Geometric Computation, September 20

Orthogonal Range Searching

Scribes: Mohammad Taghi Hajiaghayi, Brian Dean *Lecturer: Erik Demaine*

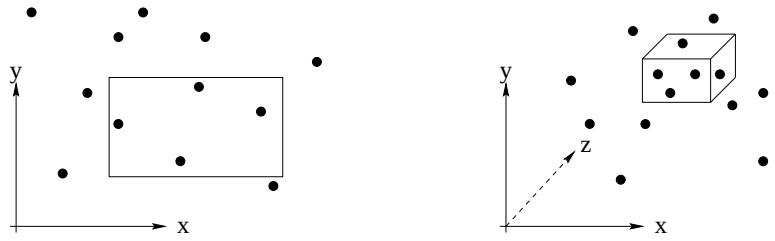

Figure 1: Query in 2(3)-dimensional space

1 Introduction

Orthogonal Range Searching can be expressed as follows. Given n points in d-dimensional space and an axis-aligned box, report all points in the box. This problem has many applications in databases and shows how computational geometry can be applied in other practical areas such as database systems. Each record having d fields in a database can be considered as a point in d-dimensional space and a query asking to report all records whose fields lie between specified values can be considered as a box contains some points. For example, in a *restaurant database*, in which each record has three fields *name, location* and *price*, a box can contain those restaurants whose locations are near our house and whose prices are in a specified range.

One naive algorithm for solving the problem is as follows: check each coordinate of every point against the corresponding coordinates of the box and find all points which satisfy all constraints. The running time of this algorithm is in $O(n \cdot d)$. In most cases, the inputs in a database are relatively static, but the queries are dynamic. Thus in the rest of this lecture, we assume that our n given points are fixed and we want to prepossess the points to support fast queries.

The result of each query can be in the following forms:

- List all points in the box
- List first k points in the box
- Count the number of points in the box
- Is there any point in the box?

Here we deal with the first type of request, in which we must list all points in the box; however, our approach can also be applied to the other types of requests. Note that for reporting k points, we need at least $\Omega(k)$ time – this property is called the *output sensitivity* of the running time of our queries.

2 The One-Dimensional Case

We first consider the case of one-dimensional space, in which our given points have only x-coordinates and our queries are just intervals. One prepossessing step is the following: sort all the points according to their x-coordinate and put them into an array. Now, to answer each query, we only need to use a binary search to find the locations of two end-points of the query interval in the array and then report all points between these two locations. Building the array takes $O(n \log n)$ time and needs $O(n)$ space. Using binary search, we can find the end-points of each interval in $O(\log n)$ time, however still we need $O(k)$ time to report all k points in the interval. Thus the overall running time is in $O(\log n + k)$ per each query.

Binary search is like traversing a balanced binary search tree. In this search tree, points are stored at leaves and some copies of them are stored at internal nodes. In only one dimension, our binary search tree will be ordered based on x coordinate: each leaf of the tree will contain a point along with its associated x coordinate, and each internal node will contain the maximum x coordinate of the points in its left subtree, in order to maintain the proper ordering of the nodes in the tree. Building this data structure takes $O(n \log n)$ time and needs $O(n)$ space. We can also perform the *add*, *delete* and *search* operations each in $O(\log n)$ time (n is the number of points). We assume for simplicity that there is no duplicate in our input points.

Answering a query in the aforementioned binary search tree is more tricky. In fact, we represent our solution using a collection of subtrees whose leaves contain all our desired

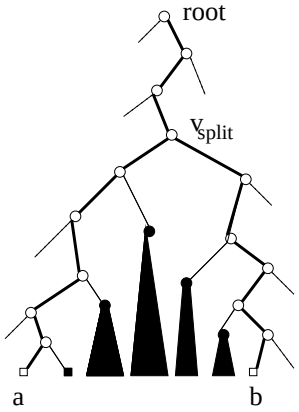

Figure 2: The selected subtrees

points. Suppose we are given an interval $[a, b]$ as a query. First we observe that our solution contains all subtrees bounded by two paths to a and b (see Figure 2). Thus, we first find a *split node* from which paths to a and b diverge. Now, starting from this split node, we compare a to the key x of each node. If $a > x$ then we conclude that the left subtree is not in our solution and thus we recurse into the right subtree. If $a \leq x$ we select the right subtree as a subtree in our solution and we continue our process into left subtree. Analogously, we perform this operation for b. One can observe that each selected subtree has different height in processing the point a (or b). Therefore each query can be answered in $O(\log n)$ time by selecting at most $O(\log n)$ subtrees (we note that, if we want to report the points, we need still need $O(\log n + k)$ time).

