6.851: Advanced Data Structures Spring 2012

Lecture 3 — February 23, 2012

Prof. Erik Demaine *Scribes: Brian Hamrick (2012)*

Ben Lerner (2012), Keshav Puranmalka (2012)

1 Overview

In the last lecture, we saw the concepts of persistence and retroactivity as well as several data structures implementing these ideas.

In this lecture, we are looking at data structures to solve geometric problems dealing with multiple dimensions, such as point location and orthogonal range queries.

In particular, we'll describe range trees, which allow us to solve the orthogonal range query problem in d dimensions with $O(\log^d n)$ query time. We then improve this to $O(\log^{d-1} n)$ query time using layered range trees, which use the technique of fractional cascading. We also discuss how to make these trees dynamic by using a general transformation, at the cost of a log factor and the query time being amortized.

These problems encompass applications such as determining which GUI element a user clicked on, what city a set of GPS coordinates is in, and certain types of database queries.

2 Planar Point Location

Planar point location is a problem in which we are given a planar graph (with no crossings) defining a map, such as the boundary of GUI elements. Then we wish to support a query that takes a point given by its coordinates (x, y) and returns the face that contains it (see Figure 1 for an example). As is often the case with these problems, there is both a *static* and *dynamic* version of this problem. In the static version, we are given the entire map beforehand and just want to be able to answer queries. For the dynamic version, we also want to allow the addition and removal of edges.

2.1 Vertical Ray Shooting

A closely related problem to planar point location is **vertical ray shooting**. Just as in planar point location, we are interested in a planar graph defining a map. In this case, when we query a point (x, y) we are interested in the first line segment lying above it. Equivalently, if we imagine shooting a vertical ray from the query point, we want to return the first map segment that it intersects (see Figure 2).

We can reduce the static planar point location problem to a vertical ray shooting problem by precomputing for each edge, what the facelying below it is.

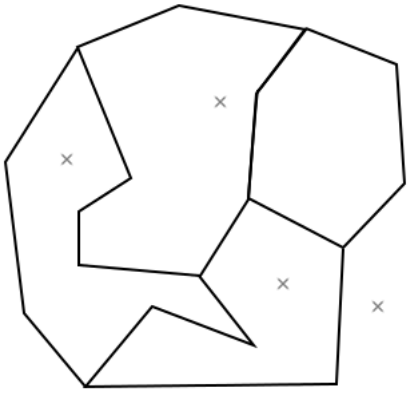

Figure 1: An example of a planar map and some query points

For the dynamic planar point location problem, we can again use this technique, but we need to maintain the face data dynamically, leading to an $O(\log n)$ additive overhead. The vertical ray shooting problem can be solved with a technique called a *line sweep*.

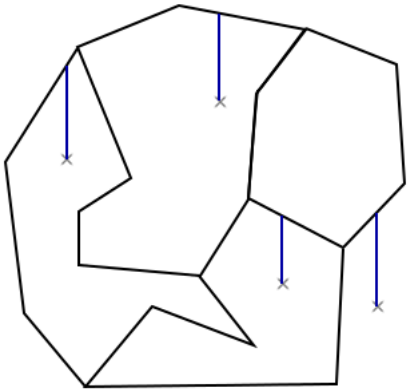

Figure 2: The same map and query points in the ray shooting view

2.2 Line Sweep

The **line sweep** is a relatively general technique for reducing the dimension of a geometric problem by 1. Given a geometric problem in d-dimensional space, we can consider a "vertical" $(d-1)$-dimensional crosssection, meaning a hyperplane perpendicular to the x_1 axis. Now we imagine our geometry problem within just that hyperplane and consider how it changes as we move the hyperplane along the x_1 axis. If this x_1 dimension has a suitable "timelike" interpretation, as it

does in vertical ray shooting, then a persistent or fully retroactive data structure for the $(d-1)$-dimensional problem will allow us to solve the d-dimensional problem.

In the case of vertical ray shooting, in one dimension we can solve the problem with a balanced binary search tree. More specifically, we are doing *successor* queries. In the last lecture we saw how to make a partially persistent balanced binary search tree with $O(\log n)$ time queries, and we know that a fully retroactive successor query structure can be made with $O(\log n)$ queries as well. Note, however, that in this case the successor structure only allows us to handle horizontal segments, as the comparison function between general segments depends on the x-coordinate. What happens when we apply line sweep to solve the two dimensional problem?

