Intermediate Computational Geometry

Alex Chen

February 17, 2012

1 Introduction

Computational geometry is useful to know, since as with many topics we've covered this year, it's useful beyond USACO and other programming contests. If you end up working with graphics, robotics, or in image processing, there is no doubt many of the contest computational programming tricks might come in handy. This topic is one of the less common ones in terms of USACO, but geometry problems tend to be hard, both to solve and to code.

The essence of computational geometry is developing algorithms to deal with geometric figures: lines, points, polygons, polyhedra, etc.

2 Basic Tricks

Solving geometry problems is often aided by knowledge of geometric tricks, such as using the cross product to determine the relative position of two line segments or using the Shoelace theorem to find the area of a polygon.

Refer to the other computational geometry lecture if you would like details on the tricks.

2.1 Monte Carlo

A Monte Carlo algorithm for computational geometry is one that relies on random numbers. It's not completely reliable all of the time (obviously, because it's *random*), but the algorithm is surprisingly useful. For example, suppose we want to find the area of a polygon. We know how to figure out if a point is within a polygon. Thus, we can generate a million random points within the polygon's bounding rectangle and calculate the fraction of the points that end up within the polygon. Using this fraction and the rectangle's area, we can estimate the area of the polygon!

There are many more applications out there, but they are for you to discover.

3 Closest Pair of Points

3.1 Problem

Given N points in two-dimensional space, find the two points that are closest together.

Warm-up: Solve this problem for one-dimensional points.

3.2 Naive Solution

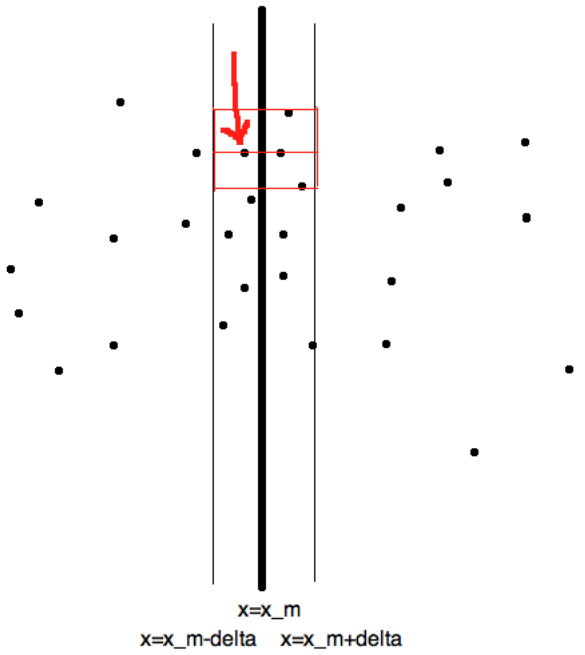
Compare every pair of points and keep track of which two are closest together. This is pretty straightforward and also incredibly simple to code. Complexity: $O(N^2)$.

3.3 Divide and Conquer

A divide-and-conquer algorithm is one that features two major steps: dividing and conquering. In this case, we will be dividing the points into approximately two groups, based on x-coordinate, and conquering the individual groups by recursively finding the closest distance within that group. Then, we will conquer the entire group by finding the closest distance between a point in group 1 and a point in group 2. The minimum of the three minimum distances (left group only, right group only, and left and right group combined) is the answer.

1. Find x_m, the median x value of the points in the set.
2. Divide the points into two groups, one of points with $x_i \leq x_m$ and one with $x_i > x_m$.
3. Recursively find δ_l, the smallest distance between any pair of points in the left group, and δ_r, the smallest distance between any pair of points in the right group.
4. Let $\delta = \min(\delta_l, \delta_m)$. Find δ_c, the minimum distance between a point in the left group and a point in the rightmost group. The answer is $\min(\delta, \delta_c)$. The tricky part is finding δ_c.

Finding δ_c: The major observation is that we only have to consider points that have x coordinates within δ from x_m, because if a point has an x value outside of that range, then there is no possible way for a counterpart across the line $x = x_m$ to improve on δ. Suppose (x_i, y_i) be a point such that $|x_i - x_m| < \delta$. How do we quickly check all the points that might be within δ of (x_i, y_i) to find an improvement upon the original δ (and find δ_c)? Consider the picture below:



Suppose we are looking for the nearest point to the one labeled with the arrow, and we're only looking for points on the right side. Because we do not care if $\delta_c > \delta$, we are only checking the points within the rectangle that has height 2δ, that is, the points with an x value within δ of x_m and a y value within δ of y_i. There are at most 3 points on the right side that have x values within δ of x_m. Why? You can prove this by contradiction. If there were 4 points, then δ would not actually be the smallest distance anymore. That means, for each point on the left, there are only at most 3 points on the right side to check. To check all the points on the left side, we have to sort all the points by y coordinate, giving us $O(N \log N)$ for this conquer step. This can actually be improved to $O(N)$.

