Solutions to Homework 2

Solution 1: Let P_n and P_m denote the two polygons, where P_n has n edges and P_m has m edges. Throughout, we assume general position.

(a) For the upper bound, observe that each edge of P_n can intersect each edge of P_m at most once, and hence the total number of intersections is at most nm. For the lower bound, see Fig. 1(a).

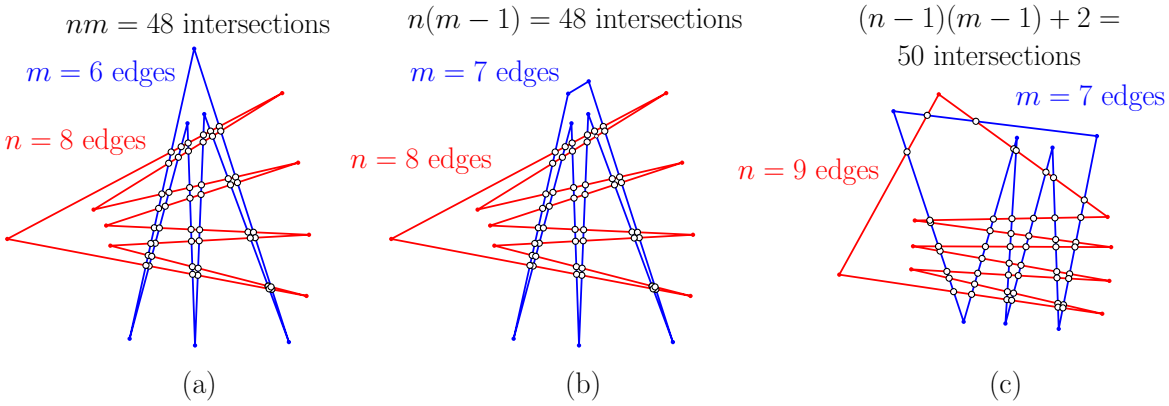


Figure 1: Number of boundary intersections between simple polygons.

(b) In the case where m is odd, observe that each edge of P_n can only intersect at most $m-1$ edges of P_m. To see why, suppose that the initial vertex of P_m lies above the line containing this edge. Each time we intersect an edge of P_m, we move to the other side of this line. Since we must return to the same side of the line, we can intersect at most $m-1$ of the edges of P_m. Repeating this argument for each of the n edges of P_n implies that the total number of intersections is $n(m-1)$, as desired. For the lower bound, see Fig. 1(b).

(c) For the lower bound, see Fig. 1(c). The upper bound seems hard to prove. A proof that the bound is of the form $(n-1)(m-1)+C$ for some constant C can be found in E. Ackerman, B. Keszegh, and G. Rote, "An Almost Optimal Bound on the Number of Intersections of Two Simple Polygons." *Discrete Comput. Geom.* 68, 1049–1077 (2022). `https://doi.org/10.1007/s00454-022-00438-0`. Unfortunately, the best known bound on the constant C is $2^{2^{67}}$ (which, as constants go, is quite large!)

Solution 2: We will make the general-position assumption that no two lines intersect the left and right sides at the same endpoint. First, sort each set R and B by their intersection with the left vertical line $x = 0$. 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. Because the segments from each set do not intersect, each set intersects the two vertical lines in the same order. Their relative orderings of the red and blue intersection points may differ, however.

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 $x = 0$ lies below this segment, and define its *right rank*, denoted R_i, to be the number of blue segments whose y-intersection along $x = 1$ lies below it along $x = 1$ (see Fig. 2). 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.

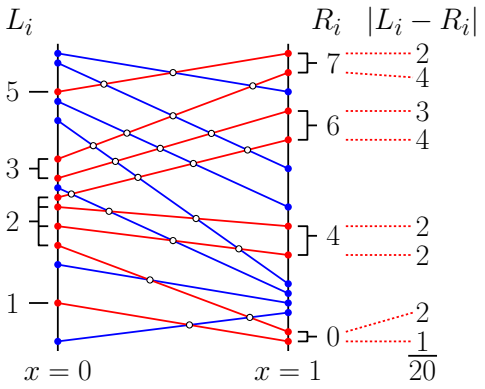


Figure 2: Counting intersections.

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.

Solution 3:

(a) We say that a point $p \in P$ is *monotonically reachable* from s if there exists an x-monotone path from s to p. Our approach will be to compute the monotonically reachable portion of P from s. Define a *channel* of P to be a vertical segment whose endpoints lie on boundary of P that lies entirely within the interior of P. Observe that for each channel, either all its points are monotonically reachable from s or none are.

