Solutions to Homework Exam 1 2018

This homework exam has 6 questions for a total of 100 points. You can earn an additional 10 points by a careful preparation of your hand-in: using a good layout, good spelling, good figures, no sloppy notation, no statements like “The algorithm runs in \( n \log n \)” (forgetting the \( O(\ldots) \) and forgetting to say that it concerns time), etc. Use lemmas, theorems, and figures where appropriate. Your final grade will be the number of points divided by 10 (with a maximum of a 10). Note that solving only a subset of the problems is sufficient to get a 10.

**Question 1**

Let \( S \) be a planar subdivision with \( n \) vertices, and let \( e \) be a half-edge of \( S \) that is incident to the outer face.

(a) (10 points)

Give pseudo-code for an algorithm that, given a pointer to \( e \), computes all vertices of \( S \) that have “hop-distance” at most one to the outer face. That is, vertices that incident to the outer face, or are adjacent to a vertex on the outer face. Your algorithm should use the ‘Twin’, ‘NextEdge’, ‘PrevEdge’, etc. fields to navigate (i.e. you cannot assume you can directly access a list of vertices).

Describe in a few sentences the main idea of your algorithm and give its running time.

**Hint:** You can use the data fields of vertices, half-edges, and faces to store marks/flags.

### Solution

Observe that \( S \) may be disconnected, and hence the outer face may contain multiple “holes” (components) each of which containing vertices that should be reported. The general strategy is therefore going to be: (1) find all components, (2) for every component find and report all vertices incident to the outer face, (3) for each of these vertices report all neighbors. To make sure we report every vertex at most once we mark them as soon as we report them for the first time, and report only unmarked vertices. The following procedures realize this:

**Algorithm** \( \text{Report}(e) \)

**Input.** (A pointer to) A half edge \( e \) incident to the outer face.

**Output.** All vertices at distance at most one from the outer face.

1. \( cs \leftarrow \text{InnerComponents}(\text{IncidentFace}(e)) \)
2. \( \text{for each } h \in cs \text{ do} \)
3. \( \text{ReportFromComponent}(h) \)

**Algorithm** \( \text{ReportFromComponent}(e) \)

**Input.** (A pointer to) A half edge \( e \) incident to the outer face.

**Output.** All vertices in the same component as \( e \) at distance at most one from the outer face.

1. \( vs \leftarrow \text{BoundaryVertices}(e) \)
2. \( \text{for each } v \in vs \text{ do} \)
3. \( \text{if } v \text{ is unmarked then output } v \text{ and mark it.} \)
4. \( \text{ReportNeighbors}(v) \)
**Algorithm** \textsc{ReportNeighbors}(v)

\textit{Input.} (A pointer to) A vertex \(v\).
\textit{Output.} Reports all neighbors of \(v\) that are not marked yet.
1. \(e_0 \leftarrow \text{IncidentEdge}(v) ; e \leftarrow e_0\)
2. \textbf{repeat}
3. \(h \leftarrow \text{Twin}(e)\)
4. \(u \leftarrow \text{Origin}(h)\)
5. \textbf{if} \(u\) is unmarked \textbf{then} output \(u\) and mark it.
6. \(e \leftarrow \text{Next}(h)\)
7. \textbf{until} \(e = e_0\)

**Algorithm** \textsc{BoundaryVertices}(e)

\textit{Input.} (A pointer to) A half edge \(e\).
\textit{Output.} A list of vertices \(vs\) on the face incident to \(e\).
1. \(h \leftarrow e ; vs \leftarrow []\)
2. \textbf{repeat}
3. \(v \leftarrow \text{Origin}(h)\)
4. Add \(v\) to \(vs\)
5. \(h \leftarrow \text{Next}(h)\)
6. \textbf{until} \(h = e\)
7. \textbf{return} \(vs\)

It is easy to argue that the above algorithm runs in time proportional to the total size of the planar subdivision (i.e. we consider every half edge and/or incidentFace field at most a constant number of times). Hence, this would result in an \(O(n)\) time algorithm.

A more careful argument can actually show that the running time is only \(O(k)\), where \(k\) is the number of vertices reported in the output. To see this consider marking all edges traversed, and delete all unmarked vertices and edges. The result is a planar subdivision with \(k\) vertices. It follows that it has complexity \(O(k)\). Since we spent only constant time per edge and vertex of this subdivision the running time is \(O(k)\).

