CIS677
Notes for lecture 10:
Geometric optimization,
spatial subdivision trees

Outline

- geometric optimization
  - The minimum-ball problem
  - algorithms:
    - incremental
  - linear programming
    - a pivoting algorithm
  - distance to polytopes
- spatial subdivision trees
  - quadtrees, k-d trees, BSP trees
  - uses and construction

Seidel's incremental algorithm

The minimum-ball algorithms given in Lecture 9 are simple and linear time, but they beg the question of computing the smallest ball enclosing a *small* set of sites, with fewer than d^2 points.

Next we discuss algorithms that need only a primitive for finding the smallest ball containing at most $d+2$ sites.

As we used in the analysis to get an expected bound on V, we know that, for a given $p \in S$, either:

$$p \in SB(S \setminus \{p\}), \text{ or } p \in \text{basis}(S).$$

This should have been proved, as follows: Suppose $p \notin \text{basis}(S)$. Then $\text{basis}(S) \subset S \setminus \{p\}$, so $SB(S) = SB(S \setminus \{p\})$, implying $p \in SB(S \setminus \{p\})$.

Seidel's algorithm takes a random $p \in S$, and computes $SB(S \setminus \{p\})$: that is, the computation works on a random subset of S, of size $n-1$. Here it's quite unlikely that $p \notin SB(S \setminus \{p\})$.

However, suppose the unlikely occurs and $p \in \text{basis}(S)$. How can we use that information?

The members of $\text{basis}(S)$ are on the bounding sphere of $SB(S)$, so we can compute the smallest ball that contains $S \setminus \{p\}$ and has p on its boundary. Since p is in $\text{basis}(S)$, the result is $SB(S)$.

That is, for $T \subset S$, define $SB(S,T)$ as the smallest ball B containing S, and with T on the boundary of B.

We can now apply this recursively; $\text{SeidelSB}(S,T)$ is:

```
if S = T,
   compute SB(S,T) by brute force, return it;
pick p ∈ S \ T at random;
compute B := SeidelSB(S \ {p}, T);
if p ∈ B then return B;
return SeidelSB(S, T ∪ {p}); /* p ∈ basis(S) */
```

The first call to the algorithm is $\text{SeidelSB}(S, \{\})$.

Analysis

To bound the running time for this algorithm, we first observe that the number of sites determining $SB(S,T)$ is at most $d+1-|T|$.

The probability that p fails to be in $SB(S\backslash\{p\},T)$ is therefore $(d+1-|T|)/n$, since $r+1=n$. Letting $f(|S|,|T|)$ denote the number of primitive computations, we have

$$f(n,m) = f(n-1,m) + (d+1-m)f(n-1,m+1)/n;$$

we see that $(d+1-m)!n$ bounds $f(n,m)$, since

$$(d+1-m)!(n-1) + (d+1-m)(d-m)!(n-1)/n \leq$$
$$(d+1-m)!(n-1)(1+1/n) \leq (d+1-m)!n.$$

The running time of SeidelSB(S, $\{\}$) is therefore $O(d!n)$.

This algorithm is worse, in its dependence on d, than the previous ones, but some seemingly small changes lead to very interesting results for other problems.

Linear programming

The problem is:

> *Given a set H of halfspaces and a vector c, find the point $x \in P(H)$ minimizing $c \cdot x$.*

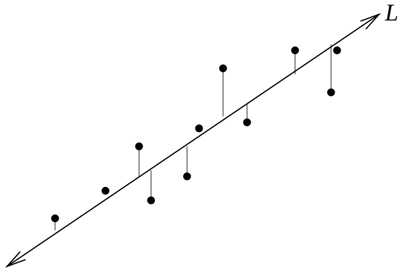
Since we can rotate the coordinate system, there's no loss in generality by assuming $c = [0 \ \ldots \ 0 \ 1]$, so that $c \cdot x = x_d$: we want to find the lowest point in $P(H)$.

In linear programming, the halfspaces H are called *constraints*, and a point is $P(H)$ is termed *feasible*.

This is only one formulation of LP; there are many others.

Example: plane fitting.

Suppose we are given a set S of points, and we want to find a nonvertical plane L that minimizes the maximum vertical distance to each point in S. That is, we want L minimizing



$$\max_{p \in S} \operatorname{dist}(p, L),$$

where $\operatorname{dist}(p, L)$ is the vertical distance between p and L.