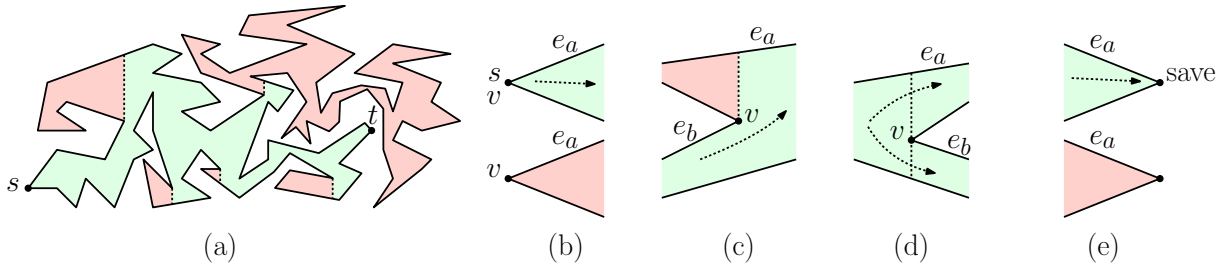


Figure 3: Monotone paths.

The algorithm involves a plane sweep through P. For each upper edge of each channel along the sweep-line status, we store a value reachable(e), which indicates whether the channel immediately below v is reachable. (In Fig. 3(a), the green shaded regions are monotonically reachable from s.). The processing is similar to that of the monotone decomposition algorithm

from Lecture 5. In the description below, we focus principally on how the reachable values 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: Let e_a be the upper edge incident to v. If $v = s$, set reachable(e_a) $\leftarrow$ true and otherwise set it to false (see Fig. 3(b)). Insert both edges into the sweep-line status.

Merge vertex v: Let e_a be the edge above v and let e_b be the lower edge incident to v. If either reachable(e_a) or reachable(e_b) then set reachable(e_a) $\leftarrow$ true, and otherwise it is false (see Fig. 3(c)). Delete both edges incident to v from the sweep-line status.

Split vertex v: Let e_a be the edge above v and let e_b be the lower edge incident to v. If reachable(e_a) then set both reachable(e_a) and reachable(e_b) to true, and otherwise set both to false (see Fig. 3(d)). Insert both edges incident to v in the sweep-line status.

End vertex v: Let e_a be the upper edge incident to v. If reachable(e_a), then save vertex v having the largest x-coordinate (see Fig. 3(e)).

Upper/Lower vertex v: For an upper vertex, we copy the reachable value from the left edge to the right edge. For a lower vertex, there is no need to alter the reachable value. We update the sweep-line status as in the triangulation algorithm.

Clearly, the rightmost monotonically reachable vertex from s will be an end vertex whose upper incident edge is labeled as reachable. Therefore, when the algorithm terminates, we return the saved vertex with the largest x-coordinate.

The correctness follows from the easily verified fact that the reachable values are properly updated for each event and the fact mentioned above that the rightmost monotonically reachable vertex from s must be an end vertex.

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.

(b) We run the algorithm from part (a) for both s_1 and s_2. We maintain two reachable values for each upper edge of each channel, one for s_1 and one for s_2. When we first discover a merge vertex v where the two incident channels are reachable, one from s_1 and one from s_2, we output this vertex (see Fig. 3(a)). By the correctness of part (a), there is a monotone path from s_1 to v and one from s_2 to v, and hence by reversing the second path, there is a unimodal path from s_1 to v and then back to s_2. If there no such vertex v found, we conclude that there is no monotone path from s_1 to s_2.

To establish correctness, we have argued above that if such a vertex v is found, there exists a unimodal path. Conversely, if there is a unimodal path from s_1 to s_2, consider the one with the leftmost turning-around point. Clearly, this point must be a merge vertex of P, and the two branches of the unimodal path approaches this vertex from opposite sides, one above v and one below v. The running time is the same as part (a), $O(n \log n)$.

