6.854 **Dan Ports**

Advanced Algorithms drkp@mit.edu

2004/11/16 Collaborators: {sarahl,pramook}

Problem Set 10

Problem 1:

a) Let S be a set of n axis-parallel rectangles in the plane, and $Q = [x : x'] \times [y : y']$ be a query rectangle. We map the problem of finding all rectangles in S that are contained in Q to a four-dimensional orthogonal range searching problem.

Define a set $\mathfrak{S}$ of 4-tuples such that $\mathfrak{S}$ contains $\mathfrak{s} = \langle x_1,\ x_2,\ y_1,\ y_2 \rangle$ if S contains the rectangle with corners at $\langle x_1,\ y_1 \rangle$ and $\langle x_2,\ y_2 \rangle$. Then we can perform a range search on $\mathfrak{S}$ over the range $\langle x,\ x,\ y,\ y \rangle$ to $\langle x',\ x',\ y',\ y' \rangle$. This finds all points that satisfy the following constraints

$$\begin{aligned} x \leq x_1 \leq x' \\ x \leq x_2 \leq x' \\ y \leq y_1 \leq y' \\ y \leq y_2 \leq y' \end{aligned}$$

which is precisely the definition of a rectangle being contained in the query rectangle.

The 4-dimensional orthogonal range query can be performed using the natural extension of the 2-dimensional orthogonal range query data structure. As before, we create the primary binary search tree on the first coordinate, and create a secondary binary search tree for each internal node in the primary tree containing all the elements in that subtree ordered by their second coordinate. In the same manner, we create a tertiary binary search tree for each internal node in the secondary tree containing all the elements in that subtree ordered by their third coordinate, and a quaternary binary search tree for each internal node in the tertiary tree containing all the elements in that subtree ordered by their fourth coordinate.

The analysis of required time and space is similar to the 2-d case. Each point is stored once in the primary tree, and in one secondary tree for each of the $\log n$ ancestors in the primary tree (since the trees are balanced). For each secondary tree it is in, it is stored in $\log n$ tertiary trees for the same reason, so $\log^2 n$ tertiary trees total. Likewise, it is stored in $\log n$ quaternary trees per tertiary tree, so $\log^3 n$ quaternary trees. So the total space requirement is $O\left(n + \log^3 n\right)$. The required time is $O\left(\log^4 n + k\right)$, since it requires searching $\log^3 n$ quaternary trees in $\log n$ time each, plus providing the output (the natural extension of the argument for the 2-d case).

b) Consider a polygon p. We can find the smallest bounding rectangle for p by taking the minimum and maximum values for both x and y. For a polygon to be in the query rectangle, it is both necessary and sufficient for this rectangle to be contained in the query rectangle. So we can replace each polygon with its bounding rectangle, thereby reducing this problem to the previous part.

Problem 2:

Consider the problem of finding all (horizontal and vertical) segments that intersect a query rectangle $Q = \langle X_1,\ Y_1,\ X_2,\ Y_2 \rangle$. A solution can be expressed as the set of all horizontal line segments that intersect the vertical line through X_1 or X_2 and lie between Y_1 and Y_2 on the vertical axis, and the corresponding set of vertical line segments (replacing "horizontal" with "vertical" and vice versa above). We give the procedure for identifying the intersecting horizontal line segments (the former case); the latter case is precisely symmetric.

Consider a horizontal segment s_i. It is defined by its left and right endpoints x_i and x_i' and its y coordinate, y_i. Our problem is to find all s_i such that $Y_1 \le y_i \le Y_2$ and the interval $[x_i, x_i']$ contains X_1 or X_2. To accomplish this, we create a data structure made up of a primary balanced BST ordered by y_i (as in range search), and associate with each internal node an interval tree of the x-coordinate intervals of the line segments in that node's subtree.

The space required for the primary tree is $O(n)$. It is a balanced tree, so each node has $O(\log n)$ ancestors. Each segment has a representation in each of the interval trees corresponding to its ancestors in the primary search tree, and there are n segments, so the space required for the interval trees is $O(n \log n)$.

