Computational Geometry - Homework I (Solutions)

K. Subramani
LCSEE,
West Virginia University,
Morgantown, WV
{ksmani@csee.wvu.edu}

1 Problems

1. **Tail Bounds:** In class, we established that the RANDOMIZED-QUICKSELECT() algorithm runs in expected time $O(n)$, when asked to find the k^{th} largest element in an array of n elements. Argue that there exists a constant c, such that the probability that more than $c \cdot n \cdot \log n$ comparisons are made in a run of RANDOMIZED-QUICKSELECT() is at most $\frac{1}{n}$.

   Solution: For this problem, we will need the Chernoff bound.

   Theorem 1.1 *Let X_1, X_2, ..., X_n denote the indicator random variables of n Bernoulli trials, each with probability of success p. The random variable $X = \sum_{i=1}^{n} X_i$ is a Binomial random variable with mean $\mu = \sum_{i=1}^{n} p = np$. The Chernoff bound states that for any $\delta > 0$,*

   $$\mathbf{Pr}[X - np \leq -\delta] \quad \leq \quad e^{\frac{-2\cdot\delta^2}{n}}$$

   See [MR95] or John Lafferty's notes for the proof of the above theorem.

   We can think of the execution of RANDOMIZED-QUICKSELECT() in terms of its computation tree. At the root, there is a node containing all the elements; after the first pivot operation, two children are created, with the left child containing elements less than the pivot element and the right child containing the remaining elements. The crucial observation is that unlike RANDOMIZED-QUICKSORT(), the computation proceeds along one branch only; this branch is called the *active* branch. Let us declare a node S on the active branch to be good, if either it is a leaf or both its children have cardinalities between $\frac{1}{4} \cdot |S|$ and $\frac{3}{4} \cdot |S|$. Observe that an active branch node has probability at least $\frac{1}{2}$ of being good. Secondly, there cannot be more than $k = \log_{\frac{4}{3}} n$ good nodes on any root-to-leaf path. (Why?)

   Assume that the active branch has $r \cdot k$ nodes in total. Clearly, the expected number of good nodes on this path is $\frac{r\cdot k}{2}$. We are interested in the value of r, such that the probability of not having at least k good nodes is less than $\frac{1}{n}$. We associate a random variable X_i, with the node of the active branch at level i of the computation tree. X_i is set to 1, if the corresponding node is good and 0, otherwise. Thus the total number of good nodes is $X = \sum_{i=1}^{r\cdot k} X_i$. Now, observe that Z is a binomial random variable and we can hence apply Theorem (1.1) to write

   $$\begin{aligned}
   \mathbf{Pr}[X \leq k] &= \mathbf{Pr}[X \leq (\frac{r \cdot k}{2} - k \cdot (\frac{r}{2} - 1))] \\
   &\leq e^{\frac{-2\cdot k^2 \cdot (\frac{r}{2}-1)^2}{r\cdot k}} \\
   &\leq e^{\frac{-2\cdot k \cdot (\frac{r}{2}-1)^2}{r}}
   \end{aligned}$$

In order to get $\mathbf{Pr}[X \leq k] \leq \frac{1}{n}$, we must have,

$$\begin{aligned}
e^{\frac{-2\cdot k\cdot(\frac{r}{2}-1)^2}{r}} &\leq \frac{1}{n} \\
\Rightarrow \frac{-2\cdot k\cdot(\frac{r}{2}-1)^2}{r} &\leq -\ln n \\
\Rightarrow \frac{2\cdot k\cdot(\frac{r}{2}-1)^2}{r} &\geq \ln n \\
\Rightarrow \frac{2\cdot 3.45\ln n\cdot(\frac{r}{2}-1)^2}{r} &\geq \ln n, \text{ exploiting logarithm identities} \\
\Rightarrow 7\cdot[\frac{r^2}{4}+1-r] &\leq r \\
\Rightarrow \frac{7}{4}\cdot r^2 - 8\cdot r + 7 &\geq 0
\end{aligned}$$

Solving for r, we get $r = \frac{(8\pm 3.87)}{\frac{7}{2}}$, from which we conclude that any $r > 4$ should suffice. In other words, with probability at least $(1-\frac{1}{n})$, the depth of the active branch is at most $5\cdot\log_{\frac{4}{3}} n$.

