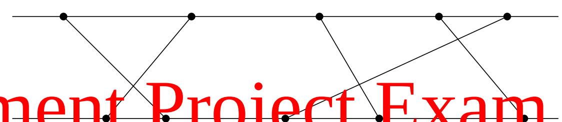
CS/ECE 374 A (Spring 2022)
Homework 10 Solutions

Problem 10.1: Consider the following geometric matching problem: Given a set A of n points and a set B of n points in 2D, find a set of n pairs $S = \{(a_1, b_1), \ldots, (a_n, b_n)\}$, with $\{a_1, \ldots, a_n\} = A$ and $\{b_1, \ldots, b_n\} = B$, minimizing $f(S) = \sum_{i=1}^{n} d(a_i, b_i)$. Here, $d(a_i, b_i)$ denotes the Euclidean distance between a_i and b_i (which you may assume can be computed in $O(1)$ time).

Assume that all points in A have y-coordinate equal to 0 and all points in B have y-coordinate equal to 1. (Thus, all points lie on two horizontal lines.) The points are not sorted. See the example below, which shows a solution that is definitely not optimal.



(a) (20 pts) Consider the following greedy strategy: pick a pair $(a, b) \in A \times B$ minimizing $d(a, b)$; then remove a from A and b from B, and repeat. Give a counterexample showing that this algorithm does not always give an optimal solution.

(b) (40 pts) Let a be the point in A with the smallest x-coordinate. Let b be the point in B with the smallest x-coordinate. Consider a solution S in which a is paired with some point b' with $b' \neq b$, and b is paired with some point a' with $a' \neq a$. Prove that the solution S can be modified to obtain a new solution S' with $f(S') < f(S)$.

(Hint: the triangle inequality might be useful.)

(c) (40 pts) Now give a correct greedy algorithm to solve the problem. (The correctness should follow from (b).) Analyze the running time.

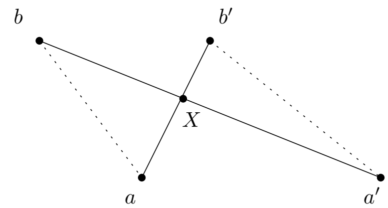
Solution:

(a) One counterexample is $A = \{(0, 0), (1, 0)\}$ and $B = \{(1, 1), (2, 1)\}$.

This greedy strategy would pair $(1, 0)$ with $(1, 1)$, since the pair has the smallest distance 1. Then $(0, 0)$ would be paired with $(2, 1)$, of distance $\sqrt{5}$. The total cost is $1 + \sqrt{5} > 3.236$.

But the optimal solution is to pair $(0, 0)$ with $(1, 1)$, and $(1, 0)$ with $(2, 1)$, with cost $2\sqrt{2} < 2.829$.

(b) By definition of a and b, we know that a is left of a' and b is right of b', as shown in the figure below. Let X be the intersection of the lines ab' and $a'b$.



By the triangle inequality, we have

$$\begin{aligned} d(a,b) + d(a',b') &< (d(a,X) + d(X,b)) + (d(a',X) + d(X,b')) \\ &= (d(a,X) + d(X,b')) + (d(a',X) + d(X,b)) \\ &= d(a,b') + d(a',b). \end{aligned}$$

Create a new solution S' from S by deleting the pairs (a,b') and (a',b) and inserting the pairs (a,b) and (a',b'). Then

$$\begin{aligned} f(S') &= f(S) + (d(a,b) + d(a',b')) - (d(a,b') + d(a',b)) \\ &< f(S) \end{aligned}$$

by the above inequality $d(a,b) + d(a',b') < d(a,b') + d(a',b)$.

(c) The algorithm is simple: pick the smallest a in A and the smallest b in B; output the pair (a,b); remove a from A and b from B; repeat.

Correctness follows from (b), because if the optimal solution does not pair a with b, then there would be a solution with strictly smaller cost: a contradiction.

To bound the running time, note that the algorithm can be equivalently redescribed as follows: sort the points $a_1, \ldots, a_n$ of A in increasing x-order and the points $b_1, \ldots, b_n$ of B in decreasing x-order; return the pairs $(a_1, b_1), \ldots, (a_n, b_n)$.

Since sorting takes $O(n \log n)$ time, the total time is $O(n \log n)$.

Problem 10.2: We are given an unweighted undirected connected graph $G = (V, E)$ with n vertices and m edges (with $m \geq n-1$), We are also given two vertices $s, t \in V$ and an ordering of the edges $e_1, \ldots, e_m \in E$. Suppose the edges $e_1, \ldots, e_m$ are deleted one by one in that order. We want to determine the first time when s and t become disconnected. In other words, we want to find the smallest index j such that s and t are not connected in the graph $G_j = (V, E - \{e_1, \ldots, e_j\})$.

A naive approach to solve this problem is to run BFS/DFS on G_j for each $j = 1, \ldots, m$, but this would require $O(mn)$ time. You will investigate a more efficient algorithm:

(a) (80 pts) Define a weighted graph G' with the same vertices and edges as G, where edge e_i is given weight $-i$. Let T be the minimum spanning tree of G'. Let π be the path from s to t in T. Let j^* be the smallest index such that e_{j^*} is in π. Prove that the answer to the above problem is exactly j^*.

