1 Geometry

Field:

- We have been doing geometry—eg linear programming
- But in computational geometry, key difference in focus: **low dimension** d
- Lots of algorithms that are great for d small, but exponential in d

1.1 Range Trees for Orthogonal Range Queries

One key idea in CG: reducing dimension

- Do some work that reduces problem to smaller dimension
- Since few dimensions, work doesn't add up much.

What points are in this box?

- goal: $O(n)$ space
- query time $O(\log n)$ plus number of points
- (can't beat $\log n$ even for 1d)
- 1d solution: binary tree.
  - Find leftmost in range
  - Walk tree till rightmost

Generalize: Solve in each coordinate "separately"

- Idea 1: solve each coord, intersecting
  - Too expensive: maybe large solution in each coord, none in intersection
- Idea:
  - we know x query will be an interval,
  - so build a y-range structure on each distinct subrange of points by x
  - Use binary search to locate right x interval
  - Than solve 1d range search on y
  - Problem: n^2 distinct intervals
  - So n^3 space and time to build

Refine idea:

- Build binary search tree on x coords

- Each internal node represents an interval containing some points
- Our query's x interval can be broken into $O(\log n)$ tree intervals
- We want to reduce dimension: on each subinterval, range search y coords **only** amound nodes in that x interval
- Solution: each internal node has a y-coord search tree on points in its subtree
- Size: $O(n \log n)$, since each point in $O(\log n)$ internal nodes
- Query time: find $O(\log n)$ nodes, range search in each y-tree, so $O(\log^2 n)$ (plus output size)
- more generally, $O(\log^d n)$
- **fractional cascading** improves to $O(\log n)$

Dynamic maintenance:

- Want to insert/delete points
- Problem to maintain tree balance
- When insert x coord, may want to rebalance
- Rotations are obious choice, but have to rebuild auxiliary structures
- Linear cost to rotate a tree.
- Remember treaps?
  - We showed expect 1 rotation
  - Can show expected size of rotated tree is small
  - Then insert y coord in $O(\log n)$ auxiliary structures
  - So, $O(\log^2 n)$ update cost

2 Sweep Algorithms

Another key idea:

- dimension is low,
- so worth expending lots of energy to reduce dimension
- plane sweep is a general-purpose dimension reduction
- Run a plane/line across space
- Study only what happens on the frontier
- Need to keep track of "events" that occur as sweep line across
- simplest case, events occur when line hits a feature

2.1 Convex Hull by Sweep Line

- define
- good for: width, diameter, filtering
- assume no 3 points on straight line.
- output:
  - points and edges on hull
  - in counterclockwise order
  - can leave out edges by hacking implementation
- $\Omega(n \log n)$ lower bound via sorting

Build upper hull:

- Sort points by x coord
- Sweep line from left to right
- maintain upper hull "so far"
- as encounter next point, check if hull turns right or left to it
- if right, fine
- if left, hull is concave. Fix by deleting some previous points on hull.
- just work backwards till no left turn.
- Each point deleted only once, so $O(n)$
- but $O(n \log n)$ since must sort by x coord.

2.2 Halfspace intersection

Duality.

- $(a, b) \rightarrow ax + by + 1 = 0$.
- line through two points becomes point at intersection of 2 lines
- point at distance d antipodal line at distance $1/d$.
- intersection of halfspace become convex hull.

So, $O(n \log n)$ time.

2.3 Segment intersections

We saw this one using persistent data structures.

- Maintain balanced search tree of segments ordered by current height.
- Heap of upcoming "events" (line intersections/crossings)
- pull next event from heap, output, swap lines in balanced tree
- check swapped lines against neighbors for new intersection events
- lemma: next event always occurs between neighbors, so is in heap
- **note:** next event is always in future (never have to backtrack).
- so sweep approach valid
- and in fact, heap is monotone!

3 Voronoi Diagram

Goal: find nearest MIT server terminal to query point.
Definitions:

- point set p
- $V(p_i)$ is space closer to p_i than anything else
- for two points, $V(P)$ is bisecting line
- For 3 points, creates a new "voronoi" point
- And for many points, $V(p_i)$ is intersection of halfplanes, so a convex polyhedron
- And nonempty of course.
- but might be infinite
- Given VD, can find nearest neighbor via planar point location:
- $O(\log n)$ using persistent trees

Space complexity:

- VD is a **planar graph**: no two voronoi edges cross (if count voronoi points)
- add one point at infinity to make it a proper graph with ends
- Euler's formula: $n_v - n_e + n_f = 2$

- (n_v is voronoi points, not original ones)
- But $n_f = n$
- Also, every voronoi point has degree at least 3 while every edge has two endpoints.
- Thus, $2n_e \geq 3(n_v + 1)$
- rewrite $2(n + n_v - 2) \geq 3(n_v + 1)$
- So $n - 2 \geq (n_v + 3)/2$, ie $n_v \leq 2n - 7$
- Gives $n_e \leq 3n - 6$

Summary: $V(P)$ has linear space and $O(\log n)$ query time.

