CS623 Winter 2012 \ Assignment #3

Ariel Stolerman

ID 12331542

1)

a.

When using a 2-D kd-tree for a partial query, it is equivalent to applying a query of the form $\{x\} \times (-\infty, \infty)$ or $(-\infty, \infty) \times \{y\}$ – i.e. a fixed value for one coordinate and all possible values for the other. Half the times (when we're at the level of the fixed point) we have a recursive call only on one side of the current split line, thus the original query time-complexity analysis doesn't apply here (those steps actually consist $O(\lg n)$ together – a simply binary search). However, the other half of the times need to account for the entire range of possible values, thus we will always go down both the left and the right subtree. This fits to the original analysis, resulting with $O\left(\sqrt{n} + k\right)$ query time (the analysis addressed one vertical line of the bounding box, and applied the same logic on the other vertical line + the 2 horizontal lines; now we simply apply it only on one pair, either that of the vertical or the horizontal lines, concluding to the same time complexity).

Note: when allowing multiple points to have the same x or y values, we simply impose an order between them in their composite representation as mentioned in the book. This should not affect the complexity here.

b.

When using a 2-D range tree to answer a partial query, it can be done in $O(\lg n + k)$ time as follows. In both cases where we are given a fixed x or fixed y value, and need to return all points with those values in the respective coordinate, we again use the composite representation mentioned in the book that imposes a total ordering in any of the coordinates.

If we are given a fixed x value, and need to return all points with that x value (and any y value), using the composite representation we obtain a range $[x_{min}: x_{max}]$. We find in $O(\lg n)$ both (which, in case no more than 1 point holds that value, is a single leaf), and then traverse over the leaves in between them, returning all points along the way (stored in those leaves).

If we are given a fixed y value, we do the same on the y-tree pointed to from the root of the x-tree, which is basically all points sorted by their y values. Again, we find $[y_{min} : y_{max}]$ (or simply y) in $O(\lg n)$ time and traverse over the leaves in between them.

In both cases finding the range ends takes $O(\lg n)$ time, and traversing over the points between them is $O(k)$ (as known and proved for the original query algorithm using range trees). Note that the composite representation of the points to impose a total order between their coordinates does not increase complexity asymptotically. The total running time in both cases is $O(\lg n + k)$.

c.

Following is a data structure with linear storage and $O(\lg n + k)$ time for 2-D partial queries:

We hold 2 balanced binary search trees of all points, one sorted with respect to the x values and one with respect to the y values. If we have repetitions of coordinate values, instead of using the composite representation, we simply hold a bin of all points with the corresponding coordinate value at each leaf. For instance, in the x-tree, all points with the same x-value will be held in a bin (list) under the leaf corresponding to that value.

Applying a query is as follows: we select the tree corresponding to the given fixed coordinate value, search its leaf in that tree, and if found - return all points in the bin under that leaf.

The storage is linear as we simply hold 2 trees with $O(n)$ leaves each, and exactly n points in total across all bins for each of the trees. As for the query time, searching the correct bin is $O(\lg n)$, and traversing and returning all values in that bin is $O(k)$ – concluding together to $O(\lg n + k)$, as required.

d.

Following is a proof that with a d-dimensional kd-tree we can solve a d-dimensional partial match query in $O\left(n^{1-s/d} + k\right)$ time, where s ($s < d$) is the number of specified coordinates:

We follow the proof of the query running time for 2-d kd-tree. There we had the recurrence $Q(n) = 2 + 2Q\left(\frac{n}{4}\right) = O(\sqrt{n})$, where $Q(n)$ is the number of regions a vertical/horizontal edge of the query rectangle intersects. Generalizing the recurrence to d-dimensions we get $Q(n) = 2^{d-1} + 2^{d-1}Q(n/2^d)$, as instead of looking at a 1-D edge, we look at a $(d-1)$-dimensional hyperplane. In this case we cross d levels down the tree and the hyperplane intersects 2^{d-1} regions along the way, and for each we call recursively with the number of points in that region: $\frac{n}{2^d}$. Eventually the total region intersections for all $(d-1)$-dimensional "sides" of the query d-dimensional rectangle is $2dQ(n)$, and using the Master Theorem we get that it equals $O\left(2dn^{1-1/d}\right) = O\left(n^{1-1/d}\right)$.

