Spring 1996 **CMSC 754: Solution to Homework 5** Samir Khuller

Due in class: April 24, 1996.

(1) The main idea is to reduce this problem to a point location problem. We sub-divide the plane into regions, so that two points are in the same region if the bullet fired to the right hits the same segment. (This is not strictly true, but describes the intuition.)

We perform a horizontal trapezoidalization of all the segments, to decompose the plane into the regions. For each face of the trapezoid it is easy to compute which segment is hit when a horizontal ray is shot to the right. The trapezoidalization can be done in $O(n \log n)$ time for n disjoint line segments. Each query can be answered in $O(\log n)$ time, once we compute which face the point is in, we know which segment is hit by the bullet shot to the right.

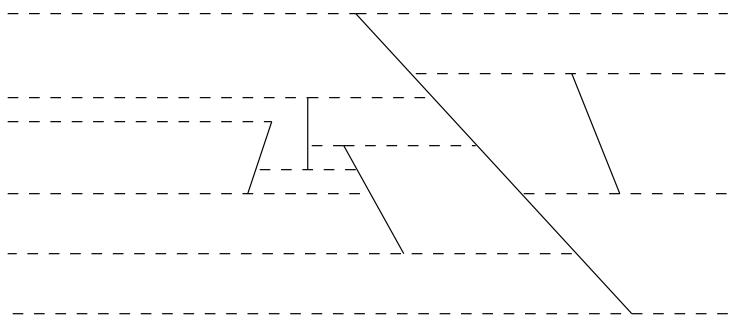

Figure 1: Preprocessing the segments.

(2) (Figures for this problem were supplied by merlin@cs)

For the L_1 distance, 5 cases can arise and are illustrated in Fig. 3. Part (a) shows the case when points p and q share the same y–coordinate. Part (b) shows the symmetric case when p and q share the same x–coordinate. Part (c) and Part (d) show the cases where p and q lie on opposite sides of an orthogonal rectangle whose height is not equal to its width. Part(e) shows the case where p and q lie on an axis aligned square. When p and q line on the other diagonal, the case is symmetric. Note that the points in the shaded areas are equidistant to p and q. The L_∞ cases are similar and are illustrated in Fig. 3.

(3) (a) Let dd' be the diameter of the convex polygon, and let aa' be the longest chord rooted at a. Suppose they do not intersect. Consider the quadrilateral shown in Fig. 4(a). Let the intersection of the lines ad' and da' be at point o. By triangle inequality, $dd' < do + od'$ and $aa' < ao + oa'$. Adding, we get $dd' + aa' < da' + ad'$. Notice that by choice of a', $aa' \geq ad'$ and $dd' \geq da'$, adding gives us a contradiction.

This is not true for maximal chords rooted at different points (see Fig. 4(b)). The maximal chord rooted at a is aa', the maximal chord rooted at b is bb', and they do not intersect.

(b) Initially, the end points of the diameter can be anywhere, and the possible range of values is m for each endpoint. From any vertex A we can find the max chord in $O(m)$ time. This divides the polygon into two chains S_1 and S_2. The diameter must have its end points in the chains S_1 and S_2, since it must intersect AA'. Assume $|S_2| \geq |S_1|$. Let B be the midpoint of S_2. Find the max chord BB' rooted at B. If B' lies in S_1, then since the diameter must intersect both chords, it must be in either BA' and AB', *or* in chains AB and $A'B'$. In either case, we recurse on two chains such that both of them are of size at most $\frac{m}{2}$. (The case when B' is in S_2 is even simpler, since we only need to recurse on the chains BB' and S_1, each of which is of size at most $\frac{m}{2}$.)

