Solutions to Quiz 1

Solution 1:

(a) Vertex c is helper(e). The helper is defined to be the closest vertex to the sweep line that lies in the connected region of P's interior that is immediately to the left of the sweep line. In the figure, d is the closest to the sweep line, but it is not the same connected region as e.

(b) There are several valid solutions. One is to test that the two diagonals of the quadrilateral, $\overline{ac}$ and $\overline{bd}$, intersect each other, or equivalently, a and c lie on opposite sides of the line $\overline{bd}$ and b and d lie on opposite sides of the line $\overline{ac}$. This yields the tests

$$\text{orient}(a, c, b) \neq \text{orient}(a, c, d) \text{ and } \text{orient}(b, d, a) \neq \text{orient}(b, d, c).$$

Another way is to test that consecutive triples of vertices turn in the same direction. This leads to the test

$$\text{orient}(a, b, c) = \text{orient}(b, c, d) = \text{orient}(c, d, a) = \text{orient}(d, a, b).$$

It is noteworthy that this approach works for a quadrilateral, but it fails if there are more points. For example, extending this idea to determine whether five points form a convex pentagon fails to work. As seen in Homework Assignment 1, seven orientation tests, not five, are needed.

(c) In class we showed that points on the upper hull are mapped to edges on the lower envelope and vice versa. We also showed that the left to right order of points on the upper hull maps to increasing slope order on the lower envelope. Slope ordering corresponds to a left-to-right order on the upper envelope and a right-to-left order on the lower envelope. Thus we have:

- If p_i and p_j are on the **upper** hull of P with p_i to the **left** of p_j, then p_i^* and p_j^* are on the **lower** envelope of P^* with p_i^* to the **right** of p_j^*.
- If p_i and p_j are on the **lower** hull of P with p_i to the **left** of p_j, then p_i^* and p_j^* are on the **upper** envelope of P^* with p_i^* to the **left** of p_j^*.

Solution 2: Here is a divide-and-conquer algorithm for computing Pareto sets. If $|P| = 1$, then we simply returns the sole point. Otherwise, we partition the point set into roughly equal-sized subsets P_1 and P_2 (arbitrarily), recursively compute the Pareto sets of each subset, and then we merge the two Pareto sets.

For the merging process, R_1 and R_2 denote the two sequences of Pareto points, sorted in increasing order of x-coordinates. The merging process begins by merging these two sequences based on their x-coordinates. Since both sequences are sorted, this can be done in $O(n)$ time. We proceed in a similar manner to Graham's scan. We iterate through all the points in increasing order of x-coordinate. We assume inductively that the Pareto set to its left is stored in a stack, where the bottom-to-top order in the stack corresponds to a left-to-right order of points on the

Pareto set. For each point $p_i = (x_i, y_i)$, we iteratively pop all points from the stack whose y-coordinate is less than or equal to y_i. Observe that because the points are sorted in left-to-right order, x_i is greater than all the points of the stack, and hence any point whose y-coordinate is smaller than y_i is dominated by p_i, and hence is not in the final Pareto set. Also observe that, by the monotonically decreasing nature of the Pareto sequence, one we see a point on the stack whose y-coordinate exceeds y_i, all prior points are not dominated by p_i. (These two observations establish the algorithm's correctness.) Following this, we push p_i on the stack.

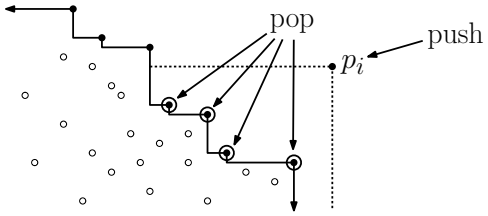


Figure 1: Merging Pareto sequences.

The correctness of the algorithm has already been established. The running time of the merging process is $O(n)$. This is the time needed to merge the two sequences based on their x-coordinates, and as in Graham's scan, each point is pushed once and popped at most once.

Since the merging process takes $O(n)$ time, the overall running time (up to constant factors) is given by the recurrence $T(n) = 2T(n/2) + n$, which is well known to solve to $O(n \log n)$, as desired.

Solution 3: We will make the general-position assumption that all segment endpoints are distinct. First, sort each set R and B by their intersection with the square's left side. Let $R = \langle r_1, \ldots, r_n \rangle$ and $B = \langle b_1, \ldots, b_m \rangle$ denote the sorted sequences. This can be done in $O(n \log n + m \log m) = O((n+m)\log(n+m))$ time. Next, consider the contour C formed by concatenating the square's lower side, right side, and top side, oriented counterclockwise (see Figure 2(a)). Observe that, because there are no intersections among segments of the same color, the order in which segments of the same color intersect this contour is the same as the order in which they intersect the left edge. (Of course, the order may differ between segments of different colors.)

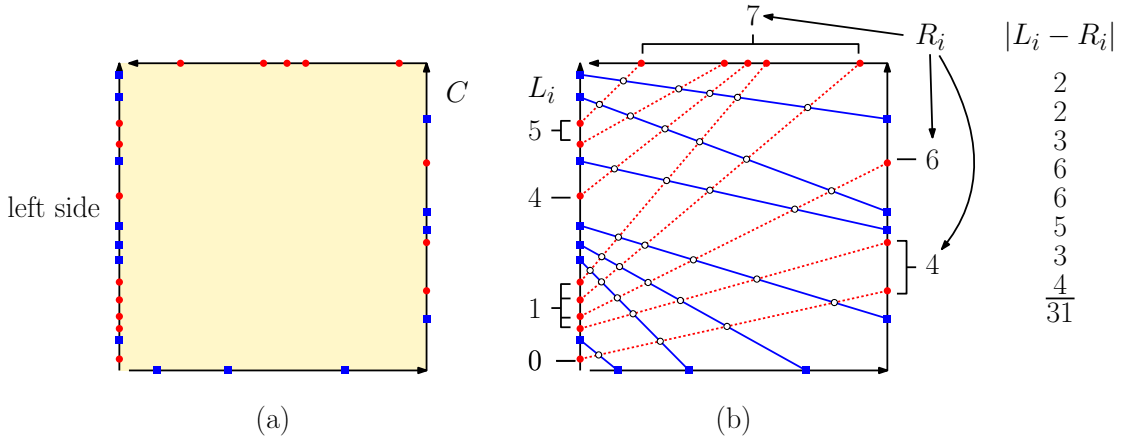


Figure 2: Counting red-blue intersections.

