1   Overview

In the last lecture we continued to study binary search trees from the geometry perspective of arboreally satisfied sets. We gave lower bounds for the runtime of a BST — two of them were Wilbur's lower bounds [Wil89], and the last was "signed greedy." We also looked at a type of BST called a Tango tree and showed that it was $O(\lg \lg n)$ competitive with an optimal dynamic tree.

In this lecture we consider the problem of orthogonal range queries, focusing mostly on $\mathbb{R}^2$ but also indicating how to extend the results to higher dimensions. An orthogonal range query is the following problem: given a set S of points in $\mathbb{R}^d$, and an axis-aligned box B, what are all elements of S that lie in B? The goal is to implement a data structure with reasonably efficient preprocessing on the set S that can answer all such queries efficiently. A natural application is to database queries (e.g. how many people lie in a given age range and income range?).

Since the output of a query could be very large, we will measure the efficiency of our data structures in terms of both n, the number of points in S, and k, the size of the output of a given query. We start by introducing range trees, which in the two-dimensional case use $O(n \lg n)$ space and pre-processing time and have a query time of $O(\lg^2 n + k)$. (In d dimensions, they use $O(n \lg^{d-1} n)$ storage and pre-processing and have a query time of $O(\lg^d n + k)$.) We then describe kd-trees for the 2-dimensional case, which are more storage-efficient ($O(n)$ space), but have a bad worst-case query time $O(\sqrt{n})$. We also introduce the idea of fractional cascading, a trick that reduces the query time for range trees from $O(\lg^d n + k)$ to $O(\lg^{d-1} n + k)$. Finally, we indicate how to deal with the case of points with duplicated coordinates (the rest of the exposition assumes that all points have distinct coordinates).

A good reference for the material in this lecture is the book *Computational Geometry: Algorithms and Applications* by deBerg et al. [dBea08].

We begin by describing the problem of orthogonal range queries in detail. We will then look at various data structures for answering orthogonal range queries efficiently.

2   Orthogonal Range Queries

The first problem we will approach is **orthogonal range searching**. Suppose we are given a set of points. For the moment we will assume that these points are in $\mathbb{R}^2$, but in general we can allow our space to be any dimension. Now draw a rectangle with sides parallel to the coordinate axes (since this is orthogonal range searching, we require a rectangle, although in principle we could allow other shapes). How can we report all of the points that are inside of the rectangle?

Before we proceed, we note that there are a number of related questions we can ask. For example,

we might just want to know how many such points there are, or even if any such points exist. These are different questions, but we will answer them with basically the same techniques.

To motivate this problem, we can think of these points as representing entries in a database. For example, suppose we were to graph a number of people's ages versus their salaries. Then our question is equivalent to finding all people whose age and salary both fall in some given range. Hence even though the problem as phrased is a geometric one, it has many applications.

For now, we will assume that the points are all distinct when viewed from each coordinate. In other words, we will assume that there is nothing like $\{(2,3), (2,4)\}$, where the first and second point both have the same first coordinate. In the last section, we will discuss how to drop this assumption.

As an example of an orthogonal range query, consider the set of points $S = \{(3,1), (2,7), (4,5)\}$ in $\mathbb{R}^2$. A possible orthogonal range query would be to ask for the points in S that lie within the rectangle $[2:3] \times [1:7]$, which would be $(3,1)$ and $(2,7)$. We could also ask for the points in $[2:4] \times [3:7]$, which would be $(2,7)$ and $(4,5)$.

3 Range Trees

The standard tool for answering orthogonal range queries is a range tree. Range trees are useful because they are reasonably space-efficient and also have efficient pre-processing and query time.

3.1 One-dimensional case

We first consider the one-dimensional case. In this case, an orthogonal range query just asks for all elements of a set of numbers that lie in a given interval. For example, we could have $S = \{2, 4, 8, 16, 17, 18, 20, 24\}$. If we ask for all points in $[10:19]$, the answer would be $16, 17, 18$.

The one-dimensional case is fairly straightforward — to get an offline algorithm, we can just create a sorted array and use binary search to get a query time of $O(\lg n + k)$ (recall that n is the size of S and k is the size of the output of a given query). If we want to support insertions to the set, then we can use a binary search tree. Below is a sketch of the BST-based algorithm:

Input: x_{min} and x_{max}, the two endpoints of an interval.

