CS634 Homework 2

Qilin Ye

October 7, 2025

Solved and typesetted in a rush. Apologies for typos, as well as casual, terse, and/or ambiguous sentences.

Solution to problem 1. For part (i), observe that if two lines contain P then they also contain the convex hull of P. Furthermore, we claim that the problem is equivalent to finding a strip of minimum width that contains P. To see this, consider a 2-D vector u, and define

$$a^+(u) = \max_{x \in P} \langle u, x \rangle, \qquad a^-(u) = \min_{x \in P} \langle u, x \rangle.$$

Then the two lines $\ell^+ = \{x : \langle u, x \rangle = a^+(u)\}$ and $\ell^-(u) = \{x : \langle u, x \rangle = a^-(u)\}$ encompass all of P and hence the convex hull of P. On the other hand, for the center line

$$\ell(u) = \{x : \langle u, x \rangle = (a^+(u) + a^-(u))/2\},$$

its maximum distance to a point is precisely $(a^+(u) - a^-(u))/2$, i.e., half the width of P in the direction of u.
Observe that the minimum-width strip containing $H = \mathrm{conv}(P)$ must have one of its boundary lines incident with an edge of H, and the other one either incident to another edge or a vertex. Otherwise, if both lines are only touching vertices of H, we may rotate the strip until one side becomes flush with an incident edge, along the way decreasing width.
With this in mind we now describe the algorithm. First compute the convex hull $H = \mathrm{conv}(P)$ which takes $\mathcal{O}(\log n)$. Let $\{v_j\}$ be the vertice sof H. We traverse through the edges of H counterclockwise. For each edge e_i, compute its normal vector u_i and compute the width of P (or H):

$$w_i = \max_j \langle u_i, v_j \rangle - \langle u_i, v_i \rangle.$$

Find the smallest w_i after we iterate over all edges e_i, say with index i^*. Then the thinnest strip must have one side incident with e_{i^*} and the other side going through the maximizer of w_{i^*}. Finally, take the middle line between these two sides and return it.

For (ii), we reduce the problem to (i) by reducing to widths of the convex hull. For a vector u let

$$h_H(u) = \max_{x \in H} \langle u, x \rangle, \qquad W_H(u) = h_H(u) - h_H(-u),$$

which define the support function, and the width of H in direction u. The only difference is that now we are taking two perpendicular strips at a time. For a direction t (say $t \in \mathbb{S}^1$ or an angle $0 \leqslant t \leqslant 2\pi$), let $^\perp$ be the orthogonal direction. Among all rectangles whose sides are parallel to t and $t^\perp$, the unique smallest one containing H is

$$R(t) = \{x : h_H(-t) \leqslant \langle t, x \rangle \leqslant h_H(t), h_H(-t^\perp) \leqslant \langle n, x \rangle \leqslant h_H(t^\perp)\},$$

with side lengths $W_H(t)$ and $W_H(t^\perp)$. Hence, the minimum-area rectangle has area

$$\min_{t\in\mathbb{S}^1} \text{area}(R(t)) = \min_{t\in\mathbb{S}^1} W_H(t)W_H(t^\perp).$$

Solution to problem 2. To facilitate the process we cite Theorem 2.6 from BCKO: overlay of two subdivisions S_1, S_2 of sizes n_1, n_2 can be built in $\mathcal{O}(n\log n + k\log n)$ time, where k is the complexity of the overlay.
We first describe the divide and conquer algorithm.

- Divide: split S into two halves, S_L, S_R, each of size $\leqslant n/2$. Recursively compute the planar subdividions $\mathcal{U}_L$,$\mathcal{U}_R$ induced by the boundaries $\partial U(S_L), \partial U(S_R)$. Each has complexity $\mathcal{O}(|S_L|)$ and $\mathcal{O}(|S_R|)$, respectively, because the union of a sub-family of translates of Δ also has a linear complexity.
- Conquer: build the overlay using BCKO Theorem 2.6, and label its faces "inside $U(S_L)$" or "inside $U(S_R)$." The faces that are inside at least one of the two will form $U(S_L \cup S_R)$; the outer boundaries of these faces are the desired boundary cycles.

