Fall 2019 B503 Homework 2 Solutions

Due date: TBD Lecturer: Qin Zhang

Problem 1 (10 points).

1. In Interval Scheduling problem, if each request not only has a starting time and a finishing time but also has weight, and we want to maximize the total weights of intervals we selected. Can we still use the greedy algorithm that we learned during the lecture (process according to the finishing time)? If yes give a proof; if no give an counterexample.

2. Give a counter example showing that Dijkstra's algorithm does not work for graphs with negative edges. Please simulate the running of Dijkstra's algorithm on your counter example and explain why Dijkstra's algorithm fails in this case.

Solution: 1. We can provide a counter example. Suppose there are three intervals with start/finish time (s_i, f_i) as $(1, 6), (2, 3), (4, 5)$ and weights $100, 1, 1$. Then the greedy algorithm would schedule the last two intervals with total weights 2. However the optimal solution has weights 100.

2. The graph contains 3 nodes s, u, v and three direct edges $(s, u), (u, v), (s, v)$. The weight of edge (s, u), denoted by l_{su} is 2, $l_{sv} = 1$, and $l_{uv} = 10$. We start running Dijkstra's algorithm from s (S is initialized to be $\{s\}$). The first step we will include v to S, and set $d(v) = 1$. However, the shortest path from s to v has length $2 + (-10) = -8 (s \to u \to v)$.

Problem 2 (10 points). During the lecture, we learned a greedy algorithm to solve the Scheduling All Intervals problem. We also learned that a naive implementation of the greedy algorithm would take $O(n^2)$ time. Please describe a more efficient implementation of the greedy algorithm with $O(n \log n)$ running time?

Solution: For each new interval to consider, instead of traversing all existing processors, we maintain a min-heap storing finishing time of last intervals in all existing processors. For each new interval, we only need to compare its start time with the value on the top of the heap. If the start time is larger, then we just add the new interval to the processor on the top, and update its value to the end time of the new interval. Otherwise, we need to use a new processor, and we push the end time of the new interval into the heap. Each heap update takes $O(\log d)$ time, and thus the total running time (including sort) is $O(n \log n + n \log d) = O(n \log n)$.

Problem 3 (10 points). On Thanksgiving day, you arrive on an island with n turkeys. You've already had thanksgiving dinner, so you don't want to eat the turkeys, but you do want to wish them all a Happy Thanksgiving.

However, the turkeys each have very different sleep schedules. Turkey i is awake only in a single closed interval $[a_i, b_i]$. Your plan is to stand in the center of the island and say loudly "Happy Thanksgiving!" at certain times $t_1, \ldots, t_m$. Any turkey who is awake at one of the times t_j will hear the message. It's okay if a turkey hears the message more than once, but you want to be sure that every turkey hears the message at least once.

Design a greedy algorithm which takes as input the list of intervals $[a_i, b_i]$ and outputs a list of times $t_1, \ldots, t_m$ so that m is as small as possible and so that every turkey hears the message at least once. Your algorithm should run in time $O(n \log n)$. Prove that your algorithm is correct.

Solution: First, the algorithm sorts the intervals by end time. Then, it goes through the intervals in order: if b_i is the next end time and Turkey i has not heard the message, then we broadcast happy thanksgiving at time t_i. The naive way of checking if Turkey i has heard the message takes $O(n)$ time but we can improve it to $O(1)$ by simply checking if a_i is before or after the last broadcast time. Then the total running time is dominated by sorting which is $O(n \log n)$.

Correctness proof (prove by contradictory):

1. Hypothesis: There exists an optimal solution that covers more turkeys at tth broadcast.
2. Base case: We add no broadcast and the optimal solution add no broadcast at the beginning of the algorithm.
3. Induction: Assume we added broadcast time $t_0, t_1, \ldots, t_{t-1}$ and we covered the same set of turkeys as the optimal solution. Assume that we will add t_t as the next broadcast time and the optimal solution will add s_t as the next broadcast time.

   Next we only need to show that t_t is as good a choice as s_t. It is easy to conclude that $s_t \leq t_t$ otherwise the turkey k with ending time $b_k = t_t$ will never be covered, then the optimal solution we assumed will not even be a valid solution. If there exits at least one turkey j that is covered by s_t but not t_t, it must have the following property: $a_j \leq s_t \leq b_j < t_t$, which cannot be true as $s_t \leq t_t$.

   So the hypothesis is false and our solution covers the same set of turkeys as the optimal solution with the same number of broadcasts.

Problem 4 (10 points). Consider the following two statements:

1. The first k edges chosen by Kruskal's algorithm have the following property: there is no cheaper acyclic subgraph of G with k edges. (Assume edge costs are distinct)
2. The first k edges chosen by Prim's algorithm (starting from some "root node" r) have the following property: there is no cheaper connected subgraph of G containing the node r along with k other nodes. (Assume edge costs are distinct)

Give a proof if the statement is true or a counterexample if the statement is false.