Write L as

$$L = \{[x\ z] \mid z = b \cdot x\},$$

and each $p \in S$ as $[p_x\ p_z]$, where $p_x \in E^{d-1}$, and $\operatorname{dist}(p, L) = |p_z - b \cdot x|$, so we want a point $[x\ t]$, where x is in E^d, such that t is as small as possible, and

$$\begin{aligned} p_z - b \cdot p_x &\le t \\ b \cdot p_x - p_z &\le t, \end{aligned}$$

and so this plane fitting problem is an instance of linear programming.

Example: testing for vertices.

Given a set S and a point $p \in S$, p is a vertex of $\operatorname{conv}(S)$ iff p is not in $\operatorname{conv}(S \setminus \{p\})$, iff there is a plane separating p from $S \setminus \{p\}$. So to check if p is a vertex, we can check if the LP

$$\begin{array}{rl} \min & z \\ \text{subject to} & x \cdot q \le z \text{ for all } q \in S \setminus \{p\} \\ \text{and} & x \cdot p \ge z. \end{array}$$

That is, we find the hyperplane "closest" to $S \setminus \{p\}$.

The general LP problem is reducible to finding a feasible point (of a different LP problem).

Linear programming duality. The Farkas Lemma says that either there is a feasible point for this problem, or else there is one for a problem involving a vector c, of length $|S|-1$, with

$$\begin{array}{rcl} \sum_{q\in S\setminus\{p\}} c_q q & = & p \\ \sum_{q\in S\setminus\{p\}} c_q & = & 1 \\ c & \geq & 0 \end{array}$$

This conditions imply that the vector c gives p as a convex combination of the points of $S \setminus \{p\}$.

Here the vector c must satisfy the d equalities involving each coordinate of p, must sum to 1, and must have all coordinates at least 0.

This is a linear programming problem, the *dual* of the original one; this duality is *not* the same as, say, polar duality.

Minimum ball as quadratic programming

The minimum ball problem, which is to find x so that

$$\max_{p\in S} \|p-x\|$$

is minimized, can also be written as

$$\max_{p\in S}(p-x)\cdot(p-x),$$

which is equal to

$$\max_{p\in S} p^2 - 2p\cdot x + x^2,$$

and we can introduce a new variable r, where we want to minimize r^2, subject to

$$p^2 - 2p\cdot x + x^2 \leq r^2 \text{ for all } p \in S,$$

or putting $h = r^2 - x^2$,

$$\min h + x^2$$

subject to

$$p\cdot x + h/2 >= p^2/2 \text{ for all } p \in S.$$

That is, we have a collection of linear inequalities in $[x\ h]$, giving a polyhedron $P(H)$, and we want to minimize a *quadratic* function, $h + x^2$, within that polyhedron.

...so the minimum ball problem is not so far from linear programming.

(In fact, LP is reducible to the min-ball problem.)

The simplex algorithm

The simplex algorithm walks from vertex to vertex of $P(H)$, always reducing the d-coordinate.

```
get some v ∈ vert(P(H))
repeat
    pick w ∈ vert(P(H)) where vw is an edge and w_d < v_d
    v := w
until no such w exists
```

When the simplex algorithm stops, it has the global optimum point: if the neighbors of v are all above v, then so is $P(H)$.

A sketchy proof:
If v is the vertex at the stopping point, the condition that v is the global optimum is equivalent to the condition that the point $[0\ 1/v_d]$ is in the polar set of $P(H)$. (Here we assume that $o \in P(H)$, so $v_d \leq 0$. Since $[0\ 1/v_d]\cdot v = 1$, and $[0\ 1/v_d] \cdot w \leq 1$ for w a neighboring vertex of v, it follows that $[0\ 1/v_d]$ is a point in the bounding plane of v^*, and is in w^* for w a neighbor of v, and so is in v^o. Running time:

It takes $O(n + d^3)$ to move from vertex to vertex.

The simplex algorithm visits each vertex of $P(H)$ at most once, so runs in at most $O(d^3 n^{\lfloor d/2 \rfloor + 1})$ time; it is always much faster than this is practice, usually visiting $O(n + d)$ vertices.

There are examples where the worst-case running time is $2^{\Omega(d)}$; a variation of Seidel's algorithm gives a randomized version of a variation of simplex algorithm that takes subexponential time.

LP algorithms in small dimension

In small dimension, the same ideas as for minimum ball also work for linear programming: rather than sample sites, we sample halfspaces. The optimum vertex $\mathrm{opt}(R)$ for $P(R)$, where R is a subset of H, has similar properties as the minimum ball of a random subset of sites:

> $\mathrm{opt}(R)$ violates an expected small number of constraints (halfspaces);
> one of the constraint halfspaces violated by $\mathrm{opt}(R)$ is in $\mathrm{basis}(H)$.