As our vertical crosssection sweeps from left to right, we first note that no two line segments ever change order because of the assumption that there are no crossings in our map. This means that the tree structure only changes at a few discrete times. In particular, when we reach the left endpoint of a segment we are performing an insertion of that segment, and when we reach the right endpoint we are performing a deletion.

Supposing we implement this line sweep using a partially persistent balanced binary search tree, to make a vertical ray shooting query for the point (x, y), we find the update corresponding to the x-coordinate and make a query (using persistence) for (x, y) in that version of the data structure. It is also useful to note that this structure can be computed in $O(n \log n)$ preprocessing time, as shown by Dobkin and Lipton in [8].

Additionally, if we use the fully retroactive successor data structure, we can solve the dynamic vertical ray shooting problem with *horizontal* segments with $O(\log n)$ time queries. See [4] and [10].

Several variants of the vertical ray shooting problem are still open. Examples include:

- **OPEN:** Can we do $O(\log n)$ dynamic vertical ray shooting in a general planar graph?
- **OPEN:** Can we do $O(\log n)$ static ray shooting when the rays do not have to be vertical? Note that the three dimensional version of this problem is motivated by ray tracing.

2.3 Finding Intersections

The line sweep method can also be used to find intersections in a set of line segments, and this problem gives a good illustration of the line sweep method.

Given a set of line segments in the plane defined by their endpoints, we wish to find all of the intersections between them. See Figure 3 for an example of the problem.

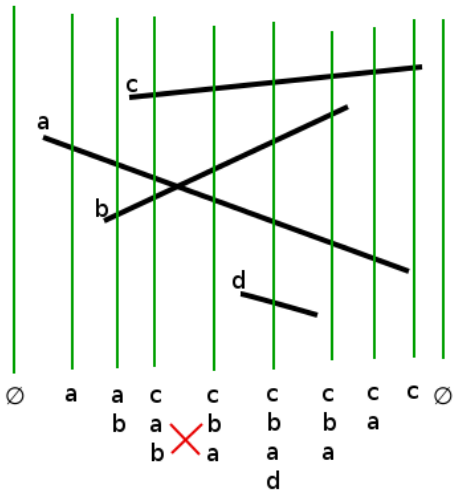

Figure 3: A line sweep for detecting intersections

To solve this problem, we use the line sweep technique where we store the line segments in a balanced binary search tree. The modifications to the search tree are as follows.

- The left endpoint of a segment causes us to insert that segment into the tree.
- The right endpoint of a segment causes us to delete that segment from the tree.
- Two segments crossing cause us to interchange their order within the tree.

We can use these ideas to solve the line segment intersection problem in $O(n \log n + k)$ when there are k intersections. To do this, we need to be able to efficiently determine the next time two segments would cross. We note that if a crossing would occur before we add or delete any more segments, it would have to involve two segments that are currently adjacent in the tree order. Then for each segment, we track when it would cross its successor in the tree and each internal node tracks the earliest crossing in its subtree. It is not difficult to maintain this extra data in $O(\log n)$ time per update, and performing a swap can be done in constant time.

3 Orthogonal range searching

In this problem we're given n points in d dimensions, and the query is determining which points fall into a given box (a box is defined as the cross product of d intervals; in two dimensions, this is just a rectangle). See Figure 4 for an example. This is, in a sense, the inverse of the previous problem, where we were given a planar graph, and the query was in the form of a point.

In the static version of the problem we can preprocess the points, while in the dynamic version points are added and deleted. In both cases, we query the points dynamically. The query has different versions:

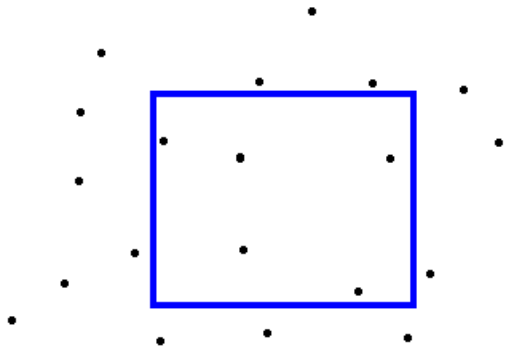

Figure 4: Orthogonal range searching

- Are there any points in the box? This is the "existence" version of the problem, and it's the easiest.
- How many points are in the box? This can solve existence as well.
- What are all the points in the box? Alternatively, what is a point, or what are ten points, in the box?

