ICS 621: Analysis of Algorithms Fall 2019

Lecture 11

Prof. Nodari Sitchinava *Scribe: Jeremy Ong, Sushil Shrestha*

1 Geometric Data Structures

1.1 Range Queries

The problem statement for the range queries is that given a bunch of points in high dimension, retrieve all the points bounded by certain boundary.

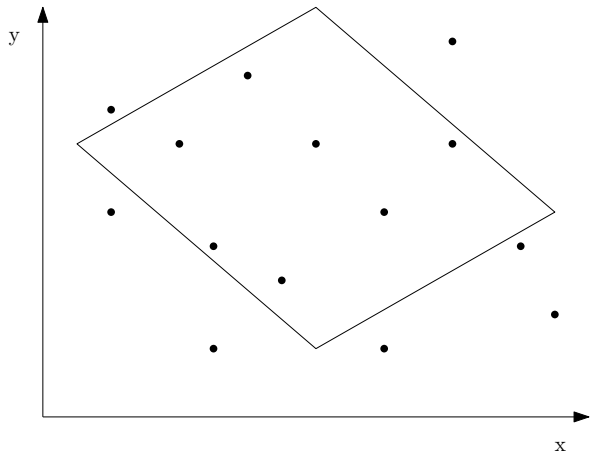

Figure 1: Problem statement: points within the bound

One way to find all the points bounded by a given boundary is to go through all the points linearly and report a point if it lies within the boundary. That would cost linear time $O(n)$ to find the points, if it takes constant time to perform each check. We can have better retrieval time which is less than $O(n)$ and for that let's look at some of the available algorithms.

1.2 1D range query

Before we move on to higher dimensions, let's start with one dimensional range queries.

Given a sequence of number(S) as an input and $[X_L, X_R]$ as a range of query, the 1D range query should output all the points that lies within that range ($\{X : X_L \leq X \geq X_R \; \forall X \in S\}$)

Here we assume that preprocessing of the input is allowed and the data is sorted as a step in preprocessing. We will use a Binary Search Tree(T) to store the sorted data. As a first step we will search our BST for X_L and X_R and find the leaf nodes for X_L and X_R. Then we can report all the nodes in between those leaf nodes as our output that lies in between X_L and X_R. To accomplish that we have to find split node in our BST where paths to X_L and X_R diverges in our tree. Then we can traverse the left and right tree of the split vertex and report the subtrees that lie within the specified range.

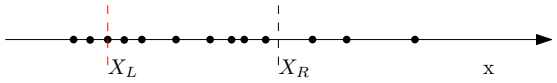

Figure 2: Single dimensional range query

Algorithm 1

```
1:  procedure 1DRangeQuery(T.root, [x_L, x_R])
2:      v = T.root
3:      while (v ≠ leaf) and (v.key ≤ x_L or v.key > x_R) do        ▷ find the splitting node
4:          if x_R ≤ v.key then
5:              v ← v.left
6:          else
7:              v ← v.right
8:          end if
9:      end while                                                   ▷ v will be our splitting node
10:     v_split ← v                                                 ▷ Traverse the left path and report points
11:     v ← v_split.left
12:     while v ≠ leaf do
13:         if x_L ≤ v.key then
14:             REPORT(v.right)
15:             v ← v.left
16:         else
17:             v ← v.right
18:         end if
19:     end while
20:     v ← v_split.right                                           ▷ Traverse the right path and report points
21:     while v ≠ leaf do
22:         if x_R ≥ v.key then
23:             REPORT(v.left)
24:             v ← v.right
25:         else
26:             v ← v.left
27:         end if
28:     end while
29:     if v.key = x_R or v.key = x_L then
30:         REPORT(v)
31:     end if
32: end procedure
```

Algorithm 1 first finds the V_{split} for a given range. Then it traverses the path from V_{split} vertex to x_L and reports all the right subtrees hanging off the path. For the path from the V_{split} to x_R, similar process is followed to report all left subtrees hanging off the path.

Now for the analysis of the algorithm, the space complexity for the binary search tree is $O(n)$ and it will

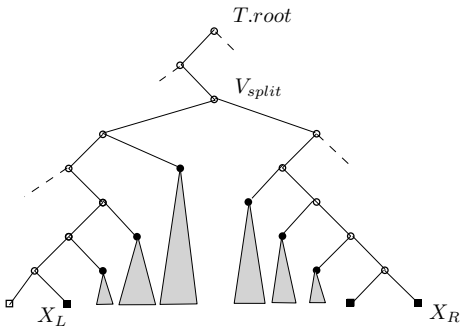

Figure 3: Selected Subtree

take $O(n \log n)$ to build the binary search tree. For the query part, REPORT(T) will take linear time with the number of items needed to be reported i.e. $O(k)$ where k is number of points reported. And searching the leaf nodes for X_L and X_R requires $O(\log n)$ time. So total query time is $O(\log n + k)$.

