Contents

1 Algorithms: Overview

TOPIC/type	Name	Best	Average	Worst	Space
Convex Hull		$\Omega(n)$		$O(n^4)$	
incremental	Naive Approach (Triples)	$\Omega(n^4)$	$\Theta(n^4)$	$O(n^4)$	$O(n)$
incremental	Slow $\mathcal{CH}$ (Pairs)	$\Omega(n^3)$	$\Theta(n^3)$	$O(n^3)$	$O(n)$
incremental	Jarvis March(Gift Wrapping)	$\Omega(nh)$	$\Theta(n \log n)$	$O(n^2)$	$O(n)$
incremental	Graham Scan	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
D&C	Quick Hull	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(n)$
prune&search	Marriage before Conquest (Ultimate)	$\Omega(n \log h)$	$\Theta(n \log h)$	$O(n \log h)$	
order decomp	Dynamic Convex Hull (updates)	$\Omega(\log^2 n)$	$\Theta(\log^2 n)$	$O(\log^2 n)$	
Intersecting Points		$\Omega(?)$		$O(n^2)$	
	Naive	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(n)$
	Plane Sweep (sweep line)	$\Omega((n+k)\log n)$	$\Theta(n^2 \log n)$	$O(n \log n)$	$O(n)$
	Hierarchical Search		$\Theta(n \log n)$	$O(\log n)$ (query)	$O(n)$
	LMSR (naive)	$\Omega(n^4)$	$\Theta(n^4)$	$O(n^4)$	$O(n)$
	LMSR (sweep)	$\Omega(n^2 \log n)$	$\Theta(n^2 \log n)$	$O(n^2 \log n)$	$O(n)$
	Building $\mathscr{A}$ (hammock)	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(n^2)$
	Topological Sweep	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(n)$
	—				
	Voronoi Diagram	$\Omega()$	$\Theta()$	$O()$	
	Delaunay Triangulation	$\Omega()$	$\Theta()$	$O()$	
	3D Convex Hulls	$\Omega()$	$\Theta()$	$O()$	
	Range Tree	$\Omega()$	$\Theta()$	$O()$	
	Interval Tree	$\Omega()$	$\Theta()$	$O()$	
	Treap	$\Omega()$	$\Theta()$	$O()$	
	Segment Tree	$\Omega()$	$\Theta()$	$O()$	

h is the number of points on $\mathcal{CH}$.

Space complexity is Auxilliary Space plus input size

k is the number of intersections, $k \leq n^2$

2 CS163 - Topics Overview

Convex Hull

- Simple Polygons
- Non-Simple Polygons
- Monotone Polygons
- Star-Shape Polygons
- Augmented Tree
- BBST
- Left Turn/orientation
- Support Line
- Test for CCW order
- Test for convexity
- Concatenable Queue
- Order Decomposable
- Divide&Conquer
- Bridge Finding
- Merge Hulls
- Maxima
- Dynamic updates

Algorithm	time	space
Naive Hull	$O(n^4)$	$O(n)$
Slow Hull	$O(n^3)$	$O(n)$
Jarvis March	$\Omega(nh)/O(n^2)$	$O(n)$
Graham Scan	$O(n \log n)$	$O(n)$
Quick Hull	$\Omega(n \log n)/O(n^2)$	$O(n)$
MBC	$O(n \log h)$	$O(n)$
Dynamic Hull update	$O(\log^2 n)$	$O(n \log n)$

Intersecting Points

- Point inclusion
- Status DS
- Stopping point DS
- DCEL
- Arrangement, $\mathscr{A}(S)$
- Least Median Square Regression
- Regularization
- Monotonization
- Triangulating
- Hierarchical
- Planar Subdivisions
- Topological Sweep
- UHT and LHT
- Ready Stack
- $v = n$
- $face \leq 2n - 4$
- $edges \leq 3n - 6$
- Total degrees $\leq 6n - 12$
- Line Sweep
-

Algorithm	time	space
Line segment intersection (naive)	$O(n^2)$	$O(n)$
Line Sweep	$\Omega((n+k)\log n)/O(n^2 \log n)$	$O(n)$
LMSR (naive)	$O(n^4)$	$O(n)$
LMSR (vertical sweep)	$O(n^2 \log n)$	$O(n)$
Build $\mathscr{A}(H)$: hammock	$O(n^2)$	$O(n^2)$
Topological Sweep	$O(n^2)$	$O(n)$

Voronoi Diagram & Delauney Triangulation

Algorithm	time	space
Fortune's Algorithm (VD)	$O(n \log n)$	$O(n)$
Naive (VD)	$O(n^2 \log n)$	
D&C (VD)	$O(n \log n)$	
DT	$O(n \log n)$	$O(n)$
3D CH	$O(n \log n)$	

3D Convex Hull

Algorithm	time	space
D&C	$O(n \log n)$	
Merge (sleeve)	$O(n)$	
Beneath & Beyond (incremental)	$O(n \log n + n^{\lfloor \frac{d+1}{2} \rfloor})$	
Gift Wrapping	$O(n^{\lfloor \frac{d+2}{2} \rfloor})$	
Shelling	$O(n^2 + \mathcal{P} \log n)$	

Linear Programming

Geometric DS & Rectilinear Contours

Algorithm	time/query	space	preprocess
1D Range Tree	$Q(\log n + A)$	$S(n)$	$P(n \log n)$
2D Range Tree (naive)	$Q(\log n + \lvert S_1 \rvert + \lvert S_2 \rvert)$	$S(n)$	$P(n \log n)$
2D Range Tree (better)	$Q(\log^2 n + A)$	$S(n \log n)$	$P(n \log n)$
d-D Range Tree	$Q(\log^d n + A)$	$S(n \log^{d-1} n)$	$P(n \log^{d-1} n)$
Interval Tree	$Q(\log n + A)$	$S(n)$	$P(n \log n)$
1D Segment Tree	$Q(\log n + A)$	$S(n \log n)$	$P(n \log n)$
2D Segment Tree	$Q(\log^2 n + A)$	$S(n \log^2 n)$	$P(n \log^2 n)$
Treap	$Q(\log n + A)$	$S(n)$	$P(n \log n)$
Lipski-Preparata	$O(n \log n + p \log \frac{n^2}{p})$	$O(n + p)$	
Wood	$O(n \log n + p)$	$O(n + p)$	
Guting	$O(n \log n + p)$	$O(n \log n + p)$	

3 Intro: Computational Geometry - A User's Guide (Notes)

Introduction to algorithims for computations that are geometric in nature. Souvaine & Dobkin describe some methods with geometric applications.

Three families of Geometric Algorithms:

- Decomposition of problem into subproblems – Hierarchical Searching Method
- Decomposition of problem into subproblems – Divide and Conquer Method
- Transform a problem into a new (maybe more tractable) format – Duality Method

Geometric Principles

- Planar Point Location
- Convex Hull Construction and Updating
- Computation of Polygon
- Computation of Disk
- Computation of Half-Space Intersections

3.1 Hierarchical Searching Method

- Geometric problem is preprocessed to a coarse representation, such that it can be broken down, and search queries can be called on localized region where the problem is solved.
- Algorithmic efficiency is then balanced against preprocessing time and storage space requirements.
- Binary Search of Sorted Array

3.1.1 Binary Search on Geometric Problems

Input: A collection of N disjointed polygons in the plane.