These questions are very similar, and we can solve them with the same efficiency. Once exception is the last question - for a given input, the answer may involve returning every point in the set, which would take $O(n)$ time. We're therefore looking for a solution with time complexity being something like $O(\log n + k)$, where k is the size of the output.

3.1 Range Trees

Let's start with the 1-dimensional case, $d = 1$. To solve it, we can just sort the points and use binary search. The query is an interval $[a, b]$; we can find the predecessor of a and the successor of b in the sorted list and use the results to figure out whether there are any points in the box; subtract the indices to determine the number of points; or directly print a list of points in the box. Unfortunately arrays don't generalize well, although we will be using them later.

We can achieve the same runtimes by using a structure called **Range Trees**. Range trees were invented by a number of people simultaneously in the late 70's [3], [2], [11], [12], [16].

We can build a range tree as follows. Consider a balanced binary search tree (BBST) with data stored in the leaves only. This will be convenient for higher dimensions. Each non-leaf node stores the min and max of the leaves in its subtrees; alternatively, we can store the max value in the left subtree if we want to store just one value per node.

Again, we search for PRED(a) and SUCC(b) (refer to Figure 5). As we search, we'll move down the tree and branch at a number of points. (As before, finding PRED(a) and SUCC(b) takes $O(\log n)$ time.) Once the paths to PRED(a) and SUCC(b) diverge, any time we turn left at a node while searching for PRED(a), we know that all the leaves of the right subtree of that node are in the given interval. The same thing is true for left subtrees of the right branch. If the left tree branches right or the right tree branches left, we don't care about the subtree of the other child; those leaves are

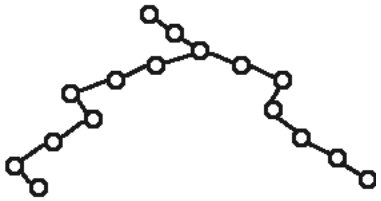

Figure 5: Branching path to PRED(a) and SUCC(b)

outside the interval. The answer is then implicitly represented as follows: If we store the size of

Figure 6: The subtrees of all the elements between a and b

each node's subtree in the node, we can compute the number of elements in our list in $O(\log n)$ time. To find the first k numbers, we can visit the first k elements after PRED(a) in $O(k)$ time (the operations in the previous two sentences might be easier to visualize with reference to Figure 6). These are the same results we received with arrays, but range trees are easier to generalize to higher d.

Let's look at the 2-D case, searching for points between a_1 and b_1 in x and a_2 and b_2 in y. We can build a range tree using only the x coordinate of each point, and then repeat the above procedure to figure out which points fall between a_1 and b_1.

We now want to sort the points by y coordinate. We can't use a range tree over all the points, because we're only interested in the points which we know are between a_1 and b_1, rather than every point in the set. We can restrict our attention to these points by creating, for each x subtree, to a corresponding one dimensional y range tree, consisting of all the points in that x subtree, but now sorted by their y-coordinate. We'll also store a pointer in each node of the x range tree to the corresponding y tree (see Figure 7 for an example). For example, α_x, β_x, and γ_x point to corresponding subtrees α_y, β_y, and γ_y. γ_x is a subtree of β_x in x, but γ_y is disjoint from β_y (even though γ_y and γ_x store the same set of points).

This structure takes up a lot of space: every point is stored in $\log n$ y range trees, so we're using $\theta(nlogn)$ space. However, we can now do search efficiently - we can first filter our point set by x coordinate, and then search the corresponding y range trees to filter by the y coordinate. We're searching through $O(\log n)$ y subtrees, so the query runs in $O(\log^2 n)$ time.

