Bryan Klingner
CS 274 Homework 5 Solutions
6 December 2006

[1] Voronoi diagrams (4 points). The worst-case complexity of a Voronoi diagram is $\Theta(n^2)$ in three dimensions, and $\Theta(n^{\lceil d/2 \rceil})$ in E^d.

(i) Prove that a planar cross-section of a d-dimensional Voronoi diagram—in other words, the planar subdivision formed by intersecting a 2-flat with a Voronoi diagram—has complexity no greater than $\mathcal{O}(n)$. Hint: what do you know about the properties of Voronoi cells?

Solution. In any dimension, a Voronoi diagram forms a convex subdivision of space. Any planar cross-section of a Voronoi diagram must therefore be a convex subdivision of the plane. The faces of this subdivision are induced by slicing through Voronoi cells in higher dimensions. Because these cells are all convex, we can get at most one face in the cross-section from each one. There are exactly n cells, so our cross section has no more than n faces. Vertices in the cross section correspond to the meeting of three (or more) hyperplane bisectors between Voronoi sites, so each vertex in the cross-section will have degree of at least three.

Armed with these two facts: (1) the cross section has no more than n faces and (2) each vertex of the cross section has degree of at least three, we can apply Euler's formula to show that the overall complexity is in $\mathcal{O}(n)$ just as we did for two-dimensional Voronoi diagrams. Euler's formula states that for connected, planar graphs with e edges, v vertices, and f faces

$$v - e + f = 2$$

Here, $f = n$. Add a vertex at infinity to connect any half-infinite edges of the cross-section to, so that we have fully-connected planar graph, and we get

$$(v + 1) - e + n = 2$$

Every edge has two vertices (it's endpoints). So, the sum of the degree of all the vertices we, we get twice the number of edges. Every vertex has a degree of at least three, so

$$2e \geq 3(v + 1)$$

Substituting, we get $v \leq 2n - 5$ and $e \leq 3n - 6$, so the overall complexity of the cross-section is in $\mathcal{O}(n)$. □

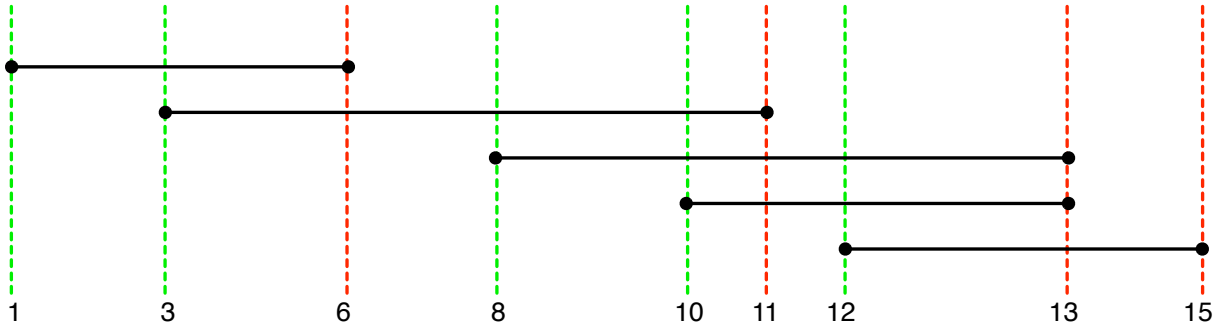
(ii) Does a planar cross-section of a d-dimensional Delaunay triangulation also always have a complexity in $\mathcal{O}(n)$? Explain.

Solution. No. Consider, for example, the three-dimensional point set that was given in class, with two sets of points along perpendicular lines that were separated from one another. In between the two lines were the $\mathcal{O}(n^2)$ tetrahedra that formed the only possible triangulation of the points. A cross section of this triangulation taken parallel to the two lines and between them would yield a triangular face for each of the $\mathcal{O}(n^2)$ tetrahedra, and hence have $\mathcal{O}(n^2)$ complexity.

[2] Overlapping intervals (7 points).