Solution 4: The input to this problem is the outer polygon P and a collection of inner island polygons $I_1, \ldots, I_m$. Since all these polygons are x-monotone, we may represent each as two x-

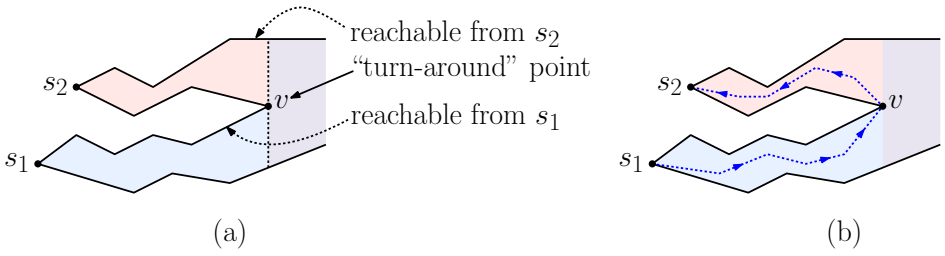


Figure 4: Unimodal path.

monotone chains of vertices: an upper chain and a lower chain. Let n denote the total number of vertices in all the polygons.

The algorithm is an application of left-to-right plane sweep. The sweep-line status consists of the line segments that intersect the sweep line from bottom to top. The events are the vertices of the polygons. Given the input, we can sort these vertices in $O(n \log n)$ time.

The basic processing is essentially the same as in the monotone subdivision algorithm for simple polygons:

Split vertex v: This vertex is the leftmost vertex of an island. We insert its two incident edges into the sweep-line status, and split the existing channel into two channels, one above the island and one below.

Merge vertex v: This vertex is the rightmost vertex of an island. We delete its two incident edges from the sweep-line status, and merge the two channels into one.

Upper/Lower vertex v: This vertex is in the interior of a polygon's upper or lower chain or on in the middle of an island. We swap out the left edge and insert the right edge.

By the nature of the input, there are no start or end vertices.

The main issue is efficiently maintaining the river's total vertical width throughout the sweep. We observe that the river's vertical width is a piecewise linear function of the sweep-line's x-coordinate, which we denote by $w(x)$. The breakpoints of this piecewise function coincide with the x-coordinates of the polygon vertices, that is, with the event points. A key observation is that, since the function is piecewise linear, its minimum must occur at an event point, that is, at a vertex event. It will be convenient to assume that all the polygons lie above the x-axis.

To maintain the width function, let $k < n$ denote the number of islands that intersect the current sweep line. The sweep-line status contains two types of edges: *upper edges*, where the river lies just above the edge, and *lower edges*, where the river is locally below the edge. Let $a_0(x)$ denote the line equation for the top edge of the river, and let $a_1(x)$ through $a_k(x)$ denote the line equations for the upper edges of the k islands (in any order). Define $b_0(x), \ldots b_k(x)$ analogously for the lower edges. By inclusion-exclusion, the total vertical width of the water portion equals the difference of the functions for the upper edges and the lower edges (see Fig. 5), that is:

$$w(x) \;=\; \sum_{i=0}^{k} b_i(x) - \sum_{i=0}^{k} a_i(x).$$

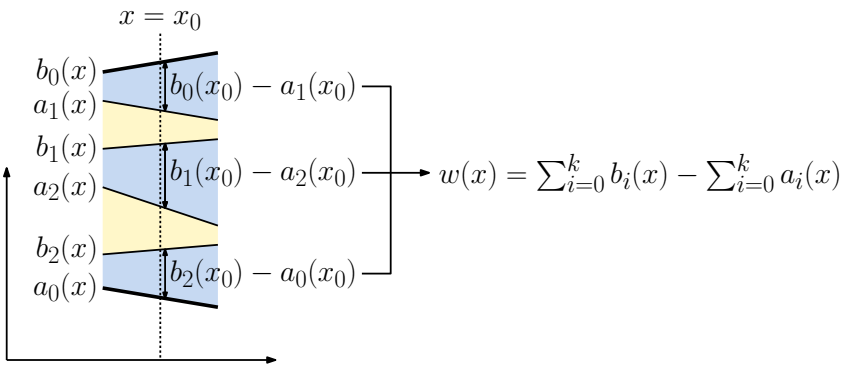


Figure 5: Minimum river width

Although it involves $2(k+1)$ terms, $w(x)$ can be expressed as a simple linear function of x, e.g., $cx+d$, for some constants c and d. We assert that whenever an event is processed, a constant number of terms in the above summation change. In particular, we remove the terms corresponding to the deleted elements and add the terms corresponding to the newly inserted elements. Thus, irrespective of the value of k, we can symbolically subtract the coordinates of the deleted functions from the formula for $w(x)$ and symbolically add the others.