3 The kd-Tree Data Structure

We now consider the generalization of this approach to 2-dimensional space in which each point has an x-coordinate and a y-coordinate. Here we need to split the points as we did in 1-dimensional space. One natural solution is to use lines for splitting. Here we use vertical and horizontal lines and for the sake of simplicity, we assume that all x coordinate values are distinct and all y coordinate values are distinct. In fact, if there exist points with equal x coordinates (or y coordinates) we can add $+\epsilon$ and $-\epsilon$ to separate them. Using axially-aligned *crosses* is another way to split the plane which results in *quadtrees*. Quadtrees are common in practice but difficult to prove good bounds. In this lecture, we only consider the first way of splitting.

Our data structure in 2D space is called a 2-dimensional kd-trees. To construct this data

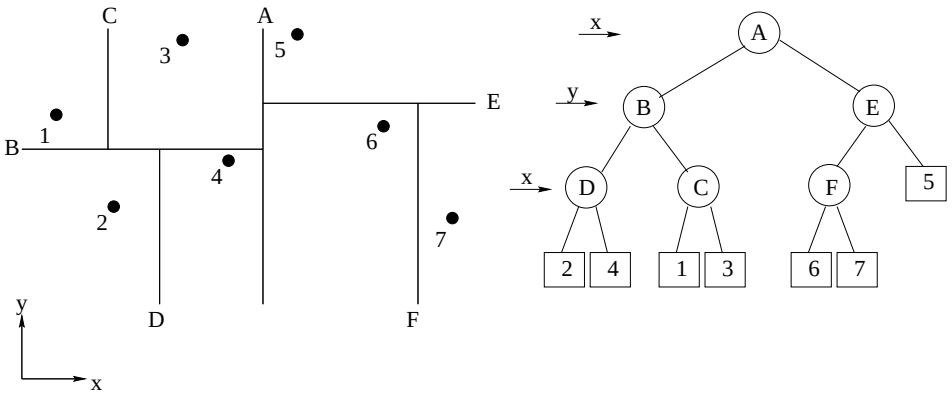

Figure 3: A *kd*-tree for 7 points

structure, we alternate between x and y branching nodes corresponding to vertical and horizontal cutting lines and each time we cut at median x or y. An example is depicted in Figure 3.

We note that this approach can be generalized to d-dimensional space (just alternate among d coordinates).

Now our algorithm to answer a query is as follows:

- For a subtree whose region does not intersect query box, we skip.
- For a subtree whose region is contained in query box, we select it and return.
- Otherwise we recurse into the subtree.

We note that in the above algorithm, the branching factor is 2 and we might visit a lot of the tree. Also it is possible that we may perform "wasted work" in the sense that we may visit subtrees that turn out to contain no points in answer. Below we analyze the efficiency of the algorithm.

Let's analyze the worst-case query time of a two-dimensional kd-tree. Every query operation forces us to transverse part of the kd-tree, so we need to consider all of the leaf nodes of the kd-tree which may potentially be touched during this transversal. Each leaf node corresponds to a rectangular region of the plane. We can consider these regions as three different groups:

1. Regions which lie completely outside the query rectangle will be discarded from the transversal and won't impact the running time.

2. Regions which are completely contained within the query rectangle will be touched during the transversal, but for each of these we will return a point. The time to visit all of these regions will be $O(k)$, where k denotes the total number of points returned by the query.

