Homework 1

Elliott Pryor
Worked with: Nathan Stouffer

24 Jan 2021

Problem 1 Let $P = \{p_1, \ldots, p_n\}$ and $P' = \{p'_1, \ldots, p'_n\}$ be the vertex sets of two upper hulls in the plane. Each set is presented as a sequence of points sorted from left to right. Let $p_i = (x_i, y_i)$ and $p'_j = (x'_j, y'_j)$ denote the point coordinates. We assume that P lies entirely to the left of P', meaning that there exists a value z such that for all i and j, $x_i < z < x'_j$.

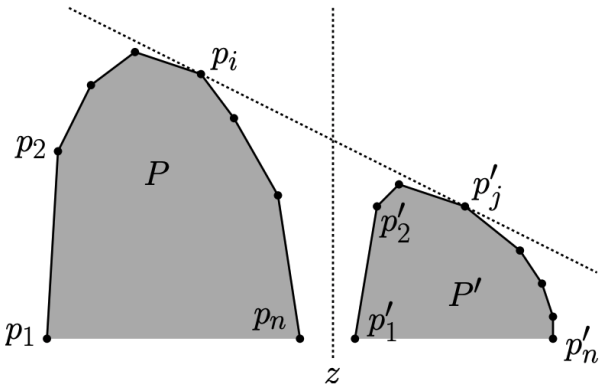

Figure 1: Problem 1: Computing the upper tangent of two hulls

Present an $O(\log n)$-time algorithm which, given P and P', compute the two points $p_i \in P$ and $p'_j \in P'$ such that their common support line passes through these two points.

Briefly justify your algorithm's correctness and drive its running time. (**Hint:** The correctness proof involves a case analysis. Please be careful, a poorly drawn figure may lead to an incorrect hypothesis.)

This is a note. Hey Dave! Nathan Stouffer and I worked for a really long time on this (probably over 10 hours on this one alone). I am very not sure about it. But this is the best we could come up with. I would be very excited to hear about the correct solution.

For this problem, we use the instructions in Figure 2. We draw a line connecting two points, $A \in P$, $C \in P'$. Then depending on which of the 4 cases this falls into in the figure we move accordingly. The step size is $\lceil \frac{n}{2^i} \rceil$. This (step size) eventually converges to 1. After which point we stop. *Note, we run this one iteration extra, so it runs for two stepsize 1 iterations if it is not tangent. We check tangency by orientation $A, C, B = A, C, D$ and orientation $C, A, Y = C, A, X$.

Running time: The step size is logarithmic. So it will converge in $O(\log(n))$. Due to our termination condition, it runs for $log(n) + 1$ iterations. During each iteration, it performs only constant time operations. Thus the running time $O(1 + \log(n)) = O(\log(n))$.

Correctness: No matter what case we are in, we always move the points A, C towards the target. Imagine the scenario where step size is fixed at 1. It will eventually reach the target points. Thus the rules are correct.

