15295 Fall 2019 #8 Geometry -- Problem Discussion

October 16, 2019

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.

A. Fancy Fence

A regular polygon with *n* sides has each of its angles equal to a = 180 * (n-2) / n. Solving this backward, a regular polygon with each of its angles being a will have n = 360 / (180 - a) sides. So, whenever this value of n is an integer, the provided angle a is valid.

B. Trace

C. Nearest vectors

One idea that comes to mind for this problem is to sort all of the vectors in the order of increasing angles with the x-axis, and then picking the pair of adjacent vectors in the sorted list with the smallest angle between them. The obvious idea is to just compute atan2(y, x) for each vector (x, y) to obtain its angle. Computing the angles between adjacent vectors then just amounts to looking at the difference of their associated angles with the x-axis. However, this is actually a bit too imprecise and will not pass the problem (it seems that using the type "long double" in C++ can make this work, though).

An alternative idea is to note the following: Supposing A and B are in the same half-plane (above or below the x-axis) we can tell which is to the right of the other entirely by looking at the sign of the cross product A x B (recall the right-hand rule). More precisely, this is the sign of the third component of the cross product (the only nonzero one for two vectors in the xy-plane).

Now we have a way to sort the vectors in increasing order of their angles without any floating point arithmetic. Finally, to complete the solution, we need to know a better way to compare the angles between two pairs of vectors that does not involve imprecise computations. One idea which works is the following: Recall that for arbitrary vectors in the plane, |A x B| = |A| |B| sin theta and A dot B = |A| |B| cos theta, where theta is the angle between them. So if we form the vector (A dot B, |A x B|), the angle it forms with the x-axis will be the angle between the vectors A and B. Since we already have a way to compare the angles that two vectors make with the x-axis, we're done (and thus have a solution that uses no floating point arithmetic whatsoever).

-- Wassim

D. Dendroctonus

Essentially we look for a circle that encloses all Reds (infected) but no Blues (not infected). We output

"No" if no circle exists and "Yes" otherwise. Note that Blues can lie ON this circle but not within. An
important observation is that the optimal circles to use are the ones formed by each triple and each pair
of points (These points also include Blues since Blues can be on the circle!)
To explain, say a circle isn't formed by a triple or a pair, then I can always expand it so that more points lie on its boundary. If these points are red, then I've included more reds, and if these points
are blue, then I've avoided including blues, so I am always better off. We need to special case for
n < 2 since in that case no pair or triple exists. We can always return "No" for n < 2. The number of
such circles is only 100^3 + 100^2 so we can bruteforce search through all circles formed by triples and pairs,
and check if any circle encloses all reds but no blues. Finally, for precision, we check if a point lies on a
circle using epsilon equality, and I used long doubles for everything.

The formula to find center from 3 points can be easily found online.

--Sheng

E. Freelancer's Dreams

This problem can be solved by linear programming. Our goal is to minimize $\sum_i x_i$

subject to $\sum_i x_i * a_i >= p$ and $\sum_i x_i * b_i >= q$, and for all i, $x_i >= 0$. The dual of this linear program is to maximize x * p + y * q, subject to x >= 0, y >= 0, and for all i, $x * a_i + y * b_i <= 1$. x and y are parameters introduced by the dual.To learn about linear programming, read these notes from 451: https://www.cs.cmu.edu/afs/cs/academic/class/15451-s19/www/lectures/lec15Old.pdf.
Now note that linear programs are inherently convex since all constraints are straight line, so the region they form is convex, and so the value of $x * p + y * q$ is convex in x too, meaning that it is unimodal (increases, then decreases). Therefore we can use a ternary on x. Note that x >= 0 and $x * a_i <= x * a_i + y * b_i <= 1$, so we can apply ternary search with lower bound 0 and upper bound $min_i\, 1 / a_i$. For each possible x, we pick y to be as large as possible: $max_i\, ((1 - x * a_i) / b_i)$, then we calcuate $x * p + y * q$.

I guess if you don't realize the convex thing, you can also use the seidel's algorithm or ellipsoid to solve the linear program, though that might run slower.

--Sheng

Dan's comment: This works, but doesn't use geometry. There is a relatively simple geometry solution that doesn’t require linear programming.

F. Runaway to a Shadow

The cockroach begins at some arbitrary location (x0, y0). It must reach a shadow within time T, so the max distance it can move is r0 = v x T. It uniformly chooses some random direction, and moves at most this distance in a straight line, stopping if it reaches a shadow.

The basic idea here is simple. If we consider all of the [0, 2 * pi] possible angles the cockroach can move in, we need to sum up (union, in reality, due to overlaps) the parts of this interval that leads to the cockroach reaching a shadow in <= r0 distance, and just divide the sum by (2 * pi) at the end to obtain the required probability.

To do this, there are a few cases we need to consider, for each plate (xi, yi, ri):

- If the cockroach starts within the shadow of this plate already, the probability of the cockroach’s survival is just 1.

- If this plate is fully disjoint from the circle formed by (x0, y0, r0), then we don’t even need to consider it in our computations, since the cockroach can never even reach it.

- The interesting case is when the cockroach’s circle intersects (including just touching) with this plate. Here, we need to find (given (x0, y0, r0) and (xi, yi, ri)) what range of angles (let’s say [Si, Ei]) are possible for lines segments of length <= r0 starting at (x0, y0) to intersect with the circle (xi, yi, ri). This gives a range of angles the cockroach can move in, to reach this particular plate.

  Due to my limited background in geometry, I couldn’t figure out how to properly calculate this angle for each circle in time, during the contest :). In the simple case, it’s just

  Let A = slope of vector from (x0, y0) to (xi, yi), then the required interval is just

  [A - Sin^{-1}(ri / distance between the centres), A + Sin^{-1}(ri / distance between the centres)]

  but other cases exist. If someone wants to add onto this, please do.

  We also need to be careful about maintaining each interval in the same form (maybe all angles being between 0 to 2 * pi, making sure things don’t negative or above 2 * pi, and bringing them back if they do).

Once we have each of these intervals [Si, Ei] for i = 1, 2, ... , n we just union and sum them to find the total possible angles the cockroach could move in, to survive. Then, this angle over (2 * pi) gives us the required result.

Yep the way I did this one is exactly as above. For the interesting case where the two circles intersects, there are 2 possibilities. Either we use the angle between cut edges (this happens when the intersection points are “behind” the cut edges) or we use the angle between intersection edges.

There are 2 ways to distinguish whether we use the cut angle or the intersection angle.

1. Just calculate both and pick whichever is larger
2. Compare the length of a cut edge from the origin to the shadow plate, to the length of the v * T. If v * T is long enough, then the cut angle should be used.

How the cut angle can be calculated is already mentioned above. To calculate the intersection angle, we can just use law of cosines to find an angle B (a drawing would be helpful here), and then the intersection angle is between angles A - B and A + B.

There are some common pitfalls in this problem:

1. There are some really large testcases, and it is easy to overflow even using the long long type in C++. For example, the v * T can be really large, so we should avoid squaring v * T.
2. In the end, when unioning and summing, we need to pay attention to wrapping around. A clean way to do this is before doing a single scan through, partition all angles that start from a positive angle and ends in a negative angle (such as 0.9PI -> -0.9PI) into two angles: 0.9PI to PI, and -PI to -0.9PI.

--Sheng