(i) Let S be a set of n intervals. Describe a data structure that uses $\mathcal{O}(n)$ storage and returns the *number* of intervals in S that intersect any query interval I in $\mathcal{O}(\log n)$ time. (Note that I didn't say "$\mathcal{O}(k+\log n)$ time.") Hint: I can think of two very different ways to do this, one of which modifies one of the data structures taught in class, but I can't think of a way that uses an interval tree.

Solution. One way of doing this would be to construct two one-dimensional arrays. The first would store, at each index i, the value of x for which i intervals have their right endpoint left of x. The other would store the value of x for which i intervals have their left endpoint to the right of x. It would look something like:



$i =$	1	2	3	4	5
i segments start right of	12	10	8	3	1
i segments end left of	6	10	13	13	15

Say the input query is $[x, x']$. Start by performing a binary search on the red array for x. This is the number of intervals l that end before the start of the query interval. Then, perform a query on the green array to find the number of intervals r that start after the end of the query interval. The number of intersecting intervals is $n - l - r$ (you would of course need to be careful about including or excluding enpoints of intervals properly, but I don't think this is a big deal). The storage required is $2n \in \mathcal{O}(n)$ for the two arrays and the query time for two binary searches is in $\mathcal{O}(\log n)$ time.

(ii) Explain how to use a two-dimensional layered range tree to answer the same counting query, also in $\mathcal{O}(\log n)$ time. (Assume you have a correct solution to Problem [3] of Homework 4; there's no need to explain that solution again. Observe that a two-dimensional range tree generally requires $\Theta(n \log n)$ space, so this isn't an answer to part (i).)

Solution. Again, let the query interval be $[x, x']$. An interval in S intersects the query interval unless both of its endpoints are less than x **or** both of its endpoints are greater than x'. In the two-dimensional range tree, store each interval in S by placing it's left endpoint as the x-value and it's right endpoint as the y-value. Now, perform two queries on the tree:

1. $l \Leftarrow$ number of points in $[-\infty, x] \times [-\infty, x]$ (intervals with both endpoints to the left of x).

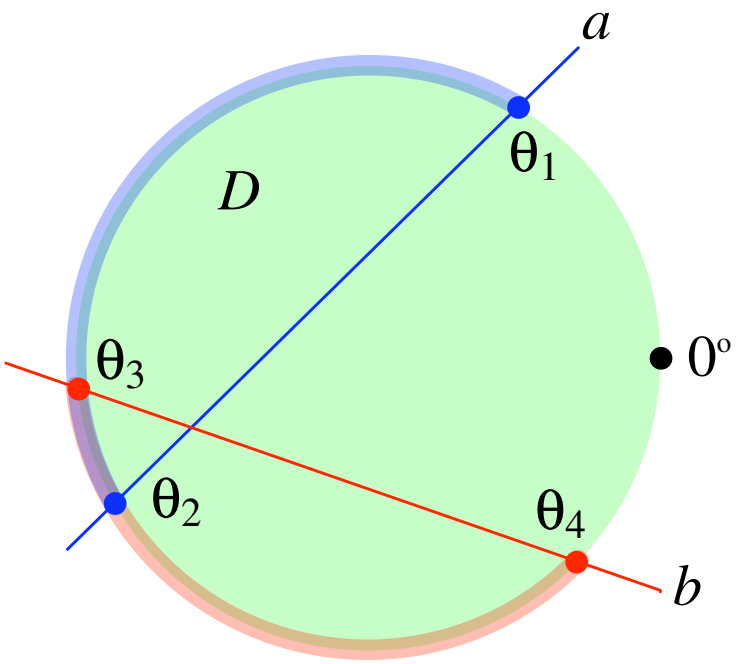
2. $r \Leftarrow$ number of points in $[x', \infty] \times [x', \infty]$ (intervals with both endpoints to the right of x').

All of the other intervals in S must intersect the query interval, so the number of intersecting intervals is $n - l - r$. Both queries take $\mathcal{O}(\log n)$ time, so the total time to get the count is in $\mathcal{O}(\log n)$.

(iii) Let D be a disk, and let L be a set of l lines in the plane. Describe an algorithm that returns, in $\mathcal{O}(l \log l)$ time, the number of pairs of lines that intersect each other inside D. For simplicity, you may assume that no two lines intersect each other precisely on the boundary of D. Hint: as best I can figure, this can be done with the range tree but not with the answer to part (i).