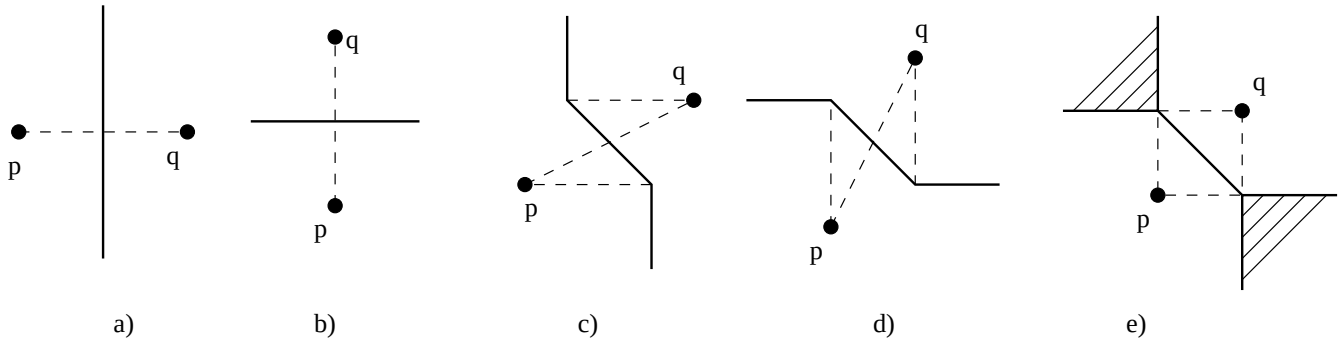

Figure 2: L_1 metric

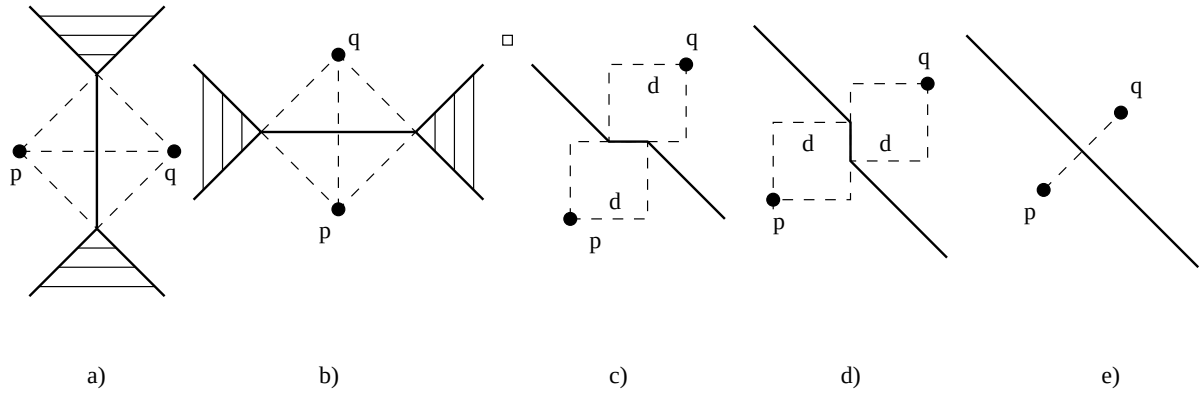

Figure 3: L_∞ metric

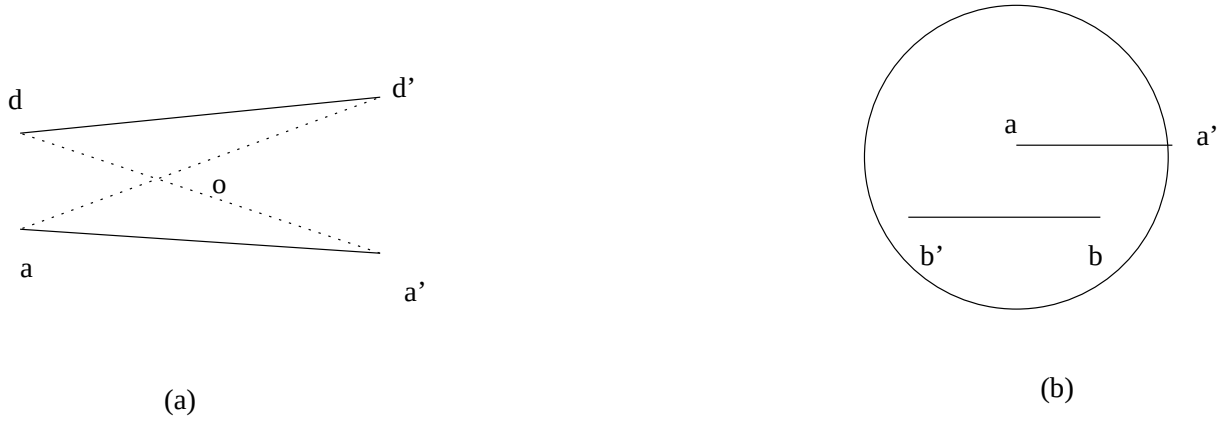

Figure 4: Proof by contradiction.

This leads to a recurrence of the form

$$T(m) = 2T(\frac{m}{2}) + O(m)$$

which has a solution $T(m) = O(m \log m)$.

(c) Suppose the vertices i, j, k form the maximum perimeter triangle, but vertex k is not a point on the convex hull of the set of points. Draw an ellipse with i and j as foci and k as a boundary point. Draw a tangent to the ellipse at point k.

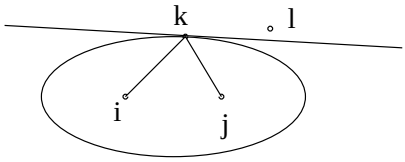

Figure 5: Proof by contradiction.

In the half plane lying on the side of the tangent not containing the ellipse, there must be at least one more point ℓ from the set (since k is not on the hull). Taking points i, j, ℓ gives us a triangle with a larger perimeter. (Recall that an ellipse has the property that the sum of the distances to the foci remain constant as we move on the boundary.)

(d) The property we wish to prove is the following: let S be the set of vertices of a convex polygon. Let M be a max perimeter triangle. Let A be the max perimeter triangle rooted at a. The boundary of M must intersect *each* of the edges of A.

We will prove this property later; let us first see the algorithm based on this property. (We will assume that we are working with the convex hull of the set of points.) Pick a point $a \in S$ and find the max perimeter triangle rooted at a. This can be done in $O(n^2)$ time by brute force search. Let this triangle be $aa'a''$. The remaining vertices are partitioned into three chains. Pick the longest chain and consider its mid point b. Find a max perimeter triangle rooted at b.