1.3 2D orthogonal range queries: k-d trees

Given a sequence of points S on XY plane and bounds $[X_L, X_R] \times [Y_L, Y_R]$, our goal is to find all points that lie within that the rectangular range.

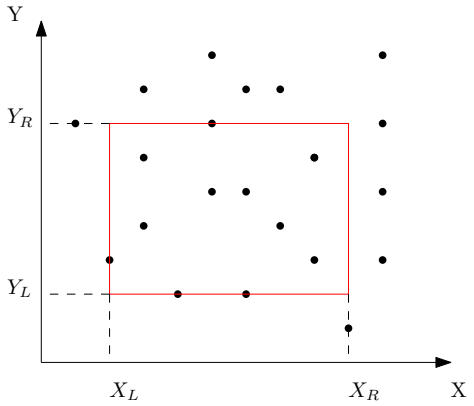

Figure 4: 2D range query

Similar to 1D range query, we can use binary search tree to find the points within the range of the bound. To create a binary search tree from the given set of points, we split sequence of points S by a vertical line(l_1) into two subsets with nearly equal number of points. Then we will use the line (l_1) as the key to our node in binary search tree. The left subtree of this node will store all the points on and left of the line l_1 and right subtree of the node will store the points right of the line l_1. Next we equally divide the points on the left of the line by a horizontal line and store horizontal line(l_2) as the key for left child of l_1. Then store all the points above the horizontal line as right child of l_2 and store all the points below and on the horizontal line as left child of l_2. We will perform similar division on the points to the right of the original vertical line l_1. We now have four sections on our plane, we will continue to divide those sections recursively (first by vertical line and then by horizontal line) until each section only contains single point.

