6.851: Advanced Data Structures Spring 2012

Lecture 4 — February 28, 2012

Prof. Erik Demaine *Scribes: Rajeev Parvathala (2017), Andrew He (2017), Brandon Tran (2012), Nathan Pinsker (2012), Ishaan Chugh (2012), David Stein (2010), Jacob Steinhardt (2010)*

1 Overview- Geometry II

We first show how to further utilize fractional cascading to remove yet another lg factor from d-dimensional orthogonal range searches, so that we can query in $\mathcal{O}(n \lg^{d-2} n)$. Recall that fractional cascading allow for:

- searching for x in k lists of length n in $\mathcal{O}(\lg n + k)$ time.
- this works even if the lists are found online by navigating a bounded-degree graph

Then we move on to the topic of kinetic data structures. These are data structures that contain information about objects in motion. They support the following three types of queries: (i) change the trajectory of an object; (ii) move forward to a specified point in time (advance); (iii) return information about the state of the objects in the current time. Examples we will go over are kinetic predecessor/successor and kinetic heaps.

2 3D Orthogonal Range Search in $\mathcal{O}(\lg n)$ Query Time

The work here is due to Chazell and Guibas [CG86].

In general we want to query on the section of our points in space defined by: $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$.

2.1 Step 1

We want to query $(-\infty, b_2) \times (-\infty, b_3)$, which we call the restricted two dimensional case. There are no left endpoints in our query, so the area we're querying basically looks like:

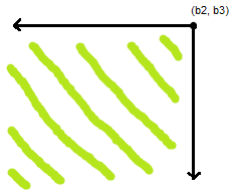

Figure 1: Query Area

We want to call points dominated by (b_2, b_3) in the yz-plane in time: $\mathcal{O}(k)$ + the time needed to search for b_3 in the list of z-coordinates. We transform this to a ray-stabbing query. The basic idea is to use horizontal rays to stab vertical rays, where the horizontal ray is our ray of query.

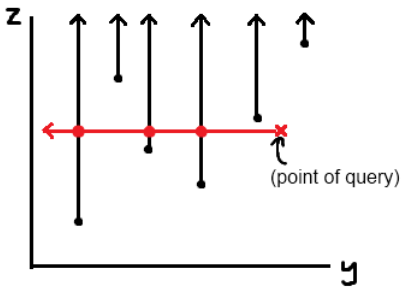

Figure 2: Stabbing Vertical Rays

A crossing indicates that the point whose vertical ray caused the crossing lies in the query range. What we want to do is walk along our horizontal ray and spend constant time per crossing, using only a single binary search on the y-coordinate initially, so that we can use fractional cascading later. We'll accomplish this by applying a (different) form of fractional cascading on this graph.

More specifically, for each vertical ray, we keep a horizontal line at the bottom point of the ray. This line is extended left and right until it hits another vertical ray. (These lines are blue in Figure 3.)

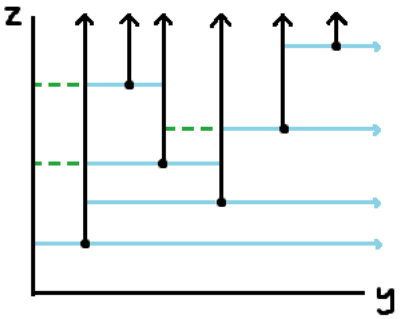

Figure 3: Extending Lines

When traversing, we binary search for the z value and then go from "face" to "face" in the picture. However, we want to bound the degree of the faces (the number of faces bordering it) to make the query more efficient. Thus we look at all the horizontal lines that end at some vertical ray and extend approximately half of them into the face to create more faces with lower degree (the green dotted lines in Figure 3). Since we only extend half, the number of lines decreases by a factor of 2 for every vertical ray. Since $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$, we only have an $\mathcal{O}(n)$ increase in space. This idea of extending the rays is analogous to the basic idea of fractional cascading: we are extending half of the rays left each time. Therefore, we are able to do a single search for b_3, then walk over the k points inside our range in $O(k)$. This construction is due to Chazelle [C86].

2.2 Step 2

Now we want to query $[a_1, b_1] \times (-\infty, b_2) \times (-\infty, b_3)$ in $\mathcal{O}(k) + \mathcal{O}(\lg n)$ searches. To do this, we use a one dimensional range tree on the x-coordinates, where each node stores the structure in Step 1 on points in its subtree. Recall that the query in Step 1 requires one binary search. However, we can actually avoid doing a binary search in all but one of our $\mathcal{O}(\lg n)$ searches.