Algorithm:

1 Find v_{split}, the vertex in the BST at which the search paths for x_{min} and x_{max} diverge.

2a Continue to x_{min} from v_{split}, outputting all right subtrees of visited vertices.

2b Continue to x_{max} from v_{split}, outputting all left subtrees of visited vertices.

As noted, queries are $O(\lg n + k)$. Also, space is $O(n)$ and pre-processing is $O(n \lg n)$. The BST implementation is the version of the data structure that generalizes well to higher dimensions, although supporting online insertions in dimensions greater than 1 is more difficult and will not be discussed in this lecture (we will only discuss how to pre-process and query).

Note that if we don't care about insertions, then building the binary search tree is actually fairly straightforward, and doesn't involve any fancy balancing algorithms. We can just store the elements of S in order as the leaves of the tree, and build up the nodes above each element to contain, for example, the minimum value of any node in the subtree (although the nodes can actually contain arbitrary values as long as the BST property is satisfied). So for example, 2 and 4 share a common parent which we can choose to contain the value 2. Similarly we choose 8 to be the parent of 8 and 16.

3.2   Two-dimensional case

We now consider the two-dimensional version of the problem. The idea is to first search for the x-range (using a 1-dimensional range tree), then report only the points with y-coordinates that fit in a specified range (by nesting range trees at each node).

Suppose we want to query for points in the range $[x_{\min} : x_{\max}]$ and $[y_{\min} : y_{\max}]$. First of all, we can use a one-dimensional range tree to find all points in the slice defined by $[x_{\min} : x_{\max}]$. However, this is different from the one-dimensional case in that we don't want to report the entire subtree but instead only want those points that fit into the given y-range. Note that this can also be done with a one-dimensional range tree!

Hence we form a multilevel datastructure that is a one-dimensional range tree on x-coordinates and that stores other one-dimensional trees on y-coordinates. Each node points to another tree containing the y-coordinates of its decendents.

To be more precise, the top level of our data structure will be a balanced binary search tree where the leaves are the x-coordinates of points in S. Each non-leaf node v will be "responsible" for the set of points whose x-coordinates lie in the subtree rooted at v. We will store a 1-dimensional range tree at v that keeps track of all such points by their y-coordinate.

To query a range $[x_{min} : x_{max}] \times [y_{min}, y_{max}]$, we will start as before by querying the top-level subtree to get $O(\lg n)$ nodes that are roots of subtrees whose union contains all points with x-coordinates in $[x_{min} : x_{max}]$. For each of these nodes u, we then perform the query $[y_{min}, y_{max}]$ on the nested range tree at u, which takes $O(\lg n + k'(u))$ time, where $k'(u)$ is the number of points output by the query. This means that the total runtime for a query is $O(\lg^2 n + \sum_u k'(u)) = O(\lg^2 n + k)$, since the sum of $k'(u)$ over all u is equal to the total size of the output.

Also note that each point is stored in $O(\lg n)$ 1-dimensional range trees, so storage is $O(n \lg n)$, and pre-processing is similarly $O(n \lg n)$ as long as we are careful to sort the elements by y-coordinate at the beginning.

We describe the algorithm for querying more formally below:

Input: x_{min}, x_{max}, y_{min}, y_{max}.

Algorithm:

1 Let $\mathcal{C}$ be a collection of (roots of) BSTs, initially empty.

2 Find v_{split}, the vertex in the top-level range tree at which the search paths for x_{min} and x_{max} diverge.

3a Continue to x_{min} from v_{split}. For each right child of a visited vertex, add the (root of) the BST stored at that node to $\mathcal{C}$.

3b Continue to x_{max} from v_{split}. For each left child of a visited vertex, add the (root of) the BST stored at that node to $\mathcal{C}$.

4 For each element of $\mathcal{C}$, perform a 1-dimensional range query with range $[y_{min} : y_{max}]$ and output the result.

3.3 d-dimensional case

The same idea as above will also work for the d-dimensional case. Instead of nesting a BST at each node, we nest a $(d-1)$-dimensional range tree. In this case we end up with $O(n \lg^{d-1} n)$ storage and preprocessing, and a query time of $O(\lg^d n + k)$. We sketch the algorithm below (the d-dimensional algorithm is given in terms of the $(d-1)$-dimensional algorithm).

