CS 573: Topics in Algorithms: Advanced Data Structures **Spring 2006**

Lecture 11 — February 21 and 23

Lecturer: Jeff Erickson *Scribe: Tracy Grauman*

Up until now, we have been talking about maintaining data structures which could handle such familiar requests as asking it a query or requesting that it change its structure in some discrete way. Now we are going to change our focus to look at *kinetic data structures*, which change continuously. Basch*, Guibas, and Hershberger [1],[4] were the first ones to nail down the general rules for kinetic data structures in that paper.

11.1 Kinetic Data Structures - General Rules

- **Values/weights/coordinates are continuous functions of time.**

  We need to be careful with our usage of the word "time" in this context. "Time" doesn't refer to computational cost (ie: counting CPU operations), but rather denotes a more environmental sense of the word, as marked by a clock on the wall.

- **Maintain the data structure that would be built at any given instant.**

  As an example, a kinetic data structure should be able to return to you if a node v exists now. Or if it exists now. How about right *now*?

- **The data structure changes discretely.**

  We want to imagine our data structure as a function of time, not that there's just some Genie popping in and rewriting data at will. That is, for the duration of some interval, our data structure remains fixed. Then, at some discrete moment, the structure changes as the result of some functions stored at its nodes.

- **Maintain a proof of correctness for the data structure. When a certificate fails, update the data structure and its proofs.**

  Updates to correct failed certificates are assumed to take no wall-clock time, though it obviously must use some cpu cycles. Essentially, one would have to ensure that updates are resolved more quickly than the time it takes for the next query to reach the expired parts of the structure.

Evaluation Criteria for Kinetic Data Structures

1. **Responsive:** When a certificate expires, we can update the data structure quickly (in $O(\log n)$ time)
2. **Local:** No object participates in many certificates, resulting in fast motion changes.
3. **Compact:** The number of certificates is small compared to a static data structure.
4. **Efficient:** $\frac{\text{\# of events in data structure in the worst case}}{\text{\# of necessary changes to keep maintained in worst case}}$ (a particularly fuzzy area)

11.2 A Kinetic Dictionary

Consider supporting two queries to a kinetic dictionary,

FIND(t, x): Asks "Is there a node whose value at time t is equal to x?"

UPDATE(t, v, y): Change function at node v to $y : \mathbb{R} \to \mathbb{R}$, where the value of the function of node v at time t, $x_v(t) = y(t)$.

We want to store our binary search tree, where the keys are functions. At any instant, the tree should be in sorted order. That is, for an adjacent pair $(x_i, x_{i+1}), x_i < x_{i+1}$. We want to compute the first moment, t, when this inequality doesn't hold. Thus, maintain in a priority queue the result of the following function:

$$t(x_i, x_{i+1}) = \min\{t | x_i(t) \geq x_{i+1}(t)\}$$

(Note: Strictly speaking, we should use "infimum" instead of "minimum.")

When this happens, surgery is needed on our binary tree, resolved by reassigning parent-child pointers. We must make one of two assumptions: either that at most one event happens at any instant or that we have a mechanism for ordering concurrent events.

EVENT:
  //(x_i, x_{i+1}) expired
  swap x_i and x_{i+1} in binary search tree
  insert $x_{i+1}, x_i)$ with new certificate
  update certificates for $(x_{i+1}, x + i)$ and (x_{i+1}, x_{i+2})

As mentioned, the evaluation of efficiency can be a fuzzy area, and here we have one example of this. In one way, our algorithm is very efficient, as it only does necessary updates. On the other hand, if we never ask the structure *any* queries, then we never really *need* to keep the structure up-to-date, making any updates arguably "inefficient."

11.3 Kinetically Maintaining the Largest Element

11.3.1 Two Not-So-Great Ideas

Maintain Entire Sorted Order

We could maintain the entire sorted order of all our elements. Clearly, though, this is simply not efficient, as the maximum number of changes is $O(\binom{n}{2})$.

