CPS234 Computational Geometry **September 22nd, 2005**

Lecture 8: Orthogonal Range Searching

Lecturer: Pankaj K. Agarwal *Scribe: Mason F. Matthews*

8.1 Range Searching

The general problem of range searching is as follows: Given input S, a set of n points in $\mathbb{R}^d$, preprocess the data into a data structure (if necessary) and answer a series of queries about the data. Queries include a d-dimensional region r and may be any of the following types:

- Emptiness: Does $r \cap S = \emptyset$? The output will be boolean.
- Reporting: Output $r \cap S$. The output will be a subset of S.
- Counting: Compute $|r \cap S|$. The output will be an integer.

Since solving the reporting problem provides an answer to the other two queries, this will be our focus. In the broadest sense, these regions can have any form, but if it is assumed that region inclusion can be computed in constant time, the naive algorithm of checking every point can be run in $O(n)$ time for each query.

In the database community, these problems are known as "indexing" problems, and any preprocessing of the input into a data structure is referred to as "building an index."

8.2 Orthogonal Range Searching

Suppose instead that our queries are limited to d-dimensional axis-aligned rectangles, input as the cross-product of 1-D intervals (i.e. $r = [a_1, b_1] \times [a_2, b_2] \times ... \times [a_d, b_d]$). Queries of this form are referred to as *rectangular range queries*, or more commonly in computational geometry as *orthogonal range queries*. With this restriction in place, the naive query time is still $O(n)$, but now a wide range of data structures can be used to speed the process.

While the topic of dynamic input sets is a rich area, insert and delete operations will not be covered in this lecture.

8.2.1 1-Dimensional Case

Consider first the simple 1-dimensional version of this problem. Here, the input S is a set of points on the real line, and a query rectangle is simply an interval $[a, b]$. See an example of a query of this type in Figure

8.1. The simple solution to this problem can be shown to be optimal with the decision tree model. To respond to queries in $O(\lg n + k)$ time, simply preprocess the points by sorting them into an array. Preprocessing requires $O(n \lg n)$ time and $O(n)$ space, but queries can be solved with the following method: first, perform a binary search for the first item larger than a ($O(\lg n)$); second, output every item in sequence until you reach a point larger than b ($O(k)$).

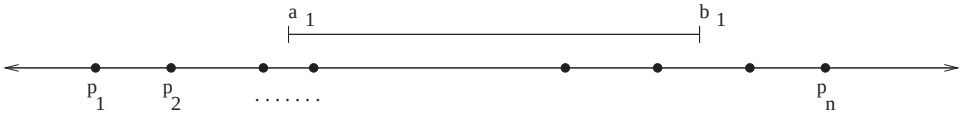


Figure 8.1: One-dimensional Range Search

8.2.2 2-Dimensional Case

Now suppose that our queries are 2-dimensional axis-aligned rectangles, input in the form $r = [a_1, b_1] \times [a_2, b_2]$. Ideally, we would like to maintain a query time of $O(\lg n + k)$. As mentioned earlier, an upper bound on our performance is still $O(n)$. Two data structures are particularly useful in solving this problem: kd-trees and range trees.

8.3 Kd-Trees

A kd-tree is a binary tree with nodes that represent rectangular regions of $\mathbb{R}^d$ [1]. Each node of the data structure stores a single point p_v from S. An example of 2-dimensional space divided into regions by a kd-tree can be found in Figure 8.2. Intuitively, the construction algorithm begins by enclosing all points in S with a rectangle, then dividing that rectangle in half in the x-direction. Each half is then divided in the y-direction, and so forth, until there is only one point in each region.

