Professor Diane Souvaine
Scribe: David Gentile

Dynamicized Convex Hulls Continued

1 Recap of Last Discussion

Last time, we briefly recapped some of the algorithms we had encountered, including Jarvis's March, Incremental Hull, and Divide & Conquer Hull. We recalled the notion of an abstract data type, mentioning concatenable queues and priority queues from CS160. We also discussed the so-called "quick hull" algorithm, which is really only quick in a handful of circumstances and generally runs in quadratic time.

2 What to Do with Unusual Hulls

So far, we have seen several deterministic algorithms for computing the complex hull of an input set X in the plane, and we reduced the problem of sorting a general list to that of finding a convex hull, thus showing that the best time bound we can hope to achieve with our convex hull algorithms is $\Omega(n \log n)$. Let us now consider some edge cases that might prompt us tweak the techniques we have developed to this point.

Consider the following example: suppose we have a data set where the vast majority lies in one region of the plane, but a handful of extreme outliers define the convex hull of the set and we find ourselves with a relatively simple polygon for our convex hull.

We saw previously that Jarvis's March ran in $\mathcal{O}(nh)$, where $h = |CH(X)|$. For large n in this particular case, this would be quiet a bit faster than our $\mathcal{O}(n \log n)$ methods. That being said, in order to know to employ the Jarvis March here, we would need to have pre-ordained knowledge about the properties of the final hull, and that just isn't a reasonable assumption to make in general. This raises a question: can we integrate this case and find a single algorithm that will perform quickly regardless of the complexity of our final hull? It turns out the answer is yes.

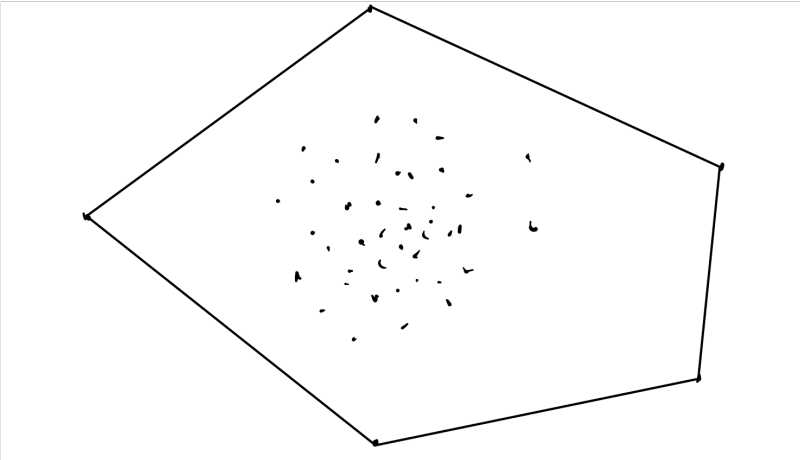


Figure 1: A large set with a simple convex hull

3 Ultimate Convex Hull

3.1 Preamble

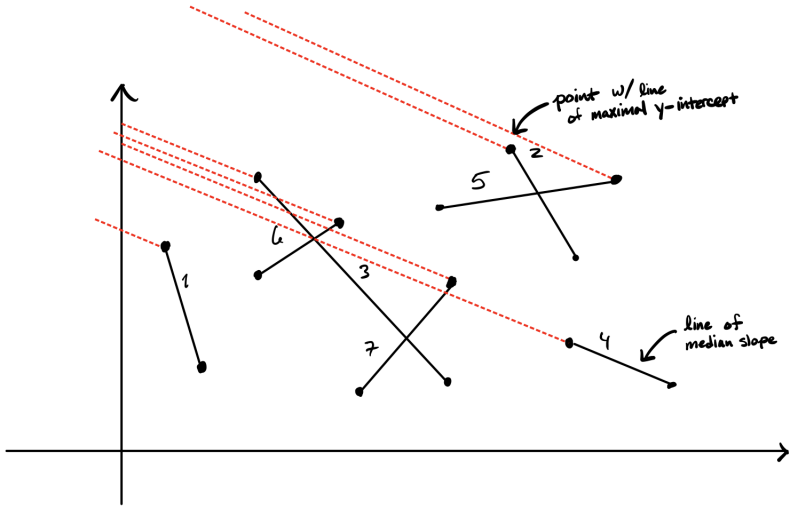
We now introduce the so-called *Ultimate Convex Hull* algorithm. We claim that this algorithm runs in $\mathcal{O}(nlogh)$, meaning that it will in fact beat Jarvis's March handily even in the situation we outlined above. Furthermore, the worse case scenario, where every point in X belongs to $CH(X)$, we see that we are still running in $\mathcal{O}(n \log n)$ time!

