Solutions to Homework 5

Solution 1: We first transform the problem to a simpler form. Observe that two unit disks intersect each other if and only if the distance between their centers is 2 or smaller. Therefore, we can reduce our problem to that of moving a point among a collection of circular disks, each of radius 2. For the remainder, we will consider the problem in this form.

Our approach is based on Voronoi diagrams. Let $P = \{p_1, \ldots, p_n\}$ denote the center points of the obstacle disks. We first compute the Voronoi diagram, $\text{Vor}(P)$. Observe that the Voronoi edges are the points that are locally farthest from any point of P. Thus, if an escape path exists, it has the form of first backing away from the closest center, until hitting a Voronoi edge, and then traveling along Voronoi edges until (if possible) reaching an unbounded edge.

(a) We compute the Voronoi of P, $\text{Vor}(P)$ in $O(n \log n)$ time, and at the same time we build a point location data structure for the diagram (see Fig. 1(a)). We visit each edge of the diagram and "erase" the portion that lies within distance 2 of the closest sites (see Fig. 1(b)). This removes at most a single subsegment from each edge, and it easily be done in $O(1)$ time per edge, for a total of $O(n)$ time. This yields a planar graph of $O(n)$ size, which may have multiple connected components. Clearly, any motion along the remaining edge fragments will be collision free. To determine which edge fragments can reach infinity, we compute the connected components of the resulting graph, and label a component as "escapable" if it contains at least one unbounded edge. All other components are labeled as "blocked". (All the edge fragments in our figure are escapable.) Because the graph of edge fragments has size $O(n)$, this can be done in $O(n)$ time, using any standard graph traversal algorithm, such as DFS.

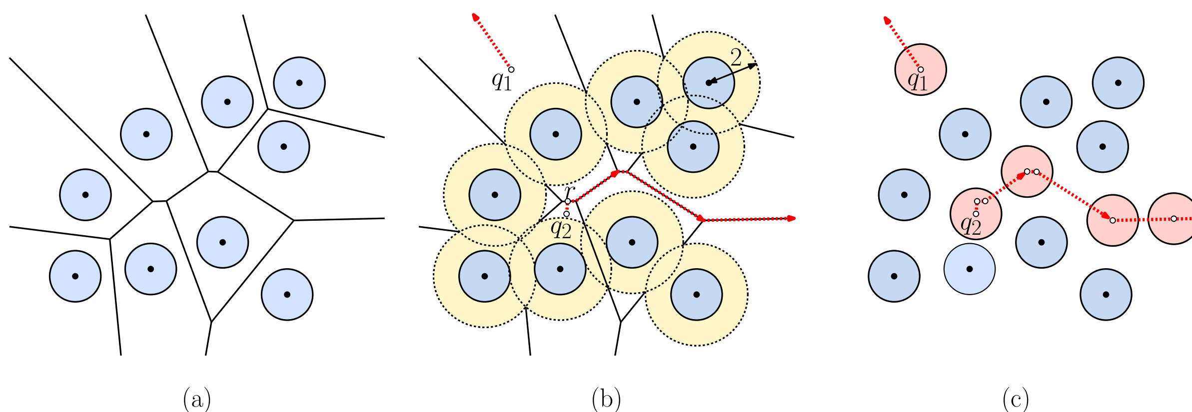

Figure 1: Escape problem data structure.

(b) Given this structure, here is how we answer a query for a point q:

- Apply our point-location data structure to locate the Voronoi cell $V(p_i)$ that contains q in $O(\log n)$ time. (By general position, we may assume that p_i is unique.)

- If $\|q - p_i\| \leq 2$, then the initial disk overlaps the disk at p_i, implying that the initial configuration is invalid.
- Otherwise, shoot a ray from p_i through q until it hits the boundary of $V(p_i)$. (Since $V(p_i)$ is a convex polygon, we can determine which edge is hit in $O(\log n)$ time by a binary search among the vertices of $V(p_i)$ based on the angle of the vector $q - p_i$.) If the ray goes all the way to ∞ without hitting the boundary, then clearly q can escape by simply traveling on this ray (see the query point q_1 in Fig. 1(b)). Otherwise, if hits the Voronoi cell boundary at some point r, if r lies on an edge of an "escapable" component, we report that q can escape (see the query point q_2 in Fig. 1(b)). Otherwise, we report that q cannot escape. The query time is dominated by the $O(\log n)$ time for point location.

