Chan’s Algorithm

- Chan [’96] described an algorithm that’s as good as both Graham’s scan and Jarvis’s March. It’s actually a combination of both algorithms, but somehow runs in only $O(n \log h)$ time.
- Let’s build up this algorithm piece-by-piece to see how it works.
- First off, Graham’s scan suggests that sorting points is useful. Unfortunately, sorting $k$ points takes $\Theta(k \log k)$ time.
- However, if we break the point set into $n / k$ subsets of size $k$, we can sort them all individually in $O((n / k) k \log k) = O(n \log k)$ time. For any constant $c$ and $k \leq h^c$, the running time for all those sorts is $O(n \log h)$.
- In fact, we even have time to run Graham’s scan on these subsets.
- Of course, we don’t know $h$ in advance, so we do not know what size subsets we can afford to sort. Let’s suppose for now we have a guess $h^*$ on the true value for $h$.
- We’ll break up the input set $P$ into $k = \lceil n / h^* \rceil$ subsets of size at most $h^*$ each and run Graham’s scan on each subset. This will create a collection of $k$ mini-hulls that may overlap.

- Computing all $k$ mini hull takes $O(k(h^* \log h^*) = O(n \log h^*)$ time total.
- And now we make a couple observations.
- First, any point strictly interior to its subset’s mini-hull will lie strictly interior to $\text{conv}(P)$. In other words, each vertex of $\text{conv}(P)$ comes from a vertex of a mini-hull.
- Second, suppose we know a vertex $p$ on $\text{conv}(P)$. If the next vertex $q$ of $\text{conv}(P)$ in ccw order lies on a vertex of mini-hull $Q$, then $<\rightarrow pq$ is a support line for $Q$.
- Finally, there are only two support lines between $p$ and $Q$, and if we’re given an array $<q_1, \ldots, q_m>$ of $Q$’s boundary vertices in ccw order, we can find both support lines in $O(\log m)$ time. This utility function as Mount calls it is essentially a pair of binary searches.
Taken together, these facts give us a faster way to run Jarvis’s March! When it’s time to compute vertex $v_i$ of $\text{conv}(P)$, we only consider the $\leq 2k$ support line vertices from the mini hulls.

We spend $O(k \log h^*) + O(k) = O(k \log h^*)$ time finding each vertex of $\text{conv}(P)$. Finding all $h$ vertices of $\text{conv}(P)$ will take $O(h (k \log h^*)) = O(n (h / h^*) \log h^*)$ time.

**Conditional Algorithm**

Again, $h^*$ is just a guess for $h$. What happens if our guess is wrong?

- If we guess high so that $h^* > h$, then both phases will take $O(n \log h^*)$ time. That’s bad if we guess really really high, but if, say, $h < h^* \leq h^2$, then $O(n \log h^*) = O(n \log h)$. We can overshoot by a polynomial larger than $h$.
- If we guess too low, then it takes $O(n \log h^*) = O(n \log h)$ time to compute the mini hulls, but completing the march may take much more time than we would like. Fortunately, a low guess is easy to detect before it becomes a problem. Just cut the algorithm off as soon as we add an $h^*$th vertex to $\text{conv}(P)$. We spend at most $O(n \log h^*)$ time if we cut the algorithm off in this way.
- Both observations motivate this procedure Mount calls ConditionalHull. Given a guess $h^*$, it tries to compute $\text{conv}(P)$, but it gives up if the guess is too low.

As we saw, the procedure will take $O(n \log h^*)$ time total.