UCH is known as a "prune & search" or "marriage before conquest" algorithm. It also has the property being a *dynamicizable* algorithm, meaning that if we can implement it in such a way thatwe can add or remove points freely and quickly. UCH will build the original solution for the set X in $\Theta(n \log n)$ time and update in $\Theta(\log^2 n)$ time.

Before proceeding, we remark that this algorithm depends upon the observation that convex hull construction is an *order-decomposable* problem. That means we can define some ordering function and merging function, where the latter operates iteratively on the ordered input set.

3.2 The Algorithm, Undynamicized

UCH is based on the divide & conquer method we saw previously. The problem with the divide & conquer method as it stands now is that the zig-zagging operation to construct the ladder and find the bridge performs an awful lot of unnecessary work in order to produce the two vertices needed to merge the hulls. Our goal will be to find the bridge first (this is the "marriage" aspect). The key idea is to focus on finding the median line dividing the left hull and the right hull. If the points that define the bridge have coordinates $(x_L, yL), (x_R, y_R)$, then we will want to delete all points satisfying $x_l < x < x_R$. Now we proceed as follows: pair the points with their neighbor in memory (i.e., arbitrarily), and compute the slopes of the resulting lines. We compute the median slope M using the medianFind algorithm from CS160, which we know to run in $\Theta(n)$. We will use this to find the brdige in linear time. For each point p, consider the line $l_{p,M}$ which passes through p with slope M. Sort these lines by their y-intercept. Look at the point corresponding to the line with maximal y-intercept, and delete all the pts on the lower bound of any line with slope less than M, since these points cannot possibly belong to the bridge.


point w/ line of maximal y-intercept
2
5
6
3
1
7
4
line of median slope

Now we consider time bounds. It's clear that our bridge-finding routine is $\mathcal{O}(n)$, since we have

$$T_{bridge}(n) \leq T_{bridge}(n - \lfloor \tfrac{n}{4} \rfloor) + \mathcal{O}(n)$$

We have now that our UCH algorithm satisfies

$$T(n,h) = \left\{ \begin{array}{ll} 0, & h = 1 \\ \mathcal{O}(n), & h = 2 \\ T(\frac{n}{2}, h_l) + T(\frac{n}{2}, h_r) + T_{bridge}(n), & h \geq 3 \end{array} \right\}$$

where $h_l + h_r = h$ Let us show that $T(n,h) \leq cn \log h$.

Proof 3.1 *We will proceed by induction on h. In the case $h = 2$, we have*

$$T(n,2) = c_1 n = c_1 n \log(2)$$

Now suppose that we have shown the claim for $h \leq k$ for some k. Then we have that

$$\begin{aligned} T(n,h) &\leq c_1 n + T(\frac{n}{2}, h_l) + T(\frac{n}{2}, h_r) \\ &\leq c_1 n + \frac{cn}{2} \log h_l + \frac{cn}{2} \log h_r \\ &\leq c_1 n + \frac{cn}{2} [\log h_l + \log h_r] \\ &\leq c_1 n + \frac{cn}{2} \log h_l h_r \\ &\leq c_1 n + \frac{cn}{2} \log h_l (h - h_l) \\ &\leq c_1 n + \frac{cn}{2} \log \frac{h}{2} \frac{h}{2} \\ &\leq c_1 n + \frac{cn}{2} \log \frac{h^2}{4} \\ &\leq c_1 n + cn \log \frac{h}{4} \\ &\leq c_1 n + cn \log h - 2cn \\ &\leq cn \log h \end{aligned}$$

Now we turn our eye towards dynamicization of the algorithm. The key problem is one of storage - recall that for performing merge sort on a list of integers, we found we had data storage issues if we used arrays. In particular we need to keep an entire extra copy of the array to actually do the merging process. This can be improved significantly by using linked lists, so that instead of manipulating copies of values, we can simply adjust pointers as necessary. This will be our inspriation for the dynamic version of UCH. Next time, we will see that by using a balanced binary search tree, in particular a concatenable queue, we can efficiently keep track of our work in constructing the hull so that points can be removed and added quickly.