Lecture 19: Range Searching in kd-trees

Reading: Today's material is discussed in Chapt. 2 of Samet's book on spatial data structures.

Range Queries: So far we have discussed insert, deletion, and nearest neighbor queries in kd-trees. Today we consider an important class of queries, called *orthogonal range queries*, or just *range queries* for short. Given a set of points S, the query consists of an axis-aligned rectangle r, and the problem is to report the points lying within this rectangle. Another variant is to count the number of points lying within the rectangle. These variants are called *range reporting queries* and *range counting queries*, respectively. (Range queries can also be asked with nonorthogonal objects, such as triangles and circles.) We will consider how to answer range queries using the kd-tree data structure. For example, in the figure below the query would report the points $\{7, 8, 10, 11\}$ or the count of 4 points.

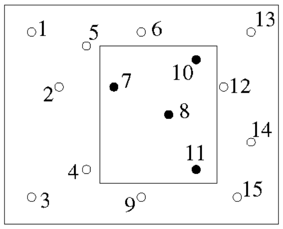

Figure 57: Orthogonal range query.

Range Searching in kd-trees: Let us consider how to answer range counting queries. We assume that the range Q is a class which supports three basic utilities.

`Q.contains(Point p)`: True if p is contained within Q.

`Q.contains(Rectangle C)`: True if rectangle C is entirely contained within Q.

`Q.isDisjointFrom(Rectangle C)`: True if rectangle C is entirely disjoint from Q.

Let t denote the current node in the tree, and let C denote the current rectangular cell associated with t. As usual let *cd* denote the current cutting dimension. The search proceeds as follows. First, if we fall out of the tree then there is nothing to count. If the current node's cell is disjoint from the query range, then again there is nothing to count. If the query range contains the current cell, then we can return all the points in the subtree rooted at t (`t.size`). Otherwise, we determine whether this point is in the range and count it if so, and then we recurse on the two children and update the count accordingly. Recall that we have overloaded the addition operation for cutting dimensions, so `cd+1` is taken modulo the dimension.

kd-tree Range Counting Query

```
int range(Range Q, KDNode t, int cd, Rectangle C) {
    if (t == null)           return 0       // fell out of tree
    if (Q.isDisjointFrom(C)) return 0       // cell doesn't overlap query
    if (Q.contains(C))       return t.size  // query contains cell
    int count = 0
    if (Q.contains(t.data)) count++         // count this data point
                                            // recurse on children
    count += range(Q, t.left,  cd+1, C.trimLeft(cd, t.data))
    count += range(Q, t.right, cd+1, C.trimRight(cd, t.data))
```

[1]Copyright, David M. Mount, 2001

```
        return count
    }
```

The figure below shows an example of a range search. Next to each node we store the size of the associated subtree. We say that a node is *visited* if a call to `range()` is made on this node. We say that a node is *processed* if both of its children are visited. Observe that for a node to be processed, its cell must overlap the range without being contained within the range. In the example, the shaded nodes are those that are not processed. For example the subtree rooted at b is entirely disjoint from the query and the subtrees rooted at n and r are entirely contained in the query. The nodes with squares surrounding them those whose points have been added individually to the count (by the second to last line of the procedure). There are 5 such nodes, and including the 3 points from the subtree rooted at r and the 2 points from the subtree rooted at n, the final count returned is 10.

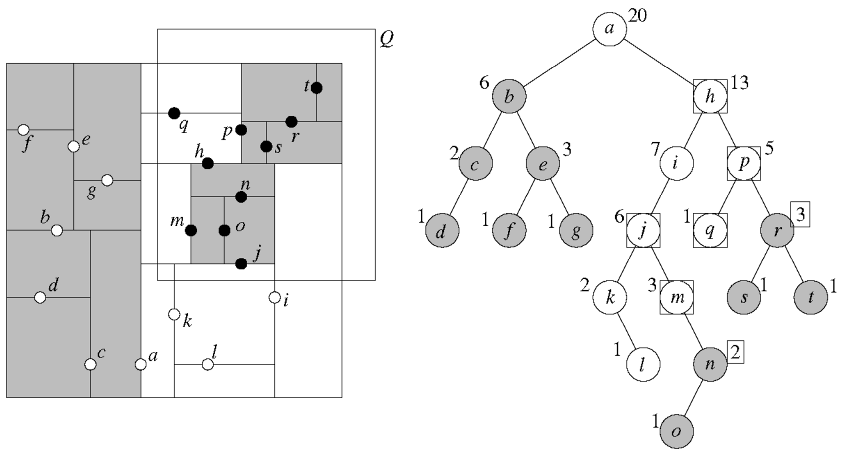

Figure 58: Range search in kd-trees.

Analysis of query time: How many nodes does this method visit altogether? We claim that the total number of nodes is $O(\sqrt{n})$ assuming a balanced kd-tree (which is a reasonable assumption in the average case).

Theorem: Given a balanced kd-tree with n points range queries can be answered in $O(\sqrt{n})$ time.

Recall from the discussion above that a node is processed (both children visited) if and only if the cell overlaps the range without being contained within the range. We say that such a cell is *stabbed* by the query. To bound the total number of nodes that are processed in the search, it suffices to count the total number of nodes whose cells are stabbed by the query rectangle. Rather than prove the above theorem directly, we will prove a simpler result, which illustrates the essential ideas behind the proof. Rather than using a 4-sided rectangle, we consider an orthogonal range having a only one side, that is, an orthogonal halfplane. In this case, the query stabs a cell if the vertical or horizontal line that defines the halfplane intersects the cell.

Lemma: Given a balanced kd-tree with n points, any vertical or horizontal line stabs $O(\sqrt{n})$ cells of the tree.

Proof: Let us consider the case of a vertical line $x = x_0$. The horizontal case is symmetrical.

Consider a processed node which has a cutting dimension along x. The vertical line $x = x_0$ either stabs the left child or the right child but not both. If it fails to stab one of the children, then it cannot stab any of the cells belonging to the descendents of this child either. If the cutting dimension is along the y-axis (or generally any other axis in higher dimensions), then the line $x = x_0$ stabs both children's cells.

Since we alternate splitting on left and right, this means that after descending two levels in the tree, we may stab at most two of the possible four grandchildren of each node. In general each time we descend two more levels we double the number of nodes being stabbed. Thus, we stab the root node, at most 2 nodes at level 2 of the tree, at most 4 nodes at level 4, 8 nodes at level 6, and generally at most 2^i nodes at level $2i$.

Because we have an exponentially increasing number, the total sum is dominated by its last term. Thus is suffices to count the number of nodes stabbed at the lowest level of the tree. If we assume that the kd-tree is balanced, then the tree has roughly $\lg n$ levels. Thus the number of leaf nodes processed at the bottommost level is $2^{(\lg n)/2} = 2^{\lg \sqrt{n}} = \sqrt{n}$. This completes the proof.

We have shown that any vertical or horizontal line can stab only $O(\sqrt{n})$ cells of the tree. Thus, if we were to extend the four sides of Q into lines, the total number of cells stabbed by all these lines is at most $O(4\sqrt{n}) = O(\sqrt{n})$. Thus the total number of cells stabbed by the query range is $O(\sqrt{n})$, and hence the total query time is $O(\sqrt{n})$. Again, this assumes that the kd-tree is balanced (having $O(\log n)$ depth). If the points were inserted in random order, this will be true on average.