CSCI 534: Homework 01

Nathan Stouffer – Collaborated with Elliott Pryor

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, $x_i < z < x'_j$.

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 derive its running time. (**Hint:** The correctness proof involves a case analysis. Please be careful, a poorly drawn figure may lead to an incorrect hypothesis.)

Answer:

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 the smallest subset of points such that for all $p_i \in P$ there exists a $p_j \in$ Pareto(P) such that $x_i \leq x_j$ and $y_i \leq y_j$.

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

1. 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. 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. 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. Devise an algorithm for computing Pareto(P) in $O(n \log h)$ time, where h is the cardinality of Pareto(P).

Answer:

1. Our goal is to provide some geometric condition that conveys whether a point is a member of Pareto(P). For the convex hull, a point p was on the convex hull of a set P if and only if there is a line passing through p such that all points of P lie on one side of the line. This is equivalent to requiring that there exists a half plane (with p on the line defining the half plane) such that all points in P are on one side of the half plane. For the Pareto(P), we have a similar requirement with a "quarter" plane. Consider a plane with p at the origin and axes in the regular directions. Then p is in Pareto(P) if and only if no points in P are inside (or on the border) of the first quadrant of the plane centered at p. If p is in the Pareto(P) then no point p' has both $x' \geq x$ and $y' \geq y$ so no values can be in the first quadrant. Proving the converse (via contrapositive), if some point p' is inside the first quadrant, then both $x' \geq x$ and $y' \geq y$ so p cannot be in Pareto(P).

2. First we give a quick description of the algorithm (relative to Graham's Scan). Instead of sorting the points in increasing order (like Graham's Scan), sort them in decreasing order according to their x coordinate: $P = \{p_1, p_2, ..., p_n\}$ where $i < j$ implies $x_i > x_j$. Then push p_1 on to the stack S. Then for i from 2 to n, if $y_i \geq S[top]_y$ (where $S[top]_y$ is the y coordinate of the point $S[top]$) then push p_i to the stack. We give the pseudocode below.

   Now let's analyze the runtime. The sorting step takes $O(n \log n)$ time and then we do a scan through all the data points in $O(n)$ time. So the total run time for this algorithm is $O(n \log n) + O(n) = O(n \log(n) + n) = O(n \log n)$.

Algorithm 1 Computing Pareto(P)

```
function ParetoScan(P)
    sort P in decreasing order according the x coordinates
    push p_1 on to stack S
    for i ← 2, 3, ..., n do
        if y_i ≥ S[top]_y then
            push p_i on the S
        end if
    end for
    return S
end function
```

Now we give a discussion of correctnes. Post sorting, we know $p_1 \in$ Pareto(P) by the following line of reasoning. The definition of Pareto(P) is the smallest subset of points such that for all $p_i \in P$ there exists a $p_j \in$ Pareto(P) such that $x_i \leq x_j$ and $y_i \leq y_j$. The point p_1 is the point with the largest x coordinate, so if $p_1 \notin$ Pareto(P) then there would be no point in Pareto(P) with a larger x coordinate. Similarly, the point with the largest y coordinate is also in Pareto(P); we will use this fact in the next paragraph.

Now let $P_i = \{p_1, p_2, ..., p_i\}$. We claim that after attempting to insert p_i to S_i (the stack at iteration i), the stack S_i contains $\mathrm{Pareto(P_i)}$ and $S[top]$ has the largest y coordinate in P_i. Consider the base case $P_1 = \{p_1\}$ then $\mathrm{Pareto(P_1)} = \{\mathrm{p_1}\}$ which matches S_1 since p_1 is inserted at the beginning and $S_1[top]$ has the largest y coordinate since there is only one point. Now suppose that we have $S_i = \mathrm{Pareto(P_i)}$ where $i \in \{1, 2, ..., n-1\}$ and $S_i[top]$ has the largest y coordinate in P_i. We attempt to prove that S_{i+1} contains $\mathrm{Pareto(P_{i+1})}$. Since $P_{i+1} = P_i \cup \{p_{i+1}\}$ and S_{i+1} at least contains $\mathrm{Pareto(P_i)}$ we need only check p_{i+1}. We already know p_{i+1} is the furthest left point we have considered. So, if $y_{i+1} < S[top]_y$ then $S[top]$ is a point that prevents p_{i+1} from being in $\mathrm{Pareto(P_{i+1})}$. If $y_{i+1} \geq S[top]_y$ then p_{i+1} has the largest y coordinate so $p_{i+1} \in$ Pareto(P). The algorithm matches these actions by inserting or not inserting p_{i+1} into S_{i+1}. Then when the algorithm terminates we have Pareto(P).