In this context $\mathrm{basis}(H)$ is the set of d halfspaces that determine $\mathrm{opt}(H)$, so the same algorithms can be used.

Some complications for LP:

- $P(H)$ may be empty;
- $\mathrm{opt}(R)$ may not be unique;
- $\mathrm{opt}(R)$ may not exist because $P(H)$ is unbounded and there is no lowest point.

If $P(H)$ is empty, the problem is said to be *infeasible*.
If $P(H)$ is empty, it turns out that there must be $H' \subset H$, with $|H'| \leq d+1$, with $P(H')$ empty.
Since any point violates (is not in) one of the halfspaces in H', the algorithms will find H' for the same reasons they find the halfspaces determining the optimum vertex.

To enforce uniqueness of $\mathrm{opt}(R)$, we use as optimum the lexicographically minimum point of $P(H)$ that achieves the smallest d'th coordinate; that is, among all points of $P(H)$ with smallest x_d, we consider only those with smallest x_{d-1}, and so on.

To handle the unbounded case, we conceptually put a big box around $P(H)$: a set $H^@$ of $2d$ halfspaces $x_i > -\delta$ and $x_i < \delta$, for $i = 1 \ldots d$, and for δ some big value; we make δ act like infinity: for any given number we calculate using the real data, δ is bigger.

Seidel's algorithm, for LP

To return to Seidel's algorithm: given sets of halfspaces G and T, let

opt(G,T) be the optimum point in $P(G)$, subject to the condition that the point in on the bounding hyperplanes of all halfspaces in T.

basis(G,T) be the halfspaces whose bounding hyperplanes contain opt(G,T), so $T \subset \text{basis}(G,T)$.

$v(G,T)$ be the x_d value of opt(G,T).

So SeidelLP(H,T) is:

```
if H = T,
    return opt(H,T);
pick h from H \ T at random;
compute x := SeidelLP(H \ {h}, T);
if x ∈ P(H), return x;
return SeidelLP(H, T ∪ {h});
```

The above algorithm is wasteful, in a way: the work done in finding basis$(H \setminus \{h\}, T)$ in not used when finding basis$(H, T \cup h)$.

Consider using $T' := \text{basis}(H \setminus \{h\}, T)$ instead of T in the last line of Seidel's algorithm: use h together with the basis obtained when computing x.

The trouble with this idea is that the set $T' \cup h$ will have $d+1$ halfspaces: there may be no point in the intersection of the bounding planes.

Instead, use the basis of this set of $d+1$ elements, found by brute force: solve a linear programming problem with $d+1$ halfspaces, which means that one halfspace of the $d+1$ will not determine the optimum. So the changed algorithm returns $\text{opt}(H, \text{basis}(T' \cup \{h\}))$.

This algorithm, due to Sharir and Welzl, turns out to run in expected time *subexponential* in d.

The algorithm is actually a form of simplex, as is the recursive algorithm I described, so this result gives a randomized form of the simplex algorithm whose *provable* running time is faster than any known in the 40+ years that simplex has been known and used.

Distance between polytopes

The distance between two polytopes P and Q is the minimum distance between them, that is

$$\min\{d(p,q) \mid p \in P, q \in Q\}.$$

The special case $Q = \{0\}$ is the problem of finding the shortest vector in the polytope P.

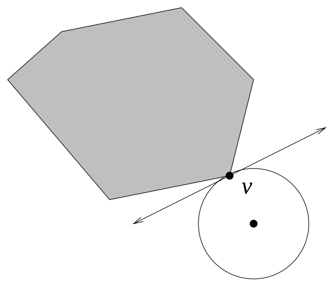
The shortest vector problem is

$$\min\{p^2 \mid p \in P\},$$

which is a quadratic programming problem, not so different from the minimum ball problem.

A shortest vector $v \in P$ has the properties that:

- the sphere $\{x \mid ||x|| = ||v||\}$ touches P at v;
- that sphere is tangent at v to the hyperplane $\{x \mid x \cdot v = v \cdot v\}$.
- the hyperplane bounds a supporting halfspace of P.