So what is the total number of comparisons performed by the RANDOMIZED-QUICKSELECT() algorithm, given that the depth of the active branch? Given that the number of comparisons at any level is at most n, it follows that the total number of comparisons is at most $5\cdot n\cdot\log_{\frac{4}{3}} n$. □

2. **Algorithm Design:** Given a set of n points in the plane, devise an algorithm that runs to check whether there exists a subset of 3 points, which are collinear. Your algorithm should run in time $O(n^2 \cdot \log n)$.

   Solution: Let $\mathbf{P}$ denote the set of points. The key observation is that if three points p, q and r are collinear, with q being the central point, then the slopes of the segments $\bar{pq}$ and $\bar{qr}$ are identical. This gives us the following algorithm:

Function CHECK-COLLINEAR($\mathbf{P}, n$)

1: **if** $(n \leq 2)$ **then**
2: **return(false)**
3: **else**
4: Sort the points in $\mathbf{P}$ based on their x-coordinates.
5: **for (** $i = 1$ **to** n**) do**
6: Compute the slopes of the segments $(P[i]\bar{,}P[j])$ for each $j = 1, 2, i-1, i+1, i+2, \ldots, n$ and store these in an auxiliary array $\mathbf{A}[i]$.
7: **end for**
8: **for (** $i = 1$ **to** n**) do**
9: Sort the auxiliary array $\mathbf{A}[i]$.
10: **end for**
11: **for** $(i = 1$ **to** $n - 2)$ **do**
12: **for** $(j = i + 1$ **to** $n - 1)$ **do**
13: Let $x = slope(P[i]\bar{,}P[j])$.
14: Perform a binary search on $\mathbf{A}[j]$ to check if there is a point $P[k]$ such that the slope of segment $(P[j]\bar{,}P[k]) = x$.
15: **if** (such a point $P[k]$ exists) **then**
16: "Points $P[i]$, $P[j]$ and $P[k]$ are collinear."
17: **end if**
18: **end for**
19: **end for**
20: Declare that no three points are collinear.
21: **end if**

Algorithm 1.1: Checking for collinearity

Analysis: Observe that sorting on the x-coordinate takes $O(n \cdot \log n)$ time. Computing the slopes of all the segments takes $O(n^2)$ time. Sorting all the auxiliary arrays takes $O(n^2 \cdot \log n)$ time. In Lines (11) through (19), the only non-trivial action is performing the binary search and hence these lines can be implemented in $O(n^2 \cdot \log n)$ time. It follows that Algorithm (1.1) takes $O(n^2 \cdot \log n)$ in the worst case. As an exercise, argue the correctness of this algorithm, using induction.

□

3. **Convex Hulls:**

   (a) Let P be a set of points in the place. Let P be a convex polygon, whose vertices are points from P and which contains all the points in P. Prove that P is uniquely defined and that it is the intersection of all convex sets containing P.

   Solution: Let Q be a convex polygon that is distinct from P, such that its vertices are points from P and it contains all the points in P. Since Q is distinct from P, we can assume without loss of generality, that there is a point in Q, which does not lie in P; however, this immediately implies that there is a vertex of Q, which is not contained in P and that contradicts the hypothesis. The case in which there is a point in P, which is not contained in Q is handled similarly. We have thus proven that P is uniquely defined.

   Let C denote the set of intersections of all convex sets containing the point set P. Let $x \in C$. Since P is a convex polygon, we must have $C \subseteq$ P and hence $x \in$ P. Likewise, let $x \in$ P. If $x \notin C$, then a vertex of P $\notin C$; however, this contradicts the hypothesis that C was the intersection of sets containing P. □

   (b) Let $p = (p_x, p_y)$ and $q = (q_x, q_y)$ be two points in the plane. We wish to test whether the points $r = (r_x, r_y)$ lies to the left or right of the segment $\bar{pq}$. Using first principles, explain how the sign of the determinant of D can be used for this purpose, where,

$$D = \begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}$$

Solution: The first principles reference is to the cross-product in vector algebra. The following theorem is easily proved using vector algebra:

Theorem 1.2 *Given two vectors $\bar{p}$ and $\bar{q}$, $\bar{p} \times \bar{q}$ is positive if and only if $\bar{p}$ is clockwise from $\bar{q}$.*