Output: For a given point P, find all polygons to which it belongs

Naive Solution: check for P in each points of the polygons.

Rectangle Search I & II & III

General & Dynamic Polygon Search

3.2 Divide and Conquer Method

- Problem broken into smaller subproblems, and solved recursively. A method is defined for combining solutions to subproblems to come up with solution for entire problem.

- Sort-merge
- Can work with heierarchical search methods, e.g. divide-and-conquer to sort a set, and then use binary search to find target.

3.3 Duality Method

- Duality is used as a transformation. Given two sets A and B and a problem about their interrelationship, apply transform T and solve the (ideally simpler) problem about T(A) and T(B).

4 Convex Hull

Covered in [1] CH. 1.1

HW 1

4.1 Geometry of Convex Hull

(a) (b)

Figure 1: Example of Convexity

- The computation of **planar convex hull** was one of the first computational geomtry problems.
- Convexity is important for issues involving point location and intersection detection, all of which have real world applications.
- *Convexity 1*: A subset S of the plane is **convex** if and only if for any pair of points $p, q \in S$, the line segment $\overline{pq}$ is completely contained in S. See Fig. 1a. The *convex hull*, $\mathcal{CH}(S)$, of a set S is the smallest convex set that contains S; it is the intersection of all convex sets that contain S.
- *Convexity 2*: How to compute the convex hull of a finite set P of n points in the plane? The area enclosed in the shaded region is the convex hull of P. See Fig. 1b. It is the unique convex polygon whose vertices are points from P.

(a) (b)

Figure 2: computing convex hull

- Fig. 2a: To compute a convex hull of a set of points, $P = \{p_1, p_2, ...p_9\}$, we compute a list of those vertices from P that are the vertices of $\mathcal{CH}(P)$, i.e. $\{p_4, p_5, p_8, p_2, p_9\}$, and list them in clockwise order. Defining $\mathcal{CH}(P)$ as a convex polygon is more useful than discussing the intersection of all convex sets.

- Fig. 2b: For points p and q that are endpoints of an edge and that are in P, we direct a line through p and q, and if $\mathcal{CH}(P)$ lies to one side, then all points in P must lie to that side of the $\overline{pq}$ line. And if all points of $P \setminus \{p, q\}$ lie to one side of $\overline{pq}$, then $\overline{pq}$ is an edge of the $\mathcal{CH}(P)$.

From Class, Algorithmic **Paradigms** covered:

- Sweepline/Incremental
- Divide & Conquer
- Prune & Search

4.2 Algorithm Naive Convex Hull - $O(n^4)$

From class:

For set of points in polygon P, for every triple i, j, k, for every point l not equal to i, j, k, if $l \in \Delta ijk$, discard l, as point inside the Δ can't be on the $\mathcal{CH}$. This runs $O(n^4)$ time $(n \times n \times n - 1 \times n - 2)$. $O(n)$ space.

4.3 Algorithm Ways to find Left/Right Turn

From class:

$\Theta(1)$ time: You can find the counter clockwise (left) or clockwise (right) turn of an angle given three points but calculating the determinant based on the order of the points that they were given in.

The determinant is twice the area of a triangle. Given points A, B, C, they form a ccw turn if the determinant is positive; o.w. if the area is negative, it is a right turn. This is verified using basic geometry or linear algebra.

$$
\begin{aligned}
D = \begin{vmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{vmatrix} &= x_a \times \begin{vmatrix} y_b & 1 \\ y_c & 1 \end{vmatrix} - y_a \times \begin{vmatrix} x_b & 1 \\ x_c & 1 \end{vmatrix} + \begin{vmatrix} x_b & y_b \\ x_c & y_c \end{vmatrix} \\
&= x_a(y_b - y_c) - y_a(x_b - x_c) + (x_b y_c - x_c y_b) \\
&= (x_a y_b - x_a y_c) + (x_c y_a - x_b y_a) + (x_b y_c - x_c y_b)
\end{aligned}
$$

A set of points is a $\mathcal{CH}$ if all the points turn in the same direction. If one different direction is detected, then the polygon is either complex or concave.

It must also be that, when you transform all vectors $\overrightarrow{v_i v_{i+1}}$ to the origin in that order, no vector crosses over any other, and it must only go around the origin once. Otherwise, it is not convex.

4.4 Algorithm SlowConvexHull(P) - Naive $O(n^3)$

Input: A set P of points in a plane.

Output: A list $\mathcal{L}$ cotnaining vertices of $\mathcal{CH}(P)$ in clockwise order.

1. $E \leftarrow \emptyset$
2. $\forall$ ordered pairs $(p, q) \in P \times P$, where $p \neq q$
3. **do** *valid* $\leftarrow$ **true**
4. $\forall r \in P, r \neq p, r \neq q$
5. **do if** r lies to the left of directed line from p to q
6. **then** *valid* $\leftarrow$ **false**
7. **if** *valid* **then** Add add directed edge $\overrightarrow{pq}$ to E.
8. From the set E of edges, construct a list $\mathcal{L}$ of vertices of $\mathcal{CH}(P)$, sorted in clockwise order.

- This is "piecewise linear" in finding the edges of a polygon.
- "Supporting Line" is such that all points of the polygon is on the line or to one side of the line (on the closed half plane).
- **Assume for now that methods to test if a point is to the right or left of a line is available. Assume this primative operation is O(1).*
- **initially ignores the degenerate case, where a point r may lie on $\overrightarrow{pq}$.* To consider the degeneracy, must specify that $\overrightarrow{pq}$ is an edge of $\mathcal{CH}(P)$ if and only if all other $r \in P$ lie strictly on the right or left of $\overrightarrow{pq}$, or they lie on the open line segment $\overline{pq}$.
- *Problems with rounding error could arise should coordinates are represented in floating point numbers, leading to unexpected results.*
- Constructing $\mathcal{L}$ takes about $O(n^2)$ time. For an edge $e_1 \in E$, take the source and destination points, add them to $\mathcal{L}$. Using the destination point of e_1, find the e_2 that has that as it's origin, and add e_2's destination point to $\mathcal{L}$. Repeat until only one edge is left in E.
- Complexity Analysis:
  - Check each of the $n^2 - n$ *pairs* of points. For each pair, look at $n - 2$ other points to see if they lie to one side. Total: $O(n^3)$.
  - Constructing $\mathcal{L}$ is $O(n^2)$.
  - Total overall: $O(n^3)$.
  - Space: $O(n)$

4.5 Algorithm Jarvis March (Gift Wrapping) - $O(n^2)$

https://iq.opengenus.org/gift-wrap-jarvis-march-algorithm-convex-hull/

- Sweepline/Incremental algorithm
- Starting with an extreme point (leftmost/rightmost/etc.), and keep wrapping the points in ccw direction.
- Finding the *next* point involves calculating if the the next candidate makes a ccw orientation with the previous two points. Decide on the directionality, but essentially:
  - Caclulate all the slopes from current point with all the other points.
  - Pick the point that gave use the minimum slope.
  - The min slope point is now the current point. Repeat this process until we return to the first point.
  - *You may need to pick a directionality when you get to the other extreme and start to "turn around".*
- This algorithm is output sensitive. So runtime is $O(nh)$, and worst is when $h = n$.