Solution. To help translate this into an overlapping intervals problem, let's label the boundary of the disk D with angles. Let the lexicographically maximum point in D be the $\theta = 0°$, with θ increasing as you move counterclockwise around the boundary of D.

Start by checking whether each line in L intersects the boundary of D in two places, meaning that it has some finite length inside D. For each line that does, record two angles θ_i and θ_j, $\theta_i < \theta_j$, of the intersection points. Also, record the number m of doubly-intersecting lines. This process takes constant time for each line, $\mathcal{O}(l)$ time overall. We can then encode each of these lines as an interval $[\theta_i, \theta_j]$. I'll refer to the first angle as the "left" endpoint and the second as the "right" endpoint. Here is an example of what two of these lines a and b (represented by $[\theta_1, \theta_2]$ and $[\theta_3, \theta_4]$, respectively) might look like:



We can see that in this example, a and b intersect and the intervals that represent them overlap. It's not true that any lines with intersecting intervals intersect in D; if one interval completely contains the other, there will be no intersection. Given a particular line l_1 encoded by $[\theta_i, \theta_j]$, there are four situations where another line l_2 encoded by $[\theta_k, \theta_l]$ will **not** intersect l_1:

1. $\theta_k < \theta_i$ and $\theta_l < \theta_i$. Both endpoints of l_2's interval are less than the right endpoint of l_1's interval.
2. $\theta_k > \theta_j$ and $\theta_l > \theta_j$. Both endpoints of l_2's interval are greater than the left endpoint of l_1's interval.
3. $\theta_k > \theta_i$ and $\theta_l < \theta_j$. l_1's interval completely contains l_2's interval.
4. $\theta_k < \theta_i$ and $\theta_l > \theta_j$. l_2's interval completely contains l_1's interval.

Create a two-dimensional layered range tree to store all of the intervals, storing the left endpoint in the x-coordinate and the right endpoint in the y-coordinate. Constructing the tree can be accomplished in $\mathcal{O}(l \log l)$ time. We can count the number of lines in each of these four classes for a query line (specified as an interval) by making four separate queries to the range tree, each counting (θ_k, θ_l) points within the appropriate query boxes. For each query line, this takes $\mathcal{O}(\log l)$ time.

We can then report the total number of lines in L that intersect the query line as m (the total number that intersect D twice) less the number returned for each of the four queries. Repeating this for every line gives us twice the total number of pairs of lines (every intersection will be reported twice) that intersect inside D. Performing all of the queries takes at worst $\mathcal{O}(l \log l)$ time, so the overall running time is in $\mathcal{O}(l \log l)$.

[3] Deterministic BSPs (4 points). Consider this deterministic algorithm for constructing a BSP tree for a set of n non-crossing line segments in the plane. Suppose you are deciding how to slice a cell C of the BSP subdivision. (When we start, C is the entire plane.) If there is a free split, take it. Otherwise, make a list of the segment endpoints (not fragment endpoints, just endpoints of the original segments) in C, and draw a vertical splitting line through the endpoint with median x-coordinate (breaking ties arbitrarily).

Prove that this algorithm always constructs a BSP tree with at most $\mathcal{O}(n \log n)$ fragments.

Solution. Imagine a particular segment s. Once s has been cut twice, the central fragment immediately causes a free split, protecting it from any further cuts. Because the cuts always occur at the median x-coordinate of vertices, there can no more than n segment endpoints each on the left and right sides of the central fragment (because there are two endpoints to each segment, we start with $2n$ endpoints).

