1   Overview

In the last lecture we started out by briefly discussing range trees (a topic we started two lectures ago). In particular, we showed how to support dynamic insertions and deletions in range trees efficiently. After this, we discussed the *vertical line stabbing problem* — given a collection of intervals, support queries of the form "How many intervals intersect the line $x = x_0$?". We showed that both interval trees and segment trees could solve this problem efficiently. Segment trees are a bit more memory-efficient, but interval trees are more flexible for other applications. We used these data structures to solve the *windowing problem* — given a collection of line segments, report how many lie in a given axis-aligned rectangle. In solving the windowing problem, we had to introduce an additional data structure called a *priority search tree*.

In this lecture we study kinetic data structures. These are data structures that contain information about objects in motion. They support the following three types of queries: (i) modify the motion path of an object; (ii) move forward to a specified point in time; (iii) return information about the state of the objects in the current time. We will go over kinetic sorting and kinetic heaps, and then state several results about kinetic data structures without proof.

After covering kinetic data structures, we will discuss the ray shooting problem — given a simple (possibly non-convex) polygon, support queries asking for the first point of intersection of a ray with the polygon. We will show how to support $\mathcal{O}(\log^2(n))$ queries and outline an approach for achieving a query time of $\mathcal{O}(\log(n))$.

2   Kinetic Data Structures

2.1   Introduction

A *kinetic data structure* is a data structure that stores moving data — that is, data that changes in some predictable way over time. Examples would be the locations of physical objects over time. We could also look at, for example, trajectories of objects in $\mathbb{R}^2$ and think of the x-coordinate as "time."

Kinetic data structures are useful, for example, in computer graphics and video games, to determine the next point in time at which two objects intersect.

Data. Each data point has a value that is a known function of time. For example, we could have affine data $(x(t) = a + bt)$. We could also have bounded-degree algebraic motion, or, more generally, pseudo-algebraic motion. *Pseudo-algebraic* means that all certificates of interest flip between true or false $\mathcal{O}(1)$ times as the objects move. It turns out that pseudo-algebraic will be the condition that we generally need in order for our kinetic data structures to run efficiently.

Operations. A kinetic data structure must support three types of operations:

- **modify**$(x, f(t))$ The previous position function describing point x is replaced by $f(t)$. This allows us to refit the structure to more up-to-date information, which is essential if the function we use to approximate the position of a point is imperfect.

- **advance**(t) The **advance** function advances the current time in the data structure to t. The current t must be less than the new t.

- **query** The final operation queries the data structure relative to the current time. The specific queries are dependent on the data structure in question, but possible queries would be find$_{\text{min}}$, find$_{\text{median}}$, etc.

In practice, we will rarely try to deal with the modify operation — the field of kinetic data structures as a whole is not very good at dealing with it rigorously.

Certificates. Before defining kinetic data structes we need to define **certificates**. A **certificate** is a testable logical statement pertaining to our data structure. A set of certificates can provide proof that a specific property of a data structure is true. For a simple example (borrowed from [Gui04]) consider the convex hull of four points:
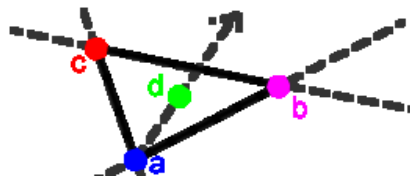
The convex hull of the points a, b, c, and d.

The certificates

- a is **left** of bc

- b is **right** of ad

- c is **left** of ad

- d is **left** of bc

provide a complete certification on the convex hull of $abcd$, as at least one certificate will change any time the convex hull of the points a, b, c, and d, changes its defining points.

Approach. The basic idea for a kinetic data structure is to store a static data structure and the conditions under which the data structure is valid, then fix the data structure as the conditions are violated. We outline this approach below:

1. store a data structure that is accurate now (i.e. at the current time)

2. augment the data structure with a set of *certificates* sufficient to prove the data structure valid.

3. compute the failure time for each certificate

4. store the failure times in a priority queue

5. as certificates fail, fix the data structure and replace the certificates as needed

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?

- **local** — what is the most number of certificates any object participates in?

- **compact** — what is the total number of certificates?

- **efficient** 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.)