4.6 Algorithm ConvexHull(P) - Incremental Algorithm $O(n \log n)$

Needs only a sorting method and a method to test if three points can make a right turn. [1]

Briefly: Given the set P of points on a plane, sort the points $p_1, ..., p_n$ ordering them by their x-coordinate. Compute the convex hull vertices on the *upper hull* first, from left to right, from point p_1 to p_n. Then compute the convex hull vertices of the *lower hull* from right to left, from p_n to p_1.

Updating of the upper hull after adding point p_i is important. Suppose there is a list $\mathcal{L}_{up}$ containing the left to right upper hull vertices seen thus far, $\{p_1, ..., p_{i-1}\}$. Append p_i to $\mathcal{L}_{up}$. It is correct if p_i is the rightmost point so far, and if the last three points in $\mathcal{L}_{up}$ make a *right* turn. Move on to p_{i+1} if p_i can be in the upper hull thus far. If a left turn is made, delete the middle point from the upper hull, and keep rechecking the last three points until a right turn is verified.

Input: A set P of points in a plane.

Output: A list $\mathcal{L}$ cotnaining vertices of $\mathcal{CH}(P)$ in clockwise order.

1. Sort the points by x-coordinate, resulting in sequence $p_1, ..., p_n$.
2. Put p_1 and p_2 in $\mathcal{L}_{up}$, with p_1 as the first point.
3. **for** $i \leftarrow 3$ **to** n
4.   **do** Append p_i to $\mathcal{L}_{up}$
5.   **while** $|\mathcal{L}_{up}| > 2$ **and** the last three points in $\mathcal{L}_{up}$ don't make a right turn,

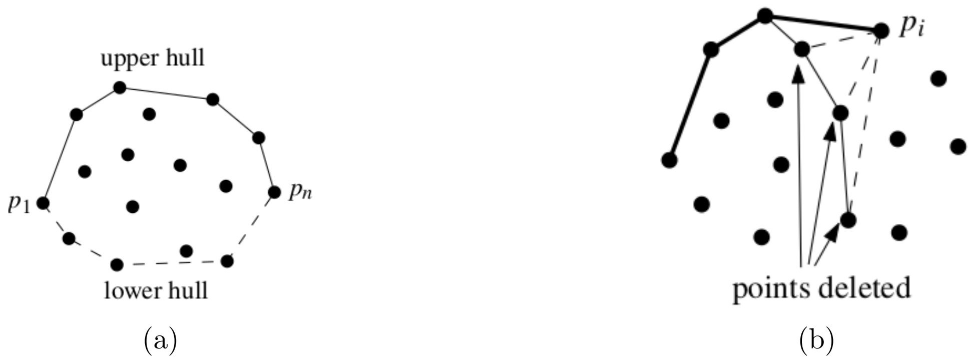


Figure 3: Convex Hull Algorithm. In (b), the order of those points being deleted is right to left, as p_i is the rightmost point thus far being checked for inclusion in the convex hull.

          do delete the middle of the last three points in $\mathcal{L}_{up}$

Put points p_n and p_{n-1} in $\mathcal{L}_{low}$, with p_n as the first point.

for $i \leftarrow n-2$ **down to** 1

   do Append p_i to $\mathcal{L}_{low}$

      while $|\mathcal{L}_{low}| > 2$ **and** the last three points in $\mathcal{L}_{low}$ don't make a right turn,

          do delete the middle of the last three points in $\mathcal{L}_{low}$

Remove the first and last points from $\mathcal{L}_{low}$ (avoid duplicate points of where upper and lower hull meet).

Append $\mathcal{L}_{low}$ to $\mathcal{L}_{up}$, call the result $\mathcal{L}$

return $\mathcal{L}$

- We assumed no two points have the same x-coordinate. To consider that, sort same-x-coord points by their y-coord.
- We will say for three collinear points (make a straight line), they make a left turn.
- Points very close together could create sharp left turns. For these, consider them the same point by rounding.
- Algorithm will compute a closed polygonal chain.
- **Theorem** *The convex hull of a set of n points in the plane can be computed in $O(n \log n)$ time.*
- **Proof**: See [1] page 8.
  - Correctness of computation of upper hull (and lower) is proof by induction. Briefly, the set $\mathcal{L}_{up}$ of $\{p_1, p_2\}$ is trivially the upper hull. $\mathcal{L}_{up}$ containing the chain $\{p_1, ..., p_{i-1}\}$ is known, by induction to only make right turns, and that all points fall below the chain. When considering point p_i, be know that p_1 is the

smallest point and p_i will be the biggest point thus far. There can be no points above the old chain, because if there were, then it would have to lie between p_{i-1} and p_i in sorted order.

– Sorting the points can be done in $O(n \log n)$ time. Computing upper hull is done in $O(n)$ time, because the for loop is executed a linear number of times, as any extra executions (from the while loop) is bound by n since extra points can only be deleted onces during the hull construction. Similarly, lower hull construction is $O(n)$ time. Therefore total time for computing convex hull is $O(n \log n)$.

– Space is $O(n)$.

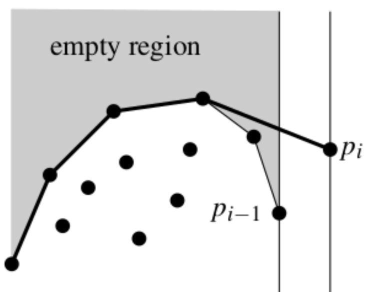


Figure 4: Convex Hull Algorithm - Correctness

4.7 Algorithm Graham Scan - $O(n \log n)$

https://iq.opengenus.org/graham-scan-convex-hull/

- See Section 4.6 for the $O(n \log n)$ time due to the pre-sorting. Forming a hull after sorting is $O(n)$ time. $O(n)$ space.
- Uses a stack to remove concavities. Incremental.
- The runtime rather comes from finding a central position (or even any), calculating all their polar angles (or slopes), and *sorting* them by the angles first. This algorithm comes with a **Pre-processing** step.
- Add the smallest y-coord point to the stack (finding it takes $O(n)$ time), add the next two points in the order to the stack (pre-processing took $O(n \log n)$). Calculate their orientation. The the next point is a candidate; calculate the orientation of the candidate and top two points in the stack. If you get the same orientation, add cancdidate to stack move on to next point; else, pop the stack, and check again. Repeat the popping until you get the orientation desired. Every element is pushed and popped from the stack at most once, and assuming each stack operation takes $O(1)$, so forming the hull is $O(n)$.

4.8 Bridges

- A *bridge* is the support line that forms the edge of $\mathcal{CH}$ or one that forms a new $\mathcal{CH}$ edge between two convex polygons when we want to form a new convex hull from their merge.

- Bridge finding is the goal of a divide and conquer method for finding convex hulls.
- Bridge finding takes logarithmic time. And there are 8 cases to examine. See Notes and Section 4.12.