Any further cuts to s will create another fragment that spans its cell, triggering another free split and reducing by half the number of segment endpoints on that side (left or right) that could cut s in the future. So, for every two new cuts (generating only two new fragments) on the segment (one left of the leftmost central fragment and one right of the rightmost central fragment), the number of potential future cutting endpoints is reduced by half. This repeated reduction by half translates into a total of $\mathcal{O}(\log n)$ possible cuts for s, meaning it can at most split into at most $\mathcal{O}(\log n)$ fragments. Because there are n segments, this means that overall we have $\mathcal{O}(n \log n)$ fragments. $\square$

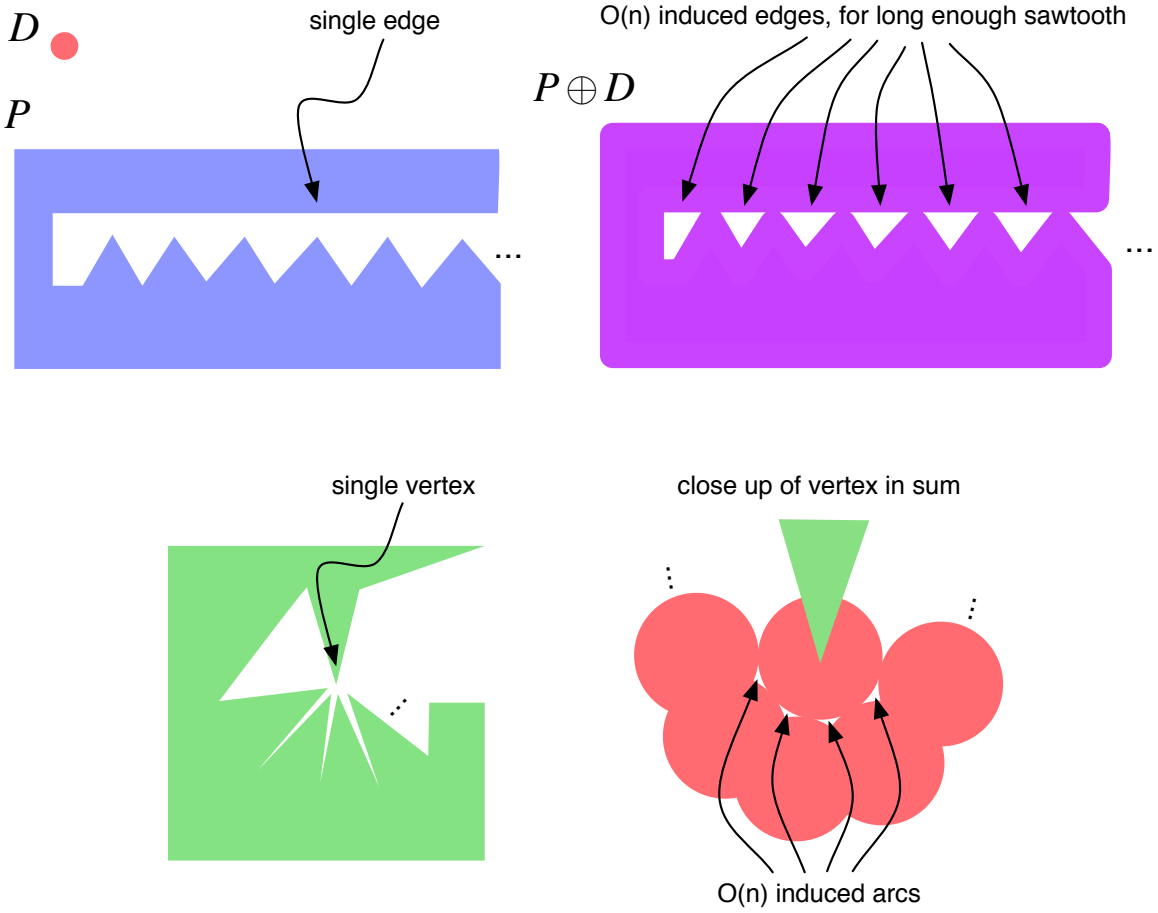
[4] Minkowski sums (7 points). Suppose we want to automatically machine a two-dimensional part by cutting away the region around it with a circular drill bit. Let D be a disk whose center is the origin and whose radius r is the radius of the drill bit. When machining a shape S, we may place the center of the drill bit at any point not in the interior of $S \oplus D$.

We wish to argue that for any n-vertex simple polygon P (not necessarily convex), the worst-case complexity of $P \oplus D$ is in $\mathcal{O}(n)$. The complexity is the total number of vertices, line segments, and circular arcs needed to represent $P \oplus D$. Line segments are induced by edges of P, offset sideways by the radius r of D; and circular arcs (of radius r) are induced by vertices of P.