3. Now we give an algorithm similar to the Jarvis march that computes Pareto(P) in $O(h \cdot n)$ time. For this algorithm, we only need to find h points that we know to be in Pareto(P). First we scan through the data set to find the point p with the right-most x coordinate. Then let p_k be the last point added to the pareto. We scan throught the data set to find the point right most point p_i such that $y_i \geq y_k$. We then repeat the previous step $h-1$ times (since we already have one point from step 1). We give the pseudocode below.

   As far as run time goes, the algorithm performs h scans of a data set with size n so the run time is $O(h \cdot n)$. Now for correctness, we assume that |Pareto(P)| = h so we need only find h points that must be part of Pareto(P). To show correctness, we will just show that each point added must be the next point in Pareto(P) (sorted in decreasing x coordinates). The first point added (the right most) is certainly in Pareto(P), as discussed before. Now suppose that we have the first i points of the Pareto(P) in decreasing order (the partial pareto). We show that the next point added to the partial is the $i+1$ point in Pareto(P). The next point p_{i+1} we add is the furthest right point in the data set that is above the p_i point (the i^{th} point in the partial). Suppose there were some point p' in the final pareto in between p_{i+1} and p_i.

Algorithm 2 Computing Pareto(P)

function ParetoStairs(P, h)
    scan P to find right most point p_1
    push p_1 on to stack S
    for $i \leftarrow 2, 3, ..., h$ **do**
        scan P to find the furthest right p_j such that $y_j \geq S[top]_x$
        push p_j to S
    end for
    return S
end function

Then we must have $x_{i+1} \leq x' \leq x_i$ and $y' \geq y_i$. But this is a contradiction since p_{i+1} is the furthest right point with a y coordinate larger than y_i so p_{i+1} is the next point in the final pareto. Thus, the output of the algorithm is Pareto(P).

4. We must give an algorithm that computes Pareto(P) in $O(n \log h)$ time where |Pareto(P)| = h. In prose, we will break the initial set P down into n/h subsets of maximum size h. Then we can find the pareto for each subset using Algorithm 2. Finally, we merge the mini paretos into one final pareto. The merging process is somewhat similar to mergesort. For each mini pareto A_k, we have an index i pointing to an element $A_k[i]$ that increments to $i+1$ when we merge the element $A_k[i]$.

Algorithm 3 Computing Pareto(P)

function ParetoChan(P, h)
    break P into $P_1, P_2, ..., P_{n/h}$ such that $|P_i| \leq h$
    compute Pareto($\mathrm{P_i}$) using Algorithm 2, storing each pareto in an array A_k
    find p_1, the furthest right point of any $A_k[0]$
    push p_1 to stack S
    for $i \leftarrow 2, 3, ..., h$ **do**
        find p_j the furthest right element in any A_k such that $y_j \geq S[top]_y$
        push p_j to S
    end for
    return S
end function

We claim the runtime of this algorithm is $O(n \log h)$. Breaking up the subsets takes $O(n)$ time and then computing the pareto of each subset takes $O(h \log h)$. Since there are n/h subsets the entire mini pareto process takes $O(n \log h)$ time. So we are on the right track, as long as we merge quickly enough. Selecting an element to merge takes $O(n/h)$ time since there are n/h mini paretos and we consider one element from each mini pareto. But we must select an element h times so the total merge time is $O(n)$. Therefore, the algorithm runs in $O(n \log h)$ time.

For correctness, we already know that Algorithm 2 is correct so we need only show that the merging is correct. The merging process is similar to proving that Algorithm 2's scan finds the next point in the pareto so we defer to the correctness to that proof.

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.

Answer: We begin by establishing a lemma about tanget lines in the context of this problem: the line $\overline{qp_k}$ is tangent to P if and only if $sgn(orient(q, p_{k+1}, p_k)) \neq sgn(orient(q, p_k, p_{k-1}))$. Going to the right, suppose that the line $\overline{qp_k}$ is tangent to P. Then the points p_{k-1}, p_{k+1} must be on the same side of $\overline{qp_k}$. Then (hopefully) Figure 1 is convincing evidence to say that $sgn(orient(q, p_{k+1}, p_k)) \neq sgn(orient(q, p_k, p_{k-1}))$.