2.2 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 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(x[i] > x[i+1]):
   swap(x[i],x[i+1],BST)
   add-certificate(x[i+1] <= x[i])
   replace-certificate(x[i-1] <= x[i],  x[i-1] <= x[i+1])
   replace-certificate(x[i+1] <= x[i+2],  x[i] <= x[i+2])
```

We now analyze our data structure relative to the metrics defined above.

Responsiveness. Because we use a balanced BST, and a certificate failure only affects $\mathcal{O}(1)$ elements, we can fix the data structure in $\mathcal{O}(\log(n))$ time.

Local. Every object only participates in $\mathcal{O}(1)$ certificates (in fact, always at most 2).

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

Efficient. Assuming pseudo-algebraic motion, every pair of objects only changes order $\mathcal{O}(1)$ times, meaning that the total number of data-structure events is $\mathcal{O}(n^2)$ (since there are $\mathcal{O}(n^2)$ pairs of objects). If we have $\frac{n}{2}$ points moving linearly with velocity 1, and $\frac{n}{2}$ points moving linearly with velocity -1, then points will change order $\Omega(n^2)$ times, so the number of "necessary" events is $\Omega(n^2)$ (since we must respond every time two objects change order). Therefore, the efficient is $\mathcal{O}(1)$.

Note that it is actually possible that there is a more efficient data structure to support kinetic predecessor. In particular, we have not actually proved that it is necessary to respond to all events.

2.3 Kinetic Heap

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 $\mathcal{O}(\log(n))$ responsiveness[1], $\mathcal{O}(1)$ locality, and $\mathcal{O}(n)$ compactness. The only non-obvious metric is efficiency. We show below that the efficiency is in fact $\mathcal{O}(\log(n))$ (by showing that the total number of events is $\mathcal{O}(n \log(n))$).

Analyzing the number of events in kinetic heap. We will show that there are at most $\mathcal{O}(n \log(n))$ 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 decendents of x that overtake x at some time after t.

Define $\Phi(t, x, y)$ as the number of decendants 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 $\mathcal{O}(n \log(n))$ 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 $\mathcal{O}(n \log(n))$. We will show that $\Phi(t)$ decreases by at least 1 each time a certificate fails, meaning that certificates can fail at most $\mathcal{O}(n \log(n))$ 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:

---

[1]The $\log(n)$ comes in because we need to use a priority queue to compute failure times.

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

and

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

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

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

From these bounds it follows that:

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

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

2.4   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: $\mathcal{O}(n^{2+\epsilon})$?

Approximatie 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].

3   Ray Shooting

We will now consider the *ray shooting problem*. This section closely follows [CEG$^+$94], and reuses several examples directly from the paper.
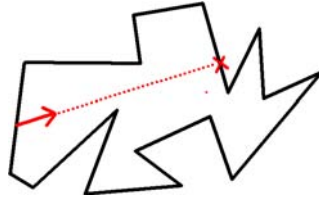
The ray shooting problem

The ray shooting problem asks us to efficiently determine when a ray first intersects a polygon.

Input. A collection of n points in $\mathbb{R}^2$ — the vertices of a simple polygon $\mathcal{P}$ in counterclockwise order. A polygon is simple if it has no holes (in other words, its boundary is connected).

Queries. A point inside $\mathcal{P}$ and a direction. The data structure should output the point in $\mathbb{R}^2$ at which the ray emanating at the given point and going in the given direction first intersects the boundary of $\mathcal{P}$.

As a warm-up, we will consider the case of a convex polygon.

3.1   Convex Polygons

If $\mathcal{P}$ is convex, we can solve the ray shooting problem by *triangulating* — decomposing $\mathcal{P}$ into triangles. Start by choosing a triangle so that $\frac{1}{3}$ of the vertices of $\mathcal{P}$ lie between any two vertices of the triangle. If we "cut out" the triangle from $\mathcal{P}$, then we are left with 3 regions, each with $\frac{n}{3}$ vertices. If we draw another triangle in each of the 3 regions, we end up with $\frac{n}{6}$ nodes per region, and so on untill there are $n-2$ triangles forming a binary tree with a ternary root. A nice property of this tree is that any ray passes through at most $\mathcal{O}(\log n)$ triangles. A diagram is shown below, in Figure 3.1:
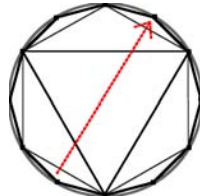
A ray cutting a regular polygon. The edges of the triangle form a tree decribing the vertecies.