(b) (20 pts) Following the approach in (a), analyze the running time needed to compute j^*.

Solution:

(a) It suffices to prove the following two claims:

Claim 1. *s and t are connected in G_{j^*-1}.*

Proof: Every edge e_i in π has $i \geq j^*$. So, the path π from s to t uses only edges in $E - \{e_1, \ldots, e_{j^*-1}\}$ and remains a path in G_{j^*-1}. $\square$

Claim 2. *s and t are not connected in G_{j^*}.*

Proof: $T - \{e_{j^*}\}$ has two connected components; call them S and $V - S$. We know that $s \in S$ and $t \in V - S$, or vice versa. By a known fact from class, the smallest-weight edge between S and $V - S$ must be in the MST. Since e_{j^*} is the only edge between S and $V - S$ in T, we know that e_{j^*} must be the smallest-weight edge between S and $V - S$. Thus, every edge e_i between S and $V - S$ has weight at least as large as e_{j^*}, i.e., $i \leq j^*$, and so there is no edge between S and $V - S$ in $E - \{e_1, \ldots, e_{j^*}\}$. So, s and t are in different components in G_{j^*}. $\square$

(b) We can compute the MST T in $O(n \log n + m)$ time by Prim's algorithm with Fibonacci heaps (or better with some of the more advanced MST algorithms not covered in class). The path π can be found in $O(n)$ time (by following parent pointers, assuming s is made the root). The index j^* can then be found in $O(n)$ time by a linear scan over π. The total time is therefore $O(n \log n + m)$.

(Alternatively, we can use Kruskal's algorithm and get $O(m \log n)$ time, which is a little worse than Prim's unless the graph is sparse. But actually, in this application, the running time of Kruskal's algorithm can be improved to $O(m\alpha(m, n))$, which is better than as good as Prim's, where $\alpha(\cdot)$ is the inverse Ackermann function; this is because the initial sorting step is trivial as the weights are just the negated indices from $-m$ to -1, i.e., the edges are already given in decreasing order of weights.)

(*Note.* There is a more clever $O(m)$-time algorithm for this problem, which uses median finding and contractions to reduce the number of edges by a half in each round...)

Problem 10.3: Consider the following search problem:

MAX-DISJOINT-TRIPLES:
Input: a set S of n positive integers and an integer L.
Output: pairwise disjoint triples $\{a_1, b_1, c_1\}, \ldots, \{a_{k^*}, b_{k^*}, c_{k^*}\} \subseteq S$, maximizing the number of triples k^*, such that $a_i + b_i + c_i \leq L$ for each i.

For example, if $S = \{3, 10, 29, 30, 35, 55, 70, 83, 90\}$ and $L = 100$, an optimal solution is $\{3, 10, 83\}, \{29, 30, 35\}$, with two triples (there is no solution with three triples).

Consider the following decision problem:

Disjoint-Triples-Decision:

Input: a set S of n positive integers, an integer L, and an integer k.

Output: True iff there exist k pairwise disjoint triples $\{a_1, b_1, c_1\}, \ldots, \{a_k, b_k, c_k\} \subseteq S$, such that $a_i + b_i + c_i \leq L$ for each i.

Prove that Max-Disjoint-Triples has a polynomial-time algorithm iff Disjoint-Triples-Decision has a polynomial-time algorithm.

(Note: One direction should be easy. For the other direction, see lab 12b for examples of this type of question. In Max-Disjoint-Triples, the output is not the optimal value k^* but an optimal set of triples, although it may be helpful to give a subroutine to compute the optimal value k^* as a first step, as in the lab examples.)

Solution:

If Max-Disjoint-Triples has a polynomial-time algorithm, then we can solve Disjoint-Triple-Decision in polynomial time easily: find an optimal solution $\{\{a_1, b_1, c_1\}, \ldots, \{a_{k^*}, b_{k^*}, c_{k^*}\}\}$, and then just return true iff $k \leq k^*$.

Conversely, suppose Disjoint-Triple-Decision has a polynomial-time algorithm A. We first find the optimal value k^*. This can be done by calling A on the input (S, L, k) for $k = 1, 2, \ldots, \lfloor n/3 \rfloor$ until the answer is false. The largest k for which the answer is true is k^*. This requires $O(n)$ calls to A and so takes polynomial time.

(Note: alternatively, with binary search, this requires just $O(\log n)$ calls to A.)

After finding k^*, the following algorithm outputs an optimal set of triples:

1. while $k^* > 0$ do
2.   for every triple of three distinct elements $a, b, c \in S$ with $a + b + c \leq L$ do
3.   call A on the input $(S - \{a, b, c\}, L, k^* - 1)$
4.   if A returns true then
5.   output $\{a, b, c\}$
6. $S \leftarrow S - \{a, b, c\}$, $k^* \leftarrow k^* - 1$
7.   break (i.e., go back to line 1)

Analysis: There are $k^* \leq O(n)$ iterations of the outer while loop, and in each such iteration, we are looping through $O(n^3)$ triples a, b, c. Thus, the total number of calls to A is $O(n^4)$. So, if A runs in polynomial time, the whole algorithm runs in polynomial time.