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

Homework #3 Solutions

March 31, 2022

Problem 1. 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 between $[\min_i a_i, \max_j b_j]$ such that any input interval contains at least one of the chosen 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 $G = \{g_1, \ldots, g_k\}$ denote 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 $O = \{o_1, \ldots, o_\ell\}$ denote 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 2. You are given two arrays of positive integers $A[1:n]$ and $B[1:n]$. The goal is to re-order the arrays A and B so that the average difference between entries of A and B with the same index, after the re-ordering, is minimized. In other words, we want to change the orders of numbers in A and B, so that

$$\frac{1}{n} \cdot \sum_{i=1}^{n} |A[i] - B[i]|$$

is minimized.

Design an $O(n \log n)$ time greedy algorithm for this problem.

Example: If $A = [3, 2, 5, 1]$ and $B = [5, 7, 2, 9]$, we can re-order A to $[1, 2, 3, 5]$ and B to $[2, 5, 7, 9]$ to achieve

$$\frac{1}{4} \cdot \sum_{i=1}^{4} |A[i] - B[i]| = \frac{1}{4} \cdot (1 + 3 + 4 + 4) = 3,$$

which you can also check is the minimum value.

Solution. The solution to this problem is similar to that of job scheduling problem studied in the class.

Algorithm:

(i) Sort array A in increasing order.

(ii) Sort array B in increasing order.

Proof of Correctness: In the following, define *cost* of a solution on A and B as $\sum_{i=1}^{n} |A[i] - B[i]|$. Note that minimizing the cost the is equivalent to solving the problem as the quantity in the question is just scaled down of cost by a fixed factor of n.

Let A^G and B^G denote the final state of the arrays A and B under the greedy solution. Note that by the definition of greedy, both A and B are sorted. Let A^O and B^O denote the final state of the arrays A and B under some optimal solution. We can assume without loss of generality A^O is still sorted: if not, sort the array A^O and whenever you swap $A^O[i]$ with $A^O[j]$, also swap $B^O[i]$ with $B^O[j]$. This way, the cost of the solution remains the same, so the new solution is also optimal. Thus, we can assume $A^O = A^G$. Note that we cannot assume B^O is sorted however in the optimal solution (as trying to sort B^O using the same approach, would make A^O *unsorted* again possibly). We are now going to do an exchange argument.

If B^O is also sorted, we are done because that means $A^G = A^O$ and $B^G = B^O$, and thus greedy also outputs an optimal solution.

If B^O is unsorted, it means that there is some index $i \in \{1, \ldots, n\}$ such that $B^O[i] \geq B^O[i+1]$. Define

$$a = A^O[i],\ b = A^O[i+1],\ c = B^O[i+1], \text{ and } d = B^O[i].$$

So, we know that $a \leq b$ (because A^O is sorted), and $c \leq d$, because $B^O[i] \geq B^O[i+1]$. Consider the new solution O' where $A^{O'} = A^O = A^G$ and $B^{O'}$ is the same as B^O except that we *swapped* $B^O[i]$ with $B^O[i+1]$ in $B^{O'}$ instead. This means that

$$\text{cost of } O - \text{cost of } O' = |a - d| + |b - c| - (|a - c| + |b - d|),$$

as all the other terms are equal. As was proven on Piazza (we will repeat the argument below), a standard mathematical inequality implies that whenever $a \leq b$ and $c \leq d$, we have,

$$|a - c| + |b - d| \leq |a - d| + |b - c|,$$

which means that

$$\text{cost of } O \geq \text{cost of } O',$$

by the equation above. Thus, cost of O' cannot increase in this exchange. We can now continue performing this exchange as long as B^O is not sorted so that we eventually sort it exactly as in B^G. This means there is an optimal solution where both $A^O = A^G$ and $B^O = B^G$, implying the correctness of our algorithm.

Runtime Analysis: Each sorting step takes $O(n \log n)$ time (by using merge sort), so the runtime is $O(n \log n)$.

Proof of the inequality: Assume without loss of generality that $a \leq c$ – otherwise, we can simply switch a by c and b by d in the following.

Assuming $a \leq c$ and since $c \leq d$ implies that $|a - d| = |a - c| + |c - d|$. Thus, $|a - d| + |b - c| = |a - c| + |c - d| + |b - c| \geq |a - c| + |b - d|$ by triangle inequality, which proves the inequality.

(For your own solution, you did not need to provide the proof of this inequality.)

Problem 3. You are stuck in some city s in a far far away land and you know that your only way out is to reach another city t. This land consists of a collection of c *cities* and p *ports* (both s and t are cities). The cities in the land are connected by one-way *roads* to other cities and ports and you can travel these roads as many times as you like. In addition, there are one-way *shipping routes* between certain ports. However, unlike the roads, you need a ticket to use these shipping routes and you only have 10 tickets; so effectively you can use at most 10 shipping routes in your journey.

Assume the map of this land is given to you as a graph with $n = c + p$ vertices corresponding to the cities and ports and m directed edges showing one-way roads and shipping routes. Design an algorithm that in $O(n + m)$ time outputs whether or not it is possible for you to go from city s to city t in this land following the rules above, i.e., by using any number of roads but at most 10 shipping routes. **(25 points)**

Solution. A complete solution consists of the algorithm, proof of correctness, and runtime analysis. For the algorithm, we give a graph reduction.