4.9 General Divide & Conquer Convex Hull Methods

- Typically $T(n) = 2\,T(\frac{n}{2}) + \Theta(n)$. Outline for an unsorted set of points S:
  - *assume pre-sort; or sort ($O(n \log n)$)*
  - Find $\mathcal{CH}(S, 1, \frac{n}{2})$ ("left")
  - Find $\mathcal{CH}(S, \frac{n}{2}+1, n)$ ("right")
  - Merge the $\mathcal{CH}$'s
- Something like binary search ($O(\log n)$) can be used to find the bridge, for $T(n) = 2\,T(\frac{n}{2}) + \Theta(\log n)$.
- Some useful data structures, such as **concatenable queue**, can make inserting a point, deleting a point, or finding a point, splitting, joining (implant) take time $O(\log n)$.

4.10 Algorithm Quick Hull - $O(n^2)$

https://iq.opengenus.org/quick-hull-convex-hull/

https://www.geeksforgeeks.org/quickhull-algorithm-convex-hull/

- Like quicksort. Divide and Conquer method.
- Find the leftmost and rightmost points, ℓ and r.
- Find the highest and lowest points relative to the $\overline{\ell r}$ slope. We call this extreme point e
- Form a triangle with ℓ, r, and e. Discard all points inside triangle $\Delta \ell r e$.
- Repeat this process with one of ℓ or r, and the e you picked (i.e. one of the new edge of the triangle).
- Best case is $O(n \log n)$. Worse case is $O(n^2)$ if all points are in the convex hull.

4.11 Algorithm Ultimate Planar Convex Hull (Marriage Before Conquest) - $O(n \log h)$

https://iq.opengenus.org/kirkpatrick-seidel-algorithm-convex-hull/

[2]

- Prune & Search. Output sensitive.

- Runs in $O(n \log h)$ time, where $h = |\mathcal{CH}(S)|$. Even in worst case, where every point is in the hull, time is still $O(n \log n)$ like Graham's Scan.
- Instead of recurisng on each half after splitting the set like in a D&C, this computes how the sets should be merged, then recurse on subsets. This finds upper hull, and then when you find the lower hull, the two sets can be concatenated.
- Briefly: *Given a set, find the median vertical. Next goal is to find the bridge that crosses the median vertical. By definition of $\mathcal{CH}$, all points beneath this line is not in the $\mathcal{CH}$, so you discard/prune these points. The remaining points are split into the L and R subsets, and you recurse on them.*
- This is also useful when you have two disjointed convex hulls and want to merge them; simply find the bridge that crosses the median between the two hulls, as opposed to trying all the vertex combinations.
- The Kirkpatrick-Seidel algorithm:
  1. *don't sort!*
  2. Find the median vertical of the set of points ($T(n) = T(\frac{n}{5}) + T(\frac{2n}{4}) + \Theta(n) = \Theta(n) = \Theta(n)$).
  3. PAIRING: Randomly pair up the points, and calculate the misc. slopes. We have $\frac{n}{2}$ pairs.
  4. Find the median slope ($O(n)$), m_{median}.
  5. Calculate all the y-intercept of all the points if they have slope m_{median}. "Sweep slope of m_{median}", and find the most extreme max y-intercept, $b_{\max}$.
  6. BridgeFind: $T_{\text{bridge}}(n) \leq T_{\text{bridge}}(n - \lfloor \frac{n}{4} \rfloor) + O(n) = O(n)$

     This is when we start removing points that don't meet our criteria:

     (a) rotate the line $y_b = m_{\text{median}} x_b + b_{\max}$. Find the bridge with the first point you hit.

     (b) For the example where $b_{\max}$ is on the right side and has a negative slope for m_{median}, if it his a point that is paired with a point that forms a negative slope, discard the lower point, as you can only ever hit the upper point to be on the hull. Keep all other points.

     (c) More succinctly, for this pruning step, using the extreme point, $p_{\max}$, you found:

       - If $p_{\max}$ is on the *right* of line x_{median} (where $x_{p_{\max}} > x_{\text{median}}$), for every line with a slope less than m_{median}, discard its lower/right point.
       - If $p_{\max}$ is on the *left* of line x_{median} (where $x_{p_{\max}} < x_{\text{median}}$), for every line with a slope greater than m_{median}, discard its upper/left point.

     (d) This process discards $\frac{1}{4}$ of the points in the set, the pruning.

(e) Repeat from the PAIRING step until 2 points are left. These are the points of the Bridge.

$$T(n,h) \leq \begin{cases} 0 & \text{if } h = 1 \\ O(n) & \text{if } h = 2 \\ T(\frac{n}{2}, h_l) + T(\frac{n}{2}, h_r) + T_{\text{bridge}}(n) & \text{if } h \geq 3, \\ & \text{where } h_l + h_r = h, h_r, h_l \geq 1 \end{cases}$$

7. Repeat from step 2 for remainder subsets L and R on either side of the bridge.

Proof

Claim: $T_{\text{bridge}}(n) \leq T_{\text{bridge}}(n - \lfloor \frac{n}{4} \rfloor) + O(n)$

Show by induction on size of h that $T(n,h) \leq cn \log h$.

Basis: If $h = 2$ (minimum for top hull of the $\mathcal{CH}$), the top edge has at least 2 points. ($\log 2 = 1$).

$$T(n,2) = c_1 n \leq cn \log 2 \qquad \text{(pick } c \geq c_1\text{)}$$

Inductive Hypothesis: $T(n,h) \leq ch \log h$, for $h \leq k$.

Inductive Step:

$$\begin{aligned} T(n,h) &\leq c_1 n + T(\frac{n}{2}, h_l) + T(\frac{n}{2}, h_r) \\ &\leq c_1 n + \frac{cn}{2} \log h_l + \frac{cn}{2} \log h_r \\ &\leq c_1 n + \frac{cn}{2} (\log h_l + \log h_r) \\ &\leq c_1 n + \frac{cn}{2} \log(h_l h_r) \\ &\leq c_1 n + \frac{cn}{2} \log(h_l(h - h_l)) \\ &\leq c_1 n + \frac{cn}{2} \log(\frac{h}{2})^2 \\ &\leq c_1 n + cn \log(\frac{h}{2}) \\ &\leq c_1 n + cn \log h - cn \log 2 \\ &\leq cn \log h \qquad \text{if } c = c_1 \end{aligned}$$

4.12 Algorithm Dynamic $\mathcal{CH}$ / Order Decomposable Problem - $O(\log^2 n)$

https://www.geeksforgeeks.org/dynamic-convex-hull-adding-points-existing-convex-hull/

See Dynamic Convex Hull Notes.

See user guide section on this.