Since the max perimeter triangle has to intersect both these triangles, we are left with two smaller sub-problems that we need to solve recursively. (We need to argue that we get 6 chains from these six points on the two triangles, and the max perimeter triangle has to hit three alternate chains – either the X chains or the Y chains.) There are more cases, but the scheme is identical to the scheme in part (b). We get a recurrence of the form

$$T(m) = 2T(\frac{m}{2}) + O(m^2)$$

which has a solution $T(m) = O(m^2)$.

The proof of the above property is a little tedious, and is omitted. I will try and include it in a later draft.

(4) In 1–dimension, any region in a k–th order Voronoi Diagram must consist of consecutive points. That is, if the points are indexed according to their sorted order $x_1, \ldots, x_n$, it must be the case that the k–order Voronoi region is of the form, $x_i, \ldots, x_{k+i-1}$. Hence, there are $n - k + 1$ such regions.

(5) One natural approach is to draw circles with radii equal to $\frac{d}{2}$ centered at the points. If two circles intersect (or touch) then there must be an edge between the two corresponding points. A sweep-line algorithm can be used to compute the "arrangement" of the set of circles in time $O((k+n)\log n)$ where k is the number of intersection points. This gives an output sensitive algorithm that works well when k is $o(n^2)$. (I have not worked out the details, but believe that this approach should work.)