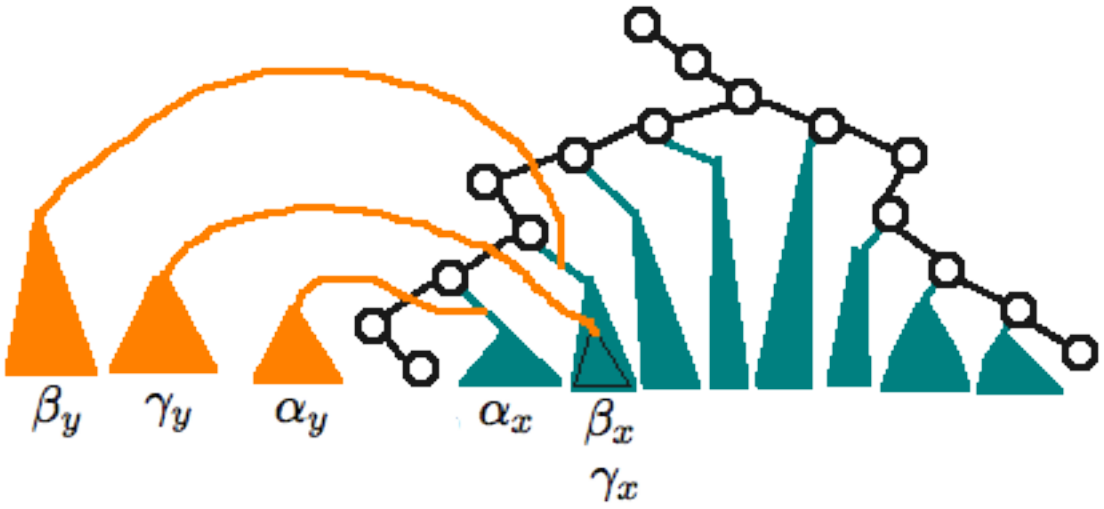

Figure 7: Each of the nodes at the x level have a pointer to all of the children of that node sorted in the y dimension, which is denoted in orange.

We can extend this idea to d dimensions, storing a y subtree for each x subtree, a z subtree for each y subtree, and so on. This results in a query time of $O(\log^d n)$. Each dimension adds a factor of $O(\log n)$ in space; we can store the tree in $O(n \log^{d-1} n)$ space in total. If the set of points is static, we can preprocess them in $O(n \log^{d-1})$ time for $d > 1$; sorting takes $O(n \log n)$ time for $d = 1$. Building the range trees in this time bound is nontrivial, but it can be done.

3.2 Layered Range Trees

We can improve on this data structure by a factor of $\log n$ using an idea called **layered range trees** (See [9], [17], [15]). This is also known as **fractional cascading**, or **cross linking**, which we'll cover in more detail later. The general idea is to reuse our searches to be able to search a value across k lists each with $O(n)$ elements in $O(\log n + k)$ instead of $O(k \log n)$.

In the 2-D case, we're repeatedly performing a search for the same subset of y-coordinates, such as when we search both α_y and γ_y for points between a_2 and b_2.

To avoid this, we do the following. Instead of a tree, store an array of all the points, sorted by y-coordinate (refer to Figure 8). As before, have each node in the x tree point to a corresponding subarray consisting of those points in the tree, also sorted by their y-coordinate.

We can do the same thing as before, taking $O(\log n)$ time to search through each array by y; but we can also do better: we can search through just one array in y, the array corresponding to the root of the tree, which contains every element, sorted by y.

We want to maintain this information as we go down the x range tree. As we move from a node v in the x tree to one of its children, say v_r, v_r will point to an array, sorted in y, that contains a subset of the y array v points to. This subset is arbitrary, as it depends on the x-coordinate of the points.

