

## COMPUTATIONAL GEOMETRY

### Homework Set # 6 – Solution Notes

(1). [20 points] *Problem 5.4(a)-(d), BKOS.*

(a). In order to show that 2-dimensional kd-tree can answer partial match queries in  $O(\sqrt{n} + k)$ , it can be argued that any vertical (horizontal) line intersects  $O(\sqrt{n})$  regions of a 2-dimensional kd-tree. The idea behind this line of reasoning is this, nodes of a kd-tree are visited iff their corresponding regions in  $\mathbb{R}^2$  are intersected by a specified query box. For our purposes, we need only focus on a given coordinate whereby our query box is reduced to a line (i.e., we wish to find all points sharing a particular coordinate value). Then the question is, how many regions (where these regions can be thought of as points) does the query line hit? The text argues this by way of a recurrence relation.

Let  $Q(n)$  denote the number of intersected regions in a kd-tree storing  $n$  points whose root contains a vertical splitting line.

$$Q(n) = 2 + 2Q\left(\frac{n}{4}\right), \text{ if } n > 1 \text{ (and } O(1), \text{ if } n = 1)$$

(For an in-depth explanation of the foregoing recurrence, please direct your attention to the text, page 106.)

This recurrence solves to  $Q(n) = O(\sqrt{n})$ . Thus, one can detect and report all points satisfying our partial match query in  $O(\sqrt{n} + k)$  time, where  $k$  is the number of reported answers.

(b). Using a 2-dimensional range tree, partial match queries can be solved as follows:

$$\text{Given: } S(n) = O(n \log n), T(n) = O(n \log n)$$

Partial match queries require  $O(\log n)$  time to locate all of the desired information for a particular  $x$ - (or  $y$ -) coordinate. Such a bound holds because the process of locating the desired coordinate requires walking through a balanced binary search tree on  $x$  (or  $y$ ). Then, in  $O(n)$  the given coordinate's subtree can be traversed. This allows one to report all points satisfying the partial match in  $O(k)$  where  $k \leq n$ .

Overall, the required query time is  $O(\log n + k)$ , where  $k$  is the number of reported answers.

(c). A balanced binary search tree is a data structure that uses linear storage and can solve 2-dimensional partial match queries in  $O(\log n + k)$  (linear arrays yield similar results). In either case,  $P = \{p_1, p_2, \dots, p_n\}$  can be stored with respect to one of the two linearly independent directions of  $\mathbb{R}^2$ . This is identical to storing points in a 1-dimensional kd-tree where every leaf and internal node uses  $O(1)$  storage, implying that the overall storage space is  $O(n)$  (since any such tree stores  $O(n)$  points). Once the nodes are sorted by  $x$  (or  $y$ ), bucket all points in the plane sharing  $x$ - (or  $y$ -) coordinates. As a result, this will permit  $O(\log n)$  lookups since any one leaf can be found in  $O(\log n)$  time (once again by binary search) and contains  $O(1)$  points.

(d). Now, dealing with the general case, it can be shown that a  $d$ -dimensional kd-tree can solve a  $d$ -dimensional partial match in query  $O(n^{1-\frac{s}{d}} + k)$  time. This stems from generalizing the recurrence relation associated with part (a). To start,  $s$  is defined to be the number of specified coordinates (with  $s < d$ ). This implies that  $d - s$  coordinates are left unspecified and thus variable.

For simplicity, let's look at the case where  $s = 2$  and  $d = 3$ . Now, imagine a 3-dimensional space where two coordinate values have been specified, for instance,  $x = a$  and  $y = b$ . This gives rise to a line with variable height (where  $z$  denotes height). In order to locate all points satisfying the given line in 3D, splits on  $x$ ,  $y$  and  $z$  are required. When splitting on  $x$  and then  $y$ , one needs to determine the side of splitting plane that contains  $x = a$  and subsequently  $y = b$  in order to see which path along the kd-tree must be taken (yielding a total of 2 comparisons). Finally, a split on  $z$  may force 2 comparisons, since  $z$ 's value is variable over the entire line where  $x = a$ ,  $y = b$  and can potentially lie on both sides of a splitting plane when splitting on  $z$ .

We now apply this reasoning to the general case of  $d$  dimensions, with coordinates  $(x_1, x_2, \dots, x_d)$ .

Let  $Q(n)$  denote the number of intersected regions for a partial match query in a kd-tree storing  $n$  points whose root contains a splitting hyperplane orthogonal to the  $x_1$ -coordinate. The total query time will then be  $O(k + Q(n))$ , where  $k$  is the number of points reported by the query.

For the case in which  $n = 1$  the query can be answered in time proportional to the number of dimensions (i.e.,  $O(d)$ , where  $d$  is considered to be constant).

