15295 Fall 2018 #12 Geometry: Problem Discussion
November 14, 2018

This is where we collectively describe algorithms for these problems. To see the problem statements follow this link. To see the scoreboard, go to this page and select this contest. The theme of this week's contest was geometry.

A. View Angle

B. Pyramids

C. Polygons

D. Large Triangle

There's a very short, nifty solution to this one. First, recall that we can find the area of a triangle by taking the absolute value of the cross product of two vectors that define its side. We then treat every point as the endpoint of a vector from the origin, so for a triangle i,j,k, we Can compute the area by adding the cross product from i to j, j to k, and then subtracting the cross product from i to k (assuming that j is the furthest from the origin in this case). This runs into some sign issues with the cross product, so some trial and error is necessary. We can precompute all the cross products in O(n^2) time, and check each triplet of points in O(n^3) time. This complexity is greater than some other solutions, but it runs fast enough because checking a triplet is just a couple of integer additions.
-Eliot

The intended solution is O(n^2 log n). Eliot's takes over 12 seconds. The original time bound was 3 seconds. But I increased it in order to get my solution to be accepted. I set it to 20 seconds when my solution was taking 11 seconds. Then I improved it to 6 seconds, but the time limit was left at 20, so Eliot's solution was accepted :-) I think I'll lower it to 14 seconds to at least make O(n^3) solutions harder to get AC.

Here's the O(n^2 log n) algorithm. It sweeps through all (n choose 2) angles among pairs of points in counterclockwise order. The angles actually go from 0 to π with each of the (n choose 2) pairs considered once.

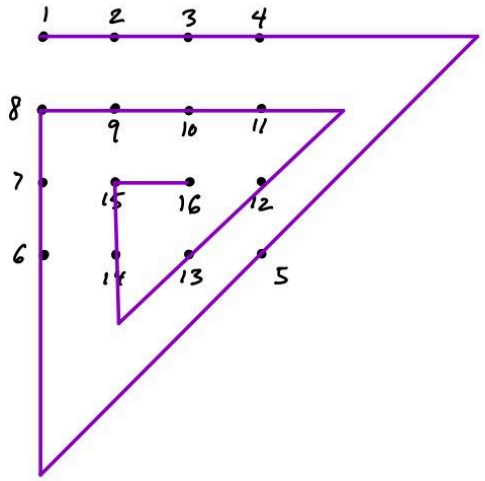
During this process it maintains an array of the points (actually their indices) in sorted order perpendicular to the current angle direction. When a pair of points (i,j) is being considered we want to see if there's a 3rd point with which we can form a triangle with the desired area. We do this with two binary searches over the array: one on each side of where i is in the array. (i and j must be neighbors in the array.) Advancing the sweep angle causes just one pair (i,j) of neighboring elements in the array to swap.

--Danny

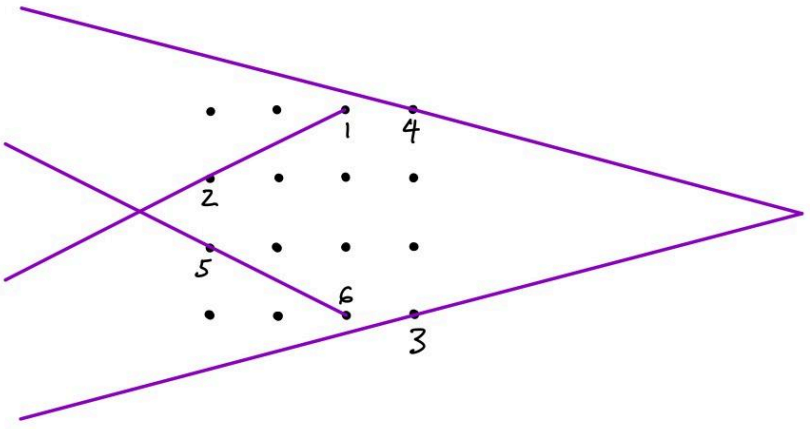
E. Connect the Dots

Let's make a couple of observations about the problem. First of all, you can assume WLOG that the first line goes through points 1 and 2. Because if it doesn't (and just goes through 1), you can replace it by a line from 1 to 2 preserving the rest of the solution.

The second observation is that sometimes it's necessary to have a line that goes through just one point. This is illustrated in the 2nd sample input, and shown below:



