Advanced Computational Geometry Topics

Jacob Steinhardt, Tom Morgan, Jeff Chen

April 13, 2007

1 Tallest Line

Given N lines of the form $y = ax+b$, compute the tallest line at any given point in $O(NlogN)$ time. How would we go about doing this? Generally in computational geometry problems, it is a good idea to sort by something. There seem to be a few choices of what to sort by, but, drawing a few pictures, we see that one line can "overtake" another line only if it has a larger slope (and, in fact, will eventually do so). So for now, we will sort the lines in increasing order of slope (note that this does not preclude the fact that slopes may be negative). Now, if a line l_1 has greater slope than line l_2, and l_2 intersects l_1 at some point, it will continue to be higher for all points coming after that point. This allows us to do the following algorithm: starting with just the line with the smallest slope, maintain a list of on what intervals each line is the highest (note that the interval for any given line should be continuous). Then progressively add in each line with a higher slope, updating the list, until we have added in all the lines (be careful with parallel lines!). In each case, we need only find the intersection of the line with the last line added. If our current line completely covers that line, remove it from the list, and proceed with the second to last added, etc. Since we can only remove each line from the list once, and only add it once, and finding the intersection of two lines if $O(1)$, the overall runtime of this is $O(N)$. The sort at the beginning, therefore, is the slowest part of the algorithm, so that it is indeed $O(N)$. Let's look at an example:

$$l_1 = -x + 4$$
$$l_2 = 4$$
$$l_3 = x + 5$$
$$l_4 = 2x + 6$$
$$l_5 = 3x + 1$$

The lines are already sorted, so start with l_1, which, as it is the only line, is tallest on $(-\infty, \infty)$. Now add l_2, which intersects l_1 at $x = 0$. Now l_1 is tallest on $(-\infty, 0]$ and l_2 is tallest on $[0, \infty)$. Add l_3, which intersects l_2 at -1. This is before the start of the interval on which l_2 is tallest, so l_3 completely covers l_2, and we should remove l_2. The intersection of l_3

with l_1 is at $x = -\frac{1}{2}$. We now have l_1 tallest on $(-\infty, -\frac{1}{2}]$ and l_3 tallest on $[-\frac{1}{2}, \infty)$. Adding l_4, we see that it insersects l_3 at $x = -1$. This again completely covers l_3, so we remove l_3 and proceed to find the intersection with l_1. The intersection with l_1 occurs at $-\frac{2}{3}$, so l_1 is tallest on $(-\infty, -\frac{2}{3}]$ and l_4 is tallest on $[-\frac{2}{3}, \infty)$. Finally, we add l_5. It intersects l_4 at $x = 5$, so it does not completely cover l_4 and we can stop here. Thus l_1 is tallest on $(-\infty, -\frac{2}{3}]$, l_4 is tallest on $[-\frac{2}{3}, 5]$ and l_5 is tallest on $[5, \infty)$. l_2 and l_3 are never tallest.

This problem recurs often as a subproblem in minimizing or maximizing many values simultaneously or across a range. As an example, consider the following problem from this year's Bitwise contest: Given an nxn grid with 0s and 1s, compute the shortest distance from each gridpoint to a 1 in $O(N^2)$ time.

We can solve this with another common method, namely *reduce the complexity of the problem by considering a group of subproblems instead.* While this does not always work if the different parts are sufficiently interrelated, oftentimes we can greatly simplify problems with this idea. In this case, we consider each column separately. Then, in each row, compute the closest 1 in that row to the given column. Now, let y be an arbitrary row in column x, and let (x_i, i) be the closest 1 to column x in row i. What is the distance from (x, y) to (x_i, i)? To simplify things, we will consider the square of the distance instead. It is simply $(x_i - x)^2 + (i - y)^2$. Note that x, x_i, and i are all fixed, so that the only variable is y. So, as y goes from 0 to n, we want to keep track of which i at each point will minimize that expression. At first sight, this is difficult, as the expression is quadratic in y. However, *for each i, the quadratic term in y will be the same, namely* y^2. Therefore, the differences in the expressions is only linear in y, and we already know how to find the tallest line at any given point. We can reduce this from $O(NlogN)$ to $O(N)$ because our sort would simply sort by the slope, namely $2i$ – since i is simply how we are indexing rows, it is easy to create an array sorted by i, and therefore $2i$, so we can skip the sorting step.

This problem, by the way, was the second-hardest problem on the contest out of 10 problems total.

