Massachusetts Institute of Technology
6.838: Geometric Computation
Professors Piotr Indyk and Seth Teller

Handout 5
October 4, 2001

Solutions to problem set 1

(mostly taken from the solution set of Jan Vondrák)

Mandatory Part

Problem 1. Application of convex hulls: diameter

Let P be a set of points on the plane. The diameter of P is defined as $\max_{p,q\in P} \|p-q\|_2$, where $\|\cdot\|$ is the Euclidean norm. Let CH be the convex hull of P.

- A point p on the convex hull is called *unnecessary*, if it lies on the segment between its predecessor and its successor on CH. Show that removing unnecessary points from P does not change the diameter of P. Also, give an $O(n)$ time algorithm which detects and removes from P all unnecessary points.
- For each CH segment $s = p - p'$, define $anti(s)$ to be the set of points furthest from the line containing s. Show that if CH does not contain any unnecessary points, then the cardinality of $anti(s)$ is at most 2.
- Show that the diameter pair is a subset of $\{p, p'\} \cup anti(p-p')$ for some segment $p-p'$.
- Based on the above observations, give an $O(n \log n)$-time algoritm for finding the diameter of P.

Solution.

- More generally, if P is a polytope and

  $$diam(P) = \|p-q\|, \qquad p, q \in P$$

  then both p and q are vertices of P:

  Suppose p is not a vertex. Then there exists a vector $z \neq 0$ such that $p \pm z \in P$. We can assume that the dot product $(p-q, z)$ is nonnegative (otherwise consider $-z$). By expanding the square of a norm as a dot product, we get

  $$\|(p+z)-q\|^2 = (p-q, p-q) + 2(p-q, z) + (z, z) \geq \|p-q\|^2 + \|z\|^2 > \|p-q\|^2$$

  which contradicts the fact that p and q maximize the distance between two points in P.

  For a given set of points which are on the boundary of the convex hull in cyclic order, we can remove the unnecessary points by simply going around the polygon and marking all points which lie on the line defined by their two neighbors.

- First, if s is a segment of the convex hull, then CH is contained in one of the corresponding halfplanes. (Otherwise points in the relative interior of s would be in the interior of the polygon.) Let l be the furthest line parallel to s which still intersects CH. Because CH is convex, the intersection $l \cap CH$ is either one point or a line segment. In case it is one point, $|anti(s)| = 1$. In case it is a line segment, assuming we have removed the unnecessary points, only the endpoints of $|l \cap CH|$ are in P and $|anti(s)| = 2$.

- Let $diam(P) = \|p - q\|$. Let l_p, l_q be two parallel lines going through p and q, respectively, perpendicular to $p - q$. Then CH is contained in the stripe between l_p and l_q, otherwise there is a point in CH further away from p than q, or vice versa. Moreover, $l_p \cap CH = \{p\}$ and $l_q \cap CH = \{q\}$.

  Now, start rotating the two lines simultaneously, until one of them hits another point in P. Denote the rotated lines by l'_p, l'_q and assume l'_p contains another point $p' \in P$. Then l'_p contains a line segment $p - p'$ and l'_q is still the furthest parallel line intersecting CH (otherwise we would stop rotating sooner because of l'_q). So we have $s = p - p'$ and $q \in anti(s) \subset l'_q$.

- The algorithm can proceed as follows: First, find the convex hull of P in $O(n \log n)$ time. Then remove the unnecessary points in $O(n)$ time. Finally, process the line segments of the convex hull in the clockwise order and for each of them, determine the set $anti(s)$. While finding $anti(s)$ for the first segment may take $O(n)$ time, finding the next one takes only $O(1 + k)$ where k is the number points separating $anti(s)$ and $anti(s')$ in the clockwise order. We simply go around the boundary of CH as long as the distance from s' increases. Due to the previous observations, there can be at most two points which maximize the distance from each line segment, and the diameter of CH can be found as the maximum of all these distances. Going around the whole polygon, we spend $O(n)$ time to determine the diameter. The total running time is dominated by the convex hull algorithm which takes $O(n \log n)$.

Problem 2. Vertical segment intersection.

In the class (Lecture 2) we have seen an algorithm for reporting all segment intersections when all segments are either horizontal or vertical. The description given in the class focused on detecting intersections between horizontal and vertical segments, leaving open the problem of detecting the intersections between vertical and vertical (or horizontal and horizontal) segments. To fill this gap, construct an efficient algorithm, which given a set of vertical (only) segments, reports all pairs of segments having nonempty intersection.

Solution.

Initially, we sort the vertical segments by their horizontal position in $O(n \log n)$ time. Then we can separate them easily into buckets corresponding to different horizontal positions. Intersections can only occur within each bucket. In each bucket, we apply the 1D algorithm to detect intersections between intervals on the line. For a bucket of size n_i, this takes time $O(n_i \log n_i + k_i) \leq O(n_i \log n + k_i)$ where k_i is the number of intersecting pairs in the bucket. The total running time is $O(n \log n + k)$ where k is the total number of intersections.

Problem 3. Textbook, exercise 3.3, p. 60.

A *rectilinear polygon* is a simple polygon of which all edges are either horizontal or vertical. Give an example to show that $\lfloor n/4 \rfloor$ cameras are sometimes necessary to guard a rectilinear polygon with n vertices.

Solution.