Solution:

1. The first statement is true. Assume the Kruskal's algorithm stops after find T_k with kth edge and during the algorithm m edges were processed. Assume there exist another set of edges S_k with k edges that are cheaper than T_k.

   For an edge e such that $e \in S_k$ and $e \notin T_k$, it falls into two categories:

   (a) e is one of the $(m-k)$ unchosen edge. Then there will always be another cheaper edge in T_k to replace e.

   (b) e has weight larger than the first m edges. Then there will always be another cheaper edge in T_k to replace e.

   So there is no cheaper acyclic subgraph of G with k edges.

2. The second statement is false. The following is a counter example. Assume a graph of five nodes has the following chain structure:

   $$a - b - c - d - e$$

   and the following edge weights:

   $$e_{ab} = 1, e_{bc} = 10, e_{cd} = 9, e_{de} = 8$$

   If we begin Prim's algorithm from node c, we will get $\{e_{cd}, e_{de}\}$ after two rounds. But there exits a cheaper connected subgraph of G containing c: $\{e_{ab}, e_{bc}\}$.

Problem 5 (10 points). Let G be a connected weighted undirected graph. In class, we defined a minimum spanning tree of G as a spanning tree T of G which minimizes the quantity

$$X = \sum_{e \in T} w_e,$$

where the sum is over all the edges in T, and w_e is the weight of edge e.

Define a "minimum-maximum spanning tree" to be a spanning tree that minimizes the quantity

$$Y = \max_{e \in T} w_e.$$

That is, a minimum-maximum spanning tree has the smallest maximum edge weight out of all possible spanning trees.

(a) Prove that a minimum spanning tree in a connected weighted undirected graph G is always a minimum-maximum spanning tree for G.

(b) Show that the converse to part (a) is not true. That is, a minimum-maximum spanning tree is not necessarily a minimum spanning tree.

Solution:

(a) Suppose toward a contradiction that T is an MST but not a minimum-maximum spanning tree, and say that T' is a minimum-maximum spanning tree. Let (u, v) be the heaviest edge in T. Then deleting (u, v) from T will disconnect it into two set of vertices: A and B. By cut property, edge (u, v) has the smallest weight among edges that connects A and B. As T' must contains an edge to connect A and B, its Y-value is at least the same with edge (u, v). So there is not a tree T' that has smaller Y-value than T, making MST T also a minimum-maximum spanning tree.

(b) Consider a triangle on vertices a, b, c with the following edge weights:

$$e_{ab} = 1, e_{bc} = 2, e_{ac} = 2$$

Then $\{e_{ac}, e_{bc}\}$ is a tree with minimal Y-value but not minimal X-value.

Problem 6 (10 points). One of the first things you learn in calculus is how to minimize a differentiable function such as $y = ax^2 + bx + c$, where $a > 0$. The Minimum Spanning Tree Problem, on the other hand, is a minimization problem of a very different flavor. One can ask what happens when these two minimization issues are brought together, and the following question is an example of this.

Suppose we have a connected graph $G = (V, E)$. Each edge e now has a time varying edge cost given by a function $f_e : R \to R$. Thus, at time t, it has cost $f_e(t)$. We will assume that all these functions are positive over their entire range. Observe that the set of edges constituting the minimum spanning tree of G may change over time. Also, of course, the cost of the minimum spanning tree of G becomes a function of the time t; we will denote this function $c_{G(t)}$. A natural problem then becomes: find a value of t at which $c_{G(t)}$ minimized.

Suppose each function f_e is a polynomial of degree 2: $f_{e(t)} = a_e t^2 + b_e t + c_e$, where $a_e > 0$. Give an algorithm that takes the graph G and the values $(a_e, b_e, c_e) : e \in E$ and returns a value of the time t at which the minimum spanning tree has minimum cost. Your algorithm should run in time polynomial in the number of nodes and edges of the graph G. You may assume that arithmetic operations on the numbers (a_e, b_e, c_e) can be done in constant time per operation.

Solution: Our algorithm works because of the following three property:

1. There are at most two intersections between two polynomials of degree 2. So the total intersections of all m edges is $O(m^2)$, dividing the time line into $O(m^2)$ intervals.

2. If the edge weights sorting order stays the same, the result of the MST is also the same as we will conduct Kruskal's algorithm with the same sequence of edges. So we only need to compute one MST in each time interval. As weight of MST in a fixed time interval is a sum of $n - 1$ polynomials of degree 2, so we could compute the minimum value inside the time interval in $O(n)$ time by adding the polynomials in $O(n)$ time and compute its minimum value in the range in $O(1)$ time.

3. If we change one edge weight of the graph, we could get a new MST by modifying the current MST in $O(m)$ time. Assume edge e changes its weight, find its new rank and perform Kruskal's algorithm based on the new order. This can be done in $O(m)$ as we only need to adjust the sorted array by one element.

