Code Jam 2010 - World Finals

Analysis: Ninjutsu

Getting something discrete

A fact that jumps out immediately is that this problem is not discrete. The rope length that gives an optimal solution might be a real number, and there does not seem to be a way to ensure that this number is rational or discrete in any other way.

Given any real-valued rope length, we can simulate and count the bends, but this number is not a convex function of the rope length, so ternary search is out.

What we really wish we could do is dynamic programming, where the state is a triple -- the point we are currently swinging around, the direction the end of the rope is pointing in, and the length of the rope. Two of these values are real numbers. However, let's see how many different real values we actually need to care about.

As we are swinging the end of the rope around some point **P**, nothing interesting happens until the moving segment of the rope touches another point, **Q**. At that... point, we need to make a decision -- we could either bend the rope and switch to swinging around point **Q**, or cut the rope at exactly point **Q** and continue swinging around **P**. There are at most $\mathbf{N}^2$ pairs of points, so we only need to consider at most $\mathbf{N}^2$ different directions. That takes care of the second DP parameter.

What about the rope length? Using the same reasoning as above, we can show that there is only a finite number of rope lengths that are interesting. Firstly, note that bending the rope does not change its length. Whenever we swing around point **P** and hit point **Q**, there is a binary decision -- is the rotating rope segment longer than the distance from **P** to **Q**, or shorter?

In other words, the interval of real numbers between 0 and **R** can be split into a finite number of of sub-intervals, such that the number-of-bends function has a constant value on each sub-interval. This observation suggests the following, naive dynamic programming (DP) solution. The DP state is a pair of points (**P**, **Q**) and a real-valued rope length, **r**. The answer is the maximum number of bends we can achieve by continuing to spin around point **P**, if we are currently poining in the direction of point **Q** and have a swinging segment of the rope of length **r**. To be a bit more precise, we will actually need to have twice as many states -- one for poining in the direction of **Q**, and one for poining 180 degrees away from **Q**. We need the second kind of state for the situation when we decide to bend the rope and continue swinging around **Q**. At that point, we will start pointing in the direction 180 degrees away from **Q**. One way to implement this solution is simply to use a point, a 2D vector and a length as the state, instead of two points and a length.

Next, let's get rid of the real-valued length as a DP parameter. Because rope length only changes when we decide to cut it, we can replace the real-valued length parameter with an integer -- the number of bends since the last time we have cut the rope. Let's also replace the first two parameters with the pair of points that caused the cut. We are now interested in all situations of the following sort -- we were swinging around point **P**, with the end of the rope poining in the direction (**dx**, **dy**), at which point we cut the rope, and since then, we have made **K** bends. This information is enough to identify a state uniquely, and it implies a particular remaining rope length. Our DP state now becomes a triple of **P**, (**dx**, **dy**) and **K**.

Floating point issues

We now have an almost completely discrete problem. The only place where floating point numbers are necessary is in the testing of whether the current rope length is long enough to hit some given point, or does the rope's end pass underneath that point? Given the guarantee that the optimal solution works for a long range of rope lengths (0.999999), we can avoid floating point rounding trouble by being conservative: only assume that the rope can hit a point if its length is longer by at least, say, 0.5 than the distance to the point. If it is shorter than that, then certainly such a solution doesn't work for a long enough range of rope lengths, so it's not the optimal solution by the guarantee.

Dealing with loops

The naive solution is too slow. Consider, for example, the case where we have a rope of length 10^9 and two points: (0, 0) and (0, 1). The optimal solution uses the full length of the rope to create 10^9-1 bends. Of course, we are just going around in loops, so we need a way to detect and handle such loops to have a chance at a polynomial-time solution.

First, let's reorganize the naive DP solution a bit to make loop detection easier. We will have a simpler, three-parameter state and use a memoized recursive function. Each call of the function will correspond to a situation of the following kind -- we are spinning around a point **P**, with the end of the rope pointing in the direction (**dx**, **dy**), and we have just cut the rope because it has touched another point, **Q**. The total number of such states is O(**N**^3), but many of them are not possible and will never be visited.

Inside the function, we will simulate the wrapping of the rope and make recursive calls in situations when we decide to cut the rope further. If we simulate the wrapping process naively, we may need to make a huge number of recursive calls. Instead, imagine that we find ourselves in a situation where we have just bent the rope around some point **A**, and we are about to bend it again around some point **B**. If this is not the first time we have seen the pair (**A**, **B**), then we are looping around the same set of points. The second time we hit this pair, we will have a shorter remaining rope length, **r**, and we can figure out how much rope one revolution consumes by subtracting the new value of **r** from the value we had when we first encountered the pair.

Once we detect a loop, we can choose the number of full revolutions we want to make before we cut the rope and enter inside the convex hull of the points we are looping around. Clearly, it never makes sense to cut off more than one whole loop perimeter. We would be throwing away free rope bends. The optimal number of revolutions is, thus, the total remaining rope length, divided by the perimeter of the loop, floored, minus one.

Putting together a complete solution

We now have a DP solution with O(**N**^3) states and O(**N**^2) work per state (linear number of recursive calls; linear amount of work to find the next point for each call). We can speed this up by precomputing next points for each state, but this is not necessary. It turns out that the number of reachable states is small enough for this solution to pass.

Remaining difficulty is in dealing with collinear points. Since all coordinates are integers, this should be done exactly, without using floating point computations.