Looking at the recursive query algorithm (for 2-D in the book), at each level that is not a leaf, if the left/right regions are not fully contained in the query rectangle, we continue to the left/right sides recursively. In the case s out of the d dimensions are set to some value rather than a range, that intersection check can be true only for one of the regions, never for two. Therefore they contribute to the coefficient of $Q(n/2^d)$ only 1 instead of 2. On the other hand, all the rest $d-s$ coordinates, since we need to return **all** their possible values, will always check recursively both sides, thus contributing 2 to the coefficient of $Q(n/2^d)$. Combining the two types of coordinates concludes to $2^{d-s}Q(n/2^d)$. By the Master Theorem:

$Q(n) = n^{\lg_{2^d} 2^{d-s}}$ where: $\lg_{2^d} 2^{d-s} = (d-s)\lg_{2^d} 2 = (d-s)\cdot\frac{1}{\lg_2 2^d} = \frac{d-s}{d} = 1 - s/d \Rightarrow Q(n) = O\left(n^{1-s/d}\right)$.

Adding $O(k)$ for the subtree-reporting procedures we get a total running time of $O\left(n^{1-s/d} + k\right)$, as required.

e.

Following is a description of a data structure that uses linear storage and solves a d-dimensional partial query in $O(\lg n + k)$ time:

Similar to question 2, we hold 2^d balanced binary search trees with bins in their leaves. Each of the search trees corresponds to a certain subset of coordinates and holds the points sorted with respect to the concatenation of all those

points together. For instance, in 5-D, the tree that corresponds to coordinates 1,2,4 will be sorted by those coordinates and, for instance, the leaf that corresponds to the values (x_1, x_2, x_4) will hold a bin of all points with those values in the 1st, 2nd and 4th coordinates, respectively.

When a query is applied, we choose the tree that addresses the given fixed points, search the correct leaf and return all points in its bin.

We can hold pointers to all those trees in an array of size 2^d, such that each entry codes in a binary representation which coordinates are included in the sorting of that tree, so for a given query we find the required tree in constant time. Since each tree is a binary search tree with sorting of at most dn distinct values, each such tree is of height $O(\lg dn) = O(\lg d + \lg n) = O(\lg n)$ (we can address d as a constant); traversing over the points in the bin (if found) is $O(k)$. In total we have a $O(\lg n + k)$ query time, as required.

As for storage, since we have 2^d trees with $O(nd)$ leaves, then each tree required $O(nd)$ space, concluding to a total of $O(d2^d n)$ space, which is linear to n, as required.

2)

a.

When performing a query of a point (a, b) as a range $[a{:}\,a] \times [b{:}\,b]$ in a 2-D kd-tree, at any recursive step (where the node v checked is not a leaf), there are two checks for each of the left and right subtree. Of course none of their corresponding regions will ever be fully contained in the query region (since it is a single point) unless it is a single point, and the intersection check will hold true for exactly one of them – the region where the point resides (either left or right, but not both, for odd depths; either bottom or top, but not both, for even depths). Therefore any such query will end up with $O(\lg n)$ recursive calls. Furthermore the $O(k)$ part, since $k = 1$, is $O(1)$. The total concludes to a query time of $O(\lg n)$.

b.

When using a range tree the running time for finding a single point (a, b) (again, represented a range $[a{:}\,a] \times [b{:}\,b]$) is $O(\lg n)$. If all n points have distinct x coordinates then the only node that will have its associated structure checked is the leaf that holds a point with $x = a$. It will take $O(\lg n)$ to get to that leaf, and since it's a leaf, only the point it holds is checked (no associated structure), therefore the total query time is $O(\lg n)$.

If we allow non-distinct coordinate values, using the composite representation of the points described in the book (where each point (x, y) is represented by $\big((x|y), (y|x)\big)$, and an ordering is defined by $(a|b) < (c|d) \Leftrightarrow a < c \vee (a = c \wedge b < d)$) yields the same result, as there will be only one leaf with x value equals to the composite $(a|b)$.

3)

Given a planar subdivision S with n vertices and edges and a query point q, following is an $O(n)$ algorithm to compute the face in S that contains q:

The idea of the following algorithm is essentially to go over the linear number of faces (using the fact each edge is part of at most 2 faces), ruling them out one by one until either we get an answer or left with no faces.

- Go over all edges and check if q is contained in the face associated with them. When checking a face, mark that edge so it will not be checked again in the future.
- For each edge consider both halves.
- Return the face that contains q, i.e. the face that all its surrounding edges have q to the left of them (i.e. q is contained in the sub-plane to the left of the line extending from that edge, with respect to the edge's direction).

We can obtain all half-edges in linear time, and obtain these edges in two lists (such that no two halves of the same edge are in the same list, and if two edges are consecutive, i.e. one is the next() of the other, they are in the same list). Every time we find a half-edge that q is not to its left, we remove all half edges from that list (i.e. the half-edges that surround that face). This way every edge is iterated at most twice – once for initial check and at most once if it needs to be removed from the list. Therefore the total running time is linear to the number of edges, i.e. $O(n)$ (note that according to Euler's formula we know that any way the number of edges in any planar subdivision is linear with respect to the number of vertices, so the restriction to n edges is irrelevant).