(b) (10 points)
Suppose that you do not get a pointer to \(e\), but instead get a pointer to some arbitrary half-edge \(f\) of \(S\). Describe an approach to find a half-edge incident to the outer face, and give its running time. (You are not required to give the pseudo-code for this approach.)

\textbf{Solution}

Observe that the vertex \(v\) with the smallest \(x\)-coordinate (i.e. the leftmost vertex) is incident to the outer face. Moreover, it’s steepest outgoing half-edge \(h\) has the outer face to its left. We can find \(v\) using something like breath-first search. To find \(h\) we then simply scan through the outgoing half-edges of \(v\) and report the half-edge with maximum slope. Finding \(v\) requires \(O(n)\) time. Finding \(h\) then requires time linear in the number of adjacent vertices, which is also at most \(O(n)\).

**Question 2** (10 points)
Let \(P\) be a polygon with \(n\) vertices and \(h\) holes. Prove that any triangulation of \(P\) consists \(n + 2h - 2\) triangles.

\textbf{Solution}

\textbf{Theorem 1.} Let \(P\) be a polygon with \(n\) vertices and \(h\) holes. Any triangulation of \(P\) consists of \(n + 2h - 2\) triangles.
Proof. Let \( T \) be a triangulation of \( P \), and observe that \( T \) is a connected planar subdivision with \( n \) vertices. Thus, we can use Euler’s formula, which states that \( n - E + F = 2 \), where \( E \) denotes the number of edges, and \( F \) denotes the total number of faces. Let \( k \) denote the number of triangles. We have \( F = k + h + 1 \) (the +1 because of the outer face). For the number of edges we have \( 2E = 3k + n \), since every triangle contributes three half edges and every “hole” (including the outer face) contributes one half edge. Combining these two results we get:

\[
\begin{align*}
2n - 2E + 2(k + h + 1) &= 4 \\
2n - 3k - n + 2k + 2h + 2 &= 4 \\
-k &= -n - 2h + 2 \\
k &= n + 2h - 2.
\end{align*}
\]

Hence the number of triangles in \( T \) is \( n + 2h - 2 \) as claimed. \( \square \)

Alternatively, you can prove this using induction on the number of holes \( h \). The base case \( h = 0 \) is proven in Theorem 3.1 in the book. For \( h > 1 \) the idea is to find a diagonal (line-segment) connecting a hole to the outer boundary. Consider cutting open the polygon along this line segment. The resulting polygon has one hole less (so we can use the induction hypothesis), but it does have two more vertices. The result then directly follows.

**Question 3**

Let \( S \) be a set of \( n \) disjoint line segments in the plane, and let \( p \in \mathbb{R}^2 \) be a point that does not lie on any of the line segments. You may assume that the segments in \( S \) are open, and that the set that contains \( p \) and all endpoints from \( S \) contains exactly \( 2n + 1 \) points and that no three such points are colinear.

(a) (10 points)

Develop an \( O(n \log n) \) time algorithm to compute the length of a longest (open) segment \( s \) that contains \( p \) but does not intersect any segment in \( S \). If segment \( s \) does not exist your algorithm should return \( \infty \). Prove that your algorithm is correct and achieves the stated running time.

**Solution**

The main idea is to use a sweepline algorithm in which the sweep line rotates around \( p \).

To simplify the presentation, assume that \( p \) and all segments lie in some big rectangle (constructed from four segments). Now a longest line segment is guaranteed to exist. If such a segment has an endpoint on the rectangle we report \( \infty \) instead.

**Geometric Properties.** Consider the halfline (ray) \( \rho_0 \) in direction \( \theta \in [0, 2\pi) \) starting in \( p \), and let \( s_0 \in S \) be the first segment hit by \( \rho_0 \).

**Observation 2.** Any segment with orientation \( \theta \) that contain \( p \) but does not intersect \( S \) cannot extend further than \( s_0 \).

**Observation 3.** For any orientation \( \theta \) there is a single longest line segment with orientation \( \theta \) that contains \( p \) but does not intersect \( S \); its endpoints lie on the segments \( s_0 \) and \( s(\theta + \pi) \mod 2\pi \).

We now proceed in two steps; first we actually compute \( s_0 \) for all \( \theta \in [0, 2\pi) \), by rotating the ray \( \rho_0 \) around \( p \). Then we rotate around \( p \) once more to compute the length of a segment with orientation \( \theta \) and report the maximum length.