To justify the claim that any ray passes through at most $\mathcal{P}(\log n)$ triangles, we take the planar dual of the triangulation, which yields, as noted above, a balanced tree. The triangles through which the ray passes must form a path in the tree, and since the tree has diameter $\mathcal{O}(\log n)$, the ray cannot pass through more than $\mathcal{O}(\log n)$ triangles.

If one were to imagine deforming any simple polygon into a convex polygon, a similar operation could be performed. By picturing each edge as an elastic band while deforming the shape back, we can picture how the tree structure and ray property would be maintained during the deformation, though many of the triangle would be deformed.
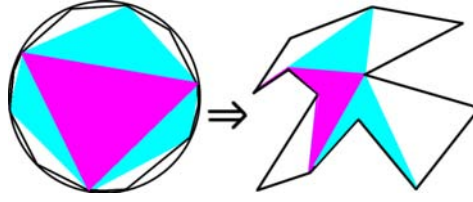
An example of a polygon broken into pseudo-triangles. Each edge of the pseudo-triangles is formed by taking the shortest path between two nodes.

More formally, suppose that we have a polygon $\mathcal{P}$ with vertices $v_1, \ldots, v_n$ in clockwise order. Let E_1 denote the shortest path within the polygon between v_1 and $v_{1+\frac{n}{3}}$. Let E_2 be the shortest path between $v_{1+\frac{n}{3}}$ and $v_{1+\frac{2n}{3}}$. Let E_3 be the shortest path between $v_{1+\frac{2n}{3}}$ and v_1. Then taking E_1, E_2, and E_3 together gives us a simple polygon (a "pseudo-triangle"), and removing this pseudo-triangle from $\mathcal{P}$ splits $\mathcal{P}$ up into some number of polygons, each with at most $\frac{n}{3}$ vertices. Once we have initially split up the polygon, we recursively split up each of the smaller polygons.

We call these new deformed triangles **pseudo-triangles**. In the next section we describe how to design a data structure to efficiently compute the point at which a ray fired into a pseudo-triangle collides with a wall.
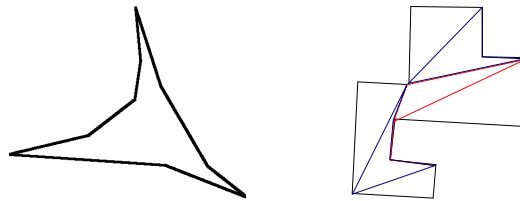
Left: A pseudo-triangle. Right:An example of a pseudo-triangulation of a specific polygon. The red polygon is the first pseudo-triangle created, and the blue polygons are the two that are created recursively.

3.1.1 Pseudotriangulations

As any three points in our polygon must either form a valid interial triangle or be deformed inwards, this algorythm will always return a valid partitioning of "pseudo-triangles". Furthurmore, because any concave edge of the polygon must be formed against a wall of the polygon, any ray leaving a pseudo-triangle will not reenter it before colliding with a boundary of the polygon. Since each division must reduce the size of each region by a factor of two (after the first partition is drawn), the process of triangulation terminates after $\mathcal{O}(\log n)$ steps.

By tracking the path of a ray from pseudo-triangle to pseudo-triangle until it exits the polygon we perform a traversal of the tree, and as the tree has depth $\mathcal{O}(\log n)$, we will only travel through $\mathcal{O}(\log n)$ pseudo-triangles in total. We only need to determine how to track the path of a ray through a given pseudo-triangle to determine runtime.

The behavior of a ray within a pseudo-triangle falls into two categories: either the ray homes into the wall it would have in triangle, or it runs into a wall that it would have missed had there been no concave edges. We can consider the cases of "homeing" and "fly-by" (see figure), to analyze these two options.

3.1.2 Ray Shooting in a psuedotriangle

We now can describe our data structure. For the edge of each pseudo-triangle we store a WBBST of the vertecies on that edge valued by their slope and weighted by their *bay size*, which we leave ambiguous as it appears on the problem set. We then create cascading links to similarly sloped line segments on other edges in our main data structure.
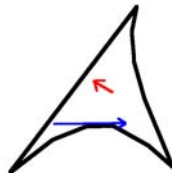
The red homes into the pseudo-triangle, the blue flys by the bottom before homing into the left.

homing   Consider first the homing case, in which the ray collides directly with a wall. Clearly a simple lookup in the BST of the edge the vetor is pointing at will find this edge.

fly-by   However, this ignores the case of the ray running into the adjacent wall (imagine the blue ray in the figure being slightly lower). By searching the BST for the ray parallel we can test if the ray falls passes under the cancave edge. If it does, this reduces back to the homing case.

runtime   Because lookup is $\mathcal{O}(\log n)$ per pseudotriangle and is run in $\mathcal{O}(\log n)$ this data structure supports ray shooting lookup in $\mathcal{O}(\log^2 n)$. In fact, this ignores potential speedups from analysis of the size of pseudo-triangles and introduction of fractional cascading, but these factors are left as an excersize and discussion in next lecture respectively.

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.

[CEG+94]   Bernard Chazelle, Herbert Edelsbrunner, Michelangelo Grigni, Leonidas J. Guibas, John Hershberger, Micha Sharir, and Jack Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12(1):54–68, 1994.

[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.

6.851 Advanced Data Structures

Spring 2010