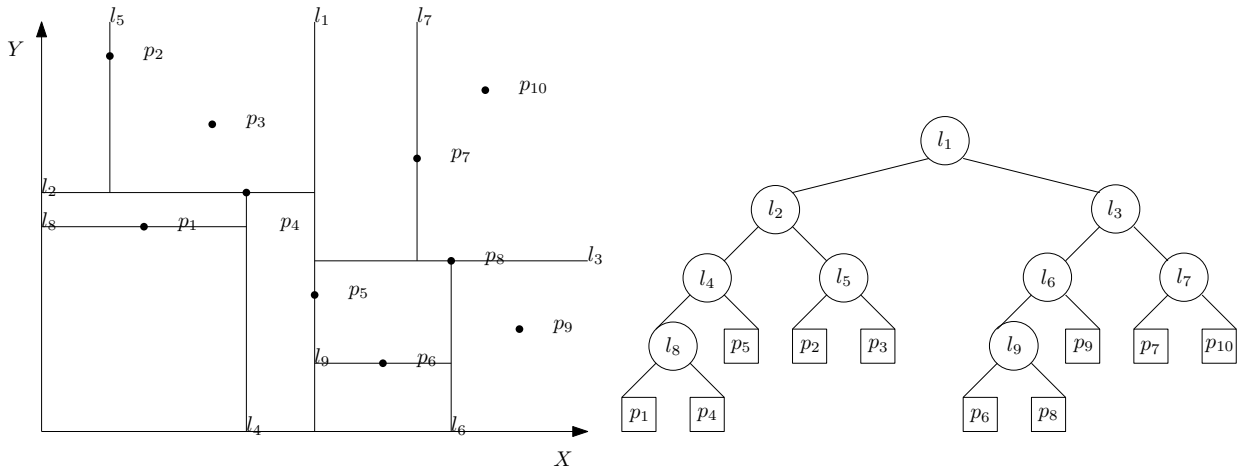

Figure 5: Constructing k-d tree by partitioning the coordinates

Algorithm 2

procedure BUILDKDTREE(S)
  Find the median X_{mid} among S
  $(S_L, S_R) \leftarrow$ PARTITION(S, X_{mid}) ▷ Partition S around X_{mid}
  Find the median Y_{mid} among S_L
  $(S'_L, S''_L) \leftarrow$ PARTITION(S_L, Y_{mid}) ▷ Partition S_L around Y_{mid}
  Find the median Y'_{mid} among S_R
  $(S'_R, S''_R) \leftarrow$ PARTITION(S_R, Y'_{mid}) ▷ Partition S_R around Y'_{mid}
  Create 3 nodes v, u, w with 3 partition lines $x = X_{mid}$, $y = Y_{mid}$, and $y = Y'_{mid}$, respectively
  $v.left \leftarrow u$; $v.right \leftarrow w$
  $u.left =$ BUILDKDTREE(S'_L) ▷ Recursively construct k-d trees on the four quandrants
  $u.right =$ BUILDKDTREE(S''_L)
  $w.left =$ BUILDKDTREE(S'_R)
  $w.right =$ BUILDKDTREE(S''_R)
  return v
end procedure

For the analysis of the algorithm, the cost for building the k-d tree is the time required to find the median among the data points partitioned around the median plus the recursive calls of the function. Given a list of points, finding median of data points can be achieved in linear time. Hence the cost of building a k-d tree can be written as:

$$T(n) = O(n) + 4T(n/4)$$

Using master theorem, the cost solves to $O(n \log n)$

While building the k-d tree we partitioned the sequence of points into different regions and then we created nodes for every bounding line. The nodes in the k-d tree represents a bounding box enclosing certain nodes based on their position. So to perform a range query on the sequence of points modeled as k-d tree, if the range includes full region represented by some node then the total subtree is reported back. If the regions are partially inside the query range, we recurse. At the leaf points are checked directly against the boundaries and are partially reported.

Algorithm 3

```
1: procedure SearchKdTree(v, R)                    ▷ R is a range query [x_L, x_R] × [y_L, y_R]
2:     if v is leaf and v lies in the region then
3:         Report(v)
4:     else if region(v.left) is fully contained in R then
5:         Report(v.left)
6:     else if region(v.left) intersects R then
7:         SearchKdTree(v.left, R)
8:     else if region(v.right) is fully contained in R then
9:         Report(v.right)
10:    else if region(v.right) intersects R then
11:        SearchKdTree(v.right, R)
12:    end if
13: end procedure
```

For the analysis of the algorithm, report takes $O(k)$ time where k is the number of reported points. The reporting linear time will account for line 4, 5, 8 and 9 in Algorithm 3. For lines 6, 7, 10 and 11, we have to find number of times the boundary of the range intersects region(v). To find out the number of such regions, we can find number of regions intersected by any vertical line and use it as upper bound of regions intersected by left and right edge of query rectangle. We can similarly find the upper bound of regions intersected by top and bottom edge of query rectangle.

Let $Q(n)$ be the number of intersected regions in a k-d tree. If we consider first vertical split and second horizontal split across the set of points, we have 4 regions with $n/4$ points each. And two of four regions correspond to intersected region so we can count the number of intersected regions recursively. The recurrence relation can be written as:

$$\begin{aligned} Q(n) &= 2 + 2 \cdot Q(n/4) \\ &= O(\sqrt{n}) \end{aligned}$$

The amount of time it takes for 2D range query is $O(\sqrt{n} + k)$.

2 Range Trees

But can we do better than $O(\sqrt{n} + k)$? Using a **range tree** instead of a kd-tree, we can get $O(\log^2 n + k)$, which is quite a bit better than our previous upper bound.

First, let us consider that in a 1-D query, reporting a specific subtree of nodes rooted at v (Report(v)) is equivalent to performing a call of 1DRangeQuery$(v, [x'_L = -\infty, x'_R = \infty])$ which is a 0-sided query on those points. Then if we were to only consider the results from this call that are within a specific y range,this is equivalent to a 2-D query with range $[y_L, y_R]$. To do so, we could store the x and y-coordinates in each leaf node, then check the y-coordinate of each one in the subtree. However, this would necessitate examining every leaf in the subtree, as y-coordinates are only in heap order, not sorted.

So how can we do this quickly? If, at each node of T, we store the points of that subtree in a BST but sorted by y-coordinate (which we call $v.ytree$), we would only need to perform a binary search to get our the first node in our y range, then iterate across the points until we reach a point with a y-coordinate greater than our range. This would only require examining $O(k_v)$ points, where k_v is the number of points in the subtree to be reported, plus $O(\log n)$ work for the binary search, giving us $O(k_v + \log n)$, the same as a 1-D query.