4)

Given a convex polygon P with n vertices represented by an array A of its chain in sorted order, following is a $O(\lg n)$ algorithm to find whether a query point q resides inside P:

We do a simple binary search: start with the points $A[0]$ and $A\left[\frac{n}{2}\right]$, cross a line between them and check if q resides to the left or to the right of that line. If to the left, continue recursively on the convex polygon with the chain $0 \to 1 \to \cdots \to \frac{n}{2} \to 0$, otherwise with the one with the chain $\frac{n}{2} \to \frac{n}{2}+1 \to \cdots \to 0 \to \frac{n}{2}$. When go to a small enough polygon (with some constant number of edges), check if that polygon contains q (with orientation tests for all edges of that polygon), and return the answer.

Clearly at each recursive step we conduct an $O(1)$ operations, since access to each array is constant, as are the line creation and orientation tests (note that the algorithm can receive the chain as an argument, with i and j indicating the current start and end index, to avoid building the chain for the next level). We conclude with $O(1)$ operations for the base case. At each step we cut in half the number of vertices in the polygon we check next, and we know for sure q will not be contained in the other half we disregard. Therefore the algorithm is correct and the total running time is $O(\lg n)$, as required.

5)

Let S be a set of non-crossing segments in the plane, and $s \notin S$ a segment that crosses no segment in S. Let Δ be a trapezoid in the trapezoidal map of S, $\mathcal{T}(S)$. Following is a proof that Δ is also a trapezoid of $\mathcal{T}(S \cup \{s\}) \Leftrightarrow s$ does not intersects the interior of Δ:

First, assume s does not intersect the interior of Δ. Following the correctness of the randomized incremental construction of the trapezoidal map $\mathcal{T}(S)$ in the book, we know $\mathcal{T}(S \cup \{s\})$ can be built upon $\mathcal{T}(S)$. When introducing s to the search

structure, since it does not intersect Δ we know the leaf corresponding to Δ in the search structure will not be removed, thus Δ will continue to be a trapezoid in $\mathcal{T}(S \cup \{s\})$.

Now assume s intersects the interior of Δ, either one of s's endpoints resides inside Δ or it completely splits Δ into 2 trapezoids. In the first case, the leaf corresponding to Δ is replaced by a structure with 3 trapezoids (of depth 2, i.e. 3 levels), and in the second case Δ is replaced by the two new trapezoids below and above s. In either case, Δ is removed from the structure, i.e. does not exist in the trapezoidal map $\mathcal{T}(S \cup \{s\})$.

6)

Following is a proof using a plane sweep argument that the trapezoidal map of n line segments in general position has at most $3n + 1$ trapezoids:

We can hold a queue of all segment endpoints, marked as either left or right, and in addition address the left edge of the bounding box as a segment (simply a vertical segment, and we allow that in the general case), so its 2 endpoints are also added to the queue. All endpoints are stored sorted from left to right, and for points on the same vertical line – from top to bottom (so vertical segments have "$-\infty$" slope). We can then cross a vertical sweep line from $-\infty$ to $+\infty$ that stops at each point in the queue. The cases don't differ that much from the original proof of the $3n + 1$ bound:

- If all points are just consecutive pairs of left and right endpoints (of the same segment), e.g. $l_1, r_1, l_2, r_2, \dots, l_k, r_k$ $(k \geq 1)$, then they all define together one vertical line, thus consist together the left edge of one trapezoid – to the right of that vertical line.

  For the rest of the cases, when there's at least one endpoint without its pair, we can ignore any vertical segments endpoints, as that one unpaired endpoint can be addressed as the one defining the trapezoids to the right. Therefore any of the following cases will look at endpoints that are NOT of vertical segments.
- When the sweep line intersects a left endpoint, if it is the topmost endpoint, it defines the left edges of 2 trapezoids – those above and below the segment of that left endpoint. For any left endpoint after it (below it; again, not part of a vertical segment) defines a left edge of one trapezoid – the one below its segment (as the one above is defined by the previous left endpoint intersecting the sweep line).
- When the sweep line intersects a right endpoint, if there are left endpoints intersecting the sweep line, they will overtake the trapezoid count, and the right endpoints can be ignored. Otherwise, if we only have ≥ 1 right endpoints only intersecting the sweep line, they all define together the left edge of 1 trapezoid – the one to the right of the line.

Therefore, in the best case scenario, when there are no vertical segments (except for the left edge of the bounding box) and every event of the sweep line is on its own (the only point intersecting the sweep line at that phase), all left endpoints define 2 trapezoids each, all right endpoints define 1 trapezoid each and the left edge of the bounding box defines 1 trapezoid. In total we get at most $3n + 1$ trapezoids, as required.