Code Jam 2017 - World Finals

Analysis: Omnicircumnavigation

Omnicircumnavigation: Analysis

The concept of omnicircumnavigation requires a given itinerary to touch every possible hemisphere. Pick any plane P that contains the origin. The plane splits the surface of the sphere into 3 parts — two open hemispheres and a circle between them. If the travel path lies entirely within one of the hemispheres, then it is not an omnicircumnavigation. The problem is to find such a *dividing plane* P or prove that one does not exist.

Continuing with that reasoning, the travel path touches a plane P if and only if one of the stops is on P, or there are stops on both hemispheres so that the connection between them passes through P. Therefore, another equivalent definition of dividing plane of a travel path is a plane that has all stops strictly on one of the hemispheres. This means the order of the input points is not important: the answer is the same for any valid permutation of a given set of points!

Notice that each actual stop S is given in the input by giving another point S' such that S is the normalized vector of S'. That means the origin, S and S' are collinear, which in turn implies that any plane P that contains the origin leaves both S and S' on the same hemisphere. Then, checking for the actual stops to be on a side of a plane is equivalent to checking the points S' given as input. And since points S' have the lovely property of having all integer coordinates, it is much better (and precise) to use them directly.

To summarize, the problem we are presented is equivalent to a simplified formulation: given a set X of points with integer coordinates, decide whether there exists a plane P that contains the origin such that all points in X lie strictly on the same side of P. Let us call a plane that goes through the origin and leaves all points strictly on one side a *dividing plane*.

Convex-hull based solutions

Notice that if there exists a dividing plane P, then P also has the convex hull of all points on one side. Moreover, by convexity, a dividing plane exists if and only if the convex hull does not contain the origin. So, one possible solution, that can even work for the Large, is to calculate the convex hull and check whether it contains the origin. If you do this using a library, it might be easier to calculate the convex hull of X plus the origin and check whether the origin is indeed a vertex of it. This solution, however, has two major drawbacks: 1. the algorithm to do convex hull in 3d is pretty hard, and 2. many implementations will have either precision issues, overflow problems, or get slowed down by handling increasingly big integers. This is because the needed plane can be really skewed, with angles within the 10^{-6} order of magnitude. Moreover, if the entire input is coplanar, then the convex hull might fail. One way to take care of both problems is to calculate the convex hull of X plus the origin plus F where F is a point really far away. F is not going to be coplanar, and it will also make the convex hull not have extremely narrow parts. Of course, the addition of F may make the convex hull contain the origin when the original did not. We can solve that with a second pass using the antipode of F, -F. If the original convex hull contained the origin, then both of the passes will. If the original convex hull didn't, then at least one of them won't (the one where F or -F is on the appropriate side of the dividing plane, since they are necessarily on different sides).

A simplified way to check for this is to notice that, in the same way there is a triangulation for any convex polygon, there is a tetrahedralization of any polyhedron. That means, we can avoid explicitly calculating the convex hull if we check all possible tetrahedra. This can't give false positives because all of them are included in the convex hull, and since some subset of those

tetrahedra will actually be a partition of the convex hull, their union is the entire convex hull, and one of them contains the origin. We can conclude that the convex hull of X contains the origin if and only if some tetrahedron with vertices in X does. The coplanar case can be simplified in this case: if the entire input is coplanar, we can check for any triangle to contain the origin. This solution, however, takes time $O(\mathbf{N}^4)$, which is definitely too slow for the Large dataset, and might even be slow for the Small, given that checking for each tetrahedron to contain the origin requires quite a few multiplications, which takes significant, though constant, time. The coplanar edge case of this solution can also be avoided by adding phantom points F and -F as above.

A speedup of the solution above that is certainly fast enough to pass the Small is to notice we can fix one of the vertices and try every possible combination of the other 3. This is because, for any vertex V, there is a tetrahedralization of the convex hull such that all tetrahedra in it have V as a vertex. This takes the running time down to $O(\mathbf{N}^3)$, which is definitely fast enough for the Small dataset, even with a large constant.

Solutions based on restricting the dividing planes

As usual in geometry problems, we can restrict the search space from the infinitely many possibilities to just a few. Suppose there is a dividing plane P. If we rotate P while touching the origin, we will eventually touch at least one point S from the input. If we rotate while around the line between S and the origin, we will eventually touch another point from the input. That means we can restrict planes P to those who touch two points from the input. Of course, the plane P is not the dividing plane (since it touches points from the input), but P represents planes epsilon away from those touching points. This means we need to take special care of inequalities to make sure that small rotation doesn't ruin the solution. In short, if there is another point touching P, we can't necessarily rotate P to make all 3 points on P to lie on one side. We need to take care of this coplanar case separately, with either solving the 2-dimensional version of the problem, or using phantom poins. Since 3 points (two points from the input and the origin) completely determine a plane, this restricts the number of planes to try to $O(\mathbf{N}^2)$ possibilities. For each one, we need another pass through the input to check on which side of the plane each point lies. This yields a solution that runs in time $O(\mathbf{N}^3)$, which is enough to pass the Small. Even in fast languages, this can be too slow for the Large, as the constant is, once again, very significant.

The solution above can be made run much faster, by randomizing the input, and thus, the order in which we check the points. For most planes, there will be several points on either side, and then when checking in a random order, the expected time to find one on each side (after which, we can stop) can be greatly reduced. Notice, however, that a case in which all the points are coplanar will not have its running time improved by this randomization, as no point will fall strictly on one side. For this to work well, we need to check for the all-coplanar case and special-case that one before launching the general case algorithm. This randomized version is indeed enough to pass the Large.

A bazooka-to-kill-a-fly solution

And finally, we can use linear programming. A plane that contains the origin is defined by an equation $Ax + By + Cz = 0$, for some A, B and C. A plane that has all points on one side has to satisfy either $AX + BY + CZ > 0$ for each point (X, Y, Z) or $AX + BY + CZ < 0$ for (X, Y, Z). Notice that if a triple (A, B, C) satisfies one of the conditions, then (-A, -B, -C) satisfies the other. So, we can restrict ourselves to one of the two cases, and then define a polytope with the set of inequalities $AX + BY + CZ > 0$ for each (X, Y, Z) in the input. The answer to the problem is whether that polytope is empty. Most LP algorithms figure that as an intermediate result towards optimization, and some libraries may provide functionality to check that directly. For others, you can just optimize the constant function 0 and see whether the answer is indeed 0 or "no solution".

As simple as the description of this solution is, it has a lot of issues to resolve. If using a library, it is highly likely that you run into similar precision / large number problems as the convex hull (and for the same reasons) that may make it either wrong or slow or both. If you want to implement your own algorithm, well, it's long and cumbersome to do it and avoid precision problems. There are tricks here, too. We can catch the possible problems and try to rotate the input to see if the library performs better. We can add additional restrictions like bound A, B and C to absolute values up to 10^6 to have a bounded polytope to begin with. That being said, judges tried 4 different LP libraries, and only one of them worked, after adding both additional restrictions (the library wouldn't handle unbounded spaces) and rotating the input a few times. Adding far-away phantom points can also help the LP, because it avoids the same problems as in the convex hull case. Of course, if you had a really robust prewritten algorithm or library, this option was best, even enabling a possible 3-line solution that passes the Small and the Large.