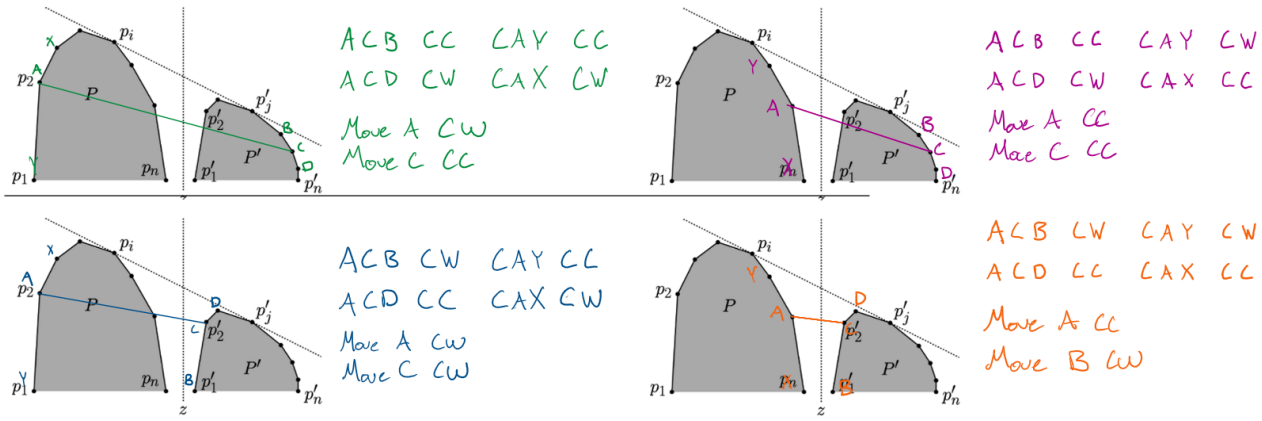

Figure 2: Problem 1 Algorithm

Since we have no way of knowing we have the right points until the end, it is possible that we have it, and move off of this point by $\lceil \frac{n}{2^i} \rceil$. But this is bad you may say! Since we always move towards the target point, and we step at $\lceil \frac{n}{2^i} \rceil$, we will end up after $\log(n)$ within 1 point of the correct target.

Examine the worst case where we step off the point on the first iteration, so we are $n/2$ away. We approach this point each iteration, so our distance away is $n/2 - n/4 - n/8 - ... - n/2^i - ...1$. By geometric series this is 1. Thus we can run our algorithm 1 iteration more (constant time) in order to reach the target.

Also if n is not a power of 2, it is possible to overshoot by the ceil operator. We can either add filler points on lines, or we know that we always move in the right direction, so if we overshoot we switch directions and head back towards the target as desired.

Problem 2

Consider a set $P = \{p_1, \ldots, p_n\}$ of points in the plane, where $p_i = (x_i, y_i)$. A *Pareto set* for P, denoted Pareto(P), (named after the Italian engineer and economist Vilfredo Pareto), is a subset of points p_i such that there is no $p_j \in P (j \neq i)$ such that $x_j \geq x_i$ and $y_j \geq y_i$. That is, each point of Pareto(P) has the property that there is no point of P that is both to the right and above it.

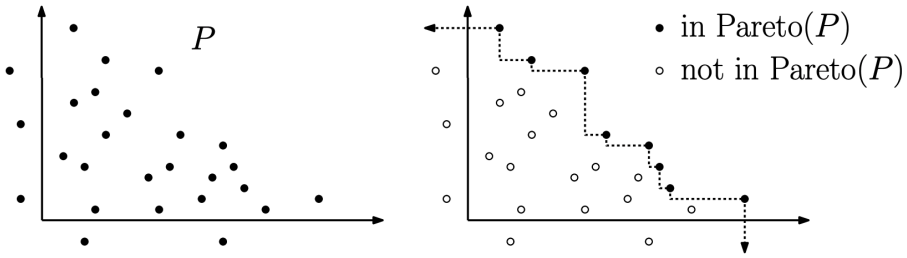

Figure 3: Problem 2: Pareto set

Pareto sets and convex hulls in the plane are similar in many respects. In this problem we will explore some of these connections.

1. (5 points) A point p lies on the convex hull of a set P if and only if there is a line passing though p such that all the points of P lie on one side of this line. Provide an analogous assertion for the points of Pareto(P) in terms of a different shape.
2. (5 points) Devise an analogue of Graham's convex-hull algorithm for computing Pareto(P) in $O(n \log n)$ time. Briefly justify your algorithm's correctness and derive its running time. (You do not need to explain the algorithm "from scratch", that is, you can explain with modifications would be made to Grahm's algorithm.)
3. (5 points) Devise an analogue of the Jarvis march algorithm for computing Pareto(P) in $O(h \cdot n)$ time, where h is the cardinality of Pareto(P). (As with the previous part, you can just explain the differences with Jarvis's algorithm.)
4. (5 points) Devise an algorithm for computing Pareto(P) in $O(n \log h)$ time, where h is the cardinality of Pareto(P).

---

1. The points on the $Pareto(P)$ fall on a series of horizontal and vertical line segments. It follows a line in 'Manhattan' distance. In order to make the analogous assertion, we define corner $C(p)$ $p \in P$ as the region $(x, y) \in \mathbb{R}^2 \;\; x \leq p_x, \;\; y \leq p_y$.