What is the complexity of this algorithm? We can write it as $T(N) = 2T(N/2) + O(N \log N)$, where $T(N/2)$ represents one half of the "divide" and $O(N \log N)$ is the conquer. This can be lowered to $T(N) = 2T(N/2) + O(N)$. How?

3.4 Even Faster Algorithms

If the floor function is computable in constant time, then there exists a solution that runs $O(N \log \log N)$. If we take advantage of some random algorithms, $O(N)$ is possible as well. There's also a dynamic version, where points are sequentially added or removed, and you must find the closest pair after each operation.

4 Line Sweep Algorithms

4.1 Problem

Given N line segments in two-dimensional space, find the number of intersections. No two segments are parallel.

4.2 Naive Solution

For each pair of segments, find the number of intersections. Complexity: $O(N^2)$.

4.3 Line Sweep Algoritm

Line sweep algorithms are somewhat like sort-and-scan. They rely on sweeping all the data points by increasing x coordinates. How can we apply line sweep to solve the line segment intersection problem quickly?

The key to line sweep is to have the sweep line hold information. This vertical sweep line works in that all the line segments to the left of it have already been counted, and the ones to the right have not. At any point, the sweep line knows which segments intersect it and the relative order of the y coordinates at which these segments intersect the sweep line (thus, the segments that intersect the sweep line are sorted based on y coordinate of intersection). As the sweep line "sweeps" through the line segments, it stops at important points, which consist of any coordinate where two segments intersect, a segment begins, or a segment ends.

Solving this problem requires a special data structure that can easily insert, delete, and swap elements. Luckily, a binary tree is good for this.

- Initialize the sweep line.
- Initialize the sorted list (sorted by x) of important points to stop at. This can be in a binary tree because points may be inserted as the line sweeps.
- Initialize the segment storage structure. This holds the order of the line segments intersecting the sweep line.
- Visit the next important point:
  - If the important point is a line segment start, insert the line segment into the segment storage structure. Add the intersections between the new segment and its intersections with its new "neighbors" into the important points list.
  - If the important point is a line segment end, remove the line segment from the segment storage structure. If two new points are now neighbors, add their intersection into the important points list.
  - If the important point is an intersection point, increment the counter and swap, in the segment storage structure, the two line segments that intersect.
- At the end, the counter will equal the number of intersections!

What is the worst case time complexity of this algorithm? Interestingly, it is not guaranteed to be faster than the naive solution based on the implementation details given above. The complexity of this algorithm depends largely on the data structures used to implement the important points list and the segment storage structure.

4.4 Bonus

Solve this problem for lines instead of line segments.

5 Other Topics

Some other computational geometry topics that are not covered in this lecture:

- Convex hulls
- Geometric duality (will be topic of a future lecture)
- Voronoi diagrams / Delaunay triangulation
- Polygon triangulation
- Nearest neighbors
- Linear programming
- and many others...

6 Problems

General tips for computational geometry:

- If a problem looks like it's computational geometry, think about it some more. Can you turn it into a graph theory problem or a problem of another type?
- Use what you know about vectors, because things you learn in math class are very useful here.
- Sort and scan! Divide and conquer! These are among two of the various useful techniques in geometry algorithms.
- The more dimensions something has, the slower the algorithm (and it's usually exponentially slower). This is *the curse of dimensionality*.
- Think about geometric duality: can we represent the lines as points, or the points as lines to make the problem easier to solve? We'll cover this in more depth in the future.
- When coding a geometry problem, be sure to check floating point errors (precision checking)!

Problems for your enjoyment:

1. (Training Pages) Given line segments that do not intersect, except at endpoints, find the smallest perimeter of a polygon that can be formed from the line segments.
2. (JAN06 Gold) Given a convex polygon with less than 150 vertices, find the maximum number of non-intersecting line segments between non-adjacent vertices. To add a complication, there are certain circles on the graph that these line segments cannot pass through.
3. (FEB12 Gold) Given N points, find the number of lines of symmetry.
4. Given a list of N polyhedra (total number of vertices $\leq 1,000$), find the minimum distance path from one point to another that does not go through any of the polyhedra.
5. (OPEN09 Gold) Given a circle and N ($N \leq 50,000$) lines, each of which intersects the circle twice, find the number of pairs of lines that intersect within the circle.
6. (OPEN10 Gold) Given N ($N \leq 100,000$) points, find the number of triangles that can be formed from these points that pass through the origin.
7. (DEC08 GOld) Given N ($N \leq 250$) points, find the greatest number of points that can be used to form a convex polygon.