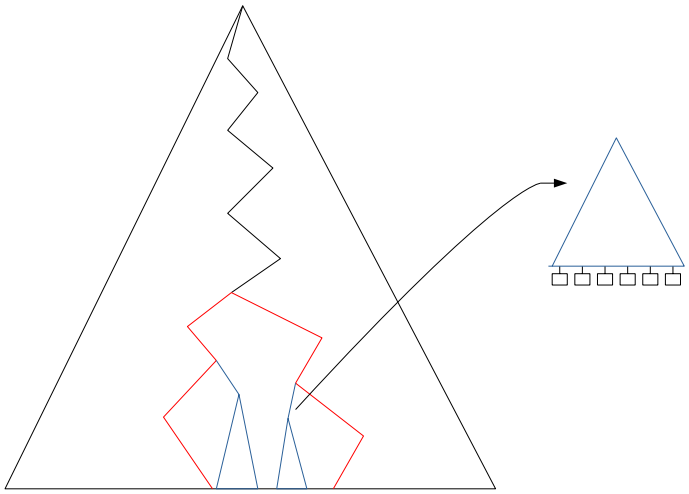

Figure 6: A 2-D range tree. Once a subtree is to be reported, instead go to the y-sorted BST stored at the node.

Algorithm 4

```
1:  procedure 2DRangeQuery(T.root, [x_L, x_R], [y_L, y_R])
2:      for each v in the traversal from the splitting point down to x_L do
3:          if the next move goes left then
4:              1DRangeQuery([y_L, y_R], v.right.ytree)
5:          end if
6:      end for
7:      for each v in the traversal from the splitting point down to x_R do
8:          if the next move goes right then
9:              1DRangeQuery([y_L, y_R], v.left.ytree)
10:         end if
11:     end for
12: end procedure
```

The analysis of this algorithm is similar to that of a 1-D range query. $O(\log n)$ nodes are examined to traverse down T on the left and right paths. At each node v that is reached, we may need to report the leaves of that subtree, requiring $O(k_v + \log n)$ time for each node. This gives us total runtime of:

$$\sum_v O(k_v + \log n)$$

We know that the summation of k_v over all v is k, the total number of nodes to be reported. Since there are $O(\log n)$ nodes v to be traversed down and possibly reported, this gives us $O(k + \log^2 n)$ total runtime.

How much space do we need to store all of these extra y-sorted BSTs? First, we must realize that at each level of T, the y-sorted BSTs stored at that level contain exactly the leaves of the entire tree, T. So if we have $O(n)$ extra storage at each level of T, and T has $O(\log n)$ levels, we have the space requirement of $O(n \log n)$.

It should be noted that using **fractional cascading**, we can reduce this runtime to $O(k + \log n)$. However, we will not cover this technique here.

3 3-D Queries

When we only need to know whether one of our search parameters is below (or above) a certain value, we can perform a 3-sided query. This is equivalent to 2DRangeQuery($T.root, [x_L, x_R], [-\infty, y_R]$). Trivially, we could answer this with a 4-sided query algorithm, but doing so in $O(k + \log n)$ time with range trees (with fractional cascading) requires $O(n \log n)$ extra space. When we only need a 3-sided query, we can do better.

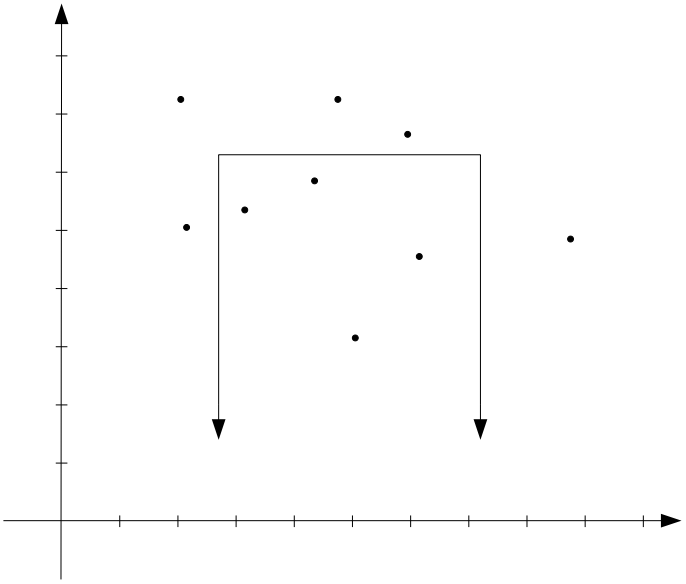

Figure 7: A 3-sided range query for all points between two x values and below an upper y value.