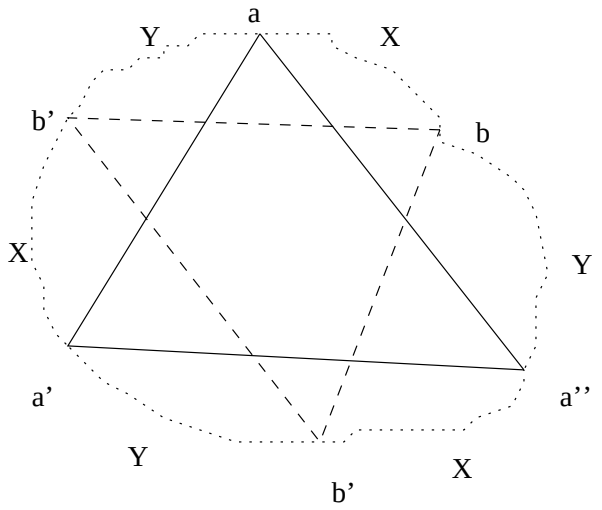

Figure 6:

Another approach that I thought of, is the following: compute a "grid" (or mesh) of size d (see Fig. 7). Each point p_i is contained in some cell of the grid. We can use the floor function to compute which cell each point belongs to. More specifically, we wish to compute for *each* cell, the list of points that belong to that cell.

Observe that if point p_i belongs to cell $C(p_i)$ then all the points to which it has edges (points within a circle of radius d) are in some cell that shares a boundary with $C(p_i)$ (or in the neighborhood of $C(p_i)$). Point p_i computes its distance to each such point p_j that belongs to a cell in its neighborhood, and decides if there should be an edge to p_j.

Consider two cells that are in each other's neighborhood. Suppose the number of points in each cell is x_i and x_j respectively. The work that is done by the algorithm for this pair of cells is $(x_i + x_j)^2$. (For the points in each of the two cells we have to compute the distance to each other point.) Notice that if we have x_i points in a single cell, then this cell has at least $\frac{1}{2}(\frac{x_i}{4})^2$ edges, since at least one fourth of the points in a cell are in a single quadrant. The same is true for the other cell. We need to "charge" the work done by the algorithm to the edges in the output. Notice that $(x_i + x_j)^2 \leq c\frac{1}{2}(\frac{x_i}{4})^2 + (\frac{x_j}{4})^2]$ (for some c) when x_i, x_j are large. We have accounted for the work done between a pair of cells by "charging" them to edges between points that are in the same cell. For any set of points in the same cell, the edges between them are charged at most constant times (since this cell only has a constant number of cells in its neighborhood). Thus the total work done in computing distances is $O(k)$ where k is the output size.

If implemented naively, the algorithm will take a huge amount of space (since we cannot bound the number of grid cells as a function of n, when d is small). However, empty cells need not be stored explicitly. We can store each non-empty cell in a data structure. For example, the set of non-empty cells in a single row can be stored in a 2-3 tree, so any cell can be accessed in $O(\log n)$ time. We can store a pointer from each cell to a list of the points in that cell. The underlying data structures can be implemented so that the entire algorithm runs in $O(k + n \log n)$ time.

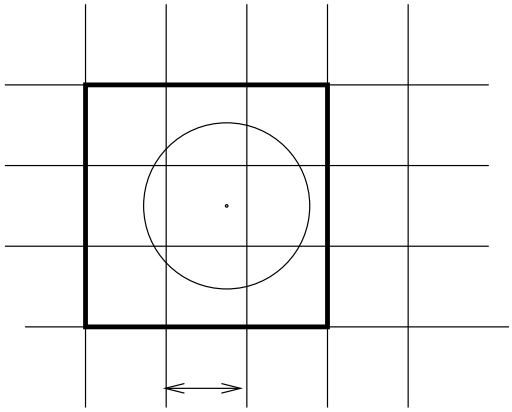

Figure 7: Constructing the grid.