We use the graphical form of fractional cascading on the z coordinate for our Step 1 data structures in the range tree. In particular, given the location of b_3 in our current node, we will be able to locate it in the subtree rooted at a child of our current node in $\mathcal{O}(1)$ time. Now, we can simply binary search for b_3 a single time at root of the range tree, and then find the location of b_3 as we walk down the range tree in $O(1)$ time per subtree.

2.3 Step 3

Now we query $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3)$. We do something similar to a range tree on the y-coordinate, where each node v stores a key $= \max(\text{left}(v))$, the structure in Step 2 on points in its right subtree, and the structure in Step 2, but inverted, on points in its left subtree. The inverted structure allows queries of $[a_1, b_1] \times (a_2, \infty) \times (-\infty, b_3)$. This can easily be done in the same way as before. At this point, we know that we can only afford a constant number of calls to the structures in Step 2.

We start by walking down the tree. At each node v, if $\text{key}(v) < a_2 < b_2$ we go to its right child. If $\text{key}(v) > b_2 > a_2$, we go to its left child. Finally, if $a_2 \leq \text{key}(v) \leq b_2$, we query on the Step 2 structure and the inverted Step 2 structure at this node. The reason this gives us all the points we want is that the structure in the left subtree allows for queries from a_2 to ∞. However, since this only stores points with y-coordinate less than or equal to $\text{key}(v)$, we get the points between a_2 and $\text{key}(v)$. Similarly, the structure in the right subtree allows for leftward queries, so we get the points between $\text{key}(v)$ and b_2. Thus we only needed two calls to the Step 2 structures and one $\lg n$ search (walking down the tree). We note that we could have used this idea for all three coordinates, but we didn't need to since range trees give us the x-coordinate for free (this requires a lg factor of extra space).

2.4 Step 4

Finally, we query $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$. We do the exact same thing as in Step 3, except now on the z-coordinates, and at each node we store the structure in Step 3.

2.5 Total Time and Space

This whole process required $\mathcal{O}(\lg n + k)$ time and $\mathcal{O}(n \lg^3 n)$ space. This means that in general, for $d \geq 3$ dimensions, orthogonal range search costs $\mathcal{O}(\lg^{d-2}(n) + k)$. The basic idea was that if using more space is not a problem, one-sided intervals are equivalent to two-sided intervals.

We can't use the same trick to get $O(\log n)$ queries on four dimensions, as the initial structure in

Step 1 doesn't generalize to more than 2 dimensions, while extending step 2 to 2 dimensions would require visiting $O(\log^2 n)$ range tree.

3 Kinetic Data Structures

The work here is due to Basch, Guibas, and Hershberger [BGH99].

Consider a video game, or a traffic simulation: different particles or objects move with different velocity, and at any point in time, we may want to query the nearest neighbor (predecessor) of a particular object. When we're not querying, we'd like to be able to advance time quickly. Also, we want to be able to account for the objects' changes in trajectories. This kind of use case motivates kinetic data structures.

More formally, the desired operations are:

- advance(t) : now $= t$
- change($x, f(t)$) : changes trajectory of x to $f(t)$

We will only be dealing with the following models of trajectories:

- affine: $f(t) = a + bt$
- bounded-degree arithmetic: $f(t) = \sum_{i=0}^{n} a_i t^i$
- pseudo-algebraic: any certificate of interest flips between true and false $\mathcal{O}(1)$ times

3.1 Approach

We'll define our approach using certificates. We define a certificate as a boolean function about the points. We'll try to pick certificates which tell us when the data structure ceases to be accurate. In more detail:

1. Store the data structure that is accurate now. Then queries about the present are easy.
2. Augment the data structures with certificates which store conditions under which the data structures is accurate and which are true now.
3. Compute the failure times when each certificate fails to hold (takes $\mathcal{O}(1)$ time).
4. Place the failure times into a priority queue. When we advance time, we look for certificates that fail and "fix" them.

3.2 Metrics

There are four metrics we generally use to measure the performance of a kinetic data structure:

- responsiveness — when an event happens (e.g. a certificate failing), how quickly can the data structure be fixed?
- locality — what is the most number of certificates any object participates in?
- compactness — what is the total number of certificates?
- efficiency — what is the ratio of the worst-case number of data structure events (disregarding modify) to the worst case number of "necessary" changes? (The notion of a "necessary change" is somewhat slippery. In practice we will define it on a per-problem basis.)