Keep Track of the Upper Hull

Assuming we know all of our functions in advance, we could plot them on a graph and look at the upper hull of the graph. However, if one of the functions should change its trajectory, we may have to recalculate the upper hull, a costly operation.

11.3.2 Kinetic tournament

In a *kinetic tournament*, our data structure consists of a balanced binary tree with n unordered leaves, one for each item. The value stored at each internal node is the larger of the values at its children, creating a certificate for each sibling comparison.

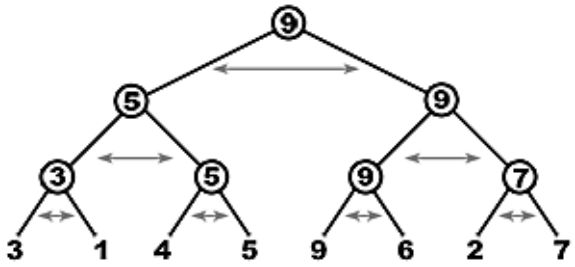

Figure 11.1. A kinetic tournament and its comparisons

As data changes, events are triggered when siblings are equal. These "ties" are merely transient occurrences – in the next instant, the faster-growing of the two children will overtake the other, causing their parent (and potentially all subsequent parents) to change identities. Indeed, this method measures up as "reasonably acceptable" in all criteria:

- **Compact**: $O(n)$
- **Local**: A winner will be involved in $O(\log n)$ certificates
- **Responsive**: $O(\log^2 n)$ per event
- **Efficient**: Assuming that we are changing linearly in the lifetime of the data structure, the winner changes in $\leq n$ events, making a total of $O(n \log n)$ events.

11.3.3 Kinetic Heaps: Generic Flavor

In a *kinetic heap*, we maintain a balanced min-heap-ordered binary tree, with certificates showing comparisons that every node is bigger than its parent. An event is swapping out a node with its parent. This structure is clearly compact, local, and responsive, but it took a while for it to be shown that it is also efficient.

- **Compact**: $O(n)$
- **Local**: $O(1)$
- **Responsive**: $O(\log n)$ per event
- **Efficient**: As we analyzed the tournament, a Kinetic Heap must be $\Omega(n \log n)$. First, Basch*, Guibas, and Ramkumar [2] proved $O(n \log^2 n)$ events, but later da Fonseca* and de Figuerdo [3] proved $O(n \log n)$ events, as described below.

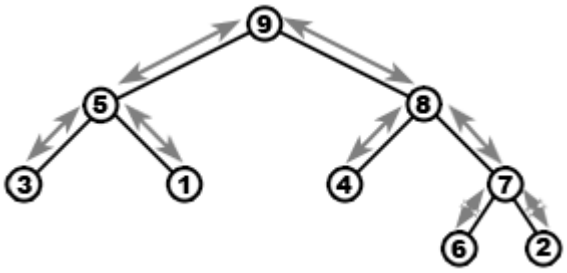

Figure 11.2. A kinetic heap and its comparisons

Proving the Efficiency of Kinetic Heaps

Let $\Phi(t)$ be the number of potential events which could occur in a tree t. Then

$$\begin{aligned}\Phi(t) &= \text{external path length}\\ &= \textstyle\sum_v \text{number of proper descendents of v}\\ &= \textstyle\sum_v \text{number of proper ancestors of v}\\ &= \text{number of edges in transitive closure}\end{aligned}$$

Let $\Delta(t, x)$ be the number of descendants of x at time t that will overtake x in the future (that is, at time $\geq t$), and let $\Delta(t) = \sum_x \Delta(t, x)$. Trivially, $\Delta(t)$ is at most the number of descendants of x, so $\Delta(t) \leq \Phi(t)$.

Let $\Delta(t, x, y)$ be the number of descendants of y at time t that will overtake x at time $> t$. Then $\Delta(t, x) = \sum_{\text{child } y \text{ of } x} \Delta(t, x, y)$.

Finally, let t^- be the old potential of y and let t^+ be the new potential of y.

Consider the following parent-child relationship of nodes x and y, which has the potential to change. The new potential of y after the change can be defined in terms terms of the old potential of x. Also notice that if a descendent of y should overtake y in the future, it must also overtake x in the future.

