Advanced Algorithms, Fall 2012

Prof. Bernard Moret

Homework Assignment #8: Solutions

Question 1.
Consider the following algorithm to calculate the convex hull of the given n points $P = \{p_1, p_2, \cdots, p_n\}$.

```
1  K ← 2;
2  while TRUE do
3  |  m ← ⌈n/K⌉;
4  |  arbitrarily divide P into m disjoint subsets, P_1, P_2, ··· , P_m, each of which contains at most K
   |  points;
5  |  for i = 1 → m do
6  |  |  compute the convex hull of P_i in time O(K · log K), and denote it as H_i;
7  |  end
8  |  compute the point with smallest y-coordinate in ∪H_i, denoted as q_1;
9  |  let q_0 be the point whose y-axis is the same with q_1, and whose x-axis is −∞;
10 |  for k = 1 → K do
11 |  |  for i = 1 → m do
12 |  |  |  compute the point p ∈ H_i that maximize ∠q_{k−1}q_k p, and let it be h_i;
13 |  |  end
14 |  |  compute the point p ∈ {h_1, h_2, ··· h_m} that maximize ∠q_{k−1}q_k p, and let it be q_{k+1};
15 |  |  if q_{k+1} = q_1 then
16 |  |  |  return {q_1, q_2, ··· , q_k};
17 |  |  end
18 |  end
19 |  K ← K^2;
20 end
```

1. *Devise an $O(\log |H_i|)$ algorithm to carry out line 12. More precisely, your algorithm is given a convex polygon with n vertices, $X = \{x_1, x_2, \cdots, x_n\}$, and two points a and b defining a line that leaves the entire polygon to one side; it outputs a vertex x of the polygon such that the angle $\angle abx$ is maximized.*
2. *Prove that the running time of the whole algorithm is $O(n \cdot \log s)$, where s is the number of points in the final convex hull. (Consider the value of K when the algorithm stops.)*

Solution:
This algorithm is known as Chan's algorithm and was published in 1996. Like package-wrapping, this algorithm is output-sensitive—but much faster. The basic idea of Chan's algorithm is to use divide-and-conquer: partition the points into subsets and build the convex hull of each subset using any $O(n \cdot \log n)$ algorithm. The trick is to control the size of the subsets in the partition so that the size decreases very rapidly—much more rapidly than in the usual halving, but not so rapidly that the last round dominates the computation. (The efficiency of any divide-and-conquer scheme depends on careful balancing of the various pieces and perhaps the most important part is to balance the work among the levels of recursion.)

1. We know that the points on a convex hull are actually sorted, so we can use binary search to find the point to maximize the angle $\angle abx$. First, calculate $\angle abx_n$, $\angle abx_1$, and $\angle abx_2$. If $\angle abx_1$ is the biggest of the three, then x_1 is the point that maximizes the angle. Otherwise, we test the middle point x_m, where $m = \lfloor n/2 \rfloor$, by calculating and comparing $\angle abx_{m-1}$, $\angle abx_m$ and $\angle abx_{m+1}$. Again, if $\angle abx_m$ is the biggest among the three, return x_m; otherwise, we can always throw away half of the points by comparing these angles, and iteratively search in the other half.

2. First we compute the running time for a fixed K in the while-loop. The running time of lines 5–7 is $O(m \cdot K \cdot \log K) = O(n \cdot \log K)$. According to the above conclusion, lines 11–13 cost $O(\sum_{i=1}^{m} \log |H_i|) = O(\sum_{i=1}^{m} \log K) = O(m \cdot \log K)$. Line 14 costs $O(m)$. Thus, the whole for-loop from line 10 to line 18 costs $O(K \cdot m \cdot \log K) = O(n \cdot \log K)$. In summary, for a fixed K, the running time is $O(n \cdot \log K)$.

   Now, consider the value of K when the algorithm stops. Notice q_1 must be on the final convex hull, and q_k is the k-th point on the final convex hull along the counter-clockwise direction. In other words, the for-loop from line 10 to line 18 will find consecutive K points on the final convex hull. Thus, when $K > s$, the whole convex hull will be found and the algorithm will stop, since we know that q_{s+1} is equal to q_1. Consequently, when the algorithm stops, we must have $K > s$ and $\sqrt{K} \leq s$.

   Suppose that the while-loop runs t rounds, i.e., in the final round $K = 2^{2^{t-1}}$. The total running time is $\sum_{i=1}^{t} n \cdot \log 2^{2^{i-1}} = n \cdot \log 2 \cdot \sum_{i=1}^{t} 2^{i-1} < n \cdot \log 2 \cdot 2^t = n \cdot \log 2^{2^t} = n \cdot \log K^2 \leq n \cdot \log s^4 = O(n \cdot \log s)$.

Question 2.
Let $P = \{p_1, p_2, \cdots, p_n\}$ be the vertices of a convex polygon. Design a linear-time algorithm to find two points from P such that the distance between them is maximized.

