Advanced Algorithms, Fall 2012

Prof. Bernard Moret

Homework Assignment #10: Solutions

Question 1. (Computational Geometry)
You are given n points on the plane, no three of them collinear. We say p_i and p_j are intimate *if and only if the circle with $\overline{p_i p_j}$ as the diameter does not contain any other points.*

1. *Give an $O(n \log n)$ algorithm to calculate all intimate pairs.*
2. *Consider the complete graph on all n points, with the weight of edge (p_i, p_j) set to the distance between p_i and p_j.*
   Prove that any minimum spanning tree of this graph is made of edges linking intimate pairs.

Solution:
We have seen that an intimate pair must be an edge of the Delaunay triangulation, although the converse need not be true. So an simple solution is to run our algorithm for computing a DT, which runs in the required time (if a deterministic one is needed, we can use our D&C algorithm for Voronoi diagrams and compute the dual in linear time). Then we examine every edge of the DT (there are a linear number of them) and, in linear time overall, we test whether the circumcircle of the edge is empty, returning all edges where the test succeeds. To test whether the circumcircle is empty, we do not need to test all other points (this would take quadratic time), but only points adjacent to one or the other endpoint of the edge under test; this means that we run two tests for each adjacent edge, or a linear number of tests in all. (Another way to run the test is to remove any DT edge that does not cross its dual Voronoi edge—assuming we also computed the Voronoi diagram.) The graph made of all intimate pairs is known as the *Gabriel Graph* (GG).

For the second part, assume that edge $\{a,b\}$ of the Euclidean Minimum Spanning tree (EMST) is not an edge of the GG. Then there exists some vertex c inside the disk of diameter $\overline{ab}$; this vertex c is connected to the EMST by one or more edges, but cannot be connected to both a and b in the EMST (or there would be a cycle). If c is connected to one of a or b in the EMST, say to a, add the other edge $\{b,c\}$; the result is a cycle $\{a,b,c\}$, with the edge $\{a,b\}$ the longest edge in the cycle; thus replacing edge $\{a,b\}$ by the edge $\{b,c\}$ results in a shorter spanning tree, contradicting the hypothesis that our original tree was an EMST. If c is not connected to either a or b in the EMST, add both $\{a,c\}$ and $\{b,c\}$ to the EMST and remove $\{a,b\}$; the resulting graph spans the points but retains one cycle, involving one of $\{a,c\}$ or $\{b,c\}$; remove whichever edge of these two is involved in the cycle (if both, pick one at random). The resulting graph is a spanning tree, which consists of the original (claimed) EMST, minus the $\{a,b\}$ edge, plus one of the $\{a,c\}$ and $\{b,c\}$ edges. The resulting tree is thus shorter than the original, once again a contradiction. Thus the EMST edge must have been a GG edge.

Question 2. (Dynamic Programming)
You are given n distinct points on a line (say on the x axis). Design an algorithm in $O(n^3)$ to find a path to visit all points such that the summation of the delay *for each point is minimized. The delay for a point p_i, with respect to a path P, is the total length of the subpath in P from the beginning of the path to the first arrival at p_i.*

For example, assume you are given 5 *points with coordinates* 0, 9, 10, 11 *and* 20. *If the path starts at* 11, *moves to* 0, *and then reverses direction to move to* 20, *then the delay is* 0 *for point* 11, 1 *for point* 10, 2 *for point* 9, 11 *for point* 0, *and* 31 *for point* 20, *for a total delay of* 45*—you can verify that this is the optimal total delay.*

Solution:
The required running time is rather slow, so we can start by sorting all points along the x axis. In the following we shall thus assume $p_1 < p_2 < \cdots < p_n$. Since any path is continuous, it can be divided into segments that alternate in their direction of travel, say $P = \{p_{i_1} \to p_{j_1}, p_{j_1} \to p_{i_2}, p_{i_2} \to p_{j_2}, \cdots, p_{i_k} \to p_{j_k}, \cdots\}$. Without loss of generality, assume that p_{i_k} is on the left side of p_{j_k}, i.e., we have $p_{i_k} < p_{j_k}$. We claim that we must then have $p_{i_{k+1}} < p_{i_k}$ and $p_{j_{k+1}} > p_{j_k}$, i.e., when the path turns, it must extend beyond the previous turning point. Suppose that such was not the case, i.e., suppose we have $p_{i_{k+1}} \geq p_{i_k}$ for some k;, then the segment $p_{j_k} \to p_{i_{k+1}}$ is redundant since every point on this segment has been already visited with segment $p_{i_k} \to p_{j_k}$. Thus we can simply skip from p_{j_k} to $p_{j_{k+1}}$ to obtain a better path. Thus, in particular, the last segment of P must be the one between the leftmost point and the rightmost point.