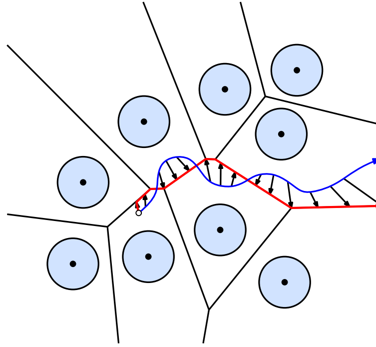

Figure 2: Escape problem correctness.

To establish correctness, we will show that the algorithm reports that q can escape if and only if an escape path exists. If we report that q can escape, it is clear that the path described above is a collision-free path (since moving away from the closest site p_i is always safe, and the remaining movement travels along fragments of Voronoi edges that are all at distance at least 2 from their closest site). Conversely, suppose that there is an escape path for q. We can map this path to one of the form described in our algorithm by mapping each point along the path directly away from its closest site until hitting the boundary of the Voronoi cell. (In Fig. 2 the original escape path shown in blue is mapped point-by-point onto the Voronoi diagram as shown in red.) The transformed path is valid, because we have moved each point further away from its closest site. This will exactly match the path that we have generated, and so our algorithm will report that q is escapable.

Solution 2: We reduce the problem to a problem of computing a path in an appropriate graph, particularly the minimum spanning tree of a graph based on the lily pad centers.

We will define two graphs denoted C and G. One to be used to prove correctness, and the other to be used in the algorithm. Both graphs have the same vertex set, which consists of the set P of lily pad center points together with two vertices v^+ and v^-, which represent the stream's upper and lower banks (at $y = y^+$ and $y = y^-$, respectively). First, in both graphs we add $2n$ edges consisting of the edges (p, v^+) with weight $y^+ - p_y$ and edge (p, v^-) with weight $p_y - y^-$ for all $p \in P$. These represent the time when the lily pad at p hits the upper or lower bank, respectively. For the graph C, we include *all* the $\binom{n}{2}$ edges joining the points of P (see Fig. 3(a)), and for the graph G, we include only the edges of the Delaunay triangulation, of which there are $O(n)$ (see Fig. 3(b). Each of these edges is given a weight of half the distance between its endpoints.

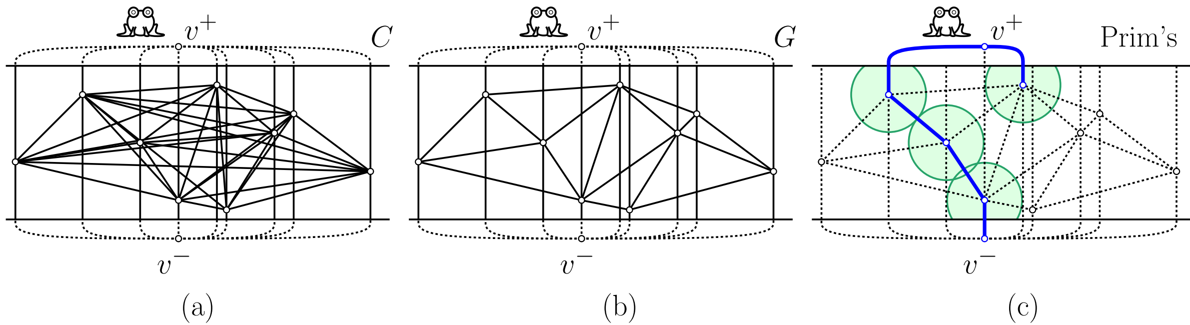

Figure 3: Frog crossing.

Given any $t \geq 0$, define the *t-restricted subgraph of C*, denoted $C(t)$, to consist of all edges of C weight t or smaller. Define $G(t)$ analogously. Let t^* be the smallest value such that v^+ and v^- are connected in $C(t^*)$. It is easy to see that t^* is the first time at which a crossing path exists for the frog (see Fig. 3(c)). The frog jumps from the upper bank (v^+) to a lily pad within distance t^*, then it traverses all the pads, each pair of which lie within distance $2t^*$ of each other, and finally it jumps to the lower bank (v^-) from a lily pad within distance t^*.