**Computing \( s_0 \) for all orientations \( \theta \).** We use a rotational sweep. The result is represented by a simple polygon \( P \) (the visibility polygon of \( p \) with respect to \( S \)) in which every edge corresponds with a range \([\theta_1, \theta_2]\) together with a segment \( s \) (such that for any \( \theta \in [\theta_1, \theta_2] \) we have \( s_0 = s \)).

Our status structure will be a binary search tree that stores the segments currently intersected by the sweep-ray \( \rho_0 \) in order along the ray. (We could also use a priority queue as status structure). We start with a horizontal rightward ray, and explicitly test for all segments if
they intersect the ray and insert them appropriately. The set of segments intersected by \( \rho \theta \), and the order in which they do so, changes only when \( \rho \theta \) sweeps over an endpoint of one of the segments in \( S \). So, we can compute all events by radially sorting the endpoints of the segments in \( S \) around \( p \). To handle an event we simply insert/remove the appropriate segment from the status structure. If the closest segment in the status structure (i.e. the segment \( s_p \) in the leftmost leaf) changes we report a vertex in \( P \).

Computing a longest segment. We now rotate once more around \( p \) to compute the length of a longest segment of orientation \( \theta \) (Observation 3). We maintain the two edges in \( P \) that are subsegments of \( s_\theta \) and \( t_\theta = s_\theta + \pi \mod 2\pi \). This can easily be done using a simultaneous scan through \( P \). For fixed pair of such edges \( s, t \) we can easily compute the orientation \( \theta^* \) for which \( s_{\theta^*} = s, t_{\theta^*} = t \) and the length of the segment through \( p \) is maximal (as well as report this length).

Analysis. The first sweep takes \( O(n \log n) \) time: computing all events takes \( O(n \log n) \) time (sorting), initializing the status structure can then be done in \( O(n \log n) \) time as well. At every event we insert or delete a single segment in/from the binary search tree. This takes \( O(\log n) \) time. At every event we report at most one new vertex/edge of \( P \), thus the resulting polygon \( P \) has complexity \( O(n) \).

The second sweep just consists of two simultaneous scans through the vertices of the simple polygon. For a given pair of segments \( s, t \) we have a constant size subproblem (defined by \( s, t, \) and \( p \)) so we can find the orientation \( \theta^* \) that maximizes the length in constant time. We thus conclude:

**Theorem 4.** Given a set \( S \) of \( n \) disjoint line segments in the plane, and a point \( p \). We can compute the length of the longest segment containing \( p \) (if it exists) in \( O(n \log n) \) time.

(b) (5 points)
Is your algorithm still correct if the segments in \( S \) may intersect? If so, argue why, if not, give an example why not, and describe how to fix it. You do not have to argue about the running time of your algorithm in this scenario.

**Solution**
If the segments may intersect the order in which the segments intersect the sweep-ray \( \rho \theta \) may change at intersection points. In particular, the two nearest segments (in a given direction) may intersect. The longest segment may be defined at such an intersection point. We can adapt the rotational sweep (the first step of the above algorithm) to detect and handle such events.

**Extra.** It is still possible to compute the visibility polygon, and therefor the longest segment in \( O(n \log n) \) time. There are two complications. The first one is that we want our first sweep to be output sensitive in the size of \( P \) (i.e. we don’t want to consider intersections that do not contribute vertices of \( p \)). The second one is that the complexity of the polygon \( P \) may be \( \Theta(n \alpha(n)) \) (where \( \alpha(n) \) is the extreme slow growing inverse Ackermann function). This is something we have not seen during the course though, and hence you did not have to argue about the running time.

**Question 4 (20 points)**
Given a set \( R \) of \( n \) “red” points and a set \( B \) of \( n \) “blue” points in \( \mathbb{R}^2 \). Develop an algorithm that can test if there exists a line \( \ell \) that separates \( R \) from \( B \), that is, such that all points in \( R \) lie right of \( \ell \) and all points in \( B \) lie left of \( \ell \). Prove that your algorithm is correct and analyze its running time. You may assume that any line contains at most two points of \( R \cup B \) (i.e. there are no three colinear points).

Note: the number of points rewarded for this question will depend on the running time of your algorithm.
Solution

**Claim 5.** Let $R$ and $B$ be two sets of $n$ points in $\mathbb{R}^2$. We can decide if there is a line that separates $R$ and $B$ in $O(n)$ expected time.

The main idea is to write the above problem as a two-dimensional linear program.

We first test if there is a vertical line that separates $R$ from $B$. We can easily test this in $O(n)$ time by computing the leftmost and rightmost red and blue points.