Because $P \oplus D$ is a planar graph, it suffices to bound the number of vertices. There are three types of vertices that can appear: intersections between two line segments, insersections between two arcs, and intersections between a line segment and an arc.

(i) Demonstrate that a single edge of P can induce up to $\Theta(n)$ line segments in $P \oplus D$, and that a single vertex of P can induce up to $\Theta(n)$ circular arcs. (A couple of simple figures should suffice to illustrate these two points—no words are necessary.)

Solution.



(ii) An edge e of P can induce two sequences of line segments in $P \oplus D$, offset r to either side of e. Suppose two line segments of $P \oplus D$, s_1 and s_2, intersect at a vertex v. Argue that v terminates one of the sequences of line segments induced by some edge of P. Use this fact to argue that $P \oplus D$ can have only $\mathcal{O}(n)$ vertices formed by pairs of line segments.

Solution. For s_1 and s_2 to intersect, they each must be part of a sequence of segments induced by two different edges of P; call them e_1 and e_2. Suppose for the sake of contradiction that v does not terminate a sequence of segments induced by e_1 or e_2. This means that there are at least two more segments in $P \oplus D$, call them s_1' and s_2' that are collinear with s_1 and s_2 and further along e_1 and e_2. These two segments will have swapped position in relation to s_1 and s_2, implying that P is self-intersecting, but this is a contradiction because we know P to be simple.

Each input edge of P can give rise to at most two sequences of segments. Each sequence has at most one "stopping" vertex. So, we can bound the number of segment-segment intersection vertices of $P \oplus D$ with the same $\mathcal{O}(n)$ bound that applies to the number of edges in P.

(iii) Use the edges of a Voronoi diagram to argue that $P \oplus D$ can have only $\mathcal{O}(n)$ vertices formed by pairs of arcs.

Solution. Thanks for the idea! It's clear that only vertices of P can induce circular arcs. Imagine all of P's vertices as sites of a Voronoi diagram. Circles centered at the sites can only form vertices by arc-arc intersection on bisectors between sites of the Voronoi diagram.

There can never be more than two arc-arc intersection vertices formed for each edge of the Voronoi diagram, because every edge corresponds to a bisector between exactly two sites. Because a Voronoi diagram has $\mathcal{O}(n)$ edges in a diagram with n sites, the number of arc-arc intersection vertices must also be in $\mathcal{O}(n)$. Because the circular arcs in this case always have the same radius (that of D), there will actually be fewer arc-arc intersection vertices in $P \oplus D$, but this gives us our bound.

[5] Point-rectangle pairs (8 points). Let P be a set of m real numbers, and let R be a set of n intervals. By building an interval tree or a segment tree, we can report all pairs $\langle p \in P, r \in R \rangle$ such that $p \in r$ in $\mathcal{O}(k + (m+n)\log n)$ time (including preprocessing), where k is the number of such pairs.

Observe that k could be as large as $\Theta(mn)$. Suppose that instead of reporting pairs individually, we are allowed to report pairs of subsets of the form $\langle \{p_1, p_2, \ldots, p_i\}, \{r_1, r_2, \ldots, r_j\} \rangle$, meaning that each point $p_1 \ldots p_i$ lies in every interval $r_1 \ldots r_j$. These *subset pairs* sometimes make it possible to express the results in $o(k)$ space. Every intersecting real–interval pair $\langle p_i, r_j \rangle$ should appear in *exactly* one of the subset pairs in the output.

(i) Describe an algorithm that runs in $\mathcal{O}((m+n)\log n)$ time (including all preprocessing) and reports the real–interval pairs in $\mathcal{O}(\min\{k, (m+n)\log n\})$ space. (This is the amount of space taken by the subset pairs; the algorithm may use more.)

Solution. Start by building a segment tree T containing all of the intervals in R, taking $\mathcal{O}(n \log n)$ time. Then, perform m queries on T to determine what elementary interval each number in P lies in. The first time a point p is located in a particular elementary interval i, record all of the intervals R_i in R that are in each node on the search path down to i, and initialize a new subset pair $\langle \{p\}, R_i \rangle$. Any subsequent points found in this same elementary interval are added to this subset pair. Performing all of the queries takes a total of $\mathcal{O}(m \log n)$ time.