How can we find t^*? We'll describe two algorithms, which actually do the same thing. The first will be easier to prove correct, and the second will be more efficient. For the first algorithm, we will run Prim's minimum spanning tree algorithm on the graph C starting at v^+, and we stop the algorithm as soon as it reaches v^-. Prim's algorithm constructs the minimum spanning tree by repeatedly adding the lowest weight edge can goes from the current spanning subtree to a new vertex. When it reaches $v-$, it has used the lowest possible edge weights to get there. Let t^* denote the maximum weight edge generated by Prim's algorithm. Every other edge is the tree produced by Prim's algorithm is of lower weight, implying that t^* is the smallest value such that v^+ and v^- are connected in $C(t^*)$.

But this algorithm is not efficient since C has $\binom{n}{2}$ edges. To address this, we will instead run the algorithm in G. This achieves the same result, since as shown in the class, the minimum spanning tree of C is a subgraph of the Delaunay triangulation, and hence is a subgraph of G. When we run Prim's algorithm on G, it will generate all the same edges that the algorithm does when it is run on C, the only difference being that the algorithm runs in $O(n \log n)$ time, because G has only linearly many edges.

Solution 3: Consider a set of sites $\{a, b, c, d\}$ that are nearly cocircular, as shown in the figure below. Points b and d are placed symmetrically about $\overline{ac}$, but much closer to a than to c. By perturbing the point c slightly inside or outside the circumcircle $\triangle abd$, we produce Delaunay triangulations T_1 or T_2, respectively. If the perturbation is sufficiently small, the relative lengths of edges and sizes of (unequal) angles will be unchanged.

(a) Clearly, $\|a - c\| > \|b - d\|$, and therefore Delaunay triangulation T_1 does not minimize the sum of edge lengths.

(b) Clearly, $\angle bad$ is indeed "bad" in the sense that it is larger than all the other angles of the quadrilateral. Hence, the Delaunay triangulation T_2 (which includes this angle) does not minimize the maximum angle.

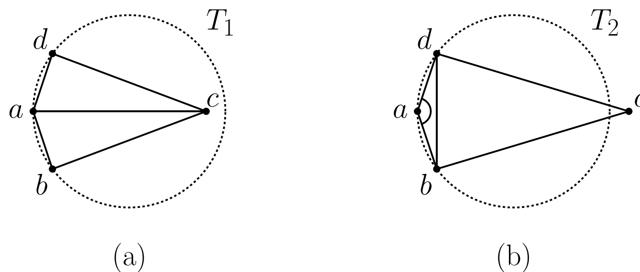

Figure 4: Delaunay counterexamples.

Solution 4: In this problem we will use two related facts from basic Euclidean geometry. First, given two adjacent triangles, $\triangle pqr$ and $\triangle pqs$, the sum of angles $\angle prq + \angle psq$ is greater/less/equal to π if and only if s lies inside/outside/on (respectively) the circumcircle of $\triangle pqr$ (see Fig. 5(a)). Second, a triangle $\triangle pqr$ is acute if and only the center of its circumcircle lies within the triangle's interior (see Fig. 5(b)).

(a) We will prove the contrapositive, namely, if a triangulation is not Delaunay, then it is not acute. Let T be any triangulation that is not Delaunay. By Delaunay's Theorem, any triangulation that is locally Delaunay is also globally Delaunay, and therefore T is not locally Delaunay. This means that there exist two adjacent triangles, $\triangle pqr$ and $\triangle pqs$ such that s lies within the circumcircle of $\triangle pqr$. Therefore, by the first geometric fact, we have $\angle prq + \angle psq > \pi$, which implies that at least one of the two angles is not acute (see Fig. 5(a)). Therefore T is not acute, as desired.

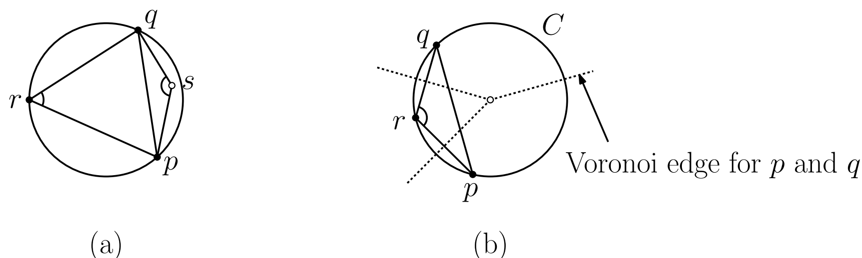

Figure 5: Solution to Problem 2.