The general distance problem is reducible to the shortest vector problem: the Minkowski difference

$$P - Q = \{p - q \mid p \in P, q \in Q\}$$

is a polytope, and the shortest vector in it is some $p' - q'$ where p' and q' form a minimum distance pair. The hyperplane

$$\{x \mid \{x \mid x \cdot (p' - q') = (p' - q')^2\},$$

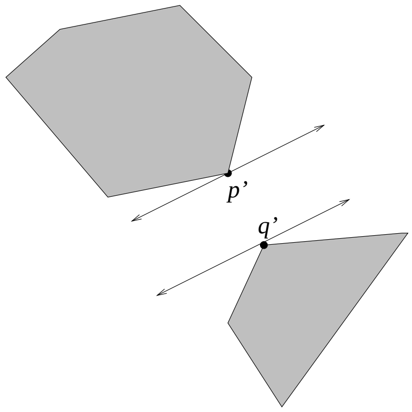
which is supporting for $P - Q$, corresponds to the condition that for every $p - q \in P - Q$,

$$(p - q) \cdot (p' - q') \geq (p' - q')^2,$$

which means in particular that for every $p \in P$,

$$(p - q') \cdot (p' - q') \geq (p' - q')^2,$$

and similarly for Q, so there is a pair of parallel hyperplanes that separate P and Q, contain p' and q', and are normal to $p' - q'$.



The approaches given so far also apply here: a small subset of the vertices of P and Q determine the minimum distance, and solving the problem for a random subset R of the vertices helps find that small subset. A point $x \in \mathrm{vert}(P) \cup \mathrm{vert}(Q)$ proves that the minimum has not been found if x is between the separating hyperplanes for the solution found for R.

Hence, this problem can be solved in $O(n)$ time.

Smallest enclosing ellipsoids

Here the optimum is determined by no more than $(d^2 + 3d)/2$ points, and otherwise the algorithm is exactly as for minimum ball.

Convex programming

These approaches actually apply to *convex programming*, where a convex function is to minimized subject to convex constraints. This characterizes a very large class of optimization problems.

Spatial subdivision trees

Trees such that:

> each node v corresponds to a region $\mathrm{Reg}(v)$, and
> $\mathrm{Reg}(\mathrm{child}(v)) \subset \mathrm{Reg}(v)$.

For each class of tree, child regions are found by splitting a node region in some way.

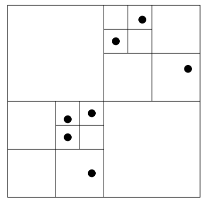
When applied to point data, leaves contain $O(1)$ input points.

Applied to region data, leaves meet $O(1)$ input objects, and input objects may meet more than one leaf: the objects are split in pieces, each piece contained in one leaf cell.

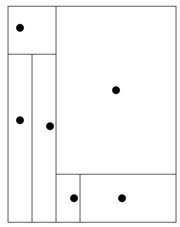
- Uses:
  - range queries
  - ray-shooting
  - set operations

- distance problems:
  - nearest neighbor searching
  - fixed-radius searching

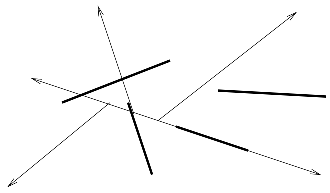
- quadtrees/octrees/hyperoctrees
  - Reg(v): a (hyper)cube;
  - children are 2^d equal-sized subcubes;
  - mesh generation
  - all-nearest-neighbors problem
  - n-body simulation
  - low-weight triangulation



- k-d trees
  - Reg(v) is a *box*, as in Lecture 2;
  - two children, from splitting by an /em axis-parallel hyperplane;
  - nearest-neighbor searching
  - MST's, TSP



- Binary Space Partition trees
  - used mostly for region data;
  - In 3d, Reg(v) is a convex polyhedron;
  - each node has two children, from splitting by a plane.
  - painters algorithm



These subdivision trees have the property that the leaves form a partition of Reg(root), and that two regions Reg(v) and Reg(v') have Reg(v) $\cap$ Reg(v') either empty, or Reg(v), or Reg(v').

Some tree schemes allow node regions to overlap. In the database literature, R-trees have boxes as Reg(v), and the children boxes of a node are chosen to minimize the total volume of the children. An R-tree node has many children: R-trees are a generalization of B-trees.

Some tree schemes split regions using something besides hyperplanes; for example, spheres.

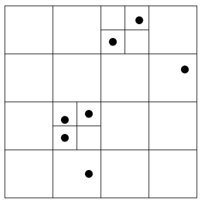
Point location

All the spatial subdivision trees allow the leaf containing a given point p to be found quickly: given that $p \in \mathrm{Reg}(v)$, each child of v is checked to see if $p \in \mathrm{Reg}(\mathrm{child}(v))$, and so on.

For k-d trees and BSP trees, this check needs only the splitting plane for the node.