**Guessing the Hull’s Size**

---

**Chan’s Algorithm for the Conditional Hull Problem**

**ConditionalHull**($P, h^*$):

1. Let $k \leftarrow \lceil n / h^* \rceil$. Partition $P$ (arbitrarily) into disjoint subsets $P_1, \ldots, P_k$, each of size at most $h^*$.
2. For $j \leftarrow 1$ to $k$, compute $H_j = \text{conv}(P_j)$ using Graham’s scan, storing each in an ordered array.
3. Let $v_0 \leftarrow (-\infty, 0)$, and let $v_1$ be the bottommost point of $P$.
4. For $i \leftarrow 1, 2, \ldots, h^*$:
   - (a) For $j \leftarrow 1$ to $k$, using the utility lemma, compute the tangents points $q_j^-$ and $q_j^+$ for $H_j$ with respect to $v_{i-1}$.
   - (b) Set $v_i$ to be the tangent point that minimizes the turning angle with respect to $v_{i-2}$ and $v_{i-1}$.
   - (c) If $v_i = v_1$ then return the pair (success, $V = \{v_1, \ldots, v_{i-1}\}$)
5. If we get here, we know that $h^* < h$, and we return (failure, $\emptyset$)

---

As we saw, the procedure will take $O(n \log h^*)$ time total.
Now we just need to find a guess $h^*$ such that $h < h^* \leq h^2$ (or any other polynomial). But how?

We'll start with a low guess like $h^* = 4$. Then we'll slowly increase it until we get one that we would like.

Adding 1 to $h^*$ after each guess is too slow. Even doubling after each guess isn't great: a detailed analysis would show that strategy runs in $O(n \log^2 h)$ time.

But we can be off by a polynomial, so why don't we *square* the guess each time, trying 4, 16, 256, …, $2^{2^i}$. Even growing this quickly, we'll still have our desired $h < h^* \leq h^2$ the first time we guess high.

But will we spend too much time on the earlier guesses?

For the “ith” guess, we use $h^*_i := 2^{2^i}$. The time taken to run ConditionalHull is $O(n \log h^*_i) = O(n \log 2^{2^i}) = O(n 2^i)$.

We'll use $\lg$ to denote log base 2. Summing the times for each guess gives us $O(n) \cdot \sum_{i = 1}^{\lg \lg h} 2^i$ time total. That's $O(n)$ times a geometric series with a constant ratio. It's well known that these series are proportional to their largest term (try to recall the Master Method if you took 6363), so the total time is $O(n 2^{\lg \lg h}) = O(n \log h)$. Hurray!

Here's a summary of the whole process:

<table>
<thead>
<tr>
<th>Hull($P$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $h^* \leftarrow 2$; status $\leftarrow$ failure</td>
</tr>
</tbody>
</table>
| (2) while (status $\neq$ failure):
| (a) Let $h^* \leftarrow \min((h^*)^2, n)$ |
| (b) (status, $V$) $\leftarrow$ ConditionalHull($P, h^*$) |
| (3) return $V$ |

Can we do better? No [Kirkpatrick and Seidel ’86]. See Mount if you want the details.

**Line Segment Intersection**

- Time to finally move on from convex hull!
- Suppose we’re given a set $S$ of $n$ line segments in the plane. We want to report (output) every one of their intersections.
- Intersection of geometry objects comes up a lot in computational geometry; for example, if you’re working with robotics, you need to know whether two objects may intersect.
- This particular problem of line segment intersection is directly applicable to computing map overlays. Say I have a road network represented as some line segments and the boundaries of counties or states represented as another set of line segments. The intersection points tell me where maintenance responsibilities change along road segments. We may also just want to know where you’re entering and exiting boundaries on a map.
- Now, you might observe that a collection of $n$ line segments might have no intersections, or it might have all (n choose 2) = $\Theta(n^2)$ intersections.
• So if all we care about is worst-case performance, then we may as well just test every pair of line segments and output the intersections we find in optimal $O(n^2)$ time.

• But consider that application I just described. Not every road in the state crosses every county boundary. We shouldn’t expect anywhere near Omega($n^2$) intersections in practice, so we should try to find algorithms that are much faster when there are few intersections.

• We want to find a good output sensitive algorithm like we did for convex hulls.

• Like before, we’ll use some “general position” assumptions:
  • The endpoints of the line segments and the intersection points have distinct x-coordinates.
  • No segment endpoint lies on another segment.

As before, the 3Marks textbook goes into more detail than I will and avoids most of these assumptions.