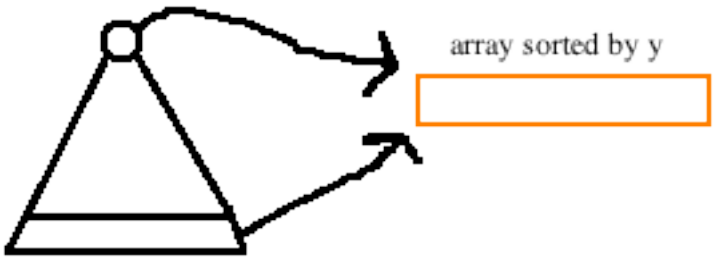

Figure 8: Storing arrays instead of range trees

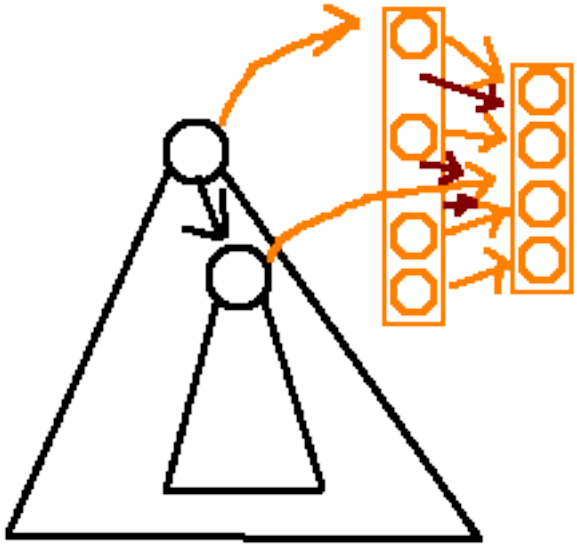

Figure 9: Here is an example of cascading arrays in the final dimension. The large array contains all the points, sorted on the last dimensions. The smaller arrays only contain points in a relevant subtree (the small subtree has a pointer to the small array). Finally, the elements in the large array has pointers to its "position" in the small array.

Let's store a pointer in each element of v's array to an element in v_r's array. If the element is in both arrays, it should point to itself; otherwise, let it point to its successor in v_r's array. (Each element will actually store two pointers, one to v_r's array and one to v_ℓ's array.) Then, in particular, we can find the predecessor and successor of a_2 and b_2 in any subtree by just following pointers. This lets you query the y-coordinate in $O(\log n + k)$ time, so we avoid the extra log factor from the previous strategy, allowing us to solve the 2-D problem in $O(\log n)$ time in total. For arbitrary d, we can use this technique for the last dimension; we can thereby improve the general query to $O(\log^{d-1} n + k)$ time for $d > 1$.

3.3 Dynamic Point Sets

We can make the previous scheme dynamic using amortization. In general, when we update a tree, we only change a few things, usually near the leaves. Updating a tree near its leaves takes constant time, as we're only changing a few things. Occasionally, we'll need to update a large section of the

tree, which will take a longer time, but this happens infrequently.

It turns out that if we have $O(n \log^{d-1} n)$ space and preprocessing time, we can make the structure dynamic for free using weight balanced trees.

3.4 Weight Balanced Trees

There are different kinds of weight balanced trees; we'll look at the oldest and simplest version, $BB[\alpha]$ trees [14]. We've seen examples of height-balanced trees: AVL trees, where the left and right subtrees have heights within an additive constant of each other, and Red-Black trees, where the heights of the subtrees are within a multiplicative factor of each other.

In a weight balanced tree, we want to keep the size of the left subtree and the right subtree roughly the same. Formally, for each node v we want

$$\text{SIZE}(\text{LEFT}(v)) \geq \alpha \cdot \text{SIZE}(v)$$

$$\text{SIZE}(\text{RIGHT}(v)) \geq \alpha \cdot \text{SIZE}(v)$$

We haven't defined size: we can use the number of nodes in the subtree, the number of leaves in the subtree, or some other reasonable definition. We also haven't selected α. If $\alpha = \frac{1}{2}$, we have a problem: the tree must be perfectly balanced at all times. Taking a small α, however, (say, $\alpha = \frac{1}{10}$), works well. Weight balancing is a stronger property than height balancing: a weight balanced tree will have height at most $\log_{1/(1-\alpha)} n$.

We can apply these trees to our layered range tree. [12][15] Updates on a weight balanced tree can be done very quickly. Usually, when we add or delete a node, it will only affect the nodes nearby. Occasionally, it will unbalance a large part of the tree; in this case, we can destroy that part of the tree and rebuild it. When we do so, we can rebuild it as a perfectly balanced tree.