The rectilinear polygon would contain $\frac{n}{4}$ parallel "alleys". At least $\frac{n}{4}$ cameras are needed because no camera can see more than one alley.

Problem 4. Textbook, excercise 4.16, p. 94.

On n parallel tracks n trains are going with constant speeds $v_1 \dots v_n$. At time $t = 0$ the trains are at positions $k_1 \dots k_n$. Give an $O(n \log n)$ time algorithm that detects all trains that at some moment are leading. To this end, use the algorithm for computing the intersection of half-spaces.

Solution.

The space-time diagram of each train is given by

$$x_i(t) = k_i + v_i t.$$

If we define a polyhedral set

$$H = \{(t, x) : \forall i; x \geq x_i(t)\}$$

a point (t^*, x^*) on the bottom boundary of H has the property that no train is beyond x^* at time t^*. A train such that $x_i(t^*) = x^*$, i.e. a train which defines the boundary at that point, is leading at time t^*. It remains to determine which trains define the boundary of H. This can be done using the half-space intersection algorithm from the book.

Optional Part

Problem A. Textbook, excercise 2.12, p. 43.

Let S be a set of n triangles in the plane. The boundaries of the triangles are disjoint, but it is possible that a triangle lies entirely inside another triangle. Let P be a set of n points in the plane. Give an $O(n \log n)$-time algorithm that reports each point in P lying outside all triangles.

If you want, you can assume that all x-coordinates of points in P as well as of vertices of triangles in T are different.

Solution.

We use the plane sweep method. First, let's sort all the points and triangle vertices by their horizontal position. We associate events with each object:

- Test Point - for a single point
- Insert Triangle - for the leftmost vertex of a triangle
- Modify Edge - for the middle vertex
- Remove Triangle - for the rightmost vertex

Between two neighboring events, the interior of a triangle is defined by two linear functions - the top and bottom edge. Therefore we can test the position of a point relative to the triangles in constant time.

During the plane sweep, we keep the "active" objects sorted by their vertical coordinate in a data structure allowing the `Search`, `Insert` and `Delete` operations in $O(\log n)$ time (for example, a dynamically balanced search tree). Since triangles don't intersect, the vertical ordering is well defined. If a triangle vertex is found to be inside another triangle, the inside triangle can be discarded.

We handle the events in this way:

- Test Point - we search for its position among active triangles. If the point turns out to be outside any active triangle, we report it.
- Insert Triangle - insert a new a triangle in the data structure and mark the two leftmost edges as active (to be used in comparisons with points and other vertices).
- Modify Edge - Instead of the edge between the leftmost vertex and the middle vertex, activate the edge between the middle and rightmost vertex.
- Remove Triangle - Remove the triangle from the data structure.

We spend $O(\log n)$ time on each event and the total running time is $O(n \log n)$.

Problem B.

The algorithm works if the horizontal edges are handled consistently. The usual way to avoid degeneracies is to rotate the polygon by a small angle, so that no two vertices have the same y-coordinate. Equivalently, we can set up rules to handle horizontal edges so that the behavior of the algorithm will be exactly the same as if we rotated the polygon slightly, for example clockwise. Specifically, these rules would be:

- When sorting vertices before the plane sweep, use lexicographic ordering - the y-coordinate and then the x-coordinate. Thus all horizontal segments will be processed from left to right.
- When deciding whether a vertex is a split vertex or a merge vertex, consider a horizontal edge on the left as going slight upward and a horizontal edge on the right as downward. Therefore vertices inside a horizontal row will be neither split nor merge vertices.

- When looking for the "helper" vertex, assume that all the vertices on a horizontal sequence of segments are "visible" from the left, but only the rightmost vertex is "visible" from the right.

The rest of the algorithm should remain intact. The same rules should be applied to the monotone triangulation algorithm as well.

Comment: the solution above is correct. However, almost no one pointed out that the triangles produced by this algorithm would have zero area, and would have to be removed by a post-processing step. Some people lost points if their solution simply referred to the text, with no elaboration.

Problem C.

Let the feasible region be

$$F = \{x \in R^d : Ax \leq b\}.$$

We define a new linear program in dimension $d+1$:

$$\max\{\lambda : \lambda \leq 1, \exists x \in R^d; Ax + j\lambda \leq b\}$$

(where j denotes the $(1, 1, \ldots, 1)$ vector).

Assuming that F is nonempty and full-dimensional, there exists a point which is "well-inside", i.e. $\exists\lambda > 0, x \in F$ such that $Ax + \lambda j \leq b$. Since we have the additional contraint $\lambda \leq 1$, the objective function is bounded and an optimum exists. Also, the optimum is strictly greater than zero. When we find any optimum solution (x, λ), we have $\lambda > 0$ and $Ax \leq b - j\lambda < b$ which means that x is well inside F.

Comment: several solutions suggested pushing the initial constraints inward by some small epsilon. This lost points unless an efficient method for determining epsilon was given.

Programming Exercise

Excellent solutions can be seen at
`http://web.mit.edu/drdaniel/www/6.838/ConvexHull/ConvexHull.html` (IE only)
and
`http://graphics.lcs.mit.edu/~jcyang/6.838/pset1/`
These are due to Daniel Vlasic and Jason Yang, respectively.

The 50 points for this section were apportioned as 10 for the basic GUI, 10 for polygon entry, 10 for Convex Hull of a simple polygon, 15 for Andrews's algorithm, and 5 for the explanatory web page.