Once we have built this data structure, we can perform queries for a given query rectangle by walking through the primary tree and querying all the interval trees corresponding to nodes between the top and bottom coordinates Y_1 and Y_2 of the query rectangle. As with range search, the size of the set S of interval trees queried is $O(\log n)$. For $s \in S$, define k_s to be the number of result segments found in s. The time required to query the interval tree s will be $O(\log n + k_s)$. Then the total time required is

$$\sum_{s \in S} O(\log n + k_s) = \sum_{s \in S} O(\log n) + \sum_{s \in S} O(k_s) = O\left(\log^2 n + k\right)$$

Problem 3:

Given a set Ξ of n walls in the plane and a player position p, we give an algorithm for finding the set of walls that are (at least partially) visible from the player position. For each $\xi \in \Xi$, define ξ^1 and ξ^2 to be the endpoints of the wall, such that ξ^1 is the first vertex encountered on a counterclockwise sweep.

We create a list of all vertices of interest (the ξ_i^1 and ξ_i^2), and sort it in order of their angle relative to the player position. There are $2n$ such vertices, so $O(n \log n)$ time is required. We sweep a half-line around the player position, while maintaining a balanced binary search tree of the "active" line walls (those that are intersected by the sweep line), ordered by their relative distance from the player. At every position of the sweep line, the wall with the minimum distance from the player is visible. Note that since even though the distance between a wall and the player changes as the line is swept, because walls do not intersect, the relative ordering of their distance from the player does not change except by adding or removing new walls to or from the tree; thus, the ordering in the tree remains valid. Note also that the active tree can only contain at most n walls, so since it is a balanced tree all operations on it can be performed in $O(\log n)$ time.

We begin by choosing some arbitrary angle and drawing the sweep line. Our initialization step is to iterate through the set of walls Ξ, testing whether each wall intersects the sweep line at that angle and if so, inserting it into the active tree (ordered by relative distance from p at that sweep line angle). This requires iterating over a set of n walls, and per wall requires at most $O(\log n)$ time, so it can be performed in $O(n \log n)$ time.

We then sweep the line counterclockwise, over the sorted list of vertices of interest. Let x be the next vertex processed. If x is the "first" endpoint seen of a wall (i.e. the wall appears counterclockwise from the endpoint from the player's perspective), we insert it into the tree. If it is the other endpoint, we remove that wall from the tree. At each vertex, we find the active wall that is closest to the player point (the minimum of the active tree), and mark that wall visible. Once we have swept through all the vertices, all visible walls will be marked, and we can simply iterate through the list of walls and output those marked visible.

The runtime is $O(n \log n)$ because there are $O(n)$ vertices, and $O(\log n)$ processing time per vertex since the tree is a balanced tree.

Problem 4:

We give an randomized incremental algorithm for finding the intersection of a set of half-planes. We maintain an incremental intersection as a convex polygon (perhaps including points at infinity) as well as a bipartite conflict graph representing vertices of the current incremental intersection and half-planes that have yet to be intersected. We begin with no half-planes intersected, so the incremental intersection is the entire plane, and choose a random ordering of the half-planes. For each half-plane l to be intersected, we identify from the conflict graph the set R of conflicting vertices from the conflict graph (if there are any). Denote these as a subinterval of vertices $v_i \cdots v_j$, where $v_1 \cdots v_n$ is the polygon representing the current incremental intersection. Because the polygon is convex, any intersections with the half-plane l must occur in the line segments connecting $(v_{i-1} \to v_i)$ and $(v_j \to v_{j+1})$. So we simply need to find the intersections of l with these two segments (call these points x_1 and x_2), and add them to the incremental intersection polygon. We next update the conflict graph, checking for every half-plane p that conflicts with one of the removed vertices in R whether p conflicts with x_1 or x_2, and add the appropriate edges to the conflict graph if so. Finally, we remove R from the polygon, and remove l and R from the conflict graph.

We now analyze the expected runtime of this algorithm. Note that each half-plane intersection creates at most two vertices in the incremental intersection polygon, so $O(n)$ time is used creating vertices. Hence, $O(n)$ vertices are removed from the polygon, since a vertex must be first be created if it is to be removed. So we need only consider the amount of time required to maintain the conflict graph. Every time we add a half-plane to the intersection, we check for each removed vertex in the intersection polygon all the half-planes that conflict with it. We add an edge to the conflict graph if it conflicts with x_1 or x_2. So the time required to update the conflict graph is linearly proportional to the number of half-planes conflicting with x_1 and x_2. (If a half-plane conflicts with neither x_1 or x_2, but did conflict with some removed vertex, it no longer conflicts with any part of the polygon and can be disregarded; the time required to check these half-planes is negligible since this happens at most once per half-plane).