(b) We will show the contrapositive. Suppose that P's Delaunay triangulation is not acute. Consider a triangle $\triangle pqr$ that is not acute at some vertex r, and consider its circumcircle C (see Fig. 5(b)). From the second observation above, C's center (which is the Voronoi vertex of p, q, and r) lies outside the interior of $\triangle pqr$. Therefore the edge $\overline{pq}$ does not intersect the Voronoi edge for p and q, implying that the triangulation is not medial.

Solution to the Challenge Problem: This can be done by *inversion counting*. We assume that the points are in general position, so there are no duplicate x- or y-coordinates. Given a list of numbers, $\langle z_1, \ldots, z_n \rangle$, an *inversion* is any pair i, j where $i < j$ by $z_i > z_j$. If we sort the points by x-coordinates and label each point with its index in this sorted list, then the number of

intersections is easily seen to be equal to the number of inversions in the list of indices along the y-axis.

Inversion-counting can be done by a simple generalization of the MergeSort algorithm. Recall that MergeSort is a classical example of divide-and-conquer. The sequence B of length n is split (e.g., down the middle) into a left and right subsequence, denoted B_1 and B_2, each of size roughly $n/2$. These two subsequences are sorted recursively, and then the resulting sorted sequences are then merged to form the final sorted sequence.

To generalize this to inversion counting, in addition to returning the sorted subsequences, the recursive calls return the counts I_1 and I_2 of the inversions *within* each of the subsequences. In the merging process we count the inversions I that occur *between* the two subsequences. That is, for each element of B_1, we compute the number of smaller elements in B_2, and add these to I. In the end, we return the total number of inversions, $I_1 + I_2 + I$.

The algorithm is presented in the code block below. To merge the subsequences, we maintain two indices i and j, which indicate the current elements of the respective subsequences B_1 and B_2. We repeatedly copy the smaller of $B_1[i]$ and $B_2[j]$ to the merged sequence M. Because both subsequences are sorted, when we copy $B_1[i]$ to M, $B_1[i]$ is inverted with respect to the elements $B_2[1 \ldots j-1]$, whose values are smaller than it. Therefore, we add $j-1$ to the count I of inversions.

The main loop stops either when i or j exceeds the number of elements in its subsequence. When we exit, one of the two subsequences is exhausted. We append the remaining elements of the other subsequence to M. In particular, if $i \leq |B_1|$, we append the remaining $|B_1| - i + 1$ elements of B_1 to M. Since these elements are all larger than any element of B_2, we add $(|B_1| - i + 1)|B_2|$ to the inversion counter. (When copying the remaining elements from B_2, there is no need to modify the inversion counter.) See the code block below for the complete code.

Inversion Counting

InvCount(B) [**Input:** a sequence B; **Output:** sorted sequence M and inversion count I.]

- If $|B| \leq 1$ then return an inversion count of zero;
- Split B into disjoint subsets B_1 and B_2, each of size at most $\lceil n/2 \rceil$, where $n = |B|$;
- $(B_1, I_1) \leftarrow \text{InvCount}(B_1)$;
  $(B_2, I_2) \leftarrow \text{InvCount}(B_2)$;
- Let $i \leftarrow j \leftarrow 1$; $I \leftarrow 0$; $M \leftarrow \emptyset$;
- While ($i \leq |B_1|$ and $j \leq |B_2|$)
  - if ($B_1[i] \leq B_2[j]$) append $B_1[i{+}{+}]$ to M and $I \leftarrow I + (j-1)$;
  - else append $B_2[j{+}{+}]$ to M;

  On exiting the loop, either $i > |B_1|$ or $j > |B_2|$.
- If $i \leq |B_1|$, append $B_1[i \ldots]$ to M and $I \leftarrow I + (|B_1| - i + 1)|B_2|$;
- Else (we have $j \leq |B_2|$), append $B_2[j \ldots]$ to M;
- return $(M, I_1 + I_2 + I)$;

The running time exactly matches that of MergeSort. It obeys the well known recurrence $T(n) = 2T(n/2) + n$, which solves to $O(n \log n)$.