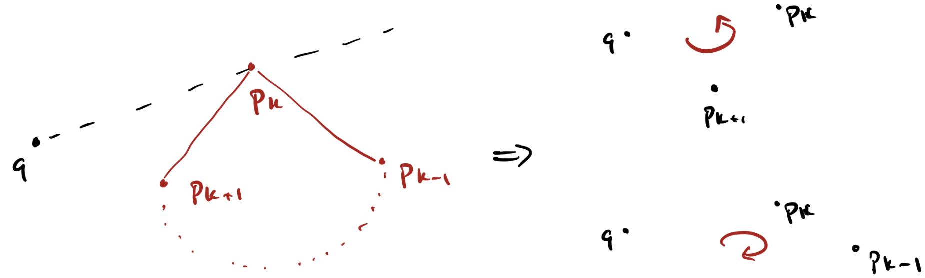

Figure 1: Line $\overline{qp_k}$ is tangent to P

Now going to the left, we will show the contrapositive. Suppose that $sgn(orient(q, p_{k+1}, p_k)) = sgn(orient(q, p_k, p_{k-1}))$. Here we can say that p_{k-1} and p_{k+1} must lie on opposite sides of the line $\overline{qp_k}$ (Figure 2 depicts an example). But then $\overline{qp_k}$ cannot be a tangent line so the contrapositive is shown.

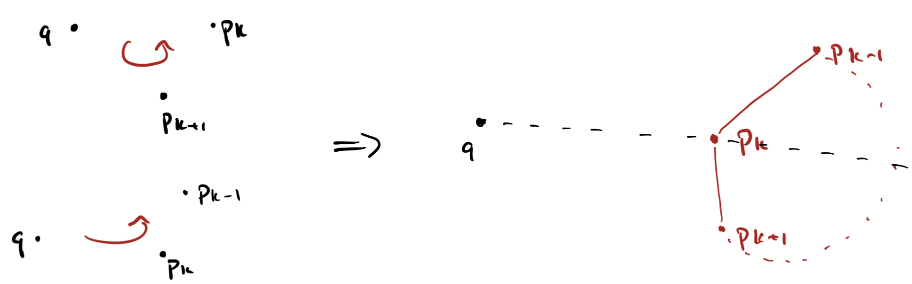

Figure 2: Line $\overline{qp_k}$ is not tangent to P

Now we can use the lemma to construct an $O(\log n)$ algorithm to find the tangent lines from q to P.

Algorithm 4 Compute tangent lines from q to P

$y \leftarrow 1$

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 $\mathrm{conv(S_1), conv(S_2), \ldots, S_k}$. [Extra credit, provide an algorithm that is faster than $O(n^2)$].

Answer: To find all $\mathrm{conv(S_i)}$ in $O(n^2)$ time, we use a modified version of Graham's Scan. In words, we run Graham's Scan on S but keep track of a stack for each $\mathrm{conv(S_i)}$. When we pop a set of points off the stack T_k we move the set down to the next stack and repeat Graham's Scan for the stack T_{k+1}. Here is the pseudocode.

Algorithm 5 Computing the onion

```
function ComputeOnion(S)
    sort points in increasing order according to x
    push p1, p2 on to the first stack T1
    for i ← 3, 4, ..., n do
        T ← T1
        popped ← Pop(A = [pi], T)
        while popped is not empty do
            T ← the next stack
            popped ← Pop(popped, S)
        end while
    end for
    return T1, T2, ..., Tk
end function
function Pop(A[1..k], T)
    initialize popped list L
    while |T| ≥ 2 and Orient(A[1], T[top], T[top − 1]) do
        append T[top] to L and pop T[top]
    end while
    push elements of A on to T
    return L.reverse() // reverse so elements are increasing in x
end function
```

We claim the run time of the algorithm is $O(n^2)$. The sorting process takes $O(n \log n)$ and we already know Graham's Scan to identify sorted points in $O(n)$ time when we only search for the outer convex hull. The maximum number of stacks that we can have is $n/3 = O(n)$ (a bunch of concentric triangles). In the very worst case, each Graham's Scan would take $O(n)$ time and there are $O(n)$ scans so identifying the points takes $O(n^2)$ time.

For correctness, a lot of the work is done because we are using Graham's Scan at each level. Suppose we have a point p that should end up in $\mathrm{conv(S_i)}$. We begin processing the point at level S_1, if $i > 1$ then Graham's Scan will not keep p. Thus it eventually pops p and will pushes it down a level. Since every level is using Graham's Scan, this process will repeat until p encounters the stack for $\mathrm{conv(S_i)}$. Since we are using Graham's Scan to find $\mathrm{conv(S_i)}$, p will never leave the stack for $\mathrm{conv(S_i)}$. Thus the algorithm is correct.