Suppose at time t, $x(t) = y(t)$ for some $x = \text{parent}(y)$.

$$\begin{aligned}\Delta(t^-, z) &= \Delta(t^+, z) \text{ for all } z \neq x, y\\ \Delta(t^+, x) &= \Delta(t^-, x, y) - 1 \text{ after swap}\\ \Delta(t^+, y) &\leq \Delta(t^-, y) + \textstyle\sum_z \Delta(t^-, x, z), \text{ where } z \text{ are children of } x \text{ (before swap)}, z \neq y\\ &\leq \Delta(t^-, y) + \Delta(t^-, x) - \Delta(t^-, x, y)\end{aligned}$$

Thus, $\Delta(t^+) \leq \Delta(t^-) - 1$; that is, every event decreases the potential by 1. We started with potential $\Phi(t)$, and thus the number of events after time t is $\leq \Delta(t) \leq \Phi(t)$.

Therefore in the linear-motion case, our efficiency is $O(n \log n)$. Unfortunately, this analysis doesn't work for higher-order motion.

11.3.4 Kinetic Heaps: Customized Flavors

Heater [2]

The basic idea of a *heater* is to give every object a random key, and keep the keys sorted left to right. In some ways, the data structure is akin to a reverse heap – a node has a priority but also a random search key. A good implementation of a heater is as a binary search tree. Its depth will be $O(\log n)$ with high probability. Note also that it is difficult both to split a heater or to merge two heaters.

Hanger [3]

A *hanger* is built from scratch from the empty set, inserting elements in decreasing order of priorities and using a random coin flip to decide whether to put the element in a left or right subtree. As opposed to heaters, the operations of splitting a tree or merging two trees are trivial to hangers.

11.4 Finding the Closest Pair for Points in the Plane

11.4.1 Using Voronoi Diagrams

Given a set of points in the plane, the closest pair of points can be computed in $O(n \log n)$ by using a Voronoi diagram. However, how can one maintain which pair is the closest as the points move around in the plane? There are potentially $\Omega(n^2)$ changes of which pair is the closest, as illustrated by figure 11.3.

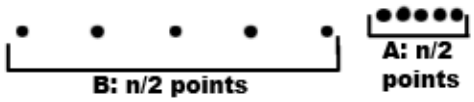

Figure 11.3. If set A moves left, n^2 comparisons will be made with set B

If one considers maintaining the Voronoi diagram, we have a solution that is indeed compact, local-ish (one cell may be complicated, but on average will be $O(1)$), and responsive. However, its efficiency is $\Omega(n^2)$ and has thus far been proven to be $O(n^{3+\epsilon})$ [Micah-Sharir].

11.4.2 Using Yao Graphs [1]

The *Yao Graph*, $Y(P)$, is a directed graph with $\leq 6n$ edges, where closest pairs create edges. We create this graph by partitioning the plane into six wedges using three lines through a point p which meet at 60° angles. Then "neighbors" of p are found by drawing equilateral triangles in each wedge such that every triangle contains only one other point besides p. We will show the algorithm for maintaining closest pairs for one wedge, and the procedure for other wedges follows symmetrically.

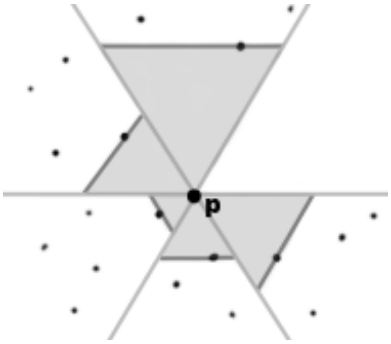

Figure 11.4. Constructing a Yao Graph for point p

Lemma 11.1. *Y(P) contains the closest pair of points.*

Proof: Refer to figure 11.5. Suppose that the closest pair consists of points p and r, but that edge pr is not in $Y(P)$. Let q be a Yao neighbor of p. Then $|qr| < |pr|$, a contradiction. □