Overall, the running time of the merge, by BCKO Theorem 2.6, is $\mathcal{O}(n\log n+k\log n)$. It mains to prove $k = \mathcal{O}(n)$. To do so, we observe that vertices of the overlay consist of two kinds. First kind are vertices already on $\partial U(S_L)$ or $\partial U(S_R)$ — these altogether give $\mathcal{O}(n)$ total. More notably, the second type are crossings between $\partial U(S_L)$ and $\partial U(S_R)$. We claim that every such crossing x is also a vertex of $\partial U(S_L \cup S_R)$. To see this, each of $\partial U(S_L)$ and $\partial U(S_R)$ divides the plane into an "inside" and an "outside" part. Hence, at a proper crossing, there are four local quadrants, among which there is one that is both outside $U(S_L)$ and $U(S_R)$, and hence outside their union. Therefore, x lies on the boundary of the overall union.
Because the boundary of $U(S_L \cup S_R)$ has $\mathcal{O}(|S_L| + |S_R|) = \mathcal{O}(n)$ vertices, the number of such crossings is $\mathcal{O}(n)$. Therefore, the overall complexity is $k = \mathcal{O}(n) + \mathcal{O}(n)$ which is still $\mathcal{O}(n)$, so the merge takes $\mathcal{O}(n\log n)$ time. Thus, the recursion is given by

$$T(n) = 2T(n/2) + \mathcal{O}(n\log n)$$

from which we conclude $T(n) = \mathcal{O}(n\log^2 n)$.

Solution to problem 3. Let $X = \{x_1 < \cdots < x_m\}$ be the sortesd list of all x-coordinates of segment endpoints of $R \cup B$. We build a segment tree T on the elementary intervals (x_i, x_{i+1}) as leaves. Each node v represents a slab $I(v) = [\ell(v), r(v)]$, the union of consecutive elementary intervals under it. Finally, for every segment s with x-span $[x_s^-, x_s^+]$, we store it at every node v whose interval $I(v)$ is a subset of s but whose parent's $I(v)$ isn't. These are just standard constructions.
Because blue segments are pairwise disjoint, their vertical order along any vertical line inside a slab is well defined and the same across the slab. For every node v with slab $[\ell(v), r(v)]$, we store:

- $B^L(v)$, the list of blue segments stored at v, sorted by their y-coordinates at $x = \ell(v)$, and
- $B^R(v)$, the list of blue segments stored at v, sorted by their y-coordinates at $x = r(v)$.

The orders are exactly the same, but we keep both sorted *value* sequences for binary search later. Again we've done something similar in class when going over segment intersection, so nothing special here.
Now fix a node v and a red segment r stored at v. Let $y_L(r)$ be the y-coordinate of r at $x = \ell(v)$, and $y_R(r)$ be the y-coordinate of r at $x = r(v)$. Let i_L be the insertion rank of $y_L(r)$ in $B^L(v)$ and i_R the insertion rank for

$B^R(v)$. Clearly, the number of blue segments that r meets inside the slab $I(v)$ is $|i_L - i_R|$ (each shift in rank is caused by precisely one intersection with some blue segment). Hence, for node v, we add $|i_L - i_R|$ to the answer for every red segment stored at v. Now iterate over all segments, and obtain the final output.

Clearly we do not undercount. To argue that we do not over count, let p be the intersection of any red r and blue b. Consider the largest (w.r.t. graph hierarchy) node v whose interval $I(v)$ is contained in both x-spans of r and b. By the storage rule, both r, b are stored at v, but at any ancestor, at least one of (r, b) is not stored. In v, the ranks of r among the blues differ between $\ell(v)$ and $r(v)$, so $|i_L - i_R| \geqslant 1$ and contributes exactly one for pair (r, b). No other node contributes to (r, b). Therefore every red-blue intersection is counted once.

Finally, for complexity, we note that each red segment is stored in $\mathcal{O}(\log n)$ nodes, and at each node, we do two binary searches on both endpoints, costing $\mathcal{O}(\log n)$ time. Overall, there are $|R| = \mathcal{O}(n)$, so multiplying everything, the overall runtime is $\mathcal{O}(n \log^2 n)$, and the space (segment tree and all lists) is $\mathcal{O}(n \log n)$.

To improve runtime, we need to optimize the bottleneck of double binary searches — currently they are performed at every visited node, which is causing too much overhead. To use fractional cascading, for each node v we augment two additional structures:

- $B'^L(v)$ obtained by merging $B^L(v)$ with every *second* element of the child's augmented list, with pointers from each entry to the predecessor position(s) in the children, and
- $B'^R(v)$ the analogous structure.

For a fixed red segment r, we now perform the following:

- Run one binary search for $y_L(r)$ in $B'^L(\text{root})$ and another one for $y_R(r)$ in $B'^R(\text{root})$. Note we no longer run this at v.
- Traverse through the segment tree to the $\mathcal{O}(\log n)$ nodes where r is stored. When going from a node to a child, follow the stored pointers and adjust in $\mathcal{O}(1)$ steps to obtain the exact ranks in the child's list. Now, instead of the double binary search, we are able to extract i_L, i_R in each visited node in constant time.
- Return the sums of $|i_L - i_R|$ like before.

Now, the processing time for each red segment reduces fro $\mathcal{O}(\log^2 n)$ to $\mathcal{O}(\log n)$, and overall the runtime becomes $\mathcal{O}(n \log n)$.

