Kinetic Data Structures

Alex Broihier

Outline

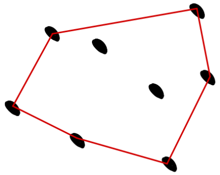
Section 1

Background

Data Structures

- Data structures store data, we will focus on data structures that store numbers or points
- Arrays - sorting elements in a list takes $O(n \log n)$ time, we will work with statically sized arrays
- Binary Trees - without any balancing, they have an expected height of $O(\log n)$ but can be $O(n)$ in the worst case
  - Heaps / Priority Queues - $O(\log n)$ time to both add an element and to remove the element with least priority

Convex Hulls



- A set is convex if for any two points in the set, if we draw a line segment between those two points, the line segment is entirely within the set
- The convex hull of a set of points is the smallest convex set that contains all of the points

Convex Hulls Continued

- When we compute the convex hull of n points, we really care about computing the set of points that form the vertices of the convex hull
- In 2D, the convex hull of n points can be computed in $O(n \log n)$ time (for instance, by Chan's Algorithm)

Section 2

Kinetic Data Structures

Points as a Function of Time

- When you first learn to code, you might expect $y = 0; x = y + 2; y = 12; print(x)$; to print out 14
- In many settings, values change and we want to update dependent values accordingly
- Instead of having data structures that store variables that have a constant value, we want to store variables that are constantly changing
- New system, x = f(t)

Kinetic Data Structure Interface

- Instead of storing x, store $x = f(t)$. We often assume that f is continuous (and often an easy to analyze function)
- $\mathit{SetTime}(t)$ updates the current time of the kinetic data structure to t. We assume that time can only increases
- $\mathit{UpdateTrajectory}(x, f'(t))$ changes the trajectory of x. We will not work with this, but it is something you can do
- The rest of the interface for the kinetic data structure is the same as the base data structure

Kinetic Data Structure Approach 1

- Upon calling $\mathit{SetTime}(t)$, we recalculate the value of each element in the data structure using t
- We then rebuild the data structure from scratch using the values of the new nodes

Example: Kinetic Sorted List

- We build a kinetic sorted list by sorting an input list of integers in $O(n \log n)$ time
- Upon calling *SetTime*(t)
  - For each element we update its value, spending $O(n)$ time total (assuming it takes $O(1)$ time to evaluate each function)
  - Then we rebuild the kinetic sorted list in $O(n \log n)$ time

Analysis

- Every *SetTime* call takes $O(n \log n)$ time
- If no elements need to be reordered, so we could have just recomputed their values in $O(n)$ time
    - Or even better, we only compute values when the data structure is actually queried

Kinetic Data Structure Approach 2

- Data structures work by upholding internal invariants
  - For a sorted list, this invariant is that an element is less than its right neighbor
  - For a heap, this invariant is that an element at a node is less than that of its two children
- For each internal invariant in our kinetic data structure, we introduce a "certificate" that the invariant is upheld
- When a certificate is no longer true, we update the data structure at the nodes involved and create a new valid certificate
- An event is a certificate failure

Certificate Approach

- Assume we can evaluate when each certificate will first fail
- If the first certificate to fail will fail at time t_0, and we advance time to $t < t_0$, we do nothing. Otherwise we fix the data structure, removing failed certificates and adding new ones
- We want to handle certificates in the order they fail: store certificates in a priority queue
- Upon calling $\mathit{SetTime}(t)$, while the top element of the certificate priority queue has a time to fail less than t
  - We pull it off the priority queue
  - Fix the data structure
  - Remove old certificates and add new certificates

Example: Improved Kinetic Sorted List

- For pair of neighboring elements a and b in our list (with a to the left of b), we introduce a certificate that $a < b$ and add these certificates to our priority queue ($O(n)$ certificates total)
- For each certificate that fails $a > b$, so we swap a and b in $O(1)$ time (and spend $O(\log n)$ time updating certificates)

Kinetic Data Structure Metrics

- There are four classic metric for evaluating kinetic data structures
- Responsiveness - time to update a kinetic data structure when a certificate fails
- Locality - max number of certificates per element
- Compactness - total number of certificates
- Efficiency - worst case number of events vs worst case number of changes as time increases

Metrics for Sorted List

- For the sorted list example:
- Responsiveness - we spend $O(\log n)$ time per certificate failure
- Locality - each element is a part of at most two certificates, so we have worst case $O(1)$ certificates per item
- Compactness - for n elements, we have $n-1$ certificates, so there are $O(n)$ certificates total
- Efficiency - For every n changes to the list, $O(n)$ events occur

Questions?