**Plane Sweep Algorithm**

• Let $n = |S|$ and $m$ = the number of intersections.

• We’ll discuss an algorithm of Bentley and Ottmann [’79] that runs in $O((n + m) \log n)$ time.

• The main method behind this algorithm is called a plane sweep.

• Imagine sweeping vertical line $ell$ continuously across the plane from left to right (I’m making $ell$ vertical, because that’s my and Mount’s preference. The book uses a horizontal line going down).

• As the line moves from left to right, we’ll be intersecting a subset of the line segments. The main idea behind a plane sweep is to take note of each moment this set of segments changes, doing a bit of work at each change. The moments (x-coordinates) where something interesting happens along the sweep line are called *event points*.

• In any plane sweep algorithm, we need to keep track of three things:
  1. the *partial solution* that has already been constructed to the left of $ell$ (here, that would be the intersections lying left of the sweep line)
  2. the *sweep-line status*, information about the set of objects intersecting $ell$ itself (here, we’ll have the line segments intersecting $ell$ ordered top to bottom)
  3. a subset of *future events* lying to the right of $ell$ that at least contains the next event
Detecting Events

- Observe that most of the time, the line segments intersecting the sweep line don’t change. In fact, they don’t even change their order along the sweep line. The event points will be the x-coordinates where meaningful change happens:
  1. Segment left endpoints
  2. Segment right endpoints
  3. Intersection points (what we’re trying to find!)

- Events of type 1. and 2. can be presorted before the sweep even runs, but we don’t know ahead of time when type 3. events occur (that’s kind of the point).

- Fortunately, we don’t need to know about all future events. We only need to know about enough of them to guarantee we’re aware of the next one that will happen.

- Lemma: Suppose $s_i$ and $s_j$ intersect at some point $p$. Then $s_i$ and $s_j$ are adjacent along the sweep line just after the event that immediately precedes $p$ in the sweep.

- Proof:
  - By general position assumptions, no three segments intersect at a common point.
  - So infinitesimally to the left of $p$, $s_i$ and $s_j$ are adjacent along the sweep line.
  - Let $q$ be the event point immediately before $p$. In the vertical slab between $q$ and $p$, no segments start, stop, or exchange position, so $s_i$ and $s_j$ are adjacent immediately after processing $q$.
  - (This last argument fails for more general curves that can double back.)

- So now we see how to anticipate events: we’ll keep an event queue of future events that include all future events of types 1. and 2., but only those events of type 3. that come from
a pair of adjacent line segments on the sweep line. As we’ll see, there’s only a constant
number of changes to the adjacencies during each event, so we’ll only need to add to or
remove events from the queue a total of $O(n + m)$ times. That’s optimal up to constant
factors!

• (Removing events isn’t strictly necessary, but if we don’t do so, we may end up with
multiple copies of an event in the queue or a queue that takes up more then $O(n)$ space.)

Data Structures

• So now can infer what data structures we’ll need. We only care about finding the earliest
upcoming event, so we’ll store events sorted by x-coordinate in a standard priority queue.
It should support:
  • $r \leftarrow \text{insert}(e, x)$: Insert event $e$ with "priority" $x$ and return a reference $r$ to its entry.
  • $\text{delete}(r)$: Delete $r$’s entry.
  • $(e, x) \leftarrow \text{extract-min}()$: Extract and return the event $e$ with smallest priority $x$.

• For sweep line status, we need a data structure that supports quick insertion and remove
of segments, along with finding what is adjacent to certain segments of interest. We’ll use
an ordered dictionary with the following interface (in future lectures, we’ll say less about
the interface to standard data structures):
  • $r \leftarrow \text{insert}(s)$: Insert segment $s$ and return a reference $r$ to its entry.
  • $\text{delete}(r)$: Delete $r$’s entry.
  • $r' \leftarrow \text{predecessor}(r)$: Return a reference $r'$ to the segment immediately above the entry
    for $r$ (or null if $r$ is the topmost segment)
  • $r' \leftarrow \text{successor}(r)$: Same as before but look below.
  • $r' \leftarrow \text{swap}(r)$: Swap $r$ and its immediate successor, returning reference $r'$ to the new
    entry for $r$.