- Convex Hull construction is Order-Decomposable: we define some ordering function and merging function, where the merging operates iteratively on the ordered input set.
- This algorithm addresses when we want to to ADD a new point to or DELETE a point from the current set that we already computed the convex hull for. This avoids recalculating the entire set for changes regarding single points. Recalculating a $\mathcal{CH}$ is of cource $O(n \log n)$ time, but we want a single update to take only $O(\log^2 n)$ time, and make our algorithms dynamic for these updates.
- Using a concatenable queue makes bridge finding between two hulls is $O(\log n)$; then we can concatenate the left portion of the left hull with the right portion of the right hull. This relies on sorting the points though, so sorting dominates the runtime at $O(n \log n)$, while the merge gives a recurrance of $T(n) = 2T(\frac{n}{2}) + O(\log n) = O(n)$.
- Best to use a dynamic (augmented) tree as the data structure. The root contains the points of the $\mathcal{CH}$, while the subtrees/node has info about the partial hulls, and can point to another tree. We can retain the info of points that aren's in the $\mathcal{CH}$. The leaves are the actual data/points.
- There are 8 cases to examine to find the bridge between hulls A and B. See Class Nodes for Dynamic Convex Hull. But succinctly, when examining if line $\overline{ab}$ could be the bridge, we consider the angles given by a and its neighbors and by b and its neighbors ($\angle a_l\ a\ a_r$ and $\angle b_l\ b\ b_r$), the angles are relative to the line $\overline{ab}$:
  1. $\angle a_l\ a\ a_r$, $\angle b_l\ b\ b_r \leq 180°$ (both edges of each angle are above $\overline{ab}$) $\longrightarrow$ $\overline{ab}$ is the bridge.
  2. $\angle b\ a\ a_l \geq 180°$ ($\overline{a\ a_l}$ falls "below" $\overline{ab}$) $\longrightarrow$ points a and all to the right of a ("above" $\overline{ab}$) can't be the bridge point, discard. The bridge will be found in B and the left subchain of A from a.
  3. $\angle a\ b\ b_r \geq 180°$ ($\overline{b\ b_r}$ falls "below" $\overline{ab}$) $\longrightarrow$ points b and all to the left of b ("above" $\overline{ab}$) can't be the bridge point, discard. The bridge will be found in A and the right subchain of B from b.
  4. Both $\angle a_l\ a\ a_r, \angle a_r\ a\ a_l > 90°$ (both $\overline{a\ a_r}$ and $\overline{b\ b_l}$ fall "below" $\overline{ab}$, and their vectors extended intersect at point $v = (x, y)$). Let M_A be the max x-coord of A, and let m_B be the min x-coord of B. There is a region between M_A and m_B:
     (a) $x < M_A$ and $x < m_B$ (x is to the left of the region), then no points on the left subchain of A including a can be a bridge point. Remove those points.
     (b) $x > m_B$ and $x > M_A$ (x is to the right of the region), then no points on the right subchain of B including b can be a bridge point. Remove those points.
     (c) $M_A < x < m_B$ (x is in the region), then no points on the left subchain of A including a and no points on the right subchain of B including b can be a bridge point. Remove those points.
  5. Only $\overline{b\ b_l}$ falls below $\overline{ab}$ $\longrightarrow$ discard the right subchain in A and B.

6. Only $\overline{a\ a_r}$ falls below $\overline{ab} \longrightarrow$ discard the left subchain in A and B.

- This bridge finding process can run in $O(\log n)$ time. And since out DS is an augmented tree, the total runtime to apply the update is $O(\log^2 n)$.

Some Order-Decomposable Problems:

- Finding intersection of Half Plane
- Finding the set of Dominant Points (Maxima)
- Finding top and bottom of a polygon with tertiary search in sub-linear time

5 Point Inclusion

- It is $O(\log n)$ time to test point inclusion with a convex polygon.
- This problem can get more complex as the polygon gets more complex. For example, point inclusion with an x-monotone polygon, arbitrary polygon.
- For more complex polygons, you can regularize them into planar subdivisions, and test point inclusion on the convex polygons they breaks down into.
- There is a Slab Method for Planar Point Location (Dobkin and Lipton)
- Future sections address these more complex spaces.

5.1 3D Convex Hull

Introduced (background) in [1] CH. 11

6 Intersecting Points

Covered in [1] CH. 2-3, 6

HW 2

6.1 Intro to Intersecting Points

- Naively find intersecting points of a set S of n line segments is for each segment $s \in S$, find its intersections with all other segments $q \in S$. $O(n^2)$.
- We want *output sensitive* or *intersection sensitive* algorithms.
- **Plane sweep algorithms**:
  - Uses *status* DS to keep track of lines that currely intersect. It updates when we change the sweep line. The updates happen at *event* or *stopping points*, also kep in a DS. At a stopping point, algorithm updates the status of sweep line, and perform intersection tests. At a point on a line, the start of it when we add it to the status, calculate the intersections of the line against those lines already in the status DS. Currently, this algorithm is *not* output sensitive.
  - Improvements: Only test for intersections on the line with its neighbor lines (at most, test current line against two other lines). This also *orders* the segments in the status DS, so we are delivering the intersection points in order, as well. Stopping point DS now also includes intersection points.
- Special cases to think about later:
  - overlapping lines
  - more than two lines intersect at one point
  - line segment has a slope matching the sweeping lstinline
-

6.2 Algorithm Intersections: Line Segment Intersections / Plane Sweep - $O(n^2 \log n)$

- Let two lines $\overline{a_0a_1}$ and $\overline{b_0b_1}$ intersect at a point p.
  - Using the sweep line method, there is an event/stopping point *before* p that reveals $\overline{a_0a_1}$ and $\overline{b_0b_1}$ are adjacent and therefore get tested for intersection. We are specifically at a point of the sweep line such that i there are no other event points on the sweep line or between the it and the line given by the line's slope and point p. When A and B are not yet adjacent, this is when the sweep line is above all the segments, so there is nothing in the status DS. So ther must be an event point q when A and B become adjacent and are tested for intersection.

- Stopping points shall include endpoints AND the intersections points calculated along the way that are to the sweeping direction of the sweeping line (no looking backwards at points already detected).
- The status DS shall maintain the ordered sequence of segments that the sweeping line intersects (the stabbing points of sweep line ℓ).
- At a halting point, algorithm must update the status DS and find intersections (if any new).
  - Example (first Endpoint Event): Suppose s_i and s_k are adjacent. The next event is a new line's endpoint of s_j. Updating status reveals that s_j is adjacent to s_i and s_k. We see that s_j and s_i have an intersecting point, p. So we make p a new stopping point. Then move on to next event point.
  - Example (Intersection point event): This is when two lines that intersect cross, so their ordering in the status DS changes. Both shall get at most one new neighbor, for which new intersections must be calculated. Let these be adjacent in order: s_j s_k s_l s_m. Let the current event be the intersection of s_k s_l. s_k s_l must switch order, and the new intersections, if any, between new neighbors, s_j s_l and s_k s_m, are calculated and added to the stopping point DS. This can be done if lines were previously found to be adjacent before.
  - Example (second Endpoint Event): Suppose the status order of lines j k l, k is adjacent to its neighbors j and l. We reach event of the second endpoint of k (so k shall be leaving status DS). This means j and l will be neighbors and thier intersections gets calculated and put in stopping point DS.
- **Invariant**: All intersection points behind the sweep line have been computed correctly.
- **Data Structures**:
  - **Stopping Point/Event Point**: Using an event queue or a modified Balanced Binary Search Tree. Ordering will depend on how the ℓ sweeps. If ℓ is vertical and sweeps along the x-direction, then order by x-coord, and pick the point with the smaller y-coord if two events share the same x. Likewise, if ℓ is horizontal and sweeps down the y-direction, then order by y-coord, and pick the point with the smaller x-coord if two points have the same y. Fetching next event and inserting new even is $O(\log m)$ time where m is the number of events. This lets us test whether an event is already in the BBST. During creation (initialization is $O(n \log n)$ for a BBST), we fill the stopping point DS with just endpoints. For each starting endpoint, we store its segment with it. The algorithm runs every time we pop a new event odd and handle it, and runs until the event queue is empty. *See page 26 of [1] for full detail on handling all kinds of event points, including degenerates.*
  - **Status SD**: Using a BBST, which must be dynamic to deal with line segments going in an out as they intersect or stop intersecting ℓ. The actual ordering will be in the leaves of the BBST. Each update and neighbor operation takes $O(\log n)$

- Runtime is $O((n+k)\log n)$, where n is the number of segments in the input set, and k is the size of output (we can specify number of intersections, which doesn't double count when more than 2 segments intersects at a common point).
- Status DS is $O(n)$ space. Stopping point DS is $O(n+I)$ space, where I is number of intersections. It may actually be $O(n)$ if we remember to remove intersection points when segments are no longer adjacent, as the intersections will go back in again if they have not been report and their segments become adjacent again later.