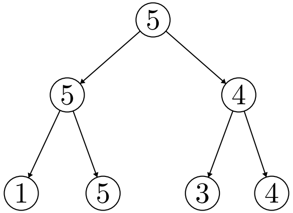
Section 3

Kinetic Heaps

Maximum Element Problem

- Possible motivation: we want to track the kinetic maximum of a set
- One approach: use a kinetic sorted list
  - But we don't really need to store an exact ordering of all the nodes
  - The efficiency is now worse - with linear functions the max values changes at most $O(n)$ times but there can be $O(n^2)$ events
- Another approach: use a heap where the maximum value appears at the root

Kinetic Tournaments

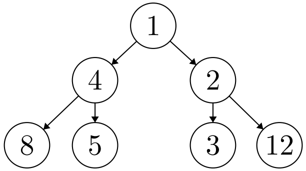


- A kinetic tournament is a binary tree where the leaves are the values, and each internal node is the larger value (the "winner") between its children
    - Hence this resembles a tournament
- Our certificates are of the form $a < b$ between pairs of children
- Now, if the minimum element becomes the maximum, we have $O(\log n)$ changes to the data structure

Kinetic Tournament Analysis

- Responsiveness - worst case $O(\log^2 n)$
- Locality - the root value is in $O(\log n)$ certificates
- Compactness - $O(n)$ (most values are not in as many certificates as the root)
- Efficiency - for linear functions, we have up to $O(n)$ changes to winner over up to $O(n \log n)$ events

Classic Kinetic Heap



- Maintain a priority queue separate from the certificate priority queue
- Each node has a certificate with its children nodes: the parent node is less than the children

Kinetic Heap Analysis

- Responsiveness - $O(\log n)$
- Locality - $O(1)$
- Compactness - $O(n)$
- Efficiency is harder to analyze

Kinetic Heap Efficiency

- Similar to the kinetic tournament, for linear functions $O(n)$ changes to the minimum element can cause at least $O(n \log n)$ events
- But it is harder to see that this is an upper bound, because unlike the kinetic tournament an element can "go down the other side of the tree"

Kinetic Heap Efficiency Continued

- Let $\Delta(x)$ be the number of descendants of x that x will swap with in the future
- At $t = -\infty$, $\sum_x \Delta(x) = O(n \log n)$ (from solving the recurrence $T(n) = O(n) + 2T(\frac{n}{2})$)
- When an event occurs where x swaps with its child y, $\Delta(x)$ becomes the old value of $\Delta(y)$, and $\Delta(y)$ becomes at most one less than the old value of $\Delta(x)$
- Thus each event can only decrease $\sum_x \Delta(x)$, so we are limited to at most $O(n \log n)$ events

Kinetic Heap Variants

- Kinetic Hanger - same as a regular kinetic heap except that the underlying tree can be unbalanced
  - In the expected case, the kinetic hanger will be balanced, yielding similar performance to the kinetic heap
- Kinetic Heater - elements consist of keys and priorities
  - The kinetic heater forms a binary search tree with respect to keys and a heap with respect to priorities
  - Either the keys or priorities are randomized

Questions?

