Discuss projects. People want extensions, see me.

1 Polynomial LP algorithms (cont)

Last time, saw ellipsoid and interior point.

1.1 Path Following

Potential function:

- Define
$$P(\mu) = cx - \mu \sum \log x_i$$
- minimize over $Ax = b$
- When μ is tiny, barrier is negligible except right at edge of polytope
- so optimum is right near LP opt, just pushed away from boundary a bit.
- For each μ, some optimum $x(\mu)$
- $\lim_{\mu \to 0} P(\mu)$ is LP opt.
- $P(\mu)$ as μ varies defines a function: *central path*
- starts where $\mu = \infty$, *analytic center* farthest from all boundaries.

Path following algorithm:

- repeatedly optimizes $P(\mu)$ for smaller and smaller μ
- when μ small enough, round to (optimal) vertex
- need to start somewhere near central path—revise problem to make this easy.
- How optimize nonlinear $P(\mu)$? gradient descent (actually second order taylor)

Path following step:

- Suppose are at $P(\mu)$
- take $\overline{\mu} = (1 - \beta)\mu$
- Then $P(\overline{\mu})$ near $P(\mu)$
- so gradient descent from $x(\mu)$ should converge fast to $x(\overline{\mu})$.

Actual implementation:

- don't wait to converge to $x(\overline{\mu})$.
- instead, trace out $y(\mu)$ that "follows" path without being on it.
- suppose have $y(\mu)$ near $x(\mu)$
- want $y(\overline{\mu})$ near $x(\overline{\mu})$
- take a (second order) taylor expansion of $P(\overline{\mu})$ near $y(\mu)$
- since $y(\mu)$ near $x(\overline{\mu})$, Taylor "accurate" (need $\beta \approx 1/\sqrt{n}$)
- take a "Newton step" from $y(\mu)$ towards minimizing $P(\overline{\mu})$
- takes us closer to $x(\overline{\mu})$
- update $\overline{\mu}$ and repear
- like potential method, $O(\sqrt{n}L)$ iterations halve potential.
- in practice, 9 iterations halve potential!

1.2 Randomized LP

New idea: focus on low dimension.
Standard incremental: $O(n^d)$ (poly!)
Randomization is crucial in geometry (actually everywhere; take class next year).
Seidel Randomized incremental algorithm

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

Bring in other random sampling techniques: best bound

$$O(d^2 n + b^{\sqrt{d \log d}} \log n)$$

Best known bound on diameter (Kalai and Kleitman): $n^{2+\log d}$

2 Geometry

Field:

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

2.1 Convex Hull by RIC

- 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

algorithm (RIC):

- random order p_i
- insert one at a time (to get S_i)
- update $conv(S_{i-1}) \to conv(S_i)$
  - 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 $p_i \vec{v}_1$ or $p_i \vec{v}_2$

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 $O(n \log n)$,

2.2 Orthogonal Range Queries

What points are in this box?

- goal: $O(n)$ space
- query time $O(\log n)$ plus number of points
- 1d: binary tree

Solve in each coordinate "separately"

- solve each coord, intersect too expensive.

2.2.1 kd trees

kd-trees:

- Split vertical, then horizontal
- size $O(n)$
- build time $O(n \log n)$

Query time:

- traverse subtree, descending into every node (region) that intersects query.
- pay one for each contained point

- this also amortizes cost of visiting any region completely contained in the box
- so only need measure number of region intersecting but not contained in region
- these hit one of the 4 boundaries
- let's see how many regions hit one vertical boundary
- vertical boundary on only one side of vertical split line
- but (worst case) on both sides of horizontal one
- so $Q(n) = 2 + 2Q(n/4)\Theta(\sqrt{n})$

2.2.2 Range Trees

Basic 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 $O(\log n)$

3 Plane Sweep Algorithms

Another key idea:

- dimension is low,
- so worth expending lots of energy to reduce dimension
- we saw this idea in LP
- 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

3.1 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!