Homework 5

Vu Tuan Phong Pham

November 2019

1. *For purpose of demonstration I also submitted a couple of scanned file demonstrating the graph for question 1. (I don't know how to draw graph using latex I'm sorry :( )*

   (a) Construct a point and a line that does not go through that point, all other points are co-linear on this line.

   (b) Construct a rectangle using 4 points. Construct a ray that does not intersect the rectangle such that if we take a random point on the ray, only two fixed edge of the rectangle is visible to the random point.

   (c) *submitted on the scanned file attached.*
   diameter of figure 1 is 3
   diameter of figure 2 is 4
   diameter of figure 3 is 4

2. Let's consider the problem on the voronoi diagram. For 2 adjacent voronoi cell, the earliest time the lily pads of 2 owners of these 2 cells will be connected is when the lily pad grows to touch the adjacent side of these 2 cells. Let's construct a graph from this:

   - Each voronoi cell will be corresponding to 1 vertex of our graph.
   - For each pair of adjacent voronoi cells, we create an edge between them with the weight t such that after t unit of time, lily pads of both cells will touch each other and exactly at the adjacent side of 2 cells.
   - For each cell, we make an edge from them to "y^+" node (and "y^-" node), assigning them weight t with the meaning that it takes t unit of time for that lily pad to grow to touch line y^+ (same meaning for node "y^-" as well).

   In our new graph, we consider a weight of a path from a node u to a node v to be the maximum weight of an edge on that path. Our problem is now to find the minimum weight of all path from node y^+ to node y^-.

   We also make an even better observation, that this path has to be inside the minimum spanning tree of our graph (graph is guaranteed to be connected due to the third construction). Now we just have to perform Kruskal algorithm but stop until node y^+ and node y^- is in the same set, we can then trace back to return our path (easily done by storing a parent array, and start tracing from y^-).

   Since we only make an edge for adjacent voronoi cells, the number of edges is $O(n)$. Our space complexity is therefore $O(n)$. Our run time complexity is based on Kruskal algorithm, so it is $O((m+n)\log n)$, which is $O(n \log n)$ (linear number of edges).

3. If we generate the voronoi diagram, we can compute the intersection of the voronoi diagram with the line l in $O(n)$ by simply just following the current cell boundary until we find the next intersection,

and move on to the next cell. Spawning a voronoi diagram takes $O(n \log n)$, so overall complexity is $O(n \log n)$.

I couldn't really think of a way to solve this in $O(n)$, but I have a few approaches that might work:

(a) Since the list of points are sorted by x-coordinate, my first intuition is to proceed with the incremental algorithm to find a triangulation. On the way, we somehow use the triangulation to compute the intersection of voronoi diagram with the line l.

(b) Observe that if a point contribute something to intersection of voronoi diagram with line l, it contributes at most 2 intersections with line l. We can start by flooding point p to its left and right, computing the boundary of its voronoi cell, and also deleting point that is on the way and does not contribute the final intersection of voronoi cell and line l. Each point is deleted at most once, so it is linear. The only problem is to determine whether a point q makes with p a final contribution to intersection of voronoi diagram with line l.

4. (a) We simply query the point using our tree. At a node, we compare the distance from the node to the query point, and update our answer if necessary. At each node we traverse to, we also have to check the other sub-tree that we do not traverse to. We only traverse the unchecked sub-tree if the circle centered at query point and has radius of the current shortest distance intersect the division line of the current node. Expected time complexity is $O(\log n)$ given that the points are randomly distributed.

(b) If we divide the set of points into set with size d, then each query would take $O(\frac{n}{d}(\sqrt{d}+k))$ (because we have to search all the tree, and each tree query takes $\sqrt{d}$). For the insertion, we just insert the new point into a random tree (as long as we keep the sets balance in size), the insertion takes $O(d \log d)$. We need to find d such that we "balance" the query and insertion time complexity. One value of d we can try is $\sqrt{n}$. If $d = \sqrt{n}$, insertion takes $O(\sqrt{n} \log n)$ and intersection query takes $O(n^{\frac{3}{4}} + k\sqrt{n})$.

(c) Deletion can be done similarly to insertion. We need to find in what tree the deleted point is in, this takes $O(\frac{n}{d})$. After that, we have to rebuild the tree, this costs $O(d \log d)$. A thing we need to consider is if we're really unlucky and keep deleting from the same tree, the balance might be broken. We can fix this by transferring a point from another set with biggest size to our current tree we work on. This takes a constant time of doing $O(d \log d)$ computations, so the running time of deletion is $O(d \log d)$. For $d = \sqrt{n}$, deletion takes $O(\sqrt{n} \log n)$.