Section 4

Applications to Convex Hulls

Existing Work

- Kinetic implementations of convex hulls exist that perform reasonable well with regards to the four metrics in the 2D case
- 3D kinetic convex hulls are not solved
- Rather than examine kinetic 2D and 3D hulls, we will apply kinetic principles to obtain an algorithm to compute 3D convex hulls

3D Convex Hull

- Previously we saw that 2D convex hulls can be constructed in $O(n \log n)$ time
- What about the 3D convex hulls?
- As with many problems, let's see if we can use the 2D case to solve the 3D case
- Since they are symmetric, we focus on the lower half of the convex hull (the lower hull) and compute the upper half identically

Reduction to 2D Convex Hull

- For each point (x, y, z) in the input, we construct a new point $(x, z - ty)$, where $t \in \mathbb{R}$
- $P = \{(x, y, z) : x, y, z \in \mathbb{R}\}$ and $P(t)' = \{(x, z - ty) : (x, y, z) \in P\}$

Lemma

$$\text{A point } (x, y, z) \in P \text{ is in the 3D lower hull of } P$$
$$\Longleftrightarrow$$
$$\exists t \text{ such that } (x, z - ty) \in P(t)' \text{ it is in the 2D lower hull of } P'$$

Proof of Lemma

Proof

$\implies$ direction:

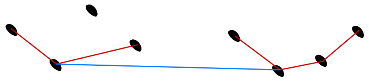
- Fix $(x_0, y_0, z_0) \in P$ and assume (x_0, y_0, z_0) is part of the lower hull of P. Then there must exist some plane $b_0 = -sx_0 - ty_0 + z_0$ that contains (x_0, y_0, z_0) and that all other points in P lie above
- Let $(x_1, y_1, z_1) \in P$, then $-sx_1 - ty_1 + z_1 = b_1 > b_0$
- Thus we have parallel planes $-sx - ty + z = b_0$ and $-sx - ty + z = b_1$. Rearranging, and setting $y' = z - ty$, we get the parallel lines $y' = sx + b_0$ and $y' = sx + b_1$, where the second line lies above the first
- Note $(x_0, z_0 - ty_0)$ lies on $y' = sx + b_0$, while our arbitrary pick of $(x_1, z_1 - ty_1)$ lies on $y' = sx + b_1$ which lies above the first line. Thus $(x_1, z_1 - ty_1)$ is part of the lower hull of $P(t)'$

The $\impliedby$ direction is very similar

Kinetic Approach

- We are interested in 2D convex hulls as t varies from $-\infty$ to ∞.
- Based off of our construction, each point $(x, z - ty)$ can join the convex hull at most once, and leave the convex hull at most once, so there will be at most $O(n)$ events

2D Convex Hull Algorithm



- First, we recursively construct lower hulls for the left and right halves of the points at $t = -\infty$
- Then in $O(n)$ time we join the two lower hulls with a line segment, which, if we remove points "inside" the bridge, creates a lower hull
- Certificates: whenever a points joins or leaves a convex hull (recursively propagate this up the lower hull), or whenever a bridge becomes concave
- Responsiveness - each certificate failure takes $O(\log n)$ time to fix
- We then advance time until no more events can occur

Convex Hull Algorithm Analysis

- To construct the initial lower hull, we solve the recurrence $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$ to get $O(n \log n)$ time
- Then we have at most $O(n)$ events that each take $O(\log n)$ time to fix, we spend an additional $O(n \log n)$ time as we increase t to ∞
- Altogether, the algorithm takes $O(n \log n)$ time, which is the same as in the 2D case

Questions?

There will be a brainteaser instead of a quote

— ALEX BROIHIER (2025)

Brainteaser

There are 2^k coins, some of which weigh 1kg, and the rest of which weigh 2kg. You are unable to tell by yourself what a coin weighs. You have a scale to compare weights with (you can put any number of coins on either side). How many comparisons do you need to make with the scale to determine which coins are 1kg and which coins are 2kg?

Bibliography I

Timothy Chan.

A minimalist's implementation of the 3-d divide-and-conquer convex hull algorithm, June 2003.

Wikipedia contributors.

Kinetic convex hull.

November 2022.

Accessed: 02-8-2025.

Wikipedia contributors.

Kinetic data structure.

May 2023.

Accessed: 02-8-2025.

Jeff Erickson and Tracy Grauman.

Cs 573: Topics in algorithms: Advanced data structures lecture 11, February 2006.