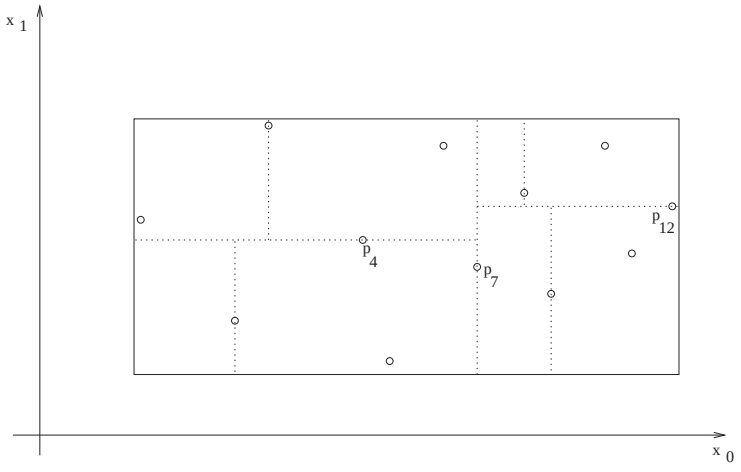


Figure 8.2: Two-dimensional Kd-tree Construction

8.3.1 Data Structure

Pseudocode used to store a set of points in a d-dimensional kd-tree is given below:

```
kdtree(S, i, d)
    if S = emptyset, return null
    p = point in S w/ median x_i-coordinate
    create a node V
    set p(V) = p
    S_- = { q is an element in S | x_i(q) < x_i(p) }
    S_+ = { q is an element in S | x_i(q) > x_i(p) }
    left(V) = kdtree(S_-, (i+1) mod d, d)
    right(V) = kdtree(S_+, (i+1) mod d, d)
    return V
end
```

Note that the dimension in which the median is found changes at each recursive call. The kd-tree constructed from Figure 8.2 can be found in Figure 8.3.

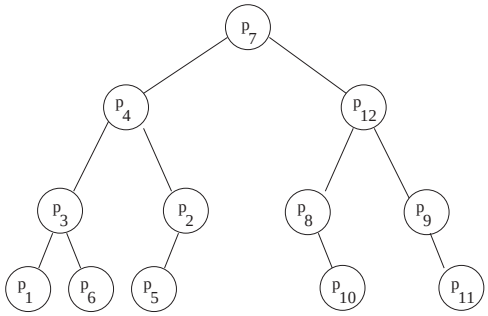


Figure 8.3: Two-dimensional Kd-tree

8.3.2 Queries

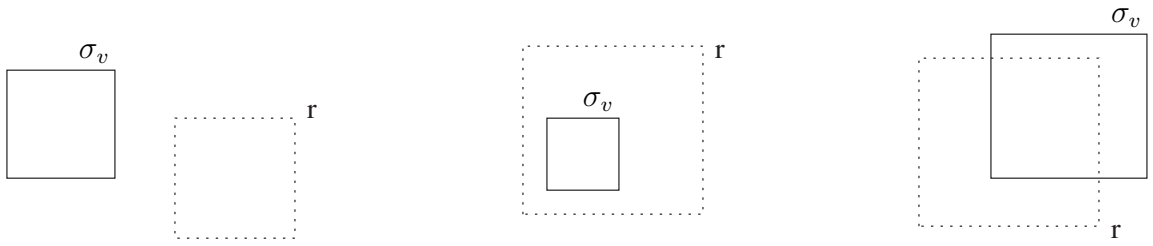
Before examining query pseudocode, let's try to gain an intuition for a tree traversal.

At each node, one of three cases can occur, and examples of each are given in Figure 8.4. In the first case, the rectangle representing the current node, σ_V, is disjoint from r. We will call these nodes "white." In the second case, $\sigma_V \subset v$. We will call nodes of this type "black." The third case is more complicated; here, there is less than total overlap, and the children of the current node will need to be searched recursively. We will call these nodes "gray" nodes.

Claim 1 *If a node is white or black, all of its children are white or black, respectively.*

Claim 2 *If a node is gray, any of its children can be of any type.*

Pseudocode for a d-dimensional kd-tree query is given below. Note that r is of the form $[a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$.



Case (a) – White Case (b) – Black Case (c) – Gray

Figure 8.4: Kd-tree Query Cases

```
query(r, V, i, d)
    if V is null, return emptyset
    if p(V) is an element in r, return p(V)
    if a_i < x_i(p(V))
        query(r, left(V), (i+1) mod d, d)
    if b_i > x_i(p(V))
        query(r, right(V), (i+1) mod d, d)
end
```

8.3.3 Time and Space Bounds

In the construction pseudocode given above, the time required to find the median is $O(n)$. Since the two recursive calls are on point sets half the size of the previous level, the preprocessing recurrence relation is $P(n) \leq 2P(n/2) + O(n)$, and its solution is $P(n) = O(n \lg n)$.