By using a **priority search tree** (PST) (similar to a treap in concept), we can reduce space usage without increasing our time bound. Like a treap, a priority search tree is a binary tree with priorities in heap order. For a 2-D query, we can store the x-coordinates of our points as the keys and the y-coordinates as priorities, with each node in the tree representing a point. When splitting by the x-coordinate of the current node, the resulting tree would have keys in BST order and is known as a **Cartesian tree**.

However, performing this splitting by the x-coordinate of the current node can result in an unbalanced tree, possibly with a depth of n. So instead, we can split each level based on the median y-coordinate of points, so that subtrees at each level are (roughly) equal in size.

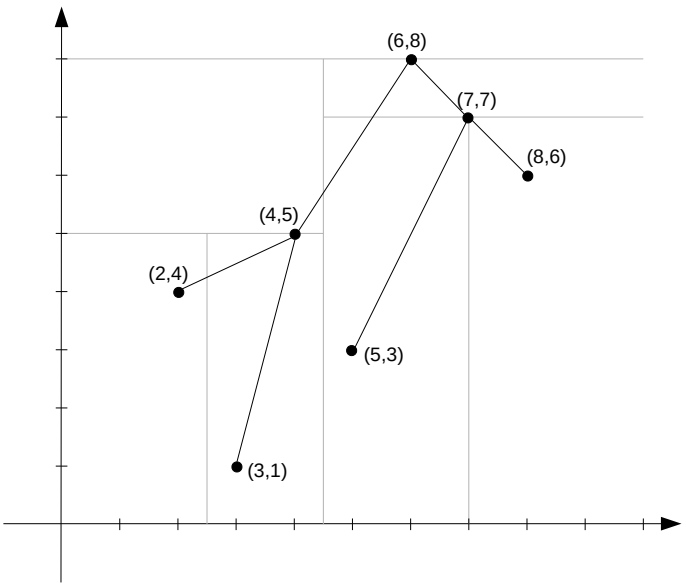

Figure 8: A priority search tree, split by x_{mid}, with the vertical grey lines showing the splits.

The following algorithm constructs such a PST:

Algorithm 5

```
1: procedure BUILD-PST(S)
2:     Create node v that is the point in S with the greatest y-coordinate
3:     x_mid ← median among x-coordinates of points in S
4:     v.mid ← x_mid
5:     (S_L, S_R) ← PARTITION(S, x_mid)
6:     v.left ← BUILD-PST(S_L)
7:     v.right ← BUILD-PST(S_R)
8:     return v
9: end procedure
```

Where PARTITION(S, x_{mid}) partitions the points in S into those to the left of x_{mid} and those to the right. Finding the median and a call to PARTITION takes $O(n)$ time, giving us a recurrence relation of:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Which solves to $O(n \log n)$ to build our priority search tree. Since we are only storing n nodes, each using a constant amount of space, we have $O(n)$ total space used, much better than the $O(n \log n)$ of a range tree.

When performing a 3-sided query on the PST, we traverse down the tree to the splitting point. Then on the left half of the split, traverse down and report the right subtree of any nodes if going to the left. Traversing the right half of the split is symmetrical.

Since T is in max-heap order with respect to y-coordinates, reporting all nodes in a subtree rooted at v with a y-coordinate greater than some y_L can be done by recursing down on each child node with y-coordinate greater than y_L.

Algorithm 6

```
1: procedure REPORT-PST(v, y_L)
2:     if v ≠ NIL AND v.y ≥ y_L then
3:         Output v
4:         REPORT-PST(v.left, y_L)
5:         REPORT-PST(v.right, y_L)
6:     end if
7: end procedure
```

Algorithm 7

```
1: procedure 3-SIDED-QUERY(T, [x_L, x_R], y_L)
2:     Search T for the splitting point v_split on x-coordinates, reporting nodes on the path that fall within [x_L, x_R] × [y_L, ∞]
3:     v ← v_split
4:     while v ≠ NIL do
5:         if v.y ≥ y_L then
6:             output v
7:         end if
8:         if x_L ≤ v.mid then
9:             REPORT-PST(v.right, y_L)
10:            v ← v.left
11:        else
12:            v ← v.right
13:        end if
14:    end while
15:    while v ≠ NIL do
16:        if v.y ≥ y_L then
17:            output v
18:        end if
19:        if x_R ≥ v.mid then
20:            REPORT-PST(v.left, y_L)
21:            v ← v.right
22:        else
23:            v ← v.left
24:        end if
25:    end while
26: end procedure
```

So we have a total time to traverse down the tree in both directions of $O(\log n)$. Additionally, the total time for reporting all nodes is $O(k)$, giving a total runtime of $O(\log n + k)$.