3.1 Construction

VD is dual of projection of lower CH of lifting of points to parabola in 3D.
And 3D CH can be done in $O(n \log n)$
Can build each vornoi cell in $O(n \log n)$, so $O(n^2 \log n)$.

3.2 Plane Sweep

Basic idea:

- Build portion of Vor behind sweep line.
- problem: not fully determined! may be about to hit a new site.
- What is determined? Stuff closer to a point than to line
- boundary is a parabola
- boundary of know space is pieces of parabolas: "beach line"
- as sweep line descends, parabolas descend too.
- We need to maintain beach line as "events" change it

Descent of one parabola:

- sweep line (horizontal) y coord is t
- Equation $(x - x_f)^2 + (y - y_f)^2 = (y - t)^2$.
- Fix x, find dy/dt
- $2(y - y_f)dy/dt = 2(y - t)(dy/dt - 1)$
- So $dy/dt = -(y - t)/(y - y_f)$

- Thus, the higher y_f (farther from sweep line) the slower parabola descends.

Site event:

- Sweep line hits site
- creates new degenerate parabola (vertical line)
- widens to normal parabola
- adds arc piece to beach line.

Claim: no other create events.

- case 1: one parabola passing through one other
  - At crossover, two parabolas are tangent.
  - then "inner" parabola has higher focus then outer
  - so descends slower
  - so outer one stays ahead, no crossover.
- case 2: new parabola descends through intersection point of two previous parabolas.
  - At crossover, all 3 parabolas intersect
  - thus, all 3 foci and sweep line on boundary of circle with intersection at center.
  - called **circle event**
  - "appearing" parabola has highest focus
  - so it is slower: won't cross over
  - In fact, this is how parabola's **disappear** from beach line
  - outer parabolas catch up with, cross inner parabola.

Summary:

- only **site events** add to beach line
- only **circle events** remove from beach line.
- n site events
- so only n circle events
- as insert/remove events, only need to check for events in newly adjacent parabolas
- so $O(n \log n)$ time

4 Randomized Incremental Constructions

BSP

- linearity of expectation. hat check problem
- Rendering an image
  - render a collection of polygons (lines)
  - painters algorithm: draw from back to front; let front overwrite
  - need to figure out order with respect to user
- define BSP.
  - BSP is a data structure that makes order determination easy
  - Build in preprocess step, then render fast.
  - Choose any hyperplane (root of tree), split lines onto correct side of hyperplane, recurse
  - If user is on side 1 of hyperplane, then nothing on side 2 blocks side 1, so paint it first. Recurse.
  - time=BSP size
- sometimes must split to build BSP
- how limit splits?
- autopartitions
- random auto
- analysis
  - $index(u, v) = k$ if k lines block v from u
  - $u \dashv v$ if v cut by u auto
  - probability $1/(1 + index(u, v))$.
  - tree size is (by linearity of E)

$$n + \sum 1/index(u, v) \quad \leq \quad \sum_u 2H_n$$

- result: **exists** size $O(n \log n)$ auto
- gives randomized construction
- equally important, gives **probabilistic existence proof** of a small BSP
- so might hope to find deterministically.

Backwards Analysis—Convex Hulls

Define.
algorithm (RIC):

- random order p_i
- insert one at a time (to get S_i)
- update $conv(S_{i-1}) \rightarrow conv(S_i)$
  - new point stretches convex hull
  - remove new non-hull points
  - revise hull structure
- Data structure:
  - point p_0 inside hull (how find?)
  - for each p, edge of $conv(S_i)$ hit by $\vec{p_0 p}$
  - say p *cuts* this edge
- To update p_i in $conv(S_{i-1})$:
  - if p_i inside, discard
  - delete new non hull vertices and edges
  - 2 vertices v_1, v_2 of $conv(S_{i-1})$ become p_i-neighbors
  - other vertices unchanged.
- To implement:
  - detect changes by moving out from edge cut by $\vec{p_0 p}$.
  - for each hull edge deleted, must update cut-pointers to $\vec{p_i v_1}$ or $\vec{p_i v_2}$

Runtime analysis

- deletion cost of edges:
  - charge to creation cost
  - 2 edges created per step
  - total work $O(n)$
- pointer update cost
  - proportional to number of pointers crossing a deleted cut edge
  - BACKWARDS analysis
    - run backwards
    - delete random point of S_i (**not** $conv(S_i)$) to get S_{i-1}

        * same number of pointers updated
        * expected number $O(n/i)$
            · what Pr[update p]?
            · Pr[delete cut edge of p]
            · Pr[delete endpoint edge of p]
            · $2/i$
        * deduce $O(n \log n)$ runtime

- 3d convex hull using same idea, time $O(n \log n)$,

4.1 Linear Programming

- define
- assumptions:
  - nonempty, bounded polyhedron
  - minimizing x_1
  - unique minimum, at a vertex
  - exactly d constraints per vertex
- definitions:
  - hyperplanes H
  - **basis** $B(H)$
  - optimum $O(H)$
- Simplex
  - exhaustive polytope search:
  - walks on vertices
  - runs in $O(n^{d/2})$ time in theory
  - often great in practice
- polytime algorithms exist, but bit-dependent!
- OPEN: strongly polynomial LP
- goal today: polynomial algorithms for small d