• We can implement both data structures to use $O(n')$ space and take $O(\log n')$ time per
operation using data structures you’ve seen before where $n'$ is the number of entries in the
data structure. Using a min-heap for the priority queue and a balance binary search tree
for the ordered dictionary would work fine.

• But there is one more detail we need to address: how will the ordered dictionary compare
line segments if the y-coordinate of the intersections with the sweep line keep changing?
Well, we already saw it suffices to look only at intersection heights immediately after event
points. Mount even describes ways to compare segments without doing explicit division in
case you’re concerned about numerical issues.

The Algorithm

• We’re going to repeatedly extract the next event and update the event queue and sweep
line status in response to what kind of event we extract. The details of what happens at
each event are easiest to explain with pseudocode:

<table>
<thead>
<tr>
<th>Line Segment Intersection Reporting</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Insert all of the endpoints of the line segments of $S$ into the event queue. The initial sweep-line status is empty.</td>
</tr>
<tr>
<td>(2) While the event queue is nonempty, extract the next event in the queue. There are three cases, depending on the type of event:</td>
</tr>
<tr>
<td><strong>Left endpoint:</strong> (see Fig. 25(a))</td>
</tr>
<tr>
<td>(a) Insert this line segment $s$ into the sweep-line status, based on the $y$-coordinate of its left endpoint.</td>
</tr>
<tr>
<td>(b) Let $s^+$ and $s^-$ be the segments immediately above and below $s$ on the sweep line. If there is an event associated with this pair, remove it from the event queue.</td>
</tr>
<tr>
<td>(c) Test for an intersection between $s$ and $s^+$, and if so, add it to the event queue. Do the same for $s$ and $s^-$.</td>
</tr>
<tr>
<td><strong>Right endpoint:</strong> (see Fig. 25(b))</td>
</tr>
<tr>
<td>(a) Let $s^+$ and $s^-$ be the segments immediately above and below $s$ on the sweep line.</td>
</tr>
<tr>
<td>(b) Delete segment $s$ from the sweep-line status.</td>
</tr>
<tr>
<td>(c) Test for an intersection between $s^+$ and $s^-$ to the right of the sweep line, and if so, add the corresponding event to the event queue.</td>
</tr>
<tr>
<td><strong>Intersection:</strong> (see Fig. 25(c))</td>
</tr>
<tr>
<td>(a) Let $s^+$ and $s^-$ be the two segments involved (with $s^+$ above just prior to the intersection). Report this intersection.</td>
</tr>
<tr>
<td>(b) Let $s^{++}$ and $s^{--}$ be the segments immediately above and below the intersection. Remove any event involving the pair $(s^+, s^{++})$ and the pair $(s^-, s^{--})$.</td>
</tr>
<tr>
<td>(c) Swap $s^+$ and $s^-$ in the sweep-line status (they must be adjacent to each other).</td>
</tr>
<tr>
<td>(d) Test for an intersection between $s^+$ and $s^{++}$ to the right of the sweep line, and if so, add it to the event queue. Do the same for $s^-$ and $s^{--}$.</td>
</tr>
</tbody>
</table>

- And here are some figures to help visualize what is happening at each type of event:

![left-endpoint event](image1)

![right-endpoint event](image2)

![intersection event](image3)

(a) (b) (c)

*Fig. 25: Plane-sweep algorithm event processing.*

**Analysis**

- The sweep-line status has $\leq n$ segments at all times. The event queue has $O(n)$ events at any time.
- Therefore, events take $O(\log n)$ time to process.
- There are $2n + m$ events processed, so the total running time is $O((n + m) \log n)$.
• There is also an $O(n \log n + m)$ time algorithm I (probably) won’t be giving in this class, as well as an $\Omega(n \log n + m)$ lower bound.