   Then a point p is on the Pareto of a set P if and only if there is no other corner $C(p')$, $\;\; p \neq p'$, containing p

2. This is almost identical to Grahm's Scan. We replace the Orient() function with a different comparison. We simply compare the y values of adjacent points. Since the points are in sorted, x, order if a a point p_i has a larger y-coordinate than p_{i-1} the p_{i-1} would be in $C(p_i)$ so is not on the Pareto. Since this comparison only needs the first point in the stack, we also adjust initialization of S to only push p_1 onto the stack, and to not terminate the while loop unless there are 0 points in the stack.

   This has the same running time as Grahm's Scam. Since after sorting, the algorithm takes a linear pass through all of the points. It also only pops a point at most once from the stack. The only difference with this algorithm from Grahm's Scan is the Orient. Since comparing y-coordinates is also a constant time lookup, our running time is $O(nlog(n))$ due to the sorting in line 2.

Algorithm 1 Pareto Scan

```
1: function ParetoScan(P)
2:     sort P by increasing x value
3:     push p_1 onto stack S
4:     for i ← 2, ..., n do
5:         while |S| ≥ 1 and p_{i,y} ≥ S[top]_y do
6:             pop S
7:         end while
8:         push p_i onto S
9:     end for
10: end function
```

Correctness: We show that algorithm 1 is correct by induction. A pareto front is at least one point, so initialization is trivially true.

We assume that the points on the stack are the pareto front for the first $i-1$ points in P.

Then, we consider point p_i. We have two cases. If $p_{i,y} < S[top]_y$ then, since points are in sorted x order, p is not in the corner of any of the points of S, and none of the points of S are in the corner of p. Thus, $S \cup p$ is the pareto front for the first i points.

Second, if $p_{i,y} \geq S[top]_y$, then $S[top]$ is in the corner of p (since in sorted x order). Then, S gets popped (all popped points meet the above condition) until $|S| < 1$ or $S[top]_y > p_{i,y}$. Thus only points in the corner of p_i were popped, so are not on the pareto front. Now our stack either has size 1 from pushing p and is trivially a pareto front. Or it is a valid pareto front for the first i points since it now meets the first condition.

Thus at the end of the loop, S is a valid pareto front for the first n datapoints, and is thus a valid pareto.

3. Our modified Jarvis march algorithm would search for the point with the greatest y value. Then it would loop and search for the point with the next greatest y-coordinate to the right of the previous point ($p_{i,x} > p_{i-1,x}$). This loop is repeated $h-1$ times (the first point was found in initialization step of finding point with largest y-coordinate).

Algorithm 2 Jarvis Stairs

```
1: function JarvisStairs(P)
2:     find point p_1 with largest y-coordinate
3:     S ← p_1
4:     for i ← 2, ..., h do
5:         find point p_i with largest y-coordinate such that p_{i,x} > p_{i-1,x}
6:     end for
7:     return S
8: end function
```

Correctness: The point with the maximum y value must be a part of the pareto. So at initialization we find the point. We then search for the next greatest point to the right of this. This point is also part of the pareto. We repeat this h times to find the h points with greatest y value to the right of the previous point.

4. We start by dividing P into n/h sets of size h. We run Pareto Scan (Algorithm 1) on each of the n/h sets. The runtime of this step is $O(h\log(h))$ repeated n/h times: $O(n\log(h))$.

   We can then merge these in $O(n)$ time. We examine this in a 2D matrix. Each row is an individual pareto, and they are all at most h long. The algorithm explores this entire matrix, which is $n/h \times h$. So covering this entire matrix takes $O(n/h \cdot h) = O(n)$ time.

   We can also arrive at this an alternative way. Line 3 takes $O(n\log(h))$. Then the for loop is run h times. Line 7 and 8 both take $O(n/h)$ time since they go through each mini-pareto once. Line 9 is constant time. So the for loop takes $O(h \cdot n/h) = O(n)$.