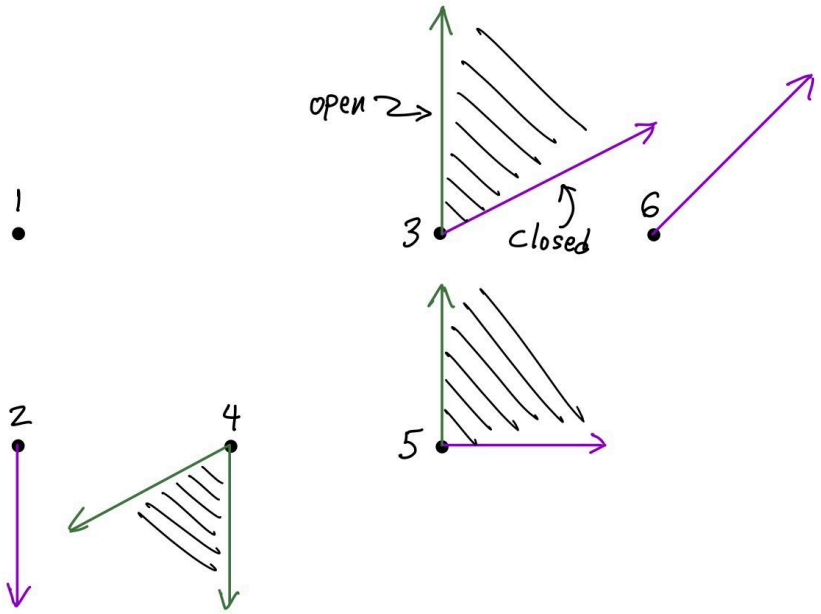
Note that the line through point 5 goes ONLY through point 5. Call this a "fulcrum line". It is natural to attack this problem by trying to limit the number of different lines that might occur in any solution. The problem with fulcrum lines is that it is hard to limit them. But maybe it's never necessary to have two fulcrum lines in a row? Oh, that's false, as shown in the following example:



(Clearly the top two (and bottom two) lines off to the left will meet eventually.) In this problem there are only 6 required points. The solution shown uses 4 lines, and there are two fulcrum lines in a row (the ones through 3 and 4). There is no other solution with just four lines.

So let's try something different. Consider a prefix of the sequence of points. Let's try, as in a

DP setup, to characterize what we know about the constraints that prefix puts on the rest of the solution. So we know that there's a line from point 1 to point 2. That line continues with a "ray" emanating from point 2. Now let's look at point 3. Whatever line goes through point 3, one end of it must intersect the ray from point 2. This leads to a "cone" of possibilities from point 3, as shown in the following picture:



Any line that intersects the purple ray emanating from point 2 then going to point 3, must end up continuing inside the cone drawn at point 3. Note that one side of this cone is shown in green, and is called "open". This means that the cone does not contain that line, but contains lines arbitrarily close to that line. The purple line means that that side of the cone is closed, and can actually be used in a solution. This cone completely characterizes all optimal solutions up to point 3. It gives us all the information we need to proceed forward building the rest of the solution.

Continuing our example, look at point 4. In this case 4 is strictly inside the "reverse" of the cone at 3. Thus, the cone at 4 has boundaries parallel to those at 3, but both of them are open. Continuing to 5, we see that its cone is again open on one side and closed on the other.

Now we come to point 6, which is inside the cone of 5. In this case we insist that we use the line from 5 directly to 6. Thus the cone from 6 is degenerate (a ray) like the one at point 2. Although there may be perfectly good solutions (maybe even optimal ones) that do not use this line from 5 to 6, there is always an optimal solution that does. Here's a proof:

> Consider a solution where the line L_5 exiting point 5 does not go through point 6. There must be some line L_6 entering point 6 and continuing on to the rest of the solution Simply adjusting all the lines up to and including L_5 so that it becomes L_{56}, the line from 5 to 6, and leaving the rest of the solution from L_6 on, produces a solution that is no longer than the previous one. QED.

So the algorithm is to just walk through the points in order, building the sequence of cones described here. There are only four types of cones that ever occur, based on whether each side is open or closed. (There may be some cases not illustrated in the above diagram, but they're all analogous to those shown, and easy to figure out.) We count 1 for each point except point 2, and ones where the point is contained in the current cone. This is the desired answer. So the algorithm is O(16).

A natural extension to this problem is to find an optimal solution with the "smallest size", i.e. one that stays as close as possible to the middle of the 4x4 grid. :-)

---Danny

F. The Last Hole!

Consider a pair of points (a,b), and vector v from a to b. Think of v as being a vertical vector pointing up. Process all the other points one at a time. Keep track of the rightmost circle defined by the a, b, and one of the other points to the left of v. These three points now define a triangle which we consider as a potential Delaunay triangle. We check that it is by making sure that its circumcircle is empty. (Use the incircle test from these notes.) The running time of this part is O(n^3).

Now the circle containing these three points is the origin of a hole if and only if the set of points that are on this circle are "smooth". By smooth I mean that the polygon of these points strictly contains the center of the circle. (Alternatively, the "angle subtended" by each side of it is < π.) This check takes O(n log n) in the worst case where the circle has O(n) points on it.

You could use the method here to compute the Voronoi diagram or Delaunay triangulation in O(n^3 log n) time.

Actually, this gives rise to a very simple O(n^2) algorithm to compute the Delaunay triangulation of a set of n sites. Actually, it's not going to find a triangulation. It's going to find a diagram in which all the sites on each face are co-circular (on the same circle), and that circle has no sites inside of it. (If the face is not triangular, then it's trivial to triangulate it to generate a Delaunay triangulation.)

The algorithm keeps a queue of directed Delaunay edges for which it still may need to find the cycle containing it. It takes linear time to find a cycle containing a given edge. After finding a cycle containing an edge, it adds all the reverse edges of the ones it found to the queue. In this way it traverses through the diagram finding everything. (To get started the algorithm finds the closest pair, which must be a Delaunay edge.) Since the diagram has O(n) edges in it, the algorithm is O(n^2).

The algorithm is described in more detail on pages 4 and 5 of these notes.

--Danny