3.3 Kinetic Predecessor

The first kinetic data structure problem we will look at is *kinetic predecessor* (also called kinetic sorting). We want to support queries asking for the predecessor of an element (assuming the elements are sorted by value). We take the following approach:

1. Maintain a balanced binary search tree.
2. Let $x_1, \ldots, x_n$ be the in-order traversal of the BST. Keep the certificates $\{(x_i < x_{i+1}) \mid i = 1, \ldots, n-1\}$.
3. Compute the failure time of each certificate as $\text{failure\_time}_i := \inf\{t \geq \text{now} \mid x_i(t) > x_{i+1}(t)\}$. For example, if the motion of each object is linear, compute the first time after the current point at which two lines intersect.
4. Implement *advance*(t) as follows:

```
while t >= Q.min:
  now = Q.min
  event(Q.delete-min)
now = t

define event (certificate (x[i] < x[i+1])):
  swap(x[i], x[i+1]) in BST
  add certificate (x[i+1] <= x[i])
  replace certificate (x[i-1] <= x[i]) with (x[i-1] <= x[i+1])
  replace certificate (x[i+1] <= x[i+2]) with (x[i] <= x[i+2])
```

Each time a certificate fails, we remove it from the priority queue and replace any certificates that are no longer applicable. It takes $\mathcal{O}(\lg n)$ time to update the priority queue per event.

The problem is that there may be $\mathcal{O}(n^2)$ events. To see this, note that if the elements initially start in some order, but eventually end up in the *reverse* order, then each pair of elements switches exactly once. Therefore, the total number of events must have been $\binom{n}{2} = \Omega(n^2)$.

We now analyze according to our metrics.

Efficiency If we need to "know" the sorted order of points, then we will need an event for every order change. Each pair of points can swap a constant number of times, yielding $\mathcal{O}(n^2)$ possible events. Therefore the efficiency is $\mathcal{O}(1)$.

Responsiveness Because we used a BBST, we can fix the constant number of certificate failures in $\mathcal{O}(\lg n)$ time.

Local Each data object participates in $\mathcal{O}(1)$ certificates. In fact, each participates in at most 2 certificates.

Compact There were $\mathcal{O}(n)$ certificates total.

3.4 Kinetic Heap

The work here is due to de Fonseca and de Figueiredo [FF03]

We next consider the kinetic heap problem. For this problem, the data structure operation we want to implement is $find_{min}$. We do this by maintaining a heap (for now, just a regular heap, no need to worry about Fibonacci heaps). Our certificates check whether each node is smaller than its two children in the heap. Whenever a certificate breaks, we simply apply the heap-up operation to fix it, and then modify the certificates of the surrounding nodes in the heap appropriately. In this case, our data structure has $\mathcal{O}(\lg n)$ responsiveness, $\mathcal{O}(1)$ locality, and $\mathcal{O}(n)$ compactness. The only non-obvious metric is efficiency. We show below that the efficiency is in fact $\mathcal{O}(\lg n)$ (by showing that the total number of events is $\mathcal{O}(n \lg n)$).

Analyzing the number of events in kinetic heap. We will show that there are at most $\mathcal{O}(n \lg n)$ events in a kinetic heap using amortized analysis. For simplicity, we will carry through our analysis in the case that all functions are linear, although the general case works the same.

Define $\Phi(t, x)$ as the number of descendents of x that overtake x at some time after t.

Define $\Phi(t, x, y)$ as the number of descendants of y (including y) that overtake x at some time greater than t. Clearly, $\Phi(t, x) = \Phi(t, x, y) + \Phi(t, x, z)$, where y and z are the children of x.

Finally, define $\Phi(t) = \sum_x (\Phi(t, x))$. Φ will be the potential function we use in our amortized analysis. Observe that $\Phi(t)$ is $\mathcal{O}(n \lg n)$ for any value of t, since it is at most the total number of descendants of all nodes, which is the same as the total number of ancestors of all nodes, which is $\mathcal{O}(n \lg n)$. We will show that $\Phi(t)$ decreases by at least 1 each time a certificate fails, meaning that certificates can fail at most $\mathcal{O}(n \lg n)$ times in total.