For quadtrees, point location requires examining the binary expansion of the point coordinates.

For quadtrees in E^2, suppose $\mathrm{Reg}(\mathrm{root}) = [0,1]^2$; then for a node v at depth k, there are some values $\hat{x}$ and $\hat{y}$ with binary expansions $.x_1x_2\ldots x_k$ and $.y_1y_2\ldots y_k$, such that $\mathrm{Reg}(v)$ is the set of all points (x,y) whose binary expansions agree with $\hat{x}$ and $\hat{y}$ through the first k bits.



Range queries

For a given region B, suppose we want all points in $S \cap B$. Any spatial subdivision tree can be used to answer such queries.

The query procedure recursively traverses the tree; if $\mathrm{Reg}(v) \cap B$ is nonempty for a given node v, then the children of v are visited. If v is a leaf, the points (or objects) in v are checked against B.

Ray shooting

Any of these schemes can be used for ray-shooting:

In the plane:

> Given a set of line segments in the plane, build a data structure so that given a ray r with origin p, the first segment hit by r can be found easily.

In three dimensions, the input data might be a set of triangles.

The leaf cell containing p is found, and then the leaves of the tree are traversed, starting at p and walking along r.

Within each leaf, the intersection points of r with the objects in the leaf cell are found, if any, the one nearest p is returned.

There are two methods to traverse the leaves:

i. When r exits leaf v, find a point p' on r just outside of v. Use the tree to locate the leaf v' containing p', and continue with v.

ii. For each leaf, keep a list of adjacent leaves. At leaf v, check each leaf v' adjacent to v to see if r meets v'.

Method (ii) wins if typical leaves have a small number of adjacent leaves. For quadtrees, this can be enforced, by the following scheme: if two leaf cells v and v' differ in depth by more than 1, so they differ in side-length by more than a factor of 2, subdivide the larger one. Repeat until any two adjacent leaves differ in depth by at most 1.

Of course, this “smoothing” increases the number of nodes, so the speed depends on the data.

Smoothing is also useful in other contexts, such as mesh generation.

Building spatial subdivision trees

It is relatively easy to build quadtrees and k-d trees; they need $O(n)$ space for points. The space that is needed for more complex data depends on the scene.

For BSP trees, the choices are more open, and the space needed is also variable. However, because of the flexible choice of splitting plane, BSP's can be much smaller than quadtrees or k-d trees for a given data set.

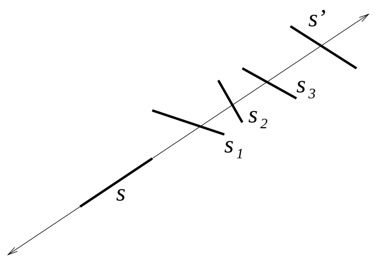
Theorem 1 *For any set S of line segments in the plane, there is a BSP tree for S that needs $O(n \log n)$ space.*

Proof. The BSP trees are built by *autopartition*, that is, the partitioning lines pass through segments of the input. The segments are considered one by one, and when segment s is visited, the line $\mathrm{aff}(s)$ is used to split all cells meeting s.

Each segment is split into a number of fragments during the algorithm; the cost of the tree is proportional to the number of fragments.

We will show that if the line segments are considered in random order, then the expected number of fragments of the resulting BSP is expected $O(n \log n)$.

A segment s' is cut by some $\mathrm{aff}(s)$ only if s and s' are in the same cell; this holds only if no segment s'' "shields" s' from s.
Here shielding means that visiting s'' before s, and splitting using $\mathrm{aff}(s'')$, puts s' and s into different cells.
Suppose there are k segments on $\mathrm{aff}(s)$ between s and s'; any of these can shield s' from $\mathrm{aff}(s)$.



So $\mathrm{aff}(s)$ cuts s' only if s is considered before s' and the k segments $s_1 \ldots s_k$; or, in the set

$$\{s, s', s_1 \ldots s_k\}$$

s must be picked first. The probability of this event is $1/(k+2)$, and so the expected number of cuts generated by aff(s) is no more than

$$\sum_{0 \le k < n_a} 1/(k+2) + \sum_{0 \le k < n_b} 1/(k+2),$$

where n_a and n_b are the numbers of segments meeting aff(s) on each side. Since $n_a + n + b \le n - 1$, the sum is no more than $2 \ln n$, and so the overall number of fragments is no more than $2n \ln n$. $\square$

In 3d, BSP trees exist that need $O(n^2)$ space, and some classes of BSP's need $\Omega(n^2)$ space.