Algorithm (Reduction):

1. Create a graph G with $11n$ vertices and $\leq 11m$ edges as follows:
   - Create 11 copies of vertices V in the map, named $V_1, V_2, V_3, \ldots, V_{11}$. For any vertex $v \in V$, we use $v_i \in V_i$ to denote the copy of v in V_i for $i \in \{1, \ldots, 11\}$.
   - For any road (u, v) in the map, add 11 edges (u_i, v_i), for $i \in \{1, \ldots, 11\}$, to G.
   - For any shipping route (w, z) in the map, add 10 edges (w_i, z_{i+1}), for $i \in \{1, \ldots, 10\}$, to G.
2. Run any graph search algorithm starting from s_1 in G. If the set of vertices reachable from s in G consists of any of the copies of t, i.e., $t_1, t_2, t_3, \ldots, t_{11}$, output it is possible to reach t from s in the map and otherwise output it is not possible.

Proof of Correctness: As any reduction, the proof consists of two parts:

- If there is a way to reach t from s in the map, then at least one of $\{t_1, \ldots, t_{11}\}$ is reachable from s_1 in G.

  Proof: Consider the path $P = s, u^1, u^2, \ldots, u^k, t$ in the map. Suppose there are exactly $\ell \leq 10$ shipping routes in this path and they correspond to edges $(u^{i_1}, u^{i_1+1}), \ldots, (u^{i_\ell}, u^{i_\ell+1})$ in P.

  Then, in G, we can take the path $s_1, u_1^1, \ldots, u_1^{i_1}$ to reach $u_1^{i_1}$ in V_1; then use the edge $(u_1^{i_1}, u_2^{i_1+1})$ to reach vertex $u_2^{i_1+1} \in V_2$; and then continue again until there is another shipping route which now will take us to V_3 and so on. As such, we will eventually end up at t_ℓ and since $\ell \leq 10$, we will never go beyond V_{11}, and thus reach one of the desired vertices.

- If at least one of $\{t_1, \ldots, t_{11}\}$ is reachable from s_1 in G, then there is a way to reach t from s in the map.

  Proof: Suppose t_ℓ for $\ell \leq 11$ is reachable from s_1 in G and consider the s_1-t_ℓ path P in G. This path can use at most 10 shipping route edges as each shipping route edge take us from some V_i to V_{i+1} but it can use any number of road edges as they keep us in the same V_i. Thus, if we translate back the edges of this path to the original roads and shipping routes in the map, the corresponding sequence consists of arbitrary number of roads but at most 10 shipping routes; so there is a valid way of reaching t from s in the map following this sequence.

Runtime analysis: Creating the graph takes $O(n+m)$ time as each vertex and edge of the original map is translated to $O(1)$ vertices and edges in G. Solving graph search can be done using DFS/BFS in $O(n+m)$ time also.

Problem 4. This question is from your textbook (Question 11 in Chapter 05).

A number maze is a $k \times k$ grid of positive integers. A token starts in the upper left corner and your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token *exactly* three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.

Design and analyze an algorithm that in $O(k^2)$ time determines if there is a way to move the token in the given number maze or not. **(25 points)**

Hint: Think of graph reductions – this problem is not that different from the fill coloring problem after all.

Solution. The solution to this problem is very similar to that of fill coloring problem studied in the class.

Algorithm (Reduction):

1. Create the following directed graph $G = (V, E)$:
   - Vertices: for any cell (i, j) in the maze A, we create a new vertex $v_{i,j}$ in the set V;
   - Edges: for any cell (i, j) in the maze A, we add a directed edge from $v_{i,j}$ to vertices

     $$v_{i+A[i][j],j},\ v_{i-A[i][j],j},\ v_{i,j+A[i][j]},\ \text{and}\ v_{i,j-A[i][j]}.$$

     (If at each point one of the indices $i \pm A[i][j]$ or $j \pm A[i][j]$ is *out of bounds* (i.e., < 1 or $> k$, we do not add that edge as there is no vertex corresponding to that entry).
2. Let $s = v_{1,1}$ be the vertex corresponding to the top left of the maze and $t = v_{k,k}$ be the vertex corresponding to the bottom right one. Run any graph search algorithm from s in G to see if the vertex t is reachable from s; if yes, return the maze can be solved, and if no, return the maze cannot be solved.

Proof of Correctness: We prove that $t = v_{k,k}$ in the newly created graph is reachable from vertex $s = v_{1,1}$, if and only if in the number maze, there is a sequence of valid moves from $(1, 1)$ that reaches (k, k). We will be done after proving this since the output of the reduction/algorithm is the same to both problems.

The first direction: let $s, v_{i_1,j_1}, \ldots, v_{k,k}$ be any path in the new graph G. Then we know that: (1) each cell (i_ℓ, j_ℓ) can move to $(i_{\ell+1}, j_{\ell+1})$ by the definition of edges of G. Hence, in the number maze also, we can just follow $(1, 1)$ to (i_1, j_1), and so on until we reach (k, k).

Now for the second direction: let $(1,1),(i_1,j_1),(i_2,j_2),\ldots,(k,k)$ be any sequence of valid moves that allows us to go from $(1,1)$ to (k,k) in the number maze. If we consider the corresponding vertices $s, v_{i_1 j_1}, \ldots, v_{k,k}$, this forms a path in the constructed graph G by the definition of edges of G. This means that s can also take this path to reach t in G.

This concludes the proof of correctness.

Runtime Analysis: It takes us $O(k^2 + k^2)$ time to create this graph on $n = k^2$ and $m \leq k^2$ edges. Running DFS/BFS for graph search also takes $O(n + m)$ time which in the original parameters of the problem is $O(k^2)$. So, the total runtime is $O(k^2)$.