Observe that the vector representation of the segment $\bar{pq}$, in the direction from p to q is $((q_x - p_x), (q_y - p_y))$; we denote this vector as $\vec{pq}$. Likewise, $\vec{pr}$ is $((r_x - p_x), (r_y - p_y))$. Now point r lies to the right of segment $\bar{pq}$ if and only if a right turn is made at point q, along $\vec{pq}$ to get to point r. But this means that the vector $\vec{pr}$ is clockwise with respect to the vector $\vec{pq}$. This forces the cross-product $\vec{pr} \times \vec{pq}$ to be positive. Likewise, point r lies to the left of segment pq if and only if a left turn is made at point q along $\vec{pq}$ to get to point r. But this means that the vector $\vec{pr}$ is counter-clockwise with respect to the vector $\vec{pq}$ and hence the cross-product $\vec{pr} \times \vec{pq}$ is negative. It is straightforward to verify that the determinant of D indeed represents the signed magnitude of the cross-product $-\vec{pr} \times \vec{pq}$. Hence, if the sign of $det(D)$ is positive, we know that r lies to the left of $\bar{pq}$ and if it is negative, it lies to the right of $\bar{pq}$. Of course, if $det(D) = 0$, then the points p, q and r are collinear. □

4. **Line Intersection:** Let S be a set of n disjoint segments in the place and let p be any point which does not lie on any segment in S. The goal is to determine all the line segments that are *visible* from p. Note that a segment l in S is visible from p, if there exists a point q on l, such that the segment $\overline{pq}$ intersects only segment l. Devise an algorithm that runs in time $O(n \cdot \log n)$ for this problem.

   Solution: The key idea is to use a rotational sweep ray L, with p as the origin. As L completes a turn of 360° in the anti-clockwise direction, it intersects various segments in S. Regardless of the number of segments that L intersects at a point, the segment with the intersection point at the closest distance from p is definitely visibile from p. It is this observation that is exploited in the angular sweep.

   We make the following assumptions about the data:

   (a) The origin is at point p; if not, simply translate the coordinate system.

   (b) Let s_i denote the line-segment such that one of its end-points, say l_i, is the closest to p among all end-points of line-segments. Using rotation, if necessary, we ensure that l_i is on the x-axis. l_i will be the start position for the angular sweep.

   (c) We assume that l_i is to the right of p; if not use reflection. Without loss of generality, we can assume that $l + i$ has the smaller y-coordinate of the two points representing s_i. Can you see why this is true?

   (d) Every segment is represented by a record that stores both its end-points in polar form. Accordingly, segment s_i in S is represented by the points $l_i = (r_i^1, \theta_i^1)$ and $u_i = (r_i^2, \theta_i^2)$, where l_i is the point touched first in the angular sweep. Note that in linear time, we can determine which end-point of every segment is touched first.

   Proving that translation, rotation and reflection do not affect the solution to the problem at hand are easy exercises.

   Using the convention that u_i makes an angle of 0° at point p, sort all the end-points and place them in an event queue $\mathcal{Q}$. It is to be noted that $\mathcal{Q}$ only needs to support $delete()$ operations, after the initial insertions and hence it can be implemented a sorted linked list. There are precisely two types of events, viz., occurrence of the first end-point of a segment and occurrence of the second end-point of a segment. In what follows, we describe how these events are handled.

   (a) If the extracted event is the first end-point of a segment s_q, say $l_q = (r_q^1, \theta_q^1)$, then insert s_q into the sweep-status, using the angle θ_q as the key, in exactly the same way, as line segments were inserted in the sweep status in the line intersection problem. Essentially, we maintain a parametric polar representation of the segment and given a particular angle, it is easy to compute the point on the segment, which is intersected by the sweep line. All the line segments that intersect the sweep line are maintained in sorted order, in terms of their distance from the origin p. Once the insertion is done, we check to determine the segment that is closest to p on the sweep line and mark it as visible.

   (b) If the extracted event is the second end-point of a segment s_t, say $u_t = (r_t^1, \theta_t^1)$, then delete s_t from the sweep-status. The deletion of s_t may cause a segment which was previously invisible to become visible. Perform the appropriate check as in the case above.

   It is a straightforward exercise to inductively prove that the above algorithm reports those and only those segments that are visible from p.