Solution:
What is asked for is the *diameter* of a convex polygon. The linear-time algorithm for this problem is one of the oldest nontrivial algorithms in computational geometry: it was published by M. Shamos in 1978. (You will see this algorithm referred to as the "rotating calipers," for reasons that will soon become apparent.) A diameter is something that can easily be measured on a circle by pushing parallel lines closer and closer until they are both tangent to the circles, while remaining parallel—this is how calipers work. A circle is just an extreme case of a convex polygon (one with an infinite number of vertices), so we can think of a similar approach. Now, however, as we move a pair of parallel lines closer and closer, it is clear that the direction of these lines is crucial: the polygon does not have perfect rotational symmetry. Moreover, we will not achieve tangency, but a related version: the lines will touch the polygon, but each will leave it entirely on one side. A line that touches a convex polygon, but leaves it on one side, is called a *support line*—the analog in the polygonal world of a tangent. Thus we want to compute a pair of parallel support lines that is farther apart than any other pair of parallel support lines. We say that two vertices of the polygon form an *antipodal pair* if they admit two parallel support lines. Shamos' algorithm simply enumerates all antipodal pairs and retains the one that gives rise to the largest distance.

First compute the polygon's extreme points in the y-axis direction. Then construct two horizontal support lines through these two points: this is our first antipodal pair and it gives us

our first bound on the diameter. Next, we rotate the two support lines (around the vertex of the polygon that they touch) simultaneously to keep them parallel until one is flush with an edge of the polygon. Since the line now touches another point, this determines a new antipodal pair, which we compare with our best so far. We continue rotating until we come back to the two extreme points.

Each rotation costs constant time because we need only check the next vertex on the perimeter from each of the two points in the current antipodal pair. Testing the new antipodal pair (computing the diameter it determines and comparing it with the largest diameter found so far) also take constant time. Each new edge can stop the rotation, but the number of edges equals the number of vertices, so the number of rotations is linear in the size of the input and so is the running time.

Question 3.
Let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of n points on the plane. Use the Voronoi diagram of P to devise an $O(n \log n)$ algorithm to compute, for each point in P, its closest neighbor in P.

Solution:
Perhaps the easiest way to solve this problem is to use Voronoi Diagrams directly. First we prove that if q is the closest point to p, then a part of the perpendicular median of $\overline{pq}$ (the segment joining p and p) forms one edge of the Voronoi polygon of p. A point x is on the boundary of the Voronoi polygons of p and q if and only if the circle centered centered at x and passing through p and q contains no other Voronoi site. Moreover, if p and q are the only sites *on* this circle, then x must be an interior point of an edge of the Voronoi polygons (i.e., it cannot be a vertex of Voronoi polygons) and the edge must be part of the perpendicular median of $\overline{pq}$.

Now consider the midpoint m of $\overline{pq}$. Let C_1 be the circle with $\overline{pq}$ as its diameter (therefore m is the center of C_1) and let C_2 be the circle centered at p with $\overline{pq}$ as its radius. Since q is the closest point to p, C_2 contains no other site. Moreover, C_1 is completely contained in C_2. Thus C_1 does not contain any other site either and p and q are the only sites on it. Thus one edge of the Voronoi polygons of p must be part of the perpendicular median of $\overline{pq}$.

The algorithm is then as follows. We first compute the Voronoi diagram of P in $O(n \log n)$ time. Then, for each site in P, we traverse the edges of its Voronoi polygon and find out its nearest neighbor. To do that, we need to traverse each edge of the whole Voronoi Diagram at most twice. Since we know the number of edges is $O(n)$, the traversal takes $O(n)$ time and the total running time of the algorithm is $O(n \log n)$.

Question 4.
You are given a subdivision of the plane into convex polygons (some finite, some infinite); the claim is that this subdivision is in fact a Voronoi diagram. Devise an efficient algorithm that will verify or contradict this claim.

Solution:
The best approach is geometric, using a locus of points. At a Voronoi vertex of degree 3, the sites associated with the 3 polygons are equidistant and determine a triangle with sides perpendicular to the three edges incident upon the Voronoi vertex. Imagine drawing a half-line from the Voronoi vertex through each of the 3 points. The three sites lie on these three lines, one on each, and are mirror images across the polygon boundaries (the perpendicular medians). Thus they form a triangle; with just three unbounded polygons, the solution is not unique, but with

additional sites, we get additional constraints. In particular, if we can determine the three half-lines associated with each Voronoi vertex, we have all of the information needed. (In fact, it would suffice to determine two half-lines associated with two neighboring Voronoi vertices and with the same Voronoi polygon: their intersection must be the Voronoi site for the polygon and all other sites can now be determined through reflection across polygon edges.)

Since the points are symmetric 2 by 2 with respect to the Voronoi edges incident upon the Voronoi vertex, we can write simple relationships to define the angles formed by the newly drawn half-lines with the Voronoi edges incident upon the Voronoi vertex as a function of the angles between these edges (three linear equations in three variables for a Voronoi vertex of degree 3). For a Voronoi vertex of degree 3, these relationships uniquely define the angles (for a Voronoi vertex of higher degree, the relationships may leave one degree of freedom—one of the angles is arbitrary, then all others follow), so that, for any Voronoi polygon with at least two vertices, we can obtain two intersecting segments that define the point associated with the polygon. All of this takes only constant time. These computations are purely local to each Voronoi vertex, so that we can begin by computing all of the angles in linear time, thereby defining, for each vertex of the subdivision, three segments issuing from the vertex and heading into each of the three faces to which the vertex belongs. Knowing all these segments, we can check each face in turn and verify that the intersection is unique and well-defined, which takes linear time overall.

So the problem can be solved in linear time—faster than it takes to compute the Voronoi diagram itself. This is not particularly surprising, since there is a lot more structure in a Voronoi diagram than in an unordered set of points.