Our data structure only has pointers in one direction - each parent points to its children nodes, but children don't point to their parents, or up the tree in general. As a result, we're free to rebuild an entire subtree whenever it's unbalanced. And once we rebuild a subtree, say for node v, we have $\text{SIZE}(\text{LEFT}(v)) = \frac{1}{2} \cdot k$, where k is the size of the subtree of v. In order to unbalance $\text{LEFT}(v)$, we need to make $\text{SIZE}(\text{LEFT}(v)) < \alpha \cdot k$. For $\alpha < \frac{1}{2}$, it's clear that we need $\theta(k)$ updates on v's subtree. The same applies for $\text{RIGHT}(v)$.

When we do need to rebuild a subtree, we can charge the process of rebuilding to the $\theta(k)$ updates we've made. Since each node we add can potentially unbalance every subtree it's a part of (a total of $\log(n)$ trees), we can update the tree in $\log(n)$ amortized time (assuming that a tree can be rebuilt in $\theta(k)$ time, which is easy).

So, for layered range trees, we have $O(\log^d n)$ amortized update, and we still have a $O(\log^{d-1} n)$ query.

3.5 Further results

For static orthogonal range searching, we can achieve a $O(\log^{d-1} n)$ query for $d > 1$ using less space: $O\left(n \frac{\log^{d-1}(n)}{\log \log n}\right)$ [5]. This is optimal in some models.

We can also achieve a $O(\log^{d-2} n)$ query for $d > 2$ using $O(n \log^d(n))$ space [6], [7]. A more recent result uses $O(n \log^{d+1-\epsilon}(n))$ space [1]. This is conjectured to be an optimal result for queries.

There are also non-orthogonal versions of this problem - we can query with triangles or general simplices, as well as boxes where one or more intervals start or end at infinity.

4 Fractional Cascading

Fractional cascading is a technique from Chazelle and Guibas in [6] and [7], and the dynamic version is discussed by Mehlhorn and Näher in [13]. It is essentially the idea from layered range trees put into a general setting that allows one to eliminate a log factor in runtime.

4.1 Multiple List Queries

To illustrate the technique of fractional cascading, we will use the following problem. Suppose you are given k sorted lists $L_1, \ldots, L_k$ each of length n, and you want to find the successor of x in each of them. One could trivially solve this problem with k separate binary searches, resulting in a runtime of $O(k \log n)$. Fractional cascading allows this problem to be solved in $O(k + \log n)$.

To motivate this solution we can look at the layered range trees above and think about how we can retain information when moving from one list to the next. In particular, if we were at a certain element of L_1, we could store where that element lies in L_2, and then quickly walk from one list to the next. Unfortunately, it might be the case that *all* of the elements of L_2 lie between two of the elements of L_1, in which case our position in L_1 doesn't tell us anything. So we should add some more information to L_1. This leads to the idea of fractional cascading.

Define new lists $L'_1, \ldots, L'_k$ by $L'_k = L_k$, and for $i < k$, let L'_i be the result of merging L_i with every other element of L'_{i+1}. Note that $|L'_i| = |L_i| + \frac{1}{2}|L'_{i+1}|$, so $|L_i| \leq 2n = O(n)$.

For each $i < k$, keep two pointers from each element. If the element came from L_i, keep a pointer to the two neighboring elements from L'_{i+1}, and vice versa. These pointers allow us to take information of our placement in L'_i and in $O(1)$ turn it into information about our placement in L_i and our placement in half of L'_{i+1}.

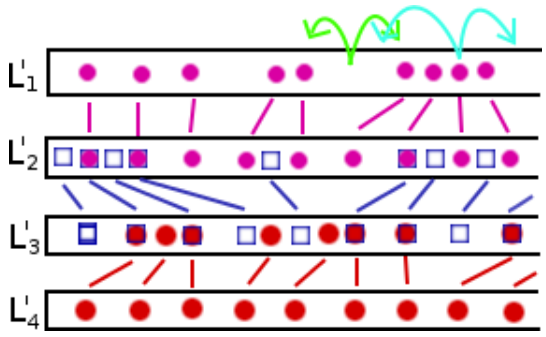

Figure 10: An illustration of fractional cascading

Now to make a query for x in all k lists is quite straightforward. First, query L'_1 for the location of x with a binary search. Now to find the location of x in L'_{i+1} from the location of x in L'_i, find the

two neighboring elements in L'_{i+1} that came from L'_i using the extra pointers. Then these elements have exactly one element between them in L'_{i+1}. To find our actual location in L'_{i+1}, we simply do a comparison with that intermediate element. This allows us to turn the information about x's location in L'_i into information about x's location in L'_{i+1} in $O(1)$ time, and we can retrieve x's location in L_i from its location in L'_i in $O(1)$, and thus we can query for x in all k lists in $O(k + \log n)$ time.