Algorithm 3 Chan Pareto

```
1: function ChanPareto(P)
2:     divide P into P_1, P_2, ...P_{n/h} where |P_i| = h
3:     Solve each P_i using Algorithm 1 and store paretos in S_i
4:     index ← list of n/h ones.
5:     initilize stack S
6:     for 1, ...h do
7:         find point p with largest y coordinate along the indicies (S_i[index[i]])
8:         Go through paretos along indicies and increment index if point S_i[index[i]] is in corner of p
9:         push p onto S
10:    end for
11:    return S
12: end function
```

Correctness: We know that each mini-pareto is correct since Algorithm 1 is correct. So we need to show that the merge operation is correct. We know that points on the pareto are part of the mini paretos. By induction, we start with an empty set which is vacuously true. Then we assume that we have the first $i-1$ points of the pareto. And that $index[i]$ points to the largest uncovered point in pareto S_i.

We show that this holds after the i^{th} iteration. To do this, we show that the point p with the largest y coordinate is part of the pareto. By our choice of p, it is certainly not covered by any of the points on $S_i[index[i]]$. Thus, we show that it is also not covered by any previous point, or future point. By the inductive assumption, p is not covered by any of the points in S. Then, suppose that p is covered by a future point p_j, that future point would also cover $S_j[index[j]]$. Then S_j would not be a valid pareto, a contradiction. So p belongs on the pareto.

Problem 3

Assume you have an orientation test available which can determine in constant time whether three points make a left turn (i.e., the third point lies on the left of the oriented line described by the first two points) or a right turn. Now, let a point q and a convex polygon $P = \{p_1, \ldots, p_n\}$ in the plane be given, where the points of P are stored in an array in counter-clockwise order around P and q is outside of P. Give pseudo-code to determine the tangents from q to P in $O(\log n)$ time.

We say $Orient(a, b, c) > 0$ if it is oriented counter clockwise (makes left turn), and $Orient(a, b, c) < 0$ if it is oriented clockwise (right turn)

Algorithm 4 Tangent Function

```
1: function TANGENT(a, P)
2:     Binary search to find point of tangency
3:     low ← 1
4:     high ← |P|
5:     p1, p2
6:     c ← n/2
7:     for i ← 1, ..., log(n) + 1 do
8:         c ← c mod n
9:         increment ← ⌈n/2^i⌉
10:        i ← i + 1
11:        b ← c − 1,  d ← c + 1    b is point counter-clockwise from a, d is point clockwise from a
12:        if Orient(a, c, b) = Orient(a, c, d) then
13:            found tangent
14:            p1 ← c
15:            break for
16:        else if Orient(a, c, b) > 0 and Orient(a, c, d) < 0 then
17:            Need to move search point left (counter-clockwise)
18:            c ← c + increment
19:        else if Orient(a, c, b) < 0 and Orient(a, c, d > 0) then
20:            need to move search point right (clockwise)
21:            c ← c − increment
22:        end if
23:    end for
24:    Repeat same for loop but flipping inequality signs to find other tangency point.
25:    return p1, p2
26: end function
```

This follows a binary search. Every operation within the while loop takes $O(1)$ time. Then, the same as regular binary search, the algorithm jumps $\lceil \frac{n}{2^i} \rceil$ every iteration, so it covers the entire list in $O(\log(n))$ time. This is geometric series jumps, so it jumps half the remaining distance (logarithmic).

Correctness: We use Figure 4 to justify the rules in the else if statements lines 18-23. This shows the orientation values when we need to move clockwise and counter-clockwise.

