Amy Bui
Notes

Contents

1 Algorithms: Overview **2**

2 Intro: Computational Geometry - A User's Guide (Notes) **3**
2.1 Hierarchical Searching Method . . . 3
2.2 Divide and Conquer Method . . . 3
2.3 Duality Method . . . 4

3 Convex Hull **5**
3.1 Geometry of Convex Hull . . . 5
3.2 **Algorithm** Naive Convex Hull - $O(n^4)$ . . . 6
3.3 **Algorithm** Ways to find Left/Right Turn . . . 6
3.4 **Algorithm** SlowConvexHull(P) - Naive $O(n^3)$ . . . 6
3.5 **Algorithm** ConvexHull(P) - Incremental Algorithm $O(n \log n)$ . . . 7
3.6 **Algorithm** Jarvis March (Gift Wrapping) - $O(n^2)$ . . . 9
3.7 **Algorithm** Graham Scan - $O(n \log n)$ . . . 10
3.8 Bridges . . . 10
3.9 Divide & Conquer Convex Hull Methods . . . 10
3.10 **Algorithm** Quick Hull - $O(n^2)$ . . . 11
3.11 **Algorithm** Ultimate Planar Convex Hull (Marriage Before Conquest) - $O(n \log h)$ . . . 11
3.12 **Algorithm** Dynamic $\mathcal{CH}$ / Order Decomposable Problem - $O(\log^2 n)$ . . . 13
3.13 3D Convex Hull . . . 14

4 Intersecting Points **15**
4.1 Intro to Intersecting Points . . . 15
4.2 **Algorithm** Intersections: Plane Sweep . . . 15
4.3 Doubly Connected Edge List . . . 18

5 Polygon Triangulation **19**
5.1 Art Gallery Problem with 3-coloring Approach - $O(n \log n)$ . . . 19
5.2 Triangulation: Partitioning a Polygon into Monotone Pieces . . . 20

1 Algorithms: Overview

TOPIC/type	**Name**	**Best**	**Average**	**Worst**	**Space**
Convex Hull		$\Omega(n)$		$O(n^4)$	
incremental	Naive Approach (Triples)	$\Omega(n^4)$	$\Theta(n^4)$	$O(n^4)$	
incremental	Slow $\mathcal{CH}$ (Pairs)	$\Omega(n^3)$	$\Theta(n^3)$	$O(n^3)$	
incremental	Jarvis March(Gift Wrapping)	$\Omega(nh)$	$\Theta(n \log n)$	$O(n^2)$	
incremental	Graham Scan	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	
D&C	Quick Hull	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	
prune&search	Marriage before Conquest (Ultimate)	$\Omega(n \log h)$	$\Theta(n \log h)$	$O(n \log h)$	
order decomp	Dynamic Convex Hull	$\Omega(\log^2 n)$	$\Theta(\log^2 n)$	$O(\log^2 n)$	
Intersecting Points		$\Omega(?)$		$O(n^2)$	
	Naive (test every segment for intersections with each other)	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	
	Plane Sweep (sweep line)	$\Omega(?)$	$\Theta(?)$	$O(?)$	
	Name	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	
	Kirkpatrick's Hierarchical Search		$\Theta(n \log n)$		$O(n)$
	Name	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	

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

2 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

2.1 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

2.1.1 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

2.2 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.

2.3 Duality Method

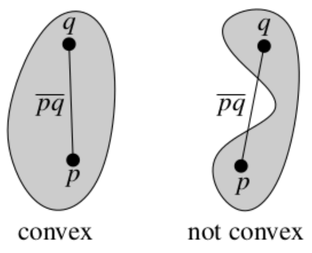
- Duality is used as a transformation. Given two sets A and B and a problem about their interrelationship, apply transform T and solve the (ideally simpler) problem about T(A) and T(B).

3 Convex Hull

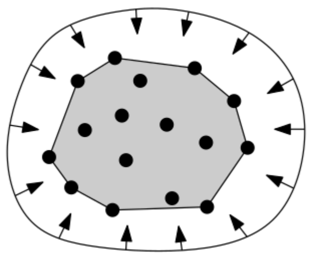
Covered in [1] CH. 1.1

HW 1

3.1 Geometry of Convex Hull



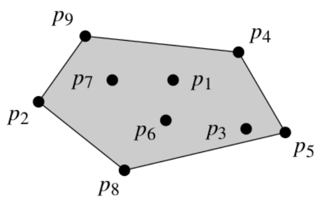
(a)



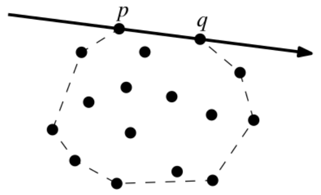
(b)

Figure 1: Example of Convexity

- The computation of **planar convex hull** was one of the first computational geomtry problems.
- *Convexity 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

3.2 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 $(n \times n \times n - 1 \times n - 2)$.