For each segment $r_i \in R$, define its *left rank*, denoted L_i, to be the number of blue segments whose y-intersection along the square's left edge lies below this segment. Define its *right rank*, denoted R_i, to be the number of blue segments whose intersection (see Fig. 2(b)). Clearly, the number of blue segments that intersect r_i is just the absolute difference $|L_i - R_i|$, since this reflects the number of blue segments that cross from one side of r_i to the other.

Therefore, the desired number of intersections is $\sum_{i=1}^{n} |L_i - R_i|$. We can compute L_i and R_i for all i in $O(n+m)$ time by merging the two sorted lists, first by the left vertical side and then by the right vertical side. Combined with the initial sorting time, the entire process takes $O((n+m)\log(n+m))$ time.

Note: We can also solve the problem by reducing it to the Homework 2 solution. Rather than sorting the right endpoints along C, we could extend all segments to the right until they intersect the vertical line through the square's right side, then sort the segments along that line. Although the relative order of segments may differ from the order along C, the R_i values are unaffected, so the result is the same.

Solution 4: We will present an answer to the full version. Observe that the simple case arises as a special case, where the number of equivalence classes is nonzero. We say that two x-monotone paths from s to t are *homotopically equivalent* if they pass above/below the same subset of holes, and otherwise they are *homotopically distinct*. Define a *channel* of P to be a vertical segment whose endpoints lie on the boundary of P, and that lies entirely within the interior of P. Observe that for any channel, every point on the channel has the same number of homotopically distinct paths reaching it from s. We will compute the number of homotopically distinct paths for each channel in the polygon (see Fig. 3(a)).

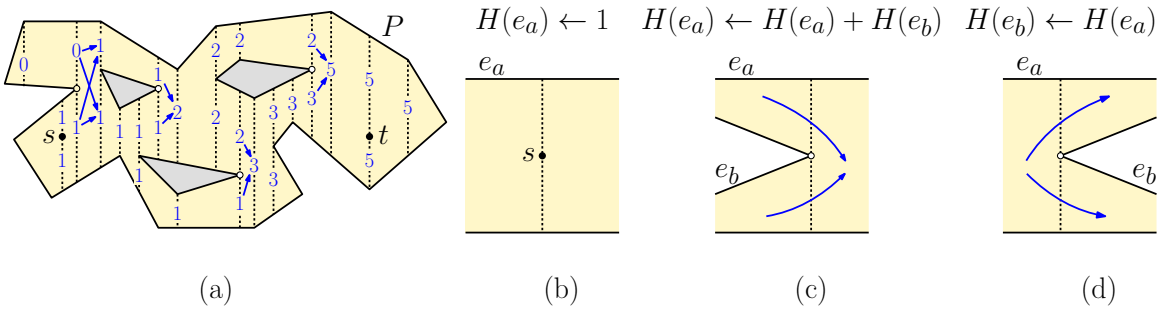


Figure 3: Counting the number of homotopy equivalence classes from s to t.

The algorithm involves a plane sweep through P. In addition to the vertices, we add s and t as two events. For each upper edge of each channel along the sweep-line status, we store a value $H(e)$, which stores the number of homotopy equivalence classes for the channel immediately below e. The processing is similar to that of the monotone decomposition algorithm from Lecture 5, and in the description below, we focus principally on how the Hvalues are updated. The processing breaks into cases, depending on the vertex type. (These types were defined in Lecture 5 on polygon triangulation.)

Start vertex v: Letting e_a denote the upper edge incident to v, set $H(e_a) \leftarrow 0$, reflecting the fact that we have not yet seen s.

See s: Letting e_a denote the edge above s in the sweep-line status, set $H(e_a) \leftarrow 1$, reflecting the fact that there is now one homotopically distinct path leaving s (see Fig. 3(b)).

Merge vertex v: Letting e_a denote the edge above v and e_b the lower edge incident to v, set $H(e_a) \leftarrow H(e_a) + H(e_b)$, reflecting the fact that the two ways around this vertex are merging (see Fig. 3(c)). Note that this works irrespective of whether v is the rightmost vertex of an hole. If not, then one of the two H values will be zero.

Split vertex v: Letting e_a denote the edge above v and e_b the lower edge incident to v, set $H(e_b) \leftarrow H(e_a)$, reflecting the fact that we can use any of the existing paths to enter into the channel below e_b (see Fig. 3(d)).

See t: Letting e_a denote the edge above t in the sweep-line status, return $H(e_a)$.

Upper vertex v: Propagate the value H to from the left edge to the right edge.

We claim that the processing given above correctly maintains the number of homotopically distinct paths into each channel. The key cases are when we encounter s, which is when we initialize the H-value to 1, and whenever we reach a merge vertex (particularly the rightmost vertex of a hole), where we take the sum of the two counts, combining the paths above the hole with the paths below the hole.

The running time is the same as for the monotone decomposition algorithm, $O(n \log n)$. There are n events, which can be presorted in $O(n \log n)$ time. Each event involves a constant number of dictionary operations, each of which takes $O(\log n)$ time, and the reachable values can be updated in constant time per event.