in any minimum spanning tree. However, this leads us to a contradiction since we e was originally taken from T, the minimum spanning tree for G. Thus, by contradiction, our proof is established.

7. (15 points) **Exercise 4.21**: Let us say that a graph $G = (V, E)$ is a *near-tree* if it is connected and has at most $n + 8$ edges, where $n = |V|$. Give an algorithm with running time $O(n)$ that takes a near-tree G with costs on its edges, and returns a minimum spanning tree of G. You may assume that all the edge costs are distinct.

Solution: Consider the following algorithm:

1. Conduct a BFS until we find the first cycle while executing this algorithm.
2. Remove the costliest edge on this cycle. By the Cycle Property, we know that it is impossible for this edge to be in the minimum spanning tree.
3. Repeat the above two steps 9 times so that we eliminate the 9 costliest edges in our graph that form cycles in the graph.

The output of this algorithm is a minimum spanning tree for our graph $G = (V, E)$.

The runtime of the algorithm is as follows: Each BFS search runs in $O(m + n)$. Since $m = n + 8$, we have $O(n + 8 + n) = O(2n + 8) = O(n)$ time. Since we run the algorithm 9 times, we have $9 \times O(n) = O(n)$.