Thus we can define subproblems as follows. Consider the points between p_i and p_j, $p_i < p_j$. Define $f(i,j,L)$ as the total delay of the optimal path for these points, under the constraint that the endpoint of the path is p_i, and define $f(i,j,R)$ similarly, but ending at p_j. When there is only one point, we set $f(i,i,L) = f(i,i,R) = 0$, and when there are two points, we set $f(i,i+1,L) = f(i,i+1,R) = d(p_i, p_{i+1})$, where $d(p_i, p_j)$ is the distance between p_i and p_j. To calculate $f(i,j,L)$, we know that the last segment must be $p_j \to p_i$. Suppose the previous left-turning point is k, then the points from i to $k-1$ are first visited by the last segment. Thus, we have the following recursion: $f(i,j,L) = \min_{i<k\leq j}\left\{f(k,j,R) + \sum_{s=i}^{k-1}\left[l(k,j,R) + d(p_j,p_s)\right]\right\}$. Here, $l(k,j,R)$ is defined as the total length of the optimal path. Let k^* be the optimal previous left turning point, i.e., $k^* = \arg\min_{i<k\leq j}\left\{f(k,j,R) + \sum_{s=i}^{k-1}\left[l(k,j,R) + d(p_j,p_s)\right]\right\}$. Then we can update $l(i,j,L)$ using the following recursion: $l(i,j,L) = l(k^*,j,R) + d(p_i,p_j)$. And the initial condition for it is $l(0,0,L) = l(0,0,R) = 0$ and $l(i,i+1,L) = l(i,i+1,R) = d(p_i,p_{i+1})$. Similarly, we have $f(i,j,R) = \min_{i\leq k<j}\left\{f(i,k,L) + \sum_{s=k+1}^{j}\left[l(i,k,L) + d(p_i,p_s)\right]\right\}$, and $l(i,j,R) = l(i,k^*,L) + d(p_i,p_j)$. Finally, the total delay of the optimal path for those n points is given by the better one of $f(1,n,L)$ and $f(1,n,R)$.
Summarily, the algorithm maintain two matrices, f and l, and l is based on the optimal parameter of f. The running time of the algorithm is dominated by filling matrices f; calculating the value of each entry of it takes time $O(n)$ (to achieve this, we need to preprocess to calculate $\sum_{s=i}^{k-1}\left[l(k,j,R) + d(p_j,p_s)\right]$ for all k in linear time); therefore the total running time is $O(n^3)$.

Question 3. (Randomized Algorithms)
Design a randomized algorithm that runs in $\tilde{O}(n+m)$ time and, given n red points and m green points on the plane (no three points are collinear), decides whether there exists a line such that all red points are on one side of this line while all green points are on the other side.

Solution:
This can be solved by using our randomized incremental algorithm for 2D linear programming. The basic idea is that any line will enforce side conditions, with equality allowed for one color only. So, if the equation of the line is $ax + by + c = 0$, then the function $f(x,y) = ax + by + c$

must be strictly less than 0 for all points of one color, and non-negative for all points of the other color. Choosing which color is which allows us to write $m+n$ inequalities (some strict, some with equality allowed). The objective function does not matter, as we simply seek to determine whether there exists any feasible solution (a,b,c) determining the line—we can set the objective function to a constant. Since we have two choices of color (one where red must obey the strict inequality, and one where it is blue that must obey the strict inequality), we must run two linear programs. As our randomized incremental LP solver did not make any provisions for strict inequalities, we can turn them into non-strict ones by requiring $ax+by+c \leq \varepsilon$, where ε is the smallest representable number in our programming language.