The more interesting case arises when  $n > 1$  and one must enumerate all nodes of the  $d$ -dimensional kd-tree that are visited during the partial match query. This results in the recurrence

$$Q(n) = 1 + 2(d - 1) + (d - s) + 2^{d-s}Q\left(\frac{n}{2^d}\right)$$

where,

- (i) the first term counts the root of the kd-tree;
- (ii) the second term counts the paths where each node following the root visits one child;
- (iii) the third term accounts for the additional splits that may arise due to the unknown (variable) coordinate values;
- (iv) the final term recursively solves  $2^{d-s}$  problems each of size  $\frac{n}{2^d}$  that are observed when the  $d$ -th level of the  $d$ -dimensional kd-tree is explored (when again the nodes correspond to a splitting hyperplane orthogonal to the  $x_1$ -coordinate).

The recurrence gives  $Q(n) = O(n^{1-\frac{s}{d}})$ , and a query time for reporting of  $O(n^{1-\frac{s}{d}} + k)$  as a worst-case bound. This is precisely what one was asked to show.

**(2).** [20 points] *Problem 5.5, BKOS.*

(a). Let all points are located on line  $y = x$ , and the query triangle is a little below line  $y = x$  with the long edge on line  $y = x - \varepsilon$ . See Figure 1.

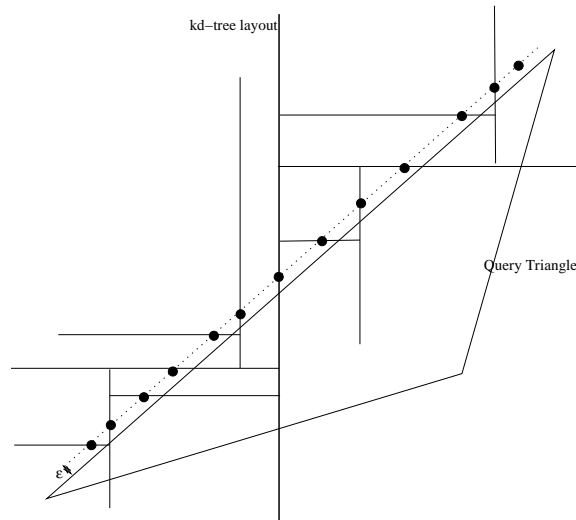


Figure 1: kd-tree and query triangle

Since line  $y = x - \varepsilon$  will intersect with all regions corresponding to all nodes in kd-tree, the query time will be  $O(n)$  even if there is no answer reported. This is one of the worst cases.

(b). Choose four coordinate axes  $\ell_1: x$ ,  $\ell_2: y$ ,  $\ell_3: y = x$ , and  $\ell_4: y = -x$ .

At the root, we split the set of points into two subsets of roughly the same size by  $\ell_1$ . At the children of the root, the partition is based on second line  $\ell_2$ , at nodes at depth two on the third line  $\ell_3$ , and at node at depth three on the fourth line  $\ell_4$ . At depth four we start all over again.

Thus, this 4-dimension kd-tree will answer a query in time  $O(n^{3/4} + k)$ .

(c). Notice that the query triangle has only three edges from the set of four lines  $\{\ell_1, \ell_2, \ell_3, \ell_4\}$ , i. e., it is composed of  $\{\ell_1, \ell_2, \ell_3\}$  or  $\{\ell_1, \ell_2, \ell_4\}$  or  $\{\ell_1, \ell_3, \ell_4\}$  or  $\{\ell_2, \ell_3, \ell_4\}$ . For all possible combinations, we create a kd-tree in each case. Given a query triangle, we first decide which three lines determine this triangle in constant time. Then in the corresponding 3-dimensional kd-tree we answer the query in time  $O(n^{2/3} + k)$ .

**(3).** [20 points] *Problem 5.11, BKOS.*

The simple idea is to do a sweep, with a vertical line  $\ell$ . The SLS stores (sorted by  $y$ ) the horizontal segments intersected by  $\ell$ . Events occur whenever  $\ell$  encounters a left endpoint (insert the horizontal segment into the SLS), a right endpoint (delete the horizontal segment from the SLS), or a vertical segment (do a 1D range count query to determine how many elements of the SLS lie between the top and bottom  $y$ -coordinates of the vertical segment – this count can be obtained in  $O(\log n)$  time by doing two binary searches). There are  $2n + m$  events, each requiring time  $O(\log(m + n))$ .