The time required for a query is a bit more complicated, but can be managed by noticing two things:

Claim 3 *Query time is proportional to the number of black nodes in a query plus the number of gray nodes.*

Claim 4 *The number of black nodes is bounded by k, the number of points in the query rectangle.*

Since we know that the query time must include a factor of k, it suffices to bound the number of gray nodes. Consider the two rectangles r and σ_V. If vertex V is a gray node, one of the edges of r intersects σ_V. We can then ask how many nodes V_2 are there such that a horizontal edge of r intersects σ_{V_2}? Call this quantity $\phi(n)$. If you consider the left side of Figure 8.5, you can see that only two of the grandchildren intersect the edge. Therefore, as you progress down the kd-tree, only the current node, the two children, and $2/4 = 1/2$ of the grandchildren can possibly be gray. Therefore, the query recurrence relation can be expressed as $\phi(n) = 2\phi(n/4) + 3$. The solution to this relation is $\phi(n) = O(\sqrt{n})$, so the total query time is $O(\sqrt{n} + k)$.

In addition, since each node in the kd-tree contains exactly one point, it is clear that the space required for this data structure is $O(n)$.

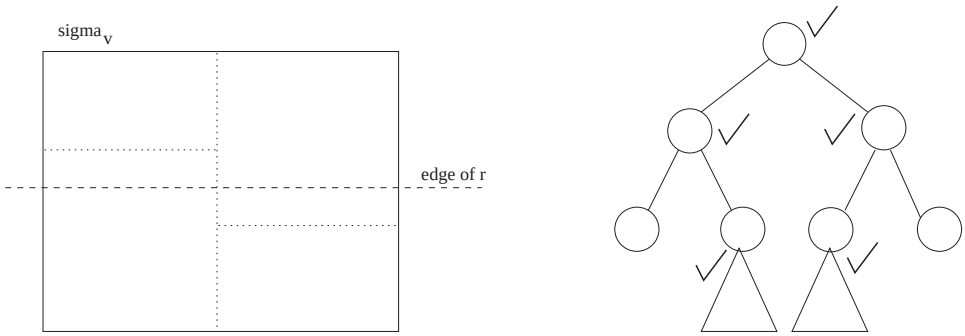


Figure 8.5: Gray Node Procedure

8.4 Range Trees

Since it has been shown that the expected query time for a kd-tree is better than $O(\sqrt{n}+k)$, it is natural to ask whether a new data structure can offer us a superior worst-case bound. The answer is yes, though the improved query time will increase the necessary storage space to $O(n \lg n)$, as will be shown. One such data structure is a range tree.

8.4.1 Data Structure

A range tree is a two-level data structure. See Figure 8.6 for an example. The primary structure is a balanced binary tree T ordered by x-coordinate such that each point in S is stored at a leaf node. At each internal node V, S_V represents the set of all points in the subtree rooted at V. The secondary data structure is an array P_V at each node which stores S_V sorted by y-coordinate. A range tree can be created by joining points containing nodes from the bottom-up, while also merging sorted P_V lists at each level.