Solution to problem 4. For part (a), let $v_1, \cdots, v_n$ be the vertices of P stored in CCW order. For a query point $q \notin P$, and each edge $e_i = (v_i, v_{i+1})$ (modulo n if needed), define $s(i) = \text{sgn}((v_{i+1} - v_i) \times (q - v_i))$ so that $s(i)$ is positive iff q lies to the left of edge e_i. It follows that as we traverse through the edges, the sign of $s(i)$ will only change twice: once it switches from an "invisible" edge to a "visible" one, and once from visible to invisible. Hence, it suffices to do binary search to find the adjacency indices at which $s(i), s(i+1)$ changes value. Doing so takes $\mathcal{O}(\log n)$ time, as stated. Once we identify the two pairs of edges, the vertices that lie in their intersections are the ones we seek. The tangents of q are precisely q to either of these vertices.

For part (b), we following the hint and partition S into $m = \lceil n/k \rceil$ disjoint subsets $S_1, \cdots, S_m$. For every i, compute the convex hull $H_i = \text{conv}(S_i)$ and store its vertices in circular order. Overall, this preprocessing takes $\sum_i |S_i| \log |S_i| = \mathcal{O}(n \log k)$ time.

For each step of Jarvis's march, given the current hull vertex p and the incoming direction, we will:

- for each group S_i with $p \notin S_i$: compute the two tangents using (a). In $\mathcal{O}(\log k)$ time this yields the touching vertex of H that gives the smallest positive turn from the reference direction. Call this c_i.
- Find the unique group S_j with $p \in S_j$. This is done in $\mathcal{O}(1)$ time since p is a vertex of H_j: just test its two neighbors and keep the one that gives smaller positive turn.
- Choose the next hull vertex. Among the m candidates $\{c_j\}$, pick the one with the smallest positive turn from the reference direction.

with $m = \lceil n/k \rceil$, this completes one step of the algorithm in $\mathcal{O}((n/k)\log k)$ time as claimed. (Note the preprocessing is unsurprisingly the bottleneck again, but in this question we are only concerned with the runtime of a step of the March.)

Solution to problem 5. We will use the "lift and linearize" approach as seen in lecture. Write the squared power distance

$$d(p,x)^2 = \|x-p\|^2 - w_p^2 = \|x\|^2 - 2p\cdot x + (\|p\|^2 - w_p^2).$$

As the query point x is fixed and independent of p, we transform the minimization into a maximization problem:

$$\operatorname*{argmin}_{p\in P} d(p,x) = \operatorname*{argmin}_{p\in P} d(p,x)^2 = \operatorname*{argmin}_{p\in P} \left(\|x\|^2 - 2p\cdot x + \|p\|^2 - w_p^2\right) = \operatorname*{argmax}_{p\in P} h_p(x)$$

where $h_p(x) = 2p\cdot x - (\|p\|^2 - w_p^2)$. And observe $\{h_p\}$ is a set of linear functions! Now we perform the first lift by mapping $(p_x, p_y) \mapsto (p_x, p_y, \|p\|^2 - w_p^2)$. Let $U(x) = \max_{q\in P} h_q(x)$ be the upper envelope of these planes. Then

$$x \in \text{cell}(p) \iff h_p(x) = U(x) \iff h_p(x) \geqslant h_q(x) \text{ for all } q \in P.$$

Therefore,

$$\text{cell}(p) = \bigcap_{q\neq p} \{x : h_p(x) \geqslant h_q(x)\},$$

which is the intersection of half planes and hence a convex polygon, as claimed, though possibly degenerate.

As for algorithm, we recall Delanuay triangulation trick: the Voronoi diagram of P is the planar dual of the Delanuay triangulation. We perform the following:

- We build the triangulation. Under the lifting, we compute the lower convex hull of these three-dimensional points, say by randomized incremental algorithm in $\mathcal{O}(n\log n)$ expected time.
- From the convex hull, compute the lower envelope [?], so each face yields a triangle of form (p_i, p_j, p_k) used in DT.
- We then dualize the problem. For every triangle (i,j,k) in DT, compute its circumcenter by solving

  $$h_i(x) = h_j(x) = h_k(x), \qquad h_\ell(x) = 2p_\ell\cdot x - (\|p_\ell\|^2 - w_\ell^2).$$

  For every primal edge (i,j): if it is adjacent to triangles (i,j,k_1) and (i,j,k_2), then add Voronoi edge $v_{i,j,k_1} \leftrightarrow v_{i,j,k_2}$. If it lies on the convex hull, emit the ray from $v_{i,j,k}$ in the direction of the line $h_i = h_j$.
- The resulting planar straight-line graph is precisely the weighted Voronoi diagram we seek. Each face corresponds to a convex cell, possibly degenerate.

For complexity, the runtime bottleneck is from RI which takes $\mathcal{O}(n\log n)$ expected time. The remaining steps are linear in the number of faces, hence $\mathcal{O}(n)$. Storage is $\mathcal{O}(n)$.