   The sweep-status line is maintained as a balanced binary search tree so that all the operations for handling end-points take $O(\log n)$ time. The running time consists of the initial sort procedure which takes $O(n \cdot \log n)$ time, followed by a constant number of tree operations for each point in the event queue. Since no point is ever reinserted into the event queue, the angular sweep determines the segments which are visible from p in $O(n \cdot \log n)$ time. □

5. **Backwards Analysis:** Professor Amarsen proposes the following algorithm to find the maximum element in an array of n elements.

Function FIND-MAX$(\mathbf{A}, n)$
1: **if** $(n = 1)$ **then**
2: **return**$(A[1])$
3: **else**
4: Extract an element randomly from $\mathbf{A}$ and call it x. $\{x$ is no longer in $\mathbf{A}.\}$
5: $y =$FIND-MAX$(\mathbf{A}, n-1)$
6: **if** $(x \leq y)$ **then**
7: **return**(y)
8: **else**
9: Compare x with all the remaining elements in $\mathbf{A}$ and return the maximum
10: **end if**
11: **end if**

Algorithm 1.2: Finding Maximum in Paranoid Fashion

Is this algorithm correct? What is the worst-case number of comparisons on a run? How many comparisons are made in the expected case?

Solution: The correctness of the algorithm can be proved inductively as follows: Clearly the algorithm is correct when $|\mathbf{A}| = 1$, i.e., the array has only one element. Assume that the algorithm works correctly when $|\mathbf{A}| = n - 1$ (inductive hypothesis). Now consider the case in which $\mathbf{A}$ has n elements. By the inductive hypothesis, Line (5) of the algorithm correctly computes the maximum of the array $\mathbf{A}$, after one element has been deleted. Thus y contains the maximum of the initial array, disregarding the element x, which was removed from the array. Now, if x is at most y, then y is the maximum of the initial array (Line (7)) and if x is larger than y, then the comparison of x with all the elements in $\mathbf{A}$, ensures that the true maximum is returned. We thus see that in all cases, the algorithm correctly returns the maximum element in array $\mathbf{A}$. Using the first principle of induction, we conclude that the algorithm works correctly.

Let $T(n)$ denote the worst case number of comparisons on a run. The maximum number of comparisons occurs when Line (9) is executed. Accordingly, we get the following recurrence:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + n \end{aligned}$$

The solution to the above recurrence is $T(n) = \Theta(n^2)$.

To analyze the number of comparisons in the expected case, we observe that Line (9) is executed if and only if the element x which was extracted in Line (4) was the true maximum of the array. Consider the case in which $|\mathbf{A}| = i$ and let $T(i)$ denote the number of comparisons made in a call to FIND-MAX$(\mathbf{A}, i)$. Let X_i denote an indicator variable, which is 1, if Line (9) is not executed and i, if Line (9) is executed. Observe that $\mathbf{Pr}[X_i = i] = \frac{1}{i}$ and hence $\mathbf{Pr}[X_i = 1] = \frac{i-1}{i}$, since the element x is chosen randomly from the i elements in $\mathbf{A}$.

The total number of comparisons in the call FIND-MAX$(\mathbf{A},\ i)$ is therefore given by the recurrence:

$$\begin{aligned} T(1) &= 1 \\ T(i) &= T(i-1) + X_i \end{aligned}$$

Thus, the total number of comparisons made in call to FIND-MAX$(\mathbf{A}, n)$ is: $T(n) = \sum_{i=2}^{n} X_i + 1$. (Why?) Since we are interested in the expected number of comparisons, we observe that

$$\mathbf{E}[T(n)] = \mathbf{E}[\sum_{i=2}^{n} X_i + 1]$$

$$
\begin{aligned}
&= 1 + \sum_{i=2}^{n} \mathbf{E}[X_i] \\
&= 1 + \sum_{1=2}^{n} (\frac{i-1}{i} \cdot 1 + \frac{1}{i} \cdot i) \\
&= 1 + \sum_{i=2}^{n} (2 - \frac{1}{i}) \\
&\leq 1 + \sum_{i=1}^{n} (2 - \frac{1}{i}) \\
&= 1 + 2 \cdot n - H_n \\
&\leq 2 \cdot n
\end{aligned}
$$

We thus see that Algorithm (1.2) performs only a linear number of comparisons in the expected case. □

References

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, June 1995.