6.3 Hierarichal Search (Triangles)

- $O(n)$ space
- Preprocessing is $O(n \log n)$
- Query time is $O(\log n)$
- Reference point is in a Δ, so the Δ can be divided into three triangles. Continue dividing it until we reach bottom.
- Example
  - $v = n = 5$
  - $f \leq 2n - 4 = 6$
  - $e \leq 3n - 6 = 9$
  - average degree < 6
  - Total degree $\leq 6n - 12$.
- Space: $S(n) = n + S(\frac{23n}{24}) = \Theta(n)$

6.4 Algorithm Line Sweep - $O((n + k) \log n)$

- A naive algorithm is to visit every intersection point in the place. $O(n^2)$ time for $O(n^2)$ intersections for n lines.
- If we use a line sweep, the choice in status and stopping point data structure can affect run time.
- Status array and Stopping priority queue gives us $O(n^2 \log n)$ time and $O(n)$ space.
- BBST status and stopping DS can give us
- Status array and Stopping priority queue gives us $O((n+k) \log n)$ time and $O(n)$ space, for n line segments with k intersections. The key is to have the status only contain the current stabbed lines in order based on who they are adjacent to, and the stopping point DS should only contain the intersections of current neighors. This involves a lot of updates, but updates in a BBST is only $O(\log n)$ time. Sweep line is output sensitive.

6.5 Regularizing Planar Subdivisions and Triangulating

- Turning an arbitrary polygon into monotone regions takes $O(n \log n)$ time.
- Turning a monotone polygon into triangles takes $O(n)$ time using a DCEL.
  - Preprocessing: $O(n)$ time
  - Triangulating: $O(n)$ time

- Testing a point for inclusion in a triangle is $O(1)$.

6.6 Doubly Connected Edge List (DCEL)

-

- DCEL contains a record for each face, edge, and vertex of a planar subdivision. (In application, it is useful for these records to also contain attribute information, such as what is in a given face.)

7 Polygon Triangulation

See CH. 3 of [1]

HW 2

7.1 Art Gallery Problem with 3-coloring Approach - $O(n \log n)$

- For triangulation of the polygon, it should deliver a DCEL, which allows for constant time stepping into another triangle/face of the triangulated polygon. This should be $O(n \log n)$ time.
- Depth-first search can be used to compute the 3-coloring of the vertices of all the triangles, which should take $O(n)$ time.
- There are at most $\lfloor n/3 \rfloor$ camera placements to find in a polygon of n vertices. The worst case being the comb-shaped polygon.

7.2 Triangulation: Partitioning a Polygon into Monotone Pieces

- $O(n^2)$ if we take $O(n)$ for each vertex to find a proper diagonal to make a triangle.
- $O(n)$ is we know polygon is convex, so we simply take the segments from one vertex to all other vertices, and we have our triangulation.
- Potential but hard strategy is to decompose the polygon into convex polygons and triangluate the sub-convex-polygons. However, we instead take an easier strategy: Decompose the polygon into monotone pieces, and triangulate those.
  - Partition into (y-)Montone pieces with diagonals: When walking from the topmost to bottomost vertices, we detect the direction we are going, namely dowards. A turn vertex is when the direction changes. So at that vertex, if the two incident edges of the turn vertex lie below it and the interior of the polygon above it, then the diagnonal for v must be above v. Likewise, if the two incident edges of v lie above v, then the diagonal should be below v. See page 50 in [1]

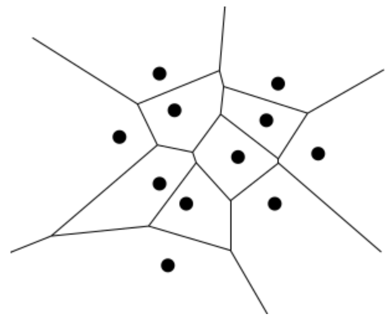
8 Least Median Square Regression

- To find a subset of a set that contains $\lceil \frac{n}{2} \rceil$ of the data points:
  - Naive: For every triple $i, j, k \in S$, make parallel lines through k and $\overline{ij}$, and check if $\lceil \frac{n}{2} \rceil$ are in that slab. $O(n^4)$ time.
  -
-

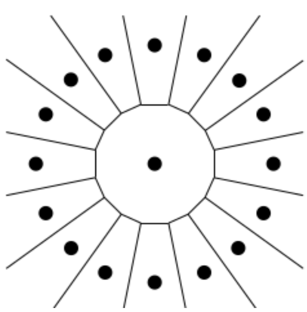
8.1 Duality

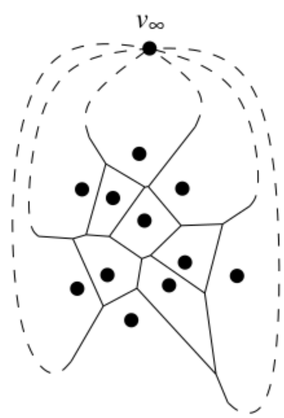
- These mappings hold from primal to dual and back:
  - $(a, b) \mapsto y = ax + b$
  - $y = cx + d \mapsto (-c, d)$
- Vertical Distance is preserved.
- For a set of n points in S, we can transform each point $p \in S$, where $p = (a, b)$, into the dual, so $T(p) = ax + b$. In the dual, we now have an arrangement of lines, which we can *sweep* over. Sinve vertical distance is preserved, we simply pick the "height" h we want, and at each stopping point in the sweep, we select the h adjacent neighbors (cuts) and record that distance. Typically, the goals is to find the "tightest" fit, or shortest distance that will contain $\lceil \frac{n}{2} \rceil$ of the data points. That gives us out LMSR.
- In the dual:
  - Points on the same segment in the primal will intersect in the dual at the pencil point ($y = cx + d \mapsto (-c, d)$).
  - The two regions created in the dual is known as dowble wedge.
  - If two primal segments intersect, then each of their pencils are in a wedge of the other ni the dual
  - if two primal segments don't intersect unless both their lines extend, then neither pencils is in the wedge of the other in the dual, but their wedges may overlap.
  - if two primal segments don't intersect unless one of the segments extends, then the pencil of the non-extended one lives in the wedge of the other in the dual.