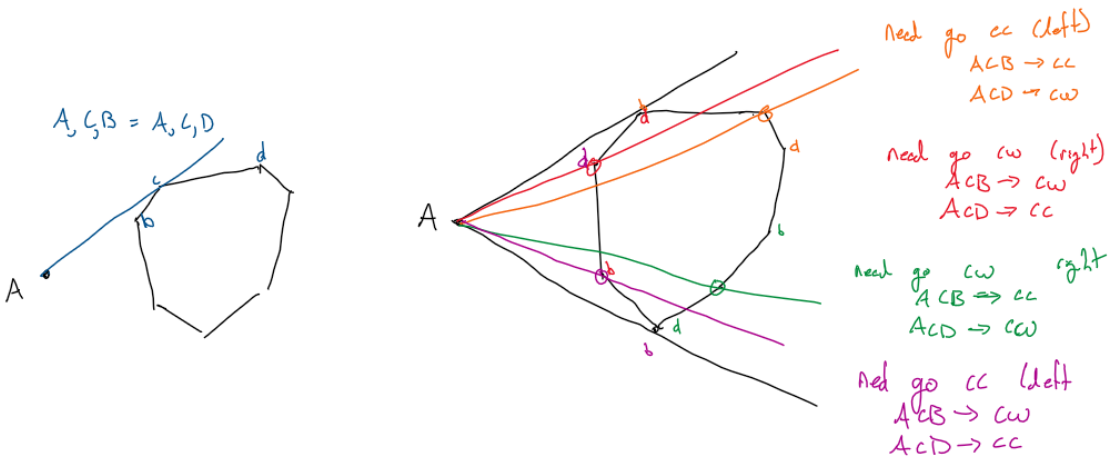

Figure 4: Orientation Rules Justification and example

We know we get there in $\log(n) + 1$ iterations. Due to our step size, in $\log(n) + 1$ iterations we can go all the way around any shape. From Figure 4 we know that we always move in the correct direction. Therefore we must reach the tangency point.

Problem 4

Given a set S of n points in the plane, consider the subsets

$$\begin{aligned} S_1 &= S, \\ S_2 &= S_1 \setminus \{\text{set of vertices of conv}(\mathrm{S_1})\} \\ &\dots \\ S_i &= S_{i-1} \setminus \{\text{set of vertices of conv}(\mathrm{S_{i-1}})\} \end{aligned}$$

until S_k has at most three elements. Give an $O(n^2)$ time algorithm that computes all convex hull $\text{conv}(\mathrm{S_1}), \text{conv}(\mathrm{S_2}), \dots, \mathrm{S_k}$. [Extra credit, provide an algorithm that is faster than $O(n^2)$].

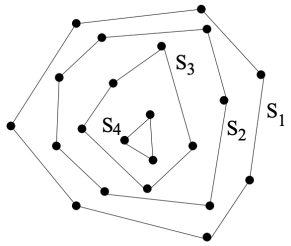

Figure 5: Problem 4: Onion peeling

We essentially do Grahm's Scan, but instead of popping and removing them, we move the popped hull sections down one layer.

So for the runtime, the sort operation takes $O(n \log(n))$. Then at worst case, every single point gets sifted down every shell. There are at most $O(n)$ shells. So it takes $O(n^2)$ time.

Correctness: We prove this inductively on each shell. So considering just the outer shell, we have the exact same as Graham's Scan algorithm, so it is correct. The inner hulls are also correct since Grahm's Scan is run on each of them.

Then for each of the lower shells, we take a convex shell segment and add it to the lower shell. We show that the shell segment gets added in its entirity.

Suppose not, suppose the entire shell segment S' should not be added. We say the first point in the shell segment is p. And the shell we consider is S. Suppose only the first portion of the shell segment can be added without breaking convexity. Then the remaining portion can be added without breaking convexity since the entire shell segment was convex.

Then suppose that some point that was popped, should be added partially through the segment. This is a contradiction since we are building the upper hull and the points are in sorted order. So any popped point has $x < p_x$. If we add the popped point back, the sorted x ordering is broken a contradiction.

Algorithm 5 Onion Problem

```
1: function RECURR(p', S)
2:     popped = []
3:     while |S| ≤ 2 and Orient(p', S[top], S[top − 1]) < 0 do
4:         popped.append(S.pop())
5:     end while
6:     return popped
7: end function
1: function ONIONS(P)
2:     sort P by increasing x
3:     push p_1, p_2 onto stack S_0
4:     for i ← 3, ..., n do
5:         Variable initialization to clean things up
6:         S ← S_0
7:         add_to_next ← [p_i]    mark p_i to be added to S
8:         popped ← recurr(p_i, S)    get points removed from S
9:         while popped is not empty do
10:            S.push(add_to_next)    add the points to this shell
11:            add_to_next ← popped
12:            p' ← popped[top]
13:            S ← next layer down stack
14:            popped ← recurr(p', S)
15:        end while
16:        S.push(add_to_next)    Add to last layer
17:    end for
18:    repeat to make the lower hull.
19: end function
```