3.3 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.

3.4 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: $O(n^3)$.

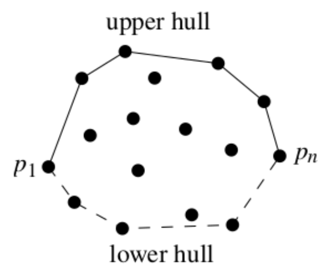
3.5 Algorithm ConvexHull(P) - Incremental Algorithm $O(n \log n)$

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

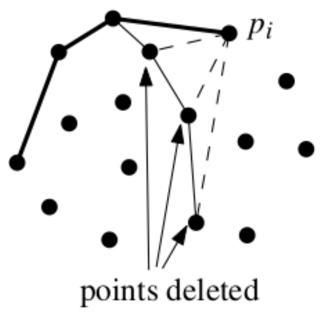
Briefly: Given the set P of points on a plane, sort the points $p_1, ..., p_n$ ordering them by their x-coordinate. Compute the convex hull vertices on the *upper hull* first, from left to right, from point p_1 to p_n. Then copute the convex hull vertices of the *lower hull* from right to left, from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



(a)



(b)

Figure 3: Convex Hull Algorithm

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,
6.                     do delete the middle of the last three points in $\mathcal{L}_{up}$
7. Put points $p_n$ and $p_{n-1}$ in $\mathcal{L}_{low}$, with $p_n$ as the first point.
8. for $i \leftarrow n-2$ down to 1
9.         do Append $p_i$ to $\mathcal{L}_{low}$
10.                while $|\mathcal{L}_{low}| > 2$ and the last three points in $\mathcal{L}_{low}$ don't make a right turn,
11.                    do delete the middle of the last three points in $\mathcal{L}_{low}$
12. Remove the first and last points from $\mathcal{L}_{low}$ (avoid duplicate points of where upper and
    lower hull meet).
13. Append $\mathcal{L}_{low}$ to $\mathcal{L}_{up}$, call the result $\mathcal{L}$
14. 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 $O(n \log n)$.

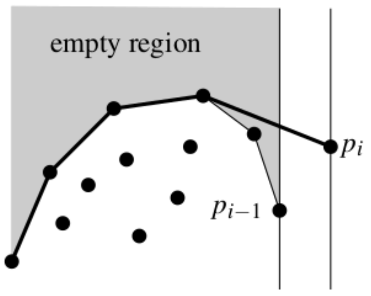


Figure 4: Convex Hull Algorithm - Correctness

3.6 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 $h = n$.

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

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

- Uses a stack to remove concavities. Incremental.
- Efficient and can run $O(n \log n)$
- 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 $O(n)$.

3.8 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.

3.9 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 $\mathcal{CH}(S, 1, \frac{n}{2})$
  - Find $\mathcal{CH}(S, \frac{n}{2} + 1, n)$

  - 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 $O(\log n)$.
-

3.10 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.
- Find the highest and lowest points relative to the left-right's slope.
- Form a triangle with the left, right, and lowest points. Discard all points inside triangle.
- Repeat this process with one of the left or right, and the extreme point you picked (i.e. one of the edge of the triangle).
- Best case is $O(n \log n)$. Worse case is $O(n^2)$.

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

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

- Prune & Search. Output sensitive.
- 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 fid 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.*
- 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 pm 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:

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

3.12 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.

- Order Decomposable Problem, Point Inclusion, Detecting Intersection
- This algorithm addresses when a new point is added to the current set we already computed the convex hull for, or delete a point from the set. This avoids recalculating the entire set for changes regarding single points.
- 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.

3.13 3D Convex Hull

Introduced (background) in [1] CH. 11

4 Intersecting Points

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

HW 2

4.1 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
-

4.2 Algorithm Intersections: Plane Sweep

- Let two lines $\overline{a_0 a_1}$ and $\overline{b_0 b_1}$ intersect at a point p.
  - Using the sweep line method, there is an event/stopping point *before* p that reveals $\overline{a_0 a_1}$ and $\overline{b_0 b_1}$ 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.

4.3 Doubly Connected Edge List

- 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.)

5 Polygon Triangulation

See CH. 3 of [1]

HW 2

5.1 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.

5.2 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]

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] Hi