For example, suppose that at $x = 2$ we just processed a vertex on the upper bank of the river, replacing the function $b_0(x) = 4x+5$ with the function $b_0(x) = 3x+7$. If the value of $w(x)$ was $cx+d$ just prior to this even, then afterwards it is $(cx+d)-(4x+5)+(3x+7) = (c-1)x+(d+2)$. Thus, setting $c \leftarrow c-1$ and $d \leftarrow d+2$ updates the width function in constant time. It is straightforward to update the function in constant time for all the events described above.

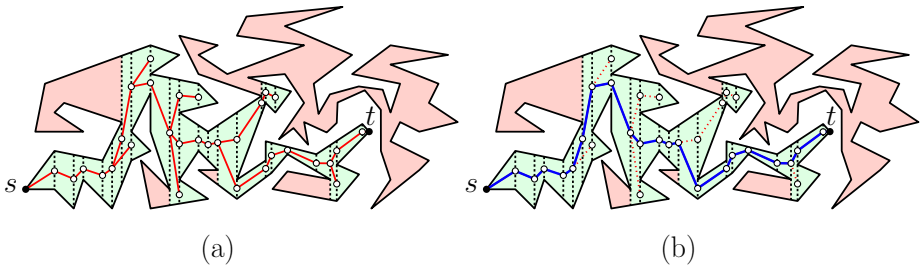
After updating the width function, we evaluate it at the current x value. We maintain the smallest such value, w^-, throughout the sweep. Finally, if $w^- < w_{\min}$ (the flood threshold), we report a flood event.

Because the width function can be updated in $O(1)$ time per event, the overall running time is dominated by the ordered dictionary operations, giving $O(n \log n)$ time.

Solution to Challenge Problem 1: We construct a graph with vertices s and t, and for each vertical segment along the sweep line reachable from s, we add a vertex in its interior. We then construct a directed graph, connecting each vertex of this graph to its immediate neighbor to the right (see Fig. 6(a)). By the notion of reachability, there exists a path from s to t in this graph. (In fact, there will only be one, because P has no holes.) We then compute such a path, say, by DFS or BFS in the graph, and connect s to t using this path (see Fig. 6(b)).

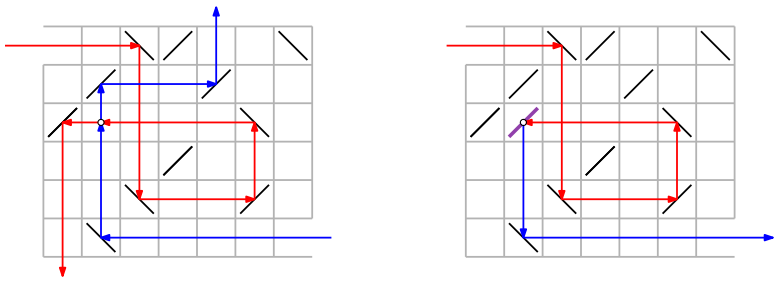
Solution to Challenge Problem 2: This problem has a very cute solution. Starting in the upper left corner, trace a path through the box of mirrors until the beam leaves the box. Do the same starting at the lower-right corner. If these two beams never intersect, no single mirror can create a successful path. Otherwise, if they intersect, find an intersection point and insert a mirror there at the appropriate angle. If the beams enter from the top and left or bottom and right, the mirror angle is $45°$; otherwise, it is $-45°$ (see Fig. 7).

The final beam consists of traversing the upper-left beam in the forward direction and the lower-right beam in the reverse direction. The running time of the algorithm is dominated by the time to trace the two beams, which is $O(n \log n)$, since the beam can hit every mirror at most



(a) (b)

Figure 6: Monotone Path problem.



(a) (b)

Figure 7: Laser beam solution.

twice, and each ray-shooting query can be done in $O(\log n)$ time, by the data structure that was provided. The line-segment intersection algorithm presented in class can compute the intersection of the two paths. The algorithm can stop as soon as it detects the first intersection, so it runs in $O(n \log n)$ time.

To show correctness, observe that if a mirror can connect the entry and exit points with a single path, then the path can be split at the mirror into two subpaths. If we reverse the portion of the path that comes after the mirror, this is the same as the path entering the box from the lower-right corner. Therefore, the two paths must intersect.

Conversely, if the two paths intersect, first note that the only possible intersection is one where one path is vertical and the other is horizontal. (If they were traveling in the same direction, they would need to be oppositely directed, which would imply that the path entering at the upper-left would eventually exit at the lower-right, which we assumed does not happen.) We can insert a mirror at this point that (subject to reversing the path entering from the lower-right corner) connects the two corners of the box.