If there is no vertical separating line, the separating line (if it exists) can be described by the linear function $y = cx + d$ (for some $c, d \in \mathbb{R}$). All red points $r \in R$ must be right of the line, and thus they generate a constraint $r_y \leq cr_x + d$. Similarly, all blue points $b \in B$ generate a constraint $b_y \geq cb_x + d$. Observe that these constrains correspond to half-planes, and that there exist parameters $c$ and $d$ so that all constrains are satisfied if (and only if) the point $(c, d)$ lies in the intersection of all constraint half-planes. Testing if such a point exists is exactly linear programming. Since we have a linear number of constraints (one per point) this requires $O(n)$ expected time.

Alternatively, the problem can be solved in worst case $O(n \log n)$ time by computing the two convex hulls and testing if they can be separated. This requires computing external tangents (which can be done in linear time).

**Extra.** Since low-dimensional linear programming can even be solved in $O(n)$ time in the worst case the same applies for the separation problem.

**Question 5** (10 points)

Let $P$ be a set of $n$ points in $\mathbb{R}^2$, and let $R$ be the shortest (in terms of Euclidean length) closed curve such that all points of $P$ lie inside (or on the boundary of) the area enclosed by $R$. Prove that $R$ is the convex hull $CH(P)$ of $P$.

**Solution**

**Observation 6.** Let $p$ and $q$ be two points in $\mathbb{R}^2$. The linesegment $pq$ is the shortest curve connecting $p$ to $q$, and any curve $H[p,q]$ connecting $p$ to $q$ that contains a point not on $pq$ is strictly longer than $pq$.

I think it is fine to state Observation 6 as is and use it without explicit proof. The argument to prove it would use something like:

Rotate the plane such that $pq$ is horizontal and $p$ is left of $q$. This maintains distances. The length of $pq$ is now simply $q_x - p_x$. Consider any point $r$ on $pq$. If $r$ lies left of $p$ or right of $q$ then the $x$-component of the distance already exceeds $q_x - p_x$ (and the $y$-component is non-negative). If $r_x \in [p_x, q_x]$ but $r_y \neq q_y = q_y$ then the $y$-component is strictly larger than zero (and the $x$-component is still at least $q_x - p_x$). Hence, if $H[p,q]$ contains $r$ its length is strictly larger than $pq$.

(To make this fully formal we need to be a bit more precise about how the length of $H[p,q]$ is defined exactly)

**Theorem 7.** The curve $R$ is the convex hull $CH(P)$ of $P$.

**Proof.** Assume, by contradiction that $R \neq CH(P)$. This means that $R$ contains a subcurve that lies strictly outside $CH(P)$ or strictly inside $CH(P)$ (or both). We argue that both cases lead to a contradiction, thus proving the theorem.

outside $CH(P)$. This means there is some edge $e$ of $CH(P)$ such that $R$ intersects the half-plane defined by $e$ that does not contain any points from $P$. Let $R_e$ be a maximal such subcurve in this halfplane (so the endpoints of $R_e$ lie on the supporting line of $e$), and let $R[p,q]$ be the shortest such maximal subcurve over all edges $e$. By Observation 6 we can now shortcut $R[p,q]$ by the linesegment $pq$. Observe that the resulting curve still encloses
all points (since there were none in the half-plane defined by \(e\)) and the curve does not intersect itself (since we picked the shortest maximal-subcurve). Since the resulting curve is shorter than \(R\) we get a contradiction.

\(R'\) inside \(CH(P)\). If \(R'\) is completely inside \(CH(P)\) it cannot contain all points of \(P\): pick a vertex \(v\) of \(CH(P)\) and consider the outward ray into an empty half-plane bounded by a line through \(v\). This ray does not intersect \(R'\) and thus \(v\) lies outside of the area bounded by \(R'\). Contradiction.

If \(R'\) intersects an edge of \(CH(P)\) we use an argument similar to the \(R'\) outside \(CH(P)\) case: let \(R[p,q]\) be a maximal subcurve that lies strictly inside \(CH(P)\) and for which the "pocket" defined by \(R[p,q]\) and \(pq\) is empty of points. Since \(R\) encloses all points in \(P\) it follows that \(p\) and \(q\) lie on a single edge \(e\) of \(CH(P)\). Observation 6 then again tells us that we can shortcut \(R[p,q]\) yielding a contradiction.

Question 6

Let \(P\) be a set of \(n\) points in \(\mathbb{R}^2\), let \(D(c)\) be a unit disk, that is, a disk of radius one and center \(c\), and let \(P_c = P \cap D(c)\) be the subset of \(P\) that lies in a unit disk centered at \(c\).