**Input:** $x_{1,min}$, $x_{1,max}$, $x_{2,min}$, $x_{2,max}$, ..., $x_{d,min}$, $x_{d,max}$.

Algorithm:

1 Let $\mathcal{C}$ be a collection of (pointers to) $(d-1)$-dimensional range trees, initially empty.

2 Find v_{split}, the vertex in the top-level range tree at which the search paths for $x_{1,min}$ and $x_{1,max}$ diverge.

3a Continue to $x_{1,min}$ from v_{split}. For each right child of a visited vertex, add (a pointer to) the $(d-1)$-dimensional range tree stored at that node to $\mathcal{C}$.

3b Continue to $x_{1,max}$ from v_{split}. For each left child of a visited vertex, add (a pointer to) the $(d-1)$-dimensional range tree stored at that node to $\mathcal{C}$.

4 For each element of $\mathcal{C}$, perform a $(d-1)$-dimensional range query with range $[x_{2,min} : x_{2,max}] \times \cdots \times [x_{d,min} : x_{d,max}]$ and output the result.

4 kd-trees

We now present an alternative to range trees, namely kd-trees (due to Bentley [BS75]). kd-trees are especially useful in two dimensions, and we will only discuss them in the two-dimensional case. kd-trees allow us to use only $O(n)$ storage, but we have to pay for this in extra query time.

We note that range tree constructions could be described as "split the points in half according to x-coordinate and then apply divide-and-conquer; next do the same for the y-coordinates." For kd-trees we will use only one divide-and-conquer. Namely, take a set of points, and slice them in half with a vertical line. Slice each resulting region in half with a horizontal line (really, a line segment, as it should stay inside the set). Now divide-and-conquer, alternating horizontal and vertical splits. Keep slicing until every cell contains exactly one point. An example is given below:
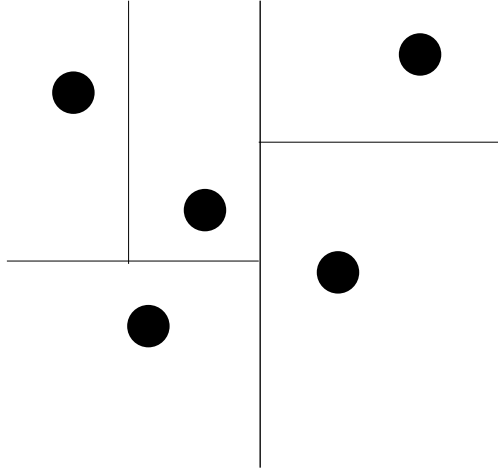
Thus a *kd*-tree ends up as a binary tree where each node v has a region in the plane (call it *region(v)*) associated with it. Furthermore, the region of a node is the union of the regions of its children, and all regions corresponding to leaves contain a unique point in the input set.

Since a region can be expressed compactly (it only requires four numbers to specify the corners of a rectangle), the storage for a kd-tree is $O(n)$, although we still require $O(n \lg n)$ pre-processing.

We perform queries by recursing through the kd-tree. Suppose that q is the query region. If we are at a node v in the tree, then we have four cases.

Case one: v is a leaf. Then we can easily determine if the unique point in *region(v)* is contained in q.

Case two: *region(v)* is disjoint from q. In this case we can just stop.

Case three: *region(v)* is contained in q. In this case we can just report all points in *region(v)* (which we can access by using the subtree rooted at v).

Case four: If *region(v)* is neither disjoint from nor contained in q, and v is not a leaf, then we need to recurse on the left and right subtrees of v.

It turns out that the worst-case runtime for queries in kd-trees is $O(\sqrt{n} + k)$. This is an exercise on the homework. It involves analyzing the number of times that the fourth case can possibly occur.

5 Fractional Cascading

In the section on range trees, we saw that queries took $O(\lg^d n + k)$ time, even though the space usage and pre-processing time for range trees was only $O(\lg^{d-1} n)$. The question arises, can we bring the query time down to $\lg^{d-1} n$ as well? It turns out the answer is yes, and the idea that allows us to do this is called fractional cascading. It was first presented by Chazelle and Guibas in [CG86].

5.1 Warm-up: Arrays

Suppose that we have 2 arrays A and B, and B stores a subset of A. For example, we could have $A = [1, 5, 7, 9, 17, 21, 23, 31, 37, 48, 90]$ and $B = [5, 9, 17, 23, 31, 48]$. How can we use the results of searches on A to speed up searches on B?