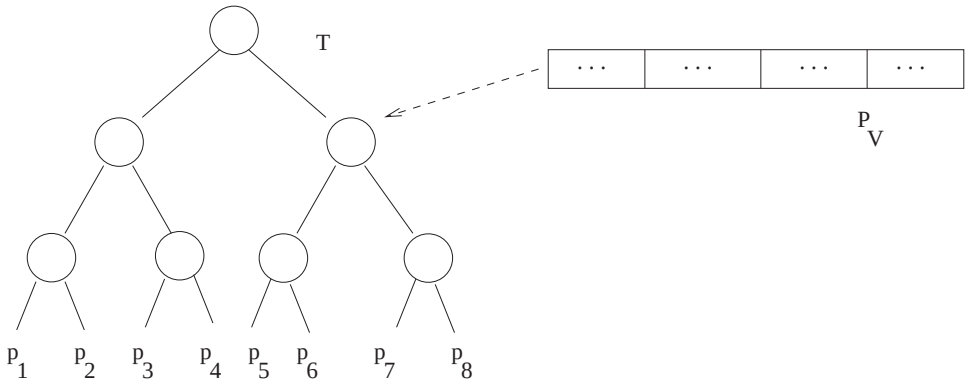


Figure 8.6: Range Tree Data Structure

8.4.2 Time and Space Bounds

Preprocessing time for a range tree is $O(n \lg n)$. Creating the primary tree requires linear time, since the total number of nodes to be created is $O(n)$. Since we are merging sorted lists as we move up $O(\lg n)$ levels, the time required to create the secondary structures is $O(n \lg n)$. Queries are performed by searching for a_1 and b_1 in the primary data structure and checking the P_V lists which fall in the correct ranges. See Figure 8.7 for an example; nodes that fall "inside" the query paths (checked) are the ones that will be searched. Since at most $2 * \lg n$ P_V lists are searched (each in $O(\lg n)$ time), the total time required for a query is $O(\lg^2 n + k)$. Since each point is stored once at each level, the total space required is $O(n \lg n)$.

The query time can be improved to $O(\lg n + k)$, which will be described in the next lecture.

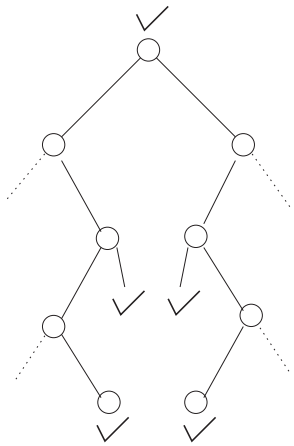


Figure 8.7: Range Tree Query

8.5 Range Search and Nearest Neighbor

The problem of range search and nearest neighbor search are closely related. For instance, a kd-tree can be used to efficiently find a target point's (t's) nearest neighbor in S. See Figure 8.8 for an example of this process [3].

Given the kd-tree constructed above, find a first approximation for the nearest neighbor by searching for the leaf node which would contain the target point. Use the point p contained therein as this approximation. p may not be the nearest neighbor, but we do know that the nearest neighbor must lie at least as close, and is therefore within the circle defined by t and p. We next back up to the parent of the current node. We then check to see whether a closer point could exist in the parent node's other child. If so, proceed down the tree and see if the next point is closer. Recompute the circle if necessary. If a closer point could not exist in p's sibling, move up the tree and repeat.

It is clear from this description that at least $\Omega(\lg n)$ nodes will be inspected by this algorithm; at least one leaf node will be examined. The number of nodes examined can also not exceed n. As it turns out, under certain assumptions the expected time for finding the nearest neighbor is $O(\lg n)$. Calculation of the expected

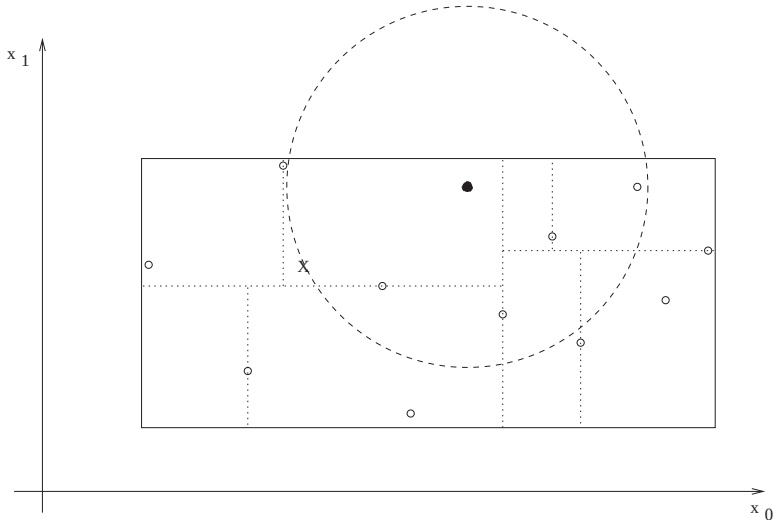


Figure 8.8: Kd-tree Nearest Neighbor Search

number of inspections is difficult because it depends on the expected distribution of the points in the kd-tree, and in the distribution of the target points. The analysis can be found in [2].

References

[1] J.L. Bentley. Multidimensional divide and conquer. *Communications of the ACM*, 23(4):214-229, 1980.

[2] J.H. Friedman, J.L. Bentley, R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209-226, 1977.

[3] A. Moore. A tutorial on kd-trees. University of Cambridge Computer Laboratory Technical Report No. 209, 1991.