9 Voronoi Diagrams

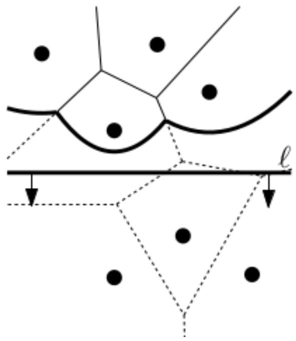


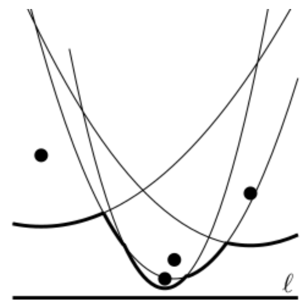
- The voronoi diagram of a set P of n points is denoted $\mathrm{Vor}(P)$. It may simply refer to the set of the edges and vertices that make up the diagram. The actual points in the set are known aas voronoi sites.
- Each cell in $\mathrm{Vor}(P)$ is denoted $\mathcal{V}(p_i)$, if the voronoi site is given by point $p_i \in P$.
- The edge between $p_i, p_j \in P$, $i \neq j$, is given by the perpendicular bisector of $\overline{p_i p_j}$.
- For n sites, each $\mathcal{V}(p_i)$ has at most $n-1$ vertices and edges; but actually, using *Euler's formula*, for $n \geq 3$, the entire $\mathrm{Vor}(P)$ has at most $2n-5$ vertices, and $3n-6$ edges. The complexity of $\mathrm{Vor}(P)$ is linear.



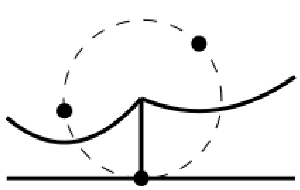


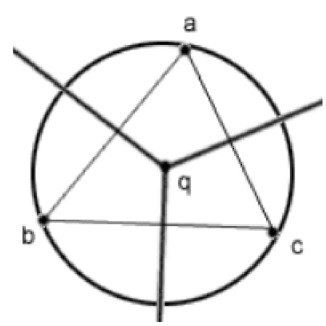
- $\mathrm{Vor}(P)$ can be computed in $\Omega(n \log n)/O(n \log n)$ time using Fortune's algorithm using $O(n)$ storage, using a plane sweep algorithm that uses a horozontal line and craetes a beach line / wavefront. We use the idea of parabolas, focus, and directrix.





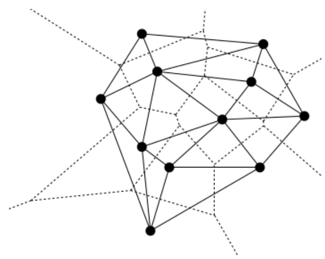
- Naive algorithm would be to find one voronoi region at a time: 1) Find the half plane for each point using the perpendicular bisector, 2) it is $O(n \log n)$ time for each half plane. Total time for all regions is $O(n^2 \log n)$.
- Given a Voronoi map, we can take the convex hull of the map. All points of the map that are on the hull are part of an infinite voronoi region (i.e. unbounded). So bounded regions are for points inside the hull.
  - D&C method of finding VD with CH:
  - Suppose the L and R VD are done and the points are presorted. We do the D&C CH algo ($O(n \log n)$) and the D&C VD algo at the same time.
  - The time to stitch the two VD together is $O(n)$.
  - $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$
- Empty Circle property: at a voronoi vertex, there should be no other voronoi site inside the circle given by the vertex (i.e. the circumcenter). All the Voronoi sites on the circumference of the circle are equidistance from this center. For general position, we say that no more than 3 points are circumcircular to this center.

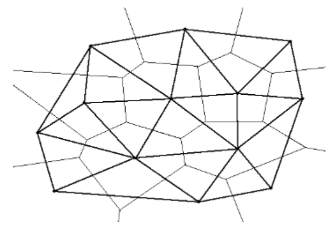




10 Delauney Triangulation

- The $\mathcal{DT}(P)$ is the dual of the $\mathcal{VD}(P)$. In a DT, two voronoi sites are connected by an arc iff their two regions are bounded by a common voronoi edge.
- Each triange of the DT is made from the 3 points of each VD's circmcenter.

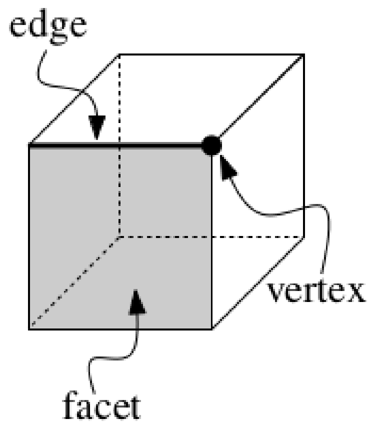


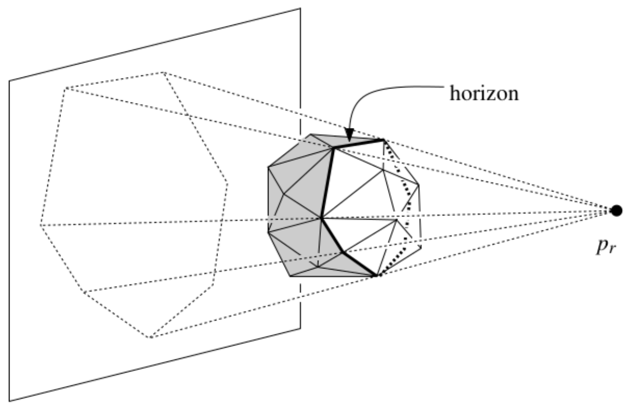


- the $\mathcal{DT}(P)$ is can be computed in $O(n \log n)$ time with $O(n)$ storage.
- Two edges of the DT intersect in their interiors.
- For all triangles of a DT, the circumcircle for the triangle must be empty in its interior. That is, no voronoi sites will lie inside the circumcircle for the triangle.
- If we translate the edges of the 3D convex hull into 2D, we are given the DT of P in 2D.

11 3D Convex Hull

- the complexity of a 3D CH of n points is $O(n)$.
- It can be computed in $O(n \log n)$ time.
- Preparata & Hong, D&C Algo. for 3D CH:
  - Given a set S if n points in $\mathbb{R}^3$, presort them with respects to x_1-axis, and P is the presorted set. Do $\mathsf{ConvexHull}(P, n)$:
  - If $n < 7$, compute the $\mathcal{CH}(P)$ by brute force.
  - Else:
    - Divide: $k = \lfloor \frac{n}{2} \rfloor$, $P_L = p_1, ..., p_k$, $P_R = p_{k+1}, ..., p_n$
    - Recurse: $\mathsf{ConvexHull}(P_L, k)$, $\mathsf{ConvexHull}(P_R, n-k)$
    - Merge: $\mathcal{CH}(P) = \mathsf{Merge}(\mathcal{CH}(P_L), \mathcal{CH}(P_R))$
  - The whole merge (i.e. finding the sleeve) takes $O(n)$ time. Recurrence is $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$
-