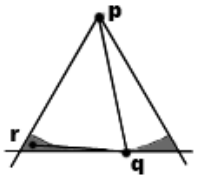

Figure 11.5. The Yao Graph must contain the closest pair of points

Construct the graph by sweeping a line over the points in $O(n\log n)$ time. When we find a new point, we shoot a ray from the point down along its 60° wedge. Assuming that we have kept track of the "skyline" of the "mountain range" already created by such wedges, we find where the rays from the new point hit the old skyline. To know where this happens, we only need to know the ordering of the sides of the peaks. We maintain the skyline in a balanced binary tree, adding each point in $O(\log\ n)$ time, giving a total for adding all n points of $O(n\ \log\ n)$ time. Additionally, we keep track of the middle "skyline trees" for use in later steps.

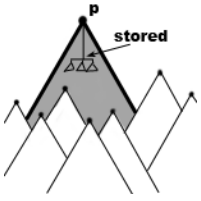

Figure 11.6. Adding a point to the skyline

We can certify everything with $\leq 3n$ comparisons, one comparison in each of our three directions for each point. There are only two ways in which our picture could change: along the y-axis, or along the 60°-axis.

Change of y-coordinate

Find and split and merge, all in $O(\log n)$ time.

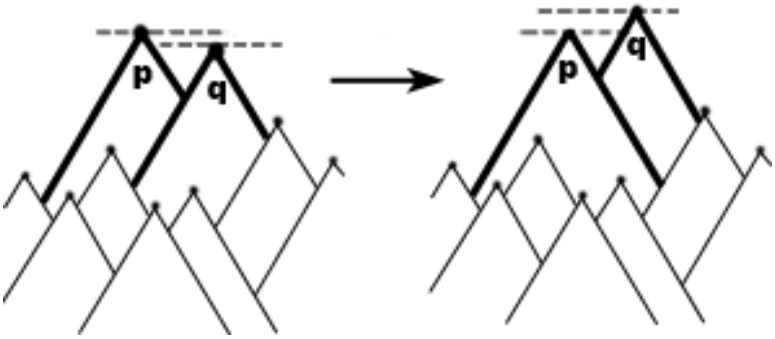

Figure 11.7. Change of y-coordinate

Change of 60°-coordinate

- **Hide**: Delete from parent(p), insert into p ($O(\log n)$ time)
- **Expose**: Delete from p, insert into parent(p) ($O(\log n)$ time)

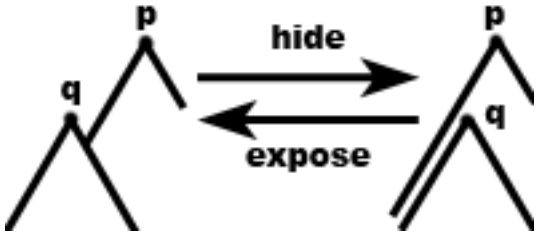

Figure 11.8. Change of 60°-coordinate

Analysis

- **Compact**: $O(n)$ space ✓
- **Local**: $O(1)$ - every point in only 6 certificates ✓
- **Responsive**: $O(\log n)$ time per event ✓
- **Efficient**: $O(n^2)$ events (for psuedo-algebraic motion) ✓

As a final note, Yao Graphs can be generalized to higher dimensions, though not all cases have been analyzed. For example, in three dimensions, one can consider maintaining on the order of twenty pyramids in place of maintaining six triangles.

Bibliography

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

[2] Julien Basch*, Leonidas Guibas, and G. D. Ramkumar*. Reporting red-blue intersections between two sets of conencted line segments. *Algorithmica* 35:1-20, 2003.

[3] Guilherme D. da Fonseca* and Celina M. H. de Figueiredo. Kinetic heap-ordered trees: tight analysis and faster algorithms. *Information Processing Letters* 85(3):165-169, 200

[4] Leonidas Guibas. Kinetic data structures. Chapter 23 of *Handbook of Data Structures and Applications* (Dinesh P. Mehta and Sartaj Sahni, eds.), 23-123-18. Chapman and Hall/CRC, 2005.