(4). [20 points] *Problem 5.13, BKOS: In many applications one wants to do range searching among objects other than points.*

a Let  $S$  be a set of  $n$  axis-parallel rectangles in the plane. We want to be able to report all rectangles in  $S$  that are completely contained in a query rectangle  $[x : x'] \times [y : y']$ . Describe a data structure for this problem that uses  $O(n \log^3 n)$  storage and has  $O(\log^4 n + k)$  query time, where  $k$  is the number of reported answers. Hint: Transform the problem to an orthogonal range searching problem in some higher-dimensional space.

b Let  $P$  consist of a set of  $n$  polygons in the plane. Again describe a data structure that uses  $O(n \log^3 n)$  storage and has  $O(\log^4 n + k)$  query time to report all polygons completely contained in the query rectangle, where  $k$  is the number of reported answers.

c Improve the query time of your solutions (both for a. and b.) to  $O(\log^3 n + k)$ .

a For each rectangle with left bottom corner  $(x_1, y_1)$  and right top corner  $(x_2, y_2)$ , it can be mapped to a point  $(x_1, y_1, x_2, y_2)$  in  $\mathbb{R}^4$  in constant time. The query rectangle is mapped to a new 4D orthogonal range  $[x : x'] \times [y : y'] \times [x : x'] \times [y : y']$ . A rectangle is completely contained in the query rectangle if and only if the respective point is contained in the 4D orthogonal range. Therefore, this problem is mapped to a 4D range query problem in linear time and can be solved in  $O(\log^4 n + k)$  query time with  $O(n \log^3 n)$  storage using standard range tree algorithm.

b A polygon is completely contained in the query rectangle if and only if the bounding rectangle of the polygon is completely contained in the query rectangle. Suppose the bounding box of a polygon can be computed in constant time, this problem can be mapped to the problem a in linear time. Thus this problem can be solved in  $O(\log^4 n + k)$  query time with  $O(n \log^3 n)$  storage using standard range tree algorithm.

c If fractional cascading range query algorithm is used, the problem of a and b can be solved in  $O(\log^3 n + k)$  time.

(5). [20 points] *Let  $S$  be a set of  $n$  axis-aligned rectangles in the plane (they are allowed to intersect). We want to preprocess  $S$  into a “small” data structure so that if I give you a query point,  $q = (x, y)$ , you can quickly (in logarithmic time, ideally) tell me how many rectangles contain point  $q$ .*

(a). *Describe a way to do this with worst-case  $O(n^2 \log n)$  preprocessing time, using  $O(n^2)$  worst-case space, so that the query time is logarithmic.*

(We can do better than the  $O(n^2 \log n)$  time bound asked!)

Build the arrangement of the  $n$  rectangles, e.g., by sweeping the plane with the Bentley-Ottmann algorithm. This is worst-case time  $O(n^2 \log n)$ , but, because the rectangles are axis-aligned, the sweep is actually optimal time,  $O(k + n \log n)$  (which also follows if we use more general methods like Chazelle-Edelsbrunner or Balaban), where  $k$  is the size of the output (the number of crossings). Of course,  $k$  is worst-case quadratic in  $n$ . Note that in the same time bound we actually build a “vertical trapezoidalization” of the arrangement, with every face decomposed into rectangles. (This is important in case some faces have  $\Omega(n)$  “floating holes”, which need to be decomposed for point location preprocessing, below.)

We store the resulting arrangement in a DCEL of size  $O(k + n)$ . Do a depth-first search of the DCEL to label the faces according to the “depth of overlap” (i.e., how many rectangles cover each face – it goes up or down by 1 (in the nondegenerate case) each time we cross from one face to another).

Now preprocess this arrangement in time  $O(k+n)$  for point location (since each face is simply connected). We answer a query in time  $O(\log n)$  by doing a point location query and reporting the depth of overlap for the resulting face containing the query point  $q$ .

(b). Assume now that the  $n$  rectangles are translates of one common rectangle. (i.e., they all have the same height and width) Now give a way to do the problem with nearly-linear (e.g.,  $O(n \log n)$  or  $O(n \log^2 n)$ ) worst-case preprocessing time and space, so that the query time is logarithmic.

If all rectangles are translates of one common rectangle  $R$ , then we can simply “shrink” the rectangles to points (e.g., their upper left corners) and think of “growing” the query point  $q$  to a rectangle,  $R_q$ , of shape  $R$ , with  $q$  at the bottom right corner. Then the rectangles we need to report are exactly those whose upper left corners lie in the query rectangle  $qR$ . Thus, this is nothing but orthogonal range search, which is easily done in preprocessing time/space  $O(n \log n)$  and query time  $O(\log n)$  (assuming we use fractional cascading).

(6). [20 points] *Problem 6.10, BKOS.*

xxx To be written. I have some prior scribed student solutions. See problem 9, hw3, 2004.

(7). [20 points] *Construct the data structure  $\mathcal{D}$  (see Chapter 6) for the following set of line segments. The numerical labels indicate the order in which they are to be inserted. Also, the segment labelled “ $i$ ” is  $s_i = p_i q_i$ . Show the structure after each insertion.*