Randomized incremental algorithm

$$T(n) \leq T(n-1,d) + \frac{d}{n}(O(dn) + T(n-1,d-1)) = O(d!n)$$

Trapezoidal decomposition:

Motivation:

- manipulate/analayze a collection of n segments
- assume no degeneracy: endpoints distinct
- (simulate touch by slight crossover)
- e.g. detect segment intersections
- e.g., point location data structure
- Basic idea:
  - Draw verticals at all points and intersects
  - Divides space into slabs
  - binary search on x coordinate for slab
  - binary search on y coordinate inside slab (feasible since lines non-crossing)
  - problem: $\Theta(n^2)$ space

Definition.

- draw altitudes from each endpoints and intersection till hit a segment.
- trapezoid graph is *planar* (no crossing edges)
- each trapezoid is a *face*
- show a face.
- one face may have many vertices (from altitudes that hit the *outside* of the face)
- but max vertex degree is 6 (assuming nondegeneracy)
- so total space $O(n + k)$ for k intersections.
- number of faces also $O(n+k)$ (at least one edge/face, at most 2 face/edge)
- (or use Euler's theorem: $n_v - n_e + n_f \geq 2$)
- standard clockwise pointer representation lets you walk around a face

Randomized incremental construction:

- to insert segment, start at left endpoint
- draw altitudes from left end (splits a trapezoid)
- traverse segment to right endpoint, adding altitudes whenever intersect

- traverse again, erasing (half of) altitudes cut by segment

Implementation

- clockwise ordering of neighbors allows traversal of a face in time proportional to number of vertices
- for each face, keep a (bidirectional) pointer to all not-yet-inserted left-endpoints in face
- to insert line, start at face containing left endpoint
- traverse face to see where leave it
- create intersection,
  - update face (new altitude splits in half)
  - update left-end pointers
- segment cuts some altititudes: destroy half
  - removing altitude merges faces
  - update left-end pointers
  - (note nonmonotonic growth of data structure)

Analysis:

- Overall, update left-end-pointers in faces neighboring new line
- time to insert s is
$$\sum_{f \in F(s)} (n(f) + \ell(f))$$
  where
  - $F(s)$ is faces s bounds after insertion
  - $n(f)$ is number of vertices on face f boundary
  - $\ell(f)$ is number of left-ends inside f.
- So if S_i is first i segments inserted, expected work of insertion i is
$$\frac{1}{i} \sum_{s \in S_i} \sum_{f \in F(s)} (n(f) + \ell(f))$$
- Note each f appears at most 4 times in sum since at most 4 lines define each trapezoid.
- so $O(\frac{1}{i} \sum_f (n(f) + \ell(f)))$.
- Bound endpoint contribution:

- note $\sum_f \ell(f) = n - i$
- so contributes n/i
- so total $O(n \log n)$ (tight to sorting lower bound)

- Bound intersection contribution
  - $\sum n(f)$ is just number of vertices in planar graph
  - So $O(k_i + i)$ if k_i intersections between segments so far
  - so cost is $E[k_i]$
  - intersection present if both segments in first i insertions
  - so expected cost is $O((i^2/n^2)k)$
  - so cost contribution $(i/n^2)k$
  - sum over i, get $O(k)$
  - **note:** adding to RIC, assumption that first i items are random.
- Total: $O(n \log n + k)$

Search structure

Starting idea:

- extend all vertical lines infinitely
- divides space into slabs
- binary search to find place in slab
- binary search in slab feasible since lines in slab have total order
- $O(\log n)$ search time

Goal: apply binary search in slabs, without n^2 space

- Idea: trapezoidal decom is "important" part of vertical lines
- problem: slab search no longer well defined
- but we show ok

The structure:

- A kind of search tree
- "x nodes" test against an altitude
- "y nodes" test against a segment
- leaves are trapezoids
- each node has two children

- **But** may have many parents

Inserting an edge contained in a trapezoid

- update trapezoids
- build a 4-node subtree to replace leaf

Inserting an edge that crosses trapezoids

- sequence of traps Δ_i
- Say Δ_0 has left endpoint, replace leaf with x-node for left endpoint and y-node for new segment
- Same for last Δ
- middle Δ:
  - each got a piece cut off
  - cut off piece got merged to adjacent trapezoid
  - Replace each leaf with a y node for new segment
  - two children point to appropriate traps
  - merged trap will have several parents—one from each premerge trap.

Search time analysis

- depth increases by one for new trapezoids
- RIC argument shows depth $O(\log n)$
  - Fix search point q, build data structure
  - Length of search path increased on insertion only if trapezoid containing q changes
  - Odds of top or bottom edge vanishing (backwards analysis) are $1/i$
  - Left side vanishes iff **unique** segment defines that side and it vanishes
  - So prob. $1/i$
  - Total $O(1/i)$ for i^{th} insert, so $O(\log n)$ overall.