Consider the event at time t^+ when a node y overtakes its parent x, and define z to be the other child of x. The nodes x and y exchange places, but no other nodes do. This means that the only changes to any of the potential functions between t and t^+ are:

$$\Phi(t^+, x) = \Phi(t, x, y) - 1$$

and

$$\Phi(t^+, y) = \Phi(t, y) + \Phi(t, y, z).$$

Since $y > x$ now, we also see that

$$\Phi(t, y, z) \leq \Phi(t, x, z).$$

Finally, we need that

$$\Phi(t, x, z) = \Phi(t, x) - \Phi(t, x, y).$$

Then

$$\begin{aligned}
\Phi(t^+, x) + \Phi(t^+, y) &= \Phi(t, x, y) - 1 + \Phi(t, y) + \Phi(t, y, z) \\
&\leq \Phi(t, x, y) - 1 + \Phi(t, y) + \Phi(t, x, z) \\
&= \Phi(t, x, y) - 1 + \Phi(t, y) + \Phi(t, x) - \Phi(t, x, y) \\
&= \Phi(t, y) + \Phi(t, x) - 1
\end{aligned}$$

From these, it follows that:

$$\Phi(t^+) \leq \Phi(t) - 1,$$

which completes the analysis. We conclude that the total number of data structure modifications is $\mathcal{O}(n \lg n)$.

3.5 Other Results

We now survey the results in the field of kinetic data structures. For a more comprehensive survey, see [Gui04].

2D convex hull. (Also diameter, with, and minimum area/perimeter rectangle.) Efficiency: $\mathcal{O}(n^{2+\epsilon})$, $\Omega(n^2)$ [BGH99]. **OPEN:** 3D case.

Smallest enclosing disk. $\mathcal{O}(n^3)$ events. **OPEN:** $\mathcal{O}(n^{2+\epsilon})$?

Approximate results. We can $(1+\epsilon)$-approximate the diameter and the smallest disc/rectangle in $\frac{1}{\epsilon}^{\mathcal{O}(1)}$ events [AHP01].

Delaunay triangulations. $\mathcal{O}(1)$ efficiency [AGMR98]. **OPEN:** how many total certificate changes are there? It is known to be $\mathcal{O}(n^3)$ and $\Omega(n^2)$.

***Any* triangulation.** $\Omega(n^2)$ changes even with Steiner points [ABdB+99]. $\mathcal{O}(n^{2+\frac{1}{3}})$ events [ABG+02]. **OPEN:** $\mathcal{O}(n^2)$ events? We can achieve $\mathcal{O}(n^2)$ for pseudo-triangulations.

Collision detection. See the work done in [KSS00], [ABG+02], and [GXZ01].

Minimal spanning tree. $\mathcal{O}(m^2)$ easy. **OPEN:** $o(m^2)$? $\mathcal{O}(n^{2-\frac{1}{6}})$ for H-minor-free graphs (e.g. planar) [AEGH98].

References

[ABdB+99] Pankaj K. Agarwal, Julien Basch, Mark de Berg, Leonidas J. Guibas, and John Hershberger. Lower bounds for kinetic planar subdivisions. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 247–254, New York, NY, USA, 1999. ACM.

[ABG+02] Pankaj K. Agarwal, Julien Basch, Leonidas J. Guibas, John Hershberger, and Li Zhang. Deformable free-space tilings for kinetic collision detection. *I. J. Robotic Res.*, 21(3):179–198, 2002.

[AEGH98] Pankaj K. Agarwal, David Eppstein, Leonidas J. Guibas, and Monika Rauch Henzinger. Parametric and kinetic minimum spanning trees. In *FOCS*, pages 596–605, 1998.

[AGMR98] Gerhard Albers, Leonidas J. Guibas, Joseph S. B. Mitchell, and Thomas Roos. Voronoi diagrams of moving points. *Int. J. Comput. Geometry Appl.*, 8(3):365–380, 1998.

[AHP01] Pankaj K. Agarwal and Sariel Hal-Peled. Maintaining approximate extent measures of moving points. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 148–157, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[BGH99] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *J. Algorithms*, 31(1):1–28, 1999.

[C86] Bernard Chazelle. Filtering Search: A new approach to query-answering. In *Siam J. Comput., Volume 15*, 1986.

[CG86] Bernard Chazelle, Leonidas J. Guibas, Fractional cascading: I. A data structuring technique. In *Algorithmica, Volume 1*, 1986.

[FF03] Guilherme D. da Fonseca, Celina M.H. de Figueiredo, Kinetic heap-ordered trees: Tight analysis and improved algorithms. In *Information Processing Letters, Volume 85, Issue 3*, 14 February 2003.

[Gui04] Leonidas J. Guibas. Kinetic data structures. 2004.

[GXZ01] Leonidas J. Guibas, Feng Xie, and Li Zhang. Kinetic collision detection: Algorithms and experiments. In *ICRA*, pages 2903–2910, 2001.

[KSS00] David Kirkpatrick, Jack Snoeyink, and Bettina Speckmann. Kinetic collision detection for simple polygons. In *SCG '00: Proceedings of the sixteenth annual symposium on Computational geometry*, pages 322–330, New York, NY, USA, 2000. ACM.