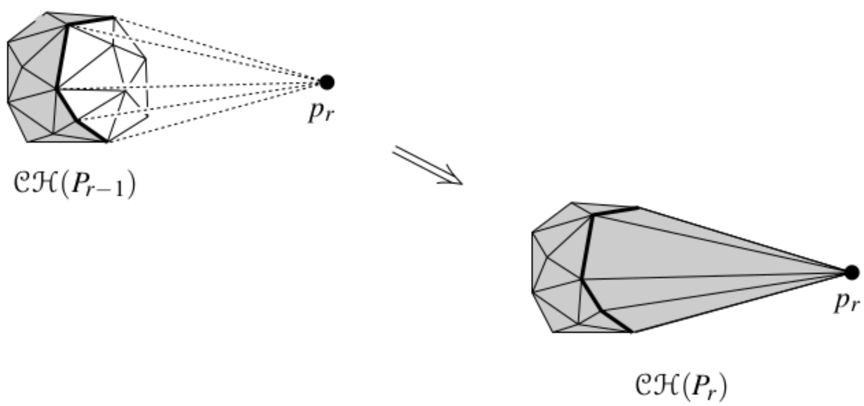


12 Linear Programming

13 Range Search

13.1 1D Range Search

- Problem: Given a set $S \subset \mathbb{R}$ of n points and a query interval $I = [\ell, h]$, find $S \cap I = \{x \in S \mid \ell \leq x \leq h\}$.
- DS: 1D Range Tree - Balanced Binary Search Tree that also has all the elements stored in the in the leaves, and the leaves are connected like a linked list.
- Algorithm: Search the tree for the leaf closes to ℓ and report the leaves to the right up to and including h.
- $P(n \log n)$
- $Q(\log n + A)$, A is size of solution set
- $S(n)$



13.2 2D Range Search

- Problem: Given a set $S \subset \mathbb{R}^2$ of n points and a query rectangle $R = [\ell_1, h_1] \times [\ell_2, h_2]$, find $S \cap R - \{(x, y) \in S \mid \ell_1 \leq x \leq \ell_2, h_1 \leq y \leq h_2\}$.
- Naive: Use a 1D range tree for the x-coord and a 1D range tree for the y-coord. Find points on the vertical range and horizontal range, and take the intersection of the two as the solution.
  - $P(n \log n)$
  - $Q(\log n + |S_1| + |S_2|)$
  - $S(n)$
- Wasteful: For all possible $\binom{n+1}{2}$ possible vertical strip, make a 1D range tree for the points in that strip; do the 1D query rectangle on the widest strip.
  - $P(n^3)$
  - $Q(\log n + A)$
  - $S(n^3)$
- Better: Let T be the BBST for S ordered by the x_1-coord. Each node $v \in T$, v has a range tree (the associated structure) for S_v ($S_v \subset S$, S_v is the canonical subset of node v) ordered by the x_2-coord. I.E. At node v, you will get points of a particular x_1-range. That node contains a 1D range tree of those points but ordered by their x_2-coord.
  - $P(n \log n)$
  - $Q(\log^2 n + A)$
  - $S(n \log n)$

13.3 d-Dimension Range Search

Similar to the 2D range search problems: Given a set $S \subset \mathbb{R}^d$ of n points and a query rectangle $R = \ell_1 \times \ell_2 \times ... \times \ell_d$, find $S \cap R$. We use a BBST on the x_1-coord. $\forall v \in T$, we make an associated secondary $(d-1)$ range tree on the $[x_2, ..., x_d]$ coord.

- $P(n \log^{d-1} n)$
- $Q(\log^d n + A)$
- $S(n \log^{d-1} n)$

14 Geometric Data Structures

14.1 Interval Tree

- Problem: Given a set S of n interval segments, and a query point q_x, determine the segments $i \in S$ that contain q_x, i.e. report all segments $i \in S$ that intersect the vertical $x = q_x$.
- Algorithm: Using the endpoints of the segments in S, so there are $2n$ endpoints. Find the median value m, and partition S into subsets S_L, S_C, and S_R. S_L and S_R contain segments exclusively to the left or right of the median. S_C contain those segments that are cut by the median. The node for m will contain: one sorted list for the left endpoints of S_C, one sorted list for the right endpoints of S_C, and its children (left and right subtrees) are recursively defined by S_L and S_R. To answer the query, use the median stored at each node as the key. At each node, search the list for segments that contain the target, using the information about the endpoints to know when you have finished traversing down the tree.
  - $P(n \log n)$
  - Median finding is $O(n)$
  - $Q(\log n + A)$
  - $S(n)$

14.2 1D Segment Tree

- Given a set S of n interval segments, and a query point $q_x \in \mathbb{R}$, determine the segments $s \in S$ that contain q_x, i.e. report all segments $\{s \in S \mid q_x \in s\}$ that intersect the vertical $x = q_x$.
- The n segments have $2n$ endpoints that break the number line into $2n + 1$ atomic intervals. Build a binary tree such that the leaves are atomic intervals. Label each node v by a subset T of S such that: s is in T if it contains the entire interval of v, and s does not contain the enitre of its sibling v'. A segment can be associated with at most two nodes per level.
  - $P(n \log n)$
  - $Q(\log n + A)$

- $S(n \log n)$, n segments appear in $O(\log n)$ nodes.

14.3 2D Segment Tree

- It is a 1D segment tree on the x intervals, and each node on the tree has a segment tree on the y intervals hanging off of it. For query (x, y), locate x on the primary tree, and y on the secondary tree. Report the rectangles encountered on the path on the secondary tree
  - $P(n \log^2 n)$
  - $Q(\log^2 n + A)$
  - $S(n \log^2 n)$

14.4 Priority Search Trees: Treaps

- Solves 2D range queries when the queries are unbounded rectangles. Ex: $[x_0, x_1] \times [y_0, \infty]$
- Find a point of y-max-coord $p \in S$ and remove it from the set. Let that be the root of the current tree.
- Find the median point m of x-coords. of the remaining points.
- Divide S into S_L and S_R.
- Create the vertex/node v for p and m. Let its children be treaps for S_L and S_R.
- Query:
- locate the two leaves that follow x_0 and precedes x_1. For each node on the path, check if the node's p is in the query range. Search all subtrees between the two paths. Report all points with y-coord at least y_0. By the heap property, we can stop searching a path when we hit a point with a y-coord less than y_0.
  - $P(n \log n)$
  - $Q(\log n + A)$
  - $S(n)$

15 Rectilinear Computaion Geometry

Algo.	Time	Space	Notes
Lipski-Preparata (PS)	$O(n \log n + p \log \frac{n^2}{p})$	$O(n + p)$	Determine vertical edges. Order left-right.
Wood (PS)	$O(n \log n + p)$	$O(n + p)$	Determine horizontal edges. Order left-right.
Guting (D&C)	$O(n \log n + p)$	$O(n \log n + p)$	Determine horizontal edges. Order left-right.

p is the output size.

References

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. Computational Geometry: Algorithms and Applications (3rd ed. ed.). Springer-Verlag TELOS, Santa Clara, CA, USA.

[2] David G Kirkpatrick and Raimund Seidel. 1986. The ultimate planar convex hull algorithm. SIAM J. Comput. 15, 1 (Feb. 1986), 287-299. https://doi.org/10.1137/0215021

[3] David G Kirkpatrick. 1983. Optimal Search in Planar Subdivisions. SIAM J. Comput. 12, 1 (Feb. 1983), 28-35.