To place a bound on this time, we use backwards analysis of the randomized incremental construction. We start with the completed intersection polygon, and remove random edges from it, adding back the half-planes that created those edges. Suppose we have an intersection polygon P_i with i edges and we remove some edge e from it. Call the set of half-planes that have not been intersected yet P_i^c (it is essentially the complement of P_i, since each half-plane corresponds to an edge in P_i once intersected.) We wish to know the expected number of half-planes conflicting with the endpoints of e; this gives the expected runtime, since they were the x_1 and x_2 when it was added. This is the same in the backwards analysis, since the order in which half-planes are chosen (and hence edges are created) in the algorithm is chosen randomly. This is at most $\frac{2}{i}$ times the total number of conflicts between vertices in P_i and half-planes yet to be intersected (call this number of conflicts $\mathcal{F}_i$). We can write this as a summation over the half-planes in P_i^c:

$$\frac{2}{i}\mathcal{F}_i = \frac{2}{i}\sum_{l \in P_i^c} \mathcal{F}_{i,l}$$

where $\mathcal{F}_{i,l}$ is the random variable equal to the number of conflicts between half-plane l and the vertices in the polygon P_i.

Since the polygon P_i contains i edges, there are $n-i$ half-planes left to be intersected. Note that if we let l_{i+1} be the next half-plane processed, since it is chosen randomly, we take expectations and obtain the identity

$$\mathrm{E}\left[\mathcal{F}_{i,l_{i+1}}\right] = \frac{1}{n-i}\sum_{l \in P_i^c} \mathrm{E}\left[\mathcal{F}_{i,l}\right] = \frac{1}{n-i}\sum_{l \in P_i^c} \mathcal{F}_{i,l}$$

Applying this to the equation above, we find that

$$\frac{2}{i}\mathcal{F}_i = \frac{2(n-i)}{i}\mathrm{E}\left[\mathcal{F}_{i,l_{i+1}}\right]$$

Recall that $\digamma_{i,l_{i+1}}$ is defined to be the number of conflicts encountered when intersecting the $i+1$th half-plane processed with the incremental intersection polygon after i steps. So $\digamma_{i,l_{i+1}}$ vertices are removed from the incremental intersection polygon on the $i+1$th intersection step. We can use this to place a bound on the sum over all i $\frac{2(n-i)}{i}\mathrm{E}\left[\digamma_{i,l_{i+1}}\right]$. Call this sum X.

Consider a vertex v that is created at time $\aleph_v$ and destroyed at time Ω_v, and Ω_v must be strictly greater than $\aleph_v$, since a vertex is never created and destroyed at the same time. So changing variables from i to v (summed over *all* vertices v)

$$X = \sum_i \frac{2(n-i)}{i}\mathrm{E}\left[\digamma_{i,l_{i+1}}\right] = 2\sum_v \frac{n-(\Omega_v+1)}{\Omega_v - 1} \leq 2\sum_v \frac{n-\aleph_v}{\aleph_v}$$

Changing variables back to i, we find that

$$X \leq 2\sum_i \frac{n-i}{i}2$$

since at most 2 vertices are created in intersection step i. Then

$$X \leq 4\sum_i \frac{n-i}{i} = 4n\sum_i \frac{1}{i} - 4\sum_i 1 = O\left(n\log n\right) - O\left(n\right) = O\left(n\log n\right)$$

by the harmonic bound.

We now claim that X is the total running time due to updating the conflict graph. Recall that X was defined as the sum over all i of $\frac{2(n-i)}{i}$ multiplied by the number of vertices removed on the $i+1$th intersection step. But we showed earlier that the expected number of half-planes conflicting with the randomly-chosen edge e removed after i (forwards) steps was at most $\frac{2}{i}\digamma_i \leq \frac{2(n-i)}{i}\mathrm{E}\left[\digamma_{i,l_{i+1}}\right]$ which is precisely $\frac{2(n-i)}{i}$ multiplied by the number of vertices removed on the $i+1$th intersection step. So X is the sum over all steps i of the expected time required to update the conflict graph at step i, which is the total running time due to updating the conflict graph. Since $X = O\left(n\log n\right)$, the algorithm's expected running time is $O\left(n\log n\right)$.

Problem 5:

Submitted separately.

Problem 6:

About 16 hours.