In summary, we have $O(m^2)$ intervals to work on. On each time interval, we need $O(m)$ time to compute the new MST tree and $O(n)$ time to compute the minimum value of the sum of $(n-1)$ polynomials. So the total running time is $O(m^2(m+n))$.

Problem 7 (10 points). We are given two arrays of n points on the number line: red points $r_1, r_2, \ldots, r_n \in R$ and blue points $b_1, b_2, \ldots, b_n \in R$. You may assume that all red points are distinct and all blue points are distinct.

We want to pair up red and blue points in the following way: find a bijection $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ so that r_i is matched with $b_{\pi(i)}$. Note that each red point is matched with a unique blue point and vice versa.

We would like to minimize the following objective function:

$$\sum_{i=1}^{n} |r_i - b_{\pi(i)}|$$

Design an $O(n \log n)$ time algorithm that finds the matching (bijection) that minimizes the sum of distances between the matched points. Prove the correctness of the algorithm and analyze its running time.

Solution: The strategy is to pair the points with the same rank. Sort the red and blue points separately. Let's renumber the points so that $r_1 < r_2 < \ldots < r_n$ and $b_1 < b_2 < \ldots < b_n$. Then match r_i with b_i. The total running time of the algorithm is dominated by sorting $O(n \log n)$ time.

We will prove the correctness of the algorithm by induction.

1. Base case: There is an optimal matching that matches r_1 to b_1.

   Without loss of generality, we could assume that $r_1 < b_1$. Assume there is a optimal matching M in which r_1 is matched to b_j and b_1 is matched to r_i. In this case we could show that

   $$|r_1 - b_1| + |r_i - b_j| \leq |r_1 - b_j| + |r_i - b_1|$$

   which violates the assumption that M is optimal.

   We know that $r_1 \leq b_1 < b_j$, then we could prove the base case by considering three possible locations of r_i.

   (a) $r_1 < r_i \leq b_1 < b_j$

   (b) $r_1 \leq b_1 \leq r_i \leq b_j$

   (c) $r_1 \leq b_1 < b_j \leq r_i$

   All three cases can lead to the above inequality, so the base case stands.

2. Inductive: There is an optimal matching that matches r_i to b_i for all $1 \leq i < j$. We only need to prove that we need to pair r_j to b_j.

   We notice that the problem has been reduced to pair $r_j, r_{j+1} \ldots, r_n$ with $b_j, b_{j+1} \ldots, b_n$. Followed by the proof on the base case, we indeed prove that we need to pair r_j to b_j.

So our algorithm is correct.

Problem 8 (10 points). Jane loves tomatoes! She eats one tomato every day, because she is obsessed with the health benefits of the potent antioxidant lycopene and because she just happens to like them very much.

The price of tomatoes rises and falls during the year, and when the price of tomatoes is low, Jane would naturally like to buy as many tomatoes as she can. Because tomatoes have a shelf-life of only d days, however, she must eat a tomato bought on day i on some day j in the range $i \leq j < i + d$, or else the tomato will spoil and be wasted. Thus, although Jane can buy as many tomatoes as she wants on any given day, because she consumes only one tomato per day, she must be circumspect about purchasing too many, even if the price is low.

Jane's obsession has led her to worry about whether she is spending too much money on tomatoes. She has obtained historical pricing data for n days, and she knows how much she actually spent on those days. The historical data consists of an array $C[1, \ldots, n]$, where $C[i]$ is the price of a tomato on day i. She would like to analyze the historical data to determine what is the minimum amount she could possibly have spent in order to satisfy her tomato-a-day habit, and then she will compare that value to what she actually spent.

Give an $O(n)$ time algorithm to determine the optimal offline purchasing strategy on the historical data. Given d, n, and $C[1, \ldots, n]$, your algorithm should output $B[1, \ldots, n]$, where $B[i]$ is the number of tomatoes to buy on day i. Please also prove the correctness of your algorithm. You may get partial credits with slower algorithm.

Solution: We notice that for each day i, we can buy the tomato in a window $W(i) = [i - d + 1, i] \cap [1, n]$. So our strategy works the following greedy way, buy tomato for day i on minimum cost day T_i in window $W(i)$.

Naive implementation takes $O(dn)$ time if we check d days price for each day. To improve the running time to $O(n)$, we need to work out a data structure to record the minimum cost day T_i for each day i in $O(n)$ time.

One possible way to use a list. For each day i

1. We first check the head of the list. If the data of the head is earlier than $max(1, i-d+1)$, remove it. This step removes the expired price.

2. Then we insert price of day i into end of the list. If the price is day i is lower than the previous prices in the list, remove until we encounter a lower price or the list is empty except our current price.

It is easy to see that the first element of the list is always the minimum cost for d size window after our two-steps maintenance. As each day i is at most added once and removed

once, the total running time for maintaining the list is $O(n)$. And each day can be processed in constant time. So the total running time is $O(n)$.