(a) (10 points)
Prove that there are at most \(O(n^2)\) different sets \(P_c\) over all points \(c \in \mathbb{R}^2\).

Solution

**Lemma 8.** There are at most \(O(n^2)\) different sets \(P_c\) over all points \(c \in \mathbb{R}^2\)

**Proof.** Fix a point \(p \in P\) and observe that all unit disks that contain \(p\) (i.e. for which \(p \in P_c\)) must have their center point \(c\) inside the unit disk \(D(p)\). Consider the overlay—a planar subdivision—\(S\) of all these disks \(D(p)\), for \(p \in P\). By the above observation we have that for any face \(F\) of \(S\) uniquely corresponds to a set of covered points \(P_F\). Stated differently: for any two centers \(c, c' \in F\) the disks \(D(c)\) and \(D(c')\) contain the same subset \(P_F = P_c = P_{c'}\) of points from \(P\). Hence, the number of different sets \(P_c\), over all points \(c \in \mathbb{R}^2\), is at most the number of faces of \(S\).

So all that remains is to show that \(S\) has \(O(n^2)\) faces. We argue that it has \(O(n^2)\) vertices. Since \(S\) is a planar subdivision the lemma follows.

Any vertex of \(S\) is an intersection point of two unit circles; the boundaries of \(D(p)\) and \(D(q)\), for some \(p, q \in P\). Two (unit) circles can intersect at most twice, and thus any pair \(p, q\) can contribute at most two vertices of \(S\). Hence, \(S\) has \(O(n^2)\) vertices, and thus faces. This completes the proof.

(b) (10 points)
Give a construction that shows that the above bound is tight in the worst case. In other words, show that there can sometimes be \(\Omega(n^2)\) different sets \(P_c\).
Let \( k = n/2 \), and let \( P = \{p_1, \ldots, p_k, q_1, \ldots, q_k\} \). We place the points \( p_1, \ldots, p_k \) in that order on a vertical line segment of length one. Similarly, we place the points \( q_1, \ldots, q_k \) on a horizontal line segment of length one. The distance between \( p_1 \) and \( q_k \) is at most one. See the above figure. Consider the unit circles centered at the points in \( P \). The overlay of these circles essentially contains a \( k \times k \) grid. Let \( F_{ij} \) be the grid cell/face that has the topmost intersection point of the disks centered at \( p_i \) and \( q_j \) as topmost point. For a unit disk with center \( c \) in a face \( F_{ij} \) we get \( P_c = \{p_1, \ldots, p_i, q_1, \ldots, q_j\} \). Hence, we get a different set of reachable cells in the grid. The number of grid cells is \( \Omega(k^2) = \Omega(n^2) \) as desired.

(c) (5 points)
Let \( k \) be a natural number. Sketch an \( O(n^2 \log n) \) time algorithm that can compute the number of subsets of \( P \) of size \( k \) that can be covered exactly (i.e., the disk contains no additional points) by a unit disk. One or two paragraphs of description is sufficient; you do not have to prove correctness or give the full analysis.

Solution
The main idea was that by the argument used in question a we are interested in the number of faces in \( S \) for which \( P_{F} \) has size \( k \), i.e. \( |P_{F}| = k \). To compute that number we can then explicitly construct the planar subdivision \( S \), label every face \( F \) with the number \( |P_{F}| \), and then report the number of faces for which \( |P_{F}| = k \).

We can compute \( S \) in \( O(n^2 \log n) \) time using a sweep line algorithm. The status structure and event queue are virtually the same as in our sweepline algorithm to compute intersection points between a set of line segments. We can then compute all face labels in \( O(n^2) \) time by traversing the subdivision and maintaining the set of intersected points; if we cross an edge and move into a neighboring face we either lose or gain exactly one point. Reporting the number of faces for which the label is \( k \) requires \( O(n^2) \) time as well.
The complication. The statement “we are interested in the number of faces for which $P_F$ has size $k$” is actually not precise enough. If a subset $P'$ of size $k$ occurs in multiple faces, i.e. $P' = P_F = P_G$ for some faces $F$ and $G$, then we actually want to count it only once. It turns out that that makes the problem a bit more complicated. A solution is to explicitly remember the sets of size $k$ that we have encountered so far, and when we get a new set of size $k$ (when we enter a new face) test if it already occurs. This would require $O(k) = O(n)$ time per face, thus resulting in an $O(n^3)$ algorithm over all.