2 Triangulation

Let's start with a problem. You are the owner of an art gallery and want to place cameras such that the whole gallery will be covered by the cameras. Each camera has a sight range of 360 degrees and is only blocked by walls. How many cameras do you have to buy, and where do you place them?

Let's use a simple polygon to represent that art gallery. Clearly, if the polygon is convex, we only need one camera. If it is complex, we can *triangulate* the polygon, or divide the polygon into non-overlapping triangles whose vertices are all vertices of the polygon. We know that if we place a camera at each triangle, since the triangles are convex polygons themselves, we can cover the entire gallery. Thus we have an upper bound for the number of cameras. Triangulation is also crucial in computer graphics implementations.

Now the question remains: how can we triangulate the polygon? First, we must prove that all simple polygons are triangulatable. We can prove this using:

2.1 The Two-Ears Theorem

A polygon P has an **ear** at vertex V if the triangle formed by V and its two adjacent vertices shares two edges with the polygon and has its last edge completely in the interior of the polygon. The Two-Ears Theorem states that all polygons with more than 3 sides have at least two non-overlapping ears.

We will prove the Two-Ears Theorem using induction.

Base Case: Trivially, a quadrilateral always has two ears. This can be proven as part of the induction.

Induction Hypothesis: A polygon P with fewer than n sides has at least two ears.

Consider a polygon P with n sides, $n \geq 4$. Let V be a concave vertex of P (otherwise, there can't be an ear at V). Let $v1$ and $v2$ be the 2 neighboring vertices. There are two cases to analyze: either there is an ear at V or there isn't.

Case 1: There is an ear at V.
We can remove this ear and draw the line connecting $v1$ and $v2$, creating another polygon with $n-1$ vertices, P'. By our hypothesis, this polygon has at least two ears. Since these ears must be non-overlapping, at least one of the ears is not at $v1$ and $v2$. Since all ears of P' are ears of P, P thus has at least 2 ears.

Case 2: There is no ear at V.
If there's no ear at V, we know that there must be an edge from V to some other vertex (we'll call it Z). Let's assume we know where VZ is, and draw it in. We have partitioned P into two polygons, P_1 and P_2. We now have two subcases to consider:

Subcase 1: One of the subpolygons is a triangle.
Wlog, let's make P_1 the triangle. Thus P_1 is an ear for P. By our hypothesis, P_2 must have at least 2 nonoverlapping ears. Since they are nonoverlapping, at least one of them is at neither V nor Z. Clearly the ears in P_2 don't overlap with P_1, so P has at least two ears

Subcase 2: Neither of the subpolygons is a triangle.
Using our hypothesis on both P_1 and P_2, each has at least two nonoverlapping ears. Thus, each must have at least one ear that is not at Z and not at V, and since they don't overlap

each other, P has at least two ears, Q.E.D.

Congratulations, we have proved the Two-Ears Theorem. Since we can always find an ear in a simple polygon P, we can keep removing ears and storing the extra edges created from the ears. Using this method, we can triangulate the polygon. This also gives us the computational algorithm:

```
//as a prerequisite, convex hull the polygon
method VanGogh(Polygon P) //because we're cutting ears.  Ha ha ha!
{
    i=1;
    notfoundear=true;
    while(notfoundear)
    {
        if(p[i] is convex)
            if(triangle p[i-1],p[i],p[i+1] contains no concave vertex)
                notfoundear=false;
        if(earnotfound)
            i++;
    return p[i];
    }
}
```

We can then run this continuously, each time cutting off the found ear and connecting $p[i-1]$ and $p[i+1]$ until we end up with a triangle. Then by adding in the edges to P, we have our triangulated polygon. Note that this algorithm is $O(kn)$, where k is the number of concave vertices and n is the number of vertices in P.

2.2 Monotone Polygons

A faster way to triangulate polygons is by using monotone polygons. A **monotone polygon** is a polygon with a boundary that can be split into two parts, each part strictly incrementing or decrementing in either the x or y dimension. I won't go into a lot of detail over the proof of this but if you want to research the material, it is readily available online. Instead, I'm going give a quick overview of this method.

First, we want to split the polygon into monotone polygons. We can do this by first partitioning the polygon intersections split by vertical or horizontal (your choice) sweeping lines. Each sweep line is constructed on a vertex of the polygon. By randomly constructing diagonals between vertices of sweep lines, we create trapezoids. All we need to do to transform these trapezoids into monotone polygons is check whether the two vertices of the original polygon lie on the same side. We can then use a greedy algorithm that cuts off the convex corners of the polygon, triangulating the polygon. This entire thing is an $O(nlogn)$ algorithm.