After all the subset pairs have been built up, report them. The total running time (construction + query) is $\mathcal{O}((m+n)\log n)$. How big is the output? It certainly can never be larger than $\mathcal{O}(k)$, the size of the output for ungrouped points. Because of the tree structure, we also know that a copy of every point and interval (of total size $\mathcal{O}(m+n)$) will appear at most $\mathcal{O}(\log n)$ times in the grouped output. If this $\mathcal{O}((m+n)\log n)$ size is smaller than $\mathcal{O}(k)$, then that will be the output size.

(ii) Improve your answer so that it runs in $\mathcal{O}(k' + m + n\log n)$ time, where k' is the size of the output. Hint: for each number in P, search the tree from bottom to top instead of top to bottom. To make this possible in the given time bound, you will need radix sort, and you will need to move items up the tree in batches rather than individually.

Solution. Take the solution to part (i). In order to get the needed speed up, we cannot query T for every point in P. Instead, sort the points in P in ascending order to form P' using radix sort, taking $\mathcal{O}(n)$ time. Start by finding the elementary interval i in T that contains the first point in P'. Now, walk through P' until you get to a number that is greater than the right endpoint of i. We now know all of the points from P that lie in the elementary interval i, so can straight away report the completed subset pair for this elementary interval.

We will have to perform one lookup on T for each elementary interval that contains a point from P, but this can never exceed $\mathcal{O}(n \log n)$ total lookup time asymptotically because there are only $\mathcal{O}(n)$ elementary intervals. So, including the tree construction and radix sort, we get the overall running time of $\mathcal{O}(k' + m + n\log n)$.

(iii) Let P be a set of m points in the plane, and let R be a set of n axis-parallel rectangles (possibly intersecting). We want to report all pairs $\langle p \in P, r \in R \rangle$ such that $p \in r$. Suppose there are k such pairs. Again we use subset pairs to save space and time. Describe an algorithm that reports the point–rectangle pairs in $k' \in \mathcal{O}(\min\{k, (m+n)\log^2 n\})$ space and runs in $\mathcal{O}(k' + m\log n + n\log^2 n)$ time.

You'll get **2 bonus points** if you can do it in $\mathcal{O}(k' + m + n \log^2 n)$ time. (I don't actually know if this is possible. It would take very careful organization of the batches of points to avoid getting charged $\log n$ for each point that is contained in few or no rectangles.)

Solution. We can accomplish this by using the multi-level segment/interval tree data structure discussed in class. We will be attempting to group the points in P into the rectangular "elementary boxes" formed by the elementary intervals of the segment tree keyed on the x intervals of R then cut horizontally by the intervals in the interval tree keyed on the y intervals. We can see that the output size here again can be as small as $\mathcal{O}(m + n)$ copied this time $\mathcal{O}(\log^2 n)$ times because of the second tree, giving us our bound on output size of $\mathcal{O}(\min\{k, (m + n) \log^2 n\})$.

As in part (ii), we use a radix sort on the x-coordinates of the points in P. We use the process from (ii) to group all of the points in P into "x-batches" that lie in the same elementary interval of the segment tree. Now, we need to split these batches into their elementary boxes by doing a lookup on each point in the interval tree to find which y interval it lies in. Once we know all of the points P_i that lie in a particular elementary box i, we can pair them with a list R_i of all rectangles from R that each elementary box lies in (this list can be obtained at no additional cost from the search paths through the segment and interval trees that led to a particular elementary box).

Construction of the multi-layer structure takes $\mathcal{O}(n \log^2 n)$ as shown in class. Querying the segment tree to create the x-batches takes at worst $\mathcal{O}(n \log n)$ as in part (ii) and so does not affect the asymptotic bound. We must, however, perform m queries to the second-level interval tree, each of which takes $\mathcal{O}(m \log)$ time, adding an asymptotic factor of $\mathcal{O}(m \log n)$. Finally, we must output all of the finished subset pairs, taking $\mathcal{O}(k')$ time, for an overall running time of $\mathcal{O}(k' + m \log n + n \log^2 n)$.