3. A region which lies "on the boundary" of the query rectangle will be visited by the transversal but may not produce a point, so we need to bound the number of such regions. To do so, consider each of the 4 edges of the query rectangle separately:

$$\begin{aligned} \#\text{ regions stradling boundary of query rectangle} &\leq \\ 4 \cdot (\#\text{ of regions stradling one edge of query rectangle}) &\leq \\ 4 \cdot (\#\text{ of regions stradling a vertical or horizontal line}) &= \\ 4 \cdot O(\sqrt{n}) &= O(\sqrt{n}) \end{aligned}$$

   The $O(\sqrt{n})$ term is obtained by the following argument: suppose we want to count the number of regions stradling a given vertical line (the same argument works for horizontal lines). During our transversal of the kd-tree, when we reach an x-split node we only need to explore one of its subtrees – the one containing the vertical line. However when we reach a y-split node we need to branch and explore both of its subtrees. Since the kd-tree has a height of $\log n$, and since our transversal branches on half of these levels, we'll end up touching $2^{(\log n)/2} = O(\sqrt{n})$ leaf nodes.

To summarize the 2-dimensional kd-tree, we have:

- Query time: $O(\sqrt{n} + k)$
- Time to build kd-tree: $O(n \log n)$
- Space required: $O(n)$

In d dimensions, we can use the same argument as above to analyze query time. Note that in this case we need to check the number of regions which stradle the $2d$ edges of the query rectangle. For each edge, we apply the reasoning above and conclude that when visiting a split node for any of the $d - 1$ dimensions perpendicular to a given edge, we'll need to branch and visit both of its subtrees, leading to the following bounds:

- Query time: $O(dn^{1-1/d} + k)$
- Time to build kd-tree: $O(n \log n)$
- Space required: $O(dn)$

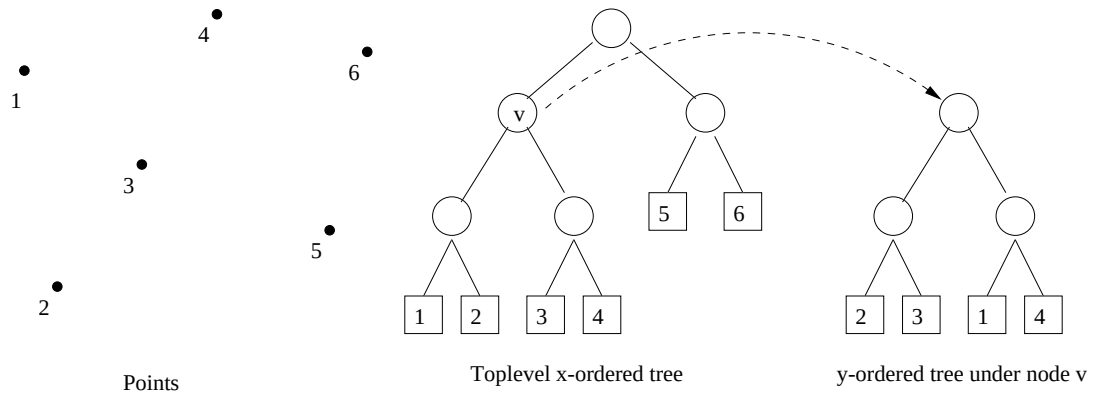

Figure 4: Sample range tree

4 Range Trees

An alternative data structure for performing orthogonal range queries is a *range tree*. For a 2-dimensional pointset, the range tree is a balanced binary search tree with points as leaves which is ordered based on x coordinate. To every node v in this tree is attached another balanced binary search tree, ordered by y coordinate, containing all of the points in v's subtree. An example of a range tree is shown in Figure 4; the x-ordered toplevel tree and one of the y-ordered lowlevel trees is shown.

How much space is required to store a range tree in 2 dimensions? Note that every point in the pointset exists in every y-ordered tree attached along the path from the root to itself in the toplevel x-ordered tree. Since every point is in $O(\log n)$ trees, the total space required is $O(n \log n)$. Queries are performed as one might expect: first the top-level x-ordered tree is used to identify all points matching the x range of the query. As shown in Figure 2, we can represent these points as the union of the leaves of at most $2 \log n$ complete subtrees within the top-level tree. We then perform y queries for the y-ordered trees corresponding to each of these subtrees to identify the points matching our query. The total query time is thus $O(\log^2 n + k)$, where k points are produced by the query.

The following table summarizes the time and space requirements for range trees:

	2 dimensions	d **dimensions**
Query time	$O(\log^2 n + k)$	$O(\log^d n + k)$
Build time	$O(n \log^{d-1} n)$	$O(n \log^{d-1} n)$
Space required	$O(n \log n)$	$O(n \log^{d-1} n)$