The answer is to keep a set of pointers from A to B. For each element x of A, let $p(x)$ be the smallest element y in B such that $y \geq x$. Then keep a pointer from x to $p(x)$ for each x. Now if we search for an element x in A, we can easily determine if that element also lies in B by looking at $p(x)$. Similarly, if we make a range query on A, then we can perform a range query on B by looking at $p(x_{min})$ and $p(x_{max})$, where x_{min} is the smallest element of A in the given range, and x_{max} is the largest such element.

5.2 Two-dimensional Cascading

It turns out that we can apply the idea from the above subsection in the case of 2-dimensional range trees. When we do this, we speed up queries to $O(\lg n + k)$. For each node v, keep pointers from the range tree at v to the range trees at the two children of v. (We construct the pointers in exactly the same way — the pointer from x points to the smallest element y of the target data structure with $y \geq x$.)

We describe the query process formally below:

Input: x_{min}, x_{max}, y_{min}, y_{max}

Output:

1. As before, find v_{split}.

2. Search for y_{min} and y_{max} at v_{split}.

3. Continue down the top-level tree to x_{min} and x_{max}. Use the pointers to each next level to avoid having to re-do searches.

Since we now only have to perform a single set of searches (for y_{min} and y_{max}), the query time is $O(\lg n + k)$. We can also check that we take no hit in memory usage or preprocessing time, so that they are both $O(n \lg^{d-1} n)$.

5.3 d-dimensional Cascading

Unfortunately, we can't continually cascade to bring the query time down to $O(\lg n + k)$ in general. However, since a d-dimensional range tree is constructed from $(d-1)$-dimensional range trees, the fact that fractional cascading speeds up 2-dimensional range trees means that we also get a speed-up in all higher dimensions (as we just make the two lowest-level dimensions of the range tree a fractionally cascaded 2D range tree). Therefore, fractional cascading gives us $O(\lg^{d-1} n + k)$ queries in general.

6   Dealing with Duplicated Coordinates

Our final topic of discussion is what to do when there are duplications in a given coordinate. We will provide an approach in two dimensions, then sketch a proof that the approach works and indicate how to generalize to d dimensions.

We can at least assume that all points in the input set are distinct, as we can always compress duplicates to a single point together with an indication of how many times that point appeared in the input.

We will define a map $f : \mathbb{R}^2 \to \mathrm{CoN}^2$. Here CoN is the "composite number space", which is just another name for $\mathbb{R}^2$ in the lexicographic order. We will represent a point in CoN by $(x \mid y)$, to distinguish from the point (x, y) in $\mathbb{R}^2$. Lexicographic order means that $(x \mid y) \geq (a \mid b)$ iff either (i) $x > a$; or (ii) $x = a$ and $y \geq b$. The map f is defined by $f((x, y)) = ((x \mid y), (y \mid x))$. Given a point p, we will denote $f(p)$ by $\hat{p}$.

Note that if $p \neq q$ then $\hat{p}$ and $\hat{q}$ will not have any coordinates in common. So, if we can describe range queries on a set R in terms of range queries on its image $\hat{R}$ under f, then we will have successfully dealt with the problem of duplicate coordinates. The following lemma tells us how to do just this:

Lemma 1. *A point p lies in $[x_1 : x_2] \times [y_1 : y_2]$ if and only if $\hat{p}$ lies in $[(x_1 \mid -\infty) : (x_2 \mid \infty)] \times [(y_1 \mid -\infty), (y_2 \mid \infty)]$.*

Proof sketch. Given a point p, let p_x and p_y denote its two coordinates. First note that $x_1 \leq p_x \leq x_2$ if and only if $(x_1 \mid -\infty) \leq (p_x \mid p_y) \leq (x_2 \mid \infty)$. Then note that $y_1 \leq p_y \leq y_2$ if and only if $(y_1 \mid -\infty) \leq (p_y \mid p_x) \leq (y_2 \mid \infty)$. These two observations together imply the lemma. □

To deal with the case of higher dimensions, we just let $f(p)$ be the point in CoN^d obtained by taking all cyclic shifts of the coordinates of p.

References

[BS75] Jon Louis Bentley and Donald F. Stanat. Analysis of range searches in quad trees. *Inf. Process. Lett.*, 3(6):170–173, 1975.

[CG86] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[dBea08] Mark de Berg et al. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 2008.

[Wil89] Robert E. Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM J. Comput.*, 18(1):56–67, 1989.

6.851 Advanced Data Structures
Spring 2010