4.2 General Fractional Cascading

To generalize the above technique, we first note that we can replace "every other element" with "α of the elements, distributed uniformly", for any small α. In this case, we used $\alpha = \frac{1}{2}$. However, by using smaller α, we can do fractional cascading on any graph, rather than just the single path that we had here. To do this, we just (modulo some details) cascade α of the set from each vertex along each of its outgoing edges. When we do this, cycles may cause a vertex to cascade into itself, but if we choose α small enough, we can ensure that the sizes of the sets stays linearly bounded.

In general, fractional cascading allows us to do the following. Given a graph where

- Each vertex contains a set of elements
- Each edge j is labeled with a range $[a_j, b_j]$
- The graph has *locally bounded in-degree*, meaning for each $x \in \mathbb{R}$, the number of incoming edges to a vertex whose set contains x is bounded by a constant.

We can support a search query that finds x in each of k vertices, where the vertices are reachable within themselves from a single node of the graph so the edges used form a tree with k vertices, and such that $x \in [a_j, b_j]$ for each edge j. As before, we could do the search query in $O(k \log n)$ with k separate binary searches and fractional cascading improves this to $O(k + \log n)$, where n is the largest size of a vertex set.

References

[1] S. Alstrup, G. Stolting Brodal, T. Rauhe, *New data structures for orthogonal range searching*, Foundations of Computer Science, 2000. Proceedings, 41st annual symposium, 198-207.

[2] J.L. Bentley, *Multidimensional Binary Search Trees in Database Applications*, IEEE Transactions on Software Engineering, 4:333-340, 1979.

[3] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, *Computational Geometry: Algorithms and Applications*, Springer, 3rd edition, 2008.

[4] G. Blelloch, *Space-efficient dynamic orthogonal point location, segment intersection, and range reporting*, SODA 2008:894-903.

[5] B. Chazelle, *Reportin and counting segment intersections*, Journal of Computer and System Sciences 32(2):156-182, 1986.

[6] B. Chazelle, L. Guibas, *Fractional Cascading: I. A Data Structuring Technique*, Algorithmica, 1(2):133-162, 1986.

[7] B. Chazelle, L. Guibas, *Fractional Cascading: II. Applications*, Algorithmica, 1(2):163-191, 1986.

[8] D. Dobkin, R. Lipton, *Multidimensional searching problems*, SIAM Journal on Computing, 5(2):181-186, 1976.

[9] Gabow H., Bentley J., Tarjan R., *Scaling and related techniques for geometry problems*, Symposium on Theory of Computing, 1984. Proceedings, 16^{th} annual

[10] Y. Giyora, H. Kaplan, *Optimal dynamic vertical ray shooting in rectilinear planar subdivisions*, ACM Transactions on Algorithms, 5(3), 2009.

[11] D.T. Lee, C.K. Wong, *Quintary trees: a file structure for multidimensional database systems*, ACM Transactions on Database Systems, 5(3), 1980.

[12] G. Lueker, *A data structure for orthogonal range queries*, Foundations of Computer Science, 1978. Proceedings, 19^{th} annual symposium, 28-34.

[13] K. Mehlhorn, S. Näher, *Dynamic Fractional Cascading*, Algorithmica, 5(2): 215-241, 1990.

[14] J. Nievergelt, E. M. Reingold, *Binary search trees of bounded balance*, Symposium on Theory of Computing, 1972. Proceedings, 4^{th} annual symposium.

[15] D.E. Willard, *New Data Structures for Orthogonal Range Queries*, SIAM Journal on Computing, 14(1):232-253. 1985.

[16] D.E. Willard, *New Data Structures for Orthogonal Queries*, Techincal Report, January 1979.

[17] D.E. Willard, Ph.D. dissertation, Harvard University, 1978.